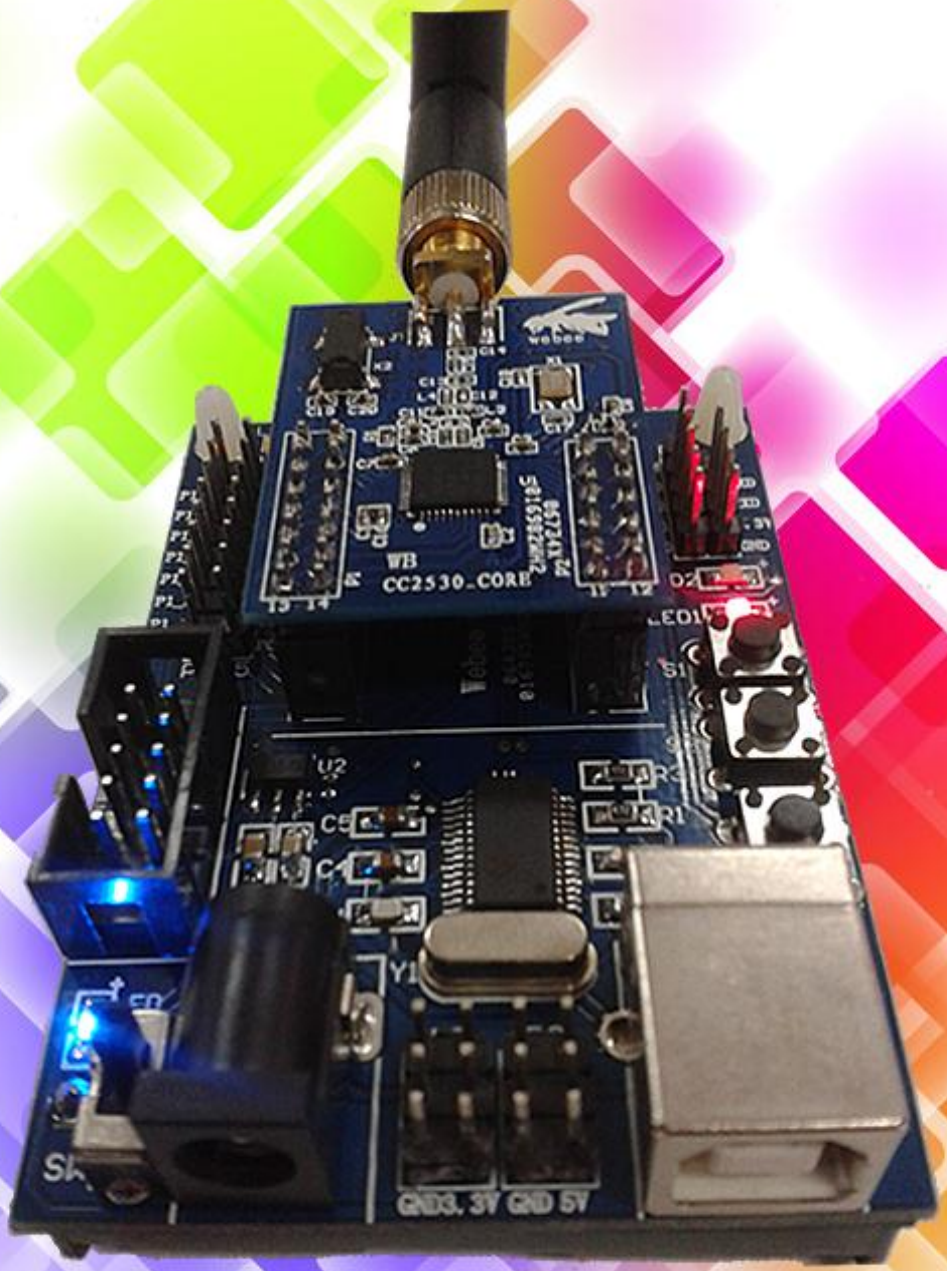
Zigbee实战演练

webee

——基础实验
——组网演练
——项目实战

网蜂团队
倾力打造



网蜂家族

引领你进入ZigBee世界

前言

为什么学习 ZigBee？

试想着有一天，你坐在电脑前，你家里的空调、冰箱、洗衣机、电视…等在你屏幕前受你掌控时，这便是物联网的时代。我是一名普通的电子爱好者，半年前由于项目需要我接触了 ZigBee,在那之前我也曾经了解过 ZigBee,感觉是个很神秘的东西，神秘的无线模块。第一天在导师里拿到 ZigBee 模块，我快速寻找着上面的 MCU，很可惜只发现 CC2530 芯片，后来才发现，原来 CC2530 芯片上集成了增强型 8051 单片机内核，我们直接烧写程序进去就可以了。这令我喜出望外，直到现在有人问我 ZigBee 是什么的时候？我会回答：单片机+无线模块。当我利用 TI 例程实验无线点灯那一刻，我感觉这就是我想要的东西。直到今天，大家都在讨论物联网，都在讨论 ZigBee。

为什么要写《ZigBee 实战演练》教程？

ZigBee 功能强大、组网方式千变万化，往往令很多初学者望而却步，连最简单的无线点灯、恐怕都要好长一段时间才弄出来，看网络协议栈的代码是，更是被里面千奇百怪的代码，无数的定义搞得头昏目眩。在这代码海洋遨游，看不见陆地。国内可以参加的较为系统的资料少之又少。在这个海洋泡了数月的我感觉找到了一个通向目的地的罗盘，再回过头来看以前的笔记，发现原来 ZigBee 学习还是有方法可寻，因此我们网蜂团队从自己的学习经历中，尽量以最简单的描写来讲述的 ZigBee 的学习方法，从基础实验、组网演练到项目实战，《ZigBee 实战演练》诞生了，电子书能让我们配上精美的彩图，平易近人的口吻讲解实验。每一个代码都是自己亲身的经历，我们的目的是为了让大家能够学好 ZigBee，战胜 ZigBee。

为什么要打造网蜂 ZigBee 学习套件？

ZigBee 在中国是一个新兴的东西，前途无限，但是网上的学习模块套件参差不齐，大多是复制 TI 公司开源的开发板来设计，用过的就知道，外国的电路设计跟国内的风格很不同，甚至常常让初学者钻牛角尖。为此，网蜂团队特意打造的中国风的 ZigBee 开发套件，《ZigBee 实战演练》上的例程也是基于这学

习板开发的，每个例程都能直接跑起，通过项目实战一章，你甚至可以用它来完成你的项目。我们基于取之于 TI，用之于国民的原则，务求开发最具性价比的 ZigBee 开发套件，软硬件和教程同步更新，力争打造国内一流的物联网开发平台。部分 PCB 模块坚持开源。为大家 ZigBee 学习保驾护航。

技术的学习是有限的，奉献的精神是无限的　--网蜂宗旨

网蜂团队

2012.8 于广州大学城

QQ：1076678176

邮箱：1076678176@qq.com

官方网店：webee.taobao.com

声明

版本说明

《ZigBee 实战演练》由蜂网团队打造，始终坚持开源原则，包括书籍内容、所有代码和部分 ZigBee 学习模块 PCB 的开源。

当前版本为 V3 第三版。采用全新教材文档编写方法，规范内容和排版；协议栈版本更新至 Z-stack 2.5.1a 官方最新版；新增网蜂 zigbee 周边学习模块介绍及多个 TI 官方重要工具的使用教程；修正了蜂迷学习第二版过程中所有发现的 BUG。

更新日期：2014.1.1

历史版本：

2012.11.11

第二版。基础实验新增 LCD12864 液晶显示；组网演练新增协议栈中的按键实验、网络通讯实验（单播、组播、广播）、Zigbee 协议栈网络管理、传感器应用等内容；项目实践新增无线点台灯、无线 IC 卡考勤机、串口通讯助手==Zigbee 聊天助手等内容。

2012.8.8

第一版。内容涵盖了 ZigBee 的简介、开发环境的建立、基础实验、部分组网实验和几个项目实践。

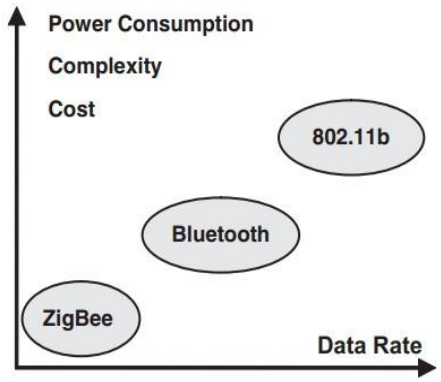
目录

第1章 ZigBee 简介和开发环境快速建立

1.1 ZigBee 简介

Zigbee 是基于 IEEE802.15.4 标准的低功耗个域网协议。根据这个协议规定的技术是一种短距离、低功耗的无线通信技术。这一名称来源于蜜蜂的八字舞，由于蜜蜂(bee)是靠飞翔和“嗡嗡”(zig)地抖动翅膀的“舞蹈”来与同伴传递花粉所在方位信息，也就是说蜜蜂依靠这样的方式构成了群体中的通信网络。其特点是近距离、低复杂度、自组织、低功耗、低数据速率、低成本。主要适合用于自动控制和远程控制领域，可以嵌入各种设备。简而言之，ZigBee 就是一种便宜的，低功耗的近距离无线组网通讯技术。国内通常会翻译成“紫蜂”，我倒不喜欢这个名字，因为它歪曲了原来的含义。

相信大部分人开始时会以为 Zigbee 是一类无线模块，我一开始也是这么的认为，所以当我首次看到 Zigbee 产品时，第一时间找它上面的 MCU，还真想知道用什么单片机来控制这东西，找了半天没发现，一头雾水。最后才发现，原来我们 CC2530 芯片上集成了 8051 内核，你没看错，我也没打错，里面集成了一片增强型的 51 单片机。只要你有 51 单片机的编程基础，就可以轻轻松松的开始玩 Zigbee 了。



	Data Rate	Typical Range	Application Examples
ZigBee	20 to 250 Kbps	10–100 m	Wireless Sensor Networks
Bluetooth	1 to 3 Mbps	2–10 m	Wireless Headset Wireless Mouse
IEEE 802.11b	1 to 11 Mbps	30–100 m	Wireless Internet Connection

图 1.1 Zigbee、蓝牙、WIFI 传输标准对比图

从图 1.1 中几种无线传输的属性中我们可以看到 Zigbee 的应用范围是低速率远距离的。这造就了 Zigbee 低功耗信息传输的优势，网上经常谈到两节普通的 5 号干电池可以使用 6 个月到 2 年的时间，免去充电和更换电池的麻烦。

ZigBee 节点所属类别主要分三种，分别是协调器(Coodinator)、路由器(Router)、终端(End Device)。同一网络中至少需要一个协调器，也只能有 1 个协调器，负责各个节点 16 位地址分配（自动分配）。理论上可以连上 65536 个节点。组网方式千变网化，如图 1.2 所示。

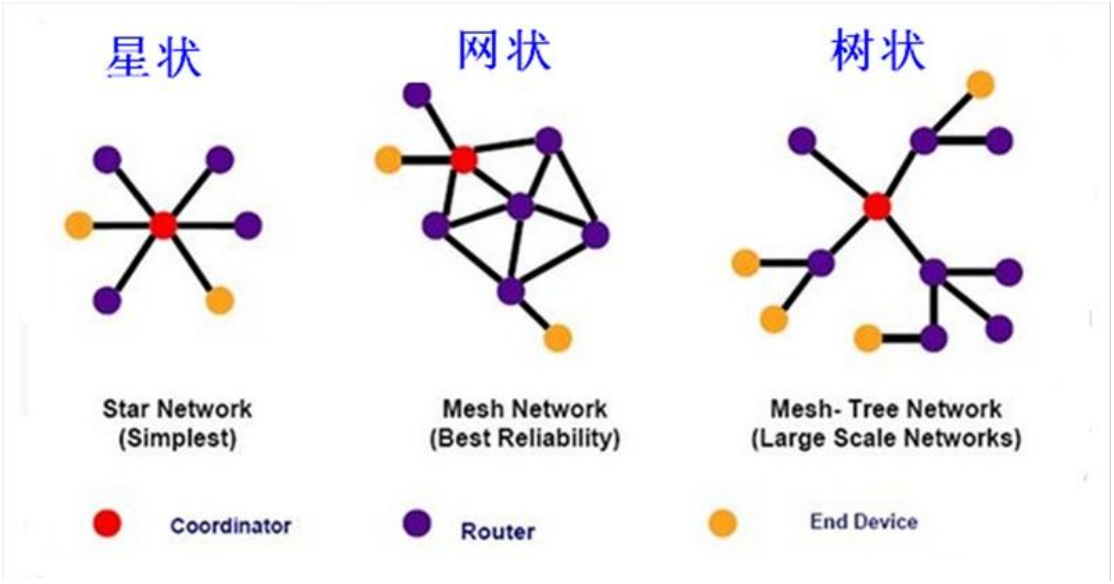


图 1.2　ZigBee 网络

目前 ZigBee 的应用领域主要有：

1. 智能家居物联网 （物联网似乎已经成了趋势，我们拭目以待）
2. 工业、农业无线监测系统
3. 个人监控、医院病人定位
4. 消费电子
5. 城市智能交通
6. 户外作业及地下矿场安全监护

…

…

图 1.3 ZigBee 应用领域

Zigbee 的应用领域很广,这里就不一一列举了。随着技术日益成熟以及价格的下降，Zigbee 在大多领域取代原始的无线模块是毋庸置疑的。举个最简单的例子，终端节点和协调器的最大通讯距离为 200 米，我们在 200 米的地方加入 1 个节点设备作为路由器，那么终端就可以通过路由器转发，也就是说通讯距离可达 400 米。而且新节点加入现有网络极为方便。我们姑且可以先把 ZigBee 当成普通的无线模块应用。

1.2 网蜂 ZigBee 开发平台介绍

为了让大家能够更方便地学习 Zigbee，网蜂团队打造了一套本土化的高性价比学习套件及周边配套模组。我们的学习平台是 IAR + Z-stack 2007 PRO，芯片是 TI 公司的 CC2530。也是目前国内最流行的且资料最全的 Zigbee 学习和应用方案。《ZigBee 实战演练》上的例程也是基于本学习平台开发的，承诺资源会不断更新，保证所有程序能直接跑起。毫不夸张地说，你甚至可以将本教材的例程和实践应用在自己的设计、项目生产上。

1.2.1 网蜂 CC2530 核心板

图 1.4 CC2530 核心板

功能特点：

体积小（3.6*2.7cm），重量轻，引出全部 IO 口，标准 2.54 排针接口。可直接应用在万用板或自制 PCB 上。模块使用 2.4G 全向天线，可靠传输距离达 250 米。自动重连距离高达 110 米。

1.2.2 网蜂 CC2530+CC2591(PA)核心板

引脚和不带 PA 的核心板完全兼容，可靠传输距离 400 米，自动重连距离 360 米。

图 1.5 CC2530+CC2591(PA) 核心板

1.2.3 网蜂 ZigBee 功能底板

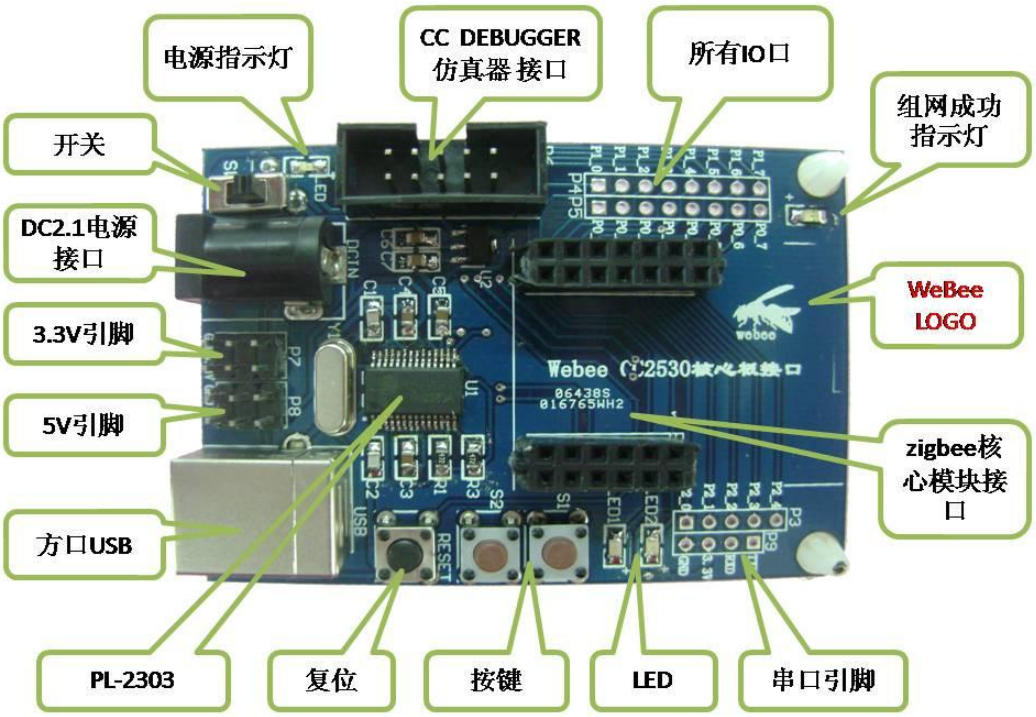


图 1.2C 功能底板

功能特点：

底板尺寸：7 * 5 cm

串口通讯：自带 USB 转串口功能（PL-2303）,方便笔记本用户

供电方式：方口 USB、DC2.1 电源座（5V）。7 号锂电池（3.7V）

功能接口： Debug 接口，兼容 TI 标准仿真工具，引出所有 IO 口，常用的串口引脚以及 5V/3.3V 引脚

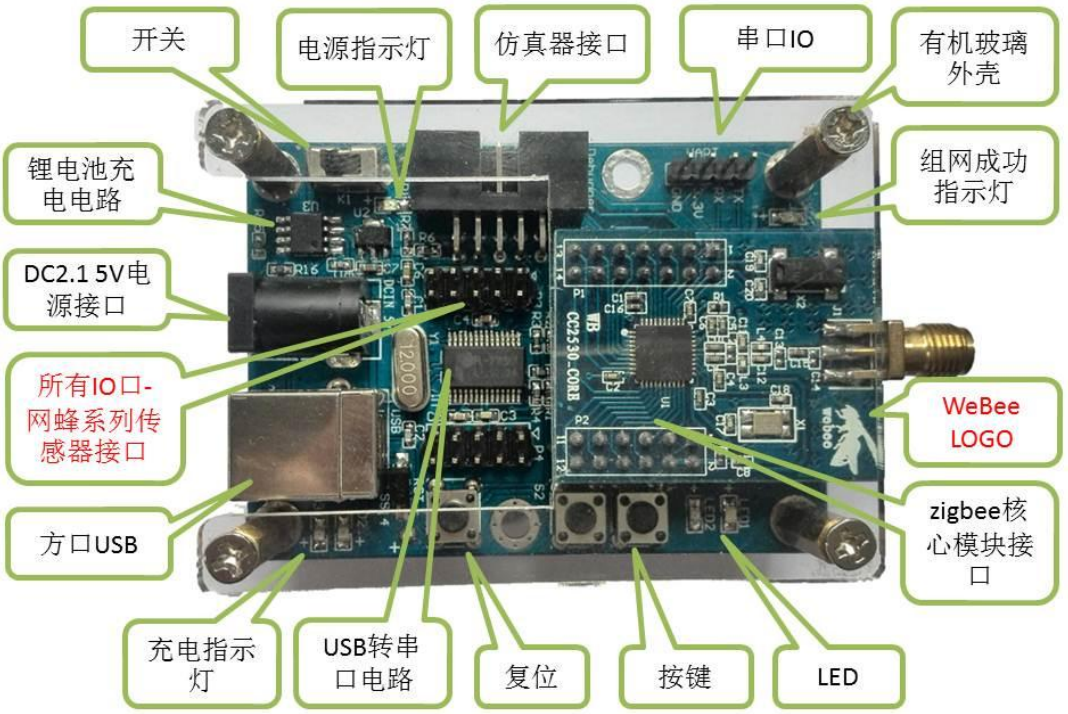
功能按键：1 个复位，2 个普通按键

LED 指示灯：电源指示灯、组网指示灯和普通 LED

模块支持：支持 WeBee CC2530 核心板,CC2530+PA（cc2591）核心板。

网蜂特色：模块可以使用 7 号 3.7V 锂电池通过 LDO 进行稳压供电，此时可以由学习板变身为移动节点，学习使用 2 不误。该设计大大提高了节点的工作时间和节约用户开支。

1.2.4 网蜂 ZigBee 增强型功能底板



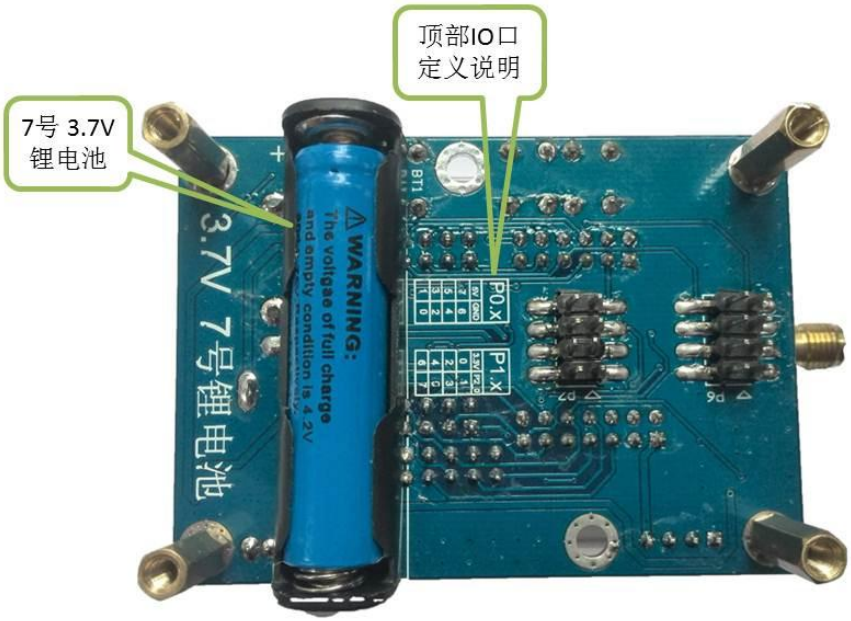


图 1.6 增强型功能底板

功能特点：

底板尺寸：7 * 5 cm

串口通讯：自带 USB 转串口功能（PL-2303）,方便笔记本用户

供电方式：方口 USB、DC2.1 电源座（5V）。7 号锂电池（3.7V）

功能接口： Debug 接口，兼容 TI 标准仿真工具，引出所有 IO 口，常用的串口引脚以及 5V/3.3V 引脚

功能按键：1 个复位，2 个普通按键

LED 指示灯：电源指示灯、组网指示灯和普通 LED

核心模块支持：支持 WeBee CC2530 核心板,CC2530+PA（cc2591）核心板。

传感器模块支持：支持网蜂标准传感器接口：温度 DS18B20/温湿度 DHT11 等全系列传感器

网蜂特色：模块可以使用 7 号 3.7V 锂电池通过 LDO 进行稳压供电，此时可以由学习板变身为移动节点，学习使用 2 不误。该设计大大提高了节点的工作时间和节约用户开支。加入锂电池充电电路，USB 即插即充电。非常方便！

1.2.5 网蜂 ZigBee 传感器开发平台

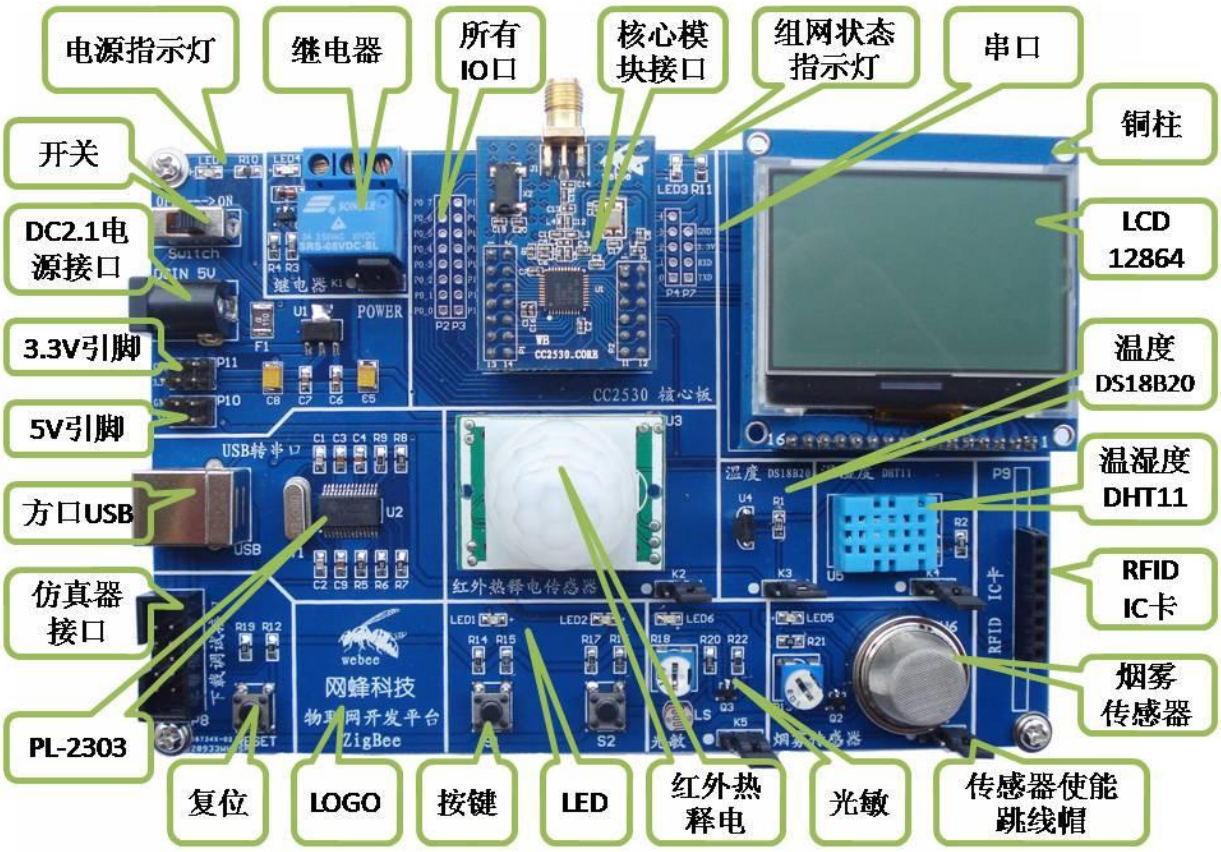


图 1.7 传感器底板

功能特点：

1. 底板尺寸：15 * 10 cm
2. 串口通讯：自带 USB 转串口功能（PL-2303）,方便笔记本用户
3. 供电方式：方口 USB、DC2.1 电源座（5V）。
4. 功能接口 ：Debug 接口，兼容 TI 标准仿真工具，引出所有 IO 口，常用的串口引脚以及 5V/3.3V 引脚
5. 功能按键：1 个复位，2 个普通按键
6. LED 指示灯：电源指示灯、组网指示灯和普通 LED
7. 传感器：温度传感器 DS18B20、温湿度传感器 DHT11、光敏传感器、烟雾传感器、红外热释电传感器

8. 其他：继电器、RFID IC 卡、LCD12864

网蜂特色：网蜂每部分传感器都自带跳线帽作为选通使能端，我们只需要选需要的传感器便可使用， 这样方便用户编程时候 IO 口复用以及扩展自己的传器等模块。

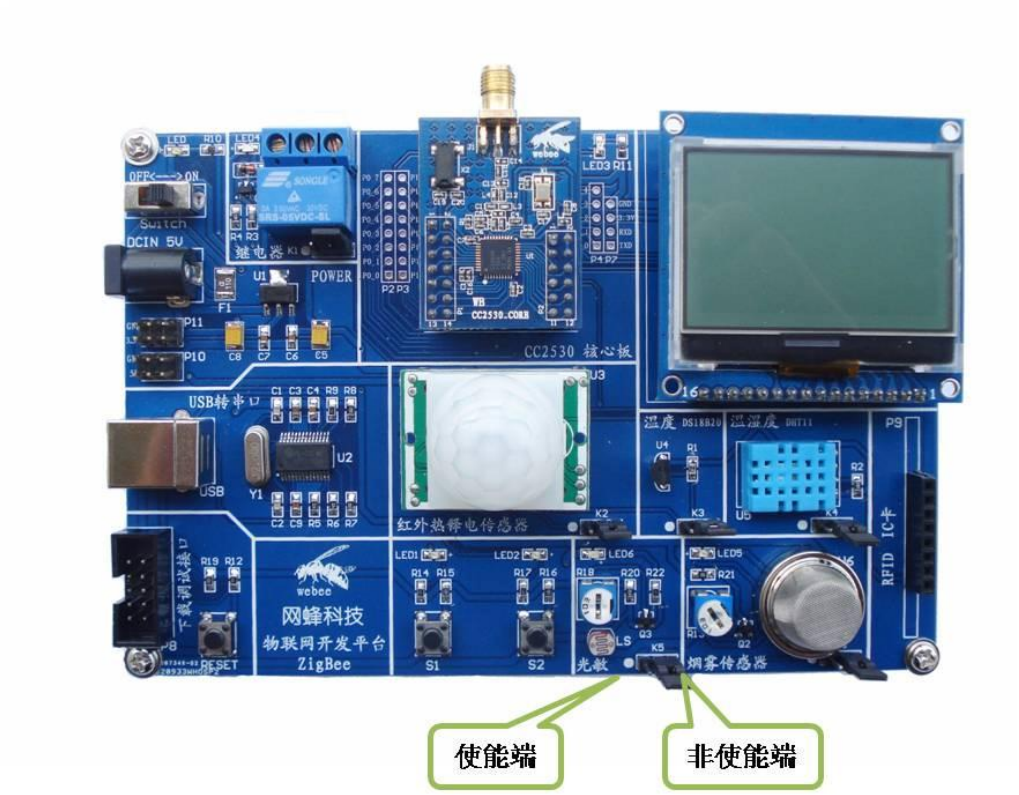


图 1.8 白色点为使能端

*在对应传感器编程时，需要将对应的跳线帽打到使能端，其他不使用的打到非使能端。

1.2.6 网蜂 ZigBee 普通底板（PCB 开源）

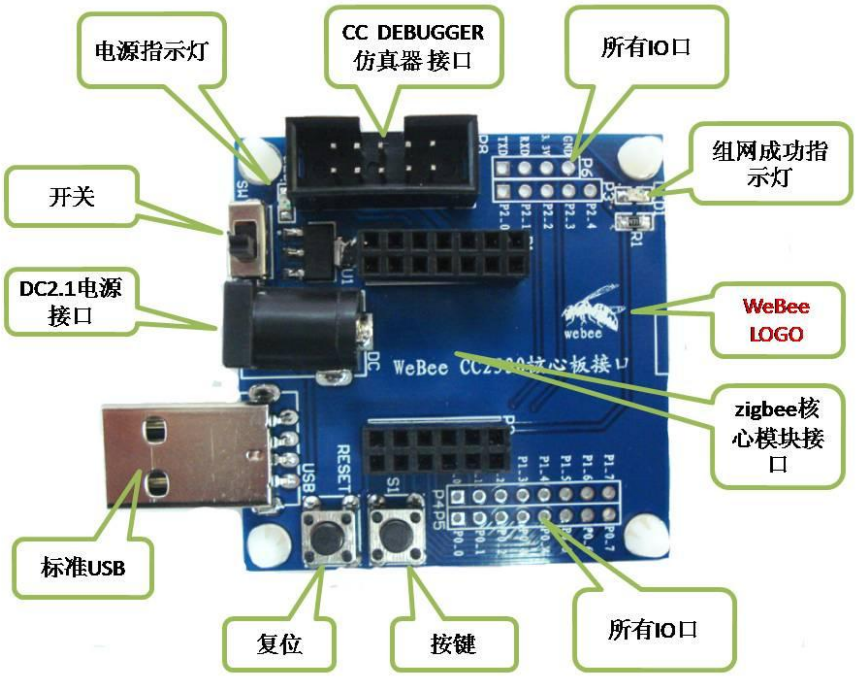


图 1.9 普通底板

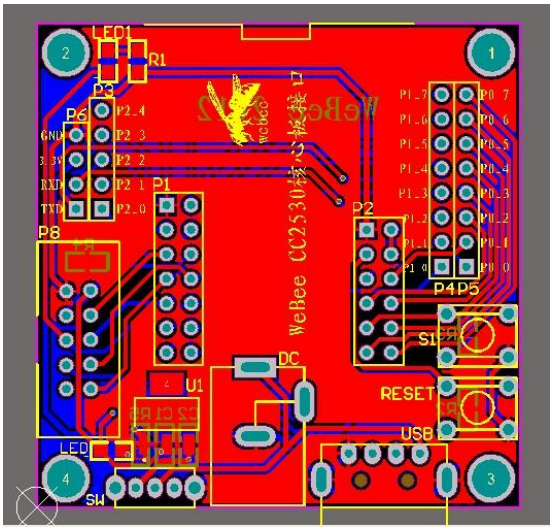


图 1.10 PCB 开源

功能特点：

底板尺寸：5 * 5 cm

供电方式：标准 USB、DC2.1 电源座（5V）。

功能接口： Debug 接口，兼容 TI 标准仿真工具，引出所有 IO 口，常用的串口引脚以及 5V/3.3V 引脚

功能按键：1 个复位，1 个普通按键

LED 指示灯：电源指示灯、组网指示灯

模块支持：支持 WeBee CC2530 核心板,CC2530+PA（cc2591）核心板。

1.2.7 网蜂 ZigBee USB Dongle

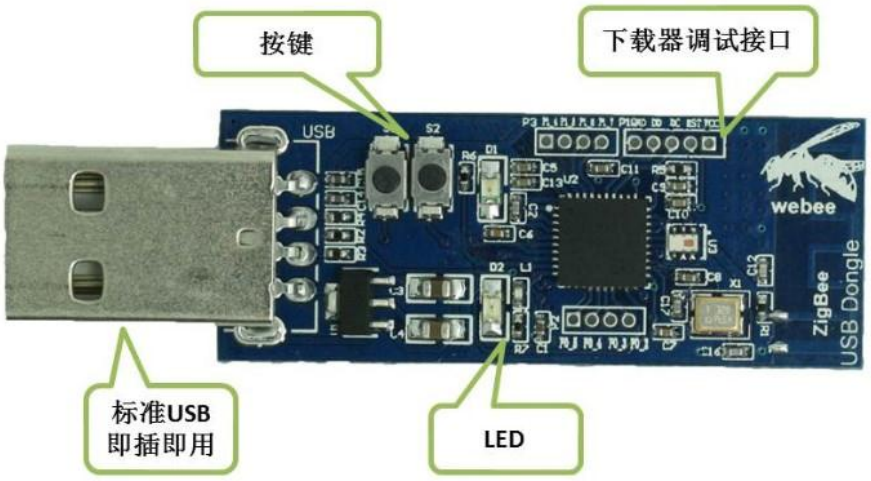


图 1.11 zigbee USB ongle

功能特点：

1.尺寸： 4.5 * 1.7 cm

2.体积小，重量轻，引出下载口和常用 IO 口，标准 2.0mm 排针接口

3.开放频段，工作频段为 2.4GHz

4.16 个传输信道，根据环境进行切换可靠通信信道。

5.无线传输速率达 250Kbps

6.功耗：接收电流<20mA,发射电流<25mA

7.使用 PCB 天线，扩展接收距离。

8.配合 SmartRF Packet Sniffer 可以实现 zigbee 数据抓包功能。

9.配合官方应用实现无线键盘鼠标实验。

网蜂特色：兼容所有官方应用。全网唯一超窄边框设计。

1.2.8 网蜂 ZigBee 仿真器 SmartRF04EB

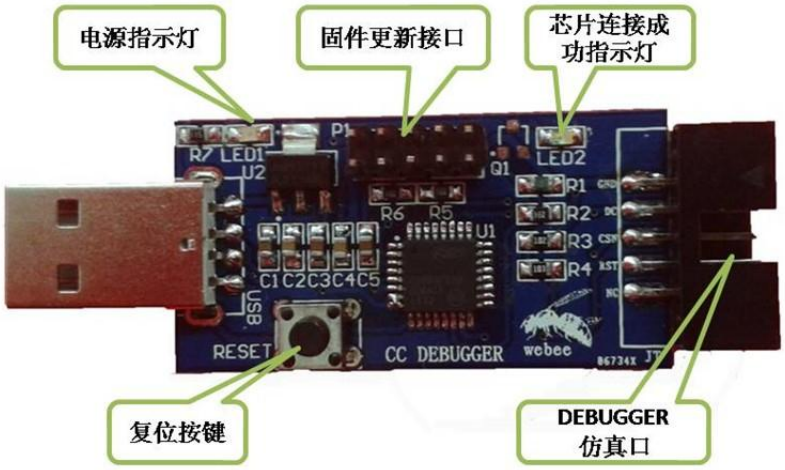


图 1.12 SmartRF04EB

功能特点：

1. 小尺寸 **4.7*2.3 cm** ，标准 **USB** 接口，直接使用。
2. 支持仿真器直接供电；
3. 支持 IAR 在线调试、程序下载、SmartRF STUDIO 和 packet sniffer 协议分析功能；
4. 支持 **TI zigbee** 系列芯片，如：**CC111x/CC243x/CC253x/CC251x**

网蜂特色：预留USB_bootloader更新接口，允许用户自行更新USB_bootloader。

1.2.9 网蜂 TI CC 系列芯片仿真器 CC DEBUGGER

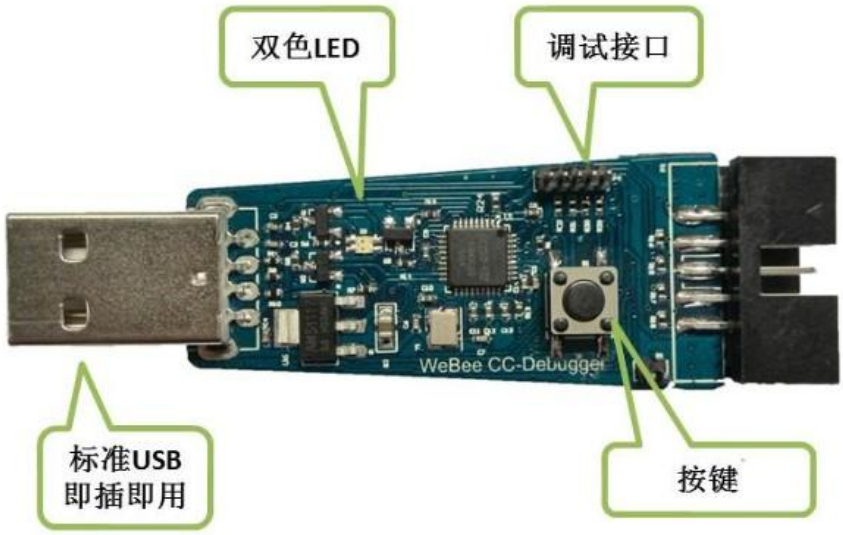


图 1.13 CC DEBUGGER

功能特点：

1. 小尺寸 4.7*2.3 cm ，标准 USB 接口，即插即用。
2. 支持 IAR 在线调试、程序下载、SmartRF STUDIO 和 packet sniffer 协议分析功能；
3. 支持 USB 更新固件；
4. 兼容 TI 全 CC 系列芯片：

CC1110，CC1111

CC2430，CC2431

CC2510，CC2511

CC2530，CC2531，CC2533

CC2540，CC2541

CC2543，CC2544，CC2545

CC1120，CC1121，CC1125，CC1175

CC1100，CC1101，CC110L，CC113L，CC115L

CC2500，CC2520

CC8520，CC8521

CC8530，CC8531

网蜂特色：支持直接 3.3V 供电模式以及 1.8V /3.3V 官方多电压输入模式调试切换。开发板无需外接电源

1.2.10 网蜂 ZigBee 开发套件

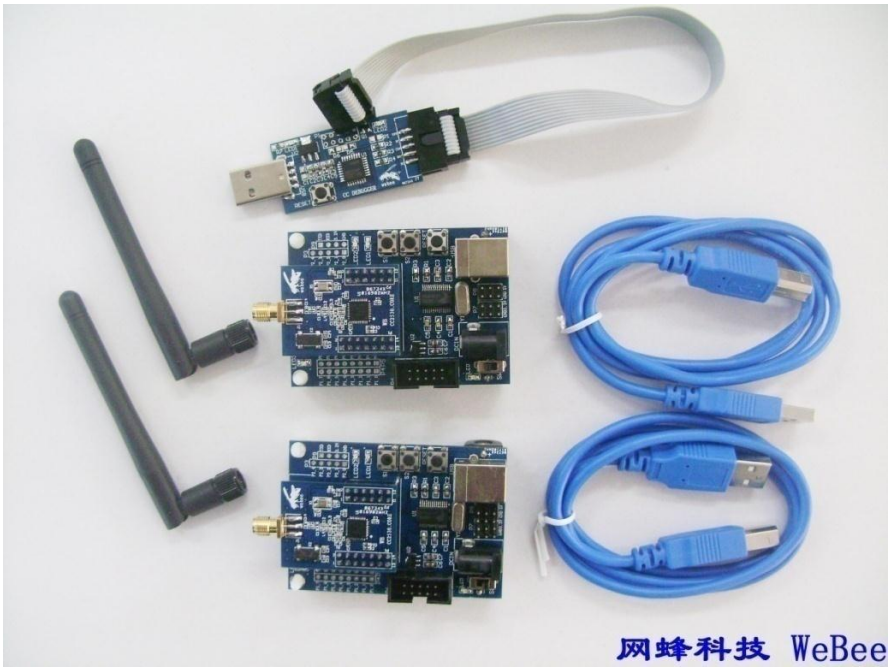


图 1.14 ZigBee 开发套件

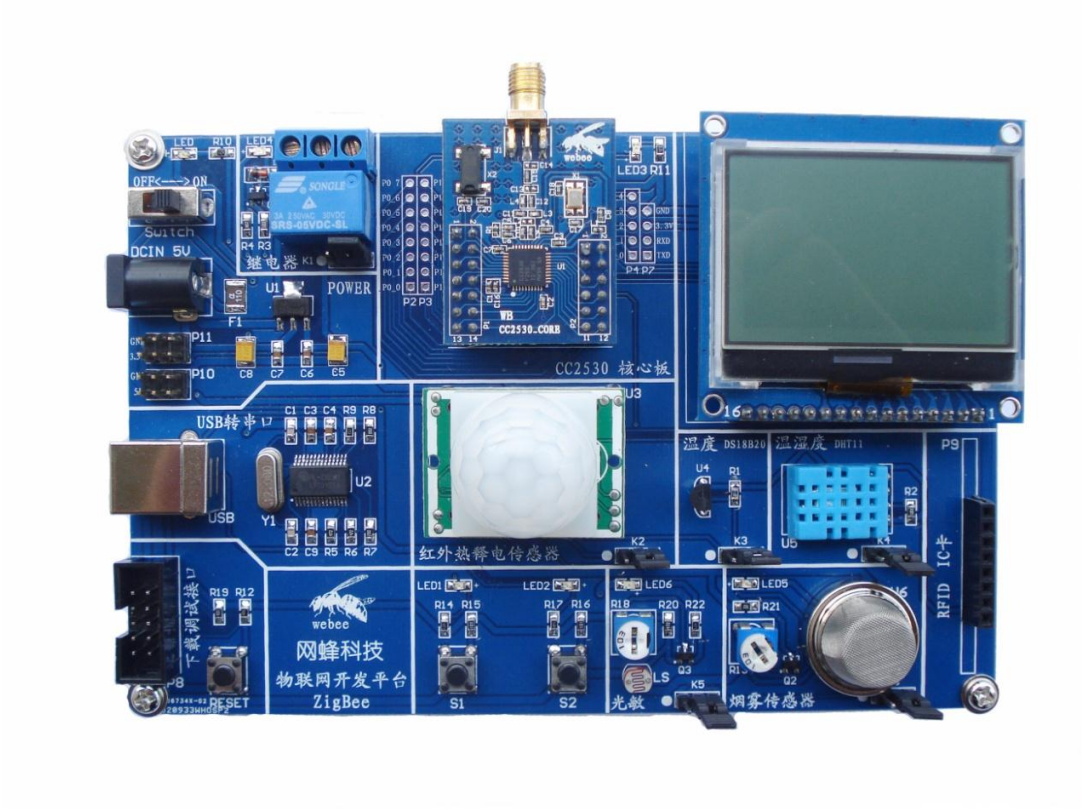


图 1.15 ZigBee 开发平台

1.3 开发环境快速建立

记得以前我们学习 51 单片机的时候相信用得最多的是 KEIL 了，类似，这里我们使用 IAR 8.10，IAR 开发最大优势就是能够直接使用 TI 公司提供的协议栈 Z-Stack 进行开发，我们只需要调用 API 接口函数。这里我们选用 ZStack-CC2530-2.5.1a （Zigbee 2007），《zigbee 实战演练》前 2 个版本用 Zstack-CC2530-2.3.0-1.4.0 版本的，目前全面更新至 2.5.1a，功能更强大！通用性较高。初学者要注意了，IAR 和 Z-Stack 的高低版本是互不兼容的，所以我们两个东西的版本安装选取一定要配合好。经过我这个白老鼠测试，IAR 8.10 和 Zstack-CC2530-2.5.1a 配合使用时从安装到开发都很友好。

本节内容分两部分：1、相关软件和驱动安装 2、IAR 项目工程文件的快速建立。

1.3.1 相关软件和驱动安装

第一步：安装 IAR 8.10 方法：

打开安装文件，选择 IAR 安装，官方推荐默认安装在系统盘：

图 1.16 打开 IAR 8.10 安装文件

提示要求输入 License，由 IAR8.10 注册机生成（参考图 1.18），选项正确后生成 License,复制到 License#处：

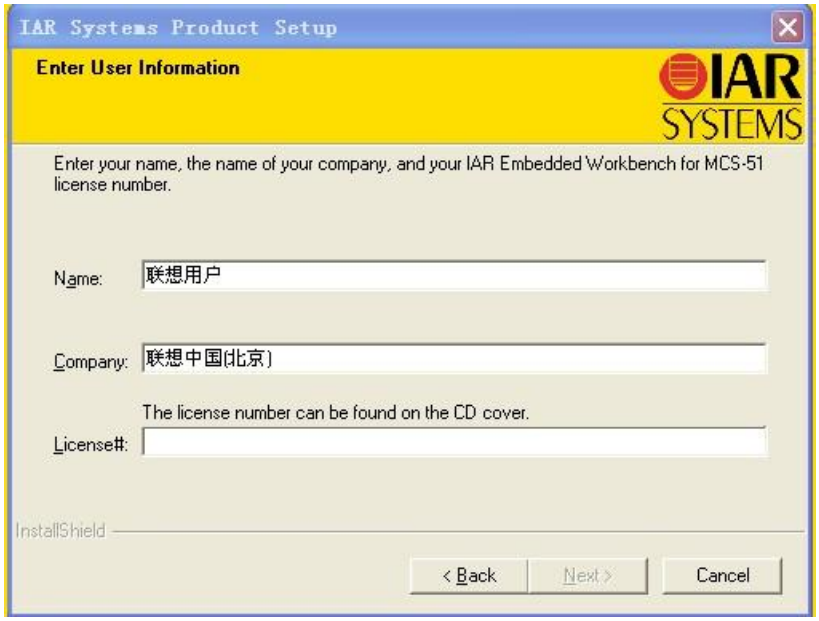


图 1.17

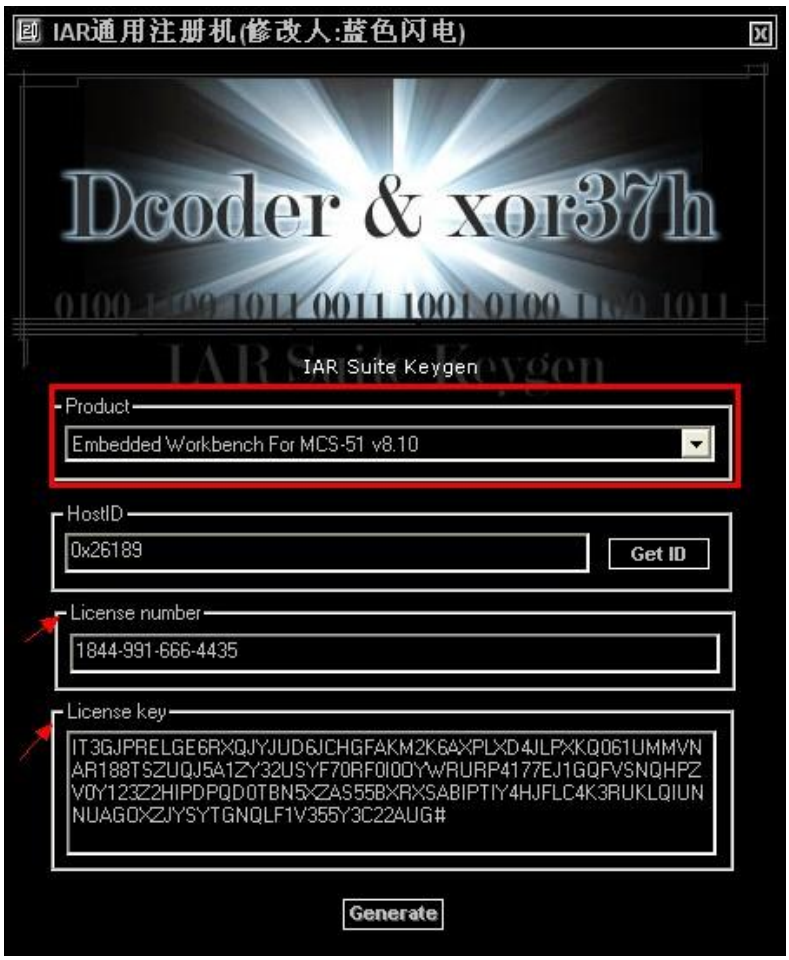


图 1.18 IAR 注册机

输入注册码后按提示一步步进行安装，直至完成程序安装。程序安装完成后默认路径为:（注意当前 C 盘为系统盘）

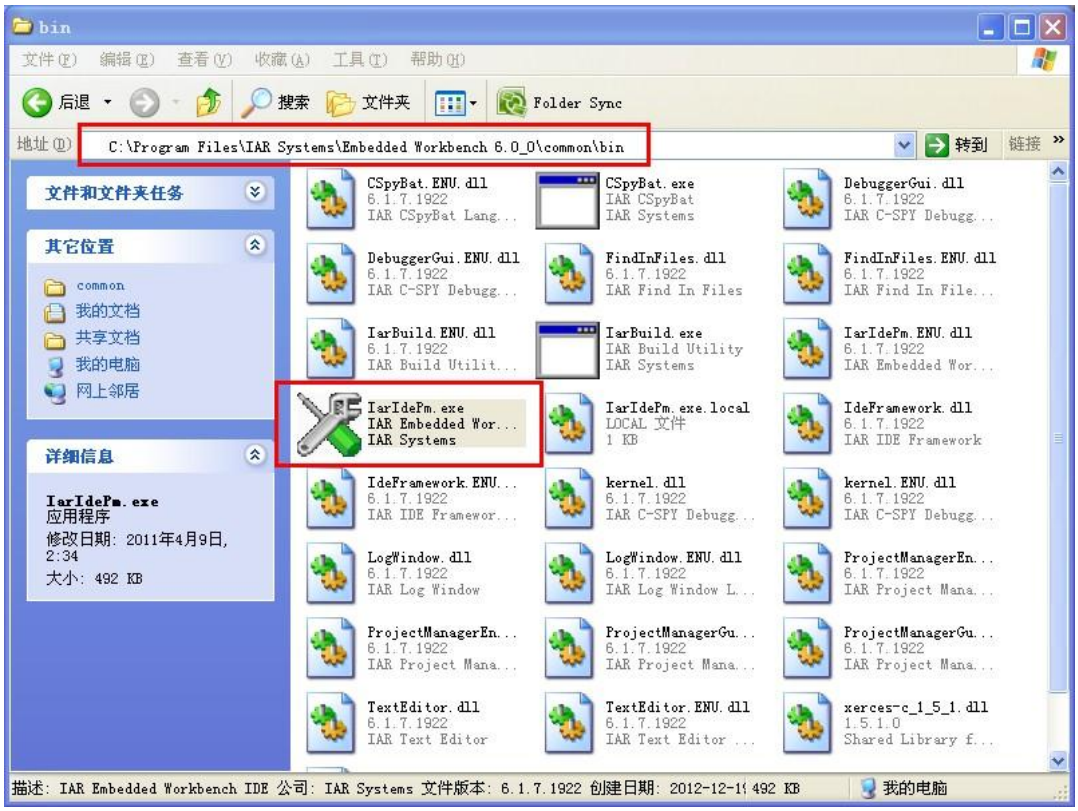


图 1.19 IAR 默认安装路径

安装完成软件界面如下：

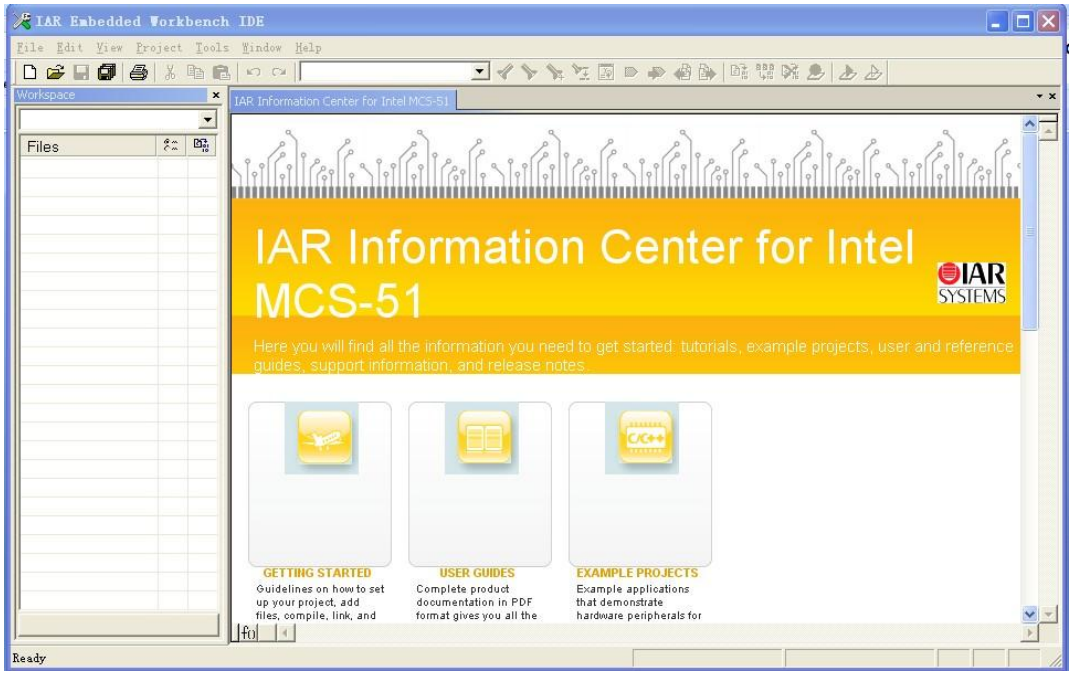


图 1.20 IAR 8.10 软件界面：

第二步：TI 协议栈 Zstack-CC2530-2.5.1a 安装方法：

Z-stack 的安装比较简单，同样安装在默认路径。

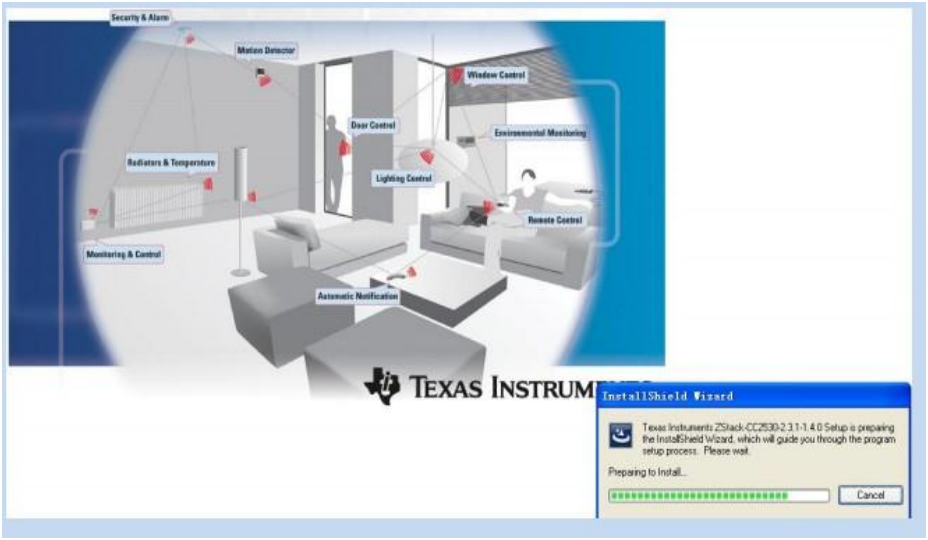


图 1.21 z-stack 安装过程

协议栈安装完成后在图这个路径（C 盘为系统盘），里面包含了例程和工具。我们将在后面讲解：

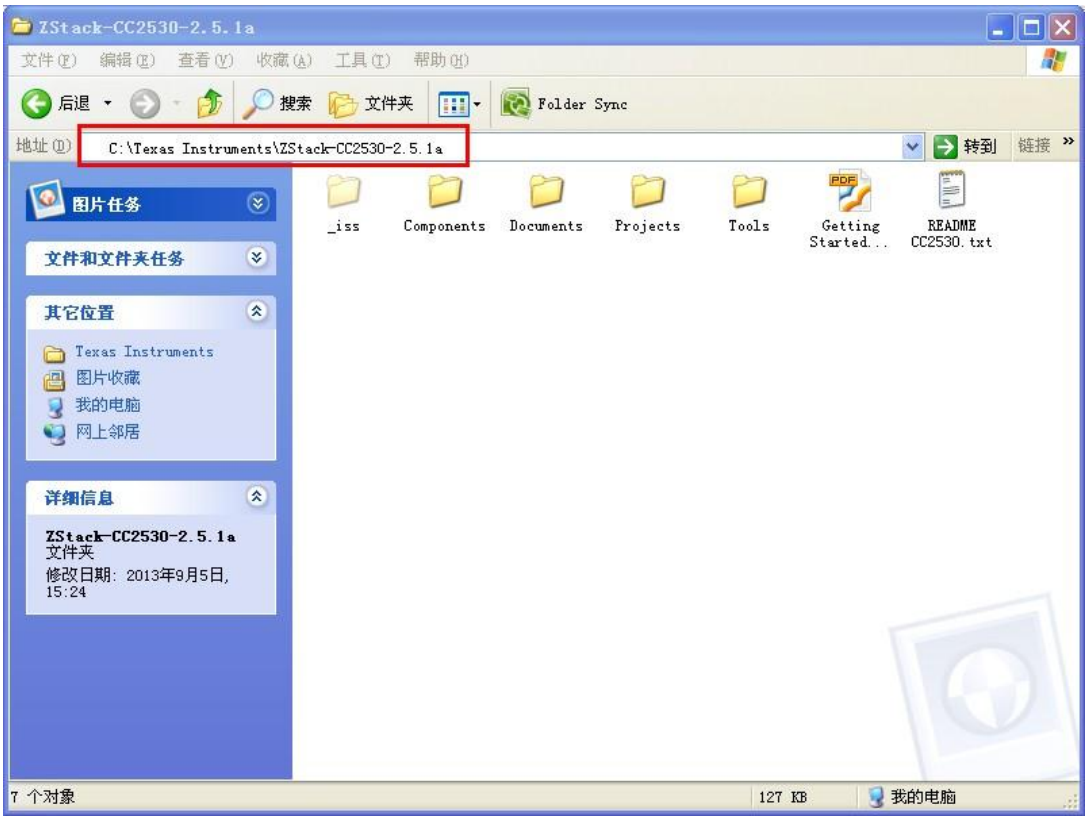


图 1.22 Z-stack 默认安装路径

Z-stack 我们还没需要用得这么快，在接下来的教程里，我们先把它当做一款 51 单片机来学习，学习其资源和内部寄存器。也就是基础实验，基础好的就当做复习一下单片机吧。

第三步： 仿真器 SRF04EB 驱动安装方法

我们将网蜂的仿真器 SRF04EB 进电脑，提示找到新硬件，选择列表安装。

图 1.23 提示安装新硬件

驱动的路径如图 1.24 所示：

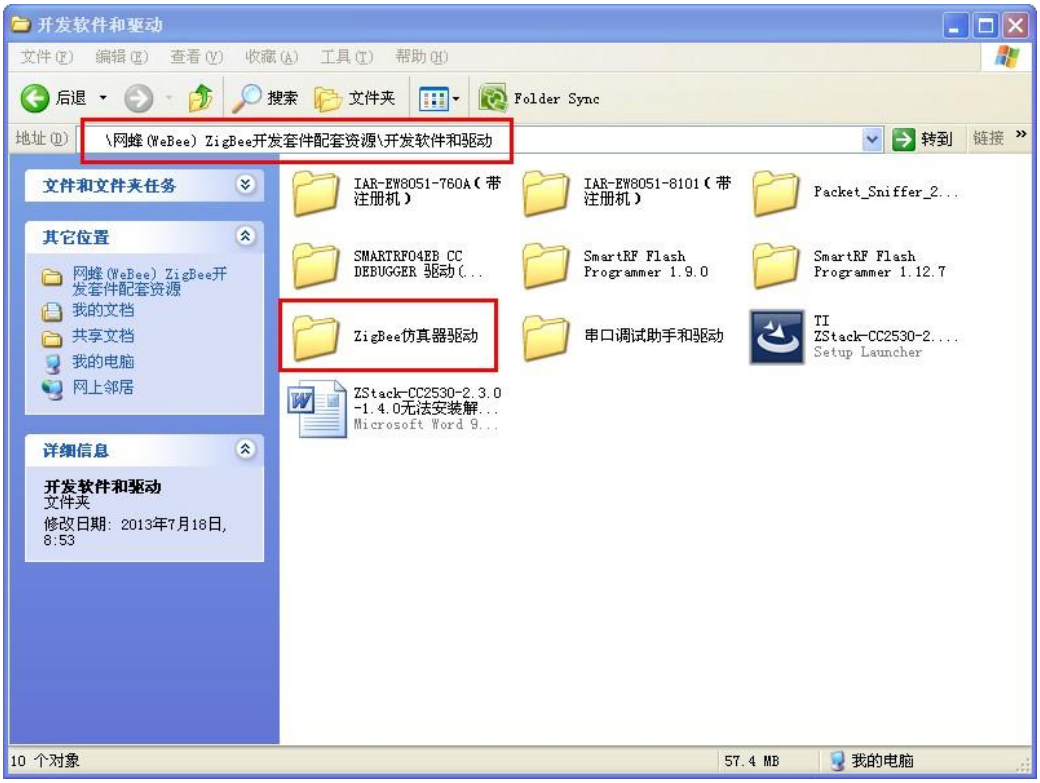


图 1.24 zigbee 仿真器驱动选择路径

安装完成后，重新拔插仿真器，在设备管理器里找到 Chipcon SRF04EB，说明驱动安装完成，如所示。

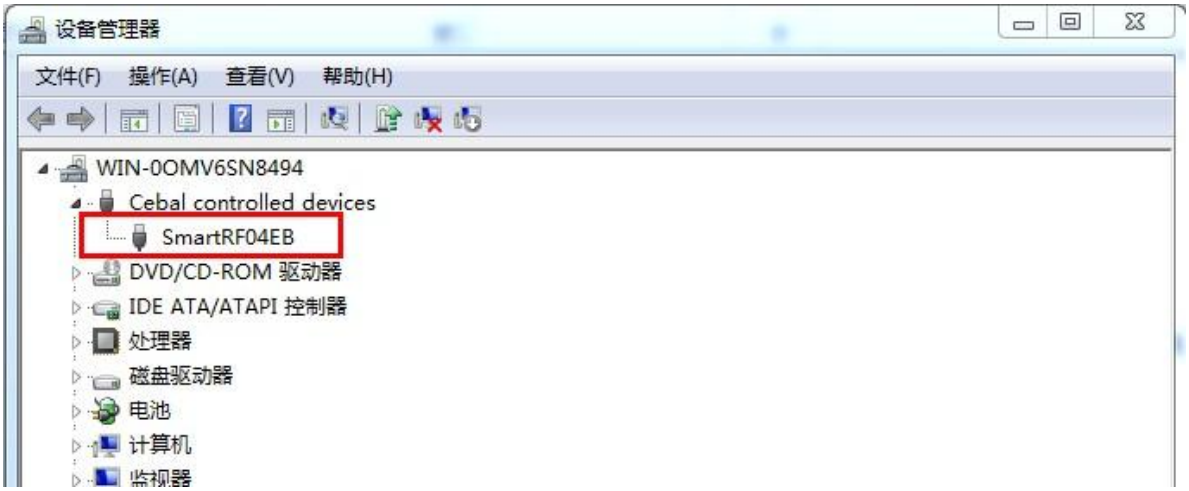


图 1.25 驱动安装完成

连接 CC2530 开发板，按下 DEBUGGER 复位键，芯片指示灯亮（表示检测到开发板上 CC2530 芯片），则完成连接工作。

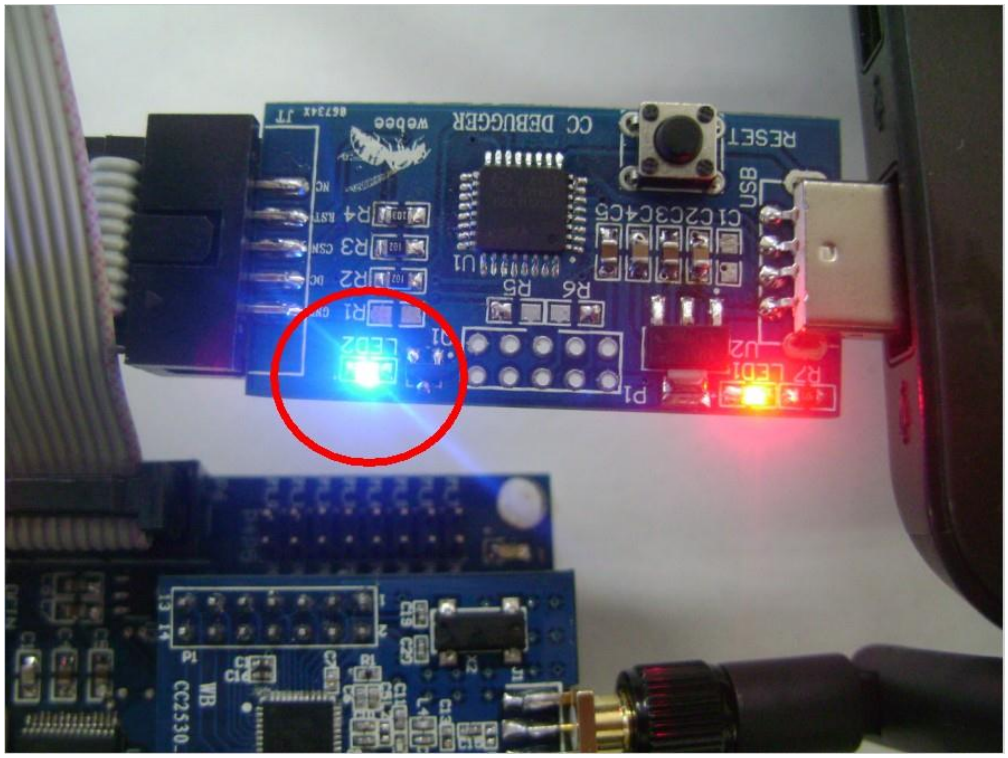


图 1.26 仿真器识别芯片指示灯亮

至此，相关开发软件和仿真器驱动都安装好了，接下来我们讲一下在 IAR 8.10 编译环境中如何快速建立自己的工程和修改相关配置。

第四步：USB 转串口驱动的安装

ZigBee 所有开发板上集成 PL2303 的 USB 转串口芯片，我们通过安装相应的驱动可通过 USB 直接开发调试。打开 PL2303_driver 软件直接进行安装。（安装时候建议 USB 线不连接 zigbee 开发板！）

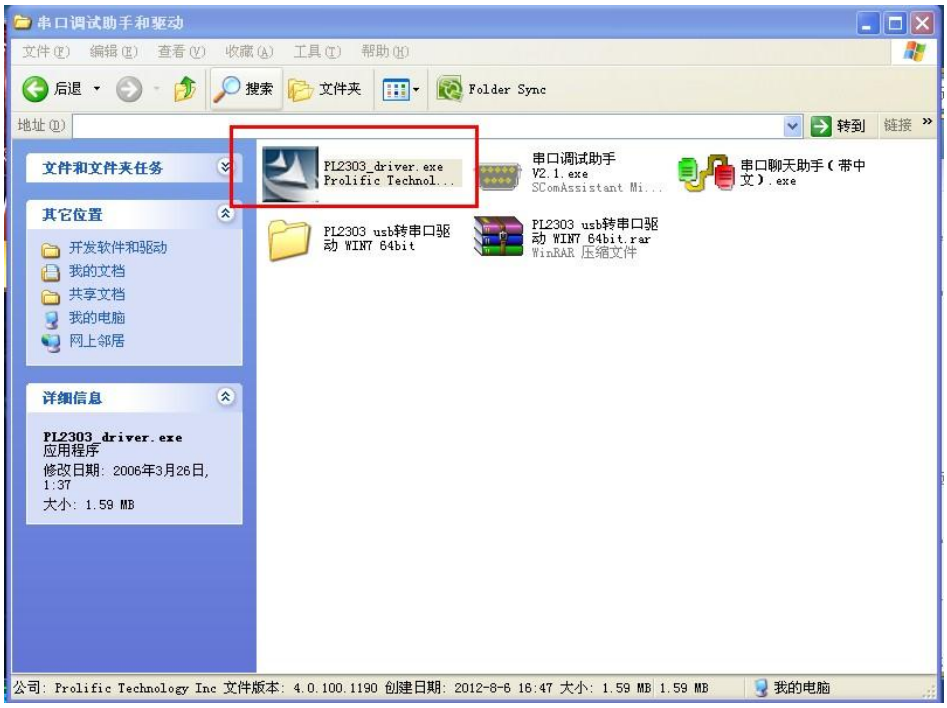


图 1.27 找到 PL2303 驱动

安装好后，通过方口 USB 线连接 zigbee 开发板，我们右键打开我的电脑--属性—硬件—设备管理器，查看到 USB-to-Serial Com，说明驱动安装成功。

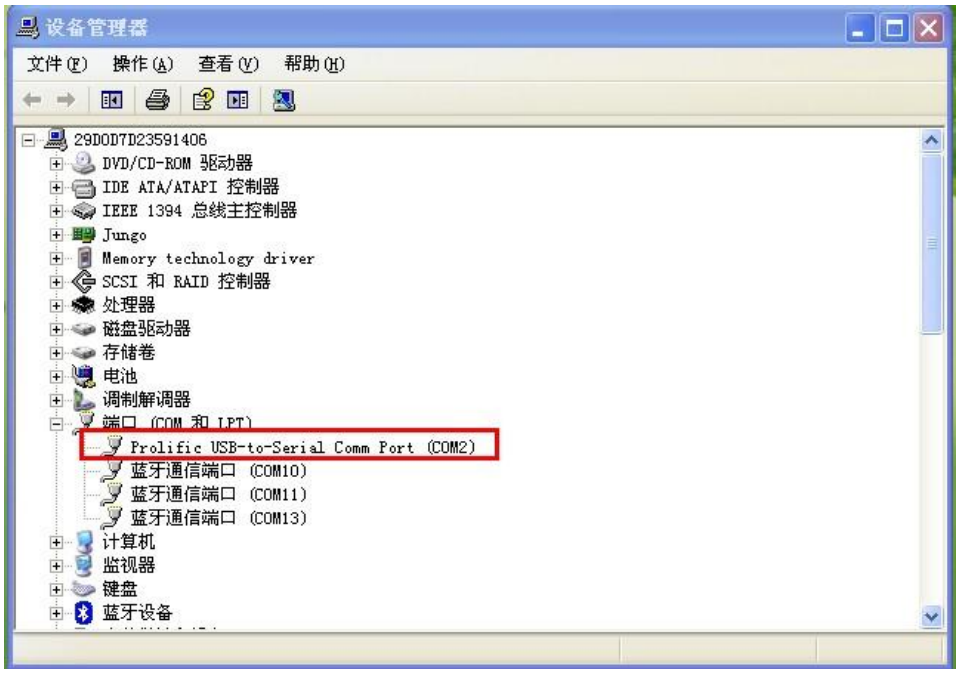


图 1.28 成功安装驱动

1.3.2 IAR 工程文件的快速建立

第一步：打开我们上次已经安装好的 IAR 软件，新建一个 Project-Create New Project，选择默认选项可以了，点击 OK。保存在自己希望的路径。

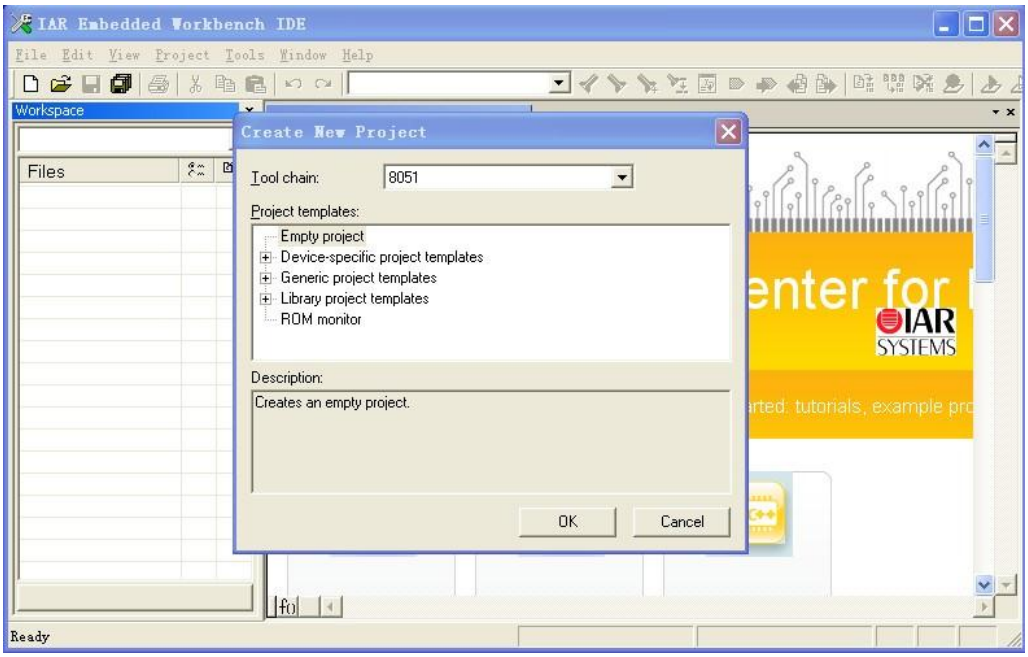


图 1.29

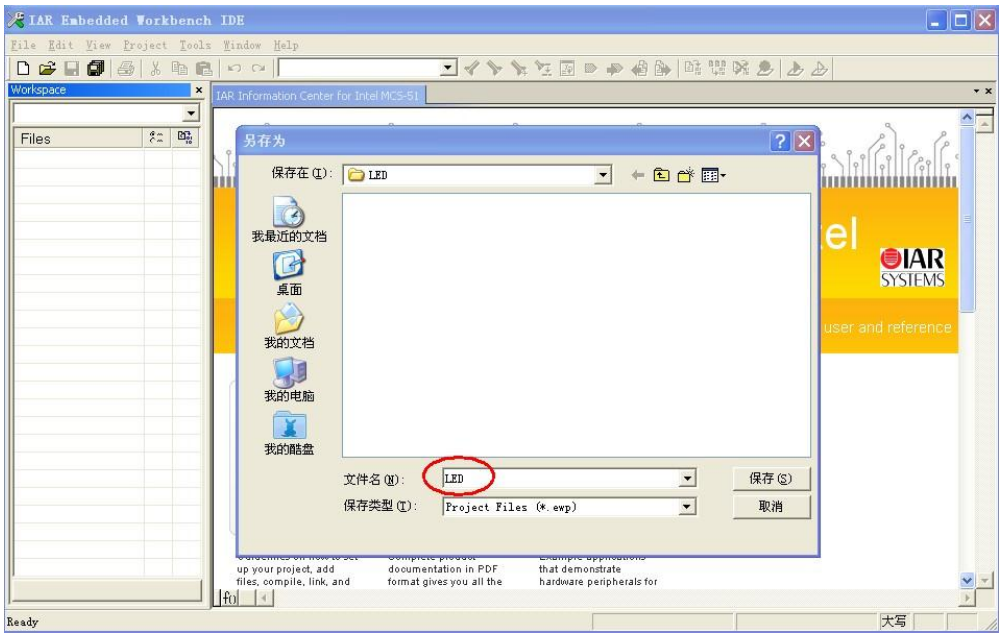


图 1.30

第二步：新建文件，输入#include<ioCC2530.h>,我们基础实验需要用到的也只有这个头文件。然后保存为.c 格式到工程文件路径下。怎么样，是不是跟 KEIL 开发 51 很相似呢？

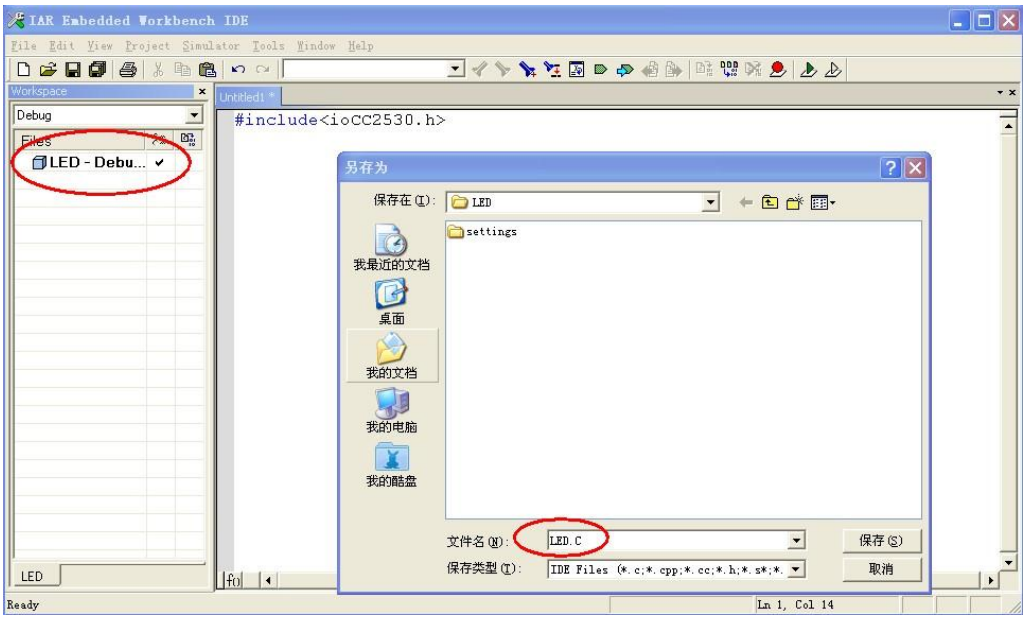


图 1.31 保存为.c 格式文件

第三步：弄好后就可以继续敲代码了，这是基础实验里点亮第一个点亮 LED 代码大家看懂没问题（具体参考基础实验）。打完后保存，记得要在左边工程里单击右键---add---刚保存的 C 文件，成功添加后如图 1.32 所示。

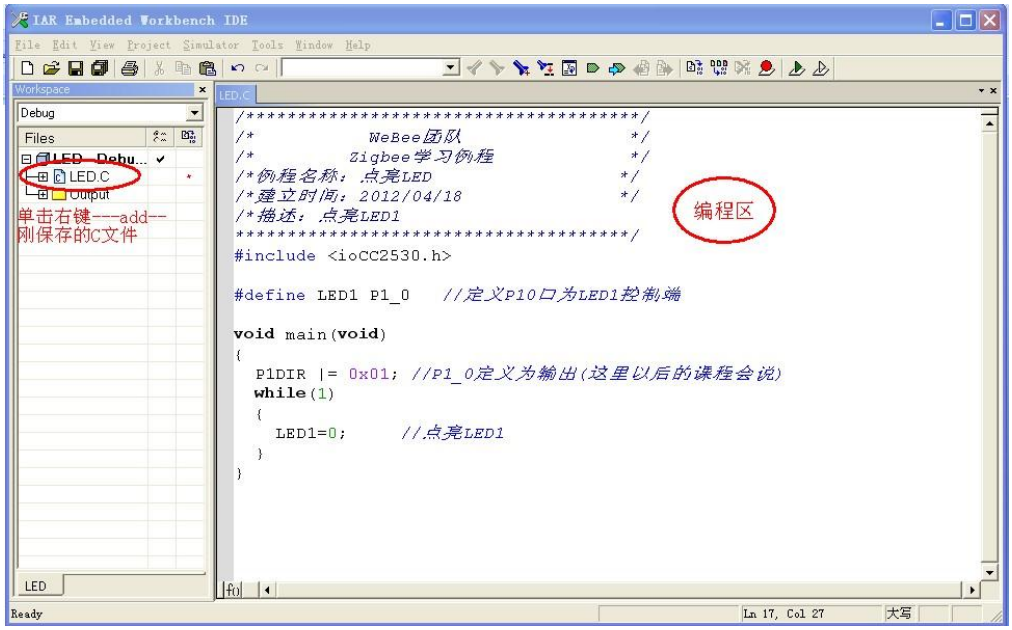


图 1.32 输入全部代码

第四步：我们还需要在 IAR 里配置一下几个选项。打开 Project---Options，General Options 配置如图 1.33 所示，单击圆圈所示按钮，先向上返回上一级目录，然后打开 Texas Instruments 文件夹，选择 CC2530F256 芯片。**（这里大家务必注意，只是第 2 章基础实验需要配置，第 3 章以后协议栈使用 TI 默认的即可，无需配置，配置了会出错！）**

选择 Linker—Config—Linker command file 选项。单击图 1.35 所示按钮，导出配置文件，先向上返回上一级目录，然后打开 Texas Instruments 文件夹，选择 lnk51ew_cc2530F256.xcl（这里是使用 CC2530F256 芯片）。

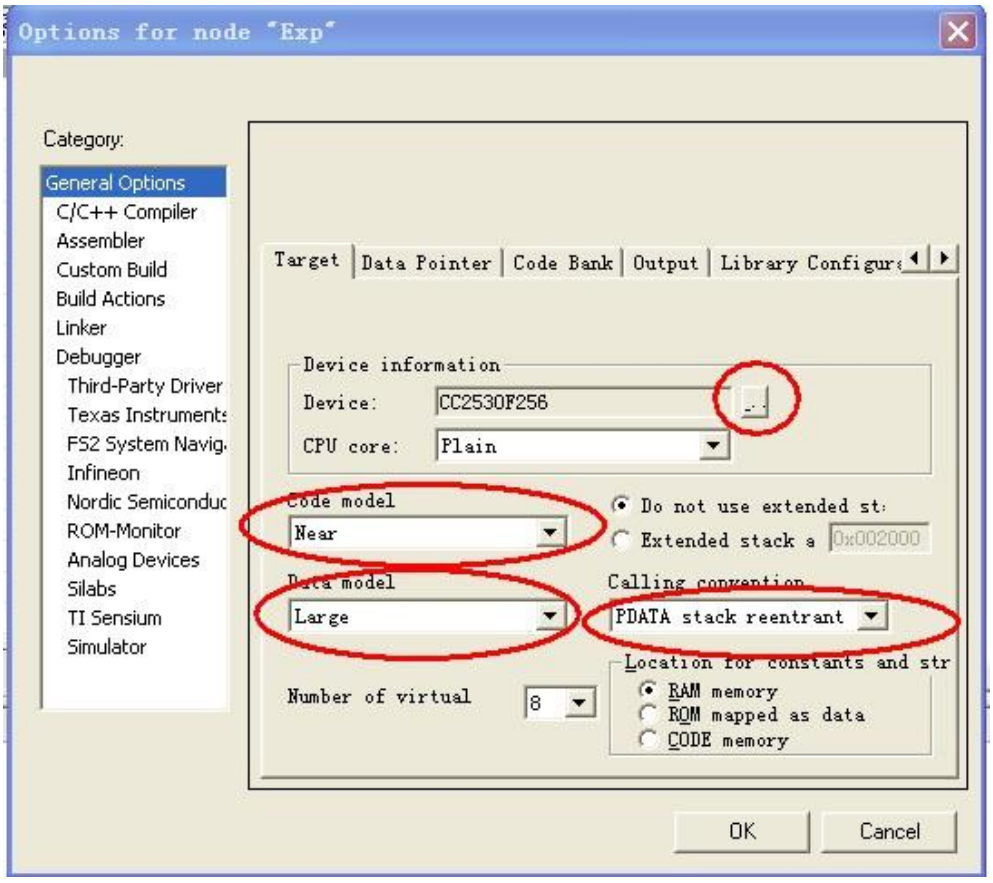


图 1.33 General Options 参数

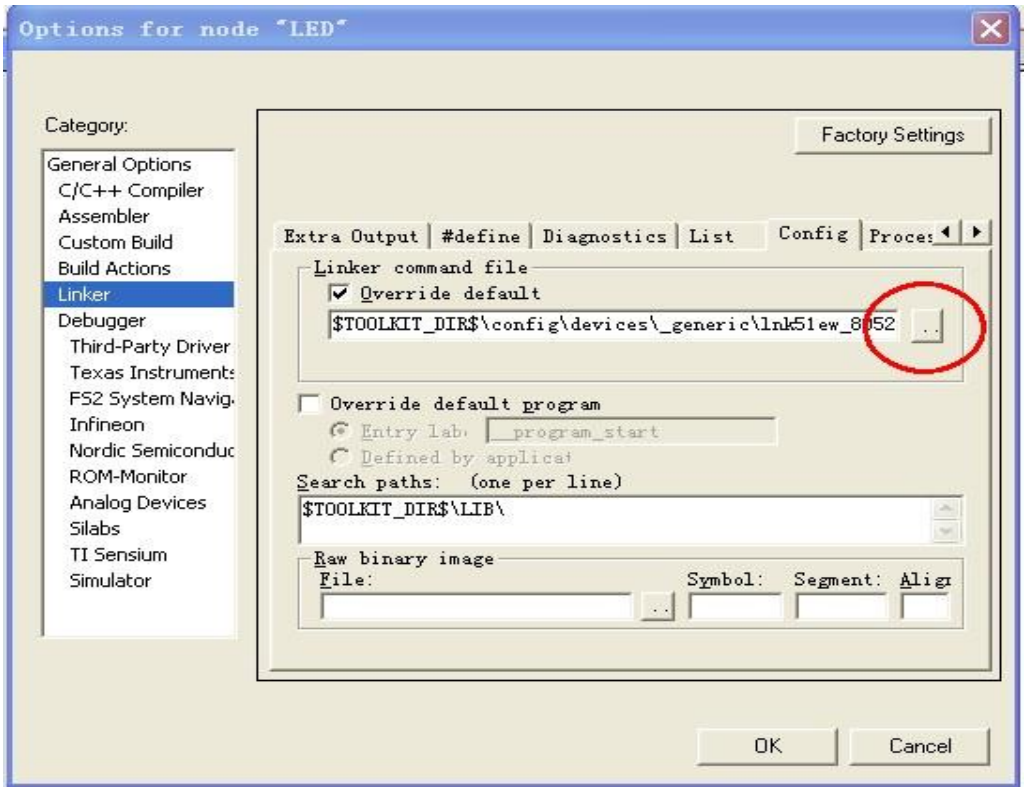


图 1.34 Linker-Config 配置

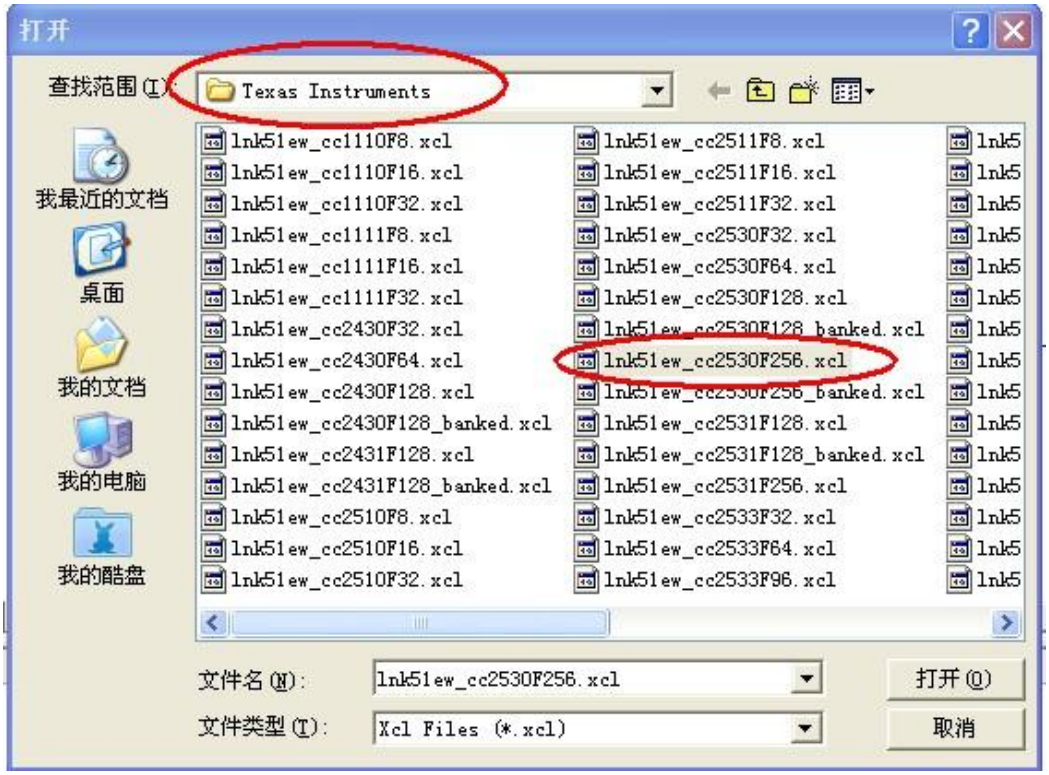


图 1.35

第五步：然后在 Debugger 选项的 Driver 里选择 Texas Instruments（使用编程器仿真）,下面选择 io8051.ddf 文件，如图 1.36 所示。至此，基本配置已经完成。其它配置以后需要用到时我们会提及。

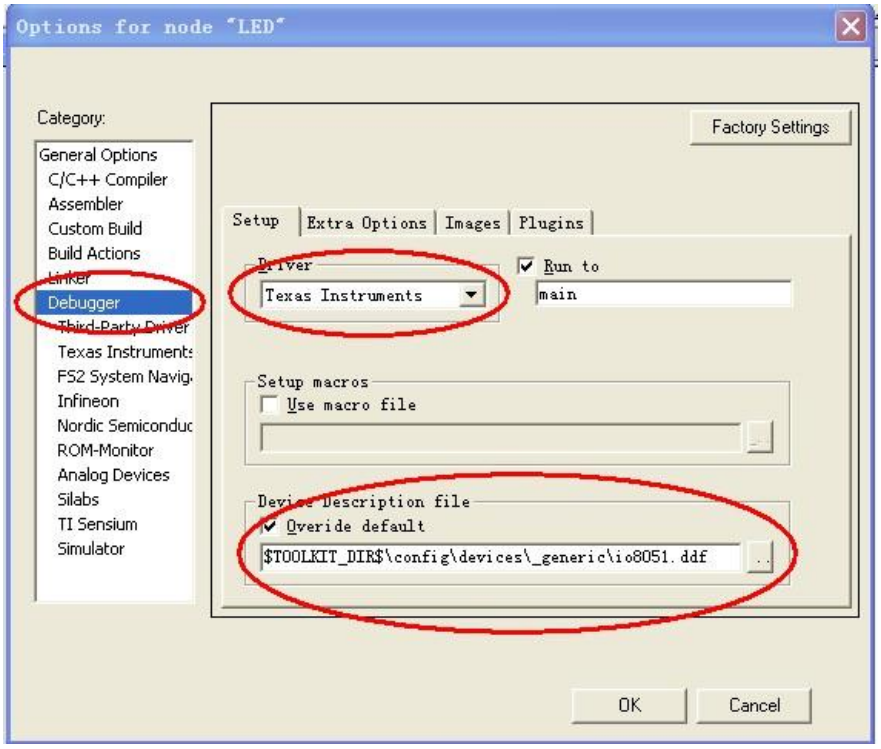


图 1.36 Debugger 参数配置

第六步：Project-Make 编译后显示 0 错误和 0 警告。将网蜂 CC DEBUGGER 和开发板连接好，然后点击:Project-Download and Debug (下载与仿真)。快捷键如图 1.37 所示：

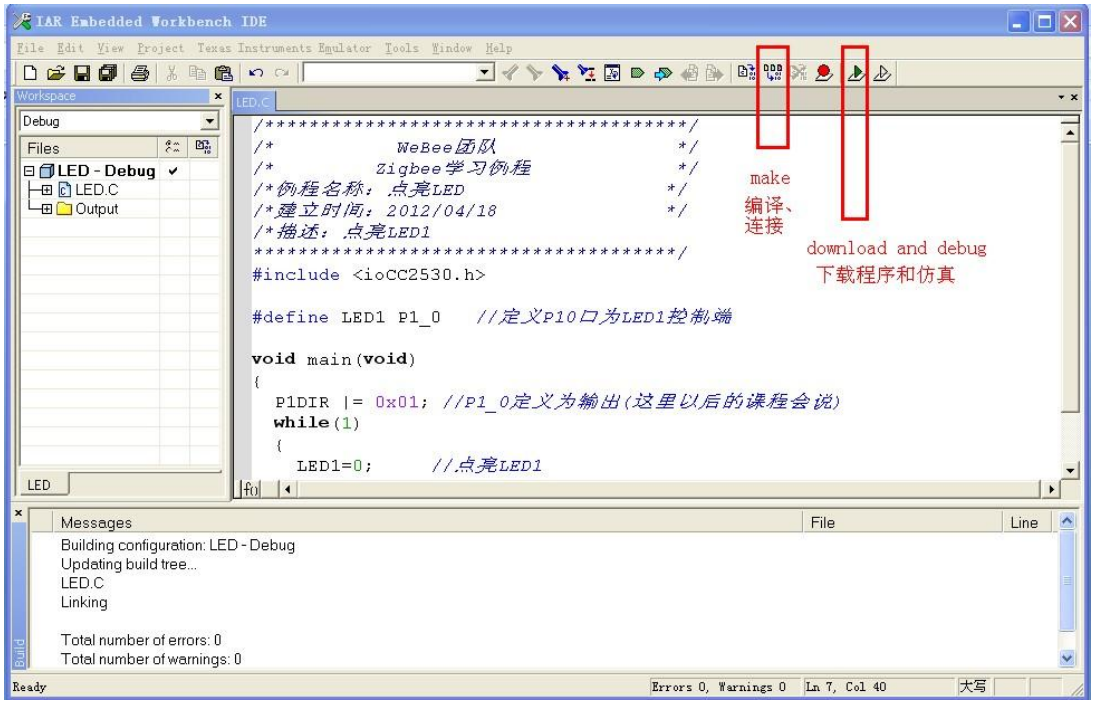


图 1.37

程序在下载中：

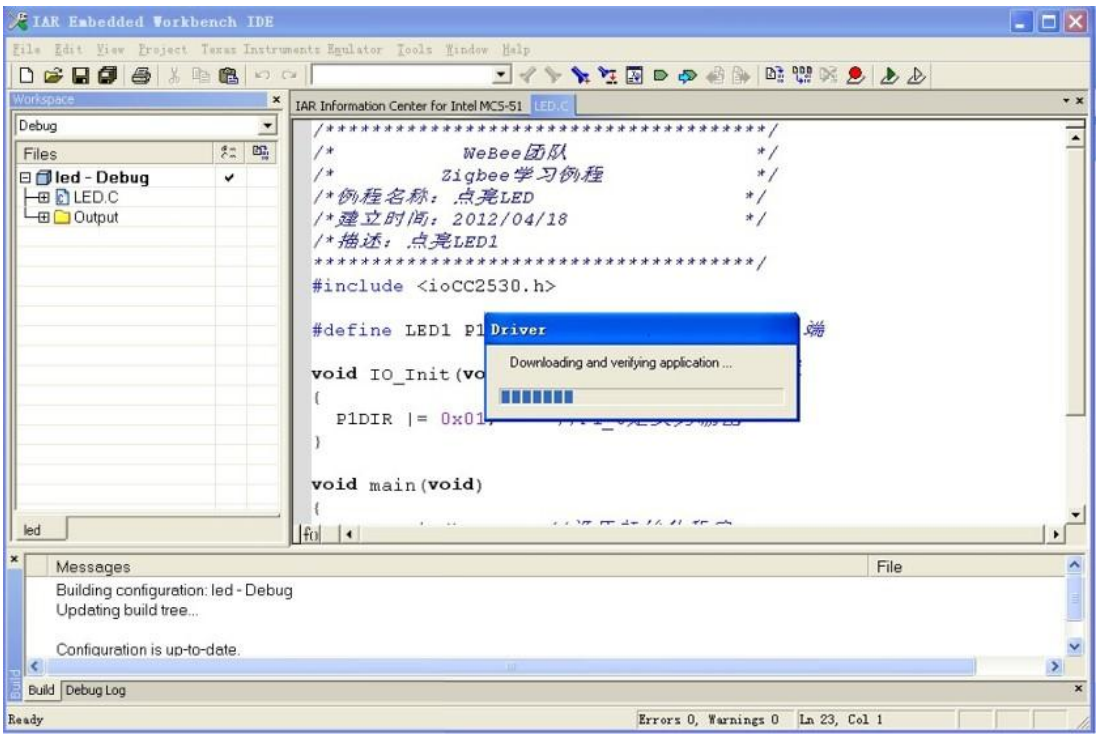


图 1.38

下载完成，进入仿真调试界面，常用按钮如图 1.39 所示。

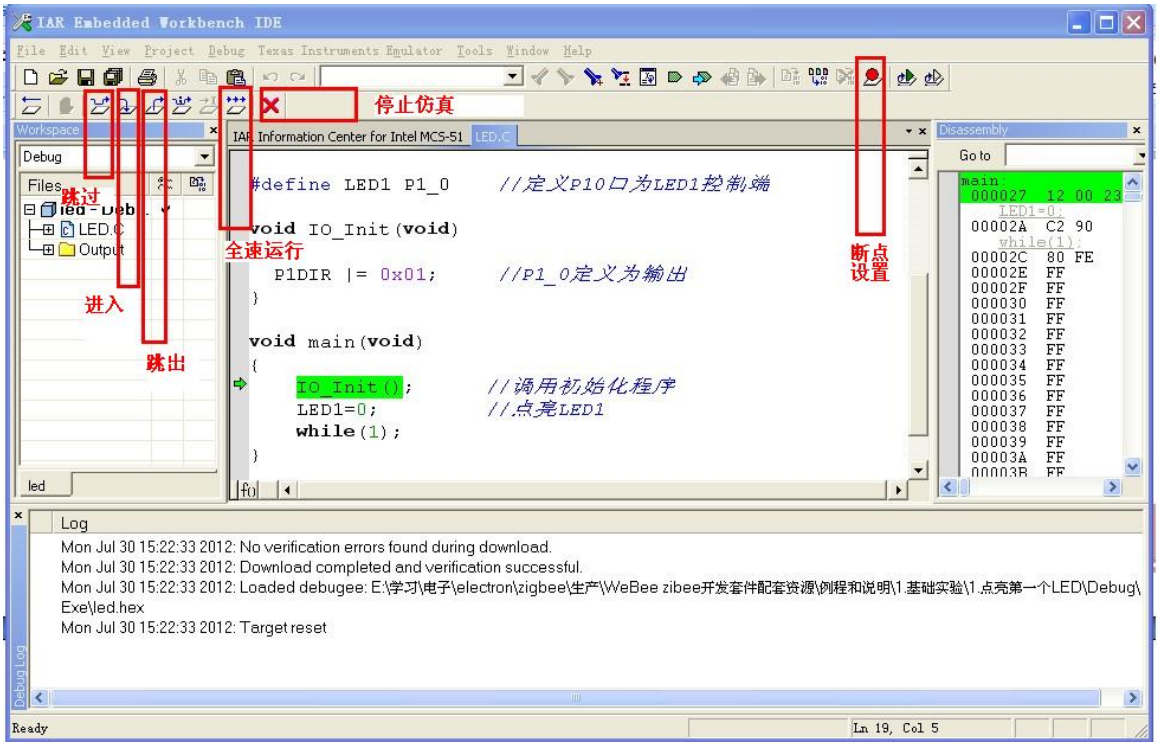


图 1.39 仿真界面

点击 GO(全速运行)，程序执行。使用 zigbee 仿真器可以直接在 IAR 中下载程序并调试。**仿真结束后程序仍然保留在芯片 flash 内，相当于烧写工具。非常方便。**

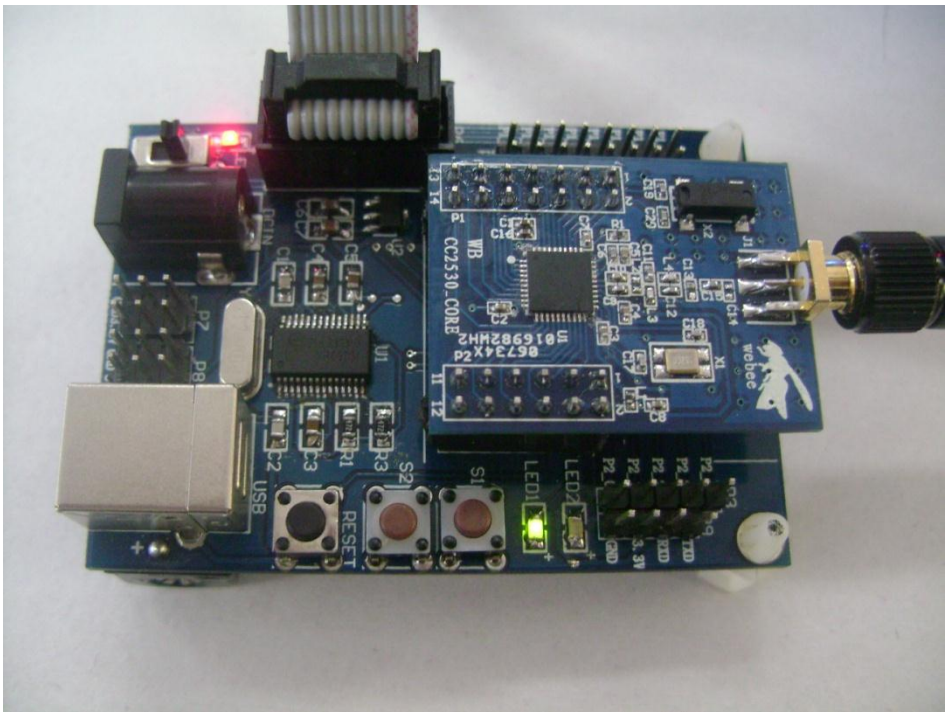


图 1.40 全速运行后，LED1 被点亮

至此，我们已经完成了 Zigbee CC2530 基于 IAR 开发环境的操作流程。无论是基础实验还是协议栈编程，其方法大同小异。通过本章学习希望你能掌握开发流程。为接下来的实验内容铺好路。

1.4 附录：

1.4.1 使用 TI SmartRF Flash Programmer 下载程序

现在大部分的 CC DEBUGGER 都支持在 IAR 编译环境中进行程序的下载和调试，非常方便。在这里我们补充一下另一种程序烧写方法，使用 TI SmartRF Flash Programmer。

第一步：配置编译器使生成.hex 文件(此方法仅仅适用于基础实验,不适合协议栈)。如图 1.41 和图 1.42。配置后点击 make 编译后，会在工程目录下的 Debug—Exe 找到生成的.hex 文件。

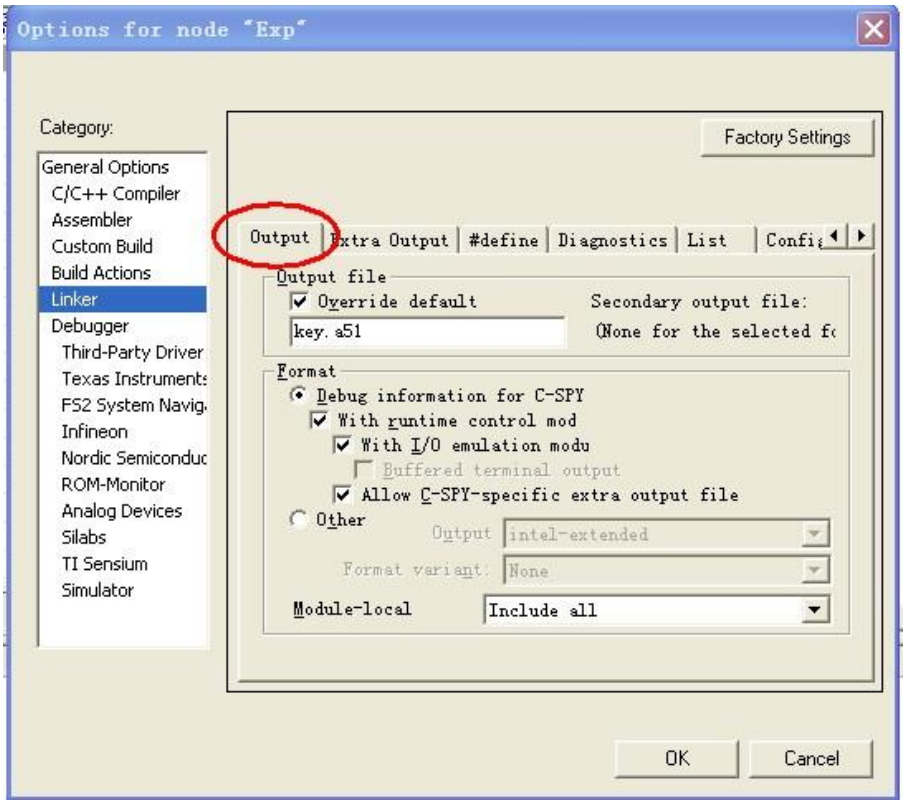


图 1.41

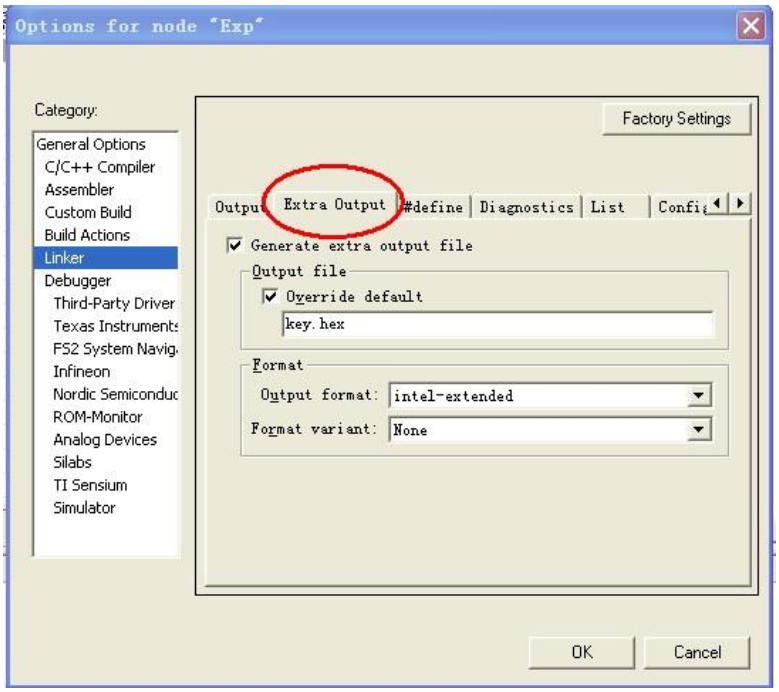


图 1.42

第二步：打开 TI SmartRF Flash Programmer，选择 System-on-chip(切记别选错)，添加刚刚生产的.hex 文件。点击程序下载按钮，.hex 文件变被下载到芯片内。

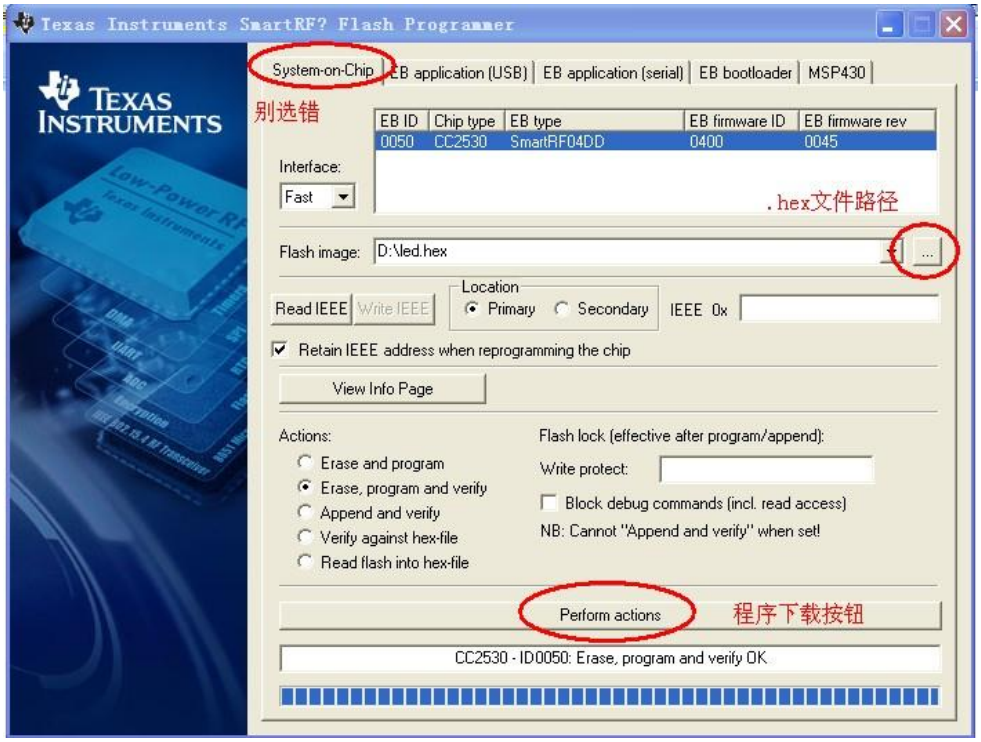


图 1.43 TI SmartRF Flash Programmer

第2章 基础实验

很多人说学习 ZigBee 重点在协议栈，这个是不争的道理。但是基础实验也有着重要的地位。基础实验说白了就是在玩增强型 51 单片机。学习本章将能令你快速掌握 CC2530 的编程方法，在以后学习完组网及数据传输的程序后我们会发现，很多应用必须是基于传感器和控制类芯片的，而这些恰好是基础实验的知识，好吧，废话少说，马上开始我们的基础实验讲解。

请先看基础实验讲解格式预览，每一节我们都会以以下形式讲解，图文并茂，务求达到快速理解的效果：

1) 标题：基础实验内容
2) 前言：简单介绍这个版块的应用
3) 实验现象：提前让大家知道此程序实现的现象。
4) 实验讲解：对寄存器、代码、编程方法详细讲解，代码为了方便大家会使用颜色区分，尽量做到像编译器一样。
5) 实验图片：记录程序下载到开发板上的图片示例。

2.1 点亮第一个 LED

前言：

相信大部分人开始学习 MCU 都会从点亮 LED 开始，我们 Zigbee 的学习也不例外，通过点亮第一个 LED 能让你对编译环境和程序架构有一定的认识，为以后的学习和更大型的程序打下基础，增加信心。

实验现象：程序实验点亮 LED1

实验讲解：我们先来看看 WeBee 底板的 LED 部分原理图：如图 2.1 所示。

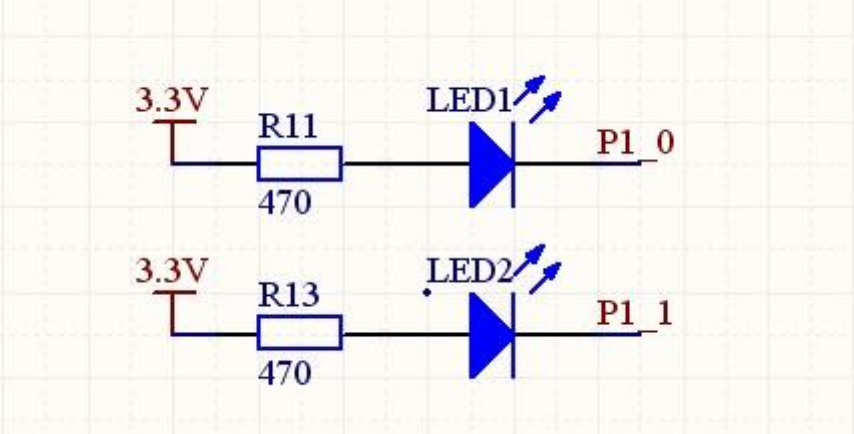


图 2.1 底板 LED 电路图

CC2530 的 IO 口配置我们需要配置三个寄存器 P1SEL 、P1DIR 、P1INP。IO 口功能如下表所示： （详细参考 CC2530 datasheet.pdf）

表 2.1 CC2530 IO 口寄存器

P0SEL(0XF3)	P0 [7:0]功能设置寄存器，默认设置为普通 I/O 口
P0INP(0X8F)	P0[7:0] 作输入口时的电路模式寄存器
P0(0X80)	P0[7:0] 可位寻址的 I/O 寄存器
P0DIR(0XFD)	P0 口输入输出设置寄存器，0：输入 ，1：输出

* P1SEL（0：普通 IO 口　1：第二功能）
* P1DIR（0：输入　　1：输出　　）
* P1INP（0：上拉/下拉　1：三态　　）

按照表格寄存器内容，我们对 LED1，也就是 P1_0 口进行配置，当 P1_0 输出低电平时 LED1 被点亮。所以配置如下：

```
P1SEL &=~0x01; //作为普通 IO 口
P1DIR |= 0x01; //P1_0 定义为输出
P1INP &=~0X01; //打开上拉
```

由于 CC2530 寄存器初始化时默认是：

```
P1SEL =0x00;
P1DIR =0x00;
P1INP =0X00;
```

所以 IO 口初始化我们可以简化初始化指令：

```
P1DIR |= 0x01; //P1_0 定义为输出
```

源程序代码（全）

```
/*************************************
程序描述：点亮 LED1
*************************************/
#include <ioCC2530.h>
#define LED1 P1_0     //定义 P10 口为 LED1 控制端
void IO_Init(void)
{
  P1DIR |= 0x01;      //P1_0 定义为输出
}
void main(void)
```

```
{
IO_Init();        //调用初始化程序
LED1=0;           //点亮 LED1
while(1);
}
```

实验图片：

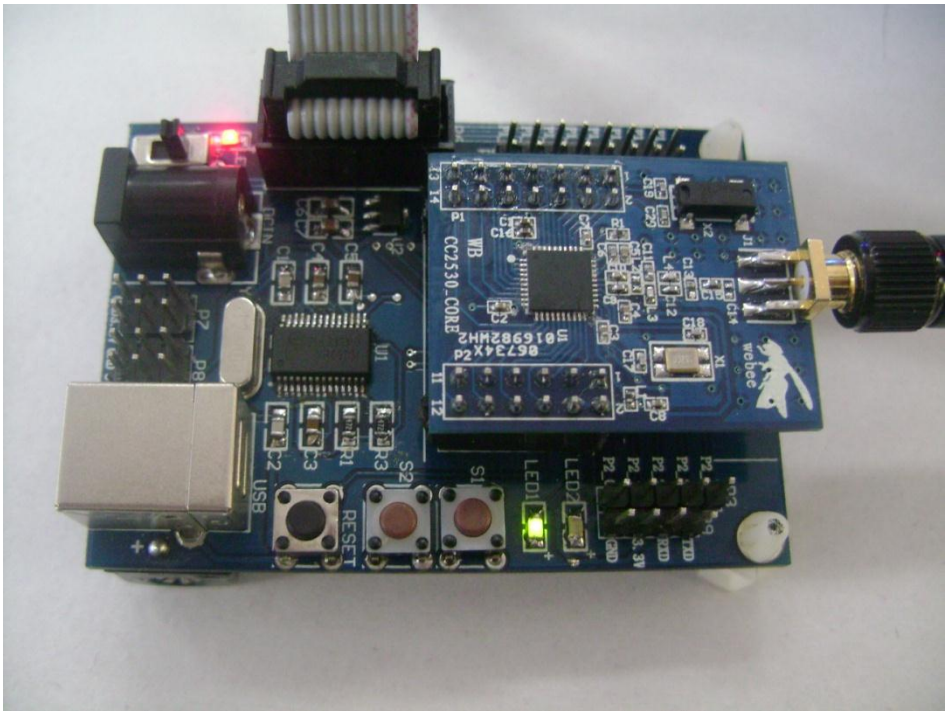


图 2.2　点亮 LED1

2.2 按键

前言：

相信大家经过例程 1 点亮 LED 实验后对 CC2530 的编程以及 IAR 的编译方法有一定的了解。我们来讲解一下 zigbee 模块的按键实验，按键是实现人机交互必不可少的东西，我们实验就用来实现按键控制 LED。

实验现象：依次按下按键 S1 控制 LED1 的亮和灭

实验讲解：我们先来看看 WeBee 底板的 KEY 和 LED 部分原理图：如图 2.3 所示。

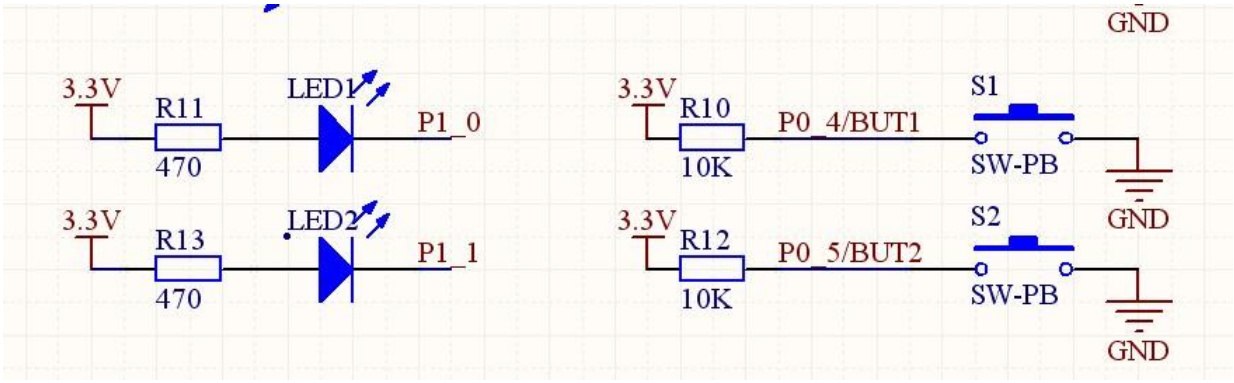


图 2.3 LED 和按键电路

CC2530 的 IO 口配置我们需要配置三个寄存器 P1SEL 、P1DIR 、P1INP。功能如下表所示： （详细参考 CC2530 datasheet.pdf）

表 2.2

P0SEL(0XF3)	P0 [7:0]功能设置寄存器，默认设置为普通 I/O 口
P0INP(0X8F)	P0[7:0] 作输入口时的电路模式寄存器
P0(0X80)	P0[7:0] 可位寻址的 I/O 寄存器
P0DIR(0XFD)	P0 口输入输出设置寄存器，0：输入 ，1：输出

* P1SEL (0：普通 IO 口 1：第二功能)

* P1DIR (0：输入 1：输出)

* P1INP (0：上拉/下拉 1：三态)

按照表格寄存器内容，我们对 LED1 和按键 S1，也就是 P1.0 和 P0.4 口进行配置，当 P1.0 输出低电平时 LED1 被点亮,S1 按下时 P0.4 被拉低。所以配置如下：

LED1 初始化：

```
P1SEL &=~0x01;  //作为普通 IO 口
P1DIR |= 0x01;  //P1_0 定义为输出
P1INP &=~0X01;  //打开上拉
```

按键 S1 初始化：

```
P0SEL &= ~0X10;     //设置 P04 为普通 IO 口
P0DIR &= ~0X10;     //按键在 P04 口，设置为输入模式
P0INP &= ~0x10;     //打开 P04 上拉电阻,不影响
```

由于 CC2530 寄存器初始化时默认是：

```
P1SEL = 0x00;
P1DIR = 0x00
P1INP = 0X00;
```

所以 IO 口初始化我们可以简化初始化指令：

```
P1DIR |= 0x01;     //P1_0 定义为输出
P0DIR &= ~0X10;    //按键在 P04 口，设置为输入模式
```

源程序代码（全）

```
/*************************************
程序描述：依次按下按键 S1 控制 LED1 的亮和灭
*************************************/
#include <ioCC2530.h>

#define uint unsigned int
#define uchar unsigned char

//定义控制 LED 灯的端口
#define LED1 P1_0          //LED1 为 P1.0 口控制
#define KEY1 P0_4        //KEY1 为 P0.1 口控制

//函数声明
void Delayms(uint);      //延时函数
void InitLed(void);      //初始化 LED1
void KeyInit();                    //按键初始化
uchar KeyScan();                   //按键扫描程序

/*****************************
延时函数
*****************************/
void Delayms(uint xms)    //i=xms 即延时 i 毫秒
{
 uint I,j;
 for(i=xms;i>0;i--)
   for(j=587;j>0;j--);
}
/*****************************
```

```
        LED 初始化函数
*****************************/
void InitLed(void)
{
   P1DIR |= 0x01;        //P1_0 定义为输出
   LED1 = 1;          //LED1 灯熄灭
}
/*****************************
      按键初始化函数
*****************************/
void InitKey()
{
  P0SEL &= ~0X10;     //设置 P04 为普通 IO 口
  P0DIR &= ~0X10;    //按键在 P04 口，设置为输入模式
  P0INP &= ~0x10;    //打开 P04 上拉电阻,不影响
}
/*****************************
      按键检测函数
*****************************/
uchar KeyScan(void)
{
  if(KEY1==0)
  {   Delayms(10); //去抖动
      if(KEY1==0)
      {
        while(!KEY1);  //松手检测
        return 1;      //有按键按下
      }
  }
```

```
  return 0;              //无按键按下
}
/*****************************
      主函数
*****************************/
void main(void)
{
   InitLed();    //调用初始化函数
   InitKey();
        while(1)
   {
         if(KeyScan())         //按键改变 LED 状态
           LED1=~LED1;
        }
}
```

实验图片：

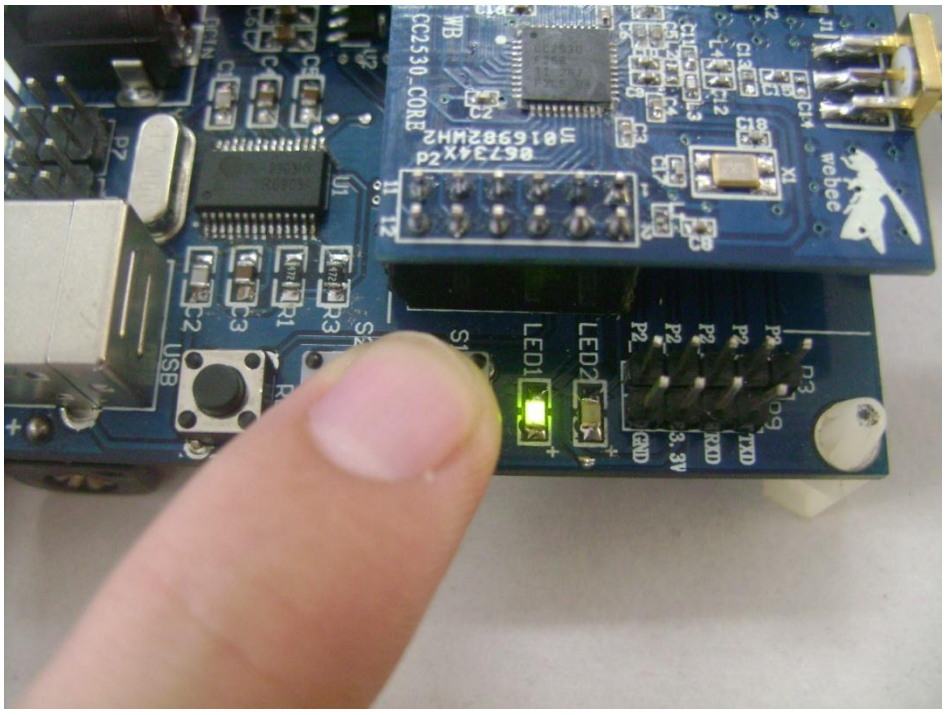


图 2.4 按键控制 LED1

2.3 外部中断

前言：

中断在 MCU 里面应用是非常广泛的，比如应用在时钟上的按键，我们可以发现基本上是不怎么使用的，如果用中断方式来代替传统的扫描方式，能节省 CPU 资源。也就是具有良好的实时性，本节将讲述 CC2530 的中断应用。

实验现象：依次按下按键 S1 控制 LED1 的亮和灭

实验讲解：我们先来看看 WeBee 底板的 KEY 和 LED 部分原理图：如图 2.5 所示。

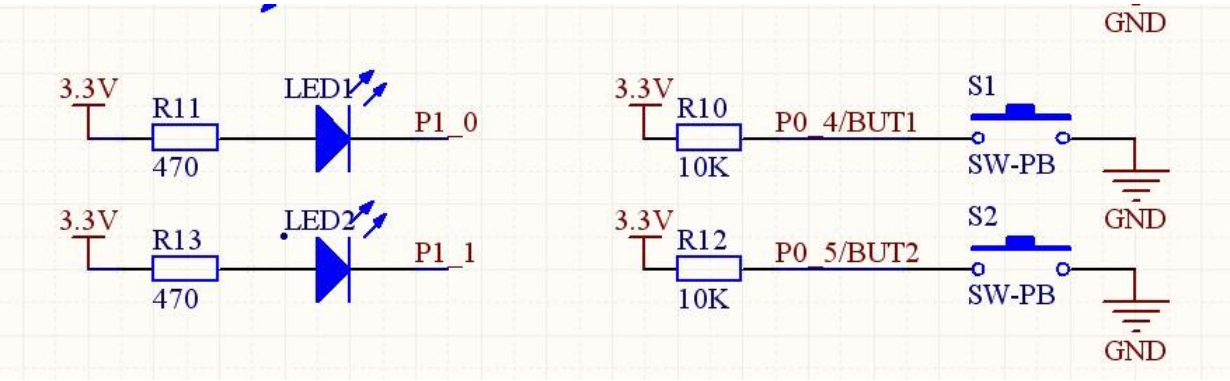


图 2.5

CC2530 的外部中断我们需要配置三个寄存器 P0IEN 、PICTL 、P0IFG、IEN1。IO 口配置请留意前 2 节教程内容。各寄存器功能如下表所示：（详细参考 CC2530 datasheet.pdf）

表 2.3

P0IEN(0XAB)	P0 [7:0]中断掩码寄存器. 0:关中断 1：开中断
PICTL（0X8C）	P 口的中断触发控制寄存器 Bit0 为 P0[0:7]的中断触发配置： 0：上升沿触发 1：下降沿触发
P0IFG(0X89)	P0[7:0] 中断标志位，在中断发生时，相应位置 1.
IEN1(0XB8)	Bit5 为 P0 [7:0]中断使能位：0:关中断 1：开中断

按照表格寄存器内容，我们对 LED1 和按键 S1，也就是 P1.0 和 P0.4 口进行配置，当 P1.0 输出低电平时 LED1 被点亮,S1 按下时 P0.4 产生外部中断从而控制 LED1 的亮灭。所以配置如下：

LED1 简化初始化：

```
P1DIR |= 0x01;  //P1_0 定义为输出
```

外部中断初始化：

```
P0IEN |= 0X10;  //P04 设置为中断方式
PICTL |= 0X01;  // 下降沿触发
IEN1  |= 0X20;   // 允许 P0 口中断;
P0IFG  = 0x00;   // 初始化中断标志位
```

源程序代码（全）

```
/*********************************************
程序描述：按键 S1 外部中断方式改变 LED1 状态
*********************************************/
#include <ioCC2530.h>

#define uint unsigned int
#define uchar unsigned char

//定义控制 LED 灯的端口
#define LED1 P1_0          //定义 LED1 为 P1.0 口控制
#define KEY1 P0_4        //中断口
//函数声明
void Delayms(uint);      //延时函数
void InitLed(void);      //初始化 P1 口
void KeyInit();                  //按键初始化
uchar KeyValue=0;
```

```
/*****************************
//延时函数
*****************************/
void Delayms(uint xms)    //i=xms 即延时 i 毫秒
{
 uint I,j;
 for(i=xms;i>0;i--)
   for(j=587;j>0;j--);
}
/*****************************
LED 初始化程序
*****************************/
void InitLed(void)
{
   P1DIR |= 0x01; //P1_0 定义为输出
   LED1 = 1;        //LED1 灯熄灭
}
/*****************************
KEY 初始化程序 - 外部中断方式
*****************************/
void InitKey()
{
  P0IEN |= 0X10;  //P04 设置为中断方式
  PICTL |= 0X10;  // 下降沿触发
  IEN1  |= 0X20;   // 允许 P0 口中断;
  P0IFG &= ~0x10;   // 初始化中断标志位
  EA = 1;          //开总中断
}
/*****************************
```

```
            中断处理函数
    ****************************/
#pragma vector = POINT_VECTOR     //格式：#pragma vector = 中断向量，
                                  // 紧接着是中断处理程序
      __interrupt void P0_ISR(void)
     {
      Delayms(6);              //去除抖动
      if(KEY1==0)
      {
      LED1=~LED1;               //改变 LED1 状态
      P0IFG &= ~0x10;           //清中断标志
      P0IF = 0;                 //清中断标志
      } }
    /*****************************
              主函数
    *****************************/
    void main(void)
    {
       InitLed();    //调用初始化函数
       InitKey();
    while(1);    //等待外部中断
    }
```

实验图片：

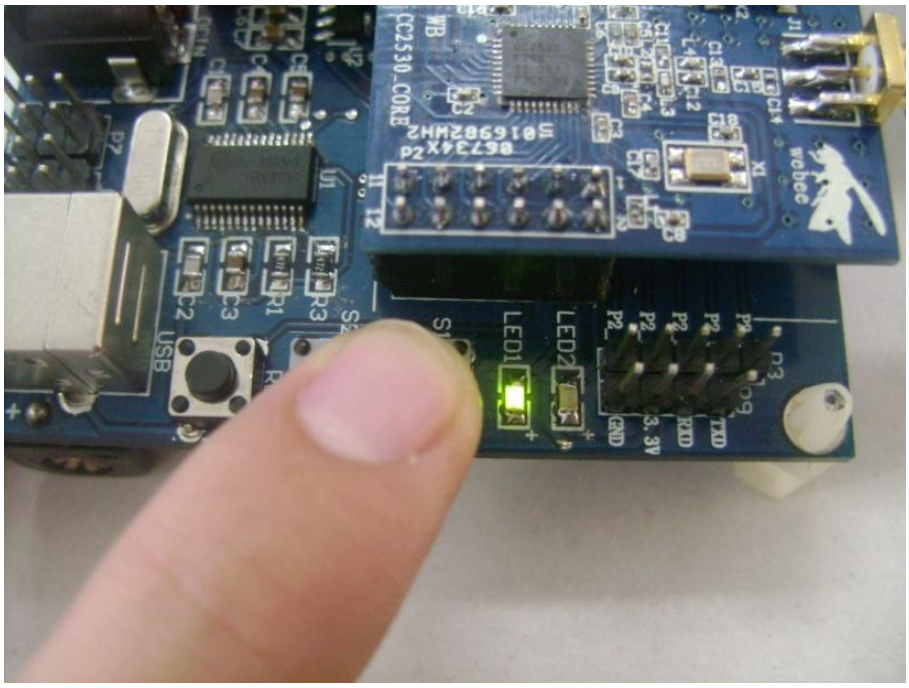


图 2.6 外部中断实验

2.4 定时器

前言：

人类最早使用的定时工具是沙漏或水漏，但在钟表诞生发展成熟之后，人们开始尝试使用这种全新的计时工具来改进定时器，达到准确控制时间的目的。MCU 的定时器博大精深，由普通定时计算、到 CPU 的分时复用，无不体现定时器的巨大作用。

实验现象：分别利用定时器 T1 和 T3 使 LED 周期性闪烁

实验讲解：我们先来看看 WeBee 底板的 LED 部分原理图：如图 2.7 所示。

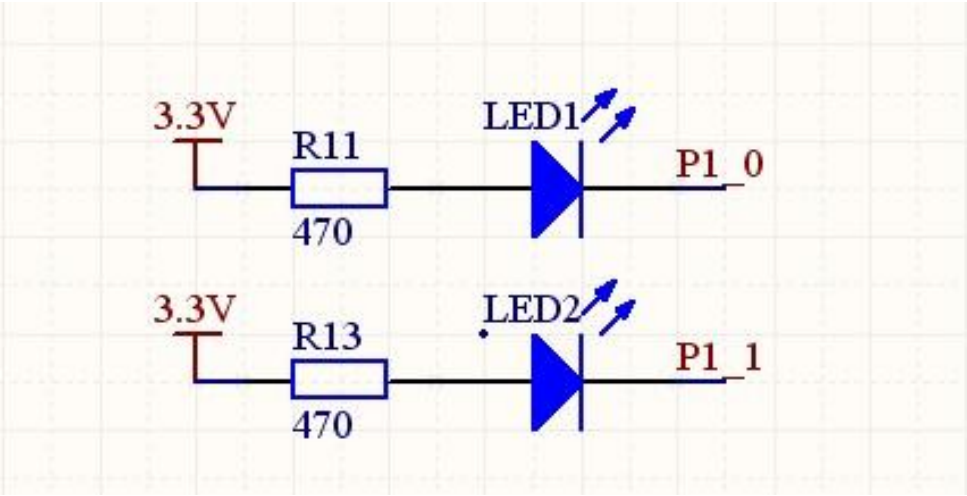


图 2.7 LED 电路图

2.4.1 定时器 T1(查询方式)

CC2530 的 T1 定时器（16 位）我们需要配置三个寄存器 T1CTL，T1STAT，IRCON。IO 口配置请留意第一节教程内容。各寄存器功能如下表所示：（详细参考 CC2530 datasheet.pdf）

表 2.4 定时器寄存器

T1CTL(0XE4)	Timer1 控制寄存器：
	Bit3:Bit2： 定时器时钟分频倍数选择：
	00：不分频 01 ：8 分频 10： 32 分频 11 ： 128 分频
	Bit1:Bit0： 定时器模式选择：
	00： 暂停 01： 自动重装 0X0000-0XFFFF 10： 比较计数 0X0000-T1CC0 11 ： PWM 方式
T1STAT(0XAF)	Timer1 状态寄存器：
	Bit5: OVFIF 定时器溢出中断标志，在计数器达到计数终值时置位 1.
	Bit4: 定时器 1 通道 4 中断标志位 Bit3: 定时器 1 通道 3 中断标志位 Bit2: 定时器 1 通道 2 中断标志位 Bit1: 定时器 1 通道 1 中断标志位 Bit0: 定时器 1 通道 0 中断标志位
IRCON(0XC0)	中断标志位寄存器：

按照表格寄存器内容，我们对 LED1 和定时器 1 寄存器进行配置。通过定时器 T1 查询方式控制 LED1 以 1S 的周期闪烁。具体配置如下：

LED1 简化初始化：

```
P1DIR |= 0x01;          //P1_0 定义为输出
```

定时器 1 初始化：

```
T1CTL  = 0x0d;          //128 分频，自动重装 0X0000-0XFFFF
T1STAT = 0x21;          //通道 0, 中断有效
```

源程序代码（全）

```
/*************************************
程序描述：通过定时器 T1 查询方式控制
          LED1 周期性闪烁
*************************************/
#include <ioCC2530.h>

#define uint unsigned int
#define uchar unsigned char

//定义控制 LED 灯的端口
#define LED1 P1_0        //定义 LED1 为 P10 口控制

//函数声明
void Delayms(uint xms);      //延时函数
void InitLed(void);          //初始化 P1 口
void InitT1();               //初始化定时器 T1
```

```
/*****************************
//延时函数
*****************************/
void Delayms(uint xms)      //i=xms  即延时 i 毫秒
{
  uint I,j;
  for(i=xms;i>0;i--)
     for(j=587;j>0;j--);
}
/*****************************
//初始化程序
*****************************/
void InitLed(void)
{
P1DIR |= 0x01;     //P1_0 定义为输出
LED1 = 1;          //LED1 灯初  始化熄灭
}
//定时器初始化
void InitT1()              //系统不配置工作时钟时使用内部 RC 振荡器，即
16MHz
{
   T1CTL = 0x0d;           //128 分频，自动重装  0X0000-0XFFFF
}
/*****************************
      主函数
*****************************/
void main(void)
{
          uchar count;
```

```
    InitLed();                          //调用初始化函数
        InitT1();
    while(1)
    {
        if(IRCON>0)                     //查询方式
        { IRCON=0;
          if(++count==1)                //约 1s 周期性闪烁
          {
            count=0;
            LED1 = !LED1;               //LED1 闪烁
          }
        }
    }
}
```

重点：系统在不配置工作频率时默认为 2 分频，即 32M/2=16M，所以定时器每次溢出时 T=1/(16M/128)*65536≈0.5s，所以总时间 Ta=T*count=0.5*1=0.5S 切换 1 次状态。所以看起来是 1S 闪烁 1 次。

2.4.2 定时器 T3(中断方式)

CC2530 的 T3 定时器主要是配置三个寄存

T3CTL, T3CCTL0, T3CC0, T3CCTL1, T3CC1。

如下表所示。

表 2.5

T3CTL(0XCB)	Timer3 控制寄存器：
	Bit7:Bit5： 定时器时钟分频倍数选择：
	000：不分频 001：2 分频 010： 4 分频 011： 8 分频 100：16 分频 101：32 分频 110： 64 分频 111：128 分频
	Bit4： T3 起止控制位
	Bit3： 溢出中断掩码 0：关溢出中断 1：开溢出中断
	Bit2： 清计数值 高电平有效
	Bit1:Bit0: T3 模式选择 00：自动重装 0X00-0XFF 01： DOWN (从 T3CC0 到 0X00 计数一次) 10： 模计数（反复从 0X00 到 T3CC0 计数） 11 ： UP/DOWN(反复从 0X00 到 T3CC0 计数再到 0X00)
T3CCTL0(0XCC)	T3 通道 0 捕获/ 比较控制寄存器：
	Bit6: T3 通道 0 中断掩码 0：关中断 1：开中断
	Bit5: Bit3 ： T3 通道 0 比较输出模式选择
	Bit2: T3 通道 0 模式选择： 0：捕获 1 ：比较
	Bit1: Bit0: T3 通道 0 捕获模式选择 00 没有捕获 01 上升沿捕获 10 下降沿捕获 11 边沿捕获
T3CC0(0XCD)	T3 通道 0 捕获/ 比较值寄存器

T3CCTL1(0XCE)	T3 通道 1 捕获/ 比较控制寄存器：
	Bit6: T3 通道 1 中断掩码 0：关中断 1：开中断
	Bit5: Bit3 ： T3 通道 1 比较输出模式选择
	Bit2: T3 通道 1 模式选择： 0：捕获 1 ：比较
	Bit1: Bit0: T3 通道 1 捕获模式选择 00 没有捕获 01 上升沿捕获 10 下降沿捕获 11 边沿捕获
T3CC1(0XCF)	T3 通道 1 捕获/ 比较值寄存器

与 上例 A 中 T1 定时器查询方式的区别就是此处使用 T3 定时器（8 位），中断方式。寄存器配置如下：

```
T3CTL |= 0x08 ;              //开溢出中断
T3IE   = 1;                  //开总中断和 T3 中断
T3CTL |=0XE0;                //128 分频,128/16000000*N=0.5S,N=65200
T3CTL &= ~0X03;              //自动重装 00->0xff  65200/256=254(次)
T3CTL |=0X10;                //启动
EA = 1;                      //开总中断
```

源程序代码（部分）

```
/**************************************
程序描述：利用定时器 T3 中断方式控制
        LED1 状态周期性改变
**************************************/
…
…
//定时器初始化
void InitT3()
{
```

```
        T3CTL |= 0x08 ;          //开溢出中断
        T3IE = 1;                //开总中断和 T3 中断
        T3CTL|=0XE0;             //128 分
频,128/16000000*N=0.5S,N=65200
        T3CTL &= ~0X03;          //自动重装 00->0xff
65200/256=254(次)
        T3CTL |=0X10;            //启动
        EA = 1;                  //开总中断
    }
    /*************************
    //主函数
    *************************/
    void main(void)
    {
        InitLed();                //调用初始化函数
        InitT3();
        while(1){ }
    }
    /*************************************
          中断函数
    *************************************/
    #pragma vector = T3_VECTOR       //定时器 T3
     __interrupt void T3_ISR(void)
     {
       IRCON = 0x00;        //清中断标志, 也可由硬件自动完成
       if(++count>254)    //254 次中断后 LED 取反,闪烁一轮(约为 0.5 秒
时间)
       {
         count = 0;       // 计数清零
```

```
        LED1=~LED1;
}
 }
```

实验图片：

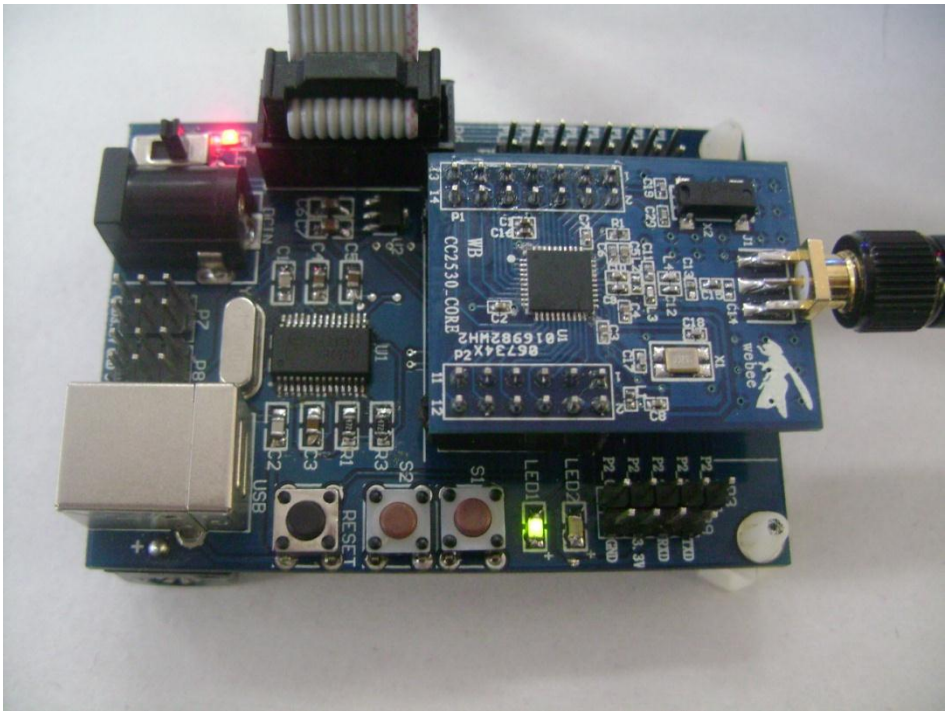


图 2.8 LED1 周期性闪烁

哈哈，你看不了闪烁吧。赶快把程序烧进去看看吧~~~^_^

2.5 串口通讯

前言：

无论学习哪款 MUC 串口对于我们进行实验调试都是非常方便实用的，我们可以把程序中涉及的某些中间量或者其他程序状态信息打印出来显示在电脑上进行调试，许多 MUC 和 PC 机通信都是通过串口来进行的。下面一起来学习 zigbee 的串口实验。

实验现象：实验将使用 WeBee 开发板实验 3 个串口功能。发送、收发、控制 LED。

实验讲解：我们先来看看 WeBee 底板的 USB 转串口部分电路原理图：如图 2.9 所示。

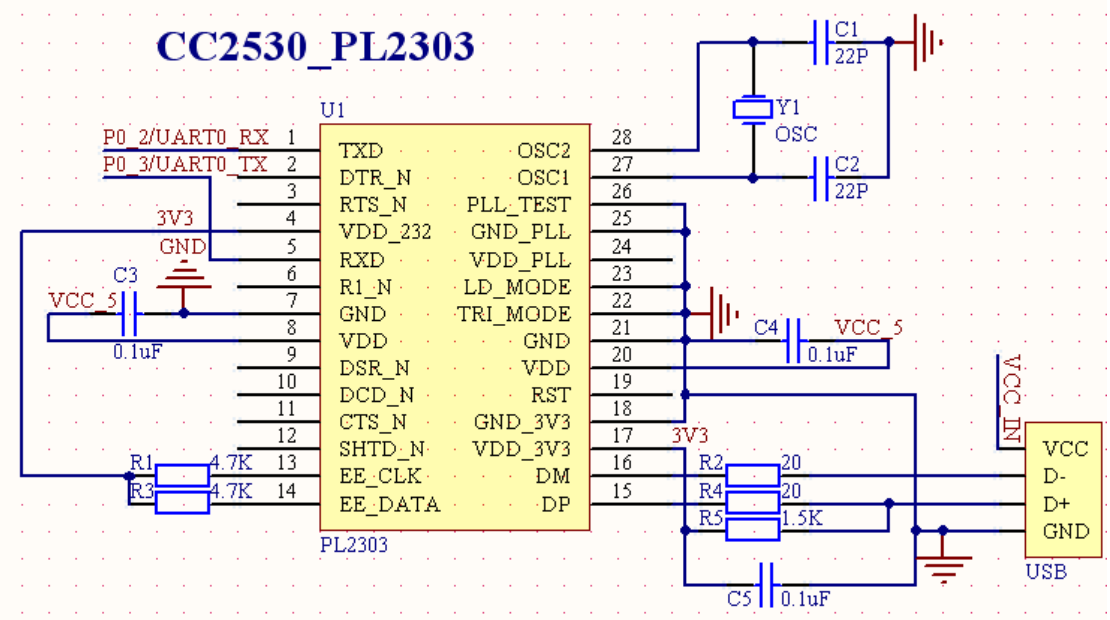


图 **2.9** PL2303 USB 转串口电路

2.5.1 串口发送(HELLO WEBEE)

查看 CC2530 的 datasheet 可知：

UART0 对应的外部设备 IO 引脚关系为：P0_2------RX

P0_3------TX

UART1 对应的外部设备 IO 引脚关系为：P0_5------RX

P0_4------TX

在 CC2530 中，USART0 和 USART1 是串行通信接口，它们能够分别运行于异步 USART 模式或者同步 SPI 模式。两个 USART 的功能是一样的，可以通过设置在单独的 IO 引脚上。

USART 模式的操作具有下列特点：

1、8 位或者 9 位负载数据

2、奇校验、偶校验或者无奇偶校验

3、配置起始位和停止位电平

4、配置 LSB 或者 MSB 首先传送

5、独立收发中断

6、独立收发 DMA 触发

注：在本次实验中，我们用到的是 UART0。

CC2530 配置串口的一般步骤：

1、 配置 IO，使用外部设备功能。此处配置 P0_2 和 P0_3 用作串口 UART0

2、 配置相应串口的控制和状态寄存器。此处配置 UART0 的工作寄存器

3、 配置串口工作的波特率。此处配置为波特率为 115200

本次实验串口相关的寄存器或者标志位有：U0CSR、U0GCR、U0BAUD、U0DBUF、UTX0IF。各寄存器功能如下表所示：　（详细参考 CC2530 datasheet.pdf）

表 2.6

U0CSR（UART0 控制和状态寄存器）	Bit7： MODE	0：SPI 模式
		1：UART 模式
	Bit6： RE	0：接收器禁止
		1：接收器使能
	Bit5： SLAVE	0：SPI 主模式
		1：SPI 从模式
	Bit4： FE	0：没有检测出帧错误
		1：收到字节停止位电平出错
	Bit3： ERR	0：没有检测出奇偶检验出错
		1：收到字节奇偶检验出错
	Bit2： RX_BYTE	0：没有收到字节
		1：收到字节就绪
	Bit1： TX_BYTE	0：没有发送字节
		1 写到数据缓冲区寄存器的最后字节已经发送
	Bit0： ACTIVE	0：USART 空闲
		1：USART 忙
U0GCR（UART0 通用控制寄存器）	Bit7： CPOL	0：SPI 负时钟极性
		1：SPI 正时钟极性
	Bit6： CPHA	0：当来自 CPOL 的 SCK 反相之后又返回 CPOL 时，数据输出到 MOSI；当来自 CPOL 的 SCK 返回 CPOL 反相时，输入数据采样到 MISO
		1:当来自 CPOL 的 SCK 返回 CPOL 反相时，数据输出到 M OSI ；当来自 CPOL 的 SCK 反相之后又返回 CPOL 时，输入数据采样到 MISO
	Bit5： ORDER	0：LSB 先传送
		1：MSB 先传送

	Bit[4-0]：BAUD_E	波特率指数值 BAUD_E 连同 BAUD_M 一起决定了 UART 的波特率
U0BAUD（UART0 波特率控制寄存器）	Bit[7-0]：BAUD_M	波特率尾数值 BAUD_M 连同 BAUD_E 一起决定了 UART 的波特率
U0DBUF （UART0 收发数据缓冲区）		串口发送/接收数据缓冲区
UTX0IF（发送中断标志）	中断标志 5 IRCON2 的 Bit1	0：中断未挂起
		1：中断挂起

串口的波特率设置可以从 CC2530 的 datasheet 中查得波特率由下式求得：

$$\text{波特率} = \frac{(256 + \text{BAUD\_M}) \times 2^{\text{BAUD\_E}}}{2^{28}} \times f$$

本次实验设置波特率为 115200bps,具体的参数设置如下：

表 16-1 32MHz 系统时钟的常用波特率设置

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

寄存器具体配置如下：

```
PERCFG = 0x00;          //位置 1 P0 口
P0SEL  = 0x0c;          //P0_2,P0_3 用作串口（外部设备功能）
P2DIR &= ~0XC0;         //P0 优先作为 UART0

U0CSR  |= 0x80;         //设置为 UART 方式
U0GCR  |= 11;
U0BAUD |= 216;          //波特率设为 115200
UTX0IF  = 0;            //UART0 TX 中断标志初始置位 0
```

串口发送函数请参考下面源程序：

源程序代码（全）

```
/*************************************
  描述：在串口调试助手上可以看到不停地
        收到 CC2530 发过来的：HELLO WEBEE
        波特率：115200bps
*************************************/
#include <ioCC2530.h>
#include <string.h>

#define   uint   unsigned int
#define   uchar unsigned char

//定义 LED 的端口
#define LED1 P1_0
#define LED2 P1_1

//函数声明
```

```
void Delay_ms(uint);
void initUART(void);
void UartSend_String(char *Data,int len);

char Txdata[14];      //存放”HELLO WEBEE     “共 14 个字符串
/*****************************************************************
延时函数
*****************************************************************/
void Delay_ms(uint n)
{
        uint I,j;
        for(i=0;i<n;i++)
        {
        for(j=0;j<1774;j++);
        }
}

void IO_Init()
{
P1DIR = 0x01;                 //P1_0IO 方向输出
LED1 = 1;                     //关 LED
}

/*****************************************************************
    串口初始化函数
*****************************************************************/
void InitUART(void)
{
PERCFG = 0x00;                //位置 1 P0 口
```

```
P0SEL = 0x0c;                //P0_2,P0_3 用作串口（外部设备功能）
P2DIR &= ~0XC0;              //P0 优先作为 UART0

U0CSR |= 0x80;               //设置为 UART 方式
U0GCR |= 11;
U0BAUD |= 216;                 //波特率设为 115200
UTX0IF = 0;                  //UART0 TX 中断标志初始置位 0
}
/************************************************************
串口发送字符串函数
************************************************************/
void UartSend_String(char *Data,int len)
{
  int j;
  for(j=0;j<len;j++)
  {
U0DBUF = *Data++;
while(UTX0IF == 0);
UTX0IF = 0;
  }
}
/************************************************************
主函数
************************************************************/
void main(void)
{
CLKCONCMD &= ~0x40;                    //设置系统时钟源为 32MHZ 晶振
while(CLKCONSTA & 0x40);               //等待晶振稳定为 32M
CLKCONCMD &= ~0x47;                    //设置系统主时钟频率为 32MHZ
```

```
IO_Init();
InitUART();
strcpy(Txdata,"HELLO WEBEE     ");        //将发送内容 copy 到 Txdata;
while(1)
    {
        UartSend_String(Txdata,sizeof("HELLO WEBEE     "));   //串口发送
数据
        Delay_ms(500);                                        //延时
        LED1=!LED1;                                           //标志发送状态
    }
}
```

实验 1 图片：

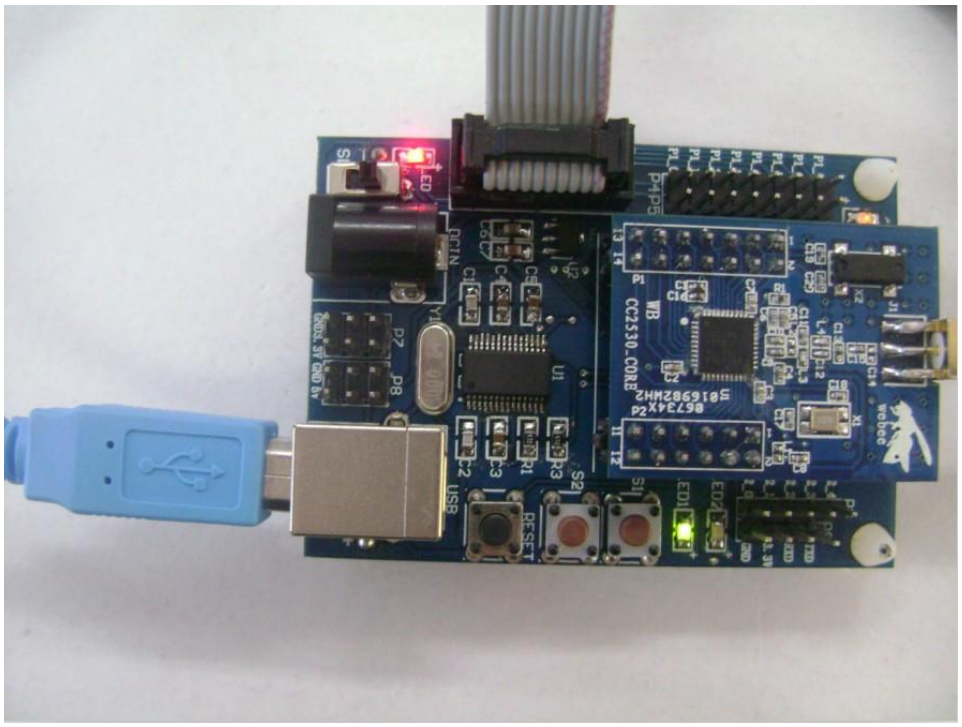


图 2.10　USB 转串口连接方法

图 2.11 上位机接收到发来的“HELLO WEBEE”

2.5.2 串口接收和发送(send & receive)

寄存器配置请参考上方实验 1 的表格。实验 1 较实验 1 增加了串口接收功能，故寄存器配置有所改变，如下。

```
CLKCONCMD &= ~0x40;          // 设置系统时钟源为 32MHZ 晶振
while(CLKCONSTA & 0x40);     // 等待晶振稳定
CLKCONCMD &= ~0x47;          // 设置系统主时钟频率为 32MHZ

PERCFG  = 0x00;             //位置 1 P0 口
POSEL   = 0x3c;             //P0_2,P0_3,P0_4,P0_5 用作串口,第二功能
P2DIR  &= ~0XC0;            //P0 优先作为 UART0 ，优先级

U0CSR  |= 0x80;               //UART 方式
U0GCR  |= 11;                 //U0GCR 与 U0BAUD 配合
U0BAUD |= 216;              // 波特率设为 115200
UTX0IF  = 0;                //UART0 TX 中断标志初始置位 1 （收发时候）
U0CSR  |= 0X40;             //允许接收
IEN0   |= 0x84;             // 开总中断，接收中断
```

源程序代码（部分） 请读者自行分析

```
/*************************************
程序描述：例以 abc#方式发送，#为结束符，
          返回 abc。波特率：115200bps
*************************************/
…
…
```

```
/*****************************************************************
        串口初始化函数
*****************************************************************/
    void InitUart()
    {
    CLKCONCMD &= ~0x40;             // 设置系统时钟源为 32MHZ 晶振
    while(CLKCONSTA & 0x40);        // 等待晶振稳定
    CLKCONCMD &= ~0x47;             // 设置系统主时钟频率为 32MHZ

    PERCFG = 0x00;                 //位置 1 P0 口
    P0SEL = 0x3c;                  //P0_2,P0_3,P0_4,P0_5 用作串口,第二功能
    P2DIR &= ~0XC0;                //P0 优先作为 UART0 ，优先级

    U0CSR |= 0x80;                 //UART 方式
    U0GCR |= 11;              //U0GCR 与 U0BAUD 配合
    U0BAUD |= 216;            // 波特率设为 115200
    UTX0IF = 0;               //UART0 TX 中断标志初始置位 1   （收发时候）
    U0CSR |= 0X40;            //允许接收
    IEN0 |= 0x84;             // 开总中断，接收中断
    }

/*****************************************************************
    串口发送字符串函数
*****************************************************************/
  void Uart_Send_String(char *Data,int len)
  {
    {
      int j;
      for(j=0;j<len;j++)
```

```
  {
    U0DBUF = *Data++;
    while(UTX0IF == 0);    //发送完成标志位
    UTX0IF = 0;
   }
  }
}

/*****************************
    主函数
*****************************/
void main(void)
{
  InitLed();                //调用初始化函数
  InitUart();
  while(1)
  {
     if(RXTXflag == 1)      //接收状态
     {
        LED1=1;             //接收状态指示
        if( temp != 0)
        {
       if((temp!=' #' )&&(datanumber<50)) //' #' 被定义为结束字
                                          符，最多能接收 50 个字
                                          符
           Rxdata[datanumber++] = temp;
           else
           {
             RXTXflag = 3;                    //进入发送状态
```

```
                LED1=0;                                    //关指示灯
              }
              temp  = 0;
            }
          }
          if(RXTXflag == 3)      //发送状态
          {
           LED2= 1;
           U0CSR &= ~0x40;       //禁止接收
           Uart_Send_String(Rxdata,datanumber); //发送已记录的字符串。
           U0CSR |= 0x40;        //允许接收
           RXTXflag = 1;         // 恢复到接收状态
           datanumber = 0;       //指针归 0
           LED2 = 0;             //关发送指示
          }
    }
    }
/**************************************************************
    串口接收一个字符：一旦有数据从串口传至 CC2530，则进入中断，将接收到的数据赋值给变量 temp.
**************************************************************/
    #pragma vector = URX0_VECTOR
      __interrupt void UART0_ISR(void)
     {
      URX0IF = 0;     // 清中断标志
      temp = U0DBUF;
     }
```

实验 2 图片：

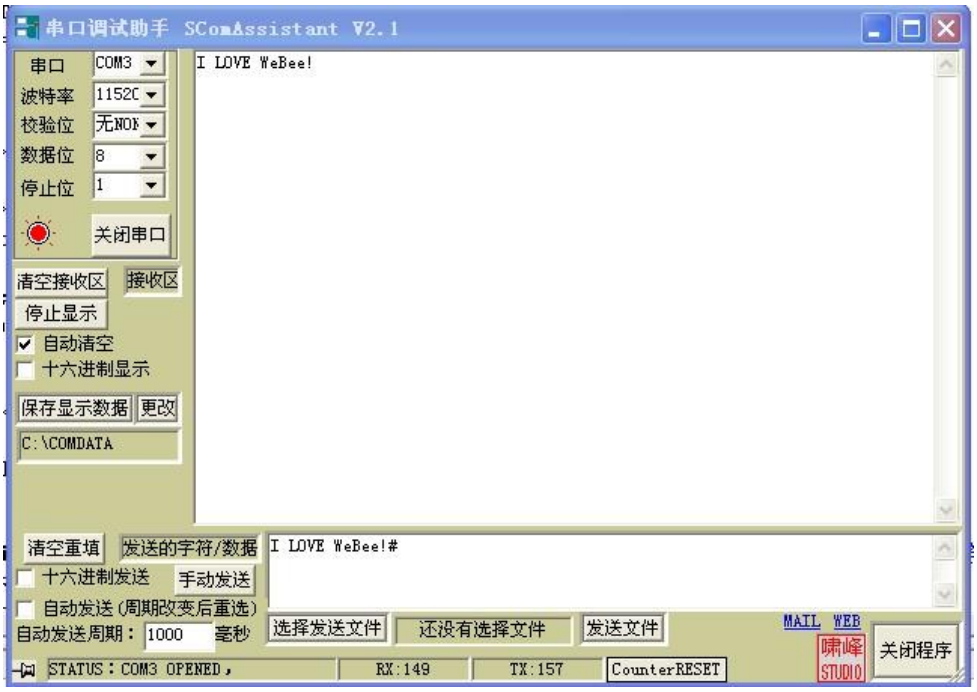


图 2.12　发送：I LOVE WeBee!#　接收到：I LOVE WeBee

2.5.3　UART0-控制 LED

发送和接收函数均和实验 2 相同，这里我们重点分析主函数判断代码：

```
/*****************************************
程序描述：依次发送 L1# L2# 指令分别控制
        LED1、LED2 亮灭，波特率：115200bps
*****************************************/
…
…
/*****************************
//主函数
*****************************/
void main(void)
{
```

```
InitLed();                    //调用初始化函数
InitUart();
while(1)
{
   if(RXTXflag == 1)          //接收状态
   {
      if( temp != 0)
      {
     if((temp!=' #' )&&(datanumber<3)) //' #' 被定义为结束字符,
                                       最//多能接收 50 个字符
         Rxdata[datanumber++] = temp;
         else
         {
           RXTXflag = 3;                  //进入发送状态
         }
          temp  = 0;
      }
 }

 if(RXTXflag == 3)                  //检测接收到的数据
 {
  if(Rxdata[0]==' L' )
switch(Rxdata[1]-48)              //很重要, ASICC 码转成数字,判
                                    断 L 后面第一个数
    {
 case 1:                     //如果是 L1
   {
     LED1=~LED1;                //低电平点亮
     break;
```

```
        }
      case 2:                    //如果是 L2
        {
          LED2=~LED2;
          break;
        }
       }
       RXTXflag = 1;
       datanumber = 0;           //指针归 0
      }
}
}
```

实验 3 图片：

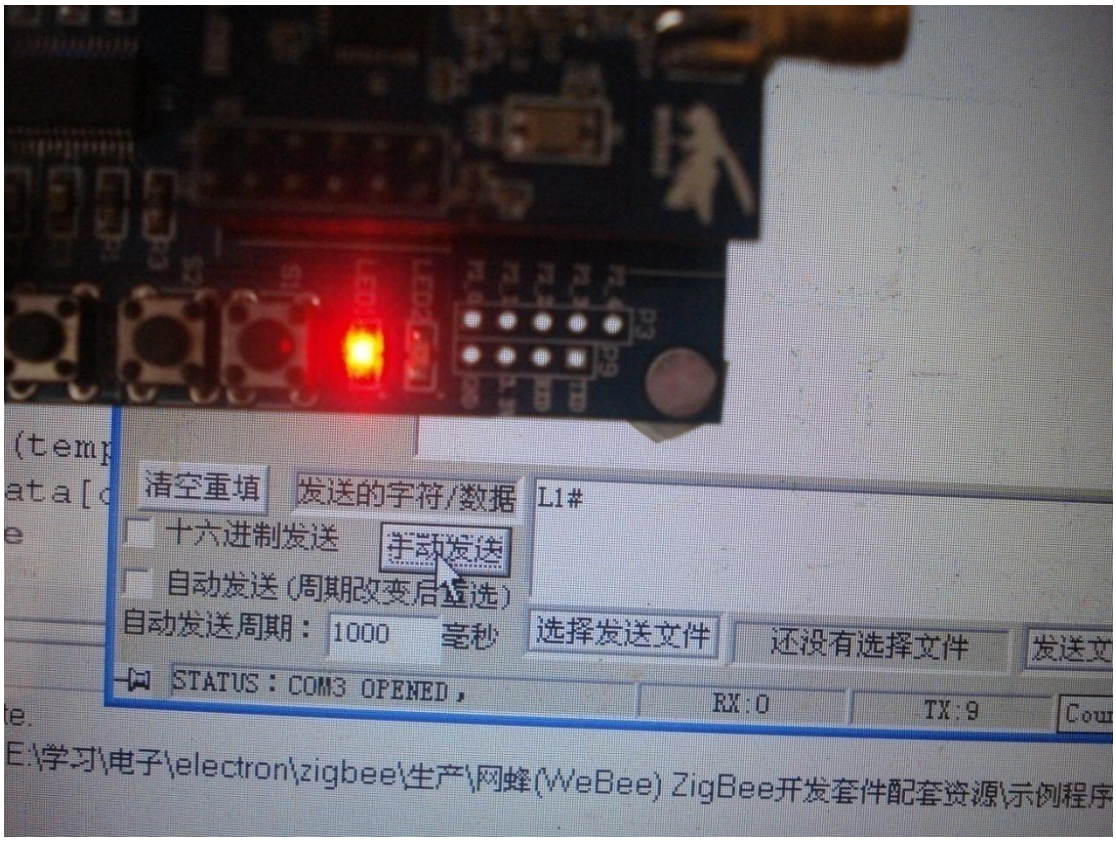


图 2.13 依次发送 L1#和 L2#

2.6 AD 控制（自带温度计）

前言：

温度传感器是我们学习 MCU 经常使用的传感器，在 CC2530 里就集成里片内的温度传感器，有人会想到如果芯片发热怎么办？这个你得问问 TI 的工程师了。而且部分芯片偏差较大，通常需要软件校准。不过这也不失为一个尝试。

实验功能：将采集到内部温度传感器信息通过串口发送到上位机。

实验讲解：CC2530 的内部温度检测需要配置的寄存器比较多，包括温度和 AD 的。CLKCONCMD,PERCFG,U0CSR,U0GSR,U0BAUD,CLKCONSTA,IEN0,U0DUB,ADCCON1,ADCCON3,ADCH,ADCL。各寄存器功能如下表所示：（详细参考 CC2530 datasheet.pdf）

表 2.7

ADCCON1(0XB4)	ADC 控制寄存器 1
	Bit7: EOC ADC 结束标志位 0：AD 转换进行中 1 ： AD 转换完成
	Bit6: ST 手动启动 AD 转换 0： 关 1 ： 启动 AD 转换(需要 Bit5:Bit4=11)
	Bit5: Bit4 AD 转换启动方式选择 00： 外部触发 01 ： 全速转换，不需要触发
	10： T1 通道 0 比较触发 11：手动触发
	Bit3: Bit2 16 位随机数发生器控制位 00： 普通模式 （13x 打开） 01： 开启 LFSR 时钟一次 （13x 打开） 10：保留位 11 ： 关
ADCCON2(0XB5)	序列 AD 转换控制寄存器 2

	Bit7:Bit6 SREF 选择 AD 转换参考电压 00：内部参考电压（1.25V） 01：外部参考电压 AIN7 输入 10：模拟电源电压 11 ：外部参考电压 AIN6-AIN7 差分输入
	Bit5: Bit4 设置 AD 转换分辨率 00： 64dec,7 位有效 01： 128dec,9 位有效 10： 256dec,10 位有效 11： 512dec,12 位有效
	Bit3: Bit0 设置序列 AD 转换最末通道，如果置位时 ADC 正在运行，则在完成序列 AD 转换后立刻开始，否则置位后立即开始 AD 转换，转换完成后自动清 0. 0000： AIN0 0001: AIN1 0010:AIN2 0011: AIN3 0100： AIN4 0101: AIN5 0110:AIN6 0111: AIN7 1000： AIN0-AIN1 差分 1001: AIN2-AIN3 差分 1010： AIN4-AIN5 差分 1011: AIN6-AIN7 差分 1100 ： GND 1101: 保留 1110 ： 温度传感器 1111 ： 1/3 模拟电源电压
ADCCON3(0XB5)	单通道 AD 转换控制寄存器 2
	Bit7:Bit6 SREF 选择单通道 AD 转换参考电压 00：内部参考电压（1.25V） 01：外部参考电压 AIN7 输入 10：模拟电源电压 11 ：外部参考电压 AIN6-AIN7 差分输入
	Bit5: Bit4 设置单通道 AD 转换分辨率 00： 64dec,7 位有效 01： 128dec,9 位有效

	10： 256dec,10 位有效 11： 512dec,12 位有效
	Bit3: Bit0 单通道 AD 转换选择,如果置位时 ADC 正在运行，则在完成 AD 转换后立刻开始，否则置位后立即开始 AD 转换，转换完成后自动清 0. 0000： AIN0 0001: AIN1 0010:AIN2 0011: AIN3 0100： AIN4 0101: AIN5 0110:AIN6 0111: AIN7 1000： AIN0-AIN1 差分 1001: AIN2-AIN3 差分 1010： AIN4-AIN5 差分 1011: AIN6-AIN7 差分 1100 ： GND 1101: 保留 1110 ： 温度传感器 1111 ： 1/3 模拟电源电压
TR0 （0x624B）	Bit0: 置 1 表示将温度传感器与 ADC 连接起来
ATEST(0x61BD)	Bit0: 置 1 表示将温度传感器启用

按照表格寄存器内容，我们对 temperature sensor 和 AD 的寄存器进行配置。具体配置如下：

温度传感器配置：

```
TR0  = 0X01;   //set '1' to connectthe temperature sensorto the
                 SOC_ADC.
ATEST= 0X01; // Enablesthe temperature sensor
```

AD 传感器配置：

```
ADCCON3  = (0x3E);   //选择 1.25V 为参考电压；14 位分辨率；片内采样
ADCCON1  |= 0x30;    //选择 ADC 的启动模式为手动
ADCCON1  |= 0x40;    //启动 AD 转换
```

注意： #include “InitUART_Timer.h” //注意在 option 里设置路径

方法如下：

1、找到例程文件夹，打开 include 文件夹，复制路径。

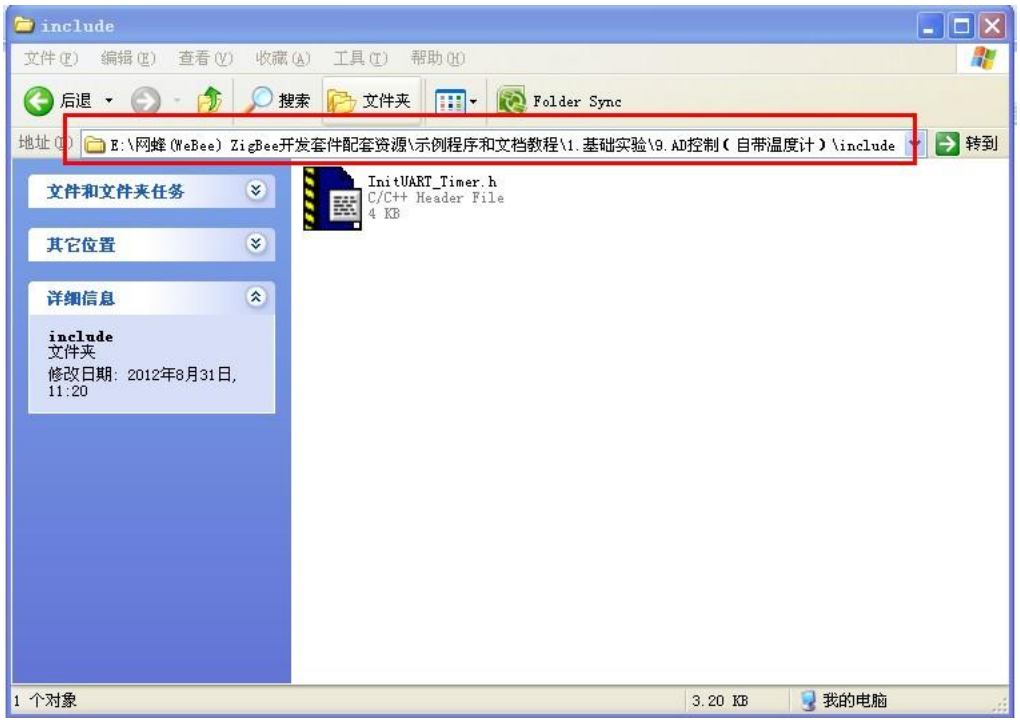


图 2.14

2.粘贴到下面这个位置

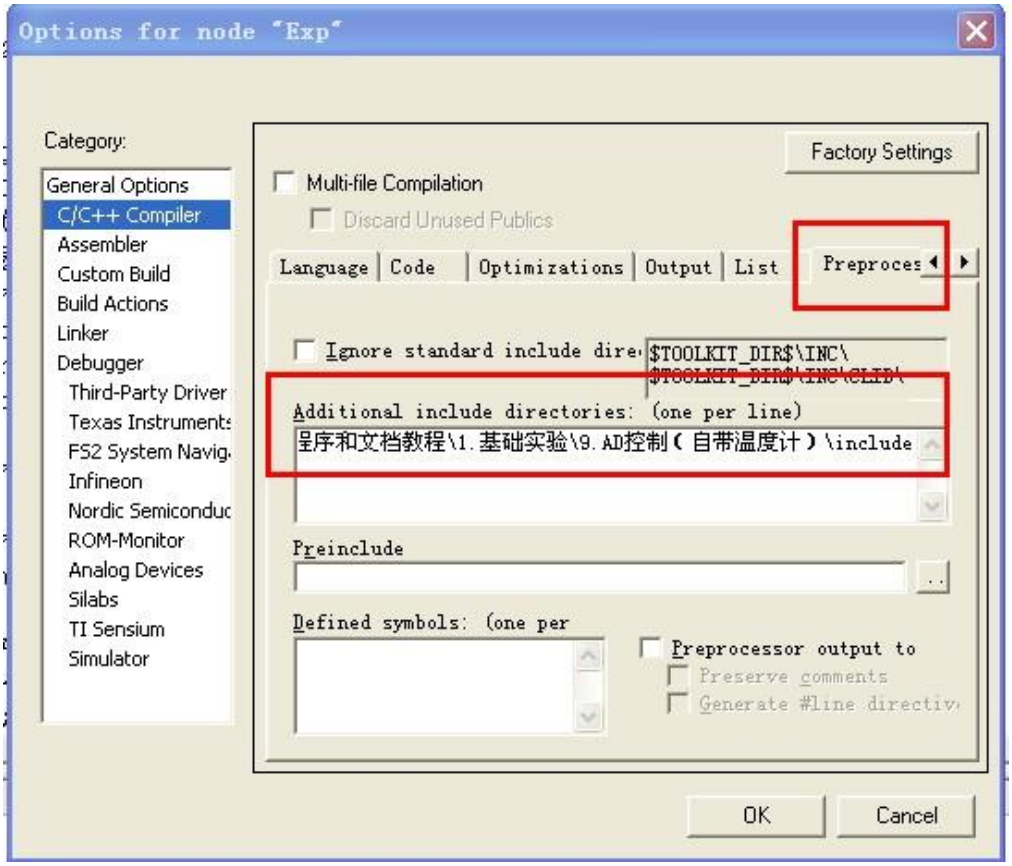


图 2.15

源程序代码（全）

```
/*************************************
程序描述：通过内部 AD 控制把温度信息通过
          串口发送给上位机，部分芯片误差
          较大，需要校准。手摸着芯片，温度
          明显变大。
*************************************/
#include <ioCC2530.h>
#include "InitUART_Timer.h"    //注意在 option 里设置路径
#include "stdio.h"
/*****************************************************************
                    温度传感器初始化函数
*****************************************************************/
void initTempSensor(void)
```

```
{
    DISABLE_ALL_INTERRUPTS();          //关闭所有中断
    InitClock();                       //设置系统主时钟为 32M
    TR0=0X01;          //set '1' to connectthe temperature sensorto the
SOC_ADC.
    ATEST=0X01;        // Enablesthe temperature sensor
}
/*****************************************************************
读取温度传感器 AD 值函数
*****************************************************************/
float getTemperature(void){

    uint   value;
ADCCON3   = (0x3E);                //选择 1.25V 为参考电压；12 位分辨率；
                                     对片内温度传感器采样
    ADCCON1 |= 0x30;                 //选择 ADC 的启动模式为手动
    ADCCON1 |= 0x40;                 //启动 AD 转化
    while(!(ADCCON1 & 0x80));        //等待 AD 转换完成
    value =   ADCL >> 4;             //ADCL 寄存器低 4 位无效
    value |= (((UINT16)ADCH) << 4);
return (value-1367.5)/4.5-4;       //根据 AD 值，计算出实际的温度,芯片、
                                   //手册有错，温度系数应该是 4.5 /℃
                                   //进行温度校正，这里减去 4℃（不同芯
                                     片根据具体情况校正）
}
/*****************************************************************
主函数
*****************************************************************/
void main(void)
```

```
{
        char I;
        char TempValue[6];
        float AvgTemp;
        InitUART0();                                    //初始化串口
        initTempSensor();                               //初始化 ADC
        while(1)
        {
          AvgTemp = 0;
          for(I = 0 ; I < 64 ; i++)
          {
            AvgTemp += getTemperature();
            AvgTemp=AvgTemp/2;                          //每次累加后除 2
          }
          /****温度转换成 ascii 码发送****/
          TempValue[0] = (unsigned char)(AvgTemp)/10 + 48;          //十位
          TempValue[1] = (unsigned char)(AvgTemp)%10 + 48;          //个位
          TempValue[2] = '.';                                       //小数点
          TempValue[3] = (unsigned char)(AvgTemp*10)%10+48;     //十分位
          TempValue[4] = (unsigned char)(AvgTemp*100)%10+48;    //百分位
          TempValue[5] = '\0';                                      //字符串结束符
          UartTX_Send_String( TempValue,6);
          Delayms(2000);          //使用 32M 晶振，故这里 2000 约等于 1S
        }
}
```

实验图片：

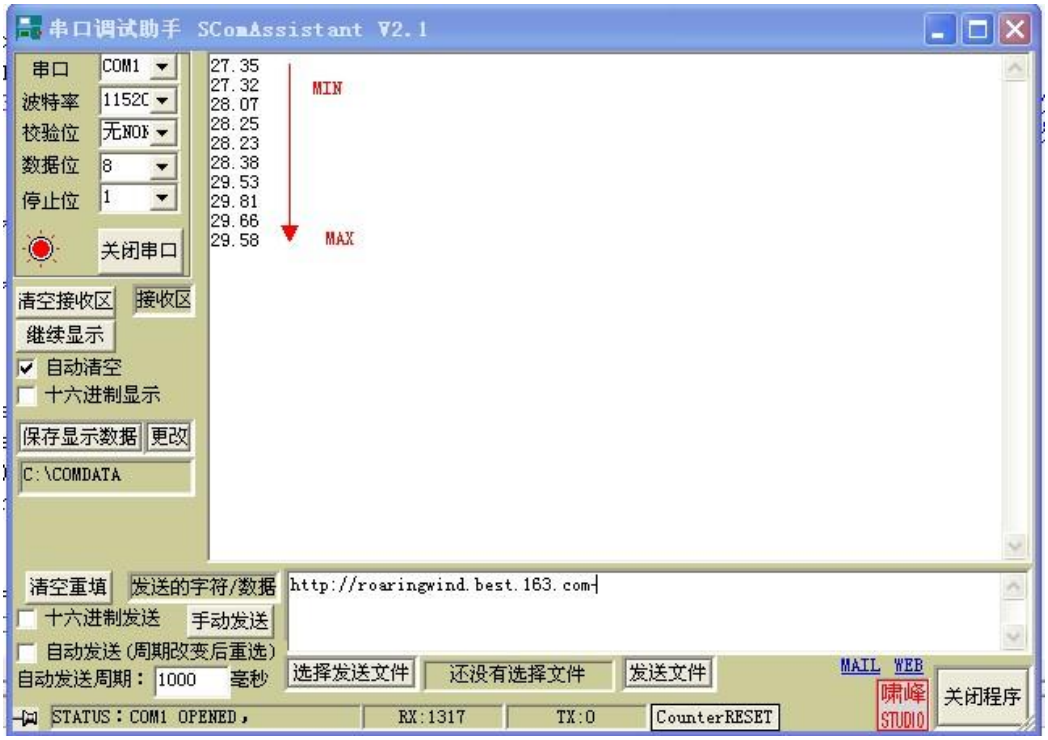


图 2.16 内部温度检测

2.7 睡眠唤醒

前言：

Zigbee 的特点就是远距离低功耗的无线传输设备，节点模块闲时可以进入睡眠模式，在需要传输数据时候进行唤醒，能进一步节省电量。本章将讲述 CC2530 在睡眠模式下的 2 种唤醒方法：外部中断唤醒和定时器唤醒。

实验功能：

将睡眠模式下的 CC2530 通过按键中断和定时器方式唤醒。通过 LED 状态展示。

实验讲解：

睡眠定时器用于设置系统进入和退出低功耗睡眠模式之间的周期。还用于当系统进入低功耗模式后，维持 MAC 定时器（T2）的定时。其特性如下：

长达 24 位定时计数器，运行在 32.768KHZ 的工作频率。24 位的比较器具有中断和 DMA 触发功能在 PM2 低功耗模式下运行

系统电源管理（工作方式如下）：

1. 全功能模式，高频晶振（16M 或者 32M ）和低频晶振（32.768K RCOSC/XOSC ）全部工作，数字处理模块正常工作。
2. PM1 ： 高频晶振（16M 或者 32M ）关闭，低频晶振（32.768K RCOSC/XOSC ）工作，数字核心模块正常工作。
3. PM2 ：低频晶振（32.768K RCOSC/XOSC ）工作，数字核心模块关闭，系统通过 RESET,外部中断或者睡眠计数器溢出唤醒。
4. PM3 ： 晶振全部关闭，数字处理核心模块关闭，系统只能通过 RESET 或外部中断唤醒。此模式下功耗最低。

我们先来看看 WeBee 底板的 LED 部分原理图：如图 2.17 所示。

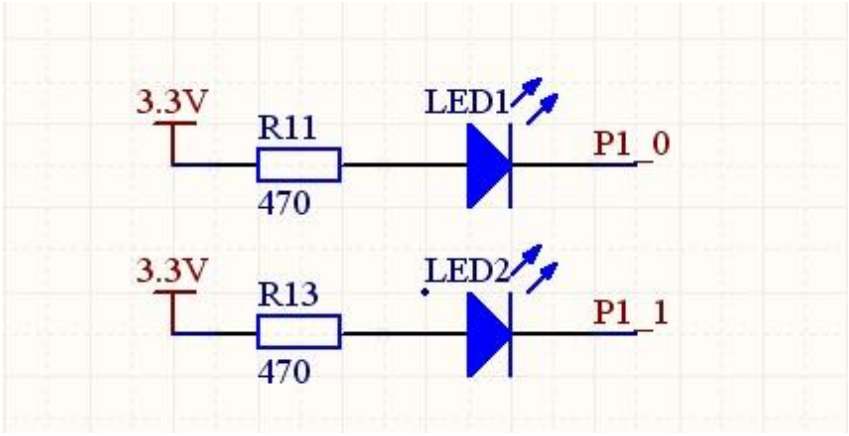


图 2.17 LED 原理图

2.7.1 中断唤醒

CC2530 需要配置的主要寄存器如下：PCON ， SLEEPCMD。功能如下表所示：（详细参考 CC2530 datasheet.pdf）

表 2.8

PCON(0x87)	Bit0. 系统电源模式控制寄存器，置 1 将强制系统进入 SLEEPCMD 所指定的电源模式，所有中断信号都可以清除此置位。
SLEEPCMD(0xBE)	Bit1:Bit0 系统电源模式设定： 00 全功能模式 01：PM1 10:PM2 11:PM3

该寄存器有以下配置方法：

```
SLEEPCMD |= i;       // 设置系统睡眠模式，i=0,1,2,3

PCON = 0x01;         // 进入睡眠模式 ,通过中断打断
PCON = 0x00;         // 系统唤醒 ， 通过中断打断
```

源程序代码（全）

```
/*********************************************
程序描述：LED2 闪烁 5 次后进入睡眠状态，通
          过按下按键 S1 产生外部中断进行唤
          醒，重新进入工作模式。
*********************************************/
#include <ioCC2530.h>

#define uint unsigned int
#define uchar unsigned char

//定义控制 LED 灯和按键的端口
#define LED2 P1_1                    //定义 LED2 为 P11 口控制
#define KEY1 P0_4

//函数声明
void Delayms(uint);              //延时函数
void InitLed(void);              //初始化 P1 口
void SysPowerMode(uchar sel);    //系统工作模式

/*************************
          延时函数
*************************/
void Delayms(uint xms)              //i=xms 即延时 i 毫秒
{
 uint I,j;
 for(i=xms;i>0;i--)
   for(j=587;j>0;j--);
}
```

```
/*****************************
//初始化程序
*****************************/
void InitLed(void)
{
   P1DIR |= 0x02;      //P1_1 定义为输出
   LED2 = 1;           //LED2 灯熄灭
   P0INP  &= ~0X10;    //设置 P0 口输入电路模式为上拉/ 下拉
   P0IEN |= 0X10;      //P01 设置为中断方式
   PICTL |= 0X10;      // 下降沿触发
}
/*****************************************************************
系统工作模式选择函数
* para1  0  1 2 3
* mode  PM0 PM1 PM2 PM3
*****************************************************************/
void SysPowerMode(uchar mode)
{
 uchar I,j;
 I = mode;
 if(mode<4)
 {
SLEEPCMD |= I;              // 设置系统睡眠模式
  for(j=0;j<4;j++);
PCON = 0x01;                // 进入睡眠模式 ,通过中断打断
  }
 else
 {
      PCON = 0x00;              // 系统唤醒 , 通过中断打断
```

```
 }
}
/*************************
        主函数
*************************/
void main(void)
{
uchar count = 0;
   InitLed();           //调用初始化函数
IEN1 |= 0X20;       //开 P0 口总中断
P0IFG |= 0x00;      //清中断标志
 EA = 1;
while(1)
{
       LED2=~LED2;
       if(++count>=10)
       {
          count=0;
          SysPowerMode(3);     //5 次闪烁后进入睡眠状态 PM3,
                               //等待按键 S1 中断唤醒
        }
        Delayms(500);
     }
}
/*****************************************
     中断处理函数-系统唤醒
*****************************************/
#pragma vector = P0INT_VECTOR
  __interrupt void P0_ISR(void)
```

```
 {
      if(P0IFG>0)
{
          P0IFG = 0;        //清标志位
}
          P0IF = 0;
          SysPowerMode(4); //正常工作模式
 }
```

实验 1 图片：

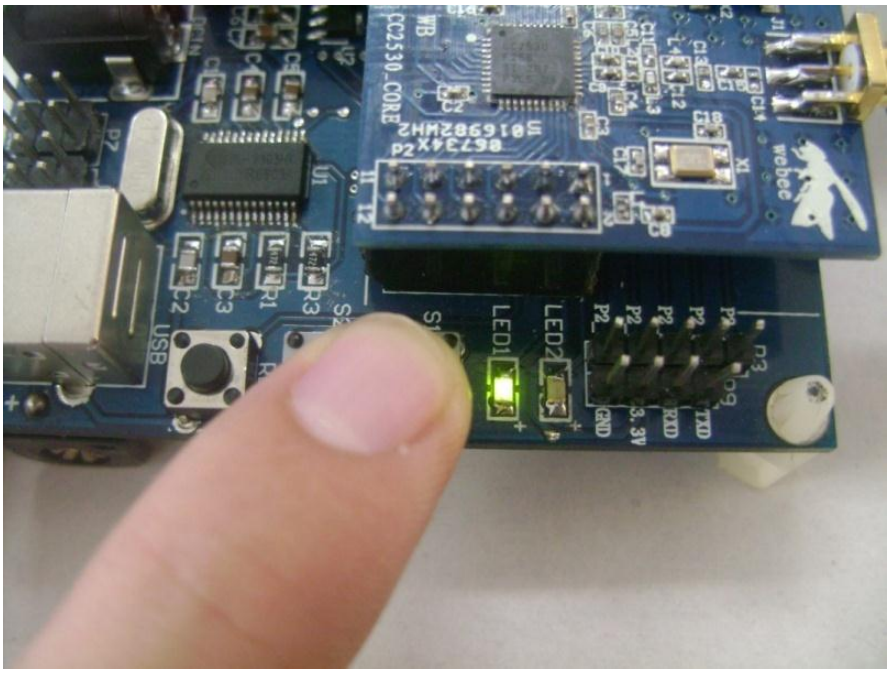


图 2.18 中断唤醒

2.7.2 定时器唤醒

CC2530 睡眠定时器除了实验 1 之外还需要配置的寄存器如下：ST0 ，ST1 ，ST2。也就是文初提及到的 24bit 定时器。如下表所示： （详细参考 CC2530 datasheet.pdf）

表 2.9

ST0(0x95)	睡眠计数器数据 Bit7 ： Bit0
ST1(0x96)	睡眠计数器数据 Bit15： Bit8
ST2(0x97）	睡眠计数器数据 Bit23： Bit16

配置唤醒时间寄存器有以下配置方法：

```
UINT32 sleepTimer = 0;
sleepTimer |= ST0;
sleepTimer |= (UINT32)ST1 <<  8;
sleepTimer |= (UINT32)ST2 << 16;
sleepTimer += ((UINT32)sec * (UINT32)32768)    //低速晶振;
ST2 = (UINT8)(sleepTimer >> 16);
ST1 = (UINT8)(sleepTimer >> 8);
ST0 = (UINT8) sleepTimer;
```

配置完毕后 sleepTimer 与 ST2<<16|ST1<<8|ST0 相差 sec 秒时间（低速晶振）

源程序代码（部分）

```
/*************************************
程序描述：通过设置定时器在特定时间内进行
          唤醒，重新进入工作模式，每次唤
          醒 LED2 闪烁 3 下。
*************************************/
…
…
/*************************************************************
系统工作模式选择函数
* para1  0  1 2 3
* mode  PM0 PM1 PM2 PM3
*************************************************************/
void SysPowerMode(uchar mode)
{
 uchar I,j;
 I = mode;
 if(mode<4)
 {
SLEEPCMD |= I;           // 设置系统睡眠模式
  for(j=0;j<4;j++);
PCON = 0x01;            // 进入睡眠模式 ,通过中断打断
  }
 else
 {
      PCON = 0x00;          // 系统唤醒 ， 通过中断打断
  }
}
```

```
/*****************************************
//初始化 Sleep Timer （设定后经过指定时间自行唤醒）
*****************************************/
void Init_SLEEP_TIMER(void)
{
  ST2 = 0X00;
  ST1 = 0X0F;
  ST0 = 0X0F;
  EA = 1;    //开中断
  STIE = 1;  //SleepTimerinterrupt enable
  STIF = 0;  //SleepTimerinterrupt flag 还没处理的
}

/*****************************************************************
        设置睡眠时间
*****************************************************************/
void Set_ST_Period(uint sec)
{
   UINT32 sleepTimer = 0;
   sleepTimer |= ST0;
   sleepTimer |= (UINT32)ST1 <<  8;
   sleepTimer |= (UINT32)ST2 << 16;
   sleepTimer += ((UINT32)sec * (UINT32)32768);//低频晶振 PM2 模式
   ST2 = (UINT8)(sleepTimer >> 16);
   ST1 = (UINT8)(sleepTimer >> 8);
   ST0 = (UINT8) sleepTimer;
}
```

```
/*************************
//主函数
*************************/
void main(void)
{
uchar I;
   InitLed();                  //调用初始化函数
Init_SLEEP_TIMER();        //初始化 SLEEPTIMER
while(1)
{
      for(i=0;i<6;i++) //闪烁 3 下
      {
         LED2=~LED2;
         Delayms(200);
      }
      Set_ST_Period(3);     //重新进入睡眠模式
      SysPowerMode(2);    //进入 PM2 低频晶振模式
}
}
//睡眠中断唤醒
#pragma vector = ST_VECTOR
 __interrupt void ST_ISR(void)
 {
   STIF = 0;            //清标志位
   SysPowerMode(4);    //进入正常工作模式
 }
```

实验 2 图片:

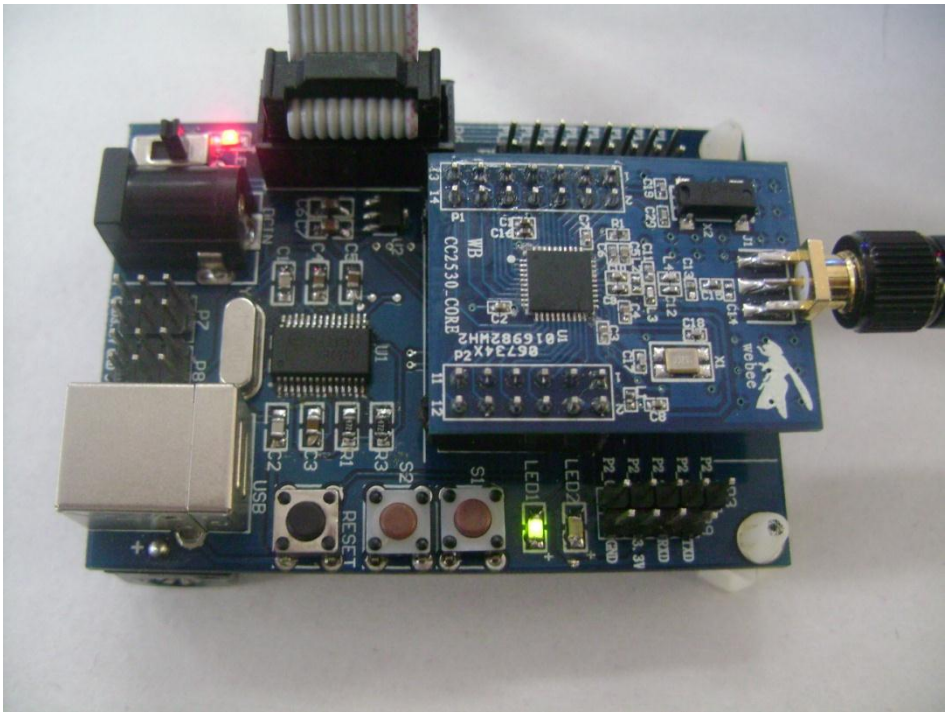


图 2.19 睡眠定时唤醒

2.8 看门狗

前言：

看门狗，眼熟的名字。无论在普通的 51，还是高级的 ARM。都离不开他的身影。一个完整的系统总需要一个看门狗，在你程序跑飞的时候帮你一把，使系统重新进入工作状态。它无疑是世界上最忠诚的狗。不过可千万别忘了喂它。

实验功能：演示打开看门狗后没有喂狗系统不断复位的情况。

实验讲解：CC2530 的看门狗很简单，只需要配置 1 个寄存器 WDCTL。功能如下表所示：（详细参考 CC2530 datasheet.pdf）

表 2.10

WDCTL (0xC9)	Bit7:Bit4 清除计数器值。在看门狗模式下，如果此四位在一个看门狗周期内先后写入 0xA,0x5, 则清除 WDT 的值。简称喂狗。
	Bit3:Bit2 WDT 工作模式选择寄存器。 00 IDLE　　01 IDLE（未使用） 10 看门狗模式　　11 定时器模式
	Bit1:Bit0 看门狗周期选择寄存器。 00 1 秒　　01 0.25 秒 10 15.625 毫秒　　11 1.9 毫秒

按照表格寄存器内容，我们对 WDCTL 具体配置可如下：

```
Init_Watchdog:
  WDCTL  = 0x00;     //这是必须的，打开 IDLE 才能设置看门狗
  WDCTL |= 0x08;    //时间间隔一秒，看门狗模式
```

```
FeedDog：
  WDCTL  = 0xa0;     //按寄存器描述来喂狗
  WDCTL  = 0x50;
```

源程序代码（全）

```
/*************************************/
程序描述：打开看门狗后，得记得喂狗，不然
          系统就会不停地复位了。把喂狗注
          释掉观察 LED1 现象
*************************************/
#include <ioCC2530.h>

#define uint unsigned int
#define uchar unsigned char

//定义控制 LED 灯的端口
#define LED1 P1_0
#define LED2 P1_1        //定义 LED2 为 P11 口控制

//函数声明
void Delayms(uint xms);     //延时函数
void InitLed(void);         //初始化 P1 口

/*****************************
//延时函数
*****************************/
void Delayms(uint xms)      //i=xms 即延时 i 毫秒
{
 uint I,j;
```

```
 for(i=xms;i>0;i--)
    for(j=587;j>0;j--);
}

/*****************************
//初始化程序
*****************************/
void InitLed(void)
{
   P1DIR |= 0x03;  //P1_0、P1_1 定义为输出
   LED1 = 1;        //LED1 灯熄灭
   LED2 = 1;       //LED2 灯熄灭
}

void Init_Watchdog(void)
{
  WDCTL = 0x00;   //这是必须的，打开 IDLE 才能设置看门狗
  WDCTL |= 0x08;    //时间间隔一秒，看门狗模式
}
void FeetDog(void)
{
  WDCTL = 0xa0;
  WDCTL = 0x50;
}
/*****************************
//主函数
*****************************/
void main(void)
{
```

```
    InitLed();                    //调用初始化函数
        Init_Watchdog();
        LED1=1;
    while(1)
    {
        LED2=~LED2;               //仅指示作用。
        Delayms(300);
        LED1=0;
        //通过注释测试，观察 LED1,系统在不停复位。
        FeetDog();    //喂狗，防止程序跑飞
    }
}
```

实验图片：

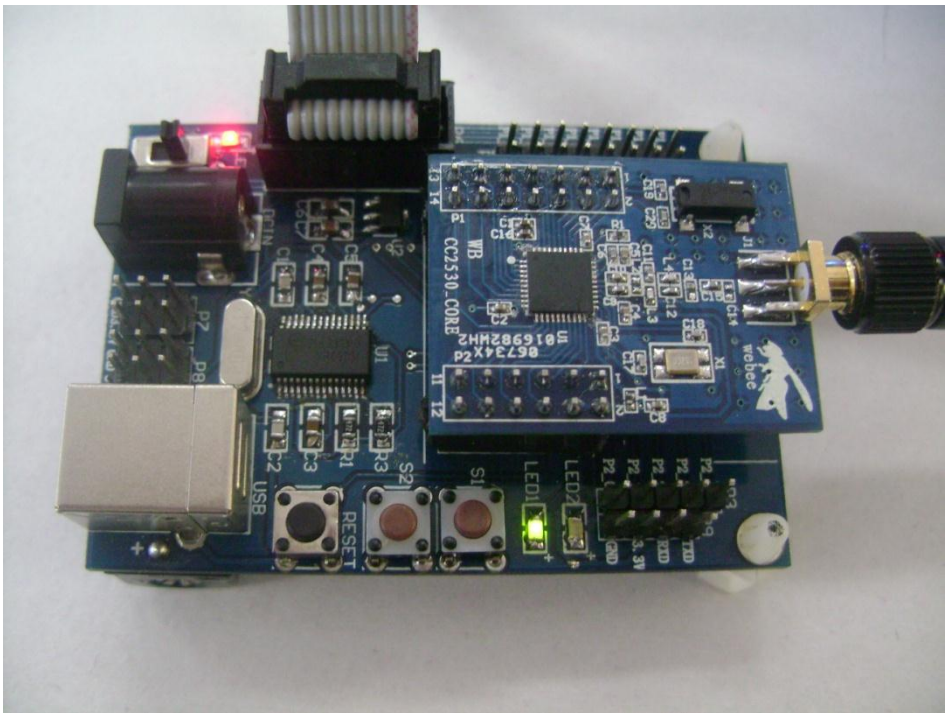


图 2.20

2.9 LCD12864 液晶显示

前言：

相信玩电子的对 LCD12864 都不会陌生，它是我们玩单片常用的东西。今天教大家用 ZigBee CC2530 来驱动我们的 LCD12864 液晶屏。LCD 调试较串口相比可操作性更强，更直观。不失为我们以后调试的方法。大家都知道 2530 的 IO 资源是非常紧缺的，所以我们使用串行方式驱动 LCD12864 液晶屏。

实验功能：使用 LCD12864 显示我们定义的内容。

实验平台：网蜂物联网 ZigBee 开发平台。

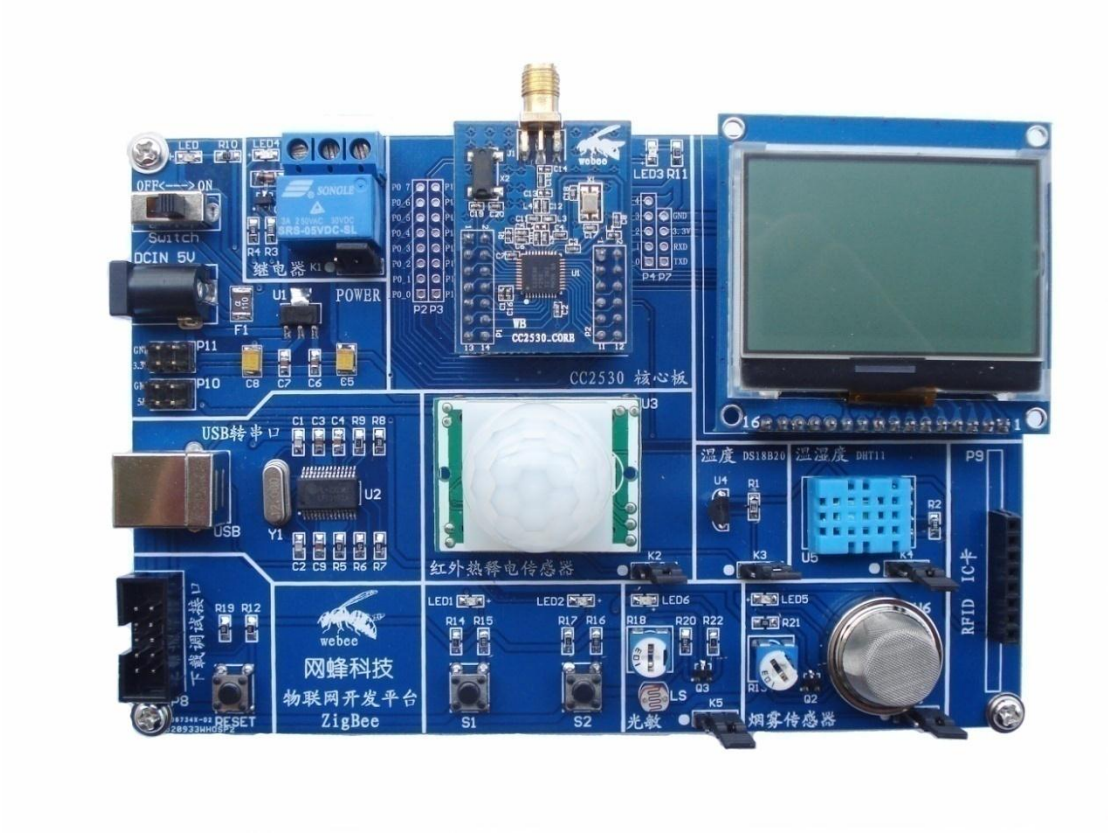


图 2.21 网蜂物联网 ZigBee 开发平台

实验讲解： 我们知道 LCD12864 可以使用串口和并口驱动，但是由于 CC2530 IO 口资源较为紧缺，所以我们优先选择串口的驱动方式。我们使用的 LCD12864 是 16 脚的。接口图如表 1：

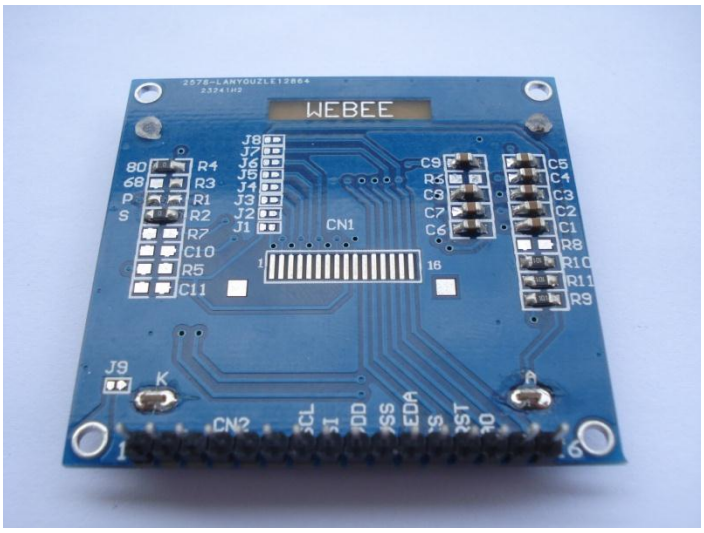


图 2.22 网蜂 LCD12864

引脚号	符 号	电 平	功 能
1	DB_0	H / L	并行模式时：DB0—DB7 串行模式时：DB0—DB5 没有用到 DB6(SCL):串行模式时钟端 DB7(SI):串行模式数据端
2	DB_1	H / L	
3	DB_2	H / L	
4	DB_3	H / L	
5	DB_4	H / L	
6	DB_5	H / L	
7	DB_6(SCL)	H / L	
8	DB_7(SI)	H / L	
9	VDD	5.0V 或 3.3V	模块逻辑电源输入端
10	VSS	0V	逻辑电源地
11	LEDA	5.0V 或 3.3V	LED 背光电源正端
12	/CS	L	芯片选通断，低电平有效
13	/RES	L	复位信号，低电平有效
14	A0	H/L	命令数据选通端，H：数据，L:命令
15	/WR(R/W)	L	80 时序作为写信号，68 时序作为读/写信好
16	/RD(E)	L	80 时序作为读信号，68 时序作为使能信号

图 2.23 LCD12864 引脚图

我们只用到串行驱动需要的引脚，具体如下：

1. VDD ----- +3.3V
2. VSS ----- GND

3. LEDA ----- LCD 背光
4. RES ----- 复位
5. A0 ----- 数据/命令
6. CS ----- 使能
7. SCL ----- 串行模式时钟端
8. SI ----- 串行模式数据端

*其中蓝色是连接 IO 口

下面是一些重要函数的定义，大家可不必深究，但需要了解其用法：

```
//串行发送 IO 口定义
#define L_CS P1_2            //_CS
#define L_LD P0_0         //A0=H data A0=L commend
#define L_CK P1_5            //SCLK
#define L_DA P1_6        //SI
#define L_BK P0_7            //backlight

/******网蜂 WeBee LCD 初始化配置参数******/
void initLCDM(void)
{
   uchar ContrastLevel;     //定义对比度
   ContrastLevel = 0xa0;    //对比度，根据不同的 LCD 调节，否则无法
  显示。
   SendCmd(0xaf);           //开显示
   SendCmd(0x40);           //显示起始行为 0
   SendCmd(0xa0);           //RAM 列地址与列驱动同顺序
   SendCmd(0xa6);           //正向显示
   SendCmd(0xa4);           //显示全亮功能关闭
   SendCmd(0xa2);           //LCD 偏压比 1/9
   SendCmd(0xc8);           //行驱动方向为反向
```

```
    SendCmd(0x2f);              //启用内部 LCD 驱动电源
    SendCmd(0xf8);              //升压电路设置指令代码
    SendCmd(0x00);              //倍压设置为 4X
    SendCmd(ContrastLevel); //设置对比度
}
```

源程序代码（主函数部分）

其中液晶部分建议阅读 “LCD.h”文件。

```
/*************************************/
/*              WeBee 团队                    */
/*          Zigbee 学习例程                   */
/*例程名称：LCD12864 显示                    */
/*建立时间：2012/09/1                        */
/*描述：使用 LCD12864 显示我们定义的信息
*************************************/

#include <ioCC2530.h>
#include “LCD.h”

#define uint unsigned int
#define uchar unsigned char


//函数声明
void Delayms(uint xms);     //延时函数
```

```
/*****************************
        延时函数
*****************************/
void Delayms(uint xms)    //i=xms 即延时 i 毫秒
{
 uint I,j;
 for(i=xms;i>0;i--)
   for(j=587;j>0;j--);
}

/*****************************
        主函数
*****************************/
void main(void)
{
        /*定义显示信息*/
        uchar *mes1 = “WeBee Technology”;
        uchar *mes2 = “ZigBee CC2530F256”;
        uchar *mes3 = “Let’s study ZigBee!”;

        P0DIR = 0XFF;
        P1DIR = 0XFF;
        ResetLCD(); //复位 LCD
        initLCDM(); //初始化 LCD
        ClearRAM(); //请液晶缓存

        delay_us(100);

        /*打印刚刚定义的信息*/
```

```
        Print8(0,0,mes1);
        Print8(0,2,mes2);
        Print8(0,4,mes3);
}
```

实验图片：

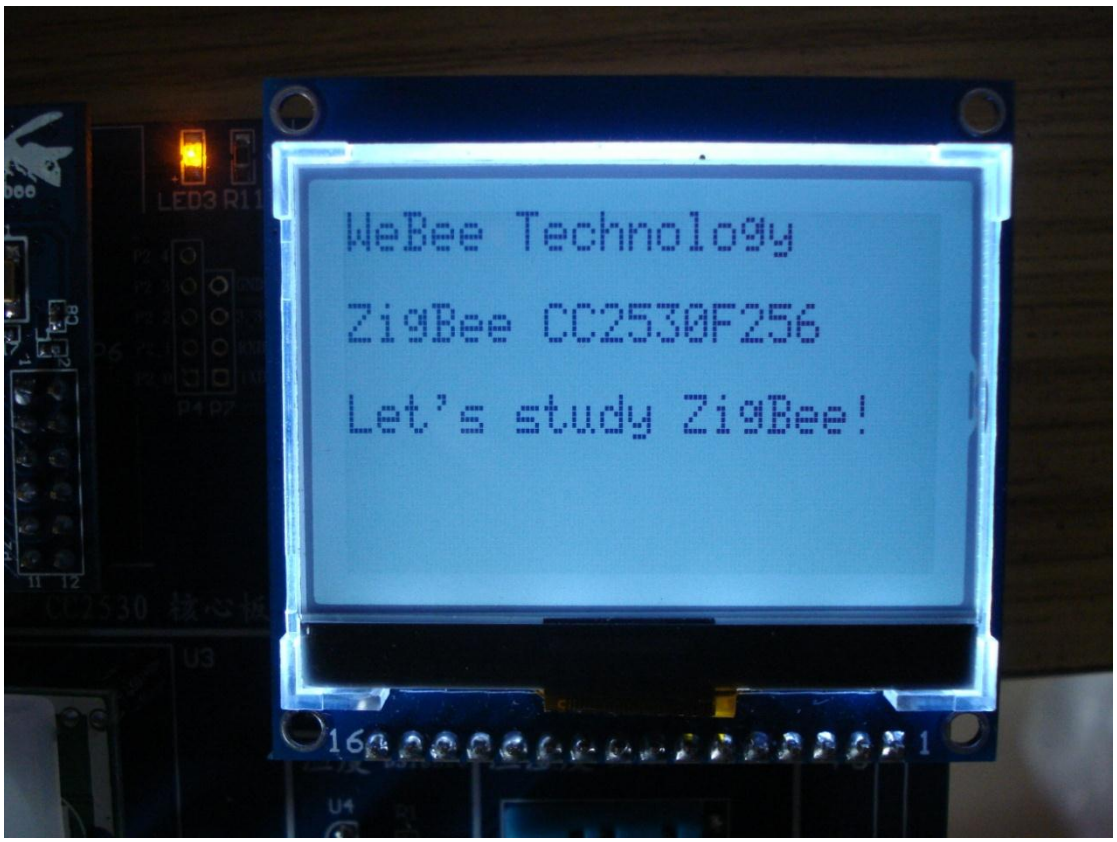


图 2.24 LCD12864 实验现象

第3章 组网演练

3.1 Zigbee 协议栈简介

本节内容仅仅是对 ZigBee 协议栈的一些大家必须理解清楚的概念进行简单的讲解，并没有对 ZigBee 协议栈的构成及工作原理进行详细的讨论。让刚接触 ZigBee 协议栈的朋友们对它有个初步的感性认识，有助于后面使用 ZigBee 协议栈进行真正的项目开发。

什么是 ZigBee 协议栈呢？它和 ZigBee 协议有什么关系呢

协议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发射和接收。协议栈是协议的具体实现形式，通俗点来理解就是协议栈是协议和用户之间的一个接口，开发人员通过使用协议栈来使用这个协议的，进而实现无线数据收发。

图 3.1 展示了 ZigBee 无线网络协议层的架构图。ZigBee 的协议分为两部分，IEEE 802.15.4 定义了 PHY（物理层）和 MAC（介质访问层）技术规范；ZigBee 联盟定义了 NWK（网络层）、APS（应用程序支持子层）、APL（应用层）技术规范。**ZigBee 协议栈就是将各个层定义的协议都集合在一起，以函数的形式实现，并给用户提供 API(应用层)，用户可以直接调用。**

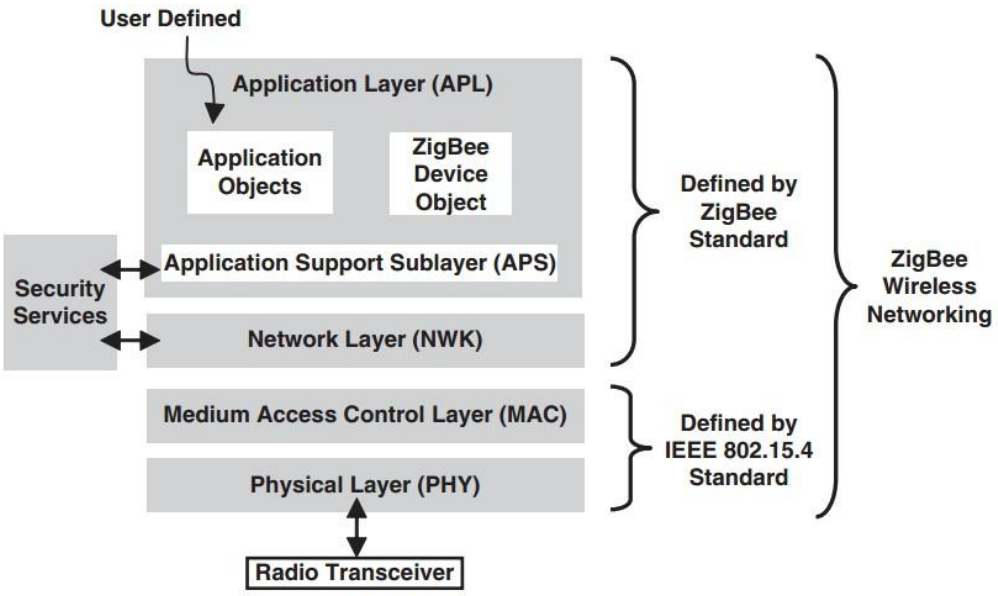


图 3.1 ZigBee 无线网络协议层

在开发一个应用时，协议较底下的层与应用是相互独立的，它们可以从第三方来获得，因此我们需要做的就只是在应用层进行相应的改动。

介绍到这里，大家应该清楚协议和协议栈的关系了吧，是不是会想着怎么样才能用协议栈来开发自己的项目呢？技术总是不断地在发展地，我们可以用 ZigBee 厂商提供的协议栈软件来方便地使用 ZigBee 协议栈（注意：不同厂商提供的协议栈是有区别的，此处介绍 TI 推出的 ZigBee 2007 协议栈也称 Z-Stack）。

Z-stack 是挪威半导体公司 Chipcon(目前已经被 TI 公司收购)推出其 CC2430 开发平台时，推出的一款业界领先的商业级协议栈软件，由于这个协议栈软件的出现，用户可以很容易地开发出具体的应用程序来，也就是大家说的掌握 10 个函数就能使用 ZigBee 通讯的原因。它使用瑞典公司 IAR 开发的 IAR Embedded Workbench for MCS-51 作为它的集成开发环境。Chipcon 公司为自己设计的 Z-Stack 协议栈中提供了一个名为操作系统抽象层 OSAL 的协议栈调度程序。对于用户来说，除了能够看到这个调度程序外，其它任何协议栈操作的具体实现细节都被封装在库代码中。用户在进行具体的应用开发时只能够通过调用 API 接口来进行，而无权知道 ZigBee 协议栈实现的具体细节，也没必要去知道。因此在这里提醒各位开发者，在使用 ZigBee 协议栈进行实际项目开发时，不需要关心协议栈是具体怎么实现的，当然有兴趣的也可以深入分析。

图 3.2 是 TI 公司的基于 ZigBee2007 的协议栈 Z-Stack-CC2530-2.3.0，所有文件目录如红色框所示，我们可以把它看做一个庞大的工程。或者是一个小型的操作系统。采用任务轮询的方法运行。

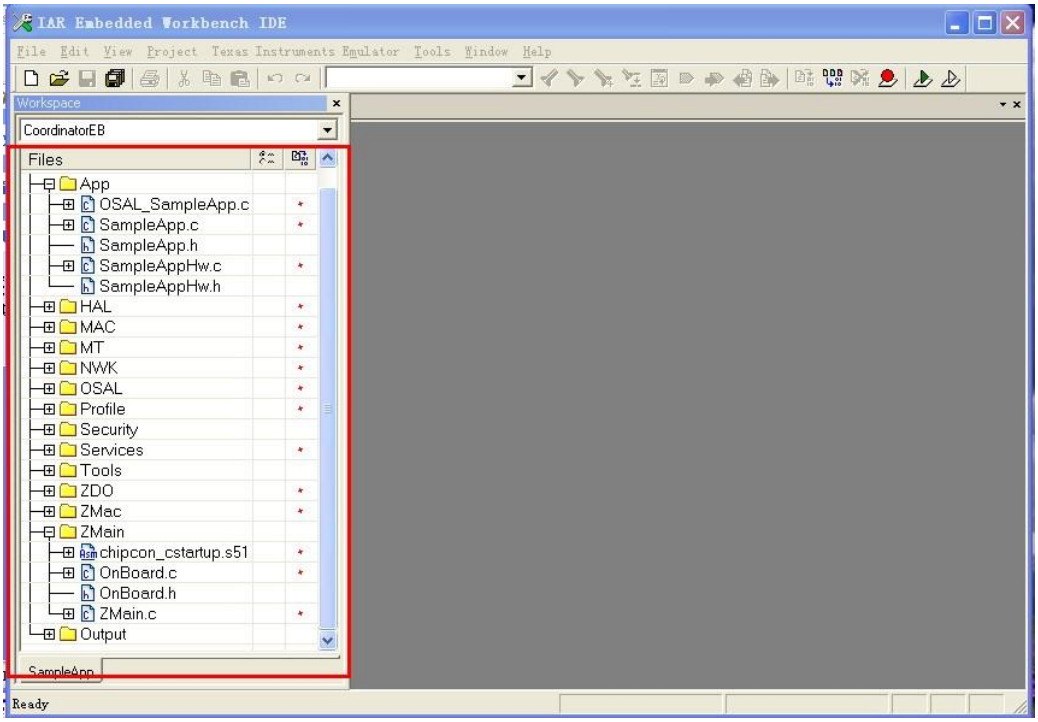


图 3.2 TI Z-stack TI Z-Stack-CC2530-2.3.0

来个小总结：ZigBee 协议栈已经实现了 ZigBee 协议，用户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 ZigBee 协议的具体实现细节，要关心的问题是：应用层的数据是使用哪些函数通过什么方式把数据发送出去或者把数据接收过来的。所以最重要的是我们要学会使用 ZigBee 协议栈。

举个例子，用户实现一个简单的无线数据通信时的一般步骤：

1、组网：调用协议栈的组网函数、加入网络函数，实现网络的建立与节点的加入。

2、发送：发送节点调用协议栈的无线数据发送函数，实现无线数据发送。

3、接收：接收节点调用协议栈的无线数据接收函数，实现无线数据接收。

看起来是不是很简单呢，是不是有动手试试的冲动。具体的例程讲解在这里就不说先了，在接下来的教程里面会详细地和大家一起讨论 ZigBee 协议栈架构中每个层所包含的内容和功能及 Z-stack 的软件架构。

3.2 无线点灯

前言：

万众期待，终于到了“无线”这一块的实验了，无线点灯是大家入门 ZigBee 的一个很好的经典例子，里面虽然还没有用到协议栈，但它体现出来的数据发送、接收和用协议栈是差不多的，而且 TI 公司的 Basic RF 的代码容易看懂，如果把这个实验掌握了（不要只是下载程序然后看试现象），到后面的协议栈就比较好入手了。废话就不多说，立马开始我们的实验。

大家可以了解一下下面的关键字：

CCM - Counter with CBC-MAC (mode of operation)

HAL - Hardware Abstraction Layer (硬件抽象层)

PAN - Personal Area Network （个人局域网）

RF - Radio Frequency （射频）

RSSI - Received Signal Strength Indicator (接收信号强度指示)

实现平台：两块 WeBee 功能底板及两块 WeBee 无线模块

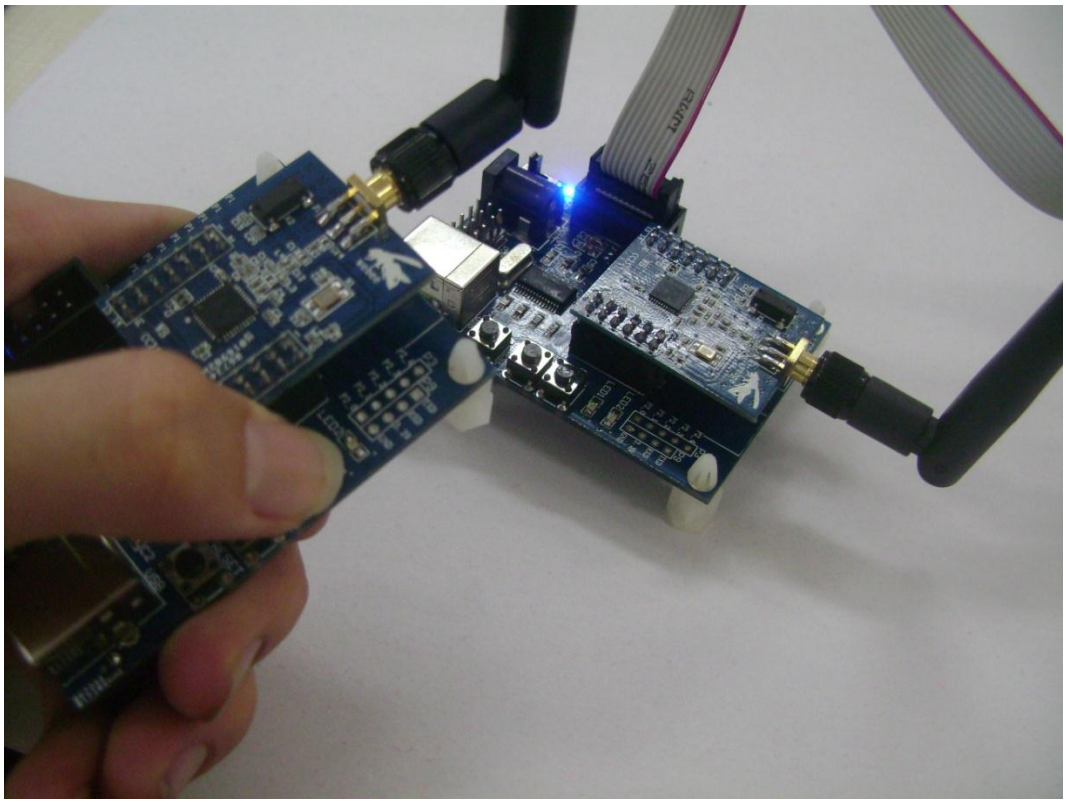


图 3.3 准备就绪

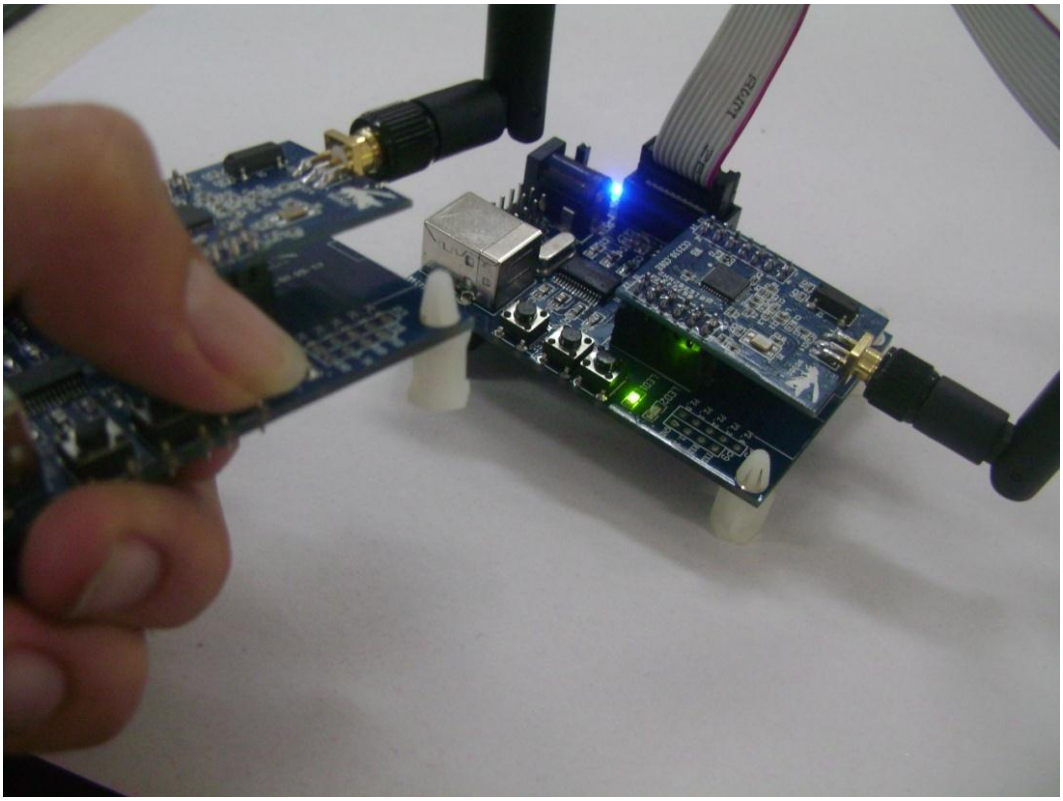


图 3.4 无线点灯

实验现象：两块 WeBee 模块通信，一个模块作发射，另外一个模块接收，发射模块依次按下按键 S1，改变接收模块 LED1 的亮灭的状态。实现无线点灯功能。

实验讲解：

例程的源代码 CC2530 BasicRF.rar 是 TI 官网上下载的，用户可以去 TI 官网注册并下载。首先说明，TI 官网的程序的开发平台是 TI 官网的开发板，硬件资料有所不同，所以要在 WeBee 板上实现无线点灯功能，必须对其代码稍作修改。

本实现讲解的主要内容有分三部分：

1、工程文件介绍

2、Basic RF layer 介绍及其工作过程

3、light_switch.c 代码详解

4.8 工程文件介绍:

解压后打开文件夹：CC2530 BasicRF 然后你会发现还有三个文件夹，然后你下意识地点进 source 文件夹，再进去后会发现，还有两个文件夹，然后你很自然地会再点入 app 文件夹（吐血，还有三个文件夹）……在这茫茫的文件夹里哪个才是我无线点灯的工程呢！

好吧，在讲解实验代码之前，还是先来看看这些看到有点头晕的文件夹吧！

文件夹结构大至如下，仅列出 CC2530 BasicRF 目录一些相关的的文件夹：每个文件夹里面放着什么东西，如果缺少其中某些，我们的灯还是否可以点亮呢？我们来一一探讨：

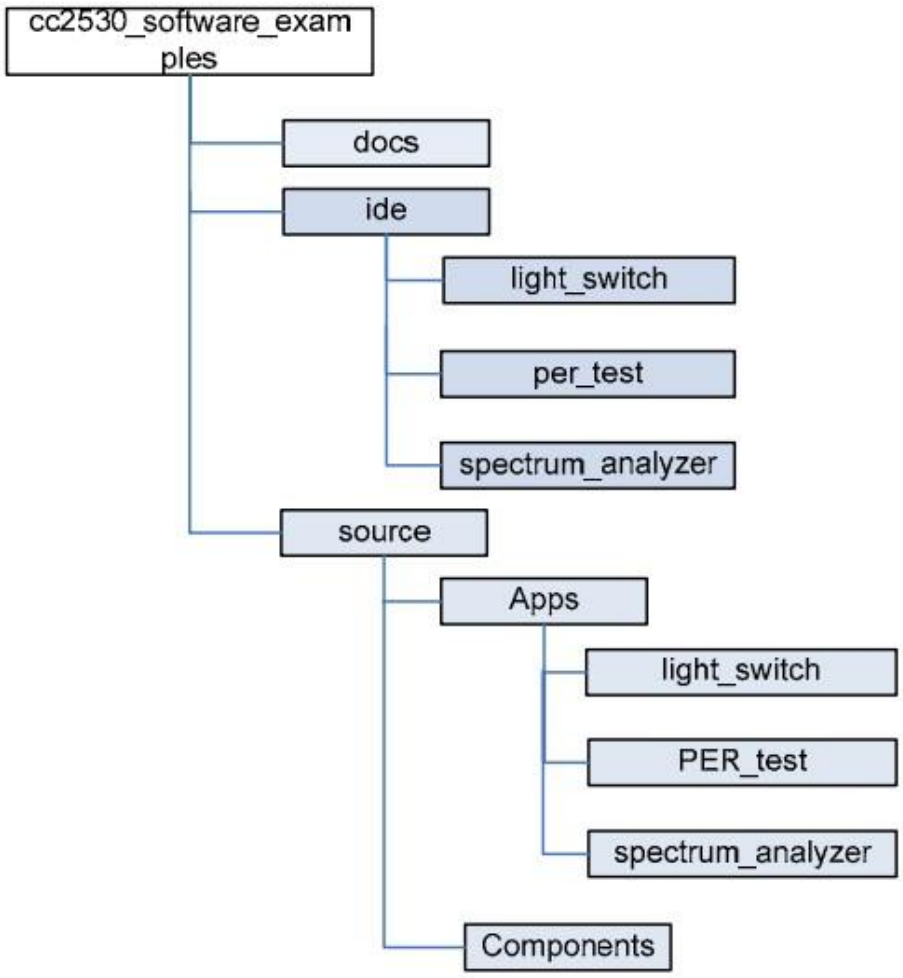


图 3.5 BasicRF 软件文件夹架构

docs 文件夹：

打开文件夹里面仅有一个名为 **CC2530_Software_Examples** 的 PDF 文档，文档的主要内容是介绍 BasicRF 的特点、结构及使用，如果读者有 TI 的开发板的话阅读这个文档就可以做 Basic RF 里面的实验了，从中我们可以知道，里面 Basic RF 包含三个实验例程：**无线点灯、传输质量检测、谱分析应用**。下面讲解的内容中也有部分内容是从这个文档中翻译所得，是一份相当有价值的参考资料。

Ide 文件夹：

打开文件夹后会有三个文件夹，及一个 cc2530_sw_examples.eww 工程，其

中这个工程是上面提及的三个实验例程工程的集合，当然也包含了我们无线点灯的实验工程！在 IAR 环境中打开，在 workspace 看到

Ide\Settings 文件夹：

是在每个基础实验的文件夹里面都会有的，它主要保存有读者自己的 IAR 环境里面的设置。

Ide\srf05_CC2530 文件夹：

里面放有三个工程，light_switch.eww、per_test.eww、spectrum_analyzer.eww 如果读者不习惯几个工程集合在一起看，也可以在这里直接打开你想要用的实验工程。

Source 文件夹：

打开文件夹里面有 apps 文件夹和 components 文件夹

Source\apps 文件夹：

存放 BasicRF 三个实验的应用实现的源代码

Source\components 文件夹：

包含着 BasicRF 的应用程序使用不同组件的源代码

打开实验工程：

打开文件夹 **WeBee CC2530 BasicRF\ide\srf05_cc2530\iar** 路径里面的工程 **light_switch.eww(无线点灯)**。我们的实验就是对它进行修改的。并点击 **application** 的 **light_switch.c** 用户的应用程序就是在里面的了

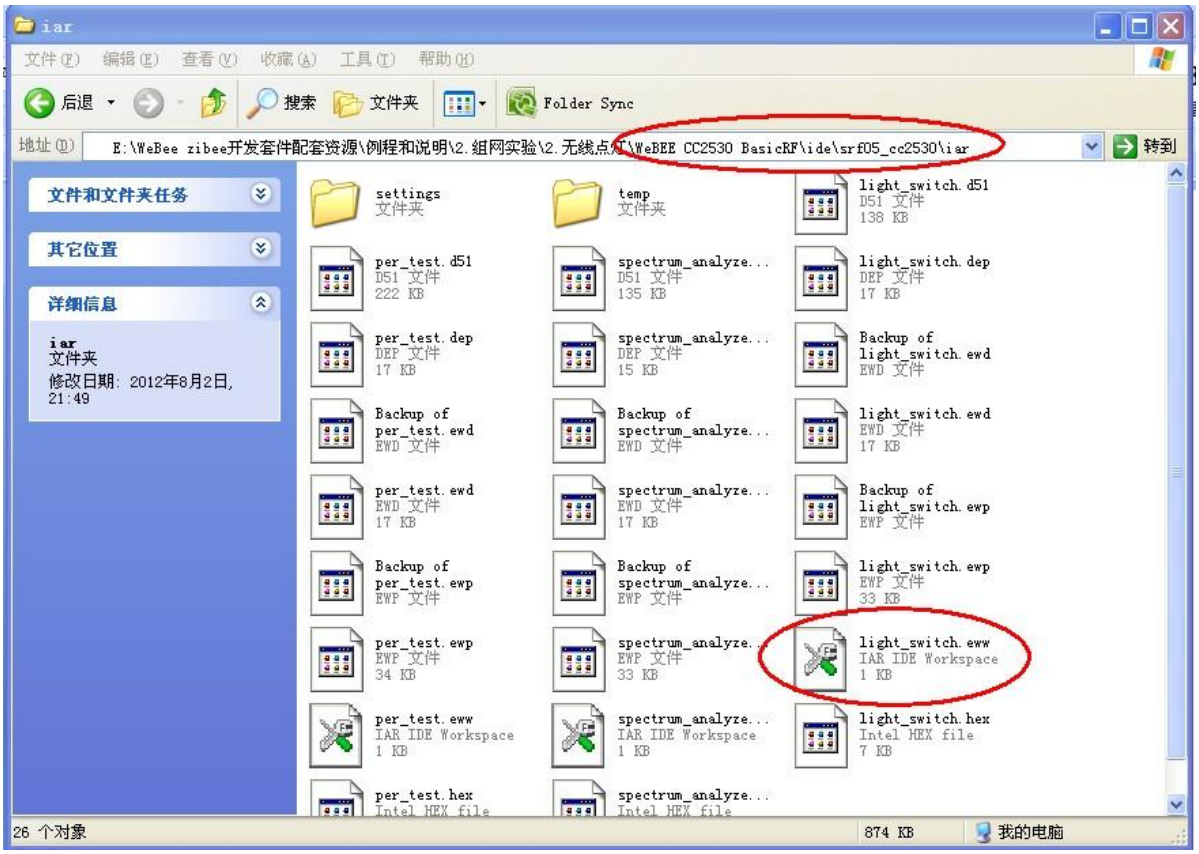


图 3.6 BasicRF 工程路径

二、Basic RF layer 介绍及其工作过程:

在介绍 Basic RF 之前，来看看这个实验例程设计的大体结构，如图所示 Basic RF 例程的软件设计框图就如一座建筑物，

Hardware layer

放在最底，肯定是你实现数据传输的基础了。

Hardware Abstraction layer

它提供了一种接口来访问 TIMER，GPIO，UART，ADC 等。这些接口都通过相应的函数进行实现。

Basic RF layer

为双向无线传输提供一种简单的协议

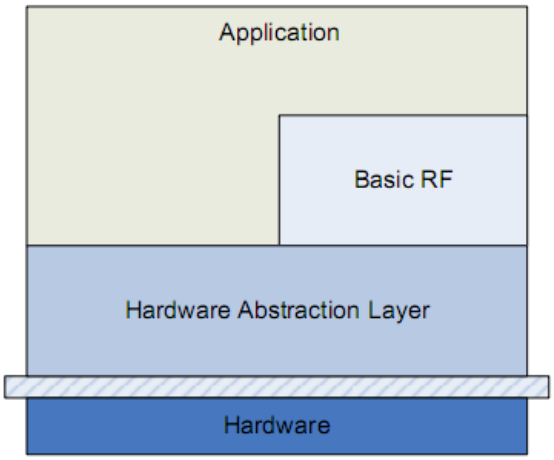


图 3.7

Application layer

是用户应用层，它相当于用户使用 Basic RF 层和 HAL 的接口，也就是说我们通过在 Application layer 就可以使用到封装好的 Basic RF 和 HAL 的函数。

本例程的要求就是读者理解掌握 Basic RF

Basic RF layer 简介:

Basic RF 由 TI 公司提供，它包含了 IEEE 802.15.4 标准的数据包的收发功能但并没有使用到协议栈，它仅仅是是让两个结点进行简单的通信，也就是说 Basic RF 仅仅是包含着 IEEE 802.15.4 标准的一小部分而已。其主要特点有：

1、不会自动加入协议、也不会自动扫描其他节点也没有组网指示灯（LED3）。

2、没有协议栈里面所说的协调器、路由器或者终端的区分，节点的地位都是相等的。

3、没有自动重发的功能。

Basic RF layer 为双向无线通信提供了一个简单的协议，通过这个协议能够进行数据的发送和接收。Basic RF 还提供了安全通信所使用的 CCM-64 身份验证和数据加密，它的安全性读者可以通过在工程文件里面定义 SECURITY_CCM

在 Project->Option 里面就可以选择，本次实验并不是什么高度机密，所以在 SECURITY_CCM 前面带 X 了。

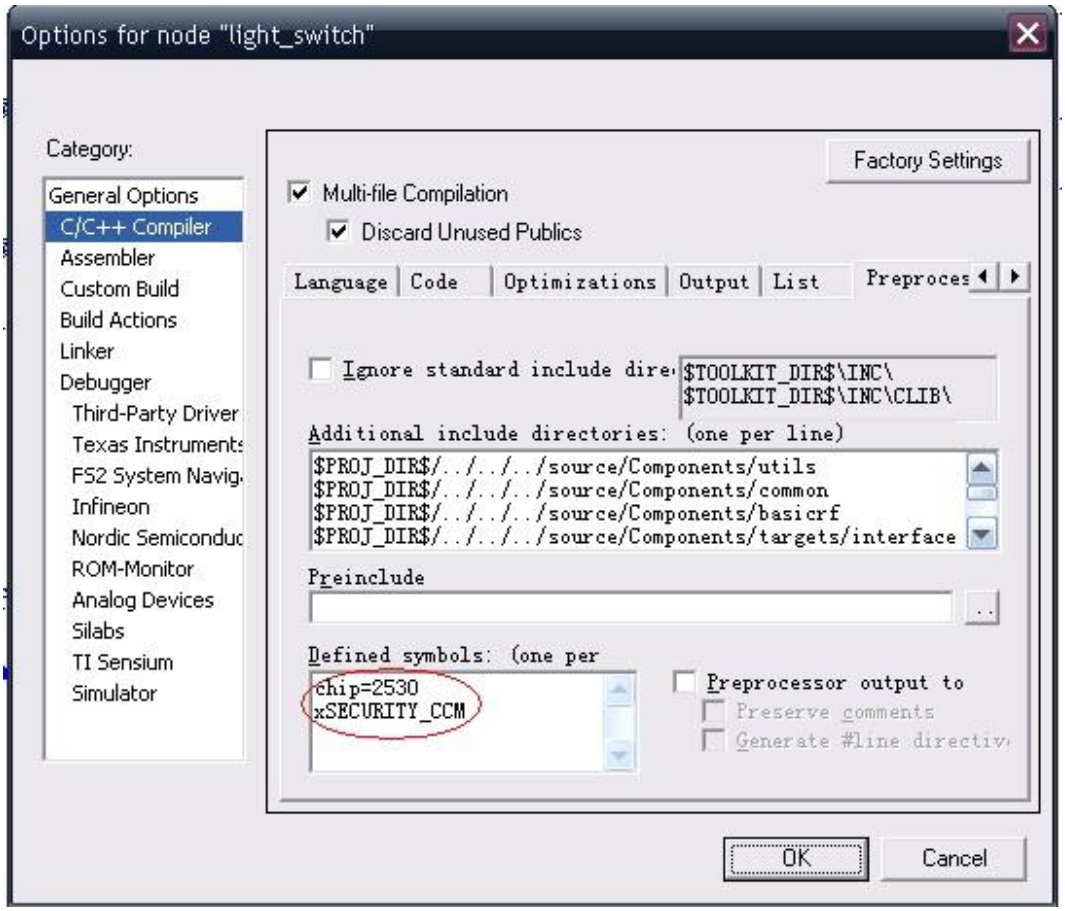


图 3.8 注释 SECURITY_CCM

Basic RF 的工作过程：启动、发射、接收 （请大家按照代码走）

启动

1、确保外围器件没有问题

2、创建一个 **basicRfCfg_t** 的数据结构，并初始化其中的成员，在 **basic_rf.h** 代码中可以找到

```
typedef struct {
uint16 myAddr;          //16 位的短地址（就是节点的地址）
uint16 panId;           //节点的 PAN ID
uint8 channel;          //RF 通道（必须在 11-26 之间）
uint8 ackRequest;       //目标确认就置 true
#ifdef SECURITY_CCM     //是否加密，预定义里取消了加密
uint8* securityKey;
uint8* securityNonce;
```

```
#endif
} basicRfCfg_t;
```

3.调用 **basicRfInit()**函数进行协议的初始化，在 **basic_rf.c** 代码中可以找到

uint8 basicRfInit(basicRfCfg_t* pRfConfig)

函数功能：对 Basic RF 的数据结构初始化，设置模块的传输通道，短地址，PAD ID。

发送

5. 创建一个 **buffer**，把 **payload** 放入其中。Payload 最大为 103 个字节

6. 调用 **basicRfSendPacket()**函数发送，并查看其返回值

在 **basic_rf.c** 中可以找到

uint8 basicRfSendPacket(uint16 destAddr, uint8* pPayload, uint8 length)

destAddr 目的短地址

pPayload 指向发送缓冲区的指针

length 发送数据长度

函数功能：给目的短地址发送指定长度的数据，发送成功刚返回 **SUCCESS**，失败则返回 **FAILED**

接收

1、上层通过 **basicRfPacketIsReady()**函数来检查是否收到一个新数据包

在 **basic_rf.c** 中可以找到

uint8 basicRfPacketIsReady(void)

函数功能：检查模块是否已经可以接收下一个数据，如果准备好刚返回 **TRUE**

2、 调用**basicRfReceive()**函数，把收到的数据复制到**buffer**中。代码可以在 **basic_rf.c** 中可以找到

uint8 basicRfReceive(uint8* pRxData, uint8 len, int16* pRssi)

函数功能：接收来自 **Basic RF** 层的数据包，并为所接收的数据和 **RSSI** 值

配缓冲区

如果能看懂启动、发射、接收就可以说你基本上能使用这个无线模块了。

看到这里大家就会觉得无线传输怎么会那么简单，真的只调用那几个函数就可以实现了吗？是的，使用 Basic RF 实现无线传输只要学会使用这些函数就可以了。

但是具体的实现过程远没有那么简单的，大家可以到….\CC2530 BasicRF\docs 里面查看 CC2530_Software_Examples 中的 5.2.4 Basic RF operation 这个章节的内容，里面详细介绍了 Basic RF 的初始化过程、Basic RF 的发射过程、Basic RF 的接收过程，具体到每个层的功能函数。WeBee 本来想将这部分的内容也详细的和读者们讲解清楚，但后来再仔细考虑还是不放上来了。因为它的具体实现过程大家看文档的那个章节就可以大概明白的了，另一方面，实验例程的模块化编程做得很好，大家只需要明白函数的作用，学会使用它就行了，至于它内部是怎么样一层一层的实现，我们也不用太过关心。

三、light_switch.c 代码详解:

无论你看哪个实验的代码，首先要找的就是 main 函数。从 main 函数开始：（部分已经屏蔽的代码并未贴出，详细的代码请看打开工程）

```
1.  void main(void)
2.  {
3.      uint8 appMode = NONE;                    //不设置模块的模式
4.    // Config basicRF
basicRfConfig.panId = PAN_ID;                  //上面讲的 Basic RF 的启动中的
5.      basicRfConfig.channel = RF_CHANNEL; //初始化 basicRfCfg_t
6.      basicRfConfig.ackRequest = TRUE;          结构体的成员。
7.
8.      #ifdef SECURITY_CCM                    //密钥安全通信，本例程不加密
9.      basicRfConfig.securityKey = key;
```

```
10.  #endif
11.
12.  // Initalise board peripherals       初始化外围设备
13.  halBoardInit();
14.  halJoystickInit();
15.
16.  // Initalise hal_rf    硬件抽象层的 rf 进行初始化
17.  if(halRfInit()= =FAILED)
18.  {
19.     HAL_ASSERT(FALSE);
20.  }
21.  /***********根据 WeBee 学习底板配置**********/
22.  halLedSet(2);                  // 关 LED2(P1_1=1)
23.  halLedClear(1);                // 开 LED1(P1_0=0)
24.
25. /******选择性下载程序，发送模块和接收模块******/
26.  appSwitch();              //节点为按键 S1          P0_4
27.  appLight();               //节点为指示灯 LED1      P1_0
28.  // Role is undefined. This code should not be reached
29.  HAL_ASSERT(FALSE);
30. }
```

第 22～23 行：关闭 WeBee 底板的 LED2，开 LED1。由于 WeBee 设计的 LED 电路是低电平点亮的，与 TI 不同，更符合以前大家学习单片机的习惯，所以 halLedSet()置 1 是使灯熄灭，不过这个没关系，关键是掌握怎么使用就可以了。

第 26～27 行：选择其中的一行，并把另外一行屏蔽掉；这两行重要啦，一个是实现发射按键信息的功能，另一个是接收按键信息并改变 LED 状态的功能。分别为 Basic RF 发射和接收。不同模块在

烧写程序时选择不同功能。

注意：　程序会在 **appSwitch();**　或者 **appLight();**里面循环或者等待，不会执行到第 **29** 行。

接下来看看 **appSwitch()**函数，它是如何实现数据发送的呢？

```
1.  static void appSwitch()
2.  {
3.    #ifdef ASSY_EXP4618_CC2420
4.    halLcdClearLine(1);
5.    halLcdWriteSymbol(HAL_LCD_SYMBOL_TX, 1);
6.    #endif
7.    // Initialize BasicRF
8.    basicRfConfig.myAddr = SWITCH_ADDR;
9.    if(basicRfInit(&basicRfConfig)==FAILED){
10.   HAL_ASSERT(FALSE);
11.   }
12.   pTxData[0] = LIGHT_TOGGLE_CMD;
13.   // Keep Receiver off when not needed to save power
14.   basicRfReceiveOff();
15.   // Main loop
16.   while (TRUE)   //程序进入死循环
17.   {
18.     if(halButtonPushed()==HAL_BUTTON_1)      //按键 S1 被按下
19.     {
20.     basicRfSendPacket(LIGHT_ADDR,pTxData,APP_PAYLOAD_LENGTH);

21.     // Put MCU to sleep. It will wake up on joystick interrupt
```

```
22.     halIntOff();
23.     halMcuSetLowPowerMode(HAL_MCU_LPM_3); // Will turn on
global
24.     // interrupt enable
25.     halIntOn();
26.     }
27.   }
28. }
```

第 3～6 行： TI 学习板上的液晶模块的定义，我们不用管他

第 8～11 行： Basic RF 启动中的初始化，就是上面所讲的 Basic RF 启动的第 3 步

第 12 行： Basic RF 发射第 1 步，把要发射的数据或者命令放入一个数据 buffer，此处把灯状态改变的命令 LIGHT_TOGGLE_CMD 放到 pTxData 中。

第 14 行： 由于模块只需要发射，所以把接收屏蔽掉以降低功耗。

第 18 行： if(halButtonPushed()==HAL_BUTTON_1)判断是否 S1 按下，函数 halButtonPushed()是 halButton.c 里面的，它的功能是：按键 S1 有被按动时，就回返回 true，则进入 basicRfSendPacket(LIGHT_ADDR, pTxData, APP_PAYLOAD_LENGTH);

第 20 行： Basic RF 发射第 2 步，**也是发送数据最关键的一步，函数功能在前面已经讲述。**

basicRfSendPacket(LIGHT_ADDR, pTxData, APP_PAYLOAD_LENGTH)就是说：将 LIGHT_ADDR、pTxData、APP_PAYLOAD_LENGTH 的实参写出来就是 basicRfSendPacket(0xBEEF ,pTxData[0] ,1)把字节长度为 1 的命令，发送到地址 0xBEEF

第 22～23 行： WeBee 开发板暂时还没有 joystick（多方向按键），不用理它先。

第 25 行： 使能中断

发送的 **appSwitch()**讲解完毕，接下来就到我们的接收 **appLight()**函数了

```
1.  static void appLight()
```

```
2.  {
3.   /***********************************************
4.     halLcdWriteLine(HAL_LCD_LINE_1, “Light”);
5.     halLcdWriteLine(HAL_LCD_LINE_2, “Ready”);
6.   ***********************************************/
7.    #ifdef ASSY_EXP4618_CC2420
8.    halLcdClearLine(1);
9.    halLcdWriteSymbol(HAL_LCD_SYMBOL_RX, 1);
10.   #endif

11.   // Initialize BasicRF

12.   basicRfConfig.myAddr = LIGHT_ADDR;
13.   if(basicRfInit(&basicRfConfig)==FAILED) {
14.   HAL_ASSERT(FALSE);
15.   }
16.   basicRfReceiveOn();

17.   // Main loop

18.   while (TRUE)
19.    {
20.      while(!basicRfPacketIsReady());
21.      if(basicRfReceive(pRxData, APP_PAYLOAD_LENGTH, NULL)>0) {
22.      if(pRxData[0] == LIGHT_TOGGLE_CMD)
23.      {
24.        halLedToggle(1);
25.        }
26.     }
```

```
27.  }
28. }
```

第 7～10 行：LCD 内容暂时不用理它

第 12～15 行：Basic RF 启动中的初始化，上面 Basic RF 启动的第 3 步

第 16 行：函数 basicRfReceiveOn()，开启无线接收功能，调用这个函数后模块一直会接收，除非再调用 basicRfReceiveOff()使它关闭接收。

第 18 行：程序开始进行不断扫描的循环

第 19 行：**Basic RF 接收的第 1 步**，while(!basicRfPacketIsReady()) 检查是否接收上层数据，

第 20 行：**Basic RF 接收的第 2 步**，if(basicRfReceive(pRxData, APP_PAYLOAD_LENGTH, NULL)>0)判断否接收到有数据

第 21 行：if(pRxData[0] == LIGHT_TOGGLE_CMD)判断接收到的数据是否就是发送函数里面的 LIGHT_TOGGLE_CMD 如果是，执行第 22 行

第 22 行：halLedToggle(1)，改变 Led1 的状态。

实验操作：

第一步：打开….\CC2530 BasicRF\ide 文件夹下面的工程 在 light_switch.c 里面找到 main 函数，找到下面内容，把 appLight(); 注释掉，下载到发射模块。

```
/************Select one and shield to another************/
  appSwitch();          //节点为按键 S1          P0_4
  // appLight();        //节点为指示灯 LED1      P1_0
```

第二步：找到相同位置，这次把 appSwitch();注释掉，下载到接收模块。

```
/************Select one and shield to another***********by boo*/
//appSwitch();          //节点为按键 S1          P0_4
  appLight();           //节点为指示灯 LED1      P1_0
```

完成烧写后上电，按下发射模块的 S1 按键，可以看到接收模块的 LED1 被点亮。

拓展：或许你觉得点亮 LED 太没意思了，我们来点台灯。

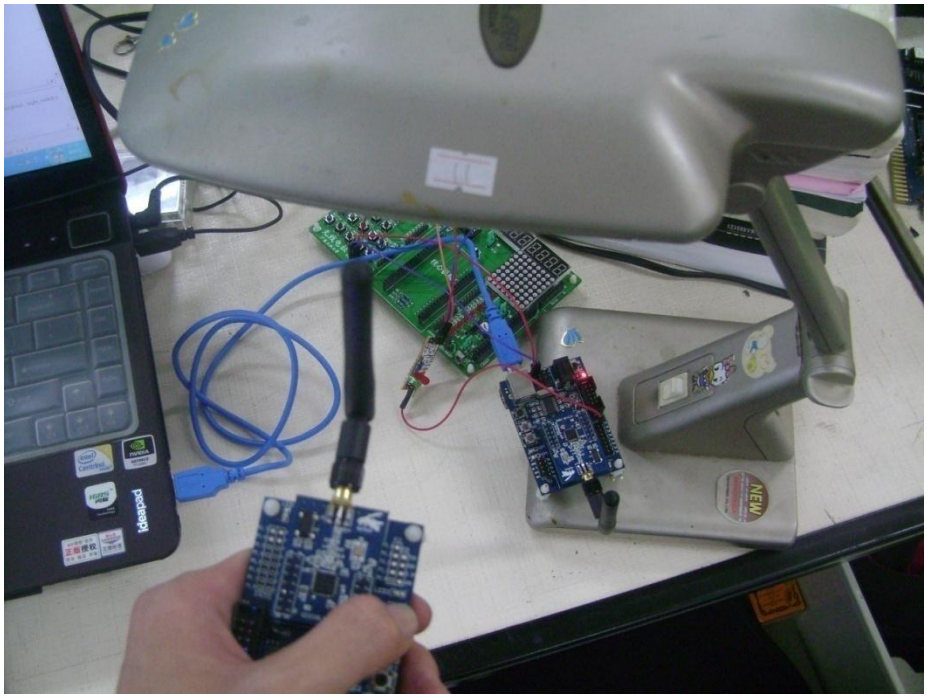


图 3.9 准备就绪

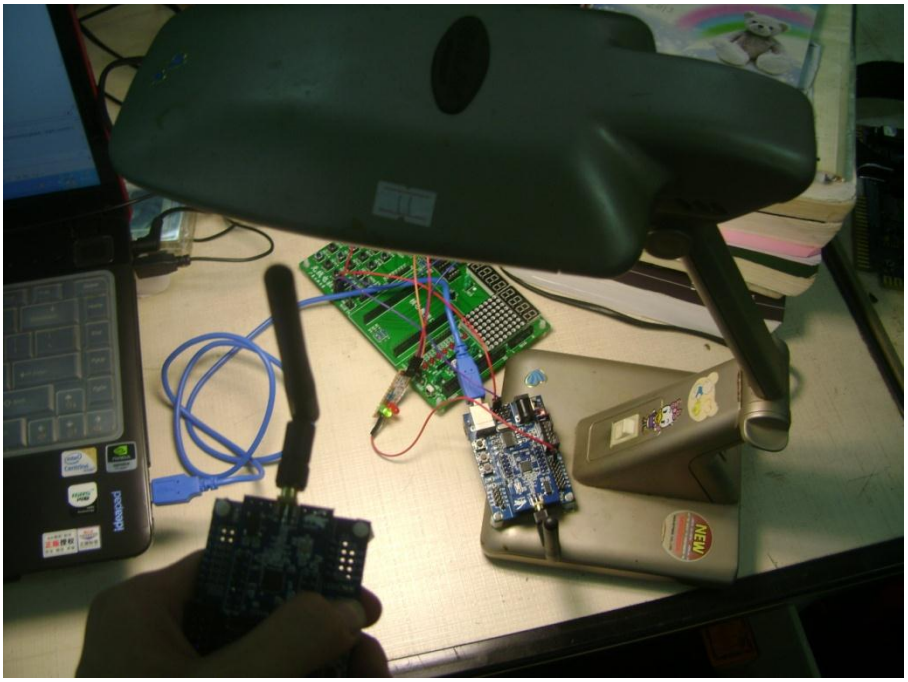


图 3.10 点亮台灯

3.3 信号传输质量检测

前言： PER（误包率检测）实验是 BasicRF 的第二个实验，和无线点灯一样是没有使用协议栈的点对点通讯。通过无线点灯大家应该对 zigbee 的发射和接收有个感性的认识，本次实验讲解不会像无线点灯一样讲得那么详细，因为接收发射的过程基本上是一样的，但也希望初学者能自己认真学习一下这个实验，相信会对无线传输会有一个更清晰的认识。

实现平台：两块 WeBee base_board 及两块 WeBee 无线模块

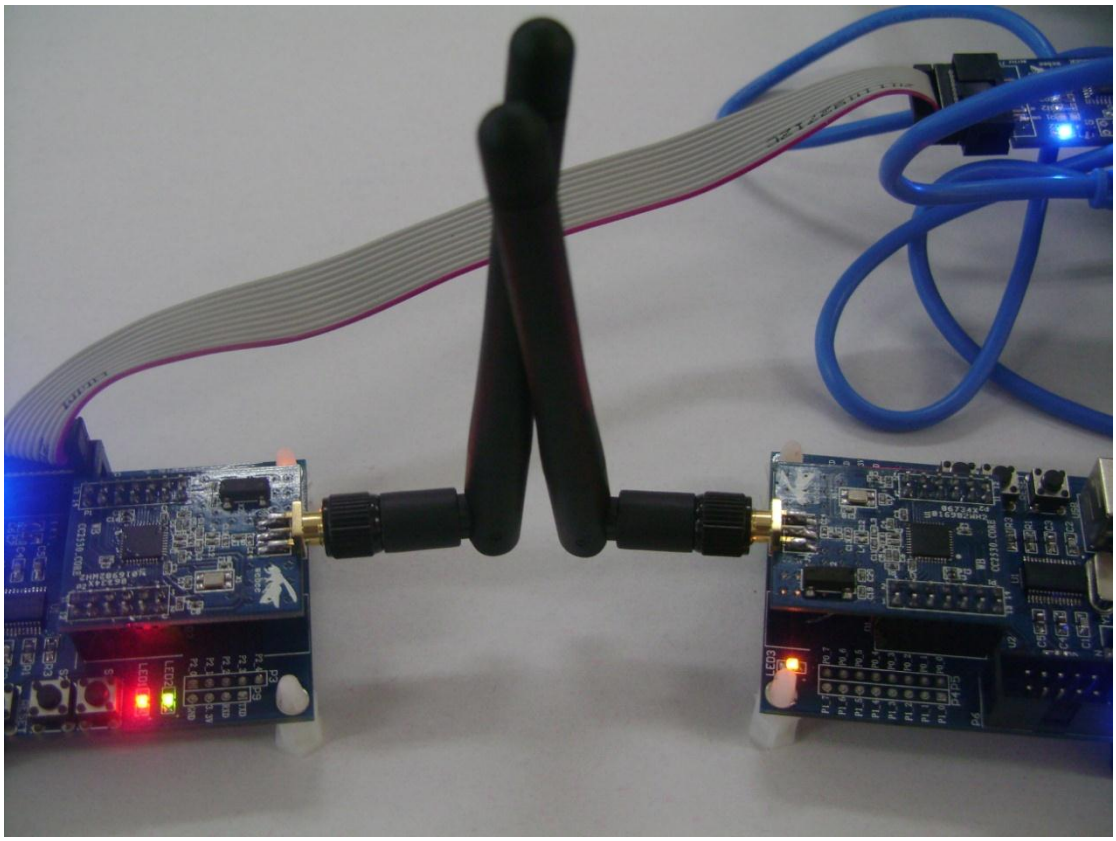


图 3.11 网蜂 ZigBee 开发平台

实验现象：两块 WeBee 模块通信，一个模块作发射，另外一个模块接收，接收模块通过串口不在 PC 机上显示当前的误包率、RSSI 值和接收到数据包的个数。

实验详解：例程的源代码 CC2530 BasicRF.rar 是 TI 官网上下载的，打开\CC2530

BasicRF\ide\srf05_cc2530\iar 里面的 per_test.eww 工程，由于我们的硬件平台不同于 TI 的开发板，所以我们需要在 per_test 中加入串口发送函数，才能在串口调试助手上看到我们的实验现象。

打开工程，在 application 文件夹点 per_test.c 我们的主要功能函数都在这里

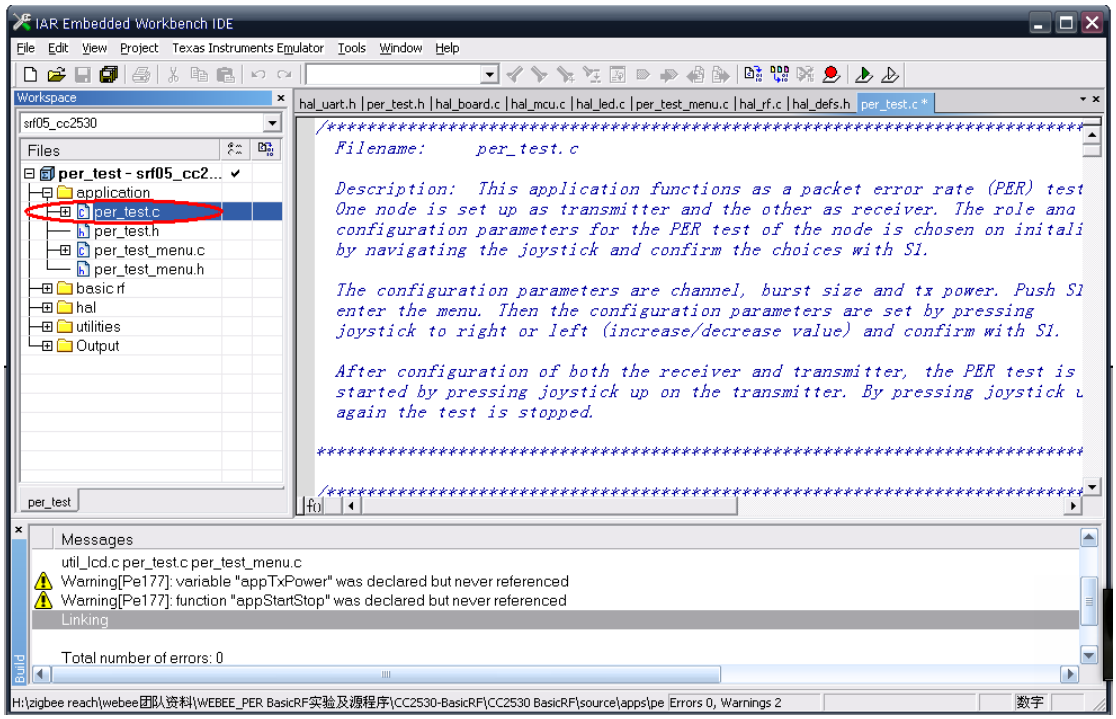


图 3.12

在这个.c 文件中添加串口发送函数

INCLUDES 中添加：#include “string.h”

```
  After configuration of both the receiver and transmitter, the PER test is
  started by pressing joystick up on the transmitter. By pressing joystick up
  again the test is stopped.

**************************************************************************************/

/**************************************************************************************
* INCLUDES
*/
#include "hal_lcd.h"
#include "hal_led.h"
#include "hal_int.h"
#include "hal_timer_32k.h"
#include "hal_joystick.h"
#include "hal_button.h"
#include "hal_board.h"
#include "hal_rf.h"
#include "hal_assert.h"
#include "util_lcd.h"
#include "basic_rf.h"
#include "per_test.h"
#include "string.h"

/**************************************************************************************
* CONSTANTS
*/
// Application states
#define IDLE                    0
#define TRANSMIT_PACKET         1


/*****************************important select or shelt**********************************/
//#define MODE_SEND               //屏蔽时：appReceiver
                                //不屏蔽时：appTransmitter
/**************************************************************************************/


/**************************************************************************************
* LOCAL VARIABLES
```

图 3.13

然后继续添加串口初始化和发送函数：

```
*/
static void appTimerISR(void);
static void appStartStop(void);
static void appTransmitter();
static void appReceiver();
void initUART(void);//*************************
void UartTX_Send_String(int8 *Data,int len);//*********************


/*************************************************************
串口初始化函数
*************************************************************/
void initUART(void)
{
    PERCFG = 0x00;                      //位置1 P0口
    P0SEL = 0x0c;                       //P0_2,P0_3用作串口（外部设备功能）
    P2DIR &= ~0XC0;                     //P0优先作为UART0

    U0CSR |= 0x80;                      //设置为UART方式
    U0GCR |= 11;
    U0BAUD |= 216;                      //波特率设为115200
    UTX0IF = 0;                         //UART0 TX中断标志初始置位0
}
/*************************************************************
串口发送字符串函数
*************************************************************/
void UartTX_Send_String(int8 *Data,int len)
{
  int j;
  for(j=0;j<len;j++)
  {
    U0DBUF = *Data++;
    while(UTX0IF == 0);
    UTX0IF = 0;
  }
}


/**************************************************************************************
```

图 3.14

很简单地就把串口搞定了，因为只有接收模块才会用到串口，所以串口的初始化只需要放在 appReceiver ( )函数中

下面分析整个工程的发送和接收过程

首先还是要先找到 main.c

```
/*************************************************************************************
* @fn          main
*
* @brief       This is the main entry of the "PER test" application.
*
* @param       basicRfConfig – file scope variable. Basic RF configuration data
*              appState – file scope variable. Holds application state
*              appStarted – file scope variable. Used to control start and stop of
*              transmitter application.
*
* @return      none
*/
1. void main (void)
2. {
3. uint8 appMode;
4. appState = IDLE;
5. appStarted = TRUE;
6. // Config basicRF     配置 Basic RF
7. basicRfConfig.panId = PAN_ID;
8. basicRfConfig.ackRequest = FALSE;

9. // Initialise board peripherals    初始化外围硬件

10. halBoardInit();

11. // Initalise hal_rf     初始化 hal_rf
```

```
12. if(halRfInit()==FAILED) {
13. HAL_ASSERT(FALSE);
14. }

15. // Indicate that device is powered

16. halLedSet(1);
17. // Print Logo and splash screen on LCD
18. utilPrintLogo("PER Tester");

19. // Wait for user to press S1 to enter menu

20. halMcuWaitMs(350);
21. // Set channel
22. //设置信道，规范要求信道只能为为 11——25。这里选择信道 11
23. basicRfConfig.channel = 0x0B;

//设置模块的模式，一个作为发射，另一个为接收，看是否 define MODE_SEND
24. #ifdef MODE_SEND
25. appMode = MODE_TX;
26. #else
27. appMode = MODE_RX;
28. #endif
29. // Transmitter application
30. if(appMode == MODE_TX) {
// No return from here      如果 define MODE_SEND 则进入 appTransmitter();
  发射模式
31. appTransmitter();
```

```
32. }
33. // Receiver application
34. else if(appMode == MODE_RX) {
35. // No return from here  如果没有 define MODE_SEND 则进入 appReceiver ();接收模式
36. appReceiver();
37. }

38. // Role is undefined. This code should not be reached

39. HAL_ASSERT(FALSE);
40. }
```

大家看注释也应该知道 main.c 做了哪些事情：

1、 一大堆的初始化（都是必须的）

2、 设置信道，发射和接收模块的信道必须一致

3、 选择为发射或者接收模式

发射函数 define MODE_SEND 则进入 appTransmitter();

```
/*************************************************************
* @fn              appTransmitter
*
* @brief           Application code for the transmitter mode. Puts MCU in endless
*                  loop
*
* @param           basicRfConfig – file scope variable. Basic RF configuration data
```

```
*                   txPacket – file scope variable of type perTestPacket_t
*                   appState – file scope variable. Holds application state
*                   appStarted – file scope variable. Controls start and stop of
*                                transmission
*
* @return          none
*/
1.  static void appTransmitter()
2.  {
3.  uint32 burstSize=0;
4.  uint32 pktsSent=0;
5.  uint8 appTxPower;
6.  uint8 n;
7.  // Initialize BasicRF                    /*  初始化 Basic RF */
8.  basicRfConfig.myAddr = TX_ADDR;
9.  if(basicRfInit(&basicRfConfig)==FAILED)
10. {
11. HAL_ASSERT(FALSE);
12. }

13. // Set TX output power                              /*  设置输出功率  */

14. halRfSetTxPower(2);                  //HAL_RF_TXPOWER_4_DBM

15. // Set burst size              /*  设置进行一次测试所发送的数据包数
量  */

16. burstSize = 1000;
```

```
17. // Basic RF puts on receiver before transmission of packet, and turns off
after packet is sent

18. basicRfReceiveOff();

19. /*************************************************************
***/

20. Config timer and IO     配置定时器和 IO
21. ************************************
************************/
22. appConfigTimer(0xC8);

23. // Initalise packet payload                    /* 初始化数据包载荷 */

24. txPacket.seqNumber = 0;
25. for(n = 0; n < sizeof(txPacket.padding); n++)
26. {
27. txPacket.padding[n] = n;
28. }
//************************进入循环*******************//
29. while (TRUE)
30. {
31. while(appStarted)
32. {
33. if (pktsSent < burstSize)
34. {
35. // Make sure sequence number has network byte order
36. UINT32_HTON(txPacket.seqNumber);    // 改变发送序号的字节顺序
```

```
37. basicRfSendPacket(RX_ADDR, (uint8*)&txPacket, PACKET_SIZE);

38. // Change byte order back to host order before increment /*  在增加序号前将字节顺序改回为主机顺序  */

39. UINT32_NTOH(txPacket.seqNumber);
40. txPacket.seqNumber++;       //数据包序列号自加 1

41. pktsSent++;

42. appState = IDLE;
43. halLedToggle(1);     //改变 LED1 的亮灭状态
44. halMcuWaitMs(500);
45. }
46. else
47. appStarted = !appStarted;
48. // Reset statistics and sequence number/*  复位统计和序号  */
49. pktsSent = 0;
50. }
51. }
52. }
```

总结 appTransmitter 函数完成的任务：

1、初始化 Basic RF

2、设置发射功率

3、设定测试的数据包量

4、配置定时器和 IO

5、初始化数据包载荷

6、进行循环函数，不断地发送数据包，每发送完一次，下一个数据包的序

列号自加 1 再发送；

接收函数没有 define MODE_SEND 则进入 appReceiver ()

接收函数比较长，担心大家看到会反感，而且查看不方便，此处只把有必要说明的地方才贴出来，具体的全部代码内容肯定是打开工程看最好啦。。。

```
/*****************************************************************
    ***************
* @fn              appReceiver
*
* @brief          Application code for the receiver mode. Puts MCU in endless loop
*
* @param          basicRfConfig – file scope variable. Basic RF configuration data
*                 rxPacket – file scope variable of type perTestPacket_t
*
* @return         none
*/
```

```
1.  static void appReceiver()
2.  {
3.  initUART();                          // 初始化串口
4.  basicRfConfig.myAddr = RX_ADDR;
5.  if(basicRfInit(&basicRfConfig)==FAILED) Initialize BasicRF //初始化 Basic
    RF
6.  {
7.  HAL_ASSERT(FALSE);
8.  }
9.  basicRfReceiveOn();
10. while (TRUE)
11. {
```

```
12. while(!basicRfPacketIsReady());   // 等待新的数据包
13. if(basicRfReceive((uint8*)&rxPacket, MAX_PAYLOAD_LENGTH, &rssi)>0)
{
14. halLedSet(3);//*************P1_4
15. UINT32_NTOH(rxPacket.seqNumber);   // 改变接收序号的字节顺序
16. segNumber = rxPacket.seqNumber
17. //若统计被复位，设置期望收到的数据包序号为已经收到的数据包序
号
18. if(resetStats)
19. {
20. rxStats.expectedSeqNum = segNumber;
21. resetStats=FALSE;
22. }
23. // Subtract old RSSI value from sum
24. rxStats.rssiSum -= perRssiBuf[perRssiBufCounter];   // 从 sum 中减去
旧的 RSSI 值
25. // Store new RSSI value in ring buffer, will add it to sum later
26. perRssiBuf[perRssiBufCounter] =  rssi;   // 存储新的 RSSI 值到环形缓
冲区，之后它将被加入 sum
27. rxStats.rssiSum += perRssiBuf[perRssiBufCounter];   // 增加新的 RSSI
值到 sum
28. if(++perRssiBufCounter == RSSI_AVG_WINDOW_SIZE) {
29. perRssiBufCounter = 0;          // Wrap ring buffer counter
30. }
31. // Check if received packet is the expected packet     检查接收到的数
据包是否是所期望收到的数据包
32. if(rxStats.expectedSeqNum == segNumber)   // 是所期望收到的数据
包
33. {
```

```
34. rxStats.expectedSeqNum++;
35. }
36. // If there is a jump in the sequence numbering this means some packets inbetween has been lost.
37. else if(rxStats.expectedSeqNum < segNumber) //大于期望收到的数据包的序号
38. {                                          // 认为丢包
39. rxStats.lostPkts += segNumber – rxStats.expectedSeqNum;
40. rxStats.expectedSeqNum = segNumber + 1;
41. }
42. else   // 小于期望收到的数据包的序号
43. {
44. rxStats.expectedSeqNum = segNumber + 1;
45. rxStats.rcvdPkts = 0;
46. rxStats.lostPkts = 0;
47. }
48. rxStats.rcvdPkts++;
49. /******************** 以下为串口打印部分的函数**********************/
50. temp_receive=(int32)rxStats.rcvdPkts;
51. if(temp_receive>1000)
52. {
53. if(halButtonPushed()==HAL_BUTTON_1){
54. resetStats = TRUE;
55. rxStats.rcvdPkts = 1;
56. rxStats.lostPkts = 0;
57. }
58. }
59. Myreceive[0]=temp_receive/100+’0’;      //打印接收到数据包的个数
```

```
60. Myreceive[1]=temp_receive%100/10+'0';
61. Myreceive[2]=temp_receive%10+'0';
62. Myreceive[3]='\0';
63. UartTX_Send_String("RECE:",strlen("RECE:"));
64. UartTX_Send_String(Myreceive,4);
65. UartTX_Send_String("      ",strlen("      "));
66. temp_per=(int32)((rxStats.lostPkts*1000)/(rxStats.lostPkts+rxStats.rcvd
Pkts));
67. Myper[0]=temp_per/100+'0';                    //打印当前计算出来的误
包率
68. Myper[1]=temp_per%100/10+'0';
69. Myper[2]='.';
70. Myper[3]=temp_per%10+'0';
71. Myper[4]='%';
72. UartTX_Send_String("PER:",strlen("PER:"));
73. UartTX_Send_String(Myper,5);
74. UartTX_Send_String("      ",strlen("      "));
75. temp_rssi=(0-(int32)rxStats.rssiSum/32);   //打印上 32 个数据包的 RSSI
值的平
76.                                              均值
77. Myrssi[0]=temp_rssi/10+'0';
78. Myrssi[1]=temp_rssi%10+'0';
79. UartTX_Send_String("RSSI:-",strlen("RSSI:-"));
80. UartTX_Send_String(Myrssi,2);
81. UartTX_Send_String("\n",strlen("\n"));
82. halLedClear(3);
83. halMcuWaitMs(300);
84.    }
85.  }
```

```
86. }
```

那么长,有点头晕是吧。不用一句句地看，撒几眼你就知道这个函数干了什么事了…但还是建议直接打开工程学习。

接收函数的作用：

1、 串口在此初始化

2、 初始化 Basic RF

3、 不断地接收数据包，并检查数据包序号是否为期望值，作出相应处理

4、 串口打印出，接收包的个数\误包率及上 32 个数据包的 RSSI 值的平均值

有几个比较重要的数据作个简要的说明一下：

为了获取传输的性能参数，接收器中包含了如下几个数据（包含在 rxStats 变量中，其类型为 perRxStats_t）

rxStats.expectedSeqNum	预计下一个数据包的序号， 其值等于“成功接收的数据包”+“丢失的数据包”+1
rxStats.rssiSum	上 32 个数据包的 RSSI 值的和
rxStats.rcvdPkts	每次 PER 测试中，成功接收到的数据包的个数
rxStats.lostPkts	丢失数据包的个数

这些数据具体是怎么得来，我们没有必要具体去分析，直接读取我们感兴趣的数据就可以了。

误包率又是怎么计数的呢？TI 公司的使用文档有说明的

The PER value per thousand packets is calculated by the formula:
PER = 1000* *rxStats.lostPkts/ (rxStats. lostPkts+ rxStats. rcvdPkts)*
(for rxStats. rcvdPkts>=1)

如果大家想了解具体的话就可以去\CC2530 BasicRF\docs 文件夹中找到 CC2530_Software_Examples.pdf 文档 4.2 章节有详细介绍的

实验操作：

a) 下载发射模块，在 per_test.c 中，找到：

```
/*******************important select or
 shelt*****************************/
#define MODE_SEND                    //屏蔽时：appReceiver
                                     //不屏蔽时：appTransmitter
/*****************************************************************/
```

不要屏蔽#define MODE_SEND

编译下载到发射模块

b) 下载接收模块，同样的找到

```
/*******************important select or
 shelt*****************************/
//#define MODE_SEND                  //屏蔽时：appReceiver
                                     //不屏蔽时：appTransmitter
/*****************************************************************/
```

要屏蔽#define MODE_SEND

编译下载到接收模块

3、 接收模块 USB 连接 PC 机并给发射模块供电，打开串口调试助手，并设置好相应的 COM 口和波特率，先开接收模块，再开发送模块。然后就可以看到我们的实验现象了。如图 5 所示。

由于距离比较近，所以掉包不明显的，有兴趣的可以把发送节点拿到较远的地方，然后观察掉包率。或者先打开发送模块，再打开接收模块来测试掉包，会显示出掉包情况。

图 3.15 实验现象 （包数量-掉包率-RSSI）

3.4 协议栈工作原理介绍

前言：

前文已经有多次地方提及到协议栈，但是迟迟没有做一个介绍。呵呵，不是不讲，时候未到！我们需要在最适合的时候做最适合的事。今天，我们来讲述一下协议栈的工作原理，这个东西将是我们以后接触得最多的东西，从学习到项目开发，你不得不和他打交道。由于我们的学习平台是基于 TI 公司的，所以讲述的当然也是 TI 的 Z-STACK。

内容讲解：

相信大家已经知道 CC2530 集成了增强型的 8051 内核，在这个内核中进行组网通讯时候，如果再像以前基础实验的方法来写程序，相信大家都会望而止步，ZigBee 也不会在今天火起来了。所以 ZigBee 的生产商很聪明，比如 TI 公司，他们为你搭建一个小型的操作系统（本质也是大型的程序），名叫 Z-stack。他们帮你考虑底层和网络层的内容，将复杂部分屏蔽掉。让用户通过 API 函数就可以轻易用 ZigBee。这样大家使用他们的产品也理所当然了，确实高明。

也就是说，协议栈是一个小操作系统。大家不要听到是操作系统就感觉到很复杂。回想我们当初学习 51 单片机时候是不是会用到定时器的功能？嗯，我们会利用定时器计时，令 LED 一秒改变一次状态。好，现在进一步，我们利用同一个定时器计时，令 LED1 一秒闪烁一次，LED2 二秒闪烁一次。这样就有 2 个任务了。再进一步…有 n 个 LED,就有 n 个任务执行了。协议栈的最终工作原理也一样。从它工作开始，定时器周而复始地计时，有发送、接收…等任务要执行时就执行。这个方式称为任务轮

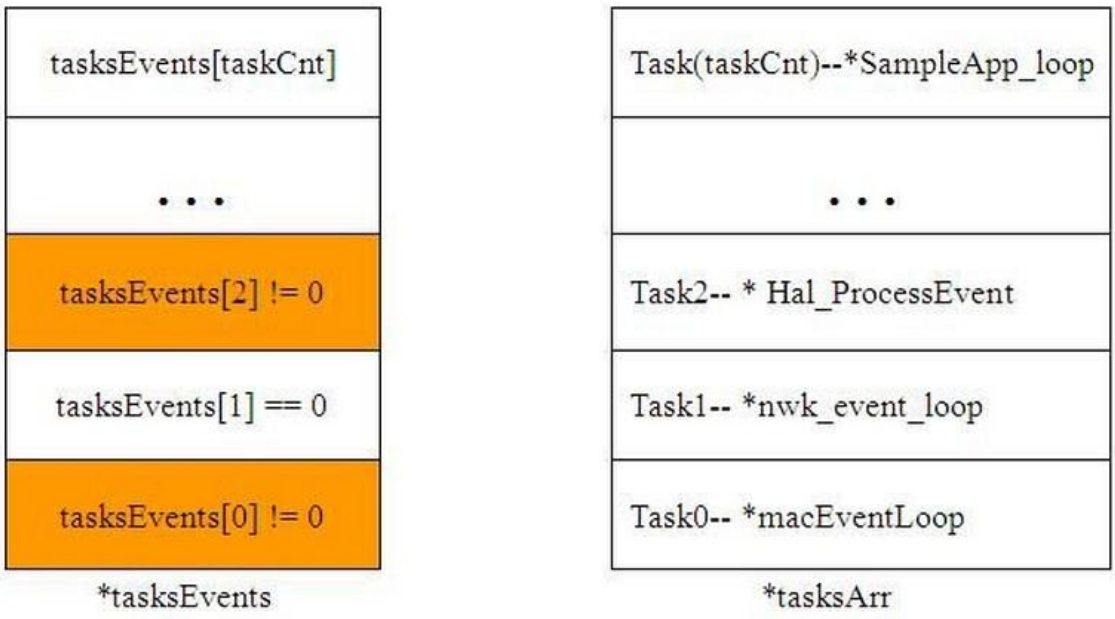


图 3.16 任务轮询

协议栈很久没打开了吧？没什么神秘的，我直接拿他们的东西来解剖！我们打开协议栈文件夹 Texas Instruments\Projects\zstack 。里面包含了 TI 公司的例程和工具。其中的功能我们会在用的的实验里讲解。再打开 Samples 文件夹：

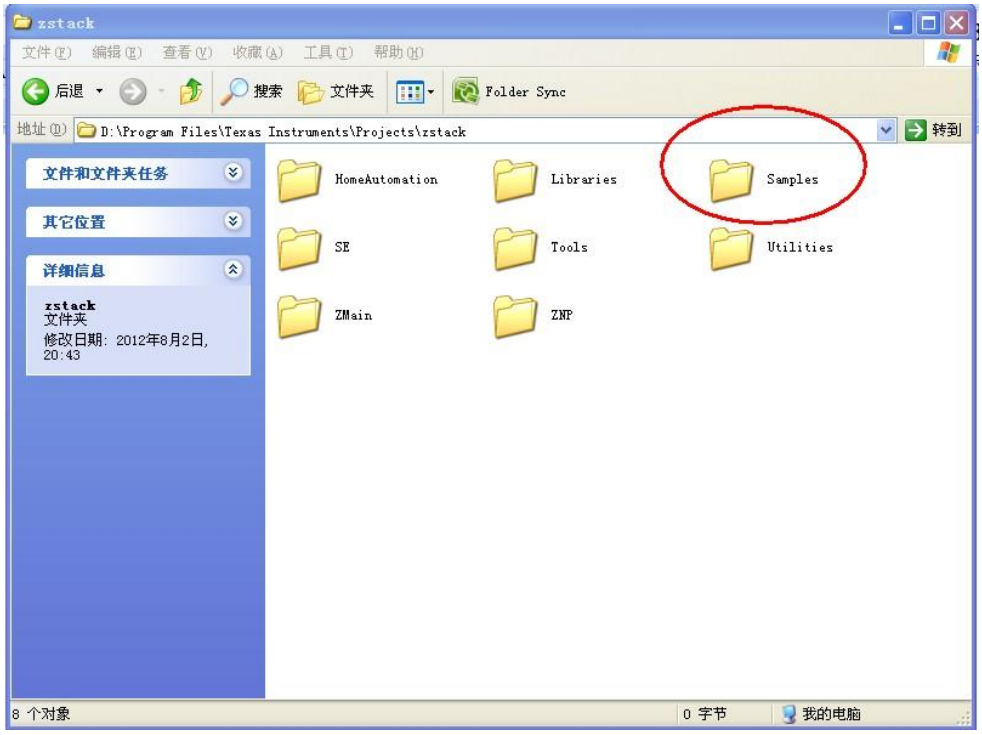


图 3.17

Samples 文件夹里面有三个例子: GenericApp、SampleApp、SimpleApp 在这里们选择 SampleApp 对协议栈的工作流程进行讲解。打开 \SampleApp\CC2530DB 下工程文件 SampleApp.eww。留意左边的工程目录，我们暂时只需要关注 Zmain 文件夹和 App 文件夹。

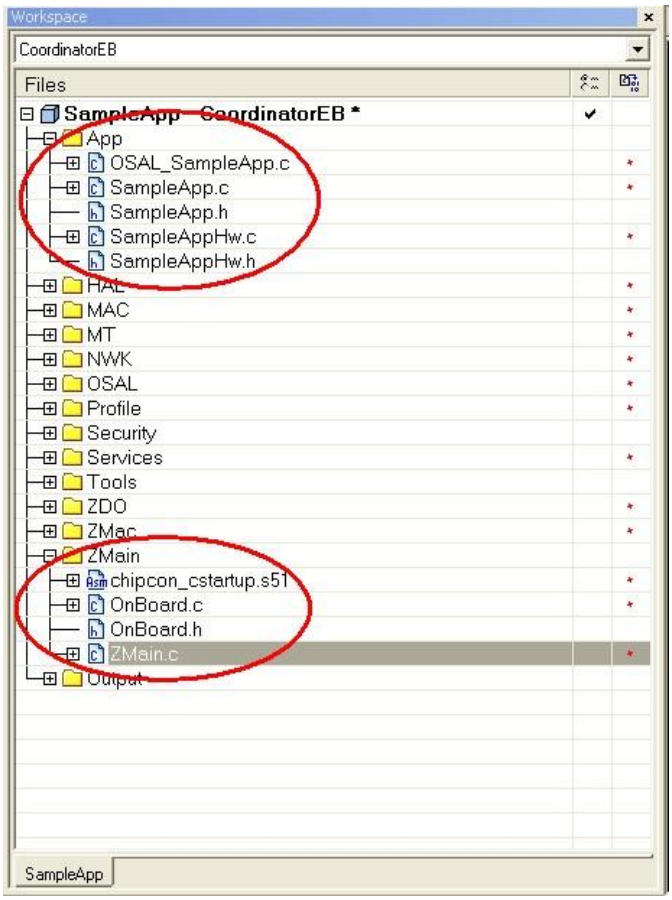


图 3.18 工作空间目录

任何程序都在 main 函数开始运行，Z-STACK 也不例外。打开 Zmain.C,找到 int main(void) 函数。我们大概浏览一下 main 函数代码：

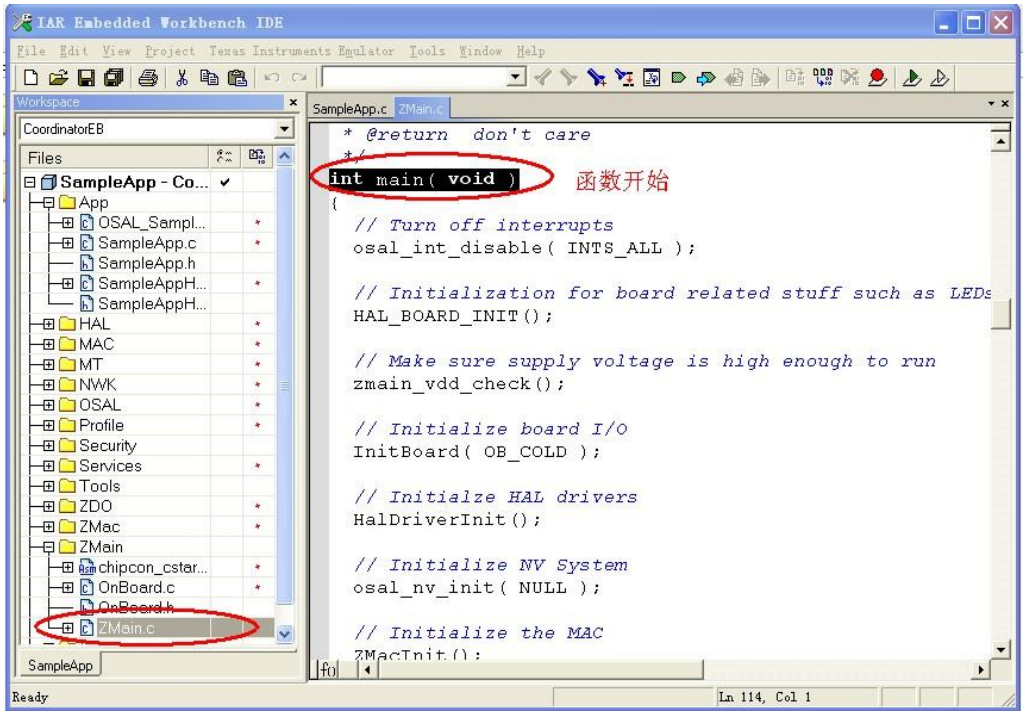


图 3.19 协议栈主函数

```
/*************************************************************************
 * @fn          main
 * @brief       First function called after startup.
 * @return      don't care
 */
int main( void )
{
  // Turn off interrupts
  osal_int_disable( INTS_ALL );     //关闭所有中断

  // Initialization for board related stuff such as LEDs
  HAL_BOARD_INIT();                 //初始化系统时钟

  // Make sure supply voltage is high enough to run
  zmain_vdd_check();                //检查芯片电压是否正常
```

```
  // Initialize board I/O
  InitBoard( OB_COLD );          //初始化 I/O ， LED 、Timer  等

  // Initialze HAL drivers
  HalDriverInit();               //初始化芯片各硬件模块

  // Initialize NV System
  osal_nv_init( NULL );           // 初始化 Flash  存储器

  // Initialize the MAC
  ZmacInit();            //初始化 MAC 层

  // Determine the extended address
  zmain_ext_addr();      //确定 IEEE  64 位地址

  // Initialize basic NV items
  zgInit();               // 初始化非易失变量

#ifndef  NONWK
  // Since the AF isn't a task, call it's initialization routine
  afInit();
#endif

  // Initialize the operating system
  osal_init_system();     // 初始化操作系统

  // Allow interrupts
  osal_int_enable( INTS_ALL );    // 使能全部中断
```

```
    // Final board initialization
    InitBoard( OB_READY );       //  初始化按键

    // Display information about this device
    zmain_dev_info();            //显示设备信息

    /* Display the device info on the LCD */
  #ifdef LCD_SUPPORTED
    zmain_lcd_init();
  #endif

  #ifdef    WDT_IN_PM1
    /* If WDT is used, this is a good place to enable it. */
    WatchDogEnable( WDTIMX );
  #endif
 osal_start_system(); // No Return from here  执行操作系统，进去后不会返回
 return 0;                 // Shouldn't get here.
  }
```

我们大概看了上面的代码后，可能感觉很多函数不认识。没关系，代码很有条理性，开始先执行初始化工作。包括硬件、网络层、任务等的初始化。然后执行 osal_start_system();操作系统。进去后可不会回来了。在这里，我们重点了解 2 个函数：

c) 初始化操作系统

```
osal_init_system();
```

d) 运行操作系统

```
osal_start_system();
```

怎么看？在函数名上单击右键——go to definition of…，便可以进入函数。

1、 我们先来看 osal_init_system();系统初始化函数，进入函数。发现里面有 6 个初始化函数，没事，我们需要做的是掐住咽喉。这里我们只关心 osalInitTasks();任务初始化函数。继续由该函数进入。

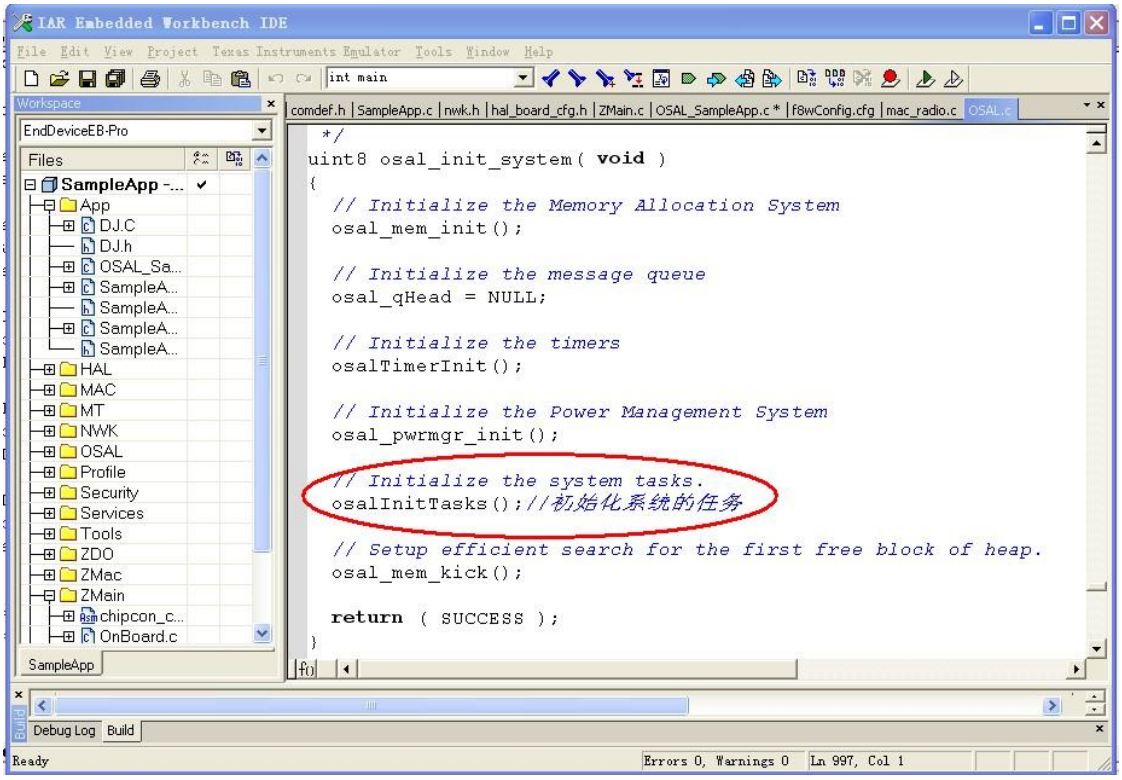


图 3.20

终于到尽头了。这一下子代码更不熟悉了。不过我们可以发现，函数好像能在 taskID 这个变量上找到一定的规律。请看下面程序注释。

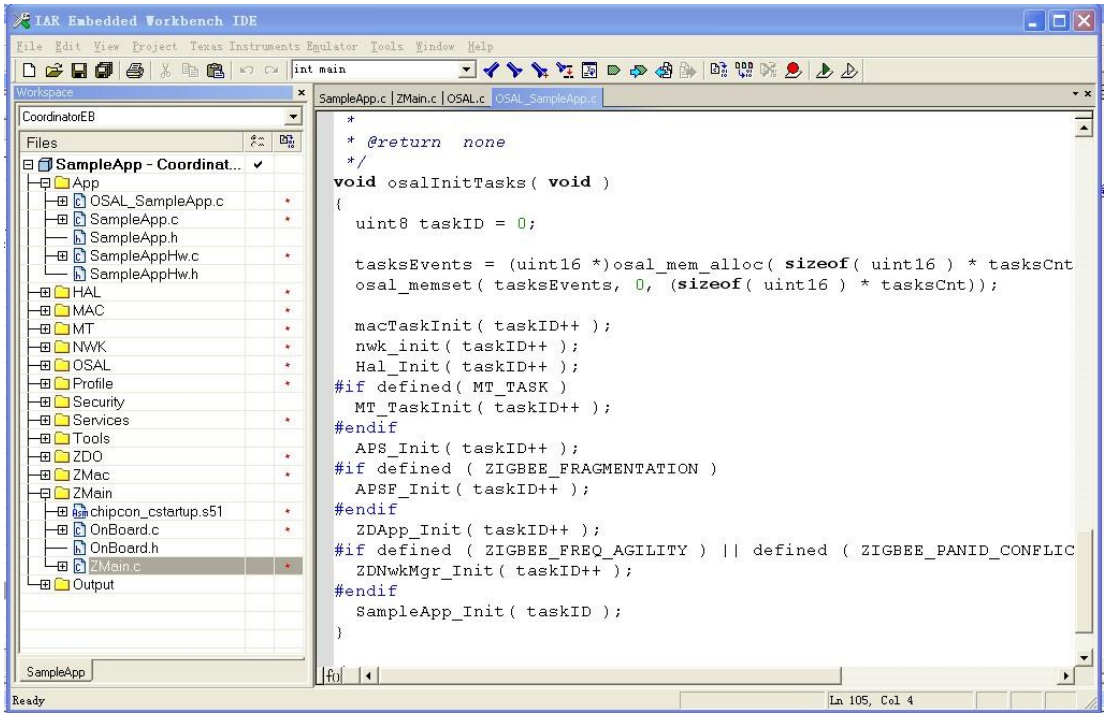


图 3.21

```
void osalInitTasks( void )
{
    uint8 taskID = 0;

// 分配内存，返回指向缓冲区的指针
tasksEvents = (uint16 *)osal_mem_alloc( sizeof( uint16 ) * tasksCnt);

    // 设置所分配的内存空间单元值为 0
osal_memset( tasksEvents, 0, (sizeof( uint16 ) * tasksCnt));

// 任务优先级由高向低依次排列，高优先级对应 taskID  的值反而小
    macTaskInit(taskID ++ );        //macTaskInit(0) ，用户不需考虑
    nwk_init(taskID ++ );           //nwk_init(1)，用户不需考虑
    Hal_Init(taskID ++ );           //Hal_Init(2) ，用户需考虑
#if defined( MT_TASK )
```

```
    MT_TaskInit(taskID ++ );
#endif
    APS_Init(taskID ++ );              //APS_Init(3) ，用户不需考虑
#if defined ( ZIGBEE_FRAGMENTATION )
    APSF_Init(taskID ++ );
#endif
    ZDApp_Init(taskID ++ );                  //ZDApp_Init(4) ，用户需考虑
#if defined ( ZIGBEE_FREQ_AGILITY ) || defined ( ZIGBEE_PANID_CONFLICT )
    ZDNwkMgr_Init(taskID ++ );
#endif
    SampleApp_Init(taskID );            // SampleApp_Init _Init(5) ，用户需考虑
}
```

我们可以这样理解，函数对 taskID 个东西进行初始化，每初始化一个，taskID++。大家看到了注释后面有些写着用户需要考虑，有些则写着用户不需考虑。没错，需要考虑的用户可以根据自己的硬件平台或者其他设置，而写着不需考虑的也是不能修改的。TI 公司出品协议栈已完成的东西。这里先提前卖个关子 SampleApp_Init(taskID);很重要，也是我们应用协议栈例程的必需要函数，用户通常在这里初始化自己的东西。

至此，osal_init_system();大概了解完毕。

2、 我们再来看第二个函数 osal_start_system();运行操作系统。同样用 go to definition 的方法进入该函数。呵呵，结果发现很不理想。甚至很多函数形式没见过。看代码吧：

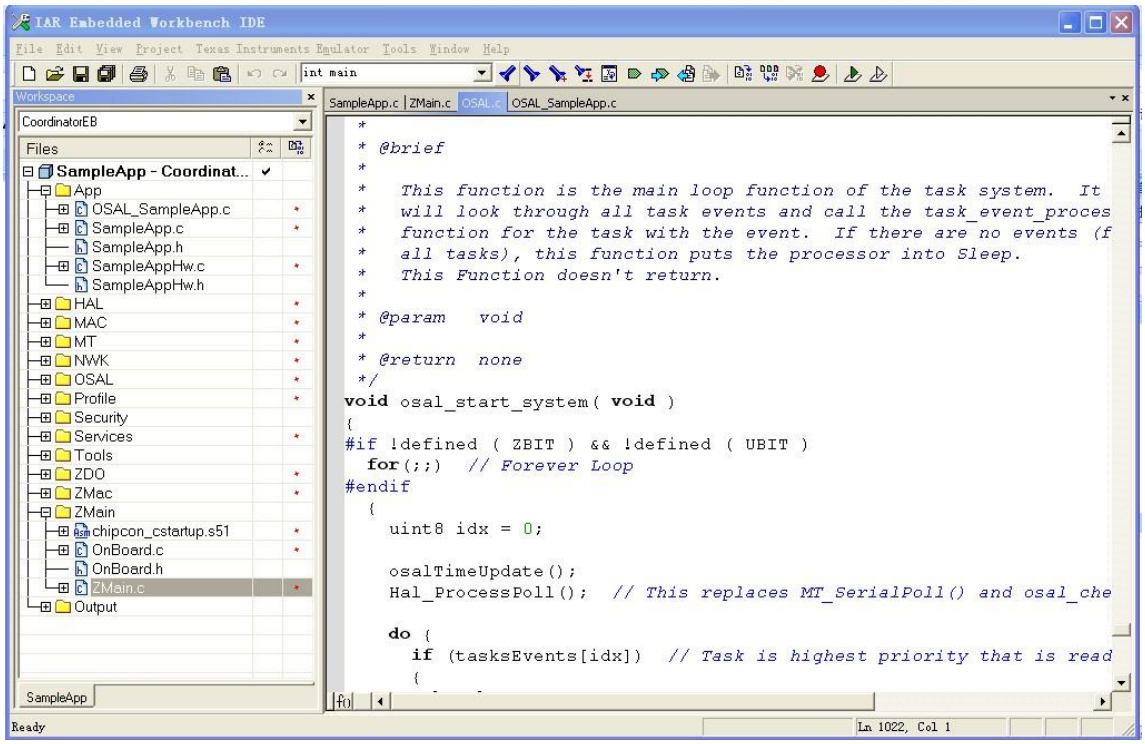


图 3.22 osal_start_system();函数

```
/*************************************************************
***
 * @fn          osal_start_system

 * @brief*
 *      This function is the main loop function of the task system.    It
 *      will look through all task events and call the task_event_processor()
 *      function for the task with the event.    If there are no events (for
 *      all tasks), this function puts the processor into Sleep.
 *      This Function doesn't return.
```

翻译：这个是任务系统轮询的主要函数。他会查找发生的事件然后调用相应的事件执行函数。如果没有事件登记要发生，那么就进入睡眠模式。这个函数是永远不会返回的。

---是不是看完官方的介绍清晰了一点？我的英语水平都可以，相信你用心也可以的。---

```
  * @param      void
  * @return    none
  ****************************************/
 void osal_start_system( void )
 {

 #if !defined ( ZBIT ) && !defined ( UBIT )
   for(;;)    // Forever Loop
 #endif
{
     uint8 idx = 0;
 osalTimeUpdate();//这里是在扫描哪个事件被触发了，然后置相应的标志位
 Hal_ProcessPoll();    // This replaces MT_SerialPoll() and osal_check_timer().
     Do {
         if (tasksEvents[idx])    // Task is highest priority that is ready.
         {
           break;    //       得到待处理的最高优先级任务索引号  idx
         }
 } while (++idx < tasksCnt);

 if (idx < tasksCnt)
 {
         uint16 events;
  halIntState_t intState;

  HAL_ENTER_CRITICAL_SECTION(intState);    //  进入临界区,保护
```

```
    events = tasksEvents[idx];              //提取需要处理的任务中的事件
    tasksEvents[idx] = 0;   // Clear the Events for this task.清除本次任务的事件
    HAL_EXIT_CRITICAL_SECTION(intState); //  退出临界区

    events = (tasksArr[idx])( idx, events );//通过指针调用任务处理函数，关键

    HAL_ENTER_CRITICAL_SECTION(intState);//进入临界区
    tasksEvents[idx] |= events;   // Add back unprocessed events to the current
                                        task.保存未处理的事件
    HAL_EXIT_CRITICAL_SECTION(intState); //  退出临界区
  }
  #if defined( POWER_SAVING )
  else    // Complete pass through all task events with no activity?
  {
        osal_pwrmgr_powerconserve(); // Put the processor/system into sleep
  }
  #endif
    }
  }
```

我们来关注一下 events = tasksEvents[idx]; 进入 tasksEvents[idx]数组定义，如图 3. 4H，发现恰好在刚刚 osalInitTasks(void)函数上面。而且 taskID 一一对应。这就是初始化与调用的关系。taskID 把任务联系起来了。

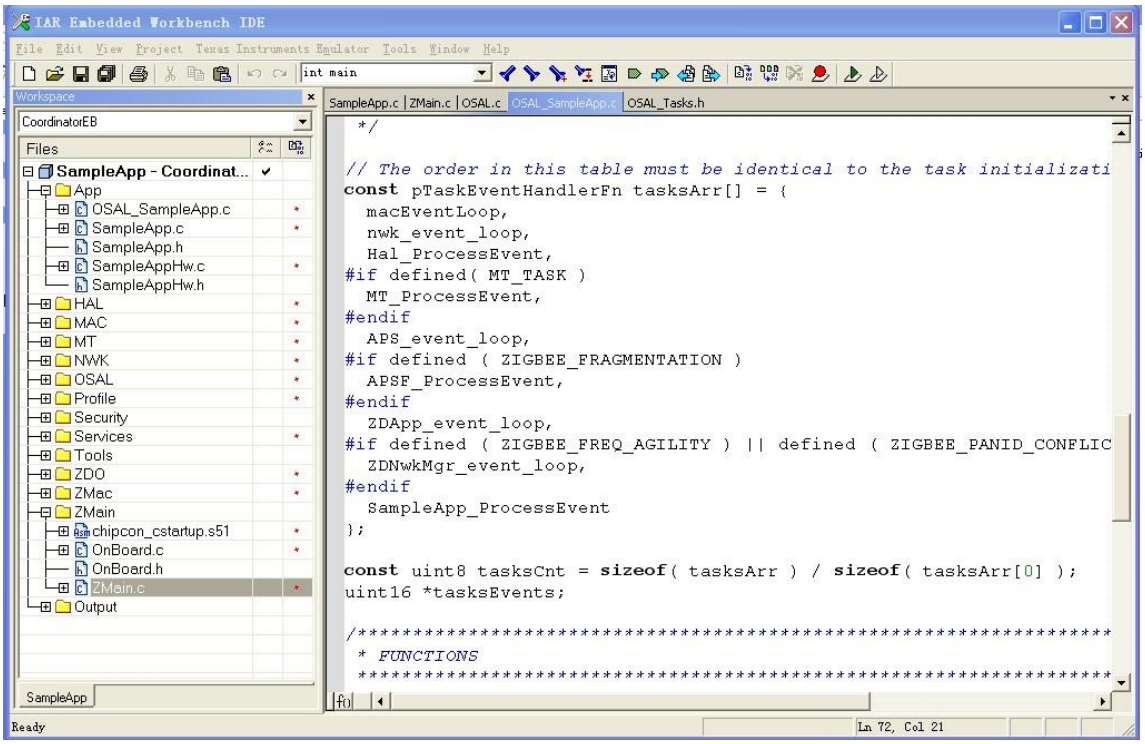
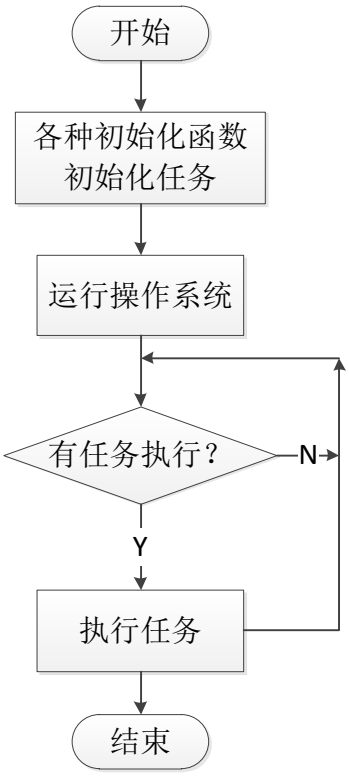


图 3.23

关于协议栈的介绍先到这里，其他会在以后的实例中结合程序来介绍，这样会更直观。大家可以根据需要再熟悉一下函数里面的内容。游一下这个代码的海洋。我们可以总结出一个协议栈简单的工作流程。



协议栈简要流程

图 3.24 协议栈工作流程

3.5 协议栈中的串口实验

前言：

相信大家经过前面 BasicRF 实验后对无线传输的原理有一定的理解，是不是迫不及待想进行数据通讯？当初本人也是这样，学完了点灯就想来的实际点的数据传输。但是我们想想，我们想传输数据的原因是？相信大部分人会答当然是采集到温度传感器等信息啦。没错，我们接收到节点发来的信息，通过串口的方法发给电脑上位机，以最直观的方法展示出来。串口作为一种最简单的协议栈和调试者接口，在 Zigbee 的学习和应用过程中具有非常重要的作用。所以，我们需要先学习在协议栈里加入串口功能。这与基础实验实现的方法不同。

实现平台： WeBee CC2530 模块及功能底板。

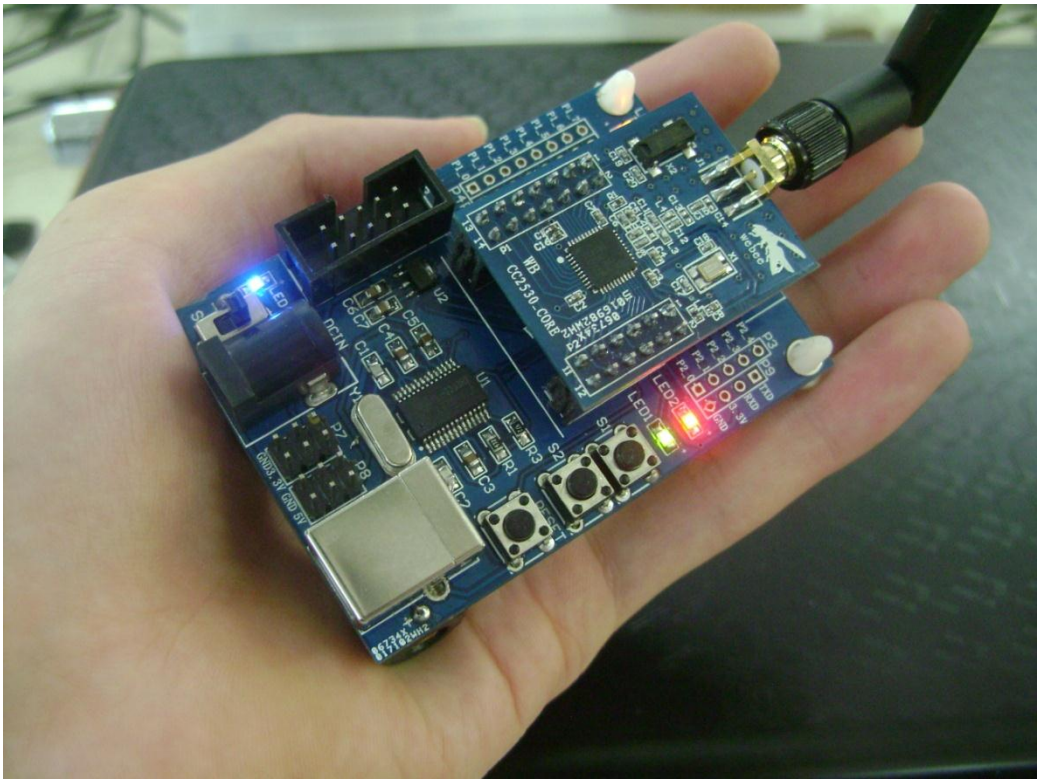


图 3.25 网蜂 CC2530 模块+功能底板

实验现象： 模块通过串口发送“HELLO WEBEE!”给电脑串口调试助手打印出来。整个实验在协议栈（TI z-stack 2.5.1a）中进行。

实验讲解：

整个例程很简单，分三步走，实际上就是三个语句，不过我们可以了解一下具体原理：代码不好啃，想长命一点的还是看网蜂教程吧。呵呵。步骤如下：

1、串口初始化

2、登记任务号

3、串口发送

我们打开 Z-stack 目录 Projects\zstack\Samples\ SamplesAPP\CC2530DB 里面的 SampleApp.eww 工程。这次试验我们直接基于协议栈的；SampleApp 来进行。

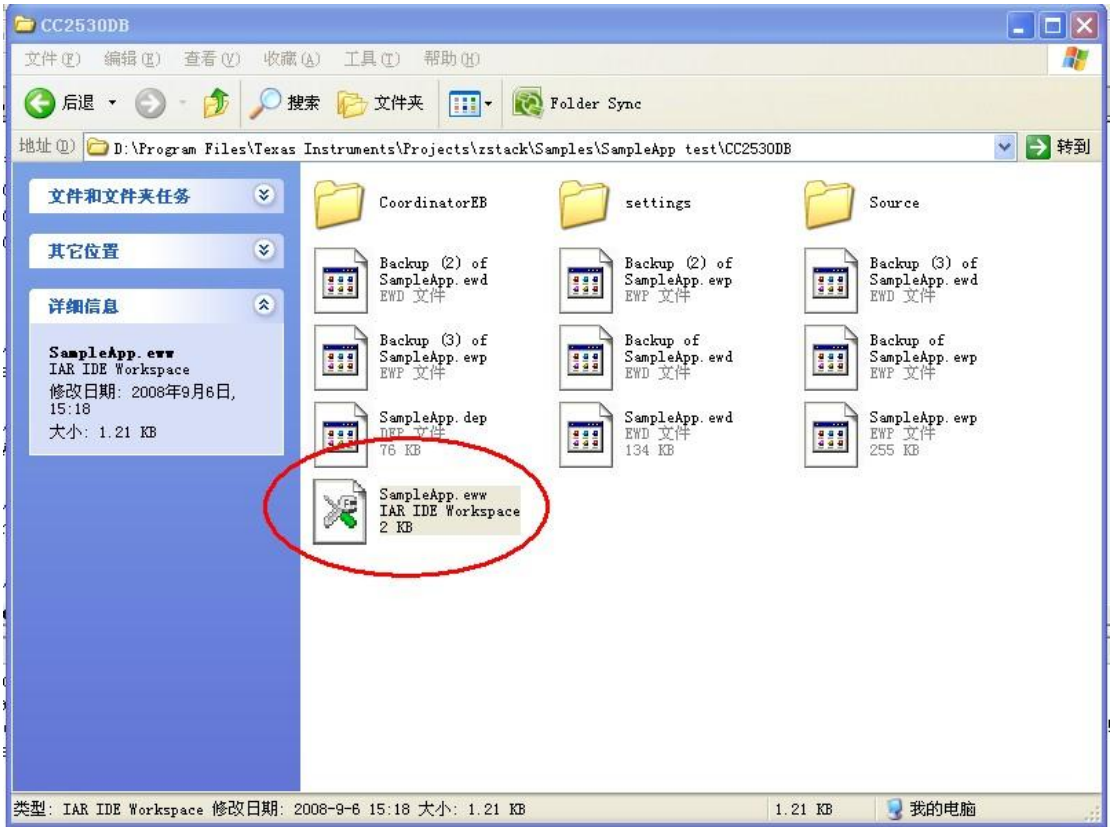


图 3.26 SampleApp.eww 工程

打开工程后，我们可以看到上一节说到 workspace 目录下比较重要的两个文件夹，Zmain 和 App。这里我们主要用到 App，这也是用户自己添加自己代码的地方。主要在 SampleApp.c 和 SampleApp.h 中就可以了,如图 3.27。

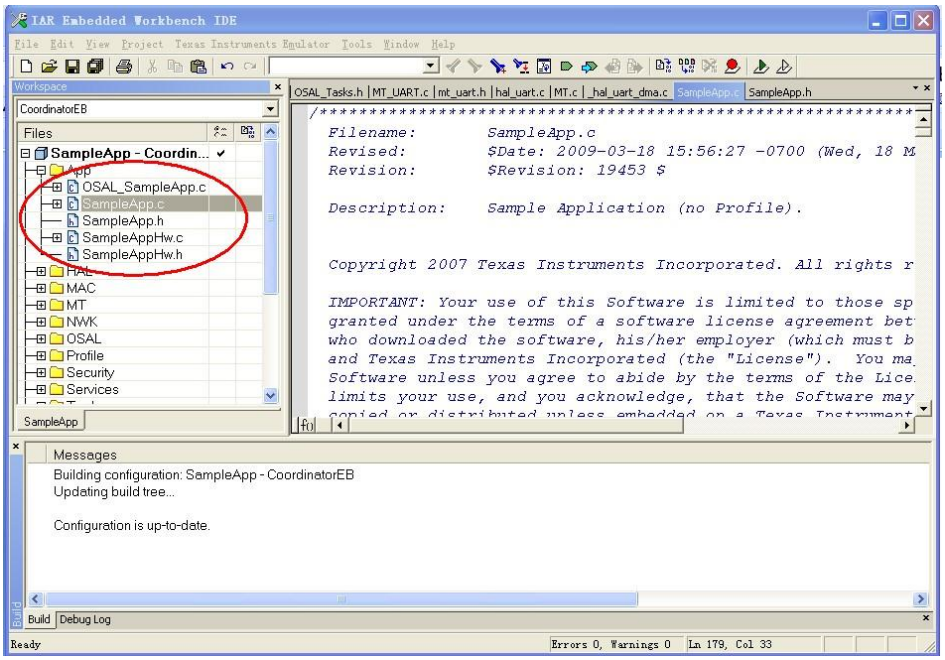


图 3.27 SampleApp.eww 工程主要文件

第一步：串口初始化

串口初始化大家很熟悉，就是配置串口号、波特率、流控、校验位等等。以前我们都是配置好寄存器然后使用。现在我们在 workspace 下找到 HAL\Target\CC2530EB\drivers 的 hal_uart.c 文件，我们可以看到里面已经包括了串口初始化、发送、接收等函数，是不是觉得很方便？

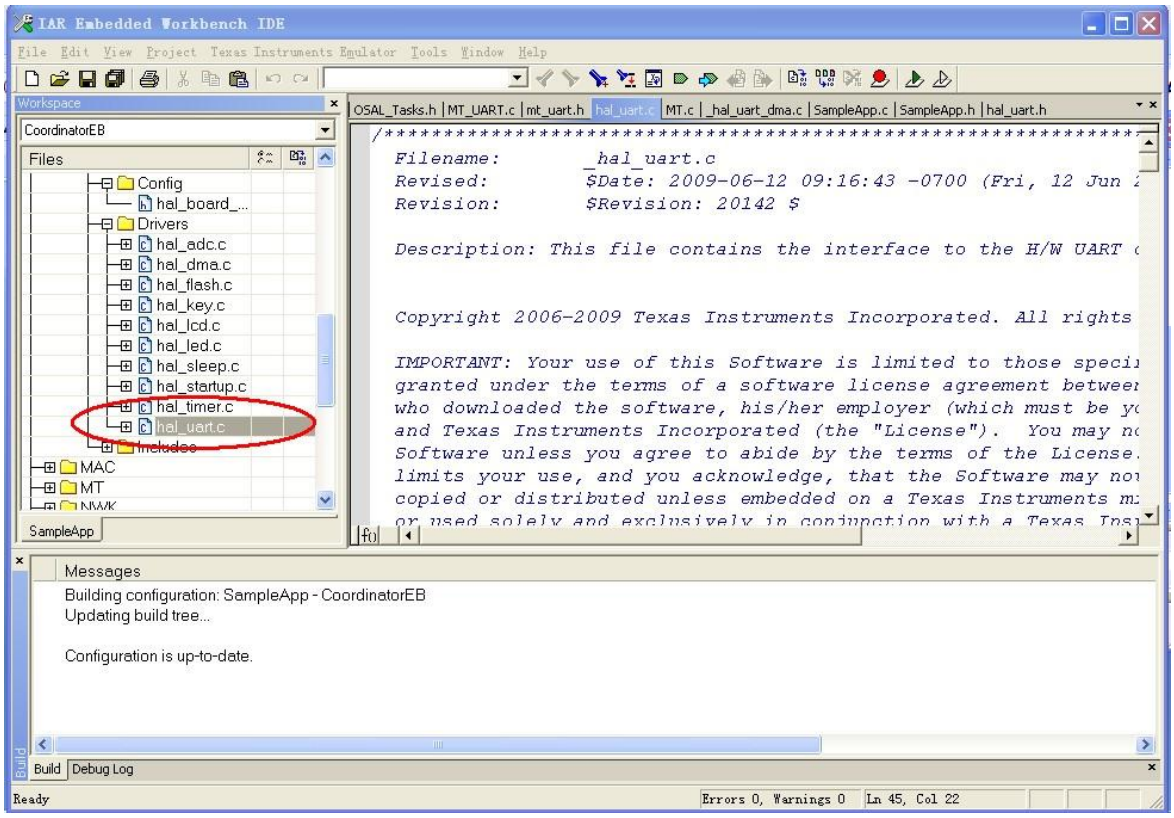


图 3.28

呵呵，浏览一下关于串口的操作函数还是挺全的，但是好的东西还在后面！！！我们看看 workspace 上的 MT 层，发觉有很多基本函数，前面带 MT。包括 MT_UART.C，我们打开这个文件。看到 MT_UartInit()函数，这里也有一个串口初始化函数的，没错 Z-stack 上有一个 MT 层，用户可以选用 MT 层配置和调用其他驱动。进一步简化了操作流程。

好了，我们已经知道串口配置的方法，那么应该在那里初始化呢？既然我们用的是 SampleApp 例程，当然是在 SampleApp 的文件下面啦。我们打开 APP 目录下的 OSAL_SampleApp.C 文件，找到上节提到的 osalInitTasks（）任务初始化函数中的 SampleApp_Init（）函数，进入这个函数，发现原来在 SampleApp.c 文件中。我们在这里加入串口初始化代码。

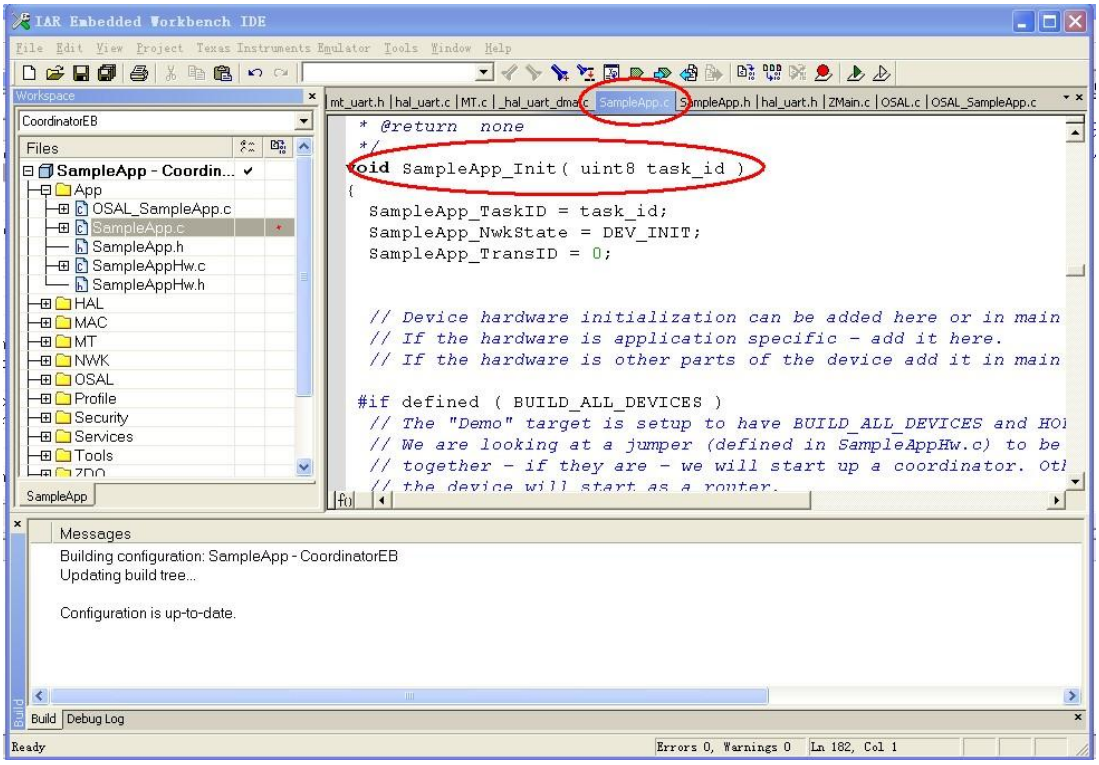


图 3.29

我们在函数第四行加入语句：MT_UartInit();如图 3.30 所示：

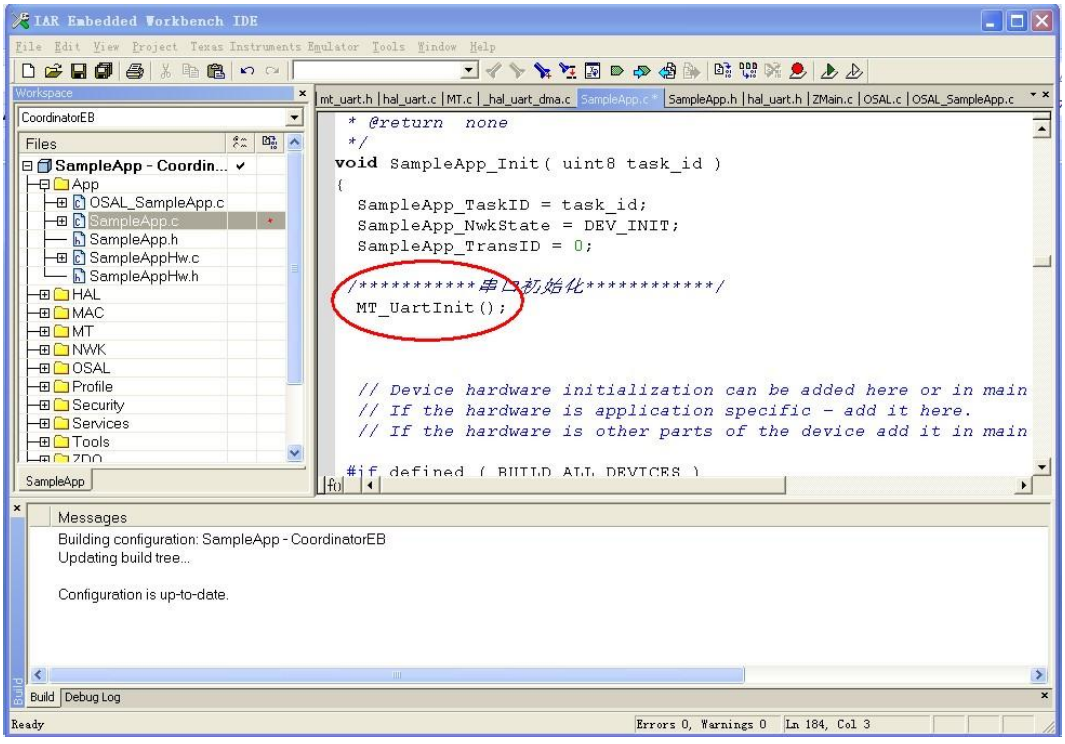


图 3.30

进入 MT_UartInit();，修改自己想要的初始化配置,进入函数后，发现代码如下。

```
1.  void MT_UartInit ()
2.  {
3.      halUARTCfg_t uartConfig;

4.      /* Initialize APP ID */

5.      App_TaskID = 0;

6.      /* UART Configuration */

7.  uartConfig.configured               = TRUE;
8.  uartConfig.baudRate                 = MT_UART_DEFAULT_BAUDRATE;
9.  uartConfig.flowControl               = MT_UART_DEFAULT_OVERFLOW;
10. uartConfig.flowControlThreshold = MT_UART_DEFAULT_THRESHOLD;
11. uartConfig.rx.maxBufSize       = MT_UART_DEFAULT_MAX_RX_BUFF;
12. uartConfig.tx.maxBufSize        = MT_UART_DEFAULT_MAX_TX_BUFF;
13. uartConfig.idleTimeout          = MT_UART_DEFAULT_IDLE_TIMEOUT;
14. uartConfig.intEnable            = TRUE;
15.
16. #if defined (ZTOOL_P1) || defined (ZTOOL_P2)
17. uartConfig.callBackFunc             = MT_UartProcessZToolData;
18. #elif defined (ZAPP_P1) || defined (ZAPP_P2)
19. uartConfig.callBackFunc             = MT_UartProcessZAppData;
20. #else
21. uartConfig.callBackFunc             = NULL;
22. #endif
```

```
23.  /* Start UART */

24.  #if defined (MT_UART_DEFAULT_PORT)
25.  HalUARTOpen (MT_UART_DEFAULT_PORT, &uartConfig);
26.  #else
27.  /* Silence IAR compiler warning */
28.  (void)uartConfig;
29.  #endif

30.  /* Initialize for Zapp */

31.  #if defined (ZAPP_P1) || defined (ZAPP_P2)
32.  /* Default max bytes that ZAPP can take */
33.  MT_UartMaxZAppBufLen    = 1;
34.  MT_UartZAppRxStatus     = MT_UART_ZAPP_RX_READY;
35.  #endif
36.  }
```

第 8 行：uartConfig.baudRate = MT_UART_DEFAULT_BAUDRATE;是配置波特率，

我们 go to definition of MT_UART_DEFAULT_BAUDRATE,

可以看到：

```
#define   MT_UART_DEFAULT_BAUDRATE   HAL_UART_BR_38400
```

默认的波特率是 38400bps,现在我们修改成 115200bps,修改如下：

```
#define   MT_UART_DEFAULT_BAUDRATE   HAL_UART_BR_115200
```

第 9 行：uartConfig.flowControl = MT_UART_DEFAULT_OVERFLOW;

语句是配置流控的，我们进入定义可以看到：

```
#define     MT_UART_DEFAULT_OVERFLOW        TRUE
```

默认是打开串口流控的，如果你是只连了 TX/RX 2 根线的方式务必关流控，像我们功能底板一样。

```
#define     MT_UART_DEFAULT_OVERFLOW        FALSE
```

注意：2 根线的通讯连接务必关流控，不然是永远收发不了信息的。

第 16~22 行：这个是预编译，根据预先定义的 ZTOOL 或者 ZAPP 选择不同的数据处理函数。后面的 P1 和 P2 则是串口 0 和串口 1。我们用 ZTOOL，串口 0。我们可以在 option——C/C++ 的 CompilerPreprocessor 里面看到,已经默认添加 ZTOOL_P1 预编译。

其他内容我们可以先不管，至此初始化配置完了。

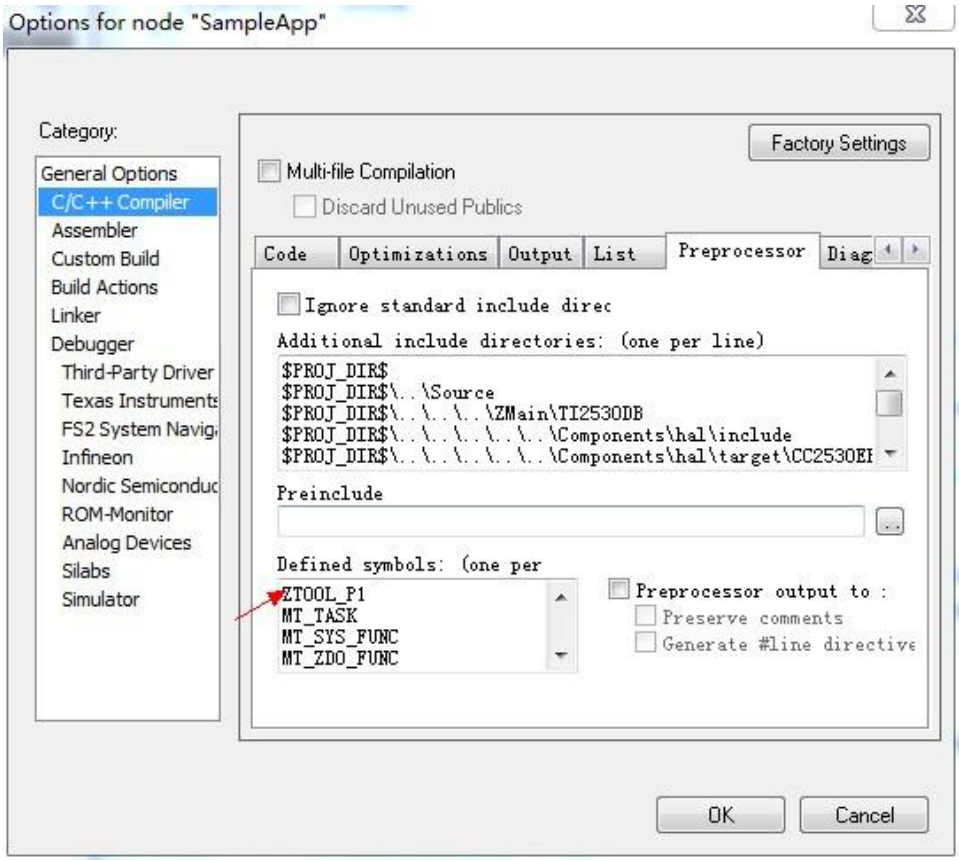


图 3.31

第二步：登记任务号

在 SampleApp_Init();刚添加的串口初始化语句下面加入语句：

MT_UartRegisterTaskID(task_id);//登记任务号

意思就是把串口事件通过 task_id 登记在 SampleApp_Init();里面。具体作用以后会提及。如图 3.32 所示：

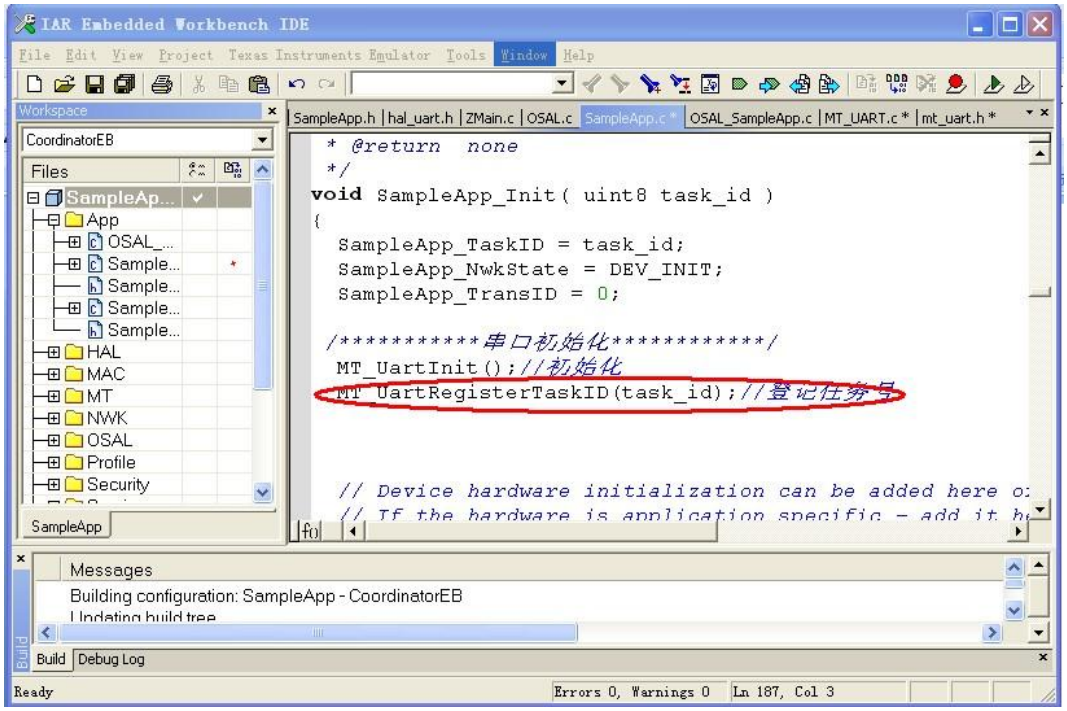


图 3.32

第三步：串口发送

经过前面两个步骤，现在串口已经可以发送信息了。我们在刚刚添加初始化代码后面加入一条上电提示 Hello World 的语句。

HalUARTWrite(0,"Hello World\n",12); （串口 0，‘字符’，字符个数。）

提示：需要在 SampleApp.c 这个文件里加入头文件语句：

#include "MT_UART.h"

连接仿真器和 USB 转串口线，选择 CoordinatorEB，点解下载并调试。全速运行，可以看到串口助手收到信息。图 3.33、图 3.34。

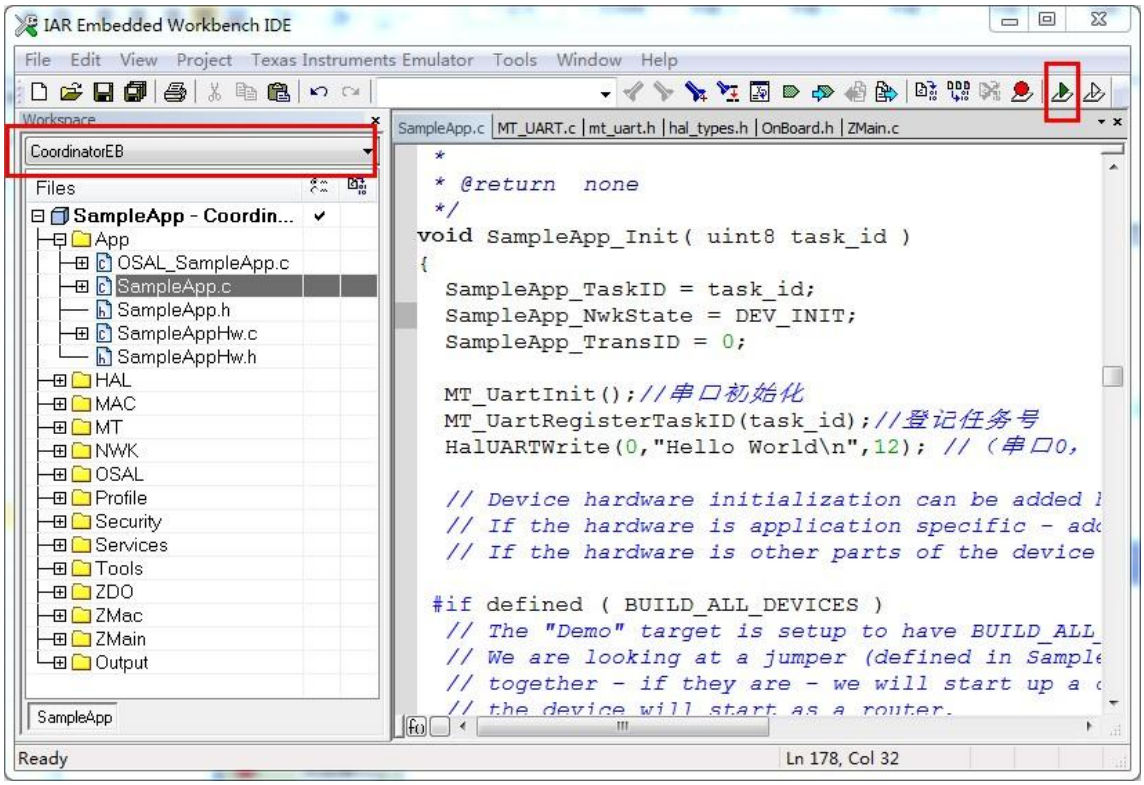


图 3.33

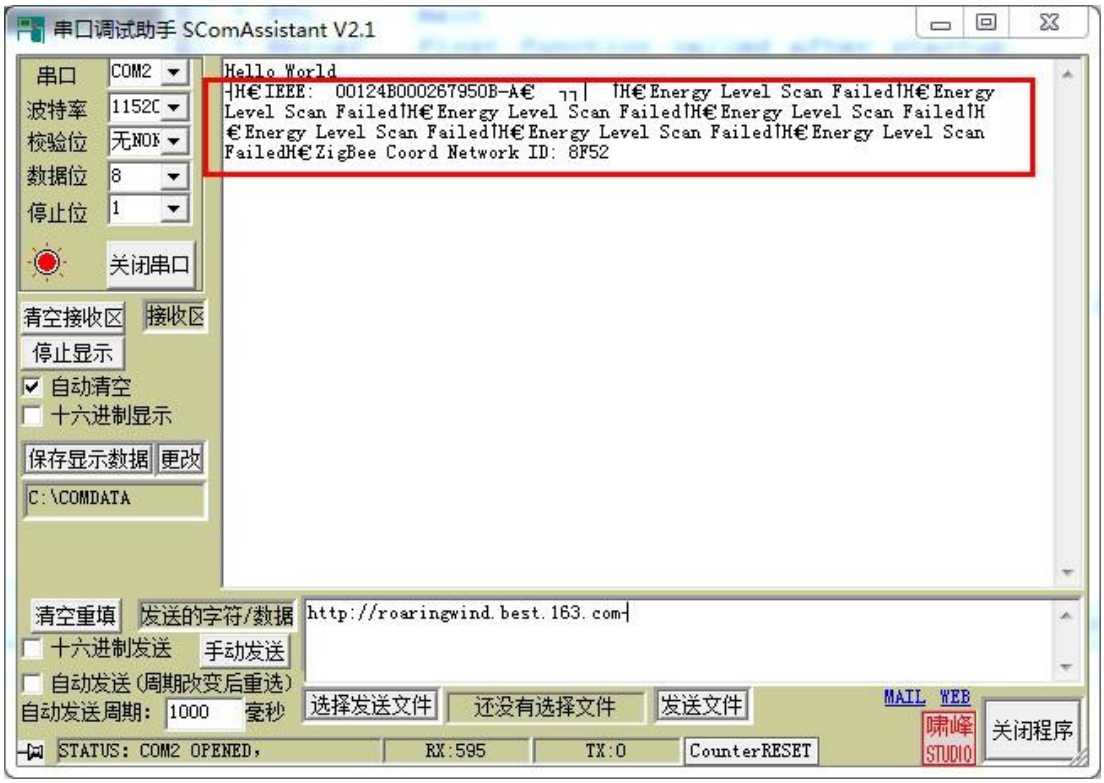


图 3.34 Hello World 后面出现乱码

图 3.34 显示可以接收代码了，是不是很简单？但是我们发现 Hello World 后面有一小段乱码。这是 Z-stack MT 层定义的串口发送格式，还有液晶提示信息。详细的以后内容会讲述。如果不想要的可以在预编译地方把 MT 和 LCD 相关内容注释。如：

ZTOOL_P1

xMT_TASK

xMT_SYS_FUNC

xMT_ZDO_FUNC

xLCD_SUPPORTED=DEBUG

xMT_TASK:表示没有定义 MT_TASK，也就是不定义了。其他几项也用这种方法，我们把改好的重新编译再下载，按复位键，观察串口已经没有乱码了。呵呵，如图 3.36, 或许就是你想要的 Hello World。

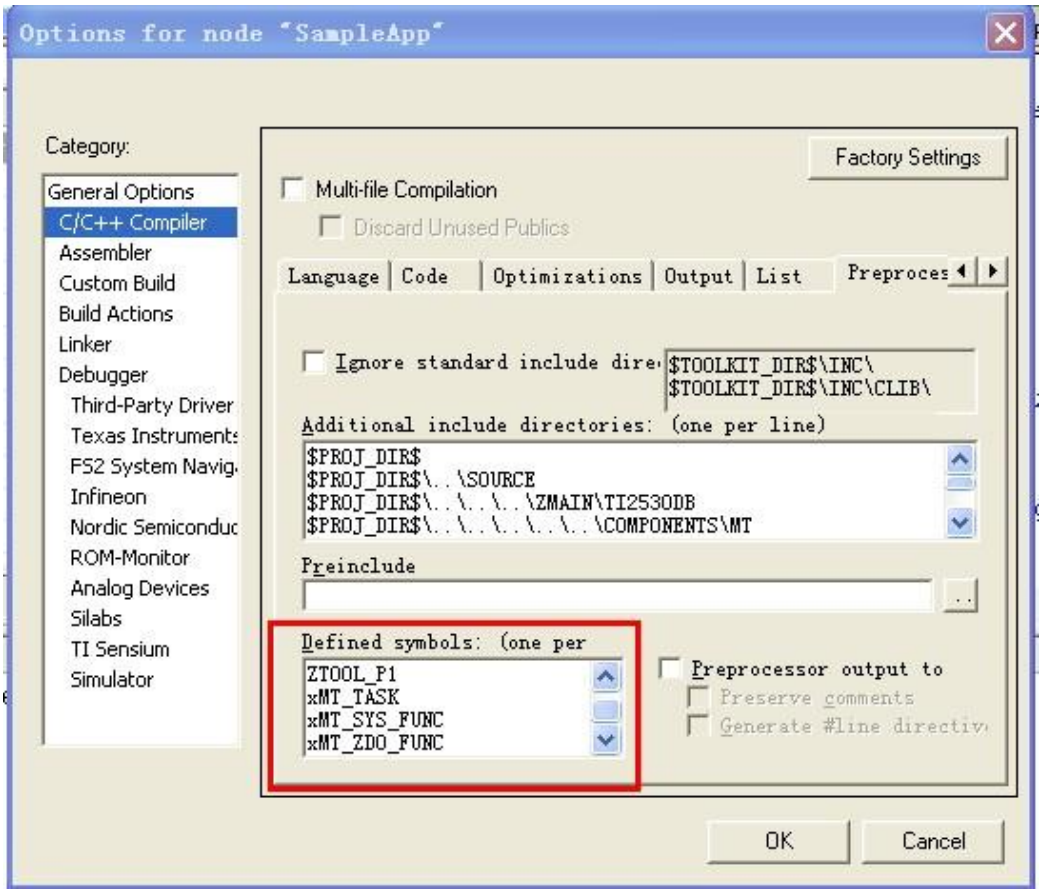


图 3.35

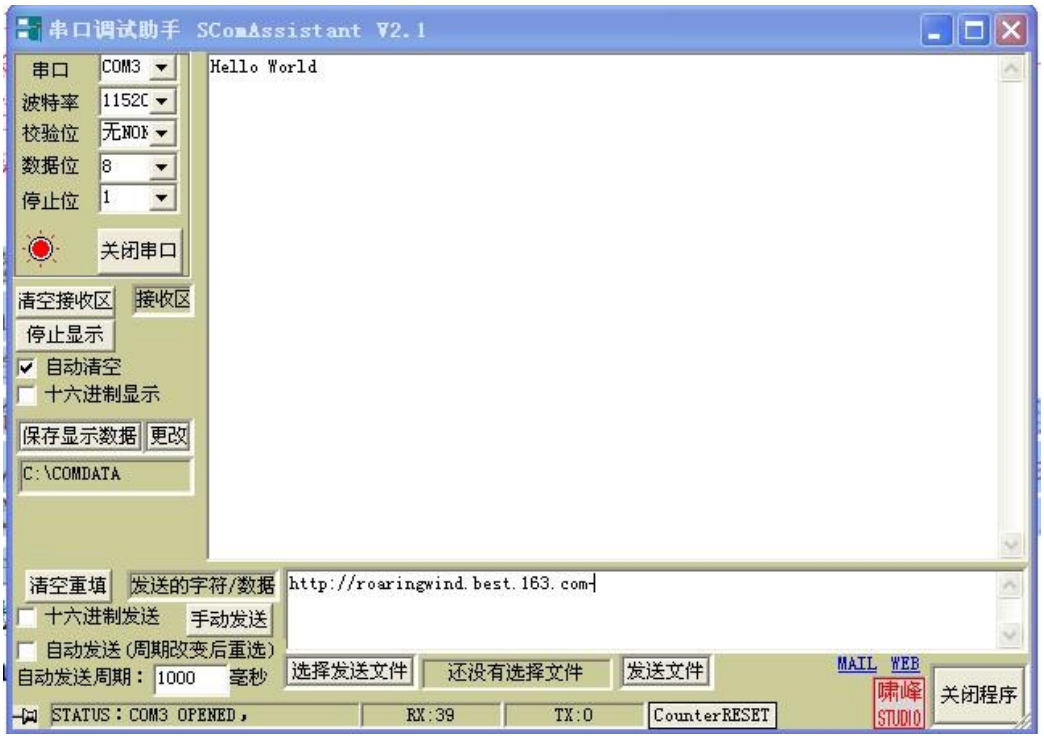


图 3.36 接收到我们想要的信息

拓展：

我们在协议栈里再做一个测试，在 osal_start_system（）函数里 for(;;)里加入：HalUARTWrite(0,”Hello,WeBee\n”,12);如图 3.37，下载运行后发现串口不停地接收到 Hello WeBee，如图 3.38 所示。这就证明了前一节的协议栈运行后会在这个函数里不停地循环查询任务、执行任务。

大家注意啦，这只是一个演示用的方法，实际应用中你可千万不能有把串口发送函数弄到这个位置然后给 PC 发信息的想法，因为这破坏了协议栈任务轮询的工作原则，相当于我们普通单片机不停用 Delay 延时函数一样，是极其低效的。

通过这个串口实验是不是让你爽了一吧？没错，协议栈我们只要掌握方法，就可以轻松应用。希望网蜂以后的教程可以让你更爽吧。

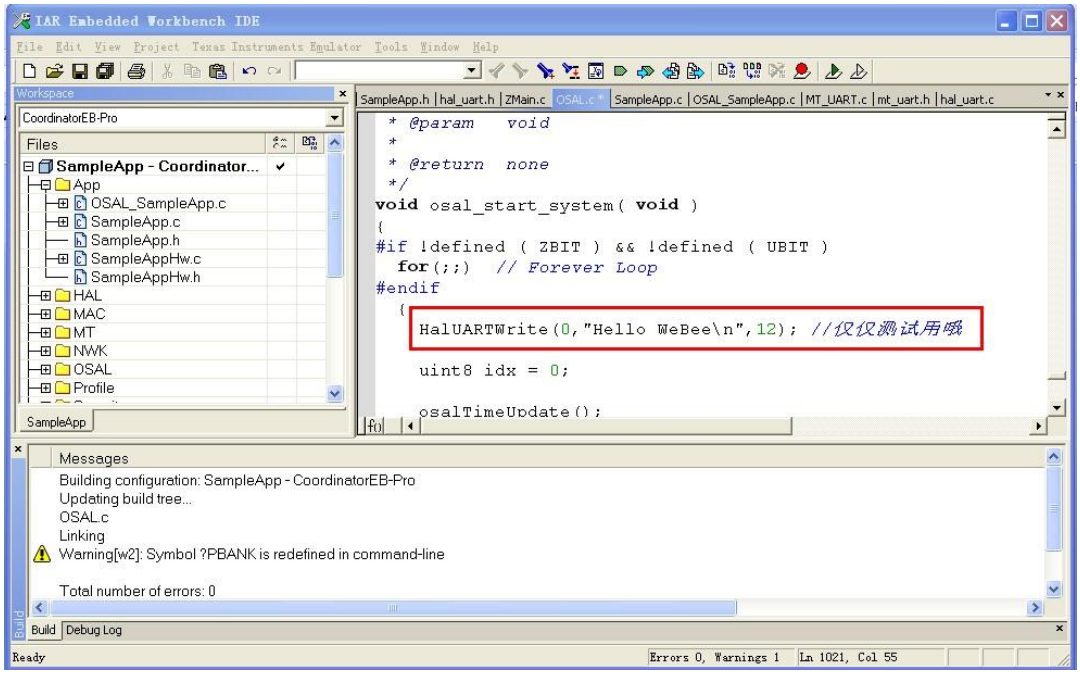


图 3.37 串口打印代码

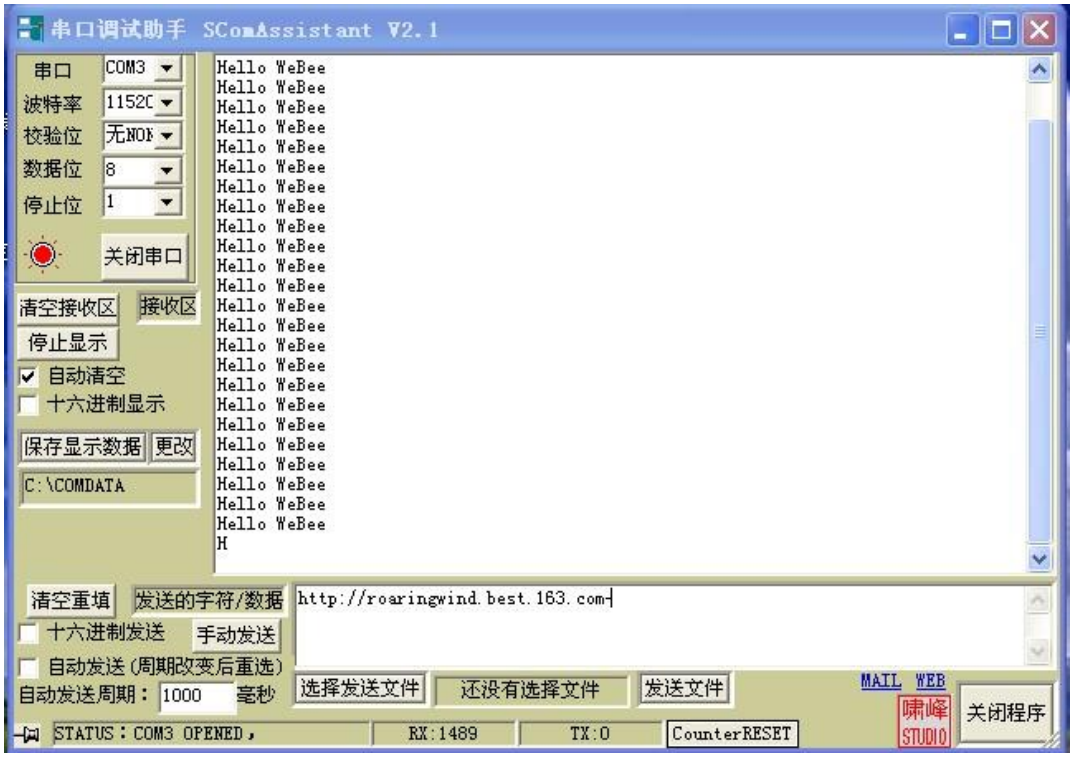


图 3.38 串口提示

3.6 协议栈中的按键实验

前言：TI 的 zigbee 协议栈 Z-stack 是针对 TI 官方套件的。所以按键的触发也不例外，到了国内，与我们设计的电路差别还是很大的。所以网蜂决定从我们自己的套件出发，打造出协议栈的按键驱动程序，通过这章的学习，大家就能将按键引脚改到自己的 IO 口上，what's more ,可以应用到自己的电路上。

实现平台：网蜂 ZigBee 节点。

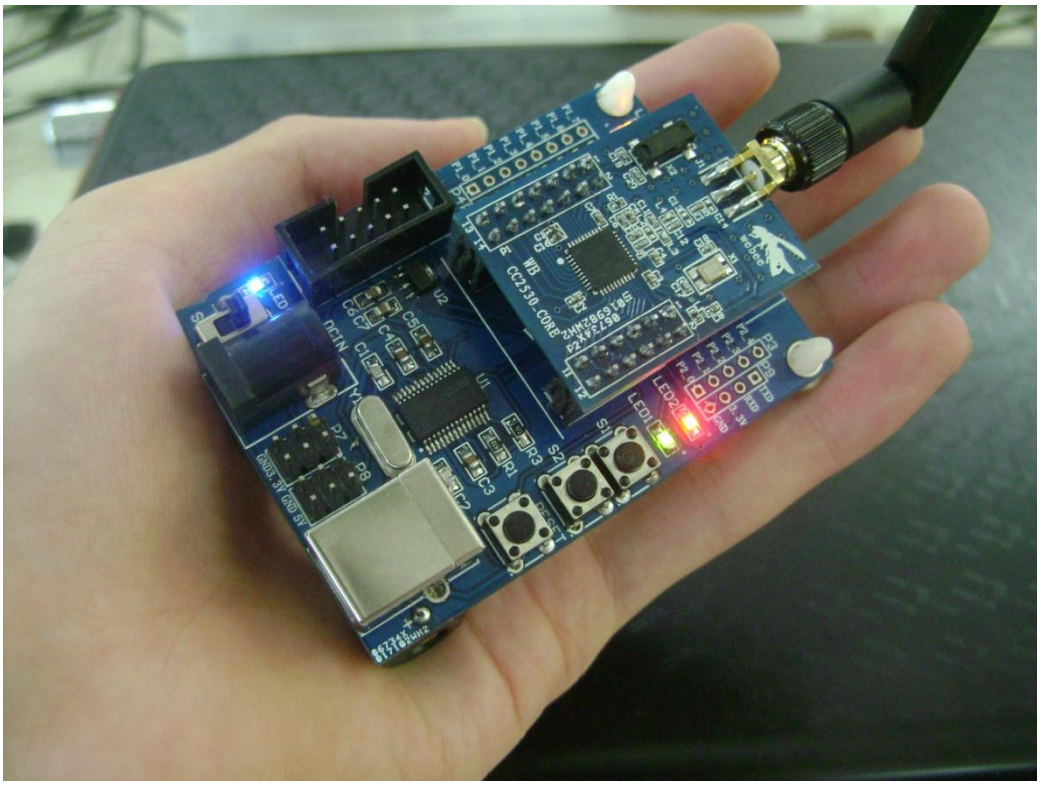


图 3.39 网蜂 ZigBee 节点

实验现象：通过节点 1 的按键 S1 中断配置，检测按键的按下情况。整个过程在协议栈 Z-STACK 的 SampleApp.eww 上完成。

实验讲解：

实验依然使用我们熟悉的 SampleApp.eww 工程来进行。看起来像一个不容易的工作，但是协议栈已经自带了按键的驱动和使用函数。所以将按键改到任

意 IO 口也不是问题了。

所谓知己知彼，百战百胜。首先我们必须了解协议栈和官方学习板的设计原理。官方的板子上有普通按键和 J-STICK 摇杆。摇杆在国内学习板比较少用，我们不用管，我们需要做的花精力将官方自带的按键 IO 改到我们学习板的 IO 口上。官方电路的按键 S1 连接的是 P0.1 引脚，网蜂 ZigBee 开发套件按键 S1 连接的是 P0.4。

打开网蜂配套例程 SampleApp 文件夹下的工程，下面我们就马上开始我们的修改工作。

第一步：修改 hal_key.C 文件.

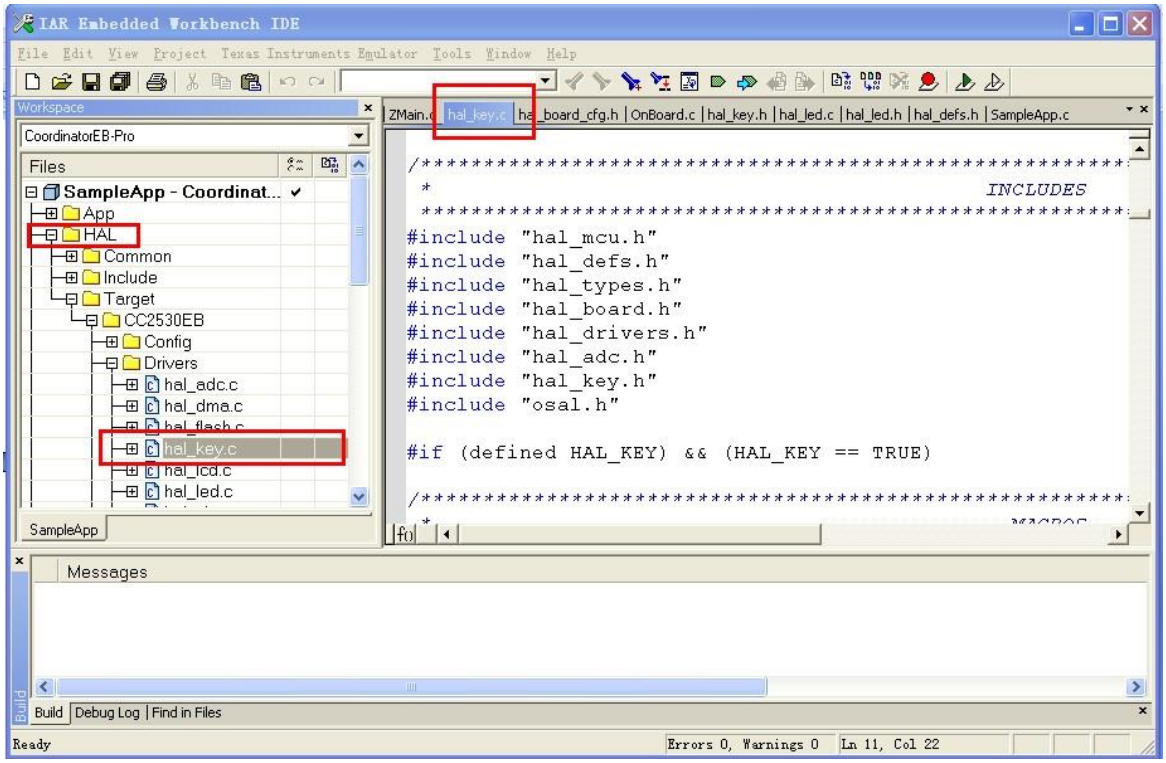


图 3.40

1、修改 SW_6 所在 IO 口

```
/* SW_6 is at P0.4 */
#define HAL_KEY_SW_6_PORT    P0
#define HAL_KEY_SW_6_BIT     BV(4) //BV(1)  改到 P0.4
#define HAL_KEY_SW_6_SEL     P0SEL
#define HAL_KEY_SW_6_DIR     P0DIR
```

2、边缘触发方式

```
/* edge interrupt */
#define HAL_KEY_SW_6_EDGEBIT   BV(0)
#define
HAL_KEY_SW_6_EDGE
HAL_KEY_RISING_EDGE//HAL_KEY_FALLING_EDGE  改成上升缘触发
```

3、中断一些相关标志位

```
/* SW_6 interrupts */
#define HAL_KEY_SW_6_IEN        IEN1   /* CPU interrupt mask register */
#define HAL_KEY_SW_6_IENBIT     BV(5) /* Mask bit for all of Port_0 */
#define HAL_KEY_SW_6_ICTL       P0IEN /* Port Interrupt Control register */
#define HAL_KEY_SW_6_ICTLBIT BV(4) //BV(1) /* P0IEN – P0.1 enable/disable
bit  改到 P0.4*/
#define HAL_KEY_SW_6_PXIFG      P0IFG /* Interrupt flag at source */
```

我们不需要用到 TI 的摇杆 J-STICK，所以把代码注释掉。如图 3.41 所示：

```
void HalKeyPoll (void)
{
  uint8 keys = 0;
```

```
if ((HAL_KEY_JOY_MOVE_PORT & HAL_KEY_JOY_MOVE_BIT))  /* Key is active HIGH */
{
  //keys = halGetJoyKeyInput();
}
```

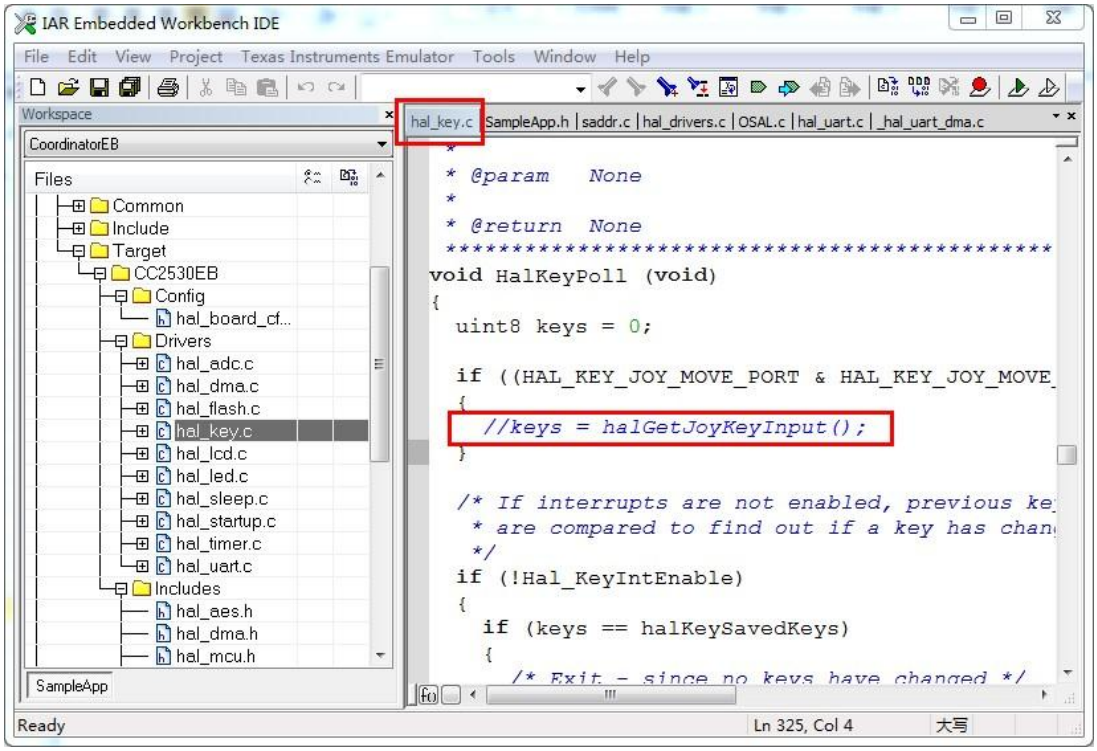


图 3.41 注释掉 TI 摇杆代码

第二步：修改 hal_board_cfg.h 文件。

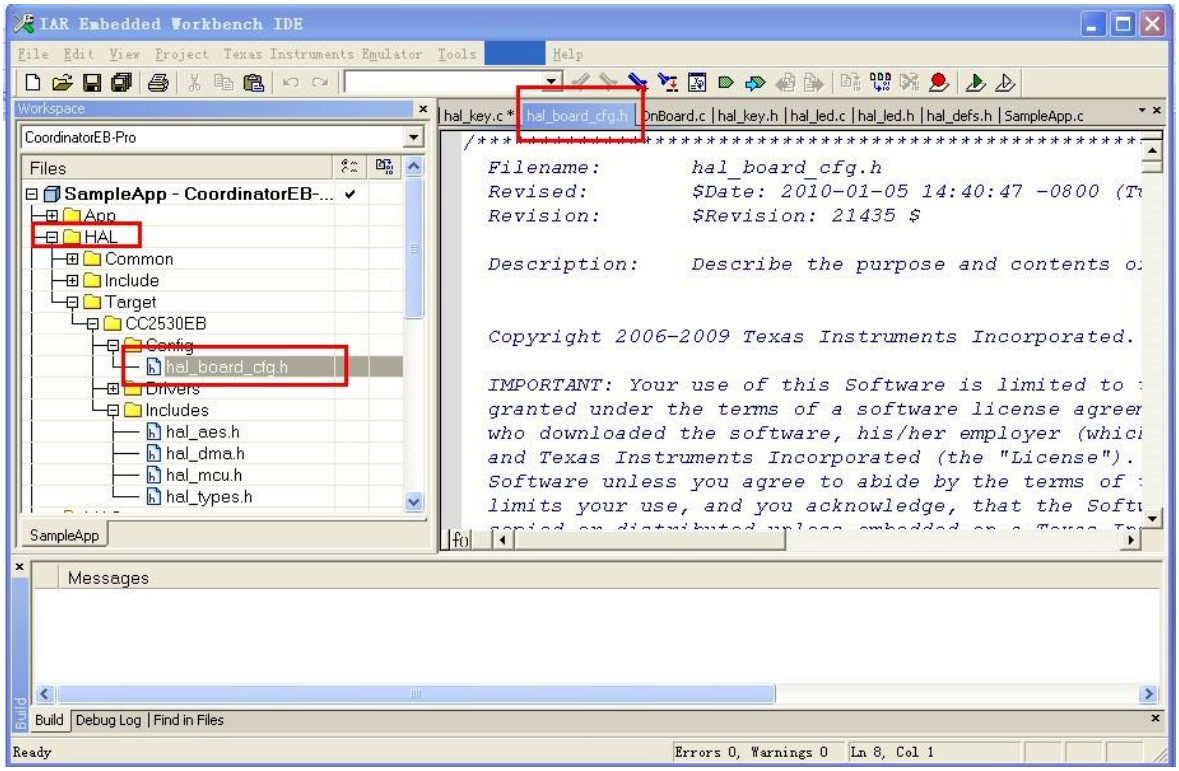


图 3.42

1、修改 SW_6 所在 IO 口

```
/* S1 */
#define PUSH1_BV      BV(4)   //BV(1)
#define PUSH1_SBIT    P0_4    //P0_1
```

第三步：修改 OnBoard.C 文件。在 ZMain.C 目录树下，如图 3.43 所示：

2、使能中断

```
HalKeyConfig(HAL_KEY_INTERRUPT_ENABLE, OnBoard_KeyCallback);
```

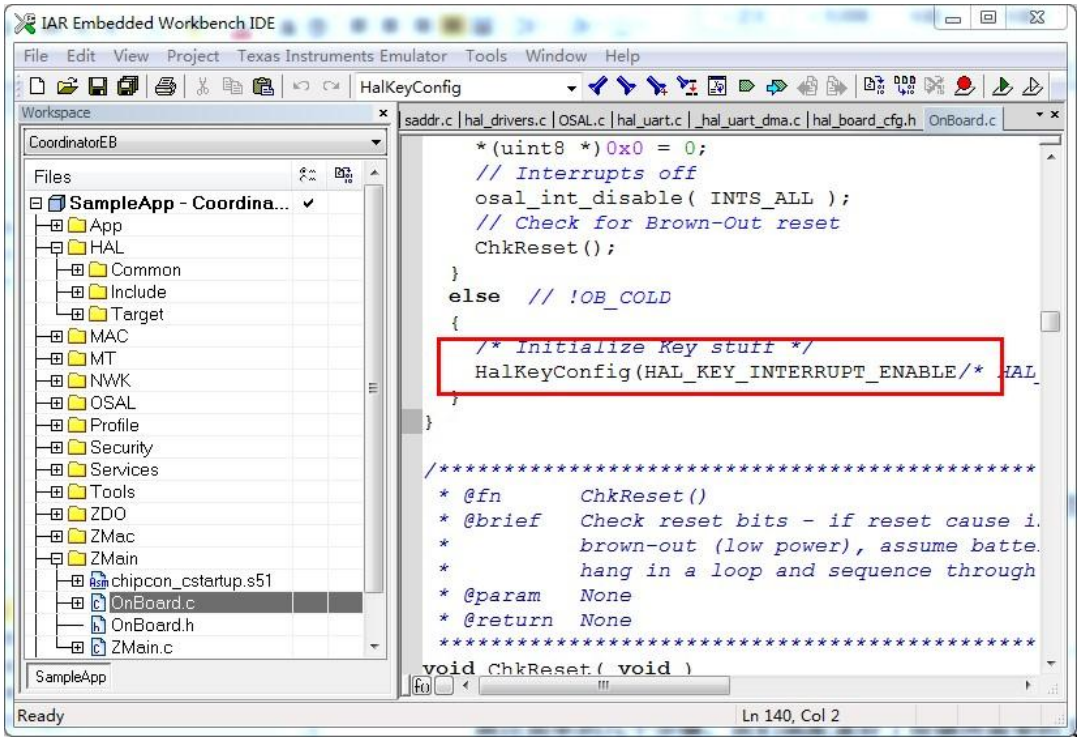


图 3.43 允许按键中断

通过简单的几个步骤，我们就配置好了按键所需要的文件。下面我们来看看协议栈是检测到按键按下时候是如何处理的，16 位必须只占 1 位，所以只能 16 个任务。

我们回到熟悉的 SampleApp.c 文件，找到按键时间处理 KEY_CHANGE 事件的函数：

```
// Received when a key is pressed
case KEY_CHANGE:
  SampleApp_HandleKeys(((keyChange_t*)MSGpkt)->state,
                       ((keyChange_t *)MSGpkt)->keys );
break;
```

当按键按下时，就会进入上面事件，我们加入串口提示：

```
// Received when a key is pressed
case KEY_CHANGE:
   HalUARTWrite(0,”KEY  “,4);//串口提示
   SampleApp_HandleKeys(          ((keyChange_t*)MSGpkt)->state,
                                  ((keyChange_t *)MSGpkt)->keys );
break;
```

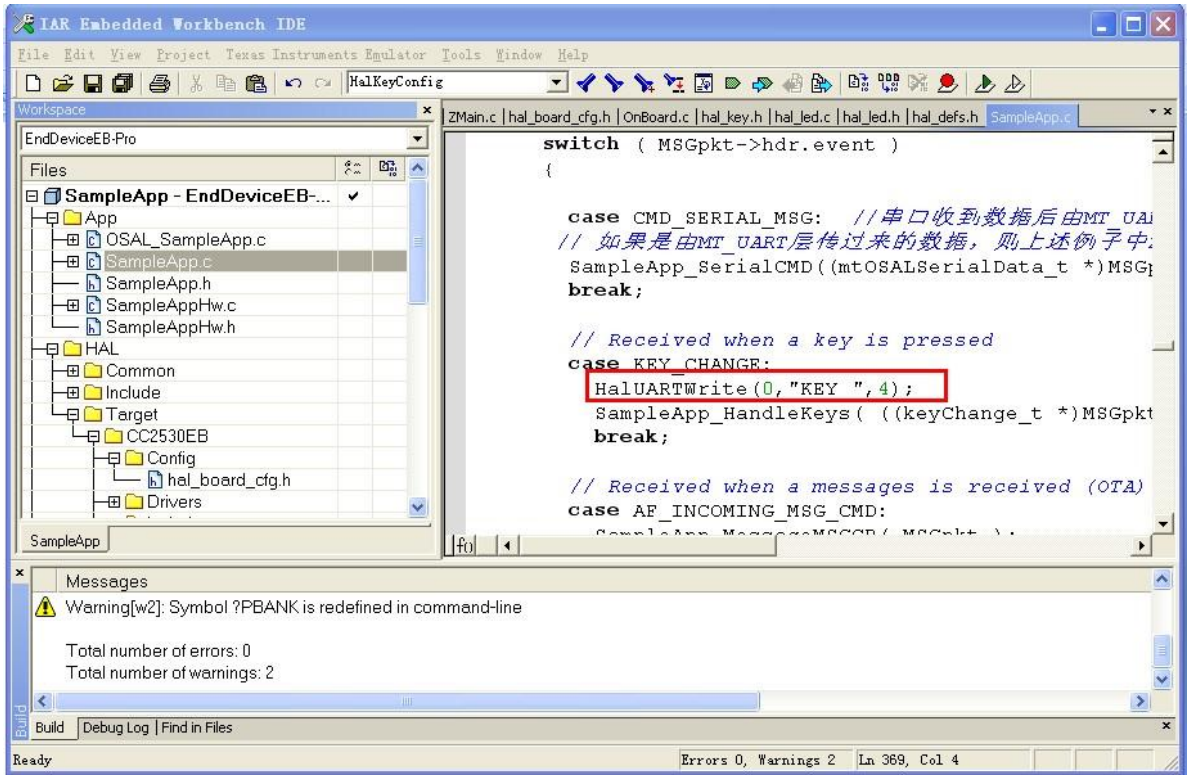


图 3.44 加入按键下提示代码

进入 SampleApp_HandleKeys（）函数，加入我们的按键处理函数。这里是 SW_6，也即是我们刚定义好的开发板上的 S1。

```
if ( keys & HAL_KEY_SW_6 )
{
  HalUARTWrite(0,”K1  “,3);                    //提示被按下的是 KEY1
  HalLedBlink( HAL_LED_1, 2,50, 500 ); //LED1 闪烁提示
}
```

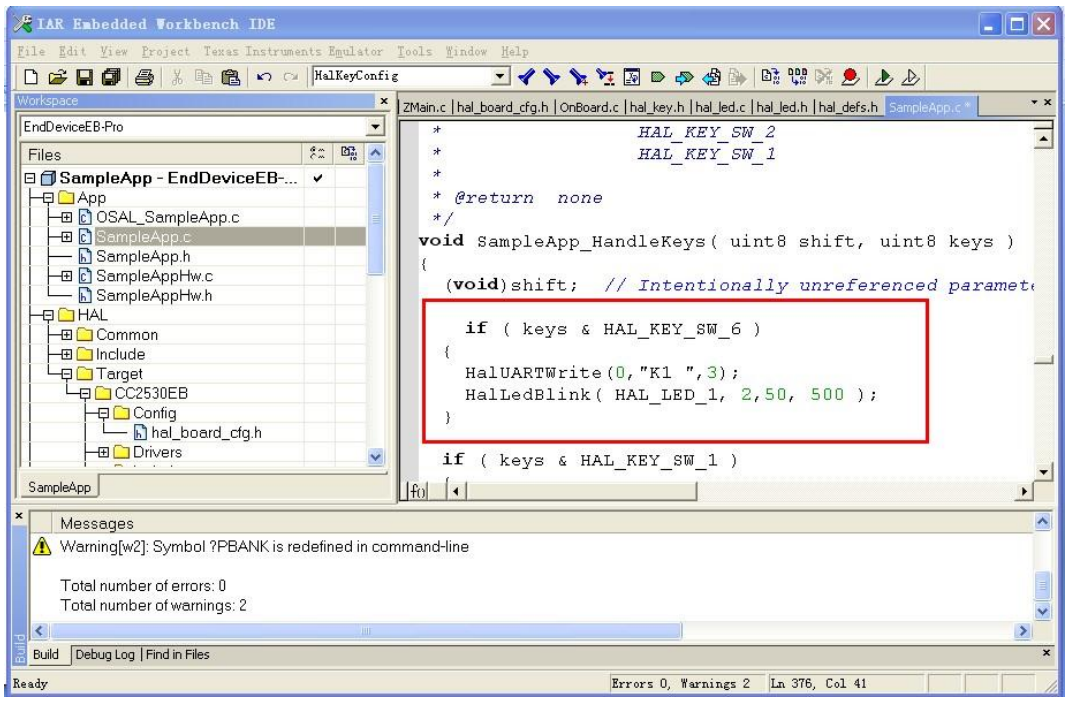


图 3.45 添加 S1 对应按键编号代码

我们下载程序到开发板。打开串口调试助手，当按下按键 S1 时候。可以看到有提示信息打印出来。

图 3.46 按键提示

到这里，网蜂开发板的按键配置完成。是不是很有成就感？通过这章的学习，我们甚至可以将按键改到我们想要的 IO 口，同时希望大家能对协议栈有一个新的认识，希望网蜂的付出能让大家有所收获。

3.7 一小时实现无线数据传输

前言：数据的无线传输，或许是每一个学习 ZigBee 的人最想得到的东西，他承载着大家的梦想，但是眼见那代码的海洋，无时无刻不在打转，更多的是迷茫啊。我们这个例程的目的就是要为大家理清思路，一步一步实验数据传输，前提示你需要有前面章节的基础。网蜂岂能让大家失望，开始计时吧！

实现平台：WeBee CC2530 模块及功能底板各两块（一个协调器，一个终端）

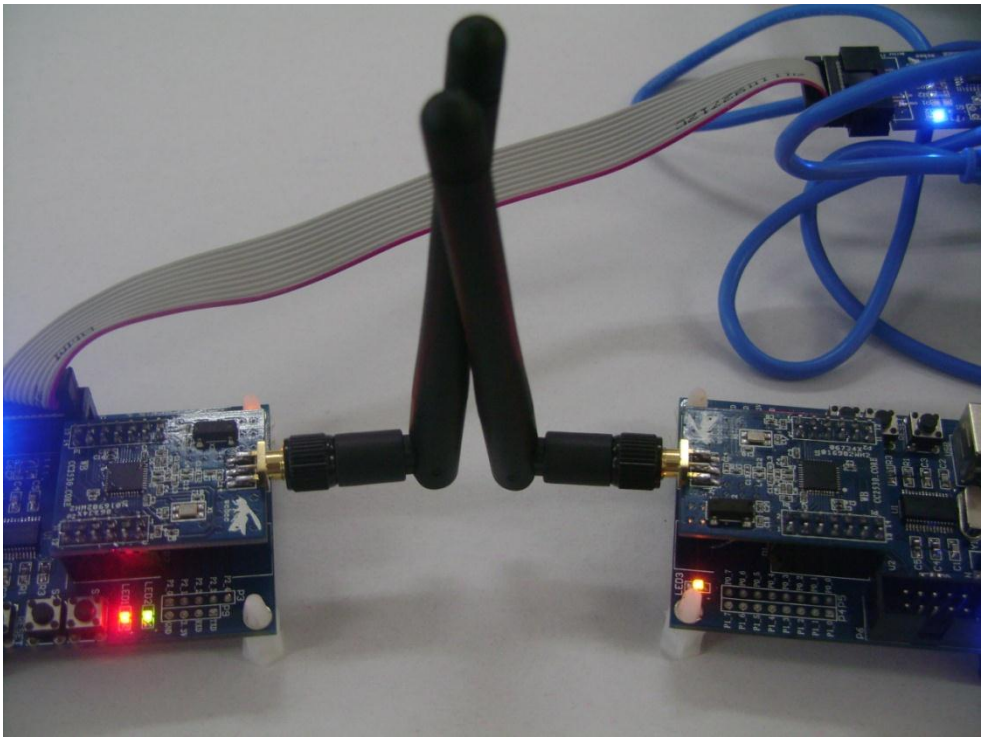


图 3.47 网蜂 zigbee 开发板

实验现象：终端节点将数据 0123456789 无线发送到协调器，协调器通过串口发送给 PC 上位机显示出来。实验基于 SampleApp 工程进行。

实验讲解：

整个例程很简单分两部分，发送部分和接收部分。每部分内容都是经过精心挑选，目的也很简单，为了大家能更容易上手。

为了方便完成我们的实验，我们用例程的周期性广播来达到我们的目的。在以前的实验我们通常是先介绍代码然后再看实验效果。今次我们用另一种方法来实验。先测试效果！只需要在代码加入 1 条语句。

我们打开 Z-stack 目录 Projects\zstack\Samples\SampleApp test\CC2530DB 里面的 SampleApp.eww 工程。这次实验我们基于协议栈的；SampleApp 来进行。

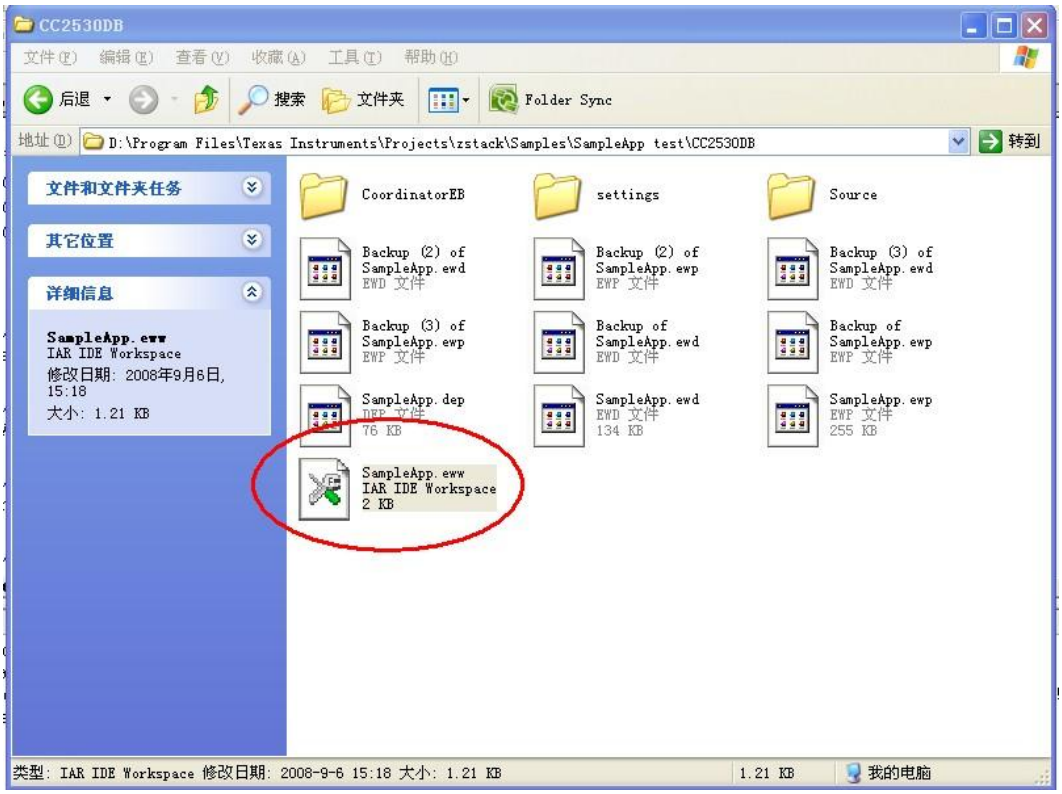


图 3.48

打开工程后，我们可以看到上一节说到 workspace 目录下比较重要的两个文件夹，Zmain 和 App。这里我们主要用到 App，这也是用户自己添加自己代码的地方。主要在 SampleApp.c 和 SampleApp.h 中就可以了。

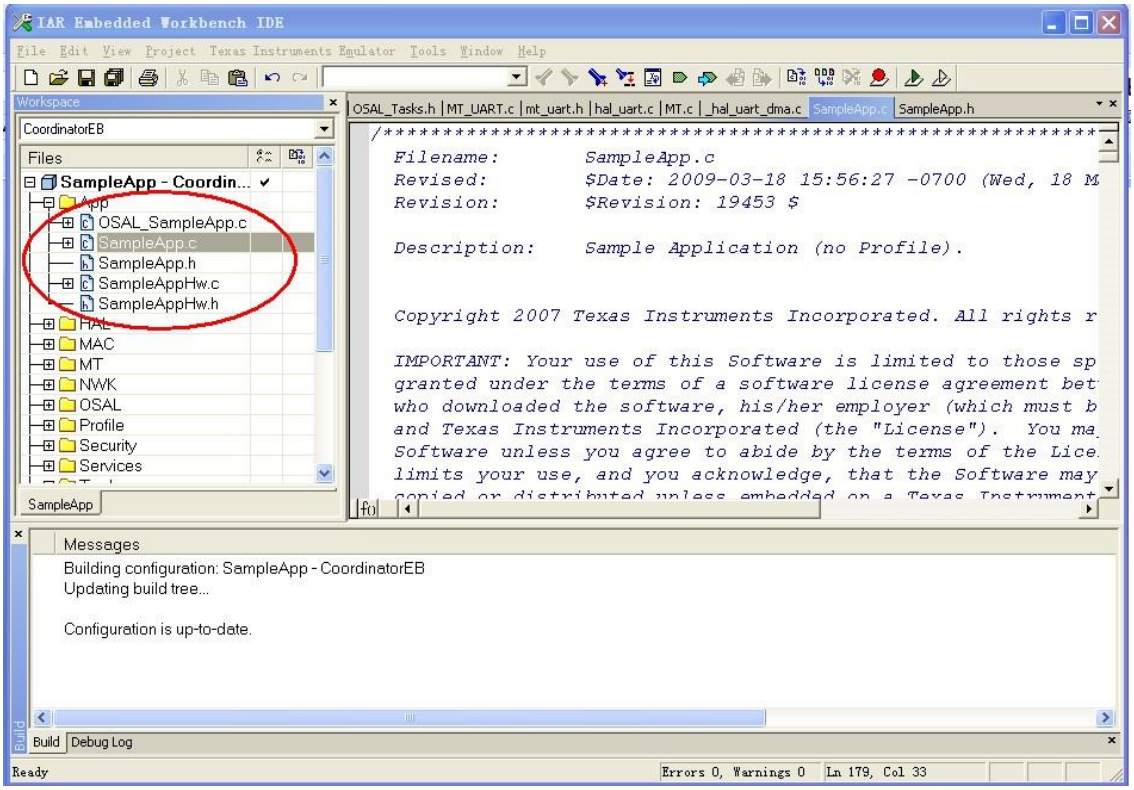


图 3.49

我们打开 SampleApp.C 文件，搜索找到函数：

void SampleApp_MessageMSGCB(afIncomingMSGPacket_t *pkt)

在 **case SAMPLEAPP_PERIODIC_CLUSTERID:**下面加入

HalUARTWrite(0,"I get data\n",11);

前提是代码已经添加了串口初始化等设置，在上一节协议栈串口中的内容。这里不再重复了。

1、选择 CoodinatorEB，下载到开发板 1；（作为协调器串口跟电脑连接）

如图 3.50 所示：

2、选择 EndDeviceEB，下载到开发板 2；（作为终端设备无线发送数据给协调器），

如图 3.51 所示：

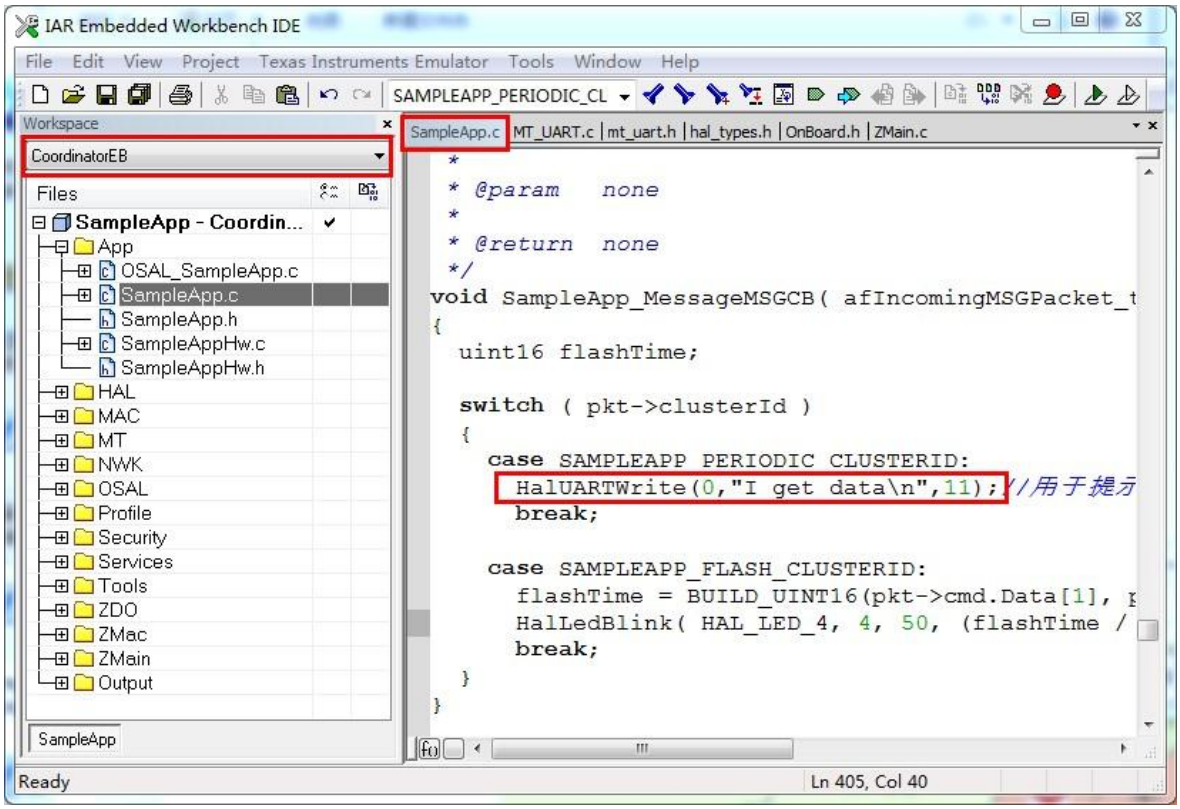


图 3.50 选择 CoodinatorEB—协调器

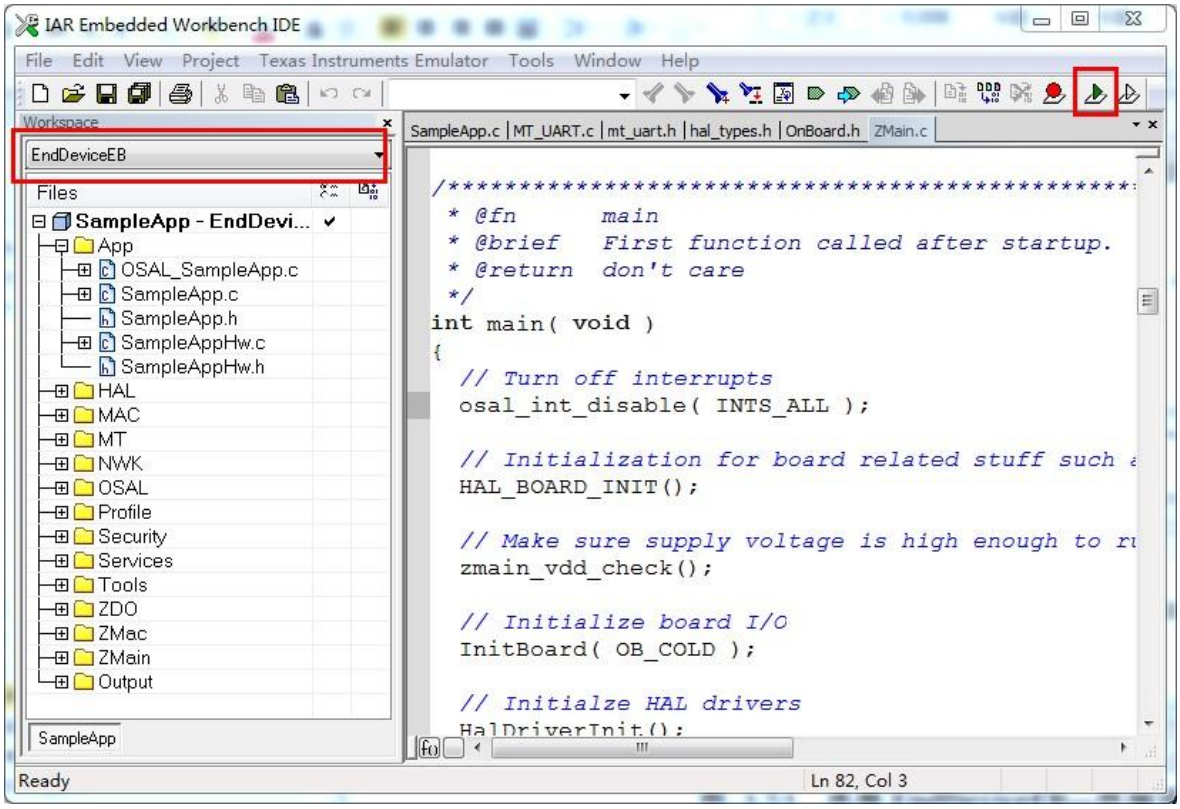


图 3.51 选择 EndDeviceEB—终端设备

给两块开发板上电，打开串口调试助手，可以看到大约 5S 中会收到 I get data 的内容，如图 3.52 所示。

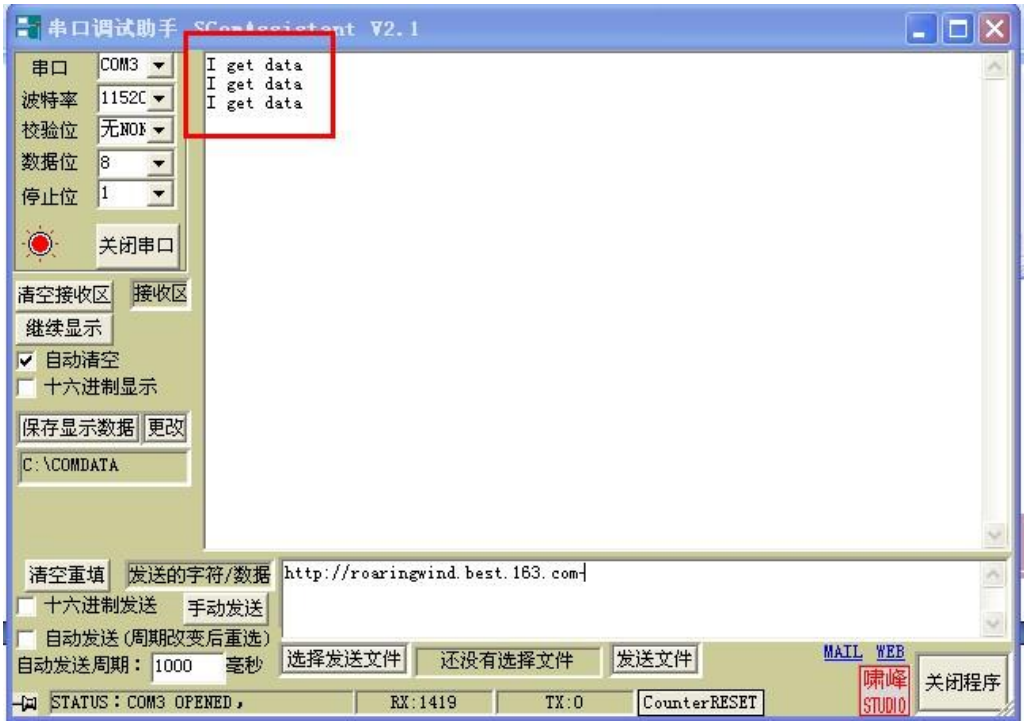


图 3.52 协调器串口周期性打印信息

到这里，会出现 2 种读者，会有下面想法：

第一种：“好厉害哦，这么快就接收到信息啦！”

第二种：“不是吧，怎么知道你是不是设置了协调器串口周期性发送给 PC 机啊”

对于第二种想法很简单，你把发射板关掉就知道了！

第三种读者想法还没想到~有的可以联系我们~

下面是详细解释：

发送部分：（可以将这部分内容理解成是发送终端需要执行的）

1、登记事件，设置编号、发送时间等。

打开 SampleApp.C 文件，找到 SampleApp 事件处理函数：

uint16 SampleApp_ProcessEvent(uint8 task_id, uint16 events)

找到函数里下面代码：

…

…

```
1、 // Received whenever the device changes state in the network
2、 case ZDO_STATE_CHANGE:     //当网络状态改变，如从未连上到连上网络
3、      SampleApp_NwkState = (185lustered185_t)(MSGpkt->hdr.status);
4、      if ( (SampleApp_NwkState == DEV_ZB_COORD     //协调器、路由器、
5、      | | (SampleApp_NwkState == DEV_ROUTER)         //或者终端都执行
6、      | | (SampleApp_NwkState == DEV_END_DEVICE) )
7、 {
8、      // Start sending the periodic message in a regular interval.
9、 osal_start_timerEx(SampleApp_TaskID,
    SAMPLEAPP_SEND_PERIODIC_MSG_EVT,
    SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT ); }
10、      else
11、      {
12、      // Device is no longer in the network
13、      }
14、      break;
```

第 9 行：代码的关键部分。这三个参数决定着周期性发送数据的命脉。我们逐一分析。分别看它们的定义。

SampleApp_TaskID：

任务 ID,函数开头定义了 SampleApp_TaskID = task_id;也就是 SampleApp 初始化的任务 ID 号。

SAMPLEAPP_SEND_PERIODIC_MSG_EVT：

// Application Events (OSAL) – These are bit weighted definitions.

```
#define SAMPLEAPP_SEND_PERIODIC_MSG_EVT       0x0001
```

同一个任务下可以有多个事件，这个是事件的号码。我们可以定义自己的事件，但是编号不能重复。而且事件号码 16 位必须只占 1 位，所以只能定义 16 个事件。

SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT：

```
// Send Message Timeout       Every 5 seconds
#define SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT     5000
```

事件重复执行的时间。这里以毫秒为单位，所以是 5s,也就是刚刚实验为什么隔约 5s 收到数据的原因。这里可以改你需要发送数据的时间间隔。

登记好事件后，看第二行代码可以知道如果网络一直连接的就不会再次进入这个函数了，所以这个相当于初始化，只执行 1 次。

设置发送内容。自动周期性地发送

在同一个函数下面可以找到如下代码：

```
1、 // Send a message out – This event is generated by a timer
2、 //    (setup in SampleApp_Init()).
3、 if ( events & SAMPLEAPP_SEND_PERIODIC_MSG_EVT )
4、 {
5、 // Send the periodic message
6、 SampleApp_SendPeriodicMessage();
7、 // Setup to send message again in normal period (+ a little jitter)
8、 osal_start_timerEx(                                SampleApp_TaskID,
    SAMPLEAPP_SEND_PERIODIC_MSG_EVT,
9、 (SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT + (osal_rand() & 0x00FF)) );
10、    // return unprocessed events
11、    return (events ^ SAMPLEAPP_SEND_PERIODIC_MSG_EVT);
12、    }
```

第 3 行：判断 SAMPLEAPP_SEND_PERIODIC_MSG_EVT (0x0001)有没有发生，如果有的就执行下面函数。

第 6 行：SampleApp_SendPeriodicMessage();是主要的代码，是我们编写需要发送内容的地方，我们进入去做一些修改。先看源代码。我们主要关心蓝色部分的内容：

```
/*********************周期性发送数据函数*************************/
1、 void SampleApp_SendPeriodicMessage( void )
2、 {
3、 if(AF_DataRequest(&SampleApp_Periodic_DstAddr, &SampleApp_epDesc,
4、                        SAMPLEAPP_PERIODIC_CLUSTERID,
5、                        1,
6、                        (uint8*)&SampleAppPeriodicCounter,
7、                        &SampleApp_TransID,
8、                        AF_DISCV_ROUTE,
9、                        AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
10、                      {
11、                      }
12、     else
13、     {
14、     // Error occurred in request to send.
15、     }
16、     }
```

第 4 行：SAMPLEAPP_PERIODIC_CLUSTERID 这是一个新东西，其定义为：

#define SAMPLEAPP_PERIODIC_CLUSTERID 1

定义的作用是和接收方建立联系，协调器收到这个标号，如果是 1，就证明是由周期性广播方式发送过来的。

第 5 行：1 是数据长度。

第 6 行：(uint8*)&SampleAppPeriodicCounter 是要发送的内容。

知道代码功能后我们可以做以下修改，红色部分为修改内容。添加数据 0~9，发送字符数 10 个，内容是 data 数组。

```
/**********************周期性发送数据函数*************************/
1、 void SampleApp_SendPeriodicMessage( void )
2、 {
3、    uint8 data[10]={'0','1','2','3','4','5','6','7','8','9'};
4、 if(AF_DataRequest(&SampleApp_Periodic_DstAddr, &SampleApp_epDesc,
5、                        SAMPLEAPP_PERIODIC_CLUSTERID,
6、                        10,
7、                        data,     //指针方式
8、                        &SampleApp_TransID,
9、                        AF_DISCV_ROUTE,
                          AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
10、     {
11、     }
12、     else
13、     {
14、     // Error occurred in request to send.
15、     }
16、     }
```

至此，发送部分代码修改完成，上电后 CC2530 会以周期 5s 来广播式发送数据 0~9。

接收部分：（可以将这部分内容理解成是连接电脑的协调器需要执行的）

接收部分需要完成 2 个任务:1、读取接收到的数据；

a) 把数据通过串口发送给 PC 机。

b) 读取接收到的数据；

同样的在 uint16 SampleApp_ProcessEvent(uint8 task_id, uint16 events) 事件处理函数中找到代码：

```
// Received when a messages is received (OTA) for this endpoint
  case AF_INCOMING_MSG_CMD:
      SampleApp_MessageMSGCB( MSGpkt );
  break;
```

其中 SampleApp_MessageMSGCB(MSGpkt);就是将接收到的数据包进行处理的函数。我们进入此函数，代码如下：

```
1、 void SampleApp_MessageMSGCB( afIncomingMSGPacket_t *pkt )
2、 {
3、    uint16 flashTime;
4、    switch ( pkt->189lustered )
5、 {
6、      case SAMPLEAPP_PERIODIC_CLUSTERID:
7、        HalUARTWrite(0,"I get data\n",11); //提示收到数据
8、      break;
9、
10、    case SAMPLEAPP_FLASH_CLUSTERID:
11、      flashTime = BUILD_UINT16(pkt->cmd.Data[1], pkt->cmd.Data[2] );
12、      HalLedBlink( HAL_LED_4, 4, 50, (flashTime / 4) );
13、    break;
    }
```

```
}
```

我们只关注蓝色部分代码，其中红色部分为自己添加内容。

第 6 行：读取发来的数据包的 ID 号，如果是：

SAMPLEAPP_PERIODIC_CLUSTERID 就执行里面函数，

这个编号是不是有点眼熟？没错，这就是我们前面发送定义的数据包的 ID 号，它的作用很明显了，如果收到是这个的话说明是我们自己定义的周期性广播。

所有的数据和信息都在函数传入来的 afIncomingMSGPacket_t *pkt 里面，进入 afIncomingMSGPacket_t 的定义，它是一个结构体，内容如下：

```
typedef struct
{
  osal_event_hdr_t hdr;          /* OSAL Message header */
  uint16 groupId;               /* Message's group ID – 0 if not set */
  uint16 clusterId;             /* Message's cluster ID */
  afAddrType_t    srcAddr;              /*Source Address, if endpoint is
                                        STUBAPS_INTER_PAN_EP,
                                        it's an InterPAN message */
  uint16 macDestAddr;          /* MAC header destination short address */
  uint8 endPoint;               /* destination endpoint */
  uint8 wasBroadcast;           /* TRUE if network destination was a broadcast
                                   address */
  uint8 LinkQuality;           /* The link quality of the received data frame */
  uint8 correlation;            /* The raw correlation value of the received data
                                   frame */
  int8    rssi;                /* The received RF power in units dBm */
  uint8 SecurityUse;           /* deprecated */
```

```
  uint32 timestamp;                    /* receipt timestamp from MAC */
  afMSGCommandFormat_t cmd;     /* Application Data */
} afIncomingMSGPacket_t;
```

里面包含了数据包的所有东西，长地址、短地址、RSSI 等，自己翻译一下了。

那么数据在哪里呢？在蓝色部分，又是一个结构体，继续进入。

```
// Generalized MSG Command Format
typedef struct
{
  byte     TransSeqNumber;
  uint16 DataLength;                    // Number of bytes in TransData
  byte    *Data;
} afMSGCommandFormat_t;
```

2、把数据通过串口发送给 PC 机

千呼万唤始出来了，我们现在就可以通过一个简单的语句将数据读出了。

下面是一个读取的方法，大家可以参考一下。

```
1、void SampleApp_MessageMSGCB( afIncomingMSGPacket_t *pkt )
2、{
3、    uint16 flashTime;
4、    switch ( pkt->191lustered )
5、{
6、      case SAMPLEAPP_PERIODIC_CLUSTERID:
7、      HalUARTWrite(0,"I get data\n",11); //提示收到数据
8、      HalUARTWrite(0, &pkt->cmd.Data[0],10); //打印收到数据
```

```
9、      HalUARTWrite(0,"\n",1);                    // 回车换行
10、     break;
11、
12、     case SAMPLEAPP_FLASH_CLUSTERID:
13、        flashTime = BUILD_UINT16(pkt->cmd.Data[1], pkt->cmd.Data[2] );
14、        HalLedBlink( HAL_LED_4, 4, 50, (flashTime / 4) );
15、     break;
      }
    }
```

分别选择 CoordinatorEB-和 EndDeviceEB 编译后对应下载到协调器和终端模块，协调器通过串口连接到电脑。激动人心的时候即将来临，我们看到串口收到了数据，如

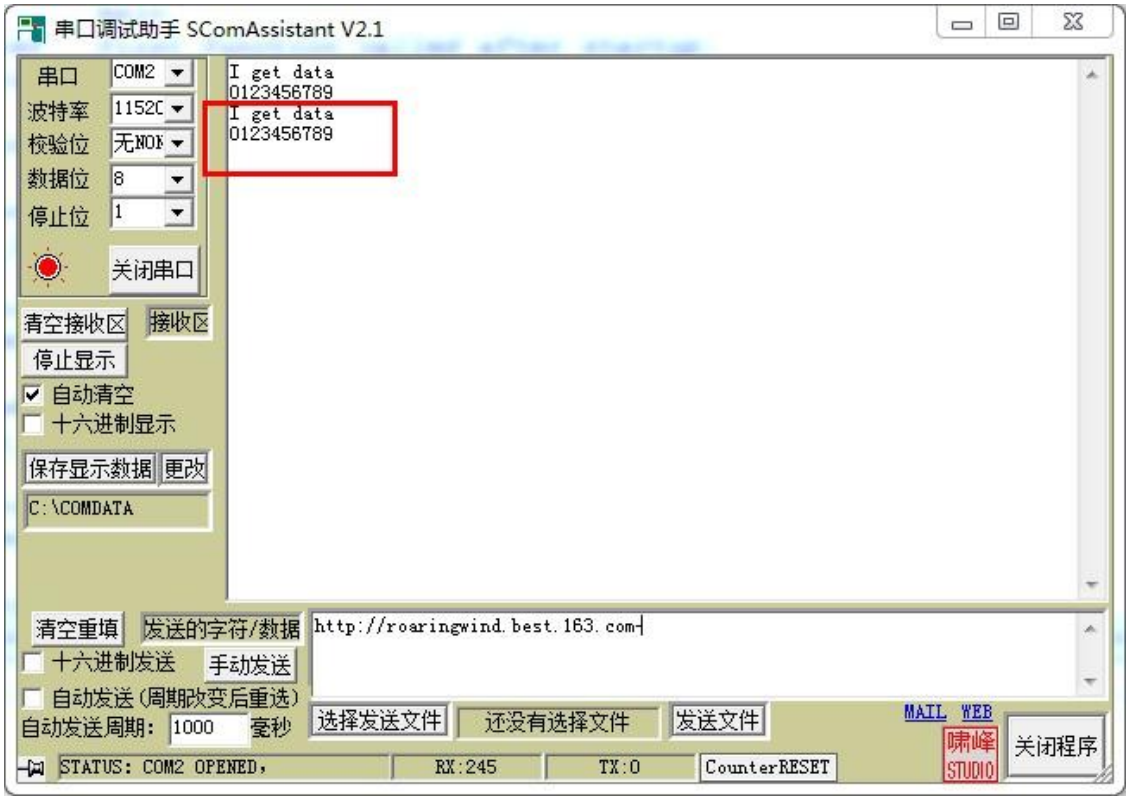


图 3.53 串口接收信息

就这样，我们实验了数据的无线传输，应该还不用一个小时。由此可以引出 N 多应用了，我们在协议栈上配置好传感器，然后把采集到的信息发送给协调器继而发给电脑，这样我们学习 ZigBee 就算用得上了。到了这里，相信你会觉得 ZigBee 学习也不是想象中的难。关键还是方法和思路。希望我们接下来的教程对大家有帮助！感谢大家关注网蜂 ZigBee。

3.8 串口透传,打造无线串口模块

前言：串口透传，这个名词相信大家在看 ZigBee 相关资料时候经常会看到，透传到底是什么呢？电脑 A 和电脑 B 通过串口相连，相互发送信息，现在我们将电脑 A 和 B 连接 ZigBee 模块，再用串口收发信息，ZigBee 的作用就相当于把有线信号转化成无线信号。这样我们电脑前面操作是一样的，但是已经变成了无线传输了，这就是串口透传！图 1 所示：

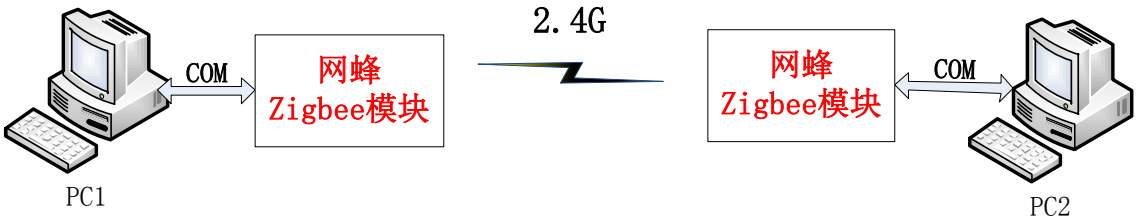


图 3.54 串口透传

网上的价格这么贵的东西原理又是怎样呢？我们能不能用自己的模块做一个呢？网蜂跟你说，完全没问题。

实现平台：WeBee CC2530 模块及功能底板各两块（一个协调器，一个终端）

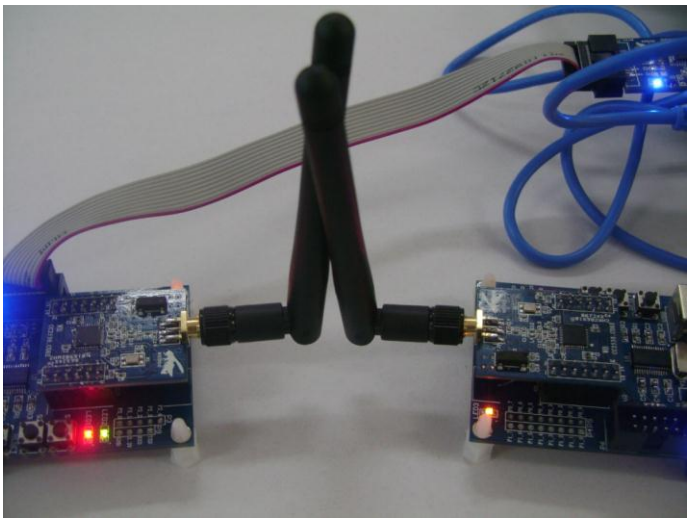


图 3.55 网蜂 zigbee 节点

实验现象：两台不同的 PC 机通过串口连接到网蜂开发板，打开串口调试助手，设置好波特率等参数。相互收发信息。没有 2 台电脑的也可以用同一台电脑的不同串口进行实验。

实验讲解：

实验依然使用我们熟悉的 SampleApp.eww 工程来进行。在前面我们曾做过串口实验和数据无线传输，这次实验也算是前面 2 个实验的一个结合。不过协议栈的串口接收有特定的格式，我们得了解一下它的传输机制。先理清我们要实现这个功能的流程：由于 2 台 PC 机所带的模块地位是相等的，所以两个模块的程序流程也一样了：

1、ZigBee 模块接收到从 PC 机发送信息，然后无线发送出去

2、ZigBee 模块接收到其它 ZigBee 模块发来的信息，然后发送给 PC 机

我们打开 Z-stack 目录 Projects\zstack\Samples\SampleApp test\CC2530DB 里面的 SampleApp.eww 工程。这次实验我们基于协议栈的；SampleApp 来进行。

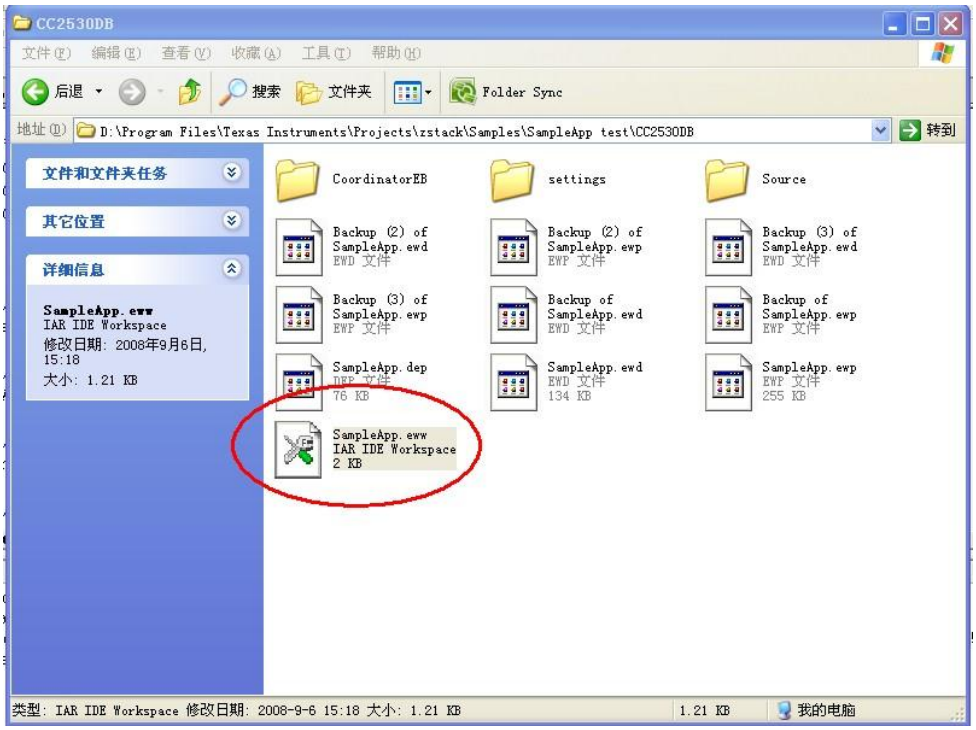


图 3.56

打开工程后，我们可以看到上一节说到 workspace 目录下比较重要的两个文件夹，Zmain 和 App。这里我们主要用到 App，这也是用户自己添加自己代码的地方。主要在 SampleApp.c 和 SampleApp.h 中就可以了。

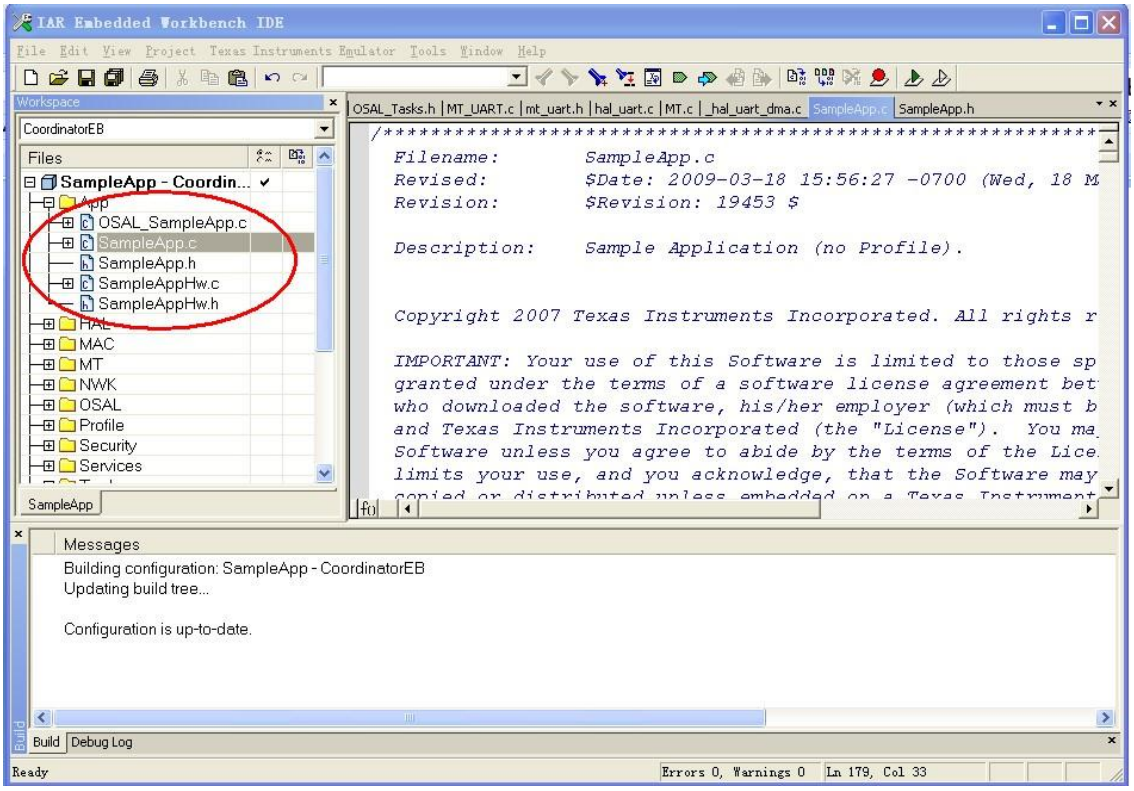


图 3.57

1、 ZigBee 模块接收到从 PC 机发送信息，然后无线发送出去

以前我们做的都是 CC2530 给 PC 机串口发信息，还没接触过 PC 机发送给 CC2530，现在我们就来完成这个任务。其主要代码在 MT_UART.C 中。我们之前协议栈串口实验对串口初始化时候已经有所了解了。

我们在这个文件里找到串口初始化函数 void MT_UartInit ()，找到下面代码：

```
#if defined (ZTOOL_P1) || defined (ZTOOL_P2)
  uartConfig.callBackFunc        = MT_UartProcessZToolData;
#elif defined (ZAPP_P1) || defined (ZAPP_P2)
  uartConfig.callBackFunc        = MT_UartProcessZAppData;
```

```
#else
  uartConfig.callBackFunc              = NULL;
#endif
```

我们定义了 ZTOOL_P1，故协议栈数据处理的函数 MT_UartProcessZToolData，进入这个函数定义。下边是对函数关键地方的解释。

```
/***************************************************************
* @fn          MT_UartProcessZToolData
*
* @brief     | SOP | Data Length   |     CMD     |     Data     |   FCS   |
*             |  1  |       1          |      2      |    0-Len     |    1    |
*
*     Parses the data and determine either is SPI or just simply serial data
*          then send the data to correct place (MT or APP)
*
* @param      port        - UART port
*             event       - Event that causes the callback
* @return    None
***************************************************************/
/* 这个函数很长，具体说来就是把串口发来的数据包进行打包，校验，生成一个消息，发给处理数据包的任务。如果你看过 MT 的文档，应该知道如果用 ZTOOL 通过串口来沟通协议栈，那么发过来的串口数据具有以下格式：
      0xFE,          DataLength, CM0, CM1, Data payload, FCS
```

翻译： 0xFE：数据帧头

DataLength：Datapayload 的数据长度，以字节计，低字节在前；

CM0：命令低字节；

CM1：命令高字节；(ZTOOL 软件就是通过发送一系列命令给 MT 实现和协议栈交互)

Data payload：数据帧具体的数据，这个长度是可变的，但是要和 DataLength 一致；

FCS ：校验和，从 DataLength 字节开始到 Data payload 最后一个字节所有字节的异或按字节操作；

也就是说，如果 PC 机想通过串口发送信息给 CC2530，由于是使用默认的串口函数，所以您必须按上面的格式发送，否则 CC2530 是收不到任何东西的，这也是我们大家在调试串口接收时一直打圈的地方。尽管这个机制是非常完善的，也能校验串口数据，但是很明显，我们需要的是 CC2530 能直接接收到串口信息，然后一成不变的发送出去，相信你在聊 QQ 的时候也不希望在每句话前面加 FE .. ..的特定字符吧，而且还要自己计算校验码。

于是我们就来个偷龙转凤，把改函数换成我们自己的串口处理函数，是不是很酷？当然，不过前提我们先要了解自带的这个函数。

```
Void MT_UartProcessZToolData ( uint8 port, uint8 event )
{
  …
  …
  while (Hal_UART_RxBufLen(port))
   /*查询缓冲区读信息,也成了这里信息是否接收完的标志*/
  {
    HalUARTRead (port, &ch, 1);
  /*一个一个地读，读完一个缓冲区就清 1 个了，？为什么这样呢，往下看*/

    switch (state)
```

```
    /*用上状态机了*/
  {
    case SOP_STATE:
      if (ch == MT_UART_SOF)   /* MT_UART_SOF 的值默认是 0xFE,所以数
                                  据必须 FE 格式开始发送才能进入下
                                  一个状态，不然永远在这里转圈*/
      state = LEN_STATE;
      break;

    case LEN_STATE:
      LEN_Token = ch;
      tempDataLen = 0;

      /* Allocate memory for the data */
    pMsg      =      (mtOSALSerialData_t      *)osal_msg_allocate(      sizeof
 ( mtOSALSerialData_t ) +MT_RPC_FRAME_HDR_SZ + LEN_Token );
 /*  分配内存空间*/

if (pMsg) /*  如果分配成功*/
      {
        /* Fill up what we can */
        pMsg->hdr.event = CMD_SERIAL_MSG;
        /*  注册事件号 CMD_SERIAL_MSG;，很有用*/
        pMsg->msg = (uint8*)(pMsg+1);
        /*定位数据位置*/
      …
      …
      …
      /* Make sure it's correct */
```

```
tmp=MT_UartCalcFCS((uint8*)&pMsg->msg[0],   MT_RPC_FRAME_HDR_SZ
     + LEN_Token);
 if (tmp == FSC_Token) /*数据校验*/
    {
       osal_msg_send( App_TaskID, (byte *)pMsg );
        /*把数据包发送到 OSAL 层，很很重要*/
    }
    else
    {
       /* deallocate the msg */
       osal_msg_deallocate ( (uint8 *)pMsg );
        /*清申请的内存空间*/

    }
    /* Reset the state, send or discard the buffers at this point */
 state = SOP_STATE; /*状态机一周期完成*/
…
…
…
```

简单看了一下代码，串口从 PC 机接收到信息会做如下处理：

1、接收串口数据，判断起始码是否为 0xFE

2、得到数据长度然后给数据包 pMsg 分配内存

3、给数据包 pMsg 装数据

4、打包成任务发给上层 OSAL 待处理

5、释放数据包内存

我们要做的是简化再简化。流程变成：

1、接收到数据

2、判断长度然后然后给数据包 pMsg 分配内存

3、打包发送给上层 OSAL 待处理

4、释放内存

网蜂独家参考程序如下：

```
1.  void MT_UartProcessZToolData ( uint8 port, uint8 event )
2.  {
3.      uint8 flag=0,i,j=0;     //flag 是判断有没有收到数据，j 记录数据长度
4.      uint8 buf[128];         //串口 buffer 最大缓冲默认是 128，我们这里用 128.
5.      (void)event;             // Intentionally unreferenced parameter

6.      while (Hal_UART_RxBufLen(port)) //检测串口数据是否接收完成

7.      {
8.       HalUARTRead (port,&buf[j], 1);   //把数据接收放到 buf 中
9.       j++;                                  //记录字符数
10.      flag=1;                               //已经从串口接收到信息
11.     }

12.     if(flag==1)           //已经从串口接收到信息

13.     {       /* Allocate memory for the data */
14.      //分配内存空间，为机构体内容+数据内容+1 个记录长度的数据
15.       pMsg = (mtOSALSerialData_t *)osal_msg_allocate( sizeof
16.                   ( mtOSALSerialData_t )+j+1);
17.      //事件号用原来的 CMD_SERIAL_MSG
```

```
18.         pMsg->hdr.event = CMD_SERIAL_MSG;
19.         pMsg->msg = (uint8*)(pMsg+1);   // 把数据定位到结构体数据部分
20.         pMsg->msg [0]= j;                //给上层的数据第一个是长度
21.         for(i=0;i<j;i++)                 //从第二个开始记录数据
22.           pMsg->msg [i+1]= buf[i];
23.         osal_msg_send( App_TaskID, (byte *)pMsg );   //登记任务，发往上层
24.          /* deallocate the msg */
25.         osal_msg_deallocate ( (uint8 *)pMsg );        //释放内存
26.     }
27. }
```

由网蜂提供的代码可以知道，数据包中数据部分的格式是：

datalen + data

到这里，数据接收的处理函数已经完成了，接下来我们要做的就是怎么在任务中处理这个包内容呢？很简单，因为串口初始化是在 SampleApp 中进行的，任务号也是 SampleApp 的 ID，所以当然是在 SampleApp.C 里面进行了。在 SampleApp.C 找到任务处理函数：

uint16 SampleApp_ProcessEvent(uint8 task_id, uint16 events)，加入下面红色代码：

```
uint16 SampleApp_ProcessEvent( uint8 task_id, uint16 events )
{
  afIncomingMSGPacket_t *MSGpkt;
  (void)task_id;  // Intentionally unreferenced parameter
  if ( events & SYS_EVENT_MSG )
  {

MSGpkt(afIncomingMSGPacket_t*)osal_msg_receive( SampleApp_TaskID );
  while ( MSGpkt )
```

```
{
    switch ( MSGpkt->hdr.event )
    {
      case CMD_SERIAL_MSG:  //串口收到数据后由 MT_UART 层传递过来的
                            数据，用网蜂方法接收，编译时不定义 MT
                            相关内容
      SampleApp_SerialCMD((mtOSALSerialData_t *)MSGpkt);
        break;
```

解释：串口收到信息后，事件号 CMD_SERIAL_MSG 就会被登记，便进入 case CMD_SERIAL_MSG:

执行 SampleApp_SerialCMD((mtOSALSerialData_t *)MSGpkt);大家是不是很奇怪怎么在协议栈里找不到这个函数，当然了，我们那边只把他打包了，然后登记任务，这个包是我们自己的，想怎么处理当然由自己来搞掂。大家应该想到这个函数应该要把信息无线发送出去吧，想到这个的话你的悟性还挺高的。

下面贴上网蜂的参考代码，用户也可以自己完成。

```
1.  void SampleApp_SerialCMD(mtOSALSerialData_t *cmdMsg)
    {
2.      uint8 i,len,*str=NULL;        //len 有用数据长度
3.      str=cmdMsg->msg;              //指向数据开头
4.      len=*str;                     //msg 里的第 1 个字节代表后面的数据长度

5.      /********打印出串口接收到的数据，用于提示*********/

6.      for(i=1;i<=len;i++)
7.        HalUARTWrite(0,str+i,1 );
8.        HalUARTWrite(0,"\n",1 );//换行
```

```
9.    /*******发送出去***参考网蜂 1 小时无线数据传输教程*********/

10.   if ( AF_DataRequest( &SampleApp_Periodic_DstAddr, &SampleApp_epDesc,
11.                        SAMPLEAPP_COM_CLUSTERID,//自己定义一个
12.                        len+1,                    // 数据长度
13.                        str,                      //数据内容
14.                        &SampleApp_TransID,
15.                        AF_DISCV_ROUTE,
16.                        AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
17.                         {
18.                         }
19.   else
20.   {
21.   // Error occurred in request to send.
22.   }
23. }
```

SAMPLEAPP_COM_CLUSTERID

这个自己定义的 ID，用于接收方判别,如图 3.58 所示:

```
#define SAMPLEAPP_MAX_CLUSTERS          3   //2
#define SAMPLEAPP_PERIODIC_CLUSTERID    1
#define SAMPLEAPP_FLASH_CLUSTERID       2
#define SAMPLEAPP_COM_CLUSTERID         3
```

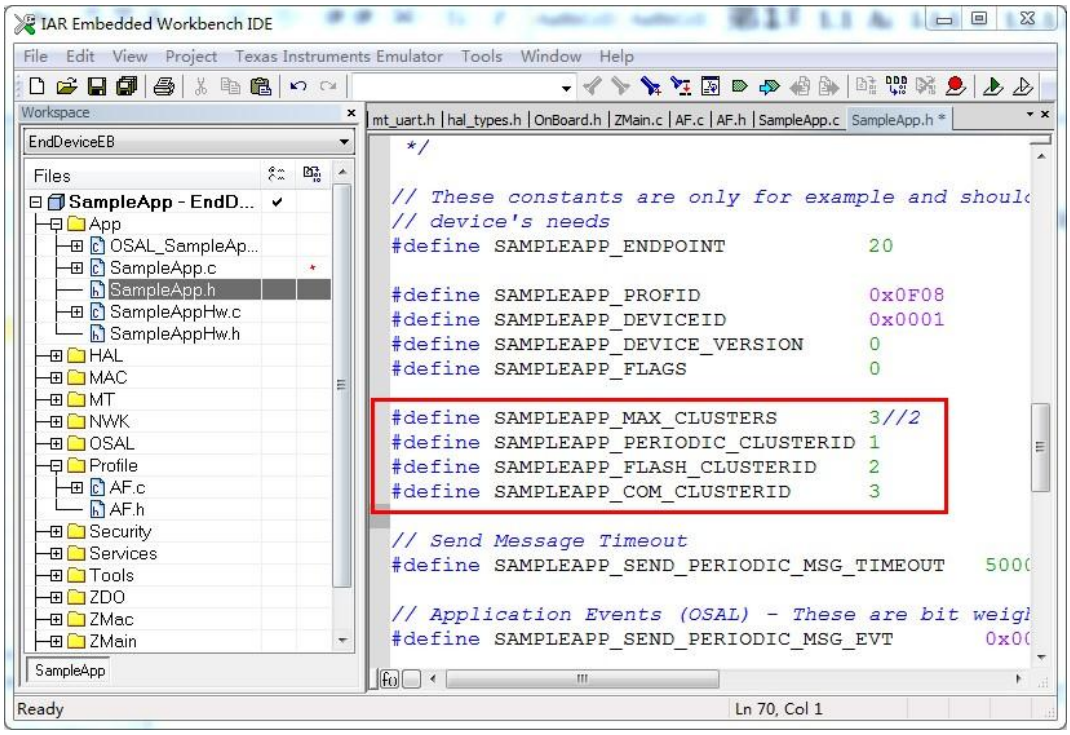


图 3.58 加入串口的 CLUSTERID

到这里，CC2530 从串口接收到信息到转发出去已经完成了，我们可以先下载程序到网蜂开发板，然后可以看到随便发什么都可以打印出来提示了。也就是说 CMD_SERIAL_MSG:事件和 void SampleApp_SerialCMD (mtOSALSerialData_t *cmdMsg)函数已经被成功执行了。图 3.8D 是不是有聊天软件的 feel 了？

图 3.59

2. ZigBe 模块接收到其它 ZigBee 模块发来的信息，然后发送给 PC 机

这部分内容我们网蜂的《ZigBee 实战演练》的 1 小时无线数据传输已经说得够透切了，我们这里只贴上代码。在 SampleApp 找到下面函数，加入红色部分内容。其他 case 这里没用上，可以先注释掉节省资源。

```
Void SampleApp_MessageMSGCB( afIncomingMSGPacket_t *pkt )
 {
   uint8 i,len;
   switch ( pkt->206lustered )
   {
   case SAMPLEAPP_COM_CLUSTERID:      //如果是串口透传的信息
       len=pkt->cmd.Data[0];
       for(i=0;i<len;i++)
       HalUARTWrite(0,&pkt->cmd.Data[i+1],1);//发给 PC 机
```

```
        HalUARTWrite(0,”\n”,1);                    // 回车换行
        break;
      /* case SAMPLEAPP_PERIODIC_CLUSTERID:

        break;

      case SAMPLEAPP_FLASH_CLUSTERID:
        flashTime = BUILD_UINT16(pkt->cmd.Data[1],
                                 pkt->cmd.Data[2] );
        HalLedBlink( HAL_LED_4, 4, 50, (flashTime / 4) );
        break;*/
    }
}
```

如果想进一步节省资源，可以将函数：

uint16 SampleApp_ProcessEvent(uint8 task_id, uint16 events)

里面除了这个透传的 case 以外的 case 事件判断都注释掉。

```
Case CMD_SERIAL_MSG:
        SampleApp_SerialCMD((mtOSALSerialData_t *)MSGpkt);
        break;
```

最后还要修改预编译，注释掉 MT 层的内容。这里注意，选择了协调器、路由器、或者终端编译时都要修改 options 的。

参考如下：如图所示

```
ZTOOL_P1
xMT_TASK
xMT_SYS_FUNC
xMT_ZDO_FUNC
xLCD_SUPPORTED=DEBUG
```

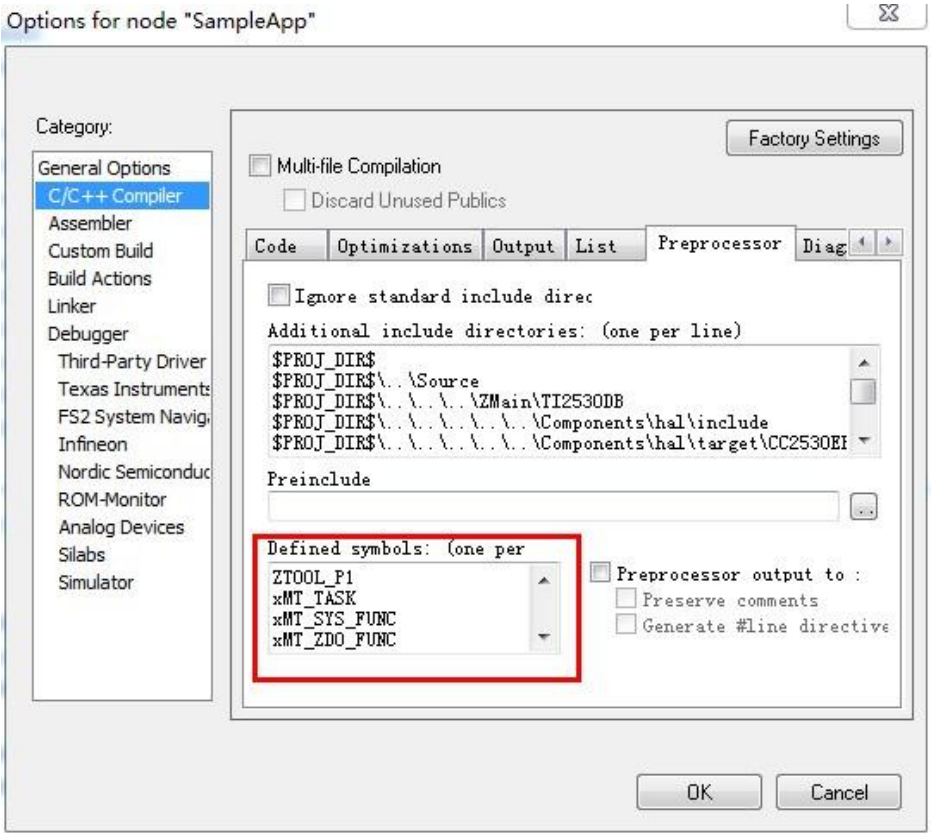


图 3.60 预定义

至此，所有配置完成，万事俱备，只欠下载了，我们把程序分别下载到 2 个 zigbee 节点模块，一个选择协调器（必需），另外一个选择路由。通过网蜂的 USB 转串口或其他串口连接到 2 台 PC 机。打开串口助手设置好参数（波特率 115200bps），可以聊天啦。没有 2 台 PC 机的可以用同一台 PC 的不同串口代替。

哈哈，赶快用串口聊天吧。

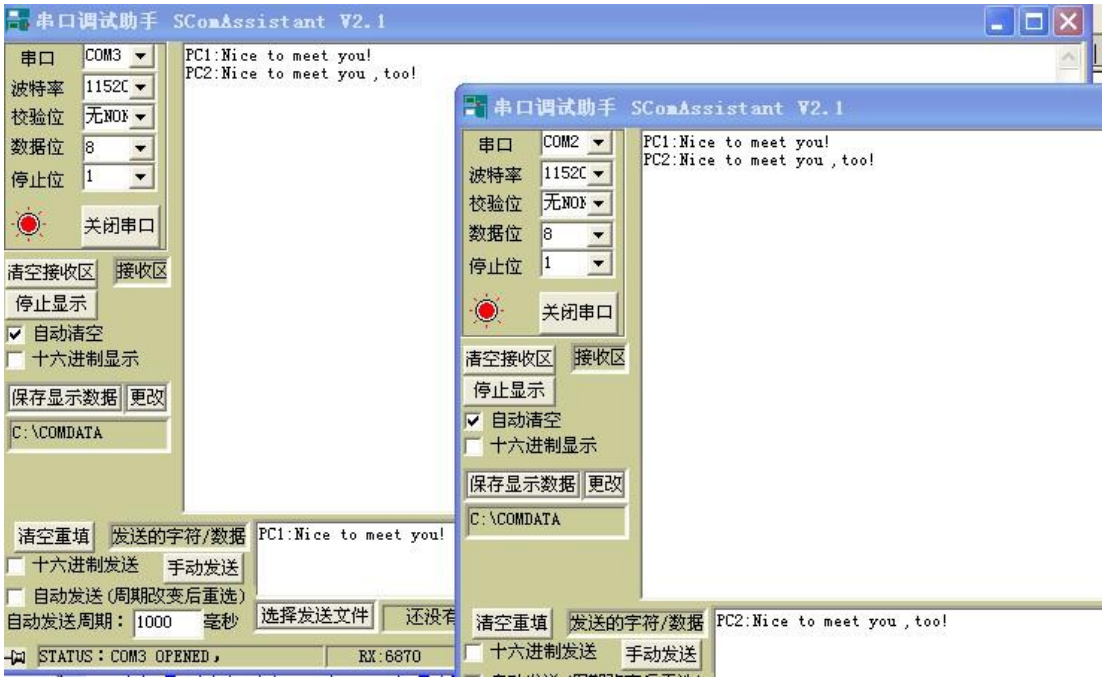


图 3.61 同一台 PC 的 2 个不同串口演示

3.9 网络通讯实验（单播、组播、广播）

前言：Zigbee 的通讯方式主要有三种点播、组播、广播。点播，顾名思义就是点对点通信，也就是 2 个设备之间的通讯，不容许有第三个设备收到信息；组播，就是把网络中的节点分组，每一个组员发出的信息只有相同组号的组员才能收到。广播，最广泛的也就是 1 个设备上发出的信息所有设备都能接收到。这也是 ZigBee 通信的基本方式。

实现平台：网蜂 ZigBee 节点 3 个以上。分别用于协调器、路由器、终端。

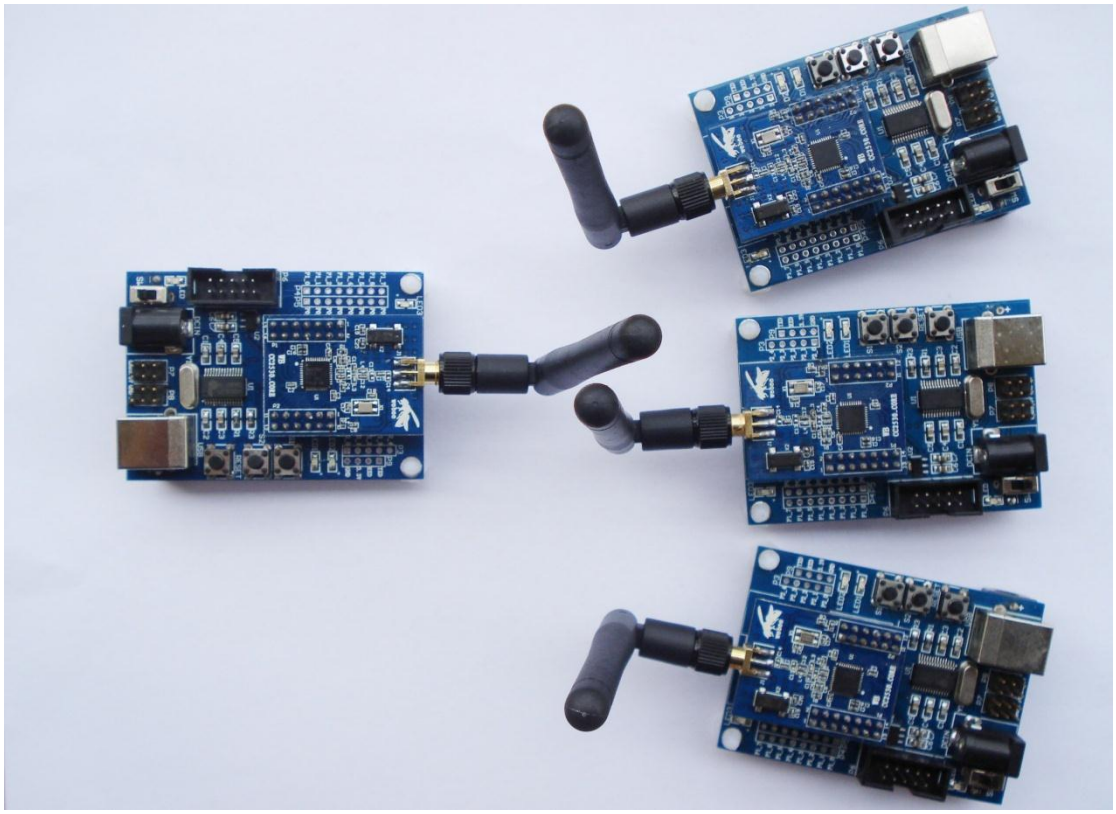


图 3.62 网蜂 ZigBee 节点设备

实验现象：

通过数据的相互传输来了解单播、组播、广播含义。掌握编程方法。

实验讲解：

实验依然使用我们熟悉的 SampleApp.eww 工程来进行。前面我们做的 1 小

时实现数据传输，其实就是一个广播的通讯，实际上 ZigBee 2 节点以上就可以组网和通讯了，大家可不要认为一定要好几个节点才能完成组网。同时，我们需要了解协议栈设计者是如何让我们通过函数实现三种数据发送形式的。

3.9.1 点播（点对点通讯）

点播描述的就是网络中 2 个节点相互通信的过程。确定通信对象的就是节点的 16bit 短地址。下面我们在 SampleApp 例程完通过简单的修改完成单播实验。为了简化大家了理解。数据发送和接收的内容按照本书“一小时实现无线数据传输”章节来描述。

我们打开 AF.h 文件，找到下面代码。

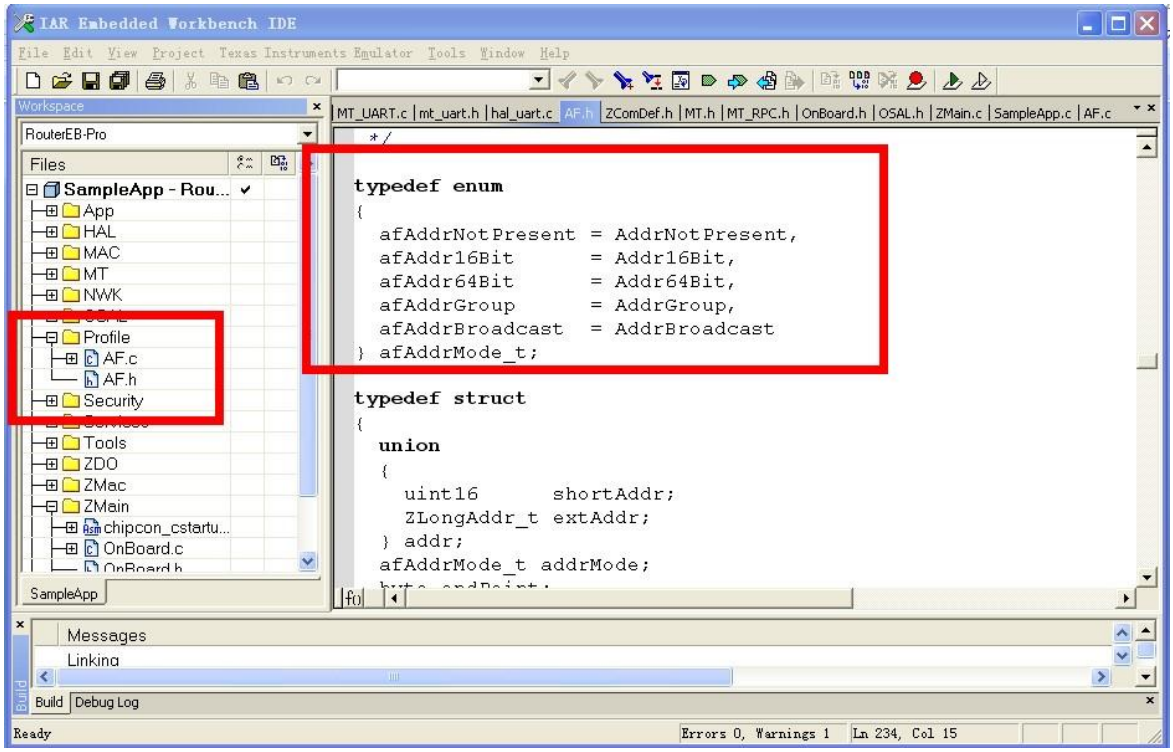


图 3.63

```
typedef enum
{
  afAddrNotPresent = AddrNotPresent,
  afAddr16Bit      = Addr16Bit,
  afAddr64Bit      = Addr64Bit,
  afAddrGroup      = AddrGroup,
  afAddrBroadcast  = AddrBroadcast
} afAddrMode_t;
```

该类型是一个枚举类型：

当 addrMode= Addr16Bit 时，对应点播方式；

当 addrMode= AddrGroup 时，对应组播方式；

当 addrMode= AddrBroadcast 时，对应广播方式；

按照以往的步骤，打开 SampleApp.c 文件

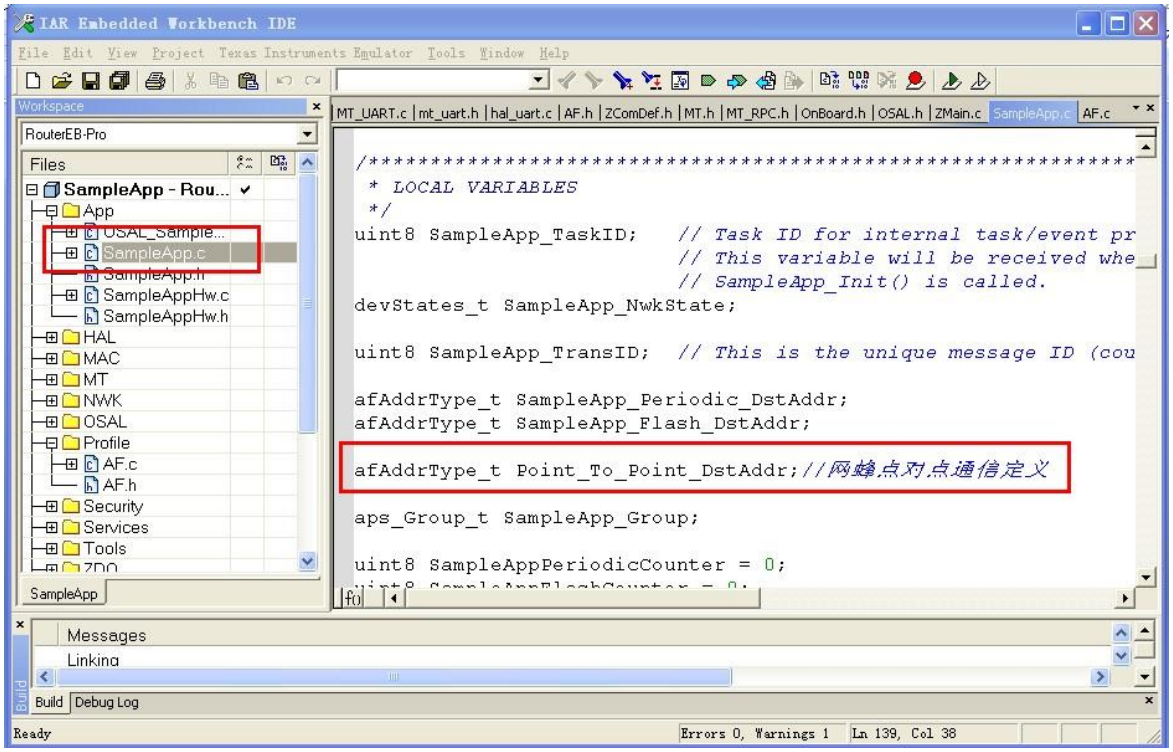


图 3.64 SampleApp.c 文件

可发现已经存在代码如下：

```
afAddrType_t SampleApp_Periodic_DstAddr;
afAddrType_t SampleApp_Flash_DstAddr;
```

分别是组播和广播前面的定义。我们按照格式来添加自己的点播如下（如图 3 所示）：

```
afAddrType_t  Point_To_Point_DstAddr;//网蜂点对点通信定义
```

提示：go to definition of afAddrType_t 可以找到刚才的枚举内容。

下边我们对 Point_To_Point_DstAddr 一些参数进行配置，找到下面的位置，参考 SampleApp_Periodic_DstAddr 和 SampleApp_Flash_DstAddr 我们进行自己的配置。加入如下代码（如图 3.65 所示）：

```
// 网蜂点对点通讯定义
Point_To_Point_DstAddr.addrMode = (afAddrMode_t)Addr16Bit;//点播
Point_To_Point_DstAddr.endPoint = SAMPLEAPP_ENDPOINT;
Point_To_Point_DstAddr.addr.shortAddr = 0x0000; //发给协调器
```

第三行的意思是点播的发送对象是 0x0000，也就是协调器的地址。节点和协调器点对点通讯。

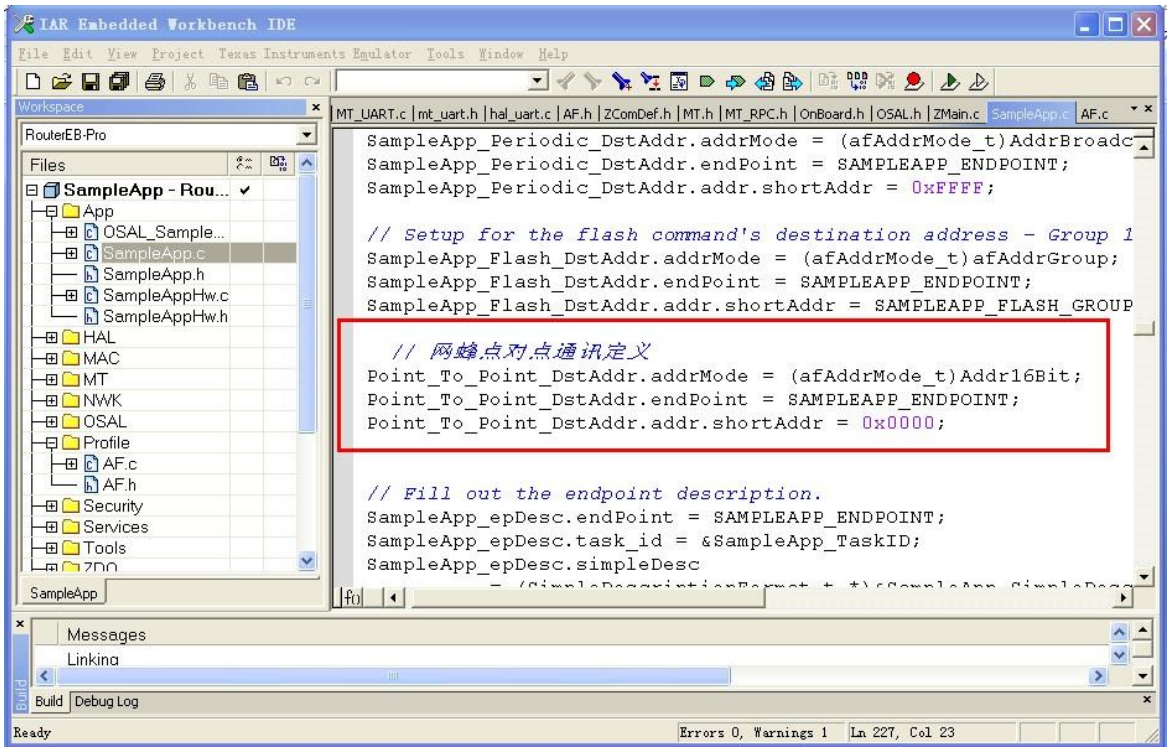


图 3.65

继续添加自己的点对点发送函数，在 SampleAPP.c 最后加入下面代码，如图 3.66 所示：

```
void SampleApp_SendPointToPointMessage( void )
{
  uint8 data[10]={0,1,2,3,4,5,6,7,8,9};
  if ( AF_DataRequest( &Point_To_Point_DstAddr,
                       &SampleApp_epDesc,
                       SAMPLEAPP_POINT_TO_POINT_CLUSTERID,
                       10,
                       data,
                       &SampleApp_TransID,
                       AF_DISCV_ROUTE,
                       AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
  {
  }
```

```
    else
    {
      // Error occurred in request to send.
    }
  }
```

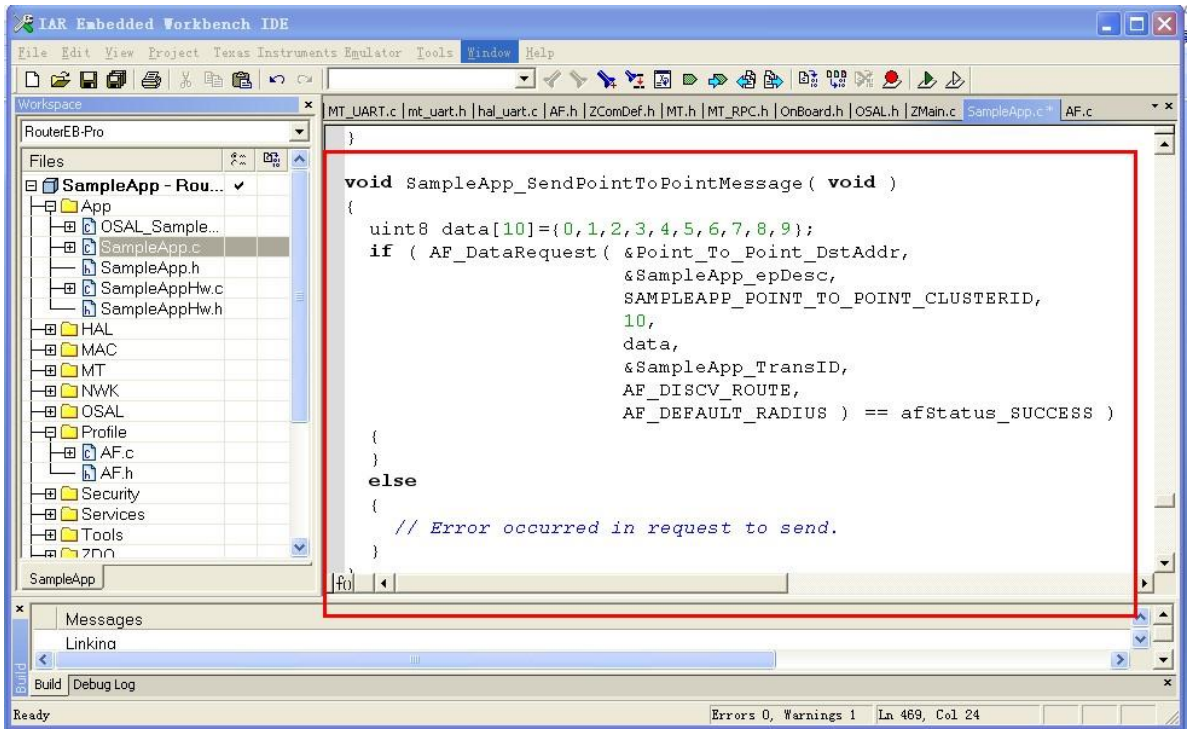


图 3.66 添加点对点发送函数

还需要在 SampleAPP.c 文件开头添加头函数声明：

```
void SampleApp_SendPointToPointMessage( void );
```

其中 Point_To_Point_DstAddr 我们之前已经定义，我们在 SampleApp.h 中加入 SAMPLEAPP_POINT_TO_POINT_CLUSTERID 的定义如所示：

```
#define SAMPLEAPP_POINT_TO_POINT_CLUSTERID  3//传输编号
```

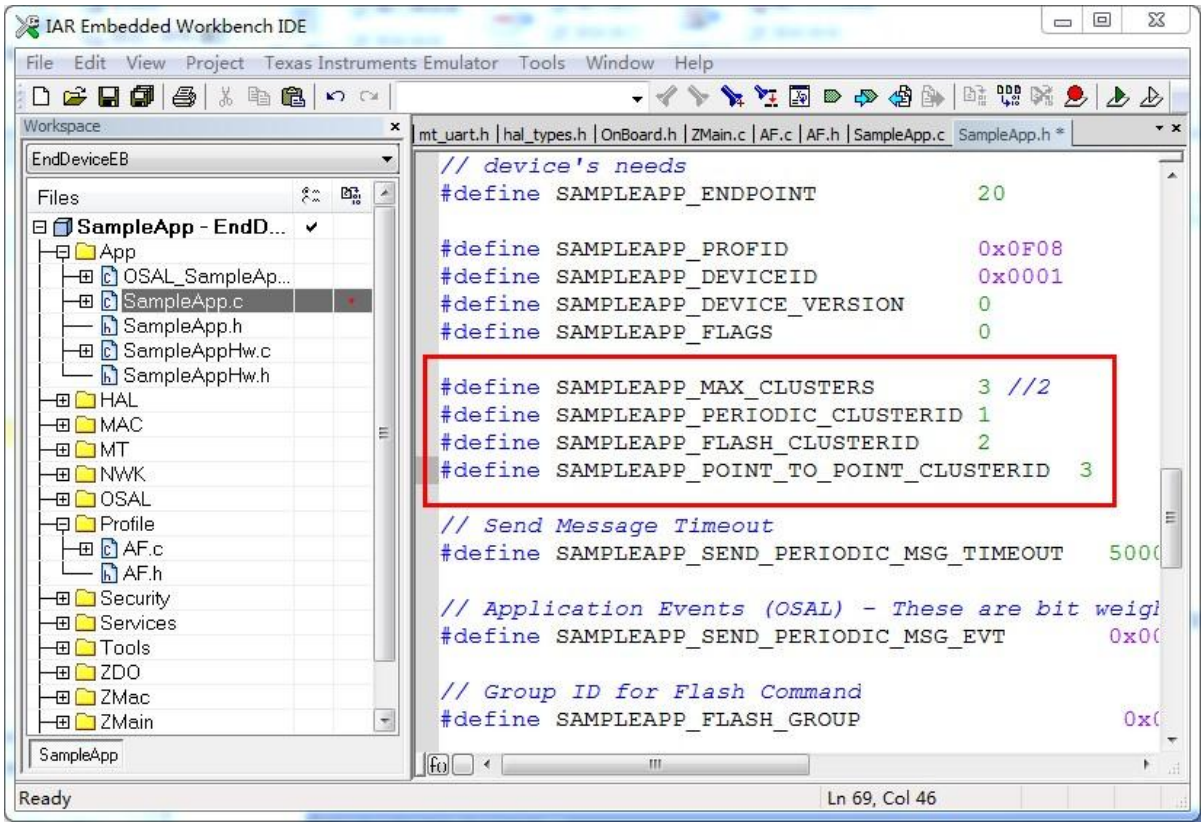


图 3.67 添加 ID 号

接下来为了测试我们的程序，我们把“1 小时实现数据传输”中 SampleApp.c 文件中的 SampleApp_SendPeriodicMessage();函数替换成我们刚刚建立的点对点发送函数 SampleApp_SendPointToPointMessage();这样的话就能实现周期性点播发送数据了（如图 3.68 所示）。

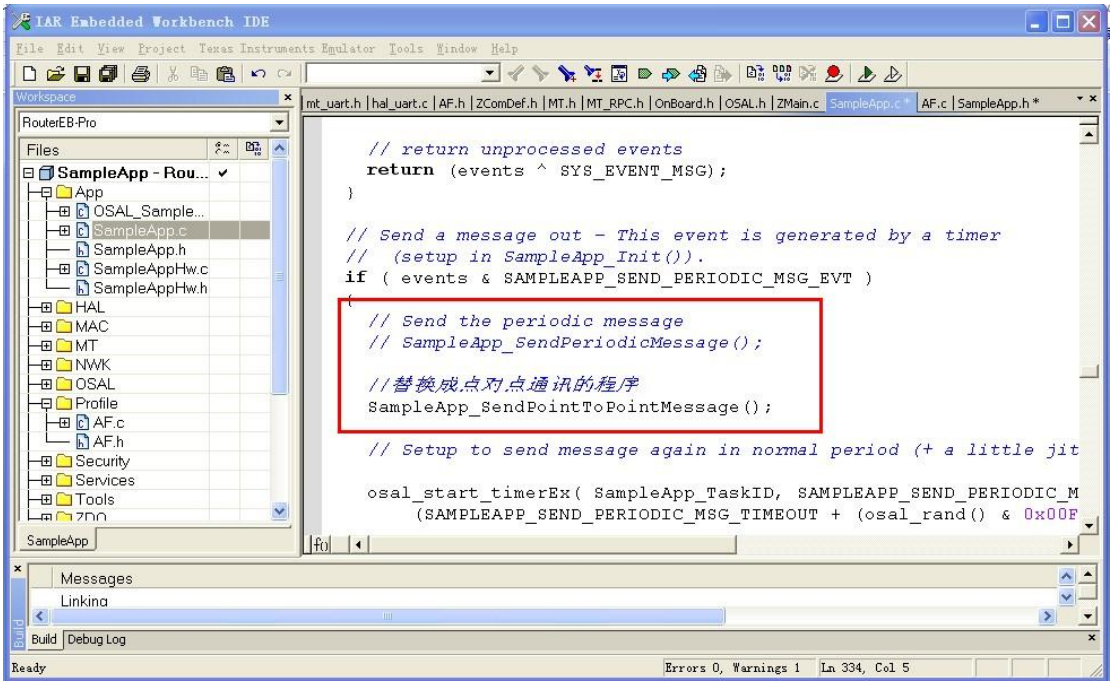


图 3.68

在接收方面，我们进行如下修改：接收 ID 我们在原来基础上改成我们刚定义的 SAMPLEAPP_POINT_TO_POINT_CLUSTERID。如图 3.69 所示：

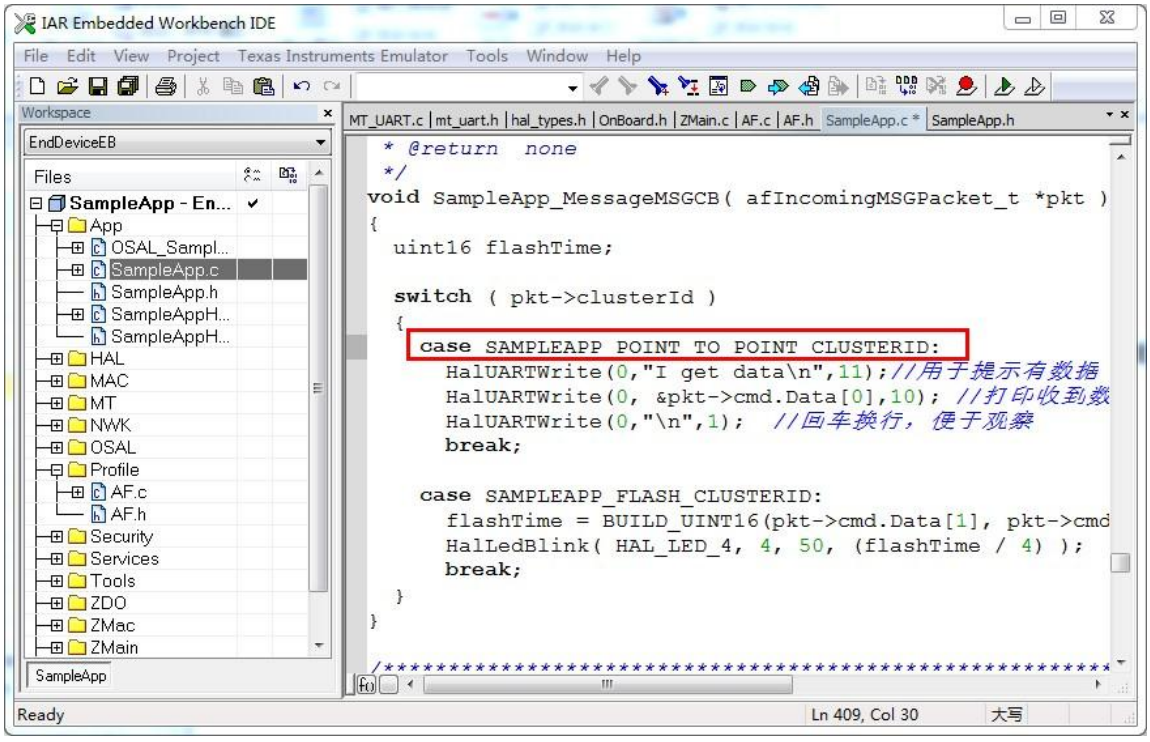


图 3.69 修改接收 ID

由于协调器不允许给自己点播，故周期性点播初始化时协调器不能初始化。如图 3.70 所示：

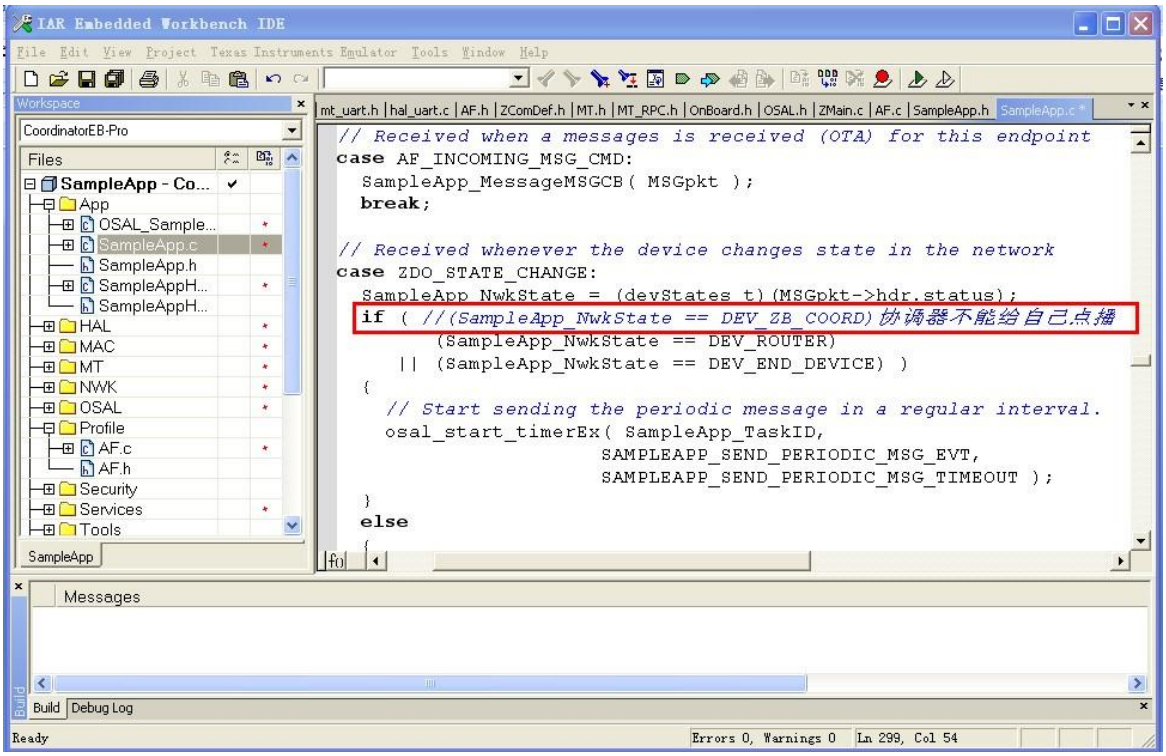


图 3.70 协调器不给自己点播

实验结果：将修改后的程序分别以协调器、路由器、终端的方式下载到 3 个节点设备中，连接串口。可以看到只有协调器在一个周期内收到信息。也就是说路由器和终端均与地址为 0x00（协调器）的设备通信，不与其他设备通信。实现点对点传输。如图 3.71 所示：

图 3.71 点播方式发送数据

3.9.2 组播

组播描述的就是网络中所有节点设备被分组后组内相互通信的过程。确定通信对象的就是节点的组号。下面我们在 SampleApp 例程完通过简单的修改完成组播实验。数据发送和接收的内容依然按照“一小时实现无线数据传输”章节格式。修改流程与点播相似。

关注 SampleApp.c 中 2 项内容：

1、组播 afAddrType_t 的类型变量

```
afAddrType_t SampleApp_Flash_DstAddr; //组播
```

2、组播内容的结构体：

```
aps_Group_t SampleApp_Group; //分组内容
```

如图 3.72 如所示：

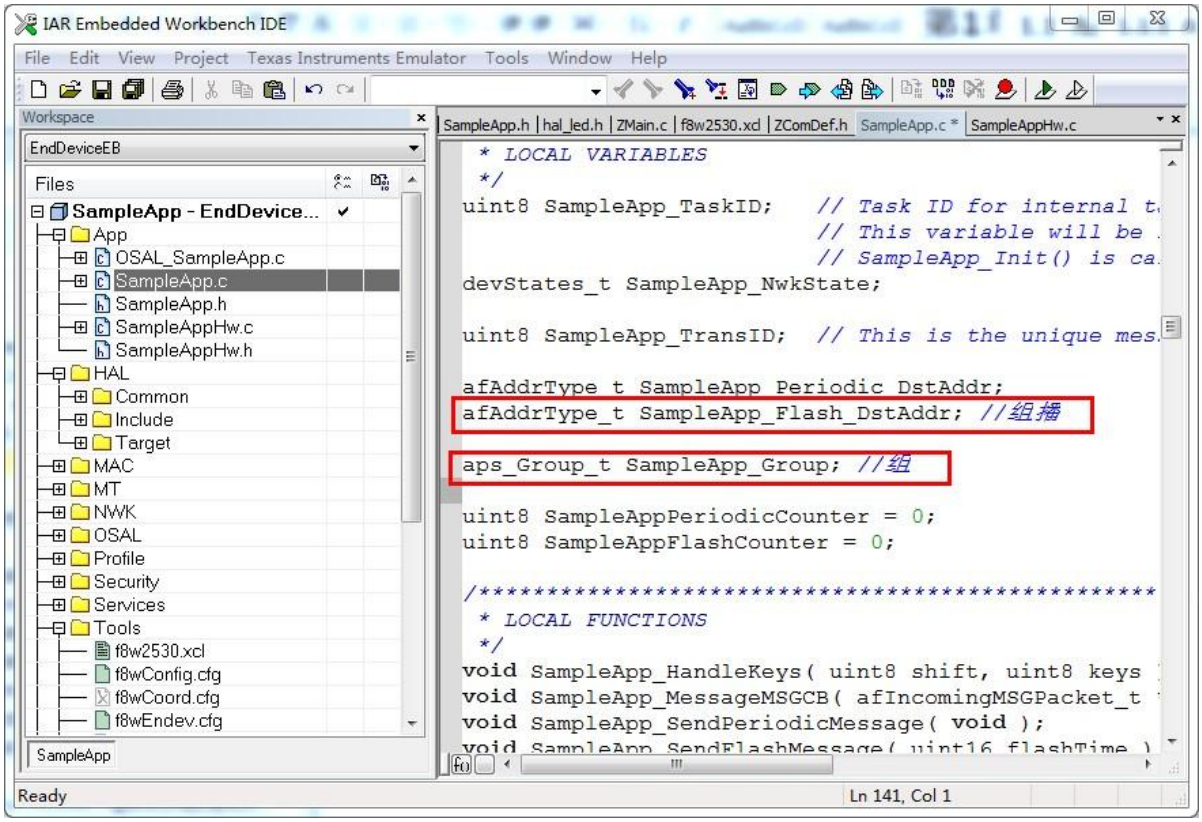


图 3.72 组播定义

组播参数的配置。代码如下（图 3.73 所示）：

```
// Setup for the flash command's destination address - Group 1
SampleApp_Flash_DstAddr.addrMode = (afAddrMode_t)afAddrGroup;
SampleApp_Flash_DstAddr.endPoint = SAMPLEAPP_ENDPOINT;
SampleApp_Flash_DstAddr.addr.shortAddr = SAMPLEAPP_FLASH_GROUP;
```

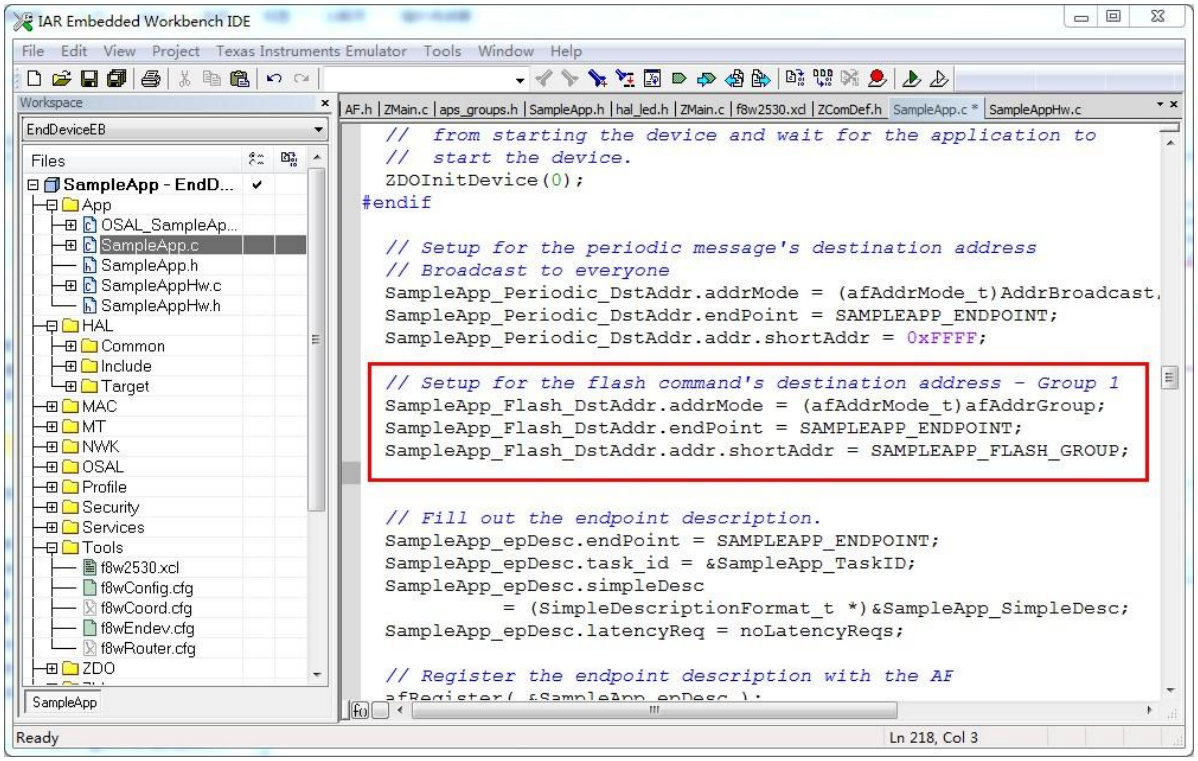


图 3.73

已经定义的组信息代码，将 ID 修改成组号相对应,方便以后自己扩展分组需要” SAMPLEAPP_FLASH_GROUP”,如图 3.74 所示：

```
// By default, all devices start out in Group 1
SampleApp_Group.ID = SAMPLEAPP_FLASH_GROUP;//0x0001;
osal_memcpy( SampleApp_Group.name, "Group 1", 7  );
aps_AddGroup( SAMPLEAPP_ENDPOINT, &SampleApp_Group );
```

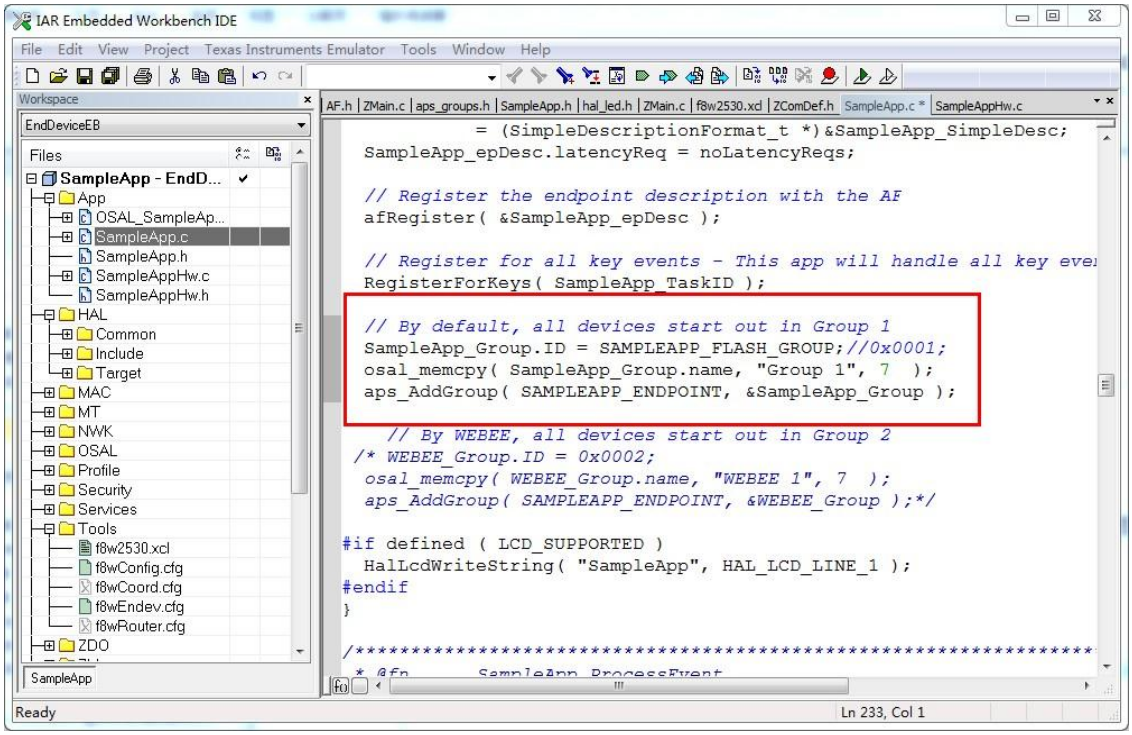


图 3.74

在 SampleApp.h 里面可以看到组号为 0x0001(如图 3.75 所示)：

```
// Group ID for Flash Command
#define SAMPLEAPP_FLASH_GROUP          0x0001
```

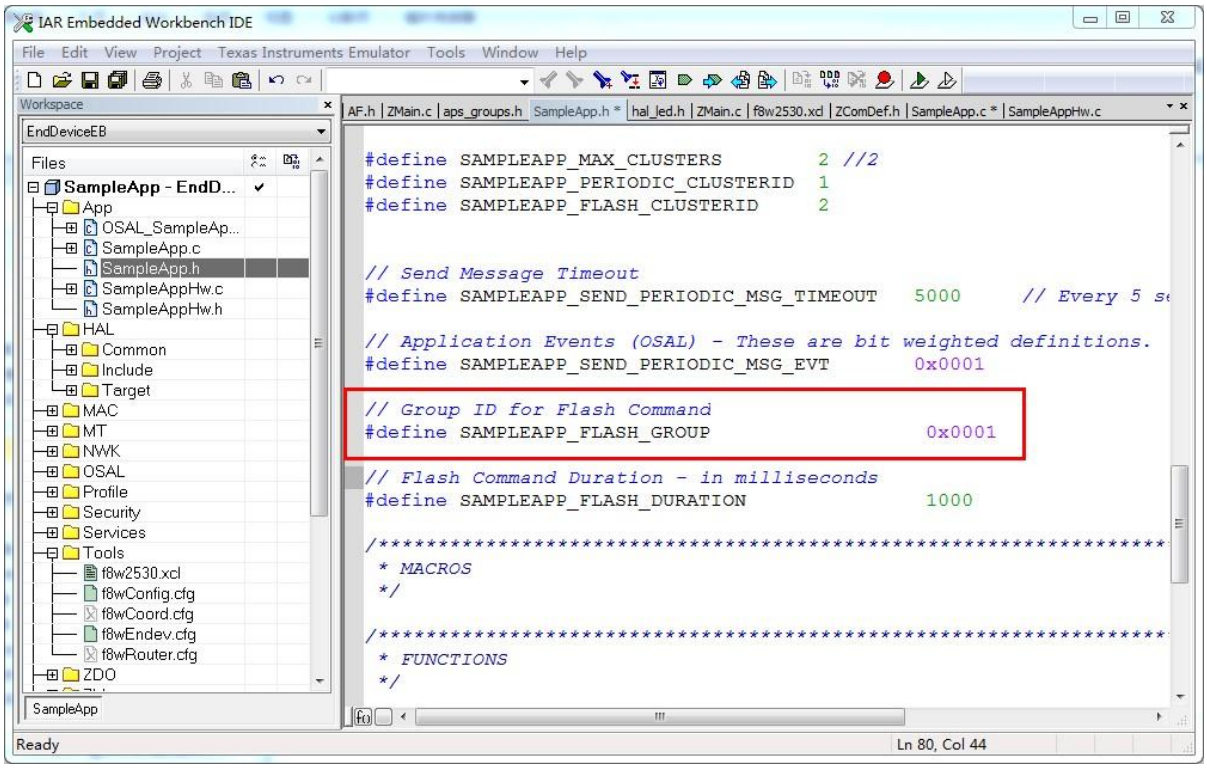


图 3.75 组 ID 号

接下来在 SampleAPP.c 最后面添加自己的组播发送函数，代码如下（图 3.76 所示）：

```
void SampleApp_SendGroupMessage( void )
{
  uint8 data[10]={'0','1','2','3','4','5','6','7','8','9'};//自定
                                                          //义数据
if ( AF_DataRequest( &SampleApp_Flash_DstAddr,
                     &SampleApp_epDesc,
                     SAMPLEAPP_FLASH_CLUSTERID,
                     10,
                     data,
                     &SampleApp_TransID,
                     AF_DISCV_ROUTE,
                     AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
  {
```

```
    }
    else
    {
      // Error occurred in request to send.
    }
  }
```

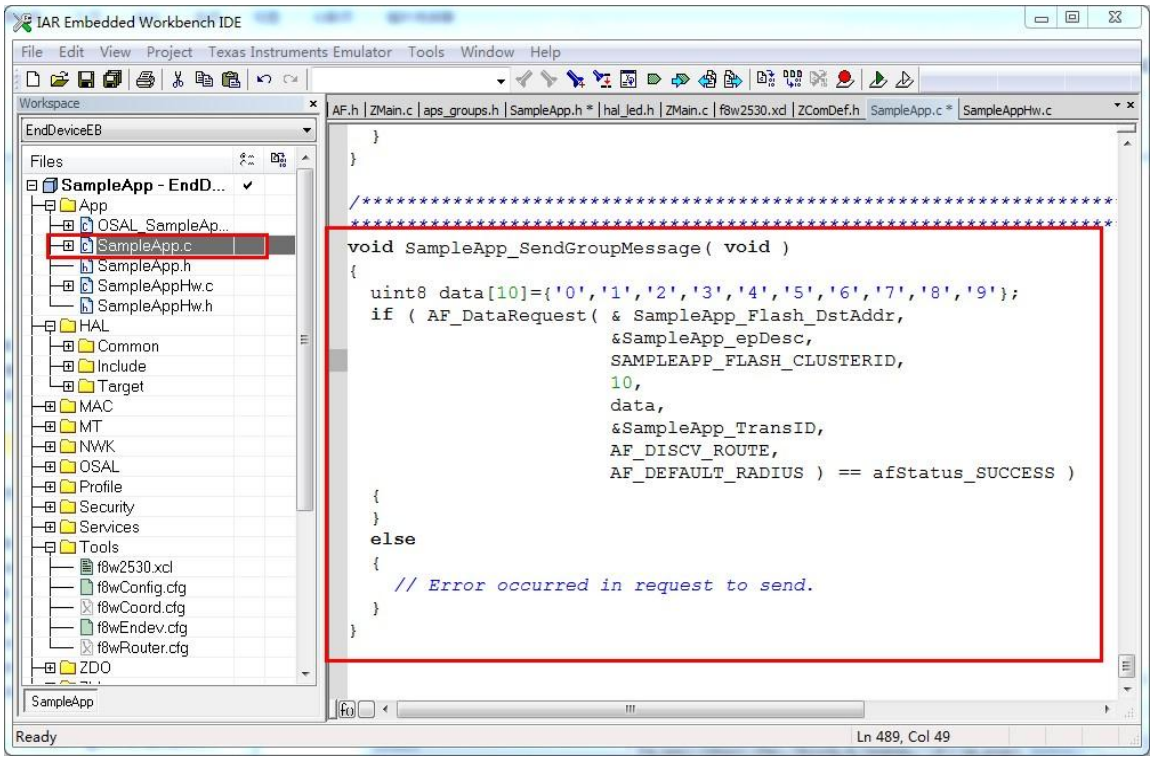


图 3.76 组播函数代码

添加函数后别忘了在 SampleApp.c 函数声明里加入：

```
void SampleApp_SendGroupMessage(void); //网蜂组播通讯发送函数定义.
```

否则编译将报错。

SAMPLEAPP_FLASH_CLUSTERID 的定义如下所示：

```
#define SAMPLEAPP_FLASH_CLUSTERID      2
```

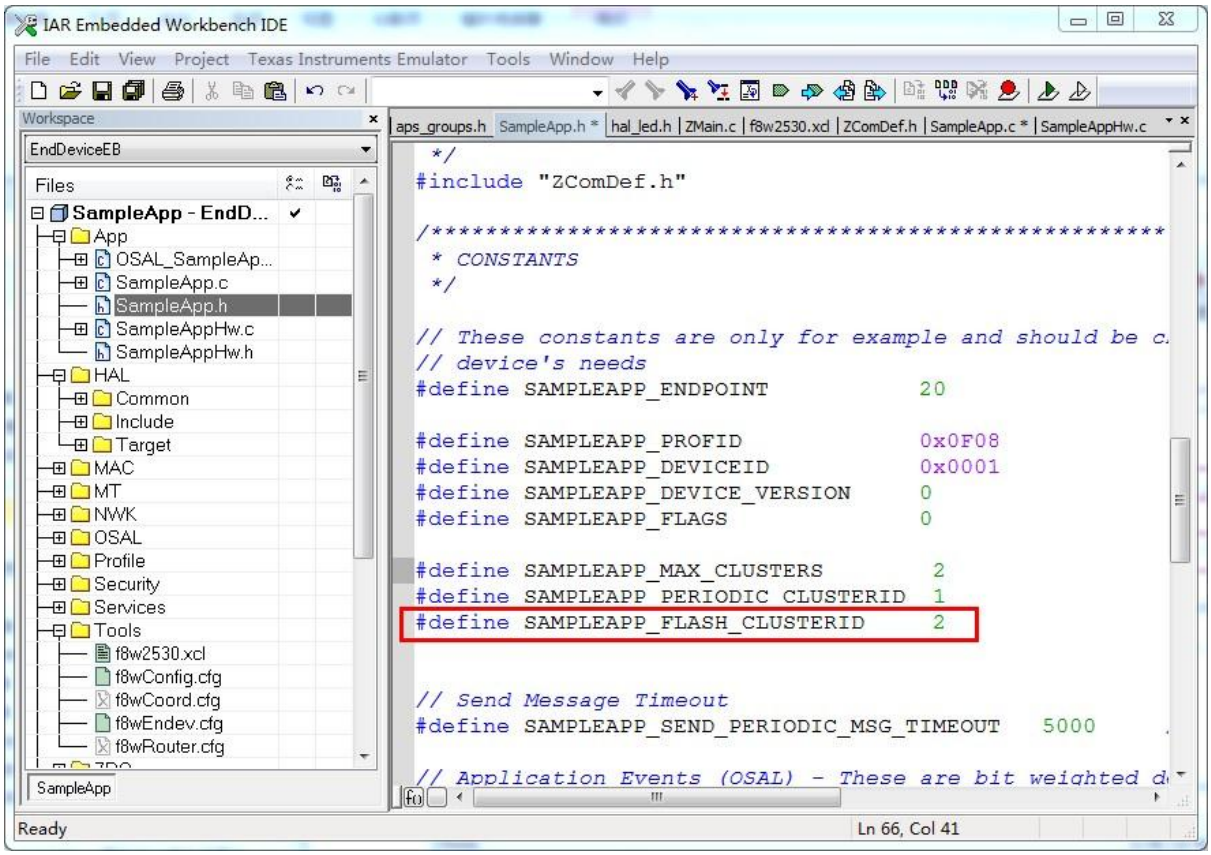


图 3.77

接下来为了测试我们的程序，我们把“1 小时实现数据传输”中 SampleApp.c 文件中的 SampleApp_SendPeriodicMessage();函数替换成我们刚刚建立的组播发送函数 SampleApp_SendGroupMessage();这样的话就能实现周期性组播发送数据了（图 3.78 所示）。

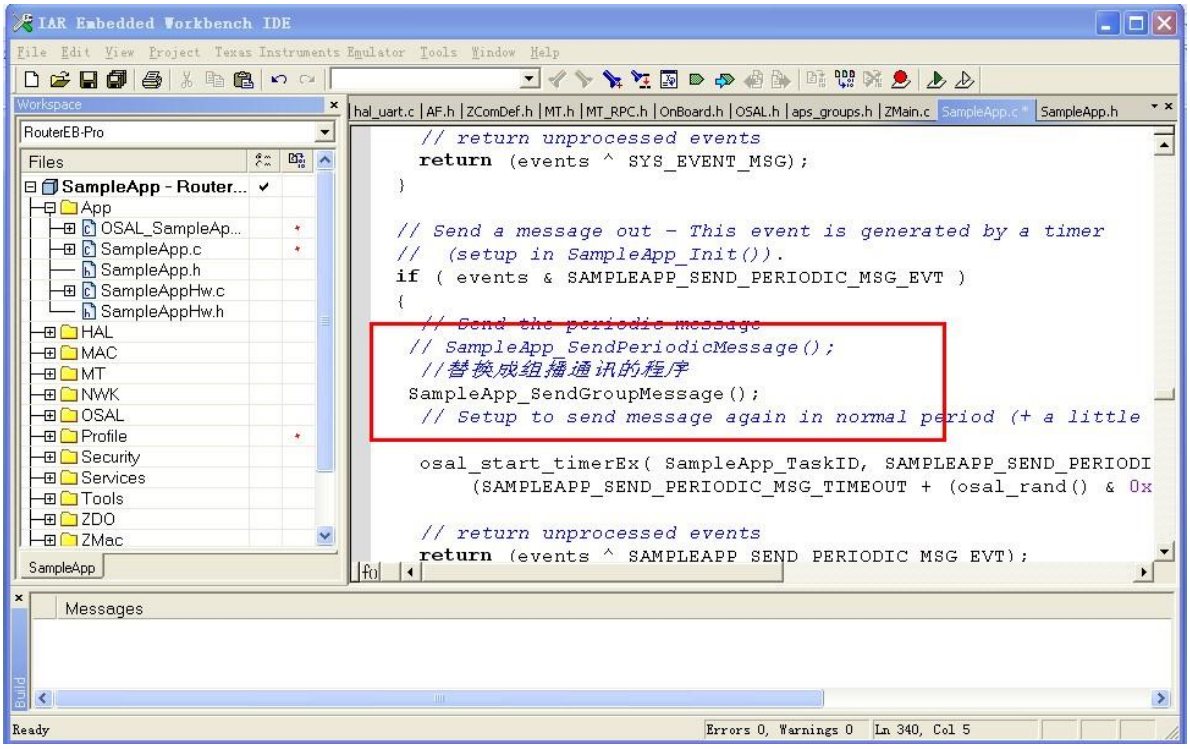


图 3.78 替换成组播函数

在接收方面，我们进行如下修改：组播接收函数改成我们自己来获取数据，（如图 3.79 所示）：

```
case SAMPLEAPP_FLASH_CLUSTERID:
    HalUARTWrite(0,"I get data\n",11);//用于提示有数据
    HalUARTWrite(0, &pkt->cmd.Data[0],10); //打印收到数据
    HalUARTWrite(0,"\n",1);  //回车换行，便于观察
```

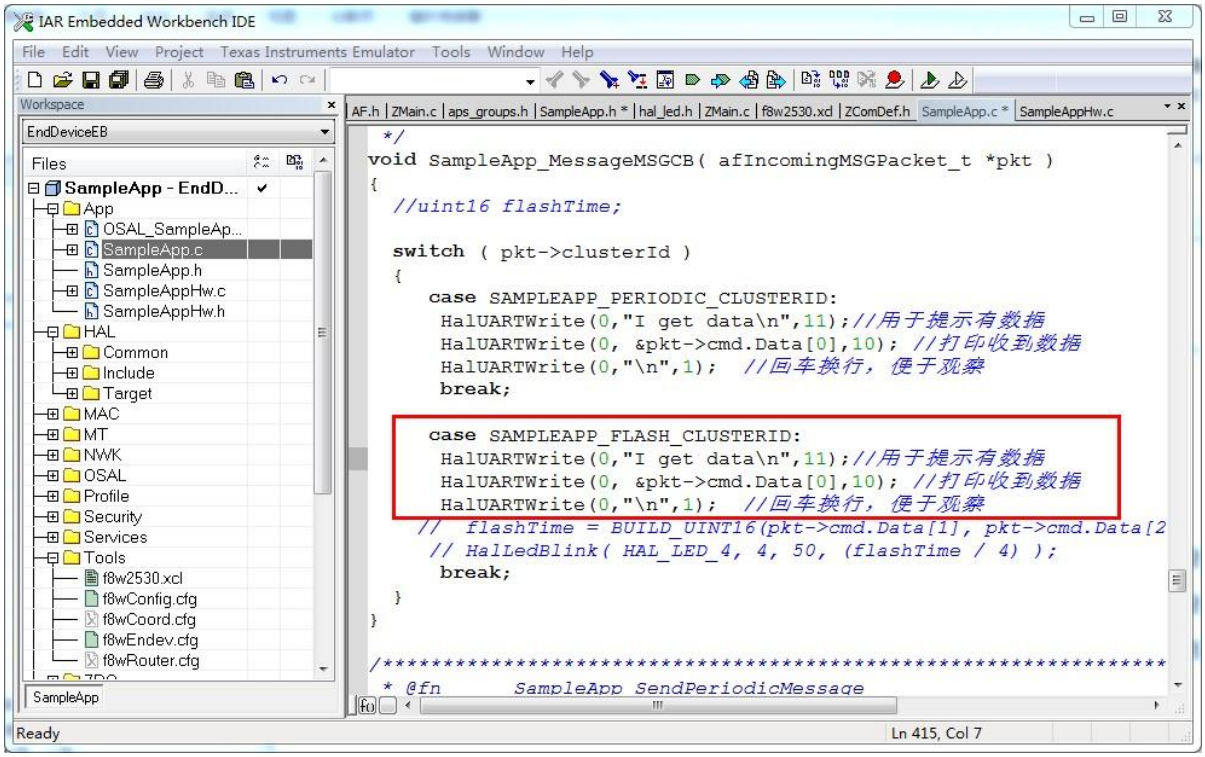


图 3.79

实验结果：将修改后的程序分别以 1 个协调器、2 个路由器的方式下载到 3 个设备，把协调器和路由器组号 1 设置成 0x0001，路由器设备 2 组号设成 0x0002。如图 3.80 所示；连接串口，可以观察到只有 0x0001 的两个设备相互发送信息。如图 3.81 所示。（注意：终端设备不参与组播实验，具体往下看！）

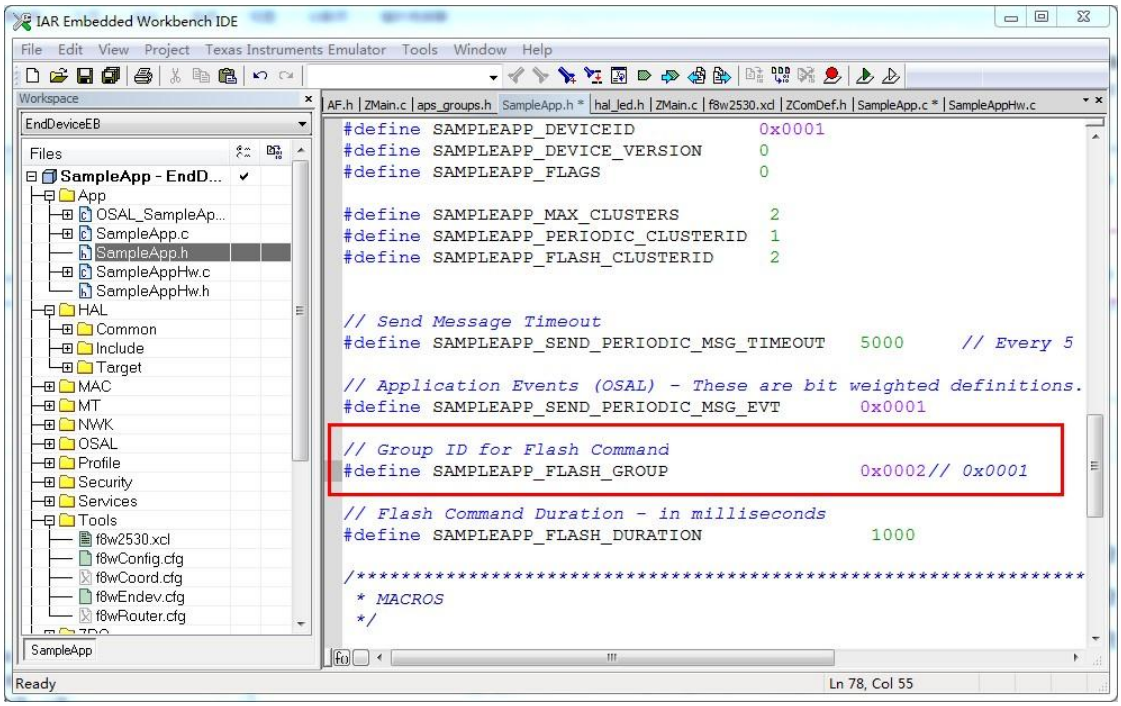


图 3.80

图 3.81 组播方式发送数据

组播知识扩展：

刚刚我们提到终端设备不参与组播，原因是 SampleAPP 例程中终端设备默认采用睡眠中断的工作方式，射频不是一直工作，我们可以下载组播例程到终端，发现不能正常接收组播信息。确实需要使用终端设备参与组播可以参考下面方法：

这个在协议规范里面是有规定的，睡眠中断不接收组播信息，如果一定想要接收的话，只有将终端的接收机一直打开，这样就可以接收到了。具体做法为：

将 f8config.cfg 配置文件中的 -RFD_RCVC_ALWAYS_ON=FALSE 改为 -RFD_RCVC_ALWAYS_ON=TRUE 就可以了！

3.9.3 广播

广播就是任何一个节点设备发出广播数据，网络中的任何设备都能收到。有了前面点播和组播的实验基础，广播的实验进行起来就得心应手了。组播的定义都是协议栈预先定义好的。所以我们直接来运用就可以了。

我们在协议栈 SampleApp 中找到广播参数的配置。代码如下。

```
SampleApp_Periodic_DstAddr.addrMode = (afAddrMode_t)AddrBroadcast;
SampleApp_Periodic_DstAddr.endPoint = SAMPLEAPP_ENDPOINT;
SampleApp_Periodic_DstAddr.addr.shortAddr = 0xFFFF;
```

0xFFFF 是广播地址。协议栈广播地址主要有 3 种类型：

具体的定义如下：

0xFFFF——数据包将被传送到网络上的所有设备，

包括睡眠中的设备。对于睡眠中的设备，数据包将被保留在其父亲节点直到查询到它，或者消息超时。

0xFFFD——数据包将被传送到网络上的所有在空闲时

打开接收的设备(RXONWHENIDLE)，也就是说，除了睡眠中的所有设备。

0xFFFC——数据包发送给所有的路由器，包括协调器。

我们使用默认的 0xFFFF,如图 3.82 所示

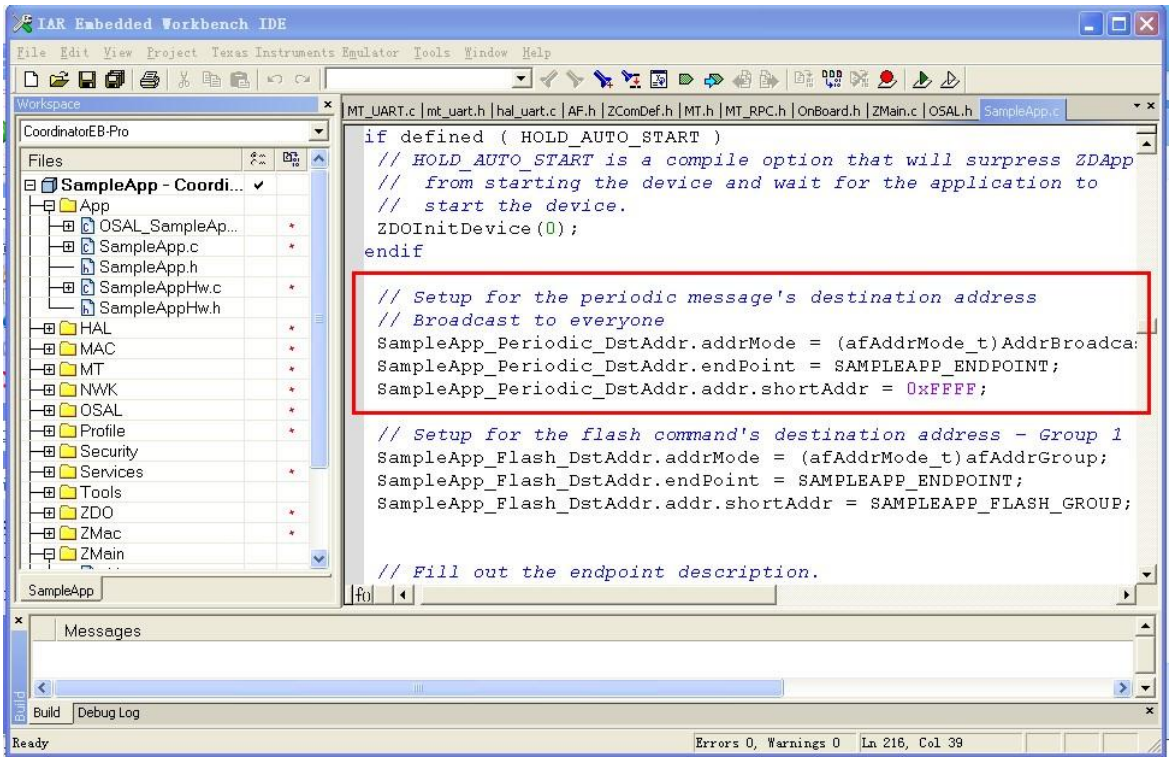


图 3.82

找到自带广播发送函数，修改代码如下（已于一小时实现无线数据传输章节完成，如图 3.83 所示）：

```
void SampleApp_SendPeriodicMessage( void )
{
  uint8 data[10]={'0','1','2','3','4','5','6','7','8','9'};//自定义
                                                            //数据
    if ( AF_DataRequest( &SampleApp_Periodic_DstAddr,
                         &SampleApp_epDesc,
                         SAMPLEAPP_PERIODIC_CLUSTERID,
                         10,
                         data,
                         &SampleApp_TransID,
                         AF_DISCV_ROUTE,
                         AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
    {
```

```
    }
    else
    {
      // Error occurred in request to send.
    }
  }
```

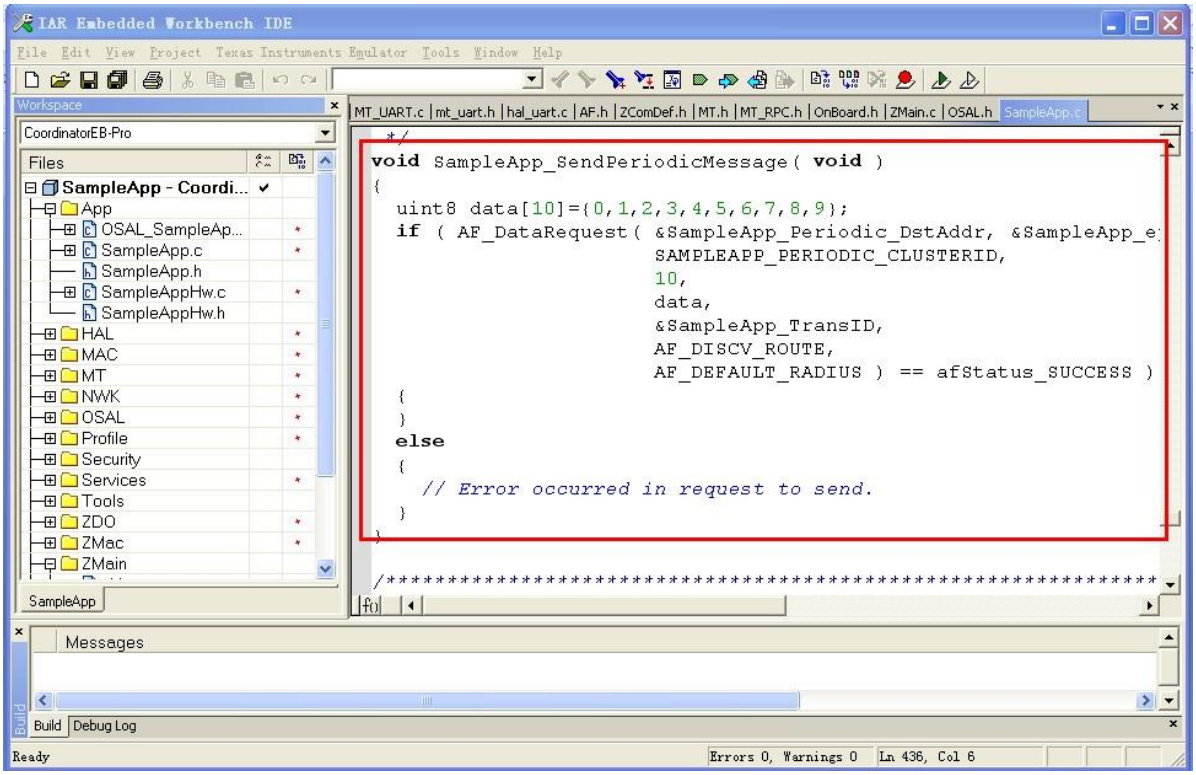


图 3.83

其中图 3.84 所示：

```
#define SAMPLEAPP_PERIODIC_CLUSTERID  1 //广播传输编号
```

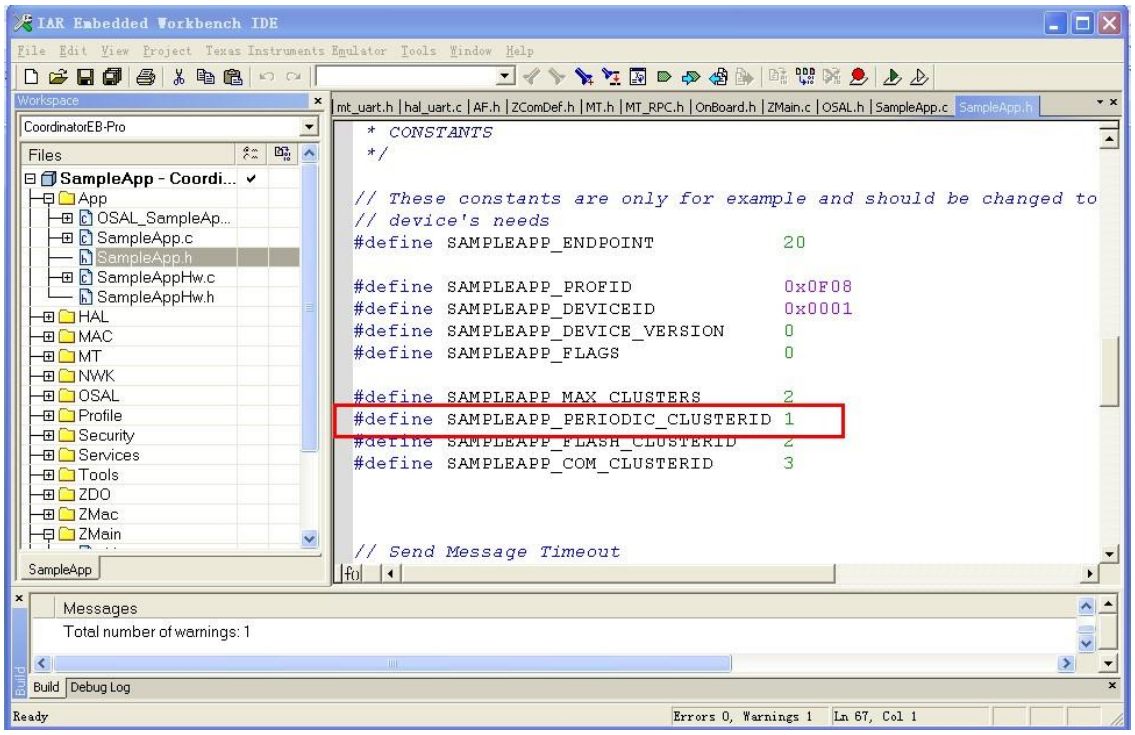


图 3.84

接下来测试我们的程序，我们按照原来代码保留函数 SampleApp_SendGroupMessage();这样的话就能实现周期性广播播发送数据了如图 3.85 所示。

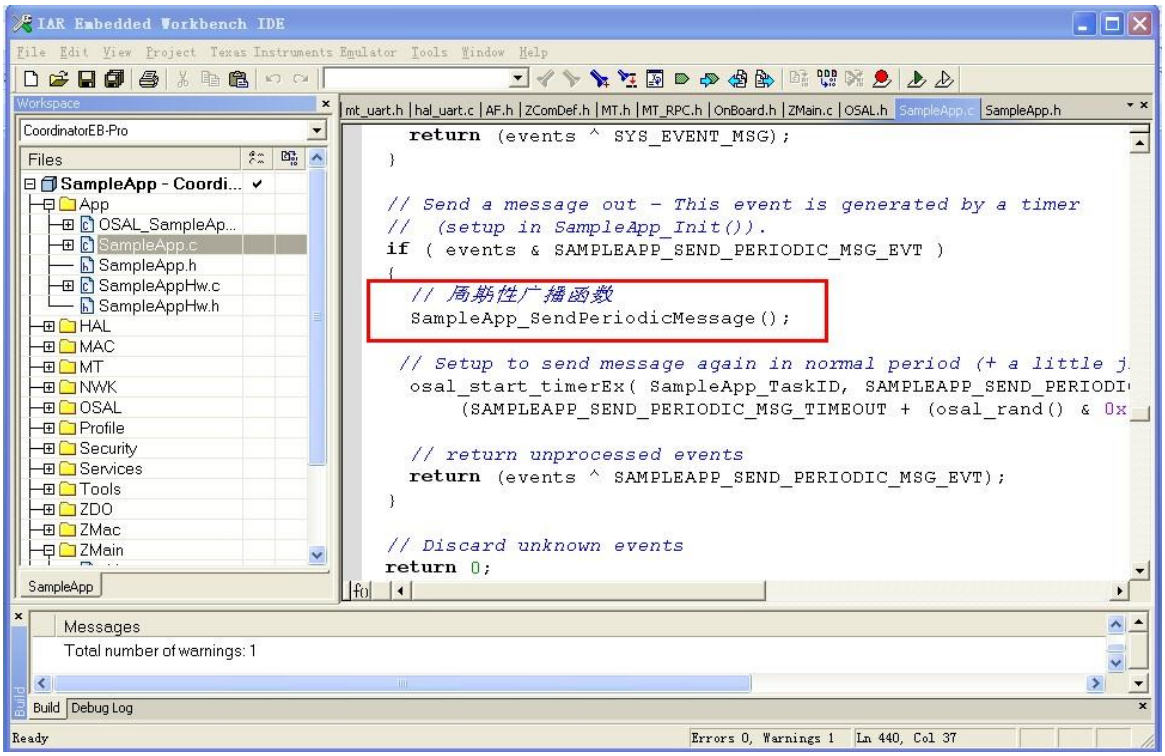


图 3.85

在接收方面，默认接收 ID 就是刚定义的周期性广播发送 ID:

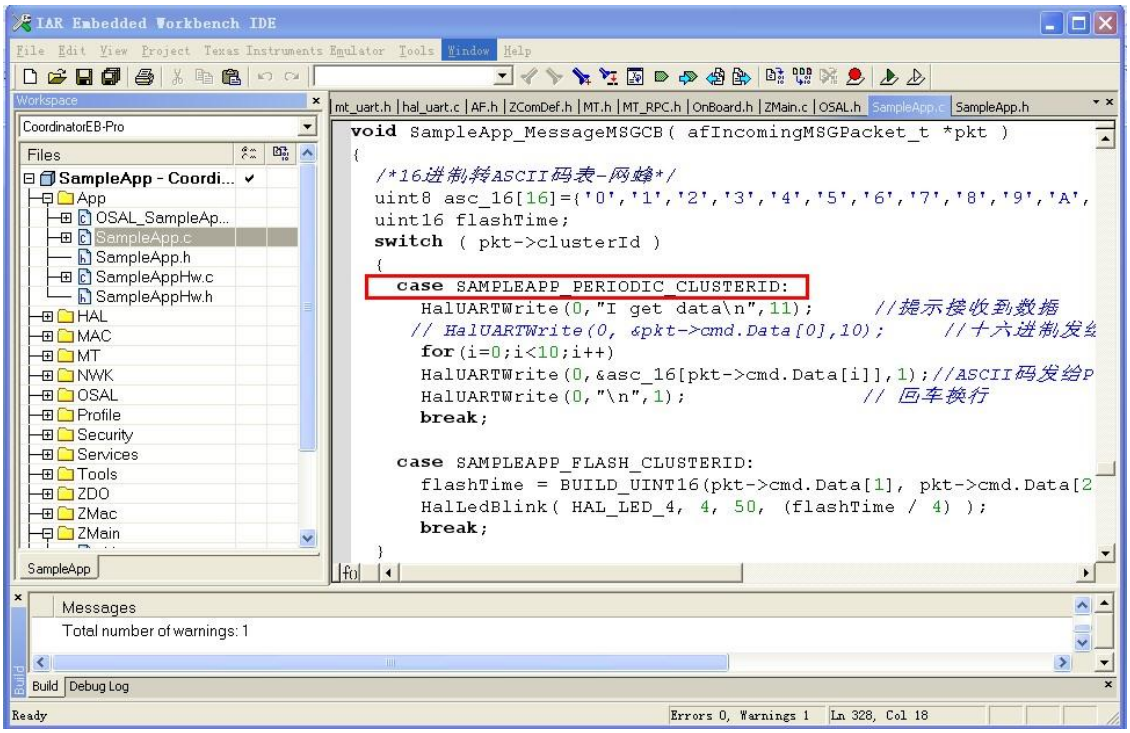


图 3.86

实验结果：将修改后的程序分别以协调器、路由器、终端的方式下载到 3 个设备，可以看到各个设备都在广播发送信息，同时也接收广播信息。

图 3.87 广播方式发送数据

通过网蜂提供的实验代码的修改、编写，相信大家对 ZigBee 网络的点播、组播、广有一定了解。在以后的实验和项目中我们会经常用的这三种数据传输方式。大家可以进一步了解函数的一些其他参数设置从而进一步深化。

3.10 Zigbee 协议栈网络管理

前言：ZigBee 协议栈网络管理这章内容主要是对新加入的设备节点的设备管。我们都知道每个 CC2530 芯片出厂时候都有一个全球唯一的 32 位 MAC 地址。当时当设备连入网络中的时候，每个设备都能获得由协调器分配的 16 位短地址，协调器默认地址（0x0000）。很多时候网络就是通过短地址进行管理。

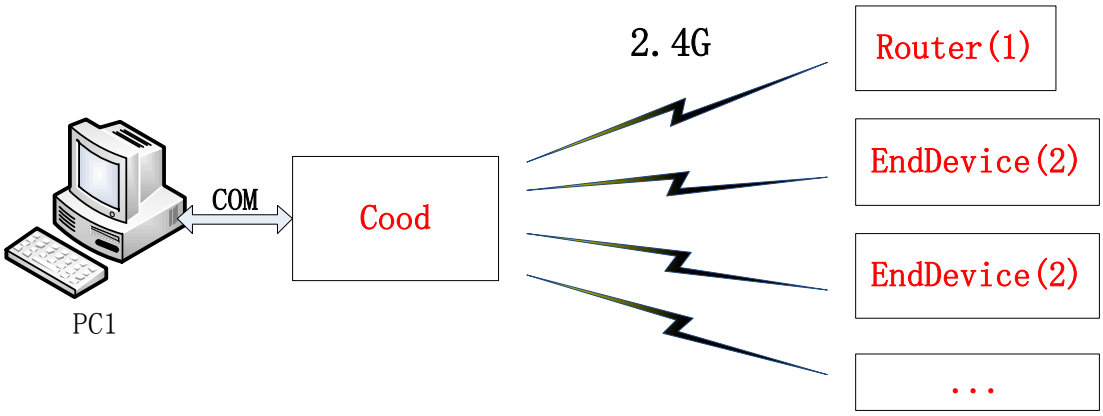


图 3.88 网络管理系统框图

实现平台：网蜂 ZigBee 节点 3 个以上。包括协调器、路由器、终端。

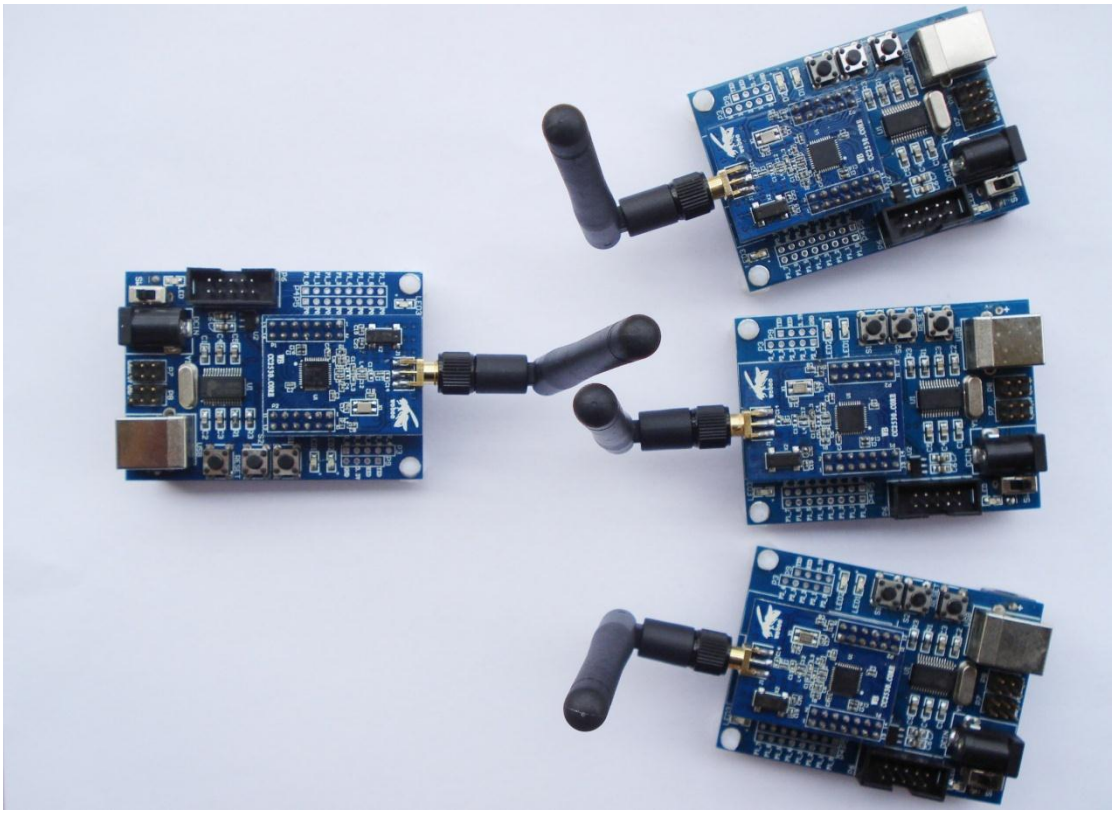


图 3.89 网蜂 ZigBee 节点

实验现象：路由器（编号 1）、终端设备（编号 2）发送自己的定义的设备号给协调器，协调器通过接收到的设备号判断设备类型，并且获取设备的短地址，通过串口打印出来。

实验讲解：实验依然使用我们熟悉的 SampleApp.eww 工程来进行。要实现协调器收集数据的功能，可以使用点播方式传输数据，点播地址为协调器地址（0x0000），避免了路由器和终端之间的互传，减少网络数据拥塞。

实验是在点播例程的基础上进行的，相关内容请参考网蜂《ZigBee 实战演练》点播（点对点通讯）章节内容。下面我们开始在点播程序基础上完成自己的实验。

修改点播信息发送函数，代码如下：

```
void SampleApp_SendPointToPointMessage( void )
{
   uint8 device;          //设备类型变量
   if ( SampleApp_NwkState == DEV_ROUTER )
        device=0x01;   //编号 1 表示路由器
   else if (SampleApp_NwkState == DEV_END_DEVICE)
        device=0x02;   //编号 2 表示终端
   else
        device=0x03;   //编号 3 表示出错

   if ( AF_DataRequest( &Point_To_Point_DstAddr,   //发送设备类型编号
                        &SampleApp_epDesc,
                        SAMPLEAPP_POINT_TO_POINT_CLUSTERID,
                        1,
                        &device,
                        &SampleApp_TransID,
                        AF_DISCV_ROUTE,
                        AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
   {
   }
   else
   {
      // Error occurred in request to send.
   }
}
```

修改完成后系统设备自动检测自己烧写的类型，然后发送对应的编号。路

由器编号为 1，终端编号为 2。

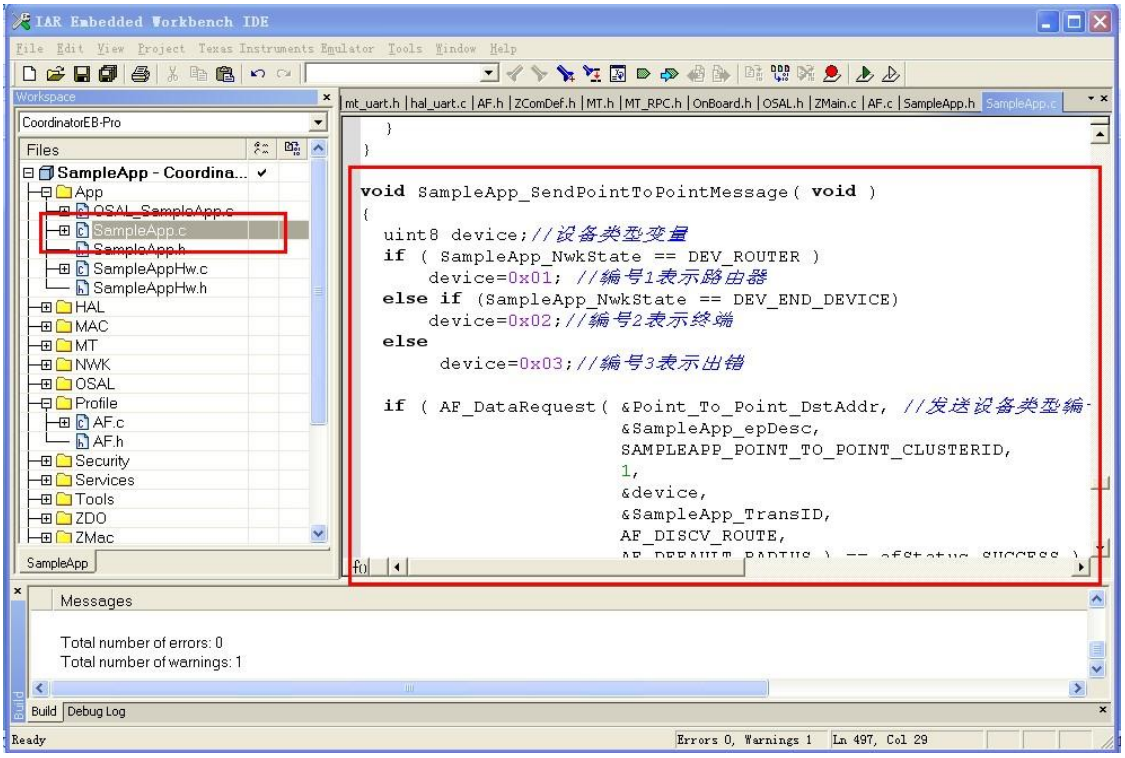


图 3.90

数据接收方面，我们对接收到的数据进行判断，区分路由器和终端设备。然后在数据包中取出 16 位短地址。通过串口打印出来。

我们先看看短地址在数据包里的存放位置。依次是 **pkt--- srcAddr--- shortAddr**。如图 3.91 所示：

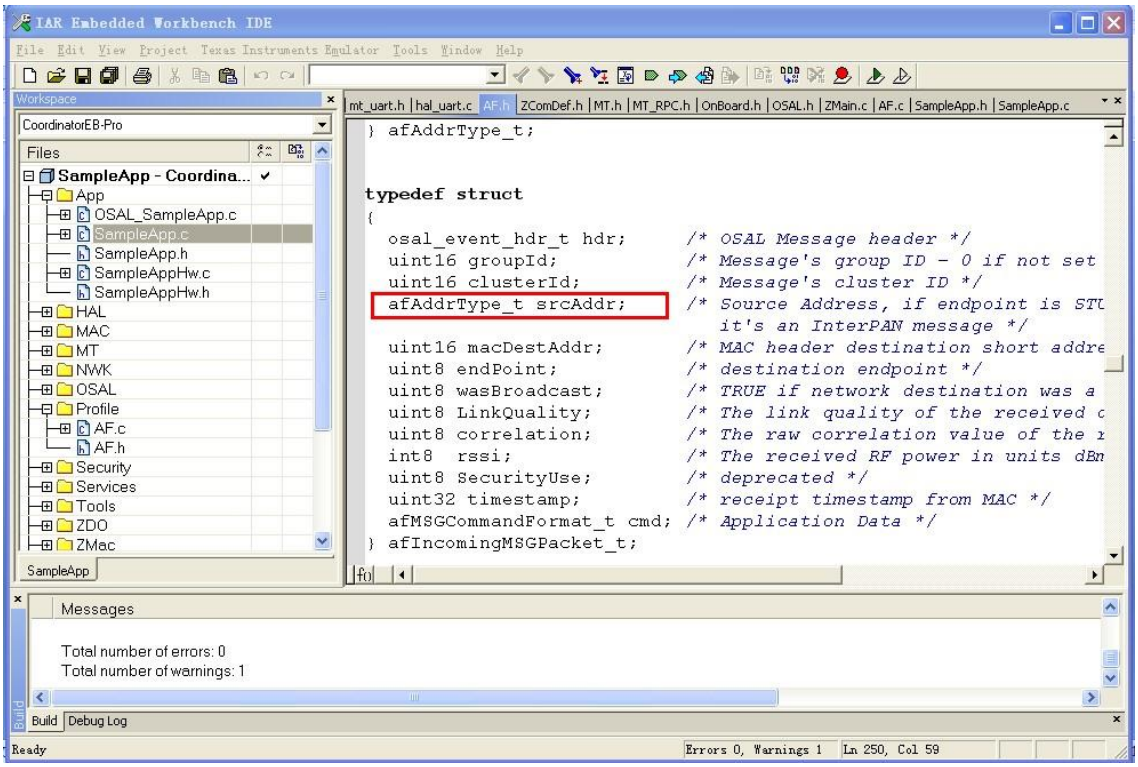

IAR Embedded Workbench IDE
File Edit View Project Texas Instruments Emulator Tools Window Help
Workspace
CoordinatorEB-Pro
Files
SampleApp - Coordina...
App
OSAL_SampleApp.c
SampleApp.c
SampleApp.h
SampleAppHw.c
SampleAppHw.h
HAL
MAC
MT
NWK
OSAL
Profile
AF.c
AF.h
Security
Services
Tools
ZDO
ZMac
SampleApp
mt_uart.h | hal_uart.c | AF.h | ZComDef.h | MT.h | MT_RPC.h | OnBoard.h | OSAL.h | ZMain.c | AF.c | SampleApp.h | SampleApp.c
} afAddrType_t;
typedef struct
{
osal_event_hdr_t hdr; /* OSAL Message header */
uint16 groupId; /* Message's group ID - 0 if not set
uint16 clusterId; /* Message's cluster ID */
afAddrType_t srcAddr; /* Source Address, if endpoint is STU
it's an InterPAN message */
uint16 macDestAddr; /* MAC header destination short addre
uint8 endPoint; /* destination endpoint */
uint8 wasBroadcast; /* TRUE if network destination was a
uint8 LinkQuality; /* The link quality of the received c
uint8 correlation; /* The raw correlation value of the r
int8 rssi; /* The received RF power in units dBm
uint8 SecurityUse; /* deprecated */
uint32 timestamp; /* receipt timestamp from MAC */
afMSGCommandFormat_t cmd; /* Application Data */
} afIncomingMSGPacket_t;
Messages
Total number of errors: 0
Total number of warnings: 1
Build
Debug Log
Ready
Errors 0, Warnings 1
Ln 250, Col 59

图 3.91

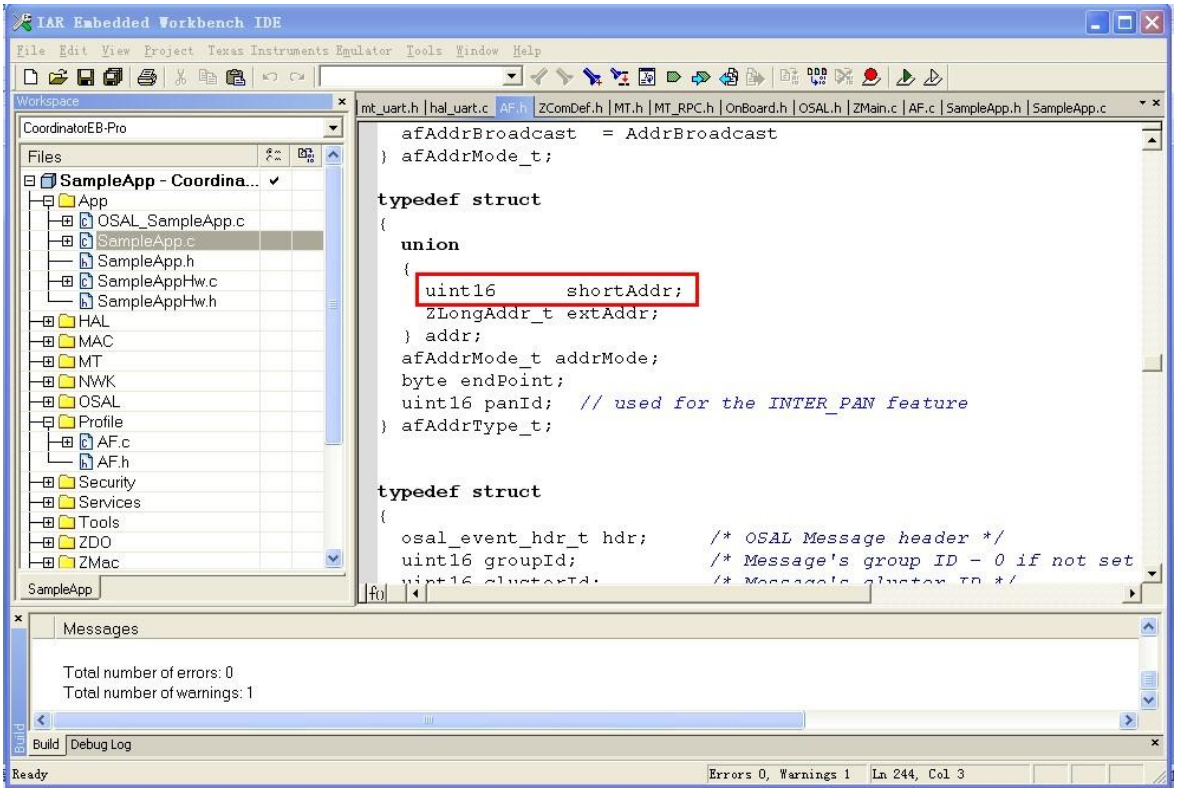

IAR Embedded Workbench IDE
File Edit View Project Texas Instruments Emulator Tools Window Help
Workspace
CoordinatorEB-Pro
Files
SampleApp - Coordina...
App
OSAL_SampleApp.c
SampleApp.c
SampleApp.h
SampleAppHw.c
SampleAppHw.h
HAL
MAC
MT
NWK
OSAL
Profile
AF.c
AF.h
Security
Services
Tools
ZDO
ZMac
SampleApp
mt_uart.h | hal_uart.c | AF.h | ZComDef.h | MT.h | MT_RPC.h | OnBoard.h | OSAL.h | ZMain.c | AF.c | SampleApp.h | SampleApp.c
afAddrBroadcast = AddrBroadcast
} afAddrMode_t;
typedef struct
{
union
{
uint16 shortAddr;
ZLongAddr_t extAddr;
} addr;
afAddrMode_t addrMode;
byte endPoint;
uint16 panId; // used for the INTER_PAN feature
} afAddrType_t;
typedef struct
{
osal_event_hdr_t hdr; /* OSAL Message header */
uint16 groupId; /* Message's group ID - 0 if not set
Messages
Total number of errors: 0
Total number of warnings: 1
Build
Debug Log
Ready
Errors 0, Warnings 1
Ln 244, Col 3

图 3.92

我们可以在接收函数中点播 ID 加入下面代码：

```
uint16    flashTime,temp;
  //网蜂 16 进制转 ASCII 码表
uint8 asc_16[16]={'0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F'};
case    SAMPLEAPP_POINT_TO_POINT_CLUSTERID:
        temp=pkt->srcAddr.addr.shortAddr; //读出数据包的 16 位短地址

      if( pkt->cmd.Data[0]==1 ) //路由器
        HalUARTWrite(0,"ROUTER ShortAddr:0x",19); //提示接收到数据
      if( pkt->cmd.Data[0]==2 ) //终端
        HalUARTWrite(0,"ENDDEVICE ShortAddr:0x",22); //提示接收到数据

        /****将短地址分解，ASC 码打印*****/
        HalUARTWrite(0,&asc_16[temp/4096],1);
        HalUARTWrite(0,&asc_16[temp%4096/256],1);
        HalUARTWrite(0,&asc_16[temp%256/16],1);
        HalUARTWrite(0,&asc_16[temp%16],1);
        HalUARTWrite(0,"\n",1);                    // 回车换行
  break;
```

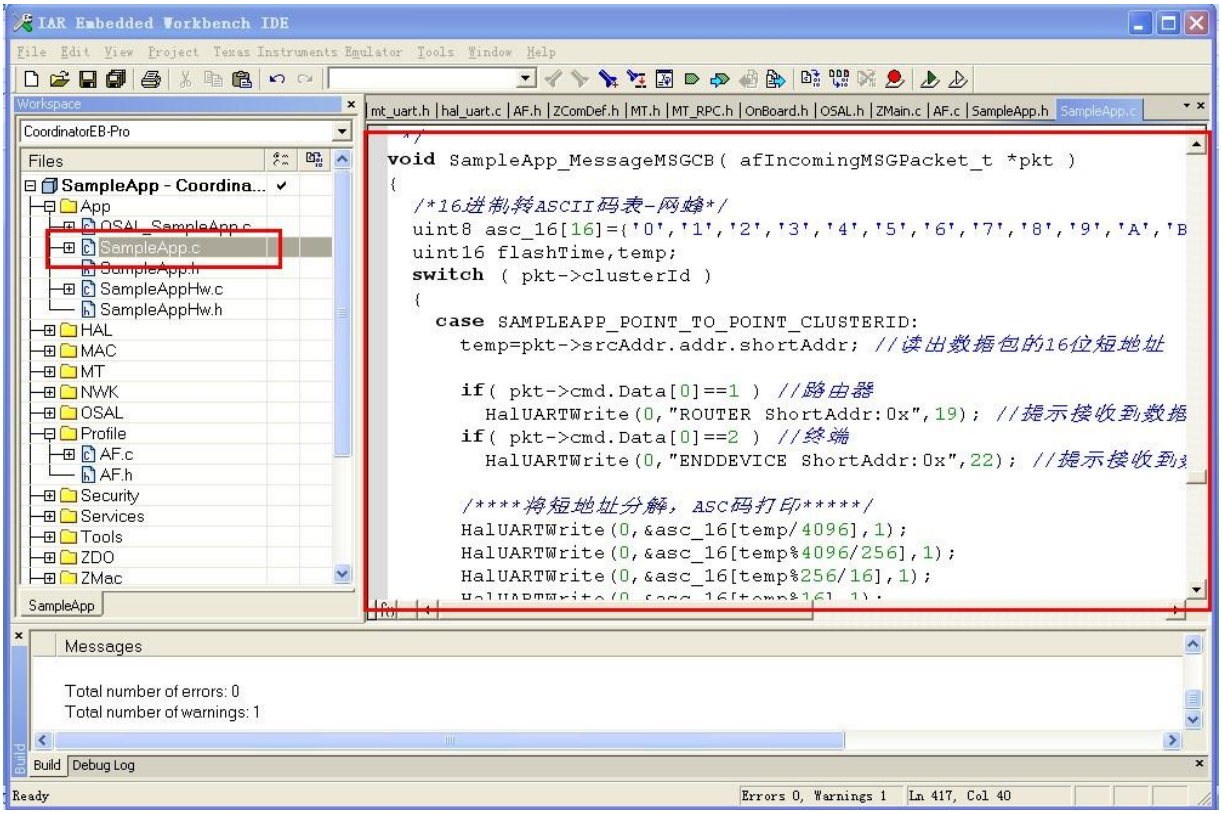


图 3.93

实验结果：将修改后的程序分别以协调器、路由器、终端的方式下载到 3 个 或以上设备，协调器连接到 PC 机。上电后每个设备往协调器发送自身编号，协调器通过串口打印出来。

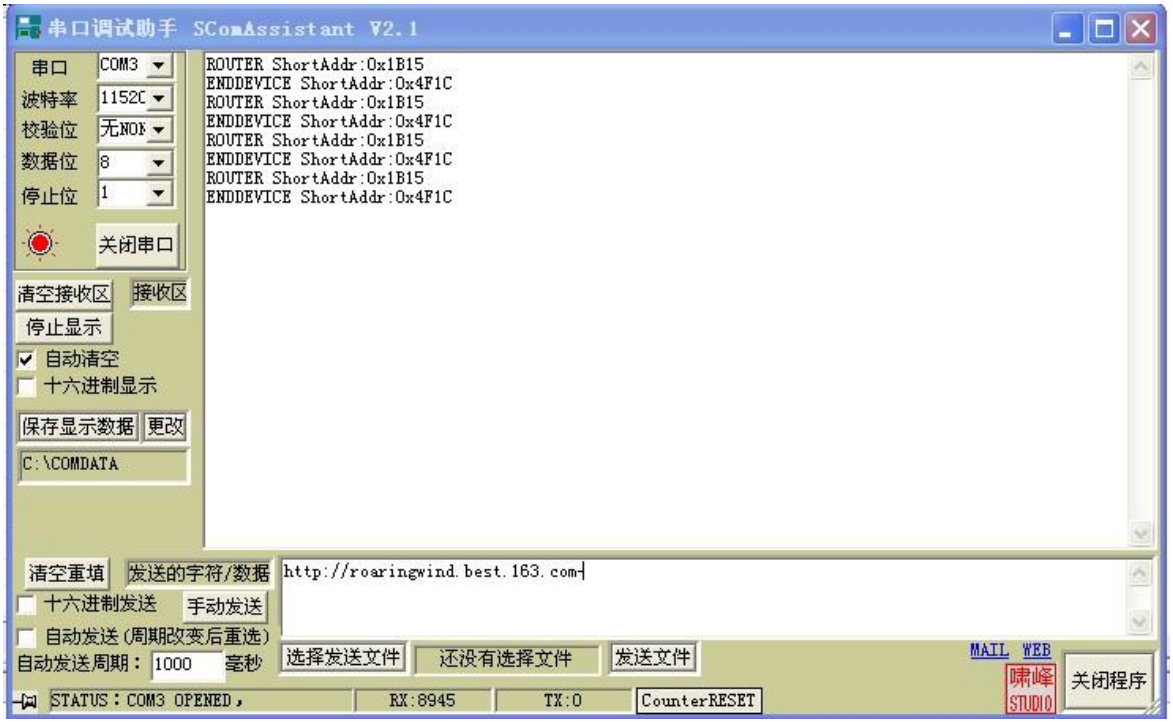


图 3.94

在这里网蜂提供了一个利用组网后短地址进行网络管理方法，有兴趣的可以利用同样的方法可以将 MAC 地址、PANID 等读取出来。或者自行设定预定义节点编号进行网络管理。

3.11 传感器应用

3.11.1 温度传感器 DS18B20

前言：通过前面的组网例子，相信大家对 ZigBee 已经有一定了解。现在大家应该思考一下，我们学习 zigbee 的目的是什么，仅仅是为了学会强大的组网功能吗？我们会发现，学习 zigbee 的最终目的是采集传感器信息，建立起无线传感网。从这一节开始，网蜂将带领大家进入传感器的世界，教会大家用 zigbee 来玩转常用的传感器，我们选择从大家很熟悉且常用的温度传感器 DS18B20 开始。逐步建立自己的无线传感网！

传感器介绍：

DS18B20 数字温度传感器接线方便，封装成后可应用于多种场合，如管道式，螺纹式，磁铁吸附式，不锈钢封装式，主要根据应用场合的不同而改变其外观。封装后的 DS18B20 可用于电缆沟测温，高炉水循环测温，锅炉测温，机房测温，农业大棚测温，洁净室测温，弹药库测温等各种非极限温度场合。耐磨耐碰，体积小，使用方便，封装形式多样，适用于各种狭小空间设备数字测温和控制领域。实物如图所示：

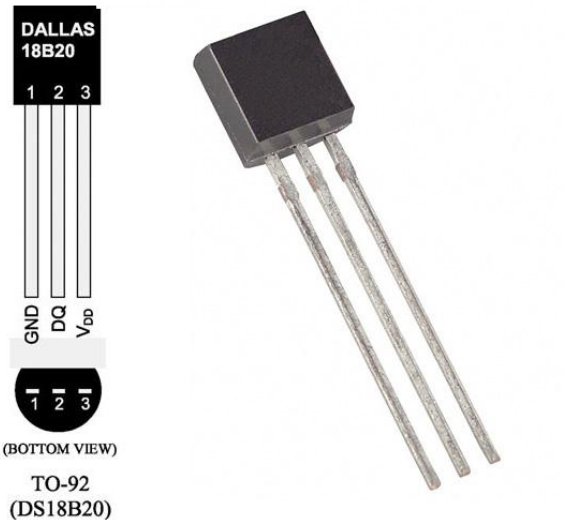


图 3.95 DS18B20 外观

实现平台：网蜂物联网 ZigBee 开发平台、ZigBee 传感器节点；

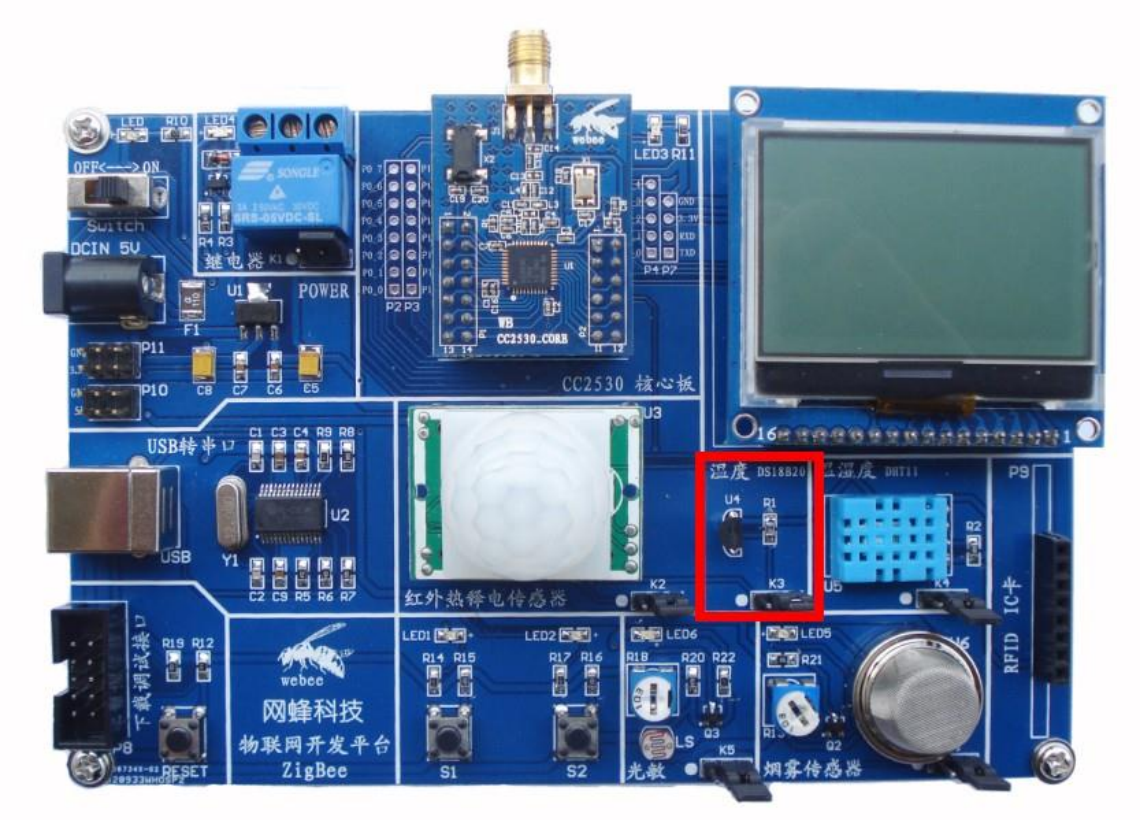


图 3.96 网蜂物联网 ZigBee 开发平台

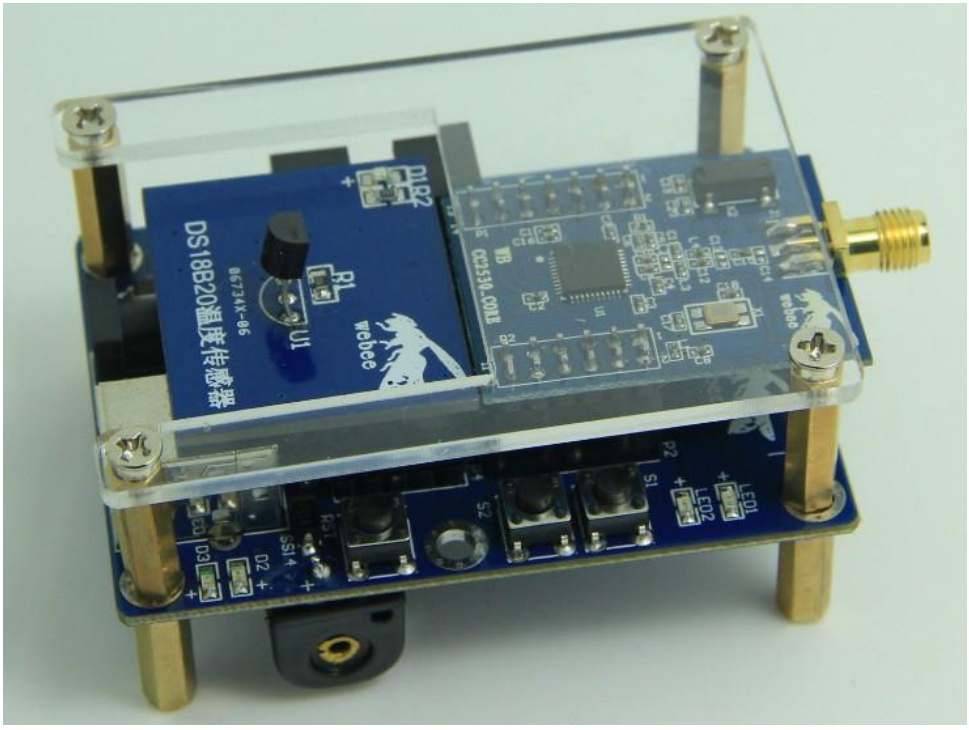


图 3.97 ZigBee 传感器器节点

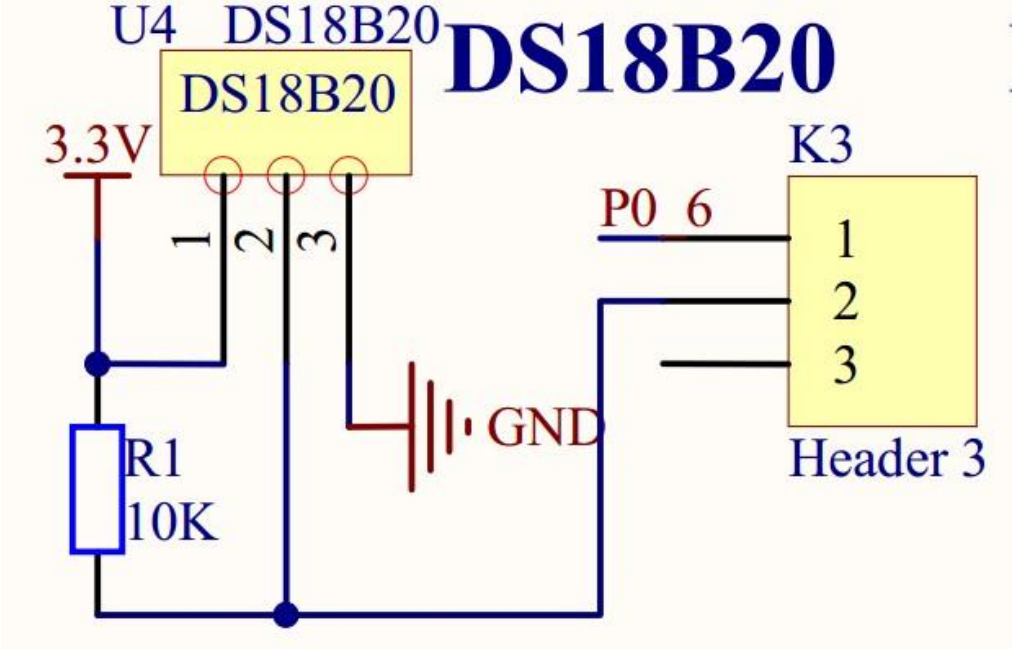


图 3.98 DS18B20 硬件电路图

实验现象：节点通过采集 DS18B20 温度信息，实时发送到协调器。协调器通过串口打印和液晶显示方式展示当前温度。

实验讲解：ZigBee 要完成采集温度传感器信息再发送到协调器的过程，必须在协议栈上完成所有代码的编程。实际上，我们可以在裸机（不带协议栈）的基础上成功驱动起传感器，然后再加载到协议栈上面。这样会有事半功倍的效果。下面我们就来看看如何把裸机上成功驱动的传感器添加到协议栈代码上，并实现数据传输。

实验过程：分三个步骤，如下：

一：在裸机上完成对 DS18B20 的驱动。

二：将程序添加到协议栈代码中

三：将数据打包并按指定的方式发送给指定设备。

一：在裸机上完成对 DS18B20 的驱动。

相信大家都接触过 51 单片机对 DS18B20 温度读取的编程。CC2530 裸机驱

动 DS18B20 也是这样的道理。只要我们掌握好单片机 C 语言编程以及了解传感器的原理就可以轻松应对。

打开配套程序下裸机文件夹—温度传感器 DS18B20 下的工程文件，看到主函数如下：（代码取用模块化编程，其他函数请看工程文件）

```
/*************************************/
/*            WeBee 团队             */
/*         Zigbee 学习例程           */
/*例程名称：温度传感器 DS18B20        */
/*建立时间：2012/10/1                */
/*描述：将采集到的温度信息通过串口打印到
        串口调试助手。
*************************************/
```

```
1.  #include "iocc2530.h"
2.  #include "uart.h"
3.  #include "ds18b20.h"
4.  #include "delay.h"

5.  void Initial()              //系统初始化
6.  {
7.    CLKCONCMD = 0x80;         //选择 32M 振荡器
8.    while(CLKCONSTA&0x40);    //等待晶振稳定
9.    UartInitial();            //串口初始化
10.   P0SEL &= 0xbf;            //DS18B20 的 io 口初始化
11. }
12. void main()
13. {
```

```
14.    char data[5]="temp="; //串口提示符
15.    Initial();
16.    while(1)
17.   {
18.      Temp_test();   //温度检测

   /*******温度信息打印 **********/
19.      UartTX_Send_String(data,5);
20.      UartSend(temp/10+48);
21.      UartSend(temp%10+48);
22.      UartSend('\n');
23.
24.      Delay_ms(1000);  //延时函数使用定时器方式，延时 1S
25.    }
26. }
```

我们来看主函数：

第 15 行：进行一些初始化工作。

第 18 行：在大循环中，检测温度。

第 19~22 行：通过串口打印温度信息

简单几行代码，就完成了对 DS18B20 的读取。大家可以在工程里进入具体函数看代码，理解 DS18B20 的读取过程。实验现象如图 **3.11.1 D** 所示：

图 3.99

二：将程序添加到协议栈代码中

有了基础实验的代码，我们的实验就完成了一大半了。至少证明 CC2530 可以驱动起我们想要的传感器。接下来我们需要做的工作就是移植到协议栈 z-stack 上面，这个过程要注意的是要了解协议栈上的 IO 口用途和晶振工作频率。

首先理清一下思路，我们要实验的功能是终端设备读取 DS18B20 温度信息，通过点播方式发送到协调器，协调器通过通常打印出来。在串口调试助手上面显示。这就实现了无线温度采集。（使用点播的原因是终端设备有针对性地发送数据给指定设备，不想广播和组播可能会造成数据冗余，关于点播内容请参考《zigbee 实战演练》点播章节，这里不再累赘。）

1）我们将裸机程序里面的 DS18B20.c 和 DS18B20.h 文件复制到 SAMPLEAPP-Source 文件夹下。

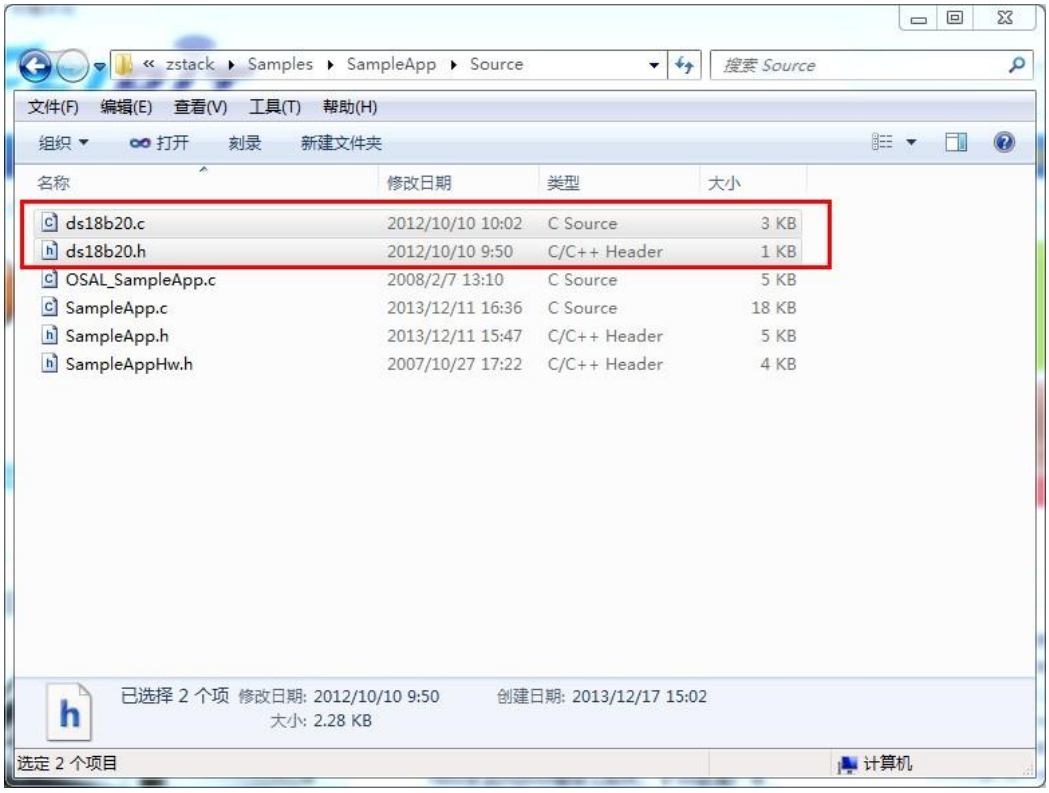


图 3.100

2）在协议栈的 APP 目录树下点击右键--Add--添加 DS18B20.C 文件

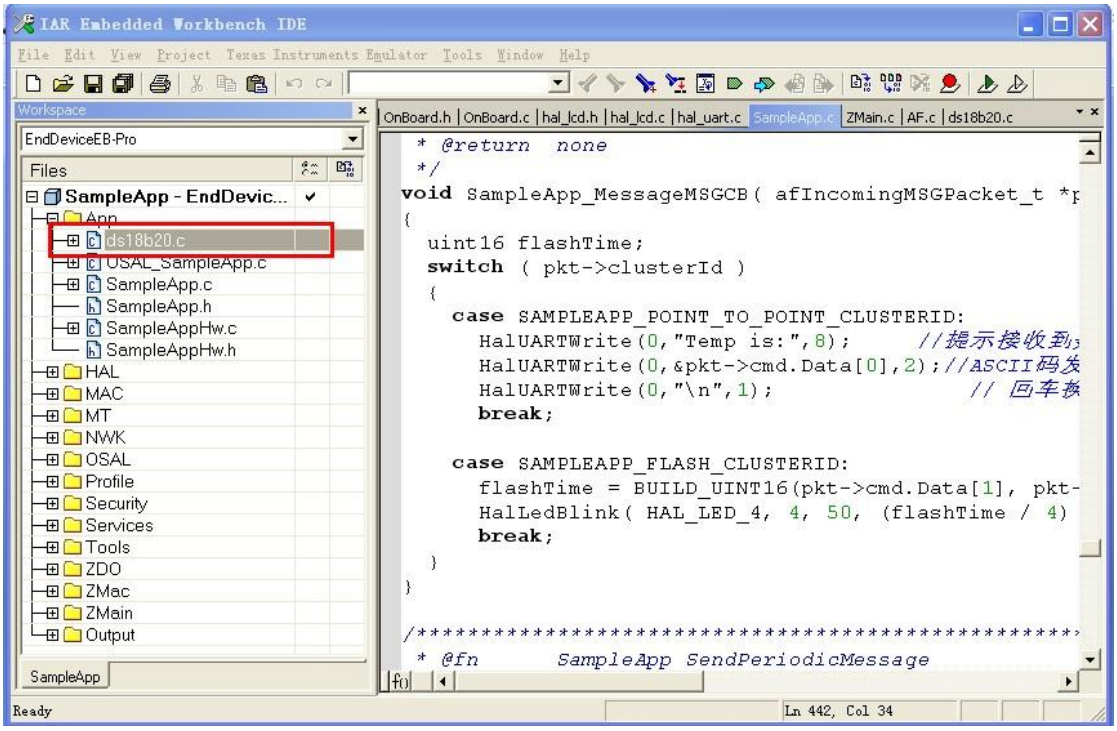


图 3.101 文件列表添加 c 文件

3）整个实验以点播为依托，我们实验也就是在点播例程的基础上完成，故函数编程也是像以前一样在 SAMPLEAPP.C 上进行。我们先包含 DS18B20.h 文件。

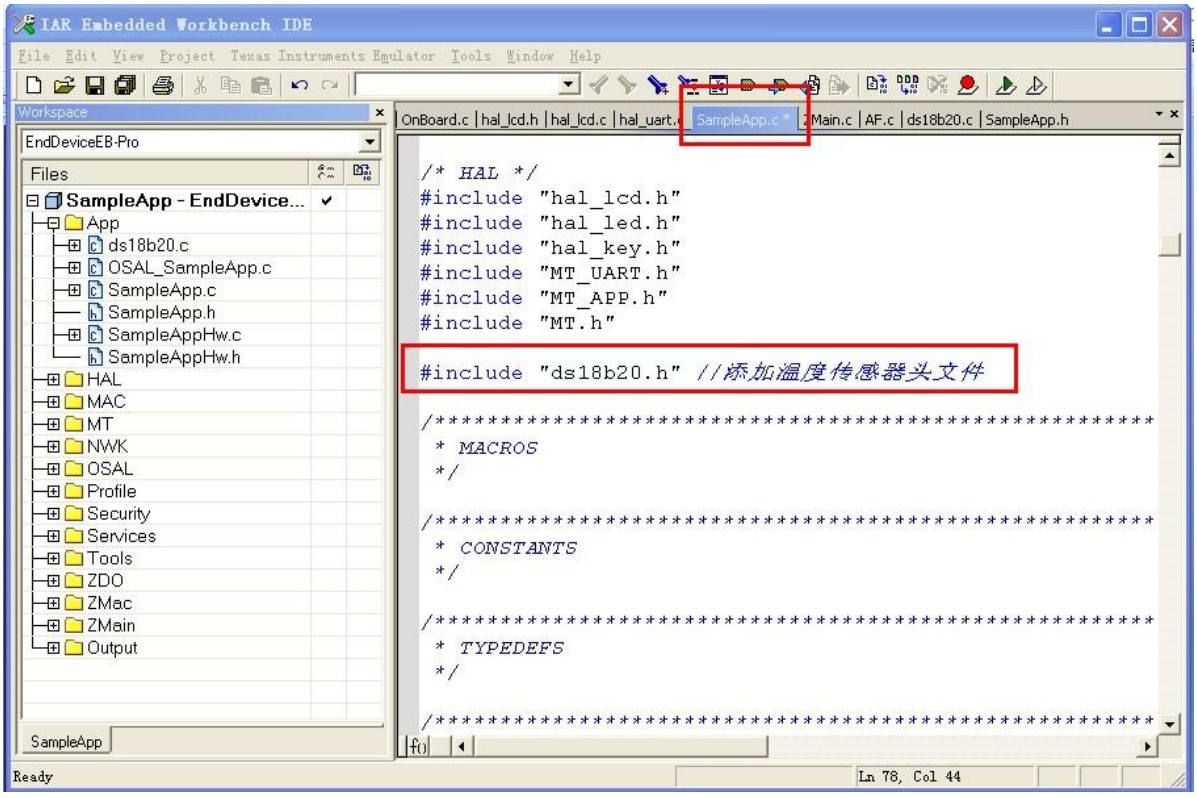


图 3.102 添加头文件

4）初始化传感器引脚 P0.6 。

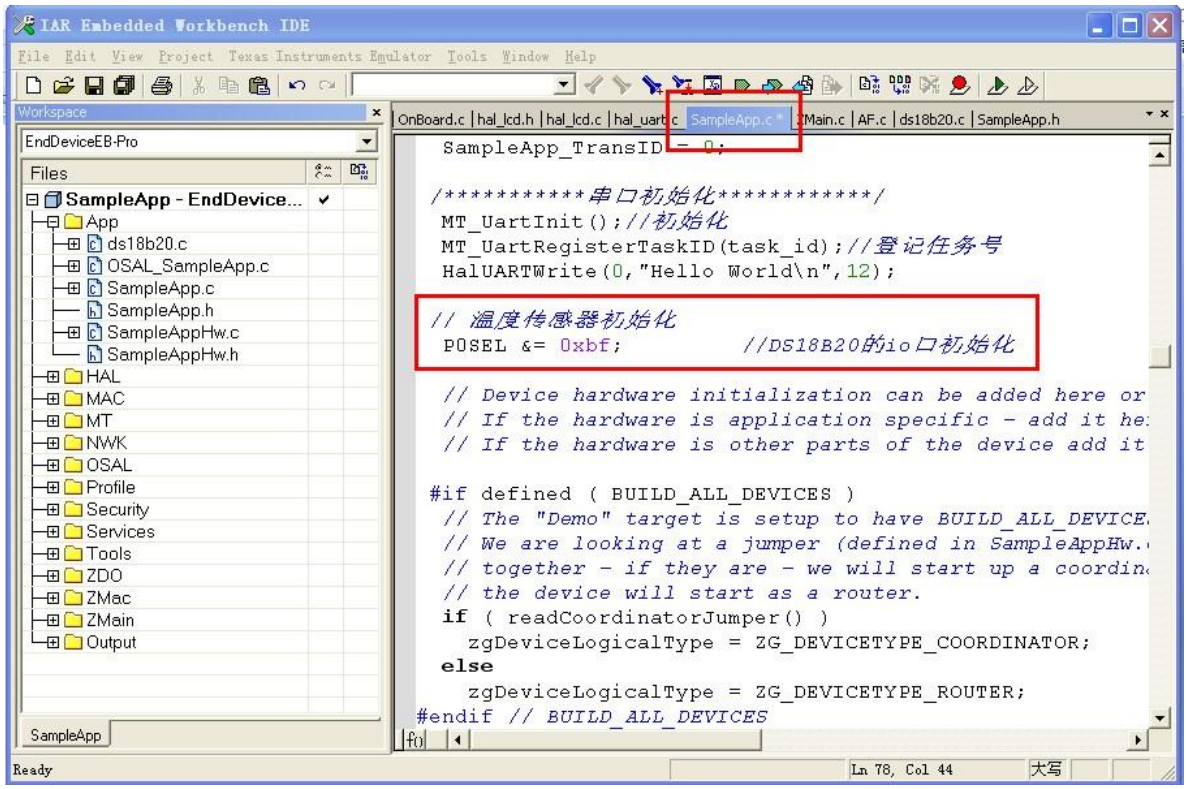


图 3.103

4）借用周期性点播函数，1s 读取温度传感器 1 次，通过液晶显示和串口打印并点对点发送给协调器 。代码如下，如图 3.104 所示：

```
 uint8 T[5];     //温度+提示符
Temp_test();    //温度检测
T[0]=temp/10+48;
T[1]=temp%10+48;
T[2]=' ';
T[3]='C';
T[4]='\0';
/*******串口打印 WEBEE*********/
HalUARTWrite(0,"temp=",5);
HalUARTWrite(0,T,2);
HalUARTWrite(0,"\n",1);
```

```
/******LCD 显示   WEBEE*********/
HalLcdWriteString("The temp is:", HAL_LCD_LINE_3 ); //LCD 显示
HalLcdWriteString( T, HAL_LCD_LINE_4 );              //LCD 显示
```

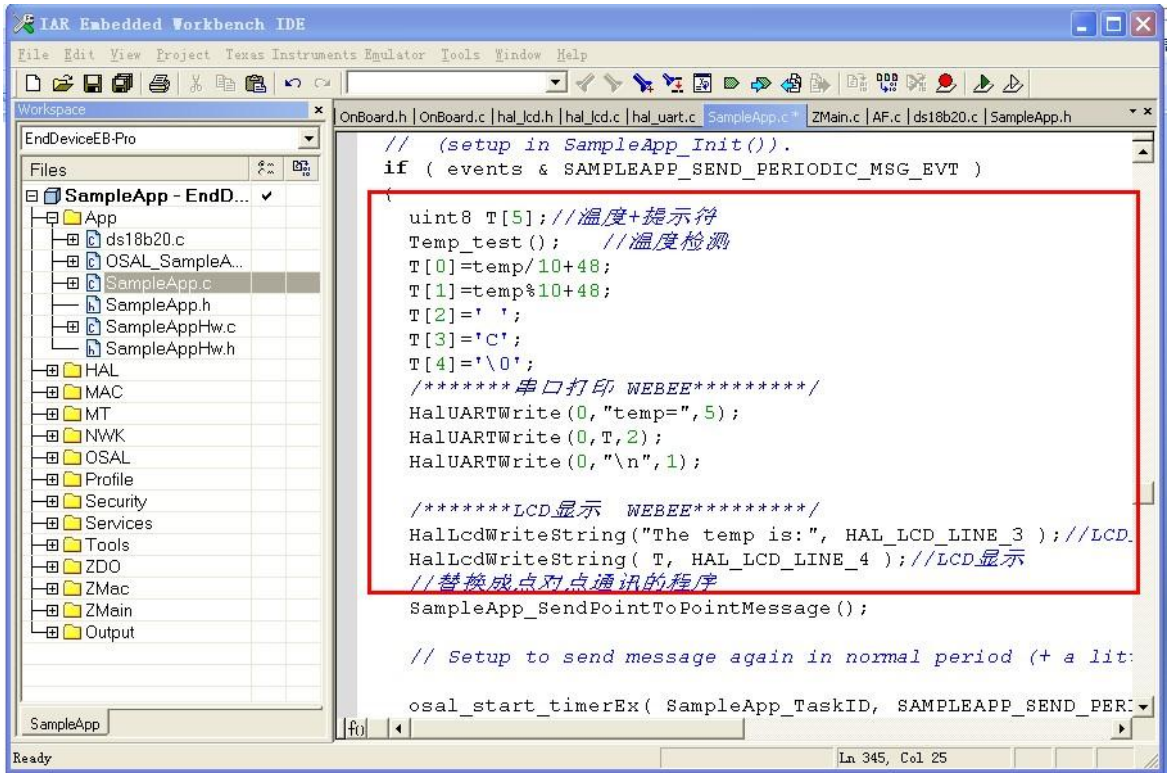


图 3.104

5） DS18B20.c 文件需要修改一个地方。打开改文件，将原来的延时函数改成协议栈自带的延时函数，保证时序的正确。

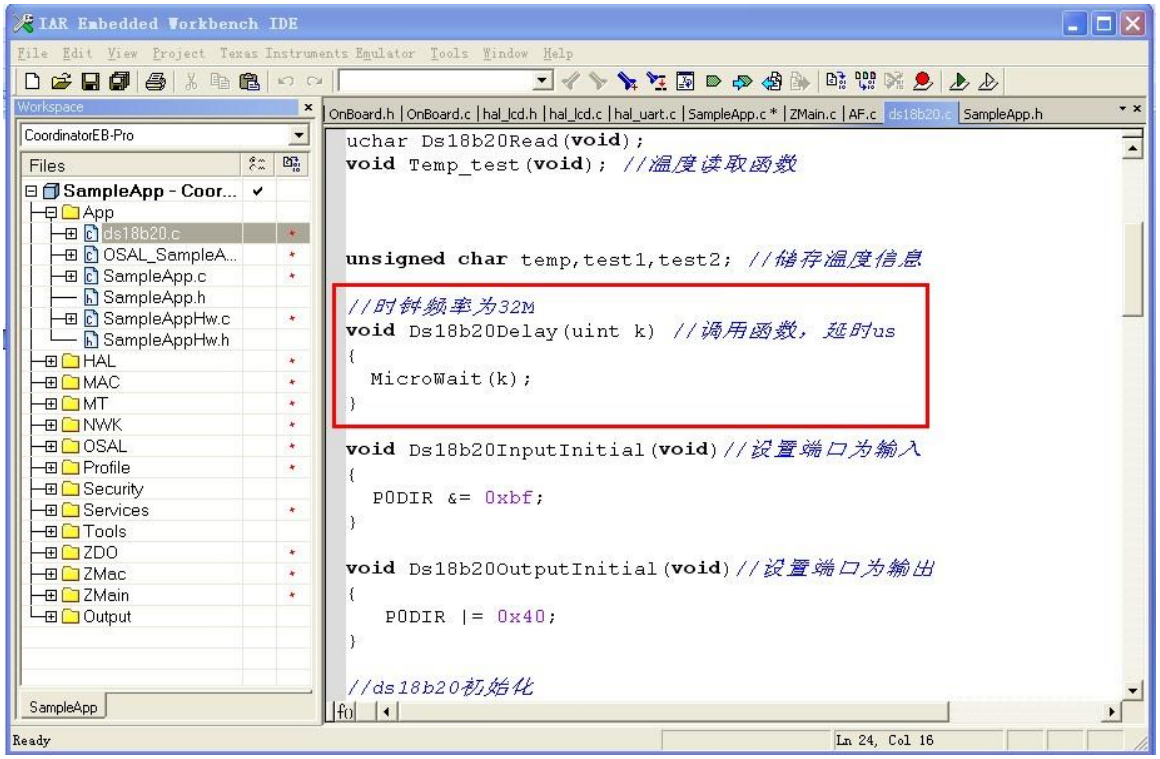


图 3.105

同时要包含 #include"OnBoard.h" 。去掉 #include"uart.h" ；

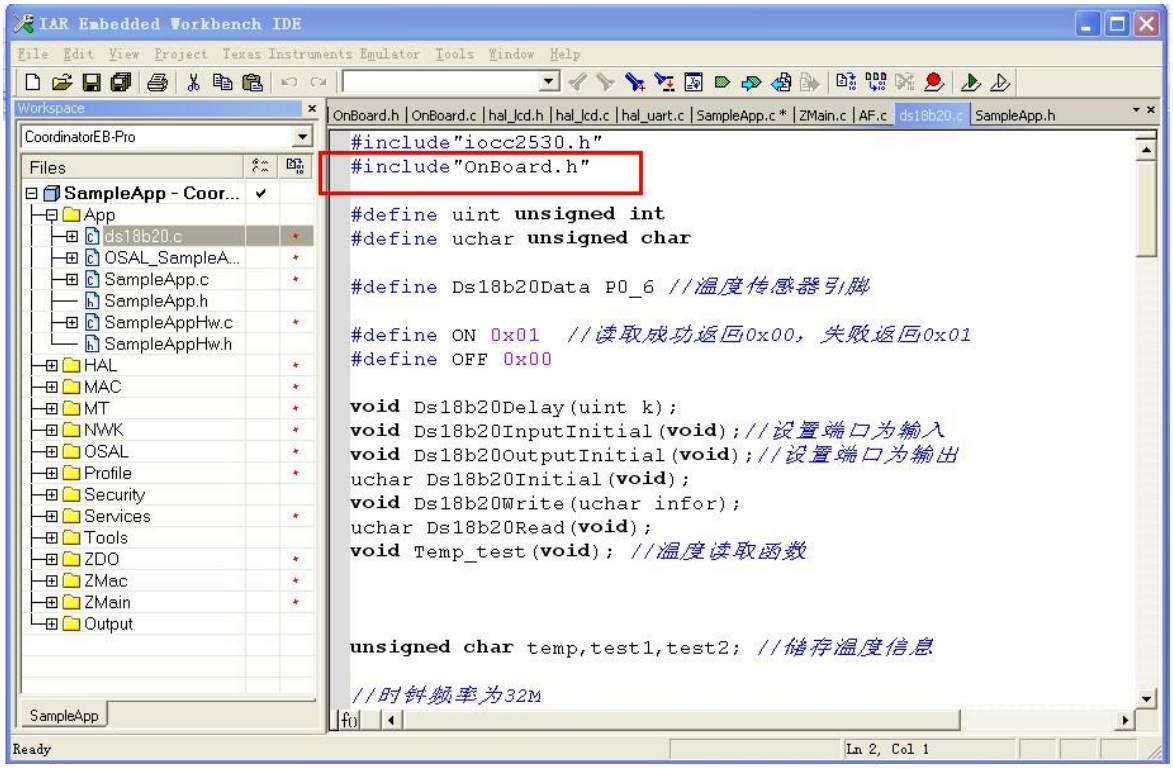


图 3.106

另外，协议栈里面是要求每个定义了的函数都必须在文件添加声明，否则报错。所以给 ds18b20.c 里面的文件都添加声明：

```
void Ds18b20Delay(uint k);
void Ds18b20InputInitial(void);//设置端口为输入
void Ds18b20OutputInitial(void);//设置端口为输出
uchar Ds18b20Initial(void);
void Ds18b20Write(uchar infor);
uchar Ds18b20Read(void);
void Temp_test(void); //温度读取函数
```

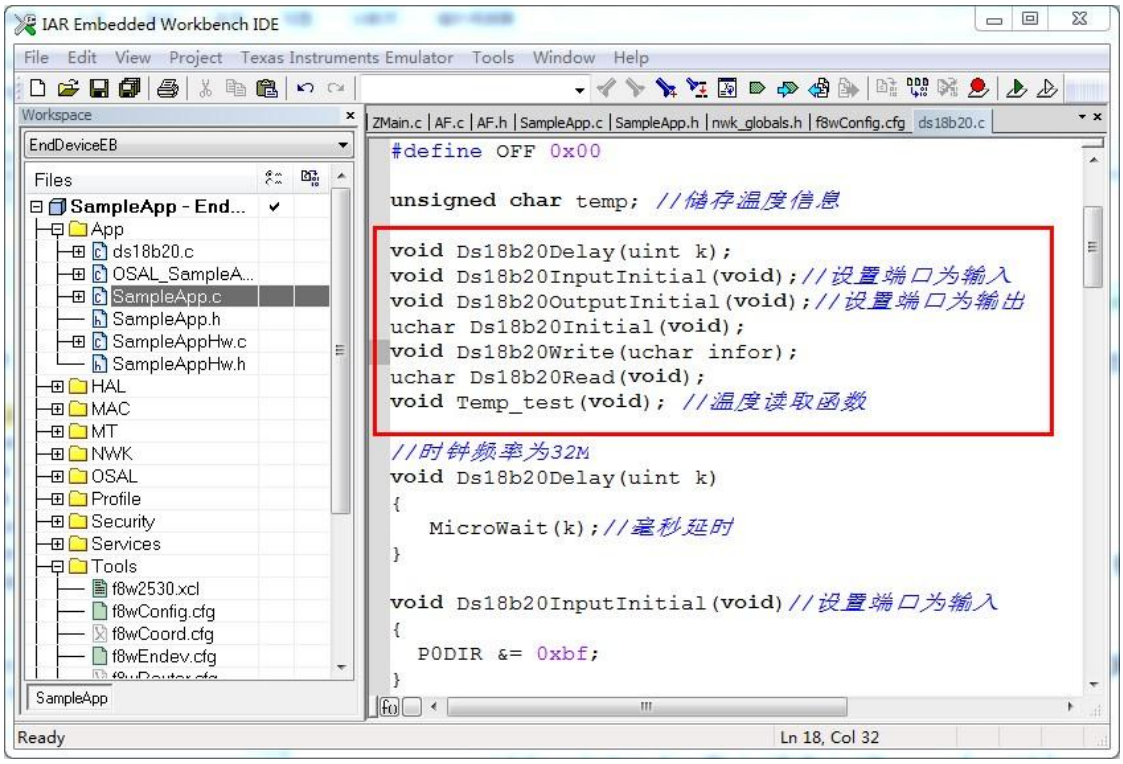


图 3.107

三：将数据打包并按指定的方式发送给指定设备。

在上一个步骤，我们完成了 DS18B20 基于协议栈的驱动，接下来，我们就用以前的知识，实现数据发送和接收就可以了。

在 EndDevice 的点播发送函数中将温度信息发送出去，代码如下，如图所示：

```
void SampleApp_SendPointToPointMessage( void )
```

```
{
  uint8  T[2];      //温度
  T[0]=temp/10+48; //格式转换
  T[1]=temp%10+48;
  if ( AF_DataRequest( &Point_To_Point_DstAddr,
                       &SampleApp_epDesc,
                       SAMPLEAPP_POINT_TO_POINT_CLUSTERID,
                       2,
                       T,
                       &SampleApp_TransID,
                       AF_DISCV_ROUTE,
                       AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
  {
  }
  else
  {
    // Error occurred in request to send.
  }
}
```

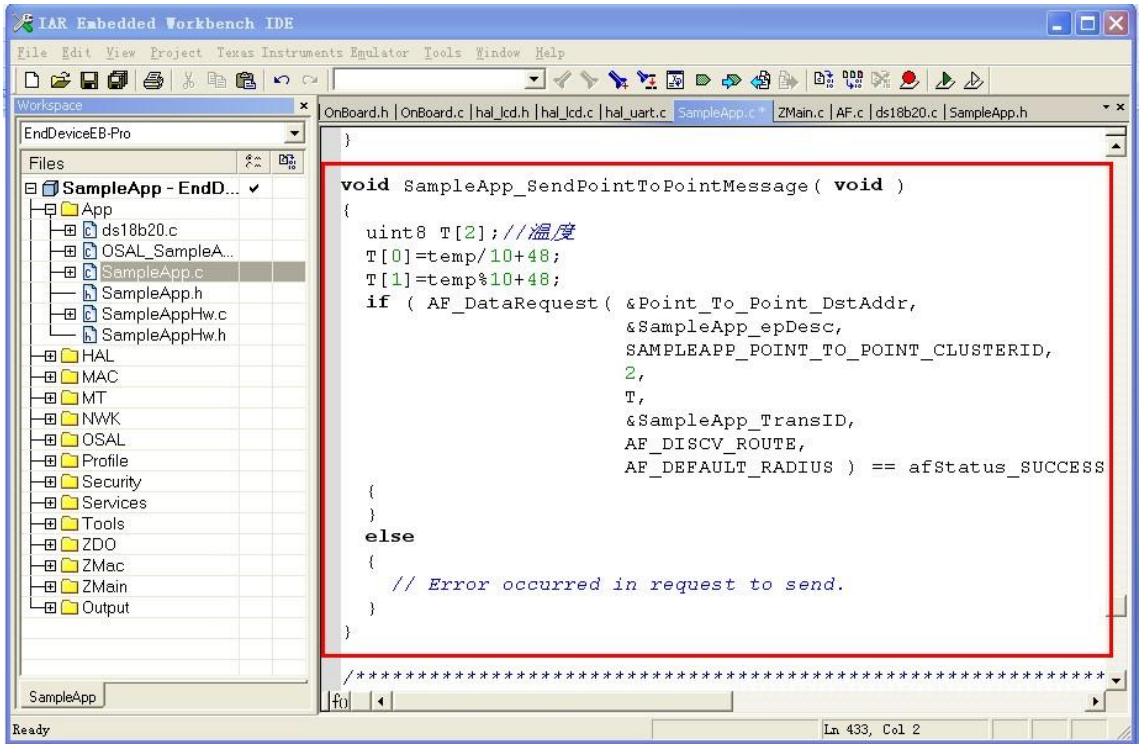


图 3.108

协调器代码如下，如图 3.11.1 M 所示：

```
case  SAMPLEAPP_POINT_TO_POINT_CLUSTERID:

  HalUARTWrite(0,"Temp is:",8);          //提示接收到数据
  HalUARTWrite(0,&pkt->cmd.Data[0],2); //ASCII 码发给 PC 机
  HalUARTWrite(0,"\n",1);                // 回车换行

break;
```

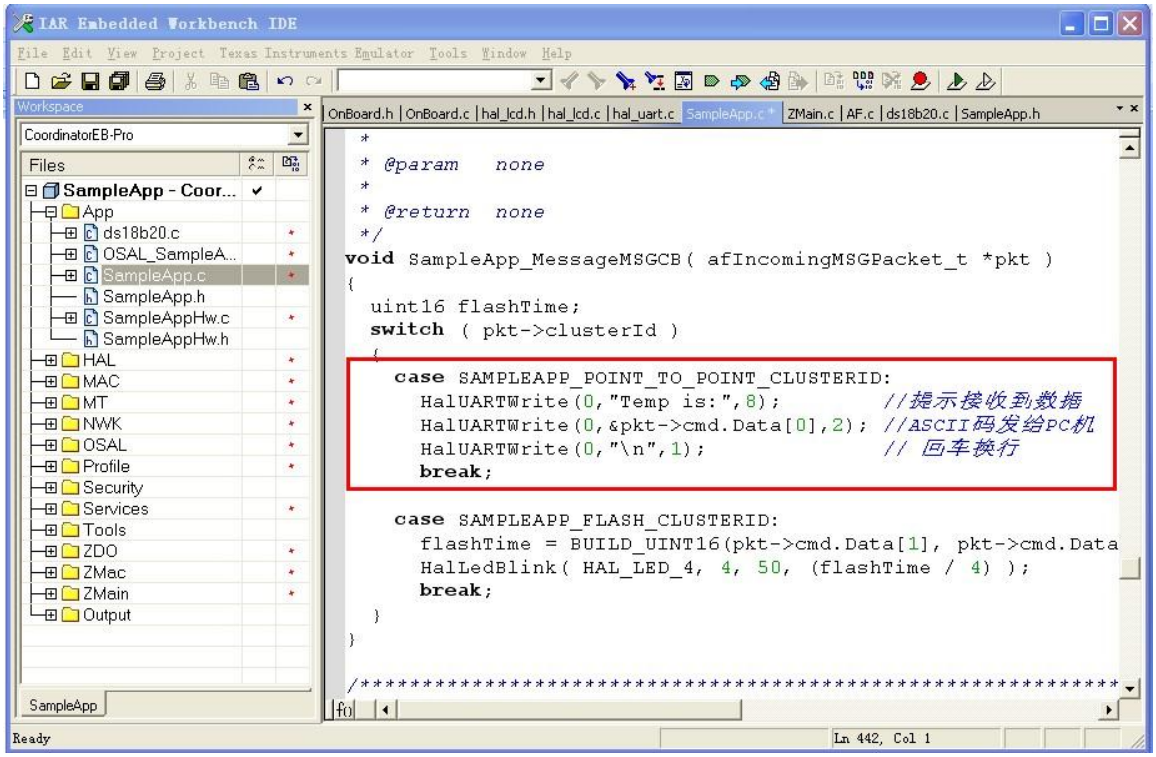


图 3.109

实验结果及图片：

观察终端设备的 LCD 以及将协调器连接到电脑，观察从终端发来的信息。以终端方式下载到开发板，当连接上协调器时，可以看到 LCD 显示当前温度信息，同时串口打印出温度传感器信息。如图所示：


EndDevice:BE80
Parent:0
The temp is:
30 C

图 3.110


串口调试助手 SComAssistant V2.1
串口
COM3
波特率
115200
校验位
无NONE
数据位
8
停止位
1
关闭串口
清空接收区
接收区
停止显示
自动清空
十六进制显示
保存显示数据
更改
C:\COMDATA
temp=27
temp=27
清空重填
发送的字符/数据
http://roaringwind.best.163.com
十六进制发送
手动发送
自动发送(周期改变后重选)
自动发送周期：
1000
毫秒
选择发送文件
还没有选择文件
发送文件
MAIL
WEB
关闭程序
STATUS：COM3 OPENED，
RX:6366
TX:0
CounterRESET

图 3.111

同时协调器收到终端发送来的信息：

图 3.112

结语：

至此，我们完成了对 DS18B20 温度数据的无线采集，也就是从初始化、采集到打包发送的过程。相信看完后你对传感器的应用不再陌生了。从这章开始，网蜂会带领你一步步组建自己的传感器网络。

3.11.2 温湿度度传感器 DHT11

前言：相信大家有了前面 DS18B20 温度传感器的应用后对 zigbee 协议栈上使用传感器不再陌生了，接下来我们再接再厉，马上进行我们下一个传感器温湿度传感器 DHT11 的实验。GO GO GO！

传感器介绍：

DHT11 数字温湿度传感器是一款含有已校准数字信号输出的温湿度复合传感器，它应用专用的数字模块采集技术和温湿度传感技术，确保产品具有极高的可靠性和卓越的长期稳定性。传感器包括一个电阻式感湿元件和一个 NTC 测温元件，并与一个高性能 8 位单片机相连接。因此该产品具有品质卓越、超快响应、抗干扰能力强、性价比极高等优点。每个 DHT11 传感器都在即为精确的湿度校验室中进行校准。校准系数以程序的形式存在 OTP 内存中，传感器内部在检测型号的处理过程中要调用这些校准系数。单线制串行接口，使系统集成变得简易快捷。超小的体积、极低的功耗，信号传输距离可达 20 米以上，使其成为给类应用甚至最为苛刻的应用场合的最佳选择。产品为 4 针单排引脚封装，连接方便。

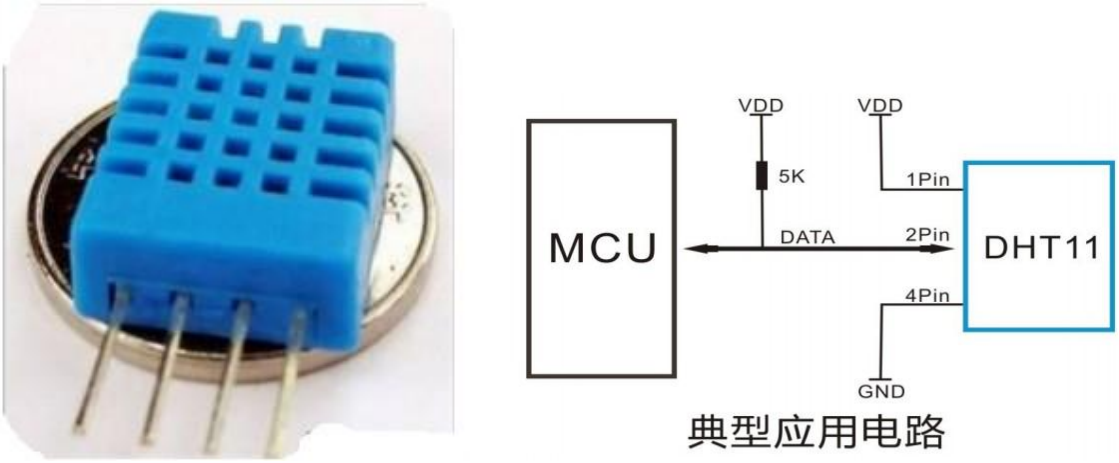


图 3.113 温湿度传感器 DHT11

实现平台：网蜂二代 ZigBee 开发平台，ZigBee 传感器节点；

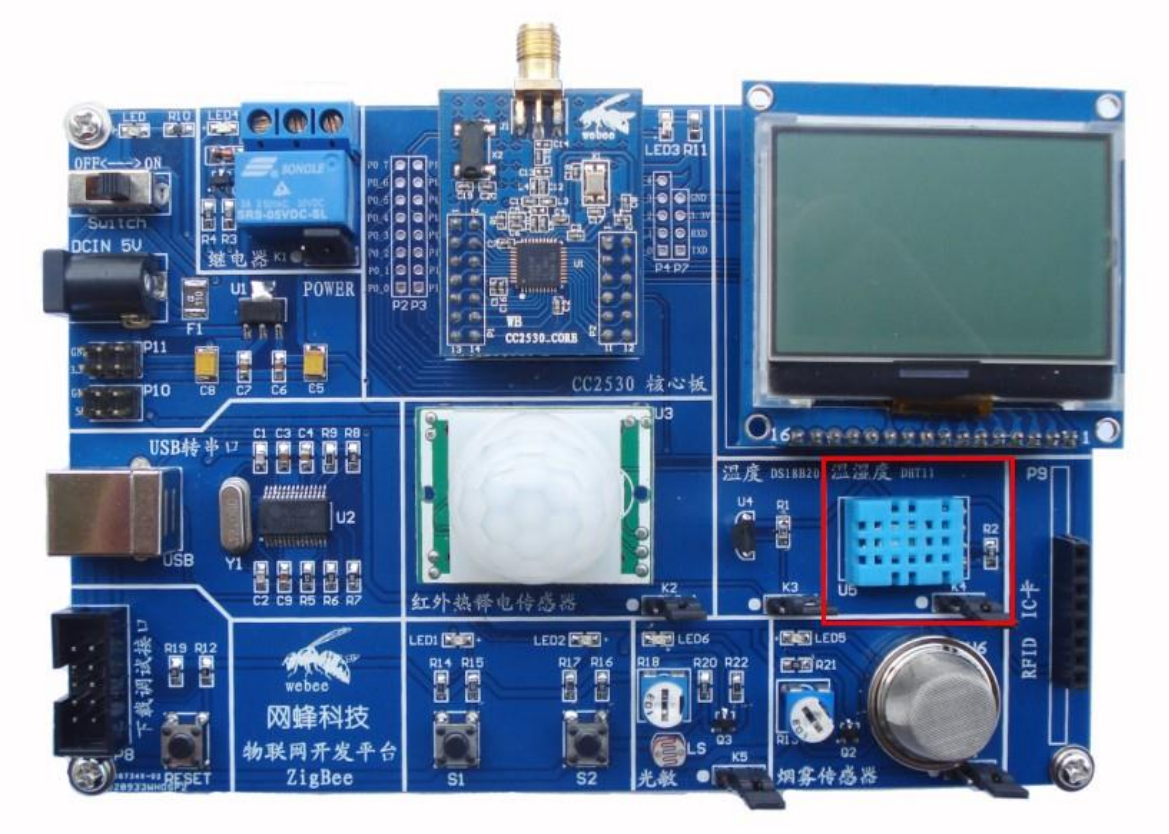


图 3.114　网蜂物联网 ZigBee 开发平台

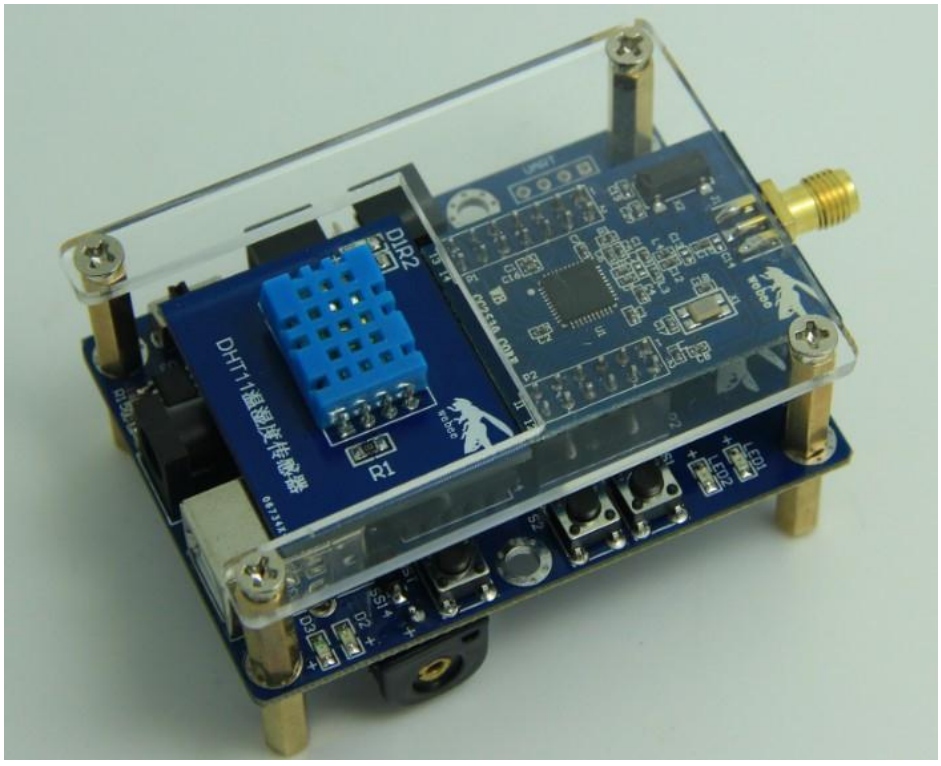


图 3.115　ZigBee 传感器节点

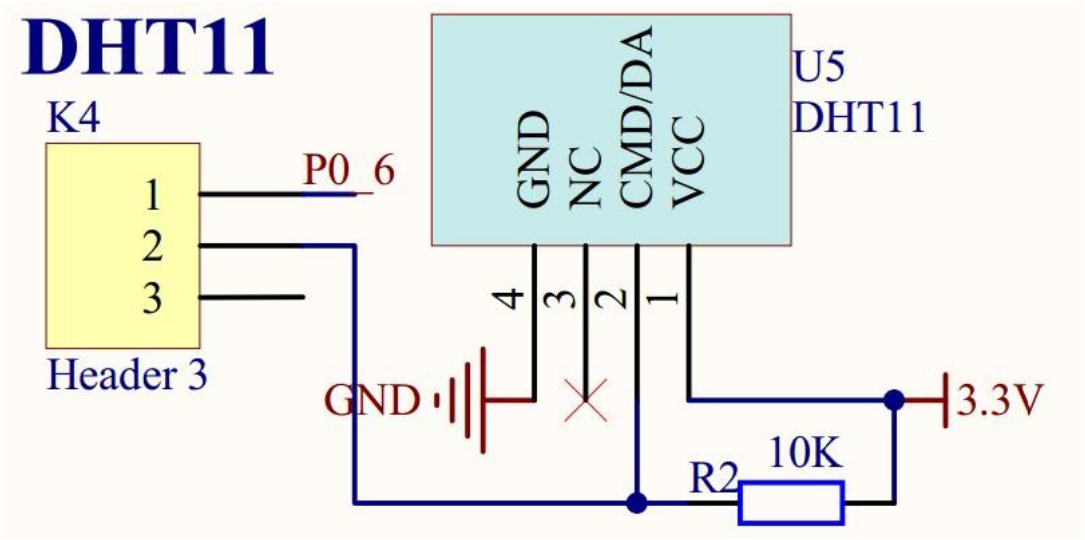


图 3.116 DHT11 硬件电路图

实验现象：节点通过采集 DHT11 的温湿度信息，实时发送到协调器。协调器通过串口打印和液晶显示方式展示当前温湿度。

实验讲解：像前面传感器例程一样，我们先实现裸机驱动 DHT11，然后把裸机上成功驱动的传感器添加到协议栈代码上，并实现数据传输。

实验过程：分三个步骤，如下：

一：在裸机上完成对 DHT11 的驱动。

二：将程序添加到协议栈代码中

三：将数据打包并按指定的方式发送给指定设备。

一：在裸机上完成对 DHT11 的驱动。

打开配套程序下裸机文件夹—温湿度传感器 DHT11 下的工程文件，看到主函数如下：（代码取用模块化编程，其他函数请看工程文件）

```
/*************************************/
/*            WeBee 团队             */
```

```
/*          Zigbee 学习例程                  */
/*例程名称：温湿度传感器 DHT11               */
/*建立时间：2012/10/2                        */
/*描述：将采集到的温湿度信息通过串口打印到
        串口调试助手。
*****************************************/
#include <ioCC2530.h>
#include <string.h>
#include "UART.H"
#include "DHT11.H"

/*****************************
            主函数
*****************************/
void main(void)
{
  Delay_ms(1000); //让设备稳定
  InitUart();     //串口初始化
  while(1)
  {
    DHT11();       //获取温湿度


    /******温湿度的 ASC 码转换*******/
    temp[0]=wendu_shi+0x30;
    temp[1]=wendu_ge+0x30;
    humidity[0]=shidu_shi+0x30;
    humidity[1]=shidu_ge+0x30;

```

```
22.     /*******信息通过串口打印********/
23.     Uart_Send_String(temp1,5);
24.     Uart_Send_String(temp,2);
25.     Uart_Send_String("\n",1);
26.
27.     Uart_Send_String(humidity1,9);
28.     Uart_Send_String(humidity,2);
29.     Uart_Send_String("\n",1);
30.     Delay_ms(2000);  //延时，使周期性 2S 读取 1 次
31.     }
32. }
```

我们来看主函数：

第 10 行：进行一些初始化工作。

第 13 行：在大循环中，检测温度。

第 16~20 行：温湿度的 ASC 码转换

第 23~29 行：信息通过串口打印

同样是简单几行代码，就完成了对 DHT11 的读取。大家可以在工程里进入具体函数看代码，理解 DHT11 的读取过程。实验现象如图 3.117 所示：

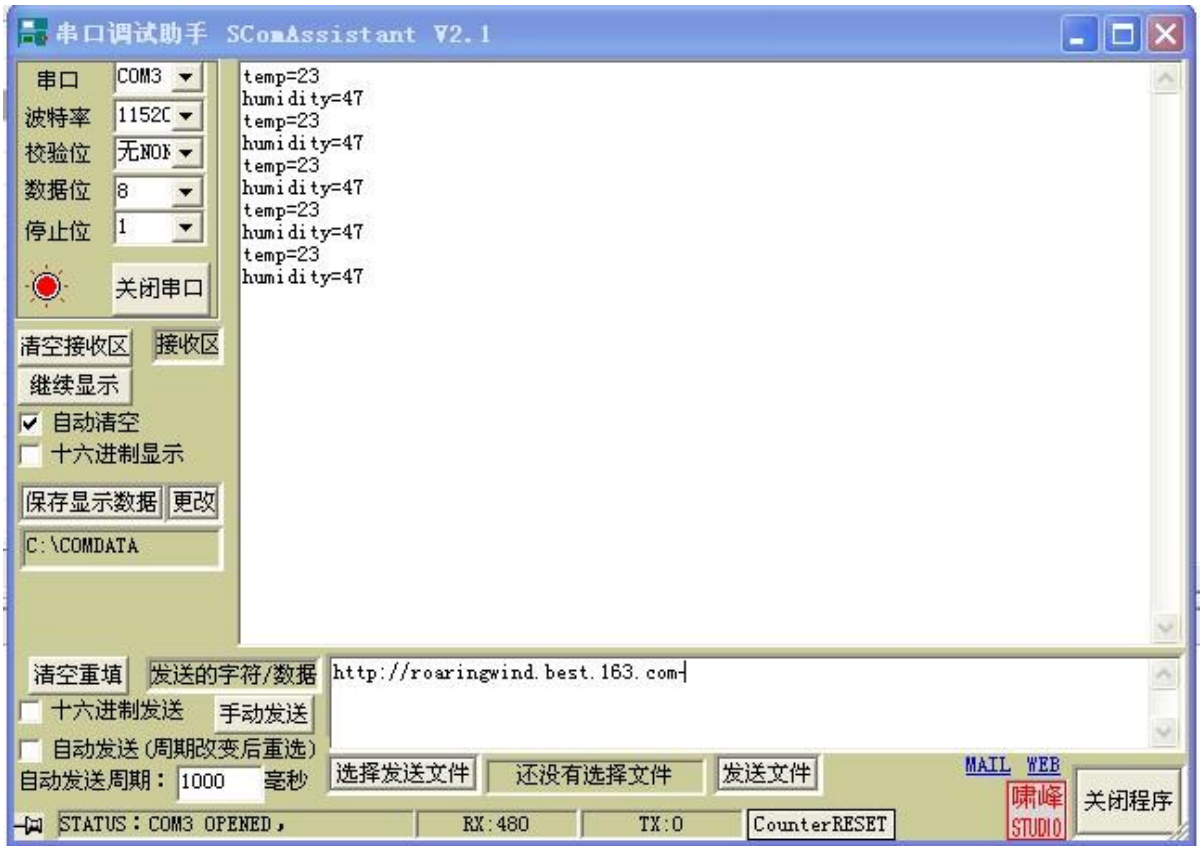


图 3.117

二：将程序添加到协议栈代码中

有了基础实验的代码，我们的实验就完成了一大半了。至少证明 CC2530 可以驱动起我们想要的传感器。接下来我们需要做的工作就是移植到协议栈 z-stack 上面，这个过程要注意的是要了解协议栈上的 IO 口用途和晶振工作频率。

首先理清一下思路，我们要实验的功能是终端设备读取 DHT11 温湿度信息，通过点播方式发送到协调器，协调器通过通常打印出来。在串口调试助手上面显示。这就实现了无线温度采集。（使用点播的原因是终端设备有针对性地发送数据给指定设备，不像广播和组播可能会造成数据冗余，关于点播内容请参考《zigbee 实战演练》点播章节，这里不再累赘。）

1）我们将裸机程序里面的 DHT11.c 和 DHT11.h 文件复制到 SAMPLEAPP --- Source 文件夹下。

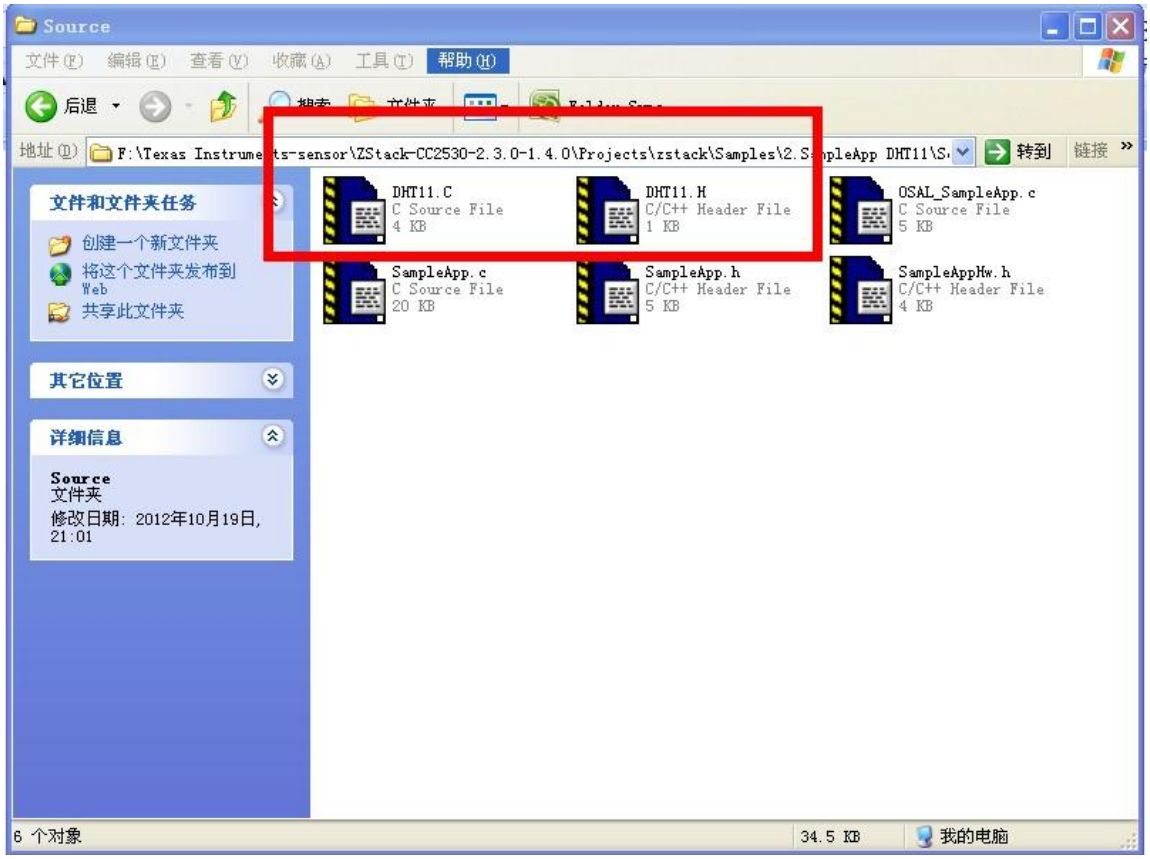


图 3.118

2）在协议栈的 APP 目录树下点击右键--Add--添加 DHT11.C 文件

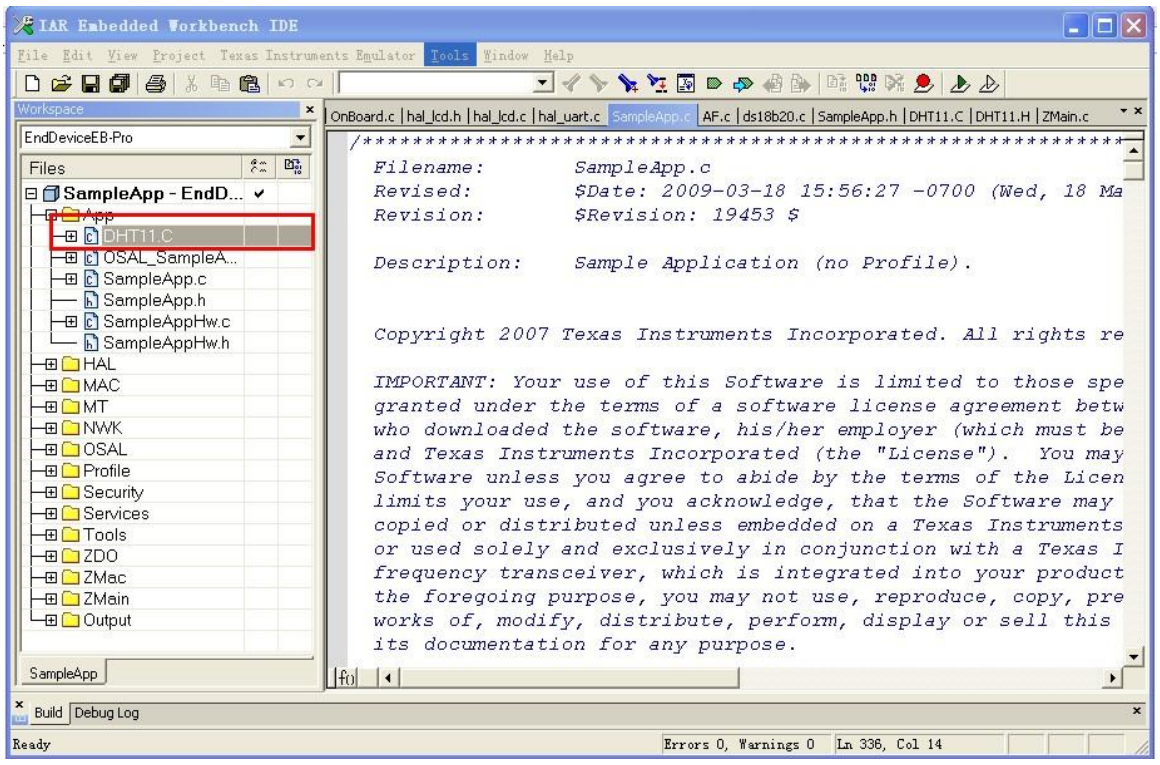


图 3.119

3）整个实验以点播为依托，我们实验也就是在点播例程的基础上完成，故函数编程也是像以前一样在 SAMPLEAPP.C 上进行。我们先包含 DHT11.h 文件。

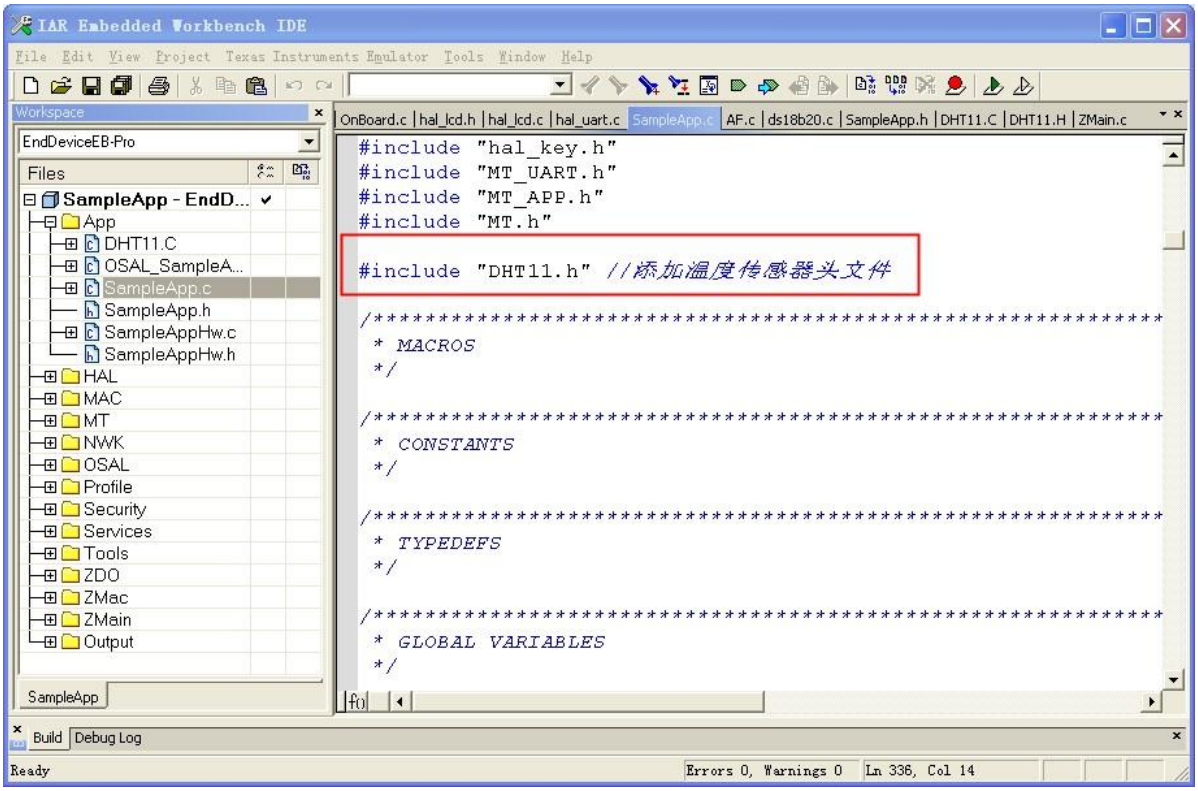


图 3.120

4）初始化传感器引脚 P0.6

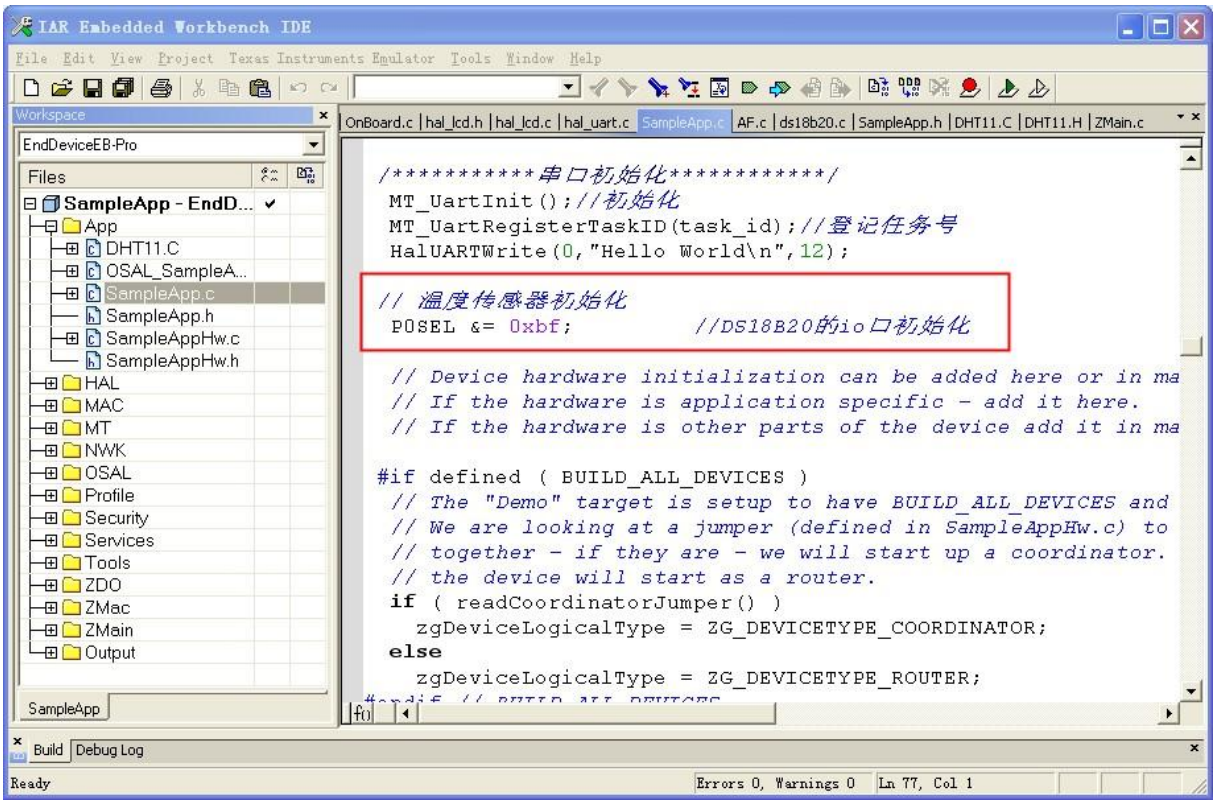


图 3.121

4）借用周期性点播函数，1s 读取温度传感器 1 次，通过液晶显示和串口打印并点对点发送给协调器 。代码如下，如图 3.122 所示：

```
uint8 T[8];      //温度+提示符
DHT11();   //温度检测
T[0]=wendu_shi+48;
T[1]=wendu_ge+48;
T[2]=' ';
T[3]=shidu_shi+48;
T[4]=shidu_ge+48;
T[5]=' ';
T[6]=' ';
T[7]=' ';
```

```
/*******串口打印 WEBEE*********/
HalUARTWrite(0,"temp=",5);
HalUARTWrite(0,T,2);
HalUARTWrite(0,"\n",1);

HalUARTWrite(0,"humidity=",9);
HalUARTWrite(0,T+3,2);
HalUARTWrite(0,"\n",1);

/*******LCD 显示  WEBEE*********/
HalLcdWriteString("Temp: humidity:", HAL_LCD_LINE_3 );//LCD 显示
HalLcdWriteString( T, HAL_LCD_LINE_4 );//LCD 显示
```

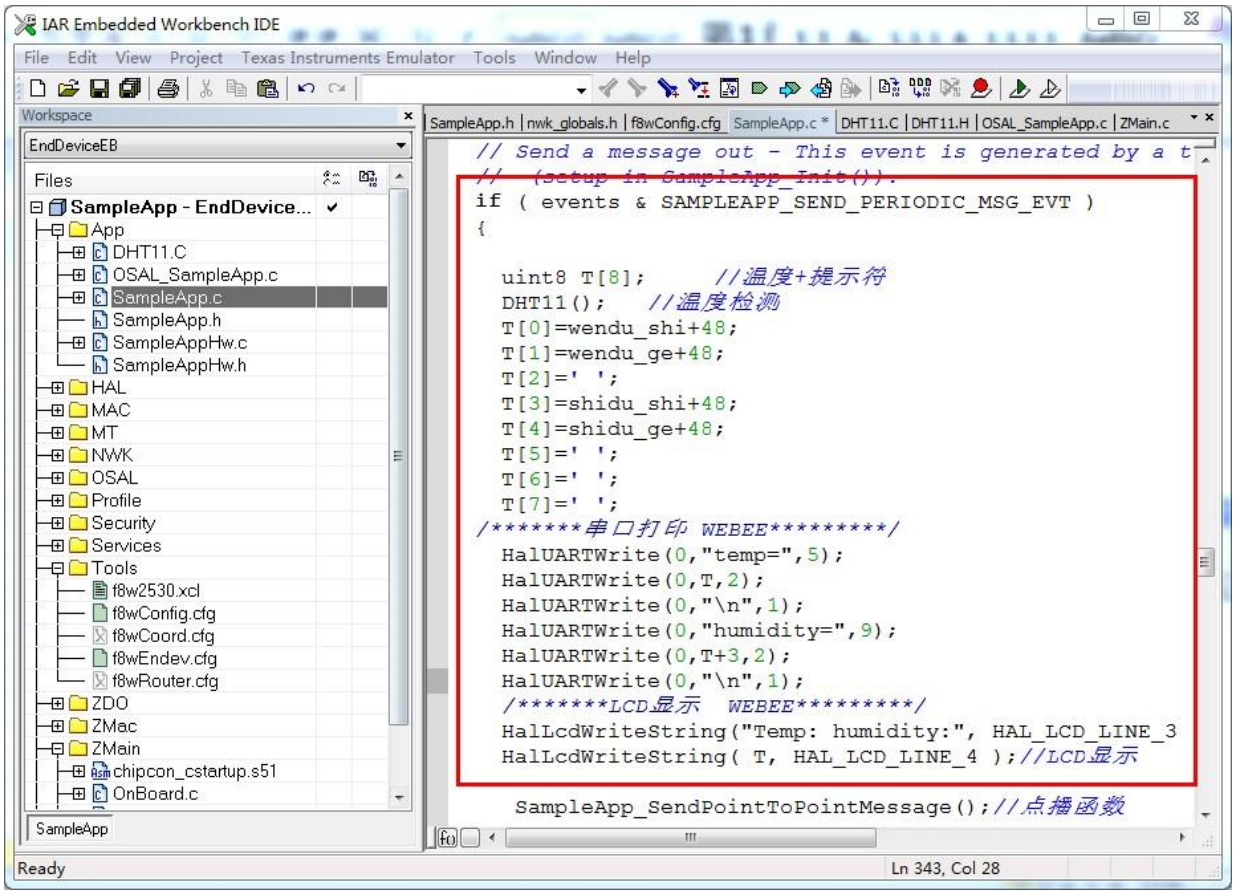


图 3.122

5） DHT11.c 文件需要修改一个地方。打开改文件，将原来的延时函数改成协议栈自带的延时函数，保证时序的正确。

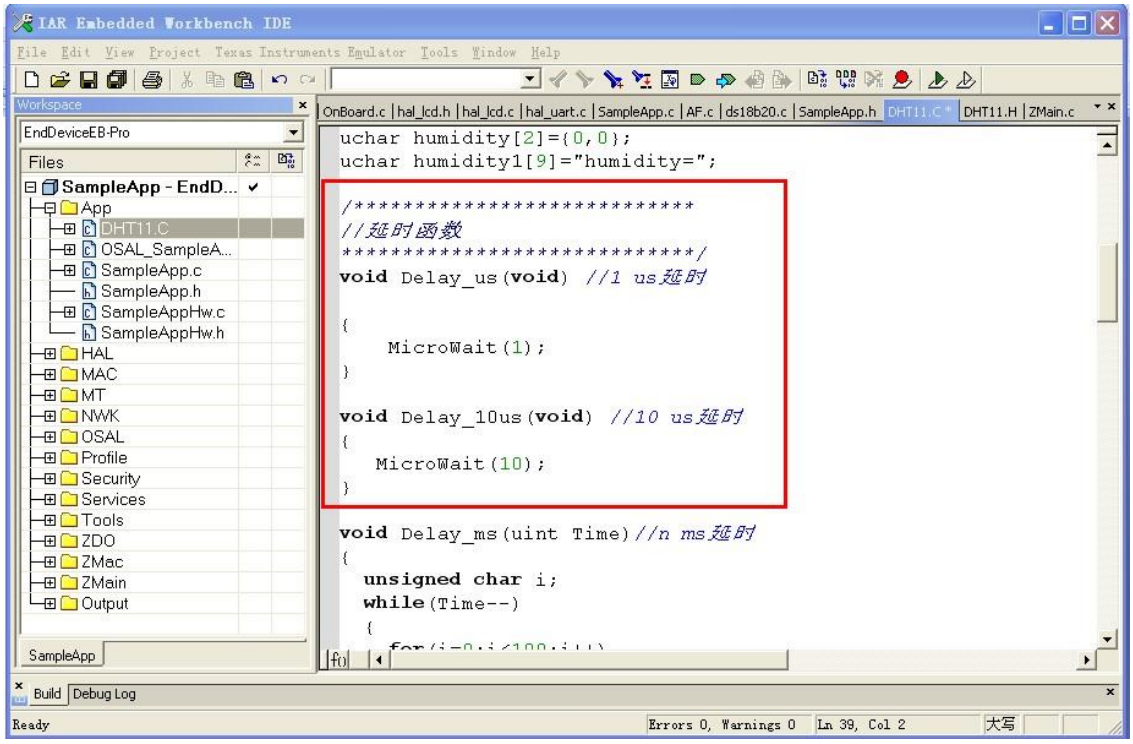


图 3.123

同时要包含 #include"OnBoard.h" 。

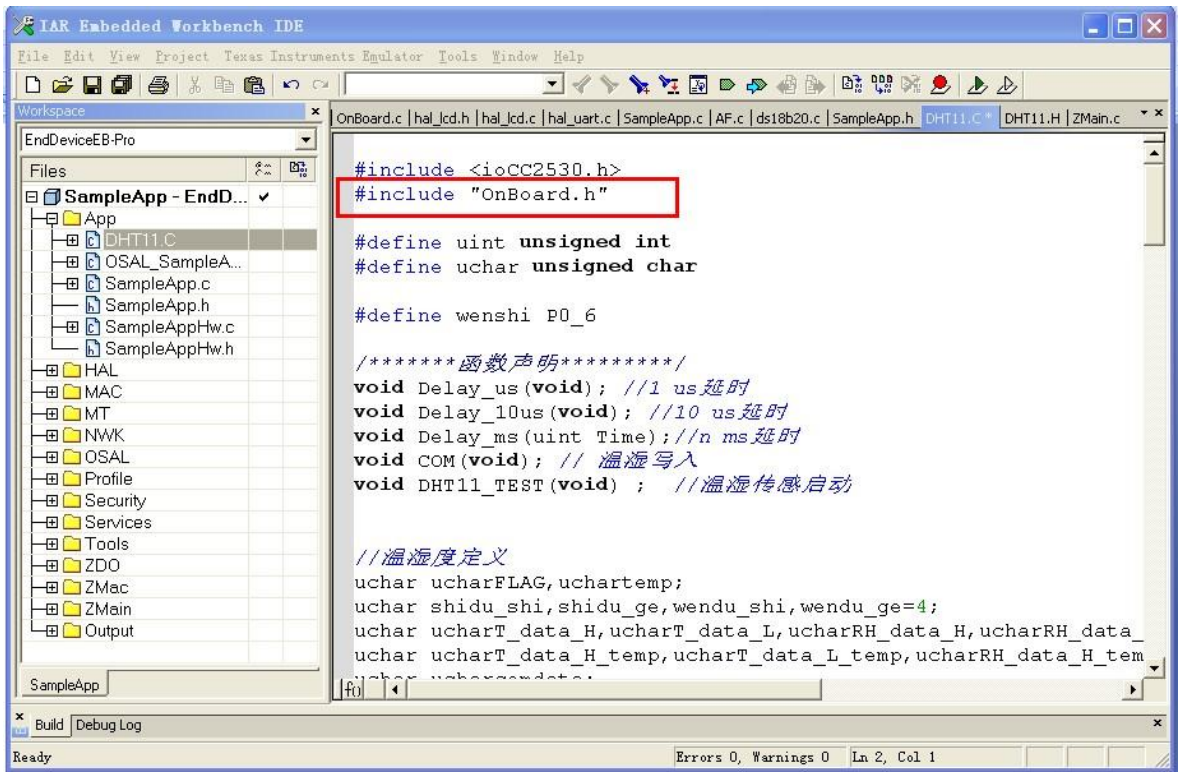


图 3.124

还要加上所有的函数声明：

```
/*******函数声明*********/
void Delay_us(void); //1 us 延时
void Delay_10us(void); //10 us 延时
void Delay_ms(uint Time);//n ms 延时
void COM(void);      // 温湿写入
void DHT11 (void) ;   //温湿传感启动
```

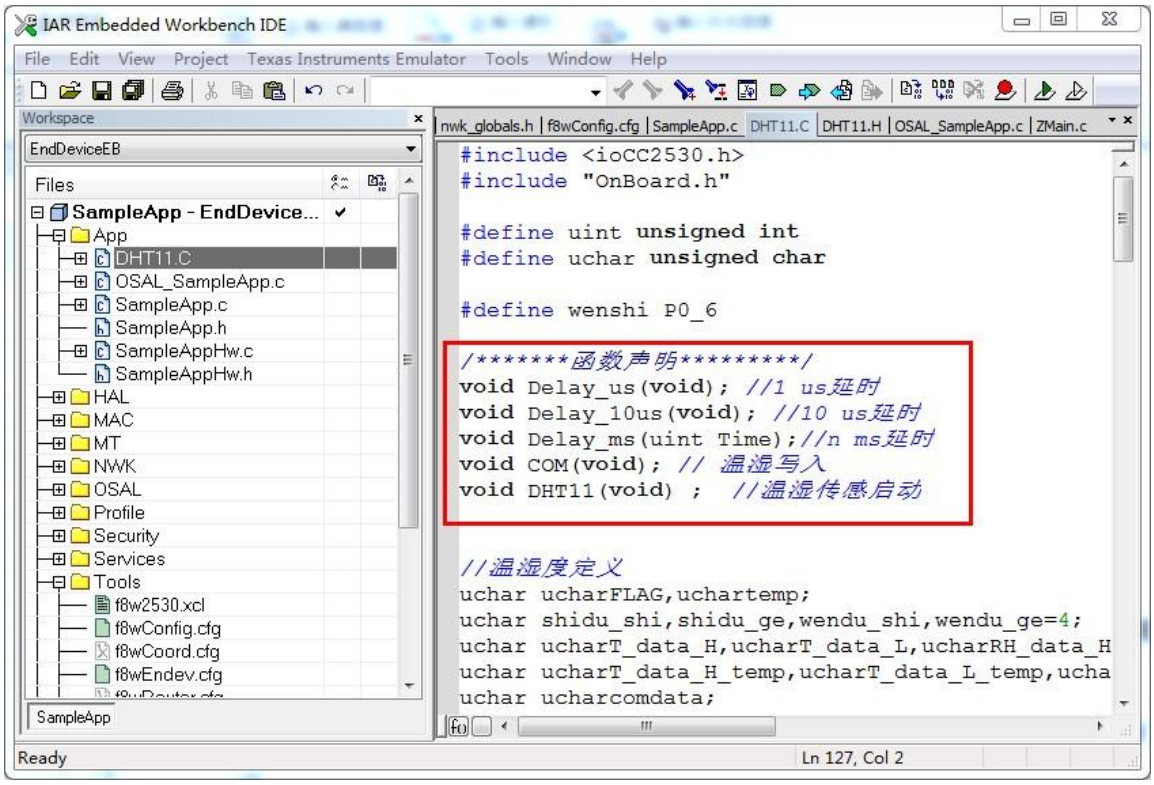


图 3.125

三：将数据打包并按指定的方式发送给指定设备。

在上一个步骤，我们完成了 DHT11 基于协议栈的驱动，接下来，我们就用以前的知识，实现数据发送和接收就可以了。

在 EndDevice 的点播发送函数中将温度信息发送出去，代码如下，如图所示：

```
void SampleApp_SendPointToPointMessage( void )
{
  uint8 T_H[4];//温湿度
  T_H[0]=wendu_shi+48;
  T_H[1]=wendu_ge%10+48;

  T_H[2]=shidu_shi+48;
  T_H[3]=shidu_ge%10+48;

  if ( AF_DataRequest( &Point_To_Point_DstAddr,
                       &SampleApp_epDesc,
                       SAMPLEAPP_POINT_TO_POINT_CLUSTERID,
                       4,
                       T_H,
                       &SampleApp_TransID,
                       AF_DISCV_ROUTE,
                       AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
  {
  }
  else
  {
    // Error occurred in request to send.
  }
}
```

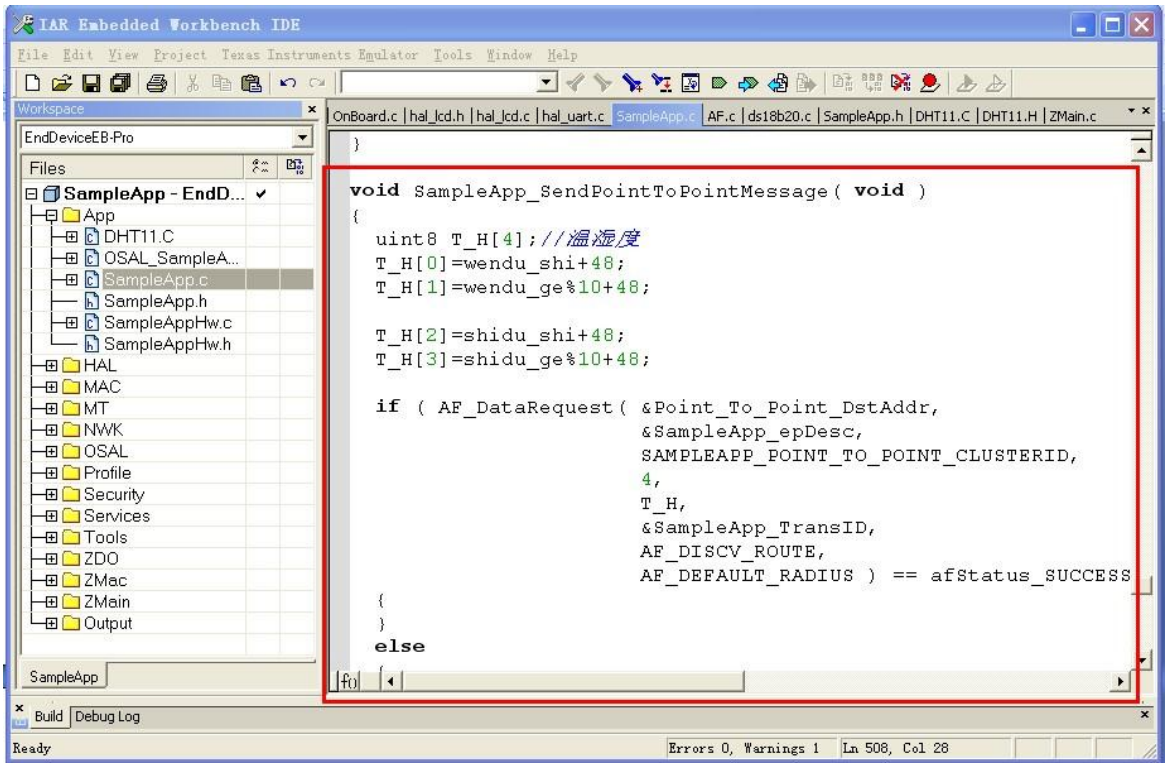


图 3.126

协调器代码如下，如图 3.127 所示：

```
case SAMPLEAPP_POINT_TO_POINT_CLUSTERID:

  /***********温度打印***************/
  HalUARTWrite(0,"Temp is:",8);          //提示接收到数据
  HalUARTWrite(0,&pkt->cmd.Data[0],2); //温度
  HalUARTWrite(0,"\n",1);                 // 回车换行

  /***************湿度打印****************/
   HalUARTWrite(0,"Humidity is:",12);     //提示接收到数据
   HalUARTWrite(0,&pkt->cmd.Data[2],2);   //湿度
   HalUARTWrite(0,"\n",1);                  // 回车换行

   break;
```

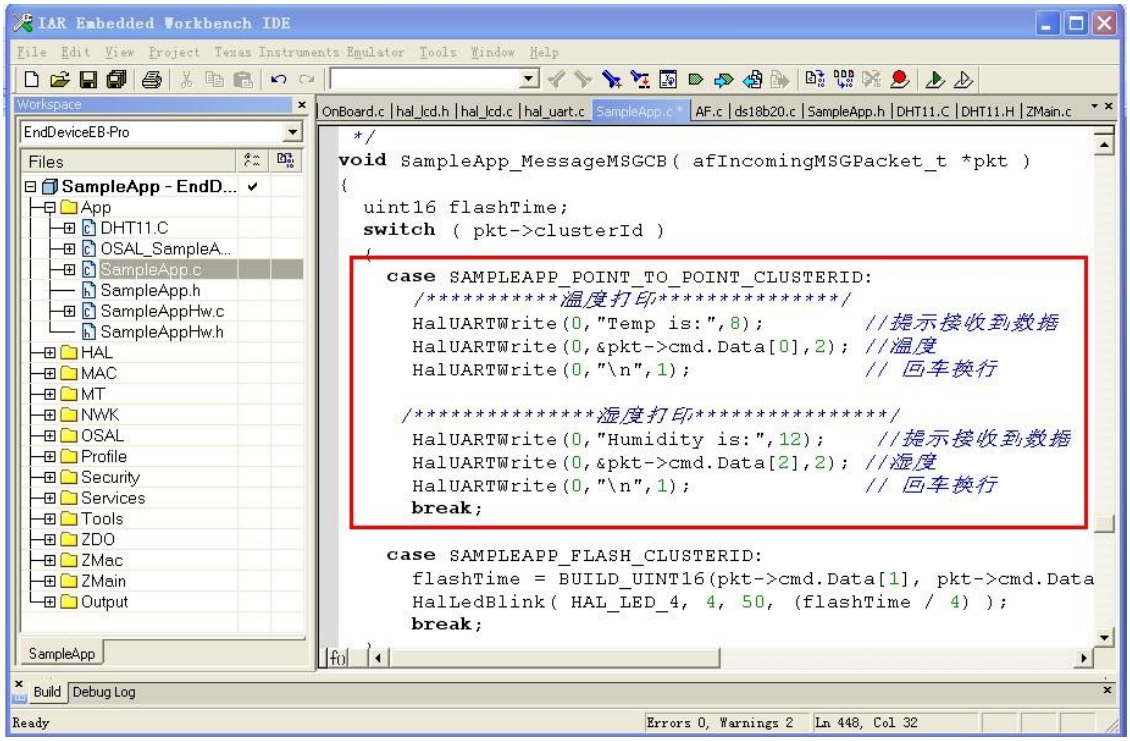


图 3.127

实验结果及图片：

观察终端设备的 LCD 以及将协调器连接到电脑，观察从终端发来的信息。以终端方式下载到开发板，当连接上协调器时，可以看到 LCD 显示当前温度信息，同时串口打印出温湿度传感器信息。如图 3.128 所示：

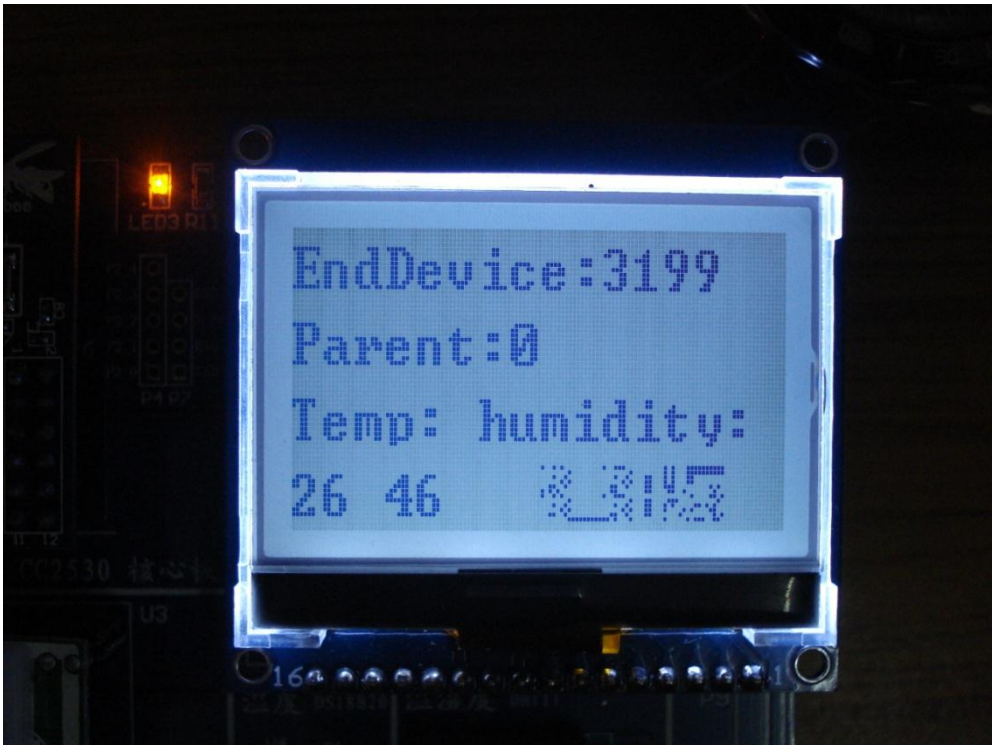

EndDevice:3199
Parent:0
Temp: humidity:
26 46

图 3.128

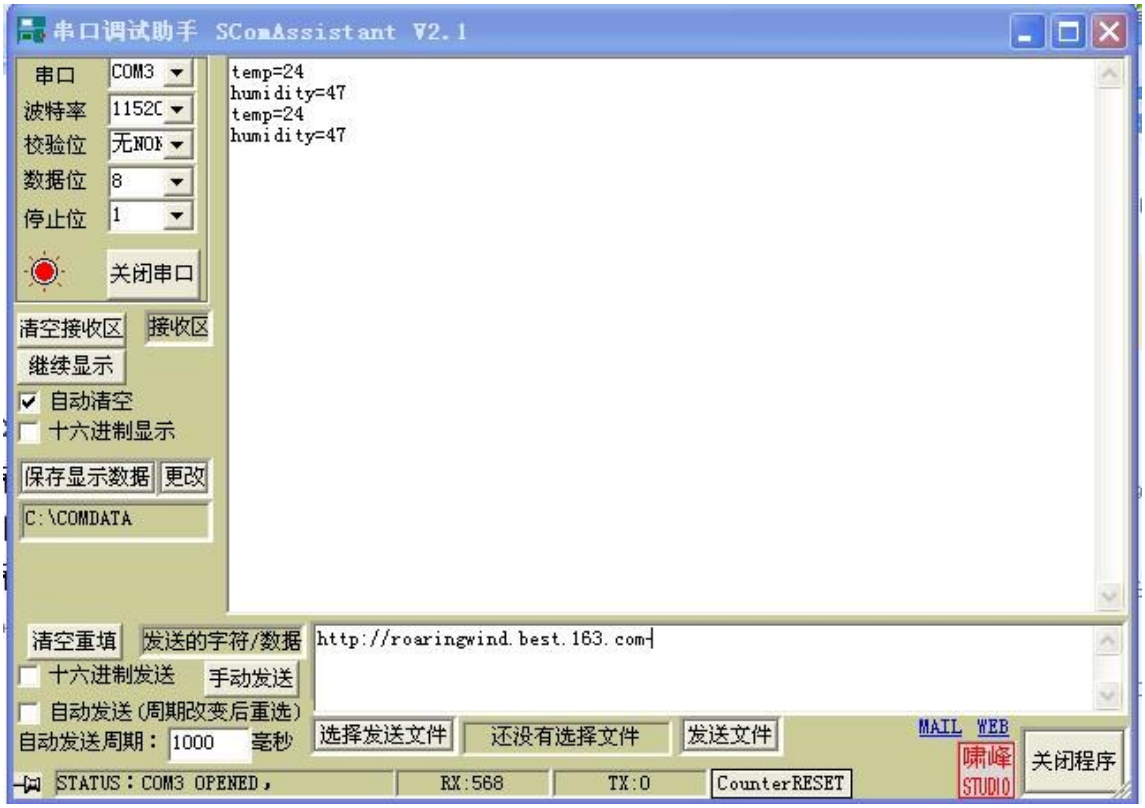

串口调试助手 SComAssistant V2.1
串口
COM3
波特率
11520
校验位
无NOP
数据位
8
停止位
1
关闭串口
清空接收区
接收区
继续显示
自动清空
十六进制显示
保存显示数据
更改
C:\COMDATA
temp=24
humidity=47
temp=24
humidity=47
清空重填
发送的字符/数据
http://roaringwind.best.163.com
十六进制发送
手动发送
自动发送(周期改变后重选)
自动发送周期:
1000
毫秒
选择发送文件
还没有选择文件
发送文件
MAIL
WEB
啸峰 STUDIO
关闭程序
STATUS:COM3 OPENED,
RX:568
TX:0
CounterRESET

图 3.129

同时协调器收到终端发送来的信息：

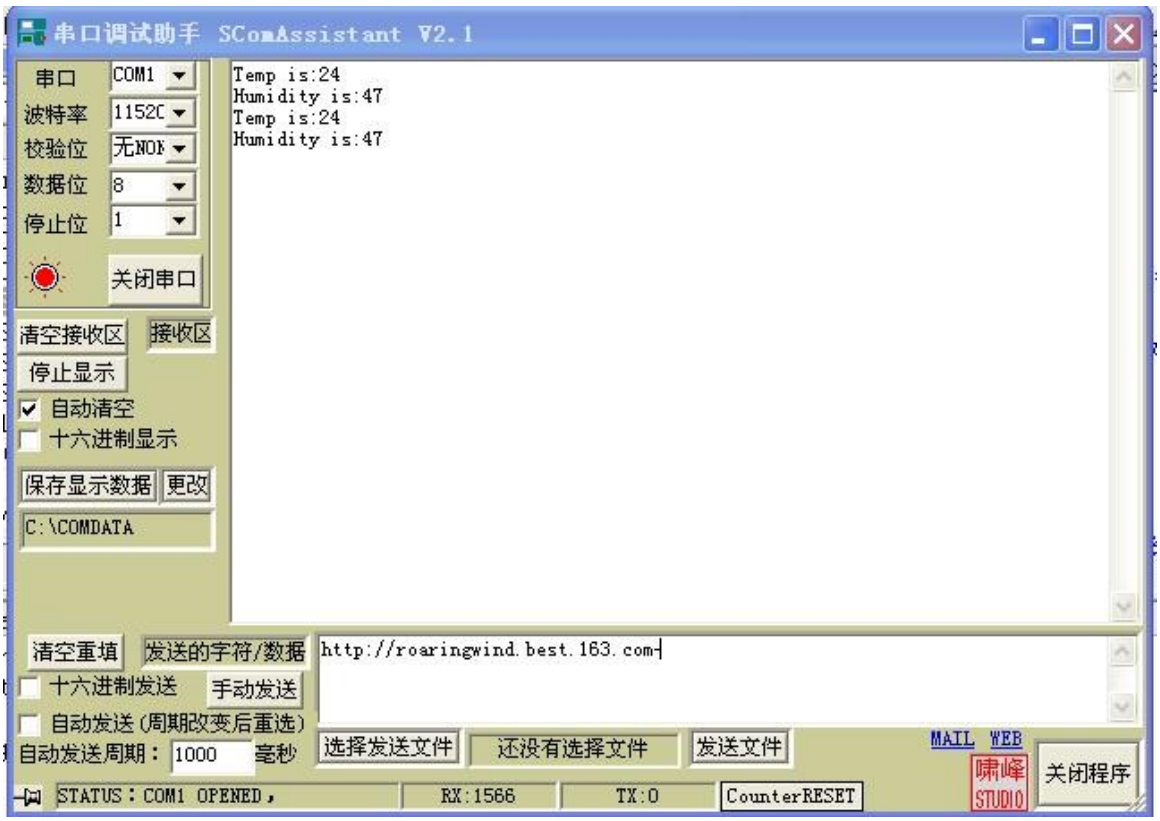


图 3.130

结语：

至此，我们完成了对 DHT11 温度数据的无线采集，也就是从初始化、采集到打包发送的过程。可以告诉你，单总线设备的驱动难度是不低的，所以我们有了网蜂 DS18B20 和 DHT11 的例程，相信大家战胜其他传感器再也不是问题了。感谢大家对网蜂的支持！

3.11.3 光敏传感器

前言：这一节我们我学习传感器部分内容中的光敏传感器，这一类型的传感器跟前面温湿度的传感器最大的区别就是控制简单，只有硬件电路搭好了，给 CC2530 的 IO 口一个高低电平就是反映外界情况。所以我们用起来就很方便。

传感器介绍：

光敏传感器是最常见的传感器之一，它的种类繁多，主要有：光电管、光电倍增管、光敏电阻、光敏三极管、太阳能电池、红外线传感器、紫外线传感器、光纤式光电传感器、色彩传感器、CCD 和 CMOS 图像传感器等。它的敏感波长在可见光波长附近，包括红外线波长和紫外线波长。光传感器不只局限于对光的探测，它还可以作为探测元件组成其他传感器，对许多非电量进行检测，只要将这些非电量转换为光信号的变化即可。光传感器是目前产量最多、应用最广的传感器之一，它在自动控制和非电量电测技术中占有非常重要的地位。最简单的光敏传感器是光敏电阻，当光子冲击接合处就会产生电流。

图 3.131 光敏电阻

实现平台：网蜂二代 ZigBee 开发平台、ZigBee 传感器节点

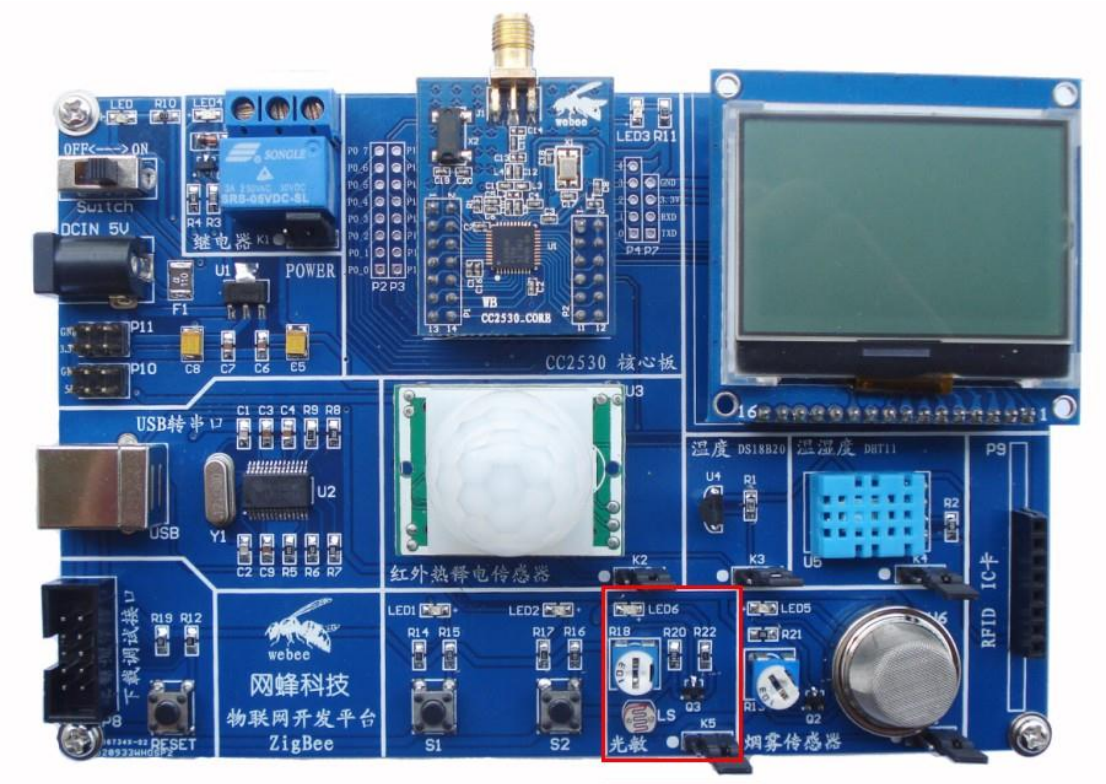


图 3.132 网蜂物联网 ZigBee 开发平台

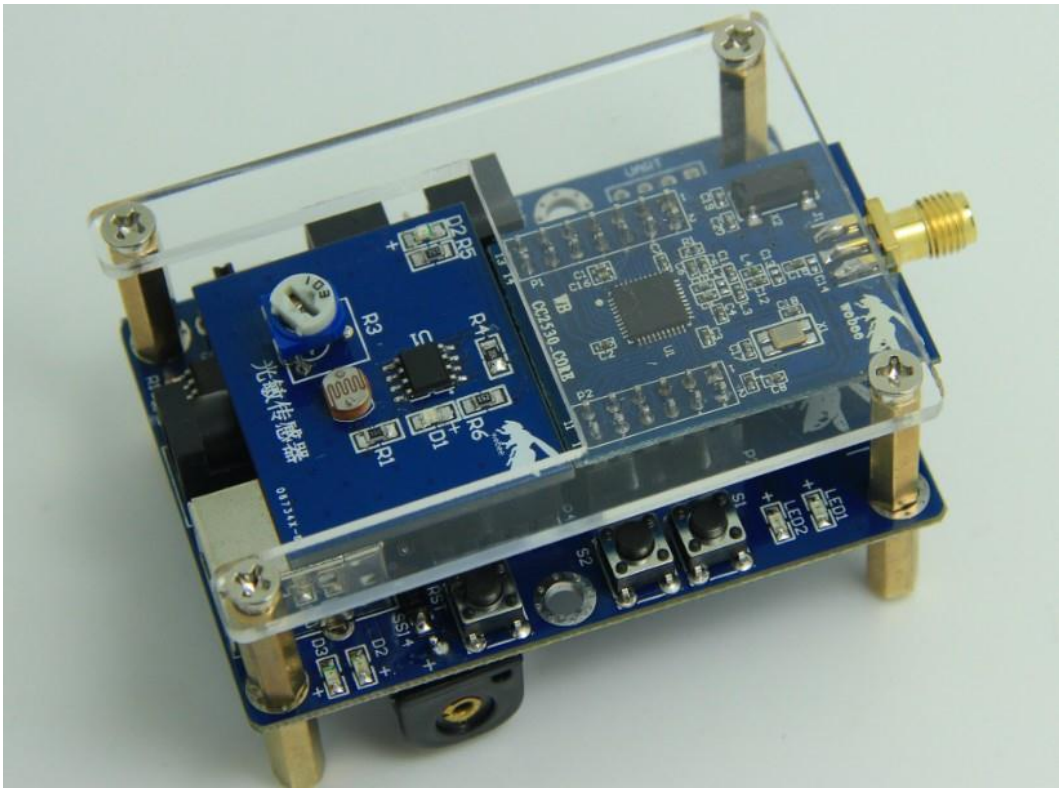


图 3.133 ZigBee 传感器节点

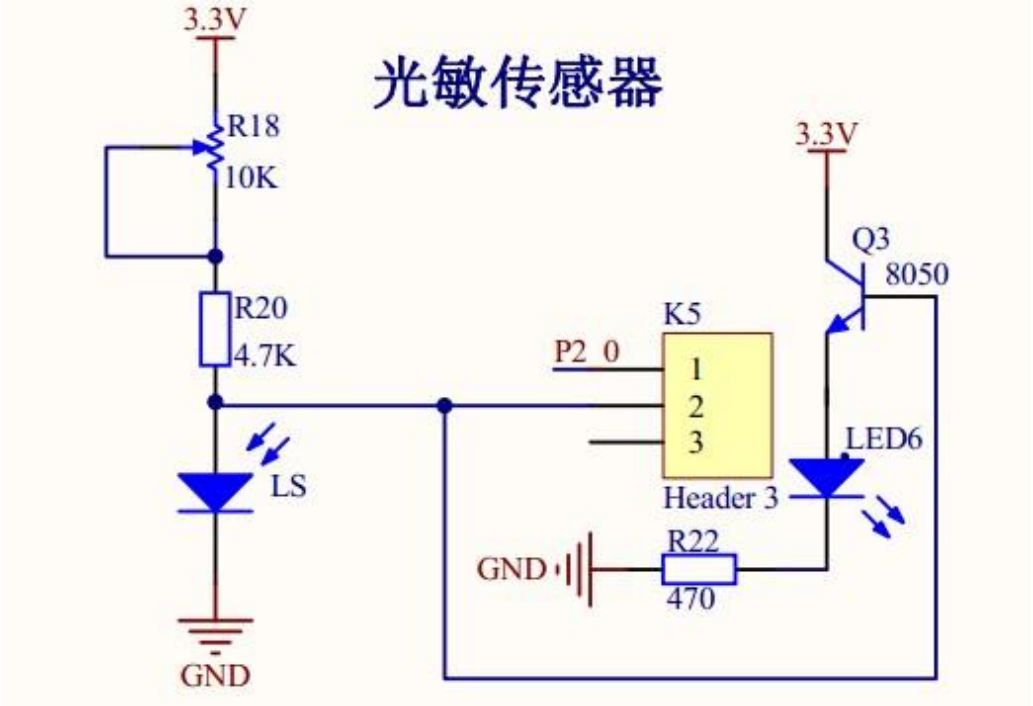


图 3.134　zigbee 开发平台光敏传感器硬件电路

实验现象：光敏电阻电路通过检测外界光线的情况，将信息通过 LCD12864 显示和串口打印出来。

实验讲解：像前面传感器例程一样，我们先实现裸机程序里检测光敏电阻电路的检测。然后在协议栈里添加相应的代码。

一：在裸机上完成对光敏电阻电路的驱动。

打开配套程序下裸机文件夹—光敏传感器下的工程文件，看到函数如下：

```
/*************************************/
/*              WeBee 团队             */
/*          Zigbee 学习例程            */
/*例程名称：光线检测                   */
/*建立时间：2012/10                    */
/*描述：通过光敏电阻检测外界光线，通过 LED1 指示
```

```
**************************************/
1.  1.#include <ioCC2530.h>

2.  2.#define uint unsigned int
3.  #define uchar unsigned char

4.  //定义控制 LED 灯的端口
5.  #define LED1 P1_0          //LED1 为 P1.0 口控制
6.  #define LIGHT P2_0         //光敏电阻为 IO 口为 P2.0

7.  //函数声明
8.  void Delayms(uint);     //延时函数
9.  void InitLed(void);     //初始化 LED1
10. void LightInit();      //光敏初始化
11. uchar LightScan ();                  //按键扫描程序

12. /*****************************
13.          延时函数
14. *****************************/
15. void Delayms(uint xms)    //i=xms 即延时 i 毫秒
16. {
17. uint i,j;
18. for(i=xms;i>0;i--)
        for(j=587;j>0;j--);
19. }
20. /*****************************
            LED 初始化函数
21. *****************************/
22. void InitLed(void)
```

```
{
    P1DIR |= 0x01;  //P1_0 定义为输出
    LED1 = 1;       //LED1 灯熄灭
}

/*****************************
    光敏电阻初始化函数
*****************************/
void LightInit()
{
    P2SEL &= ~0X01;      //设置 P20 为普通 IO 口
    P2DIR &= ~0X01;     // 在 P20 口，设置为输入模式
    P2INP &= ~0x01;    //打开 P20 上拉电阻,不影响
}
/*****************************
  i.    按键检测函数
*****************************/
uchar LightScan(void)
{
    if(LIGHT==0)
    {Delayms(10);
        if(LIGHT==0)
        {

            return 1;       //有按键按下
        }
    }
    return 0;             //无按键按下
}
```

```
35. /****************************
          主函数
36. ****************************/
37. void main(void)
38. {
39.     InitLed();                    //调用初始化函数
40.     LightInit();
41.     while(1)
        {
        if(LightScan())    //按键改变 LED 状态
                LED1=1;           //有光，LED1 灭掉
            else
                LED1=0;          //无光，LED1 点亮
         }
42. }
```

我们来看主函数：

第 39~40 行：进行一些初始化工作。

第 41~42 行：判断外界光线情况,通过 LED1 指示。

上面的代码实现了当有光线时候 LED1 灭掉，没有光线的时候 LED1 亮。实验现象如图 4 所示：

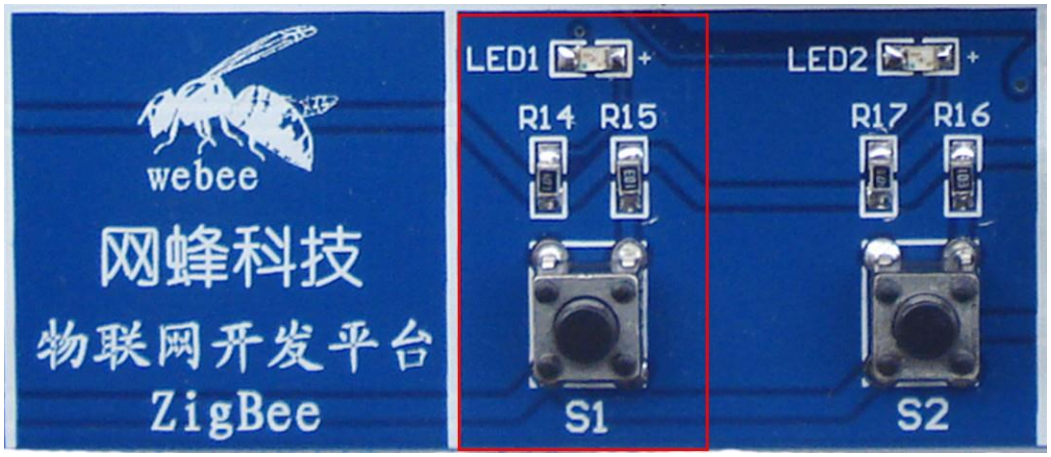


图 3.135

二：将程序添加到协议栈代码中

光敏电阻电路是对 IO 口电平的检测。所以在协议栈里检测程序比较简单。我们只需要配置好 IO 口，然后周期性检测就可以了。

1） 定义光敏传感器输入 IO P2.0

```
//光敏电阻定义
#define LIGHT P2_0        //光敏为 P2.0 口控制
```

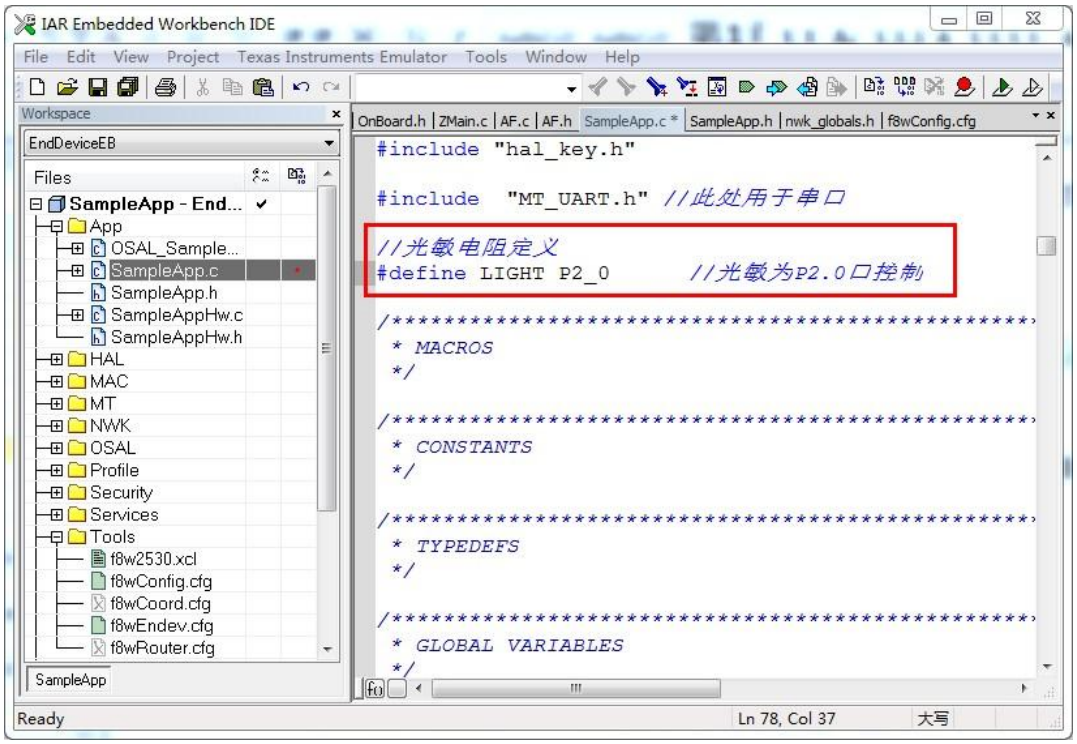


图 3.136

2） 打开例程 SampleApp.eww 工程，打开 SampleApp.c 文件。我们先初始化 P2.0 引脚。设为输入模式。

```
/******光敏电阻电路初始化******/
P2SEL &= ~0X01;     //设置 P2.0 为普通 IO 口
P2DIR &= ~0X01;     // 在 P2.0 口，设置为输入模式
P2INP &= ~0x01;     //打开 P2.0 上拉电阻
```

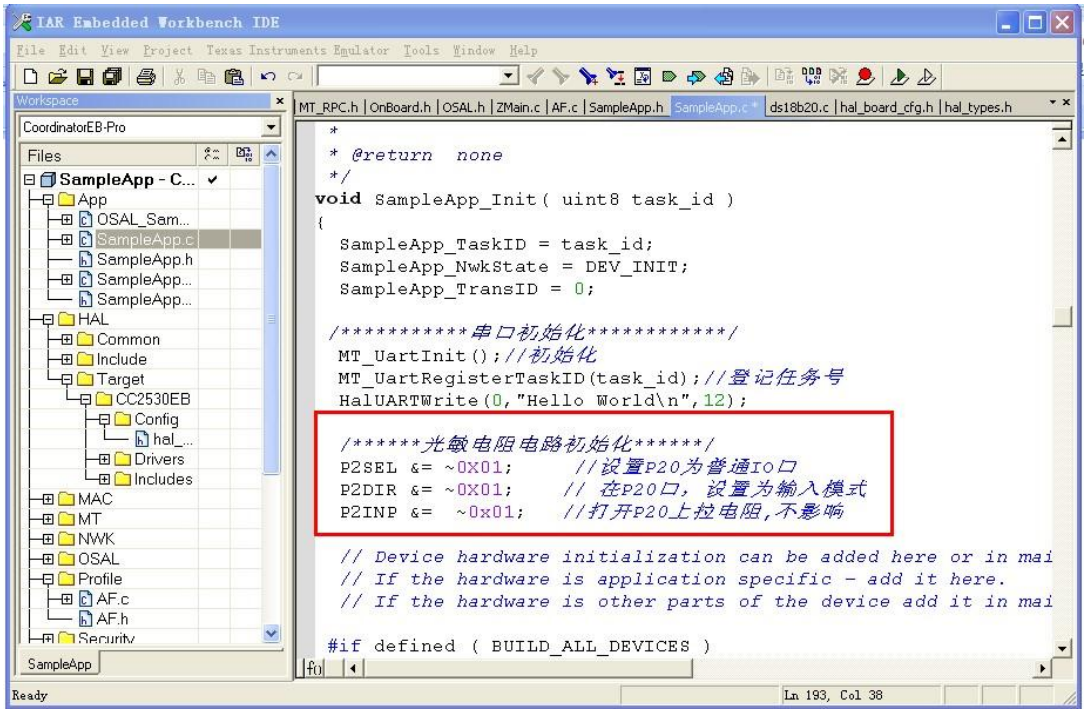


图 3.137 光敏 IO 初始化

3） 利用周期性点播的定时器作为光线信息采集时间，将采集到的信息发送给协调器。并通过 LCD 显示和串口打印。协调器只做串口打印。0.5 秒采集一次。

```
// Send Message Timeout
#define SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT   500     // Every 0.5
                                                      seconds
```

4） 终端每 0.5 秒执行点播函数一次，我们在点播函数里判断 IO 口。加入下面红色代码。

```
void SampleApp_SendPointToPointMessage( void )
{
  uint8 L;
  if(LIGHT==1)
  {
    L=1;  //有光线
    HalUARTWrite(0,"no light\n",9);      //串口
```

```
    HalLcdWriteString( "No Light", HAL_LCD_LINE_3 ); //LCD
  }
  else
  {
    L=0;  //没有光线
    HalUARTWrite(0,"got Light\n",10);        //串口
    HalLcdWriteString( "Got Light", HAL_LCD_LINE_3 );//LCD
  }
  if ( AF_DataRequest( &Point_To_Point_DstAddr,
                       &SampleApp_epDesc,
                       SAMPLEAPP_POINT_TO_POINT_CLUSTERID,
                       1,
                       &L,
                       &SampleApp_TransID,
                       AF_DISCV_ROUTE,
                       AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
  {
  }
  else
  {
    // Error occurred in request to send.
  }
}
```

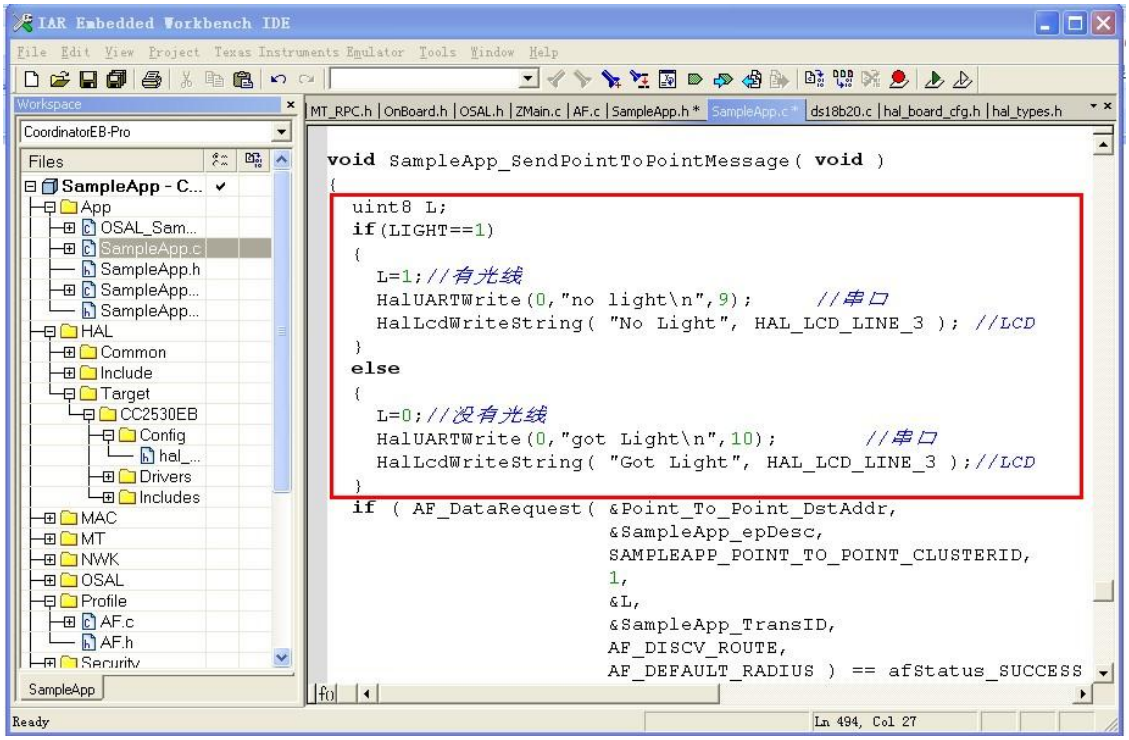


图 3.138

5) 协调器接收函数我们将数据读出来然后判断。通过串口打印传感器信息出来。

```
case SAMPLEAPP_POINT_TO_POINT_CLUSTERID:
if(pkt->cmd.Data[0])
    HalUARTWrite(0,"no light\n",9);   //没光线
else
    HalUARTWrite(0,"got light\n",10); //有光线
break;
```

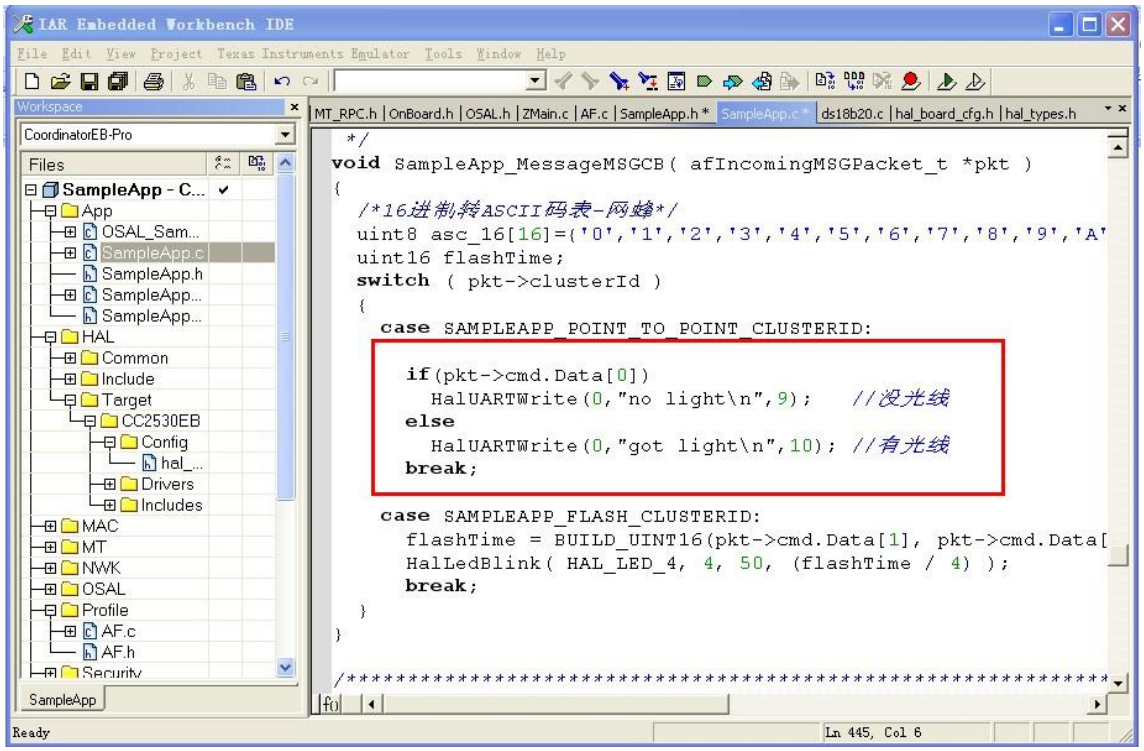


图 3.139 接收判断

实验现象：下载程序到终端（带光敏传感器）和协调器。观察 LCD12864，如下图所示：

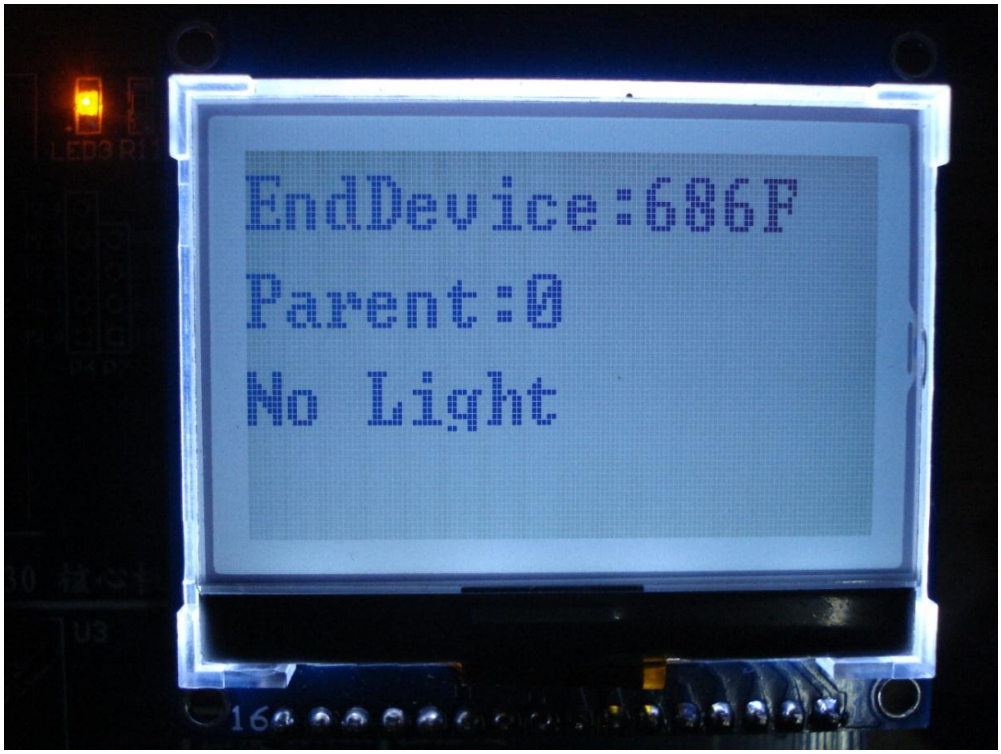


图 3.140

协调器收到的信息：

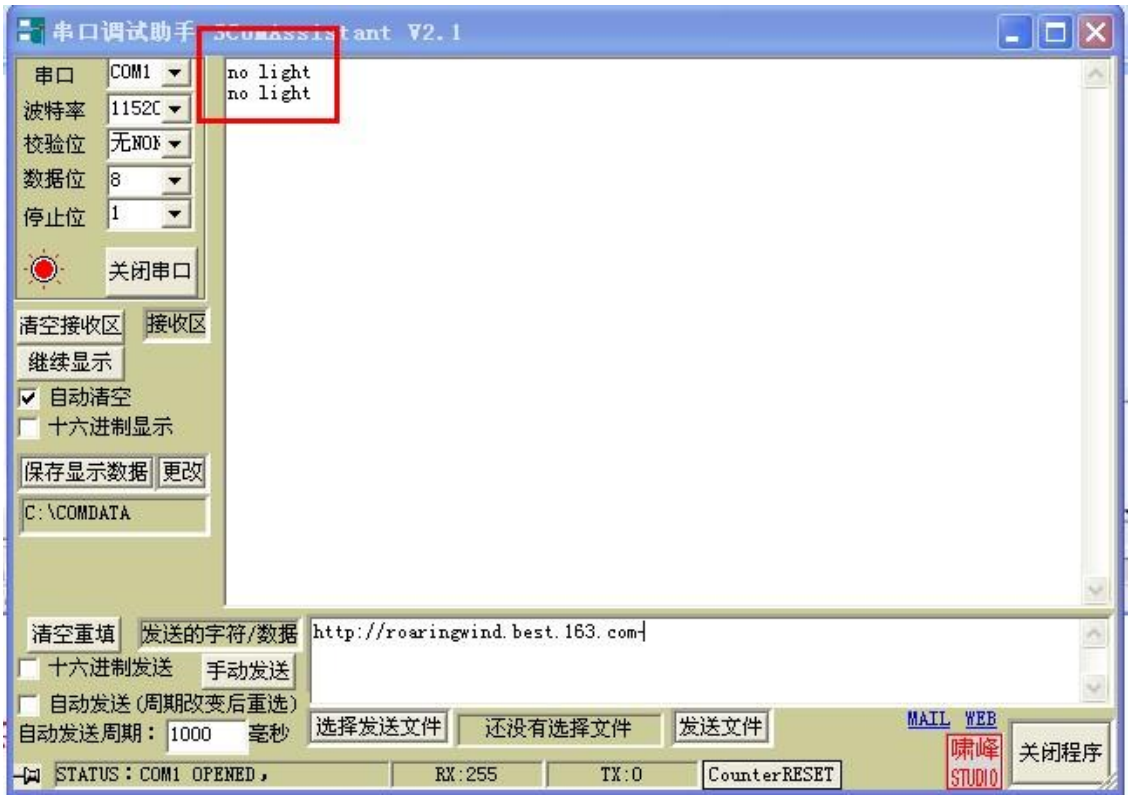


图 3.141

结语：细心的读者会发现，光敏传感器的检测实际上就是 CC2530 的 IO 口电平检测，通过硬件电路的转换将模拟量变成了数字量。接下来的部分传感器也是相关的原理。希望大家能读一通百。

3.11.4 烟雾传感器

前言：这一节我们我学习传感器部分内容中的烟雾传感器，烟雾传感器跟光敏传感器检测方法类似，硬件电路搭建好了，给 CC2530 的 IO 口一个高低电平就是反映外界情况。我们需要做的就是对 CC2530 相应 IO 口的检测。

传感器介绍：

烟雾传感器就是通过监测烟雾的浓度来实现火灾防范的，烟雾报警器内部采用离子式烟雾传感，离子式烟雾传感器是一种技术先进，工作稳定可靠的传感器，被广泛运用到各种消防报警系统中，性能远优于气敏电阻类的火灾报警器。

图 3.11.4 A　MQ-2 烟雾传感器

实现平台：网蜂物联网 ZigBee 开发平台、ZigBee 传感器节点；

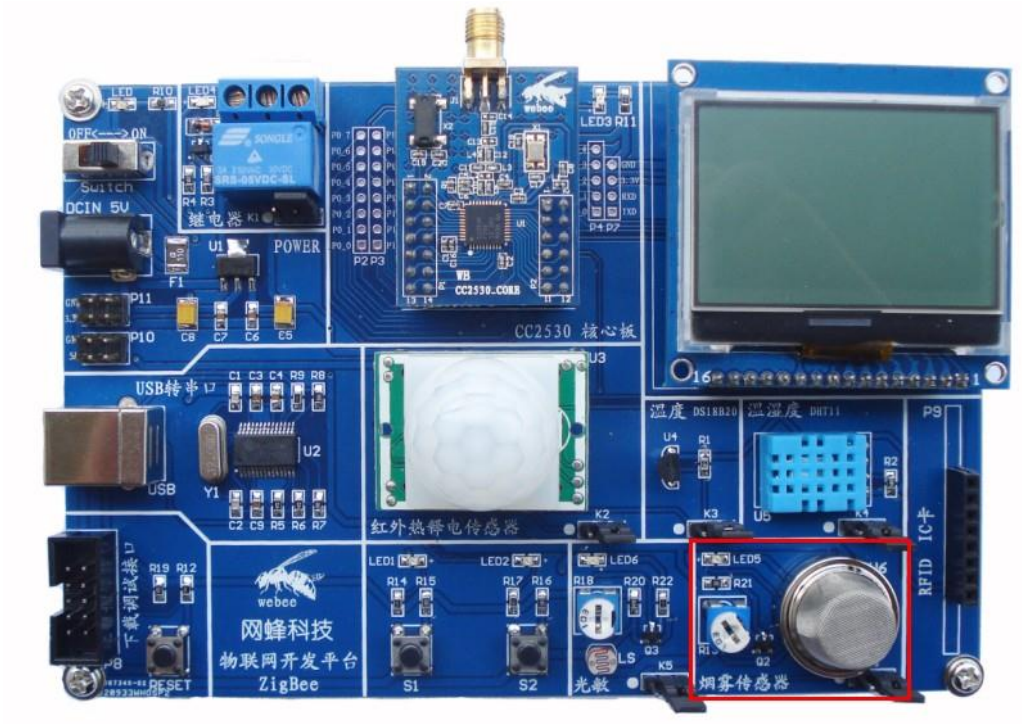


图 3.142 网蜂物联网 ZigBee 开发平台

图 3.143 ZigBee 传感器节点

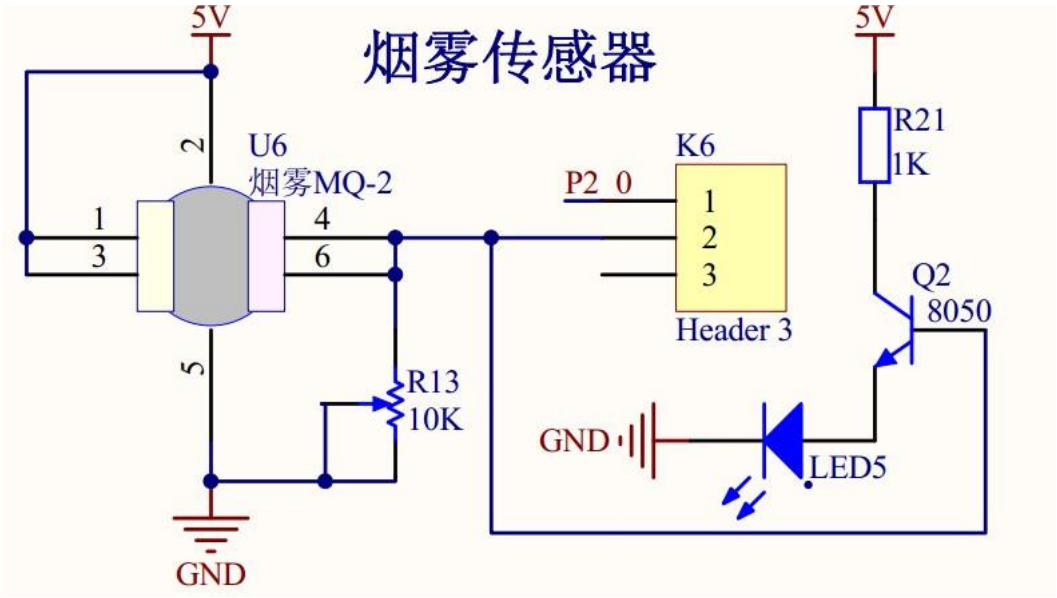


图 3.144 烟雾传感器硬件电路

实验现象：烟雾传感器电路通过检测外界有毒气体和烟雾情况的情况，将信息通过 LCD12864 显示和串口打印出来。

实验讲解：像前面传感器例程一样，我们先实现裸机程序里检测烟雾传感器电路的检测。然后在协议栈里添加相应的代码。

一：在裸机上完成对烟雾传感器的驱动。

打开配套程序下裸机文件夹—光敏传感器下的工程文件，看到函数如下：

```
1. /*****************************************/
2. /*              WeBee 团队                */
3. /*          Zigbee 学习例程               */
4. /*例程名称：烟雾传感器                    */
```

```
/*建立时间：2012/10                              */
/*描述：通过烟雾传感器，通过 LED1 指示
*************************************/
#include <ioCC2530.h>

#define uint unsigned int
#define uchar unsigned char

//定义控制 LED 灯的端口
#define LED1 P1_0       //LED1 为 P1.0 口控制
#define AIR P2_0        //光敏为 P2.0 口控制

//函数声明
void Delayms(uint);     //延时函数
void InitLed(void);     //初始化 LED1
void AirInit();                    //光敏初始化
uchar AirScan();                   //烟雾扫描程序

/*****************************
          延时函数
*****************************/
void Delayms(uint xms)    //i=xms 即延时 i 毫秒
{
  uint i,j;
  for(i=xms;i>0;i--)
    for(j=587;j>0;j--);
}

/*****************************
```

```
            LED 初始化函数
28. ****************************/
29. void InitLed(void)
30. {
        P1DIR |= 0x01;  //P1_0 定义为输出
        LED1 = 1;       //LED1 灯熄灭
31. }

32. /****************************
            按键初始化函数
33. ****************************/
34. void AirInit()
35. {
        P2SEL &= ~0X01;     //设置 P20 为普通 IO 口
        P2DIR &= ~0X01;    // 在 P20 口，设置为输入模式
        P2INP &=  ~0x01;   //打开 P20 上拉电阻,不影响
36. }

37. /****************************
            烟雾检测函数
38. ****************************/
39. uchar AirScan(void)
40. {
        if(AIR==0)
        {Delayms(10);
            if(AIR==0)
            {
                return 1;    // 无烟雾
            }
```

```
        }
        return 0;              //有烟雾
41. }

42. /*****************************
          主函数
43. *****************************/
44. void main(void)
45. {
46.     InitLed();        //调用初始化函数
47.     AirInit();
48.     while(1)
49.    {
50.      if(AirScan())       //按键改变 LED 状态
            LED1=1;          //无烟雾，LED1 灭掉
         else
            LED1=0;         //有烟雾，LED1 点亮
       }
51. }
```

我们来看主函数：

第 46~47 行：进行一些初始化工作。

第 50~51 行：判断外界光线情况,通过 LED1 指示。

上面的代码实现了当有烟雾或有毒气体的时候时候 LED1 灭掉，没有光线的时候 LED1 亮。实验现象如图所示：

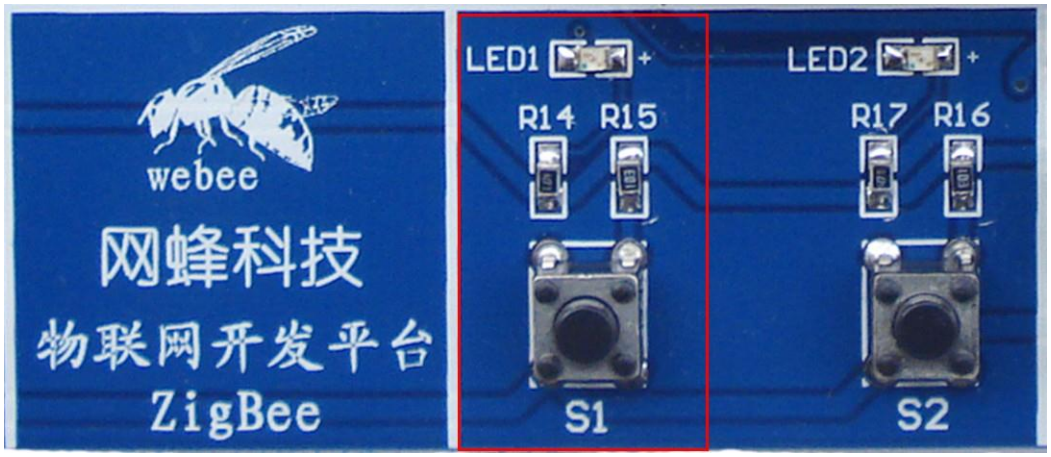


图 3.145

二：将程序添加到协议栈代码中

烟雾传感器电路是对 IO 口电平的检测。所以在协议栈里检测程序比较简单。我们只需要配置好 IO 口，然后周期性检测就可以了。

1） //烟雾传感器 IO 定义

```
#define AIR P2_0    //P2.0
```

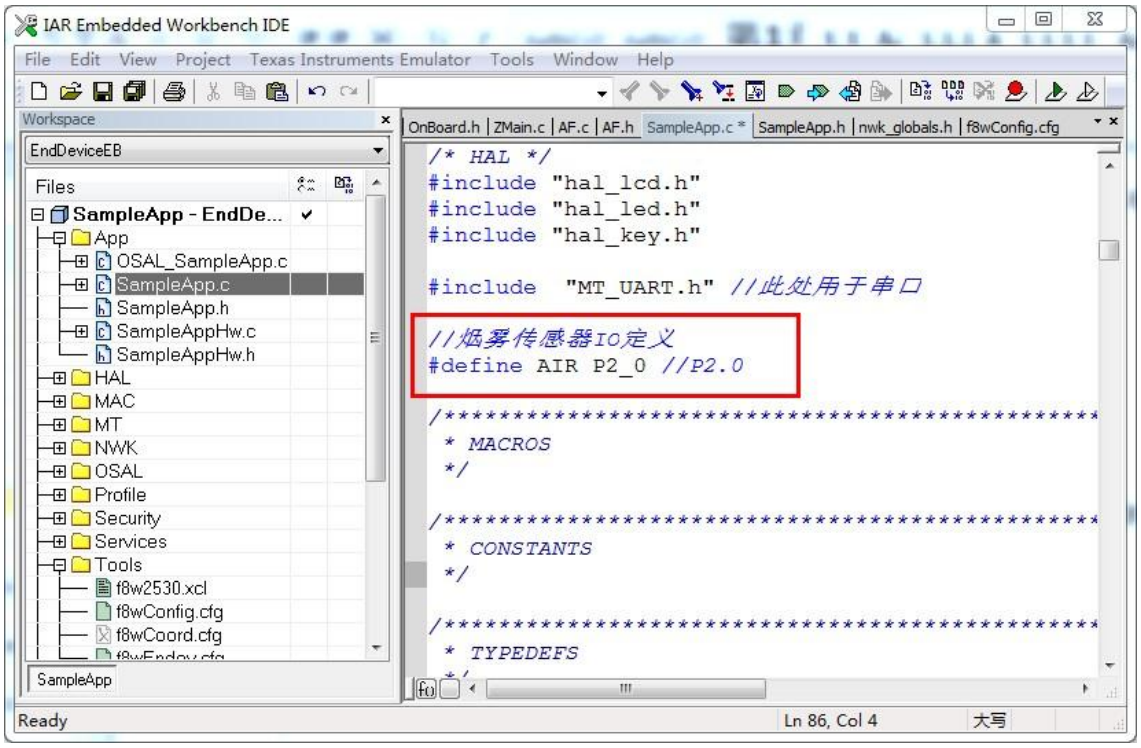


图 3.146

2） 打开例程 SampleApp.eww 工程，打开 SampleApp.c 文件。我们先初始化 P2.0 引脚。设为输入模式。

```
/******烟雾传感器电路初始化******/
P2SEL &= ~0X01;     //设置 P2.0 为普通 IO 口
P2DIR &= ~0X01;     // 在 P2.0 口，设置为输入模式
P2INP &= ~0x01;     //打开 P2.0 上拉电阻
```

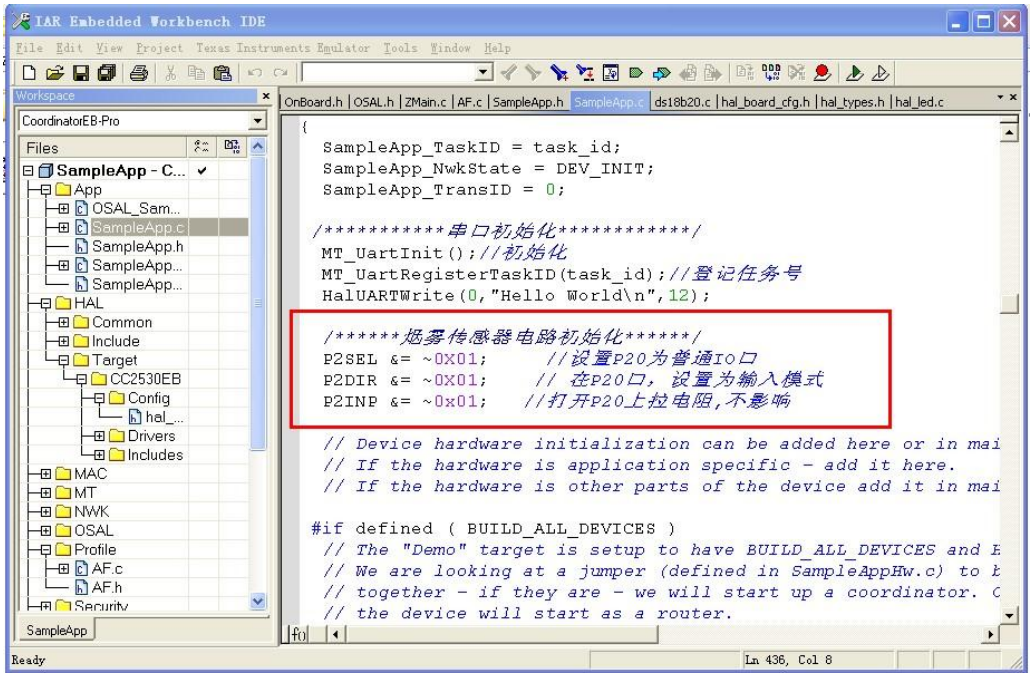


图 3.147 初始化 IO 口

3） 利用周期性点播的定时器作为烟雾信息采集时间，将采集到的信息发送给协调器。并通过 LCD 显示和串口打印。协调器只做串口打印。0.5 秒采集一次。

```
// Send Message Timeout
#define SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT   500     // Every 0.5
                                                      seconds
```

4） 终端每 0.5 秒执行点播函数一次，我们在点播函数里判断 IO 口。加入下面红色代码。

```
void SampleApp_SendPointToPointMessage( void )
{
    uint8 L;
  if(AIR==1)
  {
    L=1;//没烟雾
    HalUARTWrite(0,"Got bad Air\n",12);     //串口
    HalLcdWriteString( "Got bad Air", HAL_LCD_LINE_3 ); //LCD
  }
  else
  {
    L=0;//有烟雾
    HalUARTWrite(0,"No bad Air\n",11);       //串口
    HalLcdWriteString( "No bad Air", HAL_LCD_LINE_3 );//LCD
  }  if ( AF_DataRequest( &Point_To_Point_DstAddr,
                          &SampleApp_epDesc,
                          SAMPLEAPP_POINT_TO_POINT_CLUSTERID,
                          1,
                          &L,
                          &SampleApp_TransID,
                          AF_DISCV_ROUTE,
                          AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
  {
  }
  else
  {
    // Error occurred in request to send.
  }
}
```

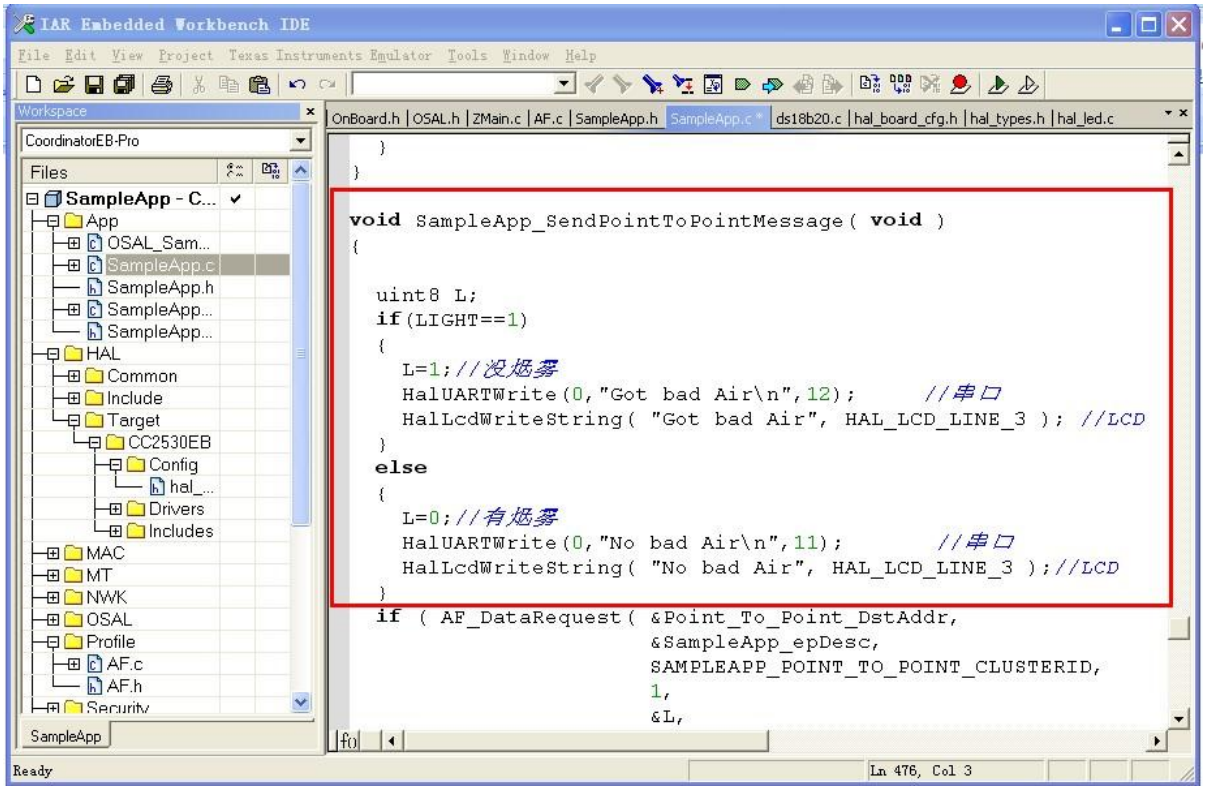


图 3.148

5) 协调器接收函数我们将数据读出来然后判断。通过串口打印传感器信息出来。

```
case SAMPLEAPP_POINT_TO_POINT_CLUSTERID:
 if(pkt->cmd.Data[0])
   HalUARTWrite(0,"Got bad Air\n",12);      //有烟雾
else
   HalUARTWrite(0,"No bad Air\n",11);      //有烟雾
break;
```

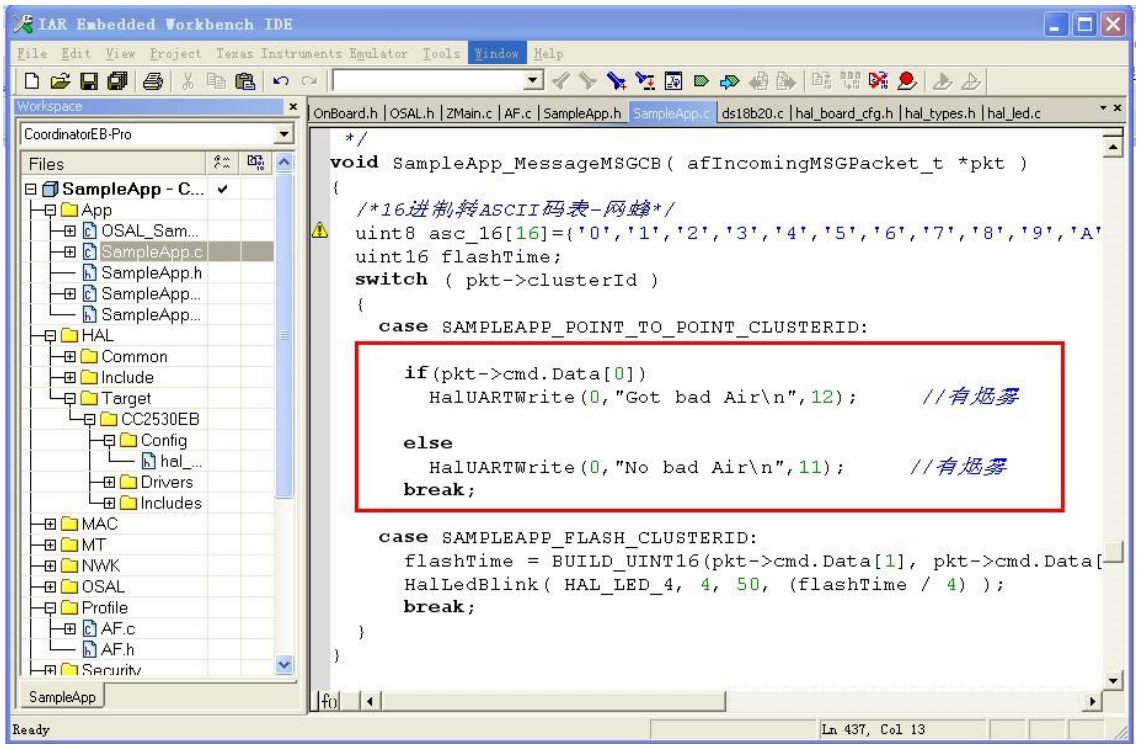


图 3.149

实验现象：下载程序到终端（带烟雾传感器）和协调器。观察 LCD12864，如下图所示：

图 3.150

协调器收到的信息：

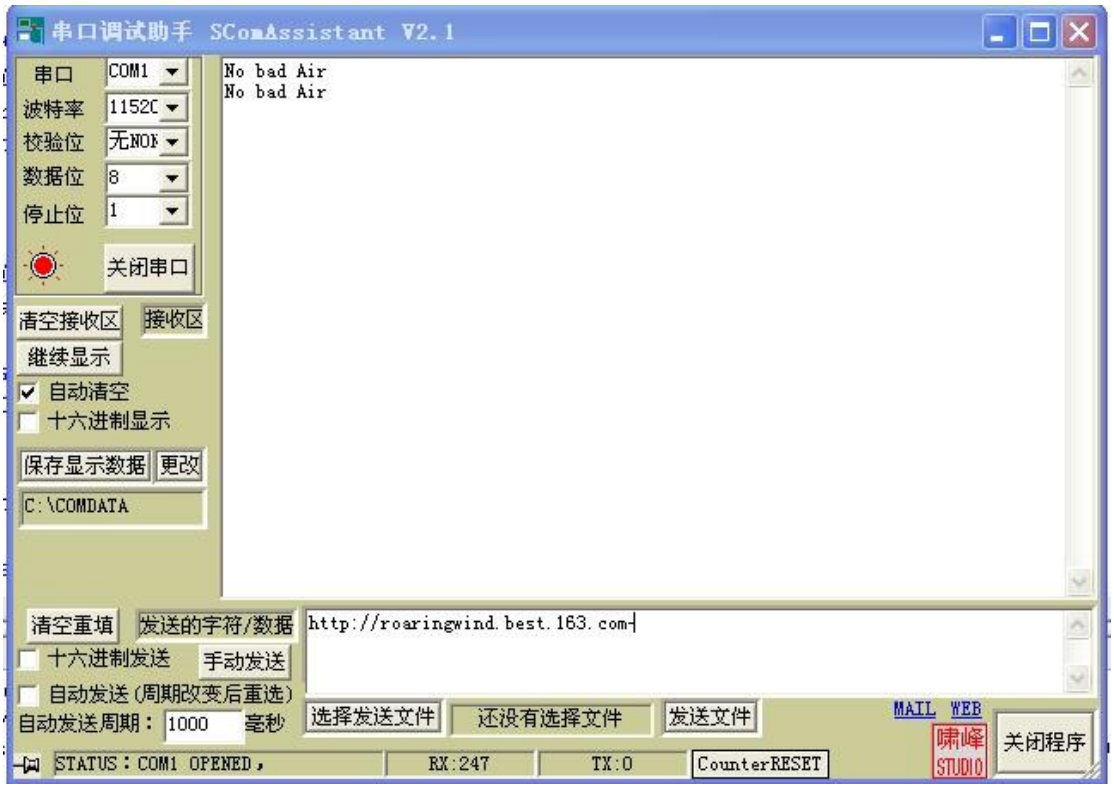


图 3.151

3.11.5 人体红外热释电传感器

前言：这一节我们我学习传感器部分内容中的红外热释电传感器，红外热释电传感器使用集成模块，当在传感器检测范围内有人的时候输出高电平（或低电平），当检测范围没人的时候输出相反的电平低电平（或高电平），改传感器可以用于安防、灯控等众多领域。

传感器介绍：

热释电红外线传感器主要是由一种高热电系数的材料，如锆钛酸铅系陶瓷、钽酸锂、硫酸三甘钛等制成尺寸为 2*1mm 的探测元件。在每个探测器内装入一个或两个探测元件，并将两个探测元件以反极性串联，以抑制由于自身温度升高而产生的干扰。由探测元件将探测并接收到的红外辐射转变成微弱的电压信号，经装在探头内的场效应管放大后向外输出。为了提高探测器的探测灵敏度以增大探测距离，一般在探测器的前方装设一个菲涅尔透镜，该透镜用透明塑料制成，将透镜的上、下两部分各分成若干等份，制成一种具有特殊光学系统的透镜，它和放大电路相配合，可将信号放大 70 分贝以上，这样就可以测出 10~20 米范围内人的行动。

图 3.152 红外热释电传感器模块

实现平台：网蜂物联网 ZigBee 开发平台、ZigBee 传感器节点；

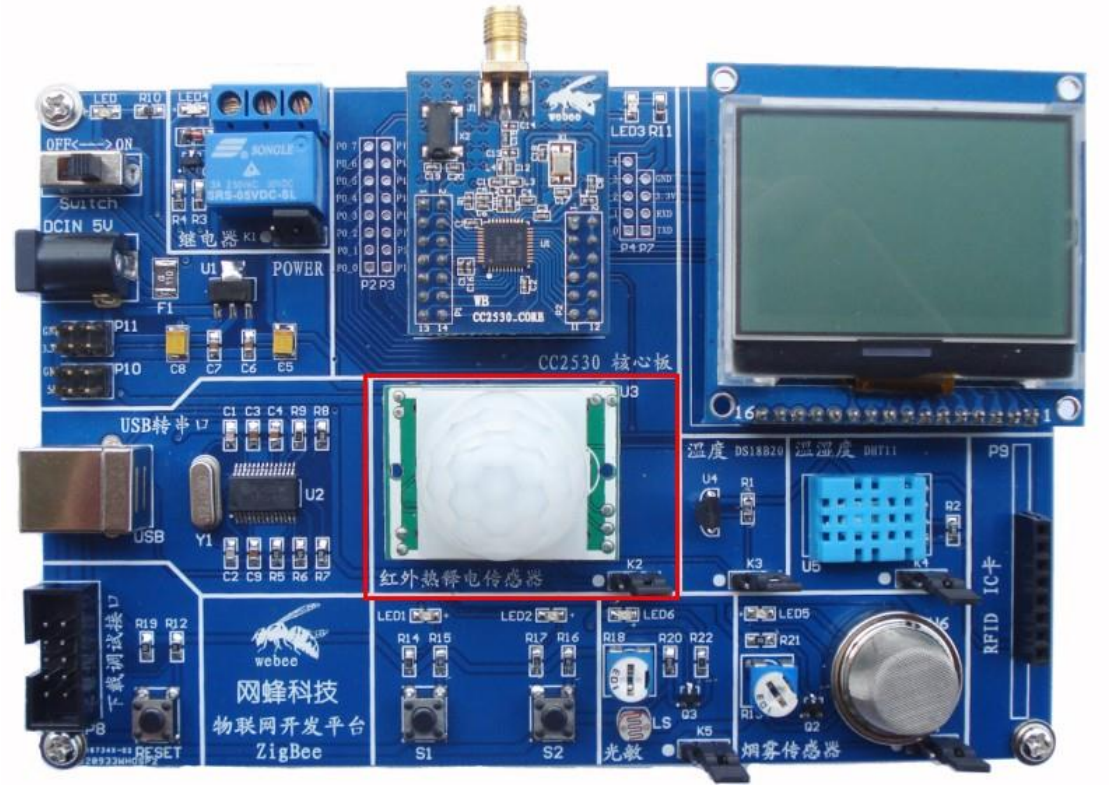


图 3.153 网蜂物联网 ZigBee 开发平台

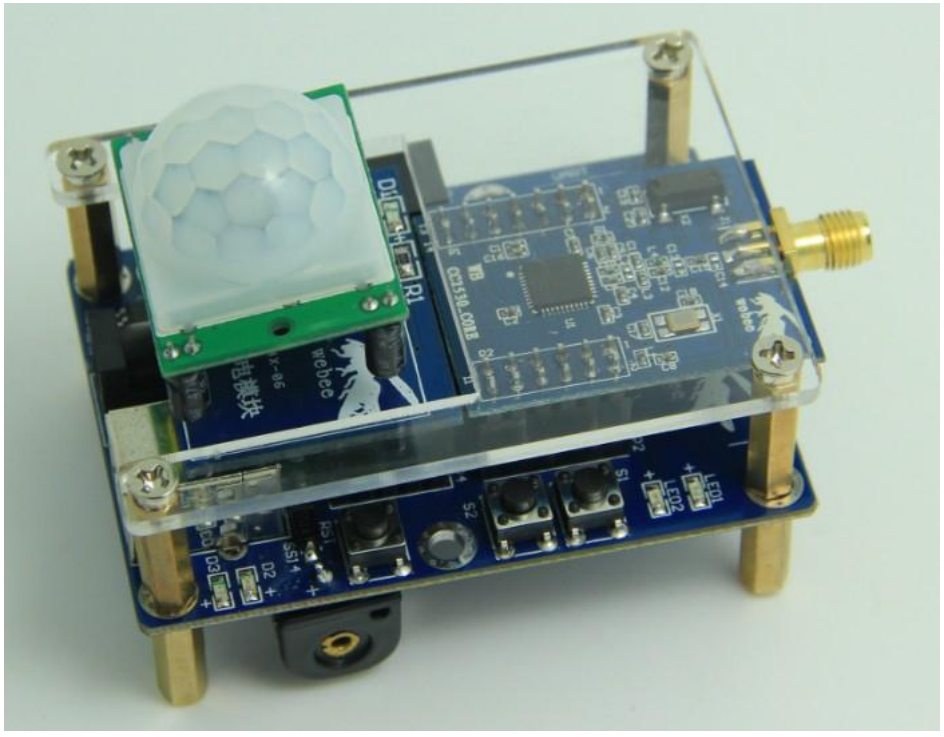


图 3.154 人体热释电传感器

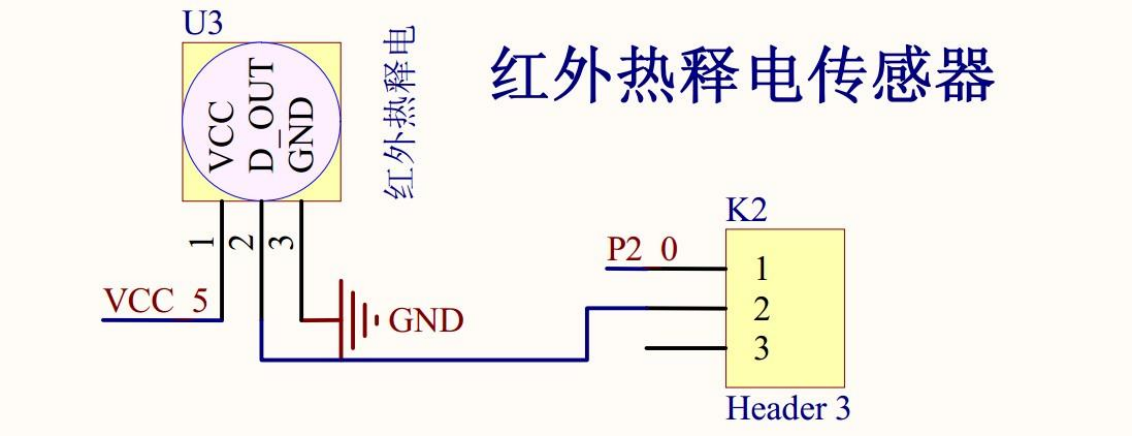


图 3.155 红外热释电电路图

实验现象：红外热释电传感器电路通过检测外界是否有人的情况，将信息通过 LCD12864 显示和串口打印出来。

实验讲解：像前面传感器例程一样，我们先实现裸机程序里实现对红外热释电传感器的检测。然后在协议栈里添加相应的代码。

一：在裸机上完成对红外热释电传感器的驱动。

打开配套程序下裸机文件夹—红外热释电传感器下的工程文件，看到函数如下：

```
1. /*************************************/
2. /*              WeBee 团队             */
3. /*          Zigbee 学习例程            */
4. /*例程名称：红外热释电传感器            */
5. /*建立时间：2012/10                    */
6. /*描述：通过红外热释电传感器监测周围环境
          是否有人员存在，通过 LED1 指示
7. *************************************/
```

```
8.  #include <ioCC2530.h>

9.  #define uint unsigned int
10. #define uchar unsigned char

11. //定义控制 LED 灯的端口
12. #define LED1 P1_0          //LED1 为 P1.0 口控制
13. #define PEOPLE P2_0        //红外热释电传感器为 P2.0 口检测

14. //函数声明
15. void Delayms(uint);       //延时函数
16. void InitLed(void);       //初始化 LED1
17. void PeopleInit();                  //光敏初始化
18. uchar PeopleScan();                 //烟雾扫描程序

19. /*****************************
20. 延时函数
21. *****************************/
22. void Delayms(uint xms)    //i=xms 即延时 i 毫秒
23. {
24. uint i,j;
25. for(i=xms;i>0;i--)
        for(j=587;j>0;j--);
26. }

27. /*****************************
          LED 初始化函数
28. *****************************/
29. void InitLed(void)
```

```
30. {
        P1DIR |= 0x01;  //P1_0 定义为输出
        LED1 = 1;       //LED1 灯熄灭
31. }

32. /*****************************
        热释电传感器 IO 初始化函数
33. *****************************/
34. void PeopleInit()
35. {
        P2SEL &= ~0X01;      //设置 P20 为普通 IO 口
        P2DIR &= ~0X01;     // 在 P20 口，设置为输入模式
        P2INP &=  ~0x01;    //打开 P20 上拉电阻,不影响
36. }

37. /*****************************
            人员检测函数
38. *****************************/
39. uchar PeopleScan(void)
40. {
        if(PEOPLE==0)
        {Delayms(10);
            if(PEOPLE==0)
            {
                return 1;    // 无人
            }
        }
        return 0;            //有人
41. }
```

```
42. /****************************
          主函数
43. ****************************/
44. void main(void)
45. {
46.   InitLed();                  //调用初始化函数
47.   PeopleInit();
48.   while(1)
49.   {
50.     if(PeopleScan())          //改变 LED 状态
51.       LED1=1;                 //无人，LED1 灭掉
52.     else
53.       LED1=0;                 //有人，LED1 点亮
      }
54. }
```

我们来看主函数：

第 46~47 行：进行一些初始化工作。

第 50~54 行：判断外界人员（有人或者没人）情况,通过 LED1 指示。

上面的代码实现了当没人的时候时候 LED1 灭掉，有人的时候 LED1 亮。实验现象如图 4 所示：

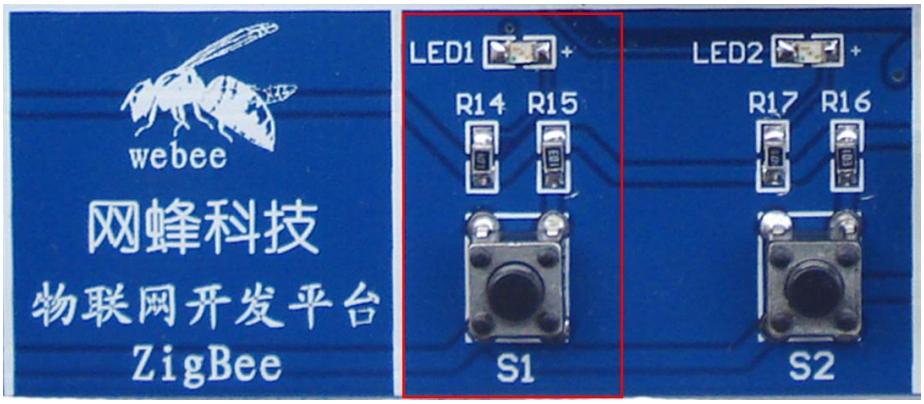


图 3.156

二：将程序添加到协议栈代码中

红外热释电传感器电路是对 IO 口电平的检测。所以在协议栈里检测程序比较简单。我们只需要配置好 IO 口，然后周期性检测就可以了。

1） //定义人体红外热释电传感器 IO

```
#define PEOPLE P2_0
```

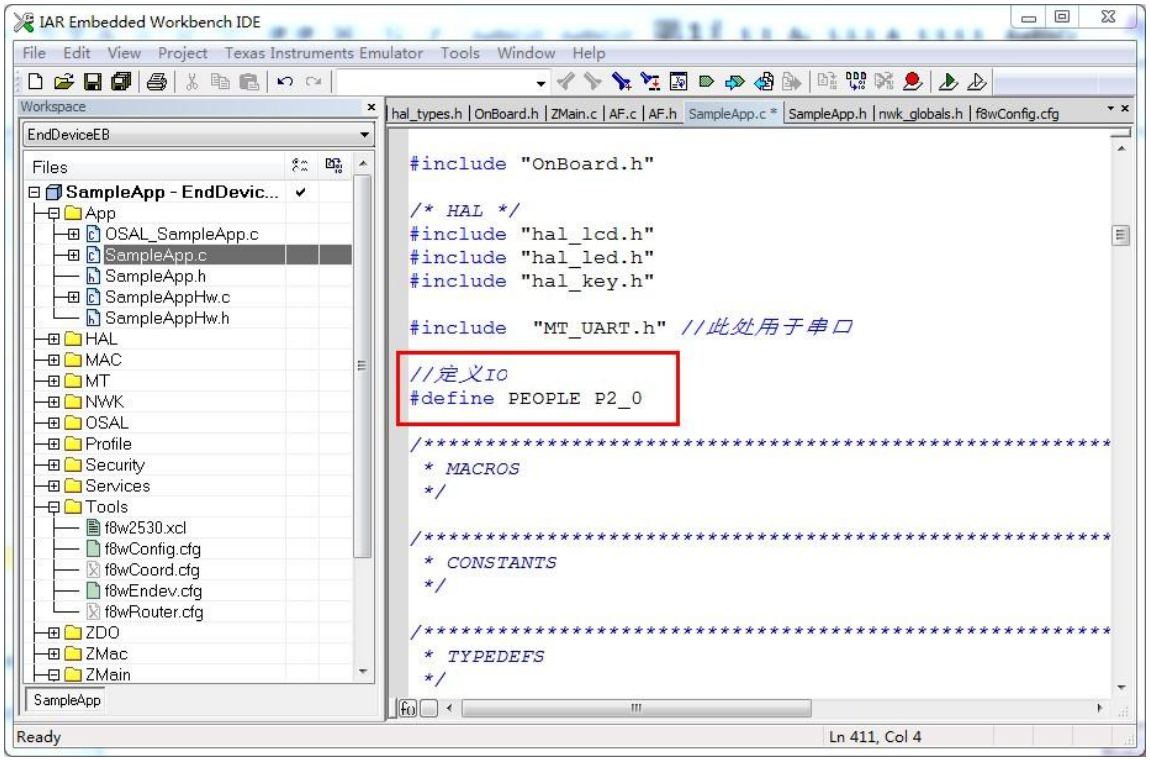


图 3.157

2） 打开例程 SampleApp.eww 工程，打开 SampleApp.c 文件。我们先初始化 P2.0 引脚。设为输入模式。

```
/******红外热释电传感器电路初始化******/
P2SEL &= ~0X01;     //设置 P2.0 为普通 IO 口
P2DIR &= ~0X01;     // 在 P2.0 口，设置为输入模式
P2INP &= ~0x01;     //打开 P2.0 上拉电阻
```

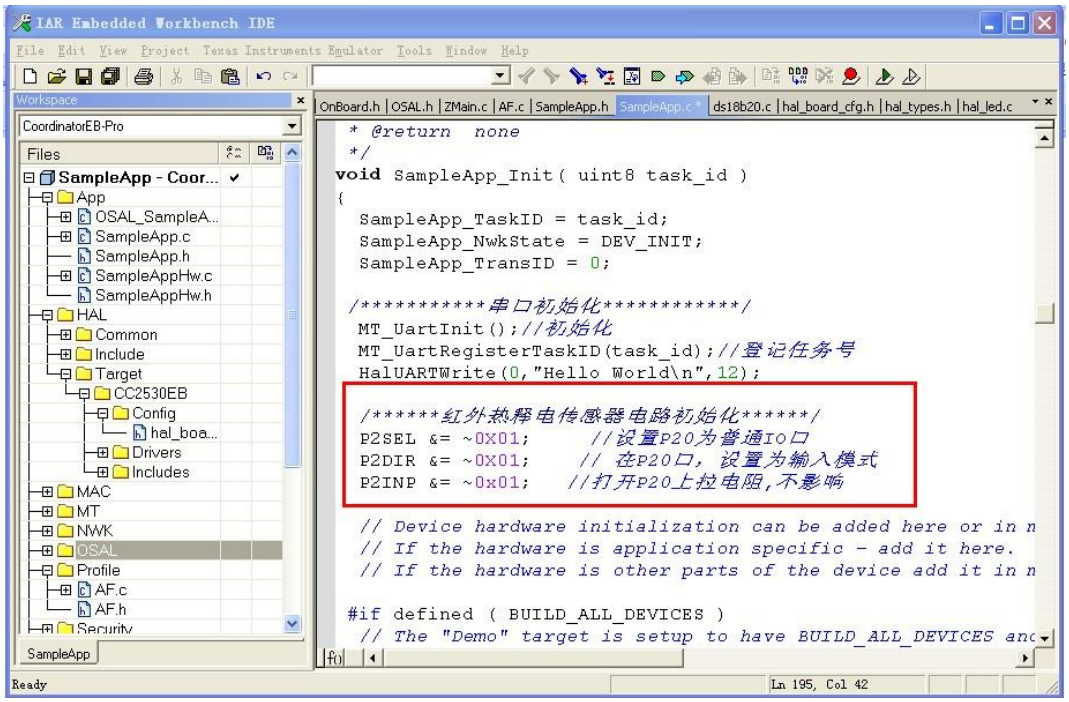


图 3.158

3） 利用周期性点播的定时器作为烟雾信息采集时间，将采集到的信息发送给协调器。并通过 LCD 显示和串口打印。协调器只做串口打印。0.5 秒采集一次。

```
// Send Message Timeout
#define SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT   500     // Every 0.5
                                                      seconds
```

4） 终端每 0.5 秒执行点播函数一次，我们在点播函数里判断 IO 口。加入下面红色代码。

```
void SampleApp_SendPointToPointMessage( void )
{
  uint8 L;
  if(PEOPLE==1)
  {
    L=1; //有人
    HalUARTWrite(0,"Got People\n",11);      //串口
```

```
    HalLcdWriteString( "Got People", HAL_LCD_LINE_3 ); //LCD
  }
  else
  {
    L=0;//没人
    HalUARTWrite(0,"No People\n",10);       //串口
HalLcdWriteString( "No People", HAL_LCD_LINE_3 );//LCD

 if ( AF_DataRequest( &Point_To_Point_DstAddr,
                      &SampleApp_epDesc,
                      SAMPLEAPP_POINT_TO_POINT_CLUSTERID,
                      1,
                      &L,
                      &SampleApp_TransID,
                      AF_DISCV_ROUTE,
                      AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
  {
  }
  else
  {
    // Error occurred in request to send.
  }
```

```
}
```

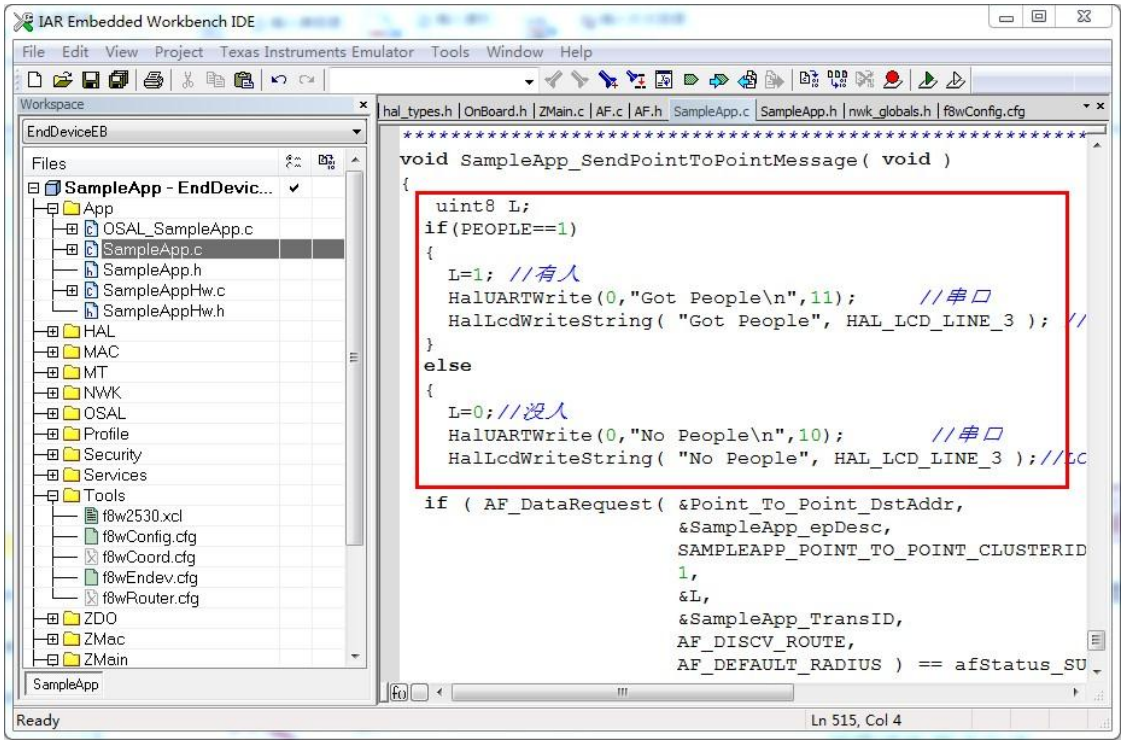


图 3.159

5） 协调器接收函数我们将数据读出来然后判断。通过串口打印传感器信息出来。

```
case SAMPLEAPP_POINT_TO_POINT_CLUSTERID:
    if(pkt->cmd.Data[0])
    HalUARTWrite(0,"Got People\n",11);      //有人
    else
    HalUARTWrite(0,"No People\n",10);       //没人
 break;
```

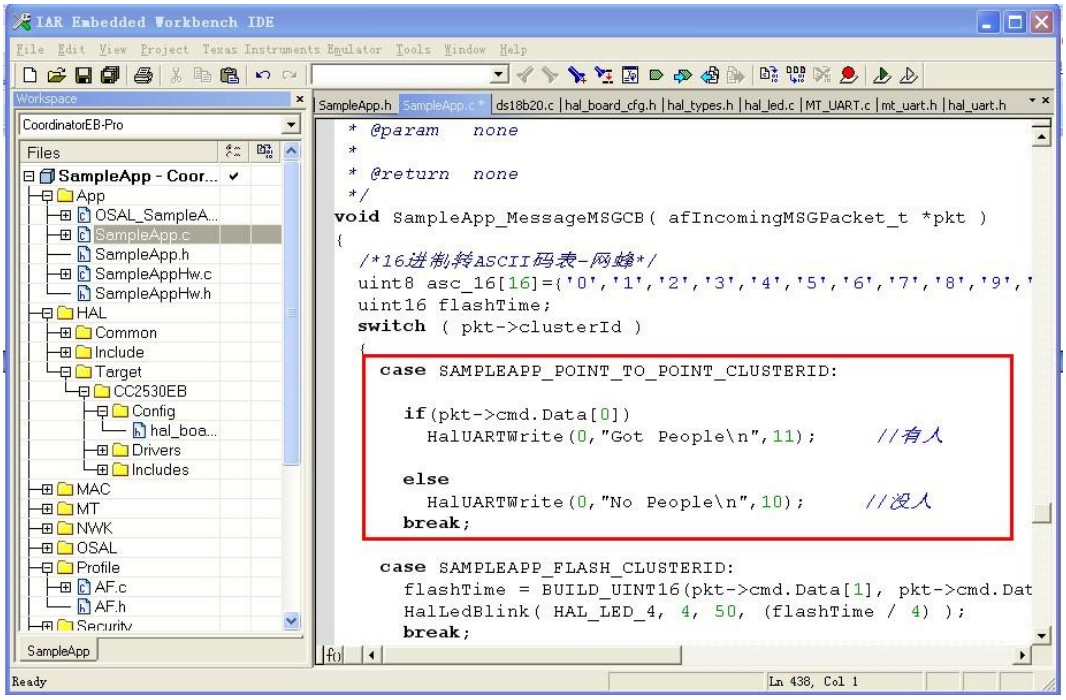


图 3.160

实验现象：下载程序到终端（带红外热释电传感器）和协调器。观察 LCD12864 如下图所示：

图 3.161

协调器收到的信息：

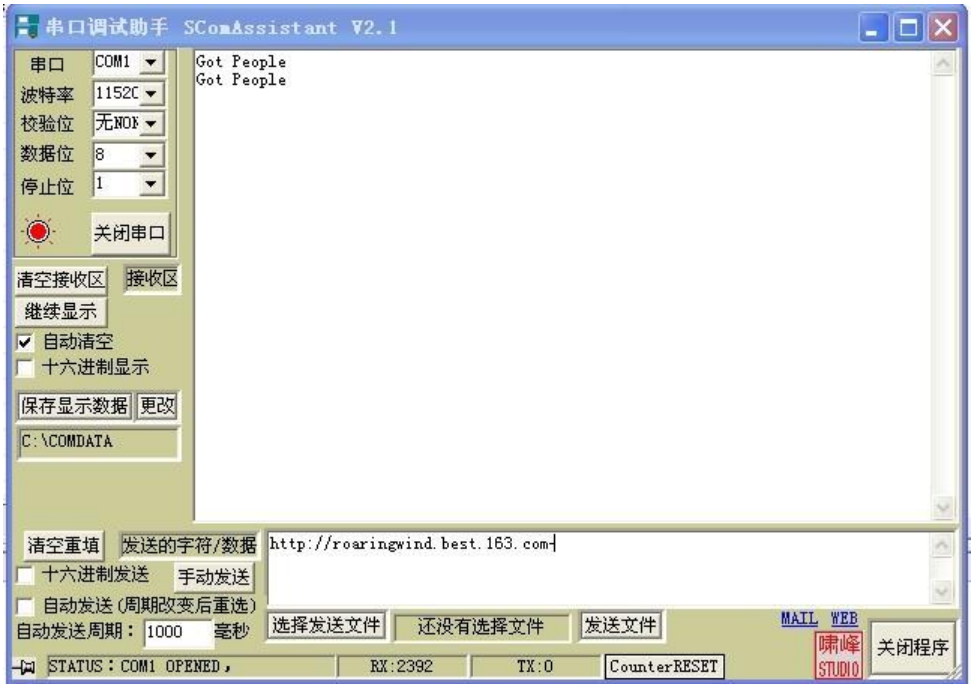


图 3.162

3.11.6 三轴加速度传感器 （文档编写中）

3.12 无线传感网数据采集系统 （文档编写中）

3.13 附录

3.13.1 CC2530+PA（CC2591）模块协议栈的使用方法

在项目或者工业生产中我们经常会用到 PA（功率放大）模块，网蜂推荐使用 TI 官方的 PA 模块，使用 CC2591 芯片。这样我们只要在协议栈里修改一项配置就能实现对带 PA 模块的驱动。所有源程序无需修改。

图 3.163 网蜂 CC2530+CC2591(PA) 核心板

打开 hal_board_cfg.h 文件，找到#define xHAL_PA_LAN。

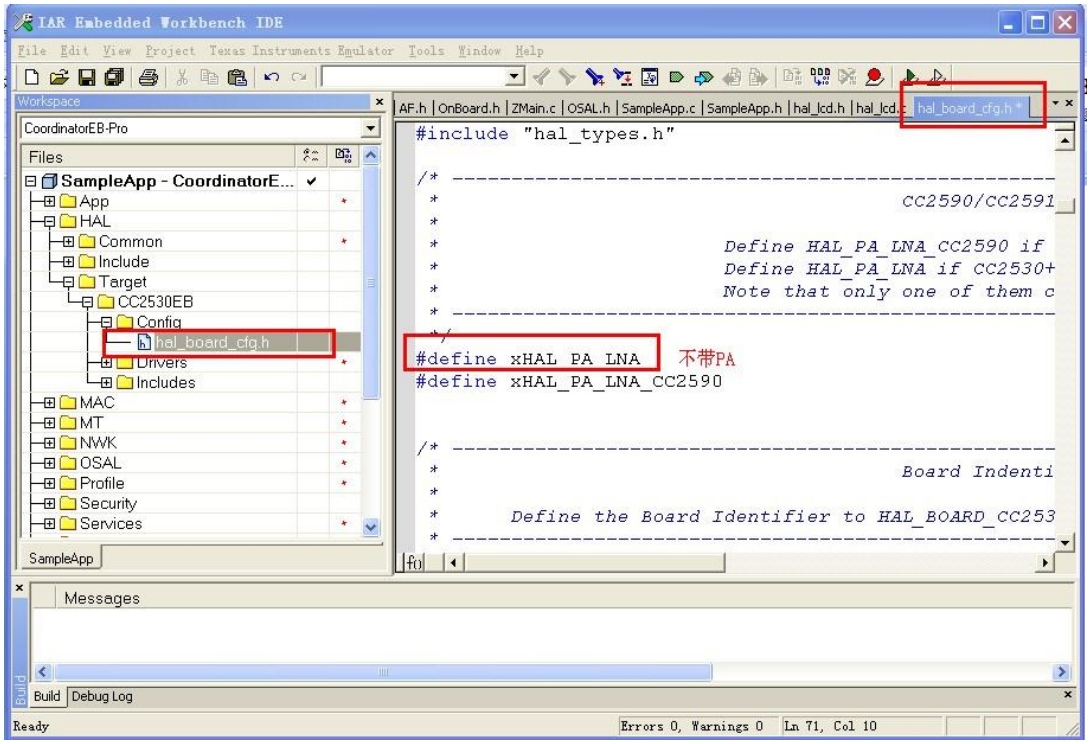


图 3.164 修改前

我们把 x 去掉，编译程序，就可以下载到 PA 模块使用。如图所示：

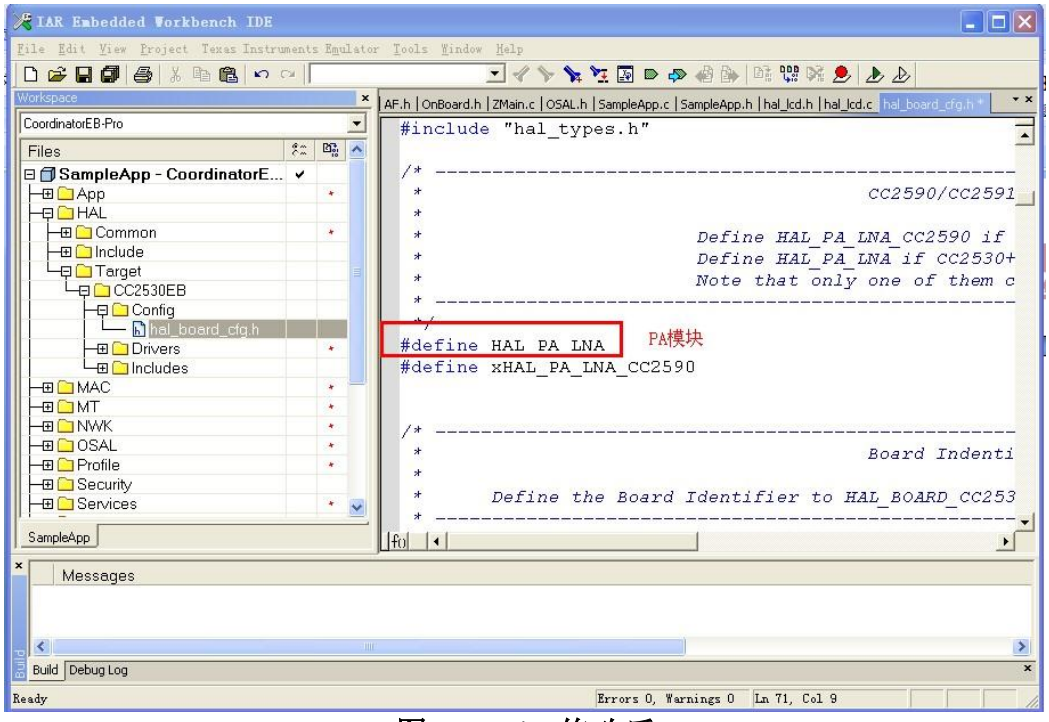


图 3.165 修改后

这样，我们就可以使用高性能的 PA 模块了。

3.13.2 如何在同一地方组建多个 ZigBee 网络

我们很多时候会遇到同一个实验室或者房间内多个人同时学习 ZigBee，那就会存在多个协调器，这样子网络就会相互冲突，解决这个问题的办法就是给不同网络的设备设置自己的 PANID，就可以实现在同一地方组建多个 ZigBee 网络。

我们打开 Tools—f8wConfig.cfg 配置文件，找到：

-DZDAPP_CONFIG_PAN_ID=0xFFFF

由此可见 ZigBee 协议栈网络默认的 PANID 是 0xFFFFF(所有设备都可以加入模式)。

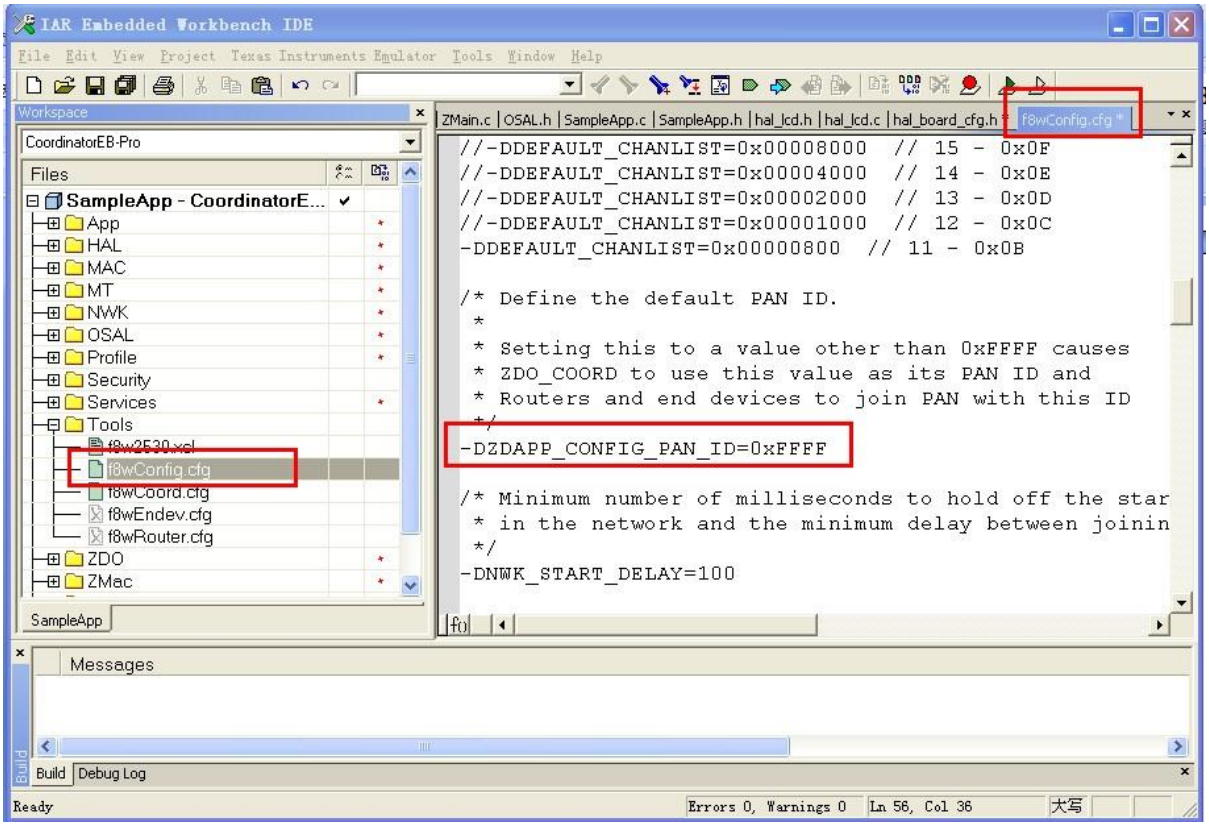


图 3.166 PANID

假设我们现在需要组建 2 组 ZigBee 网络，可以将第一组设备 PANID 设置成 0xFFF0,第二组设置成 0xFFF1,这样 2 个网络建立后就不会相互冲突了。

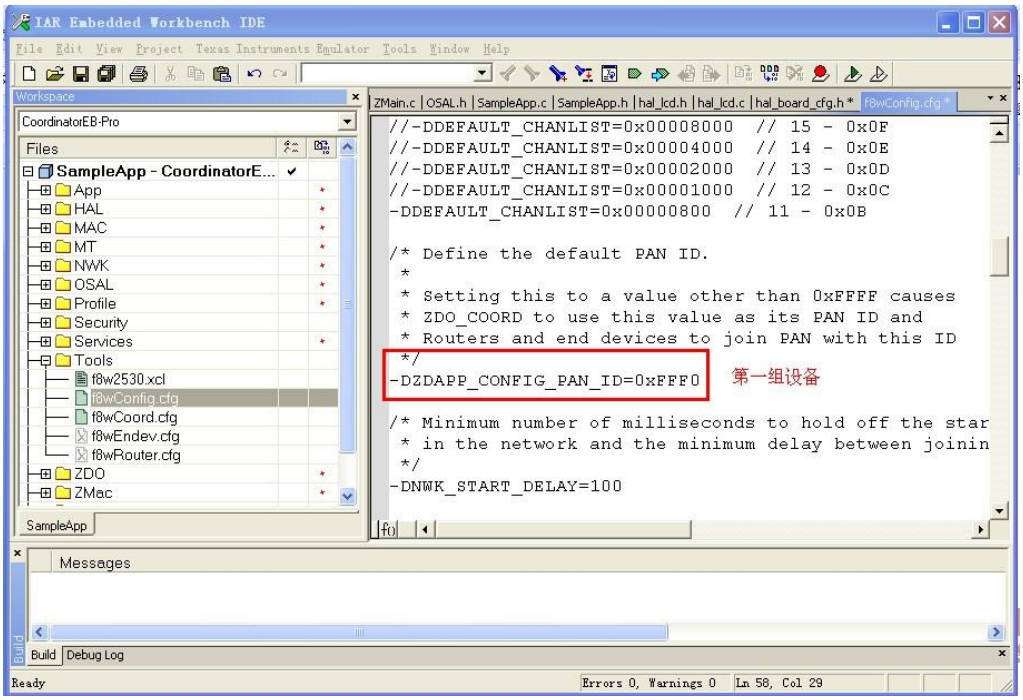


图 3.167 第一组设备

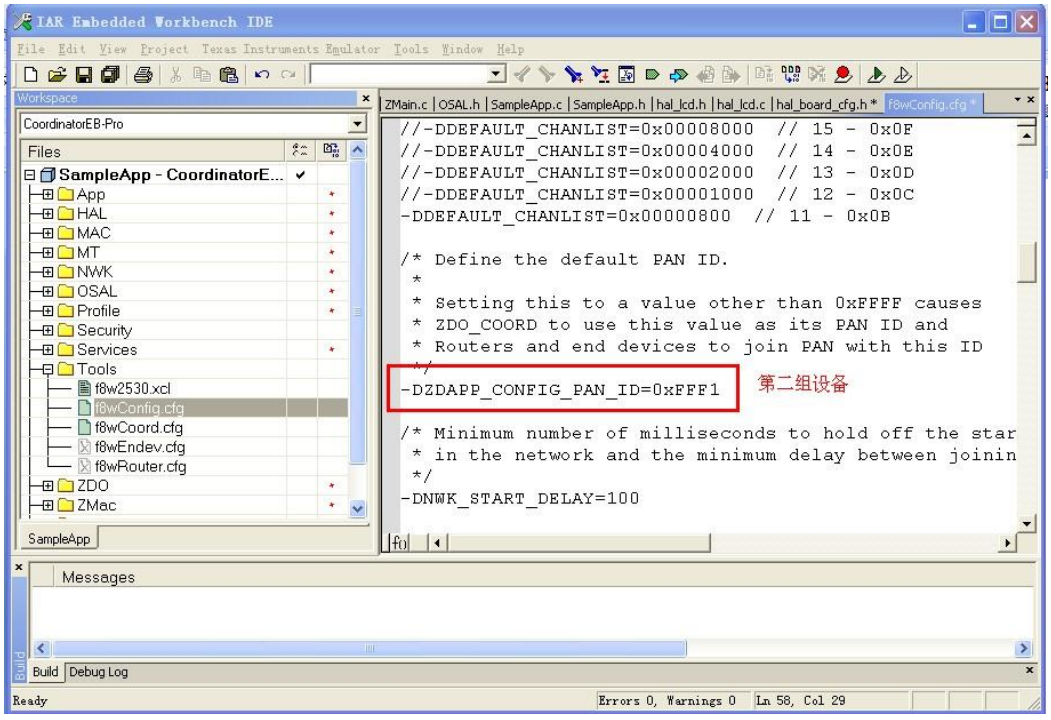


图 3.168 第二组设备

3.13.3 Packet Sniffer 协议栈分析软件使用说明

支持设备：该软件可以使用以下 3 种硬件设备来进行抓包。

1、 Zigbee USB dongle

2、 Smart04EB+ZigBee 节点

3、 CC DEBUGGER+zigbee 节点

以下使用网蜂 zigbee usb dongle 来演示：

第一步：

安装软件： 物联网教学设备\zigbee\生产\网蜂(WeBee) ZigBee 开发套件配套资源\开发软件和驱动\packet_sniffer

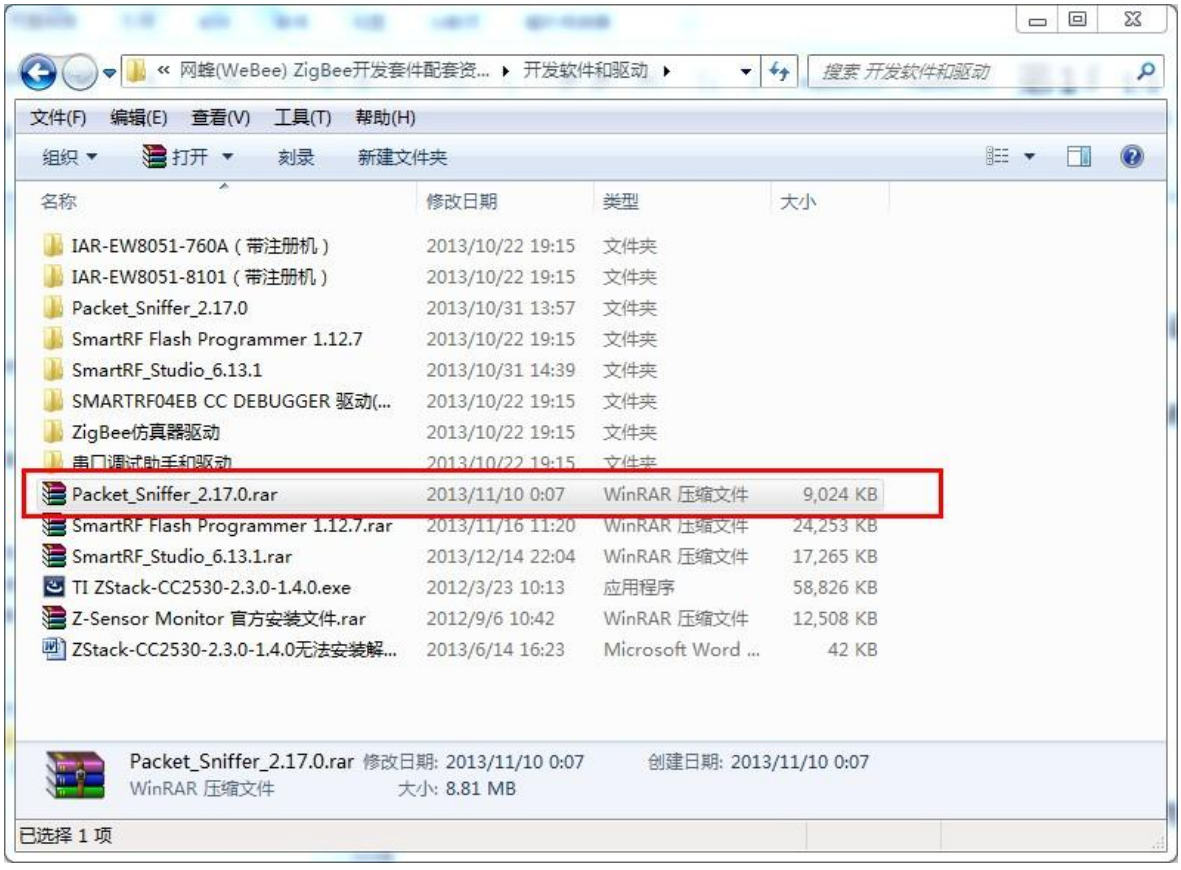


图 3.169 packet sniffer 安装包

安装完成后打开软件，出现下面界面，选择该选项，点击 start：

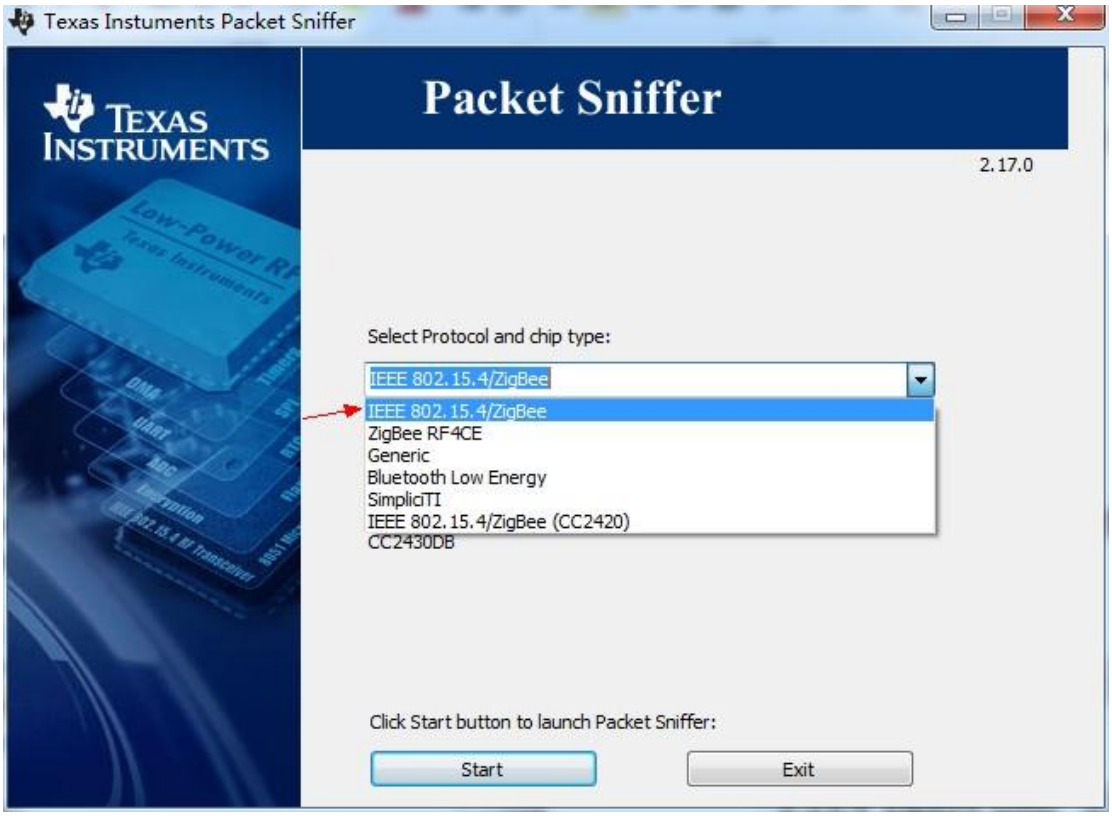


图 3.170 选择上面选项

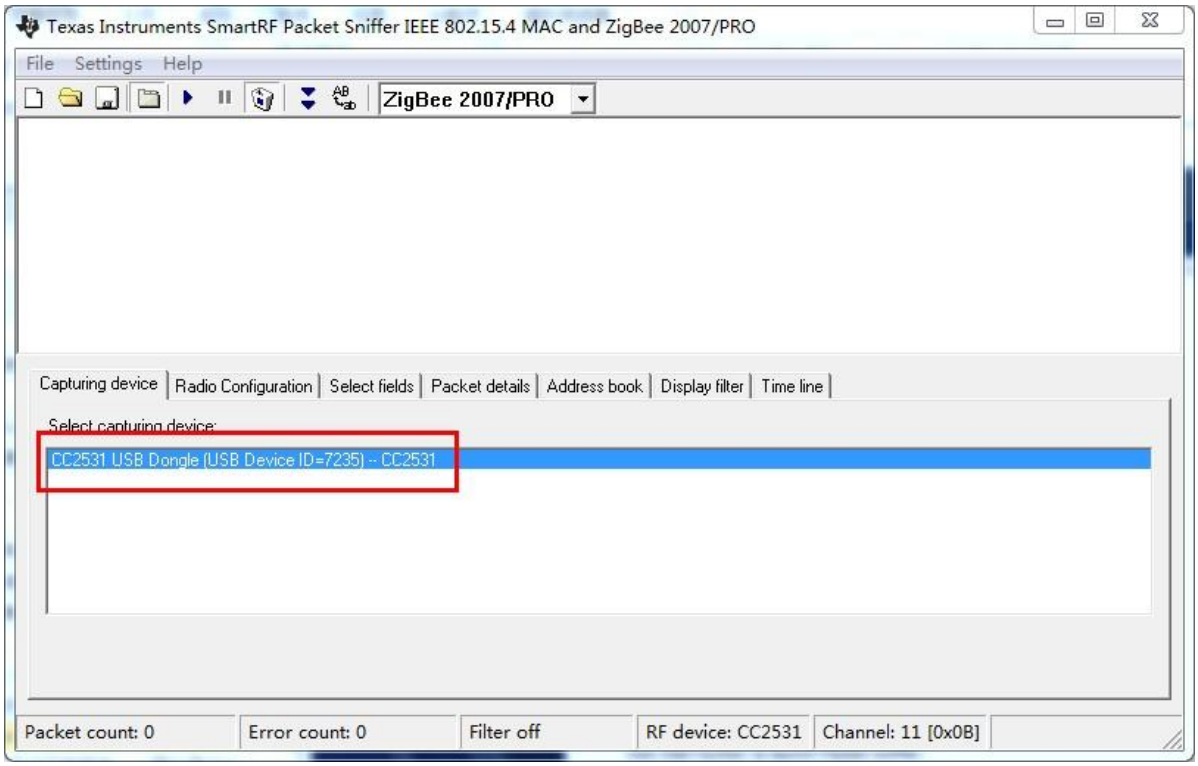


图 3.171 出现 USB DONGLE 设备

点击开始按钮：

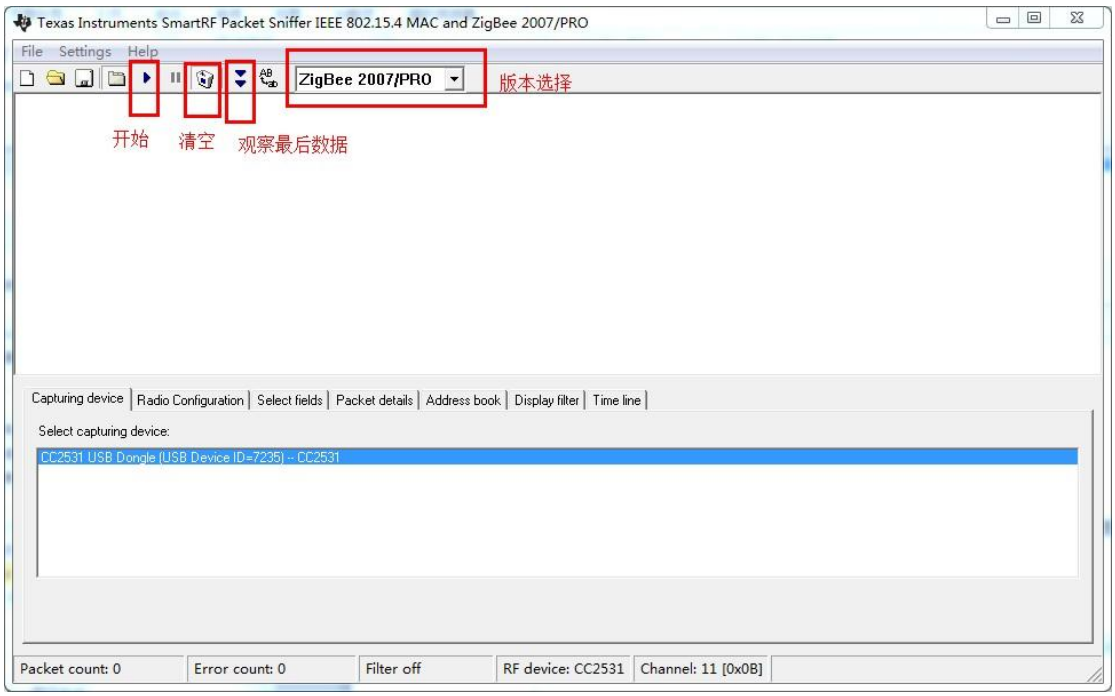


图 3.172 按钮说明

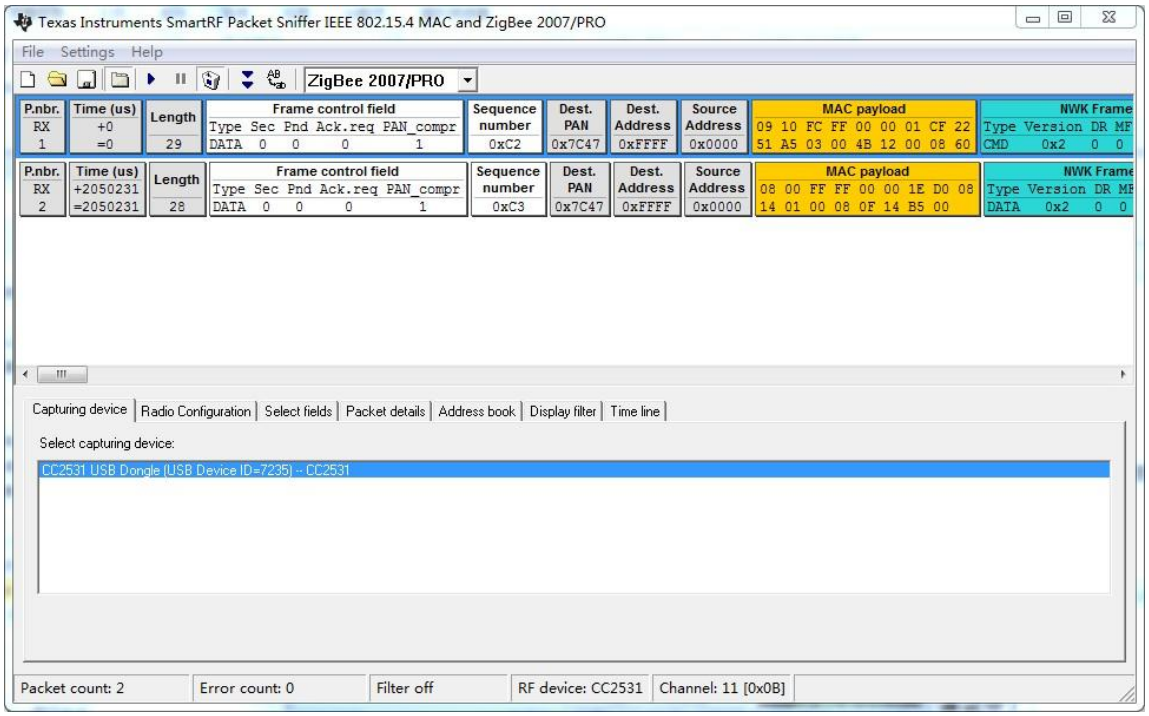


图 3.173 显示数据包

在这里，用户可以根据实际情况自行分析数据包格式。

3.13.4 ZigBee OAD 无线下载程序 （硬件调试中）

第4章 项目实战

4.1 无线点台灯

前言：网蜂陪伴大家走过了 ZigBee 基础实验、组网演练。接下来就进入精彩的项目实战阶段。通过项目的锻炼，相信大家能真正体会到 ZigBee 的魅力。任何一个大项目都从小项目开始，ZigBee 也不例外。网蜂选择了无线点台灯作为第一个项目。无线点灯，实际上是对继电器的操作，ZigBee 节点通过接收到其他设备发来的信息，控制继电器的通断，从而控制台灯的开关。

继电器介绍：

继电器（英文名称：relay）是一种电控制器件，是当输入量（激励量）的变化达到规定要求时，在电气输出电路中使被控量发生预定的阶跃变化的一种电器。它具有控制系统（又称输入回路）和被控制系统（又称输出回路）之间的互动关系。通常应用于自动化的控制电路中，它实际上是用小电流去控制大电流运作的一种“自动开关”。故在电路中起着自动调节、安全保护、转换电路等作用。

图 4.1 继电器

实现平台：网蜂物联网 ZigBee 开发平台、ZigBee 传感器节点；

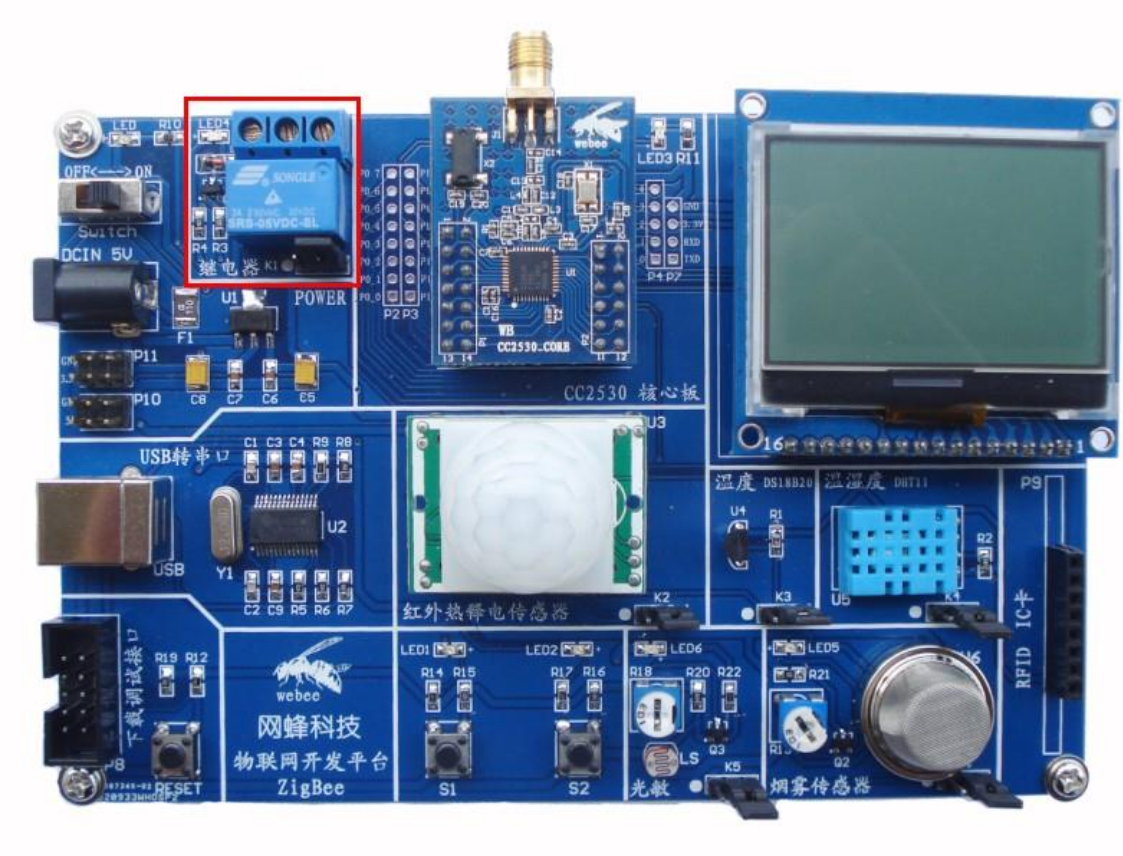


图 4.2 网蜂物联网 ZigBee 开发平台

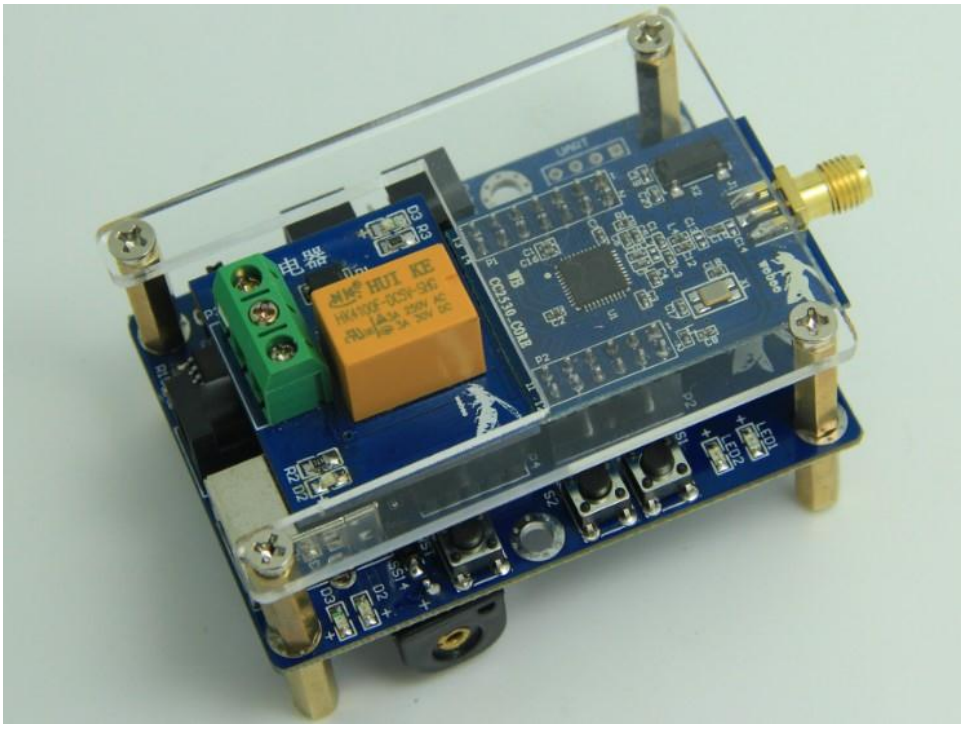


图 4.3 ZigBee 传感器节点

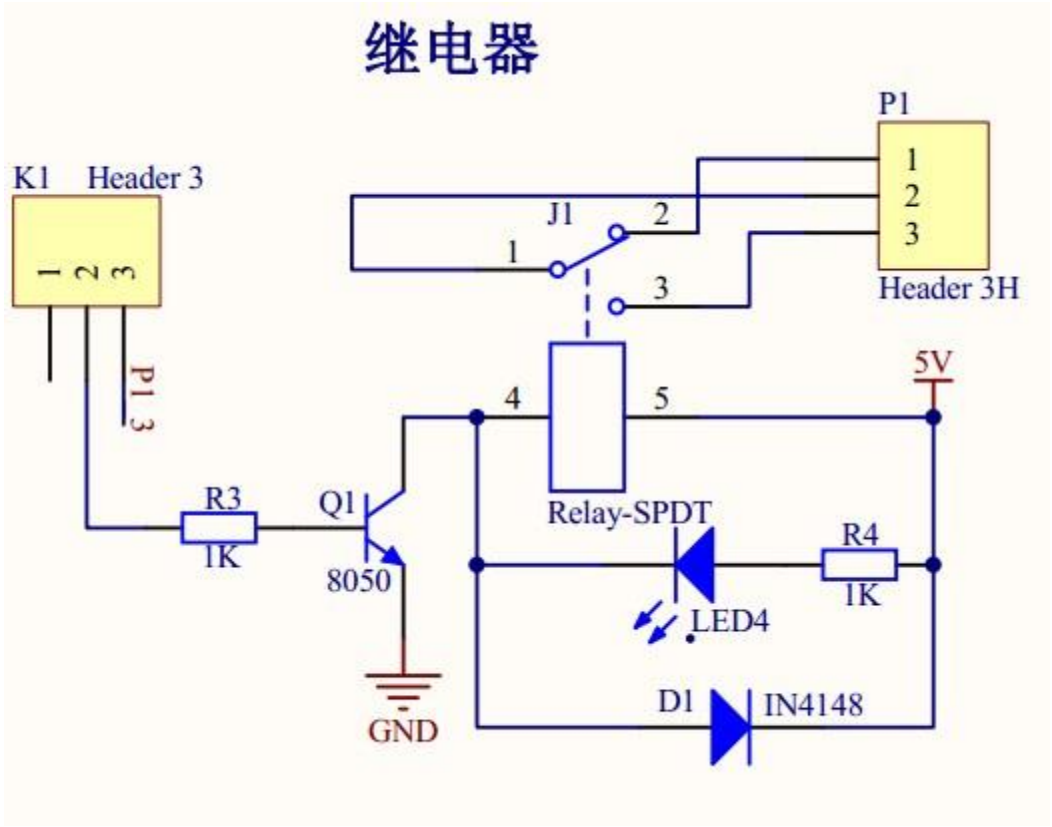


图 4.4 继电器硬件电路图

实验现象：通过节点 1 的按键 S1 控制发送数据给协调器。协调器接收到信号后执行控制继电器指令，从而控制台灯。整个过程在协议栈 Z-STACK 的 SampleApp.eww 上完成。

实验讲解：

项目实战大部分程序内容会基于《zigbee 实战演练》前面章节内容完成，也是对大家之前学习内容一下综合的检测。无线点台灯综合了协议栈的按键控制和继电器控制的应用。这个项目内容简单，但是非常有趣。按键控制部分请参考《ZigBee 实战演练》协议栈中的按键控制章节内容。此项目也是基于该例程完成的,所以按键的配置不再重复。

1）在 SampleApp.c 文件中的 SampleApp_Init 初始化函数初始化继电器的 IO 口 P1.3。设置为输出。

```
/*****继电器 IO 口初始化******/
P1DIR |= 0x08;          //P1_3 定义为输出
P1_3 = 0;               //关闭继电器
```

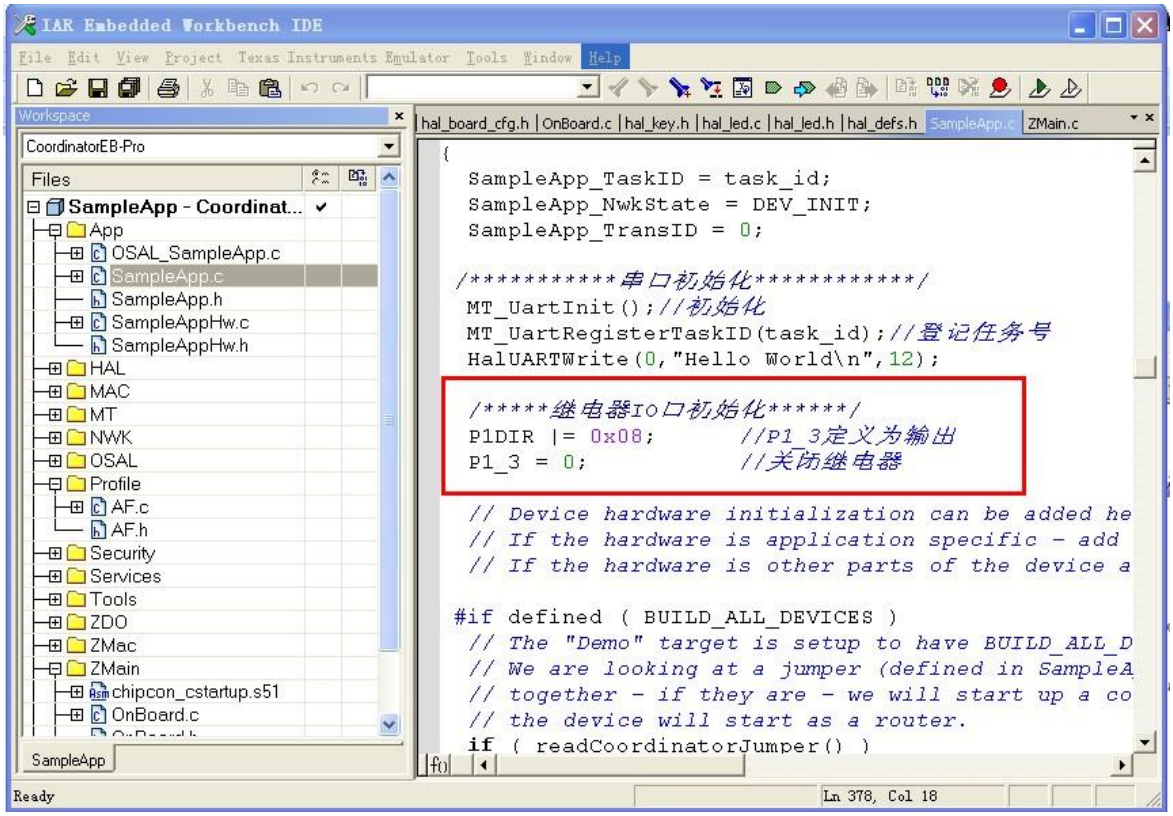


图 4.5

2）系统检测到按键 S1 事件后，串口打印“KEY”做提示。

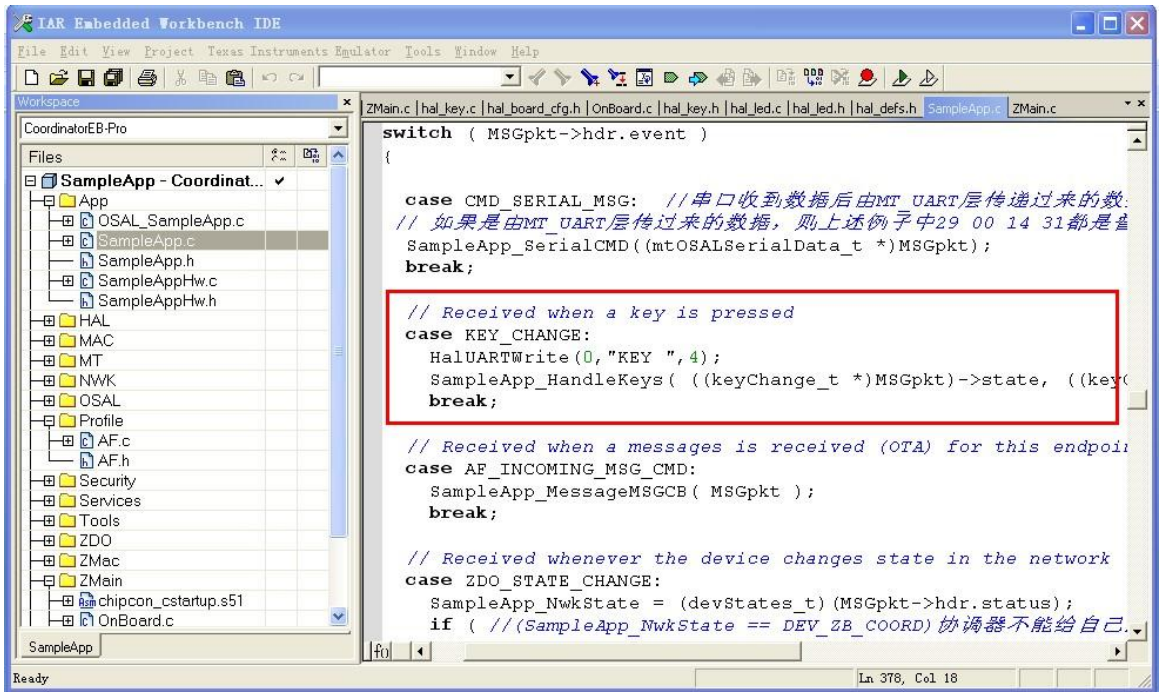


图 4.6

3）继续判断是 S1 按下，执行点播函数，往协调器发送数据指令：

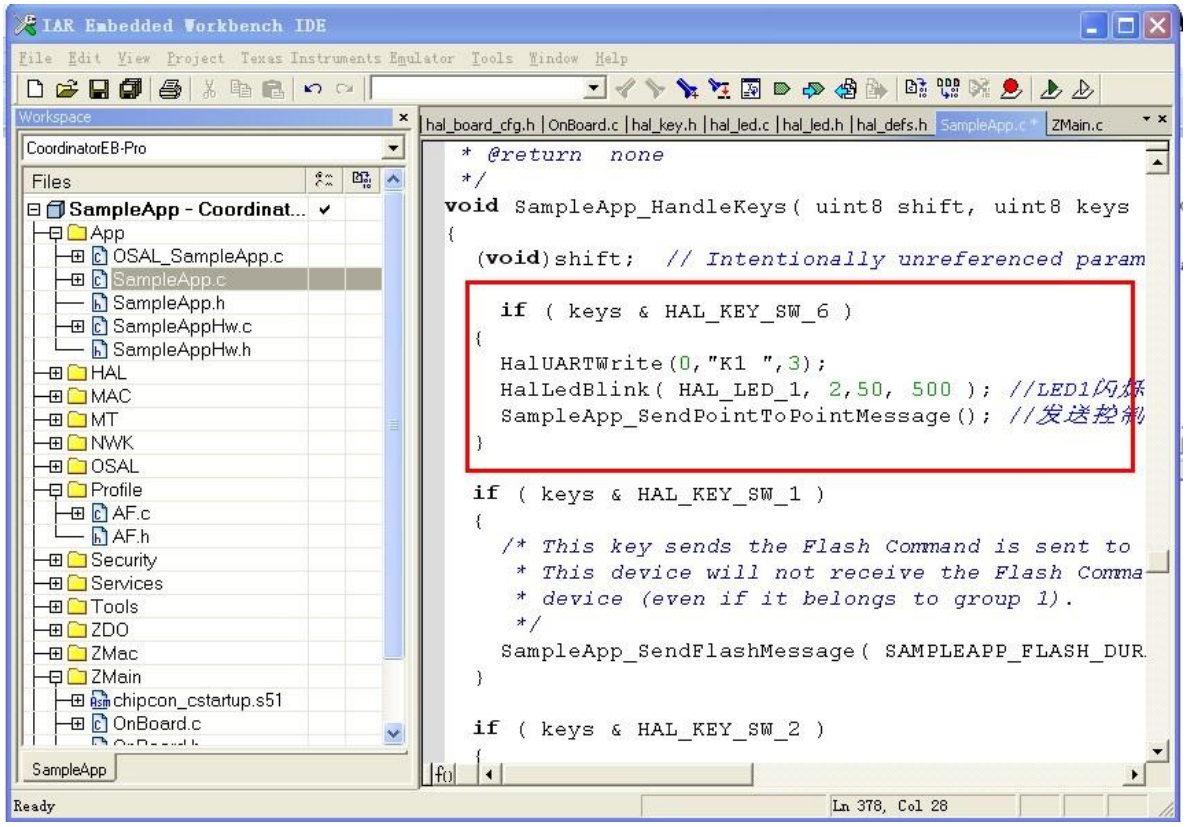


图 4.7

4）我们可以往协调器发送一个“1”，代表控制继电器。

```
void SampleApp_SendPointToPointMessage( void )
{
  uint8 data=1;
  if ( AF_DataRequest( &Point_To_Point_DstAddr,
                       &SampleApp_epDesc,
                       SAMPLEAPP_POINT_TO_POINT_CLUSTERID,
                       1,
                       &data,
                       &SampleApp_TransID,
                       AF_DISCV_ROUTE,
                       AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
  {
  }
  else
  {
    // Error occurred in request to send.
  }
```

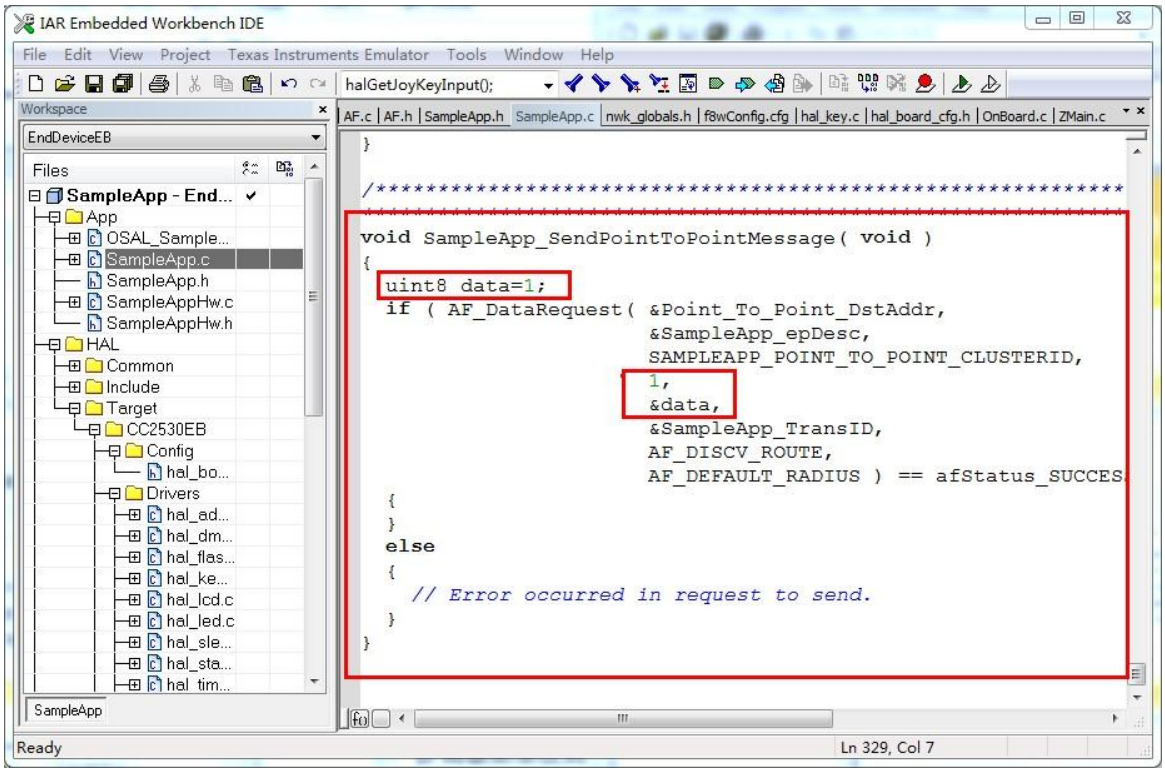


图 4.8

5）协调器如果收到“1”，则执行改变连接继电器电路的 IO 口状态：

```
P1_3= ~P1_3;    //改变继电器状态
```

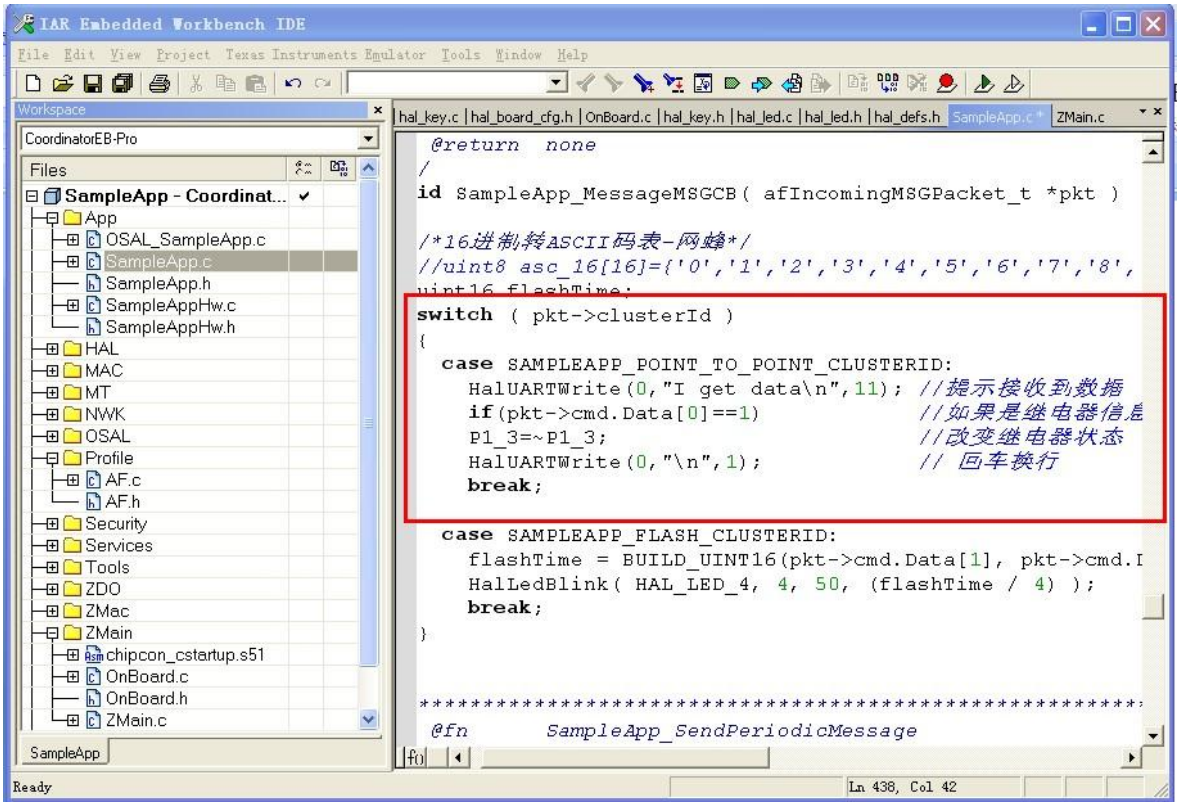


图 4.9

至此，实验代码全部完成。

实验现象：

我们把代码分别以协调器（带继电器的 ZigBee 设备）和终端的方式下载到设备，上电，按下终端节点的 S1，可以看到协调器节点上的继电器导通。

按下终端的 S1 按键，观察协调器的继电器指示灯变化：

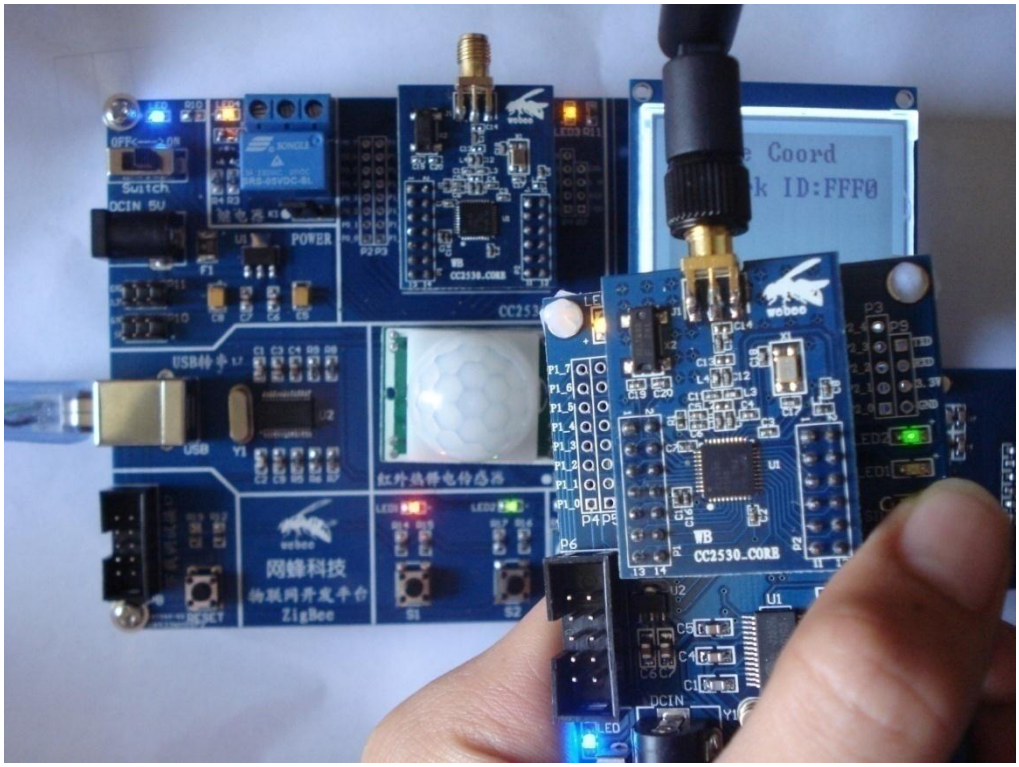


图 4.10

继电器硬件指示灯：

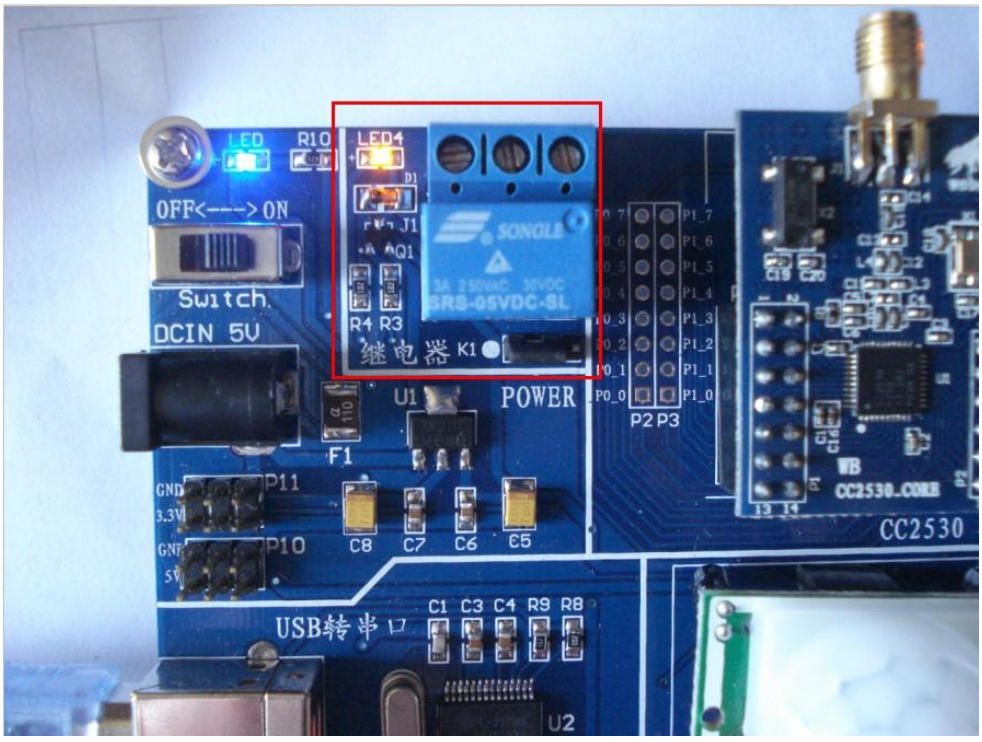


图 4.11 继电器导通，指示灯亮

有兴趣的连接到上自己的台灯，便实现了无线点灯的功能。

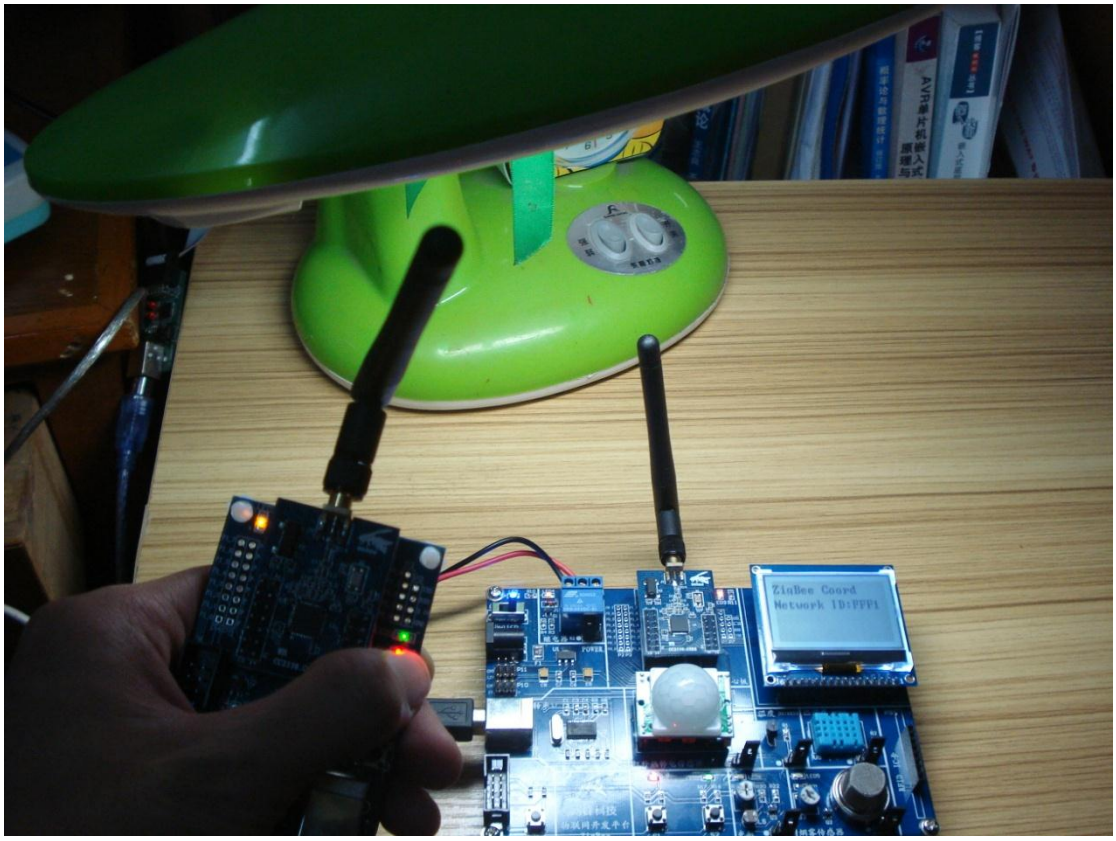


图 4.12 无线点台灯

结语：或许我们已经感受到了无线的魅力，赶紧动手一下吧。打造自己的无线点台灯或者更多的无线产品。

4.2 无线 IC 卡考勤机

前言： 前面无线点台灯的应用给大家做了个敲门砖，今天我们继续我们的项目实验，这次项目的内容很有意思，相信大家也会喜欢，那就是 IC 卡考勤机，传统的 IC 卡打卡机我们见到很多，有了 ZigBee 后，它就可以变身成为无线 IC 卡考勤机，免去了布线的繁琐，在无线应用方面前景也很广。

IC 卡介绍：

IC 卡 (Integrated Circuit Card，集成电路卡)，有些国家和地区也称智能卡(smart card)、智慧卡(intelligent card)、微电路卡(microcircuit card)或微芯片卡等。它是将一个微电子芯片嵌入符合 ISO 7816 标准的卡基中，做成卡片形式。 IC 卡读写器是 IC 卡与应用系统间的桥梁，在 ISO 国际标准中称之为接口设备 IFD(Interface Device)。IFD 内 CPU 通过一个接口电路与 IC 卡相连并进行通信。IC 卡接口电路是 IC 卡读写器中至关重要的部分，根据实际应用系统的不同，可选择并行通信、半双工串行通信和 I2C 通信等不同的 IC 卡读写芯片。

非接触式 IC 卡又称射频卡，成功地解决了无源（卡中无电源）和免接触这一难题，是电子器件领域的一大突破。主要用于公交、轮渡、地铁的自动收费系统，也应用在门禁管理、身份证明和电子钱包。

图 4.13　IC 卡考勤机

实现平台：网蜂物联网 ZigBee 开发平台。

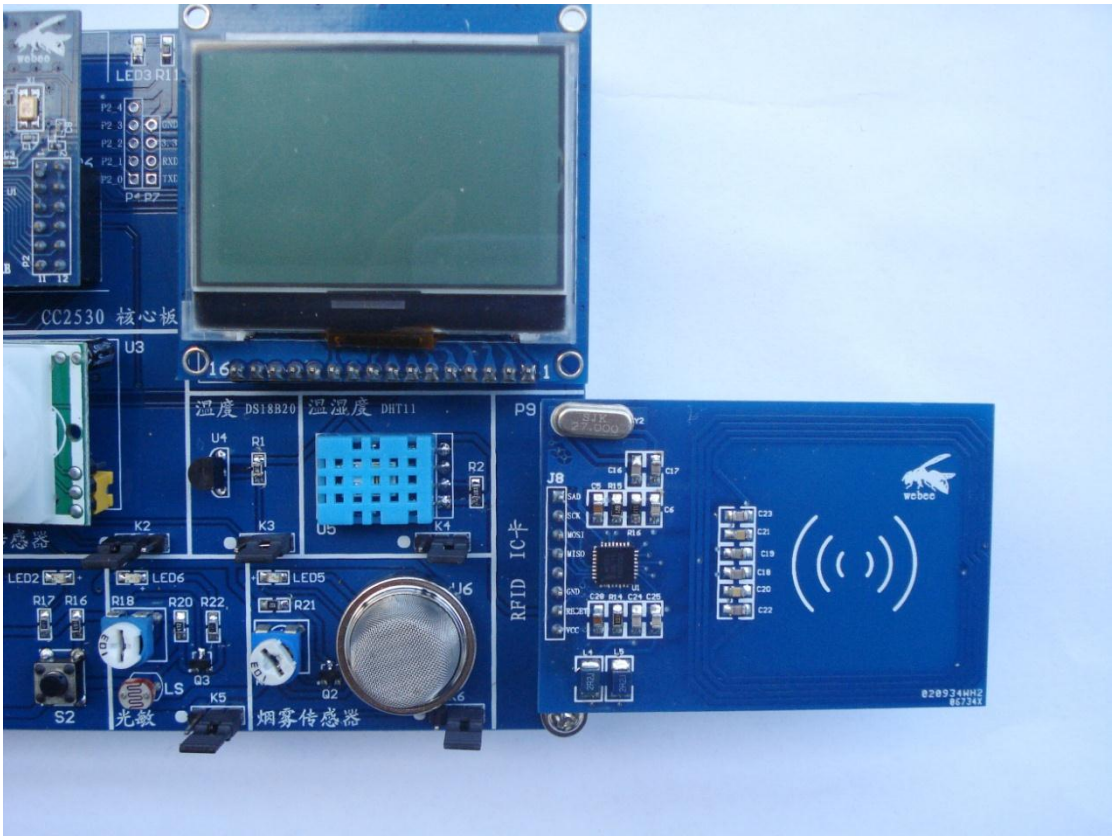


图 4.14 网蜂物联网 ZigBee 开发平台

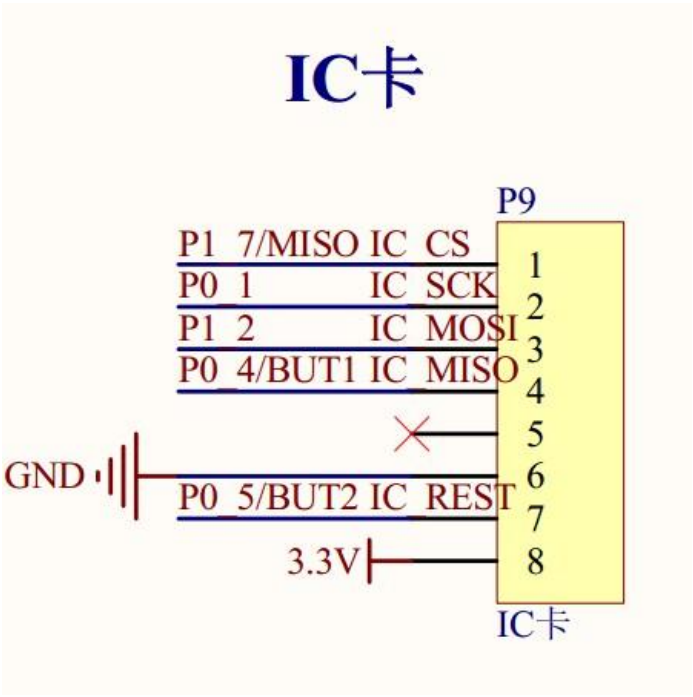


图 4.15 网蜂 IC 卡模块接口电路图

实验功能：带网蜂 IC 卡模块的 ZigBee 节点通过感应 IC 卡，将采集到的 IC 卡的 ID 编号远程发送给协调器， 协调器接收后通过串口打印出来。整个实验在 SampleApp.eww 工程完成。

实验讲解：IC 卡的操作包括防冲突处理，读卡号、读/写数据。在这里，我们完成的仅仅是前面 2 项功能，有兴趣的可以继续深入，完成 IC 卡的读/写数据操作。

实验过程：分三个步骤，如下：

一：在裸机上完成对 IC 卡的的防冲突处理，读卡号。

二：在协议栈上完成程序相关功能，终端读到 IC 卡号后无线发送到协调器。

一：在裸机上完成对 IC 卡的的防冲突处理，读卡号。

我们先来看看 IC 卡的读写流程，再通过代码控制 CC2530 完成对 IC 卡的操作。

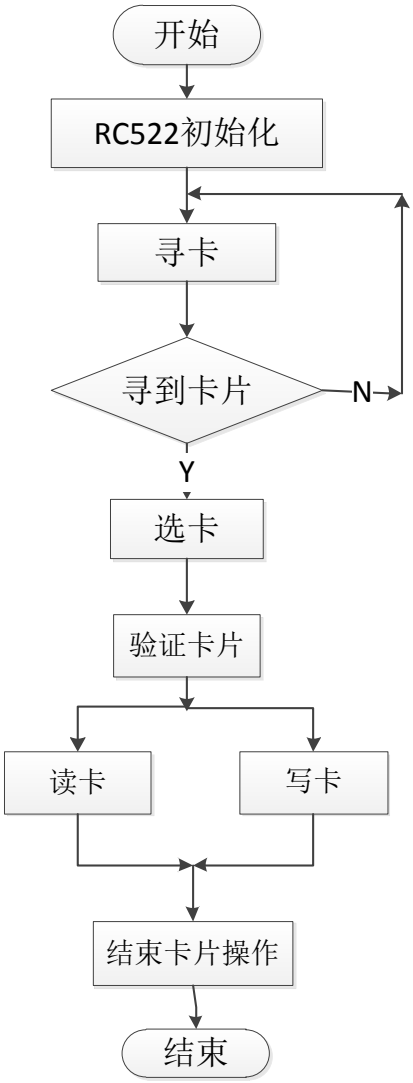


图 4.16 RC522 工作流程

部分核心代码如下，详细请参考网蜂配套程序：

```
/*************************************/
/*              WeBee 团队                */
/*          Zigbee 学习例程               */
/*例程名称：IC 卡打卡机                  */
/*建立时间：2012/10                      */
/*描述：读取 IC 卡 32 位 ID 号，通过串口打印出来
*************************************/
```

```
1. #include “variable.h”
2. #include” delay.h”
3. #include “KEY.h”
4. #include “UART.h”
5. #include “IC_w_r.h”

6. void Initial()

7. {
8.     CLKCONCMD &= ~0x40;              //设置系统时钟源为 32MHZ 晶振
9.     while(CLKCONSTA & 0x40);         //等待晶振稳定为 32M
10.    CLKCONCMD &= ~0x47;              //设置系统主时钟频率为 32MHZ
11.    UartInitial();
12.
13.    …
14.    …
       IC_SCK = 1;
       IC_CS = 1;
15. }


16. void IC_test()
17. {
18.    uint
19.    uchar qq[4];
20.    uchar find=0xaa;
21.    uchar ar;
22.
23.    while(1)
```

```
24.   {
25.     /***16 进制转 ASC 码*****/
26.   uchar
27. asc_16[16]={ ‘0’ ,’ 1’ ,’ 2’ ,’ 3’ ,’ 4’ ,’ 5’ ,’ 6’ ,’ 7’ ,’ 8’
    ,’ 9’ ,’ A’ ,’ B’ ,’ C’ ,’   D’ ,’ E’ ,’ F’ },I;
28.    uchar Card_Id[8]; //存放 32 位卡号
29.    ar = PcdRequest(0x52,qq);//寻卡
30.    if(ar != 0x26)
31.    ar = PcdRequest(0x52,qq);
32.    if(ar != 0x26)
33.    find = 0xaa;
34.    if((ar == 0x26)&&(find == 0xaa))
35.    {
36.      if(PcdAnticoll(qq) == 0x26);//防冲撞
37.      {
38.        UartSend_String(“The Card ID is: “,16);
39.
40.       /****16 进制转 ASC 码********/
41.       for(i=0;i<4;i++)
42.       {
43.         Card_Id[i*2]=asc_16[qq[i]/16];
44.         Card_Id[i*2+1]=asc_16[qq[i]%16];
45.        }
46.         UartSend_String(Card_Id,8); //打印 IC 卡号
47.         UartSend_String(“\n”,1);
48.         find = 0x00;
49.       }
50.    }
51.  }
```

```
52. }

53. void main()

54. {
55.   Initial();
56.   PcdReset();
57.    M500PcdConfigISOType( 'A' );   //设置工作方式
58.     while(1)
59.     {
60.       IC_test();//IC 卡检测
        }
61. }
```

我们留意主函数代码：

第 55~56 行：初始化工作。

第 57 行：设置 IC 卡模块工作方式。

第 60 行：检测 IC 卡。

大家可以在工程里进入具体函数看代码，理解 IC 卡初始化及读取卡号的过程。

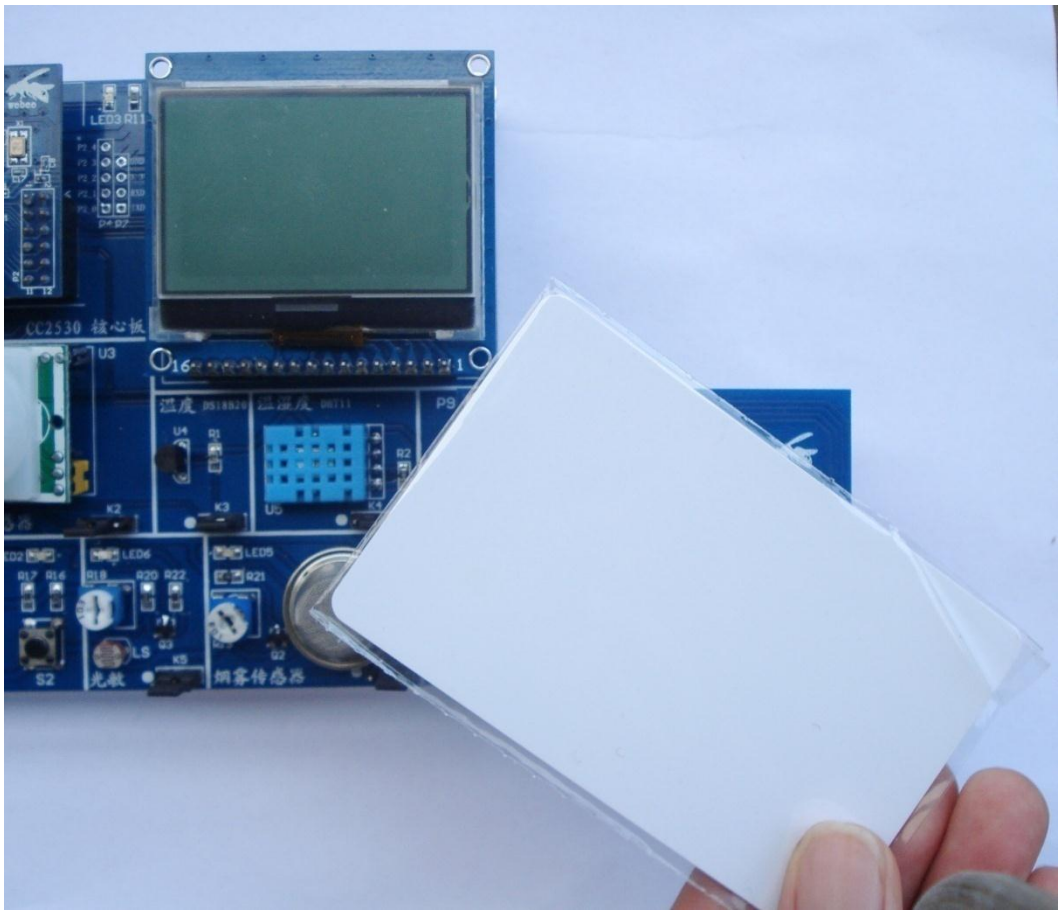


图 4.17 将 IC 卡放到网蜂 IC 卡模块

实验现象如下图所示：

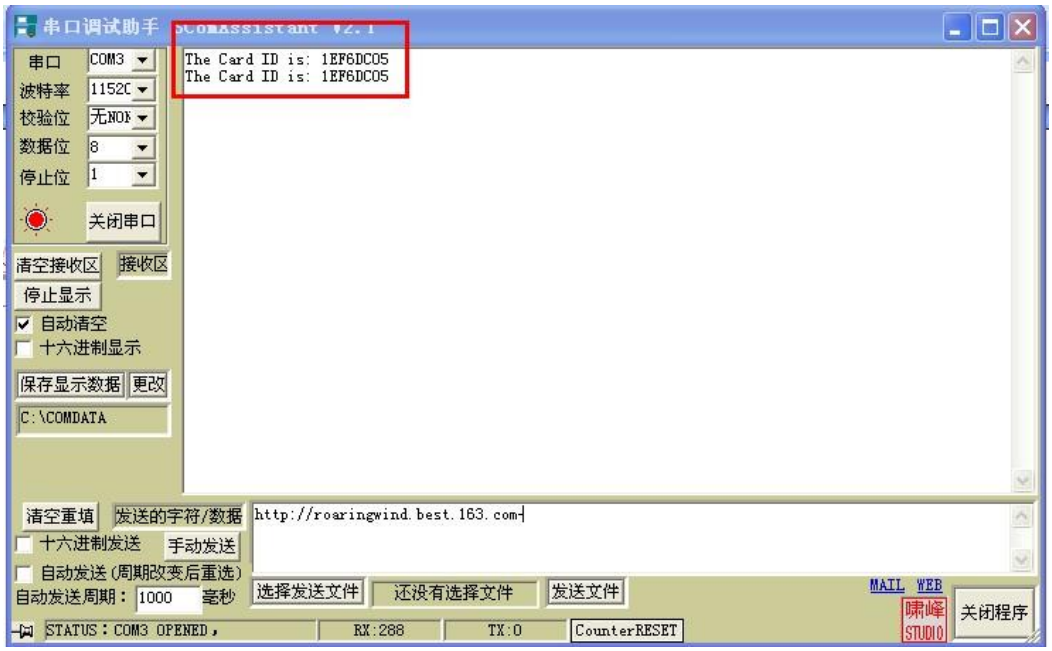


图 4.18 串口打印出卡号

二:在协议栈上完成程序相关功能,终端读到 IC 卡号后无线发送到协调器。

上面我们已经完成了用 CC2530 驱动 IC 卡模块，现在为了完成我们的无线打卡功能，我们将 IC 卡程序移植到协议栈 SampleApp.eww 工程，实现数据的收发。

软件流程是协议栈每 400ms 执行一次 IC 卡扫描程序，如果发现 IC 卡，则将卡号发送给协调器，否则不进行数据的发送。协调器收到 IC 卡号后通过串口打印出来。

1） 首先，我们将裸机程序里的文件复制到协议栈\Texas Instruments 无线 IC 卡考勤机\Zstack-CC2530-2.5.1a\Projects\zstack\Samples\SampleApp\Source 路径下。

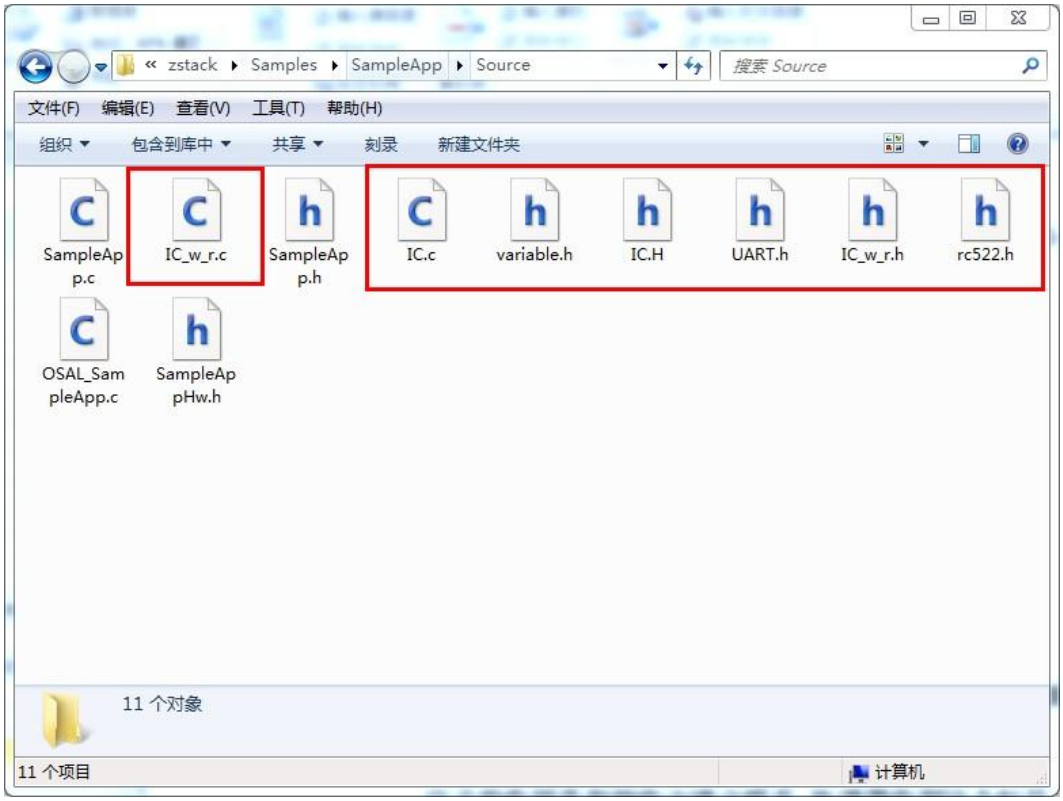


图 4.19

2） 在 SamleApp.eww 工程的 APP 路径下添加 C 文件：

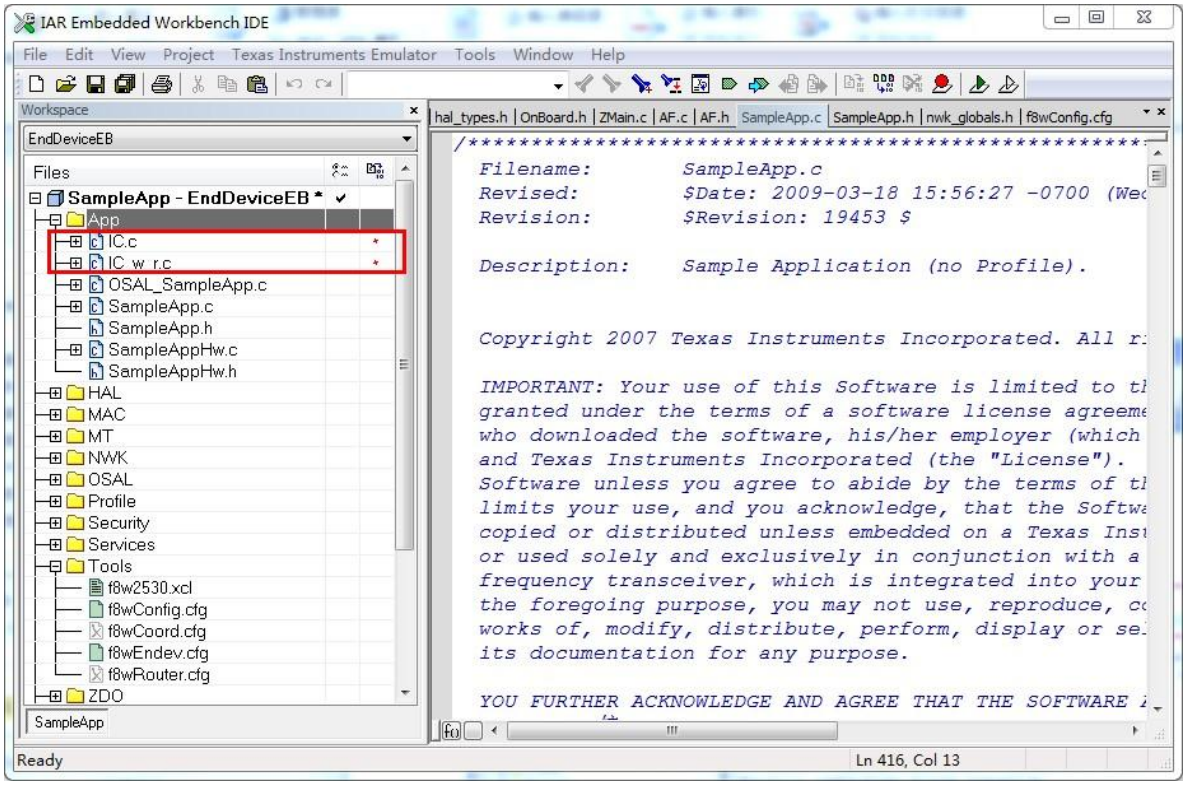


图 4.20

3） 在 SampleAPP.c 中加入 IC.h 头文件定义：

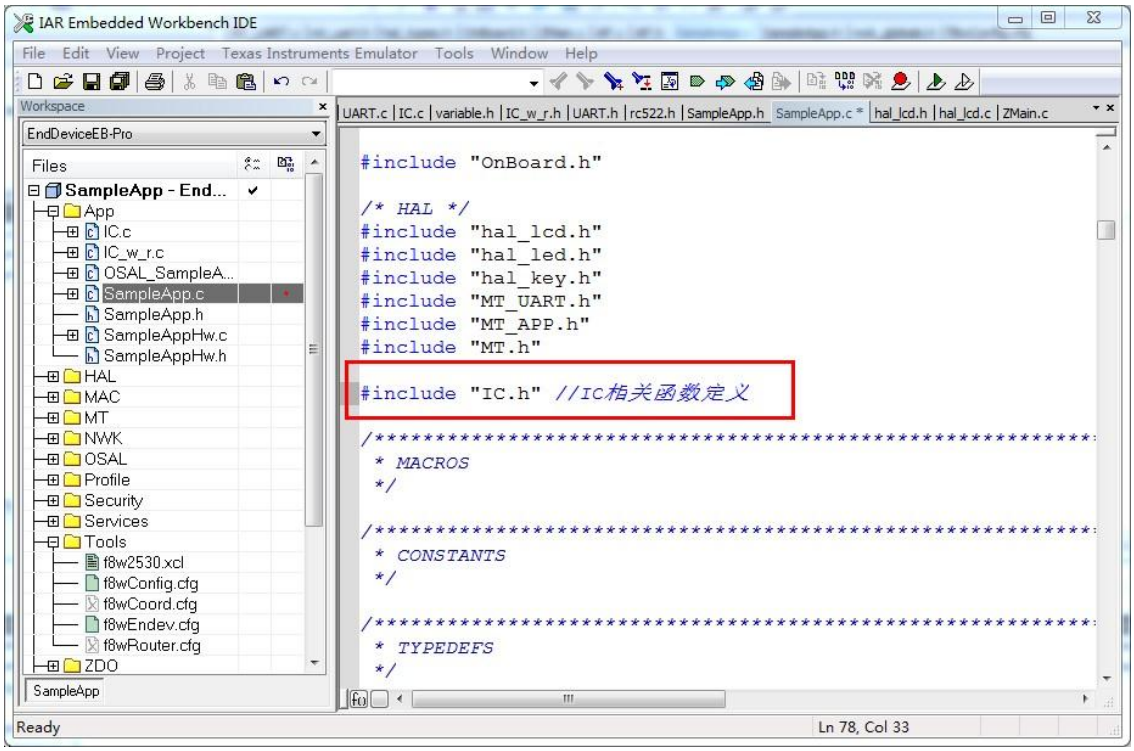


图 4.21

4） 加入 IC 模块初始化函数：

```
IC_Init(); //初始化 IC 模块
```

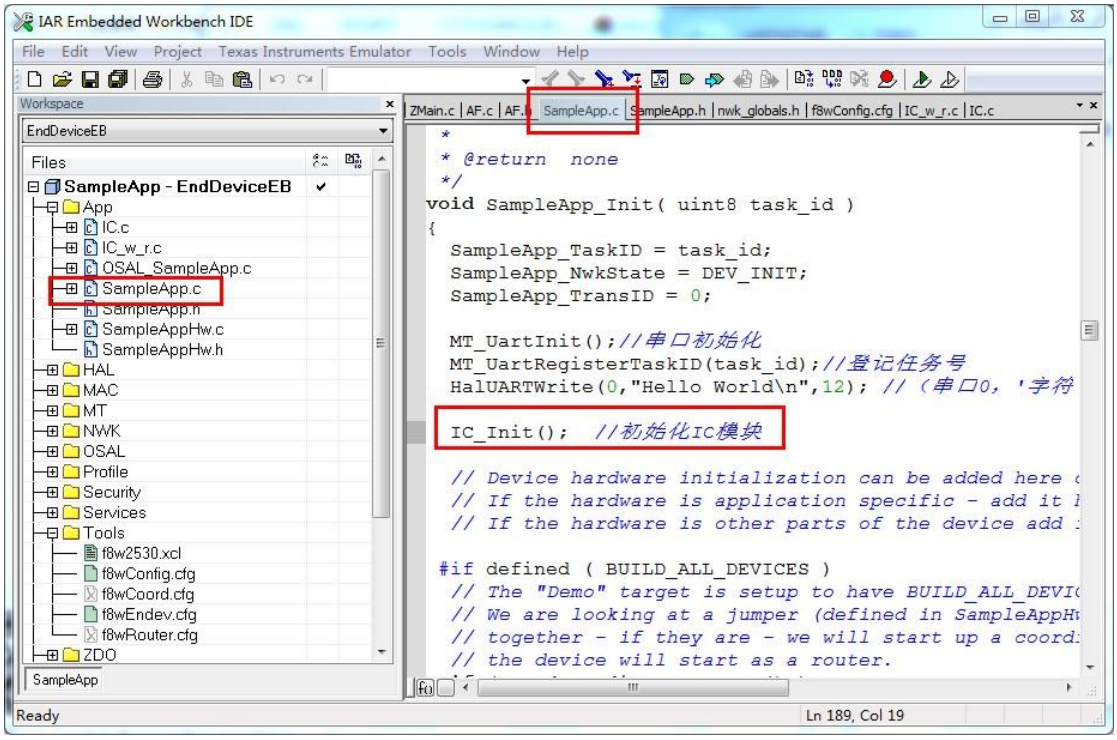


图 4.22

5） 我们还是采用 IC 卡定时扫描模式，每 400ms 扫描一次。

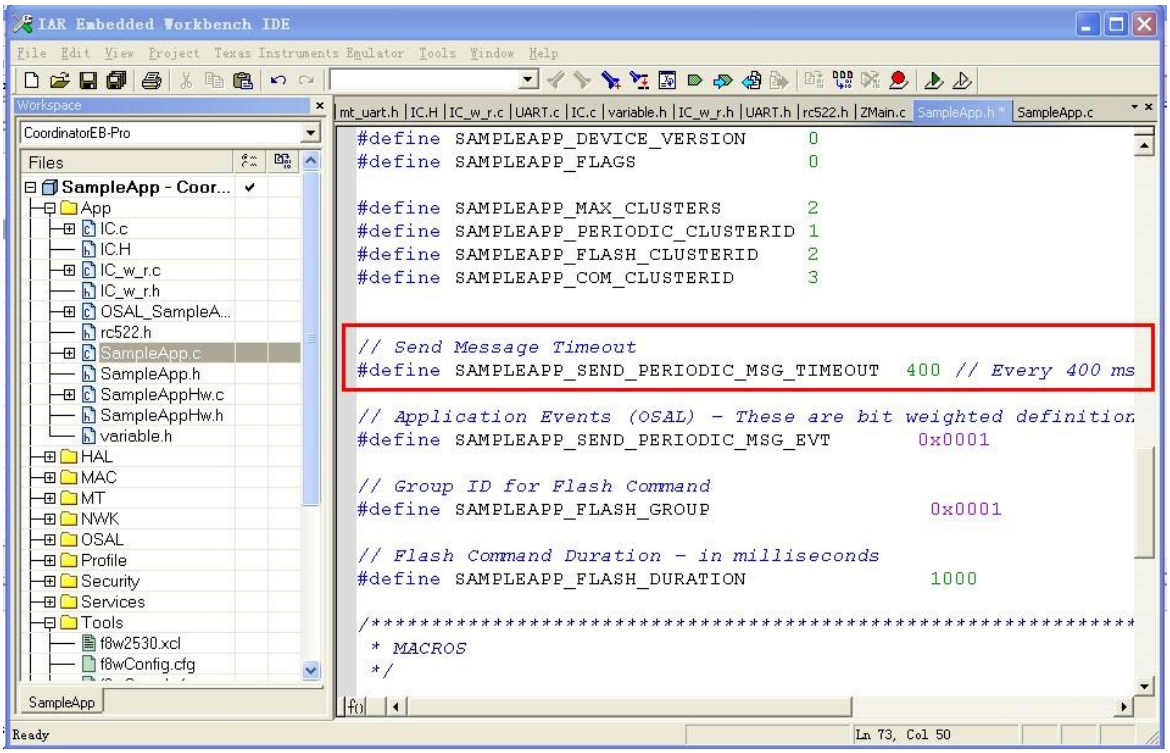


图 4.23

6） 终端每 400ms 执行一次广播函数，实际上对 IC 卡进行一次扫描。我们在点播函数里面执行 IC 卡读取和信息发送程序。

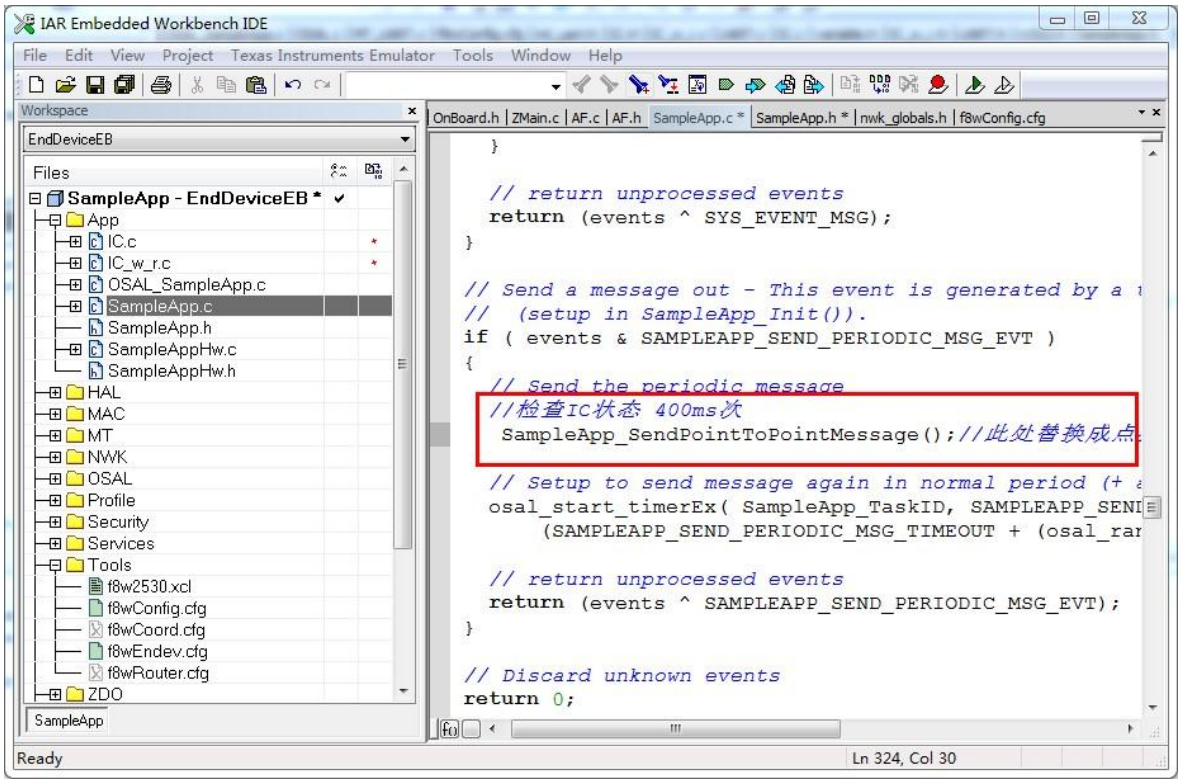


图 4.24

7） 如果检测到 IC 卡，则将卡号数据打包发送出去。

```
void SampleApp_SendPointToPointMessage( void )
{
  /*16 进制转 ASCII 码表-网蜂*/
uint8
asc_16[16]={'0','1','2','3','4','5','6','7','8','9','A','B','C
','D','E','F'},i;
  uint8 Card_Id[8]; //存放 32 位卡号

  if(IC_Test()==1) //读卡成功
  {
    /****16 进制转 ASC 码 串口打印 IC 卡号码********/
      for(i=0;i<4;i++)
```

```
        {
          Card_Id[i*2]=asc_16[qq[i]/16];
          Card_Id[i*2+1]=asc_16[qq[i]%16];
        }
         HalUARTWrite(0,"The Card ID is: ",16);
         HalUARTWrite(0,Card_Id,8);
         HalUARTWrite(0,"\n",1);                    // 回车换行

         HalLcdWriteString( "I GOT IC CARD !", HAL_LCD_LINE_4 );
        //液晶显示

         if ( AF_DataRequest( &Point_To_Point_DstAddr,
                              &SampleApp_epDesc,
                              SAMPLEAPP_POINT_TO_POINT_CLUSTERID,
                              4,
                              qq,
                              &SampleApp_TransID,
                              AF_DISCV_ROUTE,
                              AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
    {
    }
    else
    {
      // Error occurred in request to send.
    }
   }
  }
```

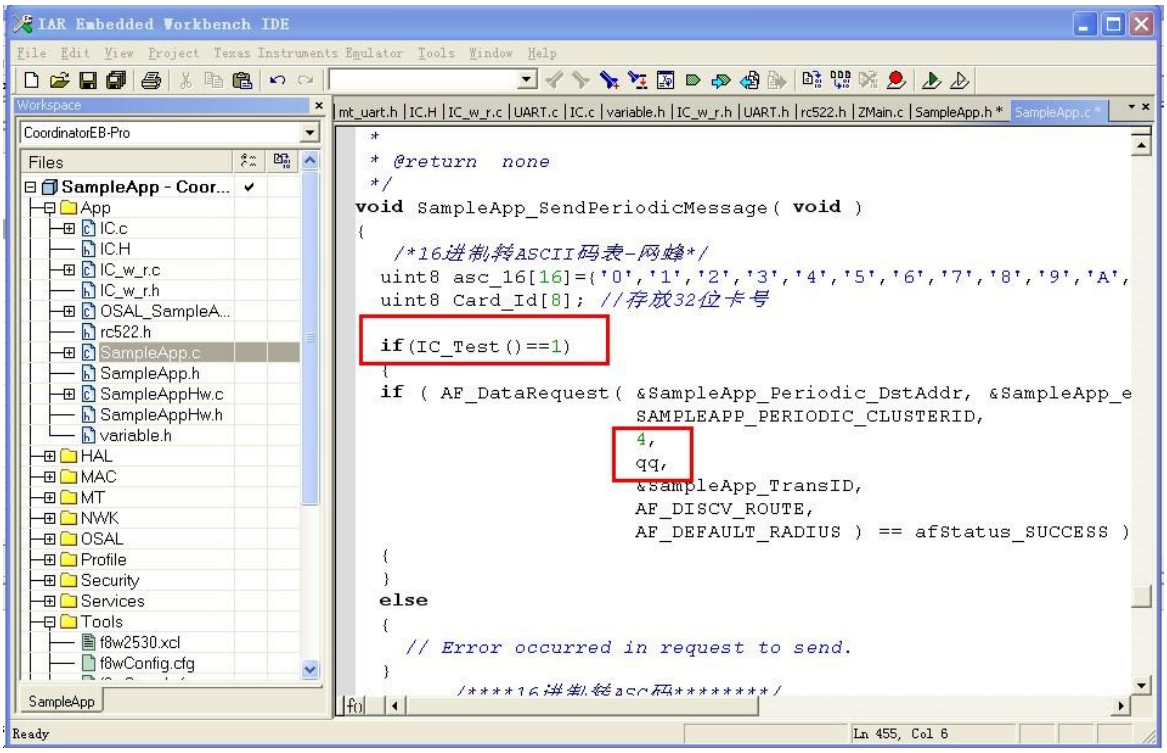


图 4.25

8） 协调器收到 IC 卡号信息后，进行处理，并通过串口打印出来。函数代码如下：

```
uint16 flashTime;
/*16 进制转 ASCII 码表-网蜂*/
uint8  asc_16[16]={'0','1','2','3','4','5','6','7','8','9',
                   'A','B','C','D','E','F'},i;
uint8 CARD_ID[8];
switch ( pkt->clusterId )
{
    case SAMPLEAPP_POINT_TO_POINT_CLUSTERID:
    /****16 进制转 ASC 码********/
      for(i=0;i<4;i++)
      {
        CARD_ID[i*2]=asc_16[(pkt->cmd.Data[i])/16];
        CARD_ID[i*2+1]=asc_16[(pkt->cmd.Data[i])%16];
```

```
}
/*串口打印*/
HalUARTWrite(0,"The Card ID is: ",16);
HalUARTWrite(0,CARD_ID,8);
HalUARTWrite(0,"\n",1);                    // 回车换行
```

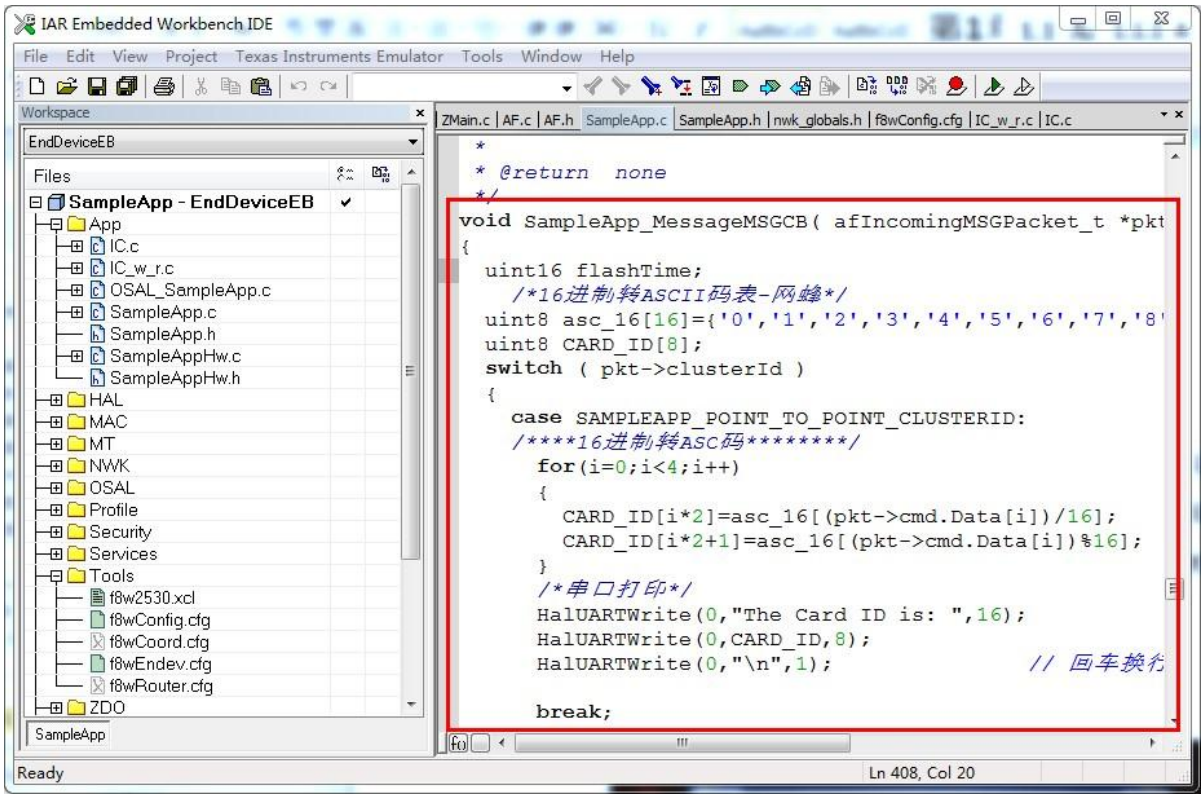


图 4.26

至此，IC 卡程序往协议栈移植完成。

实验现象：

将程序以终端方式下载到带 IC 卡模块的 ZigBee 设备节点，以协调器方式下载到普通的 ZigBee 节点。并通过 USB 线连接到电脑。打开串口调试助手。上电运行，当 IC 卡靠近模块时，我们看到 LCD 显示找到 IC 卡，同时串口助手将卡号打印出来。

LCD12864 显示已经找到 IC 卡：

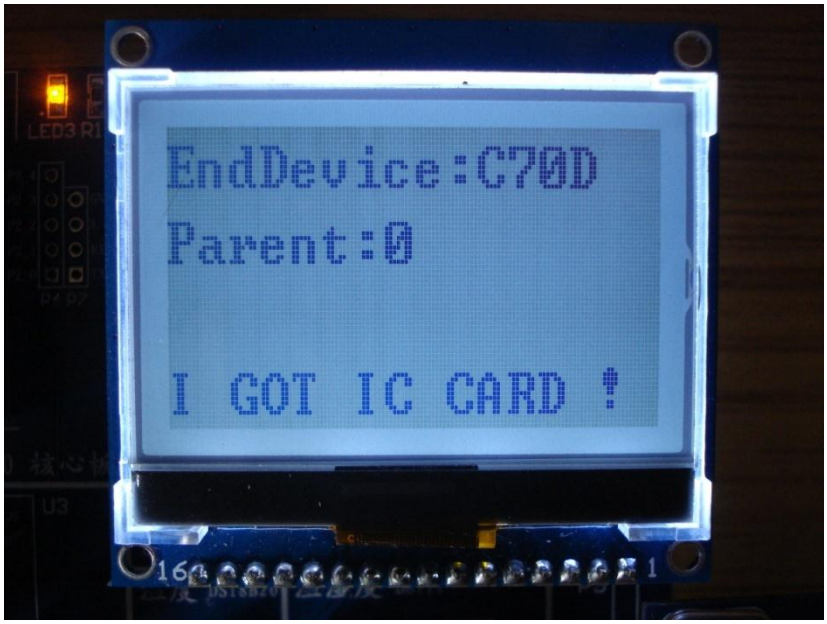


图 4.27

协调器将 IC 卡号打印出来：

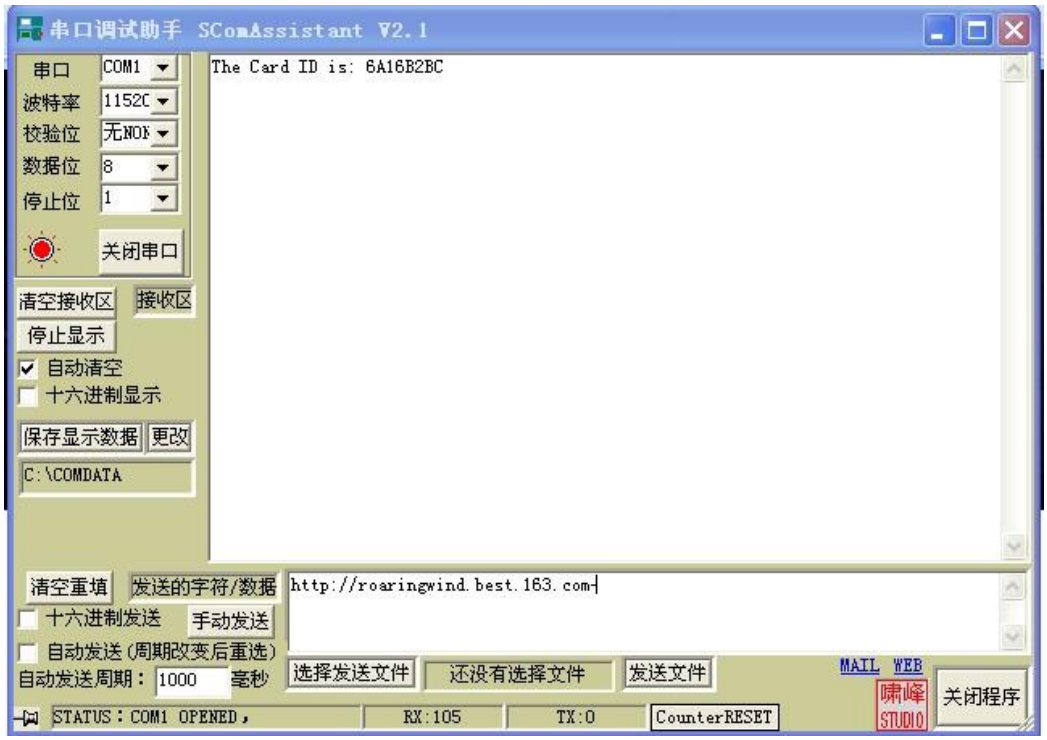


图 4.28

结语：

在项目中我们完成了对 IC 卡的防碰撞和卡号读取，我们可以将自己的 IC 卡号先读出来，然后预先存储，建立自己的小数据库。以后当检测到卡号，经过匹配处理后就可以判别是否预设卡号。适用于考勤、门禁等领域。同时网蜂

建议有兴趣的蜂迷进一步研究 IC 卡的读/写数据操作。实验数据记录、电子钱包等功能。

4.3 串口通讯助手==Zigbee 聊天助手

前言： ZigBee 大家也玩了一段时间了，今天我们拿我们手中的节点来弄个聊天工具，大家只要连接到自己电脑的串口，打开串口调试助手，设置好串口号和波特率就能实验多台电脑之前串口聊天啦，配上专用的串口助手软件，我们还可以说中文呢。是不是很有意思？

实验平台：网蜂 ZigBee 节点设备。

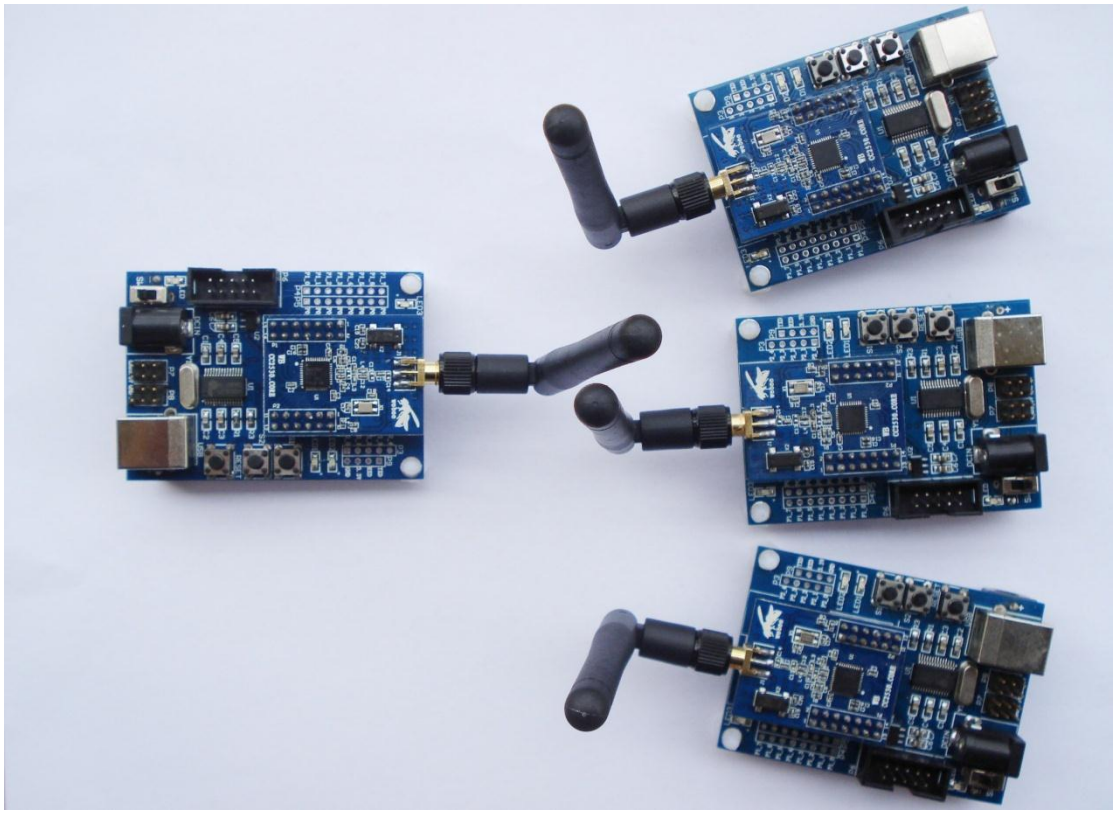


图 4.29 网蜂 ZigBee 节点

实验功能：细心的读者会发现，本项目是基于《ZigBee 实战演练》串口透传章节内容来完成。我们数据传输使用广播方式，单独的透传实现的是数据的直接传递，像 QQ 一样，我们还希望在信息前面加上自己的名字等信息。

实验过程：分二个步骤，如下：

一：以广播的方式实现串口透传功能。

二：给每个需要加入聊天的设备配上特定的名字作为前缀。

一：以广播的方式实现串口透传功能。

这是我们串口透传章节部分的内容。我们打开软件，依次往 3 个节点里下载协调器、路由器、路由器的程序。整个网络 1 个协调器带多个路由器的形式通讯。

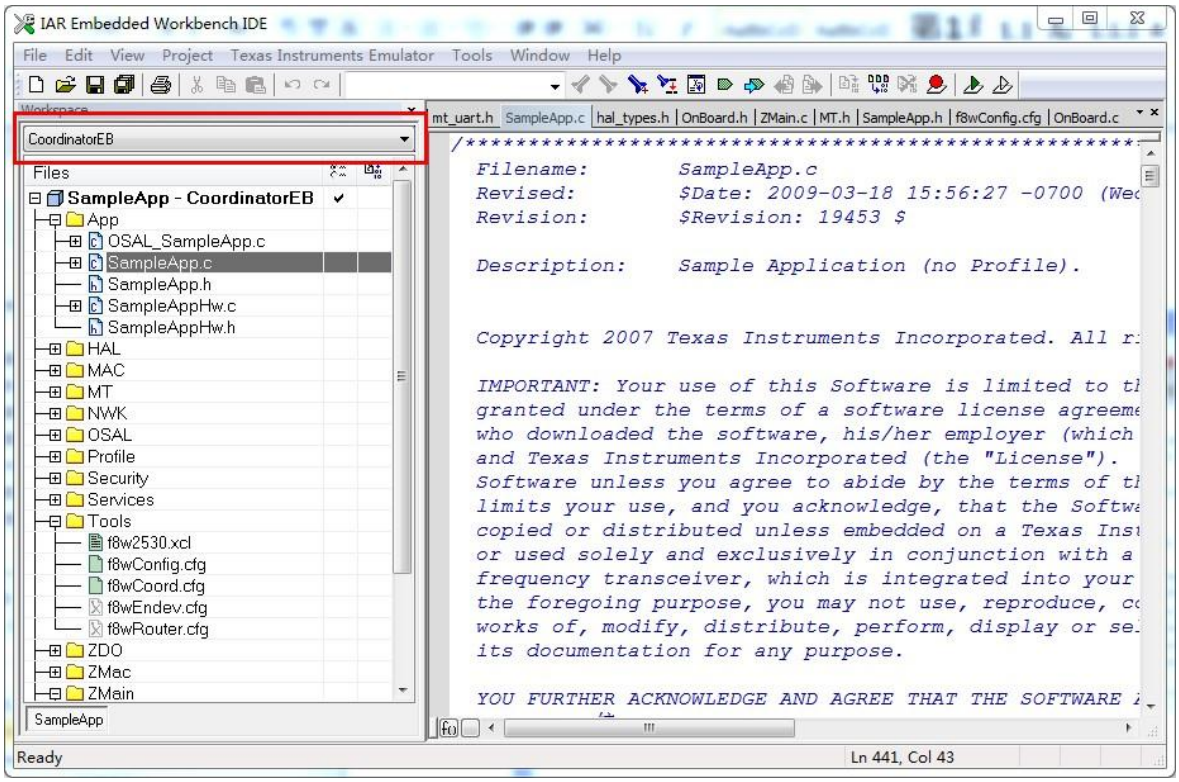


图 4.30 配置 1 个协调器

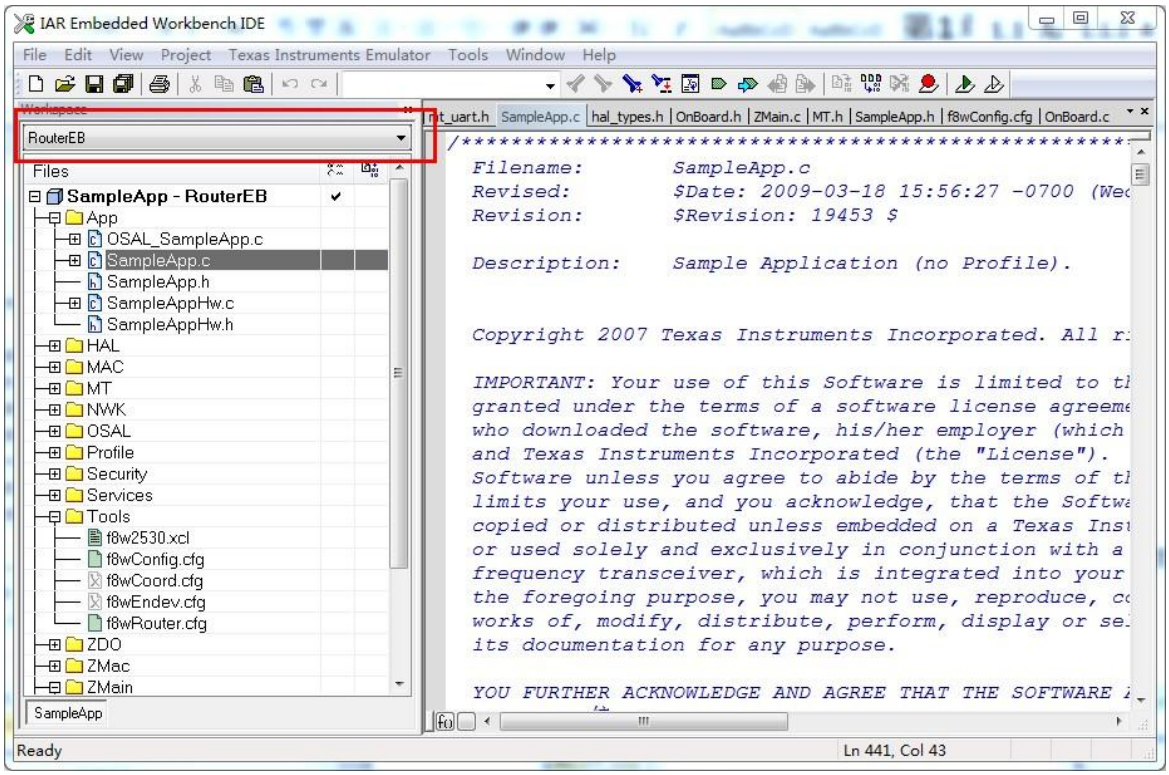


图 4.31 配置 2 个路由器

测试所有设备是否正常工作

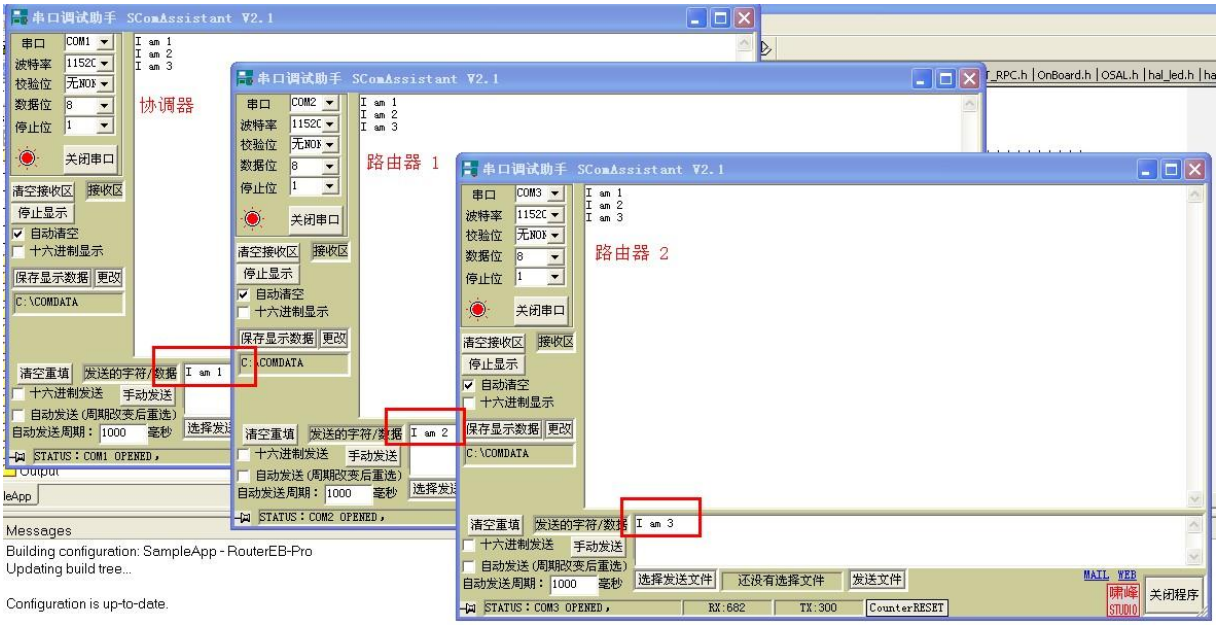


图 4.32 透传设备正常工作

二：给每个需要加入聊天的设备配上特定的名字作为前缀。

我们分别给 3 个设备配上 Jack 、Mike 、Rose 的名字。主要是在串口信息发送前面的地方。

图 4.33 加入人物前缀 Jack

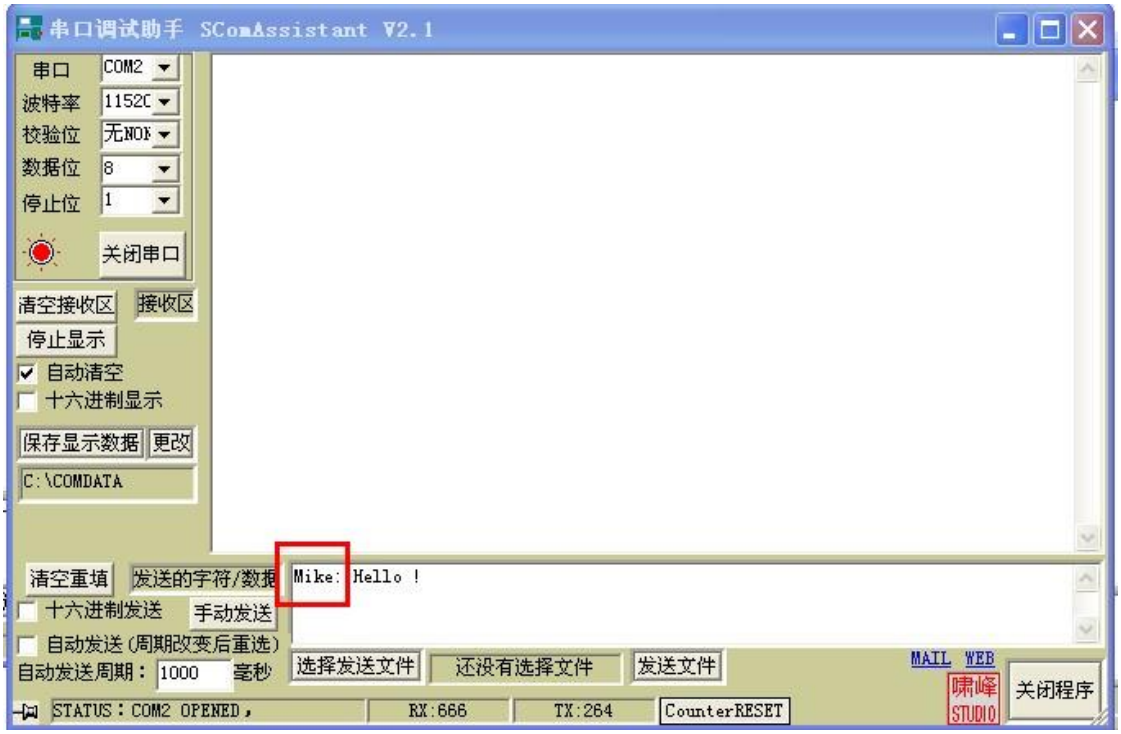


图 4.34 加入人物前缀 Mike

图 4.35 加入人物前缀 Rose

加入人物前缀后，我们发现消息的发送有了区别，显得更生动了。

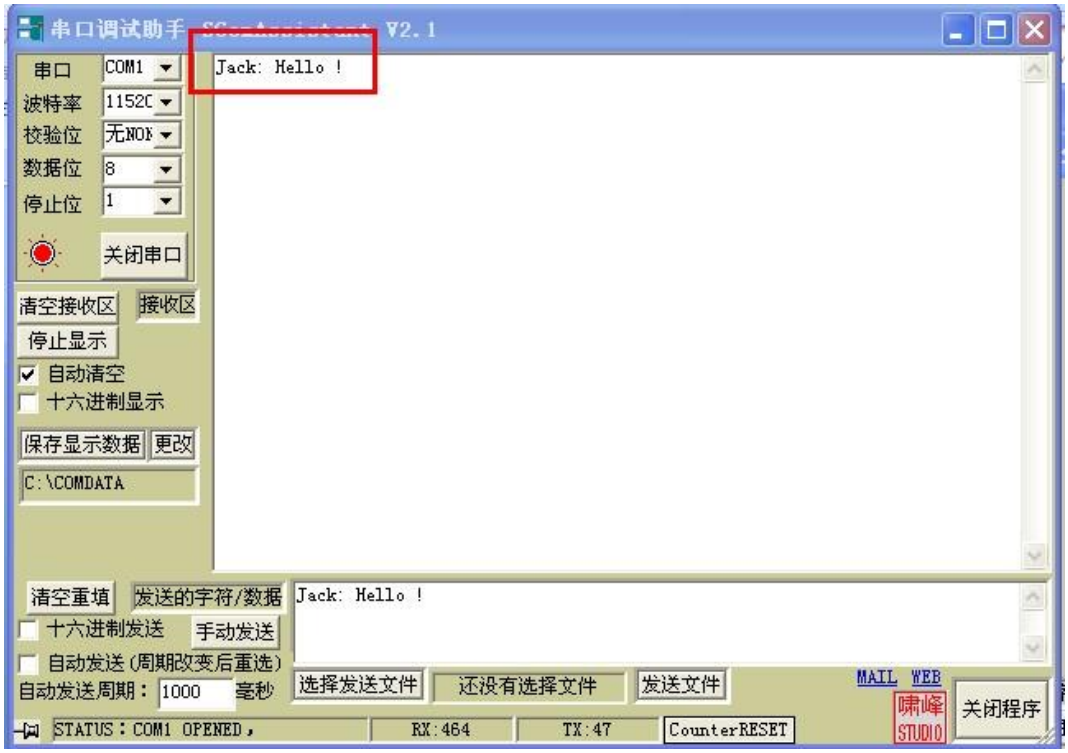


图 4.36 用串口助手测试

实验现象：接下来就是见证神奇的时刻了。我们把下载好程序的 3 个设备连接到 3 台 PC 机，分别打开三台电闹的串口助手。设置波特率 115200bps. 然后就开始聊天了。

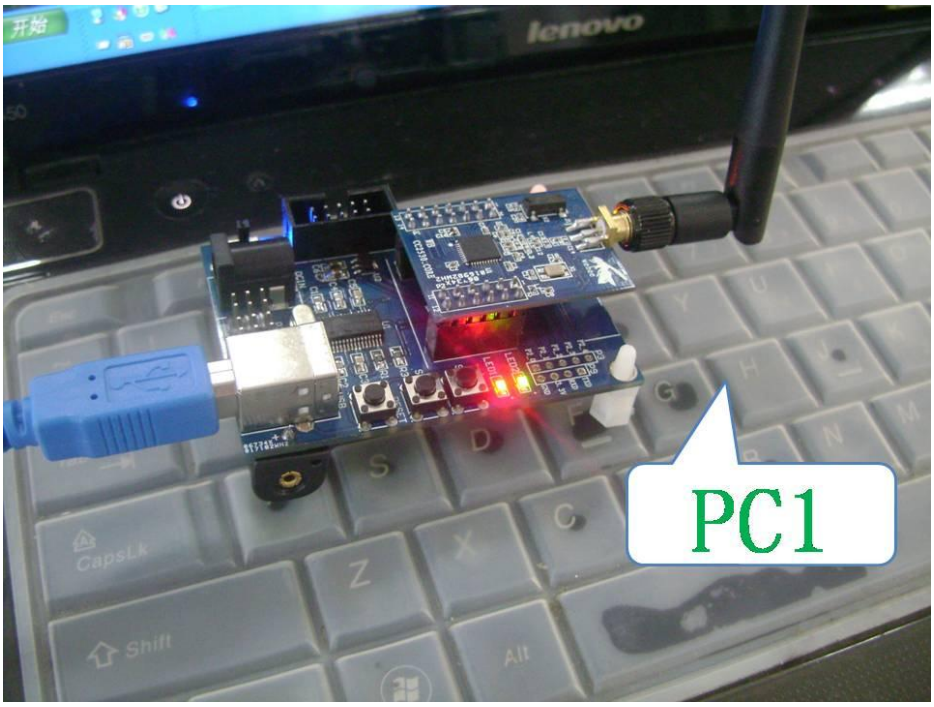


图 4.37 PC1

图 4.38 PC2

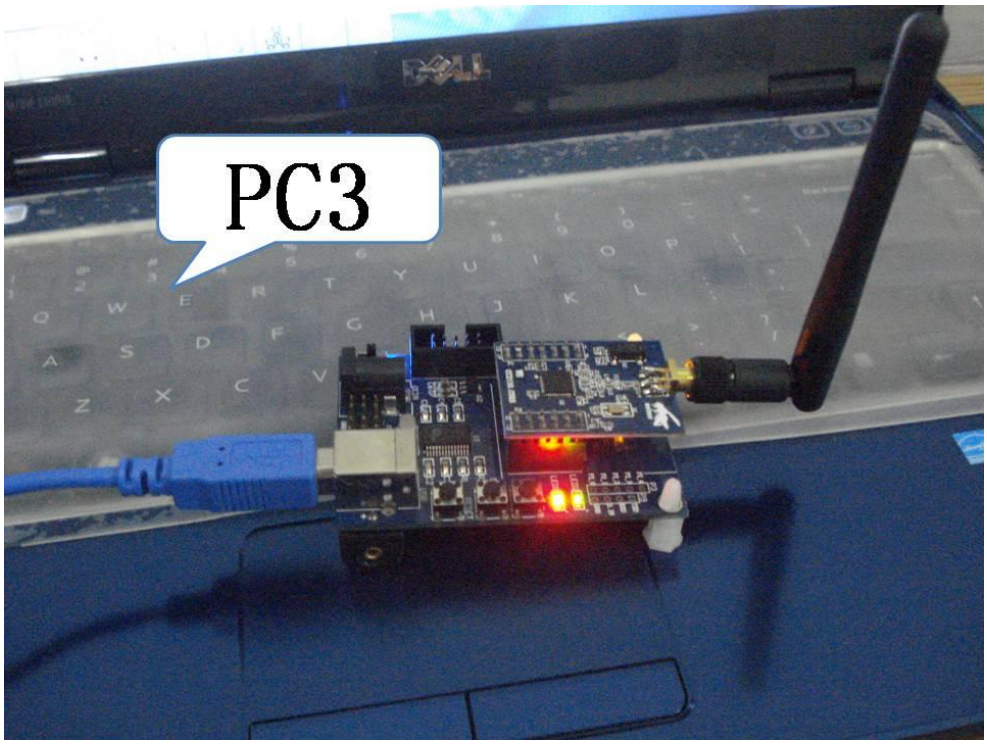


图 4.39　PC3

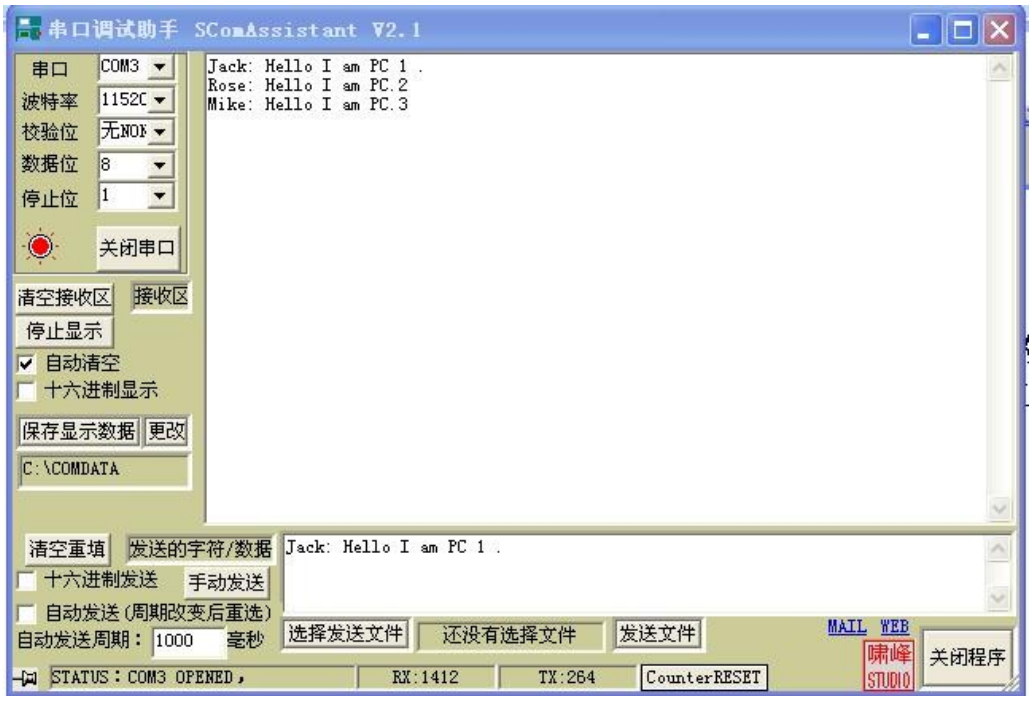


图 4.40　实现三方聊天

结语：项目使用广播的方式，所以我们可以加入多个设备，实验多台 PC 机同时聊天，在室内组建自己的 ZigBee 无线网络，那是非常酷的事。至此，

网蜂 ZigBee 串口聊天助手项目完成，希望大家能进一步拓宽思维，能用 ZigBee 实现更精彩的东西。

4.4 无线互联:ZigBee+GPRS （文档编写中）

4.5 室内定位系统 （开源项目）

4.6 家电控制无线传输协议 （开源项目）

网蜂科技

物联网实战演练系列丛书