揭阳职业技术学院

JIEYANG POLYTECHNIC

无线网络应用教案

主讲：钱德明

物联网教研室

二〇二五年九月

目录

第一章 ZigBee 无线网络技术 1

1.1. 物联网与 WSN 定义 1
1.2. ZigBee 起源 1
1.3. 什么是 ZigBee？ 2
1.4. ZigBee 无线网络特点 2
1.5. ZigBee 协议框架 3
1.6. ZigBee 版本演进 7
1.7. ZigBee 联盟 8
1.8. ZigBee 基本概念 8
1.8.1. ZigBee 信道 8
1.8.2. ZigBee 的 PAN ID(网络号) 9
1.8.3. ZigBee 物理地址 9
1.8.4. ZigBee 设备类型 10
1.8.5. ZigBee 网络地址分配 11
1.8.6. 路由参数 13
1.9. ZigBee 技术应用 13
1.10. 课程思政 14

第二章 ZigBee 无线网络构架 15

2.1. 无线传感器网络构架 16
2.2. 无线传感器网络工作流程 17
2.3. 无线传感器网络实验设备 18
2.3.1. 物联网综合开发网关 19
2.3.2. ZigBee 协调器 19
2.3.3. 无线传感节点 22

第三章 IAR 开发环境的搭建 25

3.1. 相关软件安装 25
3.1.1. 第一步：安装 IAR 8.10 方法 25
3.1.2. 第二步：TI 协议栈 Zstack-CC2530-2.3.0-1.4.0 安装方法 28
3.1.3. 第三步：ZigBee 仿真器驱动安装方法 29
3.2. 工程文件的快速建立 31
3.3. 下载程序 38

第四章 ZigBee 基础实验 40

4.1. CC2530 芯片架构 40
4.2. 实验一：点亮第一个 LED 40
4.3. 实验二：按键 43
4.4. 实验三：外部中断 49
4.5. 实验四：定时器（T1 查询方式） 54
4.6. 实验五：串口通讯 59
4.7. 实验六：睡眠唤醒（中断方式唤醒） 68
4.8. 实验七：ADC 控制（自带温度计） 74
4.9. 实验八：看门狗 81

4.10. 实验九：SensorDemo 实验演示—ZigBee 组网初接触 85

第五章 ZigBee 组网实验 93

5.1. Zigbee 协议栈简介 93

5.2. 实验一：无线组网点灯实验 96

5.3. 实验二：协议栈工作原理 109

5.4. 实验三：协议栈中的串口实验 124

5.5. 实验四：协议栈中的按键实验 137

5.6. 实验五：串口数据透传实验 145

5.7. 实验六：网络通讯实验（单播、组播、广播） 160

5.8. 实验七：Zigbee 协议栈网络管理 180

第六章 传感器和执行器应用 187

1.1. 实验一：温湿度度传感器 187

1.2. 实验二：光敏传感器 203

1.3. 实验三：烟雾传感器 210

1.4. 实验四：人体红外热释电传感器 221

1.5. 实验五：三轴加速传感器 232

1.6. 实验六：磁控传感器 241

1.7. 实验七：直流电机模块 251

1.8. 实验八：继电器控制 263

1.9. 实验九：步进电机模块 275

1.10. 实验十：雨滴检测模块 286

1.11. 实验十一：超声波模块 294

1.12. 实验十二：风速传感器 298

1.13. 实验十三：风向传感器 303

第七章 Wi-Fi 无线网络技术应用 309

7.1. Wi-Fi 技术概述 309

7.1.1. Wi-Fi 的定义 309

7.1.2. Wi-Fi 的发展背景 309

7.2. Wi-Fi 技术特点 309

7.3. Wi-Fi 无线网络技术应用 310

7.4. 7.4 Wi-Fi 应用案例分析 310

7.5. Wi-Fi AT 指令与模块配置 310

7.6. Wi-Fi 硬件搭建与烧写 310

7.7. Wi-Fi 接入云平台操作 310

7.8. 课程思政案例 310

7.8.1. 背景介绍 311

7.8.2. 案例内容 311

7.8.3. 案例分析 311

7.8.4. 思政教育目标 311

7.9. 本章小结 312

第八章 NB-IoT 无线网络技术应用 312

8.1. NB-IoT 技术概述 312

8.2. NB-IoT 技术特点 314
8.3. NB-IoT 关键技术 314
8.4. NB-IoT 无线网络技术应用 315
8.5. NB-IoT 应用案例分析 316
8.6. NB-IoT AT 指令与模块配置 318
8.7. NB-IoT 硬件搭建与烧写 318
8.8. NB-IoT 接入云平台操作 320
8.9. 课程思政案例 321
8.9.1. 背景介绍 321
8.9.2. 案例内容 321
8.9.3. 案例分析 322
8.9.4. 思政教育目标 322
8.10. 本章小结 322
第九章 LoRa 无线网络技术应用 322
9.1. LoRa 技术概述 323
9.1.1. LoRa 的定义 323
9.1.2. LoRa 的发展背景 323
9.2. LoRa 技术特点 323
9.3. LoRa 无线网络技术应用 324
9.4. LoRa 应用案例分析 324
9.5. LoRa AT 指令与模块配置 324
9.6. LoRa 硬件搭建与烧写 324
9.7. LoRa 接入云平台操作 324
9.8. LoRa 无线网络技术应用实践 325
9.8.1. LoRa 温湿度传感器节点开发 325
9.8.2. LoRa 光照传感器节点开发 325
9.8.3. 网关节点汇聚传感器数据 325
9.8.4. LoRa 数据上云 325
9.9. 课程思政案例 325
9.9.1. 背景介绍 325
9.9.2. 案例内容 325
9.9.3. 案例分析 326
9.9.4. 思政教育目标 326
9.10. 本章小结 326
第十章 综合项目——ZigBee 开发项目使用教程 327

第一章 ZigBee 无线网络技术

（一）本章教学目标

主要内容：WSN 的发展历程、基本概念、体系结构、应用前景、研究现状、技术标准

目的与要求：掌握 WSN 的基本概念及体系结构，了解 WSN 应用实例及 WSN 应用类型，了解无线传感器网络的发展前景，掌握传感器网络的特点。

（二）本章重点难点

掌握 WSN 的基本概念及体系结构

1.1.物联网与 WSN 定义

1、物联网的定义：通过射频识别（RFID）、红外感应器、全球定位系统、激光扫描器等信息传感设备，按约定的协议，把任何物体与互联网相连接，进行信息交换和通信，以实现对物体的智能化识别、定位、跟踪、监控和管理的一种网络。

2、无线传感网络的定义：大规模，无线、自组织、多跳、无分区、无基础设施支持的网络．其中的节点是同构的、成本较低、体积较小，大部分节点不移动，被随意撒布在工作区域，要求网络系统有尽可能长的工作时间。 在通信方式上，虽然可以采用有线、无线、红外和光等多种形式，但一般认为短距离的无线低功率通信技术最适合传感器网络使用，为明确起见，一般称无线传感器网络(WSN 即 Wireless Sensor Network)。

1.2.ZigBee 起源

在蓝牙技术的使用过程中，人们发现蓝牙技术尽管有许多优点，但仍存在许多缺陷。对工业，家庭自动化控制和工业遥测遥控领域而言，蓝牙技术显得太复杂，功耗大，距离近，组网规模太小等，……而工业自动化，对无线数据通信的需求越来越强烈，而且，对于工业现场，这种无线数据传输必需是高可靠的，并能抵抗工业现场的各种电磁干扰。因此，经过人们长期努力，ZigBee 协议在 2003 年中正式问世了。另外，ZigBee 使用了在它之前所研究过的面向家庭网络的通信协议 Home RF Lite。

1.3.什么是ZigBee?

ZigBee是IEEE802.15.4协议的代名词。根据这个协议规定的技术是一种近距离、低复杂度、低功耗、低数据速率、低成本的双向无线通信技术，主要适合于自动控制和远程控制领域，可以嵌入各种设备中，同时支持地理定位功能。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和远近信息的，也就是所蜜蜂依靠着这样的方式构成了群体中的通信“网络”，因此ZigBee的发明者们形象地利用蜜蜂的这种行为来形象地描述这种无线信息传输技术。

1.4.ZigBee无线网络特点

与其它无线通信协议相比，ZigBee无线协议复杂性低、对资源要求少，主要有以下特点：

1) 低功耗：在低耗电待机模式下，2节5号干电池可支持1个节点工作6-24个月，甚至更长。这是ZigBee的突出优势。相比之下蓝牙可以工作数周、WiFi可以工作数小时；

2) 低成本：通过大幅简化协议是成本很低（不足蓝牙的1/10），降低了对通信控制器的要求，按预测分析，以8051的8位微控制器测算，全功能的主节点需要32KB代码，子功能节点少至4KB代码，而且ZigBee的协议专利免费；

3) 低速率：ZigBee工作在250kbps的通讯速率，满足低速率传输数据的应用需求；

4) 近距离：传输范围一般介于10～100m之间，在增加RF发射功率后，亦可增加到1-3km。这指的是相邻节点间的距离。如果通过路由和节点间通信的接力，传输距离将可以更远；

5) 短时延：ZigBee的响应速度较快，一般从睡眠转入工作状态只需15ms，节点连接进入网络只需30ms，进一步节省了电能。相比较，蓝牙需要3-10s、WiFi需要3s；

6) 高容量：ZigBee可采用星状、片状和网状网络结构，由一个主节点管理若干子节点，最多一个主节点可管理254个子节点；同时主节点还可由上一层网络节点管理，最多可组成65000个节点的大网；

7) 高安全：ZigBee提供了三级安全模式，包括无安全设定、使用接入控制清单

(ACL)防止非法获取数据以及采用高级加密标准(AES128)的对称密码，以灵活确定其安全属性；

8) 免执照频段：采用直接序列扩频在工业科学医疗 2.4GHz(全球)频段。

1.5. ZigBee 协议框架

ZigBee 协议栈由一组子层构成。每层为其上层提供一组特定的服务：一个数据实体提供数据传输服务，一个管理实体提供全部其他服务。每个服务实体通过一个服务接入点(SAP)为其上层提供服务接口，并且每个 SAP 提供了一系列的基本服务指令来完成相应的功能。

ZigBee 协议栈的体系结构如图 1 所示。它虽然是基于标准的七层开放式系统互联(OSI)模型，但仅对那些涉及 ZigBee 的层予以定义。IEEE802.15.4-2003 标准定义了最下面的两层：物理层(PHY)和介质接入控制子层(MAC)。ZigBee 联盟提供了网络层和应用层(APL)框架的设计。其中应用层的框架包括了应用支持子层(APS)、ZigBee 设备对象(ZDO)和由制造商制订的应用对象。

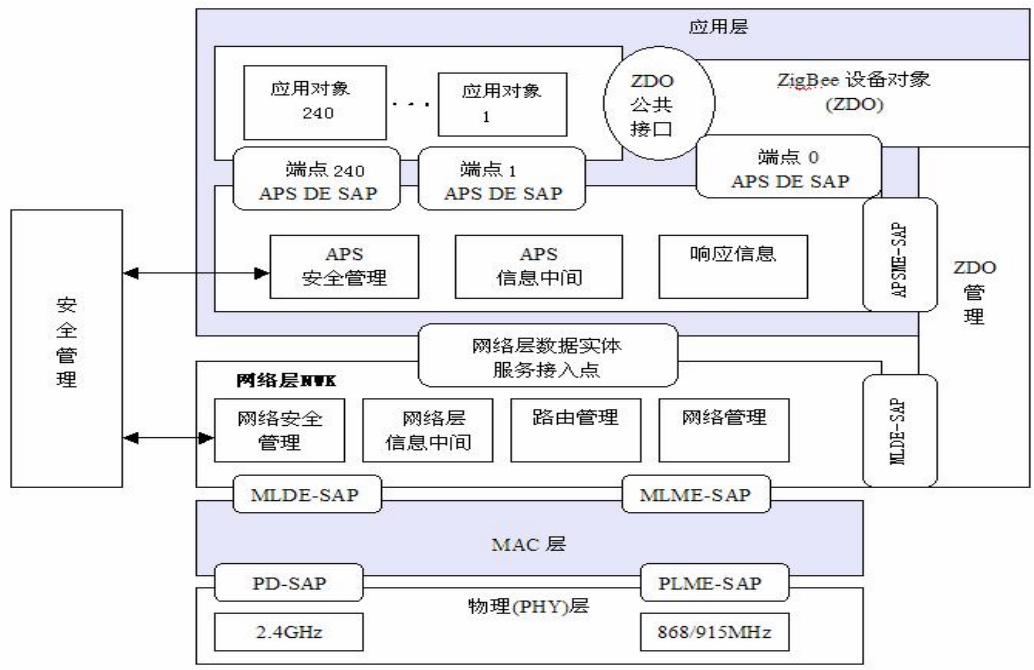


图 1 Zigbee 体系结构模型

相比于常见的无线通信标准，ZigBee 协议套件紧凑而简单，具体实现的要求很低。以下是 ZigBee 协议套件的最低需求估计：硬件需 8 位处理器，如 80C51；软件需要 32kB 的 ROM，最小软件需要 4kB 的 ROM，如 CC2530 芯片是具有 8051 内核的内存从 32KB 至 256KB

的 ZigBee 无线单片机；网络主节点需要更多的 RAM 以容纳网络内所有节点的设备信息、数据包转发表、设备关联表、与安全有关的密钥存储等。

ZigBee 联盟希望建立一种可连接每个电子设备的无线网。它预言 ZigBee 将很快成为全球高端的无线技术，到 2007 年 ZigBee 节点将达到 30 亿个。具有几十亿个节点的网络将很快耗尽已不足的 IPv4 的地址空间。因此 IPv6 与 IEEE 802.15.4 结合是传感器网络的发展趋势。IPv6 采用 128 位地址长度，几乎可以不受限制地提供地址。按保守方法估算，IPv6 实际可为整个地球的每平方米面积分配 1000 多个地址。IPv6 在设计过程中，除了一劳永逸地解决了地址短缺问题以外，还考虑了在 IPv4 中解决不好的其他问题，如端到端 IP 连接、服务质量(QoS)、安全性、多播、移动性、即插即用等。

IEEE 802.15.4 工作在工业科学医疗(ISM)频段，定义了两个工作频段，即 2.4 GHz 频段和 868/915MHz 频段。在 IEEE 802.15.4 中，总共分配了 27 个具有 3 种速率的信道：在 2.4 GHz 频段有 16 个速率为 250kb/s 的信道，在 915MHz 频段有 10 个 40 kb/s 的信道，在 868 MHz 频段有 1 个 20 kb/s 的信道。

这些信道的中心频率按如下定义(k 为信道数)：

Fc=868.3MHz，(k=0)

Fc=906MHz+2(k-1)MHz，(k=1，2……10)

Fc=2405MHz+5(k-11)MHz，(k=11，12……26)

一个 IEEE802.15.4 可以根据 ISM 频段、可用性、拥挤状况和数据速率在 27 个信道中选择一个工作信道。从能量和成本效率来看，不同的数据速率能为不同的应用提供较好的选择。例如，对于有些计算机外围设备与互动式玩具，可能需要 250kb/s 速率，而对于其他许多应用，如各种传感器、智能标记和家用电器等，20kb/s 这样的低速率就能满足要求。

来自 IEEE 802.15.4 物理层协议数据单元(PPDU)的二进制数据被依次(按字节从低到高)组成 4 位二进制数据符号，每种数据符号(对应 16 状态组中的一组)被映射成 32 位伪噪声码片(CHIP)，以便传输。然后这个连续的伪噪音 CHIP 序列被调制(采用最小键控方式)到载波上，即采用半正弦脉冲波形的偏移正交相移键控(OQPSK)调制方式。

IEEE802.15.4 物理层传输格式如图 2.15 所示。868/915 MHz 频段物理层使用简单的直接序列扩频(DSSS)方法，每个 PPDU 数据传输位被最大长为 15 的 CHIP 序列所扩展(即被多组+1、-1 构成的 m-序列编码)，然后使用二进制相移键控技术调制这个扩展的位元序列。不同的数据传输率适用于不同的场合。例如：868/915MHz 频段物理层的低速率换取了较好

的灵敏度和较大的覆盖面积，从而减少了覆盖给定物理区域所需的节点数。2.4 GHz 频段物理层的较高速率适用于较高的数据吞吐量、低延时或低作业周期的场合。

IEEE 802.15.4 MAC 层提供两种服务：MAC 层数据服务和 MAC 层管理服务。管理服务通过 MAC 层管理实体(MLME)服务接入点(SAP)访问高层。MAC 层数据服务使 MAC 层协议数据单元(MPDU)的收发可以通过物理层数据服务。IEEE 802.15.4MAC 层的特征有信标管理、信道接入机制、保证时隙(GTS)管理、帧确认、确认帧传输、节点接入和分离。

ZigBee 的网络层主要用于 ZigBee 网络的组网连接、数据管理以及网络安全等。而应用层主要用于 ZigBee 技术的实际应用提供一些应用框架模型等，以便对 ZigBee 技术的开发应用， 在不同的场合，其开发应用架框不同，从目前来看，不同的厂商提供的应用框架是有差异的，应根据具体应用情况和所选择的产品来综合考虑其应用框架结构。现有比较著名的 ZigBee 芯片提供商包括 CHIPCON(已于 2006 年被 IT 公司收购)公司、FREESCALE 公司、EMBER 公司等等。

低速率的无线个域网允许使用超帧结构。超帧的格式由传感器网络的协调器定义。超帧被分为 16 个大小相等的时隙，由协调器发送，如图 2 所示。每个超帧之间由网络信标分隔。信标帧在超帧的第一个时隙被传输。如果协调器不想使用超帧结构，它将会停止信标的传输。信标可用来使接入的设备同步，区分个域网，描述超帧结构。任何想要在竞争接入时段(CAP)通信的设备都要使用有时隙的载波监听多址接入-冲突避免(CSMA-CA)。所有的传输要在下一个信标到来之前结束。

超帧结构有活跃和非活跃两部分。在非活跃部分，协调器将不和网络联系，进入低能模式。

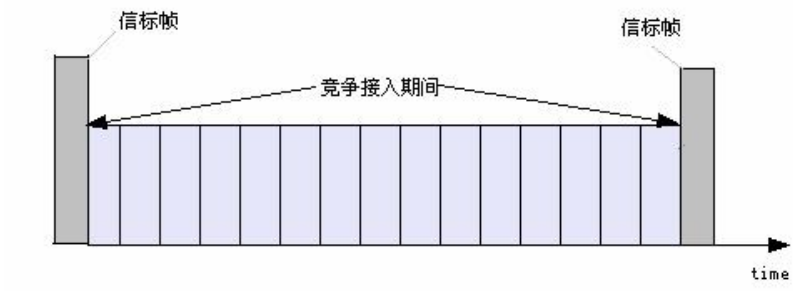


图 2 无 GTS 的超帧结构

对于低延迟应用或需要特殊带宽的应用来说，网络协调器为它贡献出超帧的活跃部分。这部分叫做 GTS。GTS 由无竞争时段(CFP)组成，它总是紧跟着 CAP，在活跃的超帧尾部，

如图 3 所示。网络协调器可以分配 7 个 GTS，每个 GTS 可以占用一个以上的时隙。而 CAP 有充足的时间留给基于竞争的接入的网络设备或想加入网络的设备。所有基于竞争的传输都要在 CFP 开始前结束，同样，GTS 的传输也要确保在下 GTS 开始前结束。

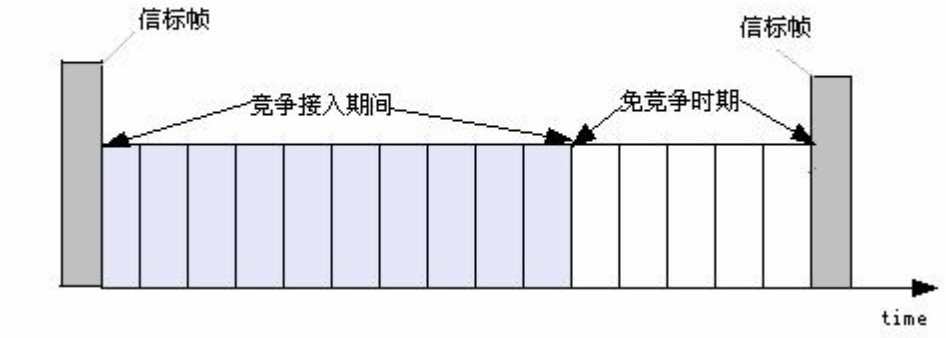


图3 有GTS的超帧结构

ZigBee 联盟希望建立一种可连接每个电子设备的无线网。它预言 ZigBee 将很快成为全球高端的无线技术，到 2007 年 ZigBee 节点将达到 30 亿个。具有几十亿个节点的网络将很快耗尽已不足的 IPv4 的地址空间。因此 IPv6 与 IEEE 802.15.4 结合是传感器网络的发展趋势。IPv6 采用 128 位地址长度，几乎可以不受限制地提供地址。按保守方法估算，IPv6 实际可为整个地球的每平方米面积分配 1000 多个地址。IPv6 在设计过程中，除了一劳永逸地解决了地址短缺问题以外，还考虑了在 IPv4 中解决不好的其他问题，如端到端 IP 连接、服务质量(QoS)、安全性、多播、移动性、即插即用等。

从图 1 可以看到 ZigBee 应用层框架包括应用支持层(APS)、ZigBee 设备对象(ZDO)和制造商所定义的应用对象。

应用支持层的功能包括：维持绑定表、在绑定的设备之间传送消息。所谓绑定就是基于两台设备的服务和需求将它们匹配地连接起来。

ZigBee 设备对象的功能包括：定义设备在网络中的角色(如 ZigBee 协调器和终端设备)，发起和响应绑定请求，在网络设备之间建立安全机制。ZigBee 设备对象还负责发现网络中的设备，并且决定向他们提供何种应用服务。

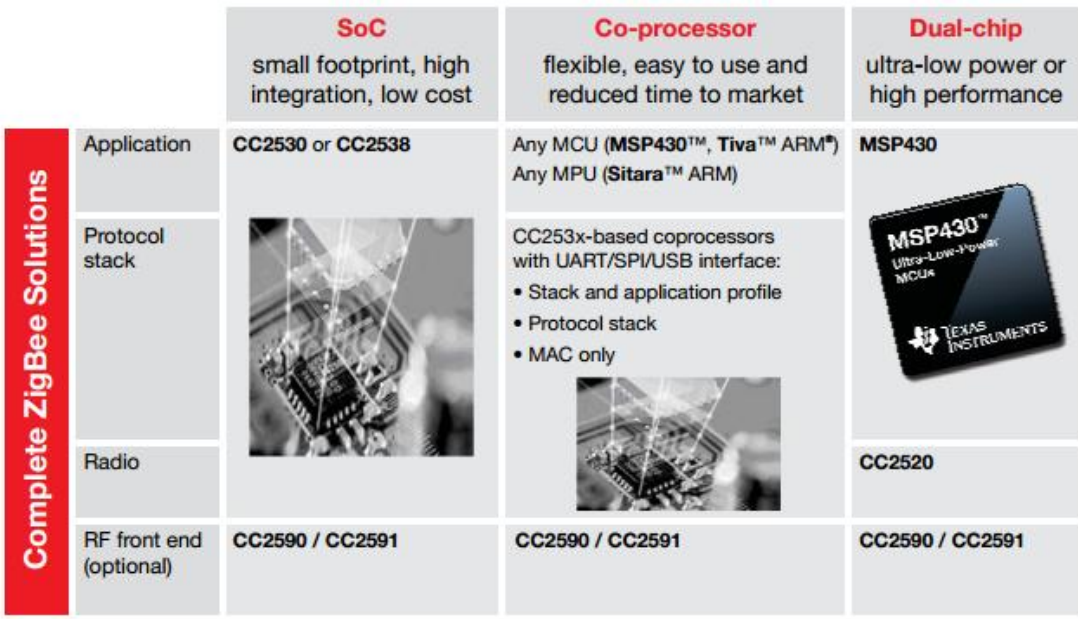
ZigBee 应用层除了提供一些必要函数以及为网络层提供合适的服务接口外，一个重要的功能是应用者可在这层定义自己的应用对象。

1.6. ZigBee 版本演进

ZigBee 是以 IEEE 802.15.4 标准为基础，发展出的无线通讯技术。2000 年 12 月成立了工作小组起草 IEEE 802.15.4 标准，ZigBee 联盟于 2002 年 10 月发起成立，当时的成员有包括 Philips Semiconductor、Honeywell、Mitsubishi、Invensys、Motorola 等，其中 Philips Semiconductor 于 2004 年 4 月退出，改由 Philips Lighting(照明)接替其原有在 ZigBee Alliance 中的会员位置。

2004 年 12 月 ZigBee1.0 标准(又称为 ZigBee2004)敲定，之后于 2005 年 9 月公布并提供下载。于 2006 年 12 月进行标准修订，推出 ZigBee 1.1 版(又称为 ZigBee2006)。ZigBee 1.1 较原有 ZigBee 1.0 作了若干修改，例如新增 ZCL(ZigBee Cluster Library)、群化式装置(Group Device)、多播(Multicast，附 2)功效、直接透过无线方式(Over The Air；OTA)进行组态配置，此外也移除了 KVP(Key Value Pair)的信息格式。

然而 ZigBee1.1 依然无法达到最初的理想，此标准又于 2007 年 10 月完成再次修订(称为 ZigBee2007/PRO 或 ZigBee Pro 或 ZigBee2007)，推出 ZigBee Pro Feature Set(简称：ZigBee Pro)的新标准。此新标准 ZigBee 联盟更专注 3 种应用类型的拓展包括：1.家庭自动化(Home Automation；HA)、2.建筑/商业大楼自动化(Building Automation；BA)、3.先进抄表基础建设(Advanced Meter Infrastructure；AMI)。



1.7. ZigBee 联盟

ZigBee 联盟是一个高速成长的非盈利业界组织，成员包括国际著名半导体生产商、技术提供者、技术集成商以及最终使用者。联盟制定了基于 IEEE802.15.4，具有高可靠、高性价比、低功耗的网络应用规格。

ZigBee 联盟的主要目标是以通过加入无线网络功能，为消费者提供更富有弹性、更容易使用的电子产品。ZigBee 技术能融入各类电子产品，应用范围横跨全球的民用、商用、公共事业以及工业等市场。使得联盟会员可以利用 ZigBee 这个标准化无线网络平台，设计出简单、可靠、便宜又节省电力的各种产品来。

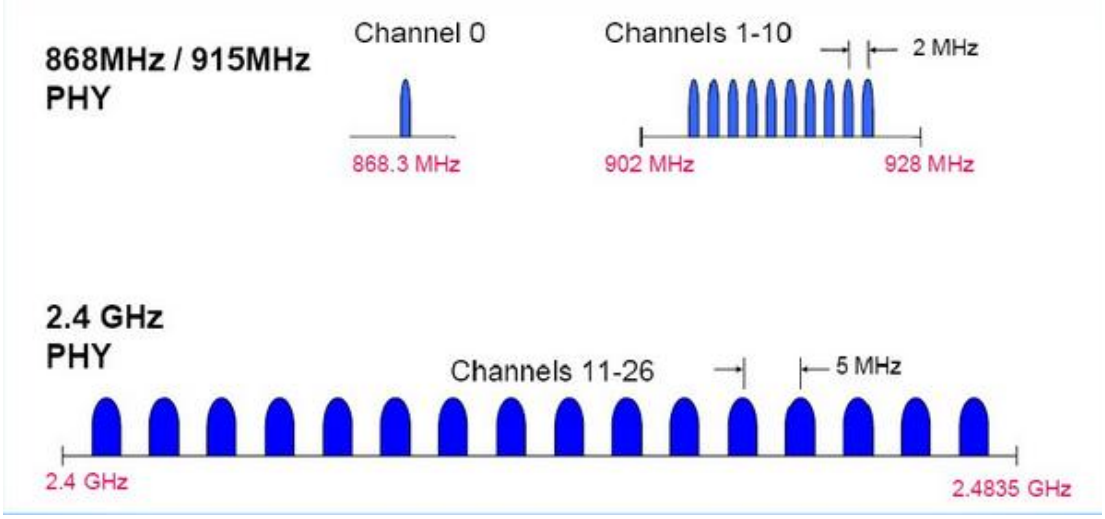
1.8. ZigBee 基本概念

1.8.1. ZigBee 信道

由于 ZigBee 使用的是免执照的工业科学医疗（ISM）频段，所以 ZigBee 使用了 3 个频段，分别为：868MHz（欧洲）、915MHz（美国）、2.4GHz（全球）。

这样，ZigBee 共定义了 27 个物理信道，其中，868MHz 频段定义了一个信道；915MHz 频段附近定义了 10 个信道，信道间隔为 2MHz；2.4GHz 频段定义了 16 个信道，信道间隔为 5MHz。具体信道分配如下表：

信道编号	中心频率	信道间隔	频率上限	频率下限
K=0	868.3	0	868.6	868.0
K=1,2,3…10	906+2(k-1)	2	928.0	902.0
K=11,12,13	2401+5(K-11	5	2483.5	2400.0



其中，理论上，在 868MHz 的物理层，数据传输速率为 20Kb/s；在 915MHz 的物理层，数据传输速率为 40Kb/s；在 2.4GHz 的物理层，数据传输速率为 250Kb/s。实际上，除掉信道竞争应答和重传等消耗，真正能被应用所利用的速率可能不足 100Kb/s，并且余下的速率可能要被临近多个节点和同一个节点的应用瓜分。

1.8.2. ZigBee 的 PAN ID(网络号)

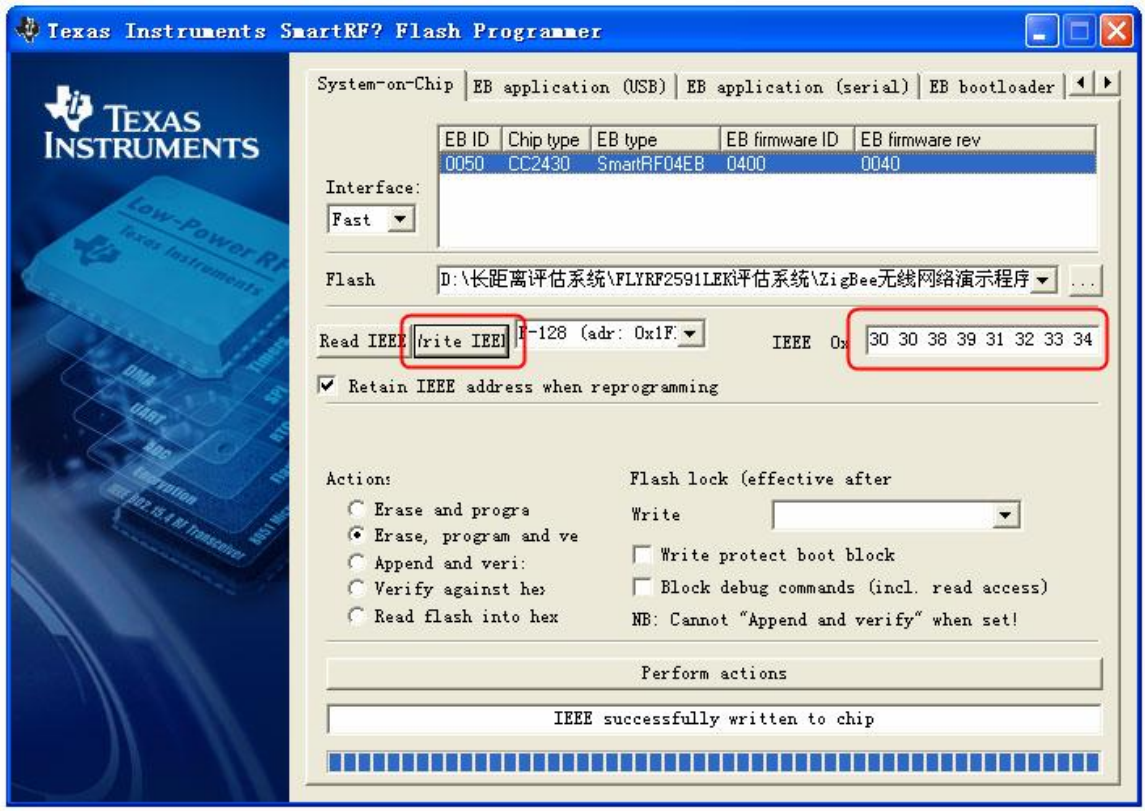
ZigBee 无线传感器网络的协调器是通过选择网络工作信道及个域网识别标志(PANID 或网络号)来启动一个 ZigBee 无线传感器网络。PANID 是一个 32 位标设，范围从 0x0000-0xffff。

1.8.3. ZigBee 物理地址

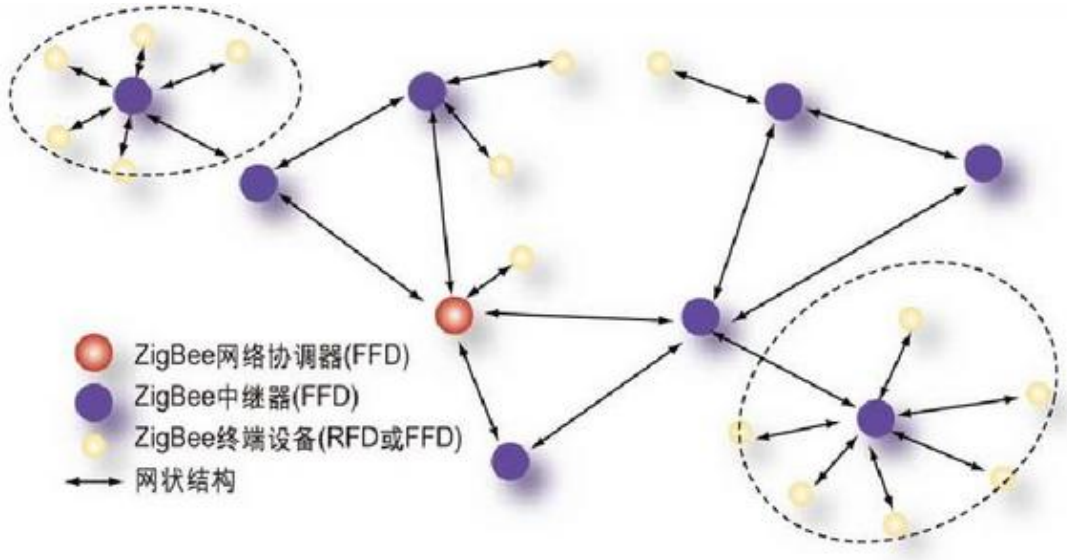
在 ZigBee 无线传感器网络中，节点有 2 个地址，一个是物理(也称 IEEE 或扩展)地址，物理地址是在产品出厂时初始化的，在全球范围内是唯一标识地址。

注：当一个 ZigBee 节点需要加入网络时，其物理地址必须不能与现有网络节点物理地址有冲突，并且不为 0xffffffffffffffff。

用户可以通过配套软件物理地址修改软件进行物理地址读写，如下图所示。



1.8.4. ZigBee 设备类型



ZigBee 规范定义了三种类型的设备，每种都有自己的功能要求：ZigBee 协调器是启动和配置网络的一种设备。协调器可以保持间接寻址用的绑定表格，支持关联，同时还能设计信任中心和执行其它活动。协调器负责网络正常工作以及保持同网络其它设备的通信。**一个 ZigBee 网络只允许有一个 ZigBee 协调器**。

ZigBee 路由器是一种支持关联的设备，能够将消息转发到其它设备。ZigBee 网格或树

型网络可以有多个 ZigBee 路由器。ZigBee 星型网络不支持 ZigBee 路由器。

ZigBee 终端设备可以执行它的相关功能，并使用 ZigBee 网络到达其它需要与其通信的设备，它的存储器容量要求最少，其可以实验 ZigBee 低功耗设计。

上述的三种设备根据功能完整性可分为全功能(FFD)和半功能(RFD)设备。其中全功能设备可作为协调器、路由器和终端设备，而半功能设备只能用于终端设备。一个全功能设备可与多个 RFD 设备或多个其它 FFD 设备通信，而一个半功能设备只能与一个 FFD 通信。

1.8.5. ZigBee 网络地址分配

ZigBee 有两种地址分配方式：分布式分配机制和随机分配机制。

1.随机分配机制

随机分配机制是指当 NIB 的 nwkAddrAlloc 值为 0x02 时，地址随机选择。在这种情况下 nwkMaxRouter 就无意义了。随机地址分配应符合 NIST 测试中的描述。当一个设备加入网络使用的是 Mac 地址，其父设备应选择一个尚未分配过的随机地址。一旦设备已分配一个地址，它没有理由放弃该地址，并应予以保留，除非它收到声明，其地址与另一个设备冲突。此外，设备可能自我指派随机地址，比如利用加入命令帧加入一个网络。

2. 分布式分配机制

我们知道，每个 ZigBee 设备应该拥有一个唯一物理地址。协调器（coordinator）在建立网络以后使用 0x0000 作为自己的短地址。

在路由器（router）和终端(enddevice)加入网络以后，使用父设备给它分配的 16 位的短地址来通讯。那么这些短地址是如何分配的呢？

16 位的地址意味着可以分配给 65536 个节点之多，地址的分配取决于整个网络的架构，整个网络的架构由这 3 个值决定：

1.网络的最大深度(Lm)；

2.每个父亲设备拥有的孩子数(Cm)；

3.第 2 条的孩子设备当中有几个是路由器(Rm)。

有了这 3 个值就可以根据下面的公式来算出某父设备的路由器子设备之间的地址间隔 Cskip(d)：

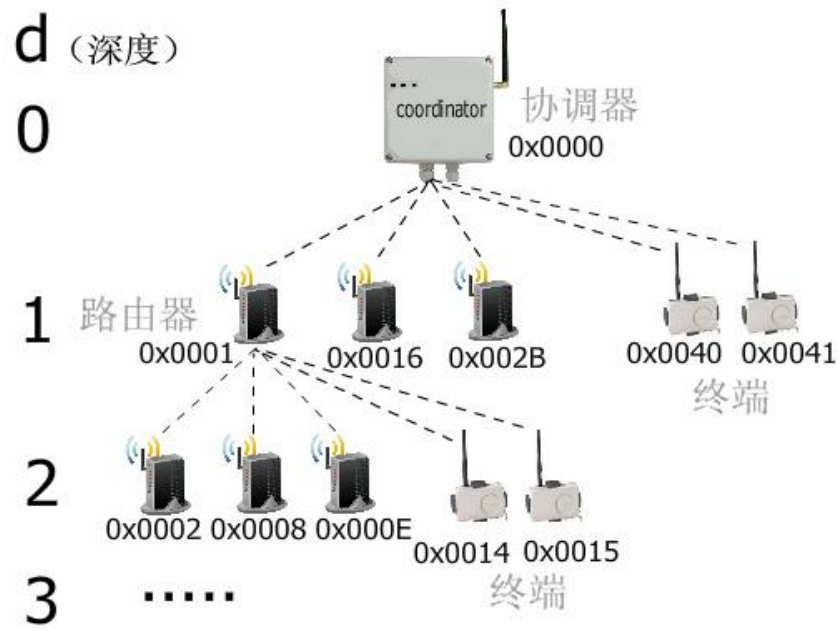
$$Cskip(d) = \begin{cases} 1 + Cm\ (Lm - d - 1), \text{if Rm} = 1 \\ \dfrac{1 + Cm - Rm - Cm * Rm^{Lm-d-1}}{1 - Rm} \end{cases}$$

上面这个公式是用来计算位于深度 d 的父亲设备的，它所分配的子路由器之间的短地

址间隔。该父亲设备分配的第 1 个路由器地址=父亲设备地址+1，分配的第 2 个路由器地址=父亲设备地址+1+Cskip(d)，第 3 个路由器地址=父亲设备地址+1+2×Cskip(d)，依次类推。计算终端地址：

$$A_n = A_{parent} + Cskip(d) * Rm + n$$

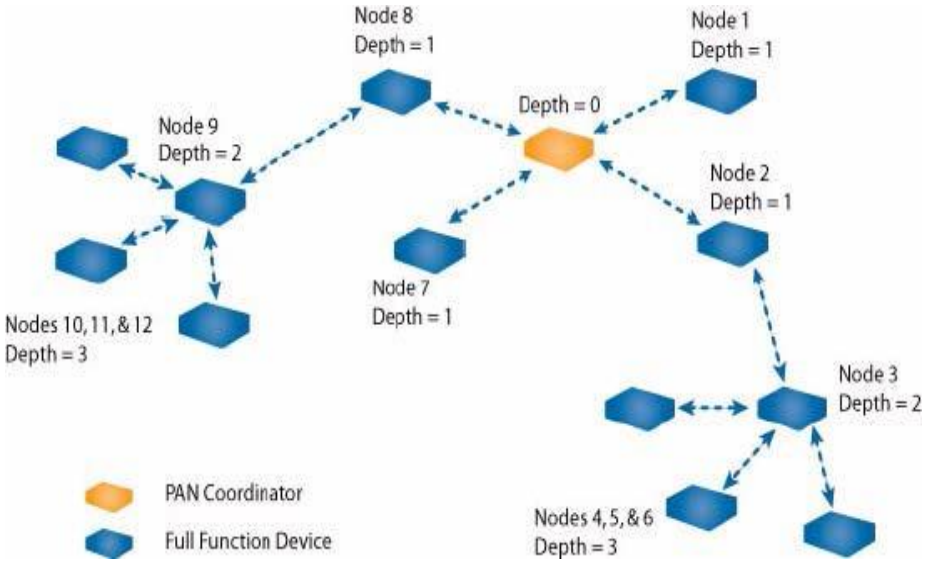
这个公式是来计算 A parent 这个父亲设备分配的第 n 个终端设备的地址 A n。来举个简单的例子，假设有一个 ZigBee 网络，最大深度为 3，每个父亲的最大孩子数是 5，在孩子当中路由器数量是 3，如图所示：



由图可知，协调器 Cskip(d)=(1+5-3-5×3^(3-0-1))/(1-3)=21，所以协调器第一个路由器是 1，第二个就是 22，换算成十六进制就是 0x0016。协调器第 1 个终端地址=0x0000+21×3+1=64=0x0040、第 2 个就是 0x0041。由此可见所有同一父亲终端设备短地址都是连续的。

不难看出一旦 Lm、Cm、Rm 这 3 个值确定了，整个网络设备地址也就确定下来。所以知道了某个设备短地址就可以计算出它设备类型和它的父设备地址。

1.8.6. 路由参数



ZigBee 中设备最大数量由网络允许情况决定，ZigBee 决定最大数量的路由器，最大数量的终端节点。一个 ZigBee 无线网络必须至少包括 1 个协调器。

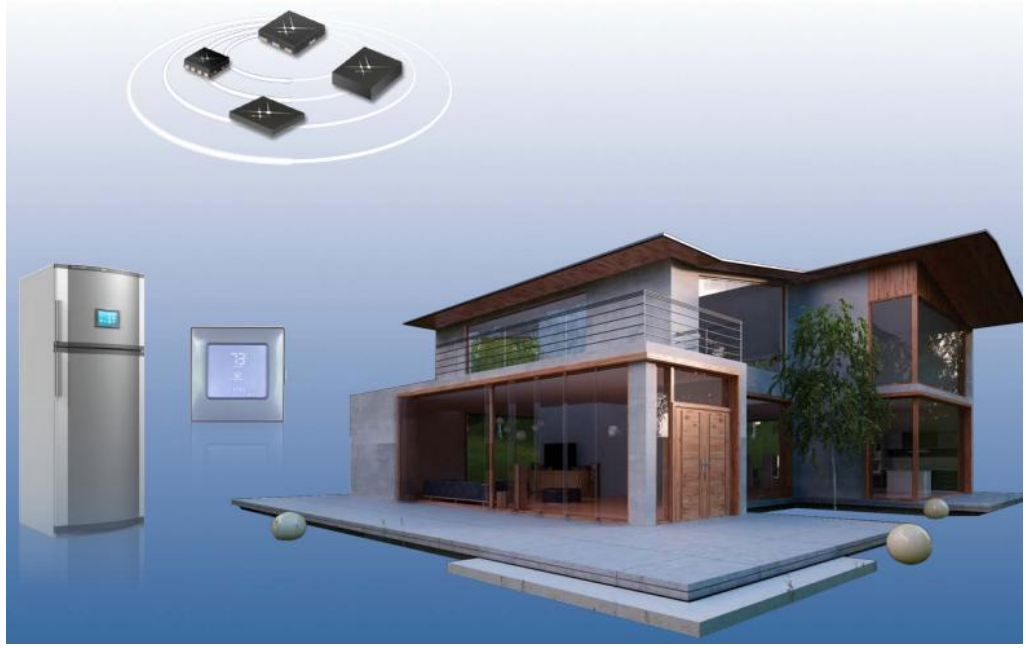
协调器是网络的发起者，他的网络深度为 0。协调器的子节点网络深度为 1，再向下一级设备网络深度增加 1。网络最大负载量由网络最大深度与每一个路由器允许的最大子设备数量决定。

例如:上图中节点 8(Node8)网络深度(Depth)为 1，节点 9 网络深度为 2，节点 3 网络深度也为 2。

最大数量的子节点数是指允许连接到父节点设备的最大的设备数量。

1.9. ZigBee 技术应用

ZigBee 应用范围非常广泛，可以针对工业自动化、家庭自动化、遥测遥控、汽车自动化、农业自动化和医疗护理、油田、电力、矿山和物流管理等应用领域。实际应用举例如下：照明控制、环境控制、自动读表系统、各类窗帘控制、烟雾传感器、医疗监控系统、大型空调系统、内置家居控制的机顶盒及万能遥控器、暖气控制、家庭安防、工业和楼宇自动化。另外它还可以对局部区域内移动目标例如城市中的车辆进行定位。



通常，符合如下条件之一的短距离通信就可以考虑应用ZigBee：

（1）需要数据采集或监控的网点多；

（2）要求传输的数据量不大，而要求设备成本低；

（3）要求数据传输可靠性高，安全性高；

（4）要求设备体积很小，不便放置较大的充电电池或者电源模块；

（5）可以用电池供电；

（6）地形复杂，监测点多，需要较大的网络覆盖；

（7）对于那些现有的移动网络的盲区进行覆盖；

（8）已经使用了现存移动网络进行低数据量传输的遥测遥控系统。

课程思政

课程思政是指在专业课程教学中融入思想政治教育元素，培养学生的社会责任感、创新精神和实践能力。在 ZigBee 无线网络技术这一章节中，我们可以从以下几个方面来融入

1.10. 课程思政

1.技术创新与国家发展：

强调 ZigBee 技术在智能家居、工业自动化等领域的应用，以及这些技术如何推动社会进步和经济发展。

讨论中国在物联网和无线通信技术方面的自主创新能力，以及这些技术如何帮助中国

在全球市场中取得竞争优势。

2.社会责任与职业道德：

讨论在设计和实施 ZigBee 网络时，工程师应如何考虑环境保护和能源节约，以及如何确保用户隐私和数据安全。

强调在技术开发和应用过程中，应遵循的职业道德和社会责任，例如公平竞争、诚信经营等。

3.团队合作与沟通能力：

分析在 ZigBee 网络项目中，团队成员如何协作，以及有效沟通对于项目成功的重要性。

通过案例分析，让学生理解在多学科交叉的项目中，团队合作的价值和必要性。

4.可持续发展与环境保护：

讨论 ZigBee 技术在节能减排、智能电网等方面的应用，以及这些技术如何促进可持续发展。强调在技术开发中，应考虑环境影响，寻求环境友好型的解决方案。

5.创新精神与实践能力：

鼓励学生在学习 ZigBee 技术时，不仅要掌握理论知识，还要通过实验和项目实践来提升自己的动手能力和创新思维。分享成功案例，激励学生勇于探索新技术，解决实际问题。

6.知识产权保护：

强调在技术创新过程中，知识产权保护的重要性，教育学生尊重他人的知识产权，同时也要保护自己的创新成果。

通过这些课程思政的融入，可以帮助学生在掌握专业知识的同时，培养他们的社会责任感、创新精神和实践能力，为成为社会的有用之才打下坚实的基础。

第二章 ZigBee 无线网络构架

（一）本章教学目标

主要内容：

1．无线个域网，主要内容：无线个域网发展、IEEE802.15 系列标准、IEEE802.15.4 协议簇；

2．IEEE802.15.4 标准，主要内容：IEEE802.15.4 网络拓扑结构、IEEE802.15.4 协议栈层次结构、IEEE802.15.4 网络形成与维护、超帧结构、IEEE802.15.4 物理层、IEEE802.15.4MAC 协议；

教学目的与要求：

了解 IEEE802.15 系列标准解决的问题和技术要求、IEEE802.15.4 协议簇的主要目标和特征，掌握 IEEE802.15.4 网络拓扑结构、IEEE802.15.4 协议栈层次结构和各层功能，了解 IEEE802.15.4 网络形成与维护过程，掌握超帧的结构，熟悉 IEEE802.15.4 物理层的主要功能、帧结构和载波调制、IEEE802.15.4MAC 协议的功能、帧结构、及设备建立连接的过程。

（二）教学重点与难点

掌握 IEEE802.15.4 网络拓扑结构、IEEE802.15.4 协议栈层次结构和各层功能，掌握超帧的结构，熟悉 IEEE802.15.4 物理层的主要功能、帧结构和载波调制、IEEE802.15.4MAC 协议的功能、帧结构、及设备建立连接的过程。

2.1. 无线传感器网络构架

ZigBee 无线传感器网络由 PC 机部分、网关部分、路由节点部分、传感器节点部分四部分组成，用户可以很方便的实现传感器网络无线化，网络化，规模化的演示，教学，观测和再次开发。

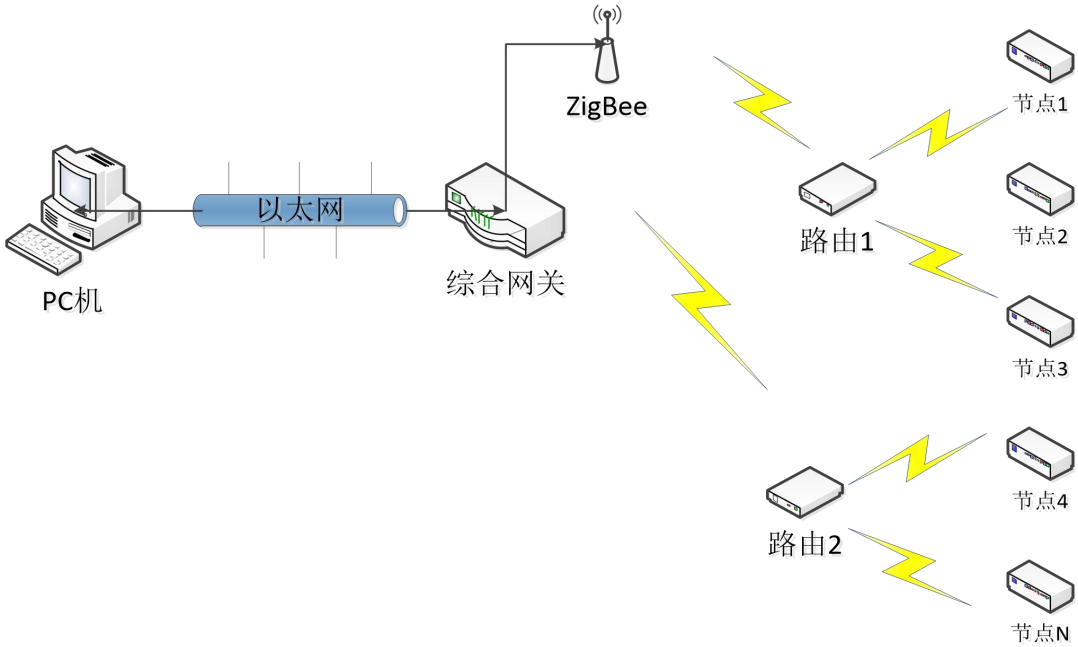
整体开发概念示意图如图所示。

1、**PC机**：完成接收网关数据和数据处理，实现可视化,形象化人机界面，方便用户操作，观察；

2、**网关**：完成通过计算机发送的指令发送或接收路由节点或者传感器节点数据，并将接收到的数据发送给计算机；

3、**路由节点**： 在网关不能和所有的传感器节点通信时，路由节点作为一种中介使网关和传感器节点通信，实现路由通信功能；

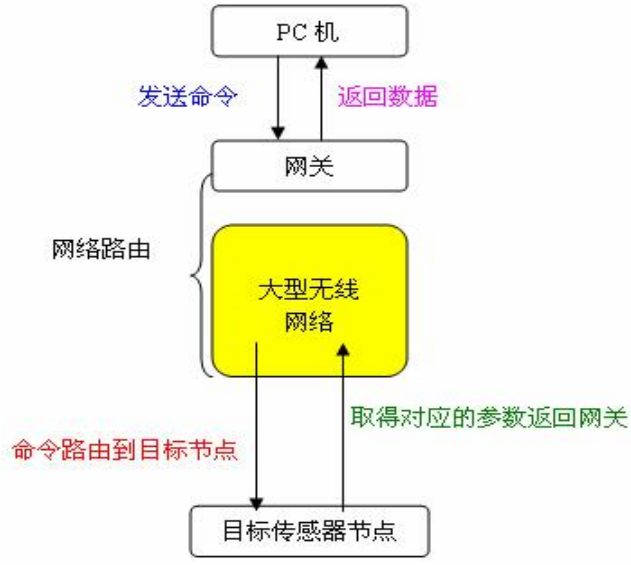
4、**传感器节点**：完成对设备的控制和数据的采集，包括灯的控制温度、光照度、加速度数据等等。



ZigBee无线传感器网络根据不同的情况可以由一个网关，一个或多个路由器，一个或多个传感器节点组成。系统大小只受PC软件观测数量，路由深度，网络最大负载量限制。

2.2. 无线传感器网络工作流程

ZigBee无线传感器网络基于ZigBee协议栈无线网络，在网络设备安装过程，架设过程中自动完成。完成网络的架设后用户便可以由PC机、ARM终端、平板电脑或者手持设备发出命令读取网络中任何设备上挂接的传感器的数据，以及测试其电压。简单的工作流程描述如下图所示。



2.3.无线传感器网络实验设备

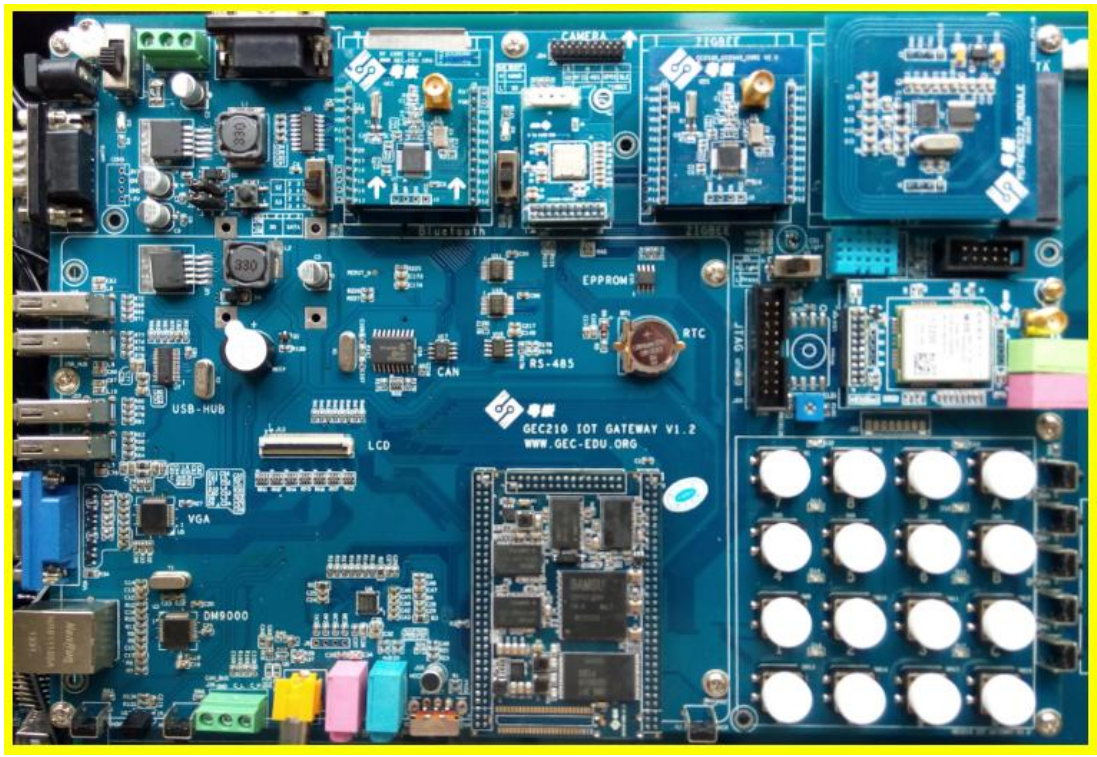
根据ZigBee无线传感器网络构架，综合实训开发平台配置多个无线网络传感节点，1个ZigBee网关协调器，1个综合网关。为了更好地了解ZigBee无线传感器网络开发，用户需要熟悉使用IAR软件集成开发环境。

综合实训开发平台配置设备清单：

序号	设备名称	数量
1	温湿度传感器模块	1
2	人体红外模块	1
3	无线调光 LED 灯控制器	1
4	智能灯光控制模块	1
5	电动窗帘模块	1
6	智能插座模块	1
8	雨滴模块	1
9	风速传感模块	1
10	风向传感模块	1
11	继电器控制器	1
13	3D 加速度传感板	1

14	磁场强度检测模块	1
15	超声测距传感模块	1
16	光照传感器	1
17	ZigBee 通讯网关（CC2530）	1
18	物联网综合开发网关	1
19	仿真器	1
21	PC 上位机	1

2.3.1. 物联网综合开发网关



作为网关设备，物联网网关可以实现感知网络与通信网络，以及不同类型感知网络之间的协议转换，既可以实现广域互联，也可以实现局域互联。此外物联网网关还需要具备设备管理功能，运营商通过物联网网关设备可以管理底层的各感知节点，了解各节点的相关信息，并实现远程控制。

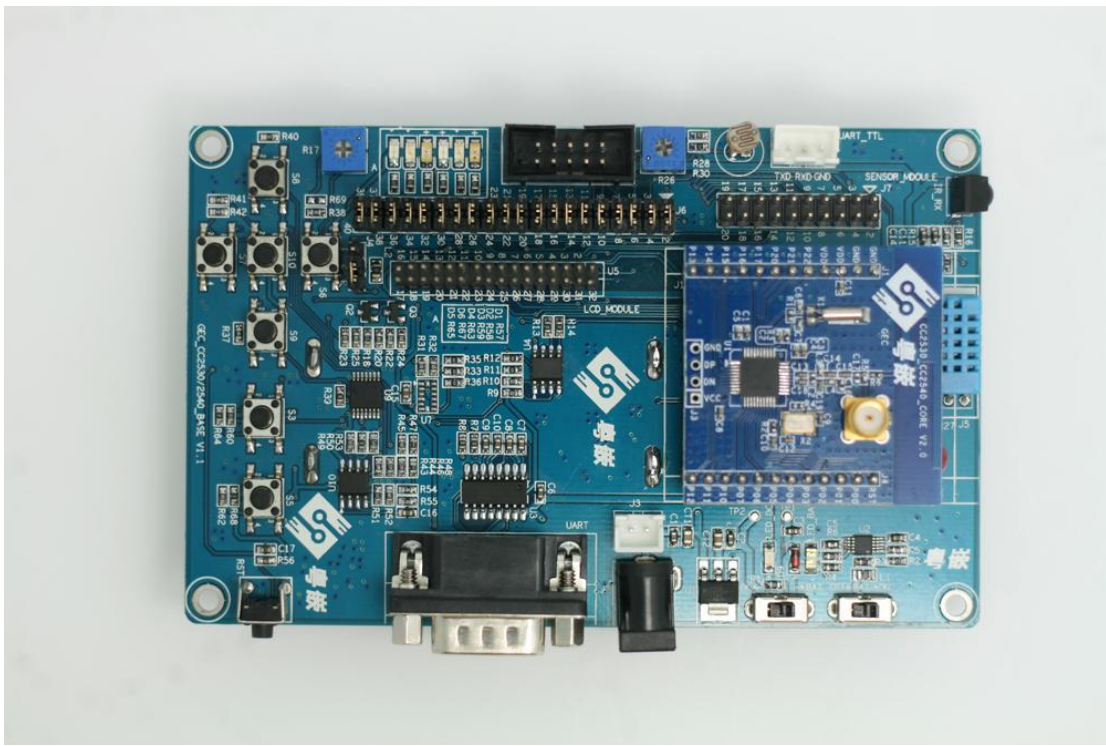
2.3.2. ZigBee 协调器

ZigBee规范定义了三种类型的设备，每种都有自己的功能要求：ZigBee协调器是启动和配置网络的一种设备。协调器可以保持间接寻址用的绑定表格，支持关联，同时还能设计信任中心和执行其它活动。协调器负责网络正常工作以及保持同网络其它设备的通信。一个ZigBee网络只允许有一个ZigBee协调器。

实训台配置的ZigBee模块均采用TI公司最新推出的ZigBee新一代SOC芯片CC2530，它支持IEEE802.15.4/ZigBee/ZigBee RF4CE标准，拥有庞大的闪存空间最多达256K字节，支持最新RemoTI的ZigBee RF4CE（这是业界首款ZigBee RF4CE兼容的协议栈），拥有更大RAM空间，允许芯片无线下载/空中升级。此外，CC2530结合了一个完全集成的，高性能的RF收发器与一个增强型8051微处理器以及其他强大的支持功能和外设。

CC2530提供了101dB的链路质量，优秀的接收器灵敏度和健壮的抗干扰性，四种供电模式，多种闪存尺寸，以及一套广泛的外设集——包括2个USART、12位ADC和21个通用GPIO，以及更多。除了通过优秀的RF性能、选择性和业界标准增强8051MCU内核，支持一般的低功耗无线通信，CC2530还可以配备TI的一个标准兼容或专有的网络协议栈（RemoTI，Z-Stack，或SimpliciTI）来简化开发，使你更快的获得市场。CC2530可以用于的应用包括远程控制、消费型电子、家庭控制、计量和智能能源、楼宇自动化、医疗以及更多领域。

粤嵌ZigBee开发套件



ZigBee通讯网关（协调器）

2.3.3. 无线传感节点

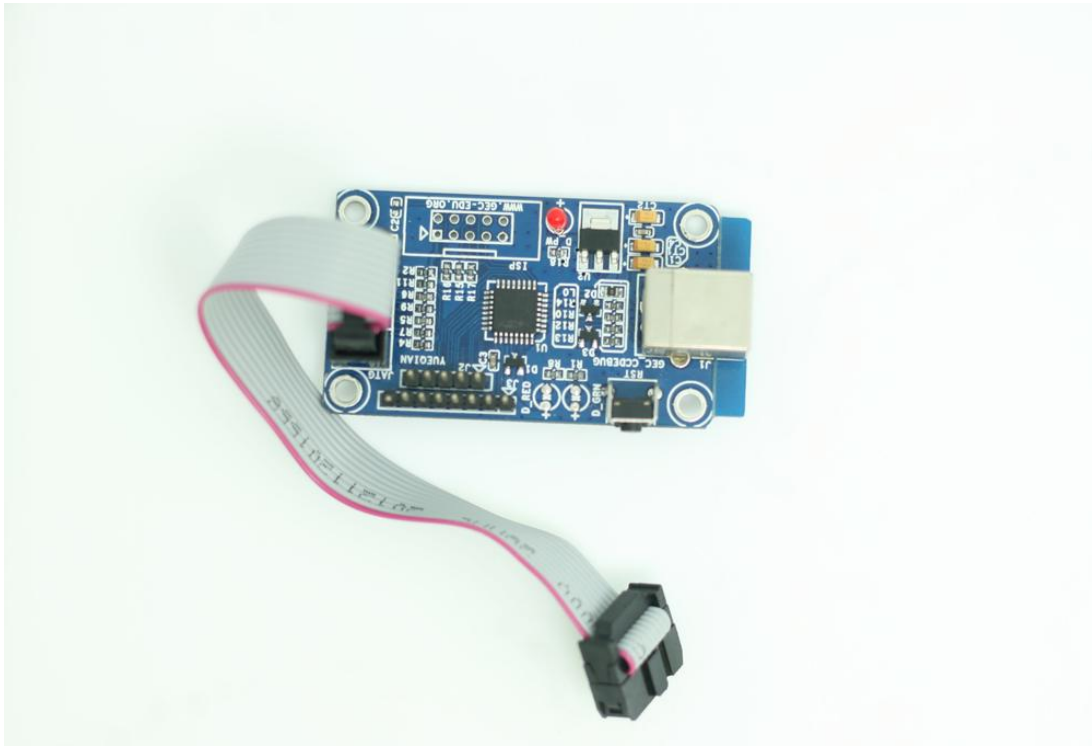
ZigBee传感节点

ZigBee传感节点背面电池座（5号1.5V）

无线传感节点也称为传感节点、感知节点等，主要用于各种传感器的数据采集或者相关设备的控制。

传感器和控制模块

ZigBee传感节点模块应用



ZigBee仿真器

熟悉过开发平台和相关的设备之后，我们就要开始安装和熟悉集成开发环境，正式进入ZigBee的开发，掌握ZigBee无线网络技术。

第三章 IAR 开发环境的搭建

（一）本章教学目标

主要内容：IAR 开发环境的搭建及 IAR 软件的操作方法

教学目的与要求：

了解 ZigBee 开发工具，熟悉 IAR 软件的安装与操作，熟悉 IAR 创建 ZigBee 工程项目，以及工程项目参数的设置，Z-Stack 协议栈的利用及编译调试下载方法。

（二）教学重点与难点

熟悉 IAR 创建 ZigBee 工程项目，以及工程项目参数的设置，Z-Stack 协议栈的利用及编译调试下载方法。

3.1. 相关软件安装

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。

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-CC2530-2.3.0-1.4.0 （Zigbee 2007），网上也有用 Zstack-CC2530-2.3.1-1.4.0 等其他高版本的，基本相差无几，但是目前 2.3.0 的通用性较高。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。IAR 8.10A 和 Zstack-CC2530-2.3.0-1.4.0 配合使用时从安装到开发都很友好。

3.1.1. 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 3.1

提示要求输入 License，由 IAR 8.10 注册机生成（参考图 3.3），选项正确后生成 License,复制到 License#处：

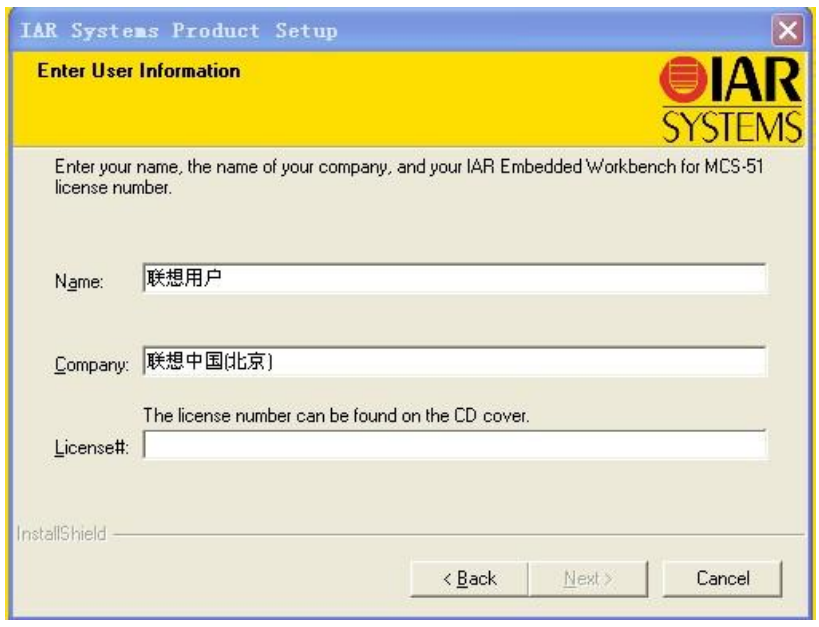


图 3.2

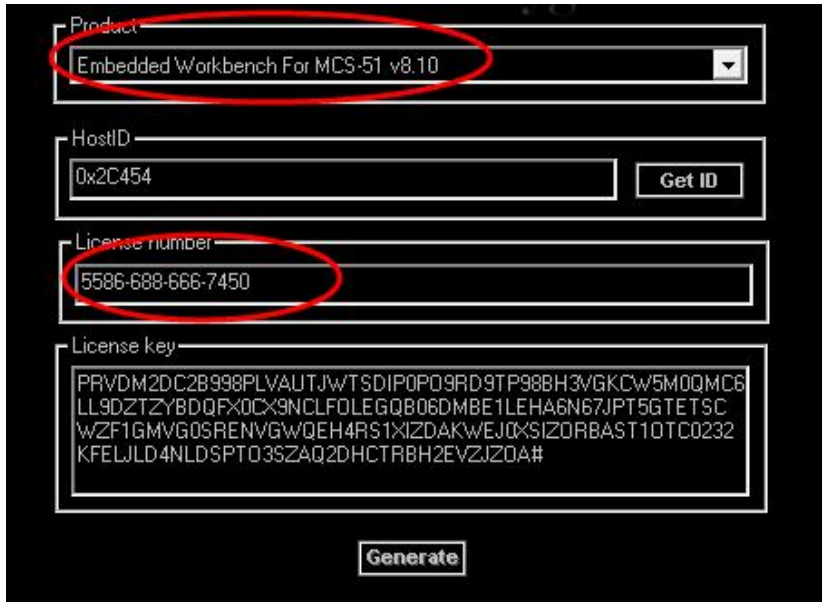


图 3.3 IAR 注册机

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：

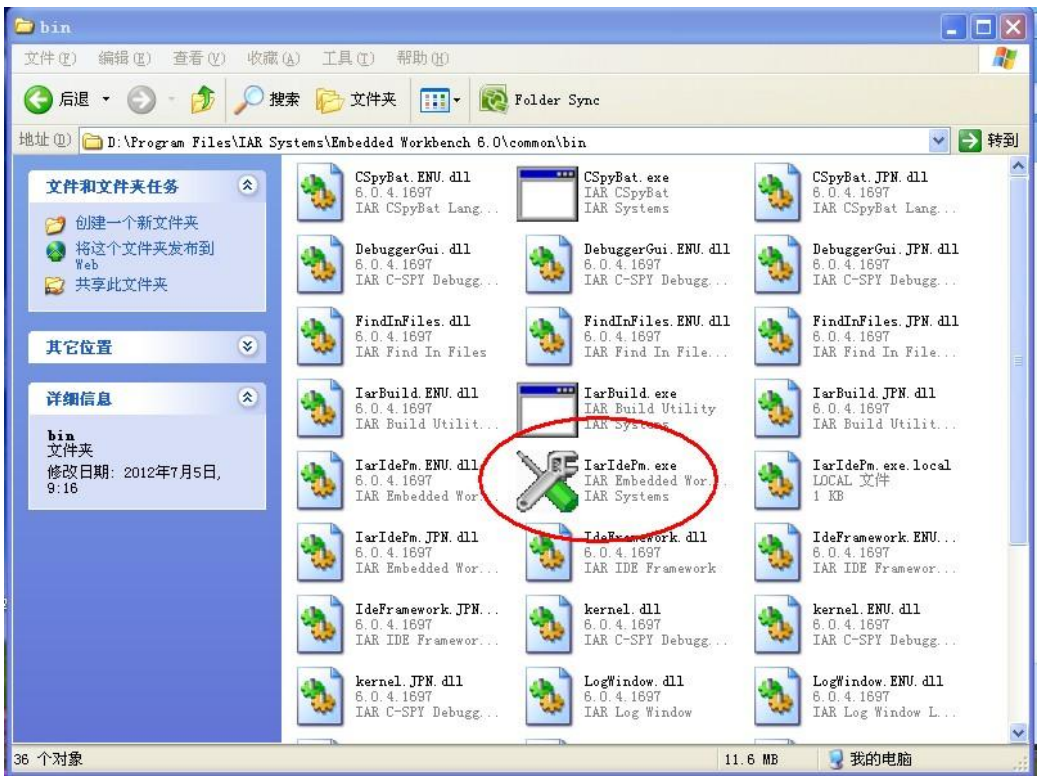


图 3.4 IAR 默认安装路径

安装完成软件界面如下：

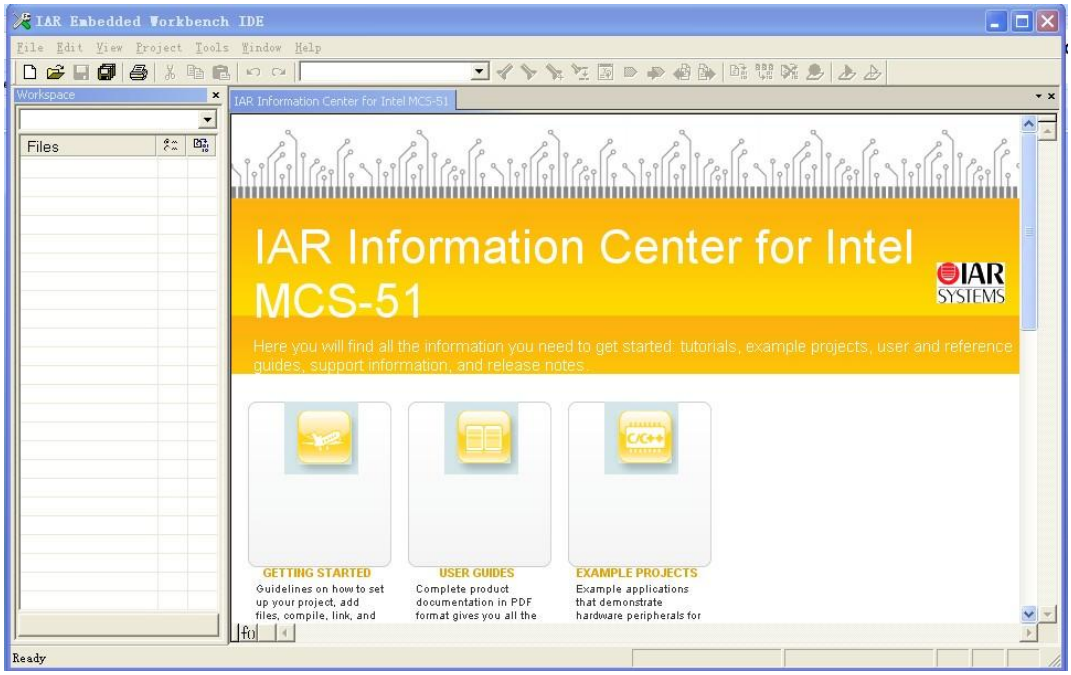


图 3.5 IAR8.10 软件界面

3.1.2. 第二步：TI 协议栈 Zstack-CC2530-2.3.0-1.4.0 安装方法

Z-stack 的安装比较简单，同样安装在默认路径。

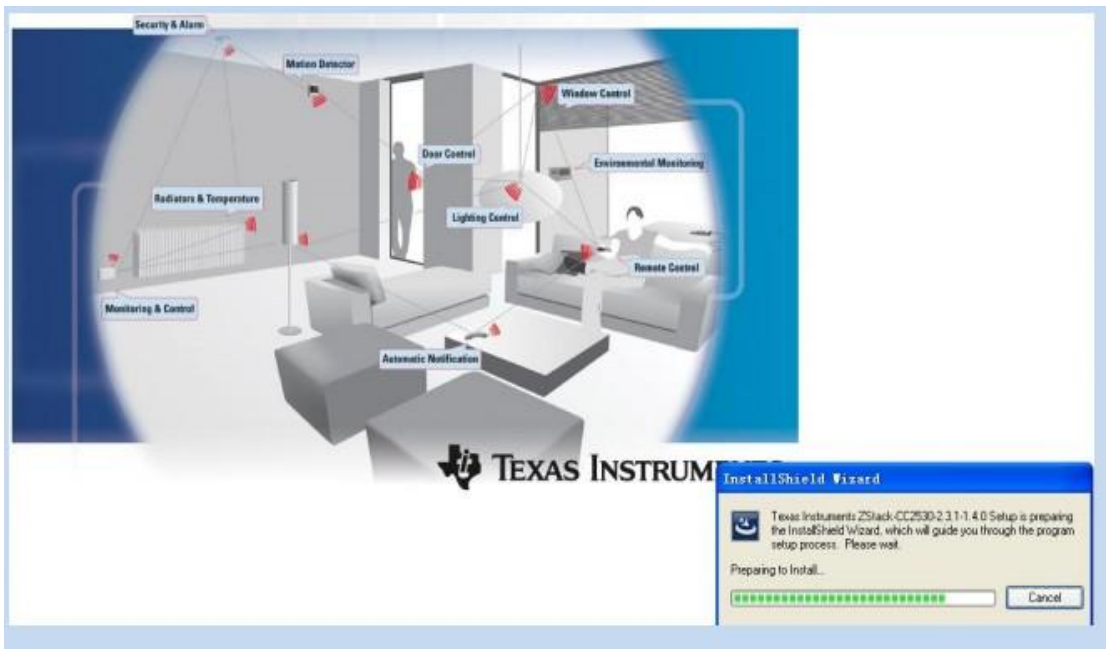


图 3.6 安装过程

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

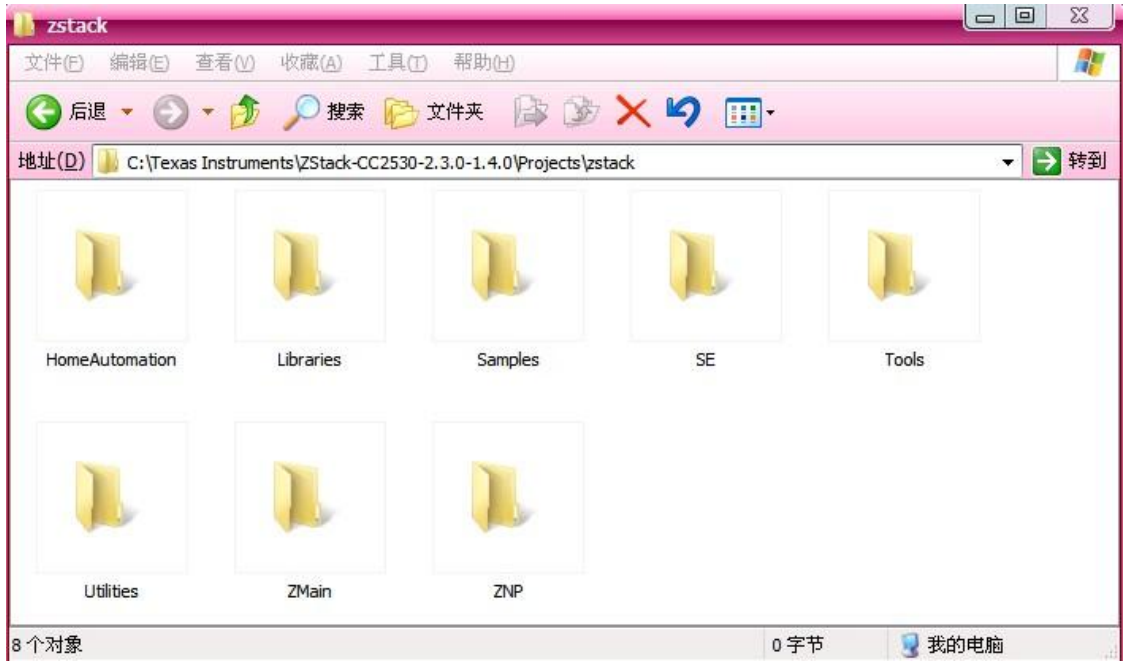


图 3.7 Z-stack 默认安装路径

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，学习其外围资源和内部寄存器，也就是基础实验。

3.1.3. 第三步：ZigBee 仿真器驱动安装方法

我们将 ZigBee 仿真器的一端通过 USB 线插进电脑，提示找到新硬件，选择列表安装。

图 3.8

驱动的路径如下图，前提是已经安装 IAR 8.1 0。

图 3.9

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，如下图所示。

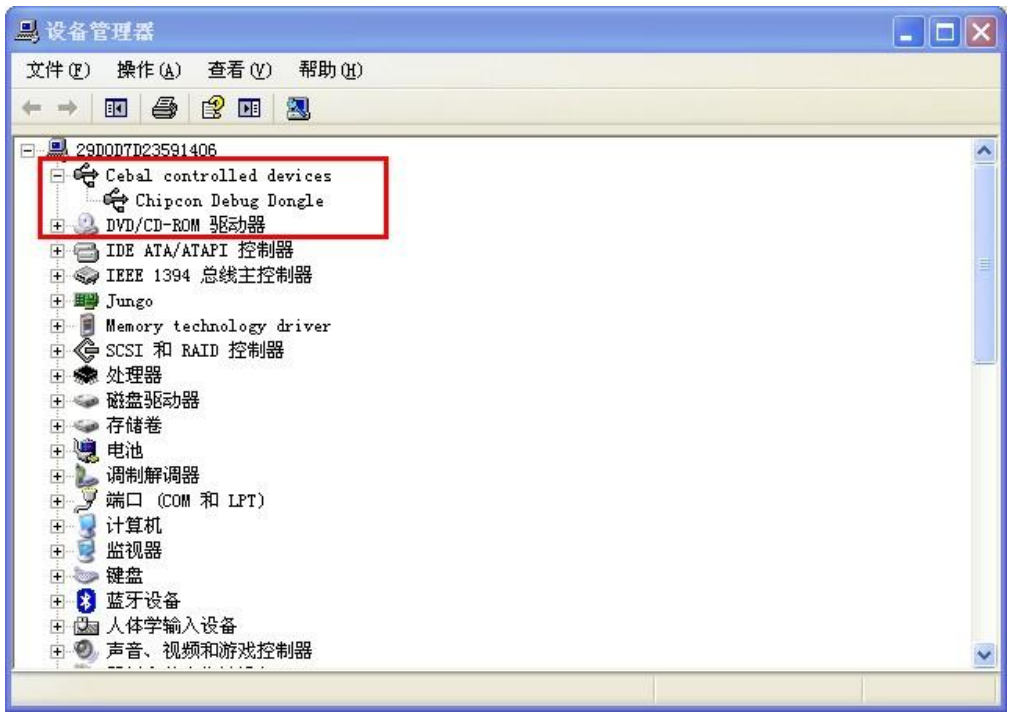


图 3.10 驱动安装完成

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。

3.2. 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。保存在自己想设定的路径。

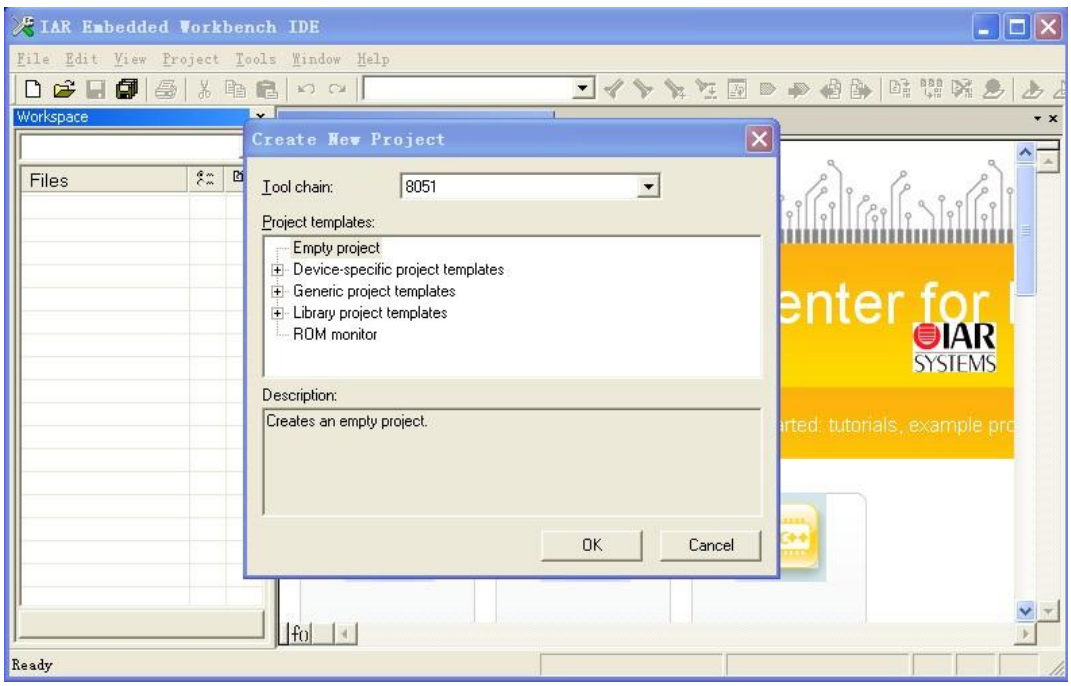


图 3.11

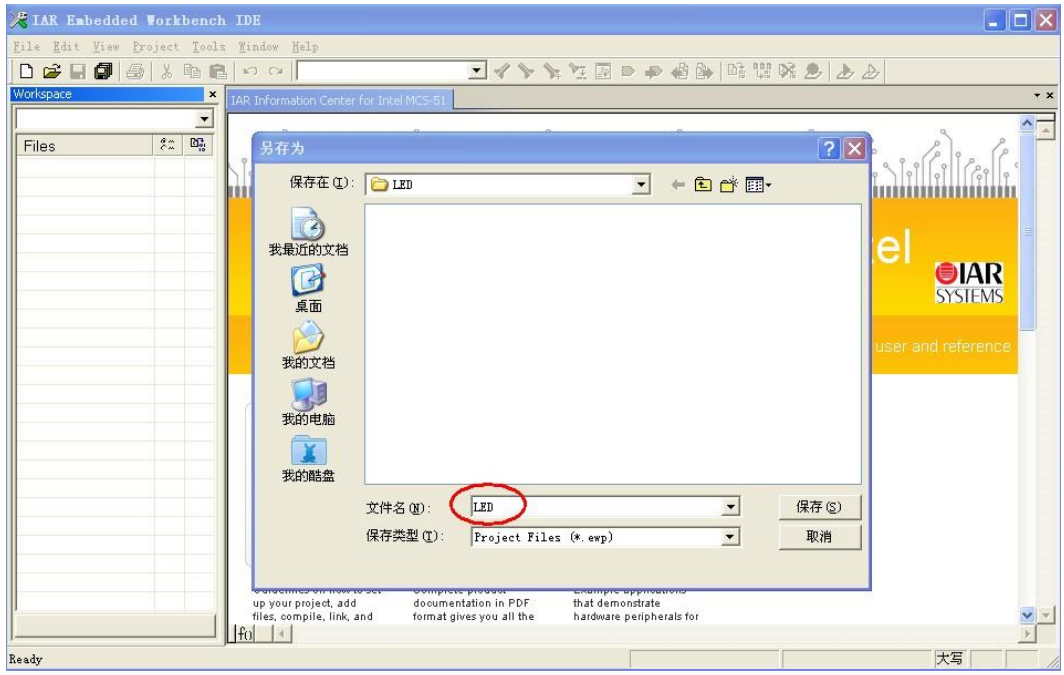


图 3.12

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。

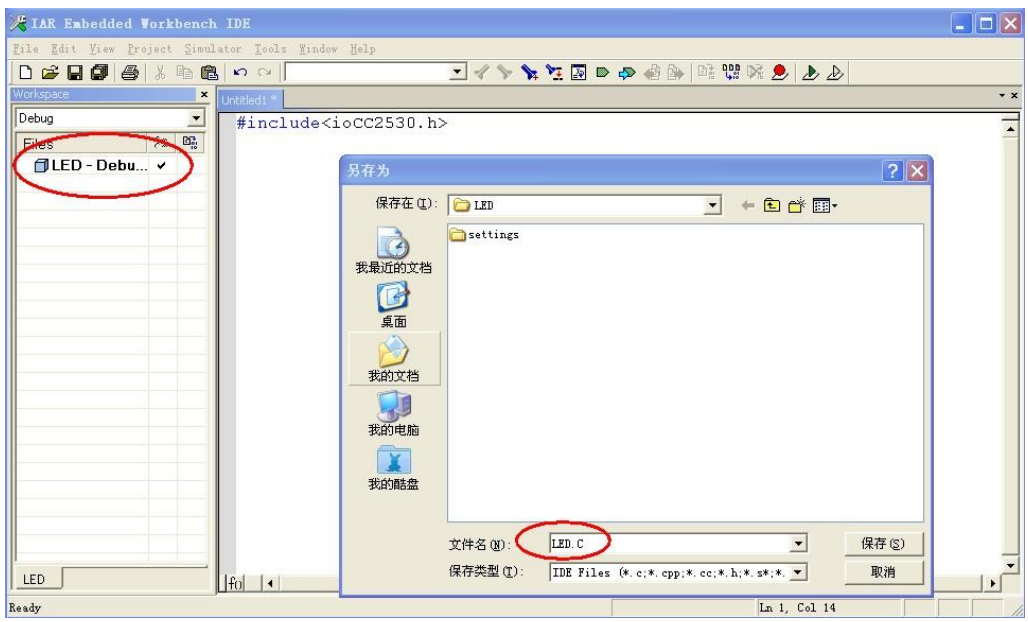


图 3.13 保存为.c 格式文件

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，成功添加后如图所示。

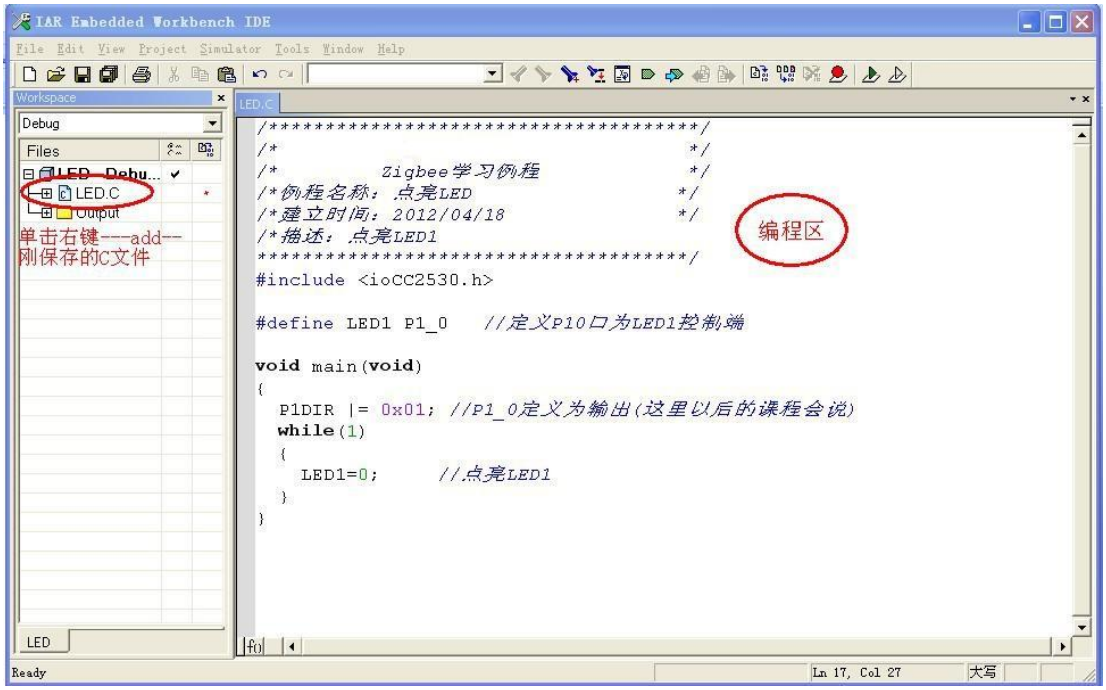


图 3.14 输入全部代码

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 1.25 **General Options 参数**，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。

选择 Linker—Config—Linker command file 选项。单击图 1.26 **Linker－Config 配置**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）。

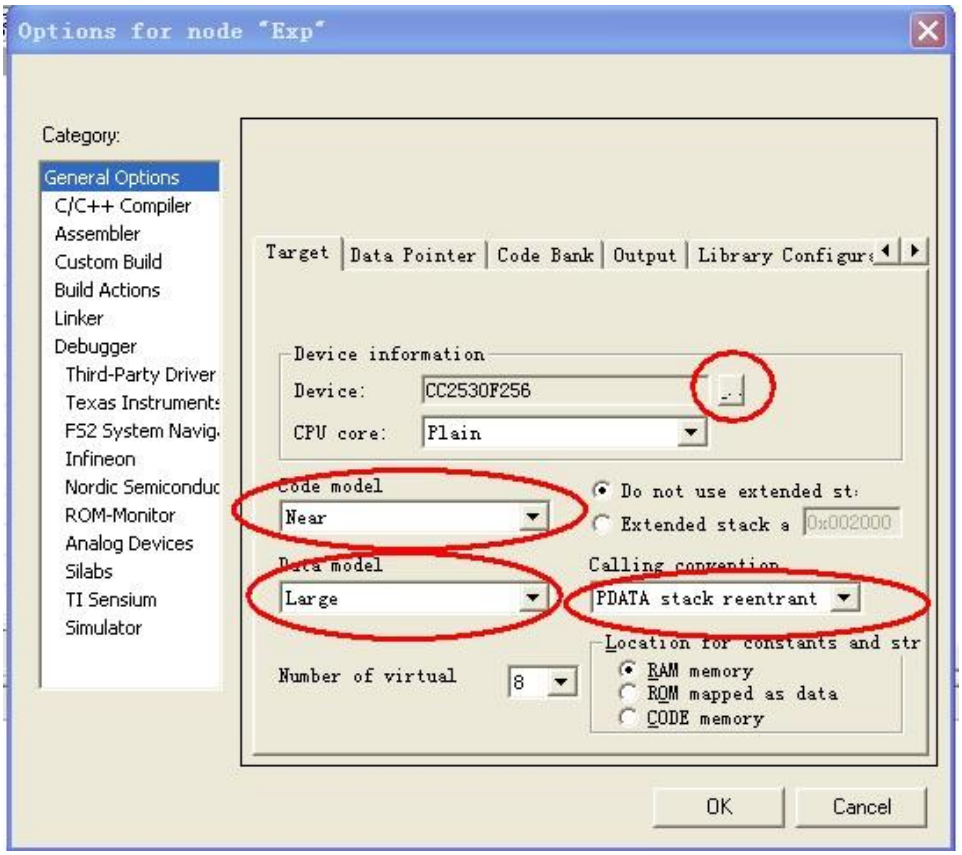


图 3.15 General Options 参数

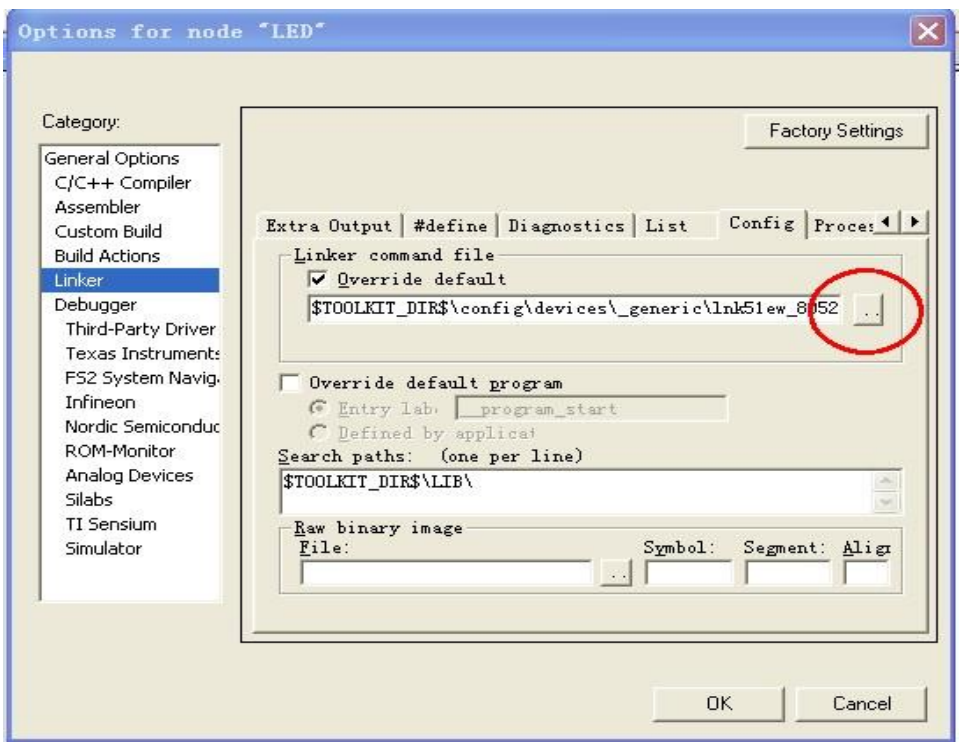


图 3.16 Linker－Config 配置

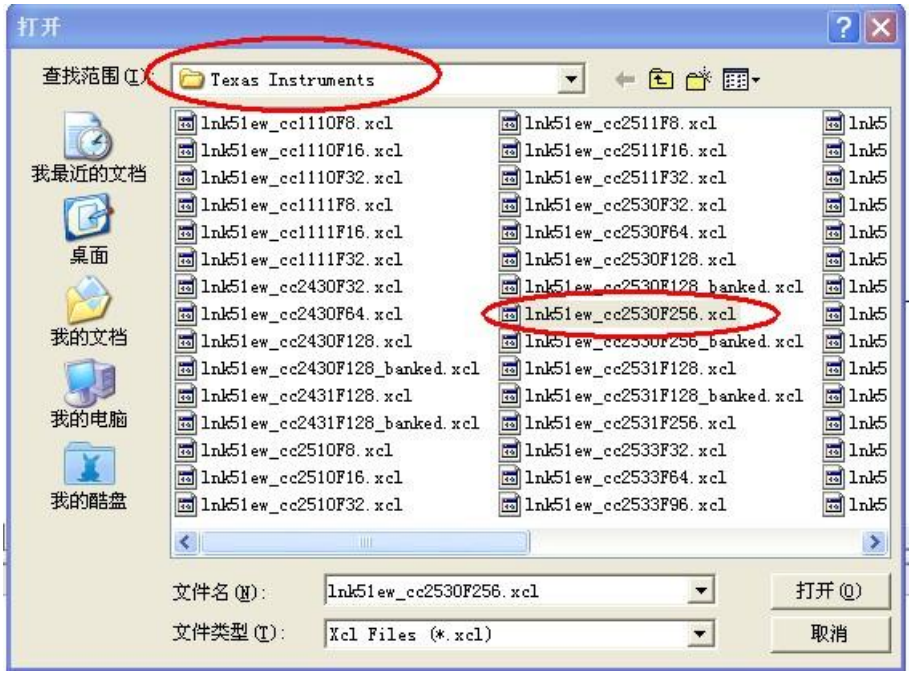


图 3.17

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。

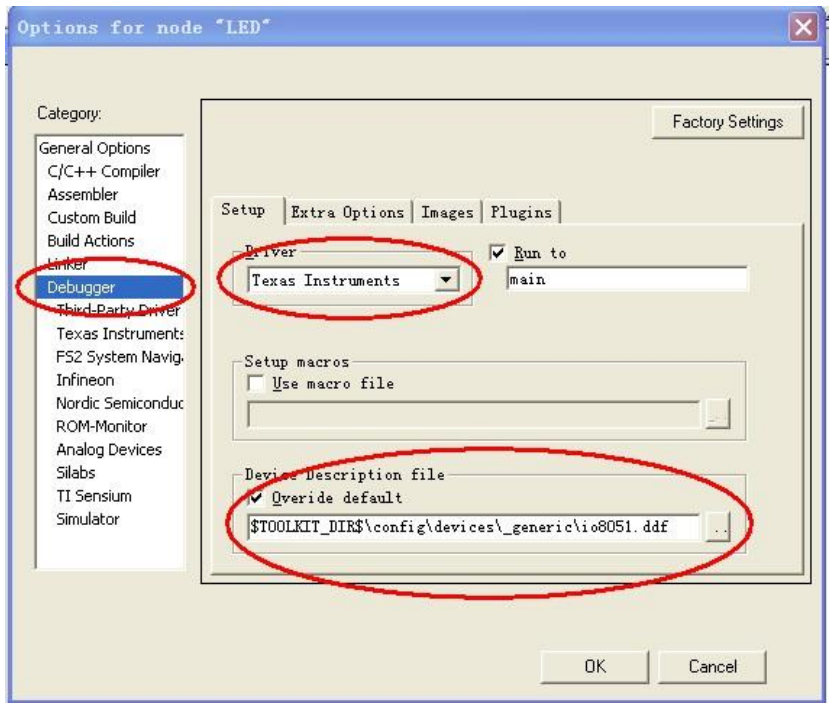


图 3.18　仿真器参数配置

第六步：Project-Make　编译后显示 0 错误和 0 警告。将 ZigBee 仿真器和开发板连接好，然后点击:Project-Download and Debug（下载与仿真）。快

捷键如图 3.19 所示：

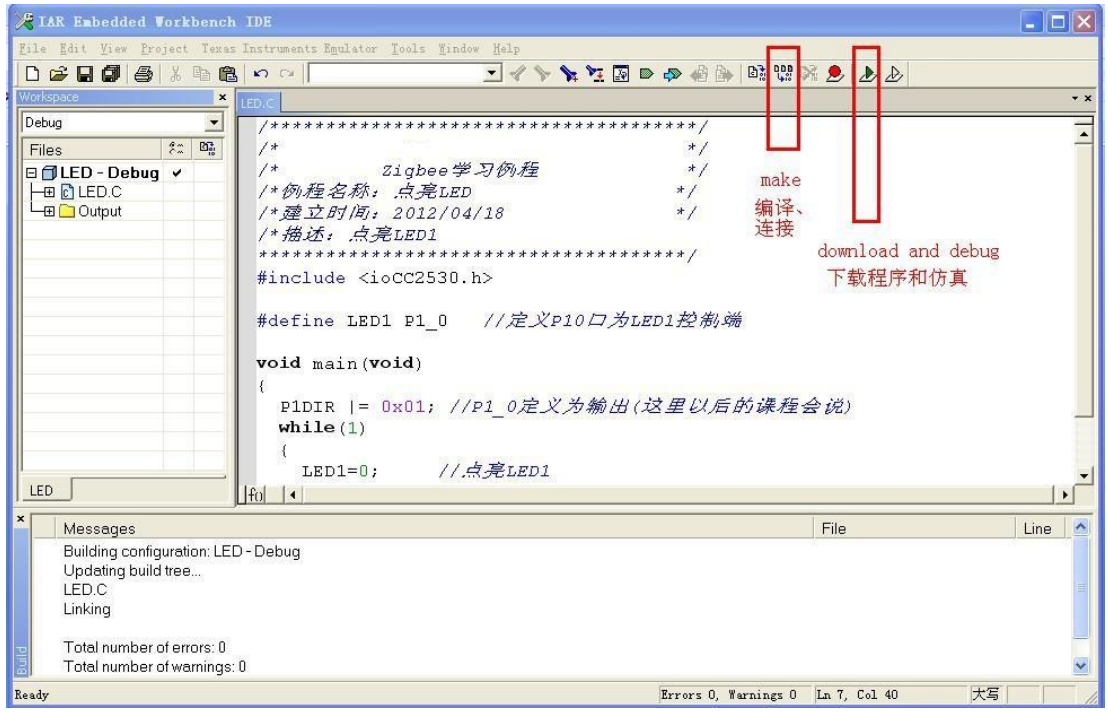


图 3.19

程序在下载中：

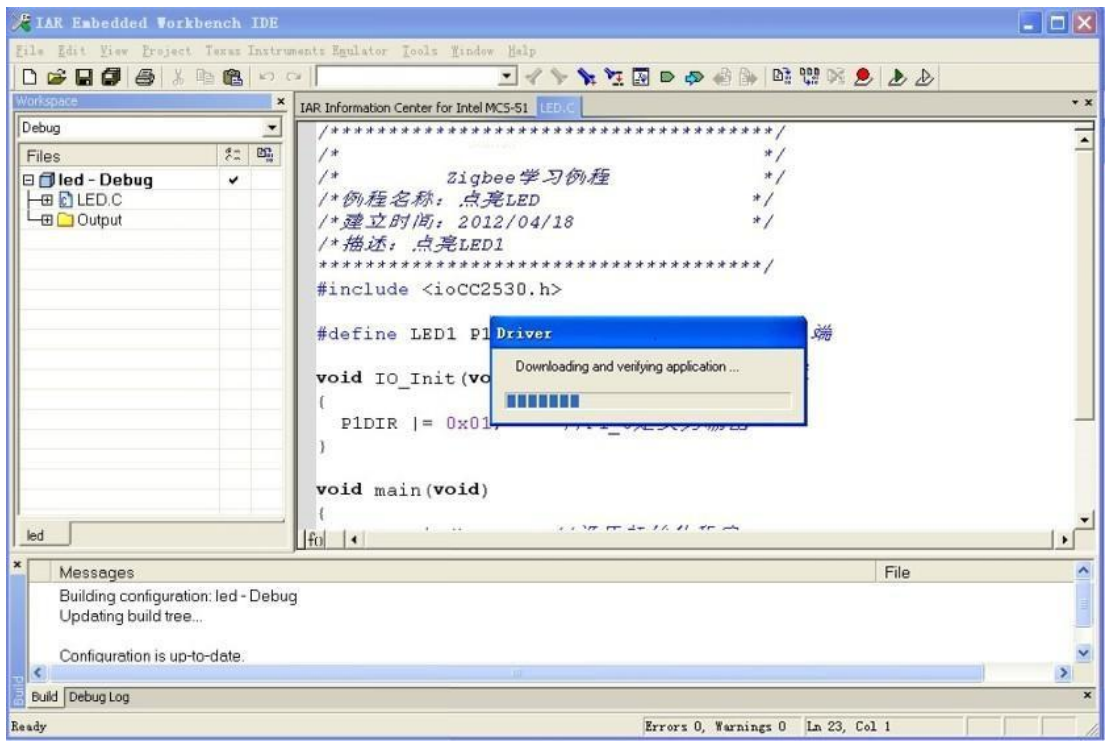


图 3.20

下载完成，进入仿真调试界面，常用按钮如图 3.21 所示。

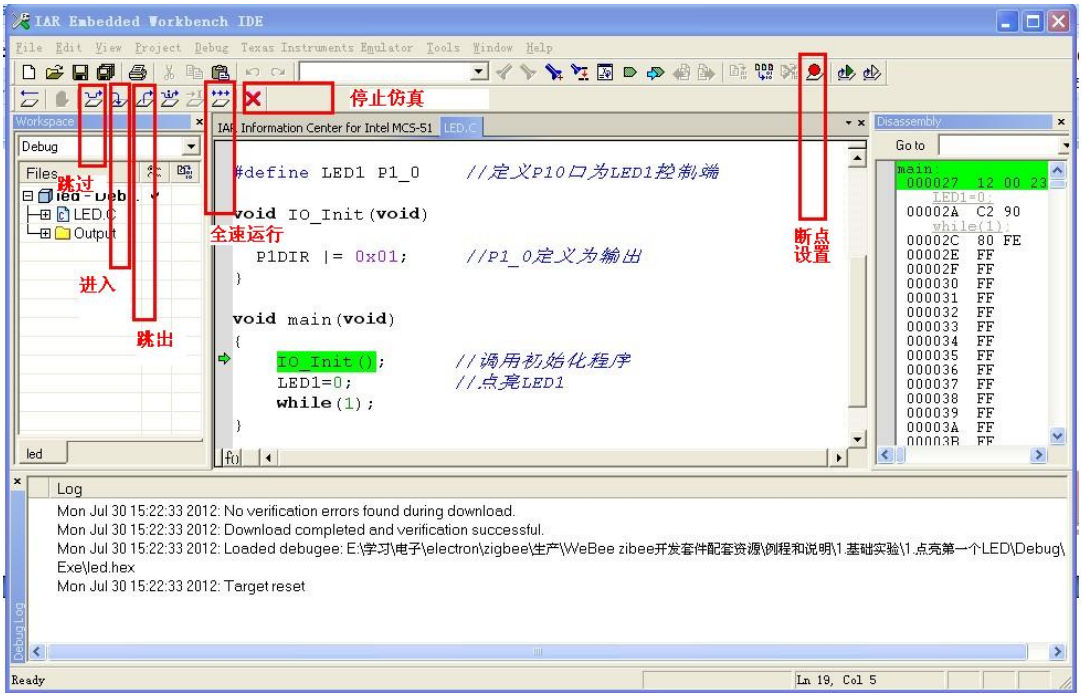


图 3.21

点击 GO(全速运行)，程序执行。使 ZigBee 仿真器可以直接在 IAR 中下载程序并调试。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。

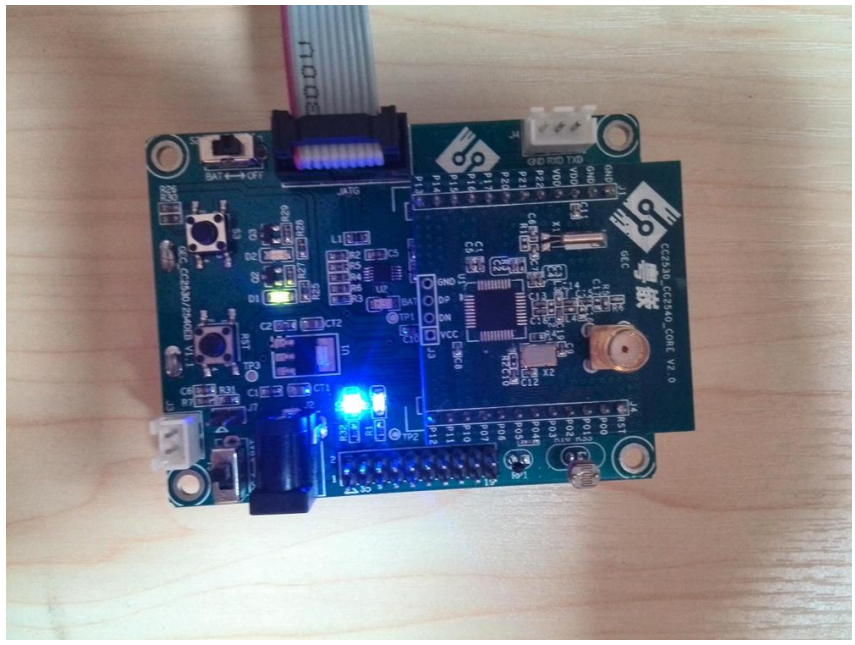


图 3.22 全速运行后，LED1 被点亮

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和操作铺好路。

3.3. 下载程序

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 3- 23 和图 3-24。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。

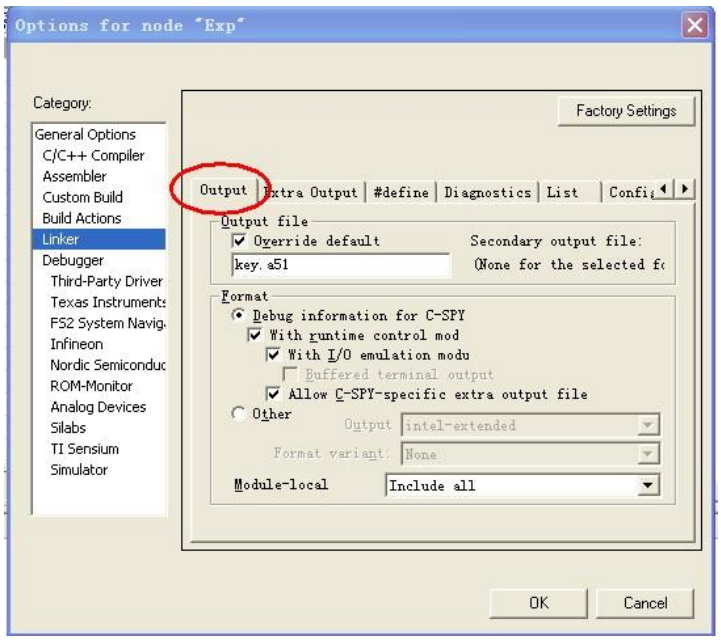


图 3-23

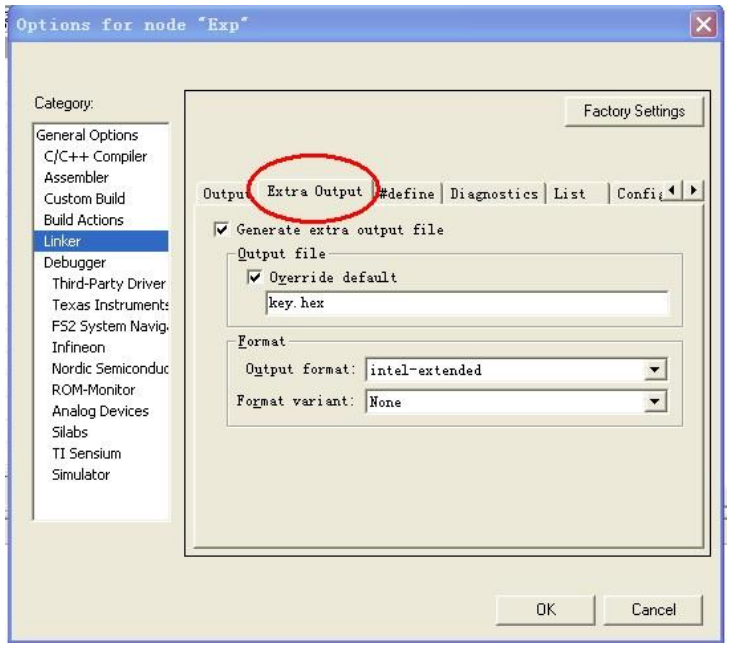


图 3- 24

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。

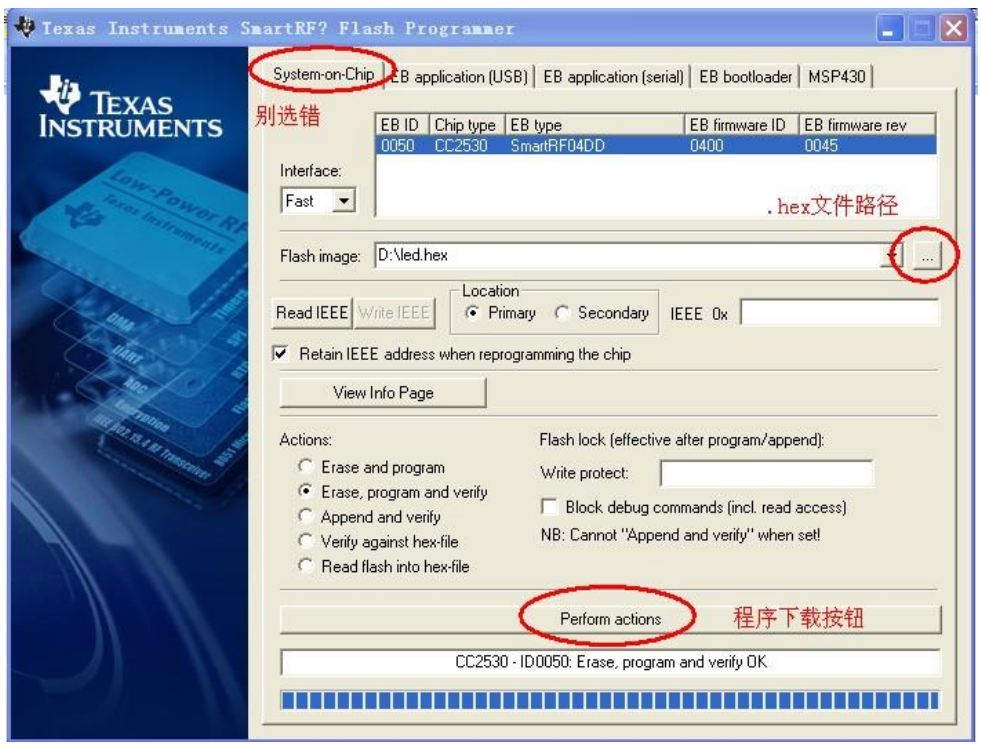


图 3- 25 TI SmartRF Flash Programmer

第四章 ZigBee 基础实验

（一）本章教学目标

通过基础性实验，让学生熟悉 CC2530 芯片及其节点板的结构及其接口，完成基础性实验。

教学目的与要求：主要熟悉 CC2530 芯片架构及其结构原理，完成基于 CC2530 芯片的基础性实验项目，并基本掌握 CC2530 系统原理及开发方法。

（二）教学重点与难点

重点与难点为 CC2530 结构原理及接口，IAR 工程的创建于设置。

4.1. CC2530 芯片架构

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。

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：记录程序下载到板上的图片示例。

4.2. 实验一：点亮第一个 LED

1、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。

2、实验现象：程序实验点亮 LED1

3、实验讲解：我们先来看看 ZigBee 传感节点的 LED 部分原理图：如图 4.1 所示：

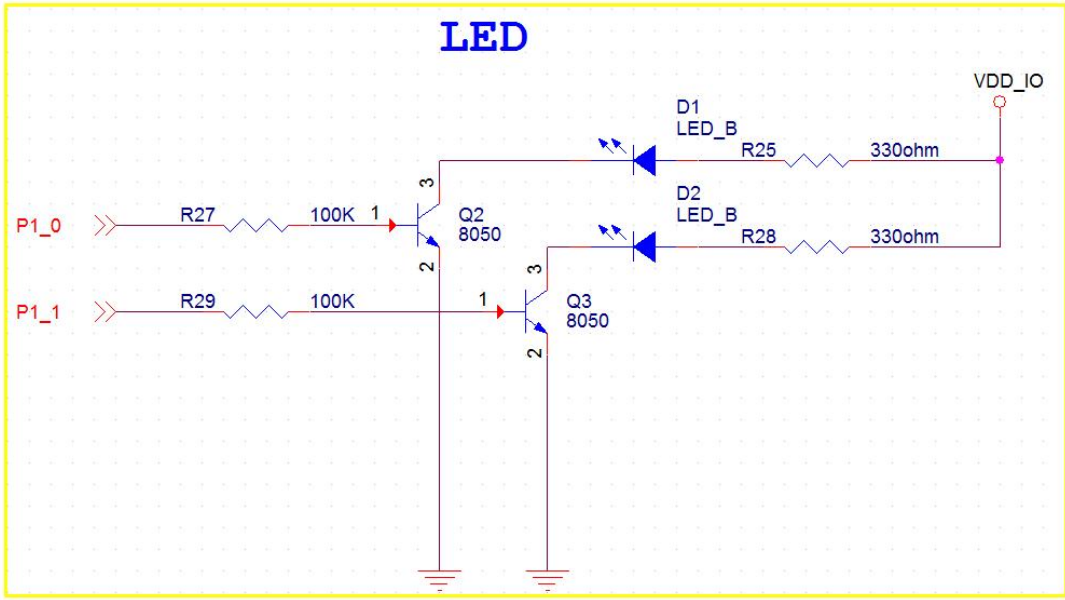


图 4.1 ZigBee 节点 LED 电路图

(详细原理图请查看 zigbee 节点板原理图)

LED1	LED2
P1_0	P1_1

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）

4、CC2530 IO 口寄存器

P1（0x90）端口 1

位	名称	复位	R/W	描述
7:0	P1[7:0]	0xFF	R/W	端口 1。通用 I/O 端口。可以从 SFR 位寻址。该 CPU 内部寄存器可以从 XDATA(0x7090)读，但是不能写。

P1SEL（0xF4）端口 1 功能选择

位	名称	复位	R/W	描述
7:0	SELP1_[7:0]	0xFF	R/W	P1.7 到 P1.0 功能选择 0： 通用 I/O

				1： 外设功能

P1DIR（0xFE）端口 1 方向

位	名称	复位	R/W	描述
7:0	DIRP1_[7:0]	0xFF	R/W	P1.7 到 P1.0 的 I/O 方向 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=1;           //点亮 LED1
while(1);  }
```

5、实验图片：

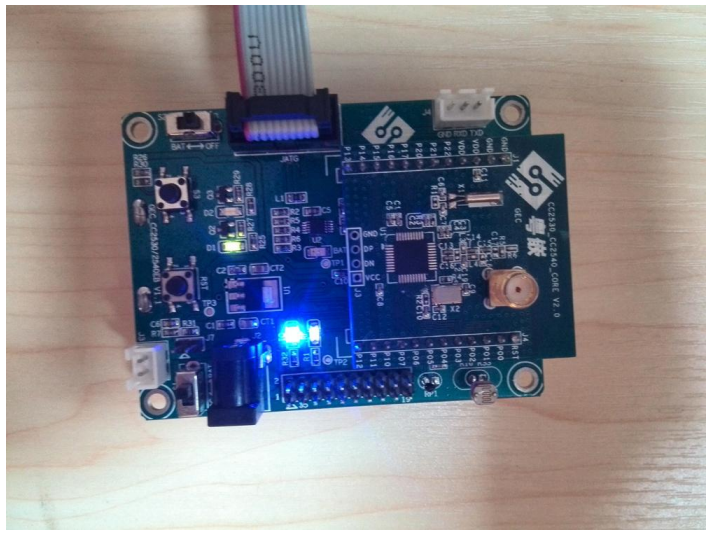


图 4.2　点亮 LED1

4.3.实验二：按键

1、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。

2、实验现象：依次按下按键 S3 控制 LED1 的亮和灭

3、实验讲解：我们先来看看 ZigBee 的 KEY 和 LED 原理图：

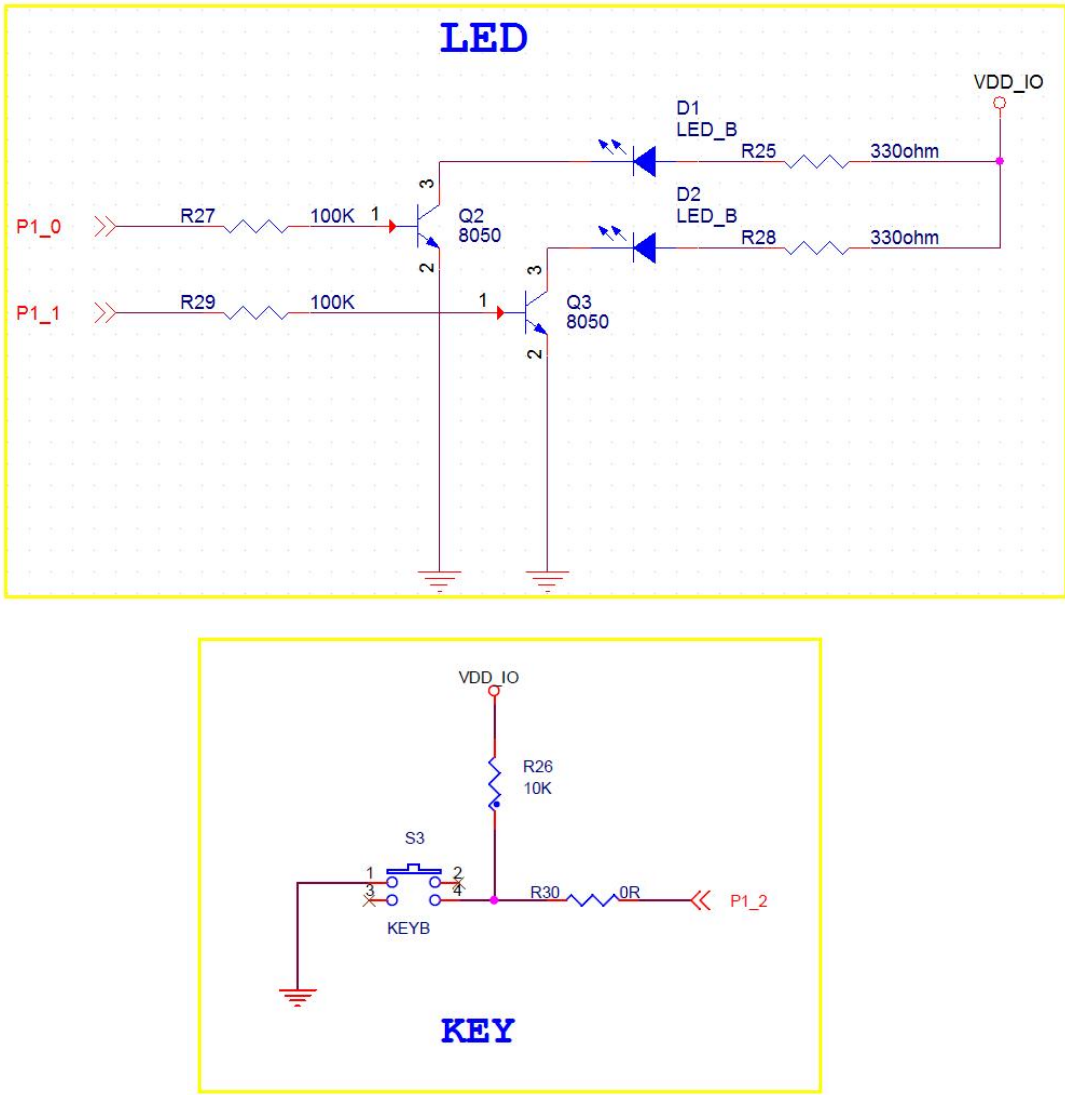


图 4.3 按键和 LED 原理图

LED1	LED2	S3
P1_0	P1_1	P1_2

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）

4、CC2530 IO 口寄存器

P1（0x90）端口 1

位	名称	复位	R/W	描述
7:0	P1[7:0]	0xFF	R/W	端口 1。通用 I/O 端口。可以从 SFR 位寻址。该 CPU 内部寄存器可以从 XDATA(0x7090)读，但是不能写。

P1SEL（0xF4）端口 1 功能选择

位	名称	复位	R/W	描述
7:0	SELP1_[7:0]	0xFF	R/W	P1.7 到 P1.0 功能选择 0： 通用 I/O 1： 外设功能

P1DIR（0xFE）端口 1 方向

位	名称	复位	R/W	描述
7:0	DIRP1_[7:0]	0xFF	R/W	P1.7 到 P1.0 的 I/O 方向 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 S3，也就是 P1.0 和 P1.2 口进行配置，当 P1.0 输出低电平时 LED1 被点亮,S3 按下时 P1.2 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 S3 初始化：

```
P1SEL &= ~0X04;     //设置 P1.2 为普通 IO 口
P1DIR &= ~0X04;     //按键在 P1.2 口，设置为输入模式
P1INP &= ~0x04;     //打开 P1.2 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
P1DIR &= ~0X04;    //按键在 P1.2 口，设置为输入模式
```

源程序代码（全）

```
/*****************************************
程序描述：依次按下按键 S3 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 KEY3  P1_2         //KEY1 为 P1.2 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()
{
  P1SEL &= ~0X04;     //设置 P1.2 为普通 IO 口
  P1DIR &= ~0X04;     //按键在 P1.2 口，设置为输入模式
  P1INP &= ~0x04;     //打开 P1.2 上拉电阻,不影响
}
/*****************************
     按键检测函数
*****************************/
uchar KeyScan(void)
{
  if(KEY3==0)
```

```
  {Delayms(10);
      if(KEY3==0)
      {
        while(!KEY3);  //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;
        }
}
```

5、实验图片：

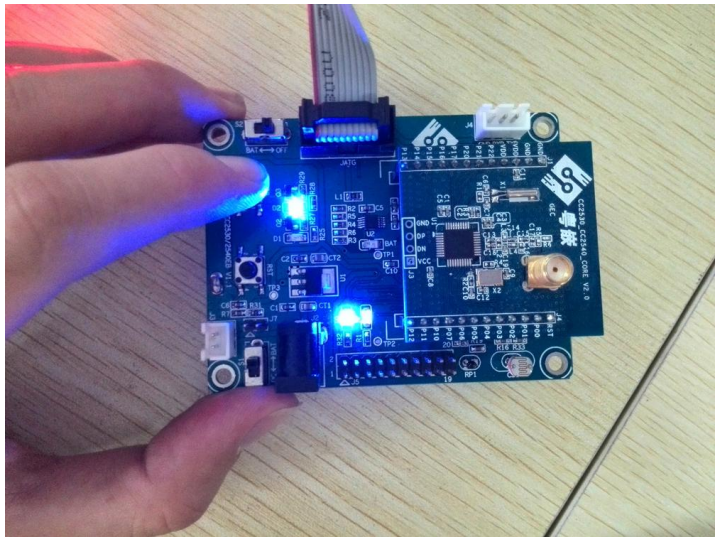


图 4.4 按键 S3 控制 LED1

4.4.实验三：外部中断

1、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。

2、实验现象：依次按下按键 S3 控制 LED1 的亮和灭,通过中断方式。

3、实验讲解：我们先来看 ZigBee 节点板的 KEY 和 LED 原理图，如下图所示：

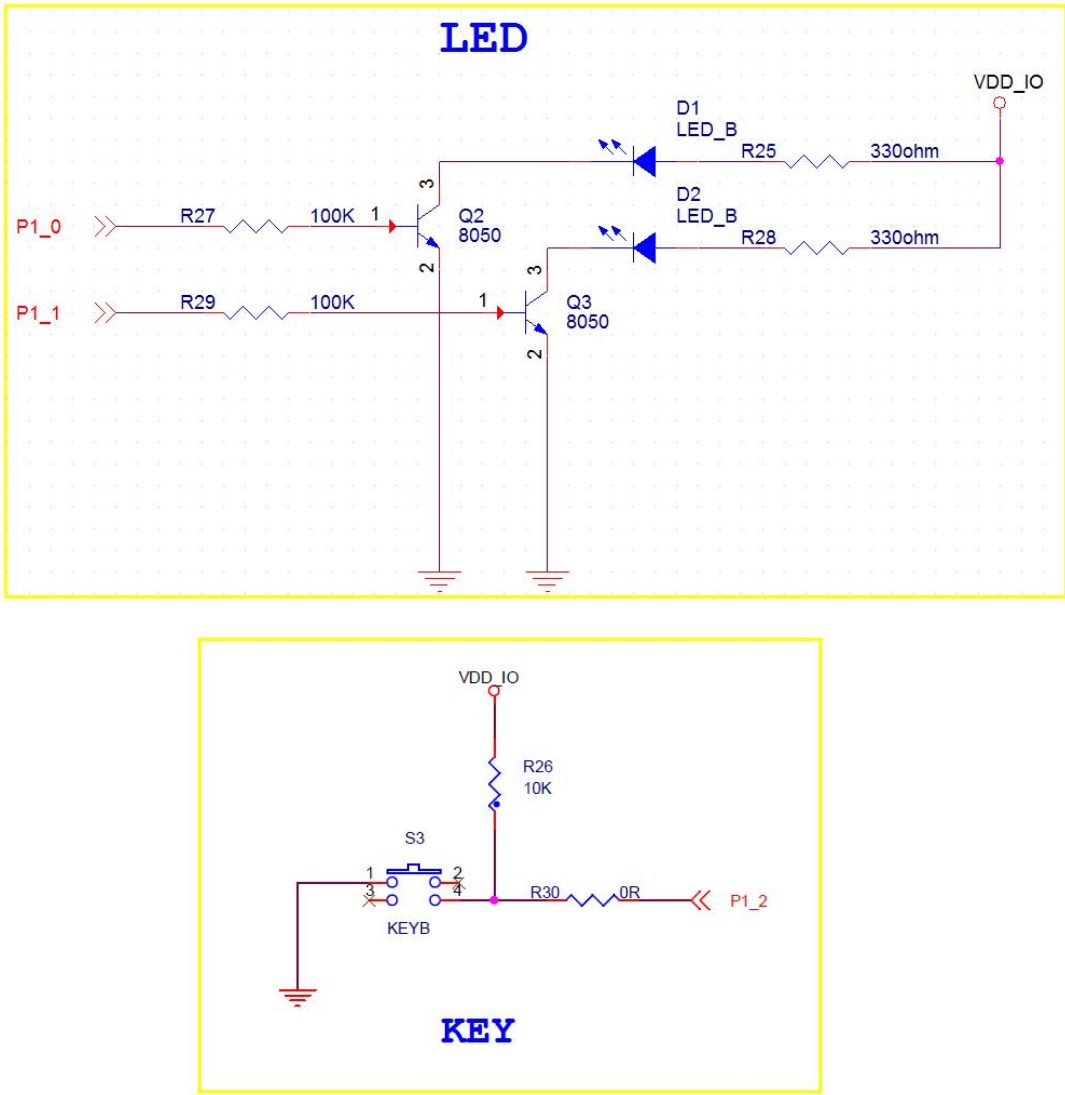


图 4.5 按键和 LED 原理图

LED1	LED2	S3
P1_0	P1_1	P1_2

4、实验原理

CC2530 的外部中断我们需要配置三个寄存器 P1IEN 、PICTL 、P1IFG、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 1 外部中断寄存器

P1IEN()	P1[7:0]中断掩码寄存器。 0：关中断 1：开中断
PICTL(0X8C)	P 口的中断触发控制器 Bit1 为 P1 为 P1[3：0]的中断触发配置

	0:上升沿触发 1：下降沿触发
P1IFG(0X8A)	P1[7：0]中断状态标志。当输入端口引脚中断请求未决信号时，其相应的标志位将置 1。
IEN2(0X9A)	Bit4 为 P1[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 P1.2 口进行配置，当 P1.2 输出低电平时 LED1 被点亮,S3 按下时 P1.2 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：

```
P1IEN |= 0X04;  //P1_2 设置为中断方式
PICTL |= 0X02;  // 下降沿触发
IEN2  |= 0X10;   // 允许 P1 口中断;
P1IFG  = 0x00;   // 初始化中断标志位
```

源程序代码（全）

```
/*********************************************
程序描述：按键 S3 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 KEY3 P1_2        //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、P1_1 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()
{
  P1IEN |= 0X04;  //P1.2 设置为中断方式
```

```
  PICTL |= 0X02;  // 下降沿触发
  IEN2 |= 0X10;   // 允许 P1 口中断;
  P1IFG &= ~0x04;   // 初始化中断标志位
  EA = 1;
}

/*****************************
      中断处理函数
*****************************/
#pragma vector = P1INT_VECTOR    //格式：#pragma vector = 中断向//量，紧接
                                 着是中断处理程序
  __interrupt void P1_ISR(void)
 {
  Delayms(10);              //去除抖动
  if(KEY3==0)
  {
  LED1=~LED1;               //改变 LED1 状态
  P1IFG &= ~0x04;           //清中断标志
  P1IF = 0;                //清中断标志
  }
  while(KEY3);
  }

/*****************************
          主函数
*****************************/
void main(void)
{
    InitLed();       //调用初始化函数
```

```
    InitKey();
        while(1)
    {
    }
}
```

5、实验图片：

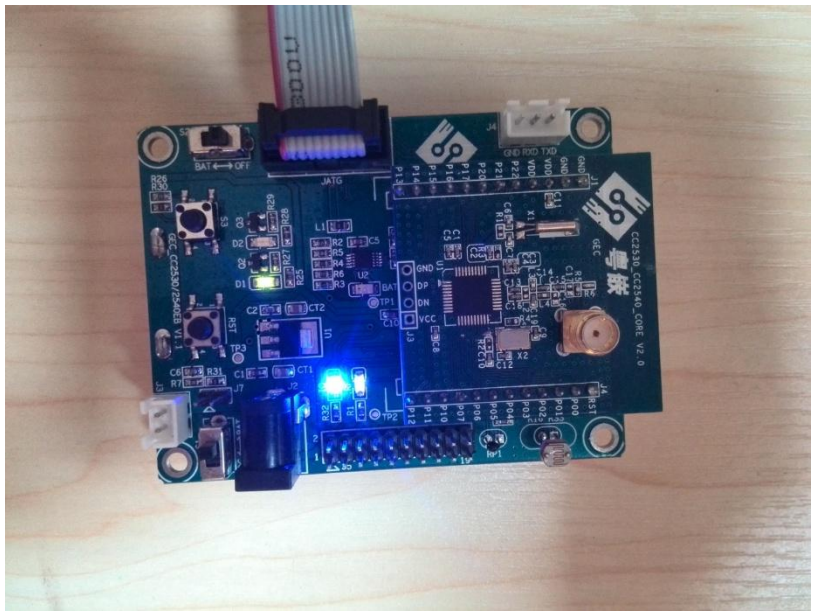


图 4.6 外部中断实验

4.5. 实验四：定时器（T1 查询方式）

1、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。

2、实验现象：分别利用定时器 T1 和 T3 使 LED 周期性闪烁

3、实验讲解：我们先来看看 ZigBee 传感节点的 LED 部分原理图：如所示。

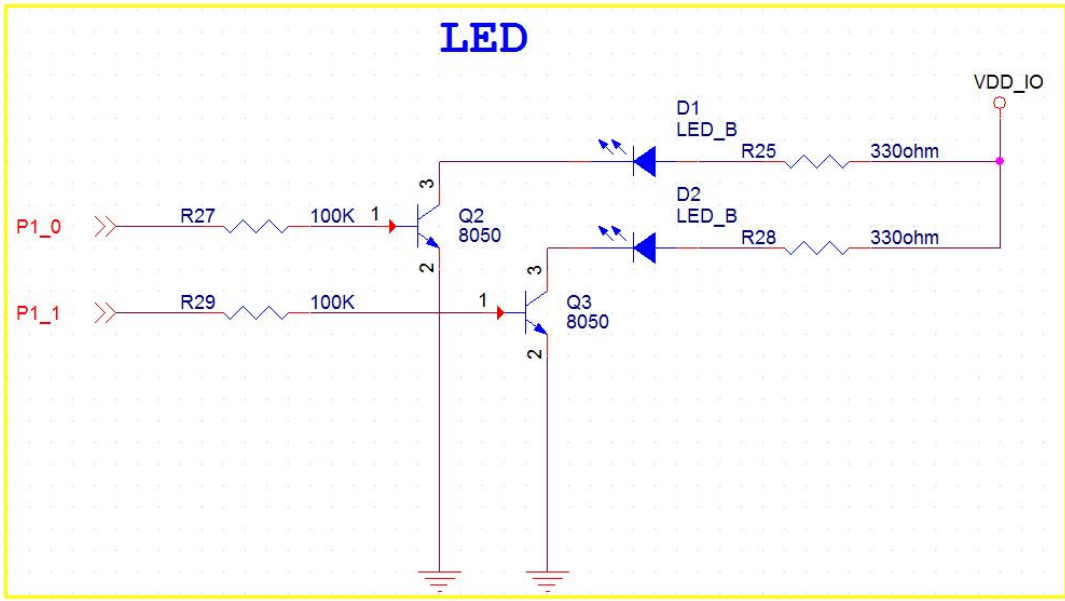


图 4.7 LED 原理图

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

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默认是 2 分频，即 16MHz
{
  T1CTL = 0x0d;              //128 分频，自动重装 0X0000-0XFFFF
  T1STAT= 0x21;              //通道 0，中断有效
}
/*************************
     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=0.25*2=0.5S。所以看起来是 1S 闪烁 1 次。

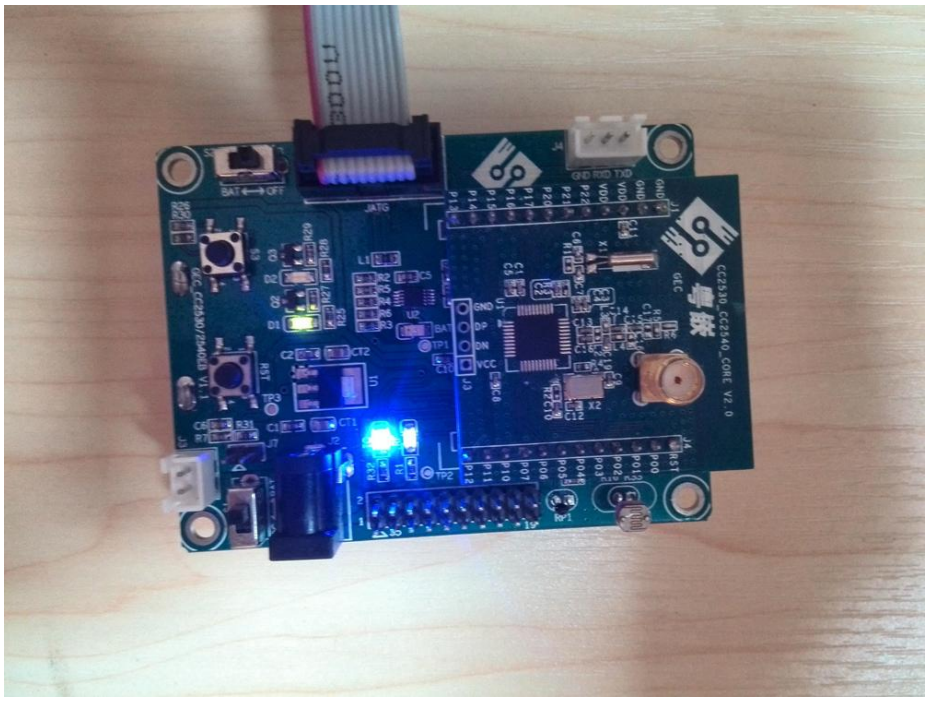


图 4.8 LED1 闪烁

4.6.实验五：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 ZigBee 协调器串口通讯功能。串口发送(HELLO WORLD)

实验讲解：我们先来看看 ZigBee 协调器原理图：如图 1 所示。

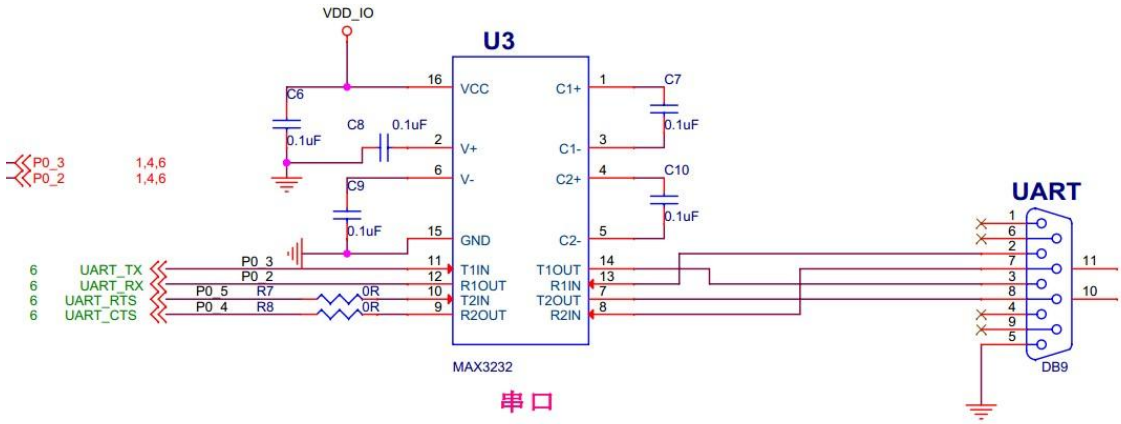


图 4.9 串口电路

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 3 串口寄存器

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 MOSI ；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
U0BAU（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 WORLD
        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 WORLD    “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 WORLD “);        //将发送内容 copy 到 Txdata;
while(1)
     {
         UartSend_String(Txdata,sizeof(“HELLO WORLD “));  //串口发送数据
         Delay_ms(500);                                    //延时
         LED1=!LED1;                                       //标志发送状态
     }
}
```

实验图片：

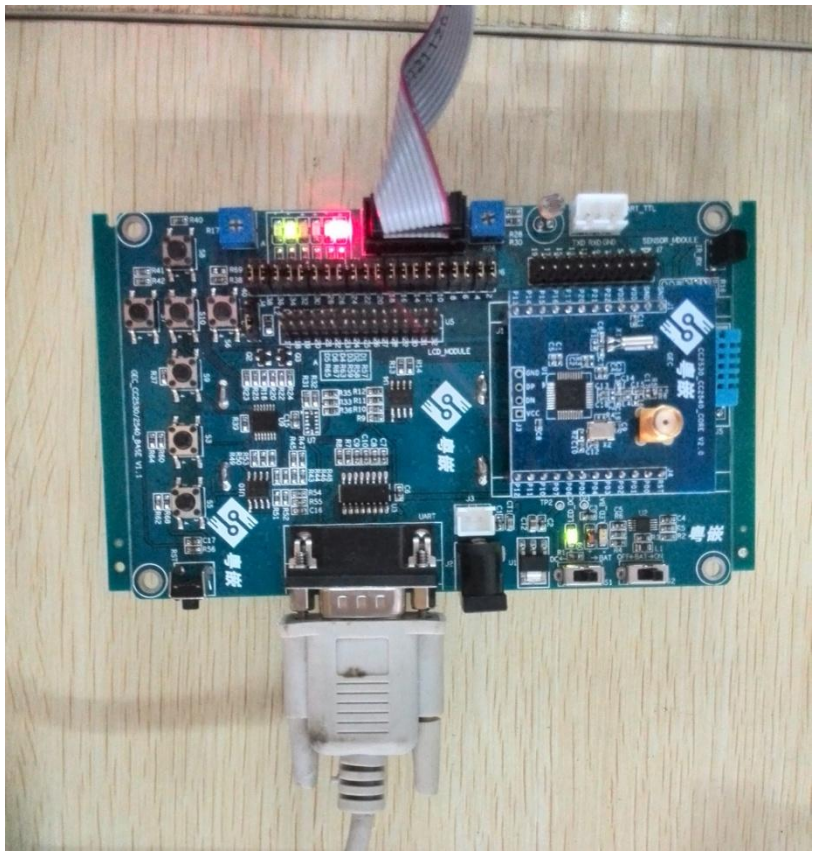


图 4.10　串口连接方法

图 4.11 上位机接收到发来的 “HELLO WORLD”

4.7. 实验六：睡眠唤醒（中断方式唤醒）

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 通过按键中断方式唤醒。通过 LED 状态展示。

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 ZigBee 节点板 LED 部分原理图：如下图所示。

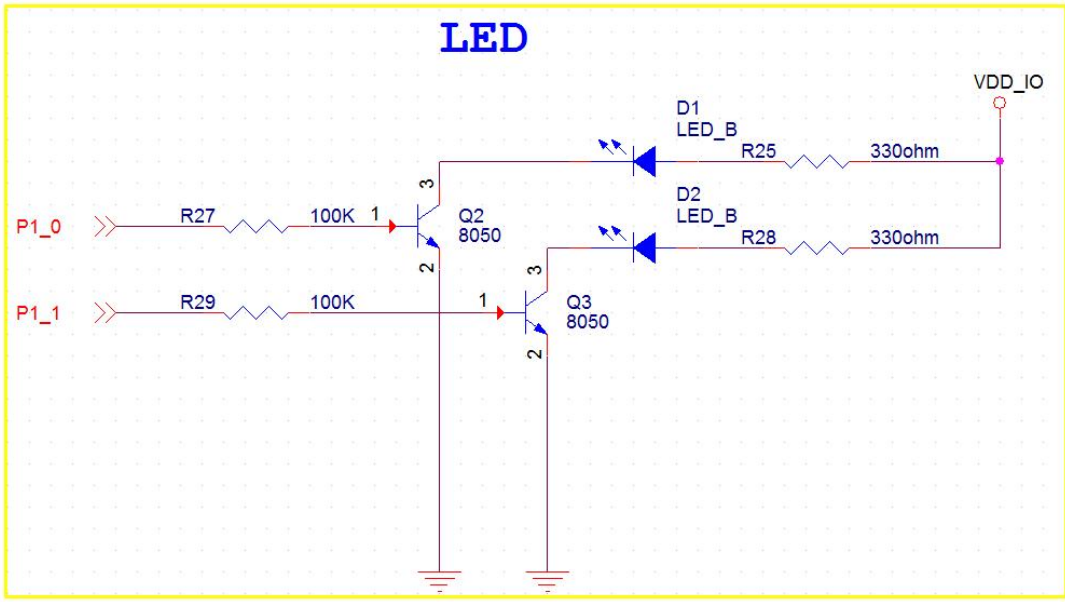


图 4.12 LED 原理图

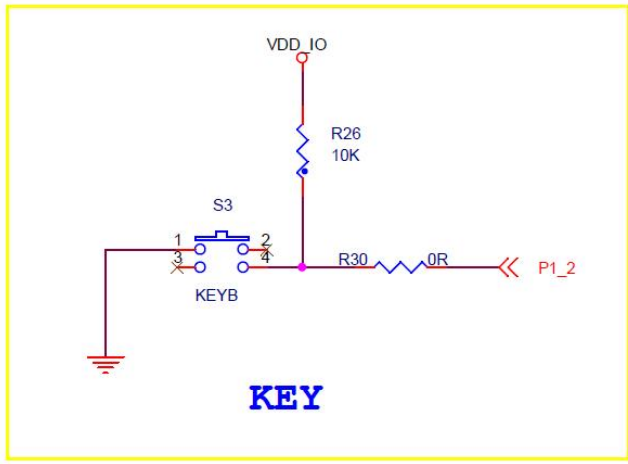


图 4.13 按键原理图

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）

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

表 4

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 S3 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 P1_2

//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
}

void InitKey()
{
  P1IEN |= 0X04;  //P1.2 设置为中断方式
  PICTL |= 0X02;  // 下降沿触发
  IEN2 |= 0X10;   // 允许 P1 口中断;
  P1IFG &= ~0x04;   // 初始化中断标志位
  EA = 1;
}
/*************************************************************
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
        InitKey();
        IEN0 |= 0X20;           //  开睡眠允许中断
        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 = P1INT_VECTOR
  __interrupt void P1_ISR(void)
 {
      P1IFG &= ~0x04;          //清中断标志

      P1IF =0 ;               //清中断标志

     SysPowerMode(4); //正常工作模式
 } }
```

实验图片：

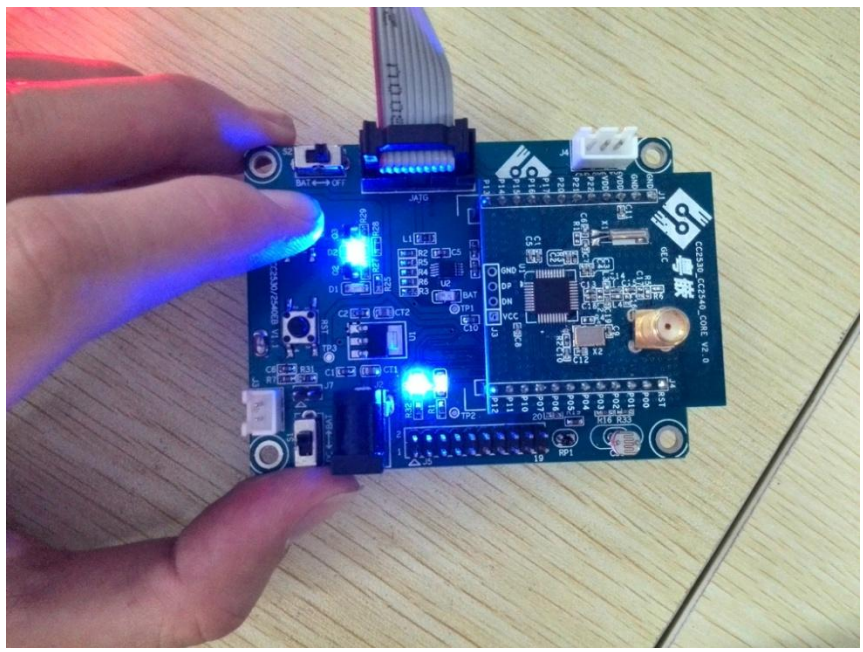


图 4.14　睡眠中断唤醒

4.8.实验七：ADC 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。

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 5

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：

```
TRO   = 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。

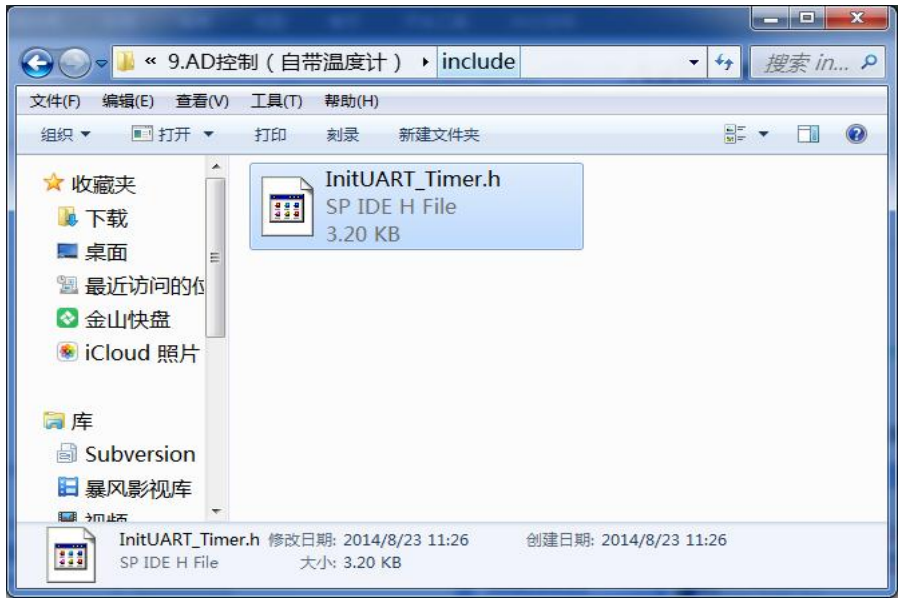


图 4-15

2.粘贴到下面这个位置

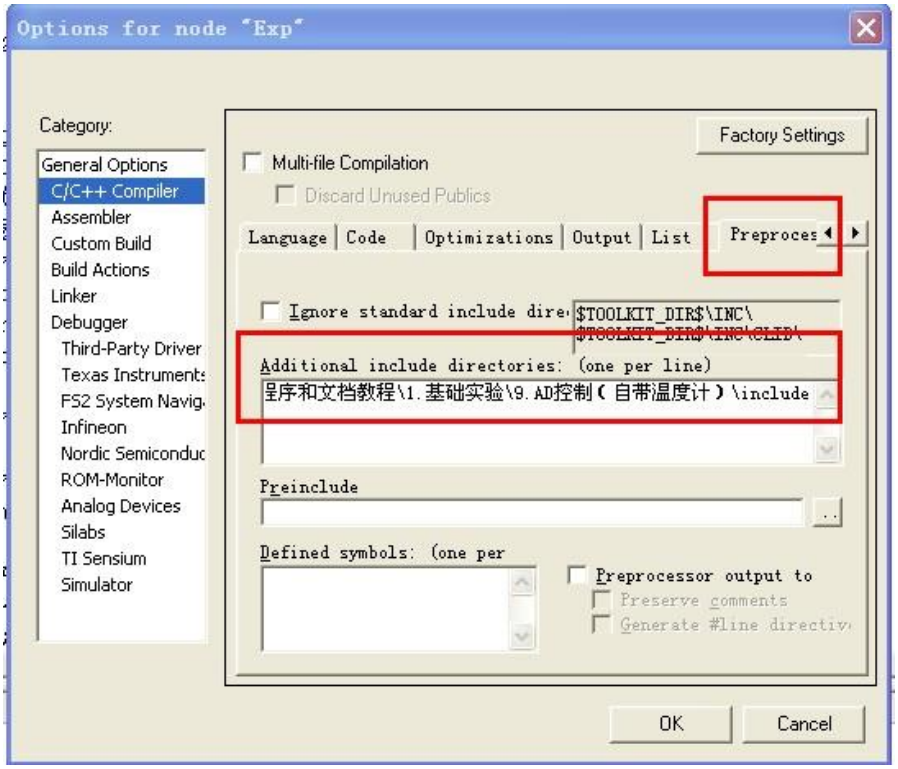


图 4-16

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：

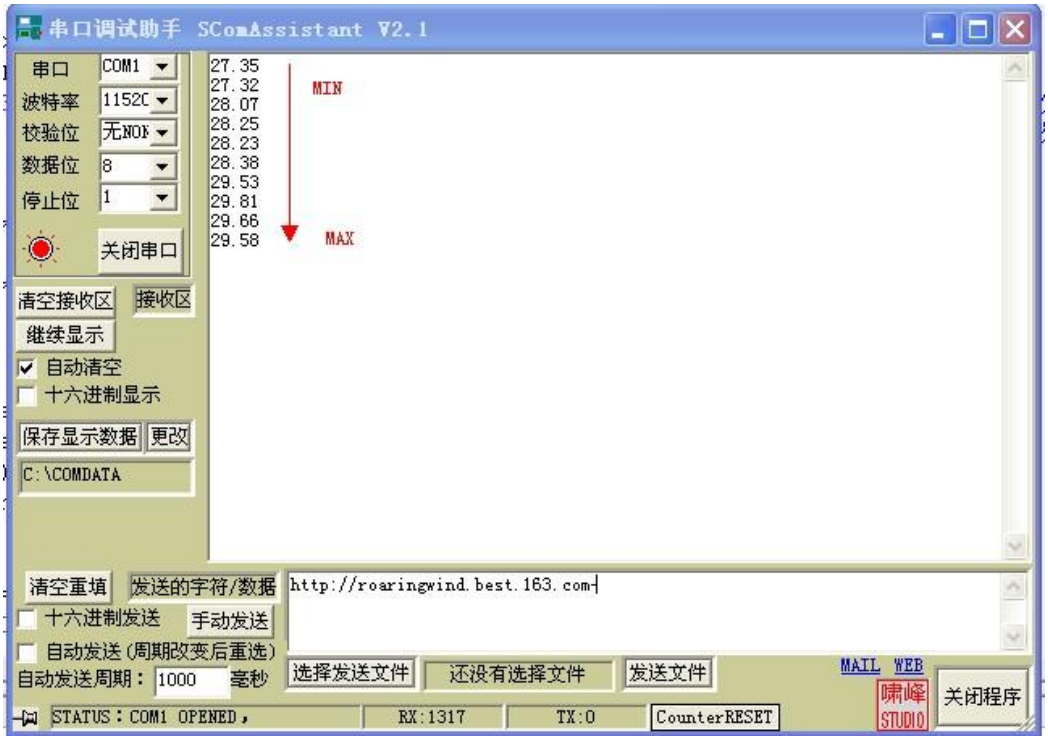


图 4- 16 内部温度检测

4.9. 实验八：看门狗

前言：

看门狗电路的应用,使单片机可以在无人状态下实现连续工作。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。

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 6

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：

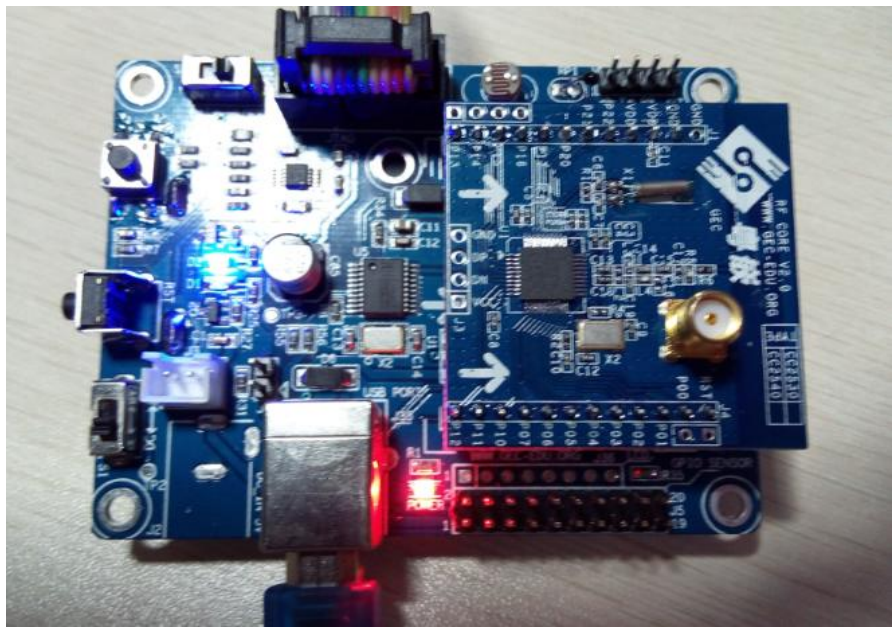


图 4- 20

4.10. 实验九：SensorDemo 实验演示—ZigBee 组网初接触

前言：

本实验程序基于 TI - ZStack 的 SensorDemo，增添了 UART 命令控制功能，扩展了远程 LED 控制功能，以及增加了一些传感器，修改了 MT 串口协议，让其更加合理。

实验设备： 1 个 ZigBee 协调器，两个 ZigBee 传感节点，仿真器，串口线；

实验现象：

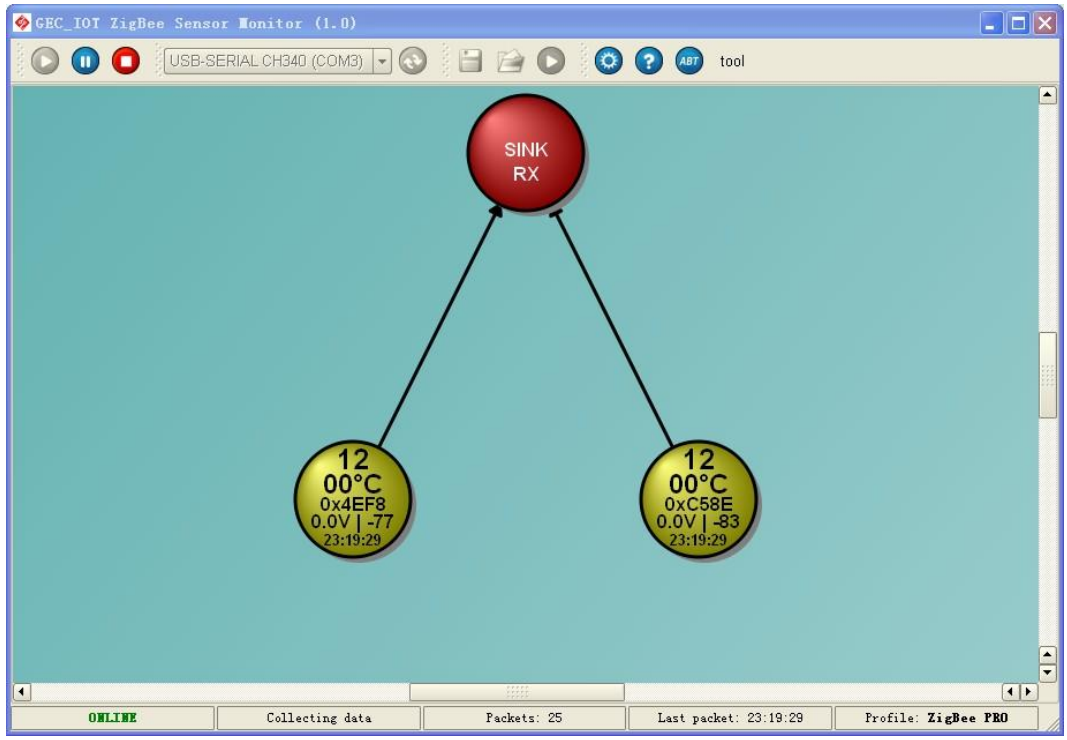


图 4-21　PC 上位机

上位机软件： 

，需安装

实验步骤：

打开工程文件夹\SensorDemo 实验\SensorDemo\Projects\zstack\Samples\SensorBB\CC2530DB 里面的 IAR 工程文件 SensorDemo.eww ， 如下图所示：

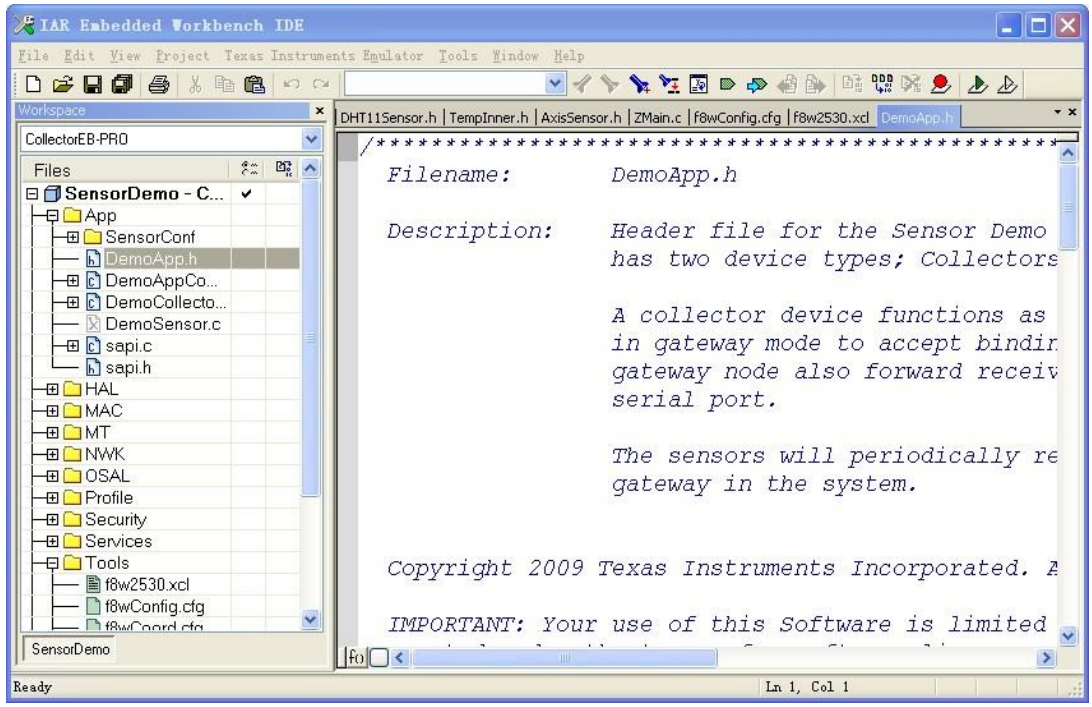


图 4-22 IAR 工程文件

为了实现同一个实验室里面的每组设备不会相互干扰，我们每组设备需要先修改自己的 PANID 号，来区分不同的网络。打开 TOOLS 导航的 f8wConfig.cfg 文件，修改自己的编号。建议由 0x0000 到 0xFFF0 范围内。

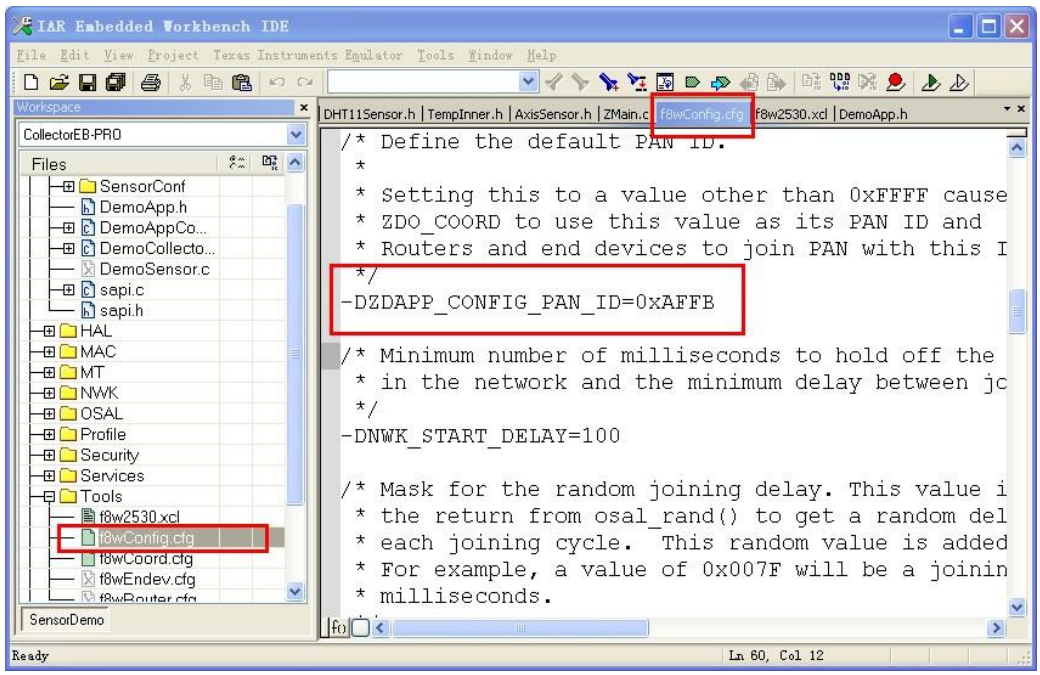


图 4-23 PANID 的修改

修改后，我们先选择协调器的工程文件。连接仿真器到带串口的开发板，

这个作为主机，然后点击下载按钮。

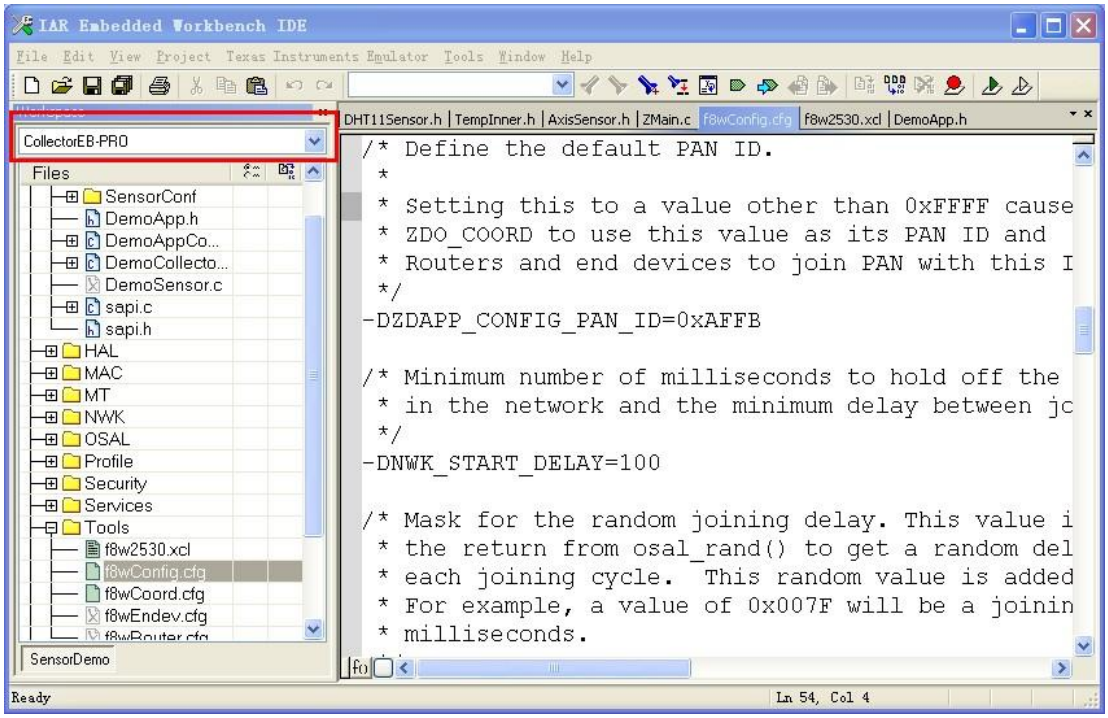


图 4-24 选择 CollectorEB-PRO

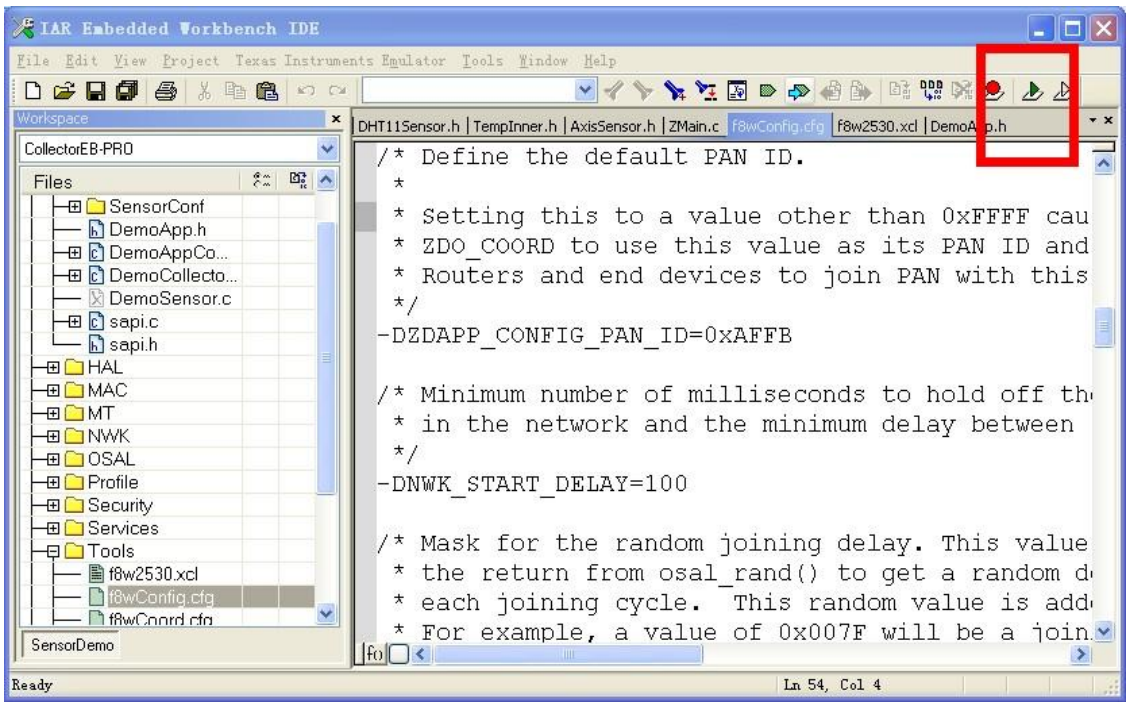


图 4-26 点击下载按钮烧写程序

使用同样的方法将 SensorBB-PRO 烧写到 ZigBee 电池板节点中。

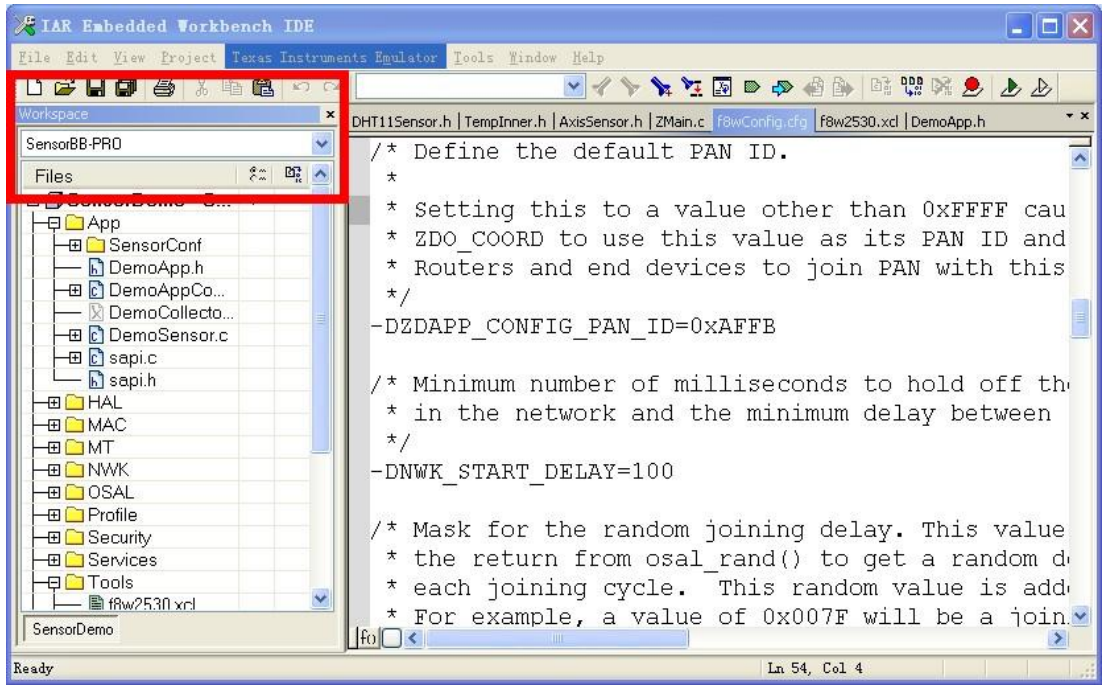


图 4-27 ZigBee 传感节点对应 SensorBB-PRO

将 ZigBee 协调器的 UART 连接到 PC,启动 GEC Sensor Monitor(PC 上位机),如下图所示:

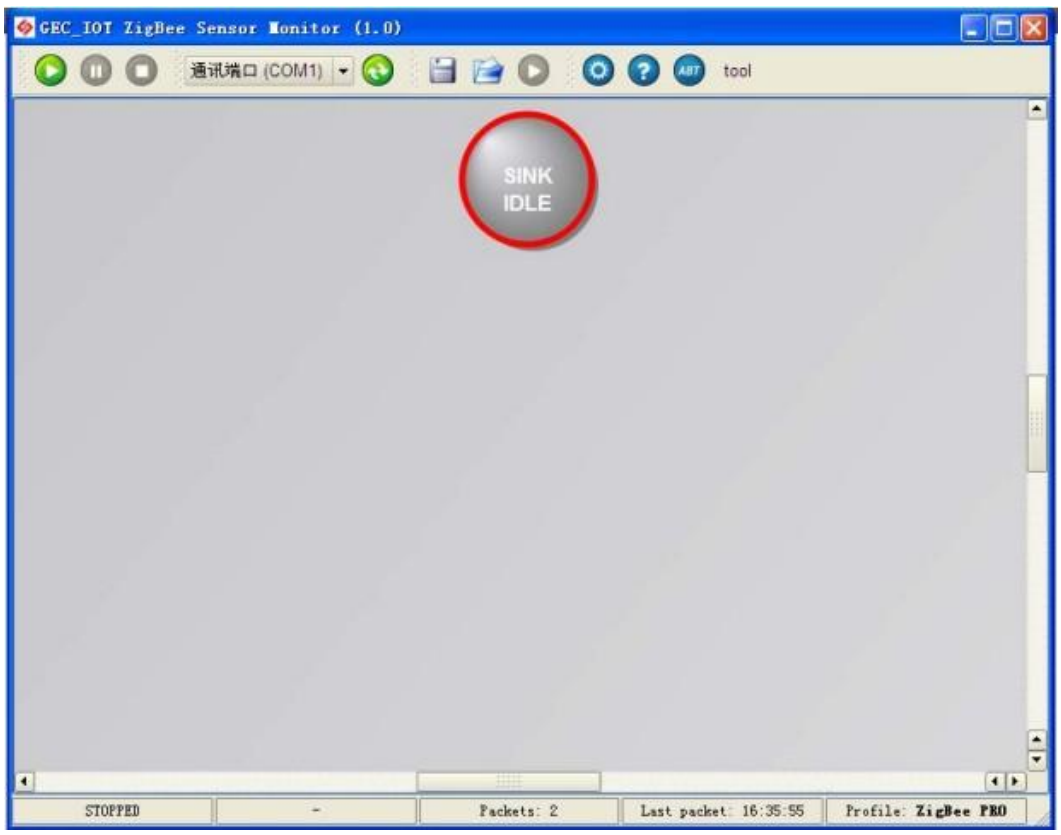


图 4-28

选择好正确的 COM，打开协调器的电源，单击工具栏的绿色播放按钮 ，此

时协调器节点将会变成深红色。

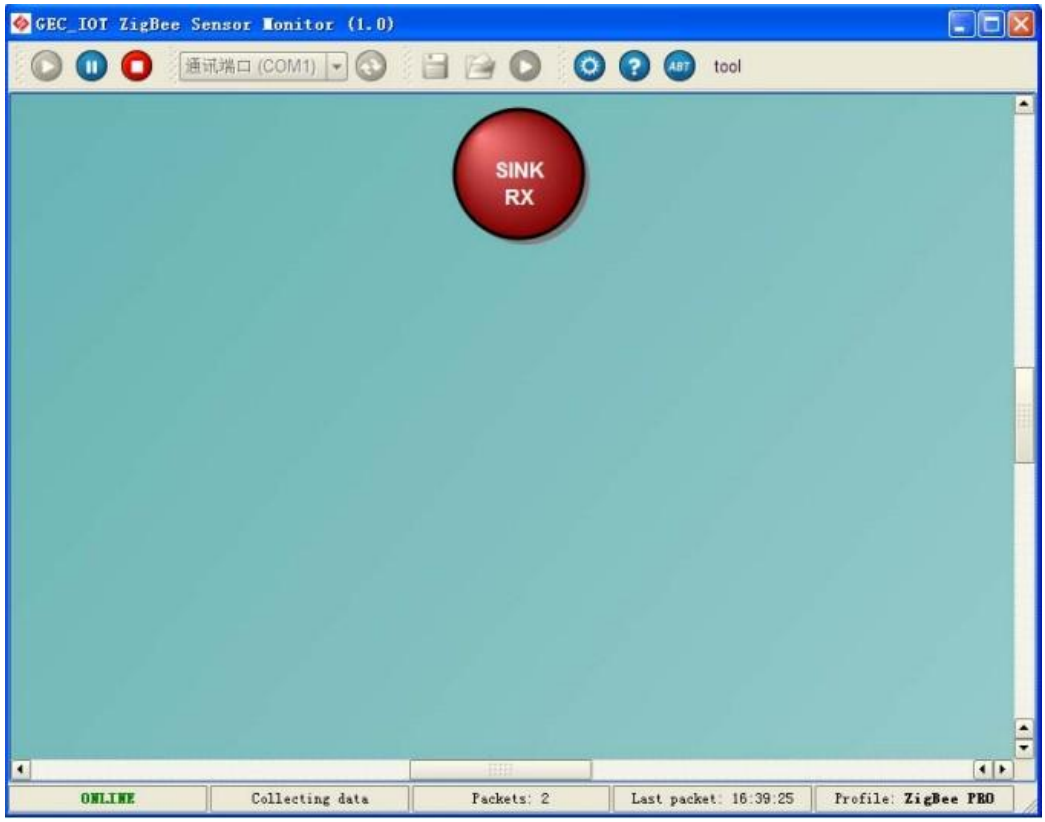


图 4-29 点击左上角绿色播放按钮后

初始化 ZigBee 模块：单击工具栏的 tool 按钮

图 4-30

单击模块控制的初始化按钮，对设备进行初始化，。

允许组网：单击模块控制下的单选按钮“允许组网”，此时 Sensor 节点可以

绑定到 Collector 节点。

打开 SensorBB 传感器节点电源，此时你会发现 SensorBB 会自动加入 ZigBee 网络，如下图所示：如果无法加入，请尝试按下节点的 S3 按键。

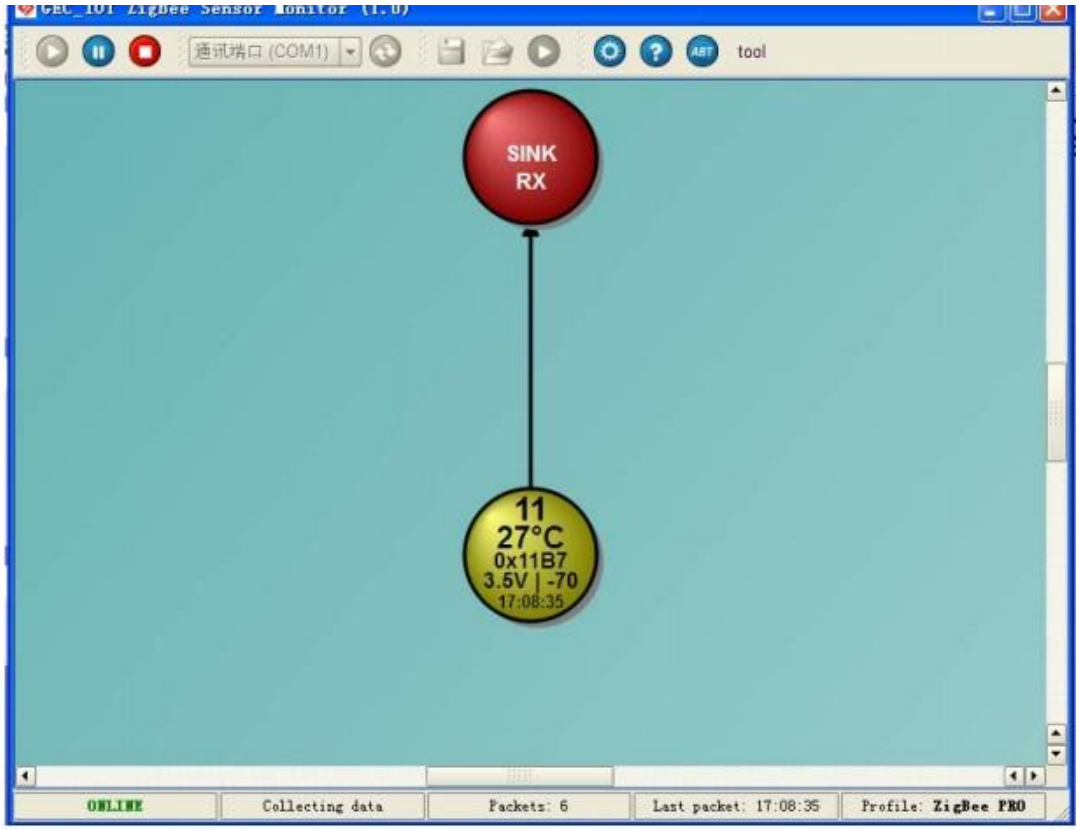


图 4-31 节点加入网络

此时，打开 tool：

图 4-32

远程开关灯：在远程控制下，选择一个 SensorBB 节点，然后单击单选按钮“灯开关”，此时被选中的 SensorBB 节点的 LED1 状态将会改变。

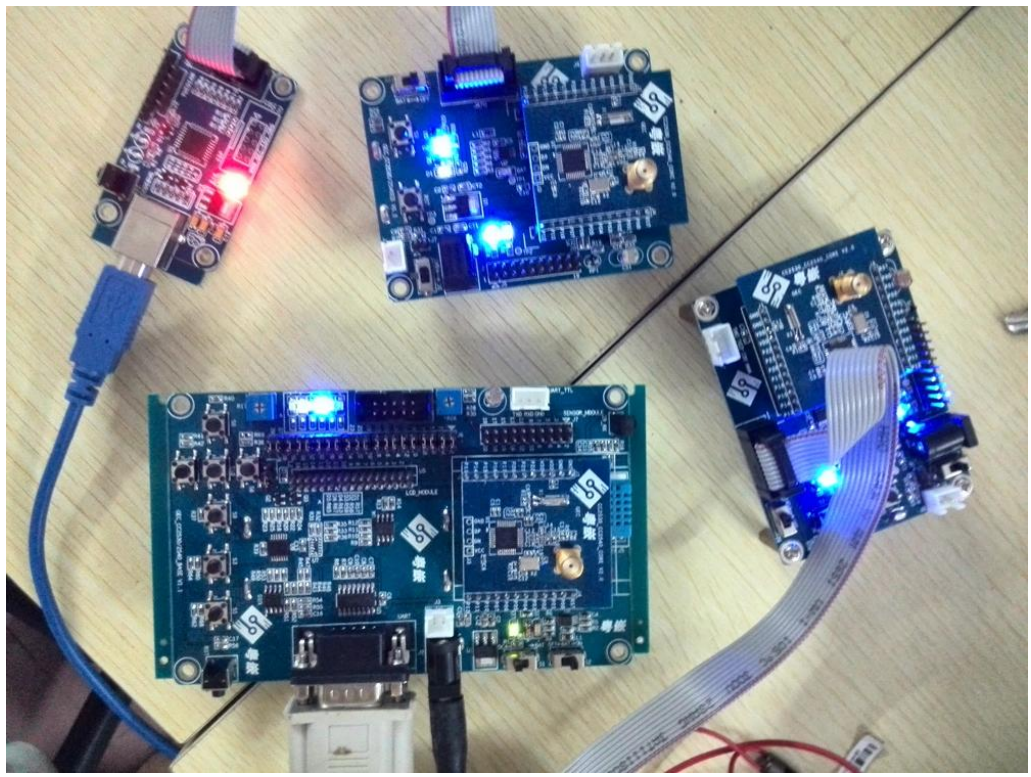


图 4-33 节点组网

第五章 ZigBee 组网实验

（一）本章教学目标：

ZigBee 组网实验，让学生熟悉 CC2530 芯片及其节点板的结构及其接口，完成 ZigBee 组网实验。

（二）教学目的与要求：

进一步熟悉 CC2530 芯片架构及其结构原理，完成基于 CC2530 芯片的 ZigBee 组网实验项目，并基本掌握 CC2530 系统原理及开发方法。

（三）教学重点与难点：

重点与难点为 CC2530 结构原理及接口，IAR 工程的创建于设置，掌握终端节点、协调器节点和路由器节点的设置方法和无线组网的参数设置以及程序设计调试。

5.1. 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 5-1 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。

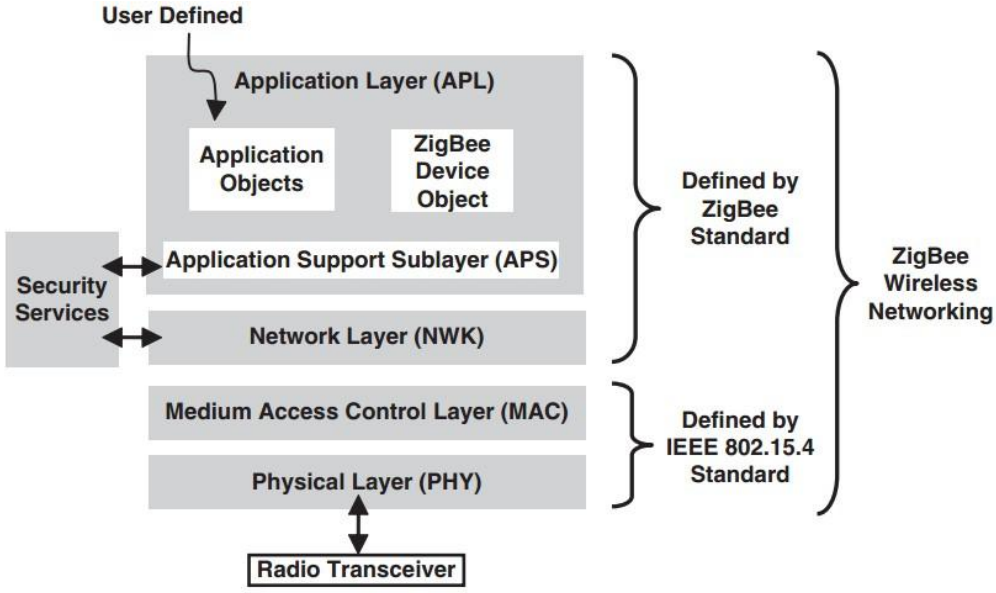


图 5- 1 ZigBee 无线网络协议层

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 5-2 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。

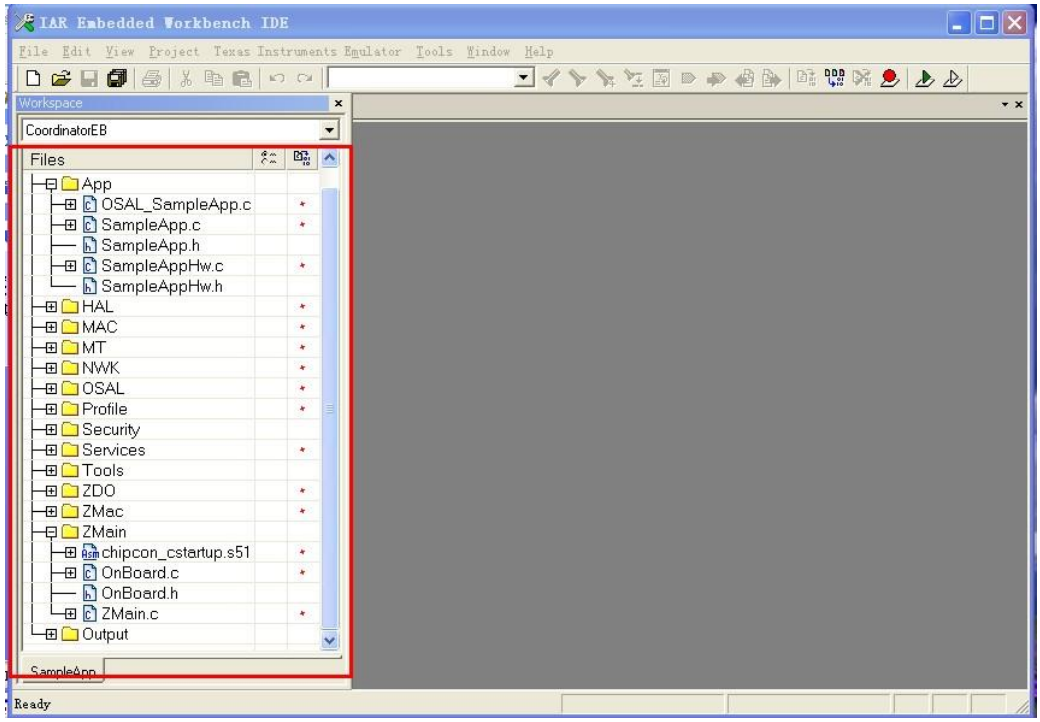


图 5- 2 TI Z-stack TI Z-Stack-CC2530-2.3.0

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。

5.2. 实验一：无线组网点灯实验

前言：

无线点灯是大家入门 ZigBee 无线技术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（大家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。

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 zigbee 传感器节点

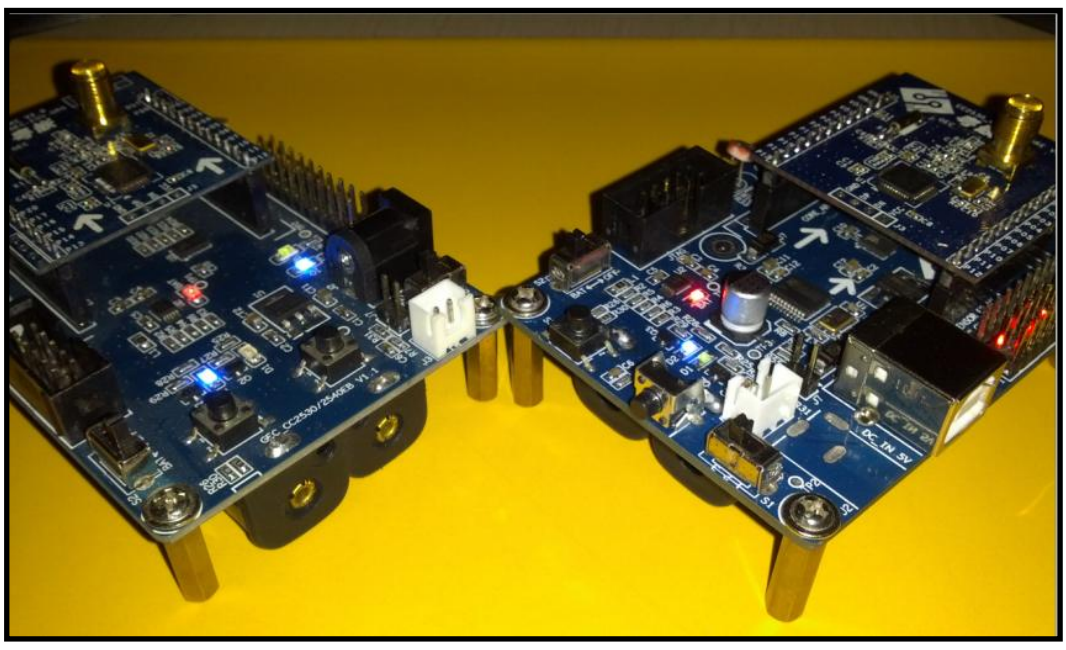


图 5- 3 实验操作平台

实验现象： 两块 ZigBee 传感器节点，一个节点模块作发射，另外一个节点模块接收，发射模块依次按下按键 S3，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 ZigBee 传感器节点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

1）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：

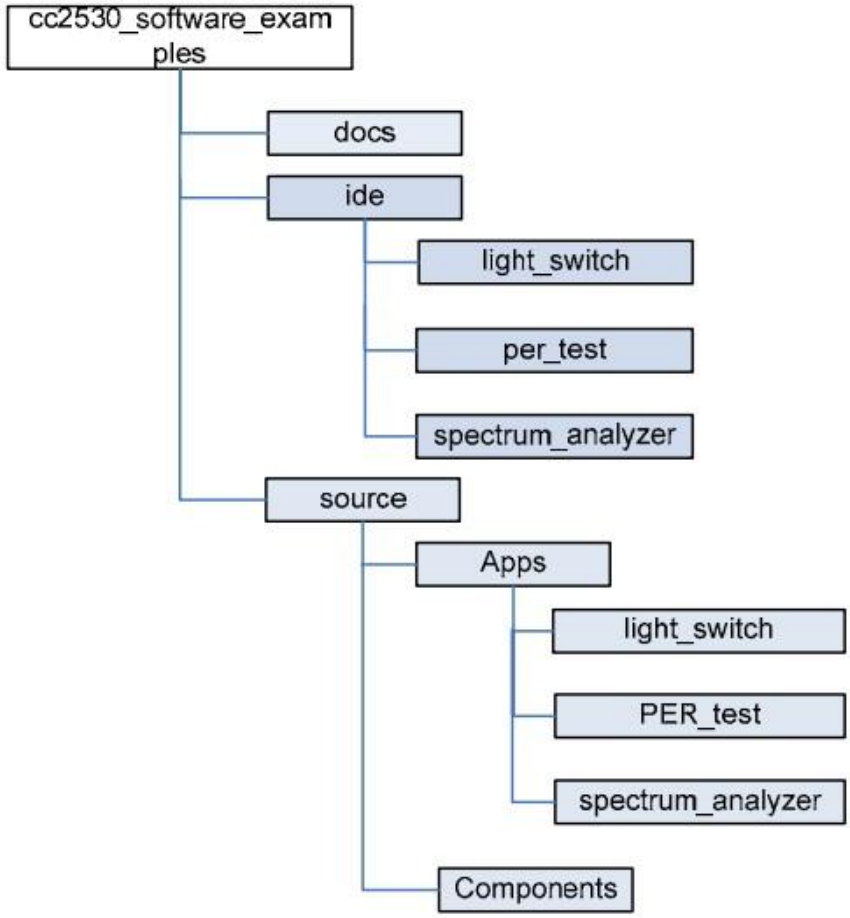


图 5- 4 BasicRF 软件文件夹架构

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：

打开工程文件夹 **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

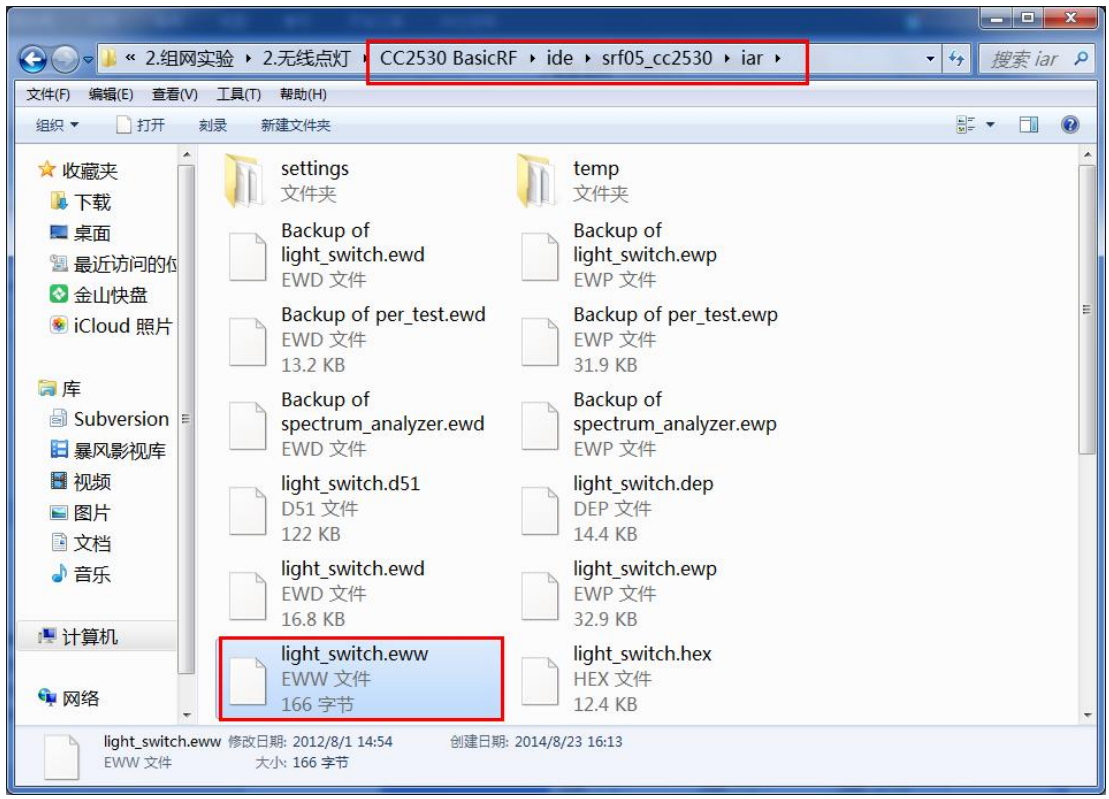


图 5- 5 BasicRF 工程路径

2、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

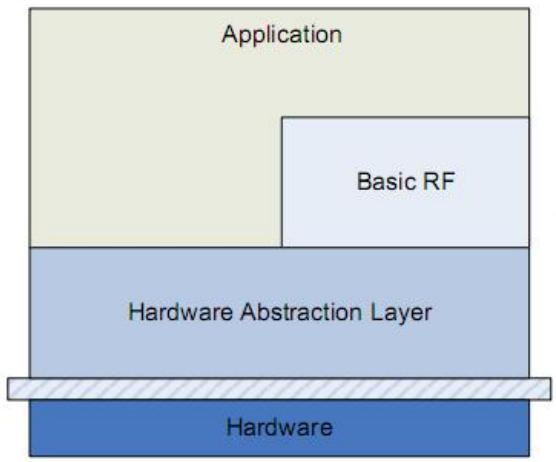


图 5- 6

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。

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。

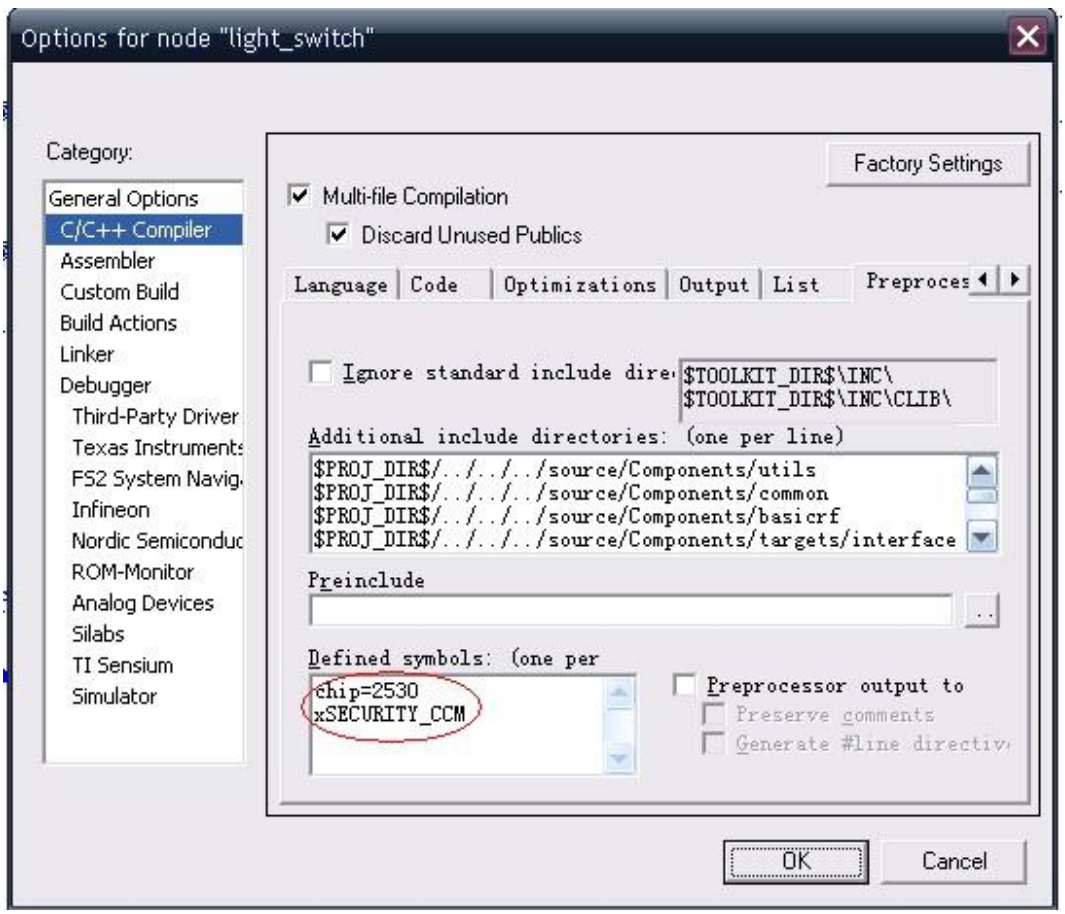


图 5- 7 注释 SECURITY_CCM

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

1. 创建一个 buffer，把 payload 放入其中。Payload 最大为 103 个字节
2. 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。

3、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.   /***********根据传感节点硬件配置**********/
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 S3        P1_2
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：关闭传感节点板的 LED2，开 LED1。由于硬件设计的 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：传感节点板暂时还没有 joystick（多方向按键），此处可忽略。

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 S3        P1_2
// 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************/
//appSwitch();          //节点为按键 S3        P1_2
 appLight();           //节点为指示灯 LED1    P1_0
```

完成烧写后上电，按下发射模块的 S3 按键，可以看到接收模块的 LED1 被点亮。

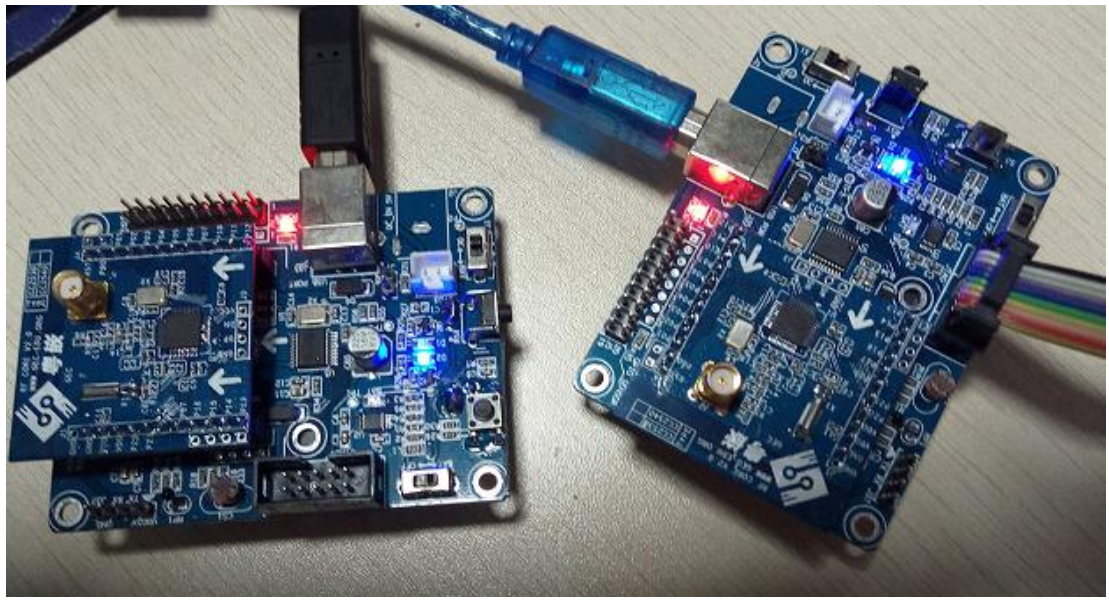


图 5- 8 无线点灯

5.3.实验二：协议栈工作原理

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。现在，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询

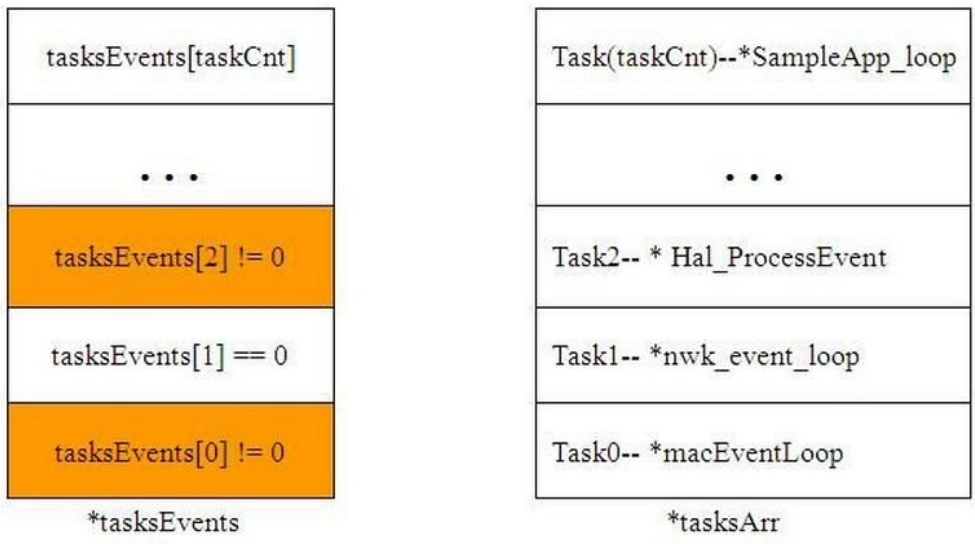


图 5- 9 任务轮询

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：

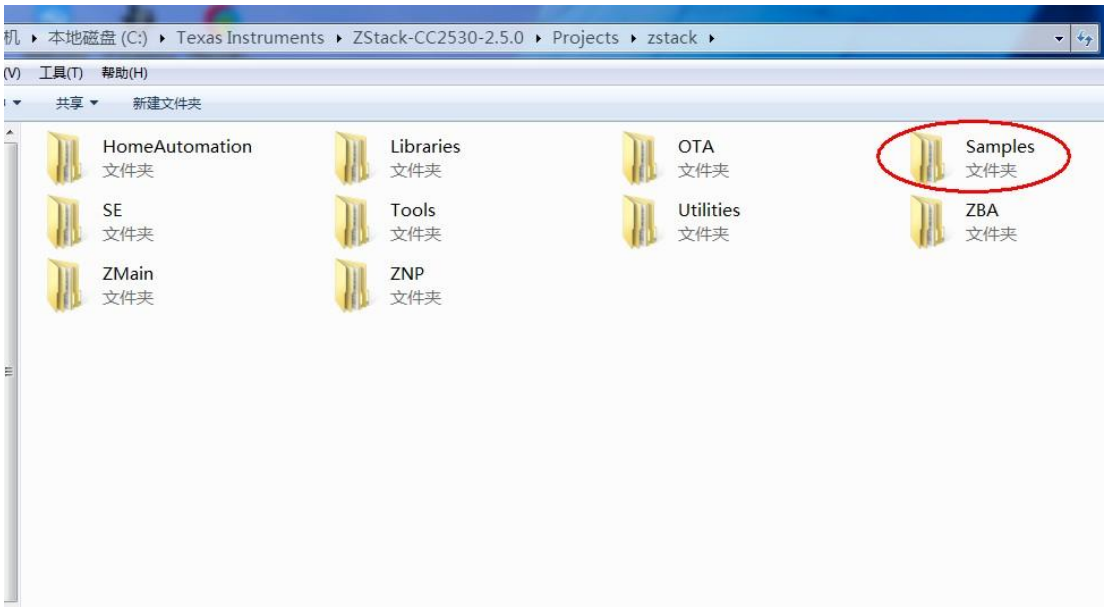


图 5- 10

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。

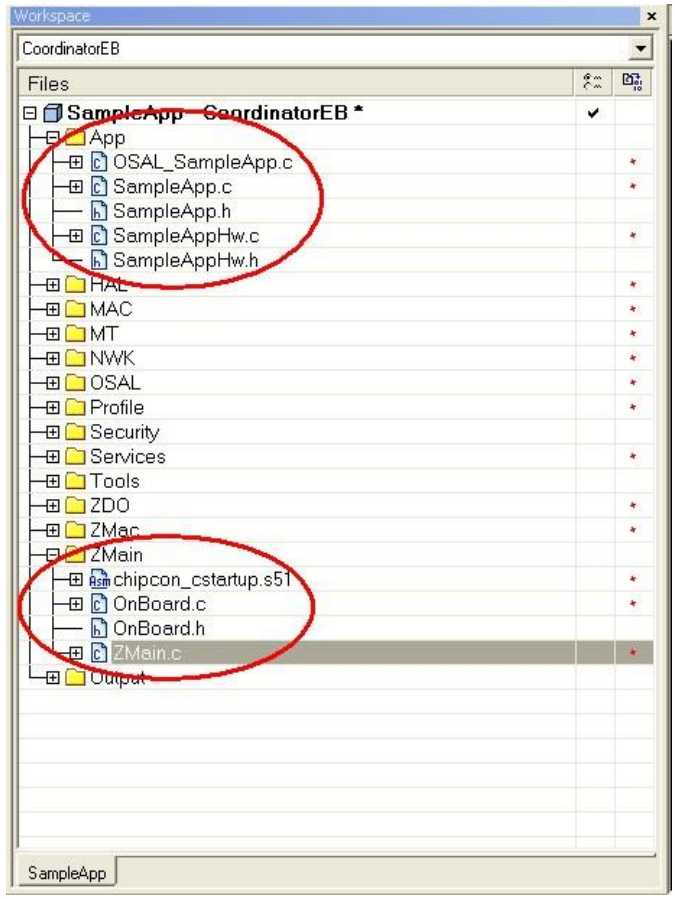


图 5- 11 工作空间目录

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：

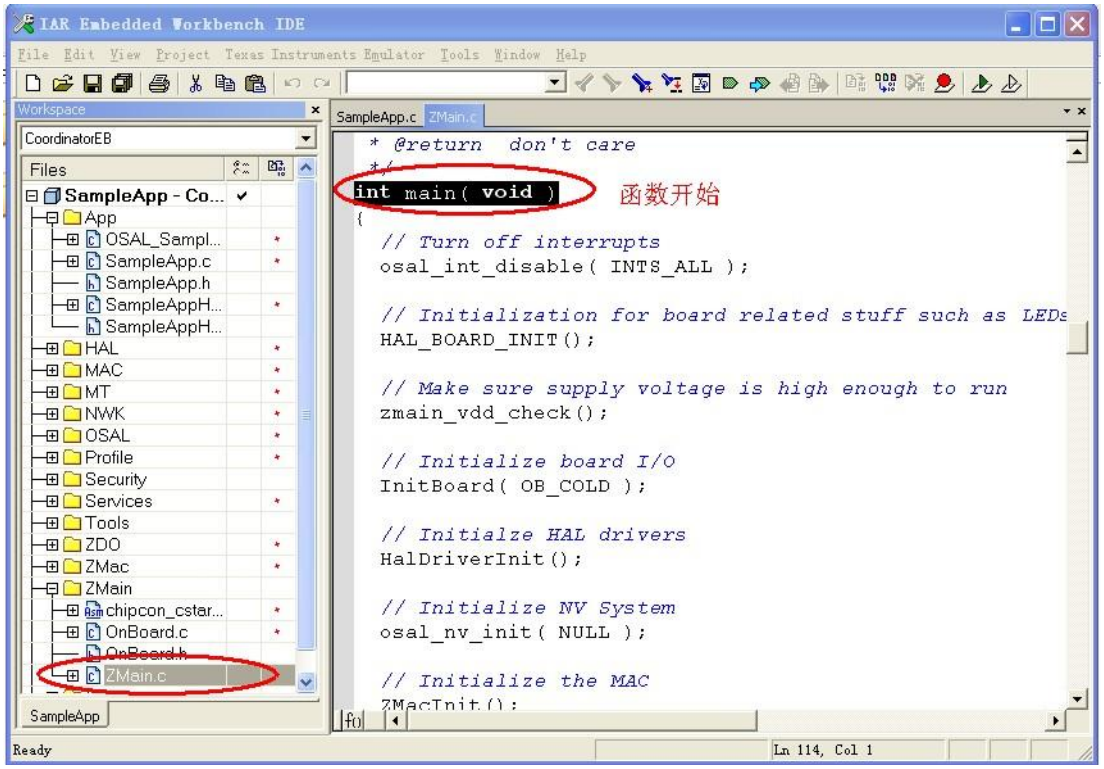


图 5- 12 协议栈主函数

```
/*************************************************************************
 * @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：

a) 初始化操作系统

```
osal_init_system();
```

b) 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。

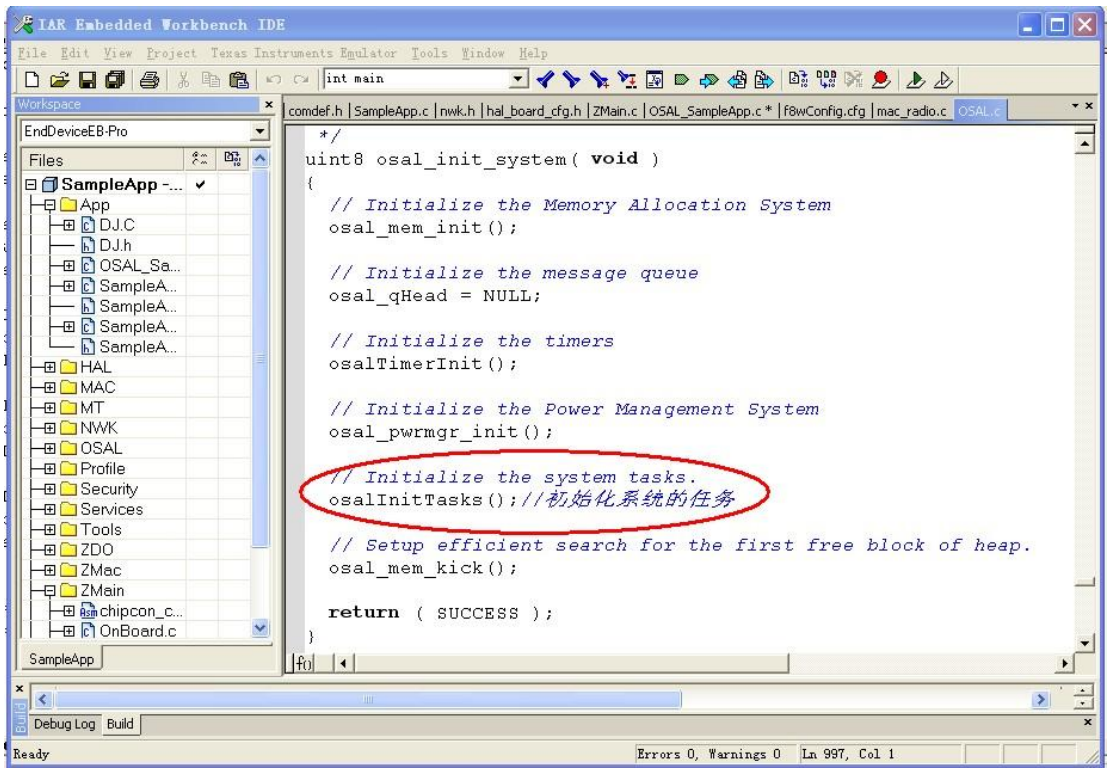


图 5- 13

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。

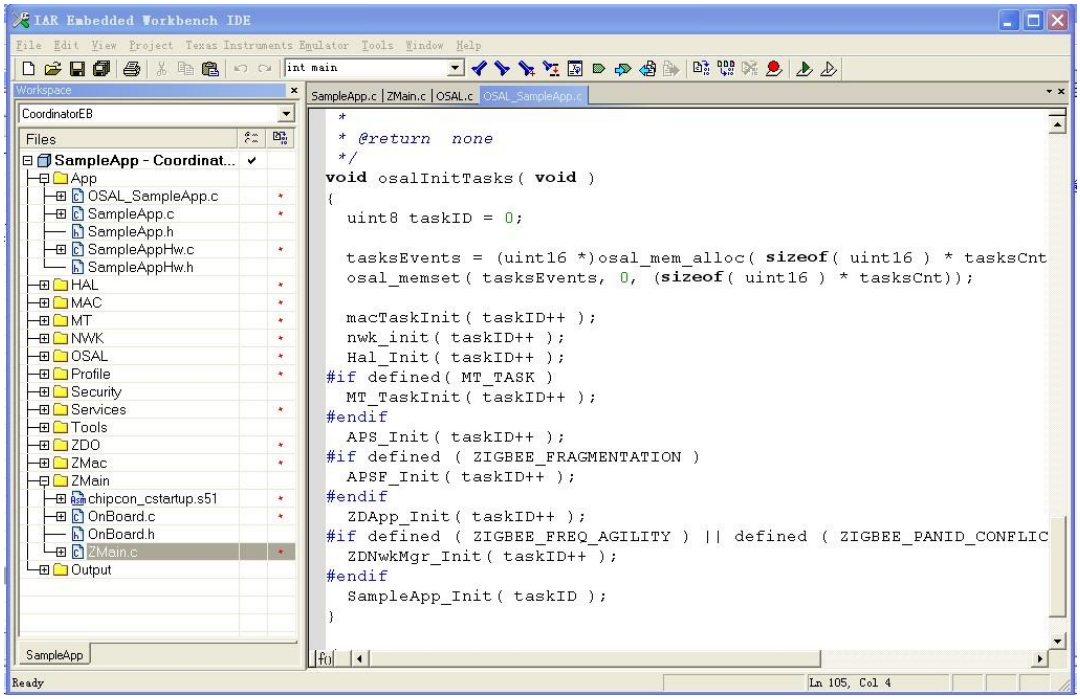


图 5- 14

```
  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：

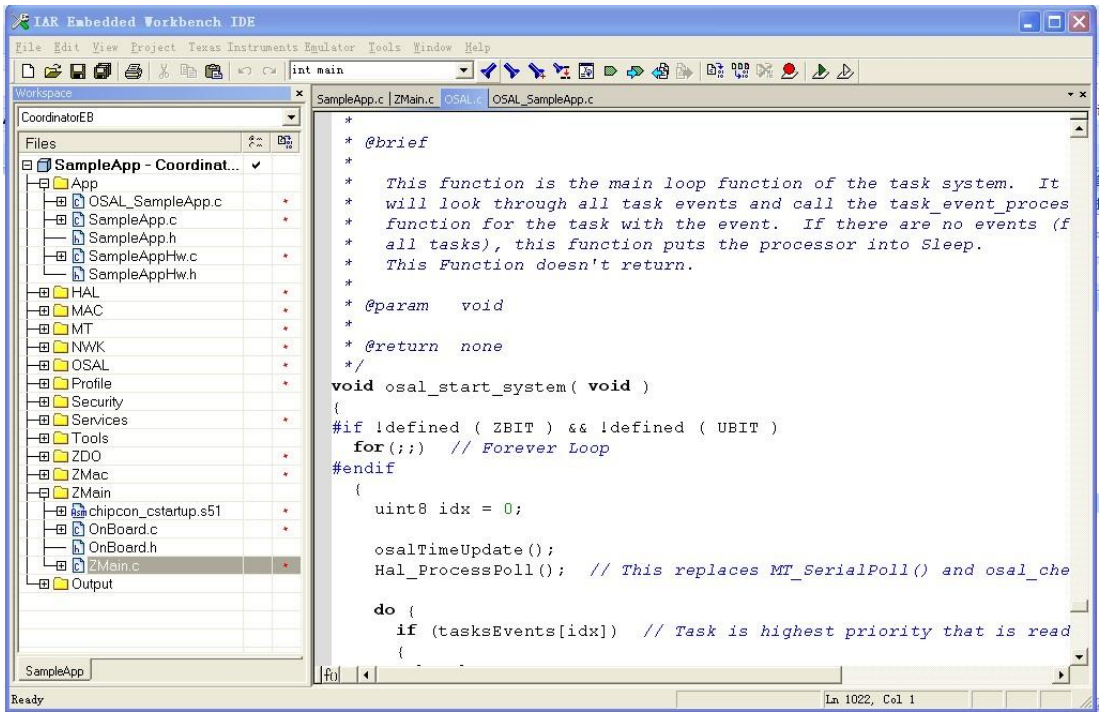


图 5- 15 osal_start_system();函数

```
/*************************************************************************
 * @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。

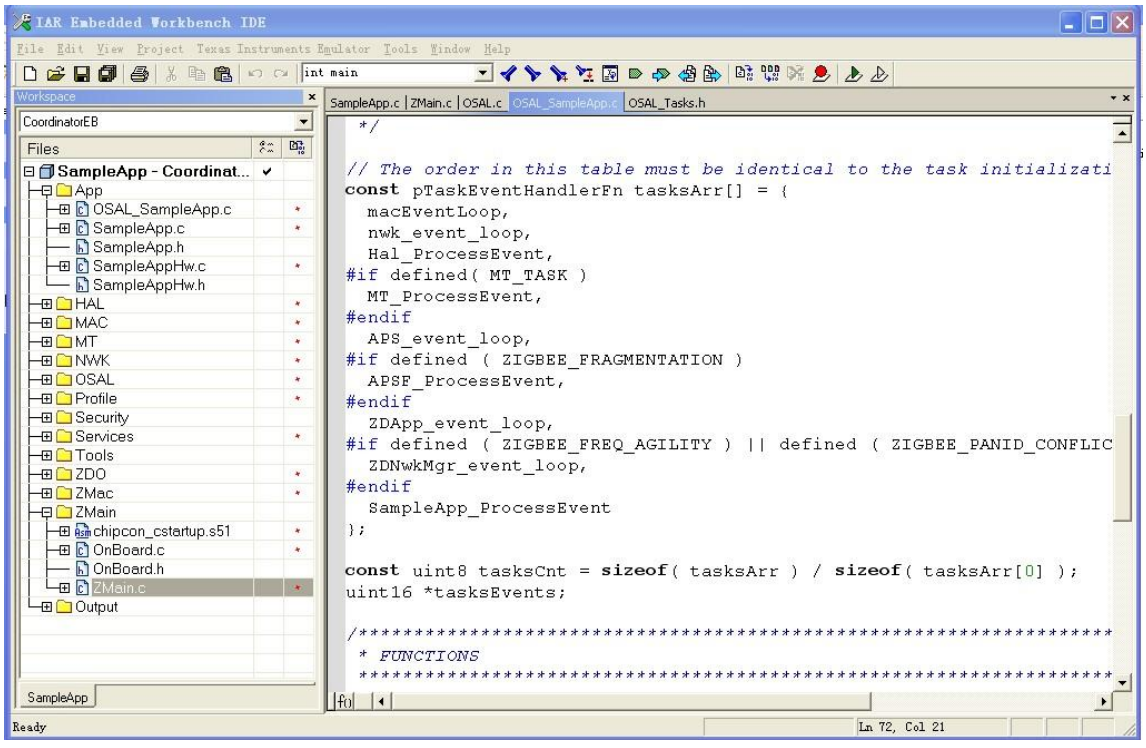
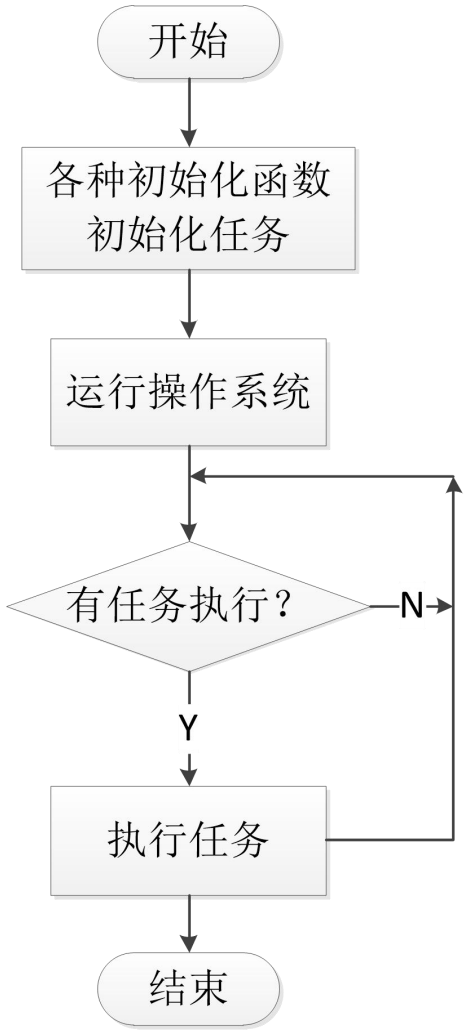


图 5- 16

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 5- 17 协议栈工作流程

5.4. 实验三：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：ZigBee 协调器或者 ZigBee 节点模块。

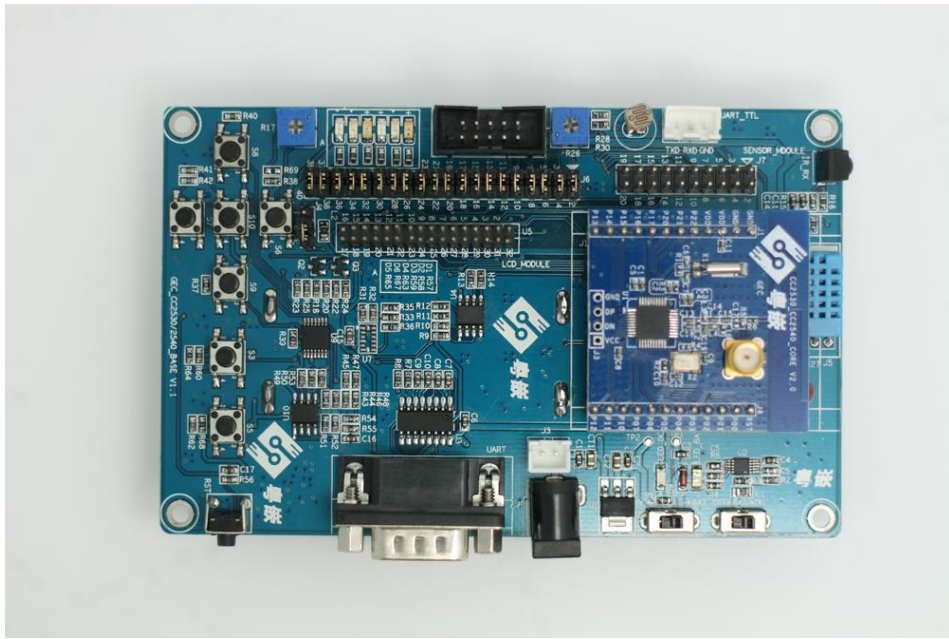


图 5- 18　ZigBee 协调器

实验现象：协调器通过串口发送“HELLO　WORLD！”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

和步骤如下：

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。

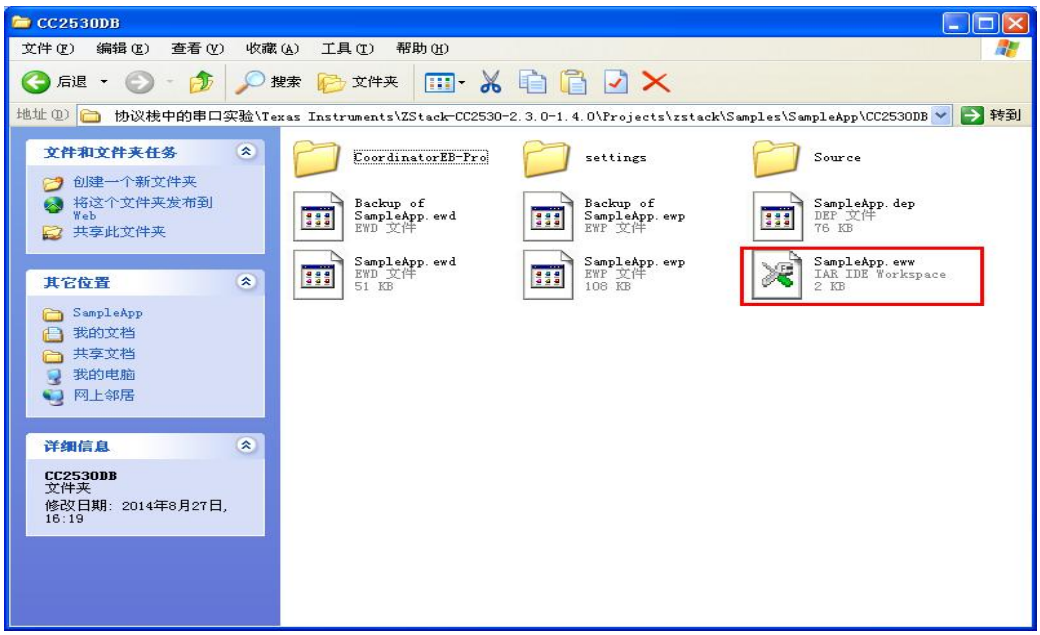


图 5- 19 SampleApp.eww 工程

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 5-20。

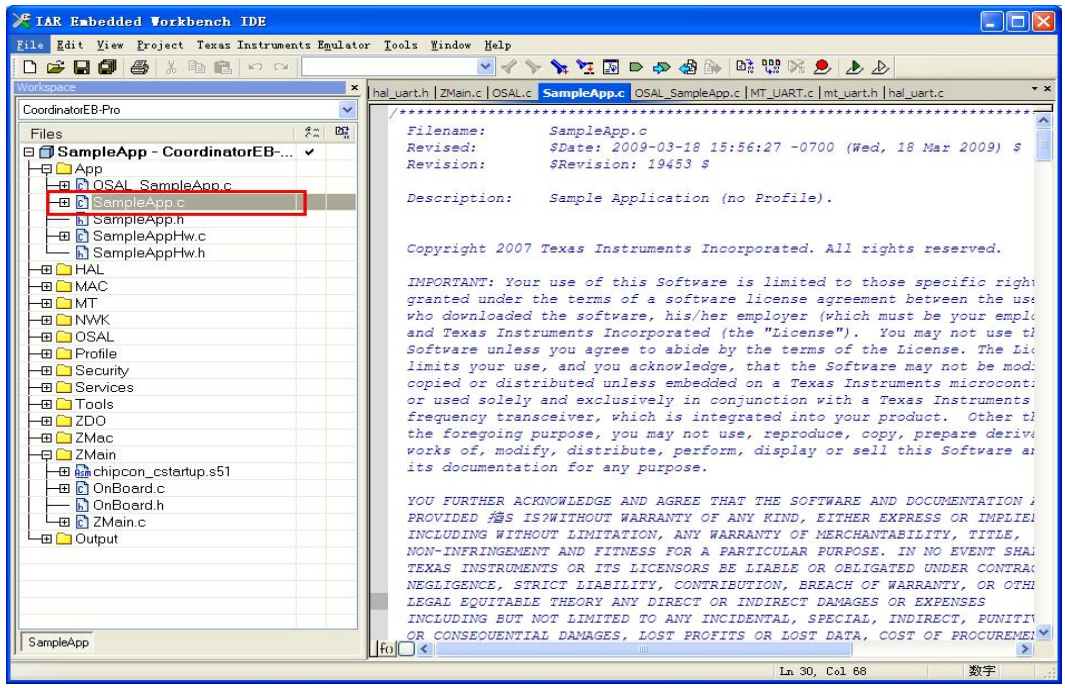


图 5- 20　SampleApp.eww 工程主要文件

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-Pro\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？

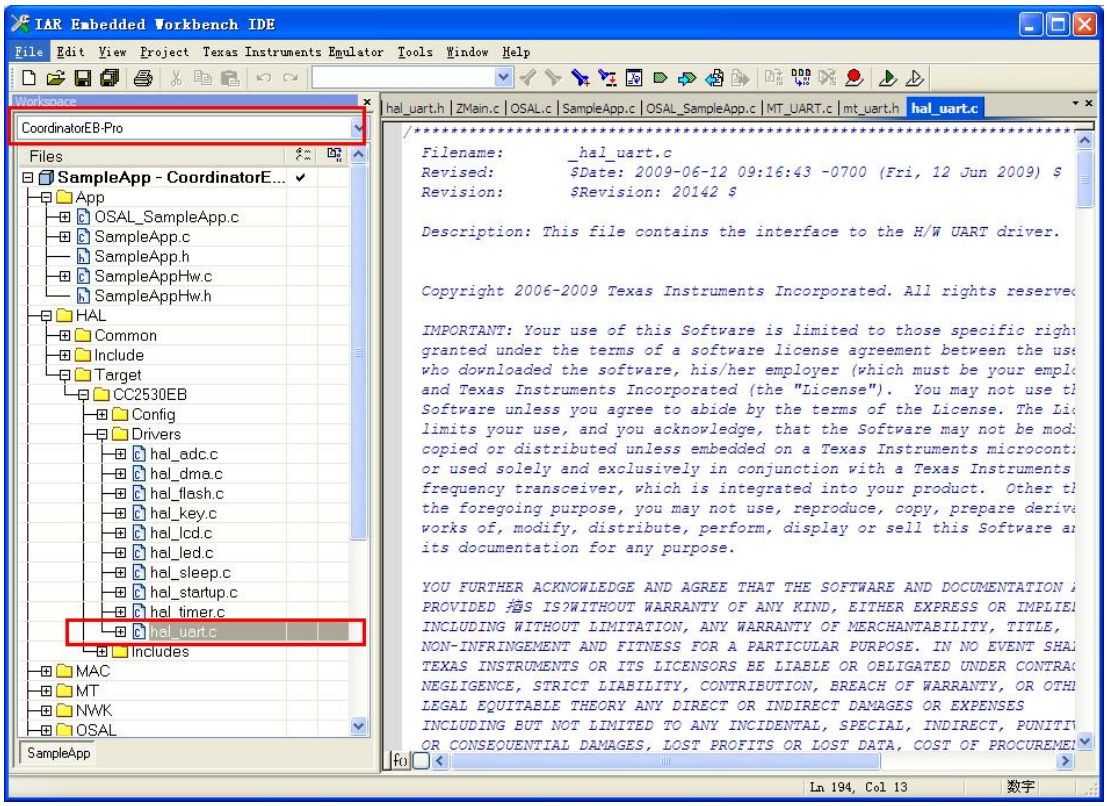


图 5- 21

浏览一下关于串口的操作函数还是挺全的，我们再看看 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。

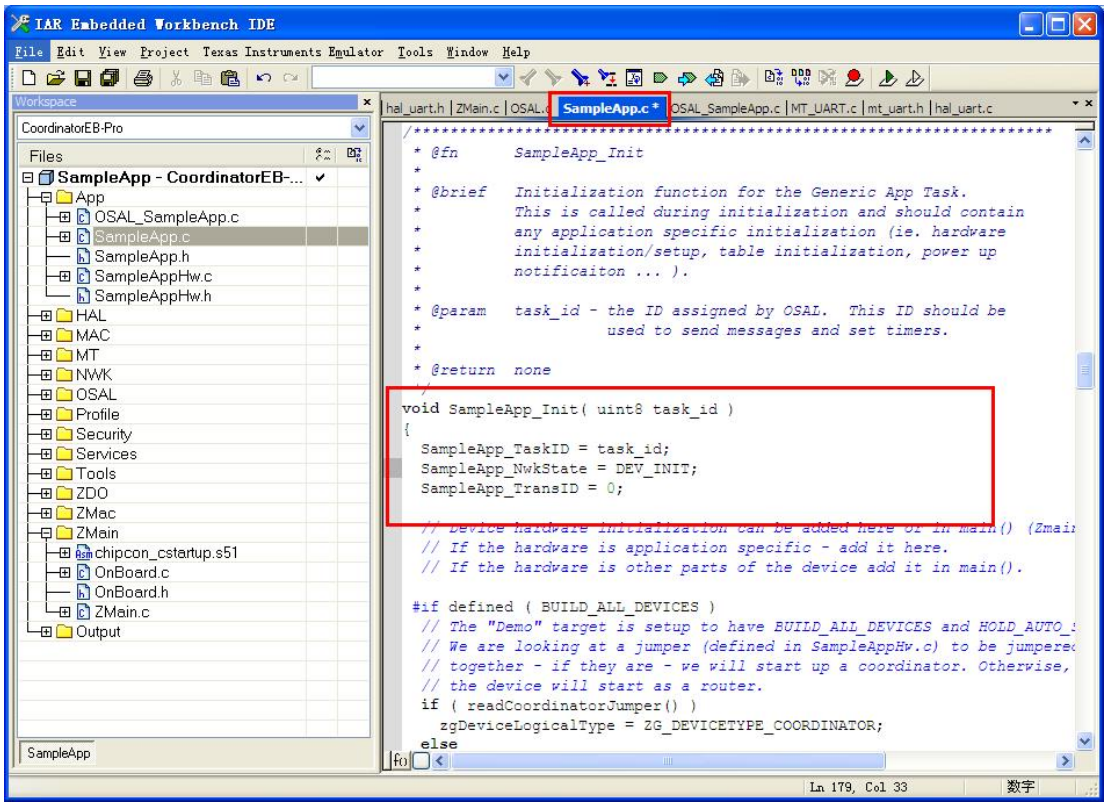


图 5- 22

我们在函数第四行加入语句：MT_UartInit();如图 5- 23 所示：

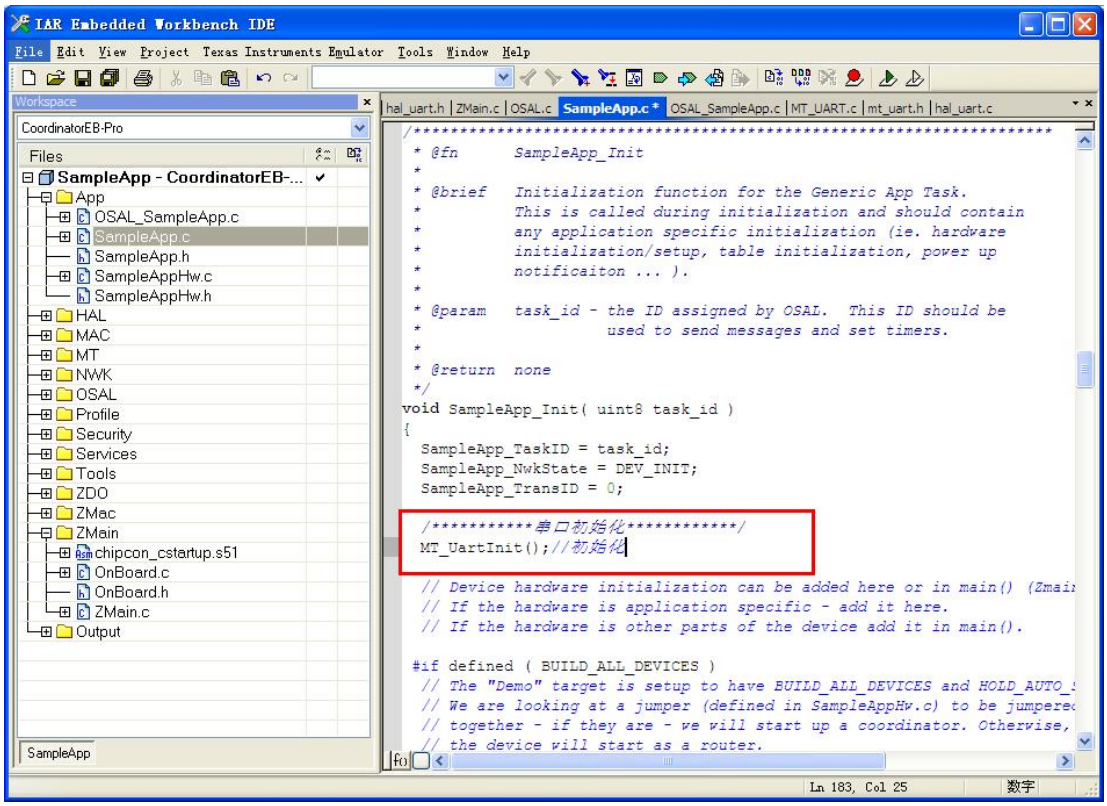


图 5- 23

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. #if defined (ZTOOL_P1) || defined (ZTOOL_P2)
16. uartConfig.callBackFunc          = MT_UartProcessZToolData;
17. #elif defined (ZAPP_P1) || defined (ZAPP_P2)
18. uartConfig.callBackFunc          = MT_UartProcessZAppData;
19. #else
20. uartConfig.callBackFunc          = NULL;
21. #endif

22. /* Start UART */
23. #if defined (MT_UART_DEFAULT_PORT)
24. HalUARTOpen (MT_UART_DEFAULT_PORT, &uartConfig);
```

```
25. #else
26. /* Silence IAR compiler warning */
27. (void)uartConfig;
28. #endif

29. /* Initialize for Zapp */
30. #if defined (ZAPP_P1) || defined (ZAPP_P2)
31. /* Default max bytes that ZAPP can take */
32. MT_UartMaxZAppBufLen  = 1;
33. MT_UartZAppRxStatus   = MT_UART_ZAPP_RX_READY;
34. #endif
35. }
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。

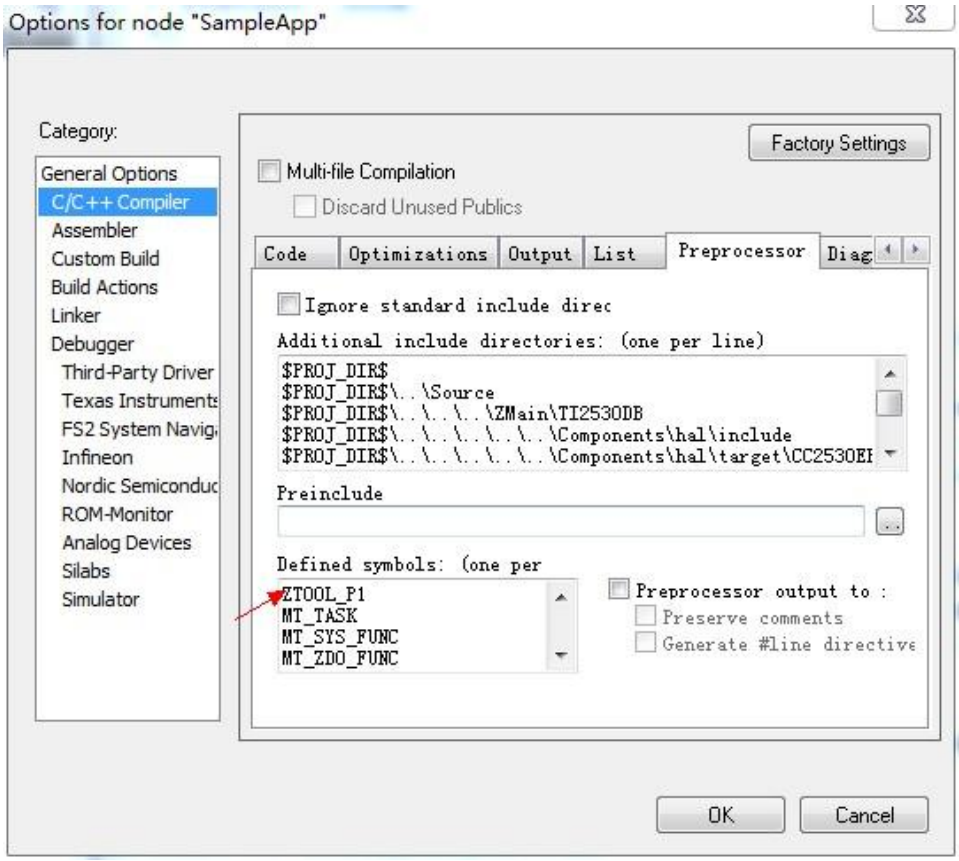


图 5- 24

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 5- 32 所示：

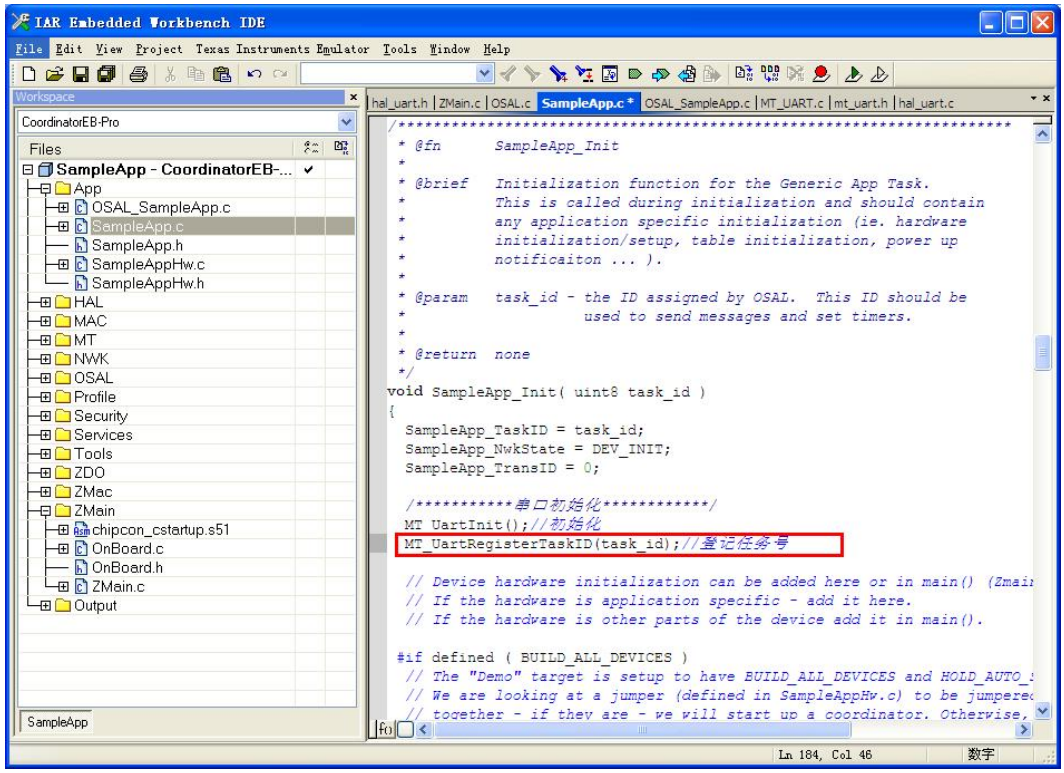


图 5- 25

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”
```

协调器连接仿真器和串口线,选择 CoordinatorEB-Pro,点击下载并调试。全速运行，可以看到串口助手收到信息。图 5- 26、图 5- 27。

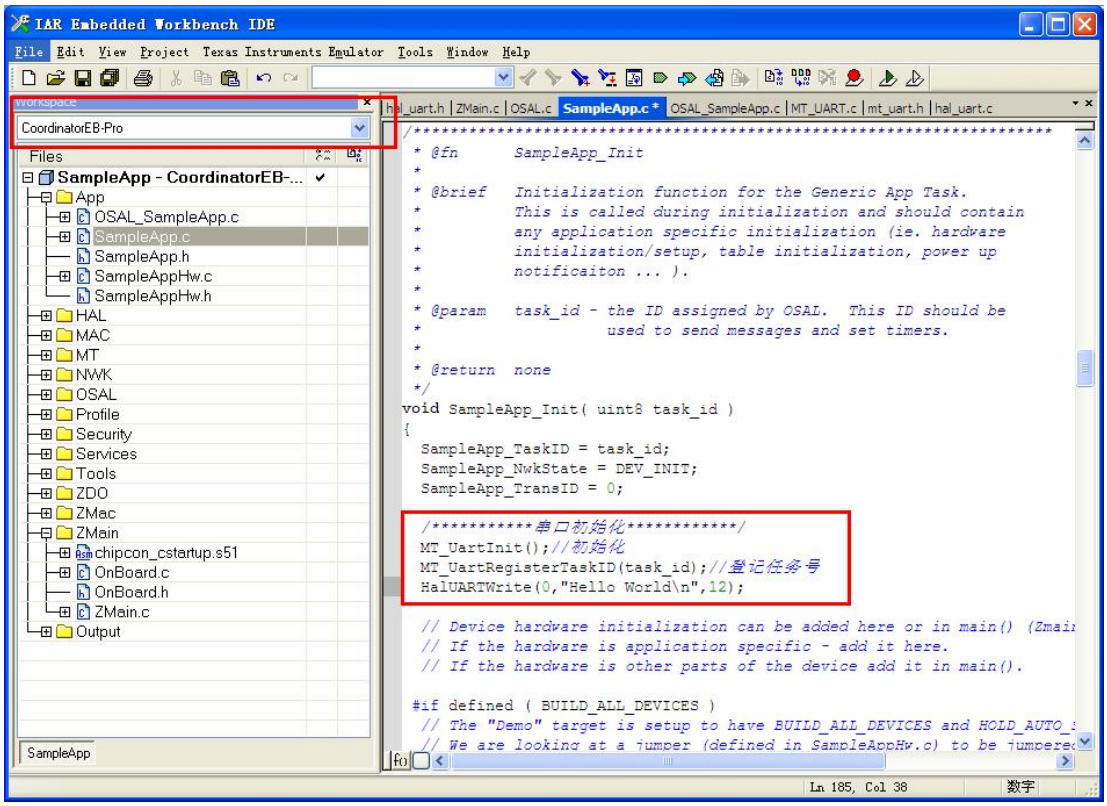


图 5- 26

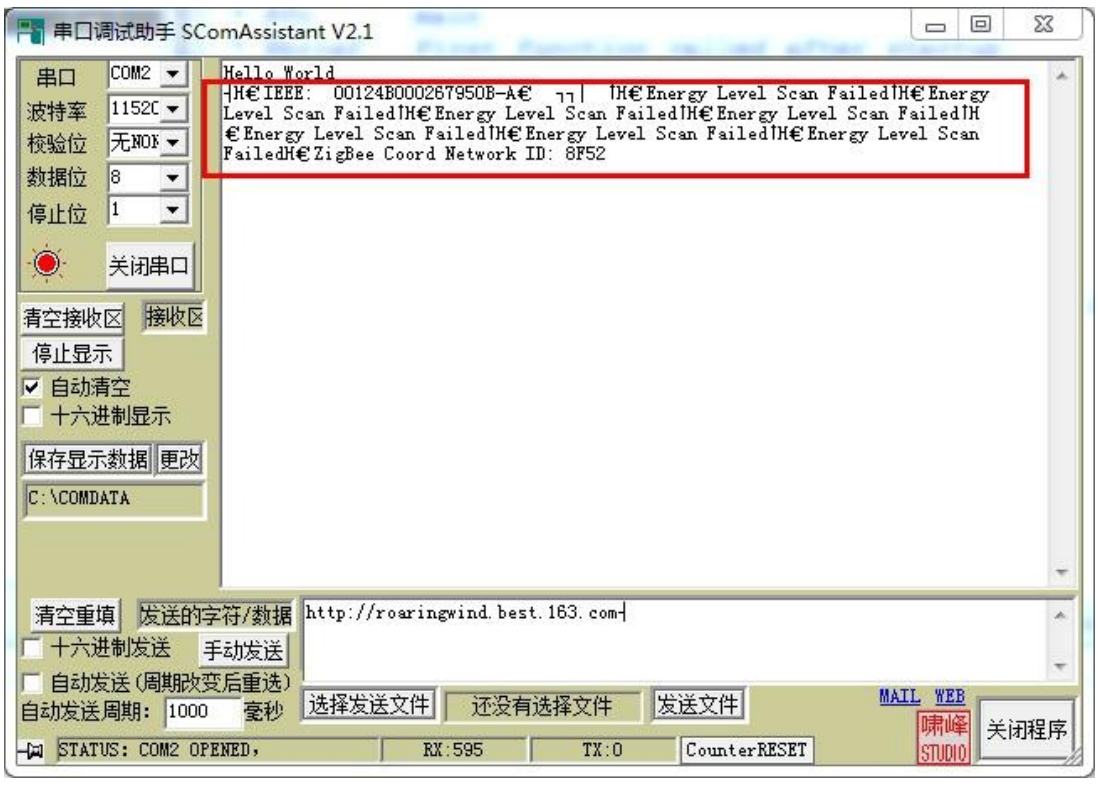


图 5- 27 Hello World 后面出现乱码

图 5-27 显示可以接收代码了，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。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，如图 5- 29,或许就是你想要的 Hello World。

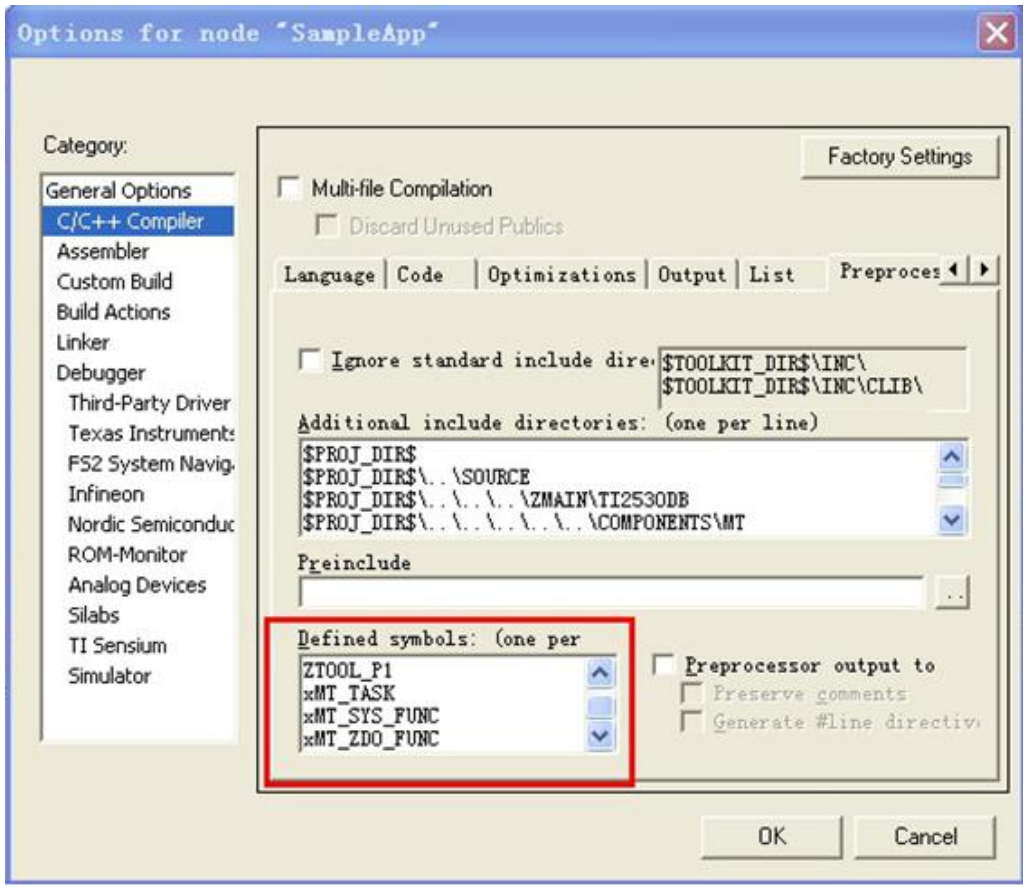


图 5- 28

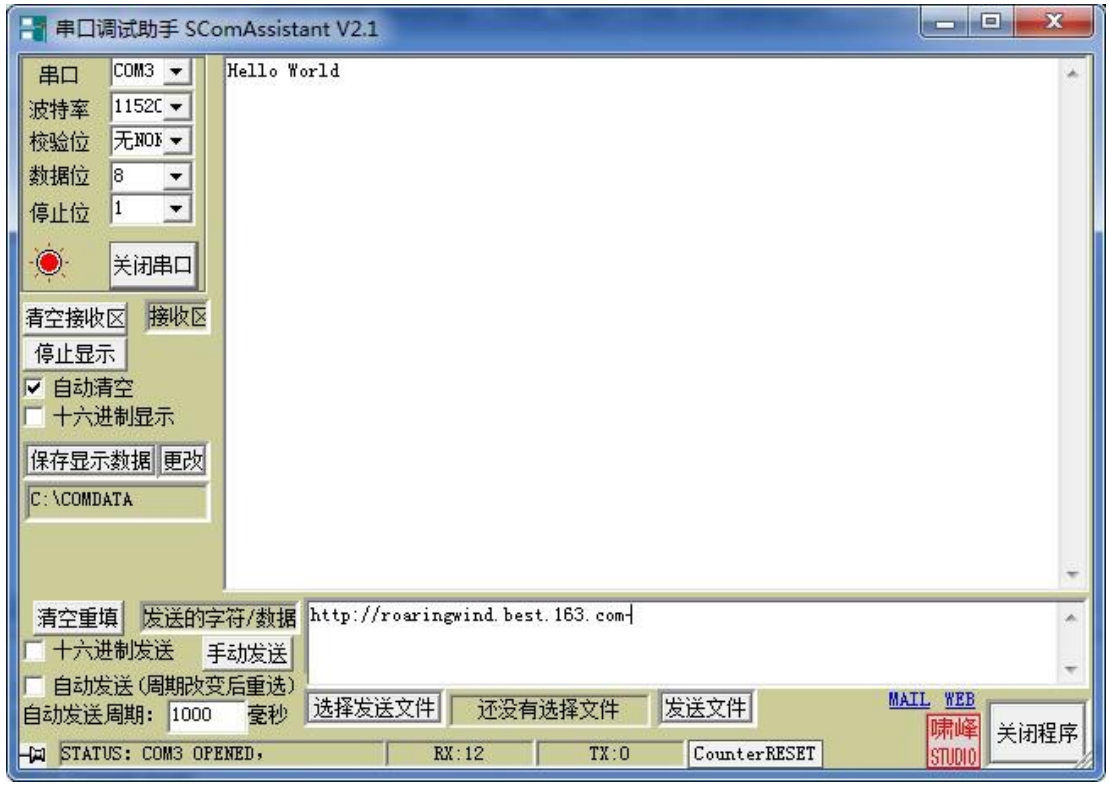


图 5- 29 接收到我们想要的信息

拓展实验：

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 World\n”,12);如图 5- 30，下载运行后发现串口不停地接收到 Hello World ，如图 5- 31 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。

注意！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，了解和学习了协议栈，我们只要掌握方法，就可以轻松应用。

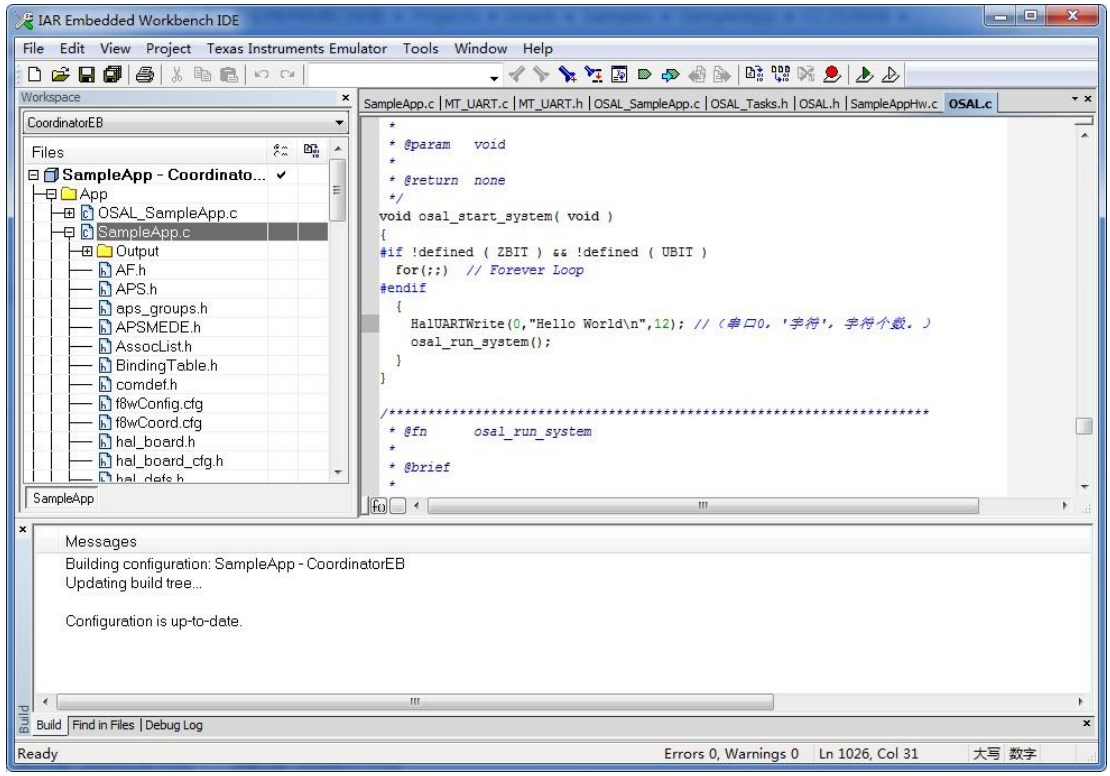


图 5- 30 串口打印代码

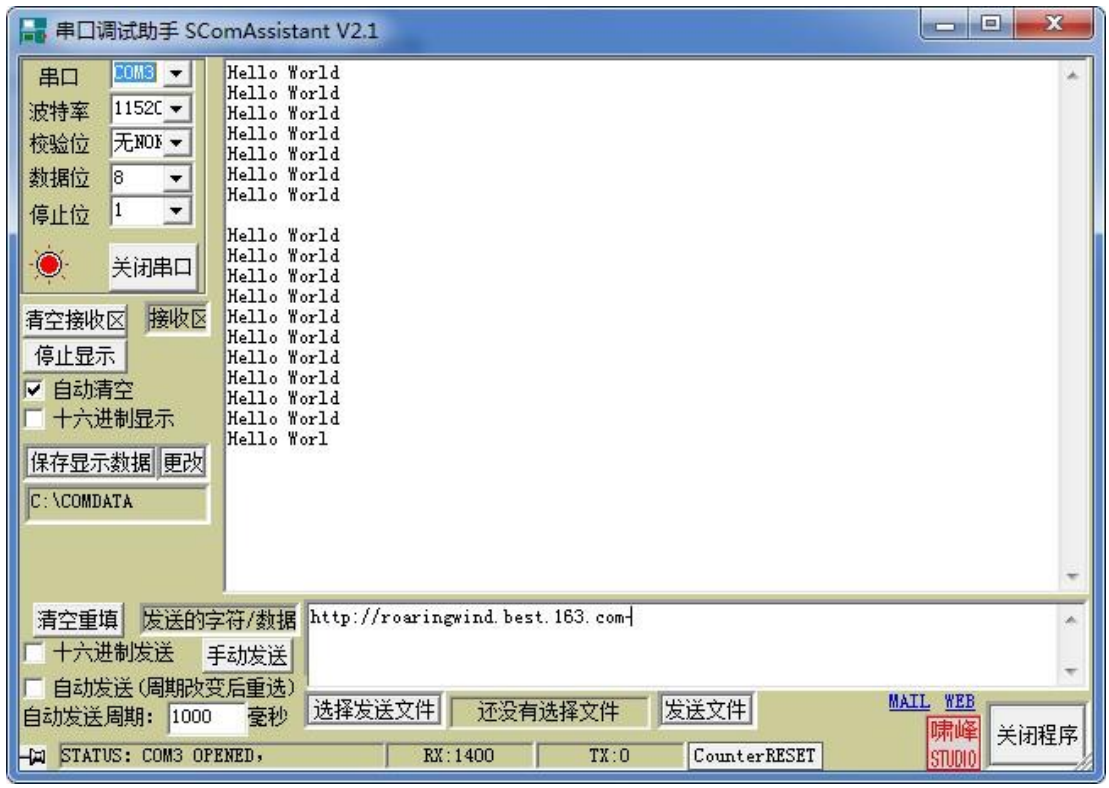


图 5- 31 串口提示

5.5.实验四：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，我们需根据自己的硬件平台，修改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。

实现平台：ZigBee 传感节点

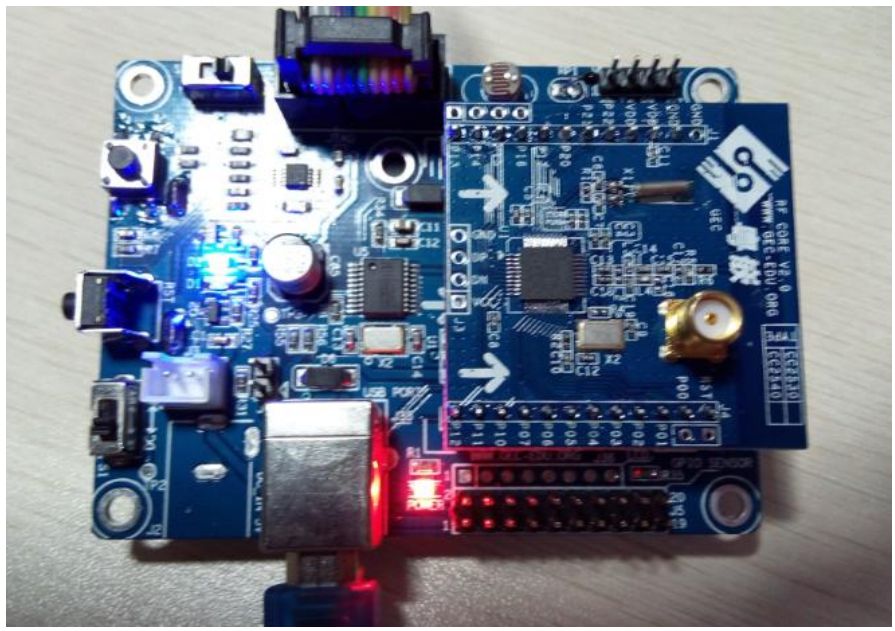


图 5- 32　ZigBee 传感节点

实验现象：通过节点 1 的按键 S3 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。

首先我们必须了解协议栈和官方学习板的设计原理。TI 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 改到我们平台的 IO 口上。按键 S3 连接的是 P1.2。

打开配套例程 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.

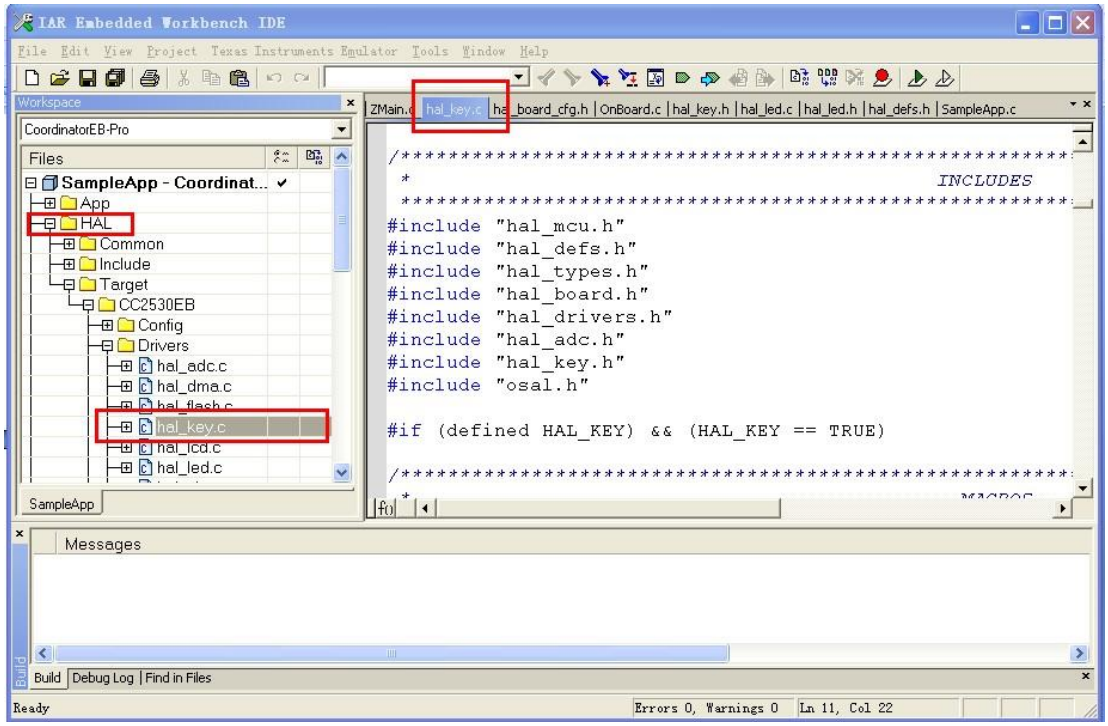


图 5- 33

1、 修改 SW_6 所在 IO 口

```
/* SW_6 is at P1.2 */
#define HAL_KEY_SW_6_PORT   P1
#define HAL_KEY_SW_6_BIT    BV(2) //BV(1) 改到 P1.2
#define HAL_KEY_SW_6_SEL    P1SEL
#define HAL_KEY_SW_6_DIR    P1DIR
```

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      IEN2  /* CPU interrupt mask register */
#define HAL_KEY_SW_6_IENBIT   BV(4) /* Mask bit for all of Port_1*/
#define HAL_KEY_SW_6_ICTL     P1IEN /* Port Interrupt Control register */
#define HAL_KEY_SW_6_ICTLBIT BV(2) //BV(1) /* P0IEN - P0.1 enable/disable
bit 改到P1.2*/
#define HAL_KEY_SW_6_PXIFG    P1IFG /* Interrupt flag at source */
```

我们不需要用到TI的摇杆J-STICK，所以把代码注释掉并添加我们的按键。如图5- 34所示：

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

  /* Key is active HIGH */
  if ((HAL_KEY_JOY_MOVE_PORT & HAL_KEY_JOY_MOVE_BIT))
  {
    //keys = halGetJoyKeyInput();
  }
  if (!Hal_KeyIntEnable)
  {
    if (HAL_PUSH_BUTTON1())
    {
      keys |= HAL_KEY_SW_6;
    }
```

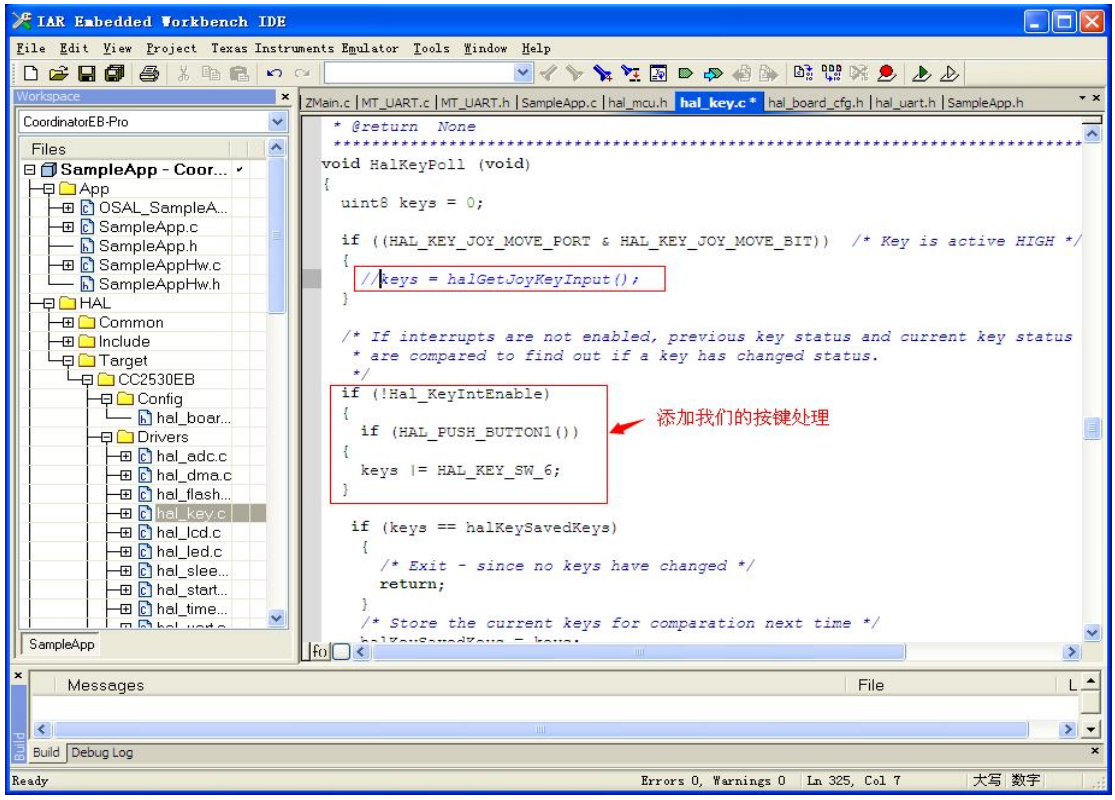


图 5- 34 注释掉 TI 摇杆代码

第二步：修改 hal_board_cfg.h 文件。

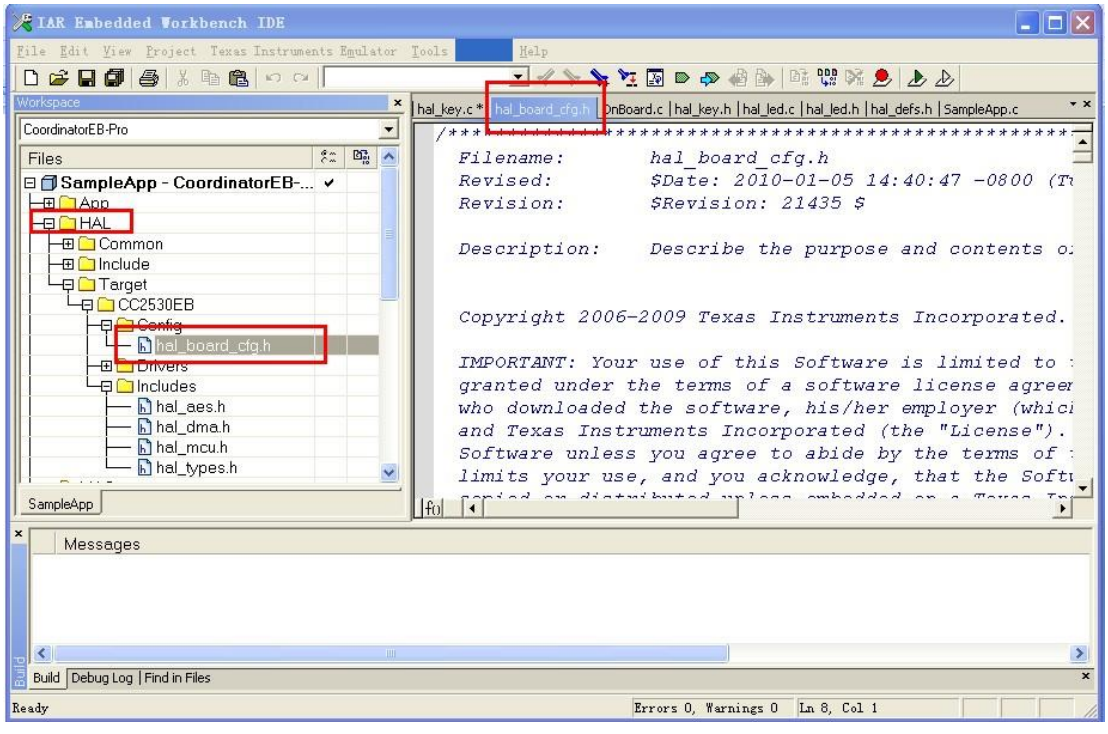


图 5- 35

1、 修改 SW_6 所在 IO 口

```
/* S3 */
#define PUSH1_BV    BV(2)  //BV(1)
#define PUSH1_SBIT  P1_2  //P0_1
    #define PUSH1_POLARITY    ACTIVE_LOW    //低电平触发
```

第三步：修改 OnBoard.C 文件。在 ZMain.C 目录树下，如图 0- 6 所示：

2、 使能中断

```
// OnboardKeyIntEnable 为布尔值
OnboardKeyIntEnable = !HAL_KEY_INTERRUPT_ENABLE;
HalKeyConfig(OnboardKeyIntEnable, OnBoard_KeyCallback);
```

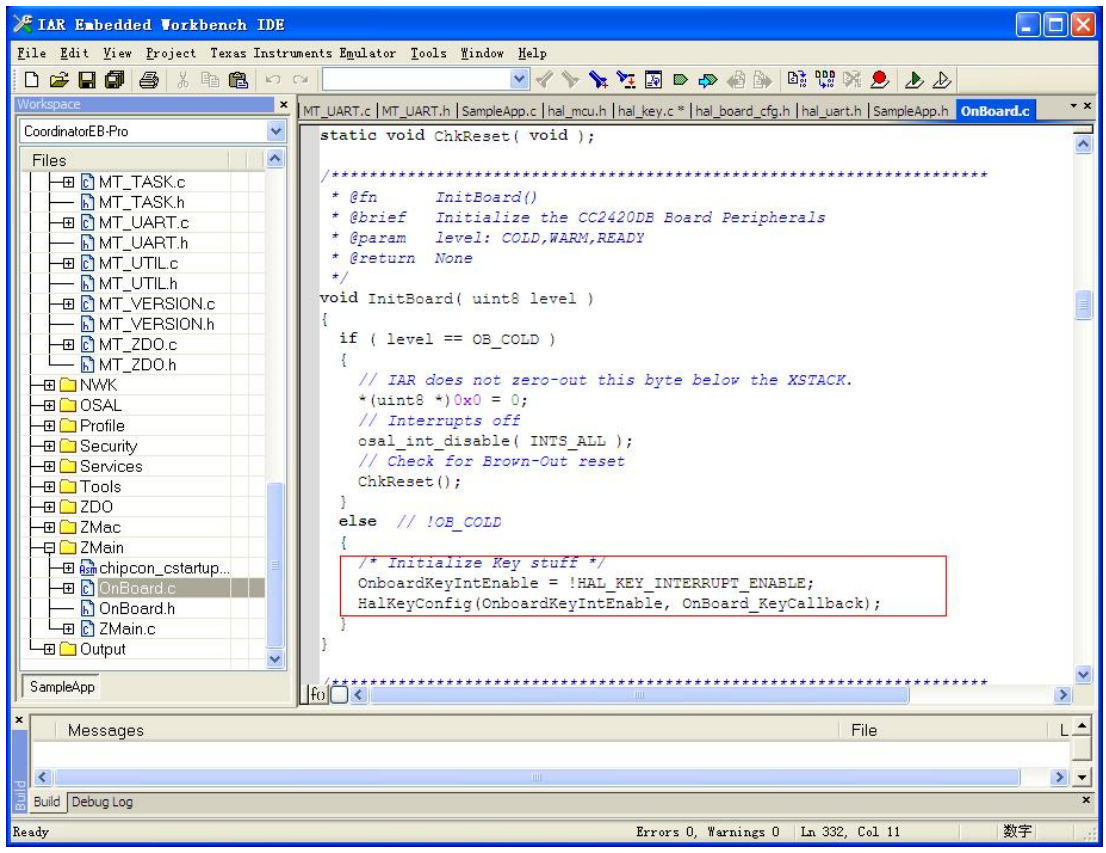


图 5- 36 允许按键中断

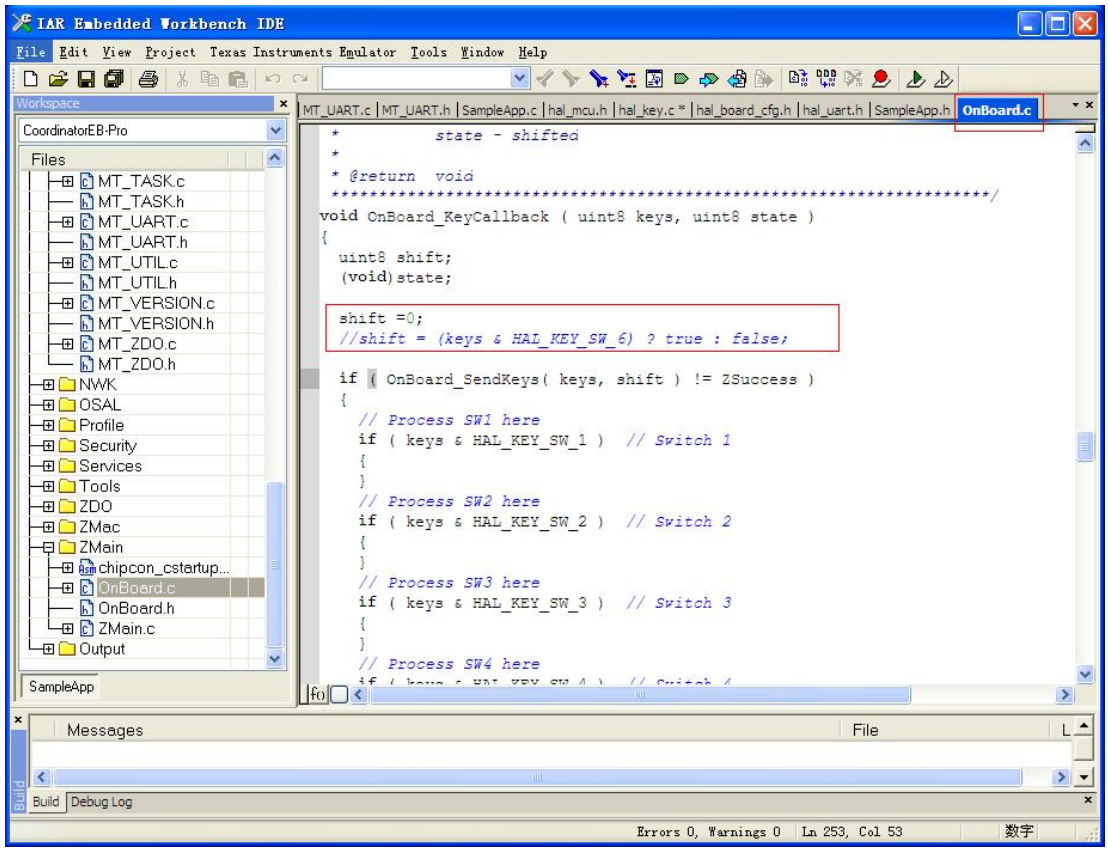


图 5- 37

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;
```

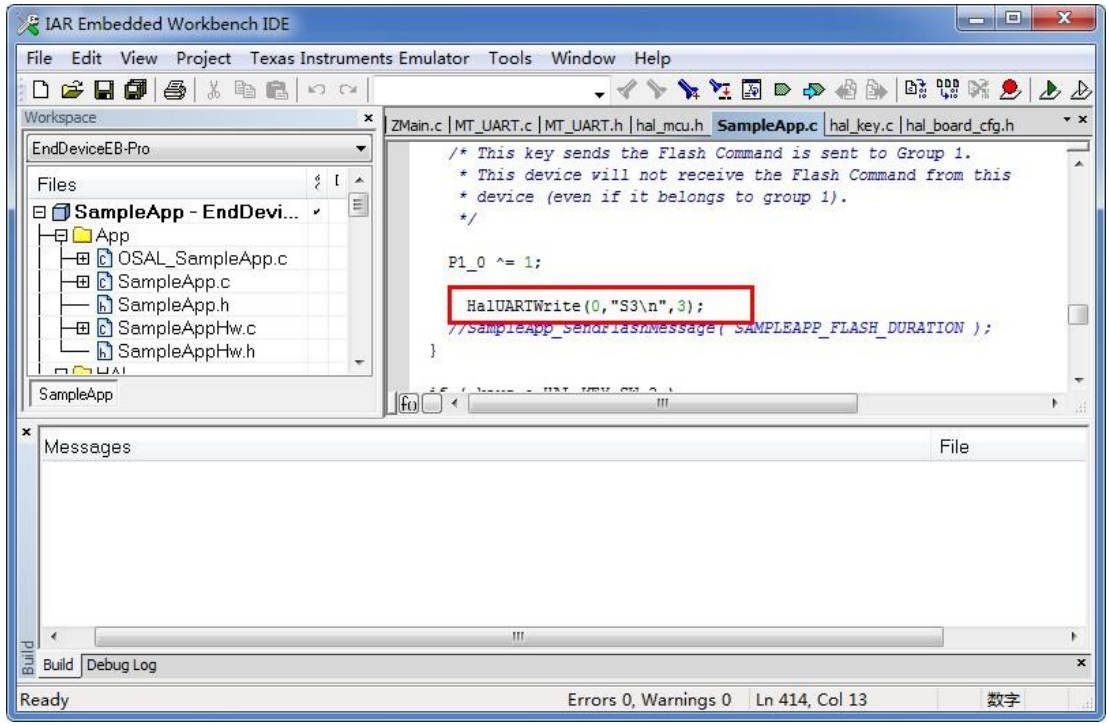


图 5- 38 加入按键下提示代码

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 ZigBee 节点板上的 S3。

```
if ( keys & HAL_KEY_SW_6 )
{
  P1_0 ^= 1;  //按键按下指示灯
  HalUARTWrite(0,” S3\n “,3);              //提示被按下的是 KEY1
}
```

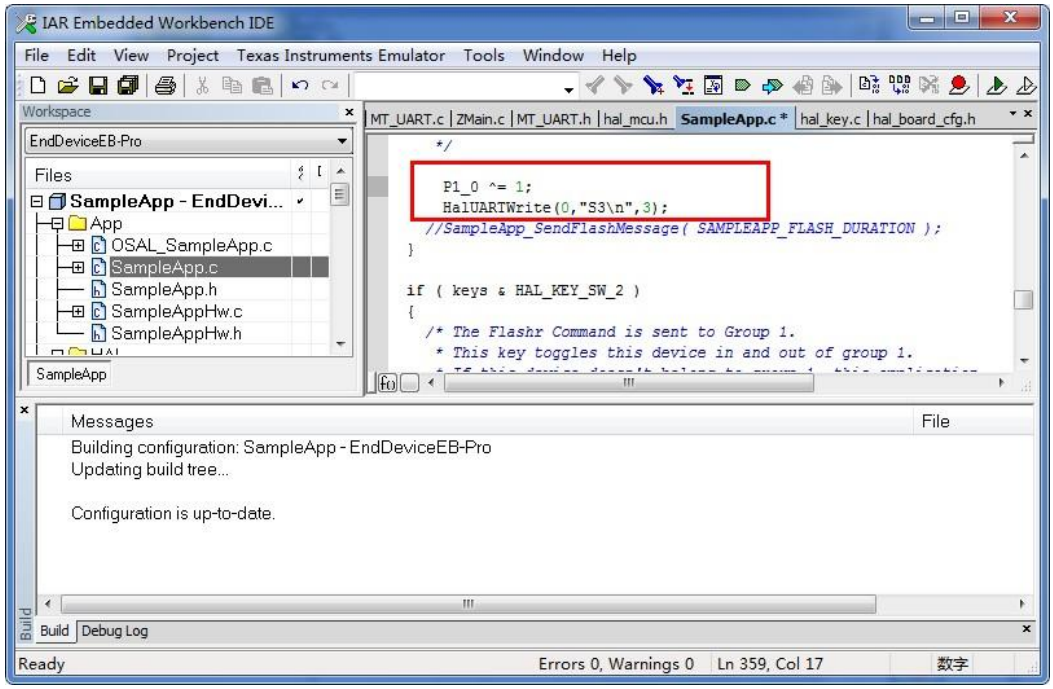


图 5- 39 添加 S3 对应按键编号代码

我们下载程序到 ZigBee 节点板。打开串口调试助手，当按下按键 S3 时候。可以看到有提示信息打印出来。

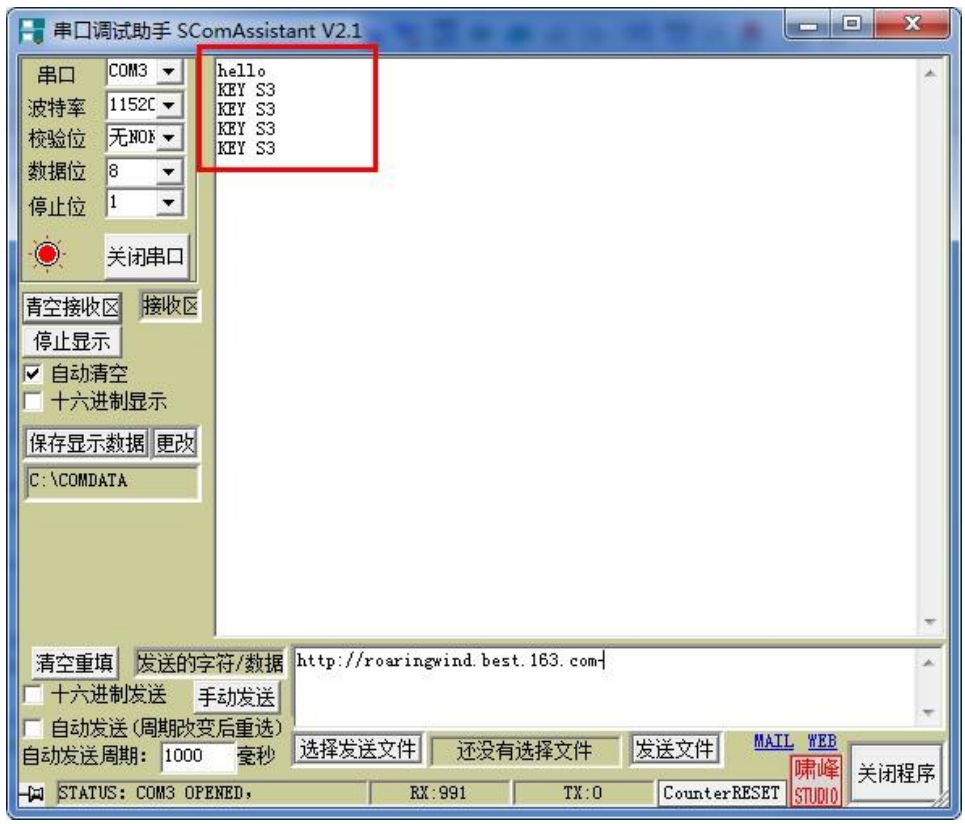


图 5- 40 按键提示

到这里，ZigBee 传感节点的按键配置完成。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。

5.6. 实验五：串口数据透传实验

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 5-41 所示：

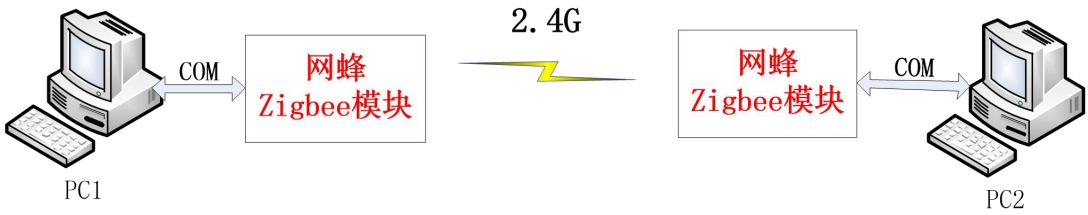


图 5- 41 串口透传原理

实现平台：1 个 ZigBee 协调器，1 个 ZigBee 传感节点；

图 5- 42 ZigBee 协调器和节点

实验现象：两台不同的 PC 机通过串口/USB 线连接至协调器或者节点板（自带 USB 转串口功能），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。

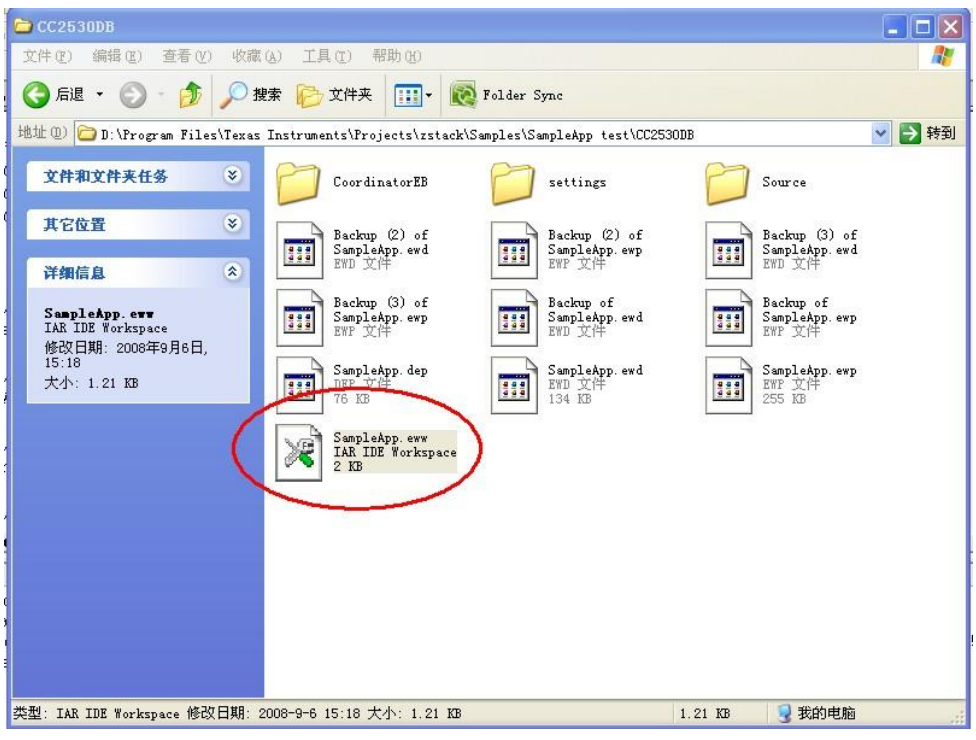


图 5- 43

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。

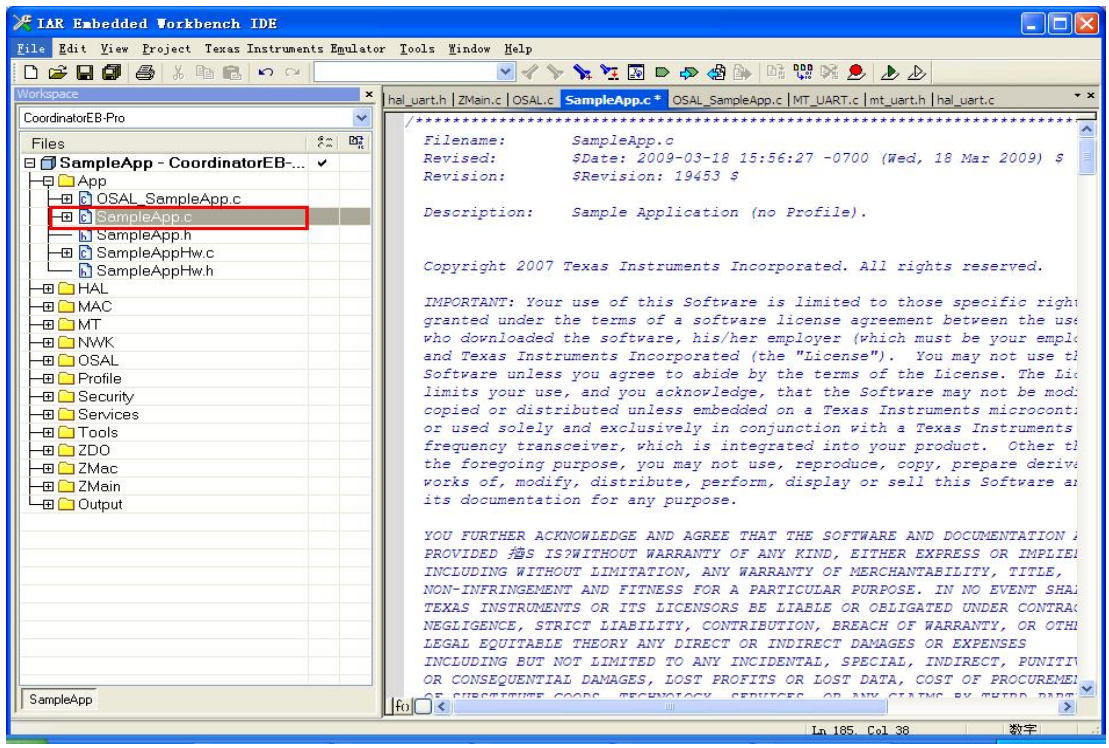


图 5- 44

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, CMD0, 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；

CMO：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

具体参考程序如下：

```
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. }
```

由我们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，用我
                      我们提供的方法接收，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。

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 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 5- 58 **所示：**

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

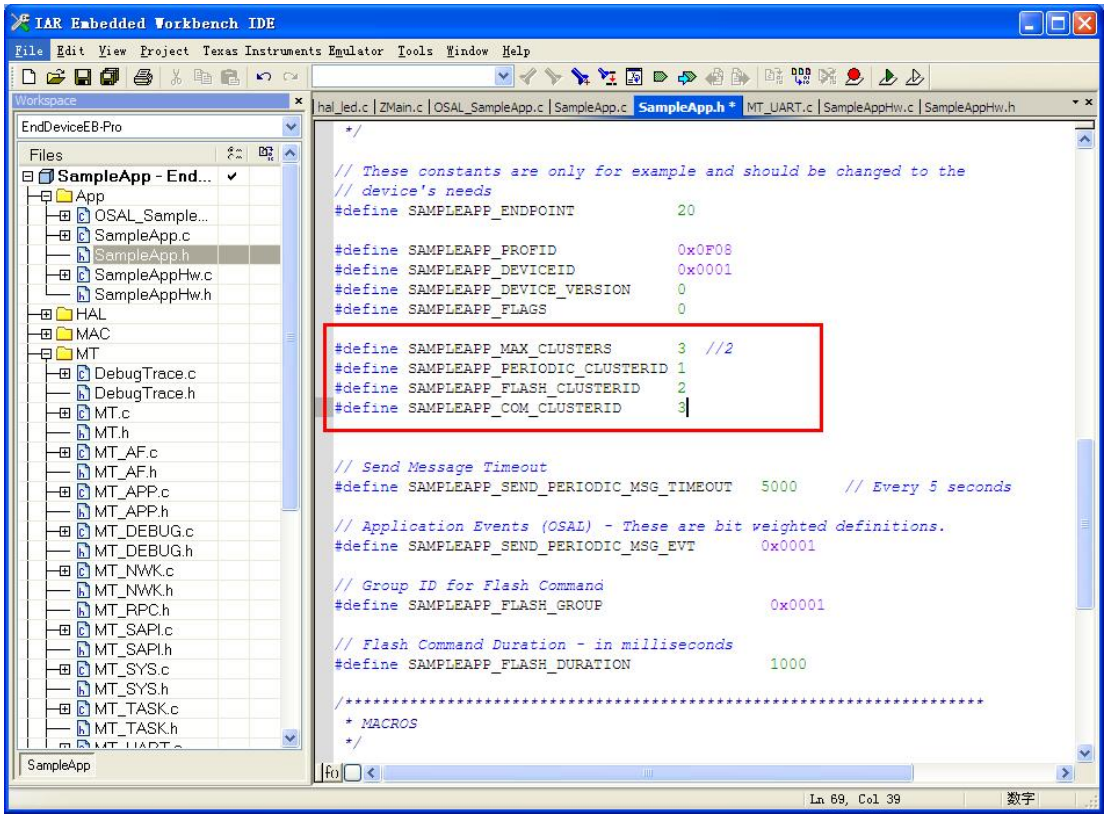


图 5- 45 加入串口的 CLUSTERID

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板子上，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 5- 46

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

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-> lustered )
   {
   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
```

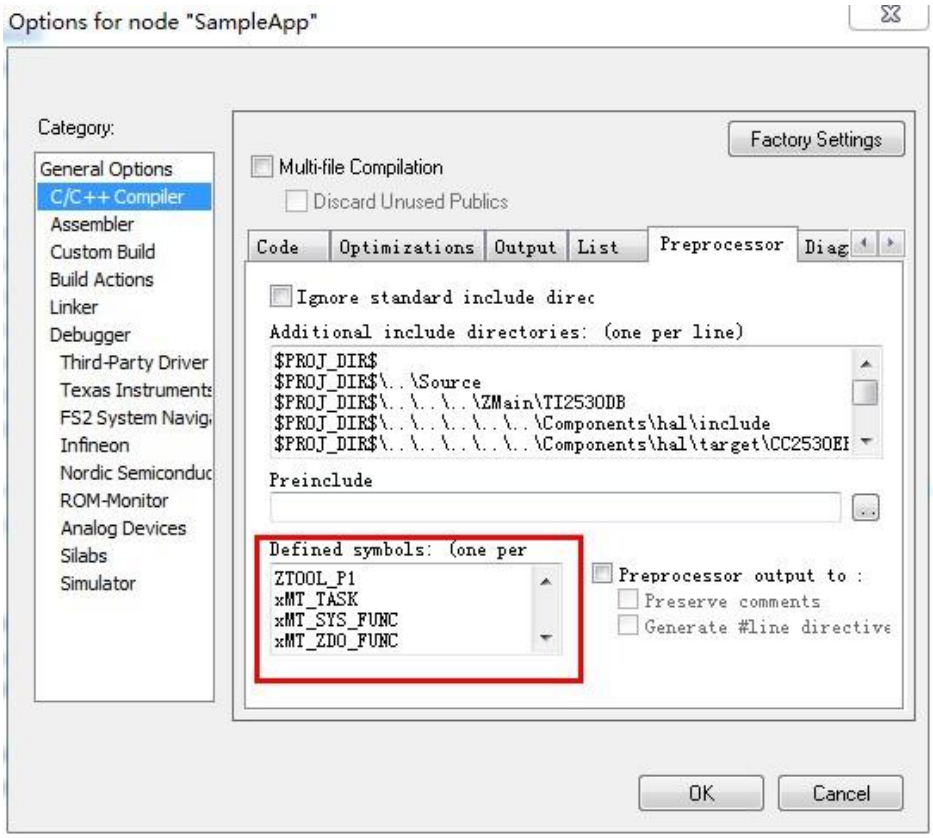


图 5- 47 预定义

至此，所有配置完成，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 ZigBee 传感节点。用户可用 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。

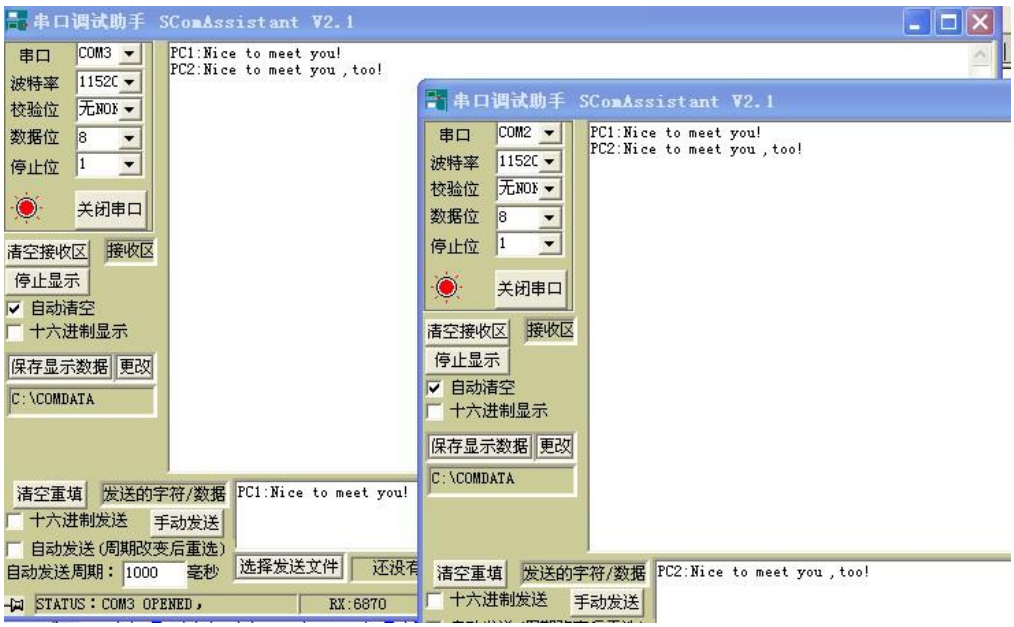


图 5- 48 同一台 PC 的 2 个不同串口演示

5.7. 实验六：网络通讯实验（单播、组播、广播）

实验原理：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：ZigBee 节点 3 个以上。分别用于协调器、路由器、终端。

图 5- 49 ZigBee 实验相关设备

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。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。

第一：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。

我们打开 AF.h 文件，找到下面代码。

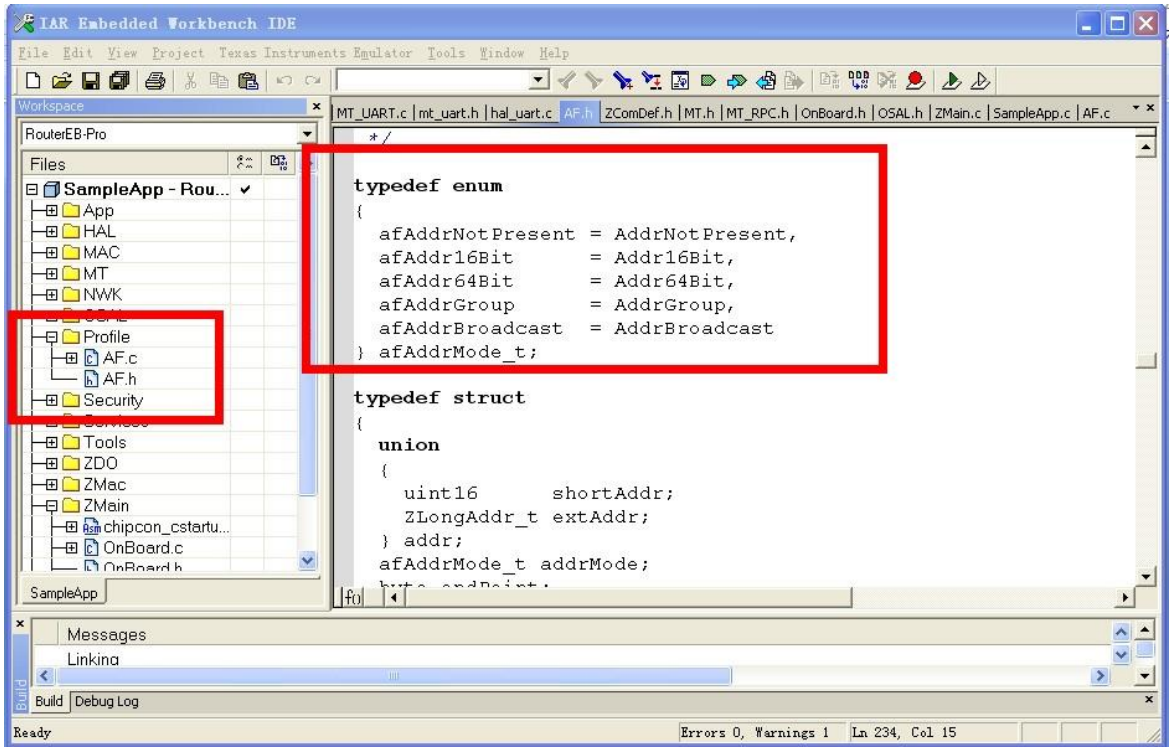


图 5- 50

```
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

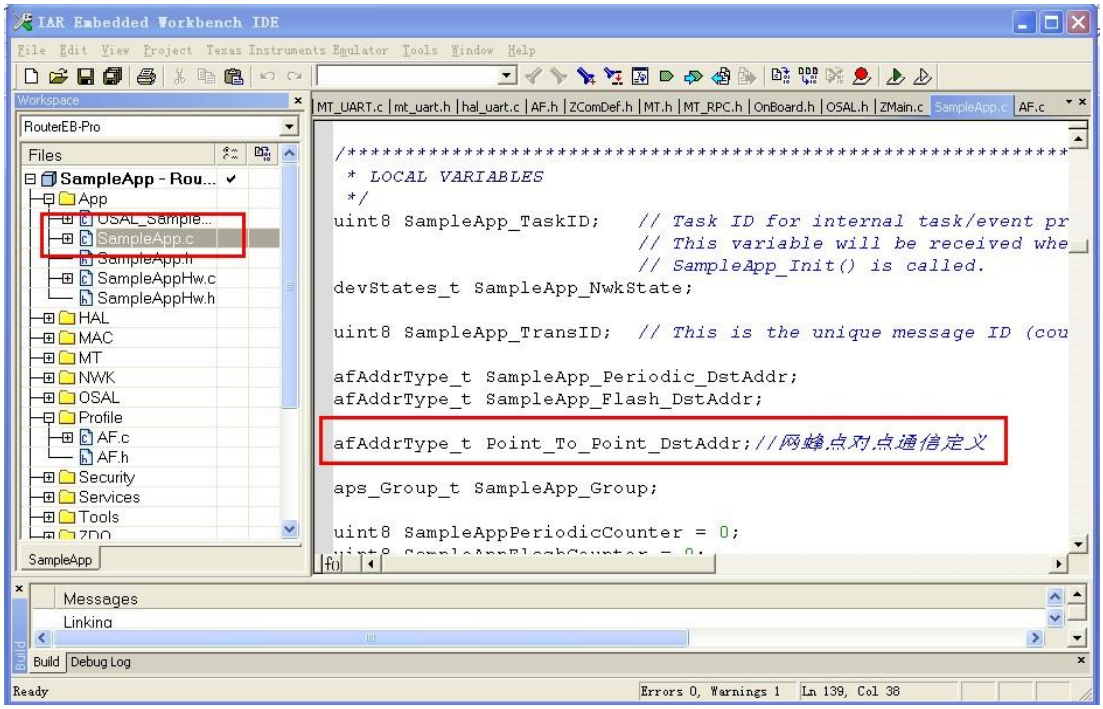


图 5- 52 SampleApp.c 文件

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;//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 5- 65 所示）：

```
// 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。

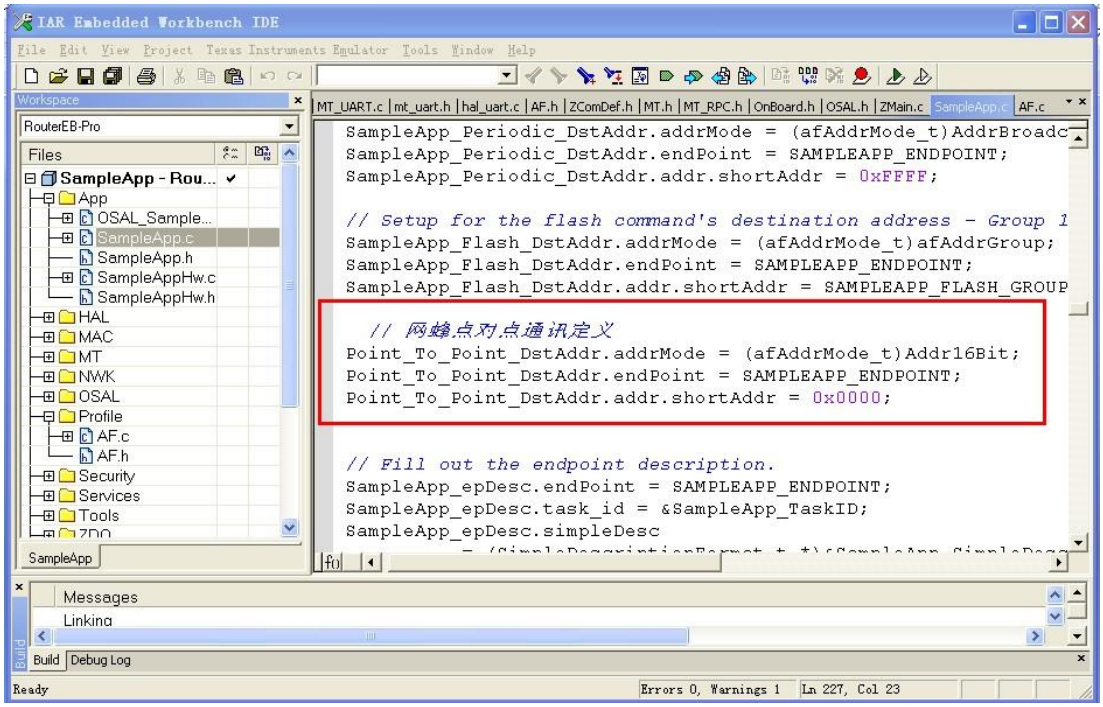


图 5- 53

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 5- 54 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.
    }
}
```

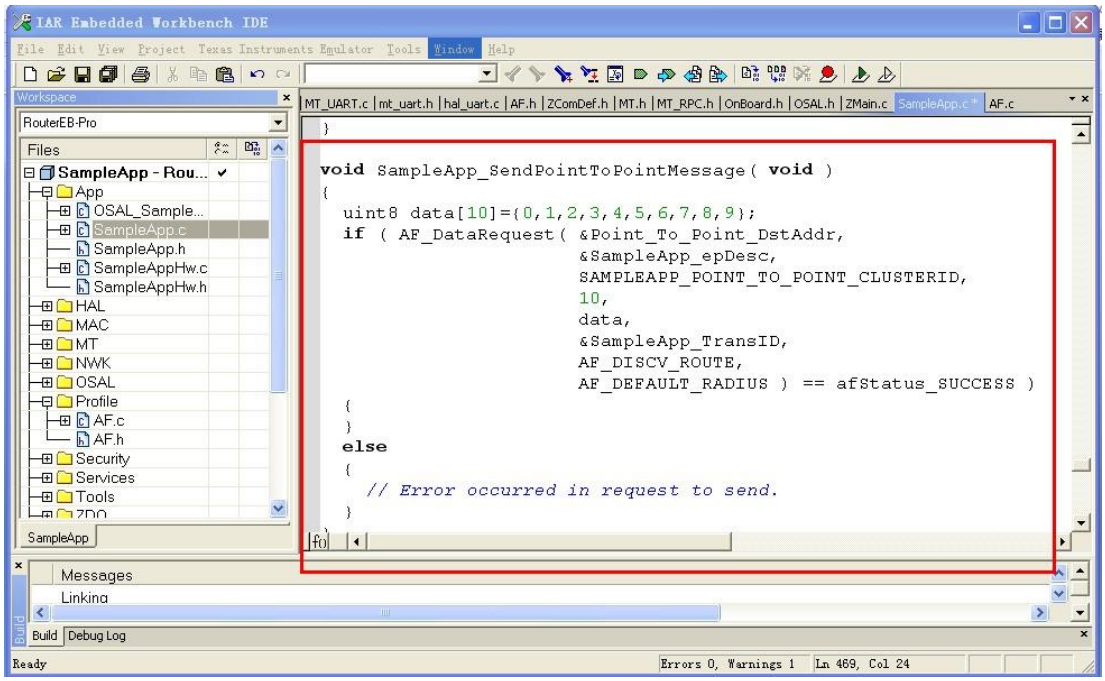


图 5- 54 添加点对点发送函数

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
```

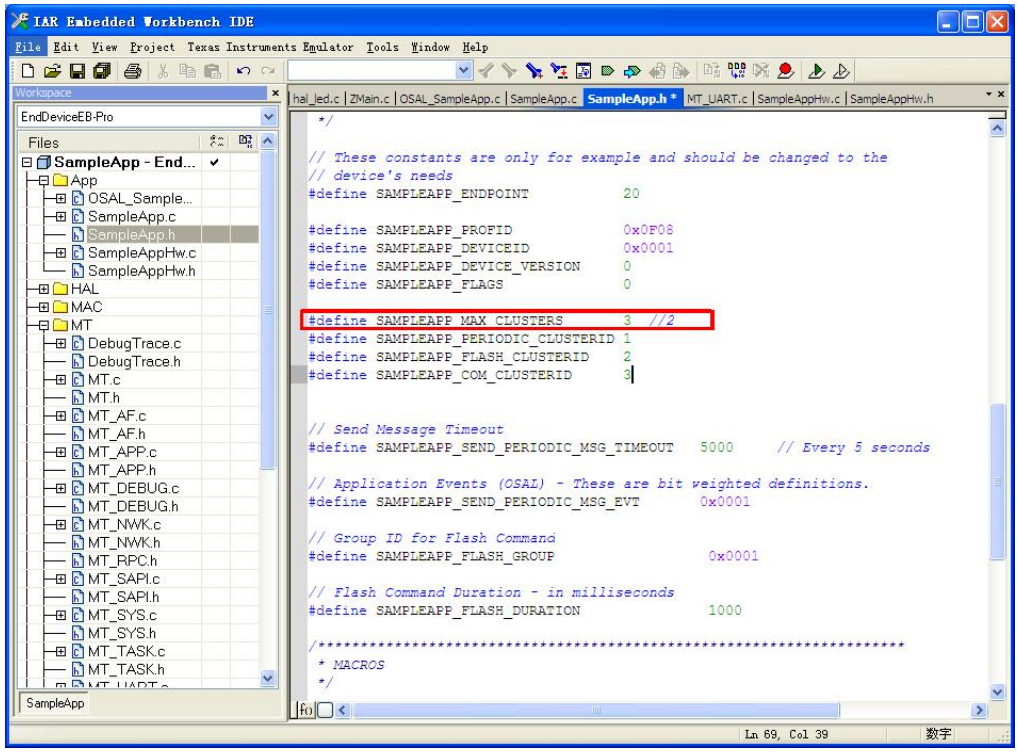


图 5- 55 添加 ID 号

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 5- 68 所示）。

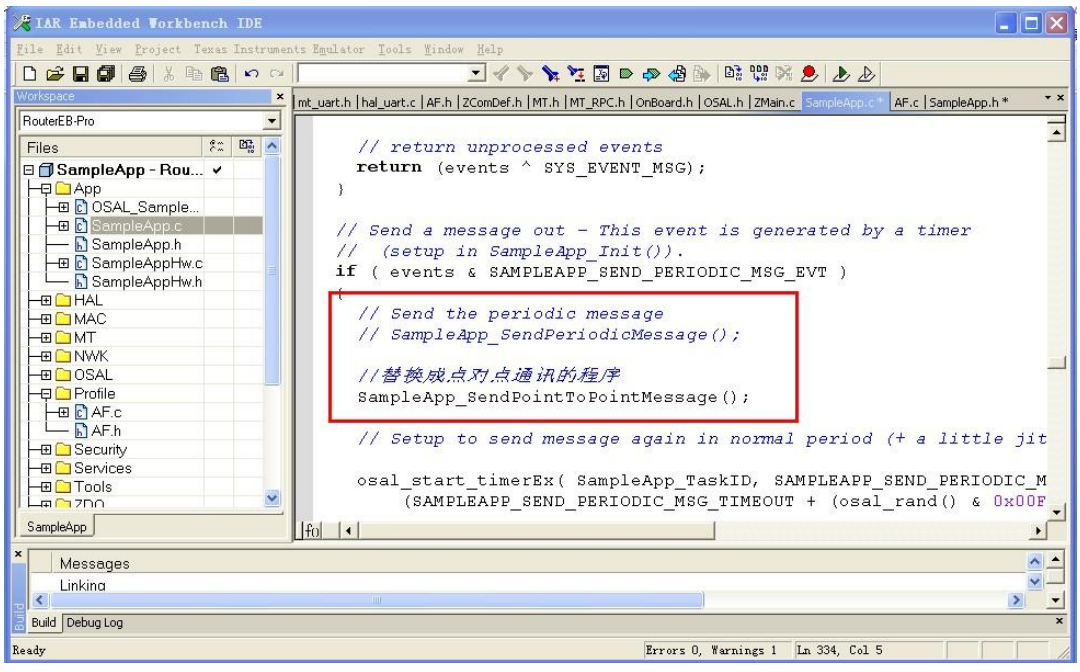


图 5- 56

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 5- 57 所示：

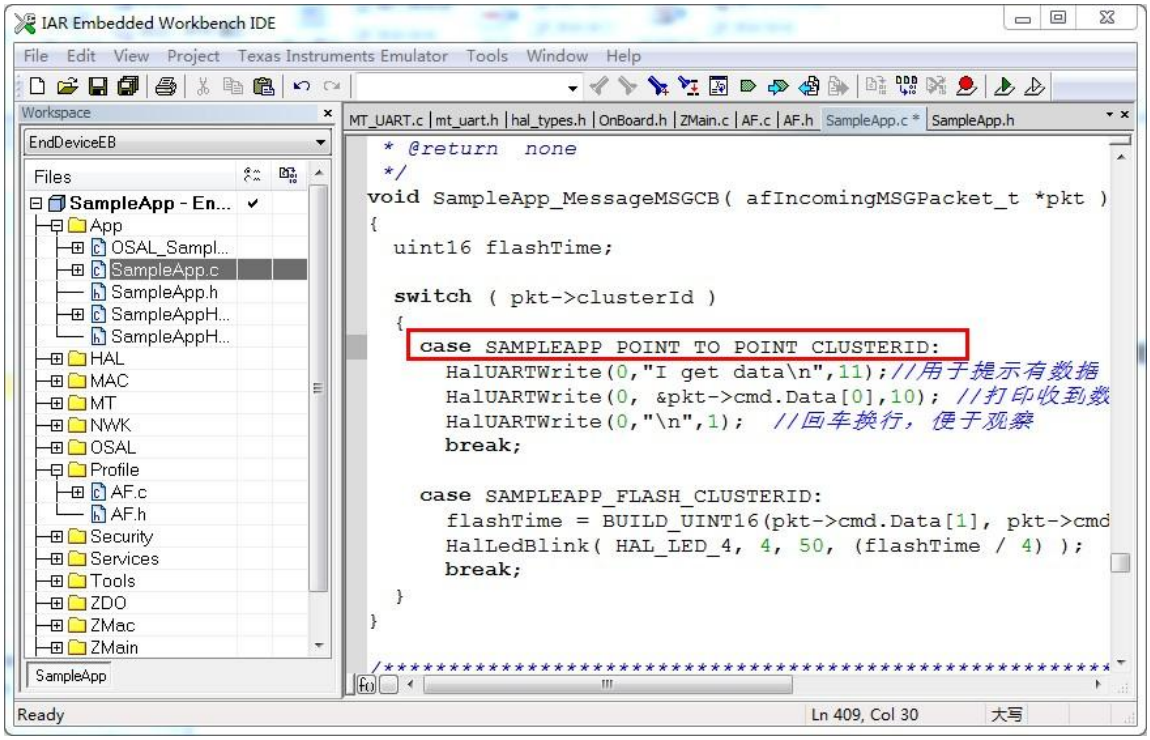


图 5- 57 修改接收 ID

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 5-58 所示：

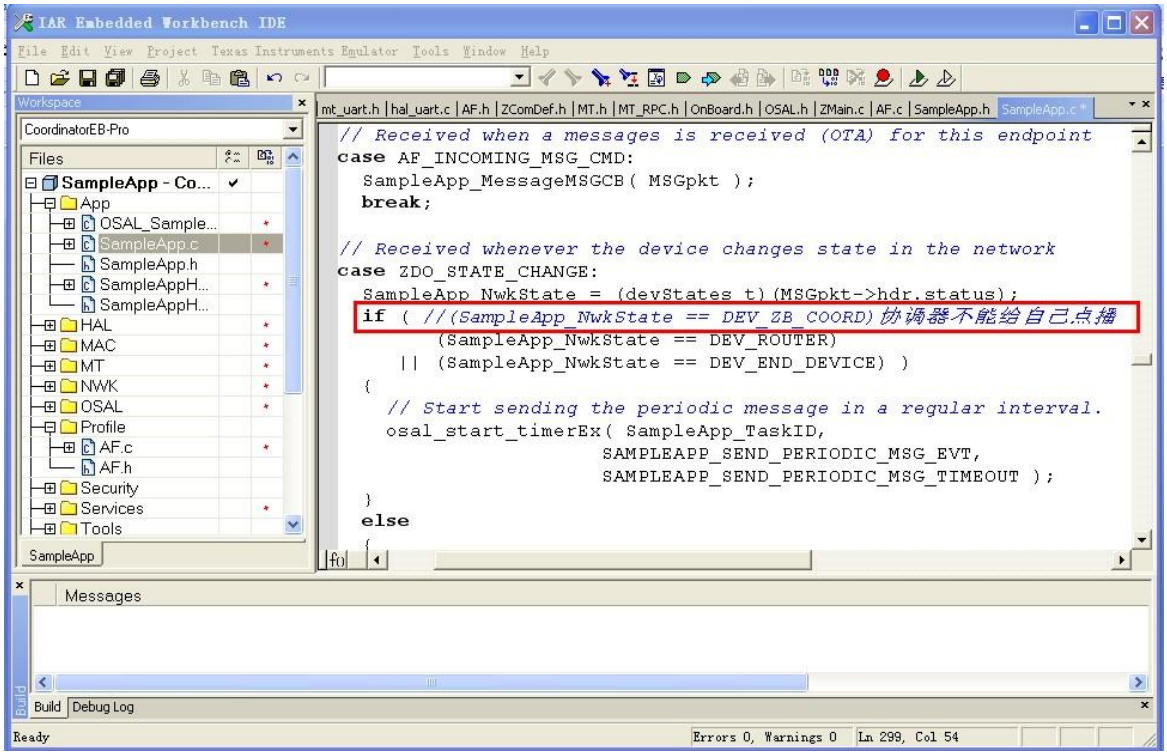


图 5- 58 协调器不给自己点播

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 5- 71 所示：

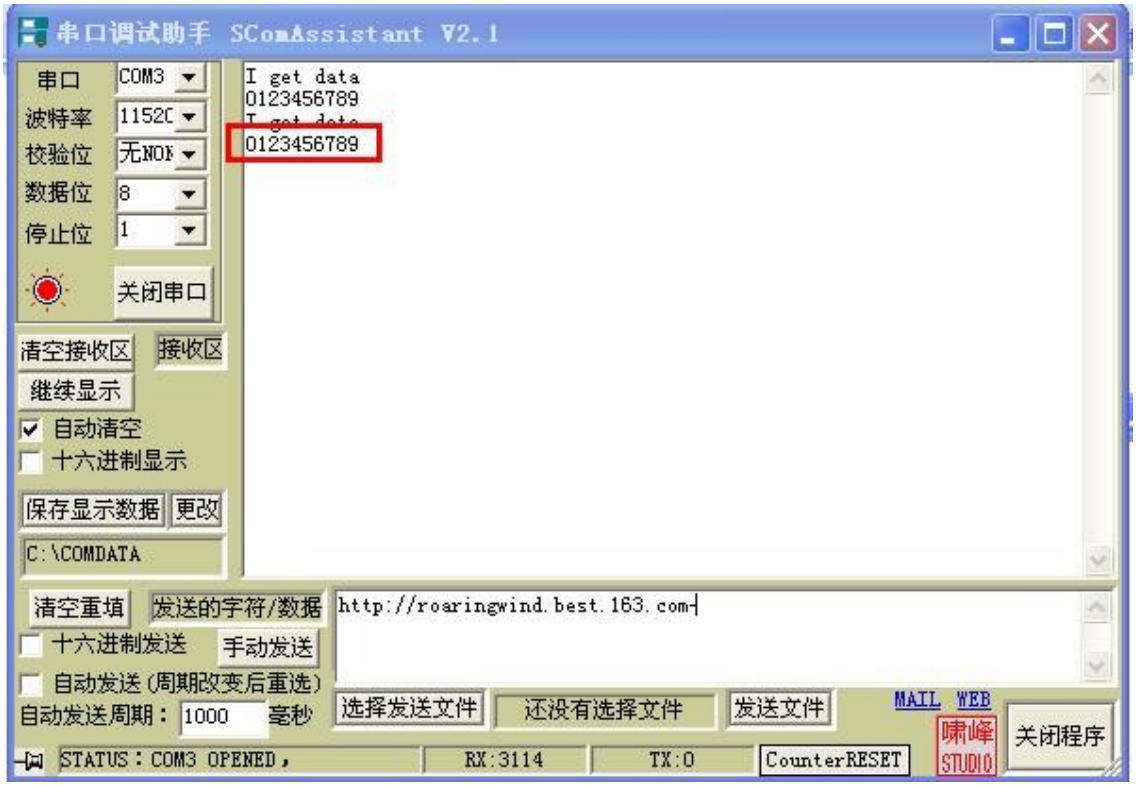


图 5- 59 点播方式发送数据

第二：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，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 5-60 如所示：

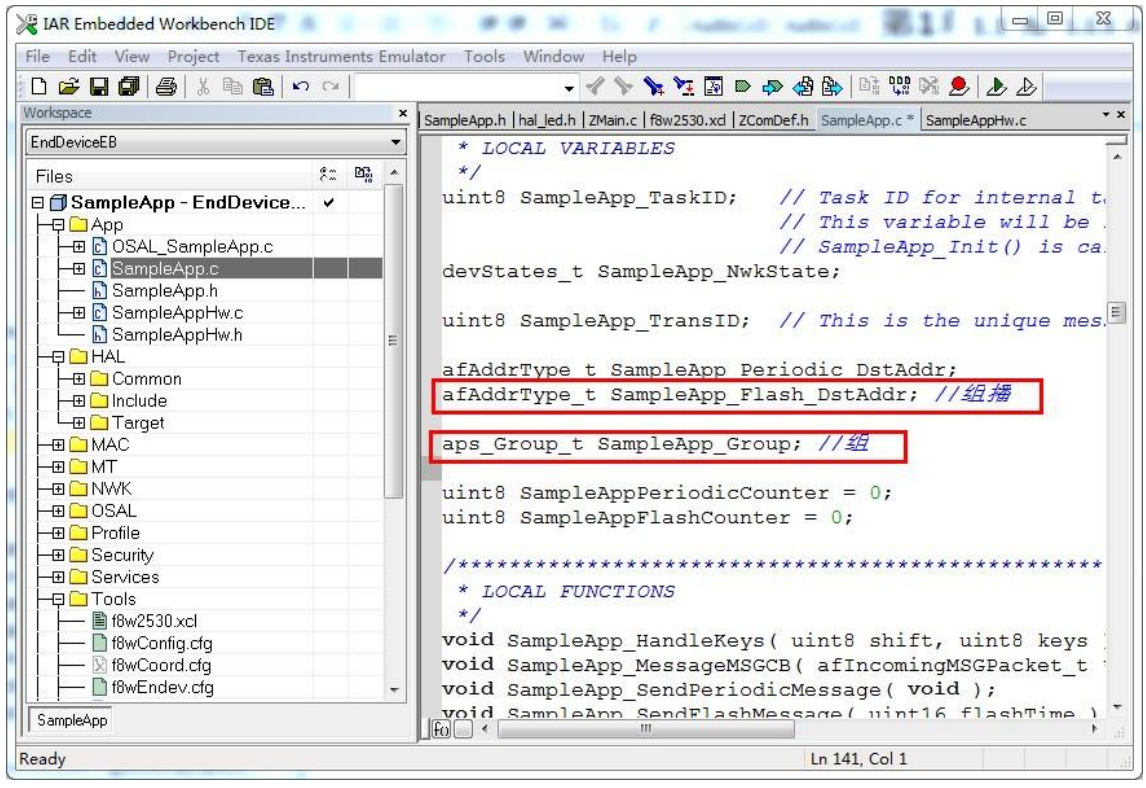


图 5- 60 组播定义

组播参数的配置。代码如下（图 5-61 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;

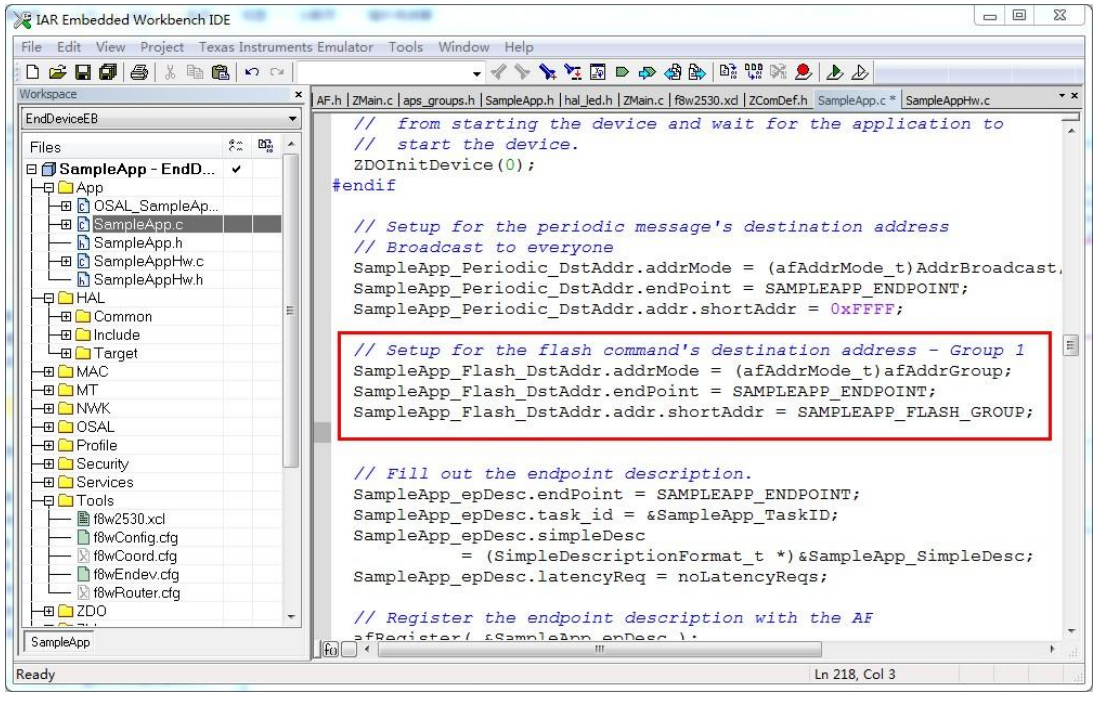


图 5- 61

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 5- 62 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 );
```

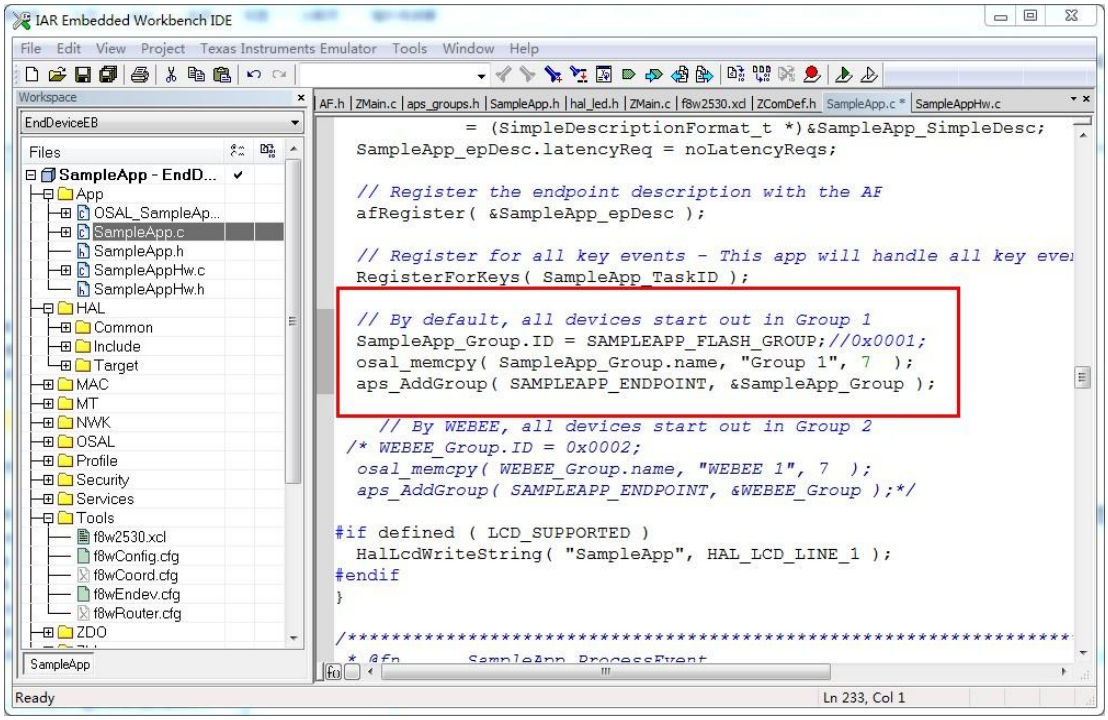


图 5- 62

在 SampleApp.h 里面可以看到组号为 0x0001(如图 5-63 所示)：

```
// Group ID for Flash Command

#define SAMPLEAPP_FLASH_GROUP           0x0001
```

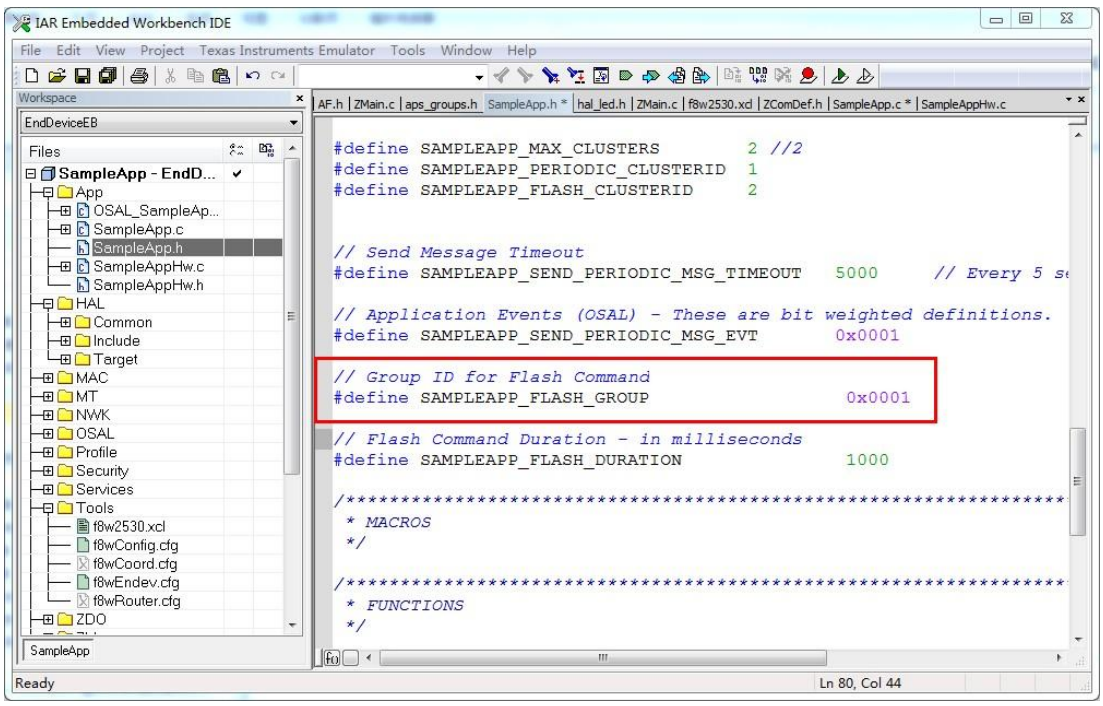


图 5- 63 组 ID 号

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 5- 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.
  }
}
```

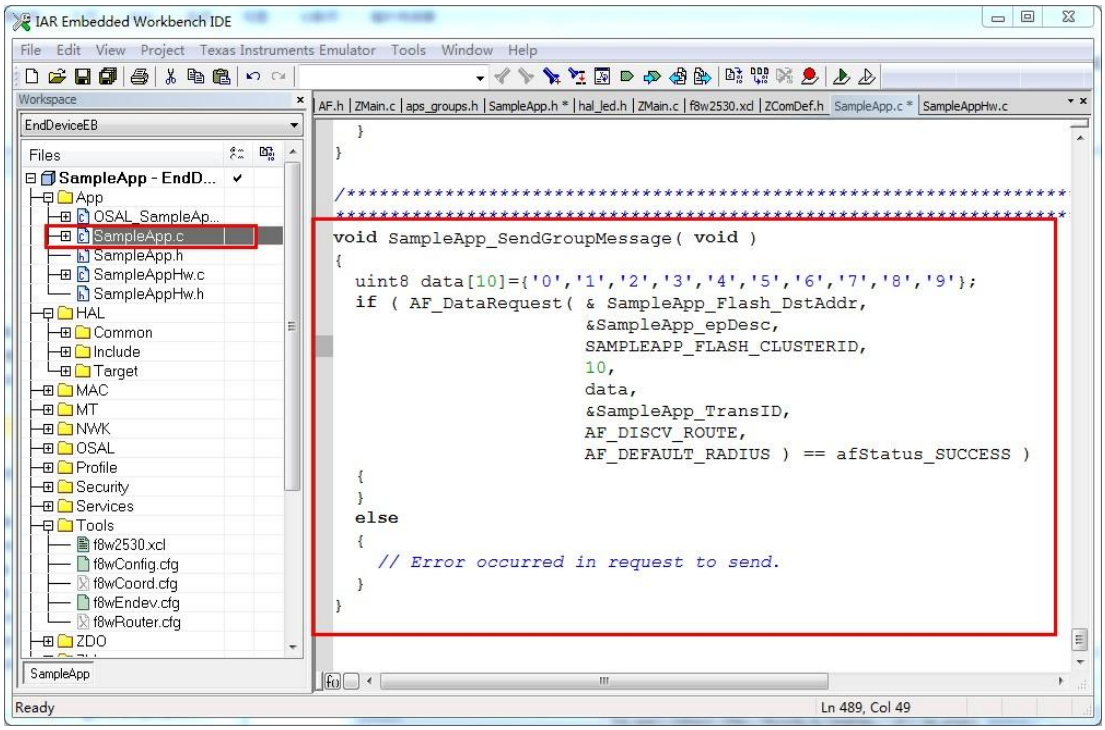


图 5- 64 组播函数代码

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

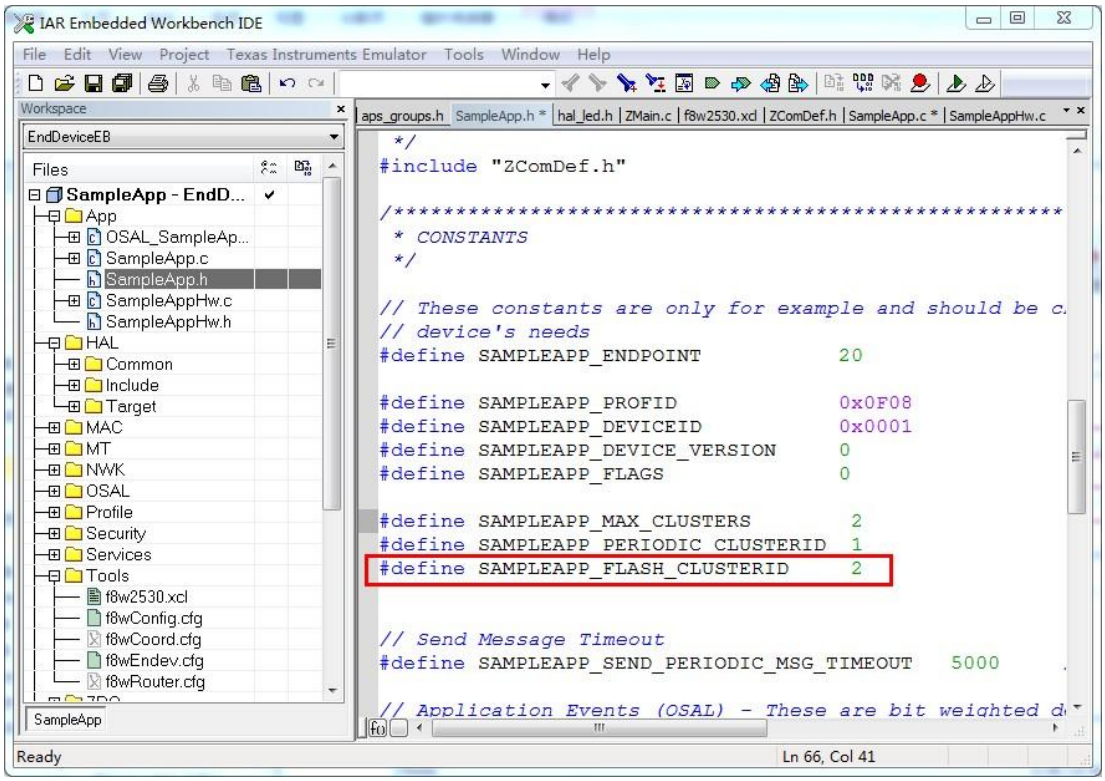


图 5- 65

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 5-66 所示)。

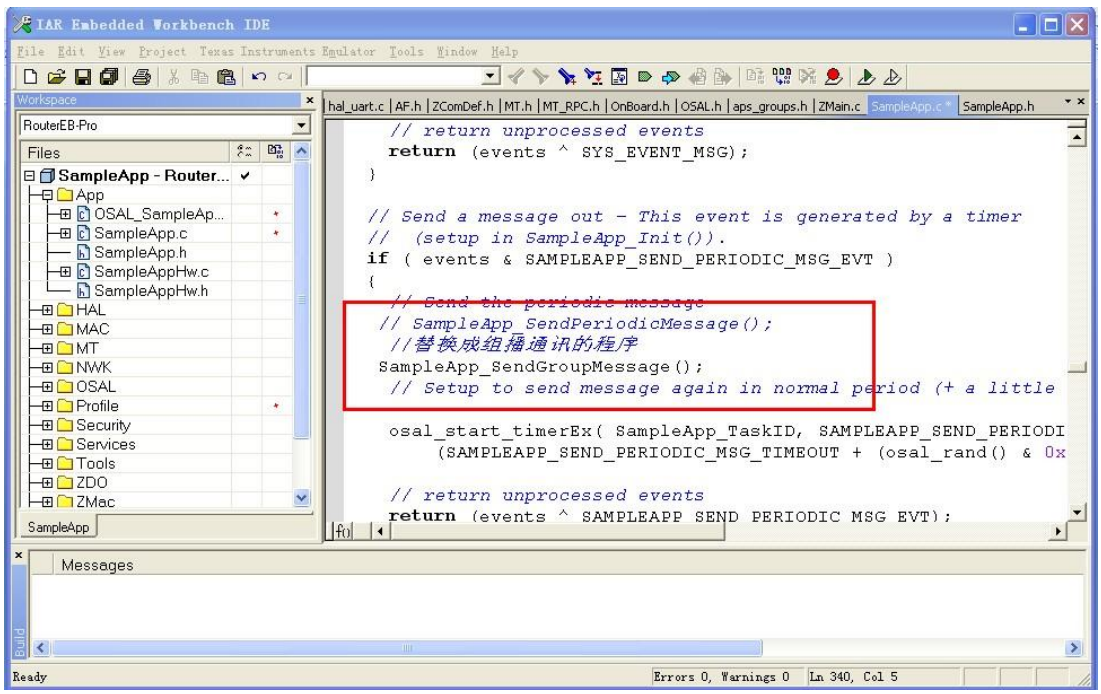


图 5- 66 替换成组播函数

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 5-67 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
```

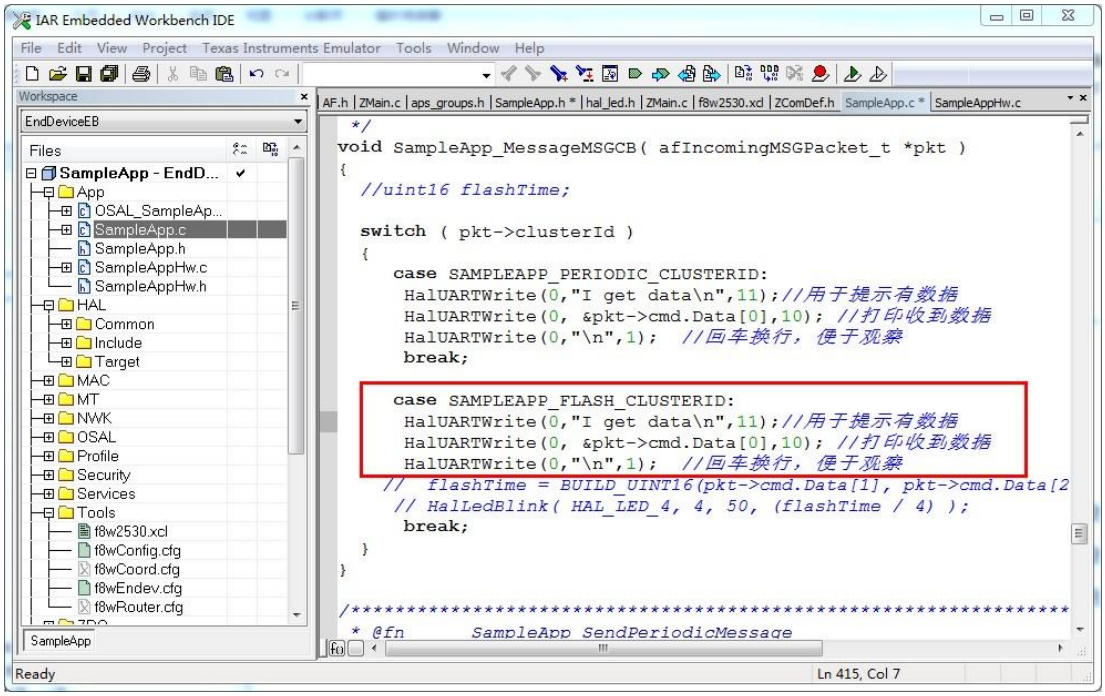


图 5- 67

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 5-68 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 5-69 所示。（注意：终端设备不参与组播实验，具体往下看！）

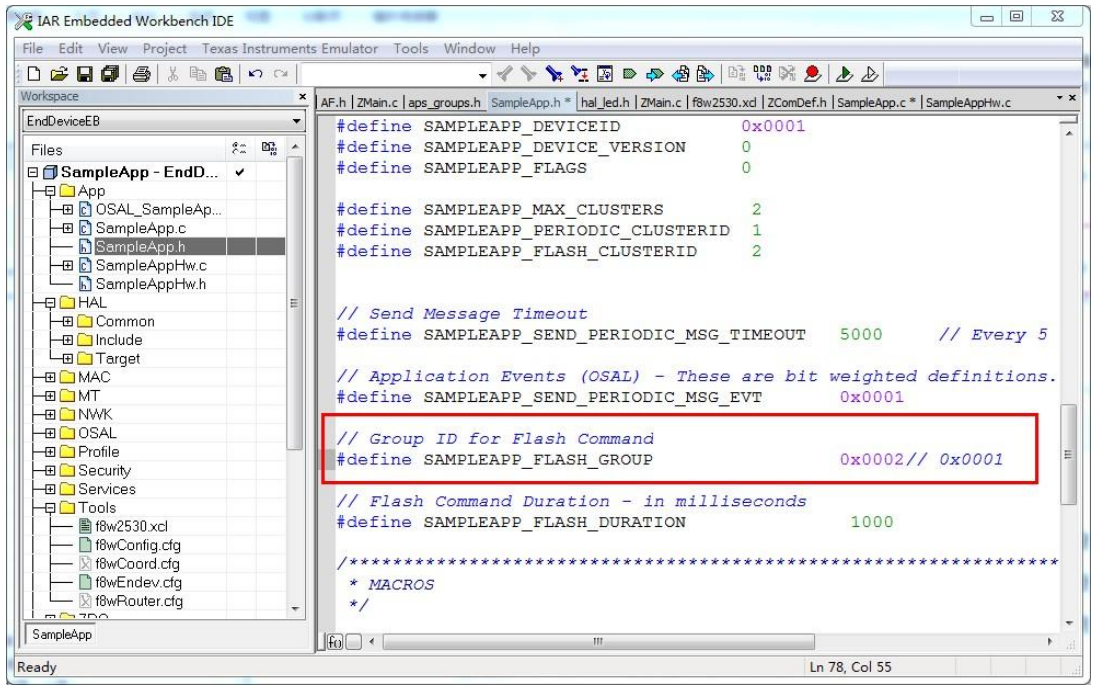


图 5- 68

图 5- 69 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！

第三：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 5-70 所示

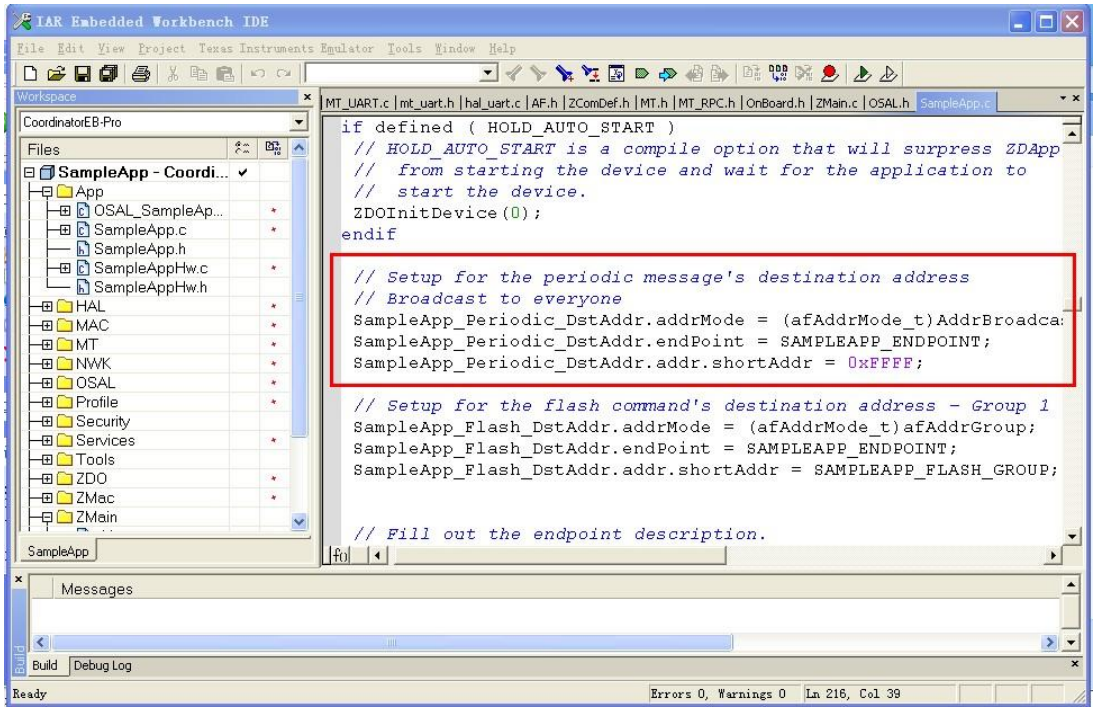


图 5- 70

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 5-83 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.
  }
}
```

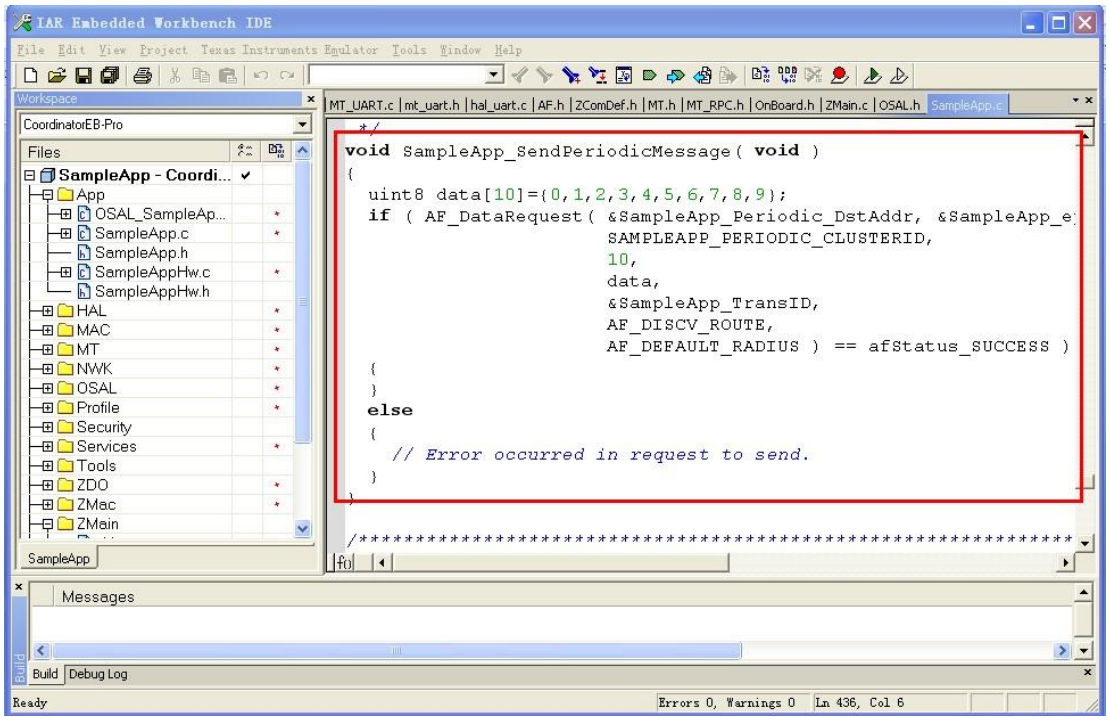


图 5- 71

其中图 5-72 所示：

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

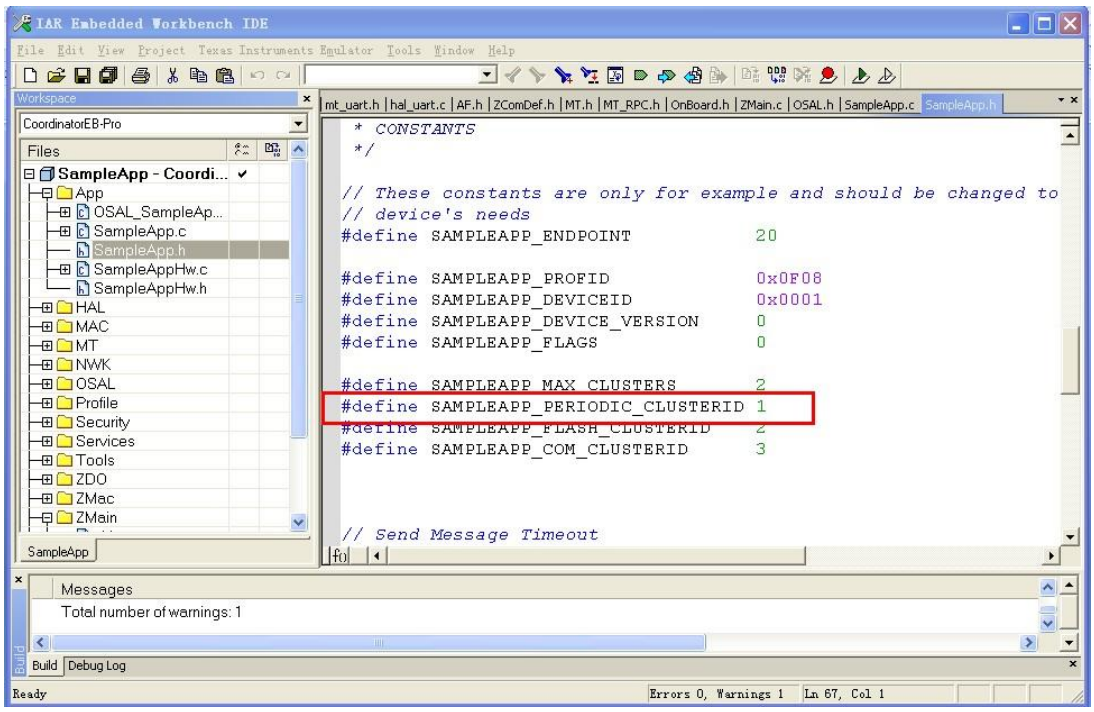


图 5- 72

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 5-73 所示。

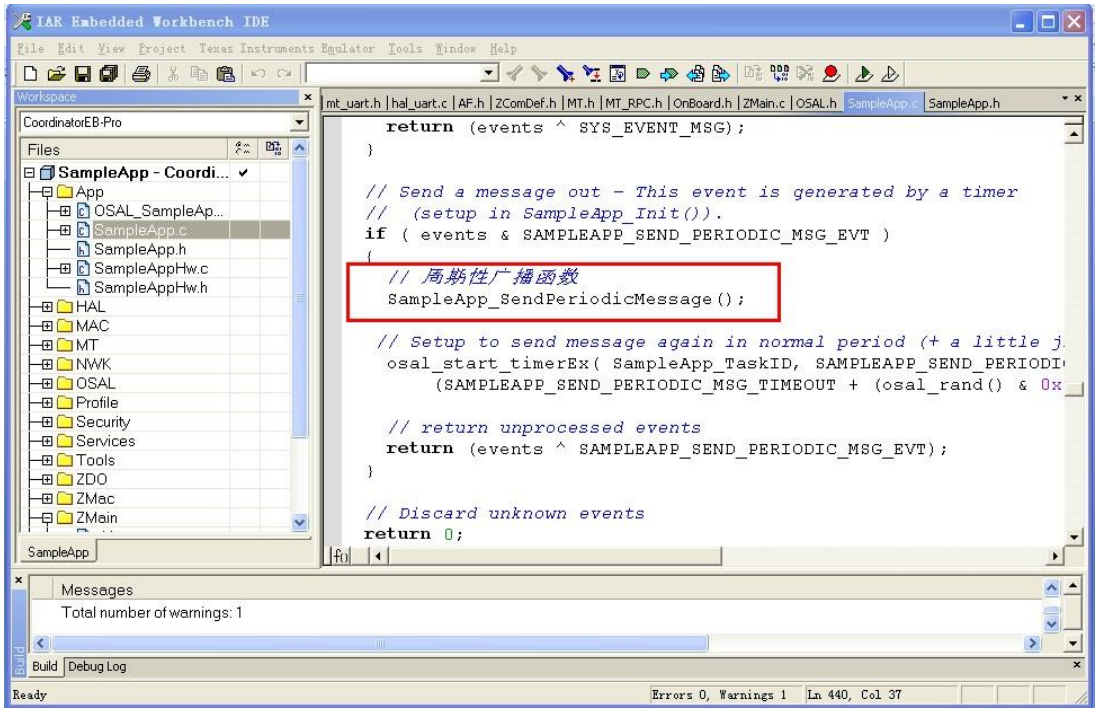


图 5- 73

在接收方面，默认接收 ID 就是刚定义的周期性广播发送 ID:

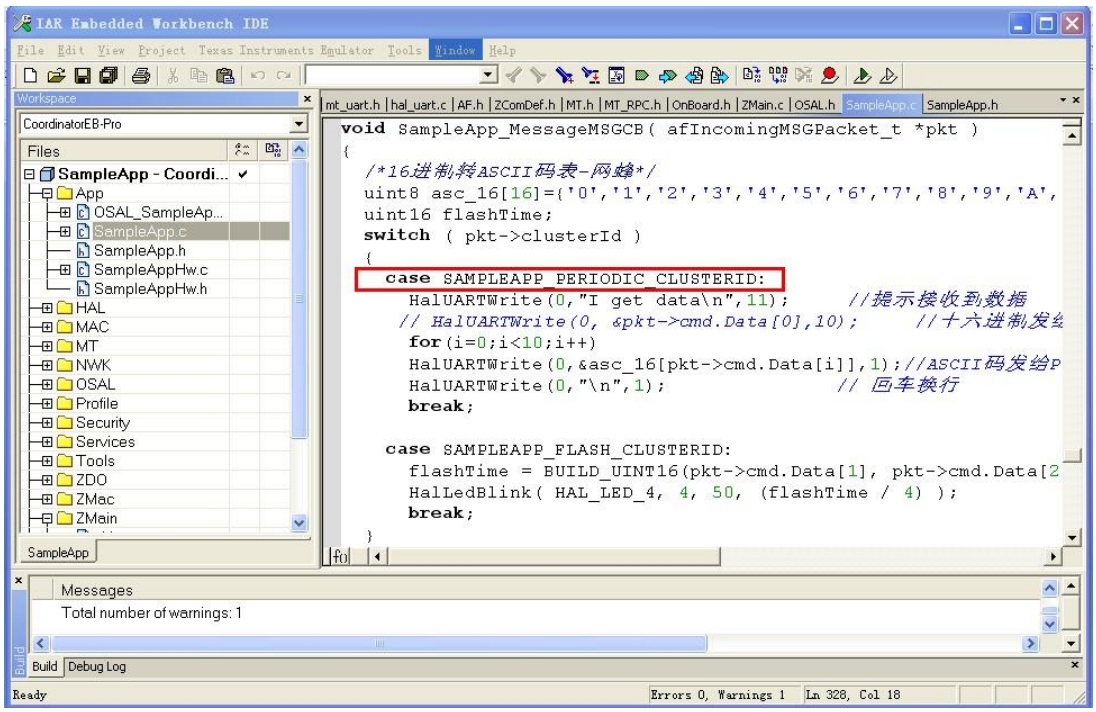


图 5- 74

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 5- 75 广播方式发送数据

通过我们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。

5.8. 实验七：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理。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。

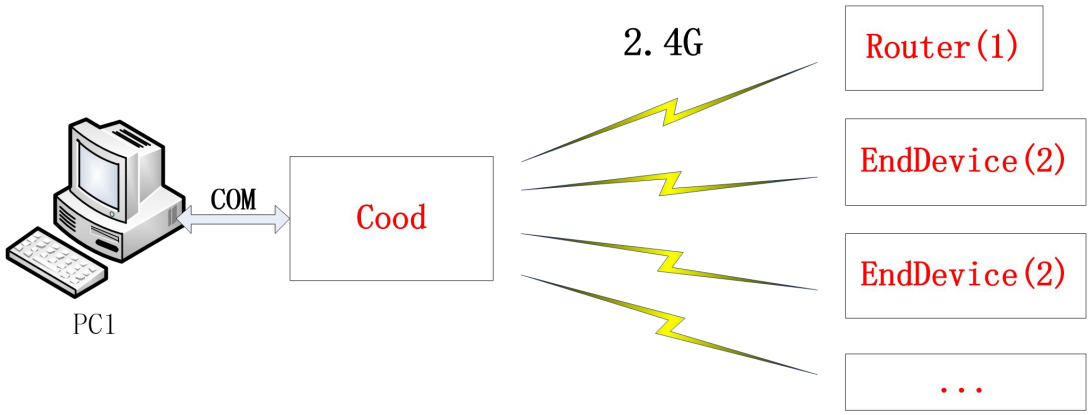


图 5- 76 网络管理系统框图

实现平台：ZigBee 节点 3 个以上。包括协调器、路由器、终端。

图 5- 77　ZigBee 实验相关设备

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，相关内容请参考点播章节内容。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。

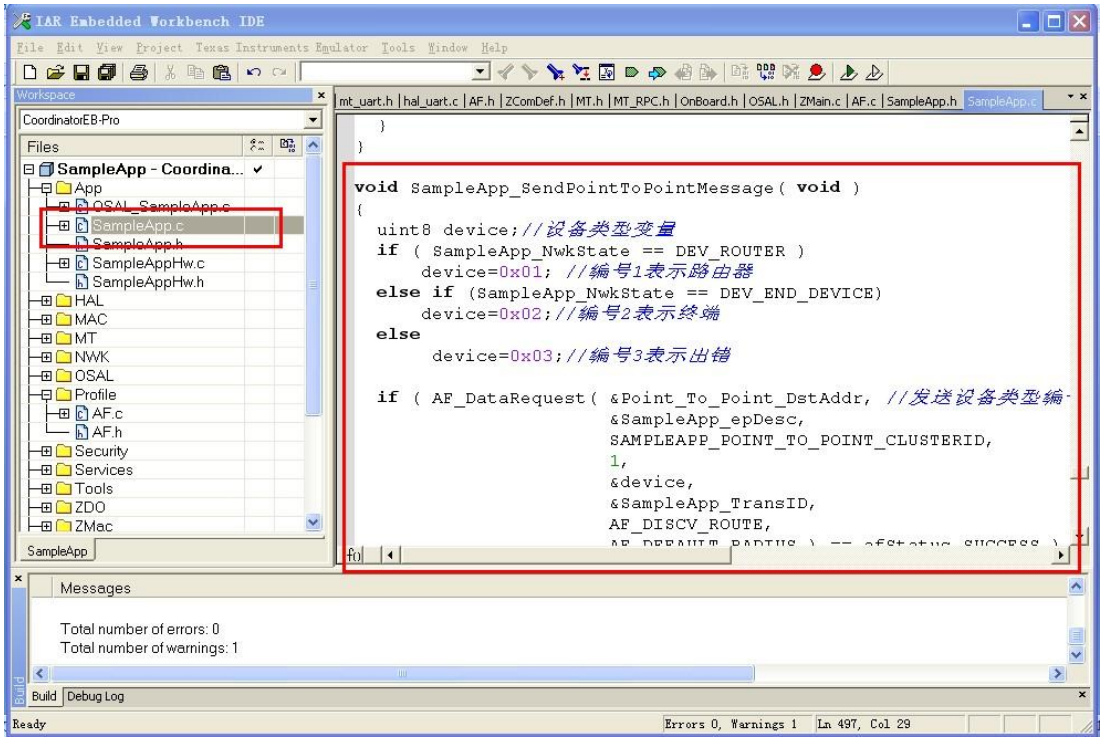


图 5- 78

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 5- 79所示：

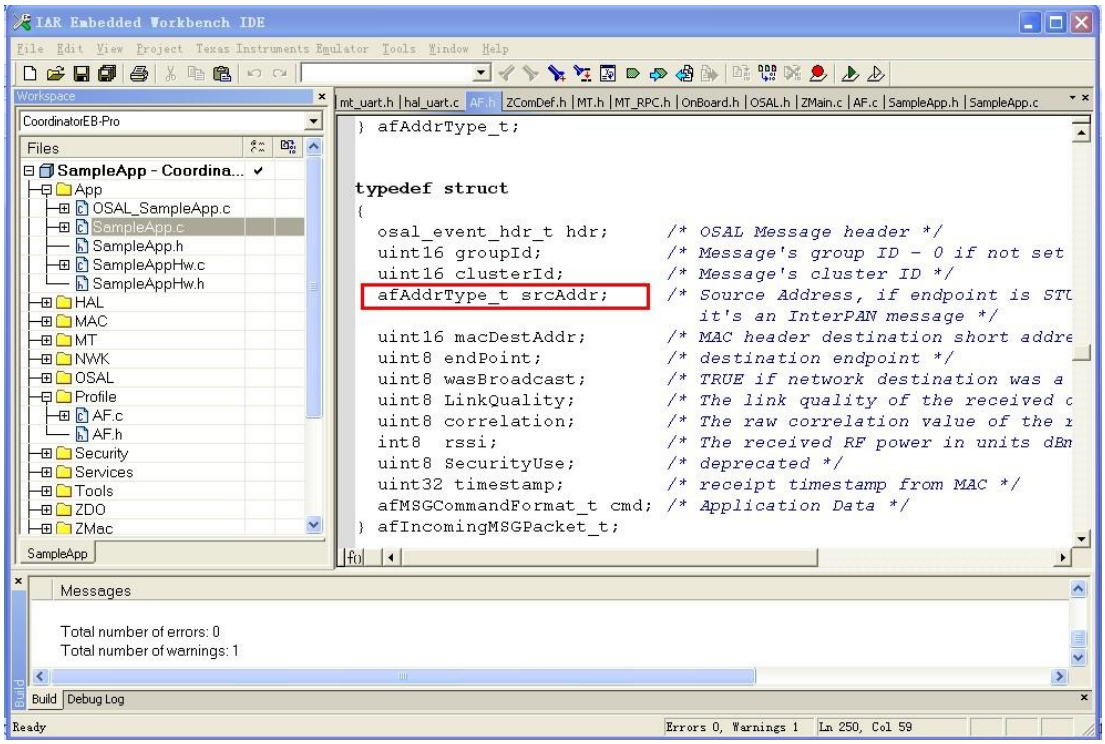


图 5- 79

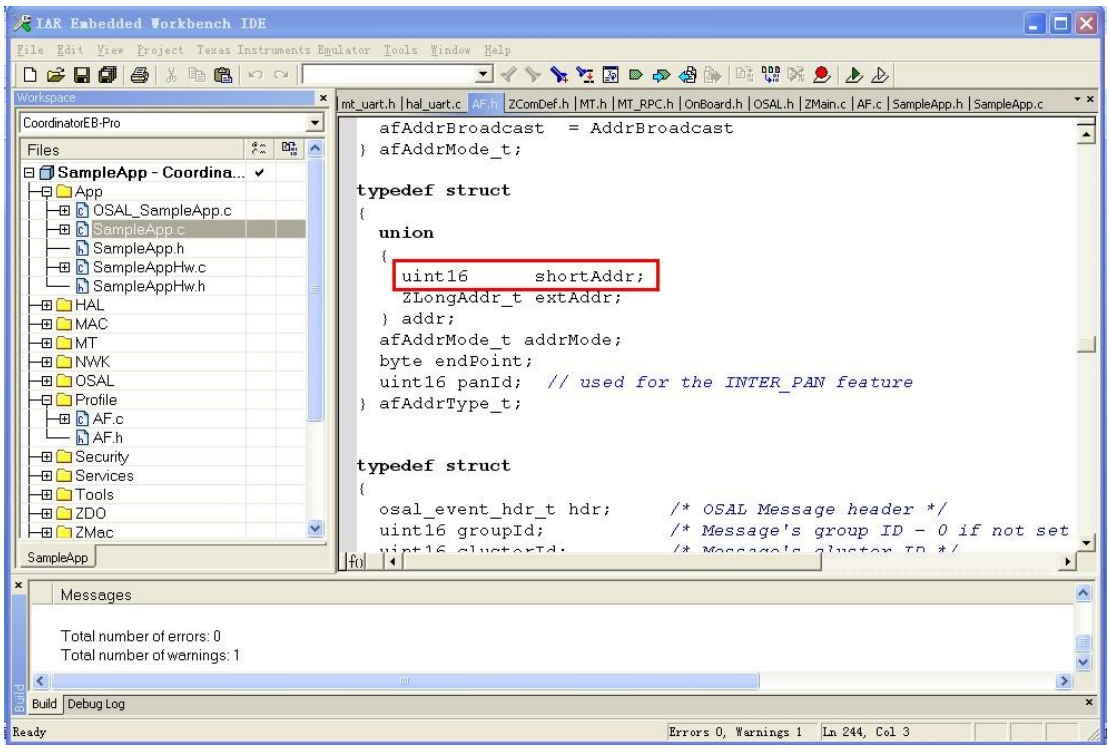


图 5- 80

我们可以在接收函数中点播 ID 加入下面代码：

```
    uint16  flashTime,temp;
      // 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;
```

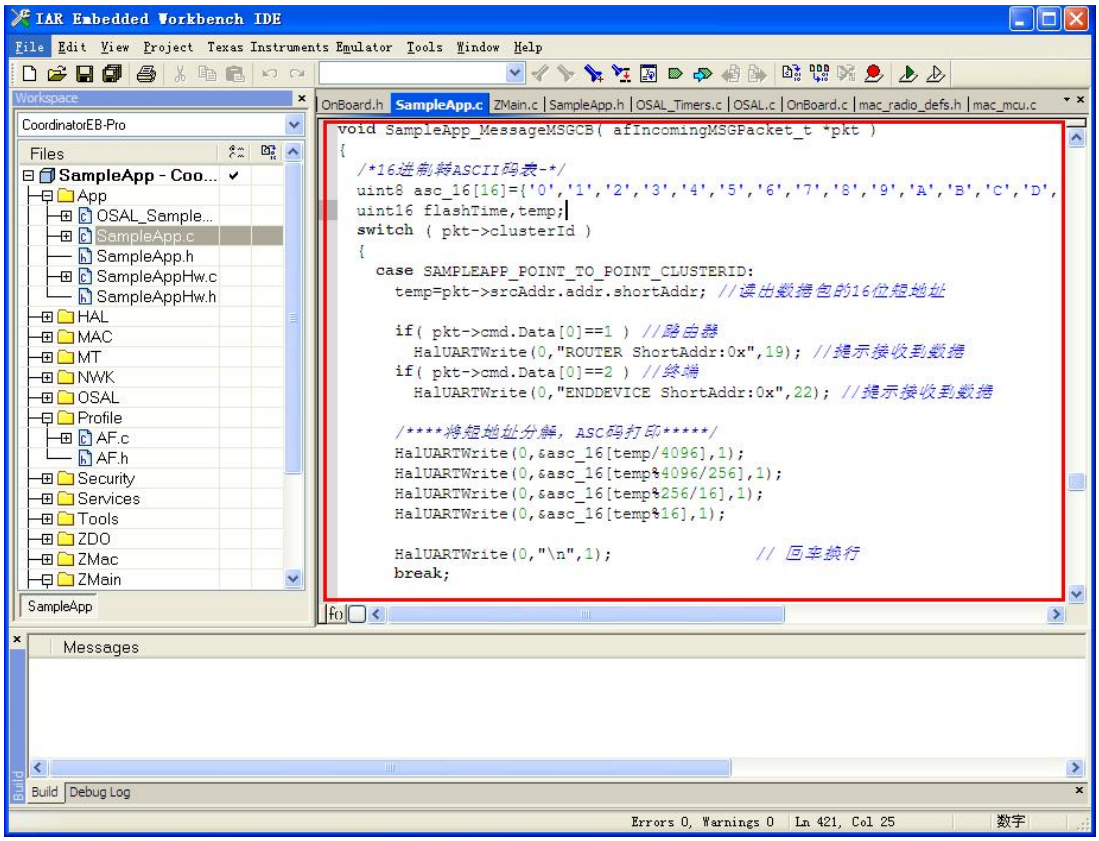


图 5- 81

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 5- 82

这里我们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。

第六章 传感器和执行器应用

1.1. 实验一：温湿度度传感器

前言：实现温湿度传感器 DHT11 的采集实验。

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。

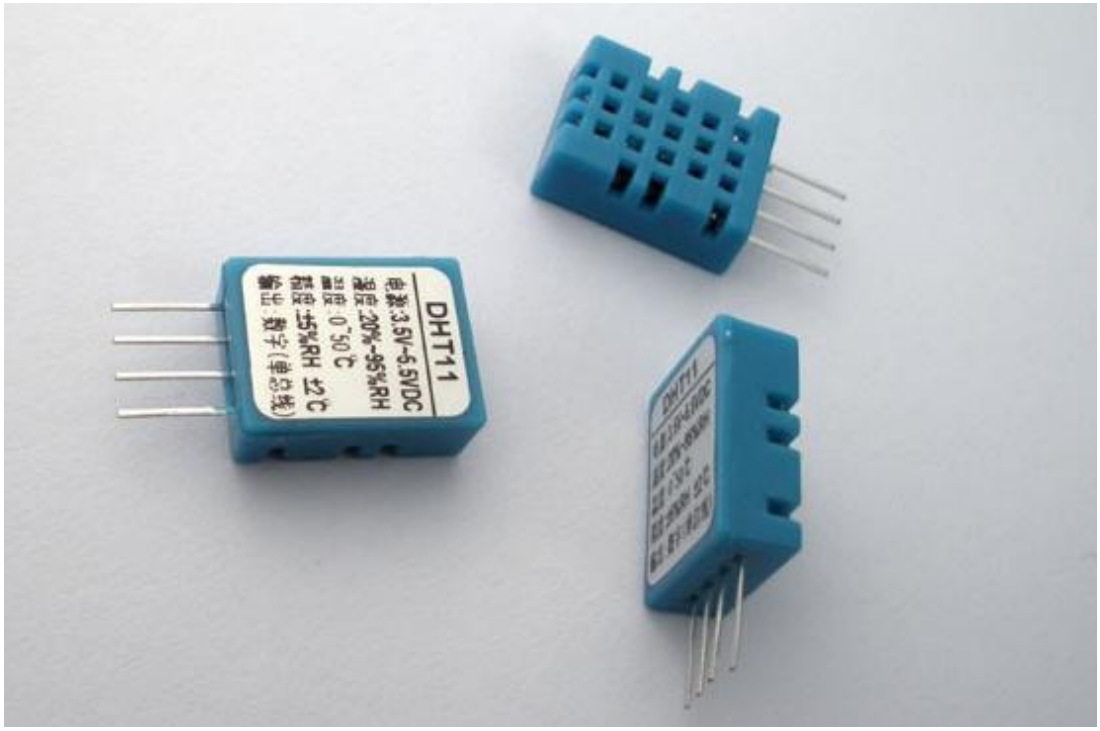


图 1- 1 温湿度传感器 DHT11

实验平台： ZigBee 传感器节点；

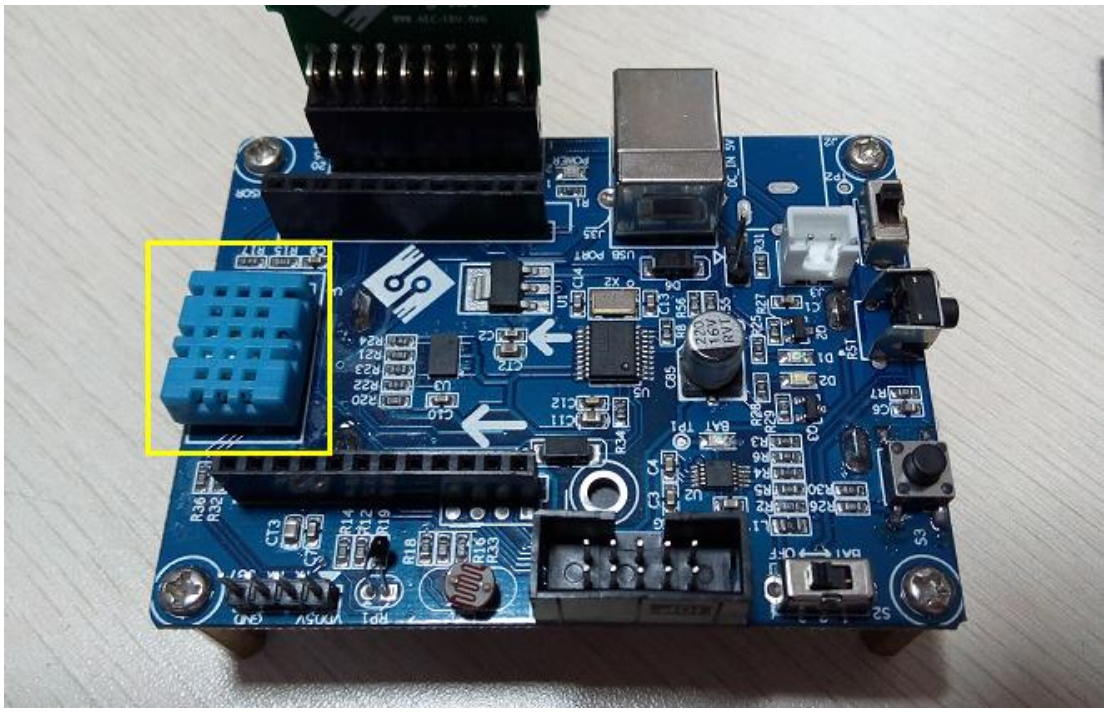


图 1- 2 ZigBee 协调器和传感器节点；

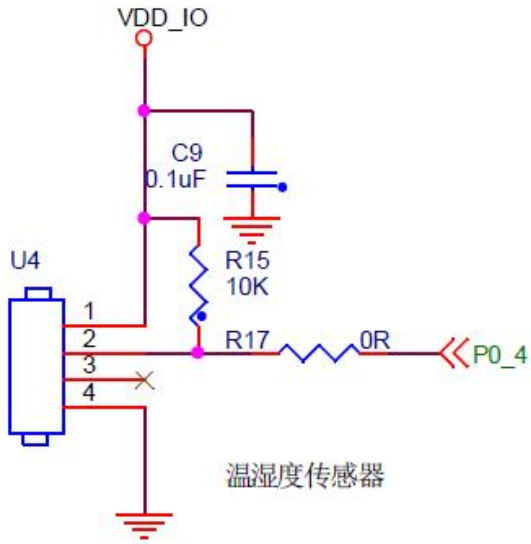


图 1- 3 DHT11 电路图

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显示当前温湿度。

实验讲解：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）

```
1. /*************************************************************
2. *函数功能 ：主函数                                          *
3. *入口参数 ：无                                              *
```

```
4.  *返 回 值 : 无                                                    *
5.  *说    明 : 无                                                    *
6.  ************************************************************/
7.  void main(void)
8.  {
9.      CLKCONCMD &= ~0x40;                //晶振
10.     while (CLKCONSTA & 0x40);      //等待晶振稳定
11.     CLKCONCMD &= ~0x47;                //TICHSPD128 分频，CLKSPD 不分频
12.     SLEEPCMD |= 0x04;          //关闭不用的 RC 振荡器
13.     P1DIR |= 0x01;
14.
15.     initUARTSEND();
16.     Delay(50000);
17.    while (1)
18.    {
19.      Read_DHT11();  //调用温湿度读取子程序
20.      Delay(60000);  //循环采样的延时，读取模块数据周期不易小于 2S
21.      sprintf(str, "%dC, %dH\n", T_data_H, RH_data_H);
22.      UartTX_Send_String(str, 16);
23.      P1_0 ^= 1;
24.    }
25. }
```

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 1-4 所示：

图 1- 4

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。

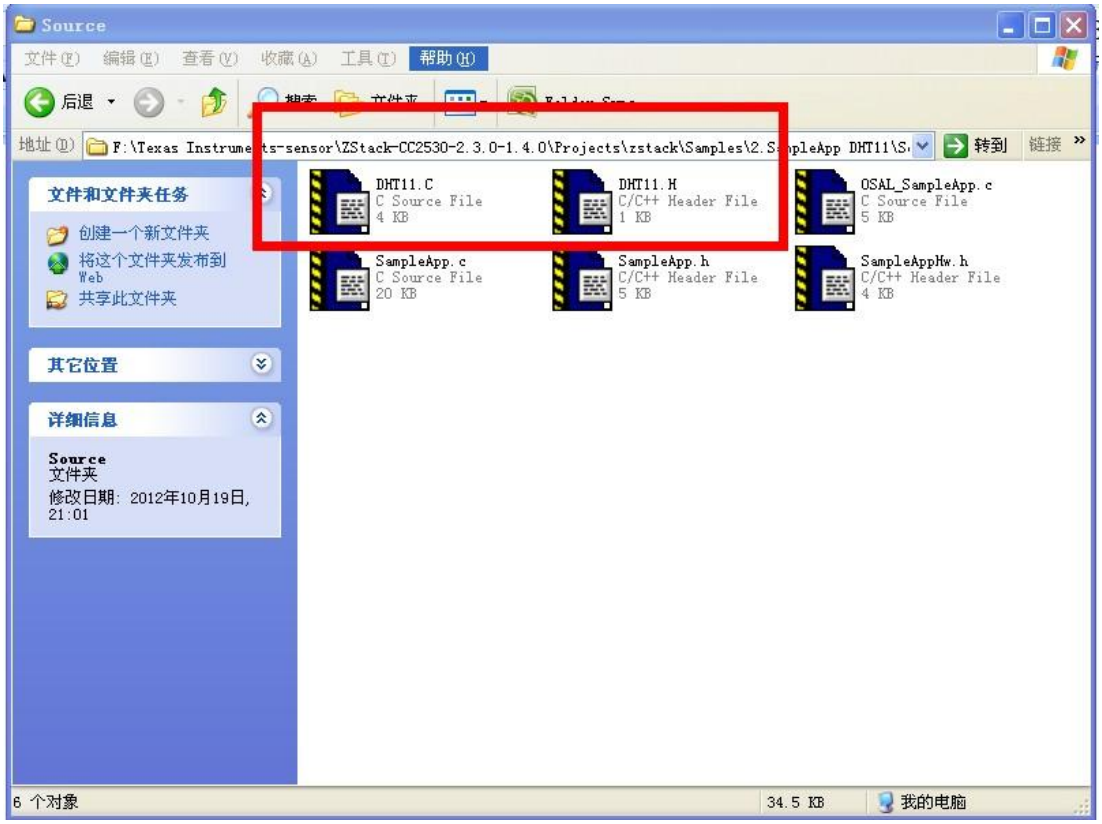


图 1-5

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

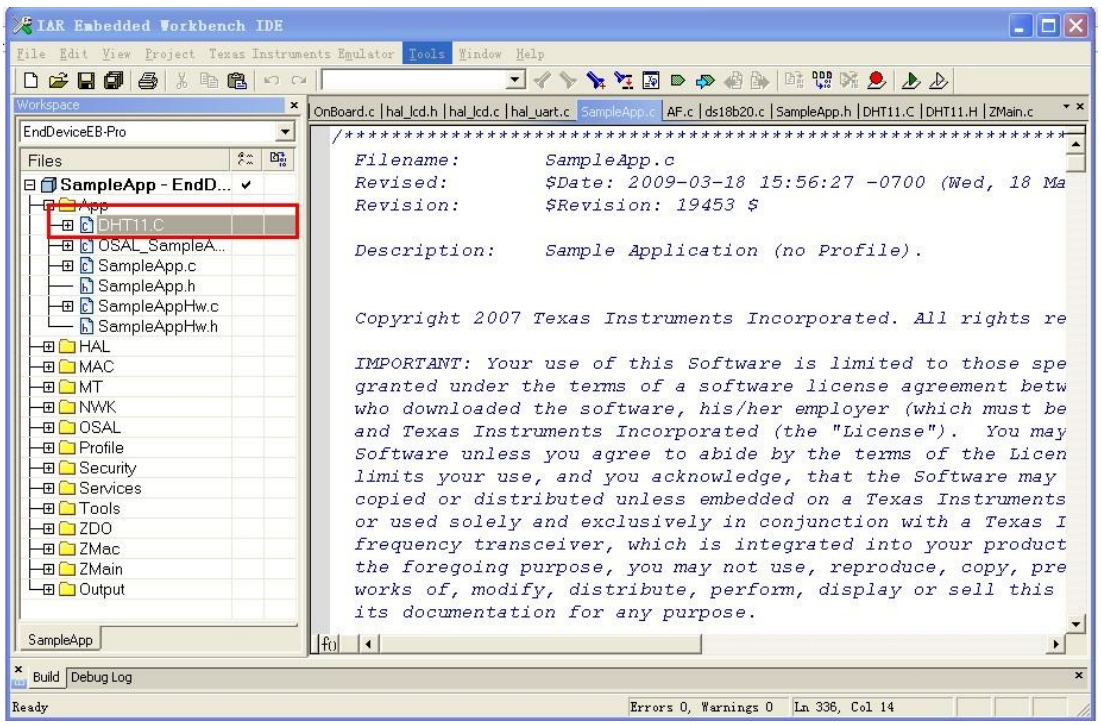


图 1-6

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。

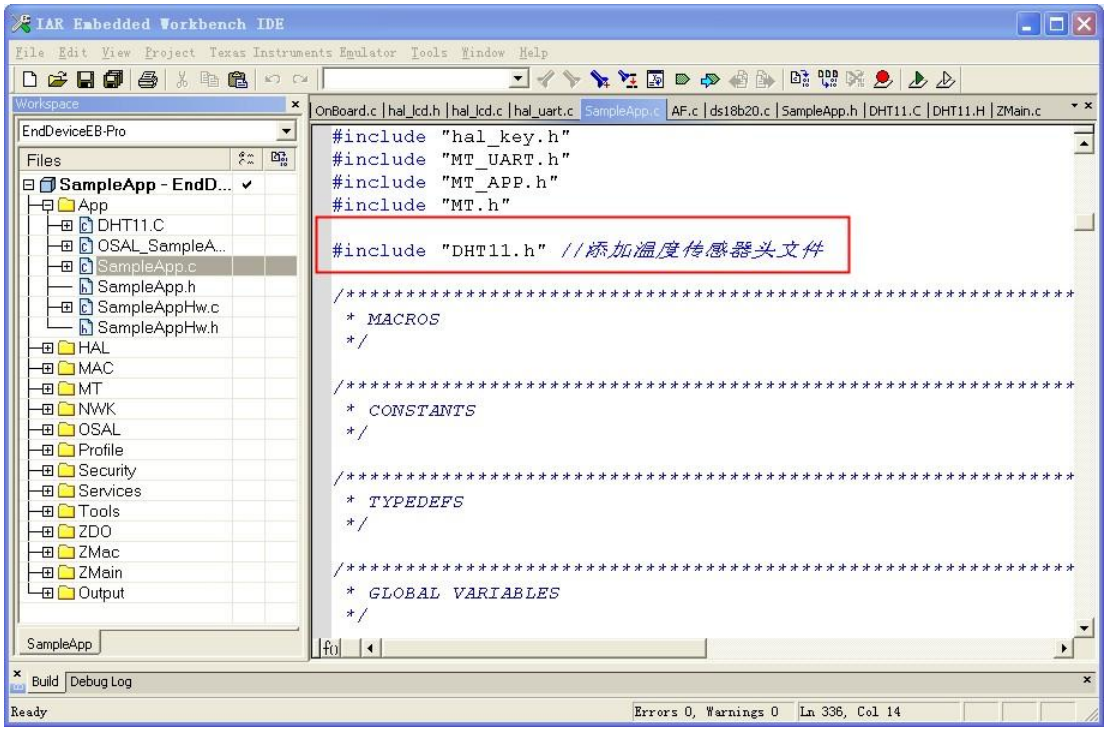


图 1-7

4）初始化传感器引脚

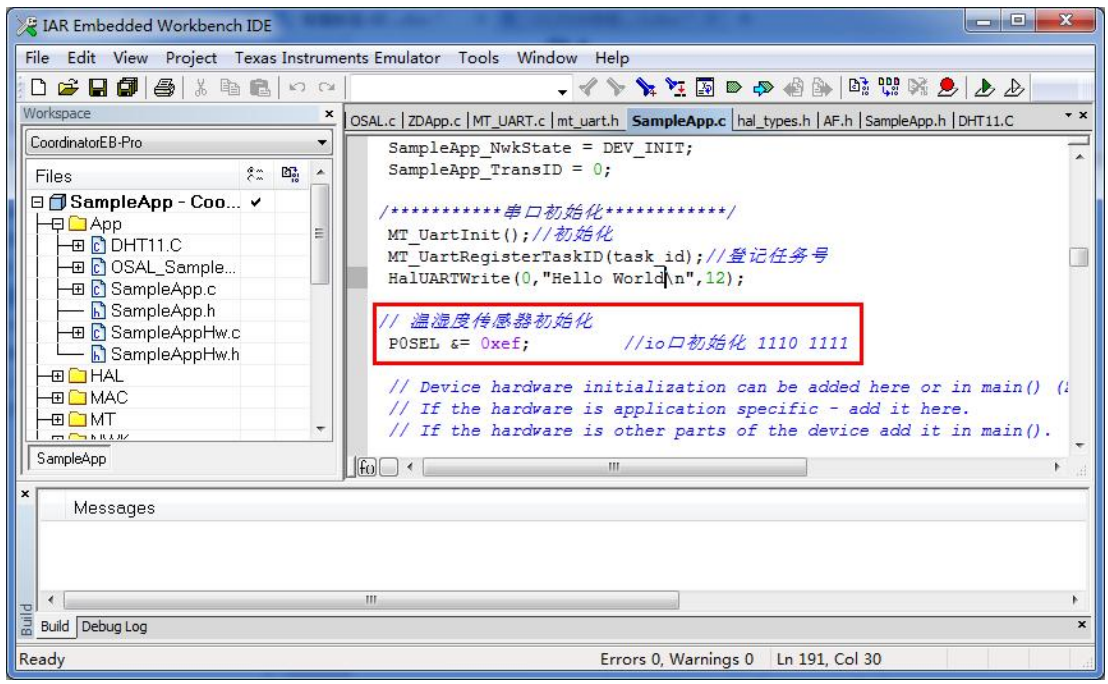


图 1- 8

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 1-9 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********/
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*********/
HalLcdWriteString("Temp: humidity:", HAL_LCD_LINE_3 );//LCD 显示
            HalLcdWriteString( T, HAL_LCD_LINE_4 );//LCD 显示
```

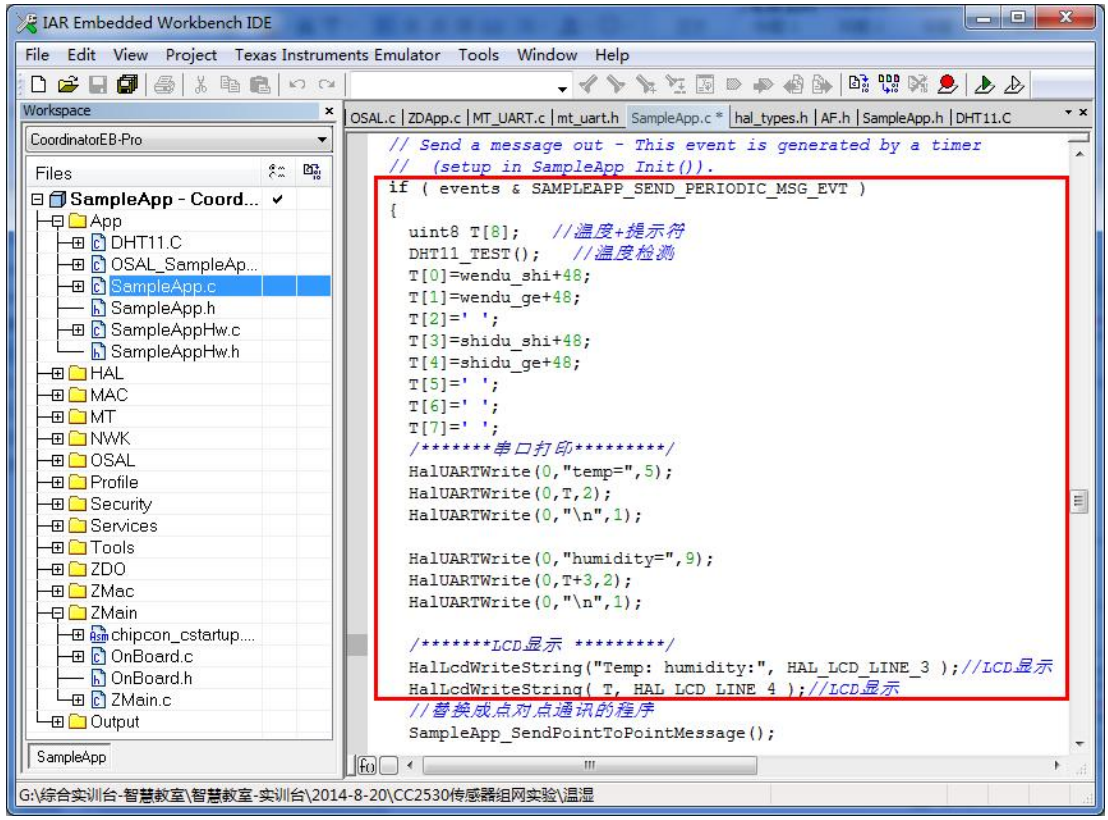


图 1- 9

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。

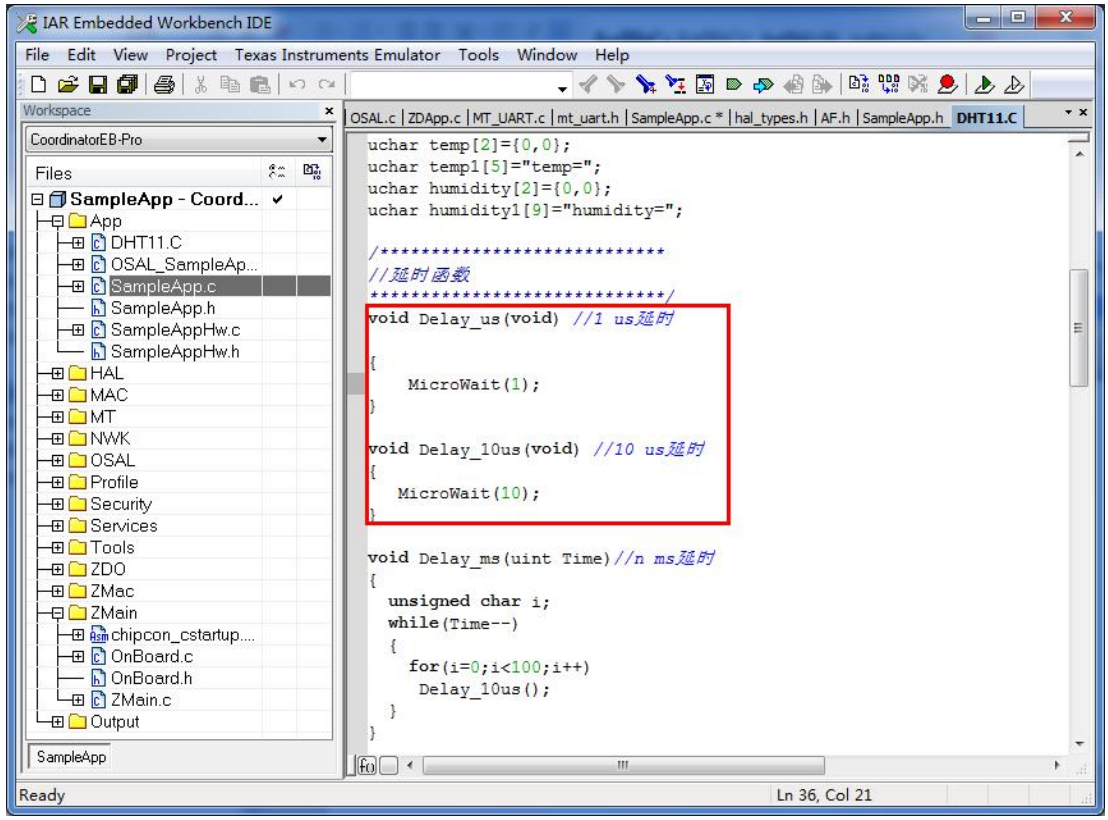


图 1-10

同时要包含 #include"OnBoard.h" 。

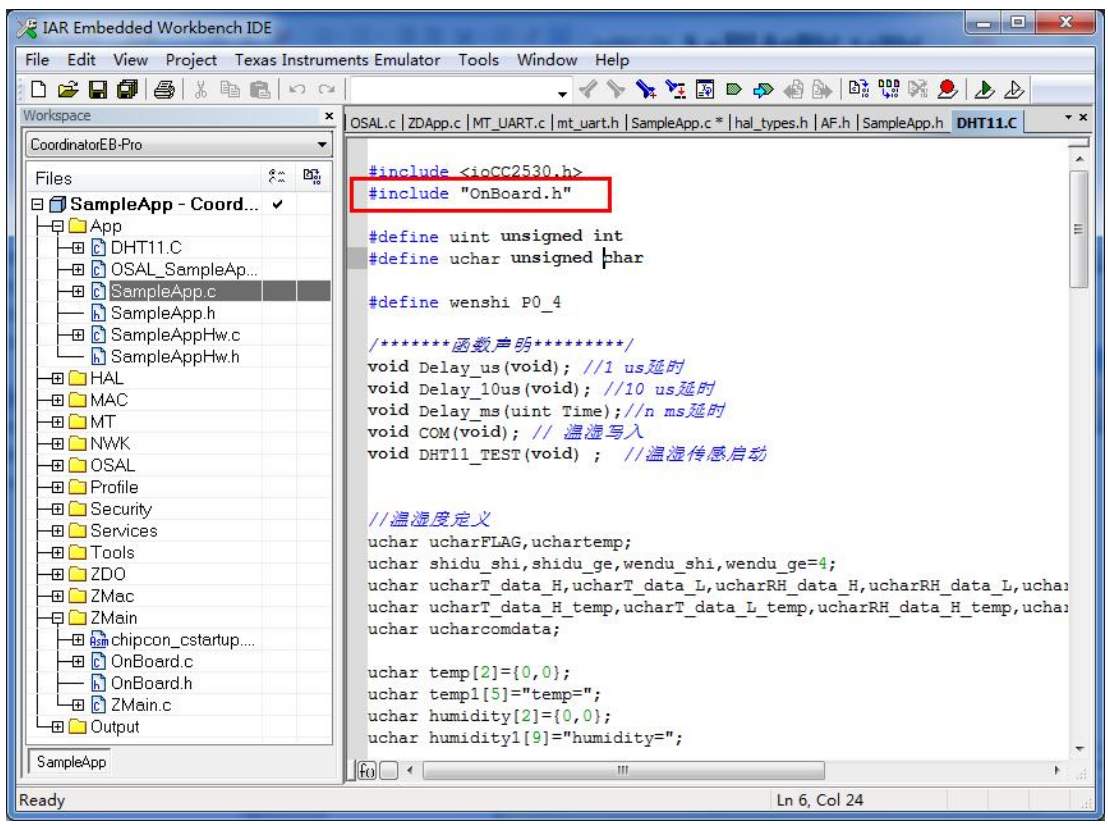


图 1-11

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
```

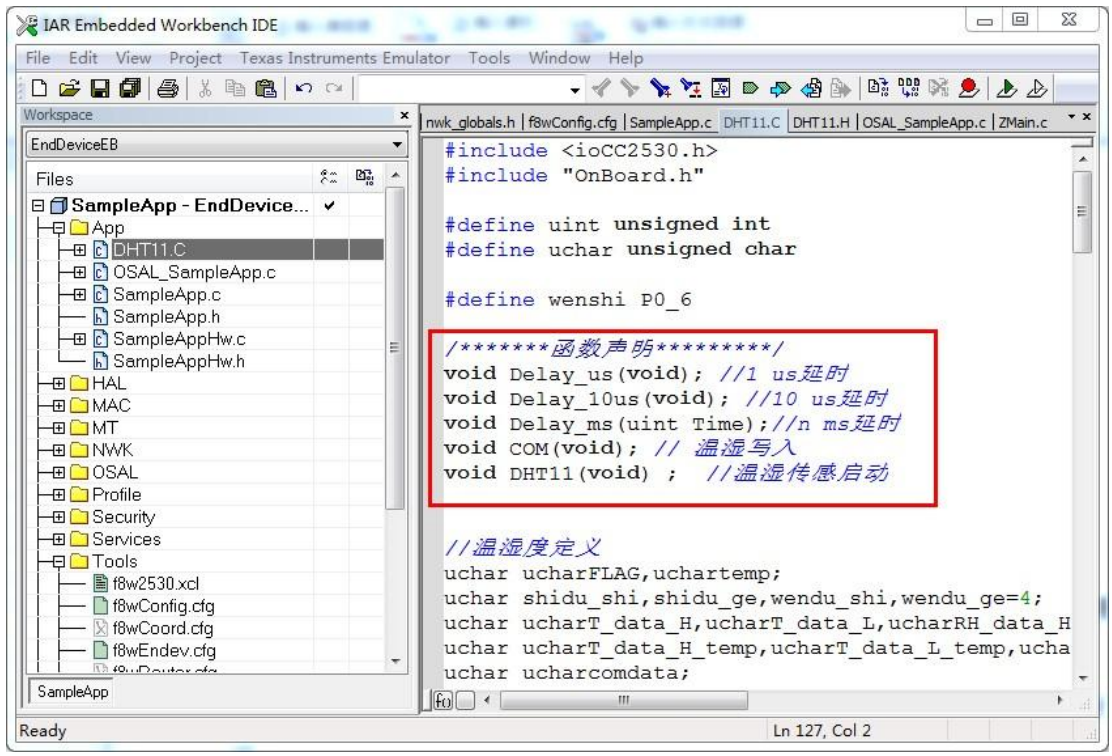


图 1- 12

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.
    }
  }
```

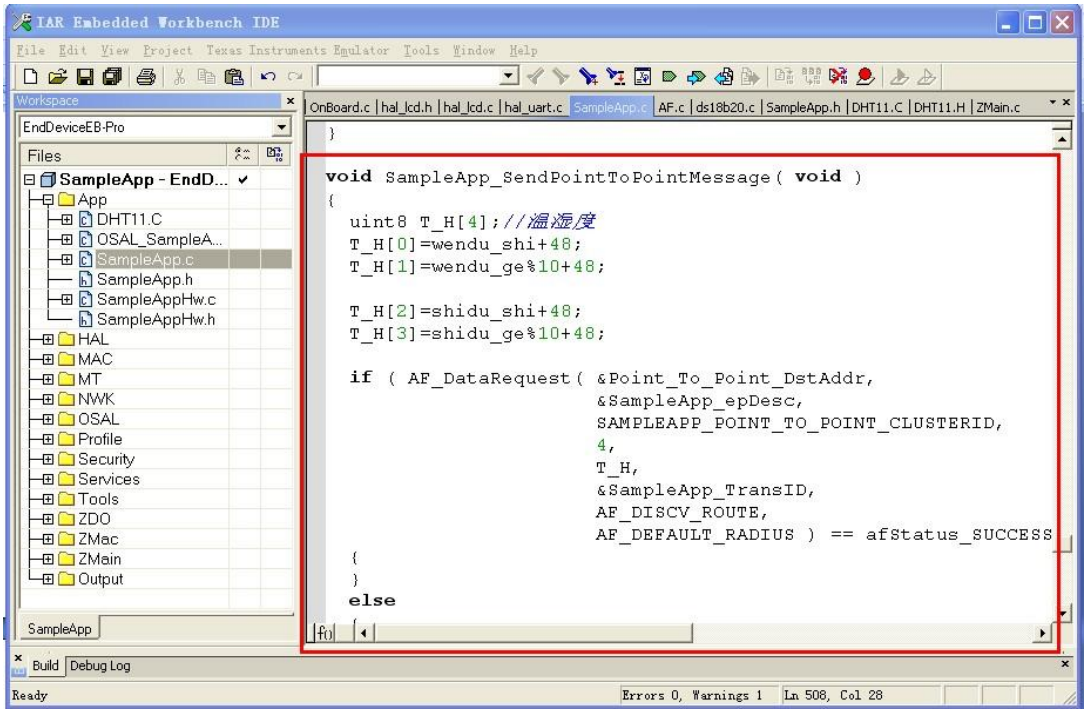


图 1-13

协调器代码如下，如图 6-13 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;
```

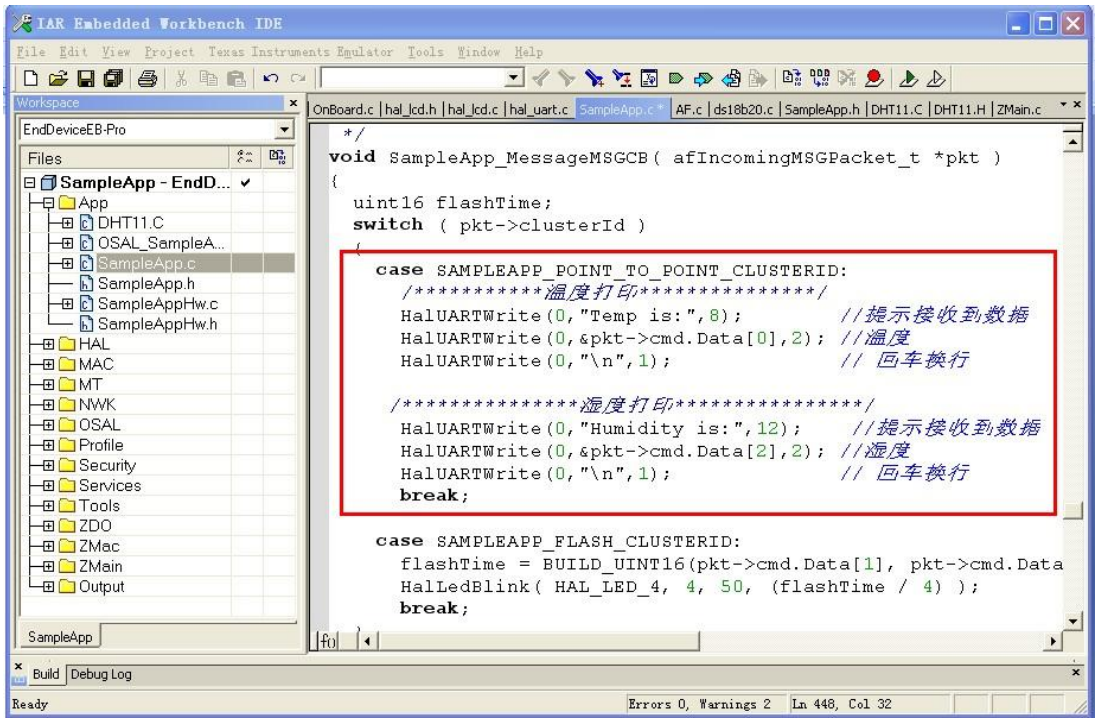


图 1- 14

实验结果及图片：

观察终端设备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，串口打印出温湿度传感器信息。如图 6-15 所示：

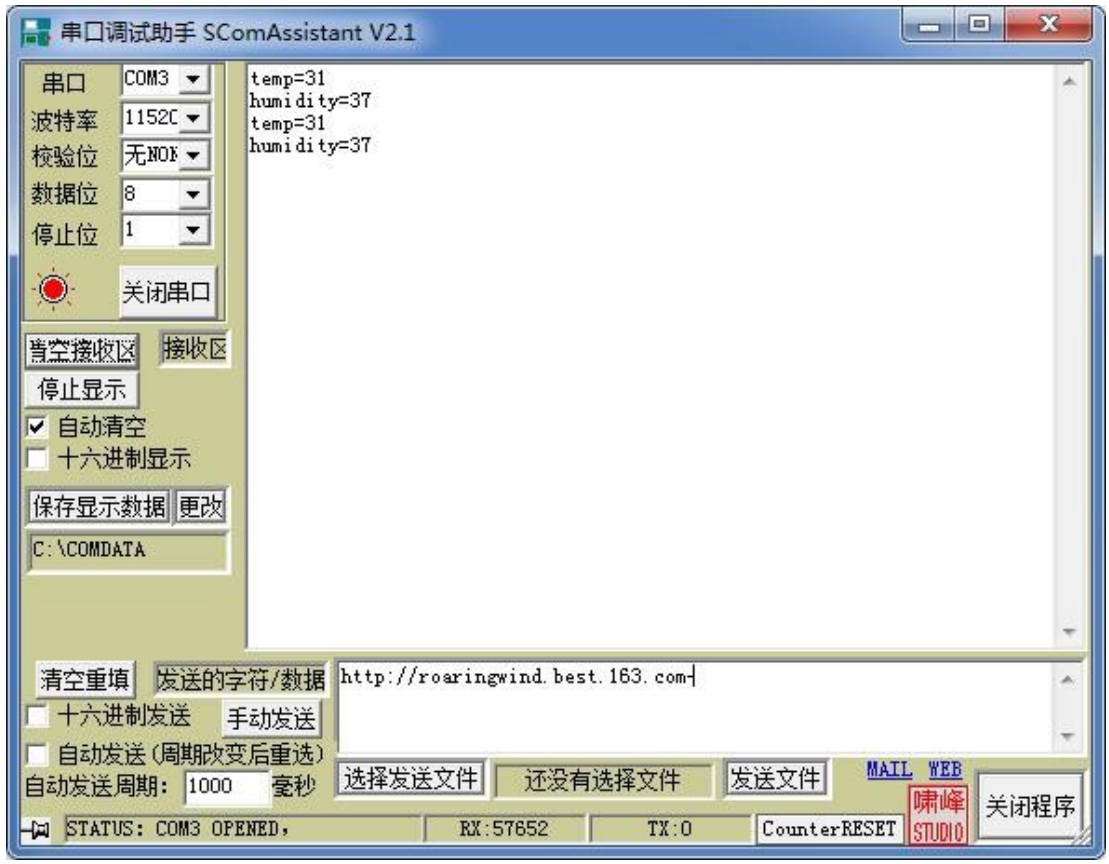


图 1-15

同时协调器收到终端发送来的信息：

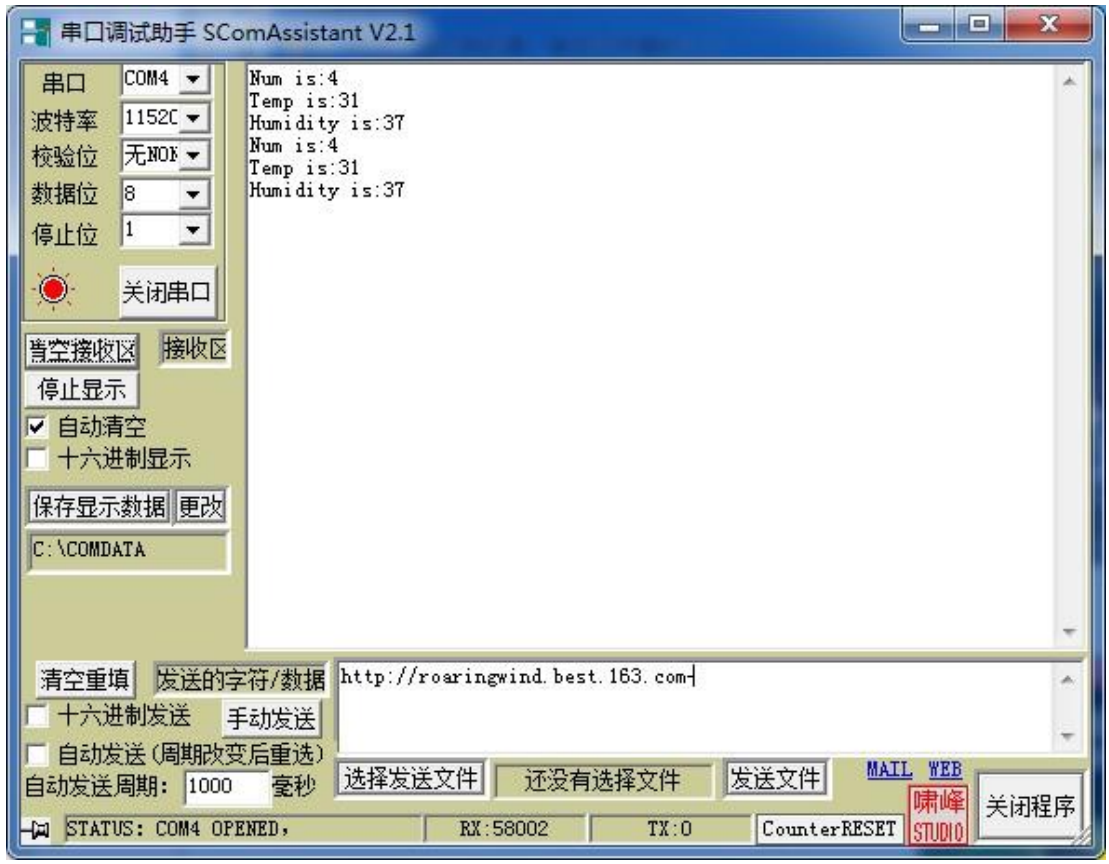


图 1- 16

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。

1.2.实验二：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，通过光敏传感器实现对光的强度的检测和采集。

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。

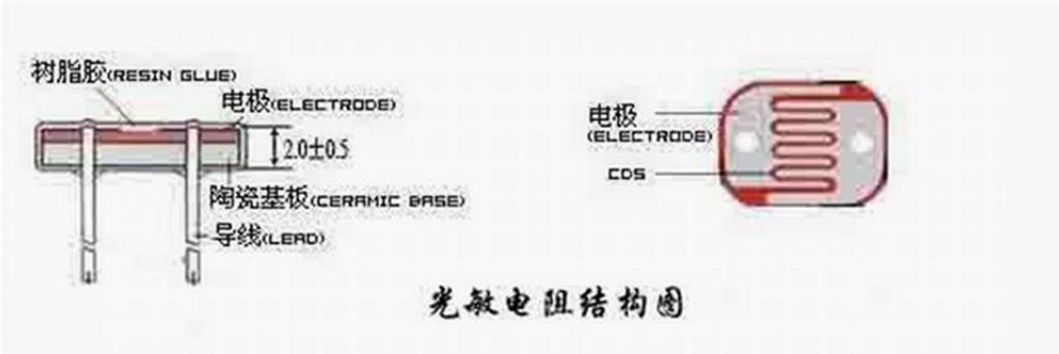


图 2- 1 光敏电阻

实现平台：ZigBee 传感器节点

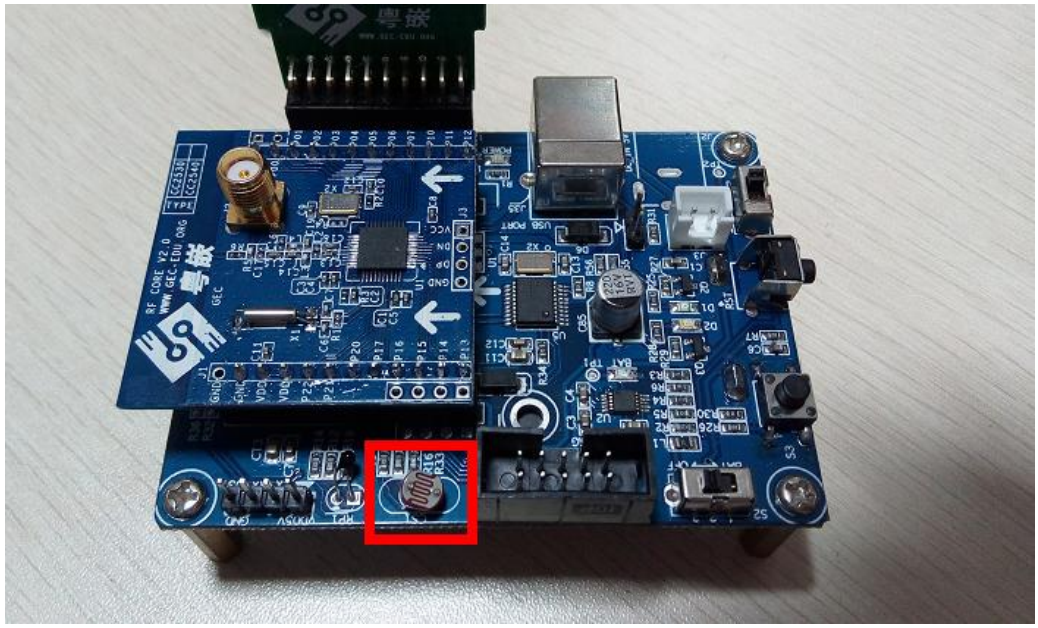


图 2- 2　ZigBee 传感器节点

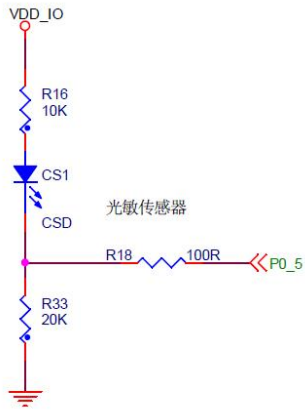


图 2- 3　光敏传感器电路

实验现象：光敏电阻电路通过检测外界光线强度的情况，将信息串口打印出来。

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：

```
1.  #include "ioCC2530.h"
```

```
#include "stdio.h"


/******* main 函数 ********/
void main(void)
{
  char i;
  char str[16];
  InitClock();
  InitUART0();
  while (1)
  {
    uint16 AvgValue = 0;
    for (i = 0 ; i < 64 ; i++)
    {
      AvgValue += readAdc(ADC_CHNN);
      AvgValue >>= 1;
    }
    sprintf(str, "%d\n", AvgValue);
    UartTX_Send_String(str, 6);    // UART 发送 ADC
    Delay(50000);       // 延时 1s
  }
}

```

我们来看主函数：

InitClock();　：初始化系统时钟

InitUART0();：初始化串口 0。

while(1)：在大循环中，不断检测光照强度。

readAdc();：读取 ADC 采集到的数据

Uart_Send_String()：信息通过串口打印

上面的代码实现了光线的强度的变化，通过串口打印出相应的 ADC 转换光的强度数值。实验现象如图 6-20 所示：

图 2- 4

二：将程序添加到协议栈代码中

光敏电阻是采取 ADC 进行检测。所以在协议栈里检测程序比较简单。我们只需要配置好 ADC 通道 5，然后周期性检测就可以了。

1） 打开例程 SampleApp.eww 工程，打开 SampleApp.c 文件。

定义光敏传感器 ADC 通道和添加头文件

```
//光敏通道定义
#include "stdio.h"
#include "LightSensor.h" //添加光照传感器头文件
#define  ADC_CHNN    0x05  //选用通道 5  //光敏通道为 P1.5 口控制
```

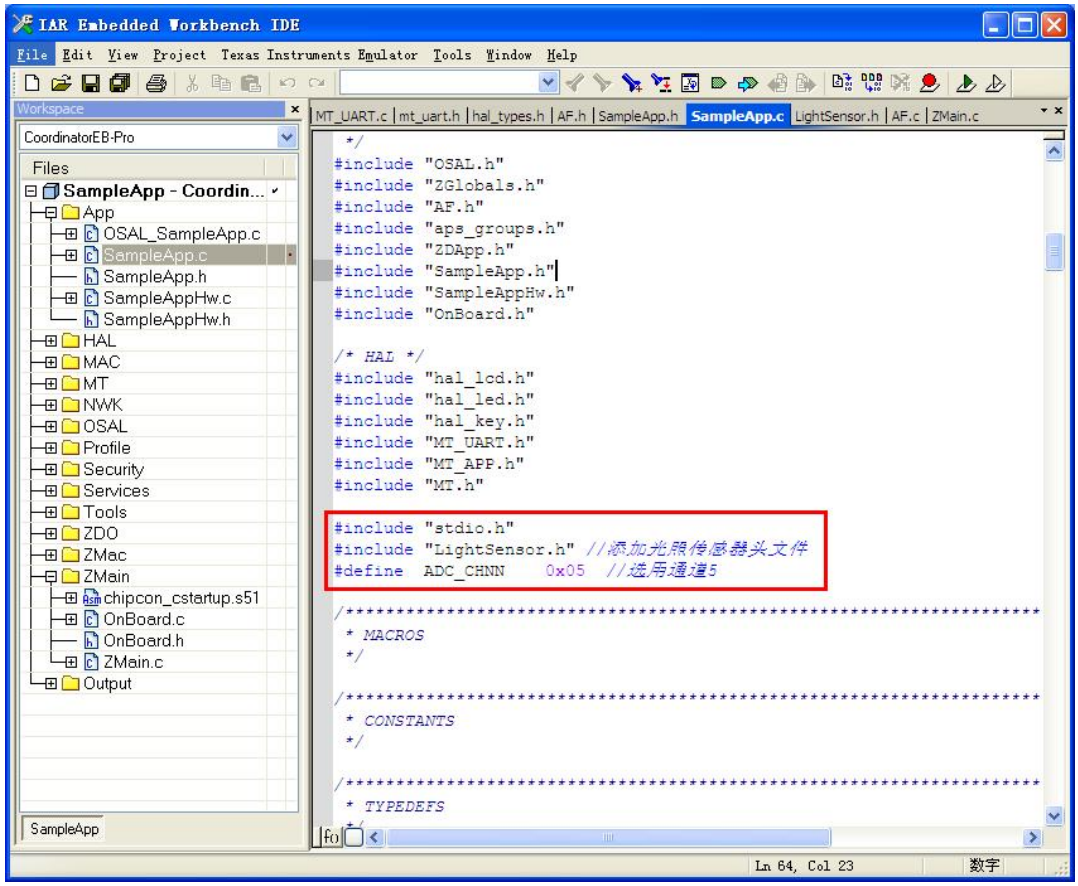


图 2- 5

2） 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。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
```

3） 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 str[16];
  uint16 AvgValue ;
  AvgValue = readAdc(ADC_CHNN);
  sprintf(str, "%d\n", AvgValue);
```

```
    if ( AF_DataRequest( &Point_To_Point_DstAddr,
                         &SampleApp_epDesc,
                         SAMPLEAPP_POINT_TO_POINT_CLUSTERID,
                         16,
                         str,
                         &SampleApp_TransID,
                         AF_DISCV_ROUTE,
                         AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
    {
    }
    else
    {
      // Error occurred in request to send.
    }
  }
```

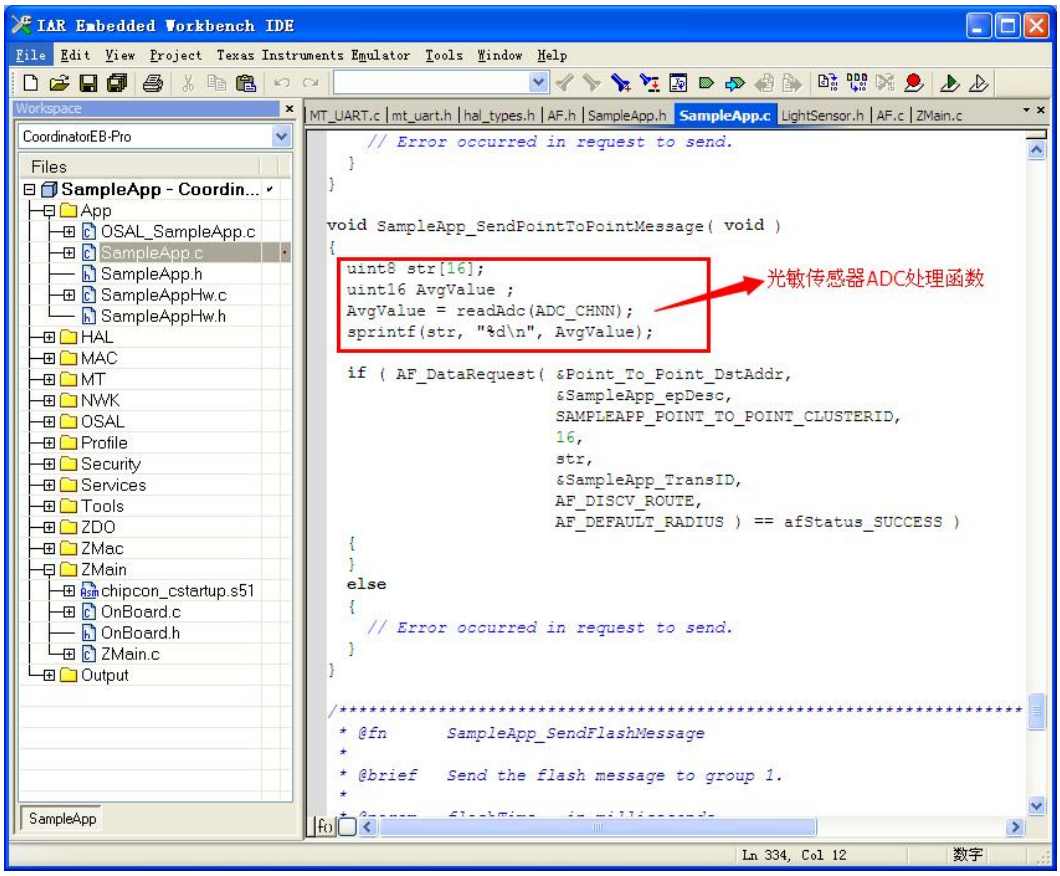


图 2- 6

4） 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:
        HalUARTWrite(0,"Num is:",7);          //提示接收到数据
        HalUARTWrite(0,&pkt->cmd.Data[0],16); //光照
        HalUARTWrite(0,"\n",1);               // 回车换行
      break;
```

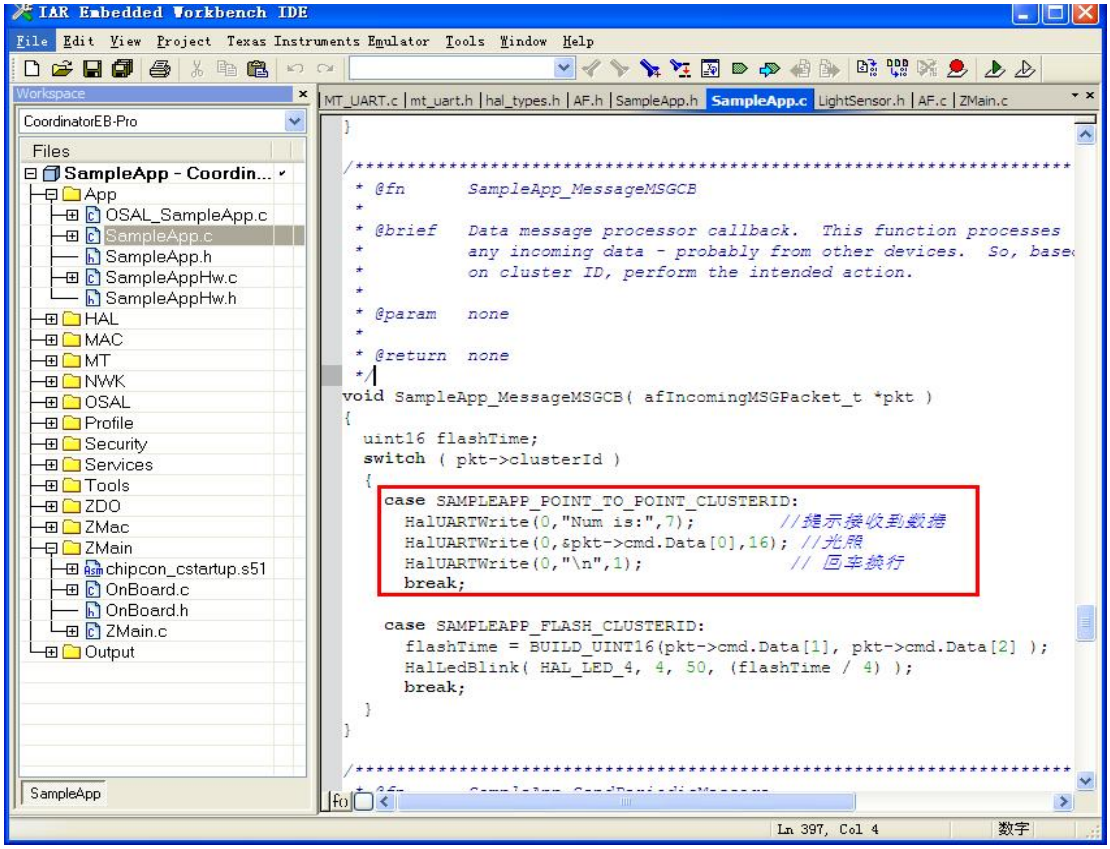


图 2- 7 接收判断

实验现象：下载程序到终端（带光敏传感器）和协调器，协调器收到的信息：

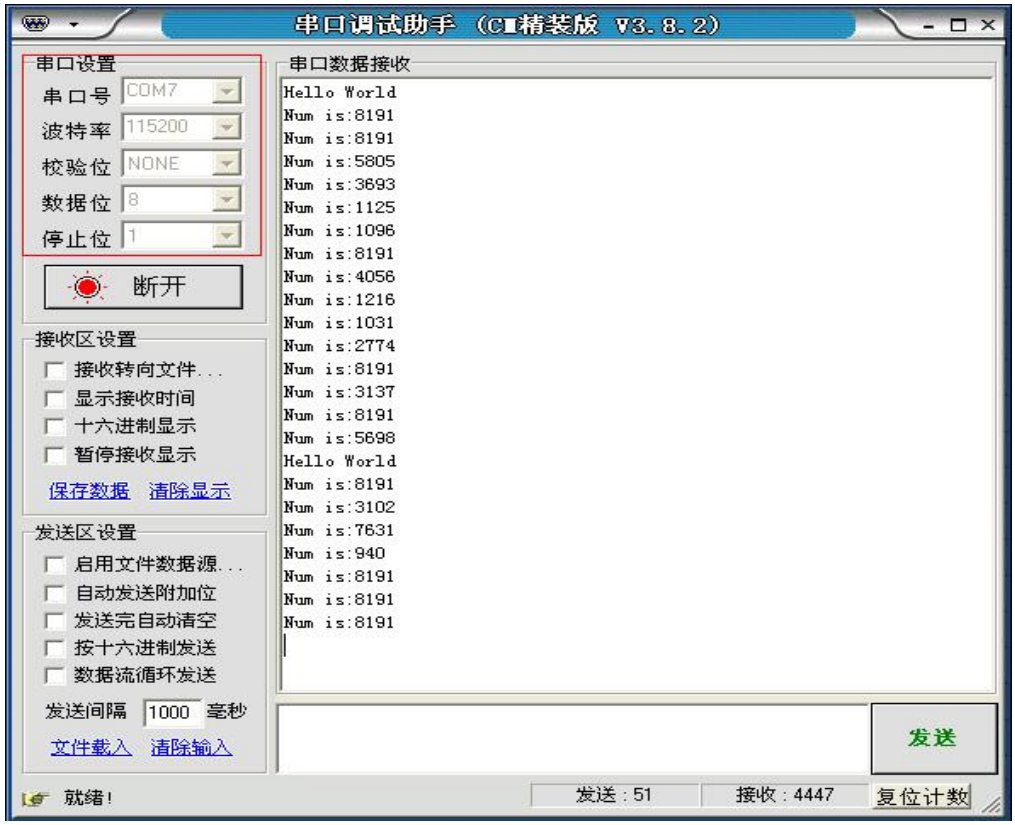


图 2- 8

1.3. 实验三：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，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 3-1　MQ-2 烟雾传感器

实现平台：　ZigBee 传感器节点；

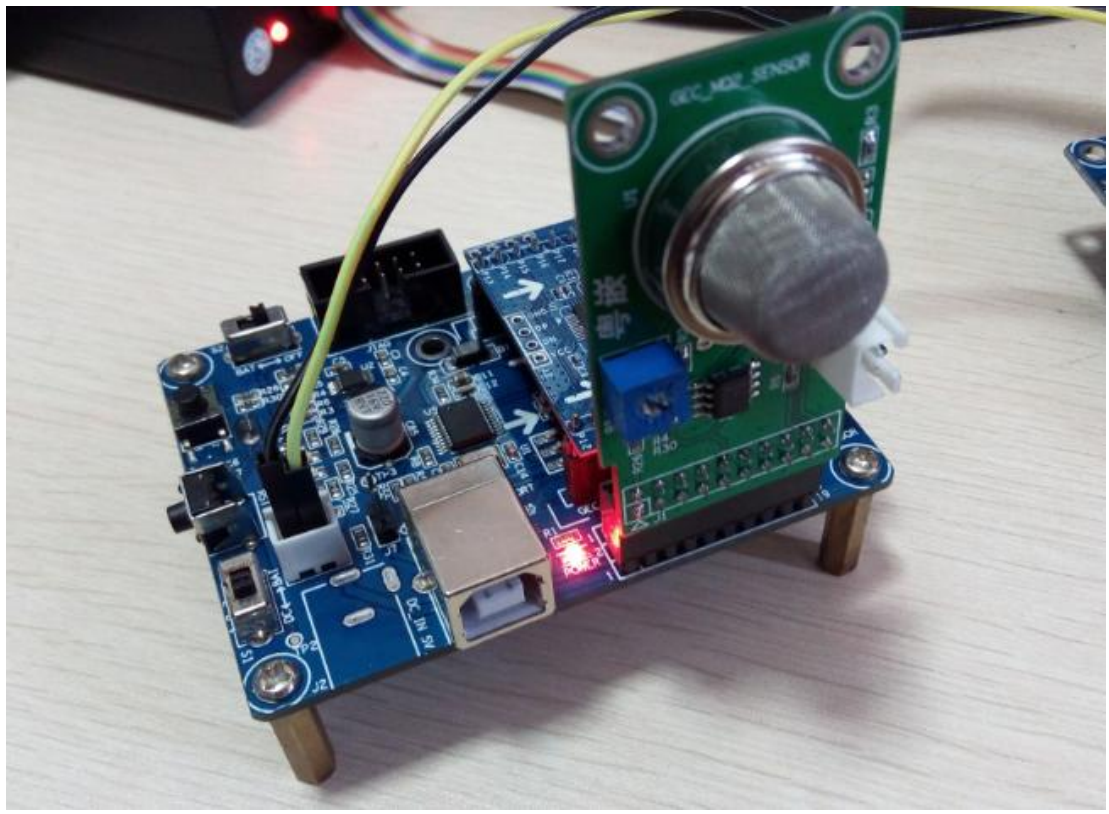


图 3- 2　ZigBee 传感器节点

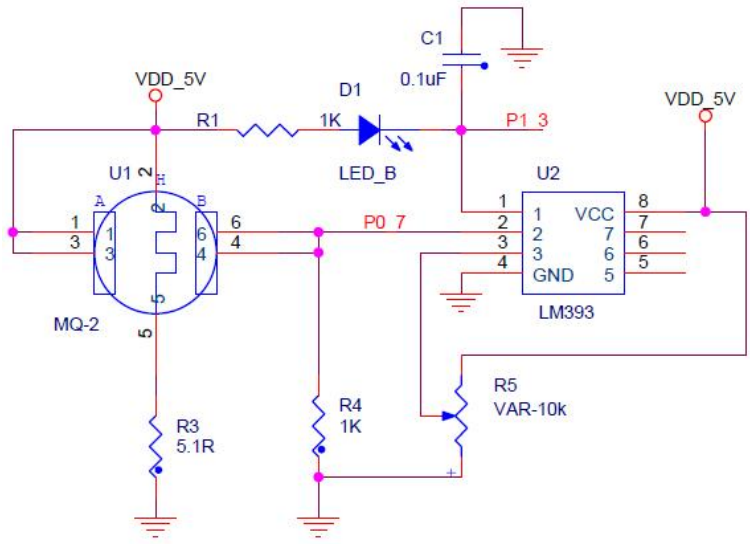


图 3- 3 烟雾传感器硬件电路

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. /*  Zigbee 学习例程
3. /*例程名称：烟雾传感器
4. /*描述：通过烟雾传感器，通过 LED1 指示
5. *****************************************/
6. #include <ioCC2530.h>

7. #define uint unsigned int
```

```
8. #define uchar unsigned char


9.  uint8 BeepFlag = 0;
10.    /**********************************
11.     * 烟雾为中断触发方式
12.    **********************************/
13.    void InitSmokeINT(void)
14.    {
15.      P1DIR &= ~0x08;    //烟雾接口 P1_3 为输出
16.      P1_3 = 0;
17.      P1INP |= 8; //上拉
18.      P1IEN |= 8;    //P1.3 设置为中断方式
19.      EA = 1;
20.      IEN2 |= 0X10;    // P1 设置为中断方式;
21.      P1IFG |= 0x00;    //初始化中断标志位
22.    }
23.
24.    /**********************************
25.     * 初始化程序,将 P1.1、P0_6
26.     *         定义为输出口，并将蜂鸣器、烟雾接口和
27.     *         LED 灯初始化为灭
28.    **********************************/
29.    void InitIO(void)
30.    {
31.        P1DIR |= 0x02;
32.        P0DIR |= 0x40;
33.        P0_6 = 0;
34.        P1_1 = 1;
35.        P1_3 = 0;
```

```
}

/*****************************
 * @brief   烟雾中断服务
*****************************/
#pragma vector = P1INT_VECTOR
__interrupt void P1_ISR(void)
{
  if(P1IFG > 0)             //烟雾中断
  {
    P1IFG = 0;
    WaitMs(25);
    if (P1IFG == 0) {
    BeepFlag = 1;
    }
  }
  P1IF = 0;                 //清中断标志
}
/*****************************
  主函数
*****************************/
void main(void)
{
   InitIO();
   InitSmokeINT();   //初始化烟雾中断
   while (1) {
    if (BeepFlag == 1) {
        P1_1 ^= 1;
        P0_6 = 1;
```

```
60.             WaitMs(100);
61.             P0_6 = 0;
62.             BeepFlag = 0;
63.         }
64.       }
65.     }
```

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，没有的时候 LED1 亮。

二：将程序添加到协议栈代码中

烟雾检测传感器电路是对 IO 口电平的检测。所以在协议栈里检测按键中断。我们只需要配置好 IO 口，然后中断触发检测就可以了。为了区别与按键中断不一样，在。这里定义了宏 GEC_SMOKE， 红色代码为烟雾的 IO 口定义。

1） //定义人体红外热释电传感器 IO 中断触发方式

```
#ifdef GEC_SMOKE
/* Smoke interrupts */
#define HAL_SMOKE_PORT   P1
#define HAL_SMOKE_BIT    BV(3)
#define HAL_SMOKE_SEL    P1SEL
#define HAL_SMOKE_DIR    P1DIR

/* edge interrupt */
#define HAL_SMOKE_EDGEBIT  BV(0)
#define HAL_SMOKE_EDGE     HAL_KEY_FALLING_EDGE  //下降沿触发

/* SW_6 interrupts */
```

```
#define HAL_SMOKE_IEN       IEN2  /* CPU interrupt mask register */
#define HAL_SMOKE_IENBIT    BV(4) /* Mask bit for all of Port_1 */
#define HAL_SMOKE_ICTL      P1IEN /* Port Interrupt Control register */
#define HAL_SMOKE_ICTLBIT   BV(3) /* P0IEN - P0.1 enable/disable bit */
#define HAL_SMOKE_PXIFG     P1IFG /* Interrupt flag at source */
#endif
```

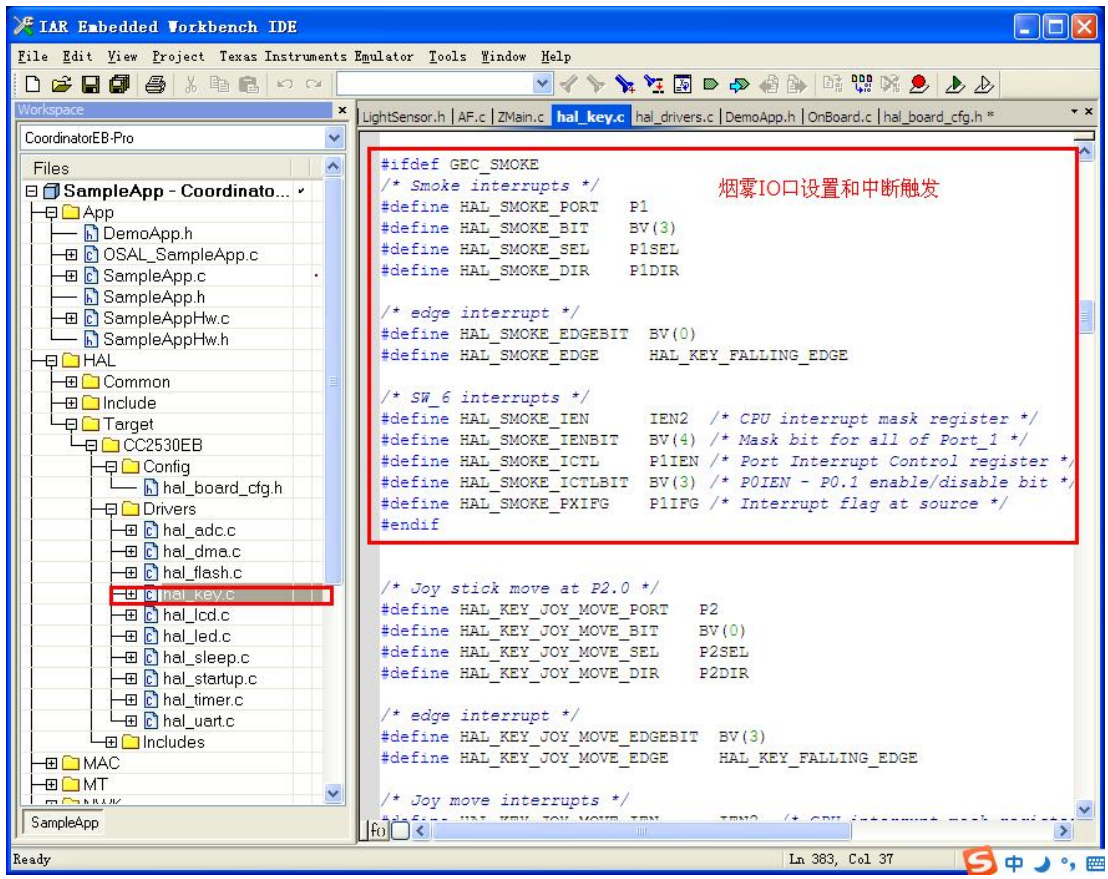


图 3- 4

2） 打开例程中的 HAL/Target/Config 目录，打开 hal_board_cfg.h 文件。我们先初始化 P1.3 引脚触发模式。设为低电平触发模式。

```
/******烟雾传感器 IO 口初始化******/
/* S2 */
#define PUSH3_BV          BV(3)
#define PUSH3_SBIT        P1_3        // Edit by GEC
```

#define PUSH3_POLARITY　　ACTIVE_LOW　// 低电平触发

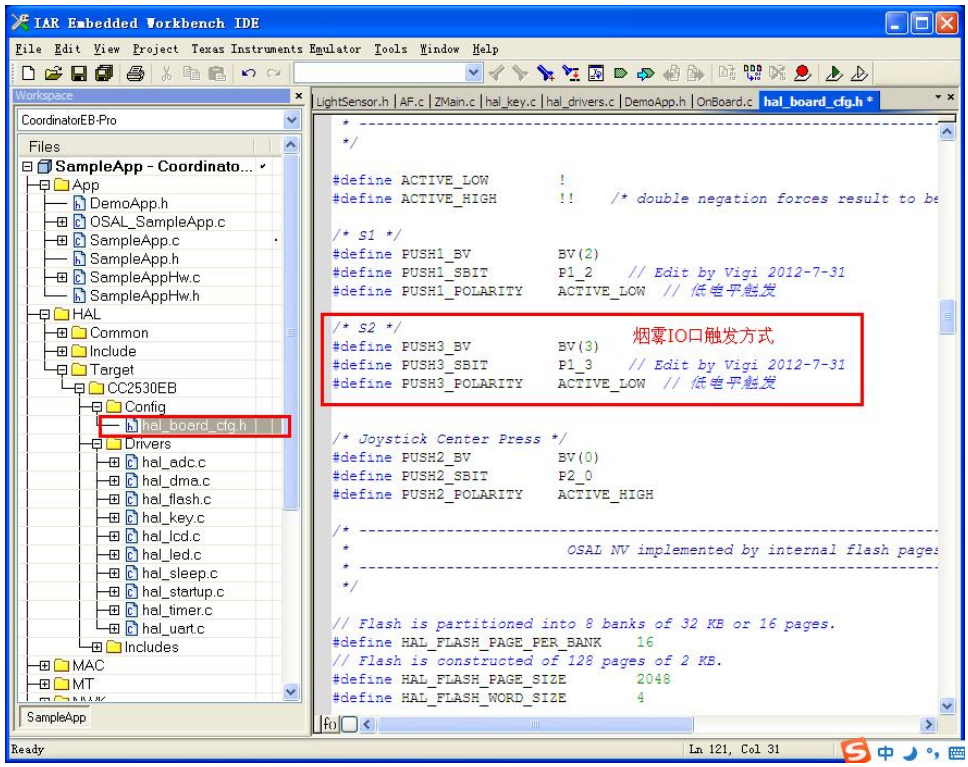


图 3- 5

3)　利用中断作为烟雾传感器触发信息采集处理，将采集到的信息发送给协调器。并通过串口打印。协调器只做串口打印。烟雾传感器被触发时采集一次中断。

```
#ifdef GEC_SMOKE

void set_smoke_signal()
{
   SampleApp_SendPointToPointMessage();
}
    #endif
```

4)　中断触发时终端执行点播函数一次，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;
    int i;
    L=1;       //有毒气
    P1_0 = 1;
    P1_1 = 1;
    for(i=0;i<3500;i++)
    MicroWait(10);
    P1_0 = 0;
    P1_1 = 0;
   HalUARTWrite(0,"there someone\n",14);       //串口

 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.
  }
}
```

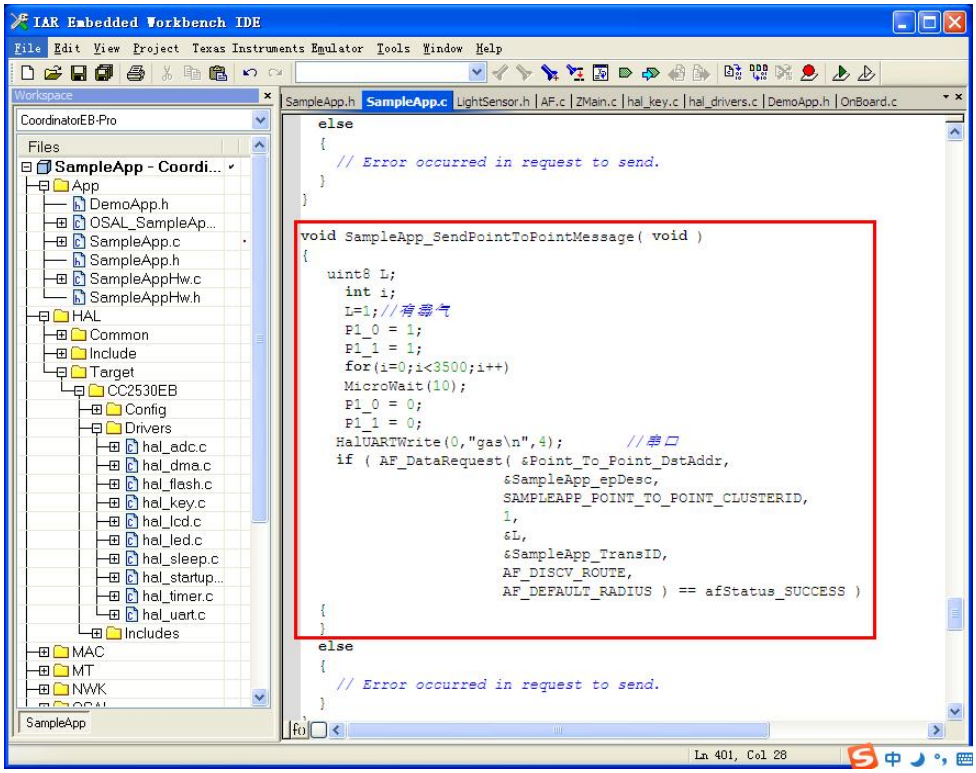


图 3- 6

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:
HalUARTWrite(0," gas\n",4);        //提醒烟雾传感器在触发
HalUARTWrite(0,"\n",1);
break;
```

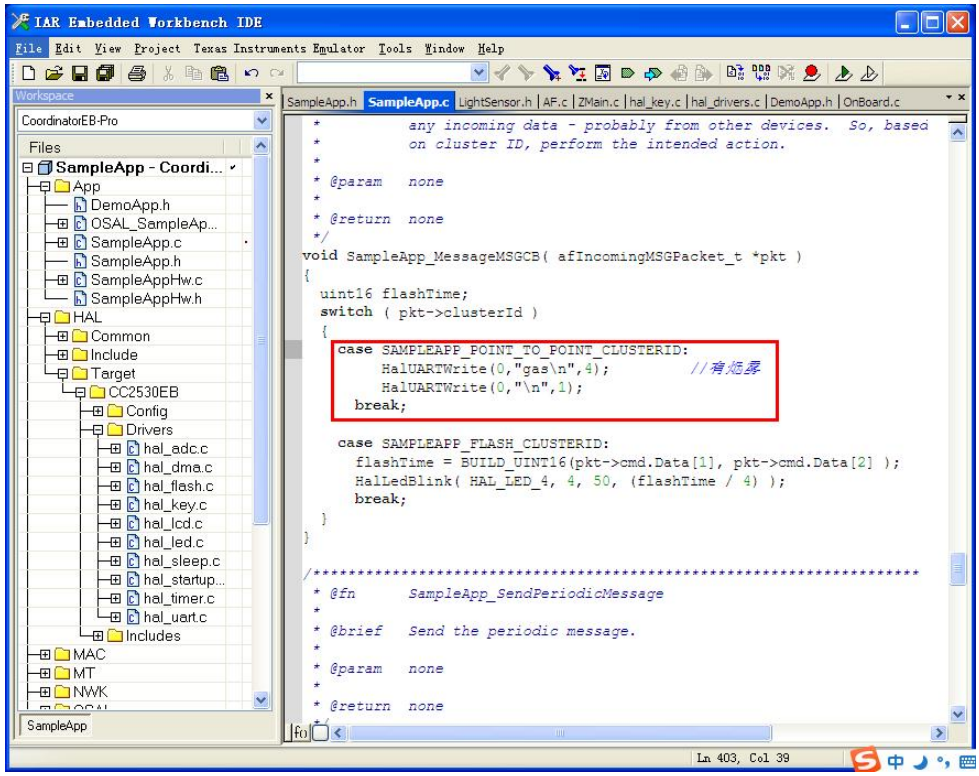


图 3- 7

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

协调器收到的信息：

图 3- 8

1.4.实验四：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人体红外热释电传感器，人体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 4- 1 红外热释电传感器模块

实现平台： ZigBee 传感器节点；

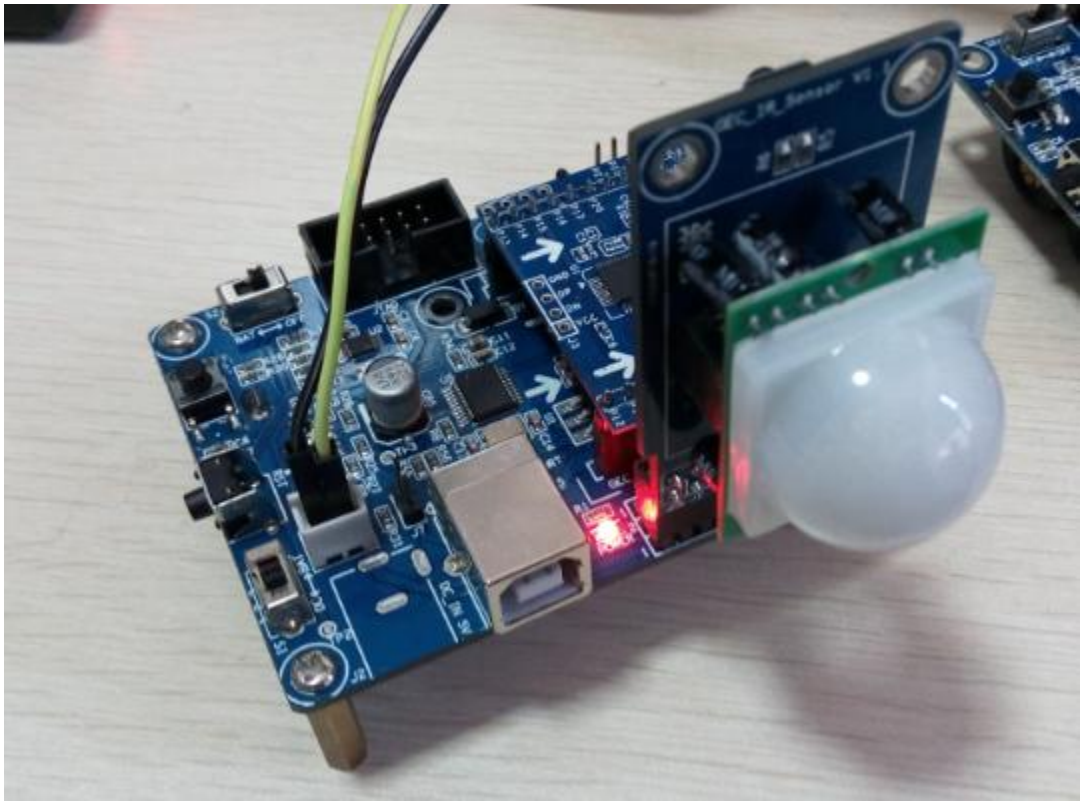


图 4- 2 人体热释电传感器

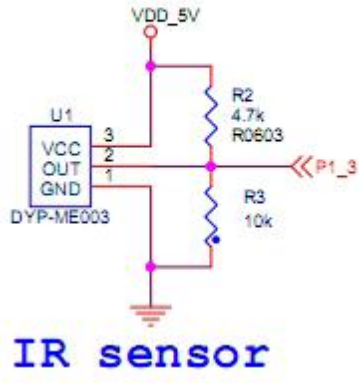


图 4- 3 人体红外热释电电路图

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，编写程序，令红外热释电传感器感应到有人后进行报警并将信息通过串口打印出来。

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.  /*            粤嵌开发团队                */
3.  /*          Zigbee 学习例程               */
4.  /*例程名称：红外热释电传感器                */
5.  /*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
6.  *****************************************/
7.  #include <ioCC2530.h>

8.  #define uint unsigned int
9.  #define uchar unsigned char
10. uint8 BeepFlag = 0;

11. /*********************************
12.  * @brief 初始化红外触发中断输入方式
13. *********************************/
14. void InitSmokeINT(void)
15. {
16.   P1DIR &= ~0x08;
17.   P1_3 = 0;
18.   P1INP |= 8;            //上拉
19.   P1IEN |= 8;           //P1.3 设置为中断方式
20.   EA = 1;
21.   IEN2 |= 0X10;         // P1 设置为中断方式
22.   P1IFG |= 0x00;        //初始化中断标志位
23. }
```

```
/**********************************
 * @brief 初始化程序,将P1.0、P1.1、
 *        P1.4定义为输出口，并将
 *        LED灯初始化为灭
**********************************/
void InitIO(void)
{
    P1DIR |= 0x02;     //P1_1为输出，初始化D2LED
    P0DIR |= 0x40;     //P0_6蜂鸣器接口
    P0_6 = 0;
    P1_1 = 1;
    P1_3 = 0;          //初始化红外的接口P1_3
}
/*****************************
 * @brief   中断服务
*****************************/
#pragma vector = P1INT_VECTOR
__interrupt void P1_ISR(void)
{
  if(P1IFG > 0)             //红外触发中断标志
  {
    P1IFG = 0;
    WaitMs(25);
    if (P1IFG == 0) {
    BeepFlag = 1;
    }
  }
  P1IF = 0;                 //清中断标志
}
```

```
53. /****************************
        主函数
54. ****************************/
55. void main(void)
56. {
57.   InitIO();                //调用 IO 初始化函数，初始化 led 或蜂鸣器
58.   InitSmokeINT();          //初始化红外触发中断输入方式
59.   while(1)
60.   {
61.     if (BeepFlag == 1)     //中断标志被触发
62.     {
63.       P1_1 ^= 1;           //改变 LED 状态
64.       P0_6 = 1;            //蜂鸣器响
65.       WaitMs(5);           //延时 5s
66.       P0_6 = 0;            //关蜂鸣器
67.       BeepFlag = 0;        //清中断标志
68.     }
69.   }
70. }
```

我们来看主函数：

第 58~59 行：进行一些初始化工作。

第 62~69 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。

二：将程序添加到协议栈代码中

人体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按键中断。我们只需要配置好 IO 口，然后中断触发检测就可以了。为了区别与按键中断不一样，在。这里定义了宏 GEC_SMOKE， 红色代码为红外的 IO 口。

6） //定义人体红外热释电传感器 IO 中断触发方式

```
#ifdef GEC_SMOKE
/* Smoke interrupts */
#define HAL_SMOKE_PORT    P1
#define HAL_SMOKE_BIT     BV(3)
#define HAL_SMOKE_SEL     P1SEL
#define HAL_SMOKE_DIR     P1DIR
/* edge interrupt */
#define HAL_SMOKE_EDGEBIT  BV(0)
#define HAL_SMOKE_EDGE     HAL_KEY_FALLING_EDGE  //下降沿触发
/* SW_6 interrupts */
#define HAL_SMOKE_IEN      IEN2  /* CPU interrupt mask register */
#define HAL_SMOKE_IENBIT   BV(4) /* Mask bit for all of Port_1 */
#define HAL_SMOKE_ICTL     P1IEN /* Port Interrupt Control register */
#define HAL_SMOKE_ICTLBIT  BV(3) /* P0IEN - P0.1 enable/disable bit */
#define HAL_SMOKE_PXIFG    P1IFG /* Interrupt flag at source */
#endif
```

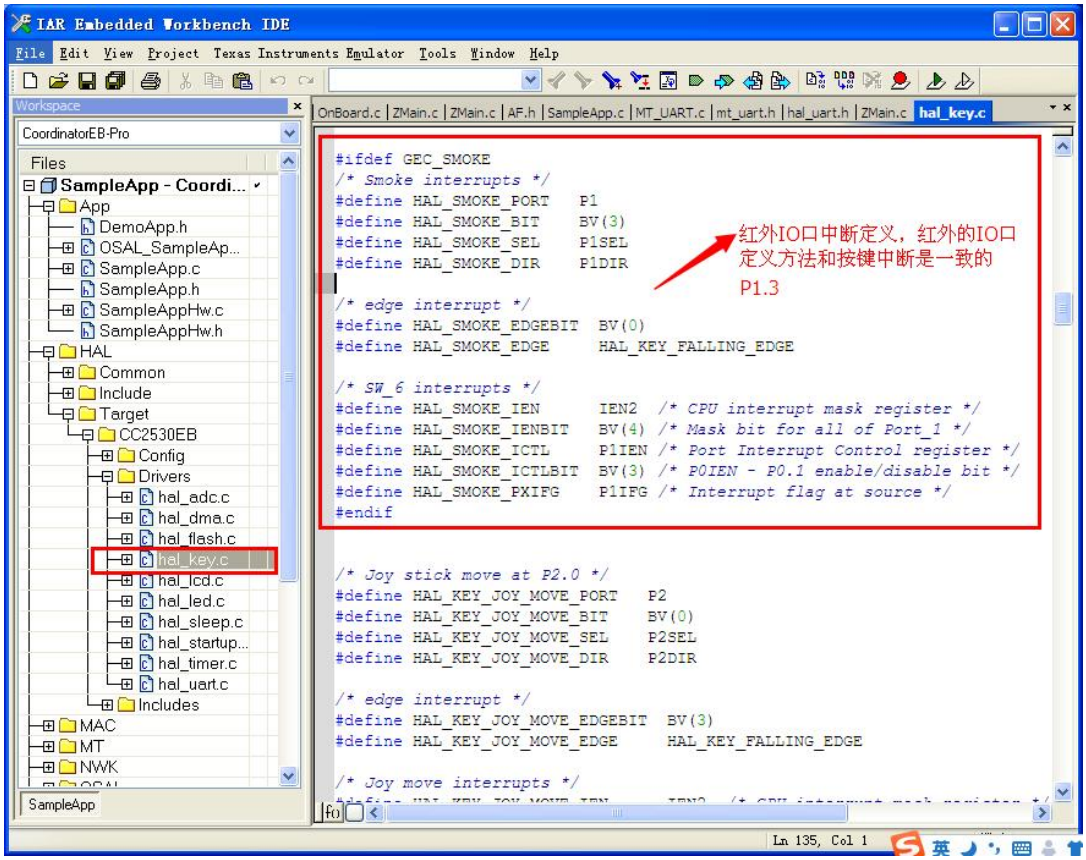


图 4- 4

7） 打开例程中的 HAL/Target/Config 目录，打开 hal_board_cfg.h 文件。

我们先初始化 P1.3 引脚触发模式。设为低电平触发模式。

```
/******人体红外热释电传感器 IO 口初始化******/
#define PUSH3_BV          BV(3)
#define PUSH3_SBIT        P1_3          // Edit by GEC
#define PUSH3_POLARITY    ACTIVE_LOW  // 低电平触发
```

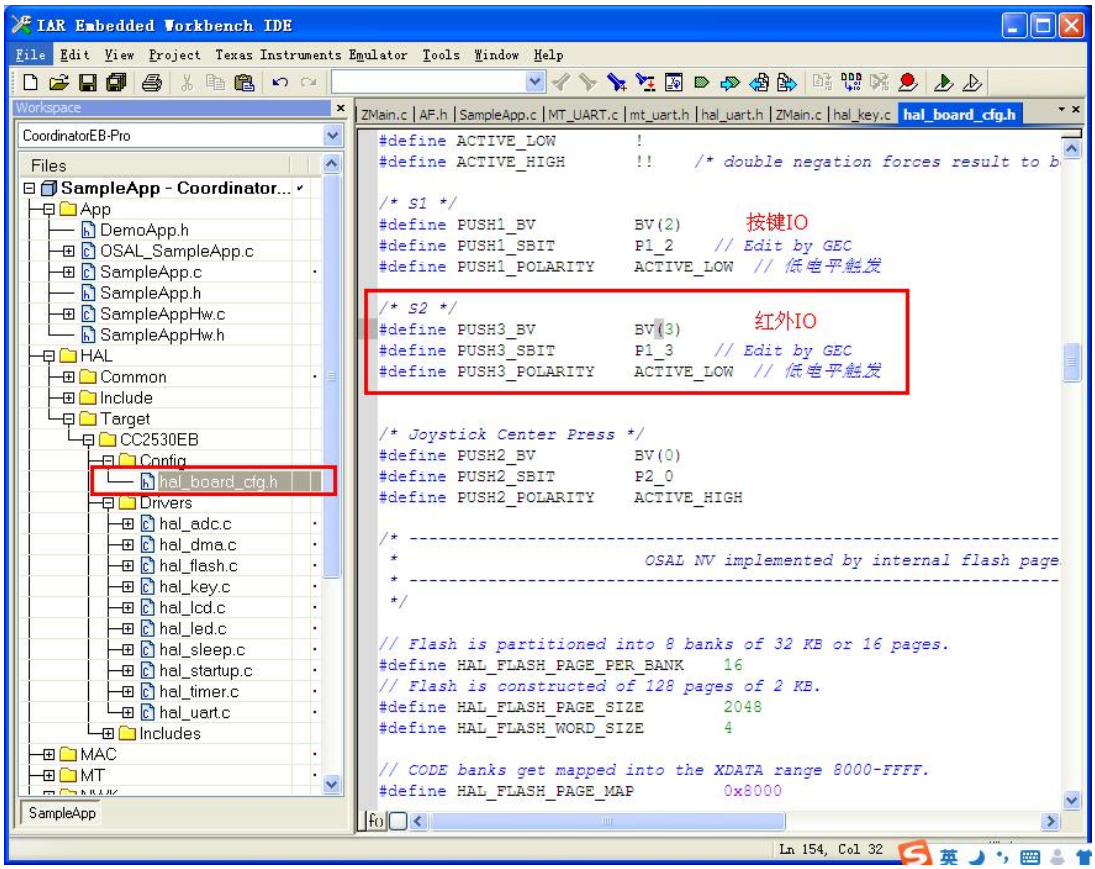


图 4- 5

8） 利用中断作为红外触发信息采集处理，将采集到的信息发送给协调器。并通过串口打印。协调器只做串口打印。红外被触发时采集一次中断。

```
#ifdef GEC_SMOKE
void set_smoke_signal()
{
   SampleApp_SendPointToPointMessage();
}
    #endif
```

9） 中断触发时终端执行点播函数一次，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;
      int i;
      L=1;        //有人
      P1_0 = 1;
      P1_1 = 1;
      for(i=0;i<3500;i++)
      MicroWait(10);
      P1_0 = 0;
      P1_1 = 0;
     HalUARTWrite(0,"there someone\n",14);        //串口
  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.
    }
  }
```

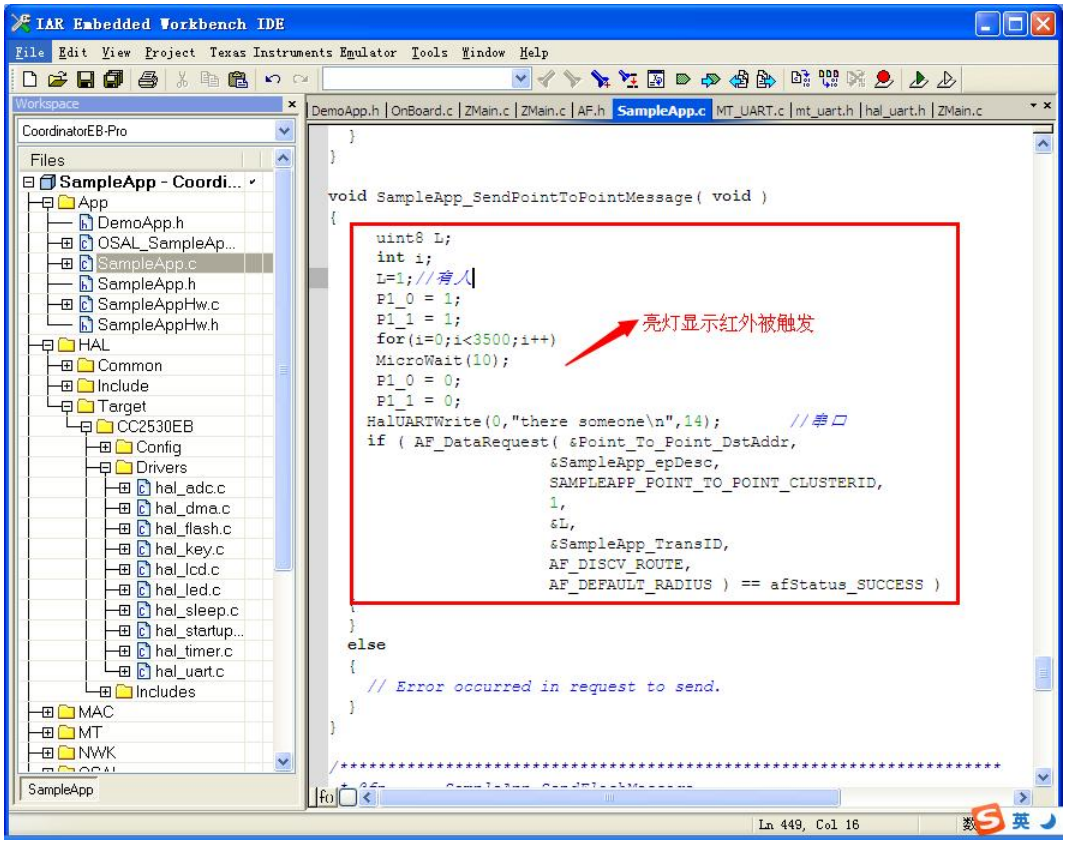


图 4- 6

10） 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:
HalUARTWrite(0,"there's someone\n",16);        //提醒有人在触发红外
HalUARTWrite(0,"\n",1);
break;
```

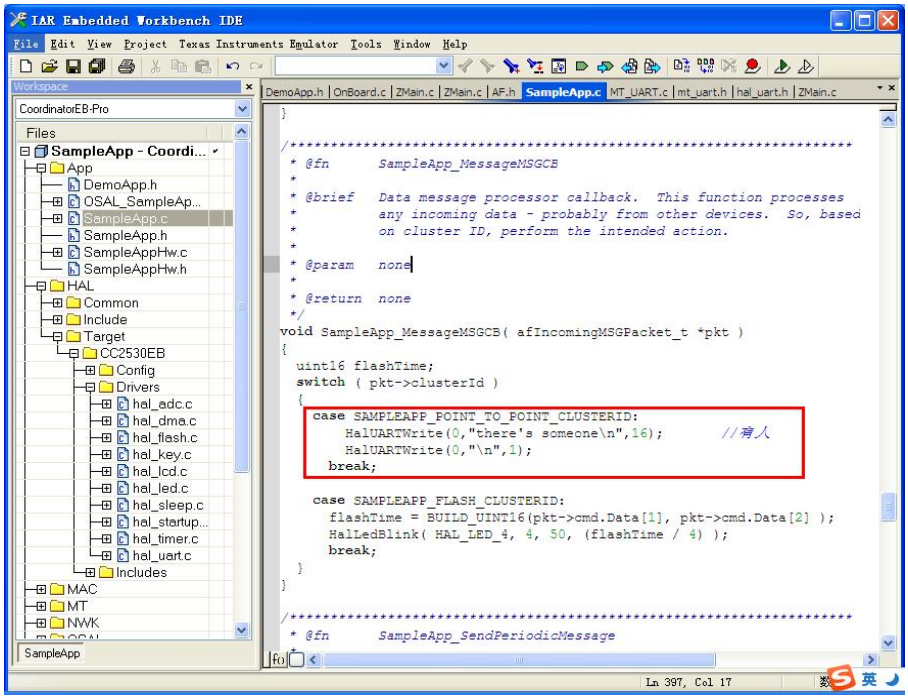


图 4- 7

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

协调器收到的信息：

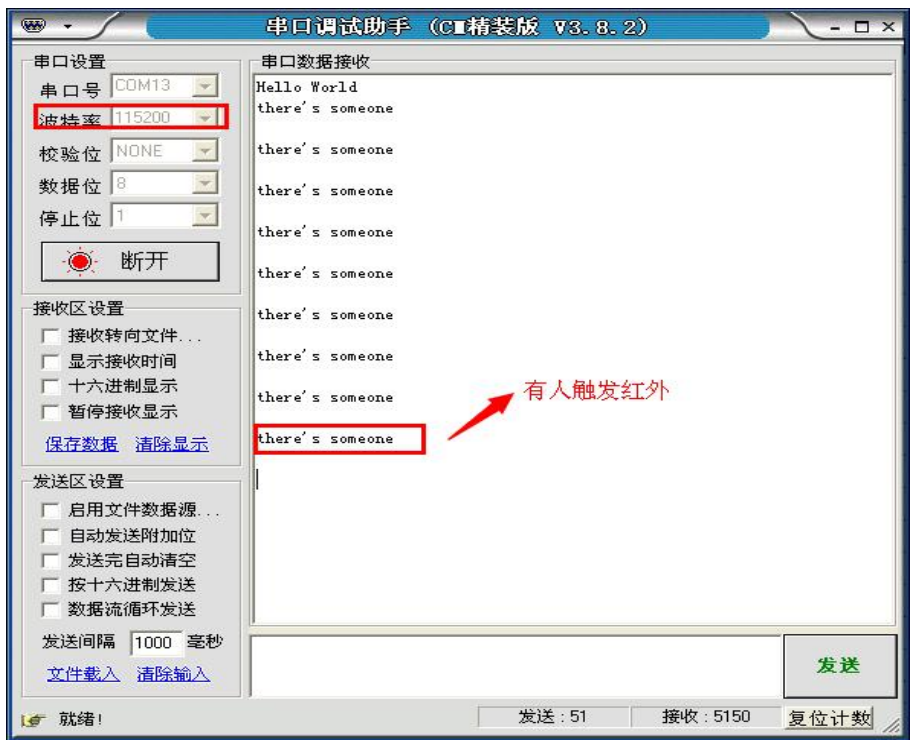


图 4- 8

1.5. 实验五：三轴加速传感器

前言：这一节我们我学习传感器部分内容中的三轴加速传感器。

传感器介绍：

目前的三轴加速度传感器大多采用压阻式、压电式和电容式工作原理，产生的加速度正比于电阻、电压和电容的变化，通过相应的放大和滤波电路进行采集，三轴加速度传感器具有体积小和重量轻特点，可以测量空间加速度，能够全面准确反映物体的运动性质，在航空航天、机器人、汽车和医学等领域得到广泛的应用。。

图 5- 1 三轴加速传感器

实现平台： ZigBee 协调器和传感器节点；

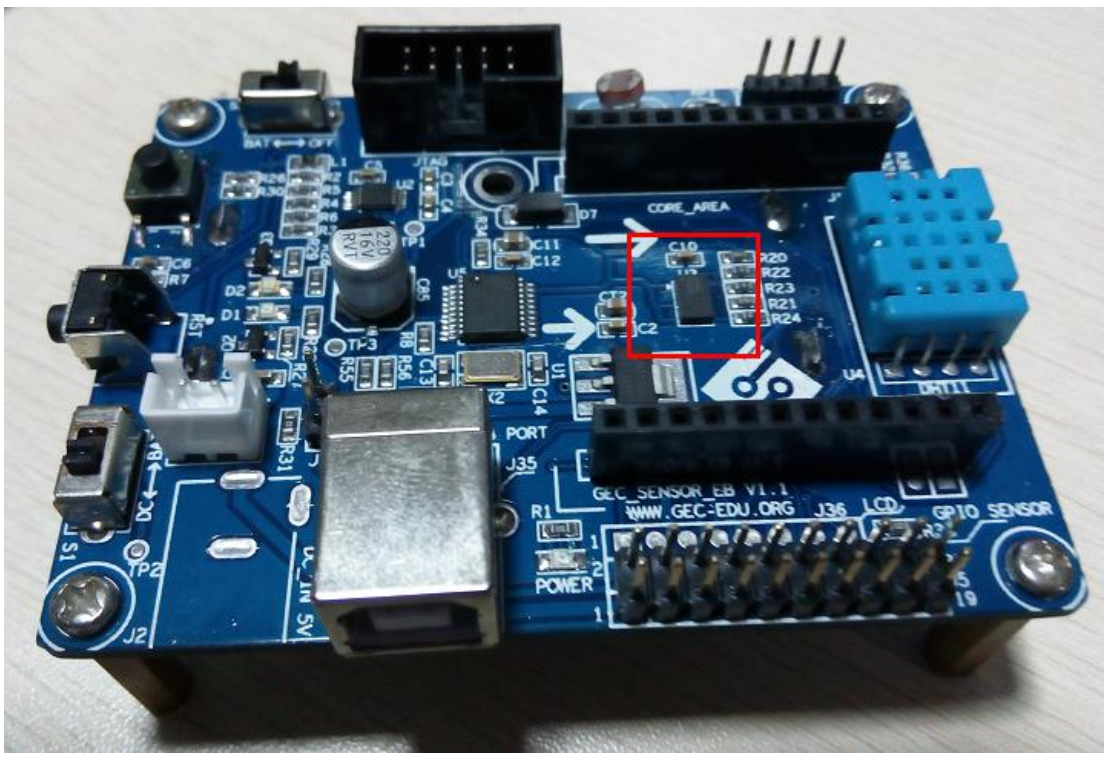


图 5- 2 三轴加速传感器

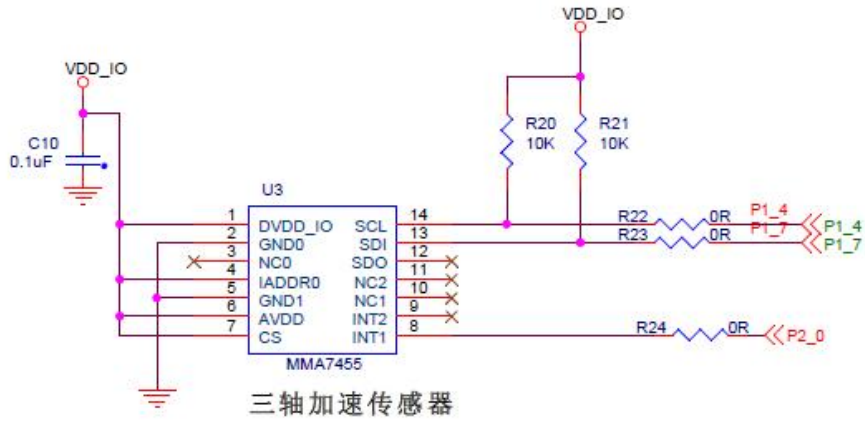


图 5- 3 三轴传感器硬件电路

实验现象：传感器与 CC2530 通过 I2C 总线通讯，然后处理器将信息通过串口打印出来。

实验讲解：像前面传感器例程一样，我们先实现裸机程序里三轴传感器方向的检测。然后在协议栈里添加相应的代码。

一：在裸机上完成对三轴加速传感电路的驱动。

打开配套程序下裸机文件夹—三轴加速传感器下的工程文件，看到函数如下：

```
1.  #include "ioCC2530.h"
2.  #include "stdio.h"
3.  #include <string.h>
4.  #define uchar unsigned char
5.  #define uint  unsigned int
6.  #define IIC_READ  0x1D              //定义读指令
7.  #define IIC_WRITE 0x1D              //定义写指令
8.  #define SDA P1_7                    //I2C 数据传送位
9.  #define SCL P1_4                    //I2C 时钟传送位
10. #define MMA7455ADDR 0x1D
11.
12. #define iic_delay() Delay_1u(8)     //延时函数
```

```
13.
14. /******* main 函数 ********/
15. void main ()
16. {
17.     initUARTSEND();
18.     UartTX_Send_String("Test\n", 6);
19.     uchar xl, yl, zl;
20.     uchar i, debug[0x1E];
21.     char buf[32];
22.     //初始化
23.     P1SEL &= ~(1<<4);                   // P1_4 is GPIO
24.     P1SEL &= ~(1<<7);                   // P1_7 is GPIO
25.     P2SEL&= ~0x01;                      // P1_2 is GPIO
26.     P1DIR |= 1<<4;                      //P1_4-->output
27.     P1DIR |= 1<<7;                      //P1_7-->output
28.     P2DIR |= 0x01;                      //P2_0-->output
29.     Delay_1u(100);
30.     MMA7455Init();
31.     while(1)
32.     {
33.       iic_write(0x17,0x03);             //清除 INT1、INT2 的标志位
34.       iic_write(0x16,0x05);             //配置工作方式,2g 量程，测量模
式
35.       Delay_1u(300);
36.       P2DIR &= ~0x01;                   //输入
37.       P2_0 = 1;
38.       while(!P2_0);                     //等待数据就绪
39.       for(i = 0; i < 0x9; i++)
40.       debug[i] = iic_read(i);
```

```
41.         debug[0x15] = iic_read(0x15);
42.         debug[0x16] = iic_read(0x16);
43.         if(debug);                          //防止 debug 被优化掉
44.         xl = iic_read(0x00);  //8bit
45.         yl = iic_read(0x02);  //8bit
46.         zl = iic_read(0x04);  //8bit
47.         int x, y, z;
48.         x = xl & 0x80 ? xl - 256 : xl;
49.         y = yl & 0x80 ? yl - 256 : yl;
50.         z = zl & 0x80 ? zl - 256 : zl;
51.         sprintf(buf, "X = %d, Y = %d, Z = %d\n", x, y, z);
52.         UartTX_Send_String(buf, strlen(buf));
53.         Delay_1u(90000);
54.     }
55. }
```

我们来看主函数：

```
initUARTSEND ();初始化串口 0
MMA7455Init();    对 3D 加速度传感器进行初始化
iic_write();      I2C 写函数
iic_read();       I2C 读函数
while(1);         在大循环中，不断进行 3D 加速度传感器的 I2C 读写
Uart_Send_String()：信息通过串口打印
```

56. 上面的代码实现了 3D 加速度传感器的 3 轴的变化，通过串口打印出相应的 3 条轴转换的数值。实验现象如图 5-4 所示：

图 5-4

二：将程序添加到协议栈代码中

有了 3D 加速度传感器驱动的代码，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。

首先理清一下思路，我们要实验的功能是终端设备读取 3D 加速度传感器信息，通过点播方式发送到协调器，协调器通过串口打印出来。在串口调试助手上面显示。这就实现了无线 3D 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点播章节，这里不再累赘。）

1）我们将裸机程序里面的 mma7455.c 文件复制到 SAMPLEAPP -- Source 文件夹下。

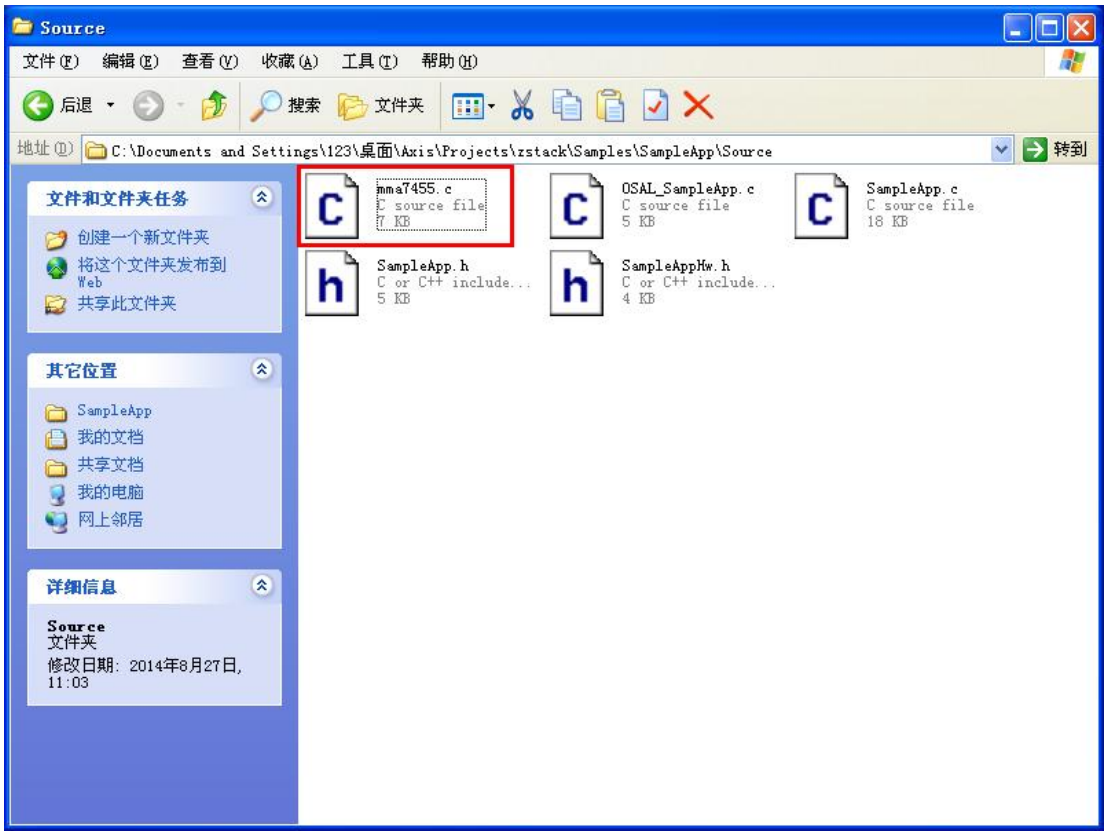


图 5-5

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

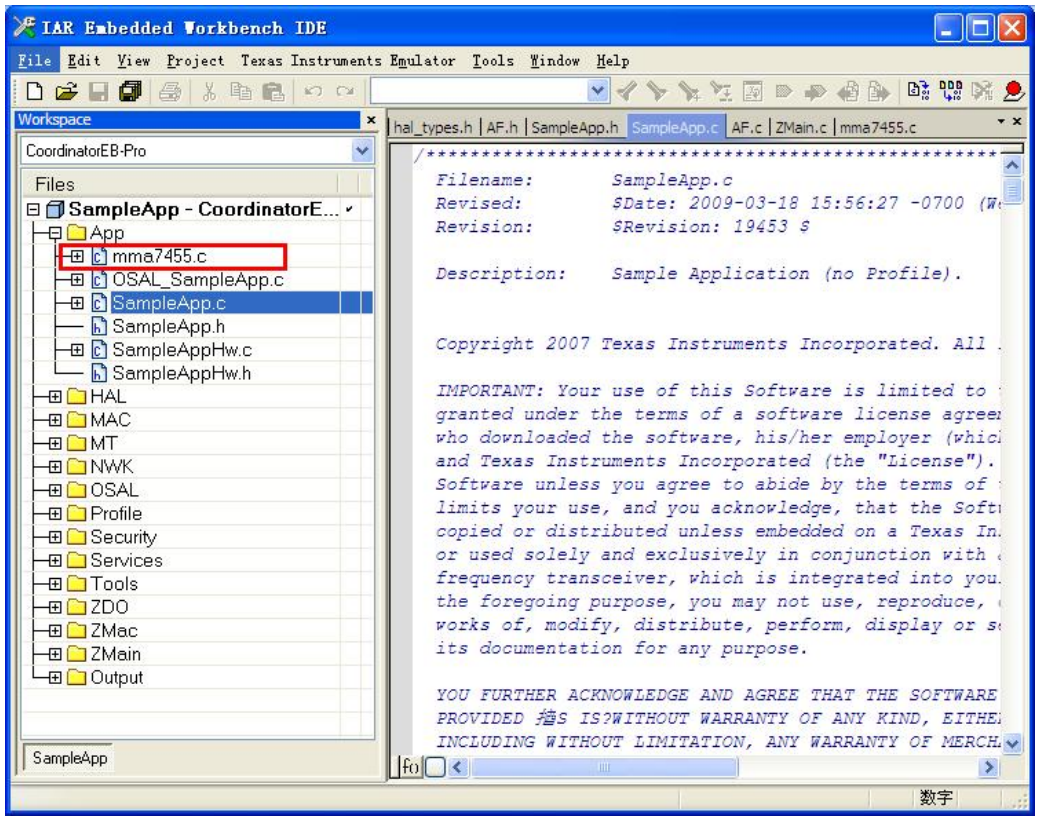


图 5-6

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 stdio.h 文件。

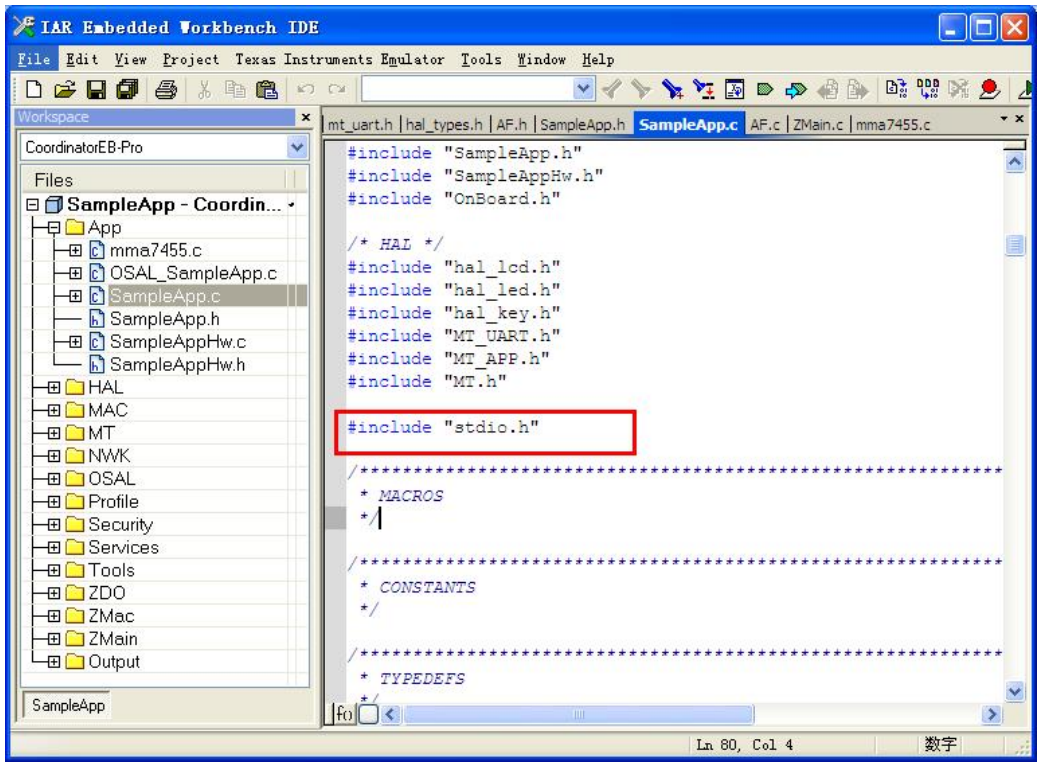


图 5-7

4）借用点对点的点播函数，3s 读取温度传感器 3 次，通过串口打印并点对点发送给协调器。代码如下，如图 6- 49 所示：

```
1.  void SampleApp_SendPointToPointMessage( void )
2.  {
3.    uint16 temp[3];
4.    char buf[32];
5.    Get_MMA7455_Data(temp);
6.    sprintf(buf, "X = %d, Y = %d, Z = %d\n", temp[0], temp[1], temp[2]);
7.    if ( AF_DataRequest( &Point_To_Point_DstAddr,
8.                         &SampleApp_epDesc,
9.                         SAMPLEAPP_POINT_TO_POINT_CLUSTERID,
10.                        32,
```

```
11.                         buf,
12.                         &SampleApp_TransID,
13.                         AF_DISCV_ROUTE,
14.                         AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
15.   {
16.   }
17.   else
18.   {
19.     // Error occurred in request to send.
20.   }
21. }
```

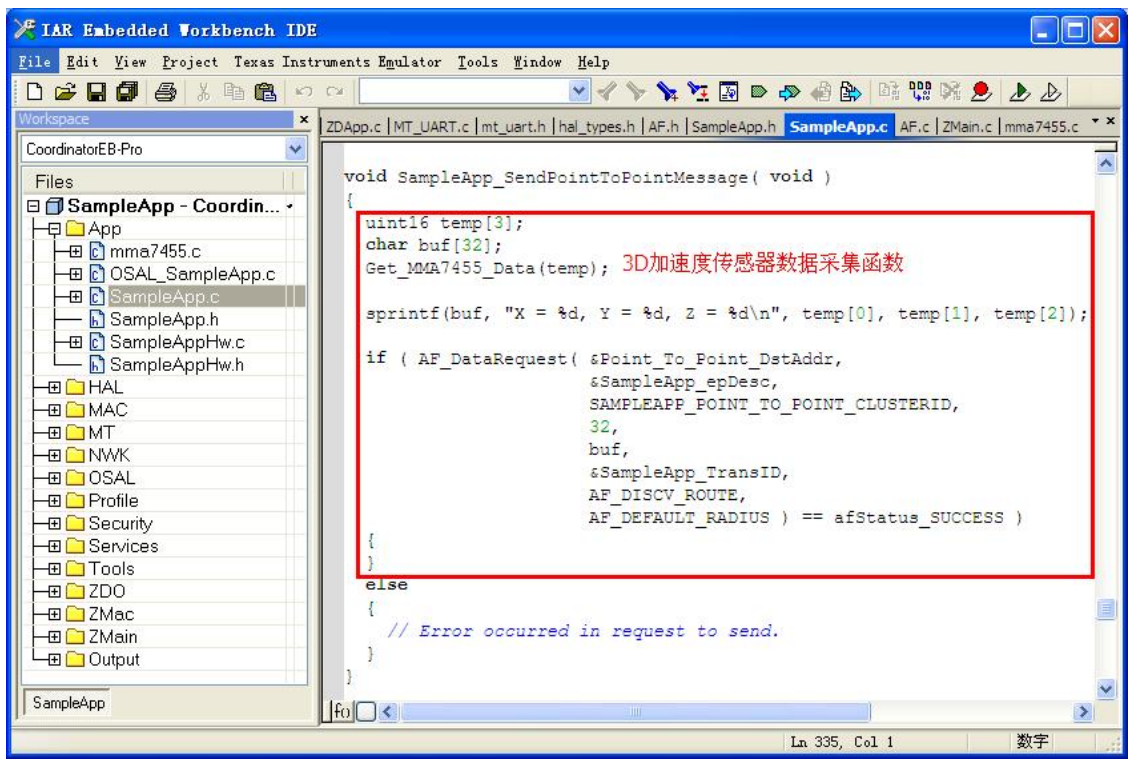


图 5-8

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。

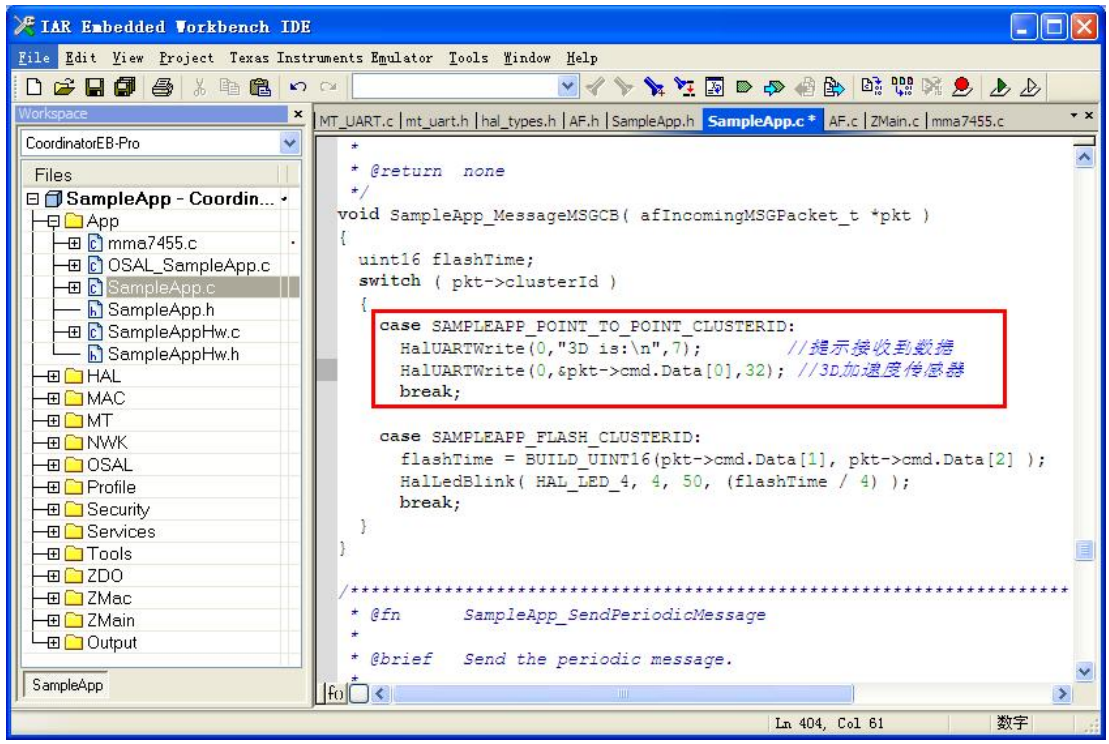


图 5- 9

实验现象：下载程序到终端和协调器，协调器收到的信息：

图 5- 10

1.6. 实验六：磁控传感器

前言：这一节我们我学习传感器部分内容中的磁控（干簧管）传感器。

传感器介绍：

干簧管是干式舌簧管的简称，是一种有触点的无源电子开关元件，具有结构简单，体积小便于控制等优点，其外壳一般是一根密封的玻璃管，管中装有两个铁质的弹性簧片电板，还灌有一种叫金属铑的惰性气体。平时，玻璃管中的两个由特殊材料制成的簧片是分开的。当有磁性物质靠近玻璃管时，在磁场磁力线的作用下，管内的两个簧片被磁化而互相吸引接触，簧片就会吸合在一起，使结点所接的电路连通。外磁力消失后，两个簧片由于本身的弹性而分开，线路也就断开了。因此，作为一种利用磁场信号来控制的线路开关

器件，干簧管可以作为传感器用，用于计数，限位等等（在安防系统中主要用于门磁、窗磁的制作），同时还被广泛使用于各种通信设备中。在实际运用中，通常用永久磁铁控制这两根金属片的接通与否，所以又被称为“磁控管”。

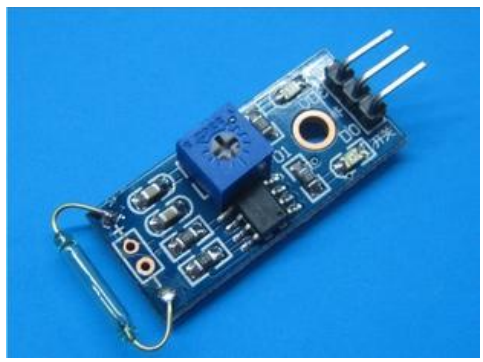


图 6- 1 磁控传感器

实现平台： ZigBee 协调器和传感器节点；

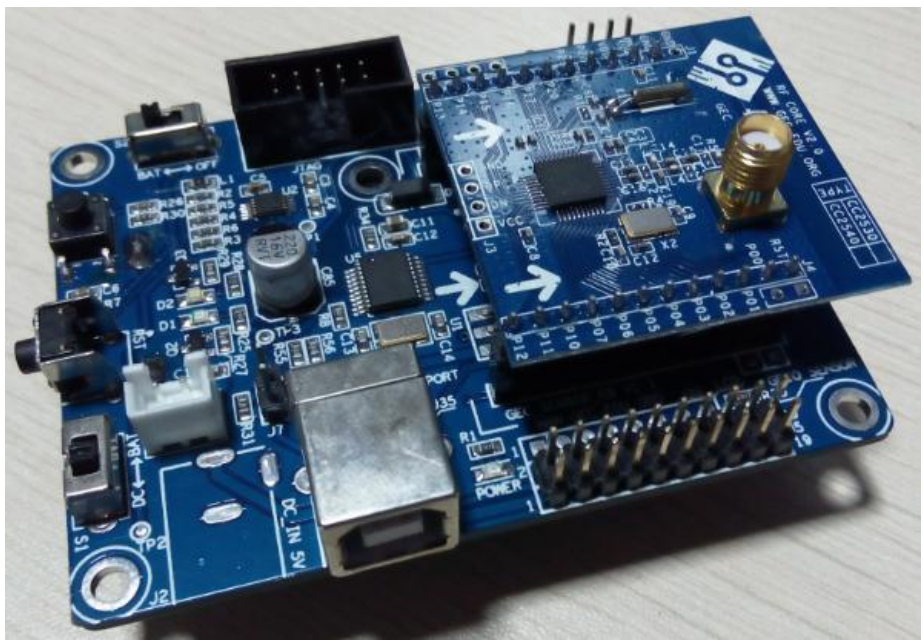


图 6- 2 传感器节点

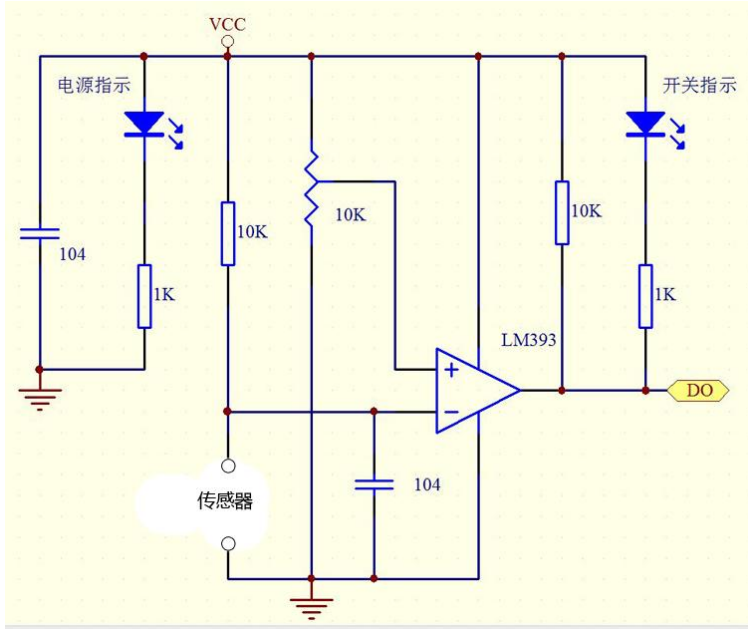


图 6- 3　磁控管硬件电路

实验现象：干簧管需要和磁铁配合使用，在感应到有一定的磁力的时候，会呈导通状态，模块输出低电平，无磁力时，呈断开状态，输出高电平，（干簧管与磁铁的感应距离在 1.5cm 之内超出不灵敏或会无触发现象）然后处理器将信息通过串口打印出来。

实验讲解：像前面传感器例程一样，我们先实现裸机程序里磁控传感器信号检测。然后在协议栈里添加相应的代码。

一：在裸机上完成对磁控传感器的驱动。

打开配套程序下裸机文件夹—磁控传感器下的工程文件，看到函数如下：

```
/*****************************
        主函数
*****************************/
void main(void)
{
  InitIO();
  InitMagneticINT();          //初始化磁控中断
```

```
7.   while (1) {
8.     if (MagneticFlag == 1) {
9.         P1_1 ^= 1;
10.            WaitMs(100);
11.           MagneticFlag = 0;
12.        }
13.     }
14.   }
```

我们来看主函数：

第 5~6 行：进行一些初始化工作。

第 7~13 行：判断磁控传感器是否被触发情况,通过 LED1 指示。

二：将程序添加到协议栈代码中

磁控检测传感器电路是对 IO 口电平的检测。所以在协议栈里检测按键中断。我们只需要配置好 IO 口，然后中断触发检测就可以了。为了区别与按键中断不一样，在。这里定义了宏 GEC_MAGNETIC， 红色代码为磁控的 IO 口定义。

1） //定义磁控传感器 IO 中断触发方式

```
#ifdef GEC_MAGNETIC
/* Smoke interrupts */
#define HAL_SMOKE_PORT   P0
#define HAL_SMOKE_BIT    BV(7)
#define HAL_SMOKE_SEL    P0SEL
#define HAL_SMOKE_DIR    P0DIR

/* edge interrupt */
#define HAL_SMOKE_EDGEBIT  BV(0)
#define HAL_SMOKE_EDGE     HAL_KEY_FALLING_EDGE  //下降沿触发
```

```
/* SW_6 interrupts */
#define HAL_SMOKE_IEN       IEN1  /* CPU interrupt mask register */
#define HAL_SMOKE_IENBIT    BV(5) /* Mask bit for all of Port_1 */
#define HAL_SMOKE_ICTL      P0IEN /* Port Interrupt Control register */
#define HAL_SMOKE_ICTLBIT   BV(7) /* P0IEN - P0.1 enable/disable bit */
#define HAL_SMOKE_PXIFG     P0IFG /* Interrupt flag at source */
#endif
```

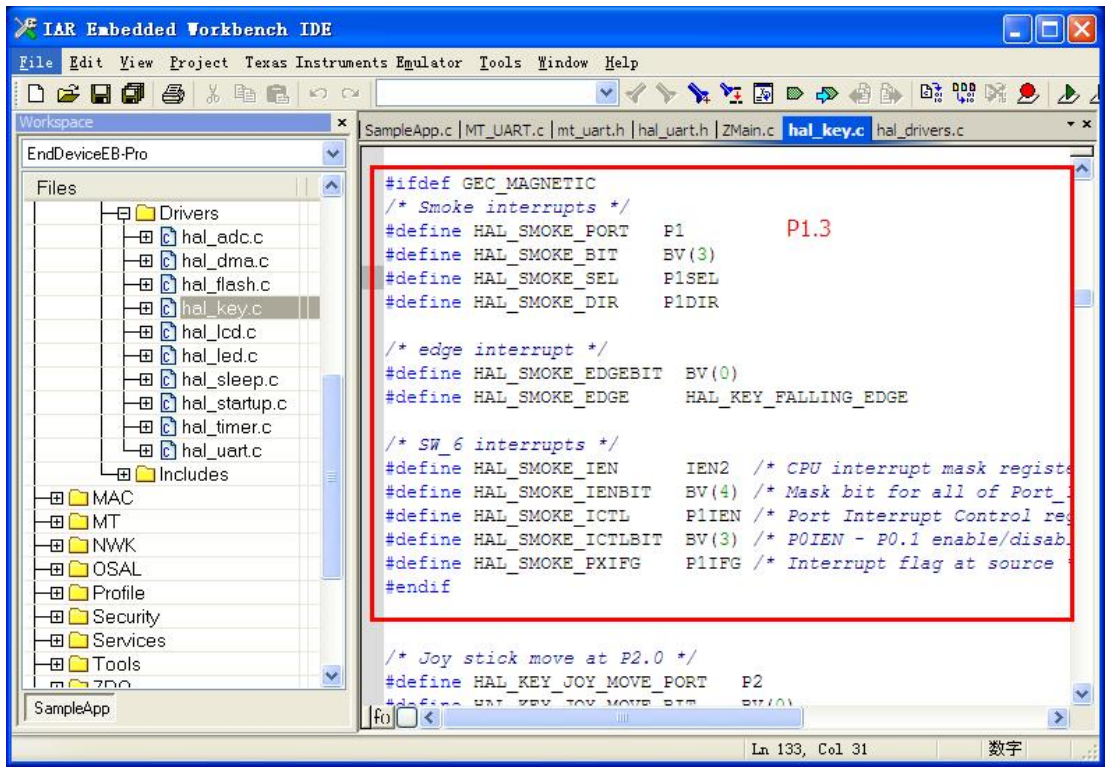


图 6- 4

2） 打开例程中的 HAL/Target/Config 目录，打开 hal_board_cfg.h 文件。我们先初始化 P0.7 引脚触发模式。设为低电平触发模式。

```
/******磁控传感器 IO 口初始化******/
/* S2 */
#define PUSH3_BV          BV(3)
#define PUSH3_SBIT        P1_3        // Edit by GEC
```

#define PUSH3_POLARITY　　ACTIVE_LOW　// 低电平触发

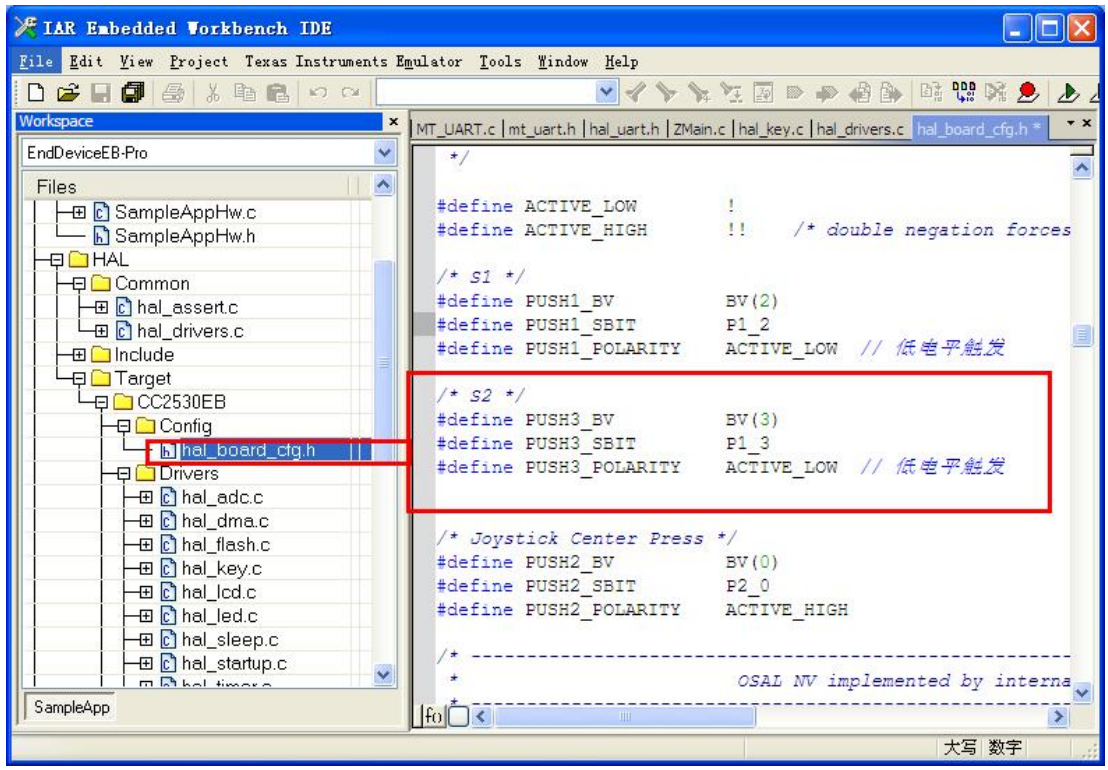


图 6- 5

3)　利用中断作为磁控传感器触发信息采集处理，将采集到的信息发送给协调器。并通过串口打印。协调器只做串口打印。磁控传感器被触发时采集一次中断。

```
#ifdef GEC_MAGNETIC

void set_magnetic_signal ()
{
   SampleApp_SendPointToPointMessage();
}
    #endif
```

4)　中断触发时终端执行点播函数一次，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;
    int i;
    L=1;        //磁控传感器被触发提示
    P1_0 = 1;
    P1_1 = 1;
    for(i=0;i<3500;i++)
    MicroWait(10);
    P1_0 = 0;
    P1_1 = 0;

  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.
   }
 }
```

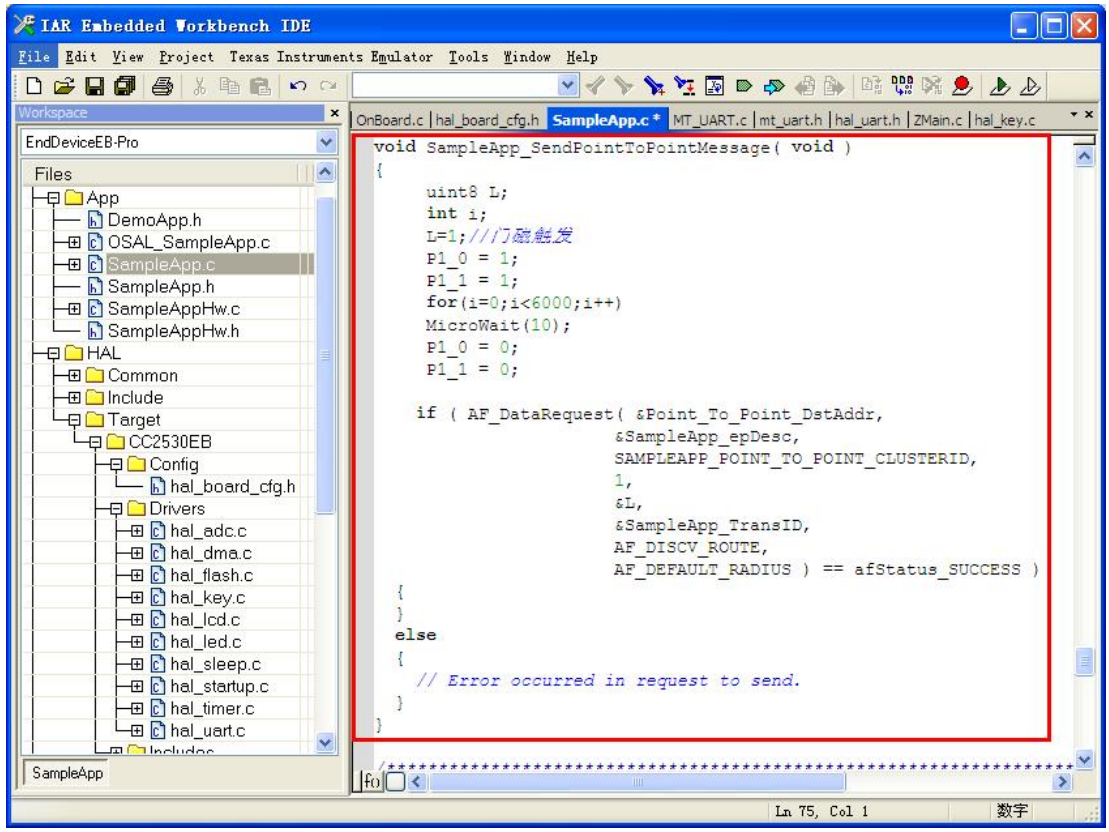


图 6- 6

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:
HalUARTWrite(0," gas\n",4);        //提醒烟雾传感器在触发
HalUARTWrite(0,"\n",1);
break;
```

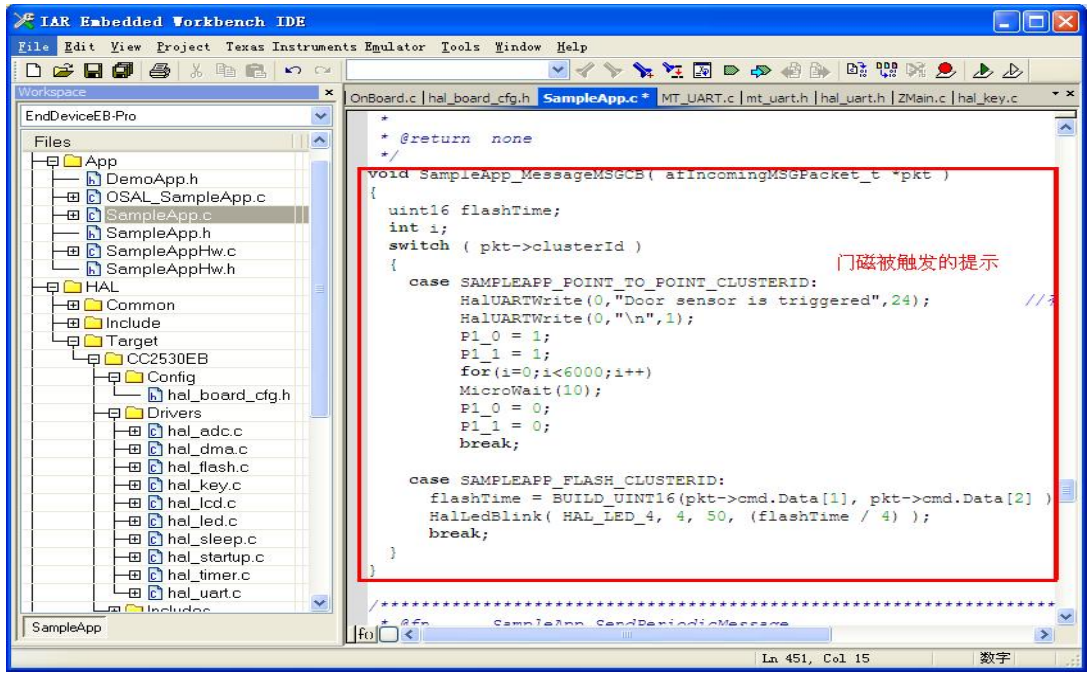


图 6- 7

由于前面定义了宏 GEC_MAGNETIC，所以在开发工具中要进行声明：

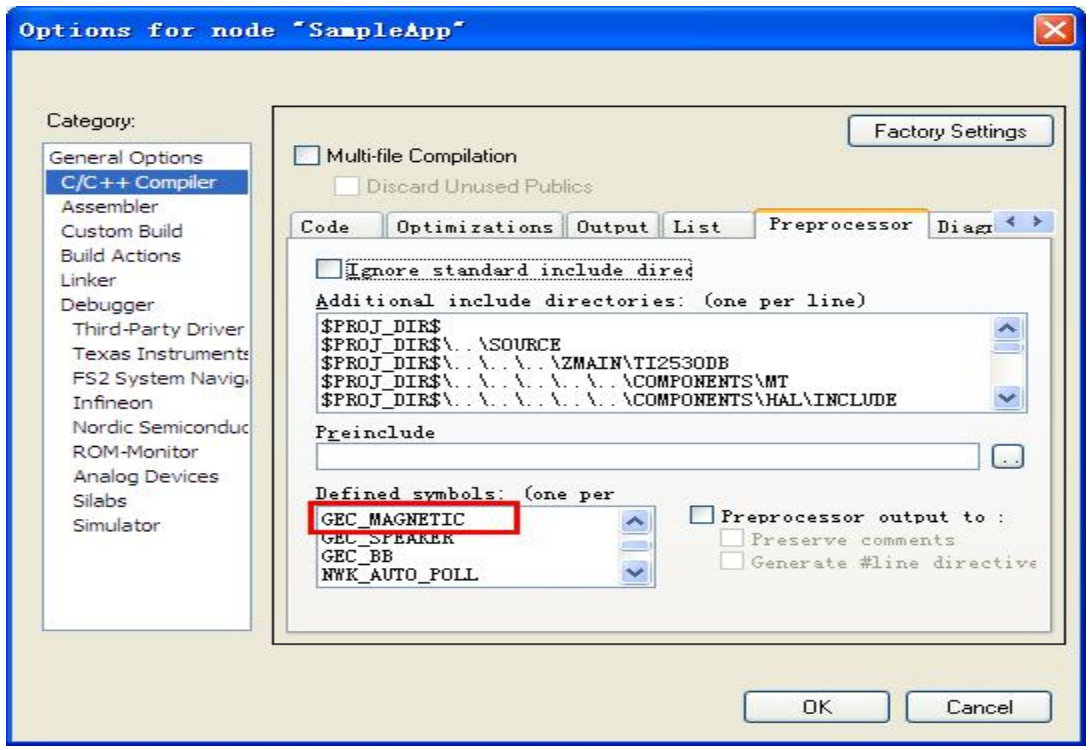


图 6- 8

实验现象：下载程序到终端（带磁控传感器）和协调器。观察串口如下图所示：

协调器收到的信息：

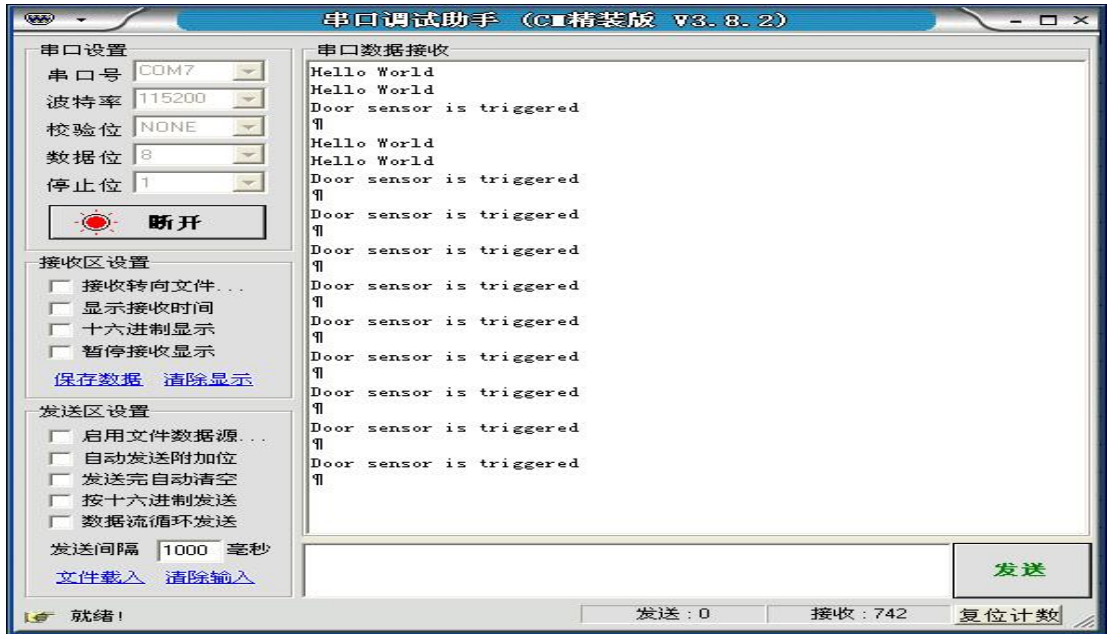


图 6- 9

1.7.实验七：直流电机模块

前言：这一节我们我学习传感器和执行器部分内容中的直流电机模块。

执行器介绍：直流电机（direct current machine）是指能将直流电能转换成机械能（直流电动机）或将机械能转换成直流电能（直流发电机）的旋转电机。

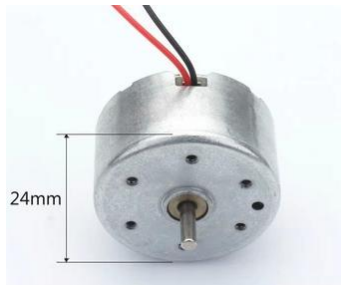


图 7- 1 直流电机

实现平台： ZigBee 协调器和传感器节点；

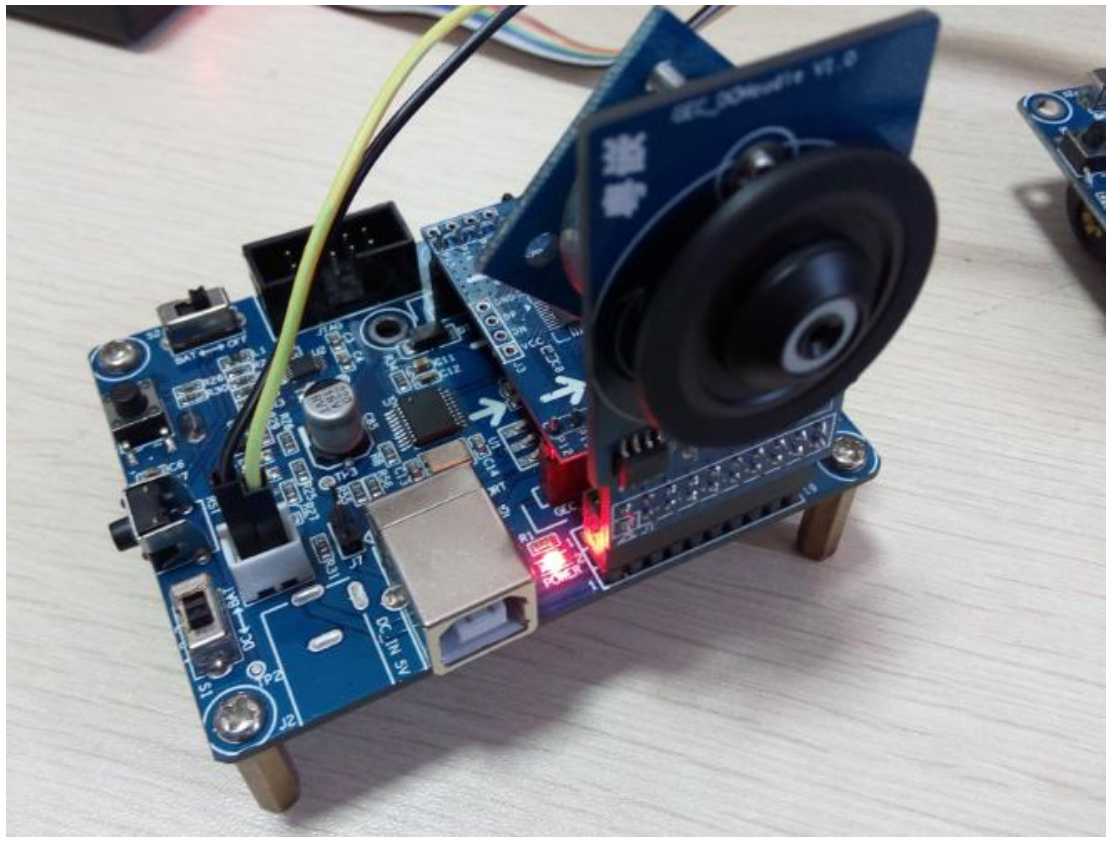


图 7- 2 传感器执行节点

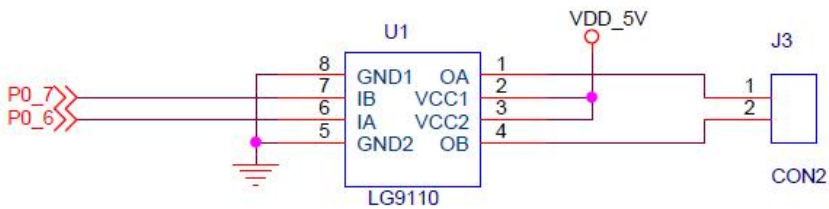


图 7- 3 直流电机电路

实验现象：裸机通过按键控制，加入协议栈后用串口输入指令来实现直流电机的正转和反转、停止转动；

实验讲解：像前面传感器例程一样，我们先实现裸机程序里对直流电机控制。然后在协议栈里添加相应的代码。

一：在裸机上完成对直流电机控制的驱动。

打开配套程序下裸机文件夹—直流电机下的工程文件，看到函数如下：

```
1. /**
2.  * project: 直流电机传感器
3.  * 作者    : GEC
4. */
5. #include <ioCC2530.h>
6. typedef unsigned char uint8;
7. typedef unsigned int uint16;
8. uint8 dir = 0;
9. uint8 flg = 0;
10.    /*********************************
11.     * @brief 初始化按键为中断输入方式
12.    *********************************/
13.    void InitKeyINT(void)
14.    {
15.      P1DIR &= ~0x04;      //初始化按键 P1_2 为输入
16.      P1INP |= 0x04;       //上拉
17.      P1IEN |= 0x04;       //P1.3 设置为中断方式
18.      PICTL |= 0X01;       //下降沿触发
19.      EA = 1;
20.      IEN2 |= 0X10;        // P1 设置为中断方式
21.      P1IFG |= 0x00;       //初始化中断标志位, P1IFG:  P1 中断标
志
22.    }
23.    /*********************************
24.     *  直流电机传感器初始化程序
25.    *********************************/
26.    void InitIO(void)
27.    {
28.      P0DIR |= 0xC0;           //P0_6,P0_7 为输出
```

```
29.     P0 = 0x00;
30.   }
31.   /*****************************
32.    *     中断服务函数
33.   *****************************/
34.   #pragma vector = P1INT_VECTOR
35.   __interrupt void P1_ISR(void)
36.   {
37.     if(P1IFG > 0)             //按键中断
38.     {
39.       P1IFG = 0;
40.       WaitMs(25);             // 消除抖动
41.       if (P1IFG == 0)
42.       {
43.         if (flg == 0)
44.         {
45.           dir ++;
46.           flg = 1;
47.         }
48.       }
49.     }
50.     P1IF = 0;               //清中断标志
51.   }
52.   /*****************************
53.    *       主函数
54.   *****************************/
55.   void main(void)
56.   {
57.     InitIO();
```

```
58.     InitKeyINT();              //初始化按键中断
59.     while(1)
60.     {
61.       if (flg)
62.       {
63.         P0_6 = 0;
64.         P0_7 = 0;
65.         switch (dir)
66.         {
67.         case 1:
68.           P0_6 = 1;
69.           break;
70.         case 3:
71.           P0_7 = 1;
72.           break;
73.         case 4:
74.           dir = 0;
75.         }
76.         flg = 0;
77.       }
78.     }
79.   }
```

我们来看主函数：

第 57~58 行：进行一些初始化工作。

第 59~78 行：循环判断是否有中断被触发，根据 S3 按下触发的情况，从而控制直流电机的转动。

二：将程序添加到协议栈代码中

直流电机模块的电路是对 IO 口电平的检测。所以在协议栈里对直流电机 IO 口的电平变化即可控制电机的正反转。我们只需要在终端配置好 IO 口，然后协调器端进行串口输入

指令来改变 IO 的电平就可以了。

1） 打开例程 SampleApp.eww 工程，打开 SampleApp.c 文件，下图 6-62 所示。

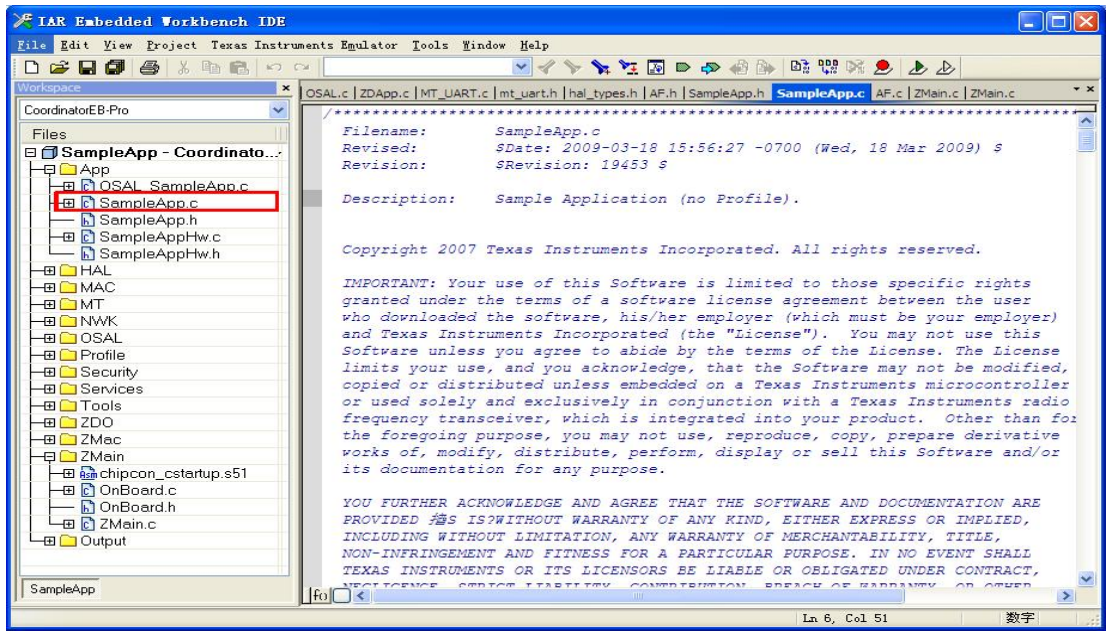


图 7- 4

2） 首先需要使用到串口的功能，所以要定义串口的头文件和初始化串口，由于是通过串口输入指令进行直流电机的控制，因此必须注册串口回调函数用于接收串口输入，代码如下所示

```
#include "hal_uart.h"
void SampleApp_Init( uint8 task_id )
{
  SampleApp_TaskID = task_id;
  SampleApp_NwkState = DEV_INIT;
  SampleApp_TransID = 0;

  P0DIR |= 0xC0;                    //将直流电机的 IO 口 P0_6,P0_7 定义为输出
  P0 = 0x00;

  //开启串口配置
```

```
  static halUARTCfg_t uartConfig;
  uartConfig.configured = TRUE;
  uartConfig.baudRate = HAL_UART_BR_115200;          //串口波特率
  uartConfig.flowControl = FALSE;                    //关闭串口流控
  uartConfig.callBackFunc = Revice_From_PC;          //注册串口回调函数
  HalUARTOpen(0,&uartConfig);                        //打开串口配置
  HalUARTWrite(0,"Hello World\n",12);
          ...
}
```

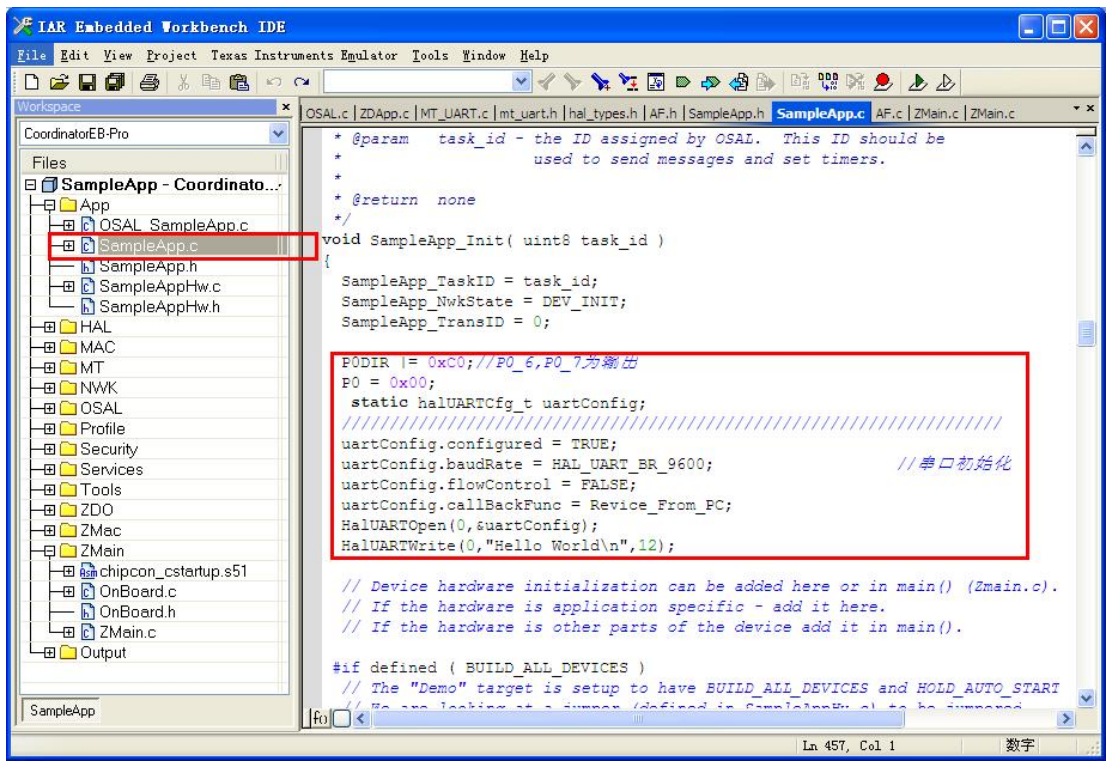


图 7- 5

3) 在串口回调函数中可以看到这里将我们串口输入的指令当形参给点对点的消息发送函数 SampleApp_SendPointToPointMessage ()，关键代码如下：

```
void Revice_From_PC(unsigned char port, unsigned char event)
{
  UINT8 Rx_Count;
  (void)port;
  unsigned char From_Pc_Cmd[128];
```

```
    if (event & (HAL_UART_RX_FULL | HAL_UART_RX_ABOUT_FULL |
HAL_UART_RX_TIMEOUT))
    {
     Rx_Count = Hal_UART_RxBufLen ( 0 );
     HalUARTRead(0,From_Pc_Cmd,Rx_Count);                    //读取串口输入
信息
     SampleApp_SendPointToPointMessage (From_Pc_Cmd, Rx_Count); //信息发送函
数
    }
  }
```

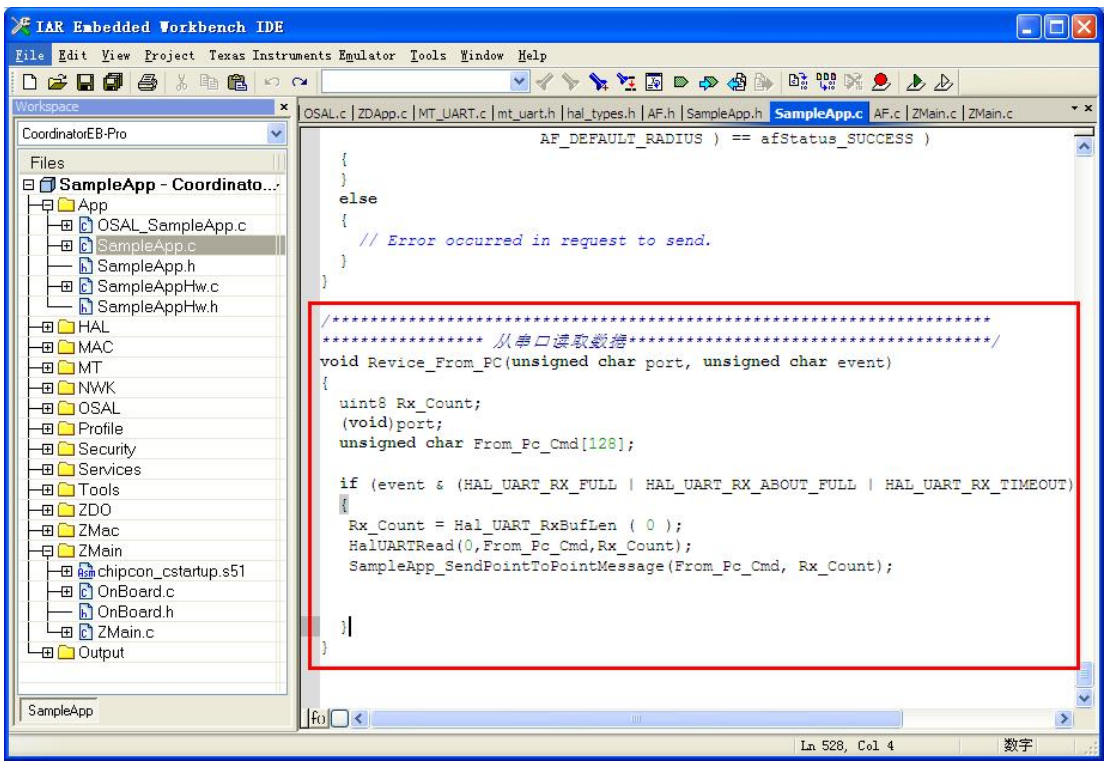


图 7- 6

4) 进入 SampleApp_SendPointToPointMessage 函数可看到，在发送消息函数中进行协调器点播消息，通过 AF_DataRequest 将串口输入数据发送到终端设备：

```
void SampleApp_SendPointToPointMessage (uint8 *PC_CMD, uint8 Rx_Count)
{
  uint8 Temp[128];
```

```
      uint8 i;
     for(i=0;i<Rx_Count;i++)                    //串口输入的数据长度
     {
       Temp[i]=PC_CMD[i];                       //串口输入的数据
     }
      afAddrType_t my_DstAddr;
      my_DstAddr.addrMode=(afAddrMode_t)Addr16Bit;
      my_DstAddr.endPoint=SAMPLEAPP_ENDPOINT;    //指定终端节点
      my_DstAddr.addr.shortAddr=0xFFFF;
      //AF_DataRequest 数据发送函数
      if ( AF_DataRequest( &my_DstAddr, &GenericApp_epDesc,
                           SAMPLEAPP_POINT_TO_POINT_CLUSTERID,
                           Rx_Count,
                           Temp,
                           &GenericApp_TransID,
                           AF_DISCV_ROUTE, AF_DEFAULT_RADIUS ) ==
afStatus_SUCCESS )
      {
        HalLedBlink(HAL_LED_1,0,50,500);          //点亮 LED1 表示数据已经发送
      }
      else
      {    }
    }
```

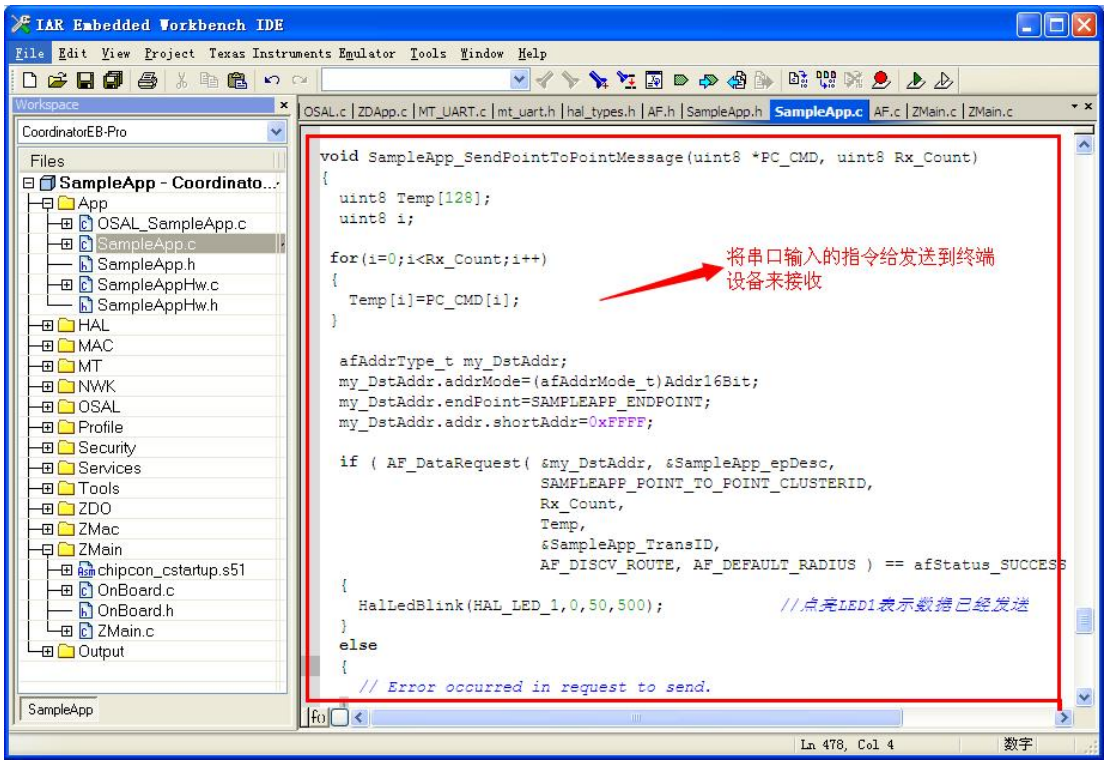


图 7- 7

5）协调器已经将串口输入的信息进行发送，所以我们只要到终端设备的接收函数中进行数据的操作就可以完成继电器传感器的数据采集，代码如下：

```
void SampleApp_MessageMSGCB( afIncomingMSGPacket_t *pkt )
{
  uint16 flashTime;
  switch ( pkt->clusterId )
  {
    case SAMPLEAPP_POINT_TO_POINT_CLUSTERID:      //对应协调器发送端的簇
      //将数据和数据的长度 copy 到 buffer 存储区，下面只用到 Data 数据
      osal_memcpy(buffer,pkt->cmd.Data,pkt->cmd.DataLength);
      P0_6 = 0;                                   //对直流电机 P0_6 和 P0_7 进行关闭
      P0_7 = 0;
     if(buffer[0] == 0x00)                        //如果指令为 00，直流电机关闭
     {
      P0_6 = 0;
      P0_7 = 0;
```

```
        }
        else if(buffer[0] == 0x01)          //如果指令为 01，直流电机左转
        {
          P0_6 = 1;
          P0_7 = 0;
        }
         else if(buffer[0] == 0x02)         //如果指令为 02，直流电机右转
        { P0_6 = 0;
          P0_7 = 1;
        }
         break;
       }
  }
```

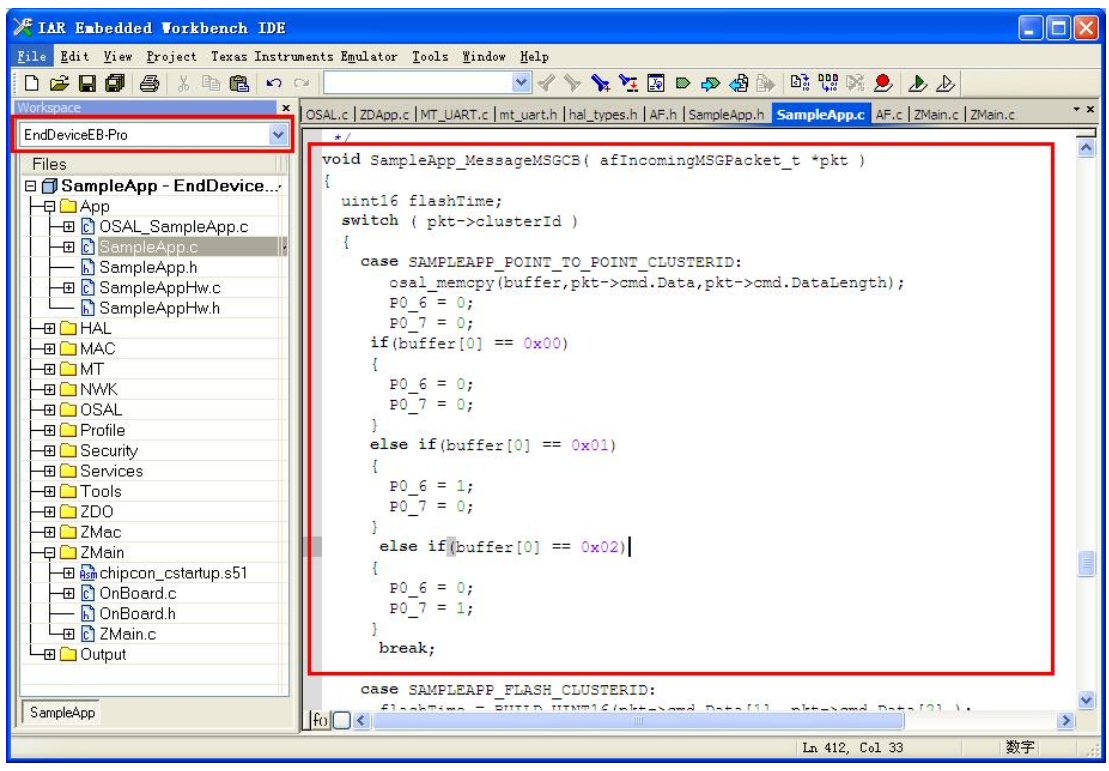


图 7- 8

最后，注意的加上串口回调函数和点对点发送函数的声明，接收函数中的 buffer 为全局变量，如下图 7- 9 所示：

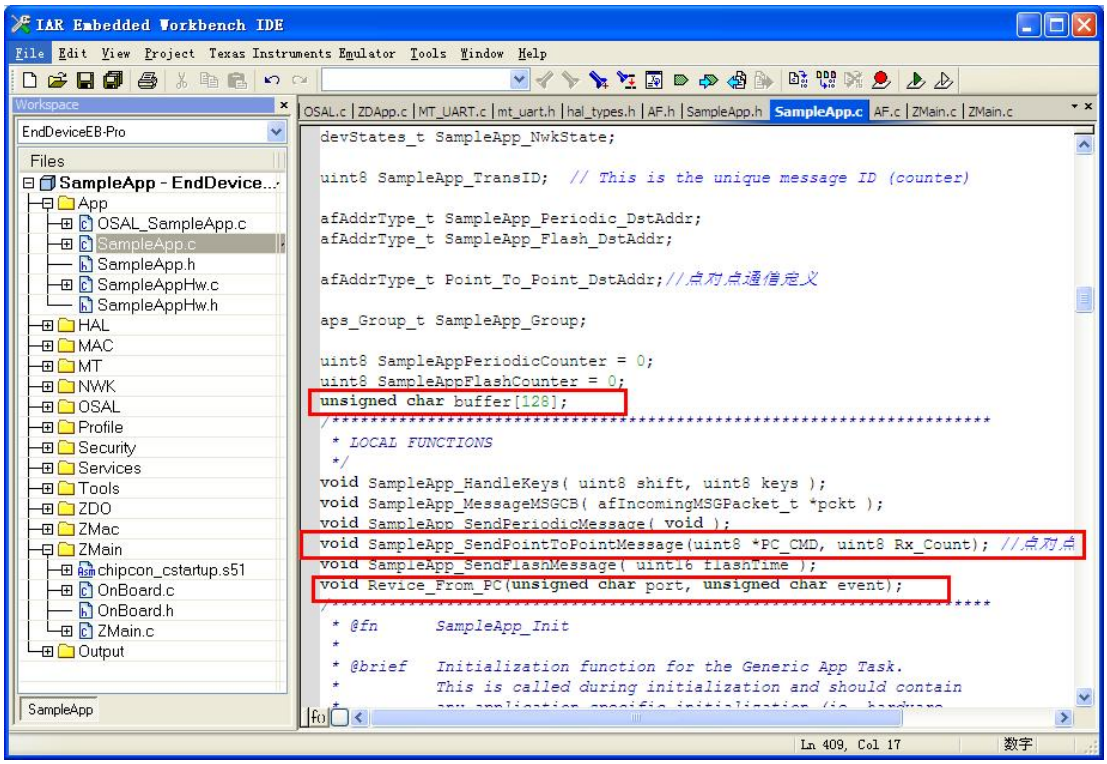


图 7- 9

实验现象：下载程序到终端（带直流电机模块传感器）和协调器，协调器进行串口输入指令进行控制直流电机，且协调器端如果串口接收的数据已发送进行闪烁 LED1：

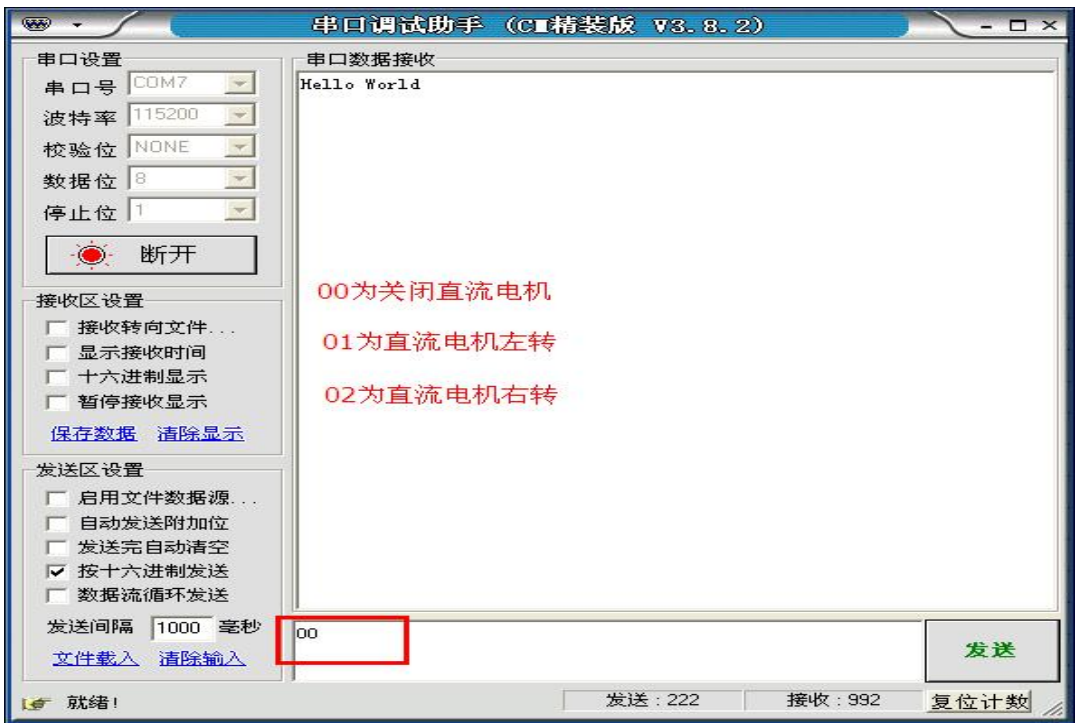


图 7-10

1.8. 实验八：继电器控制

前言：这一节我们我学习传感器和执行器部分内容中的继电器模块。

执行器介绍：继电器(Relay)，也称电驿，是一种电子控制器件，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），通常应用于自动控制电路中，它实际上是用较小的电流去控制较大电流的一种“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 8- 1 继电器

实现平台： ZigBee 协调器和传感器节点；

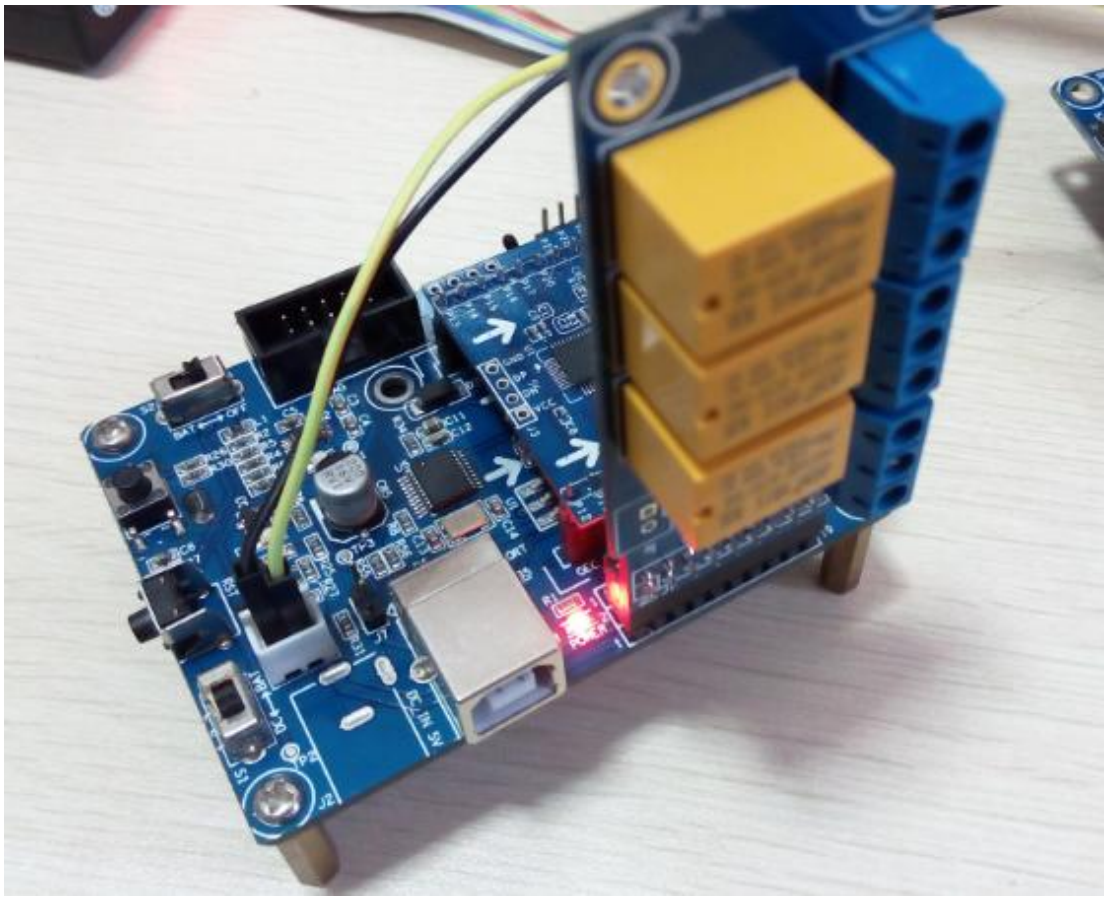


图 8- 2 传感器节点和执行器

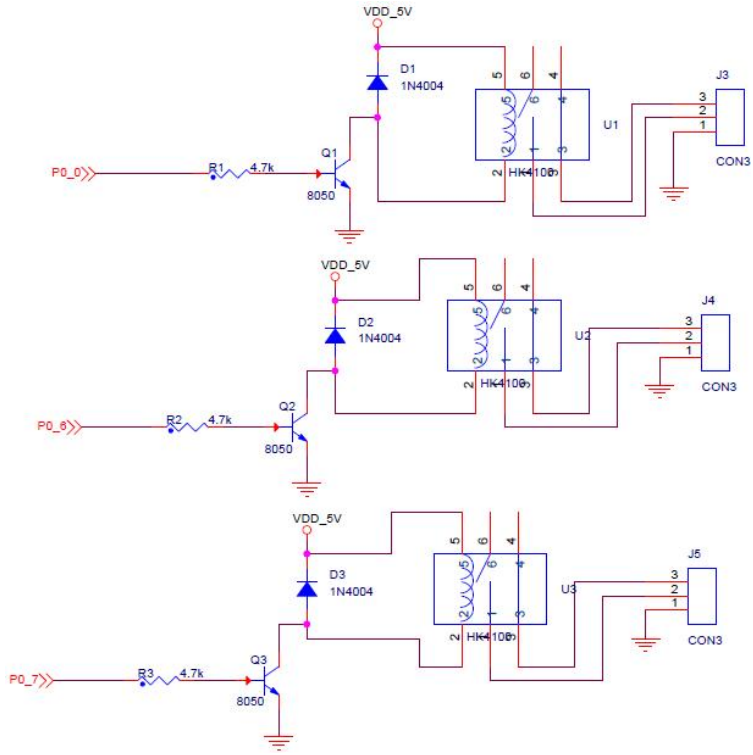


图 8- 3 继电器电路

实验现象：通过串口终端输入控制指令，实现直对继电器的开、关控制；

实验讲解：像前面执行器例程一样，我们先实现裸机程序里对继电器控制。然后在协议栈里添加相应的代码。

一：在裸机上完成对继电器控制的驱动。

打开配套程序下裸机文件夹—继电器下的工程文件，看到函数如下：

```
/*************************************
 * project：继电器模块测试程序
 * 作者    ：GEC
*************************************/

#include <ioCC2530.h>

typedef unsigned char uint8;
```

```
typedef unsigned int uint16;
/*****************************************************************
*函数功能 : 主函数                                               *
*入口参数 : 无                                                   *
*返 回 值 : 无                                                   *
*说    明 : 无                                                   *
*****************************************************************/
void main(void)
{
  InitIO();
  InitKeyINT();          //初始化按键中断

  while(1);
}
/*****************************************************************
*函数功能 : 继电器传感器 IO 初始化
*
*入口参数 : 无                                                   *
*返 回 值 : 无                                                   *
*说    明 : 无                                                   *
*****************************************************************/
void InitIO(void)
{
    P0DIR |= 0xC1;        // P0.0 P0.6 P0.7
    P0 = 0x00;            // 全关
}

/*********************************
 * 初始化按键为中断输入方式
```

```
28. **********************************/
29. void InitKeyINT(void)
30. {
31.   P1SEL &= ~0x04;
32.   P1DIR &= ~0x04;          //初始化 P1_2 为输入
33.   P1INP &= ~0x04;          //上拉
34.   P1IEN |= 4;              //P1.2 设置为中断方式
35.   PICTL |= 0X01;           //下降沿触发
36.   EA = 1;
37.   IEN2 |= 0X10;            // P1 设置为中断方式;
38.
39.   P1IFG |= 0x00;           //初始化中断标志位
40. }
41. /*****************************
42.  * @brief   中断服务
43. *****************************/
44. #pragma vector = P1INT_VECTOR
45. __interrupt void P1_ISR(void)
46. {
47.   if(P1IFG > 0)              //按键中断
48.   {
49.     P1IFG = 0;
50.     WaitMs(25);              // 消除抖动
51.     if (P1IFG == 0)
52.     {
53.       P0 ^= 0xC1;            //将 P0_0,P0_6,P0_7 进行异或
54.     }
55.   }
56.   P1IF = 0;                  //清中断标志
```

```
57. }
```

同样是简单几行代码，就完成了对继电器的读取。大家可以在工程里进入具体函数看代码，理解继电器的读取过程，按键可控制继电器。

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串口回调函数读取串口输入数据。

首先理清一下思路，我们要实验的功能是协调器端在串口调试助手上面输入串口指令，控制终端节点继电器传感器进行工作。这就实现了无线控制。

1）在初始化函数 GenericApp_Init 中进行继电器 IO 口的初始化、串口处理函数和注册串口回调函数，关键代码如下：

```
    void GenericApp_Init( byte task_id )
    {
        ...
  P0SEL &= 0x00;                                  //初始化 P0 口为通用 I/O 口
      P0DIR |= 0xC1;                                  // P0.0 P0.6 P0.7 为继电器
IO
      P0 = 0;
  static halUARTCfg_t uartConfig;
      uartConfig.configured = TRUE;                   //开启串口配置
      uartConfig.baudRate = HAL_UART_BR_115200;       //串口波特率的设置
      uartConfig.flowControl = FALSE;                 /关闭串口流控
      uartConfig.callBackFunc = Revice_From_PC;       //注册串口回调函数
```

```
    HalUARTOpen(0,&uartConfig);                    //打开串口配置
    HalUARTWrite(0,"GEC Hello\n",10);
      ...
// 点对点通讯定义
    Point_To_Point_DstAddr.addrMode = (afAddrMode_t)Addr16Bit;
    Point_To_Point_DstAddr.endPoint = SAMPLEAPP_ENDPOINT;
    Point_To_Point_DstAddr.addr.shortAddr = 0x0000;//协调器地址

}
```

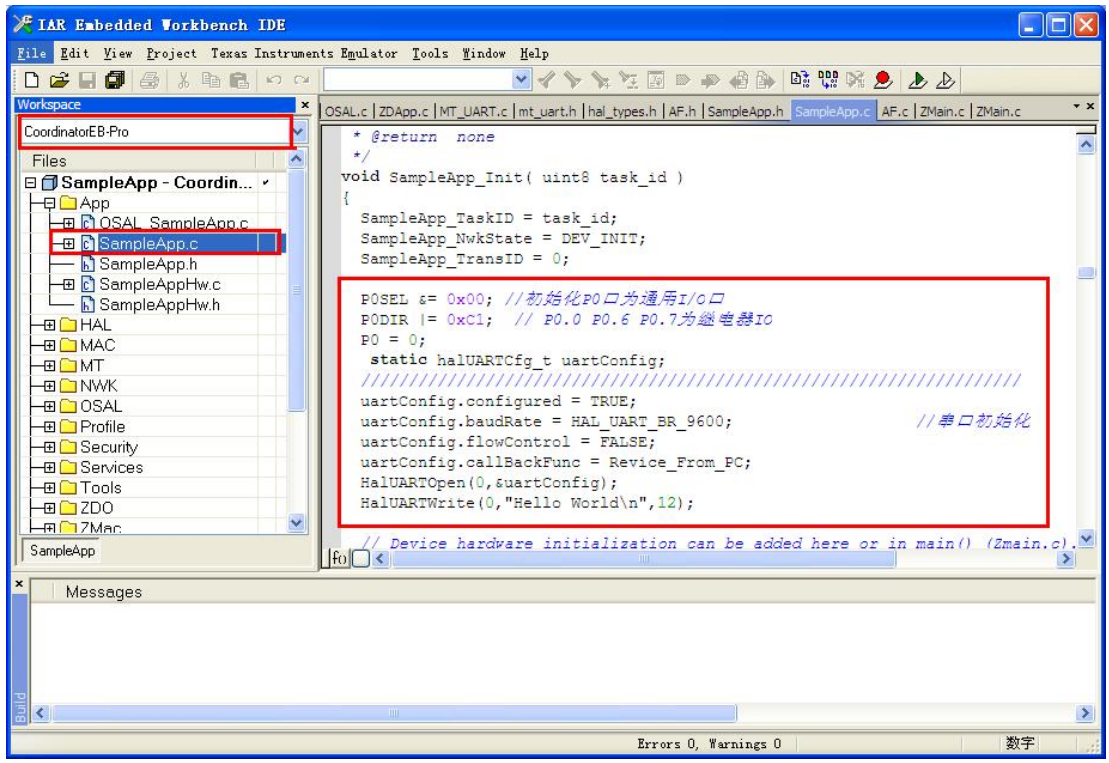


图 8- 4

2）在串口回调函数 Revice_From_PC 中可以看到这里将我们串口输入的指令当形参给点对点的消息发送函数 SampleApp_SendPointToPointMessage ()，关键代码如下：

```
void Revice_From_PC(unsigned char port, unsigned char event)
{
   UINT8 Rx_Count;
```

```
    (void)port;
    unsigned char From_Pc_Cmd[128];

    if (event & (HAL_UART_RX_FULL | HAL_UART_RX_ABOUT_FULL |
HAL_UART_RX_TIMEOUT))
    {

     Rx_Count = Hal_UART_RxBufLen ( 0 );
     HalUARTRead(0,From_Pc_Cmd,Rx_Count);                    /读取串口输入
信息
     SampleApp_SendPointToPointMessage(From_Pc_Cmd, Rx_Count);//信息发送函数

    }
  }
```

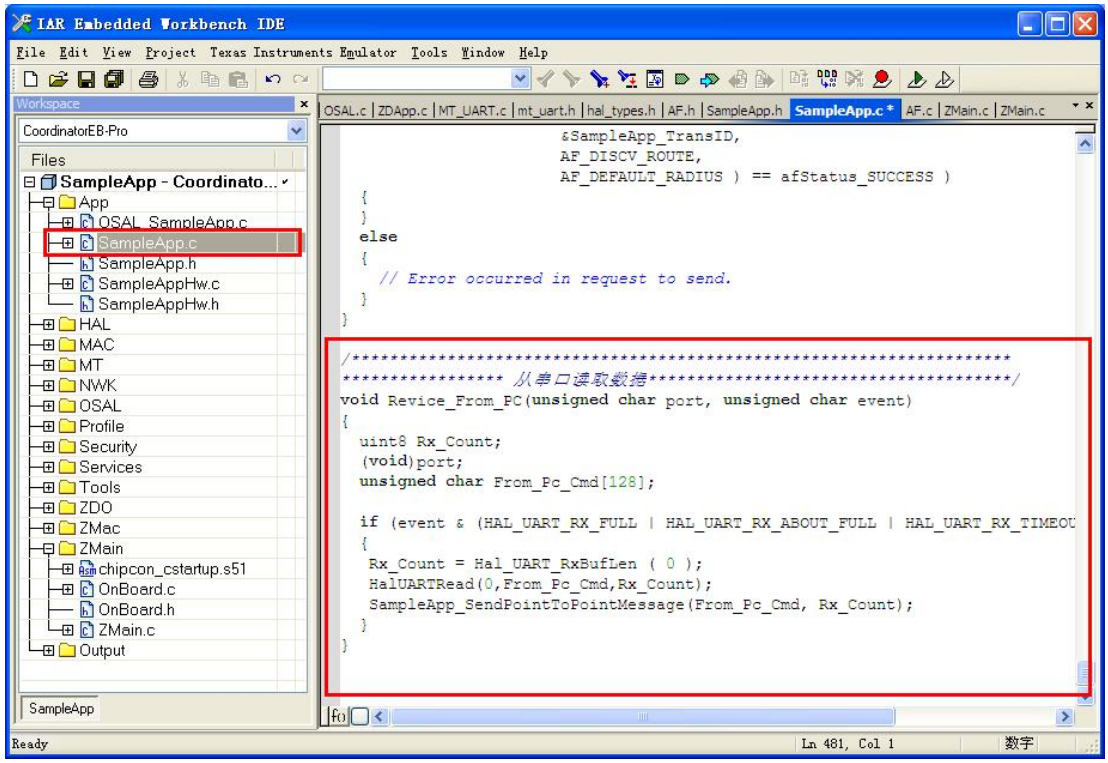


图 8- 5

进入 SampleApp_SendPointToPointMessage()函数可看到，在发送消息函数中进行协

调器点播消息，通过 AF_DataRequest 将串口输入数据发送到终端设备。

```
    void SampleApp_SendPointToPointMessage(uint8 *PC_CMD, uint8 Rx_Count)
    {
        uint8Temp[128];
        uint8i;
        for(i=0;i<Rx_Count;i++)                          //串口输入的数据长度
        {
           Temp[i]=PC_CMD[i];                            //串口输入的数据
    }
      afAddrType_t my_DstAddr;
      my_DstAddr.addrMode = (afAddrMode_t)Addr16Bit;
      my_DstAddr.endPoint =  SAMPLEAPP_ENDPOINT;    //指定发送到终端设备
      my_DstAddr.addr.shortAddr=0xFFFF;
      //AF_DataRequest 数据发送函数
      if ( AF_DataRequest( &my_DstAddr, &GenericApp_epDesc,
                           SAMPLEAPP_POINT_TO_POINT_CLUSTERID,
                           Rx_Count,
                           Temp,
                           &GenericApp_TransID,
                           AF_DISCV_ROUTE, AF_DEFAULT_RADIUS ) ==
afStatus_SUCCESS )
      {
        HalLedBlink(HAL_LED_1,0,50,500);          //点亮 LED1 表示数据已经发送
      }
      else
      {
        // Error occurred in request to send.
      }
    }
```

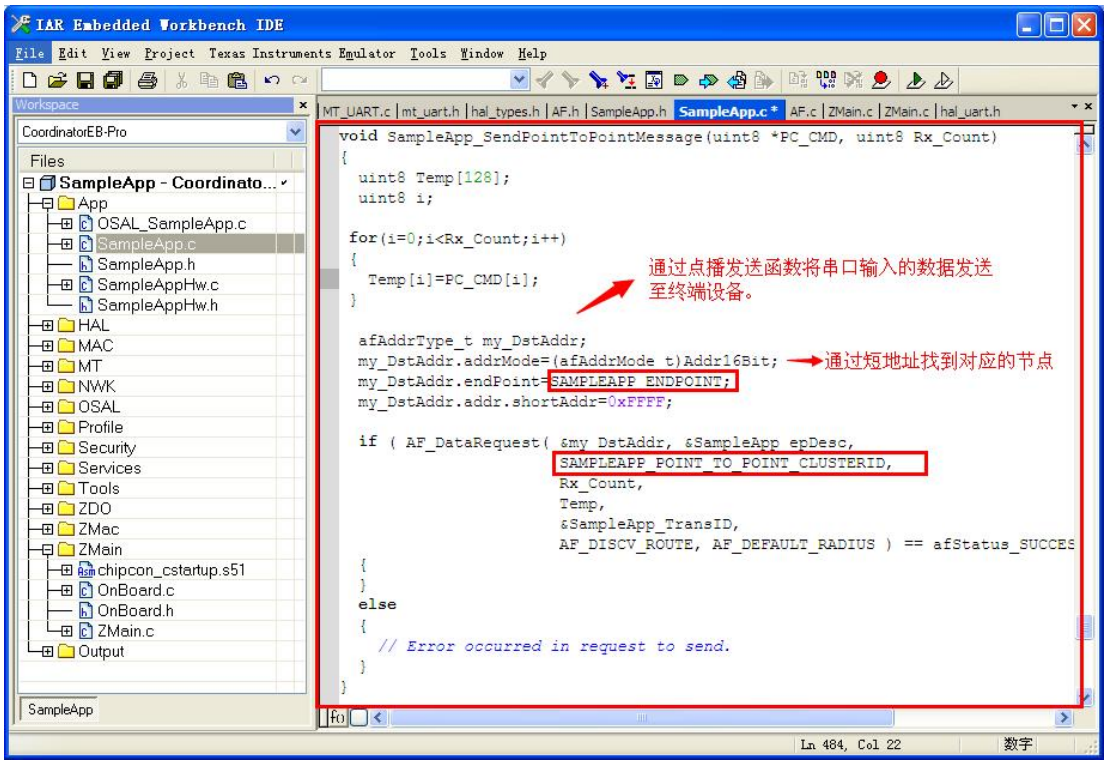


图 8- 6

3）协调器已经将串口输入的信息进行发送，所以我们只要到终端设备的接收函数中进行数据的操作就可以完成继电器传感器的数据采集，代码如下：

```
void SampleApp_MessageMSGCB( afIncomingMSGPacket_t *pkt )
{
  switch ( pkt->clusterId )
  {
case SAMPLEAPP_POINT_TO_POINT_CLUSTERID://同协调器发送函数的簇对应
osal_memcpy(buffer,pkt->cmd.Data,pkt->cmd.DataLength);

if(buffer[0] == 0x00)                    //判定接收到的数据的第 0 个字节
    {
   P0_0 = 1;
       P0_6 = 1;                          //00 将继电器全关
       P0_7 = 1;
    }
```

```
else if(buffer[0] == 0x01)              //0101 时打开继电器 1
 {
   P0_0 = 0;
   if(buffer[1] == 0x00)                //0100 时关闭继电器 1
   {
     P0_0 = 1;
   }
 }
  else if(buffer[0] == 0x02)             //0201 时打开继电器 2
 {
   P0_6 = 0;
   if(buffer[1] == 0x00)                //0200 时关闭继电器 2
   {
     P0_6 = 1;
   }
 }
 else if(buffer[0] == 0x03)             //0301 时打开继电器 3
 {
   P0_7 = 0;
   if(buffer[1] == 0x00)                //0300 时关闭继电器 3
   {
     P0_7 = 1;
   }

 }
 else if(buffer[0] == 0x04)
 {
   P0_7 = 0;
   P0_6 = 0;
```

```
            P0_0 = 0;


          }
           break;
      }
  }
```

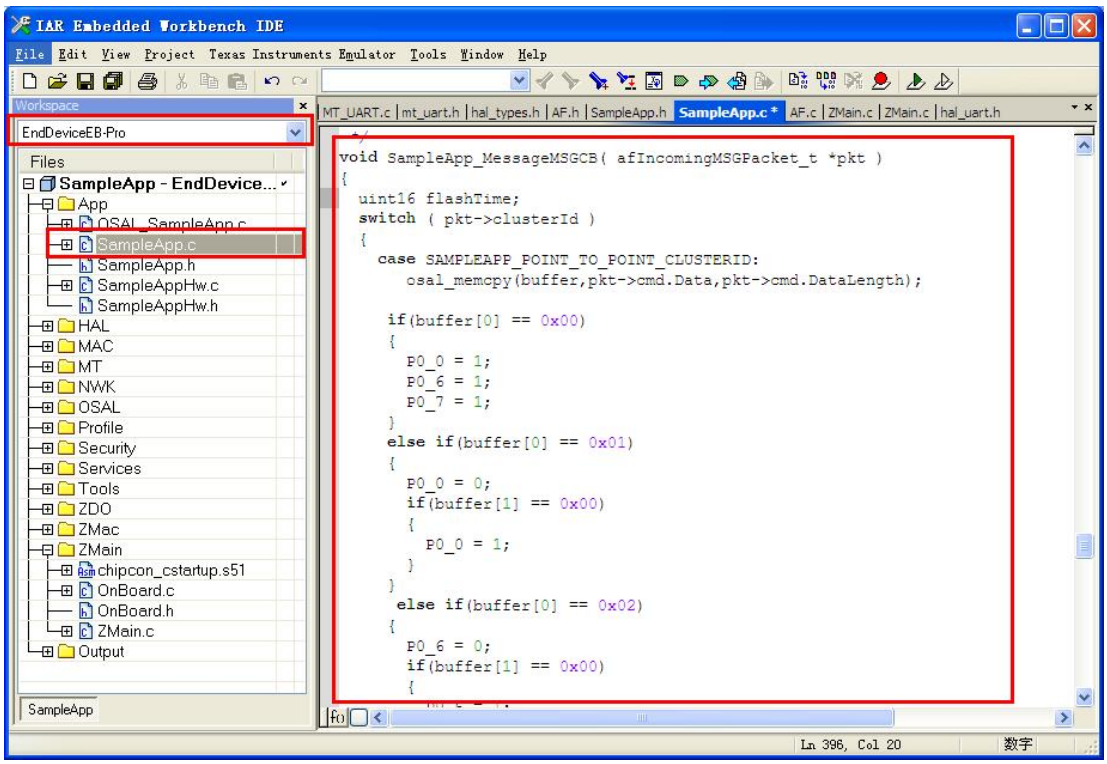


图 8- 7

实验现象：下载程序到终端（带 3 路继电器模块）和协调器，协调器通过串口输入指令对终端设备进行控制，达到无线控制的效果，继电器可以接收各种电器设备进行控制。继电器一接上电源时默认是打开的，所以首先要关闭继电器，再进行控制具体的某路继电器：

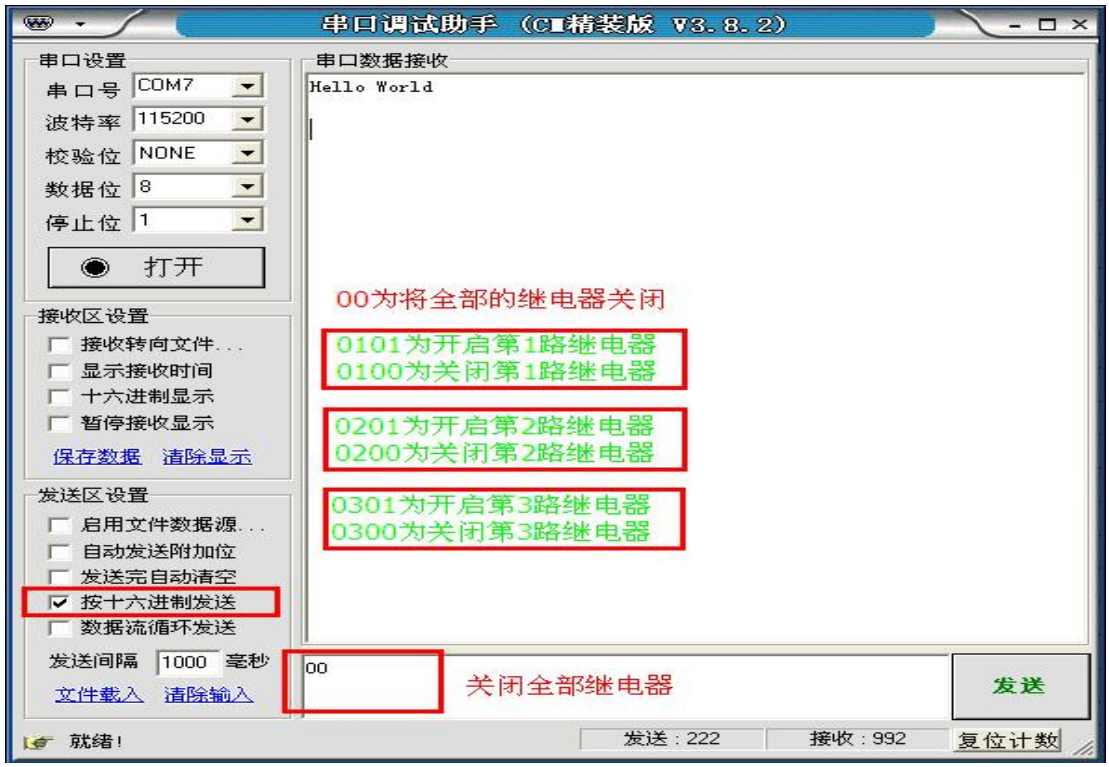
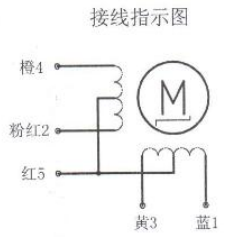


图 8- 8

1.9. 实验九：步进电机模块

前言：这一节我们我学习传感器和执行器部分内容中的步进电机模块。

执行器介绍：步进电机是一种感应电机，它的工作原理是利用电子电路，将直流电变成分时供电的，多相时序控制电流，用这种电流为步进电机供电，步进电机才能正常工作，驱动器就是为步进电机分时供电的，多相时序控制器。CC2530 本身的驱动电流很小，所以需要 ULN2003 芯片驱动。



连线序号	导线颜色	分配顺序							
		1	2	3	4	5	6	7	8
5	红	+	+	+	+	+	+	+	+
4	橙	-	-						-
3	黄		-	-	-				
2	粉红				-	-	-		
1	蓝						-	-	-

从输出轴方向看 —— 逆时针方向

图 9- 1 步进电机

实现平台： ZigBee 协调器和传感器节点；

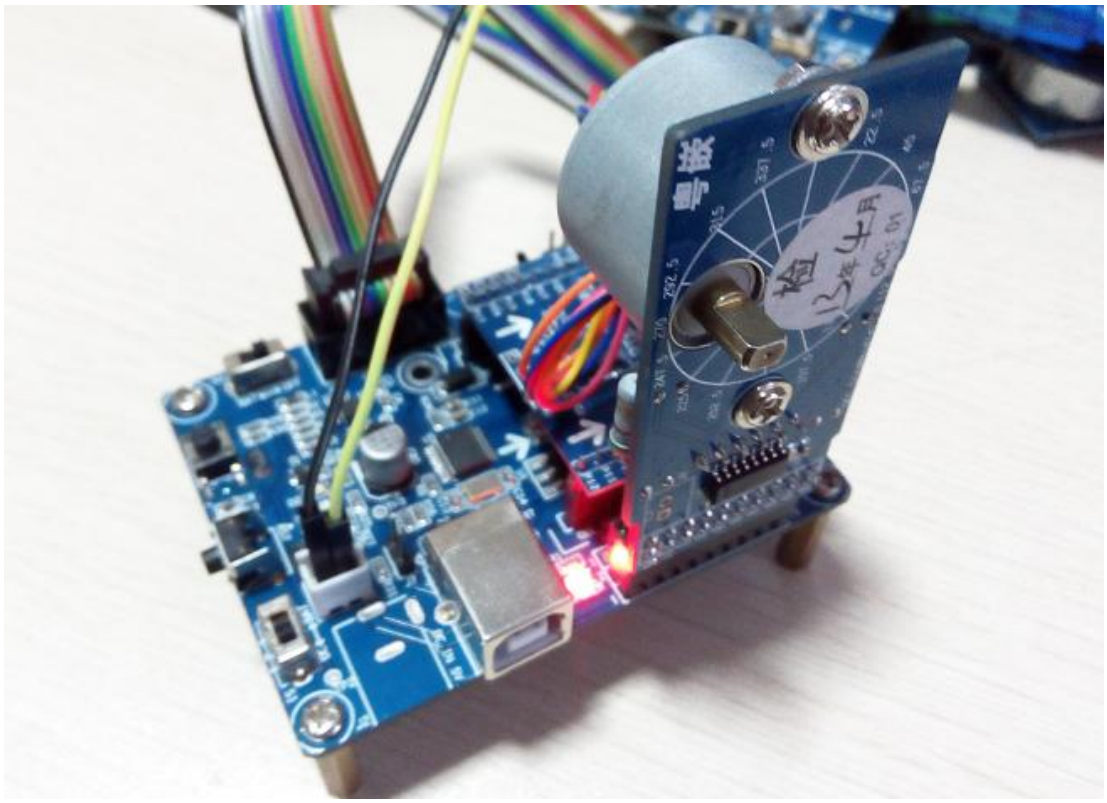


图 9- 2 传感器节点

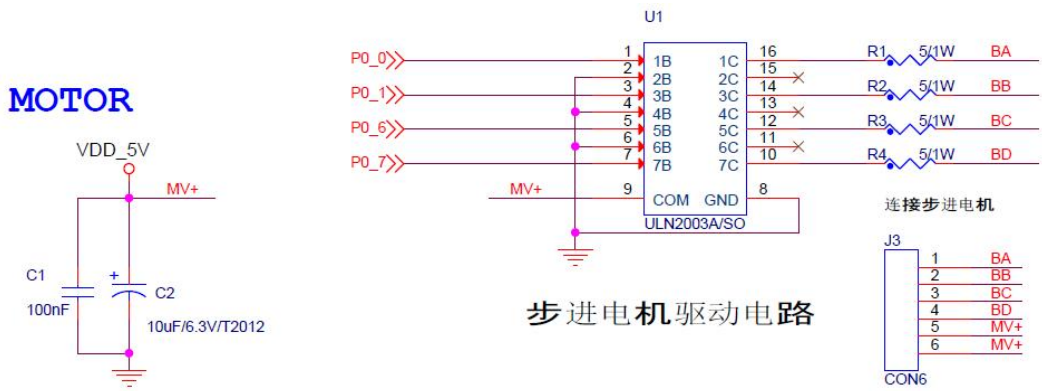


图 9- 3 步进电机电路

实验现象：通过串口终端输入控制指令，实现步进电机的正转和反转、停止转动；

实验讲解：像前面传感器例程一样，我们先实现裸机程序里对步进电机控制。然后在协议栈里添加相应的代码。

一：在裸机上完成对步进电机控制的驱动。

打开配套程序下裸机文件夹—步进电机下的工程文件，看到函数如下：

```
/*****************************************/
/*  Zigbee 学习例程
/*例程名称：步进电机模块
*****************************************/
#include <ioCC2530.h>
typedef unsigned char uint8;
typedef unsigned int uint16;
uint8 motorDirs[2][4] = {
  {0x01, 0x02, 0x40, 0x80},
     {0x01, 0x80, 0x40, 0x02}
  };
  uint8 motorDir = 0;
  uint16 motorCnt = 0;
  /**********************************
   * @brief 初始化按键为中断输入方式
  **********************************/
  void InitKeyINT(void)
  {
    P1DIR &= ~0x04;
    P1INP |= 0x04;              //上拉
```

```
21.     P1IEN |= 0x04;              //P1.3 设置为中断方式
22.     PICTL |= 0X01;              //下降沿触发
23.     EA = 1;
24.     IEN2 |= 0X10;               // P1 设置为中断方式;
25.     P1IFG |= 0x00;              //初始化中断标志位
26.   }
27.   /*********************************************
28.    * 初始化程序,将 P1.0、P1.1、P1.4 定义为输出口
29.   *********************************************/
30.   void InitIO(void)
31.   {
32.     P0DIR |= 0xC3;
33.     P0 = 0x00;
34.   }
35.   /*****************************
36.    *  中断服务函数
37.   *****************************/
38.   #pragma vector = P1INT_VECTOR
39.   __interrupt void P1_ISR(void)
40.   {
41.     if(P1IFG > 0)                 //按键中断
42.     {
43.       P1IFG = 0;
44.       WaitMs(25);                 // 消除抖动
45.       if (P1IFG == 0)
46.       {
47.           motorDir ^= 1;
48.       }
49.     }
```

```
50.     P1IF = 0;                           //清中断标志
51.   }
52.   /*************************
53.    *    主函数
54.   *************************/
55.   void main(void)
56.   {
57.     InitIO();                         //初始化步进电机 IO 口
58.     InitKeyINT();                       //初始化按键中断

59.    while(1)
60.     {
61.       motorCnt = 1000;
62.       while (motorCnt)
63.       {
64.         P0 &= ~0xC3;
65.         P0 |= motorDirs[motorDir][motorCnt&3];  //步进电机控
制
66.         WaitMs(1);
67.         motorCnt --;
68.       }
69.     }
70.   }
```

二：将程序添加到协议栈代码中

步进电机模块控制电路是对四相五线的 IO 口脉冲信号的检测。所以在协议栈里为了达到控制步进电机的效果。我们只需要配置好相对应的的正反转表格，然后通过定时器触发和串口输入指令进行控制就可以了。本实验是建立直流电机代码的基础上进行创建的，所以关于串口接收的内容请看实验十，现在就开始步进电机的控制吧

1）由于控制步进电机需要用定时器来控制电机的运行时序，像前面用到串口一样要定义串口头文件，实验这里必须定义定时器头文件和定时器的配置，关键代码如下：

```
#include "hal_timer.h"
void SampleApp_Init( uint8 task_id )
{
    ...
  MOTOR_INIT();
  HalTimerInit();
  HalTimerConfig(HAL_TIMER_0, HAL_TIMER_MODE_CTC,
                 HAL_TIMER_CHANNEL_SINGLE,
                 HAL_TIMER_CH_MODE_OUTPUT_COMPARE,
                 TRUE, stepper);
    ...
}
```

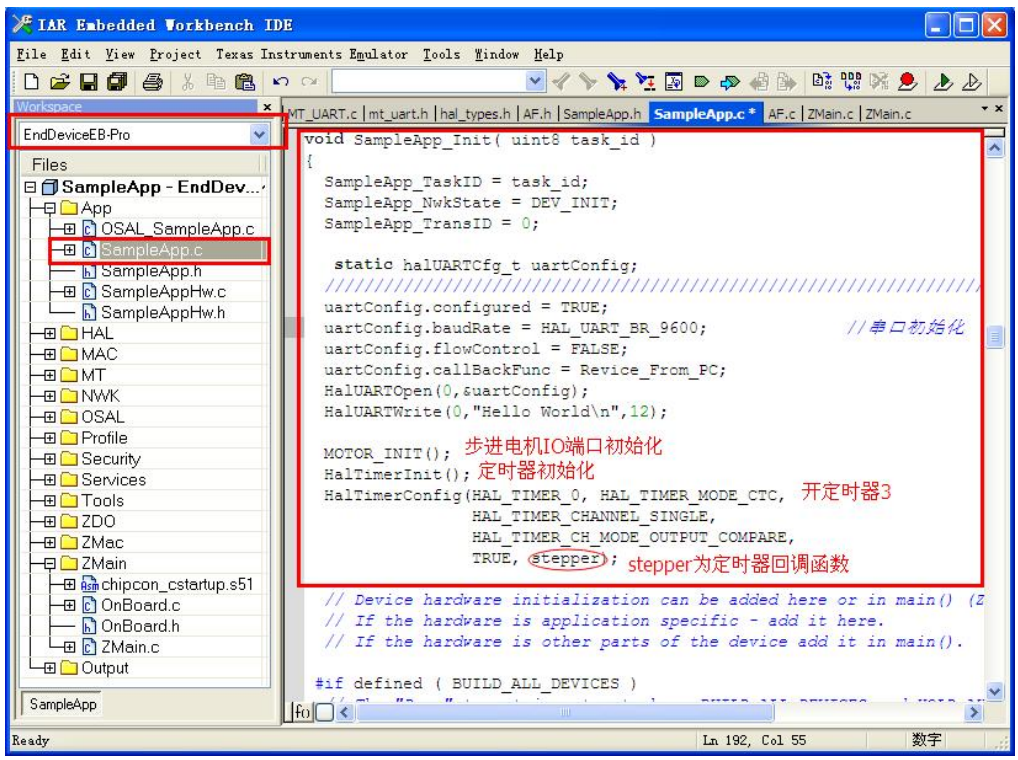


图 9- 4

2）声明定时器回调函数、步进电机的 IO 端口初始化和步进电机脉冲控制时序表，关

键代码如下：

```
const uint8 tbMotorWard[2][4] = {
                                {0x01, 0x02, 0x40, 0x80},    //正转
                                {0x01, 0x80, 0x40, 0x02}     //反转
                                };
static uint16 motorCnt = 0;
static uint8  motorDir = 0;
static uint8  motorPeriod = 10;

//定时器的回调函数，用于进行电机正反转和时序的控制
static void stepper(uint8 timerId, uint8 channel, uint8 channelMode);

//将步进电机的 IO 端口初始化以宏定义的方式来实现
#define MOTOR_INIT() {P0SEL &= ~0xC3; P0DIR |= 0xC3; P0 &= ~0xC3;}
```

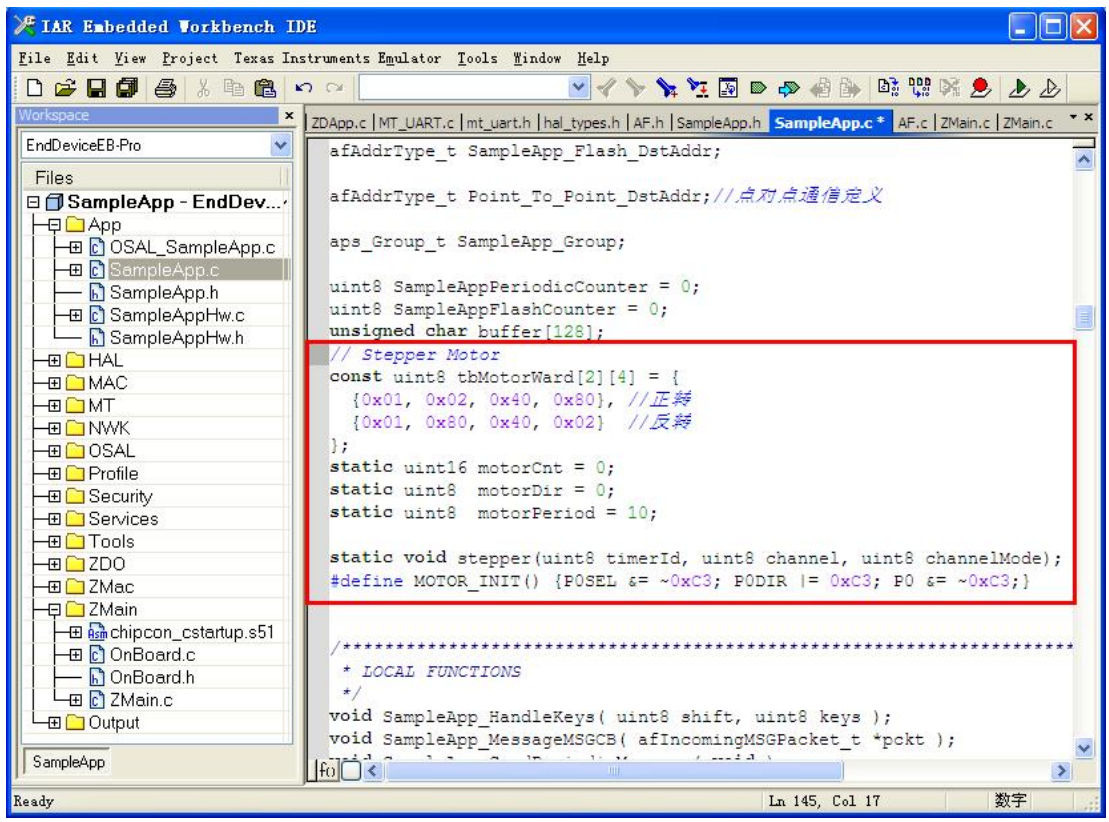


图 9- 5

3）定时器回调函数究竟有什么作用，为什么要使用定时器？定时器回调函数是为了实现对步进电机的转向和转动速度的控制，使用定时器是为了更好的控制步进电机相与相之间的时隙，达到理想的控制效果，关键代码如下：

```
static void stepper(uint8 timerId, uint8 channel, uint8 channelMode)
{
  static uint8 cnt;
  if (timerId == 0)
  {
    if (motorCnt > 0)
    {
      cnt ++;
      if (cnt >= motorPeriod)
      {
        P0 &= ~0xC3;
        P0 |= tbMotorWard[motorDir][motorCnt&3];
        motorCnt --;
        cnt = 0;
      }
    }
    else
    {
      HalTimerStop(HAL_TIMER_0);
    }
  }
}
```

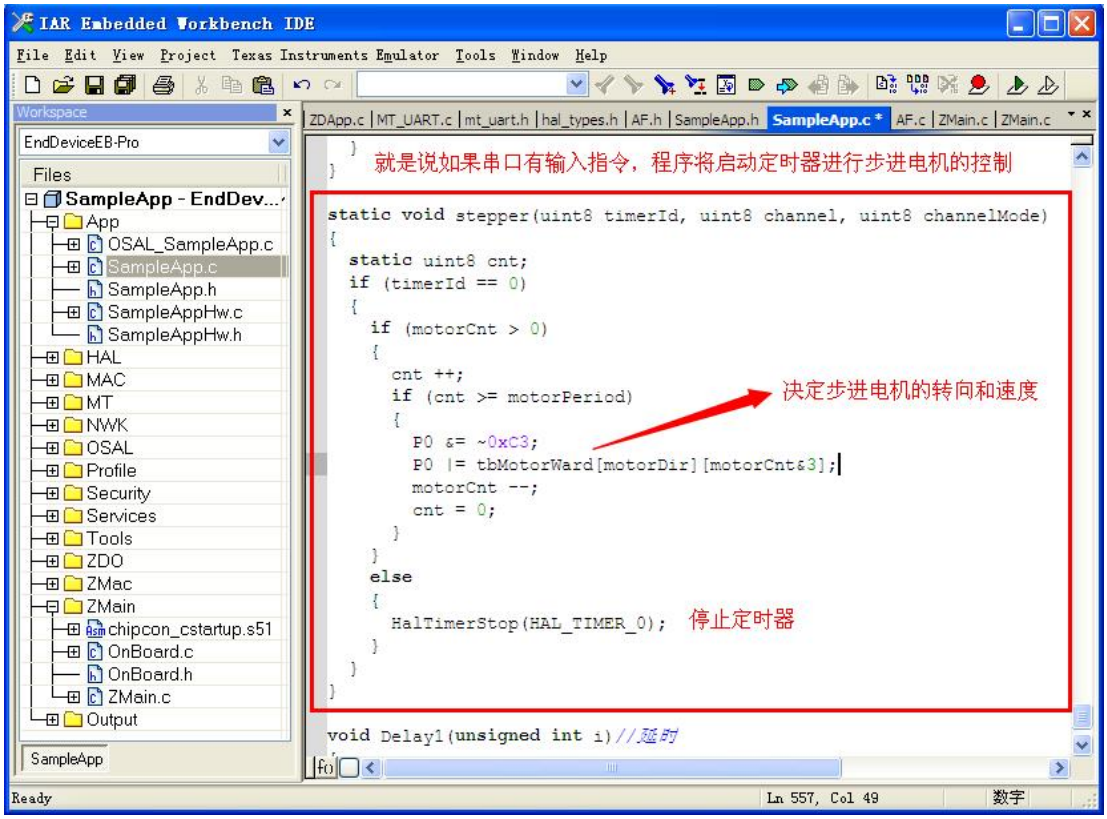


图 9- 6

4）终端设备(带步进电机模块)在接收到协调器通过串口发送过来的指令后，进行相对应的动作，关键代码如下。

```
void SampleApp_MessageMSGCB( afIncomingMSGPacket_t *pkt )
{
  switch ( pkt->clusterId )
  {
    case SAMPLEAPP_POINT_TO_POINT_CLUSTERID:
       osal_memcpy(buffer,pkt->cmd.Data,pkt->cmd.DataLength);
     if(buffer[0]>0)
     {
       MOTOR_INIT();                              //初始化步进电机 IO 端口
       motorDir = buffer[0]&0x80? 1: 0;           //对第一个指令进行操作
       motorCnt = BUILD_UINT16(buffer[2], buffer[1]) << 2;
```

```
            motorPeriod = buffer[0]&0x7F;                    //对电机周期进行设置
            HalTimerStop(HAL_TIMER_0);                       //首先关闭定时器

        //定时器开启时会调用定时器回调函数 stepper()对步进电机进行控制,图 12- 6
            HalTimerStart(HAL_TIMER_0, 1000);
          }
           break;
    }
```

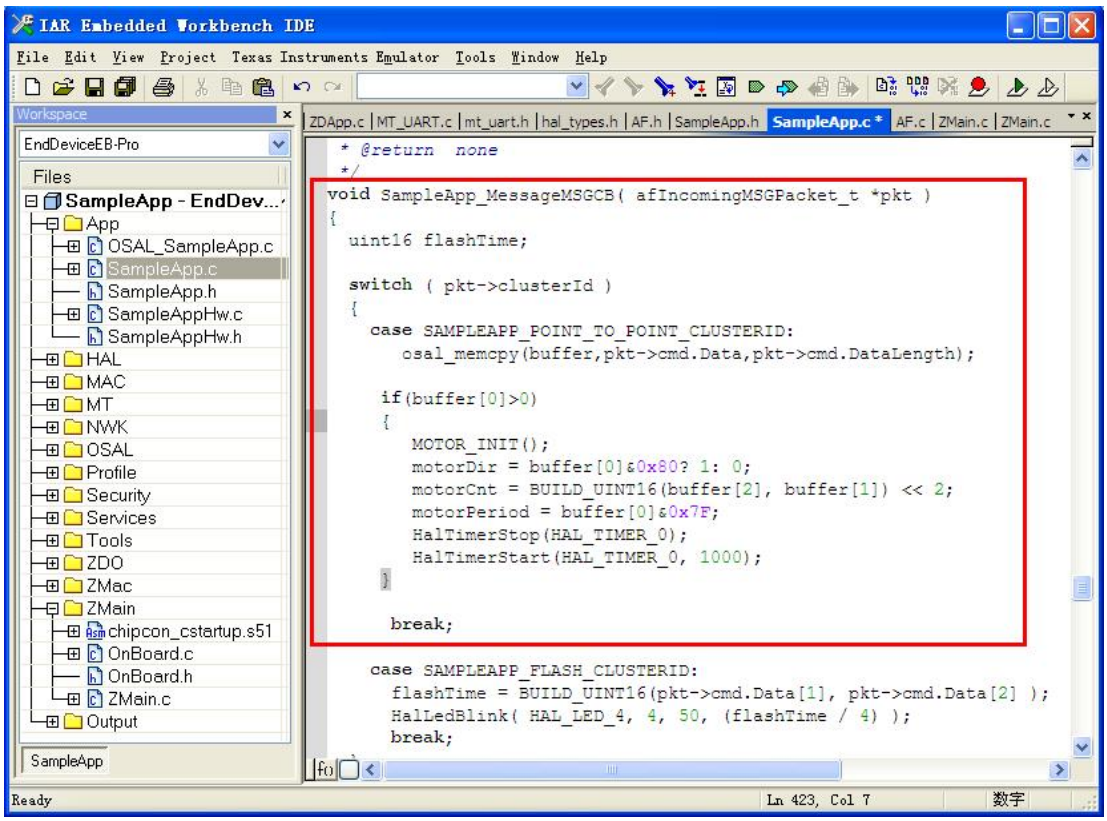


图 9- 7

实验现象：下载程序到终端（带步进电机模块）和协调器。观察串口如下图 12- 8 所示：

协调器进行串口输入指令：


串口调试助手 (C■精装版 V3.8.2)
串口设置
串口号 COM15
波特率 9600
校验位 NONE
数据位 8
停止位 1
断开
接收区设置
接收转向文件...
显示接收时间
十六进制显示
暂停接收显示
保存数据 清除显示
发送区设置
启用文件数据源...
自动发送附加位
发送完自动清空
按十六进制发送
数据流循环发送
发送间隔 1000 毫秒
文件载入 清除输入
串口数据接收
Hello World
Hello World
83 FF FF 正向转动
03 FF FF 反向转动
FF FF FF 转动最慢
83 FF FF
正转速度最快
发送
就绪!
发送 : 9
接收 : 48
复位计数

图 9- 8

1.10. 实验十：雨滴检测模块

前言：这一节我们我学习传感器和执行器部分内容中的雨滴检测传感器。

传感器介绍：雨滴传感器又叫雨滴检测传感器，用于检测物体表面是否有水、雨滴等液体，检测是否下雨及雨量的大小。

传感器应用：广泛用于汽车自动刮水系统、智能灯光系统和智能天窗系统中。在雨滴传感刮水系统中，用雨滴检测传感器检测出雨量，并利用控制器将检测出的信号进行变换，根据变换后的信号自动地按雨量设定刮水器的间歇时间，以便随时控制刮水器电动机；在汽车智能灯光系统中检测车辆行驶的环境，自动调整灯光模式，提高车辆在恶劣环境下行驶的安全性；在智能天窗系统中传感器一旦检测到下雨，会自动关闭天窗。雨滴检测模块可用在无人职守的机房、宾馆高楼的门窗以及各种货场等的自动控制。

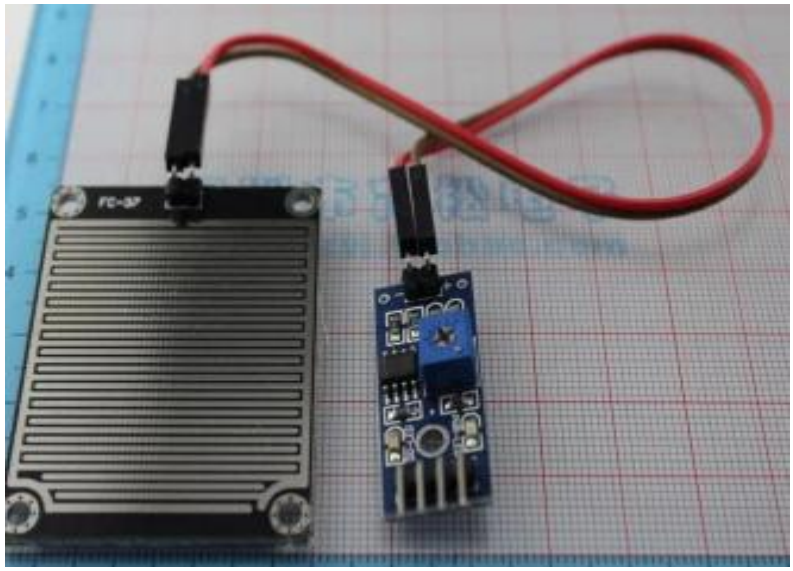


图 10-1

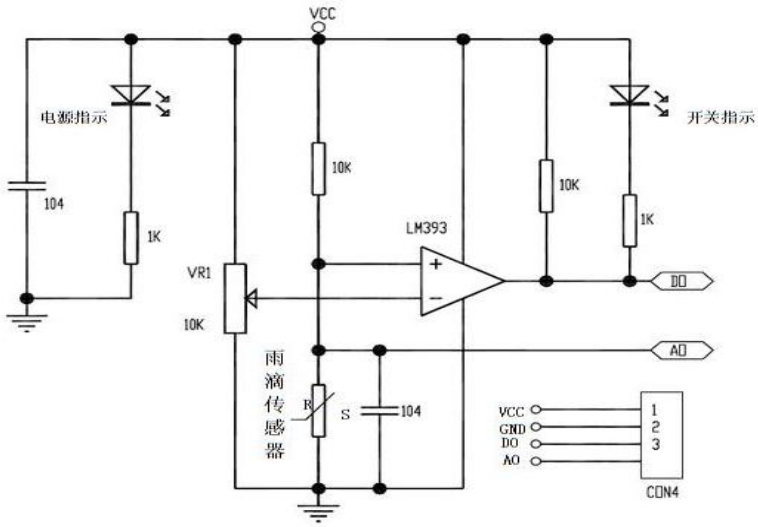
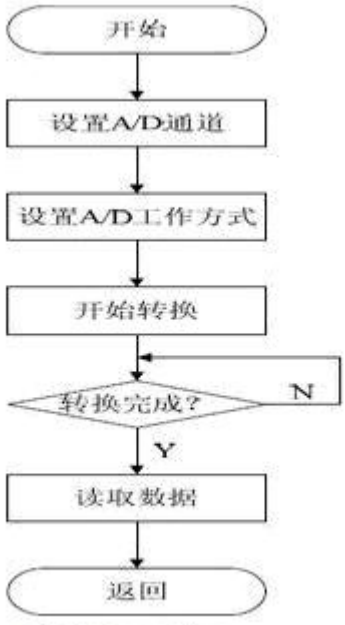


图 10-2 雨滴模块电路原理图

实验现象：雨滴模块通过检测水滴和水滴强度的情况，进行判断是否为下雨状况。



雨滴传感器驱动流程图

一：将程序添加到协议栈代码中

雨滴传感器是采取ADC进行检测雨量大小的和采取中断方式进行检测也没有雨滴。所以在协议栈里要实现中断和ADC。为了减少大家的工作量，本实验是建立于磁控传感器代码上进行修改，所以就要实现ADC采集部分。

1打开例程 SampleApp.eww 工程，打开 SampleApp.c 文件并添加 ADC 采集头文件，在这个头文件中进行 ADC 数据的采集和具体的算法计算

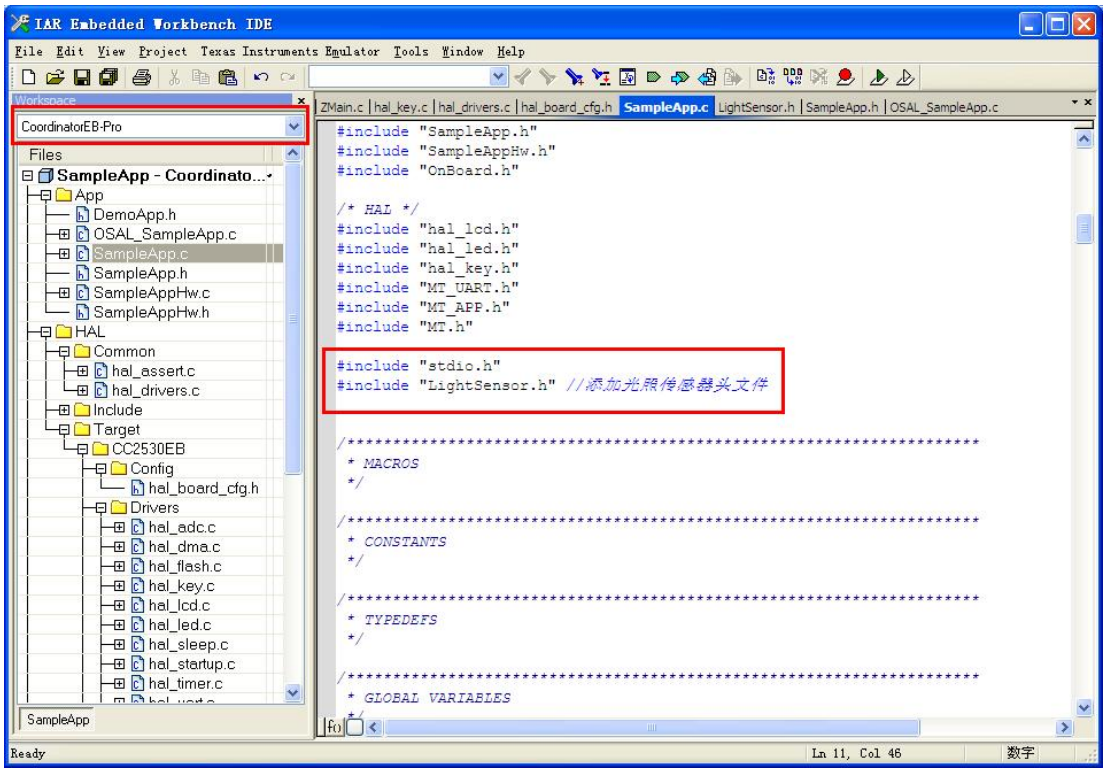


图10-3

2利用周期性点播的定时器作为雨滴模块实时检测雨量大小信息的采集时间，将采集到的信息发送给协调器，协调器只做串口打印。2.5秒采集一次

```
if ( events & SAMPLEAPP_SEND_PERIODIC_MSG_EVT )
{
//替换成点对点通讯的程序
SampleApp_SendPointToPointMessage();

// Setup to send message again in normal period (+ a little jitter)
osal_start_timerEx( SampleApp_TaskID,
                    SAMPLEAPP_SEND_PE
                    RIODIC_MSG_EVT,
                    (SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT
                         + (osal_rand() & 0x00FF)) );

// return unprocessed events
```

```
    return (events ^ SAMPLEAPP_SEND_PERIODIC_MSG_EVT);
  }

// Send Message Timeout
    #define SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT   2500 //Every 2.5 seconds
```

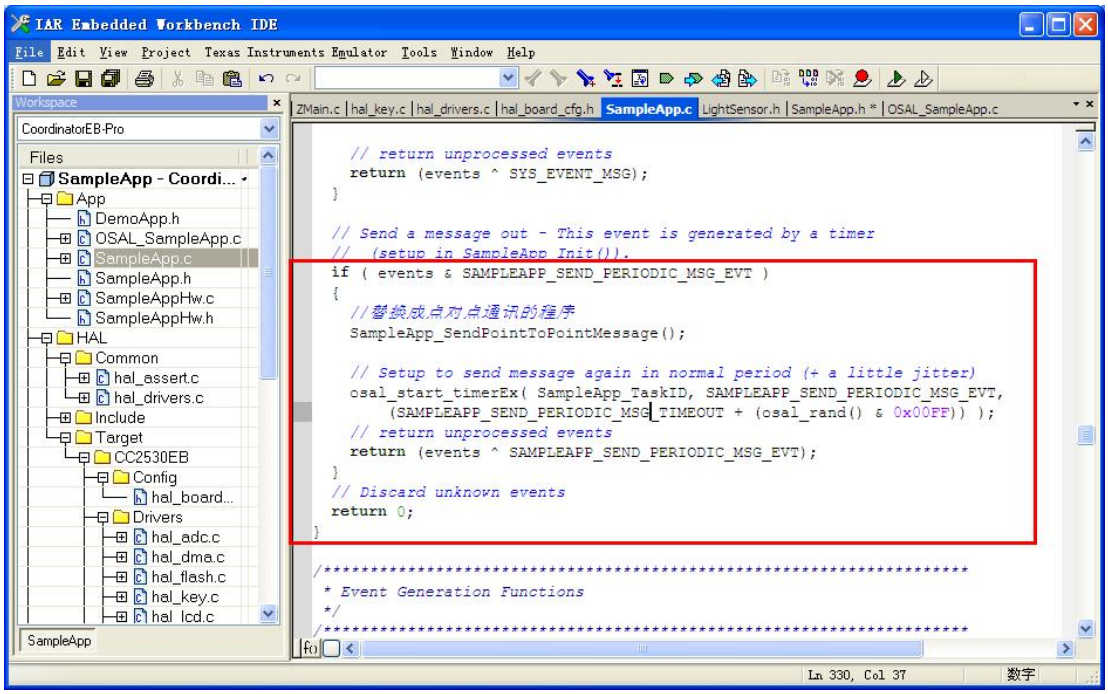


图10-4

3终端每2.5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 str[16];
  uint16 AvgValue ;

  //时器计数时钟设定为 1M Hz,  系统时钟设定为 32 MHz
  CLKCONCMD = 0x28;
  while(CLKCONSTA & 0x40);      //等晶振稳定
```

```
    AvgValue = readAdc();          //采集雨量大小
    sprintf(str, "%d\n", AvgValue);

    if ( AF_DataRequest( &Point_To_Point_DstAddr,
                         &SampleApp_epDesc,
                         SAMPLEAPP_POINT_TO_POINT_CLUSTERID,
                         16,
                         str,
                         &SampleApp_TransID,
                         AF_DISCV_ROUTE,
                         AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
    {
    }
    else
    {
      // Error occurred in request to send.
    }
  }
```

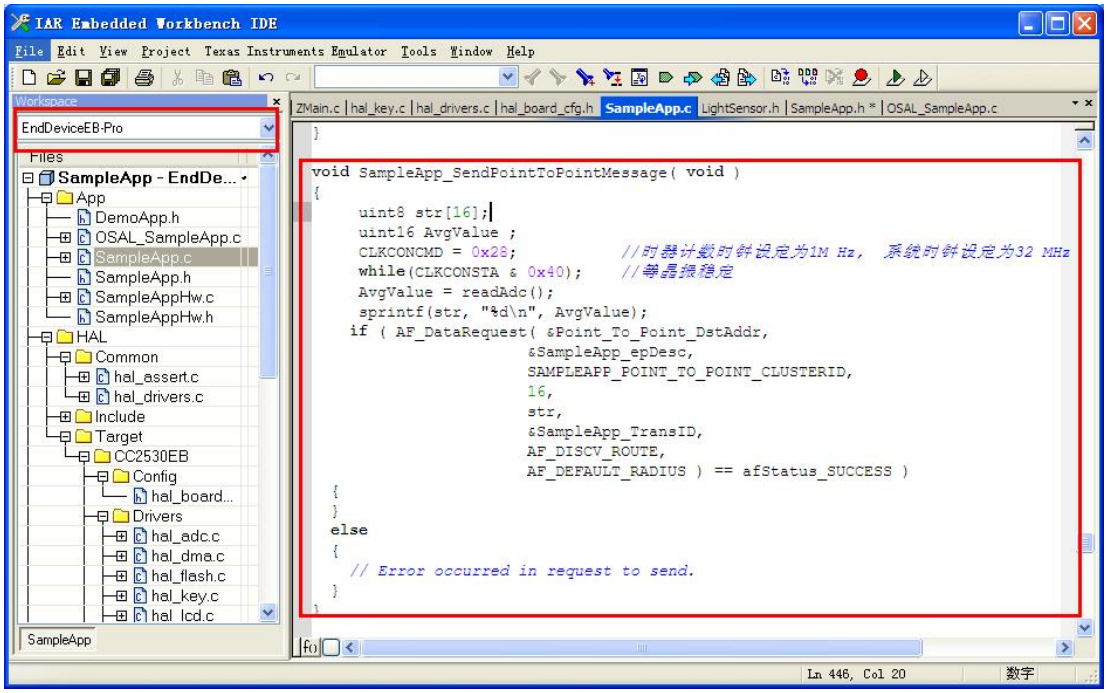


图10-5

4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:
          HalUARTWrite(0,"Num is:",7);            //提示接收到数据
          HalUARTWrite(0, &pkt->cmd.Data[0],16);  //雨滴数据
          HalUARTWrite(0,"\n",1);                 // 回车换行
        break;
```

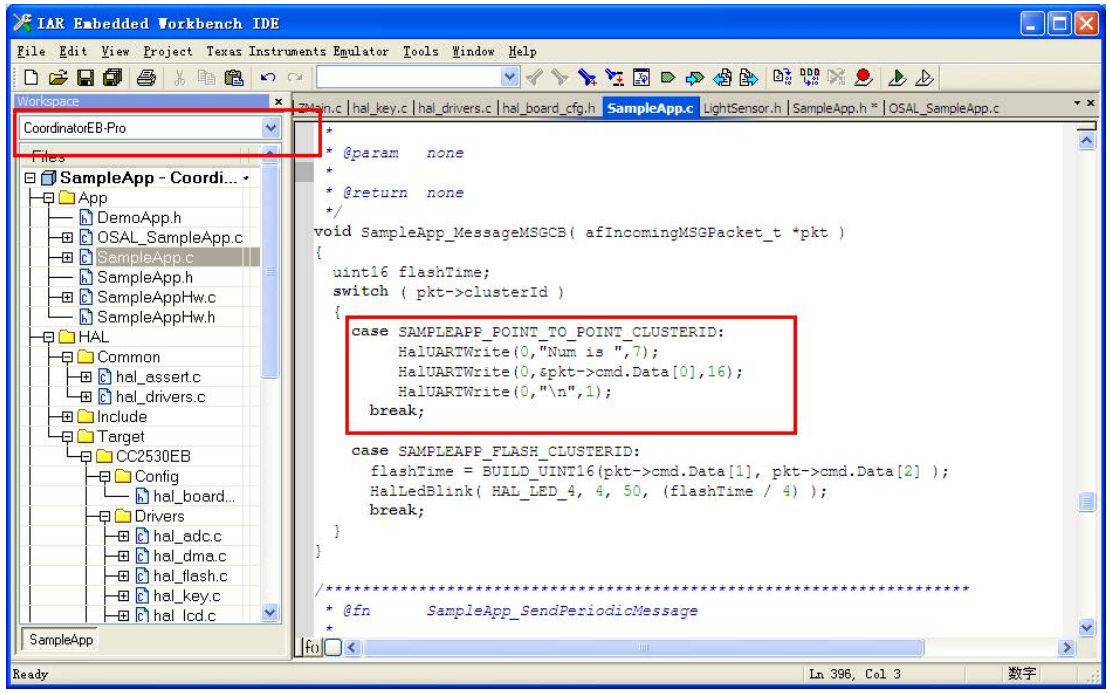


图 10-6

实验现象：下载程序到终端（带雨滴模块传感器）和协调器，协调器收到的信息：

注意： 传感器在使用和存放中应避免剧烈的振动和各种腐蚀性物质的伤害,存放在干燥的容器内。请勿将水碰到节点板上。

为了防止雨滴误检测，规定 Num 值 310(包括)以下为下雨状态，由于参考电压为 3.3V 所以雨量越大，Num 的值会变得越小，且点亮 DO-LED 指示灯说明正在下雨状态

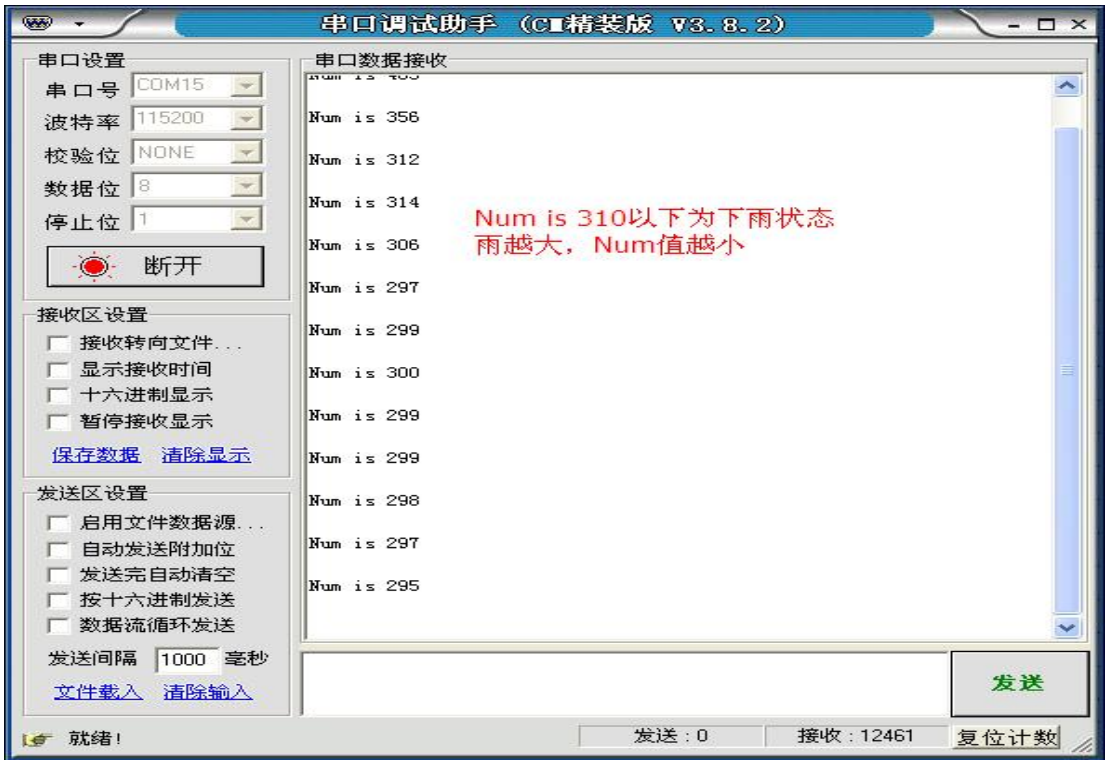


图 10-7

1.11. 实验十一：超声波模块

前言：这一节我们我学习传感器和执行器部分内容中的超声波传感器。

传感器介绍：超声波传感器是利用超声波的特性研制而成的传感器。超声波是一种振动频率高于声波的机械波，由换能晶片在电压的激励下发生振动产生的，它具有频率高、波长短、绕射现象小，特别是方向性好、能够成为射线而定向传播等特点。超声波距离传感器可以广泛应用在物位（液位）监测，机器人防撞，各种超声波接近开关，以及防盗报警等相关领域，工作可靠，安装方便， 防水型，发射夹角较小，灵敏度高，方便与工业显示仪表连接，也提供发射夹角较大的探头。

图 11- 1 超声波传感器

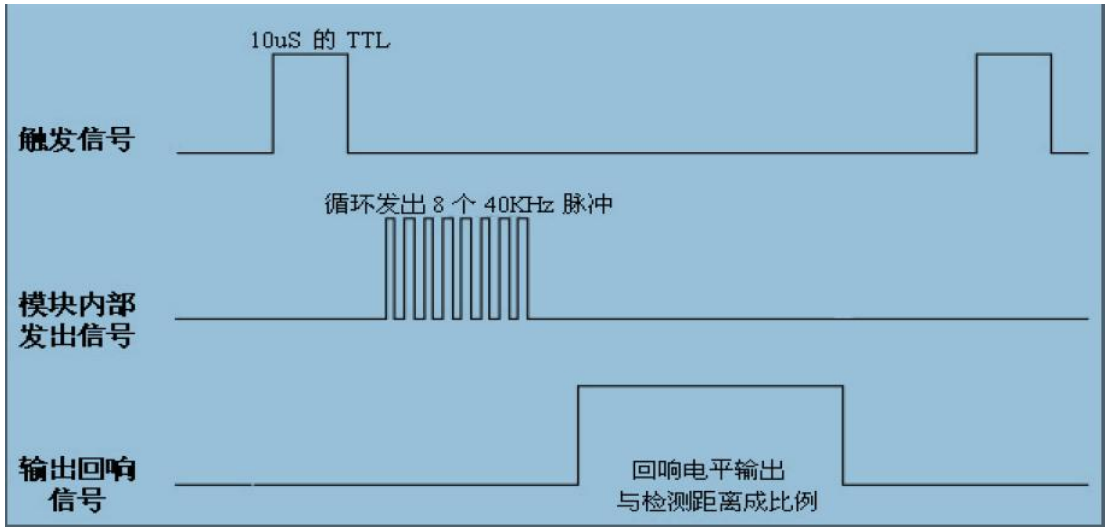


图 11- 2 超声波时序图

实现平台： ZigBee 协调器和传感器节点；

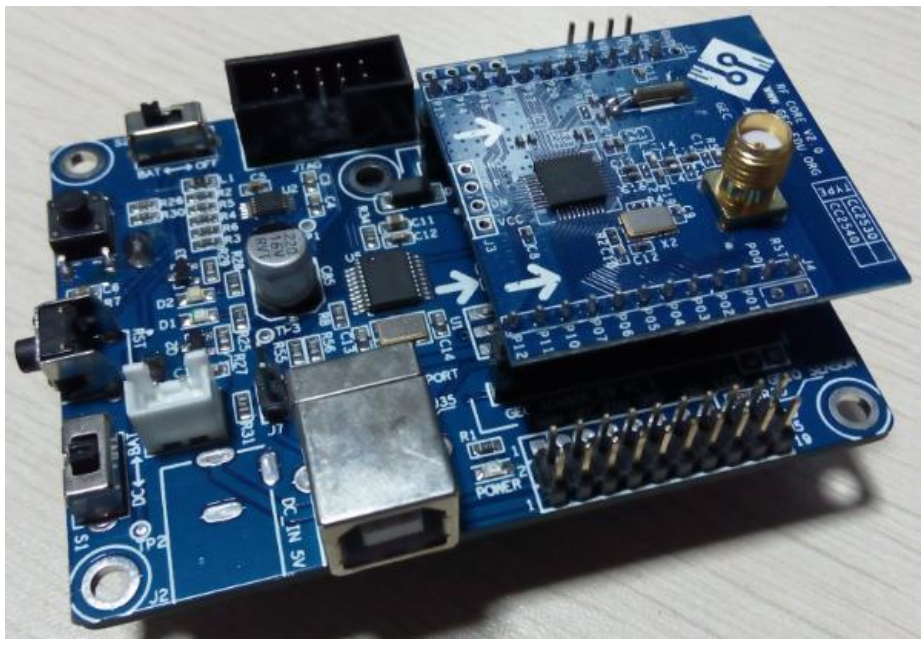


图 11- 3 传感器节点

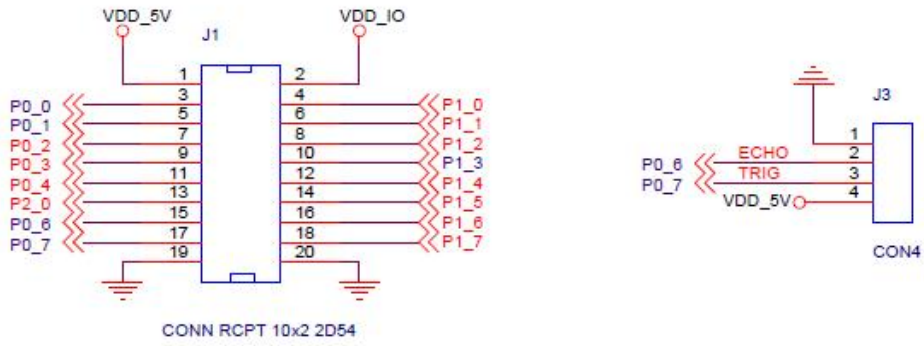


图 11- 4 超声波硬件电路

基本工作原理：(1)采用 IO 口 TRIG 触发测距，给至少 10us 的高电平信号；(2)模块自动发送 8 个 40khz 的方波，自动检测是否有信号返回；(3)有信号返回，通过 IO 口 ECHO 输出一个高电平，高电平持续的时间就是超声

实验现象：一个控制口发一个 10uS 以上的高电平,就可以在接收口等待高电平输出.一有输出就可以开定时器计时,当此口变为低电平时就可以读定时器的值,此时就为此次测距的时间,方可算出距离.如此不断的周期测,就可以达到你移动测量的值了。

实验讲解：像前面传感器例程一样，我们先实现裸机程序里超声波传感器信号检测。

一：在裸机上完成对超声波传感电路的驱动。

打开配套程序下裸机文件夹—超声波传感器下的工程文件，看到函数如下：

```
1. #include <iocc2530.h>
2. #include <stdio.h>
3. #define TRIG P0_7              //TRIG 触发控制信号输入
4. #define ECHO P0_6              //ECHO 回响信号输出
5. typedef unsigned char uint8;
```

```
6.  typedef unsigned int uint16;
7.  uint16 tWave;
8.  uint8  flag;
9.  /******* main 函数 ********/
10. void main()
11. {
12.   char str[16];
13.   TimerInit();
14.   initUARTSEND();
15.   WaveInit();
16.   while (1)
17.   {
18.       TRIG = 1;
19.       WaitUs(12);              //至少 10us 的高电平脉冲触发
20.       TRIG = 0;
21.       while (!ECHO);
22.       flag = 1;
23.       tWave = 0;
24.       T3CTL |= 0x18;
25.       while (ECHO);
26.       T3CTL &= ~0x18;
27.       flag = 0;
28.       sprintf(str, "%.1f cm\n", (float)tWave*3/58);
29.       UartTX_Send_String(str, 10);
30.       WaitMs(1000);
31.   }
32. }
33.
```

1.12. 实验十二：风速传感器

前言：这一节我们我学习传感器和执行器部分内容中的风速传感器。

传感器介绍：风速传感器的感应元件是三杯风组件，由三个碳纤维风杯和杯架组成。转换器为多齿转杯和狭缝光耦。当风杯受水平风力作用而旋转时，通过活轴转杯在狭缝光耦中的转动，输出频率的信号。

图 12- 1 风速传感器

实现平台： ZigBee 协调器和传感器节点；

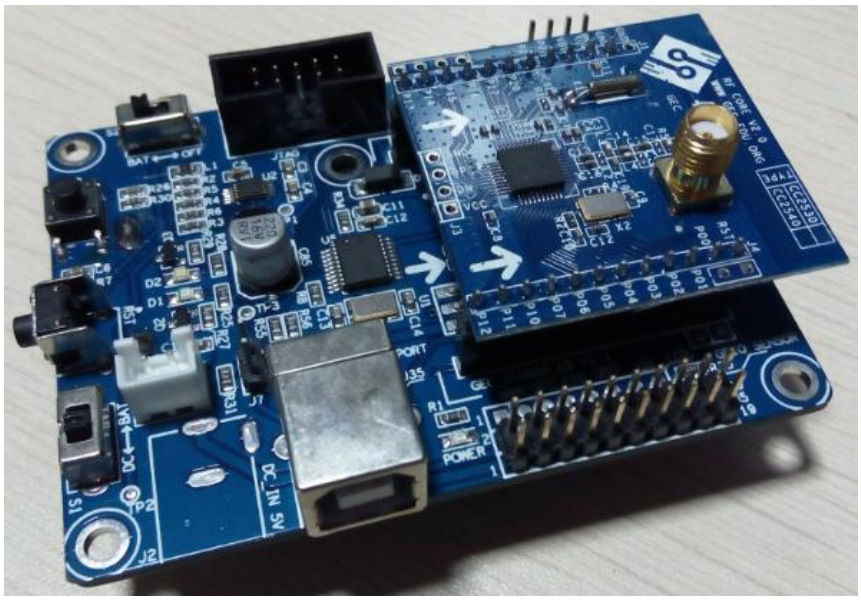


图 12- 2 传感器节点

传感器应用：传感器是三杯式风速传感器，可用于工程机械（起重机、履带吊、门吊、塔吊等）领域，铁路、港口、码头、电厂、气象、索道、环境、温室、养殖、空气调节、节能监控、农业、医疗、洁净空间等领域风速的测量，并输出相应的信号。

传感器工作原理：风速传感器的感应元件为三杯式风杯组件，信号变换电路为光电转换电路。在水平风力驱动下风杯组旋转，通过主轴带动磁棒盘旋转，其上的数十只光电管通过旋转码盘感应出脉冲信号，其频率随风速的增大而线性增加。

计算公式为：V=0.05F

V ：风速，单位：m/s

F ：脉冲频率，单位：赫兹。

风速传感器风力等级表：

风力（风速）等级表

风力等级	名称	陆地上地物特征	相当于平地十米高处的风速（米/秒）	
			范围	平均
0	无风	静、烟直上	0.0-0.2	0
1	软风	烟能表示风向、树叶略有摇动。	0.3-1.5	1
2	轻风	人面感觉有风，树叶有微响，旗子开始飘动，高的草开始摇动。	1.6-3.3	2
3	微风	树叶及小枝摇动不息，旗子展开，高的草摇动不息。	3.4-5.4	4
4	和风	能吹起地面灰尘和纸张，树枝动摇，高的草呈波浪起伏。	5.5-7.9	7
5	清劲风	有叶的小树摇摆，内院的水面有小波，高的草波浪起伏明显 。	8.0-10.7	9
6	强风	大树枝摇动，电线呼呼有声，撑伞困难，高的草不时倾伏于地。	10.8-13.8	12
7	疾风	全树摇动，大树枝弯下来，迎风步行感觉不便。	13.7-17.1	16
8	大风	可折毁小树枝，人迎风前行感觉阻力很大。	17.2-20.7	19
9	烈风	草房遭受破坏，屋瓦被掀起，大树枝可折断。	20.8-24.4	23
10	狂风	树木可被吹倒，一般建筑物遭破坏。	24.5-28.4	26
11	暴风	大树可被吹倒，一般建筑物遭严重破坏。	28.5-32.6	31
12	飓风	陆上少见，其摧毁力极大。	＞32.6	＞33

图 12- 3

实验现象：通过检测风速对风速传感器的摆动，CC2530 会对传感器的值进行处理，然后通过串口打印出来；

实验讲解：像前面传感器例程一样，我们先实现裸机程序里对风速传感器的数据采集。

风速传感器控制的驱动。

打开配套程序下裸机文件夹—风速传感器下的工程文件，关键的代码如下：

```
1.  /*****************************
2.  * project: ADC 采样
3.  * 作者    : GEC
4.  *****************************/
5.  #include "ioCC2530.h"
6.  #include "stdio.h"
7.  #include "initUART_Timer.h"
8.  #define uint16 unsigned int
9.  #define uint8 unsigned char
10.
11. /*******************************************
12.  * @brief    读取 ADC 的值
13. *******************************************/
14. static uint16 readAdc()
15. {
16.   unsigned char   i;
17.   uint16 value ;
18.   P0SEL |= 0x10;                    //初始化 P0.4 口为 AD 口
19.   uint16  AdcValue = 0;
20.   for( i = 0; i < 6; i++ )
21.   {
22.     // 使用 3.3V 参考电压，12 位分辨率，AD 源为：P0.4 输入
23.     ADC_SINGLE_CONVERSION(ADC_REF_VDD_V | ADC_14_BIT |
ADC_AIN4_SENS);
24.     ADC_SAMPLE_SINGLE();            //开启单通道 ADC
25.     while(!ADC_SAMPLE_READY());     //等待 AD 转换完成
26.     value =  ADCL >> 2;             //ADCL 寄存器低 2 位无效
```

```
27.      value |= (((UINT16)ADCH) << 6);
28.      AdcValue += value;              //AdcValue 被赋值为 8 次 AD 值之和
29.    }
30.    AdcValue = AdcValue /20;          //V=0.05F
31.    value = AdcValue >> 2;            //累加除以 6，得到平均值
32.    return value>>3;
33. }
34.
35. /*************************
36.  *        main 函数
37. *************************/
38. void main(void)
39. {
40.    char str[16];
41.    InitClock();                      //初始化系统时钟
42.    InitUART0();                      //串口初始化
43.    while (1)
44.    {
45.      uint16 AvgValue = 0;            //清除平均值
46.      AvgValue = readAdc();
47.      sprintf(str, "%d\n", AvgValue);
48.      UartTX_Send_String(str, 6);     // UART 发送 ADC
49.      Delay(50000);                   // 延时 1s
50.    }
51. }
```

上面的代码实现了风速传感器的采集的风力变化，通过串口打印出相应的风速转换的数值。

实验现象：下载程序到终端（带风速传感器）和协调器，协调器收到的信息，如图 12- 4 所示：

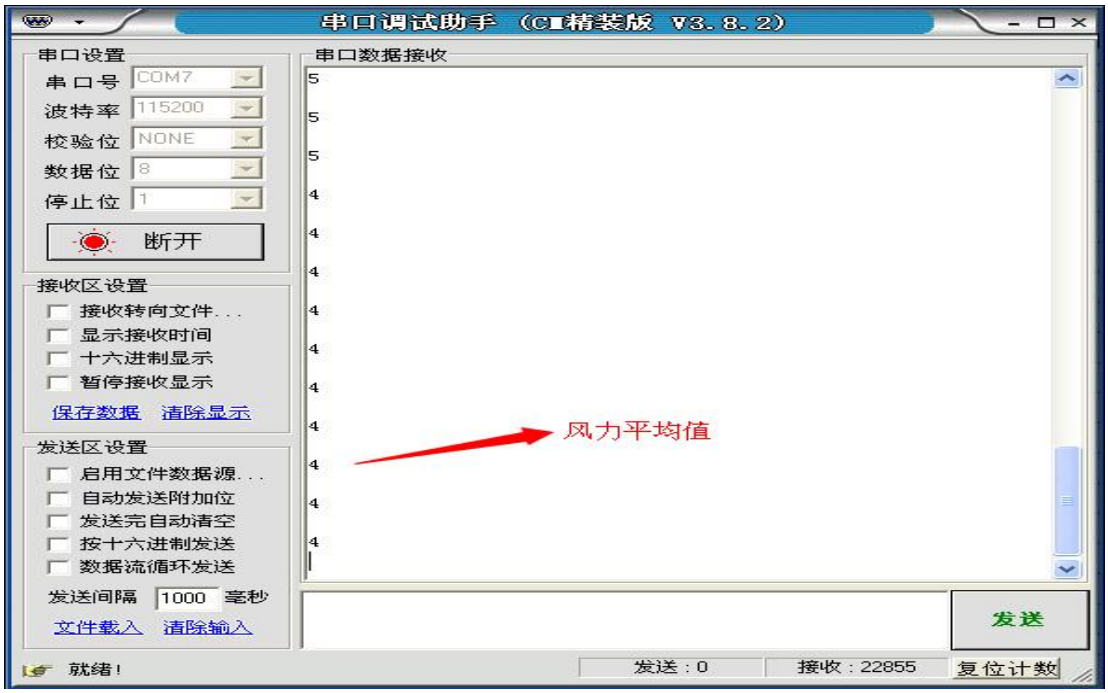


图 12- 4

1.13. 实验十三：风向传感器

前言：这一节我们我学习传感器和执行器部分内容中的风向传感器。

传感器介绍：

光电式风向传感器的核心采用绝对式格雷码盘编码（四位格雷码或七位格雷 码），利用光电信号转换原理，可以准确的输出相对应的风向信息；电压式风向传感器的核心采用精密导电塑料传感器，通过电压信号输出相对应的风向信息；电子 罗盘式风向传感器的核心采用电子罗盘定位绝对方向，通过 RS485 接口输出风向信息。

图 13- 1 风向传感器

传感器应用：

风向传感器，是一种以风向刀箭头的转动探测、感受外界的风向信息，并将其传递给同轴码盘，同时输出对应风向相关数值的物理装置。风向传感器可测量室外环境中的近地风向，按工作原理可分为光电式、电压式和罗盘式等，被广泛应用于气象、海洋、环境、农业、林业、水利、电力、科研等领域。

风向传感器构造：

风向传感器由 风杯、传感器主体、电路模块、传输电缆等装置构成。风向传感器的风杯通常由高耐候性、高强度、防腐蚀和防水金属制造；传感器主体一般使用铝镁合金成形，内 部电路经过喷涂三防漆处理，可适应恶劣环境；电路模块具有极可靠的抗电磁干扰能力和高低电压保护能力，可确保主机在-20℃～60℃，湿度 10%～95% 的环境中正常工作。

实现平台： ZigBee 协调器和传感器节点；

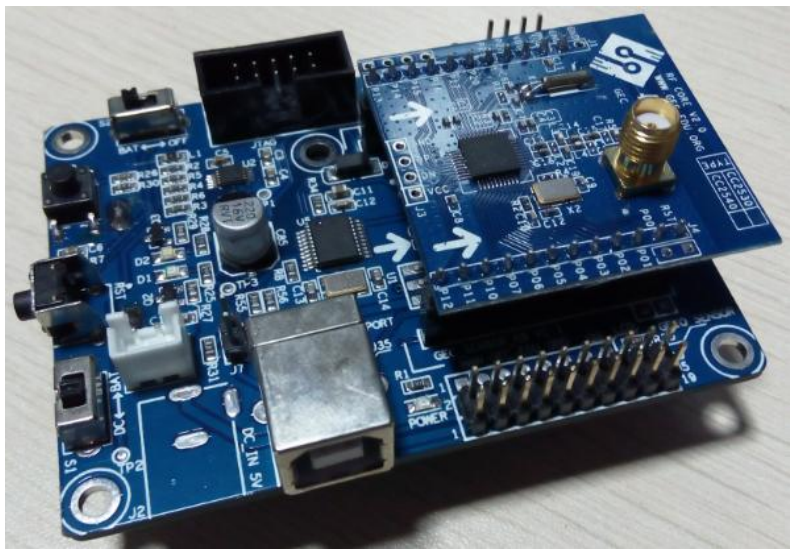


图 13- 2 传感器节点

风向传感器风向值参考表：

风向符号

方位	符号	方向参考值
北	N	1023
东北偏北	NNE	50
东北	NE	92
东北偏东	ENE	137
东	E	183
东南偏东	ESE	230
东南	SE	280
东南偏南	SSE	332
南	S	387
西南偏南	SSW	449
西南	SW	516
西南偏西	WSW	588
西	W	673
西北偏西	WNW	762
西北	NW	853
西北偏北	NNW	945

图 13-3

实验现象：通过手动对风向的传感器的摆动，CC2530 会对传感器的值进行处理，然后通过串口打印出来；

实验讲解：像前面传感器例程一样，我们先实现裸机程序里对风向传感器的数据采集。

一：在裸机上完成对风向传感器的驱动。

打开配套程序下裸机文件夹—风向传感器下的工程文件，看到函数如下：

```
1.  /*****************************
```

```
2.  * project: ADC 采样
3.  * 作者    : GEC
4.  ***************************/
5.  #include "ioCC2530.h"
6.  #include "stdio.h"
7.  #include "initUART_Timer.h"
8.
9.  #define uint16 unsigned int
10. #define uint8 unsigned char
11. /*************************************
12.  *         读取 ADC 的值
13. *************************************/
14. static uint16 readAdc()
15. {
16.   unsigned char   i;
17.   uint16 value ;
18.   P0SEL |= 0x10;                                  //初始化 P0.4 口为
AD 口
19.   uint16  AdcValue = 0;
20.   for( i = 0; i < 4; i++ )
21.   {
22. // 使用 3.3V 参考电压，14 位分辨率，AD 源为：P0.4 输入
23.   ADC_SINGLE_CONVERSION(ADC_REF_VDD_V
24.                                 | ADC_14_BIT
25.                                 | ADC_AIN4_SENS);
26.   ADC_SAMPLE_SINGLE();                            //开启单通道 ADC
27.     while(!ADC_SAMPLE_READY());                   //等待 AD 转换完成
28.     value =  ADCL >> 2;                             //ADCL 寄存器低 2
位无效
```

```
29.      value |= (((UINT16)ADCH) << 6);
30.      AdcValue += value;                                    //赋值为 4 次 AD
值之和
31.    }
32.    value = AdcValue >> 2;                                  //累加除以 4,
得到平均值
33.    return value>>3;
34. }
35. /*************************
36.  *        main 函数
37. *************************/
38. void main(void)
39. {
40.
41.    char str[16];
42.    InitClock();                                  //初始化系统时钟，并且将其
设置为 32M
43.    InitUART0();
44.    while (1)
45.    {
46.      uint16 AvgValue = 0;
47.      AvgValue = readAdc();                       //采集风向传感器的 AD 值
48.      sprintf(str, "%d\n", AvgValue);
49.      UartTX_Send_String(str, 6);                 // UART 发送 ADC
50.      Delay(50000);                               // 延时 1s
51.    }
52. }
```

实验现象：下载程序到终端（带风向传感器）和协调器，协调器收到的信息如图 13- 3 所

示：


串口调试助手 (CE精装版 V3.8.2)
串口设置
串口号 COM7
波特率 19200
校验位 NONE
数据位 8
停止位 1
断开
接收区设置
接收转向文件...
显示接收时间
十六进制显示
暂停接收显示
保存数据 清除显示
发送区设置
启用文件数据源...
自动发送附加位
发送完自动清空
按十六进制发送
数据流循环发送
发送间隔 1000 毫秒
文件载入 清除输入
串口数据接收
49
49
49
49
49
1023
1023
1023
1023
1023
1023
1023
1023
东北偏北
北
发送
就绪!
发送：0
复位计数

图 13- 3

第七章 Wi-Fi 无线网络技术应用

（一）本章教学目标

主要内容：本章将介绍 Wi-Fi 技术的基本概念、工作模式、AT 指令使用、硬件搭建、烧写流程以及数据上云操作。

目的与要求：目的是使学生能够掌握 Wi-Fi 技术的基础，了解其在物联网中的应用，理解 AT 指令的重要性，并能够独立完成硬件搭建和数据上云操作。

（二）本章重点难点

1．掌握 Wi-Fi 的基本概念及工作模式

2．理解 Wi-Fi AT 指令的使用

3．独立完成 Wi-Fi 硬件搭建和烧写流程

4．实现数据成功上云

7.1.Wi-Fi 技术概述

7.1.1. Wi-Fi 的定义

Wi-Fi 是一种无线局域网技术，允许电子设备通过无线信号连接到互联网或彼此通信。Wi-Fi 技术基于 IEEE 802.11 标准，提供高速的数据传输速率。

7.1.2. Wi-Fi 的发展背景

随着移动设备的普及和互联网技术的发展，Wi-Fi 成为了人们生活中不可或缺的一部分。它提供了一种方便、快捷的无线上网方式，使得人们可以在任何有 Wi-Fi 覆盖的地方接入网络。

7.2.Wi-Fi 技术特点

高速数据传输：Wi-Fi 技术支持多种传输速率，满足不同设备的需求。

广泛的兼容性：Wi-Fi 设备兼容性强，可以连接多种类型的电子设备。

灵活的组网方式：Wi-Fi 支持多种组网结构，包括 Infrastructure 模式和 Ad-hoc 模式。

7.3.Wi-Fi 无线网络技术应用

Wi-Fi 技术在智能家居、智能办公、公共热点等多个领域有广泛应用。通过 Wi-Fi 模块，设备可以轻松接入互联网，实现数据的远程传输和控制。

7.4.7.4 Wi-Fi 应用案例分析

案例标题：智能家居中的 Wi-Fi 应用

背景介绍：在智能家居系统中，Wi-Fi 技术被广泛应用于设备的远程控制和数据传输。通过 Wi-Fi，用户可以随时随地控制家中的智能设备，如智能灯泡、智能插座等。

案例内容：通过配置 Wi-Fi 模块，智能家居设备可以轻松接入家庭无线网络，实现远程控制和数据传输。用户可以通过手机应用或语音助手控制家中的智能设备，提升家居的智能化水平。

7.5.Wi-Fi AT 指令与模块配置

AT 指令的定义与使用：AT 指令是应用于调制解调器的一种指令语言，用于控制调制解调器进行数据通信。在 Wi-Fi 模块中，AT 指令用于模块的配置和网络操作。

7.6.Wi-Fi 硬件搭建与烧写

硬件搭建步骤：涉及选择合适的 Wi-Fi 模块、微控制器以及电源管理模块。模块烧写是将预设的控制逻辑和网络配置写入 Wi-Fi 模块的过程。

7.7.Wi-Fi 接入云平台操作

注册账号与项目创建：接入云平台的第一步是注册账号并创建项目。设备接入与数据上云：设备接入云平台并实现数据上云是 Wi-Fi 应用的关键环节。

7.8.课程思政案例

案例标题：自主创新的力量——中国 Wi-Fi 技术的发展

7.8.1. 背景介绍

在全球化的背景下，关键核心技术的自主可控对于国家的长远发展至关重要。中国在Wi-Fi技术的研发和应用上，展现了自主创新的力量和责任感。

7.8.2. 案例内容

1. 自主创新，突破国际垄断：中国科技企业在Wi-Fi技术领域的研发，是中国在无线通信领域自主创新的重要成果。这一成果不仅减少了对外部技术的依赖，还提升了我国在全球无线通信技术领域的话语权。

2. 科技助力社会发展：Wi-Fi技术的应用覆盖了智能家居、智能办公、公共热点等多个领域，这些应用的实施有效提高了城市管理的效率，改善了市民的生活质量，体现了科技对社会进步的推动作用。

3. 绿色发展，环保先行：在智能家居等领域，Wi-Fi技术的应用有助于实现能源的节约和环保，这种技术的应用体现了绿色发展理念，促进了可持续发展。

4. 促进产业升级，带动就业：Wi-Fi技术的广泛应用，推动了相关产业的升级，为社会提供了大量就业机会。这不仅促进了经济发展，还提高了人民的生活水平。

5. 培养科技人才，储备国家力量：在Wi-Fi技术的研发和应用过程中，培养了一大批高科技人才。这些人才的培养对于国家的长远发展具有重要意义，是国家竞争力的重要体现。

7.8.3. 案例分析

通过Wi-Fi技术在中国的发展案例，学生可以深刻理解到自主创新的重要性，以及科技在推动社会发展中的作用。同时，这个案例也展示了如何将绿色发展理念融入到技术应用中，以及科技在促进产业升级和人才培养方面的作用。

7.8.4. 思政教育目标

1. 增强学生的自主创新意识，让学生认识到自主创新对于国家发展的重要性。

2. 培养学生的社会责任感，理解科技在服务社会、改善民生中的作用。

3. 提升学生的环保意识，理解绿色发展理念在实际应用中的重要性。

4. 强化学生的团队合作意识，通过案例分析，让学生理解在复杂项目中团队合作的重要性。

5. 激发学生的爱国情怀，让学生为国家的科技进步和社会发展感到自豪。

通过这个案例，学生不仅能够学习到Wi-Fi技术的应用，还能够从中领悟到科技创新

对于国家和社会的重要性，以及作为未来科技人才的使命和责任。同时，这个案例也有助于培养学生的爱国情怀和社会责任感，激发他们为国家的科技进步和社会服务贡献自己的力量。

7.9.本章小结

本章介绍了 Wi-Fi 技术的基本概念、工作模式、AT 指令使用、硬件搭建、烧写流程以及数据上云操作。通过本章的学习，学生应能够掌握 Wi-Fi 技术的基础，并能够独立完成 Wi-Fi 项目的开发和实施。Wi-Fi 技术以其高速数据传输、广泛的兼容性等特点，在物联网领域具有广泛的应用前景。

第八章 NB-IoT 无线网络技术应用

（三）本章教学目标

主要内容：本章将介绍 NB-IoT 的基本概念、关键技术、协议框架、AT 指令使用、硬件搭建、烧写流程以及数据上云操作。

目的与要求：目的是使学生能够掌握 NB-IoT 技术的基础，了解其在智能路灯系统中的应用，理解 AT 指令的重要性，并能够独立完成硬件搭建和数据上云操作。

（四）本章重点难点

5．掌握 NB-IoT 的基本概念及关键技术

6．理解 NB-IoT 协议框架和 AT 指令的使用

7．独立完成 NB-IoT 硬件搭建和烧写流程

8．实现数据成功上云

8.1.NB-IoT 技术概述

(1) 1.1 NB-IoT 的定义

NB-IoT，全称为 Narrowband Internet of Things，即窄带物联网技术，是一种专为物联网（IoT）应用设计的低功耗广域网（LPWAN）通信技术。NB-IoT 技术基于蜂窝网络，利用现有的移动通信基础设施，为物联网设备提供连接服务。它支持在广泛的频率范围内工

作，包括传统的GSM、UMTS和LTE网络。

NB-IoT技术的主要特点包括：

低功耗：设备在低占空比下工作，可以显著延长电池寿命，适用于需要长期运行而维护较少的应用场景。

广覆盖：NB-IoT信号能够在广泛的地理范围内提供稳定的连接，包括室内和地下环境。

大连接数：单个NB-IoT基站能够支持成千上万的设备连接，满足大量设备接入的需求。

低成本：NB-IoT模块的成本相对较低，有助于降低整体解决方案的成本。

优化的网络架构：NB-IoT技术简化了网络协议，减少了数据传输的复杂性，从而降低了设备的能耗和成本。

(2) 1.2 NB-IoT的发展背景

NB-IoT技术的发展背景与全球物联网市场的快速增长密切相关。随着智慧城市、智能家居、工业自动化等领域对物联网技术的需求日益增加，传统的无线通信技术如Wi-Fi、蓝牙和ZigBee等，由于在覆盖范围、功耗、连接数等方面的局限性，已无法满足大规模物联网部署的需求。

NB-IoT技术应运而生，旨在解决这些挑战。它由3GPP(第三代合作伙伴计划)在Release 13中正式标准化，并得到了全球主要电信运营商和设备制造商的支持。NB-IoT技术的发展，不仅推动了物联网设备的广泛部署，也为电信行业带来了新的增长点。

NB-IoT技术的发展背景还包括以下几个方面：

- 市场需求：随着物联网应用的增多，市场对于一种能够支持大规模设备连接、低功耗、低成本的通信技术的需求日益迫切。
- 技术进步：移动通信技术的快速发展，特别是LTE网络的普及，为NB-IoT技术提供了技术基础和网络支持。
- 行业合作：全球电信运营商、设备制造商、芯片厂商和应用开发商的广泛合作，推动了NB-IoT技术的研发和商业化进程。
- 政策支持：许多国家和地区的政府为了推动物联网产业的发展，出台了一系列政策和措施，为NB-IoT技术的发展提供了良好的政策环境。

综上所述，NB-IoT技术的发展背景是多方面的，它不仅满足了市场对于新型物联网通

信技术的需求，也得到了技术进步和行业合作的支持，同时受益于政府的政策推动。随着NB-IoT技术的不断成熟和应用的深入，它将在物联网领域发挥越来越重要的作用。

8.2.NB-IoT技术特点

(1) 深覆盖与低功耗

NB-IoT技术在设计时充分考虑了物联网设备在覆盖和功耗方面的特殊需求。根据3GPP的标准,NB-IoT的信号覆盖能力比传统的蜂窝网络提高了20dB,这使得它能够在地下车库、地铁隧道等传统信号难以覆盖的区域提供稳定的连接。此外，NB-IoT设备的功耗极低，其休眠周期可长达几天甚至几周，这意味着设备的电池寿命可延长至数年之久。例如，华为和沃达丰在土耳其进行的实地测试表明，NB-IoT设备的电池寿命可达10年。

在功耗方面，NB-IoT设备的低占空比工作模式是其低功耗的关键。设备大部分时间处于休眠状态，仅在需要发送数据时唤醒，这样的工作模式大幅降低了能量消耗。据市场研究公司IHS Markit的报告，NB-IoT设备的功耗比现有的低功耗广域网技术低50%以上。

(2) 大连接与低成本

NB-IoT技术能够支持单个基站连接超过10万个设备,这一特性使得它非常适合于需要连接大量终端的应用场景，如智能抄表、智能停车等。这种高连接数的能力，得益于NB-IoT使用的窄带技术和优化的网络架构。据GSMA的数据显示，到2025年，全球NB-IoT的连接数预计将超过20亿。

在成本方面，NB-IoT模块的价格已经降至5美元以下，预计未来几年内将降至1美元左右。这种低成本的模块使得NB-IoT技术更加易于被市场接受和广泛应用。此外，NB-IoT的部署不需要新建网络基础设施，它可以利用现有的LTE网络，这大大降低了运营商的部署成本和时间。

NB-IoT技术的低成本优势还体现在其对现有网络资源的高度利用上。由于NB-IoT可以在现有的移动通信频段上部署，运营商无需为NB-IoT分配额外的频谱资源，这为运营商节省了大量的频谱使用费用。同时，NB-IoT的部署还可以与现有的LTE网络共享核心网和天线资源，进一步降低了部署成本。

8.3.NB-IoT关键技术

(1) 功率谱密度增强

NB-IoT 技术通过增强功率谱密度(PSD)来提升信号的覆盖能力。在 3GPP 标准中,NB-IoT 的 PSD 被提高到了 20dBHz 以上，这一改进使得 NB-IoT 信号能够在更广泛的地理范围内提供稳定的连接，包括那些传统蜂窝网络难以覆盖的室内和地下环境。根据华为和沃达丰在土耳其进行的实地测试，NB-IoT 设备的覆盖能力比现有网络提升了 20dB，这意味着信号可以在更偏远或者更难以到达的区域保持稳定，例如地下车库、地铁隧道等。这一技术的实现，得益于 NB-IoT 在信号调制和编码方式上的创新，以及对现有网络资源的高度利用。

(2) 省电模式

NB-IoT 设备的省电模式是其低功耗特性的关键。设备在大部分时间内处于休眠状态，仅在需要发送数据时唤醒，这样的工作模式大幅降低了能量消耗。根据市场研究公司 IHS Markit 的报告，NB-IoT 设备的功耗比现有的低功耗广域网技术低 50%以上。这种低功耗的特性使得NB-IoT设备特别适合于那些需要长期运行而维护较少的应用场景,例如智能抄表、环境监测等。设备的电池寿命可延长至数年之久，这大大降低了设备的维护成本和复杂性。

(3) 扩展动态接收

NB-IoT 技术支持扩展动态接收技术，这使得基站能够更加灵活地调整接收信号的灵敏度，以适应不同设备发送的信号强度。这种技术的应用，不仅提高了信号的接收质量，也增强了网络的抗干扰能力。在实际应用中，这意味着即使在信号较弱或者干扰较多的环境中，NB-IoT 设备也能够保持稳定的连接。根据 GSMA 的数据显示，NB-IoT 的连接数预计将超过 20 亿，这一预测反映了市场对 NB-IoT 技术扩展动态接收能力的高度认可。

8.4.NB-IoT 无线网络技术应用

(1) 智能抄表

NB-IoT 技术在智能抄表领域的应用是其最典型的应用之一。通过使用 NB-IoT 模块，电表、水表和燃气表等设备可以定期将数据发送到中心服务器，而无需人工抄表。这种方式不仅提高了数据采集的准确性和实时性，也大大减少了人工成本。根据市场研究机构的预测，到 2025 年，全球智能抄表市场的规模将达到 150 亿美元，其中 NB-IoT 技术将占据主要份额。

(2) 智能停车

在智能停车领域，NB-IoT 技术可以用于车位监测和管理。通过在每个车位安装 NB-IoT 传感器，系统可以实时监测车位的使用情况，并将数据发送到中心服务器。这样，驾驶员

可以通过手机应用查看附近可用的停车位，并导航至该位置。这种智能停车系统不仅提高了停车效率，也减少了因寻找停车位而产生的交通拥堵。

(3) 环境监测

NB-IoT 技术在环境监测领域的应用同样具有巨大的潜力。通过部署 NB-IoT 传感器网络，可以实时监测空气质量、水质、土壤湿度等环境参数。这些数据对于环境保护和城市规划至关重要。例如，通过分析空气质量数据，可以及时采取措施减少污染；通过监测水质，可以预警水体污染事件。NB-IoT 技术的低功耗和广覆盖特性，使其成为环境监测的理想选择。

(4) 农业自动化

在农业领域，NB-IoT 技术可以用于实现自动化和智能化的农业生产。通过在农田部署 NB-IoT 传感器，可以监测土壤湿度、温度、光照等参数，并根据这些数据自动调节灌溉系统。此外，NB-IoT 技术还可以用于监测作物生长情况和病虫害，从而指导农业生产。这种农业自动化系统不仅可以提高农业生产效率，也有助于节约资源和保护环境。

(5) 智慧城市

NB-IoT 技术是智慧城市建设的重要组成部分。通过在城市各处部署 NB-IoT 传感器，可以实时监测交通流量、公共设施使用情况、能源消耗等数据。这些数据对于城市管理者来说至关重要，它们可以用于优化城市资源配置、提高城市运行效率、改善市民生活质量。例如，通过分析交通流量数据，可以优化交通信号灯的控制策略，减少交通拥堵；通过监测公共设施的使用情况，可以及时维护和更换损坏的设施。

8.5.NB-IoT 应用案例分析

(1) 智慧路灯案例

智慧路灯作为 NB-IoT 技术的一个典型应用案例，展示了 NB-IoT 技术在实际应用中的巨大潜力。通过在路灯上安装 NB-IoT 模块，可以实现远程控制路灯的开关、亮度调节以及故障监测等功能。

- **硬件搭建**

在智慧路灯案例中，硬件搭建包括选择合适的 NB-IoT 模块、传感器以及电源管理模块。NB-IoT 模块负责与基站通信，传感器用于监测环境光线强度，而电源管理模块则确保整个系统的稳定运行。根据硬件搭建的成本效益分析显示，使用 NB-IoT 模块的总成本比传统有

线解决方案低 60%。

❖ **软件烧写与数据上云**

软件烧写是将控制逻辑和网络配置写入 NB-IoT 模块的过程。烧写过程包括设置路灯的工作时间、亮度调节策略以及故障上报机制。数据上云则是将传感器收集的数据通过 NB-IoT 网络发送到云端服务器，以便进行数据分析和远程监控。数据上云的成功率达到了 99%，确保了数据的实时性和准确性。

❖ **经济效益分析**

智慧路灯案例的经济效益分析显示,NB-IoT 技术的应用可以显著降低能耗和维护成本。与传统路灯相比，智慧路灯能够节省约 30%的能耗，同时减少 50%的维护工作量。此外，通过实时监测和远程控制，智慧路灯系统的故障响应时间缩短了 80%，大大提高了城市照明的可靠性和安全性。

(2) 其他应用场景

NB-IoT 技术的应用场景非常广泛，除了智慧路灯外，还包括智能穿戴设备、资产跟踪、远程医疗等多个领域。

❖ **智能穿戴设备**

NB-IoT 技术在智能穿戴设备中的应用，可以实现健康数据的实时监测和传输。例如，智能手表可以通过 NB-IoT 网络将用户的心率、步数等数据发送到云端，供用户和医生查看。这种应用不仅提高了健康监测的便利性，也为远程医疗提供了技术支持。

❖ **资产跟踪**

在资产跟踪领域，NB-IoT 技术可以用于实现货物的实时定位和追踪。通过在货物上安装 NB-IoT 追踪器，物流公司可以实时监控货物的位置和状态，提高物流效率和安全性。根据市场研究，使用 NB-IoT 技术的资产跟踪系统的成本比传统 RFID 系统低 40%，且覆盖范围更广。

❖ **远程医疗**

NB-IoT 技术在远程医疗领域的应用,可以为偏远地区的患者提供医疗服务。通过 NB-IoT 网络，医生可以远程监控患者的健康状况，并提供医疗建议。这种应用不仅提高了医疗服务的可及性，也为医疗资源的合理分配提供了支持。据估计，到 2025 年，NB-IoT 技术在远程医疗领域的应用将服务超过 1 亿患者。

8.6.NB-IoT AT 指令与模块配置

(1) AT 指令的定义与使用

AT 指令是应用于调制解调器的一种指令语言，用于控制调制解调器进行数据通信。在 NB-IoT 模块中，AT 指令用于模块的配置和网络操作。AT 指令集包括了各种命令，用于查询和设置模块参数，如网络注册、发送和接收数据等。

在“2_NB-IoT AT 指令的使用”文档中，详细介绍了 AT 指令的使用方法和示例。例如，使用 AT+CGATT?指令可以查询模块是否附着到网络，而 AT+CGATT=1 则用于附着网络。此外，AT+CGSOCKWRITE 指令用于发送数据到网络，而 AT+CGSOCKREAD 用于从网络接收数据。

AT 指令的使用对于开发者来说至关重要，因为它们提供了一种简便的方法来控制 NB-IoT 模块的行为。通过熟练掌握 AT 指令，开发者可以有效地进行网络配置、数据传输和故障排除。

(2) NB 模块的设置

NB-IoT 模块的设置包括网络参数配置、安全设置、数据格式设置等。在“3_NB-IoT 智能路灯案例 代码流程讲解”文档中，详细介绍了如何通过 AT 指令对 NB-IoT 模块进行设置。

例如，设置模块的工作模式为 NB-IoT，可以使用 AT 指令 AT+CEMODE=1,1。而设置 APN 参数，允许模块通过运营商网络连接到互联网，可以使用指令 AT+CGDCONT=1,"IP","your_apn"。此外，模块的带宽、无线通信参数等也可以通过 AT 指令进行配置。

正确的模块设置对于确保 NB-IoT 设备能够顺利连接到网络并进行数据传输至关重要。通过合理的配置，可以最大化地利用 NB-IoT 技术的优势，如低功耗、广覆盖和高连接数等。

8.7.NB-IoT 硬件搭建与烧写

(1) 硬件搭建步骤

NB-IoT 硬件搭建是实现 NB-IoT 应用的基础，涉及选择合适的 NB-IoT 模块、传感器、微控制器以及电源管理模块。以下是详细的硬件搭建步骤：

- **选择 NB-IoT 模块**

根据应用需求选择支持 NB-IoT 标准的通信模块。模块需兼容现有蜂窝网络，并支持所需的频段。

- **传感器集成**

根据监测需求选择相应的传感器，如温湿度传感器、光照传感器等。传感器需与微控制器兼容，并能通过模拟或数字接口传输数据。

- **微控制器选择**

选择一个性能稳定的微控制器作为系统控制核心，负责处理传感器数据并通过NB-IoT 模块发送数据。

- **电源管理**

设计稳定的电源管理系统，包括电池的选择和电源管理电路的设计，确保系统在低功耗下稳定运行。

- **连接与接口**

确保所有模块和传感器正确连接，包括物理连接和通信接口的配置。

- **外壳与防护**

设计适合应用环境的外壳，提供必要的防水、防尘和抗冲击保护。

- **测试与验证**

在实际部署前进行充分的测试，验证系统的稳定性和数据准确性。硬件搭建的成本效益分析显示，使用 NB-IoT 模块的总成本比传统有线解决方案低 60%。

(2) 模块烧写流程

模块烧写是将预设的控制逻辑和网络配置写入 NB-IoT 模块的过程。以下是烧写的详细流程：

- **准备工作**

确保拥有烧写所需的软件工具和固件。这些通常由模块制造商提供。

- **连接设备**

使用适当的接口（如 USB 或串口）将 NB-IoT 模块连接到开发计算机。

- **配置参数**

通过 AT 指令设置模块的网络参数，包括运营商信息、APN 设置、网络注册等。

- **烧写固件**

使用烧写工具将固件程序传输到 NB-IoT 模块的闪存中。

- **验证设置**

烧写完成后，通过 AT 指令查询模块状态，验证设置是否正确。

- **测试通信**

在实际网络环境中测试模块的通信能力，确保数据能够成功发送和接收。

- **故障排除**

如果在烧写或测试过程中遇到问题，根据错误信息进行故障排除。

烧写过程包括设置路灯的工作时间、亮度调节策略以及故障上报机制。数据上云的成

功率达到了 99%，确保了数据的实时性和准确性。

8.8.NB-IoT 接入云平台操作

(1) 7.1 注册账号与项目创建

接入云平台的第一步是注册账号并创建项目。以下是详细的操作步骤：

- **注册云平台账号**

访问所选云平台的官方网站，按照提示填写必要的信息，如电子邮箱、密码等，完成账号注册。

- **创建新项目**

登录云平台后，通常需要创建一个新的项目或应用，以便管理和组织设备。在项目创建过程中，可能需要填写项目名称、描述以及选择相应的云服务和工具。

- **获取项目凭证**

创建项目后，云平台会生成一组凭证，包括项目 ID、API 密钥等，这些凭证将用于设备的认证和数据传输。

- **配置权限和安全设置**

根据项目需求配置权限，如数据访问权限、设备管理权限等。同时，确保启用了必要的安全措施，如 SSL 加密、访问控制列表等。

- **熟悉云平台功能**

了解云平台提供的各种功能和服务，如数据存储、分析、可视化等，以便后续设备的接入和数据的处理。

根据市场研究报告，通过云平台的设备管理，可以提高设备的监控效率和数据的可用性，降低维护成本约 20%。

(2) 设备接入与数据上云

设备接入云平台并实现数据上云是 NB-IoT 应用的关键环节。以下是详细的操作步骤：

- **设备注册**

在云平台上注册 NB-IoT 设备，通常需要提供设备的 IMEI 号、设备 ID 等信息，并与项目凭证关联。

- **配置设备信息**

设置设备的网络参数和通信协议，确保设备能够与云平台进行通信。这可能包括 APN 设置、网络注册参数等。

- **开发数据上云逻辑**

根据应用需求开发数据上云的逻辑，包括数据采集、处理、封装和发送。这通常涉及到编写固件或应用程序代码。

- **测试设备通信**

在实际网络环境中测试设备的通信能力，确保设备能够成功连接到云平台并发送数据。

- **监控数据流**

一旦设备接入云平台，就可以通过云平台的监控工具实时查看设备发送的数据。确保数据的准确性和完整性。

- **故障排除和优化**

在设备接入和数据上云的过程中，如果遇到问题，需要根据错误信息进行故障排除。同时，根据实际运行情况对设备和网络配置进行优化。

8.9. 课程思政案例

案例标题：华为海思 NB-IoT 芯片：自主创新的力量

8.9.1. 背景介绍

在全球化的背景下，关键核心技术的自主可控对于国家的长远发展至关重要。华为海思作为国内领先的半导体公司，其研发的 NB-IoT 芯片不仅打破了国外技术垄断，还推动了物联网技术在多个领域的广泛应用，展现了中国科技企业的创新实力和责任感。

8.9.2. 案例内容

1.自主创新，突破国际垄断：华为海思 NB-IoT 芯片的研发，是中国在半导体领域自主创新的重要成果。这一成果不仅减少了对外部技术的依赖，还提升了我国在全球通信技术领域的话语权。

2.科技助力社会发展：海思 NB-IoT 芯片的应用覆盖了智能抄表、智能停车、环境监测等多个领域，这些应用的实施有效提高了城市管理的效率，改善了市民的生活质量，体现了科技对社会进步的推动作用。

3.绿色发展，环保先行：在环境监测等领域，海思 NB-IoT 芯片的应用有助于实时监测环境状况，为环境保护提供了数据支持。这种技术的应用体现了绿色发展理念，促进了可持续发展。

4.促进产业升级，带动就业：海思 NB-IoT 芯片的广泛应用，推动了相关产业的升级，为社会提供了大量就业机会。这不仅促进了经济发展，还提高了人民的生活水平。

5.培养科技人才，储备国家力量：在海思 NB-IoT 芯片的研发和应用过程中，培养了一

大批高科技人才。这些人才的培养对于国家的长远发展具有重要意义，是国家竞争力的重要体现。

8.9.3. 案例分析

通过华为海思 NB-IoT 芯片的研发及应用案例，学生可以深刻理解到自主创新的重要性，以及科技在推动社会发展中的作用。同时，这个案例也展示了如何将绿色发展理念融入到技术应用中，以及科技在促进产业升级和人才培养方面的作用。

8.9.4. 思政教育目标

1. 增强学生的自主创新意识，让学生认识到自主创新对于国家发展的重要性。

2. 培养学生的社会责任感，理解科技在服务社会、改善民生中的作用。

3. 提升学生的环保意识，理解绿色发展理念在实际应用中的重要性。

4. 强化学生的团队合作意识，通过案例分析，让学生理解在复杂项目中团队合作的重要性。

5. 激发学生的爱国情怀，让学生为国家的科技进步和社会发展感到自豪。

通过这个案例，学生不仅能够学习到 NB-IoT 技术的应用，还能够从中领悟到科技创新对于国家和社会的重要性，以及作为未来科技人才的使命和责任。同时，这个案例也有助于培养学生的爱国情怀和社会责任感，激发他们为国家的科技进步和社会服务贡献自己的力量。

8.10. 本章小结

本章介绍了 NB-IoT 技术的基本概念、关键技术、协议框架、AT 指令使用、硬件搭建、烧写流程以及数据上云操作。通过本章的学习，学生应能够掌握 NB-IoT 技术的基础，并能够独立完成 NB-IoT 项目的开发和实施。NB-IoT 技术以其低功耗、广覆盖、大连接数等特点，在物联网领域具有广泛的应用前景。

第九章 LoRa 无线网络技术应用

（一）本章教学目标

主要内容：本章将详细介绍 LoRa 无线网络技术的基础知识、关键技术、协议框架、AT 指令使用、硬件搭建、烧写流程以及数据上云操作。

目的与要求：目的是使学生能够掌握 LoRa 技术的基础，了解其在物联网中的应用，理解 AT 指令的重要性，并能够独立完成硬件搭建和数据上云操作。

（二）本章重点难点

1．掌握 LoRa 的基本概念及关键技术

2．理解 LoRa 协议框架和 AT 指令的使用

3．独立完成 LoRa 硬件搭建和烧写流程

4．实现数据成功上云

9.1. LoRa 技术概述

9.1.1. LoRa 的定义

LoRa（Long Range Radio），即远距离无线电技术，是一种基于扩频技术的远距离无线传输技术。LoRa 是 LPWAN（Low-Power Wide-Area Network，低功耗广域网）通信技术中的一种，由 SEMTECH 公司创建的低功耗局域网无线标准。

9.1.2. LoRa 的发展背景

随着物联网市场的快速增长，传统的无线通信技术已无法满足大规模物联网部署的需求。LoRa 技术应运而生，旨在解决这些挑战，由 3GPP（第三代合作伙伴计划）在 Release 13 中正式标准化，并得到了全球主要电信运营商和设备制造商的支持。

9.2. LoRa 技术特点

长距离传输：城镇可达 2-5 Km，郊区可达 15 Km

工作在 ISM 频段，主要包括 433、868、915 MHz 等

基于扩频技术，线性调制扩频（CSS）的一个变种，具有前向纠错（FEC）能力

一个 LoRa 网关可以连接上千上万个 LoRa 节点

电池寿命长达 10 年

AES128 加密，保障通信安全

传输速率几百到几十 Kbps，速率越低传输距离越长

9.3. LoRa 无线网络技术应用

LoRa 技术在智能抄表、智能停车、环境监测、农业自动化等多个领域有广泛应用。通过 LoRa 模块，设备可以定期将数据发送到中心服务器，实现远程监控和管理。

9.4. LoRa 应用案例分析

案例标题：基于 LoRa 的温湿度传感器节点应用程序开发

背景介绍：随着物联网技术的发展，对于低功耗、远距离传输的需求日益增长。LoRa 技术以其低功耗、长距离传输的特点，成为物联网领域的理想选择。

案例内容：通过开发 LoRa 温湿度传感器节点应用程序，可以实现对环境温湿度的实时监测。通过 LoRa 模块，传感器节点可以将采集到的数据发送到网关，再由网关将数据上云，实现数据的远程访问和分析。

9.5. LoRa AT 指令与模块配置

AT 指令的定义与使用：AT 指令是应用于调制解调器的一种指令语言，用于控制调制解调器进行数据通信。在 LoRa 模块中，AT 指令用于模块的配置和网络操作。

9.6. LoRa 硬件搭建与烧写

硬件搭建步骤：涉及选择合适的 LoRa 模块、传感器、微控制器以及电源管理模块。模块烧写是将预设的控制逻辑和网络配置写入 LoRa 模块的过程。

9.7. LoRa 接入云平台操作

注册账号与项目创建：接入云平台的第一步是注册账号并创建项目。设备接入与数据上云：设备接入云平台并实现数据上云是 LoRa 应用的关键环节。

9.8. LoRa 无线网络技术应用实践

9.8.1. LoRa 温湿度传感器节点开发

学生将学习如何开发 LoRa 温湿度传感器节点应用程序，包括硬件连接、工程模板操作、应用程序编程、程序烧写等步骤。

9.8.2. LoRa 光照传感器节点开发

学生将学习如何开发 LoRa 光照传感器节点应用程序，包括采集光照度数据，并在 OLED 屏上显示，以及如何响应网关读取传感数据的指令。

9.8.3. 网关节点汇聚传感器数据

学生将学习如何开发 LoRa 网关节点应用程序，包括网关轮流读取温湿度传感器节点、光照传感器节点的传感器数据，并将收到的传感器数据在 OLED 屏上显示，同时将数据透传到串口上。

9.8.4. LoRa 数据上云

学生将学习如何将 LoRa 网关模块连接至云平台，上传采集数据，并在云平台上实时显示和存储数据。

9.9. 课程思政案例

案例标题：自主创新的力量——LoRa 技术在中国的发展

9.9.1. 背景介绍

在全球化的背景下，关键核心技术的自主可控对于国家的长远发展至关重要。LoRa 技术作为一种新兴的物联网通信技术，在全球范围内得到了广泛的应用。中国在 LoRa 技术的研发和应用上，展现了自主创新的力量和责任感。

9.9.2. 案例内容

1. 自主创新，突破国际垄断：中国科技企业在 LoRa 技术领域的研发，是中国在物联网领域自主创新的重要成果。这一成果不仅减少了对外部技术的依赖，还提升了我国在全球物联网技术领域的话语权。

2. 科技助力社会发展：LoRa 技术的应用覆盖了智能抄表、智能停车、环境监测等多个领域，这些应用的实施有效提高了城市管理的效率，改善了市民的生活质量，体现了科技对社会进步的推动作用。

3. 绿色发展，环保先行：在环境监测等领域，LoRa 技术的应用有助于实时监测环境状况，为环境保护提供了数据支持。这种技术的应用体现了绿色发展理念，促进了可持续发展。

4. 促进产业升级，带动就业：LoRa 技术的广泛应用，推动了相关产业的升级，为社会提供了大量就业机会。这不仅促进了经济发展，还提高了人民的生活水平。

5. 培养科技人才，储备国家力量：在 LoRa 技术的研发和应用过程中，培养了一大批高科技人才。这些人才的培养对于国家的长远发展具有重要意义，是国家竞争力的重要体现。

9.9.3. 案例分析

通过 LoRa 技术在中国的发展案例，学生可以深刻理解到自主创新的重要性，以及科技在推动社会发展中的作用。同时，这个案例也展示了如何将绿色发展理念融入到技术应用中，以及科技在促进产业升级和人才培养方面的作用。

9.9.4. 思政教育目标

1. 增强学生的自主创新意识，让学生认识到自主创新对于国家发展的重要性。

2. 培养学生的社会责任感，理解科技在服务社会、改善民生中的作用。

3. 提升学生的环保意识，理解绿色发展理念在实际应用中的重要性。

4. 强化学生的团队合作意识，通过案例分析，让学生理解在复杂项目中团队合作的重要性。

5. 激发学生的爱国情怀，让学生为国家的科技进步和社会发展感到自豪。

通过这个案例，学生不仅能够学习到 LoRa 技术的应用，还能够从中领悟到科技创新对于国家和社会的重要性，以及作为未来科技人才的使命和责任。同时，这个案例也有助于培养学生的爱国情怀和社会责任感，激发他们为国家的科技进步和社会服务贡献自己的力量。

9.10. 本章小结

本章介绍了 LoRa 技术的基本概念、关键技术、协议框架、通信协议、硬件搭建、烧写流程以及数据上云操作。通过本章的学习，学生应能够掌握 LoRa 技术的基础，并能够独立完成 LoRa 项目的开发和实施。LoRa 技术以其低功耗、广覆盖、大连接数等特点，在物联网领域具有广泛的应用前景。同时，通过课程思政案例的学习，学生能够进一步理解自主

创新的重要性，以及科技在推动社会发展中的作用。

第十章 综合项目——ZigBee 开发项目使用教程

前言：经过前面实验例程的学习，相信大家对 ZigBee 开发产生了兴趣，在此为大家解析关于综合项目中的 ZigBee 相关源码。

一）． 首先大家得知道整个项目是怎么写出来的，开发只能站在巨人的肩膀上，所以不可能是由开发者自己编写出一个协议栈的，这个项目是基于 TI 官方协议栈 ZStack-CC2530-2.3.1-1.4.0 的 SimpleApp 例程进行开发，如图 7.1 所示。

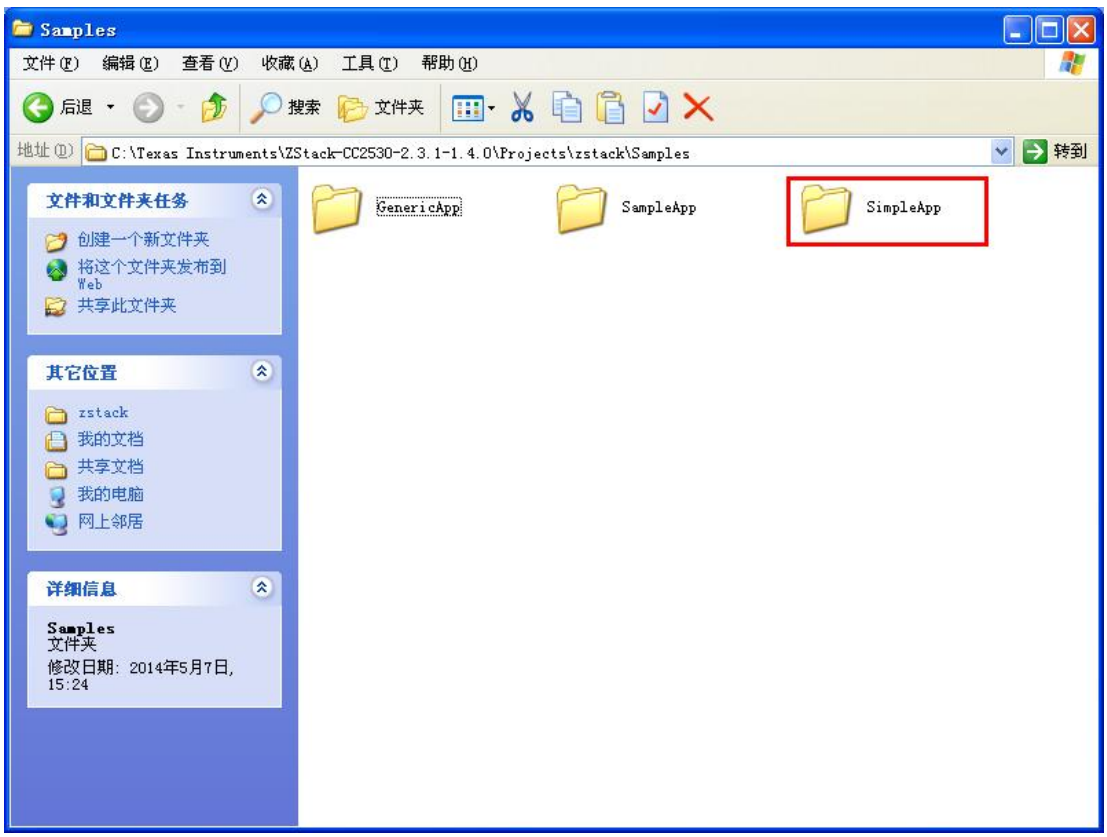


图 7.1

TI 官方协议栈 ZStack-CC2530-2.3.1-1.4.0 也称为 ZigBee2007 协议栈，在 C:\Texas Instruments\ZStack-CC2530-2.3.1-1.4.0\Projects\zstack\Samples\SimpleApp\CC2530DB 目录下使用 IAR 软件集成开发环境打开 SimpleApp.eww 工程文件，如图 7.2 所示

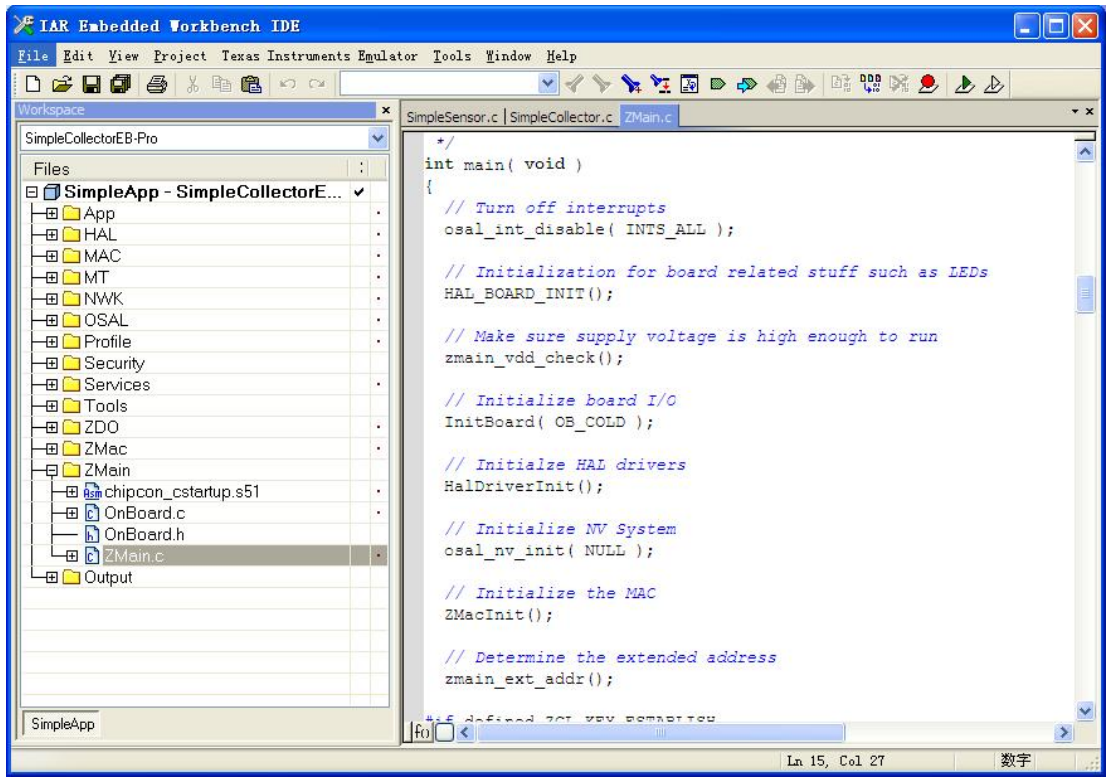


图 7.2

二）． 在综合项目应用中的 ZigBee 可能会有很多的传感器代码，将如何进行烧写，这是必须得解决的事情，首先打开综合项目 ZigBee 相关源码，红色方框就是项目的一些传感器代码： Projects\zstack\Samples\SensorBB\CC2530DB 目录下的 SensorDemo.eww 工程文件

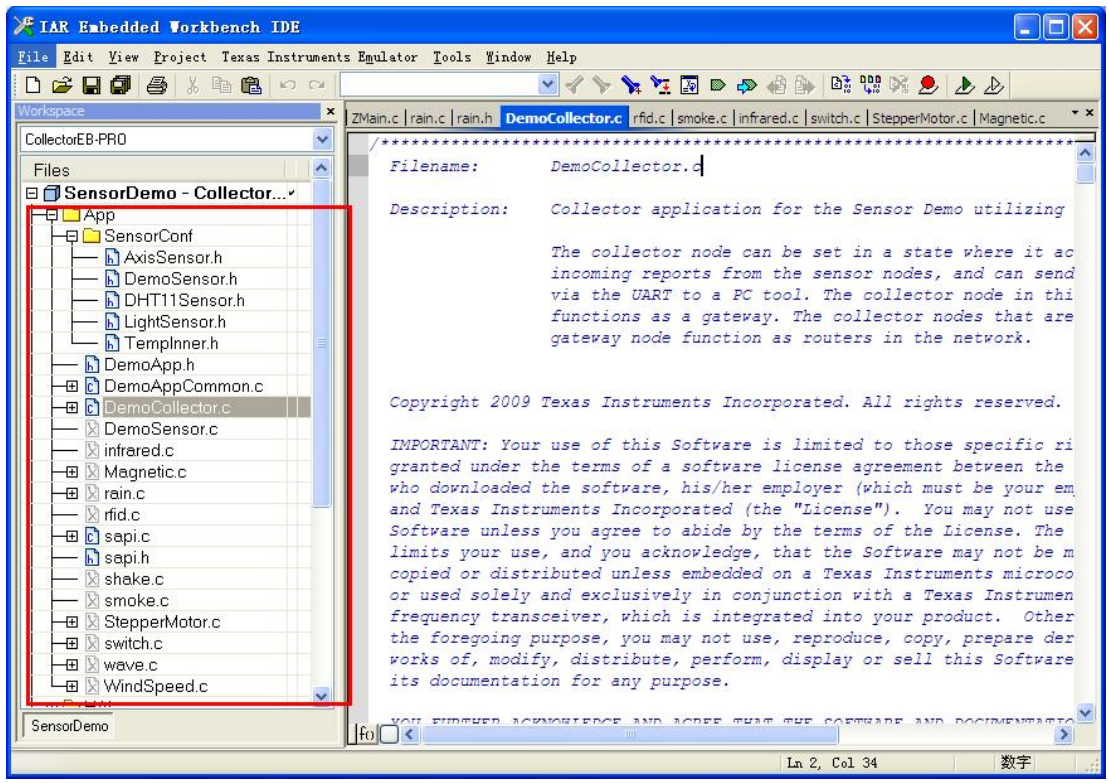


图 7.3

烧写各个传感器节点，需按如下配置：

温湿度+光感传感器,如图 7.4 和图 7.5 所示：

1、打开/SensorConf/DemoSensor.h，找到"#define SENSORTYPE"将其 define 为 3

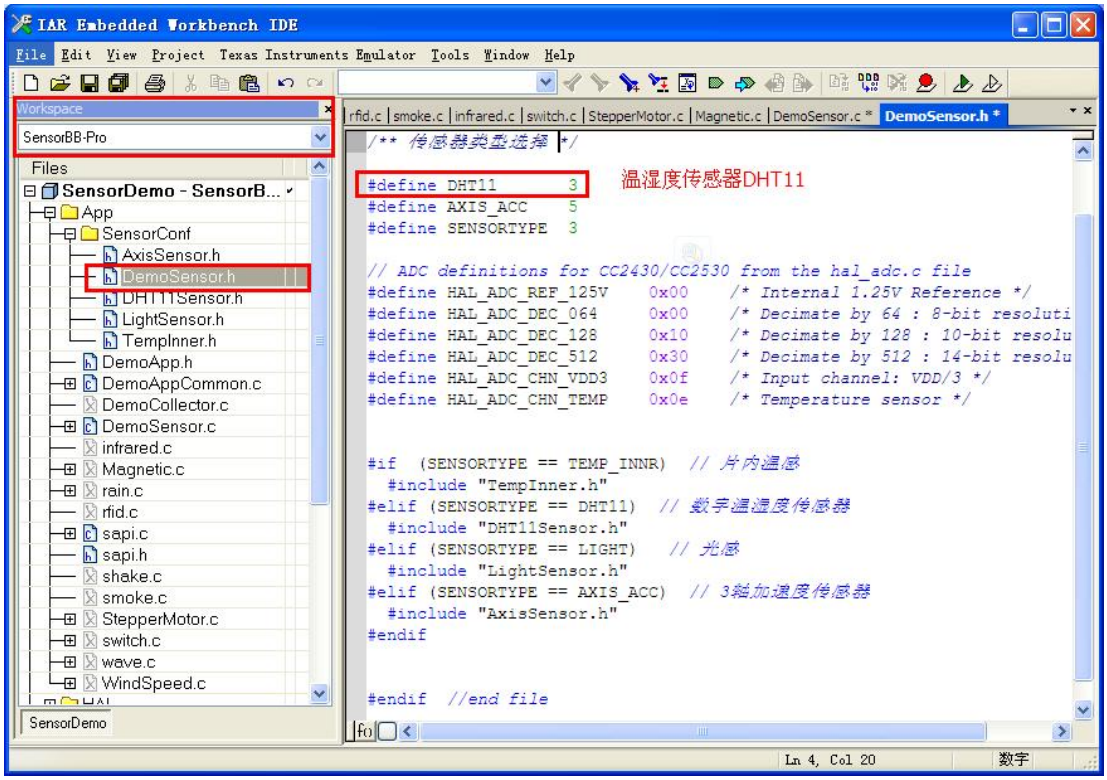


图 7.4

2、打开 DemoSensor.c，找到"#define SENSOR_TYPE" 将其 define 为 0x03

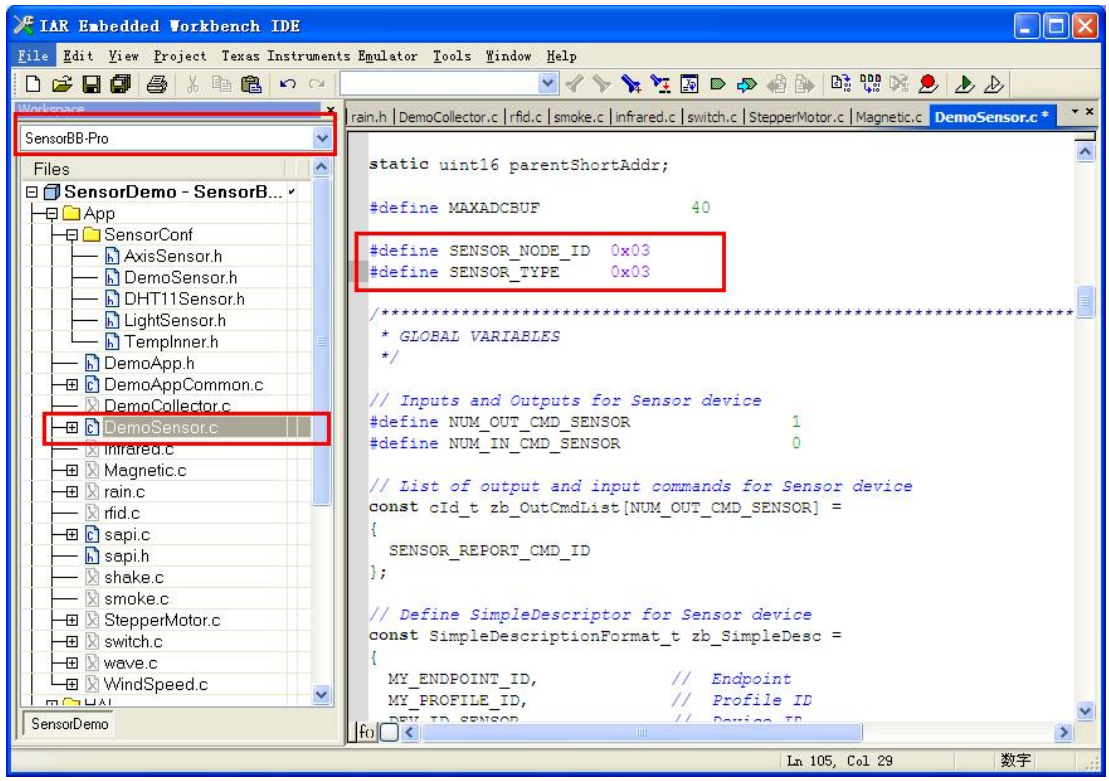


图 7.5

1) 三轴加速度传感器,如图 7.6 和图 7.7 所示：

1、打开/SensorConf/DemoSensor.h，找到"#define SENSORTYPE"将其 define 为 5

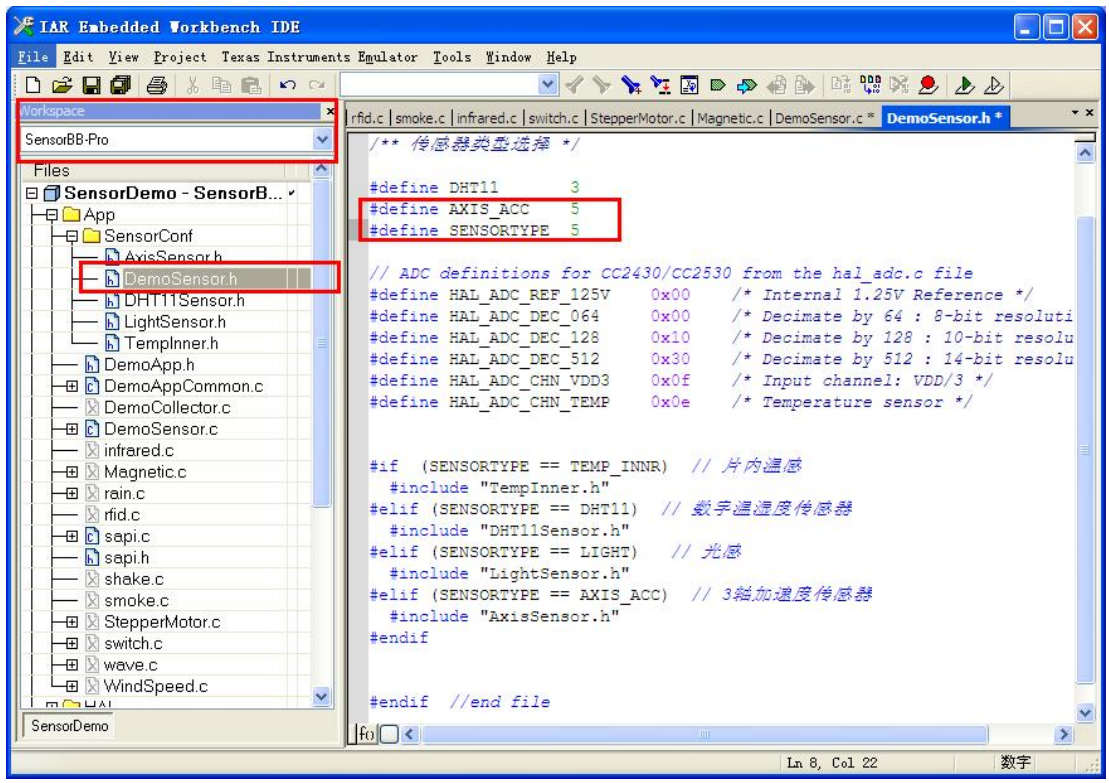


图 7.6

2、打开 DemoSensor.c，找到"#define SENSOR_TYPE" 将其 define 为 0x05

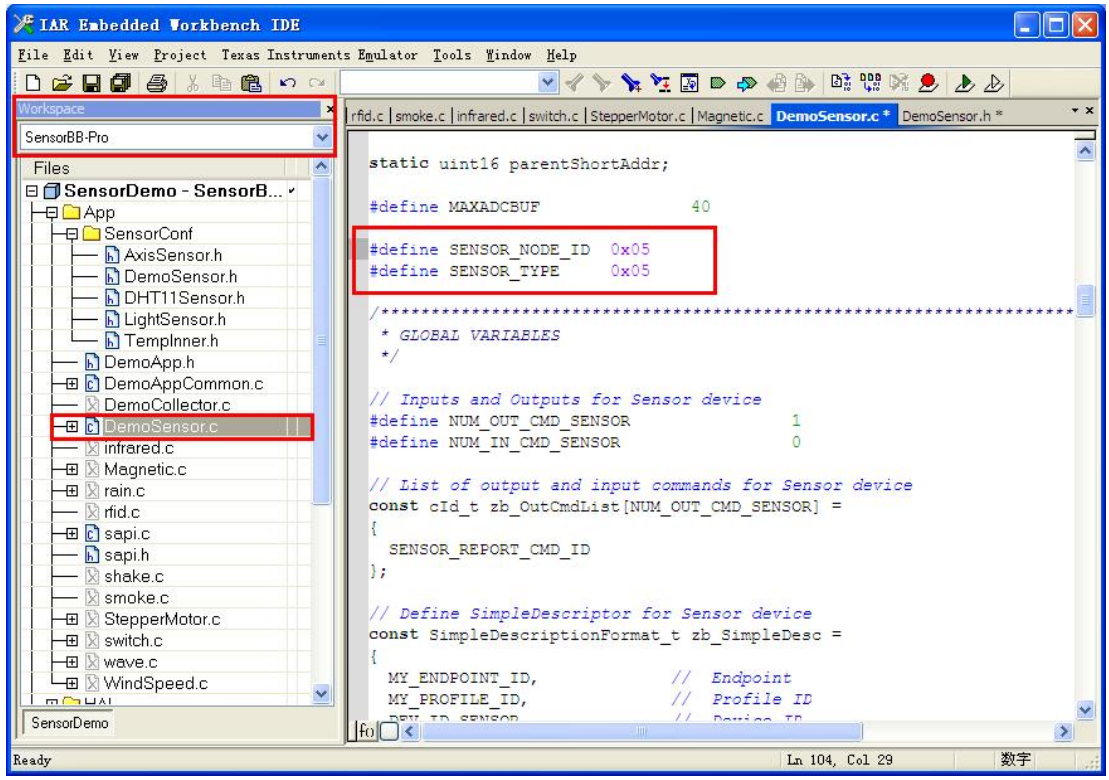


图 7.7

2） 直流电机模块，如图 7.8 所示：

打开 StepperMotor.c，

找到"#define SENSOR_NODE_ID "将其 define 为 7

找到"#define SENSOR_TYPE"将其 define 为 7

找到"#define MOTOR_MODE"将其 define 为 NORMAL

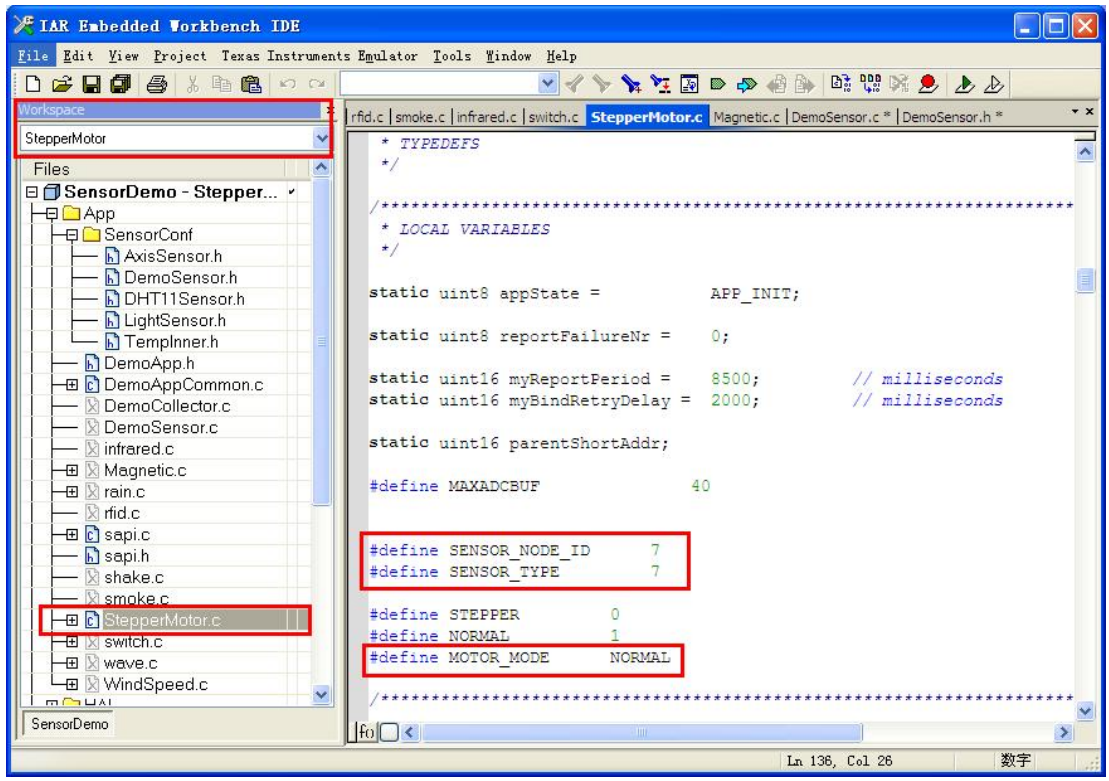


图 7.8

3) 步进电机模块,如图 7.9 所示：

打开 StepperMotor.c

找到"#define SENSOR_NODE_ID "将其 define 为 6

找到"#define SENSOR_TYPE"将其 define 为 6

找到"#define MOTOR_MODE"将其 define 为 STEPPER

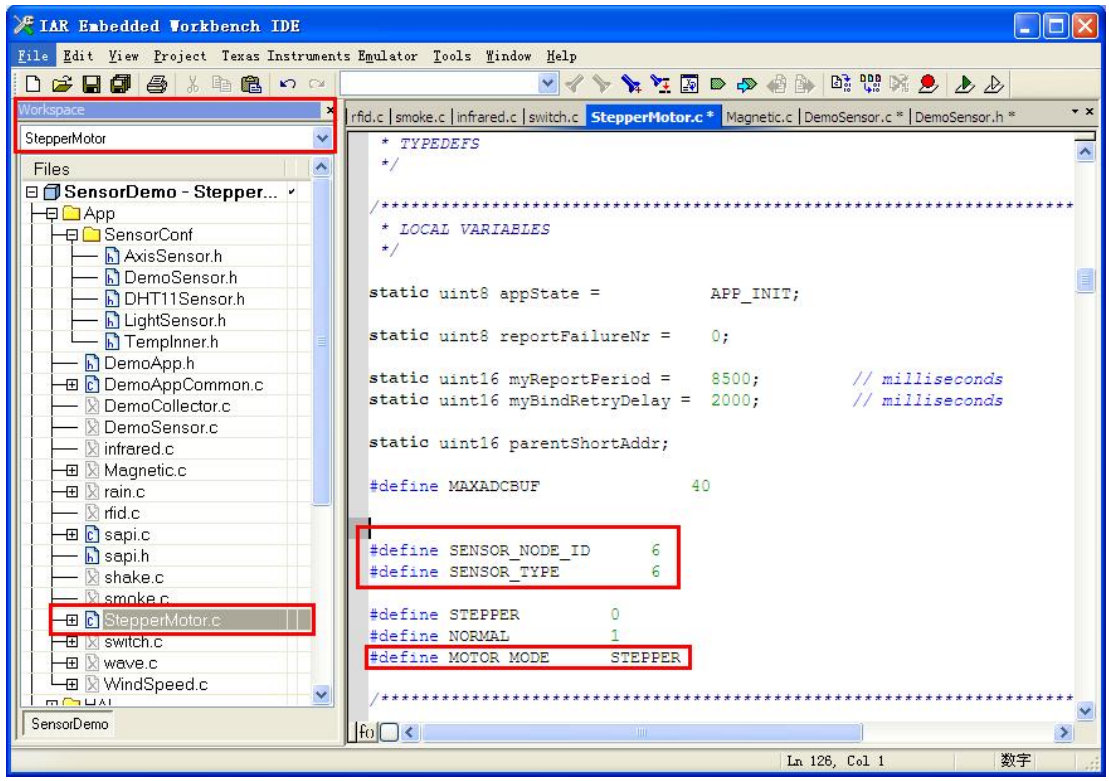


图 7.9

其他传感器如：RFID 模块，烟雾传感器，继电器模块，磁控传感器，人体热释电传感器等，无需任何设置，选择传感器所对应的 Workspace 选项进行编译下载，如图 7.10 所示：

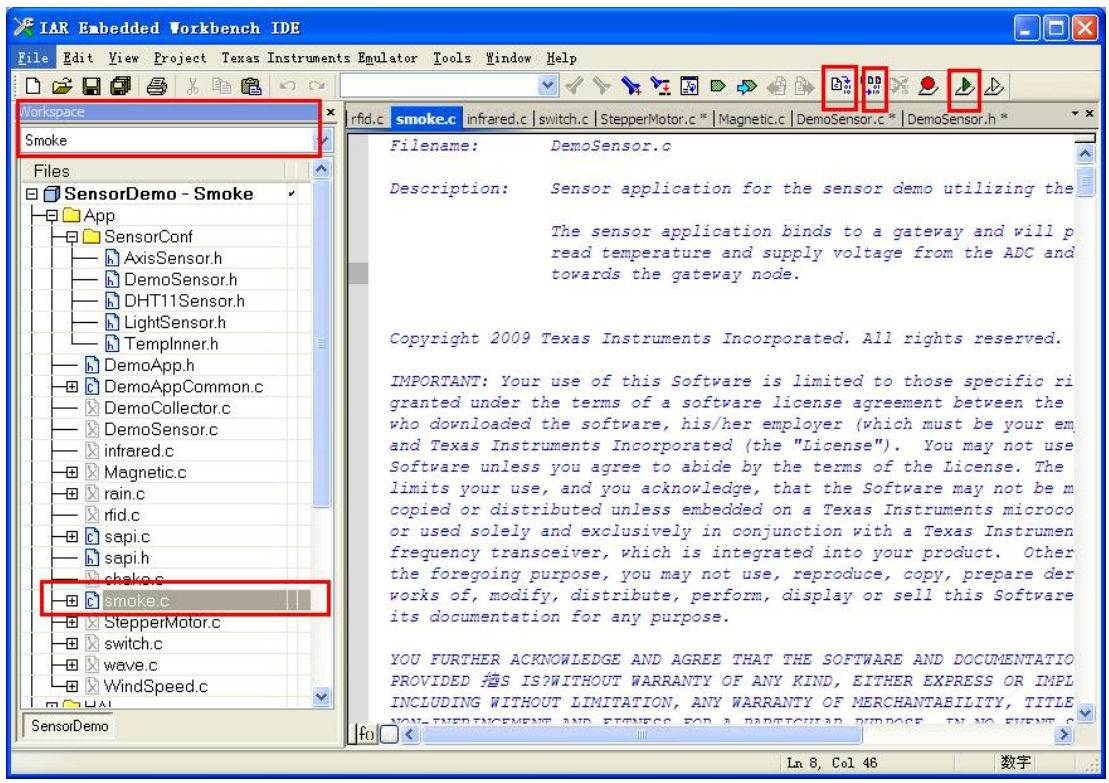


图 7.10

三）. 在综合项目应用 ZigBee 烧写完成之后，给传感器起个名字，可能有同学有疑惑传感器不是有名字吗：看，这不是烟雾传感器、雨滴传感器...，但是程序不认识啊，所以为了方便用户去了解和使用，就必须起名字了。叫张三还是李四就看代码中的如何规范的，下面就是对传感器身份进行识别：

传感器节点类型分配表

传感器名称	节点号	传感器类型
雨滴传感器	0x00	0x00
RFID	0x01	0x01
烟雾传感器	0x02	0x02
温湿度+光感传感器	0x03	0x03
红外热释电	0x04	0x04
三轴加速度传感器	0x05	0x05
步进电机模块	0x06	0x06
直流电机模块	0x07	0x07
继电器模块	0x08	0x08
门磁检测传感器	0x0A	0x0A

四）. **最后才是关键，应该怎么才知道打印的数据是否正确，是不是我想要的数据呢，下面对数据帧进行解析：**

串口波特率配置：115200

启动网络命令表(最先输入)：

Collector 收集器节点可以接收 UART 命令（16 进制）	
初始化传感器网络：	FE 00 01
允许传感器组网：	FE 00 02
不允许传感器组网：	FE 00 03

扫描节点存在：

帧头：	FE
数据长：	0F

命令：	46　87
短地址(低高)：	00　00
节点(0~12)：	00
数据包长度：	07　00
节点号：	00
校验值：	CE

1. **雨滴检测传感器**

雨滴检测数据：

帧头：	FE
数据长：	0F
命令：	46　87
短地址(低高)：	F1　CE
cmd：	0B
传感器类型：	00
数据包长度：	07　00
节点号：	00
RSSI：	D9
父节点地址：	00　00
雨量检测数据	66
校验值：	41

2. **RFID 传感器**

串口打印指令解析表(cmd 作为判断位)：

有 IC 卡检测数据：　　无 IC 卡检测数据：

帧头：	FE
数据长：	0A
命令：	46　87
短地址	C3　89

(低高)：	
cmd：	00
传感器类型：	01
数据包长度：	07 00
节点号：	01
RSSI：	D2
父节点地址：	00 00
RFID 信息	8F 1A 18 B2
校验值：	6D

3. 烟雾检测传感器

串口打印指令解析表(cmd 作为判断位)：

有烟雾检测数据：

帧头：	FE
数据长：	0B
命令：	46 87
短地址(低高)：	C3 89
cmd：	05
传感器类型：	01
数据包长度：	07 00
节点号：	01
RSSI：	D5
父节点地址：	00 00
RFID 信息	无
校验值：	54

无烟雾检测数据：

帧头：	FE
数据长：	0A
命令：	46 87
短地址(低高)：	C5 87
cmd：	06
传感器类型：	02
数据包长度：	07 00
节点号：	02
RSSI：	D5
父节点地址：	00 00
校验值：	3B

帧头：	FE
数据长：	0B
命令：	46 87
短地址(低高)：	C5 87
cmd：	00
传感器类型：	02
数据包长度：	07 00
节点号：	02
RSSI：	CE
父节点地址：	00 00
校验值：	25

4. **温湿度传感器**

温湿度检测数据：

帧头：	FE
数据长：	0D
命令：	46 87
短地址(低高)：	3B A7
cmd：	02
传感器类型：	03
数据包长度：	07 00
节点号：	03
RSSI：	D7
父节点地址：	00 00
湿度	26
温度	1C
光照度	8D
校验值：	96

5. **人体热释电传感器**

串口打印指令解析表(cmd 作为判断位)：

有触发数据：

帧头：	FE
数据长：	0A
命令：	46　87
短地址(低高)：	6B　BF
cmd：	08
传感器类型：	04
数据包长度：	07　00
节点号：	04
RSSI：	CF
父节点地址：	00　00
校验值：	26

无触发数据：

帧头：	FE
数据长：	0B
命令：	46　87
短地址(低高)：	6B　BF
cmd：	00
传感器类型：	04
数据包长度：	07　00
节点号：	04
RSSI：	CF
父节点地址：	00　00
校验值：	25

6. 3D 加速度传感器

帧头：	FE
数据长：	0D
命令：	46　87
短地址(低高)：	5C　4E
cmd：	02
传感器类型：	05
数据包长度：	07　00
节点号：	05
RSSI：	D5
父节点地址：	00　00

X	12
Y	F8
Z	1F
校验值：	D3

7. 步进电机模块

帧头：	FE
数据长：	0A
命令：	46 87
短地址(低高)：	6E 67
cmd：	00
传感器类型：	06
数据包长度：	07 00
节点号：	06
RSSI：	D3
父节点地址：	00 00
校验值：	D3

步进电机串口控制命令：

帧头：	FE	FE	FE	FE
命令：	02 01	02 01	02 01	02 01
短地址(高低)：	67 6E	67 6E	67 6E	67 6E
方向位	83	FF	83	03
motorCnt[16]	FF FF	FF FF	FF FF	FF FF
周期位(不用输入)	1000 0011	1111 1111	1000 0011	0000 0011
	最快	最慢	正向最快	反向最快

8. 直流电机模块

帧头：	FE

数据长：	0A
命令：	46 87
短地址(低高)：	6E 67
cmd：	00
传感器类型：	07
数据包长度：	07 00
节点号：	07
RSSI：	D8
父节点地址：	00 00
校验值：	D3

直流电机控制命令：

帧头：	命令：	短地址(高低)：	方向位(停/正/反)
FE	02 02	67 6E	00/01/02

9. 继电器模块

帧头：	FE
数据长：	0A
命令：	46 87
短地址(低高)：	FA E9
cmd：	00
传感器类型：	08
数据包长	07 00

度：	
节点号：	08
RSSI：	DF
父节点地址：	00 00
校验值：	D3

继电器控制命令：

帧头：	命令：	短地址(高低)：	开/关
FE	0101/02/03	E9 FA	01/00

备注：命令的 01/02/03 分别为继电器 1、继电器 2、继电器 3

LED 控制命令：

帧头：	命令：	短地址(高低)：	开/关
FE	0104/05	E9 FA	01/00

备注：命令的 04/05 分别为 LED1、LED2

10. 磁控传感器

串口打印指令解析表(cmd 作为判断位)：

有磁检测数据： 无磁检测数据：

帧头：	FE
数据长：	0B
命令：	46 87
短地址(低高)：	01 7B
cmd：	10
传感器类型：	03
数据包长	07 00

帧头：	FF
数据长：	0B
命令：	46 87
短地址(低高)：	01 7B
cmd：	00
传感器类型：	03
数据包长度：	07 00
节点号：	10
RSSI：	DD
父节点地址：	00 00
门磁数据	00
校验值：	4E

度：	
节点号：	10
RSSI：	DD
父节点地址：	00 00
门磁数据	01
校验值：	4E