BLE 权威教程

CC254xEK 开发套件用户手册

Ghostyu.com

2013-10-01

目录

前言

第 1 章讲解了 BLE 协议的基础知识，结合我们的 SmartRF 系列 BLE 开发板，这也是本文的硬件平台。第 2 章对 IAR 开发环境进行了讲解，突出我们实际使用中密切相关的功能，其余予以略过。

第 3 章对 CC254X 开发板硬件资源进行了讲解。

第 4 章对 Bluetooth-LE 低功耗蓝牙中的数据传输进行了讲解。

第 5 章对 BLE 协议栈中的 OSAL 进行了讲解。

第 6 章对 BLE 无线网络管理进行了讲解和阐述。

第 7 章对 BLE 无线网络中，常用的项目开发经验和技巧进行了阐述。

希望读者阅读完本书后，节后自己的项目需求，对相应的协议栈源代码精心修改，再实验，只有通过不断的实践学习，才能真正掌握 BLE 无线网络的开发。

第 1 章 BLE 简介

如今，物联网技术得到快速的发展，而物联网技术里的主角则是已发展成熟的 BLE 无线网络，…作为开发者的我们，使用 TI 的软件和硬件资源，即可轻松搭建自己的无线网络。

1.1 无线网络数据传输协议对比

我们比较熟悉的网络有 Zigee，WIFI、Bluetooth（传统蓝牙，新一代蓝牙也发生了巨大的变化），他们三者之间的关系可以从下图中看出来。

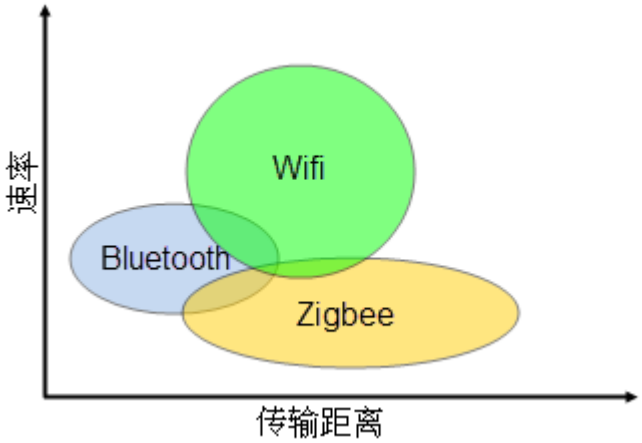


图 1-1 三种常见的网络对比

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利传输距离有各自的使用范围。

Zigbee、蓝牙以及 WIFI 标准都是工作在 2.4GHz 频段的无线通信标准。

下面将 BLE 与传统蓝牙、WIFI 标准进行简要的比较，帮助读者快速地了解。

• 传统蓝牙数据传输速率小于 3Mbps，典型数据传输距离为 2-10m，蓝牙技术的典型应用是在两部手机之间进行小量数据的传输。

• WIFI 最高数据传输速率可达 50Mbps，典型数据传输距离在 30-100m，WIFI 技术提供了一种 Intemet 的无线接入技术。

1.2 蓝牙 4.0

1.2.1 什么是蓝牙 4.0

图 1.1 蓝牙 4.0 logo 和 BLE logo

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。它集成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，低功耗蓝牙即 ble 是蓝牙 4.0 的核心规范，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，可应用与对成本和功耗都有严格要求的无线方案，而且随之智能机的发展将有着更加广泛的领域。

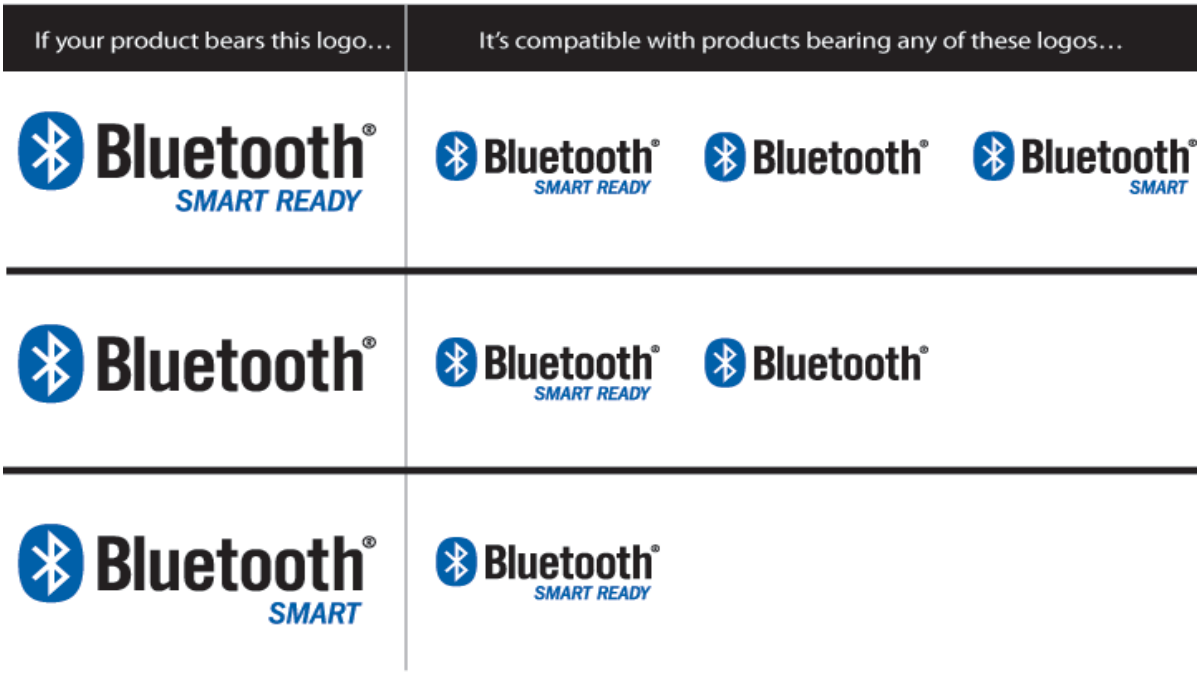
当前，支持 ble 的智能设备除了 iPhone（iOS 系统）外，Android 也正式加入了 ble 的队伍，从 4.3 系统开始，Android 将提供官方的 API 接口，在不久的将来，BLE 将会出现在生活中的各个领域。

BLE 是一种标准，该标准定义了短距离、低数据传输速率无线通信所需要的一系列通信协议。基于 BLE 的无线网络所使用的工作频段为 868MHz、915MHz 和 2.4GHz，最大数据传输速率为 250kbps。

下面通过一个具体酌例子向读者展示一下 BLE 的具体应用。在病人监控系统中，病人的血压可以通过特定的传感器检测，因此，可以将血压传感器和 BLE 设备相连，BLE 设备定期检测病人的血压，将血压数据以无线的方式发送到服务器，服务器可以将数据传输到医生的电脑上，医生就可以根据病人的血压数据进行恰当的诊断。

1.2.2 蓝牙 4.0 与传统蓝牙之间的关系

Smart Ready 和 Smart 以及传统蓝牙之间是什么关系呢，请看下图：



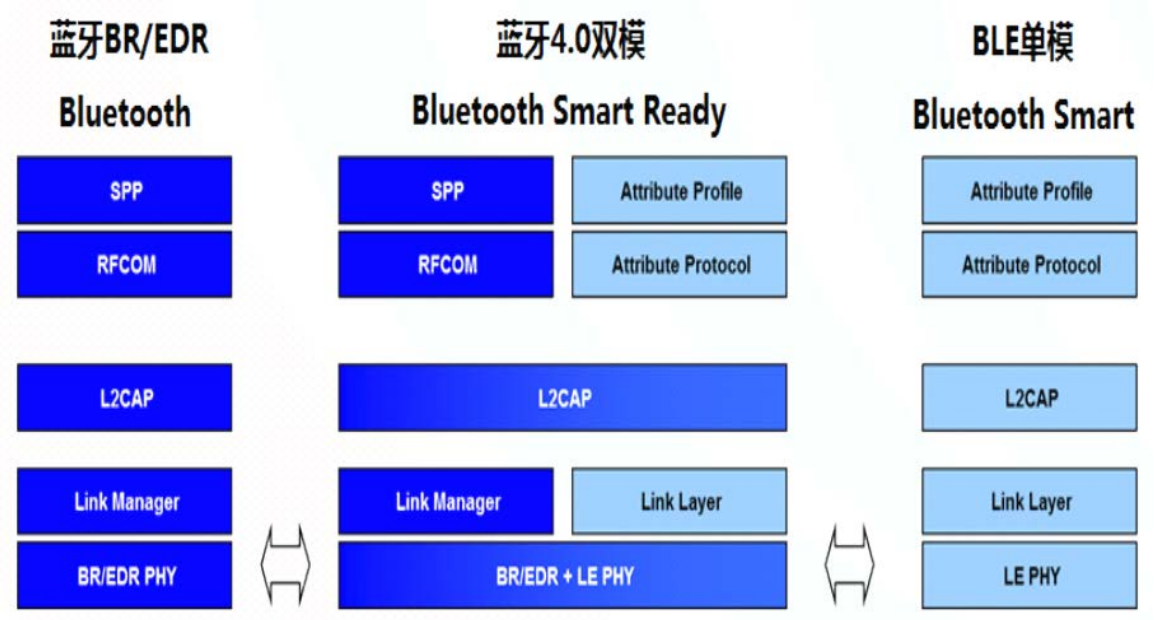
☆Smart Ready 可以和 Smart Ready、传统蓝牙，以及 Smart 之间相互连接和通信。

☆传统蓝牙可以和 Smart Ready、传统蓝牙之间连接和通信

☆Smart 可以和 Smart、Smart Ready 之间连接和通信

很多客户都比较关注 CC2540 是否向下兼容，看了上图就应该明白，答案是否定的，CC2540 是 BLE 单模芯片，属于 Smart，所以只能和 Smart Ready 或者 Smart 之间连接和通信，是不兼容传统蓝牙的。

低功耗蓝牙与其他蓝牙兼容性示意



双模设备

1：BasicRate（BR），2：Bluetooth Low Energy（BLE），同时支持 BR 和 BLE 的设备为 dual-mode（双模）设备，也就是这里讨论的 Smart Ready。Smart Ready 是蓝牙 4.0 里的主体，一

般具有稳定电源供电的设备，如手机，PC 等采用的均是双模的蓝牙芯片。很多 android 手机都表明支持蓝牙 4.0，其实很大部分只支持 Smart Ready 里的 BR。而软件里不支持 LE。

目前 Android4.3 系统才开始全面支持 BLE。

iOS 设备对蓝牙 4.0 支持的最好，只要是 iPhone4S 和以后的设备均完美完全支持蓝牙 4.0

单模设备

那么 Smart 又是什么呢，Smart 是蓝牙 4.0 里的低功耗蓝牙的商标，也就是 Bluetooth Low Energy，缩写为 LE 或者 BLE，网上关于蓝牙 4.0 一节纽扣电池能够使用一年均是针对 BLE 而言。Smart Ready 功耗还是很大的，需要有稳定的电源供电，像手机、PC 等设备，而 Smart 由于功耗低，一般使用电池、或纽扣电池供电。Ti 的 CC2540 便是 BLE 设备。

Smart 的最主要特点是低功耗和低速率

1.3 Bluetooth Low Energy

1.3.1 BLE 简介

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。初始化安全特征和设备配置。

☆协议栈中的 GATT 层用于已连接的蓝牙设备之间的数据通信。

1.3.2 BLE 特点

总体而言，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 在指定蓝牙 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，链路管理协议中采用可靠性措施，包括：差错检测和矫正，进行数据编解码，数据加噪等，极大的提供蓝牙无线数据传输的可靠性，另外，使用自适应调频技术，最大程度的减少和其他 2.4G 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，一般智能机上采用双模，外设一般采用 BLE 单模，例如采用 CC254x 作为 BLE 从机。

BLE 技术可以应用于 8-bit MCU，目前 TI 公司推出的兼容 BluetoothLE 协议的 SoC 芯片 CC254X 每片价格在 9 元左右，外接几个阻容器件构成的滤波电路和 PCB 天线即可实现网络节点的构建。

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 90%。

☆传统蓝牙设备的待机耗电量一直是其缺陷之一，这与传统蓝牙技术采用 16——32 个频道进行广播有很大关系，而低功耗蓝牙仅适用 3 个广播通道，且每次广播时射频的开启时间也有传统的 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更多的能源。

(3)快速启动、瞬间连接

此前蓝牙版本的启动速度非常缓慢，2.1 版本的蓝牙启动连接需要 6s 时间，而蓝牙 4.0 版本仅需要 3ms 即可完成，几乎是瞬间连接。

(4)传输距离极大提供

传统蓝牙传输距离一般 2-10m，而蓝牙 4.0 的有效传输距离可以达到 60~100m，传输距离提升了 10 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认证，对于初学阶段，安全性问题可以暂时不考虑。

例如，一般情况下，市面上每节 5 号电池的电量为 1500mA．h，对于两节 5 号电池供电的终端节点而言，总电量为 3000mA．h，即电池以 ImA 电流放电，可以连续放电 3000h（理论值），如果放电电流为 lOOmA，则可以连续放电 30h。

• 终端节点在数据发送期间需要的瞬时电流是 29mA；

• 数据接收期间所需要的瞬时电流为 24mA。

再加上各种传感器所需的工作电流，为了讨论问题方便，假设各种传感器所需的工作电流为 30mA（这个工作电流已经很大了），那么数据发送期间所需要的总电 流为 59mA，数据接收期间所需要的总电流为 54mA，为了讨论问题方便，总电流取 60mA，表面上 2 节 5 号电池可以供终端节点连续工作 50h。 但是，对应实际系统，终端节点对数据的采集一般是定时采集，例如采集 50s 数据，由于温度变化减慢，所以可以定时采集，在此假设终端节点每小时工作 50s，其他时间都在休眠（其他时间都在休眠，休眠时工作电流在微安级'所以可以忽略不计）。

那么实际上情况是：系统采用 2 节 5 号电池供电，终端节点工作电流为 60mA，每小时工作 50s（其他时间都在休眠，休眠时工作电流在微安级，所以可以忽略不计），可以计算出 2 节 5 号电池可以供终端节点工作时间为：3600h=150 天，即大约半年时间，这也就是很多介绍 BLE 技术的书籍中提到的“对于 BLE 终端节点' 使用 2 节 5 号电池供电，可以工作半年的时间”的理论依据。请读者注意，上述分析是针对的终端节点，对于路由节点和协调器而言，要一直供电来确保数据的正确路由，所以一般不谈低功耗问题。

(3)高安全性

为了保证数据传输的安全性，可以使用 AES-128 加密技术，但是对于初学阶段，安全性问题可以不予考虑。

(4)低数据速率

无线控制系统对数据传输的可靠性和安全性、系统功耗和成本等方面有着特殊的要求，因此，目前的无线网络协议没有很好地解决这些特殊的要求。

1.4 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？

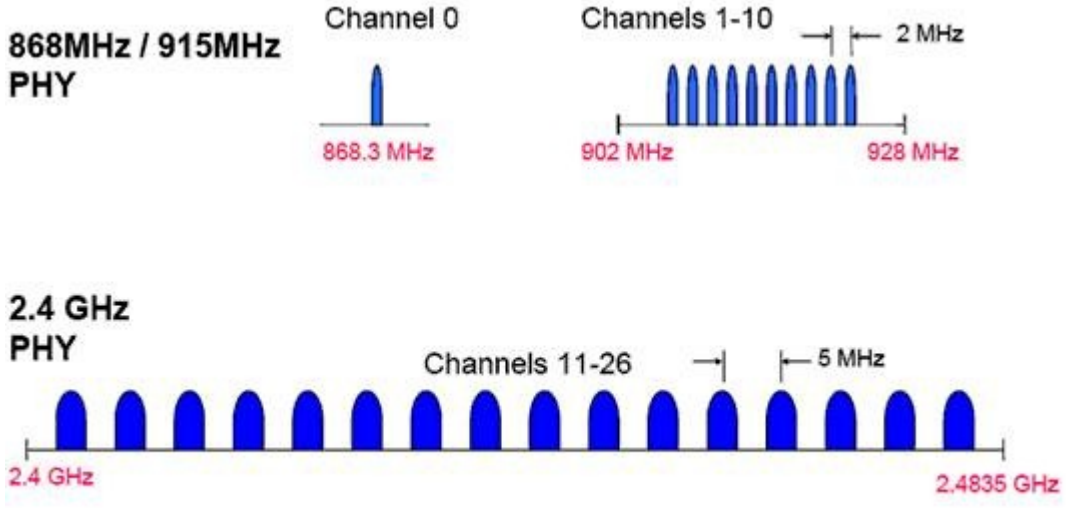
一般情况，不同的电波具有不用的频谱，无线通信系统的频谱有几十兆赫兹到几千兆赫兹，包括了收音机、手机、卫星电视等使用的波段，这些电波都使用空气作为传输介质来传播，为了防止不同的应用之间相互干扰，就需要对无线通信系统的通信信道进行必要的管理。各个国家都有自己的无线电管理结构，如美国的联邦通信委员会(FCC)、欧洲的典型标准委员会(ETSI)，我国的无线电管理机构称为中国无线电管理委员会，其主要职责是负责无线电频率的划分、分配与指配、卫星轨道位置协调和管理、无线电监测、检测、干扰查处，协调处理电磁干扰事宜和维护空中电波秩序等。

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-I 业、科学和医疗）频带。

除了 ISM 频带外，在我国，低于 135kHz，在北美、日本等地，低于 400kHz 的频带也是免费频段。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》。

BLE 工作在 ISM 频带，定义了两个频段，2.4GHz 频段和 896/915MHz 频带。在 IEEE 802.15.4 中共规定了 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 868MHz 频段，有 1 个信道，信道通信速率为 20kbpS。



BLE 工作在 2.4GHz 频段，仅适用 3 个广播通道，适用所有蓝牙规范版本通用的自适应调频技术。

自适应调频技术是建立在自动信道质量分析基础上的一种频率自使用和功率自适应控制相结合的技术，他能使调频通信过程中自动避开被干扰的调频频点并以最小的发射功率、最低的被截获概率，达到在无干扰的调频信道上长时间保持优质通信的目的。

1.5 BLE 无线网络拓扑结构

BLE 网络可以点对点或者点对多点，一个 ble 主机可以连接多个 ble 从机，组成星型网络，另外还有一种有广播设备和多个扫描设备组成的广播组结构，不同的网络拓扑对应不同的应用领域。

1.6 BLE 技术的应用领域

一直以来，蓝牙技术在配件方面的应用都更受关注，但随着移动时代的迅猛发展，BLE 将会有更大的用武之地。事实上，BLE 的低功耗技术，在设计之初便主打医疗与健康监控等特殊市场 ，而总的来说，蓝牙 4.0 的发展方向将是运动管理、医疗健康照护、智能仪表、智能家居以及各种物联网相关应用。

在医疗健康领域，过去不少健康类的应用都是基于蓝牙 2.1 协议去做的，但因受限于耗电问题而未能掀动太大波澜。BLE 化解这一难题后，市场被强力激活。如由英特尔发起，并由许多不同医疗技术与保健机构成立的 Continua 健康联盟，便已决议将 BLE 纳入日后的标准传输技术中。现在市场上已有许多采用蓝牙 2.1 规格的医疗产品，如血压计、血糖仪等，未来，通过 Continua 健康联盟正式认证的蓝牙 4.0 规格的医疗类产品肯定会越来越多。健康应用方面，BLE 也有广阔的市场空间，其可以与健身设备进行无缝结合，人们在使用健身器材时，就能通过相关设备如计步器、脉搏机等来传送并记录运动情况进入移动设备，保存个人的健康信息。

BLE 与安卓的结合更将对当下如火如荼的“物联网”起到推波助澜的作用。目前市场上的所有智能设备都是物联网生态发展的推动力量，但 BLE 能够起到打通物联网的和传感器设备之间的“关节”的节点作用，这将从关键意义上推动物联网的真正发展。由于蓝牙技术一向关注上层应用，有统一标准，因此各种各样的底层硬件虽出自不同制造厂家，却可以互联互通，能够形成完善的生态环境，为自身及物联网产品市场都创造了良好环境。

有分析认为，当 BLE 把每个人的安卓或者其他移动设备变为一个传感器标签时，它所能做的将不仅仅是通过应用软件去找东西，而是将拥有巨大的可拓展性，如它可以通过 App 和传感器来构建一个 P2P 的网络以模拟 GPS 的功能等。总之，当 BLE 传感器无处不在时，定然蕴藏着巨大商机。

1.7 CC254xEK 开发套件硬件资源概述

进行 BLE 无线网络的开发，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 BLUETOOTH-LE 协议的 SoC-CC254X，同时也推出了相应的开发套件；但是价格较高。不适合国内的学习环境，因此我们依照 TI 官方开发板，在最大程度兼容 TI 官方的基础上，2013 年初，我们开发了 CC254xDK（SmartRF 系列）开发套件。正在帮

助着众多的公司和 BLE 爱好者实现他们的低功耗蓝牙产品，在技术支持过程中，我们了解到了更多的产品需求，以及许多珍贵的意见反馈，因此，我们重新设计并推出了第二代 BLE 开发套件：CC254xEK（New SmartRF 系列）开发套件。

两代套件对比如下：

	第二代CC254x**EK**（New SmartRF系列）	第一代CC254x**DK**（SmartRF系列）
核心板	CC254xEM**v2**，**25*17**mm，**PCB天线/外接SMA天线**	CC254xEM，35*20mm，PCB天线
串口	**USB形式**（使用CH340G转USB）	RS232形式（使用MAX3232）
扩展	引出**全部**（包括I2C/USB）开发更方便	引出UART、SPI、常用GPIO
供电	**仿真器、外接锂电池或者USB串口供电**	仿真器或电池供电

第一代 CC254xDK 套件介绍

SmartRF 开发板。

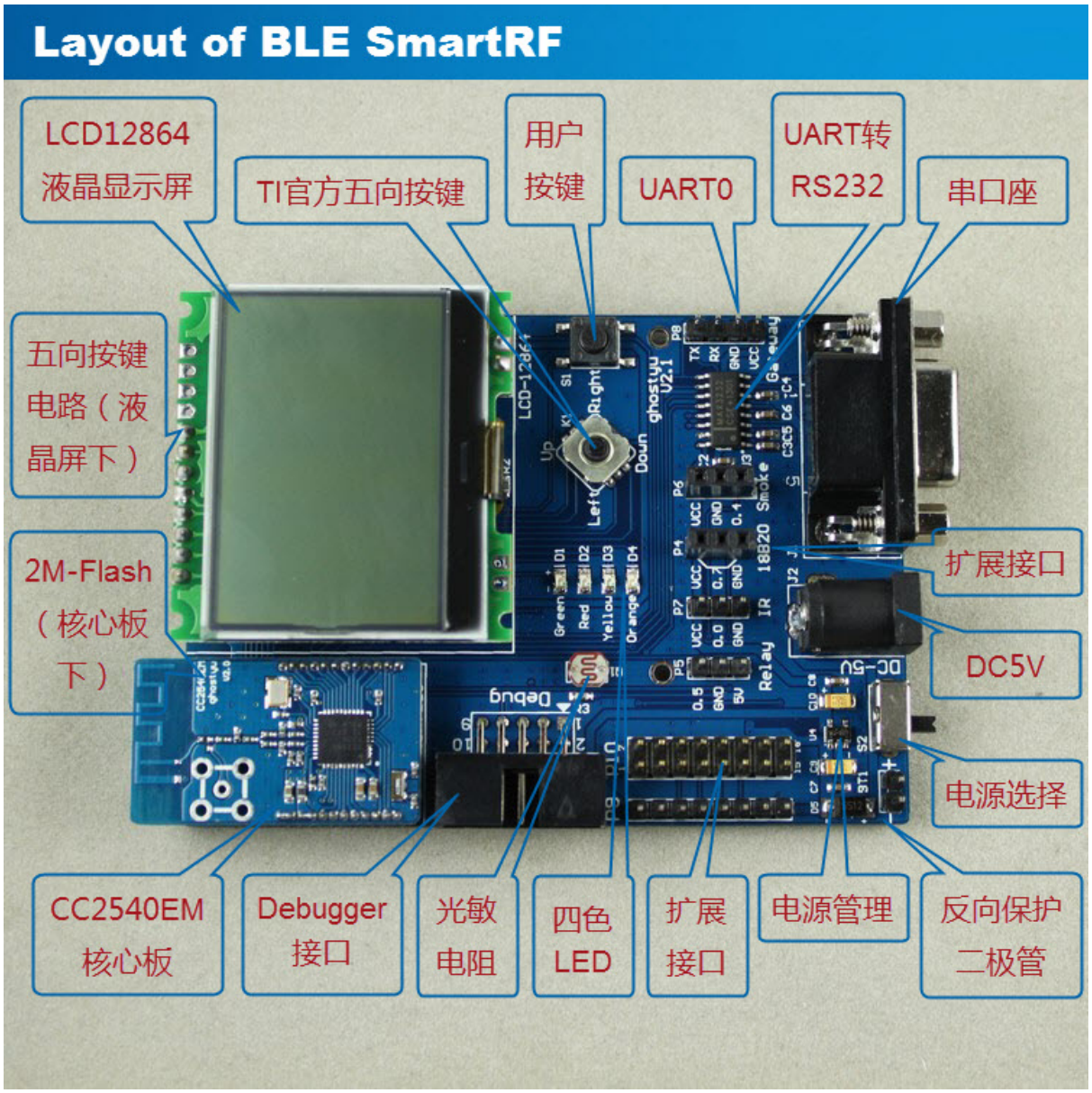


图 SmartRF 开发板

SmartRF-BB 开发板

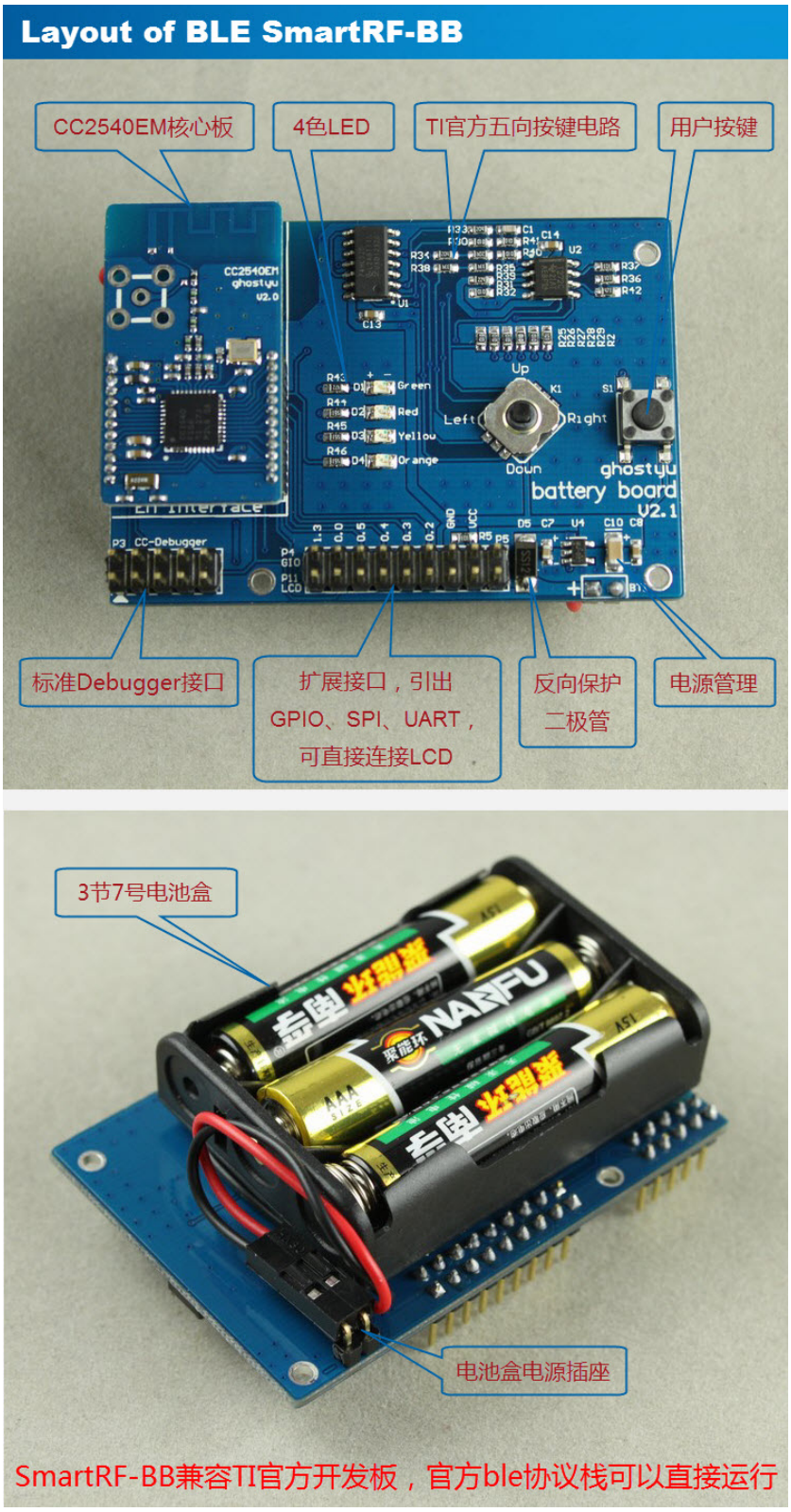


图 SmartRF-BB 板（电池板）

SmartRF-BB 开发板特点：

- 完全兼容 TI 官方开发板，BLE 可以直接运行。
- 背面为 3 节 7 号电池盒，方便节点开发、测试。
- 可以直插 LCD。

• 外接 SMA 天线，通信距离高达 270 米（无功放）。

CC2540USBdongle

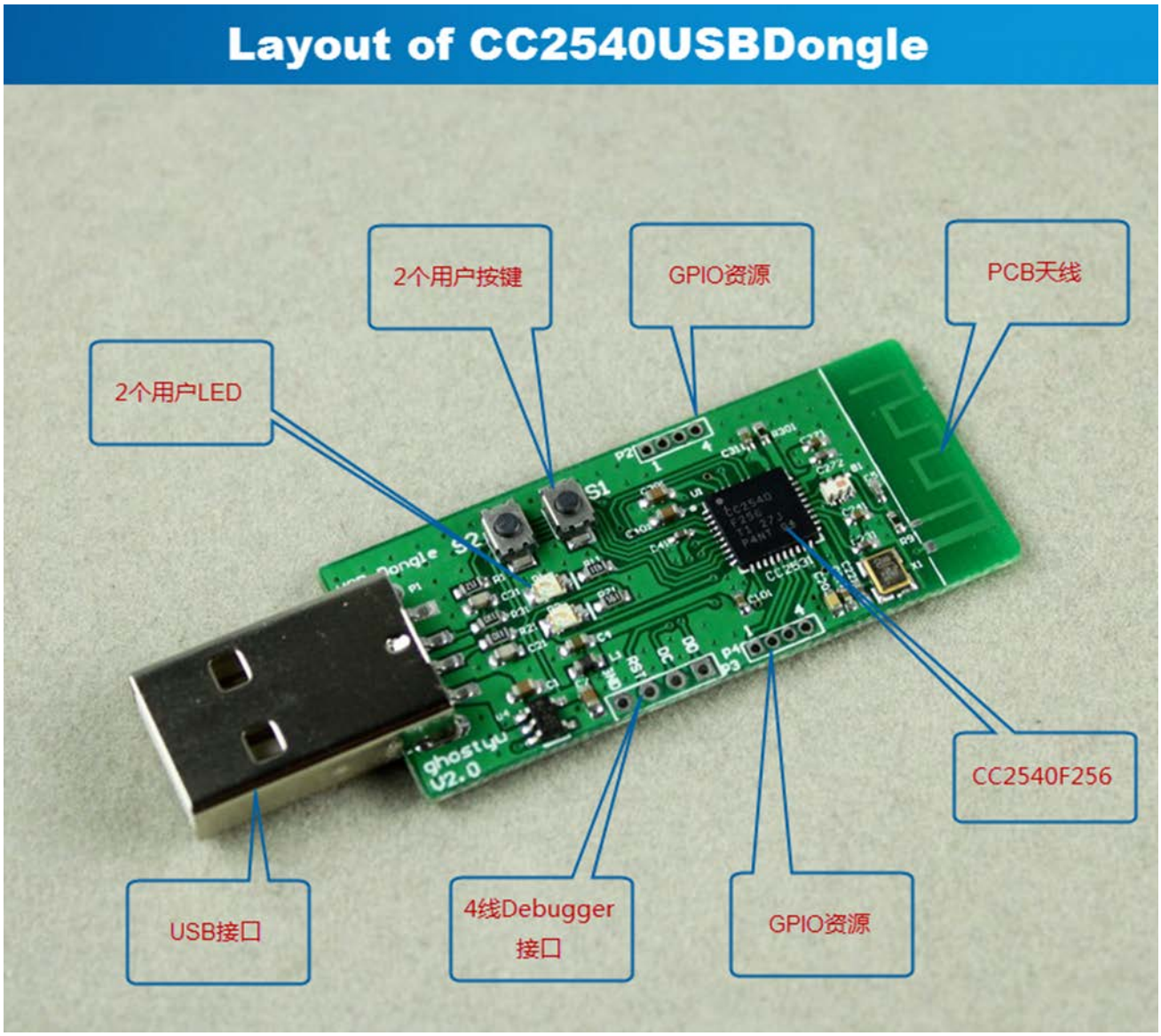
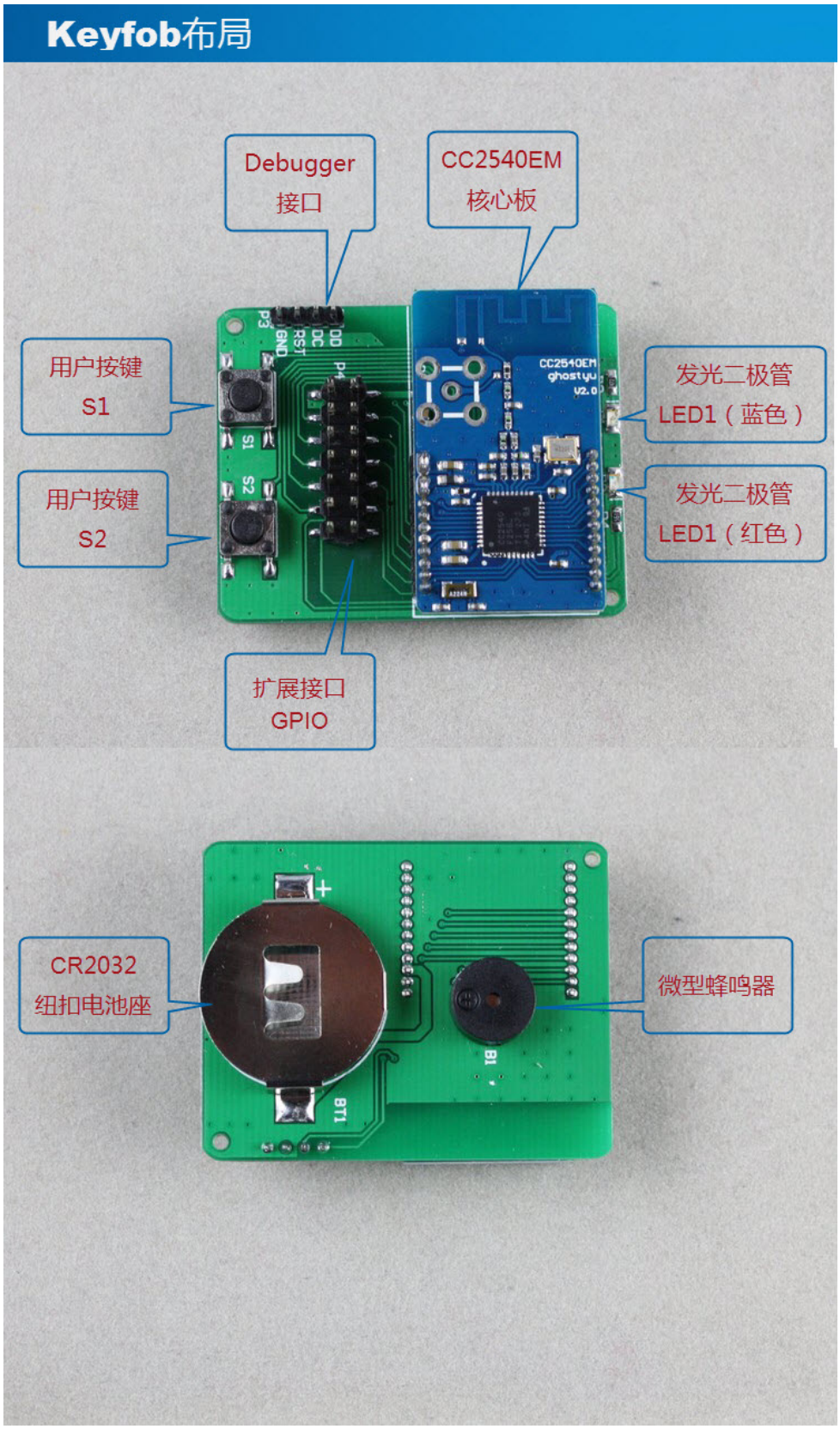


图 CC2531USBdongle

配合 TI PacketSniffer 软件实现 BLE 的无线抓包，另外可以配合 PC 端的 BTOOL 软件实现 PC 端的 BTool 主机。

关于开发板的详细使用手册和注意事项，请参见硬件手册。

Keyfob 开发板



Keyfob 开发板，可做防丢器，纽扣电池供电，方便开发测试。

1.8 本章小结

本章主要讲述了 BLE 协议的基础知识，此外还有我们的开发套件的简单介绍，使读者对 BLE 有一个整体的概念。

第 2 章 IAR 集成开发环境及程序下载流程

由于 BLUETOOTH-LE 协议的发布，以及相关公司推出的协议栈逐渐完善，市场上 现了各种各样的 BLE 技术解决方案，但是对于初学 BLE 的用户来说，如何准确地选择一款适合自己的开发平台至关重要。下面给出我的建议。

• 尽可能选用与 TI 官方接近的开发板

我们的开发板协议栈部分完全兼容 TI 官方，协议栈可以在我们的开发板上直接运行，无需做任何移植工作，另外，由于 BLE 会经常更新，这样就非常有好处，大家可以随意升级，而不用担心硬件环境发生变化。

• 带有丰富的开发资料以及实践项目

这里说的丰富的开发资料是指与开发板配套的资料，并非网络上搜集的 BLE 资料，对于初学者来说，一套能够直接上手的开发板和资料非常重要，不然会大家初学者的积极性。

本章主要讲述 IAR 开发环境进行 CC254X SoC 的开发，如果用户已经熟悉 IAR 开发环境，完全可以跳过本章，直接进行后面章节的学习。

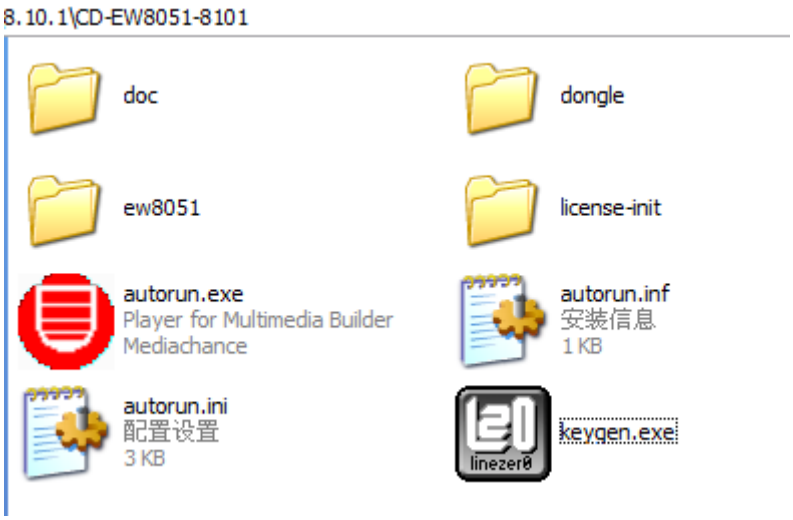
2.1 IAR 集成开发环境简介

IAR Embedded Workbench（又称 EM）的 C 交叉编译器是一款完整、稳定且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 位的 MCU

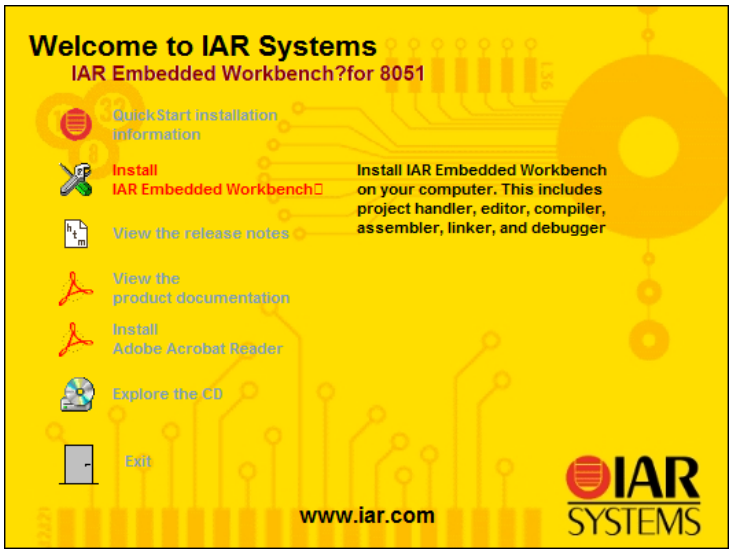
• 完全兼容标准 C 语言
• 内建相应芯片的程序苏荷和内部优化器
• 高效浮点支持
• 内存模式选择

2.1.1 安装 IAR8.10

程序安装包位于\Software\IAR\8.10.4 目录下，如下图



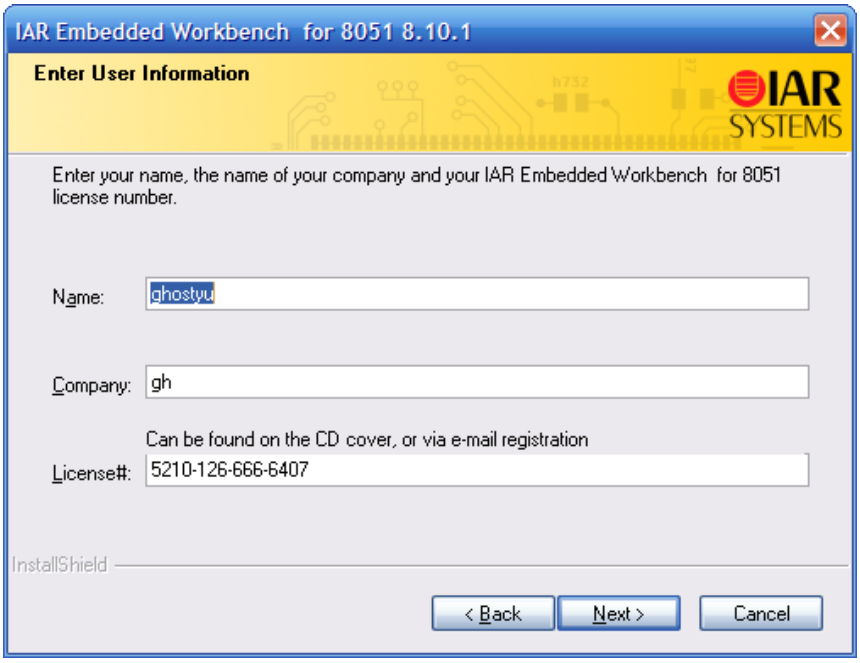
双击运行 autorun.exe，然后再跳出的画面中选择第二项，Install IAR Embedded Workbench



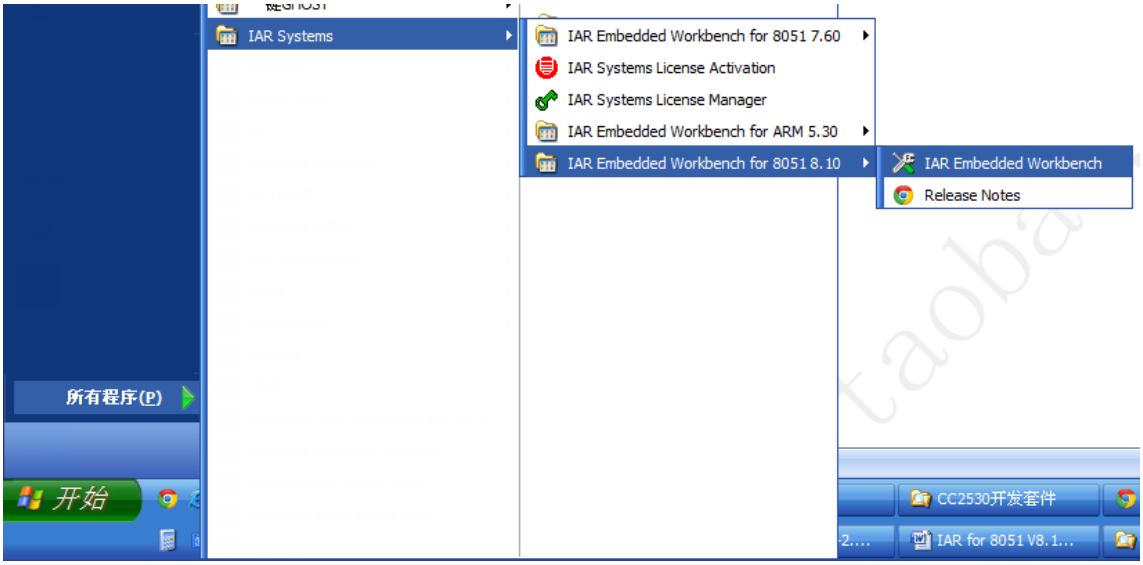
根据提示一路 next，到 Enter User Information 这一项，提示输入 license。
这里使用 keygen 生成 iar 的 license，如果大家有能力使用正版，最好使用正版软件。
运行资料里的 keygen 生成 license。位于如下图位置。

运行后选择第一项 for MCS-51 v8.10

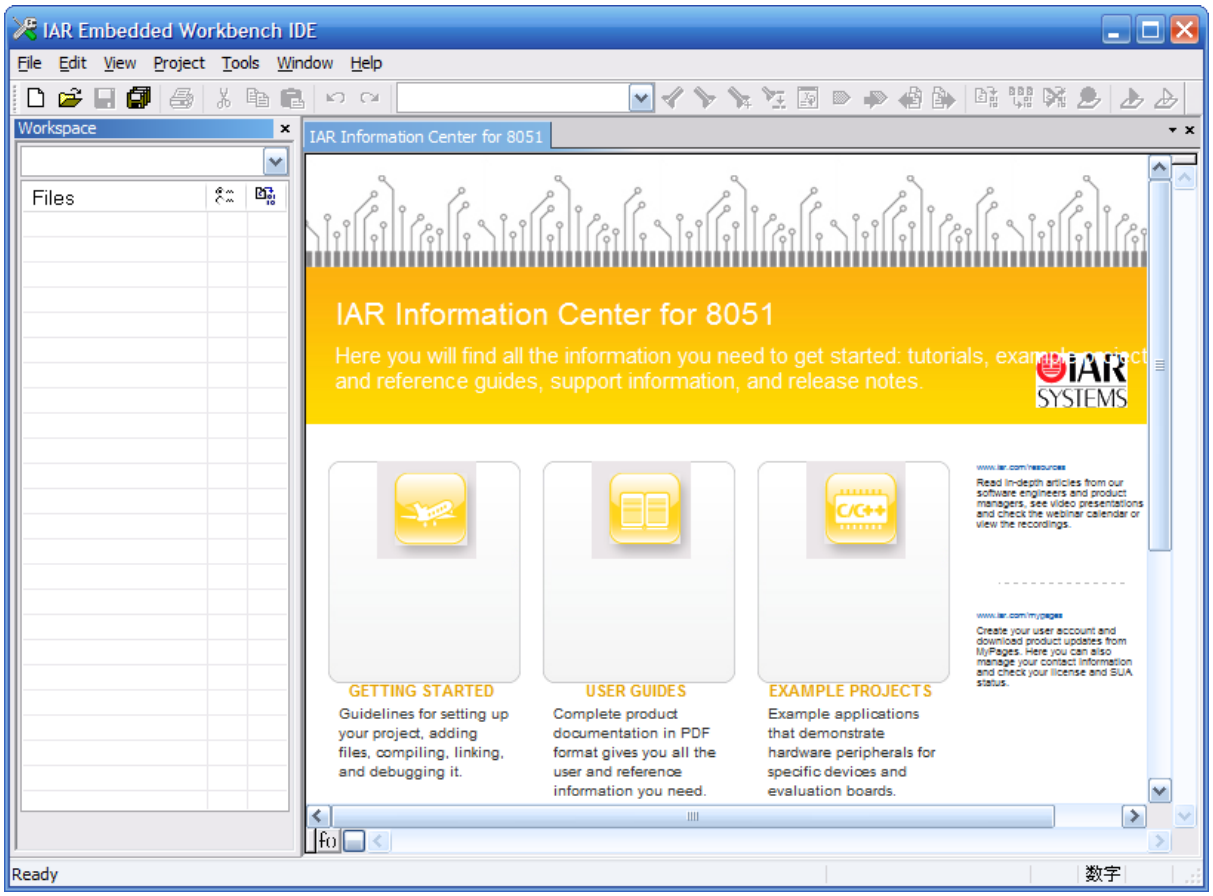
复制 license number 到 iar 安装向导中，next 后再复制 license key



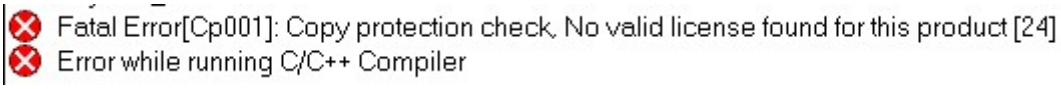
最后一路 next，安装结束后在开始菜单中找到 IAR 软件，默认安装的位置如下图：



运行的 IAR 软件如下图：



如果再 iar 的使用过程中出现如下错误



这是由于 iar 的 license 未能安装成功，请严格按照软件目录下的几个文档，多数有由于未能使用管理员运行注册机导致。

使用 win7 的用户请注意，上述的方法可能无法破解，需要在使用第二种命令行的破解方式。

在 Software\IAR\8.10.4\命令行注册方法文件夹下，有命令行方式的破解教程，务必按照要求一条一条的做，会成功破解，如果还出现上述 license 错误，基本上可以确定是因为没有按照要求做。

2.2 工程的编辑与修改

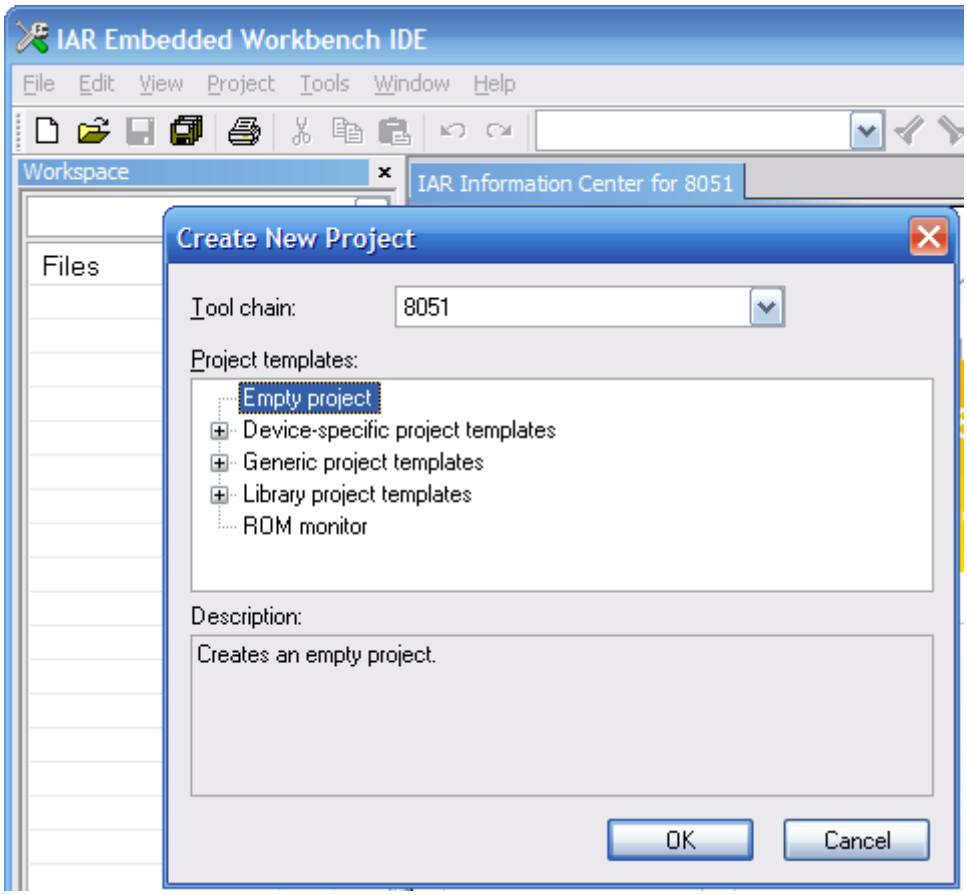
IAR 集成开发环境中，对用工程的编辑操作主要涉及以下几个方面的内容：

- 如何建立、保存一个工程
- 如何向工程中添加源文件
- 如果编译文件
- 工程配置在哪里

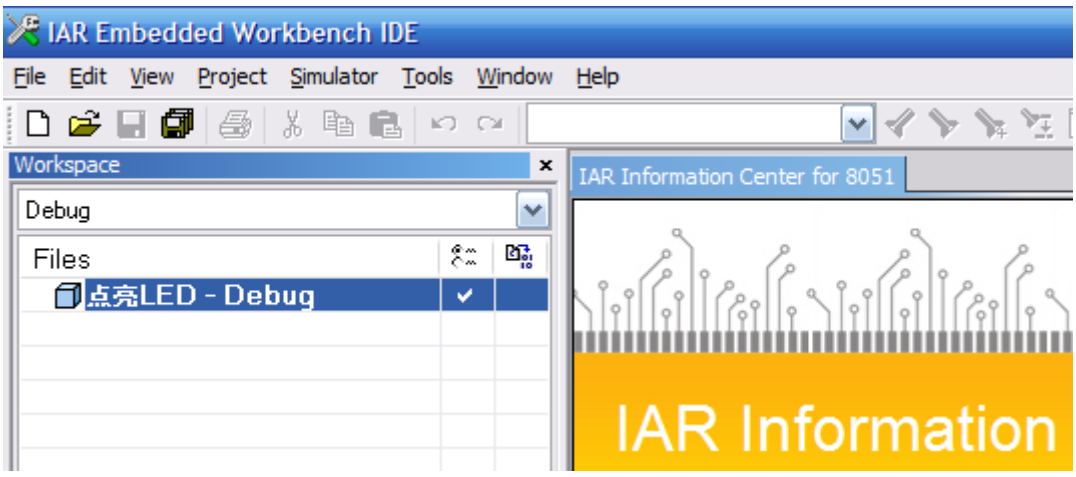
下面进行详细的讲解

2.2.1 建立一个新工程

打开 iar，点击菜单栏的 Project，在弹出的下拉惨淡中选择 Create New　Project，如图 2-2 所示

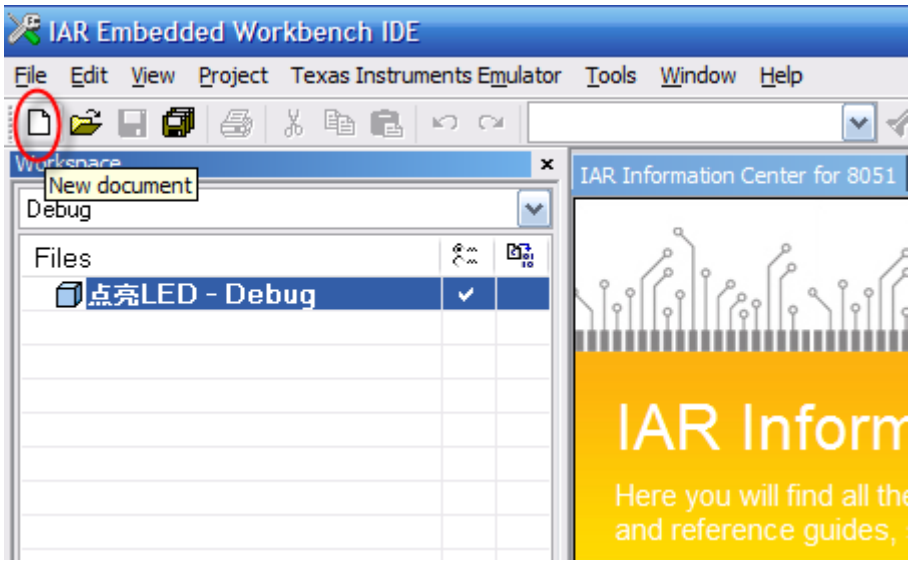


选择“Empty project”，单击 OK，然后会询问保存 project，选择一个合适的目录，我这里保存的目录在 CC254X 基础测试程序\1_点亮 LED 目录下，然后填入合适的工程名，然后单击 OK，如下图：

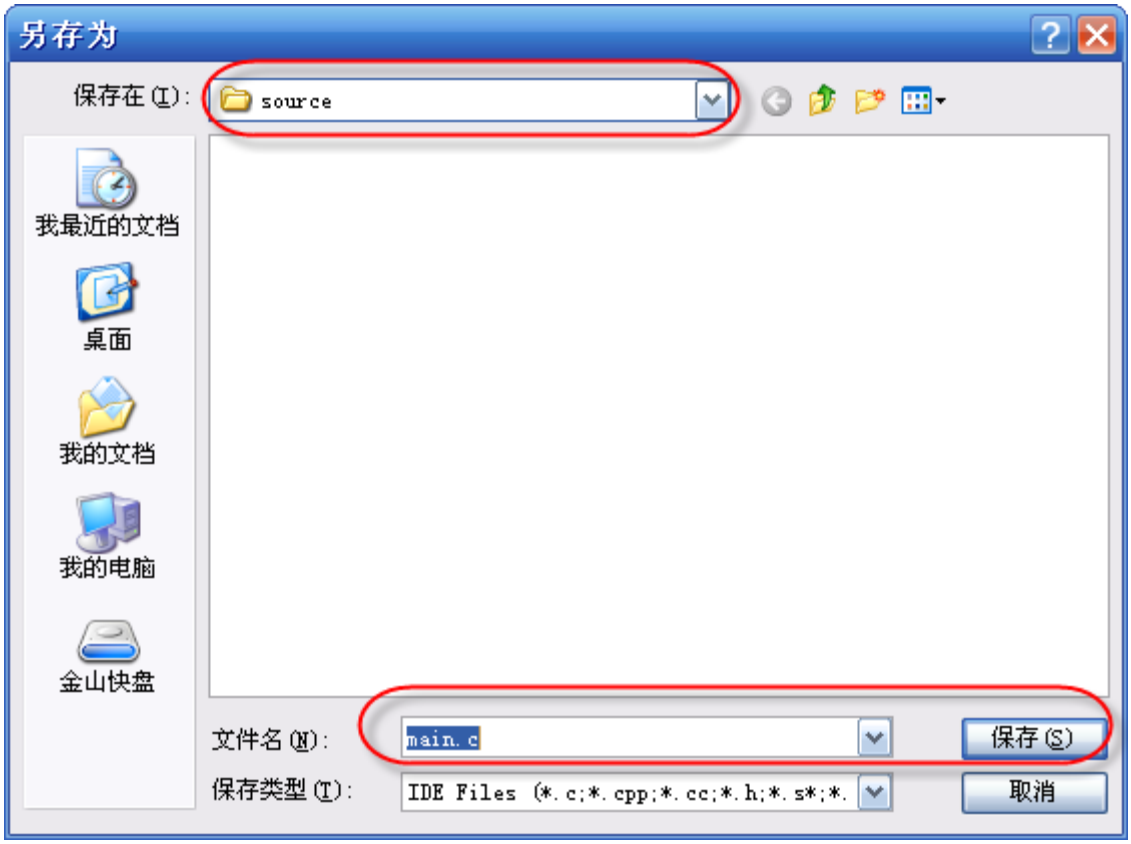


2.2.2 建立一个源文件

单击 New document 按钮，新建一个文本文件。

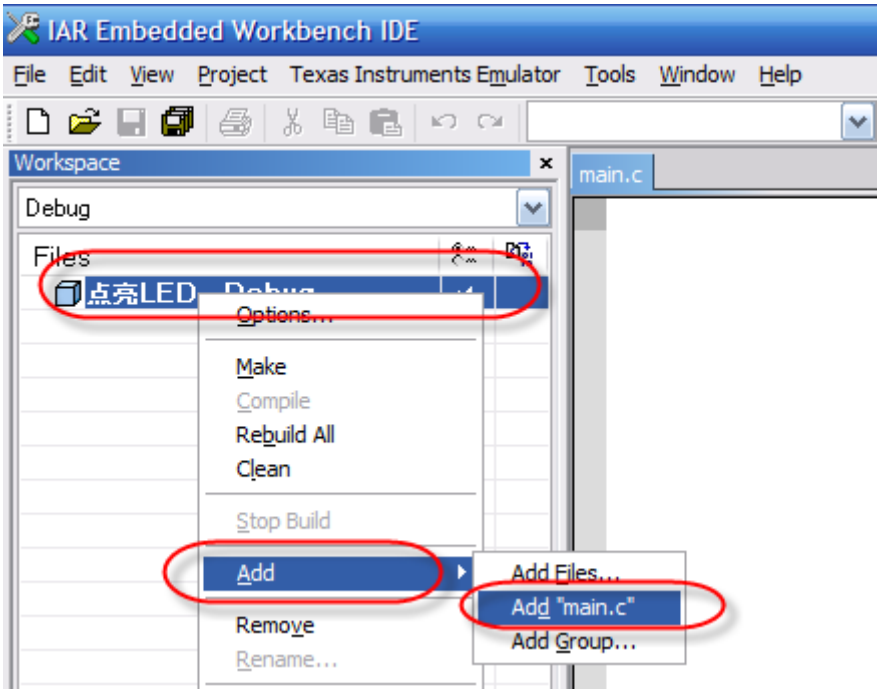


新建了文件之后单价保存按钮，保存为文件名为：main.c 到 source 目录下（source 是在 IAR 工程目录内新建的用来专门保存源码的目录）



2.2.3 添加源文件到工程

右击工程名，选择 add->Add main.c，注意，也可以使用 add files，手动选择 main.c

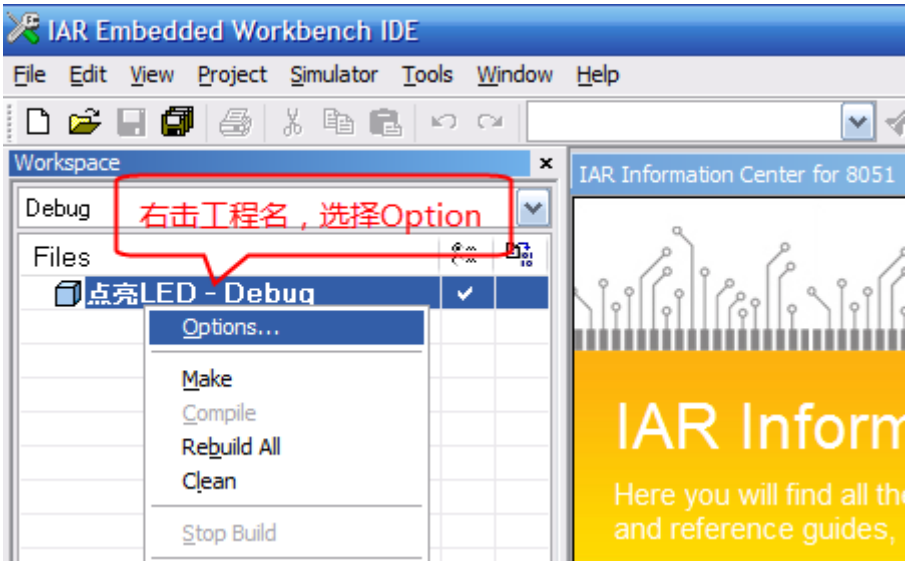


向 main.c 中输入一下代码，然后保存。

```
#include <ioCC2530.h>

//常用的宏定义
/*
位操作，作用是将第 n 位置 1
这在单片机中是非常常见的操作。
*/
#define BV(n) (1<<(n))
void delay(unsigned int time)
{
   int i,j;
   for(i=0;i<time,i++)
      for(j=0;j<1000;j++);
}
int main()
{
   //设置 P1.0 端口方向为输出
   P1DIR |= BV(0);

   //设置 P1.0 端口为 GPIO 功能
   P1SEL &= ~BV(0);


   //死循环
   while(1){
      P1_0=1;//点亮 led
      delay(1000);
      P1_0=0;//熄灭 led
      delay(1000);
   }
}
```

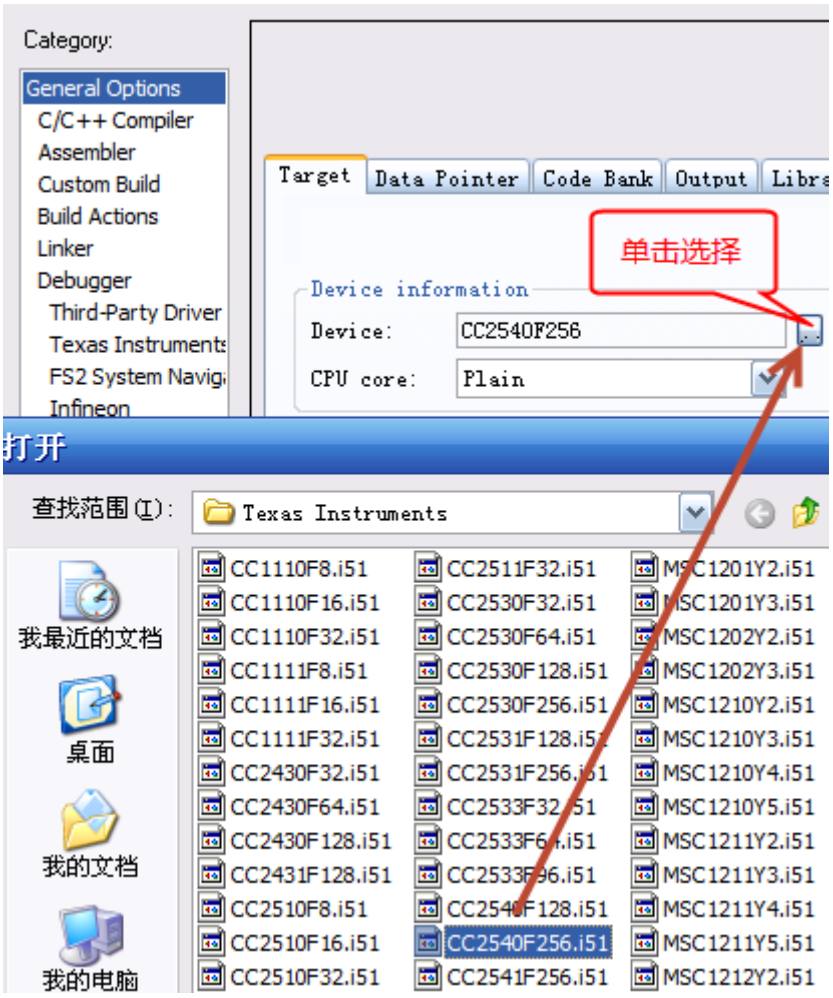
2.2.4 工程设置

在左边的 Workspace 中右击工程名，然后选择 Option，进入 Option 工程配置对话框，注意，Option 对话框我们将经常用到，先记住它是如何打开的，如下图：



配置目标芯片

在出现的对话框中，第一件事情就是选择该 project 所使用的 Device，左边选择 General Option，然后在右边的一些列的选项卡中选择 Target，如下图，设置 Device，我们这里使用的芯片是 CC254XF256，因此选择 CC254XF256.i51,（该文件的完整的默认路径为：C:\Program Files\IAR Systems\Embedded Workbench 6.0\8051\config\devices\Texas Instruments），截图中用的是 CC2540，使用 2541 的用户相应选择 CC2541F256.i51 即可。



设置 Code 和 Memory Model

在 code 类型中有 Near 和 Banked 两项可选择

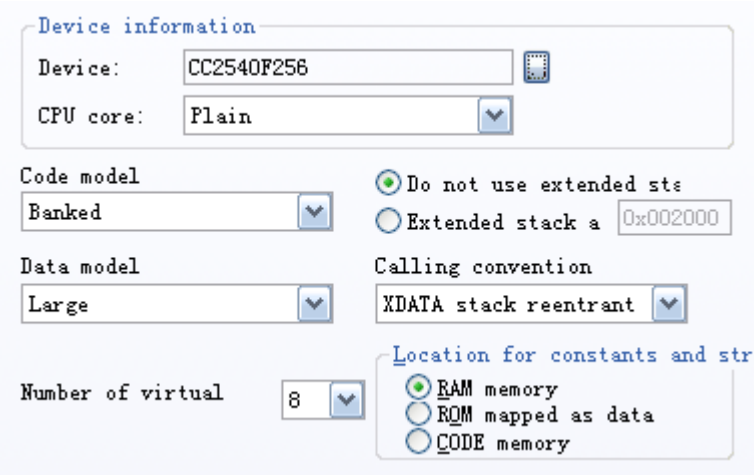
“Near”当不需要 Bank 支持支持是可以选择 Near，例如，你之需要访问 64K flash 空间的时候，不需要更多的 flash 空间，比如你使用的是 CC254XF32 或 CC254XF64，或者使用的 CC254XF256 但并不需要那么大的 flash 空间时，可以选择 Near。

“Banked”选择该项时标明你需要更多的空间能够仿真 CC253xF128 或者 CC253xF256 的整个 Flash 空间。

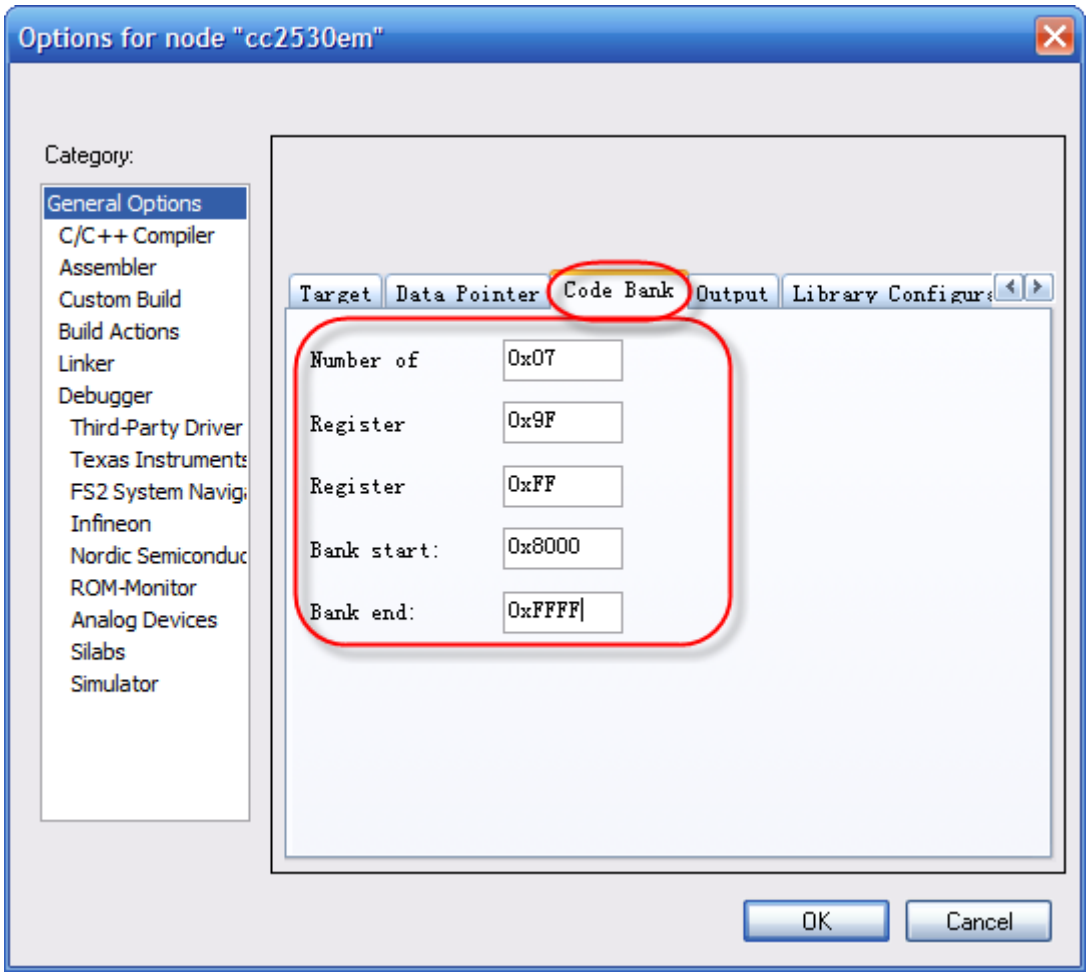
默认 Near code model 中的 data model 是 Small，默认的 Banked，data model 为 Large，data model 决定编译器或者连接器如何使用 8051 的内存来存储变量，选择 small data model，变量典型的存储在 DATA 内存空间，如果使用 Large data model，变量存储在 XDATA 空间。在 CC254X 用户手册和 IAR 8051 编译器参考手册中会详细描述变量内存空间。

在这里，重要的事情是，8051 使用不同的指令来访问 various memory spaces 访问 IDATA，一般情况下，比仿真 XDATA 要快，但通常 XDATA 的空间会比 IDATA 大。

在 BLE 协议栈中，使用 large memory model 来支持 CC254XF256，这样协议栈可以存储在 XDATA 区域，以上设置结束后，如下图所示。

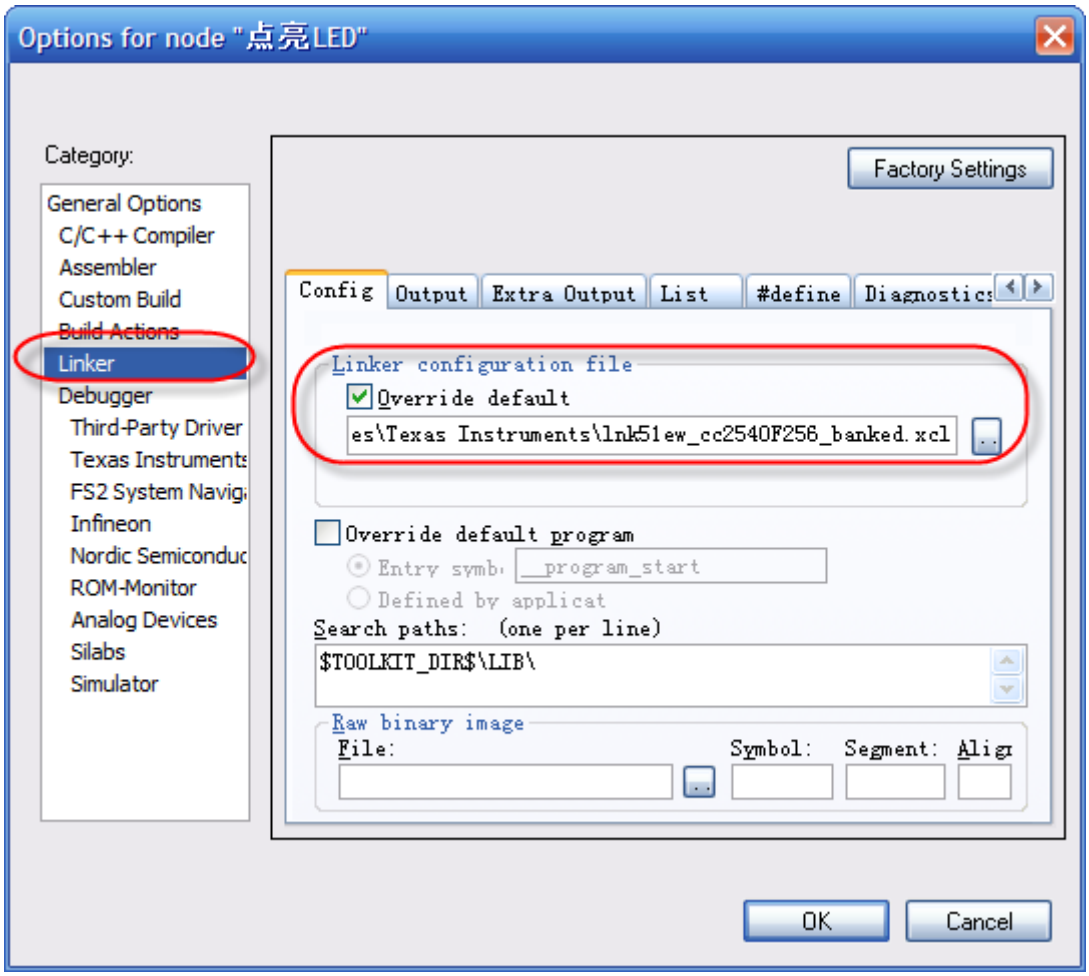


在 Banked code model 中，有一些额外的选项需要注意，选择 Code Bank tab，如下图，CC254X 使用 7 个 code banks，为了访问整个 256K 的 Flash 空间，Number of 必须设置为 0x07，Register 0x9F 是 CC254X 的 FMAP 寄存器，用来控制当前那个 code bank 映射到 8051 的地址空间，第三个 Register 未使用，最好设置 0xFF。



设置链接器

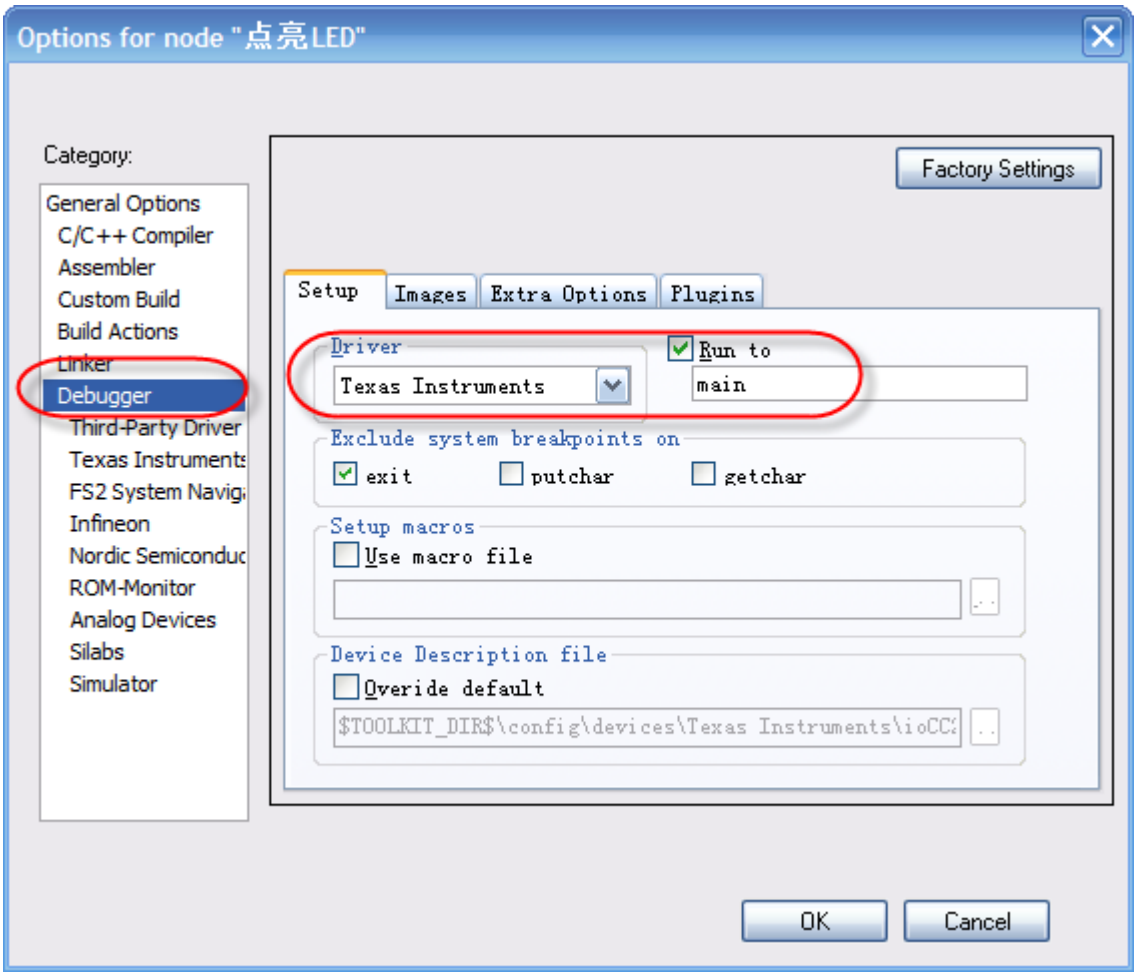
在左边的选项中选择 Linker，并在右边的选项卡中选择 Config 一页，在 Linker Command file 中复选 Override default，例如，我们选择 lnk51ew_CC254XF256_banked.xcl，banked 表示使用 banked code model。



默认路径为：$TOOLKIT_DIR$\config\devices\Texas Instruments\lnk51ew_CC254XF256_banked.xcl

设置仿真器调试

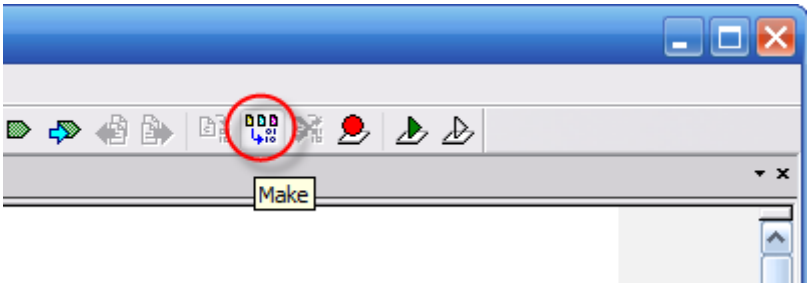
最后，在 Debugger 选项中，选择 Texas Instruments 为 Driver。

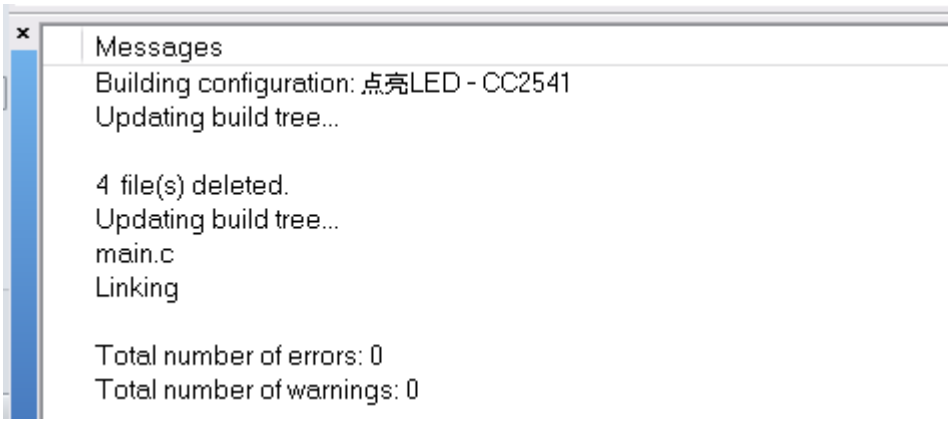


这样 IAR 工程就配置好了，单击 OK 确认。

2.2.5 源文件的编译

编译过程中如果出现错误，请根据错误提示修改不小心造成的语法问题。

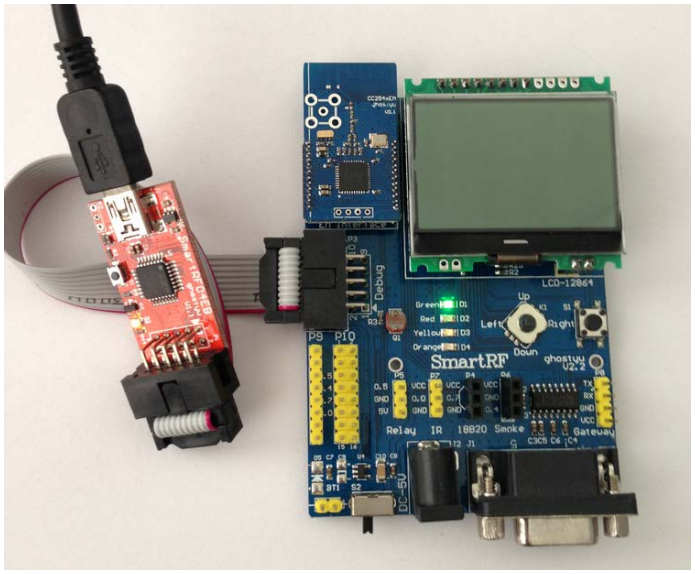


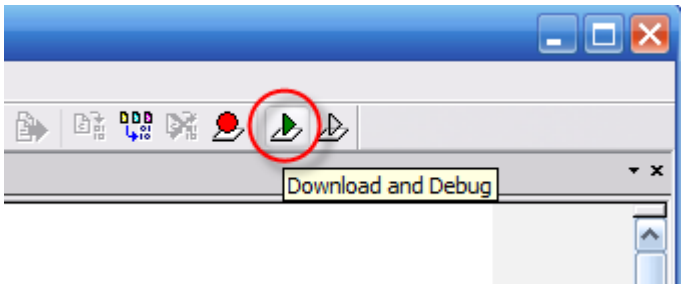


2.3 仿真调试与下载

在运行代码之前，首先将仿真器和开发板连接好，仿真器的使用 miniusb 线连接到电脑 PC 端，然后使用灰色排线连接仿真器和 SmartRF 开发板的 P3 debugger 座，如下图：

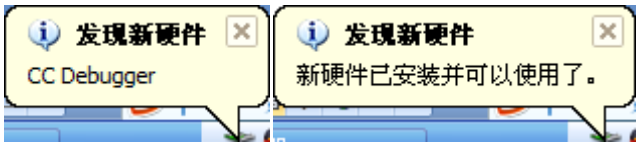
NOTE：需要注意的是，不管使用 CC-Debugger 仿真器还是 SmartRF04EB 仿真器，在调试下载程序之间，必须要仿真器的复位按钮，等到仿真器识别到开发板（CC-Debugger 仿真器灯由红变绿，或者 04EB 仿真器灯变亮）之后，在进行下载操作，否则会导致一些列不可预知的问题。





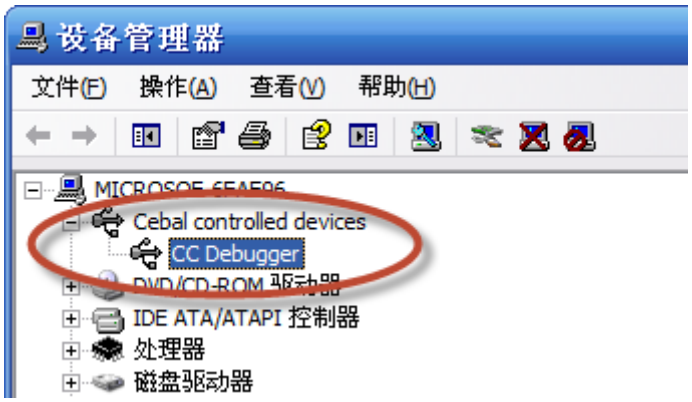
2.3.1 仿真调试器驱动的安装

连接 CC-Debugger 仿真器或者 SmartRF04eb 仿真器，PC 通知栏会告知发现新硬件，如果电脑中存在仿真器的驱动，则随机会跳出驱动安装成功的提示：

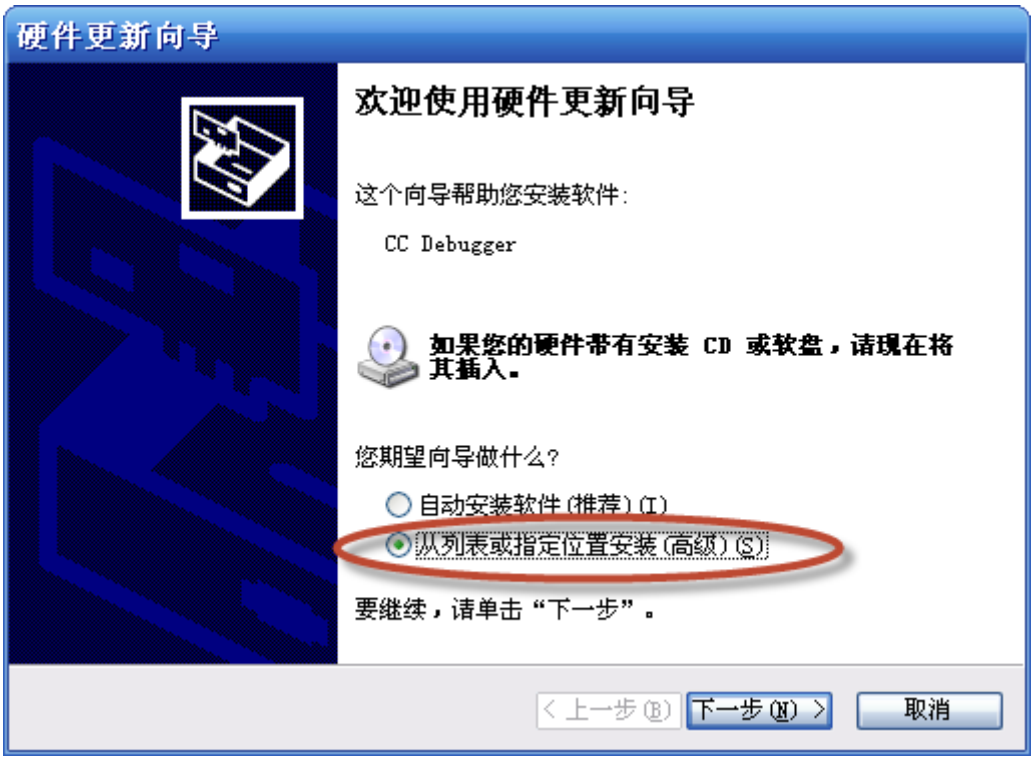


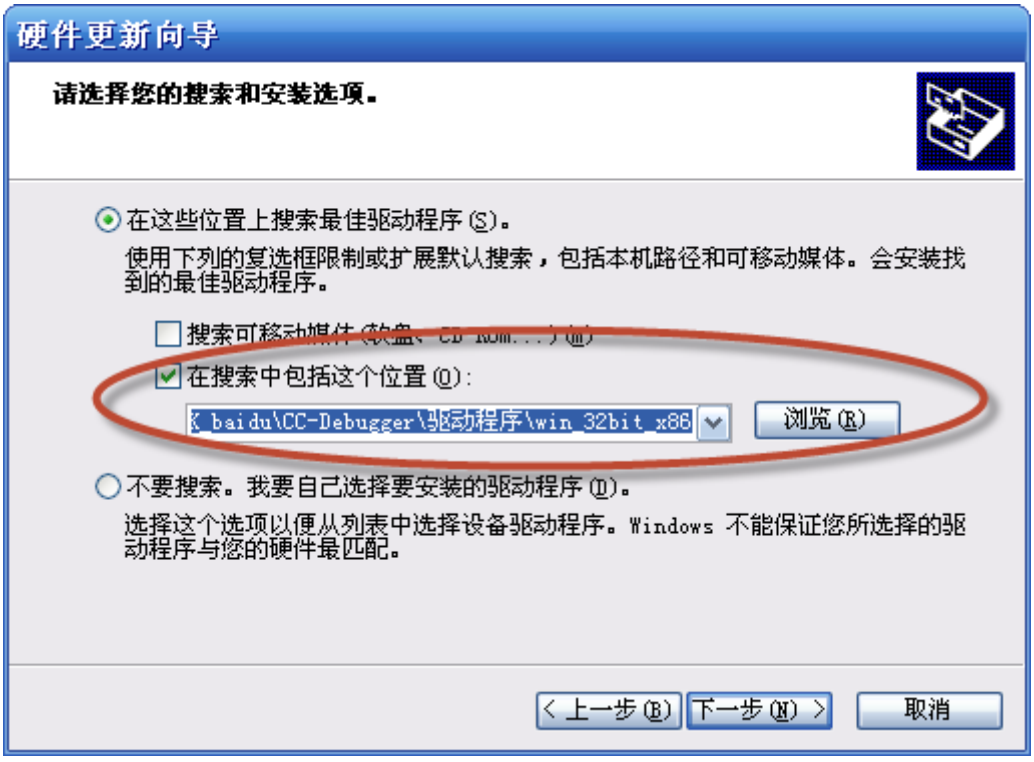
如果没有，出现安装成功。则先确定 cc-debugger 是否与 pcusb 连接 OK，连接 OK 后，电脑上肯定会出现新新硬件的提醒。如果插上电脑没有任何反应，更换 mini-usb 线。

如果有提示新硬件，但显示驱动未能安装成功，打开电脑的设备管理器。如下图，



如果框起来的有感叹号或者别的，需要更新一下仿真器的驱动。右击 CC Debugger，或者 SmartRF04EB，选择更新驱动程序。





定位到我们提供的仿真器驱动即可，该驱动同样适用于 SmartRF04EB 和 USBdongle。或者将驱动程序定位到 Flash Programmer 的安装目录下的驱动程序文件夹，默认的安装路径为：C:\Program Files\Texas Instruments\SmartRF Tools\Drivers\cebal\win_32bit_x86 同级目录还有一个 win_64bit_x64 的文件夹，这里的驱动适用于 64 位系统。

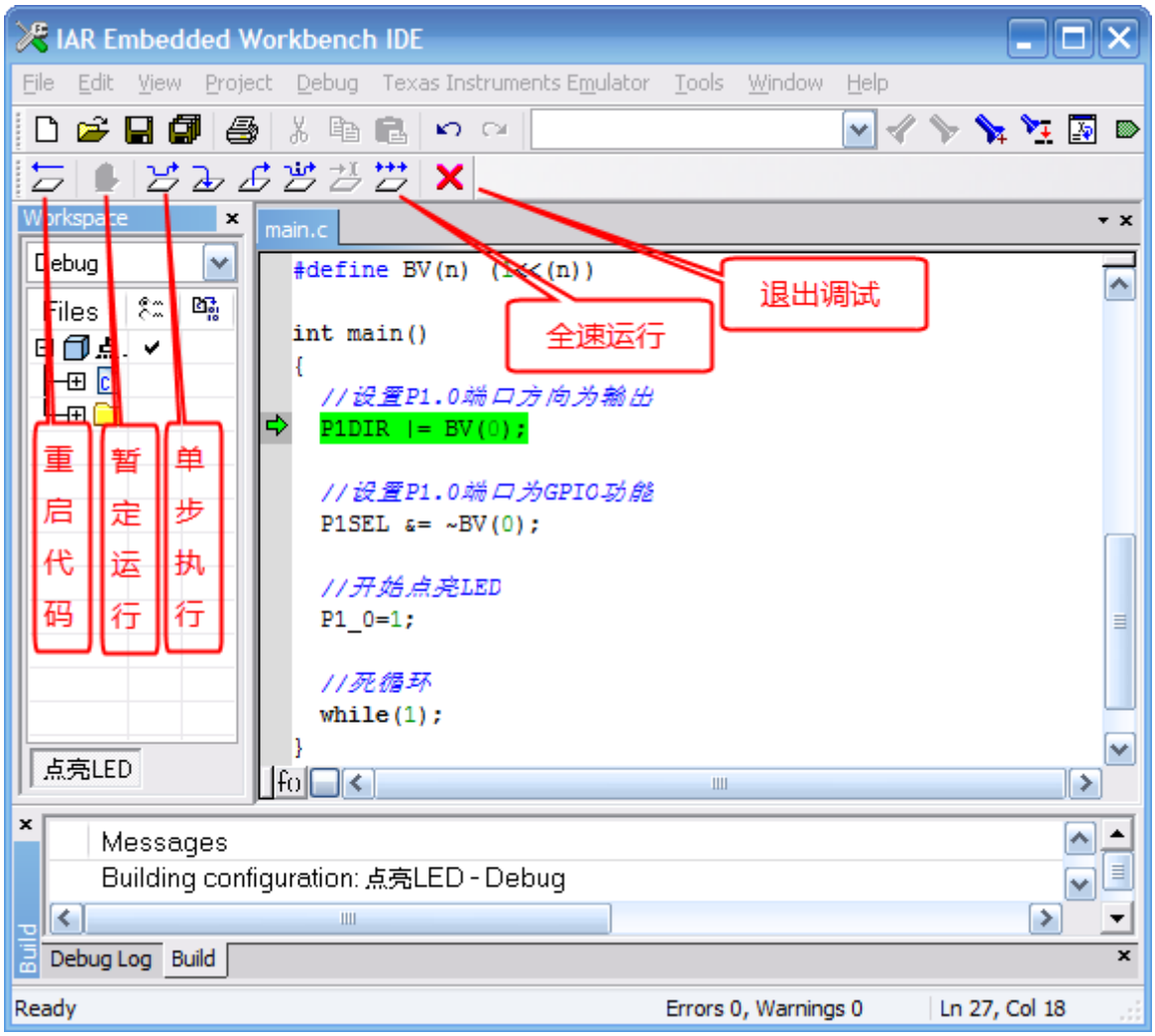
CC-Debugger 仿真器注意事项

CC-Debugger 仿真器支持宽目标电压，这是因为仿真与目标芯片之间有四颗电平转

换芯片，这四颗电平转换芯片需要目标板提供工作电压，也就是说，你的目标板有电压有多高，芯片就会将信号转到多高。因此仿真器调试接口的第 2 脚需要开发板提供工作电压，也可以简单的将 CC-Debugger 的第 9 脚和第 2 脚短接。（3.3V 工作电压，仅用四线调试接口的话）

2.3.2 程序仿真调试

开始运行的 iar 截图如下。



至此，我们完成了第一个 IAR 工程的创建和第一个程序的编写编译与运行，日后的开发过程中几乎都是重复这样的工作，只不过代码会越来越多，需要管理的文件也越来越多。但请保持信心，终有有一天你能轻松驾驭他们。

其他 CC254X 基础基础程序

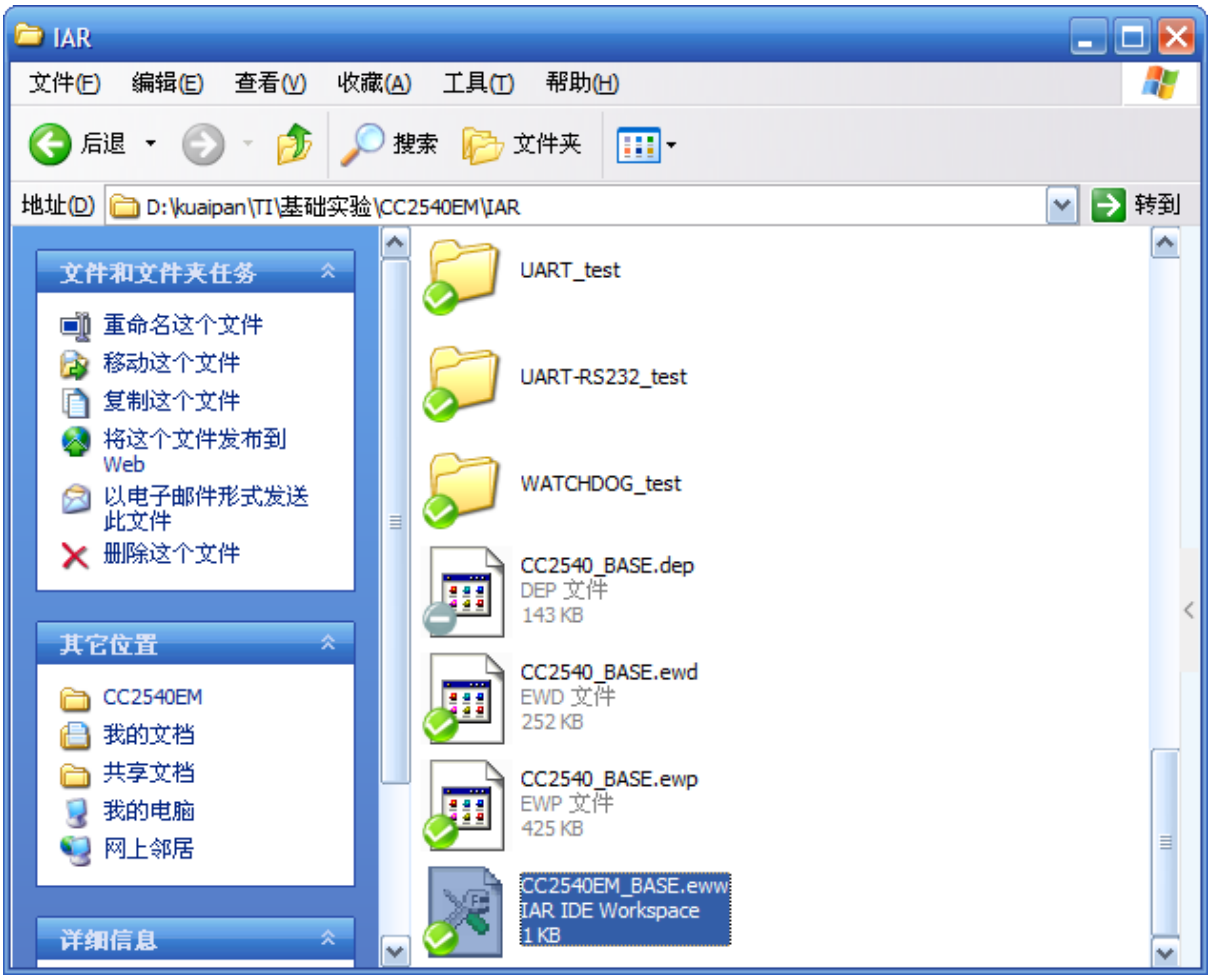
我们已经创建好了所有的基础测试程序，他们被创建在同一个 IAR 工程内，源码目录为 D:\baidupan\CC254XDK_baidu\源码\CC254X 基础测试代码\ CC254XEM.rar
解压该工程文件，会得到如下目录：

Source 文件夹

内为所有源码文件

IAR 文件夹

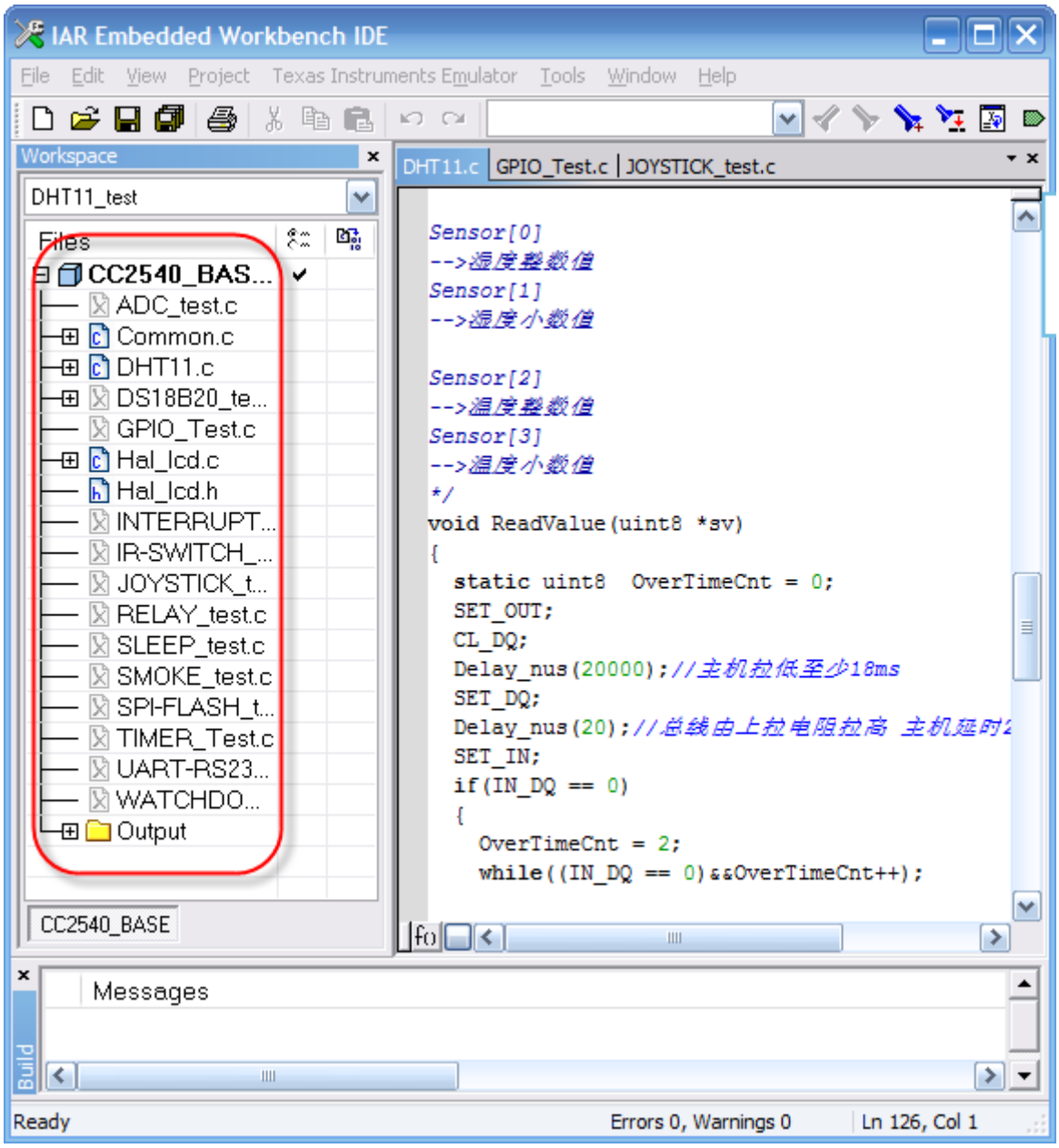
为 IAR 工程文件，进入 IAR 文件夹，然后双击打开 IAR 工程 CC254XEM_BASE.eww，如下图：



打开后的 IAR 工程如下图：

在 Workspace 下面的配置列表中，配置了各个测试程序，例如选择 DHT11_test，Source 目录中有关 DHT11_test 的将被自动包含进来，如下图，每个测试程序共用了 hal_lcd.c Common.c 两个源文件，main 函数在以测试程序名的源文件内。并且代码做了可读性注释。

其他测试程序都是类似的，和点亮 LED 一样简单，学习 BLE 协议栈最重要的是快速入门 BLE 协议栈的开发，因为底层的驱动几乎都是做好的，需要我们修改的部分极少。



2.4 本章小结

本章主要讲述了使用 IAR 进行 BLE 开发的基本流程，讲解了工程的简历、源文件的添加、编译与调试，最后介绍了程序仿真调试的基本方法。

本章附录

使用 Flash Programmer 直接烧写 hex 到芯片中。

我们详细的介绍一下使用 TI 的 flash programmer 烧写工具烧写 hex 文件，flash programmer 是 ti 开发的 hex 文件烧写工具，通过 Flash Programmer 不光可以给目标芯片烧写程序，而且还而已更新 CC-Debugger 仿真器的固件程序，功能非常强大。

设置 IAR 产生 Hex 文件

我们这里分为三种情况

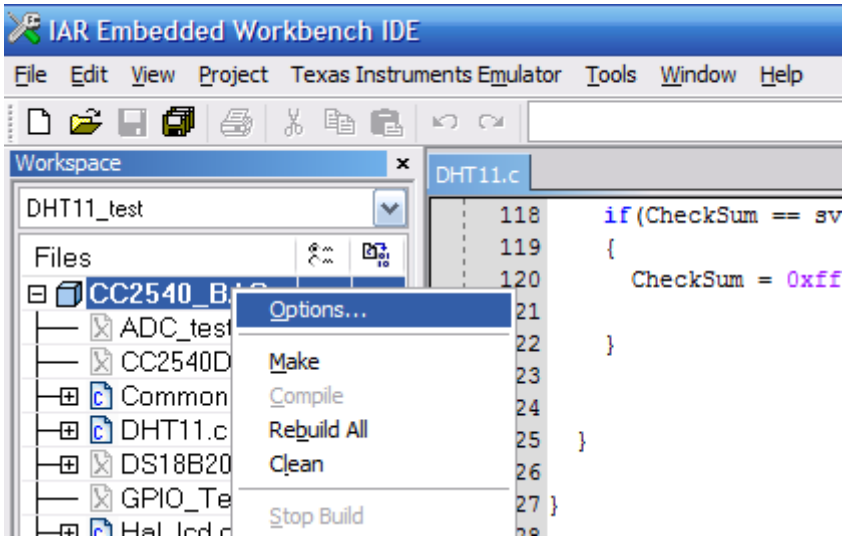
- 普通 IAR 工程产生 Hex 文件。
- BLE 协议栈工程产生 Hex 文件。

下面详细介绍

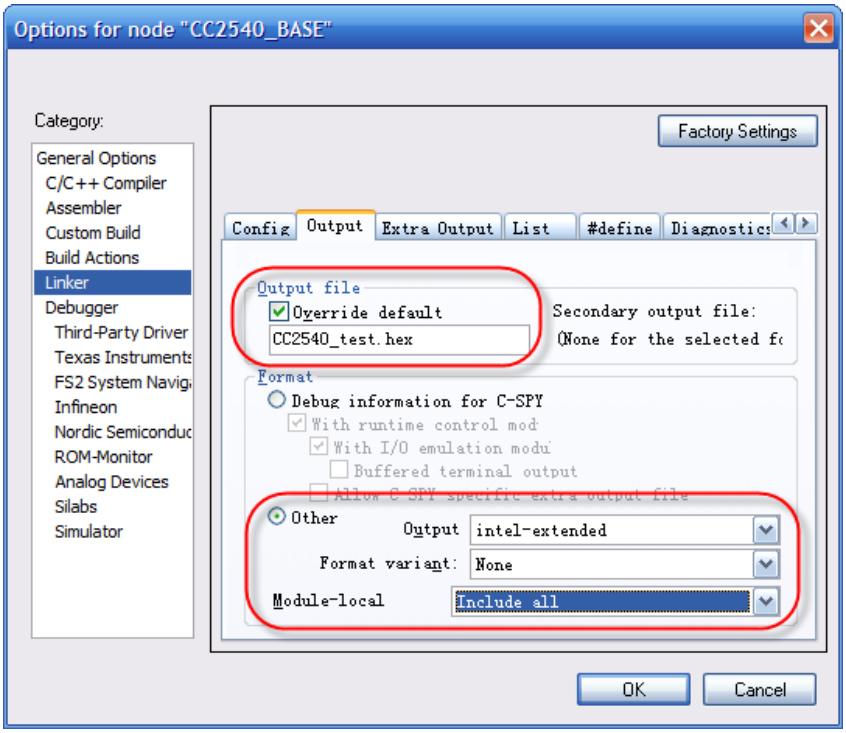
普通 IAR 工程产生 Hex 文件。

以 CC2540EM 测试代码为例，使用 2540EM，相应处选择 CC254Xxxx 即可

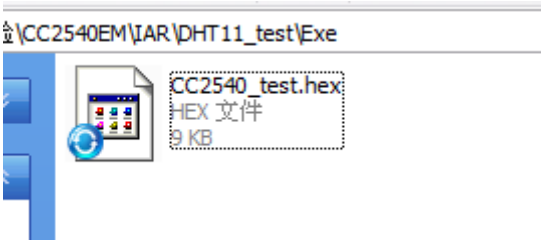
打开 CC2540EM_BASE.eww，右击 Workspace 中的项目名称，然后打开 Options 对话框。如下图。



在打开的 Options 中选择 Linkder 中的 Output 选项卡，设置如下图：



然后重新编译工程，这时，会在工程的相应配置目录（对应 Workspace 中列表框中的名称）下的/Exe 文件夹下产生可烧写的 hex 文件。如下图：



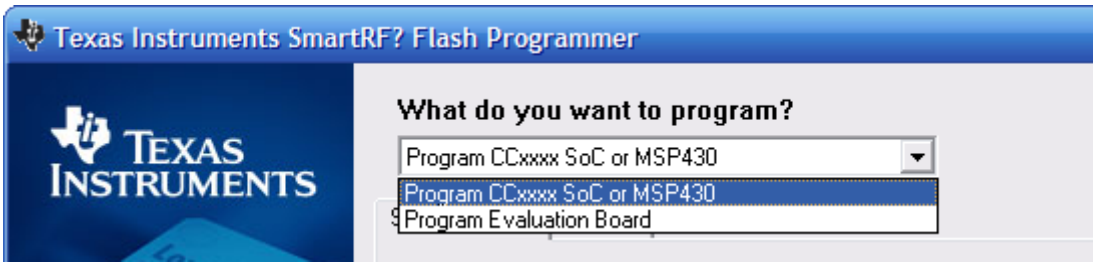
BLE 协议栈工程产生 Hex 文件。

BLE 协议栈工程默认设置了生成 hex 文件，无需另外配置。

烧写过程介绍

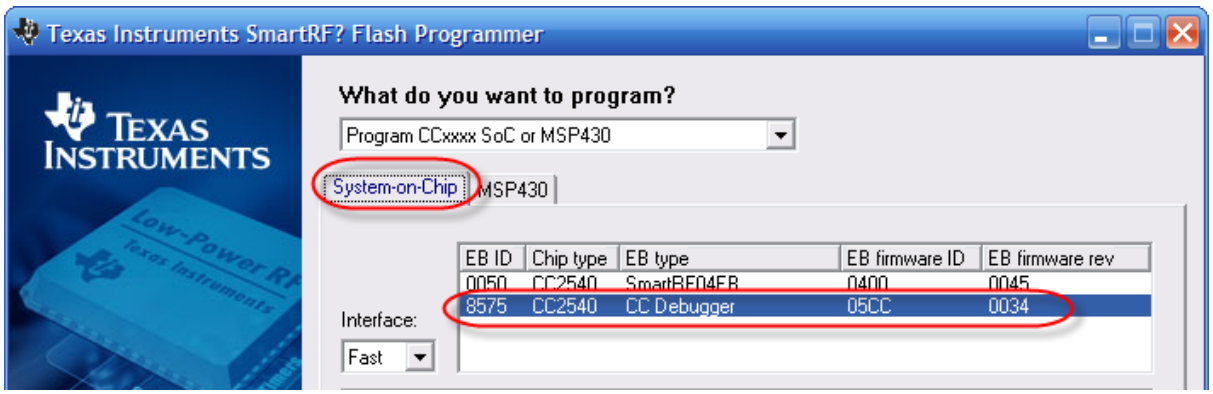
选择烧写类别

打开 Flash Programmer。在“What do you want to program”中选择“Program CCxxxx Soc or MSP430”，如下图，请注意，Program Evaluation Board，是用来烧写 cc-debugger 仿真器固件用的，请勿随意操作，以免损坏仿真器。



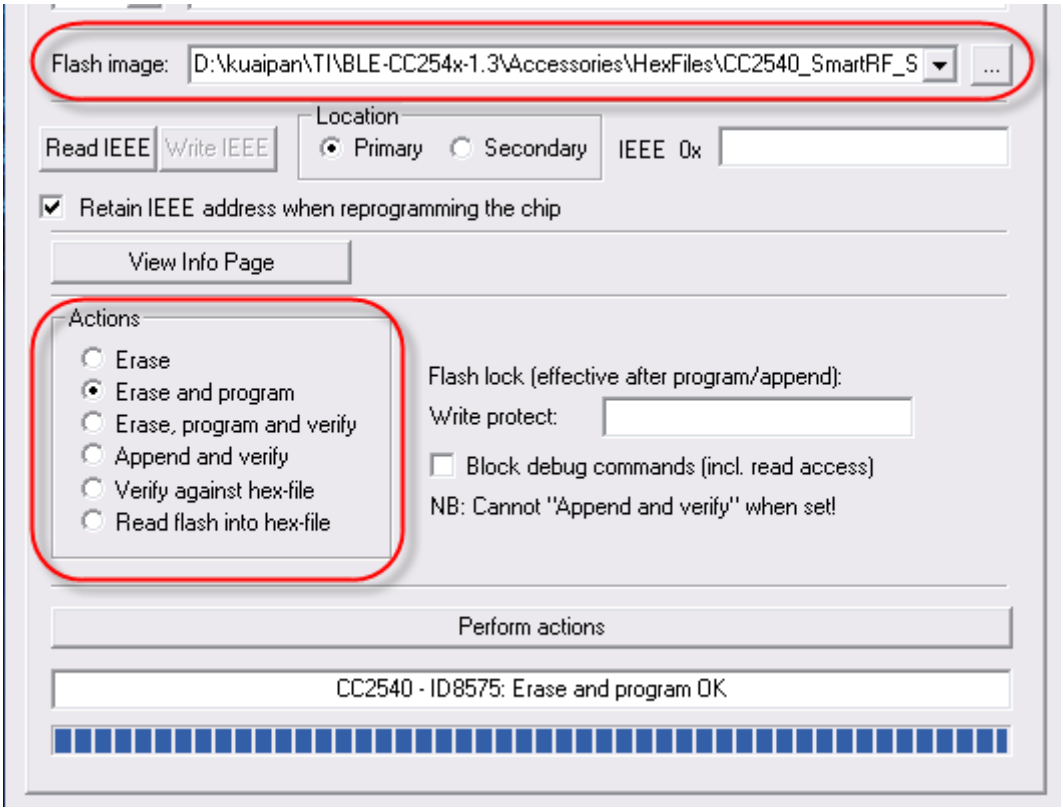
查看设备连接

选择已连接的需要烧写的连接，如果没有出现列表，请检查仿真器与目标板的连接是否 OK，并且按仿真器复位按钮，CC-Debugger 变绿灯或者 SmartRF04EB 亮灯表示识别到芯片，这时 Flash Programmer 中会出现设备列表，如果仿真器已经识别到芯片，而软件中没有出现设备列表，则请检查仿真器驱动是否安装成功，请参考前面的章节。



选择需要烧写的设备和需要烧写的 hex 文件

在 Flash Image 中定位到需要烧写的 hex 文件，并且在 Actions 中选择合适的操作，如下图：



Action 中的常见操作：

Erase：擦除芯片，当遇到芯片被锁住时，可以点该选项擦除芯片。

Erase and program：擦除并且编程，快速对芯片编程

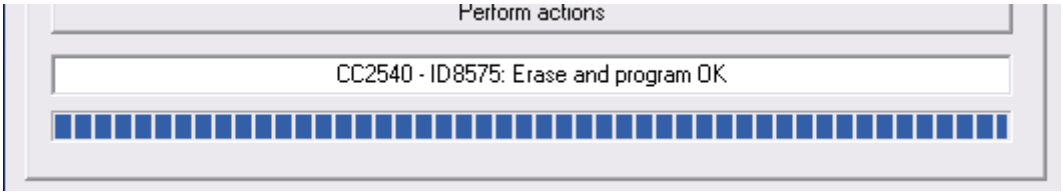
Erase program and verify：步骤 2 基础上添加 hex 验证操作，比较耗时。

Read flash into hex-file：从芯片中读出 hex，并且写到（覆盖）在 Flash image 中选择的 hex 文件。

其他选项含义请查阅 ti 帮助手册。

执行烧写

点击 Perform actions 开始烧写，烧写成功后，会显示　program OK。

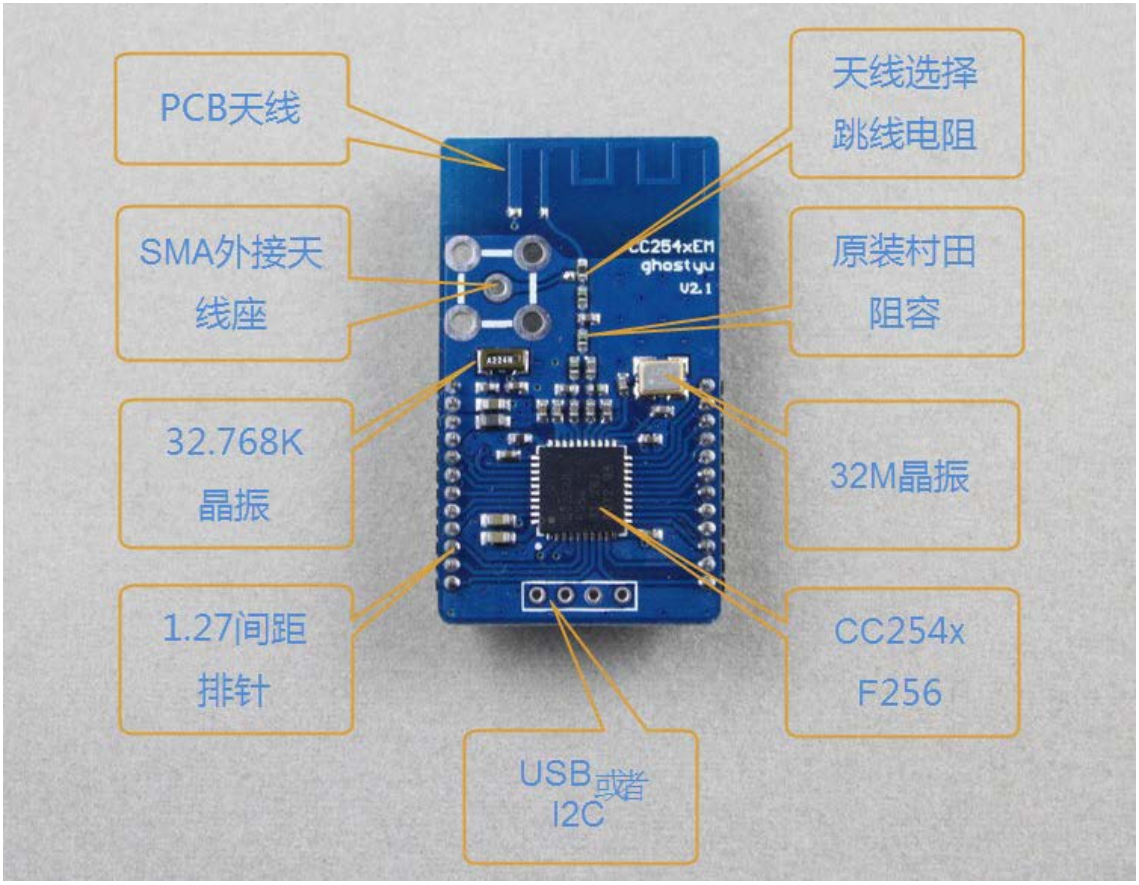


第 3 章 CC254X 开发板硬件资源详解

进行 BLE 无线网络的开发，首先，需要有相应的硬件支持（尤其是需要支持 BLE 协议栈的硬件）。

3.1 核心板硬件资源

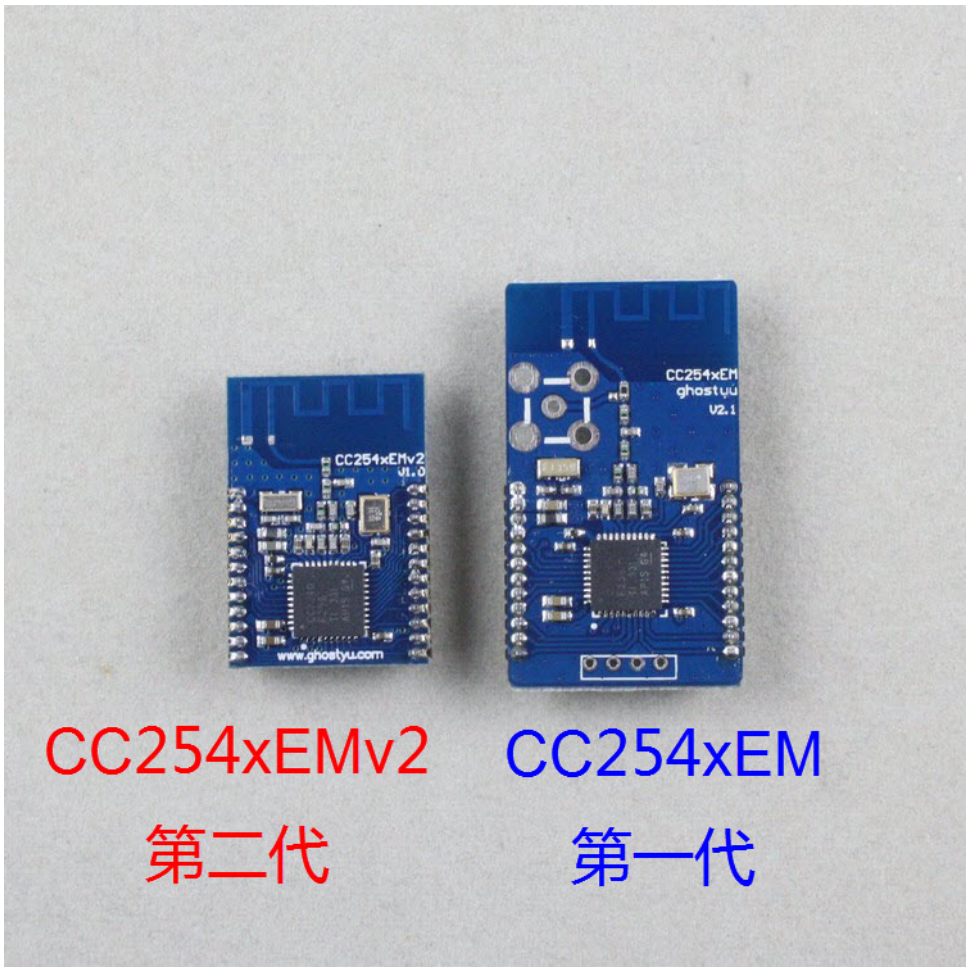
第一代 CC254xDK 开发套件配套核心板如下图：
CC254xEM



第二代 CC254xEK 开发套件配套核心板如下，有两个版本，PCB 天线和 SMA 外接天线。
CC254xEMv2-PCB 天线

CC254xEMv2-外接 SMA 天线

第一代 CC254xEM 和第二代 CC254xEMv2 核心板对比如下：
两个方面的改动比较大，首先是外形尺寸，第一代为 35MM*20MM，第二代为 25*17MM，面积缩小了 40%
另外第二代 CC254xEMv2 提供两个天线形式，PCB 天线和外接 SMA 天线，虽然在 ble 低功耗蓝牙应用中，更多使用的是小体积的天线，例如 PCB 天线、陶瓷天线，但是也有一些特殊的应用范围，例如室内定位等需要外接 SMA 来提供可靠的信号。



3.1.1 CC254X 简介

CC2254x 内部嵌入了一颗增强的 8051 内核，他和普通 51 单片机不同之处在于，它有更快的运行速率 32Mhz，一条指令一个时钟周期。另外集成了 2.4-GHz IEEE 802.15.4 Compliant RF Transceiver，全面支持 BLE 协议栈。

更低的功耗

- RF 收发电流 24ma
- 睡眠电流只有 1uA
- 宽工作电压 2V~3.6V
- 8K 的 RAM 以及高达 256K 的 FLASH
- 丰富的外设，包括 SPI、UART、GPIO、TIMER，USB/I2C 等。

3.1.2 天线及巴伦匹配电路设计

CC254X 外部仅需几个简单的阻容网络即实现复杂的 RF 前端。这部分的电路也叫做巴伦匹配电路，这部分的结构好坏对通信距离，系统功耗都有较大的影响。TI 已经提供了非常可靠的参考设计，我们按照 ti 的参考设计开发自己的电路即可。

天线设计可以使用 PCB 天线，也可以使用外接 SMA 的杆状天线。根据不同的应用来选择，天线寄巴伦匹配电路设计如图 3-2 所示

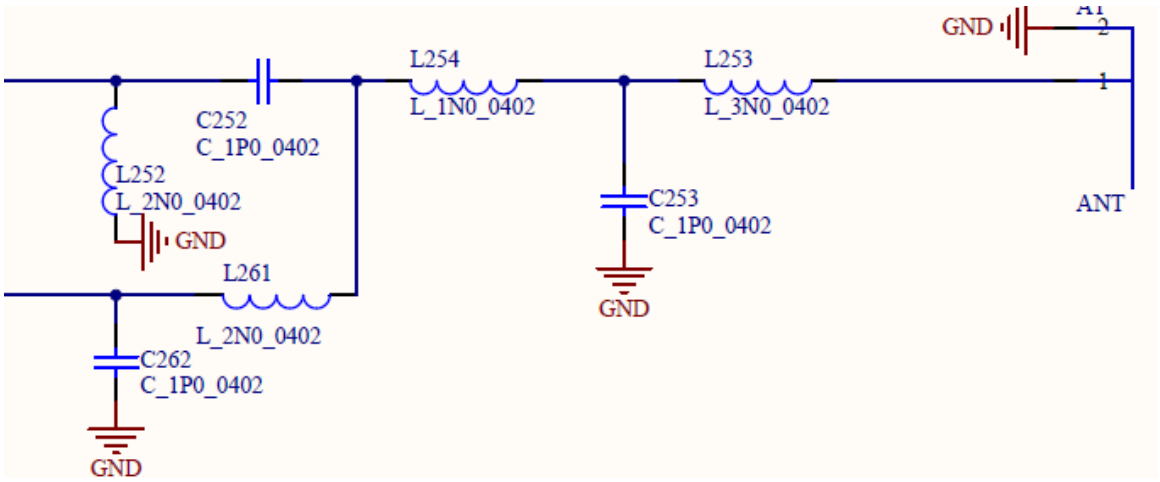
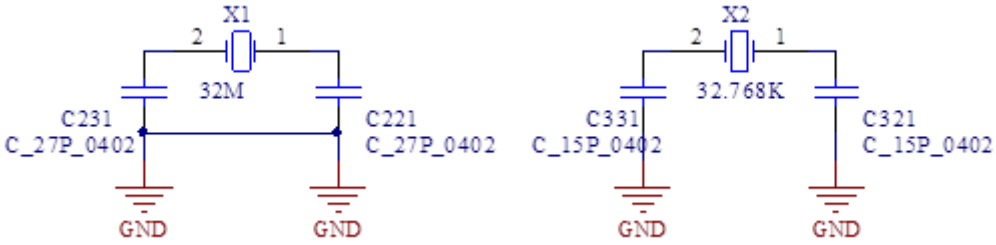


图 3-2 巴伦结构

3.1.3 晶振电路设计

CC254X 需要 2 个晶振，32MHz 和 32.768K，晶振电路接口如图 3-3



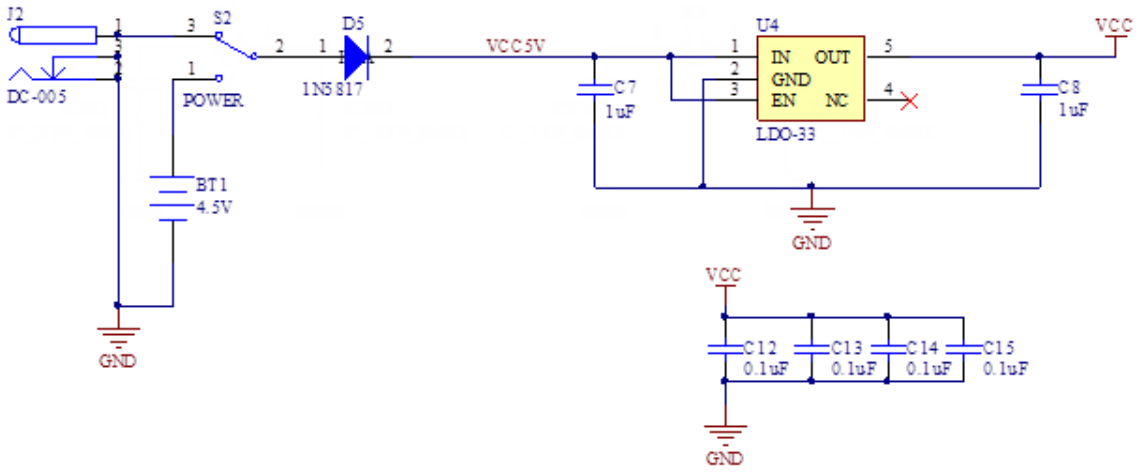
如果不需要休眠，32.768K 外部晶振可以不用。

3.2 底板硬件资源

我们的开发板采用核心板和底板分离的设计，这样可以复用底板，因为 CC2531、CC254X 以及低功耗蓝牙芯片 CC254x，硬件上是相同的。

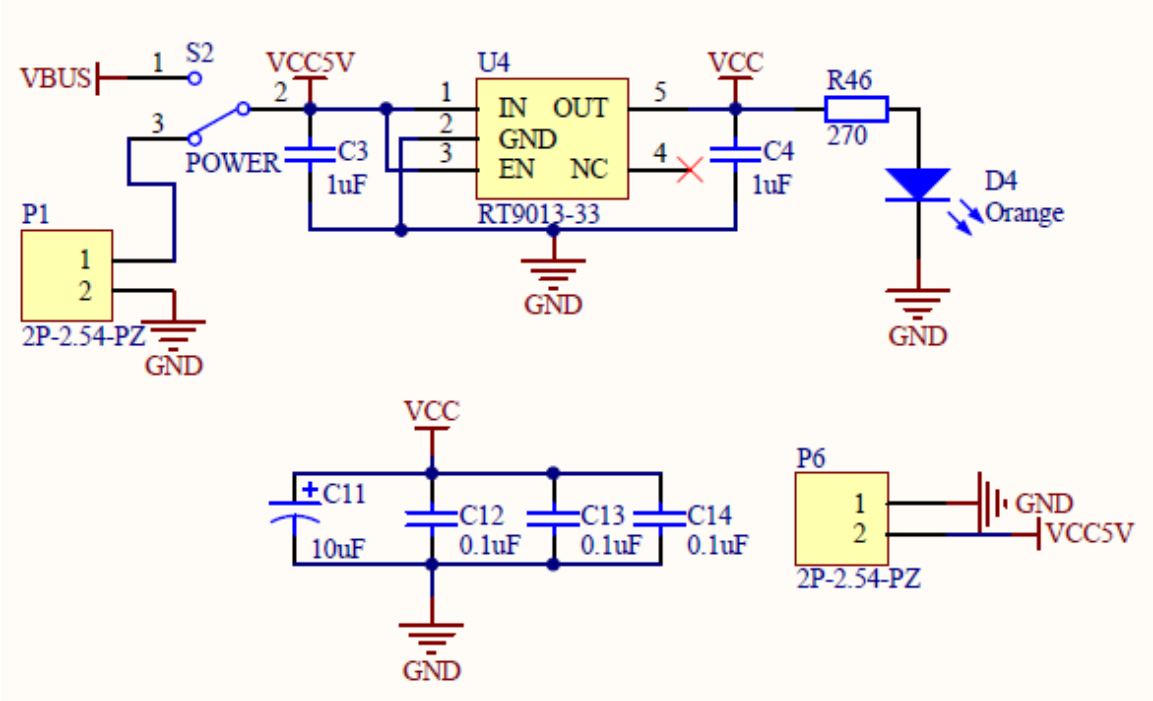
3.2.1 电源电路设计

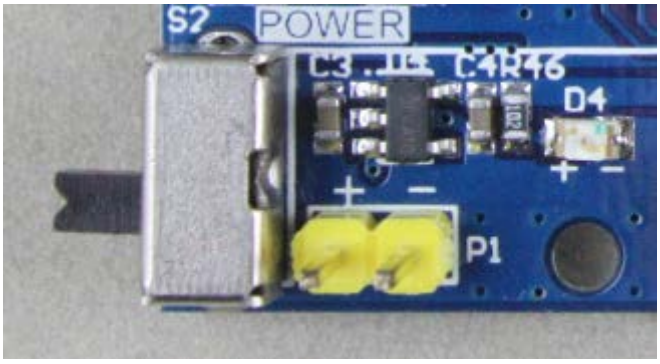
第一代 CC254xDK 开发套件中的 SmartRF 开发板可以使用我们提供的 USB 转 DC 电源从 USB 接口上取电，另外也可以使用锂电池等供电，直接插在 BT1 上，电压输入范围为 3.4V 到 6V。



第一代 SmartRF 开发板电源管理电路

第二代 CC254xEK 开发套件中的 New SmartRF 开发板使用 USB 转串口的 USB 供电或外接锂电池供电，与第一代相比，加入了一颗电源指示 LED，如下图：

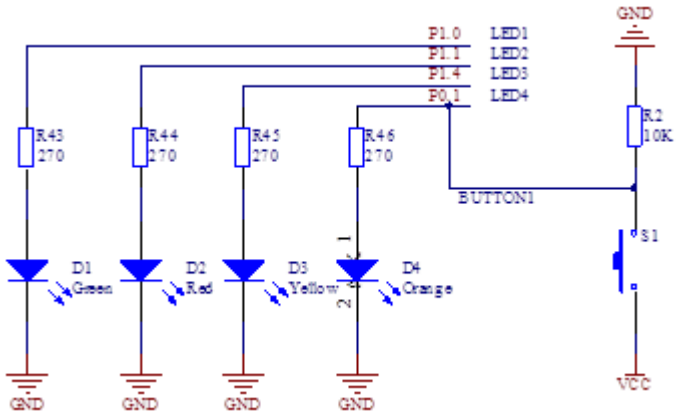




第二代 New SmartRF 开发板电源管理电路

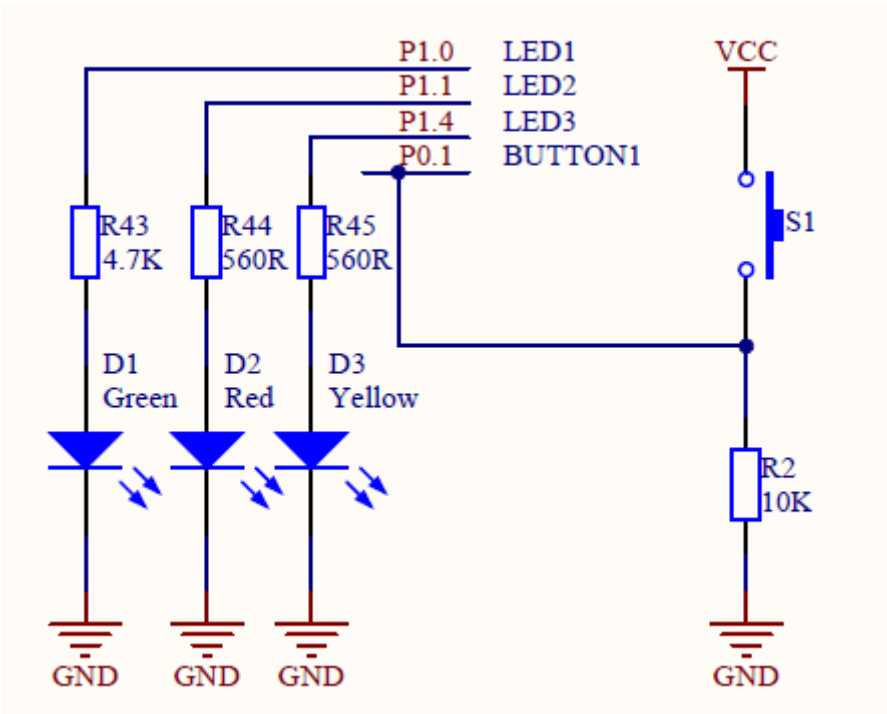
3.2.2 LED 电路设计

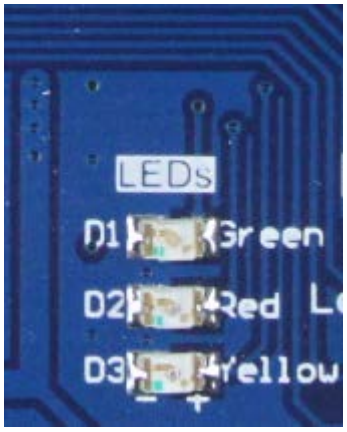
第一代 SmartRF 开发板采用与 TI 完全兼容的四色 LED，高电平驱动，另外 D4 和 S1 按键公用一个端口。



第一代 LED 驱动和按键 S1 驱动电路

第二代 New SmartRF 开发板在 TI 基础上去掉了协议栈中未使用的 LED4，然后改为电源指示灯。如下图：

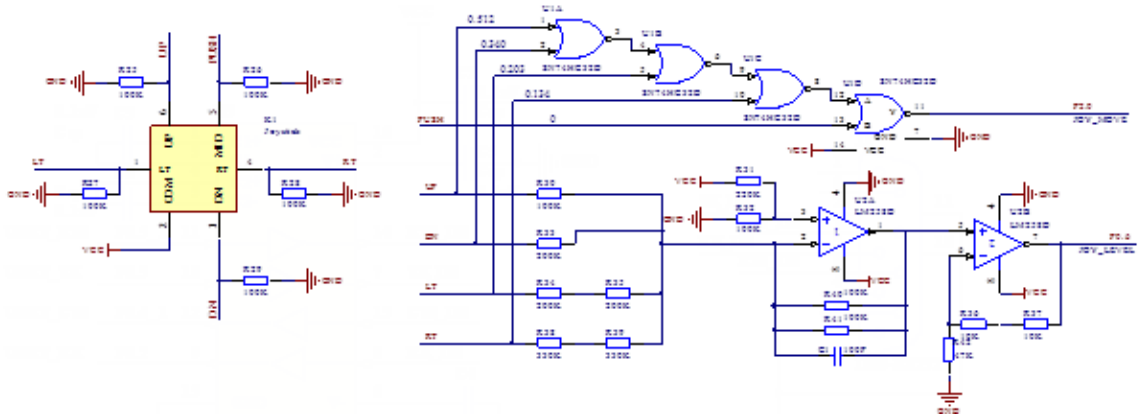


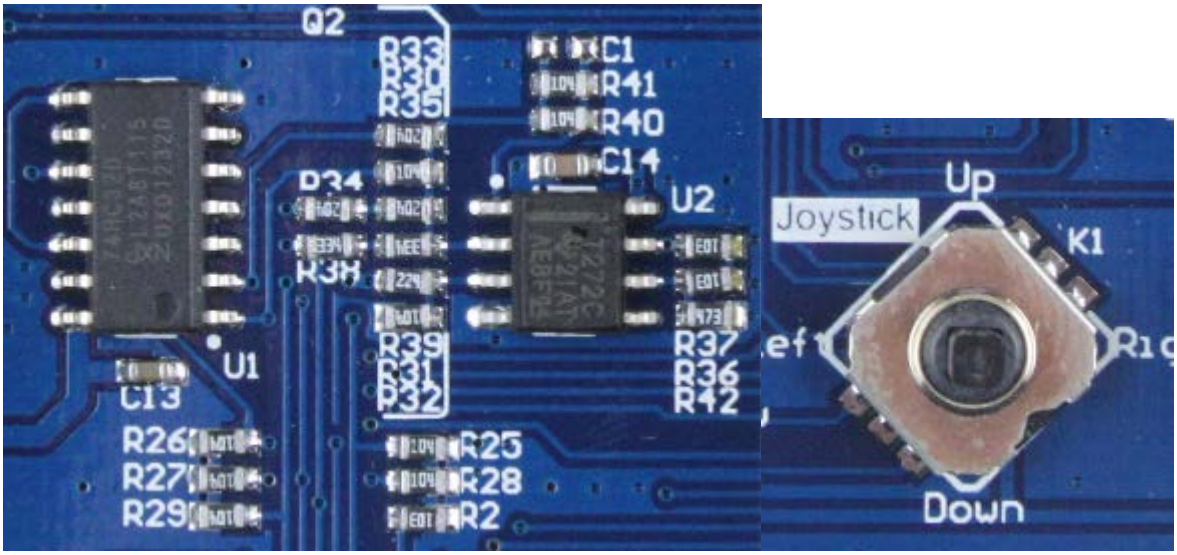


3.2.3 五向按键电路设计

协议栈另外一个非常重要的扩展电路就是五向按键，几乎每个协议栈 demo 都会用到无五向按键来辅助操作。

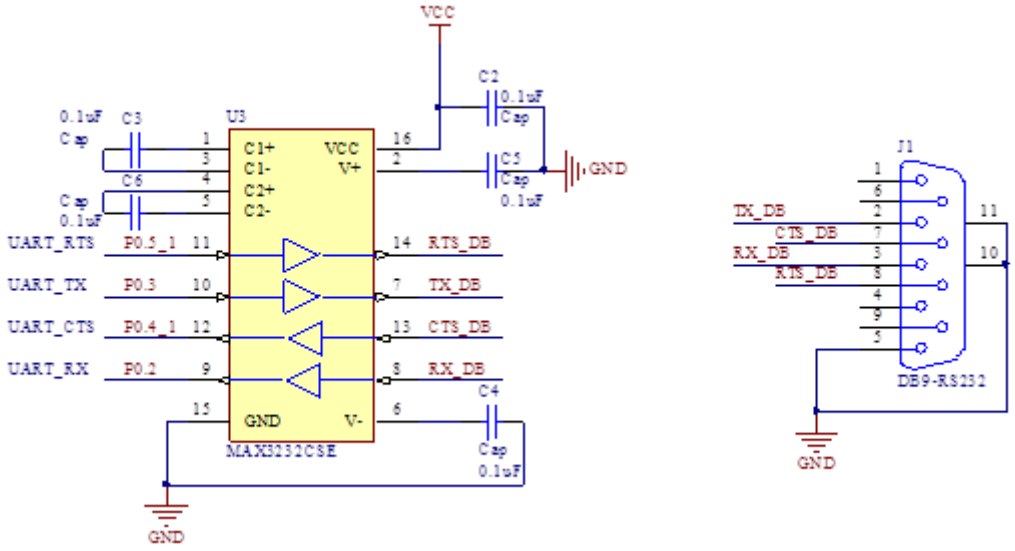
五向按键的电路比较复杂，但是原理非常简单，当按键按下时首先产生一个高电平，触发一个 GPIO 中断，然后通过放大器输出不同的电压值，当 CC254X 接收到中断后开始去读五向按键的电压，不同个方向按下产生的电压值不同，这样就实现了 joystick。



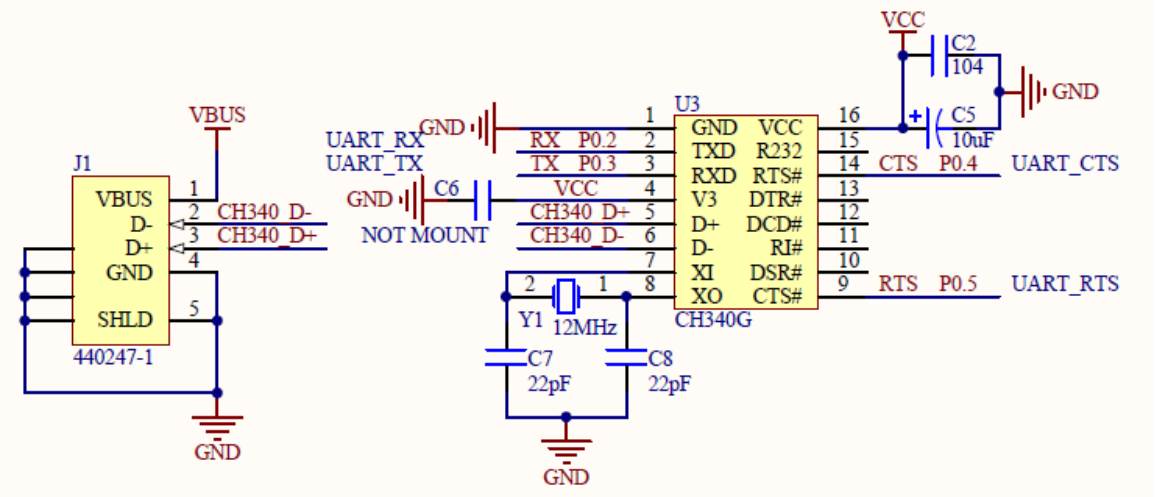


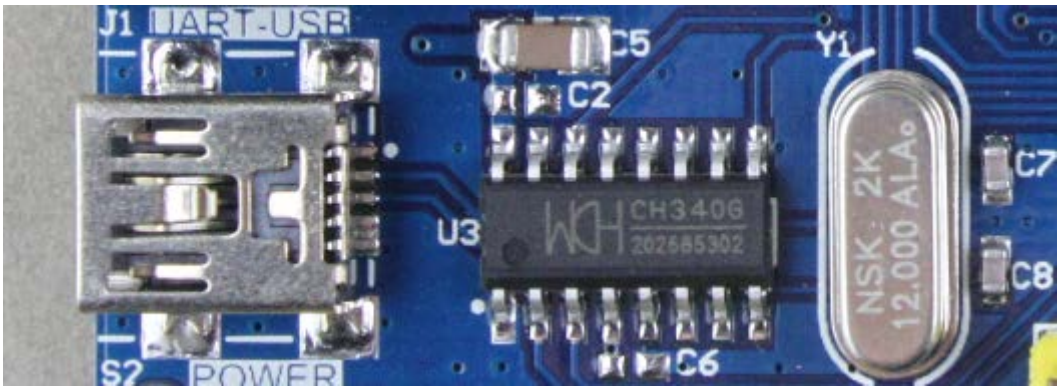
3.2.4 串口电路设计/USB 转 UART

在第一代中的 SmartRF 开发板，我们使用 RS3232 将 CC254X 的 UART 转成 RS232，然后方便与 PC 连接。（BB 板的 uart 是没有转换，是 ttl 电平。），这部分值得注意的地方时串口的 flow control，底板上我们复用了 CTS 和 RTS 这两个引脚。CTS 和 RTS 并没有与 max3232 芯片直连，而是通过一个接插件 P10，如果软件中需要使用流控制，需要用杜邦线或者跳线帽短接 P10 的 1、2 两脚以及 3、4 两脚。



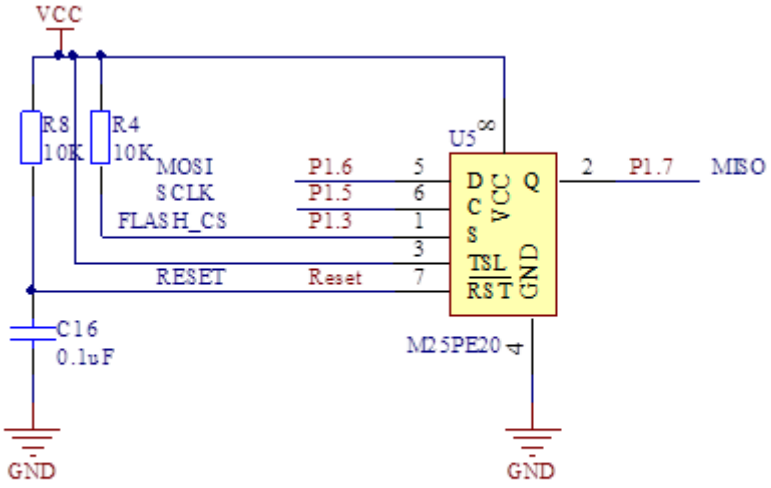
第二代 NewSmartRF 开发板采用 CH340G 芯片将 uart 直接转 USB，现在电脑上 RS232 接口用的比较脚，采用 USB 接口，方便笔记本等无串口电脑使用。





3.2.5 外部 Flash 电路设计

CC254X 内部有 256Kflash 用来存储程序以及数据，当用到 TI 的 OAD 技术时，需要使用外部的 flash 作为缓存。

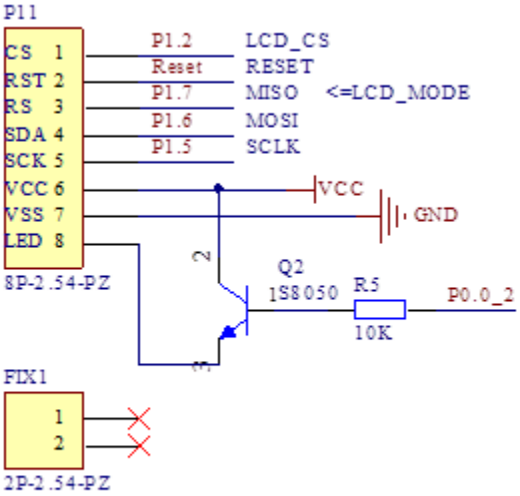


2MByte Flash

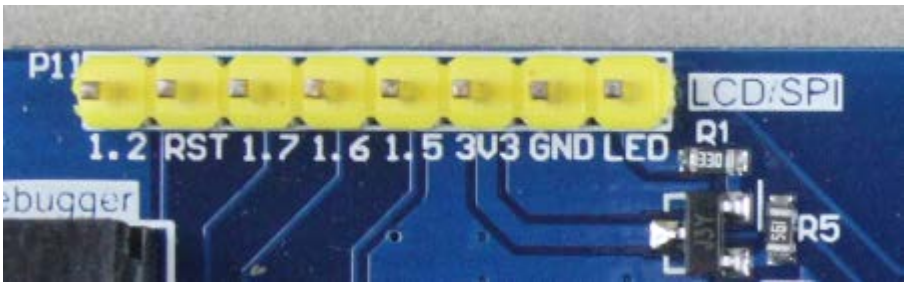
第二代 NewSmartRF 开发板上移除此刻芯片，BLE-OAD 技术通常使用内部的 Flash 空间。

3.2.6 LCD12864 电路设计

我们采用小型的 12864 作为系统的显示系统，该 lcd 采用 spi 接口。

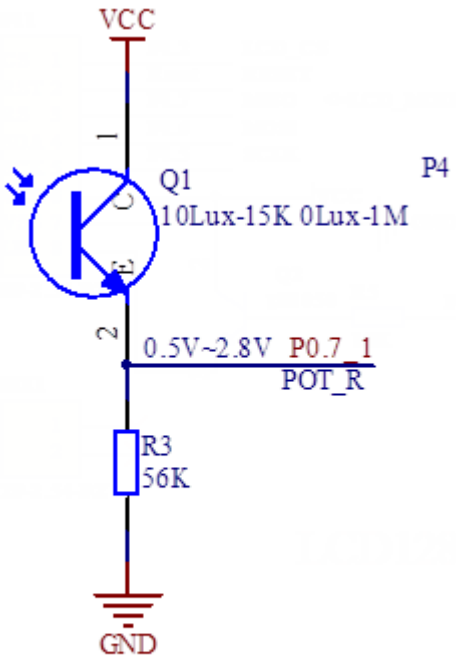


LCD12864



3.2.7 光敏电阻电路设计

SmartRF 开发板板载一个光敏电路，通过 56K 的电阻分压连接到 P10 接插件的 P0.7_1，可以使用跳线帽短接 P0.7_1 与 CC254x 的 P0.7 引脚，这样可以通过 ADC 采用输出电压，判断当前的光线。

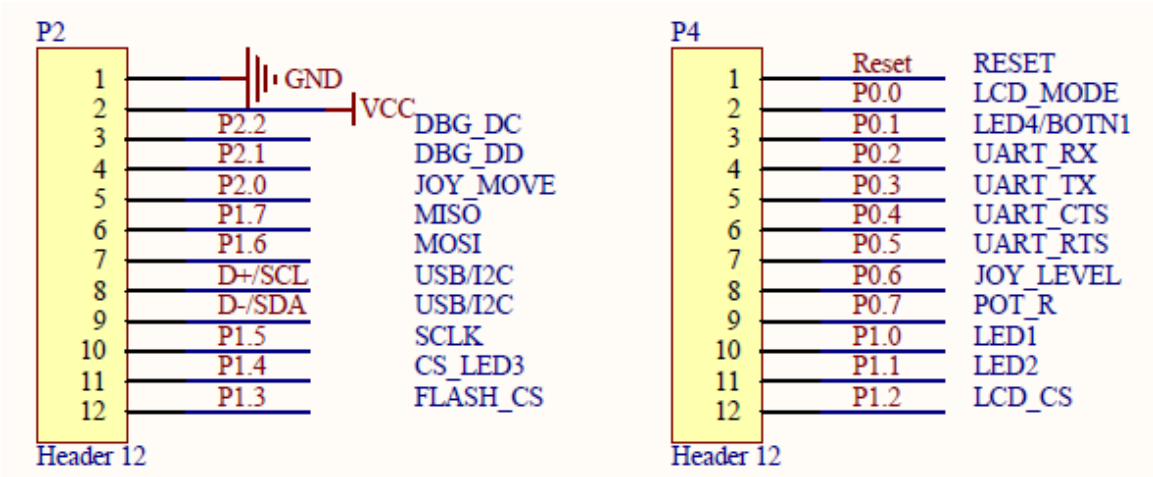


光敏电阻

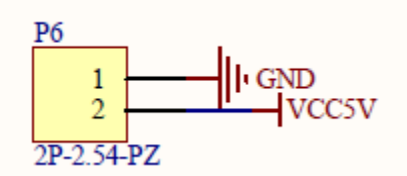
第二代 New SmartRF 开发板移除了该光敏电阻。

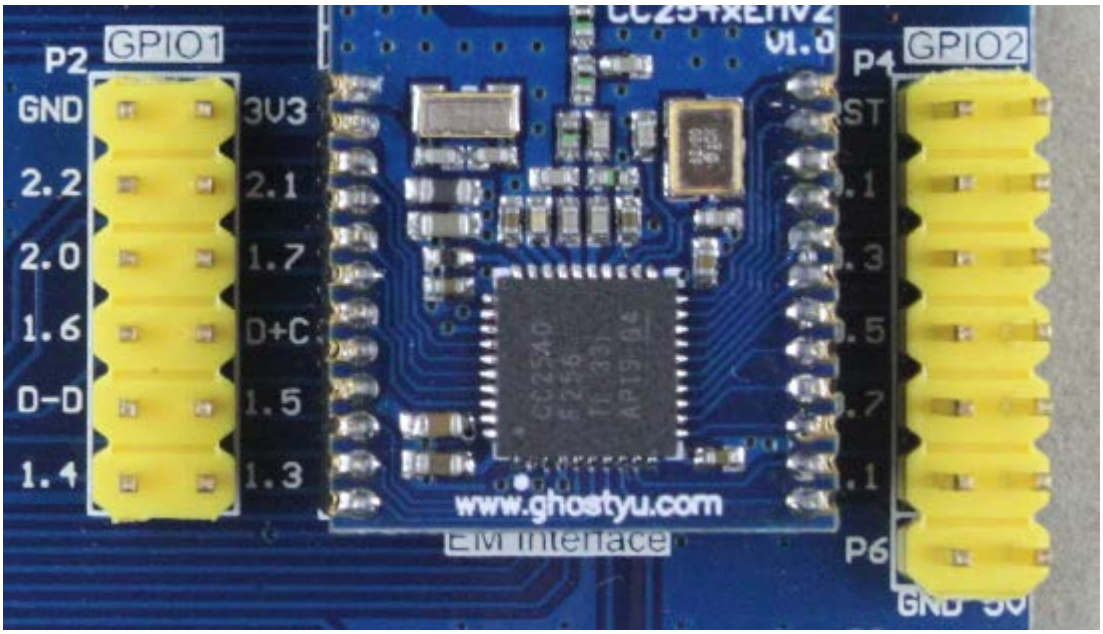
3.2.8 开发板扩展接口设计

第二代 NewSmartRF 开发板将 CC254x 的 GPIO 全部通过 2.54 间距的双排针引出，极大方便了各种开发需求。



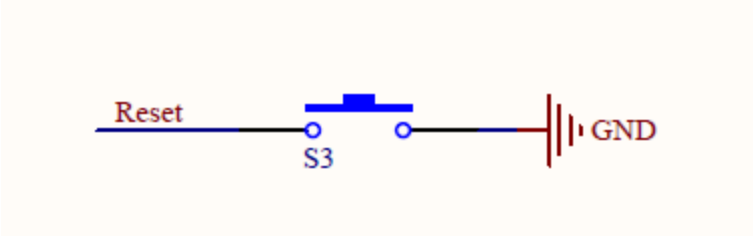
另外还有板载的 5V 电源接口





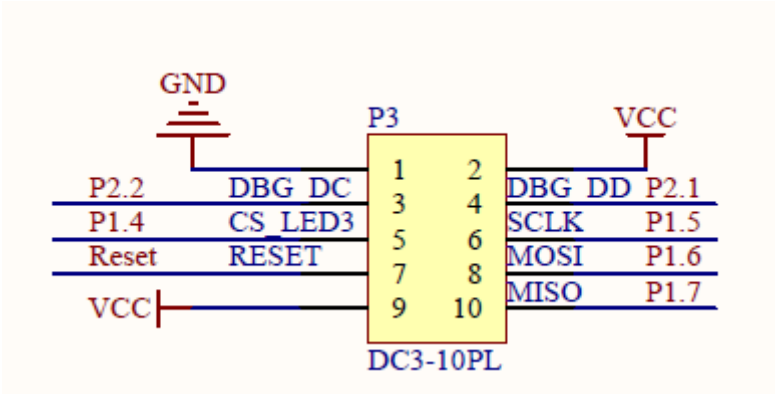
3.2.9 复位电路

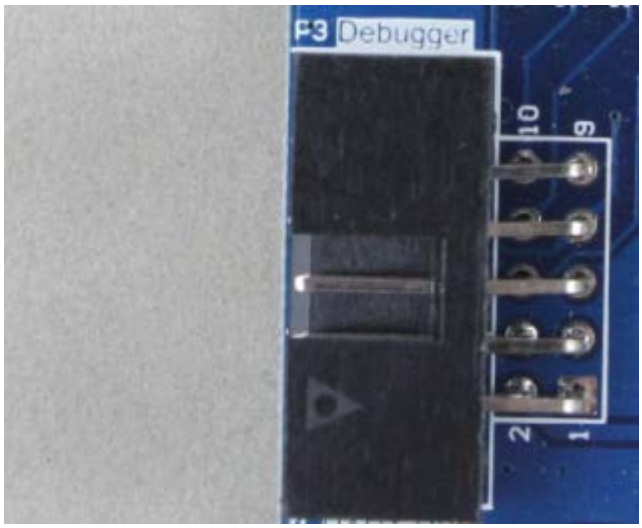
CC254x 内部集成了上电复位电路，为了方便程序调试，我们在板子上加了一个按键用来在线复位， 尤其但是用 uart 转串口时，直接断电会导致 PC 设备异常，这样 reset 按键就能够起到很好的系统复位功能。



3.2.10 Debugger 电路

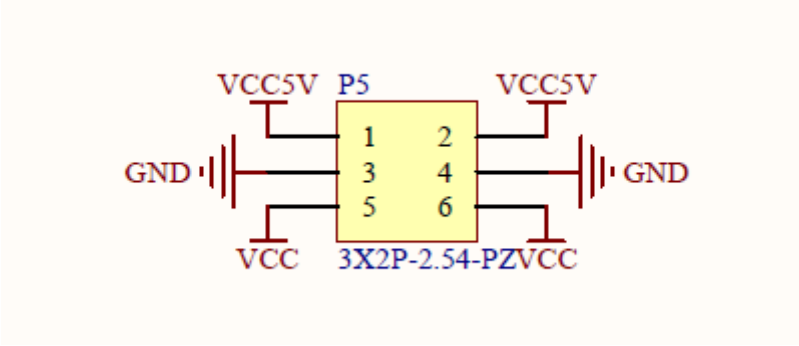
NewSmartRF 开发板使用标准的 CC-Debugger 调试接口，为了方便调试，特意将 dbg 接口的 9 脚和 2 脚短接，这样可以使用 CC-Debugger 为开发板供电。如下图：





3.2.11 电源扩展电路

开发过程总会经常使用开发板给外接的电路提供工作电压，这时就需要开发板能够有这样的接口方便向外输出电压。我们开发板上提供了两种电压输出 3.3V 和 5V（具体电压取决于开发板的电源输入）。



3.3 本章小结

本章主要讲解了 SmartRF 开发板的硬件构成，给出了具体的硬件电路，在具体项目开发过程中，用户需要借口系统所需重新设计扩展电路。

第 4 章 BLE 协议栈入门

BLE 无线网络涉及电子、电路、通信、射 Zi 等多学科的知识，这对于入门级学习来说，无形中增加了学习难度，很多读者看 BLE 协议、射频电路…. . 学了半年甚至更长的时间，但是连基本的点对点通信都无法实现，更别说根据对 BLE 协议的理解来实现正常的无线网络部署工作了。

基于此原因，本书推荐另一种学习思路，不是将学习重点放在复杂的 BLE 协议、射频、天线等知识，而是直接进行 BLE 无线网络点对点通信的学习，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，并给出相应的指示。很简单的功能，但是这里涉及以下问题：

- 数据在协议栈里面是如何流动的；
- 如何调用 BLE 协议栈提供的发送函数：
- 如何使用 BLE 协议栈进行数据的接收；
- 如何理解 BLE 协议栈；
- 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 BLE 协议栈提供的函数来实现基本的无线传感器网络应用程序开发：
- 系统硬件对 BLE 协议都提供了哪些支持。

一个看似简单得不能再简单的实验引起了读者对于 Bluetooth-LE 低功耗蓝牙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 BLE 无线网络的技术内幕，触摸那神圣的无线通信世界，感知那“传说中”的无线传感器网络，读者的 Bluetooth-LE 低功耗蓝牙开发之旅由此开始……

本章只是带领读者从功能上理解协议栈，并没有给出具体的概念性的知识点，展示了 BLE 无线网络中的数据传输过程，并没有对 BLE 协议栈进行深入的讨论，在本书第 5 章中会对 BLE 协议栈的构成及工作原理进行讨论，本章的主要目的是使读者对 BLE 协议栈开发有个感性的认识。

4.1 BLE 协议栈

进行 Bluetooth-LE 低功耗蓝牙的开发，首先面临的问题是，什么是 BLE 协议栈，以及由此引发的如下问题：

- BLE 协议栈和 BLE 协议是什么关系；
- 如何使用 BLE 协议栈进行应用程序的开发。

下面对上述问题进行逐一讲解。

4.1.1 什么是 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。

BLE 协议栈将各个层定义的协议都集合在一起，以函数库的形式实现，并给用户提供一些应用层 API，供用户调用。

使用 BLE 协议栈进行开发的基本思路可以概括为如下三点：

• 用户对于 BLE 无线网络的开发就简化为应用层的 C 语言程序开发，用户不需要深入研究复杂的 BLE 协议栈；

• Bluetooth-LE 低功耗蓝牙中数据采集，只需要用户在应用层加入传感器的读取函数即可：

• 如果考虑到节能，可以根据数据采集周期进行定时，定时时间到就唤醒 BLE。

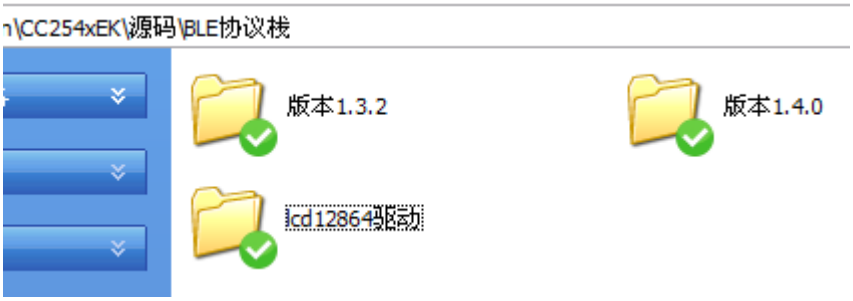
4.1.2 如何使用 BLE 协议栈

既然 BLE 协议栈已经实现了 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，在开发过程中完全不必关心 BLE 协议的具体实现细节，只需要关心一个核心的问题：应用程序数据从哪里来到哪里去。

在后面的章节中，我们会介绍数据的发送与接收。

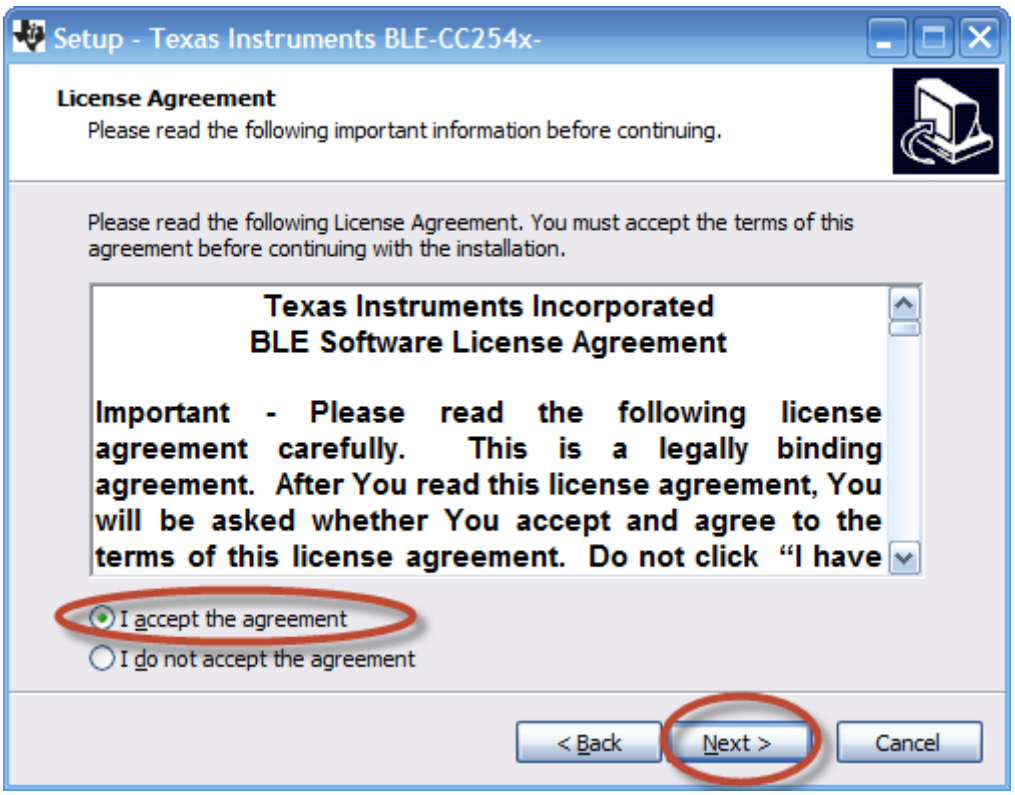
4.1.3 BLE 协议栈的安装、编译与下载

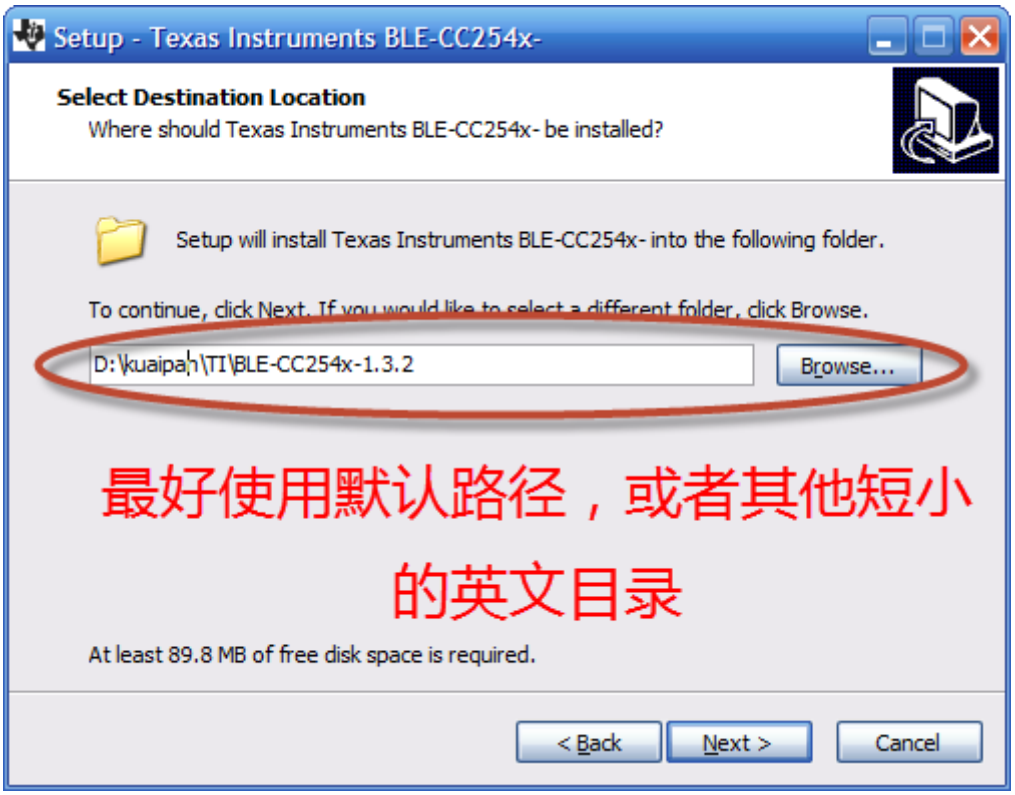
在【CC254xEK\源码\BLE 协议栈\】目录下，存放的是协议栈源码，由于 TI 会陆续更新协议栈版本，所以在这里会出现多个协议栈版本，目前最新的协议栈版本已经升级到 1.4.0,并且此协议栈版本配套的 IAR 软件版本为 8.20.2，由于当前该版本的 IAR 软件不不能完美的破解，因此，当前我们继续使用 1.3.2 的协议栈以及 8.10.4 的 IAR 版本。



在协议栈目录还有一个 lcd12854 驱动文件夹，这里存放的是开发板上使用的 12864 协议栈驱动程序 hal_lcd.c，这里也是和 TI 官方开发板唯一电路不同个地方。

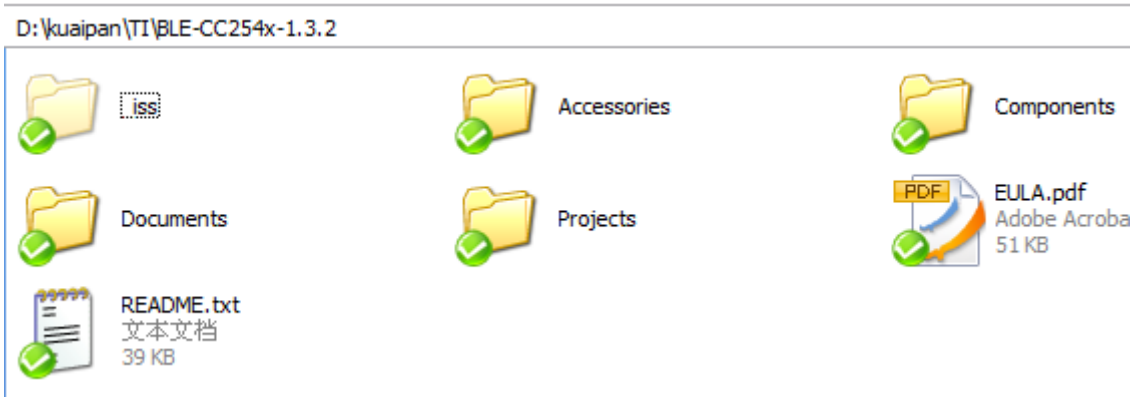
进入【版本 1.3.2】协议栈目录，解压 BLE-CC254x-1_3_2.zip，然后开始安装 BLE 协议栈。大家应该注意到还有一个 Hex 文件夹，这里存放的是协议栈编译好的 demo 例程，可以使用 flash programmer 直接烧写到开发板中运行。





其他一路 next。最后安装结束。

安装后得到如下的协议栈文件夹



目录详情

目录【Accessories】

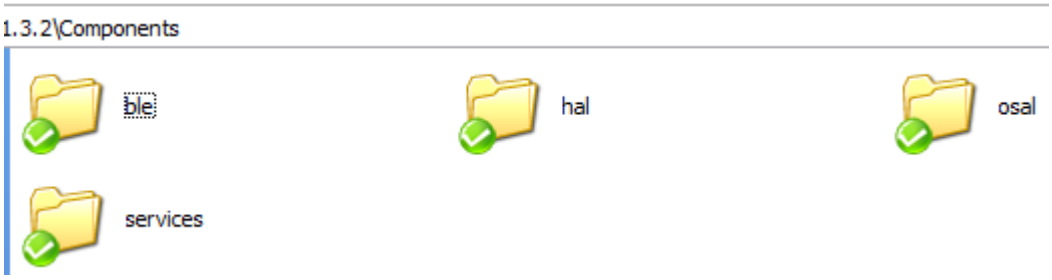
\Accessories\Drivers 里面存放的是烧写了 HostTestRelease 程序的 CC2540USBdongle 的 usb 转串口驱动程序，很多用户反应说 usbdongle 插到电脑上没有被识别成串口号，这里就要注意，usbdongle 出厂时烧写的是 packetsniffer 的固件，是协议分析仪，只有当 usbdongle 烧写了 HostTestRelease 程序时才会表现为一个串口，此时 usbdongle 的驱动程序即在 Drivers 目录下。

\Accessories\HexFiles 里面存放的是 TI 开发板上的预先编译的 hex 文件，由于我们的开发板更换了 lcd 型号，因此这里的部分 hex 虽然可以烧写到我们的 SmartRF 开发板上，但是 lcd 是不会显示的。替换驱动文件

我们的开发板当前版本只有 LCD 驱动与官方不同，因此需要替换我们提供的 lcd 驱动文件

目录【Components】

目录【Components】存放的是最终要的协议栈组件，包括底层的 ble，还有开发板硬件层 hal，还有类似操作系统的 osal。初学者只需要关注一下 hal 和 osal，这两个是与 ble 的应用程序密切相关的。



目录【Documents】

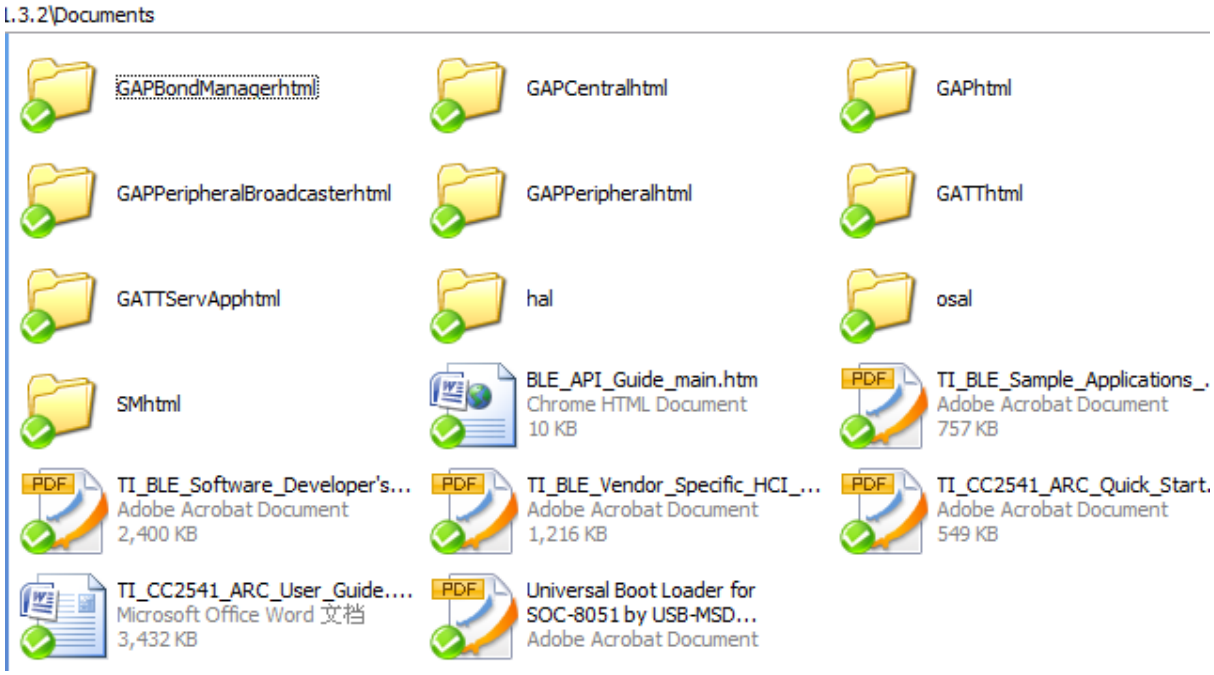
目录【Documents】存放的是 TI 提供的关于协议栈和协议栈 demo 的相关介绍和开发文档，因为该目录下的文件非常重要，虽然全部是英文，也请大家仔细的阅读。几个重要的文档如下：

《TI_BLE_Sample_Applications_Guide.pdf》协议栈 demo 操作指南，协议栈里所有 demo 的说明都在这里，值得参考。

《TI_BLE_Software_Developer's_Guide.pdf》ble 协议栈指南，介绍 ble 和 ti 的 ble 协议栈，必须要看的手册。

《BLE_API_Guide_main.htm》BLE API 文档，协议栈里调用的 api 函数还有调用时序，均

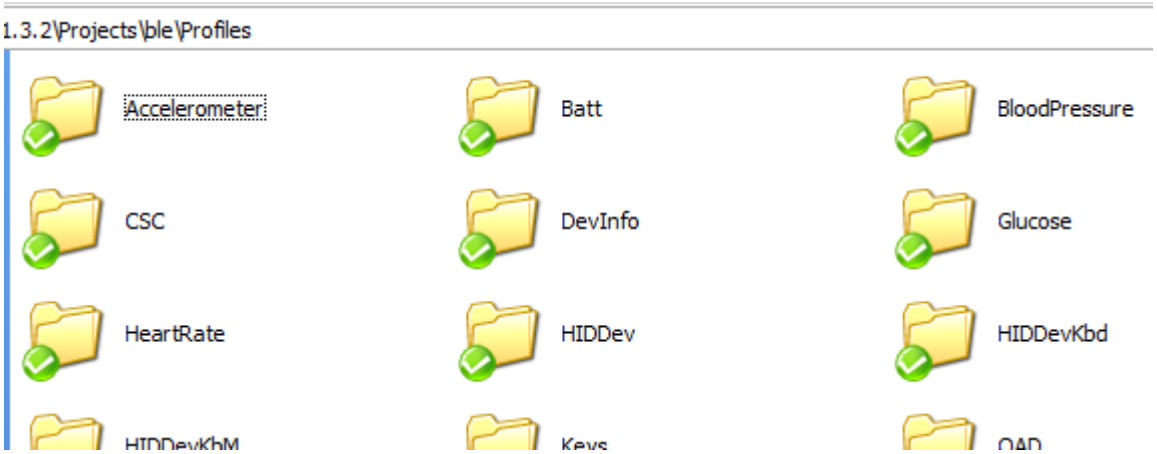
在此文档中，也是必须要阅读的手册。



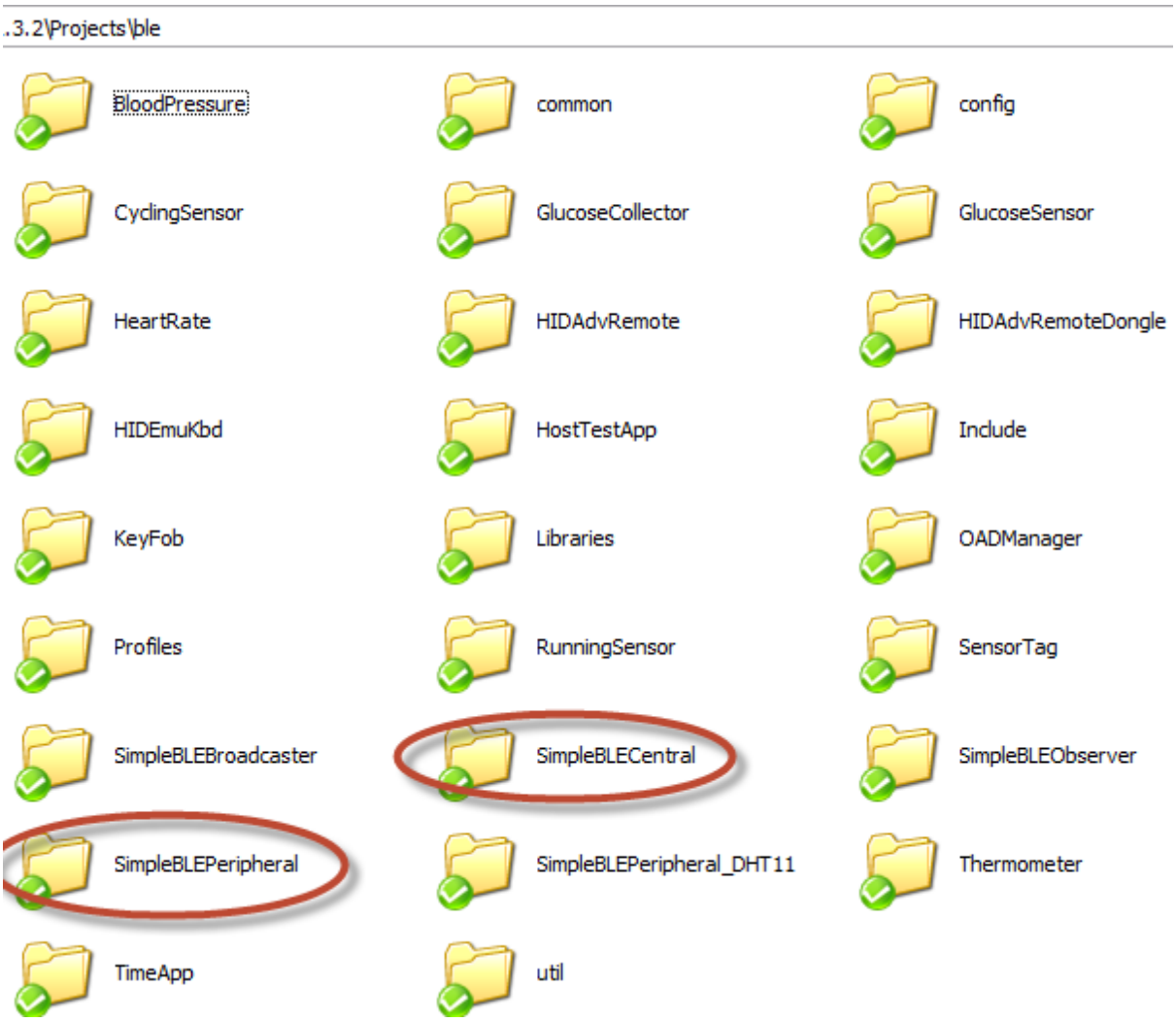
目录【Projects\ble】

目录【Projects\ble】,最后一个，也是最重要的目录，基于协议栈的 demo 工程都在这里。我们暂且只讨论我们将要做的两个实验：SimpleBLEPeripheral 从机和 SimpleBLECentral 主机，另外就是与这两个主机从机相关的【Profiles】文件夹下的【SimpleProfile】

我们需要注意的是，协议栈的每个例程都不是单独存在的，蓝牙是为了能够通信，想要通信就必须遵守一定的规则， Profile 就可以理解为相互约定的规则，因为每个协议栈 demo 都会有一个 Profile 与之对应，我们这里的 SimpleBLExxx 对应的就是 simpleGATTprofile，然后大多数书 profile 都输蓝牙组织 SIG 规定好的，看一下 Profiles 目录下的其他内容：



注意一下 SimpleBLECentral 和 SimpleBLEPeripheral 的位置，所有的协议栈 demo 都要放到 Projects/ble 这个目录下编译运行，因为 IAR 工程配置中使用的是相对路径，一旦 IAR 工程位置和整个协议栈源码的相对位置发生变化，就无法找到 ble 的其他组件，编译时会产生大量的无法找到文件的错误，请用户一定要注意，我们开发的协议栈 demo 均是只打包了 IAR 工程文件夹，必须要放到这里来编译。



替换驱动文件

我们的开发板当前版本只有 LCD 驱动与官方不同，因此需要替换我们提供的 lcd 驱动文件

替换 hal_lcd.c

我们的开发板使用新的 lcd12864 的 lcd 型号，因此需要替换协议栈 hal 层的 hal_lcd.c。

我们的 hal_lcd.c 文件目录在【CC254xEK\源码\BLE 协议栈\lcd12864 驱动】

协议栈中 hal_lcd.c 文件目录在【BLE-CC254x-x.x.x\Components\hal\target\CC2540EB】，用我们的 hal_lcd.c 替换该目录下的 hal_lcd.c 即可。

虽然有两种芯片 CC2540 和 CC2541，但是当 CC2541 配置为 SmartRF 开发板上运行时，连接的文件也是在 CC2540EB 目录下的。请注意。

打开协议栈程序（SimpleBLECentral 主机和 SimpleBLEPeripheral 从机）

使用 CC2540 的用户请在路径 BLE-CC254x-1.3.2\Projects\ble\SimpleBLEPeripheral\CC2540DB 下找到 SimpleBLEPeripheral.eww，如图 4-1 所示，打开该工程即可（注意后缀是 eww）。

使用 CC2541 的用户请在路径 BLE-CC254x-1.3.2\Projects\ble\SimpleBLEPeripheral\CC2541DB 下找到

SimpleBLEPeripheral.eww

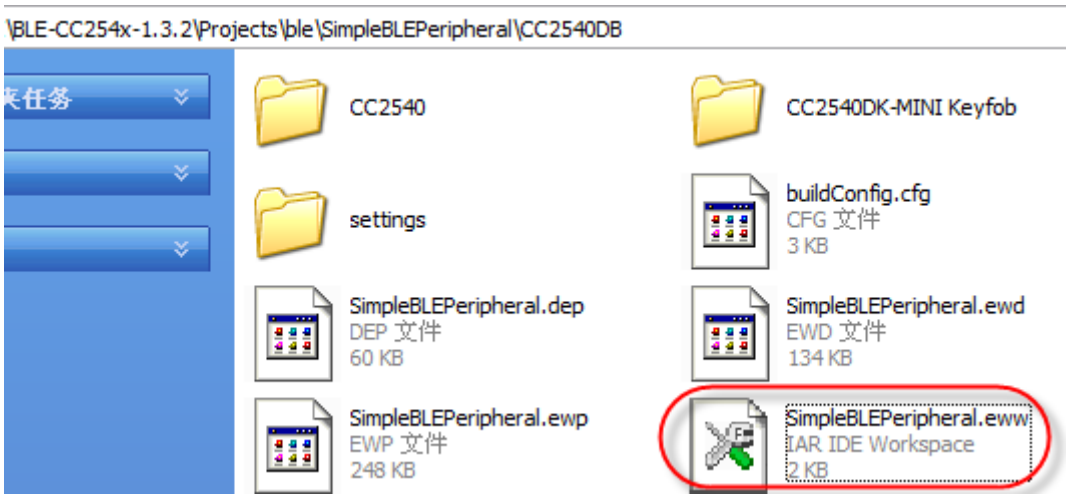


图 4-1 iar 工程文件

打开工程后，可以看到 SimpleBLEPeripheral 工程文件布局，如图 4-2 所示

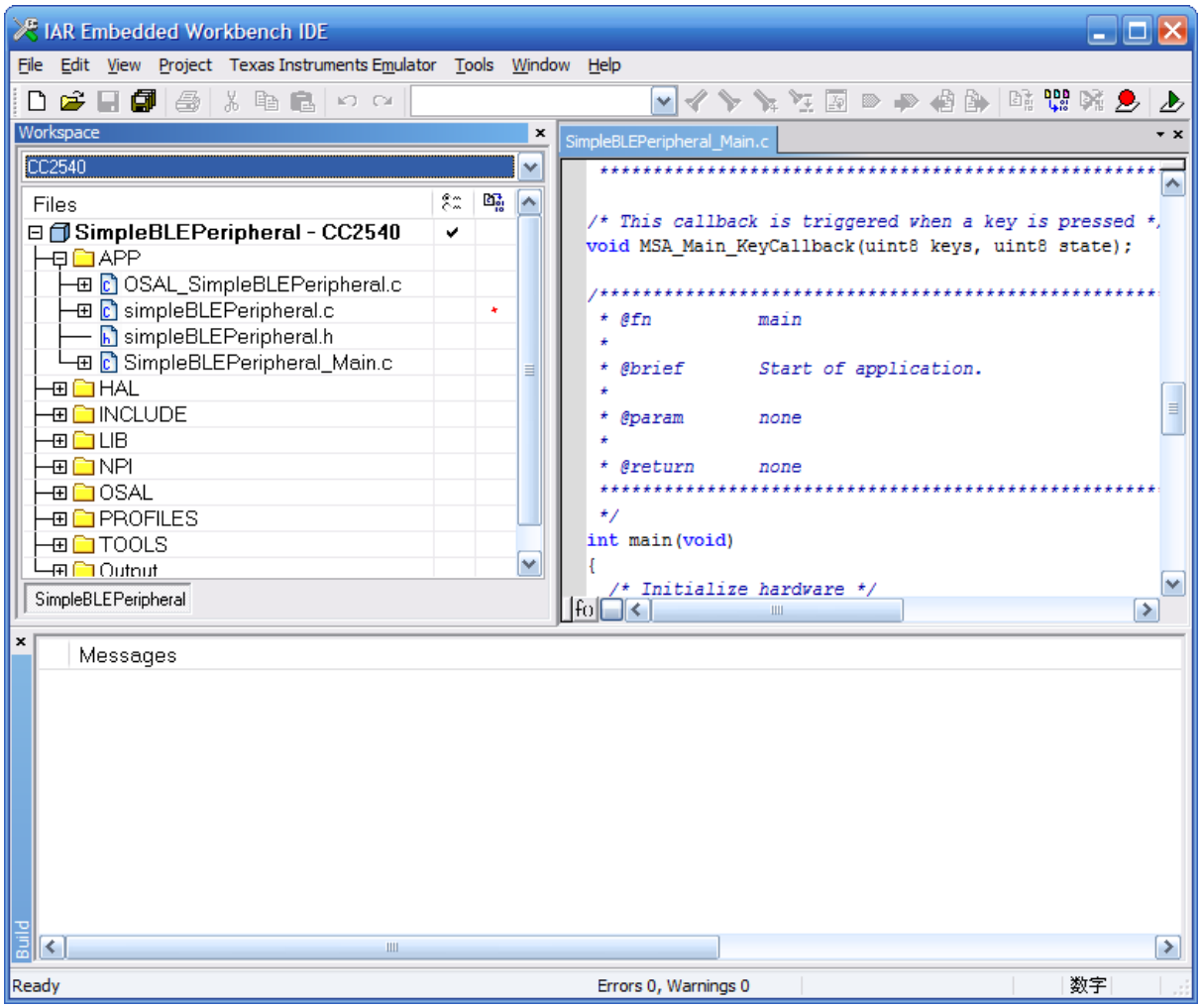
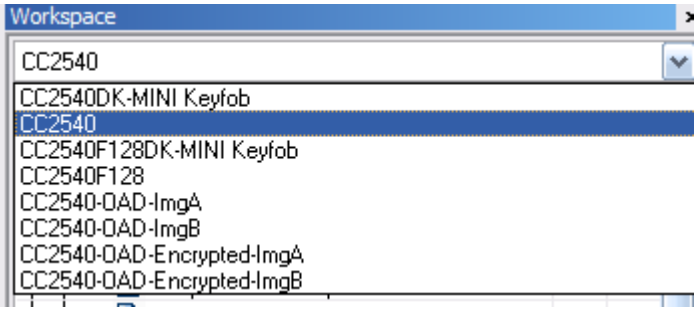


图 4-2 SimpleBLEPeripheral

这里值得注意的是在 workspace 下面的下拉列表：



当使用 SmartRF 系列开发板时需要选择 CC254x（我们均使用 256K 的芯片），而 CC254xDK-MINI Keyfob 则对应的是 Keyfob 开发板，在 keyfob 开发板里是没有 LCD 显示的，另外 Keyfob 配置的 SimpleBLEPeripheral 从机程序上电后不广播，需要按 S1 触发广播。而 SmartRF 配置的 SimpleBLEPeripheral 上电后默认广播。

在图 4-2 所示的文件布局中，左侧有很多文件夹，如 App、HAL、LIB 等，这些文件夹对应了 BLE 协议栈中不同的层，使用 BLE 协议栈进行应用程序的开发，一般之需要修改 App 目录下的文件即可。

4.2 BLE 协议栈基础实验：数据传输实验

尽管到此为止，读者对 BLE 协议的基本内容都不了解，甚至 BLE 协议栈是什么也可能存在诸多的疑问与不解，但是笔者也是从这些“困难”中走出来的，也理解此时读者的心情，与其阅读那“深奥”的 BLE 协议栈，不如通过一个数据传输实验来对 BLE 协议以及 BLE 协议栈建立一个形象、直观的认识，这将有助于读者对 BLE 协议的理解数据传输实验的基本功能：两个 BLE 节点进行点对点通信，BLE 主机向 BLE 从机发送一个字节，然后再把写入的字节读回来，以测试主从之间的通信。

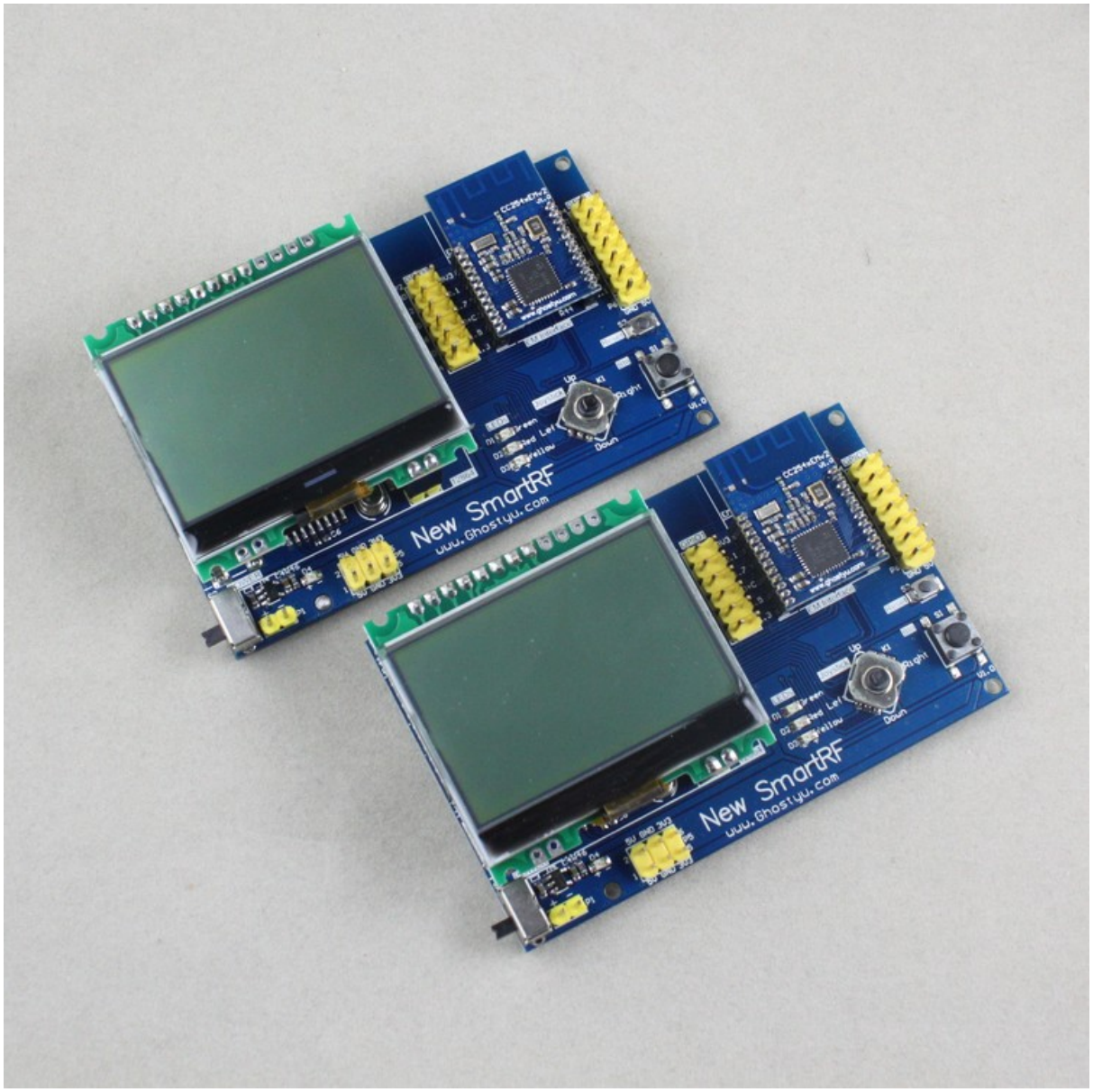


图 4-3

4.2.1 SimpleBLECentral 主机编程

在 Bluetooth-LE 低功耗蓝牙中有四种设备类型：Central 主机、Peripheral 从机、Observer 观察者、Broadcaster 广播者，通常 Central 和 Peripheral 一起使用，然后 Observer 和 Broadcaster 一起使用。

Central 和 Peripheral 连接后交换数据，我们平时使用到的基本上都是这个模式，而像多个温度采集器，通常采用 Observer 和 Broadcaster 无需连接的方式。

我们首先从最简单的主机 SimpleBLECentral 和从机 SimpleBLEPeripheral 开始。

打开 SimpleBLECentral 工程。

【Projects\ble\SimpleBLECentral\CC2540\SimpleBLECentral.eww】

在 APP 组下通常可以看到四个文件：

OSAL_xxx.c

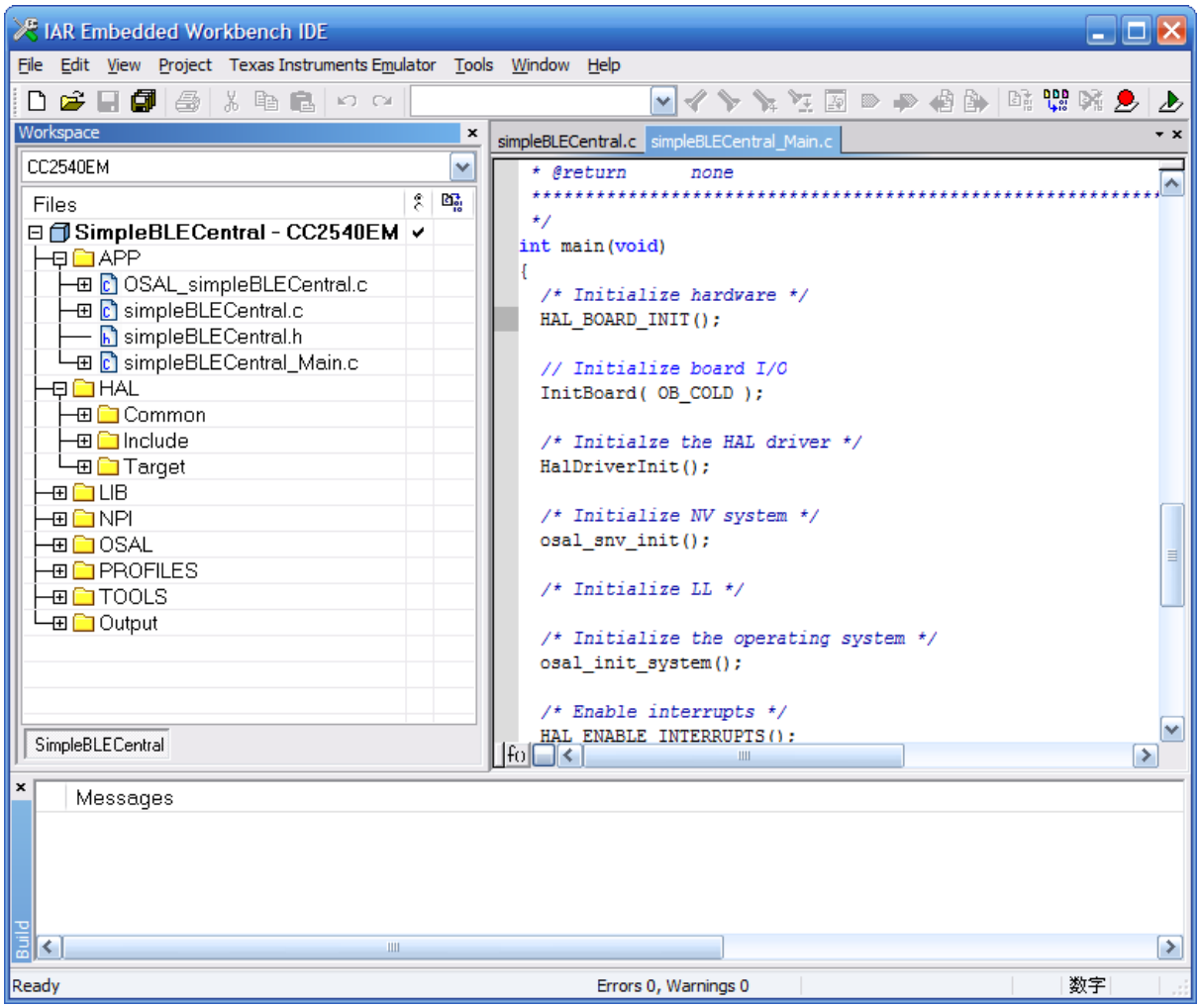
xxx.c

xxx.h

xxx_Main.c

OSAL_xxx.c 和 xxx_Main.c 是 OSAL 系统抽象层的一部分代码留给了用户自定义，比如用户任务函数的初始化还任务函数主体，这里的任务函数可以理解为操作系统里的一个进程。关于 osal，我们会在后面详细介绍。

而 xxx.c 和 xxx.h 是刚才提高的用户任务函数初始化和实现的主体。具体的代码均在这里，通常，整个协议栈里需要我们修改的也只有这两个文件。TI 已经帮我们做好了大部分工作。



simpleBLECentral.c 中的任务初始化函数

```
00266: void SimpleBLECentral_Init( uint8 task_id )
00267: {
00268:   simpleBLETaskId = task_id;
00269:
00270:   // Setup Central Profile
00271:   {
00272:     uint8 scanRes = DEFAULT_MAX_SCAN_RES;
00273:     GAPCentralRole_SetParameter ( GAPCENTRALROLE_MAX_SCAN_RES, sizeof( uint8 ), &scanRes );
00274:   }
00275:
00276:   // Setup GAP
00277:   GAP_SetParamValue( TGAP_GEN_DISC_SCAN, DEFAULT_SCAN_DURATION );
00278:   GAP_SetParamValue( TGAP_LIM_DISC_SCAN, DEFAULT_SCAN_DURATION );
00279:   GGS_SetParameter( GGS_DEVICE_NAME_ATT, GAP_DEVICE_NAME_LEN, (uint8 *) simpleBLEDeviceName );
00280:
00281:   // Setup the GAP Bond Manager
00282:   {
00283:     uint32 passkey = DEFAULT_PASSCODE;
00284:     uint8 pairMode = DEFAULT_PAIRING_MODE;
00285:     uint8 mitm = DEFAULT_MITM_MODE;
00286:     uint8 ioCap = DEFAULT_IO_CAPABILITIES;
00287:     uint8 bonding = DEFAULT_BONDING_MODE;
00288:     GAPBondMgr_SetParameter( GAPBOND_DEFAULT_PASSCODE, sizeof( uint32 ), &passkey );
00289:     GAPBondMgr_SetParameter( GAPBOND_PAIRING_MODE, sizeof( uint8 ), &pairMode );
00290:     GAPBondMgr_SetParameter( GAPBOND_MITM_PROTECTION, sizeof( uint8 ), &mitm );
00291:     GAPBondMgr_SetParameter( GAPBOND_IO_CAPABILITIES, sizeof( uint8 ), &ioCap );
00292:     GAPBondMgr_SetParameter( GAPBOND_BONDING_ENABLED, sizeof( uint8 ), &bonding );
00293:   }
00294:
00295:   // Initialize GATT Client
00296:   VOID GATT_InitClient();
00297:
00298:   // Register to receive incoming ATT Indications/Notifications
00299:   GATT_RegisterForInd( simpleBLETaskId );
00300:
00301:   // Initialize GATT attributes
00302:   GGS_AddService( GATT_ALL_SERVICES );          // GAP
00303:   GATTServApp_AddService( GATT_ALL_SERVICES ); // GATT attributes
00304:
00305:   // Register for all key events - This app will handle all key events
00306:   RegisterForKeys( simpleBLETaskId );
00307:
00308:   // makes sure LEDs are off
00309:   HalLedSet( (HAL_LED_1 | HAL_LED_2), HAL_LED_MODE_OFF );
00310:
00311:   // Setup a delayed profile startup
00312:   osal_set_event( simpleBLETaskId, START_DEVICE_EVT );
00313: } ? end SimpleBLECentral_Init ?
```

273 行：GAPCentralRole_SetParameter 函数，设置 GAPCENTRALROLE_MAX_SCAN_RES（最大的扫描回应）为 8 个（272 行宏定义为 8），也就是说如果广播的从机超过了 9 个，我们的 central 也只能扫描到先 RSP 的从机。

296 行：GATT_InitClientGATT 函数，初始化 GATT Client，GATT 有 Service 和 Client，Service 作为服务端，对 GATT Client 提供 read/write 接口，一般情况下，Central 作为 Client，Peripheral 作为 Service，所以主机 Central 会调用 GATT_WriteCharValue 或者 GATT_ReadCharValue 来和作为 Service 端的 Peripheral 从机通信，而 Peripheral 需要通过 notify 的方式，也就是调用 GATT_Notification 发起和主机的通信，在后面我们会详细介绍他们。

299 行，注册当前任务为 GATT 的 notify 和 indicate 的接收端。也就是说当从机 Peripheral 通过 GATT_Notification 发来数据时，当前的任务函数会接收到消息，如果不注册，则无法接收。

306 行：RegisterForKeys，注册按键服务，当开发板上的五向按键或者 S1 按键被按下时，会发送按键消息，只有注册了按键服务的任务 id 才会接收到该消息。

321 行：osal_set_event，该函数会启动一个事件，这里是 START_DEVICE_EVT，也就是任务函数的开始运行的地方，由此，会正式转入系统的运行。

simpleBLECentral.c 中的任务函数：

```
00328: uint16 SimpleBLECentral_ProcessEvent( uint8 task_id, uint16 events )
00329: {
00330:
00331:   VOID task_id; // OSAL required parameter that isn't used in this function
00332:
00333:   if ( events & SYS_EVENT_MSG )
00334:   {
00335:     uint8 *pMsg;
00336:
00337:     if ( (pMsg = osal_msg_receive( simpleBLETaskId )) != NULL )
00338:     {
00339:       simpleBLECentral_ProcessOSALMsg( (osal_event_hdr_t *)pMsg );
00340:
00341:       // Release the OSAL message
00342:       VOID osal_msg_deallocate( pMsg );
00343:     }
00344:
00345:     // return unprocessed events
00346:     return (events ^ SYS_EVENT_MSG);
00347:   }
00348:
00349:   if ( events & START_DEVICE_EVT )
00350:   {
00351:     // Start the Device
00352:     VOID GAPCentralRole_StartDevice( (gapCentralRoleCB_t *) &simpleBLERoleCB );
00353:
00354:     // Register with bond manager after starting device
00355:     GAPBondMgr_Register( (gapBondCBs_t *) &simpleBLEBondCB );
00356:
00357:     return ( events ^ START_DEVICE_EVT );
00358:   }
00359:
00360:   if ( events & START_DISCOVERY_EVT )
00361:   {
00362:     simpleBLECentralStartDiscovery( );
00363:
00364:     return ( events ^ START_DISCOVERY_EVT );
00365:   }
00366:
00367:   // Discard unknown events
00368:   return 0;
00369: } ? end SimpleBLECentral_ProcessEvent ?
```

如果觉得上面这个函数很陌生，不用着急，技术所有的任务函数都是这样的结构，在多接触后，会变得很简单。

333 行：SYS_EVENT_MSG 系统消息事件，当有按键触发，或者 ble 协议栈接收到数据后都会想任务函数发送系统事件，告知用户准备接收。

339 行：simpleBLECentral_ProcessOSALMsg 函数，处理系统消息的函数。

346 行：当处理完当前的消息后，一定要对此消息事件异或操作，也就是告诉系统，我当前的消息事件，我处理完了，可以处理下一条了。

349 行：START_DEVICE_EVT，这个很熟悉是吧，在初始化函数的最后，通过 osal_set_event 设置了该事件，告诉系统，所有的初始化工作都结束了，可以开始了。

352 行：GAPCentralRole_StartDevice，开始启动主机，并且传递了两个回调函数地址。分别是：

```
00234: // GAP Role Callbacks
00235: static const gapCentralRoleCB_t simpleBLERoleCB =
00236: {
00237:   simpleBLECentralRssiCB,       // RSSI callback
00238:   simpleBLECentralEventCB       // Event callback
00239: };
```

读 RSSI 值回调函数，和主机事件回调函数。

所谓回调函数，是会被其他系统自动调用的一个函数，我们可以在这个函数里接收

系统传递过来的参数。例如 simpleBLECentralRssiCB

```
00657: static void simpleBLECentralRssiCB( uint16 connHandle, int8 rssi )
00658: {
00659:     LCD_WRITE_STRING_VALUE( "RSSI -dB:", (uint8) (-rssi), 10, HAL_LCD_LINE_1 );
00660:     SerialPrintValue("RSSI -dB:", (uint8) (-rssi), 10);SerialPrintString("\r\n")
00661: }
```

系统通过参数 rssi 告诉用户，当前的 rssi 信号值，我们可以在这个函数里添加对信号值的处理，比如输出到 LCD、或者串口（输出到串口是我们添加）。

simpleBLECentralEventCB 回调函数的处理则比较复杂，这个函数用用来通知用户当前的主机状态，然后由用户决定下一步如何操作。例如下面的 676 行：告知用户，主机已经初始化完毕，这时我么就可以通过 lcd 显示当前为 ble central 主机了。700 行：告诉用户，发现了 ble 从机。

```
00672: static void simpleBLECentralEventCB( gapCentralRoleEvent_t *pEvent )
00673: {
00674:   switch ( pEvent->gap.opcode )
00675:   {
00676:     case GAP_DEVICE_INIT_DONE_EVENT:
00677:       {
00678:         LCD_WRITE_STRING( "BLE Central", HAL_LCD_LINE_1 );
00679:         SerialPrintString("BLE Central: ");
00680:         LCD_WRITE_STRING( bdAddr2Str( pEvent->initDone.devAddr ),  HAL_LCD_LINE_2 );
00681:         SerialPrintString((uint8*)bdAddr2Str( pEvent->initDone.devAddr ));SerialPrintString("\r\n");
00682:       }
00683:       break;
00684:
00685:     case GAP_DEVICE_INFO_EVENT:
00686:       {
00687:         // if filtering device discovery results based on service UUID
00688:         if ( DEFAULT_DEV_DISC_BY_SVC_UUID == TRUE )
00689:         {
00690:           if ( simpleBLEFindSvcUuid( SIMPLEPROFILE_SERV_UUID,
00691:                                      pEvent->deviceInfo.pEvtData,
00692:                                      pEvent->deviceInfo.dataLen ) )
00693:           {
00694:             simpleBLEAddDeviceInfo( pEvent->deviceInfo.addr, pEvent->deviceInfo.addrType );
00695:           }
00696:         }
00697:       }
00698:       break;
00699:
00700:     case GAP_DEVICE_DISCOVERY_EVENT:
00701:       {
00702:         // discovery complete
00703:         simpleBLEScanning = FALSE;
00704:
00705:         // if not filtering device discovery results based on service UUID
00706:         if ( DEFAULT_DEV_DISC_BY_SVC_UUID == FALSE )
00707:         {
00708:           // Copy results
00709:           simpleBLEScanRes = pEvent->discCmpl.numDevs;
00710:           osal_memcpy( simpleBLEDevList, pEvent->discCmpl.pDevList,
```

继续我们刚才的代码分析。

369 行：START_DISCOVERY_EVT，主机扫描从机 Service。通过调用下一行的 simpleBLECentralStartDiscovery 函数，开始扫描 ble 从机的 Service。该事件是主机发起连接时，如果还未发现从机 service 时会调用。

系统消息处理函数

```
00389: static void simpleBLECentral_ProcessOSALMsg( osal_event_hdr_t *pMsg )
00390: {
00391:   switch ( pMsg->event )
00392:   {
00393:     case KEY_CHANGE:
00394:       simpleBLECentral_HandleKeys( ((keyChange_t *)pMsg)->state, ((keyChange_t *)pMsg)->keys );
00395:       break;
00396:
00397:     case GATT_MSG_EVENT:
00398:       simpleBLECentralProcessGATTMsg( (gattMsgEvent_t *) pMsg );
00399:       break;
00400:   }
00401: }
```

在 SYS_EVENT_MSG 下的这个函数用来专门处理系统消息，之所以放到一个独立的函数里，是因为要处理的内容是在太多了，仅仅按键处理就一大堆。

393 行：KEY_CHANGE，按键消息，有按键按下时会向注册了按键服务的任务函数发送该消息，我们进入 simpleBLECentral_HandleKeys 进一步分析主机 Central 的有哪些按键处理。目前我么只看一下 UP 按键具体都有哪些操作。

```
00411:   if ( keys & HAL_KEY_UP )
00412:   {
00413:     // Start or stop discovery
00414:     if ( simpleBLEState != BLE_STATE_CONNECTED )
00415:     {
00416:       if ( !simpleBLEScanning )
00417:       {
00418:         simpleBLEScanning = TRUE;
00419:         simpleBLEScanRes = 0;
00420:
00421:         LCD_WRITE_STRING( "Discovering...", HAL_LCD_LINE_1 );
00422:         LCD_WRITE_STRING( "", HAL_LCD_LINE_2 );
00423:
00424:         GAPCentralRole_StartDiscovery( DEFAULT_DISCOVERY_MODE,
00425:                                        DEFAULT_DISCOVERY_ACTIVE_SCAN,
00426:                                        DEFAULT_DISCOVERY_WHITE_LIST );
00427:       }
00428:       else
00429:       {
00430:         GAPCentralRole_CancelDiscovery();
00431:       }
00432:     }
00433:     else if ( simpleBLEState == BLE_STATE_CONNECTED &&
00434:               simpleBLECharHdl != 0 &&
00435:               simpleBLEProcedureInProgress == FALSE )
00436:     {
00437:       uint8 status;
00438:
00439:       // Do a read or write as long as no other read or write is in progress
00440:       if ( simpleBLEDoWrite )
00441:       {
00442:         // Do a write
00443:         attWriteReq_t req;
00444:
00445:         req.handle = simpleBLECharHdl;
00446:         req.len = 1;
00447:         req.value[0] = simpleBLECharVal;
00448:         req.sig = 0;
00449:         req.cmd = 0;
00450:         status = GATT_WriteCharValue( simpleBLEConnHandle, &req, simpleBLETaskId );
00451:       }
00452:       else
00453:       {
00454:         // Do a read
00455:         attReadReq_t req;
00456:
00457:         req.handle = simpleBLECharHdl;
00458:         status = GATT_ReadCharValue( simpleBLEConnHandle, &req, simpleBLETaskId );
00459:       }
00460:
```

411 行：HAL_KEY_UP，判断是否是 UP 键被按键，当 up 键被按下时，如果还没有主从机还未连接，会调用 424 行的 GAPCentralRole_StartDiscovery 函数开始扫描从机，如果当前正在扫描，再按 UP 键时，会调用 430 行的 GAPCentralRole_CancelDiscovery，取消扫描并立即返回，注意，蓝牙 4.0 的相应非常快，我们连续按两次 up 会立刻返回搜索到的从机，无需等待。

433 行：如果当前主从机已连接时按下的 UP 键，程序首先会调用 GATT_WriteCharValue，向从机的 char1 写入一个字节。还记得我们在前面有说过，从机一般作为 GATT 的 Service，主机一般作为 GATT 的 client，client 可以调用 Write（就是这里的 GATT_WriteCharValue）或者 Read（也就是 458 行的 GATT_ReadCharValue）来发起和从机的通信

刚才提到的 GATT_WriteCharValue 和 GATT_ReadCharValue 是想 GATT 递交 read 和

write 请求，最后是否成功执行，会通过一个系统消息告知用户，这个系统消息就是刚在在 SYS_EVENT_MSG 里的其中一个，397 行的 GATT_MSG_EVENT，也就是在 simpleBLECentralProcessGATTMsg 处理各种请求后结果。如下代码。

```
00586: static void simpleBLECentralProcessGATTMsg( gattMsgEvent_t *pMsg )
00587: {
00588:   if ( simpleBLEState != BLE_STATE_CONNECTED )
00589:   {
00590:     // In case a GATT message came after a connection has dropped,
00591:     // ignore the message
00592:     return;
00593:   }
00594:
00595:   if ( ( pMsg->method == ATT_READ_RSP ) ||
00596:        ( ( pMsg->method == ATT_ERROR_RSP ) &&
00597:          ( pMsg->msg.errorRsp.reqOpcode == ATT_READ_REQ ) ) )
00598:   {
00599:     if ( pMsg->method == ATT_ERROR_RSP )
00600:     {
00601:       uint8 status = pMsg->msg.errorRsp.errCode;
00602:
00603:       LCD_WRITE_STRING_VALUE( "Read Error", status, 10, HAL_LCD_LINE_1 );
00604:       SerialPrintValue("Read Error", status, 10);SerialPrintString("\r\n");
00605:     }
00606:     else
00607:     {
00608:       // After a successful read, display the read value
00609:       uint8 valueRead = pMsg->msg.readRsp.value[0];
00610:
00611:       LCD_WRITE_STRING_VALUE( "Read rsp:", valueRead, 10, HAL_LCD_LINE_1 );
00612:       SerialPrintValue("Read rsp:", valueRead, 10);SerialPrintString("\r\n");
00613:     }
00614:
00615:     simpleBLEProcedureInProgress = FALSE;
00616:   }
00617:   else if ( ( pMsg->method == ATT_WRITE_RSP ) ||
00618:        ( ( pMsg->method == ATT_ERROR_RSP ) &&
00619:          ( pMsg->msg.errorRsp.reqOpcode == ATT_WRITE_REQ ) ) )
00620:   {
00621:
00622:     if ( pMsg->method == ATT_ERROR_RSP == ATT_ERROR_RSP )
00623:     {
00624:       uint8 status = pMsg->msg.errorRsp.errCode;
00625:
00626:       LCD_WRITE_STRING_VALUE( "Write Error", status, 10, HAL_LCD_LINE_1 );
00627:       SerialPrintValue( "Write Error", status, 10);SerialPrintString("\r\n");
00628:     }
00629:     else
00630:     {
00631:       // After a succesful write, display the value that was written and incr
00632:       uint8 temp=simpleBLECharVal;
```

595 行：判断消息类型是否是 ATT_READ_RSP，或者是错误操作为 ATT_READ_REQ。如果读请求失败，会进入 ATT_ERROR_RSP 中，然后我们可以做一些出错管理，比如通过交互系统输出等等。如果读请求成功，会返回读取到的值，如 609 行，会将 value 保存到内存中，待使用，大家注意，为什么是 value[0]，一个数组的第一个字节，这是因为 SimpleBLEPeripheral 从机中使用的 simpleProfile 中的 char1，只定义了一个字节的可读可写的 characteristic 特征值。

617 行：判断消息类型是否是 ATT_WRITE_RSP，或者是错误操作为 ATT_WRITE_REQ。如果写请求失败，会进入 att_write_req 这个 att_error_rsp，和上面的 ATT_READ_RSP 类似。

以上是 SimpleBLECentral 主机代码的简单分析。下面我们继续 SimpleBLEPeripheral 从机代码段分析

4.2.2 SimpleBLEPeripheral 从机编程

在 TI 的 ble 协议栈中，虽然主机和从机的主体结构类似（均基于 osal），但是从机和主机有着很大的区别，从机里包含了一个叫做 profile 的相关代码，这个 profile 决定了从机的功能。例如防丢器、血压仪、心率计等均是蓝牙组织规定的 profile。

打开 SimpleBLEPeripheral 从机工程。

【Projects\ble\SimpleBLECentral\CC2540\SimpleBLEPeripheral.eww】

与主机程序结构类似，我们直接进入 simpleBLEPeripheral.c 主体源文件。

SimpleBLEPeripheral_Init 任务初始化函数。

```
00281: void SimpleBLEPeripheral_Init( uint8 task_id )
00282: {
00283:   simpleBLEPeripheral_TaskID = task_id;
00284:
00285:   // Setup the GAP
00286:   VOID GAP_SetParamValue( TGAP_CONN_PAUSE_PERIPHERAL, DEFAULT_CONN_PAUSE_PERIPHERAL );
00287:
00288:   // Setup the GAP Peripheral Role Profile
00289:   {
00290:     #if defined( CC2540_MINIDK )
00291:       // For the CC2540DK-MINI keyfob, device doesn't start advertising until button is pressed
00292:       uint8 initial_advertising_enable = FALSE;
00293:     #else
00294:       // For other hardware platforms, device starts advertising upon initialization
00295:       uint8 initial_advertising_enable = TRUE;
00296:     #endif
00297:
00298:     // By setting this to zero, the device will go into the waiting state after
00299:     // being discoverable for 30.72 second, and will not being advertising again
00300:     // until the enabler is set back to TRUE
00301:     uint16 gapRole_AdvertOffTime = 0;
00302:
00303:     uint8 enable_update_request = DEFAULT_ENABLE_UPDATE_REQUEST;
00304:     uint16 desired_min_interval = DEFAULT_DESIRED_MIN_CONN_INTERVAL;
00305:     uint16 desired_max_interval = DEFAULT_DESIRED_MAX_CONN_INTERVAL;
00306:     uint16 desired_slave_latency = DEFAULT_DESIRED_SLAVE_LATENCY;
00307:     uint16 desired_conn_timeout = DEFAULT_DESIRED_CONN_TIMEOUT;
00308:
00309:     // Set the GAP Role Parameters
00310:     GAPRole_SetParameter( GAPROLE_ADVERT_ENABLED, sizeof( uint8 ), &initial_advertising_enable );
00311:     GAPRole_SetParameter( GAPROLE_ADVERT_OFF_TIME, sizeof( uint16 ), &gapRole_AdvertOffTime );
00312:
00313:     GAPRole_SetParameter( GAPROLE_SCAN_RSP_DATA, sizeof ( scanRspData ), scanRspData );
00314:     GAPRole_SetParameter( GAPROLE_ADVERT_DATA, sizeof( advertData ), advertData );
00315:
00316:     GAPRole_SetParameter( GAPROLE_PARAM_UPDATE_ENABLE, sizeof( uint8 ), &enable_update_request );
00317:     GAPRole_SetParameter( GAPROLE_MIN_CONN_INTERVAL, sizeof( uint16 ), &desired_min_interval );
00318:     GAPRole_SetParameter( GAPROLE_MAX_CONN_INTERVAL, sizeof( uint16 ), &desired_max_interval );
00319:     GAPRole_SetParameter( GAPROLE_SLAVE_LATENCY, sizeof( uint16 ), &desired_slave_latency );
00320:     GAPRole_SetParameter( GAPROLE_TIMEOUT_MULTIPLIER, sizeof( uint16 ), &desired_conn_timeout );
```

从机的 init 函数有着更多的设置。

314 行：设置从机广播数据。

313 行：设置主机扫描回应数据。

主机和从机是这样开机工作的：从机开启广播，然后主机扫描广播的从机，当从机接收到主机的扫描请求后，会向主机发送扫描回应数据。然后主机发起链接，然后开始通信。这里设计的广播内容和扫描回应内容就在这里设置。具体的内容分析在后面有介绍。

310~320 行：设置 BLE 低功耗蓝牙系统里几个非常重要的时间参数。

```
00350:   // Initialize GATT attributes
00351:   GGS_AddService( GATT_ALL_SERVICES );            // GAP
00352:   GATTServApp_AddService( GATT_ALL_SERVICES );    // GATT attributes
00353:   DevInfo_AddService();                           // Device Information Service
00354:   SimpleProfile_AddService( GATT_ALL_SERVICES );  // Simple GATT Profile
00355: #if defined FEATURE_OAD
00356:   VOID OADTarget_AddService();                    // OAD Profile
00357: #endif
00358:
00359:   // Setup the SimpleProfile Characteristic Values
00360:   {
00361:     uint8 charValue1 = 1;
00362:     uint8 charValue2 = 2;
00363:     uint8 charValue3 = 3;
00364:     uint8 charValue4 = 4;
00365:     uint8 charValue5[SIMPLEPROFILE_CHAR5_LEN] = { 1, 2, 3, 4, 5 };
00366:     SimpleProfile_SetParameter( SIMPLEPROFILE_CHAR1, sizeof ( uint8 ), &charValue1 );
00367:     SimpleProfile_SetParameter( SIMPLEPROFILE_CHAR2, sizeof ( uint8 ), &charValue2 );
00368:     SimpleProfile_SetParameter( SIMPLEPROFILE_CHAR3, sizeof ( uint8 ), &charValue3 );
00369:     SimpleProfile_SetParameter( SIMPLEPROFILE_CHAR4, sizeof ( uint8 ), &charValue4 );
00370:     SimpleProfile_SetParameter( SIMPLEPROFILE_CHAR5, SIMPLEPROFILE_CHAR5_LEN, charValue5 );
00371:   }
```

354 行：添加 SimpleProfile。

366~370 行：设置 SimpleProfile 初始数据。

```
00432:   // Setup a delayed profile startup
00433:   osal_set_event( simpleBLEPeripheral_TaskID, SBP_START_DEVICE_EVT );
00434:
00435: } ? end SimpleBLEPeripheral_Init ?
```

最后 433 行：启动 ble 从机。开始进入任务函数循环。

```
00450: uint16 SimpleBLEPeripheral_ProcessEvent( uint8 task_id, uint16 events )
00451: {
00452:
00453:   VOID task_id; // OSAL required parameter that isn't used in this function
00454:
00455:   if ( events & SYS_EVENT_MSG )
00456:   {
00457:     uint8 *pMsg;
00458:
00459:     if ( (pMsg = osal_msg_receive( simpleBLEPeripheral_TaskID )) != NULL )
00460:     {
00461:       simpleBLEPeripheral_ProcessOSALMsg( (osal_event_hdr_t *)pMsg );
00462:
00463:       // Release the OSAL message
00464:       VOID osal_msg_deallocate( pMsg );
00465:     }
00466:
00467:     // return unprocessed events
00468:     return (events ^ SYS_EVENT_MSG);
00469:   }
00470:
00471:   if ( events & SBP_START_DEVICE_EVT )
00472:   {
00473:     // Start the Device
00474:     VOID GAPRole_StartDevice( &simpleBLEPeripheral_PeripheralCBs );
00475:
00476:     // Start Bond Manager
00477:     VOID GAPBondMgr_Register( &simpleBLEPeripheral_BondMgrCBs );
00478:
00479:     // Set timer for first periodic event
00480:     osal_start_timerEx( simpleBLEPeripheral_TaskID, SBP_PERIODIC_EVT, SBP_PERIODIC_EVT_PERIOD );
00481:
00482:     return ( events ^ SBP_START_DEVICE_EVT );
00483:   }
00484:
00485:   if ( events & SBP_PERIODIC_EVT )
00486:   {
00487:     // Restart timer
00488:     if ( SBP_PERIODIC_EVT_PERIOD )
00489:     {
00490:       osal_start_timerEx( simpleBLEPeripheral_TaskID, SBP_PERIODIC_EVT, SBP_PERIODIC_EVT_PERIOD );
00491:     }
00492:
00493:     // Perform periodic application task
00494:     performPeriodicTask();
00495:
00496:     return (events ^ SBP_PERIODIC_EVT);
00497:   }
```

可以看出，所有的任务函数的结构几乎都是一致的，有一个任务入口点，还有系统消息事件处理，以及其他的任务事件处理。

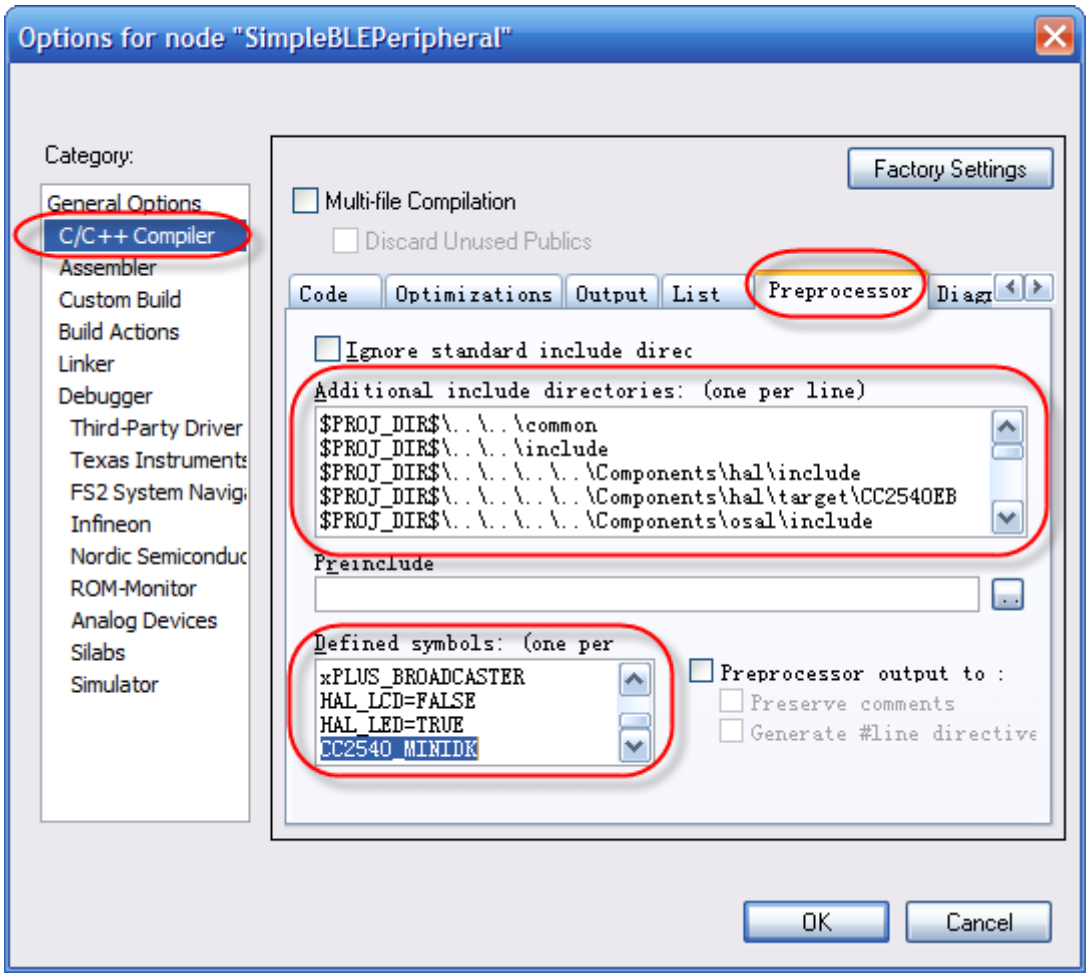
455 行：系统消息事件，同样包括按键消息事件，以及从机的当前请求状态回复。

471 行：由 init 函数启动的任务函数入口点，启动从机程序，并且开启周期性的任务处理，这个周期性的任务并不是必须的，这里是作为一个简单地 notify 示例。

系统消息处理函数。

```
00523: static void simpleBLEPeripheral_ProcessOSALMsg( osal_event_hdr_t *pMsg )
00524: {
00525:   switch ( pMsg->event )
00526:   {
00527:   #if defined( CC2540_MINIDK )
00528:     case KEY_CHANGE:
00529:       simpleBLEPeripheral_HandleKeys( ((keyChange_t *)pMsg)->state, ((keyChange_t *)pMsg)->keys
00530:       break;
00531:   #endif // #if defined( CC2540_MINIDK )
00532:
00533:   default:
00534:     // do nothing
00535:     break;
00536:   }
00537: }
```

从这个函数内容可以看到，在按键的处理前有一个宏定义，CC2540_MINIDK，当使用 keyfob 开发板运行该从机程序时，会有按键处理。当使用 SmartRF 开发板时不做任何处理。这个宏定义在工程的 Options 里的 Preprocessor 中定义。如下图，该图信息量很大，请注意看，未来会多次涉及到这里。这个 Options 对话框可以通过菜单 Project/Options 打开。



继续我们的分析。

留心的朋友可能发现了一个问题，在刚才 central 主机代码里 SYS_EVENT_MSG 下又一个 GATT_MSG_EVENT，当主机调用 GATT_ReadCharValue 后，读取到的 value，会在 GATT_MSG_EVENT 中通知我们，如下图

```
00380: static void simpleBLECentral_ProcessOSALMsg( osal_eve
00381: {
00382:   switch ( pMsg->event )
00383:   {
00384:     case KEY_CHANGE:
00385:       simpleBLECentral_HandleKeys( ((keyChange_t *)pMsg)->state,
00386:       break;
00387:
00388:     case GATT_MSG_EVENT:
00389:       simpleBLECentralProcessGATTMsg( (gattMsgEvent_t *) pMsg );
00390:       break;
00391:   }
00392: }
```

主机中

但是 Peripheral 从机程序里似乎没有这个消息的处理，那么主机发送过来的数据，从机在哪里接收呢？

```
00523: static void simpleBLEPeripheral_ProcessOSALMsg
00524: {
00525:   switch ( pMsg->event )
00526:   {
00527:   #if defined( CC2540_MINIDK )
00528:     case KEY_CHANGE:
00529:       simpleBLEPeripheral_HandleKeys( ((keyChange_t *)pMsg
00530:       break;
00531:   #endif // #if defined( CC2540_MINIDK )
00532:
00533:   default:
00534:     // do nothing
00535:     break;
00536:   }
00537: }
```

从机中

刚才我们有提到，这里的 Peripheral 是作为 GATT 的 service 端，而主机是作为 GATT 的 client 端，两者在数据的通信接口上有很大的区别。

在从机里，接收数据是通过一个 GATT Callback 回调函数，系统在接收到数据时会调用这个 callback 向我们发出通知。在 simpleBLEPeripheral.c 的开头有这个回调函数的定义。如下图：

```
00257: // Simple GATT Profile Callbacks
00258: static simpleProfileCBs_t simpleBLEPeripheral_SimpleProfileCBs =
00259: {
00260:   simpleProfileChangeCB    // Charactersitic value change callback
00261: };
```

每当 profile 中的 characteristic value 有变化，都会产生一次回调。在回调函数中，我们判断是哪个 characteristic，然后准备数据接收，这样就实现了主机到从机的数据接收工作。下面的 simpleProfileChangeCB 是 peripheral 从机中的 simpleProfile 回调函数。

```
00752: static void simpleProfileChangeCB( uint8 paramID )
00753: {
00754:   uint8 newValue;
00755:
00756:   switch( paramID )
00757:   {
00758:     case SIMPLEPROFILE_CHAR1:
00759:       SimpleProfile_GetParameter( SIMPLEPROFILE_CHAR1, &newValue );
00760:
00761:       #if (defined HAL_LCD) && (HAL_LCD == TRUE)
00762:         HalLcdWriteStringValue( "Char 1:", (uint16)(newValue), 10,
00763:       #endif // (defined HAL_LCD) && (HAL_LCD == TRUE)
00764:
00765:       break;
00766:
00767:     case SIMPLEPROFILE_CHAR3:
00768:       SimpleProfile_GetParameter( SIMPLEPROFILE_CHAR3, &newValue );
00769:
00770:       #if (defined HAL_LCD) && (HAL_LCD == TRUE)
00771:         HalLcdWriteStringValue( "Char 3:", (uint16)(newValue), 10,
00772:       #endif // (defined HAL_LCD) && (HAL_LCD == TRUE)
00773:
00774:       break;
00775:
00776:     default:
00777:       // should not reach here!
00778:       break;
```

758 行：判断时候是 Characteristic 1，如果是，将 characteristic1 的 value 复制到 newValue 中，然后可以通过 lcd 输出。待会我们在 SimpleBLECentral 和 SimpleBLEPeripheral 的通信实验中会看到这个现象。

4.2.3 Central 和 Peripheral 从机通信测试

分别将编译 SimpleBLECentral 和 SimpleBLEPeripheral，然后下载到 SmartRF 开发板中。

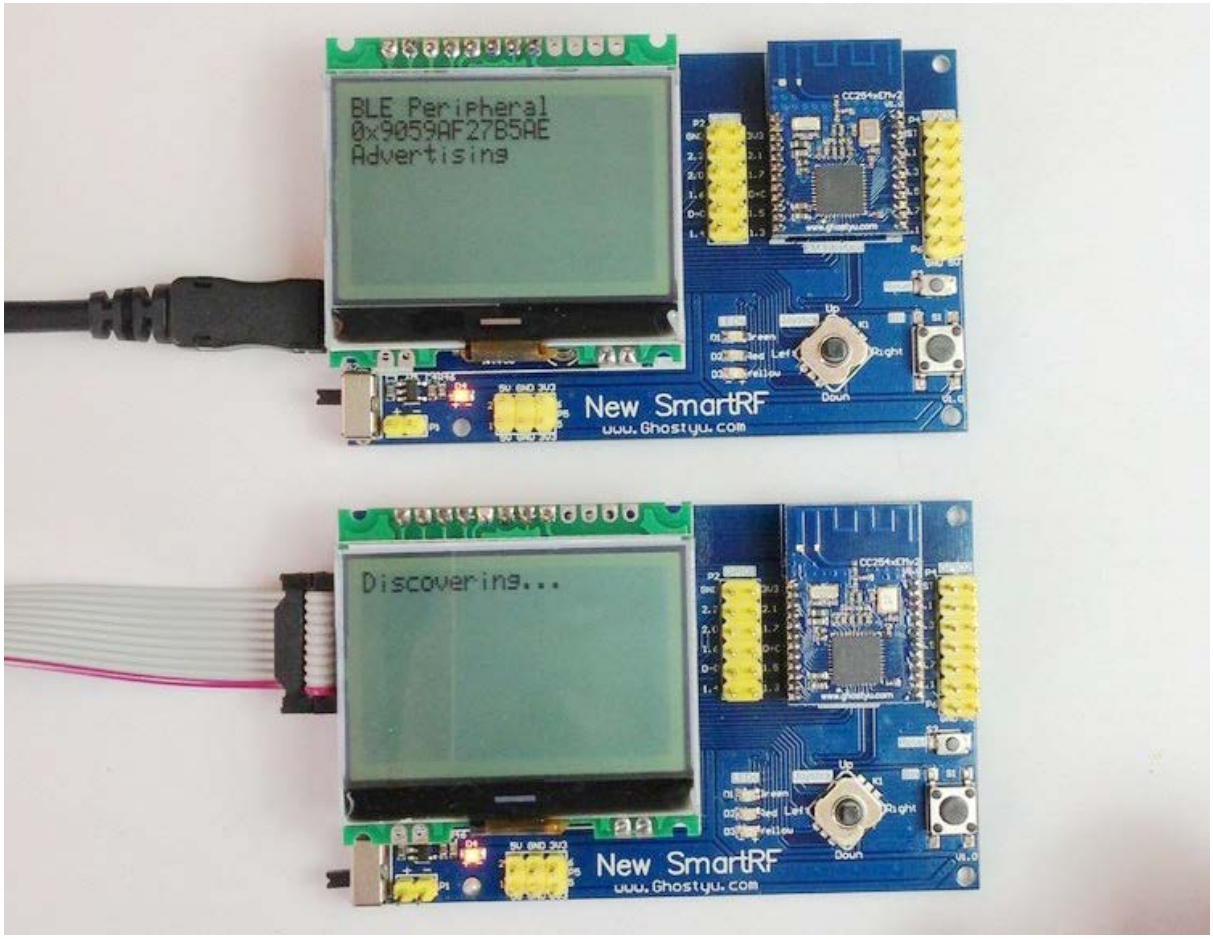
我们的 New SmartRF 开发板可以使用仿真器供电、USB 供电、外接锂电池供电。选择任意一种即可。

4.2.3.1 开机

上一节我们程序分析时讲过，如果设备初始化成功，然后会启动各自的角色，并且在 LCD 上显示相关信息，SimpleBLECentral 和 SimpleBLEPeripheral 开始后如下图所示：

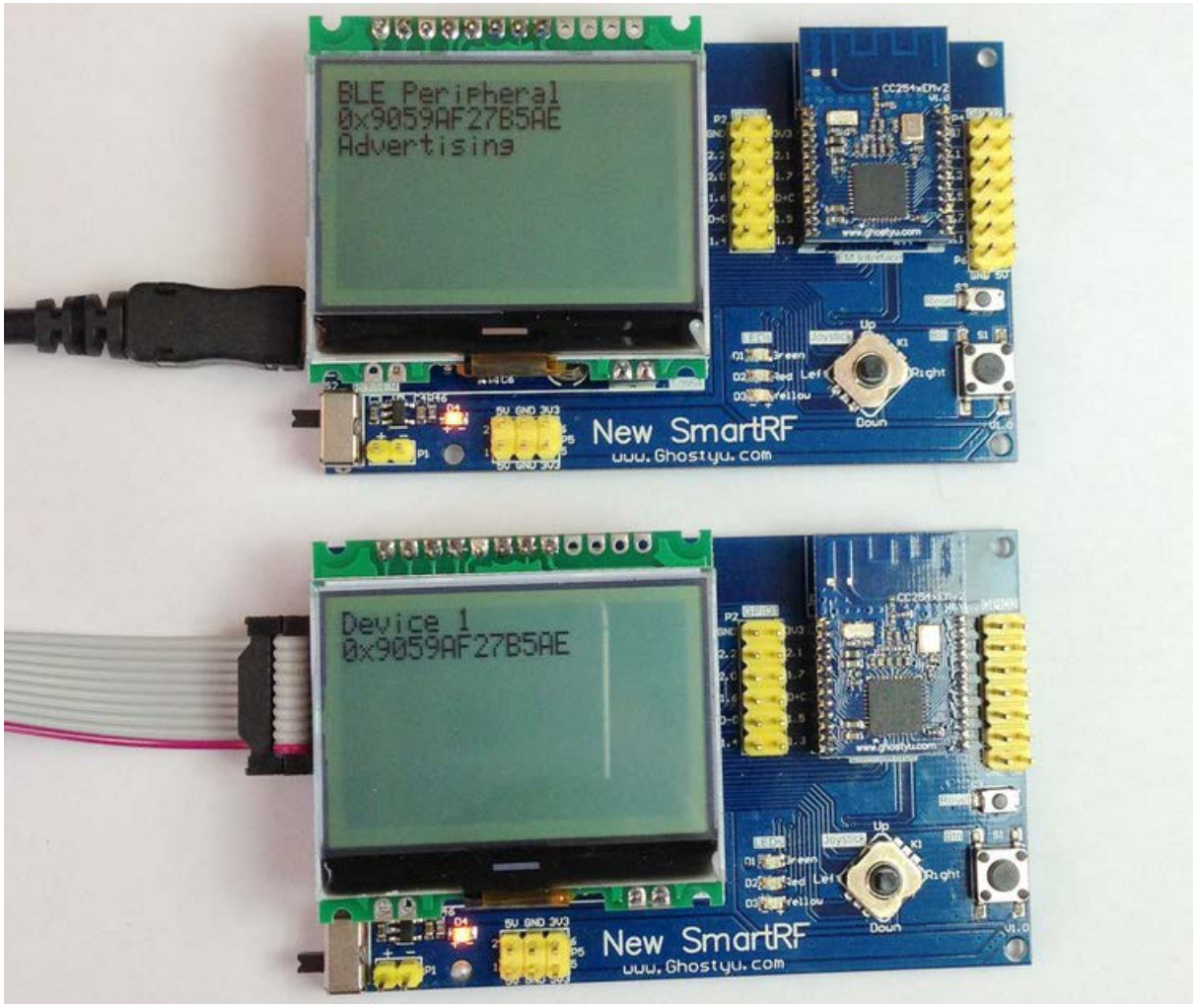
4.2.3.2 搜索从机

按下 SmartRF 开发板的 Joystick UP 按键，开始搜索从机，等待一会，会返回搜索到的从机（若不想等待，立刻再按一次 UP 按键，会立刻返回搜索到的从机）。



4.2.3.3 查看搜索到的从机列表

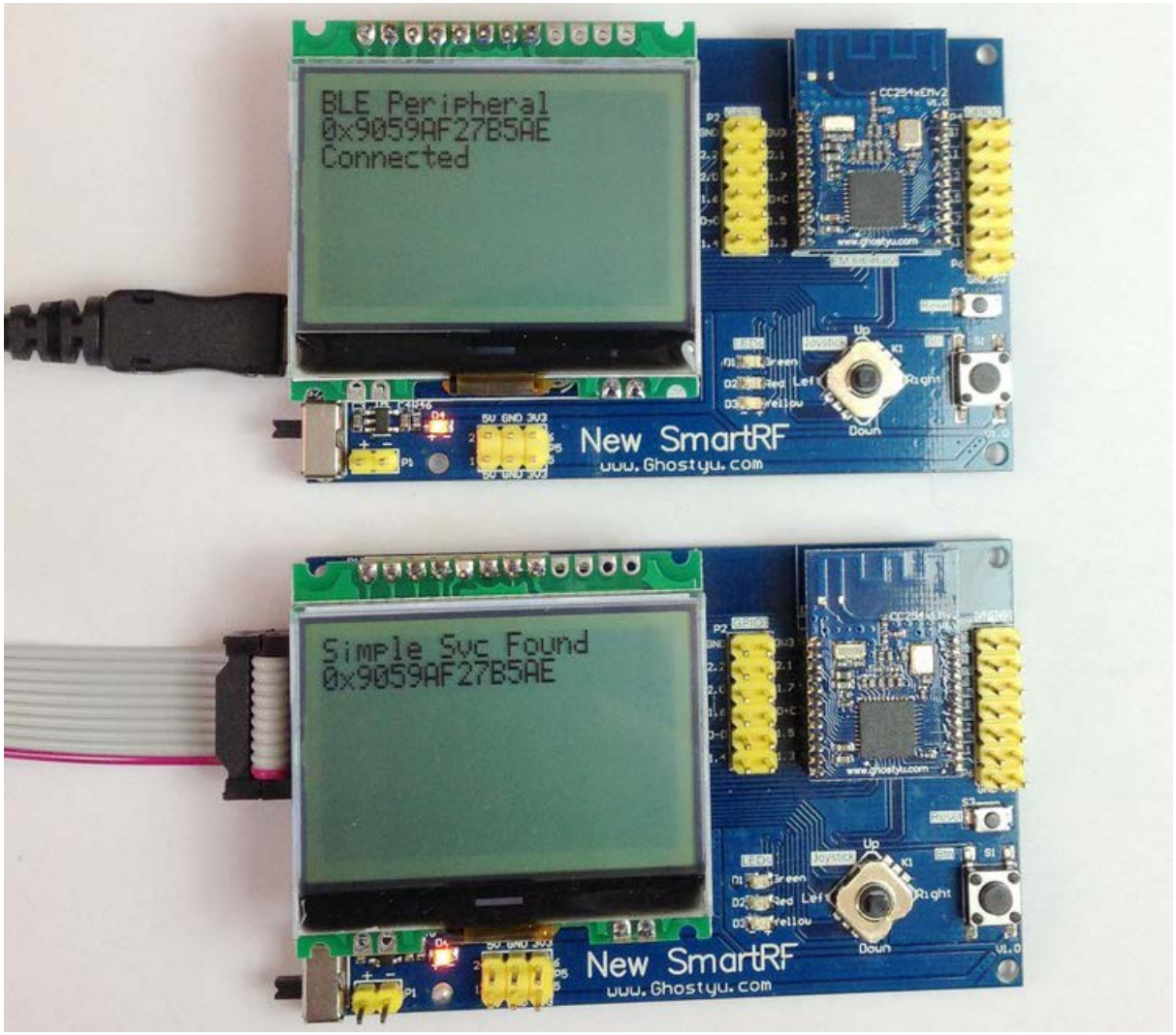
按下 Joystick Left 按键，进入搜索到的从机列表。



4.2.3.5 选择从机并且连接

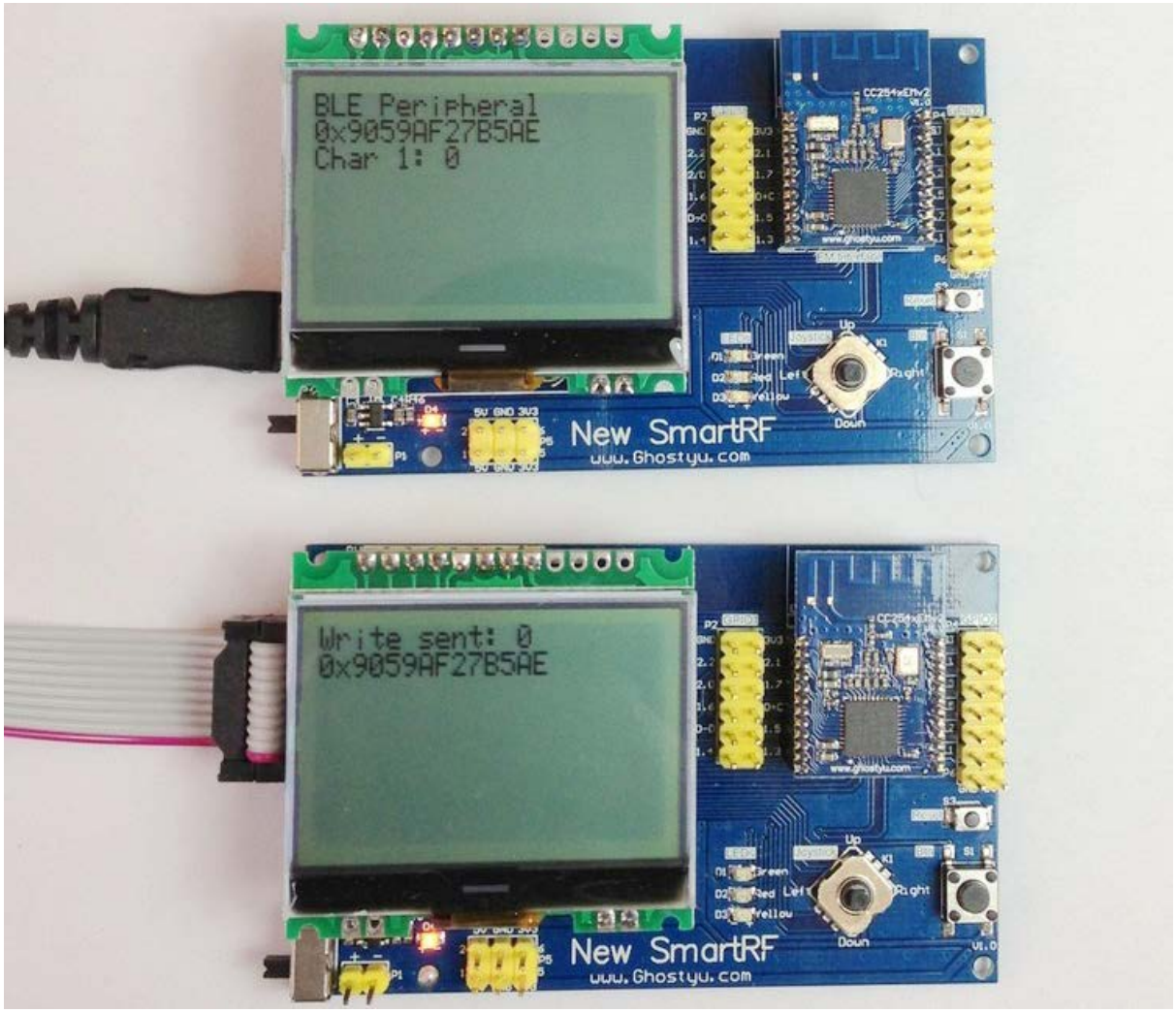
按下 Joystick Center 按键，开始连接选择的从机。

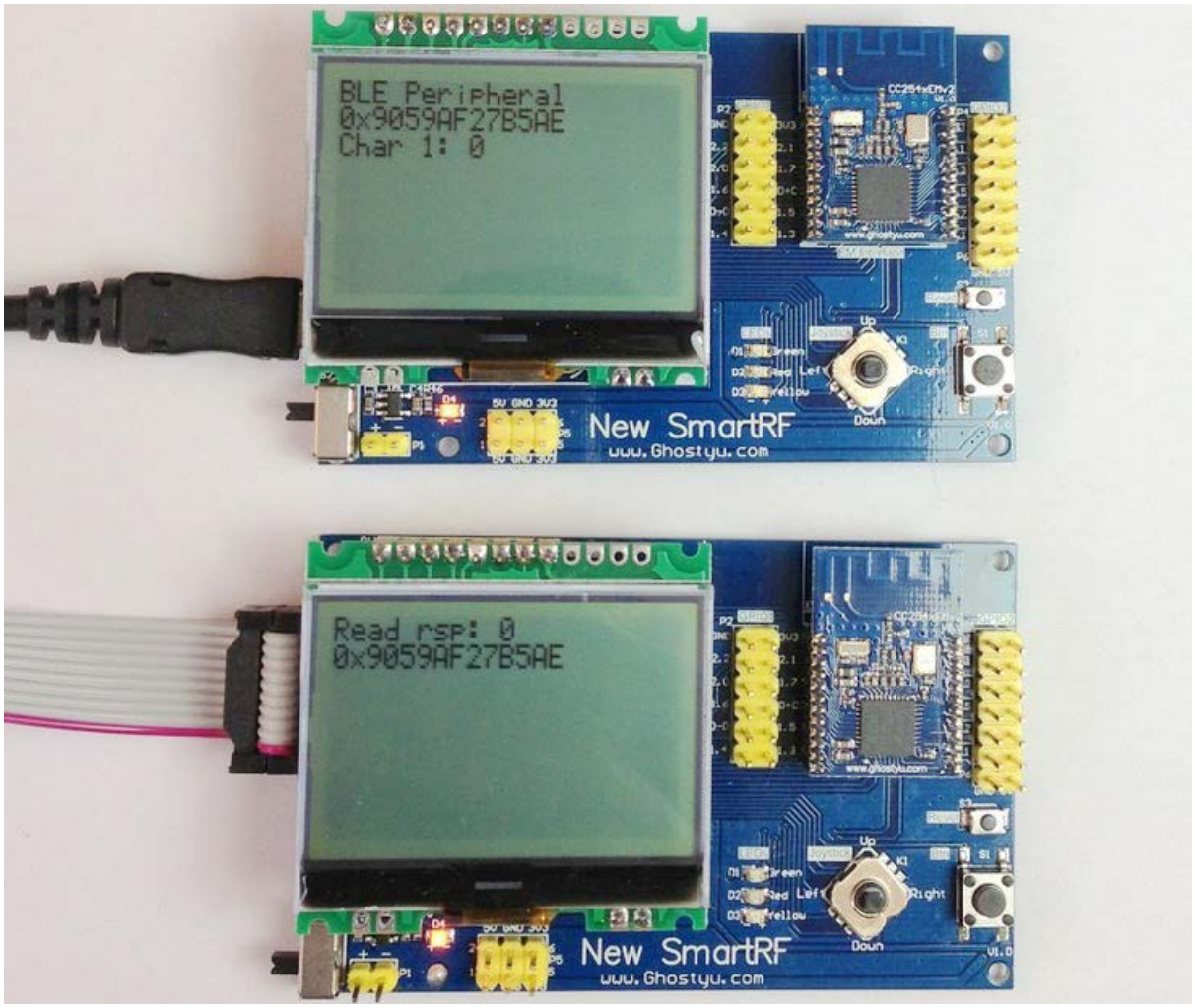
连接成功后会在 SmartRF 开发板的 LCD 上显示 Connected。



4.2.3.6 数据通信

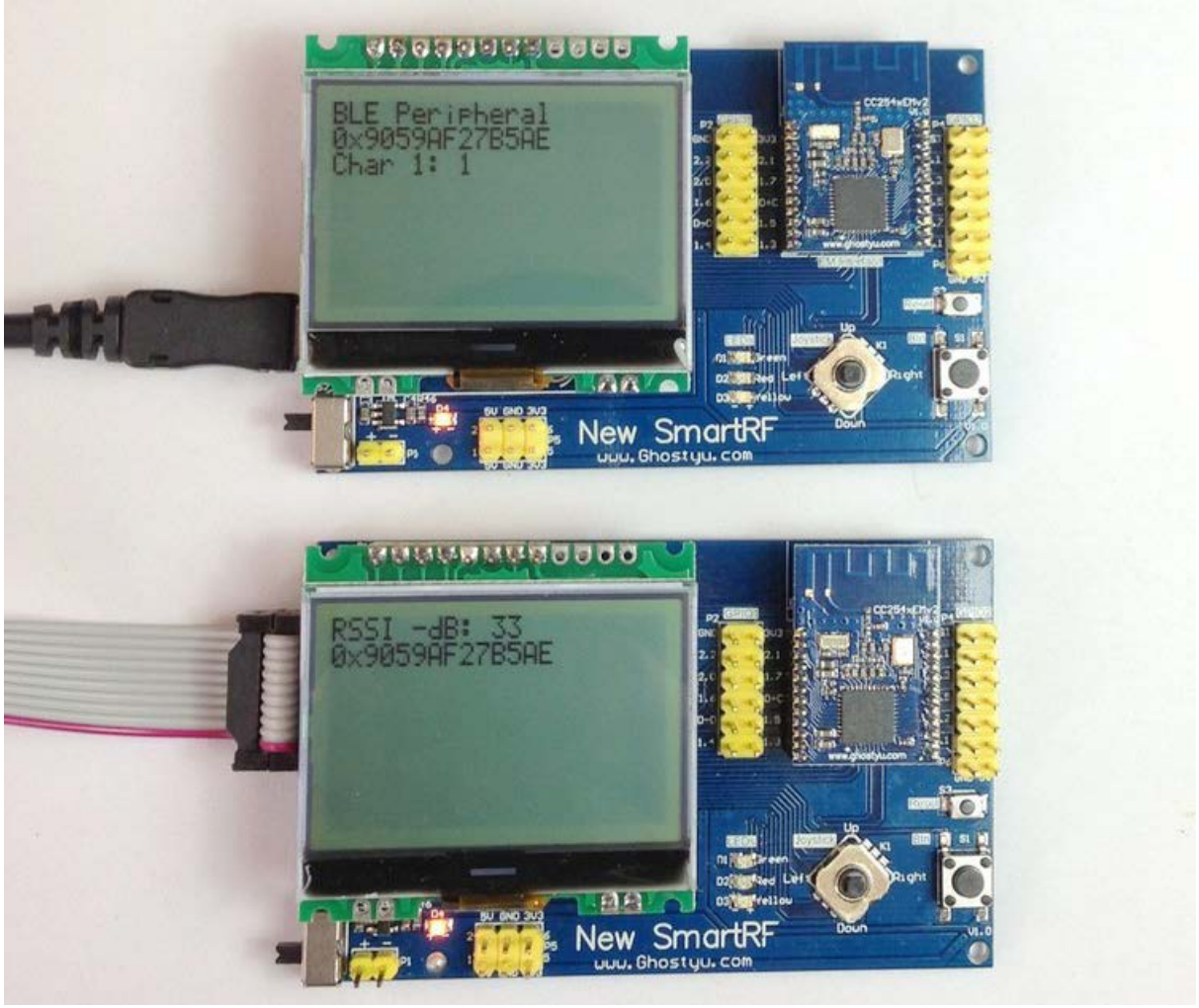
连接成功后，再按下 Joystick UP，会执行读写 char，按一次先 write char，然后再按一次是 read char，每一次循环，读写的 char 值增加 1。


BLE Peripheral
0x9059AF27B5AE
Char 1: 0
Write sent: 0
0x9059AF27B5AE
New SmartRF
www.Ghostyu.com
Up
Down
Left
Right



4.2.3.7 实时查询 RSSI 信号值

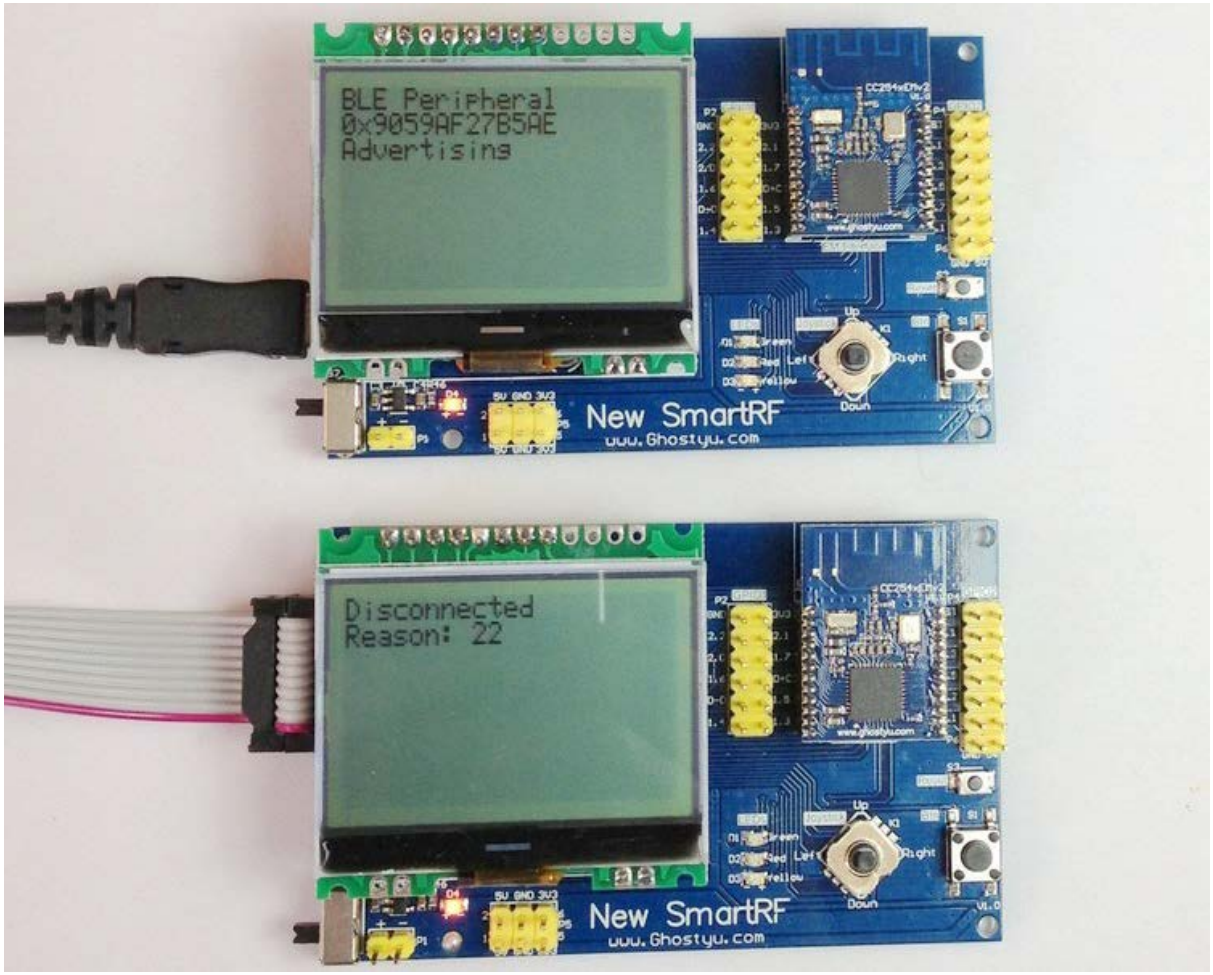
按 Joystick Down 可以获取从机的 RSSI 值，再按一下是取消 RSSI 值的显示。



注意显示的 RSSI 值为-33dB，将两个 CC2540 放在一起，RSSI 值能在-35dB 以内的，表明天线的效率很性能是很高的。

4.2.3.8 断开连接

最后断开连接是再次按下 Joystick Center 按键。断开后，SmartRF 的 LCD 会显示“Disconneced”



以上是通过 New SmartRF 实现的主从之间的通信，当然，我们完全可以使用智能机作为主机，来和 New SmartRF 上的从机通信。

iPhone 与开发板之间的通信实验

前言

有些用户肯定会问，除了 CC2540 之间的蓝牙通信的 Demo，如何让开发板与智能手机或者 PC 做通信实验呢。下面我们来一次做 iPhone 和 PC 与开发板的通信实验。由于 Android 手机目前没有原生支持 ble，所以暂时不考虑 Android 平台，需要客户自行研究。如果用户已经阅读了我们的 FAQ 手册，应该知道，只有 iPhone4S（含）以后的设备才支持低功耗蓝牙 BLE，我们这里做的实验使用的是 iPhone5 和 iPad4.并且已经从 APP Store 里下载安装了 LightBlue 程序。

通常，智能机设备作为主机，CC254x 作为从机，当然，CC254x 也可以作为主机，去连接当前状态为从机的智能机设备。说要说明的是，由于 SmartRF 开发板中烧写的主机程序搜索时限制了从机的 UUID，只有当从机的 UUID 为 FFF0 时才能被 SmartRF 开发板上的主机搜到，因此这里。

打开 iPhone 系统蓝牙开关

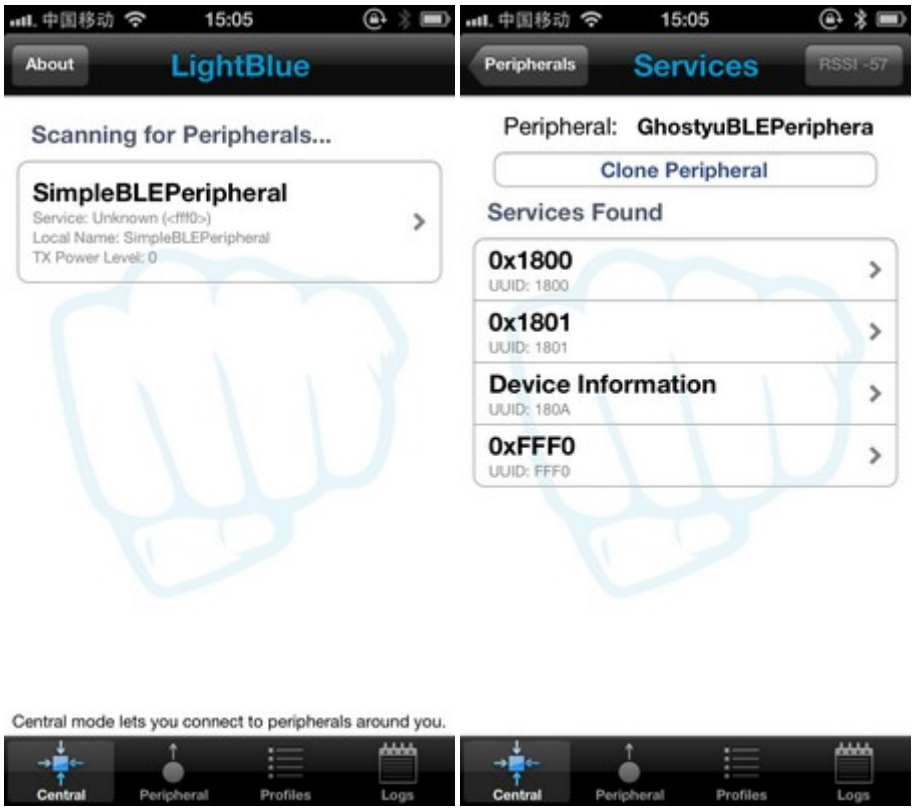
打开 iPhone 的蓝牙，然后运行 LightBlue 程序。

运行 Lightblue 程序

LightBlue 运行时，会自动搜索从机

搜索从机

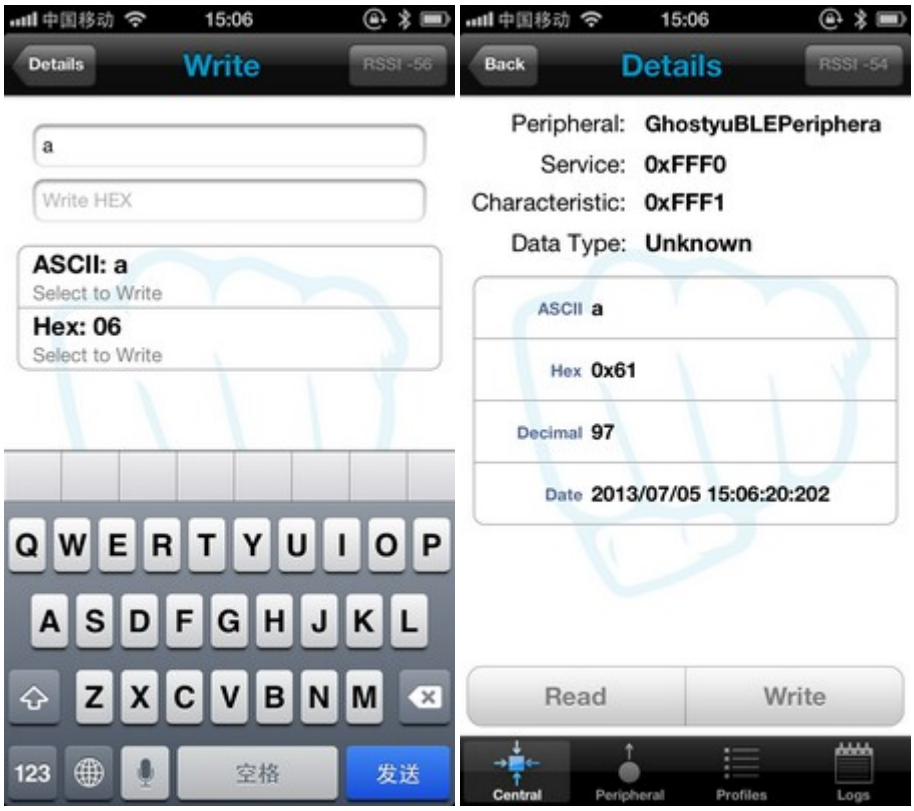
手动下拉 Scanning for Peripherals 可以手动搜索从机。搜索到从机后，会显示从机列别，并且包含主要信息，Services 的 UUID，还有发射功率，设备名称等。



连接从机
点击从机列表，iphone 会开始连接从机
Service 枚举
当连接到从机后程序会自动搜索从机的所有 Services，在第二幅图中显示的便是从机的所有 Services。
Characteristic 枚举
点击相应的 Service 会进入该 Service 中包含的 characteristics，如第三幅图。

数据通信

然后点击 Characteristics 列表中的具体的 Characteristic，会进入 Characteristic 的通信界面，就是读写 char 或者 Notify。



比如单击 Write 向 Characteristic 为 FFF1 的写入 ascii 码“A”，然后在点击 Read 会读到刚才写入的“A”。

LightBlue 是 iOS 上非常有用的 ble 程序，平时开发 2540 的从机时，可以用该程序做测试。

PC 与开发板之间的通信实验

前言

我们这里说的 PC 与开发板之间的通信实验，并非是使用 PC 上的蓝牙适配器，而是使用 TI 的 btool，btool 是 TI 开发板的 window 上的蓝牙调试软件，配合烧写 HostTestRelease 程序的 CC2540，作为 PC 端的 BLE 调试软件。

运行 BTool 有两种方式：

SmartRF 开发板烧写 HostTestRelease，通过 RS232 连接 PC，然后运行 BTool。

CC2540USBdongle 烧写 HostTestRelease，通过 USB 连接 PC，安装 TI 的驱动程序，将 usbdongle 模拟成串口，然后运行 BTool。

有关 SmartRF 开发板或 USBdongle 烧写运行 HostTestRelease，详情请参见【开发资料】目录下的《BTool 使用指南.pdf》。我们这里仅讨论如何使用 Btool，与烧写了 SimpleBLEPeripheral 从机程序的 SmartRF-BB 板的通信。

开发板通电

给烧写了 SimpleBLEPeripheral 从机程序的 SmartRF-BB 开发板上电。

运行 BTool 软件

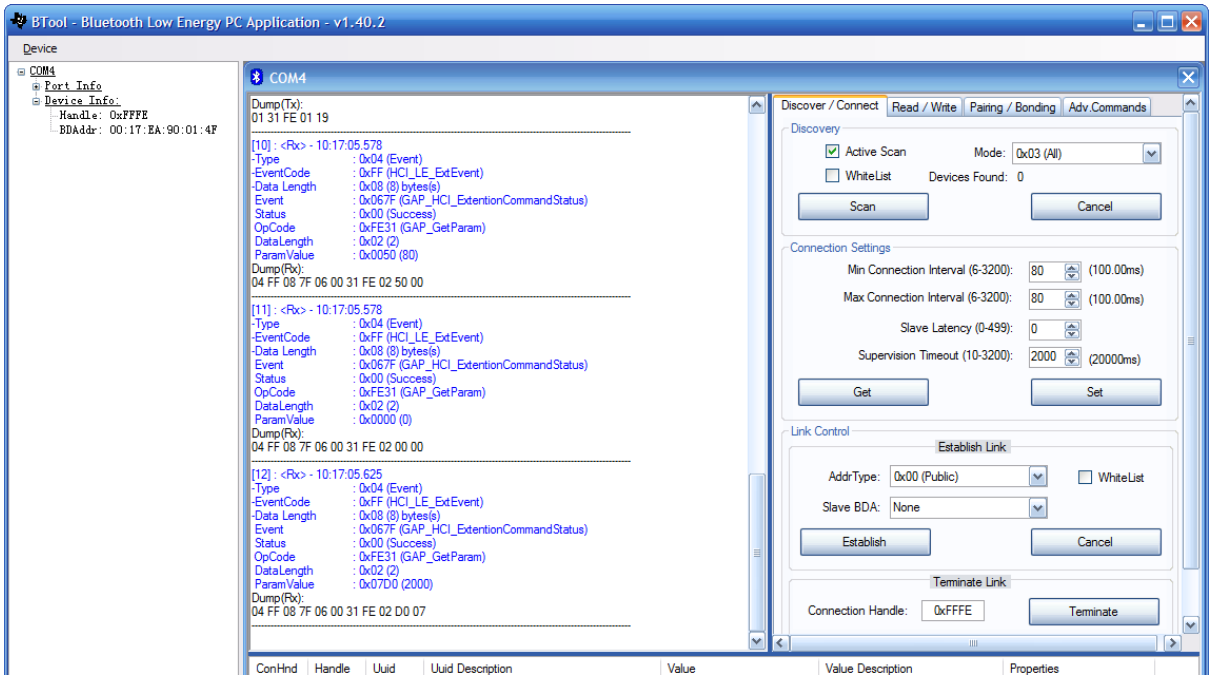
打开 BTool 软件，会自动跳出串口设置的对话框。需要注意的是 BTool 不并不能独立运行，需要 SmartRF 开发板或者 CC2540USBdongle 配合，SmartRF 开发板默认烧写主机程序，CC2540USBdongle 默认烧写协议分析仪固件，因此做该实验，需要对二者任选其一

重新烧写 HostTestRelease 固件。

端口设置

具体设置如下图，Port 选择开发所连接的端口，Band 设置为 115200，HostTestRelease 程序默认的波特率为 115200，Flow 流控制设为 CTS/RTS，Parity 设置 Nonw，StopBits 停止位设为 1，DataBits 数据位设为 8，单后单击 OK。

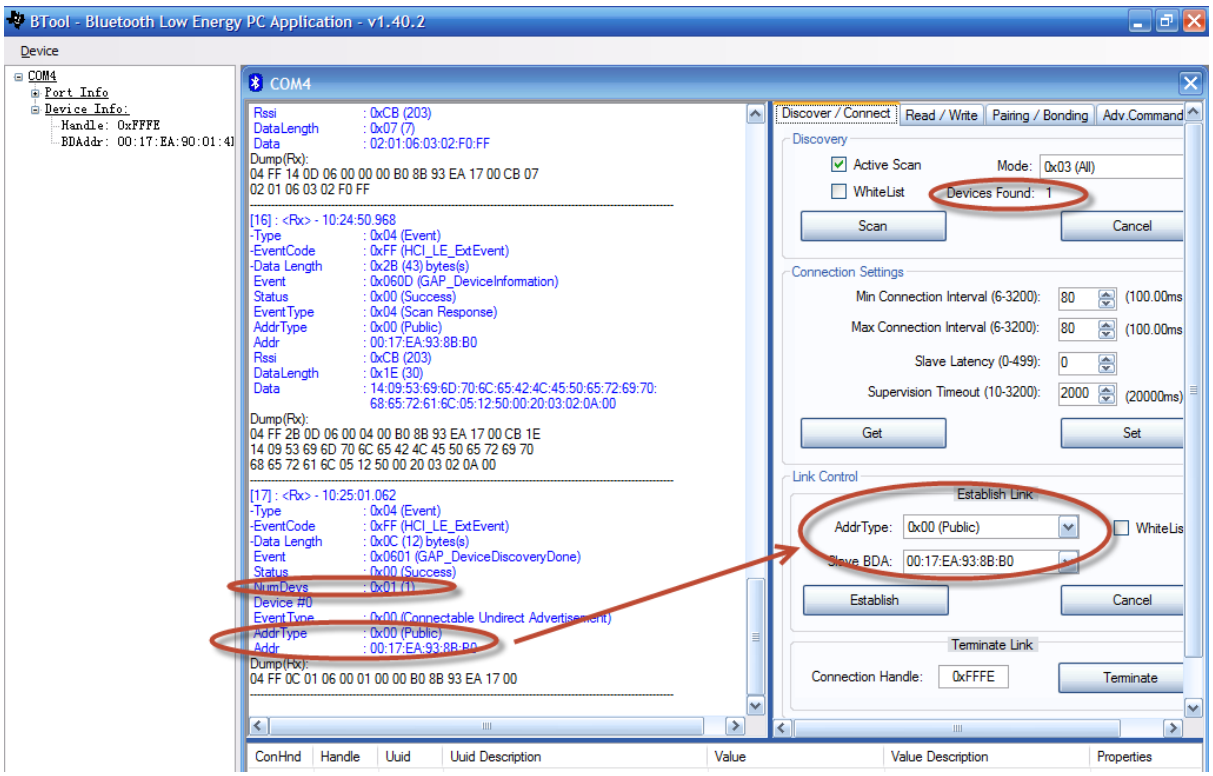
单击 OK 后会出现下列界面，如果出现超时等错误，请检查



Btool 程序界面，主要分为三个部分，左边的设备列表，中间的收发信息心中和右边的控制中心。

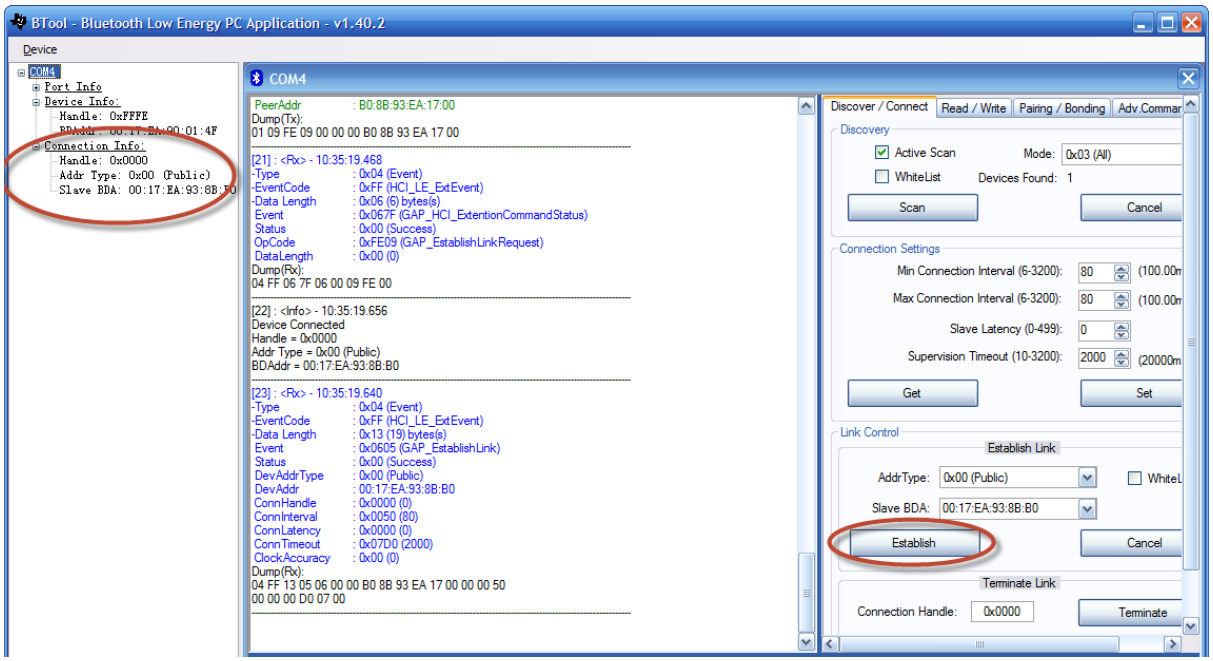
搜索从机

单击控制中心的 Scan 按钮开始搜索从机设备。过一会返回搜索的结果，如下图，已经找到一个从机设备。



连接从机

单击 Link Control 中的 Establish，开始连接从机，连接正确后，如下图，在设备列表中，会出现 Connection Info。

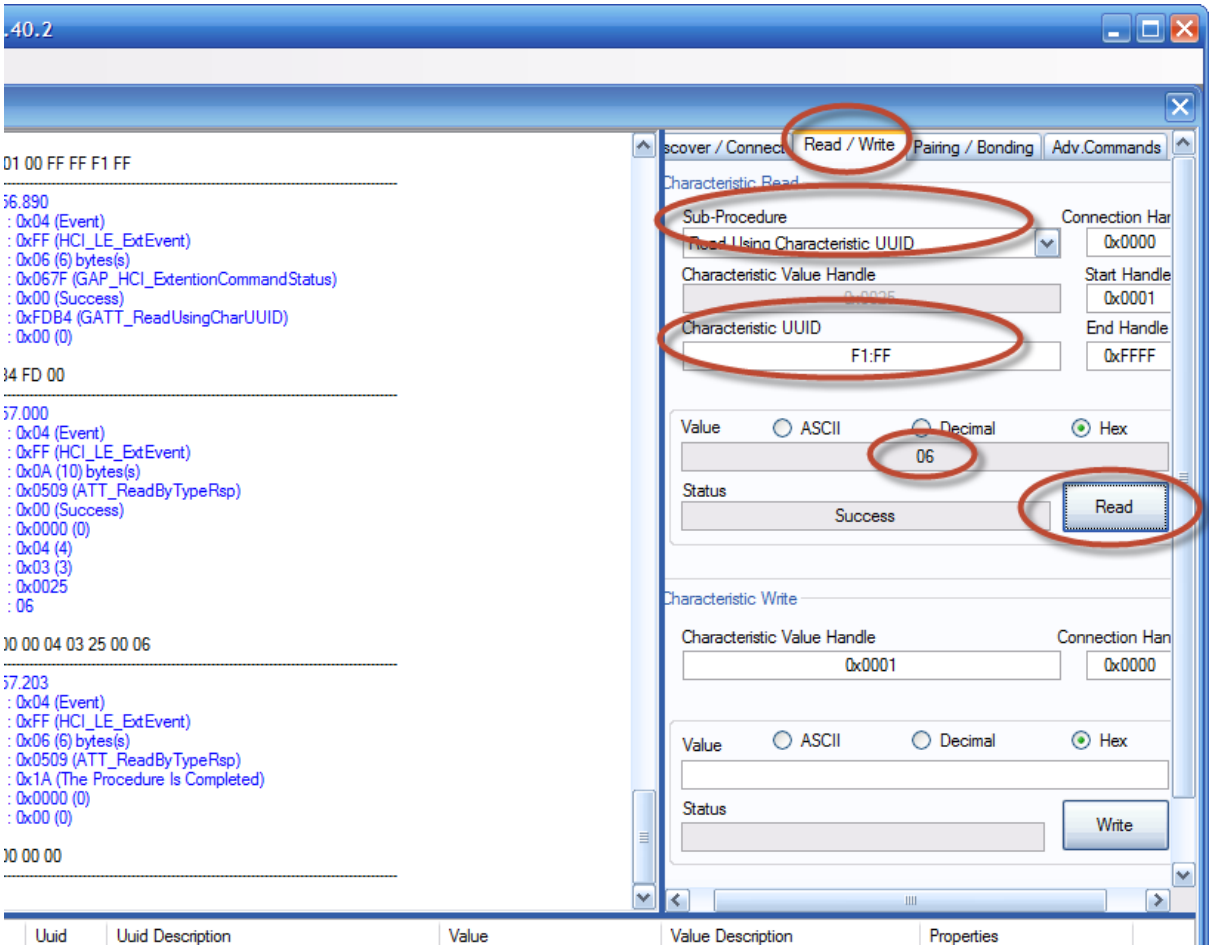


数据通信

执行读 char 操作。

成功连接后，即可进行 char 的读写实验，单击控制中心的 Read/Write，进入 Characteristic 读写页面。然后在 Characteristic Read 里的 Sub-Proceduce 里选择第二条：Read Using Chracteristic UUID，表示通过 UUID 来读 Char。然后在 Characteristic UUID 中填入 F1:FF，这里注意，UUID 的正确形式是 FFF1，这里高低字节需要反一下。然后单击 Read，执行

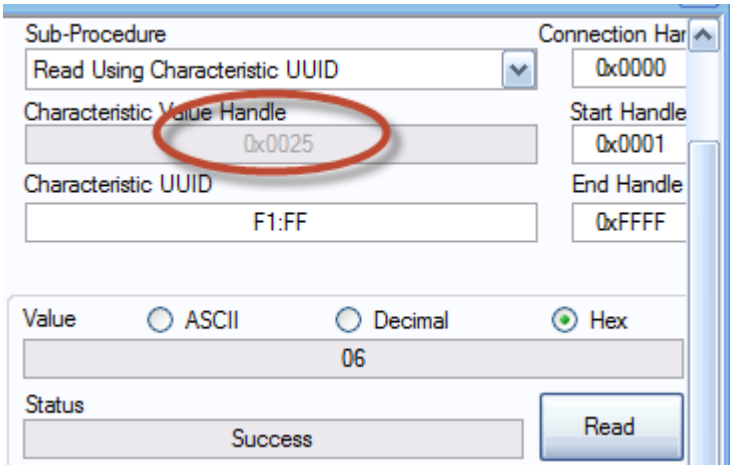
读操作，如下图：



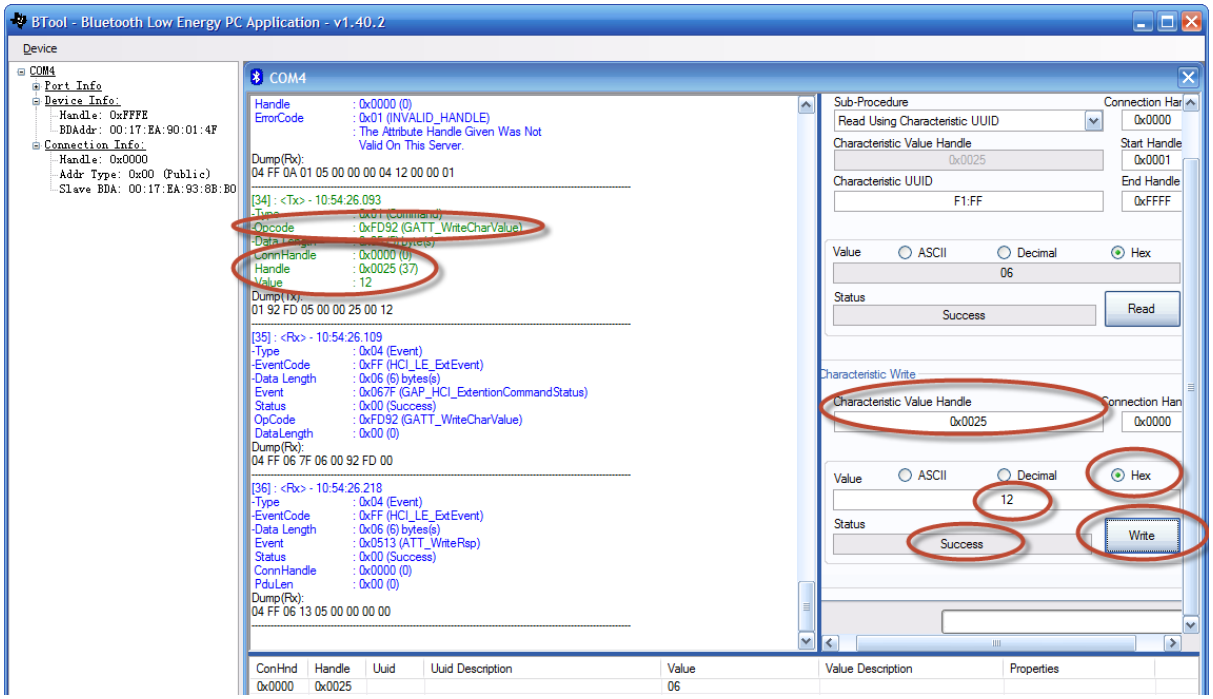
图中读到的 16 进制数 06 是我先前通过手机写进去的值。

执行写 char 操作。

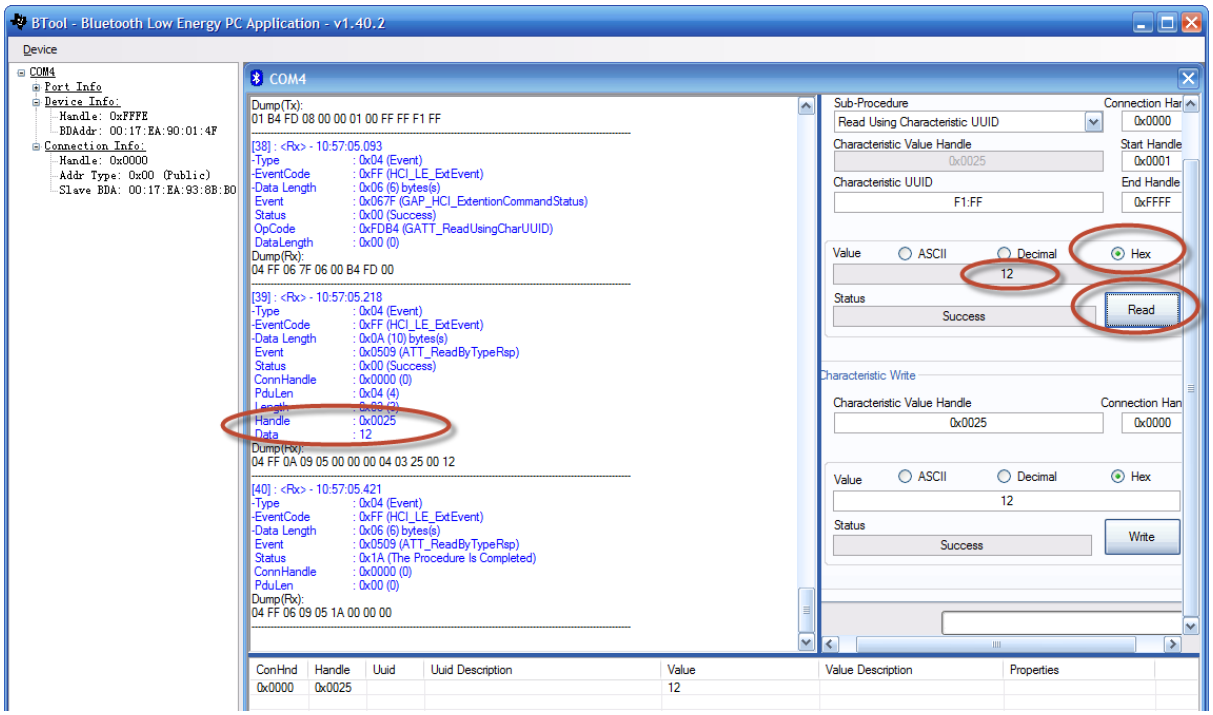
写 char 和读 char 有略微不同，我们读是通过 UUID，SimpleBLEPeripheral 中的 5 个 characteristic 的 UUID 从 FFF1 到 FFF5.读可以通过这几个 UUID，但是写只能通过 Characteristic Value Handle。但是怎样得到 FFF1 对应的 Characteristic Value Handle 呢。还是通过上一步的读操作，如下图，不同的 Characteristic UUID 对应的 Handle 已经自动出现在了 Characteristic Value Handle 中。



这样，我们在 Characteristic Write 栏目的 Characteristic Value Handle 中填入：0x0025，然后在 Value 中随意写一个 16 进制数，最后单击 Write，如下图



在 Status 中显示 Success，表明写 char 成功，然后在通过 Read，看下是否已经成功将 12 写到从机上。如下图，实验成功。对于其他的 characteristic UUID 操作类似。



到这里，我们通过 PC 完成了简单的 BLE 的通信实验，这里仅仅是简单的演示，btool 有着非常强大的功能，在后面我们会详细介绍。

到这里，可能读者有很多问题。如：

- 什么是 Central 主机，什么是 Peripheral 从机，Profile 和 Characteristic 又都是什么
- 主从之间是如何建立数据通信的

• GATT_WriteCharValue 和 GATT_ReadCharValue 是如何实现的
• 从机如何主动向主机发送数据
• 什么是 OSAL，系统消息又是什么
• 为什么上述代码中有很多固定的部分不需要改动呢；
• BLE 协议栈包含了很多代码，从哪里开始执行呢；

也许读者还有其他很多类似的问题，虽然有诸多的问题，但是却实现了主从之间的无线数据传输。这就是协议栈。也是使用协议栈进行程序开发的便利之处，用户只需要关注所发送的数据。尽量少考虑 BLE 协议栈的具体实现细节。

上述问题会一直困扰着读者，但是由于这些问题暂时无法解决，所以读者需要不断去学习理论知识，去做实验观察效果. 进而最终把上述问题解决，相信读者把上述问题解决的时候，会感觉 BLE 协议已经基本理解并能熟练使用了。

4.3 BLE 数据传输实验剖析

本节这是对上述实验进行原理上的讨论，具体的函数代码并没有过多的讨论，目的是为了使读者明白实验思路，具体的代码只要用多了自然就熟悉了。

在讲数据通信之前，我们先普及一下知识，也就是刚才我们遇到的默认词汇。

1、 Profile

Profile 可以理解为一种规范，一个标准的通信协议，Profile 存在于从机中。蓝牙组织规定了一些列的标准 Profile，例如 HID OVER GATT、防丢器、心率计等。每个 Profile 中会包含多个 Service，每个 Service 代表从机的一种能力。

2、 Service

Service 可以理解为一个服务，在 ble 从机中，通过有多个服务，例如电量信息服务、系统信息服务等，每个 Service 里又包含多个 Characteristic 特征值。每个具体的 Characteristic 特征值，才是 ble 通信的主体。比如当前的电量是 80%，所以会通过电量的 characteristic 特征值存在从机的 profile 里，这样主机就可以通过这个 characteristic，来读取 80%这个数据

3、 Characteristic

Characteristic 特征值，BLE 主从机的通信均是通过 Characteristic 来实现，可以理解为一个标签，通过这个标签可以获取或者写入想要的内容。

4、 UUID

UUID，统一识别码，我们刚才提到的 Service 和 Characteristic，都需要一个唯一的 UUID 来标识。

整理一下，每个从机都会有一个叫做 profile 的东西存在，不管是上面的自定义的 simpleProfile，还是标准的防丢器 Profile，他们都是由一些列 Service 组成，然后每个 Service 又包含了多个 Characteristic，主机和从机之间的通信，均是通过 Characteristic 来实现。

4.3.1 数据发送

在 BLE 协议栈中进行数据分为两个方面，一个是 GATT 的 client 主动向 service 发送数据，另外一个是 GATT 的 service 主动向 client 发送数据

我们暂且简单地分为主机向从机发送数据，从机主动向主机发送数据。

4.3.1.1 主机向从机发送数据

发送可以调用 GATT_WriteCharValue 函数实现，该函教会调用协议栈里面与硬件相关的函数最终将数据通过天线发送出去，这里面涉及对射频模块的操作，例如：打开发射机，调整发射机的发送功率等内容，这些部分协议栈已经实现了，用户不需要自己写代码去实现，只需要掌握 GATT_WriteCharValue 函数的使用方法即可。需要发送的数据填充到 value 中，然后数据长度填充到 len 中。如下图。

```
00442:          // Do a write
00443:          attWriteReq_t req;
00444:
00445:          req.handle = simpleBLECharHdl;
00446:          req.len = 1;
00447:          req.value[0] = simpleBLECharVal;
00448:          req.sig = 0;
00449:          req.cmd = 0;
00450:          status = GATT_WriteCharValue( simpleBLEConnHandle, &req, simpleBLETaskId );
```

4.3.1.2 从机向主机发送数据

从机向主机发送数据，需要调用 GATT_Notification 函数实现，上面的主从机没用使用到该函数。开发串口透传等项目时，就需要在从机中主动向主机发送数据，需要该函数来实现。下面的代码是来自我们的串口透传中。需要填充的和 GATT_WriteCharValue 类似，value 和 len。

```
00081:    static attHandleValueNoti_t pReport;
00082:    pReport.len = length;
00083:    osal_memcpy(pReport.value, pBuffer, length);
00084:    GATT_Notification( 0, &pReport, FALSE );
```

4.3.2 数据接收

数据接收和数据发送一样，同样分为两个方面。

从机接收主机发来的数据和主机接收来自从机的数据

4.3.2.1 从机接收主机发来的数据

当从机接收到主机发来的数据后，从机会产生一个 GATT Profile Callback 调用，我们在这个 callback 中接收主机发送的数据。这个 callback 在从机初始化时向 Profile 注册。

```
00257: // Simple GATT Profile Callbacks
00258: static simpleProfileCBs_t simpleBLEPeripheral_SimpleProfileCBs =
00259: {
00260:   simpleProfileChangeCB     // Charactersitic value change callback
00261: };

00417:   // Register callback with SimpleGATTprofile
00418:   VOID SimpleProfile_RegisterAppCBs( &simpleBLEPeripheral_SimpleProfileCBs );
```

```
00752: static void simpleProfileChangeCB( uint8 paramID )
00753: {
00754:   uint8 newValue;
00755:
00756:   switch( paramID )
00757:   {
00758:     case SIMPLEPROFILE_CHAR1:
00759:       SimpleProfile_GetParameter( SIMPLEPROFILE_CHAR1, &newValue );
00760:
00761:       #if (defined HAL_LCD) && (HAL_LCD == TRUE)
00762:         HalLcdWriteStringValue( "Char 1:", (uint16)(newValue), 10,  HAL_LCD_LINE_3 );
00763:       #endif // (defined HAL_LCD) && (HAL_LCD == TRUE)
00764:
00765:       break;
```

到此为止，读者至少理清楚这条线索：主机通过 BLE 提供的数据发送接口发送数据后，从机的协议栈收到数据后，做相应处理，取得自己需要的数据即可，其他工作，都由 BLE 协议栈自动完成了。

4.4 BLE 数据包的捕获

以上是从原理上对 BLE 网络的数据流进行的分析，这是建立对 BLE 协议栈充分熟悉的基础上，初学者阶段很难理解，因此，我们推荐读者可以使用 BLE 协议分析仪进行通信抓包，然后分析捕获的数据包，今儿更加形象的理解数据解的传输过程。（注意，由于蓝牙通信时是跳频，因此无法捕获通信是的数据，我们仅分析主机和从机的连接过程。）

4.4.1 如何构建 BLE 协议分析仪

使用 BLE 协议分析仪需要硬件和软件上的支持，使用 BLE 抓包一般需要以下的硬件和软件。

硬件	CC2540USBdongle
	CC-Debugger+SmartRF 开发板
软件	Texas Instruments PacketSniffer

① 用 CC2540USBdongle 抓包，非常简单，因为 CC2540usbdongle 出厂的时候，已经烧写了 packetsniffer 的固件，只需要将 CC2540usbdongle 查到 pc 的 usb 上，安装 ti 提供的驱动程序，然后打开 PacketSniffer 软件即可。

② 用 SmartRF 开发板抓包时步骤要多一些，需要 cc-debugger 连接 SmartRF 开发板，然后通过 CC-Debugger 连接至 PC 的 USB，然后通过 flash programmer 下载 SmartRF 上的协议分析仪固件（默认路劲为 C:\Program Files\Texas Instruments\SmartRF Tools\Packet Sniffer\bin\general\firmware\sniffer_fw_cc2540.hex），然后打开 PacketSniffer 软件。开始抓包。

打开 TI PacketSniffer 软件（该软件使用前需要安装，安装包在 Software/TI/目录下），如图 4-22 所示，选择 IEEE 802.15.4/BLE，然后单价 Start 按钮。

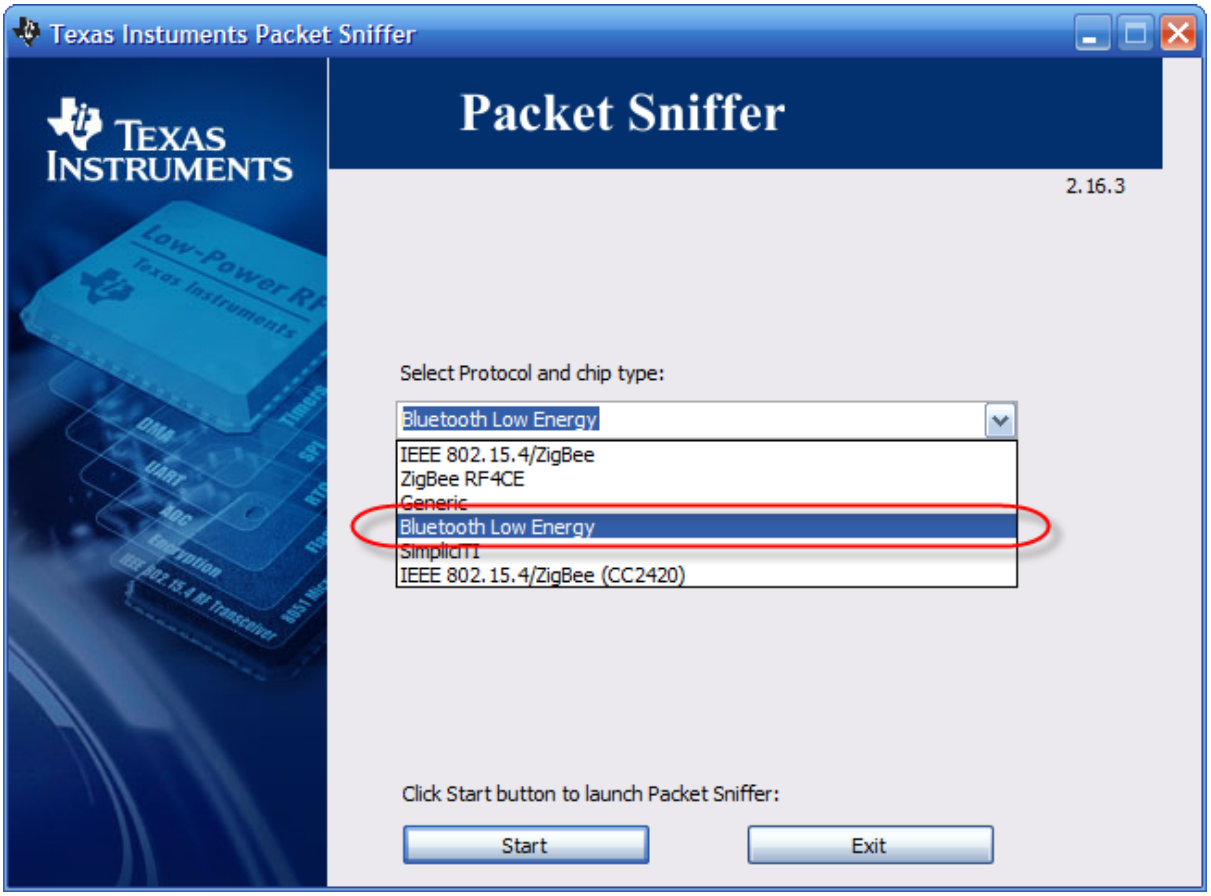
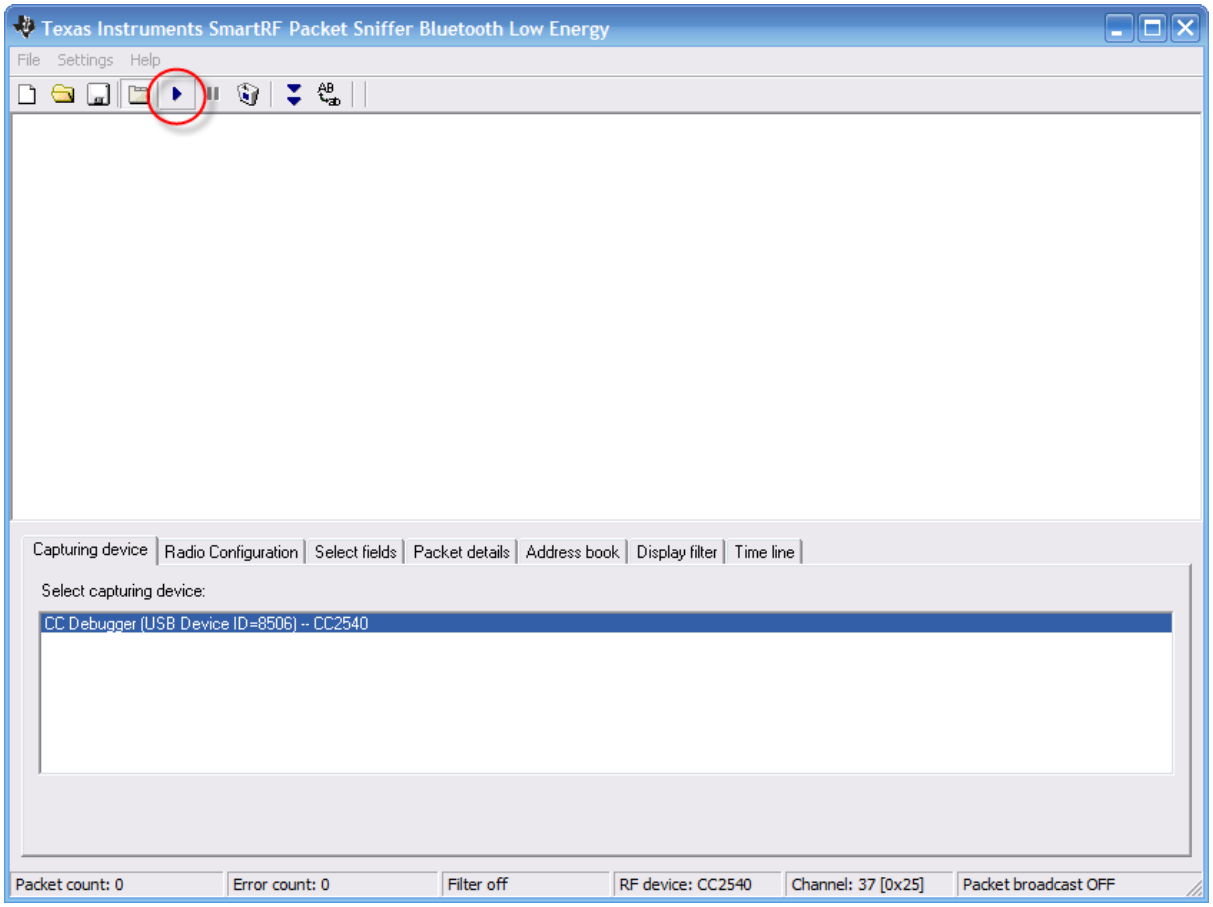
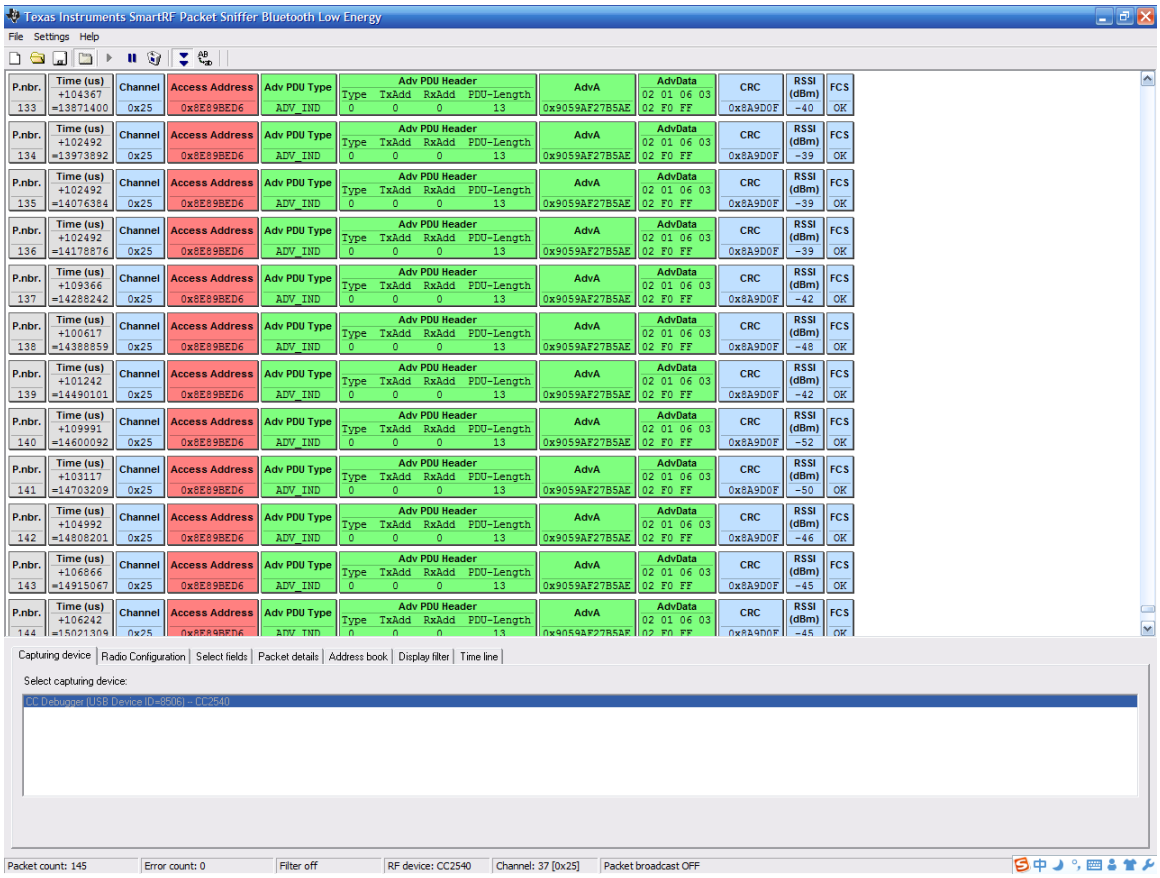


图 4-22 PacketSniffer 软件

此时，会弹出 packetSniffer 的主窗口，在窗口的底部 Select capturing device 中已经发现了开发板，单击蓝色小三角按钮（开始抓包）即可进行无线抓包。



现在就可以进行抓包了，一次打开协调器电源和终端节点电源，此时在 PacketSniffer 就可以显示捕获的数据包，需要有从机在广播，如图 4024 所示



4.4.2 BLE 数据包的结构

那么折现跳出来的数据都是怎么出现、又是代表什么意思呢？这就需要好好的分析 SimpleBLEPeripheral 的广播。
最简单的从机例程 SimpleBLEPeripheral

4.4.2.1 广播 Advertising

之前描述的五种状态中，从机广播是其中一种，只有当从设备出于广播状态，主机才能搜索到，那么从机广播那些内容呢？

4.4.2.2 代码中的广播数据

打开 Source Insight3.5 创建完整的 ble 源码工程，然后定位到 SimpleBLEPeripheral 从机工程源码，在 SimpleBLEPeripheral.c 第 200 行有如下代码，定义了广播时的内容，如下图所示。

```
00198: // GAP - Advertisement data (max size = 31 bytes, though this is
00199: // best kept short to conserve power while advertisting)
00200: static uint8 advertData[] =
00201: {
00202:   // Flags; this sets the device to use limited discoverable
00203:   // mode (advertises for 30 seconds at a time) instead of gener
00204:   // discoverable mode (advertises indefinitely)
00205:   0x02,   // length of this data
00206:   GAP_ADTYPE_FLAGS,
00207:   DEFAULT_DISCOVERABLE_MODE | GAP_ADTYPE_FLAGS_BREDR_NOT_SUPPORT
00208:
00209:   // service UUID, to notify central devices what services are i
00210:   // in this peripheral
00211:   0x03,   // length of this data
00212:   GAP_ADTYPE_16BIT_MORE,      // some of the UUID's, but not all
00213:   LO_UINT16( SIMPLEPROFILE_SERV_UUID ),
00214:   HI_UINT16( SIMPLEPROFILE_SERV_UUID ),
00215:
00216: };
```

从 avertData 数组的名称就可以看出，是广播数据。
第 205 行：0x02，表示接下来这一段数据的长度，注意，改长度不包括 0x02 数据本身。
第 206 行：GAP_ADTYPE_FLAGS，表示将要本段数据的类型或者用途等定义。
第 207 行：DEFAULT_DISCOVERABLE_MODE | GAP_ADTYPE_FLAGS_BREDR_NOT_SUPPORTED，第一段数据的主体，设置默认的默认的广播格式，并且不支持 BR/EDR 模式，CC2540 是 ble 单模芯片，是不支持 BR/EDR 的。
第 211 行：0x03，与 205 行意思相同，接下来的数据长度
第 212 行：与 206 行意思相同，广播时所展示的 Service 的 UUID
第 213、214 行：Service 的 UUID，广播时必须要携带一个主要的 service 的 uuid，SimpleBLEPeripheral 从机工程中使用的是 FFF0，central 搜索时也是更具 fff0 这个 uuid 来搜索这个从机的，如果从机的 uuid 不是 fff0，则 central 主机将无法搜索到该从机。

4.4.2.3 空中的广播数据

刚才是 C 语言展示的广播数据，现在用 Packetsniffer 抓包

P.nbr.	Time (us)	Channel	Access Address	Adv PDU Type	Adv PDU Header				AdvA	AdvData	CRC	RSSI (dBm)	FCS
	+101245				Type	TxAdd	RxAdd	PDU-Length		02 01 06 03			
264	=21035910	0x25	0x8E89BED6	ADV_IND	0	0	0	13	0x78C5E56EACD2	02 F0 FF	0x56BACE	-45	OK

个重要的参数：
Channel：0x25，我们的 packetsniffer 设置的 channel 为第一个频道：37，也就是 0x25，其实 ble 从设备广播的频道一共有三个。
AdvA：0x78C5E56EACD2,从设备地址，48 位的 IEEE 地址。
AdvData：02 01 06 03 02 F0 FF，从设备广播数据，这也就是上一节代码看到的广播数组 avertData 里的内容：02 是第一段的数据长度，然后 01 是 GAP_ADTYPE_FLAGS 宏定义的值，

```
#define GAP_ADTYPE_FLAGS                0x01 //!< Discovery Mode:
```

06 是宏定义 04 和 02 的组合值，03 是第二段数据项的长度，最后的 F0 和 FF 正是主要 Service 的 UUID：FFF0，这里注意下，只要是数值数据，低字节在前，高字节在后，在前面，我们使用 btool 读 uuid 的 value 时，填入的 uuid 也是第字节在前，高字节在后，例如 F1:FF（读 FFF1 的 value）。

4.4.2.4 扫描请求

扫描请求由主机发出，一下请求数据是通过 SimpleBLECentral 发出。

P.nbr.	Time (us)	Channel	Access Address	Adv PDU Type	Adv PDU Header				ScanA	AdvA	CRC	RSSI (dBm)
	+334				Type	TxAdd	RxAdd	PDU-Length				
265	=21036244	0x25	0x8E89BED6	ADV_SCAN_REQ	3	1	0	12	0x6F0AB5A6DFE0	0x70C7ED2EBCD6	0x084012	-33

Adv PDU Type：ADV_SCAN_REQ，实现为主机的扫描请求，Scan Request。

4.4.2.5 扫描回应 Scan Response

当我们使用 lightblue 软件或者 CC254x 的 Central 程序来扫描从机时，如果从机正在广播，将被扫描到，并且可以看到从机的设备名，发射功率等信息，而这些信息并未出现在刚才的广播数据中，这是因为什么呢。如下图，使用 LightBlue 扫描到的从设备。

当从机接收到主机发来的扫描请求时，会有一个扫描回应 Scan Response，在这个 response 中，携带了从机设备名，发送功率等信息，我们依然可以通过 PacketSniffer 来抓包分析。

P.nbr.	Time (us)	Channel	Access Address	Adv PDU Type	Adv PDU Header				AdvA	ScanRspData	CRC	RSSI (dBm)	FCS
	+326				Type	TxAdd	RxAdd	PDU-Length					
266	=21036570	0x25	0x8E89BED6	ADV_SCAN_RSP	4	0	0	36	0x78C5E56EACD2	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	0x944A74	-45	OK

从 Adv PDU Type 中可以看到，显示的扫描相应，SCAN_RSP。
在 ScanRspData 中，显示的是从机接收到扫描请求后想主机发送的相应数据，这段数据和广播数据一样，有对应的 C 代码。
代码中的扫描回应 Scan Response
从刚才的 ScanRspData 看到了一大串数据

ScanRspData
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

可以猜想，这串数据一定携带了设备名等其他更多的信息。我们在 SimpleBLEPeripheral.c 源文件里可以看到下面的代码：

```
00158: // GAP - SCAN RSP data (max size = 31 bytes)
00159: static uint8 scanRspData[] =
00160: {
00161:   // complete name
00162:   0x14,    // length of this data
00163:   GAP_ADTYPE_LOCAL_NAME_COMPLETE,
00164:   0x53,    // 'S'
00165:   0x69,    // 'i'
00166:   0x6d,    // 'm'
00167:   0x70,    // 'p'
00168:   0x6c,    // 'l'
00169:   0x65,    // 'e'
00170:   0x42,    // 'B'
00171:   0x4c,    // 'L'
00172:   0x45,    // 'E'
00173:   0x50,    // 'P'
00174:   0x65,    // 'e'
00175:   0x72,    // 'r'
00176:   0x69,    // 'i'
00177:   0x70,    // 'p'
00178:   0x68,    // 'h'
00179:   0x65,    // 'e'
00180:   0x72,    // 'r'
00181:   0x61,    // 'a'
00182:   0x6c,    // 'l'
00183:
00184:   // connection interval range
00185:   0x05,    // length of this data
00186:   GAP_ADTYPE_SLAVE_CONN_INTERVAL_RANGE,
00187:   LO_UINT16( DEFAULT_DESIRED_MIN_CONN_INTERVAL ),
00188:   HI_UINT16( DEFAULT_DESIRED_MIN_CONN_INTERVAL ),
00189:   LO_UINT16( DEFAULT_DESIRED_MAX_CONN_INTERVAL ),
00190:   HI_UINT16( DEFAULT_DESIRED_MAX_CONN_INTERVAL ),
00191:
00192:   // Tx power level
00193:   0x02,    // length of this data
00194:   GAP_ADTYPE_POWER_LEVEL,
00195:   0        // 0dBm
00196: };
```

和上面的 ScanRspData 对比一下，是不是一模一样

4.4.4 数据收发实验回顾

4.5 本章小结

本章主要是通过一个 BLE 无线网络中点对点通信实验来讲解数据收发的基本流程，对代码的讲解较为浅显，同事还给出了 BLE 协议分析仪的构建以及使用方法。

第 5 章 BLE 协议栈开发提高

上一章中给出了一个简单的点对点的无线通信实验，使读者对 Bluetooth-LE 低功耗蓝牙中的数据通信方法有个感性的认识。使用 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 BLE 协议栈的具体细节，但是读者需要对 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 BLE 协议栈提供的函数充分地融入自己的实际项目开发过程中。

本章将着重讨论 BLE 协议栈的构成以及内部 OSAL 的工作机理，在此基础上讲解一下 BLE 协议栈中的串口工作原理。

5.1 深入理解 BLE 协议栈的构成

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，控制接口层，主机部分包括：裸机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（GAP），通用属性配置文件层（GATT）。BLE 协议的构成如图 5-1 所示。

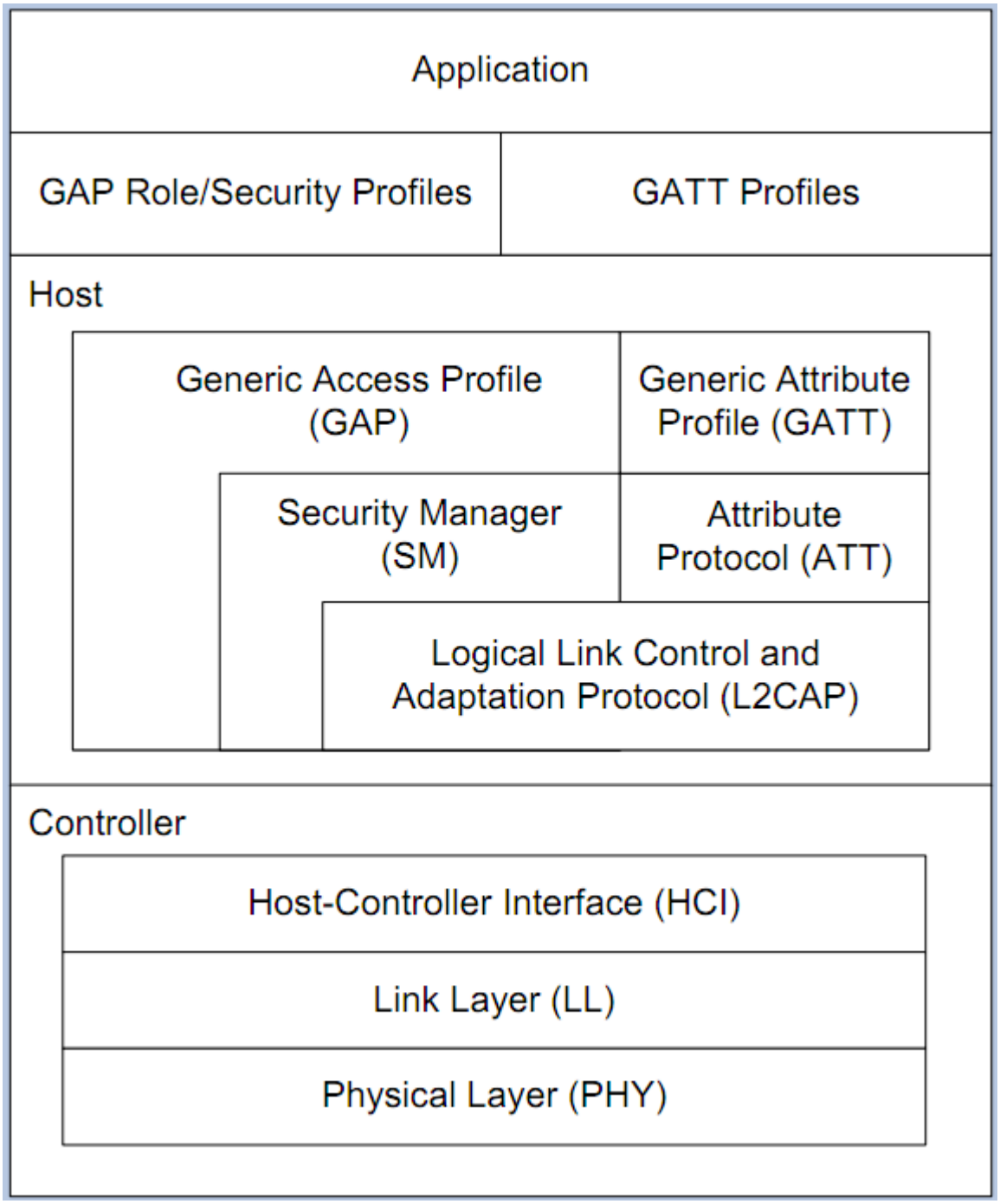


图 5-1

5.1.1 BLE 协议层

物理层（PHY）

RF 规格特性

-运行在 2.4GHz ISM band

-GFSK 调制方式（高斯频移键控）

-40 频道 2MHz 的通道间隙

*3 个固定的广播通道

*37 个自适应自动调频数据通道

LL 层，RF 控制层，控制芯片工作在 standby（准备）、advertising（广播）、scanning（监听/扫描），initiating（发起连接）、connected（已连接）这五个状态中的一种。五种状态的切换描述为：advertising（广播）不需要连接就可以发送数据（告诉所有人，我来了），scanning（监听/扫描）来自广播的数据，initiator（发起人）将携带 connection request（连接请求）来相应广播者，如果 advertiser（广播者）同意该请求，那么广播这和发起者都会进入已连接状态，发起连接的设备变为 master（主机），接收连接请求的设备变为 slave（从机）。

HCI 层

通信层，向 host 和 controller 提供一个标准化的接口。该层可以由软件 api 实现或者使用硬件接口 uart、spi、usb 来控制。

L2CAP 层

相当于快递，将数据打包，可以让客户点对点的通信。

SM 层

安全服务层，提供配对和密钥的分发，实现安全连接和数据交换。

ATT 层

允许设备向另外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，也就是说，主机设备可以是 ATT 服务器，也可以是 ATT 客户端。从机也一样。

GATT 层

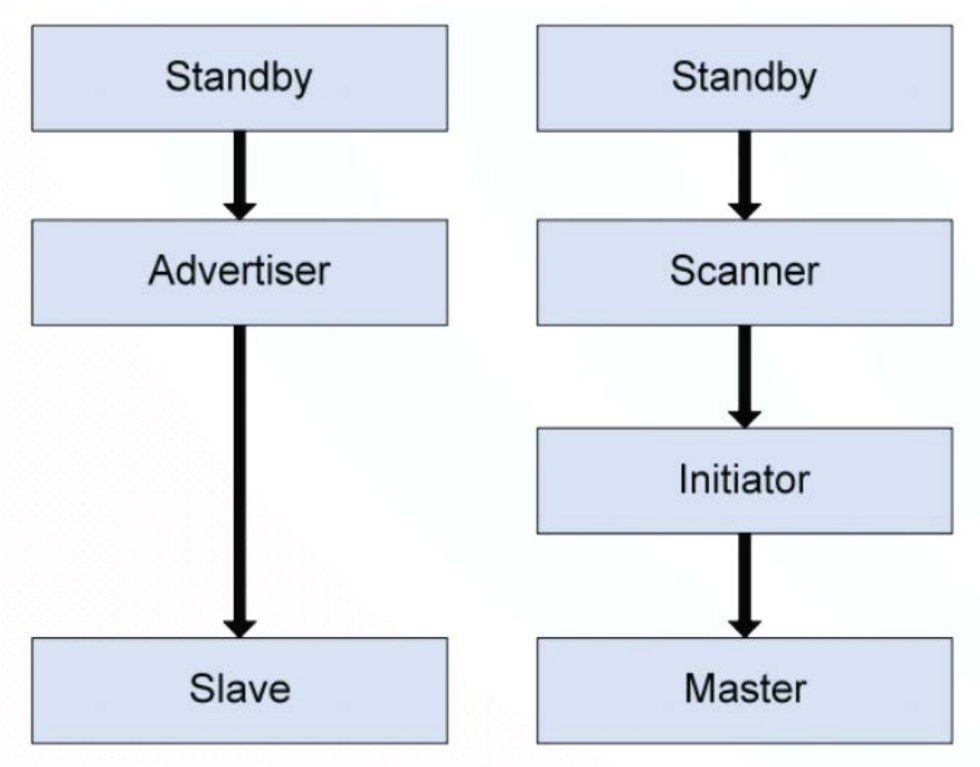
从名字就能看出，GATT 是在 ATT 上面的一层结构，定义了使用 ATT 的服务框架，GATT 规定了配置文件（鼎鼎有名的 profile）的结构，在 BLE 中，所有被 profile 或者服务用到的数据块都称为“特性，characteristic”两个建立连接的设备之间的所有数据通信都是通过 GATT 子程序处理，应用程序和 profile 直接使用 GATT 层，在后面具体的代码中，我们会经常见到 GATT。

以上各层，全部封装在 lib 库中，对外提供接口函数。函数库意外还有绿色的部分：OSAL、HAL、Profile、APPS 等均是源码的形式提供，

5.1.2 拓扑结构和设备状态

星形拓扑结构
主设备管理着连接，并且可以连接多个从机设备
一个从设备只能链接一个主设备
六中设备状态
待机状态（Standby）：设备没有传输和发送数据，并且没有连接到任何设备。
广播状态（Advertiser）：周期性广播状态。
扫描状态（Scanner）：主动的寻找正在广播的设备。
发起连接状态（Initiator）：主动向魔钩设备发起连接。
主设备（Master）：作为主设备连接到其他设备。
从设备（Slave）：作为从设备连接到其他设备。

5.1.3 BLE 状态以及连接过程



5.1.4 BLE 和快递服务类比

BLE		快递服务	
Master	Slave	某淘宝店	派送/收件员
待机模式	待机模式	无快递需求	失业中
扫描模式	广播模式	寻找快递公司	上门咨询需求
扫描请求	扫描回应	咨询能提供哪些服务（西藏送不送？能代收货款？等）	告知服务（业务范围，收费标准，等）
连接请求	-	签署业务合同	-
连接间隙	**（如：1 秒 1 次）**	**取件频率**	**（如：一天取一次）**
潜伏期	**如：2 秒 1 次**	**如果没有收包时间，是否同意间隔取件**	**如：隔一次取一次**
超时监管	**如：5 秒至少连 1 次**	**需要定时报道，否则终止合作**	**如：至少一周取一次**
连接时间	参数更新请求	准备邮包，等待取件	请求更改取件习惯
参数更新回应	-	同意更改（也可以不同意）	-
连接更新请求	-	重新签署业务合同（新服务方式）	-

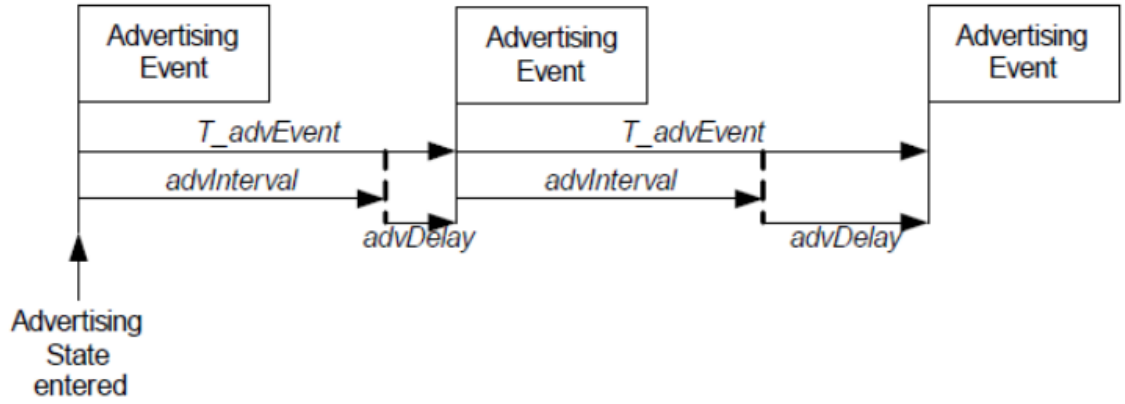
连接事件	定时收包/发包	准备邮包，等到取件	按时送包/取包
连接事件	。。。	准备邮包，等待取件	。。。

5.1.5 BLE 广播事件

广播包的发送是单向的，不需要任何连接。设备发送广播包进入广播状态。
-广播包可以包含特定的数据定义，最大 31 个字节。
-广播包可以直接指向摸个特定的设备，也可以不指定。
-广播包中可以声明是可被连接的设备，或者是不可连接的设备
在一个广播时间中，广播包会分别在三个广播通道中被发送一次（37，38, 39）

5.1.6 BLE 广播间隔

广播间隔，是两次广播事件之间的最小时间间隔，一般取值范围在 20ms~10.24s 之间。链路层会在每次广播时间器件产生一个随机广播延时时间（0ms~10ms），这个延时被加在广播时间中，这样来避免多设备之间的数据碰撞。



5.1.7 BLE 扫描事件

每次扫描设备打开 Radio 接收器去监听广播设备，这样称为一个扫描时间
扫描时间交替发生在三个特定的广播通道中：37、38、39

扫描频宽比（Duty-Cycle），关于扫描的两个时间参数：
-扫描时间：即扫描设备的扫描频度
-扫描窗口：每次扫描事件持续的时间

5.1.8 BLE 发起连接

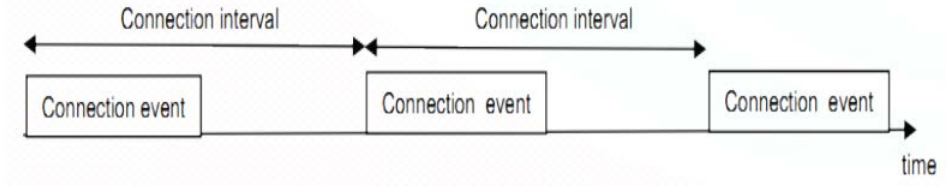
除了扫描，设备也可以主动发起连接，发起状态的设备和扫描状态的设备区别在于：当他监听到一个可连接的广播，发起设备会发送一个连接请求，而扫描设备会发送一个扫描请求。
连接请求包括一套为从设备准备的连接参数，安排连接时间发生的通道和时间。
如果广播设备接收了连接，两个设备会进入连接状态，发起方会称为 Master，而广播方会称为 Slave

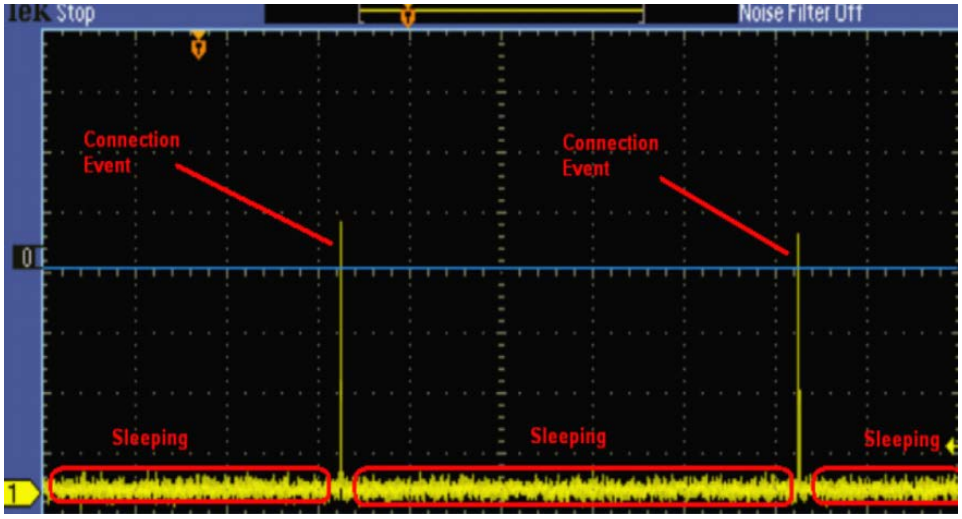
5.1.9 BLE 连接参数

通道映射，指示连接使用的频道。
调频增量，一个 5~16 之间的随机，参与通道选择的算法。
连接间隔，1.25ms 的倍数，7.5ms~4s 之间
监督超市，10ms 的倍数，100ms~32s 之间，必须大于（1+slaveLatency）*ConnInterval
从机潜伏，0~499 之间，不能超过（SupervisionTimeout/connInterval）-1

5.1.10 BLE 连接事件

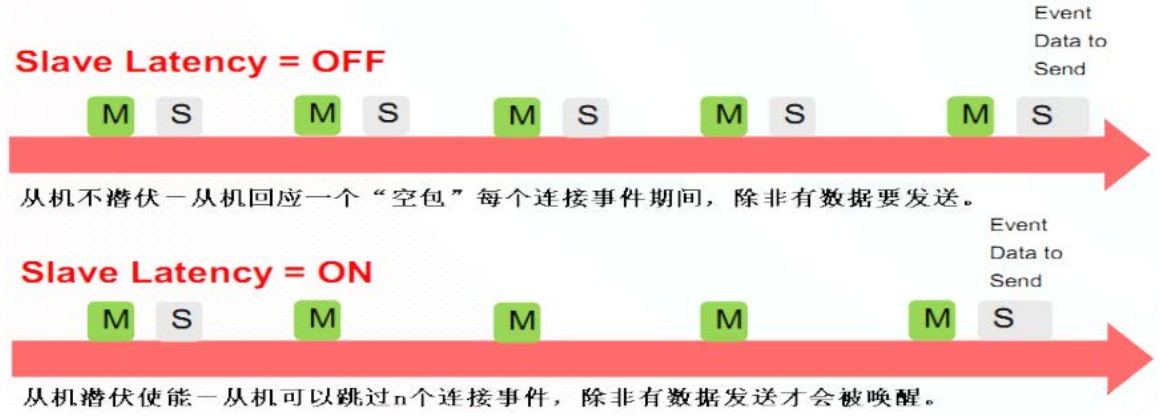
所有的通信都发生在两个设备的连接事件期间，连接事件周期的发生，按照连接参数指定的间隔联系，每个事件发生在某个数据通道（0~36），调频增量参数决定了下次连接时间发生的通道，在每个连接时间期间，Master 先发送，Slave 会在 150us 之后做出回应，即使一个连接事件发生（或两者），双方都没有数据发送（例外情况是从设备潜伏使能），这允许两个设备都承认对方仍然存在并保持活跃的连接。





5.1.11 Slave 的潜伏

潜伏，Slave 如果没有数据发送，允许跳过连接时间
连接参数中的 Slave 的潜伏值，是允许设备跳过的最大连接次数，在连接事件中，如果 slave 没有对 master 的包做出回应，master 将会在后来的连接时间中重复发送，知道 slave 回应。两个有效的连接事件之间的最大时间跨度（假设 slave 跳过了最大数目的连接时间）称为“有效连接间隔”
从设备的潜伏范围是 0~499，但是有效的连接间隔必须小于 32s



5.1.12 连接参数的设定

☆短间隔的连接事件：
两设备都会以高能耗运行
高数据吞吐量
发送等到时间短
☆长间隔的连接事件：
两设备都会以低能耗运行

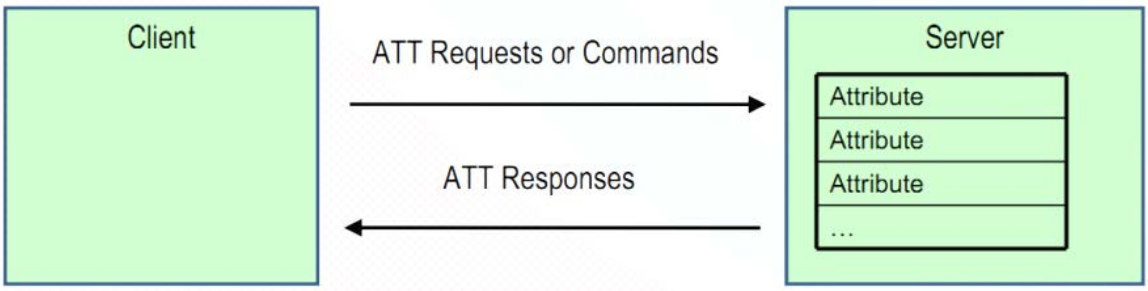
低数据吞吐量
发送等待时间长
☆低或者 0 潜伏值：
从设备以高能耗运行
从设备可以快速的收到来自中心设备的数据
☆高潜伏值：
外围设备在没有数据发送的情况下可以低能耗运行
外围设备无法及时收到来自中心设备的数据
中心设备能及时收到来自外围设备的数据

5.1.13 终止连接

Master 和 Slave 都可以主动断开连接
一边发起断开，另一边必须在断开连接之前回应这个断开请求。
监视超时而断开连接
监视超时参数都指定了两个数据包之间的最大时间跨度
监视超时时间必须大于有效连接间隔而小雨 32s
Slave 和 master 双方都维持着自己的监视超时计时器，在每次收到数据包时清零。
如果连接超时，设备会认为连接丢失，并且推出连接状态，返回广播，扫描或者待机模式。

5.1.14 ATT 的 Client/Server 架构

服务设备提供数据，客户端使用这些数据
服务端通过操作属性的方式，提供数据访问服务
设备的服务/客户角色，不依赖于 GAP 层中心设备/外围设备角色，和 LL 层的 master/slave 角色定义
一个设备可能同时做为一个客户端和服务端，而两个设备上的属性不会相互影响。



5.1.15 ATT 的 AttributeTable Example（属性表示例）

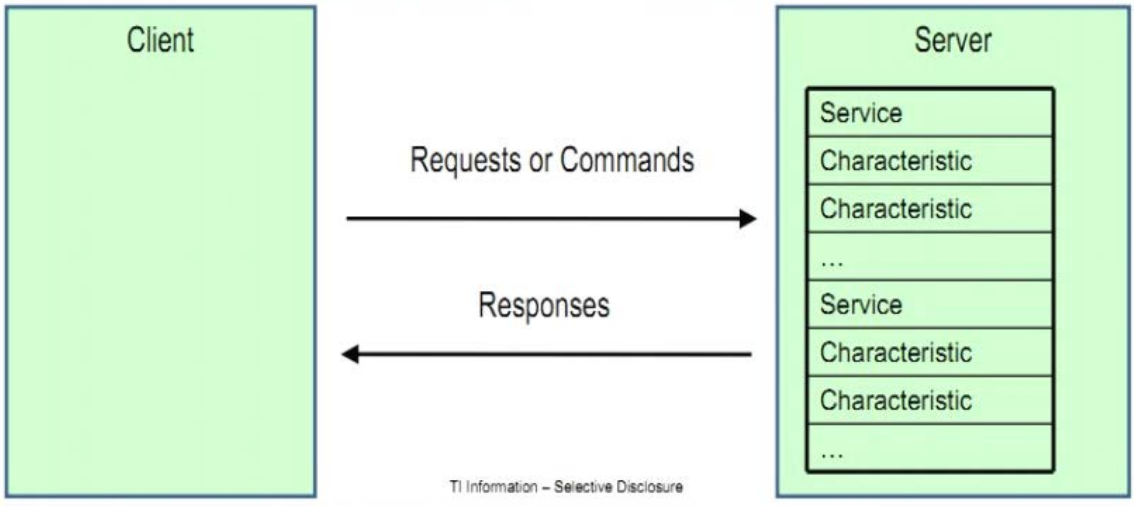
Handle，属性在列表中的地址
Type，说明代表什么数据，可以是 Bluetooth SIG 分配或者客户自定义的 UUID（统一识

别码，唯一性，通用性）
Permissions，权限，定义了 client 是否可以访问属性的值，以及特定的访问方式。

Handle	Type	Permissions	Value
39	0x2800 (GATT Service UUID)	Read	E0:FF (2 bytes)
40	0x2803 (GATT Characteristic UUID)	Read	10:29:00:E1:FF (5 bytes)
41	0xFFE1 (Simple Keys state)	(none)	00 (1 byte)
42	0x2902 (GATT Client Characteristic Configuration UUID)	Read and Write	00:00 (2 bytes)

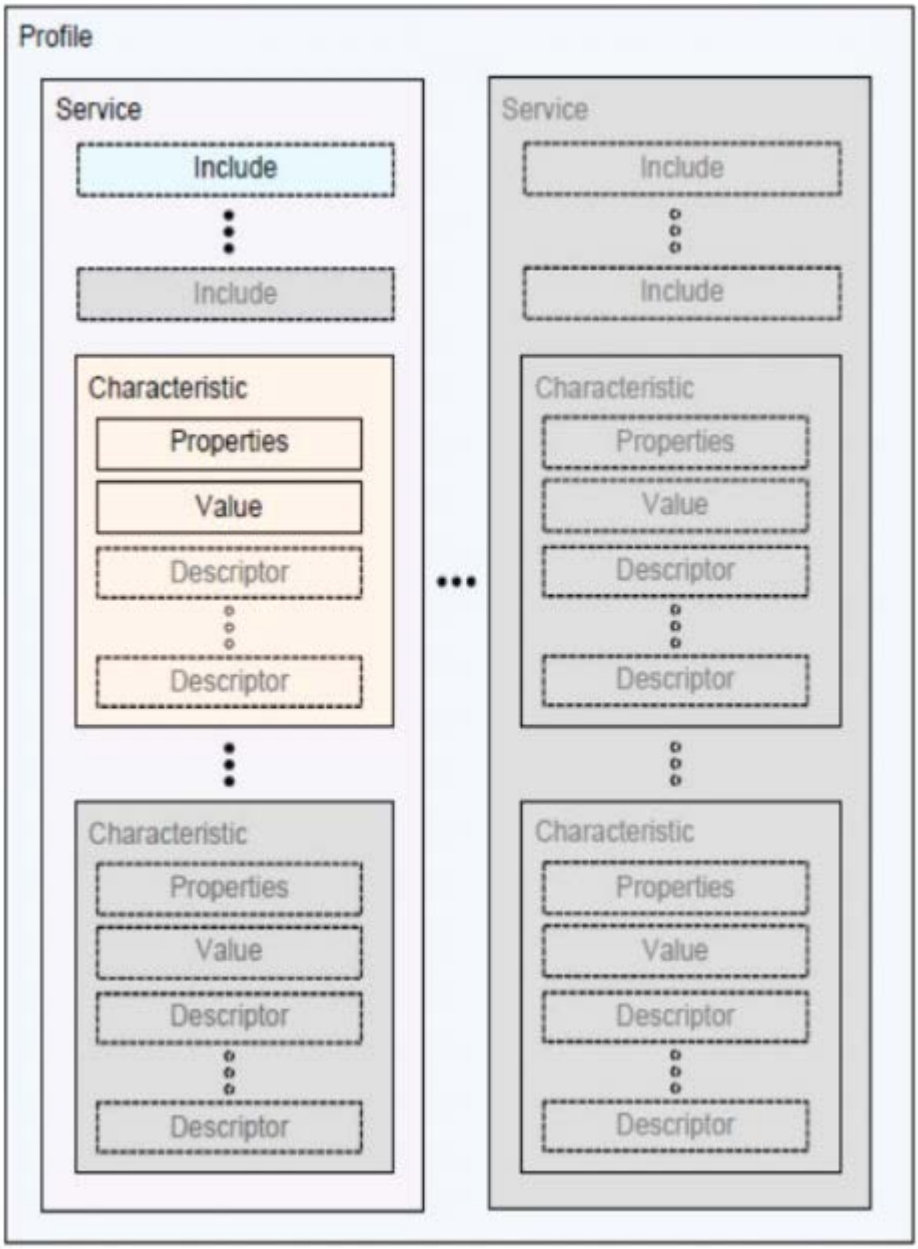
5.1.16 GATT 的 Client/Server 架构

GATT 指定了 profile 数据交换所在的结构
除了数据的封装方式不同，client server 和 Attribute 协议结构相同，数据封装在“Services”里，用“Characteristic”表示



5.1.17 GATT 的 Profile 层次结构

为了实现用户的应用，profile 通常有一个或者多个“Services”组成。
一个 service 或许包含某个特征值“characteristic values”，（例如，在一个温度采集设备中，通常会包含一个温度的特征值）
每一个特征值必须有占用一个特征申明结构，其中包括他的其他特性，它是服务端和客户端共享的读写空间
这个特征值可以包含一个可选的描述（descriptor 字串），来只是这个特征值的含义。



5.1.18 GATT Service Example

Service 起始于 handle 39，用 0x2800 来指示这个起始位置，这个 0x2800 是 Bluetooth SIG 的相关数据手册定义的，作为 GATT Service 的 UUID。
Handle 39 所指向的属性值为 0xFFE0，这个是用户自动以 profile 中的按键服务 UUID（这里仅仅是一个例子，0xFFE0 可能已经被 Bluetooth SIG 使用）
这个按键服务包含了所有后面的属性，知道下一个服务 UUID 定义处，或者到服务列表的结尾处，在这个例子中，按键服务的最后的属性所在地址为 handle 42，因此新的服务开始位置回事 handle 43

Handle	Type	Permissions	Value
39	0x2800 (GATT Primary Service UUID)	Read	E0:FF (2 bytes) (0xFFE0 = Simple Keys Service custom UUID)
40	0x2803 (GATT Characteristic Declaration UUID)	Read	10:29:00:E1:FF (5 bytes) (0xFFE1 = Simple Keys Value custom UUID) (0x0029 = handle 41) (0x10 = characteristic properties: notify only)
41	0xFFE1 (Simple Keys state)	(none)	00 (1 byte) (value indicates state of keys)
42	0x2902 (GATT Client Characteristic Configuration UUID)	Read and Write	00:00 (2 bytes) (value indicates whether notifications or indications are enabled)
43	0x2800 (GATT Primary Service UUID)	Read	A1:DD (2 bytes) (0xDDA1 = Other Service custom UUID)

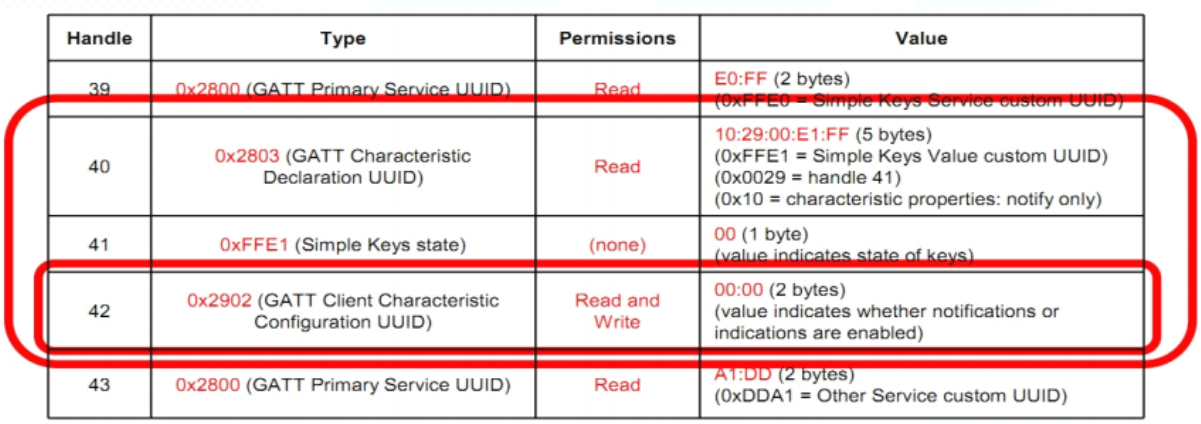
5.1.19 GATT 的 Characteristic Declaration

Handle 40 是一个特征值的声明，用 0x2803 来指示，这个 0x2803 同样也是 Bluetooth SIG 的相关数据手册定义的，作为 GATT Characteristic Declaration 的 UUID
特征值的属性值包含 5 个字节的长度 10：29：00：E1：FF
*0xFFE1，表明特征值的属类型（0xFFE1：客户自定义特征值的 UUID）
*0x0029，是这个值所保存的位置 handle（0x0029=41）
*0x10，表明这个特征值的操作权限 0x10：notify only

Handle	Type	Permissions	Value
39	0x2800 (GATT Primary Service UUID)	Read	E0:FF (2 bytes) (0xFFE0 = Simple Keys Service custom UUID)
40	0x2803 (GATT Characteristic Declaration UUID)	Read	10:29:00:E1:FF (5 bytes) (0xFFE1 = Simple Keys Value custom UUID) (0x0029 = handle 41) (0x10 = characteristic properties: notify only)
41	0xFFE1 (Simple Keys state)	(none)	00 (1 byte) (value indicates state of keys)
42	0x2902 (GATT Client Characteristic Configuration UUID)	Read and Write	00:00 (2 bytes) (value indicates whether notifications or indications are enabled)
43	0x2800 (GATT Primary Service UUID)	Read	A1:DD (2 bytes) (0xDDA1 = Other Service custom UUID)

5.1.20 GATT 的 Characteristic Configuration

另外作为特征值声明，可以有一个可选的描述信息
这个例子中，handle 42 包含了特征值的配置信息，0x2902，这个值同样也是 Bluetooth SIG 的相关数据手册定义的，作为 GATT Client Characteristic Configuration 的 UUID。
这个配置值有读写权限，意味着，GATT 客户端可以改变这个值。
如果把这个值（通知开关使能）从 0x0000（Notification off）改为 0x0001（notification on），GATT 服务器将开始发送这个特征值的通知到 GATT 客户端。

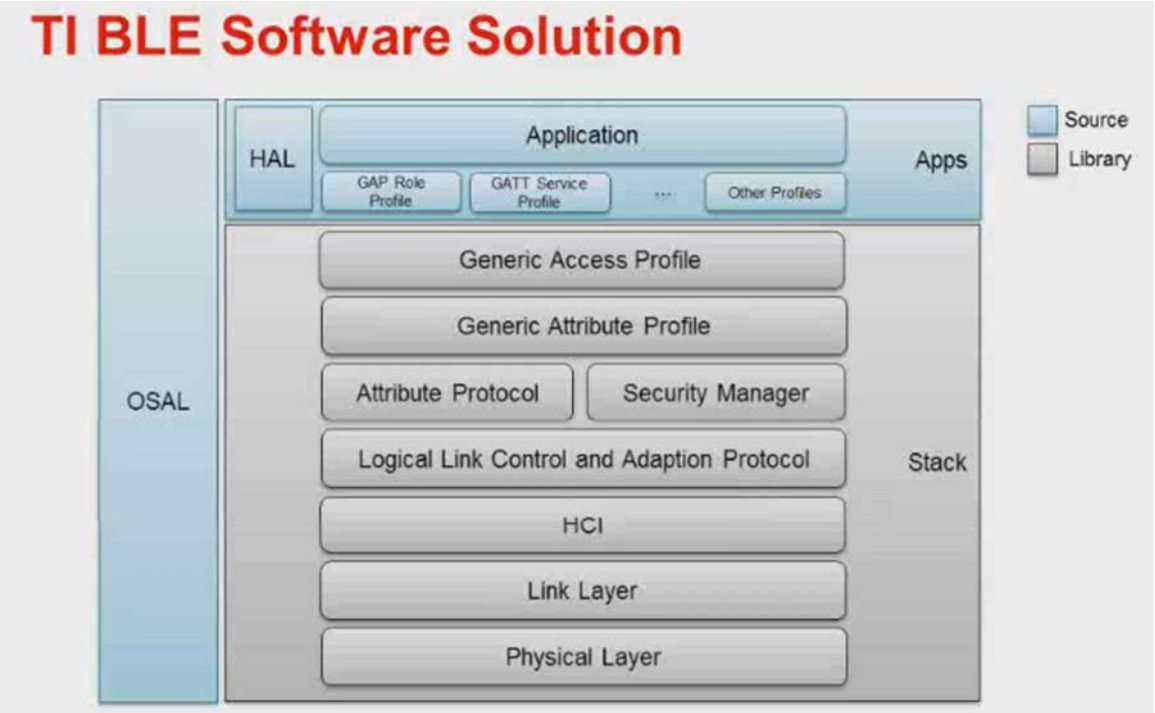


Handle	Type	Permissions	Value
39	0x2800 (GATT Primary Service UUID)	Read	E0:FF (2 bytes) (0xFFE0 = Simple Keys Service custom UUID)
40	0x2803 (GATT Characteristic Declaration UUID)	Read	10:29:00:E1:FF (5 bytes) (0xFFE1 = Simple Keys Value custom UUID) (0x0029 = handle 41) (0x10 = characteristic properties: notify only)
41	0xFFE1 (Simple Keys state)	(none)	00 (1 byte) (value indicates state of keys)
42	0x2902 (GATT Client Characteristic Configuration UUID)	Read and Write	00:00 (2 bytes) (value indicates whether notifications or indications are enabled)
43	0x2800 (GATT Primary Service UUID)	Read	A1:DD (2 bytes) (0xDDA1 = Other Service custom UUID)

5.1.21 GATT 的 Client Commands

当两个 BLE 设备处理连接状态，客户端和服务端设备的通信方式：
Discover Characteristic by UUID，搜索服务端设备所能提供的所有匹配 UUID 规范的属性
Read Characteristic Value，使用指定的 handle 读特征值。
Write Characteristic Value，使用指定的 handle 写特征值。
除此之外，如果通知被只能，服务设备会自动像客户端设备发出下列信息：
Notification，某个特征值被发送到客户端设备，而没有被读请求，并且不需要应答。
Indication，某个特征值被发送到客户端，没有被读请求的情况，但是在其他数据被发送之前必须被确认。

5.2 TI-BLE 协议栈简介



在上一节，我们详细介绍了 BLE 技术，其实在初学时，可以统统不要考虑，因为 TI 的 BLE 协议栈已经做好全部，我们入门时只需要利用 TI 提供的 demo，学习他，修改他，等到了解之后，再去深入学习。

BLE 技术是 Bluetooth SIG 规定的一套通信协议，在协议变成具体的代码之前，都只存在文档中，TI、Nordic、CSR 等厂商，根据 SIG 发布的 BLE 技术协议，配合自身的芯片开发了一整套源码，并且这套源码经过了 SIG 的测试，服务 BLE 协议。这套源码就叫做协议栈，协议栈是协议的实现。不同的芯片厂商都有各自的协议栈。
我们下面开始具体的 ble 协议栈的介绍，使用 TI 的 BLE 协议栈 1.3.2 版本。
当我们对具体的协议栈有了形象的认识后，在回过头来看上一节的 BLE 技术。

协议栈 demo 分类
在 TI 提供的协议栈里 demo 程序中，可以分为两类

第一类 单芯片方案

我们上一章演示的 demo 均属于这一类，控制器、主机、配置文件 profile、以及需要我们修改的底层代码都在一片 CC254x 里实现，这是最简单也是最常见的配置，这种方式提供最低的成本和功耗，所以我们学习 ble 的大部分时间里也是在学习这一类的 demo。

第二类 网络处理器

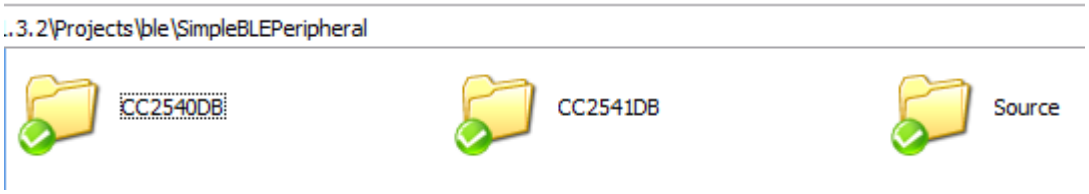
还记得前面演示 btool 软件连接从机的实验吗，使用的 usbdongle 里面烧写的 HostTestRelease 程序就属于第二类，他把控制器和主机部分在 CC254x 上实现，而应用程序和 Profile 在另外一个设备执行，应用程序和 profiles 通过厂商特定的 HCI 命令与

CC2540 通信，这一过程，使用了 CC2540USBDongle 的 USB 模拟的串口。

协议栈 demo 工程目录结构

还记得我们之前在开发板里烧写的两个协议栈例程吗，他们是 SimpleBLECentral 主机程序和 SimpleBLEPeripheral 从机程序，这两个是最简单也是最有代表性的 ble 协议栈 demo，下面我们来认真的介绍一下他们：

以 SimpleBLEPeripheral 从机工程为例。进入 SimpleBLEPeripheral 工程目录，一般的 demo 工程都会有下面三个目录：



目录【CC2540DB】

如果使用的是 CC2540 的开发板，则请双击运行该目录下的：【SimpleBLEPeripheral.eww】IAR 工程。

目录【CC2541DB】

如果使用的是 CC2541 的开发板，则请双击运行该目录下的【SimpleBLEPeripheral.eww】IAR 工程。

目录【Source】

SimpleBLEPeripheral 从机相关的 source 源码。绝大部分的应用中，只需要 souce 下的源文件即可。也请注意，只有 source 目录下的源文件是可以任意修改而不会影响其他的 demo 工程。

IAR Project 结构

打开 IAR 后如下图：

在 IAR 工程的左侧有很多文件夹，如 App、HAL、LIB 等，如图 5-2 所示，这些文件夹下面包含了很多源代码，这种实现方式与上述 BLE 协议栈结构是一一对应的，尽量将实现某些功能的函数放在同一个文件夹下。

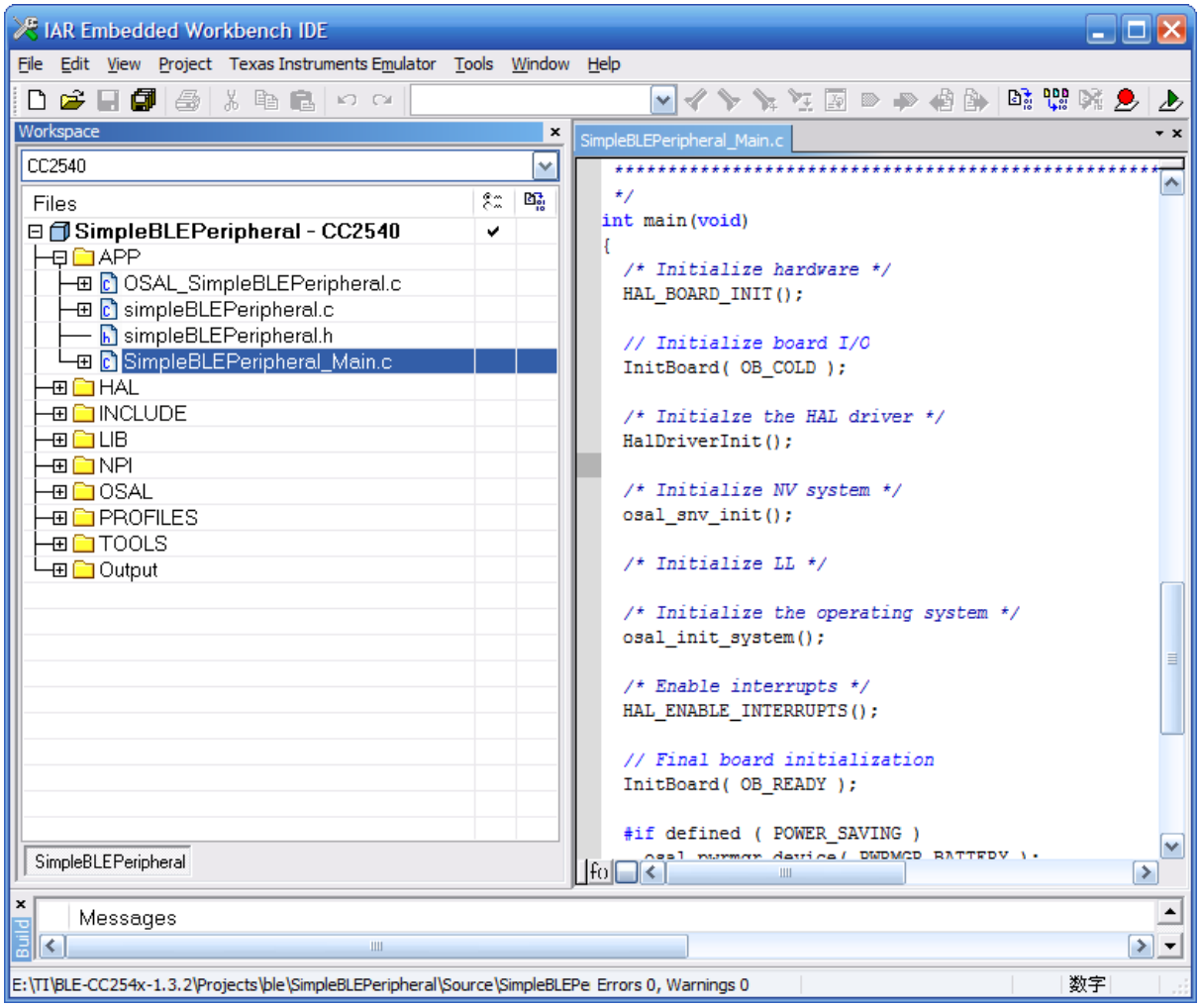


图 5-2

需要注意的有两个地方

Configuration 列表

Configuration 允许同一个 IAR 工程可以有多重配置，比如针对不同的开发板需要包含不同的底层驱动文件，使用不同存储空间的 CC2540 需要不同的内存配置等。

workspace 的下拉列表

很多用户反映说，已经替换 hal_lcd.c 文件，而且重新编译烧写了，lcd 上却没有显示。原因很可能出在这里，选错了配置。

CC254xDK-MINI Keyfob

从名字上应该看出来，这个配置的程序应当烧写到 keyfob 开发板上，而非 SmartRF 开发板。即便是在 Option 的宏预处理里添加了 HAL_LCD=TURE，同样是不会显示的。这里一定要注意。

CC254x

这个配置对应的是 SmartRF 平台的开发板（包含 SmartRF 和 SmartRF-BB）

其他配置

其他的不用理会（我们使用的芯片全部是 256K Flash 版本）

Files 中的文件分类

这里的各个文件夹文类是逻辑上的分类，这些文件夹只存在 IAR 的工程文件中，在磁盘上并不存在。

群组【APP】

与用户密切相关的源码，也就是在先前 Souce 文件夹的文件。

群组【HAL】

当前配置中所包含的 hal 层的驱动文件。比如底层的 lcd 驱动，led 驱动等。

群组【INCLUDE】

ble 底层全部是以 lib 库的形式提供，include 目录下是包含的 lib 库对应的头文件。

群组【LIB】

当前配置中用到的蓝牙 lib 库，TI 提供了多个 lib

群组【OSAL】

操作系统抽象层，应用层的编程基础是 osal，所有代码均建立在 osal 的框架上。

群组【PROFILES】

当前配置所用到的 profile 的全部相关文件。

群组【TOOLS】

工程配置，请不要随意修改，会影响其他 demo 工程。

BLE 协议栈执行流程

整个协议栈是从哪里开始执行的呢？请观察图 5-2 中，在 App 文件夹下有个 SimpleBLEPeripheral_Main.c 文件，打开该文件可以找到 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 main()函数主要做了哪些工作，main 函数原型如下：

```
00074: int main(void)
00075: {
00076:   /* Initialize hardware */
00077:   HAL_BOARD_INIT();
00078:
00079:   // Initialize board I/O
00080:   InitBoard( OB_COLD );
00081:
00082:   /* Initialze the HAL driver */
00083:   HalDriverInit();
00084:
00085:   /* Initialize NV system */
00086:   osal_snv_init();
00087:
00088:   /* Initialize LL */
00089:
00090:   /* Initialize the operating system */
00091:   osal_init_system();
00092:
00093:   /* Enable interrupts */
00094:   HAL_ENABLE_INTERRUPTS();
00095:
00096:   // Final board initialization
00097:   InitBoard( OB_READY );
00098:
00099:   #if defined ( POWER_SAVING )
00100:     osal_pwrmgr_device( PWRMGR_BATTERY );
00101:   #endif
00102:
00103:   /* Start OSAL */
00104:   osal_start_system(); // No Return from here
00105:
00106:   return 0;
00107: }
```

在 mian()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 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 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 BLE 协议栈 OSAL 介绍

BLE 协议栈包含了 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，(其实从 zigbee 协议栈开始就加入了 OSAL)，BLE 协议栈内加入了实时操作系统(并非真正意义上的操作系统)，称为 OSAL(操作系统抽象层，Operating System Abstraction Layer)。

非计算机专业的读者对操作系统知识较为欠缺，但是 BLE 协议栈里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 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，先介绍操作系统有关的部分基础知识。

操作系统(OS)基本术语如下。

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) -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 PU 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；
- 禁止任务切换；
- 使用信号量。

其中，在 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 BLE 协议栈中，OSAL 主要提供如下功能：

- 任务注册、初始化和启动：
- 任务的同步、互斥：
- 中断处理；
- 存储器分配和管理。

5.3.2 OSAL 运行机理

如图 5-1 中从宏观上表现了 BLE 协议的结构，但并没有 OSAL 的踪迹。BLE 协议栈与 ZgBee 协议之间并不能完全画等号。BLE 协议栈仅仅是 BLE 协议的具体实现。

在 BLE 协议中可以找到使用 OSAL 的某些“根源”。在基于 BLE 协议栈的应用程序开发过程中，用户只需要实现应用层的程序开发即可。从底层链路层到上面的 GATT，以及最顶层的用户层，每层都需要执行某些任务，因此，需要一个机制来实现任务的切换、同步和互斥，这就是 OSAL 产生的根源。

因此，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有一定区别的，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 并不能称为真正意义上的操作系统。

为了探究 OSAL 运行机理，请读者回顾一下第 4 章讲解的主从数据传输实验：在左边工作空间中口打开 App 文件夹，可以看到四个文件，分别是 OSAL_SimpleBLEPeripheral.c，simpleBLEPeripheral.c，simpleBLEPeripheral.h ，SimpleBLEPeripheral_Main.c 整个程序所实现的功能都包含在这四个源文件中。

打开 simpleBLEPeripheral.c 文件，可以看到两个比较重要的函数 SimpleBLEPeripheral_Init 和 SimpleBLEPeripheral_ProcessEvent，SimpleBLEPeripheral_Init 是任务的初始化函数，SimpleBLEPeripheral_ProcessEvent 是用户级的任务函数，主体部分是判断事件类型，然后执行相应的事件处理工作。

那么，事件和任务的时间处理是如何联系起来的呢？

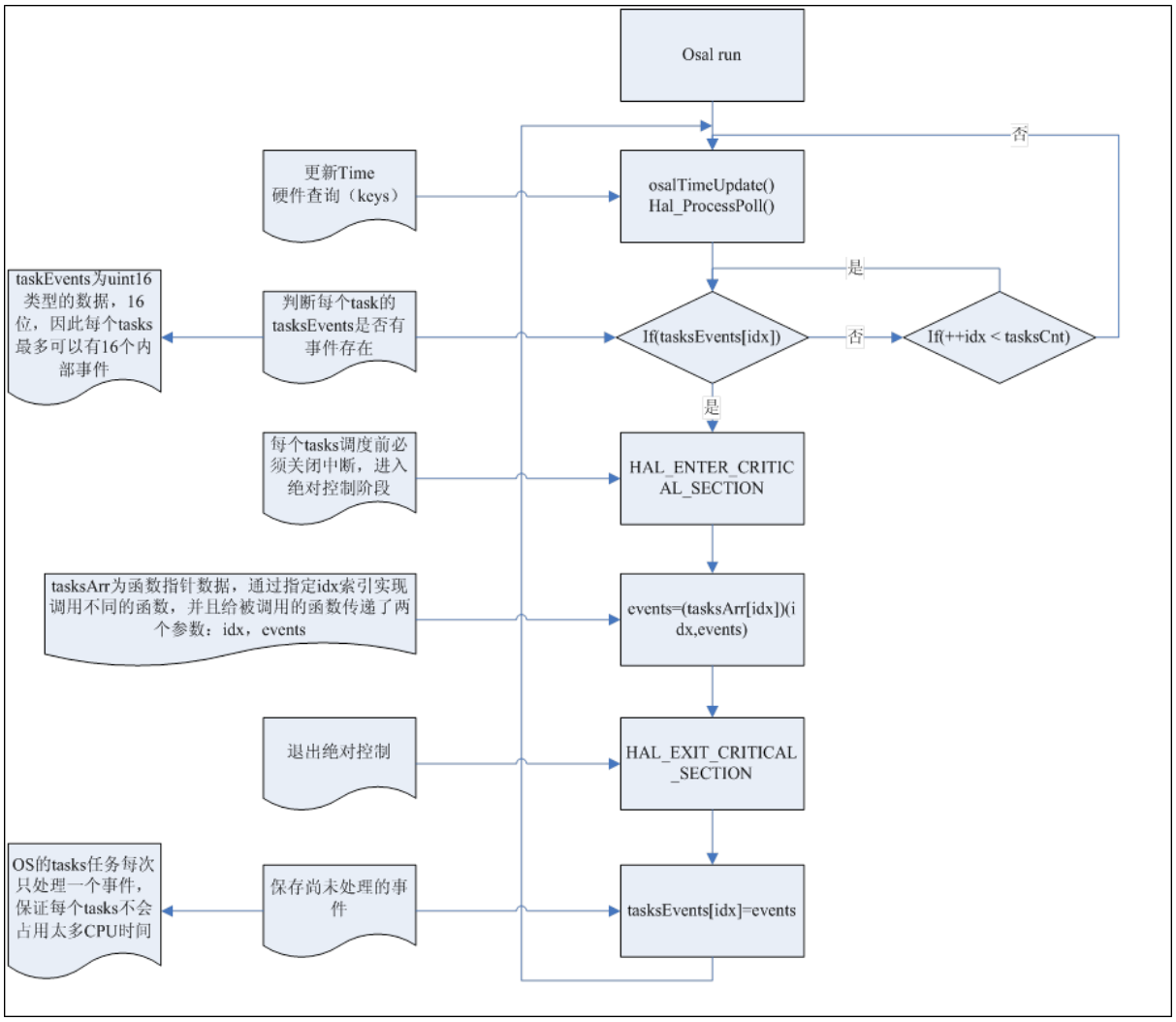
BLE 中采用的方法是：建议一个事件表，保存各个任务的对应的时间，建立另一个函数，保存每个任务函数的地址，然后将这两张表建立某种对应关系，当某一时间发生时，则查找函数表，找到对应的任务函数的指针，然后通过函数指针，调用该函数即可。

如下图，将 osal 简单的理解为一个大循环的可以如下图所示，首先 osal 会初始化每个任务函数的 init 函数，例如 SimpleBLEPeripheral_Init ()，初始化结束后，开始进入任务函数的循环，只不过，并不是每个任务函数都执行一次，而是通过一个叫做 task event 的标识来判断时候需要执行该任务函数。

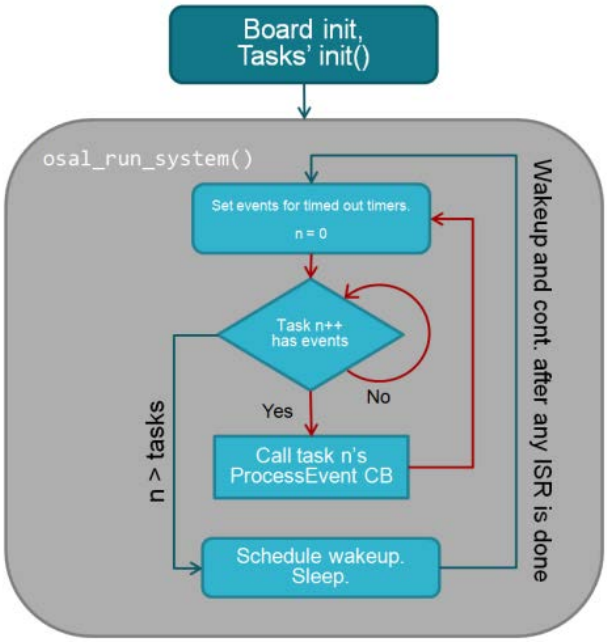
在 OSAL_RUN 中，也就是 osal_run_system()函数中，每次循环前都会判断 tasksEvents，osal_run_system()函数原型如下：

```
01052: void osal_run_system( void )
01053: {
01054:   uint8 idx = 0;
01055:
01056:   osalTimeUpdate();
01057:   Hal_ProcessPoll();
01058:
01059:   do {
01060:     if (tasksEvents[idx])   // Task is highest priority that is ready.
01061:     {
01062:       break;
01063:     }
01064:   } while (++idx < tasksCnt);
01065:
01066:   if (idx < tasksCnt)
01067:   {
01068:     uint16 events;
01069:     halIntState_t intState;
01070:
01071:     HAL_ENTER_CRITICAL_SECTION(intState);
01072:     events = tasksEvents[idx];
01073:     tasksEvents[idx] = 0;   // Clear the Events for this task.
01074:     HAL_EXIT_CRITICAL_SECTION(intState);
01075:
01076:     activeTaskID = idx;
01077:     events = (tasksArr[idx])( idx, events );
01078:     activeTaskID = TASK_NO_TASK;
01079:
01080:     HAL_ENTER_CRITICAL_SECTION(intState);
01081:     tasksEvents[idx] |= events;   // Add back unprocessed events to the cu
01082:     HAL_EXIT_CRITICAL_SECTION(intState);
01083:   }
01084: #if defined( POWER_SAVING )
01085:   else   // Complete pass through all task events with no activity?
01086:   {
01087:     osal_pwrmgr_powerconserve();   // Put the processor/system into sleep
01088:   }
01089: #endif
01090:
01091:   /* Yield in case cooperative scheduling is being used. */
01092: #if defined (configUSE_PREEMPTION) && (configUSE_PREEMPTION == 0)
01093:   {
01094:     osal_task_yield();
01095:   }
01096: #endif
01097: } ? end osal_run_system ?
```

我们根据上面的代码执行流程绘制了一张流程图，如下所示。



简化如下，这种机制的好处在于，不必浪费大量的时间空等待，这对于资源有限的单片机来说是一种非常不错的编程方式。大家完全可以将 BLE 协议栈里的 osal 一直到自己常用的 mcu 上，这样，不同的项目，也会有很多可以直接复用大代码，节约时间，提高效率。



OSAL task loop

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：主机 GATT 接收到数据后，会产生 GATT_MSG_EVENT 消息，但是任务的事件处理函数在处理这个事件的时候，还需要得到所收到的数据。

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。

5.3.4 OSAL 添加新任务

在使用 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。

打开 OSAL_SimpleBLEPeripheral.c 文件，可以找到数组 taskArr[]和函数 osalInitTasks()。

taskArr 数组里存放了所有的任务函数指针，这里有一个概念，函数指针，与变量指针类似，通过地址就可以仿真变量的值，而通过函数指针就可以调用该函数，所有的回调函数均是通过函数指针调用。

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

osalInitTasks 函数是 osal 的任务初始化函数，所有的任务的初始化工作都在这里面完成，并且给每个任务分配一个任务 id，就是 taskID。

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
- 新任务的时间处理函数。

例如 SimpleBLEPeripheral 的 SimpleBLEPeripheral_Init()和 SimpleBLEPeripheral_ProcessEvent()。

5.3.5 OSAL 应用编程接口

Osal 提供了丰富的编程接口 API 函数，我们只需要了解 API 如何调用就可以驾驭 osal。总体而言，osal 提供了 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；
- 非易失闪存管理。

下面，选取部分典型的 api 进行讲解。

消息管理 api。

Osal_msg_allocate()为消息分配缓存空间，分配之后，可以填充消息，然后经过 osal_msg_sned()将消息发送出去，然后任务函数中通过 osal_msg_reveive()函数接收属于自己的消息，并处理，最后调用 osal_msg_deallocate()函数销毁由 Osal_msg_allocate()分配的内存空间。

5.3.5.1 任务同步 api

当我们需要立刻启动一个事件时，调用 osal_set_event()，调用后立即产生一个事件，等到 osal 下次循环时，就会判断到这个事件，然后调用相应的任务函数处理该事件。如果想过一段时间在启动一个事件，或者说，我需要延时 5 秒然后查询一下 gpio 的状态，这时就需要 osal_start_timerEx 来创建一个定时器事件，当设定的时间到达后，就会产生某个事件。加入刚才已经创建了一个定时器事件，但是中途改变主意了，不需要去轮询 gpio 状态，就可以调用 osal_stop_timerEx()来停止已经启动的定时器。

5.3.5.2 内存管理 api

单片机的内存资源非常有限，CC254X 内部只有 8K 的 ram，osal 提供了一种“静态内存池”的技术，简单地讲系统运行的时候开辟了一个非常大的数组，然后提供一些接口给用户，当用户需要使用内存的时候，调用分配 api 取得一些内存，等到用户使用完之后，不在需要这个内存的时候，在调用销户内存的 api，释放先前占用的内存，这样最大程度复用了有限的内存空间。其实我们仔细想想，平时编程的时候，通常的做法时设立一个全局的变量，其实这个变量并不是从头到尾一直在使用，这样就造成了浪费，因为你不用的时候，别人也无法使用这部分的空间。
osal_mem_alloc()用来分配内存，用完之后，通过调用 osal_mem_free 释放内存。

5.3.5.3 非易失闪存管理。

任何一个工程项目，均会有一些参数需要掉电保存，以前，我们通常是在单片机外面挂一个 eeprom 设备通过 i2c 接口来存储参数。而在 2530 里，osal 提供了内部 flash 的存储管理接口。通过调用这些接口，就可以掉电保存参数了。
我们首先需要为要保存的数据创建一个唯一的 item 标识，用来区分其他的已经保存的数据的 item 标识，然后调用 osal_nv_write()函数写入参数，等到需要的时候可以调用 osal_nv_read()来读取上一次保存的值。

5.3.6 OSAL 使用范例分析

在这一节，我们分析来分析应用程序使用 OSAL 层的两种最常见的范例分析，将它的来龙去脉展现给用户。

以 SimpleBLEPeripheral 为例，一种是用户事件的启动和接收，第二种是消息的发送和接收。

5.3.6.1 用户事件的启动和接收

在 Z-Stack 协议栈的用户程序中，为了实现特定的功能，除了使用系统事件外，用户自定义一些用户事件是非常有必要的。比如要周期性的去检测某个 GPIO 的状态，或者每隔一段时间发送一个命令等，都需要自定义事件来完成。

在 SimpleBLEPeripheral 中，常见的用户事件

```
// Simple BLE Peripheral Task Events
#define SBP_START_DEVICE_EVT                              0x0001
#define SBP_PERIODIC_EVT                                  0x0002
#define SBP_ADV_IN_CONNECTION_EVT                         0x0004
```

每个任务函数都类似这样的格式：

```
uint16 SimpleBLEPeripheral_ProcessEvent( uint8 task_id,
{

  if ( events & SYS_EVENT_MSG )
  {
    uint8 *pMsg;
    // return unprocessed events
    return (events ^ SYS_EVENT_MSG);
  }

  if ( events & SBP_START_DEVICE_EVT )
  {
    return ( events ^ SBP_START_DEVICE_EVT );
  }

  if ( events & SBP_PERIODIC_EVT )
  {
    return (events ^ SBP_PERIODIC_EVT);
  }
  // Discard unknown events
  return 0;
} ? end SimpleBLEPeripheral_ProcessEvent ?
```

每一个 if 语句均对应一个 EVENT 事件。首先处理的是 SYS_EVENT_MSG 系统消息事件。其次是用户自定的事件，注意，每个任务函数每次只处理一个事件（注意每个 if 中的 return），也就是说本次处理完了 SYS_EVENT_MSG 事件会直接 return。然后准备下一次 OSAL 层任务调度到该任务时再处理用户自定义的 SBP_START_DEVICE_EVT，如果此时正好又发生了一个 SYS_EVENT_MSG 系统消息事件，那么将继续先处理 SYS_EVENT_MSG，直到空闲的时候才会去处理任务自定义的事件。

留心的读者应该发现了一个问题，这个 if 框架是处理事件的，也就是说事件已经发了才会来处理，那么事件是在哪发起的呢？

我们先寻找 SBP_START_DEVICE_EVT 事件的启动。

在 SimpleBLEPeripheral_Init 任务初始化函数结尾处启动了该事件。

```
void SimpleBLEPeripheral_Init( uint8 task_id )
{
  simpleBLEPeripheral_TaskID = task_id;
```

。。。

```
// Setup a delayed profile startup
osal_set_event( simpleBLEPeripheral_TaskID, SBP_START_DEVICE_EVT );
```

osal_set_event()函数是事件启动函数，参数 1 是需要启动定时器的任务 ID，第二个参数是启动的事件 ID，调用改函数后会立刻出发启动的事件，接着在该任务函数中就会接收到 SBP_START_DEVICE_EVT 事件，进而进入下列代码中执行：

```
if ( events & SBP_START_DEVICE_EVT )
{
  // Start the Device
  VOID GAPRole_StartDevice( &simpleBLEPeripheral_PeripheralCBs );

  // Start Bond Manager
  VOID GAPBondMgr_Register( &simpleBLEPeripheral_BondMgrCBs );

  // Set timer for first periodic event
  osal_start_timerEx( simpleBLEPeripheral_TaskID, SBP_PERIODIC_EVT,

  return ( events ^ SBP_START_DEVICE_EVT );
}
```

对该事件作出何种相应，完全由用户决定。注意，定时器定时均为一次性的，不会自动装载。

至于 osal_set_event 和 osal_start_timerEx 函数具体做了什么，我们将在后续的文档开发资料用详细的阐述，本文档只为 OSAL 编程做指导。OSAL 实现的细节不在讨论范围内。

5.3.6.2 MSG 消息的发送与接收

在 SimpleBLEPeripheral 任务初始化函数中有这样一条代码：

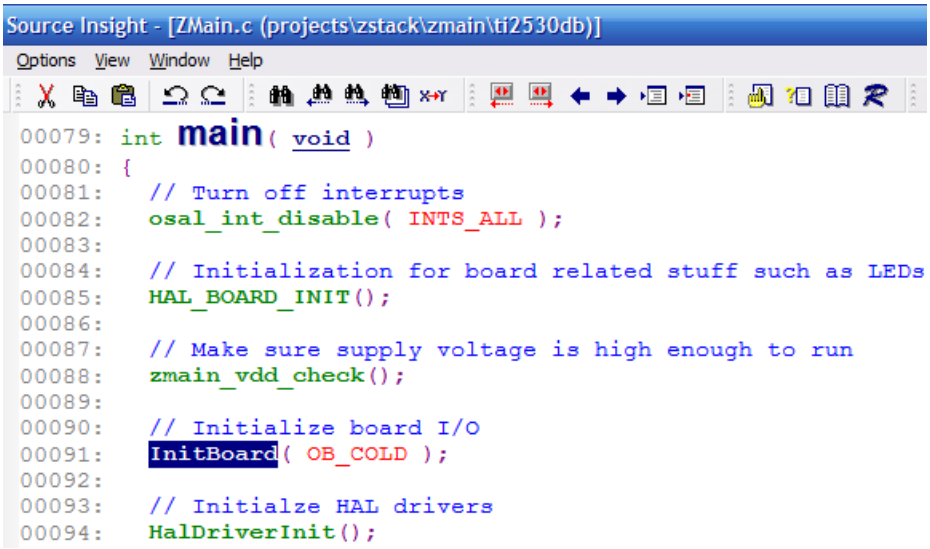
```
// Register for all key events - This app will handle all key events
RegisterForKeys( simpleBLEPeripheral_TaskID );
```

这个函数来自 OnBoard.c 源文件中

```
00188: uint8 RegisterForKeys( uint8 task_id )
00189: {
00190:   // Allow only the first task
00191:   if ( registeredKeysTaskID == NO_TASK_ID )
00192:   {
00193:     registeredKeysTaskID = task_id;
00194:     return ( true );
00195:   }
00196:   else
00197:     return ( false );
00198: }
```

向一个全局变量中赋值自己的任务 ID，从代码中可以看出，只能第一个调用改函数的任务才能成功注册到按键服务。那这个全局变量在何时使用呢？

我们暂且放下该问题，在 main.c 初始化过程有如下代码：




```
00079: int main( void )
00080: {
00081:   // Turn off interrupts
00082:   osal_int_disable( INTS_ALL );
00083:
00084:   // Initialization for board related stuff such as LEDs
00085:   HAL_BOARD_INIT();
00086:
00087:   // Make sure supply voltage is high enough to run
00088:   zmain_vdd_check();
00089:
00090:   // Initialize board I/O
00091:   InitBoard( OB_COLD );
00092:
00093:   // Initialze HAL drivers
00094:   HalDriverInit();
```

该函数的实现如下：

```
00124: void InitBoard( uint8 level )
00125: {
00126:   if ( level == OB_COLD )
00127:   {
00128:     // IAR does not zero-out this byte below the XSTACK.
00129:     *(uint8 *)0x0 = 0;
00130:     // Interrupts off
00131:     osal_int_disable( INTS_ALL );
00132:     // Check for Brown-Out reset
00133:     ChkReset();
00134:   }
00135:   else  // !OB_COLD
00136:   {
00137:     /* Initialize Key stuff */
00138:     HalKeyConfig(HAL_KEY_INTERRUPT_DISABLE, OnBoard_KeyCallback);
00139:   }
00140: }
```

该函数通过 HalKeyConfig 接口注册了一个按键回调函数 OnBoard_KeyCallback，首先看 HalKeyConfig 函数的实现：

将上述的回调函数的地址复制给了了函数指针变量。通过跟踪发现该函数指针变量在按键的轮询函数中调用，如下图：

```
00318: void HalKeyPoll (void)
00319: {
00320:   uint8 keys = 0;
00321:
00322:   if ((HAL_KEY_JOY_MOVE_PORT & HAL_KEY_JOY_MOVE_BIT))   /*
00323:   {
00324:     keys = halGetJoyKeyInput();
00325:   }
00326:
00327:   /* If interrupts are not enabled, previous key status an
00328:    * are compared to find out if a key has changed status.
00329:    */
00330:   if (!Hal_KeyIntEnable)
00331:   {
00332:     if (keys == halKeySavedKeys)
00333:     {
00334:       /* Exit - since no keys have changed */
00335:       return;
00336:     }
00337:     /* Store the current keys for comparation next time */
00338:     halKeySavedKeys = keys;
00339:   }
00340:   else
00341:   {
00342:     /* Key interrupt handled here */
00343:   }
00344:
00345:   if (HAL_PUSH_BUTTON1())
00346:   {
00347:     keys |= HAL_KEY_SW_6;
00348:   }
00349:
00350:   /* Invoke Callback if new keys were depressed */
00351:   if (keys && (pHalKeyProcessFunction))
00352:   {
00353:     (pHalKeyProcessFunction) (keys, HAL_KEY_STATE_NORMAL);
00354:   }
00355: } ? end HalKeyPoll ?
```

回调函数的一个非常的做用是可以隔离每一层，而且还有一点点面向对象的开发方式，在这里底层的按键查询函数调用一个函数指针，而非具体的函数，这样就将处理按键的接口留给了上层，上层应用中，只需解析函数指针传入的参数 1：keys 就知道是哪个按键被按下了。

我们再回到刚才的 OnBoard_KeyCallback 回调函数处，该回调函数代码如下：

```
00242: void OnBoard_KeyCallback ( uint8 keys, uint8 state )
00243: {
00244:   uint8 shift;
00245:   (void)state;
00246:
00247:   shift = (keys & HAL_KEY_SW_6) ? true : false;
00248:
00249:   if ( OnBoard_SendKeys( keys, shift ) != ZSuccess )
00250:   {
00251:     // Process SW1 here
00252:     if ( keys & HAL_KEY_SW_1 )  // Switch 1
00253:     {
00254:     }
00255:     // Process SW2 here
00256:     if ( keys & HAL_KEY_SW_2 )  // Switch 2
00257:     {
00258:     }
00259:     // Process SW3 here
00260:     if ( keys & HAL_KEY_SW_3 )  // Switch 3
00261:     {
00262:     }
00263:     // Process SW4 here
00264:     if ( keys & HAL_KEY_SW_4 )  // Switch 4
00265:     {
00266:     }
00267:     // Process SW5 here
00268:     if ( keys & HAL_KEY_SW_5 )  // Switch 5
00269:     {
00270:     }
00271:     // Process SW6 here
00272:     if ( keys & HAL_KEY_SW_6 )  // Switch 6
00273:     {
00274:     }
00275:   } ? end if OnBoard_SendKeys(keys... ?
00276: } ? end OnBoard_KeyCallback ?
```

该函数又将按键值向下传递到 OnBoard_SendKeys()函数里，通过名字应该能够看出来，应该是发送了按键消息。果然是调用了 osal_msg_send 发送了一个消息，并且在消息中附加了消息事件名称 KEY_CHANGE，按键值，和状态，按键的处理权又向下传递，最后发送到谁第一次注册按键服务端任务函数中去。

```
00210: uint8 OnBoard_SendKeys( uint8 keys, uint8 state )
00211: {
00212:   keyChange_t *msgPtr;
00213:
00214:   if ( registeredKeysTaskID != NO_TASK_ID )
00215:   {
00216:     // Send the address to the task
00217:     msgPtr = (keyChange_t *)osal_msg_allocate( sizeof(keyChange_t) );
00218:     if ( msgPtr )
00219:     {
00220:       msgPtr->hdr.event = KEY_CHANGE;
00221:       msgPtr->state = state;
00222:       msgPtr->keys = keys;
00223:
00224:       osal_msg_send( registeredKeysTaskID, (uint8 *)msgPtr );
00225:     }
00226:     return ( ZSuccess );
00227:   }
00228:   else
00229:     return ( ZFailure );
00230: } ? end OnBoard_SendKeys ?
```

注册了按键服务的任务ID

最后又回到了 SimpleBLEPeripheral 任务函数中。最终用户按了哪个按键，如何响应该按键在系统事件 SYS_EVENT_MSG 中处理。疑问又来了，为什么通过 osal_msg_send 发送的消息会出现在 SYS_EVENT_MSG 中呢？这个疑问，我们将在下一节中解答。

```
static void simpleBLEPeripheral_ProcessOSALMsg( osal_eve
{
  switch ( pMsg->event )
  {
  #if defined( CC2540_MINIDK )
    case KEY_CHANGE:
      simpleBLEPeripheral_HandleKeys( ((keyChange_t *)pMsg)->state,
      break;
  #endif // #if defined( CC2540_MINIDK )

  default:
    // do nothing
    break;
  }
}
```

在 SimpleBLEPeripheral 中，对按键的响应如下：joystick left(SW2)出发广播的开启和关闭

```
if ( keys & HAL_KEY_SW_2 )
{

  SK_Keys |= SK_KEY_RIGHT;

  // if device is not in a connection, pressing the right key should toggle
  // advertising on and off
  if( gapProfileState != GAPROLE_CONNECTED )
  {
    uint8 current_adv_enabled_status;
    uint8 new_adv_enabled_status;

    //Find the current GAP advertisement 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;
    }

    //change the GAP advertisement status to opposite of current status
    GAPRole_SetParameter( GAPROLE_ADVERT_ENABLED, sizeof( uint8 ), &new_adv_ena
  }
```

以上，整个按键消息的发送和处理就结束了，现在读者应该比较清晰的理解的按键消息的处理过程，阅读协议栈代码比较重要的一点是：只跟踪函数肯定会跟丢，要求全局分析。Osal 层中使用了很多非常好的编程机制，值得我们好好的学习。

5.3.6.3 发送和接收自定义的 MSG 消息

启动和处理用户自定的 EVENT 事件是比较简单、容易理解的，那么如何发送和接收用户自定义的消息呢？上一节中也遗留了一个问题，也同样需要在这里回答，上一节的问题是为什么通过 osal_msg_send 发送的消息会出现在 SYS_EVENT_MSG 中呢？如果先前留意代码的读者应该知道了该问题的答案，看 osal_msg_send()函数的注释中色线标记的区域，该函数同时会向目标任务列表中发起一个 message ready event 事件，所以会出

现在 SYS_EVENT_MSG 中。在第一节中的第三个问题就描述这个问题。

```
00475: /*********************************************************************
00476:  * @fn      osal_msg_send
00477:  *
00478:  * @brief
00479:  *
00480:  *    This function is called by a task to send a command message to
00481:  *    another task or processing element.  The sending_task field must
00482:  *    refer to a valid task, since the task ID will be used
00483:  *    for the response message.  This function will also set a message
00484:  *    ready event in the destination tasks event list.
00485:  *
00486:  *
00487:  * @param   uint8 destination task - Send msg to?  Task ID
00488:  * @param   uint8 *msg_ptr - pointer to new message buffer
00489:  * @param   uint8 len - length of data in message
00490:  *
00491:  * @return  SUCCESS, INVALID_TASK, INVALID_MSG_POINTER
00492:  */
00493: uint8 osal_msg_send( uint8 destination_task, uint8 *msg_ptr )
00494: {
00495:   if ( msg_ptr == NULL )
00496:     return ( INVALID_MSG_POINTER );
00497:
00498:   if ( destination_task >= tasksCnt )
```

那么如何发送和接收用户自定义的消息呢？

在 ComDef.h 头文件中定义了先前看到的消息命令的宏定义，然后在稍微下面一点有这样的注释代码：OSAL 系统消息保留给用于应用程序的 IDs 和 EVENTs，为 0xE0~0xFC。

```
#define SYS_EVENT_MSG           0x8000  // A message is waiting event

/*********************************************************************
 * Global Generic System Messages
 */

#define KEY_CHANGE              0xC0    // Key Events

// OSAL System Message IDs/ Events Reserved for applications (user applications)
// 0xE0 ? 0xFC
```

这里要说明一点，消息往往用于不同任务函数之间的数据传递，在同一任务函数中使用全局函数或者自动以用户事件完全能够胜任。如果用户是在需要自定义用户消息，可以参考下列操作。

在 xxxApp.h 中宏定义用户 MSG 消息，注意取值范围为 0xE0~0xFC

使用 osal_msg_allocate 函数创建所要发送的消息的结构体空间，例如 KEY_CHANGE 消息：

```
msgPtr = (keyChange_t *)osal_msg_allocate( sizeof(keyChange_t) );
```

填充消息结构体成员，例如 KEY_CHANGE

```
msgPtr->hdr.event = KEY_CHANGE;
msgPtr->state = state;
msgPtr->keys = keys;
```

使用 osal_msg_send 函数想目标任务 ID 发送该消息

```
osal_msg_send( registeredKeysTaskID, (uint8 *)msgPtr );
```

最后在目标任务的任务处理函数中的处理刚才填充消息

```
MSGpkt = (afIncomingMSGPacket_t *)osal_msg_receive( SampleApp_Ta
while ( MSGpkt )
{
  switch ( MSGpkt->hdr.event )
  {
    // Received when a key is pressed
    case KEY_CHANGE:
      SampleApp_HandleKeys( ((keyChange_t *)MSGpkt)->state, ((ke
      break;
    case USR_MSG:
      //do something here

      break;
```

此时，应该完全体会到了 EVENT 与 MSG 消息的区别。

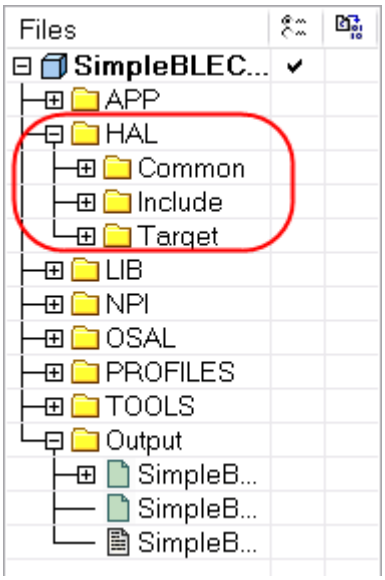
EVNET 用于同一任务函数传递命令，而 MSG 则用于不同的任务函数传递命令。

5.4 硬件抽象层 HAL

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

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 相关的头文件，hal 文件存放在三个不同的目录之下，如下图：



Common

文件夹包含协议栈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，主要包括下三个函数：

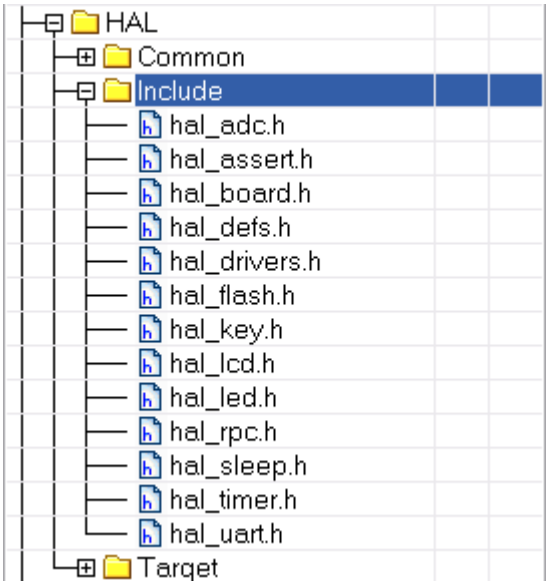
Hal_init()他由 osalTaskAdd 调用，以便在 osal 中注册 hal 的驱动

halDriverInit()他由 main()函数调用来初始化硬件驱动，读者可以更具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发运行时完。hal_ProcessEvent()这个函数用来处理 osal 中 hal 相关的驱动时间，例如按键、led、lcd、蜂鸣器、休眠定时

等。

Include

文件加包含 hal 驱动级 hal 相关文件的头文件，从文件名能看出各文件的作用。如下



Target

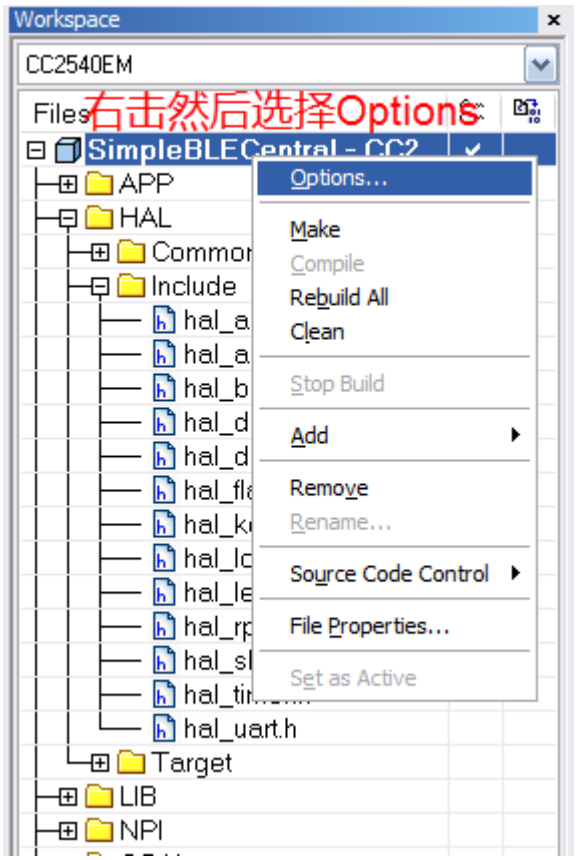
该文件夹包含了当前工程所使用的芯片及硬件平台，例如 CC2540EM、或者 CC2541ARC 等。

5.4.1 硬件抽象层驱动编译

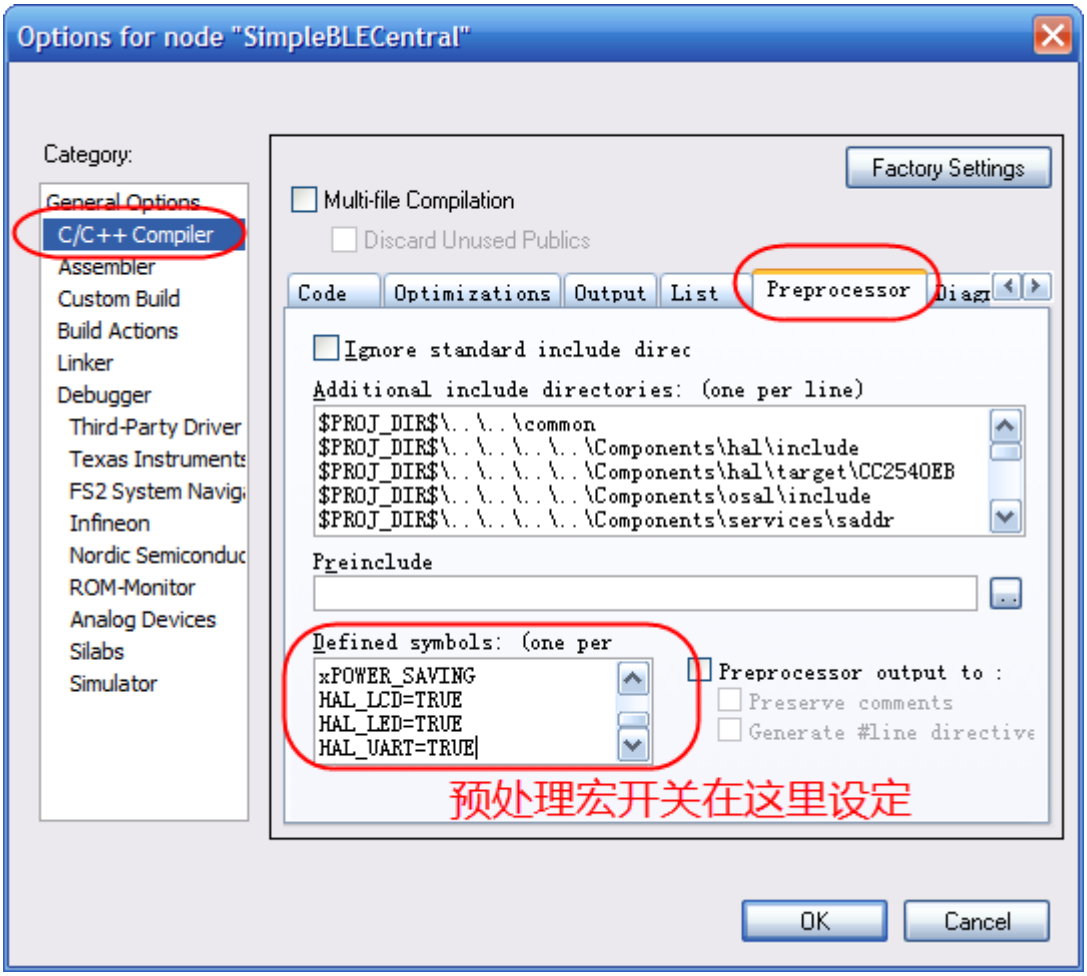
Ti 提供的 ble 协议栈及协议栈 demo 中，通常并不需要所有的硬件服务，因此 TI 提供了一些预编译宏定义，来决定是否添加相关的硬件驱动，常见的与编译符号如下：

HAL_ADC
HAL_LCD
HAL_LED
HAL_KEY
HAL_UART
HAL_DMA 等等

我们可以在工程选项设置中设置需要的驱动，以 SimpleBLECentral 工程为例，打开 Option 对话框，如下图：



如下图设定预处理宏定义开关



注意上图，在 POWER_SAVING 的前面有一个 x（英文字母 x 的小写），意思是该条宏定义不生效。

5.5 BLE 协议栈的 GAP 和 GATT

低功耗蓝牙里面，焦点几乎都在 GAP 和 GATT 上。

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，包括这杯发现、建立连接、终止连接、初始化安全管理，和设备配置，所以会在 ble 协议栈红发现有不少函数均是以 GAP 为前缀，这些函数会负责以上的内容。
GAP 层总是作为下面四个钟角色之一
☆Broadcaster 广播者——不可以连接的一直在广播的设备；
☆Observer 观测者——可以扫描广播设备，但不能发起建立连接的设备；
☆Peripheral 从机——可以被连接的广播设备，可以在单个链路层连接中中作为从机
☆Central 主机——可以扫描广播设备并发起连接，在单个链路层或多链路层中作为主机，在 TI 的 ble 协议栈中，一个主机可以连接三个从机。

在典型的蓝牙低功耗系统中，从机设备广播特定的数据，来是主机知道他是一个可以连接的设备，广播内容包括设备地址以及一些额外的数据，如设备名、服务等。主机收到广播数据后，会向从机发送扫描请求 Scan Request，然后从机将特定的数据回应给主机，称为扫描回应 Scan Response。主机收到扫描回应后，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，主机可以向从机发起建立连接的请求，连接请求包括下面一些参数。

△连接间隔——在两个 BLE 设备的连接中使用调频机制，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，两个设备任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时间之间的时间间隔，连接间隔以 1.25ms 为单位，连接间隔的值为 6（7.5ms）~3200（4s）。

不同的应用可以要求不同的时间间隔，长的时间间隔的优势是显著的节省功耗，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，可以更快的收发数据，不利之处是设备因连接时间的到来而频繁的唤醒，会有较多的功耗。

△从机延时——这个参数的设置可以使从机跳过若干连接时间，这个从机更多的灵活度，如果它没有数据发送时，可以选择跳过连接时间继续休眠，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的间隔，如果超过了这个时间而没有成功的连接时间，设备被认为丢失连接，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+从机延时）]。

从机可以通过向主机发送“连接参数更新请求”来改变连接设置，主机接收到请求后，可以选择接受或者拒绝这些新的参数。

连接可以被主机或者从机以任何原因主动终止。当一方发起终止链接时，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 连接中的安全管理。只有已认证的连接中。特定的数据数据才能被读写，一旦连接建立，两个设备进行配对，，当配对完成后，形成加密连接的密钥，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，当主机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，同一对外设和主机会不时的连接和断开，ble 的安全有一项特性允许两个设备之间建立长久的安全密钥信息，这种特性称为绑定，他允许两设备连接时快速的完成加密认证，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——他是为 GATT 客户端提供数据服务的设备。

△GATT 客户端——他是从 GATT 服务器读写应用数据的设备。

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接中的主机和从机完全独立，主机可以是 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者多个 GATT 服务，GATT 服务是完成特定功能的

一些列数据的集合，在协议栈 demo 工程 SimpleBLEPeripheral 中，有三个 GATT 服务。

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

△强制的 GATT 服务——该服务包含了 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，这部分也是在 ble 协议栈库文件中。

△SimpleBLEProfile 服务——这个服务包含应用数据的信息，与应用程序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。

Characteristic 特征值是服务用到的值，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 服务器上的特征值机器内容和配置信息（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；

△Handle 句柄——属性在表中的地址，每个属性有唯一的句柄；

△type 类型——表示数据代表的事务，通常是蓝牙技术联盟规定的或由用户自定义 UUID

△权限——对顶了 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。

△读特征值——客户端设备请求读取局并处的特征值，服务器将此值回应给客户端。

△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，服务器将所有与制定类型匹配的特性的句柄和值回应给客户端设备。

△读多个特性值——客户端一次请求中读取几个句柄的特征值，服务器将这些特征值回应给客户端。

△读特性描述符——客户端请求读特定句柄处的特征描述符。服务器将特征描述符的值回应给客户设备。

使用 UUID 发现特征值——客户端通过发送特征的类型 UUID 来请求发现这个特征的句柄，服务器将这个特征的声明回应给客户端设备。

△写特征值——客户端设备请求向服务器特定的局并处写入特征描述符，服务器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，客户端设备不需要想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，想要使能服务通知，首先要配置好特征，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的通过设备的 GATT 服务器来注册服务，GATT 服务器将整个服务加到属性表中，并未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，其值是有蓝牙技术联盟定义；

△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“特征声明”紧随其后的是 GATT 特征值；

△GATT_CLIENT_CHAR_CFG_UUID——这一属性代表特征描述符，它与属性表中它前面最近的局并处的特征值有关，他允许 GATT 客户端设备使能特征值通知。

△GATT_CHAR_USER_DESC_UUID——这一属性代表特征值描述符，他与属性表中他前面最近的句柄处的特征值相关，包含一个 ASCCI 字符串，是对相关的特征的描述。

5.5.3 使用 BLE 协议栈提供的 GAP 和 GATT 的 API 函数

应用程序和配置文件直接调用 GAP 和 GATT 的 API 函数来实现 BLE 相关功能，如广播，连接、读写特征值等。

TIble 协议栈中有许多 api 的调用参考，关于 api 函数的具体用法可以参考 ti 提供的 html 格式的文档，在协议栈的 documents 目录下。

5.6 BLE 协议栈中串口打印

我们刚才的实验均是通过 LCD 输出设备信息，例如连接状态，设备地址，数据通信等，但是，在更多的情况下，尤其是开发阶段的调试，串口打印几乎是最重要的，甚至是唯一的调试手段。算然仿真器可以单步调试并且设置断点，但是在 ble 协议栈中，断点意味着 ble 连接将会中断，所以连接状态下的数据或者状态分析通常使用串口打印。

这一节，我们指导大家如何在协议栈中添加串口打印功能。

我们在上一节的 SimpleBLECentral 和 SimpleBLEPeripheral，两个最简单的主机从机例子上修改。

5.6.1 编写 UART 驱动程序

TI BLE 协议栈中已经做了 UART 底层驱动，因此并不需要我们重头编写 UART 的驱动代码，而是直接调用 hal_uart.c 中的 api 函数。该驱动源文件在如下目录：
\BLE-CC254x-1.3\Components\hal\target\CC2540EB\hal_uart.c

那如何调用 hal_uart.c 提供的 api 呢？

1、 编写串口初始化函数，配置 UART 波特率、流控制、缓冲区大小，数据接收回调函数等参数后，打开串口
2、 编写数据接收回调函数。
3、 封装串口打印函数。

5.6.1.1 编写串口初始化函数

在 hal_uart.c 中有一个串口初始化函数：HalUARTInit()，我们这里说的并不是他，HalUARTInit()在芯片上电阶段就会调用。而我们这里说的串口初始化函数，是我们需要在任务函数中调用的初始化串口配置用。

函数代码如下：

```
00024: /*
00025: uart初始化代码，配置串口的波特率、流控制等
00026: */
00027: void serialAppInitTransport( )
00028: {
00029:   halUARTCfg_t uartConfig;
00030:
00031:   // configure UART
00032:   uartConfig.configured           = TRUE;
00033:   uartConfig.baudRate             = SBP_UART_BR;//波特率
00034:   uartConfig.flowControl          = SBP_UART_FC;//流控制
00035:   uartConfig.flowControlThreshold = SBP_UART_FC_THRESHOLD;//流控制阈值，当开启flowControl时
00036:   uartConfig.rx.maxBufSize        = SBP_UART_RX_BUF_SIZE;//uart接收缓冲区大小
00037:   uartConfig.tx.maxBufSize        = SBP_UART_TX_BUF_SIZE;//uart发送缓冲区大小
00038:   uartConfig.idleTimeout          = SBP_UART_IDLE_TIMEOUT;
00039:   uartConfig.intEnable            = SBP_UART_INT_ENABLE;//是否开启中断
00040:   uartConfig.callBackFunc         = sbpSerialAppCallback;//uart接收回调函数，在该函数中读取
00041:
00042:   // start UART
00043:   // Note: Assumes no issue opening UART port.
00044:   (void)HalUARTOpen( SBP_UART_PORT, &uartConfig );
00045:
00046:   return;
00047: } ? end serialAppInitTransport ?
```

第 33 行：设置波特率，我们这里使用的是 57600。

第 34 行：设置 Flow Control，值为 TRUE 或者 FALSE，当为 TRUE 时，除了 TX、RX 外还要连接 CTS 和 RTS。如果设置成 FALSE，只需要 TX 和 RX 就可以外接通信。一般 FALSE 就可以实现一般应用。

第 40 行：设置回调函数。当程序接收到硬件发来的串口数据时，会调用该函数，通知用户做好接收工作。

第 44 行：以上面的配置，打开需要的串口（254x 有两个 uart，HAL_UART_PORT_0 或者 HAL_UART_PORT_1）。

```
00012: /*
00013: 串口设备初始化，
00014: 必须在使用串口打印之前调用该函数进行uart初始化
00015: */
00016: void SerialApp_Init( uint8 taskID )
00017: {
00018:   //调用uart初始化代码
00019:   serialAppInitTransport();
00020:   //记录任务函数的taskID，备用
00021:   //sendMsgTo_TaskID = taskID;
00022: }
```

5.6.1.2 编写数据接收回调函数

当程序接收到硬件发来的串口数据时，会调用刚才配置的回调函数，通知我们做好接收工作。

```
00049: /*
00050: uart接收回调函数
00051: 当我们通过pc向开发板发送数据时，会调用该函数来接收
00052: */
00053: void sbpSerialAppCallback(uint8 port, uint8 event)
00054: {
00055:   uint8  pktBuffer[SBP_UART_RX_BUF_SIZE];
00056:   // unused input parameter; PC-Lint error 715.
00057:   (void)event;
00058:   int i=0;
00059:   for(i=6000;i>0;i--){
00060:     asm("nop");
00061:   }
00062:   //HalLcdWriteString("Data form my UART:", HAL_LCD_LINE_4 );
00063:   //返回可读的字节
00064:   if ( (numBytes = Hal_UART_RxBufLen(port)) > 0 ){
00065:     //读取全部有效的数据，这里可以一个一个读取，以解析特定的命令
00066:     (void)HalUARTRead (port, pktBuffer, numBytes);
00067:   }
00068:
00069: }
```

第 55 行，开辟临时数据缓冲区，用来接收数据。

第 64 行，调用 Hal_UART_RxBufLen 函数，返回当前可读的数据长度。

第 66 行，调用 HalUARTRead 读取 uart 缓存里的数据到 pktBuffer 中。我们这里仅仅是接收，并没有做其他工作，大家完全可以在这里添加串口数据的解析程序，这样可以通过串口终端发送 AT 命令来控制协议栈。

5.6.1.3 封装串口打印函数。

细心的读者应该会发现，在 hal_uart.c 中提供了 HalUARTWrite()函数，用来向 uart 硬件发送数据。函数原型如下：

```
00200: /*************************************************************
00201:  * @fn      HalUARTWrite
00202:  *
00203:  * @brief   Write a buffer to the UART.
00204:  *
00205:  * @param   port - UART port
00206:  *          buf  - pointer to the buffer that will be writt
00207:  *          len  - length of
00208:  *
00209:  * @return  length of the buffer that was sent
00210:  *************************************************************
00211: uint16 HalUARTWrite(uint8 port, uint8 *buf, uint16 len)
```

虽然可以直接调用该函数，但是不够直接，也不够方便，比如我们想这样 printString(“string”)就能发送字符串，或者想向串口打印数值 printValue(10);这就要求我们在其基础上做进一步封装。

SerialPrintString()封装了端口号和数据长度两个参数，直接向用户提供单一的字符串参数。方便使用。

```
00078: /*
00079: 打印一个字符串
00080: str不可以包含0x00，除非结尾
00081: */
00082: void SerialPrintString(uint8 str[])
00083: {
00084:   HalUARTWrite (SBP_UART_PORT, str, osal_strlen((char*)str));
00085: }
```

SerialPrintValue 函数向用户提供了打印数值功能，并且可以控制打印的数值显示格式，如 10 进制或者 16 进制等。

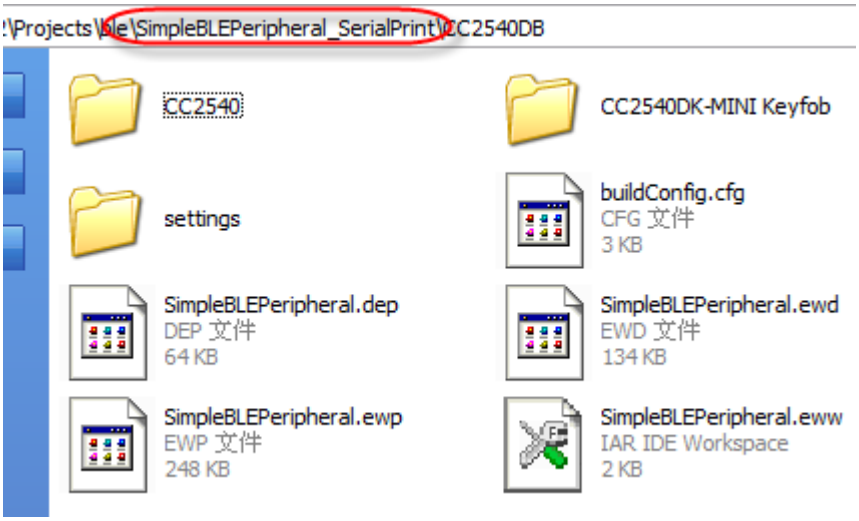
```
00086: /*
00087: 打印指定的格式的数值
00088: 参数
00089: title,前缀字符串
00090: value,需要显示的数值
00091: format,需要显示的进制，十进制为10,十六进制为16
00092: */
00093: void SerialPrintValue(char *title, uint16 value, uint8 format)
00094: {
00095:   uint8 tmpLen;
00096:   uint8 buf[256];
00097:   uint32 err;
00098:
00099:   tmpLen = (uint8)osal_strlen( (char*)title );
00100:   osal_memcpy( buf, title, tmpLen );
00101:   buf[tmpLen] = ' ';
00102:   err = (uint32)(value);
00103:   _ltoa( err, &buf[tmpLen+1], format );
00104:   SerialPrintString(buf);
00105: }
```

5.6.2 新建协议栈工程

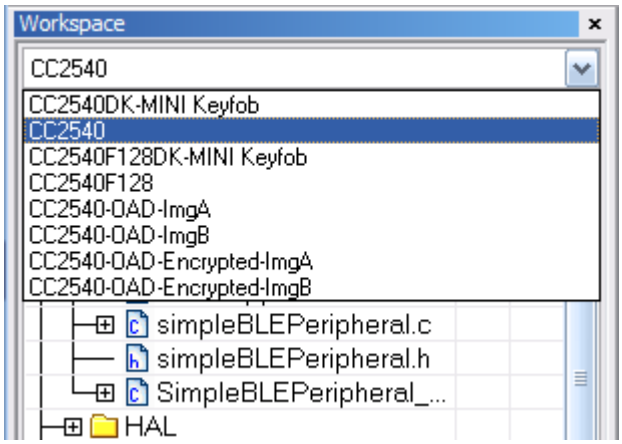
Ble 协议栈中，在 Projects/ble 目录下提供了非常多的协议栈工程，当我们需要建立自己的工程时，可以在现有的基础上修改，加快开发的步伐。

我们当前使用 SimpleBLECentral 和 SimpleBLEPeripheral 为原型，新建带有串口打印功能的 SimpleBLECentral_SerialPrint 和 SimpleBLEPeripheral_SerialPrint

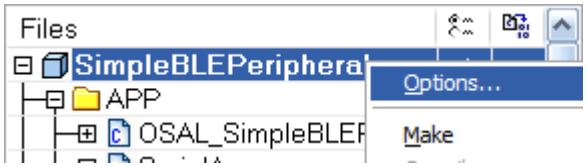
1、复制 SimpleBLEPeripheral 文件夹为 SimpleBLEPeripheral_SerialApp，然后打开 SimpleBLEPeripheral_SerialPrint 的 IAR 工程



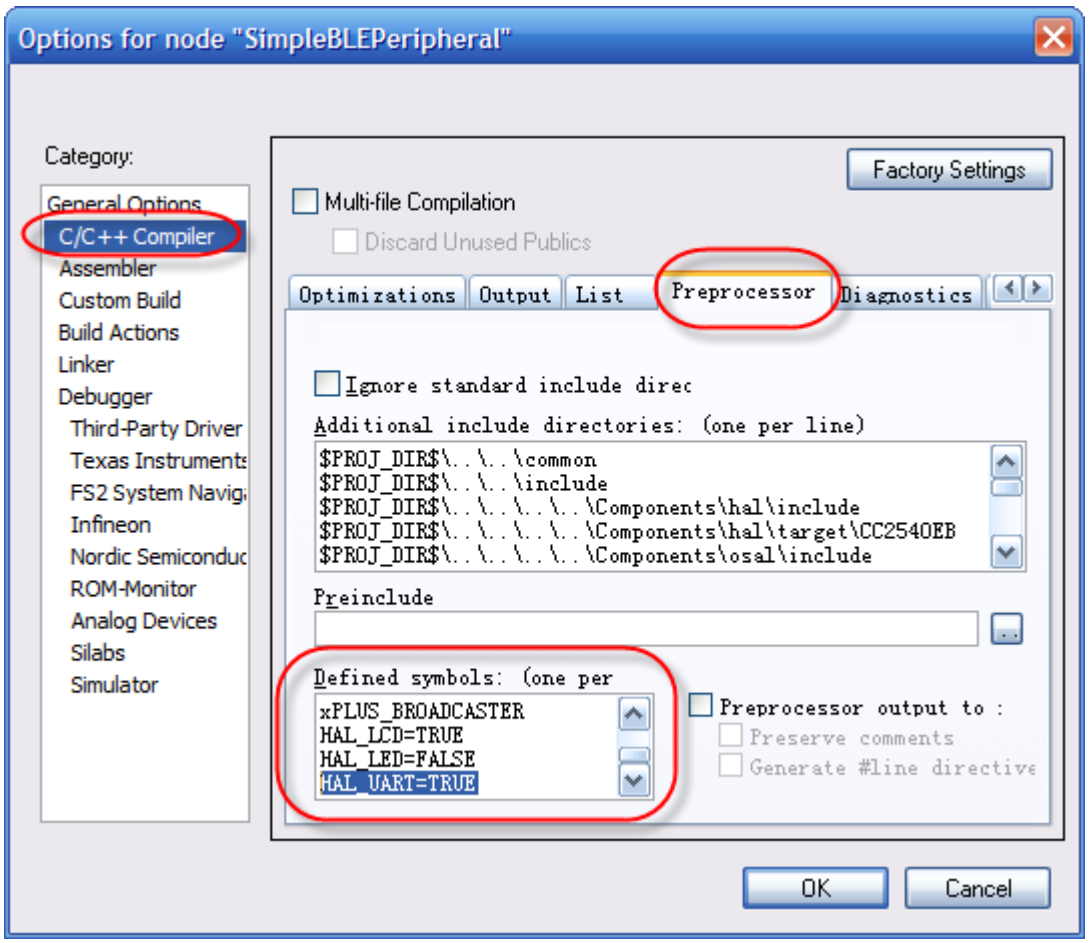
2、然后在 IAR 的 Workspace 中选择合适的 Configuration，当前我们全部给予 SmartRF 平台，因此这里需要选择 CC2540 或者 CC2541。在其他的 IAR 工程中类似。



3、打开该配置的 Option 属性，右击 SimpleBLEPeripheral，在出现的对话框中选择左边的 C/C++Compiler 选项。

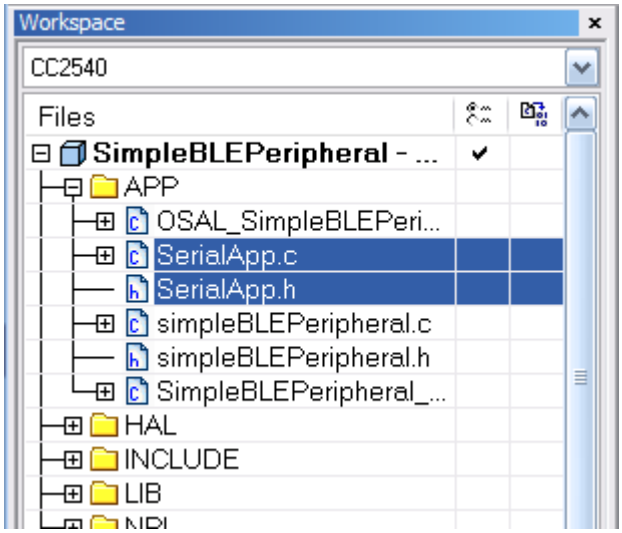


然后再单击右边的 Preprocessor 选项卡，并在 Defined symbols 中添加 HAL_UART=TRUE，如下图：



虽然协议栈代码中是包含 UART 驱动源码的，但是由宏定义 HAL_UART 来决定是否开启 UART，这里提供一个预处理的宏定义，这样 IAR 工程就会包含 UART，在配合自己编写的 UART 初始化代码就可以在协议栈中使用 UART 功能了。

4、添加 UART 初始化代码，在 SimpleBLEPeripheral 的 APP 目录下创建两个文件：SerialApp.c 和 SerialApp.h，我们将 UART 的相关配置放到这两个文件里。



SerialApp.h 截图如下：

```
00001: #ifndef _SERIAL_APP_H_
00002: #define _SERIAL_APP_H_
00003:
00004: #ifdef __cplusplus
00005: extern "C"
00006: {
00007: #endif
00008:
00009:
00010: #define SBP_UART_PORT                   HAL_UART_PORT_0
00011: //#define SBP_UART_FC                     TRUE
00012: #define SBP_UART_FC                     FALSE
00013: #define SBP_UART_FC_THRESHOLD           48
00014: #define SBP_UART_RX_BUF_SIZE            128
00015: #define SBP_UART_TX_BUF_SIZE            128
00016: #define SBP_UART_IDLE_TIMEOUT           6
00017: #define SBP_UART_INT_ENABLE             TRUE
00018: #define SBP_UART_BR                     HAL_UART_BR_57600
00019:
00020:
00021: // Serial Port Related
00022: extern void SerialApp_Init(uint8 taskID);
00023: extern void sbpSerialAppCallback(uint8 port, uint8 event);
00024: void serialAppInitTransport();
00025: void sbpSerialAppWrite(uint8 *pBuffer, uint16 length);
00026: void SerialPrintString(uint8 str[]);
00027: void SerialPrintValue(char *title, uint16 value, uint8 format);
00028:
00029: #ifdef __cplusplus
00030: }
00031: #endif
00032:
00033: #endif
```

SerialApp.c 部分截图如下：

```
00016: void SerialApp_Init( uint8 taskID )
00017: {
00018:   //调用uart初始化代码
00019:   serialAppInitTransport();
00020:   //记录任务函数的taskID，备用
00021:   //sendMsgTo_TaskID = taskID;
00022: }
00023:
00024: /*
00025: uart初始化代码，配置串口的波特率、流控制等
00026: */
00027: void serialAppInitTransport( )
00028: {
00029:   halUARTCfg_t uartConfig;
00030:
00031:   // configure UART
00032:   uartConfig.configured           = TRUE;
00033:   uartConfig.baudRate             = SBP_UART_BR;//波特率
00034:   uartConfig.flowControl          = SBP_UART_FC;//流控制
00035:   uartConfig.flowControlThreshold = SBP_UART_FC_THRESHOLD;//流控制阈值，当开启flowContr
00036:   uartConfig.rx.maxBufSize        = SBP_UART_RX_BUF_SIZE;//uart接收缓冲区大小
00037:   uartConfig.tx.maxBufSize        = SBP_UART_TX_BUF_SIZE;//uart发送缓冲区大小
00038:   uartConfig.idleTimeout          = SBP_UART_IDLE_TIMEOUT;
00039:   uartConfig.intEnable            = SBP_UART_INT_ENABLE;//是否开启中断
00040:   uartConfig.callBackFunc         = sbpSerialAppCallback;//uart接收回调函数，在该函数中
00041:
00042:   // start UART
00043:   // Note: Assumes no issue opening UART port.
00044:   (void)HalUARTOpen( SBP_UART_PORT, &uartConfig );
00045:
00046:   return;
00047: } ? end serialAppInitTransport ?
00048:   uint16 numBytes;
00049: /*
00050: uart接收回调函数
00051: 当我们通过pc向开发板发送数据时，会调用该函数来接收
00052: */
00053: void sbpSerialAppCallback(uint8 port, uint8 event)
00054: {
00055:   uint8  pktBuffer[SBP_UART_RX_BUF_SIZE];
00056:   // unused input parameter; PC-Lint error 715.
00057:   (void)event;
00058:   HalLcdWriteString("Data form my UART:", HAL_LCD_LINE_4 );
00059:   //返回可读的字节
00060:   if ( (numBytes = Hal_UART_RxBufLen(port)) > 0 ){
00061:     //读取全部有效的数据，这里可以一个一个读取，以解析特定的命令
00062:     (void)HalUARTRead (port, pktBuffer, numBytes);
00063:     HalLcdWriteString(pktBuffer, HAL_LCD_LINE_5 );
```

5、将代码集成到 BLE 协议栈任务函数中，在任务函数中调用刚才提到的初始化接口函数即可；

```
void SimpleBLEPeripheral_Init( uint8 task_id )
{
  simpleBLEPeripheral_TaskID = task_id;
  //serial port initialization
  SerialApp_Init(task_id);

  // Setup the GAP Peripheral Role Profile
  {
```

6、添加 ble 通信串口打印，我们把原先通过 lcd 显示的内容，均通过串口输出。这里需要的内容较多。不一一举例。

例如：

```
00474:   if ( events & SBP_START_DEVICE_EVT )
00475:   {
00476:     // Start the Device
00477:     VOID GAPRole_StartDevice( &simpleBLEPeripheral_PeripheralCBs );
00478:
00479:     // Start Bond Manager
00480:     VOID GAPBondMgr_Register( &simpleBLEPeripheral_BondMgrCBs );
00481:
00482:     SerialPrintString("BLE Stack is running\r\n");
```

例如：

```
00648:     case GAPROLE_ADVERTISING:
00649:       {
00650:         #if (defined HAL_LCD) && (HAL_LCD == TRUE)
00651:           HalLcdWriteString( "Advertising",  HAL_LCD_LINE_3 );
00652:           SerialPrintString("Advertising\r\n");
00653:         #endif // (defined HAL_LCD) && (HAL_LCD == TRUE)
00654:       }
00655:       break;
00656:
00657:     case GAPROLE_CONNECTED:
00658:       {
00659:         #if (defined HAL_LCD) && (HAL_LCD == TRUE)
00660:           HalLcdWriteString( "Connected",  HAL_LCD_LINE_3 );
00661:           SerialPrintString("Connected\r\n");
00662:         #endif // (defined HAL_LCD) && (HAL_LCD == TRUE)
00663:       }
00664:       break;
00665:
00666:     case GAPROLE_WAITING:
00667:       {
00668:         #if (defined HAL_LCD) && (HAL_LCD == TRUE)
00669:           HalLcdWriteString( "Disconnected",  HAL_LCD_LINE_3 );
00670:           SerialPrintString("Disconnected\r\n");
00671:         #endif // (defined HAL_LCD) && (HAL_LCD == TRUE)
00672:       }
00673:       break;
```

等等。

7、SimpleBLECentral_SerialPrint 和 SimpleBLEPeripheral_SerialPrint 需要修改的内容非常相似。大家可以自行修改，也可以直接使用我们提供的修改后的工程（工程所在位置：CC254xEK\实验与实战\串口打印 SerialPrint）。

5.6.3 编译下载及测试

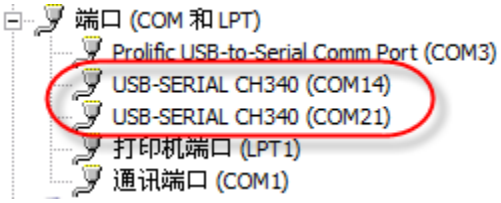
我们重新编译 SimpleBLECentral_SerialPrint 和 SimpleBLEPeripheral_SerialPrint。然后分别下载到两块 SmartRF 开发板中。

注意，连接完仿真器和开发板后，必须先按仿真器的复位按键，等到仿真器识别到开发板（CC-Debugger 灯由红变绿，SmartRF04EB 仿真器由灭变量）后再进行下载或者调试。

程序下载结束后，使用 mini-usb 连接 New SmartRF。打开电源开关。如下图：

如果大家第一次通过 mini-usb 连接 PC，会出现安装 USB 转串口驱动。我们开发板上使用 uart 直转 usb 芯片是 CH340G，市场上非常常见，驱动程序位于【Software\串口\USB 转串口驱动】

驱动安装成功后，会在设备管理器中出现两个虚拟串口。如下图：



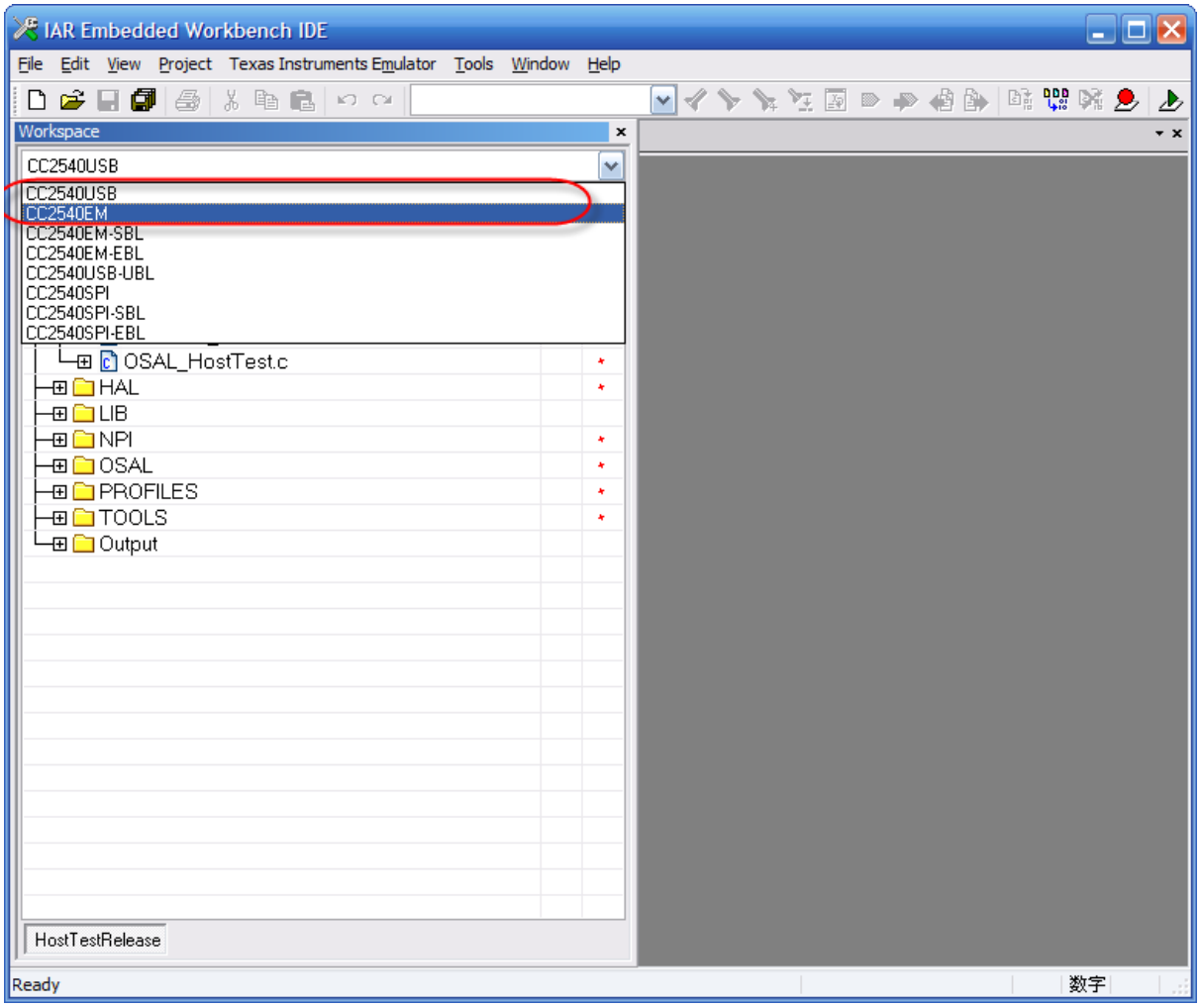
我们这里 COM21 对应 central，14 对应 peripheral，此时，就可以通过串口来监控协议栈的运行状态了。

5.7 BLE 协议栈里的网络处理器 HostTestRelease

TI 为开发者免费提供的协议栈中包含了一个 HostTestRelease 网络处理器 demo，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 CC254x 上执行，而应用程序和配置文件在另一个设备上执行，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 SmartRF 开发板的串口或者 CC2540USBdongle 模拟出来的 CDC 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 MCU 或者 PC 通过串口来发送控制命令，当设备受到任何消息或者需要采取某种操作，它会将消息发送给外部 MCU 或者 PC。

5.7.1 工程概述

HostTestRelease 网络处理器工程结构和一般的协议栈 demo 一样，同样适用硬件抽象层，操作系统虚拟层，虽然他也包含了一个称之为 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，这些转换的所有源码都包含在 hci_ext_app.c 文件中，HostTestRelease 网络处理器工程如下图所示：



我们打开的是CC2540DB，在workspace下拉列表中注意红圈，CC2540EM或者CC2541EM，使用的是New SmartRF系列的开发板，通过254x内部的UART转串口连接PC串口，注意，除了RX和TX外，还要连接CTS和RTS。

CC2540USB，则使用的是CC2540USBdongle，直接插到USB接口上，通过CDC驱动模拟一个虚拟串口。

关于Btool的使用，可以参见BTOOL使用手册。

第 6 章 BLE 协议栈实践与实战