安全提示

非常感谢您购买方源智能科技的产品，在打开包装箱后请首先依据物件清单检查配件，若发现物件有所损坏、或是有任何配件短缺的情况，请尽快与您的经销商联络。

- 产品使用前，务必仔细阅读产品相关说明。
- 主板与电源连接时，请确认电源电压。
- 为了保证您是使用安全，请使用平台的专用电源。
- 接触平台主板前，应将手先置于接地金属物体上一会儿，以释放身体及手中的静电。
- 为避免人体被电击或产品被损坏，在每次对主板、扩展卡进行拔插或重新配置时，须先关闭交流电源或将交流电源线从电源插座中拔掉。
- 在对平台进行搬动前，先将交流电源线从电源插座中拔掉。
- 当您需连接或拔除设备前，须确定所有的电源线事先已被拔掉。
- 设备在使用过程中出现异常情况，请找专业人员处理。

版本声明

本文档为全功能物联网教学科研平台配套实验说明。所述实验内容仅限该实验平台使用。

修订描述				
日期	修订版本	描述	编辑	备注
2012-06-20	V0.5	Alpha Edition	i-csource	
2012-07-20	V1.0	Release Edition	i-csource	
2012-09-13	V1.5	Release Edition	i-csource	
2013-02-20	V2.0	Release Edition	i-csource	
2013-09-11	V2.5	Release Edition	i-csource	
2014-09-22	V3.0	Release Edition	i-csource	
2015-02-27	V3.5	Release Edition	i-csource	
2015-04-26	V4.0	Release Edition	smartt	
2016-07-04	V4.5	Release Edition	syj	
2019-05-20	V4.6	文档格式修订	chensd	

ICS-IOT-CEP 全功能物联网教学科研实验平台配套实验指导文档共包含三册，本册为第一册，可用于 ICS-IOT-CEP 全功能物联网教学科研实验平台配套智能传感器、蓝牙、WIFI、IPv6、ZigBee 等模块技术部分课程实验参考。

备注：此外全功能平台非标配的硬件模块对应相关软件资料及文档，请查阅其他相关资料，此文档中不再涉及。

目录

第一章. 实验环境与软件工具

本章主要介绍基于 ICS-IOT-CEP 全功能物联网教学科研平台软硬件的资源，以及相应实验体系的开发环境搭建过程，并着重讲解在使用该平台过程中用到的一些常见开发软件和相关服务设置。

本章内容作为 ICS-IOT-CEP 全功能物联网教学科研平台实验教学的基础内容，是顺利完成后续章节实验内容的重要基石，请用户或读者在动手实验之前务必首先仔细阅读本章内容。

1. 开发平台简介

1.1 产品概述

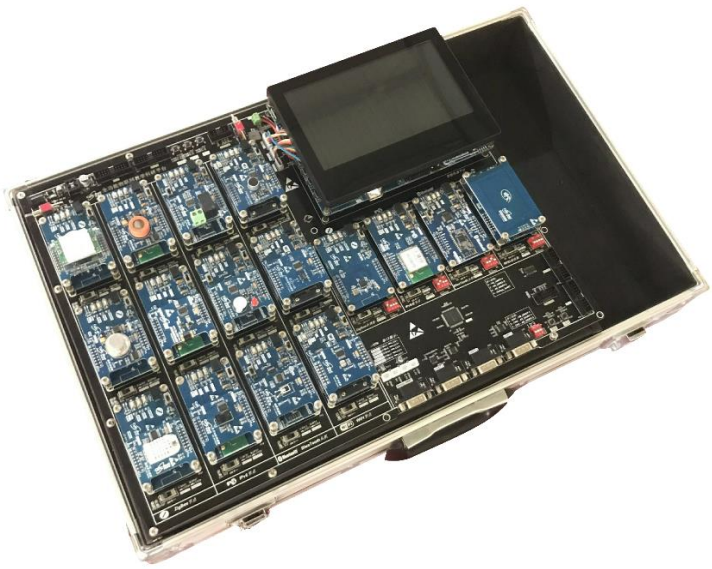


图 1.1.1.1　ICS-IOT-CEP 开发平台

全功能物联网教学科研平台是方源智能（北京）科技有限公司基于物联网多功能、全方位教学科研需求，推出的一款集无线 ZigBee、IPv6、Bluetooth、Wifi、RFID 和智能传感器等通信模块于一体的全功能物联网教学科研平台，以强大的 Cortex-A53（可支持 Linux/Android）嵌入式处理器作为核心智能终端，支持多种无线传感器通讯模块组网方式。由浅入深，提供丰富的实验例程和文档资料，便于物联网无线网络、传感器网络、RFID 技术、嵌入式系统及下一代互联网等多种物联网课程的教学和实践。

全功能物联网教学科研平台的应用结构拓朴图如下图 1.1.1.2 所示：

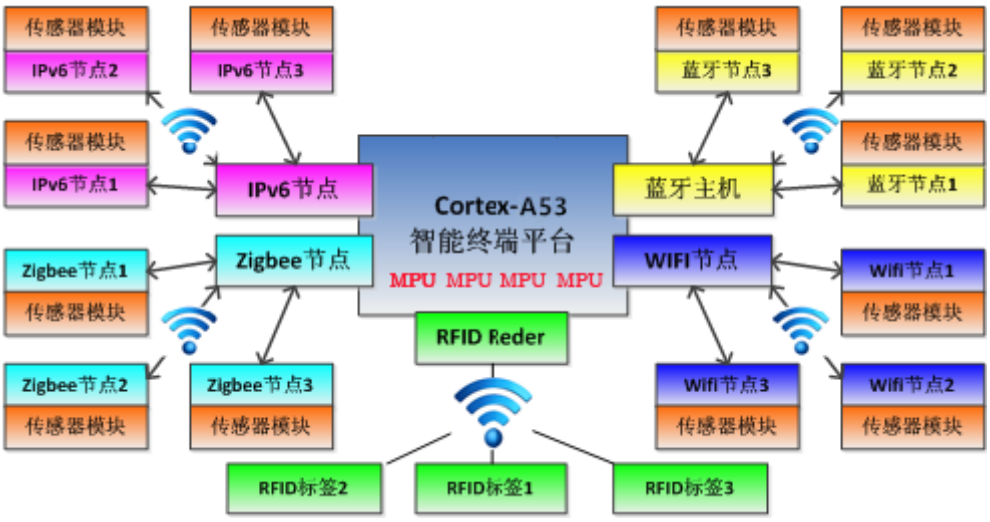


图 1.1.1.2　ICS-IOT-CEP 平台应用拓扑结构

1.2 产品特点

- 丰富快捷的无线组网功能

系统配备 ZigBee、IPv6、蓝牙、WIFI 四种无线通信节点及 RFID 读/写卡器，可以快速构成小规模 ZigBee、IPv6、蓝牙、WIFI 无线传感器通信网络。

- 丰富的传感器数据采集和扩展功能

配备温湿度、光敏、震动、三轴加速计、红外热释、烟雾等 12 种基于 MCU 的智能传感器模块，可以通过标准接口与通信节点建立连接，实现传感器数据的快速采集和通信。

- 可视化终端界面开发

基于 Qt 的跨平台图形界面开发，用户可以快速开发友好的人机界面。

1.3 平台硬件资源

全功能物联网科研教学平台硬件由 Cortex-A53 嵌入式智能终端、无线通讯模块和智能传感器模块几部分构成。

Cortex-A53 智能终端	
CPU 处理器	处理器 Samsung S5P6818,八核心处理器,基于 ARM Quad CortexM-A53,运行主频 1.4GHz 内置 Mali-400 MP 高性能图形引擎 支持流畅的 2D/3D 图形加速 最高可支持 1080p@60fps 硬件解码视频流畅播放，格式可为 MPEG4,H.263,H.264 等 最高可支持 1080p@30fps 硬件编码(Mpeg-2/VC1)视频输入
RAM 内存	1G DDR3 32bit 数据总线，单通道 运行频率: 400MHz
FLASH 存储	emmc 8GB
显示	7 寸 LCD 液晶电阻触摸屏
接口	1 路 HDMI 输出 4 路串口，RS232 *2、TTL 电平 *4 USB Host 2.0、mini USB Slave 2.0 接口 3.5mm 立体声音频(WM8960 专业音频芯片)输出接口、板载麦克风 1 路标准 TF 卡座 千兆以太网 RJ45 接口

	CMOS 摄像头接口 其中 AIN0 外接可调电阻，用于测试 I2C-EEPROM 芯片(256byte) ，主要用于测试 I2C 总线 用户按键(中断式资源引脚) *5 PWM 控制蜂鸣器 板载实时时钟备份电池
电源	电源适配器 5V(支持睡眠唤醒)

表 1.1.3.1 Cortex-A53 智能终端

无线通信节点	
ZigBee 节点（TI 方案标配）	● 处理器 CC2530，内置增强型 8 位 51 单片机和 RF 收发器，符合 IEEE802.15.4/ZigBee 标准规范，频段范围 2045M-2483.5M,板载天线
	● 存储器：256KB 闪存和 8KB RAM
	● 射频数据速率：250kbps，可编程的输出功率高达 4.5 dB
	● 用户自定制：按键 *2，LED *2
	● 供电电压：2V-3.6V，支持电池供电
	● 扩展调试接口
IPv6 节点	● 处理器 STM32W108，基于 ARM Cortex-M3 高性能的微处理器，集成了 2.4GHz IEEE 802.15.4 射频收发器，板载天线
	● 存储器：128KB 闪存和 8KB RAM，
	● 用户自定制：按键 *1，LED *2
	● 供电电压：3.7V 收发电流： 27mA/40mA，支持电池供电
	● 扩展 J-OB 仿真器和 J-OB 转接板调试接口
蓝牙节点	● CC2540 蓝牙模块，板载天线
	● 处理器 STM32F103 基于 ARM Cortex-M3 内核，主频 72MHz
	● 完全兼容蓝牙 4.0 规范，硬件支持数据和语音传输，最高可支持 3M 调制模式
	● 支持 UART 透传，IO 配置
	● 扩展 J-OB 仿真器和 J-OB 转接板接口，外设主从开关，支持一键主从模式转换
	● 支持电池供电
WIFI 节点	● 型号：嵌入式 wifi 模块（支持 802.11b/g/n 无线标准）内置板载天线
	● 处理器 STM32F103 基于 ARM Cortex-M3 内核，主频 72MHz
	● 支持多种网络协议：TCP/IP/UD，支持 UART/以太网数据通讯接口
	● 支持无线工作在 STA/AP 模式，支持路由/桥接模式网络架构
	● 支持透明协议数据传输模式，支持串口 AT 指令
	● 扩展 J-OB 仿真器和 J-OB 转接板接口
	● 支持电池供电
RFID 阅读器	● MF RC531（高集成非接触读写卡芯片）支持 ISO/IEC 14443A/B 和 MIFARE 经典协议
	● 处理器 STM8S105 高性能 8 位架构的微控制器，主频 16MHz
	● 支持 mifare1 S50 等多种卡类型
	● 用户自定制：按键 *1，LED *1
	● 最大工作距离：100mm，最高波特率：424kb/s
	● Crypto1 加密算法并含有安全的非易失性内部密匙存储器
	● 扩展 ST-Link 接口

表 1.1.3.2　无线通讯模块

传感器模块		
处理器	● STM8S103 高性能 8 位框架结构的微控制器，主频 1MHz	
外设	● LED 灯、UART 串口及电源接口	
传感器种类	● 磁检测传感器 ● 光照传感器 ● 红外对射传感器 ● 红外反射传感器 ● 结露传感器 ● 酒精传感器	● 人体检测传感器 ● 三轴加速度传感器 ● 声响检测传感器 ● 温湿度传感器 ● 烟雾传感器 ● 振动检测传感器

表 1.1.3.3　传感器模块

外扩辅助模块	
USB-UART 扩展板	**核心芯片：** FT232RL
	功能：连接 PC 机与网络节点串口调试功能
	接口：VCC GND TXD GND RXD
电池模块	**功能：**锂电池供电,提供低电压报警,提示用户充电
	接口：3.7V
电池充电板	5V 电源适配器,双路锂电池充电
调试工具	ST-Link 仿真调试工具、J-OB 仿真器和 J-OB 转接板仿真调试工具

表 1.1.3.4　外扩模块

1.4 平台软件资源

Cortex-A53 智能终端平台	**操作系统：** Linux-3.4.9 + Qt4.7/Qtopia2/Qtopia4、Android 5.1 **实验内容：**可进行 Linux 系统嵌入式编程开发，包括开发环境搭建、Bootloader 开发、嵌入式操作系统移植、驱动程序调试与开发、应用程序的移植与项目开发等。
ZigBee 通信节点	**开发环境：**基于 IAR for 8051 **协议:** ZigBee pro 协议(Z-Stack2007 协议栈) **功能：**自动组网、自动路由、无线数据传输等
IPv6 通信节点	**操作系统：**Contiki 2.5 **协议：**基于 Contiki OS 在 802.15.4 平台上实现完整的 IPv6 协议(Contiki OS uIPv6 协议栈) **功能：**自动组网、自动路由、无线数据传输等
蓝牙通信节点	**协议：**完整的蓝牙通信 4.0 协议 **功能：**蓝牙模块组网、SPP 蓝牙串行服务、无线数据传输等。
WIFI 通信节点	**网络类型：**Station/AP 模式 **安全机制：**WEP/WAP-PSK/WAP2-PSK/WAPI **加密类型：**WEP64/WEP128/TKIP/AES **工作模式：**透明传输模式，协议传输模式 **串口命令：**AT+命令结构 **网络协议：**TCP/UDP/ARP/ICMP/DHCP/DNS/HTTP **最大 TCP 连接数：**32 **功能：**自动组网、支持 AP 模式/AT 命令、无线数据传输等
RFID 阅读器	**功能：**支持与节点通信、组网，支持快速 CRYPTO1 加密算法、IC 卡识别、IC 卡读写

传感器模块	功能：基于 IAR for STM8 的开发环境，实现传感器数据采集与串口协议通讯

表 1.1.4.1　软件资源

1.5 平台的连线使用

1、电源

平台使用 5V 直流电源适配器进行供电，切勿插错电源！以免损坏器件。主板上的 Cortex-A53 终端和无线通讯模块接有单独的电源开关，使用时，请有选择的打开电源。另外主板上的无线通讯模块可以单独使用平台配套锂电池供电。

注意：主板电源与锂电池不可同时使用。

2、连线

Cortex-A53 智能终端在进行实验时，通常需要连接平台配套串口线和网线。串口线连接至 Cortex-A53 默认串口 0 接口（RS232）。

无线通讯模块在进行实验时，根节点（LCD 下方的模块）可以使用主板上的 RS232 口，子节点（LCD 左侧 12 个模块）和传感器模块默认无法使用 RS232 串口，需要时可以拔下子节点或传感器模块，通过平台配套 USB2UART 模块连接至 PC 机转接出来串口使用，此时采用 USB 线供电。

3、启动与调试

Cortex-A53 终端的启动可以支持从 SD 卡和 FLASH 两种方式，可自动选择启动方式。

连接平台配套 5V 电源适配器，启动 Cortex-A53 智能终端电源开关，我们可以通过平台配套的串口线与 PC 机连接。PC 端需要安装相应串口终端软件，这种软件有很多，介绍通过 Xshell 软件与平台 ARM Linux 系统连接。（在光盘 Tools 目录下，选择 Xshell 进行安装，安装过程中选择“免费为学校/教育”）

打开安装完成的 Xshell 软件：

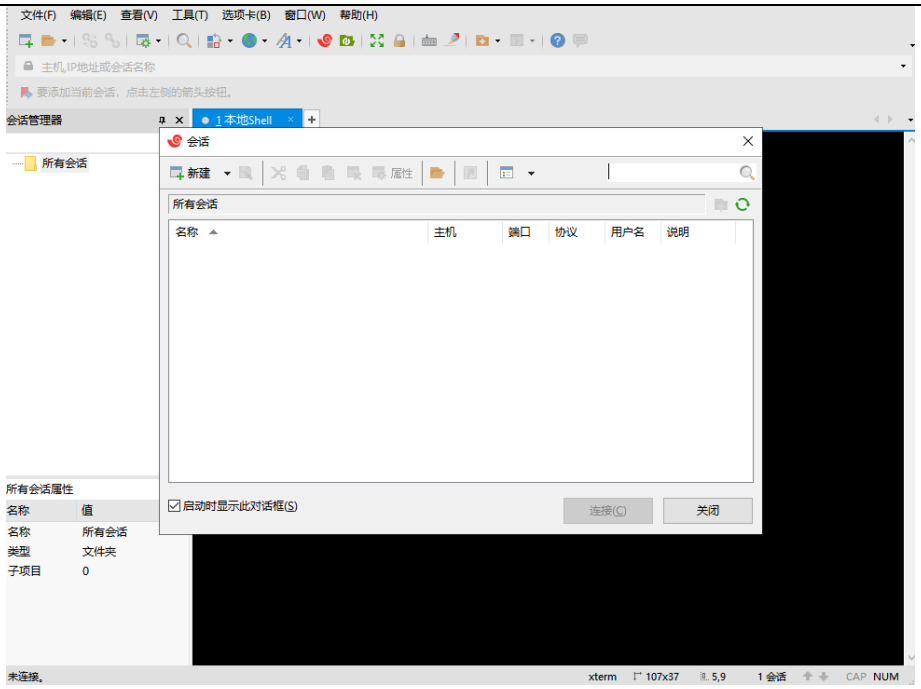


图 1.1.5.1　Xhell

在弹出的位置对话框中点击新增按钮。

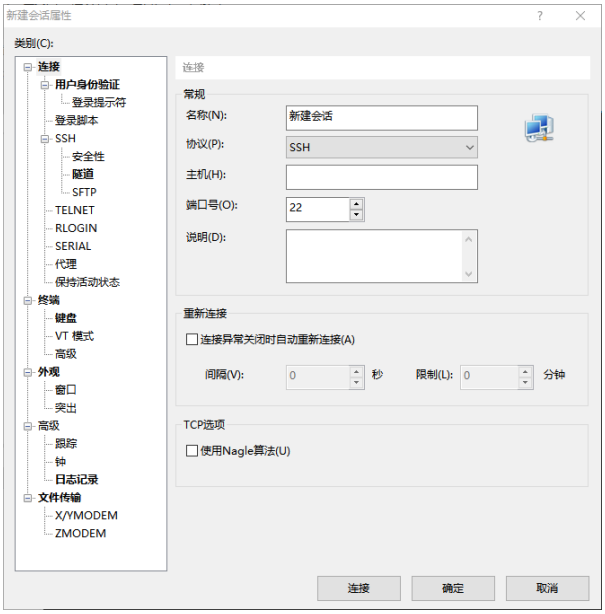


图 1.1.5.2　超级终端 电话选项

选择协议为“SERIAL”。

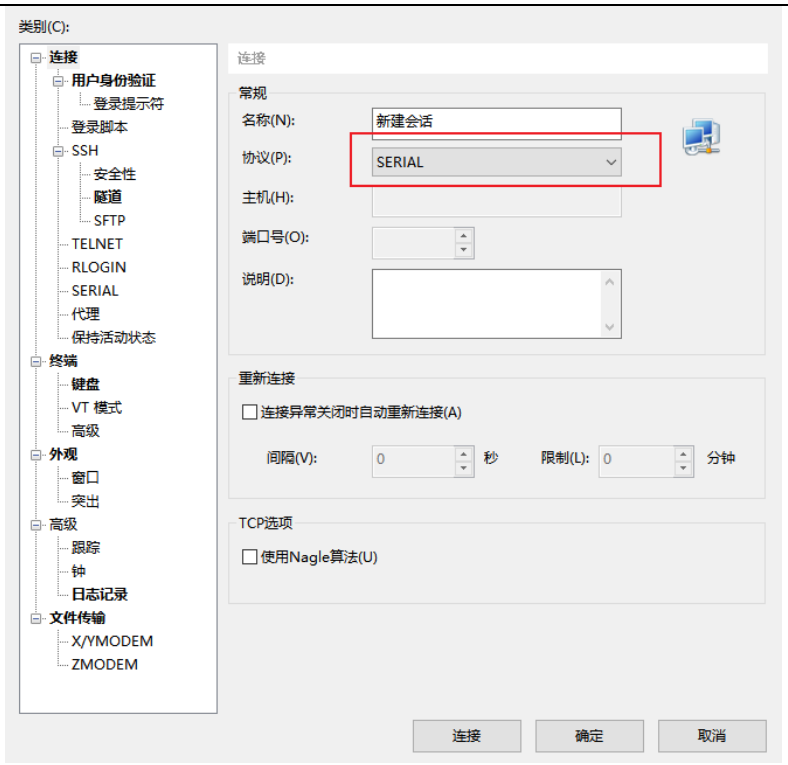


图 1.1.5.3　连接协议

选择左侧的“SERIAL”配置功能，根据 PC 机串口设备号，选择 COM1，设置串口属性：波特率 115200，数据位 8，奇偶校验 无，停止位 1，流控制 无，选择应用，选择确定。

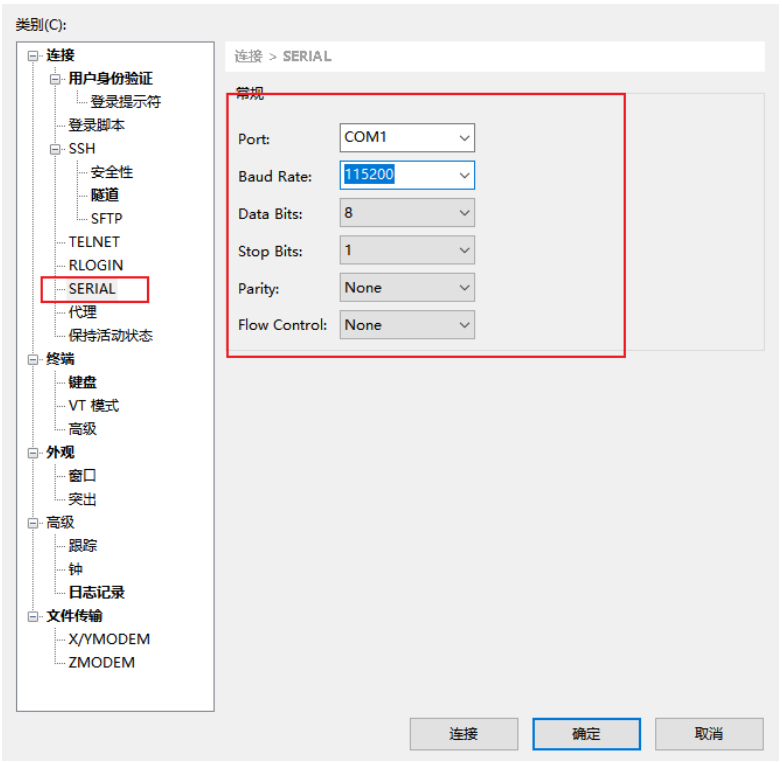


图 1.1.5.4　串口配置

双击会话中新增的条目。

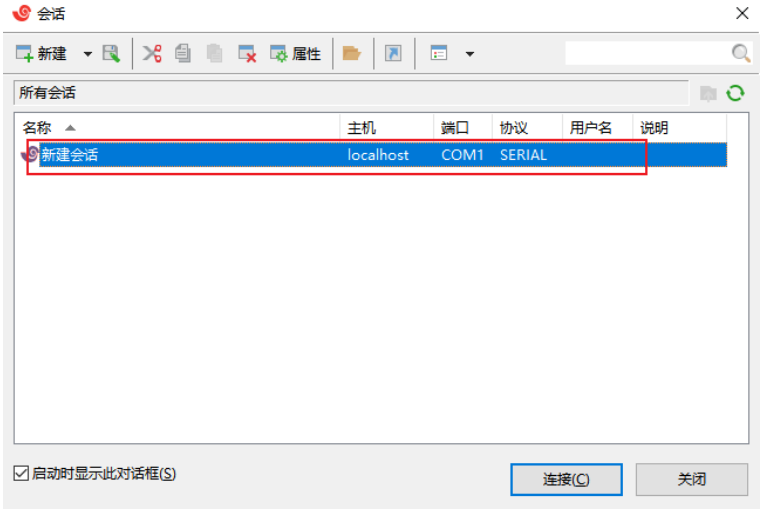


图 1.1.5.5　连接串口

之后便可以重启 ICS-IOT-CEP Cortex-A53 智能终端系统，在超级终端中登录系统了。

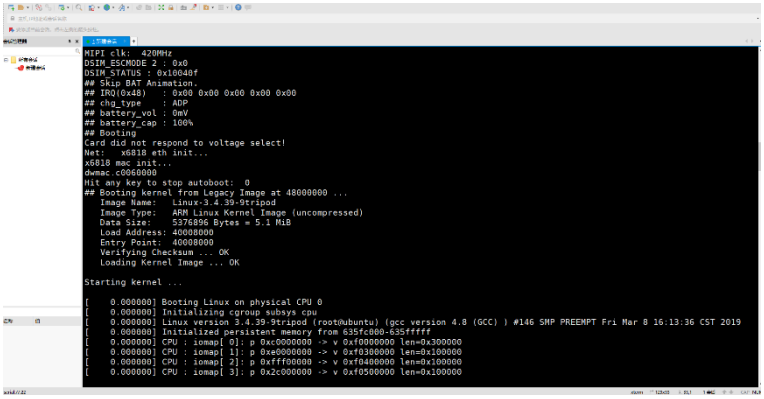


图 1.1.5.6　启动信息

主板上有 2 组共 4 个 20P 标准 JTAG 接口（主板标记为 JLink 与 STLink），其中 1 组（P20/P21）用于根节点模块的下载和调试，一组（P14/P15）用于子节点和传感器模块的下载和调试：

1）J-OB 仿真器和 J-OB 转接板接口用于连接 J-OB 仿真器和 J-OB 转接板或者 ZigBee Debuger 仿真器完成对无线通讯模块的程序下载和调试。

2）ST-Link 接口用于连接 ST-Link 仿真器完成传感器模块的程序下载和调试。

由于多个模块复用 JTAG 调试口，上述目标模块调试的选定，可以通过主板上的 2 个选择按键进行目标模块的选择（有指示灯提示）。

4、UART 串口的使用

平台默认引出 4 路 RS232 串口，其中 Debug UART 为根节点调试使用串口，其余 3 个 UART 为 Cortex-A53 终端扩展串口，用于预留扩展使用其他模块。

关于根节点串口的连接选择是复用的，可以使用相应根节点下方的 4 位拨码开关 1234，其中 123 位分别对应根节点与 Cortex-A53 相应串口的连接，4 用于根节点与 Debug UART 串口的连接。同一时间，只能有一个根节点使用 Debug UART 串口，如果串口使用出错，相应的 UART ERROR 串口指示灯会点亮。

如果要使用 Cortex-A53 终端平台的 RS232 串口连接外扩模块，需要使用主板右下角 UART-1 RS232 串口一侧的 3 位拨码开进行相应选择，同时关闭根节点上方 4 位拨码开关的设置。拨码开关（向上为打开，向下为关闭）。

5、其他

1）USB2UART 模块用于平台配套无线模块和传感器模块的串口调试工作，需要安装平台光盘提供的驱动方可正常使用。

2）锂电池模块非标配），配有配套的电源适配器，可以通过平台配套的锂电池充电模块进行充电。

3）更详细的硬件说明，请参考平台配套硬件说明书。

2. Windows 系统开发环境

2.1 ZigBee 模块

ICS-IOT-CEP 全功能物联网平台，ZigBee 模块（TI 方案）采用 TI 的 CC2530 处理器，其上位机 windows 开发环境使用的是嵌入式集成开发环境 IAR EWARM。该开发环境针对目标处理器集成了良好的函数库和工具支持。

- **软件安装准备工作**

1）嵌入式集成开发环境 EW8051-EV-8103-Web 安装包

2）ZStack-CC2530-2.3.0-1.4.0 ZigBee 协议栈安装包

3）Setup_SmartRFstudio_6.11.6.exe 仿真器驱动安装包

4）Setup_SmartRFProgr_1.6.2 烧写工具安装包（可不安装）

- **软件安装**

1.嵌入式集成开发环境 IAR EWARM 安装

1）打开 IAR 安装包进入安装界面，双击 EW8051-EV-8103-Web 进行安装：

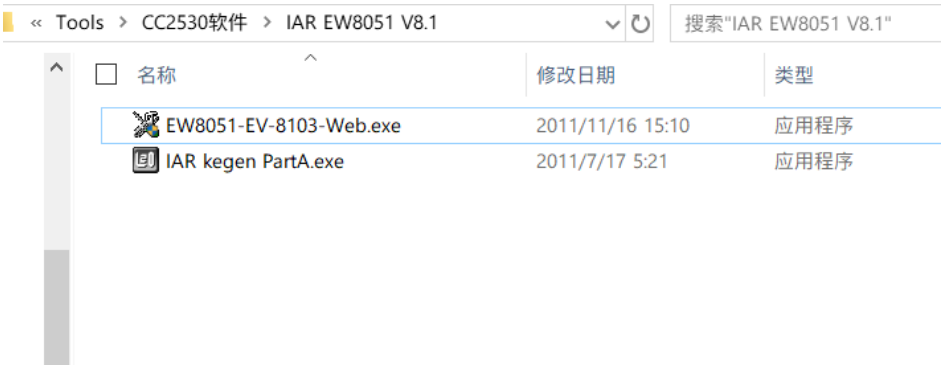


图 1.2.2.1　安装软件包

2）出现如下对话框，“next”：

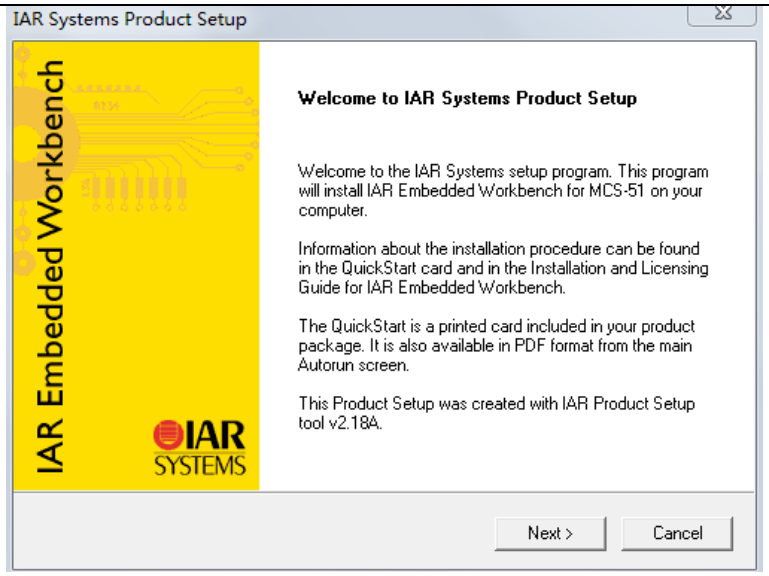


图 1.2.2.2

3）接受许可协议：

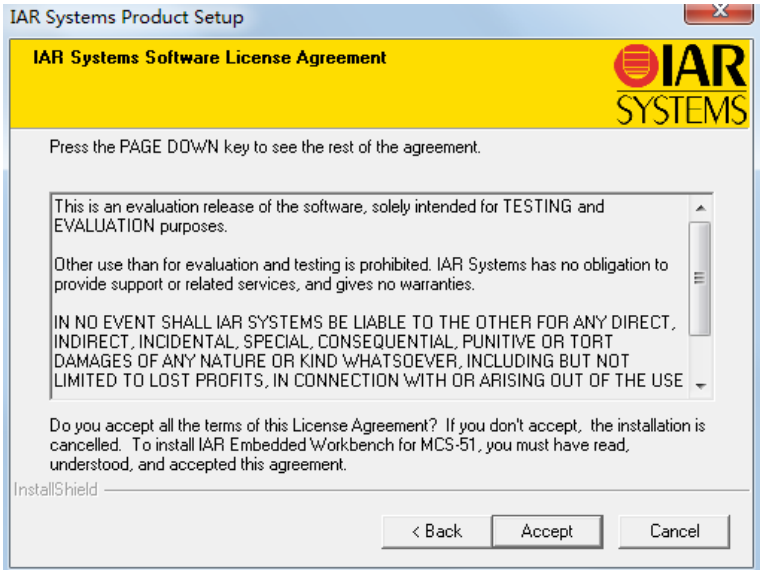


图 1.2.2.3 接受许可协议

4）输入正确的序列号和 KEY，输入到如下对话框中：

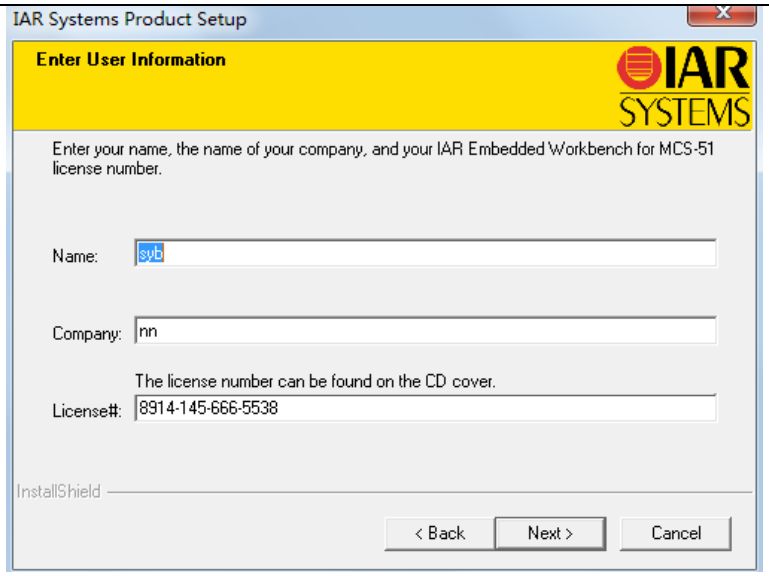


图 1.2.2.4 输入序列号

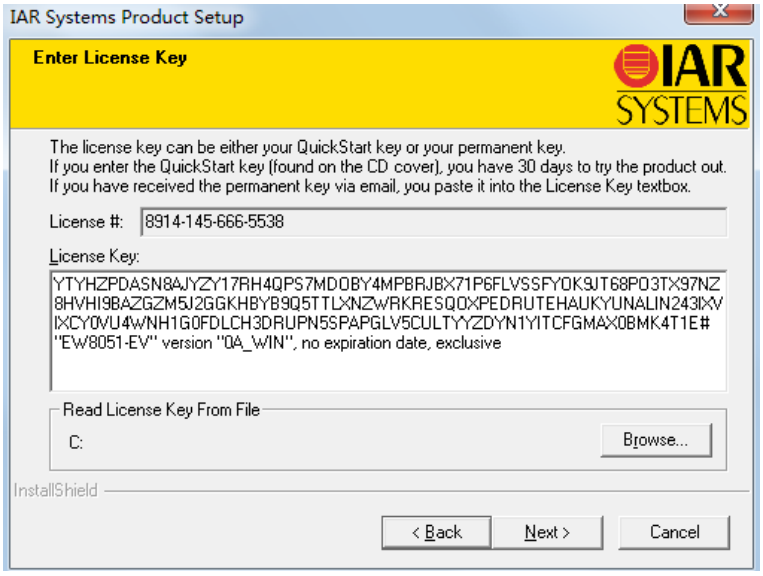


图 1.2.2.5 输入 KEY

5）设置安装路径，next：

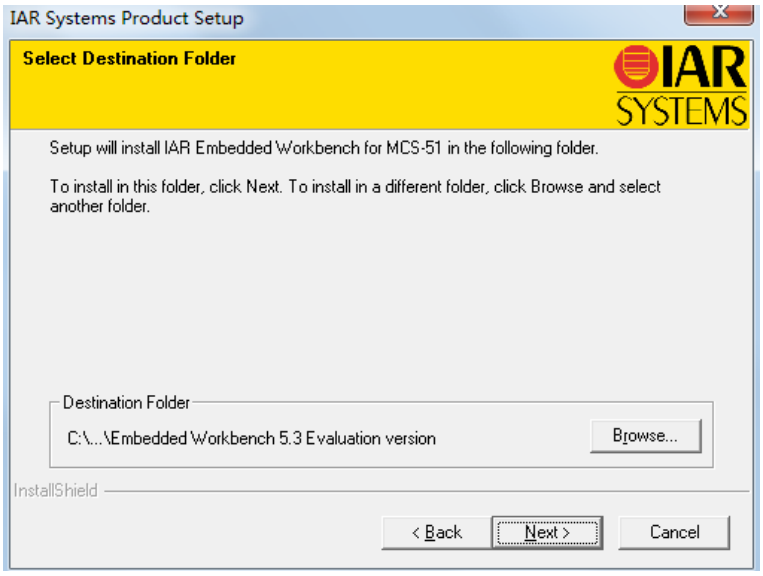


图 1.2.2.6 安装路径

6）选择完全安装，next：

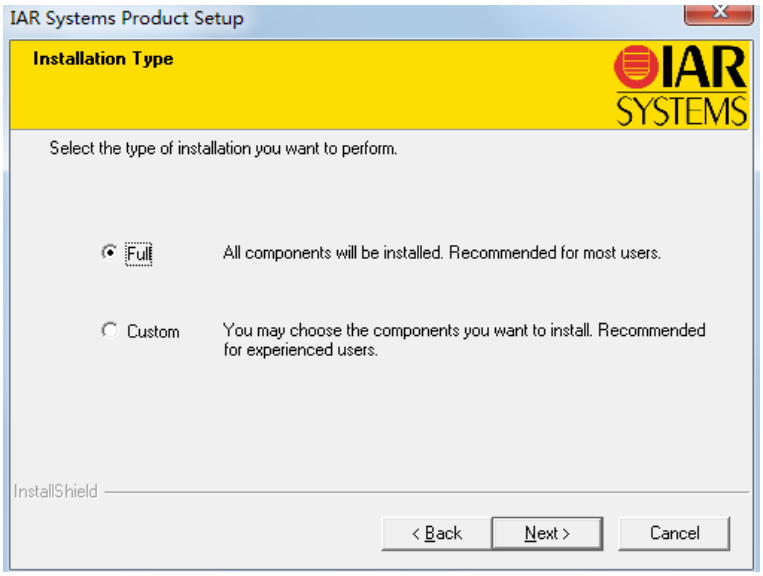


图 1.2.2.7 安装模式

7）next：

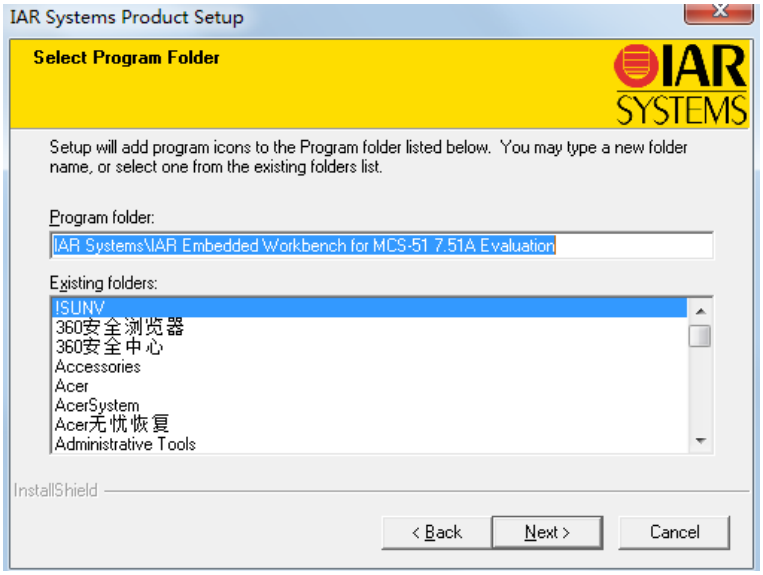


图 1.2.2.8

8）next：

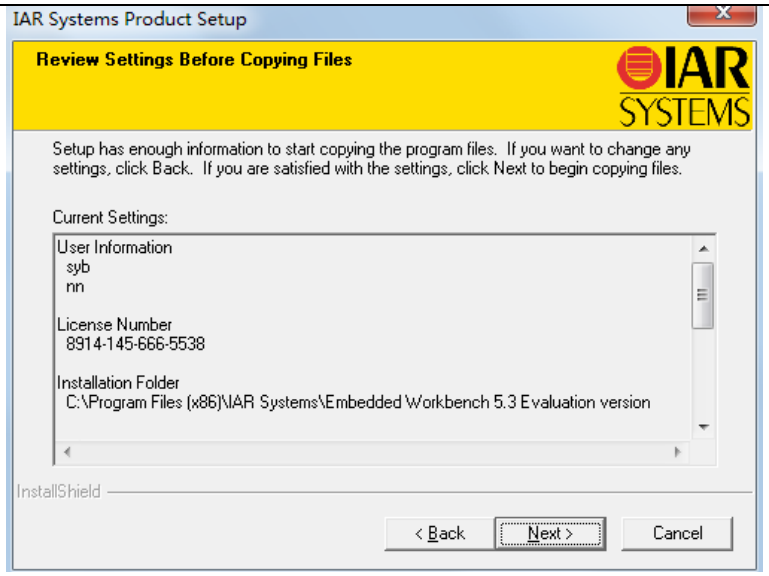


图 1.2.2.9

9）开始安装：

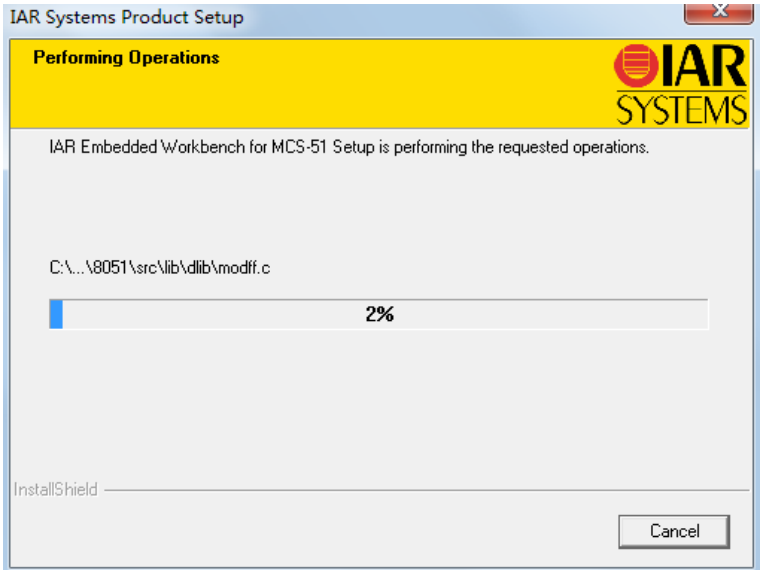


图 1.2.2.10 安装进度

10）安装完成。

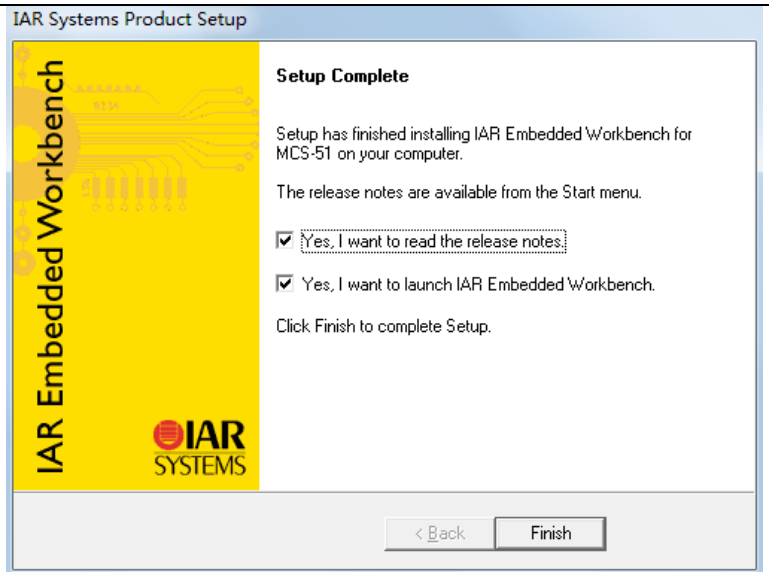


图 1.2.2.11 安装完成

2.ZStack-CC2530-2.3.0-1.4.0 ZigBee 协议栈安装过程

1）双击 ZStack-CC2530-2.3.0-1.4.0 开始安装：

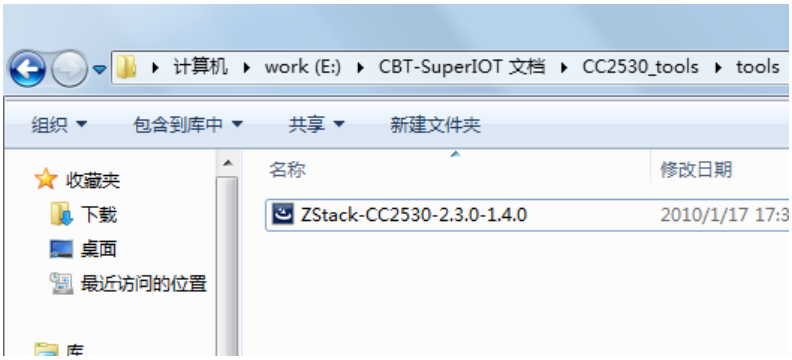


图 1.2.2.12 协议栈安装包

2）选择 Modify，next：

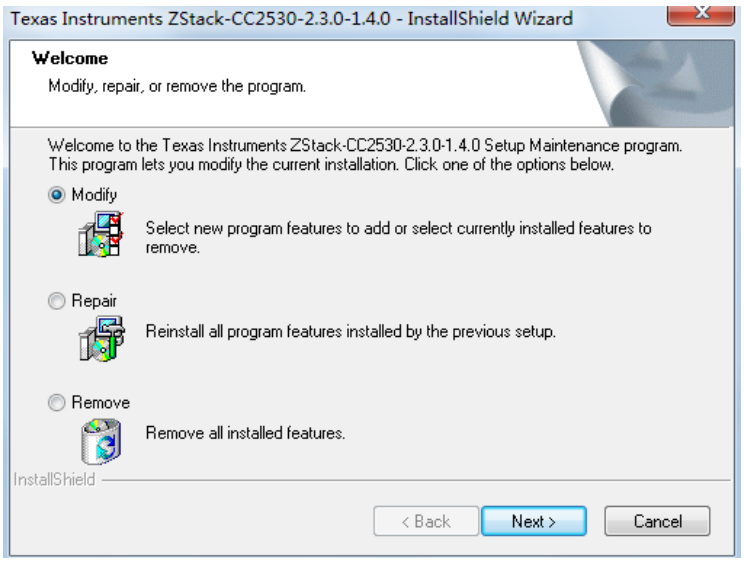


图 1.2.2.13 选择典型安装

3）选择安装支持的工具：

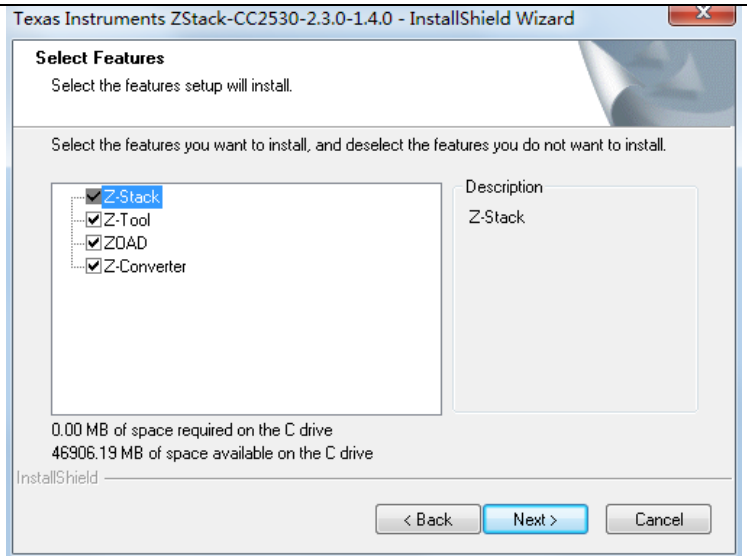


图 1.2.2.14 工具

4）如下开始安装：

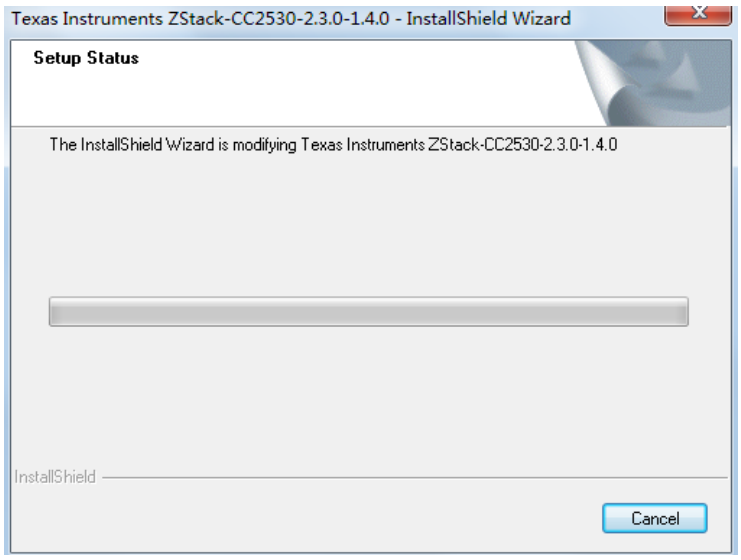


图 1.2.2.15 安装进度

5）安装完成：

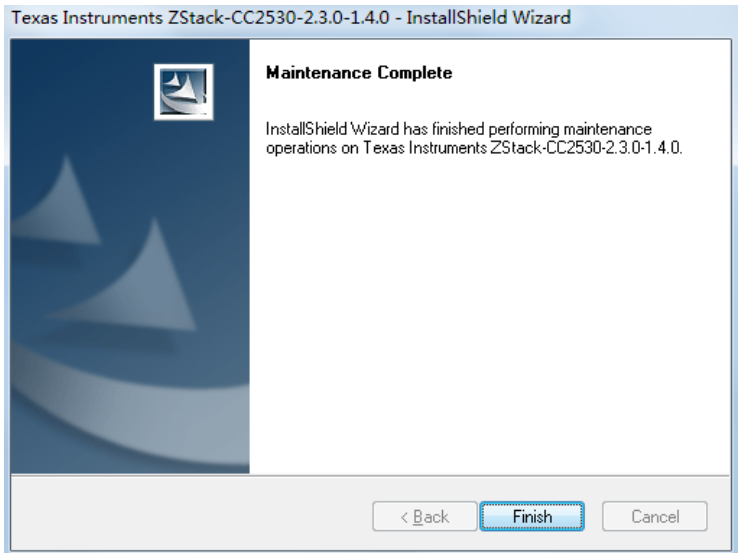


图 1.2.2.16 安装完成

3.Setup_SmartRFstudio_6.11.6.exe 仿真器驱动安装过程

1）双击 Setup_SmartRF_Studio_6.11.6 开始安装，next：

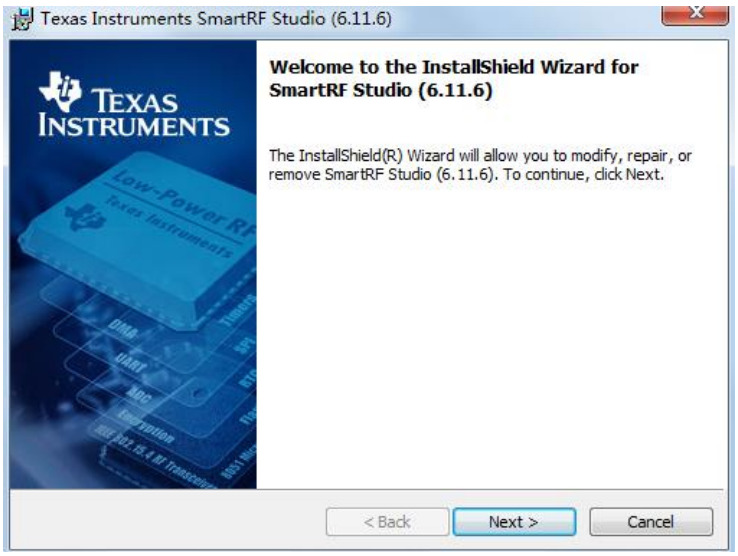


图 1.2.2.17

2）next：

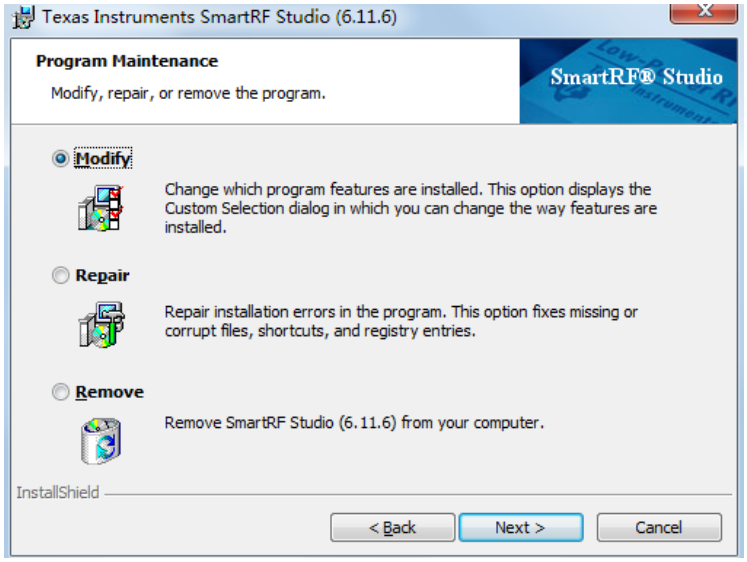


图 1.2.2.18 安装模式

3）next：

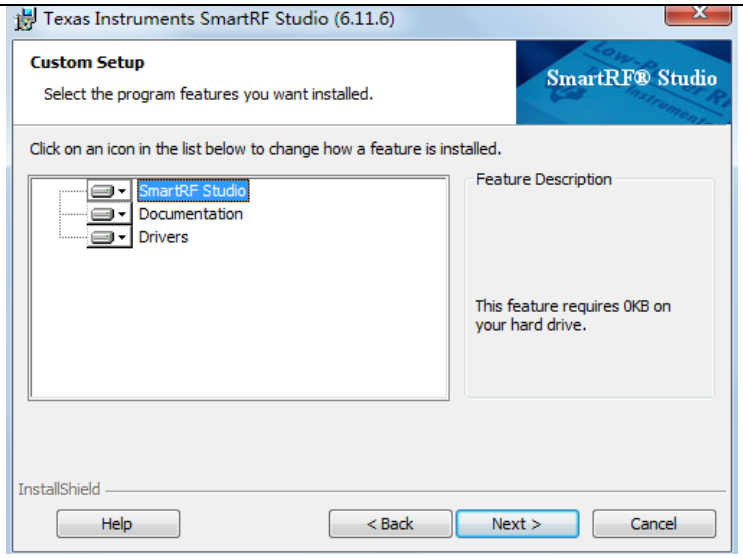


图 1.2.2.19 安装选项

4）Install：

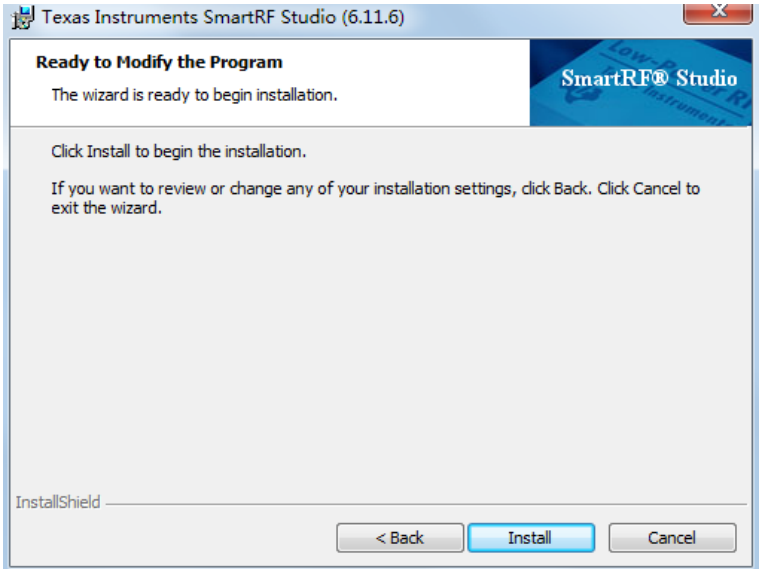


图 1.2.2.20 开始安装

5）安装完成：

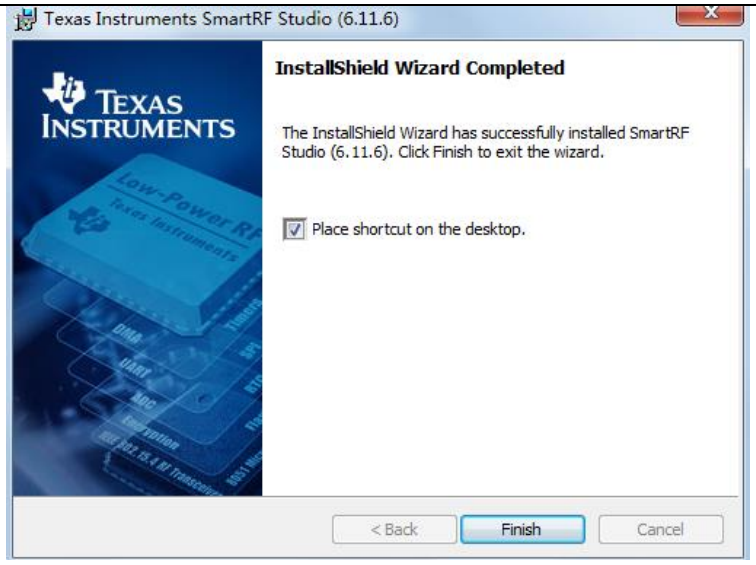


图 1.2.2.21 安装完成

6）将仿真器通过开发系统附带的 USB 电缆连接到 PC 机，在 Windows XP 系统下，系统找到新硬件后提示如下对话框，选择 自动安装软件 ，点下一步。

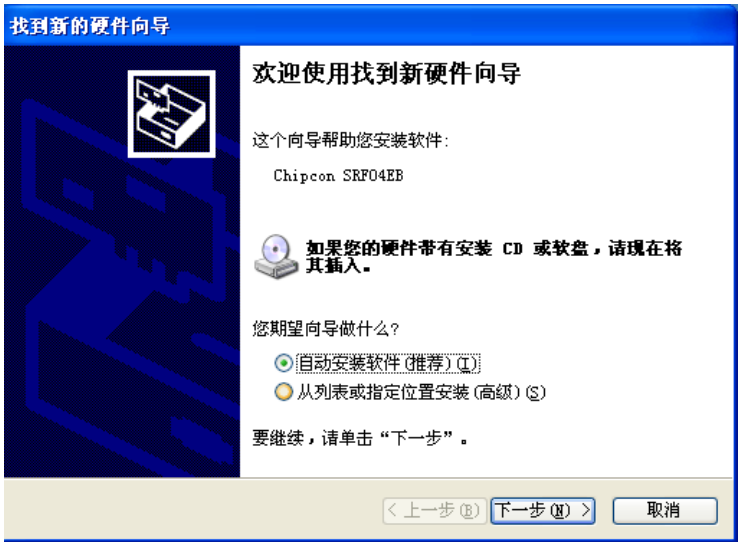


图 1.2.2.22 自动安装

7）向导会自动搜索并复制驱动文件到系统。系统安装完驱动后提示完成对话框，点击完成 退出安装。

图 1.2.2.23 安装完成

4.Setup_SmartRFProgr_1.6.2 烧写工具安装过程

1）双击 Setup_SmartRFProgr_1.6.2 安装，next：

图 1.2.2.24 烧写工具安装

2）next：

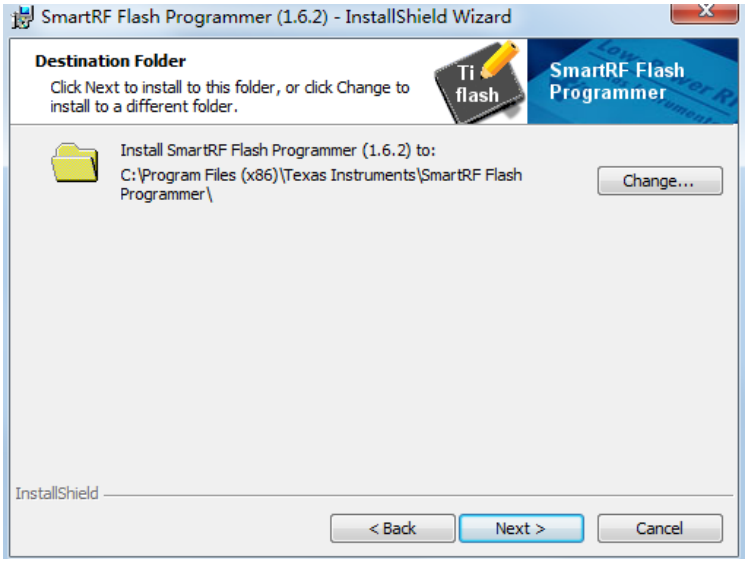


图 1.2.2.25 安装路径

3）选择完整安装，next：

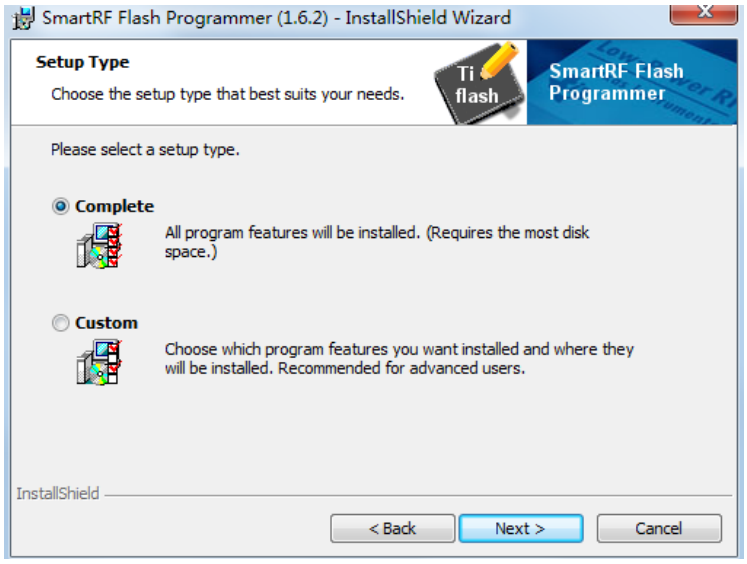


图 1.2.2.26 完全安装

4）Install：

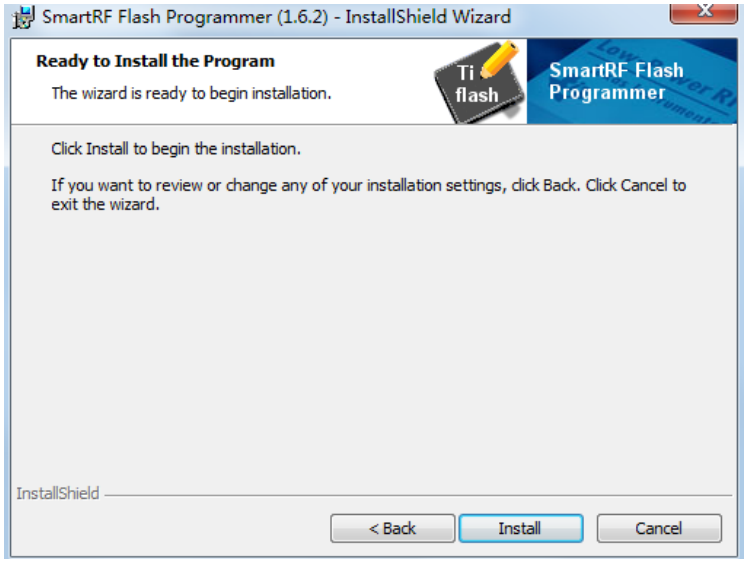


图 1.2.2.27

5）安装过程如下：

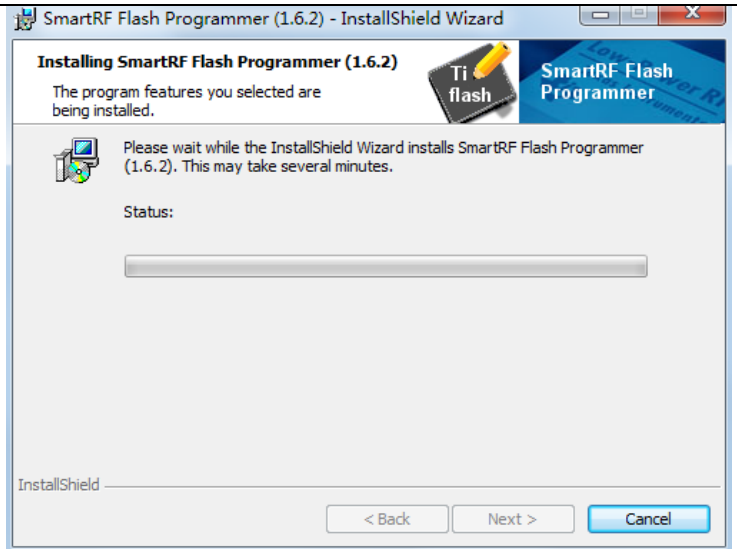


图 1.2.2.28 安装进度

6）安装完成：

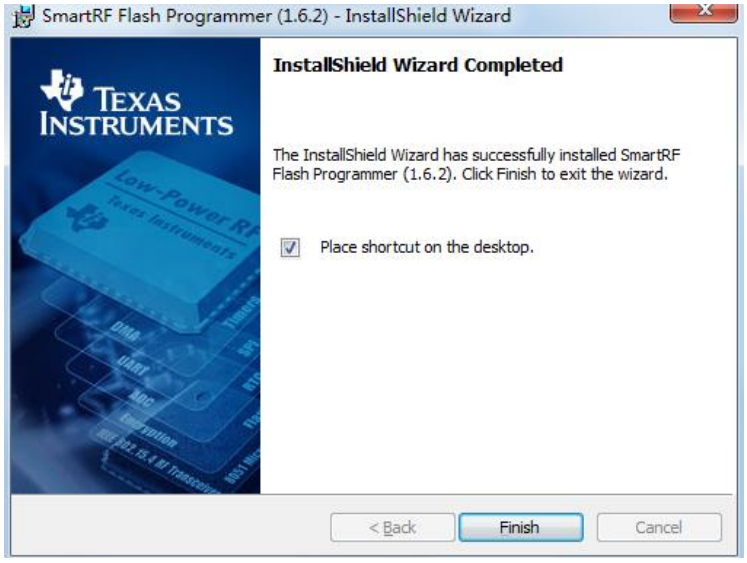


图 1.2.2.29 安装完成

2.2 IPv6 模块

ICS-IOT-CEP 全功能物联网平台，IPv6 模块采用 32 位高性能低功耗的 STM32W108 处理器，其上位机 windows 开发环境使用的是嵌入式集成开发环境 IAR EWARM 和 Cygwin Linux 环境。该开发环境使用 Cygwin 开发环境与 IAR(编译器)工具配合完成 IPv6 模块的程序编译与下载。

- **软件安装准备工作**

1）嵌入式集成开发环境 IAR EWARM 5.41 安装包

2）J-LINK 驱动程序安装包

3）cygwin-2011-11-2.isz 光盘镜像，Daemon tools 虚拟光驱

◆ 软件安装

1.嵌入式集成开发环境 IAR EWARM 安装

该步骤具体参见上一节 1.2.1 内容，这里不再赘述

2.Cygwin 环境安装

1）使用 Daemon tools 虚拟光驱打开 cygwin-2011-11-2.isz 光盘镜像开始安装：

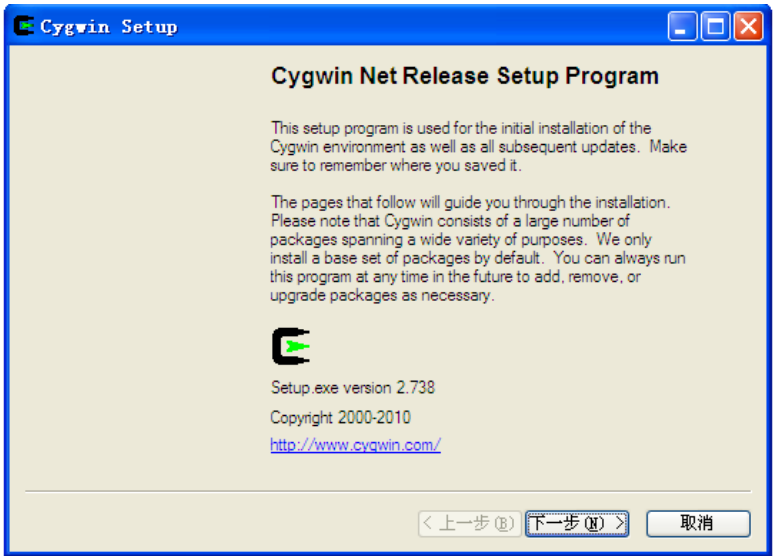


图 1.2.3.1　打开安装包

2）选择下一步安装，选择 Install from Local Directory：

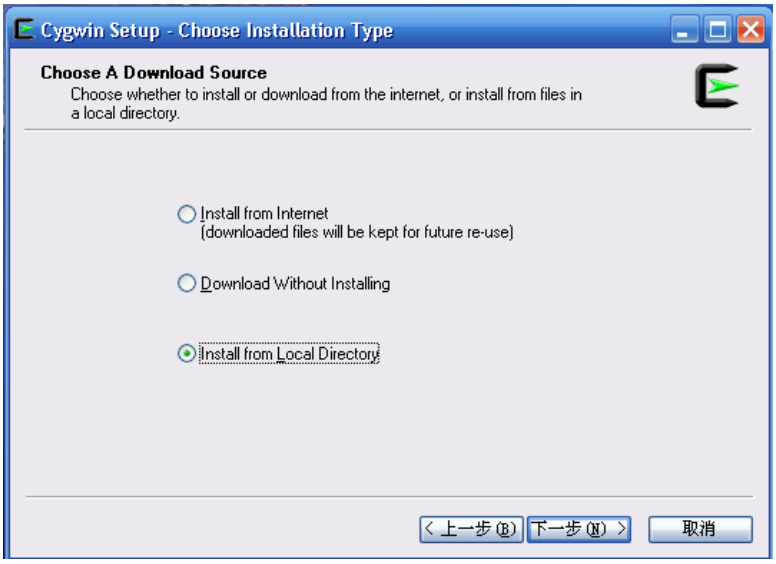


图 1.2.3.2　从本地安装

3）安装 root 目录选择 默认 c 盘 cygwin

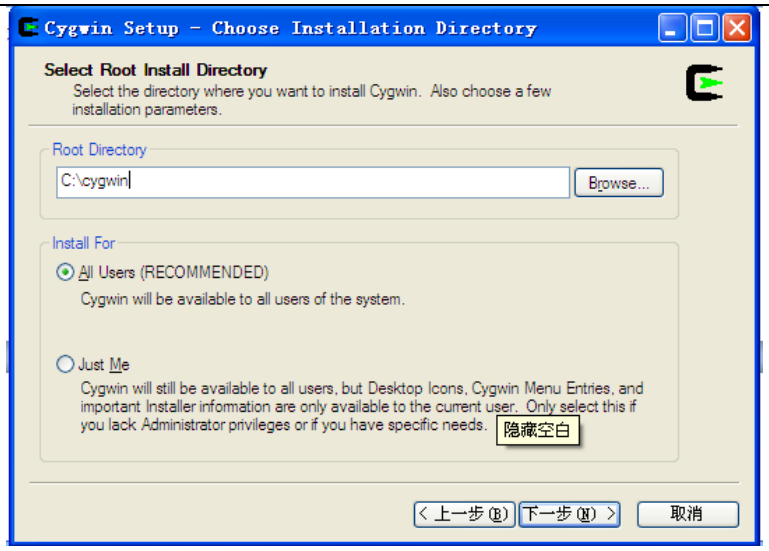


图 1.2.3.3　安装目录

4）选择本地安装包，根据实际情况，指定为虚拟光驱盘符中的 H:\release 目录：

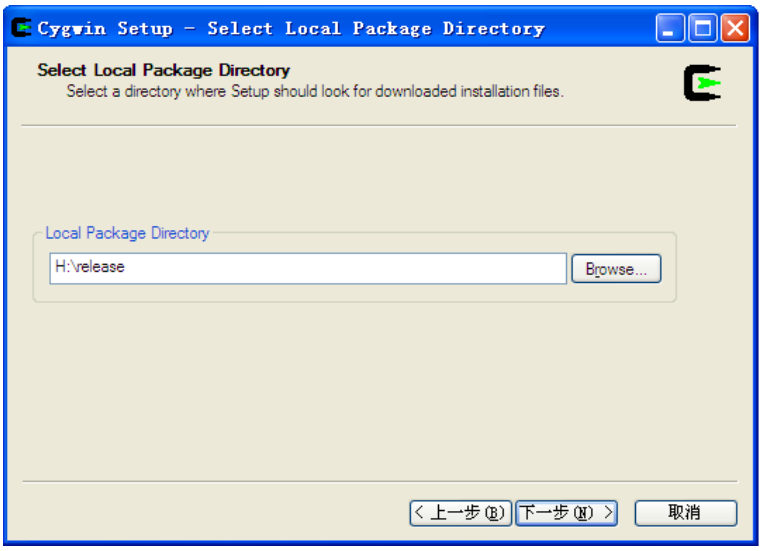


图 1.2.3.4　选择光驱 release 目录

注意，该目录不要选错，要选择虚拟光驱中的 release 目录，才可以正确安装。

5）选择安装包，可以根据需要进行选择，默认即可。

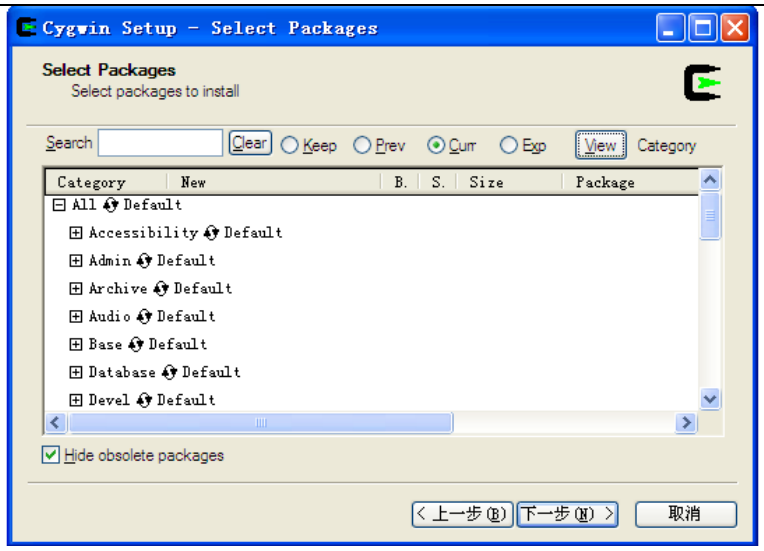


图 1.2.3.5　选择 packages

6）开始安装，本环节比较大，需要较长时间和较大磁盘空间(约 4G)来进行安装。

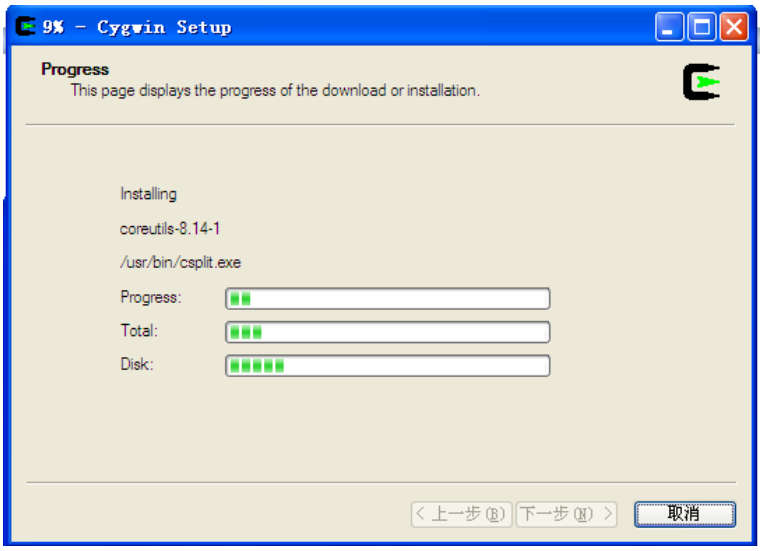


图 1.2.3.6　开始安装

7）安装完毕。默认会在桌面创建 Singular 和 cygwin 快捷方式。

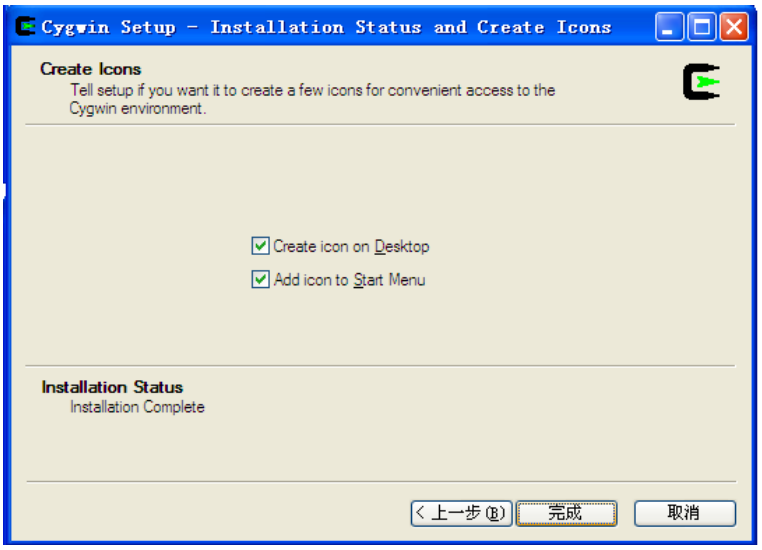


图 1.2.3.7　安装完成

2.3 Bluetooth/WiFi 模块

ICS-IOT-CEP 全功能物联网平台，Bluetooth 模块与 WiFi 模块采用相同的 32 位高性能低功耗的 STM32F103 处理器，其上位机 windows 开发环境使用的是 IAR 集成开发环境。该开发环境针对目标处理器集成了良好的函数库和工具支持。

- **软件安装准备工作**

1）嵌入式集成开发环境 IAR EWARM 5.41 安装包

2）J-LINK 驱动程序安装包

- **软件安装**

1. 嵌入式集成开发环境 IAR EWARM 安装

1）打开 IAR 安装包进入安装界面。

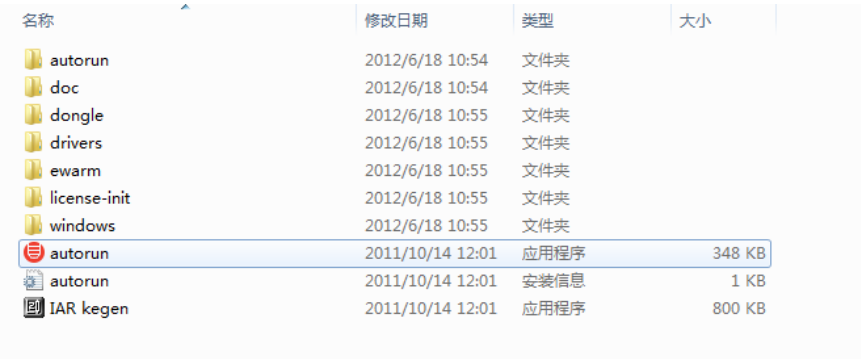


图 1.2.1.1　打开安装包

2）选择 Install Embedded Workbench 安装选项。

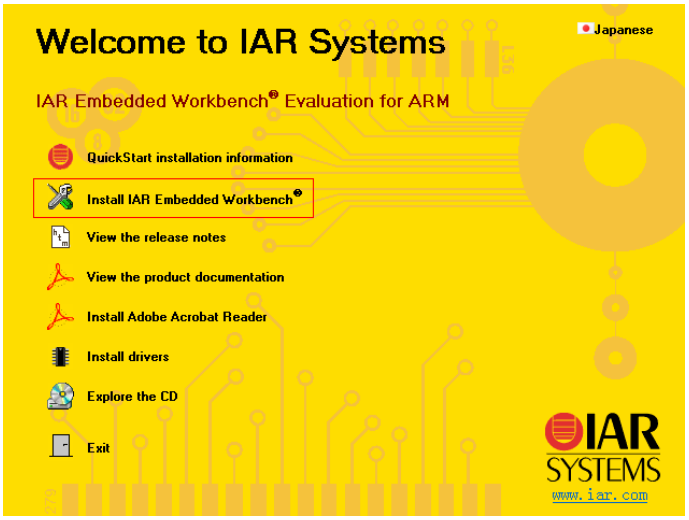


图 1.2.1.2　安装 IAR(1)

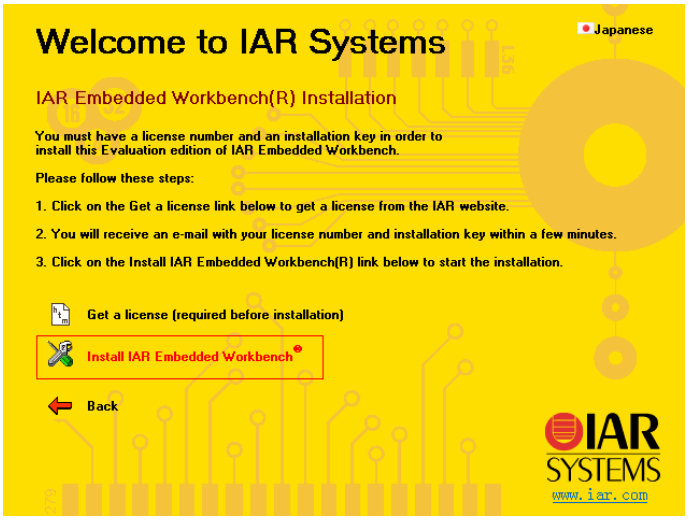


图 1.2.1.3　安装 IAR(2)

3）进入 IAR 安装过程，选择 Next。

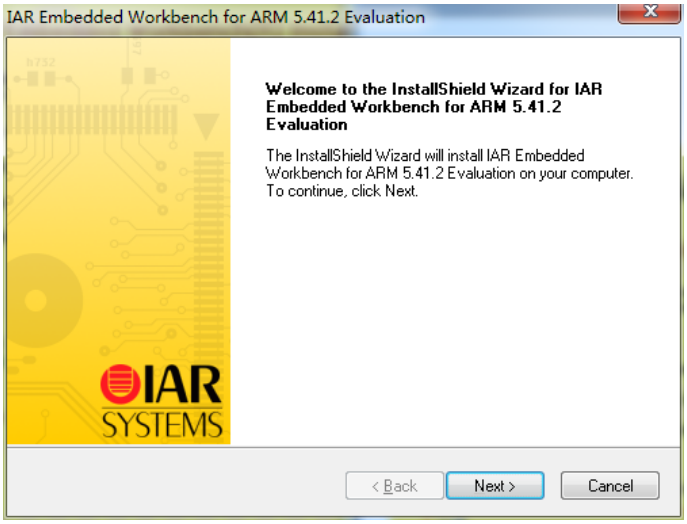


图 1.2.1.4　安装 IAR(3)

4）进入 License 号输入界面。

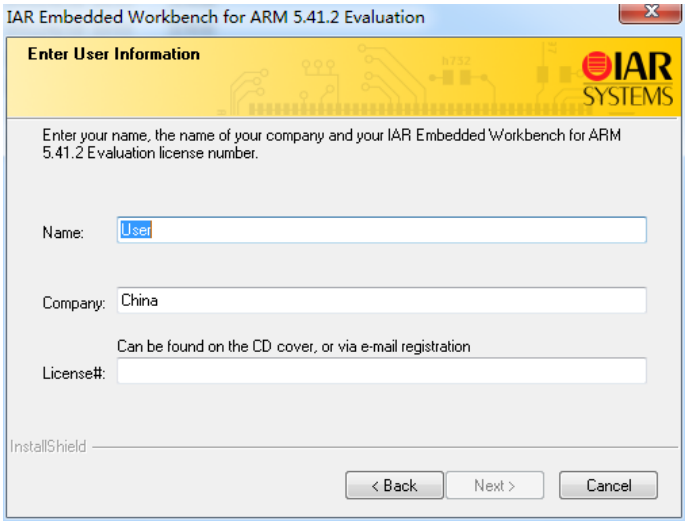


图 1.2.1.5　输入用户信息和 License

5）输入 KEY。

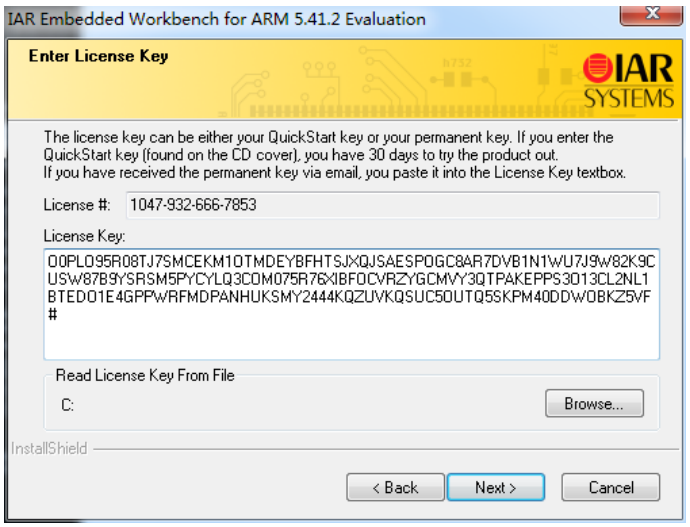


图 1.2.1.6　输入 KEY

6）选择 IAR 软件安装路径，这里选择默认的 C 盘 Program Files，建议默认安装。

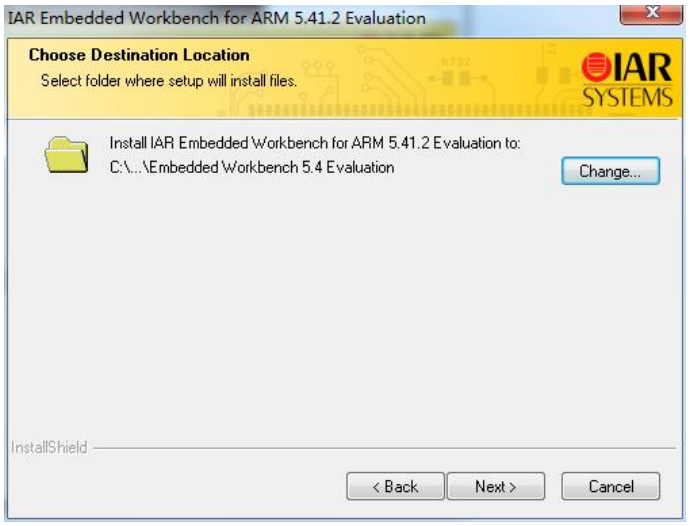


图 1.2.1.7　安装到 C 盘默认路径

7）进入安装过程界面。

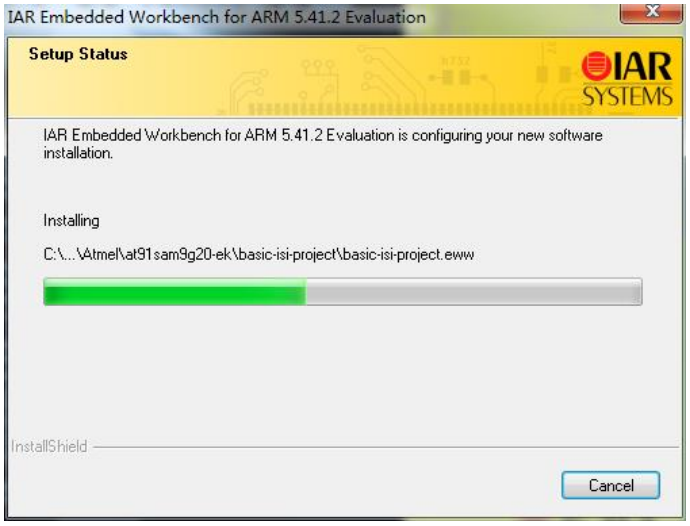


图 1.2.1.8　开始安装

9）安装完成。

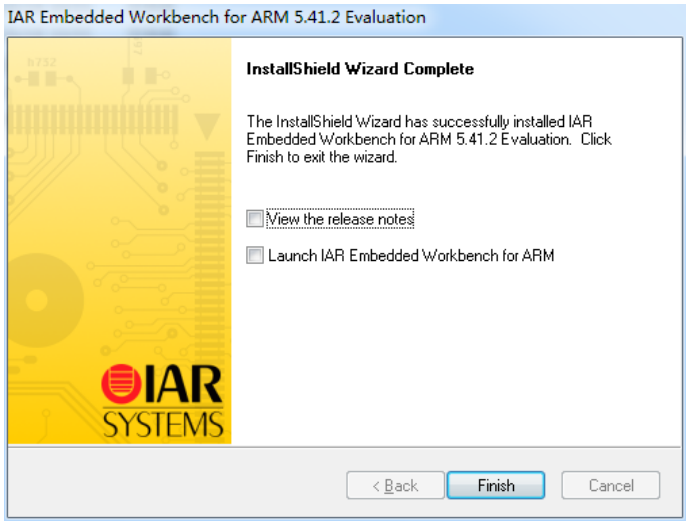


图 1.2.1.9　安装完成

2．Jlink 4.20 驱动程序安装过程

1）运行安装程序 Setup_JLinkARM_V440p 安装包，并选择 Yes。

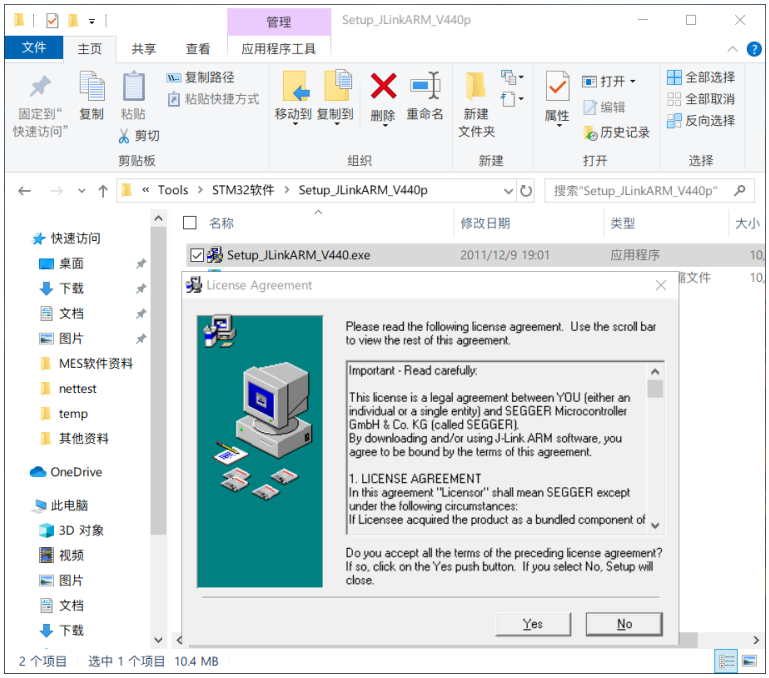


图 1.2.1.10　安装仿真器驱动

2）选择 Next，继续安装过程。

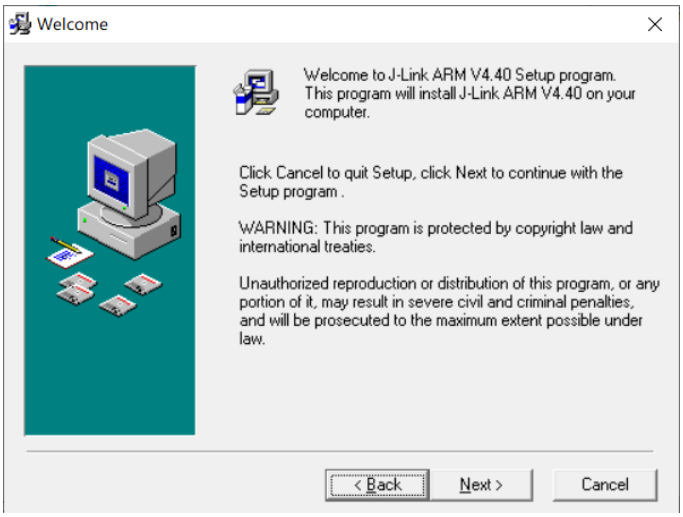


图 1.2.1.11　安装 下一步

3）选择驱动安装路径，点击 Next

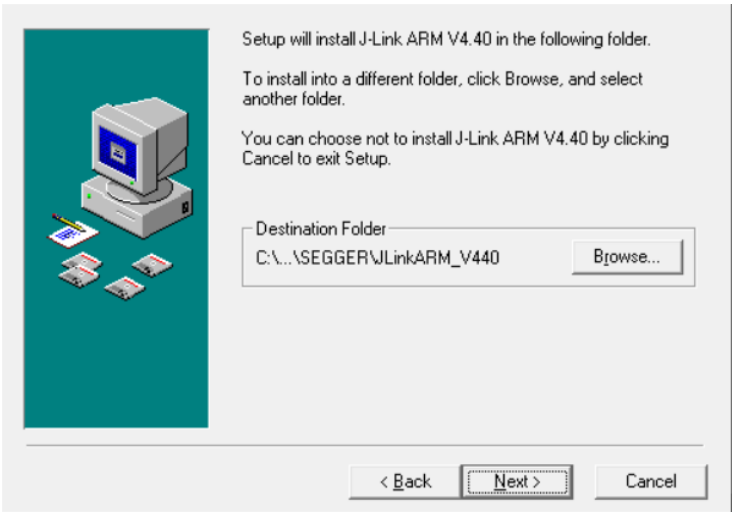


图 1.2.1.12　默认安装路径

4）选择在桌面创建快捷方式，点击 Next

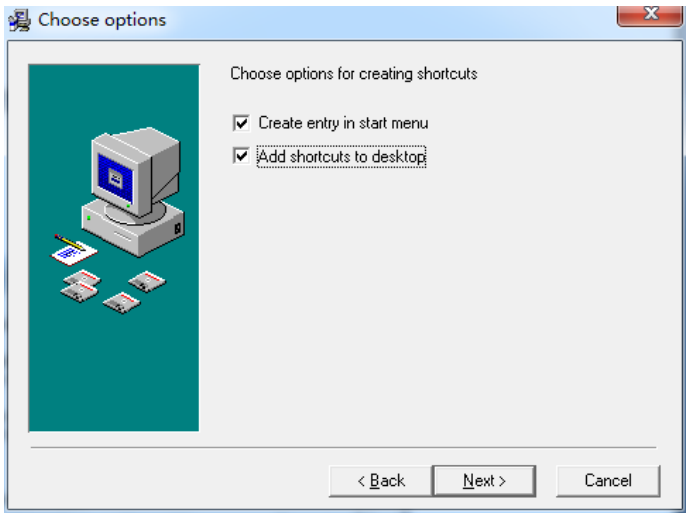


图 1.2.1.13　创建图标

5）进入安装状态

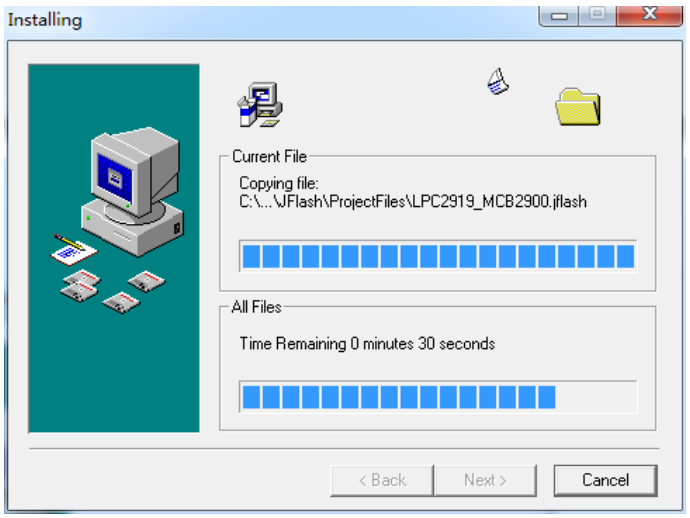


图 1.2.1.14　开始安装

6）进入 SEGGER J-OB 仿真器和 J-OB 转接板 DLL Updater V4.20p 界面，勾选相应的 IAR 版本，点击 Next。

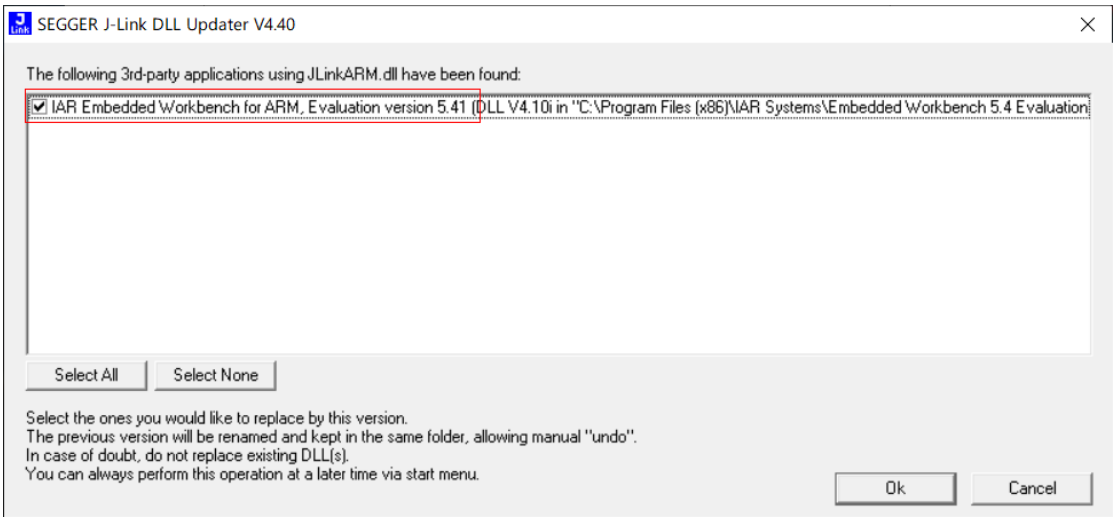


图 1.2.1.15　选择第三方环境支持

7）完成 Jlink 驱动程序安装

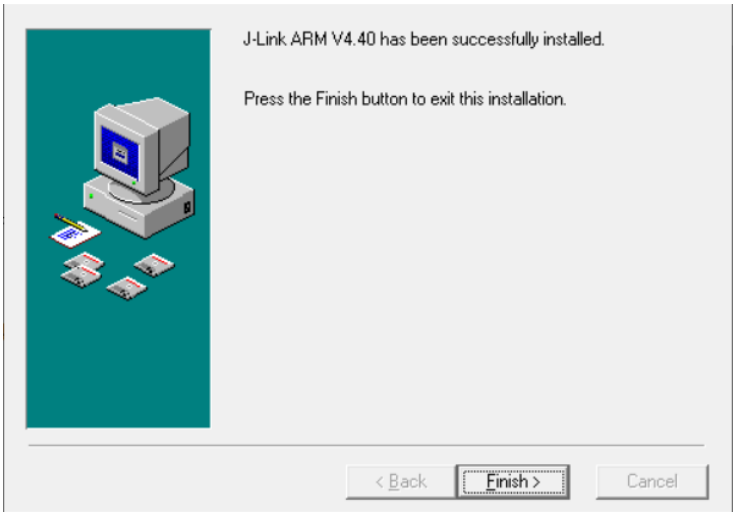


图 1.2.1.16　完成安装

2.4 Rfid/传感器模块

ICS-IOT-CEP 全功能物联网平台，Rfid 模块与传感器模块采用相同的 8 位 STM8S 处理器，其上位机 windows 开发环境使用的是 IAR SWSTM8 集成开发环境。该开发环境针对目标处理器集成了良好的函数库和工具支持。

- **软件安装准备工作**

1）嵌入式开发环境 IAR EWSTM8 1.30 软件安装包

◆ 软件安装

1）进入安装页面，点击 Install，开始安装

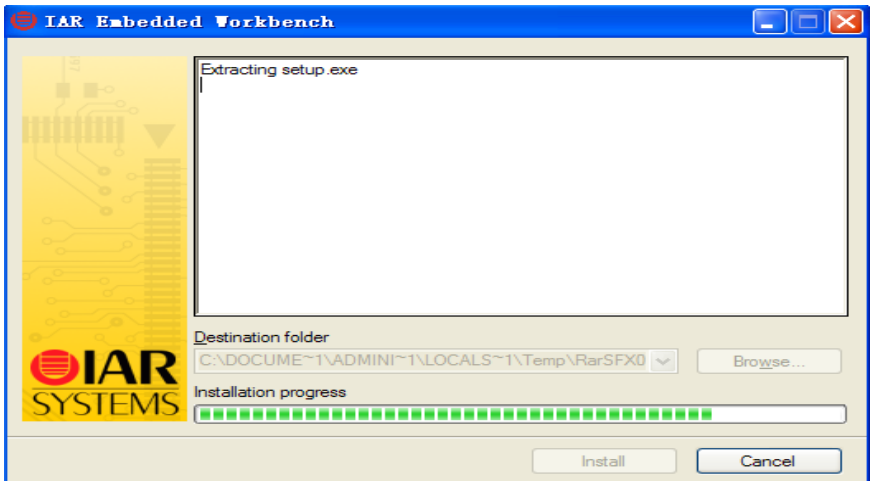


图 1.2.5.1　开始安装

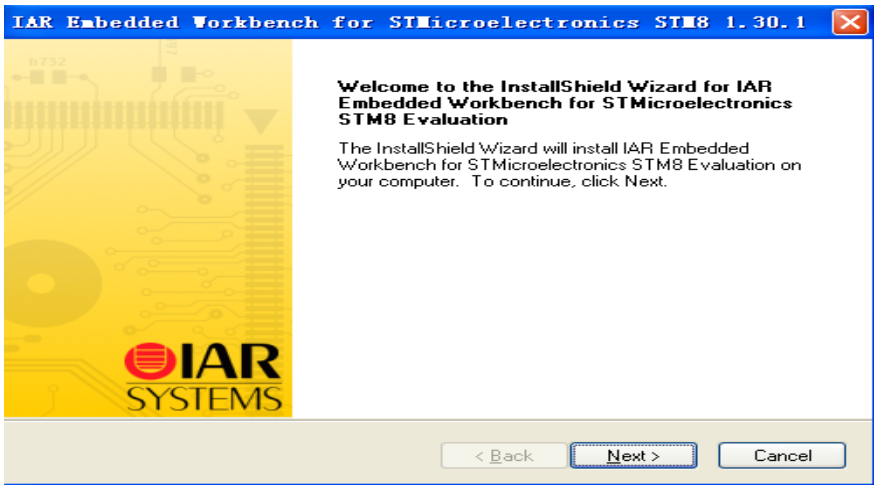


图 1.2.6.2　下一步安装

2）进入 License 号输入界面，输入 License 号，点击 Next

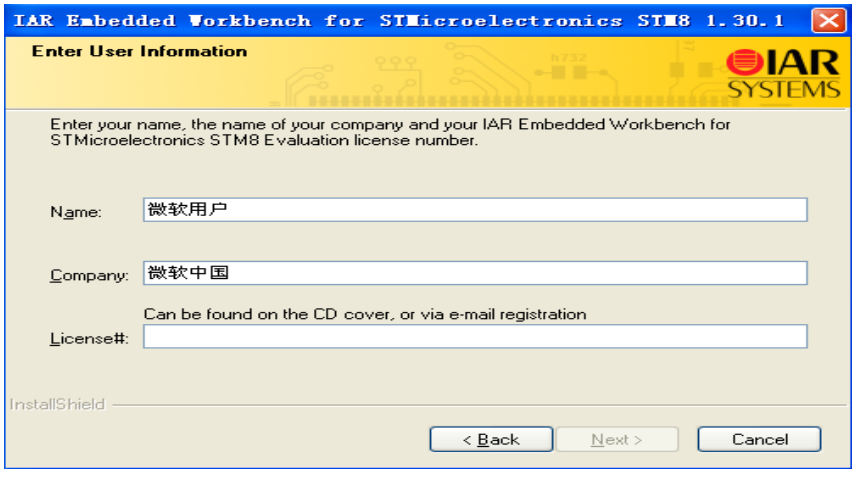


图 1.2.5.3　输入 license

3）输入 STM8 的 License number 和 Key，点击 Next

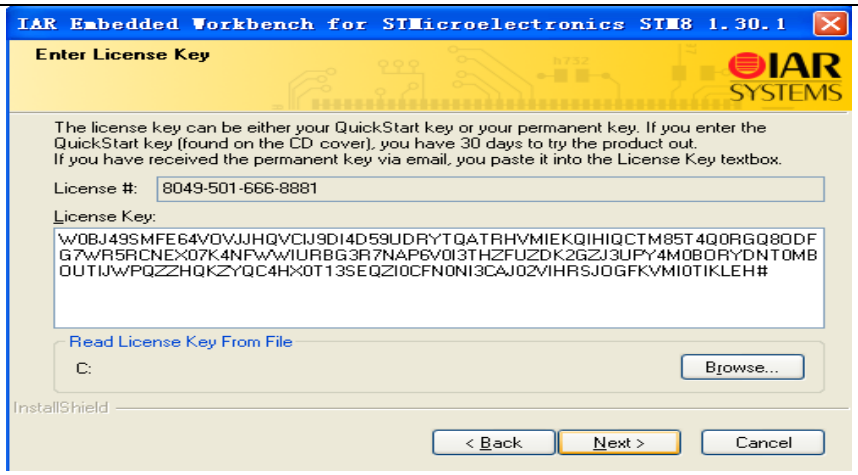


图 1.2.5.4　输入 KEY

4）选择 IAR 软件安装路径，点击 Next

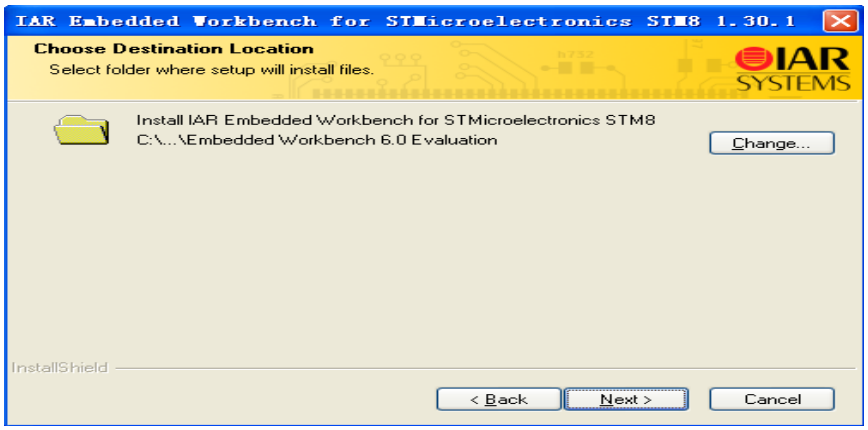


图 1.2.5.5　默认 C 盘安装路径

5）选择 install，开始安装

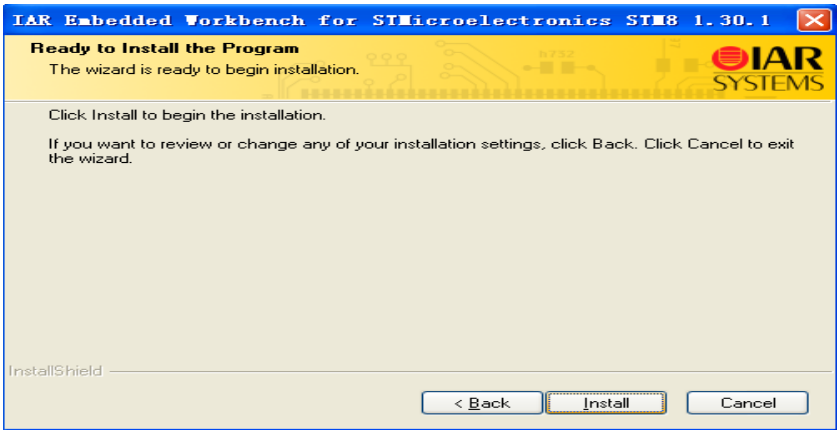


图 1.2.5.6　开始安装

6）进入安装过程

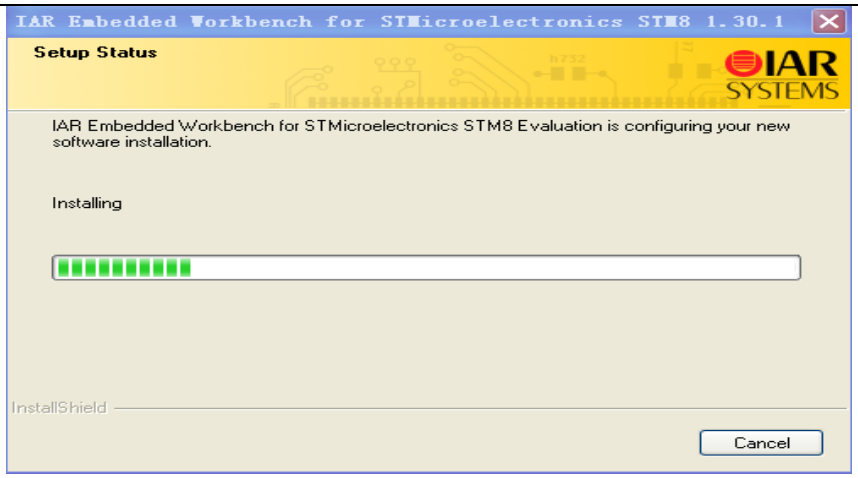


图 1.2.5.7　安装中

7）安装完成

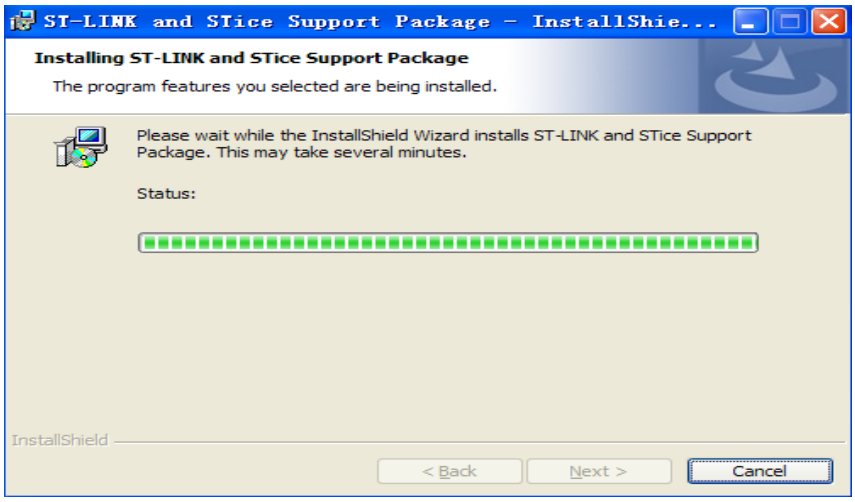


图 1.2.5.8　安装完成

第二章. 智能传感器模块部分

本章主要介绍 ICS-IOT-CEP 全功能物联网教学实验平台配套传感器模块部分内容。使用的传感器模块采用 STM8S 处理器，外接多种传感器电路，如温湿度、可燃气体、震动、三轴加速度、光感、人体监测、红外对射、声感、结露、酒精、磁检测和红外反射等传感器。传感器模块软件部分通过 STM8S 处理统一处理，并由自定义的串口协议实现与无线模块的通讯。

通过本章的内容学习，读者可以了解常见传感器的工作原理与相关编程使用方法，并通过产品出厂设置好的传感器串口协议，来使用其他通讯模块对传感器数据进行处理和应用。

注意：本章所涉及的传感器独立实验，涉及到串口的调试信息打印，在实验箱主板上无法完成实验，需要将传感器模块从主板上取下来，插接到平台配套的 USB2UART 模块上使用，用户在插拔传感器模块过程中切记不要带电插拔或插接错误！

如果一般用户使用，同样可以略过本章内容，把传感器模块当成非可编程的模块来使用，通过平台配套的传感器串口协议来与其他通讯模块通讯，这样用户可无需对传感器进行编程和插拔。

实验一. 磁检测传感器

1. 实验目的

- 了解干簧管硬件接口原理。
- 掌握磁检测传感器的工作原理。

2. 实验环境

- 软件：IAR SWSTM8 1.30。
- 硬件：ICS-IOT-CEP 教学实验平台，磁检测传感器模块，USB2UART 模块。

3. 实验原理

◆ 磁检测传感器简介

磁检测传感器使用的是干簧管。干簧管（Reed Switch）也称舌簧管或磁簧开关，是一种磁敏的特殊开关。它通常有两个软磁性材料做成的、无磁时断开的金属簧片触点，有的还有第三个作为常闭触点的簧片。这些簧片触点被封装在充有惰性气体(如氮、氦等)或真空的玻璃管里，玻璃管内平行封装的簧片端部重叠，并留有一定间隙或相互接触以构成开关的常开或常闭触点。干簧管比一般机械开关结构简单、体积小、速度高、工作寿命长，而与电子开关相比，它又有抗负载冲击能力强等特点，工作可靠性很高。

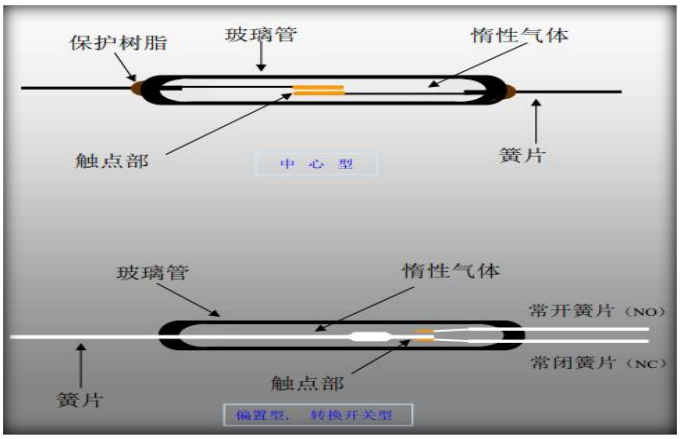


图 3.1 构造图

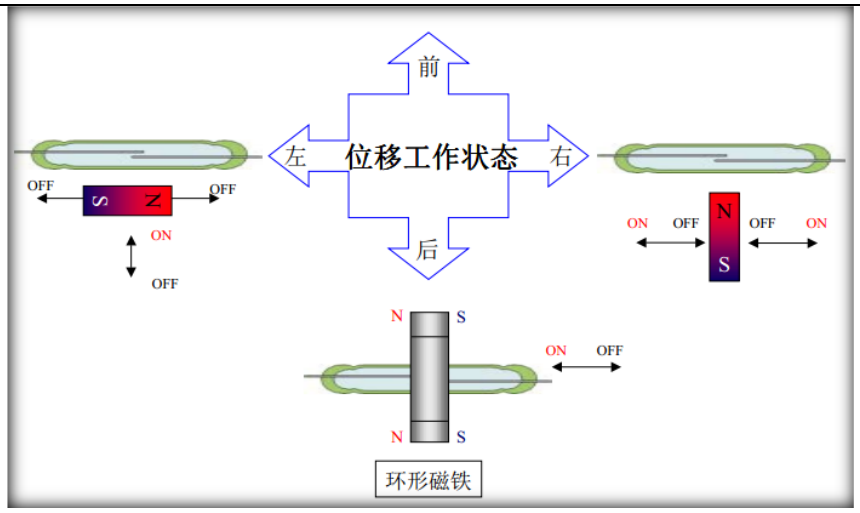


图 3.2 状态示意图

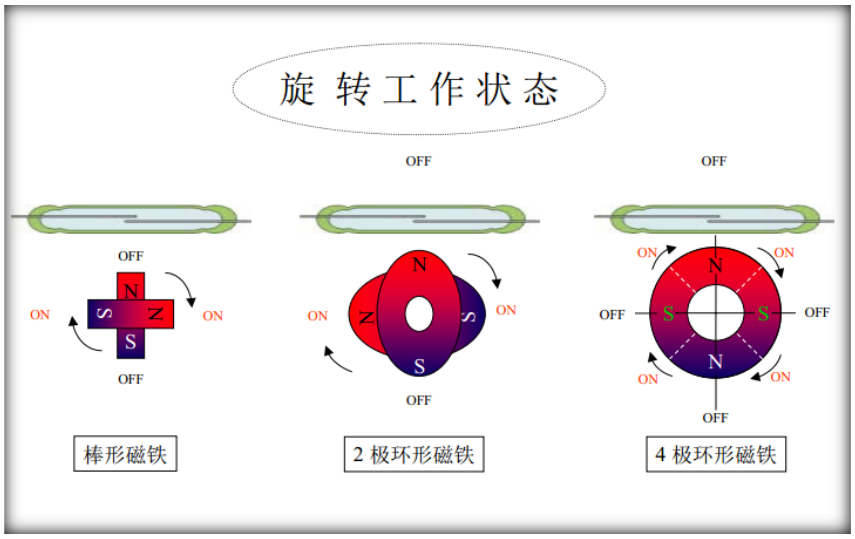


图 3.3 状态示意图

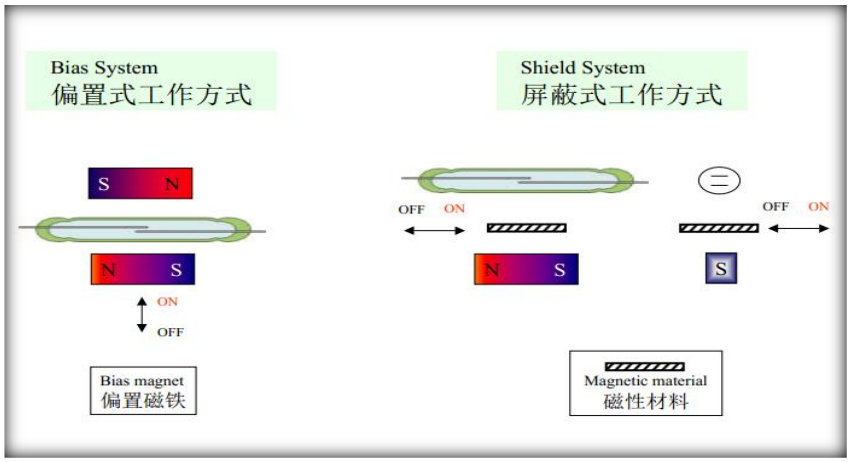


图 3.4 工作方式图

- **磁检测传感器模块原理图**

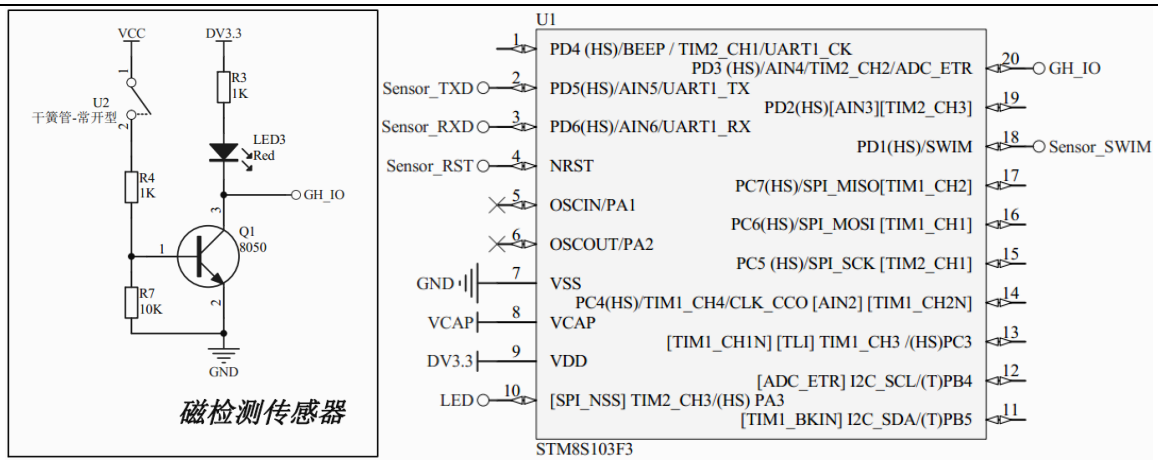


图 3.5 传感器模块原理图

如原理图所示，磁检测传感器干簧管断开时，三极管 8050 截止，LED3 熄灭，GH_IO 输出高电平；当磁场发生变化时，干簧管闭合，三极管 8050 导通，LED3 亮，GH_IO 输出低电平，STM8 读取连接 GP_IO 引脚的 PD3 IO 电平数值，判断是否有磁场，再通过串口协议输出状态。

- **源码分析**

串口函数为：

```
void UART1_SendString(u8* Data,u16 len)
{
    u16 i=0;
    for(;i<len;i++)
        UART1_SendByte(Data[i]);
 }
void UART1_SendByte(u8 data)
{
    UART1_SendData8((unsigned char)data);
    /* Loop until the end of transmission */
    while (UART1_GetFlagStatus(UART1_FLAG_TXE) == RESET);
}
```

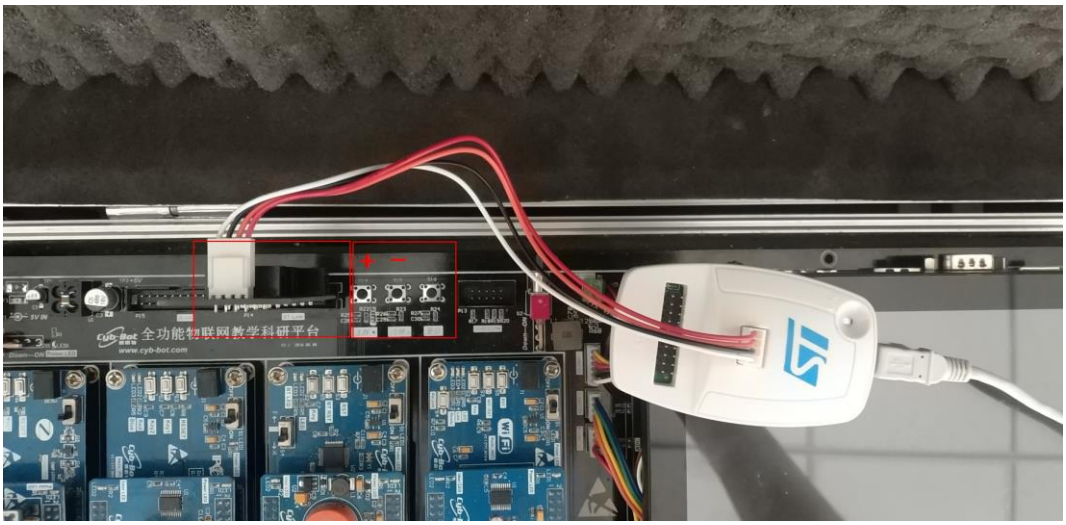
软件主要有配置干簧管，UART，LED 灯，实现代码如下：

```
u8 CMD_rx_buf[8];     // 命令缓冲区
u8 DATA_tx_buf[14]; // 返回数据缓冲区
u8 CMD_ID = 0;    // 命令序号
u8 Sensor_Type = 0; // 传感器类型编号
u8 Sensor_ID = 0;     // 相同类型传感器编号
u8 Sensor_Data[6];    // 传感器数据区
u8 Sensor_Data_Digital = 0; // 数字类型传感器数据
u16 Sensor_Data_Analog = 0; // 模拟类型传感器数据
u16 Sensor_Data_Threshod = 0;// 模拟传感器阈值
/* 根据不同类型的传感器进行修改 */
    Sensor_Type = 1;
```

```
    Sensor_ID = 1;
        CMD_ID = 1;
    DATA_tx_buf[0] = 0xEE;
    DATA_tx_buf[1] = 0xCC;
    DATA_tx_buf[2] = Sensor_Type;
    DATA_tx_buf[3] = Sensor_ID;
    DATA_tx_buf[4] = CMD_ID;
    DATA_tx_buf[13] = 0xFF;
    delay_ms(1000);
        while (1)
    {
        // 获取传感器数据
        if(GPIO_ReadInputPin(GPIOD, GPIO_PIN_3))
            Sensor_Data_Digital = 0;      // 无磁场
        else
            Sensor_Data_Digital = 1;      // 有磁场
                // 组合数据帧
        DATA_tx_buf[10] = Sensor_Data_Digital;
                // 发送数据帧
        UART1_SendString(DATA_tx_buf, 14);// 串口发送
        LED_Toggle();
        delay_ms(1000);
}
```

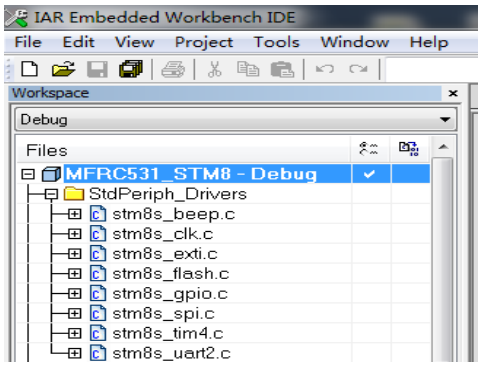
4. 实验步骤

1）首先我们要把传感器插到实验箱的主板上子节点的串口上，再把 ST-Link 配合 JTAG 转接板插到标有 ST-Link 标志的 JTAG 口上，最后把仿真器一端的 USB 线插到 PC 机的 USB 端口，通过主板上的“加”“减”按键选择要编程实验的传感器（会有黄色 LED 灯提示），硬件连接完毕。

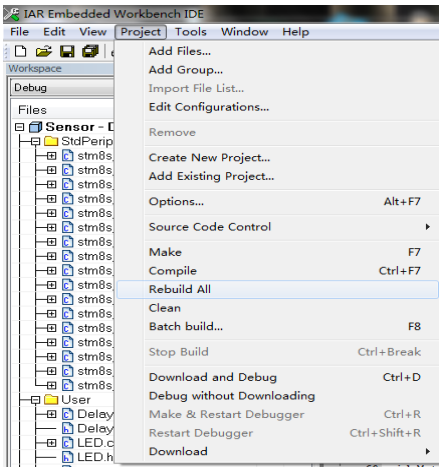


2）我们用 IAR SWSTM8 1.30 软件，打开..\1-Sensor_磁检测传感器\Project\Sensor.ew

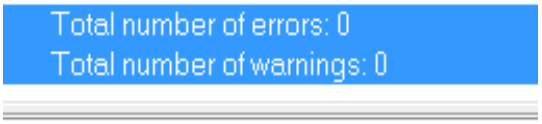
W。



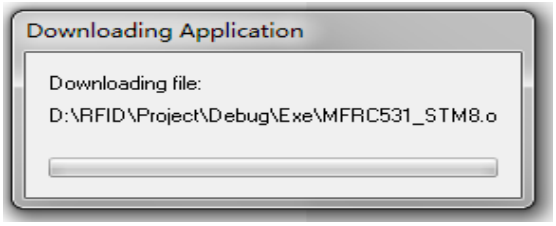
3）打开后点击“Project”下面的“Rebuild All”或者选中工程文件右键“Rebuild All”把我们的工程编译一下。



4）点击“Rebuild All”编译完后，无警告，无错误。



5）编译完后我们要把程序烧到模块里，点击“  ”中间的 Download and Debug 进行烧写。



6）烧写完毕后，把传感器模块从主板上取下来，连接到平台配套的 USB 转串口模块上，将 USB2UART 模块的 USB 线连接到 PC 机的 USB 端口，然后打开串口工具，配置好串口，波特率 115200，8 个数据位，一个停止位，无校验位。

7）传感器底层串口协议返回 14 个字节，第 1 位字节和 2 位字节是包头，第 3 位字节是传感器类型，第 4 位字节是传感器 ID，第 5 位字节是节点命令 ID，第 6 位字节到 11 位字节是数据位，其中第 11 位字节是传感器的状态位，第 12 位字节和第 13 位字节是保留位，第

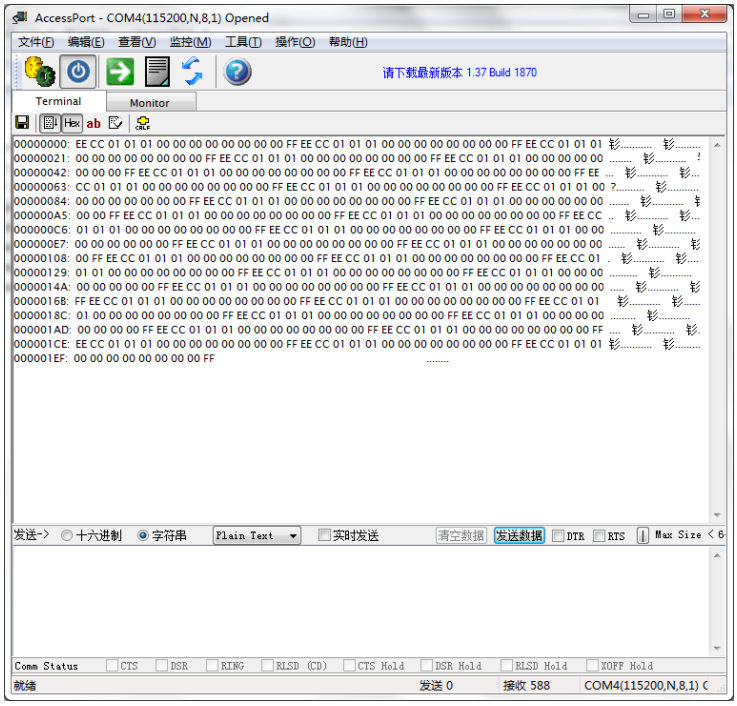
14 位字节是包尾。

例如：返回“EE CC 01 01 01 00 00 00 00 00 00 00 00 FF”时，第 11 位字节为“0”时，表示无磁，返回“EE CC 01 01 01 00 00 00 00 00 01 00 00 FF”时,第 11 位字节为“1”是表示有磁。

```
/* 根据不同类型的传感器进行修改 */
    Sensor_Type = 1;    //传感器类型
    Sensor_ID = 1;      //传感器 ID
    CMD_ID = 1;         //节点命令 ID
    DATA_tx_buf[0] = 0xEE;//包头
    DATA_tx_buf[1] = 0xCC;//包头
    DATA_tx_buf[2] = Sensor_Type;
    DATA_tx_buf[3] = Sensor_ID;
    DATA_tx_buf[4] = CMD_ID;
    DATA_tx_buf[13] = 0xFF;//包尾
```

下图为测试结果

更详细的协议说明，请用户参见《模块通讯协议 V2.6.pdf》文档。



实验二. 光照传感器

1. 实验目的

- 了解光敏电阻特性。
- 了解光敏传感器的工作原理。

2. 实验环境

- 软件：IAR SWSTM8 1.30
- 硬件：ICS-IOT-CEP 教学实验平台，光照传感器模块，USB2UART 模块

3. 实验原理

◆ 光敏电阻器

光照传感器使用的是光敏电阻。光敏电阻又称光导管，常用的制作材料为硫化镉，另外还有硒、硫化铝、硫化铅和硫化铋等材料。这些制作材料具有在特定波长的光照射下，其阻值迅速减小的特性。这是由于光照产生的载流子都参与导电，在外加电场的作用下作漂移运动，电子奔向电源的正极，空穴奔向电源的负极，从而使光敏电阻器的阻值迅速下降。

光敏电阻器是一种对光敏感的元件，它的电阻值能随着外界光照强弱（明暗）变化而变化。光敏电阻器的结构与特性 光敏电阻器通常由光敏层、玻璃基片（或树脂防潮膜）和电极等组成，光敏电阻器是利用半导体光电导效应制成的一种特殊电阻器，对光线十分敏感。它在无光照射时，呈高阻状态；当有光照射时，其电阻值迅速减小。光敏电阻器的应用 光敏电阻器广泛应用于各种自动控制电路（如自动照明灯控制电路、自动报警电路等）、家用电器（如电视机中的亮度自动调节，照相机中的自动曝光控制等）及各种测量仪器中。

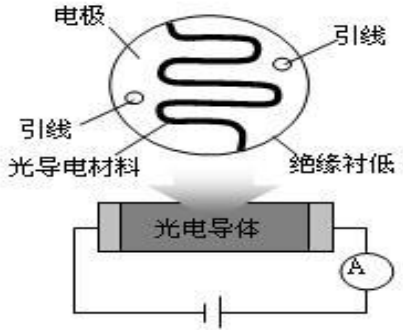


图 3.0

◆ 光敏电阻器的主要参数

光敏电阻器的主要参数有亮电阻（RL）、暗电阻（RD）、最高工作电压（VM）、亮电流（IL）、暗电流（ID）、时间常数、温度系数灵敏度等。

- 亮电阻：亮电阻是指光敏电阻器受到光照射时的电阻值。
- 暗电阻：暗电阻是指光敏电阻器在无光照射（黑暗环境）时的电阻值。
- 最高工作电压：最高工作电压是指光敏电阻器在额定功率下所允许承受的最高电压。
- 亮电流：亮电流是指在无光照射时，光敏电阻器在规定的外加电压受到光照时所通过的电流。
- 暗电流：暗电流是指在无光照射时，光敏电阻器在规定的外加电压下通过的电流。
- 时间常数：时间常数是指光敏电阻器从光照跃变开始到稳定亮电流的 63%时所需的时间。
- 电阻温度系数：温度系数是指光敏电阻器在环境温度改变 1℃时，其电阻值的相对变化。
- 灵敏度：灵敏度是指光敏电阻器在有光照射和无光照射时电阻值的相对变化。

伏安特性：在一定照度下，流过光敏电阻的电流与光敏电阻两端的电压的关系称为光敏电阻的伏安特性。下图为硫化镉光敏电阻的伏安特性曲线。由图可见，光敏电阻在一定的电压范围内，其曲线为直线。

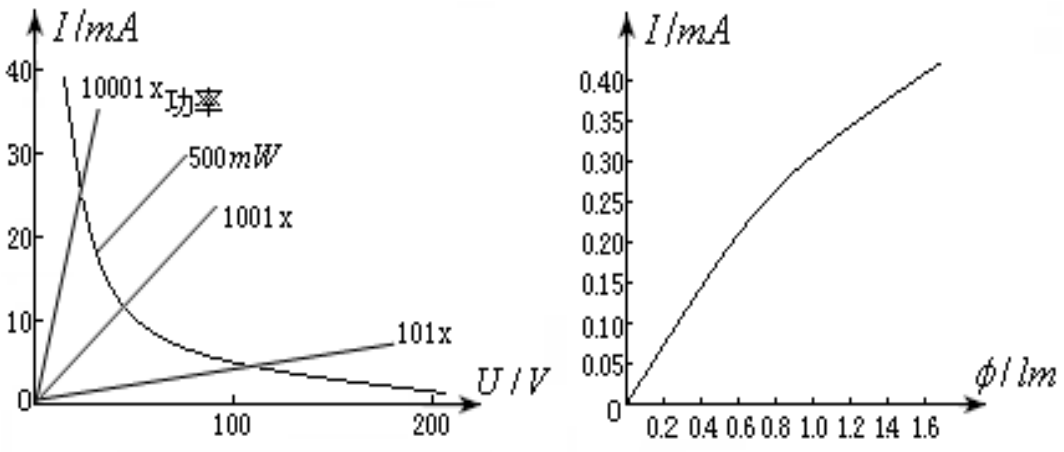


图 3.1 伏安特性曲线

光照特性：光敏电阻的光照特性是描述光电流和光照强度之间的关系，不同材料的光照特性是不同的，绝大多数光敏电阻光照特性是非线性的。下图为硫化光敏电阻的光照特性。

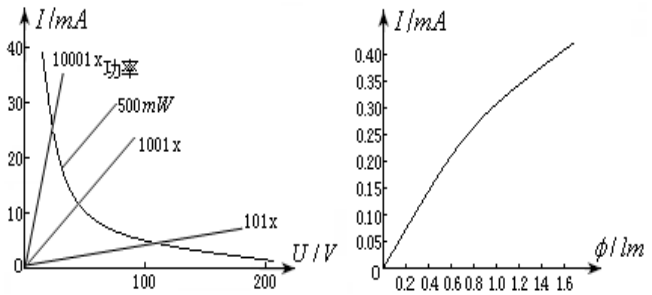


图 3.2 硫化光敏电阻

光谱特性：光敏电阻对入射光的光谱具有选择作用，即光敏电阻对不同波长的入射光有不同的灵敏度。光敏电阻的相对光敏灵敏度与入射波长的关系称为光敏电阻的光谱特性，亦称为光谱响应。下图 为几种不同材料光敏电阻的光谱特性。 对应于不同波长，光敏电阻的

灵敏度是不同的，而且不同材料的光敏电阻光谱响应曲线也不同。

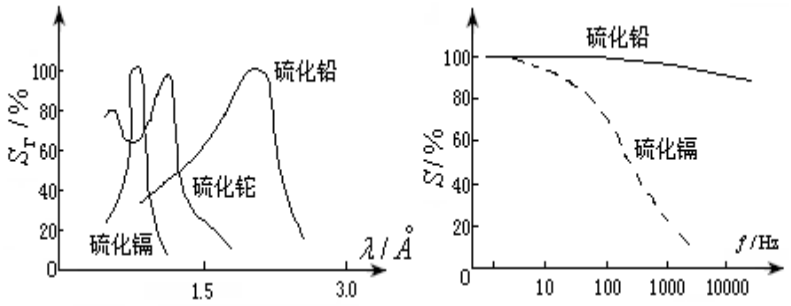


图 3.3 光敏电阻响应时间

- **光敏传感器模块原理图**

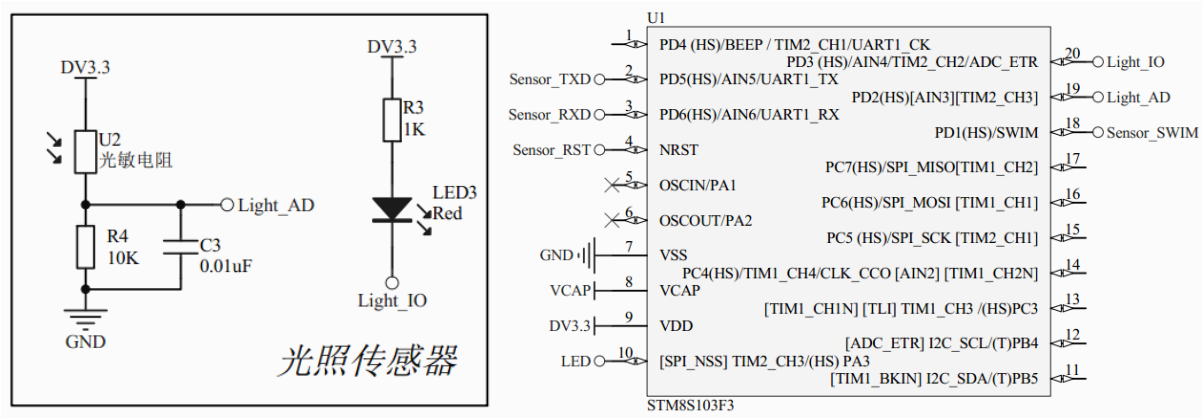


图 3.4 原理图

光敏电阻阻值随光照强度变化而变化，在引脚 Light_AD 输出电压值也随之变化。用 STM8 的 PD2 引脚采集 Light_AD 电压模拟量并转为数字量，当采集的 AD 值大于某一阈值（本程序设置为 700），则将 PD3 即 Light_IO 引脚置低，表明有光照。

传感器使用的光敏电阻的暗电阻为 2M 欧姆左右，亮电阻为 10K 左右。可以计算出：在黑暗条件下，Light_AD 的数值为 3.3V * 2000K /(2000K + 10K) = 3.28V。在光照条件下，Light_AD 的数值为 3.3V * 10K /(10K + 10K) = 1.65V。STM8 单片机内部带有 10 位 AD 转换器，参考电压为供电电压 3.3V。根据上面计算结果，选定 1.65V（需要根据实际测量结果进行调整）作为临界值。当 Light_AD 为 1.65V 时，AD 读数为 1.65 / 3.3 * 1024 = 512，

当 AD 读数大于 512 时说明无光照，当 AD 读数小于 512 时说明有光照，并点亮 LED3 作为指示。并通过串口函数来传送触发（有光照时）信号。

- **源码分析**

实现代码如下：

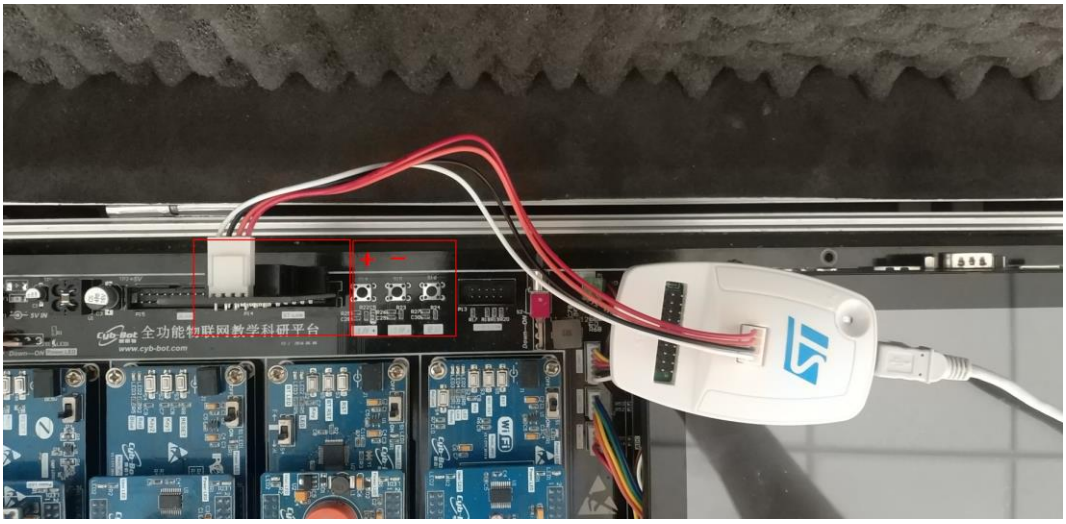
```
u8 CMD_rx_buf[8];     // 命令缓冲区
u8 DATA_tx_buf[14]; // 返回数据缓冲区
u8 CMD_ID = 0;    // 命令序号
u8 Sensor_Type = 0; // 传感器类型编号
u8 Sensor_ID = 0;     // 相同类型传感器编号
u8 Sensor_Data[6];    // 传感器数据区
```

```
u8 Sensor_Data_Digital = 0; // 数字类型传感器数据
u16 Sensor_Data_Analog = 0; // 模拟类型传感器数据
u16 Sensor_Data_Threshod = 0;// 模拟传感器阈值
/* 根据不同类型的传感器进行修改 */
    Sensor_Type = 2;
    Sensor_ID = 1;
    CMD_ID = 1;
    DATA_tx_buf[0] = 0xEE;
    DATA_tx_buf[1] = 0xCC;
    DATA_tx_buf[2] = Sensor_Type;
    DATA_tx_buf[3] = Sensor_ID;
    DATA_tx_buf[4] = CMD_ID;
    DATA_tx_buf[13] = 0xFF;
    GPIO_Init(GPIOD, GPIO_PIN_3, GPIO_MODE_OUT_PP_HIGH_SLOW);
    // ADC
    ADC1_Init(ADC1_CONVERSIONMODE_CONTINUOUS,
              ADC1_CHANNEL_3,
              ADC1_PRESSEL_FCPU_D4,
              ADC1_EXTTRIG_TIM,
              DISABLE,
              ADC1_ALIGN_RIGHT,
              ADC1_SCHMITTTRIG_CHANNEL3,
              DISABLE);
    ADC1_Cmd(ENABLE);
    ADC1_StartConversion();
    Sensor_Data_Analog = 0;
    Sensor_Data_Threshod = 700;
    delay_ms(1000);
    while (1)
    {
        // 获取传感器数据
        Sensor_Data_Analog = ADC1_GetConversionValue();
        if(Sensor_Data_Analog < Sensor_Data_Threshod)
        {
            Sensor_Data_Digital = 0;      // 无光照
            GPIO_WriteHigh(GPIOD, GPIO_PIN_3);
        }
        else
        {
            Sensor_Data_Digital = 1;      // 有光照
            GPIO_WriteLow(GPIOD, GPIO_PIN_3);
        }
                // 组合数据帧
        DATA_tx_buf[10] = Sensor_Data_Digital;
        // 发送数据帧
        UART1_SendString(DATA_tx_buf, 14);//串口发送
        LED_Toggle();
        delay_ms(1000);
    }
}
```

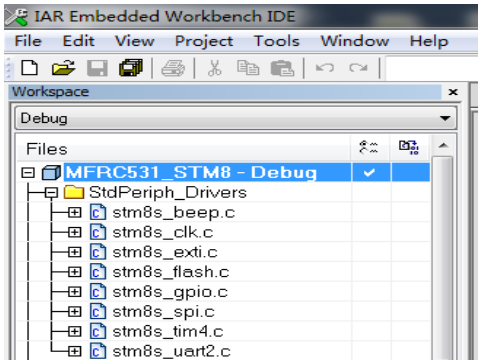
4. 实验步骤

1）首先我们要把传感器插到实验箱的主板上子节点的串口上，再把 ST-Link 配合 JTAG 转接板插到标有 ST-Link 标志的 JTAG 口上，最后把仿真器一端的 USB 线插到 PC 机的 USB 端口，通过主板上的“加”“减”按键选择要编程实验的传感器（会有黄色 LED 灯提示），

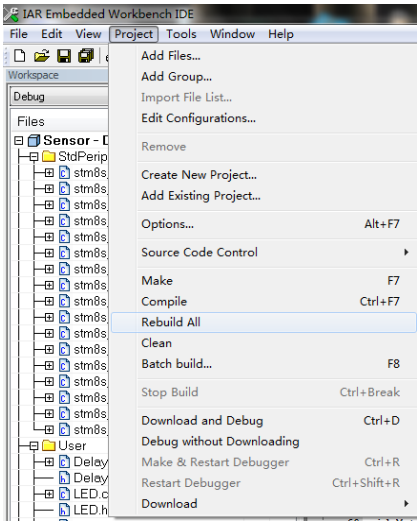
硬件连接完毕。



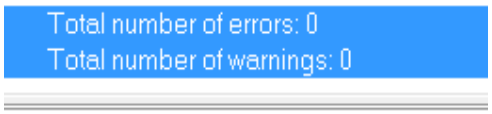
2）我们用 IAR SWSTM8 1.30 软件，打开..\2-Sensor_光照传感器\Project\Sensor.eww。



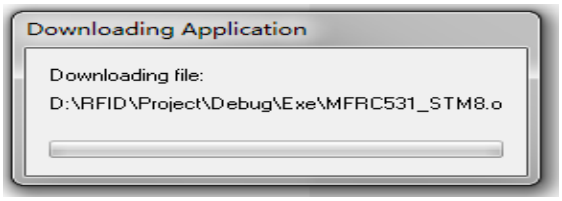
3）打开后点击“Project”的“Rebuild All”或者选中工程文件右键“Rebuild All”把我们的工程编译一下。



4）点击“Rebuild All”编译完后，无警告，无错误。



5）编译完后我们要把程序烧到模块里，点击“  ”中间的 Download and Debug 进行烧写。



6）烧写完毕后，把传感器模块从主板上取下来，连接到平台配套的 USB 转串口模块上，将 USB2UART 模块的 USB 线连接到 PC 机的 USB 端口，然后打开串口工具，配置好串口，波特率 115200，8 个数据位，一个停止位，无校验位。

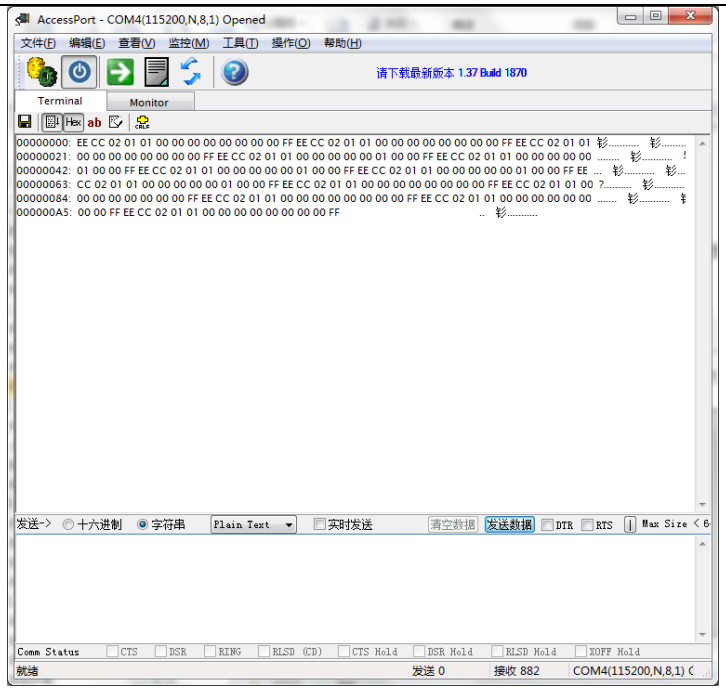
7）传感器底层串口协议返回 14 个字节，第 1 位字节和 2 位字节是包头，第 3 位字节是传感器类型，第 4 位字节是传感器 ID,第 5 位字节是节点命令 ID，第 6 位字节到 11 位字节是数据位，其中第 11 位字节是传感器的状态位，第 12 位字节和第 13 位字节是保留位，第 14 位字节是包尾。

例如：返回“EE CC 02 01 01 00 00 00 00 00 00 00 FF”时，第 11 位字节为“0”时，表示无光照，返回“EE CC 02 01 01 00 00 00 00 00 01 00 00 FF”时,第 11 位字节为“1”是表示有光照。

```
/* 根据不同类型的传感器进行修改 */
    Sensor_Type = 1;     //传感器类型
    Sensor_ID = 1;       //传感器 ID
    CMD_ID = 1;          //节点命令 ID
    DATA_tx_buf[0] = 0xEE;//包头
    DATA_tx_buf[1] = 0xCC;//包头
    DATA_tx_buf[2] = Sensor_Type;
    DATA_tx_buf[3] = Sensor_ID;
    DATA_tx_buf[4] = CMD_ID;
    DATA_tx_buf[13] = 0xFF;//包尾
```

下图为测试结果

更详细的协议说明，请用户参见《模块通讯协议 V2.6.pdf》文档。


AccessPort - COM4(115200,N,8,1) Opened
文件(F) 编辑(E) 查看(V) 监控(M) 工具(T) 操作(O) 帮助(H)
请下载最新版本 1.37 Build 1870
Terminal
Monitor
00000000: EE CC 02 01 01 00 00 00 00 00 00 00 00 FF EE CC 02 01 01 00 00 00 00 00 00 00 00 FF EE CC 02 01 01
00000021: 00 00 00 00 00 00 00 00 FF EE CC 02 01 01 00 00 00 00 00 00 01 00 00 FF EE CC 02 01 01 00 00 00 00 00
00000042: 01 00 00 FF EE CC 02 01 01 00 00 00 00 00 00 01 00 00 FF EE CC 02 01 01 00 00 00 00 00 00 01 00 00 FF EE
00000063: CC 02 01 01 00 00 00 00 00 00 01 00 00 FF EE CC 02 01 01 00 00 00 00 00 00 00 00 00 FF EE CC 02 01 01 00
00000084: 00 00 00 00 00 00 00 FF EE CC 02 01 01 00 00 00 00 00 00 00 00 00 FF EE CC 02 01 01 00 00 00 00 00 00
000000A5: 00 00 FF EE CC 02 01 01 00 00 00 00 00 00 00 00 FF
发送-> 十六进制 字符串 Plain Text 实时发送 清空数据 发送数据 DTR RTS Max Size
Comm Status CTS DSR RING RLSD (CD) CTS Hold DSR Hold RLSD Hold XOFF Hold
就绪 发送 0 接收 882 COM4(115200,N,8,1)

实验三. 红外对射传感器

1. 实验目的

- 了解红外线的应用。
- 掌握红外对射传感器工作原理。

2. 实验环境

- 软件：IAR SWSTM8 1.30
- 硬件：ICS-IOT-CEP 教学实验平台，红外对射传感器模块，USB2UART 模块

3. 实验原理

◆ 红外对射传感器简介

红外对射传感器使用的是槽型红外光电开关。红外光电传感器是捕捉红外线这种不可见光，采用专用的红外发射管和接收管，转换为可以观测的电信号。红外光电传感器有效地防止周围可见光的干扰，进行无接触探测，不损伤被测物体。红外光电传感器在一般情况下，有三部分构成，它们分为：发送器、接收器和检测电路。红外光电传感器的发送器对准目标发射光束，当前面有被检测物体时，物体将发射器发出的红外光线反射回接收器，于是红外光电传感器就“感知”了物体的存在，产生输出信号。

图 3.1 元器件

如图 3.1 所示，槽型红外光电开关把一个红外光发射器和一个红外光接收器面对面地装在一个槽的两侧。发光器能发出红外光，在无阻情况下光接收器能收到光。但当被检测物体从槽中通过时，光被遮挡，光电开关便动作，输出一个开关控制信号，切断或接通负载电流，从而完成一次控制动作。槽形开关的检测距离因为受整体结构的限制一般只有几厘米。

◆ 红外对射传感器模块原理图

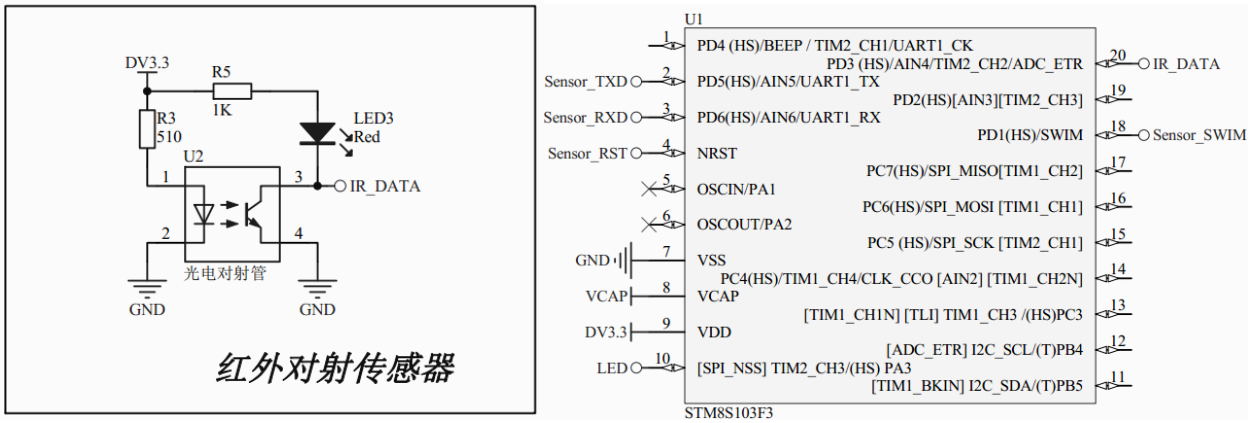


图 3.2 原理图

如图 3.2 所示，当槽型光电开关 U2 中间有障碍物遮挡时，IR_DATA 为高电平，LED3 熄灭；当槽型光电开关 U2 中间无障碍物遮挡时，IR_DATA 为低电平，LED3 点亮。通过 STM8 单片机读取连接 IR_DATA 引脚的 PD3 IO 的高低电平状态，即可获知红外对射传感器是否检测到障碍物，通过串口通信传输信号。

- **源码分析**

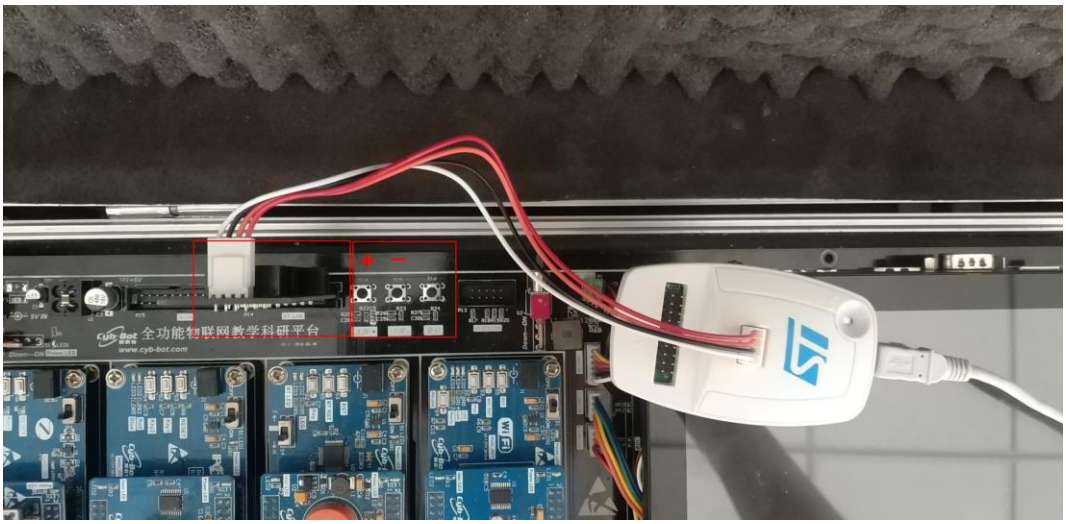
实现代码如下：

```
u8 CMD_rx_buf[8];     // 命令缓冲区
u8 DATA_tx_buf[14]; // 返回数据缓冲区
u8 CMD_ID = 0;    // 命令序号
u8 Sensor_Type = 0; // 传感器类型编号
u8 Sensor_ID = 0;     // 相同类型传感器编号
u8 Sensor_Data[6];    // 传感器数据区
u8 Sensor_Data_Digital = 0; // 数字类型传感器数据
u16 Sensor_Data_Analog = 0; // 模拟类型传感器数据
u16 Sensor_Data_Threshod = 0;// 模拟传感器阈值
/* 根据不同类型的传感器进行修改 */
    Sensor_Type = 3;
    Sensor_ID = 1;
    CMD_ID = 1;
    DATA_tx_buf[0] = 0xEE;
    DATA_tx_buf[1] = 0xCC;
    DATA_tx_buf[2] = Sensor_Type;
    DATA_tx_buf[3] = Sensor_ID;
    DATA_tx_buf[4] = CMD_ID;
    DATA_tx_buf[13] = 0xFF;
    delay_ms(1000);
    while (1)
    {
        // 获取传感器数据
        if(!GPIO_ReadInputPin(GPIOD, GPIO_PIN_3))
        {
            Sensor_Data_Digital = 0;      // 无障碍
        }
        else
        {
            Sensor_Data_Digital = 1;      // 有障碍
        }
```

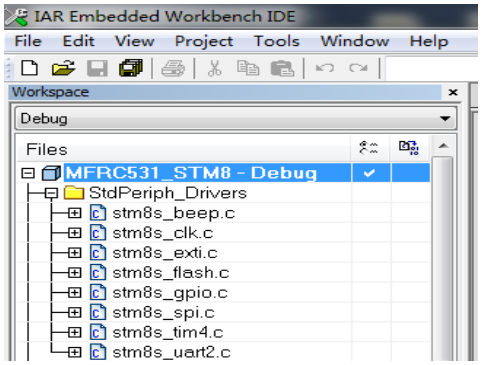
```
            // 组合数据帧
            DATA_tx_buf[10] = Sensor_Data_Digital;
            // 发送数据帧
            UART1_SendString(DATA_tx_buf, 14);
            LED_Toggle();
            delay_ms(1000);
    }
```

4. 实验步骤

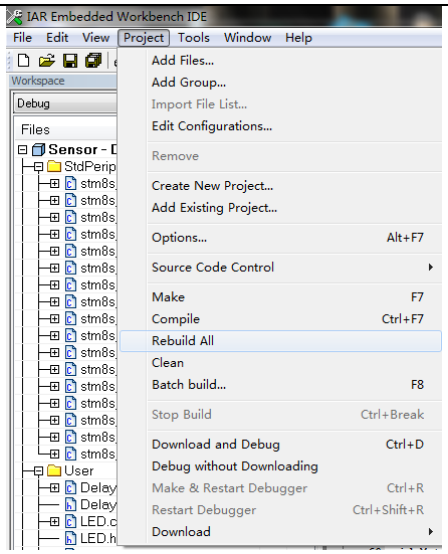
1）首先我们要把传感器插到实验箱的主板上子节点的串口上，再把 ST-Link 配合 JTAG 转接板插到标有 ST-Link 标志的 JTAG 口上，最后把仿真器一端的 USB 线插到 PC 机的 USB 端口，通过主板上的“加”“减”按键选择要编程实验的传感器（会有黄色 LED 灯提示），硬件连接完毕。



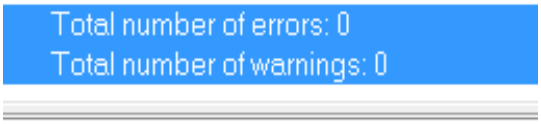
2）我们用 IAR SWSTM8 1.30 软件，打开..\3-Sensor_红外对射传感器\Project\Sensor.eww。



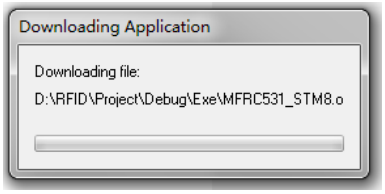
3）打开后点击“Project”下面的“Rebuild All”或者选中工程文件右键“Rebuild All”把我们的工程编译一下。

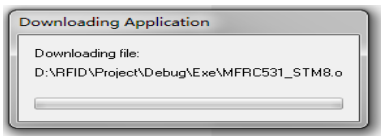


4）点击“Rebuild All”编译完后，无警告，无错误。



5）编译完后我们要把程序烧到模块里，点击“  ”中间的 Download and Debug 进行烧写。





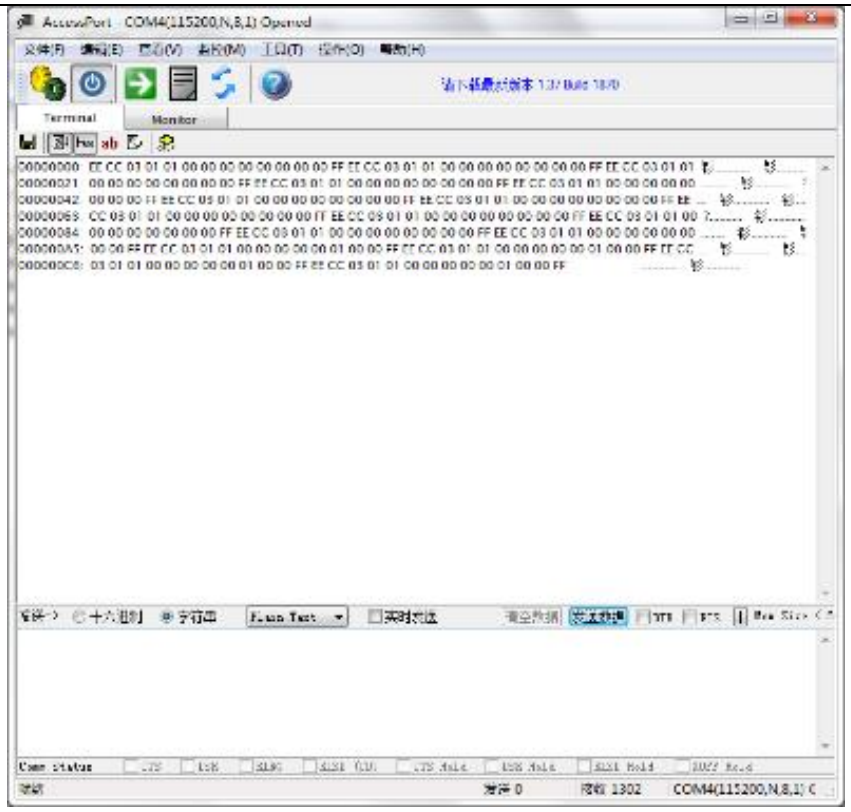
6）烧写完毕后，把传感器模块从主板上取下来，连接到平台配套的 USB 转串口模块上，将 USB2UART 模块的 USB 线连接到 PC 机的 USB 端口，然后打开串口工具，配置好串口，波特率 115200，8 个数据位，一个停止位，无校验位。

7）传感器底层串口协议返回 14 个字节，第 1 位字节和 2 位字节是包头，第 3 位字节是传感器类型，第 4 位字节是传感器 ID,第 5 位字节是节点命令 ID，第 6 位字节到 11 位字节是数据位，其中第 11 位字节是传感器的状态位，第 12 位字节和第 13 位字节是保留位，第 14 位字节是包尾。

例如：返回“EE CC 03 01 01 00 00 00 00 00 00 00 FF”时，第 11 位字节为“0”时，表示无障碍，返回“EE CC 03 01 01 00 00 00 00 00 01 00 00 FF”时,第 11 位字节为“1”是表示有障碍。

下图为测试结果。

更详细的协议说明，请用户参见《模块通讯协议 V2.6.pdf》文档。



实验四. 红外反射传感器

1. 实验目的

- 加深对红外线的了解。
- 理解红外反射传感器的工作原理。

2. 实验环境

- 软件：IAR SWSTM8 1.30
- 硬件：ICS-IOT-CEP 教学实验平台，红外反射传感器模块，USB2UART 模块

3. 实验原理

♦ 红外反射传感器简介

红外反射传感器使用的是反射型红外光电开关，反射型红外光电开关把一个红外光发射器和一个红外光接收器装在一个同一面上，前方装有滤镜，滤除干扰光。发光器能发出红外光，在无阻情况下光接收器不能收到光。但当前方有障碍物时，光被反射回接收器，光电开关便动作，输出一个开关控制信号，切断或接通负载电流，从而完成一次控制动作。反射型光电开关的检测距离从几 cm 到几 m 不等，在工业测控、安防等方面具有很广的应用。

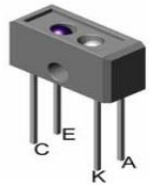


图 3.1 元器件

♦ 红外反射传感器模块原理图

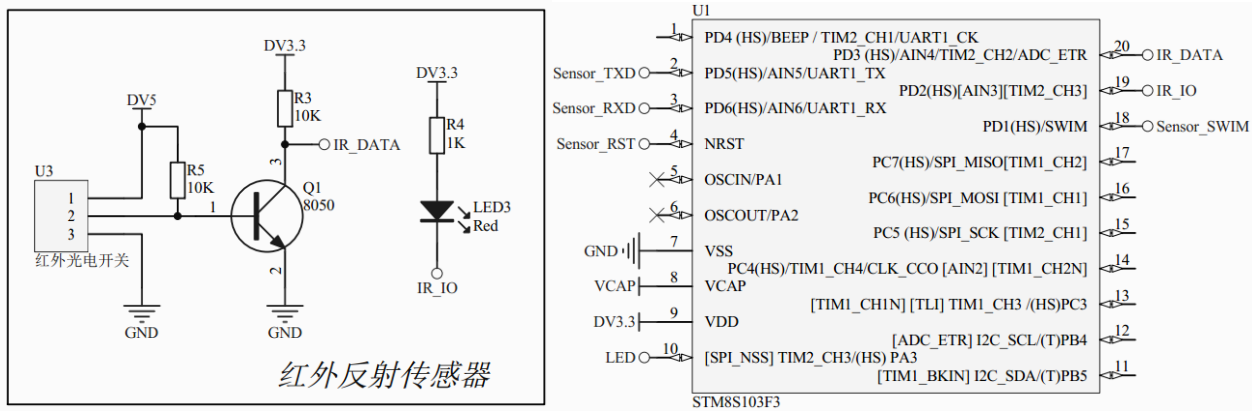


图 3.2 原理图

如图 3.2 所示，红外光电开关 U3 供电电压为 5V，集电极开路输出。当无障碍物时，U3 的 1 脚输出高电平，Q1 导通，IR_DATA 为低电平；当有障碍物时，U3 的 1 脚输出低电平，Q1 截止，IR_DATA 为高电平。通过 STM8 单片机读取连接 IR_DATA 引脚的 PD3 高低电平状态，即可获知红外反射传感器是否检测到障碍物，当检测到障碍物时，可以点亮 LED3 作为指示，通过串口传输信号。

♦ 源码分析

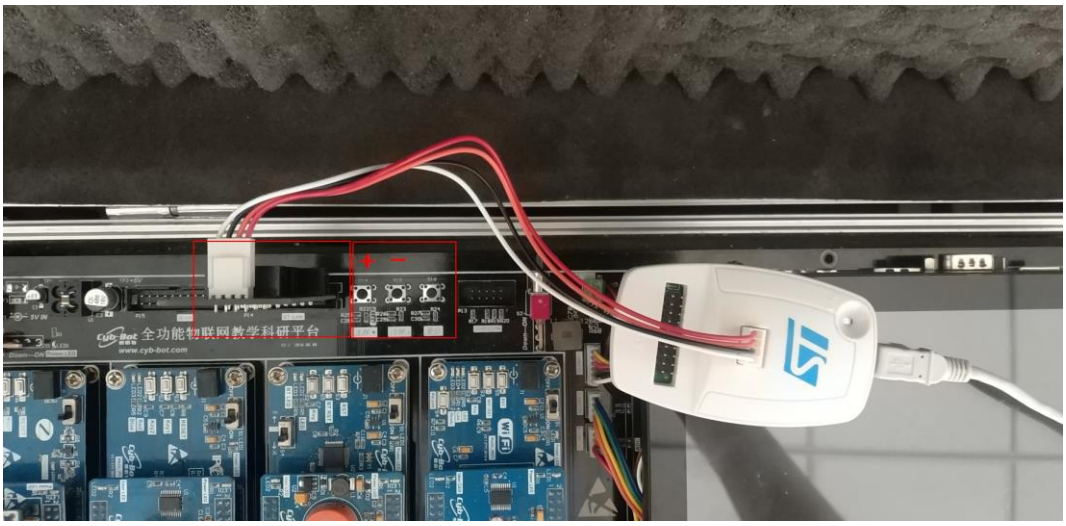
实现代码如下：

```
u8 CMD_rx_buf[8];     // 命令缓冲区
u8 DATA_tx_buf[14]; // 返回数据缓冲区
u8 CMD_ID = 0;    // 命令序号
u8 Sensor_Type = 0; // 传感器类型编号
u8 Sensor_ID = 0;     // 相同类型传感器编号
u8 Sensor_Data[6];    // 传感器数据区
u8 Sensor_Data_Digital = 0; // 数字类型传感器数据
u16 Sensor_Data_Analog = 0; // 模拟类型传感器数据
u16 Sensor_Data_Threshod = 0;// 模拟传感器阈值
/* 根据不同类型的传感器进行修改 */
    Sensor_Type = 4;
    Sensor_ID = 1;
    CMD_ID = 1;

    DATA_tx_buf[0] = 0xEE;
    DATA_tx_buf[1] = 0xCC;
    DATA_tx_buf[2] = Sensor_Type;
    DATA_tx_buf[3] = Sensor_ID;
    DATA_tx_buf[4] = CMD_ID;
```

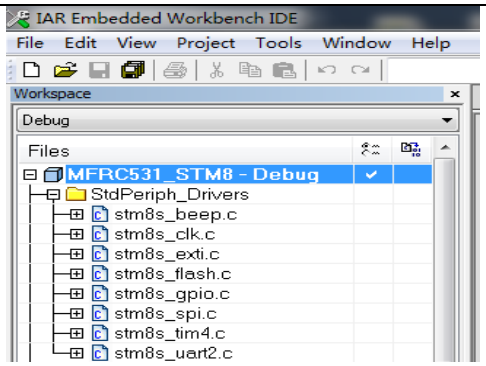
```
    DATA_tx_buf[13] = 0xFF;
    delay_ms(1000);
    while (1)
    {
        // 获取传感器数据
        if(!GPIO_ReadInputPin(GPIOD, GPIO_PIN_3))
        {
            Sensor_Data_Digital = 0;      // 无障碍
            GPIO_WriteHigh(GPIOD, GPIO_PIN_2);
        }
        else
        {
            Sensor_Data_Digital = 1;      // 有障碍
            GPIO_WriteLow(GPIOD, GPIO_PIN_2);
        }
        // 组合数据帧
        DATA_tx_buf[10] = Sensor_Data_Digital;
        // 发送数据帧
        UART1_SendString(DATA_tx_buf, 14);//串口发送
        LED_Toggle();
        delay_ms(1000);
    }
```

4. 实验步骤

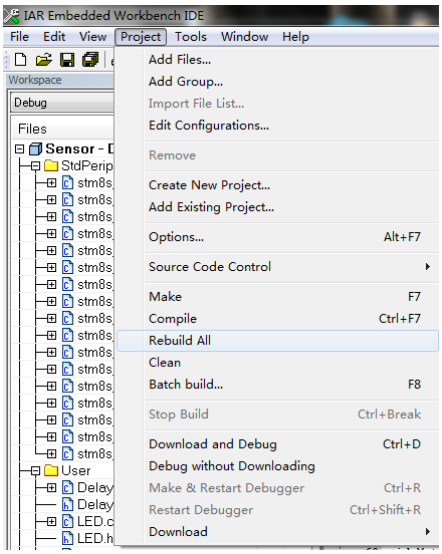
1）首先我们要把传感器插到实验箱的主板上子节点的串口上，再把 ST-Link 配合 JTAG 转接板插到标有 ST-Link 标志的 JTAG 口上，最后把仿真器一端的 USB 线插到 PC 机的 USB 端口，通过主板上的“加”“减”按键选择要编程实验的传感器（会有黄色 LED 灯提示），硬件连接完毕。



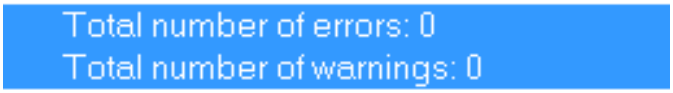
2）我们用 IAR SWSTM8 1.30 软件，打开..\4-Sensor_红外反射传感器\Project\Sensor.eww。

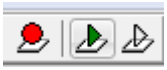


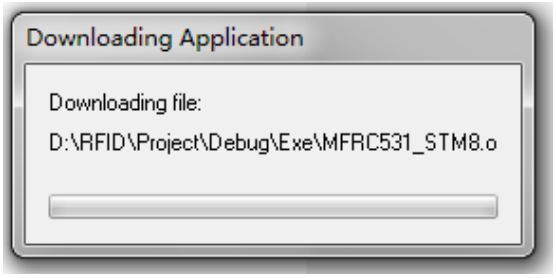
3）打开后点击“Project”下面的“Rebuild All”或者选中工程文件右键“Rebuild All”把我们的工程编译一下。



4）点击“Rebuild All”编译完后，无警告，无错误。



5）编译完后我们要把程序烧到模块里，点击“  ”中间的 Download and Debug 进行烧写。



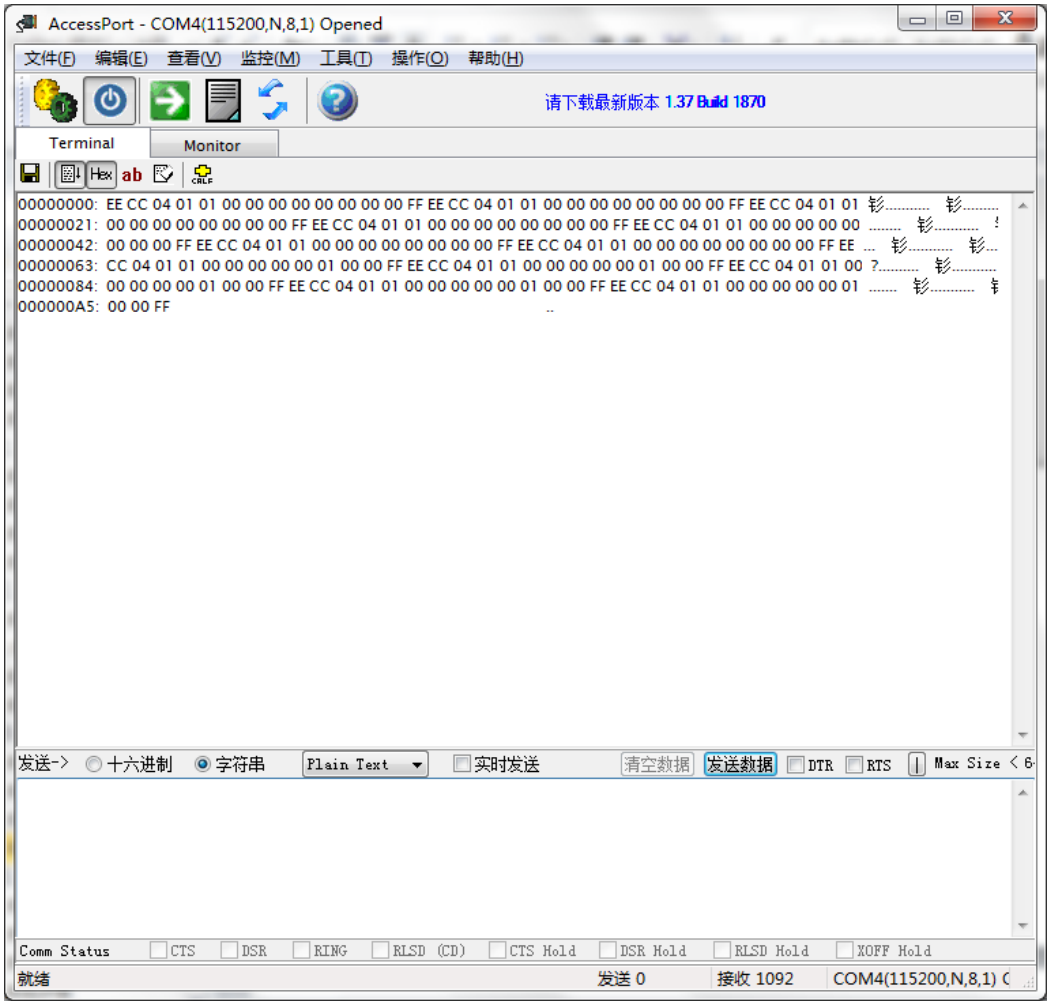
6）烧写完毕后，把传感器模块从主板上取下来，连接到平台配套的 USB 转串口模块上，将 USB2UART 模块的 USB 线连接到 PC 机的 USB 端口，然后打开串口工具，配置好串口，波特率 115200，8 个数据位，一个停止位，无校验位。

7）传感器底层串口协议返回 14 个字节，第 1 位字节和 2 位字节是包头，第 3 位字节是传感器类型，第 4 位字节是传感器 ID,第 5 位字节是节点命令 ID，第 6 位字节到 11 位字节

是数据位，其中第 11 位字节是传感器的状态位，第 12 位字节和第 13 位字节是保留位，第 14 位字节是包尾。

例如：返回“EE CC 04 01 01 00 00 00 00 00 00 00 FF”时，第 11 位字节为“0”时，表示无障碍，返回“EE CC 04 01 01 00 00 00 00 00 01 00 00 FF”时,第 11 位字节为“1”是表示有障碍。

下图为测试结果



实验五. 结露传感器

1. 实验目的

- 了解 HDS05 电气元件。
- 了解结露传感器的工作原理

2. 实验环境

- 软件：IAR SWSTM8 1.30

■ 硬件：ICS-IOT-CEP 教学实验平台，结露传感器模块，USB2UART 模块

3. 实验原理

♦ HDS05 简介

HDS05 结露传感器是正特性开关型元件，对低湿不敏感而仅对高湿度敏感，可在直流电压下工作，质量稳定，可靠性达到国际先进水平。

HDS 特点

- 高湿环境下具有极高敏感性
- 具有开关功能
- 直流电压下工作
- 响应速度快
- 抗污染能力强
- 高可靠性，稳定性好

使用参数：

供电电压：	0.8V DC（安全电压）
使用温度范围：	1~80℃
使用湿度范围：	1 ～ 100%RH
结露测试范围：	94 ～ 100%RH

图 3.1 使用参数

特性参数：

项目		试验条件	规格	
电阻值		75%RH 25℃	10K Ω Max	
		93%RH 25℃	70K Ω Max	
		100%RH 60℃	200K Ω Max	
响应速度		25 ℃， 60%RH → 60 ℃， 100%RH	5秒以下	
温度循环		-40 ℃，30 分钟 → 85 ℃，30 分钟 5 个 循环	试验后的特征 1、响应速度： 10 秒以下 2、电阻值 ↓	
放置	高温	85 ℃， 2000 小时		
	低温	-40 ℃， 2000 小时		
	低温	40 ℃， 5%RH， 2000 小时	相对湿度	电阻值
	高温	40 ℃， 90 ～ 95%RH， 2000 小时	25℃ 75%RH	20K Ω Max
高湿负载		40 ℃， 90-95% 中加 DC 0.8V 2000 小时	25℃ 93%RH	90~100KΩ
结露循环		25 ℃， 60%RH， 57 分钟 ←→ 25 ℃， 100%RH 3 分钟 2000 次	60 ℃ 100%RH	100KΩ 以上

图 3.2 特性参数

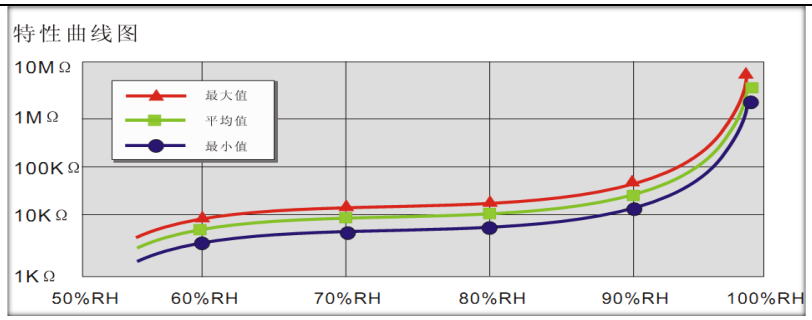


图 3.3 特性曲线图

- **结露传感器模块原理图**

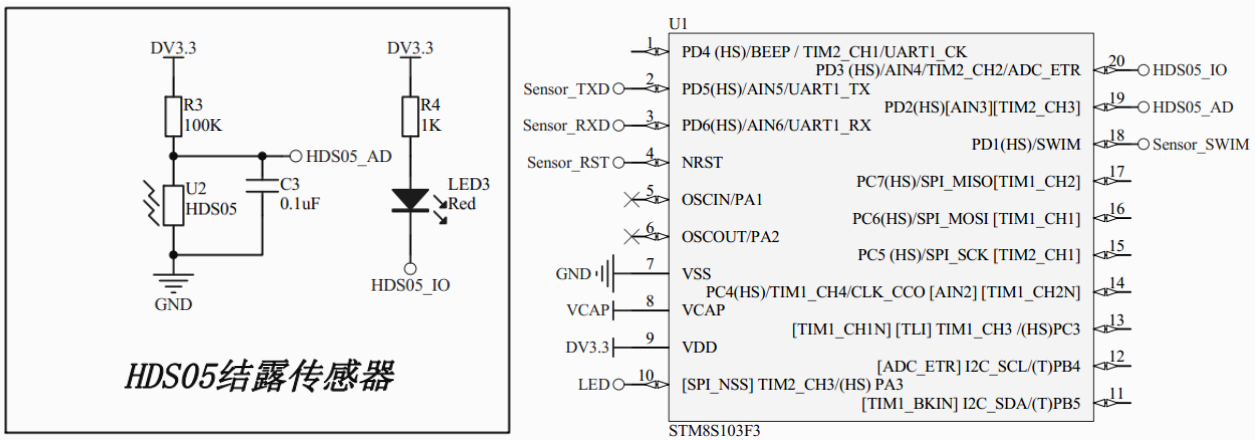


图 3.4 HDS05 原理图

如图 5.4 所示，用 STM8 的 PD2 即 HDS05_AD 采集电压模拟量并转换为数字量，当大于某一阈值（程序设置为 120）时，判断已经结露，置低 HDS05_IO，点亮 LED3。HDS05_AD 最大值为 47K / (47K + 150K) * 3.3V = 0.787V，小于 HDS05 的最大供电电压 0.8V。由特性参数表中可知，75% RH 25℃条件下，HDS05 电阻为 10K 欧姆，此时，容易计算出 HDS05_AD 为 0.172V。当湿度增加时，电阻增大，HDS05_AD 增大，选定一个临界值（根据实际情况选择），比如 0.172V ，此时 AD 读数为 0.172 / 3.3 * 1024 = 53，当 AD 采集的数值大于 53 时表明有结露，并点亮 LED3 作为指示，再通过串口通信传输信号。

- **源码分析**

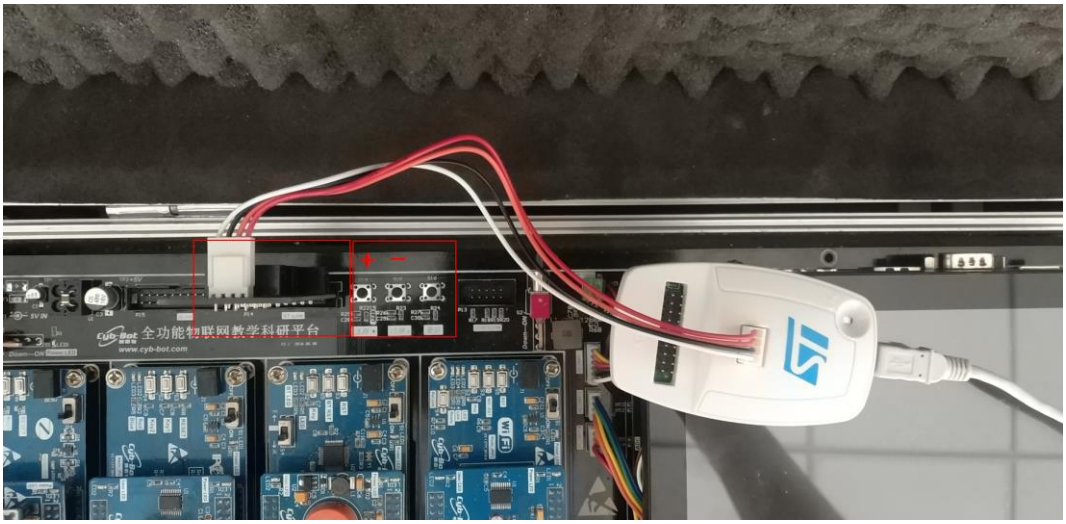
实现代码如下：

```
u8 CMD_rx_buf[8];     // 命令缓冲区
u8 DATA_tx_buf[14]; // 返回数据缓冲区
u8 CMD_ID = 0;    // 命令序号
u8 Sensor_Type = 0; // 传感器类型编号
u8 Sensor_ID = 0;     // 相同类型传感器编号
u8 Sensor_Data[6];    // 传感器数据区
u8 Sensor_Data_Digital = 0; // 数字类型传感器数据
u16 Sensor_Data_Analog = 0; // 模拟类型传感器数据
u16 Sensor_Data_Threshod = 0;// 模拟传感器阈值
/* 根据不同类型的传感器进行修改 */
   Sensor_Type = 5;
   Sensor_ID = 1;
```

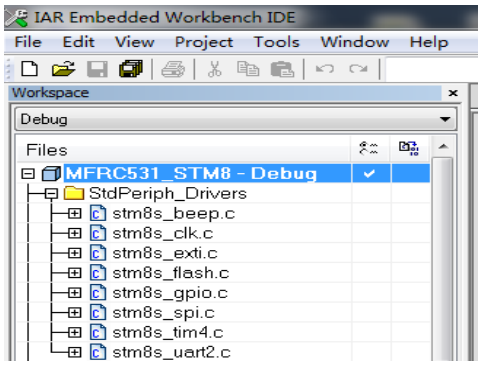
```
    CMD_ID = 1;
    DATA_tx_buf[0] = 0xEE;
    DATA_tx_buf[1] = 0xCC;
    DATA_tx_buf[2] = Sensor_Type;
    DATA_tx_buf[3] = Sensor_ID;
    DATA_tx_buf[4] = CMD_ID;
    DATA_tx_buf[13] = 0xFF;
    GPIO_Init(GPIOD, GPIO_PIN_3, GPIO_MODE_OUT_PP_HIGH_SLOW);
     // ADC
    ADC1_Init(ADC1_CONVERSIONMODE_CONTINUOUS,
              ADC1_CHANNEL_3,
              ADC1_PRESSEL_FCPU_D4,
              ADC1_EXTTRIG_TIM,
              DISABLE,
              ADC1_ALIGN_RIGHT,
              ADC1_SCHMITTTRIG_CHANNEL3,
              DISABLE);
    ADC1_Cmd(ENABLE);
    ADC1_StartConversion();
    Sensor_Data_Threshod = 120;
    delay_ms(1000);
    while (1)
    {
        Sensor_Data_Analog = ADC1_GetConversionValue();
        // 获取传感器数据
        if(Sensor_Data_Analog < Sensor_Data_Threshod)
        {
            Sensor_Data_Digital = 0;      // 无结露
            GPIO_WriteHigh(GPIOD, GPIO_PIN_3);
        }
        else
        {
            Sensor_Data_Digital = 1;      // 有结露
            GPIO_WriteLow(GPIOD, GPIO_PIN_3);
        }
         // 组合数据帧
        DATA_tx_buf[10] = Sensor_Data_Digital;
        // 发送数据帧
        UART1_SendString(DATA_tx_buf, 14);
        LED_Toggle();
        delay_ms(1000);
    }
}
```

4. 实验步骤

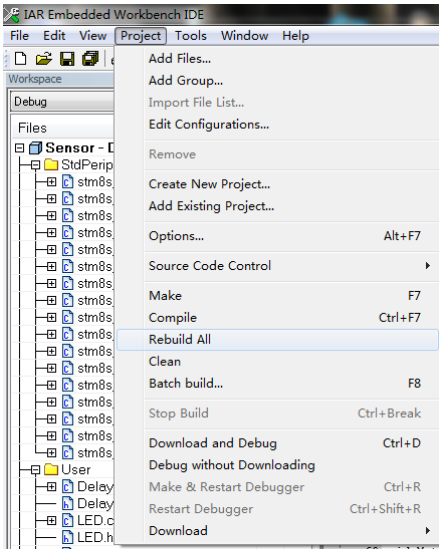
1）首先我们要把传感器插到实验箱的主板上子节点的串口上，再把 ST-Link 配合 JTAG 转接板插到标有 ST-Link 标志的 JTAG 口上，最后把仿真器一端的 USB 线插到 PC 机的 USB 端口，通过主板上的“加”“减”按键选择要编程实验的传感器（会有黄色 LED 灯提示），硬件连接完毕。



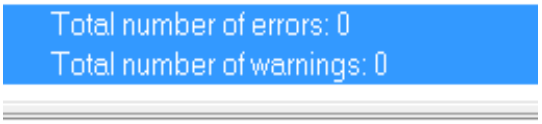
2）用 IAR SWSTM8 1.30 软件，打开..\5-Sensor_结露传感器\Project\Sensor.eww。

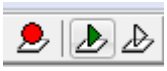


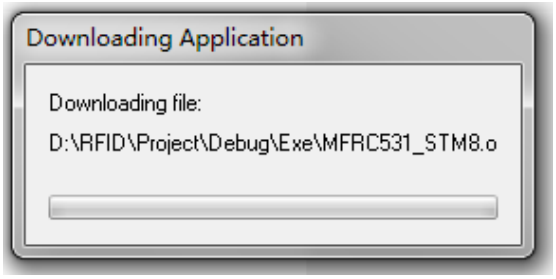
3）打开后点击“Project”下面的“Rebuild All”或者选中工程文件右键“Rebuild All”把我们的工程编译一下。



4）点击“Rebuild All”编译完后，无警告，无错误。



5）编译完后我们要把程序烧到模块里，点击“  ”中间的 Download and Debug 进行烧写。



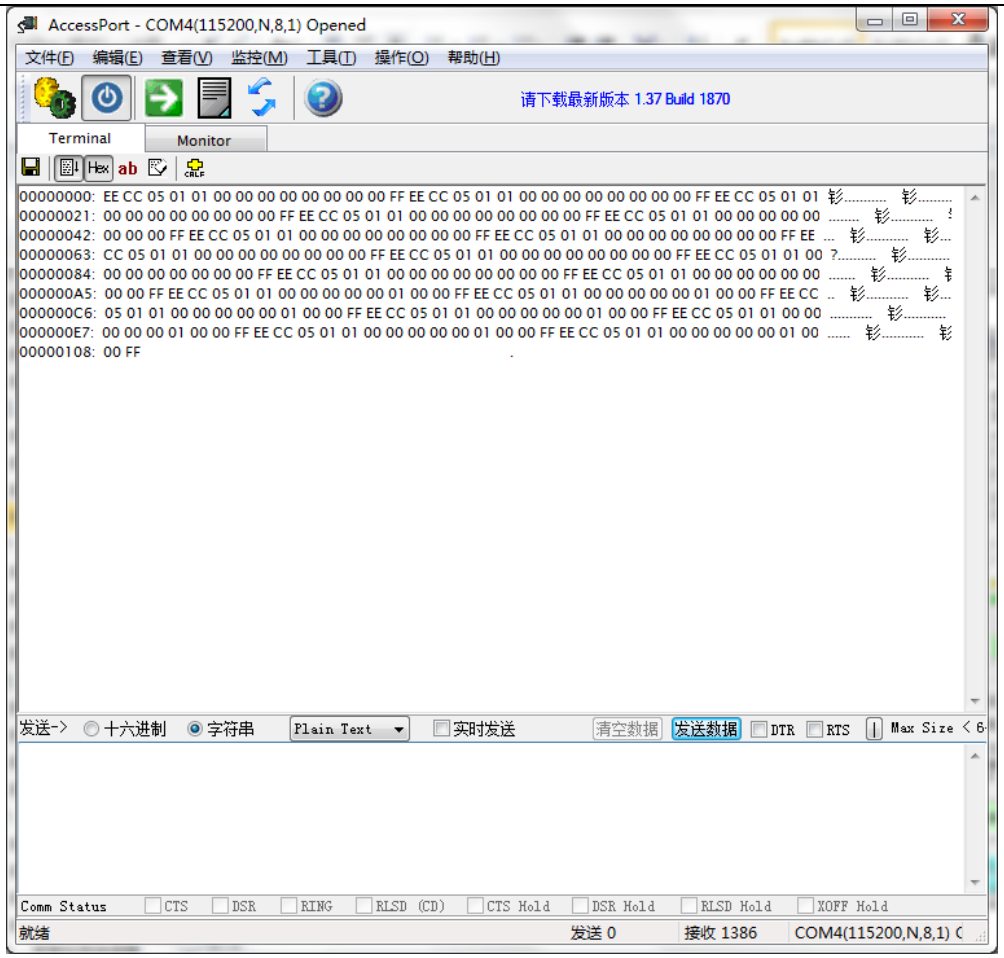
6）烧写完毕后，把传感器模块从主板上取下来，连接到平台配套的 USB 转串口模块上，将 USB2UART 模块的 USB 线连接到 PC 机的 USB 端口，然后打开串口工具，配置好串口，波特率 115200，8 个数据位，一个停止位，无校验位。

7）传感器底层串口协议返回 14 个字节，第 1 位字节和 2 位字节是包头，第 3 位字节是传感器类型，第 4 位字节是传感器 ID,第 5 位字节是节点命令 ID，第 6 位字节到 11 位字节是数据位，其中第 11 位字节是传感器的状态位，第 12 位字节和第 13 位字节是保留位，第 14 位字节是包尾。

例如：返回“EE CC 05 01 01 00 00 00 00 00 00 00 FF”时，第 11 位字节为“0”时，表示无结露，返回“EE CC 05 01 01 00 00 00 00 00 01 00 00 FF”时,第 11 位字节为“1”是表示有结露。

下图为测试结果。

更详细的协议说明，请用户参见《模块通讯协议 V2.6.pdf》文档。


AccessPort - COM4(115200,N,8,1) Opened
文件(F) 编辑(E) 查看(V) 监控(M) 工具(T) 操作(O) 帮助(H)
请下载最新版本 1.37 Build 1870
Terminal
Monitor
00000000: EE CC 05 01 01 00 00 00 00 00 00 00 00 00 FF EE CC 05 01 01 00 00 00 00 00 00 00 00 FF EE CC 05 01 01
00000021: 00 00 00 00 00 00 00 00 FF EE CC 05 01 01 00 00 00 00 00 00 00 00 FF EE CC 05 01 01 00 00 00 00 00
00000042: 00 00 00 FF EE CC 05 01 01 00 00 00 00 00 00 00 00 FF EE CC 05 01 01 00 00 00 00 00 00 00 00 FF EE
00000063: CC 05 01 01 00 00 00 00 00 00 00 00 FF EE CC 05 01 01 00 00 00 00 00 00 00 00 FF EE CC 05 01 01 00
00000084: 00 00 00 00 00 00 00 FF EE CC 05 01 01 00 00 00 00 00 00 00 00 FF EE CC 05 01 01 00 00 00 00 00 00
000000A5: 00 00 FF EE CC 05 01 01 00 00 00 00 00 00 01 00 00 FF EE CC 05 01 01 00 00 00 00 00 01 00 00 FF EE CC
000000C6: 05 01 01 00 00 00 00 00 00 01 00 00 FF EE CC 05 01 01 00 00 00 00 00 00 01 00 00 FF EE CC 05 01 01 00 00
000000E7: 00 00 00 01 00 00 FF EE CC 05 01 01 00 00 00 00 00 00 01 00 00 FF EE CC 05 01 01 00 00 00 00 00 01 00
00000108: 00 FF
发送-> 十六进制 字符串 Plain Text 实时发送 清空数据 发送数据 DTR RTS Max Size
Comm Status CTS DSR RING RLSD (CD) CTS Hold DSR Hold RLSD Hold XOFF Hold
就绪 发送 0 接收 1386 COM4(115200,N,8,1)

实验六. 酒精传感器

1. 实验目的

- 了解 MQ-3 元器件
- 熟悉酒精传感器工作原理。

2. 实验环境

- 软件：IAR SWSTM8 1.30
- 硬件：ICS-IOT-CEP 教学实验平台，酒精传感器模块，USB2UART 模块

3. 实验原理

◆ MQ-3 气体传感器简介

MQ-3 气体传感器所使用的气敏材料是在清洁空气中电导率较低的二氧化锡(SnO2)。当传感器所处环境中存在酒精蒸汽时，传感器的电导率随空气中酒精气体浓度的增加而增大。使用简单的电路即可将电导率的变化转换为与该气体浓度相对应的输出信号。

MQ-3 气体传感器对酒精的灵敏度高，可以抵抗汽油、烟雾、水蒸气的干扰。这种传感器可检测多种浓度酒精气氛，是一款适合多种应用的低成本传感器。

特点：

- 对酒精气体具有良好的灵敏度。
- 长寿命，低成本，简单的驱动电路即可。

MQ-3 气敏元件的内部构造,由微型 Al2O3 陶瓷管、SnO2 敏感层,测量电极和加热器构成的敏感元件固定在塑料或不锈钢制成的腔体内，加热器为气敏元件提供了必要的工作条件。封装好的气敏元件有 6 只针状管脚，其中 4 个用于信号取出， 2 个用于提供加热电流。

◆ 酒精传感器模块原理图

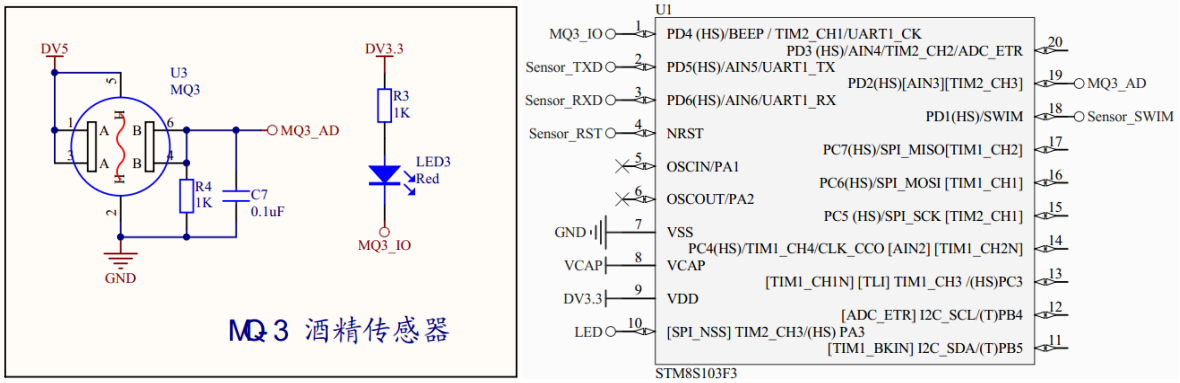


图 3.1 原理图

如图所示，MQ-3 传感器的供电电压 Vc 和加热电压 Vh 都为 5V，负载电阻 R4 为 1K 欧姆。从技术指标表中(参加配套 MQ3 数据手册)可知，在 0.4mg/L 酒精中，传感器电阻 Rs 为 2K~20K，取 Rs = 10K。假设检测到酒精浓度为 10mg/L 时报警，由灵敏度特性曲线可知，MQ3 电阻值为 10K * 0.12 = 1.2K，MQ3_AD = 5V * 1K / (1K + 1.2K) = 2.27V，AD 读数为 2.27 / 3.3 * 1024 = 704，当 STM8 单片机引脚 PD2 AD 采集的数值大于 704 时表明检测到酒精，并点亮 LED3 作为指示。

- **源码分析**

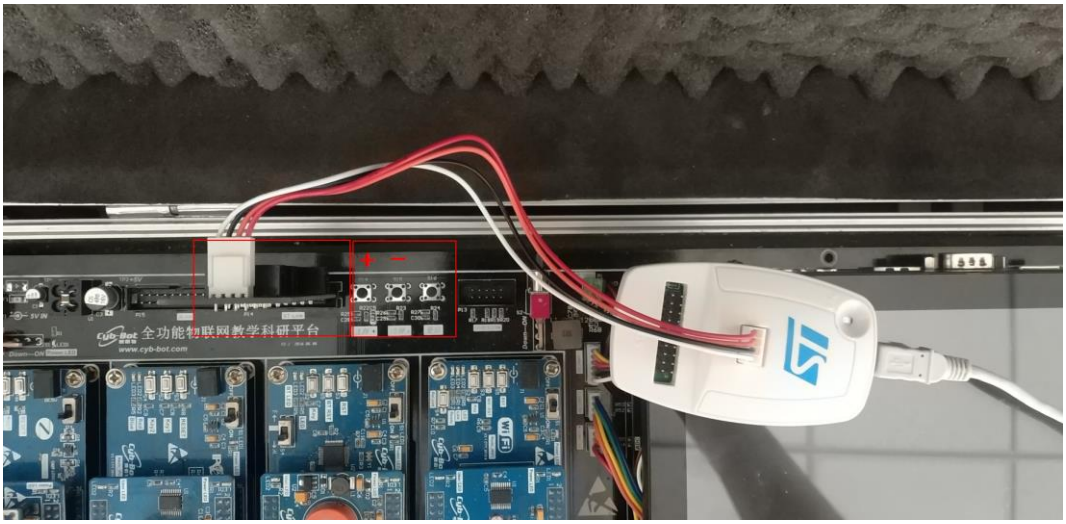
实现代码如下

```
u8 CMD_rx_buf[8];     // 命令缓冲区
u8 DATA_tx_buf[14]; // 返回数据缓冲区
u8 CMD_ID = 0;    // 命令序号
u8 Sensor_Type = 0; // 传感器类型编号
u8 Sensor_ID = 0;     // 相同类型传感器编号
u8 Sensor_Data[6];    // 传感器数据区
u8 Sensor_Data_Digital = 0; // 数字类型传感器数据
u16 Sensor_Data_Analog = 0; // 模拟类型传感器数据
u16 Sensor_Data_Threshod = 0;// 模拟传感器阈值
    /* 根据不同类型的传感器进行修改 */
  Sensor_Type = 6;
  Sensor_ID = 1;
  CMD_ID = 1;
  DATA_tx_buf[0] = 0xEE;
  DATA_tx_buf[1] = 0xCC;
  DATA_tx_buf[2] = Sensor_Type;
  DATA_tx_buf[3] = Sensor_ID;
  DATA_tx_buf[4] = CMD_ID;
  DATA_tx_buf[13] = 0xFF;
  GPIO_Init(GPIOD, GPIO_PIN_4, GPIO_MODE_OUT_PP_HIGH_SLOW);
  // ADC
  ADC1_Init(ADC1_CONVERSIONMODE_CONTINUOUS,
            ADC1_CHANNEL_3,
            ADC1_PRESSEL_FCPU_D4,
            ADC1_EXTTRIG_TIM,
            DISABLE,
            ADC1_ALIGN_RIGHT,
            ADC1_SCHMITTTRIG_CHANNEL3,
            DISABLE);
  ADC1_Cmd(ENABLE);
  ADC1_StartConversion();
  Sensor_Data_Analog = 0;
  Sensor_Data_Threshod = 400;
  delay_ms(1000);
  while (1)
  {
      // 获取传感器数据
      Sensor_Data_Analog = ADC1_GetConversionValue();
      if(Sensor_Data_Analog > Sensor_Data_Threshod)
      {
          Sensor_Data_Digital = 0;      // 无酒精
          GPIO_WriteHigh(GPIOD, GPIO_PIN_4);
      }
```

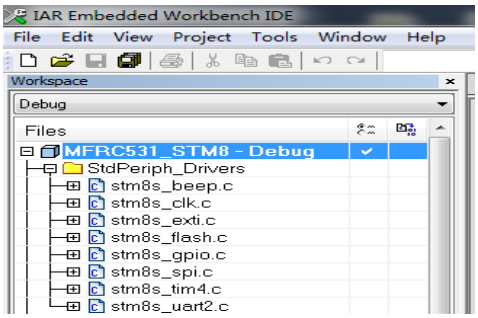
```
            else
            {
                Sensor_Data_Digital = 1;       // 有酒精
                GPIO_WriteLow(GPIOD, GPIO_PIN_4);
            }
             // 组合数据帧
            DATA_tx_buf[10] = Sensor_Data_Digital;
            // 发送数据帧
            UART1_SendString(DATA_tx_buf, 14);
            LED_Toggle();
            delay_ms(1000);
        }
}
```

4. 实验步骤

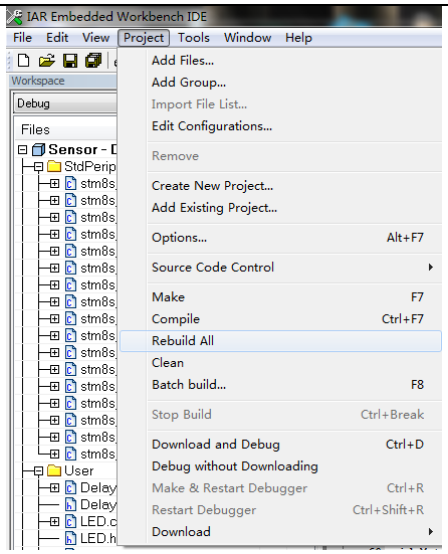
1）首先我们要把传感器插到实验箱的主板上子节点的串口上，再把 ST-Link 配合 JTAG 转接板插到标有 ST-Link 标志的 JTAG 口上，最后把仿真器一端的 USB 线插到 PC 机的 USB 端口，通过主板上的“加”“减”按键选择要编程实验的传感器（会有黄色 LED 灯提示），硬件连接完毕。



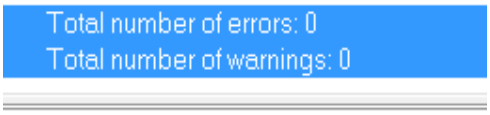
2）我们用 IAR SWSTM8 1.30 软件，打开..\6-Sensor_酒精传感器\Project\Sensor.eww。



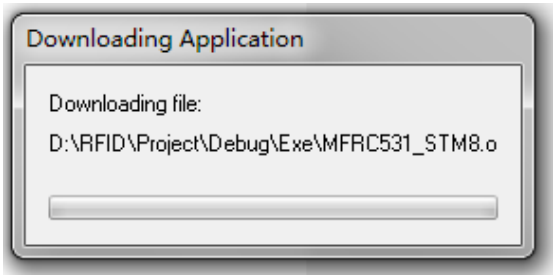
3）打开后点击“Project”下面的“Rebuild All”或者选中工程文件右键“Rebuild All”把我们的工程编译一下。



4）点击“Rebuild All”编译完后，无警告，无错误。



5）编译完后我们要把程序烧到模块里，点击“  ”中间的 Download and Debug 进行烧写。



6）烧写完毕后，把传感器模块从主板上取下来，连接到平台配套的 USB 转串口模块上，将 USB2UART 模块的 USB 线连接到 PC 机的 USB 端口，然后打开串口工具，配置好串口，波特率 115200，8 个数据位，一个停止位，无校验位。

7）传感器底层串口协议返回 14 个字节，第 1 位字节和 2 位字节是包头，第 3 位字节是传感器类型，第 4 位字节是传感器 ID,第 5 位字节是节点命令 ID，第 6 位字节到 11 位字节是数据位，其中第 11 位字节是传感器的状态位，第 12 位字节和第 13 位字节是保留位，第 14 位字节是包尾。

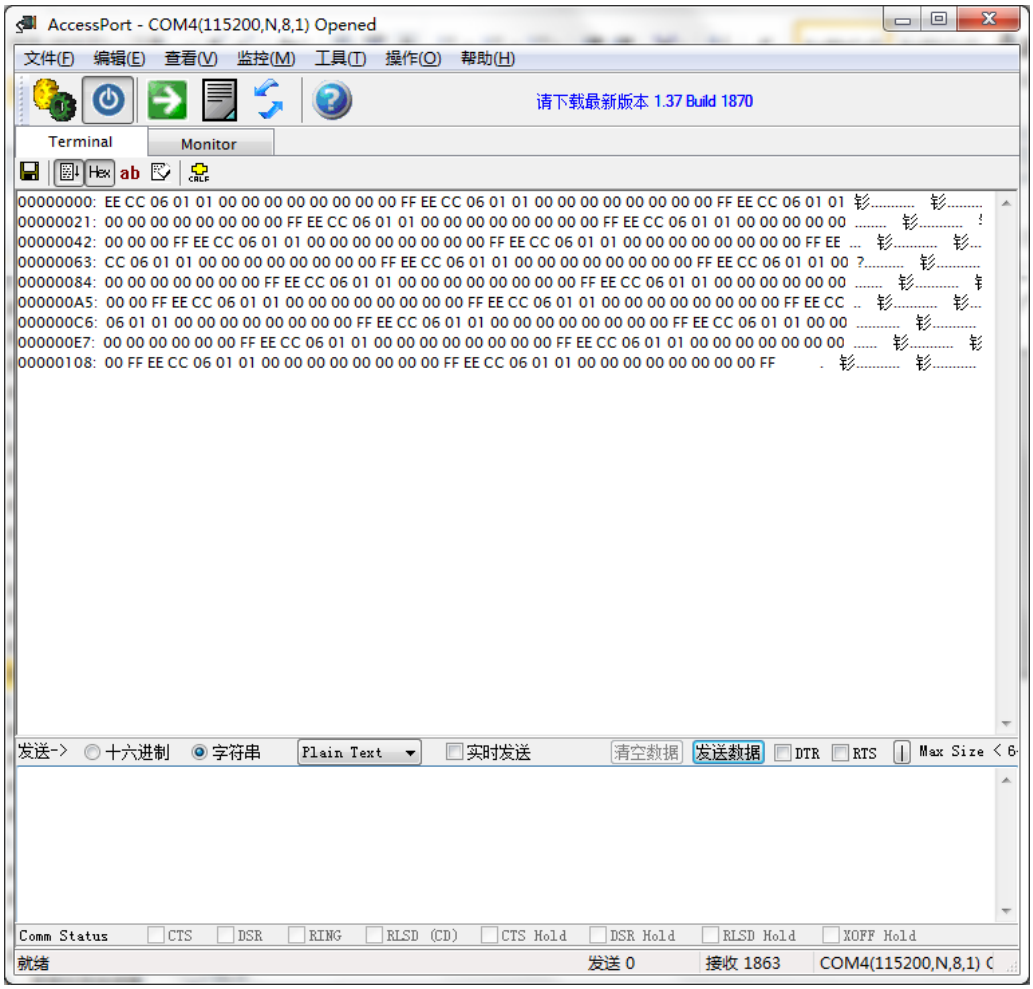
例如：返回“EE CC 06 01 01 00 00 00 00 00 00 00 FF”时，第 11 位字节为“0”时，表示无酒精，返回“EE CC 06 01 01 00 00 00 00 00 01 00 00 FF”时,第 11 位字节为“1”是表示有酒精。

```
/* 根据不同类型的传感器进行修改 */
    Sensor_Type = 1;    //传感器类型
    Sensor_ID = 1;      //传感器 ID
    CMD_ID = 1;         //节点命令 ID
    DATA_tx_buf[0] = 0xEE;//包头
    DATA_tx_buf[1] = 0xCC;//包头
```

```
DATA_tx_buf[2] = Sensor_Type;
DATA_tx_buf[3] = Sensor_ID;
DATA_tx_buf[4] = CMD_ID;
DATA_tx_buf[13] = 0xFF;//包尾
```

下图为测试结果。

更详细的协议说明，请用户参见《模块通讯协议 V2.6.pdf》文档。



实验七. 人体检测传感器

1. 实验目的

- 了解 DYP-ME003。
- 掌握人体检测传感器工作原理。

2. 实验环境

- 软件：IAR SWSTM8 1.30
- 硬件：ICS-IOT-CEP 教学实验平台，人体监测传感器模块，USB2UART 模块

3. 实验原理

♦ 人体红外线感简介

人体红外线感应模块是基于红外线技术的自动控制产品，灵敏度高，可靠性强，超低电压工作模式。广泛应用于各类自动感应电器设备，尤其是干电池供电的自动控制产品。

电气参数	DYP-ME003 人体感应模块
工作电压范围	DC 4.5-20V
静态电流	<50uA
电平输出	高 3.3 V /低 0V
触发方式	L 不可重复触发/H 重复触发
延时时间	5S(默认)可制作范围零点几秒-几十分钟
封锁时间	2.5S(默认)可制作范围零点几秒-几十秒
电路板外形尺寸	32mm*24mm
感应角度	<100 度锥角
感应距离	7 米以内
工作温度	-15-+70 度
感应透镜尺寸	直径:23mm(默认)

图 3.1 参数

♦ 人体红外线感使用说明

1）感应模块通电后有一分钟左右的初始化时间，在此期间模块会间隔地输出 0-3 次，一分钟后进入待机状态。

2）应尽量避免灯光等干扰源近距离直射模块表面的透镜，以免引进干扰信号产生误动作；使用环境尽量避免流动的风，风也会对感应器造成干扰。

3）感应模块采用双元探头，探头的窗口为长方形，双元（A 元 B 元）位于较长方向的

两端，当人体从左到右或从右到左走过时,红外光谱到达双元的时间、距离有差值，差值越大，感应越灵敏，当人体从正面走向探头或从上到下或从下到上方向走过时，双元检测不到红外光谱距离的变化，无差值，因此感应不灵敏或不工作；所以安装感应器时应使探头双元的方向与人体活动最多的方向尽量相平行，保证人体经过时先后被探头双元所感应。为了增加感应角度范围，本模块采用圆形透镜，也使得探头四面都感应，但左右两侧仍然比上下两个方向感应范围大、灵敏度强，安装时仍须尽量按以上要求。

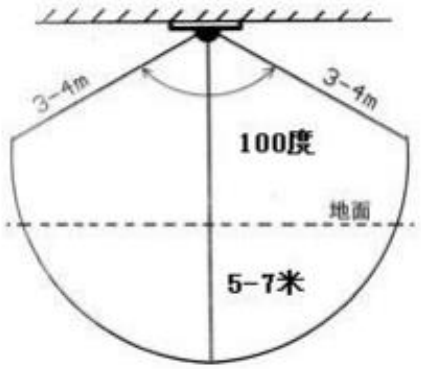


图 3.2 感应范围

- **人体检测传感器模块原理图**

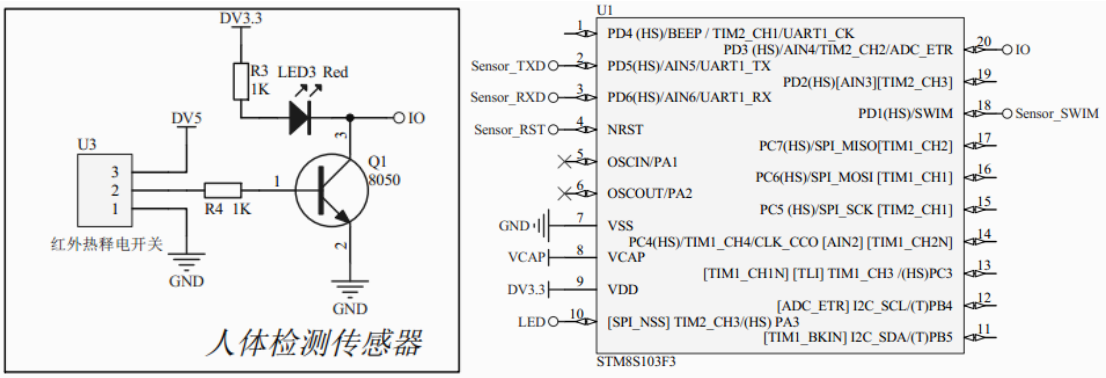


图 3.3 原理图

原理图所示，传感器检测到人时，输出高电平，Q1 导通，IO 输出低电平；未检测到人时，Q1 截止，IO 输出高电平。通过 STM8 单片机读取 PD3 IO 值可知现在的传感器状态。热释电人体红外线感应模块只对人体活动产生感应信号，对静止的人体不做反应，因此，使用时在模块上方挥舞手模拟人体活动。

- **源码分析**

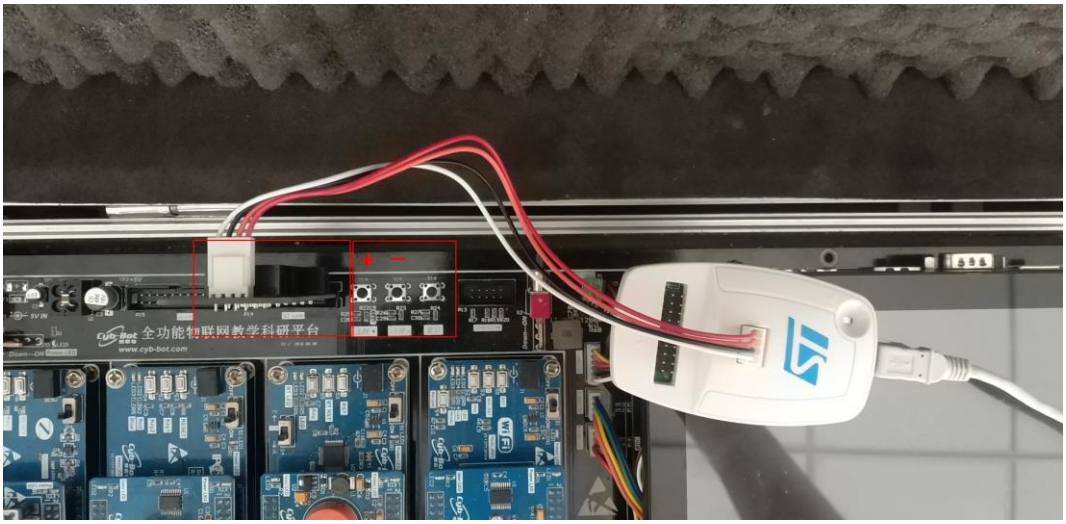
实现代码如下：

```
u8 CMD_rx_buf[8];     // 命令缓冲区
u8 DATA_tx_buf[14]; // 返回数据缓冲区
u8 CMD_ID = 0;    // 命令序号
u8 Sensor_Type = 0; // 传感器类型编号
u8 Sensor_ID = 0;     // 相同类型传感器编号
u8 Sensor_Data[6];    // 传感器数据区
u8 Sensor_Data_Digital = 0; // 数字类型传感器数据
u16 Sensor_Data_Analog = 0; // 模拟类型传感器数据
```

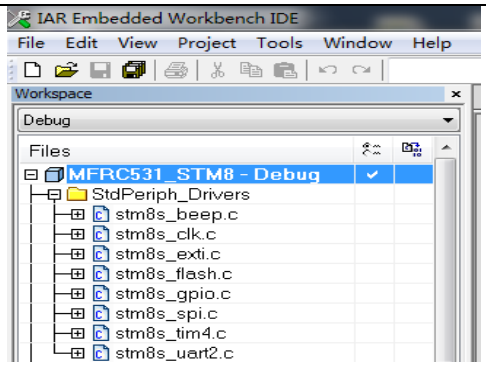
```
u16 Sensor_Data_Threshod = 0;// 模拟传感器阈值
/* 根据不同类型的传感器进行修改 */
    Sensor_Type = 7;
    Sensor_ID = 1;
    CMD_ID = 1;
    DATA_tx_buf[0] = 0xEE;
    DATA_tx_buf[1] = 0xCC;
    DATA_tx_buf[2] = Sensor_Type;
    DATA_tx_buf[3] = Sensor_ID;
    DATA_tx_buf[4] = CMD_ID;
    DATA_tx_buf[13] = 0xFF;
     delay_ms(1000);
     while (1)
    {
        // 获取传感器数据
        if(GPIO_ReadInputPin(GPIOD, GPIO_PIN_3))
            Sensor_Data_Digital = 0;      // 无人
        else
            Sensor_Data_Digital = 1;      // 有人
        // 组合数据帧
        DATA_tx_buf[10] = Sensor_Data_Digital;
            // 发送数据帧
        UART1_SendString(DATA_tx_buf, 14);
        LED_Toggle();
        delay_ms(1000);
    }
```

4. 实验步骤

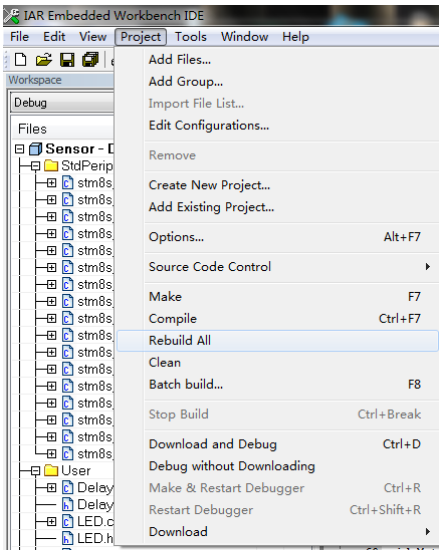
1）首先我们要把传感器插到实验箱的主板上子节点的串口上，再把 ST-Link 配合 JTAG 转接板插到标有 ST-Link 标志的 JTAG 口上，最后把仿真器一端的 USB 线插到 PC 机的 USB 端口，通过主板上的“加”“减”按键选择要编程实验的传感器（会有黄色 LED 灯提示），硬件连接完毕。



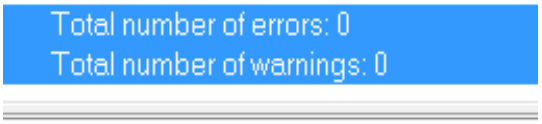
2）我们用 IAR SWSTM8 1.30 软件，打开..\7-Sensor_人体检测传感器\Project\Sensor.eww。

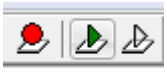


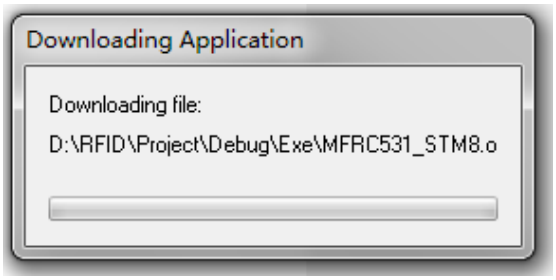
3）打开后点击“Project”下面的“Rebuild All”或者选中工程文件右键“Rebuild All”把我们的工程编译一下。



4）点击“Rebuild All”编译完后，无警告，无错误。



5）编译完后我们要把程序烧到模块里，点击“  ”中间的 Download and Debug 进行烧写。



6）烧写完毕后，把传感器模块从主板上取下来，连接到平台配套的 USB 转串口模块上，将 USB2UART 模块的 USB 线连接到 PC 机的 USB 端口，然后打开串口工具，配置好串口，波特率 115200，8 个数据位，一个停止位，无校验位。

7）传感器底层串口协议返回 14 个字节，第 1 位字节和 2 位字节是包头，第 3 位字节是

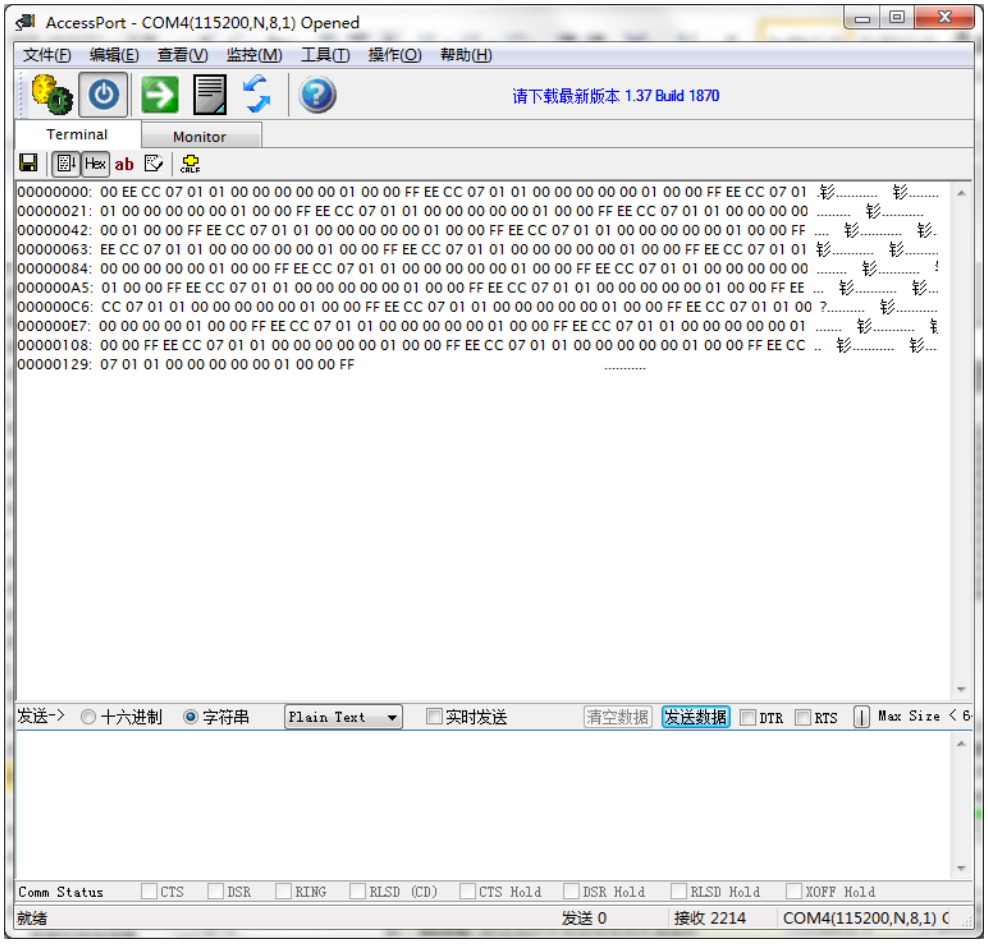
传感器类型，第 4 位字节是传感器 ID,第 5 位字节是节点命令 ID，第 6 位字节到 11 位字节是数据位，其中第 11 位字节是传感器的状态位，第 12 位字节和第 13 位字节是保留位，第 14 位字节是包尾。

例如：返回“EE CC 07 01 01 00 00 00 00 00 00 00 FF”时，第 11 位字节为“0”时，表示无人，返回“EE CC 07 01 01 00 00 00 00 00 01 00 00 FF”时,第 11 位字节为“1”是表示有人。

```
/* 根据不同类型的传感器进行修改 */
    Sensor_Type = 1;     //传感器类型
    Sensor_ID = 1;       //传感器 ID
    CMD_ID = 1;          //节点命令 ID
    DATA_tx_buf[0] = 0xEE;//包头
    DATA_tx_buf[1] = 0xCC;//包头
    DATA_tx_buf[2] = Sensor_Type;
    DATA_tx_buf[3] = Sensor_ID;
    DATA_tx_buf[4] = CMD_ID;
    DATA_tx_buf[13] = 0xFF;//包尾
```

下图为测试结果

更详细的协议说明，请用户参见《模块通讯协议 V2.6.pdf》文档。



实验八. 三轴加速度传感器

1. 实验目的

- 了解三轴加速度传感器器。
- 掌握三轴加速度传感器工作原理。

2. 实验环境

- 软件：IAR SWSTM8 1.30
- 硬件：ICS-IOT-CEP 教学实验平台，三轴加速度传感器模块，USB2UART 模块

3. 实验原理

♦ ADXL345 简介

ADXL345 是一款小而薄的超低功耗 3 轴加速度计，分辨率高(13 位)，测量范围达±16g 。数字输出数据为 16 位二进制补码格式，可通过 SPI(3 线或 4 线)或 I2C 数字接口访问。

ADXL345 非常适合移动设备应用。它可以在倾斜检测应用中测量静态重力加速度，还可以测量运动或冲击导致的动态加速度。其高分辨率(3.9mg/LSB)，能够测量不到 1.0° 的倾斜角度变化。该器件提供多种特殊检测功能。活动和非活动检测功能通过比较任意轴上的加速度与用户设置的阈值来检测有无运动发生。敲击检测功能可以检测任意方向的单振和双振动作。自由落体检测功能可以检测器件是否正在掉落。这些功能可以独立映射到两个中断输出引脚中的一个。正在申请专利的集成式存储器管理系统采用一个 32 级先进先出 (FIFO)缓冲器，可用于存储数据，从而将主机处理器负荷降至最低，并降低整体系统功耗。低功耗模式支持基于运动的智能电源管理，从而以极低的功耗进行阈值感测和运动加速度测量。ADXL345 采用 3 mm × 5 mm × 1 mm，14 引脚小型超薄塑料封装。

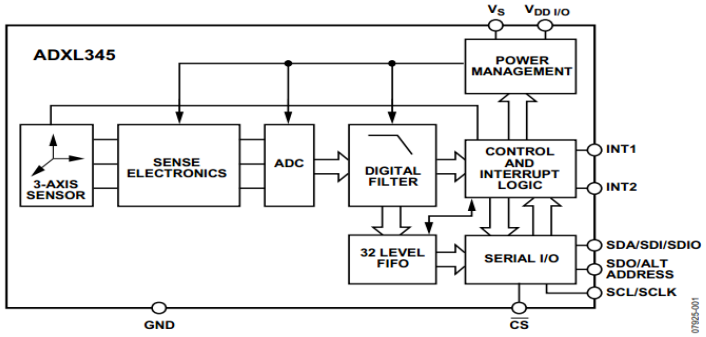


图 3.1 功能框图

ADXL345 是一款完整的 3 轴加速度测量系统，可选择的测量范围有±2 g ，±4 g ，±8 g 或±16 g。既能测量运动或冲击导致的动态加速度，也能测量静止加速度，例如重力加速度，使得器件可作为倾斜传感器使用。该传感器为多晶硅表面微加工结构，置于晶圆顶

部。由于应用加速度，多晶硅弹簧悬挂于晶圆表面的结构之上，提供力量阻力。差分电容由独立固定板和活动质量连接板组成，能对结构偏转进行测量。加速度使惯性质量偏转、差分电容失衡，从而传感器输出的幅度与加速度成正比。相敏解调用于确定加速度的幅度和极性。

参数	额定值
加速度	
任意轴，无电	10,000g
任意轴，有电	10,000g
V_S	-0.3 V至+3.9 V
V_{DD} I/O	-0.3 V至+3.9 V
数字引脚	−0.3 V至V_{DD} I/O+ 0.3 V或3.9 V，取较小者
所有其他引脚	-0.3 V至+3.9 V
输出短路持续时间(任意引脚接地)	未定
温度范围	
有电	-40℃至+105℃
存储	-40℃至+105℃

图 3.2 绝对最大额定值

注意，超出上述绝对最大额定值可能会导致器件永久性损坏。这只是额定最值，不表示在这些条件下或者在任何其它超出本技术规范操作章节中所示规格的条件下，器件能够正常工作。长期在绝对最大额定值条件下工作会影响器件的可靠性。

- **串行通信**

可采用和 SPI 数字通信。上述两种情况下，ADXL345 作为从机运行。CS 引脚上拉至 VDD I/O，I2C 模式使能。CS 引脚应始终上拉至 VDD I/O 或由外部控制器驱动，因为 CS 引脚无连接时，默认模式不存在。因此，如果没有采取这些措施，可能会导致该器件无法通信。SPI 模式下，CS 引脚由总线主机控制。SPI 和 I2C 两种操作模式下，ADXL345 写入期间，应忽略从 ADXL345 传输到主器件的数据。

对于 SPI，可 3 线或 4 线配置，如图 3.3 和图 3.4 的连接图所示。在 DATA_FORMAT 寄存器(地址 0x31)中，选择 4 线模式清除 SPI 位(位 D6)，选择 3 线模式则设置 SPI 位。最大负载为 100 pF 时，最大 SPI 时钟速度为 5 MHz，时序方案按照时钟极性(CPOL)= 1 、时钟相位(CPHA)= 1 执行。如果主处理器的时钟极性和相位配置之前，将电源施加到 ADXL345，CS 引脚应在时钟极性和相位改变之前连接至高电平。使用 3 线 SPI 时，推荐将 SDO 引脚上拉至 VDD I/O 抑或通过 10 kΩ 电阻下拉至接地。

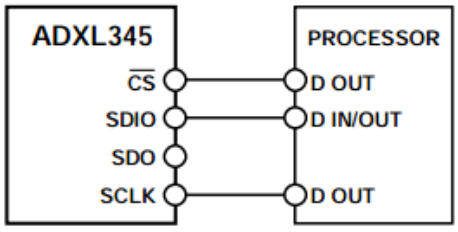


图 3.3

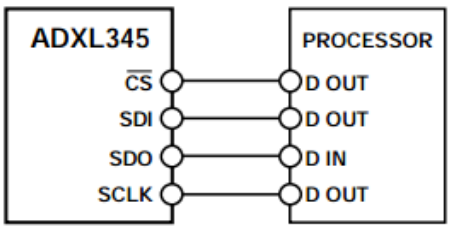


图 3.4

CS 为串行端口使能线，由 SPI 主机控制。如图所示，此线必须在传输起点变为低电平，传输终点变为高电平。SCLK 为串行端口时钟，由 SPI 主机提供。无传输期间，SCLK 为空闲高电平状态。SDI 和 SDO 分别为串行数据输入和输出。SCLK 下降沿时数据更新，SCLK 上升沿时进行采样。要在单次传输内读取或写入多个字节，必须设置位于第一个字节传输(MB ，图 3.5 至图 3.7)R/W 位后的多字节位。寄存器寻址和数据的第一个字节后，时

钟脉冲的随后每次设置 (8 个时钟脉冲) 导致 ADXL345 指向下一个寄存器的读取/ 写入。时钟脉冲停止后，移位才中止，CS 失效。要执行不同不连续寄存器的读取或写入，传输之间 CS 必须失效，新寄存器另行处理。图 3.4 显示了 3 线式 SPI 读取或写入的时序图。图 3.5 和图 3.6 分别显示了 4 线式 SPI 读取和写入的时序图。要进行该器件的正确操作，任何时候都必须满足表 9 和表 10 中的逻辑阈值和时序参数。SPI 通信速率大于或等于 2 MHz 时，推荐采用 3200 Hz 和 1600 Hz 的输出数据速率。只有通信速度大于或等于 400kHz 时，推荐使用 800 Hz 的输出数据速率，剩余的数据传输速率按比例增减。例如，200 Hz 输出数据速率时，推荐的最低通信速度为 100kHz。以高于推荐的最大值输出数据速率运行，可能会对加速度数据产生不良影响，包括采样丢失或额外噪声。

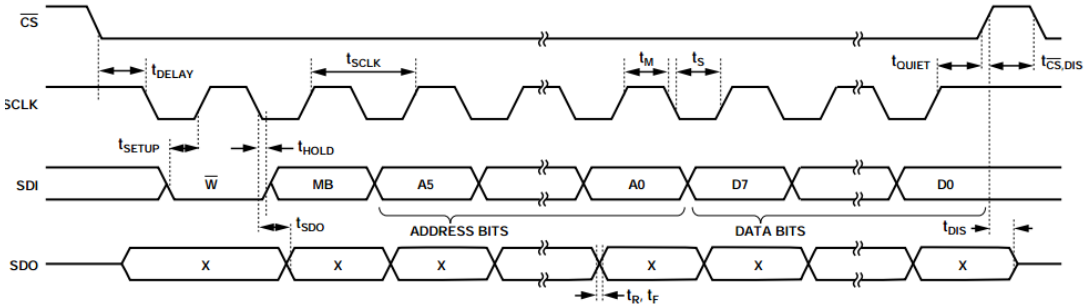


图 3.5 3 线式 SPI 写入

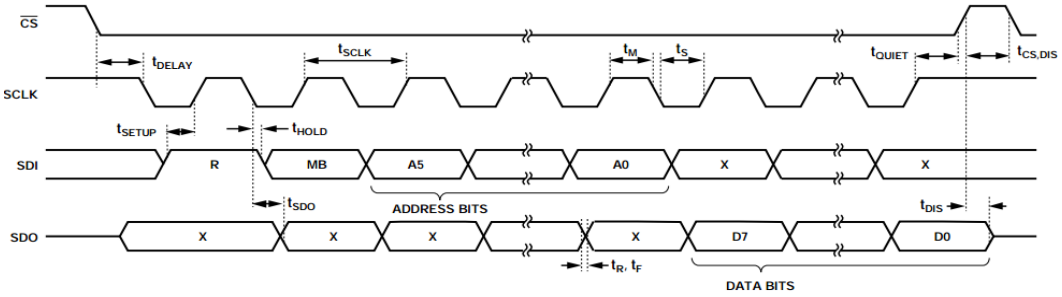


图 3.6 4 线式 SPI 读取

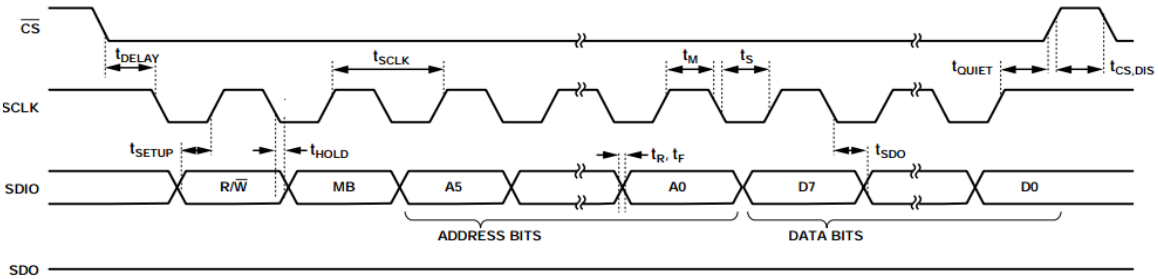


图 3.7 4 线式 SPI 读取/写入

ADXL345 包含嵌入式存储器管理系统，采用 32 位 FIFO，可将主机处理器负荷降至最低。缓冲分四种模式：旁路模式、FIFO 模式、流模式和触发器模式(参见 FIFO 模式) 。在 FIFO_CTL 寄存器(地址 0x38) 内设置 FIFO_MODE 位(位[D7 ： D6]) ，可选择各模式。

在 FIFO 模式下，x、y、z 轴的测量数据存储在 FIFO 中。当 FIFO 中的采样数与 FIFO_CTL 寄存器(地址 0x38)采样数位规定的数量相等时，水印中断置位。FIFO 继续收集样本，直到填满(x、y 和 z 轴测量的 32 位样本) ，然后停止收集数据。FIFO 停止收集数据后，该器件继续工作，因此，FIFO 填满时，敲击检测等功能可以使用。水印中断继续发生，直到 FIFO 样本数少于 FIFO_CTL 寄存器的样本位存储值。

◆ 三轴加速度传感器模块原理图

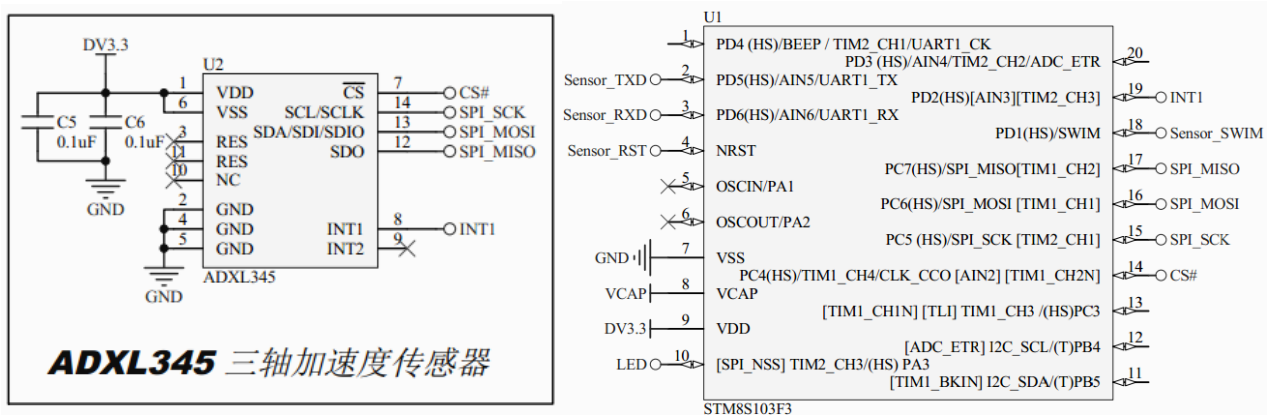


图 3.3 原理图

ADXL345 与 STM8 单片机通过 SPI 接口进行通信，并通过连接 INT1 引脚的 PD2 IO 作为中断输入。

◆ 源码分析

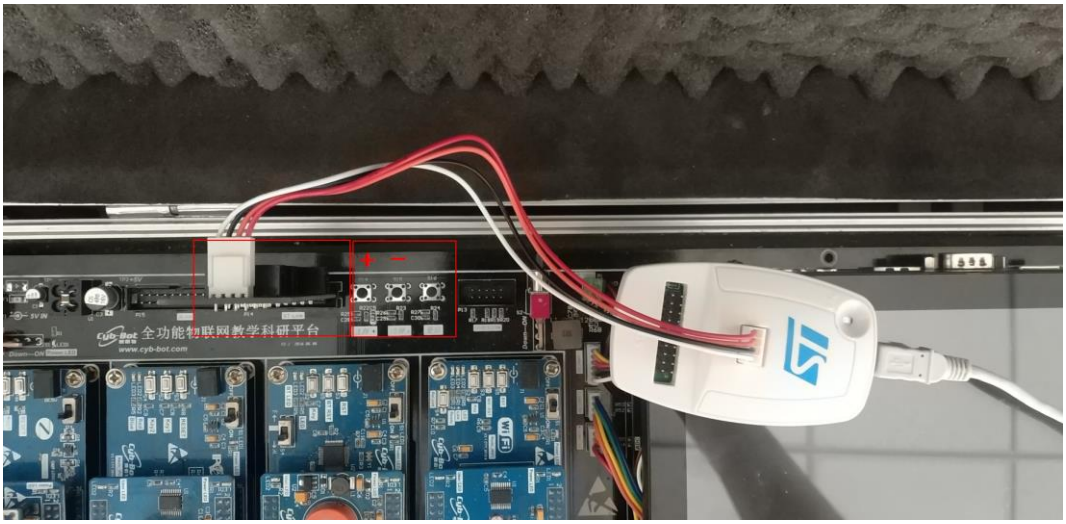
实现代码如下：

```
u8 CMD_rx_buf[8];     // 命令缓冲区
u8 DATA_tx_buf[14]; // 返回数据缓冲区
u8 CMD_ID = 0;    // 命令序号
u8 Sensor_Type = 0; // 传感器类型编号
u8 Sensor_ID = 0;     // 相同类型传感器编号
u8 Sensor_Data[6];    // 传感器数据区
u8 Sensor_Data_Digital = 0; // 数字类型传感器数据
u16 Sensor_Data_Analog = 0; // 模拟类型传感器数据
u16 Sensor_Data_Threshod = 0;// 模拟传感器阈值
/* 根据不同类型的传感器进行修改 */
    Sensor_Type = 8;
    Sensor_ID = 1;
    CMD_ID = 1;
    DATA_tx_buf[0] = 0xEE;
    DATA_tx_buf[1] = 0xCC;
    DATA_tx_buf[2] = Sensor_Type;
    DATA_tx_buf[3] = Sensor_ID;
    DATA_tx_buf[4] = CMD_ID;
    DATA_tx_buf[13] = 0xFF;
    delay_ms(1000);
    while (1)
    {
        // 获取传感器数据
        ADXL345_MultiRead(ADXL345_DATAX0, 6, Sensor_Data);
        // 组合数据帧
        DATA_tx_buf[5] = Sensor_Data[1];
        DATA_tx_buf[6] = Sensor_Data[0];
        DATA_tx_buf[7] = Sensor_Data[3];
        DATA_tx_buf[8] = Sensor_Data[2];
        DATA_tx_buf[9] = Sensor_Data[5];
        DATA_tx_buf[10] = Sensor_Data[4];
        // 发送数据帧
```

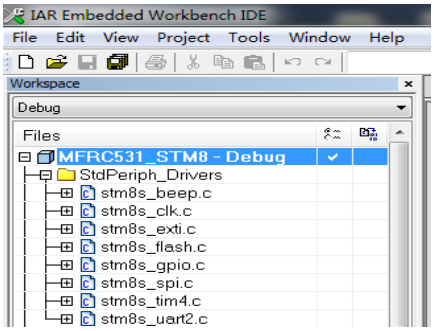
```
            UART1_SendString(DATA_tx_buf, 14);
            LED_Toggle();
            delay_ms(1000);
        }
void SPI_Config(void)     //SPI 配置
{
        SPI_DeInit();
        // SCK - PC5 ; MOSI - PC6 ; MISO - PC7
        // SPI initial
        SPI_Init(SPI_FIRSTBIT_MSB,
                    SPI_BAUDRATEPRESCALER_16,
                    SPI_MODE_MASTER,
                    SPI_CLOCKPOLARITY_HIGH,
                    SPI_CLOCKPHASE_2EDGE,
                    SPI_DATADIRECTION_2LINES_FULLDUPLEX,
                    SPI_NSS_SOFT,
                    0x07);
        SPI_Cmd(ENABLE);
}
void ADXL345_Init(void)
{
        // CS - PC4
        GPIO_Init(ADXL345_CS_PORT, ADXL345_CS_PIN, GPIO_MODE_OUT_PP_HIGH_FAST);
        ADXL345_SPI_DIS();
        SPI_Config();
        delay_ms(100);
        /*****************************ADXL345 初始化设置*****************************/
        ADXL345_WriteReg(ADXL345_DATA_FORMAT,0x2B);
//数据通信格式;设置为自检功能禁用,4 线制 SPI 接口,低电平中断输出,13 位全分辨率,输出数据右对齐,16g 量程
        ADXL345_WriteReg(ADXL345_OFSX,0x00); //X 轴误差补偿; (15.6mg/LSB)
        ADXL345_WriteReg(ADXL345_OFSY,0x00); //Y 轴误差补偿; (15.6mg/LSB)
        ADXL345_WriteReg(ADXL345_OFSZ,0x00); //Z 轴误差补偿; (15.6mg/LSB)
        ADXL345_WriteReg(ADXL345_BW_RATE,0x0E); //1600Hz
        ADXL345_WriteReg(ADXL345_POWER_CTL,0x28); //开启 Link,测量功能;关闭自动休眠,休眠,唤醒功能
        ADXL345_WriteReg(ADXL345_INT_ENABLE,0x00); //所有中断均关闭
        ADXL345_WriteReg(ADXL345_INT_MAP,0x00); //INT1
        ADXL345_WriteReg(ADXL345_FIFO_CTL,0x00); //不使用 FIFO
}
```

4. 实验步骤

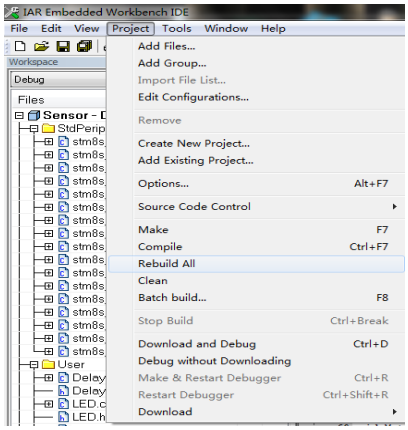
1）首先我们要把传感器插到实验箱的主板上子节点的串口上，再把 ST-Link 配合 JTAG 转接板插到标有 ST-Link 标志的 JTAG 口上，最后把仿真器一端的 USB 线插到 PC 机的 USB 端口，通过主板上的“加”“减”按键选择要编程实验的传感器（会有黄色 LED 灯提示），硬件连接完毕。



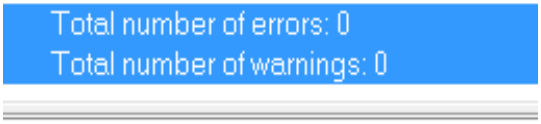
2）我们用 IAR SWSTM8 1.30 软件，打开..\8-Sensor_三轴加速度传感器\Project\Sensor.eww。



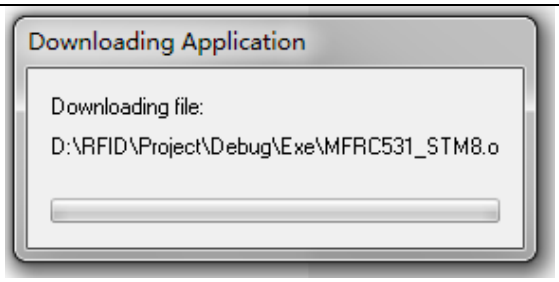
3）打开后点击“Project”下面的“Rebuild All”或者选中工程文件右键“Rebuild All”把我们的工程编译一下。



4）点击“Rebuild All”编译完后，无警告，无错误。



5）编译完后我们要把程序烧到模块里，点击“  ”中间的 Download and Debug 进行烧写。



6）烧写完毕后，把传感器模块从主板上取下来，连接到平台配套的 USB 转串口模块上，将 USB2UART 模块的 USB 线连接到 PC 机的 USB 端口，然后打开串口工具，配置好串口，波特率 115200，8 个数据位，一个停止位，无校验位。

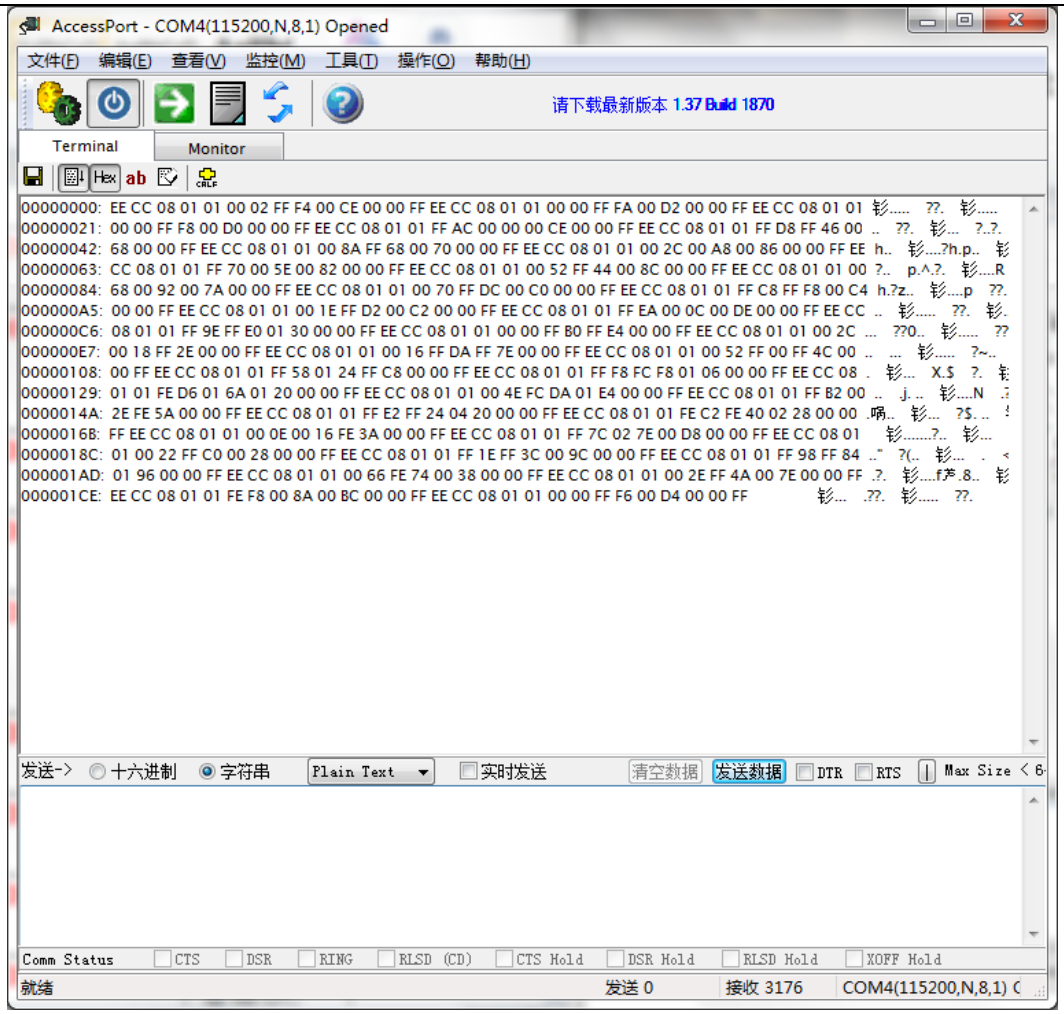
7）传感器底层串口协议返回 14 个字节，第 1 位字节和 2 位字节是包头，第 3 位字节是传感器类型，第 4 位字节是传感器 ID,第 5 位字节是节点命令 ID，第 6 位字节到 11 位字节是数据位，其中第 11 位字节是传感器的状态位，第 12 位字节和第 13 位字节是保留位，第 14 位字节是包尾。

例如：返回 EE CC 08 01 01 XH XL YH YL ZH ZL 00 00 FF，XH XL 代表 X 轴的动态，YH YL 代表 Y 轴的动态，ZH ZL 代表 Z 轴的动态。

```
/* 根据不同类型的传感器进行修改 */
    Sensor_Type = 1;     //传感器类型
    Sensor_ID = 1;       //传感器 ID
    CMD_ID = 1;          //节点命令 ID
    DATA_tx_buf[0] = 0xEE;//包头
    DATA_tx_buf[1] = 0xCC;//包头
    DATA_tx_buf[2] = Sensor_Type;
    DATA_tx_buf[3] = Sensor_ID;
    DATA_tx_buf[4] = CMD_ID;
    DATA_tx_buf[13] = 0xFF;//包尾
```

下图为测试结果

更详细的协议说明，请用户参见《模块通讯协议 V2.6.pdf》文档。


AccessPort - COM4(115200,N,8,1) Opened
文件(F) 编辑(E) 查看(V) 监控(M) 工具(T) 操作(O) 帮助(H)
请下载最新版本 1.37 Build 1870
Terminal
Monitor
00000000: EE CC 08 01 01 00 02 FF F4 00 CE 00 00 FF EE CC 08 01 01 00 00 FF FA 00 D2 00 00 FF EE CC 08 01 01
00000021: 00 00 FF F8 00 D0 00 00 FF EE CC 08 01 01 FF AC 00 00 00 CE 00 00 FF EE CC 08 01 01 FF D8 FF 46 00
00000042: 68 00 00 FF EE CC 08 01 01 00 8A FF 68 00 70 00 00 FF EE CC 08 01 01 00 2C 00 A8 00 86 00 00 FF EE
00000063: CC 08 01 01 FF 70 00 5E 00 82 00 00 FF EE CC 08 01 01 00 52 FF 44 00 8C 00 00 FF EE CC 08 01 01 00
00000084: 68 00 92 00 7A 00 00 FF EE CC 08 01 01 00 70 FF DC 00 C0 00 00 FF EE CC 08 01 01 FF C8 FF F8 00 C4
000000A5: 00 00 FF EE CC 08 01 01 00 1E FF D2 00 C2 00 00 FF EE CC 08 01 01 FF EA 00 0C 00 DE 00 00 FF EE CC
000000C6: 08 01 01 FF 9E FF E0 01 30 00 00 FF EE CC 08 01 01 00 00 FF B0 FF E4 00 00 FF EE CC 08 01 01 00 2C
000000E7: 00 18 FF 2E 00 00 FF EE CC 08 01 01 00 16 FF DA FF 7E 00 00 FF EE CC 08 01 01 00 52 FF 00 FF 4C 00
00000108: 00 FF EE CC 08 01 01 FF 58 01 24 FF C8 00 00 FF EE CC 08 01 01 FF F8 FC F8 01 06 00 00 FF EE CC 08
00000129: 01 01 FE D6 01 6A 01 20 00 00 FF EE CC 08 01 01 00 4E FC DA 01 E4 00 00 FF EE CC 08 01 01 FF B2 00
0000014A: 2E FE 5A 00 00 FF EE CC 08 01 01 FF E2 FF 24 04 20 00 00 FF EE CC 08 01 01 FE C2 FE 40 02 28 00 00
0000016B: FF EE CC 08 01 01 00 0E 00 16 FE 3A 00 00 FF EE CC 08 01 01 FF 7C 02 7E 00 D8 00 00 FF EE CC 08 01
0000018C: 01 00 22 FF C0 00 28 00 00 FF EE CC 08 01 01 FF 1E FF 3C 00 9C 00 00 FF EE CC 08 01 01 FF 98 FF 84
000001AD: 01 96 00 00 FF EE CC 08 01 01 00 66 FE 74 00 38 00 00 FF EE CC 08 01 01 00 2E FF 4A 00 7E 00 00 FF
000001CE: EE CC 08 01 01 FE F8 00 8A 00 BC 00 00 FF EE CC 08 01 01 00 00 FF F6 00 D4 00 00 FF
发送-> 十六进制 字符串 Plain Text 实时发送 清空数据 发送数据 DTR RTS Max Size
Comm Status CTS DSR RING RLSD (CD) CTS Hold DSR Hold RLSD Hold XOFF Hold
就绪 发送 0 接收 3176 COM4(115200,N,8,1)

实验九. 声音检测传感器

1. 实验目的

- 了解声音检测传感器。
- 掌握声音检测传感器工作原理。

2. 实验环境

- 软件：IAR SWSTM8 1.30
- 硬件：ICS-IOT-CEP 教学实验平台，声音检测传感器模块，USB2UART 模块

3. 实验原理

◆ 声音检测传感器简介

声音检测传感器使用麦克风（咪头）作为拾音器，经过运算放大器放大，单片机 AD 采集，获取声响强度信号。咪头，是将声音信号转换为电信号的能量转换器件，和喇叭正好相反。我们选用的是驻极体电容式咪头。

驻极体传声器的结构

以全向 MIC,振膜式极环连接式为例

- 防尘网：保护咪头，防止灰尘落到振膜上，防止外部物体刺破振膜，还有短时间的防水作用。
- 外壳：整个咪头的支撑件，其它件封装在外壳之中，是传声器的接地点，还可以起到电磁屏蔽的作用。
- 振膜：是一个声-电转换的主要零件，是一个绷紧的特氟珑塑料薄膜（聚氯乙烯）粘在一个金属薄圆环上,薄膜与金属环接触的一面镀有一层很薄的金属层,薄膜可以充有电荷，也是组成一个可变电容的一个电极板，而且是可以振动的极板。
- 垫片：支撑电容两极板之间的距离，留有间隙，为振膜振动提供一个空间，从而改变电容量。
- 背极板：电容的另一个电极，并且连接到了 FET（场效应管）的 G（栅）极上。
- 铜环：连接极板与 FET（场效应管）的 G（栅）极，并且起到支撑作用。
- 腔体：固定极板和极环，从而防止极板和极环对外壳短路（FET（场效应管）的 S（源极），G（栅）极短路）。
- PCB 组件：装有 FET，电容等器件，同时也起到固定其它器件的作用。
- PIN：有的传声器在 PCB 上带有 PIN（脚），可以通过 PIN 与其他 PCB 焊接在一

起，起连接另外前极式，背极式在结构上也略有不同。

- **声音检测传感器模块原理图**

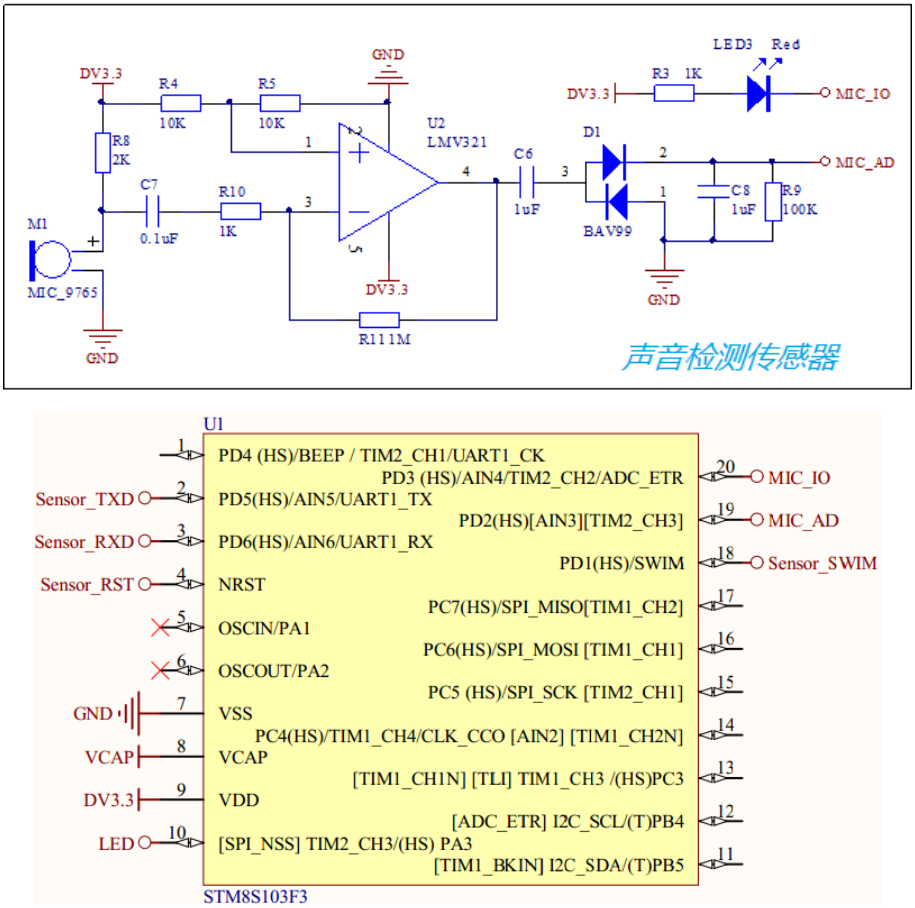


图 3.1 原理图

原理图所示，由于麦克风输出的信号微弱，必须经过运放放大才能保证 AD 采样的精度。麦克风输入的是交流信号，C7 和 C6 用于耦合输入；运放 lmv321 将信号放大了 101 倍，经过 D1 保留交流信号的正向信号，最后输入到单片机 AD 进行采样。在实验室测得，静止条件下，MIC_AD 为 0V；给一个拍手的声响信号，MIC_AD 最大到 1V 左右，此时 AD 值约为 300.因此，取 300 作为临界值，AD 采样值大于 300 时，表明检测到声响，并点亮 LED3 作为指示。

- **源码分析**

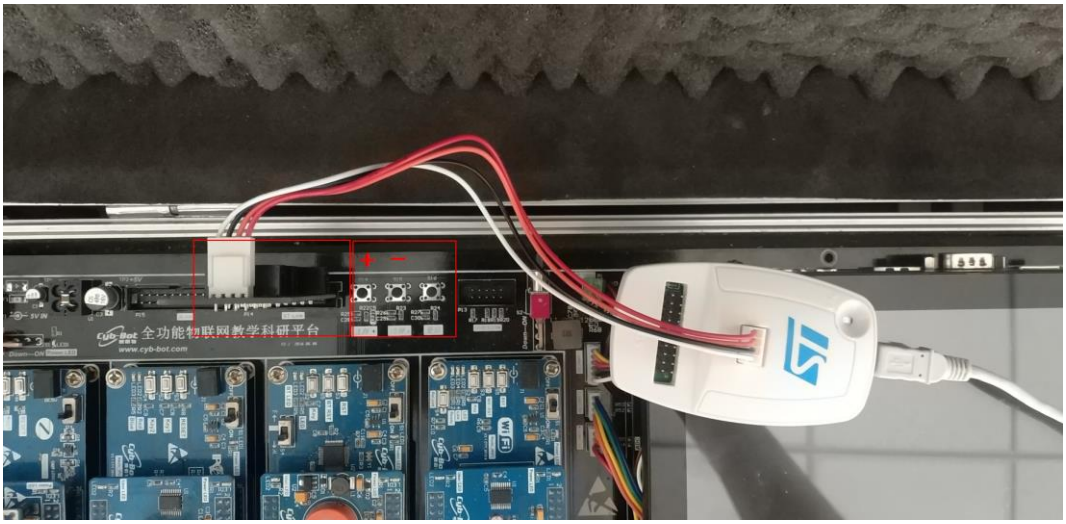
实现代码如下：

```
u8 CMD_rx_buf[8];     // 命令缓冲区
u8 DATA_tx_buf[14]; // 返回数据缓冲区
u8 CMD_ID = 0;    // 命令序号
u8 Sensor_Type = 0; // 传感器类型编号
u8 Sensor_ID = 0;     // 相同类型传感器编号
u8 Sensor_Data[6];    // 传感器数据区
u8 Sensor_Data_Digital = 0; // 数字类型传感器数据
u16 Sensor_Data_Analog = 0; // 模拟类型传感器数据
u16 Sensor_Data_Threshod = 0;// 模拟传感器阈值
```

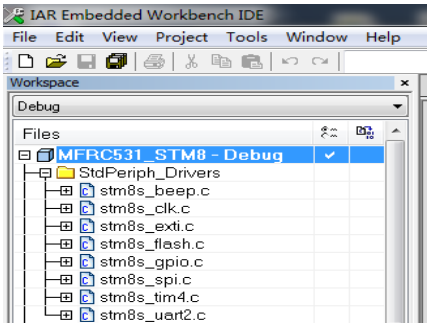
```
/* 根据不同类型的传感器进行修改 */
    Sensor_Type = 9;
    Sensor_ID = 1;
    CMD_ID = 1;
    DATA_tx_buf[0] = 0xEE;
    DATA_tx_buf[1] = 0xCC;
    DATA_tx_buf[2] = Sensor_Type;
    DATA_tx_buf[3] = Sensor_ID;
    DATA_tx_buf[4] = CMD_ID;
    DATA_tx_buf[13] = 0xFF;
    GPIO_Init(GPIOD, GPIO_PIN_3, GPIO_MODE_OUT_PP_HIGH_SLOW);
    // ADC
    ADC1_Init(ADC1_CONVERSIONMODE_CONTINUOUS,
              ADC1_CHANNEL_3,
              ADC1_PRESSEL_FCPU_D4,
              ADC1_EXTTRIG_TIM,
              DISABLE,
              ADC1_ALIGN_RIGHT,
              ADC1_SCHMITTTRIG_CHANNEL3,
              DISABLE);
    ADC1_Cmd(ENABLE);
    ADC1_StartConversion();
    Sensor_Data_Analog = 0;
    Sensor_Data_Threshod = 300;
    enableInterrupts();
    delay_ms(1000);
    while (1)
    {
        // 获取传感器数据
        Sensor_Data_Analog = ADC1_GetConversionValue();
        if(Sensor_Data_Analog > Sensor_Data_Threshod)
        {
            Sound_Cnt = 1;
        }
        delay_ms(100);
    }
```

4. 实验步骤

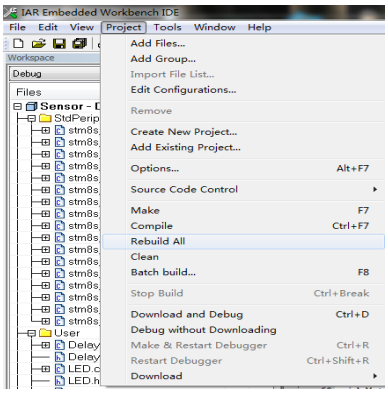
1）首先我们要把传感器插到实验箱的主板上子节点的串口上，再把 ST-Link 配合 JTAG 转接板插到标有 ST-Link 标志的 JTAG 口上，最后把仿真器一端的 USB 线插到 PC 机的 USB 端口，通过主板上的“加”“减”按键选择要编程实验的传感器（会有黄色 LED 灯提示），硬件连接完毕。



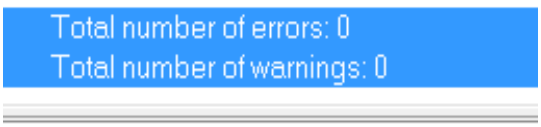
2）我们用 IAR SWSTM8 1.30 软件，打开..\9-Sensor_声音检测传感器\Project\Sensor.eww。

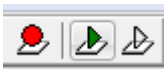


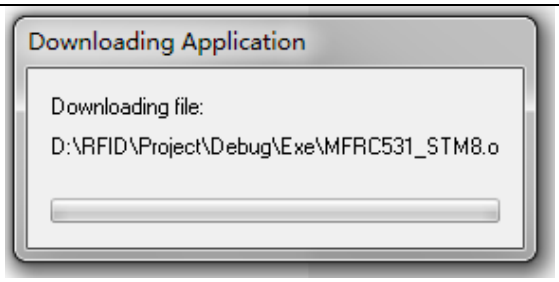
3）打开后点击“Project”下面的“Rebuild All”或者选中工程文件右键“Rebuild All”把我们的工程编译一下。



4）点击“Rebuild All”编译完后，无警告，无错误。



5）编译完后我们要把程序烧到模块里，点击“  ”中间的 Download and Debug 进行烧写。



6）烧写完毕后，把传感器模块从主板上取下来，连接到平台配套的 USB 转串口模块上，将 USB2UART 模块的 USB 线连接到 PC 机的 USB 端口，然后打开串口工具，配置好串口，波特率 115200，8 个数据位，一个停止位，无校验位。

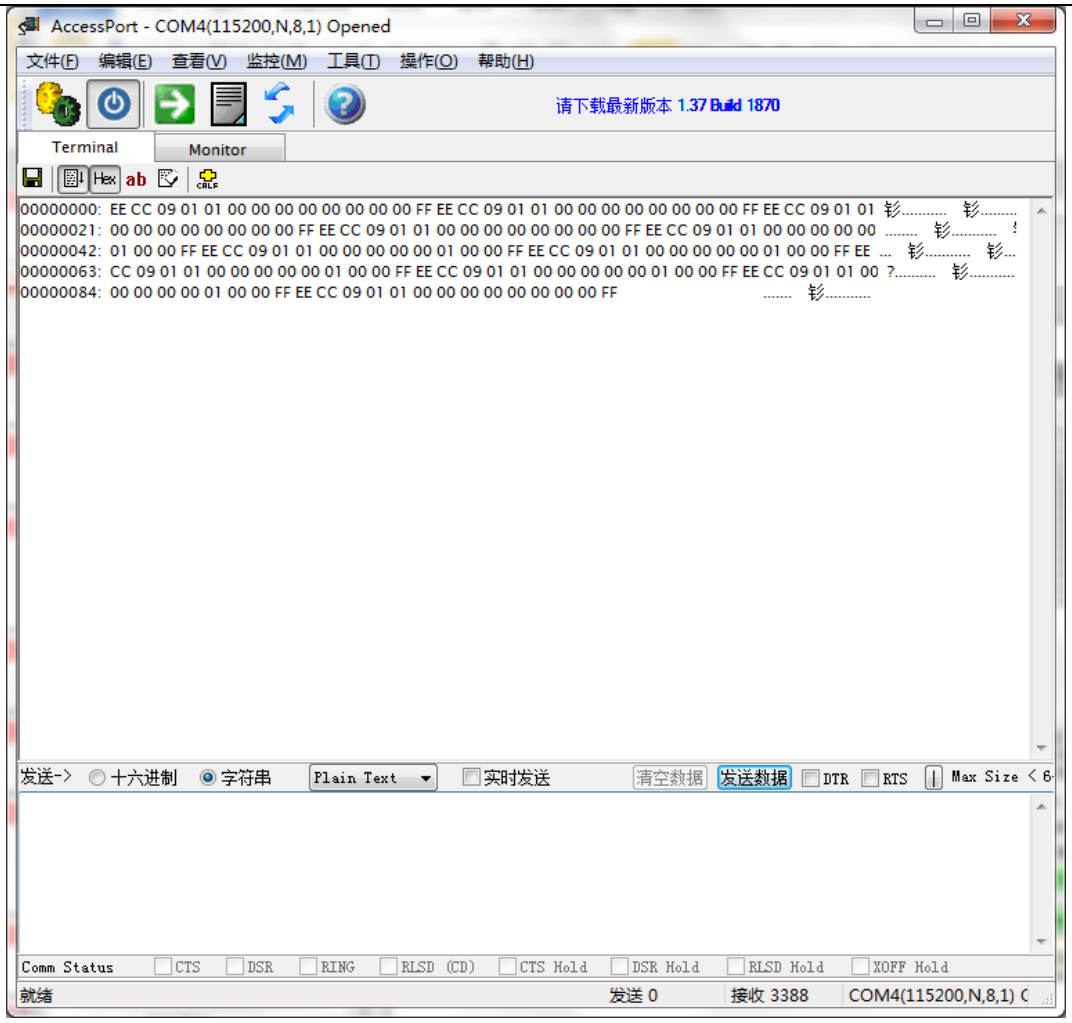
7）传感器底层串口协议返回 14 个字节，第 1 位字节和 2 位字节是包头，第 3 位字节是传感器类型，第 4 位字节是传感器 ID,第 5 位字节是节点命令 ID，第 6 位字节到 11 位字节是数据位，其中第 11 位字节是传感器的状态位，第 12 位字节和第 13 位字节是保留位，第 14 位字节是包尾。

例如：返回“EE CC 09 01 01 00 00 00 00 00 00 00 FF”时，第 11 位字节为“0”时，表示无声， 返回“EE CC 09 01 01 00 00 00 00 00 01 00 00 FF”时,第 11 位字节为“1”是表示有声。

```
/* 根据不同类型的传感器进行修改 */
    Sensor_Type = 1;     //传感器类型
    Sensor_ID = 1;       //传感器 ID
    CMD_ID = 1;          //节点命令 ID
    DATA_tx_buf[0] = 0xEE;//包头
    DATA_tx_buf[1] = 0xCC;//包头
    DATA_tx_buf[2] = Sensor_Type;
    DATA_tx_buf[3] = Sensor_ID;
    DATA_tx_buf[4] = CMD_ID;
    DATA_tx_buf[13] = 0xFF;//包尾
```

下图为测试结果

更详细的协议说明，请用户参见《模块通讯协议 V2.6.pdf》文档。


AccessPort - COM4(115200,N,8,1) Opened
文件(F) 编辑(E) 查看(V) 监控(M) 工具(T) 操作(O) 帮助(H)
请下载最新版本 1.37 Build 1870
Terminal
Monitor
00000000: EE CC 09 01 01 00 00 00 00 00 00 00 00 FF EE CC 09 01 01 00 00 00 00 00 00 00 00 FF EE CC 09 01 01
00000021: 00 00 00 00 00 00 00 00 FF EE CC 09 01 01 00 00 00 00 00 00 00 00 FF EE CC 09 01 01 00 00 00 00 00
00000042: 01 00 00 FF EE CC 09 01 01 00 00 00 00 00 00 01 00 00 FF EE CC 09 01 01 00 00 00 00 00 00 01 00 00 FF EE
00000063: CC 09 01 01 00 00 00 00 00 00 01 00 00 FF EE CC 09 01 01 00 00 00 00 00 00 01 00 00 FF EE CC 09 01 01 00
00000084: 00 00 00 00 01 00 00 FF EE CC 09 01 01 00 00 00 00 00 00 00 00 FF
发送-> 十六进制 字符串 Plain Text 实时发送 清空数据 发送数据 DTR RTS Max Size
Comm Status CTS DSR RING RLSD (CD) CTS Hold DSR Hold RLSD Hold XOFF Hold
就绪 发送 0 接收 3388 COM4(115200,N,8,1)

实验十. 温湿度传感器

1. 实验目的

- 了解温湿度传感器。
- 掌握温湿度传感器工作原理。

2. 实验环境

- 软件：IAR SWSTM8 1.30
- 硬件：ICS-IOT-CEP 教学实验平台，温湿度传感器模块，USB2UART 模块

3. 实验原理

◆ 温湿度传感器简介

AM2302 湿敏电容数字温湿度模块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与卓越的长期稳定性。传感器包括一个电容式感湿元件和一个高精度测温元件，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。每个传感器都在极为精确的湿度校验室中进行校准。校准系数以程序的形式储存在单片机中，传感器内部在检测信号的处理过程中要调用这些校准系数。标准单总线接口，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，使其成为各类应用甚至最为苛刻的应用场合的最佳选择。产品为 3 引线（单总线接口）连接方便。

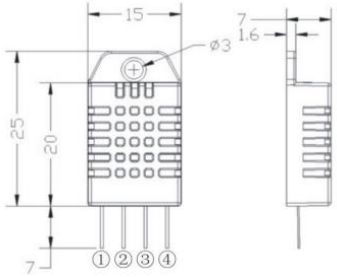


图 3.1 元件大小

引脚	名称	描 述
①	VDD	电源 (3.5V~5.5V)
②	SDA	串行数据，双向口
③	NC	空脚
④	GND	地

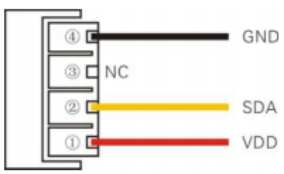


图 3.2 引脚分配

特点

超低能耗、传输距离远、全部自动化校准、采用电容式湿敏元件、完全互换、标准数字单总线输出、卓越的长期稳定性、采用高精度测温元件。

单总线说明

AM2302 器件采用简化的单总线通信。单总线即只有一根数据线，系统中的数据交换、控制均由数据线完成。设备（微处理器）通过一个漏极开路或三态端口连至该数据线，以允许设备在不发送数据时能够释放总线，而让其它设备使用总线；单总线通常要求外接一个约 5.1kΩ的上拉电阻，这样，当总线闲置时，其状态为高电平。由于它们是主从结构，只有主机呼叫传感器时，传感器才会应答，因此主机访问传感器都必须严格遵循单总线序列，如果出现序列混乱，传感器将不响应主机。

单总线通信特殊说明：

- 典型应用电路中建议连接线长度短于 30 米时用 5.1K 上拉电阻，大于 30 米时根据实际情况降低上拉电阻的阻值。
- 使用 3.3V 电压供电时连接线长度不得大于 30cm。否则线路压降会导致传感器供电不足，造成测量偏差。
- 读取传感器最小间隔时间为 2S；读取间隔时间小于 2S，可能导致温湿度不准或通信不成功等情况。
- 每次读出的温湿度数值是上一次测量的结果，欲获取实时数据，需连续读取两次，建议连续多次读取传感器，且每次读取传感器间隔大于 2 秒即可获得准确的数据。

名 称	单总线格式定义
起始信号	微处理器把数据总线（SDA）拉低一段时间(至少 800μs)[1]，通知传感器准备数据。
响应信号	传感器把数据总线（SDA）拉低 80μs，再接高 80μs 以响应主机的起始信号。
数据格式	收到主机起始信号后，传感器一次性从数据总线（SDA）串出 40 位数据，高位先出
湿度	湿度分辨率是 16Bit，高位在前；传感器串出的湿度值是实际湿度值的 10 倍。
温度	温度分辨率是 16Bit，高位在前；传感器串出的温度值是实际温度值的 10 倍； 温度最高位（Bit15）等于 1 表示负温度，温度最高位（Bit15）等于 0 表示正温度； 温度除了最高位（Bit14~Bit0）表示温度值。
校验位	校验位＝湿度高位+湿度低位+温度高位+温度低位

图 3.3 AM2302 通信格式说明

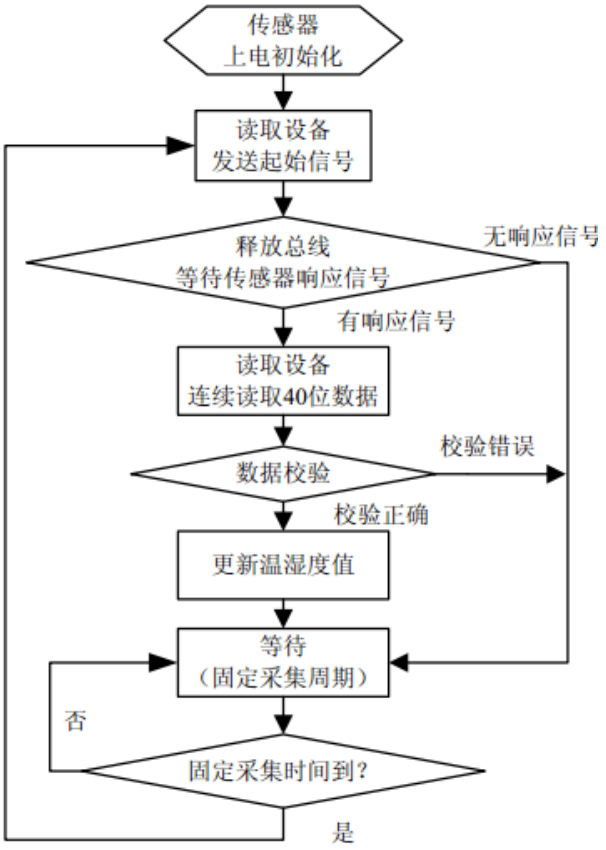


图 3.4 单总线读取流程图

- **温湿度传感器模块原理图**

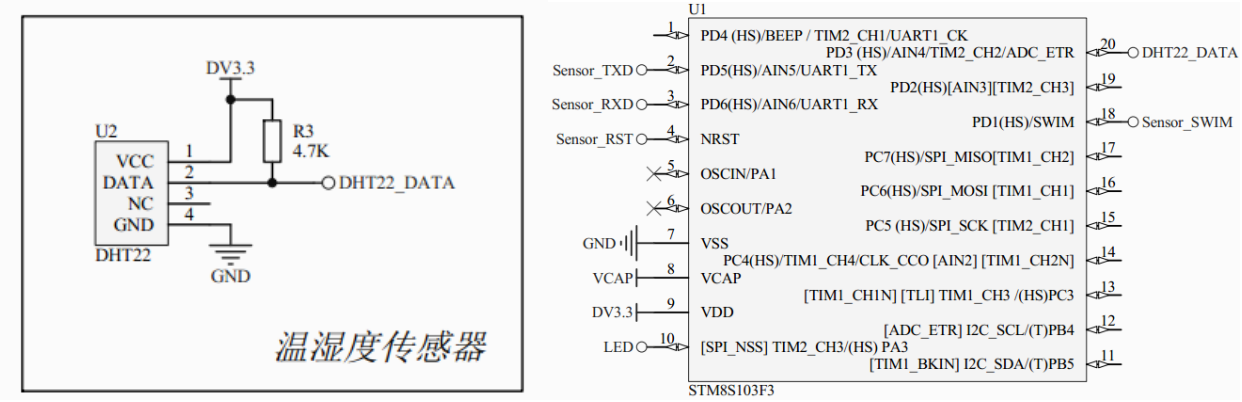


图 3.5 原理图

如原理图所示，STM8 单片机通过 PD3 引脚，软件实现总线线序完成对 DHT22 温湿度传感器的数据采样。

- **源码分析**

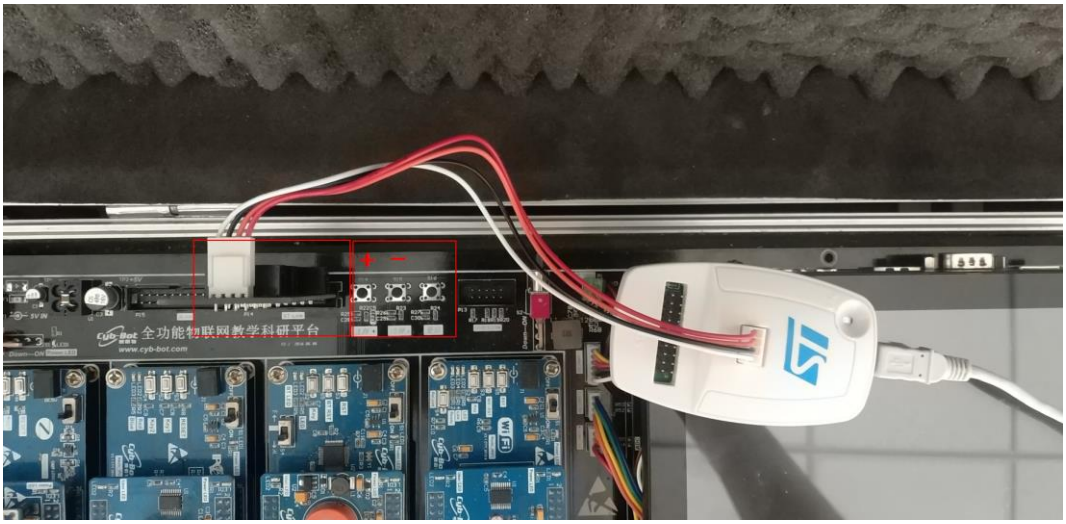
实现代码如下：

```
u8 CMD_rx_buf[8];     // 命令缓冲区
u8 DATA_tx_buf[14]; // 返回数据缓冲区
```

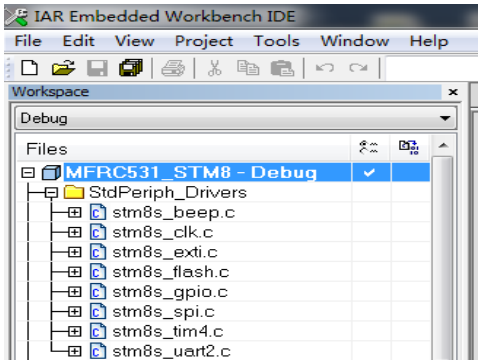
```
u8 CMD_ID = 0;   // 命令序号
u8 Sensor_Type = 0; // 传感器类型编号
u8 Sensor_ID = 0;    // 相同类型传感器编号
u8 Sensor_Data[6];   // 传感器数据区
u8 Sensor_Data_Digital = 0; // 数字类型传感器数据
u16 Sensor_Data_Analog = 0; // 模拟类型传感器数据
u16 Sensor_Data_Threshod = 0;// 模拟传感器阈值
/* 根据不同类型的传感器进行修改 */
    Sensor_Type = 10;
    Sensor_ID = 1;
    CMD_ID = 1;
    DATA_tx_buf[0] = 0xEE;
    DATA_tx_buf[1] = 0xCC;
    DATA_tx_buf[2] = Sensor_Type;
    DATA_tx_buf[3] = Sensor_ID;
    DATA_tx_buf[4] = CMD_ID;
    DATA_tx_buf[13] = 0xFF;
    delay_ms(1000);
    while (1)
    {
        // 获取传感器数据
        if(DHT22_Read())
        {
            Sensor_Data[2] = Humidity >> 8;
            Sensor_Data[3] = Humidity&0xFF;
            Sensor_Data[4] = Temperature >> 8;
            Sensor_Data[5] = Temperature&0xFF;
        }
        // 组合数据帧
        for(i = 0;i < 6;i++)
            DATA_tx_buf[5+i] = Sensor_Data[i];
        // 发送数据帧
        UART1_SendString(DATA_tx_buf, 14);
        LED_Toggle();
        delay_ms(1000);
}
void DHT22_Init(void)
{
    DHT22_DQ_IN();
    DHT22_DQ_PULL_UP();
    delay_s(2);
}
H_H = DHT22_ReadByte();
    H_L = DHT22_ReadByte();
    T_H = DHT22_ReadByte();
    T_L = DHT22_ReadByte();
    Check = DHT22_ReadByte();
        temp = H_H + H_L + T_H + T_L;
    if(Check != temp)
        return 0;
    else
    {
        Humidity      = (unsigned int)(H_H<<8)+(unsigned int)H_L;
        Temperature   = (unsigned int)(T_H<<8)+(unsigned int)T_L;
    }
```

4. 实验步骤

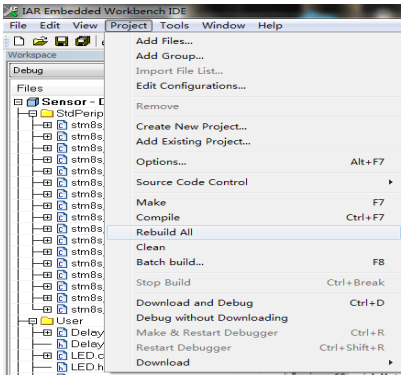
1）首先我们要把传感器插到实验箱的主板上子节点的串口上，再把 ST-Link 配合 JTAG 转接板插到标有 ST-Link 标志的 JTAG 口上，最后把仿真器一端的 USB 线插到 PC 机的 USB 端口，通过主板上的“加”“减”按键选择要编程实验的传感器（会有黄色 LED 灯提示），硬件连接完毕。



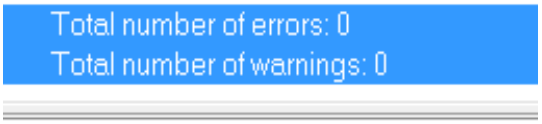
2）我们用 IAR SWSTM8 1.30 软件，打开..\10-Sensor_温湿度测传感器\Project\Sensor.eww。



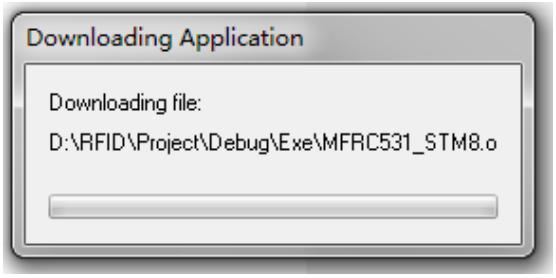
3）打开后点击“Project”下面的“Rebuild All”或者选中工程文件右键“Rebuild All”把我们的工程编译一下。



4）点击“Rebuild All”编译完后，无警告，无错误。



5）编译完后我们要把程序烧到模块里，点击“  ”中间的 Download and Debug 进行烧写。



6）烧写完毕后，把传感器模块从主板上取下来，连接到平台配套的 USB 转串口模块上，将 USB2UART 模块的 USB 线连接到 PC 机的 USB 端口，然后打开串口工具，配置好串口，波特率 115200，8 个数据位，一个停止位，无校验位。

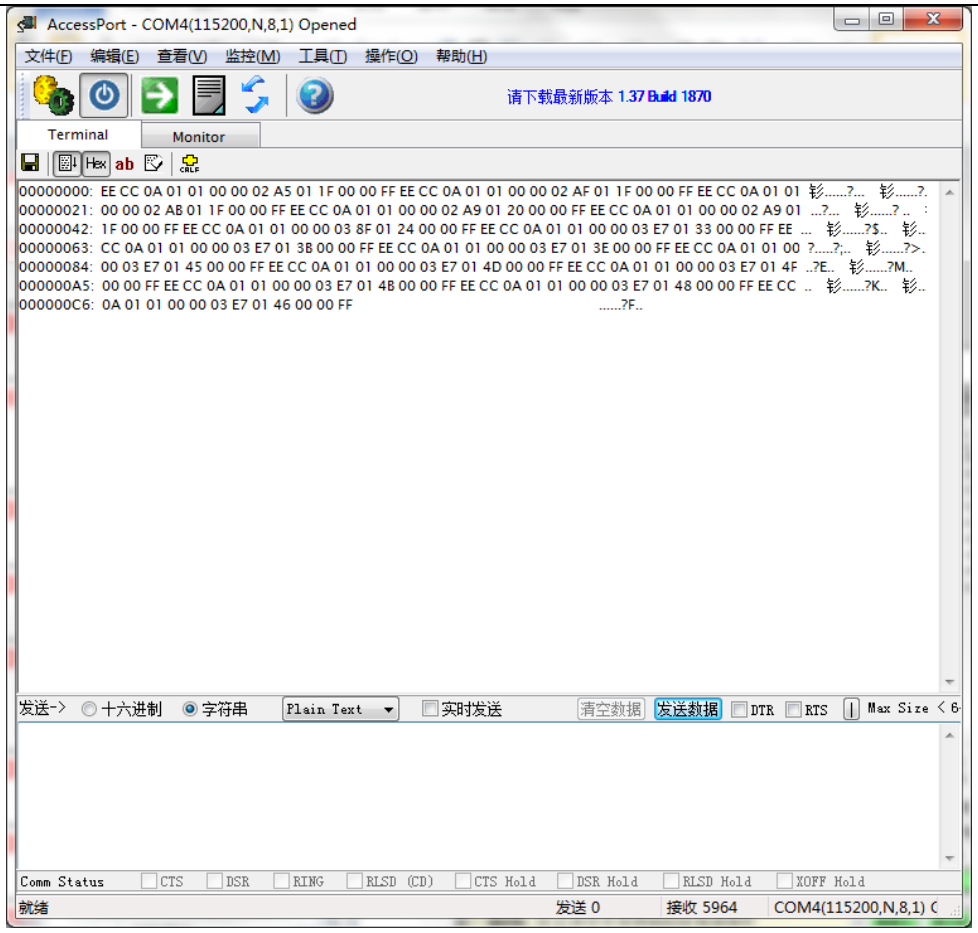
7）传感器底层串口协议返回 14 个字节，第 1 位字节和 2 位字节是包头，第 3 位字节是传感器类型，第 4 位字节是传感器 ID,第 5 位字节是节点命令 ID，第 6 位字节到 11 位字节是数据位，其中第 11 位字节是传感器的状态位，第 12 位字节和第 13 位字节是保留位，第 14 位字节是包尾。

例如：返回 EE CC 0A 01 01 00 00 HH HL TH TL 00 00 FF， HH ，HL 代表温度变化，TH ，TL 代表湿度变化。

```
/* 根据不同类型的传感器进行修改 */
  Sensor_Type = 1;    //传感器类型
  Sensor_ID = 1;      //传感器 ID
  CMD_ID = 1;         //节点命令 ID
  DATA_tx_buf[0] = 0xEE;//包头
  DATA_tx_buf[1] = 0xCC;//包头
  DATA_tx_buf[2] = Sensor_Type;
  DATA_tx_buf[3] = Sensor_ID;
  DATA_tx_buf[4] = CMD_ID;
  DATA_tx_buf[13] = 0xFF;//包尾
```

下图为测试结果

更详细的协议说明，请用户参见《模块通讯协议 V2.6.pdf》文档。


AccessPort - COM4(115200,N,8,1) Opened
文件(F) 编辑(E) 查看(V) 监控(M) 工具(T) 操作(O) 帮助(H)
请下载最新版本 1.37 Build 1870
Terminal
Monitor
00000000: EE CC 0A 01 01 00 00 02 A5 01 1F 00 00 FF EE CC 0A 01 01 00 00 02 AF 01 1F 00 00 FF EE CC 0A 01 01
00000021: 00 00 02 AB 01 1F 00 00 FF EE CC 0A 01 01 00 00 02 A9 01 20 00 00 FF EE CC 0A 01 01 00 00 02 A9 01
00000042: 1F 00 00 FF EE CC 0A 01 01 00 00 03 8F 01 24 00 00 FF EE CC 0A 01 01 00 00 03 E7 01 33 00 00 FF EE
00000063: CC 0A 01 01 00 00 03 E7 01 3B 00 00 FF EE CC 0A 01 01 00 00 03 E7 01 3E 00 00 FF EE CC 0A 01 01 00
00000084: 00 03 E7 01 45 00 00 FF EE CC 0A 01 01 00 00 03 E7 01 4D 00 00 FF EE CC 0A 01 01 00 00 03 E7 01 4F
000000A5: 00 00 FF EE CC 0A 01 01 00 00 03 E7 01 4B 00 00 FF EE CC 0A 01 01 00 00 03 E7 01 48 00 00 FF EE CC
000000C6: 0A 01 01 00 00 03 E7 01 46 00 00 FF
发送-> 十六进制 字符串 Plain Text 实时发送 清空数据 发送数据 DTR RTS Max Size
Comm Status CTS DSR RING RLSD (CD) CTS Hold DSR Hold RLSD Hold XOFF Hold
就绪 发送 0 接收 5964 COM4(115200,N,8,1)

实验十一. 烟雾传感器

1. 实验目的

- 了解烟雾传感器，
- 掌握烟雾传感器工作原理。

2. 实验环境

- 软件：IAR SWSTM8 1.30
- 硬件：ICS-IOT-CEP 教学实验平台，烟雾传感器模块，USB2UART 模块

3. 实验原理

◆ MQ-2 气体传感器简介

MQ-2 气体传感器所使用的气敏材料是在清洁空气中电导率较低的二氧化锡(SnO2)。当传感器所处环境中存在可燃气体时，传感器的电导率随空气中可燃气体浓度的增加而增大。使用简单的电路即可将电导率的变化转换为与该气体浓度相对应的输出信号。

MQ-2 气体传感器对液化气、丙烷、氢气的灵敏度高，对天然气和其它可燃蒸汽的检测也很理想。这种传感器可检测多种可燃性气体，是一款适合多种应用的低成本传感器。

特点

- 在较宽的浓度范围内对可燃气体有良好的灵敏度
- 对液化气、丙烷、氢气的灵敏度较高
- 长寿命、低成本
- 简单的驱动电路即可

该传感器需要施加 2 个电压：加热器电压（VH）和测试电压（VC）。其中 VH 用于为传感器提供特定的工作温度。VC 则是用于测定与传感器串联的负载电阻（RL）上的电压（VRL）。这种传感器具有轻微的极性， VC 需用直流电源。在满足传感器电性能要求的前提下，VC 和 VH 可以共用同一个电源电路。为更好利用传感器的性能，需要选择恰当的 RL 值。

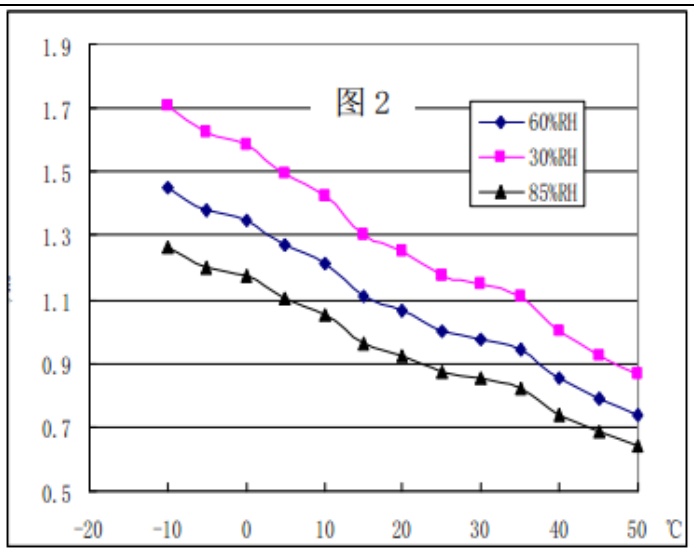


图 3.1 温度 湿度 特性曲线

图中纵坐标是传感器的电阻比（Rs/Ro）。Rs 表示在含 1000ppm 丙烷，不同温/湿度下传感器的电阻值，Ro 表示在含 1000ppm 丙烷、20℃/65%RH 环境条件下传感器的电阻值。

- **烟雾传感器模块原理图**

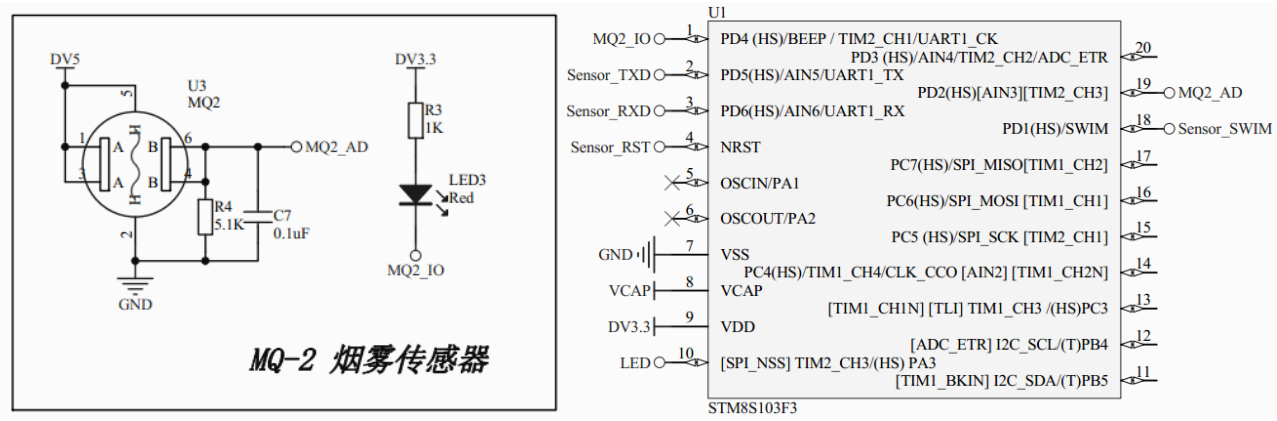


图 3.2 原理图

STM8 的 PD2 即 MQ2_AD 作为模拟量输入，采集 MQ_2 输出端的随烟雾浓度而变化电压值，转换成数字量，当采集的数值大于一定阈值（本程序设置为 50），则判断为有烟雾，并置低 MQ2_IO 引脚，点亮 LED3。

4. 源码分析

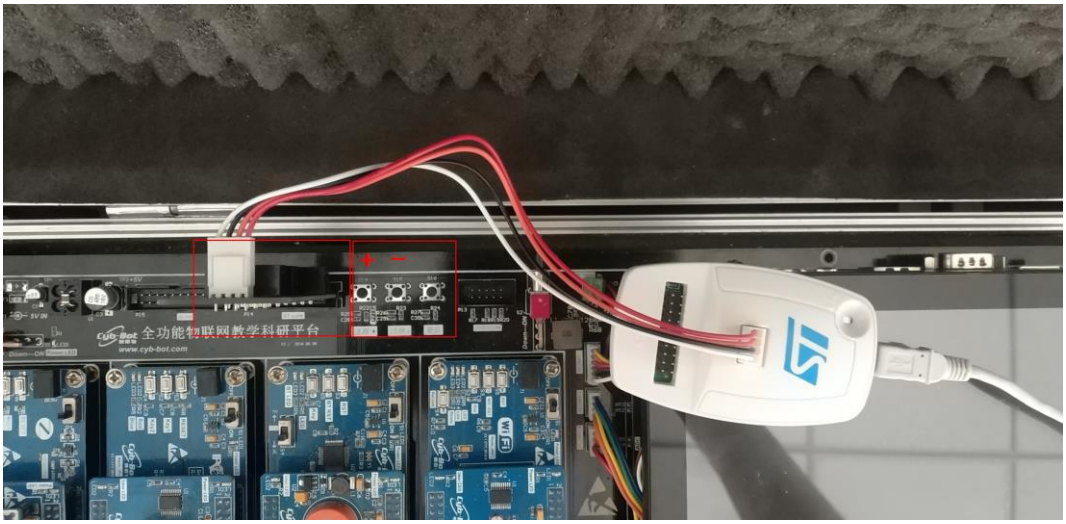
实现代码如下：

```
u8 CMD_rx_buf[8];     // 命令缓冲区
u8 DATA_tx_buf[14]; // 返回数据缓冲区
u8 CMD_ID = 0;    // 命令序号
u8 Sensor_Type = 0; // 传感器类型编号
u8 Sensor_ID = 0;     // 相同类型传感器编号
u8 Sensor_Data[6];    // 传感器数据区
u8 Sensor_Data_Digital = 0; // 数字类型传感器数据
```

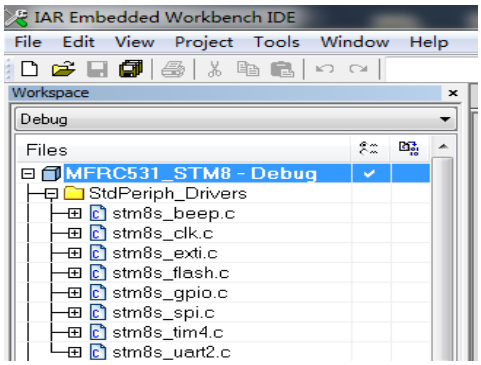
```
u16 Sensor_Data_Analog = 0; // 模拟类型传感器数据
u16 Sensor_Data_Threshod = 0;// 模拟传感器阈值
/* 根据不同类型的传感器进行修改 */
    Sensor_Type = 11;
    Sensor_ID = 1;
    CMD_ID = 1;
    DATA_tx_buf[0] = 0xEE;
    DATA_tx_buf[1] = 0xCC;
    DATA_tx_buf[2] = Sensor_Type;
    DATA_tx_buf[3] = Sensor_ID;
    DATA_tx_buf[4] = CMD_ID;
    DATA_tx_buf[13] = 0xFF;
    GPIO_Init(GPIOD, GPIO_PIN_4, GPIO_MODE_OUT_PP_HIGH_SLOW);
    // ADC
    ADC1_Init(ADC1_CONVERSIONMODE_CONTINUOUS,
              ADC1_CHANNEL_3,
              ADC1_PRESSEL_FCPU_D4,
              ADC1_EXTTRIG_TIM,
              DISABLE,
              ADC1_ALIGN_RIGHT,
              ADC1_SCHMITTTRIG_CHANNEL3,
              DISABLE);
    ADC1_Cmd(ENABLE);
    ADC1_StartConversion();
    Sensor_Data_Analog = 0;
    Sensor_Data_Threshod = 50;
      delay_ms(1000);
      while (1)
    {
       // 获取传感器数据
       Sensor_Data_Analog = ADC1_GetConversionValue();
       if(Sensor_Data_Analog > Sensor_Data_Threshod)
       {
            Sensor_Data_Digital = 0;      // 无烟雾
            GPIO_WriteHigh(GPIOD, GPIO_PIN_4);
       }
       else
       {
            Sensor_Data_Digital = 1;      // 有烟雾
            GPIO_WriteLow(GPIOD, GPIO_PIN_4);
       }
        // 组合数据帧
       DATA_tx_buf[10] = Sensor_Data_Digital;
        // 发送数据帧
       UART1_SendString(DATA_tx_buf, 14);    //串口通讯
       LED_Toggle();
       delay_ms(1000);
    }
```

5. 实验步骤

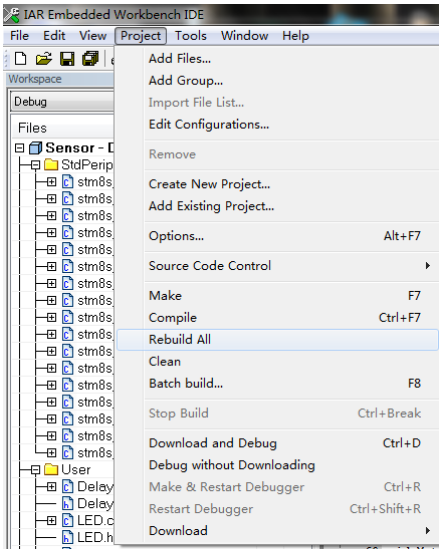
1）首先我们要把传感器插到实验箱的主板上子节点的串口上，再把 ST-Link 配合 JTAG 转接板插到标有 ST-Link 标志的 JTAG 口上，最后把仿真器一端的 USB 线插到 PC 机的 USB 端口，通过主板上的“加”“减”按键选择要编程实验的传感器（会有黄色 LED 灯提示），硬件连接完毕。



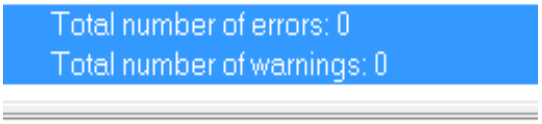
2）我们用 IAR SWSTM8 1.30 软件，打开..\11-Sensor_烟雾传感器\Project\Sensor.eww。

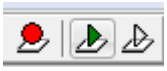


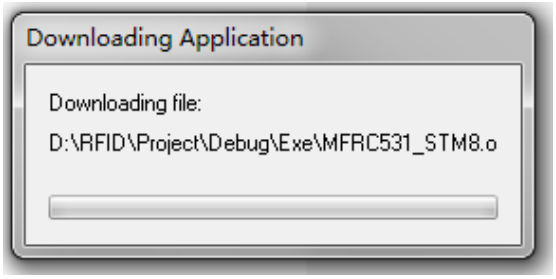
3）打开后点击“Project”下面的“Rebuild All”或者选中工程文件右键“Rebuild All”把我们的工程编译一下。



4）点击“Rebuild All”编译完后，无警告，无错误。



5）编译完后我们要把程序烧到模块里，点击“  ”中间的 Download and Debug 进行烧写。



6）烧写完毕后，把传感器模块从主板上取下来，连接到平台配套的 USB 转串口模块上，将 USB2UART 模块的 USB 线连接到 PC 机的 USB 端口，然后打开串口工具，配置好串口，波特率 115200，8 个数据位，一个停止位，无校验位。

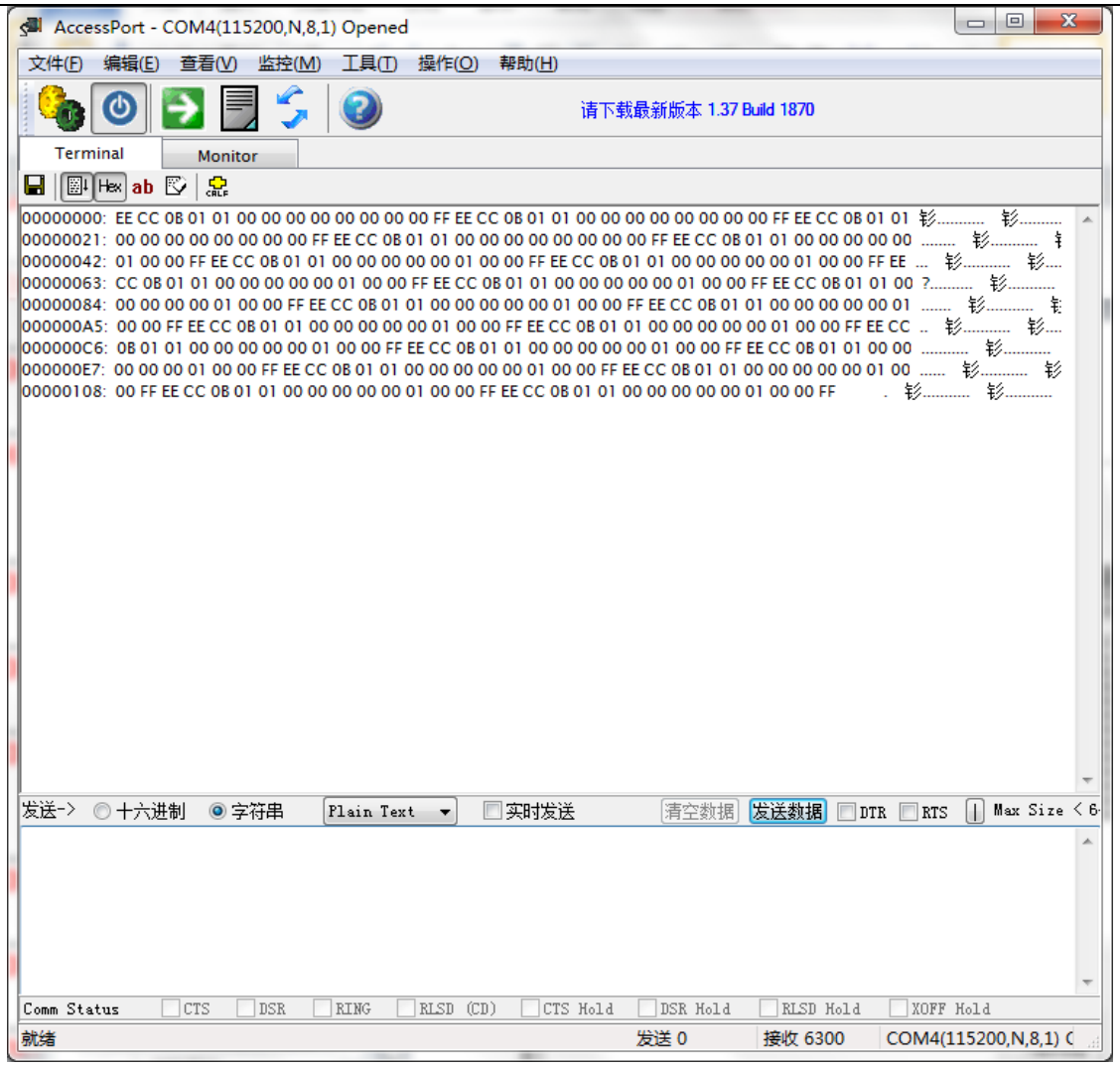
7）传感器底层串口协议返回 14 个字节，第 1 位字节和 2 位字节是包头，第 3 位字节是传感器类型，第 4 位字节是传感器 ID,第 5 位字节是节点命令 ID，第 6 位字节到 11 位字节是数据位，其中第 11 位字节是传感器的状态位，第 12 位字节和第 13 位字节是保留位，第 14 位字节是包尾。

例如：返回“EE CC 0B 01 01 00 00 00 00 00 00 00 FF”时，第 11 位字节为“0”时，表示无烟雾，返回“EE CC 0B 01 01 00 00 00 00 00 01 00 00 FF”时,第 11 位字节为“1”是表示有烟雾。

```
/* 根据不同类型的传感器进行修改 */
    Sensor_Type = 1;     //传感器类型
    Sensor_ID = 1;       //传感器 ID
    CMD_ID = 1;          //节点命令 ID
    DATA_tx_buf[0] = 0xEE;//包头
    DATA_tx_buf[1] = 0xCC;//包头
    DATA  tx  buf[2] = Sensor  Type;
    DATA  tx  buf[3] = Sensor  ID;
    DATA_tx_buf[4] = CMD_ID;
    DATA_tx_buf[13] = 0xFF;//包尾
```

下图为测试结果

更详细的协议说明，请用户参见《模块通讯协议 V2.6.pdf》文档。


AccessPort - COM4(115200,N,8,1) Opened
文件(F) 编辑(E) 查看(V) 监控(M) 工具(T) 操作(O) 帮助(H)
请下载最新版本 1.37 Build 1870
Terminal
Monitor
00000000: EE CC 0B 01 01 00 00 00 00 00 00 00 FF EE CC 0B 01 01 00 00 00 00 00 00 00 00 FF EE CC 0B 01 01
00000021: 00 00 00 00 00 00 00 00 FF EE CC 0B 01 01 00 00 00 00 00 00 00 00 FF EE CC 0B 01 01 00 00 00 00
00000042: 01 00 00 FF EE CC 0B 01 01 00 00 00 00 00 00 01 00 00 FF EE CC 0B 01 01 00 00 00 00 00 00 01 00
00000063: CC 0B 01 01 00 00 00 00 00 00 01 00 00 FF EE CC 0B 01 01 00 00 00 00 00 00 01 00 00 FF EE CC 0B
00000084: 00 00 00 00 00 01 00 00 FF EE CC 0B 01 01 00 00 00 00 00 00 01 00 00 FF EE CC 0B 01 01 00 00 00
000000A5: 00 00 FF EE CC 0B 01 01 00 00 00 00 00 00 01 00 00 FF EE CC 0B 01 01 00 00 00 00 00 00 01 00 00
000000C6: 0B 01 01 00 00 00 00 00 00 01 00 00 FF EE CC 0B 01 01 00 00 00 00 00 00 01 00 00 FF EE CC 0B 01
000000E7: 00 00 00 01 00 00 FF EE CC 0B 01 01 00 00 00 00 00 00 01 00 00 FF EE CC 0B 01 01 00 00 00 00 00
00000108: 00 FF EE CC 0B 01 01 00 00 00 00 00 00 01 00 00 FF EE CC 0B 01 01 00 00 00 00 00 00 01 00 00 FF
发送-> 十六进制 字符串 Plain Text 实时发送 清空数据 发送数据 DTR RTS Max Size
Comm Status CTS DSR RING RLSD (CD) CTS Hold DSR Hold RLSD Hold XOFF Hold
就绪 发送 0 接收 6300 COM4(115200,N,8,1)

实验十二. 振动检测传感器

1. 实验目的

- 了解振动传感器硬件接口原理。
- 掌握振动检测传感器的工作原理。

2. 实验环境

- 软件：IAR SWSTM8 1.30。
- 硬件：ICS-IOT-CEP 教学实验平台，振动检测传感器模块。

3. 实验原理

◆ 振动检测传感器简介

振动传感器，就是在感应振动力大小将感应结果传递到电路装置，并使电路启动工作的电子开关。业内的叫法一般分开为两大类，弹簧开关与滚珠开关。振动开关主要应用于电子玩具、小家电、运动器材以及各类防盗器等产品中。振动开关因为拥有灵活且灵敏的触发性，成为许多电子产品中不可或缺的电子元件。

以专业角度来分析，我们还是分为弹簧开关与滚珠开关两大类来看。两大类开关都有两个比较重要的指标特性，灵敏度和方向性。弹簧开关的灵敏度是指不同的产品，在实际装置中会产生因感应振动力大小不同的差异，此差异称为灵敏度。使用者会因为不同产品的需求，而选择不同感应振动力大小的振动开关来满足自己产品的灵敏度。例如一个玩具拿在手上轻微摇晃和一个球丢到地上或墙上，就会要求不同感应的弹簧开关来感应振动力与电子电路匹配。方向性是指受力方向，而受力方向粗略分为立体的六面，上下左右前后等六面。一般的产品只有灵敏度的要求并没有方向性的要求，因此要先了解使用者的产品的用途，才能建议使用者使用那种型号的弹簧开关。而滚珠开关与弹簧开关最大的区别在于：弹簧开关是感应振动力或离心力的大小，最好为直立使用。而滚珠开关是感应角度的变化，最好平铺使用。滚珠开关的灵敏度，就是感应角度大小，将感应结果传递到电路装置使电路启动。在实际装置中就会产生因不同的产品感应角度大小不同的差异,此差异称为灵敏度。使用者会因为不同产品的需求，而要求不同感应角度大小的滚珠开关来满足产品的灵敏度。例如用手拿起一个杯子在轻微角度倾斜时，电路装置就必须使 IC 启动 LED 闪亮或发出声音。客户就会要求不同感应的滚珠开关来感应角度，与电子电路匹配。滚珠开关的方向性是指倾斜角度的方向，其方向粗略为左右二面。

◆ 振动检测传感器模块原理图

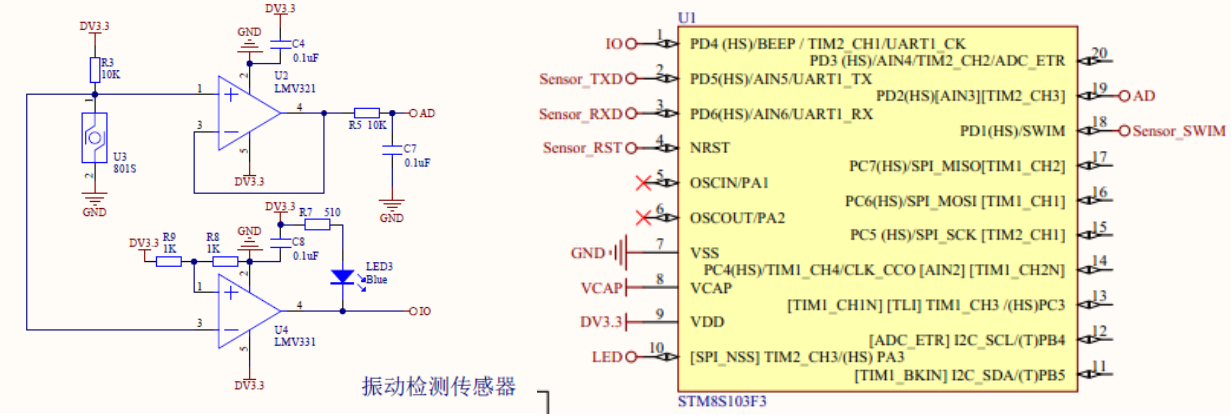


图 1.1 振动传感器模块原理图

振动模块不震动时，传感器相当于通路，运放输入 0V，AD 采集值为 0。振动模块振动时，相当于以一定的频率（取决于振动的激烈程度和方式）进行通断路，AD 的值从 0-0x3FF 变化。

可以通过取几十次的 AD 采样值的平均值判断，也可以通过检测大于某个阈值的 AD 采样点个数。设置振动模块定时器 1ms 溢出中断 1 次，设置了 1s 的计数标志，主函数检测 1s 内大于 0x300 的 AD 采样次数，大于 100 次则判断有振动，点亮 LED。

可以通过修改阈值 0x300 和 100 次检测不同类型的振动。

- **源码分析**

串口函数为：

```
void UART1  SendString(u8* Data,u16 len)
{
      u16 i=0;
      for(;i<len;i++)
            UART1  SendByte(Data[i]);
  }
void UART1_SendByte(u8 data)
{
      UART1_SendData8((unsigned char)data);
      /* Loop until the end of transmission */
      while (UART1_GetFlagStatus(UART1_FLAG_TXE) == RESET);
}
```

软件主要有配置振动传感器，UART，LED 灯，实现代码如下：

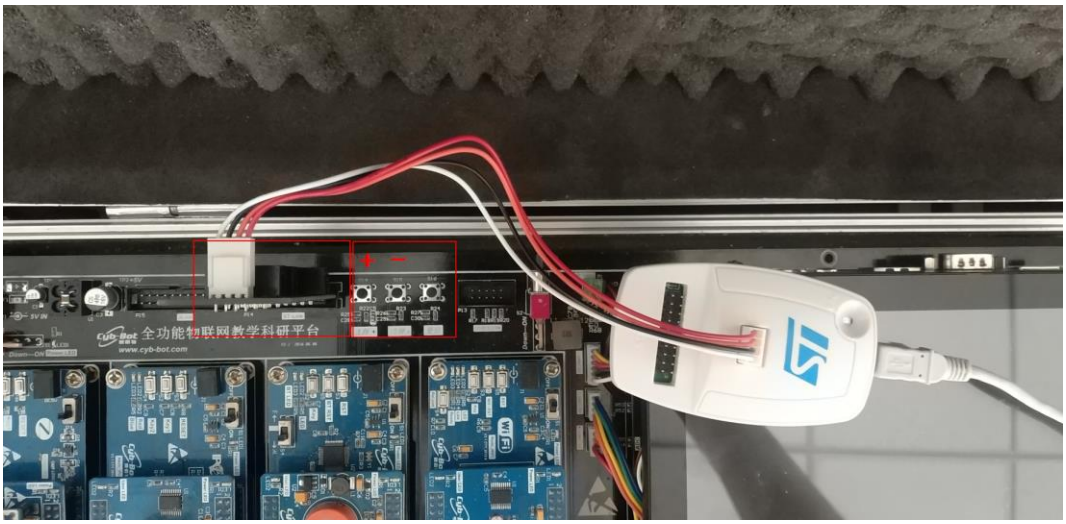
```
#include "main.h"

u8 CMD_rx_buf[8];     // 命令缓冲区
u8 DATA_tx_buf[14]; // 返回数据缓冲区
u8 CMD_ID = 0;    // 命令序号
u8 Sensor_Type = 0; //传感器类型编号
u8 Sensor_ID = 0;     // 相同类型传感器编号
u8 Sensor_Data[6];    // 传感器数据区

u8 Sensor_Data_Digital = 0; // 数字类型传感器数据
u16 Sensor_Data_Analog = 0; // 模拟类型传感器数据
u16 Sensor_Data_Threshod = 0;// 模拟传感器阈值
```

```
u8 n = 0;

void main(void)
{
    u8 i = 0;

    CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV1);          //设置内部时钟 16M 为主时钟

    Uart1_Init();
    TIM4_Init();
    LED_Init();
    GPIO_Init(GPIOD, GPIO_PIN_4, GPIO_MODE_OUT_PP_HIGH_SLOW);

    // ADC
    ADC1_Init(ADC1_CONVERSIONMODE_CONTINUOUS,
              ADC1_CHANNEL_3,
              ADC1_PRESSEL_FCPU_D4,
              ADC1_EXTTRIG_TIM,
              DISABLE,
              ADC1_ALIGN_RIGHT,
              ADC1_SCHMITTTRIG_CHANNEL3,
              DISABLE);
    ADC1_Cmd(ENABLE);
    ADC1_StartConversion();
    Sensor_Data_Analog = 0;

    for(i = 0; i < 14; i++)
    {
      DATA_tx_buf[i] = 0;
    }

    for(i = 0; i < 8; i++)
    {
      CMD_rx_buf[i] = 0;
    }

    /* 根据不同类型的传感器进行修改*/
    Sensor_Type = 19;
    Sensor_ID = 1;

    CMD_ID = 1;

    DATA_tx_buf[0] = 0xEE;
    DATA_tx_buf[1] = 0xCC;
    DATA_tx_buf[2] = Sensor_Type;
    DATA_tx_buf[3] = Sensor_ID;
    DATA_tx_buf[4] = CMD_ID;
    DATA_tx_buf[13] = 0xFF;

    zhendong_flag=0;
    while (1)
    {
        // 获取传感器数据
      Sensor_Data_Analog = ADC1_GetConversionValue();
      if(Sensor_Data_Analog > 768)
      {
        n++;
      }
```

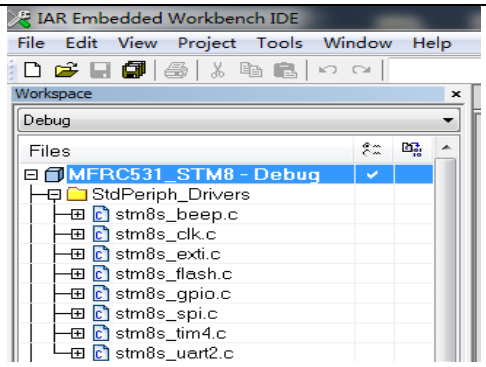
```
            if (zhendong_flag == 1)
              {
               zhendong_flag=0;
               if (n>100) //100 次为基准
                 {
                   DATA_tx_buf[10] = 0x01; //   有振动
                   UART1_SendString(DATA_tx_buf,14); //发送数据
                   LED_On();
                   delay_ms(2000);
                   n=0;
                   LED_Off();
                 }
              else
                 {
                   DATA_tx_buf[10] = 0x0; //   无振动
                   UART1_SendString(DATA_tx_buf,14); //发送数据
                 }
              }
        }
}
```

4. 实验步骤

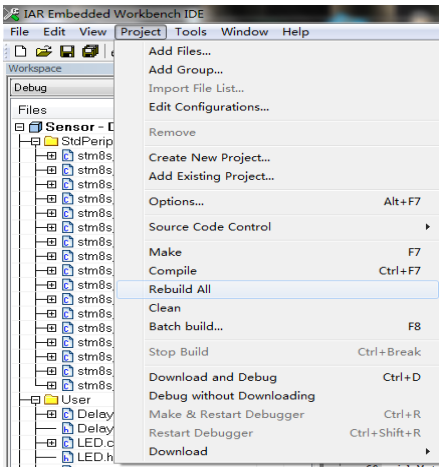
1）首先我们要把传感器插到实验箱的主板上子节点的串口上，再把 ST-Link 配合 JTAG 转接板插到标有 ST-Link 标志的 JTAG 口上，最后把仿真器一端的 USB 线插到 PC 机的 USB 端口，通过主板上的“加”“减”按键选择要编程实验的传感器（会有黄色 LED 灯提示），硬件连接完毕。



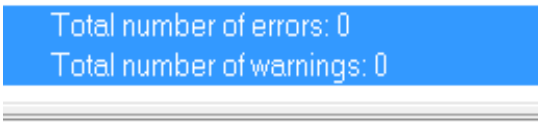
2）我们用 IAR SWSTM8 1.30 软件，打开 12-Sensor_振动检测传感器\Project\Sensor.eww。



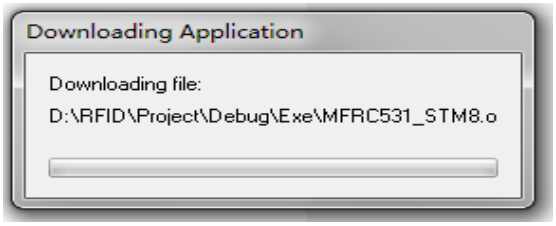
3）打开后点击“Project”下面的“Rebuild All”或者选中工程文件右键“Rebuild All”把我们的工程编译一下。



4）点击“Rebuild All”编译完后，无警告（有警告一般情况下可以忽略），无错误。



5）编译完后我们要把程序烧到模块里，点击“  ”中间的 Download and Debug 进行烧写。



6）烧写完毕后，把传感器模块从主板上取下来，连接到平台配套的 USB 转串口模块上，将 USB2UART 模块的 USB 线连接到 PC 机的 USB 端口，然后打开串口工具，配置好串口，波特率 115200，8 个数据位，一个停止位，无校验位。

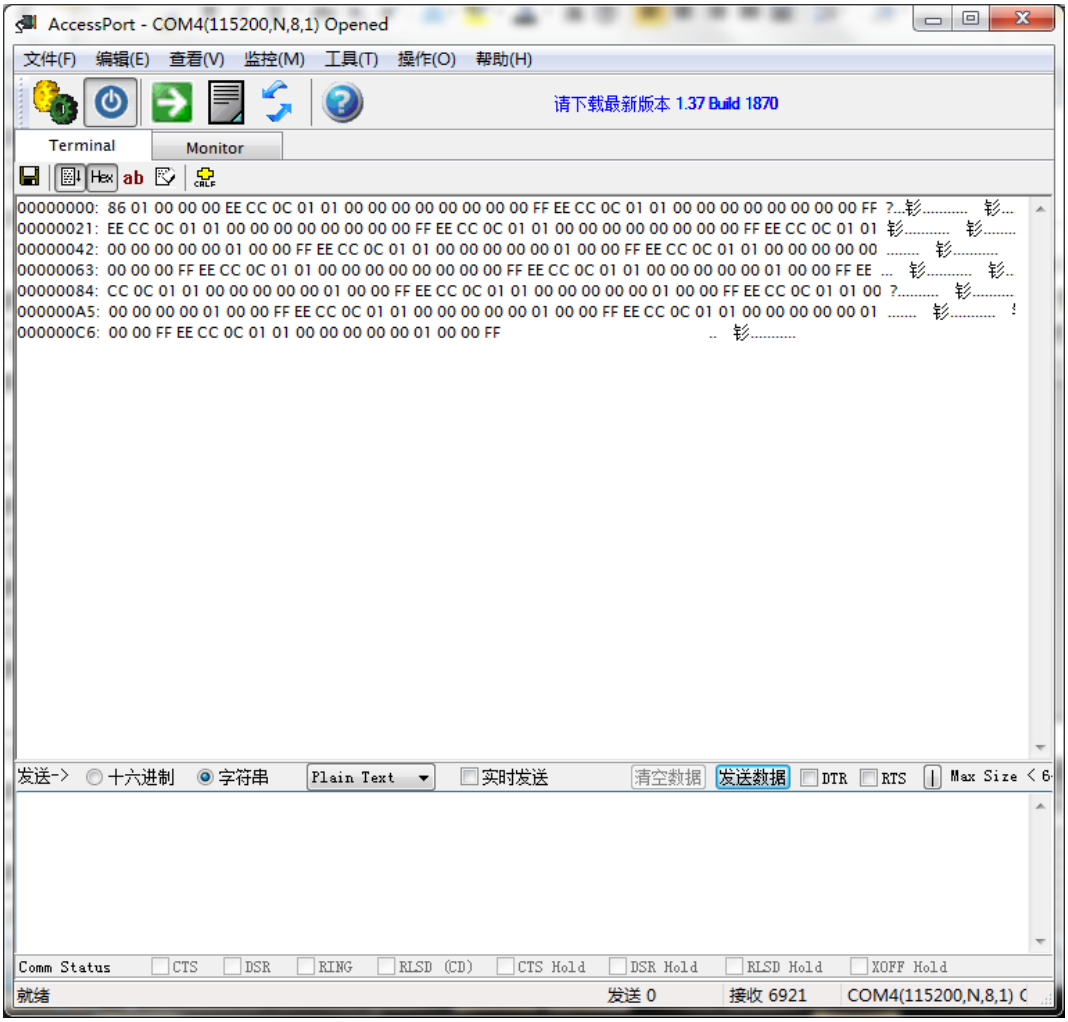
7）传感器底层串口协议返回 14 个字节，第 1 位字节和 2 位字节是包头，第 3 位字节是传感器类型，第 4 位字节是传感器 ID,第 5 位字节是节点命令 ID，第 6 位字节到 11 位字节是数据位，其中第 11 位字节是传感器的状态位，第 12 位字节和第 13 位字节是保留位，第 14 位字节是包尾。

例如：返回“EE CC 12 01 01 00 00 00 00 00 00 00 FF”时，第 11 位字节为“0”时，表示无振动，返回“EE CC 12 01 01 00 00 00 00 00 01 00 00 FF”时,第 11 位字节为“1”是表示有振动。

```
/* 根据不同类型的传感器进行修改 */
    Sensor_Type = 1;    //传感器类型
    Sensor_ID = 1;      //传感器 ID
    CMD_ID = 1;         //节点命令 ID
    DATA_tx_buf[0] = 0xEE;//包头
    DATA_tx_buf[1] = 0xCC;//包头
    DATA_tx_buf[2] = Sensor_Type;
    DATA_tx_buf[3] = Sensor_ID;
    DATA_tx_buf[4] = CMD_ID;
    DATA_tx_buf[13] = 0xFF;//包尾
```

下图为测试结果

更详细的协议说明，请用户参见《模块通讯协议 V2.6.pdf》文档。



实验十三. 步进电机驱动

1. 实验目的

- 了解步进电机驱动。
- 掌握步进电机驱动工作原理。

2. 实验环境

- 软件：IAR SWSTM8 1.30。
- 硬件：ICS-IOT-CEP 教学实验平台，步进电机驱动模块，USB2UART 模块

3. 实验原理

◆ 硬件原理

步进电机是一种将电脉冲转化为角位移的执行机构。通俗一点讲：当步进驱动器接收到一个脉冲信号，它就驱动步进电机按设定的方向转动一个固定的角度（及步进角）。可以通过控制脉冲个来控制角位移量，从而达到准确定位的目的；同时也可以通过控制脉冲频率来控制电机转动的速度和加速度，从而达到调速的目的。

步进电机驱动模块采用的是 StepMotor 型号的驱动模块，处理器为 STM8S 芯片，供电电压为 5V，接口为 UART（TTL 电平），驱动方式为四相八拍，转矩（工作频率 100Hz）为≥300gf.cm。。当对步进电机施加一系列连续不断的控制脉冲时，它可以连续不断地转动。每一个脉冲信号对应步进电机的某一相或两相绕组的通电状态改变一次，也就对应转子转过一定的角度（一个步距角）。当通电状态的改变完成一个循环时，转子转过一个齿距。四相步进电机可以在不同的通电方式下运行，常见的通电方式有单（单相绕组通电）四拍（A-B-C-D-A…），双（双相绕组通电）四拍（AB-BC-CD-DA-AB-…），八拍（A-AB-B-BC-C-CD-D-DA-A…）

驱动方式：(4-1-2 相驱动)

导线颜色	1	2	3	4	5	6	7	8
5 红	+	+	+	+	+	+	+	+
4 橙	–	–						–
3 黄		–	–	–				
2 粉				–	–	–		
1 蓝						–	–	–

图 13.1 逆向相序表（顺向是相反的）

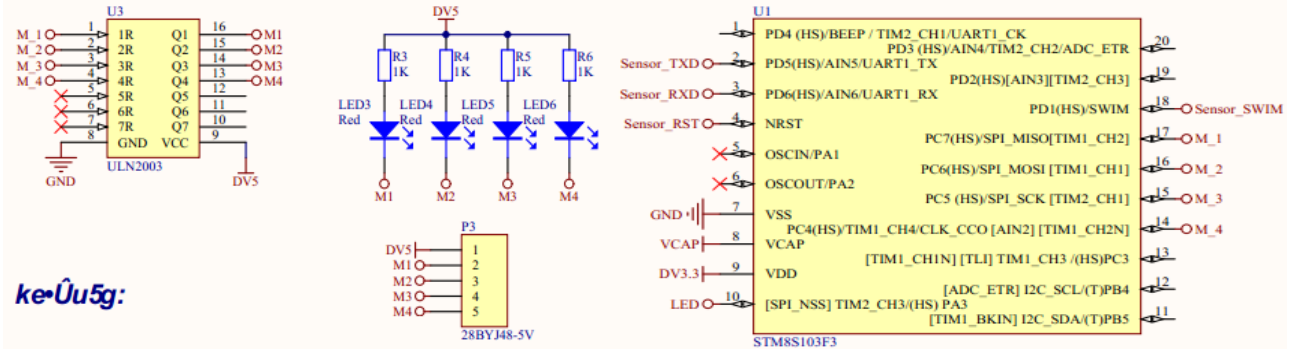


图 13.1 原理图

如上图所示，步进电机的四个管脚与 STM8S 的 PC4,PC5,PC6,PC7 四个管脚相连，每个管脚都接了 LED 灯，识别管脚状态。

- **详细代码**

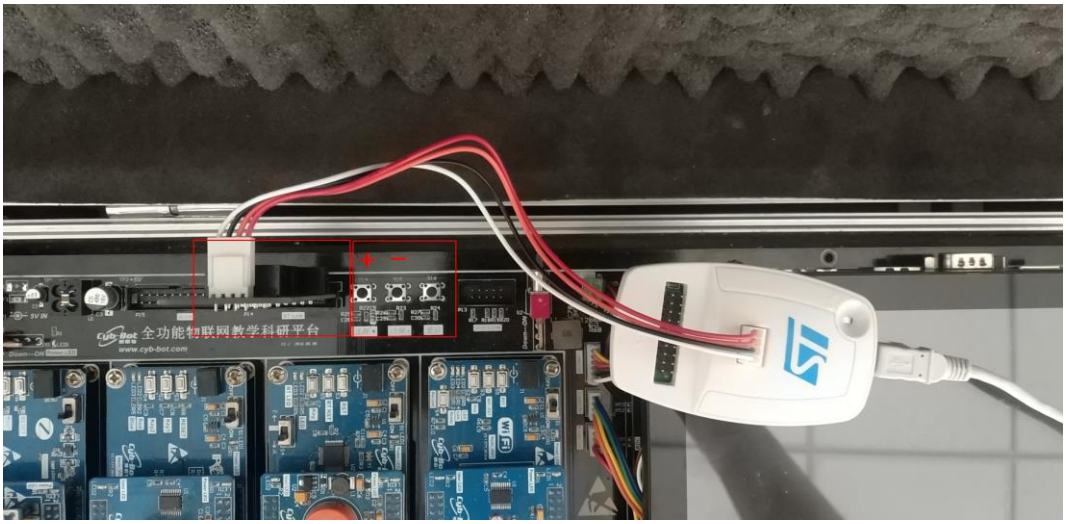
如下：

```
// 换相表
const unsigned char CCW_Tab[8] = {0x80,0xc0,0x40,0x60,0x20,0x30,0x10,0x90}; // 逆时针相序表
const unsigned char CW_Tab[8] = {0x90,0x10,0x30,0x20,0x60,0x40,0xc0,0x80}; // 顺时针相序表
void main(void)
{
    u8 i = 0;
    u16 num = 0;
    u8 mode = 0;
    CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV1);          // 设置内部时钟 16M 为主时钟

    Uart1_Init();

    LED_Init();

    Motor  Init();

    for(i = 0;i < 14;i++)
        DATA_tx_buf[i] = 0;
    for(i = 0;i < 8;i++)
        CMD_rx_buf[i] = 0;

    /* 根据不同类型的传感器进行修改 */
    Sensor  Type = 0x10;
    Sensor  ID = 1;

    CMD_ID = 1;

    DATA  tx  buf[0] = 0xEE;
    DATA_tx_buf[1] = 0xCC;
    DATA_tx_buf[2] = Sensor_Type;
    DATA_tx_buf[3] = Sensor_ID;
    DATA_tx_buf[4] = CMD_ID;
    DATA  tx  buf[13] = 0xFF;

    delay_ms(1000);

    enableInterrupts();
```

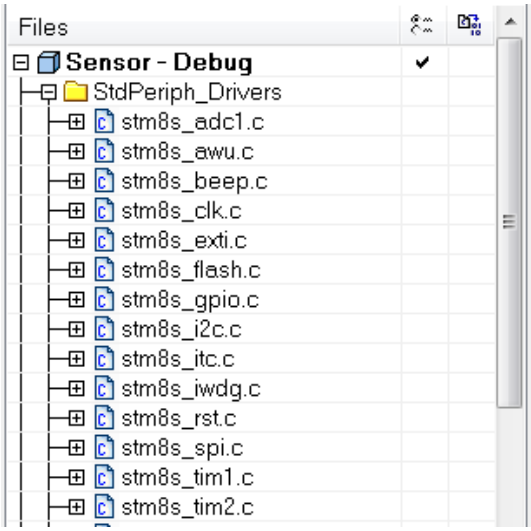
```
    while (1)
    {

u8 buf;

num ++;
if(num == 10){
      DATA_tx_buf[10] = mode;
      UART1_SendString(DATA_tx_buf, 14);
            num = 0;
}

if(Uart_RecvFlag == 1){

      switch(rx_buf[10]){
          case 0x00:// 逆时针旋转一周
              mode = 0;
              Uart_RecvFlag = 0;
              LED  Toggle();
              Motor  Step  CCW(2*64,1);
              break;
          case 0x01:// 顺时针转转一周
              mode = 1;
              Uart  RecvFlag = 0;
              LED_Toggle();
              Motor_Step_CW(2*64,1);
              break;
          default:
              Uart  RecvFlag = 0;
              break;
      }
}

delay  ms(100);

    }
}
```

4. 实验步骤

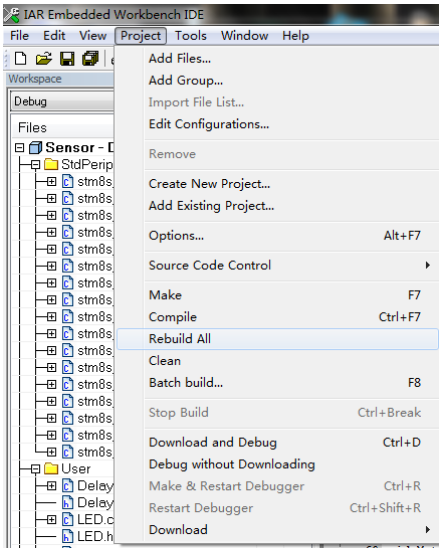
1）首先我们要把传感器插到实验箱的主板上子节点的串口上，再把 ST-Link 配合 JTAG 转接板插到标有 ST-Link 标志的 JTAG 口上，最后把仿真器一端的 USB 线插到 PC 机的 USB 端口，通过主板上的“加”“减”按键选择要编程实验的传感器（会有黄色 LED 灯提示），硬件连接完毕。



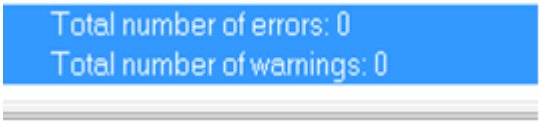
2）我们用 IAR SWSTM8 1.30 软件，打开..\ 13-Sensor_步进电机\Project\Sensor.eww。



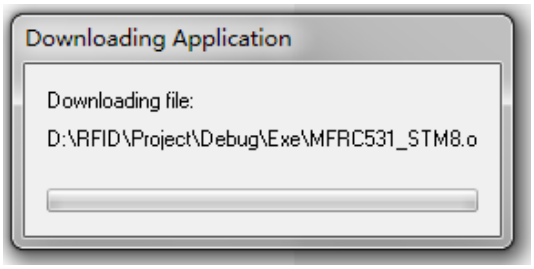
3）打开后点击“Project”下面的“Rebuild All”或者选中工程文件右键“Rebuild All”把我们的工程编译一下。



4）点击“Rebuild All”编译完后，无警告，无错误。

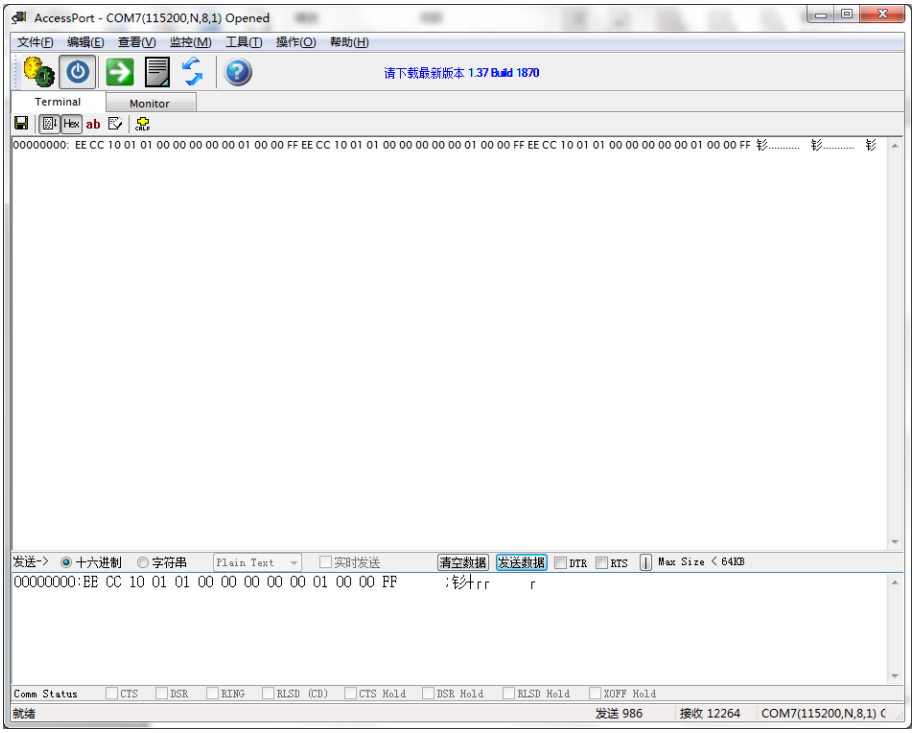


5）编译完后我们要把程序烧到模块里，点击“  ”中间的 Download and Debug 进行烧写。

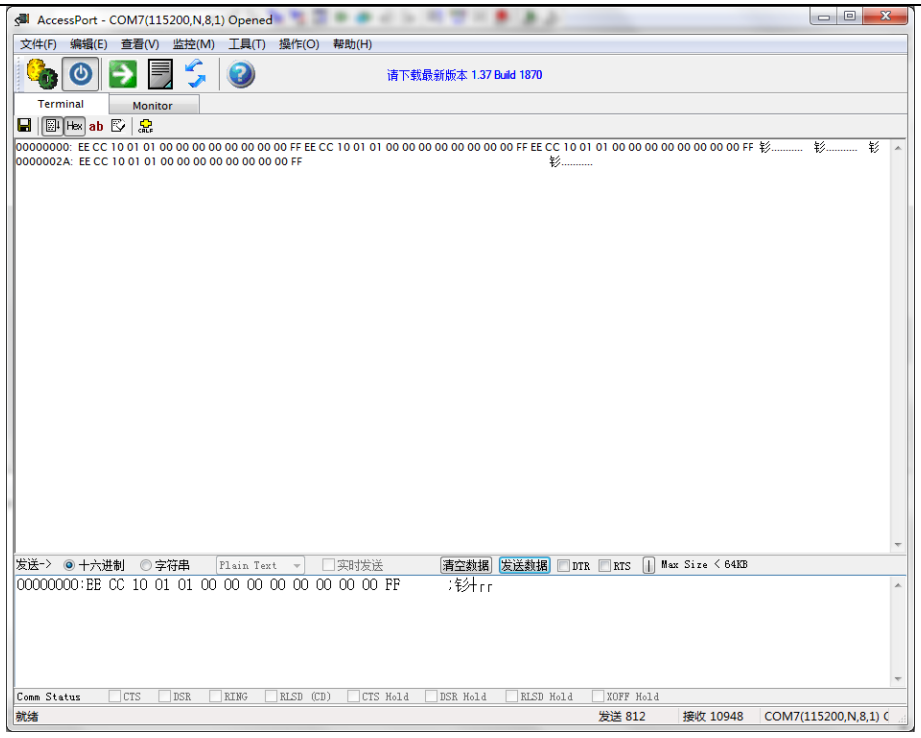


6）烧写完毕后，把传感器模块从主板上取下来，连接到平台配套的 USB 转串口模块上，将 USB2UART 模块的 USB 线连接到 PC 机的 USB 端口，然后打开串口工具，配置好串口，波特率 115200，8 个数据位，一个停止位，无校验位。

7）发送“EE CC 10 01 01 00 00 00 00 00 01 00 00 FF”使电机顺时针旋转一周。



8）发送”EE CC 10 01 01 00 00 00 00 00 00 00 00 FF”使电机逆时针旋转一周。



每当发送命令协议时 LED3-LED6 会有变化，不发送时 LED3 亮，发送时 LED3-LED6 亮。

实验十四. 声光报警传感器

1. 实验目的

- 了解声光报警传感器。
- 掌握声光报警传感器工作原理。

2. 实验环境

- 软件：IAR SWSTM8 1.30。
- 硬件：ICS-IOT-CEP 教学实验平台，声光报警传感器模块，USB2UART 模块

3. 实验原理

◆ 硬件原理

声光报警是一种通过声音与光同时进行的报警的装置，该声光报警模块使用的是一个蜂鸣器与一个 LED 红灯，芯片为 STM8S，芯片的 PD3,PD4 两个管脚与蜂鸣器 BEEP 和 HL_LED 相连，通过这两个管脚来进行控制，当 BEEP 引脚高电平时，蜂鸣器响，当 HL_LED 引脚高电平时，LED 灯点亮。

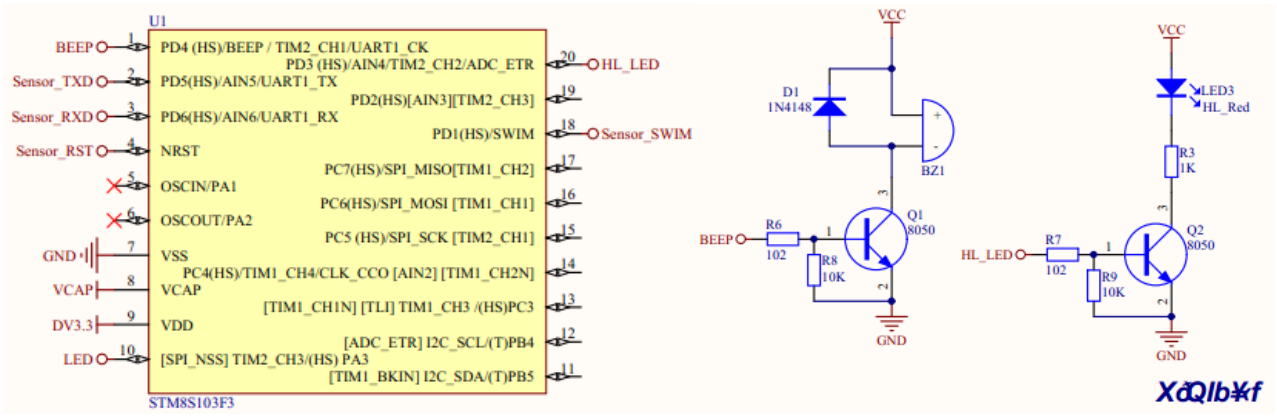


图 14.1 原理图

◆ 详细代码

代码如下：

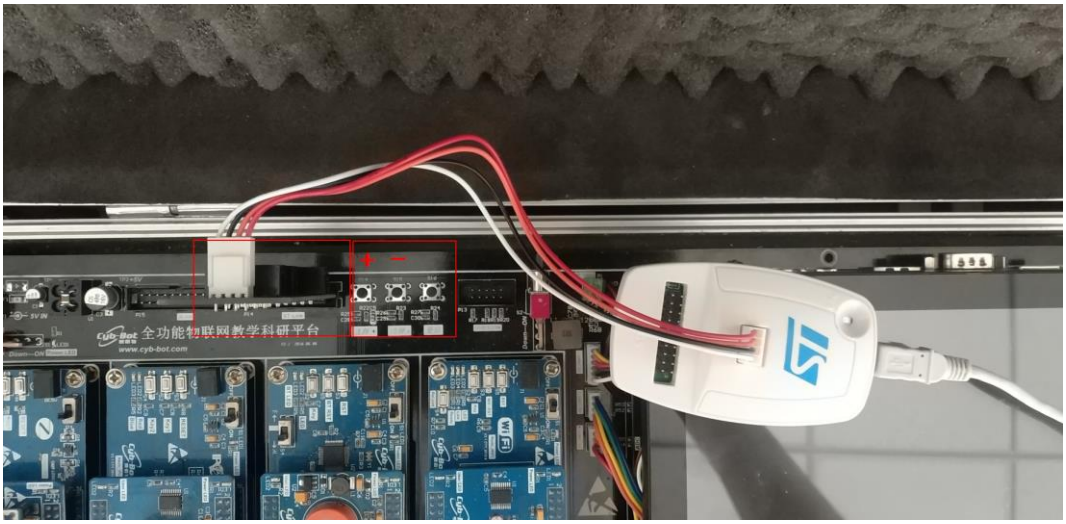
```
void main(void)
{
    u8 i = 0;
    u16 num = 0;
    u8 mode = 0;
    CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV1);        // 设置内部时钟 16M 为主时钟
```

```
    Uart1_Init();

    LED  Init();

    BEEP_Config();

    for(i = 0;i < 14;i++)
        DATA  tx  buf[i] = 0;
    for(i = 0;i < 8;i++)
        CMD_rx_buf[i] = 0;

    /* 根据不同类型的传感器进行修改 */
    Sensor_Type = 0x0E;
    Sensor_ID = 1;

    CMD  ID = 1;

    DATA_tx_buf[0] = 0xEE;
    DATA_tx_buf[1] = 0xCC;
    DATA  tx  buf[2] = Sensor  Type;
    DATA  tx  buf[3] = Sensor  ID;
    DATA_tx_buf[4] = CMD_ID;
    DATA_tx_buf[13] = 0xFF;

    delay  ms(1000);

    enableInterrupts();

    while (1)
    {
u8 buf;

num ++;
if(num == 10){
        DATA  tx  buf[10] = mode;
        UART1_SendString(DATA_tx_buf, 14);
                num = 0;
}

if(mode == 1){
        SENSOR_ON();
        SENSOR_OFF();
}

if(Uart  RecvFlag == 1){

        switch(rx_buf[10]){
            case 0x00:// close
                mode = 0;
                Uart  RecvFlag = 0;
                break;
            case 0x01:/触发传感器
                mode = 1;
                Uart_RecvFlag = 0;
                break;
            default:
                Uart_RecvFlag = 0;
                break;
        }
}
```

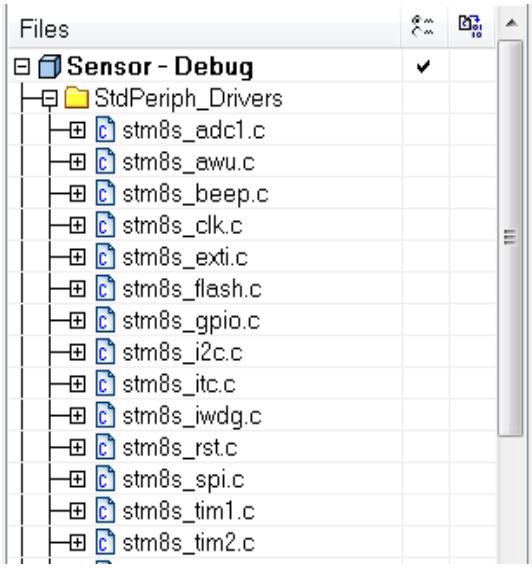
```
    delay_ms(100);

        }
}
```

4. 实验步骤

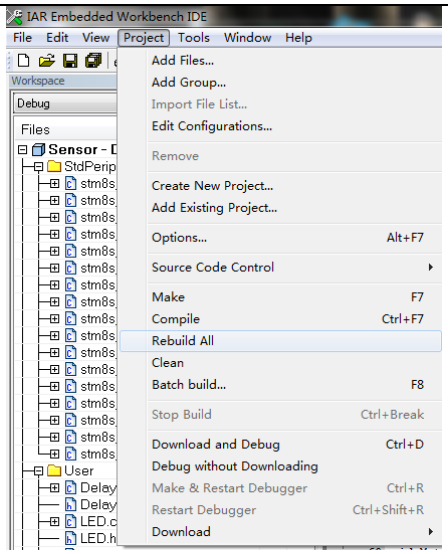
1）首先我们要把传感器插到实验箱的主板上子节点的串口上，再把 ST-Link 配合 JTAG 转接板插到标有 ST-Link 标志的 JTAG 口上，最后把仿真器一端的 USB 线插到 PC 机的 USB 端口，通过主板上的“加”“减”按键选择要编程实验的传感器（会有黄色 LED 灯提示），硬件连接完毕。



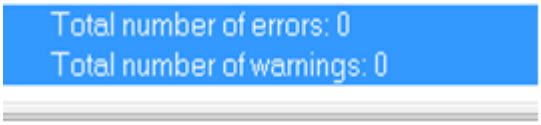
2）我们用 IAR SWSTM8 1.30 软件，打开..\ 14-Sensor_声光报警\Project\Sensor.eww。



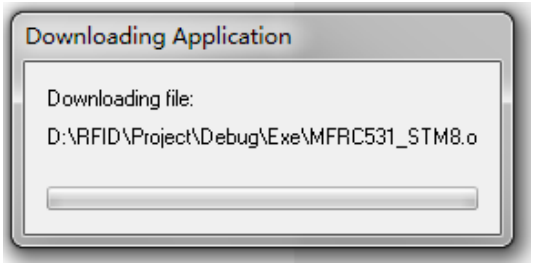
3）打开后点击“Project”下面的“Rebuild All”或者选中工程文件右键“Rebuild All”把我们的工程编译一下。



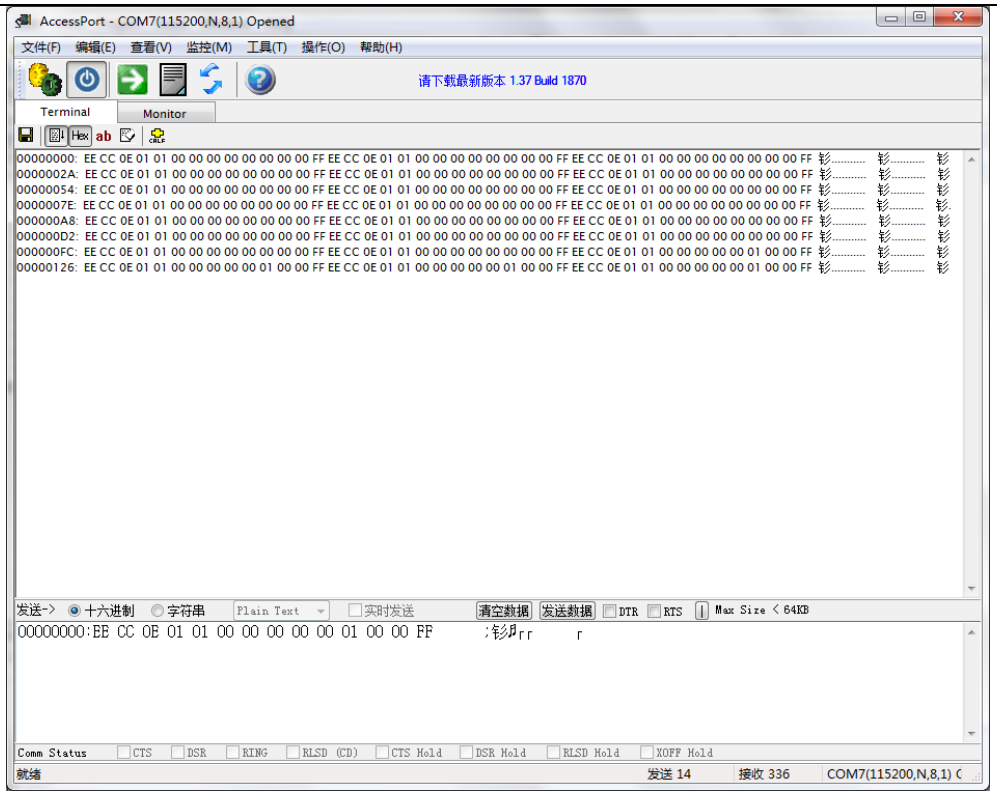
4）点击“Rebuild All”编译完后，无警告，无错误。



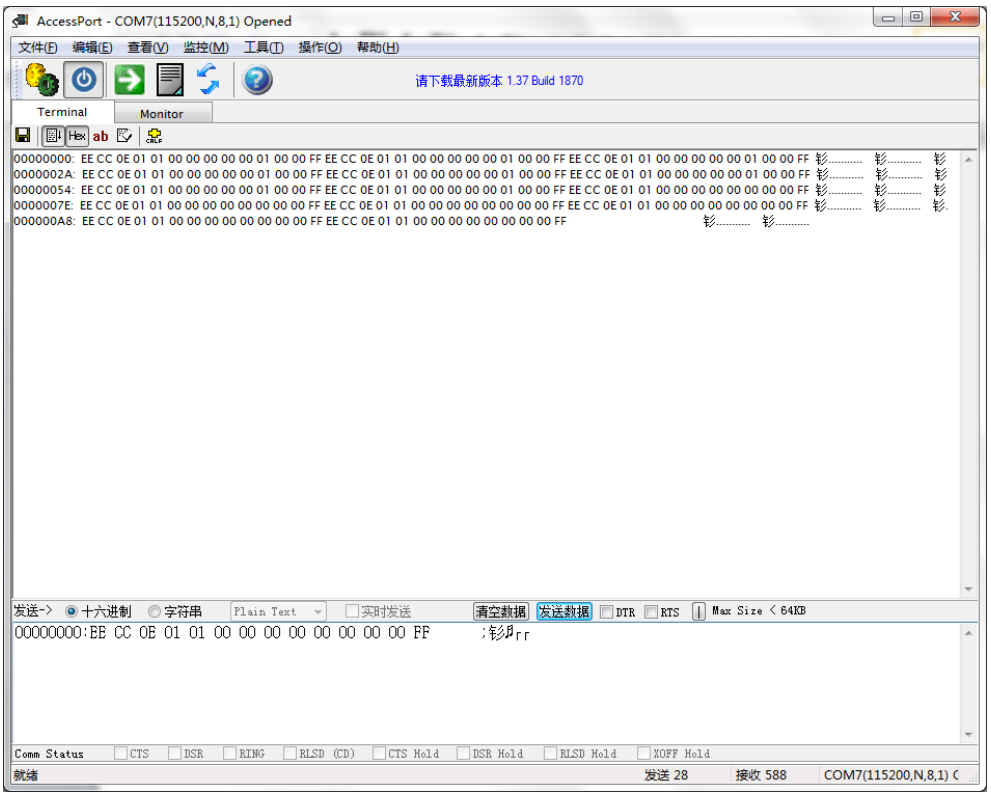
5）编译完后我们要把程序烧到模块里，点击“  ”中间的 Download and Debug 进行烧写。



6）烧写完毕后，把传感器模块从主板上取下来，连接到平台配套的 USB 转串口模块上，将 USB2UART 模块的 USB 线连接到 PC 机的 USB 端口，然后打开串口工具，配置好串口，波特率 115200，8 个数据位，一个停止位，无校验位。发送“EE CC 0E 01 01 00 00 00 00 00 01 00 00 FF”触发声光报警传感器，传感器会不停地报警，蜂鸣器与红灯一起触发。



7）发送“EE CC 0E 01 01 00 00 00 00 00 00 00 FF”声光报警会停止报警。



实验十五. 继电器控制实验

1. 实验目的

- 了解继电器的作用。
- 掌握继电器的工作原理。

2. 实验环境

- 软件：IAR SWSTM8 1.30。
- 硬件：ICS-IOT-CEP 教学实验平台，继电器模块，USB2UART 模块

3. 实验原理

◆ 硬件原理

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

继电器模块使用的是 RelaySwitch 型号的模块，处理器芯片为 STM8S，接口为 UART(TTL 电平)，动作时间≤8ms，释放时间≤5ms，最大切换电压 250VAC/30VDC，最大切换电流 10A，最大切换功率 1250VA/150W。

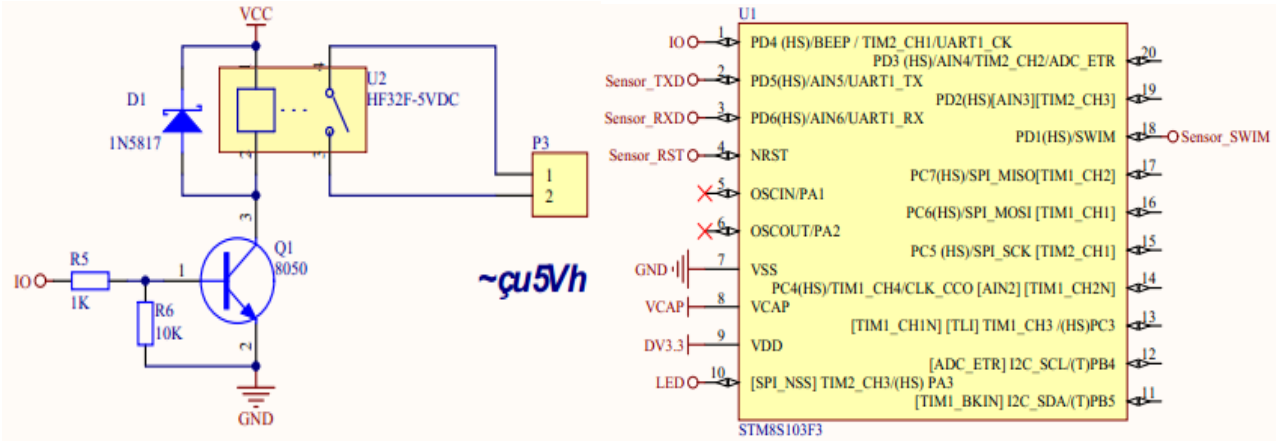


图 15.1 原理图

通过（15.1）原理图得知 STM8S 的 PD4 管脚连接 IO 来控制继电器的闭合与导通。

♦ 详细代码

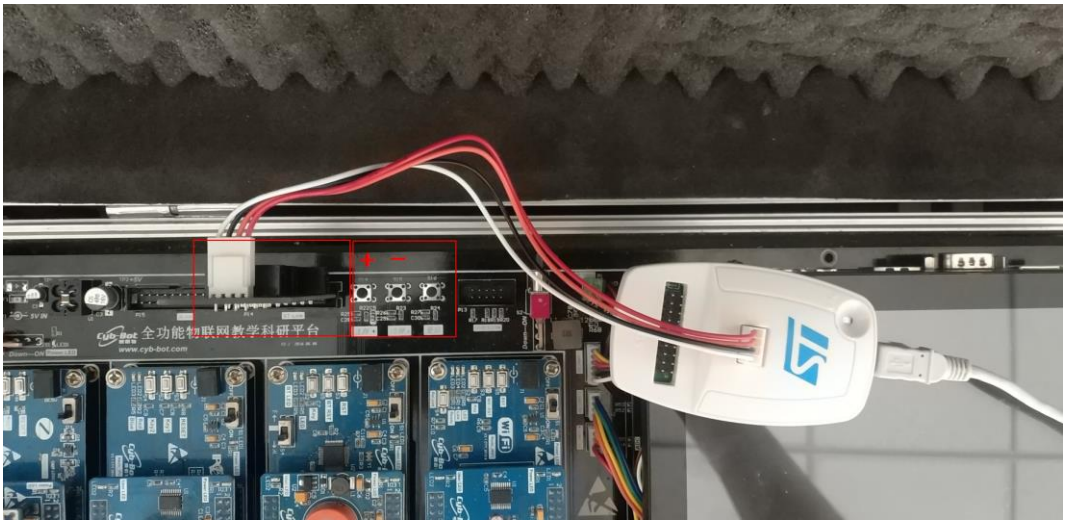
如下：

```
void main(void)
{
    u8 i = 0;
    u8 mode = 0;
    u16 num = 0;
    CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV1);          // 设置内部时钟 16M 为主时钟

    Uart1_Init();

    LED  Init();

    GPIO_Init(GPIOD, GPIO_PIN_4, GPIO_MODE_OUT_PP_LOW_SLOW);

    //Relay_On();
    for(i = 0;i < 14;i++)
        DATA_tx_buf[i] = 0;
    for(i = 0;i < 8;i++)
        CMD_rx_buf[i] = 0;

    /* 根据不同类型的传感器进行修改 */
    Sensor_Type = 0x0F;
    Sensor_ID = 1;

    CMD  ID = 1;

    DATA_tx_buf[0] = 0xEE;
    DATA_tx_buf[1] = 0xCC;
    DATA  tx  buf[2] = Sensor  Type;
    DATA  tx  buf[3] = Sensor  ID;
    DATA_tx_buf[4] = CMD_ID;
    DATA_tx_buf[13] = 0xFF;

    delay  ms(1000);

    enableInterrupts();

    while (1)
    {
  //u8 buf;

  num ++;
  if(num == 10){
      DATA  tx  buf[10] = mode;
      UART1_SendString(DATA_tx_buf, 14);
          num = 0;
  }

  if(Uart  RecvFlag == 1){

      switch(rx_buf[10]){
          case 0x00:// 断开继电器
              mode = 0;
              Uart_RecvFlag = 0;
```

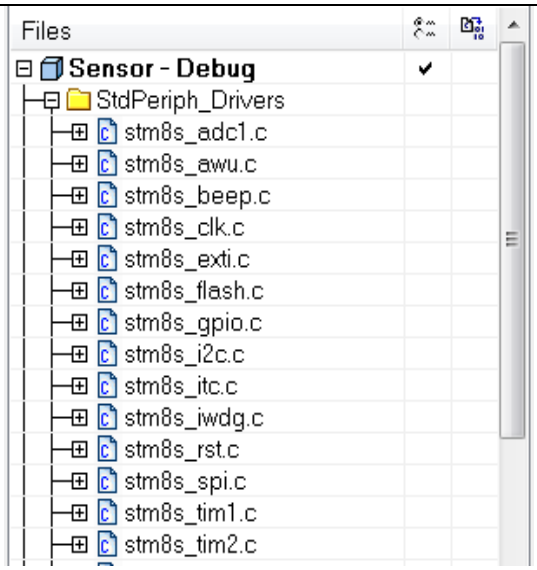
```
                                        LED_Off();
                                        Relay  Off();
                    break;
                case 0x01:// 导通继电器
                    mode = 1;
                    Uart  RecvFlag = 0;
                                        LED  On();
                                        Relay_On();
                    break;
                default:
                    Uart  RecvFlag = 0;
                    break;
            }
        }

    delay  ms(100);
            /*
            UART1_SendByte(0xAA);
            LED_Toggle();
            Relay  Toggle();
            delay  ms(1000);
            */
        }
}
```

4. 实验步骤

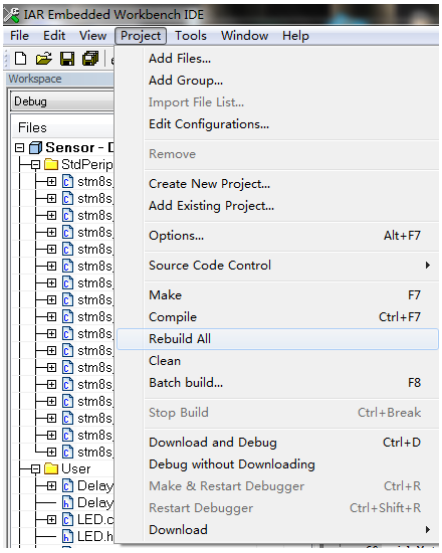
1）首先我们要把传感器插到实验箱的主板上子节点的串口上，再把 ST-Link 配合 JTAG 转接板插到标有 ST-Link 标志的 JTAG 口上，最后把仿真器一端的 USB 线插到 PC 机的 USB 端口，通过主板上的“加”“减”按键选择要编程实验的传感器（会有黄色 LED 灯提示），硬件连接完毕。



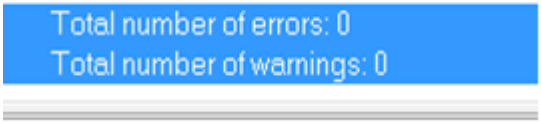
2）我们用 IAR SWSTM8 1.30 软件，打开..\ 15-Sensor_继电器\Project\Sensor.eww。



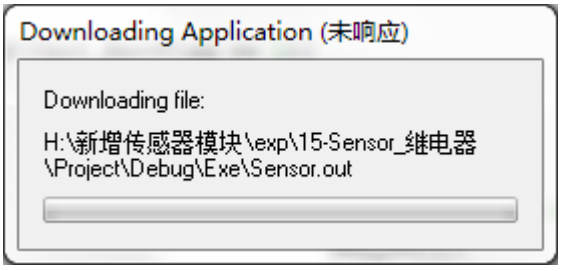
3）打开后点击“Project”下面的“Rebuild All”或者选中工程文件右键“Rebuild All”把我们的工程编译一下。



4）点击“Rebuild All”编译完后，无警告，无错误。

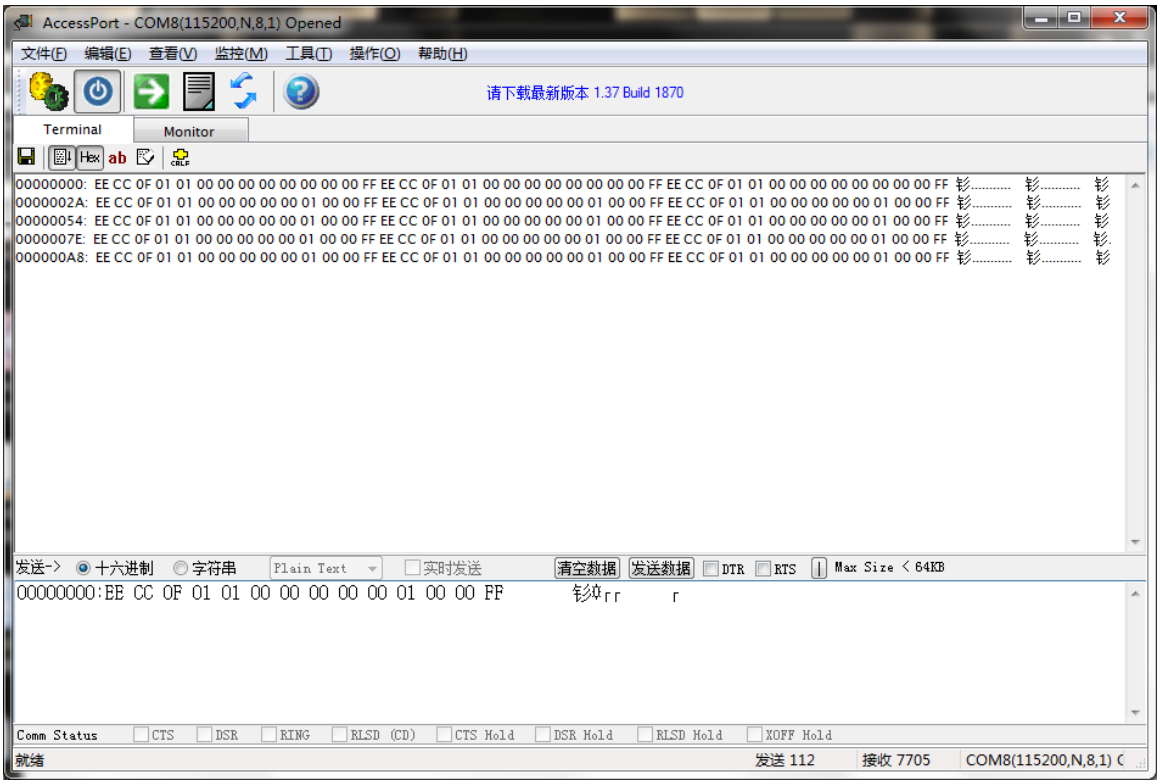


5）编译完后我们要把程序烧到模块里，点击“  ”中间的 Download and Debug 进行烧写。

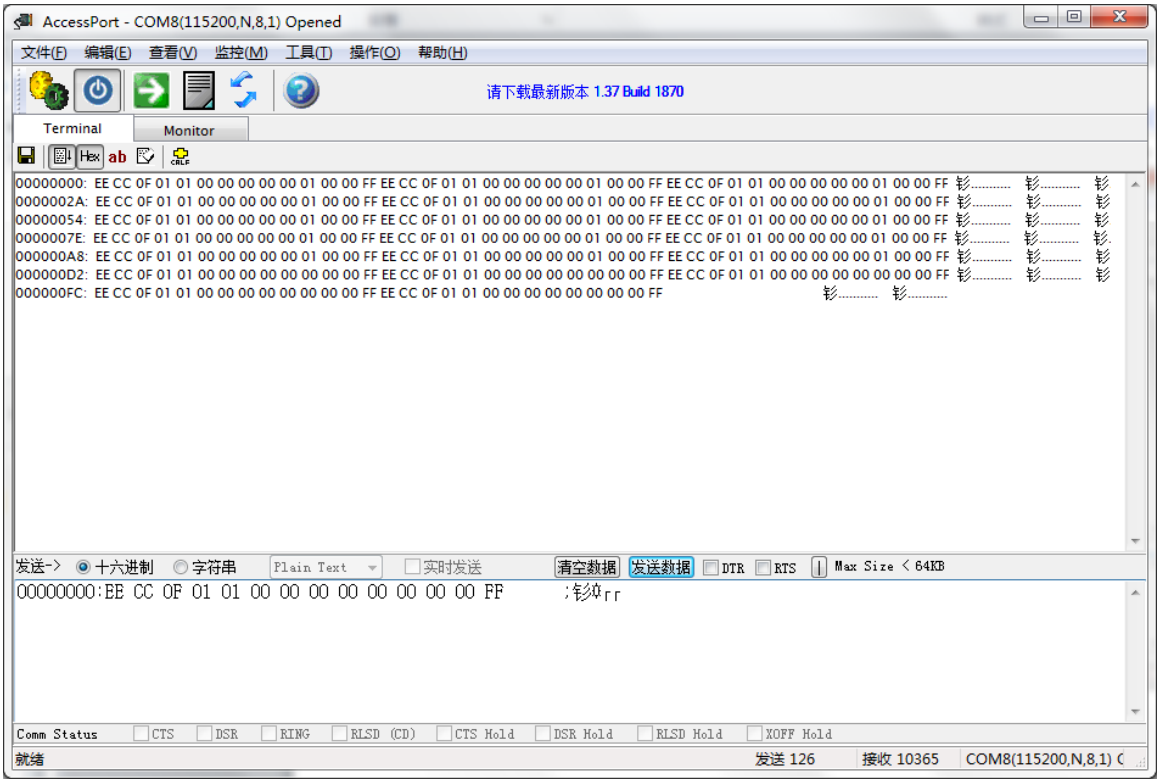


6）烧写完毕后，把传感器模块从主板上取下来，连接到平台配套的 USB 转串口模块上，将 USB2UART 模块的 USB 线连接到 PC 机的 USB 端口，然后打开串口工具，配置好串口，波特率 115200，8 个数据位，一个停止位，无校验位。

7）发送“EE CC 0F 01 01 00 00 00 00 00 01 00 00 FF”可使继电器导通，可以用万能表测试一下。



8）发送“EE CC 0F 01 01 00 00 00 00 00 00 00 FF”可使继电器断开。



第三章. 无线通讯模块之单片机（STM32）实验

本章主要介绍 STM32 无线通讯模块单片机部分的实验内容，ICS-IOT-CEP 平台配套的 Bluetooth 模块、WIFI 模块皆采用 STM32F103 处理器，IPv6 模块采用 STM32W108 处理器，两款芯片有区别。本部分实验主要是针对蓝牙模块的相关 STM32F103 单片机接口实验，可通过更改实验代码等方式扩展针对 WIFI 模块、IPv6 模块的单片机实验。内容由浅入深，通过本章实验内容，读者即可以迅速掌握 STM32 这款 ARM 硬件接口实验的开发方法。

实验一. STM32 LED 灯的控制实验

1. 实验目的

- 熟悉 IAR 开发软件的应用。
- 掌握 STM32F103CB 对 LED 的控制的方法。

2. 实验设备

- 硬件：ICS-IOT-CEP 教学实验平台，PC 机，J-OB 仿真器和 J-OB 转接板。
- 软件：IAR 开发软件。

3. 实验内容

- 阅读蓝牙模块硬件部分文档，熟悉蓝牙模块中 STM32 与 LED2 灯部分硬件相关接口。
- 用 IAR 开发环境，设计编写程序实现对 LED2 灯的控制。

4. 实验原理

4.1 硬件接口原理

- **LED 灯与 STM32F103CB 的硬件接口**

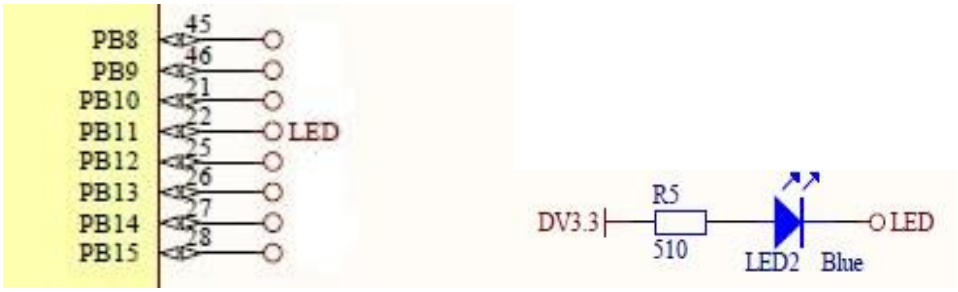


图 4.1.1 LED 灯与 STM32f103CB 连接电路

LED2 Blue 灯连接在 STM32 的 PB11 引脚上，当引脚输出为低电平时 LED2 为亮；反之 LED2 灭。通过程序语言的控制可以控制该 LED 的亮灭，实现实验。

4.2 关键代码分析

主函数。

```
int main(void)
```

```
{
    RccInitialisation(); /*  设置系统时钟  */
GpioInitialisation();/*  设置 GPIO 端口  */
 SystickInitialisation();/*  设置 Systick 定时器  */
LED_Init();      /******LED 灯初始化，初始化后状态为熄灭*******/
/**********LED 灯的状态变化**********/
while（1）
{
LED  USER  On();
      LED_USER_Off();
      LED_USER_On();
    LED_USER_Off();
       }
}
```

设置系统各部分时钟函数。

```
void RccInitialisation(void)
{
  ErrorStatus HSEStartUpStatus;
  RCC_DeInit();
  RCC_HSEConfig(RCC_HSE_ON);
  HSEStartUpStatus = RCC_WaitForHSEStartUp();
  if(HSEStartUpStatus == SUCCESS)
     {
    RCC_HCLKConfig(RCC_SYSCLK_Div1);
    RCC_PCLK2Config(RCC_HCLK_Div1);
    RCC_PCLK1Config(RCC_HCLK_Div2);
    FLASH  SetLatency(FLASH  Latency  2);
    FLASH  PrefetchBufferCmd(FLASH  PrefetchBuffer  Enable);
    RCC_PLLConfig(RCC_PLLSource_HSE_Div1, RCC_PLLMul_9);
    RCC_PLLCmd(ENABLE);
    while(RCC_GetFlagStatus(RCC_FLAG_PLLRDY) == RESET);
    RCC  SYSCLKConfig(RCC  SYSCLKSource  PLLCLK);
    while(RCC  GetSYSCLKSource() != 0x08);
     }
}
```

GPIO 初始化函数。

```
void GpioInitialisation(void)
{
 GPIO_InitTypeDef GPIO_InitStructure;
RCC_APB2PeriphClockCmd(LED_USER_GPIO_CLK, ENABLE);
GPIO  InitStructure.GPIO  Pin = LED  USER  GPIO  PIN;
GPIO  InitStructure.GPIO  Mode = GPIO  Mode  Out  PP;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
GPIO_Init(LED_USER_GPIO_PORT, &GPIO_InitStructure);
}
```

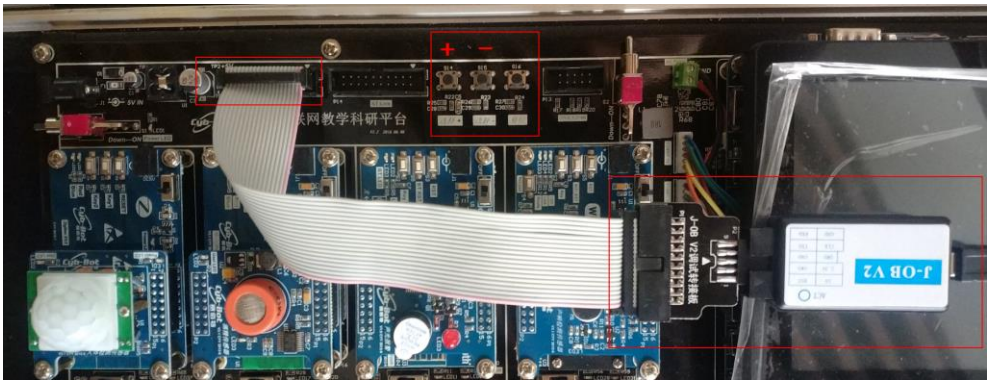
系统定时器初始化函数。

```
void SystickInitialisation(void)
{
 SysTick_CounterCmd(SysTick_Counter_Disable);
 SysTick  CLKSourceConfig(SysTick  CLKSource  HCLK);
 SysTick  CounterCmd(SysTick  Counter  Clear);
 SysTick_SetReload(72000 * 20);
 SysTick_CounterCmd(SysTick_Counter_Enable);
```

```
}
```

5. 实验步骤

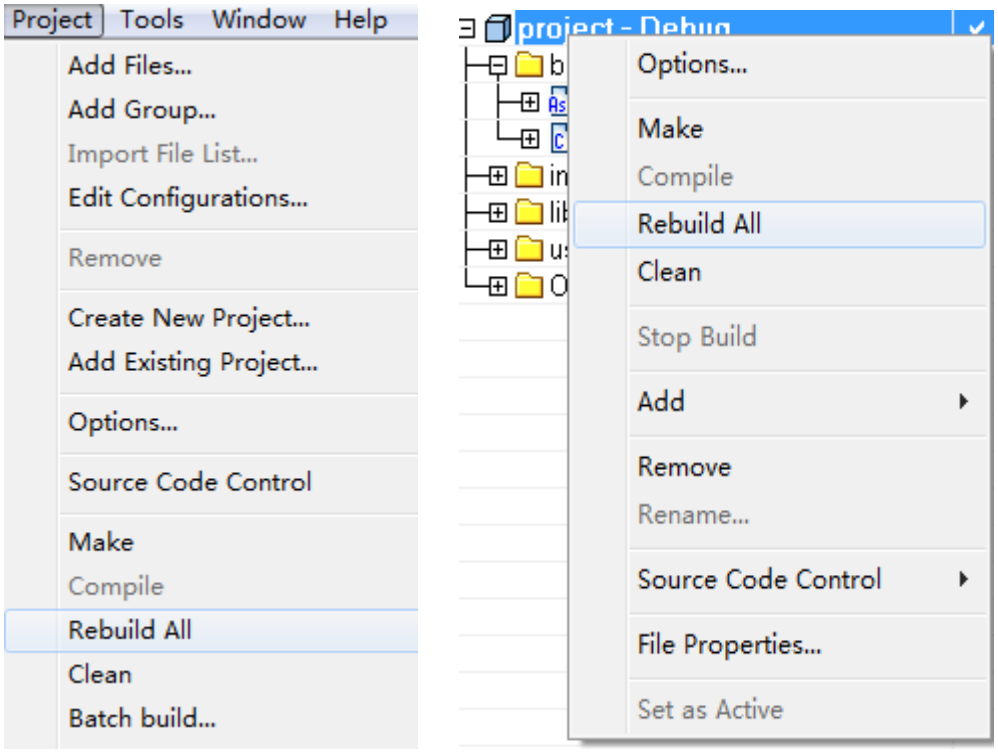
1）使用 J-OB 仿真器和 J-OB 转接板和 J-OB 转接板连接 PC 机和设备，用电源线将实验箱连上电源并给设备上电，确保打开蓝牙模块开关供电。



2）使用“加”、“减”按键选择 J-OB 仿真器和 J-OB 转接板要连接的蓝牙设备模块（根据指示灯判断）。

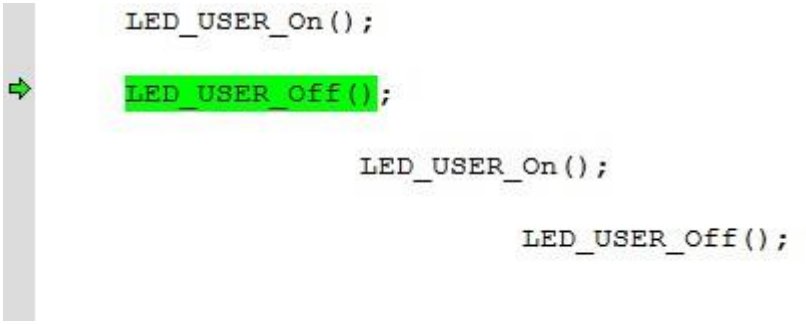
3）启动 IAR 软件，打开这次实验所用的的工程文件中运行 project 文件，文件类型为 IAR IDE Workspace，如下图：

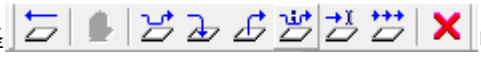
4）打开工程文件之后，需要对文件进行编译，对文件进行编译可选择 IAR 环境菜单栏中的“Project”中的“Rebuild All”，或者右键选择工程 Workspace 中“project- Debug”， 然后选择“Rebuild All”进行编译，如下图。



5）编译完成后，编译完成后选择界面中的  中间的  进行下载调试，这次采用单步调试的方法观察实验现象。

6）按 F10 进行调试,进行到下图所示的语句时观察实验现象，注意每条语句执行完后与开始时有何不同。



7）调试完成后，结束调试，可选择  中的  结束调试。

实验二. STM32 定时器实验

1. 实验目的

- 了解 STM32 TIMER 的原理及使用方法。
- 掌握 TIMER 的定时功能和对定时器的处理。

2. 实验设备

- 硬件：ICS-IOT-CEP 教学实验平台蓝牙模块，PC 机，J-OB 仿真器和 J-OB 转接板。
- 软件：IAR 开发软件。

3. 实验内容

- 预读 STM32 芯片部分文档，熟悉 TIMER，查看蓝牙模块电路原理图，熟悉 LED2 与 STM32 的连接原理图。
- 使用 IAR 开发环境，编写实验程序，实现定时器功能。

4. 实验原理

4.1 蓝牙模块 LED 灯与 STM32 连接电路图

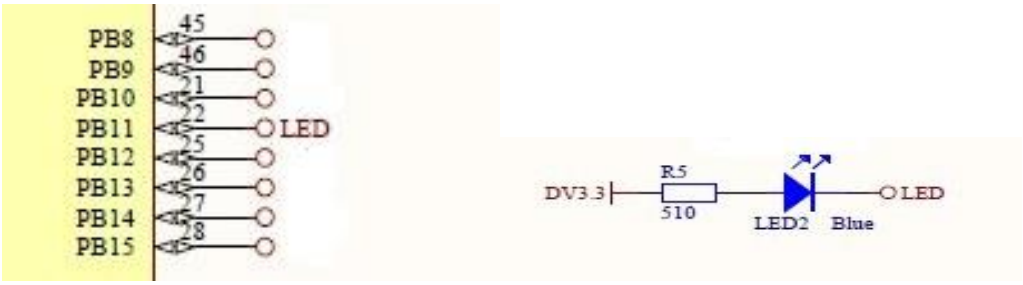


图 4.1.1 LED 灯与 STM32 连接电路图

由电路图可知，LED2 接有上拉电阻。当 STM32 引脚 PB11 输出为高电平时 LED2 状态为灭，当 PB11 输出为低电平时 LED2 为亮。

4.2 定时器功能及功能描述

♦ **定时器功能**

通用定时器功能有 16 位向上、向下、向上/向下自动装载定时器；16 位可编程预分频器，计数器时钟频率的分频系数为 1~65536 之间的任意数值；输入捕获、输出比较、PWM

生成、单脉冲模式输出 4 个独立通道；使用外部信号控制定时器和定时器互连的同步电路；当计数器向上溢出/向下溢出，计数器初始化或触发事件（计数器启动、停止、初始化或者由内部外部触发计数器）或输入比较或输出比较或支持针对定位的增量（正交）编码器和霍尔传感电路或者处罚输入作为外部时钟的情况发生时产生中断。

- **功能描述**

时基单元：可编程通用定时器主要部分是一个 16 位计数器和与其相关的自动装载寄存器。这个计数器可以向上计数、向下计数或者向上向下双向计数。

向上计数模式：向上计数模式中，计数器从 0 计数到自动加载值，然后重新从 0 开始计数并产生一个计数器溢出事件。每次计数器溢出时可以产生更新事件，在 TIMx_EGR 寄存器中设置 UG 位也同样可以产生一个更新事件。

向下计数模式：向下计数模式中，计数器从自动装入的值开始向下计数到 0，然后从自动装入的值重新开始并且产生一个计数器向下溢出事件。

中央对齐模式：在中央对齐模式，计数器从 0 开始计数到自动加载的值-1，产生一个计数器溢出事件，然后向下计数到 1 并且产生一个计数器向下溢出事件；然后在从 0 开始重新计数。在这个模式中，不能写入 TIM_CR1 中的 DIR 方向位。它由硬件更新并指示当前的计数方向。更新事件可以产生在每次计数溢出和每次计数下溢；也可以通过（软件或者使用从模式控制器）设置 TIMx_EGR 寄存器中的 UG 位产生，此时，计数器重新从 0 开始计数，预分频器也从 0 开始计数。

4.3 关键代码分析

主函数部分。

```
int main(void)
{
  /* 设置系统时钟 */
  RCC_Configuration();
  /* 设置 NVIC */
  NVIC_Configuration();
  /* 初始化 GPIO 端口 */
  GpioInitialisation();
  /* 设置 TIM */
  TIM_Configuration();
/*******循环函数，等待定时器中断的产生********/
  while (1);
  {
  }
}
```

设置系统各部分时钟函数。

```
void RCC_Configuration(void)
{
  /* 定义枚举类型变量 HSEStartUpStatus */
  ErrorStatus HSEStartUpStatus;
       /* 复位系统时钟设置 */
       RCC_DeInit();
```

```
        /* 开启 HSE */
        RCC_HSEConfig(RCC_HSE_ON);
        /* 等待 HSE 起振并稳定 */
        HSEStartUpStatus = RCC_WaitForHSEStartUp();
    /* 判断 HSE 起是否振成功，是则进入 if()内部 */
        if(HSEStartUpStatus == SUCCESS)
        {
        /* 选择 HCLK（AHB）时钟源为 SYSCLK 1 分频 */
        RCC_HCLKConfig(RCC_SYSCLK_Div1);
        /* 选择 PCLK2 时钟源为 HCLK（AHB） 1 分频 */
        RCC_PCLK2Config(RCC_HCLK_Div1);
        /* 选择 PCLK1 时钟源为 HCLK（AHB） 2 分频 */
        RCC PCLK1Config(RCC HCLK Div2);
        /* 设置 FLASH 延时周期数为 2 */
        FLASH_SetLatency(FLASH_Latency_2);
        /* 使能 FLASH 预取缓存 */
        FLASH PrefetchBufferCmd(FLASH PrefetchBuffer Enable);
        /* 选择锁相环（PLL）时钟源为 HSE 1 分频，倍频数为 9，则输出频率为 72MHz */
        RCC_PLLConfig(RCC_PLLSource_HSE_Div1, RCC_PLLMul_9);
        /* 使能 PLL */
        RCC_PLLCmd(ENABLE);
        /* 等待 PLL 输出稳定 */
        while(RCC_GetFlagStatus(RCC_FLAG_PLLRDY) == RESET);
        /* 选择 SYSCLK 时钟源为 PLL */
        RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK);
        /* 等待 PLL 成为 SYSCLK 时钟源 */
        while(RCC_GetSYSCLKSource() != 0x08);
        }
/* 打开 TIM2 时钟 */
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);
        /* 打开 APB 总线上的 GPIOA，USART1 时钟 */
        RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
}
```

初始化 GPIO 端口。

```
    void GpioInitialisation(void)
{
      /* 定义 GPIO 初始化结构体 GPIO_InitStructure*/
      GPIO_InitTypeDef GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);
        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11;
        GPIO InitStructure.GPIO Mode = GPIO Mode Out PP;
        GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
        GPIO_Init(GPIOB, &GPIO_InitStructure);
}
```

设置 TIM 各通道函数。

```
void TIM Configuration(void)
{
  /* 定义 TIM_TimeBase 初始化结构体 TIM_TimeBaseStructure */
  TIM TimeBaseInitTypeDef  TIM TimeBaseStructure;
  /* 定义 TIM_OCInit 初始化结构体 TIM_OCInitStructure */
  TIM_OCInitTypeDef  TIM_OCInitStructure;
  /*
  *    计数重载值为 65535
  *    预分频值为(7199 + 1 = 7200)
```

```
    *     时钟分割 0
    *     向上计数模式
    */
    TIM_TimeBaseStructure.TIM_Period = 10000;
    TIM_TimeBaseStructure.TIM_Prescaler = 0;
    TIM_TimeBaseStructure.TIM_ClockDivision = 0;
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
    TIM_TimeBaseInit(TIM2 , &TIM_TimeBaseStructure);
    /* 设置预分频值，且立即装入 */
    TIM_PrescalerConfig(TIM2 , 7199 , TIM_PSCReloadMode_Immediate);
/*
    *     设置 OC1 通道
    *     工作模式为计数器模式
    *     使能比较匹配输出极性
    *     时钟分割 0
    *     向上计数模式
    */
TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_Timing;
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;
    TIM_OC1Init(TIM2, &TIM_OCInitStructure);

    /* 禁止预装载寄存器 */
    TIM_OC1PreloadConfig(TIM2 , TIM_OCPreload_Disable);
    TIM_OC2PreloadConfig(TIM2 , TIM_OCPreload_Disable);
    TIM_OC3PreloadConfig(TIM2 , TIM_OCPreload_Disable);
    TIM_OC4PreloadConfig(TIM2 , TIM_OCPreload_Disable);
    /* 使能 TIM 中断 */
    TIM_ITConfig(TIM2 , TIM_IT_CC1 , ENABLE);
    /* 启动 TIM 计数 */
    TIM_Cmd(TIM2 , ENABLE);
}
```

设置 NVIC 参数函数。

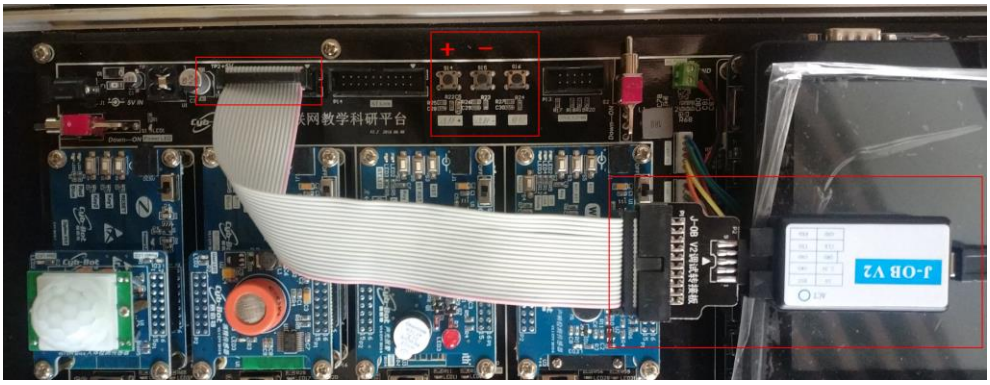
```
void NVIC_Configuration(void)
{
    /* 定义 NVIC 初始化结构体 */
    NVIC_InitTypeDef NVIC_InitStructure;
        /* #ifdef...#else...#endif 结构的作用是根据预编译条件决定中断向量表起始地址*/
#ifdef  VECT_TAB_RAM
            /* 中断向量表起始地址从 0x20000000 开始 */
        NVIC_SetVectorTable(NVIC_VectTab_RAM , 0x0);
#else   /* VECT_TAB_FLASH */
            /* 中断向量表起始地址从 0x80000000 开始 */
            NVIC_SetVectorTable(NVIC_VectTab_FLASH , 0x0);
#endif
    /* 选择优先级分组 0 */
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0);
    /* 开启 TIM2 中断, 0 级先占优先级，0 级后占优先级 */
    NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQChannel;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}
```

定时器 2 中断处理函数。

```
void TIM2_IRQHandler(void)
{
  vu16  capture = 0;                    /* 当前捕获计数值局部变量 */
  /*
  *    TIM2 时钟 = 72 MHz, 分频数 = 7299 + 1, TIM2 counter clock = 10KHz
  *    CC1 更新率 = TIM2 counter clock / CCRx_Val
  */
  if (TIM_GetITStatus(TIM2, TIM_IT_CC1) != RESET)
  {
      GPIO_WriteBit(GPIOB, GPIO_Pin_11, (BitAction)(1 - GPIO_ReadOutputDataBit(GPIOB, GPIO_Pin_11)));
      /* 读出当前计数值 */
      capture = TIM_GetCapture1(TIM2);
      /* 根据当前计数值更新输出捕获寄存器 */
      TIM_SetCompare1(TIM2, capture + CCR1_Val);
      TIM_ClearITPendingBit(TIM2, TIM_IT_CC1);
  }
}
```

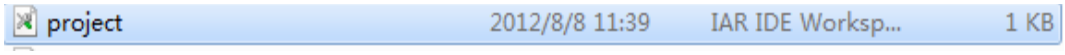
5. 实验步骤

1）使用 J-OB 仿真器和 J-OB 转接板和 J-OB 转接板连接 PC 机和设备，用电源线将实验箱连上电源并给设备上电，确保打开蓝牙模块开关供电。

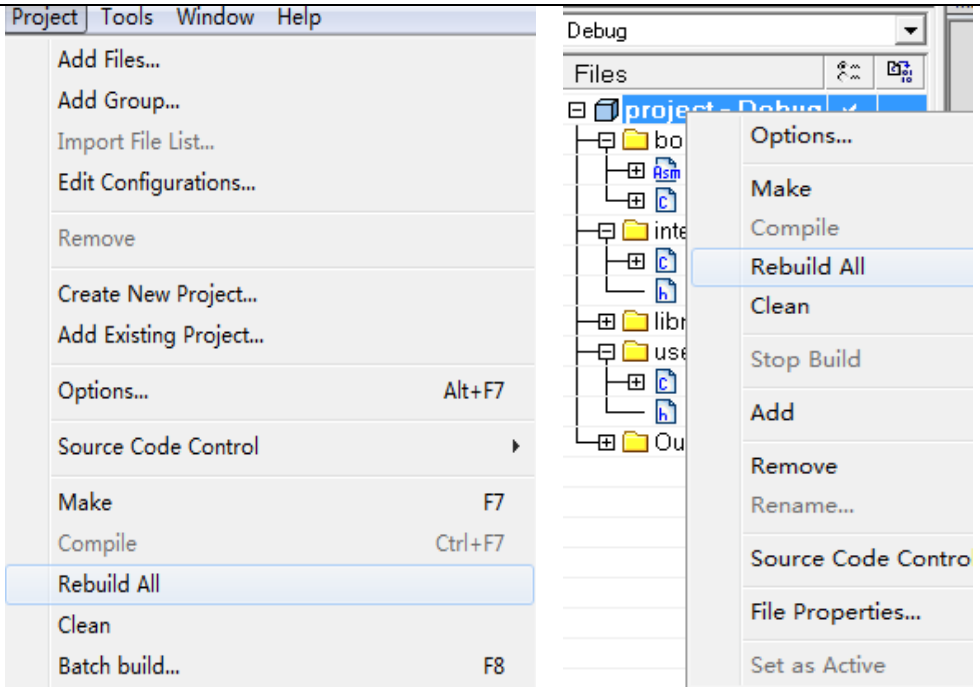


2）使用实验箱上的“+”、“-”按钮选择需要下载调试的蓝牙模块，可根据 LED 灯进行判断。

3）启动 IAR 软件，选择工程文件并打开。如下图：



4）将打开的的工程进行编译，可以选择界面上的“project”中的“Rebuild All ”或者选中工程右键选择“Rebuild All ”选项。如下图：



5）编译完成后，用界面中中的  按钮将编译完成的程序下载到步骤 2）中选择的模块中。

6）下载完成后，使用  中的  全速执行程序，观察模块上 LED2 的变化，定时器每触发一次中断，在中断处理的过程中灯的状态就会发生一次变化。

实验三. STM32 A/D 转换实验

1. 实验目的

- 熟悉 STM32 芯片 A/DC 的使用。
- 掌握用 STM32 实现 A/DC 的原理及方法。

2. 实验设备

- 硬件：ICS-IOT-CEP 教学实验平台，PC 机，J-OB 仿真器和 J-OB 转接板。
- 软件：IAR 开发软件，串口工具。

3. 实验内容

- 阅读有关 STM32F103CB 文档，熟悉 STM32F103CB 芯片 A/DC 转化的原理和使用方法。
- 编写程序，观察内部参照电压 V_{REFINT} 的实时值。
- 利用 IAR 开发环境，编写程序实现 A/DC。

4. 实验原理

4.1 A/DC 介绍

STM32F103 处理器自带的 12 位 ADC 是一种逐次逼近型模拟数字转换器。他有 18 个通道，可测量 16 个外部和两个内部信号源。各通道的 A/D 转换可以单次、连续、扫描或间断模式执行。ADC 的结果可以左对齐或右对齐的方式存储在 16 位数据寄存器中。

模拟看门狗特性允许应用程序检测输入电压是否超出用户定义的高/低阈值值。

4.2 A/DC 功能描述

- **ADC 管脚**

名称	信号类型	注解
V_{REF+}	输入，摸拟参考正极	ADC 使用的高端/正极参考电压， $2.4V \leqslant V_{REF+} \leqslant V_{DDA}$
V_{DDA}	输入，模拟电源	等效于 V_{DD} 的模拟电源且，

		2.4V≤V_{DDA}≤V_{DD} (3.6V)
V_{REF-}	输入，模拟参考负极	ADC 是用的低端负极参考电压，V_{REF-} =V_{SSA}
V_{SSA}	输入，模拟电源地	等效于 V_{SS} 的模拟电源地
ADC_IN[15:0]	模拟输入信号	16 个模拟输入通道

表 4.2.1 ADC 管脚

- **ADC 开关控制**

通过设置 ADC_CR1 寄存器给 ADON 位可给 ADC 上电。当第一次设置 ADON 位时，它将 ADC 从断电状态下唤醒。

当 ADC 上电延迟一段时间后，再次设置 ADON 位时开始进行转换。

通过清除 ADON 位可以停止转换，并将 ADC 置于断电模式。在这个模式中，ADC 几乎不耗电。

- **A/DC 时钟**

由于时钟控制器提供的 ADCCLK 时钟和 PCLK2（APB2S 时钟）同步。RCC 控制器为 ADC 时钟提供一个专用的可编程预分频器。

4.3 关键代码分析

主函数部分。

```
int main(void)
{
  float VolValue = 0.00;              /* 转换结果，双精度浮点数 */
  u32 ticks = 0;                      /* ADC 显示延时参数 */

      /* 设置系统时钟 */
      RCC_Configuration();
      /* 设置 GPIO 端口 */
      GPIO_Configuration();
      /* 设置 USART */
  USART_Configuration();
  /* 设置 ADC */
  ADC_Configuration();
printf("\r\n The AD_value is：---------- \r\n");
while(1)
  {
      if (ticks++ >= 2000000)
      {
```

```
            ticks = 0;
            VolValue = 2.56 * ADC_GetConversionValue(ADC1) / 0X0FFF;
            printf( "\r\nThe current VolValue = %.2fv\r\n", VolValue);
        }
    }
}
```

设置系统各部分时钟函数。

```
void RCC_Configuration(void)
{
  /* 定义枚举类型变量 HSEStartUpStatus */
  ErrorStatus HSEStartUpStatus;
        /* 复位系统时钟设置 */
        RCC_DeInit();
        /* 开启 HSE */
        RCC_HSEConfig(RCC_HSE_ON);
        /* 等待 HSE 起振并稳定 */
        HSEStartUpStatus = RCC_WaitForHSEStartUp();
  /* 判断 HSE 起是否振成功，是则进入 if()内部 */
        if(HSEStartUpStatus == SUCCESS)
        {
        /* 选择 HCLK（AHB）时钟源为 SYSCLK 1 分频 */
        RCC_HCLKConfig(RCC_SYSCLK_Div1);
        /* 选择 PCLK2 时钟源为 HCLK（AHB）1 分频 */
        RCC_PCLK2Config(RCC_HCLK_Div1);
        /* 选择 PCLK1 时钟源为 HCLK（AHB）2 分频 */
        RCC_PCLK1Config(RCC_HCLK_Div2);
        /* 设置 FLASH 延时周期数为 2 */
        FLASH_SetLatency(FLASH_Latency_2);
        /* 使能 FLASH 预取缓存 */
        FLASH_PrefetchBufferCmd(FLASH_PrefetchBuffer_Enable);
        /* 选择锁相环（PLL）时钟源为 HSE 1 分频，倍频数为 9，则 PLL 输出频率为 8MHz * 9
= 72MHz */
        RCC_PLLConfig(RCC_PLLSource_HSE_Div1, RCC_PLLMul_9);
        /* 使能 PLL */
        RCC_PLLCmd(ENABLE);
        /* 等待 PLL 输出稳定 */
        while(RCC_GetFlagStatus(RCC_FLAG_PLLRDY) == RESET);
        /* 选择 SYSCLK 时钟源为 PLL */
        RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK);
        /* 等待 PLL 成为 SYSCLK 时钟源 */
        while(RCC_GetSYSCLKSource() != 0x08);
        }
        /* 使能各个用到的外设时钟 */
        RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1 | RCC_APB2Periph_GPIOA |
                               RCC_APB2Periph_ADC1 | RCC_APB2Periph_GPIOB, ENABLE);
}
```

设置各 GPIO 端口功能函数。

```
void GPIO_Configuration(void)
{
  /* 定义 GPIO 初始化结构体 GPIO_InitStructure */
        GPIO_InitTypeDef GPIO_InitStructure;

  /* 设置 USART1 的 Tx 脚（PA.9）为第二功能推挽输出功能 */
        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
```

```
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA , &GPIO_InitStructure);

    /* 设置 USART1 的 Rx 脚（PA.10）为浮空输入脚 */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_Init(GPIOA , &GPIO_InitStructure);
/* 将 PB.0 设置为模拟输入脚 */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;
    GPIO_Init(GPIOB , &GPIO_InitStructure);
}
```

初始化并启动 ADC 转换函数。

```
void ADC_Configuration(void)
{
  /* 定义 ADC 初始化结构体 ADC_InitStructure */
  ADC_InitTypeDef ADC_InitStructure;
  /* 配置 ADC 时钟分频 */
  RCC_ADCCLKConfig(RCC_PCLK2_Div4);
  /** 独立工作模式;
  *   多通道扫描模式式;
  *   转换触发方式：;
  *   连续模数转换模转换由软件触发启动;
  *   ADC 数据右对齐 ;
  *   进行规则转换的 ADC 通道的数目为 1;
  */
  ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;
  ADC_InitStructure.ADC_ScanConvMode = ENABLE;
  ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
  ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;
  ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
  ADC_InitStructure.ADC_NbrOfChannel = 1;
  ADC_Init(ADC1, &ADC_InitStructure);
  /* 设置 ADC1 使用 8 转换通道，转换顺序 1，采样时间为 55.5 周期 */
  ADC_RegularChannelConfig(ADC1, ADC_Channel_8, 1, ADC_SampleTime_55Cycles5);
  /* 使能 ADC1 */
  ADC_Cmd(ADC1, ENABLE);
  /* 复位 ADC1 的校准寄存器 */
  ADC_ResetCalibration(ADC1);
  /* 等待 ADC1 校准寄存器复位完成 */
  while(ADC_GetResetCalibrationStatus(ADC1));
  /* 开始 ADC1 校准 */
  ADC_StartCalibration(ADC1);
  /* 等待 ADC1 校准完成 */
  while(ADC_GetCalibrationStatus(ADC1));
  /* 启动 ADC1 转换 */
  ADC_SoftwareStartConvCmd(ADC1, ENABLE);
}
```

设置 USART1 函数。

```
void USART_Configuration(void)
{
  /* 定义 USART 初始化结构体 USART_InitStructure */
    USART_InitTypeDef USART_InitStructure;
```

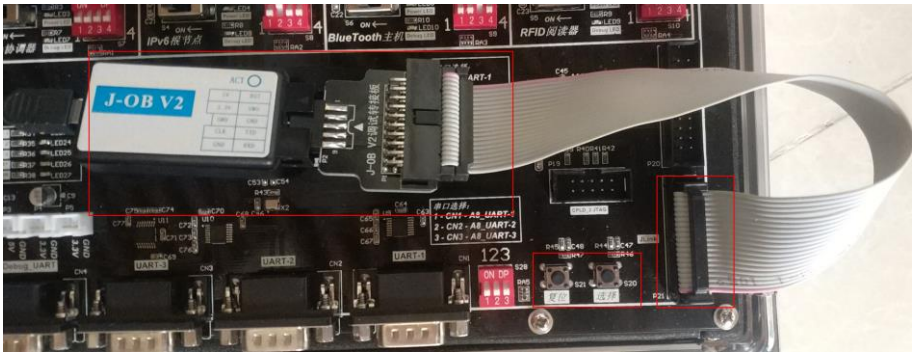
```
    /*    波特率为 115200bps;
    *     8 位数据长度;
    *     1 个停止位，无校验;
    *     禁用硬件流控制;
    *     禁止 USART 时钟;
    *     时钟极性低;
    *     在第 2 个边沿捕获数据
    *     最后一位数据的时钟脉冲不从 SCLK 输出；  */
USART_InitStructure.USART_BaudRate = 115200;
   USART_InitStructure.USART_WordLength = USART_WordLength_8b;
   USART_InitStructure.USART_StopBits = USART_StopBits_1;
   USART_InitStructure.USART_Parity = USART_Parity_No ;
   USART_InitStructure.USART_HardwareFlowControl=
USART_HardwareFlowControl_None;
   USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
   USART_Init(USART1 , &USART_InitStructure);
      /* 使能 USART1 */
        USART_Cmd(USART1 , ENABLE);
}
```

将 printf 函数重定位到 USATR1 函数。

```
int fputc(int ch, FILE *f)
{
   USART_SendData(USART1, (u8) ch);
   while(USART_GetFlagStatus(USART1, USART_FLAG_TC) == RESET);
   return ch;
}
```

5. 实验步骤

1）使用 J-OB 仿真器和 J-OB 转接板将实验设备和 PC 机连在一起，用实验箱配套的电源线将实验箱连在电源上并给实验箱上电，打开蓝牙根节点模块（A53 平台下方）电源开关。

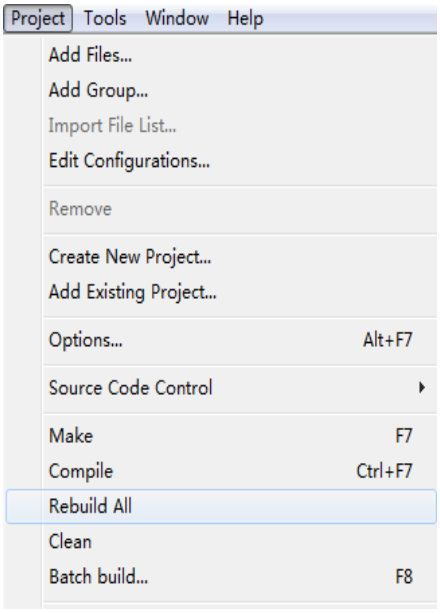


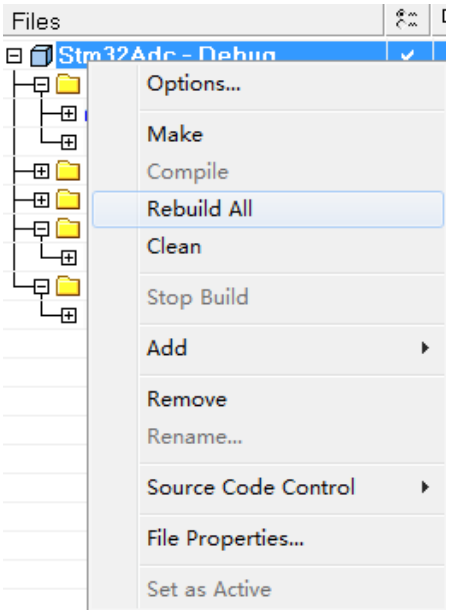
2）由于实验中用到串口设备，因此本次实验推荐使用蓝牙根节点模块。并通过根节点模块下方的 4 组拨码开关，选择蓝牙根节点使用 Debug Uart 设备与 PC 机电脑相连接。

3）使用实验箱上的“选择”按键选择需要下载调试的模块，可根据 LED 灯判断。

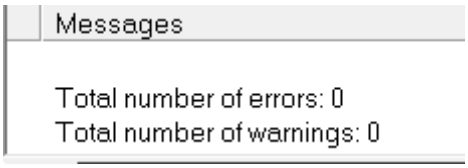
4）使用 IAR 开发环境，选择实验工程的文件夹，选择文件并打开。如下图：

5）打开工程后，使用界面中"project"中的"Rebuild All "或者选中工程文件右键选择"Rebuild All "对实验的工程文件进行编译。



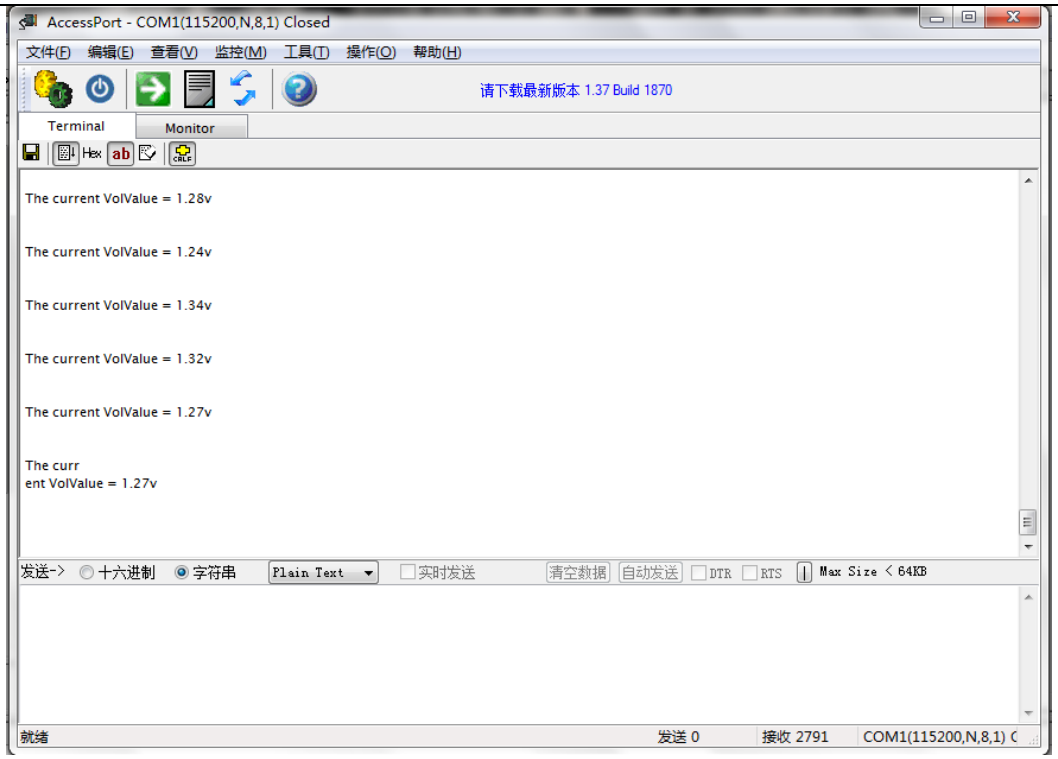


6）编译完成后提示框中会出现编译情况的提示信息。



7）使用界面中  选项中的  将实验程序下载到实验步骤 2）所选中的模块中。

8）将双母口的串口线将实验平台上的 UART-Debug 串口与 PC 机相连，打开串口软件对串口进行设置，波特率为 115200，检验位为 NONE，数据位为 8，停止位 1，然后打开串口，观察实验线结果。



实验四. STM32 按键控制实验

1. 实验目的

- 熟悉 EWARM　IAR 开发软件的应用
- 熟悉 STM32 外部中断的使用方法
- 掌握 STM32 按键的控制方法

2. 实验设备

- 硬件：ICS-IOT-CEP 教学实验平台，PC 机，J-OB 仿真器和 J-OB 转接板
- 软件：EWARM　IAR 开发软件

3. 实验内容

- 阅读蓝牙模块硬件部分文档，熟悉蓝牙模块部分硬件相关接口
- 用 EWARM　IAR 开发环境，设计编写程序实现通过按键对 LED 灯的控制

4. 实验原理

4.1 STM32 外部中断简介

♦ **外部中断/事件控制器 EXTI**

STM32F103 ARM 处理器外部中断/事件控制器由 19 个产生事件/中断要求的边沿检测器组成。每个输入线可以独立地配置输入类型(脉冲或挂起)和对应的触发事件(上升沿或下降沿或者双边沿都触发)。每个输入线都可以被独立的屏蔽。挂起寄存器保持着状态线的中断要求。

♦ **外部中断/事件线路映像**

通用 I/O 端口以下图的方式连接到 16 个外部中断/事件线上：

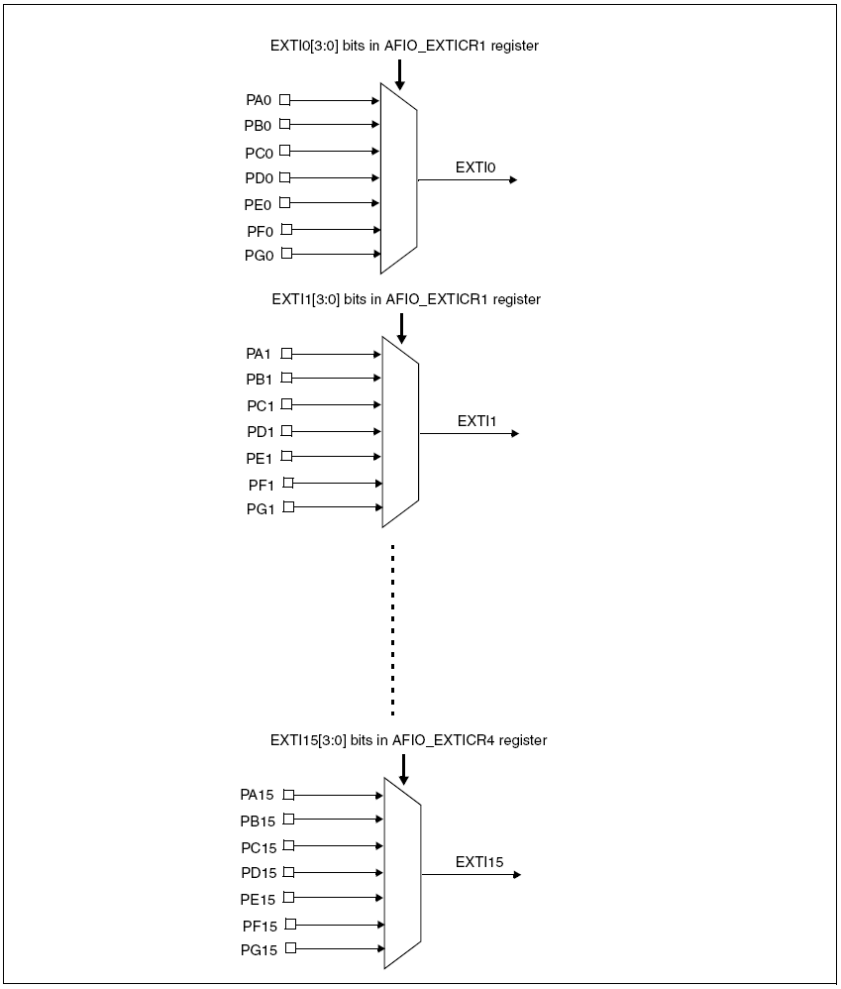


图 4.1.1 外部中断通用 I/0 映像

另外三种其他的外部中断/事件控制器的连接如下：

1）EXTI 线 16 连接到 PVD 输出

2）EXTI 线 17 连接到 RTC 闹钟事件

3）EXTI 线 18 连接到 USB 唤醒事件

4.2 硬件连接图

- 按键及 LED2 与 STM32 连接原理图

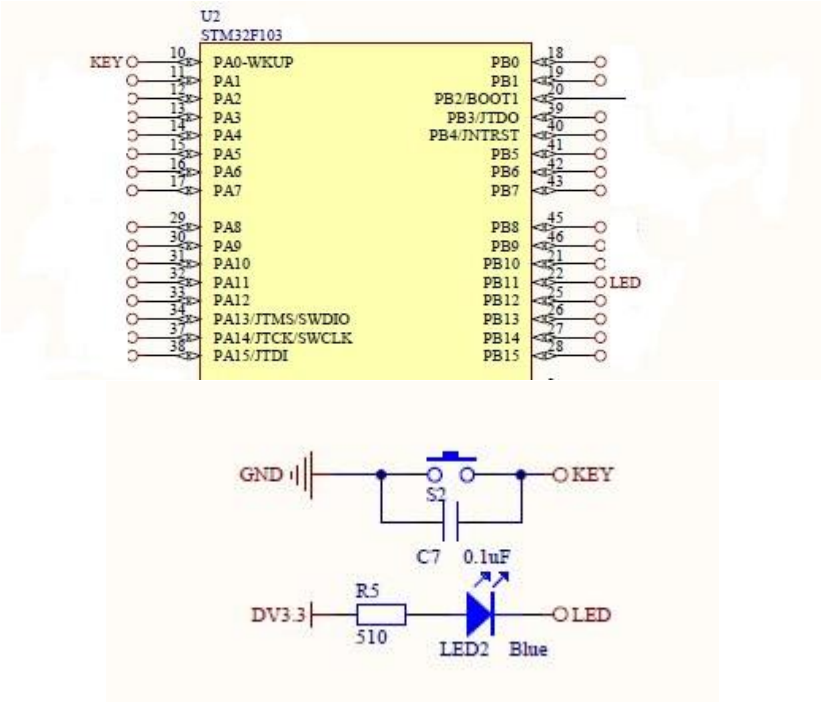


图 4.1.2 按键及 LED2 与 STM32 连接原理图

按键连接在 STM32 的 PA0 管脚上，LED 灯连接在 STM32 的 PB8 管脚上，通过编写程序，使 PB11 的引脚输出为低电平时 LED2 亮，PB11 输出为高电平时 LED 灯灭，可以实现按键对 LED 灯的控制。

4.3 关键代码分析

主函数部分。

```
int main()
{
        RCC_Configuration();        //系统时钟配置函数
        GPIO_Configuration();       //GPIO 配置函数
        EXTI_Configuration();       //中断配置函数
        NVIC_Configuration();       //中断嵌套配置函数
      /**********循环函数，等待中断***********/
      while(1)
      {
      }
}
```

系统时钟配置函数。

```
void RCC_Configuration(void)
{
```

```
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB | RCC_APB2Periph_AFIO, ENABLE);        //
开 GPIOB 和复用 IO 引脚
}
```

中断嵌套配置函数。

```
void NVIC_Configuration(void)
{
    NVIC_InitTypeDef NVIC_InitStructure;      //中断嵌套初始化
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //选择优先分组第一组
    NVIC_InitStructure.NVIC_IRQChannel = EXTI0_IRQChannel;//使能 EXTI0 中断
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;//指定抢占优先级别 0
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;//指定响应优先级别 0
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}
```

GPIO 配置函数。

```
void GPIO_Configuration(void)
{
    GPIO  InitTypeDef GPIO  InitStructure;

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;//中断引脚为第 0 引脚
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;//GPIO 方式为上拉输入
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;//GPIO 时钟为 50MHz
    GPIO_Init(GPIOA,&GPIO_InitStructure);//GPIOB 初始化

    GPIO  InitStructure.GPIO  Pin = GPIO  Pin  11;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;//GPIO 方式为推挽输出
    GPIO_Init(GPIOB,&GPIO_InitStructure);
}
```

终端配置函数。

```
void EXTI_Configuration(void)
{
    EXTI_InitTypeDef EXTI_InitStructure;

    EXTI_ClearITPendingBit(EXTI_Line0);//清除中断标志位
    GPIO_EXTILineConfig(GPIO_PortSourceGPIOB, GPIO_PinSource0);//中断源为 GPIOA0
    EXTI_InitStructure.EXTI_Line = EXTI_Line0;//选择中断线为 0
    EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;//中断方式为事件中断
    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;//中断触发方式为上升沿
    EXTI_InitStructure.EXTI_LineCmd = ENABLE;//中断使能
    EXTI_Init(&EXTI_InitStructure);
}
```

中断处理函数。

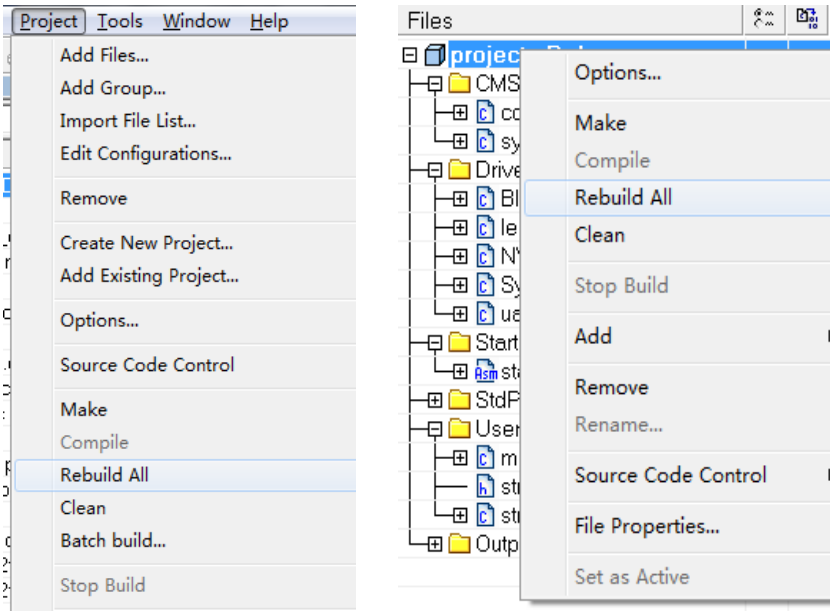
```
void EXTI0  IRQHandler(void)
{
     GPIO_WriteBit(GPIOB,GPIO_Pin_11,      (BitAction)(1      -      GPIO_ReadOutputDataBit(GPIOB,
GPIO_Pin_11)));
    EXTI_ClearITPendingBit(EXTI_Line0); //清除中断标志位
}
```

5. 实验步骤

1）使用配套 USB 线连接 PC 机和设备,设备上电，确保打开蓝牙模块开关供电。

2）使用“加”、“减”按键选择仿真器要连接的蓝牙设备模块（根据 LED 指示灯判断）。

3）用 IAR 软件打开实验工程，将实验工程进行编译，具体方法可以选择“Project”中的“Rebuild All”或者选中工程栏中的工程文件然后右键选择“Rebuild All ”进行编译。



4）将编译好的实验工程烧录进实验箱上的蓝牙模块中。

5）按下蓝牙模块上的 KEY 按键，可以观察到每按下一次按键，模块上的 LED2 灯的状态会发生一次变化。

第四章. 无线通讯模块之 Bluetooth 通信实验

本章主要介绍无线通讯模块部分的 Bluetooth 通信的实验内容，采用 ICS-IOT-CEP 平台配套的 Bluetooth 模块。内容由浅入深，前面已经讲述相应模块的硬件接口实验，本章主要针对组网实验及传感器网络实验等。通过本章实验内容，读者即可以迅速掌握基上述 Blueto oth 无线模块的开发方法，以及相应网络结构的传感器数据通讯应用设计。

实验一. BlueTooth 组网实验

1. 实验目的

- 熟悉 IAR 开发软件的应用
- 熟悉两个 BlueTooth 模块之间组网
- 掌握蓝牙之间组网原理

2. 实验设备

- 硬件:BlueTooth 模块（2 个），PC 机，J-OB 仿真器和 J-OB 转接板
- 软件:IAR 开发软件，串口助手

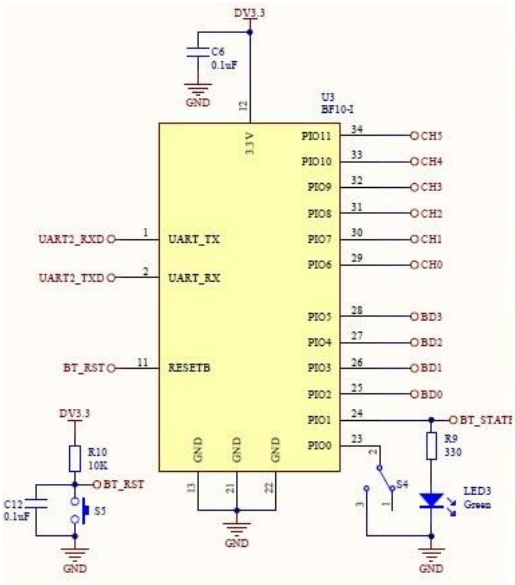
3. 实验内容

- 阅读 Bluetooth 硬件部分文档，熟悉 Bluetooth 相关硬件接口。
- 阅读 Bluetooth 硬件接口定义，熟悉 Bluetooth 模块的连接使用。
- 使用 IAR 开发环境设计程序，实现 Bluetooth 模块之间的点对点的组网配置。

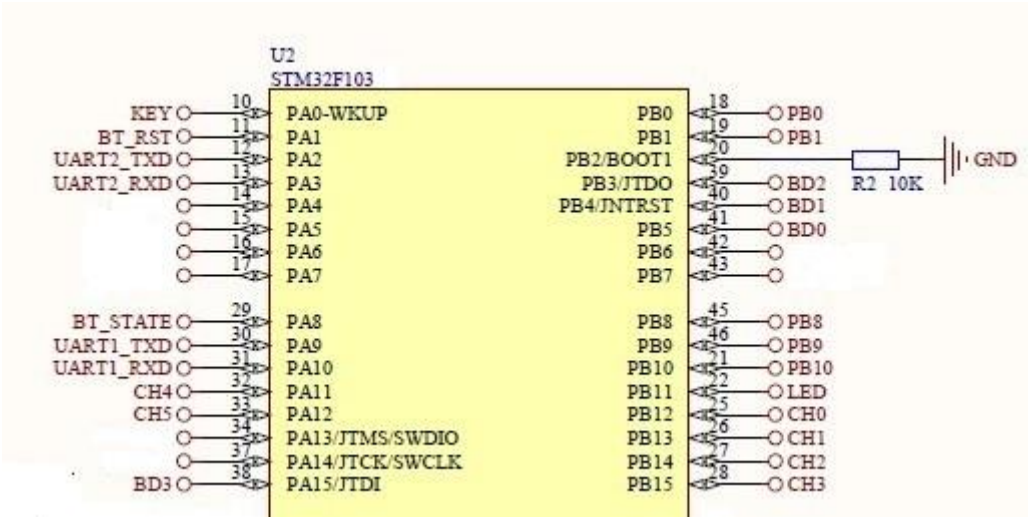
4. 实验原理

4.1 硬件接口原理

蓝牙模块与 STM32 连接原理图。



蓝牙模块原理图，其中 BF10-I 型模块与蓝牙 CC2540 模块硬件管脚兼容，此处原理图仍沿用 BF10-I 型。



该蓝牙模块已经内置应用程序，用户只需要通过蓝牙模块的串口管脚 UART_TX 和 UART_RX，发送 AT 指令即可完成对蓝牙模块的配置和控制。

上图为蓝牙模块和 STM32 连接的电路图，STM32 的 UART2 与蓝牙模块的串口管脚相连接。

注意：因为蓝牙模块固件的更新，原有主从设置开关已失效，变为可恢复出厂设置的开关，如果想了解用法请参考 AT 指令集中的 PIO0 引脚的用法。为避免该引脚在实验中产生影响，请将所有主从开关拨为从。

4.2 4.2 软件接口

AT 命令参见文件《蓝牙模组 AT 指令集》。

操作方式：

替代串口线透明数据传输需要 2 个蓝牙模块，一个模块工作在主模式下，一个模块工作在从模式下。当两模块设置为相同的波特率。上电之后，主从模块则自动连接形成串口透传，此时的数据传输则是全双工的。

1）发送 AT 指令，设置主、从模块相同的波特率。

2）发送 AT 指令，设置主模块为主模式，从模块为从模式。

3）发送 AT 指令，设置主模块为搜索所有从设备。

4）模块上电，主模块则自动去查找该通道的从模块，此时主模块和从模块的 PIO1 脚都是输出为高低脉冲。若连接成功之后，主从模块的 PIO1 管脚输出为高电平，连接一个 BT LED 进行显示状态。

5）连接成功之后，两个模块两端就能进行串口数据全双工通信了。

4.3 代码分析

- **蓝牙主节点代码部分**

蓝牙模块控制程序 bluetooth.c 文件

```
void BT_Pin_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA,ENABLE);
    GPIO_InitStructure.GPIO_Mode=GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Pin=GPIO_Pin_1;
    GPIO_InitStructure.GPIO_Speed=GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
    GPIO_SetBits(GPIOA,GPIO_Pin_1);//蓝牙 RST 引脚高电平

    GPIO_InitStructure.GPIO_Mode=GPIO_Mode_IPD;
    GPIO_InitStructure.GPIO_Pin=GPIO_Pin_8;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
}
```

主节点程序 main.c 文件。

```
int main(void)
{
LED_GPIO_Init();//蓝色 LED 灯初始化
Key_Init();//按键引脚初始化
BT_Pin_Init();//蓝牙引脚初始化
NVIC_Configuration();//设置中断优先级
BT_BAUD_AUTO();//蓝牙串口自适应
USART1_Config(9600);//USART1 初始化
while(1)
{
    if(NumCount>=100)//计时 5S 时间到   100
    {
        NumCount=0;
        TIM_DeInit(TIM3);
        if(GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0)==0) //如果按键还按下,恢复出厂蓝牙设备
        {
            GPIO_SetBits(GPIOB, GPIO_Pin_11);
          BT_BAUD_AUTO();
            BT_RENEW();
        }else //如果按键已经抬起,恢复 USART1
        {
            GPIO_ResetBits(GPIOB, GPIO_Pin_11);
        }
    }
```

```
}
}
```

- **蓝牙从节点代码部分**

主函数部分 main.c。

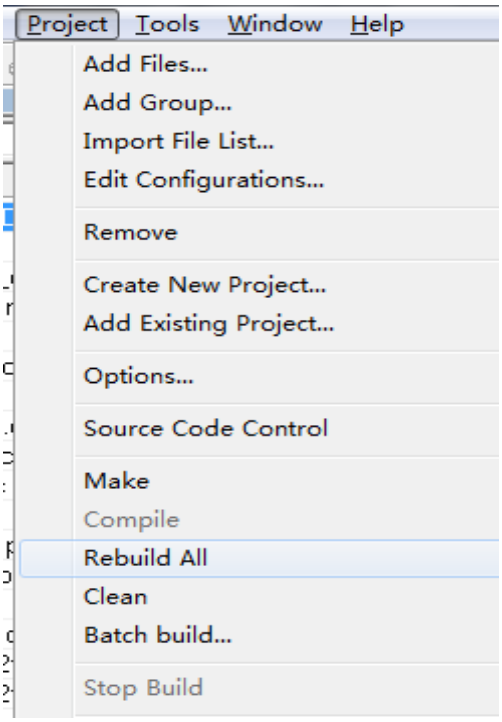
```
/**********************请更改以下内容 密码 名称**********************/
/**************密码由六位数字组成，用于多组实验避免冲突*************/
/****************名称建议与密码一致，方便连接时寻找设备*************/
uint8_t Pass[6] ="123456";
uint8_t Name[11]="123456";
/*********************************************************************/
int main(void)
{
LED_GPIO_Init();
Key_Init();
BT_Pin_Init();
NVIC_Configuration();
delay_nms(500);
BT_BAUD_AUTO();
USART1_Config(9600);
BT_NAME();//设置名称
BT_PASS();//设置密码
BT_PASSTYPE();//设置验证方式为密码验证
BT_PIO1();//设置蓝牙指示灯工作状态
BT_Y_RESET();//重启蓝牙设备
TIM4_Int_Init(499,7199);
while(1)
{
    if(NumCount>=100)//计时 5S 时间到   100*50ms
    {
        NumCount=0;
        TIM_DeInit(TIM3);
        if(GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0)==0) //如果按键还按下,恢复出厂蓝牙设备
        {
            GPIO_SetBits(GPIOB, GPIO_Pin_11);
          BT_BAUD_AUTO();
            BT_RENEW();//恢复出厂设置
        }else//如果按键已经抬起，点亮 LED 灯，退出循环
        {
            GPIO_ResetBits(GPIOB, GPIO_Pin_11);
        }
    }
    if(TIM4Count>=50)//计时 2.5S
    {
        TIM4Count=0;
```

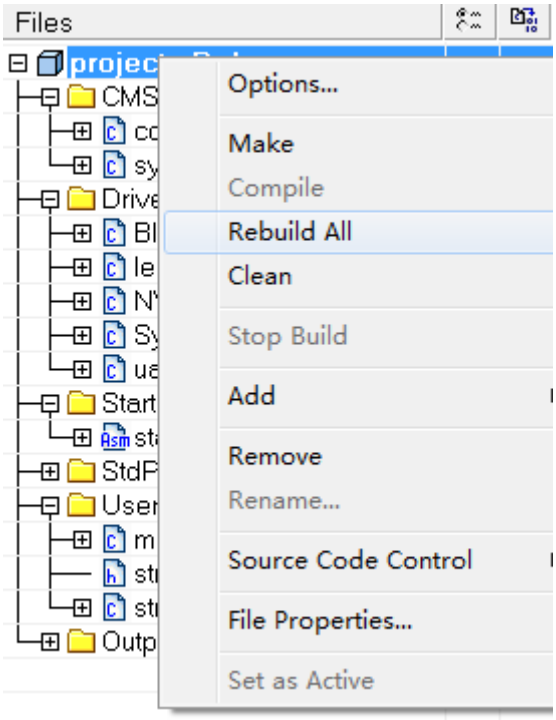
```
            if(GPIO_ReadInputDataBit(GPIOA,GPIO_Pin_8))//读取 BT 状态引脚
            {
                UART_PutStr(USART2,"Hello BlueTooth Msater I Am Slaver\n");//连接状态，发送信息
            }
    }
}
}
```

5. 实验步骤

1）用 J-OB 仿真器和 J-OB 转接板连接 PC 机与实验箱，用实验箱配套的电源给实验箱供电，并给模块上电。

2）用 IAR 软件打开实验工程，将实验工程进行编译，具体方法可以选择“Project”中的“Rebuild All”或者选中工程栏中的工程文件然后右键选择“Rebuild All ”进行编译。



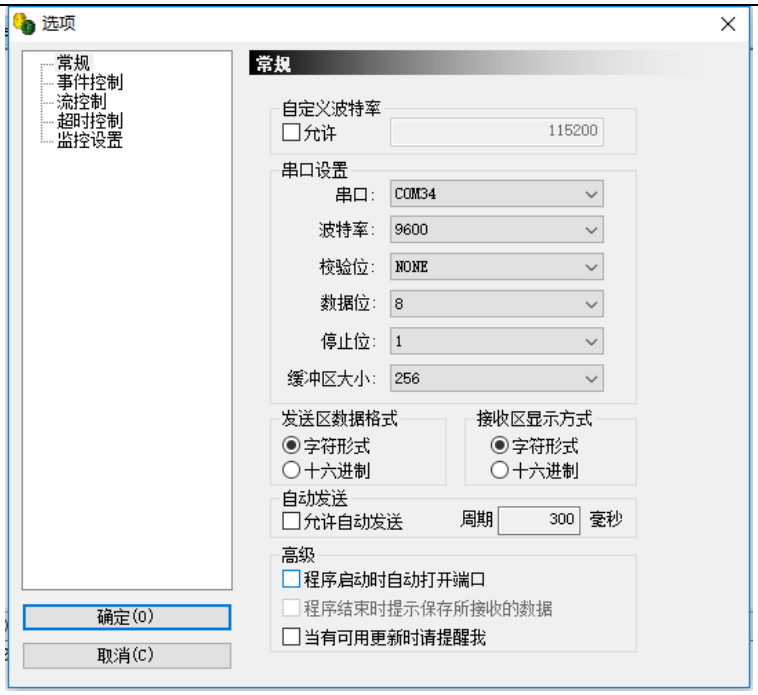


3）将其他蓝牙模块电源关闭，仅保留，一个主节点一个从节点保持开启。

4）用实验箱上的“+”、“-”按键分别选中主、从机模块，将 Master 和 Slaver 程序分别烧录到蓝牙主、从机模块里，并重启模块或者使用“RST”键复位模块。

注意：如需多组学生同时进行试验，请修改从机中 main.c 文件中的密码和名称，编译下载到从模块。主模块不需要单独设置。保证每组同学的密码唯一，便可以多组同学同时试验。

5）使用串口线连接主机与电脑串口，使用串口助手软件打开串口。



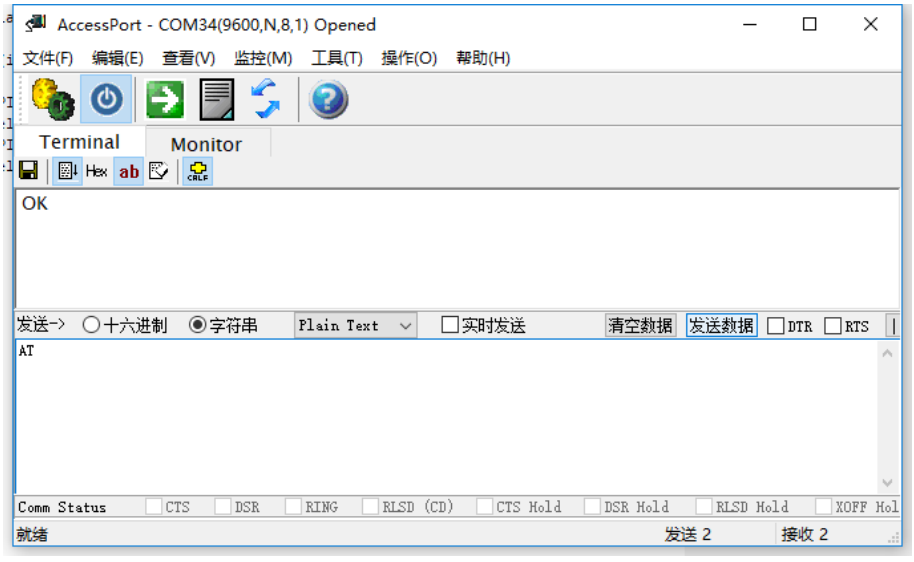
6）参数如图，对应串口号 波特率 9600 校验位 NONE 数据位 8 停止位 1。

7）如果程序下载完成两个蓝牙模块就已经连接，可以通过长按 KEY 按键 5S 以上进行对蓝牙模块的恢复出厂设置。

- 按下 KEY 按键开始闪烁，一直按按键到达 5S 时蓝色 LED 灯熄灭进入恢复出厂模式，如果 5S 内松开按键，不会执行恢复出厂设置。
- 恢复出厂设置的顺序为，先对主节回复出厂设置，再对从节点恢复出厂设置，然后复位两个 STM32 设备，使 STM32 进入正常程序，初始化蓝牙模块。
- 如果出现蓝色 LED 灯快闪几次然后慢闪几次则证明程序卡在了串口波特率的设置部分，此时应该关闭另一个蓝牙模块，并复位蓝牙设备重新等待初始化。
- 蓝色 LED 灯常亮是为程序正常运行状态。

8）在主节点串口中输入指定命令，操作主节点去连接从节点。

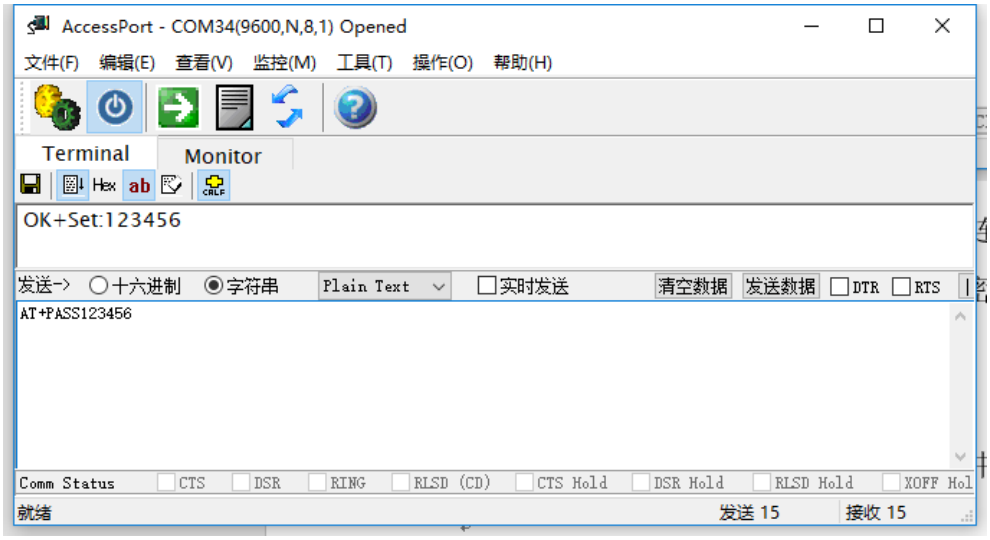
- 首先发送大写的 AT，测试是否可用 正常情况会收到 OK，如图。



如果不能收到请检查蓝牙模块是否已经连接，或复位设备，或尝试恢复出厂设置，然后对蓝牙模块主节点进行设置，设置密码，设置验证方式，设置为主节点模式。

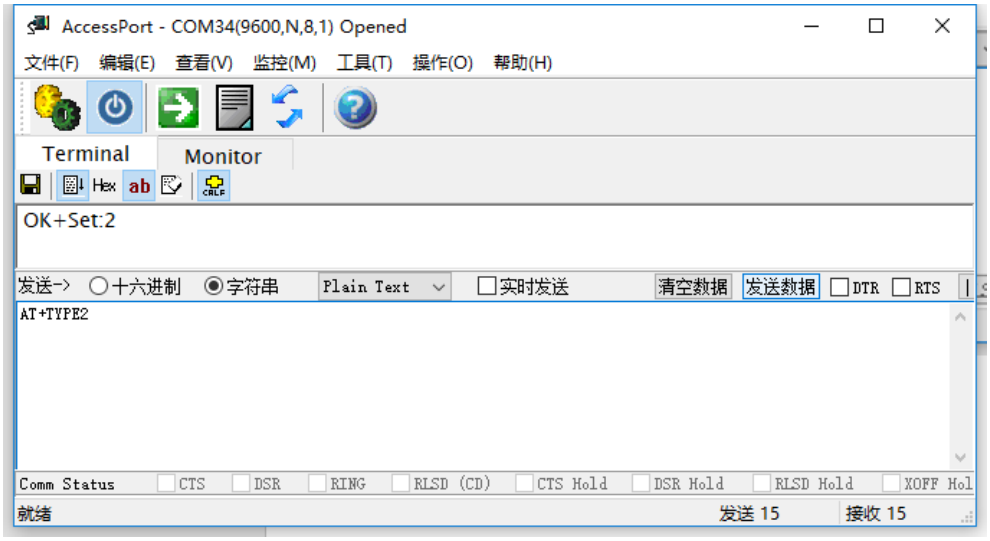
- 设置密码

发送 AT+PASS 密码（密码为从节点程序中的密码）会收到 OK+Set:密码。



- 设置验证方式

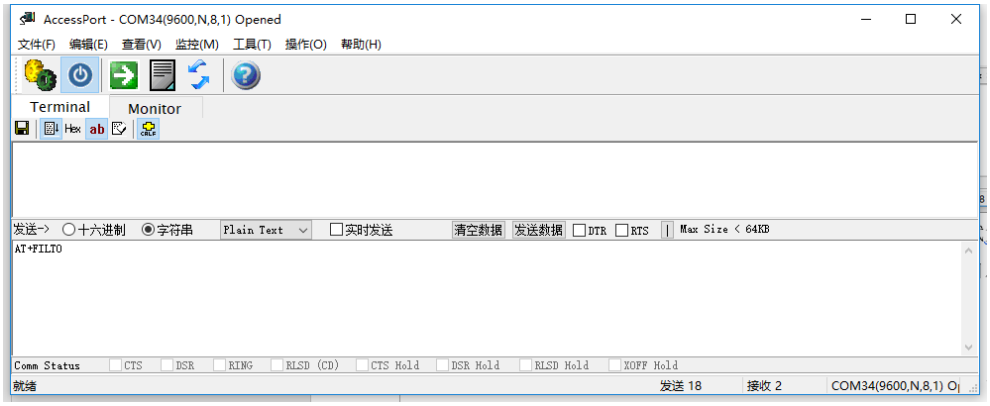
发送 AT+TYPE2，会收到 OK+Set:2，模式 2 为采用密码验证连接。



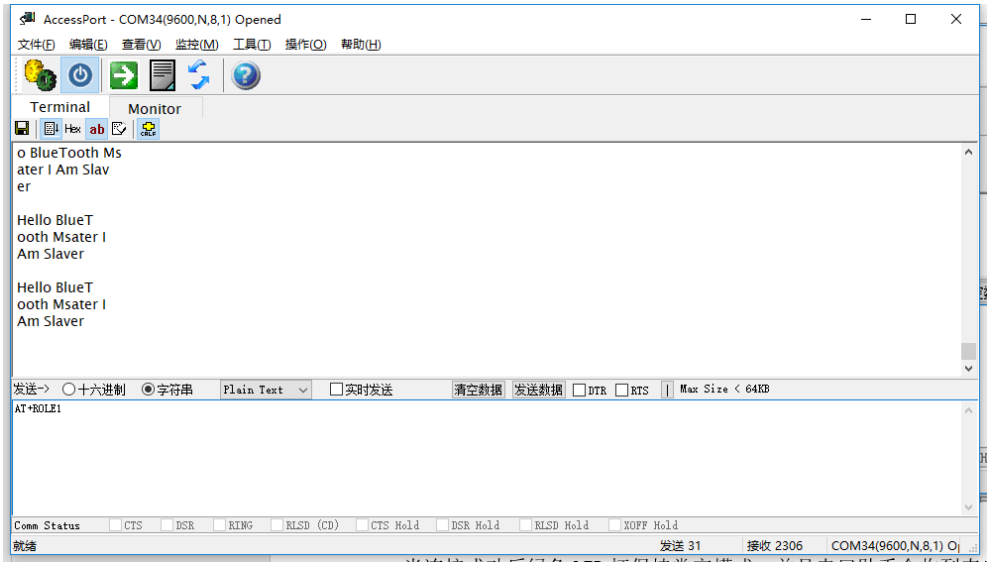
- 设置为主节点模式

发送 AT+ROLE1，会收到 OK+Set:1。

此时等待蓝牙模块自动搜索从机连接，如果长时间无法连接有可能是模块默认不自动搜索设备，需要更改，发送 AT+FILT0。



注意：并不一定会收到反馈，如果还是长时间不能连接成功，请尝试复位蓝牙模块，当连接成功后绿色 LED 灯保持常亮模式。并且串口助手会收到来自从机发送的数据。



实验二. BlueTooth 与手机连接实验

1. 实验目的

- 熟悉 IAR 开发软件的应用
- 熟悉蓝牙模块与手机的连接
- 掌握蓝牙之间组网原理

2. 实验设备

- 硬件:BlueTooth 模块（1 个），PC 机，J-OB 仿真器和 J-OB 转接板
- 软件:IAR 开发软件

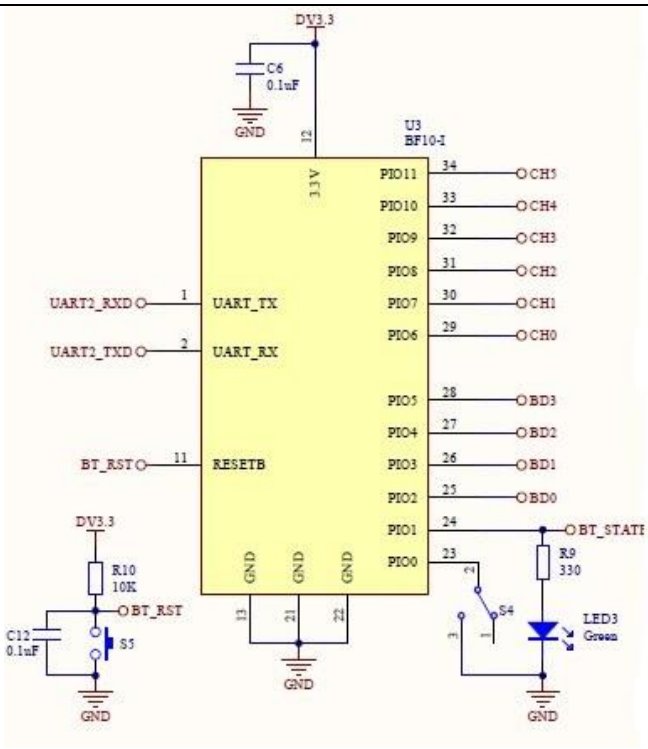
3. 实验内容

- 阅读 Bluetooth 硬件部分文档，熟悉 Bluetooth 相关硬件接口。
- 阅读 Bluetooth 硬件接口定义，熟悉 Bluetooth 模块的连接使用。
- 使用 IAR 开发环境设计程序，实现 Bluetooth 模块之间的点对点的组网配置。

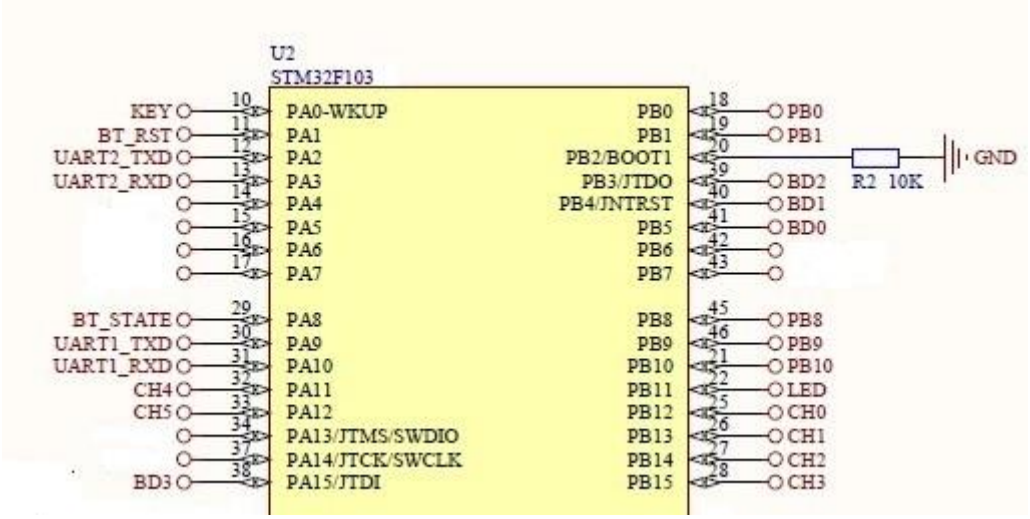
4. 实验原理

4.1 硬件接口原理

蓝牙模块与 STM32 连接原理图。



蓝牙模块原理图，其中 BF10-I 型模块与蓝牙 CC2540 模块硬件管脚兼容，此处原理图仍沿用 BF10-I 型。



该蓝牙模块已经内置应用程序，用户只需要通过蓝牙模块的串口管脚 UART_TX 和 UART_RX，发送 AT 指令即可完成对蓝牙模块的配置和控制。

上图为蓝牙模块和 STM32 连接的电路图，STM32 的 UART2 与蓝牙模块的串口管脚相连接。

注:因为蓝牙模块固件的更新，原有主从设置开关已失效，变为可恢复出厂设置的开关，如果想了解用法请参考 AT 指令集中的 PIO0 引脚的用法。为避免该引脚在实验中产生影响，请将所有主从开关拨为从。

4.2 软件接口

AT 命令参见文件《蓝牙模组 AT 指令集》。

操作方式：

替代串口线透明数据传输需要 2 个蓝牙模块，一个模块工作在主模式下，一个模块工作在从模式下。当两模块设置为相同的波特率。上电之后，主从模块则自动连接形成串口透传。此时的数据传输则是全双工的。

1）发送 AT 指令，设置主、从模块相同的波特率。

2）发送 AT 指令，设置主模块为主模式，从模块为从模式。

3）发送 AT 指令，设置主模块为搜索所有从设备。

4）模块上电，主模块则自动去查找该通道的从模块，此时主模块和从模块的 PIO1 脚都是输出为高低脉冲。若连接成功之后，主从模块的 PIO1 管脚输出为高电平，连接一个 BT LED 进行显示状态。

5）连接成功之后，两个模块两端就能进行串口数据全双工通信了。

4.3 代码分析

- **蓝牙从节点代码部分**

蓝牙模块控制程序 bluetooth.c 文件。

```
void BT_Pin_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA,ENABLE);
    GPIO_InitStructure.GPIO_Mode=GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Pin=GPIO_Pin_1;
    GPIO_InitStructure.GPIO_Speed=GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
    GPIO_SetBits(GPIOA,GPIO_Pin_1);//蓝牙 RST 引脚高电平

    GPIO_InitStructure.GPIO_Mode=GPIO_Mode_IPD;
    GPIO_InitStructure.GPIO_Pin=GPIO_Pin_8;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
}
```

从节点主函数　main.c 文件

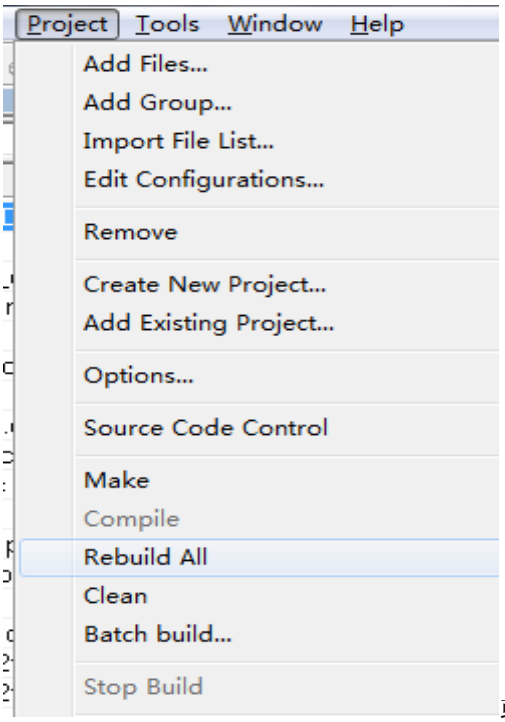
```
/**********************请更改以下内容 密码 名称**********************/
/**************密码由六位数字组成，用于多组实验避免冲突*************/
/***************名称建议与密码一致，方便连接时寻找设备*************/
/***********************本实验此处无需修改密码***********************/
//uint8_t Pass[6] ="123456";
uint8_t Name[11]="123456";
/*********************************************************************/
int main(void)
{
```

```
    LED_GPIO_Init();
    Key_Init();
    BT_Pin_Init();
    NVIC_Configuration();
    BT_BAUD_AUTO();
    USART1_Config(9600);
    delay_nms(500);
    BT_NAME();
    /*
    //此处注释密码操作，
    //因为在测试中发现由于现在手机的安卓版本与蓝牙版本较高，
    //与密码验证存在问题，故不采用密码验证，避免此问题
    BT_PASS();//设置密码
    BT_PASSTYPE();//设置验证方式为密码验证
     */
    BT_PIO1();//设置蓝牙指示灯工作状态
    BT_Y_RESET();//重启蓝牙设备
    TIM4_Int_Init(499,7199);
    while(1)
    {
        if(NumCount>=100)//计时 5S 时间到   100*50ms
        {
            NumCount=0;
            TIM_DeInit(TIM3);
            if(GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0)==0) //如果按键还按下,恢复出厂蓝牙设备
            {
                GPIO_SetBits(GPIOB, GPIO_Pin_11);
                                BT_BAUD_AUTO();
                BT_RENEW();//恢复出厂设置
            }else                                   //如果按键已经抬起，点亮 LED 灯，退出循环
            {
                GPIO_ResetBits(GPIOB, GPIO_Pin_11);
            }
        }
        if(TIM4Count>=50)//计时 2.5S
        {
            TIM4Count=0;
            if(GPIO_ReadInputDataBit(GPIOA,GPIO_Pin_8))//读取 BT 状态引脚
            {
                UART_PutStr(USART2,"Hello BlueTooth Phone I Am Slaver\n");//连接状态，发送信息
            }
        }
    }
}
```

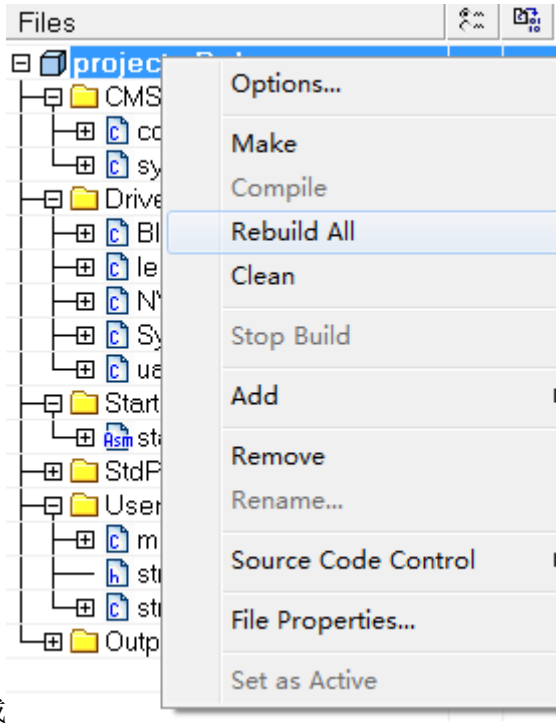
5. 实验步骤

1）用 J-OB 仿真器和 J-OB 转接板连接 PC 机与实验箱，用实验箱配套的电源给实验箱供电，并给模块上电。

2）用 IAR 软件打开实验工程，将实验工程进行编译，具体方法可以选择“Project”中的“Rebuild All”或者选中工程栏中的工程文件然后右键选择“Rebuild All ”进行编译。



或



3）将其他蓝牙模块电源关闭，仅保留一个从节点保持开启。

4）用实验箱上的“+”、“-”按键分别选中从机模块，将 Slaver 程序烧录到蓝牙从机模块里，并重启模块或者使用“RST”键复位模块。

注意：如需多组学生同时进行试验，请修改从机中 main.c 文件中的密码和名称，编译下载到从模块。主模块不需要单独设置。保证每组同学的密码唯一，便可以多组同学同时试验。

5）建议先将蓝牙恢复出厂设置在进行实验。可以通过长按 KEY 按键 5S 以上进行对蓝牙模块的恢复出厂设置。

- 按下 KEY 按键开始闪烁，一直按按键到达 5S 时蓝色 LED 灯熄灭进入恢复出厂模式，如果 5S 内松开按键，不会执行恢复出厂设置。
- 恢复出厂设置的顺序为，先对主节回复出厂设置，再对从节点恢复出厂设置，然后复位两个 STM32 设备，使 STM32 进入正常程序，初始化蓝牙模块。
- 如果出现蓝色 LED 灯快闪几次然后慢闪几次则证明程序卡在了串口波特率的设置部分，此时应该关闭另一个蓝牙模块，并复位蓝牙设备重新等待初始化。
- 蓝色 LED 灯常亮是为程序正常运行状态。

6）等待蓝色 LED 灯常亮，使用安卓手机安装对应软件。如果软件提示需要获取权限请

允许。

打开手机蓝牙，打开软件，点击搜索设备，找到名称为程序中设置的设备。单击即可连接。

连接成功后会接受到从机发送的数据。

实验三. BlueTooth 传感网实验

1. 实验目的

- 熟悉 IAR 开发软件的应用。
- 掌握蓝牙之间组网原理。

2. 实验设备

- 硬件:BlueTooth 模块（2 个），PC 机，J-OB 仿真器和 J-OB 转接板，传感器模块。
- 软件:IAR 开发软件，串口助手。

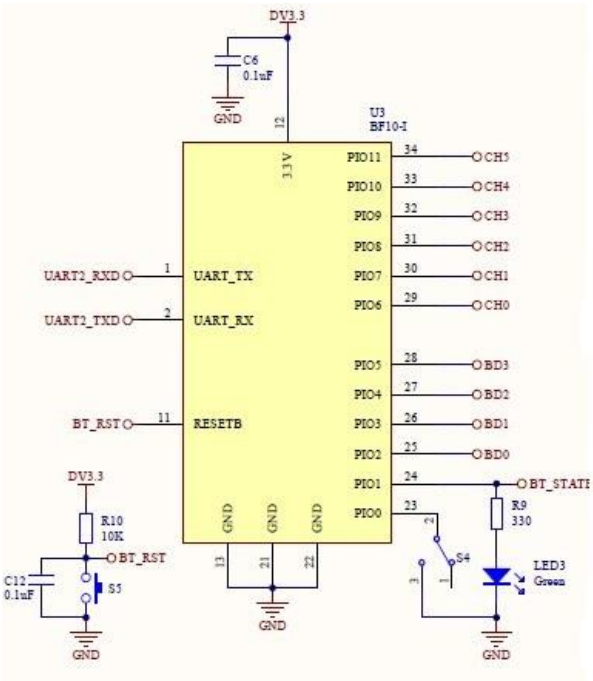
3. 实验内容

- 阅读 Bluetooth 硬件部分文档，熟悉 Bluetooth 相关硬件接口。
- 阅读 Bluetooth 硬件接口定义，熟悉 Bluetooth 模块的连接使用。
- 使用 IAR 开发环境设计程序，实现 Bluetooth 模块之间的点对点的组网配置。

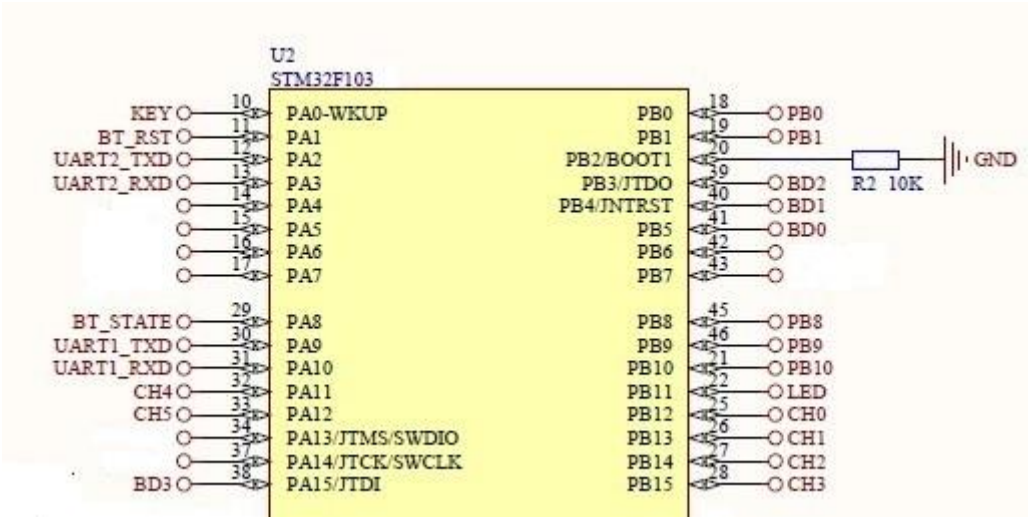
4. 实验原理

4.1 硬件接口原理

蓝牙模块与 STM32 连接原理图。



蓝牙模块原理图，其中 BF10-I 型模块与蓝牙 CC2540 模块硬件管脚兼容，此处原理图仍沿用 BF10-I 型。



该蓝牙模块已经内置应用程序，用户只需要通过蓝牙模块的串口管脚 UART_TX 和 UART_RX，发送 AT 指令即可完成对蓝牙模块的配置和控制。

上图为蓝牙模块和 STM32 连接的电路图，STM32 的 UART2 与蓝牙模块的串口管脚相连接。

注意：因为蓝牙模块固件的更新，原有主从设置开关已失效，变为可恢复出厂设置的开关，如果想了解用法请参考 AT 指令集中的 PIO0 引脚的用法。为避免该引脚在实验中产生影响，请将所有主从开关拨为从

4.2 软件接口

AT 命令参见文件《蓝牙模组 AT 指令集》。

操作方式：

替代串口线透明数据传输需要 2 个蓝牙模块，一个模块工作在主模式下，一个模块工作在从模式下。当两模块设置为相同的波特率。上电之后，主从模块则自动连接形成串口透传。此时的数据传输则是全双工的。

1）发送 AT 指令，设置主、从模块相同的波特率。

2）发送 AT 指令，设置主模块为主模式，从模块为从模式。

3）发送 AT 指令，设置主模块为搜索所有从设备。

4）模块上电，主模块则自动去查找该通道的从模块，此时主模块和从模块的 PIO1 脚都是输出为高低脉冲。若连接成功之后，主从模块的 PIO1 管脚输出为高电平，连接一个 BT LED 进行显示状态。

5）连接成功之后，两个模块两端就能进行串口数据全双工通信了。

4.3 代码分析

- **蓝牙从节点代码部分。**

蓝牙模块控制程序 bluetooth.c 文件。

```
void BT_Pin_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA,ENABLE);
    GPIO_InitStructure.GPIO_Mode=GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Pin=GPIO_Pin_1;
    GPIO_InitStructure.GPIO_Speed=GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
    GPIO_SetBits(GPIOA,GPIO_Pin_1);//蓝牙 RST 引脚高电平

    GPIO_InitStructure.GPIO_Mode=GPIO_Mode_IPD;
    GPIO_InitStructure.GPIO_Pin=GPIO_Pin_8;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
}
```

从节点主函数 main.c 文件

```
while(1)
    {
        if(NumCount>=100)//计时 5S 时间到   100*50ms
        {
            NumCount=0;
            TIM_DeInit(TIM3);
            if(GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0)==0) //如果按键还按下,恢复出厂蓝牙设备
            {
                                    USART_DeInit(USART1);
                GPIO_SetBits(GPIOB, GPIO_Pin_11);
                                    BT_BAUD_AUTO();
                BT_RENEW();//恢复出厂设置
            }else                                                    //如果按键已经抬起，点亮 LED 灯，
退出循环
            {
                GPIO_ResetBits(GPIOB, GPIO_Pin_11);
            }
        }
                if(USART1_Flag==1&&BT_State)//接收到数据
                {
                  if(Uart1_Recv[0]==0xEE&&Uart1_Recv[1]==0xCC&&Uart1_Recv[13]==0xFF)
                  {
                    TIM4_Int_Init(499,7199);
                    memset(Uart_Send,0,26);
```

```
                    for(i=0;i<14;i++)
                      Uart_Send[i]=Send_Start[i];
                   for(i=14;i<23;i++)
                      Uart_Send[i]=Uart1_Recv[(i-12)];
                    for(i=23;i<26;i++)
                      Uart_Send[i]=Send_End[(i-23)];
                    UART_SendString(USART2,Uart_Send,26);
                    USART1_Flag=0;
                    GPIO_ResetBits(GPIOB, GPIO_Pin_11);
                    memset(Uart1_Recv,0,100);
                  }else
                  {
                    USART1_Flag=0;
                    memset(Uart1_Recv,0,100);
                    memset(Uart_Send,0,26);
                  }
                }
                if(USART2_Flag==1)//接收到数据
                {

                  if(Uart2_Recv[0]==0xEE&&Uart2_Recv[1]==0xCC&&Uart2_Recv[25]==0xFF)
                  {
                    memset(Uart_Send,0,26);
                    Uart_Send[0]=0xEE;
                    Uart_Send[1]=0xCC;
                    for(i=2;i<14;i++)
                    {
                      Uart_Send[i]=Uart2_Recv[(i+12)];
                    }
                    UART_SendString(USART1,Uart_Send,14);
                    USART2_Flag=0;
                    memset(Uart2_Recv,0,100);
                  }else
                  {
                    USART2_Flag=0;
                    memset(Uart2_Recv,0,100);
                    memset(Uart_Send,0,26);
                  }
                }
    }
}
```

- **主节点程序**

主程序 main.c 文件

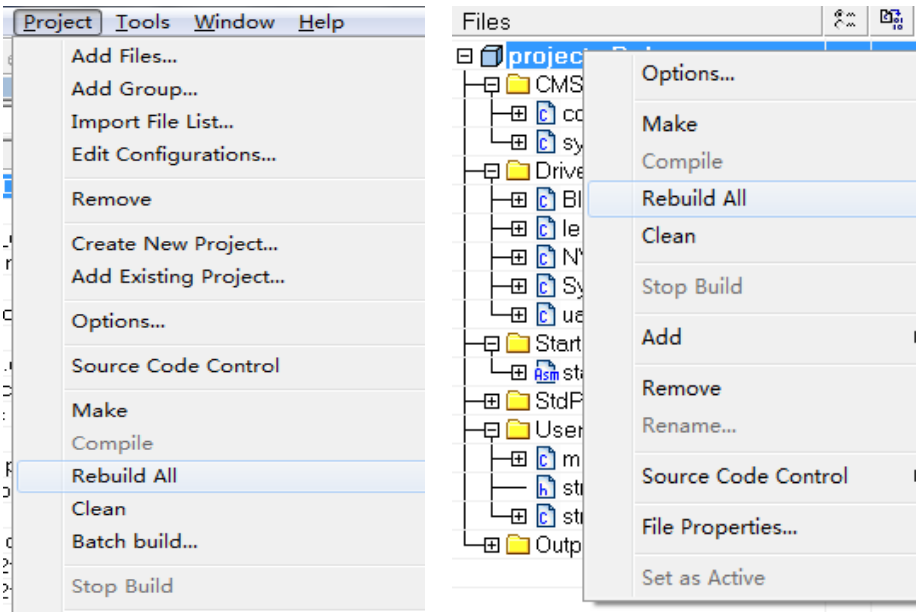
```
while(1)
```

```
{
    if(NumCount>=100)//计时 5S 时间到   100
    {
        NumCount=0;
        TIM_DeInit(TIM3);
        if(GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0)==0) //如果按键还按下,恢复出厂蓝牙设备
        {
          USART_DeInit(USART1);
            GPIO_SetBits(GPIOB, GPIO_Pin_11);
          BT_BAUD_AUTO();
            BT_RENEW();
        }else      //如果按键已经抬起,恢复 USART1
        {
            GPIO_ResetBits(GPIOB, GPIO_Pin_11);
        }
    }
                if(USART1_Flag==1)
                {
                  if(Uart1_Recv[0]==0xEE&&Uart1_Recv[1]==0xCC)
                    UART_SendString (USART2,Uart1_Recv,26);
                  else
                  UART_PutStr(USART2,Uart1_Recv);
                  USART1_Flag=0;
                  memset(Uart1_Recv,0,100);
                }
                if(USART2_Flag==1)
                {
                  if(Uart2_Recv[0]==0xEE&&Uart2_Recv[1]==0xCC)
                    UART_SendString (USART1,Uart2_Recv,26);
                  else
                    UART_PutStr(USART1,Uart2_Recv);
                  USART2_Flag=0;
                   memset(Uart2_Recv,0,100);
                 }
    }
```

5. 实验步骤

1）用 J-OB 仿真器和 J-OB 转接板连接 PC 机与实验箱，用实验箱配套的电源给实验箱供电，并给模块上电。

2）用 IAR 软件打开实验工程，将实验工程进行编译，具体方法可以选择“Project”中的“Rebuild All”或者选中工程栏中的工程文件然后右键选择“Rebuild All ”进行编译。

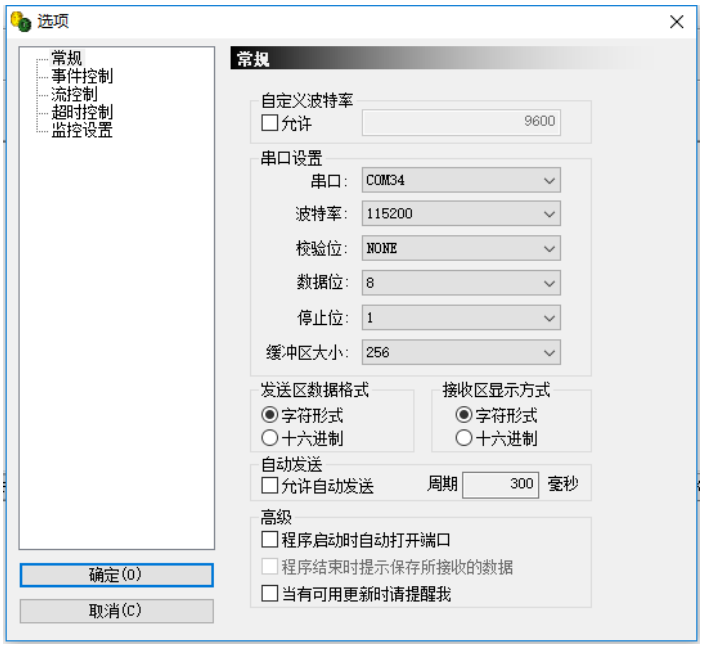


2）将其他蓝牙模块电源关闭，仅保留，一个主节点一个从节点保持开启。

用实验箱上的“+”、“-”按键分别选中主、从机模块，将 Master 和 Slaver 程序分别烧录到蓝牙主、从机模块里，并重启模块或者使用“RST”键复位模块。

注意：如需多组学生同时进行试验，请修改从机中 main.c 文件中的密码和名称，编译下载到从模块。主模块不需要单独设置。保证每组同学的密码唯一，便可以多组同学同时试验。

3）使用串口线连接主机与电脑串口，使用串口助手软件打开串口。



参数如图，对应串口号、波特率 115200、校验位 NONE、数据位 8、停止位 1

4）如果程序下载完成两个蓝牙模块就已经连接，可以通过长按 KEY 按键 5S 以上进行对蓝牙模块的恢复出厂设置。

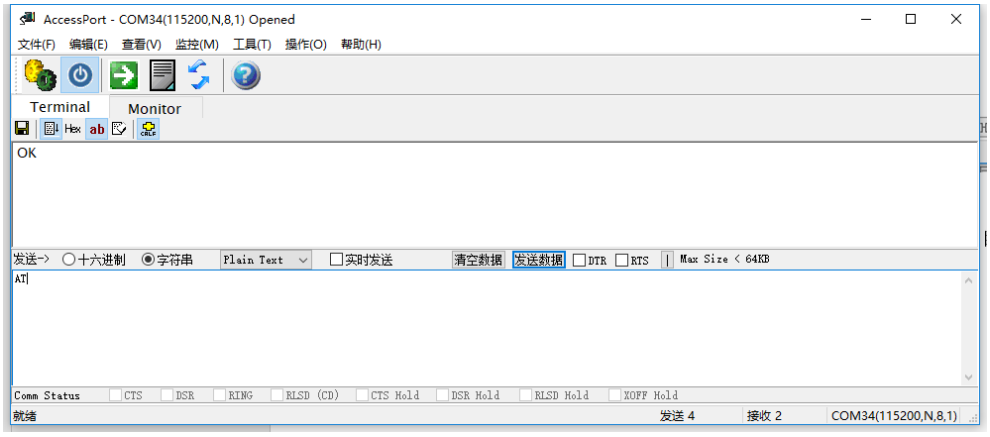
- 按下 KEY 按键开始闪烁，一直按按键到达 5S 时蓝色 LED 灯熄灭进入恢复出厂模式，如果 5S 内松开按键，不会执行恢复出厂设置。

- 恢复出厂设置的顺序为，先对主节回复出厂设置，再对从节点恢复出厂设置，然后复位两个 STM32 设备，使 STM32 进入正常程序，初始化蓝牙模块。
- 如果出现蓝色 LED 灯快闪几次然后慢闪几次则证明程序卡在了串口波特率的设置部分，此时应该关闭另一个蓝牙模块，并复位蓝牙设备重新等待初始化。
- 蓝色 LED 灯常亮是为程序正常运行状态。

5）将传感器模块插在蓝牙从节点上，因为传感器有两种，只上报的传感器和可控制的传感器，本实验文档采用声光报警传感器，不仅可以上报还可以控制。

6）在主节点串口中输入指定命令，操作主节点去连接从节点。

- 首先发送大写的 AT，测试是否可用 正常情况会收到 OK，如图

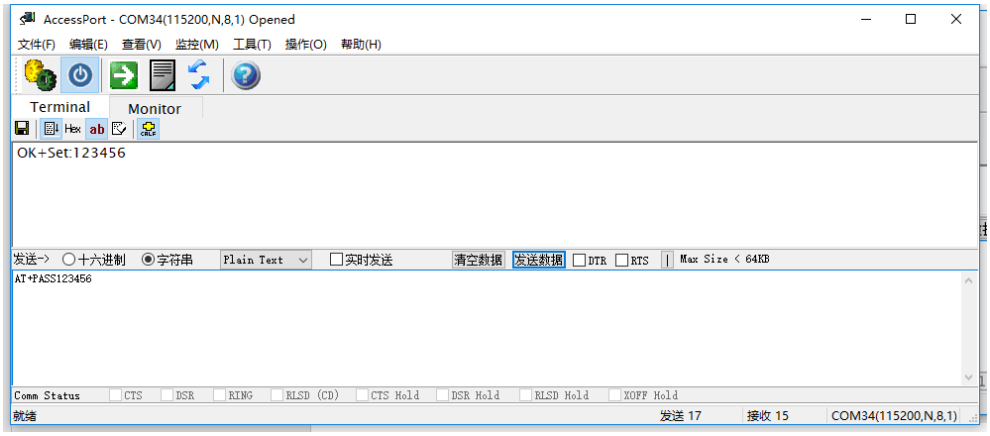


如果不能收到请检查蓝牙模块是否已经连接，或复位设备，或尝试恢复出厂设置

然后对蓝牙模块主节点进行设置，设置密码，设置验证方式，设置为主节点模式

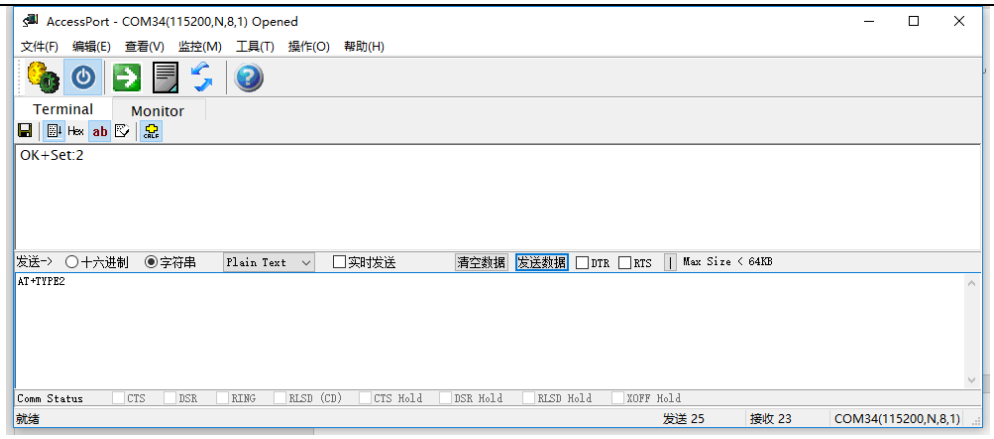
- 设置密码

发送 AT+PASS 密码（密码为从节点程序中的密码）会收到 OK+Set:密码



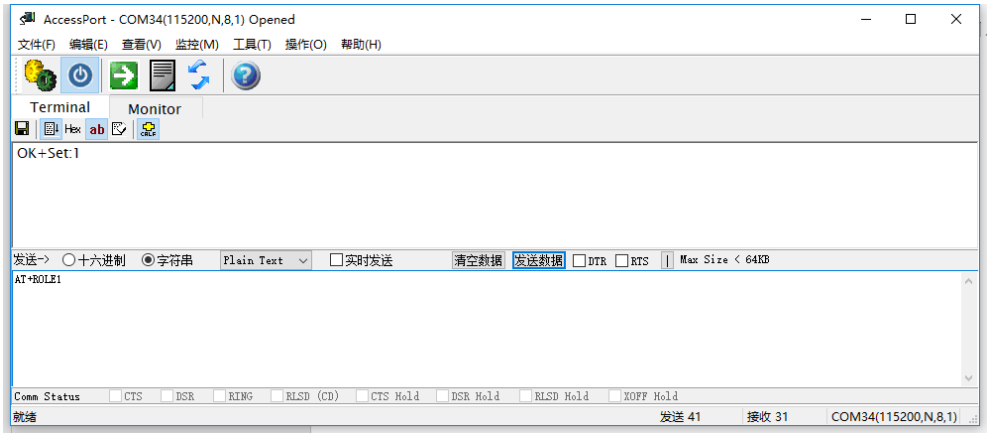
- 设置验证方式。

发送 AT+TYPE2，会收到 OK+Set:2，模式 2 为采用密码验证连接。

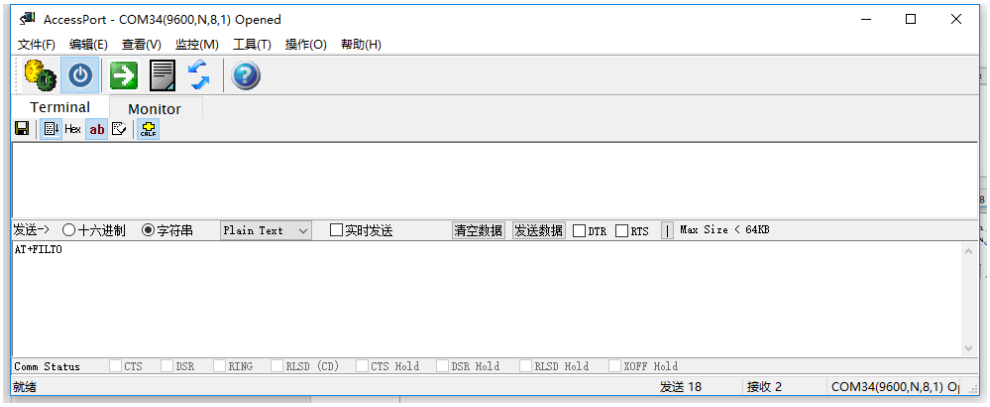


◆ 设置为主节点模式

发送 AT+ROLE1，会收到 OK+Set:1。

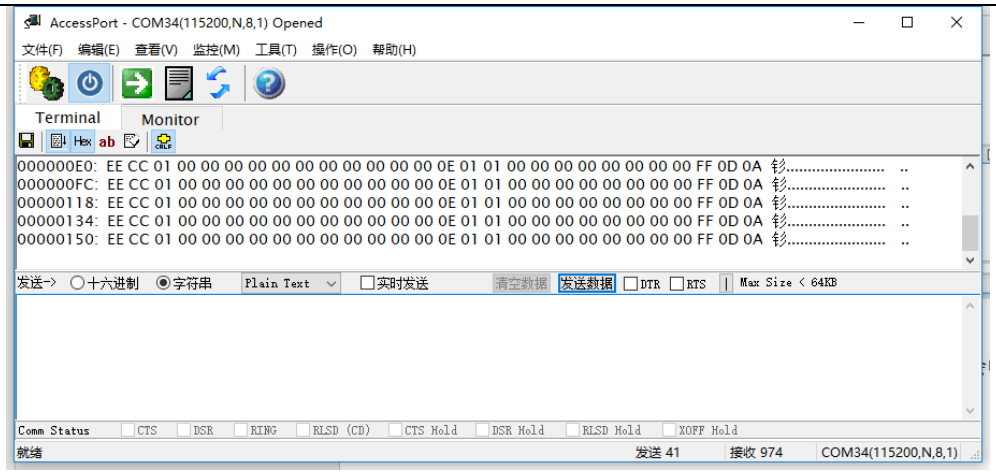


此时等待蓝牙模块自动搜索从机连接，如果长时间无法连接有可能是模块默认不自动搜索设备，需要更改，发送 AT+FILT0。

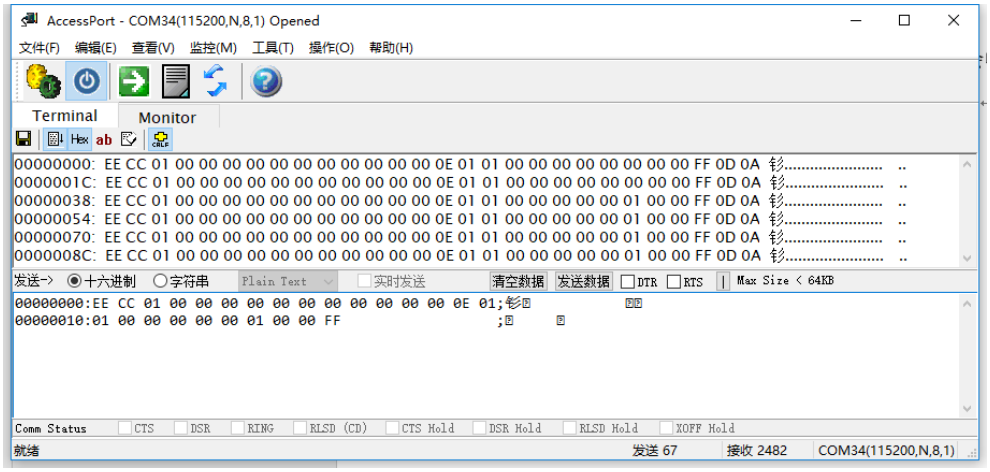


注意：并不一定会收到反馈，如果还是长时间不能连接成功，请尝试复位蓝牙模块，当连接成功后绿色 LED 灯保持常亮模式。

7）并且串口助手会收到来自从机发送的数据，此时将串口助手切换为 16 进制显示便可以接收到传感器数据，并且从机的蓝色 LED 灯快闪两下表示发送数据。



可以发送 16 进制命令控制声光报警器发声和闪烁：EE CC 01 00 00 00 00 00 00 00 00 00 00 0E 01 01 00 00 00 00 00 01 00 00 FF



第五章. 无线通讯模块之 WiFi 通信实验

本章主要介绍无线通讯模块部分的 WiFi 通信的实验内容，采用 ICS-IOT-CEP 平台配套的 WiFi 模块。内容由浅入深，前面已经讲述相应模块的硬件接口实验，本章主要针对 WIFI 模块的配置组网实验及传感器网络实验等。通过本章实验内容，读者即可以迅速掌握基上述 WiFi 无线模块的开发方法，以及相应网络结构的传感器数据通讯应用设计。

实验一. WiFi 模块组网配置实验

1. 实验目的

- 了解如何使用网络的方法设置 WiFi 模块的工作模式及参数。
- 了解如何将 WiFi 模块进行组网。

2. 实验设备

- ICS-IOT-CEP 教学实验平台 WiFi 模块两个，带有无线网卡的 PC 机一台。

3. 实验内容

- 用 PC 机使用无线网络的方法对 WiFi 模块进行模式和工作参数设置，使两个模块进行组网。
- 将 WiFi 模块进行设置并对设置是否成功进行观察。

4. 实验说明

ICS-IOT-CEP 教学实验平台中的 WiFi 模块使用的是 HF-A11x 模块，模块默认为 AP 接口。用户可以通过 PC 机的连接 HF-A11x 的 AP 接口，并用 web 管理页面配置。

默认情况下，HF-A11x 的 AP 接口 SSID 为 HF-A11x_AP,IP 地址和用户名，密码如下：

参数	默认设置
SSID	HF-A11x_AP
IP 地址	10.10.100.254
子网掩码	255.255.255.0
用户名	admin
密码	admin

表 4.1 HF-A11x 网络默认设置表

5. 实验步骤

1）用配套线将实验设备和电源相连，并给设备供电；打开 WiFi 模块上的电源开关，给模块供电。

2）使用 WiFi 节点部分模块上的 ReLoad 按键将模块恢复默认设置。方法是模块上的 Ready 指示灯亮后按下 ReLoad 按键一段时间(约五秒钟)，然后松手，再按下该键一段时间至 Ready 指示灯熄灭，此时模块已经恢复默认设置。

3）等 WiFi 的启动指示灯亮后，用 PC 机的无线网连接 HF-A11x_AP。等连接后，打开

IE，在地址栏中输入 http：//10.10.100.254，回车。在弹出弹出来的对话框中输入用户和密码，然后“确认”。

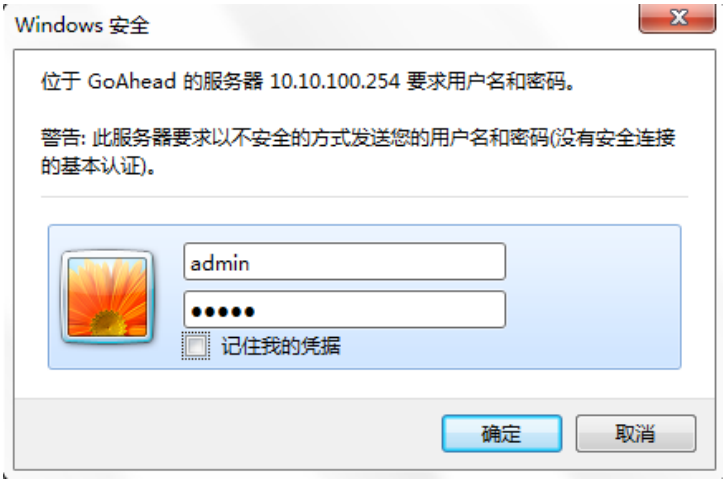


图 5.4.1 登陆网页界面

然后网页上会出现 HF-A11 的管理界面，页面支持中英文刻在网页的右上角进行选择。网页管理有五个页面，分别为“模式选择”、“无线接入点设置”、“无线终端接口设置”、“应用程序设置”、“模块管理”。

4）无线模式选择

网页的第一页可以设置模式工作在 AP 模式或是 STA 模式。需要工作在 AP 模式的模块在这一步选择 AP 模式，需要工作在 STA 模式的模块可以选择 Station 模式，本实验使用两个模块，一个模块按照步骤 5 设置，另一模块按照步骤 6 设置然后确定。

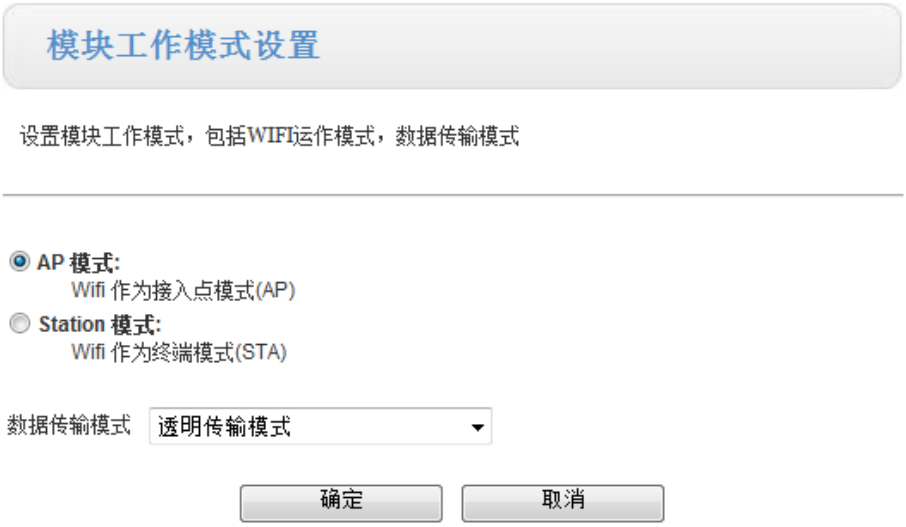


图 5.4.2 无线模式选择

5）无线接入点设置（一个模块设置为 AP 模式）

HF-A11x 支持 AP 接口，通过这个接口可以十分方便的对模块进行管理，而且可以实现自组网，管理页面如下图。包括“无线网络”、“安全模式”、“局域网设置”部分。需要说明的是设置成 STA 模式的模块不需要进行这一步设置，直接进行步骤 6）。

- **无线网络设置和安全模式**

在无线网络设置中，网络模式选择“11b/g/n mixed mode”，网络名称选择默认即可，广播网络名称选择“启用”，AP Isolation 和 MBSSID AP Isolation 都选用“停用”选项，频率（频道）设置为“自动选择”。然后点击“确定”。再进入该页面进行安全模式设置，安全模式选择“自动选择”，后点击“确定”。

图 5.5.1 无线网络和安全模式

♦ **局域网设置**

在这一步，IP 地址设置成“192.168.1.1”，子网掩码设置成“255.255.255.0”，MAC 地址值为默认值，DHCP 类型选择为“服务器”，DHCP 网关设置改为 192.168.1.1，后点击“确定”。

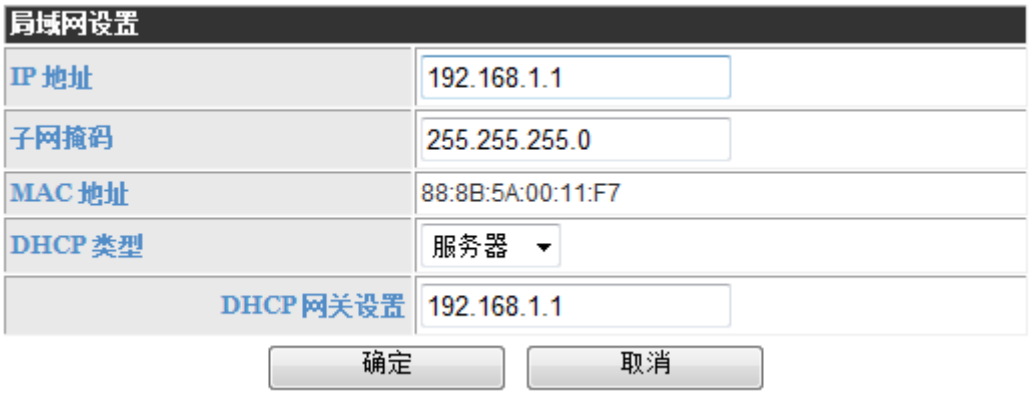


图 5.5.3 局域网设置

6）无线终端接口设置（设置另外一个模块设置为 STA 模式）

无线终端接口，即 STA 接口。HF-A11x 可以通过 STA 接口接入到其他无线网络中，设置页面分为无线终端接口参数和 DHCP 两部分。广域网联机模式选择动态，这时将 DHCP 服务器地址栏中填入 192.168.1.1，然后确定。广域网联机模式设置为“静态（固定 IP）”时，点击后出现三个列表格，IP 地址设置为“192.168.1.234”、子网掩码设置为

“255.255.255.0”、网关设置为“192.168.1.1”，更改完成后点击“确定”。（设置为静态的目的是便于检验设置是否成功）需要注意的是在进行设置时，设置为 AP 模式的模块不需要进行无线终端接口设置这一步。

图 5.6 无线终端和 DHCP 模式

7）串口及其他设置

应用程序设置是对 WiFi 转 UART 应用参数的设置包括：串口参数设置及网络协议的设置。

- **串口设置**

串口的设置为：波特率 115200，数据位 8，校验位 None，停止位 1，CTSRTS 设置为 Disable。然后点击“确定”。再进入此页面进行下一步设置。

图 5.7.1 串口设置

- **UART 自动生成帧设置**

UART 自动生成帧设置表中，需将 UART 自动成帧选择为“Enable”，会出现自动成帧时间和自动成帧字节数两项，这两列分别设成 500 和 46，如下图。设置完成后，点击“确定”，再进入页面，进行下一步的设置。

图 5.7.2 UART 自动成帧设置

- **网络设置**

在这一项的设置中，网络模式选择 Client，协议为 UDP，端口设置成 45000，服务器的地址设置为 192.168.1.120。设置完成后，点击“确定”，再进入模块管理进行设置。

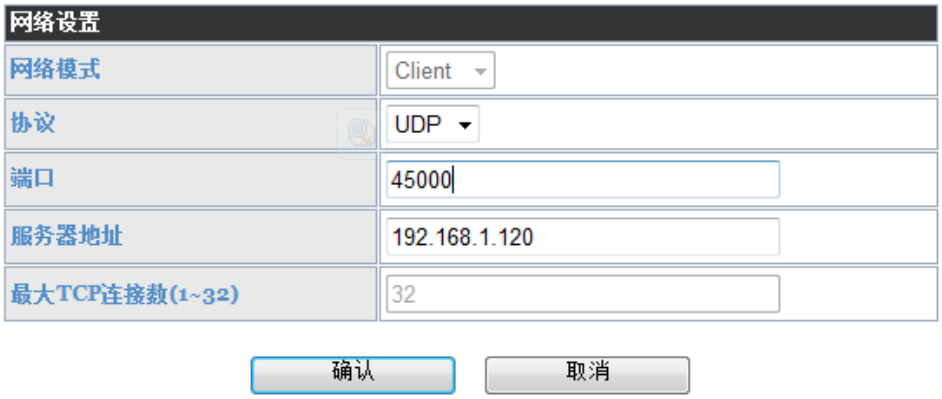


图 5.7.3 网络设置

注意：先只更改网络模式为 Client 后点击“确认”按钮，之后返回该界面重新设置其他参数。（若一起设置后点“确认”网络模式会仍然为 Server）

8）模块管理

模块管理包括用户名/密码设置、重启模块、恢复出厂设置及软件升级功能。在这一步中，只需进行重启模块中的“重启”即可，重启模块后，对模块的各项设置会生效。

图 5.8 模块管理

9）结果检验

在 PC 机的开始栏中或 win+R 键后，输入 cmd 后回车，然后输入“ping 192.168.1.1”然后回车，观察实验现象。若结果显示为有来自“192.168.1.1”的回复，则 AP 模块配置设置成功。

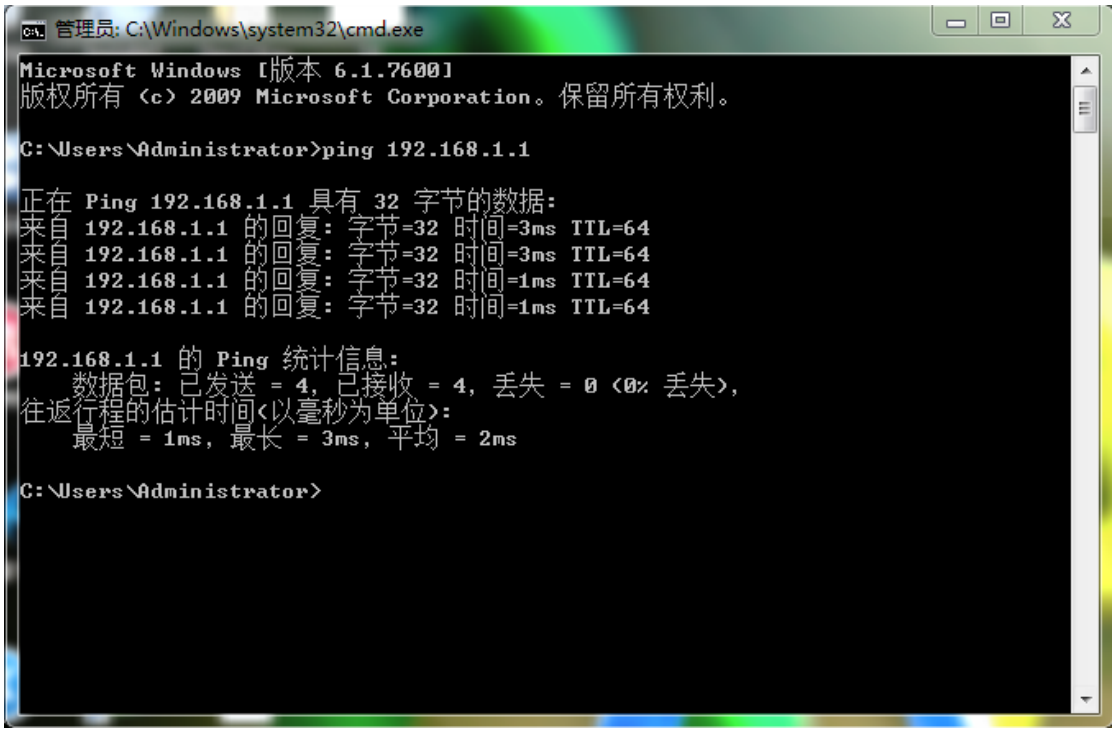


图 5.9.1 AP 模块设置检验结果图

若输入“ping 192.168.1.234”，结果显示有来自“192.168.1.234”的回复则说明 STA 模块设置成功。具体如下图。

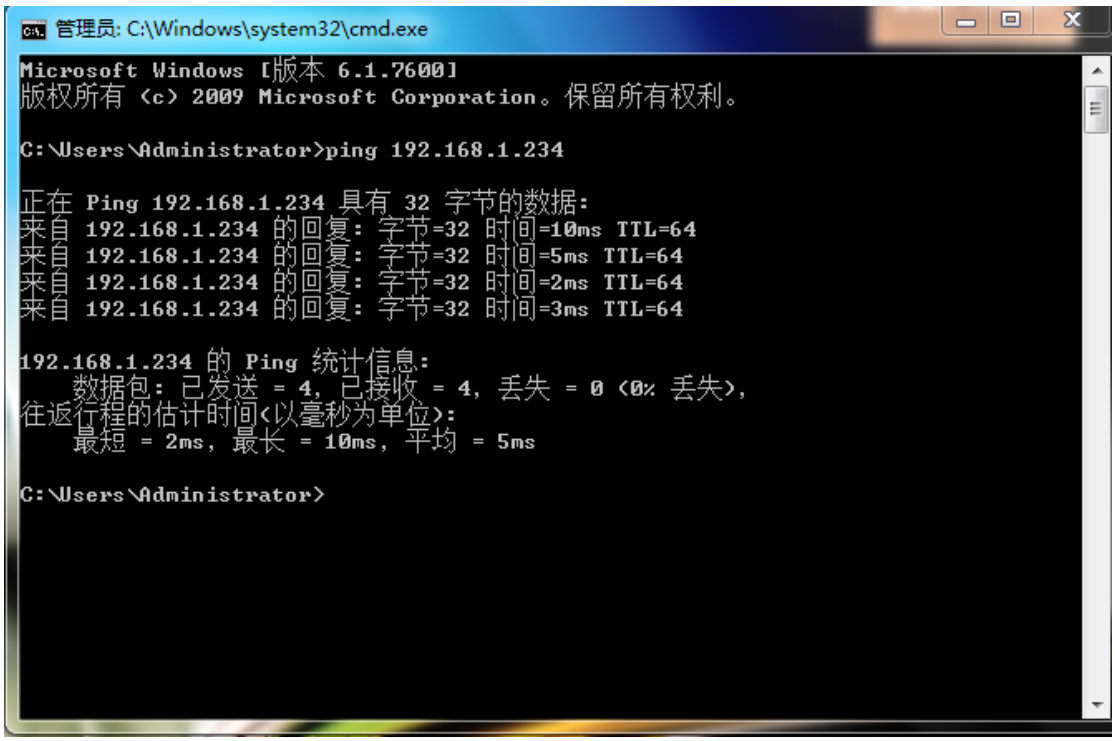


图 5.9.2 STA 模块设置结果检验图

6. 注意事项

在实验步骤中涉及到有关 SSID 和服务器的的说明：

在实验步骤 5）中，设置 AP 节点的网络名称和 STA 节点的 SSID 须一致，否则会影响实验结果。

在实验步骤 7）中，“网络设置”部分，服务器地址的设置与否不会影响本实验的的结果，但在“基于 WiFi 的网络传感网实验”会影响实验结果，至于如何设置，在“基于 WiFi 的网络传感网实验”中会有相关说明。

实验二. 基于 WiFi 网络传感网实验

1. 实验目的

- 了解传感器的使用。
- 了解 WiFi 的组网原理。
- 了解 WiFi 的在组网中的串口通信。
- 掌握 STM32 串口的使用。

2. 实验设备

- 硬件：ICS-IOT-CEP 教学实验平台，PC 机，J-OB 仿真器和 J-OB 转接板。
- 软件：串口调试软件，IAR 开发软件。

3. 实验内容

- 阅读 WiFi 部分文档，熟悉 WiFi 模块相关接口定义。
- 阅读 WiFi 部分文档，熟悉 WiFi 组网的基本原理。
- 通过程序控制实现 STA 节点上传感器采集到的的数据通过 WiFi 发送到 AP 节点上，并从 AP 节点模块上的 STM32 的串口 1 中打印出来。

4. 实验原理

4.1 硬件结构原理

- **WiFi 模块电路图**

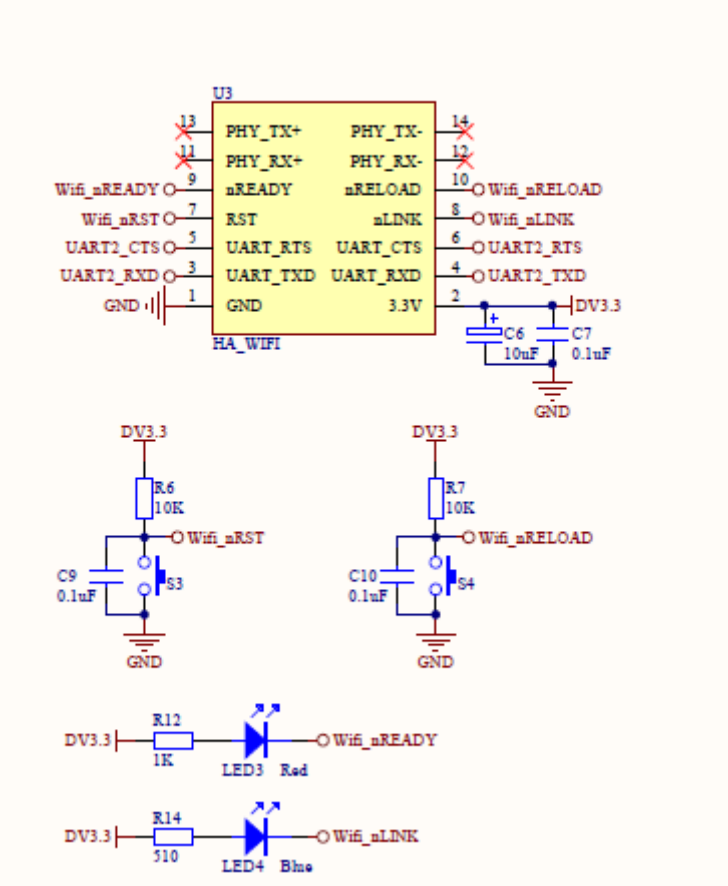


图 4.1.1WiFi 模块电路图

“Wifi_nRST”为 WiFi 模块复位引脚，低电平复位，不按下 S3 时该引脚为高电平，按下 S3 时该引脚为低电平，复位时间需大于 300ms，按下 S3 能实现复位 WiFi 模块；S4 为恢复出厂设置按键，按下 S4 持续五秒钟，然后松手，再按下 S4 直到“Ready”指示灯由亮变灭，WiFi 模块完成恢复出厂设置。LED3 为模块启动状态指示灯，模块启动完毕后“Wifi_nREADY”引脚输出 0，否则输出 1，当该引脚输出低电平时 LED3 为“亮”，反之，LED3 为“灭”；LED4 为 WiFi 模块连接状态指示灯，WiFi 有连接时“Wifi_nLink”引脚输出 0，LED4 亮，反之输出 1，LED4 灭。UART_TXD、UART_RXD、UART_CTS、UART_RTS 为串口通信接口，带硬件流控制，与 STM32 的 UART2 连接。

4.2 WiFi 传输模式

- **透明传输模式**

HF-A11x 支持串口透明传输模式，可以实现串口即插即用，从而最大程度的降低用户使用的复杂度。在此模式下，所有需要手法的数据都被在串口与 WiFi 接口之间做透明传输，不做任何解析。在透明传输模式下，可以兼容用户原有的软件平台。用户设备基本不用做软件的改动就可以实现支持无线数据传输。

- **协议传输模式**

如果用户的数据要求 100%精确，或者用户的上位机（MCU）处理速度太低，可以采用协议传输模式保证 UART 数据的无误码传输。

协议传输模式主要保证 UART 接口上数据的准确性,在这种传输模式下，定义了串口线上传输数据结构，校验方式及两边设备握手方式。

在协议传输模式下，用户可以发送数据给命令给 HF-A11x 模块，模块发到数据后回确认命令。HF-A11x 模块不会主动把数据发给用户设备，只有当用户设备向模块发送命令要求数据时，模块才会把数据发给用户设备，在 HF-A11x 模块内部有 1M Byte 的 FIFO 保存用户数据。

4.3 软件工作原理

在 WiFi 传感网实验中，子节点上的传感器把采集到的信息通过 WiFi 发送给根节点的 WiFi，在通过串口发送给 STM32。利用串口工具可以采集根节点收到的信息，再根据 WiFi 串口通信协议和传感器底层协议对传感器接收到的物理信息进行判断。

- **WiFi 串口通信协议**

```
u8 DataHeadH;       //包头 0xEE
u8 DataDeadL;       //包头  0xCC
u8 NetID;           //所属网络标识  00(zigbee) 01(蓝牙)02(WiFi)03(IPv6)04(RFID)
u8 NodeAddress[4];  //节点地址
u8 FamilyAddress[4];  //根节点地址
u8 NodeState;       //节点状态
u8 NodeChannel;     //Wifi 节点编号
u8 ConnectPort;     //通信端口
u8 SensorType;      //传感器类型编号
u8 SensorID;        //相同类型传感器 ID
u8 SensorCMD;       //节点命令序号
u8 Sensordata1;     //节点数据 1
u8 Sensordata2;     //节点数据 2
u8 Sensordata3;     //节点数据 3
u8 Sensordata4;     //节点数据 4
u8 Sensordata5;     //节点数据 5
u8 Sensordata6;     //节点数据 6
u8 Resv1;           //保留字节 1
u8 Resv2;           //保留字节 2
u8 DataEnd;         //节点包尾 0xFF
```

一帧数据为定长 26 字节。

- **传感器说明**

传感器名称	传感器类型编号	传感器输出数据说明
磁检测传感器	0x01	1-有磁场；0-无磁场
光照传感器	0x02	1-有光照；0-无光照
红外对射传感器	0x03	1-有障碍；0-无障碍
红外反射传感器	0x04	1-有障碍；0-无障碍
结露传感器	0x05	1-有结露；0-无结露
酒精传感器	0x06	1-有酒精；0-无酒精
人体检测传感器	0x07	1-有人；0-无人
三轴加速度传感器	0x08	XH, XL, YH, YL, ZH, ZL
声响检测传感器	0x09	1-有声音；0-无声音
温湿度传感器	0x0A	HH, HL, TH, TL
烟雾传感器	0x0B	1-有烟雾；0-无烟雾
振动检测传感器	0x0C	1-有振动；0-无振动
传感器扩展板	0xFF	用户自定义

- **传感器底层协议**

传感器模块	发送	返回	意义
磁检测传感器	CC EE 01 NO 01 00 00 FF 查询是否有磁场	EE CC 01 NO 01 00 00 00 00 00 00 00 FF	无磁场
		EE CC 01 NO 01 00 00 00 00 00 01 00 00 FF	有磁场
光照传感器	CC EE 02 NO 01 00 00 FF 查询是否有光照	EE CC 02 NO 01 00 00 00 00 00 00 00 00 FF	无光照
		EE CC 02 NO 01 00 00 00 00 00 01 00 00 FF	有光照
红外对射传感器	CC EE 03 NO 01 00 00 FF 查询红外对射传感器是否有障碍	EE CC 03 NO 01 00 00 00 00 00 00 00 00 FF	无障碍
		EE CC 03 NO 01 00 00 00 00 00 01 00 00 FF	有障碍
红外反射传感器	CC EE 04 NO 01 00 00 FF 查询红外反射传感器是否有障碍	EE CC 04 NO 01 00 00 00 00 00 00 00 00 FF	无障碍
		EE CC 04 NO 01 00 00 00 00 00 01 00 00 FF	有障碍
结露传感器	CC EE 05 NO 01 00 00 FF 查询是否有结露	EE CC 05 NO 01 00 00 00 00 00 00 00 00 FF	无结露
		EE CC 05 NO 01 00 00 00 00 00 01 00 00 FF	有结露
酒精传感器	CC EE 06 NO 01 00 00 FF 查询是否检测到酒	EE CC 06 NO 01 00 00 00 00 00 00 00 00 FF	无酒精
		EE CC 06 NO 01 00 00 00 00 00	有酒精

	精	01 00 00 FF	
人体检测传感器	CC EE 07 NO 01 00 00 FF 查询是否检测到人	EE CC 07 NO 01 00 00 00 00 00 00 00 FF	无人
		EE CC 07 NO 01 00 00 00 00 00 01 00 00 FF	有人
三轴加速度传感器	CC EE 08 NO 01 00 00 00 FF 查询 XYZ 轴加速度	EE CC 08 NO 01 XH XL YH YL ZH ZL 00 00 FF	XYZ 轴加速度
声响检测传感器	CC EE 09 NO 01 00 00 00 FF 查询是否有声响	EE CC 09 NO 01 00 00 00 00 00 00 00 00 FF	无声响
		EE CC 09 NO 01 00 00 00 00 00 01 00 00 FF	有声响
温湿度传感器	CC EE 0A NO 01 00 00 00 FF 查询湿度和温度	EE CC 0A NO 01 00 00 HH HL TH TL 00 00 FF	湿度和温度值
烟雾传感器	CC EE 0B NO 01 00 00 00 FF 查询是否检测到烟雾	EE CC 0B NO 01 00 00 00 00 00 00 00 00 FF	无烟雾
		EE CC 0B NO 01 00 00 00 00 00 01 00 00 FF	有烟雾
振动检测传感器	CC EE 0C NO 01 00 00 00 FF 查询是否检测到振动	EE CC 0C NO 01 00 00 00 00 00 00 00 00 FF	无振动
		EE CC 0C NO 01 00 00 00 00 00 01 00 00 FF	有振动

备注：更多传感器模块通讯协议资料，请参加产品光盘附带传感器协议说明文档。

- **主要代码分析**

```
// 接收传感器数据缓冲区
u8 rx_buf[14];
u8 rx_counter;
u8 Uart_RecvFlag = 0;
// 发送缓冲区
u8 tx_buf[14];

    GPIO_InitTypeDef GPIO_InitStructure;
    u8 i = 0;
    NVIC_Configuration();    //中断设置
    CLI();
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOB |
                       RCC_APB2Periph_GPIOC, ENABLE);
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_All;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
    GPIO_Init(GPIOB, &GPIO_InitStructure);
```

```
    GPIO_Init(GPIOC, &GPIO_InitStructure);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOB |
                           RCC_APB2Periph_GPIOC, DISABLE);
    //JTAG_Remap
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA|RCC_APB2Periph_GPIOB|RCC_APB2Periph_AFIO, ENABLE);
    // JTAG-DP Disabled and SW-DP Enabled
    GPIO_PinRemapConfig(GPIO_Remap_SWJ_JTAGDisable, ENABLE);
    Systick_Init(72);   //系统时钟设置
    LED_Init();
    LED_USER_On();
    UART1_Configuration();     //串口 1 设置
    UART2_Configuration();     //串口 2 设置
    Wifi_Init();
    // 等待 Wifi ready
    while(GPIO_ReadInputDataBit(GPIOB,GPIO_Pin_5) != Bit_RESET);
    // 等待 Wifi link
    while(GPIO_ReadInputDataBit(GPIOB,GPIO_Pin_3) != Bit_RESET);
    delay_ms(1000);
    SEI();   //开中断
```

AP 节点输出函数。

```
    while(1)
    {
        Uart_RecvFlag = 0; // 允许串口接收数据
        while(Uart_RecvFlag == 0); // 等待接受数据

        // 处理并发送数据
        for(i = 0; i < 14;i++)
            tx_buf[i] = rx_buf[i];

        UART1_SendString(tx_buf, 14);     //串口 1 发送数据

        LED_USER_Toggle();
        delay_ms(2000);
    }
```

STA 向 AP 发送发送数据函数。

```
while(1)
    {
        Uart_RecvFlag = 0; // 允许串口接收数据
        while(Uart_RecvFlag == 0); // 等待接受数据

        // 处理并发送数据
        for(i = 0; i < 14;i++)
```

```
            tx_buf[i] = rx_buf[i];

        UART2_SendString(tx_buf, 14);

        LED_USER_Toggle();
        delay_ms(2000);
    }
```

STA 串口 1 中断处理函数。

```
void USART1_IRQHandler(void)
{
    u8 data;
    if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)
    {
        CLI();
        data = USART_ReceiveData(USART1);
        if (!Uart_RecvFlag)
        {
            rx_buf[rx_counter] = data;
            switch (rx_counter)
            {
                case 0:
                    if (data == 0xEE)       rx_counter = 1;
                    break;
                case 1:
                    if (data == 0xCC)       rx_counter = 2;
                    else      rx_counter = 0;
                    break;
                case 2:
                case 3:
                case 4:
                case 5:
                case 6:
                case 7:
                case 8:
                case 9:
                case 10:
                case 11:
                case 12:
                    rx_counter++;
                    break;
                case 13:
                    if (data == 0xFF)
                        Uart_RecvFlag = 1;
                    rx_counter = 0;
                    break;
```

```
            }
        }
        else
            rx_counter = 0;
        SEI();
        /* Clear the USART1 Receive interrupt */
        USART_ClearITPendingBit(USART1, USART_IT_RXNE);
    }
}
```

AP 节点串口 2 中断处理函数。

```
void USART2_IRQHandler(void)
{
    u8 data;
    if(USART_GetITStatus(USART2, USART_IT_RXNE) != RESET)
    {
        CLI();
        data = USART_ReceiveData(USART2);
        if (!Uart_RecvFlag)
        {
            rx_buf[rx_counter] = data;
            switch (rx_counter)
            {
                case 0:
                    if (data == 0xEE)       rx_counter = 1;
                    break;
                case 1:
                    if (data == 0xCC)       rx_counter = 2;
                    else      rx_counter = 0;
                    break;
                case 2:
                case 3:
                case 4:
                case 5:
                case 6:
                case 7:
                case 8:
                case 9:
                case 10:
                case 11:
                case 12:
                    rx_counter++;
                    break;
                case 13:
                    if (data == 0xFF)
                        Uart_RecvFlag = 1;
```

```
                    rx_counter = 0;
                    break;
                }
            }
            else
                rx_counter = 0;
            SEI();
            /* Clear the USART2 Receive interrupt */
            USART_ClearITPendingBit(USART2, USART_IT_RXNE);
        }
}
```

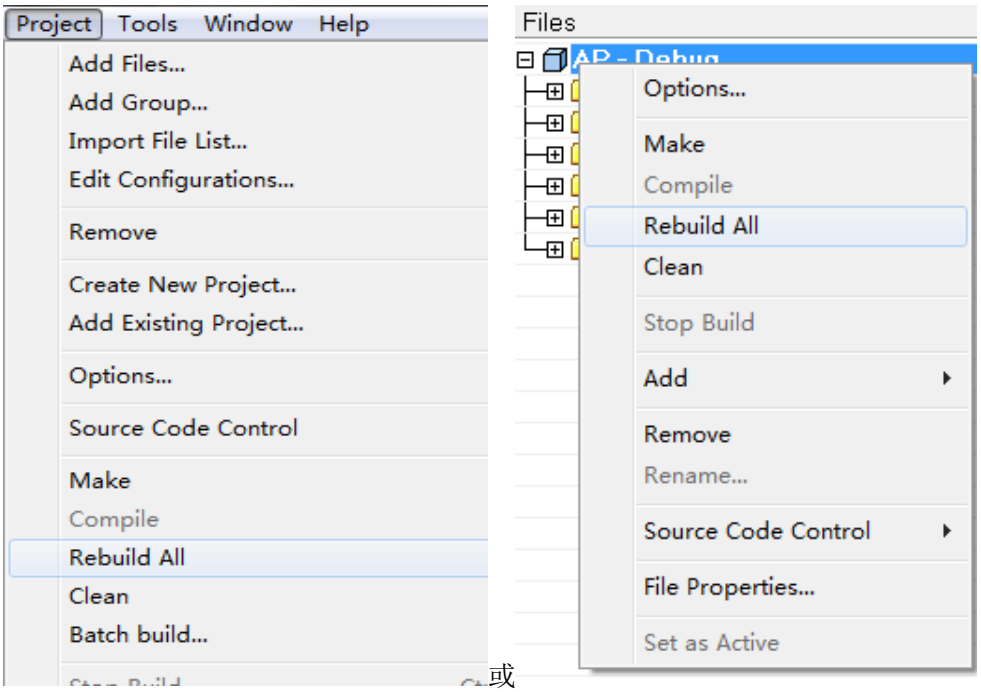
5. 实验步骤

1）用 J-OB 仿真器和 J-OB 转接板将实验箱与 PC 机相连，用实验箱配套的电源线将实验箱与电源相连，并打开开关给设备供电。

2）使用实验设备上的“+”、“-”键选中实验设备上的 WiFi 模块，打开盖模块的电源开关。

3）用“WiFi 组网实验”的方法分别将 WiFi 模块设定成 AP 模式（一个）和 STA 模式（一个或多个）,在进行 WiFi 模块设置时，服务器地址应设置为“192.168.1.1”。服务器地址的设置方法请参见“基于 WiFi 的配置组网（网络方式）实验”或是“基于 WiFi 的配置组网（AT 命令方式）实验”。在“基于 WiFi 的配置组网（网络方式）实验”中设置服务器地址在实验步骤 7）中的网络设置部分。

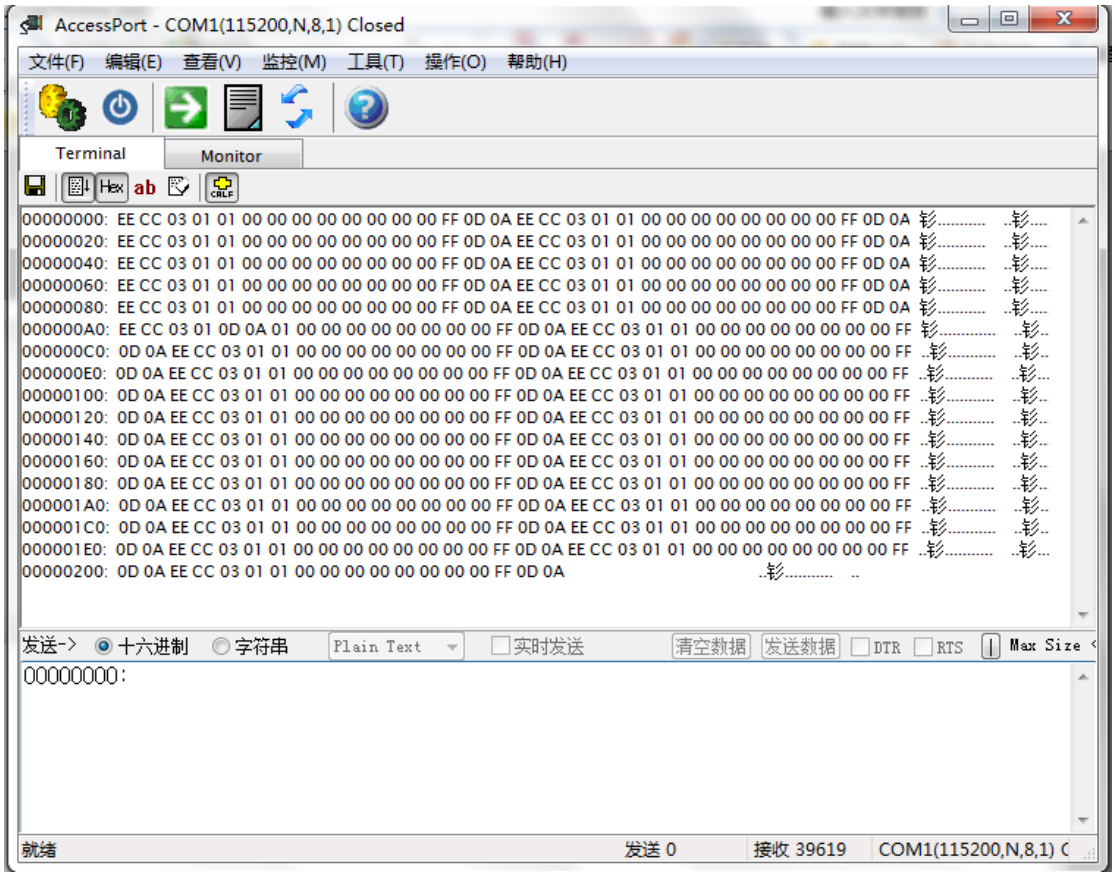
4）打开本次实验的工程，对这次的工程进行编译，编译方法可使用界面中的“Project”中的“Rebuild All ”或者选中工程栏中的工程文件，然后右键选择“Rebuild All”进行编译。具体方法如下图：



或

5）将编译无错误的程序分别烧录进各个对应的模块中，然后将各个模块断电，把传感器分别插在各个模块中，先给 AP 模块上电，到 Ready 指示灯亮后再给 STA 模块上电。

6）将 AP 节点模块从实验箱上取下，将模块放在 USB2UART 上，并用连接线将其连接在 PC 机上，打开串口调试工具，对串口调试工具进行设置，波特率 115200，检验位 NONE，数据位 8，停止位 1，用串口工具观察 AP 点接收到的信息，可参见下图，图中具体内容由 STA 模块上的传感器的情况决定，不代表最终数据。



第六章. 无线通讯模块之 RFID 通信实验

本章主要介绍无线通讯模块部分的 RFID 相关的实验内容，采用 ICS-IOT-CEP 平台配套的 RFID 模块。通过本章实验内容，读者即可以迅速掌握基上述 RFID 射频模块的应用开发。

实验一. RFID 自动读卡实验

1. 实验目的

- 了解 RFID 相关知识。
- 掌握 RFID 模块自动识别 IC 卡工作原理。

2. 实验环境

- 软件：IAR SWSTM8 1.30。
- 硬件：RFID 射频模块，电子标签，ST-Link。

3. 实验内容

- 编程使用 RFID 模块，完成自动识别读取 IC 卡卡号功能。

4. 实验原理

4.1 STM8S 处理器概述

本实验所使用 RFID 模块由 STM8 处理器和 MFRC531(高集成非接触读写芯片)两片芯片搭建而成的。

STM8 是基于 8 位框架结构的微控制器，其 CPU 内核有 6 个内部寄存器，通过这些寄存器可高效地进行数据处理。STM8 的指令集支持 80 条基本语句及 20 种寻址模式，而且 CPU 的 6 个内部寄存器都拥有可寻址的地址。

STM8 内部的 FLASH 程序存储器和数据 EEPROM 由一组通用寄存器来控制。用户可以使用这些寄存器来编程或擦除存储器的内容、设置写保护、或者配置特定的低功耗模式。用户也可以对器件的选项字节(Option byte) 进行编程。

FLASH

- **STM8S EEPROM 分为两个存储器阵列：**

— 最多至 128K 字节的 FLASH 程序存储器，不同的器件容量有所不同。

— 最多至 2K 字节的数据 EEPROM(包括 option byte —选择字节)，不同的器件容量有所不同。

- **编程模式**

— 字节编程和自动快速字节编程(没有擦除操作)

— 字编程

— 块编程和快速块编程(没有擦除操作)

— 在编程/ 擦除操作结束时和发生非法编程操作时产生中断

- **读同时写(RWW)功能。该特性并不是所有 STM8S 器件都拥有。**
- **在应用编程(IAP)和在线编程(ICP)能力。**
- **保护特性**

— 存储器读保护(ROP)

— 基于存储器存取安全系统(MASS 密钥)的程序存储器写保护

— 基于存储器存取安全系统(MASS 密钥)的数据存储器写保护

— 可编程的用户启动代码区域(UBC) 写保护

- **在待机(Halt) 模式和活跃待机(Active-halt)模式下，存储器可配置为运行状态和掉电状态。**

数据 EEPROM(DATA) 区域可用于存储用户具体项目所需的数据。默认情况下，DATA 区域是写保护的，这样可以在主程序工作在 IAP 模式时防止 DATA 区域被无意地修改。只有使用特定的 MASS 密钥才能对 DATA 区域的写保护解锁。STM8 我们就简单介绍这么多（详见 STM8 数据手册）。

4.2 MFR531 概述

MF RC531 是应用于 13.56MHz 非接触式通信中高集成读写卡芯片系列中的一员。该读写卡芯片系列利用了先进的调制和解调概念，完全集成了在 13.56MHz 下所有类型的被动非接触式通信方式和协议。芯片管脚兼容 MF RC500、MF RC530 和 SL RC400。

MF RC531 支持 ISO/IEC14443A/B 的所有层和 MIFARE®经典协议，以及与该标准兼容的标准。支持高速 MIFARE®非接触式通信波特率。内部的发送器部分不需要增加有源电路就能够直接驱动近操作距离的天线（可达 100mm）。接收器部分提供一个坚固而有效的解调和解码电路，用于 ISO14443A 兼容的应答器信号。数字部分处理 ISO14443A 帧和错误检测（奇偶&CRC）。此外，它还支持快速 CRYPTO1 加密算法，用于验证 MIFARE 系列产品。与主机通信模式有 8 位并行和 SPI 模式，用户可根据不同的需求选择不同的模式，这样给读卡器/ 终端的设计提供了极大的灵活性。

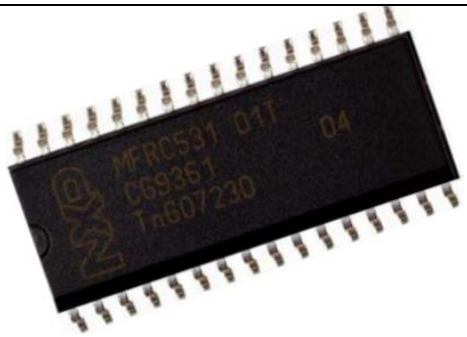


图 1.1 MFRC531

特性

- 高集成度的调制解调电路；
- 采用少量外部器件，即可输出驱动级接至天线；
- 最大工作距离 100mm；
- 支持 ISO/IEC14443 A/B 和 MIFARE®经典协议；
- 支持非接触式高速通信模式，波特率可达 424kb/s；
- 采用 Crypto1 加密算法并含有安全的非易失性内部密匙存储器；
- 管脚兼容 MF RC500、MF RC530 和 SL RC400；
- 与主机通信的 2 种接口：并行接口和 SPI，可满足不同用户的需求；
- 自动检测微处理器并行接口类型；
- 灵活的中断处理；
- 64 字节发送和接收 FIFO 缓冲区；
- 带低功耗的硬件复位；
- 可编程定时器；
- 唯一的序列号；
- 用户可编程初始化配置；
- 面向位和字节的帧结构；
- 数字、模拟和发送器部分经独立的引脚分别供电；
- 内部振荡器缓存器连接 13.56MHz 石英晶体；
- 数字部分的电源（DVDD ）可选择 3.3V 或 5V；
- 在短距离应用中，发送器（天线驱动）可以用 3.3V 供电。

MF RC531 适用于各种基于 ISO/IEC 14443 标准，并且要求低成本、小尺寸、高性能以及单电源的非接触式通信的应用场合。

- 公共交通终端；
- 手持终端；

- 板上单元；
- 非接触式 PC 终端；
- 计量；
- 非接触式公用电话。

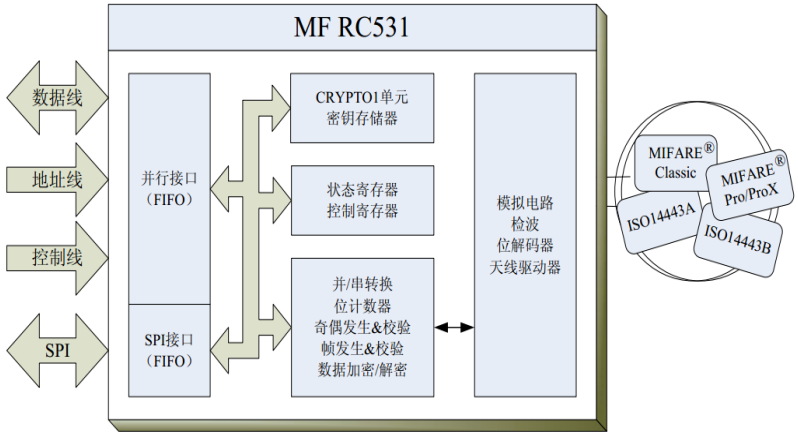


图 1.2 功能框图

并行微控制器接口自动检测连接的 8 位并行接口的类型。它包含一个的双向 FIFO 缓冲区和一个可配置的中断输出。这样就为连接各种 MCU 提供了很大的灵活性。即使使用非常低成本的器件也能满足高速非接触式通信的要求。带 FIFO 的 SPI 从机接口，其串行时钟 SCK 由主机提供。

数据处理部分执行数据的并行一串行转换。它支持的帧包括 CRC 和奇偶校验。它以完全透明的模式进行操作，因而支持 ISO14443A 的所有层。状态和控制部分允许对器件进行配置以适应环境的影响并使性能调节到最佳状态。当与 MIFARE Standard 和 MIFARE 产品通信时，使用高速 CRYPTO1 流密码单元和一个可靠的非易失性密匙存储器。

模拟电路包含了一个具有非常低阻抗桥驱动器输出的发送部分。这使得最大操作距离可达 100mm。接收器可以检测到并解码非常弱的应答信号。由于采用了非常先进的技术，接收器已不再是限制操作距离的因素了。

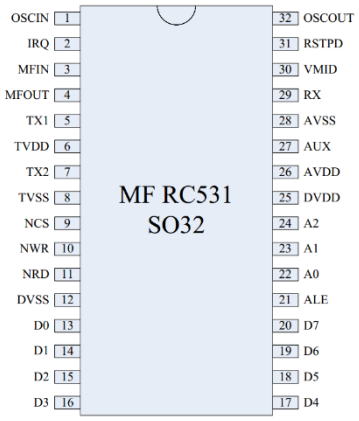


图 1.3 管脚图

非接触式天线使用以下 4 个管脚：

名 称	类 型	功 能
TX1, TX2	输出缓冲	天线驱动器
WMID	模拟	参考电压
RX	输入模拟	天线输入信号

表 1.1 天线管脚描述

为了驱动天线，MF RC531 通过 TX1 和 TX2 提供 13.56MHz 的能量载波。根据寄存器的设定对发送数据进行调制得到发送的信号。卡采用 RF 场的负载调制进行响应。天线拾取的信号经过天线匹配电路送到 RX 脚。MF RC531 内部接收器对信号进行检测和解调并根据寄存器的设定进行处理。然后数据发送到并行接口由微控制器进行读取。

MF RC531 支持 MIFARE®有源天线的概念。它可以处理管脚 MFIN 和 MFOUT 处的 MIFARE®核心模块的基带信号 NPAUSE 和 KOMP 。

名称	类型	功能
MFIN	带施密特触发器的输入	MIFARE 接口输入
MFOUT	输出	MIFARE 接口输出

表 1.2 MIFARE 接口管脚描述

MIFARE®接口可采用下列方式与 MF RC531 的模拟或数字部分单独通信：

- 模拟电路可通过 MIFARE 接口独立使用。这种情况下，MFIN 连接到外部产生的 NPAUSE 信号。MFOUT 提供 KOMP 信号。
- 数字电路可通过 MIFARE®接口驱动外部信号电路。这种情况下，MFOUT 提供内部产生的 NPAUSE 信号而 MFIN 连接到外部输入的 KOMP 信号。

4 线 SPI 接口：

名称	类型	功能
A0	带施密特触发器的 I/O	MOSI
A2	带施密特触发器的 I/O	SCK
D0	带施密特触发器的 I/O	MISO
ALE	带施密特触发器的 I/O	NSS

表 1.3 SPI 接口管脚描述

代码实现如下

```
//////////////////////////////////////////////////////////////////////
//功      能：寻卡
//参数说明: req_code[IN]:寻卡方式
//                0x52 = 寻感应区内所有符合 14443A 标准的卡
//                0x26 = 寻未进入休眠状态的卡
//          pTagType[OUT]：卡片类型代码
//                0x4400 = Mifare_UltraLight
//                0x0400 = Mifare_One(S50)
//                0x0200 = Mifare_One(S70)
//                0x0800 = Mifare_Pro
//                0x0403 = Mifare_ProX
```

```
//                          0x4403 = Mifare_DESFire
//返       回: 成功返回 MI_OK
/////////////////////////////////////////////////////////////////////
signed char PcdRequest(unsigned char req_code,unsigned char *pTagType)
{
    signed char status;
    struct TransceiveBuffer MfComData;
    struct TransceiveBuffer *pi;
    pi = &MfComData;
    MFRC531_WriteReg(RegChannelRedundancy,0x03);
    MFRC531_ClearBitMask(RegControl,0x08);
    MFRC531_WriteReg(RegBitFraming,0x07);
    MFRC531_SetBitMask(RegTxControl,0x03);
    MFRC531_SetTimer(4);
    MfComData.MfCommand = PCD_TRANSCEIVE;
    MfComData.MfLength    = 1;
    MfComData.MfData[0] = req_code;
    status = MFRC531_ISO14443_Transceive(pi);
    if (!status)
    {
          if (MfComData.MfLength != 0x10)
          {     status = MI_BITCOUNTERR;     }
    }
    *pTagType         = MfComData.MfData[0];
    *(pTagType+1) = MfComData.MfData[1];
    return status;
}
/////////////////////////////////////////////////////////////////////
//将存在 RC531 的 EEPROM 中的密钥匙调入 RC531 的 FIFO
//input: startaddr=EEPROM 地址
/////////////////////////////////////////////////////////////////////
char PcdLoadKeyE2(unsigned int startaddr)
{
     char status;
     struct TransceiveBuffer MfComData;
     struct TransceiveBuffer *pi;
     pi = &MfComData;
     MfComData.MfCommand = PCD_LOADKEYE2;
     MfComData.MfLength    = 2;
     MfComData.MfData[0] = startaddr & 0xFF;
     MfComData.MfData[1] = (startaddr >> 8) & 0xFF;
     status = MFRC531_ISO14443_Transceive(pi);
      return status;
}
/////////////////////////////////////////////////////////////////////
//功能：将已转换格式后的密钥送到 RC531 的 FIFO 中
```

```
//input:keys=密钥
//////////////////////////////////////////////////////////////////////
signed char PcdAuthKey(unsigned char *pKeys)
{
    signed char status;
    struct TransceiveBuffer MfComData;
    struct TransceiveBuffer *pi;
    pi = &MfComData;
    MFRC531_SetTimer(4);
    MfComData.MfCommand = PCD_LOADKEY;
    MfComData.MfLength    = 12;
    memcpy(&MfComData.MfData[0], pKeys, 12);
    status = MFRC531_ISO14443_Transceive(pi);
    return status;
}
//////////////////////////////////////////////////////////////////////
//功能：用存放 RC531 的 FIFO 中的密钥和卡上的密钥进行验证
//input:auth_mode=验证方式,0x60:验证 A 密钥,0x61:验证 B 密钥
//          block=要验证的绝对块号
//          g_cSNR=序列号首地址
//////////////////////////////////////////////////////////////////////
signed char PcdAuthState(unsigned char auth_mode,unsigned char block,unsigned char *pSnr)
{
    signed char status;
    struct TransceiveBuffer MfComData;
    struct TransceiveBuffer *pi;
    pi = &MfComData;
    MFRC531_WriteReg(RegChannelRedundancy,0x0F);
    MFRC531_SetTimer(4);
    MfComData.MfCommand = PCD_AUTHENT1;
    MfComData.MfLength    = 6;
    MfComData.MfData[0] = auth_mode;
    MfComData.MfData[1] = block;
    memcpy(&MfComData.MfData[2], pSnr, 4);
    status = MFRC531_ISO14443_Transceive(pi);
    if (status == MI_OK)
    {
        if (MFRC531_ReadReg(RegSecondaryStatus) & 0x07)
        {    status = MI_BITCOUNTERR;      }
        else
        {
            MfComData.MfCommand = PCD_AUTHENT2;
            MfComData.MfLength    = 0;
            status = MFRC531_ISO14443_Transceive(pi);
            if (status == MI_OK)
            {
```

```
                    if (MFRC531_ReadReg(RegControl) & 0x08)
                    {    status = MI_OK;    }
                    else
                    {    status = MI_AUTHERR;    }

                }
            }
        }
        return status;
}
//////////////////////////////////////////////////////////////////////
//读 mifare_one 卡上一块(block)数据(16 字节)
//input: addr =  要读的绝对块号
//output:readdata =  读出的数据
//////////////////////////////////////////////////////////////////////
signed char PcdRead(unsigned char addr,unsigned char *pReaddata)
{
        signed char status;
        struct TransceiveBuffer MfComData;
        struct TransceiveBuffer *pi;
        pi = &MfComData;
        MFRC531_SetTimer(4);
        MFRC531_WriteReg(RegChannelRedundancy,0x0F);
        MfComData.MfCommand = PCD_TRANSCEIVE;
        MfComData.MfLength    = 2;
        MfComData.MfData[0] = PICC_READ;
        MfComData.MfData[1] = addr;
        status = MFRC531_ISO14443_Transceive(pi);
        if (status == MI_OK)
        {
            if (MfComData.MfLength != 0x80)
            {    status = MI_BITCOUNTERR;    }
            else
            {    memcpy(pReaddata, &MfComData.MfData[0], 16);    }
        }
        return status;
}
//////////////////////////////////////////////////////////////////////
//写数据到卡上的一块
//input:adde=要写的绝对块号
//          writedata=写入数据
//////////////////////////////////////////////////////////////////////
signed char PcdWrite(unsigned char addr,unsigned char *pWritedata)
{
        signed char status;
        struct TransceiveBuffer MfComData;
```

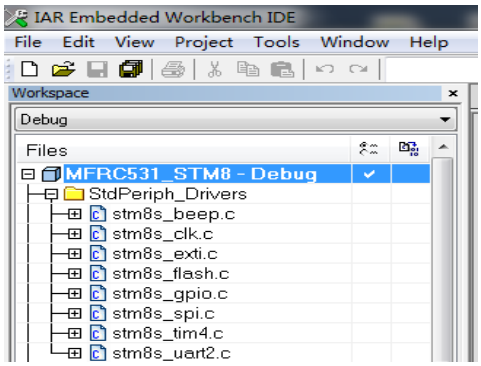
```
struct TransceiveBuffer *pi;
pi = &MfComData;
MFRC531_SetTimer(5);
MFRC531_WriteReg(RegChannelRedundancy,0x07);
MfComData.MfCommand = PCD_TRANSCEIVE;
MfComData.MfLength    = 2;
MfComData.MfData[0] = PICC_WRITE;
MfComData.MfData[1] = addr;
status = MFRC531_ISO14443_Transceive(pi);
if (status != MI_NOTAGERR)
{
    if(MfComData.MfLength != 4)
    {    status=MI_BITCOUNTERR;    }
    else
    {
        MfComData.MfData[0] &= 0x0F;
        switch (MfComData.MfData[0])
        {
            case 0x00:
                status = MI_NOTAUTHERR;
                break;
            case 0x0A:
                status = MI_OK;
                break;
            default:
                status = MI_CODEERR;
                break;
        }
    }
}
if (status == MI_OK)
{
    MFRC531_SetTimer(5);
    MfComData.MfCommand = PCD_TRANSCEIVE;
    MfComData.MfLength    = 16;
    memcpy(&MfComData.MfData[0], pWritedata, 16);
    status = MFRC531_ISO14443_Transceive(pi);
    if (status != MI_NOTAGERR)
    {
        MfComData.MfData[0] &= 0x0F;
        switch(MfComData.MfData[0])
        {
            case 0x00:
                status = MI_WRITEERR;
                break;
            case 0x0A:
```

```
                    status = MI_OK;
                    break;
                default:
                    status = MI_CODEERR;
                    break;
            }
        }
        MFRC531_SetTimer(4);
    }
    return status;
}
```

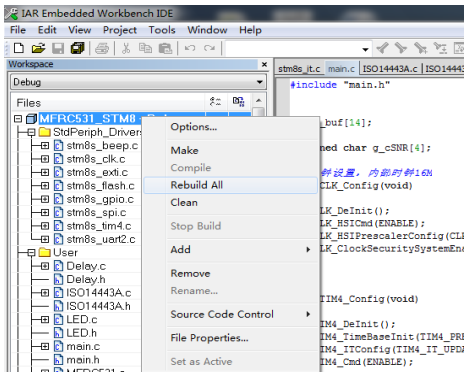
5. 实验步骤

1）首先我们要把 RFID 模块插到实验箱的主板上的串口（注意：不要插到无线模块上的串口，直接插到主板上的串口），再把 ST-Link 配合 JTAG 仿真器插到标有 ST-Link 标志的串口上，最后把仿真器一端的 USB 线插到 PC 机的 USB 端口，通过主板上的“加”“减”按键调整要实验的 RFID 模块（会有黄色 LED 灯提示），硬件连接完毕。

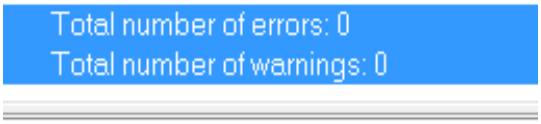
2）我们用 IAR SWSTM8 1.30 软件，打开..\RFID_读卡号实验\Project\MFRC531_ATM8.eww。



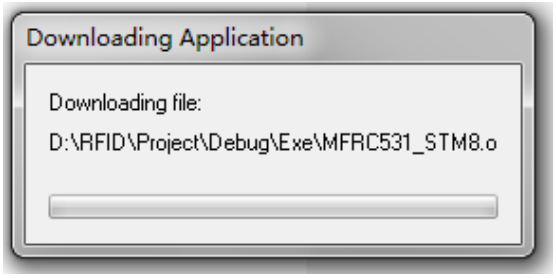
3）打开后点击“Project”下面的“Rebuild All”或者选中工程文件右键“Rebuild All”把我们的工程编译一下。



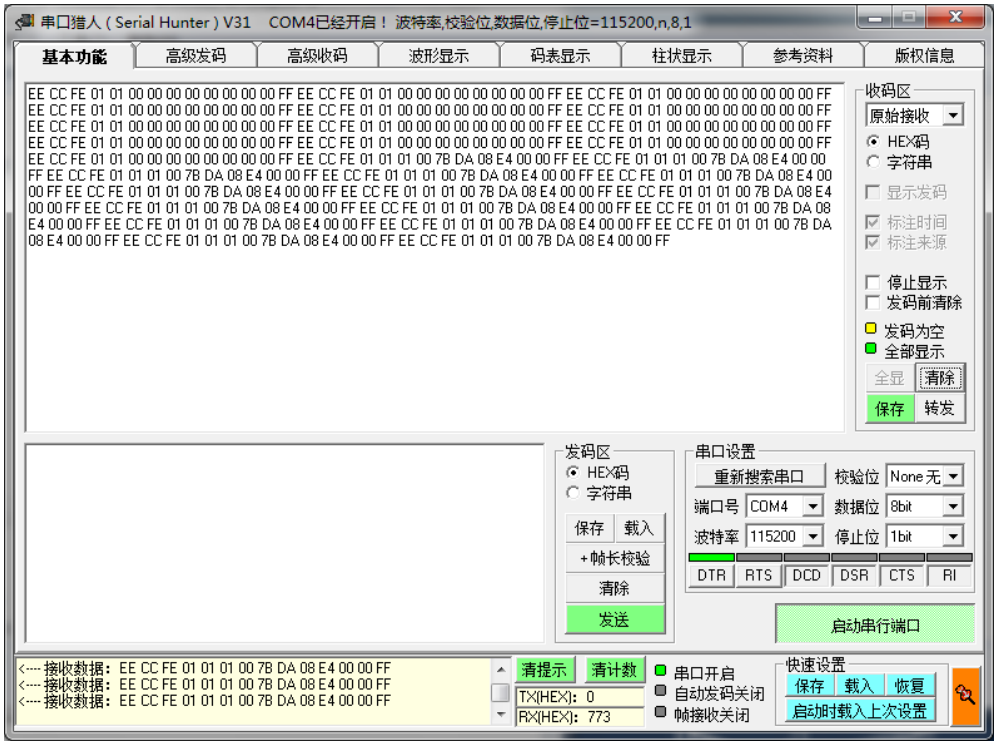
4）点击“Rebuild All”编译完后，无警告，无错误。



5）编译完后我们要把程序烧到模块里，点击“  ”中间的 Download and Debug 烧录成功会听到蜂鸣器响一声。



6）我们用串口测试一下，把我们的传感器模块连接到我们的串口转 USB 模块上将 USB2UART 模块的 USB 线连接到 PC 机的 USB 端口，然后打开串口工具，配置好串口，波特率 115200，8 个数据位，一个停止位，无校验位，串口开始工作，无卡时串口返回：EE CC FE 01 01 00 00 00 00 00 00 00 00 FF，当有卡时串口返回 EE CC FE 01 01 01 00 7B DA 08 E4 00 00 FF。



实验二. 基于 RFID 的电子钱包应用实验

1. 实验目的

- 了解 RFID 相关知识。
- 掌握 RFID 模块读写 IC 卡数据的原理与方法。

2. 实验环境

- 软件：IAR SWSTM8 1.30。
- 硬件：RFID 射频模块，电子标签，ST-Link。

3. 实验内容

- 编程使用 RFID 模块，完成自动识别读、写取 IC 卡信息功能，实现 RFID 电子钱包软件功能。

4. 实验原理

4.1 STM8S 处理器概述

本实验所使用 RFID 模块由 STM8 处理器和 MFRC531(高集成非接触读写芯片)两片芯片搭建而成的。

STM8 是基于 8 位框架结构的微控制器，其 CPU 内核有 6 个内部寄存器，通过这些寄存器可高效地进行数据处理。STM8 的指令集支持 80 条基本语句及 20 种寻址模式，而且 CPU 的 6 个内部寄存器都拥有可寻址的地址。

STM8 内部的 FLASH 程序存储器和数据 EEPROM 由一组通用寄存器来控制。用户可以使用这些寄存器来编程或擦除存储器的内容、设置写保护、或者配置特定的低功耗模式。用户也可以对器件的选项字节(Option byte) 进行编程。

FLASH

- **STM8S EEPROM 分为两个存储器阵列：**

— 最多至 128K 字节的 FLASH 程序存储器，不同的器件容量有所不同。

— 最多至 2K 字节的数据 EEPROM(包括 option byte —选择字节)，不同的器件容量有所不同。

- **编程模式**

— 字节编程和自动快速字节编程(没有擦除操作)

— 字编程

— 块编程和快速块编程(没有擦除操作)

— 在编程/ 擦除操作结束时和发生非法编程操作时产生中断

- **读同时写(RWW)功能。该特性并不是所有 STM8S 器件都拥有。**
- **在应用编程(IAP)和在线编程(ICP)能力。**
- **保护特性**

— 存储器读保护(ROP)

— 基于存储器存取安全系统(MASS 密钥)的程序存储器写保护

— 基于存储器存取安全系统(MASS 密钥)的数据存储器写保护

— 可编程的用户启动代码区域(UBC) 写保护

- **在待机(Halt) 模式和活跃待机(Active-halt)模式下，存储器可配置为运行状态和掉电状态。**

数据 EEPROM(DATA) 区域可用于存储用户具体项目所需的数据。默认情况下，DATA 区域是写保护的，这样可以在主程序工作在 IAP 模式时防止 DATA 区域被无意地修改。只有使用特定的 MASS 密钥才能对 DATA 区域的写保护解锁。STM8 我们就简单介绍这么多（详见 STM8 数据手册）。

4.2 MFR531 概述

MF RC531 是应用于 13.56MHz 非接触式通信中高集成读写卡芯片系列中的一员。该读写卡芯片系列利用了先进的调制和解调概念，完全集成了在 13.56MHz 下所有类型的被动非接触式通信方式和协议。芯片管脚兼容 MF RC500、MF RC530 和 SL RC400。

MF RC531 支持 ISO/IEC14443A/B 的所有层和 MIFARE®经典协议，以及与该标准兼容的标准。支持高速 MIFARE®非接触式通信波特率。内部的发送器部分不需要增加有源电路就能够直接驱动近操作距离的天线（可达 100mm）。接收器部分提供一个坚固而有效的解调和解码电路，用于 ISO14443A 兼容的应答器信号。数字部分处理 ISO14443A 帧和错误检测（奇偶＆CRC）。此外，它还支持快速 CRYPTO1 加密算法，用于验证 MIFARE 系列产品。与主机通信模式有 8 位并行和 SPI 模式，用户可根据不同的需求选择不同的模式，这样给读卡器/ 终端的设计提供了极大的灵活性。

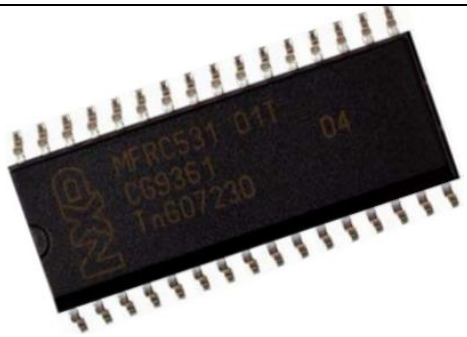


图 1.1 MFRC531

MF RC531 适用于各种基于 ISO/IEC 14443 标准，并且要求低成本、小尺寸、高性能以及单电源的非接触式通信的应用场合。

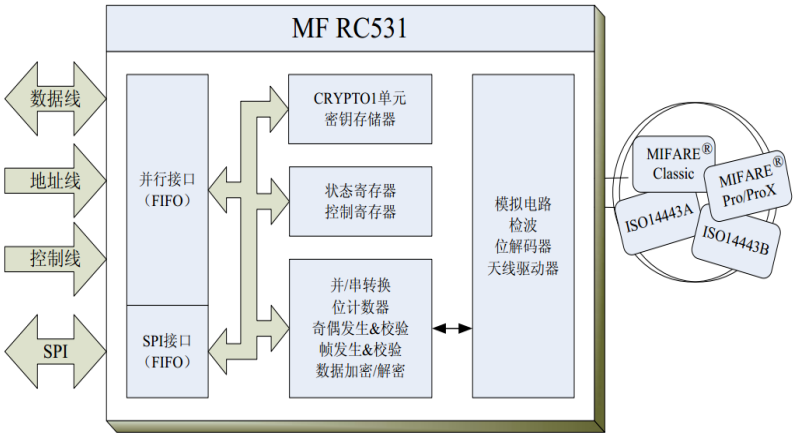


图 1.2 功能框图

并行微控制器接口自动检测连接的 8 位并行接口的类型。它包含一个的双向 FIFO 缓冲区和一个可配置的中断输出。这样就为连接各种 MCU 提供了很大的灵活性。即使使用非常低成本的器件也能满足高速非接触式通信的要求。带 FIFO 的 SPI 从机接口，其串行时钟 SCK 由主机提供。

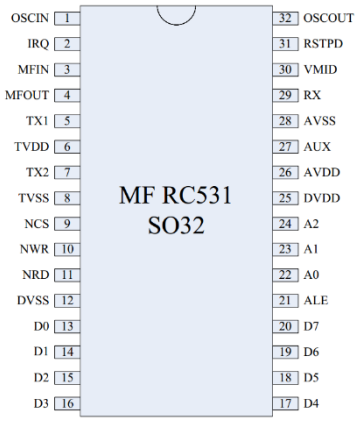


图 1.3 管脚图

非接触式天线使用以下 4 个管脚：

名 称	类 型	功 能
TX1, TX2	输出缓冲	天线驱动器
WMID	模拟	参考电压
RX	输入模拟	天线输入信号

表 1.1 天线管脚描述

为了驱动天线，MF RC531 通过 TX1 和 TX2 提供 13.56MHz 的能量载波。根据寄存器的设定对发送数据进行调制得到发送的信号。卡采用 RF 场的负载调制进行响应。天线拾取的信号经过天线匹配电路送到 RX 脚。MF RC531 内部接收器对信号进行检测和解调并根据寄存器的设定进行处理。然后数据发送到并行接口由微控制器进行读取。

MF RC531 支持 MIFARE®有源天线的概念。它可以处理管脚 MFIN 和 MFOUT 处的 MIFARE®核心模块的基带信号 NPAUSE 和 KOMP 。

名称	类型	功能
MFIN	带施密特触发器的输入	MIFARE 接口输入
MFOUT	输出	MIFARE 接口输出

表 1.2 MIFARE 接口管脚描述

MIFARE®接口可采用下列方式与 MF RC531 的模拟或数字部分单独通信：

- 模拟电路可通过 MIFARE 接口独立使用。这种情况下，MFIN 连接到外部产生的 NPAUSE 信号。MFOUT 提供 KOMP 信号。
- 数字电路可通过 MIFARE®接口驱动外部信号电路。这种情况下，MFOUT 提供内部产生的 NPAUSE 信号而 MFIN 连接到外部输入的 KOMP 信号。

4 线 SPI 接口：

名称	类型	功能
A0	带施密特触发器的 I/O	MOSI
A2	带施密特触发器的 I/O	SCK
D0	带施密特触发器的 I/O	MISO
ALE	带施密特触发器的 I/O	NSS

表 1.3 SPI 接口管脚描述

代码实现如下。

```
// RC531 初始化，上电后需要延时一段时间 500ms
signed char MFRC531_Init(void)
{
    signed char status = MI_OK;
    signed char n = 0xFF;
    unsigned int i = 3000;
    // CS - PC4
    GPIO_Init(MFRC531_CS_PORT, MFRC531_CS_PIN, GPIO_MODE_OUT_PP_HIGH_FAST);
    MFRC531_SPI_DIS();
    // RST - PC3
    GPIO_Init(MFRC531_RST_PORT, MFRC531_RST_PIN, GPIO_MODE_OUT_PP_HIGH_FAST);
// 读寄存器
unsigned char MFRC531_ReadReg(unsigned char addr)
{
```

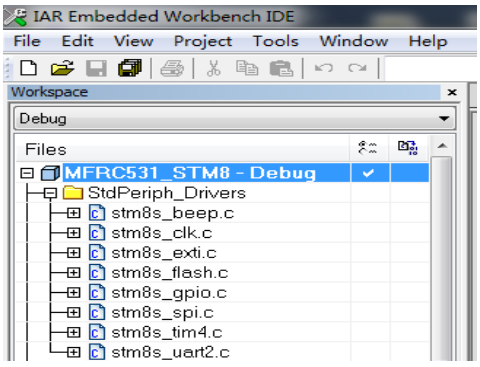
```
    unsigned char SndData;
    unsigned char ReData;
        // 处理第一个字节，bit7:MSB=1,bit6~1:addr,bit0:0
    SndData  = (addr << 1);
    SndData |= 0x80;
    SndData &= 0xFE;
     MFRC531_SPI_EN();
    SPI_RWByte(SndData);
    ReData = SPI_RWByte(0x00);
    MFRC531_SPI_DIS();
    return ReData;
}
// 写寄存器
void MFRC531_WriteReg(unsigned char addr, unsigned char data)
{
    unsigned char SndData;
     // 处理第一个字节，bit7:MSB=0,bit6~1:addr,bit0:0
    SndData  = (addr << 1);
    SndData &= 0x7E;
    MFRC531_SPI_EN();
    SPI_RWByte(SndData);
    SPI_RWByte(data);
    MFRC531_SPI_DIS();
}
// 置 RC531 寄存器位
void MFRC531_SetBitMask(unsigned char addr,unsigned char mask)
{
    unsigned char temp;
    temp = MFRC531_ReadReg(addr);
    MFRC531_WriteReg(addr, temp | mask);
}
// 清 RC531 寄存器位
void MFRC531_ClearBitMask(unsigned char addr,unsigned char mask)
{
    unsigned char temp;
    temp = MFRC531_ReadReg(addr);
    MFRC531_WriteReg(addr, temp & ~mask);
}
//清空缓冲区
unsigned char MFRC531_ClearFIFO(void)
{
    unsigned char i;
     MFRC531_SetBitMask(RegControl, 0x01);
    delay_us(100);
    // 判断 FIFO 是否被清楚
    i = MFRC531_ReadReg(RegFIFOLength);
```

```
    if(i == 0)
            return 1;
      else
            return 0;
}
//读缓冲区
unsigned char MFRC531_ReadFIFO(unsigned char *Send_Buf)
{
      unsigned char len, i;
    len = MFRC531_ReadReg(RegFIFOLength);
      for(i = 0;i < len; i++)
            Send_Buf[i] = MFRC531_ReadReg(RegFIFOData);
       return len;
}
    //写缓冲区
void   MFRC531_WriteFIFO(unsigned char *Send_Buf,unsigned char Length)
{
      unsigned char i;
      for(i = 0; i < Length; i++)
            MFRC531_WriteReg(RegFIFOData, Send_Buf[i]);
}
```

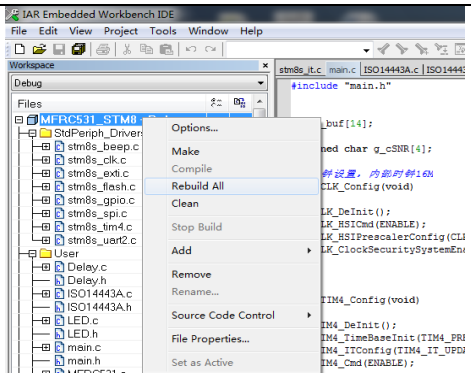
5. 实验步骤

1）首先我们要把 RFID 模块插到实验箱的主板上的串口（注意：不要插到无线模块上的串口，直接插到主板上的串口），再把 ST-Link 配合 JTAG 仿真器插到标有 ST-Link 标志的串口上，最后把仿真器一端的 USB 线插到 PC 机的 USB 端口，通过主板上的“加”“减”按键调整要实验的 RFID 模块（会有黄色 LED 灯提示），硬件连接完毕。

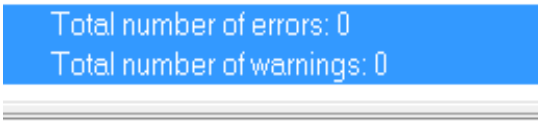
2）我们用 IAR SWSTM8 1.30 软件，打开 RFID_电子钱包实验\Project\MFRC531_ATM8.eww。

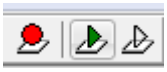


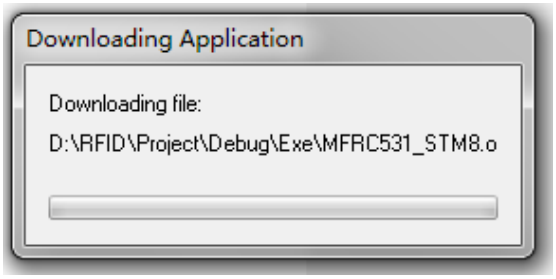
3）打开后点击“Project”下面的“Rebuild All”或者选中工程文件右键“Rebuild All”把我们的工程编译一下。



4）点击“Rebuild All”编译完后，无警告，无错误。



5）编译完后我们要把程序烧到模块里，点击“  ”中间的 Download and Debug 烧录成功会听到蜂鸣器响一声。



6）我们用串口工具（用户也可自行选择其他串口软件）测试一下，打开串口工具，配置一下端口，波特率 115200，8 个数据位，一个停止位，无校验位。串口开始工作。

7）我们先发送一组充值命令：CC EE FE NO 01 XX XX XX XX FF（HEX 格式数据），串口会返回一组字符。如下图：


串口猎人 (Serial Hunter) V31 COM1已经开启！ 波特率,校验位,数据位,停止位=115200,n,8,1
基本功能
高级发码
高级收码
波形显示
码表显示
柱状显示
参考资料
版权信息
EE CC FE 01 03 4D 48 36 E8 00 45 03 67 FF
EE CC FE 01 01 4D 48 36 E8 00 45 03 68 FF
EE CC FE 01 02 4D 48 36 E8 00 45 03 67 FF
收码区
原始接收
HEX码
字符串
显示发码
标注时间
标注来源
停止显示
发码前清除
收发不同
前端隐藏
全显
清除
保存
转发
CC EE FE 01 02 00 00 00 01 FF
发码区
HEX码
字符串
保存
载入
+帧长校验
清除
发送
串口设置
重新搜索串口
校验位
None 无
端口号
COM1
数据位
8bit
波特率
115200
停止位
1bit
DTR
RTS
DCD
DSR
CTS
RI
启动串行端口
手动发送II
----> 发送数据：CC EE FE 01 02 00 00 00 01 FF
<---- 接收数据：EE CC FE 01 02 4D 48 36 E8 00 45 03 67 FF
清提示
清计数
TX(HEX)：20
RX(HEX)：42
串口开启
自动发码关闭
帧接收关闭
快速设置
保存
载入
恢复
启动时载入上次设置

第七章. 无线通讯模块之 IPv6 模块通信实验

本章主要介绍无线通讯模块部分的 IPv6 通信的实验内容，采用 ICS-IOT-CEP 平台配套的 IPv6 模块硬件。内容由浅入深，前面已经讲述相应模块的硬件接口实验，本章主要针对网络协议栈实验及传感器网络实验等。通过本章实验内容，读者即可以迅速掌握基上述 IPv6 无线模块的开发方法，以及相应网络结构的传感器数据通讯应用设计。

实验一. 基于 IPv6 的 Contiki 系统入门实验

1. 实验目的

- 了解 Cygwin 开发环境及 Contiki 系统相关内容。
- 掌握 IPv6 模块的编程及下载使用方法。

2. 实验环境

- 硬件：ICS-IOT-CEP 教学实验平台，PC 机，J-OB 仿真器和 J-OB 转接板
- 软件：Cygwin +IAR EWARM，contiki OS，串口终端

3. 实验内容

- 使用 Cygwin 开发环境，利用 IAR 编译工具对 contiki 系统进行编译，下载程序到 IPv6 模块运行测试。

4. 实验原理

4.3 Contiki 系统简介

Contiki 是一个开源多任务事件驱动操作系统，为网络嵌入式设备设计。它轻量级的足印（footprint）很适合内存有限的 MCU。

Contiki 集成了数个独立的模块，在一个事件驱动的类线程多任务的环境中，包含了 protothread library、uIP TCP/IP(v4 和 v6)协议栈、无线传感器网络的协议套件—Rime 协议栈。

Contik 主要为网络应用程序而设计，但也可运行仅使用它的事件驱动内核的非网络程序。

Contiki 适用于只有极少量内存的嵌入式系统。Contiki 只需几 kilobyte 的代码和几百字节的内存就能提供多任务环境和内建 TCP/IP 支持。

在一个较为典型的配置中，Contiki 系统只需 2Kb 的 RAM 与 40Kb 的 ROM。 Contiki 包括了一个事件驱动的内核，因此可以在运行时动态载入上层应用程序。Contiki 中使用轻量级的 protothreads 进程模型，可以在事件驱动内核上提供一种线性的、类似于线程的编程风格。

Contiki 可运行于各种平台上，包括嵌入式微控制器（例如 TI MSP430 及 Atmel AVR）以及旧的家用电脑。程序代码量只有几 K，存储器的使用量也只有几十 Kb。

- **Contiki 的特性**

多任务内核。

每个应用程序中可选的先占式多线程。

Protothreads 模型。

TCP/IP 网络支持，包括 IPv6。

视窗系统与 GUI。

基于 VNC 的网络化远程显示。

网页浏览器。

个人网络服务器。

简单的 Telnet 客户端。

- **事件**

Contiki 内核基于事件驱动。这类系统的核心思想是，程序的每次执行都是一个事件的响应。整个系统(内核+链接库+用户代码)可以多进程并行执行。

不同的进程一般执行一段时间，然后等待事件发生。在等待时，这个进程的状态称为阻塞。当一个事件发生时，内核执行由事件传递来的信息指向的进程。在所等待的事件发生时，内核负责调用相对应的进程。

事件被分为以下三种：

- 定时器事件（timer events）：进程可以设置一个定时器，在给定的时间之后生成一个事件，进程一直阻塞直到定时器终止，才继续执行。这对周期性操作很有用，或者用于网络协议，比如涉及同步。
- 外部事件（external events）：外围设备连接至具有中断功能的 MCU 的 IO 引脚，触发中断时可能生成事件。例如按键，射频芯片或脉冲探测加速器都是可以产生中断的装置，可以生成此类事件。进程可以等待到这类事件后相应地响应。
- 内部事件（internal events）：任何进程都有可以为自身或其他进程指定事件。这对进程间通讯很有用，例如通知某个进程，数据已经准备好可以进行计算。

对事件的操作被称为投递(posted)，一个中断服务程序将投递一个事件至一个进程当它被执行时。事件具有以下信息：

- process：进程被事件寻址，它可以是特定的进程或所有注册进程。
- event type：事件类型。用户可以为进程定义一些事件类型用来区分它们，比如一个类型为收到数据包，另一个为发数据包。
- data：一些数据可以同事件一起提供给进程。

Contiki 操作系统主要的理念是：事件被投递给进程，进程触发后开始执行直到阻塞，然后等待下一个事件。

- **uIP TCP/IP stack**

Contiki 包括了一个轻量级的 TCP/IP 协议栈叫做 uIP。它实现了 RFC-定义的 IPv4，IPv6，TCP 和 UDP(后两个兼容 IPv4 和 IPv6)。uIP 很高效，只实现了协议要求的特性。例如整个协议栈只有一个 buffer，用于接收和发送数据报。

1）Application API

这里有两种方法使用 uIP 协议栈写程序：

- raw API：uIP raw API 很适合实现一个简单的应用，例如一个简单的'echo' server 可以监听一些 TCP 端口并且将它收到的每个数据发送回去。然而当实现一个全功能的程序时编码变得越来越复杂，或者当多个程序需要共同使用时。甚至 TCP 连接状态机都有些烦人。
- protosocket API：protosocket library 利用 protothread library，提供一个更灵活的方式编写 TCP/IP 程序。这个库提供了一个类似于标准 BSD sockets 的接口，并允许在一个进程中编写程序。

2）Lower Layers

拥有一个有效的 TCP/IP 协议栈并且某些程序可以运行在其上当然很好，但是还不够。uIP 协议栈要求一个更低的层(根据 OSI 模型)为了和 peers 通讯。我们将讨论两种类型的 peers。

节点(nodes)：节点间通信是由无线连接实现的。uIP 协议栈需要能够发送和接受数据包。取决于 uIP 版本，Contiki 遵循不同的方法：

当使用 IPv6，Contiki 将遵循 route-over 配置。因此，uIP6 使用一个被称为 sicslowmac 的简单 MAC 层。除了头部压缩由 6loWPAN 模型提供，它仅通过射频转发数据包。

然而，对于 IPv4，Contiki 选用 mesh-under 配置。这由 Rime 通讯协议栈完成。Rime 提供 mesh routing 和路由发现，因此 uIP 使用它来转发网络上的数据包。从 IP point 的角度来看，所有的传感器网络节点组成了一个本地子网，虽然 multiple radio hops 可能被要求。

网关(gateways)：与 WSN 以外的网络通信，必须通过网关。网关连接 WSN 和另一个网络。一般接入的是 PC，也可以是其他嵌入式系统。PC 和 mote 是串口线连接的。IP 报文通过 SLIP 协议在两者之间发送。在计算机端，必须运行一个程序作为串口线和网络接口的接口。根据 uIP 协议栈版本，节点的作用不同。

使用 uIPv6 时，有一个节点将被装载一个非常简单的程序，转发从射频收到的所有数据包至串口线，或者反向发送，它不做任何的地址比对，没有 IP 协议栈运行在它上面，除了头部压缩/解压机制（6loWPAN）。这个节点被 PC 端仅视为一个以太网接口，由 PC 完成所有工作。

使用 uIPv4 时则完全不同，节点作为网关连接至 PC，有完整的 IP 协议栈在节点上运行。每当它收到数据包要发送，节点将检查报文的 IP 地址：如果它属于 WSN 子网范围，它将使用射频发送，否则它将使用串口线发送至 PC。由 PC 运行一个程序来创建 IP 层网络接口。

4.4 Contiki 源代码结构

Contiki 是一个高度可移植的操作系统，它的设计就是为了获得良好的可移植性，因此源代码的组织很有特点。本文为大家简单介绍 Contiki 的源代码组织结构以及各部分代码的作用。

Contiki 源文件目录可以在 Contiki Studio 安装目录中的 workspace 目录下找到。打开 Contiki 源文件目录，可以看到主要有 apps、core、cpu、doc、 examples、platform、tools 等目录。下面将分别对各个目录进行介绍。

core

core 目录下是 Contiki 的核心源代码，包括网络（net）、文件系统（cfs）、外部设备（dev）、链接库（lib）等等，并且包含了时钟、I/O、ELF 装载器、网络驱动等的抽象。

cpu

cpu 目录下是 Contiki 目前支持的微处理器，例如 arm、avr、msp430 等等。如果需要支持新的微处理器，可以在这里添加相应的源代码。

platform

platform 目录下是 Contiki 支持的硬件平台，例如 mx231cc、micaz、sky、win32、mb851 等等。Contiki 的平台移植主要在这个目录下完成。这一部分的代码与相应的硬件平台相关。

apps

apps 目录下是一些应用程序，例如 ftp、shell、webserver 等等，在项目程序开发过程中可以直接使用。使用这些应用程序的方式为， 在项目的 Makefile 中，定义 APPS = [应用程序名称]。在以后的示例中会具体看到如何使用 apps。

examples

examples 目录下是针对不同平台的示例程序。Smeshlink 的示例程序也在其中。

doc

doc 目录是 Contiki 帮助文档目录，对 Contiki 应用程序开发很有参考价值。使用前需要先用 Doxygen 进行编译。

tools

tools 目录下是开发过程中常用的一些工具，例如 CFS 相关的 makefsdata、网络相关的 tunslip、模拟器 cooja 和 mspsim 等。

为了获得良好的可移植性，除了 cpu 和 platform 中的源代码与硬件平台相关以外，其他目录中的源代码都尽可能与硬件无关。编译时，根据指定的平台来链接对应的代码。

4.5 Contiki 程序实例

- **源码**

这里以 contiki 系统 examples 目录下的 hello-world 例程为例（用于向串口打印"Hello World"）。

```
#include "contiki.h"

#include <stdio.h> /* For printf() */
/*---------------------------------------------------------------------------*/
PROCESS(hello_world_process, "Hello world process");/*定义进程名称和对应处理函数*/
AUTOSTART_PROCESSES(&hello_world_process);/*声明此进程为自启动方式加入系统*/
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(hello_world_process, ev, data)/*进程处理器函数*/
{
  PROCESS_BEGIN();/*启动进程，系统要求必须调用*/
  while(1){
  printf("Hello, world\r\n");/*进程执行语句，循环打印*/
   }
  PROCESS_END();/*结束进程，系统要求必须调用*/
}
/*---------------------------------------------------------------------------*/
```

在 contiki 系统中，所有进程都是以上述模版进行定义和实现的，我们以 hello-world 例程为例进行分析。

- **主要宏及函数分析**

contiki 程序有着非常规范的程序步骤，PROCESS 宏是用来声明一个进程；AUTOSTART 宏是使这个进程开机自启动，PROCESS_THREAD 里面定义的是程序的主体，并且主体内部要以 PROCESS_BEGIN()开头，以 PROCESS_END()来结束。

1）PROCESS 宏

PROCESS 宏完成两个功能：

声明一个函数，该函数是进程的执行体，即进程的 thread 函数指针所指的函数。

定义一个进程。

源码展开如下：

```
//PROCESS(hello_world_process, "Hello world");
#define PROCESS(name, strname) PROCESS_THREAD(name, ev, data); \
structprocessname={NULL,strname, process_thread_##name }
```

对应参数展开为：

```
#define PROCESS((hello_world_process, "Hello world")
PROCESS_THREAD(hello_world_process, ev, data); \
structprocesshello_world_process = { NULL, "Hello world", process_thread_hello_world_process };
```

a）PROCESS_THREAD 宏

PROCESS_THREAD 宏用于定义进程的执行主体，宏展开如下

```
#define PROCESS_THREAD(name, ev, data) \
staticPT_THREAD(process_thread_##name(struct pt *process_pt, process_event_t ev, process_data_t data))
```

对应参数展开为：

```
//PROCESS_THREAD(hello_world_process, ev, data);
staticPT_THREAD(process_thread_hello_world_process(struct pt *process_pt, process_event_t ev, process_data_t data));
```

PT_THREAD 宏

PT_THREAD 宏用于声明一个 protothread，即进程的执行主体，宏展开如下：

```
#define PT_THREAD(name_args) char name_args
```

展开之后即为：

```
//static PT_THREAD(process_thread_hello_world_process(struct pt *process_pt, process_event_t ev, process_data_t data));
staticcharprocess_thread_hello_world_process(struct pt *process_pt, process_event_t ev, process_data_t data);
```

另外，struct pt *process_pt 可以直接理解成 lc，用于保存当前被中断的地方（保存程序断点），以便下次恢复执行。

b）定义一个进程

PROCESS 宏展开的第二句，定义一个进程 hello_world_process，源码如下：

```
struct process hello_world_process = { NULL, "Hello world", process_thread_hello_world_process };
```

结构体 process 定义如下:

```
struct process
{
structprocess*next;
constchar*name; /*此处略作简化，源代码包含了预编译#if。即可以通过配置，使得进程名称可有可无*/
PT_THREAD((* thread)(struct pt *, process_event_t, process_data_t));
structpt pt;
unsigned char state, needspoll;
};
```

可见进程 hello_world_process 的 lc、state、needspoll 都默认置为 0。

2）AUTOSTART_PROCESSES 宏

AUTOSTART_PROCESSES 宏实际上是定义一个指针数组，存放 Contiki 系统运行时需自动启动的进程，宏展开如下：

```
//AUTOSTART_PROCESSES(&hello_world_process);
#define AUTOSTART_PROCESSES(...) \ struct process * const autostart_processes[] = {__VA_ARGS__, NULL}
```

这里用到 C99 支持可变参数宏的特性，如：#define debug(…) printf(__VA_ARGS__) ，

缺省号代表一个可以变化的参数表，宏展开时，实际的参数就传递给 printf()了。例：debug("Y = %d\n", y); 被替换成 printf("Y = %d\n", y); 。那么，AUTOSTART_PROCESSES(&hello_world_process); 实际上被替换成：

struct process * const autostart_processes[] = {&hello_world_process, NULL};

这样就知道如何让多个进程自启动了，直接在宏 AUTOSTART_PROCESSES()加入需自启动的进程地址，比如让 hello_process 和 world_process 这两个进程自启动，如下：

AUTOSTART_PROCESSES(&hello_process，&world_process);

最后一个进程指针设成 NULL，则是一种编程技巧，设置一个哨兵(提高算法效率的一个手段)，以提高遍历整个数组的效率。

3）PROCESS_THREAD 宏

PROCESS(hello_world_process, "Hello world"); 展开成两句，其中有一句是也是 PROCESS_THREAD(hello_world_process, ev, data) ;。这里要注意到分号，是一个函数声明。而这 PROCESS_THREAD(hello_world_process, ev, data)没有分号，而是紧跟着"{}"，是上述声明函数的实现。关于 PROCESS_THREAD 宏的分析，最后展开如下。

```
static char process_thread_hello_world_process(struct pt *process_pt, process_event_t ev, process_data_t data);
```

提示：在阅读 Contiki 源码，手动展开宏时，要特别注意分号。

4）PROCESS_BEGIN 宏和 PROCESS_END 宏

原则上，所有代码都得放在 PROCESS_BEGIN 宏和 PROCESS_END 宏之间(如果程序全部使用静态局部变量，这样做总是对的。倘若使用局部变量，情况就比较复杂了，当然，不建议这样做)，看完下面宏展开，就知道为什么了。

a）PROCESS_BEGIN 宏

PROCESS_BEGIN 宏一步步展开如下：

```
#define PROCESS_BEGIN() PT_BEGIN(process_pt)
```

process_pt 是 struct pt*类型，在函数头传递过来的参数，直接理解成 lc，用于保存当前被中断的地方，以便下次恢复执行。继续展开：

```
#define PT_BEGIN(pt) { char PT_YIELD_FLAG = 1; LC_RESUME((pt)->lc)
#define LC_RESUME(s) switch(s) { case 0:
```

把参数替换，结果如下：

```
{

charPT_YIELD_FLAG = 1; /*将 PT_YIELD_FLAG 置 1，类似于关中断？？？*/

switch(process_pt->lc) /*程序根据 lc 的值进行跳转，lc 用于保存程序断点*/

{
```

```
case0: /*第一次执行从这里开始，可以放一些初始化的东东*/

;
```

b） PROCESS_END 宏

PROCESS_END 宏一步步展开如下：

```
#define PROCESS_END() PT_END(process_pt)

#define PT_END(pt) LC_END((pt)->lc); PT_YIELD_FLAG = 0; \ PT_INIT(pt); return PT_ENDED; }


#define LC_END(s) }

#define PT_INIT(pt) LC_INIT((pt)->lc)
#define LC_INIT(s) s = 0;

#define PT_ENDED 3
```

整理下，实际上如下代码：

```
  }
PT_YIELD_FLAG = 0;

(process_pt)->pt = 0;

  return3;
}
```

♦ **main()函数——进程执行**

针对本实验平台，main 函数在 platform/mb851 的 contiki-main.c 文件，主要完成硬件、时钟、进程底层协议等的初始化。要执行 hello_world.c 的 process，主要是用到 main 函数的这个语句：

```
autostart_start(autostart_processes);
```

autostart_start()是一个函数，定义在 autostart_start.c 文件里面，函数定义如下：

```
void
autostart_start(struct process * const processes[])
{
   int i;

   for(i = 0; processes[i] != NULL; ++i) {
      process_start(processes[i], NULL);
      PRINTF("autostart_start: starting process '%s'\n", processes[i]->name);
   }
```

```
}
```

struct process * const process[]维护的是一个进程队列，然后轮流执行，此例中只有一个 hello_world 进程。要启动一个进程，必须要用到 process_start 函数，此函数的定义如下：

```
void
process_start(struct process *p, const char *arg)//可以传递 arg 给进程 p，也可以不传，直接“NULL”
{
  struct process *q;

  /* First make sure that we don't try to start a process that is
     already running. *//*参数验证：确保进程不在进程链表中*/

  for(q = process_list; q != p && q != NULL; q = q->next);

  /* If we found the process on the process list, we bail out. */
  if(q == p) {
    return;
  }
  /* Put on the procs list.*//*把进程加到进程链表首部*/
  p->next = process_list;
  process_list = p;
  p->state = PROCESS_STATE_RUNNING;
  PT_INIT(&p->pt);//将 p->pt->lc 设为 0，使得进程从 case 0 开始执行

  PRINTF("process: starting '%s'\n", PROCESS_NAME_STRING(p));

  /* Post a synchronous initialization event to the process. */
  process_post_synch(p, PROCESS_EVENT_INIT, (process_data_t)arg);// 给进程传递一个
PROCESS_EVENT_INIT 事件，让其开始执行
}
```

process_start 所干的工作就是先把将要执行的进程加入到进程队列 process_list 的首部，如果这个进程已经在 process_list 中，就 return；接下来就把 state 设置为 PROCESS_STATE_RUNNING 并且初始化 pt。最后通过函数 process_post_synch()执行这个进程，并给这个进程传递一个 PROCESS_EVENT_INIT（事件），看看 process_post_synch()这个函数的定义：

```
void
process_post_synch(struct process *p, process_event_t ev, process_data_t data)//process_post_synch()直接调
用 call_process()，期间需要保存 process_corrent，这是因为当调用 call_process 执行这个进程 p 时，
process_current 就会指向当前进程 p，而进程 p 可能会退出或者被挂起等待一个事件
{
  struct process *caller = process_current;//相当于 PUSH，保存现场 process_current
  call_process(p, ev, data);
  process_current = caller;process_current = caller; //相当于 POP，恢复现场 process_current
}
```

为什么 process_post_synch()中要把 process_current 保存起来呢，process_current 指向的是一个正在运行的 process，当调用 call_process 执行这个 hello_world 这个进程时，process_current 就会指向当前的 process 也就是 hello_world 这个进程，而 hello_world 这个进程它可能会退出或者正在被挂起等待一个事件，这时 process_current = caller 语句正是要恢复先前的那个正在运行的 process。

接下来展开 call_process()，开始真正的执行这个 process 了：

```
static void
call_process(struct process *p, process_event_t ev, process_data_t data)//如果进程 process 的状态为 PROCESS_STATE_RUNNING，并且进程中的 thread 函数指针(相当于该进程的主函数)不为空的话，就执行该进程。如果返回值表示退出、结尾或者遇到 PROCESS_EVENT_EXIT，进程退出，否则进程被挂起，等待事件。
{
  int ret;
#if DEBUG
  if(p->state == PROCESS_STATE_CALLED) {
    printf("process: process '%s' called again with event %d\n", PROCESS_NAME_STRING(p), ev);
  }
#endif /* DEBUG */

  if((p->state & PROCESS_STATE_RUNNING) &&
     p->thread != NULL) {////thread 是函数指针
    PRINTF("process: calling process '%s' with event %d\n", PROCESS_NAME_STRING(p), ev);
    process_current = p;
    p->state = PROCESS_STATE_CALLED;
    ret = p->thread(&p->pt, ev, data);//才真正执行 PROCESS_THREAD(name, ev, data)定义的内容
    if(ret == PT_EXITED ||
       ret == PT_ENDED ||
       ev == PROCESS_EVENT_EXIT) {//// 如果返回值表示退出、结尾或者遇到 PROCESS_EVENT_EXIT，进程退出
      exit_process(p, p);
    } else {
      p->state = PROCESS_STATE_RUNNING;//进程挂起等待事件
    }
  }
}
```

这个函数中才是真正的执行了 hello_world_process 的内容。假如进程 process 的状态是 PROCESS_STATE_RUNNING 以及进程中的 thread 函数不为空的话，就执行这个进程：

首先把 process_current 指向 p，接着把 process 的 state 改为 PROCESS_STATE_CALLED，执行 hello_world 这个进程的 body 也就是函数 p->thread，并将返回值保存在 ret 中，如果返回值表示退出或者遇到了 PROCESS_EVENT_EXIT 的时事件后，便执行 exit_process()函数，process 退出。不然程序就应该在挂起等待事件的状态，那么就继续把 p 的状态维持为 PROCESS_STATE_RUNNING。

5. 实验步骤

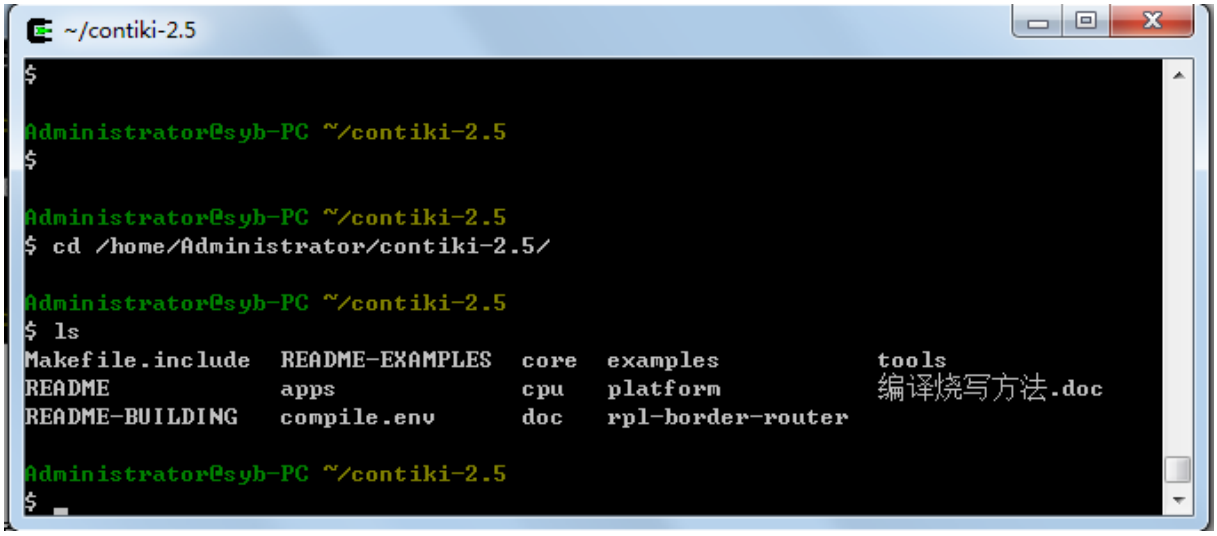
1）安装 contiki 系统源码

将产品光盘配套的 contiki 系统源码目录 contiki-2.5 拷贝至 Cygwin 开发环境安装目录的\home\Administrator 目录下。关于 Cygwin 开发环境的安装，请查看前面章节内容(1.2.3 IPv6 模块)部分，这里不再赘述。

如果 Cygwin 环境 home 目录下没有 Administrator 目录，说明您还未登陆过 Cygwin 环境，需要登陆后才可自动创建 Administrator 用户目录。

2）登陆 Cygwin

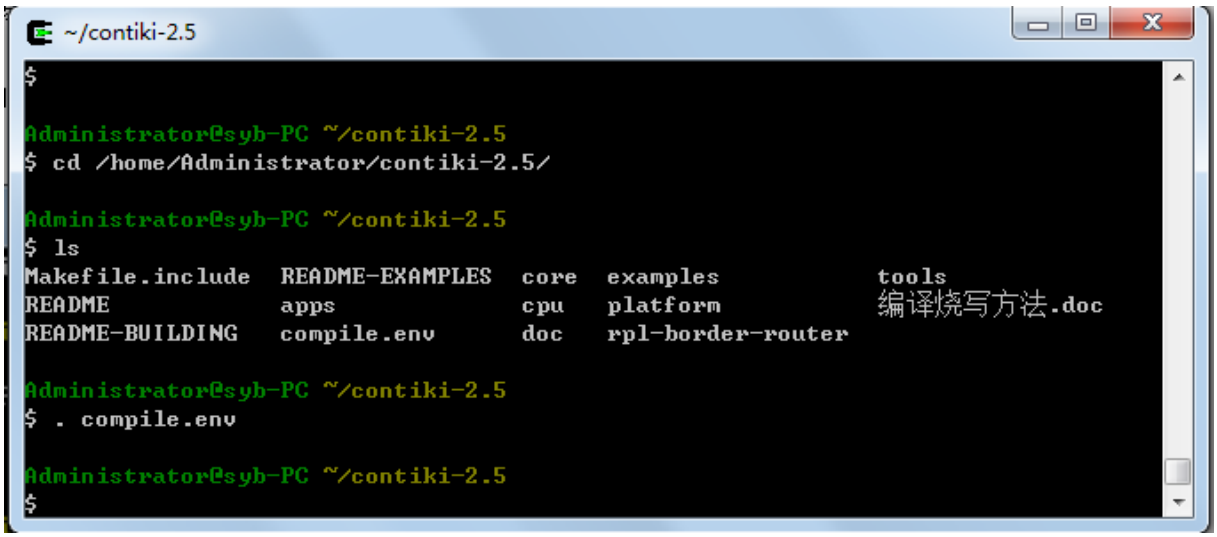
打开 Cygwin 开发环境，登陆进去。进入刚刚拷贝的 contiki-2.5 目录下，如图所示：



3）设置 IAR 编译器环境变量

Cygwin 开发环境使用前面安装的 IAR EWARM 环境编译工具，因此要在此环境中加入 IAR 工具的安装路径，才可以使用相关编译工具对源码工程进行编译和下载。

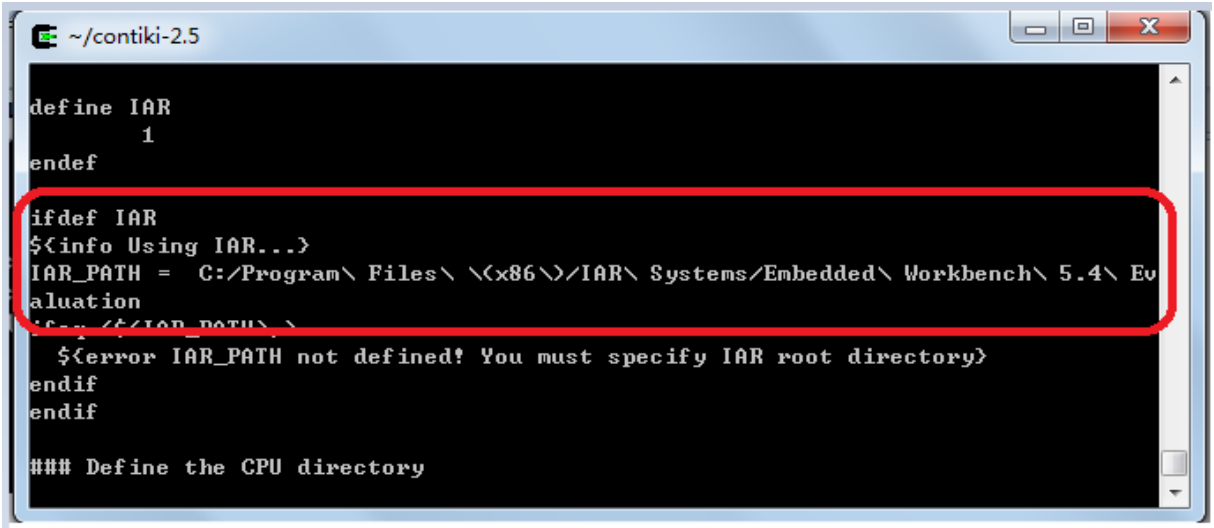
执行. compile.env 命令（注意 .和 compile.env 中间有个空格）即可完成环境的设置。



其中 compile.env 文件的内容为：export PATH=/cygdrive/c/Program\ Files/IAR\

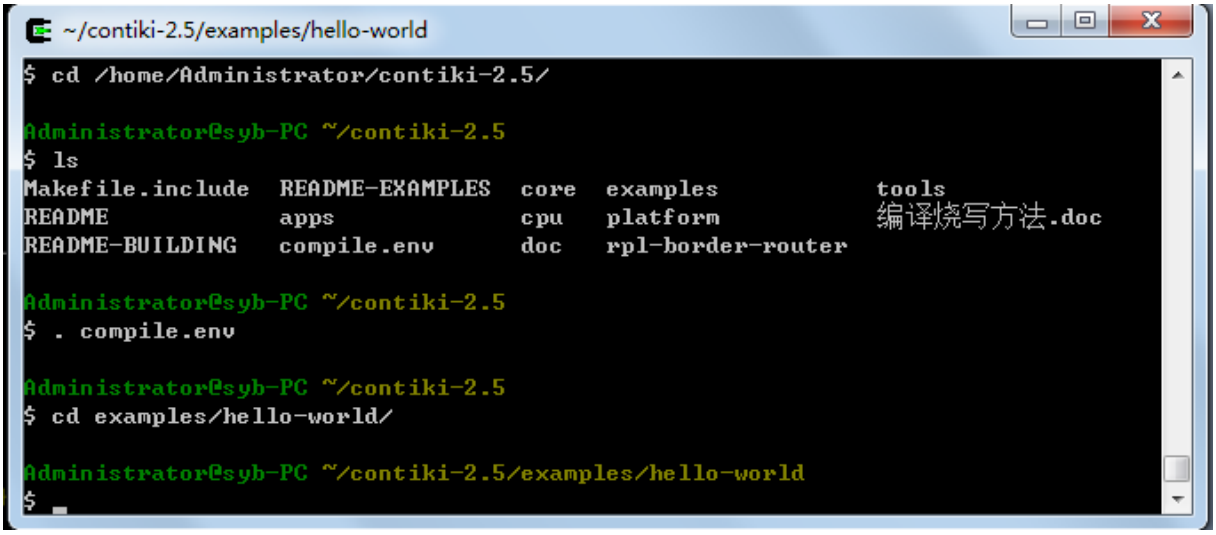
Systems/Embedded\ Workbench\ 5.4\ Evaluation/arm/bin:$PATH（其中/cygdrive/引用 windows 各个盘的路径）。注意：具体变量用户可以根据自己环境中的 IAR 安装路径进行修改。

使用 vi 编辑 cpu/stm32w108/Makefile.stm32w108 文件，根据 IAR 实际的安装路径进行修改，如下：

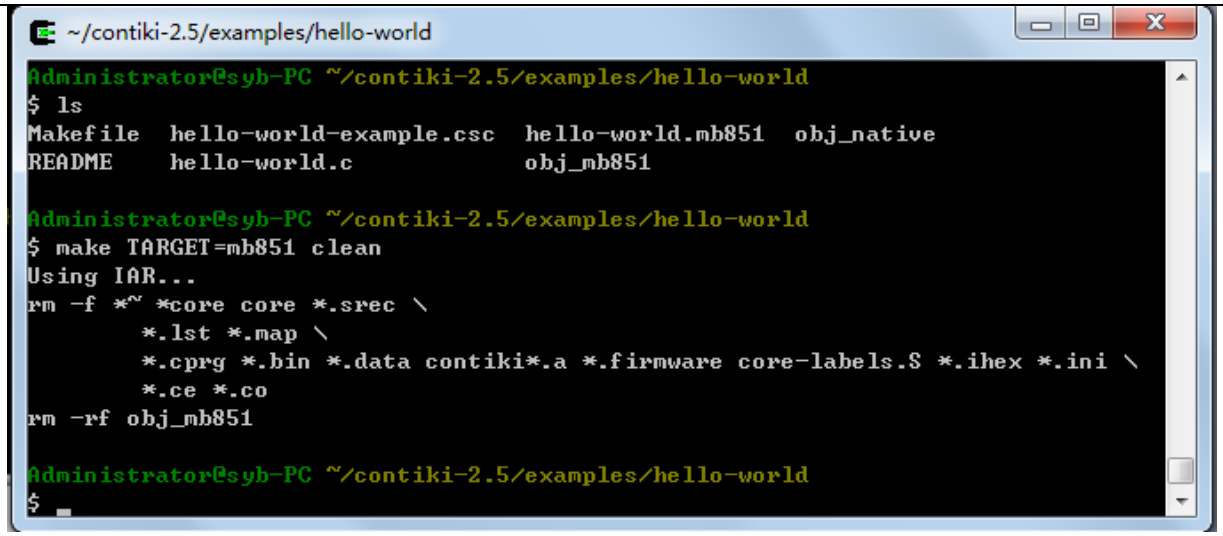


4）编译 hello-world 程序

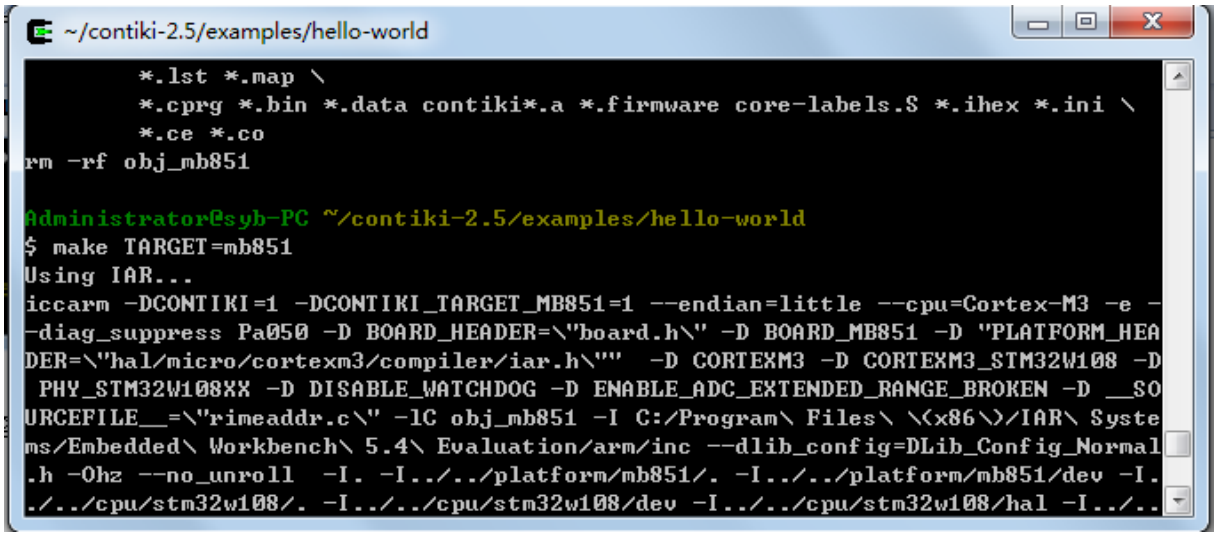
进入 contiki-2.5 目录下的 examples/hello-world 实验目录下：



执行 make TARGET=mb851 clean （TARGET=mb851 指定针对的平台），清除工程中间文件：



执行 make TARGET=mb851 命令进行编译：



注意，一般编译前，需要 clean 以下工程，使用 make TARGET=mb851 clean 命令。

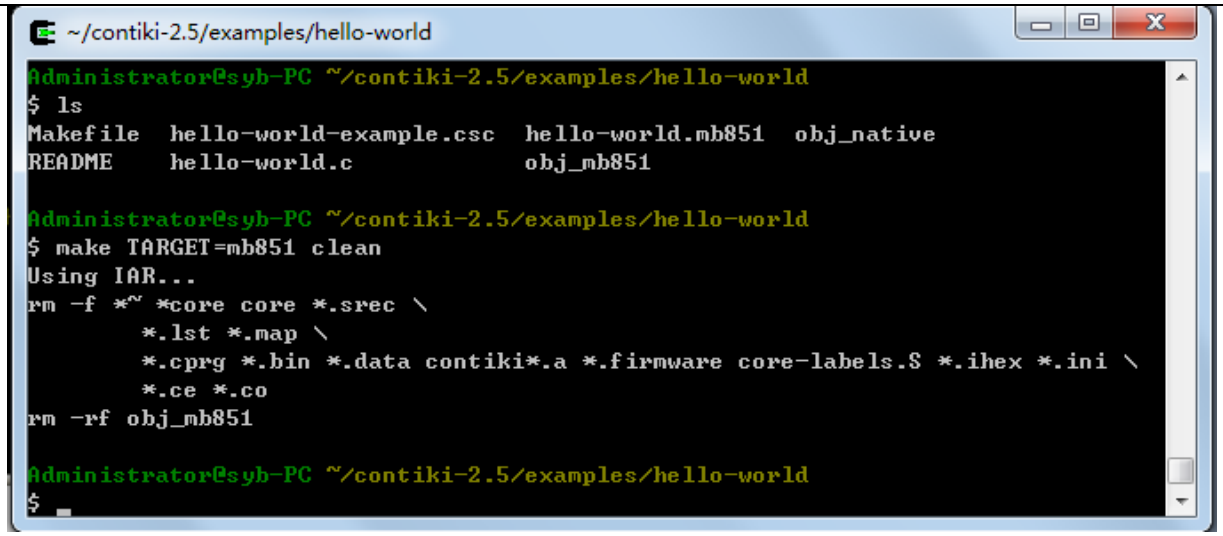
若编程出错，可执行如下命令删除中间文件后，重新编译。

```
rm *.d
```

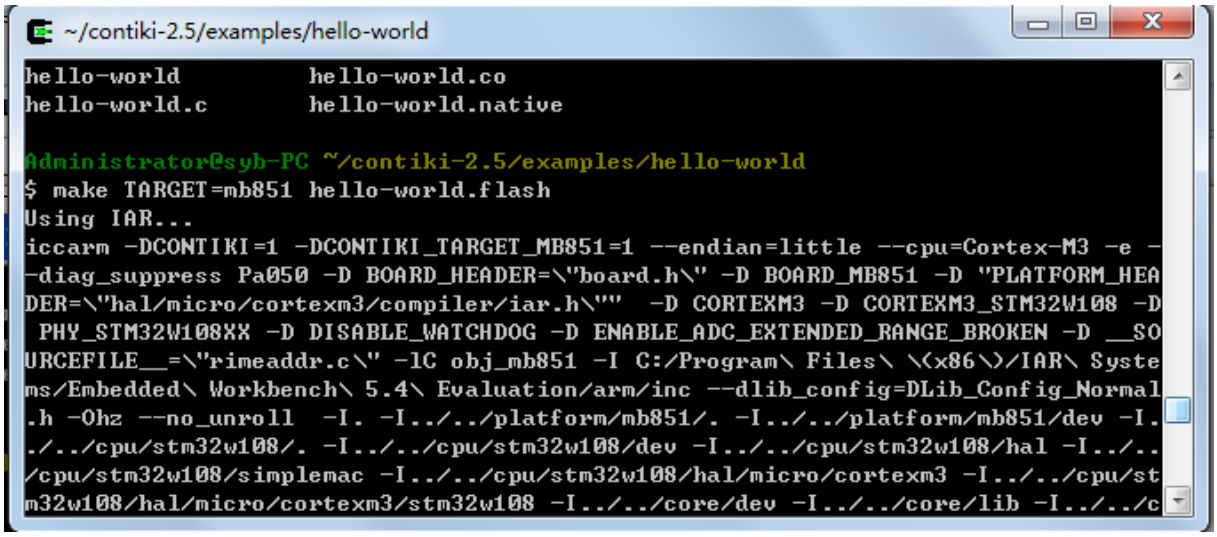
5）下载程序，运行测试

开启实验设备电源，使用 J-OB 仿真器和 J-OB 转接板连接 IPv6 模块，通过平台的“+”“-”按键选择目标模块，建议选择平台上 IPv6 根节点进行编程实验，因为其可以使用平台主板上的 RS232 串口。之后即可在 Cygwin 环境下烧写下载程序。

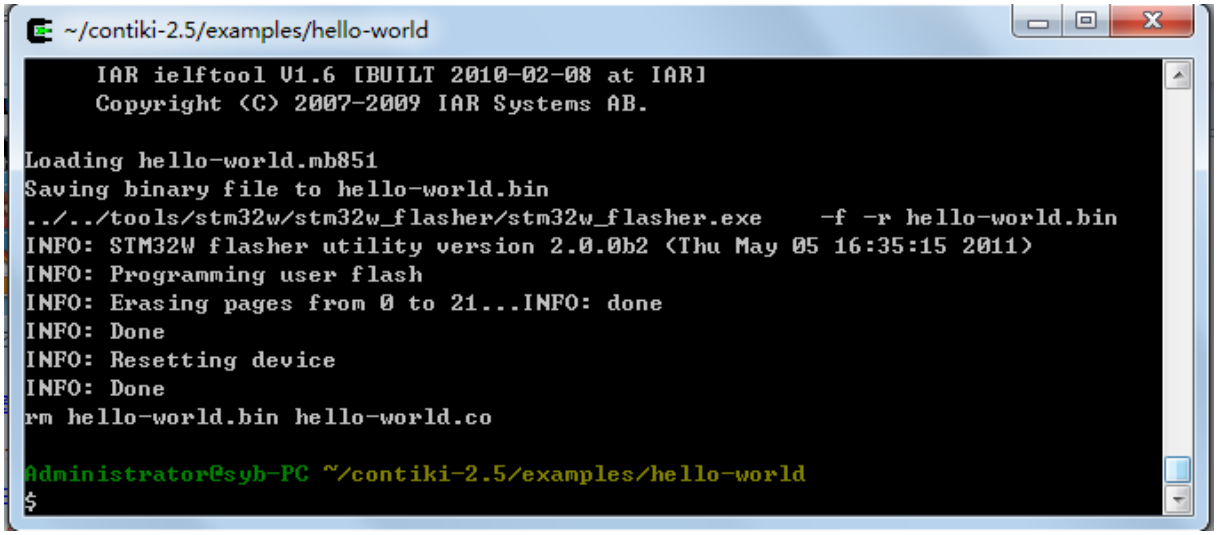
烧写之前，要先 clean：



然后运行 make TARGET=mb851 hello-world.flash 命令进行烧写：



烧写完成将提示如下信息：



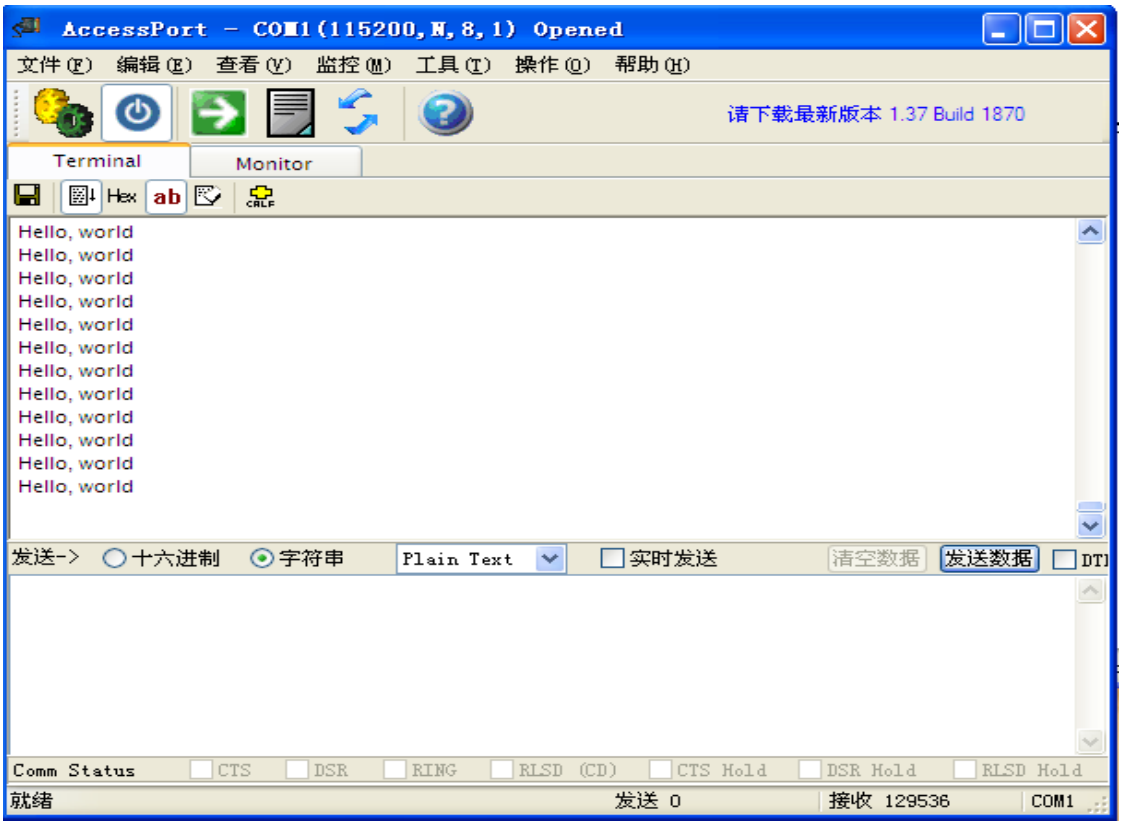
上述烧写过程将自动调用 IAR 相关工具完成。

6）通过串口查看打印信息

使用串口连接 IPv6 模块，也可以使用平台上 IPv6 根节点的串口拨码跳线(0001)连接底

板的 Debug UART 串口。

在电脑端打开串口终端软件，正确设置，波特率 115200,无校验，8 数据位，1 位停止位，无硬件流，即可查看到 IPv6 模块运行打印的信息：



实验二. 基于 IPv6 模块的进程间交互的实验

1. 实验目的

- 了解 Cygwin 开发环境及 Contiki 系统相关内容。
- 了解 Contiki 系统进程间交互的理论。
- 掌握 IPv6 模块的编程及下载使用方法。

2. 实验环境

- 硬件：ICS-IOT-CEP 教学实验平台，PC 机，J-OB 仿真器和 J-OB 转接板。
- 软件：Cygwin +IAR EWARM，contiki OS，串口终端。

3. 实验内容

- 在 helloword 例程的基础上，实现两个进程分别打印 hello 和 word，理解进程间交互实现机制；
- 使用 Cygwin 开发环境，利用 IAR 编译工具对 contiki 系统进行编译，下载程序到 IPv6 模块运行测试。

4. 实验原理

4.6 Contiki 中事件驱动和 protothread 机制

Contiki 的两个主要机制：事件驱动和 protothread 机制，前者是为了降低功耗，后者是为了节省内存。

- **事件驱动**

嵌入式系统常常被设计成响应周围环境的变化，而这些变化可以看成一个个事件。事件来了，操作系统处理之，没有事件到来，就跑去休眠了(降低功耗)，这就是所谓的事件驱动，类似于中断。在前一章中已经介绍了，这里只是简单介绍。

1）事件结构体

事件也是 Contiki 重要的数据结构，其定义如下：

```
struct event_data
{
process_event_t ev;
process_data_t data;
```

```
structprocess*p;
};
typedef unsigned char process_event_t;
typedef void * process_data_t;
```

各成员变量含义如下：

ev-----标识所产生事件

data---保存事件产生时获得的相关信息，即事件产生后可以给进程传递的数据

p------指向监听该事件的进程

2）事件分类

事件可以被分为三类：时钟事件(timer events)、外部事件、内部事件。那么，Contiki 核心数据结构就只有进程和事件了，把 etimer 理解成一种特殊的事件。

3）事件队列

Contiki 用环形队列组织所有事件(用数组存储)，如下：

```
static struct event_data events[PROCESS_CONF_NUMEVENTS];
```

图示事件队列如下：

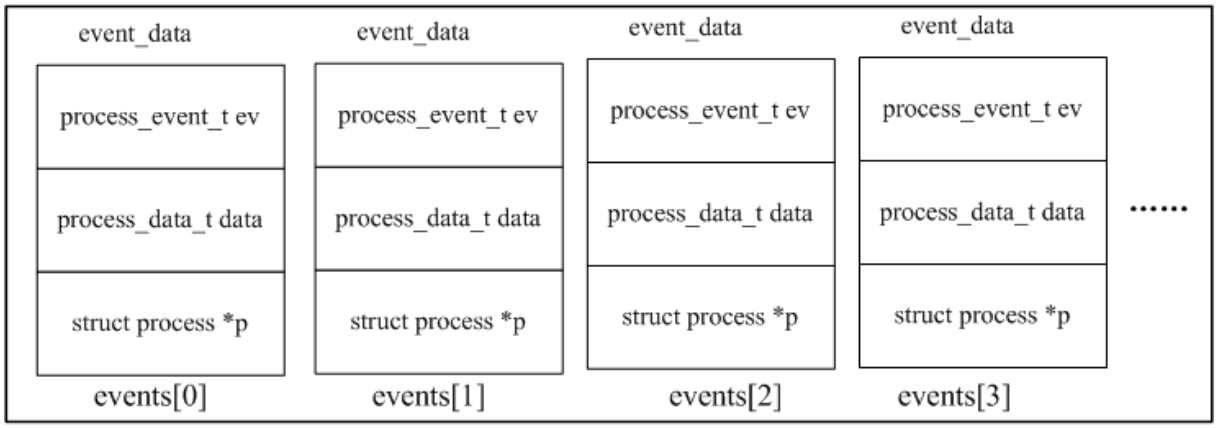


图 events图示

4）系统定义的事件

系统定义了 10 个事件，源码和注释如下：

```
/*配置系统最大事件数*/
#ifndef PROCESS_CONF_NUMEVENTS
  #define PROCESS_CONF_NUMEVENTS 32
#endif

#define PROCESS_EVENT_NONE 0x80 // 函数 dhcpc_request 调用 handle_dhcp(PROCESS_EVENT_NONE,NULL)
#define PROCESS_EVENT_INIT 0x81 //启动一个进程 process_start，通过传递该事件
#define PROCESS_EVENT_POLL 0x82 //在 PROCESS_THREAD(etimer_process, ev, data)使用到
#define PROCESS_EVENT_EXIT 0x83 //进程退出，传递该事件给进程主体函数 thread
```

```
#define PROCESS_EVENT_SERVICE_REMOVED 0x84
#define PROCESS_EVENT_CONTINUE 0x85 //PROCESS_PAUSE 宏用到这个事件
#define PROCESS_EVENT_MSG 0x86
#define PROCESS_EVENT_EXITED 0x87 //进程退出，传递该事件给其他进程
#define PROCESS_EVENT_TIMER 0x88 //etimer 到期时，传递该事件
#define PROCESS_EVENT_COM 0x89
#define PROCESS_EVENT_MAX 0x8a /*进程初始化时，让 lastevent=PROCESS_EVENT_MAX，即新产生的事件从 0x8b 开始，函数 process_alloc_event 用于分配一个新的事件*/
```

注：PROCESS_EVENT_EXIT 与 PROCESS_EVENT_EXITED 区别

事件 PROCESS_EVENT_EXIT 用于传递给进程的主体函数 thread，如在 exit_process 函数中的 p->thread(&p->pt, PROCESS_EVENT_EXIT, NULL)。而 PROCESS_EVENT_EXITED 用于传递给进程，如 call_process(q, PROCESS_EVENT_EXITED, (process_data_t)p)。(助记：EXITED 是完成式，发给进程，让整个进程结束。而 g 一般式 EXIT，发给进程主体 thread，只是使其退出 thread)

5）一个特殊事件

如果事件结构体 event_data 的成员变量 p 指向 PROCESS_BROADCAST，则该事件是一个广播事件。在 do_event 函数中，若事件的 p 指向的是 PROCESS_BROADCAST，则让进程链表 process_list 所有进程投入运行。部分源码如下：

```
#define PROCESS_BROADCAST NULL //广播进程

/*保存待处理事件的成员变量*/
ev = events[fevent].ev;
data = events[fevent].data;
receiver=events[fevent].p;

if(receiver==PROCESS_BROADCAST)
{
for(p = process_list; p != NULL; p = p->next)
{
if(poll_requested)
{
do_poll();
}
call_process(p, ev, data);
}
}
```

4.7 Contiki 中进程概念

- **进程结构体**

进程结构体源码如下：

```
struct process
{
structprocess*next;//指向下一个进程

/******进程名称**********/
#if PROCESS_CONF_NO_PROCESS_NAMES
#define PROCESS_NAME_STRING(process) ""
#else
constchar*name;
#define PROCESS_NAME_STRING(process) (process)->name
#endif

PT_THREAD((*thread)(struct pt *, process_event_t, process_data_t)); //
structpt pt; //
unsigned char state; //
unsigned char needspoll; //
};
```

1）进程名称

运用 C 语言预编译指令，可以配置进程名称，宏 PROCESS_NAME_STRING(process)用于返回进程 process 名称，若系统无配置进程名称，则返回空字符串。在以后讨论中，均假设配有进程名称。

2）PT_THREAD 宏

PT_THREAD 宏定义如下：

```
#define PT_THREAD(name_args) char name_args
```

故“PT_THREAD((*thread)(struct pt *, process_event_t, process_data_t)); ”语句展开如下：

```
char (*thread)(struct pt *, process_event_t, process_data_t);
```

声明一个函数指针 thread，指向的是一个含有 3 个参数，返回值为 char 类型的函数。这是进程的主体，当进程执行时，主要是执行这个函数的内容。另，声明一个进程包含在宏 PROCESS(name, strname) 里，通过宏 AUTOSTART_PROCESSES(...) 将进程加入自启动数组中。

3）pt

pt 结构体一步步展开如下：

```
struct pt
{
lc_t lc;
};

typedef unsigned short lc_t;
```

如此，可以把 struct pt pt 直接理解成 unsigned short lc，以后如无特殊说明，pt 就直接理

解成 lc。lc(local continuations)用于保存程序被中断的行数(只需两个字节，这恰是 protothread 轻量级的集中体现)，被中断的地方，保存行数(s=__LINE__)接着是语句 case __LINE__。当该进程再次被调度时，从 PROCESS_BEGIN()开始执行，而该宏展开含有这条语句 switch(process_pt->pt)，从而跳到上一次被中断的地方(即 case __LINE__)，继续执行。

4）进程状态

进程共 3 个状态，宏定义如下：

```
#define PROCESS_STATE_NONE 0 /*类似于 Linux 系统的僵尸状态，进程已退出，只是还没从进程链表删除*/
#define PROCESS_STATE_RUNNING 1 /*进程正在执行*/
#define PROCESS_STATE_CALLED 2 /*实际上是返回，并保存 lc 值*/
```

5）needspoll

简而言之，needspoll 为 1 的进程有更高的优先级。具体表现为，当系统调用 process_run()函数时，把所有 needspoll 标志为 1 的进程投入运行，而后才从事件队列取出下一个事件传递给相应的监听进程。

与 needspoll 相关的另一个变量 poll_requested，用于标识系统是否存在高优先级进程，即标记系统是否有进程的 needspoll 为 1。

```
static volatile unsigned char poll_requested;
```

- **进程链表**

基于上述分析，将代码展开或简化，得到如下进程链表 process_list：

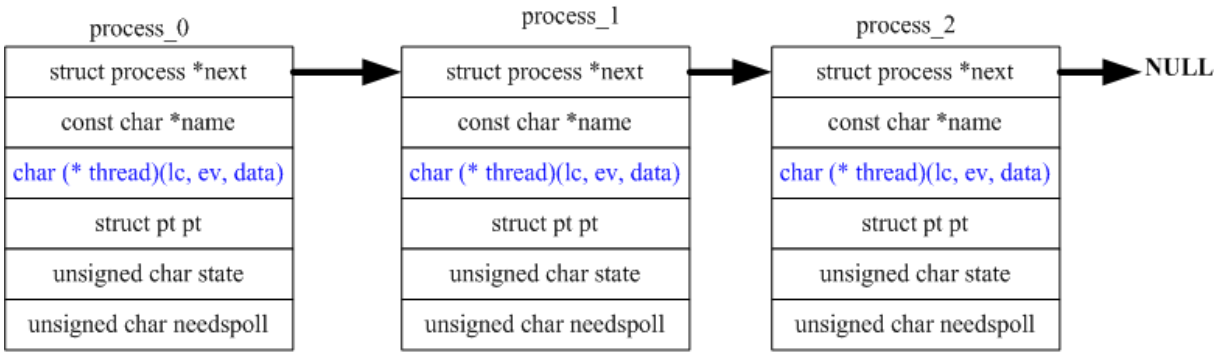


图 Contiki进程链表 (注：为直观显示，将部分宏展开)

- **创建进程及启动进程**

创建进程主要由 PROCESS 宏(声明进程)和 PROCESS_THREAD 宏(定义进程执行主体)完成，详情前一章 helloword 分析，启动进程由 process_start 函数完成，总是把进程加入到进程链表的头部。

4.8 进程、事件、etimer 三者关系

进程 process、事件 event_data、etimer 都是 Contiki 的核心数据结构，理清这三者关系，

将有助于对系统的理解。

- **事件与 etimer 关系**

事件即可以分为同步事件、异步事件，也可以分为定时器事件、内部事件、外部事件。而 etimer 属于定时器事件的一种，可以理解成 Contiki 系统把 etimer 单列出来，方便管理(由 etimer_process 系统进程管理)。

当 etimer_process 执行时，会遍历 etimer 链表，检查 etimer 是否有到期的，凡有 timer 到期就把事件 PROCESS_EVENT_TIMER 加入到事件队列中，并将该 etimer 成员变量 p 指向 PROCESS_NONE。在这里，PROCESS_NONE 用于标识该 etimer 是否到期，函数 etimer_expired 会根据 etimer 的 p 是否指向 PROCESS_NONE 来判断该 etimer 是否到期。

- **进程与 etimer 关系**

etimer 与 process 还不是一一对应的关系，一个 etimer 必定绑定一个 process，但 process 不一定非得绑定 etimer。etimer 只是一种特殊事件罢了。

- **进程与事件关系**

当有事件传递给进程时，就新建一个事件加入事件队列，并绑定该进程，所以一个进程可以对应于多个事件(即事件队列有多个事件跟同一个进程绑定)，而一个事件可以广播给所有进程，即该事件成员变量 p 指向空。当调用 do_event 函数时，将进程链表所有进程投入运行。

4.9 源码分析

系统启动，执行一系列初始化(串口、时钟、进程等)，接着启动系统进程 etimer_process，而后启动进程 print_hello_process 和 print_world_process。

- **源码**

阅读光盘内本节实验代码，如下：

```
//filename:process-interact.c 进程间交互
#include "contiki.h"
#include <stdio.h> /* For printf() */
static process_event_t event_data_ready;

/*声明两个进程*/
PROCESS(print_hello_process, "Hello");
PROCESS(print_world_process, "world");
AUTOSTART_PROCESSES(&print_hello_process, &print_world_process);   //让该两进程自启动

/*定义进程 print_hello_process*/
PROCESS_THREAD(print_hello_process, ev, data)
```

```
{
  PROCESS_BEGIN();
  static struct etimer timer;
  etimer_set(&timer, CLOCK_CONF_SECOND / 10); //#define CLOCK_CONF_SECOND 10 将 timer 的 interval 设为 1
  printf("***print hello process start***\n");
  event_data_ready = process_alloc_event(); //return lastevent++; 新建一个事件，事实上是用一组 unsigned char 值来标识不同事件
  while (1)
  {
    PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER); //即 etimer 到期继续执行下面内容
    printf("Hello\n");
    process_post(&print_world_process, event_data_ready, NULL); // 传递异步事件给 print_world_process，直到内核调度该进程才处理该事件。
    etimer_reset(&timer); //重置定时器
  }
  PROCESS_END();
}

/*定义进程 print_world_process*/
PROCESS_THREAD(print_world_process, ev, data)
{
  PROCESS_BEGIN();
  printf("***print world process start***\n");
  while (1)
  {
    PROCESS_WAIT_EVENT_UNTIL(ev == event_data_ready);
    printf("world\n");
  }
  PROCESS_END();
}
```

进程 print_hello_process 一直执行到 PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER)，此时 etimer 还没到期，进程被挂起。转去执行 print_world_process，待执行到 PROCESS_WAIT_EVENT_UNTIL(ev == event_data_ready)被挂起(因为 print_hello_process 还没 post 事件)。而后再转去执行系统进程 etimer_process，若检测到 etimer 到期，则继续执行 print_hello_process，打印 Hello，并传递事件 event_data_ready 给 print_world_process，初始化 timer，待执行到 PROCESS_WAIT_EVENT_UNTIL(while 死循环)，再次被挂起。转去执行 print_world_process，打印 world，待执行到 PROCESS_WAIT_EVENT_UNTIL(ev == event_data_ready)又被挂起。再次执行系统进程 etimer_process，如此反复执行。

1）PROCESS_WAIT_EVENT_UNTIL 宏

```
//PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER);

/*宏 PROCESS_WAIT_EVENT_UNTIL，直到条件 c 为真时，进程才得以继续向前推进*/
#definePROCESS_WAIT_EVENT_UNTIL(c) PROCESS_YIELD_UNTIL(c)
```

```
#definePROCESS_YIELD_UNTIL(c) PT_YIELD_UNTIL(process_pt, c) //Yield the currently running process until a condition occurs

#definePT_YIELD_UNTIL(pt, cond) \
do\
{\
PT_YIELD_FLAG = 0; \
LC_SET((pt)->lc); \ //注：保存断点，下次从这里开始执行!!!
if((PT_YIELD_FLAG == 0) || !(cond)) { \
returnPT_YIELDED; \
}\
}while(0)


#defineLC_SET(s) s = __LINE__; case __LINE__: //保存行数，即 lc 被设置成该 LC_SET 所在的行
```

从源代码可以看出，第一次执行 PROCESS_WAIT_EVENT_UNTIL 总是被挂起（因为 PT_YIELD_FLAG==0 为真，会直接执行 return PT_YIELDED）。

该进程下一次被调试的时候，总是从 PROCESS_BEGIN()开始，而 PROCESS_BEGIN 宏包含两条语句：其一，将 PT_YIELD_FLAG 置 1；再者，switch(pt->lc)，从而跳转到 case __LINE__(见上述源码的注释)。接着判断 if 条件，此时 PT_YIELD_FLAG=1，若条件为真，则不执行 return，继续后续的内容，从而进程接着执行。

2）process_post 函数

精简后（去除一些用于调试的代码）源码如下：

```
//process_post(&print_world_process, event_data_ready, NULL); 即把事件 event_data_ready 加入事件队列
intprocess_post(struct process *p, process_event_t ev, process_data_t data)
{
staticprocess_num_events_t snum;

if(nevents == PROCESS_CONF_NUMEVENTS)
{
returnPROCESS_ERR_FULL;
}

snum = (process_num_events_t)(fevent + nevents) % PROCESS_CONF_NUMEVENTS;
events[snum].ev = ev;
events[snum].data = data;
events[snum].p = p;
++nevents;

#if PROCESS_CONF_STATS
if(nevents > process_maxevents)
{
```

```
process_maxevents = nevents;
}
#endif

returnPROCESS_ERR_OK;
}
```

3）etimer_reset 函数

```
/*Reset the timer with the same interval.*/
voidetimer_reset(struct etimer *et)
{
timer_reset(&et->timer);
add_timer(et); //详情见前面
}

voidtimer_reset(struct timer *t)
{
t->start += t->interval; /
}
```

5. 实验步骤

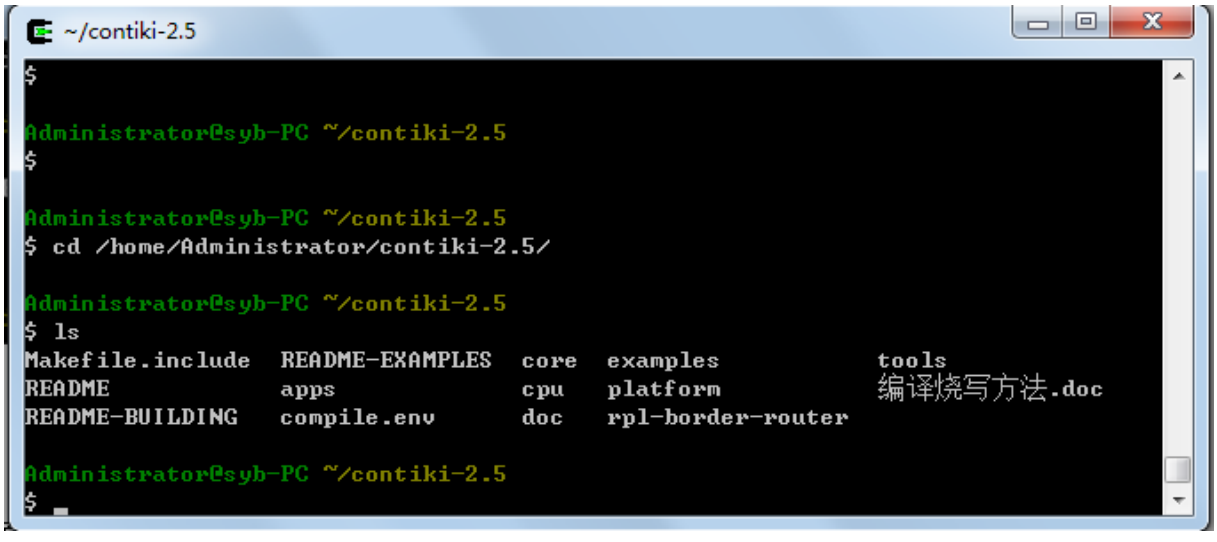
1）安装 contiki 系统源码（如果之前已经拷贝，则不用拷贝）

将产品光盘配套的 contiki 系统源码目录 contiki-2.5 拷贝至 Cygwin 开发环境安装目录的\home\Administrator 目录下。关于 Cygwin 开发环境的安装，请查看前面章节内容(1.2.2 IPv6 模块)部分，这里不再赘述。

如果 Cygwin 环境 home 目录下没有 Administrator 目录，说明您还未登陆过 Cygwin 环境，需要登陆后才可自动创建 Administrator 用户目录。

2）登陆 Cygwin

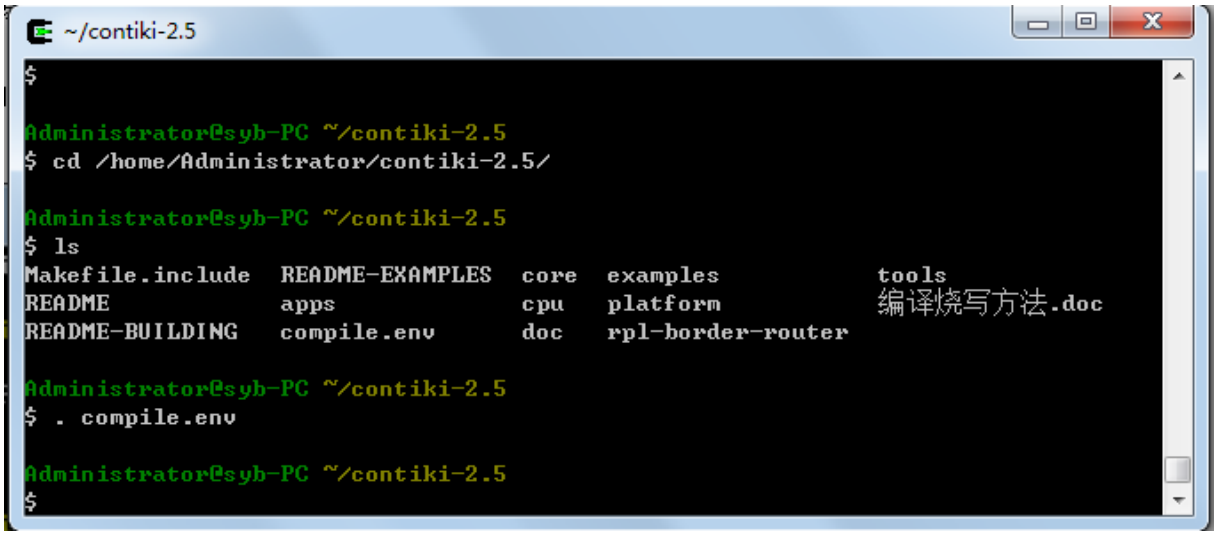
打开 Cygwin 开发环境，登陆进去。进入刚刚拷贝的 contiki-2.5 目录下，如图所示：



3）设置 IAR 编译器环境变量

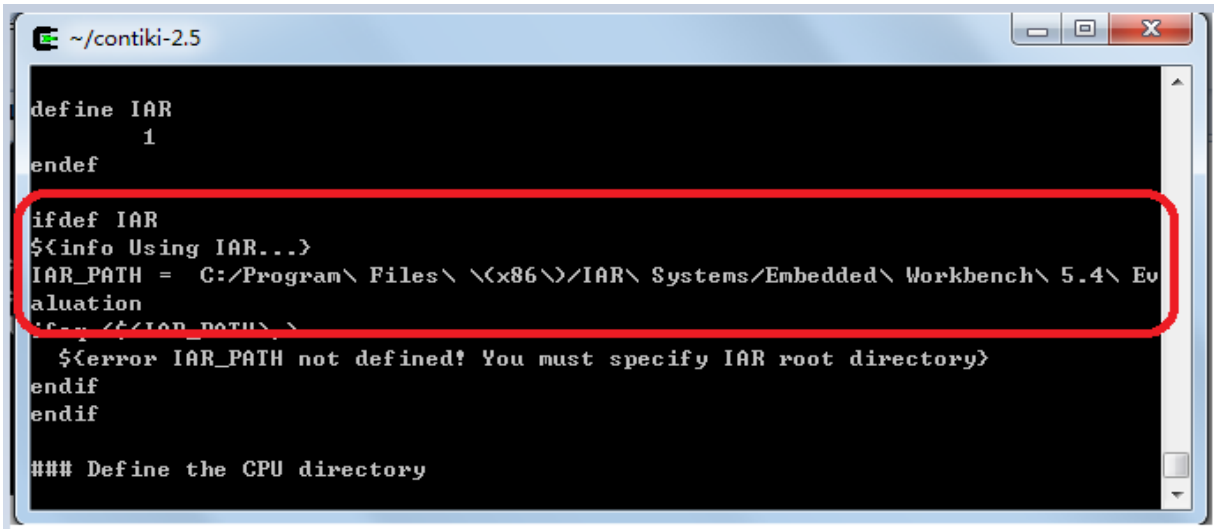
Cygwin 开发环境使用前面安装的 IAR EWARM 环境编译工具，因此要在此环境中加入 IAR 工具的安装路径，才可以使用相关编译工具对源码工程进行编译和下载。

执行. compile.env 命令（注意 .和 compile.env 中间有个空格）即可完成环境的设置。



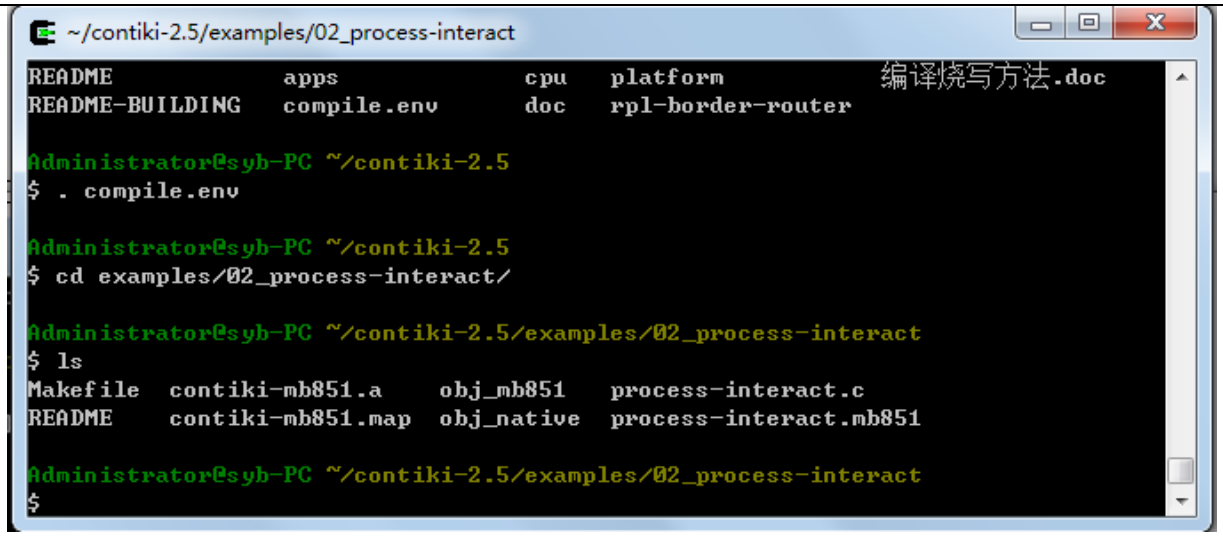
其中 compile.env 文件的内容为：export PATH=/cygdrive/c/Program\ Files/IAR\ Systems/Embedded\ Workbench\ 5.4\ Evaluation/arm/bin:$PATH（其中/cygdrive/引用 windows 各个盘的路径）。注意：具体变量用户可以根据自己环境中的 IAR 安装路径进行修改。

使用 vi 编辑 cpu/stm32w108/Makefile.stm32w108 文件，根据 IAR 实际的安装路径进行修改（如果之前已经修改则不用修改），如下：

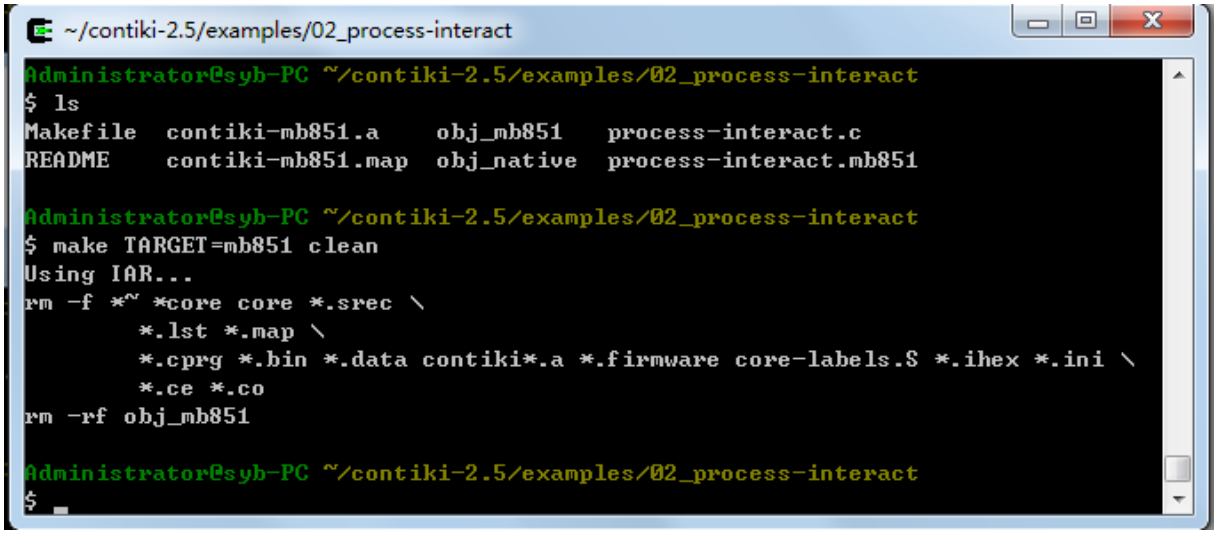


4）编译 process-interact 程序

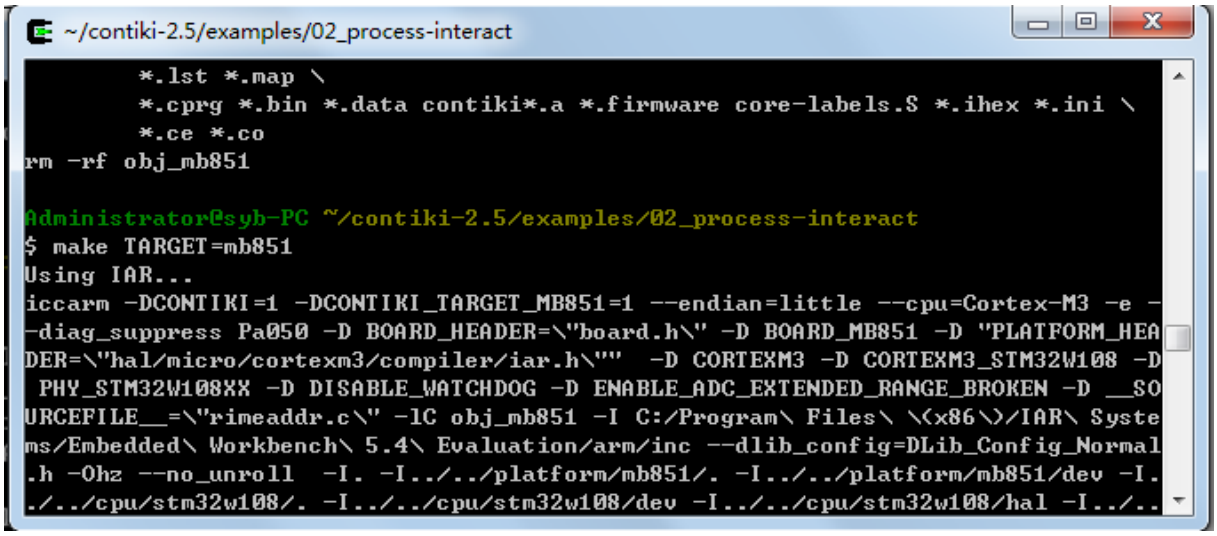
进入 contiki-2.5 目录下的 examples/实验目录下,将产品配套实验目录 02_process-interact 拷贝至此目录下，并进入该目录。



执行 make TARGET=mb851 clean （TARGET=mb851 指定针对的平台），清除工程中间文件：



执行 make TARGET=mb851 命令进行编译：

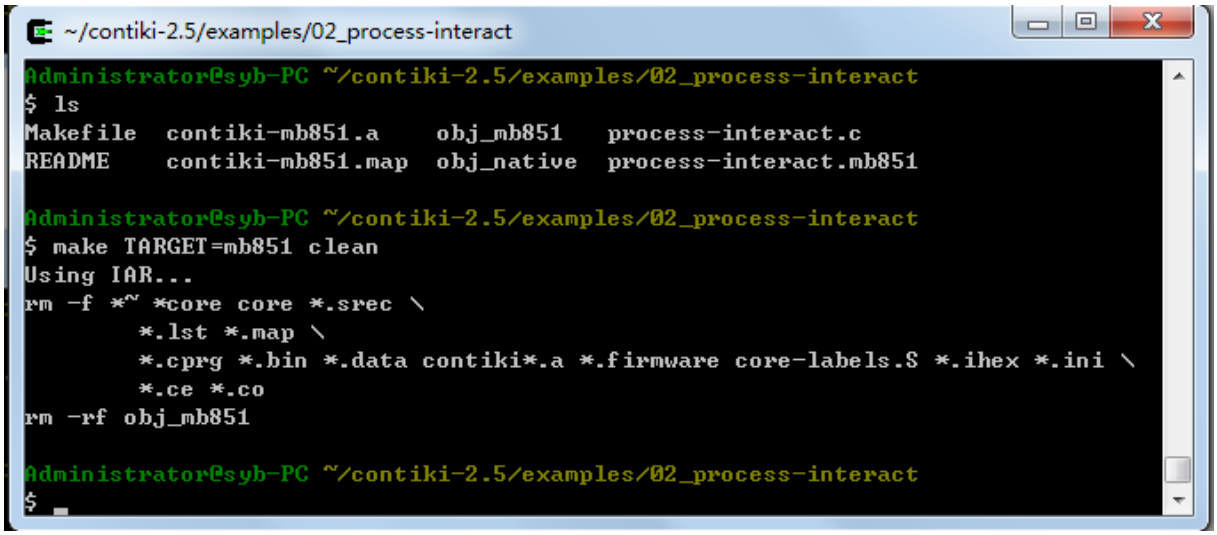


注意，一般编译前，需要 clean 以下工程，使用 make TARGET=mb851 clean 命令。

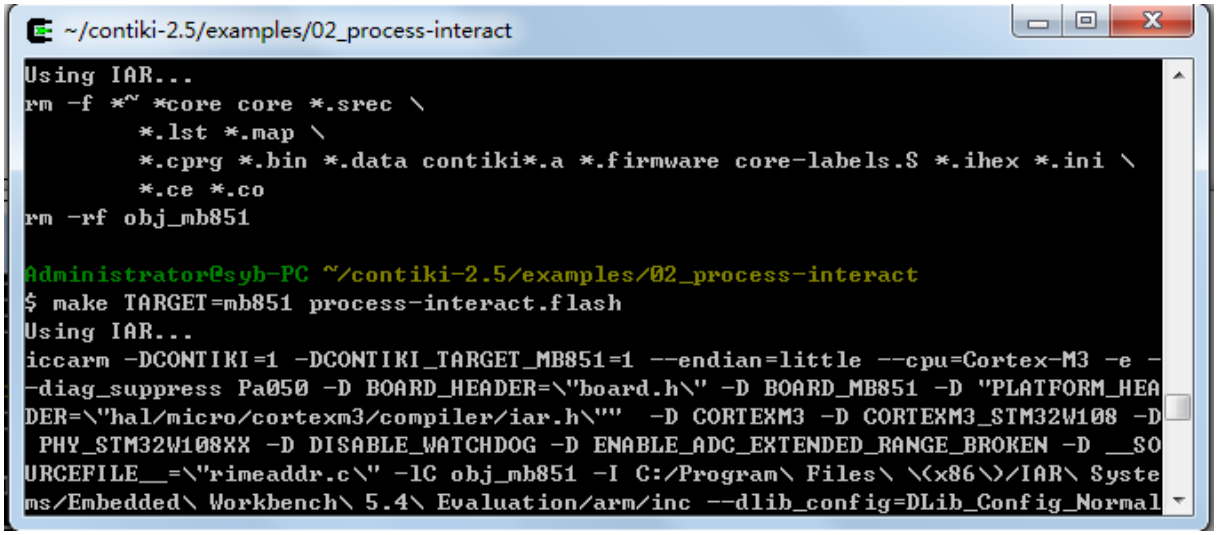
5）下载程序，运行测试

开启实验设备电源，使用 J-OB 仿真器和 J-OB 转接板连接 IPv6 模块，通过平台的“+”“-”按键选择目标模块，建议选择平台上 IPv6 根节点进行编程实验，因为其可以使用平台主板上的 RS232 串口。之后即可在 Cygwin 环境下烧写下载程序。

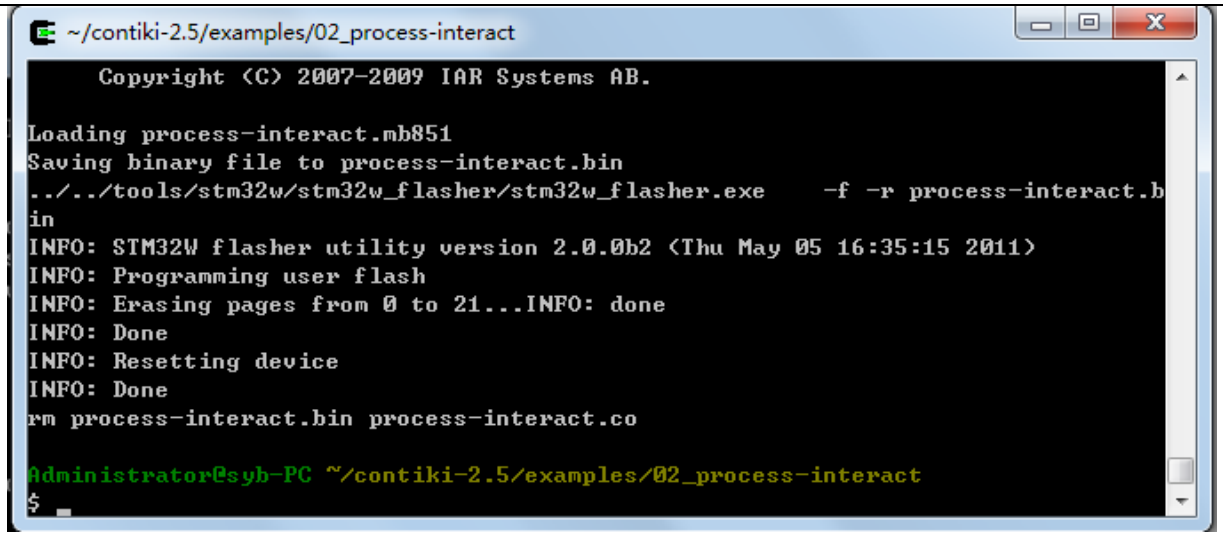
烧写之前，要先 clean：



然后运行 make TARGET=mb851 process-interact.flash 命令进行烧写：



烧写完成将提示如下信息：

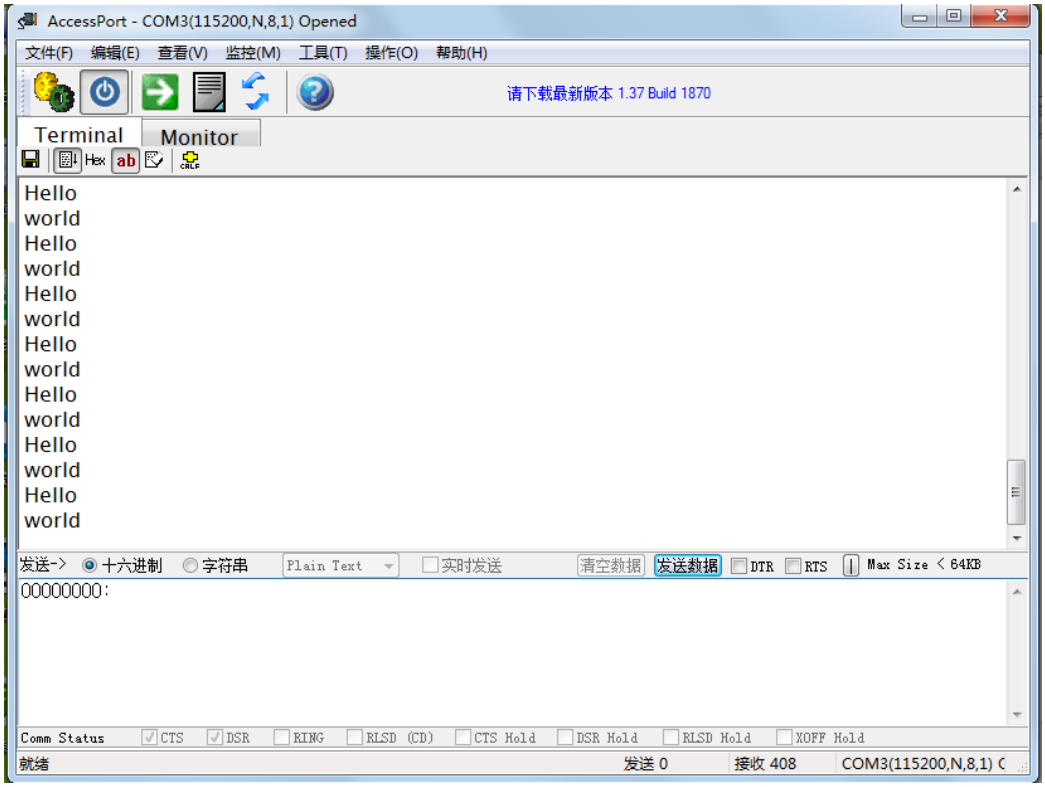


上述烧写过程将自动调用 IAR 相关工具完成。

6）通过串口查看打印信息

使用串口连接 IPv6 模块，也可以使用平台上 IPv6 根节点的串口拨码跳线(0001)连接底板的 Debug UART 串口。

在电脑端打开串口终端软件，正确设置，波特率 115200,无校验，8 数据位，1 位停止位，无硬件流，即可查看到 IPv6 模块运行打印的信息：



实验三. 基于 RPL 的点对点通信实验

1. 实验目的

- 掌握 Contiki 系统下 IPv6 协议 UDP 编程方法。
- 了解 Contiki 系统下 RPL 实现过程。

2. 实验环境

- 硬件：ICS-IOT-CEP 教学实验平台， PC 机，J-OB 仿真器和 J-OB 转接板
- 软件：Vmware Workstation +RHEL6 + MiniCom/超级终端，Cygwin +IAR EWARM，contiki OS，串口终端

3. 实验内容

- 编程实现基于 RPL 的 IPv6 根节点与 IPv6 节点的点到点的通讯。

4. 实验原理

4.10 RPL（Routing Protocol for LLN）协议

♦ **RPL 协议简介**

IETF RoLL（Routing over Lossy and Low-power Networks）工作组于 2008 年 2 月成立，属于 IETF 路由领域的工作组，致力于制定低功耗网络中 IPv6 路由协议的规范。RoLL 工作组的思路是从各个应用场景的路由需求开始，目前已经制定了 4 个应用场景的路由需求，包括家庭自动化应用（Home Automation，RFC5826）、工业控制应用（Industrial Control，RFC5673）、城市应用（Urban Environment，RFC5548）和楼宇自动化应用（Building Automation，draft-ietf-roll-building-routing-reqs）。

为了制订出适合低功耗网络的路由协议，RoLL 工作组首先对现有的传感器网络的路由协议进行了综述分析，工作组文稿 draft-ietf- roll-routing-survey 分析了相关协议的特点以及不足。然后研究了路由协议中路径选择的定量指标。RoLL 工作组文稿 draft- ietf-roll-routing-metrics 包含两个方面的定量指标，一方面是节点选择指标，包括节点状态，节点能量，节点跳数（Hop Count）；另一方面是链路指标，包括链路吞吐率、链路延迟、链路可靠性、ETX、链路着色（区分不同流类型）。为了辅助动态路由，节点还可以设计目标 函数（Objective Function）来指定如何利用这些定量指标来选择路径。

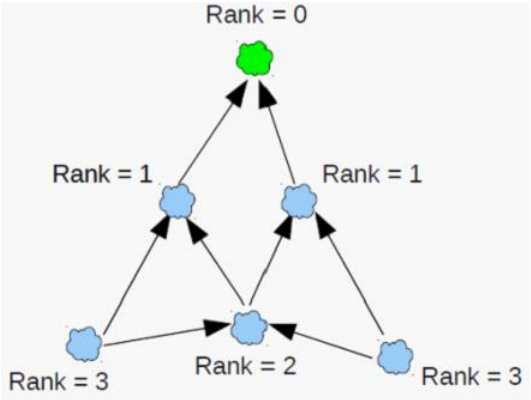
在路由需求、链路选择定量指标等工作的基础上，RoLL 工作组研究制定了 RPL（Routing Protocol for LLN）协议。RPL 协议目前是一个工作组文稿（draft-ietf-roll-rpl），已经更新到第 19 版本。

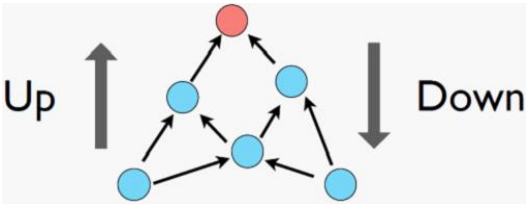
RPL 不同于传统网络路由协议，针对低功耗有损网络中的路由问题。考虑到在特定应用的文档中列出的广泛需求，RPL 被设计成高度模块化。其路由协议的核心满足特定应用的路由需求的交集，而对于特定的需求，可以通过添加附加模块的方式满足。RPL 是一个距离向量协议，它创建一个 DODAG， 其中路径从网络中的每个节点到 DODAG 根。使用距离向量路由协议而不是链路状态协议，这是有很多原因的。其中主要原因是低功耗有损网络中节点资源受限的性质。链路状态路由协议更强大，但是需要大量的资源，例如内存和用于同步 LSDB 的控制流量。DAG 可以有效防止路由环路问题，DAG 的根节点通过广播路由限制条件来过滤掉网络中的一些不满足条件的节点，然后节点通过路由度 量来选择最优的路径。

- **RPL 数据通信模型**

RPL 协议支持 3 种类型的数据通信模型：

- 低功耗节点到主控设备的多点到点的通信
- 主控设备到多个低功耗节点的点到多点通信
- 以及低功耗节点之间点到点的通信





- **RPL 相关术语**

1）DAG （Directed Acyclic Graph）：有向非循环图。一个所有边缘以没有循环存在的方式的有向图。

2）DAG Root：DAG 根节点。DAG 内没有外出边缘的节点。因为图是非循环的，所以按照定义所有的 DAGs 必须有至少一个 DAG 根，并且所有路径终止于一个根节点。

3）DODAG （Destination OrientedDAG）：面向目的地的有向非循环图。以单独一个目的地生根的 DAG。

4）DODAGRoot：一个 DODAG 的 DAG 根节点。它可能会在 DODAG 内部担当一个边界路由器，尤其是可能在 DODAG 内部聚合路由， 并重新分配 DODAG 路由到其他路由协议内。

5）Rank：等级。一个节点的等级定义了该节点相对于其他节点关于一个 DODAG 根节点的惟一位置。

6）OF （Objective Function）：目标函数。定义了路由度量，最佳目的，以及相关函数如何被用来计算出 Rank 值。此外，OF 指出了在 DODAG 内如何选择父节点从而形成

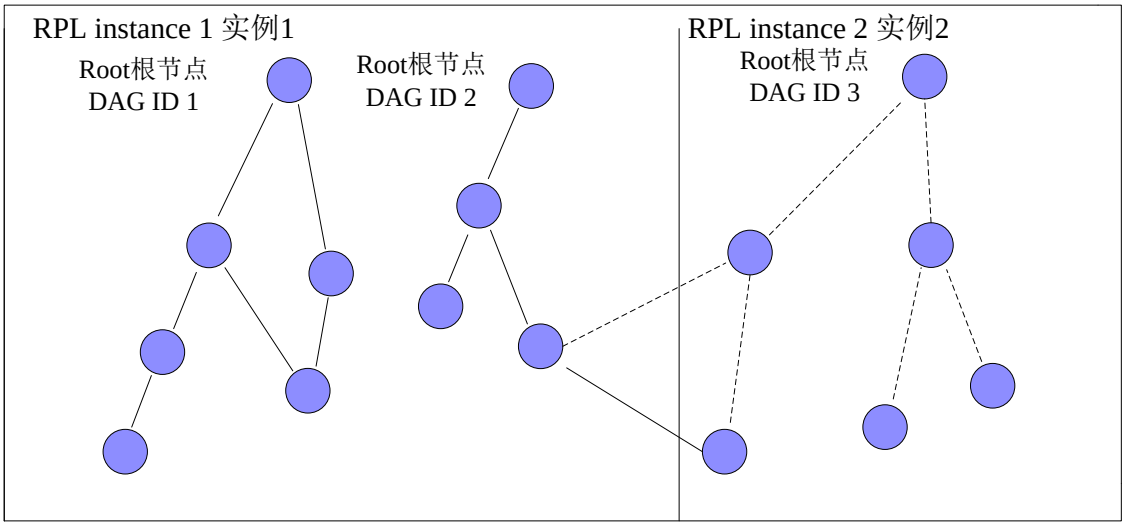
DODAG。

7） RPLInstanceID：一个网络的惟一标识。具有相同 RPLInstanceID 的 DODAG 共享相同的 OF。

8）RPL Instance：RPL 实例。共享同一个 RPL InstanceID 的一个或者多个 DODAG 的一个集合。

- **RPL 拓补图**

RPL 中规定，一个 DODAG 时一系列由有向边连接的顶点，之间没有直接的环路。RPL 通过构造从每个叶节点到 DODAG 根的路径集合来创建 DODAG。与树形拓扑相比，DODAG 提供了多余的路径。在使用 RPL 路由协议的网络中，可以包含一个或多个 RPLInstance。在每个 RPLInstance 中会存在多个 DODAG，每个 DODAG 都有一个不同的 Root。一个节点可以加入不同的 RPL Instanace， 但是在一个 Instance 内只能属于一个 DODAG。图 2 显示了使用 RPL 构造的网络拓扑图。



RPL 规定了 3 种消息， 即 DODAG 信息对象（DIO），DODAG 目的地通告对象（DAO），DODAG 信息请求（DIS）。DIO 消息是由 RPL 节点发送的，来通告 DODAG 和它的特征， 因此 DIO 用于 DODAG 发现、构成和维护。DIO 通过增加选项携带了一些命令性的信息。DAO 消息用于在 DODAG 中向上传播目的地消息，以填充祖先节点的路由表来支持 P2MP 和 P2P 流量。DIS 消息与 IPv6 路由请求消息相似，用于发现附近的 DODAG 和从附近的 RPL 节点请求 DIO 消息。DIS 消息没有附加的消息体。

- **RPL 路由建立**

当一个节点发现多个 DODAG 邻居时（可能是父节点或兄弟节点）， 它会使用多种规则来决定是否加入该 DODAG。一旦一个节点加入到一个 DODAG 中，它就会拥有到 DODAG 根的路由（可能是默认路由）。在 DODAG 中，数据路由传输分为向上路由和向下路由。向上路由指的是数据从叶子节点传送到根节点，可以支持 MP2P（多点到点）的传输；向下路由指的是数据从根节点传送到叶子节点，可以支持 P2MP（点到多点）和 P2P（点到点）传输。P2P 传输先通过向上路由到一个能到达目的地的祖先节点，然后再进行向下路由传输。对于不需要进行 P2MP 和 P2P 传输的网络来说，向下路由不需要建立。

向上路由建立通过 DIS 和 DIO 消息来完成。每个已经加入到 DAG 的节点会定时地发送多播地址的 DIO 消息，DIO 中包含了 DAG 的基本信息。新节点加入 DAG 时，会收到邻居节点发送的 DIO 消息，节点根据每个 DIO 中的 Rank 值， 选择一个邻居节点作为最佳的父节点， 然后根据 OF 计算出自己在 DAG 中的 Rank 值。节点加入到 DAG 后， 也会定时地发送 DIO 消息。另外，节点也可以通过发送 DIS 消息，让其它节点回应 DIO 消息。

向下路由建立通过 DAO 和 DAO-ACK 消息来完成。DAG 中的节点会定时向父节点发送 DAO 消息，里面包含了该节点使用的前缀信息。父节点收到 DAO 消息后， 会缓存子节点的前缀信息， 并回应 DAO-ACK。这样在进行路由时，通过前缀匹配就可以把数据包路由到目的地。

- **回路避免和回路检测机制**

与传统网络不同，在智能物件网络中由于低速率流量和网络不稳定性的特点，回路可能存在。RPL 不能从根本上保证消除回路，这意味着要在控制层面上使用开销很大的机制，并且这可能不太适合有损耗的和不稳定的环境。RPL 使用通过数据路径验证的回路检测机制作为替代，尽量避免回路。RPL 中两条基本的回路避免规则：

1）如果一个节点的邻节点的级别大于它的级别和 DAGMaxRankIncrease 的和， 那这个节点不允许被选作邻节点的父节点。

2）一个节点不允许是贪婪的并试着在 DODAG 中移动到更深的位置， 以增加 DODAG 父节点的选择，这样可能造成回路和不稳定性。RPL 中的路由检测机制通过在数据包的包头设定标志位来附带路由控制数据。携带这些标志位的确切位置还没有定义（如流标签）。主要思想是在包头里设定标志位， 以验证正在转发的包是用于检测回路的，还是用于检测 DODAG 不一致性的。

4.11 Contiki 中 IPv6 地址分配

在 IPV6 中，一种称为无状态自动配置的机制使用 EUI-64 地址来自动配置 IPV6 地址，EUI-64 格式即扩展惟一标识符，相当于 MAC-48 地址， 其主要用于 FireWire、IPv6、802.15.4 中。所谓无状态自动配置是指在网络中没有 DHCP 服务器的情况下，允许节点自行配置 IPV6 地址 的机制。

工作原理：自动将 48bit 的以太网 MAC 地址扩展成 64bit，再挂在一个 64bit 的前缀后面，组成一个 IPV6 地址，步骤如下：

1）将 48 位的 MAC 地址从中间分开，插入一个固定数值 FFFE；

2）将第 7 个比特位反转，如果原来是 0，就变为 1,如果原来是 1，就变为 0；

3）加上 64 位的网络前缀这就是一个完整的 IPV6 地址。

反转的原因：在 MAC 地址中，第 7 比特为 1 表示本地管理，为 0 表示全球管理；在 EUI-64 格式中，第 7 位为 1 表示全球惟一，为 0 表示本地惟一。

结点 IPv6 地址构造（IPv6 地址为 128 位，16 个字节）

根据 MAC 构造：

```
uip_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 0);//设置了 64 位的网络前缀
uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr);//函数利用自动配置机制补充了后 64 位主机寻址部分
uip_ds6_addr_add(&ipaddr, 0, ADDR_AUTOCONF);
```

直接设置

```
uip_ip6addr(&server_ipaddr, 0x2002, 0xc0a8, 0x64, 0xffff, 0, 0, 0, 1);

void
uip_ds6_set_addr_iid(uip_ipaddr_t *ipaddr, uip_lladdr_t * lladdr)
{
  /* We consider only links with IEEE EUI-64 identifier or
   * IEEE 48-bit MAC addresses */
#if (UIP_LLADDR_LEN == 8)
  memcpy(ipaddr->u8 + 8, lladdr, UIP_LLADDR_LEN);
  ipaddr->u8[8] ^= 0x02;
#elif (UIP_LLADDR_LEN == 6)
  memcpy(ipaddr->u8 + 8, lladdr, 3);
  ipaddr->u8[11] = 0xff;
  ipaddr->u8[12] = 0xfe;
  memcpy(ipaddr->u8 + 13, (uint8_t *)lladdr + 3, 3);
  ipaddr->u8[8] ^= 0x02;//反转
#else
#error uip-ds6.c cannot build interface address when UIP_LLADDR_LEN is not 6 or 8
#endif
}
```

4.12 Contiki 中的 UDP 编程

ContikiOS 实现了 TCP/IP 协议栈，但不是标准的 Socket 函数。在 IPv6 通信中，它有自己特殊的单播、多播实现方式。

- **Contiki 中 Socket 编程**

在 Contiki 中，过程如下：

1）首先通过一个函数 udp_new 建立一个 udp 连接：

```
server_conn = udp_new(NULL, UIP_HTONS(27000), NULL);
```

2）然后 bind：

```
udp_bind(server_conn, UIP_HTONS(UDP_SERVER_PORT));
```

3）接收和发送

Contiki 中接收数据是用事件的机制实现，当检测有 tcpip_event 事件时，调用 tcpip_handler()，在这个函数里面处理接收到的数据：

```
if(ev == tcpip_event) {
        tcpip_handler();
}
//tcpip_handler 代码如下,收到的数据放在 uip_appdata 指向的区域
static void tcpip_handler(void)
{
  char *appdata;


  //pasing
  if(uip_newdata()) {
appdata = (char *)uip_appdata;
……
}
……
}
```

发送数据：

```
uip_udp_packet_sendto(server_conn, buf, strlen(buf),
                      &server_ipaddr, UIP_HTONS(UDP_SERVER_PORT));
```

其中，参数 server_conn 为 udp 连接号，server_ipaddr 为要发往的 IPv6 地址。

4.13 关键代码分析

udp-server.c 本实验采用 IPv6 根节点作为服务器，其主要代码如下：

```
/*---------------------------------------------------------------------------*/
static void
tcpip_handler(void)/*解析协议数据包处理函数*/
{
  char *appdata;
  if(uip_newdata()) {//uip_newdata()为真，即远程连接的主机有发送新数据。
  leds_on(LEDS_GREEN);
  clock_delay(8000);
  leds_off(LEDS_GREEN);
    appdata = (char *)uip_appdata;
    appdata[uip_datalen()] = 0;
    PRINTF("DATA recv '%s' from ", appdata);
    PRINTF("%d",
           UIP_IP_BUF->srcipaddr.u8[sizeof(UIP_IP_BUF->srcipaddr.u8) - 1]);
    PRINTF("\n\r");
#if SERVER_REPLY//并未定义
    PRINTF("DATA sending reply\n");
    uip_ipaddr_copy(&server_conn->ripaddr, &UIP_IP_BUF->srcipaddr);
    uip_udp_packet_send(server_conn, "Reply", sizeof("Reply"));
```

```
      uip_create_unspecified(&server_conn->ripaddr);
#endif
   }
}
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(udp_server_process, ev, data)
{
   uip_ipaddr_t ipaddr;
   struct uip_ds6_addr *root_if;
   PROCESS_BEGIN();/*进程开启*/
   PROCESS_PAUSE();/*进程阻塞*/
   SENSORS_ACTIVATE(button_sensor);/*进程激活*/
   PRINTF("\n\rUDP server started\n");

#if UIP_CONF_ROUTER//是否定义 RPL
#if 0
/* Mode 1 - 64 bits inline */
    uip_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 1);
#elif 1
/* Mode 2 - 16 bits inline */
   /*设置 IPv6 地址*/
   uip_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0x00ff, 0xfe00, 1);
#else
/* Mode 3 - derived from link local (MAC) address */
   uip_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 0);
   uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr);
#endif

   uip_ds6_addr_add(&ipaddr, 0, ADDR_MANUAL);
   root_if = uip_ds6_addr_lookup(&ipaddr);//检查回路
   if(root_if != NULL) {
      rpl_dag_t *dag;
      rpl_set_root((uip_ip6addr_t *)&ipaddr);//设置 DAG root 为 ipaddr
      dag = rpl_get_dag(RPL_ANY_INSTANCE);//获得 RPL_ANY_INSTANCE 标识的 DAG
      uip_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 0);
      rpl_set_prefix(dag, &ipaddr, 64);//设置 DIO 消息的 DODAGID
      PRINTF("\r\ncreated a new RPL dag\n");
   } else {
      PRINTF("failed to create a new RPL DAG\n");
   }
#endif /* UIP_CONF_ROUTER */

   print_local_addresses();   /*打印地址信息*/

   /* The data sink runs with a 100% duty cycle in order to ensure high
      packet reception rates. */
```

```
    NETSTACK_MAC.off(1);
/*UDP 服务建立*/
    server_conn = udp_new(NULL, UIP_HTONS(UDP_CLIENT_PORT), NULL);
    udp_bind(server_conn, UIP_HTONS(UDP_SERVER_PORT));

    PRINTF("\r\nCreated a server connection with remote address ");
    PRINT6ADDR(&server_conn->ripaddr);
    PRINTF(" local/remote port %u/%u\n", UIP_HTONS(server_conn->lport),
            UIP_HTONS(server_conn->rport));

    while(1) {/*处理 UDP 数据包*/
      PROCESS_YIELD();
      if(ev == tcpip_event) {
        tcpip_handler();
      } else if (ev == sensors_event && data == &button_sensor) {
        PRINTF("Initiaing global repair\n");
        rpl_repair_dag(rpl_get_dag(RPL_ANY_INSTANCE));
      }
    }

    PROCESS_END();/*进程结束*/
}
```

udp-client.c 本实验采用 IPv6 节点作为客户端，其主要代码如下：

```
/*---------------------------------------------------------------------------*/
PROCESS(udp_client_process, "UDP client process");//定义 udp_client_process 进程
AUTOSTART_PROCESSES(&udp_client_process);//加入到上电自启动列表中
/*---------------------------------------------------------------------------*/
static void
tcpip_handler(void)
{
    char *str;

    if(uip_newdata()) {
      str = uip_appdata;
      str[uip_datalen()] = '\0';
      printf("DATA recv '%s'\n\r", str);
    }
}
/*---------------------------------------------------------------------------*/
static void
send_packet(void *ptr)
{
    static int seq_id;
    char buf[MAX_PAYLOAD_LEN];
```

```
  seq_id++;
  PRINTF("\r\nDATA send to %d 'Hello %d'\n",
         server_ipaddr.u8[sizeof(server_ipaddr.u8) - 1], seq_id);
  sprintf(buf, "Hello %d from the client", seq_id);
  uip_udp_packet_sendto(client_conn, buf, strlen(buf),
                        &server_ipaddr, UIP_HTONS(UDP_SERVER_PORT));
}
/*---------------------------------------------------------------------------*/
static void
print_local_addresses(void)
{
  int i;
  uint8_t state;

  PRINTF("\r\nClient IPv6 addresses: ");
  for(i = 0; i < UIP_DS6_ADDR_NB; i++) {
    state = uip_ds6_if.addr_list[i].state;
    if(uip_ds6_if.addr_list[i].isused &&
       (state == ADDR_TENTATIVE || state == ADDR_PREFERRED)) {
      PRINT6ADDR(&uip_ds6_if.addr_list[i].ipaddr);
      PRINTF("\n\r");
      /* hack to make address "final" */
      if (state == ADDR_TENTATIVE) {
  uip_ds6_if.addr_list[i].state = ADDR_PREFERRED;
      }
    }
  }
}
/*---------------------------------------------------------------------------*/
static void
set_global_address(void)
{
  uip_ipaddr_t ipaddr;

  uip_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 0);
  uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr);
  uip_ds6_addr_add(&ipaddr, 0, ADDR_AUTOCONF);

#if 0
/* Mode 1 - 64 bits inline */
   uip_ip6addr(&server_ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 1);
#elif 1
/* Mode 2 - 16 bits inline */
  uip_ip6addr(&server_ipaddr, 0xaaaa, 0, 0, 0, 0, 0x00ff, 0xfe00, 1);
#else
/* Mode 3 - derived from server link-local (MAC) address */
```

```
    uip_ip6addr(&server_ipaddr, 0xaaaa, 0, 0, 0, 0x0250, 0xc2ff, 0xfea8, 0xcd1a); //redbee-econotag
#endif
}
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(udp_client_process, ev, data)
{
   static struct etimer periodic;
   static struct ctimer backoff_timer;
   PROCESS_BEGIN();/*进程开启*/
   PROCESS_PAUSE();/*进程阻塞*/
   set_global_address();
   PRINTF("\r\nUDP client process started\n");
   print_local_addresses();
/*UDP 服务建立*/
   /* new connection with remote host */
   client_conn = udp_new(NULL, UIP_HTONS(UDP_SERVER_PORT), NULL);
   udp_bind(client_conn, UIP_HTONS(UDP_CLIENT_PORT));
   PRINTF("\r\nCreated a connection with the server ");
   PRINT6ADDR(&client_conn->ripaddr);
   PRINTF(" local/remote port %u/%u\n",
     UIP_HTONS(client_conn->lport), UIP_HTONS(client_conn->rport));

   etimer_set(&periodic, SEND_INTERVAL);//定义定时器
   while(1) {/*处理 UDP 数据包*/
      PROCESS_YIELD();
      if(ev == tcpip_event) {
         tcpip_handler();
      }

      if(etimer_expired(&periodic)) {
         etimer_reset(&periodic);//定时器复位
         ctimer_set(&backoff_timer, SEND_TIME, send_packet, NULL);//定义回调定时器，该定时器是驱
动某一个回调函数的
     //有一点需要注意的是，ctimer 实际是要靠 etimer 来驱动的。因为他本身就是一个进程
     }
   }

   PROCESS_END();
}
```

5. 实验步骤

1）安装 contiki 系统源码（如果之前已经拷贝，则不用拷贝）

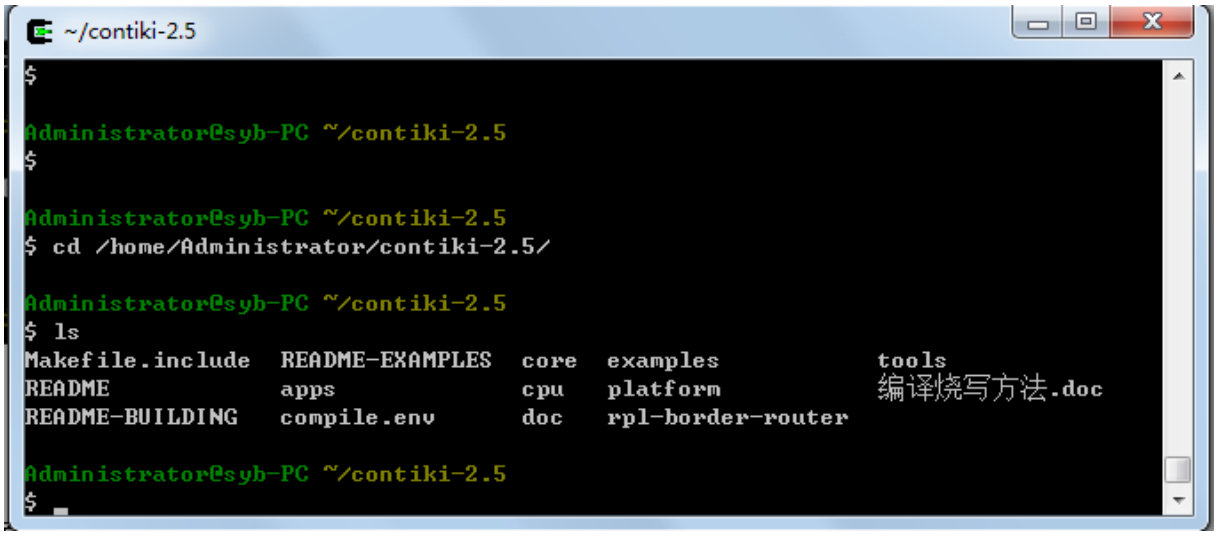
将产品光盘配套的 contiki 系统源码目录 contiki-2.5 拷贝至 Cygwin 开发环境安装目录的\home\Administrator 目录下。关于 Cygwin 开发环境的安装，请查看前面章节内容(1.2.2 IPv6

模块)部分，这里不再赘述。

如果 Cygwin 环境 home 目录下没有 Administrator 目录，说明您还未登陆过 Cygwin 环境，需要登陆后才可自动创建 Administrator 用户目录。

2）登陆 Cygwin

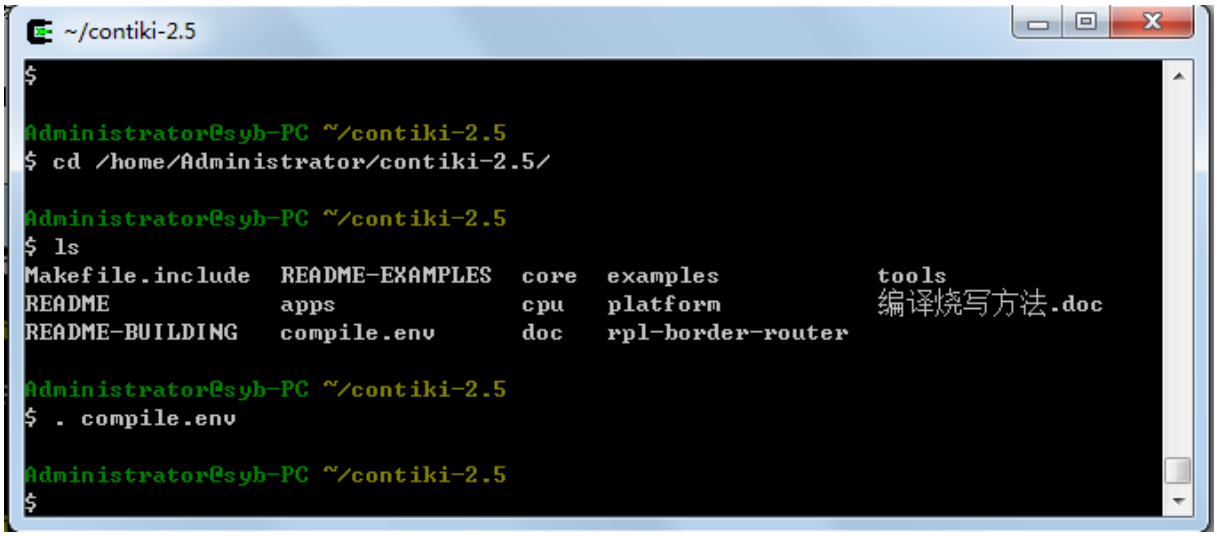
打开 Cygwin 开发环境，登陆进去。进入刚刚拷贝的 contiki-2.5 目录下，如图所示：



3）设置 IAR 编译器环境变量

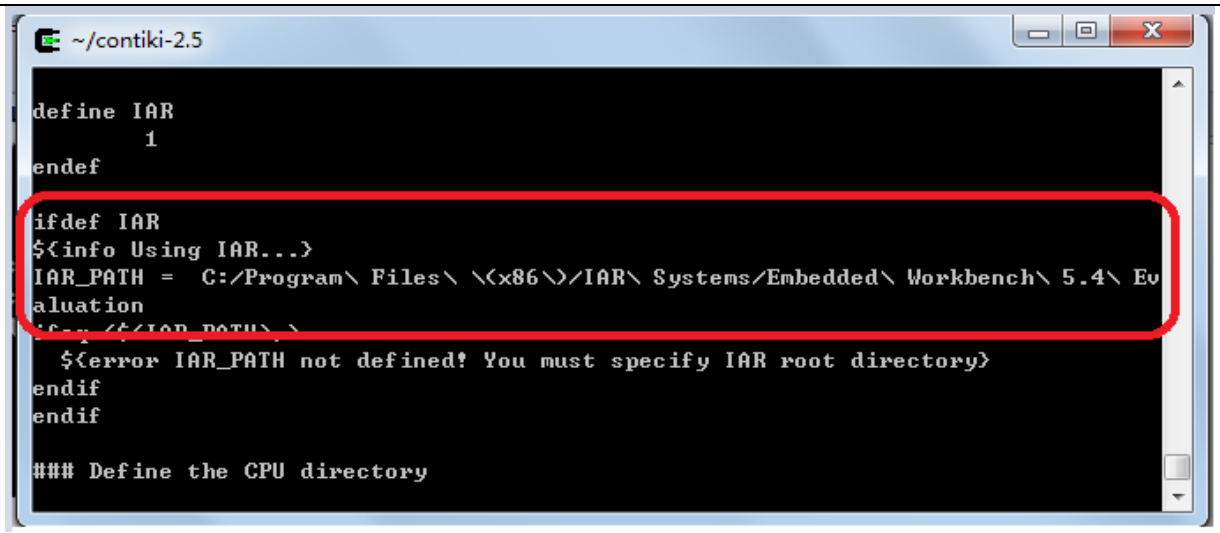
Cygwin 开发环境使用前面安装的 IAR EWARM 环境编译工具，因此要在此环境中加入 IAR 工具的安装路径，才可以使用相关编译工具对源码工程进行编译和下载。

执行. compile.env 命令（注意 .和 compile.env 中间有个空格）即可完成环境的设置



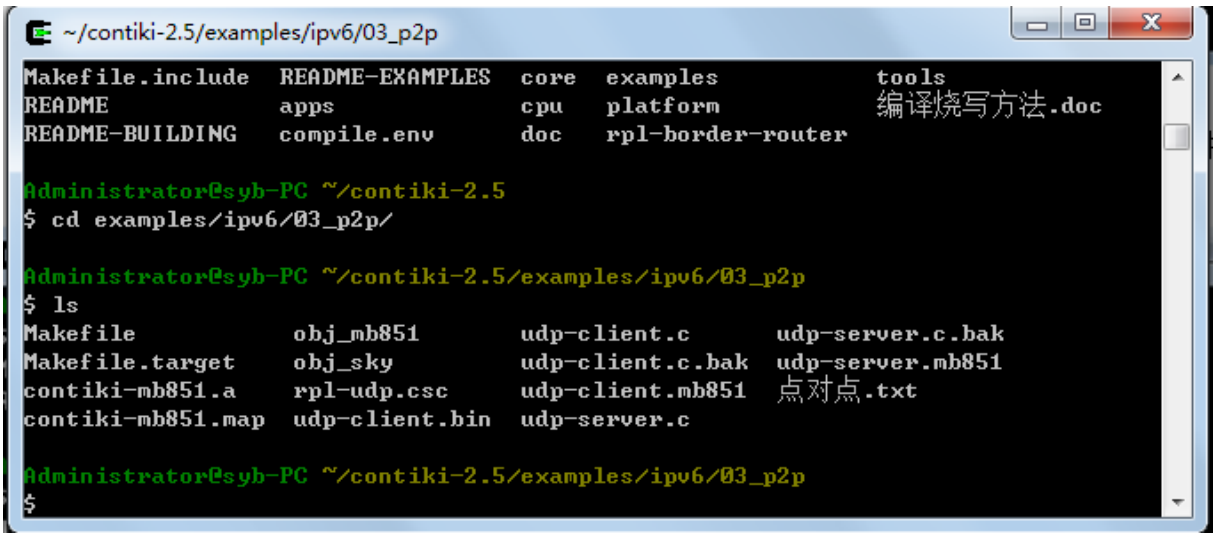
其中 compile.env 文件的内容为：export PATH=/cygdrive/c/Program\ Files/IAR\ Systems/Embedded\ Workbench\ 5.4\ Evaluation/arm/bin:$PATH（其中/cygdrive/引用 windows 各个盘的路径）。注意：具体变量用户可以根据自己环境中的 IAR 安装路径进行修改。

使用 vi 编辑 cpu/stm32w108/Makefile.stm32w108 文件，根据 IAR 实际的安装路径进行修改（如果之前已经修改则不用修改），如下：

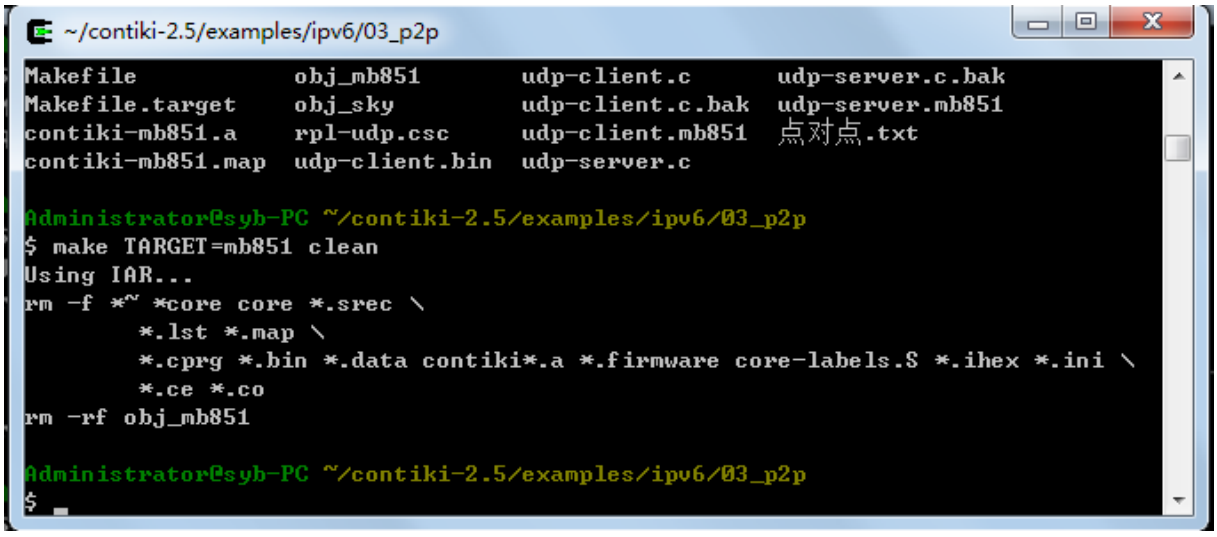


4）编译程序

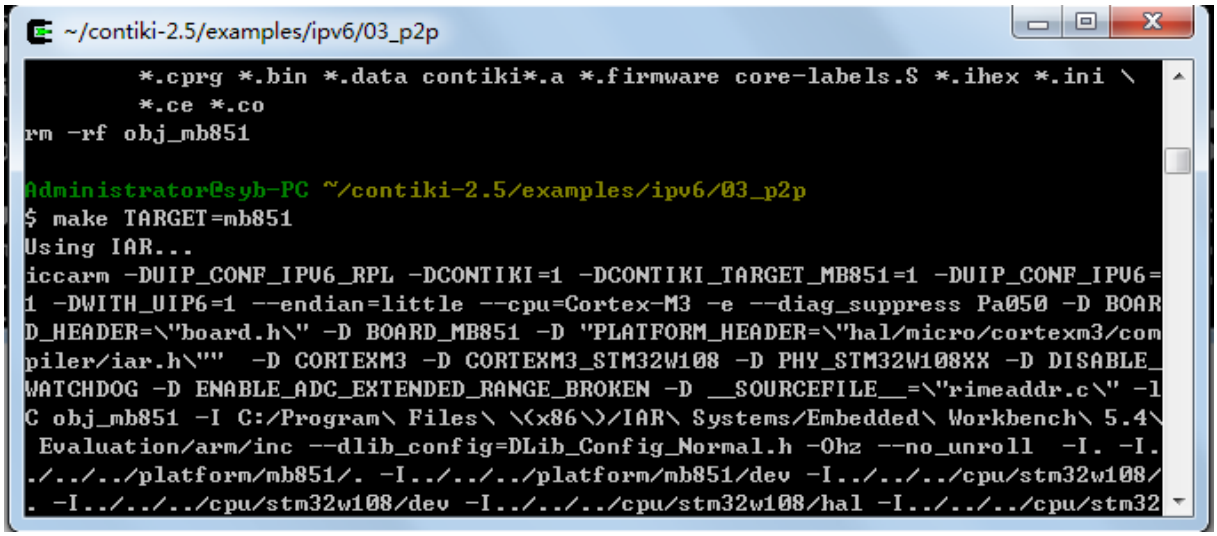
进入 contiki-2.5 目录下的 examples/ipv6 实验目录下,将产品配套实验目录 03_p2p 文件夹拷贝至此目录下，并进入该目录。



执行 make TARGET=mb851 clean （TARGET=mb851 指定针对的平台），清除工程中间文件：



执行 make TARGET=mb851 命令进行编译：

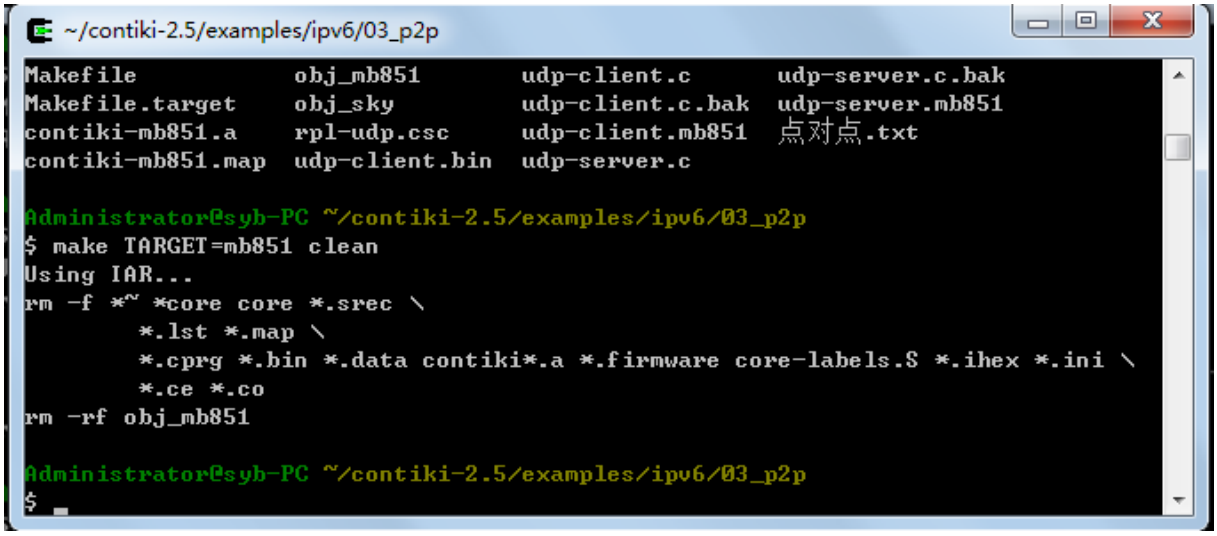


注意，一般编译前，需要 clean 以下工程，使用 make TARGET=mb851 clean 命令。

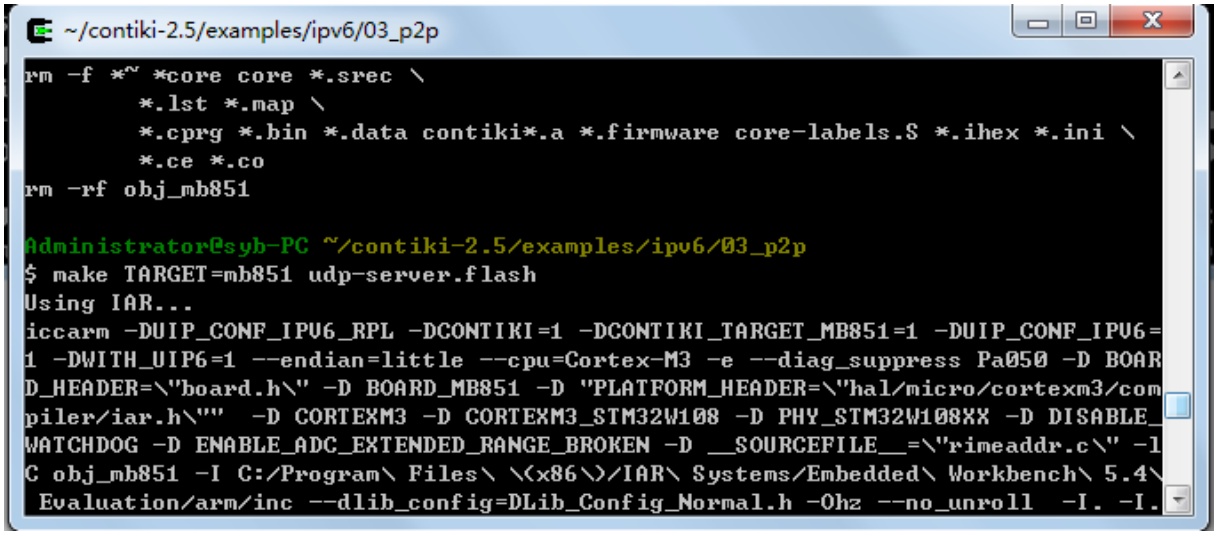
5）下载服务器程序

开启实验设备电源，使用 J-OB 仿真器和 J-OB 转接板连接 IPv6 模块，通过平台的“+”“-”按键选择目标模块，选择平台上 IPv6 根节点进行编程实验，因为其可以使用平台主板上的 RS232 串口。之后即可在 Cygwin 环境下烧写下载程序。

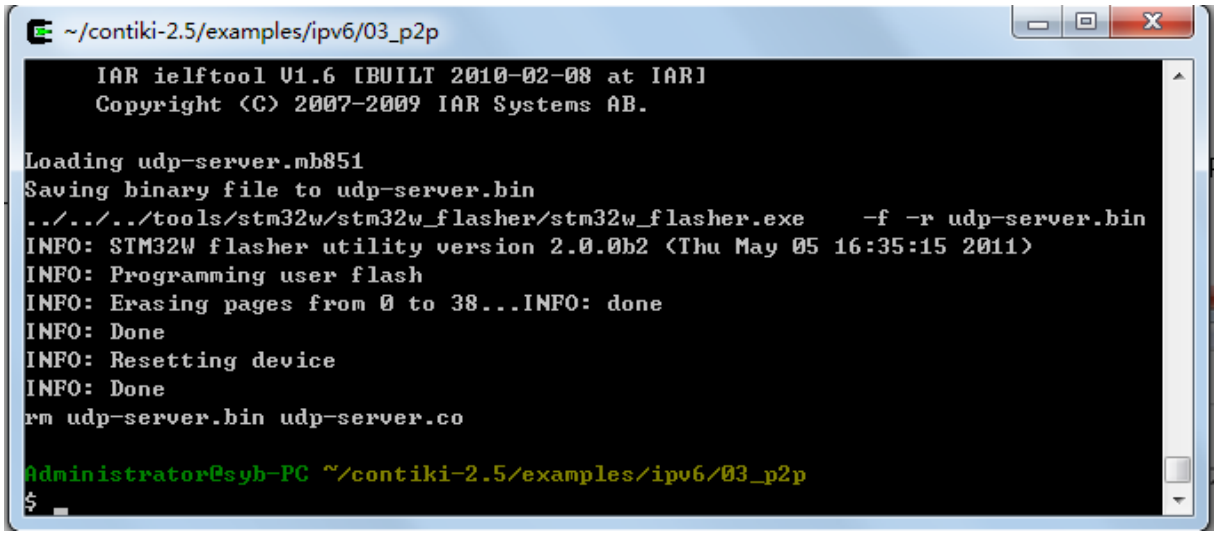
烧写之前，要先 clean：



然后运行 make TARGET=mb851 udp-server.flash 命令进行烧写：



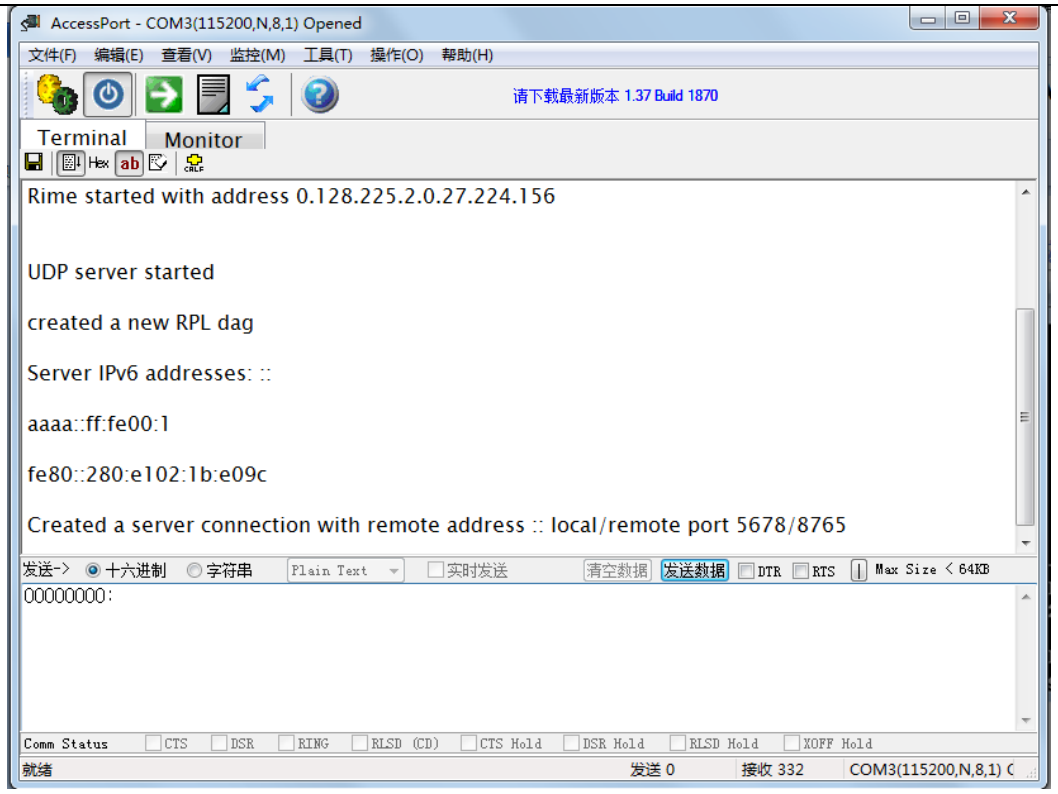
烧写完成将提示如下信息：



上述烧写过程将自动调用 IAR 相关工具完成。

烧写完毕，串口拨码跳线(0001)连接底板的 Debug UART 串口。

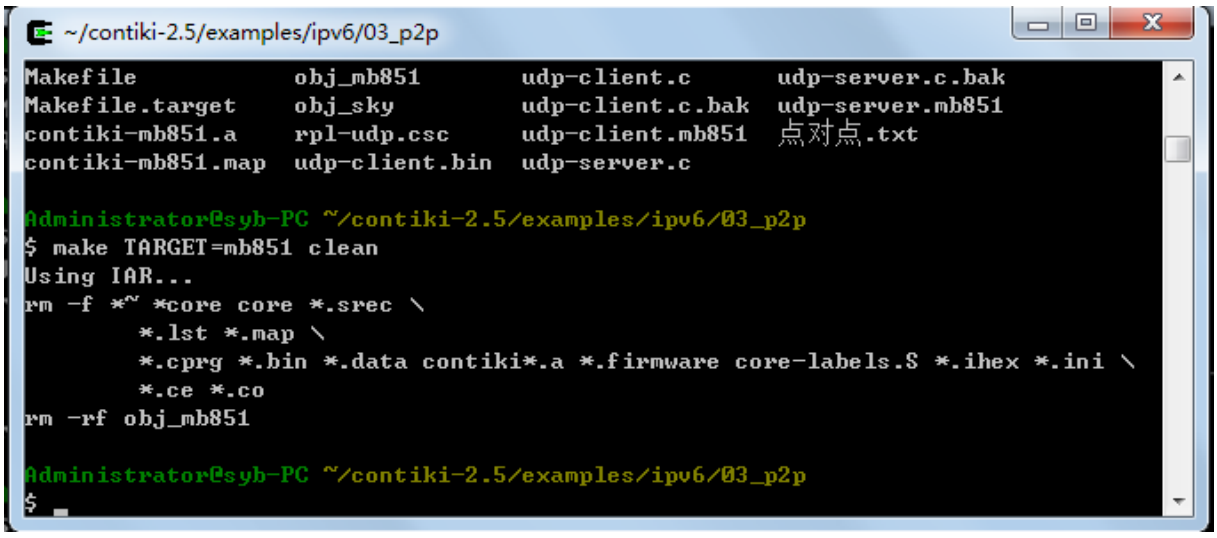
在电脑端打开串口终端软件，正确设置，波特率 115200,无校验，8 数据位，1 位停止位，无硬件流，即可查看到 IPv6 模块运行打印的信息（IPV6 模块重新上电）：



6）下载客户端程序

使用 J-OB 仿真器和 J-OB 转接板连接任意一个 IPv6 节点，通过平台的“+”“-”按键选择目标模块，之后即可在 Cygwin 环境下烧写下载程序。

烧写之前，要先 clean：



然后运行 make TARGET=mb851 udp-client.flash 命令进行烧写：

```
~/contiki-2.5/examples/ipv6/03_p2p
        *.ce *.co
rm -rf obj_mb851

Administrator@syb-PC ~/contiki-2.5/examples/ipv6/03_p2p
$ make TARGET=mb851 udp-client.flash
Using IAR...
iccarm -DUIP_CONF_IPV6_RPL -DCONTIKI=1 -DCONTIKI_TARGET_MB851=1 -DUIP_CONF_IPV6=
1 -DWITH_UIP6=1 --endian=little --cpu=Cortex-M3 -e --diag_suppress Pa050 -D BOAR
D_HEADER=\"board.h\" -D BOARD_MB851 -D "PLATFORM_HEADER=\"hal/micro/cortexm3/com
piler/iar.h\""  -D CORTEXM3 -D CORTEXM3_STM32W108 -D PHY_STM32W108XX -D DISABLE_
WATCHDOG -D ENABLE_ADC_EXTENDED_RANGE_BROKEN -D __SOURCEFILE__=\"rimeaddr.c\" -l
C obj_mb851 -I C:/Program\ Files\ \(x86\)/IAR\ Systems/Embedded\ Workbench\ 5.4\
 Evaluation/arm/inc --dlib_config=DLib_Config_Normal.h -Ohz --no_unroll  -I. -I.
./../../platform/mb851/. -I../../../platform/mb851/dev -I../../../cpu/stm32w108/
. -I../../../cpu/stm32w108/dev -I../../../cpu/stm32w108/hal -I../../../cpu/stm32
w108/simplemac -I../../../cpu/stm32w108/hal/micro/cortexm3 -I../../../cpu/stm32w
```

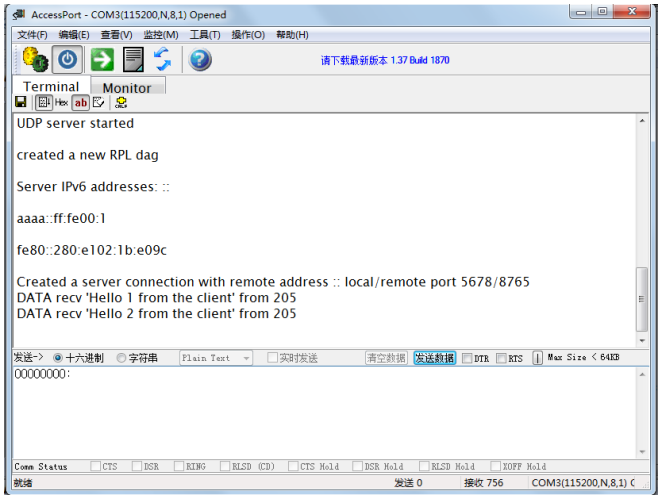
烧写完成将提示如下信息：

```
~/contiki-2.5/examples/ipv6/03_p2p
     IAR ielftool V1.6 [BUILT 2010-02-08 at IAR]
     Copyright (C) 2007-2009 IAR Systems AB.

Loading udp-client.mb851
Saving binary file to udp-client.bin
../../../tools/stm32w/stm32w_flasher/stm32w_flasher.exe    -f -r udp-client.bin
INFO: STM32W flasher utility version 2.0.0b2 (Thu May 05 16:35:15 2011)
INFO: Programming user flash
INFO: Erasing pages from 0 to 38...INFO: done
INFO: Done
INFO: Resetting device
INFO: Done
rm udp-client.co udp-client.bin

Administrator@syb-PC ~/contiki-2.5/examples/ipv6/03_p2p
$
```

上述烧写过程将自动调用 IAR 相关工具完成。

7）通过服务器端串口查看两个节点通信信息

在 IPV6 根节点（服务器程序）连接的串口上，可以看到由节点客户端发送过来的数据包，如下所示：



实验四. 基于 IPv6 模块的单播与多播通信实验

1. 实验目的

- 掌握 Contiki 系统下 IPv6 地址分配方法。
- 了解 Contiki 系统下单播与组播实现方法。

2. 实验环境

- 硬件：ICS-IOT-CEP 教学实验平台， PC 机，J-OB 仿真器和 J-OB 转接板。
- 软件：Vmware Workstation +RHEL6 + MiniCom/超级终端，Cygwin +IAR EWARM，contiki OS，串口终端。

3. 实验内容

- 编程实现基于 IPv6 节点间的单播与多播通讯。

4. 实验原理

4.14 IPv6 寻址架构

- **IPv6 地址类型**

IPv6 地址为接口和接口组指定了 128 位的标识符。有三种地址类型：

单播。一个单播接口有一个标识符。发送给一个单播地址的包传递到由该地址标识的接口上。

任意点播。一般属于不同节点的一组接口有一个标识符。发送给一个任意点播地址的包传送到该地址标识的、根据选路协议距离度量最近的一个接口上。

多播。一般属于不同节点的一组接口有一个标识符。发送给一个多播地址的包传递到该地址所标识的所有接口上。

在 IPv6 中没有广播地址，它的功能正在被多播地址所代替。在本文中，地址内的字段给予一个规定的名字，例如“用户”。当名字后加上标识符一起使用(如“用户 ID”)时，则用来表示名字字段的内容。当名字和前缀一起使用时(如“用户前缀”)则表示一直到包括本字段在内的全部地址。在 IPv6 中，任何全“0”和全“1”的字段都是合法值，除非特殊地排除在外的。特别是前缀可以包含“0”值字段或以“0”为终结。

- **寻址模型**

所有类型的 IPv6 地址都被分配到接口，而不是节点。一个 IPv6 单播地址属于单个接口。因为每个接口属于单个节点，多个接口的节点，其单播地址中的任何一个可以用作该节点的标识符。所有接口至少需要有一个链路本地单播地址。一个单接口可以指定任何类型的多个 IPv6 地址(单播、任意点播、组播)或范围。具有大于链路范围的单播地址，对这样的接口是不需要的，也就是从非邻居或者到非邻居的这些接口，不是任何 IPv6 包的起源或目的地。这有时适用于点到点接口。对这样的寻址模型有一个例外：

如果处理多个物理接口的实现呈现在 Internet 层好像一个接口的话，一个单播地址或一组单播地址可以分配给多个物理接口。这对于在多个物理接口上负载共享很有用。目前的 IPv6 延伸了 IPv4 模型，一个子集前缀与一条链路相关联。多个子集前缀可以指定给同一链路。

- **地址的文本表示**

用文本串表示的 IPv6 地址有三种规范形式：

1）优先选用的形式为 x:x:x:x:x:x:x:x，其中 x 是 8 个 16 位地址段的十六进制值。

例如：

```
FEDC:BA538:7654:3210:FEDC:BA538:7654:3210
1080:0:0:0:8:800:200C:417A
```

个别字段中前面的 0 可以不写，但是每段必须至少有一位数字((2)中描述的情形除外)。

2）址的书写，指定了一个特殊的语法来压缩 0。使用“::”符号指示有多个 0 值的 16 位组。“::”符号在一个地址中只能出现一次。该符号也能用来压缩地址中前部和尾部的 0。

用下面的例子来说明：

```
1080:0:0:0:8:800:200C:417A 单播地址
FF01:0:0:0:0:0:0:101 组播地址
0:0:0:0:0:0:0:1 回返地址
0:0:0:0:0:0:0:0 未指定地址
```

可用下面的压缩格式表示：

```
1080::8:800:200C:417A 单播地址
FF01::101 组播地址
::1 回返地址
::未指定地址
```

3）当谈到 IPv4 和 IPv6 节点这样一个混合环境时，有时更适合于采用另一种表示形式：x:x:x:x:x:x:d.d.d.d,其中 x 是地址中 6 个高阶 16 位段的十六进制值，d 是地址中 4 个低价 8 位段的十进制值(标准 IPv4 表示)。

举例说明：

```
0:0:0:0:0:0:13.1.68.3
0:0:0:0:0:FFFF:129.144.52.38
```

写成压缩形式为：

```
::13.1.68.3
::FFFF.129.144.52.38
```

- **地址前缀的文本表示**

IPv6 地址前缀的表示方式和 IPv4 地址前缀在 CIDR 中的表示方式很相似。一个 IPv6 地址前缀可以表示为如下的形式：IPv6 地址/前缀长度

其中，IPv6 地址是上面表示的任何形式的 IPv6 地址。而前缀长度是组成前缀的十进制值，说明地址最左边的连续的地址位的长度。

例如，60 位长的前缀 12AB00000000CD3(十六进制)可用下面的合法格式来表示：

```
12AB:0000:0000:CD30:0000:0000:0000:0000/60
12AB::CD30:0:0:0:0/60
12AB:0:0:CD30::/60
```

但是，下面的表示方式是不合法的。

```
12AB:0:0:CD3/60 在任何一个 16 位段的地址块中，可以省略前部的 0。但不能省略尾部的 0。
12AB::CD30/60/左边的地址会展开成 12AB:0000:0000:0000:0000:000:0000:CD30
12AB::CD3/60/左边的地址会展开成 12AB:0000:0000:0000:0000:000:0000:0CD3
```

当书写节点地址和它的子网前缀两者时，可以组合成如下表示：

节点地址：

```
12AB:0:0:CD30:123:4567:89AB:CDEF
```

和它的子网号：

```
12AB:0:0:CD30::/60
```

可以缩写成为：

```
12AB:0:0:CD30:123:4567:89AB:CDEF/60
```

- **地址类型表示**

一个 IPv6 地址的具体类型是由地址的前面几位来指定的。包含这前面几位的可变长度字段称为格式前缀(FP)。这些前缀的初始分配如下：

```
保留 000000001/256
未分配 000000011/256
为 NSAP 地址保留 00000011/128
为 IPX 地址保留 00000101/128
未分配 00000111/128
未分配 000011/32
未分配 00011/16
可集聚全球单播地址 0011/8
未分配 0101/8
```

未分配 0111/8
未分配 1001/8
未分配 1011/8
未分配 1101/8
未分配 11101/16
未分配 111101/32
未分配 1111101/64
未分配 11111101/128
未分配 1111111001/512
链路本地单播地址 11111110101/1024
站点本地单播地址 11111110111/1024
组播地址 111111111/256

这样的分配方案支持可集聚地址、本地用地址和组播地址的直接分配，并有保留给 NSAP 地址和 IPX 地址的空间。其余的地址空间留给将来用。可用于已有使用的扩展(如附加可集聚地址等)或者新的用途(如将定位符和标识符分开)。地址空间的 15%是初始分配的，其余 85%的地址空间留作将来使用。单播地址和组播地址是由地址的高阶字节值来区分的：值为 FF(11111111)标识一个地址为组播地址，其他值则标识一个地址为单播地址。任意点播地址取自单播地址空间，和单播地址在语法上是无法区分的。

- **单播地址**

IPv6 单播地址是用连续的位掩码集聚的地址，类似于 CIDR 的 IPv4 地址。IPv6 中的单播地址分配有多种形式，包括全部可集聚全球单播地址、NSAP 地址、IPX 分级地址、站点本地地址、链路本地地址以及运行 IPv4 的主机地址。将来还可以定义另外的地址类型。

IPv6 节点对 IPv6 地址的内部结构可能知之甚多或知之甚少，这是由节点的作用决定的(例如，主机还是路由器)。在最简单的情况下，节点把单播地址(包括它本身)看成是无内部结构的的 128 位地址。

一个稍完善但仍很简单的主机可能还知道它所连接的链路的子网前缀，在这种场合下，不同地址可能有不同值。更完善的主机可能知道单播地址中其他分级边界。虽然一个非常简单的路由器可能对 IPv6 单播地址的内部结构一无所知，但为了运行选路协议，路由器对一个或多个分级边界要有更为普遍的知识。知道边界随路由器不同而不同，是由路由器在选路分级中所处的位置决定的。

- **接口标识符**

在 IPv6 单播地址中接口标识符用来标识链路的接口。标识符在该链路上应是唯一的。也可能在较宽范围内是唯一的。在许多情况下，一个接口标识符与该接口的链路层地址相同。在一个单节点上，同一个接口标识符可以用在多个接口上。在一个单节点的多个接口上，用同样的接口标识符不会影响接口标识符的全球唯一性，或由接口标识符创建的每个 IPv6 地址的全球唯一性。

在许多格式前缀中，接口标识符要求 64 位长，并构成 IEEEEUI-64 格式。基于 EUI-64 的接口标识符，当全球令牌可用时(如 IEEE48 位 MAC)，具有全球范围的意义。当全球令牌不可用时(如串行链路、隧道终点等)，则只具有本地范围的意义。当由 EUI-64 形成接口标识符时，若 u 位(IEEEEUI-64 术语中称全球/本地位)置 1，则表示全球范围；若 u 位置 0，则

表示本地范围。一个 EUI-64 标识符的头三个字节的二进制表示如下所示。

```
000112
|078563|
+----+----+----+----+----+----+----+
|cccc|ccug|cccc|cccc|cccc|cccc|
+----+----+----+----+----+----+----+
```

按 Internet 标准中的位序，其中 u 是全球/本地位，g 是个体/团体位，c 是公司标识符。

当形成接口标识符时，使用 u 位的动机是当硬件令牌不可用，即在串行链路、隧道终点等情况下，便于系统管理员人工配置本地范围标识符。另一种方法是用 0200:0:0:1、0200:0:0:2 等形式代替十分简单的::1、::2 等形式。在 IEEEEUI-64 标识符中使用全球/本地位的目的是为了将来技术的发展能利用具有全球范围的接口标识符所带来的好处。形成接口标识符的细节定义在 IPover<link>技术规范中，诸如 IPoverEthernet[ETHER]、IPoverFDDI[FDDI]等。

- **未指定地址**

地址 0:0:0:0:0:0:0:0 称为未指定地址。它不能分配给任何节点。意思是没有这个地址。它的一个应用示例是初始化主机时，在主机未取得自己的地址以前，可在它发送的任何 IPv6 包的源地址字段放上未指定地址。未指定地址不能在 IPv6 包中用作目的地址，也不能用在 IPv6 选路头中。

- **嵌有 IPv4 地址的 IPv6 地址**

IPv6 过渡机制[TRAN]包括一种技术，使主机和路由器能在 IPv4 选路基础设施上动态地以隧道方法传送 IPv6 包。使用该技术的 IPv6 节点要指定特殊的 IPv6 单播地址，它在低阶 32 位上携带 IPv4 地址。这种地址类型称其为“与 IPv4 兼容的 IPv6 地址”，并具有下面的格式：

```
|80 位|16|32 位|
+----------------------------------+------------------------+
|0000.............................0000|0000|IPv4 地址|
+----------------------------------+----+-------------------+
```

第二种类型的 IPv6 地址嵌有 IPv4 地址。该地址用来表示只支持 IPv4，而不支持 IPv6 的节点的 IPv6 地址。这种地址类型称为“与 IPv4 映射的 IPv6 地址”，并具有下面的格式：

```
|80 位|16|32 位|
+----------------------------------+------------------------+
|0000.............................0000|FFFF|IPv4 地址|
+----------------------------------+----+-------------------+
```

- **NSAP 地址**

NSAP 地址到 IPv6 地址的映射定义在[NSAP]中。对于已经规划或应用 OSINSAP 寻址计划，并希望应用 IPv6 或向 IPv6 过渡的网络实现者，该文件应该重新设计成 IPv6 寻址计划

来满足他们的需要。另外还定义了一套机制，用来在 IPv6 网络中支持 OSINSAP 寻址。如果需要这种支持的话，则必须要有这样的机制。该文件还定义了 OSI 地址格式内 IPv6 地址的映射，这应该是必需的。

- **IPX 地址**

IPX 地址到 IPv6 地址的映射表示如下：

```
|7|121 位|
+-------+-------------------------------------------------------+
|0000010|待定义|
+-------+-------------------------------------------------------+
```

- **可集聚全球单播地址**

全部可集聚全球单播地址定义在[AGGR]中。设计这样的地址格式为了既支持基于当前供应商的集聚，又支持被称为交换局的新的集聚类型。其组合使高效的选路集聚可用于直接连接到供应商和连接到交换局两者的站点上。站点可以选择连接到两种类型中的任何一种集聚点。

IPv6 可集聚全球单播地址格式如下所示：

```
|3|13|8|24|16|64 位|
+--+-----+---+--------+--------+-------------------------------+
|FP|TLA|RES|NLA|SLA|接口号|
||ID||ID|ID||
+--+-----+---+--------+--------+-------------------------------+
```

其中：

```
001(FP)用于可集聚全球单播地址的格式前缀(3 位)；
TLAID 为顶级集聚标识符；RES 保留将来用；
NLAID 为下一级集聚标识符；SLAID 为站点级集聚标识符；
INTERFACEID 为接口标识符。
在[AGGR]中，还规定了内容、字段长度和分配规则。
```

- **本地用 IPv6 单播地址**

规定了链路本地和站点本地两种类型的本地使用单播地址。链路本地地址用在单链路上，而站点本地地址用在单站点上。链路本地地址格式表示如下：

```
|10 位|54 位|64 位|
+----------+----------------------+-------------------------+
|1111111010|0|接口号|
+----------+----------------------+-------------------------+
```

设计链路本地地址的目的是为了用于诸如自动地址配置、邻居发现或无路由器存在的单链路的寻址

路由器不能将带有链路本地源地址或目的地址的任何包转发到其他链路上去。站点本地地址具有下面的地址格式：

```
|10 位|38 位|16 位|64 位|
+----------+------------+----------+--------------------------+
|1111111011|0|子网号|接口号|
+----------+------------+----------+--------------------------+
```

站点本地地址的设计目的是为了用于无需全球前缀的站点内部寻址。路由器不应转发站点外具有站点本地源或目的地址的任何包。

- **任意点播地址**

IPv6 任意点播地址是分配给一般属于不同节点的多个接口。根据这个特性，发送给任意点播地址的包，总是发送到具有该地址并按照选路协议测得距离为最近的接口。任意点播地址从单播地址空间分配而来，可用任何一种规定的单播地址格式。这样，任意点播地址和单播地址在语法上是无法区别的。当一个单播地址分配给多个接口时，如果把它转为任意点播地址，那么被分配该地址的节点，必须显式地配置，以便知道这是一个任意点播地址。

对于任何已分配的单播地址，有一个最长的地址前缀 P 用于标识拓扑地区。在该地区中，所有接口均属于该任意点播地址。在由 P 标识的区域内，任意点播组的每个成员，被告知在选路系统中作为一个独立实体(通常称之为“主机路由”)。在 P 标识的区域以外，任意点播地址可以集合在前缀 P 的选路通告中。

- **要求的任意点播地址**

预定的子网路由器任意点播地址，其格式如下：

```
|n 位|128-n 位|
+-----------------------------------------+---------------+
|子网前缀|00000000000000|
+-----------------------------------------+---------------+
```

在任意点播地址中，子网前缀用来标识一条特定链路。对于接口标识符置 0 的链路上的一个接口，其任意点播地址和单播地址语法上是相同的。发送给子网路由器任意点播地址的包会传递到子网上的一个路由器。与子网有接口的所有路由器需要支持子网路由器任意点播地址。子网路由器任意点播地址企图用在某些应用场合，即一个节点需要和远程子网上一组路由器中的一个进行通信的场合。例如当移动主机要和一个位于本子网的移动代理通信的场合。

- **多播地址**

IPv6 多播地址是一组节点的标识符。一个节点可以归属于任意数量的多播组。多播地址具有下面的格式：

```
|8|4|4|112 位|
+-------+----+----+---------------------------------------+
|11111111|flgs|scop|组号|
+--------+----+----+---------------------------------------+
```

地址开始的 11111111 标识该地址为多播地址。标志由 4 位组成：

```
+-+-+-+-+
|0|0|0|T|
+-+-+-+-+
```

前面 3 位为保留位，初始设置为 0。

T=0 指示一个永久分配的(熟知的)组播地址，由全球 Internet 编号机构进行分配。

T=1 指示一个非永久分配(临时)的组播地址。

4 位的多播范围值用来限制多播组的范围。该字段的可能值如下表：

```
0 保留 8 组织本地范围
点本地范围 9(未分配)
2 链路本地范围 A(未分配)
3(未分配)B(未分配)
4(未分配)C(未分配)
5 站点本地范围 D(未分配)
6(未分配)E 全球范围
7(未分配)F 保留
```

组标识符字段标识给定范围内的多播组，可以是永久的，也可以是临时的。永久分配的多播地址，意思是独立于范围值。例如，如果为 NTP 服务器组指定一个组标识符为 101(十六进制)的永久多播地址，于是：

```
FF01:0:0:0:0:0:0:101 指在同一节点上的所有 NTP 服务器。
FF02:0:0:0:0:0:0:101 指在同一链路上的所有 NTP 服务器。
FF05:0:0:0:0:0:0:101 指在同一站点上的所有 NTP 服务器。
FF0E:0:0:0:0:0:0:101 指 Internet 上的所有 NTP 服务器。
```

非永久分配的多播地址仅在给定范围内才有意义。例如，在某个站点由非永久的站点本地多播地址 FF15:0:0:0:0:0:0:101 标识的组与一个不同站点中使用同一个组标识符的组没有关系，与不同范围内使用同一个组标识符分配非永久地址的组也没有关系，与具有同一个组标识符的永久组也没有关系。多播地址在 IPv6 包中不能用作源地址或出现在任何选路头中。

- **预定义的多播地址**

下面为熟知的预定义的多播地址。

保留的多播地址：

```
FF00:0:0:0:0:0:0:0
FF01:0:0:0:0:0:0:0
FF02:0:0:0:0:0:0:0
FF03:0:0:0:0:0:0:0
FF04:0:0:0:0:0:0:0
FF05:0:0:0:0:0:0:0
FF06:0:0:0:0:0:0:0
FF07:0:0:0:0:0:0:0
```

```
FF08:0:0:0:0:0:0:0
FF09:0:0:0:0:0:0:0
FF0A:0:0:0:0:0:0:0
FF0B:0:0:0:0:0:0:0
FF0C:0:0:0:0:0:0:0
FF0D:0:0:0:0:0:0:0
FF0E:0:0:0:0:0:0:0
FF0F:0:0:0:0:0:0:0
```

上面列出的是保留的多播地址，且永远不能分配给任何多播组。所有节点地址：

```
FF01:0:0:0:0:0:0:1
FF02:0:0:0:0:0:0:1
```

上面列出的多播地址标识了范围 1(节点本地)或范围 2(链路本地)内的所有 IPv6 节点的组。所有路由器地址：

```
FF01:0:0:0:0:0:0:2
FF02:0:0:0:0:0:0:2
FF05:0:0:0:0:0:0:2
```

以上的多播地址标识了范围 1(节点本地)、范围 2(链路本地)或范围 5(站点本地)内的所有 IPv6 路由器的组。

路由器需要时刻保持收听以下多点传送地址上的信息流：

- 节点本地范围内的所有节点组播地址（FF01::1）；
- 节点本地范围内的所有路由器组播地址（FF01::2）；
- 链路本地范围内的所有节点组播地址（FF02::1）；
- 链路本地范围内的所有路由器组播地址（FF02::2）；
- 站点本地范围内的所有路由器组播地址（FF05::2）；

请求节点地址：FF02:0:0:0:0:1:FFXX:XXXX

上面的多播地址是从节点的单播和任意点播地址计算而得的。取单播或任意点播地址的低 24 位，并将其附加到前缀 FF02:0:0:0:0:1:FF00::/104 上形成一个请求节点多播地址，其范围 FF02:0:0:0:0:1:FF00:0000 至 FF02:0:0:0:0:1:FFFF:FFFF 之间。

例如，对应 IPv6 地址 4037::01:800:200E:8C6C 的请求节点组播地址是 FF02::1:FF0E:8C6C。IPv6 地址差别仅在高位，譬如，由于与不同的集聚相关联的多个高位前缀，将映射到同一个请求节点地址，因此减少了一个节点必须加入的组播地址数。对每个指定的单播和任意点播地址，一个节点需要计算并加入相关的请求节点组播地址。

♦ **新 IPv6 多播地址的分配**

目前将 IPv6 多播地址映射到 IEEE802MAC 地址的方法是用 IPv6 多播地址的低阶 32 位来创建 MAC 地址。值得提出的是令牌网有不同的处理方法，定义见[TOKEN]。32 位组标识符将生成唯一的 MAC 地址。由于新 IPv6 多播地址应当分配，所以组标识符总是在低阶 32 位上，如下图所示：

```
|8|4|4|80bits|32bits|
+-------+----+----+------------------------+----------------+
|11111111|flgs|scop|reservedmustbezero|groupID|
+--------+----+----+------------------------+----------------+
```

尽管将永久 IPv6 多播组数限制在 232，但在将来不可能成为极限。如果将来必须要超过这个限度，多播仍然能工作，只是处理稍慢而已。其他 IPv6 多播地址的定义和注册由 IANA[MASGN]完成

4.15 实现原理

- **单播**

首先，双方都注册一个 udp 连接：

```
simple_udp_register(&unicast_connection, UDP_PORT, NULL, UDP_PORT, receiver);
```

接收数据使用了自身的回调函数 receiver，定义如下：

```
static void
receiver(struct simple_udp_connection *c,
         const uip_ipaddr_t *sender_addr,
         uint16_t sender_port,
         const uip_ipaddr_t *receiver_addr,
         uint16_t receiver_port,
         const uint8_t *data,
         uint16_t datalen)
```

数据就在 data 中。

- **多播**

和 unicast 类似，先注册 udp 连接：

```
simple_udp_register(&broadcast_connection, UDP_PORT,NULL, UDP_PORT,receiver);
```

然后设置一个广播地址：

```
uip_create_linklocal_allnodes_mcast(&addr);
```

再用 simple_udp_sendto()发送，例如

```
simple_udp_sendto(&broadcast_connection, "Test", 4, &addr);
```

接收数据还是在

```
static void
receiver(struct simple_udp_connection *c,
```

const uip_ipaddr_t *sender_addr,

uint16_t sender_port,

const uip_ipaddr_t *receiver_addr,

uint16_t receiver_port,

const uint8_t *data,

uint16_t datalen)

uip_create_linklocal_allnodes_mcast(&addr)为：

/** \brief set IP address a to the link local all-nodes multicast address *///链路本地范围内所有节点组播地址

#define uip_create_linklocal_allnodes_mcast(a) uip_ip6addr(a, 0xff02, 0, 0, 0, 0, 0, 0, 0x0001)

/** \brief set IP address a to the link local all-routers multicast address *///链路本地范围内所有路由组播地址

#define uip_create_linklocal_allrouters_mcast(a) uip_ip6addr(a, 0xff02, 0, 0, 0, 0, 0, 0, 0x0002)

5. 单播实验步骤

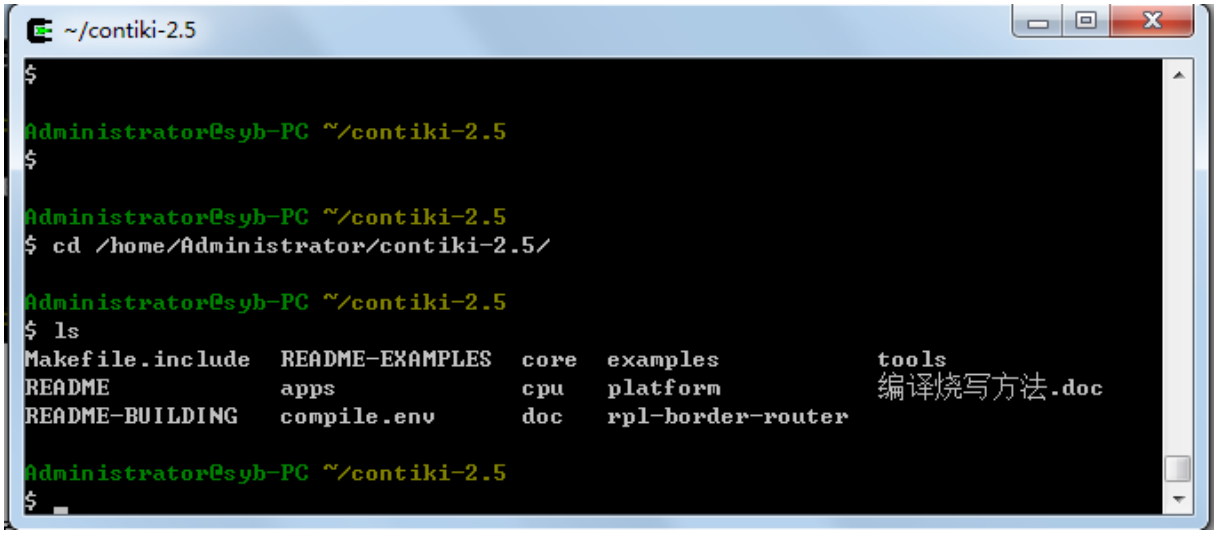
1）安装 contiki 系统源码（如果之前已经拷贝，则不用拷贝）

将产品光盘配套的 contiki 系统源码目录 contiki-2.5 拷贝至 Cygwin 开发环境安装目录的\home\Administrator 目录下。关于 Cygwin 开发环境的安装，请查看前面章节内容(1.2.2 IPv6 模块)部分，这里不再赘述。

如果 Cygwin 环境 home 目录下没有 Administrator 目录，说明您还未登陆过 Cygwin 环境，需要登陆后才可自动创建 Administrator 用户目录。

2）登陆 Cygwin

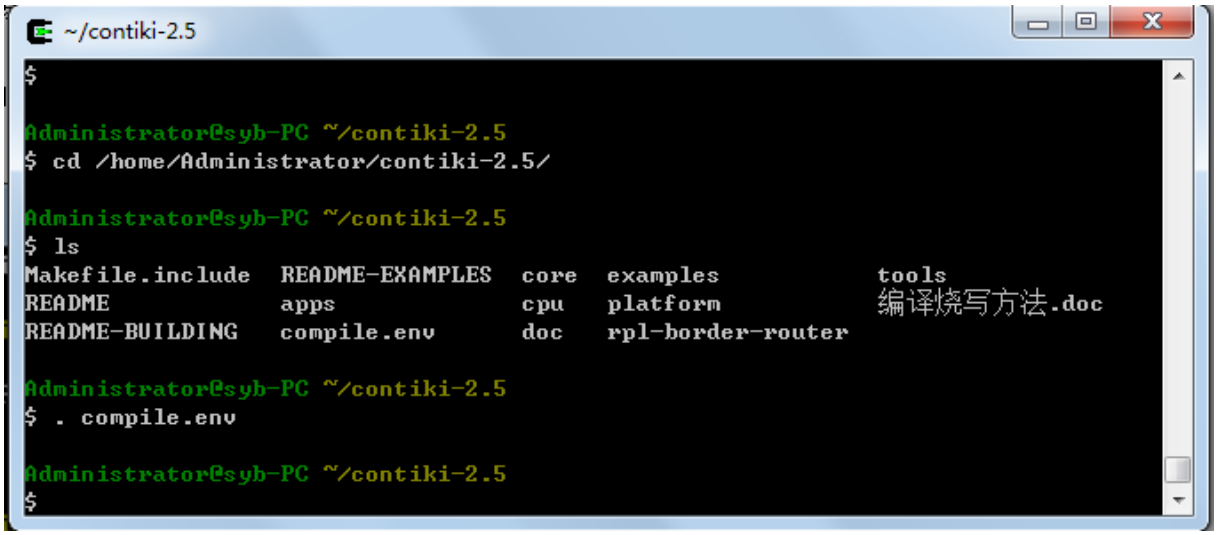
打开 Cygwin 开发环境，登陆进去。进入刚刚拷贝的 contiki-2.5 目录下，如图所示：



3）设置 IAR 编译器环境变量

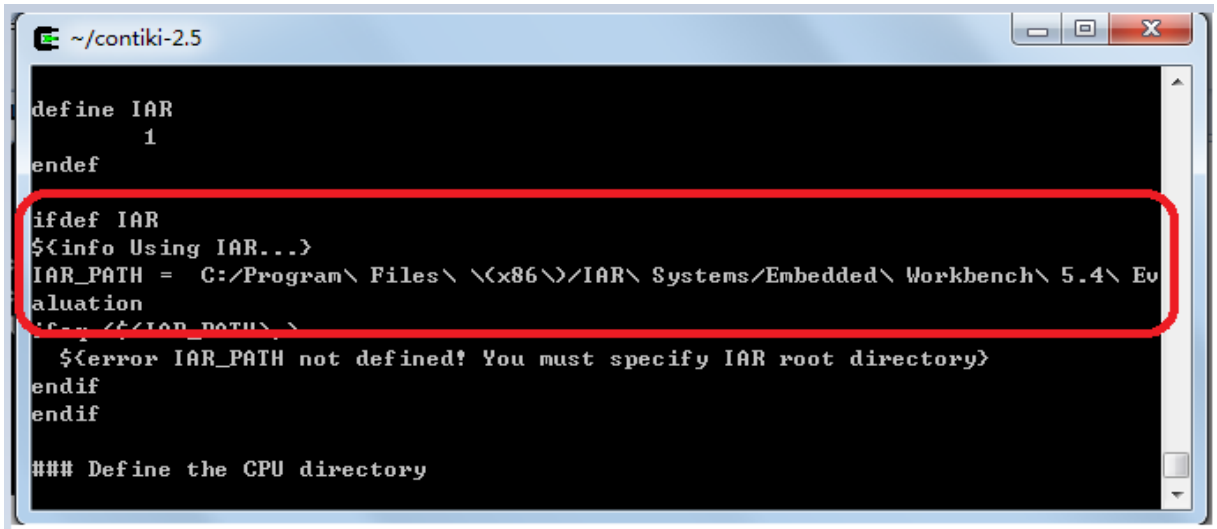
Cygwin 开发环境使用前面安装的 IAR EWARM 环境编译工具，因此要在此环境中加入 IAR 工具的安装路径，才可以使用相关编译工具对源码工程进行编译和下载。

执行. compile.env 命令（注意 .和 compile.env 中间有个空格）即可完成环境的设置



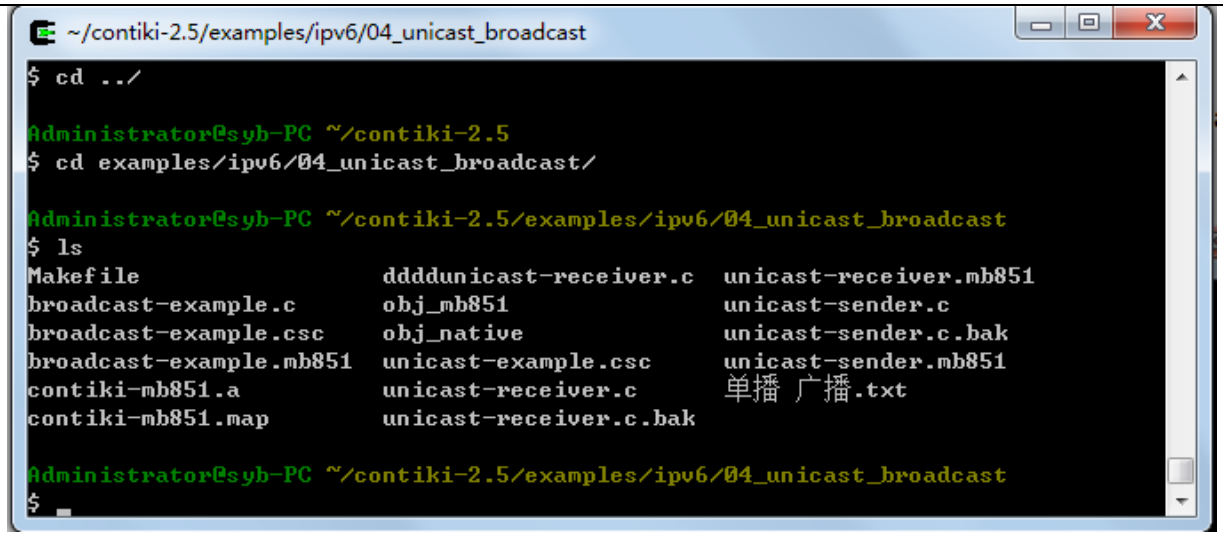
其中 compile.env 文件的内容为：export PATH=/cygdrive/c/Program\ Files/IAR\ Systems/Embedded\ Workbench\ 5.4\ Evaluation/arm/bin:$PATH（其中/cygdrive/引用 windows 各个盘的路径）。注意：具体变量用户可以根据自己环境中的 IAR 安装路径进行修改。

使用 vi 编辑 cpu/stm32w108/Makefile.stm32w108 文件，根据 IAR 实际的安装路径进行修改（如果之前已经修改则不用修改），如下：

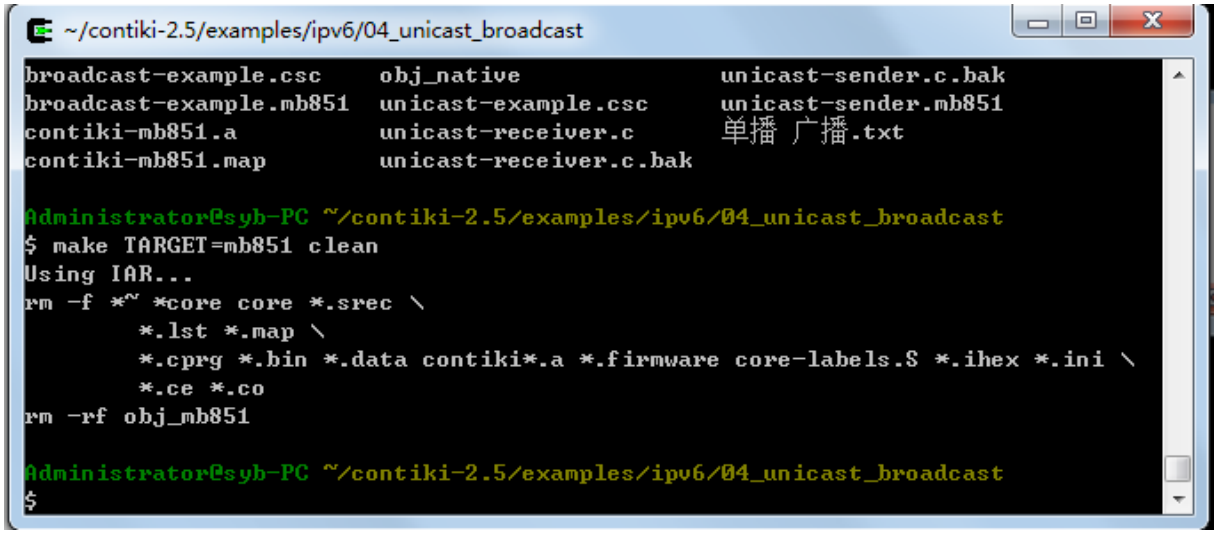


4）编译程序

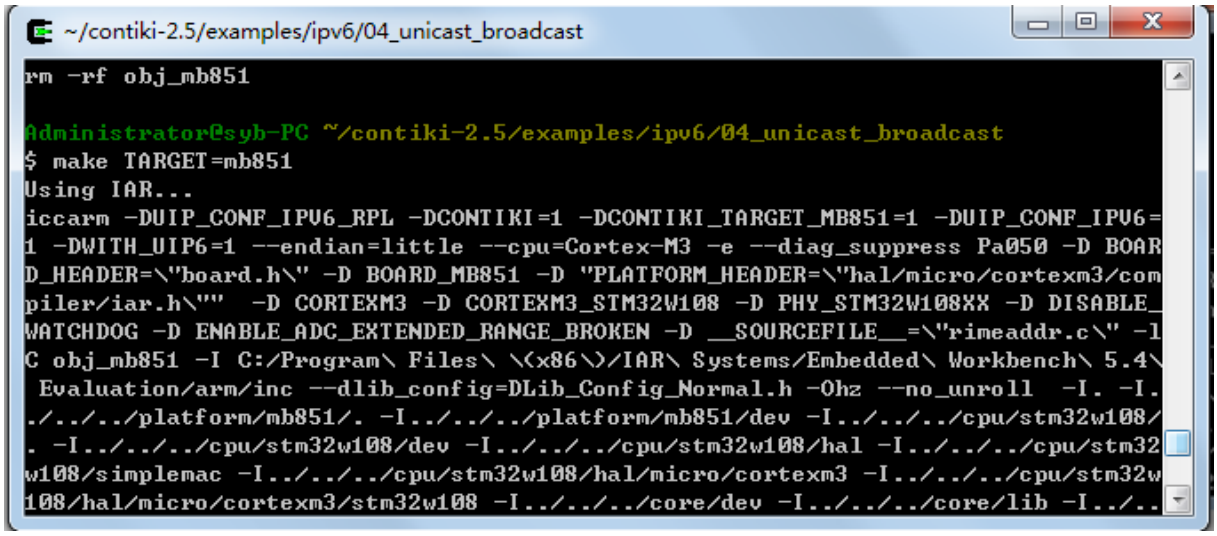
进入 contiki-2.5 目录下的 examples/ipv6 实验目录下,将产品配套实验目录 04_unicast_broadcast 文件夹拷贝至此目录下，并进入该目录。



执行 make TARGET=mb851 clean （TARGET=mb851 指定针对的平台），清除工程中间文件：



执行 make TARGET=mb851 命令进行编译：

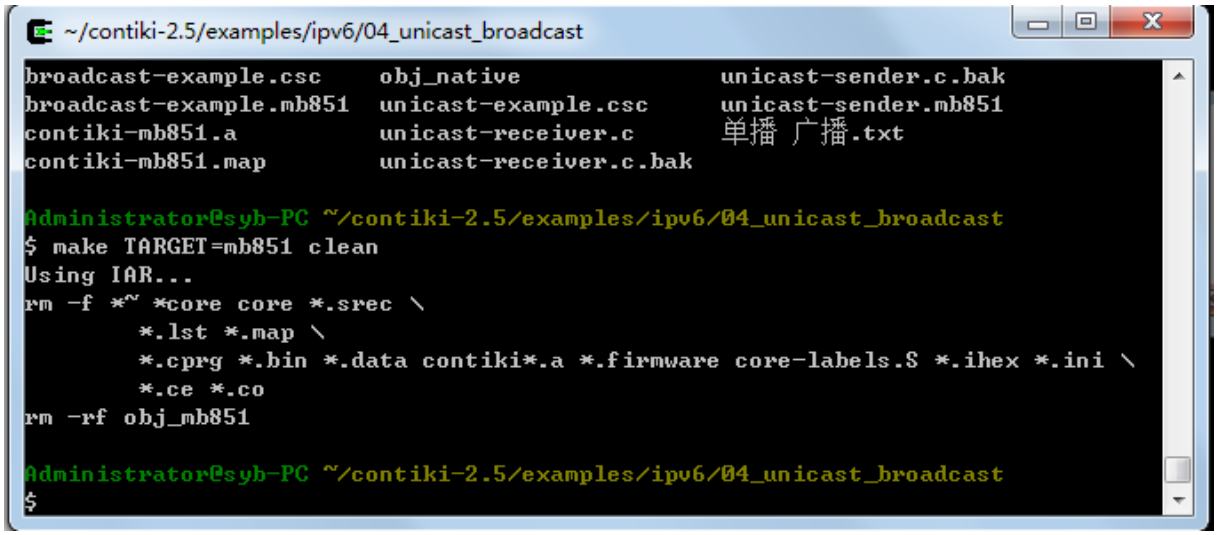


注意，一般编译前，需要 clean 以下工程，使用 make TARGET=mb851 clean 命令。

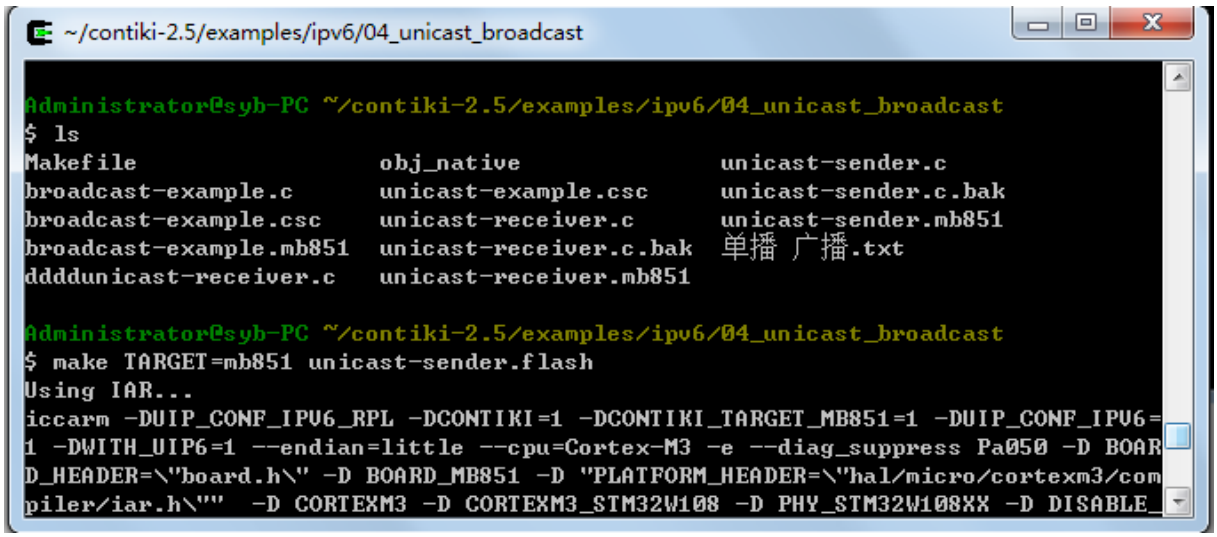
5）下载发送方程序

开启实验设备电源，使用 J-OB 仿真器和 J-OB 转接板连接 IPv6 模块，通过平台的“+”“-”按键选择目标模块，选择平台上 IPv6 根节点进行编程实验，因为其可以使用平台主板上的 RS232 串口。之后即可在 Cygwin 环境下烧写下载程序。

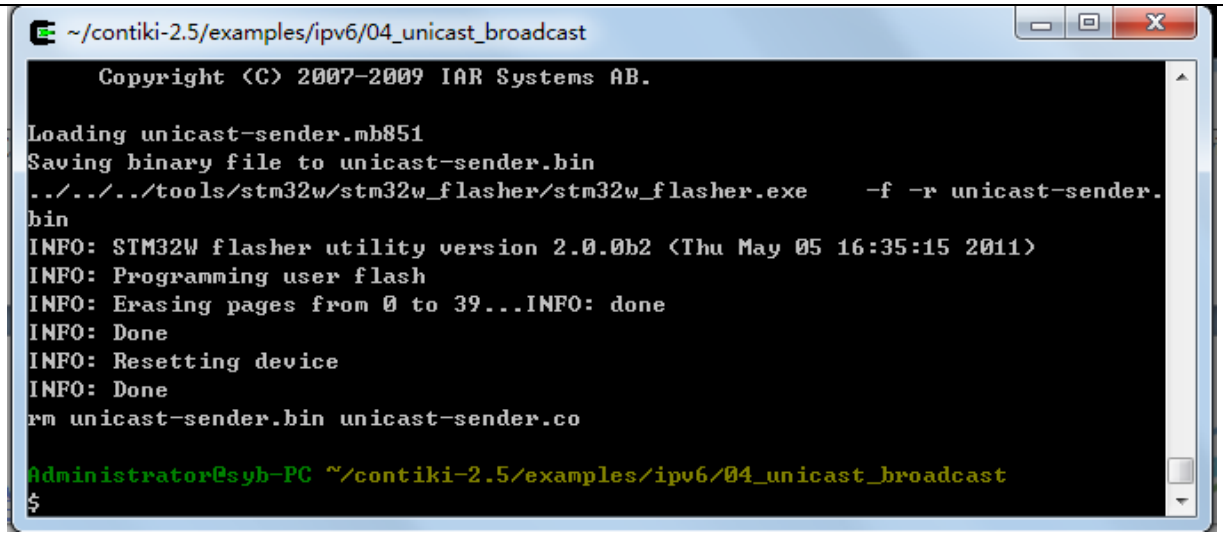
烧写之前，要先 clean：



然后运行 make TARGET=mb851 unicast-sender.flash 命令进行烧写：



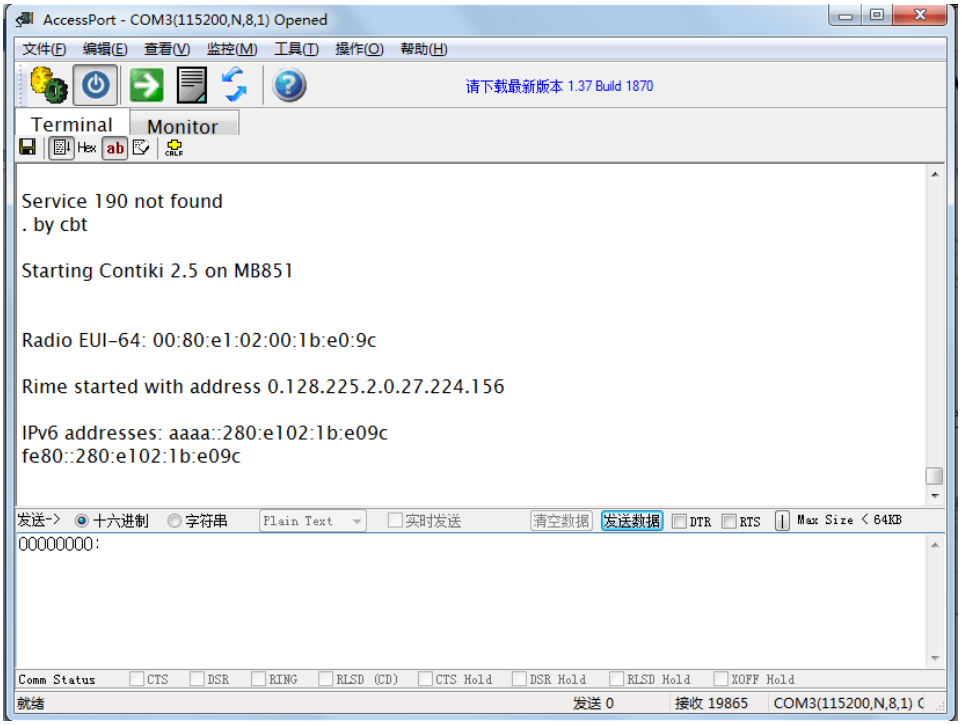
烧写完成将提示如下信息：



上述烧写过程将自动调用 IAR 相关工具完成。

烧写完毕，串口拨码跳线(0001)连接底板的 Debug UART 串口。

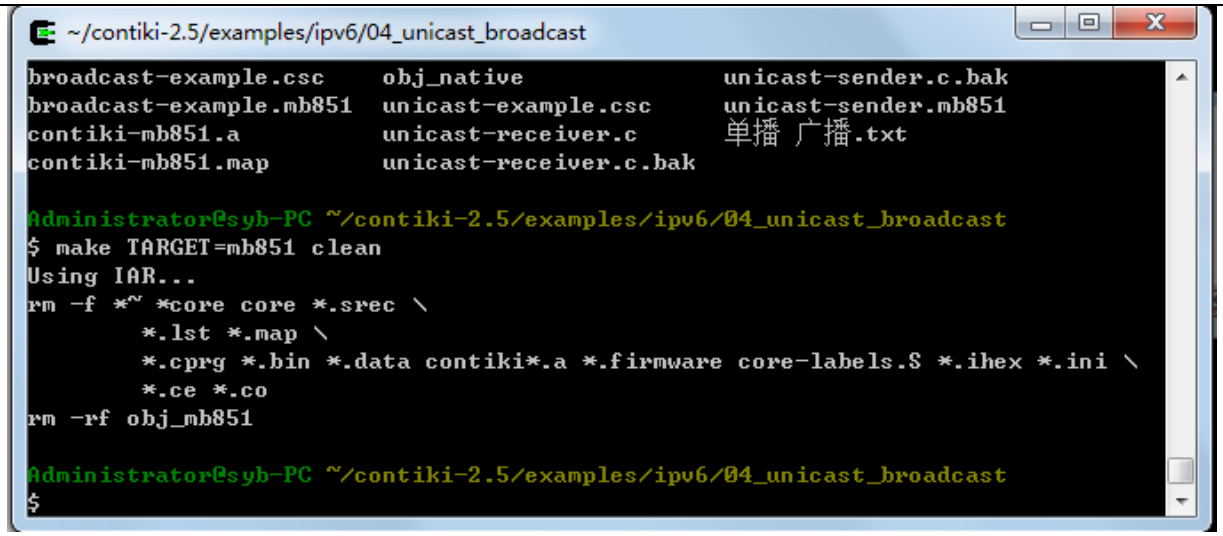
在电脑端打开串口终端软件，正确设置，波特率 115200,无校验，8 数据位，1 位停止位，无硬件流，即可查看到 IPv6 模块运行打印的信息（IPV6 模块重新上电）：



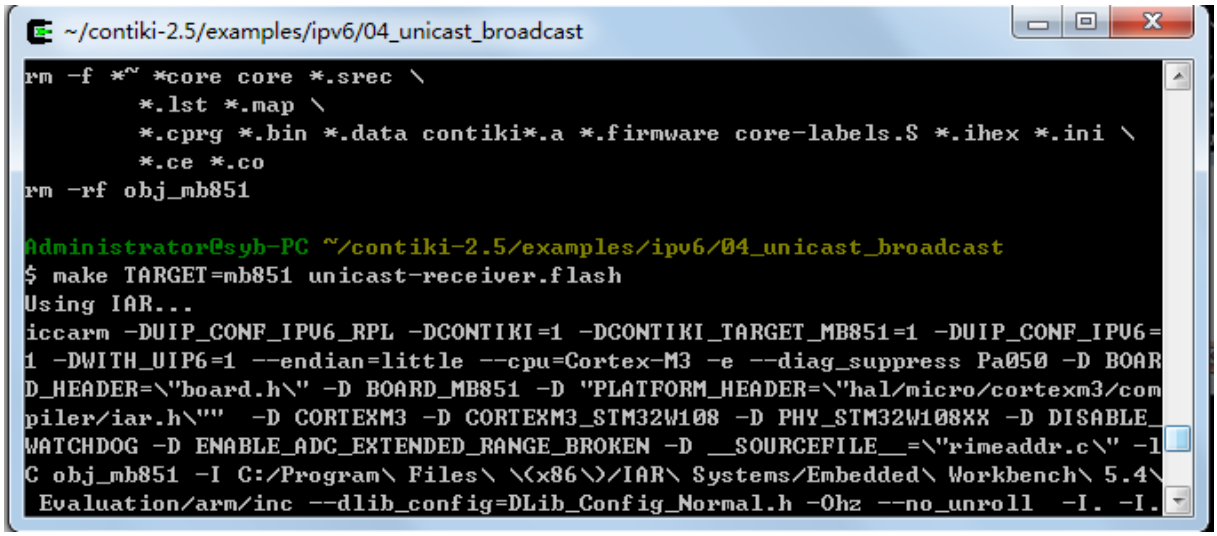
6）下载接收端程序

使用 J-OB 仿真器和 J-OB 转接板连接任意一个 IPv6 节点，通过平台的“+”“-”按键选择目标模块，之后即可在 Cygwin 环境下烧写下载程序。

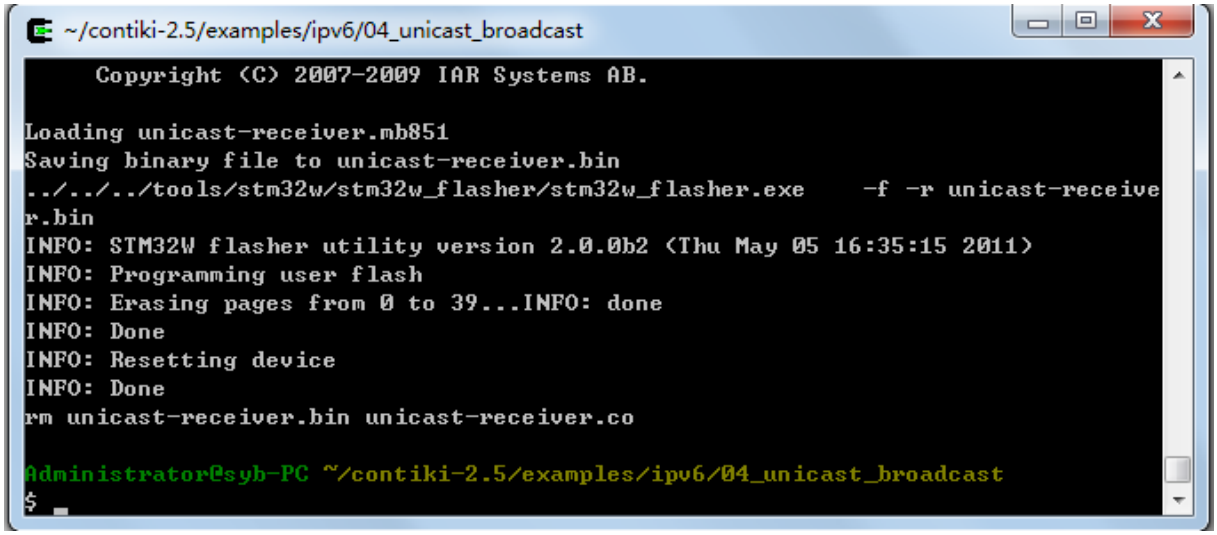
烧写之前，要先 clean：



然后运行 make TARGET=mb851 unicast-receiver.flash 命令进行烧写：



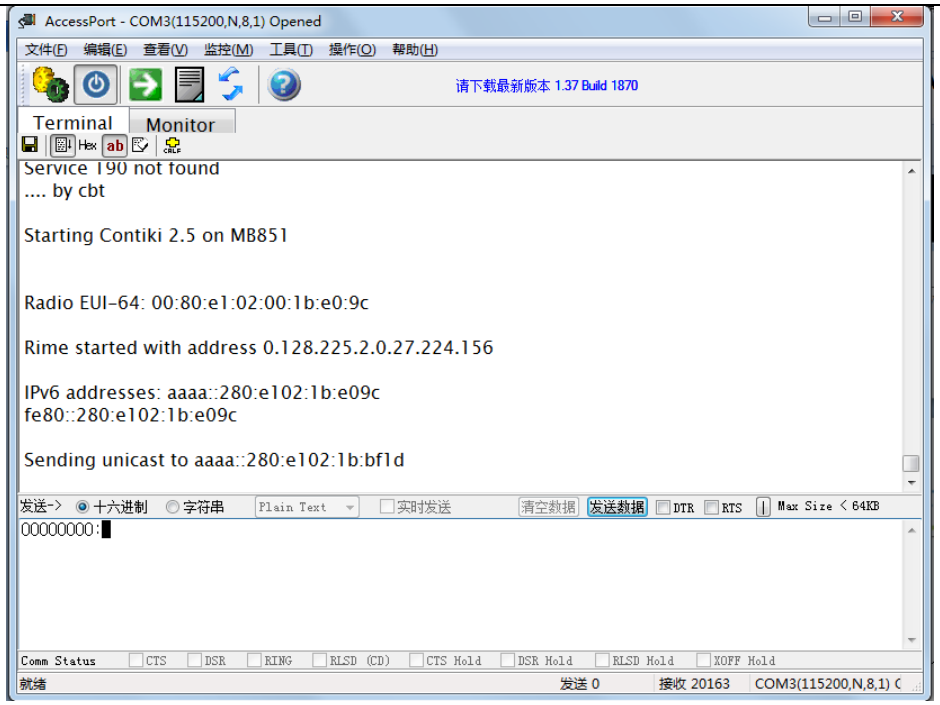
烧写完成将提示如下信息：



上述烧写过程将自动调用 IAR 相关工具完成。

7）通过服务器端串口查看两个节点通信信息

在 IPV6 根节点连接的串口上，可以看到由节点发送过来的数据包，如下所示：

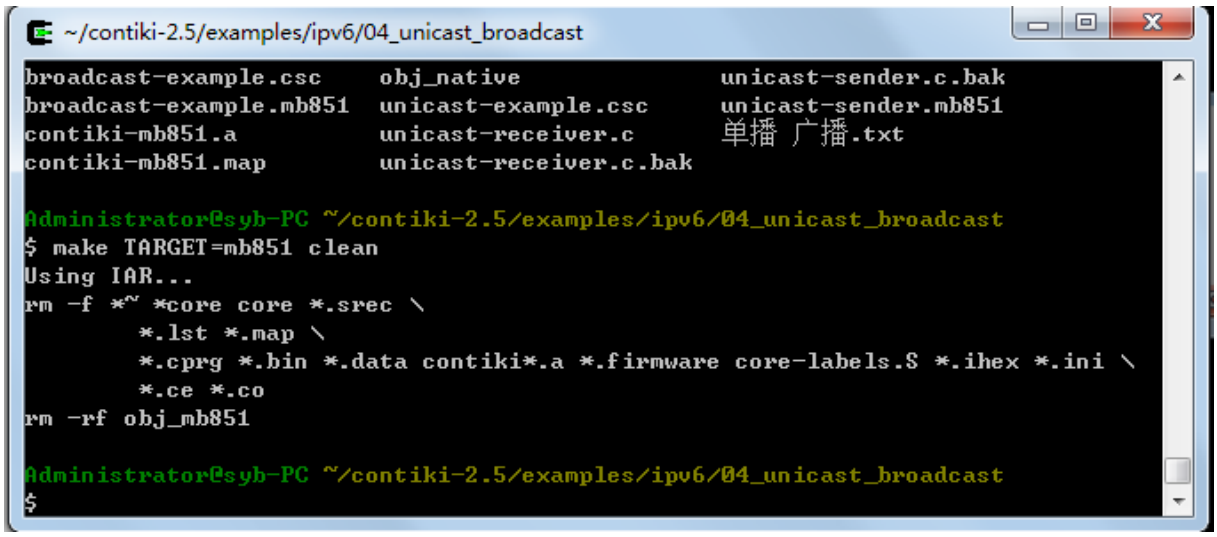


6. 多播实验步骤

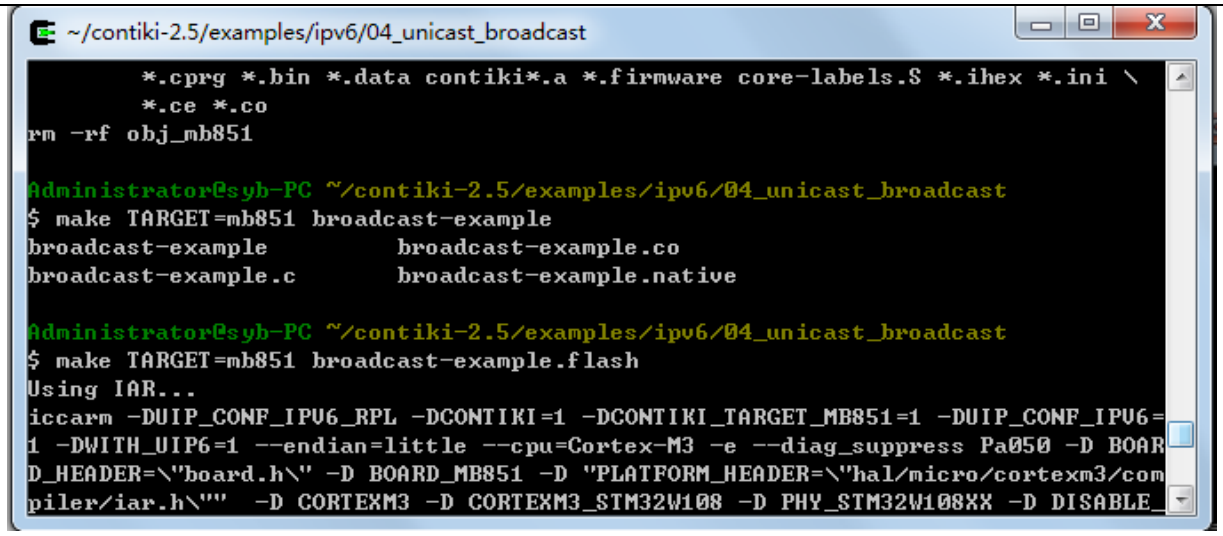
1）下载多播中的节点一

开启实验设备电源，使用 J-OB 仿真器和 J-OB 转接板连接 IPv6 模块，通过平台的“+”“-”按键选择目标模块，选择平台上 IPv6 根节点进行编程实验，因为其可以使用平台主板上的 RS232 串口。之后即可在 Cygwin 环境下烧写下载程序。

烧写之前，要先 clean：



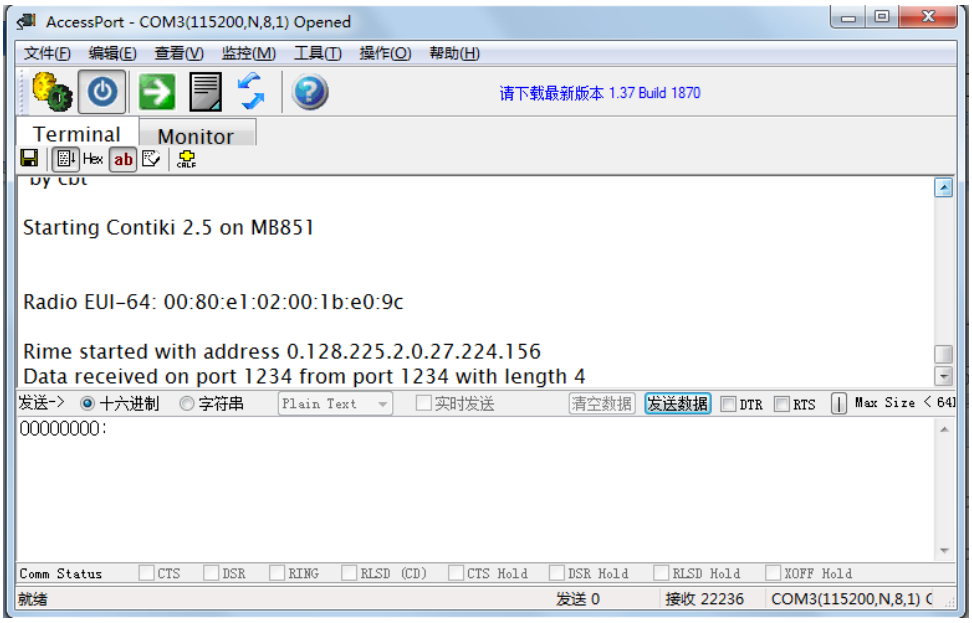
然后运行 make TARGET=mb851 broadcast-example.flash 命令进行烧写：



上述烧写过程将自动调用 IAR 相关工具完成。

烧写完毕，串口拨码跳线(0001)连接底板的 Debug UART 串口。

在电脑端打开串口终端软件，正确设置，波特率 115200,无校验，8 数据位，1 位停止位，无硬件流，即可查看到 IPv6 模块运行打印的信息（IPV6 模块重新上电）：

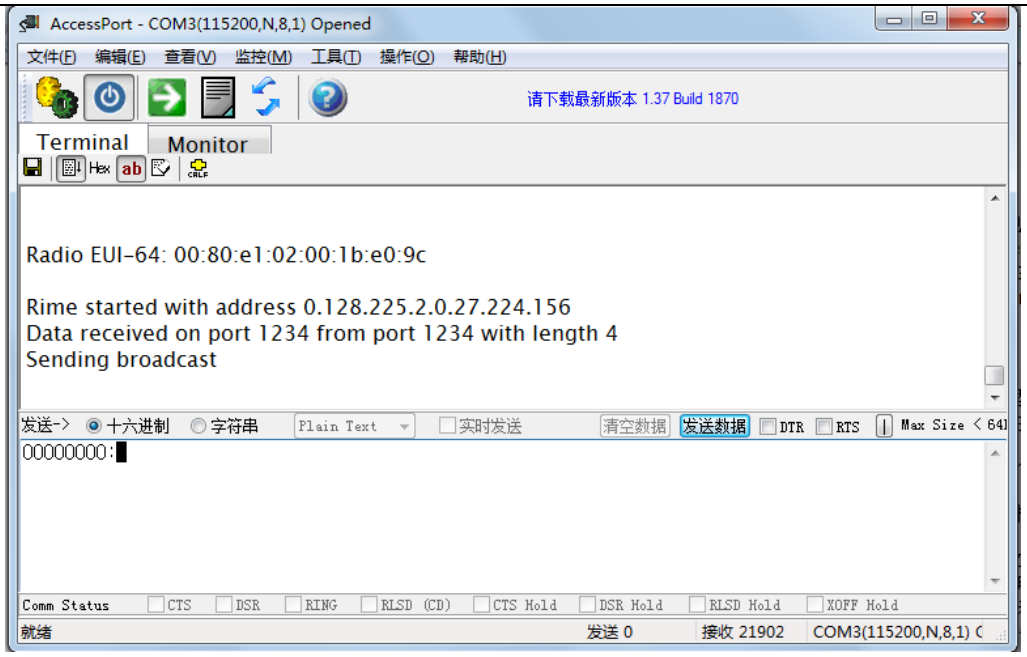


2）下载多播中的节点二

使用 J-OB 仿真器和 J-OB 转接板连接任意一个 IPv6 节点，通过平台的“+”“-”按键选择目标模块，之后即可在 Cygwin 环境下烧写下载程序。烧写步骤与“下载多播中的节点一”完全一致。

3）测试

两个节点烧写完后即可在一方看到如下信息：


AccessPort - COM3(115200,N,8,1) Opened
文件(F) 编辑(E) 查看(V) 监控(M) 工具(T) 操作(O) 帮助(H)
请下载最新版本 1.37 Build 1870
Terminal
Monitor
Radio EUI-64: 00:80:e1:02:00:1b:e0:9c
Rime started with address 0.128.225.2.0.27.224.156
Data received on port 1234 from port 1234 with length 4
Sending broadcast
发送-> 十六进制 字符串 Plain Text 实时发送 清空数据 发送数据 DTR RTS Max Size < 64
00000000:
Comm Status CTS DSR RING RLSD (CD) CTS Hold DSR Hold RLSD Hold XOFF Hold
就绪 发送 0 接收 21902 COM3(115200,N,8,1)

第八章. 无线通讯模块之单片机实验

本章主要介绍 TI 公司的 CC2530 无线通讯模块单片机部分的实验内容，从 LED、定时器、AD、串口到看门狗等涉及 15 个实验，通过本章实验学习，读者即可以迅速熟悉 ICS-IOT-CEP 平台配套的 ZigBee 模块（TI）的硬件接口，为后面通信实验打好基础。

实验一. ZigBee CC2530 入门

1. 概述

ZigBee2530 开发套件是方源智能科技面向国内高校计算机、电子、通讯和物联网相关学科和专业，在 IEEE802.15.4 标准上进行无线传感器网络技术设计与研发教学，而设计一款开发模块硬件。本手册重点在于对 ZigBee(CC2530)模块和传感器模块两部分硬件相关操作及实验进行指导和描述。

2. 适用范围

- 2.4GHz IEEE 802.15.4 标准规范
- ZigBee 2007 协议规范
- RF4CE 遥控控制系统
- ZigBee 系统/家居/楼宇自动化
- 照明系统
- 消费电子
- 工业控制及检测
- 低功耗无线传感器网络
- 教学仪器和工具
- 健康与医疗保健等

3. 功能特征

- 支持 USB 高速下载、IAR 集成开发环境；
- 支持在线下载、仿真、调试、烧写功能；
- 支持 USB 供电、电池供电方式；
- C51 编程开发，简单、方便、快捷；
- 板载 LED 指示灯、RS232 串口；
- 可外扩多种传感器模块(温湿度、红外、烟雾、光感等)；

4. 主要参数

4.1 RF 收发

- 收发频率范围 2045MHz~2480MHz；
- 测试天线 3dBi 鞭状天线；
- 输出功率 4.5(最小-8，最大 10)dBm；
- 最大功率输出距离 >300m；

4.2 功耗

- 接受模式：24mA；
- 发送模式：29mA；
- 宽电源电压范围：2.0~3.6V；

4.3 微控制器

- 高性能和低功耗的增强型 8051 微控制器内核；
- 32/64/128/256KB 系统可编程闪存、支持硬件调试；
- 8KB RAM

4.4 外设接口

- 21 个可配置通用 IO 引脚；
- 2 个同步串口；
- 1 个看门狗定时器；
- 5 通道 DMA 传输；
- 1 个 IEEE802.15.4 标准 MAC 定时器和 3 个通用定时器；
- 1 个 32MHz 睡眠定时器；
- 1 数字接收信号强度指示 RSSI/LQI 支持；
- 8 通道 12 位 AD 模数转换器，可配分辨率，内置电压、温度传感器检测；
- 1 个 AES 安全加密协处理器；

4.5 工作温度

- 工作环境温度：0ºC~+85ºC；
- 储藏环境温度：-40ºC~+125ºC；

5. IAR 实验开发环境

5.1 建立模板工程样例

ZigBee2530 开发套件实验部分，使用的软件开发环境为 IAR Embedded Wordbench for MCS-51。本节将介绍如何使用该 IAR 环境搭建配套实验工程。后续实验的工程建立方法参照本节设置建立，将不再赘述。关于 IAR 的详细说明文档请浏览 IAR 官方网站或软件安装文件夹下 8051\doc 里的支持文档。

以下将通过一个简单的 LED 闪灯测试程序工程带领用户逐步熟悉 IAR for 51 实验开发环境。

1）建立新工程

打开 IAR 软件，默认进入建立工作区菜单，我们先选择-取消，进入 IAR IDE 环境。点击 Project 菜单，选择 Greate New Project ...如图 5.1.1 所示：

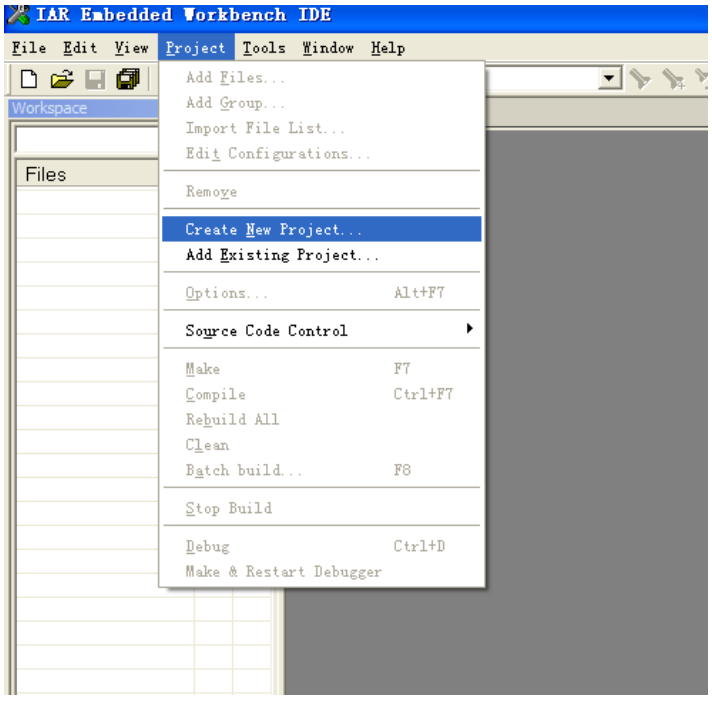


图 5.1.1 建立一个新工程

弹出图 5.1.2 建立新工程对话框，确认 Tool chain 栏已经选择 8051，在 Project templates：栏选择 Empty project 点击下方 OK 按钮。

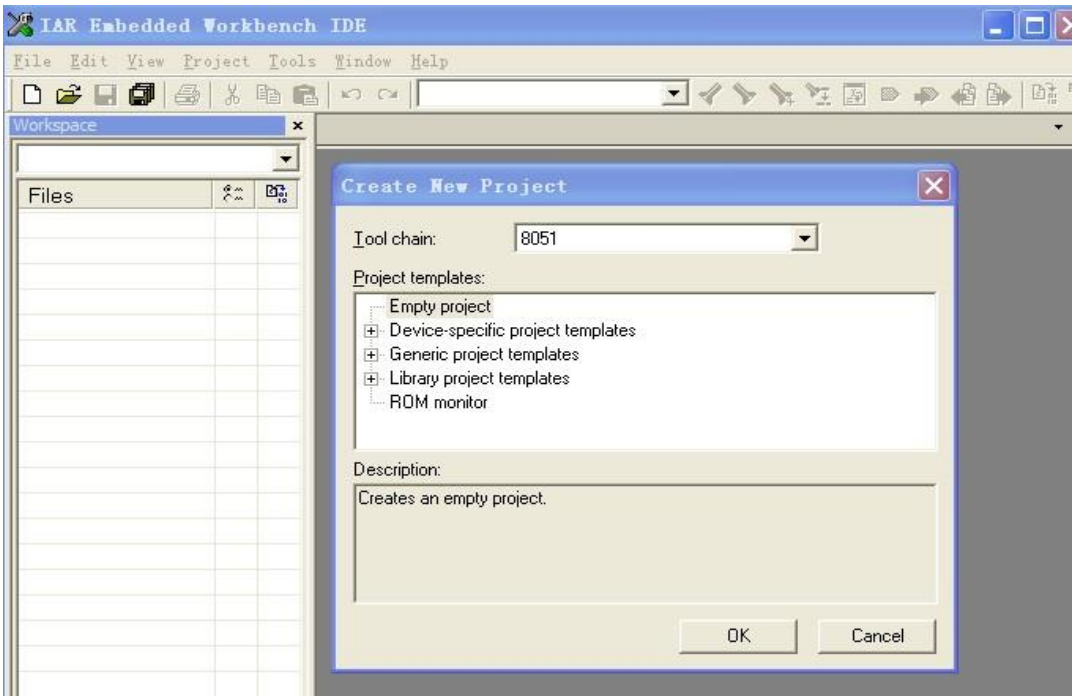


图 5.1.2 选择工程类型

点击右上角快捷方式，创建新文件夹。在计算机相应目录下，创建工程目录，本例创建了 test_iar 目录，用来存放工程，进入到创建的 test_iar 文件夹中，更改工程名，如 test 点击 Save ，这样便建立了一个空的工程。

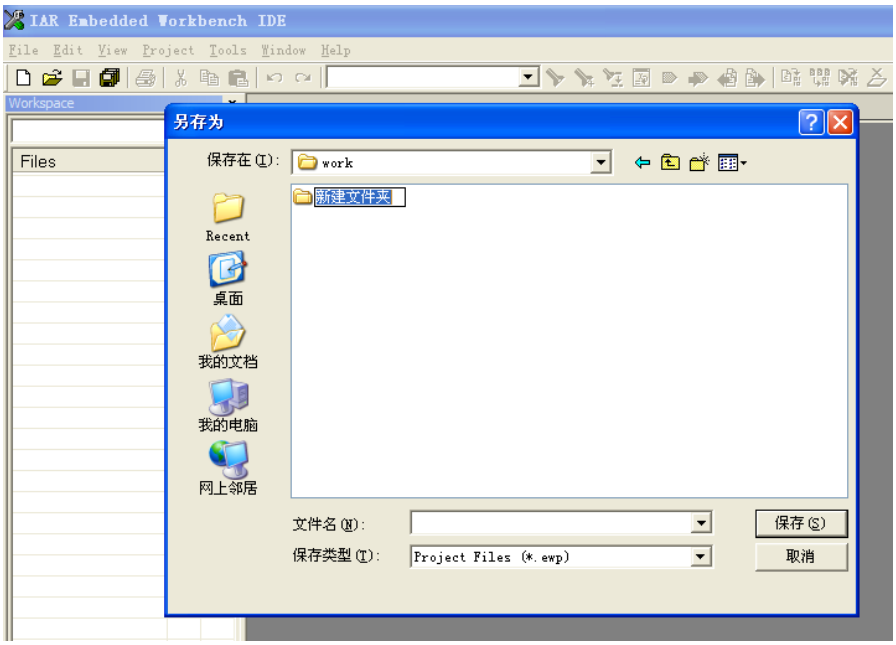


图 5.1.3 创建工程目录

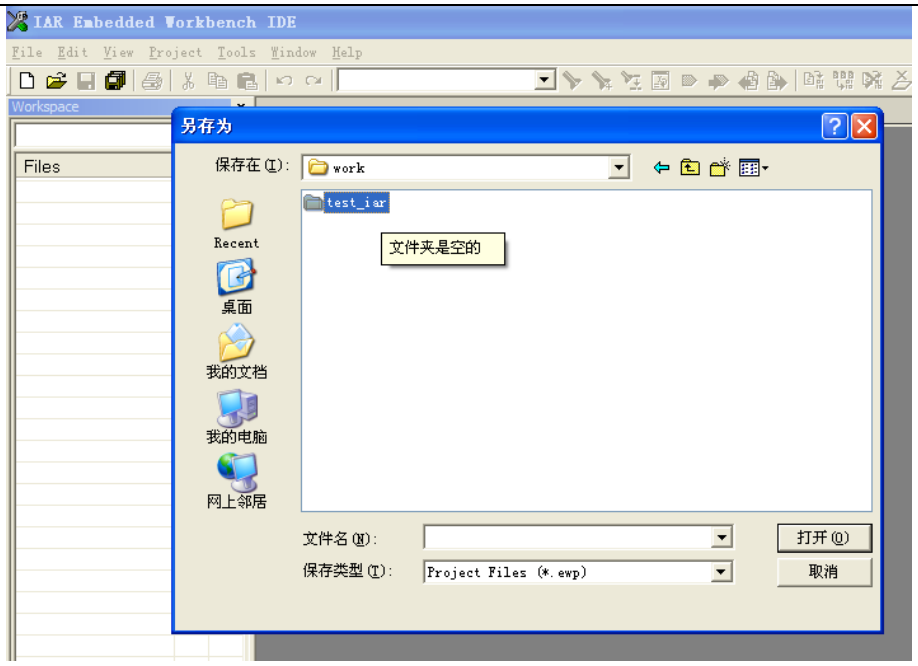


图 5.1.4 创建工程目录 test_iar

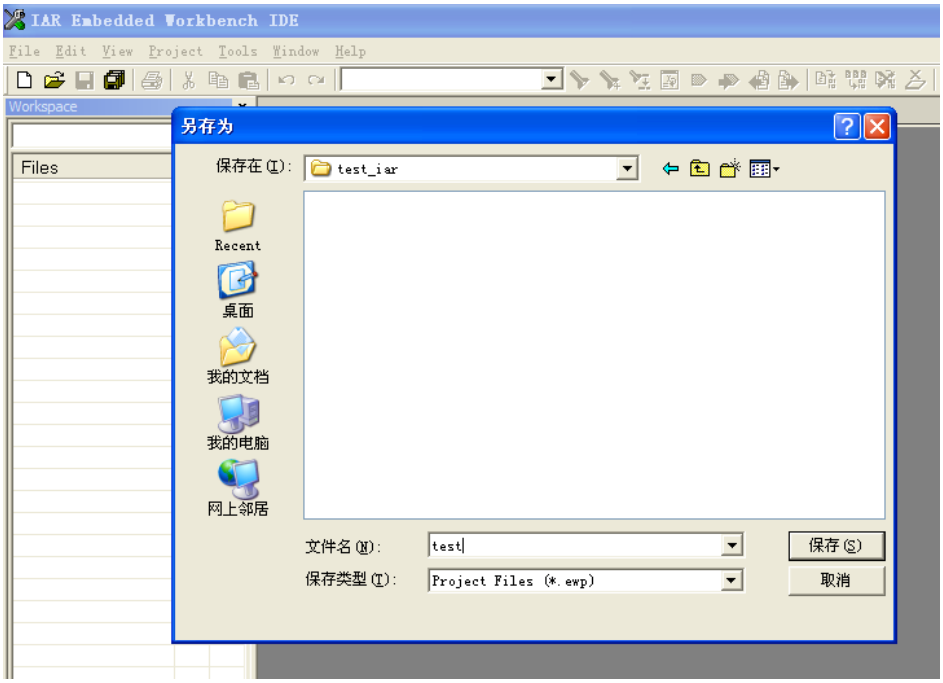


图 5.1.5 创建工程目录配置文件

这样工程就出现在工作区窗口中了。

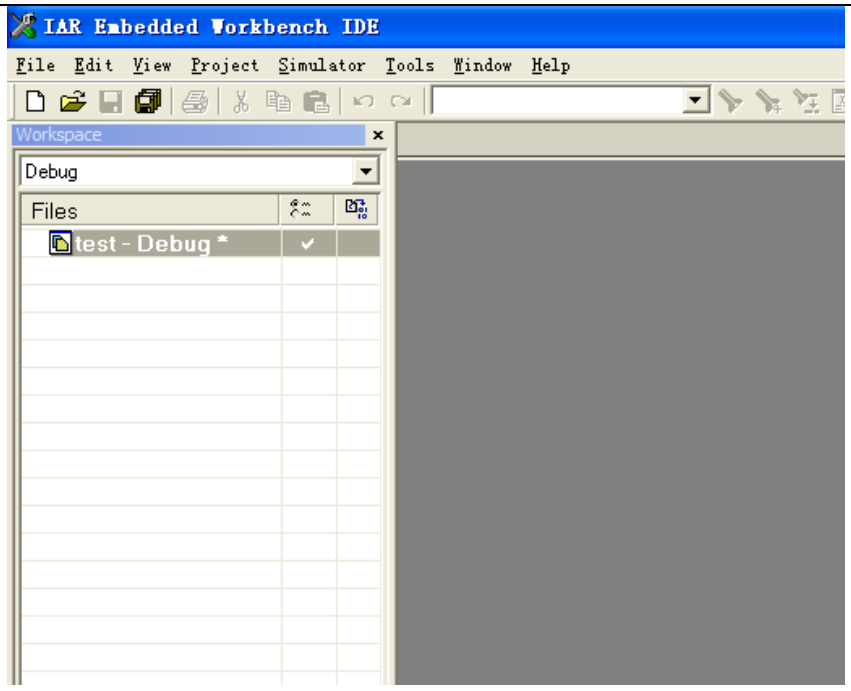


图 5.1.6 创建工程加入工作区

系统产生两个创建配置：调试和发布，在这里我们只使用 Debug。

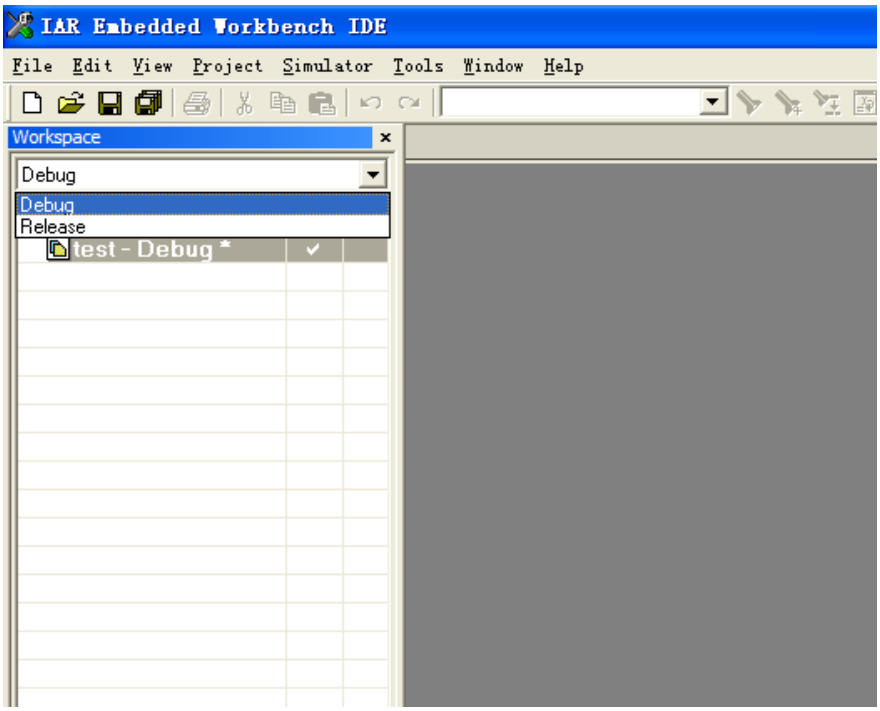


图 5.1.7 选择 debug 模式工程

项目名称后的星号指示修改还没有保存。

选择菜单 File\Save\Workspace ,保存工作区文件，并指明存放路径，这里把它放到新建的工程 test_iar 目录下。点击 Save 保存工作区。

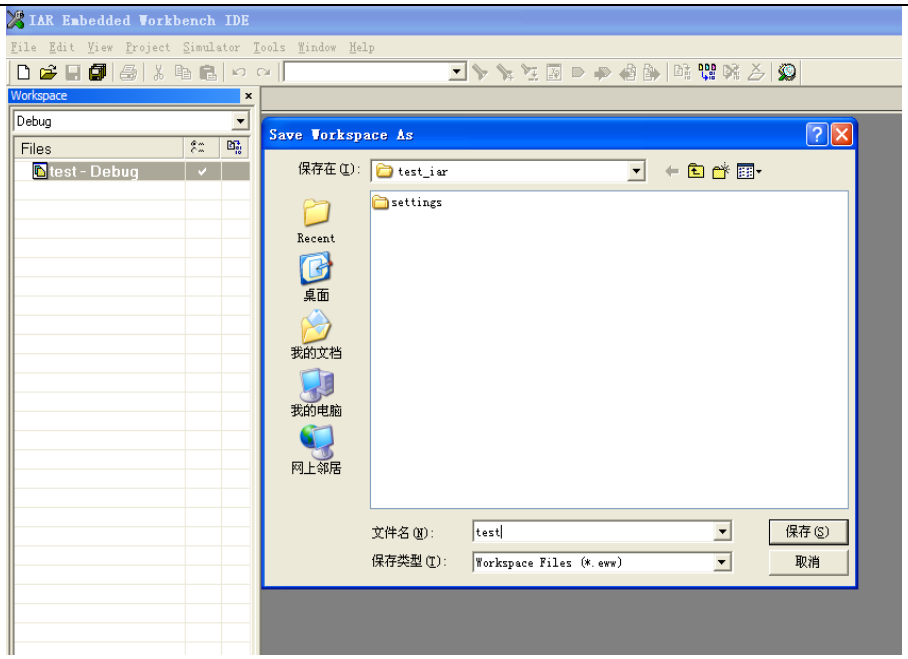


图 5.1.8 保存工作区

2）添加工程文件

选择菜单 Project\Add File 或在工作区窗口中，在工程名上点右键，在弹出的快捷菜单中选择 Add File ，弹出文件打开对话框，选择需要的文件点击 打开 退出。

如没有建好的程序文件也可点击工具栏上的  按钮或选择菜单 File\New\File 新建一个空文本文件。

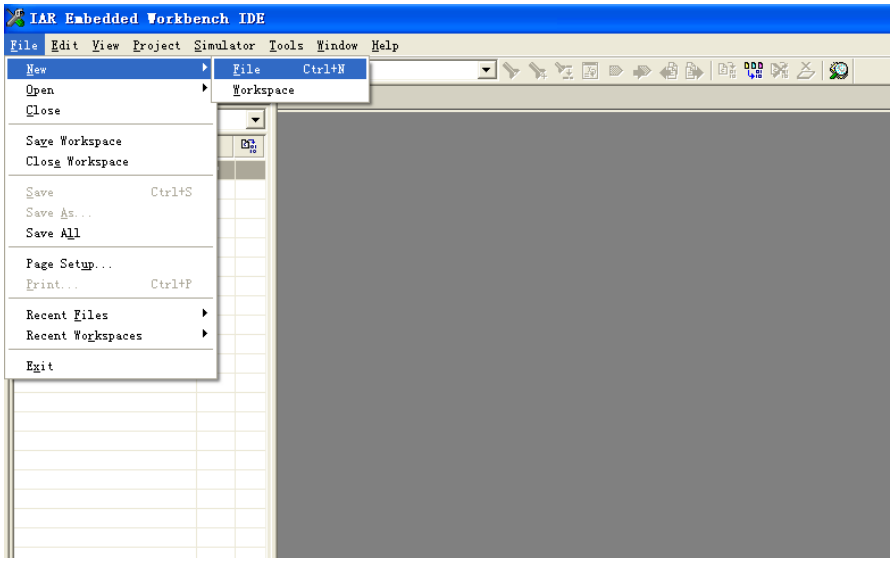


图 5.2.1 新建文件

向文件里添加如下代码：

```
void Delay(unsigned char n)
{
    unsigned char i;
    unsigned int j;
    for(i = 0; i < n; i++)
```

```
    for(j = 1; j < 1000; j++) ;
}

void main(void)
{
    P1SEL = 0x00;            //P1.0 为普通 I/O 口
    P1DIR = 0x3;             //P1.0 P1.1 输出
    while(1)
    {
        P1_1 = 1;
        Delay(10);
        P1_0 = 0;
        Delay(10);
        P1_1 = 0;
        Delay(10);
        P1_0 = 1;
        Delay(10);
    }
}
```

选择菜单 File\Save 弹出保存对话框,填写文件名为 test.c,点击保存。

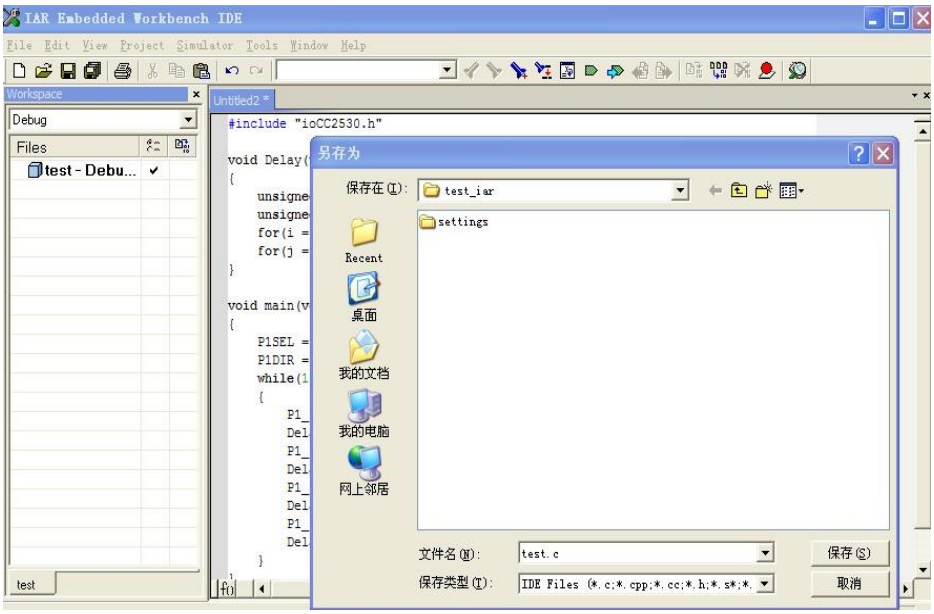


图 5.2.2 保存新建文件

按照前面添加文件的方法将 test.c 添加到当前工程里，在工程中右键添加文件：

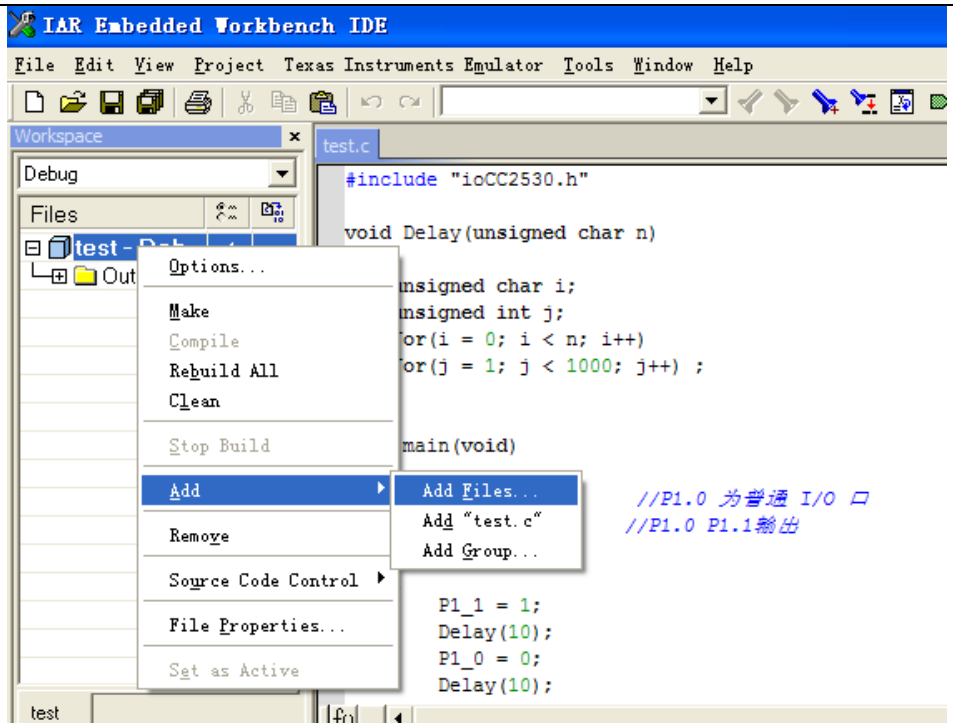


图 5.2.3 将新建文件加入工程

选择刚刚编写好的文件 test.c。

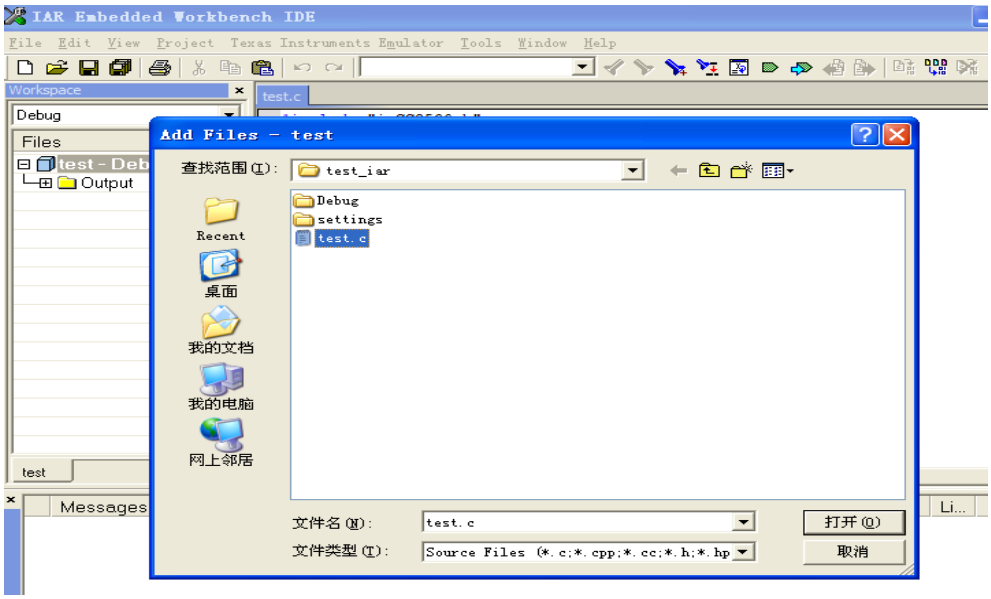


图 5.2.4 将新建文件 test.c 加入工程

完成的结果如下图：

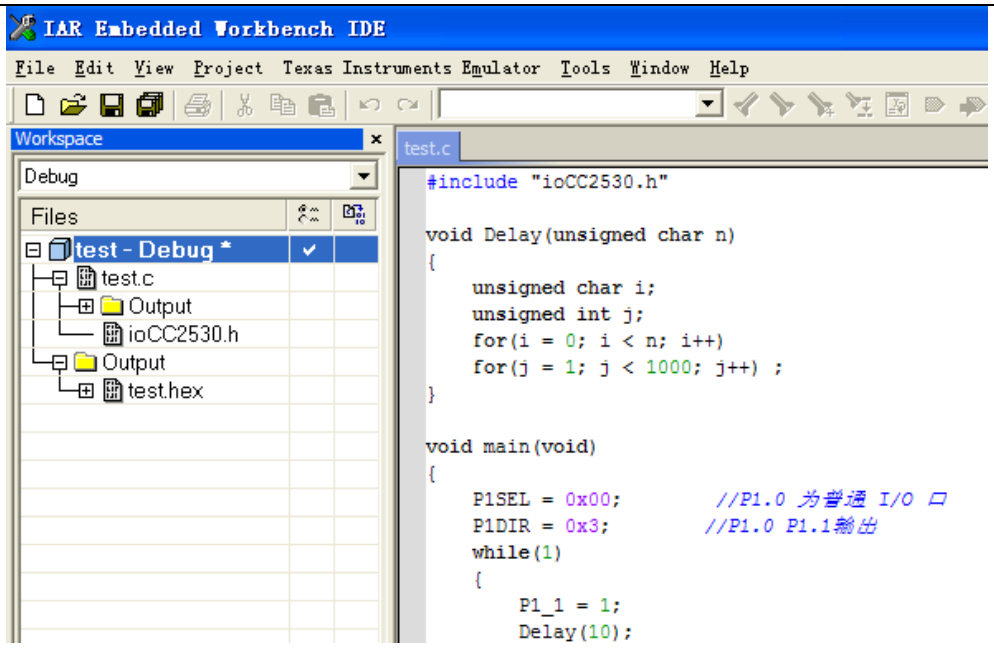


图 5.2.5 加入新文件后的工程

3）配置工程选项

选择 Project 菜单下的 Options... 配置与 CC2530 相关的选项。

General Options--Target 标签：

按下图配置 Target ， 选择 Code model 为 Near 和 Data model 为 Large ， Calling convention 为 XDATA stack reentrant 以及其它参数。

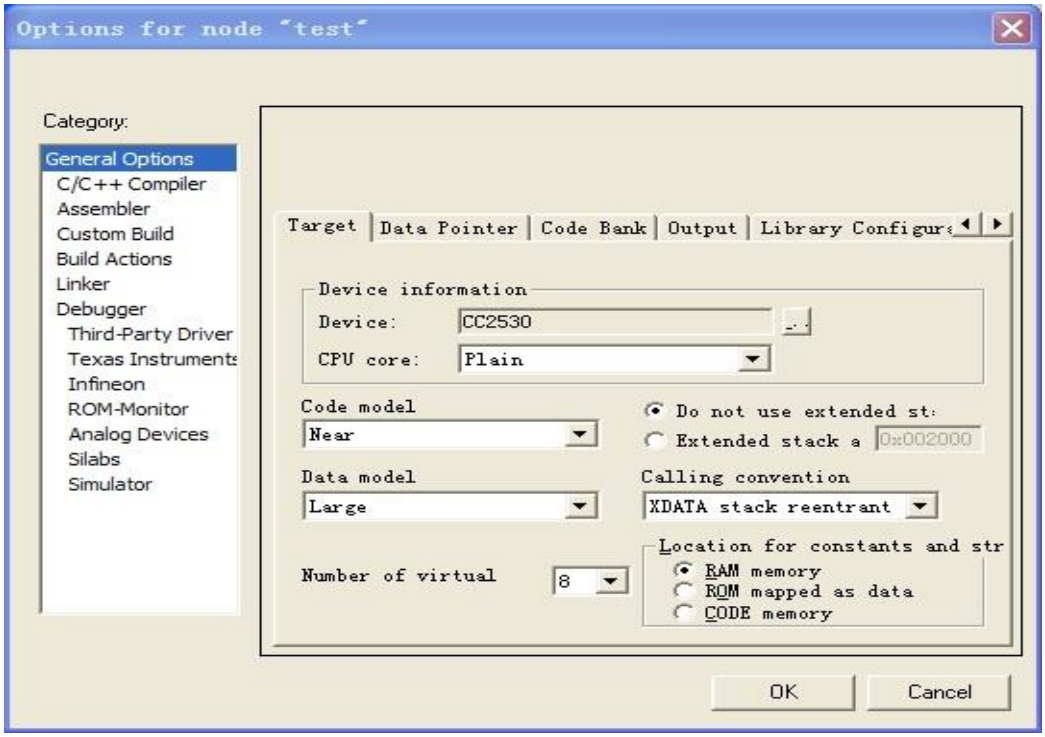


图 5.3.1 target 标签配置

点击 Derivative information 栏右边的 按钮，选择程序安装位置如这里是 IAR Systems\Embedded Workbench 5.3 Evaluation version\8051\config\devices\Texas Instruments 下

的文件 CC2530.i51。

DataPointer 标签：选择数据指针数 1 个，16 位.默认即为该配置。

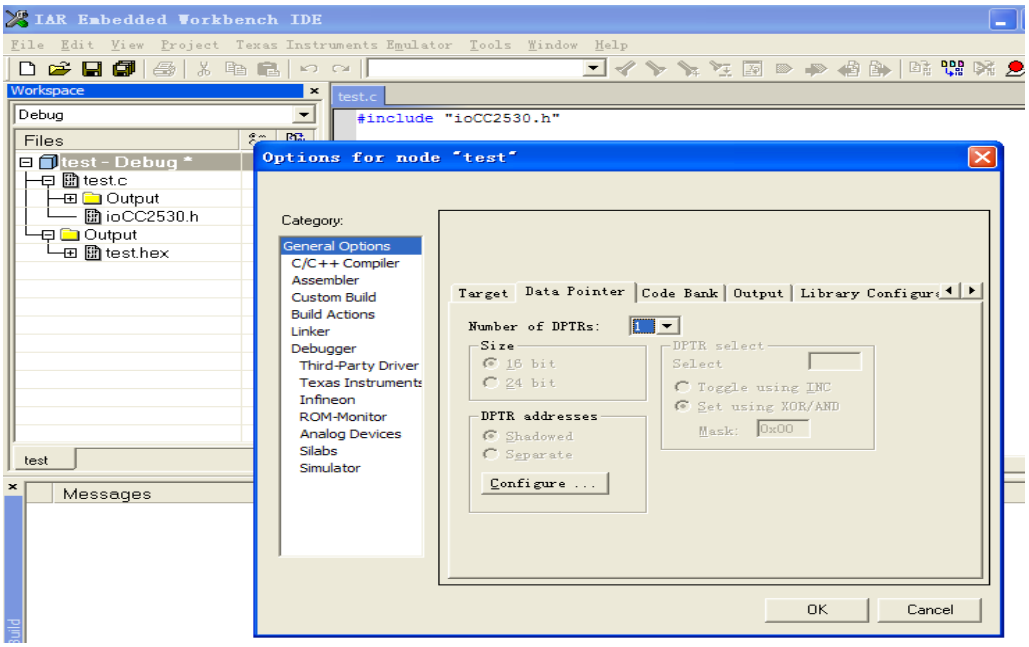


图 5.3.2 数据指针选择

Stack/Heap 标签：改变 XDATA 栈大小到 0x1FF。

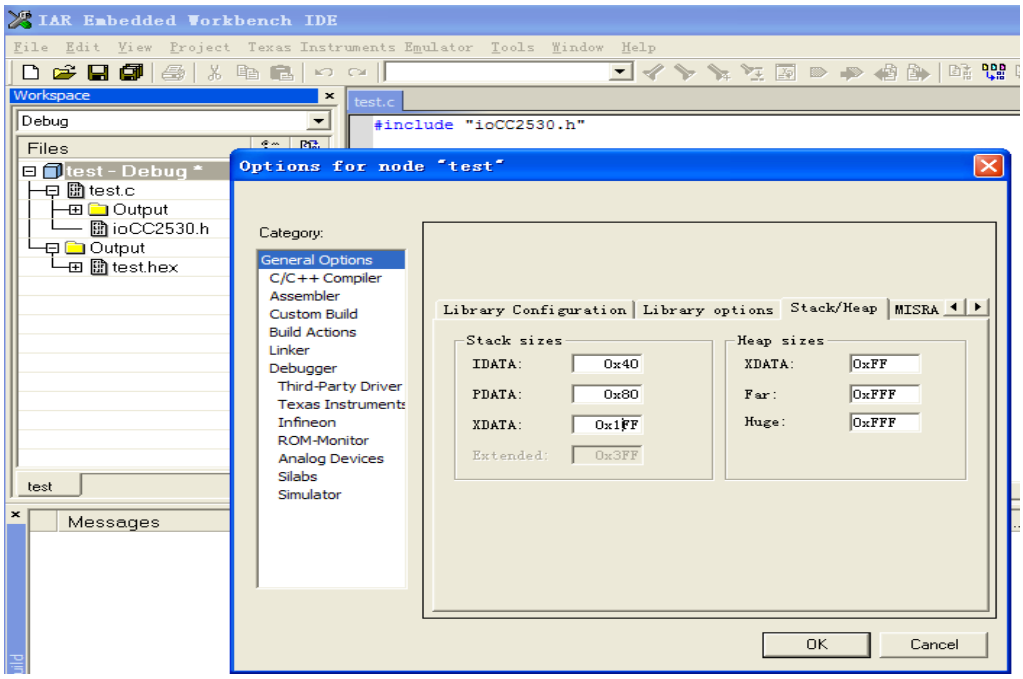


图 5.3.3 Stack/Heap 设置

Linker 选项，Output 标签：选中 Override default 可以在下面的文本框中更改输出文件名。如果要用 C-SPY 进行调试，选中 format 下面的 Debug information for C-SPY。

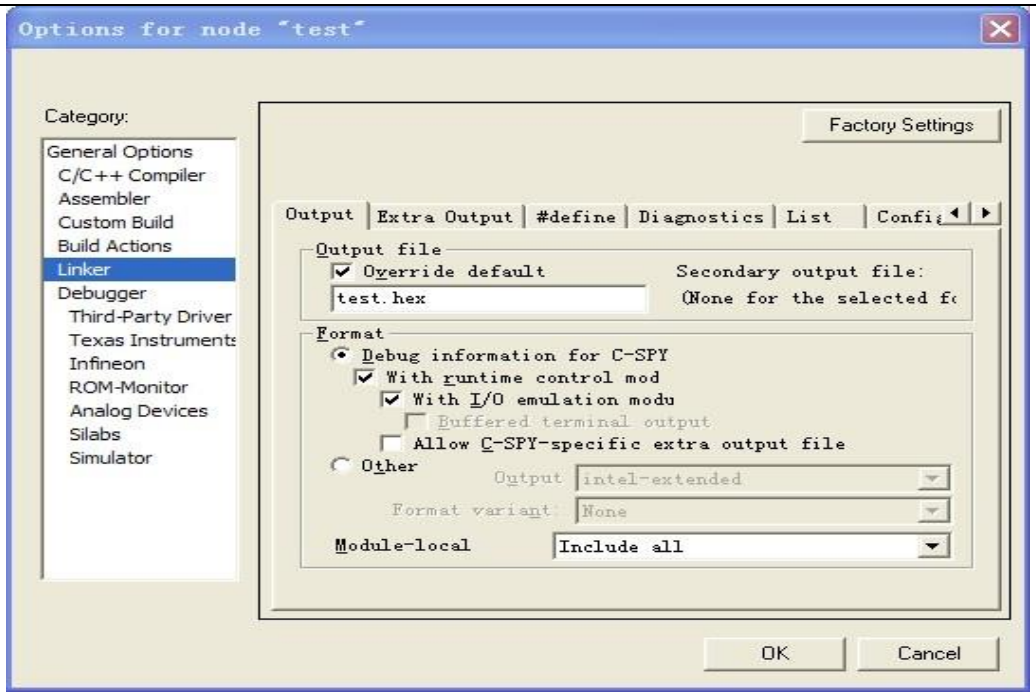
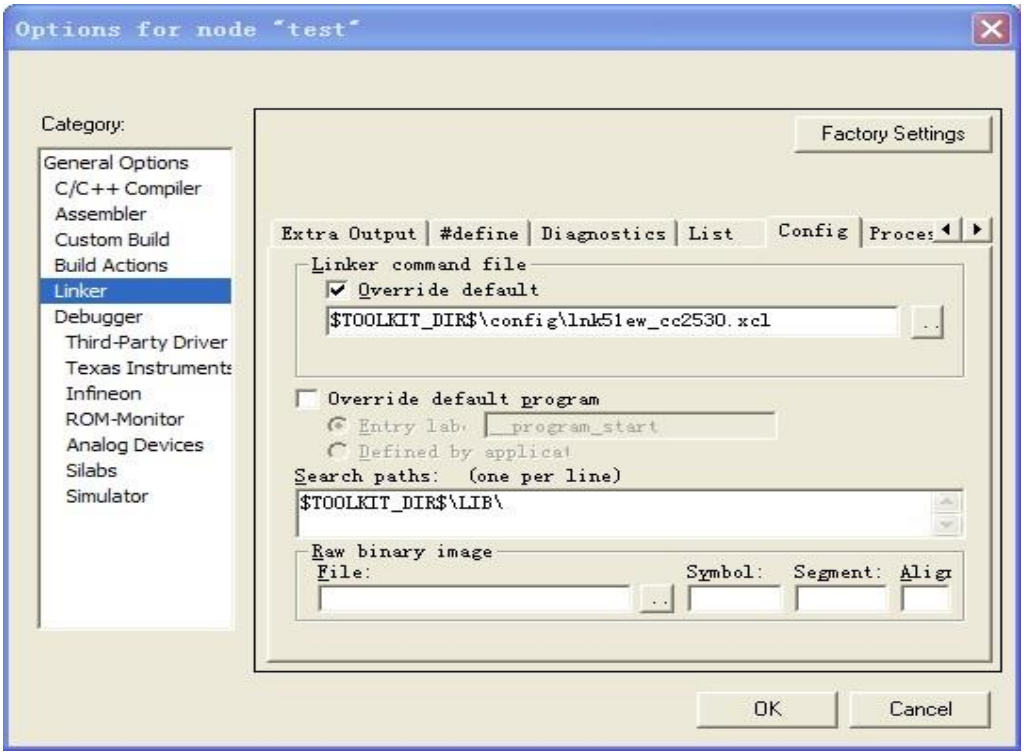


图 5.3.4 输出文件设置

Config 标签：点击 Linker command file 栏文本框右边的 按钮。选择正确的连接命令文件 lnk51ew_cc2530.xcl，如图：



Debugger：在 Setup 标签按下图设置 driver 选项为 Texas Instruments:

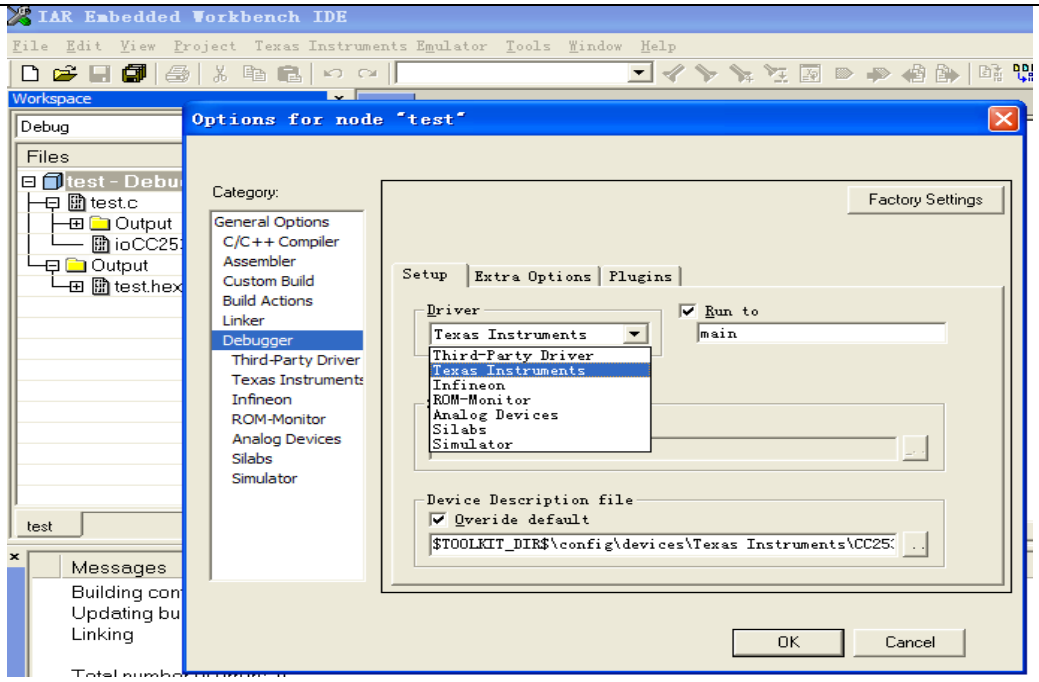


图 5.3.6 Debugger 配置

在 Device Description file 选择 CC2530.ddf 文件，其位置在程序安装文件夹下如 C:\Program Files\IAR Systems\Embedded Workbench 5.3 Evaluation version\8051\config\devices\Texas Instruments

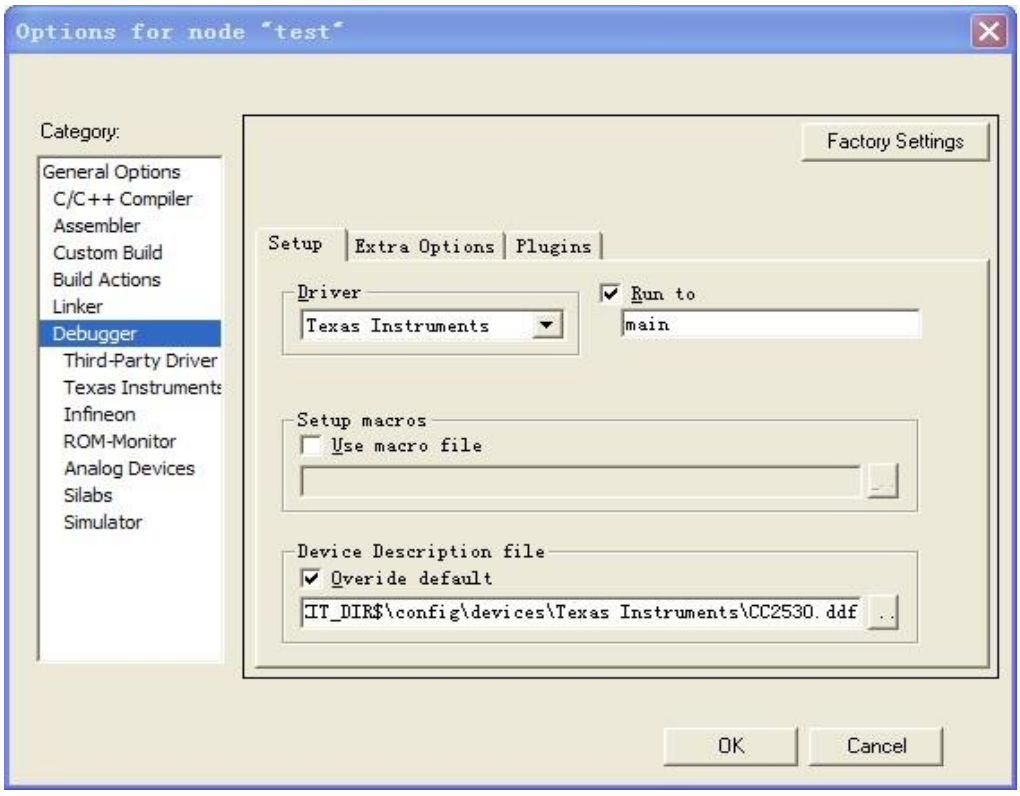


图 5.3.7 配置调试器

4）编译和链接

选择 Project\Make 或 按 F7 键编译和连接工程。

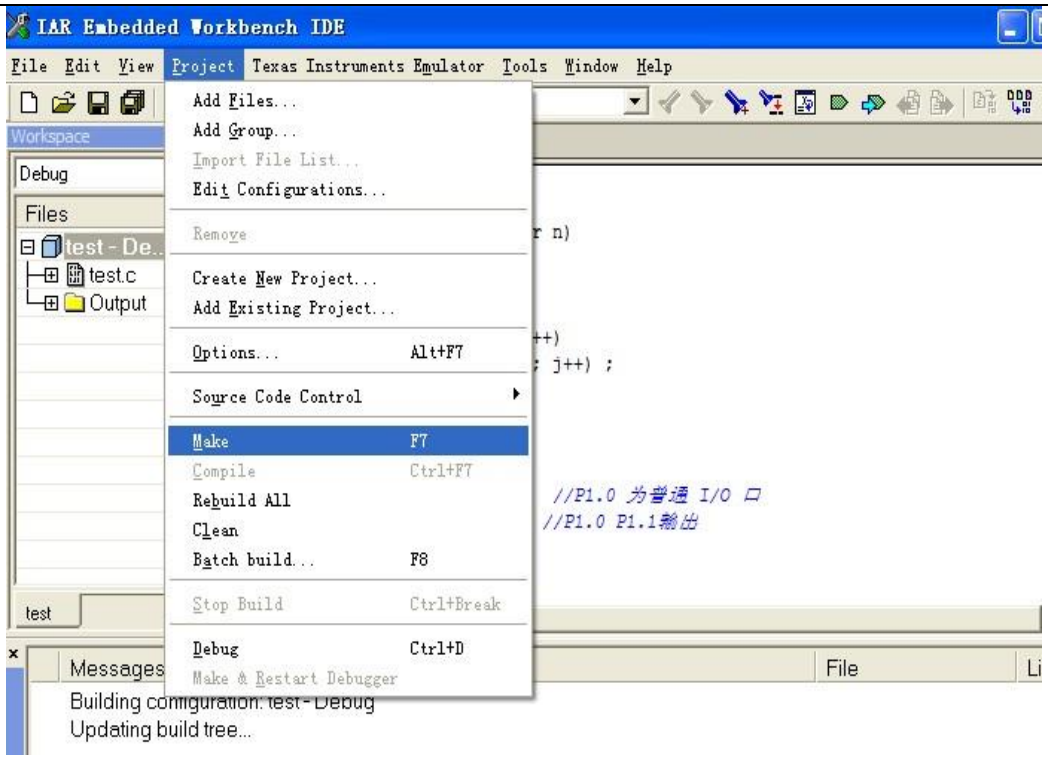


图 5.4.1 编译和链接工程

5.2 下载和调试

- **安装仿真器驱动**

安装仿真器前确认 IAR Embedded Workbench 已经安装。

将系统配套的 USB 接口仿真器一端连接到 PC 机上，另一端 20Pin 排线连接到平台主板的 JLink JTAG 口中。通过主板上的选择按键，选择即将编程下载的模块(有指示灯)，注意模块要先上电打开。

1）打开 CC2530 光盘 tools 目录，安装 Setup_SmartRFstudio_6.11.6.exe.

图 5.2.1.1

2）自动安装：

将仿真器通过开发系统附带的 USB 电缆连接到 PC 机，在 Windows XP 系统下，系统找到新硬件后提示如下对话框，选择 自动安装软件 ，点下一步。

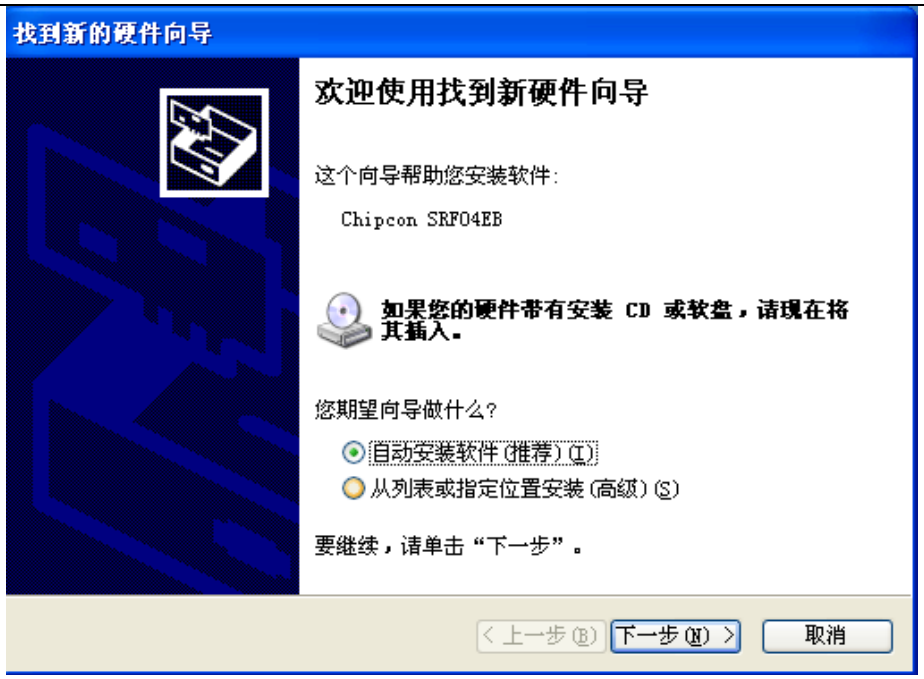


图 5.2.1.2

向导会自动搜索并复制驱动文件到系统。系统安装完驱动后提示完成对话框，点击 完成 退出安装。

图 5.2.1.3

3）仿真器驱动检查

双击桌面上的 SmartRF_Studio 图标，打开上步安装的 SmartRFID Studio 软件，插上 ZigBee DeBuger 仿真器到计算机（建议还是上步安装时的那个计算机的 USB 口）

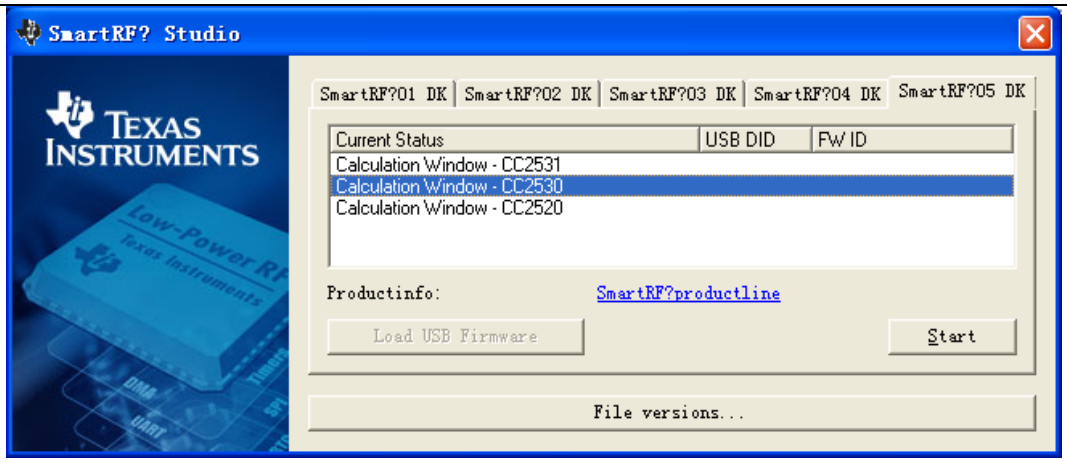


图 5.2.1.4 测试是否安装成功

上图表示仿真器驱动安装成功。

♦ **调试和运行**

1）选择菜单 Project\Debug 或按快捷键 CTRL+D 进入调试状态，也可按工具栏上的 按钮进入调试。

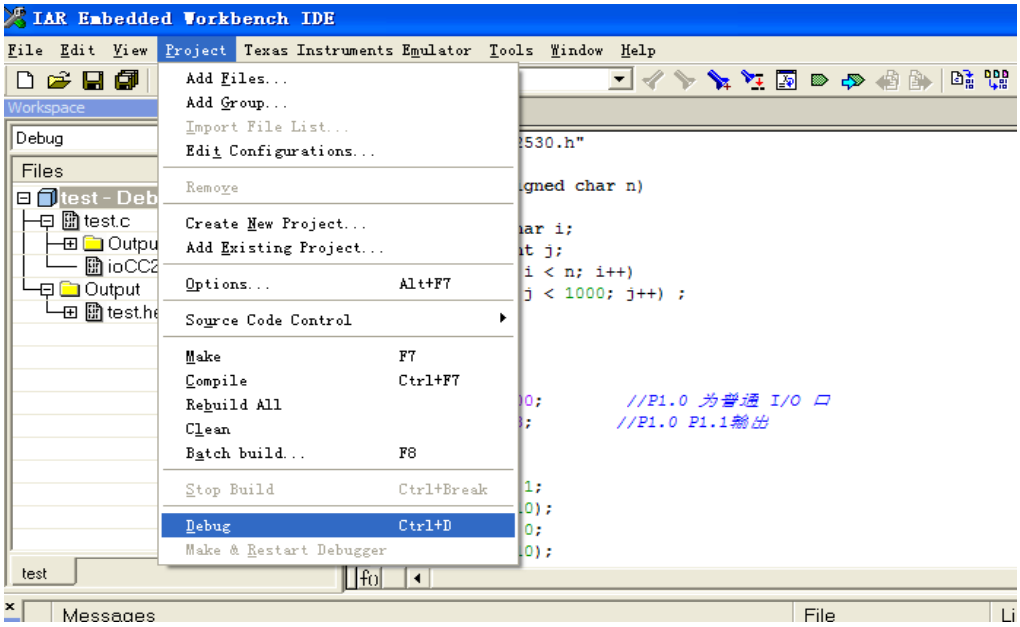


图 5.2.2.1 进入调试

查看源文件语句：

Step Into 执行内部函数或子进程的调用

Step Over 每步执行一个函数调用

Next statement 每次执行一个语句

这些命令在工具栏上都有对应的快捷键。

2）查看变量：

C-SPY 允许用户在源代码中查看变量或表达式，可在程序运行时跟踪其值的变化。使用自动窗口

选择菜单 View\Auto ,开启窗口。自动窗口会显示当前被修改过的表达式。连续步进观察 j 的值的变化情况。

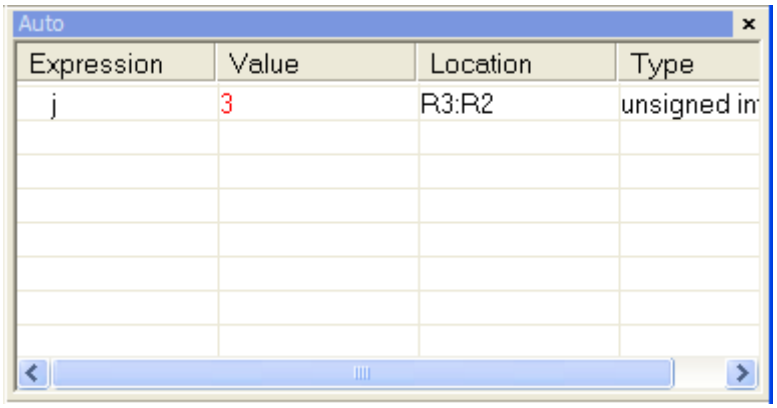


图 5.2.2.2 查看变量

3）设置监控点：

使用 Watch 窗口来查看变量。选择菜单 View\Watch ，打开 Watch 窗口。点击 Watch 窗口中的虚线框，出现输入区域时键入 j 并回车。也可以先选中一个变量将其从编辑窗口拖到 Watch 窗口。

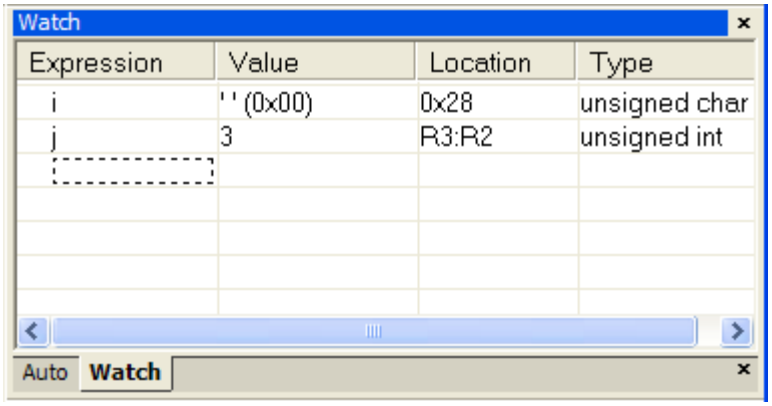


图 5.2.2.3 设置观察变量

单步执行，观察 i 和 j 的变化。如果要在 Watch 窗口中去掉一个变量，先选中然后点击键盘上的 Delete 键或点右键删除。

4）设置并监控断点：

使用断点最便捷的方式是将其设置为交互式的，即将插入点的位置指到一个语句里或靠近一个语句，然后选择 Toggle Breakpoint 命令。

在 i++语句出插入断点：在编辑窗口选择要插入断点的语句，选择菜单 Edit\Toggle Breakpoint。或者在工具栏上点击  按钮。

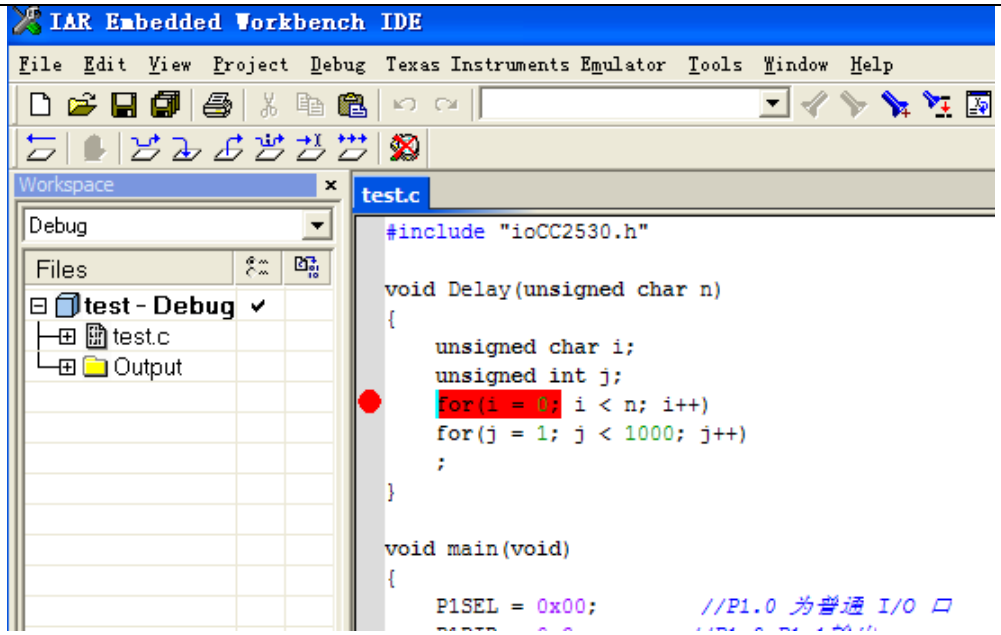


图 5.2.2 4 设置断点

这样在这个语句设置好一个断点，用高亮表示并且在左边标注一个红色的圆显示有一个断点存在。可选择菜单 View\Bradkpoint 打开断点窗口，观察工程所设置的断点。在主窗口下方的调试日志 Debug Log 窗口中可以查看断点的执行情况。如要取消断点，在原来断点的设置处再执行一次 Toggle Breakpoint 命令。

5）反汇编模式：

在反汇编模式，每一步都对应一条汇编指令，用户可对底层进行完全控制。

选择菜单 View\Disassembly，打开反汇编调试窗口，用户可看到当前 C 语言语句对应的汇编语言指令。

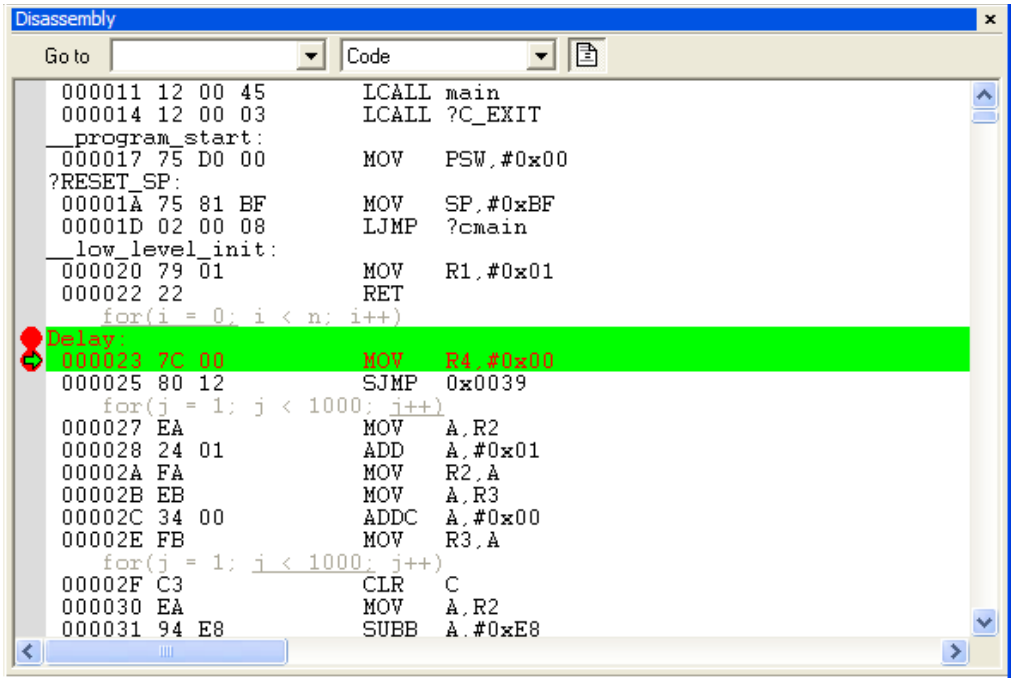


图 5.2.2.5 反汇编窗口

6）监控寄存器：

寄存器窗口允许用户监控并修改寄存器的内容。选择菜单 View\Regisster，打开寄存器窗口。

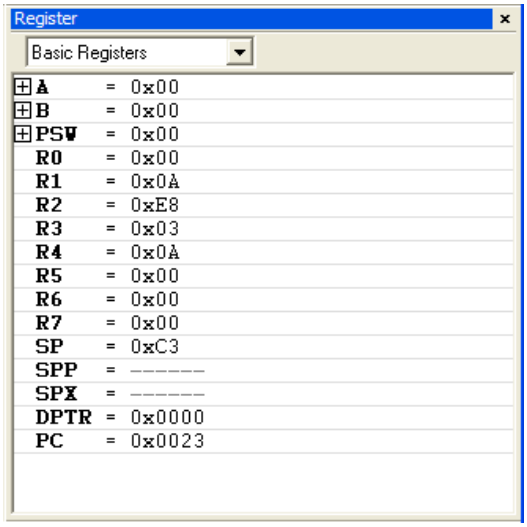


图 5.2 2.6 查看寄存器内容

选择窗口上部的下拉列表，选择不同的寄存器分组。单步运行程序观察寄存器值的变化情况。

7）监控存储器：

存储器窗口允许用户监控寄存器的指定区域。选择菜单 View\Memory，打开存储器窗口。

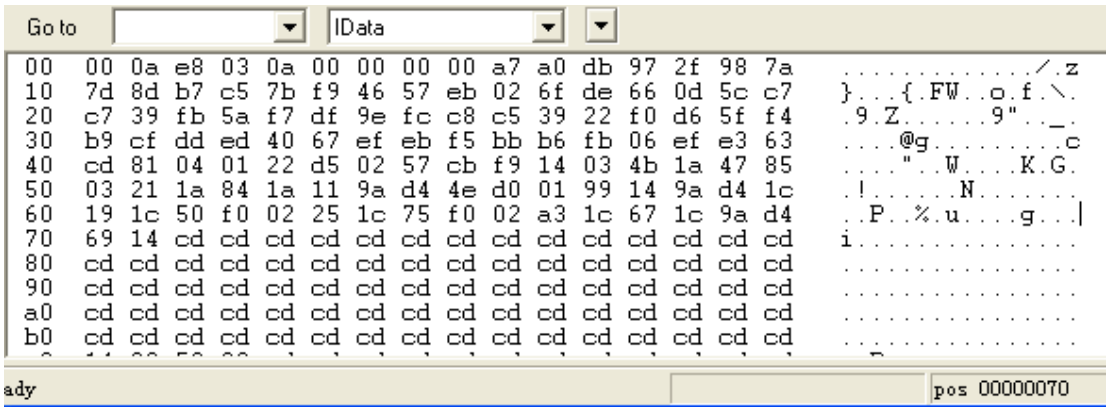


图 5.2.2.7 查看存储器

8）运行程序：

选择菜单 Debug\Go，或点击调试工具栏上按钮如果没有断点，程序将一直运行下去。可以看到 LED1、LED2 间隙点亮。如果要停止，选择菜单 Debug\Break 或点调试工具栏上按钮，停止程序运行。

9）退出调试：

选择菜单 Debug\Stop Debugging 或点击调试工具栏上的按钮。退出调试模式。

实验二. LED 灯控制实验

1. 实验环境

- 硬件：ZigBee(CC2530)模块，ZigBee 下载调试板，USB 仿真器，PC 机。
- 软件：IAR Embedded Workbench for MCS-51

2. 实验内容

- 阅读 ZigBee2530 开发套件 ZigBee 模块硬件部分文档，熟悉 ZigBee 模块硬件接口。
- 使用 IAR 开发环境设计程序，利用 CC2530 的 IO 控制 LED 外设的闪烁。

3. 实验原理

3.1 硬件接口原理

♦ **ZigBee(CC2530)模块 LED 硬件接口**

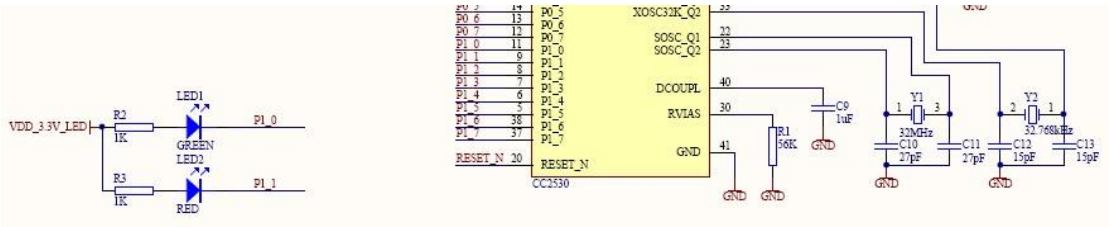


图 3.1.1 LED 硬件接口

ZigBee（CC2530）模块硬件上设计有 2 个 LED 灯，用来编程调试使用。分别连接 CC2530 的 P1_0、P1_1 两个 IO 引脚。从原理图上可以看出，2 个 LED 灯共阳极，当 P1_0、P1_1 引脚为低电平时候，LED 灯点亮。

♦ **CC2530 IO 相关寄存器**

P1 (0x90) – Port 1

Bit	Name	Reset	R/W	Description
7:0	P1[7:0]	0xFF	R/W	Port 1. General-purpose I/O port. Bit-addressable from SFR. This CPU-internal register is readable, but not writable, from XDATA (0x7090).

表 3.1.2 P1 寄存器

P1DIR (0xFE) – Port 1 Direction

Bit	Name	Reset	R/W	Description
7:0	DIRP1_[7:0]	0x00	R/W	P1.7 to P1.0 I/O direction 0: Input 1: Output

表 3.1.3 P1DIR 寄存器

以上图表列出了关于 CC2530 处理器的 P1 IO 相关寄存器，其中只用到了 P1 和 P1DIR 两个寄存器的设置，P1 寄存器为可读写的数据寄存器，P1DIR 为 IO 输入输出选择寄存器，其他 IO 寄存器的功能，使用默认配置。详情请用户参考 CC2530 的芯片手册。

3.2 软件设计

```
#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
#define LED2 P1_1   //定义 LED2 为 P11 口控制

//函数声明
void Delay(uint);       //延时函数
void Initial(void);           //初始化 P1 口
/*****************************
//初始化程序
*****************************/
void Initial(void)
{
  P1DIR |= 0x03; //P1_0、P1_1 定义为输出

  LED1 = 1;           //LED1 灯熄灭
  LED2 = 1;     //LED2 灯熄灭
}

/***************************
//主函数
***************************/
void main(void)
{
  Initial();        //调用初始化函数
  LED1 = 0;           //LED1 点亮
  LED2 = 0;           //LED2 点亮
  while(1)
  {
      LED2 = !LED2;             //LED2 闪烁
      Delay(50000);
  }
}
```

程序通过配置 CC2530 IO 寄存器的高低电平来控制 LED 灯的状态，用循环语句来实现程序的不间断运行。

4. 实验步骤

1）使用 ZigBee Debuger USB 仿真器连接 PC 机和 ZigBee(CC2530)模块，打开 ZIGBEE 模块开关供电。

2）启动 IAR 开发环境，新建工程，或直接使用 Exp1 实验工程。

3）在 IAR 开发环境中编译、运行、调试程序。

实验三. Timer1 控制实验

1. 实验环境

- 硬件：ZigBee(CC2530)模块，ZigBee 下载调试板，USB 仿真器，PC 机。
- 软件：IAR Embedded Workbench for MCS-51

2. 实验内容

- 阅读 ZigBee2530 开发套件 ZigBee 模块硬件部分文档，熟悉 ZigBee 模块硬件接口。
- 使用 IAR 开发环境设计程序，利用 CC2530 的 Timer1 定时器控制 LED 外设的闪烁。

3. 实验原理

3.1 硬件接口原理

T1CTL (0xE4) – Timer 1 Control

Bit	Name	Reset	R/W	Description
7:4	–	0000 0	R0	Reserved
3:2	DIV[1:0]	00	R/W	Prescaler divider value. Generates the active clock edge used to update the counter as follows: 00: Tick frequency/1 01: Tick frequency/8 10: Tick frequency/32 11: Tick frequency/128
1:0	MODE [1:0]	00	R/W	Timer 1 mode select. The timer operating mode is selected as follows: 00: Operation is suspended. 01: Free-running, repeatedly count from 0x0000 to 0xFFFF. 10: Modulo, repeatedly count from 0x0000 to T1CC0. 11: Up/down, repeatedly count from 0x0000 to T1CC0 and from T1CC0 down to 0x0000.

IRCON (0xC0) – Interrupt Flags 4

Bit	Name	Reset	R/W	Description
7	STIF	0	R/W	Sleep Timer interrupt flag 0: Interrupt not pending 1: Interrupt pending
6	–	0	R/W	Must be written 0. Writing a 1 always enables the interrupt source.
5	P0IF	0	R/W	Port 0 interrupt flag 0: Interrupt not pending 1: Interrupt pending
4	T4IF	0	R/W H0	Timer 4 interrupt flag. Set to 1 when Timer 4 interrupt occurs and cleared when CPU vectors to the interrupt service routine. 0: Interrupt not pending 1: Interrupt pending
3	T3IF	0	R/W H0	Timer 3 interrupt flag. Set to 1 when Timer 3 interrupt occurs and cleared when CPU vectors to the interrupt service routine. 0: Interrupt not pending 1: Interrupt pending
2	T2IF	0	R/W H0	Timer 2 interrupt flag. Set to 1 when Timer 2 interrupt occurs and cleared when CPU vectors to the interrupt service routine. 0: Interrupt not pending 1: Interrupt pending
1	T1IF	0	R/W H0	Timer 1 interrupt flag. Set to 1 when Timer 1 interrupt occurs and cleared when CPU vectors to the interrupt service routine. 0: Interrupt not pending 1: Interrupt pending
0	DMAIF	0	R/W	DMA-complete interrupt flag 0: Interrupt not pending 1: Interrupt pending

表 3.1.3 IRCON 寄存器

以上图表列举了和 CC2530 处理器 Timer1 定时器相关的寄存器，其中 T1CTL 为 Timer1 定时器控制状态寄存器，通过该寄存器来设置定时器的模式和预分频系数。IRCON 寄存器为中断标志位寄存器，通过该寄存器可以判断相应控制器 Timer1 的中断状态。

3.2 软件设计

```
#include <ioCC2530.h>
#define uint unsigned int
#define uchar unsigned char
#define LED1 P1_0
#define LED2 P1_1

uint counter=0;                 //统计溢出次数
uint TempFlag;                  //用来标志是否要闪烁

void Initial(void);
void Delay(uint);

/*****************************
//延时程序
*****************************/
void Delay(uint n)
{
      uint i,t;
```

```
    for(i = 0;i<5;i++)
      for(t = 0;t<n;t++);
}

/*****************************
//初始化程序
*****************************/
void Initial(void)
{
    //初始化 P1
    P1DIR = 0x03;   //P1_0 P1_1 为输出
    LED1 = 1;
    LED2 = 1;       //熄灭 LED

    //初始化 T1 定时器
    T1CTL = 0x0d;  //中断无效,128 分频;自动重装模式(0x0000->0xffff);
}

/*****************************
//主函数
*****************************/
void main()
{
    Initial();      //调用初始化函数
    LED1 = 0;    //点亮 LED1
    while(1)      //查询溢出
    {
        if(IRCON > 0)
        {
            IRCON = 0;                    //清溢出标志
            TempFlag = !TempFlag;
        }
        if(TempFlag)
        {
            LED2 = LED1;
            LED1 = !LED1;
            Delay(6000);
        }
    }
}
```

程序通过配置 CC2530 处理器的 Timer1 定时器进行自动装载计数，通过查询 IRCON 中断标志来检查 Timer1 定时器计数溢出中断状态，从而控制 LED 灯的闪烁状态。

4. 实验步骤

1）使用 ZigBee Debuger USB 仿真器连接 PC 机和 ZigBee(CC2530)模块，打开 ZigBee 模块开关供电。

2）启动 IAR 开发环境，新建工程，或直接使用 Exp2 实验工程。

3）在 IAR 开发环境中编译、运行、调试程序。

实验四. Timer2 控制实验

1. 实验环境

- 硬件：ZigBee(CC2530)模块，ZigBee 下载调试板，USB 仿真器，PC 机。
- 软件：IAR Embedded Workbench for MCS-51

2. 实验内容

- 阅读 ZigBee2530 开发套件 ZigBee 模块硬件部分文档，熟悉 ZigBee 模块硬件接口。
- 使用 IAR 开发环境设计程序，利用 CC2530 的 Timer2 定时器控制 LED 外设的闪烁。

3. 实验原理

T2MSEL (0xC3) – Timer 2 Multiplex Select

Bit No.	Name	Reset	R/W	Function
7:0	–	0	R0	Reserved. Read as 0
6:4	T2MOVFSEL	0	R/W	The value of this register selects the internal registers that are modified or read when accessing T2MOVF0, T2MOVF1, and T2MOVF2 . 000: t2ovf (overflow counter) 001: t2ovf_cap (overflow capture) 010: t2ovf_per (overflow period) 011: t2ovf_cmp1 (overflow compare 1) 100: t2ovf_cmp2 (overflow compare 2) 101 to 111: Reserved
3	–	0	R0	Reserved. Read as 0
2:0	T2MSEL	0	R/W	The value of this register selects the internal registers that are modified or read when accessing T2M0 and T2M1. 000: t2tim (timer count value) 001: t2_cap (timer capture) 010: t2_per (timer period) 011: t2_cmp1 (timer compare 1) 100: t2_cmp2 (timer compare 2) 101 to 111: Reserved

表 3.1.4 T2MSEL 寄存器

T2M0 (0xA2) – Timer 2 Multiplexed Register 0

Bit No.	Name	Reset	R/W	Function
7:0	T2M0	0	R/W	Indirectly returns/modifies bits [7:0] of an internal register depending on the T2MSEL.T2MSEL value. When reading the T2M0 register with T2MSEL.T2MSEL set to 000 and T2CTRL.LATCH_MODE set to 0, the timer (t2tim) value is latched. When reading the T2M0 register with T2MSEL.T2MSEL set to 000 and T2CTRL.LATCH_MODE set to 1, the timer (t2tim) and overflow counter (t2ovf) values are latched.

T2M1 (0xA3) – Timer 2 Multiplexed Register 1

Bit No.	Name	Reset	R/W	Function
7:0	T2M1	0	R/W	Indirectly returns/modifies bits [15:8] of an internal register, depending on T2MSEL.T2MSEL value. When reading the T2M0 register with T2MSEL.T2MSEL set to 000, the timer (t2tim) value is latched. Reading this register with T2MSEL.T2MSEL set to 000 returns the latched value of t2tim[15:8].

表 3.1.5 T2M0 和 T2M1 寄存器

T2MOVF0 (0xA4) – Timer 2 Multiplexed Overflow Register 0

Bit No.	Name	Reset	R/W	Function
7:0	T2MOVF0	0	R/W	Indirectly returns/modifies bits [7:0] of an internal register, depending on the T2MSEL.T2MOVFSEL value. When reading the T2MOVF0 register with T2MSEL.T2MOVFSEL set to 000 and T2CTRL.LATCH_MODE set to 0, the overflow counter value (t2ovf) is latched. When reading the T2M0 register with T2MSEL.T2MOVFSEL set to 000 and T2CTRL.LATCH_MODE set to 1, the overflow counter value (t2ovf) is latched.

T2MOVF1 (0xA5 – Timer 2 Multiplexed Overflow Register 2

Bit No.	Name	Reset	R/W	Function
7:0	T2MOVF1	0	R/W	Indirectly returns/modifies bits [15:8] of an internal register, depending on the T2MSEL.T2MSEL value. Reading this register with T2MSEL.T2MOVFSEL set to 000 returns the latched value of t2ovf [15:8].

T2MOVF2 (0xA6) – Timer 2 Multiplexed Overflow Register 2

Bit No.	Name	Reset	R/W	Function
7:0	T2MOVF2	0	R/W	Indirectly returns/modifies bits [23:16] of an internal register, depending on the T2MSEL.T2MOVFSEL value. Reading this register with T2MSEL.T2MOVFSEL set to 000 returns the latched value of t2ovf [23:16].

表 3.1.6 T2MOVF0 T2MOVF1 和 T2MOVF2 寄存器

T2IRQF (0xA1) – Timer 2 Interrupt Flags

Bit No.	Name	Reset	R/W	Function
7:6	–	0	R0	Reserved. Read as 0
5	TIMER2_OVF_COMPARE2F	0	R/W0	Set when the Timer 2 overflow counter counts to the value set at t2ovf_cmp2
4	TIMER2_OVF_COMPARE1F	0	R/W0	Set when the Timer 2 overflow counter counts to the value set at Timer 2 t2ovf_cmp1
3	TIMER2_OVF_PERF	0	R/W0	Set when the Timer 2 overflow counter would have counted to a value equal to t2ovf_per, but instead wraps to 0
2	TIMER2_COMPARE2F	0	R/W0	Set when the Timer 2 counter counts to the value set at t2_cmp2
1	TIMER2_COMPARE1F	0	R/W0	Set when the Timer 2 counter counts to the value set at t2_cmp1
0	TIMER2_PERF	0	R/W0	Set when the Timer 2 counter would have counted to a value equal to t2_per, but instead wraps to 0.

表 3.1.7 T2IRQF 寄存器

T2IRQM (0xA7) – Timer 2 Interrupt Mask

Bit No.	Name	Reset	R/W	Function
7:6	–	0	R0	Reserved. Read as 0
5	TIMER2_OVF_COMPARE2M	0	R/W	Enables the TIMER2_OVF_COMPARE2 interrupt
4	TIMER2_OVF_COMPARE1M	0	R/W	Enables the TIMER2_OVF_COMPARE1 interrupt
3	TIMER2_OVF_PERM	0	R/W	Enables the TIMER2_OVF_PER interrupt
2	TIMER2_COMPARE2M	0	R/W	Enables the TIMER2_COMPARE2 interrupt
1	TIMER2_COMPARE1M	0	R/W	Enables the TIMER2_COMPARE1 interrupt
0	TIMER2_PERM	0	R/W	Enables the TIMER2_PER interrupt

表 3.1.8 T2IRQM 寄存器

T2CTRL (0x94) – Timer 2 Control Register

Bit No.	Name	Reset	R/W	Function
7:4	–	0	R0	Reserved. Read as 0
3	LATCH_MODE	0	R/W	0: Reading T2M0 with T2MSEL.T2MSEL = 000 latches the high byte of the timer, making it ready to be read from T2M1. Reading T2MOVF0 with T2MSEL.T2MOVFSEL = 000 latches the two most-significant bytes of the overflow counter, making it possible to read these from T2MOVF1 and T2MOVF2. 1: Reading T2M0 with T2MSEL.T2MSEL = 000 latches the high byte of the timer and the entire overflow counter at once, making it possible to read the values from T2M1, T2MOVF0, T2MOVF1, and T2MOVF2.
2	STATE	0	R	State of Timer 2 0: Timer idle 1: Timer running
1	SYNC	1	R/W	0: Starting and stopping of timer is immediate, i.e., synchronous with clk_rf_32m. 1: Starting and stopping of timer happens at the first positive edge of the 32-kHz clock. Read Section 20.4 for more details regarding timer start and stop.
0	RUN	0	R/W	Write 1 to start timer, write 0 to stop timer. When read, it returns the last written value.

表 3.1.9 T2CTRL 寄存器

T2EVTCFG (0x9C) – Timer 2 CSP Interface Configuration

Bit No.	Name	Reset	R/W	Function
7	–	0	R0	Reserved. Read as 0
6:4	TIMER2_EVENT2_CFG	0	R/W	Selects the event that triggers a T2_EVENT2 pulse 000: t2_per_event 001: t2_cmp1_event 010: t2_cmp2_event 011: t2ovf_per_event 100: t2ovf_cmp1_event 101: t2ovf_cmp2_event 110: Reserved 111: No event
3	–	0	R0	Reserved. Read as 0
2:0	TIMER2_EVENT1_CFG	0	R/W	Selects the event that triggers a T2_EVENT1 pulse 000: t2_per_event 001: t2_cmp1_event 010: t2_cmp2_event 011: t2ovf_per_event 100: t2ovf_cmp1_event 101: t2ovf_cmp2_event 110: Reserved 111: No event

表 3.1.10 T2EVTCFG 寄存器

IEN0 (0xA8) – Interrupt Enable 0

Bit	Name	Reset	R/W	Description
7	EA	0	R/W	Disables all interrupts. 0: No interrupt is acknowledged. 1: Each interrupt source is individually enabled or disabled by setting its corresponding enable bit.
6	–	0	R0	Reserved. Read as 0
5	STIE	0	R/W	Sleep Timer interrupt enable 0: Interrupt disabled 1: Interrupt enabled
4	ENCIE	0	R/W	AES encryption/decryption interrupt enable 0: Interrupt disabled 1: Interrupt enabled
3	URX1IE	0	R/W	USART 1 RX interrupt enable 0: Interrupt disabled 1: Interrupt enabled
2	URX0IE	0	R/W	USART0 RX interrupt enable 0: Interrupt disabled 1: Interrupt enabled
1	ADCIE	0	R/W	ADC interrupt enable 0: Interrupt disabled 1: Interrupt enabled
0	RFERRIE	0	R/W	RF TXFIFO/RXFIFO interrupt enable 0: Interrupt disabled 1: Interrupt enabled

表 3.1.11 IEN0 寄存器

IEN1 (0xB8) – Interrupt Enable 1

Bit	Name	Reset	R/W	Description
7:6	–	00	R0	Reserved. Read as 0
5	P0IE	0	R/W	Port 0 interrupt enable 0: Interrupt disabled 1: Interrupt enabled
4	T4IE	0	R/W	Timer 4 interrupt enable 0: Interrupt disabled 1: Interrupt enabled
3	T3IE	0	R/W	Timer 3 interrupt enable 0: Interrupt disabled 1: Interrupt enabled
2	T2IE	0	R/W	Timer 2 interrupt enable 0: Interrupt disabled 1: Interrupt enabled
1	T1IE	0	R/W	Timer 1 interrupt enable 0: Interrupt disabled 1: Interrupt enabled
0	DMAIE	0	R/W	DMA transfer interrupt enable 0: Interrupt disabled 1: Interrupt enabled

表 3.1.12 IEN1 寄存器

以上图表列举了和 CC2530 处理器 Timer2 定时器相关的寄存器，其中包括 T2CTRL Timer2 控制寄存器，用来控制定时器的开关， T2MSEL 控制寄存器，用来对 Timer2 功能的选择，T2M0 和 T2M1 用来存放 16 位计数值，T2MOVF0,T2MOVF1,T2MOVf2 用来存放计数值溢出的次数， T2IRQF 为 Timer2 中断标志寄存器， T2IRQM 为中断使能屏蔽寄存器， IEN0 和 IEN1 两个寄存器分别控制系统中断总开关和 Timer2 定时器中断源开关。

3.1 软件设计

```
#include <emot.h>

uint counter=0;                    //统计溢出次数
uchar TempFlag;                    //用来标志是否要闪烁


/*****************************
//延时程序
*****************************/
void Delay(uint n)
{
   uint i,t;
            for(i = 0;i<5;i++)
   for(t = 0;t<n;t++);
}

/*****************************
//初始化程序
*****************************/
void Initial(void)
{
      LED_ENALBLE();

      //设置 T2 定时器相关寄存器
      SET_TIMER2_CAP_INT();                          //开溢出中断
      SET_TIMER2_CAP_COUNTER(0x55);                  //设置溢出值
}

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

      LED1 = 0;         //LED1 常亮
      LED2 = 1;

TIMER2_RUN();

      while(1)            //等待中断
      {
            if(TempFlag)
```

```
            {
                LED2 = !LED2;
                TempFlag = 0;
            }
        }
}


/*************************
//中断处理函数
*************************/
#pragma vector = T2_VECTOR    //重定位中断向量表
__interrupt void T2_ISR(void)     //定义中断处理函数
{
    TIMER2_STOP();
    SET_TIMER2_CAP_COUNTER(0X55);        //设置溢出值
    CLEAR_TIMER2_INT_FLAG();              //清 T2 中断标志
    if(counter<100)counter++;          //100 次中断 LED 闪烁一轮
    else
    {
        counter = 0;                          //计数清零
        TempFlag = 1;                          //改变闪烁标志
    }
}
```

程序通过配置 CC2530 处理器的 Timer2 定时器进行计数中断设置，从而控制 LED2 的闪烁状态。

4. 实验步骤

1）使用 ZigBee Debuger USB 仿真器连接 PC 机和 ZigBee(CC2530)模块，打开 ZigBee 模块开关供电。

2）启动 IAR 开发环境，新建工程，或直接使用 Exp3 实验工程。

3）在 IAR 开发环境中编译、运行、调试程序。

实验五. Timer3 控制实验

1. 实验环境

- 硬件：ZigBee(CC2530)模块，ZigBee 下载调试板，USB 仿真器，PC 机。
- 软件：IAR Embedded Workbench for MCS-51

2. 实验内容

- 阅读 ZigBee2530 开发套件 ZigBee 模块硬件部分文档，熟悉 ZigBee 模块硬件接口。
- 使用 IAR 开发环境设计程序，利用 CC2530 的 Timer3 定时器控制 LED 外设的闪烁。

3. 实验原理

3.1 硬件接口原理

T3CTL (0xCB) – Timer 3 Control

Bit	Name	Reset	R/W	Description
7:5	DIV[2:0]	000	R/W	Prescaler divider value. Generates the active clock edge used to clock the timer from CLKCONCMD.TICKSPD as follows: 000: Tick frequency/1 001: Tick frequency/2 010: Tick frequency/4 011: Tick frequency/8 100: Tick frequency16 101: Tick frequency /32 110: Tick frequency/64 111: Tick frequency/128
4	START	0	R/W	Start timer. Normal operation when set, suspended when cleared
3	OVFIM	1	R/W0	Overflow interrupt mask 0: Interrupt is disabled. 1: Interrupt is enabled.
2	CLR	0	R0/W1	Clear counter. Writing a 1 to CLR resets the counter to 0x00 and initialize all output pins of associated channels. Always read as 0.
1:0	MODE[1:0]	00	R/W	Timer 3 mode. Select the mode as follows: 00: Free running, repeatedly count from 0x00 to 0xFF 01: Down, count from T3CC0 to 0x00 10: Modulo, repeatedly count from 0x00 to T3CC0 11: Up/down, repeatedly count from 0x00 to T3CC0 and down to 0x00

表 3.1.4 T3CTL 寄存器

T3CCTL0 (0xCC) – Timer 3 Channel 0 Capture/Compare Control

Bit	Name	Reset	R/W	Description
7	–	0	R0	Unused
6	IM	1	R/W	Channel 0 interrupt mask 0: Interrupt is disabled. 1: Interrupt is enabled.
5:3	CMP[2:0]	000	R/W	Channel 0 compare output mode select. Specified action on output when timer value equals compare value in T3CC0 000: Set output on compare 001: Clear output on compare 010: Toggle output on compare 011: Set output on compare-up, clear on 0 100: Clear output on compare-up, set on 0 101: Set output on compare, clear on 0xFF 110: Clear output on compare, set on 0x00 111: Initialize output pin. CMP[2:0] is not changed
2	MODE	0	R/W	Mode. Select Timer 3 Channel 0 mode 0: Capture mode 1: Compare mode
1:0	CAP[1:0]	00	R/W	Capture mode select 00: No capture 01: Capture on rising edge 10: Capture on falling edge 11: Capture on both edges

表 3.1.5　T3CCTL0 和 T3CC0 寄存器

T3CCTL1 (0xCE) – Timer 3 Channel 1 Capture/Compare Control

Bit	Name	Reset	R/W	Description
7	–	0	R0	Unused
6	IM	1	R/W	Channel 1 interrupt mask 0: Interrupt is disabled. 1: Interrupt is enabled.
5:3	CMP[2:0]	000	R/W	Channel 1 compare output-mode select. Specified action on output when timer value equals compare value in T3CC1 000: Set output on compare 001: Clear output on compare 010: Toggle output on compare 011: Set on compare-up, clear on compare down in up-down mode. Otherwise set output on compare, clear on 0 100: Clear output on compare-up, set on compare-down in up-down mode. Otherwise Clear output on compare, set on 0. 101: Set output on compare, clear on 0xFF 110: Clear output on compare, set on 0x00 111: Initialize output pin. CMP[2:0] is not changed
2	MODE	0	R/W	Mode. Select Timer 3 channel 1 mode 0: Capture mode 1: Compare mode
1:0	CAP[1:0]	00	R/W	Capture mode select 00: No capture 01: Capture on rising edge 10: Capture on falling edge 11: Capture on both edges

T3CC1 (0xCF) – Timer 3 Channel 1 Capture/Compare Value

Bit	Name	Reset	R/W	Description
7:0	VAL[7:0]	0x00	R/W	Timer capture/compare value channel 1. Writing to this register when T3CCTL1.MODE=1 (compare mode) causes the T3CC1.VAL[7:0] update to the written value to be delayed until T3CNT.CNT[7:0]=0x00.

表 3.1.6 T3CCTL1 和 T3CC1 寄存器

IEN0 (0xA8) – Interrupt Enable 0

Bit	Name	Reset	R/W	Description
7	EA	0	R/W	Disables all interrupts. 0: No interrupt is acknowledged. 1: Each interrupt source is individually enabled or disabled by setting its corresponding enable bit.
6	–	0	R0	Reserved. Read as 0
5	STIE	0	R/W	Sleep Timer interrupt enable 0: Interrupt disabled 1: Interrupt enabled
4	ENCIE	0	R/W	AES encryption/decryption interrupt enable 0: Interrupt disabled 1: Interrupt enabled
3	URX1IE	0	R/W	USART 1 RX interrupt enable 0: Interrupt disabled 1: Interrupt enabled
2	URX0IE	0	R/W	USART0 RX interrupt enable 0: Interrupt disabled 1: Interrupt enabled
1	ADCIE	0	R/W	ADC interrupt enable 0: Interrupt disabled 1: Interrupt enabled
0	RFERRIE	0	R/W	RF TXFIFO/RXFIFO interrupt enable 0: Interrupt disabled 1: Interrupt enabled

表 3.1.7 IEN0 寄存器

IEN1 (0xB8) – Interrupt Enable 1

Bit	Name	Reset	R/W	Description
7:6	–	00	R0	Reserved. Read as 0
5	P0IE	0	R/W	Port 0 interrupt enable 0: Interrupt disabled 1: Interrupt enabled
4	T4IE	0	R/W	Timer 4 interrupt enable 0: Interrupt disabled 1: Interrupt enabled
3	T3IE	0	R/W	Timer 3 interrupt enable 0: Interrupt disabled 1: Interrupt enabled
2	T2IE	0	R/W	Timer 2 interrupt enable 0: Interrupt disabled 1: Interrupt enabled
1	T1IE	0	R/W	Timer 1 interrupt enable 0: Interrupt disabled 1: Interrupt enabled
0	DMAIE	0	R/W	DMA transfer interrupt enable 0: Interrupt disabled 1: Interrupt enabled

表 3.1.8 IEN1 寄存器

以上图表列举了和 CC2530 处理器 Timer3 定时器相关的寄存器，其中包括 T3CTL 控制寄存器，用来控制定时器的开关和模式。 T3CCTL0 和 T3CC0 为 Timer3 通道 0 比较/捕获控制寄存器和值寄存器， T3CCTL1 和 T3CC1 为 Timer3 通道 1 比较/捕获控制寄存器和值寄存器。 IEN0 与 IEN1 两个寄存器分别控制系统中断总开关和 Timer3 定时器中断源开关。

3.2 软件设计

```
#include <ioCC2530.h>

#define YLED P1_0
#define RLED P1_1

#define uchar unsigned char

/*****************************************
//定义全局变量
*****************************************/
uchar counter = 0;

/*****************************************
//T3 配置定义
*****************************************/
// Where _timer_ must be either 3 or 4
// Macro for initialising timer 3 or 4
//将 T3/4 配置寄存复位
#define TIMER34_INIT(timer)     \
    do {                                \
        T##timer##CTL     = 0x06; \
        T##timer##CCTL0 = 0x00; \
        T##timer##CC0     = 0x00; \
        T##timer##CCTL1 = 0x00; \
        T##timer##CC1     = 0x00; \
  } while (0)

//Macro for enabling overflow interrupt
//设置 T3/4 溢出中断
#define TIMER34_ENABLE_OVERFLOW_INT(timer,val) \
   do{T##timer##CTL =    (val) ? T##timer##CTL | 0x08 : T##timer##CTL & ~0x08; \
      EA = 1;         \
      T3IE = 1;     \
      }while(0)

//启动 T3
#define TIMER3_START(val)                                   \
      (T3CTL = (val) ? T3CTL | 0X10 : T3CTL&~0X10)

//时钟分步选择
#define TIMER3_SET_CLOCK_DIVIDE(val)                \
   do{                          \
      T3CTL &= ~0XE0;                                   \
```

```
        (val==2) ? (T3CTL|=0X20):                    \
        (val==4) ? (T3CTL|=0x40):                    \
        (val==8) ? (T3CTL|=0X60):                    \
        (val==16)? (T3CTL|=0x80):                    \
        (val==32)? (T3CTL|=0xa0):                    \
        (val==64) ? (T3CTL|=0xc0):                   \
        (T3CTL|=0X00);                /* 1 */        \
   }while(0)


//Macro for setting the mode of timer3
//设置 T3 的工作方式
#define TIMER3_SET_MODE(val)                              \
   do{                        \
      T3CTL &= ~0X03; \
      (val==1)?(T3CTL|=0X01):    /*DOWN                */ \
      (val==2)?(T3CTL|=0X02):    /*Modulo              */ \
      (val==3)?(T3CTL|=0X03):    /*UP / DOWN           */ \
      (T3CTL|=0X00);             /*free runing */      \
   }while(0)


#define T3_MODE_FREE        0X00
#define T3_MODE_DOWN        0X01
#define T3_MODE_MODULO    0X02
#define T3_MODE_UP_DOWN 0X03


/****************************************
//T3 及 LED 初始化
****************************************/
void Init_T3_AND_LED(void)
{
      P1DIR = 0X03;
      RLED = 1;
      YLED = 1;

      TIMER34_INIT(3);                              //初始化 T3
      TIMER34_ENABLE_OVERFLOW_INT(3,1);   //开 T3 中断
 //时钟 32 分频 101
      TIMER3_SET_CLOCK_DIVIDE(16);
      TIMER3_SET_MODE(T3_MODE_FREE);       //自动重装 00－>0xff
      TIMER3_START(1);                              //启动
};
```

```
/*****************************************
//主函数
*****************************************/
void main(void)
{
   Init_T3_AND_LED();
   YLED = 0;
   while(1);                              //等待中断
}

#pragma vector = T3_VECTOR
 __interrupt void T3_ISR(void)
 {
  //IRCON = 0x00;                 //清中断标志,硬件自动完成
            if(counter<200)counter++;      //200 次中断 LED 闪烁一轮
           else
           {
              counter = 0;                          //计数清零
              RLED = !RLED;                          //改变小灯的状态
           }
 }
```

程序通过配置 CC2530 处理器的 Timer3 定时器进行计数中断设置，从而控制 RLED 灯的闪烁状态。

4. 实验步骤

1）使用 ZigBee Debuger USB 仿真器连接 PC 机和 ZigBee(CC2530)模块，打开 ZigBee 模块开关供电。

2）启动 IAR 开发环境，新建工程，或直接使用 Exp4 实验工程。

3）在 IAR 开发环境中编译、运行、调试程序。

实验六. Timer4 控制实验

1. 实验环境

- 硬件：ZigBee(CC2530)模块，ZigBee 下载调试板，USB 仿真器，PC 机。
- 软件：IAR Embedded Workbench for MCS-51

2. 实验内容

- 阅读 ZigBee2530 开发套件 ZigBee 模块硬件部分文档，熟悉 ZigBee 模块硬件接口。
- 使用 IAR 开发环境设计程序，利用 CC2530 的 Timer4 定时器控制 LED 外设的闪烁。

3. 实验原理

3.1 硬件接口原理

T4CTL (0xEB) – Timer 4 Control

Bit	Name	Reset	R/W	Description
7:5	DIV[2:0]	000	R/W	Prescaler divider value. Generates the active clock edge used to clock the timer from CLKCONCMD.TICKSPD as follows: 000: Tick frequency/1 001: Tick frequency/2 010: Tick frequency/4 011: Tick frequency/8 100: Tick frequency/16 101: Tick frequency/32 110: Tick frequency/64 111: Tick frequency/128
4	START		R/W	Start timer. Normal operation when set, suspended when cleared
3	OVFIM	1	R/W	Overflow interrupt mask
2	CLR	0	R0/W1	Clear counter. Writing a 1 to CLR resets the counter to 0x00 and initialize all output pins of associated channels. Always read as 0.
1:0	MODE[1:0]	0	R/W	Timer 4 mode. Select the mode as follows: 00: Free running, repeatedly count from 0x00 to 0xFF 01: Down, count from T4CC0 to 0x00 10: Modulo, repeatedly count from 0x00 to T4CC0 11: Up/down, repeatedly count from 0x00 to T4CC0 and down to 0x00

表 3.1.4 T4CTL 寄存器

T4CCTL0 (0xEC) – Timer 4 Channel 0 Capture/Compare Control

Bit	Name	Reset	R/W	Description
7	–	0	R0	Unused
6	IM	1	R/W	Channel 0 interrupt mask
5:3	CMP[2:0]	000	R/W	Channel 0 compare output mode select. Specified action on output when timer value equals compare value in T4CC0 000: Set output on compare 001: Clear output on compare 010: Toggle output on compare 011: Set output on compare-up, clear on 0 100: Clear output on compare-up, set on 0 101: Set output on compare, clear on 0xFF 110: Clear output on compare, set on 0x00 111: Initialize output pin. CMP[2:0] is not changed
2	MODE	0	R/W	Mode. Select Timer 4 channel 0 mode 0: Capture mode 1: Compare mode
1:0	CAP[1:0]	00	R/W	Capture mode select. 00 - No Capture, 01 - Capture on rising edge, 10 - Capture on falling edge, 11 - Capture on both edges

T4CC0 (0xED) – Timer 4 Channel 0 Capture/Compare Value

Bit	Name	Reset	R/W	Description
7:0	VAL[7:0]	0x00	R/W	Timer capture/compare value channel 0. Writing to this register when T4CCTL0.MODE=1 (compare mode) causes the T4CC0.VAL[7:0] update to the written value to be delayed until T4CNT.CNT[7:0]=0x00.

表 3.1.5 T4CCTL0 和 T4CC0 寄存器

T4CCTL1 (0xEE) – Timer 4 Channel 1 Capture/Compare Control

Bit	Name	Reset	R/W	Description
7	–	0	R0	Unused
6	IM	1	R/W	Channel 1 interrupt mask
5:3	CMP[2:0]	000	R/W	Channel 1 compare output mode select. Specified action on output when timer value equals compare value in T4CC1 000: Set output on compare 001: Clear output on compare 010: Toggle output on compare 011: Set on compare-up, clear on compare down in up-down mode. Otherwise set output on compare, clear on 0 100: Clear output on compare-up, set on compare-down in up-down mode. Otherwise Clear output on compare, set on 0. 101: Set output on compare, clear on 0xFF 110: Clear output on compare, set on 0x00 111: Initialize output pin. CMP[2:0] is not changed
2	MODE	0	R/W	Mode. Select Timer 4 channel 1 mode 0: Capture mode 1: Compare mode
1:0	CAP[1:0]	00	R/W	Capture mode select. 00 - No Capture, 01 - Capture on rising edge, 10 - Capture on falling edge, 11 - Capture on both edges

T4CC1 (0xEF) – Timer 4 Channel 1 Capture/Compare Value

Bit	Name	Reset	R/W	Description
7:0	VAL[7:0]	0x00	R/W	Timer capture/compare value, channel 1. Writing to this register when T4CCTL1.MODE=1 (compare mode) causes the T4CC1.VAL[7:0] update to the written value to be delayed until T4CNT.CNT[7:0]=0x00.

表 3.1.6 T4CCTL1 和 T4CC1 寄存器

IEN0 (0xA8) – Interrupt Enable 0

Bit	Name	Reset	R/W	Description
7	EA	0	R/W	Disables all interrupts. 0: No interrupt is acknowledged. 1: Each interrupt source is individually enabled or disabled by setting its corresponding enable bit.
6	–	0	R0	Reserved. Read as 0
5	STIE	0	R/W	Sleep Timer interrupt enable 0: Interrupt disabled 1: Interrupt enabled
4	ENCIE	0	R/W	AES encryption/decryption interrupt enable 0: Interrupt disabled 1: Interrupt enabled
3	URX1IE	0	R/W	USART 1 RX interrupt enable 0: Interrupt disabled 1: Interrupt enabled
2	URX0IE	0	R/W	USART0 RX interrupt enable 0: Interrupt disabled 1: Interrupt enabled
1	ADCIE	0	R/W	ADC interrupt enable 0: Interrupt disabled 1: Interrupt enabled
0	RFERRIE	0	R/W	RF TXFIFO/RXFIFO interrupt enable 0: Interrupt disabled 1: Interrupt enabled

表 3.1.7 IEN0 寄存器

IEN1 (0xB8) – Interrupt Enable 1

Bit	Name	Reset	R/W	Description
7:6	–	00	R0	Reserved. Read as 0
5	P0IE	0	R/W	Port 0 interrupt enable 0: Interrupt disabled 1: Interrupt enabled
4	T4IE	0	R/W	Timer 4 interrupt enable 0: Interrupt disabled 1: Interrupt enabled
3	T3IE	0	R/W	Timer 3 interrupt enable 0: Interrupt disabled 1: Interrupt enabled
2	T2IE	0	R/W	Timer 2 interrupt enable 0: Interrupt disabled 1: Interrupt enabled
1	T1IE	0	R/W	Timer 1 interrupt enable 0: Interrupt disabled 1: Interrupt enabled
0	DMAIE	0	R/W	DMA transfer interrupt enable 0: Interrupt disabled 1: Interrupt enabled

表 3.1.8 IEN1 寄存器

以上图表列举了和 CC2530 处理器 Timer4 定时器相关的寄存器，其中包括 T4CTL 控制寄存器，用来控制定时器的开关和模式。T4CCTL0 和 T4CC0 为 Timer4 通道 0 比较/捕获控制寄存器和值寄存器，T4CCTL1 和 T4CC1 为 Timer4 通道 1 比较/捕获控制寄存器和值寄存器。 IEN0 与 IEN1 两个寄存器分别控制系统中断总开关和 Timer4 定时器中断源开关。

3.2 软件设计

```
#include <ioCC2530.h>

#define led1 P1_0
#define led2 P1_1

#define uchar unsigned char

/****************************************
//定义全局变量
****************************************/
uchar counter = 0;

/****************************************
//T4 配置定义
****************************************/
// Where _timer_ must be either 3 or 4
// Macro for initialising timer 3 or 4
#define TIMER34_INIT(timer)     \
    do {                                \
        T##timer##CTL     = 0x06; \
        T##timer##CCTL0 = 0x00; \
        T##timer##CC0     = 0x00; \
        T##timer##CCTL1 = 0x00; \
        T##timer##CC1     = 0x00; \
    } while (0)

//Macro for enabling overflow interrupt
#define TIMER34_ENABLE_OVERFLOW_INT(timer,val) \
   do {
       (T##timer##CTL =    (val) ? T##timer##CTL | 0x08 : T##timer##CTL & ~0x08);
       EA=1\
       T4IE=1\
      }while(0)



// Macro for configuring channel 1 of timer 3 or 4 for PWM mode.
#define TIMER34_PWM_CONFIG(timer)                           \
    do{                         \
        T##timer##CCTL1 = 0x24;                             \
        if(timer == 3){                                     \
            if(PERCFG & 0x20) {                             \
                IO_FUNC_PORT_PIN(1,7,IO_FUNC_PERIPH); \
```

```
            }                   \
            else {          \
                IO_FUNC_PORT_PIN(1,4,IO_FUNC_PERIPH); \
            }                   \
        }                       \
        else {              \
            if(PERCFG & 0x10) {                                 \
                IO_FUNC_PORT_PIN(2,3,IO_FUNC_PERIPH);\
            }                   \
            else {          \
                IO_FUNC_PORT_PIN(1,1,IO_FUNC_PERIPH); \
            }                   \
        }                       \
    } while(0)

// Macro for setting pulse length of the timer in PWM mode
#define TIMER34_SET_PWM_PULSE_LENGTH(timer, value) \
    do {                        \
        T##timer##CC1 = (BYTE)value;                            \
    } while (0)


// Macro for setting timer 3 or 4 as a capture timer
#define TIMER34_CAPTURE_TIMER(timer,edge)                   \
    do{                         \
        T##timer##CCTL1 = edge;                                 \
        if(timer == 3){                                     \
            if(PERCFG & 0x20) {                                 \
                IO_FUNC_PORT_PIN(1,7,IO_FUNC_PERIPH);    \
            }                   \
            else {          \
                IO_FUNC_PORT_PIN(1,4,IO_FUNC_PERIPH); \
        }               \
         }                      \
        else {              \
            if(PERCFG & 0x10) {                                 \
                IO_FUNC_PORT_PIN(2,3,IO_FUNC_PERIPH);    \
            }                   \
           else {           \
              IO_FUNC_PORT_PIN(1,1,IO_FUNC_PERIPH);     \
           }                    \
        }                       \
   }while(0)

//Macro for setting the clock tick for timer3 or 4
#define TIMER34_START(timer,val)                                    \
```

```
    (T##timer##CTL = (val) ? T##timer##CTL | 0X10 : T##timer##CTL&~0X10)

#define TIMER34_SET_CLOCK_DIVIDE(timer,val)              \
  do{                          \
    T##timer##CTL &= ~0XE0; \
      (val==2) ? (T##timer##CTL|=0X20):                    \
      (val==4) ? (T##timer##CTL|=0x40):                    \
      (val==8) ? (T##timer##CTL|=0X60):                    \
      (val==16)? (T##timer##CTL|=0x80):                    \
      (val==32)? (T##timer##CTL|=0xa0):                    \
      (val==64) ? (T##timer##CTL|=0xc0):                   \
      (val==128) ? (T##timer##CTL|=0XE0):                  \
      (T##timer##CTL|=0X00);                 /* 1 */              \
  }while(0)

//Macro for setting the mode of timer3 or 4
#define TIMER34_SET_MODE(timer,val)                        \
  do{                              \
    T##timer##CTL &= ~0X03; \
    (val==1)?(T##timer##CTL|=0X01):   /*DOWN                  */ \
    (val==2)?(T##timer##CTL|=0X02):   /*Modulo              */ \
    (val==3)?(T##timer##CTL|=0X03):   /*UP / DOWN           */ \
    (T##timer##CTL|=0X00);              /*free runing */       \
  }while(0)


/*****************************************
//T3 及 LED 初始化
*****************************************/
void Init_T4_AND_LED(void)
{
    P1DIR = 0X03;
    led1 = 1;
    led2 = 1;

    TIMER34_INIT(4);                              //初始化 T4
    TIMER34_ENABLE_OVERFLOW_INT(4,1);   //开 T4 中断

    TIMER34_SET_CLOCK_DIVIDE(4,128);
    TIMER34_SET_MODE(4,0);                        //自动重装 00－>0xff

    TIMER34_START(4,1);                           //启动
};

void main(void)
{
```

```
    Init_T4_AND_LED();                          //初始化 LED 和 T4
    while(1);                                   //等待中断
}


#pragma vector = T4_VECTOR
 __interrupt void T4_ISR(void)
 {
        IRCON = 0x00;               //可不清中断标志,硬件自动完成
         led2 = 0;                                  //for test
         if(counter<200)counter++; //200 次中断 LED 闪烁一轮
         else
         {
           counter = 0;                         //计数清零
           led1 = !led1;                          //改变小灯的状态
         }
 }
```

程序通过配置 CC2530 处理器的 Timer4 定时器进行计数中断设置，从而控制 LED1 灯的闪烁状态。

4. 实验步骤

1）使用 ZigBee Debuger USB 仿真器连接 PC 机和 ZigBee(CC2530)模块，打开 ZigBee 模块开关供电。

2）启动 IAR 开发环境，新建工程，或直接使用 Exp5 实验工程。

3）在 IAR 开发环境中编译、运行、调试程序。

实验七. 片上温度 AD 实验

1. 实验环境

- 硬件：ZigBee(CC2530)模块，ZigBee 下载调试板，USB 仿真器，PC 机。
- 软件：IAR Embedded Workbench for MCS-51

2. 实验内容

- 阅读 ZigBee2530 开发套件 ZigBee 模块硬件部分文档，熟悉 ZigBee 模块相关硬件接口。
- 使用 IAR 开发环境设计程序，利用 CC2530 的内部温度传感器作为 A\D 输入源，将转换后的温度数值利用串口发送给 PC 机终端。

3. 实验原理

3.1 硬件接口原理

CLKCONCMD (0xC6) – Clock Control Command

Bit	Name	Reset	R/W	Description
7	OSC32K	1	R/W	32 kHz clock-source select. Setting this bit initiates a clock-source change only. CLKCONSTA.OSC32K reflects the current setting. The 16 MHz RCOSC must be selected as system clock when this bit is to be changed. 0: 32 kHz XOSC 1: 32 kHz RCOSC
6	OSC	1	R/W	System clock-source select. Setting this bit initiates a clock-source change only. CLKCONSTA.OSC reflects the current setting. 0: 32 MHz XOSC 1: 16 MHz RCOSC
5:3	TICKSPD[2:0]	001	R/W	Timer ticks output setting. Cannot be higher than system clock setting given by OSC bit setting. 000: 32 MHz 001: 16 MHz 010: 8 MHz 011: 4 MHz 100: 2 MHz 101: 1 MHz 110: 500 kHz 111: 250 kHz Note that CLKCONCMD.TICKSPD can be set to any value, but the effect is limited by the CLKCONCMD.OSC setting; i.e., if CLKCONCMD.OSC = 1 and CLKCONCMD.TICKSPD = 000, CLKCONSTA.TICKSPD reads 001, and the real TICKSPD is 16 MHz.
2:0	CLKSPD	001	R/W	Clock speed. Cannot be higher than system clock setting given by the OSC bit setting. Indicates current system-clock frequency 000: 32 MHz 001: 16 MHz 010: 8 MHz 011: 4 MHz 100: 2 MHz 101: 1 MHz 110: 500 kHz 111: 250 kHz Note that CLKCONCMD.CLKSPD can be set to any value, but the effect is limited by the CLKCONCMD.OSC setting; i.e., if CLKCONCMD.OSC = 1 and CLKCONCMD.CLKSPD = 000, CLKCONSTA.CLKSPD reads 001, and the real CLKSPD is 16 MHz. Note also that the debugger cannot be used with a divided system clock. When running the debugger, the value of CLKCONCMD.CLKSPD should be set to 000 when CLKCONCMD.OSC = 0 or to 001 when CLKCONCMD.OSC = 1.

CLKCONSTA (0x9E) – Clock Control Status

Bit	Name	Reset	R/W	Description
7	OSC32K	1	R	Current 32-kHz clock source selected: 0: 32-kHz XOSC 1: 32-kHz RCOSC
6	OSC	1	R	Current system clock selected: 0: 32-MHz XOSC 1: 16-MHz RCOSC
5:3	TICKSPD[2:0]	001	R	Current timer ticks output setting 000: 32 MHz 001: 16 MHz 010: 8 MHz 011: 4 MHz 100: 2 MHz 101: 1 MHz 110: 500 kHz 111: 250 kHz
2:0	CLKSPD	001	R	Current clock speed 000: 32 MHz 001: 16 MHz 010: 8 MHz 011: 4 MHz 100: 2 MHz 101: 1 MHz 110: 500 kHz 111: 250 kHz

表 3.1.4　CLKCONCMD 和 CLKCONSTA 寄存器

SLEEPCMD (0xBE) – Sleep-Mode Control Command

Bit	Name	Reset	R/W	Description
7	OSC32K_CALDIS	0	R/W	Disable 32-kHz RC oscillator calibration 0: 32-kHz RC oscillator calibration is enabled. 1: 32-kHz RC oscillator calibration is disabled. This setting can be written at any time, but does not take effect before the chip has been running on the 16-MHz high-frequency RC oscillator.
6:3	–	0000	R0	Reserved
2	–	1	R/W	Reserved. Always write as 1
1:0	MODE[1:0]	00	R/W	Power-mode setting 00: Active / Idle mode 01: Power mode 1 (PM1) 10: Power mode 2 (PM2) 11: Power mode 3 (PM3)

SLEEPSTA (0x9D) – Sleep-Mode Control Status

Bit	Name	Reset	R/W	Description
7	OSC32K_CALDIS	0	R	32-kHz RC oscillator calibration status SLEEPSTA.OSC32K_CALDIS shows the current status of disabling of the 32-kHz RC calibration. The bit is not set to the same value as SLEEPCMD.OSC32K_CALDIS before the chip has been run on the 32-kHz RC oscillator.
6:5	–	00	R	Reserved
4:3	RST[1:0]	XX	R	Status bit indicating the cause of the last reset. If there are multiple resets, the register only contains the last event. 00: Power-on reset and brownout detection 01: External reset 10: Watchdog Timer reset 11: Clock loss reset
2:1	–	00	R	Reserved
0	CLK32K	0	R	The 32-kHz clock signal (synchronized to the system clock)

表 3.1.5　SLEEPCMD 和 SLEEPSTA 控制寄存器

PERCFG (0xF1) – Peripheral Control

Bit	Name	Reset	R/W	Description
7	–	0	R0	Reserved
6	T1CFG	0	R/W	Timer 1 I/O location 0: Alternative 1 location 1: Alternative 2 location
5	T3CFG	0	R/W	Timer 3 I/O location 0: Alternative 1 location 1: Alternative 2 location
4	T4CFG	0	R/W	Timer 4 I/O location 0: Alternative 1 location 1: Alternative 2 location
3:2	–	00	R/W	Reserved
1	U1CFG	0	R/W	USART 1 I/O location 0: Alternative 1 location 1: Alternative 2 location
0	U0CFG	0	R/W	USART 0 I/O location 0: Alternative 1 location 1: Alternative 2 location

表 3.1.6　PERCFG 寄存器

U0CSR (0x86) – USART 0 Control and Status

Bit	Name	Reset	R/W	Description
7	MODE	0	R/W	USART mode select 0: SPI mode 1: UART mode
6	RE	0	R/W	UART receiver enable. Note do not enable receive before uart is fully configured. 0: Receiver disabled 1: Receiver enabled
5	SLAVE	0	R/W	SPI master or slave mode select 0: SPI master 1: SPI slave
4	FE	0	R/W0	UART framing error status. This bit is automatically cleared on a read of the U0CSR register or bits in the U0CSR register. 0: No framing error detected 1: Byte received with incorrect stop-bit level
3	ERR	0	R/W0	UART parity error status. This bit is automatically cleared on a read of the U0CSR register or bits in the U0CSR register. 0: No parity error detected 1: Byte received with parity error
2	RX_BYTE	0	R/W0	Receive byte status. UART mode and SPI slave mode. This bit is automatically cleared when reading U0DBUF, clearing this bit by writing 0 to it, effectively discards the data in U0DBUF. 0: No byte received 1: Received byte ready
1	TX_BYTE	0	R/W0	Transmit byte status. UART mode and SPI master mode 0: Byte not transmitted 1: Last byte written to data-buffer register transmitted
0	ACTIVE	0	R	USART transmit/receive active status. In SPI slave mode, this bit equals slave select 0: USART idle 1: USART busy in transmit or receive mode

表 3.1.7　U0CSR 寄存器

U0GCR (0xC5) – USART 0 Generic Control

Bit	Name	Reset	R/W	Description
7	CPOL	0	R/W	SPI clock polarity 0: Negative clock polarity 1: Positive clock polarity
6	CPHA	0	R/W	SPI clock phase 0: Data is output on *MOSI* when *SCK* goes from CPOL inverted to CPOL, and data input is sampled on *MISO* when *SCK* goes from CPOL to CPOL inverted. 1: Data is output on *MOSI* when *SCK* goes from CPOL to CPOL inverted, and data input is sampled on *MISO* when *SCK* goes from CPOL inverted to CPOL.
5	ORDER	0	R/W	Bit order for transfers 0: LSB first 1: MSB first
4:0	BAUD_E[4:0]	00000	R/W	Baud rate exponent value. BAUD_E along with BAUD_M determines the UART baud rate and the SPI master SCK clock frequency.

表 3.1.8 U0GCR 寄存器

U0DBUF (0xC1) – USART 0 Receive/Transmit Data Buffer

Bit	Name	Reset	R/W	Description
7:0	DATA[7:0]	0x00	R/W	USART receive and transmit data. When writing this register, the data written is written to the internal transmit-data register. When reading this register, the data from the internal read-data register is read.

U0BAUD (0xC2) – USART 0 Baud-Rate Control

Bit	Name	Reset	R/W	Description
7:0	BAUD_M[7:0]	0x00	R/W	Baud-rate mantissa value. BAUD_E along with BAUD_M decides the UART baud rate and the SPI master SCK clock frequency.

表 3.1.9 U0DBUF 和 U0BAUD 寄存器

ADCCON1 (0xB4) – ADC Control 1

Bit	Name	Reset	R/W	Description
7	EOC	0	R/H0	End of conversion. Cleared when ADCH has been read. If a new conversion is completed before the previous data has been read, the EOC bit remains high. 0: Conversion not complete 1: Conversion completed
6	ST	0		Start conversion. Read as 1 until conversion has completed 0: No conversion in progress 1: Start a conversion sequence if ADCCON1.STSEL = 11 and no sequence is running.
5:4	STSEL[1:0]	11	R/W1	Start select. Selects the event that starts a new conversion sequence 00: External trigger on P2.0 pin 01: Full speed. Do not wait for triggers 10: Timer 1 channel 0 compare event 11: ADCCON1.ST = 1
3:2	-	00	R/W	See ADCCON1 (0xB4) – ADC Control 1 description in Section 14.3.
1:0	–	11	R/W	Reserved. Always set to 11

表 3.1.10 ADCCON1 寄存器

ADCCON3 (0xB6) – ADC Control 3

Bit	Name	Reset	R/W	Description
7:6	EREF[1:0]	00	R/W	Selects reference voltage used for the extra conversion 00: Internal reference 01: External reference on AIN7 pin 10: AVDD5 pin 11: External reference on AIN6–AIN7 differential input
5:4	EDIV[1:0]	00	R/W	Sets the decimation rate used for the extra conversion. The decimation rate also determines the resolution and time required to complete the conversion. 00: 64 decimation rate (7 bits ENOB) 01: 128 decimation rate (9 bits ENOB) 10: 256 decimation rate (10 bits ENOB) 11: 512 decimation rate (12 bits ENOB)
3:0	ECH[3:0]	0000	R/W	Single channel select. Selects the channel number of the single conversion that is triggered by writing to ADCCON3. 0000: AIN0 0001: AIN1 0010: AIN2 0011: AIN3 0100: AIN4 0101: AIN5 0110: AIN6 0111: AIN7 1000: AIN0–AIN1 1001: AIN2–AIN3 1010: AIN4–AIN5 1011: AIN6–AIN7 1100: GND 1101: Reserved 1110: Temperature sensor 1111: VDD/3

表 3.1.11　ADCCON3　寄存器

以上图表列举了和 CC2530 处理器、内部温度传感器操作相关的寄存器，其中包括 CLKCONCMD 和 CLKCONSTA 控制寄存器，用来控制系统时钟源和状态， SLEEPCMD 和 SLEEPSTA 寄存器用来控制各种时钟源的开关和状态。PERCFG 寄存器为外设功能控制寄存器，用来控制外设功能模式。 U0CSR、 U0GCR、U0BUF、U0BAUD 等为串口相关寄存器。 ADCCON1 和 ADCCON3 分别为 AD 转换控制器和 AD 转换设置寄存器。

3.2 软件设计

```
#include <iocc2530.h>
#include <stdio.h>
#include "./uart/hal_uart.h"

#define uchar unsigned char
#define uint unsigned int
#define uint8 uchar
#define uint16 uint
#define TRUE 1
#define FALSE 0

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

```
#define LED2 P1_1   //定义 LED2 为 P11 口控制

//#define HAL_MCU_CC2530 1

// ADC definitions for CC2530/CC2530 from the hal_adc.c file
#define HAL_ADC_REF_125V      0x00      /* Internal 1.25V Reference */
#define HAL_ADC_DEC_064       0x00      /* Decimate by 64 : 8-bit resolution */
#define HAL_ADC_DEC_128       0x10      /* Decimate by 128 : 10-bit resolution */
#define HAL_ADC_DEC_512       0x30      /* Decimate by 512 : 14-bit resolution */
#define HAL_ADC_CHN_VDD3      0x0f      /* Input channel: VDD/3 */
#define HAL_ADC_CHN_TEMP      0x0e      /* Temperature sensor */

/*****************************
//延时函数
*****************************/
void Delay(uint n)
{
  uint i,t;
          for(i = 0;i<5;i++)
  for(t = 0;t<n;t++);
}

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
}

/*********************************************************************************
 * @fn              readTemp
 * @brief           read temperature from ADC
 * @param           none
 * @return          temperature
 */
static char readTemp(void)
{
    static uint16 voltageAtTemp22;
    static uint8  bCalibrate=TRUE; // Calibrate the first time the temp sensor is read
    uint16 value;
    char temp;

    ATEST = 0x01;
    TR0  |= 0x01;

    ADCIF = 0;    //clear ADC interrupt flag
```

```
    ADCCON3 = (HAL_ADC_REF_125V | HAL_ADC_DEC_512 | HAL_ADC_CHN_TEMP);


    while ( !ADCIF );                          //wait for the conversion to finish
    value = ADCL;                               //get the result
    value |= ((uint16) ADCH) << 8;
    value >>= 4;                               // Use the 12 MSB of adcValue

    /*
      * These parameters are typical values and need to be calibrated
      * See the datasheet for the appropriate chip for more details
      * also, the math below may not be very accurate
      */
    /* Assume ADC = 1480 at 25C and ADC = 4/C */
    #define VOLTAGE_AT_TEMP_25              1480
    #define TEMP_COEFFICIENT                4

    // Calibrate for 22C the first time the temp sensor is read.
    // This will assume that the demo is started up in temperature of 22C
    if(bCalibrate)
    {
       voltageAtTemp22=value;
       bCalibrate=FALSE;
    }

    temp = 22 + ( (value - voltageAtTemp22) / TEMP_COEFFICIENT );
    // Set 0C as minimum temperature, and 100C as max
    if( temp >= 100)         return 100;
    else if(temp <= 0)      return 0;
    else                            return temp;
}

/*********************************************************************************
  * @fn                readVoltage
  * @brief           read voltage from ADC
  * @param           none
  * @return          voltage
  */
/*
static uint8 readVoltage(void)
{
    uint16 value;

    ADCIF = 0;      // Clear ADC interrupt flag
    ADCCON3 = (HAL_ADC_REF_125V | HAL_ADC_DEC_128 | HAL_ADC_CHN_VDD3);
```

```
    while ( !ADCIF );                       // Wait for the conversion to finish
    value = ADCL;                           // Get the result
    value |= ((uint16) ADCH) << 8;

    // value now contains measurement of Vdd/3
    // 0 indicates 0V and 32767 indicates 1.25V
    // voltage = (value*3*1.25)/32767 volts
    // we will multiply by this by 10 to allow units of 0.1 volts
    value = value >> 6;     // divide first by 2^6
    value = (uint16)(value * 37.5);
    value = value >> 9;     // ...and later by 2^9...to prevent overflow during multiplication

    return value;
}
*/
void main(void)
{
  char temp_buf[20];//, vol_buf[20];
  uint8 temp;//, vol;
  InitUart();       //baudrate:   57600
  InitLed();
  LED1 = 0;
  while(1)
  {
    LED2 = !LED2;           //LED2 blink  表示程序运行正常
    temp = readTemp();   //read temperature value
    //vol = readVoltage();
    sprintf(temp_buf, (char*)"temperature:%d\r\n",   temp);
    //sprintf(vol_buf,   (char*)"vol:%d\r\n", vol);
    prints(temp_buf);
    //prints(vol_buf);
    Delay(50000);     Delay(50000);   Delay(50000);
  }
}
```

程序通过配置 CC2530 处理器的 A\D 控制器来将片内温度传感器转化，并在串口将温度值输出。

4. 实验步骤

1）使用 ZigBee Debuger USB 仿真器连接 PC 机和 ZigBee(CC2530)模块，打开 ZigBee 模块开关供电。将系统配套串口线一端连接 PC 机，一端连接 ZigBee 调试板的串口上。

启动 IAR 开发环境，新建工程，或直接使用 Exp6 实验工程。

2）在 IAR 开发环境中编译、运行、调试程序。

3）使用 PC 机自带的超级终端连接串口，将超级终端设置为串口波特率 57600、8 位、无奇偶校验、无硬件流模式，即可在终端收到模块传递过来的温度值。如图所示。

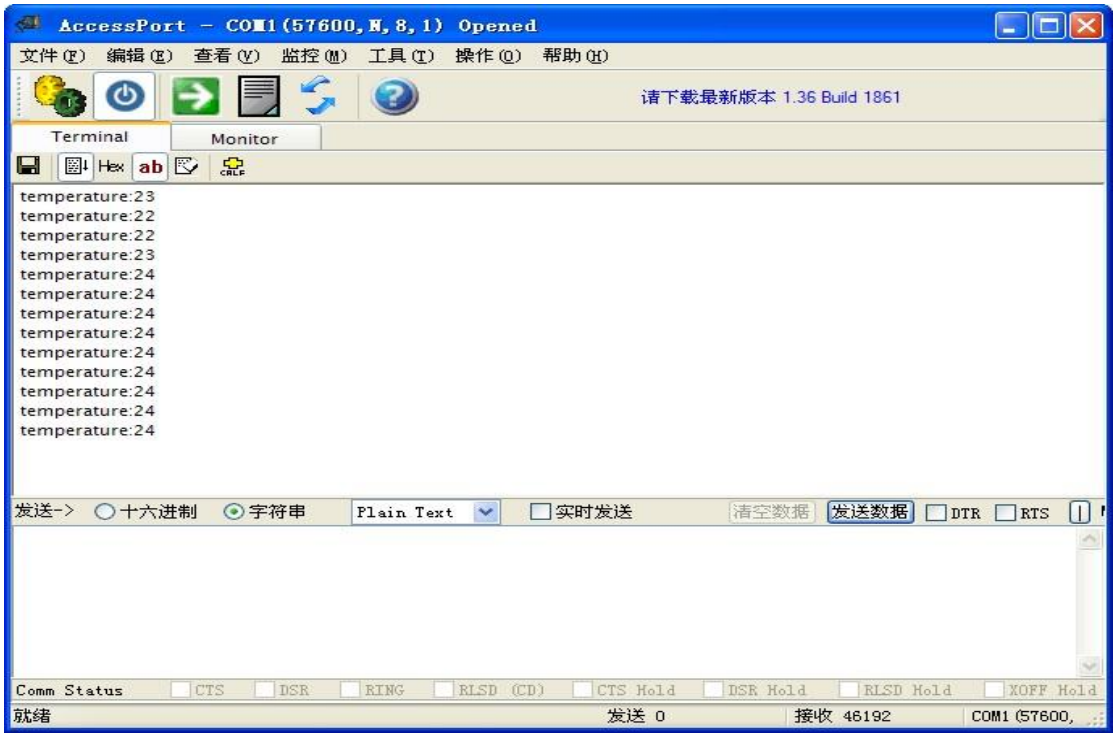


图 4.1 串口输出显示

实验八. 模拟电压 AD 转换实验

1. 实验环境

- 硬件：ZigBee(CC2530)模块，ZigBee 下载调试板，USB 仿真器，PC 机。
- 软件：IAR Embedded Workbench for MCS-51

2. 实验内容

- 阅读 ZigBee2530 开发套件 ZigBee 模块硬件部分文档，熟悉 ZigBee 模块硬件接口。
- 使用 IAR 开发环境设计程序，利用 CC2530 的 1/3 模拟电压源作为 A\D 输入源，将转换后的数值利用串口发送给 PC 机终端。

3. 实验原理

3.1 硬件接口原理

CLKCONCMD (0xC6) – Clock Control Command

Bit	Name	Reset	R/W	Description
7	OSC32K	1	R/W	32 kHz clock-source select. Setting this bit initiates a clock-source change only. CLKCONSTA.OSC32K reflects the current setting. The 16 MHz RCOSC must be selected as system clock when this bit is to be changed. 0: 32 kHz XOSC 1: 32 kHz RCOSC
6	OSC	1	R/W	System clock-source select. Setting this bit initiates a clock-source change only. CLKCONSTA.OSC reflects the current setting. 0: 32 MHz XOSC 1: 16 MHz RCOSC
5:3	TICKSPD[2:0]	001	R/W	Timer ticks output setting. Cannot be higher than system clock setting given by OSC bit setting. 000: 32 MHz 001: 16 MHz 010: 8 MHz 011: 4 MHz 100: 2 MHz 101: 1 MHz 110: 500 kHz 111: 250 kHz Note that CLKCONCMD.TICKSPD can be set to any value, but the effect is limited by the CLKCONCMD.OSC setting; i.e., if CLKCONCMD.OSC = 1 and CLKCONCMD.TICKSPD = 000, CLKCONSTA.TICKSPD reads 001, and the real TICKSPD is 16 MHz.
2:0	CLKSPD	001	R/W	Clock speed. Cannot be higher than system clock setting given by the OSC bit setting. Indicates current system-clock frequency 000: 32 MHz 001: 16 MHz 010: 8 MHz 011: 4 MHz 100: 2 MHz 101: 1 MHz 110: 500 kHz 111: 250 kHz Note that CLKCONCMD.CLKSPD can be set to any value, but the effect is limited by the CLKCONCMD.OSC setting; i.e., if CLKCONCMD.OSC = 1 and CLKCONCMD.CLKSPD = 000, CLKCONSTA.CLKSPD reads 001, and the real CLKSPD is 16 MHz. Note also that the debugger cannot be used with a divided system clock. When running the debugger, the value of CLKCONCMD.CLKSPD should be set to 000 when CLKCONCMD.OSC = 0 or to 001 when CLKCONCMD.OSC = 1.

CLKCONSTA (0x9E) – Clock Control Status

Bit	Name	Reset	R/W	Description
7	OSC32K	1	R	Current 32-kHz clock source selected: 0: 32-kHz XOSC 1: 32-kHz RCOSC
6	OSC	1	R	Current system clock selected: 0: 32-MHz XOSC 1: 16-MHz RCOSC
5:3	TICKSPD[2:0]	001	R	Current timer ticks output setting 000: 32 MHz 001: 16 MHz 010: 8 MHz 011: 4 MHz 100: 2 MHz 101: 1 MHz 110: 500 kHz 111: 250 kHz
2:0	CLKSPD	001	R	Current clock speed 000: 32 MHz 001: 16 MHz 010: 8 MHz 011: 4 MHz 100: 2 MHz 101: 1 MHz 110: 500 kHz 111: 250 kHz

表 3.1.4 CLKCONCMD 和 CLKCONSTA 寄存器

SLEEPCMD (0xBE) – Sleep-Mode Control Command

Bit	Name	Reset	R/W	Description
7	OSC32K_CALDIS	0	R/W	Disable 32-kHz RC oscillator calibration 0: 32-kHz RC oscillator calibration is enabled. 1: 32-kHz RC oscillator calibration is disabled. This setting can be written at any time, but does not take effect before the chip has been running on the 16-MHz high-frequency RC oscillator.
6:3	–	0000	R0	Reserved
2	–	1	R/W	Reserved. Always write as 1
1:0	MODE[1:0]	00	R/W	Power-mode setting 00: Active / Idle mode 01: Power mode 1 (PM1) 10: Power mode 2 (PM2) 11: Power mode 3 (PM3)

SLEEPSTA (0x9D) – Sleep-Mode Control Status

Bit	Name	Reset	R/W	Description
7	OSC32K_CALDIS	0	R	32-kHz RC oscillator calibration status SLEEPSTA.OSC32K_CALDIS shows the current status of disabling of the 32-kHz RC calibration. The bit is not set to the same value as SLEEPCMD.OSC32K_CALDIS before the chip has been run on the 32-kHz RC oscillator.
6:5	–	00	R	Reserved
4:3	RST[1:0]	XX	R	Status bit indicating the cause of the last reset. If there are multiple resets, the register only contains the last event. 00: Power-on reset and brownout detection 01: External reset 10: Watchdog Timer reset 11: Clock loss reset
2:1	–	00	R	Reserved
0	CLK32K	0	R	The 32-kHz clock signal (synchronized to the system clock)

表 3.1.5 SLEEPCMD 和 SLEEPSTA 控制寄存器

PERCFG (0xF1) – Peripheral Control

Bit	Name	Reset	R/W	Description
7	–	0	R0	Reserved
6	T1CFG	0	R/W	Timer 1 I/O location 0: Alternative 1 location 1: Alternative 2 location
5	T3CFG	0	R/W	Timer 3 I/O location 0: Alternative 1 location 1: Alternative 2 location
4	T4CFG	0	R/W	Timer 4 I/O location 0: Alternative 1 location 1: Alternative 2 location
3:2	–	00	R/W	Reserved
1	U1CFG	0	R/W	USART 1 I/O location 0: Alternative 1 location 1: Alternative 2 location
0	U0CFG	0	R/W	USART 0 I/O location 0: Alternative 1 location 1: Alternative 2 location

表 3.1.6　PERCFG 寄存器

U0CSR (0x86) – USART 0 Control and Status

Bit	Name	Reset	R/W	Description
7	MODE	0	R/W	USART mode select 0: SPI mode 1: UART mode
6	RE	0	R/W	UART receiver enable. Note do not enable receive before uart is fully configured. 0: Receiver disabled 1: Receiver enabled
5	SLAVE	0	R/W	SPI master or slave mode select 0: SPI master 1: SPI slave
4	FE	0	R/W0	UART framing error status. This bit is automatically cleared on a read of the U0CSR register or bits in the U0CSR register. 0: No framing error detected 1: Byte received with incorrect stop-bit level
3	ERR	0	R/W0	UART parity error status. This bit is automatically cleared on a read of the U0CSR register or bits in the U0CSR register. 0: No parity error detected 1: Byte received with parity error
2	RX_BYTE	0	R/W0	Receive byte status. UART mode and SPI slave mode. This bit is automatically cleared when reading U0DBUF, clearing this bit by writing 0 to it, effectively discards the data in U0DBUF. 0: No byte received 1: Received byte ready
1	TX_BYTE	0	R/W0	Transmit byte status. UART mode and SPI master mode 0: Byte not transmitted 1: Last byte written to data-buffer register transmitted
0	ACTIVE	0	R	USART transmit/receive active status. In SPI slave mode, this bit equals slave select. 0: USART idle 1: USART busy in transmit or receive mode

表 3.1.7　U0CSR 寄存器

U0GCR (0xC5) – USART 0 Generic Control

Bit	Name	Reset	R/W	Description
7	CPOL	0	R/W	SPI clock polarity 0: Negative clock polarity 1: Positive clock polarity
6	CPHA	0	R/W	SPI clock phase 0: Data is output on *MOSI* when *SCK* goes from CPOL inverted to CPOL, and data input is sampled on *MISO* when *SCK* goes from CPOL to CPOL inverted. 1: Data is output on *MOSI* when *SCK* goes from CPOL to CPOL inverted, and data input is sampled on *MISO* when *SCK* goes from CPOL inverted to CPOL.
5	ORDER	0	R/W	Bit order for transfers 0: LSB first 1: MSB first
4:0	BAUD_E[4:0]	00000	R/W	Baud rate exponent value. BAUD_E along with BAUD_M determines the UART baud rate and the SPI master SCK clock frequency.

表 3.1.8　U0GCR 寄存器

U0DBUF (0xC1) – USART 0 Receive/Transmit Data Buffer

Bit	Name	Reset	R/W	Description
7:0	DATA[7:0]	0x00	R/W	USART receive and transmit data. When writing this register, the data written is written to the internal transmit-data register. When reading this register, the data from the internal read-data register is read.

U0BAUD (0xC2) – USART 0 Baud-Rate Control

Bit	Name	Reset	R/W	Description
7:0	BAUD_M[7:0]	0x00	R/W	Baud-rate mantissa value. BAUD_E along with BAUD_M decides the UART baud rate and the SPI master SCK clock frequency.

表 3.1.9　U0DBUF 和 U0BAUD 寄存器

ADCCON1 (0xB4) – ADC Control 1

Bit	Name	Reset	R/W	Description
7	EOC	0	R/H0	End of conversion. Cleared when ADCH has been read. If a new conversion is completed before the previous data has been read, the EOC bit remains high. 0: Conversion not complete 1: Conversion completed
6	ST	0		Start conversion. Read as 1 until conversion has completed 0: No conversion in progress 1: Start a conversion sequence if ADCCON1.STSEL = 11 and no sequence is running.
5:4	STSEL[1:0]	11	R/W1	Start select. Selects the event that starts a new conversion sequence 00: External trigger on P2.0 pin 01: Full speed. Do not wait for triggers 10: Timer 1 channel 0 compare event 11: ADCCON1.ST = 1
3:2	-	00	R/W	See ADCCON1 (0xB4) – ADC Control 1 description in Section 14.3.
1:0	–	11	R/W	Reserved. Always set to 11

表 3.1.10　ADCCON1 寄存器

ADCCON3 (0xB6) – ADC Control 3

Bit	Name	Reset	R/W	Description
7:6	EREF[1:0]	00	R/W	Selects reference voltage used for the extra conversion 00: Internal reference 01: External reference on AIN7 pin 10: AVDD5 pin 11: External reference on AIN6–AIN7 differential input
5:4	EDIV[1:0]	00	R/W	Sets the decimation rate used for the extra conversion. The decimation rate also determines the resolution and time required to complete the conversion. 00: 64 decimation rate (7 bits ENOB) 01: 128 decimation rate (9 bits ENOB) 10: 256 decimation rate (10 bits ENOB) 11: 512 decimation rate (12 bits ENOB)
3:0	ECH[3:0]	0000	R/W	Single channel select. Selects the channel number of the single conversion that is triggered by writing to ADCCON3. 0000: AIN0 0001: AIN1 0010: AIN2 0011: AIN3 0100: AIN4 0101: AIN5 0110: AIN6 0111: AIN7 1000: AIN0–AIN1 1001: AIN2–AIN3 1010: AIN4–AIN5 1011: AIN6–AIN7 1100: GND 1101: Reserved 1110: Temperature sensor 1111: VDD/3

表 3.1.11 ADCCON3 寄存器

以上图表列举了和 CC2530 处理器 A/D 转换操作 相关的寄存器，其中包括 CLKCONCMD 和 CLKCONSTA 控制寄存器，用来控制系统时钟源和状态，SLEEPCMD 和 SLEEPSTA 寄存器用来控制各种时钟源的开关和状态。PERCFG 寄存器为外设功能控制寄存器，用来控制外设功能模式。 U0CSR、 U0GCR、U0BUF、 U0BAUD 等为串口相关寄存器。ADCCON1 和 ADCCON3 分别为 AD 转换控制器和 AD 转换设置寄存器。

3.2 软件设计

```
/*************************************************************************
*函数功能 ：初始化 ADC                                    *
*入口参数 ：无                                            *
*返 回 值 ：无                                        *
*说      明 ：参考电压 AVDD，转换对象是 AVDD              *
*************************************************************************/
void InitialAD(void)
{
P1DIR = 0x03;  //P1 控制 LED
led1 = 1;
    led2 = 1;          //关 LED

    ADCCON1 &= ~0X80;    //清 EOC 标志
```

```
    ADCCON3=0xbf;    //单次转换,参考电压为电源电压，对 1/3 AVDD 进行 A/D 转换
                     //12 位分辨率
    ADCCON1 = 0X30; //停止 A/D
    ADCCON1 |= 0X40; //启动 A/D
}

/************************************************************
*函数功能 ： 主函数                                        *
*入口参数 ： 无                                            *
*返 回 值 ： 无                                            *
*说     明 ： 无                                            *
************************************************************/
void main(void)
{
    char temp[2];
    float num;
    InitUart();                 //   baudrate: 57600
    InitialAD();                //   初始化 ADC

    led1 = 1;
    while(1)
    {
        if(ADCCON1>=0x80)
        {
            led1 = 1;           //转换完毕指示
            temp[1] = ADCL;
            temp[0] = ADCH;
            ADCCON1 |= 0x40;        //开始下一转换

            temp[1] = temp[1]>>2;               //数据处理
            temp[1] |= temp[0]<<6;

            temp[0] = temp[0]>>2;
            temp[0] &= 0x3f;

            num = (temp[0]*256+temp[1])*3.3/4096;   //12 位，  取 2^12;
            num = num/2+0.05;                   //四  舍五入处理
            //定参考电压为 3.3V。12 位精确度
            adcdata[1] = (char)(num)%10+48;
            adcdata[3] = (char)(num*10)%10+48;

            prints(adcdata);    //将模拟电压值发送到串口

            Delay(30000);
            led1 = 0;                           //完成数据处理
            Delay(30000);
```

```
        }
    }
}
```

程序通过配置 CC2530 处理器的 1/3 模拟电压作为 A/D 转换的输入源，并将转换后的结果在串口输出。

4. 实验步骤

1）使用 ZigBee Debuger USB 仿真器连接 PC 机和 ZigBee(CC2530)模块，打开 ZigBee 模块开关供电。将系统配套串口线一端连接 PC 机，一端连接 ZigBee 调试板的串口上。

2）启动 IAR 开发环境，新建工程，或直接使用 Exp7 实验工程。

3）在 IAR 开发环境中编译、运行、调试程序。

4）使用 PC 机自带的超级终端连接串口，将超级终端设置为串口波特率 57600、8 位、无奇偶奇校验、无硬件流模式，即可在终端收到模块传递过来的模拟电压经过 A\D 转换后的数值。

实验九. 电源电压 AD 转换实验

1. 实验环境

- 硬件：ZigBee(CC2530)模块，ZigBee 下载调试板，USB 仿真器，PC 机。
- 软件：IAR Embedded Workbench for MCS-51

2. 实验内容

- 阅读 ZigBee2530 开发套件 ZigBee 模块硬件部分文档，熟悉 ZigBee 模块硬件接口
- 使用 IAR 开发环境设计程序，利用 CC2530 的电源作为 A\D 输入源，将转换后的数值利用串口发送给 PC 机终端

3. 实验原理

3.1 硬件接口原理

CLKCONCMD (0xC6) – Clock Control Command

Bit	Name	Reset	R/W	Description
7	OSC32K	1	R/W	32 kHz clock-source select. Setting this bit initiates a clock-source change only. CLKCONSTA.OSC32K reflects the current setting. The 16 MHz RCOSC must be selected as system clock when this bit is to be changed. 0: 32 kHz XOSC 1: 32 kHz RCOSC
6	OSC	1	R/W	System clock-source select. Setting this bit initiates a clock-source change only. CLKCONSTA.OSC reflects the current setting. 0: 32 MHz XOSC 1: 16 MHz RCOSC
5:3	TICKSPD[2:0]	001	R/W	Timer ticks output setting. Cannot be higher than system clock setting given by OSC bit setting. 000: 32 MHz 001: 16 MHz 010: 8 MHz 011: 4 MHz 100: 2 MHz 101: 1 MHz 110: 500 kHz 111: 250 kHz Note that CLKCONCMD.TICKSPD can be set to any value, but the effect is limited by the CLKCONCMD.OSC setting; i.e., if CLKCONCMD.OSC = 1 and CLKCONCMD.TICKSPD = 000, CLKCONSTA.TICKSPD reads 001, and the real TICKSPD is 16 MHz.
2:0	CLKSPD	001	R/W	Clock speed. Cannot be higher than system clock setting given by the OSC bit setting. Indicates current system-clock frequency 000: 32 MHz 001: 16 MHz 010: 8 MHz 011: 4 MHz 100: 2 MHz 101: 1 MHz 110: 500 kHz 111: 250 kHz Note that CLKCONCMD.CLKSPD can be set to any value, but the effect is limited by the CLKCONCMD.OSC setting; i.e., if CLKCONCMD.OSC = 1 and CLKCONCMD.CLKSPD = 000, CLKCONSTA.CLKSPD reads 001, and the real CLKSPD is 16 MHz. Note also that the debugger cannot be used with a divided system clock. When running the debugger, the value of CLKCONCMD.CLKSPD should be set to 000 when CLKCONCMD.OSC = 0 or to 001 when CLKCONCMD.OSC = 1.

CLKCONSTA (0x9E) – Clock Control Status

Bit	Name	Reset	R/W	Description
7	OSC32K	1	R	Current 32-kHz clock source selected: 0: 32-kHz XOSC 1: 32-kHz RCOSC
6	OSC	1	R	Current system clock selected: 0: 32-MHz XOSC 1: 16-MHz RCOSC
5:3	TICKSPD[2:0]	001	R	Current timer ticks outputs setting 000: 32 MHz 001: 16 MHz 010: 8 MHz 011: 4 MHz 100: 2 MHz 101: 1 MHz 110: 500 kHz 111: 250 kHz
2:0	CLKSPD	001	R	Current clock speed 000: 32 MHz 001: 16 MHz 010: 8 MHz 011: 4 MHz 100: 2 MHz 101: 1 MHz 110: 500 kHz 111: 250 kHz

表 3.1.4　CLKCONCMD 和 CLKCONSTA 寄存器

SLEEPCMD (0xBE) – Sleep-Mode Control Command

Bit	Name	Reset	R/W	Description
7	OSC32K_CALDIS	0	R/W	Disable 32-kHz RC oscillator calibration 0: 32-kHz RC oscillator calibration is enabled. 1: 32-kHz RC oscillator calibration is disabled. This setting can be written at any time, but does not take effect before the chip has been running on the 16-MHz high-frequency RC oscillator.
6:3	–	0000	R0	Reserved
2	–	1	R/W	Reserved. Always write as 1
1:0	MODE[1:0]	00	R/W	Power-mode setting 00: Active / Idle mode 01: Power mode 1 (PM1) 10: Power mode 2 (PM2) 11: Power mode 3 (PM3)

SLEEPSTA (0x9D) – Sleep-Mode Control Status

Bit	Name	Reset	R/W	Description
7	OSC32K_CALDIS	0	R	32-kHz RC oscillator calibration status SLEEPSTA.OSC32K_CALDIS shows the current status of disabling of the 32-kHz RC calibration. The bit is not set to the same value as SLEEPCMD.OSC32K_CALDIS before the chip has been run on the 32-kHz RC oscillator.
6:5	–	00	R	Reserved
4:3	RST[1:0]	XX	R	Status bit indicating the cause of the last reset. If there are multiple resets, the register only contains the last event. 00: Power-on reset and brownout detection 01: External reset 10: Watchdog Timer reset 11: Clock loss reset
2:1	–	00	R	Reserved
0	CLK32K	0	R	The 32-kHz clock signal (synchronized to the system clock)

表 3.1.5　SLEEPCMD 和 SLEEPSTA 控制寄存器

PERCFG (0xF1) – Peripheral Control

Bit	Name	Reset	R/W	Description
7	–	0	R0	Reserved
6	T1CFG	0	R/W	Timer 1 I/O location 0: Alternative 1 location 1: Alternative 2 location
5	T3CFG	0	R/W	Timer 3 I/O location 0: Alternative 1 location 1: Alternative 2 location
4	T4CFG	0	R/W	Timer 4 I/O location 0: Alternative 1 location 1: Alternative 2 location
3:2	–	00	R/W	Reserved
1	U1CFG	0	R/W	USART 1 I/O location 0: Alternative 1 location 1: Alternative 2 location
0	U0CFG	0	R/W	USART 0 I/O location 0: Alternative 1 location 1: Alternative 2 location

表 3.1.6　PERCFG 寄存器

U0CSR (0x86) – USART 0 Control and Status

Bit	Name	Reset	R/W	Description
7	MODE	0	R/W	USART mode select 0: SPI mode 1: UART mode
6	RE	0	R/W	UART receiver enable. Note do not enable receive before uart is fully configured. 0: Receiver disabled 1: Receiver enabled
5	SLAVE	0	R/W	SPI master or slave mode select 0: SPI master 1: SPI slave
4	FE	0	R/W0	UART framing error status. This bit is automatically cleared on a read of the U0CSR register or bits in the U0CSR register. 0: No framing error detected 1: Byte received with incorrect stop-bit level
3	ERR	0	R/W0	UART parity error status. This bit is automatically cleared on a read of the U0CSR register or bits in the U0CSR register. 0: No parity error detected 1: Byte received with parity error
2	RX_BYTE	0	R/W0	Receive byte status. UART mode and SPI slave mode. This bit is automatically cleared when reading U0DBUF, clearing this bit by writing 0 to it, effectively discards the data in U0DBUF. 0: No byte received 1: Received byte ready
1	TX_BYTE	0	R/W0	Transmit byte status. UART mode and SPI master mode 0: Byte not transmitted 1: Last byte written to data-buffer register transmitted
0	ACTIVE	0	R	USART transmit/receive active status. In SPI slave mode, this bit equals slave select. 0: USART idle 1: USART busy in transmit or receive mode

表 3.1.7　U0CSR 寄存器

U0GCR (0xC5) – USART 0 Generic Control

Bit	Name	Reset	R/W	Description
7	CPOL	0	R/W	SPI clock polarity 0: Negative clock polarity 1: Positive clock polarity
6	CPHA	0	R/W	SPI clock phase 0: Data is output on *MOSI* when *SCK* goes from CPOL inverted to CPOL, and data input is sampled on *MISO* when *SCK* goes from CPOL to CPOL inverted. 1: Data is output on *MOSI* when *SCK* goes from CPOL to CPOL inverted, and data input is sampled on *MISO* when *SCK* goes from CPOL inverted to CPOL.
5	ORDER	0	R/W	Bit order for transfers 0: LSB first 1: MSB first
4:0	BAUD_E[4:0]	00000	R/W	Baud rate exponent value. BAUD_E along with BAUD_M determines the UART baud rate and the SPI master SCK clock frequency.

表 3.1.8　U0GCR 寄存器

U0DBUF (0xC1) – USART 0 Receive/Transmit Data Buffer

Bit	Name	Reset	R/W	Description
7:0	DATA[7:0]	0x00	R/W	USART receive and transmit data. When writing this register, the data written is written to the internal transmit-data register. When reading this register, the data from the internal read-data register is read.

U0BAUD (0xC2) – USART 0 Baud-Rate Control

Bit	Name	Reset	R/W	Description
7:0	BAUD_M[7:0]	0x00	R/W	Baud-rate mantissa value. BAUD_E along with BAUD_M decides the UART baud rate and the SPI master SCK clock frequency.

表 3.1.9　U0DBUF 和 U0BAUD 寄存器

ADCCON1 (0xB4) – ADC Control 1

Bit	Name	Reset	R/W	Description
7	EOC	0	R/H0	End of conversion. Cleared when ADCH has been read. If a new conversion is completed before the previous data has been read, the EOC bit remains high. 0: Conversion not complete 1: Conversion completed
6	ST	0		Start conversion. Read as 1 until conversion has completed 0: No conversion in progress 1: Start a conversion sequence if ADCCON1.STSEL = 11 and no sequence is running.
5:4	STSEL[1:0]	11	R/W1	Start select. Selects the event that starts a new conversion sequence 00: External trigger on P2.0 pin 01: Full speed. Do not wait for triggers 10: Timer 1 channel 0 compare event 11: ADCCON1.ST = 1
3:2	-	00	R/W	See ADCCON1 (0xB4) – ADC Control 1 description in Section 14.3.
1:0	–	11	R/W	Reserved. Always set to 11

表 3.1.10　ADCCON1 寄存器

ADCCON3 (0xB6) – ADC Control 3

Bit	Name	Reset	R/W	Description
7:6	EREF[1:0]	00	R/W	Selects reference voltage used for the extra conversion 00: Internal reference 01: External reference on AIN7 pin 10: AVDD5 pin 11: External reference on AIN6–AIN7 differential input
5:4	EDIV[1:0]	00	R/W	Sets the decimation rate used for the extra conversion. The decimation rate also determines the resolution and time required to complete the conversion. 00: 64 decimation rate (7 bits ENOB) 01: 128 decimation rate (9 bits ENOB) 10: 256 decimation rate (10 bits ENOB) 11: 512 decimation rate (12 bits ENOB)
3:0	ECH[3:0]	0000	R/W	Single channel select. Selects the channel number of the single conversion that is triggered by writing to ADCCON3. 0000: AIN0 0001: AIN1 0010: AIN2 0011: AIN3 0100: AIN4 0101: AIN5 0110: AIN6 0111: AIN7 1000: AIN0–AIN1 1001: AIN2–AIN3 1010: AIN4–AIN5 1011: AIN6–AIN7 1100: GND 1101: Reserved 1110: Temperature sensor 1111: VDD/3

表 3.1.11 ADCCON3 寄存器

以上图表列举了和 CC2530 处理器 A/D 转换操作相关的寄存器，其中包括 CLKCONCMD 和 CLKCONSTA 控制寄存器，用来控制系统时钟源和状态， SLEEPCMD 和 SLEEPSTA 寄存器用来控制各种时钟源的开关和状态。 PERCFG 寄存器为外设功能控制寄存器，用来控制外设功能模式。 U0CSR、 U0GCR、U0BUF、U0BAUD 等为串口相关寄存器。 ADCCON1 和 ADCCON3 分别为 AD 转换控制器和 AD 转换设置寄存器。

3.2 软件设计

```
/*************************************************************
*函数功能 ：初始化 ADC                                    *
*入口参数 ：无                                            *
*返 回 值 ：无                                        *
*说      明 ：参考电压 AVDD，转换对象是 AVDD              *
*************************************************************/
void InitialAD(void)
{
    P1DIR = 0x03;  //P1 控制 LED
    led1 = 1;
    led2 = 1;          //关 LED

    ADCCON1 &= ~0X80;    //清 EOC 标志
```

```
    ADCCON3=0xbd;    //单次转换,参考电压为电源电压，AVDD 进行 A/D 转换
                     //12 位分辨率
    ADCCON1 = 0X30;  //停止 A/D
    ADCCON1 |= 0X40; //启动 A/D
}

/*************************************************************
*函数功能 ：主函数                                          *
*入口参数 ：无                                              *
*返 回 值 ：无                                              *
*说    明 ：无                                              *
*************************************************************/
void main(void)
{
    char temp[2];
    float num;
    InitUart();                 //   baudrate: 57600
    InitialAD();                //   初始化 ADC

    led1 = 1;
    while(1)
    {
        if(ADCCON1>=0x80)
        {
            led1 = 1;           //转换完毕指示
            temp[1] = ADCL;
            temp[0] = ADCH;
            ADCCON1 |= 0x40;    //开始下一转换

            temp[1] = temp[1]>>2;           //数据处理
            temp[1] |= temp[0]<<6;

            temp[0] = temp[0]>>2;
            temp[0] &= 0x3f;

            num = (temp[0]*256+temp[1])*3.3/2048;
            num += 0.05;    //四舍五入处理
            //定参考电压为 3.3V。12 位精确度
            adcdata[1] = (char)(num)%10+48;
            adcdata[3] = (char)(num*10)%10+48;

            prints(adcdata);    //将电源电压值发送到串口

            Delay(30000);
            led1 = 0;                       //完成数据处理
            Delay(30000);
```

```
        }
    }
}
```

程序通过配置 CC2530 处理器的电源电压作为 A/D 转换的输入源，并将转换后的结果在串口输出。

4. 实验步骤

1）使用 ZigBee Debuger USB 仿真器连接 PC 机和 ZigBee(CC2530)模块，打开 ZigBee 模块开关供电。将系统配套串口线一端连接 PC 机，一端连接 ZigBee 调试板的串口上。

2）启动 IAR 开发环境，新建工程，或直接使用 Exp8 实验工程。

3）在 IAR 开发环境中编译、运行、调试程序。

4）使用 PC 机自带的超级终端连接串口，将超级终端设置为串口波特率 57600、8 位、无奇偶奇校验、无硬件流模式，即可在终端收到模块传递过来的模拟电压经过 A\D 转换后的数值。

实验十. 串口收发数据实验

1. 实验环境

- 硬件：ZigBee(CC2530)模块，ZigBee 下载调试板，USB 仿真器，PC 机。
- 软件：IAR Embedded Workbench for MCS-51。

2. 实验内容

- 阅读 ZigBee2530 开发套件 ZigBee 模块硬件部分文档，熟悉 ZigBee 模块硬件接口。
- 使用 IAR 开发环境设计程序，利用 CC2530 的串口 0 进行数据收发通讯。

3. 实验原理

3.1 硬件接口原理

CLKCONCMD (0xC6) – Clock Control Command

Bit	Name	Reset	R/W	Description
7	OSC32K	1	R/W	32 kHz clock-source select. Setting this bit initiates a clock-source change only. CLKCONSTA.OSC32K reflects the current setting. The 16 MHz RCOSC must be selected as system clock when this bit is to be changed. 0: 32 kHz XOSC 1: 32 kHz RCOSC
6	OSC	1	R/W	System clock-source select. Setting this bit initiates a clock-source change only. CLKCONSTA.OSC reflects the current setting. 0: 32 MHz XOSC 1: 16 MHz RCOSC
5:3	TICKSPD[2:0]	001	R/W	Timer ticks output setting. Cannot be higher than system clock setting given by OSC bit setting. 000: 32 MHz 001: 16 MHz 010: 8 MHz 011: 4 MHz 100: 2 MHz 101: 1 MHz 110: 500 kHz 111: 250 kHz Note that CLKCONCMD.TICKSPD can be set to any value, but the effect is limited by the CLKCONCMD.OSC setting; i.e., if CLKCONCMD.OSC = 1 and CLKCONCMD.TICKSPD = 000, CLKCONSTA.TICKSPD reads 001, and the real TICKSPD is 16 MHz.
2:0	CLKSPD	001	R/W	Clock speed. Cannot be higher than system clock setting given by the OSC bit setting. Indicates current system-clock frequency 000: 32 MHz 001: 16 MHz 010: 8 MHz 011: 4 MHz 100: 2 MHz 101: 1 MHz 110: 500 kHz 111: 250 kHz Note that CLKCONCMD.CLKSPD can be set to any value, but the effect is limited by the CLKCONCMD.OSC setting; i.e., if CLKCONCMD.OSC = 1 and CLKCONCMD.CLKSPD = 000, CLKCONSTA.CLKSPD reads 001, and the real CLKSPD is 16 MHz. Note also that the debugger cannot be used with a divided system clock. When running the debugger, the value of CLKCONCMD.CLKSPD should be set to 000 when CLKCONCMD.OSC = 0 or to 001 when CLKCONCMD.OSC = 1.

CLKCONSTA (0x9E) – Clock Control Status

Bit	Name	Reset	R/W	Description
7	OSC32K	1	R	Current 32-kHz clock source selected: 0: 32-kHz XOSC 1: 32-kHz RCOSC
6	OSC	1	R	Current system clock selected: 0: 32-MHz XOSC 1: 16-MHz RCOSC
5:3	TICKSPD[2:0]	001	R	Current timer ticks output setting 000: 32 MHz 001: 16 MHz 010: 8 MHz 011: 4 MHz 100: 2 MHz 101: 1 MHz 110: 500 kHz 111: 250 kHz
2:0	CLKSPD	001	R	Current clock speed 000: 32 MHz 001: 16 MHz 010: 8 MHz 011: 4 MHz 100: 2 MHz 101: 1 MHz 110: 500 kHz 111: 250 kHz

表 3.1.4　CLKCONCMD 和 CLKCONSTA 寄存器

SLEEPCMD (0xBE) – Sleep-Mode Control Command

Bit	Name	Reset	R/W	Description
7	OSC32K_CALDIS	0	R/W	Disable 32-kHz RC oscillator calibration 0: 32-kHz RC oscillator calibration is enabled. 1: 32-kHz RC oscillator calibration is disabled. This setting can be written at any time, but does not take effect before the chip has been running on the 16-MHz high-frequency RC oscillator.
6:3	–	0000	R0	Reserved
2	–	1	R/W	Reserved. Always write as 1
1:0	MODE[1:0]	00	R/W	Power-mode setting 00: Active / Idle mode 01: Power mode 1 (PM1) 10: Power mode 2 (PM2) 11: Power mode 3 (PM3)

SLEEPSTA (0x9D) – Sleep-Mode Control Status

Bit	Name	Reset	R/W	Description
7	OSC32K_CALDIS	0	R	32-kHz RC oscillator calibration status SLEEPSTA.OSC32K_CALDIS shows the current status of disabling of the 32-kHz RC calibration. The bit is not set to the same value as SLEEPCMD.OSC32K_CALDIS before the chip has been run on the 32-kHz RC oscillator.
6:5	–	00	R	Reserved
4:3	RST[1:0]	XX	R	Status bit indicating the cause of the last reset. If there are multiple resets, the register only contains the last event. 00: Power-on reset and brownout detection 01: External reset 10: Watchdog Timer reset 11: Clock loss reset
2:1	–	00	R	Reserved
0	CLK32K	0	R	The 32-kHz clock signal (synchronized to the system clock)

表 3.1.5　SLEEPCMD 和 SLEEPSTA 控制寄存器

PERCFG (0xF1) – Peripheral Control

Bit	Name	Reset	R/W	Description
7	–	0	R0	Reserved
6	T1CFG	0	R/W	Timer 1 I/O location 0: Alternative 1 location 1: Alternative 2 location
5	T3CFG	0	R/W	Timer 3 I/O location 0: Alternative 1 location 1: Alternative 2 location
4	T4CFG	0	R/W	Timer 4 I/O location 0: Alternative 1 location 1: Alternative 2 location
3:2	–	00	R/W	Reserved
1	U1CFG	0	R/W	USART 1 I/O location 0: Alternative 1 location 1: Alternative 2 location
0	U0CFG	0	R/W	USART 0 I/O location 0: Alternative 1 location 1: Alternative 2 location

表 3.1.6 PERCFG 寄存器

U0CSR (0x86) – USART 0 Control and Status

Bit	Name	Reset	R/W	Description
7	MODE	0	R/W	USART mode select 0: SPI mode 1: UART mode
6	RE	0	R/W	UART receiver enable. Note do not enable receive before uart is fully configured. 0: Receiver disabled 1: Receiver enabled
5	SLAVE	0	R/W	SPI master or slave mode select 0: SPI master 1: SPI slave
4	FE	0	R/W0	UART framing error status. This bit is automatically cleared on a read of the U0CSR register or bits in the U0CSR register. 0: No framing error detected 1: Byte received with incorrect stop-bit level
3	ERR	0	R/W0	UART parity error status. This bit is automatically cleared on a read of the U0CSR register or bits in the U0CSR register. 0: No parity error detected 1: Byte received with parity error
2	RX_BYTE	0	R/W0	Receive byte status. UART mode and SPI slave mode. This bit is automatically cleared when reading U0DBUF, clearing this bit by writing 0 to it, effectively discards the data in U0DBUF. 0: No byte received 1: Received byte ready
1	TX_BYTE	0	R/W0	Transmit byte status. UART mode and SPI master mode 0: Byte not transmitted 1: Last byte written to data-buffer register transmitted
0	ACTIVE	0	R	USART transmit/receive active status. In SPI slave mode, this bit equals slave select. 0: USART idle 1: USART busy in transmit or receive mode

表 3.1.7 U0CSR 寄存器

U0GCR (0xC5) – USART 0 Generic Control

Bit	Name	Reset	R/W	Description
7	CPOL	0	R/W	SPI clock polarity 0: Negative clock polarity 1: Positive clock polarity
6	CPHA	0	R/W	SPI clock phase 0: Data is output on *MOSI* when *SCK* goes from CPOL inverted to CPOL, and data input is sampled on *MISO* when *SCK* goes from CPOL to CPOL inverted. 1: Data is output on *MOSI* when *SCK* goes from CPOL to CPOL inverted, and data input is sampled on *MISO* when *SCK* goes from CPOL inverted to CPOL.
5	ORDER	0	R/W	Bit order for transfers 0: LSB first 1: MSB first
4:0	BAUD_E[4:0]	00000	R/W	Baud rate exponent value. BAUD_E along with BAUD_M determines the UART baud rate and the SPI master SCK clock frequency.

表 3.1.8　U0GCR 寄存器

U0DBUF (0xC1) – USART 0 Receive/Transmit Data Buffer

Bit	Name	Reset	R/W	Description
7:0	DATA[7:0]	0x00	R/W	USART receive and transmit data. When writing this register, the data written is written to the internal transmit-data register. When reading this register, the data from the internal read-data register is read.

U0BAUD (0xC2) – USART 0 Baud-Rate Control

Bit	Name	Reset	R/W	Description
7:0	BAUD_M[7:0]	0x00	R/W	Baud-rate mantissa value. BAUD_E along with BAUD_M decides the UART baud rate and the SPI master SCK clock frequency.

表 3.1.　U0DBUF 和 U0BAUD 寄存器

以上图表列举了和 CC2530 处理器串口操作相关的寄存器，其中包括 CLKCONCMD 控制寄存器，用来控制系统时钟源， SLEEPCMD 和 SLEEPSTA 寄存器用来控制各种时钟源的开关和状态。 PERCFG 寄存器为外设功能控制寄存器，用来控制外设功能模式。U0CSR、 U0GCR、 U0BUF、 U0BAUD 等为串口相关寄存器。

3.2 软件设计

```
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 main(void)
{
    char receive_buf[30];
    uchar counter =0;
    uchar RT_flag=1;

    InitUart();       // baudrate:57600
    InitLed();

    while(1)
```

```
    {
        if(RT_flag == 1)            //接收
        {
          LED2=0;                   //接收状态指示
          if( temp != 0)
          {
              if((temp!='\r')&&(counter<30))   //'\r'回车键为结束字符 //最多能接收 30 个字符
              {
                receive_buf[counter++] = temp;
              }
              else
              {
                RT_flag = 3;                        //进入发送状态
              }
              if(counter == 30)   RT_flag = 3;
              temp    = 0;
          }
        }

        if(RT_flag == 3)                //发送
        {
            LED2 = 1;                                   //关 LED2
            LED1 = 0;                   //发送状态指示
            U0CSR &= ~0x40;                 //禁止接收
            receive_buf[counter] = '\0';
            prints(receive_buf);
            prints("\r\n");
            U0CSR |= 0x40;      //允许接收
            RT_flag = 1;                    //恢复到接收状态
            counter = 0;            //指针归 0
            LED1 = 1;               //关发送指示
        }
    }
}

/*****************************************************************
*函数功能 ： 串口接收一个字符
*入口参数 : 无
*返 回 值 ： 无
*说      明 ： 接收完成后打开接收
*****************************************************************/
#pragma vector = URX0_VECTOR
 __interrupt void UART0_ISR(void)
{
    URX0IF = 0;                 //清中断标志
    temp = U0DBUF;
```

```
}
```

程序通过配置 CC2530 处理器的串口相关控制寄存器来设置串口 0 的工作模式为串口模式，波特率为 57600，使用中断方式接收串口数据并向串口输出。

4. 实验步骤

1）使用 ZigBee Debuger USB 仿真器连接 PC 机和 ZigBee(CC2530)模块，打开 ZigBee 模块开关供电。将系统配套串口线一端连接 PC 机，一端连接 ZigBee 调试板的串口上。

2）启动 IAR 开发环境，新建工程，或直接使用 Exp9 实验工程。

3）在 IAR 开发环境中编译、运行、调试程序。

4）使用 PC 机自带的超级终端（注意：字符串必须以回车键结束或输入字符串长度超过 30 个字符，才会显示）连接串口，将超级终端设置为串口波特率 57600、8 位、无奇偶校验，无硬件流模式，当向串口终端输入数据输入回车结束符时，将在超级终端看到串口输入的数据。

实验十一. 串口控制 LED 实验

1. 实验环境

- 硬件：ZigBee(CC2530)模块，ZigBee 下载调试板，USB 仿真器，PC 机。
- 软件：IAR Embedded Workbench for MCS-51。

2. 实验内容

- 阅读 ZigBee2530 开发套件 ZigBee 模块硬件部分文档，熟悉 ZigBee 模块硬件接口。
- 使用 IAR 开发环境设计程序，利用 CC2530 的串口 0 对板载 LED 灯进行控制。

3. 实验原理

3.1 硬件接口原理

请参考实验二和实验十的硬件接口原理

3.2 软件设计

```
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 main(void)
{
    char receive_buf[3];
    uchar counter =0;
    uchar RT_flag=1;

    InitUart();       // baudrate:57600
    InitLed();

    prints("input: 11----->led1 on     10----->led1 off     21----->led2 on     20----->led2 off\r\n");
    while(1)
    {
        if(RT_flag == 1)              //接收
```

```
        {
          if( temp != 0)
          {
              if((temp!='\r')&&(counter<3))   //'\r'回车键为结束字符 //最多能接收 3 个字符
              {
                receive_buf[counter++] = temp;
              }
              else
              {
                RT_flag = 3;                          //进入 led 设置状态
              }
              if(counter == 3)   RT_flag = 3;
              temp   = 0;
          }
        }

        if(RT_flag == 3)              //led 状态设置
        {
            U0CSR &= ~0x40;                      //禁止接收
            receive_buf[2] = '\0';
           // prints(receive_buf);          prints("\r\n");
            if(receive_buf[0] == '1')
            {
              if(receive_buf[1] == '1')            { LED1 = 0;     prints("led1 on\r\n");     }
              else if(receive_buf[1] == '0')     { LED1 = 1;     prints("led1 off\r\n");    }
            }
            else if(receive_buf[0] == '2')
            {
              if(receive_buf[1] == '1')            { LED2 = 0;     prints("led2 on\r\n");     }
              else if(receive_buf[1] == '0')     { LED2 = 1;     prints("led2 off\r\n");    }
            }
            U0CSR |= 0x40;                  //允许接收
            RT_flag = 1;                    //恢复到接收状态
            counter = 0;             //指针归 0
        }
    }
}

/*********************************************************************
*函数功能 ：串口接收一个字符
*入口参数 ：无
*返 回 值 ：无
*说      明 ：接收完成后打开接收
*********************************************************************/
#pragma vector = URX0_VECTOR
 __interrupt void UART0_ISR(void)
```

```
{
    URX0IF = 0;                    //清中断标志
    temp = U0DBUF;
}
```

程序通过配置 CC2530 处理器的串口相关控制寄存器来设置 串口 0 的工作模式为串口模式，波特率为 57600，通过判断串口的输入来控制 LED 灯的状态。

4. 实验步骤

1）使用 ZigBee Debuger USB 仿真器连接 PC 机和 ZigBee(CC2530)模块，打开 ZigBee 模块开关供电。将系统配套串口线一端连接 PC 机，一端连接 ZigBee 调试板的串口上。

2）启动 IAR 开发环境，新建工程，或直接使用 Exp10 实验工程中。

3）在 IAR 开发环境中编译、运行、调试程序。

4）使用 PC 机自带的超级终端连接串口，将超级终端设置为串口波特率 57600、8 位、无奇偶校验，无硬件流模式，当向串口输入相应数据格式的数据时，即可控制 LED 灯的开关。

LED1 开：　输入 11 回车

LED1 关：　输入 10 回车

LED2 开：　输入 21 回车

LED2 关：　输入 20 回车

实验十二. 时钟显示实验

1. 实验环境

- 硬件：ZigBee(CC2530)模块，ZigBee 下载调试板，USB 仿真器，PC 机。
- 软件：IAR Embedded Workbench for MCS-51

2. 实验内容

- 阅读 ZigBee2530 开发套件 ZigBee 模块硬件部分文档，熟悉 ZigBee 模块硬件接口。
- 使用 IAR 开发环境设计程序，利用 CC2530 的 定时器 Timer1 产生秒信号，模拟时钟向串口显示。

3. 实验原理

3.1 硬件接口原理

T1CTL (0xE4) – Timer 1 Control

Bit	Name	Reset	R/W	Description
7:4	–	0000 0	R0	Reserved
3:2	DIV[1:0]	00	R/W	Prescaler divider value. Generates the active clock edge used to update the counter as follows: 00: Tick frequency/1 01: Tick frequency/8 10: Tick frequency/32 11: Tick frequency/128
1:0	MODE [1:0]	00	R/W	Timer 1 mode select. The timer operating mode is selected as follows: 00: Operation is suspended. 01: Free-running, repeatedly count from 0x0000 to 0xFFFF. 10: Modulo, repeatedly count from 0x0000 to T1CC0. 11: Up/down, repeatedly count from 0x0000 to T1CC0 and from T1CC0 down to 0x0000.

表 3.1.4 T1CTL 寄存器

T1CCTL0 (0xE5) – Timer 1 Channel 0 Capture/Compare Control

Bit	Name	Reset	R/W	Description
7	RFIRQ	0	R/W	When set, use RF interrupt for capture instead of regular capture input.
6	IM	1	R/W	Channel 0 interrupt mask. Enables interrupt request when set.
5:3	CMP[2:0]	000	R/W	Channel 0 compare-mode select. Selects action on output when timer value equals compare value in T1CC0 000: Set output on compare 001: Clear output on compare 010: Toggle output on compare 011: Set output on compare-up, clear on 0 100: Clear output on compare-up, set on 0 101: Reserved 110: Reserved 111: Initialize output pin. CMP[2:0] is not changed.
2	MODE	0	R/W	Mode. Select Timer 1 channel 0 capture or compare mode 0: Capture mode 1: Compare mode
1:0	CAP[1:0]	00	R/W	Channel 0 capture-mode select 00: No capture 01: Capture on rising edge 10: Capture on falling edge 11: Capture on all edges

表 3.1.5　T1CCTL0 寄存器

T1CC0H (0xDB) – Timer 1 Channel 0 Capture/Compare Value, High

Bit	Name	Reset	R/W	Description
7:0	T1CC0[15:8]	0x00	R/W	Timer 1 channel 0 capture/compare value high order byte. Writing to this register when T1CCTL0.MODE = 1 (compare mode) causes the T1CC0[15:0] update to the written value to be delayed until T1CNT = 0x0000.

T1CC0L (0xDA) – Timer 1 Channel 0 Capture/Compare Value, Low

Bit	Name	Reset	R/W	Description
7:0	T1CC0[7:0]	0x00	R/W	Timer 1 channel 0 capture/compare value low order byte. Data written to this register is stored in a buffer but not written to T1CC0[7:0] until, and at the same time as, a later write to T1CC0H takes effect.

表 3.1.6　T1CC0H 和 T1CC0L 寄存器

IEN0 (0xA8) – Interrupt Enable 0

Bit	Name	Reset	R/W	Description
7	EA	0	R/W	Disables all interrupts. 0: No interrupt is acknowledged. 1: Each interrupt source is individually enabled or disabled by setting its corresponding enable bit.
6	–	0	R0	Reserved. Read as 0
5	STIE	0	R/W	Sleep Timer interrupt enable 0: Interrupt disabled 1: Interrupt enabled
4	ENCIE	0	R/W	AES encryption/decryption interrupt enable 0: Interrupt disabled 1: Interrupt enabled
3	URX1IE	0	R/W	USART 1 RX interrupt enable 0: Interrupt disabled 1: Interrupt enabled
2	URX0IE	0	R/W	USART0 RX interrupt enable 0: Interrupt disabled 1: Interrupt enabled
1	ADCIE	0	R/W	ADC interrupt enable 0: Interrupt disabled 1: Interrupt enabled
0	RFERRIE	0	R/W	RF TXFIFO/RXFIFO interrupt enable 0: Interrupt disabled 1: Interrupt enabled

表 3.1.7 EIN0 寄存器

IEN1 (0xB8) – Interrupt Enable 1

Bit	Name	Reset	R/W	Description
7:6	–	00	R0	Reserved. Read as 0
5	P0IE	0	R/W	Port 0 interrupt enable 0: Interrupt disabled 1: Interrupt enabled
4	T4IE	0	R/W	Timer 4 interrupt enable 0: Interrupt disabled 1: Interrupt enabled
3	T3IE	0	R/W	Timer 3 interrupt enable 0: Interrupt disabled 1: Interrupt enabled
2	T2IE	0	R/W	Timer 2 interrupt enable 0: Interrupt disabled 1: Interrupt enabled
1	T1IE	0	R/W	Timer 1 interrupt enable 0: Interrupt disabled 1: Interrupt enabled
0	DMAIE	0	R/W	DMA transfer interrupt enable 0: Interrupt disabled 1: Interrupt enabled

表 3.1.8 IEN1 寄存器

IRCON (0xC0) – Interrupt Flags 4

Bit	Name	Reset	R/W	Description
7	STIF	0	R/W	Sleep Timer interrupt flag 0: Interrupt not pending 1: Interrupt pending
6	–	0	R/W	Must be written 0. Writing a 1 always enables the interrupt source.
5	P0IF	0	R/W	Port 0 interrupt flag 0: Interrupt not pending 1: Interrupt pending
4	T4IF	0	R/W H0	Timer 4 interrupt flag. Set to 1 when Timer 4 interrupt occurs and cleared when CPU vectors to the interrupt service routine. 0: Interrupt not pending 1: Interrupt pending
3	T3IF	0	R/W H0	Timer 3 interrupt flag. Set to 1 when Timer 3 interrupt occurs and cleared when CPU vectors to the interrupt service routine. 0: Interrupt not pending 1: Interrupt pending
2	T2IF	0	R/W H0	Timer 2 interrupt flag. Set to 1 when Timer 2 interrupt occurs and cleared when CPU vectors to the interrupt service routine. 0: Interrupt not pending 1: Interrupt pending
1	T1IF	0	R/W H0	Timer 1 interrupt flag. Set to 1 when Timer 1 interrupt occurs and cleared when CPU vectors to the interrupt service routine. 0: Interrupt not pending 1: Interrupt pending
0	DMAIF	0	R/W	DMA-complete interrupt flag 0: Interrupt not pending 1: Interrupt pending

表 3.1.9 IRCON 寄存器

上述寄存器组为与 Timer 1 定时器相关的控制寄存器和中断控制寄存器，其中包括 T1CTL 控制寄存器，用来控制定时器的开关和模式。 T1CCTL0 为 Timer1 通道 0 比较/捕获控制寄存器，T1CC0H 和 T1CC0L 为 Timer1 通道 0 比较/捕获控制值寄存器。IEN0 与 IEN1 两个寄存器分别控制系统中断总开关和 Timer1 定时器中断源开关。

CLKCONCMD (0xC6) – Clock Control Command

Bit	Name	Reset	R/W	Description
7	OSC32K	1	R/W	32 kHz clock-source select. Setting this bit initiates a clock-source change only. CLKCONSTA.OSC32K reflects the current setting. The 16 MHz RCOSC must be selected as system clock when this bit is to be changed. 0: 32 kHz XOSC 1: 32 kHz RCOSC
6	OSC	1	R/W	System clock-source select. Setting this bit initiates a clock-source change only. CLKCONSTA.OSC reflects the current setting. 0: 32 MHz XOSC 1: 16 MHz RCOSC
5:3	TICKSPD[2:0]	001	R/W	Timer ticks output setting. Cannot be higher than system clock setting given by OSC bit setting. 000: 32 MHz 001: 16 MHz 010: 8 MHz 011: 4 MHz 100: 2 MHz 101: 1 MHz 110: 500 kHz 111: 250 kHz Note that CLKCONCMD.TICKSPD can be set to any value, but the effect is limited by the CLKCONCMD.OSC setting; i.e., if CLKCONCMD.OSC = 1 and CLKCONCMD.TICKSPD = 000, CLKCONSTA.TICKSPD reads 001, and the real TICKSPD is 16 MHz.
2:0	CLKSPD	001	R/W	Clock speed. Cannot be higher than system clock setting given by the OSC bit setting. Indicates current system-clock frequency 000: 32 MHz 001: 16 MHz 010: 8 MHz 011: 4 MHz 100: 2 MHz 101: 1 MHz 110: 500 kHz 111: 250 kHz Note that CLKCONCMD.CLKSPD can be set to any value, but the effect is limited by the CLKCONCMD.OSC setting; i.e., if CLKCONCMD.OSC = 1 and CLKCONCMD.CLKSPD = 000, CLKCONSTA.CLKSPD reads 001, and the real CLKSPD is 16 MHz. Note also that the debugger cannot be used with a divided system clock. When running the debugger, the value of CLKCONCMD.CLKSPD should be set to 000 when CLKCONCMD.OSC = 0 or to 001 when CLKCONCMD.OSC = 1.

CLKCONSTA (0x9E) – Clock Control Status

Bit	Name	Reset	R/W	Description
7	OSC32K	1	R	Current 32-kHz clock source selected: 0: 32-kHz XOSC 1: 32-kHz RCOSC
6	OSC	1	R	Current system clock selected: 0: 32-MHz XOSC 1: 16-MHz RCOSC
5:3	TICKSPD[2:0]	001	R	Current timer ticks output setting 000: 32 MHz 001: 16 MHz 010: 8 MHz 011: 4 MHz 100: 2 MHz 101: 1 MHz 110: 500 kHz 111: 250 kHz
2:0	CLKSPD	001	R	Current clock speed 000: 32 MHz 001: 16 MHz 010: 8 MHz 011: 4 MHz 100: 2 MHz 101: 1 MHz 110: 500 kHz 111: 250 kHz

表 3.1.10 CLKCONCMD 和 CLKCONSTA 寄存器

SLEEPCMD (0xBE) – Sleep-Mode Control Command

Bit	Name	Reset	R/W	Description
7	OSC32K_CALDIS	0	R/W	Disable 32-kHz RC oscillator calibration 0: 32-kHz RC oscillator calibration is enabled. 1: 32-kHz RC oscillator calibration is disabled. This setting can be written at any time, but does not take effect before the chip has been running on the 16-MHz high-frequency RC oscillator.
6:3	–	0000	R0	Reserved
2	–	1	R/W	Reserved. Always write as 1
1:0	MODE[1:0]	00	R/W	Power-mode setting 00: Active / Idle mode 01: Power mode 1 (PM1) 10: Power mode 2 (PM2) 11: Power mode 3 (PM3)

SLEEPSTA (0x9D) – Sleep-Mode Control Status

Bit	Name	Reset	R/W	Description
7	OSC32K_CALDIS	0	R	32-kHz RC oscillator calibration status SLEEPSTA.OSC32K_CALDIS shows the current status of disabling of the 32-kHz RC calibration. The bit is not set to the same value as SLEEPCMD.OSC32K_CALDIS before the chip has been run on the 32-kHz RC oscillator.
6:5	–	00	R	Reserved
4:3	RST[1:0]	XX	R	Status bit indicating the cause of the last reset. If there are multiple resets, the register only contains the last event. 00: Power-on reset and brownout detection 01: External reset 10: Watchdog Timer reset 11: Clock loss reset
2:1	–	00	R	Reserved
0	CLK32K	0	R	The 32-kHz clock signal (synchronized to the system clock)

表 3.1.11　SLEEPCMD 和 SLEEPSTA 控制寄存器

PERCFG (0xF1) – Peripheral Control

Bit	Name	Reset	R/W	Description
7	–	0	R0	Reserved
6	T1CFG	0	R/W	Timer 1 I/O location 0: Alternative 1 location 1: Alternative 2 location
5	T3CFG	0	R/W	Timer 3 I/O location 0: Alternative 1 location 1: Alternative 2 location
4	T4CFG	0	R/W	Timer 4 I/O location 0: Alternative 1 location 1: Alternative 2 location
3:2	–	00	R/W	Reserved
1	U1CFG	0	R/W	USART 1 I/O location 0: Alternative 1 location 1: Alternative 2 location
0	U0CFG	0	R/W	USART 0 I/O location 0: Alternative 1 location 1: Alternative 2 location

表 3.1.12　PERCFG 寄存器

U0CSR (0x86) – USART 0 Control and Status

Bit	Name	Reset	R/W	Description
7	MODE	0	R/W	USART mode select 0: SPI mode 1: UART mode
6	RE	0	R/W	UART receiver enable. Note do not enable receive before uart is fully configured. 0: Receiver disabled 1: Receiver enabled
5	SLAVE	0	R/W	SPI master or slave mode select 0: SPI master 1: SPI slave
4	FE	0	R/W0	UART framing error status. This bit is automatically cleared on a read of the U0CSR register or bits in the U0CSR register. 0: No framing error detected 1: Byte received with incorrect stop-bit level
3	ERR	0	R/W0	UART parity error status. This bit is automatically cleared on a read of the U0CSR register or bits in the U0CSR register. 0: No parity error detected 1: Byte received with parity error
2	RX_BYTE	0	R/W0	Receive byte status. UART mode and SPI slave mode. This bit is automatically cleared when reading U0DBUF, clearing this bit by writing 0 to it, effectively discards the data in U0DBUF. 0: No byte received 1: Received byte ready
1	TX_BYTE	0	R/W0	Transmit byte status. UART mode and SPI master mode 0: Byte not transmitted 1: Last byte written to data-buffer register transmitted
0	ACTIVE	0	R	USART transmit/receive active status. In SPI slave mode, this bit equals slave select. 0: USART idle 1: USART busy in transmit or receive mode

表 3.1.13 U0CSR 寄存器

U0GCR (0xC5) – USART 0 Generic Control

Bit	Name	Reset	R/W	Description
7	CPOL	0	R/W	SPI clock polarity 0: Negative clock polarity 1: Positive clock polarity
6	CPHA	0	R/W	SPI clock phase 0: Data is output on *MOSI* when *SCK* goes from CPOL inverted to CPOL, and data input is sampled on *MISO* when *SCK* goes from CPOL to CPOL inverted. 1: Data is output on *MOSI* when *SCK* goes from CPOL to CPOL inverted, and data input is sampled on *MISO* when *SCK* goes from CPOL inverted to CPOL.
5	ORDER	0	R/W	Bit order for transfers 0: LSB first 1: MSB first
4:0	BAUD_E[4:0]	00000	R/W	Baud rate exponent value. BAUD_E along with BAUD_M determines the UART baud rate and the SPI master SCK clock frequency.

表 3.1.14 U0GCR 寄存器

U0DBUF (0xC1) – USART 0 Receive/Transmit Data Buffer

Bit	Name	Reset	R/W	Description
7:0	DATA[7:0]	0x00	R/W	USART receive and transmit data. When writing this register, the data written is written to the internal transmit-data register. When reading this register, the data from the internal read-data register is read.

U0BAUD (0xC2) – USART 0 Baud-Rate Control

Bit	Name	Reset	R/W	Description
7:0	BAUD_M[7:0]	0x00	R/W	Baud-rate mantissa value. BAUD_E along with BAUD_M decides the UART baud rate and the SPI master SCK clock frequency.

表 3.1.15 U0DBUF 和 U0BAUD 寄存器

ADCCON1 (0xB4) – ADC Control 1

Bit	Name	Reset	R/W	Description
7	EOC	0	R/H0	End of conversion. Cleared when ADCH has been read. If a new conversion is completed before the previous data has been read, the EOC bit remains high. 0: Conversion not complete 1: Conversion completed
6	ST	0		Start conversion. Read as 1 until conversion has completed 0: No conversion in progress 1: Start a conversion sequence if ADCCON1.STSEL = 11 and no sequence is running.
5:4	STSEL[1:0]	11	R/W1	Start select. Selects the event that starts a new conversion sequence 00: External trigger on P2.0 pin 01: Full speed. Do not wait for triggers 10: Timer 1 channel 0 compare event 11: ADCCON1.ST = 1
3:2	-	00	R/W	See ADCCON1 (0xB4) – ADC Control 1 description in Section 14.3.
1:0	–	11	R/W	Reserved. Always set to 11

表 3.1.16　ADCCON1 寄存器

ADCCON3 (0xB6) – ADC Control 3

Bit	Name	Reset	R/W	Description
7:6	EREF[1:0]	00	R/W	Selects reference voltage used for the extra conversion 00: Internal reference 01: External reference on AIN7 pin 10: AVDD5 pin 11: External reference on AIN6–AIN7 differential input
5:4	EDIV[1:0]	00	R/W	Sets the decimation rate used for the extra conversion. The decimation rate also determines the resolution and time required to complete the conversion. 00: 64 decimation rate (7 bits ENOB) 01: 128 decimation rate (9 bits ENOB) 10: 256 decimation rate (10 bits ENOB) 11: 512 decimation rate (12 bits ENOB)
3:0	ECH[3:0]	0000	R/W	Single channel select. Selects the channel number of the single conversion that is triggered by writing to ADCCON3. 0000: AIN0 0001: AIN1 0010: AIN2 0011: AIN3 0100: AIN4 0101: AIN5 0110: AIN6 0111: AIN7 1000: AIN0–AIN1 1001: AIN2–AIN3 1010: AIN4–AIN5 1011: AIN6–AIN7 1100: GND 1101: Reserved 1110: Temperature sensor 1111: VDD/3

表 3.1.17 ADCCON3　寄存器

以上图表列举了和 CC2530 处理器 Timer1 定时器操作相关的寄存器，其中包括 CLKCONCMD 控制寄存器，用来控制系统时钟源，SLEEP 寄存器用来控制各种时钟源，SLEEPCMD 和 SLEEPSTA 寄存器用来控制各种时钟源的开关和状态。PERCFG 寄存器为外设功能控制寄存器，用来控制外设功能模式。U0CSR、U0GCR、U0BUF、U0BAUD 等为串口相关寄存器。

3.2 软件设计

```
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 InitT1(void)
{
    T1CCTL0 = 0X44;
    //T1CCTL0 (0xE5)
    //T1 ch0  中断使能
    //比较模式

    T1CC0H = 0x03;
    T1CC0L = 0xe8;
    //0x03e8 = 1000D)

    T1CTL |= 0X02;
    //start count
    //在这里没有分频。
    //使用比较模式  MODE = 10(B)

    IEN1 |= 0X02;
    IEN0 |= 0X80;
    //开 T1 中断
}
void setTimeTemp(char *p)
{
    char tmp;
    tmp = time[0];
    time[0] = (p[2]-'0')*10+(p[3]-'0');
    if((time[0]<0) || (time[0]>23))      { time[0] = tmp;    return; }
    tmp = time[1];
    time[1] = (p[5]-'0')*10+(p[6]-'0');
    if((time[1]<0) || (time[1]>59))      { time[1] = tmp;    return; }
    tmp = time[2];
    time[2] = (p[8]-'0')*10+(p[9]-'0');
    if((time[2]<0) || (time[2]>59))      { time[2] = tmp;    return; }
}

void main(void)
{
```

```
    char uartBuf[20];

    InitUart();     // baudrate:57600
    InitLed();
    InitT1();

    while(1)
    {
        if(printFlag)
        {
            printFlag = 0;
            if(time[2] == 60)
            {
                time[2] = 0;   time[1]++;
                if(time[1] ==60)
                {
                    time[0]++;
                    time[1] = 0;
                    if(time[0] == 24)   time[0] = 0;
                }
            }
            sprintf(uartBuf,"%2.2d:%2.2d:%2.2d\r\n",time[0],time[1],time[2]);
            uartBuf[15]= '\0';
            prints(uartBuf);
        }
    }
}
/*********************************************************************
*函数功能 ： 串口接收一个字符
*入口参数 ： 无
*返 回 值 ： 无
*说      明 ： 接收完成后打开接收
*********************************************************************/
#pragma vector = URX0_VECTOR
 __interrupt void UART0_ISR(void)
{
    URX0IF = 0;                     //清中断标志
    //temp = U0DBUF;
    if( (rFlag == 1) || (U0DBUF == 's') )
    {
        rFlag = 1;
        timeSet[i++] = U0DBUF;
    }
    if(i == 10)
    {
        i = 0;      rFlag = 0;
```

```
            setTimeTemp(timeSet);
        }
}
/*****************************************************************
*函数功能 ：T1 中断函数
*入口参数 ：无
*返 回 值 ：无
*说      明 ：
*****************************************************************/
#pragma vector = T1_VECTOR
  __interrupt void T1_ISR(void)
{
      IRCON &= ~0x02;//清中断标志
      counter++;
      if(counter == 30000)
      {
            counter = 0;
            printFlag = 1;
            time[2]++;

            LED1 = ~LED1;                  // 调试指示用
      }
}
```

程序通过配置 CC2530 处理器的 Timer1 定时器来模拟产生秒信号，进行时间计数，初始时间设定为 00:00:00，也可以通过串口相应的命令格式来设置时间计数，如 s+12+50+30 设置时间为 12 时 50 分 30 秒。

4. 实验步骤

1）使用 ZigBee Debuger USB 仿真器连接 PC 机和 ZigBee(CC2530)模块，打开 ZigBee 模块开关供电。将系统配套串口线一端连接 PC 机，一端连接 ZigBee 调试板的串口上。

2）启动 IAR 开发环境，新建工程，或直接使用 Exp11 实验工程。

3）在 IAR 开发环境中编译、运行、调试程序。

4）使用 PC 机自带的超级终端连接串口，将超级终端设置为串口波特率 57600、8 位、无奇偶校验位、无硬件流模式，运行程序，即可看到模拟时间的计数。当向串口输入相应格式的数据时，即可设置时间：s+12+50+30 设置时间为 12 时 50 分 30 秒。

实验十三. 看门狗实验

1. 实验环境

- 硬件：ZigBee(CC2530)模块，ZigBee 下载调试板，USB 仿真器，PC 机。
- 软件：IAR Embedded Workbench for MCS-51

2. 实验内容

- 阅读 ZigBee2530 开发套件 ZigBee 模块硬件部分文档，熟悉 ZigBee 模块硬件接口。
- 使用 IAR 开发环境设计程序，利用 CC2530 的 看门狗定时器来实现对系统复位状态的控制。

3. 实验原理

3.1 硬件接口原理

T1CC0H (0xDB) – Timer 1 Channel 0 Capture/Compare Value, High

Bit	Name	Reset	R/W	Description
7:0	T1CC0[15:8]	0x00	R/W	Timer 1 channel 0 capture/compare value high order byte. Writing to this register when T1CCTL0.MODE = 1 (compare mode) causes the T1CC0[15:0] update to the written value to be delayed until T1CNT = 0x0000.

T1CC0L (0xDA) – Timer 1 Channel 0 Capture/Compare Value, Low

Bit	Name	Reset	R/W	Description
7:0	T1CC0[7:0]	0x00	R/W	Timer 1 channel 0 capture/compare value low order byte. Data written to this register is stored in a buffer but not written to T1CC0[7:0] until, and at the same time as, a later write to T1CC0H takes effect.

表 3.1.4 T1CC0H 和 T1CC0L 寄存器

CLKCONCMD (0xC6) – Clock Control Command

Bit	Name	Reset	R/W	Description
7	OSC32K	1	R/W	32 kHz clock-source select. Setting this bit initiates a clock-source change only. CLKCONSTA.OSC32K reflects the current setting. The 16 MHz RCOSC must be selected as system clock when this bit is to be changed. 0: 32 kHz XOSC 1: 32 kHz RCOSC
6	OSC	1	R/W	System clock-source select. Setting this bit initiates a clock-source change only. CLKCONSTA.OSC reflects the current setting. 0: 32 MHz XOSC 1: 16 MHz RCOSC
5:3	TICKSPD[2:0]	001	R/W	Timer ticks output setting. Cannot be higher than system clock setting given by OSC bit setting. 000: 32 MHz 001: 16 MHz 010: 8 MHz 011: 4 MHz 100: 2 MHz 101: 1 MHz 110: 500 kHz 111: 250 kHz Note that CLKCONCMD.TICKSPD can be set to any value, but the effect is limited by the CLKCONCMD.OSC setting; i.e., if CLKCONCMD.OSC = 1 and CLKCONCMD.TICKSPD = 000, CLKCONSTA.TICKSPD reads 001, and the real TICKSPD is 16 MHz.
2:0	CLKSPD	001	R/W	Clock speed. Cannot be higher than system clock setting given by the OSC bit setting. Indicates current system-clock frequency 000: 32 MHz 001: 16 MHz 010: 8 MHz 011: 4 MHz 100: 2 MHz 101: 1 MHz 110: 500 kHz 111: 250 kHz Note that CLKCONCMD.CLKSPD can be set to any value, but the effect is limited by the CLKCONCMD.OSC setting; i.e., if CLKCONCMD.OSC = 1 and CLKCONCMD.CLKSPD = 000, CLKCONSTA.CLKSPD reads 001, and the real CLKSPD is 16 MHz. Note also that the debugger cannot be used with a divided system clock. When running the debugger, the value of CLKCONCMD.CLKSPD should be set to 000 when CLKCONCMD.OSC = 0 or to 001 when CLKCONCMD.OSC = 1.

CLKCONSTA (0x9E) – Clock Control Status

Bit	Name	Reset	R/W	Description
7	OSC32K	1	R	Current 32-kHz clock source selected: 0: 32-kHz XOSC 1: 32-kHz RCOSC
6	OSC	1	R	Current system clock selected: 0: 32-MHz XOSC 1: 16-MHz RCOSC
5:3	TICKSPD[2:0]	001	R	Current timer ticks outputs setting 000: 32 MHz 001: 16 MHz 010: 8 MHz 011: 4 MHz 100: 2 MHz 101: 1 MHz 110: 500 kHz 111: 250 kHz
2:0	CLKSPD	001	R	Current clock speed 000: 32 MHz 001: 16 MHz 010: 8 MHz 011: 4 MHz 100: 2 MHz 101: 1 MHz 110: 500 kHz 111: 250 kHz

表 3.1.5 CLKCONCMD 和 CLKCONSTA 寄存器

WDCTL (0xC9) – Watchdog Timer Control

Bit	Name	Reset	R/W	Description
7:4	CLR[3:0]	0000	R0/W	Clear timer. In Watchdog mode, when 0xA followed by 0x5 is written to these bits, the timer is cleared (i.e. loaded with 0). Note that the timer is only cleared when 0x5 is written within one watchdog clock period after 0xA was written. Writing these bits when the Watchdog Timer is IDLE has no effect. When operating in timer mode, the timer can be cleared to 0x0000 (but not stopped) by writing 1 to CLR[0] (the other 3 bits are don't care).
3:2	MODE[1:0]	00	R/W	Mode select. These bits are used to start the WDT in Watchdog mode or Timer mode. Setting these bits to IDLE stops the timer when in Timer mode. Note: to switch to Watchdog mode when operating in Timer mode, first stop the WDT - then start the WDT in Watchdog mode. When operating in Watchdog mode, writing these bits has no effect. 00: IDLE 01: IDLE (unused, equivalent to 00 setting) 10: Watchdog mode 11: Timer mode
1:0	INT[1:0]	00	R/W	Timer interval select. These bits select the timer interval defined as a given number of 32 kHz oscillator periods. Note that the interval can only be changed when the WDT is IDLE, so the interval must be set at the same time as the timer is started. 00: Clock period × 32,768 (~1 s) when running the 32 kHz XOSC 01: Clock period × 8192 (~0.25 s) 10: Clock period × 512 (~15.625 ms) 11: Clock period × 64 (~1.9 ms) When clock division is enabled through CLKCONCMD.CLKSPD the length of the watchdog timer interval is reduced by a factor equal to the current oscillator clock frequency divided by the set clock speed. E.g. if 32-MHZ crystal is selected and clock speed is set to 4 MHz then the watchdog timeout is reduced by a factor 32-MHZ/4-MHz = 8. If the watchdog interval set by WDCTL.INT was 1 s nominally it is 1/8 s with this clock division factor.

表 3.1.6　WDCTL 寄存器

以上图表列举了和 C2530 处理器看门狗定时器操作相关的寄存器，其中 WDCTL 控制寄存器用来控制看门狗定时器的工作模式及复位状态。

3.2 软件设计

```
void Init_IO(void)
{
  P1DIR = 0x03;

  led1 = 1;
  led2 = 1;
}

void Init_Watchdog(void)
{
  WDCTL = 0x00;
  //时间间隔一秒，看门狗模式
  WDCTL |= 0x08;
  //启动看门狗
}

void Init_Clock(void)
{
  CLKCONCMD = 0X00;
}

void FeetDog(void)          //喂狗
```

```
{
  WDCTL = 0xa0;
  WDCTL = 0x50;
}

void Delay(void)
{
  uint n;

  for(n=50000;n>0;n--);
  for(n=50000;n>0;n--);
  for(n=50000;n>0;n--);
  for(n=50000;n>0;n--);
  for(n=50000;n>0;n--);
  for(n=50000;n>0;n--);
  for(n=50000;n>0;n--);
}

void main(void)
{
  Init_Clock();
  Init_IO();
  Init_Watchdog();

  led1=0;
  Delay();
  led2=0;

  while(1)
  {
      FeetDog();
  }   //喂狗指令（加入后系统不复位，小灯不闪烁）

}
```

程序通过配置 CC2530 处理器的看门狗定时器来产生复位信号，如果在复位周期内喂狗，既可避免看门狗强制复位系统，软件用 LED 灯来实现系统复位的监测。

4. 实验步骤

1）使用 ZigBee Debuger USB 仿真器连接 PC 机和 ZigBee(CC2530)模块，打开 ZigBee 模块开关供电。将系统配套串口线一端连接 PC 机，一端连接 ZigBee 调试板的串口上。

2）启动 IAR 开发环境，新建工程，或直接使用 Exp12 实验工程。

3）在 IAR 开发环境中编译、运行、调试程序。

实验十四. 系统休眠与低功耗实验

1. 实验环境

- 硬件：ZigBee(CC2530)模块，ZigBee 下载调试板，USB 仿真器，PC 机。
- 软件：IAR Embedded Workbench for MCS-51。

2. 实验内容

- 阅读 ZigBee2530 开发套件 ZigBee 模块硬件部分文档，熟悉 ZigBee 模块硬件接口。
- 使用 IAR 开发环境设计程序，利用 CC2530 的电源管理控制寄存器控制系统工作状态。

3. 实验原理

3.1 硬件接口原理

PCON (0x87) – Power Mode Control

Bit	Name	Reset	R/W	Description
7:1	–	0000 000	R/W	Reserved, always write as 0000 000.
0	IDLE	0	R0/W H0	Power mode control. Writing 1 to this bit forces the device to enter the power mode set by SLEEPCMD.MODE (note that MODE = 0x00 AND IDLE = 1 stops the CPU core activity). This bit is always read as 0. All enabled interrupts clear this bit when active, and the device re-enters active mode.

表 3.1.4 PCON 寄存器

SLEEPCMD (0xBE) – Sleep-Mode Control Command

Bit	Name	Reset	R/W	Description
7	OSC32K_CALDIS	0	R/W	Disable 32-kHz RC oscillator calibration 0: 32-kHz RC oscillator calibration is enabled. 1: 32-kHz RC oscillator calibration is disabled. This setting can be written at any time, but does not take effect before the chip has been running on the 16-MHz high-frequency RC oscillator.
6:3	–	0000	R0	Reserved
2	–	1	R/W	Reserved. Always write as 1
1:0	MODE[1:0]	00	R/W	Power-mode setting 00: Active / Idle mode 01: Power mode 1 (PM1) 10: Power mode 2 (PM2) 11: Power mode 3 (PM3)

SLEEPSTA (0x9D) – Sleep-Mode Control Status

Bit	Name	Reset	R/W	Description
7	OSC32K_CALDIS	0	R	32-kHz RC oscillator calibration status SLEEPSTA.OSC32K_CALDIS shows the current status of disabling of the 32-kHz RC calibration. The bit is not set to the same value as SLEEPCMD.OSC32K_CALDIS before the chip has been run on the 32-kHz RC oscillator.
6:5	–	00	R	Reserved
4:3	RST[1:0]	XX	R	Status bit indicating the cause of the last reset. If there are multiple resets, the register only contains the last event. 00: Power-on reset and brownout detection 01: External reset 10: Watchdog Timer reset 11: Clock loss reset
2:1	–	00	R	Reserved
0	CLK32K	0	R	The 32-kHz clock signal (synchronized to the system clock)

表 3.1.5　SLEEPCMD 和 SLEEPSTA 控制寄存器

IEN0 (0xA8) – Interrupt Enable 0

Bit	Name	Reset	R/W	Description
7	EA	0	R/W	Disables all interrupts. 0: No interrupt is acknowledged. 1: Each interrupt source is individually enabled or disabled by setting its corresponding enable bit.
6	–	0	R0	Reserved. Read as 0
5	STIE	0	R/W	Sleep Timer interrupt enable 0: Interrupt disabled 1: Interrupt enabled
4	ENCIE	0	R/W	AES encryption/decryption interrupt enable 0: Interrupt disabled 1: Interrupt enabled
3	URX1IE	0	R/W	USART 1 RX interrupt enable 0: Interrupt disabled 1: Interrupt enabled
2	URX0IE	0	R/W	USART0 RX interrupt enable 0: Interrupt disabled 1: Interrupt enabled
1	ADCIE	0	R/W	ADC interrupt enable 0: Interrupt disabled 1: Interrupt enabled
0	RFERRIE	0	R/W	RF TXFIFO/RXFIFO interrupt enable 0: Interrupt disabled 1: Interrupt enabled

表 3.1.6　IEN0 寄存器

IEN2 (0x9A) – Interrupt Enable 2

Bit	Name	Reset	R/W	Description
7:6	–	00	R0	Reserved. Read as 0
5	WDTIE	0	R/W	Watchdog Timer interrupt enable 0: Interrupt disabled 1: Interrupt enabled
4	P1IE	0	R/W	Port 1 interrupt enable 0: Interrupt disabled 1: Interrupt enabled
3	UTX1IE	0	R/W	USART 1 TX interrupt enable 0: Interrupt disabled 1: Interrupt enabled
2	UTX0IE	0	R/W	USART 0 TX interrupt enable 0: Interrupt disabled 1: Interrupt enabled
1	P2IE	0	R/W	Port 2 and USB interrupt enable 0: Interrupt disabled 1: Interrupt enabled
0	RFIE	0	R/W	RF general interrupt enable 0: Interrupt disabled 1: Interrupt enabled

表 3.1.7　IEN2 寄存器

以上图表列举了和 CC2530 处理器 低功耗相关的寄存器，其中包括 PCON 电源模式控制寄存器，SLEEPCMD 和 SLEEPSTA 寄存器用来控制各种时钟源的开关和状态，IEN0 和 IEN2 两个寄存器分别控制系统中断总开关和 PORT1 中断源开关。

3.2 软件设计

```
/*************************************************************
*函数功能：延时
*入口参数：无
*返回值　：无
*说　明　：可在宏定义中改变延时长度
*************************************************************/
void Delay(void)
{
  uint tt;
  for(tt = 0;tt<DELAY;tt++);
  for(tt = 0;tt<DELAY;tt++);
  for(tt = 0;tt<DELAY;tt++);
  for(tt = 0;tt<DELAY;tt++);
  for(tt = 0;tt<DELAY;tt++);
}

/*************************************************************
*函数功能：初始化电源
*入口参数：para1,para2,para3,para4
*返回值　：无
*说　明　：para1,模式选择
*                                                          *
```

```
* para1    0        1     2     3                                                    *
* mode       PM0PM1PM2PM3                                    *
*                                                                                   *
*************************************************************************/
void PowerMode(uchar sel)
{
   uchar i,j;
   i = sel;
   if(sel<4)
   {
        SLEEPCMD &= 0xfc;
        SLEEPCMD |= i;
        for(j=0;j<4;j++);
        PCON = 0x01;
   }
   else
   {
        PCON = 0x00;
   }
}

/*************************************************************
* 函数功能：初始化 I/O,控制 LED
* 入口参数：无
* 返回值  ：无
* 说  明  ：初始化完成后关灯
*************************************************************/
void Init_IO_AND_LED(void)
{
     P1DIR = 0X03;
     RLED = 1;
     YLED = 1;

     EA = 1;
     IEN2 |= 0X10;   //P1IE = 1;

}
/*************************************************************
* 函数功能：主函数
* 入口参数：
* 返回值  ：无
* 说  明  ：10 次绿色 LED 闪烁后进入睡眠状态
*************************************************************/
void main()
{
   uchar count = 0;
```

```
    Init_IO_AND_LED();

    RLED = 0 ;          //开红色 LED，系统工作指示
    Delay();            //延时
    Delay();
    Delay();
    Delay();

    while(1)
    {
        YLED = !YLED;
                    RLED = 0;
        count++;
        if(count >= 20)
                    {
                      count = 0;
                      RLED = 1;
                      PowerMode(3);
          //10 次闪烁后进入睡眠状态
                    }

        //Delay();
        Delay();
                    //延时函数无形参，只能通过改变系统时钟频率
                    //来改变小灯的闪烁频率
    };
}
```

程序通过配置 CC2530 处理器的电源管理相关寄存器，从而让系统在 LED 闪烁 10 次后进入休眠状态。

4. 实验步骤

1）使用 ZigBee Debuger USB 仿真器连接 PC 机和 ZigBee(CC2530)模块，打开 ZigBee 模块开关供电。将系统配套串口线一端连接 PC 机，一端连接 ZigBee 调试板的串口上。

2）启动 IAR 开发环境，新建工程，或直接使用 Exp13 实验工程。

3）在 IAR 开发环境中编译、运行、调试程序。

4）系统在 LED 闪烁 10 次后进入休眠状态。

实验十五. 按键实验

1. 实验环境

- 硬件：ZigBee(CC2530)模块，ZigBee 下载调试板，USB 仿真器，PC 机。
- 软件：IAR Embedded Workbench for MCS-51

2. 实验内容

- 阅读 ZigBee2530 开发套件 ZigBee 模块硬件部分文档，熟悉 ZigBee 模块按键接口。
- 使用 IAR 开发环境设计程序，利用 CC2530 的电源管理控制寄存器控制系统工作状态。

3. 实验原理

3.1 硬件接口原理

- **按键接口，如图 3.1.1 所示。**

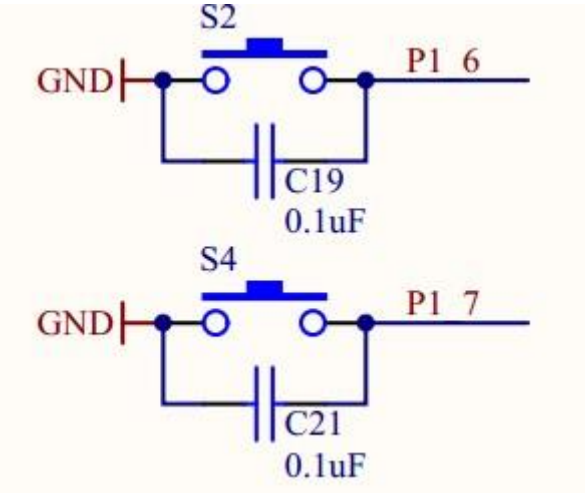


图 3.1.1

CC2530 开发板有三个按键，一个复位按键。其余两个按键可以通过编程进行控制。当按键按下时，相应的管脚输出低电平。在此我们采用下降沿触发中断的方式来检测是否有按键按下。

- **CC2530 相关寄存器**

P1 (0x90) – Port 1

Bit	Name	Reset	R/W	Description
7:0	P1[7:0]	0xFF	R/W	Port 1. General-purpose I/O port. Bit-addressable from SFR. This CPU-internal register is readable, but not writable, from XDATA (0x7090).

图 3.1.2 P1 寄存器

P1SEL (0xF4) – Port 1-Function Select

Bit	Name	Reset	R/W	Description
7:0	SELP1_[7:0]	0x00	R/W	P1.7 to P1.0 function select 0: General-purpose I/O 1: Peripheral function

图 3.1.3 P1SEL 寄存器

P1DIR (0xFE) – Port 1 Direction

Bit	Name	Reset	R/W	Description
7:0	DIRP1_[7:0]	0x00	R/W	P1.7 to P1.0 I/O direction 0: Input 1: Output

图 3.1.4 P1DIR 寄存器

P1INP (0xF6) – Port 1 Input Mode

Bit	Name	Reset	R/W	Description
7:2	MDP1_[7:2]	0000 00	R/W	P1.7 to P1.2 I/O input mode 0: Pullup/pulldown [see P2INP (0xF7) – Port 2 input mode] 1: 3-state
1:0	–	00	R0	Reserved

图 3.1.5 P1INP 寄存器

P2INP (0xF7) – Port 2 Input Mode

Bit	Name	Reset	R/W	Description
7	PDUP2	0	R/W	Port 2 pullup/pulldown select. Selects function for all Port 2 pins configured as pullup/pulldown inputs. 0: Pullup 1: Pulldown
6	PDUP1	0	R/W	Port 1 pullup/down select. Selects function for all Port 1 pins configured as pullup/pulldown inputs. 0: Pullup 1: Pulldown
5	PDUP0	0	R/W	Port 0 pullup/pulldown select. Selects function for all Port 0 pins configured as pullup/pulldown inputs. 0: Pullup 1: Pulldown
4:0	MDP2_[4:0]	0 0000	R/W	P2.4 to P2.0 I/O input mode 0: Pullup/pulldown 1: 3-state

图 3.1.6 P2INP 寄存器

PICTL (0x8C) – Port Interrupt Control

Bit	Name	Reset	R/W	Description
7	PADSC	0	R/W	Drive strength control for I/O pins in output mode. Selects output drive strength enhancement to account for low I/O supply voltage on pin DVDD (this to ensure the same drive strength at lower voltages as at higher). 0: Minimum drive strength enhancement. DVDD1/2 equal to or greater than 2.6 V 1: Maximum drive strength enhancement. DVDD1/2 less than 2.6 V
6:4	–	000	R0	Reserved
3	P2ICON	0	R/W	Port 2, inputs 4 to 0 interrupt configuration. This bit selects the interrupt request condition for Port 2 inputs 4 to 0. 0: Rising edge on input gives interrupt. 1: Falling edge on input gives interrupt.
2	P1ICONH	0	R/W	Port 1, inputs 7 to 4 interrupt configuration. This bit selects the interrupt request condition for the high nibble of Port 1 inputs. 0: Rising edge on input gives interrupt. 1: Falling edge on input gives interrupt
1	P1ICONL	0	R/W	Port 1, inputs 3 to 0 interrupt configuration. This bit selects the interrupt request condition for the low nibble of Port 1 inputs. 0: Rising edge on input gives interrupt. 1: Falling edge on input gives interrupt.
0	P0ICON	0	R/W	Port 0, inputs 7 to 0 interrupt configuration. This bit selects the interrupt request condition for all Port 0 inputs. 0: Rising edge on input gives interrupt. 1: Falling edge on input gives interrupt.

图 3.1.7 PICTL 寄存器

P1IEN (0x8D) – Port 1 Interrupt Mask

Bit	Name	Reset	R/W	Description
7:0	P1_[7:0]IEN	0x00	R/W	Port P1.7 to P1.0 interrupt enable 0: Interrupts are disabled. 1: Interrupts are enabled.

图 3.1.8 P1IEN 寄存器

IEN2 (0x9A) – Interrupt Enable 2

Bit	Name	Reset	R/W	Description
7:6	–	00	R0	Reserved. Read as 0
5	WDTIE	0	R/W	Watchdog Timer interrupt enable 0: Interrupt disabled 1: Interrupt enabled
4	P1IE	0	R/W	Port 1 interrupt enable 0: Interrupt disabled 1: Interrupt enabled
3	UTX1IE	0	R/W	USART 1 TX interrupt enable 0: Interrupt disabled 1: Interrupt enabled
2	UTX0IE	0	R/W	USART 0 TX interrupt enable 0: Interrupt disabled 1: Interrupt enabled
1	P2IE	0	R/W	Port 2 and USB interrupt enable 0: Interrupt disabled 1: Interrupt enabled
0	RFIE	0	R/W	RF general interrupt enable 0: Interrupt disabled 1: Interrupt enabled

图 3.1.9 IEN2 寄存器

3.2 软件设计

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

```
 * 函数名称：EINT_ISR
 * 功　　能：外部中断服务函数
 * 入口参数：无
 * 出口参数：无
 * 返 回 值：无
 ******************************************************************************/
#pragma vector=P1INT_VECTOR
__interrupt void EINT_ISR(void)
{
  EA = 0;                 // 关闭全局中断

  /* 若是 P2.0 产生的中断 */
  if(P1IFG & 0x40)
  {
    /* 切换 LED1(绿色)的亮灭状态 */
    if(P1_0 == 0)       // 若之前是控制 LED1(绿色)点亮，则现在熄灭 LED1
    {
      P1_0 = 1;
    }
    else                  // 若之前是控制 LED1(绿色)熄灭，则现在点亮 LED1
    {
      P1_0 = 0;
    }

    /* 切换 LED2(红色)的亮灭状态 */
    if(P1_1 == 0)       // 若之前是控制 LED2(红色)点亮，则现在熄灭 LED2
    {
      P1_1 = 1;
    }
    else                  // 若之前是控制 LED2(红色)熄灭，则现在点亮 LED2
    {
      P1_1 = 0;
    }

    /* 切换 LED3(黄色)的亮灭状态 */

    /* 等待用户释放按键，并消抖 */
    while(P1_6 & 0x40);
    delay(10);
    while(P1_6 & 0x40);

    /* 清除中断标志 */
    P1IFG &= ~0x40;     // 清除 P1.6 中断标志
    IRCON2 &= ~0x08;    // 清除 P1 口中断标志
  }
    if(P1IFG & 0x80)
```

```
  {
    /* 切换 LED1(绿色)的亮灭状态 */
    if(P1_0 == 0)      // 若之前是控制 LED1(绿色)点亮，则现在熄灭 LED1
    {
      P1_0 = 1;
    }
    else               // 若之前是控制 LED1(绿色)熄灭，则现在点亮 LED1
    {
      P1_0 = 0;
    }

    /* 切换 LED2(红色)的亮灭状态 */
    if(P1_1 == 0)      // 若之前是控制 LED2(红色)点亮，则现在熄灭 LED2
    {
      P1_1 = 1;
    }
    else               // 若之前是控制 LED2(红色)熄灭，则现在点亮 LED2
    {
      P1_1 = 0;
    }

    /* 切换 LED3(黄色)的亮灭状态 */

    /* 等待用户释放按键，并消抖 */
    while(P1_7 & 0x80);
    delay(10);
    while(P1_7& 0x80);

    /* 清除中断标志 */
    P1IFG &= ~0x80;    // 清除 P1.7 中断标志
    IRCON2 &= ~0x08;   // 清除 P1 口中断标志
  }
  EA = 1;              // 使能全局中断
}


/*************************************************************************
 * 函数名称：main
 * 功      能：main 函数入口
 * 入口参数：无
 * 出口参数：无
 * 返 回 值：无
 *************************************************************************/
void main(void)
{
  /*
```

```
        由于 CC253x 系列片上系统上电复位后，所有 21 个数字 I/O 均默认为具有
        上拉的通用输入 I/O，因此本实验只需要改变作为 LED 控制信号的 P1.0 和 P1.1
        和 P1.4 方向为输出即可。另外还需要将 P2.0 设置为输入下拉模式。
        在用户的实际应用开发中，我们建议用户采用如下步骤来配置数字 I/O：
            1. 设置数字 I/O 为通用 I/O
            2. 设置通用 I/O 的方向
            3. 若通用 I/O 的方向被配置为输入，可配置上拉/下拉/三态模式,在此实验中不需要配置
            4. 若通用 I/O 的方向被配置为输出，可设置其输出高/低电平
     */
    P1SEL=0;
    /* 配置 P1.0、P1.1 和 P1.4 的方向为输出 */
    P1DIR |= 0x03;   // 0x13 = 0B00010011
    P1_0 = 1;   // P1.0 输出低电平熄灭其所控制的 LED1(绿色)
    P1_1 = 1;   // P1.1 输出低电平熄灭其所控制的 LED2(红色)
    /* 配置 P1 口的中断边沿下降沿产生中断 */
    PICTL |= 0x02;
    /* 使能 P1.6 P1.7 中断 */
    P1IEN |= 0xC0;
    /* 使能 P1 口中断 */
    IEN2 |= 0x10;
    /* 使能全局中断 */
    EA = 1;
    while(1);
}
```

4. 实验步骤

1）启动 IAR 开发环境，新建工程，或直接使用 Exp14 实验工程。

2）在 IAR 开发环境中编译、运行、下载程序。

3）通过两个按键来控制两个 LED 的亮灭。

第九章. 无线通讯模块之 ZigBee 通信实验

本章主要介绍无线通讯模块部分的 ZigBee（TI）通信的实验内容，采用 ICS-IOT-CEP 平台配套的 ZigBee 模块（TI）。内容由浅入深，前面已经讲述相应模块的硬件接口实验，本章主要针对网络协议栈实验及传感器网络实验等。通过本章实验内容，读者即可以迅速掌握基上述 ZigBee 无线模块的开发方法，以及相应网络结构的传感器数据通讯应用设计。

实验一. 点对点无线通讯实验

1. 实验环境

- ZigBee(CC2530)模块(2 个)，ZigBee 下载调试板，USB 仿真器，PC 机。
- 软件： IAR Embedded Workbench for MCS-51

2. 实验内容

- 了解 CC2530 芯片点对点通讯操作过程，熟悉该模块射频软件接口配置。
- 使用 IAE 开发环境设计程序，利用 2 个 CC2530 ZigBee 模块实现点对点无线通讯。

3. 实验原理

3.1 ZigBee(CC2530)模块 LED 硬件接口

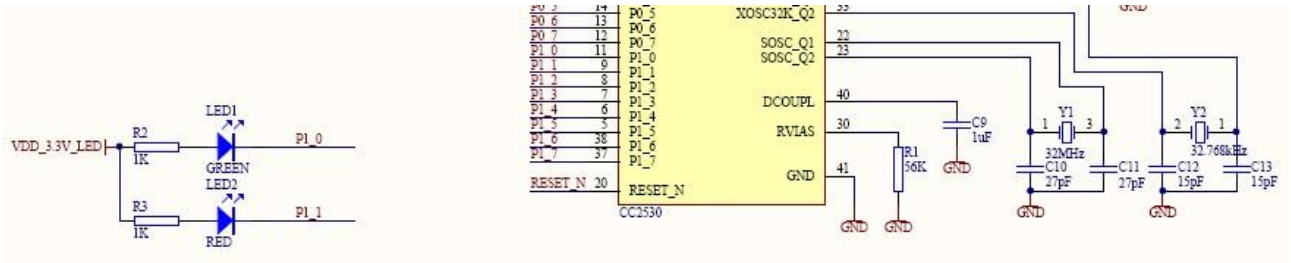


图 3.1.1 LED 硬件接口

ZigBee(CC2530)模块硬件上设计有 2 个 LED 灯，用来编程调试使用。分别连接 CC2530 的 P1_0、P1_1 两个 IO 引脚。从原理图上可以看出，2 个 LED 灯共阳极，当 P1_0、P1_1 引脚为低电平时候，LED 灯点亮。

3.2 软件流程图

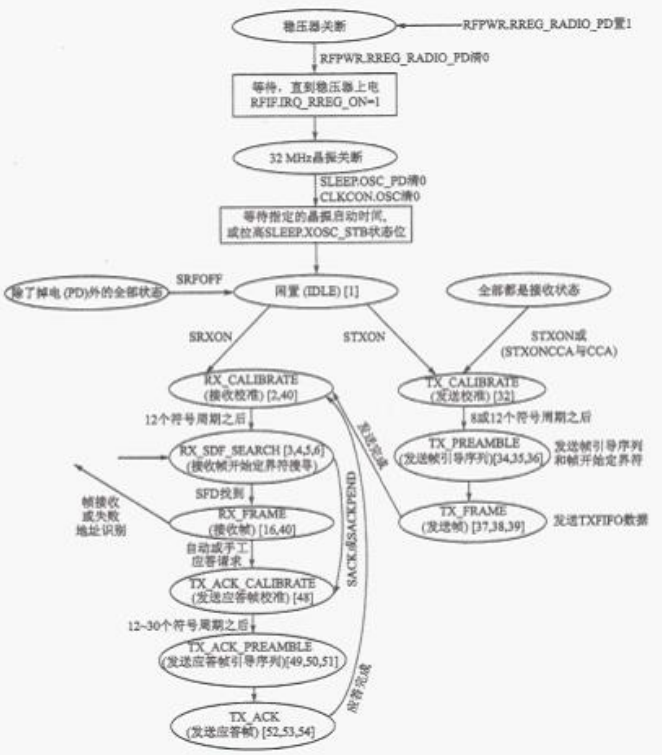


图 3.2.1 P2P 软件路程图

3.3 关键函数分析

1）射频初始化函数

uint8 halRfInit(void)

功能描述：zigbee 通信设置，自动应答有效，设置输出功率 0dbm，Rx 设置，接收中断有效。

参数描述：无

返回：配置成功返回 SUCCESS

2）发送数据包函数

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

功能描述：发送包函数。

入口参数：destAddr　　目标网络短地址

pPayload　　发送数据包头指针，

length　　包的大小

出口参数：无

返 回 值：成功返回 SUCCESS，失败返回 FAILED

3）接收数据函数

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

功能描述： 从接收缓存中拷贝出最近接收到的包。

参数：接收数据包头指针

接收包的大小

返回：实际接收的数据字节数

源码实现：

per_test.c

```
void main (void)
{
    uint8 i;
    appState = IDLE;                // 初始化应用状态为空闲
    appStarted = FALSE;             // 初始化启动标志位 FALSE
    /* 初始化 Basic RF */
    basicRfConfig.panId = PAN_ID;           // 初始化个域网 ID
    basicRfConfig.ackRequest = FALSE;       // 不需要确认
      halBoardInit();
      if(halRfInit()==FAILED)         //初始化 hal_rf
      HAL_ASSERT(FALSE);
          /* 快速闪烁 8 次 led1,led2 */
    for(i = 0; i < 16; i++)
    {
      halLedToggle(1);   // 切换 led1 的亮灭状态
      halLedToggle(2);   // 切换 led2 的亮灭状态
      halMcuWaitMs(50); // 延时大约 50ms
    }

    halLedSet(1);           // led1 指示灯亮，指示设备已上电运行
    halLedClear(2);

    basicRfConfig.channel = 0x0B;           // 设置信道

#ifdef MODE_SEND
    appTransmitter();       // 发送器模式
#else
    appReceiver();          // 接收器模式
#endif

    HAL_ASSERT(FALSE);
}
```

通过上面的代码分析可知，程序通过宏 MODE_SEND 来确定是发送器还是接收器，

appTransmiter()是发送器的主要功能函数，appReceiver()是接收器的主要功能函数，这两个函数最终都会进入一个无限循环状态。

```
static void appTransmitter()
{
  uint32 burstSize=0;
  uint32 pktsSent=0;
  uint8 appTxPower;
  uint8 n;
  /* 初始化 Basic RF */
  basicRfConfig.myAddr = TX_ADDR;
  if(basicRfInit(&basicRfConfig)==FAILED)
  {
    HAL_ASSERT(FALSE);
  }
  /* 设置输出功率 */
  //appTxPower = appSelectOutputPower();
  halRfSetTxPower(2);//HAL_RF_TXPOWER_4_DBM
//  halRfSetTxPower(appTxPower);

  /* 设置进行一次测试所发送的数据包数量 */
  //burstSize = appSelectBurstSize();
  burstSize = 100000;
  /* Basic RF 在发送数据包前关闭接收器，在发送完一个数据包后打开接收器 */
  basicRfReceiveOff();
  /* 配置定时器和 IO */
  //n= appSelectRate();
  appConfigTimer(0xC8);
  //halJoystickInit();

  /* 初始化数据包载荷 */
  txPacket.seqNumber = 0;
  for(n = 0; n < sizeof(txPacket.padding); n++)
  {
    txPacket.padding[n] = n;
  }
  /* 主循环 */
  while (TRUE)
  {
      if (pktsSent < burstSize)
      {
          UINT32_HTON(txPacket.seqNumber);  // 改变发送序号的字节顺序
          basicRfSendPacket(RX_ADDR, (uint8*)&txPacket, PACKET_SIZE);

          /* 在增加序号前将字节顺序改回为主机顺序 */
          UINT32_NTOH(txPacket.seqNumber);
          txPacket.seqNumber++;
```

```
            pktsSent++;
            appState = IDLE;
            halLedToggle(1);     //切换 LED1 的亮灭状态
            halLedToggle(2);     //切换 LED2 的亮灭状态
            halMcuWaitMs(1000);
        }

      /* 复位统计和序号 */
      pktsSent = 0;
    }
}
```

在发送主功能函数里面，通过 basicRfSendPacket();发送接口函数不停向外发送数据，并改变 LED1，LED2 的状态。

```
static void appReceiver()
{
   uint32 segNumber=0;                                    // 数据包序列号
   int16 perRssiBuf[RSSI_AVG_WINDOW_SIZE] = {0};      // 存储 RSSI 的环形缓冲区
   uint8 perRssiBufCounter = 0;                          // 计数器用于 RSSI 缓冲区统计
   perRxStats_t rxStats = {0,0,0,0};                    // 接收状态
   int16 rssi;
   uint8 resetStats=FALSE;
   int16 MyDate[10];                  //串口数据串数字
    initUART();        // 初始化串口
#ifdef INCLUDE_PA
   uint8 gain;
   // 选择增益 (仅 SK - CC2590/91 模块有效)
   gain =appSelectGain();
   halRfSetGain(gain);
#endif
        /* 初始化 Basic RF */
   basicRfConfig.myAddr = RX_ADDR;
   if(basicRfInit(&basicRfConfig)==FAILED)
   {
     HAL_ASSERT(FALSE);
   }
   basicRfReceiveOn();
   /* 主循环 */
   while (TRUE)
   {
     while(!basicRfPacketIsReady());   // 等待新的数据包
     if(basicRfReceive((uint8*)&rxPacket, MAX_PAYLOAD_LENGTH, &rssi)>0)
     {
       halLedSet(1);   // 点亮 LED1
       //halLedSet(2);   // 点亮 LED2
```

```
        UINT32_NTOH(rxPacket.seqNumber);   // 改变接收序号的字节顺序
        segNumber = rxPacket.seqNumber;
        /* 如果统计被复位，设置期望收到的数据包序号为已经收到的数据包序号 */
        if(resetStats)
        {
           rxStats.expectedSeqNum = segNumber;
                    resetStats=FALSE;
        }
        rxStats.rssiSum -= perRssiBuf[perRssiBufCounter];   // 从 sum 中减去旧的 RSSI 值
        perRssiBuf[perRssiBufCounter] =   rssi;   // 存储新的 RSSI 值到环形缓冲区，之后它将被加入
sum
        rxStats.rssiSum += perRssiBuf[perRssiBufCounter];   // 增加新的 RSSI 值到 sum
          MyDate[4] = rssi;                    ////
          MyDate[3] = rxStats.rssiSum;////
        if(++perRssiBufCounter == RSSI_AVG_WINDOW_SIZE)
        {
           perRssiBufCounter = 0;
        }
        /* 检查接收到的数据包是否是所期望收到的数据包 */
        if(rxStats.expectedSeqNum == segNumber)   // 是所期望收到的数据包
        {
           MyDate[0] = rxStats.expectedSeqNum;////
           rxStats.expectedSeqNum++;
        }
        else if(rxStats.expectedSeqNum < segNumber)   // 不是所期望收到的数据包（收到的数据包的序
号大于期望收到的数据包的序号）
        {                       // 认为丢包
           rxStats.lostPkts += segNumber - rxStats.expectedSeqNum;
           MyDate[2] = rxStats.lostPkts;///
           rxStats.expectedSeqNum = segNumber + 1;
           MyDate[0] = rxStats.expectedSeqNum;///
        }
        else   // 不是所期望收到的数据包（收到的数据包的序号小于期望收到的数据包的序号）
        {       // 认为是一个新的测试开始，复位统计变量
           rxStats.expectedSeqNum = segNumber + 1;
           MyDate[0] = rxStats.expectedSeqNum;///
           rxStats.rcvdPkts = 0;
           rxStats.lostPkts = 0;
        }
        MyDate[1] = rxStats.rcvdPkts;///
        rxStats.rcvdPkts++;
        UartTX_Send_String(MyDate,5);
        halMcuWaitMs(300);
        halLedClear(1);     //熄灭 LED1
        halLedClear(2);     //熄灭 LED2
```

```
        halMcuWaitMs(300);
      }
    }
}
```

在接收主功能函数中，程序通过 basicRfReceive();接口接收发送器发过来的数据，并用 LED1 灯作指示，每接收到一次数据，灯闪烁一次。

4. 实验步骤

说明：实验之前请按照第一章 windows 开发环境搭建 ZigBee（TI）部分安装开发环境。

1）使用 ZigBee Debuger USB 仿真器连接 PC 机和 ZigBee(CC2530)模块，打开 ZigBee 模块开关供电。

2）打开产品光盘资料里 Compnents\ZigBee\TI\exp\zigbee\点对点无线通信 \ide\srf05_cc2530\iar 里的 per_test.eww 工程。

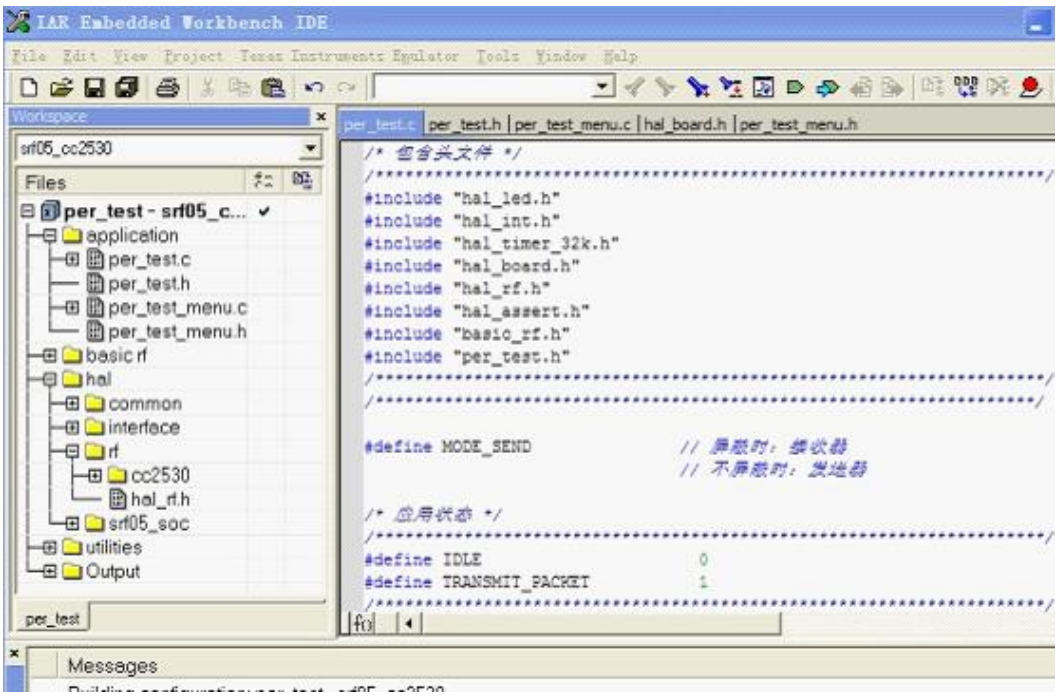


图 4.1 per_test 工程截图

3）在 IAR 开发环境中编译、运行、调试程序。注意，本工程需要编译两次，一次编译为发送器的，一次编译为接收器的，通过 MODE_SEND 宏选择，并分别下载入 2 个 ZigBee 模块中。

in per_test.c

```
//#define MODE_SEND                 // 屏蔽时：接收器
                                    // 不屏蔽时：发送器
```

图 4.2 设备类型选择截图

4）通讯测试：依次打开 2 个分别烧写入发送和接收的 ZigBee 模块，两个模块的 LED1 和 LED2 快速闪烁 8 次后开始通讯，接着发送器的 LED1 和 LED2 交替闪烁，接收器的 LED1 接收到一次数据闪烁一次，LED2 熄灭。

实验二. 点对多点无线通讯实验

1. 实验环境

- 硬件：ZigBee(CC2530)模块(三个)，ZigBee 下载调试板，USB 仿真器，PC 机。
- 软件：IAR Embedded Workbench for MCS-51

2. 实验内容

- 了解 CC2530 芯片点对多点通讯操作过程，掌握 FDMA 方式对该模块进行配置。
- 使用 IAR 开发环境设计程序，利用 1 个接收模块实现对 2 个不同频道上的发送模块进行点对多点无线通讯。

3. 实验原理

3.1 ZigBee(CC2530)模块 LED 硬件接口

图 3.1.1 LED 硬件接口

ZigBee(CC2530)模块硬件上设计有 2 个 LED 灯，用来编程调试使用。分别连接 CC2530 的 P1_0、P、1_1 两个 IO 引脚。从原理图上可以看出，2 个 LED 灯共阳极，当 P1_0、P1_1 引脚为低电平时候，LED 灯点亮。

- **FDMA(频分多址)简介**

无线通讯与有线连接在诸多重要环节上完全不同，这些环节中的异同导致了他们之间的通信质量的差异：

1）无线链路是通过相同的传输媒介——空气来传播无线电信号；

2）误码率比常规有线系统高几个数量级。由于存在上述差异，RF 链路的可靠性比有线 链路低。

3）为了实现在同一范围内多点间通讯，必须考虑防止数据包在空气中的传输时相互碰撞，为了建立可靠的无线传输通路，必须采用各种方法。例如 TDMA/FDMA/CSMA 等都是 无线通讯中常用的办法。

FDMA 是数据通信中的一种技术，即不同的用户分配在时隙相同而频率不同的信道

上。 按照这种技术，把在频分多路传输系统中集中控制的频段根据要求分配给用户。同固定分配系统相比，频分多址使通道容量可根据要求动态地进行交换。

在 FDMA 系统中，分配给用户一个信道，即一对频谱，一个频谱用作前向信道即基站向移动台方向的信道，另一个则用作反向信道即移动台向基站方向的信道。这种通信系统的基站必须同时发射和接收多个不同频率的信号，任意两个移动用户之间进行通信都必须经过基站的中转，因而必须同时占用 2 个信道(2 对频谱)才能实现双工通信。

以往的模拟通信系统一律采用 FDMA。频分多址（FDMA）是采用调频的多址技术，信道在不同的频段分配给不同的用户。如 TACS 系统、AMPS 系统等。频分多址是把通信系统的总频段划分成若干个等间隔的频道(也称信道)分配给不同的用户使用。这些频道互不交叠，其宽度应能传输一路数字话音信息，而在相邻频道之间无明显的串扰。

频分多址(FDMA)技术将可用的频率带宽拆分为具有较窄带宽的子信道，如图所示。 这样每个子信道均独立于其它子信道，从而可被分配给单个发送器。其优点是软件控制上比较简单，其缺陷是子信道之间必须间隔一定距离以防止干扰，频带利用率不高。

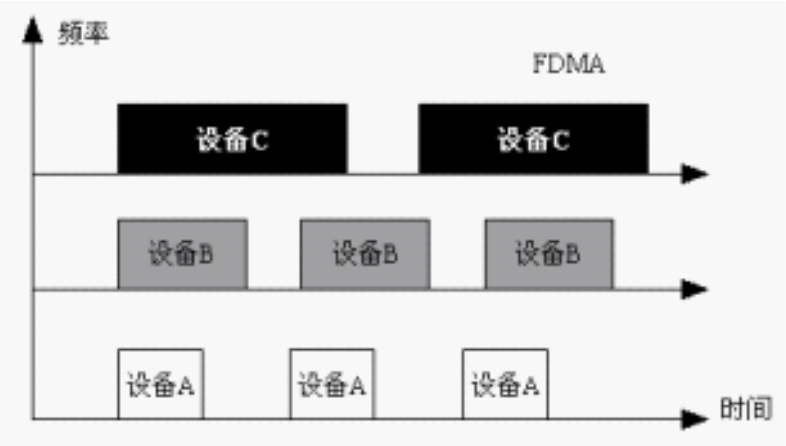


图 3.1.2 FDMA 原理图

3.2 软件流程图

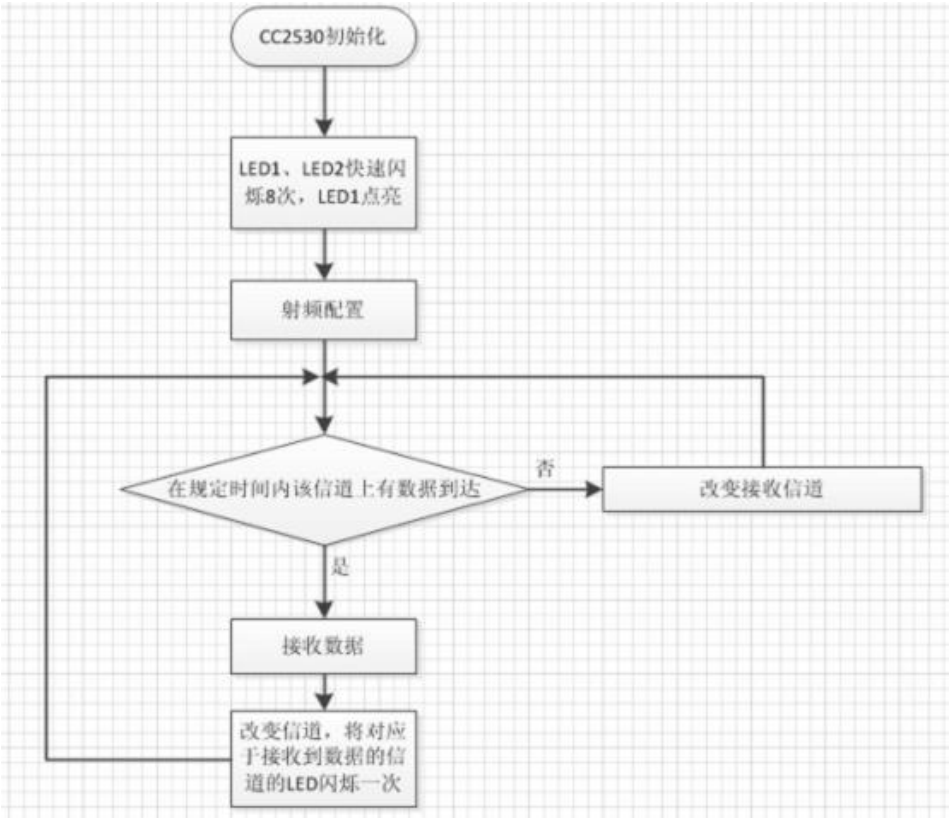


图 3.2.1 接收流程图

FDMA 接收程序主要是在两个频道上循环监听，如果有收到发送模块的信号或一定时间内没有接收到该频道上的信号，就跳到 另外一个频道继续监听。

程序首先是初始化程序，初始化射频部分和内部 CPU。然后程序进入主循环部分，等待接收信号，如果接收到 0x0b 频道上的数据，LED1 闪烁一次，并改变频道为 0x0c,如果接收到 0x0c 频道上的数据，LED2 闪烁一次，并改变频道为 0x0b.

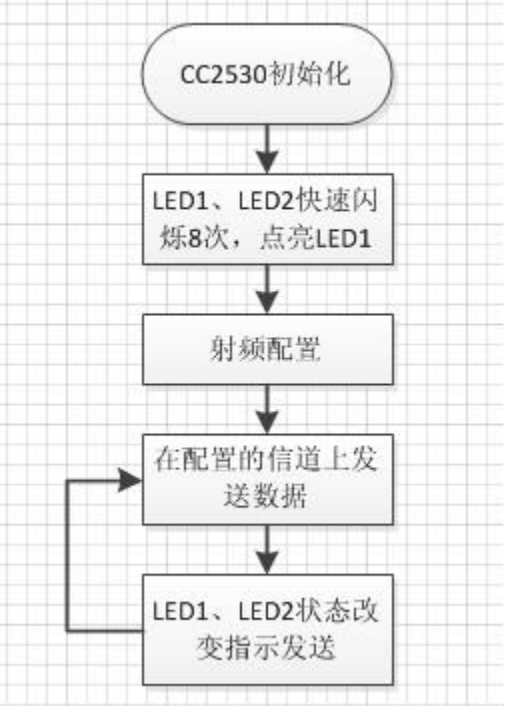


图.3.2.2 发送流程图

FDMA 发送程序主要功能为循环发送数据，程序开始同样是初始化程序，初始化射频部分和内部 CPU，然后两个发送模块在各自的频道上循环发送数据给接收模块。并使 LED1 和 LED2 交替闪烁来指示。

3.3 关键函数分析

1）射频初始化函数

uint8 halRfInit(void)

功能描述：zigbee 通信设置，自动应答有效，设置输出功率 0dbm，Rx 设置，接收中断有效。

参数描述：无

返回：配置成功返回 SUCCESS

2）发送数据包函数

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

功能描述：发送包函数。

入口参数：destAddr 目标网络短地址

pPayload 发送数据包头指针，

length 包的大小

出口参数：无

返 回 值：成功返回 SUCCESS，失败返回 FAILED

3）接收数据函数

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

功能描述： 从接收缓存中拷贝出最近接收到的包。

参数：接收数据包头指针

接收包的大小

返回：实际接收的数据字节数

源码实现：

per_test.c

```
void main (void)
{
    uint8 i;

    appState = IDLE;                // 初始化应用状态为空闲
    appStarted = FALSE;             // 初始化启动标志位 FALSE

    /* 初始化 Basic RF */
    basicRfConfig.panId = PAN_ID;           // 初始化个域网 ID
    basicRfConfig.ackRequest = FALSE;       // 不需要确认

    halBoardInit();

    if(halRfInit()==FAILED)         //初始化 hal_rf
      HAL_ASSERT(FALSE);

    /* 快速闪烁 8 次 led1,led2 */
    for(i = 0; i < 16; i++)
    {
      halLedToggle(1);   // 切换 led1 的亮灭状态
      halLedToggle(2);   // 切换 led2 的亮灭状态
      halMcuWaitMs(50); // 延时大约 50ms
    }

    halLedSet(1);           // led1 指示灯亮，指示设备已上电运行
    halLedClear(2);

    basicRfConfig.channel = 0x0B;           // 设置信道

#ifdef MODE_SEND
    appTransmitter();       // 发送器模式
#else
    appReceiver();          // 接收器模式
#endif
```

```
    HAL_ASSERT(FALSE);
}
```

通过上面的代码分析可知，程序通过宏 MODE_SEND 来确定是发送器还是接收器，appTransmiter()是发送器的主要功能函数，appReceiver()是接收器的主要功能函数，这两个函数最终都会进入一个无限循环状态。

```
static void appTransmitter()
{
    uint32 burstSize=0;
    uint32 pktsSent=0;
    uint8 n;
    uint16 i;

    /* 初始化 Basic RF */
    basicRfConfig.myAddr = TX_ADDR;
#ifdef MODE_SEND_1
    basicRfConfig.channel = 0x0b;          // 设置信道
#else
    basicRfConfig.channel = 0x0c;          // 设置信道
#endif

    if(basicRfInit(&basicRfConfig)==FAILED)
    {
      HAL_ASSERT(FALSE);
    }

    halRfSetTxPower(2);          //HAL_RF_TXPOWER_4_DBM   设置输出功率

    burstSize = 100000;          //设置进行一次测试所发送的数据包数量
    basicRfReceiveOff();         //Basic RF 在发送数据包前关闭接收器，在发送完一个数据包后打开接收器

    appConfigTimer(0xC8);        //配置定时器和 IO

    /* 初始化数据包载荷 */
    txPacket.seqNumber = 0;
    for(n = 0; n < sizeof(txPacket.padding); n++)
    {
      txPacket.padding[n] = n;
    }

    /* 主循环 */
    while (TRUE)
    {
        if (pktsSent < burstSize)
```

```
        {
            UINT32_HTON(txPacket.seqNumber);   // 改变发送序号的字节顺序
            basicRfSendPacket(RX_ADDR, (uint8*)&txPacket, PACKET_SIZE);

            UINT32_NTOH(txPacket.seqNumber);      //在增加序号前将字节顺序改回为主机顺序
            txPacket.seqNumber++;

            pktsSent++;
            appState = IDLE;
            if(i%300 == 0)
            {
                halLedToggle(1);     //切换 LED1 的亮灭状态
                halLedToggle(2);     //切换 LED2 的亮灭状态
            }
            i++;
           // halMcuWaitMs(1000);
        }

        pktsSent = 0;     // 复位统计和序号
    }
}
```

在发送主功能函数里面，首先通过宏定义 MODE_SEND_1 来对发送信道进行选择，然后在该信道上通过 basicRfSendPacket();发送接口函数不停向外发送数据，并改变 LED1，LED2 的状态。

```
static void appReceiver()
{
    uint32 segNumber=0;                                    // 数据包序列号
    int16 perRssiBuf[RSSI_AVG_WINDOW_SIZE] = {0};      // 存储 RSSI 的环形缓冲区
    uint8 perRssiBufCounter = 0;                          // 计数器用于 RSSI 缓冲区统计
    perRxStats_t rxStats = {0,0,0,0};                    // 接收状态
    int16 rssi;
    uint8 resetStats=FALSE;
    uint16 rxTimerOut=5000;

    /* 初始化 Basic RF */
    basicRfConfig.myAddr = RX_ADDR;
    basicRfConfig.channel = 0x0b;              // 设置信道
    if(basicRfInit(&basicRfConfig)==FAILED)
    {
        HAL_ASSERT(FALSE);
    }
    basicRfReceiveOn();

    /* 主循环 */
    while (TRUE)
```

```
    {
        while(!basicRfPacketIsReady())      // 等待新的数据包
        {
            if(!(rxTimerOut--))
            {
              changeChannel();              //改变接收通道
              rxTimerOut=50000;
              continue;
            }
        };
        rxTimerOut=5000;
        if(basicRfReceive((uint8*)&rxPacket, MAX_PAYLOAD_LENGTH, &rssi)>0)
        {
            if(basicRfConfig.channel == 0x0b)          halLedSet(1);          // 如果在 0x0b 通道上接收到数据，点亮 LED1
            if(basicRfConfig.channel == 0x0c)          halLedSet(2);          // 如果在 0x0c 通道上接收到数据，点亮 LED2
            changeChannel();        //改变接收通道

            UINT32_NTOH(rxPacket.seqNumber);   // 改变接收序号的字节顺序
            segNumber = rxPacket.seqNumber;

            /* 若果统计被复位，设置期望收到的数据包序号为已经收到的数据包序号 */
            if(resetStats)
            {
                rxStats.expectedSeqNum = segNumber;
                resetStats=FALSE;
            }

            rxStats.rssiSum -= perRssiBuf[perRssiBufCounter];   // 从 sum 中减去旧的 RSSI 值
            perRssiBuf[perRssiBufCounter] =    rssi;                    // 存储新的 RSSI 值到环形缓冲区，之后它将被加入 sum

            rxStats.rssiSum += perRssiBuf[perRssiBufCounter];   // 增加新的 RSSI 值到 sum
            if(++perRssiBufCounter == RSSI_AVG_WINDOW_SIZE)
            {
              perRssiBufCounter = 0;
            }

            /* 检查接收到的数据包是否是所期望收到的数据包 */
            if(rxStats.expectedSeqNum == segNumber)   // 是所期望收到的数据包
            {
                rxStats.expectedSeqNum++;
            }
            else if(rxStats.expectedSeqNum < segNumber)   // 不是所期望收到的数据包（收到的数据包的序号大于期望收到的数据包的序号）
```

```
            {                    // 认为丢包
              rxStats.lostPkts += segNumber - rxStats.expectedSeqNum;
              rxStats.expectedSeqNum = segNumber + 1;
            }
            else  // 不是所期望收到的数据包（收到的数据包的序号小于期望收到的数据包的序号）
            {      // 认为是一个新的测试开始，复位统计变量
              rxStats.expectedSeqNum = segNumber + 1;
              rxStats.rcvdPkts = 0;
              rxStats.lostPkts = 0;
            }
            rxStats.rcvdPkts++;

            halMcuWaitMs(300);
            halLedClear(1);     //熄灭 LED1
            halLedClear(2);     //熄灭 LED2
            halMcuWaitMs(300);
          }
        }
}
```

在接收主功能函数中，程序通过 basicRfReceive();接口在不同信道上接收数据，并用 LED1 和 LED2 来指示是在哪一信道上接收到数据，如果接收到其中一个信道上的数据或一定时间内未接收到该信道上的数据，则通过 changeChannel();来跳到另外一个信道上接收数据。

4. 实验步骤

1）使用 ZigBee Debuger USB 仿真器连接 PC 机和 ZigBee(CC2530)模块，打开 ZigBee 模块开关供电。

2）打开光盘资料里 Compnents\ZigBee\TI\exp\zigbee\点对多点无线通信 FDMA\ide\srf05_cc2530\iar 里的 per_test.eww 工程。

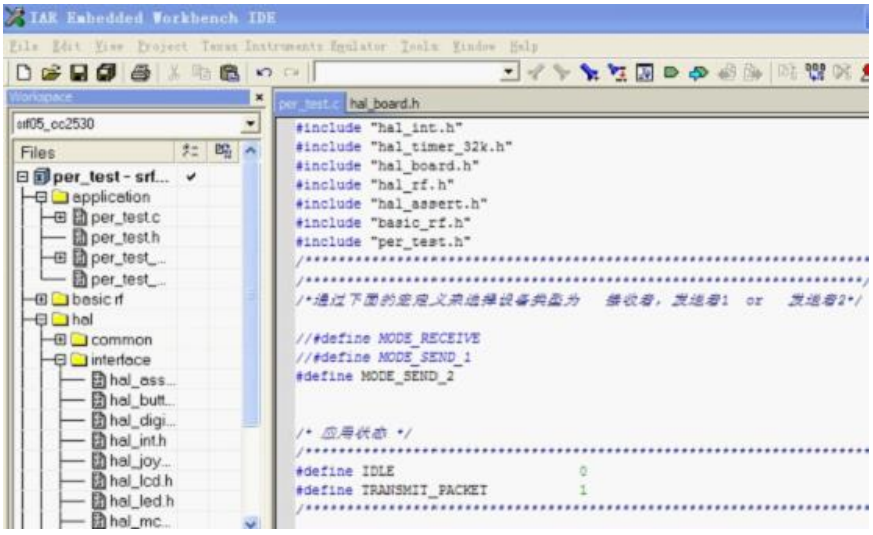


图 4.1　per_test 工程截图

3）在 IAR 开发环境中编译、运行、调试程序。注意，本工程需要编译三次，分别编译为发送器 1、发送器 2、接收器，通过 MODE_RECEIVE、MODE_SEND1、MODE_SEND2 宏选择，并分别下载入 3 个 ZigBee 模块中。

In per_test.c

```
/*********************************************************************/
/*通过下面的宏定义来选择设备类型为   接收者，发送者1  or   发送者2*/

//#define MODE_RECEIVE
//#define MODE_SEND_1
#define MODE_SEND_2
```

图 4.2　设备类型选择截图

4）通讯测试：依次打开 3 个分别烧写入发送 1、发送 2 和接收的 ZigBee 模块，三个模块的 LED1 和 LED2 快速闪烁 8 次后开始通讯，接着发送器的 LED1 和 LED2 交替闪烁，接收器接收到发送器 1 发过来的数据 LED1 闪烁一次，接收到发送器 2 发过来的数据 LED2 闪烁一次。

实验三. TI Z-stack2007 协议栈入门实验

1. 实验环境

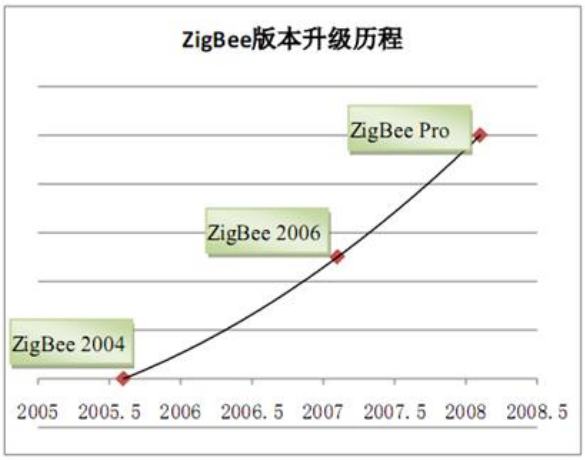
- 硬件：ZigBee(CC2530)模块(两个)，ZigBee 下载调试板，USB 仿真器，PC 机。
- 软件：IAR Embedded Workbench for MCS-51　ZStack-2.3.0-1.4.0 协议栈

2. 实验内容

- 学习 TI Z-Stack2007 协议栈软件架构，掌握 TI Z-Stack 协议栈软件开发流程。有关 Z-Stack2007 协议栈的具体内容，请参考 TI 官方文档。
- 安装 TI Z-Stack2007 协议栈，学习协议栈相关 IAR 工程的配置，及常见软件工具的使用方法。

3. 实验原理

3.1 Z-Stack 概述



ZigBee V1.0：这是第一个 ZigBee 标准公开版，于 2005 年 6 月开放下载。
ZigBee V1.1：第二个 ZigBee 标准公开版，于 2007 年 1 月开放下载，又称为 ZigBee 2006。
ZigBee V1.2：第三个 ZigBee 标准公开版，于 2008 年 1 月开放下载，又称为 ZigBee Pro、ZigBee 2007。

图 3.1.1 ZigBee 规范的版本

目前 TI 的 Z-Stack 协议栈实际上已经成为了 ZigBee 联盟认可并推广的指定软件规范。因此，掌握 Z-Stack 协议栈相关的软件架构及开发流程，是我们学习 ZigBee 无线网络的关键步骤。

3.2 Z-Stack 软件架构

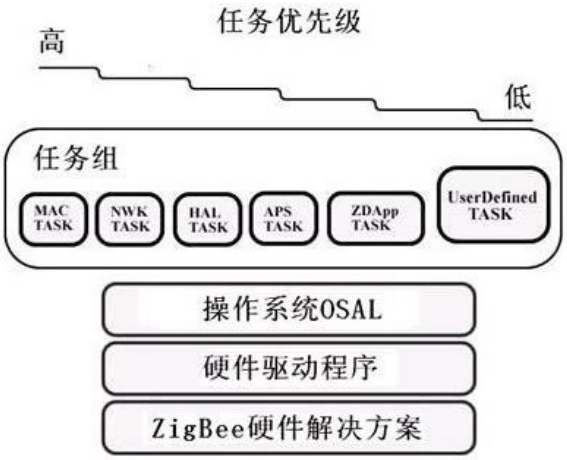


图 3.2.1 Z-Stack 软件架构

协议栈定义了通信硬件和软件在不同层次如何协调工作。在网络通信领域，在每个协议层的实体通过对信息打包与对等实体通信。在通信的发送方，用户需要传递的数据包按照从高层到低层的顺序依次通过各个协议层，每一层的实体按照最初预定消息格式在数据信息中加入自己的信息，比如每一层的头信息和校验等，最终抵达最低层的物理层，变成数据位流，在物理连接间传递。在通信的接收方数据包依次向上通过协议栈，每一层的实体能够根据预定的格式准确的提取需要在本层处理的数据信息，最终用户应用程序得到最终的数据信息并进行处理。

ZigBee 无线网络的实现，是建立在 ZigBee 协议栈的基础上的，协议栈采用分层的结构。协议分层的目的是为了使各层相对独立，每一层都提供一些服务，服务由协议定义，程序员只需关心与他的工作直接相关的那些层的协议，它们向高层提供服务，并由底层提供服务。

在 ZigBee 协议栈中，PHY、MAC 层位于最低层，且与硬件相关；NWK、APS, APL 层以及安全层建立在 PHY 和 MAC 层之上，并且完全与硬件无关。分层的结构脉络清晰、一目了然，给设计和调试带来极大的方便。

整个 Z-Stack 采用分层的软件结构，硬件抽象层（HAL）提供各种硬件模块的驱动，包括定时器 Timer，通用 I/O 口 GPIO，通用异步收发传输器 UART，模数转换 ADC 的应用程序接口 API，提供各种服务的扩展集。操作系统抽象层 OSAL 实现了一个易用的操作系统平台，通过时间片轮转函数实现任务调度，提供多任务处理机制。用户可以调用 OSAL 提供的相关 API 进行多任务编程，将自己的应用程序作为一个独立的任务来实现。

3.3 Z-Stack 软件流程

整个 Z-stack 的主要工作流程，大致分为系统启动，驱动初始化，OSAL 初始化和启动，进入任务轮循几个阶段。

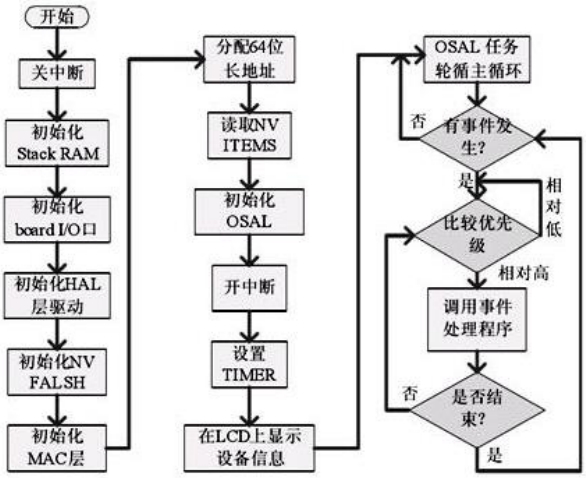


图 3.3.1 Z-Stack 软件流程图

- **系统初始化**

系统上电后，通过执行 ZMain 文件夹中 ZMain.c 的 int main()函数实现硬件的初始化，其中包括关总中断 osal_int_disable(INTS_ALL)、初始化板上硬件设置 HAL_BOARD_INIT()、初始化 I/O 口 InitBoard(OB_COLD)、初始化 HAL 层驱动 HalDriverInit()、初始化非易失性存储器 sal_nv_init(NULL)、初始化 MAC 层 ZMacInit()、分配 64 位地址 zmain_ext_addr()、初始化操作系统 osal_init_system()等。

硬件初始化需要根据 HAL 文件夹中的 hal_board_cfg.h 文件配置寄存器 8051 的寄存器。TI 官方发布 Z-Stack 的配置针对的是 TI 官方的开发板 CC2530EB 等，如采用其他开发板，则需根据原理图设计改变 hal_board_cfg.h 文件配置，例如本文档配套硬件模块与 TI 官方的 I/O 口配置略有不同，需要参考硬件原理图进行相应修改。

当顺利完成上述初始化时，执行 osal_start_system()函数开始运行 OSAL 系统。该任务调度函数按照优先级检测各个任务是否就绪。如果存在就绪的任务则调用 tasksArr[]中相对应的任务处理函数去处理该事件，直到执行完所有就绪的任务。如果任务列表中没有就绪的任务，则可以使处理器进入睡眠状态实现低功耗。程序流程如图所示。osal_start_system()一旦执行，则不再返回 Main()函数。

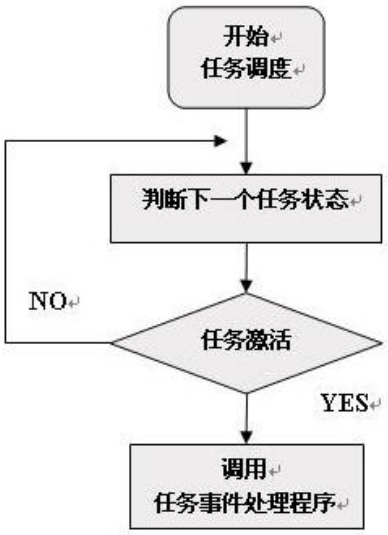


图 3.3.2 OSAL 任务循环

♦ **OSAL 任务初始化**

OSAL 是协议栈的核心，Z-Stack 的任何一个子系统都作为 OSAL 的一个任务，因此在开发应用层的时候，必须通过创建 OSAL 任务来运行应用程序。通过 osalInitTasks()函数创建 OSAL 任务，其中 TaskID 为每个任务的唯一标识号。任何 OSAL 任务必须分为两步：一是进行任务初始化；二是处理任务事件。任务初始化主要步骤如下：

1）初始化应用服务变量。

const pTaskEventHandlerFn tasksArr[]数组定义系统提供的应用服务和用户服务变量，如 MAC 层服务 macEventLoop、用户服务 SampleApp_ProcessEvent 等。

2）分配任务 ID 和分配堆栈内存

void osalInitTasks(void)主要功能是通过调用 osal_mem_alloc()函数给各个任务分配内存空间，和给各个已定义任务指定唯一的标识号。

3）在 AF 层注册应用对象

通过填入 endPointDesc_t 数据格式的 EndPoint 变量，调用 afRegister()在 AF 层注册 EndPoint 应用对象。

通过在 AF 层注册应用对象的信息，告知系统 afAddrType_t 地址类型数据包的路由端点，例如用于发送周期信息的 SampleApp_Periodic_DstAddr 和发送 LED 闪烁指令的 SampleApp_Flash_DstAddr。

4）注册相应的 OSAL 或 HAL 系统服务

在协议栈中，Z-Stack 提供键盘响应和串口活动响应两种系统服务，但是任何 Z-Stask 任务均不自行注册系统服务，两者均需要由用户应用程序注册。值得注意的是，有且仅有一个 OSAL Task 可以注册服务。例如注册键盘活动响应可调用 RegisterForKeys()函数。

5）处理任务事件

处理任务事件通过创建“ApplicationName”_ProcessEvent()函数处理。一个 OSAL 任务除了强制事件（Mandatory Events）之外还可以定义 15 个事件。

SYS_EVENT_MSG（0x8000）是强制事件。该事件主要用来发送全局的系统信息，包括以下信息：

AF_DATA_CONFIRM_CMD：该信息用来指示通过唤醒 AF DataRequest( )函数发送的数据请求信息的情况。ZSuccess 确认数据请求成功的发送。如果数据请求是通过 AF_ACK_REQUEST 置位实现的，那么 ZSuccess 可以确认数据正确的到达目的地。否则，ZSuccess 仅仅能确认数据成功的传输到了下一个路由。

AF_INCOMING_MSG_CMD：用来指示接收到的 AF 信息。

KEY_ CHANGE：用来确认按键动作。

ZDO_ NEW_ DSTADDR：用来指示自动匹配请求。

ZDO_STATE_CHANGE：用来指示网络状态的变化。

3.4 Z-Stack 协议栈目录结构

正如前面所描述的那样，我们以 Z-Stack 协议栈安装后，自带的一个工程 SampleApp 样例为模板，了解下协议栈的目录结构及相关软件流程。

进入到安装好的 TI Z-Stack 协议栈目录 C:\Texas Instruments\ZStack-2.3.0-1.4.0\Projects\zstack\Samples\SampleApp\CC2530DB 中，打开 SampleApp.eww 工程。如图所示：

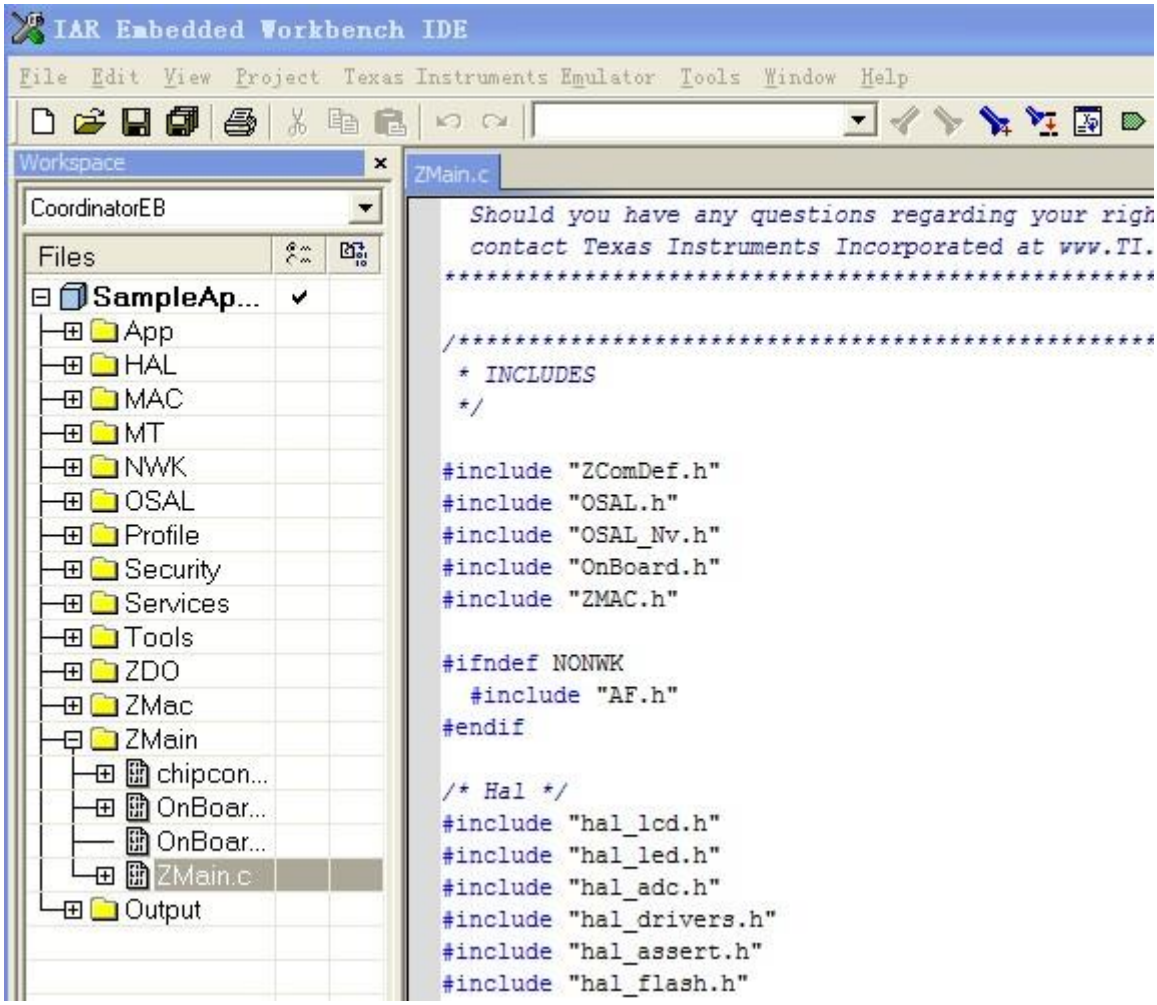


图 3.4.1　SampleApp 工程

APP：应用层目录，这是用户创建各种不同工程的区域，在这个目录中包含了应用层的内容和这个项目的主要内容，在协议栈里面一般是以操作系统的任务实现的。在 APP 目录下添加三个文件即可完成一个新的任务项目（1、主文件：存放具体的任务事件处理函数；2、主文件的头文件；3、操作系统接口文件：以 Osal 开头，专门存放任务处理函数数组 tasksArr[]）。

HAL：硬件层目录，包含有与硬件相关的配置和驱动及操作函数。

Common 目录下为公用文件，基本与硬件无关

Hal_assert.c ：断言文件，用于调试

Hal_drivers.c ：驱动文件

Including 目录下包含各个硬件模块头文件

Target 目录下文件与硬件平台相关

MAC：MAC 层目录，包含了 MAC 层的参数配置文件及其 MAC 的 LIB 库的函数接口文件。

MAC 层分高层和底层，include 目录下包括 MAC 层参数配置文件及其 mac 的 lib 库函数接口文件

MT：监控调试层目录，实现通过串口调试各层，与各层进行直接交互。

NWK：网络层目录，含网络层配置参数文件及网络层库的函数接口文件，APS 层库的函数接口

OSAL：协议栈的操作系统。

Profile：AF 层目录，包含 AF 层处理函数文件。

Security：安全层目录，安全层处理函数，比如加密函数等。

Services：地址处理函数目录，包括着地址模式的定义及地址处理函数。

Tools：工程配置目录，包括空间划分及 ZStack 相关配置信息。

ZDO：ZDO 目录。

ZigBee 设备对象，方便用户用自定义的对象调用 aps 子层的服务和 nwk 层的服务。

ZMac： MAC 层目录，包括 MAC 层参数配置及 MAC 层 LIB 库函数回调处理函数。

Zmac.c 是 z-stack mac 导出层接口文件

Zmac_cb.c 是 zmac 需要调用的网络层函数

ZMain：主函数目录，包括入口函数及硬件配置文件。

在 onboard.c 包含对硬件开发平台各类外设进行控制的接口函数。

Output：输出文件目录，这个 EW8051 IDE 自动生成的。

3.5 ZigBee 系统初始化流程

Z-Stack 的 main 函数在 ZMain.c 中：系统初始化、执行操作系统实体

Osal_int_disable(INTS_ALL):关闭所有中断

HAL_BOARD_INIT():初始化系统时钟

Zmain_vdd_check():检测芯片电压是否正常

Zmain_ram_init():初始化堆栈

InitBoard(OB_COLD):初始化 LED，配置系统定时器

HalDriverInit():初始化芯片各个硬件模块

Osal_nv_init():初始化 FLASH 存储

Zmain_ext_addr():形成节点 MAC 地址

zgInit():初始化一些非易失变量

zmacInit():初始化 mac 层

Afinit():初始化应用框架层

Osal_init_system():初始化操作系统

Osal_int_enabled(inis_all):使能全部中断

Initboard(ob_ready):初始化按键

Zmain_dev_info():在液晶上显示设备信息

Osal_start_system():执行操作系统

4. 实验步骤

- **安装 TI ZStack-2.3.0-1.4.0 协议栈（按照第一章 windows 开发环境搭建 ZigBee 部分安装）。**

安装完成后，默认会在 C 盘 Texas Instruments 目录下发现 ZStack-2.3.0-1.4.0 目录。

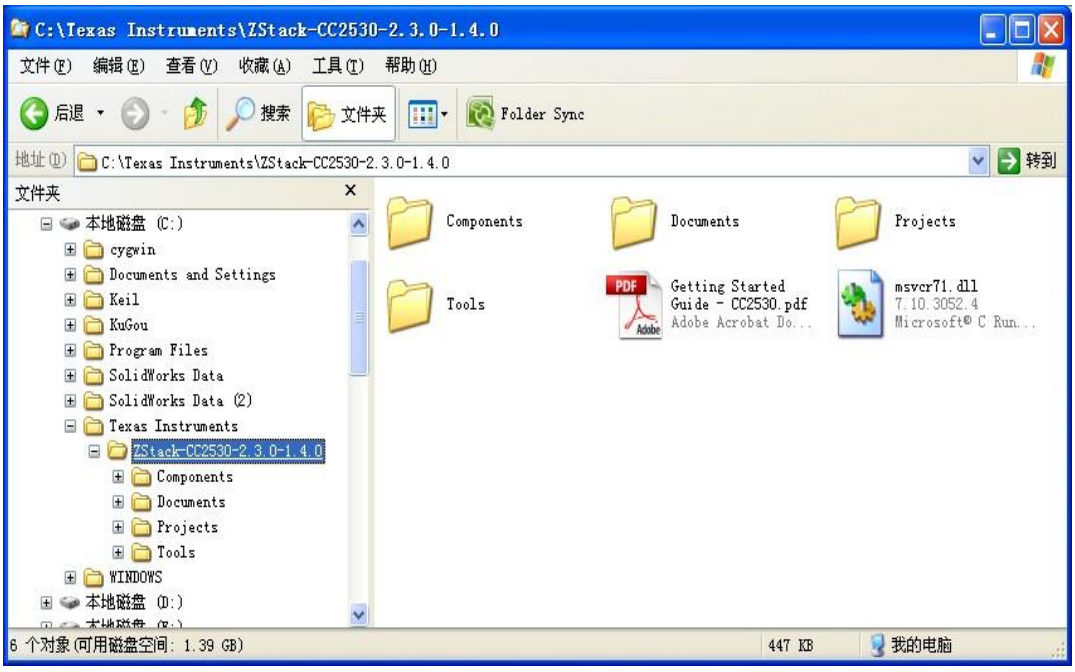


图 4.8　Z-Stack-2.3.0-1.4.0 目录

根目录下有一个安装卸载协议栈的 PDF 说明书，另外包含 Documents、Components、Projects、Tools 文件夹。

1）Document 文件夹包含了对整个协议栈进行说明的说有文档。用户可以把该文件夹下的文档作为学习使用的参考手册。

2）Compnents 文件夹是 Z-Stack 协议栈各个功能部件的实现，包含协议栈各个层次的目录结构。

3）Projects 文件夹包含了 Z-Stack 功能演示的各个项目例程。

4）Tool 文件夹下存放着 TI 自带的网络工具。

♦ **熟悉 TI ZStack-2.3.0-1.4.0 协议栈自带例程的 IAR 工程配置。**

打开 C:\Texas Instruments\ZStack-CC2530-2.3.0-1.4.0\Projects\zstack\Samples\SampleApp\CC2530DB 目录下的工程文件，如图

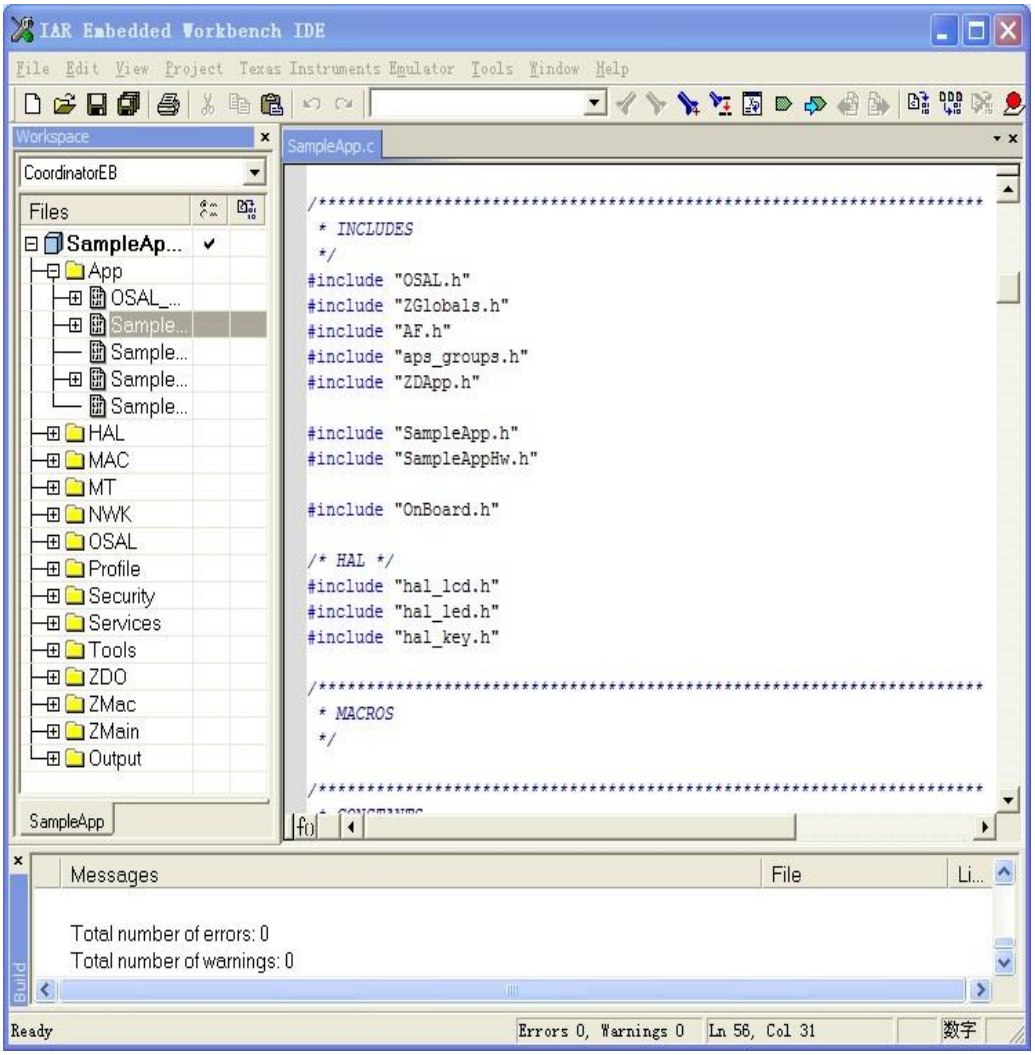


图 4.9　SampleApp 工程界面

在 Workspace 工作区，查看工程模板。本 SampleApp 工程共有 8 个模板，分别对应不同的硬件设备和不同的 ZigBee 设备类型。

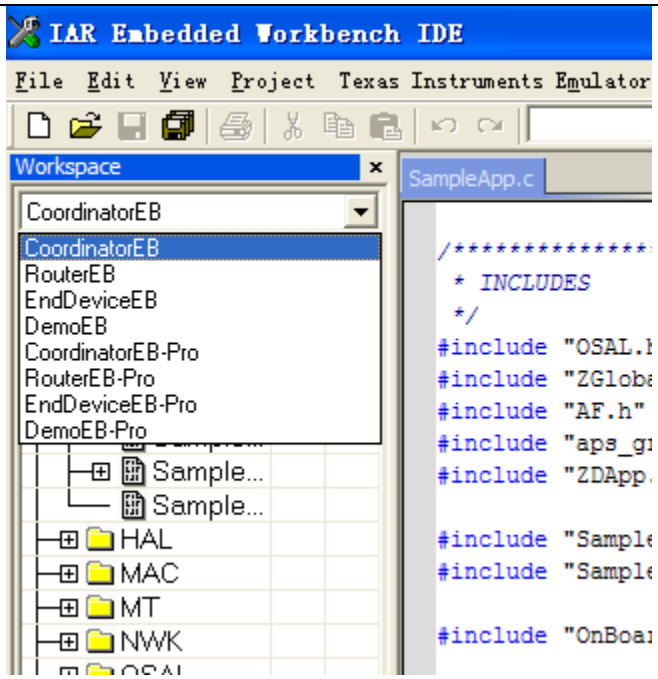


图 4.10 工程模板选择

以 CoordinatorEB 为例，该硬件模板最接近我们实验使用的硬件设备，查看工程相关配置，其中协议栈中使用相关的宏定义来控制设备流程及类型，可以在工程的 Options->C/C++Compiler->Preprocessor 中查看：

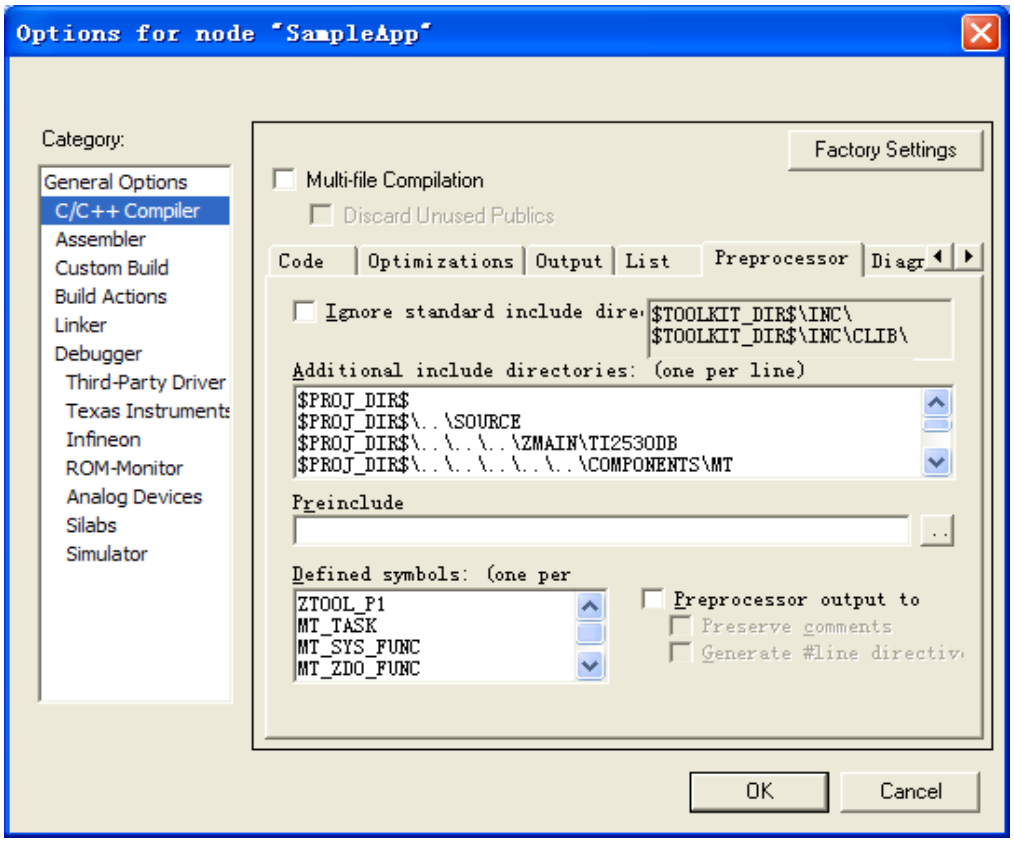


图 4.11 宏定义配置

通过对比不同的工程，我们可以发现不同工程的 Defined symbols 定义不同，从而实现不同功能流程的控制。

此外，我们还可以通过工程的 Options->C/C++Compiler->Extra Options 选项查看该工程模板的配置文件：

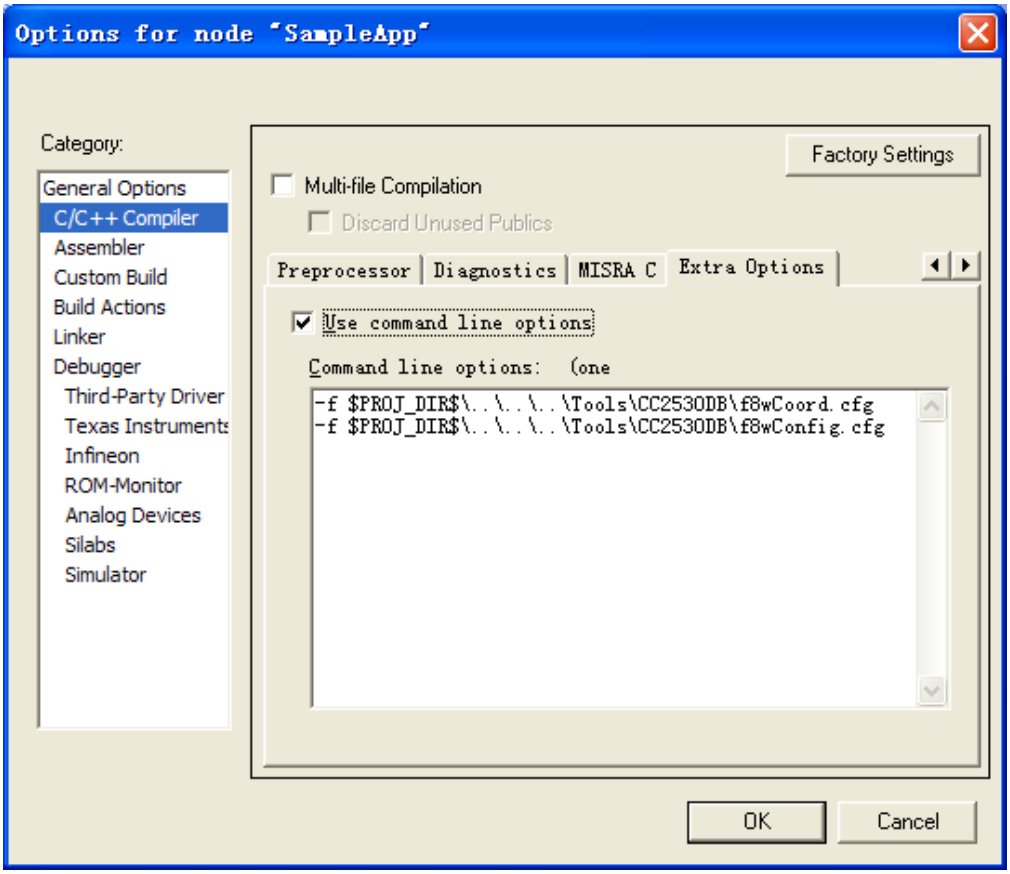


图 4.12 工程配置文件

在 f8wConfig.cfg 等配置文件中定义了工程相关的网络通讯设置。其中比较重要的是和 ZigBee 通信相关的信道通道的设置，和 PAN ID 的设置，用户可以通过更改该文件中的相关宏定义，来控制 ZigBee 网络的通道和 PAN ID。以此来解决多个 ZigBee 网络的冲突问题。

其他工程设置，用户可以自行查阅。

由于我们的开发板硬件资源与官方的有区别，为了使 LED 灯能正常使用，请打开 hal_board_cfg.h 。按照下面的代码进行修改。

```
#define LED1_BV              BV(0)
#define LED1_SBIT            P1_0
#define LED1_DDR             P1DIR
#define LED1_POLARITY        ACTIVE_LOW

#if defined (HAL_BOARD_CC2530EB_REV17)
  /* 2 - Red */
  #define LED2_BV              BV(1)
  #define LED2_SBIT            P1_1
  #define LED2_DDR             P1DIR
  #define LED2_POLARITY        ACTIVE_LOW

  /* 3 - Yellow */
  #define LED3_BV              BV(0)
```

```
  #define LED3_SBIT             P1_0
  #define LED3_DDR              P1DIR
  #define LED3_POLARITY         ACTIVE_LOW
#endif
```

- **编译工程，并下载调试**

选择相应的模板工程，进行编译，下载调试。

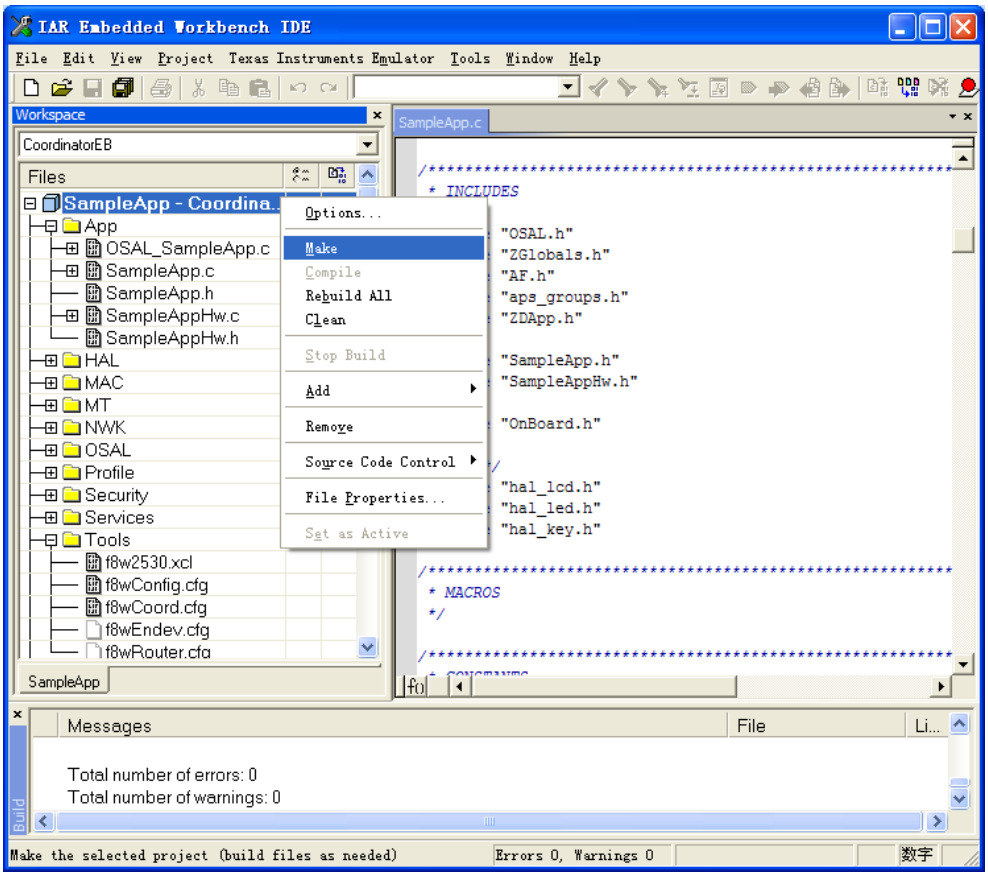


图 4.13 编译工程

连接 ZigBee 模块硬件，使用 ZigBee Debuger USB 仿真器连接 ZigBee 模块，上电后，即可通过 IAR 工程的 Debug 来下载并调试。

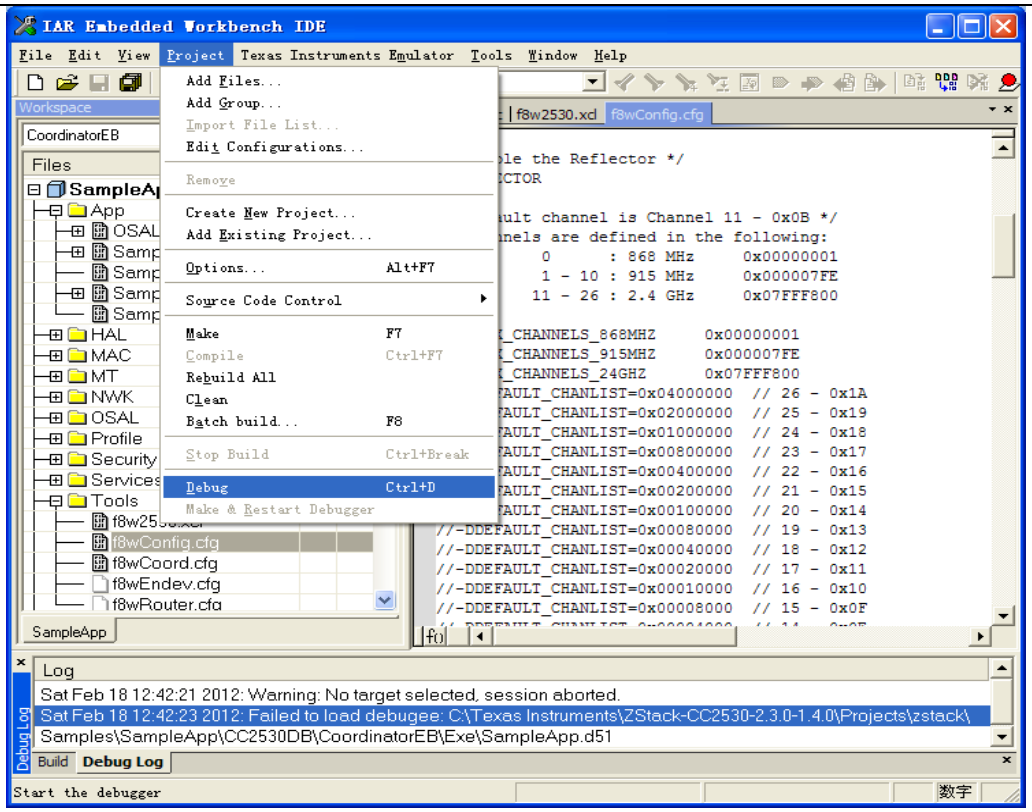


图 4.14 下载调试工程

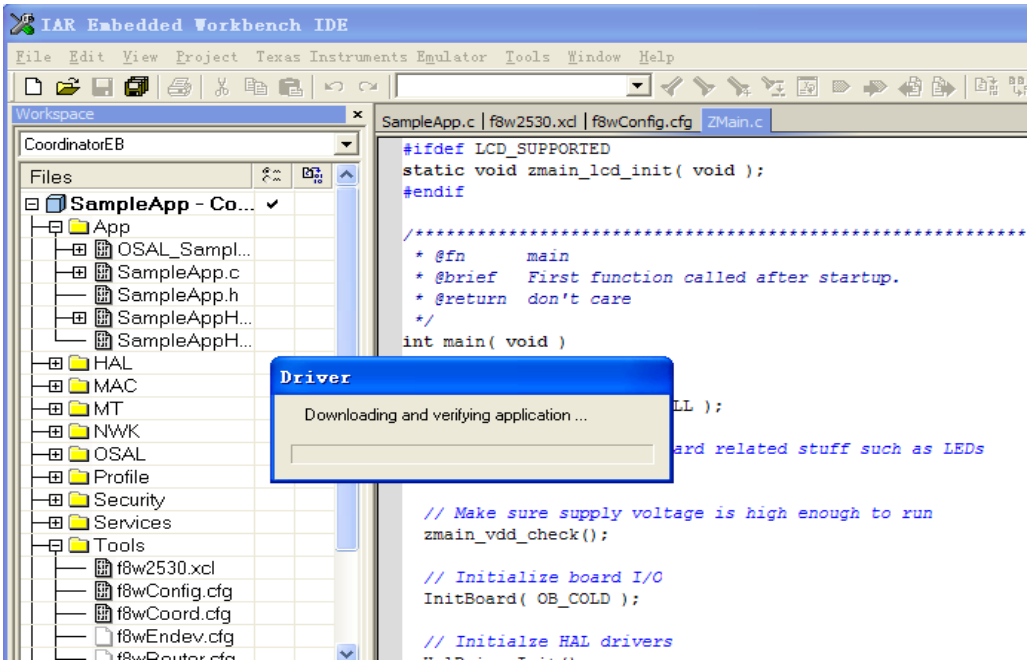


图 4.15 下载工程中

如果下载调试出现异常，可以尝试重启 USB 仿真器或重启 ZigBee 模块。

点击运行程序

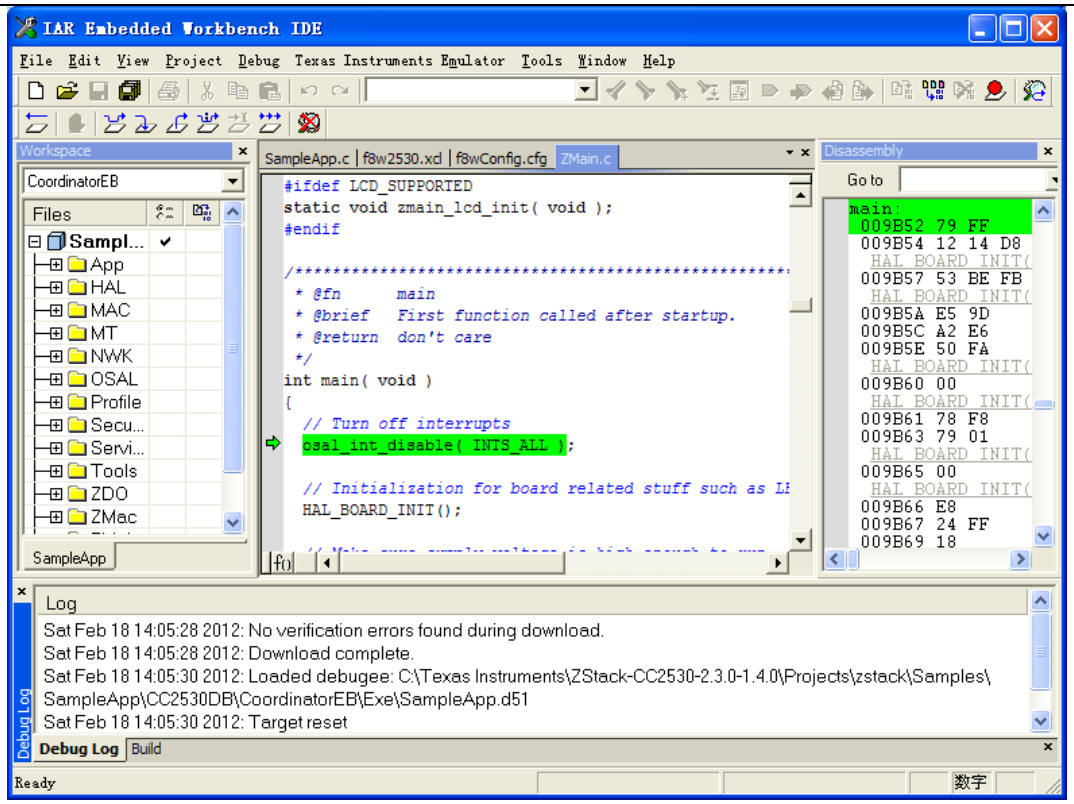


图 4.16 运行工程

第一次使用 ZigBee 模块，可能会因为 ZigBee 模块的 MAC 地址无效(默认协议栈工程使用按键事件来设置 MAC 地址，而我们的设备无按键，因此需要手动修改 MAC 地址)，而无法正常运行程序，需要我们使用 ChipconFlashProgrammer 软件来手动修改 MAC 地址，才能正确运行程序。

- **使用 ChipconFlashProgrammer 软件更改 MAC 物理地址(非必须项)**

ZigBee 模块默认出厂地址为 64 位的 0xFF 无效地址，该地址为 ZigBee 的全球唯一地址。因此可以通过 SmartRFProgr_1.6.2 软件来更改该物理地址，以此完成基于 Z-Stack 协议栈的实验。

1）打开光盘 tools 目录，运行 Setup_SmartRFProgr_1.6.2.exe 程序

图 4.17 SmartRFProgr_1.6.2 软件

2）连接 好 ZigBee 模块和 USB 仿真器后，即可检测到 ZigBee 模块。

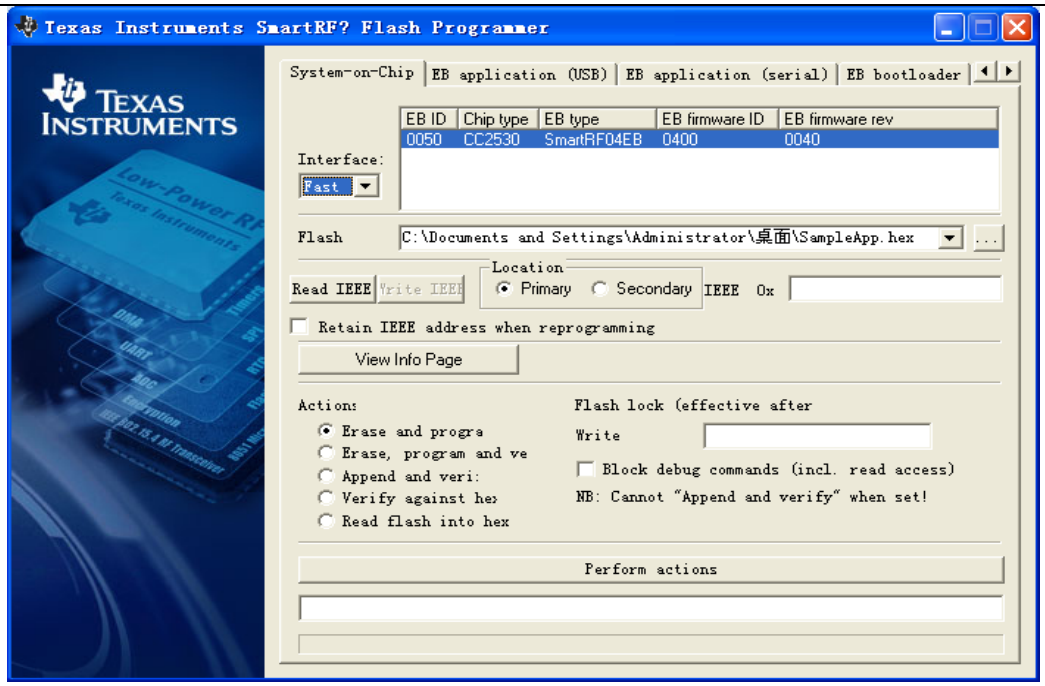


图 4.18 检测设备

连接时候保证硬件连接真确，可以手动复位 USB 仿真器或 ZigBee 模块，且保证 IAR 工程中退出 Debug 调试模式。

3）使用 Reed IEEE 来读 ZigBee 物理地址

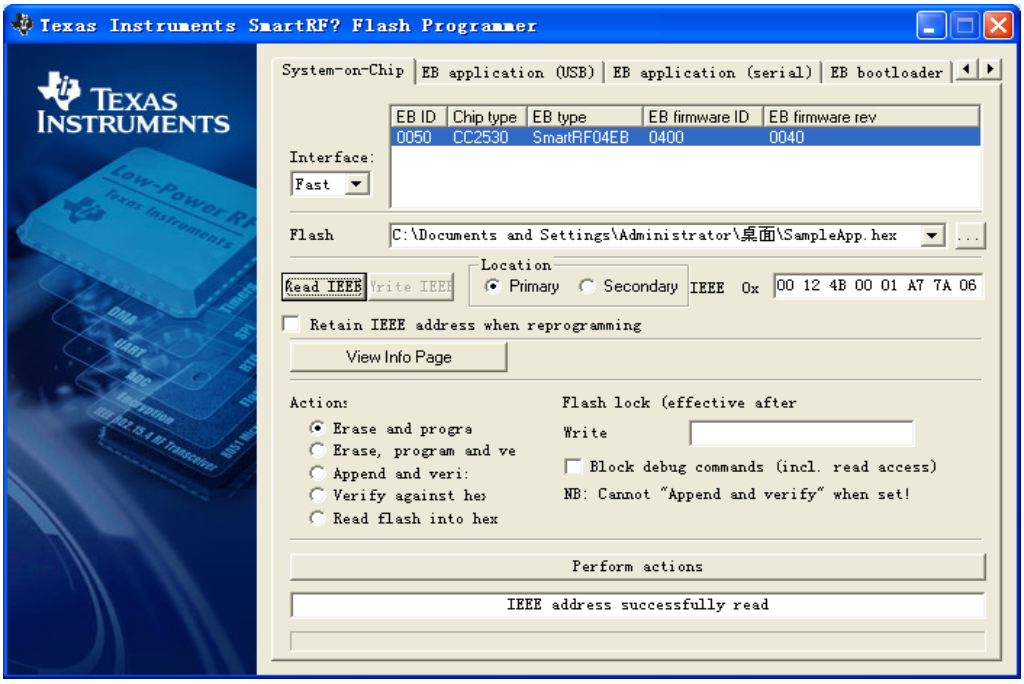


图 4.19 读 IEEE 地址

如果监测到 ZigBee 模块，即可读出该模块的 64 位物理地址。

4）手动修改 IEEE 物理地址，使用 Write IEEE 烧写。

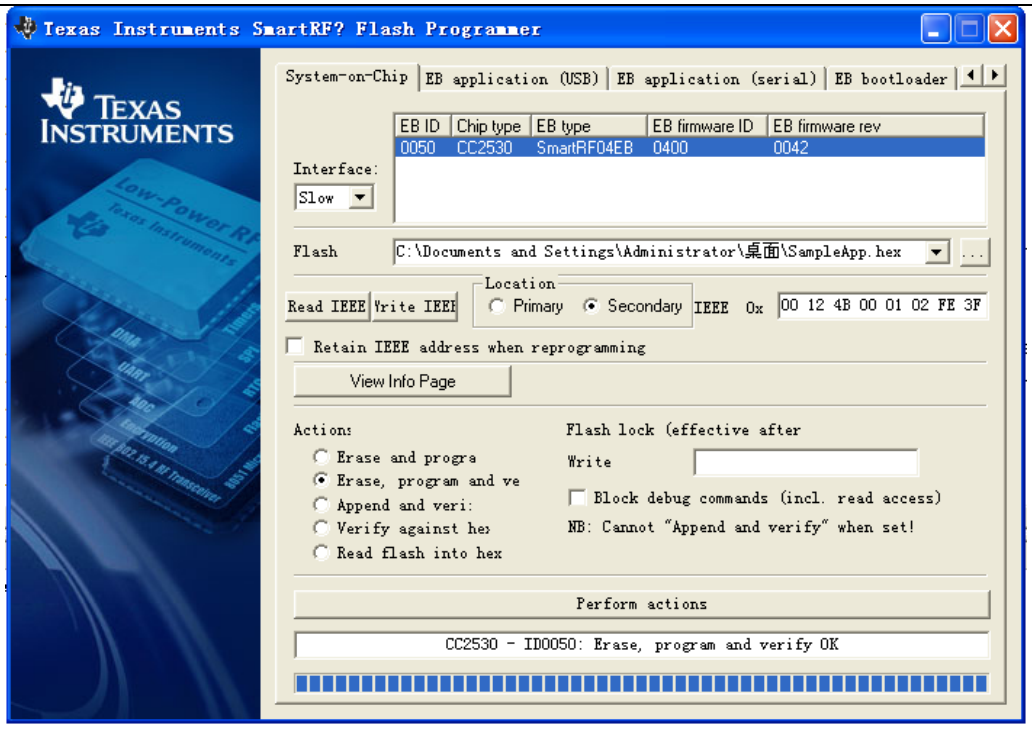


图 4.20 修改 IEEE 地址

保证写入的 IEEE 地址为非 0xFF 且不与局域网中其他模块 IEEE 地址冲突即可。当底侧显示 IEEE address successfully written to chip 表示写入成功。修改好 ZigBee 模块的 IEEE 地址后，即可重新下载调试程序。

此时，如果 ZigBee 模块下载了协调器的工程二进制程序，则该 ZigBee 模块可以作为协调器运行并会自动创建网络，如果下载了 EndDeviceDB 工程，在 ZigBee 模块作为节点端模块，同样，如果下载了 RouterDB 工程，则 ZigBee 作为路由模块使用。

用户可以自行运行，下载工程文件，进行 ZigBee 网络的功能测试。本实验目的在于让读者熟悉 TI Z-Stack 协议栈的软件结构及开发流程，以及一些常用工具的使用。具体的工程功能实现，将在后续实验讲述。

实验四. 基于 Z-Stack 的无线组网实验

1. 实验环境

- 硬件：ZigBee(CC2530)模块(两个)，ZigBee 下载调试板，USB 仿真器，PC 机。
- 软件：IAR Embedded Workbench for MCS-51　ZStack-2.3.0-1.4.0 协议栈。

2. 实验内容

- 学习 TI ZStack2007 协议栈内容，掌握 CC2530 模块无线组网原理及过程。有关 Z-Stack2007 协议栈的具体内容，请参考附录中相关说明及 TI 官方文档。
- 使用 IAR 开发环境设计程序，ZStack-2.3.0-1.4.0 协议栈源码例程 SampleApp 工程基础上，实现无线组网及通讯。即协调器自动组网，终端节点自动入网，并发送周期信息“~HELLO!~”广播，协调器接收到消息后将数据通过串口发送给 PC 计算机。

3. 实验原理

3.1 ZigBee(CC2530)模块 LED 硬件接口

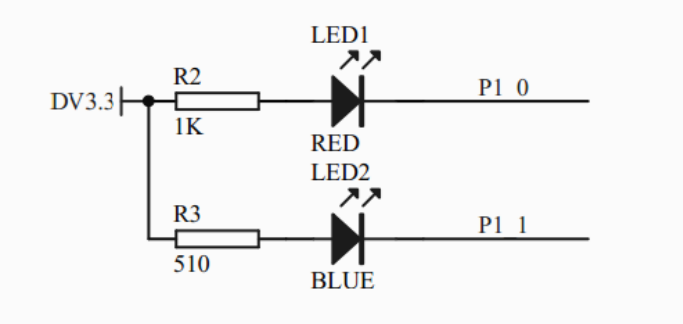


图 3.1.1 LED 硬件接口

ZigBee(CC2530)模块硬件上设计有 2 个 LED 灯，用来编程调试使用。分别连接 CC2530 的 P1_0、P、1_1 两个 IO 引脚。从原理图上可以看出，2 个 LED 灯共阳极，当 P1_0、P1_1 引脚为低电平时候，LED 灯点亮。

3.2 SampleApp 实验简介

SampleApp 实验是协议栈自带的 ZigBee 无线网络自启动(组网)样例，该实验实现的功能主要是协调器自启动(组网)，节点设备自动入网。之后两者建立无线通讯，数据的发送主要有 2 中方式，一种为周期定时发送信息(本次实验采用该方法测试)，另一种需要通过按键事件触发发送 FLASH 信息。由于实验配套 ZigBee 模块硬件上与 TI 公司的 ZigBee 样板有差异，因此本次实验没有采用按键触发方式。

接下来我们分析发送 periodic 信息流程(发送按键事件 flash 流程略)

Periodic 消息是通过系统定时器开启并定时广播到 group1 出去的,因此在 SampleApp_ProcessEvent 事件处理函数中有如下定时器代码：

```
            case ZDO_STATE_CHANGE:
            SampleApp_NwkState = (devStates_t)(MSGpkt->hdr.status);
            if ( (SampleApp_NwkState == DEV_ZB_COORD)
                || (SampleApp_NwkState == DEV_ROUTER)
                || (SampleApp_NwkState == DEV_END_DEVICE) )
                {
                  // Start sending the periodic message in a regular interval.
                  HalLedSet(HAL_LED_1, HAL_LED_MODE_ON);
                 osal_start_timerEx( SampleApp_TaskID,
SAMPLEAPP_SEND_PERIODIC_MSG_EVT,
SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT );
                }
                else
                {
                    // Device is no longer in the network
                }
               break;
```

当设备加入到网络后，其状态就会变化，对所有任务触发 ZDO_STATE_CHANGE 事件，开启一个定时器。当定时时间一到，就触发广播 periodic 消息事件，触发事件 SAMPLEAPP_SEND_PERIODIC_MSG_EVT,相应任务为 SampleApp_TaskID,于是再次调用 SampleApp_ProcessEvent()处理 SAMPLEAPP_SEND_PERIODIC_MSG_EVT 事件，该事件处理函数调用 SampleApp_SendPeriodicMessage()来发送周期信息。

```
if ( events & SAMPLEAPP_SEND_PERIODIC_MSG_EVT )
    {
      SampleApp_SendPeriodicMessage();      // Send the periodic message
      // Setup to send message again in normal period (+ a little jitter)
      osal_start_timerEx( SampleApp_TaskID, SAMPLEAPP_SEND_PERIODIC_MSG_EVT,
          (SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT + (osal_rand() & 0x00FF)) );
      return (events ^ SAMPLEAPP_SEND_PERIODIC_MSG_EVT);        // return unprocessed events
    }
```

- **MT 层串口通讯**

协议栈将串口通讯部分放到了 MT 层的 MT 任务中去处理了，因此我们在使用串口通讯的时候要在编译工程（通常是协调器工程）时候在编译选项中加入 MT 层相关任务的支持：MT_TASK,ZTOOL_P1 或 ZAPP_P1。

- **关于无线组网实验关键代码分析**

```
void SampleApp_SendPeriodicMessage( void )
{
```

```
    char buf[]="~HELLO!~";
    AF_DataRequest( &SampleApp_Periodic_DstAddr, &SampleApp_epDesc,
                                SAMPLEAPP_PERIODIC_CLUSTERID,
                                8,
                                (unsigned char*)buf,
                                &SampleApp_TransID,
                                AF_DISCV_ROUTE,
                                AF_DEFAULT_RADIUS );
}
```

这个函数是终端节点要完成的功能，通过上面对周期事件的分析，可以知道这个函数是会被周期调用的，通过 AF_DataRequest()向协调器周期发送字符串“~HELLO!~”

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

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

              // Received whenever the device changes state in the network
              case ZDO_STATE_CHANGE:
                SampleApp_NwkState = (devStates_t)(MSGpkt->hdr.status);
                if ( (SampleApp_NwkState == DEV_ZB_COORD)
                    || (SampleApp_NwkState == DEV_ROUTER)
                    || (SampleApp_NwkState == DEV_END_DEVICE) )
                {
                  // Start sending the periodic message in a regular interval.
                  HalLedSet(HAL_LED_1, HAL_LED_MODE_ON);
                  osal_start_timerEx( SampleApp_TaskID,
      SAMPLEAPP_SEND_PERIODIC_MSG_EVT,
```

```
        SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT );
                    }
                    else
                    {
                      // Device is no longer in the network
                    }
                    break;

                  default:
                    break;
                }
                osal_msg_deallocate( (uint8 *)MSGpkt );         // Release the memory
                MSGpkt = (afIncomingMSGPacket_t *)osal_msg_receive( SampleApp_TaskID );     // Next - if
one is available
          }

          return (events ^ SYS_EVENT_MSG);       // return unprocessed events
      }

      // Send a message out - This event is generated by a timer
      //   (setup in SampleApp_Init()).
      if ( events & SAMPLEAPP_SEND_PERIODIC_MSG_EVT )
      {
        SampleApp_SendPeriodicMessage();     // Send the periodic message
        // Setup to send message again in normal period (+ a little jitter)
        osal_start_timerEx( SampleApp_TaskID, SAMPLEAPP_SEND_PERIODIC_MSG_EVT,
            (SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT + (osal_rand() & 0x00FF)) );
        return (events ^ SAMPLEAPP_SEND_PERIODIC_MSG_EVT);        // return unprocessed events
      }

      return 0;     // Discard unknown events
}
```

SampleApp_ProcessEvent() 函数为应用层事件处理函数，当接收到网络数据（即发生 AF_INCOMING_MSG_CMD 事件）时，会调用 SampleApp_MessageMSGCB(MSGpkt);处理函数，现在来分析这个函数

```
void SampleApp_MessageMSGCB( afIncomingMSGPacket_t *pkt )
{
      uint16 flashTime;
      unsigned char *buf;

      switch ( pkt->clusterId )
      {
          case SAMPLEAPP_PERIODIC_CLUSTERID:
            buf = pkt->cmd.Data;
            HalUARTWrite(0, buf, 8);
```

```
            HalUARTWrite(0,"\r\n", 2);
            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;
    }
}
```

这个函数是协调器要完成的工作，对终端发过来的消息进行格式转换后发给串口终端。

更详细的处理流程，具体见工程源代码。

4. 实验步骤

使用 ZigBee Debuger USB 仿真器连接 PC 机和 ZigBee(CC2530)模块，打开 ZigBee 模块开关供电。

产品光盘资料里 Compnents\ZigBee\TI\exp\zigbee\无线自组网实验\Projects\zstack\Samples\SampleApp\ CC2530DB 里的工程。

1）打开时若出现下图错误，推荐使用管理员权限重新打开

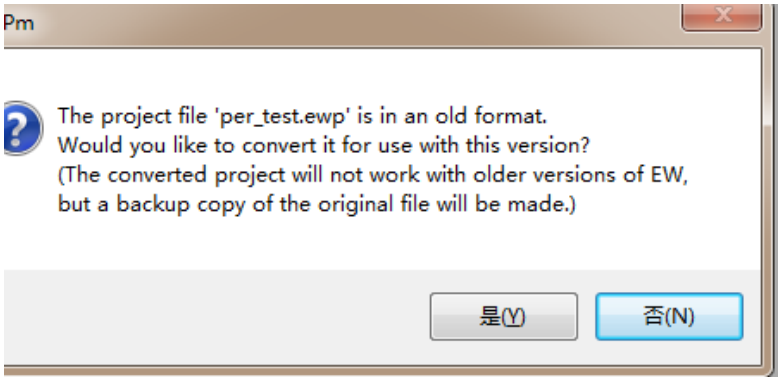


图 4.1 工程打开失败

2）选择 CoordinatorEB 工程，编译下载到 ZigBee COORDINATOR 模块中。

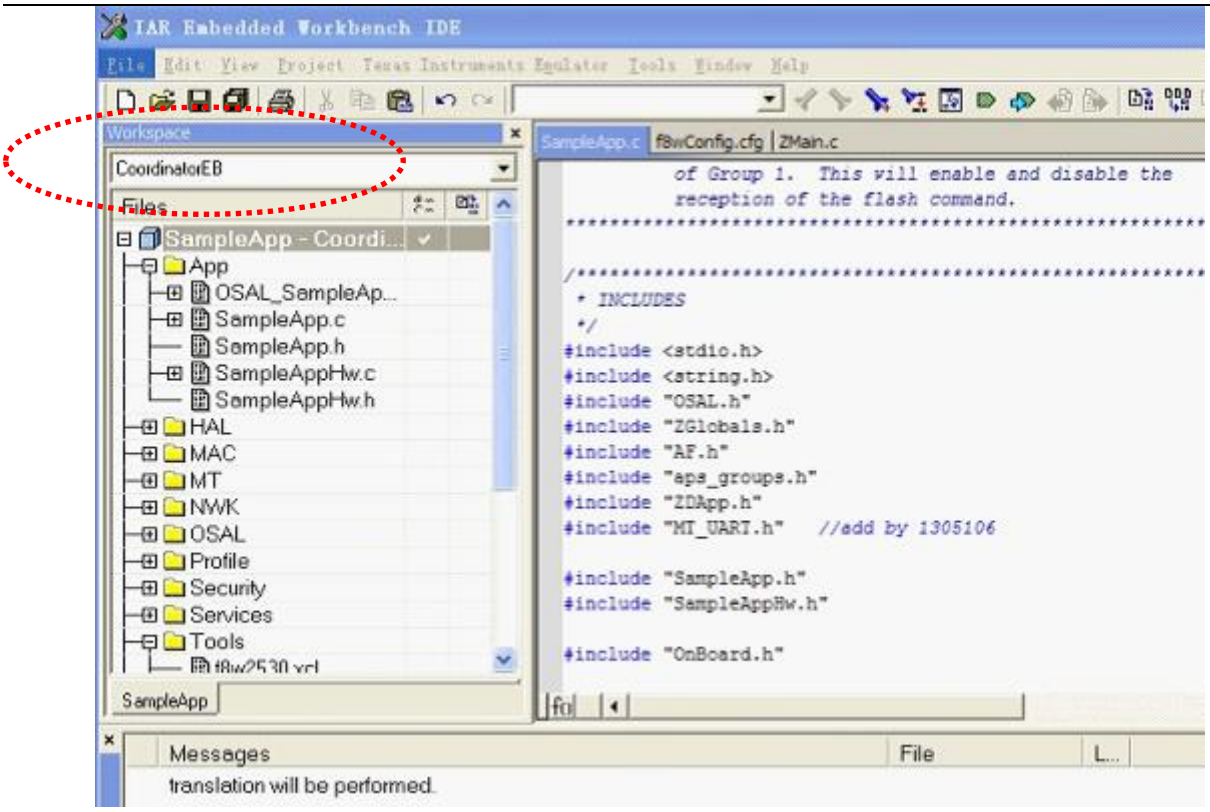


图 4.2 CoordinatorEB 工程

3）选择 EndDeviceEB 工程，编译下载到终端节点

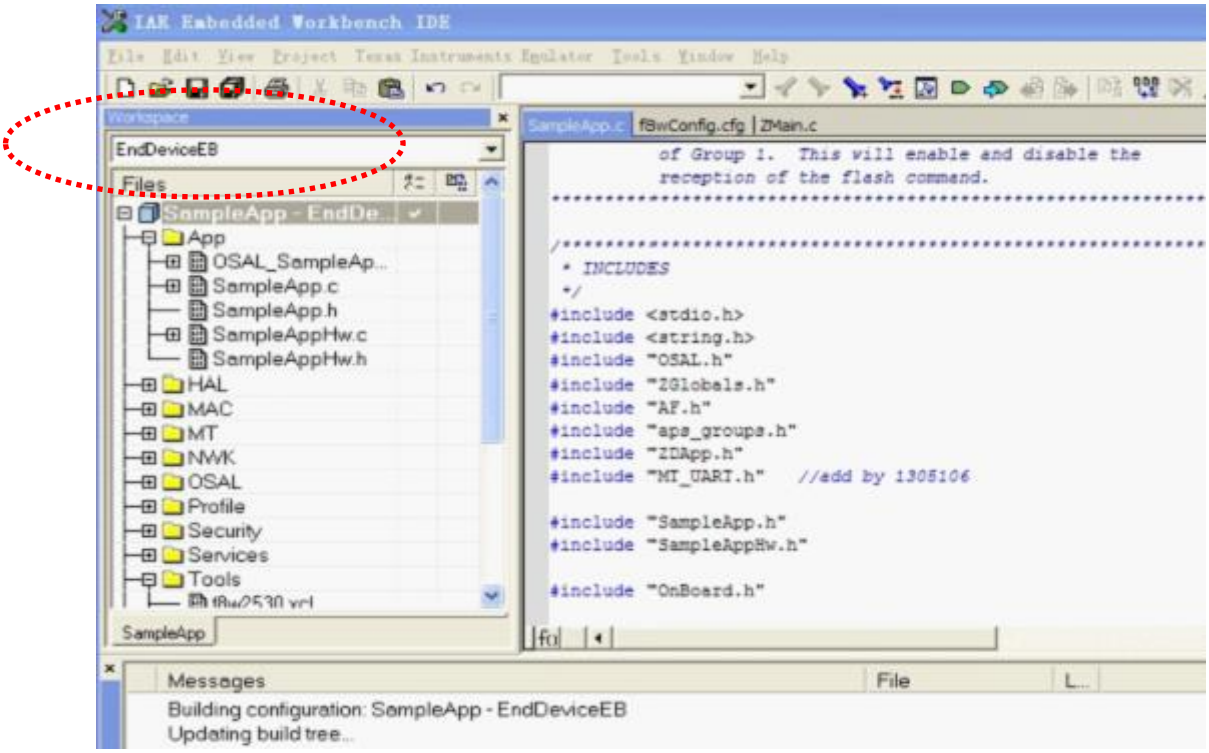


图 4.2 EndDeviceEB 工程

4）启动设备测试，首先启动协调器模块，建立网络时 LED1 闪烁，成功后 LED1 点亮停止闪烁，再启动节点端 ZigBee 模块，入网成功后 LED1 点亮停止闪烁，网络组建成功后，将 PC 机串口线连接到 ZigBee 协调器模块对应的串口上，打开串口终端，设置波特率为 38400、8 位、无奇偶校验、无硬件流模式，即可在超级终端上看到终端节点发送过来的

“~HELLO!~”字符串。

实验截图：

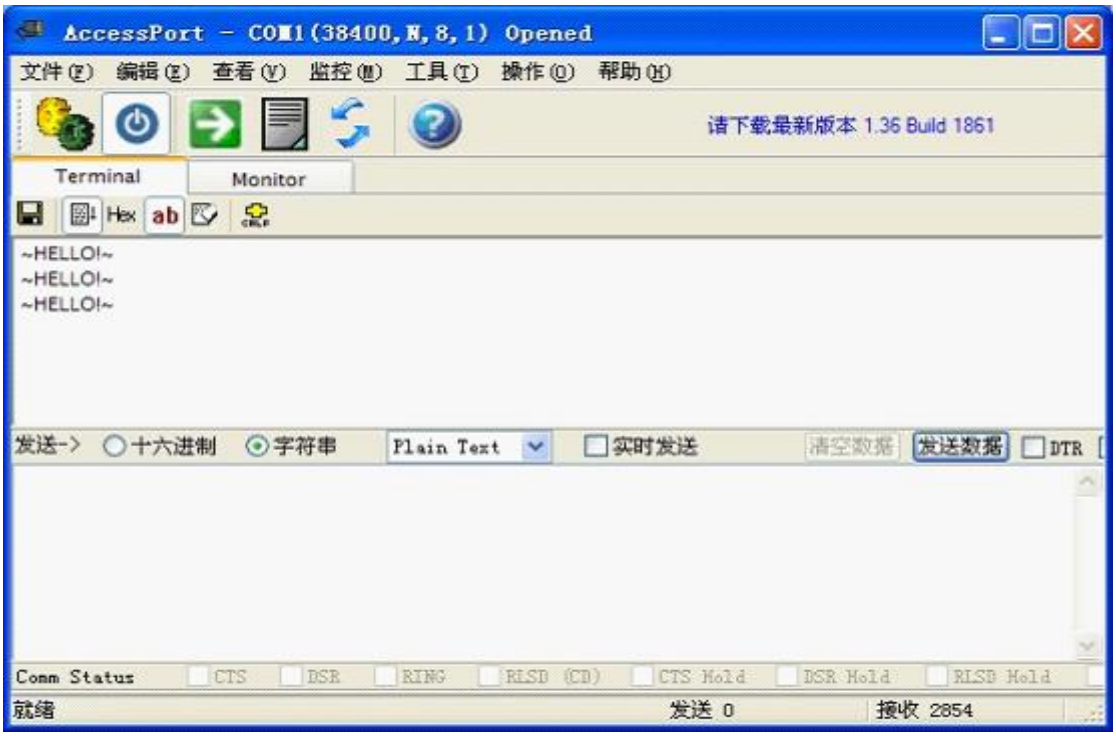


图 4.3 串口终端显示

备注：如果多套实验设备同时在运行此工程实验(局域网中存在多个相同工程编译出来运行的协调器模块)，为避免相同工程的 ZigBee 网络间的组网冲突，需要用户手动更改本工程下的 Tools 目录下的 f8wConfig.cfg 文件，将其中默认的 DZDAPP_CONFIG_PAN_ID=0xFFFF 宏 更改为唯一的特定值(0-0x3FFF 之间)，重新编译下载相应工程，运行。这样可以避免各个 ZigBee 网络(协调器)的冲突。

实验五. 基于 Z-Stack 的串口控制 LED 实验

1. 实验环境

- 硬件：ZigBee(CC2530)模块(两个)，ZigBee 下载调试板，USB 仿真器，PC 机。
- 软件：IAR Embedded Workbench for MCS-51　ZStack-2.3.0-1.4.0 协议栈。

2. 实验内容

- 学习 TI ZStack2007 协议栈内容，掌握 CC2530 模块无线组网原理及过程。掌握 ZStack 协议中关于串口 MT 层的处理流程。学习协议栈中关于串口的基本设置和操作。
- 使用 IAR 开发环境设计程序，ZStack-2.3.0-1.4.0 协议栈源码例程 SampleApp 工程基础上，实现无线组网及通讯。即协调器自动组网，终端节点自动入网，并设计上位机串口控制程序，实现上位机 PC 串口对 ZigBee 模块的控制，如 LED 的控制等。

3. 实验原理

3.1 ZigBee(CC2530)模块 LED 硬件接口

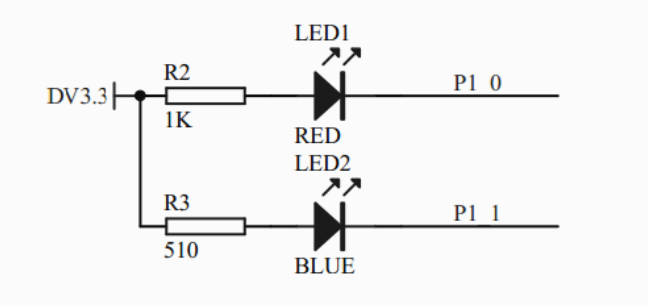


图 3.1.1 LED 硬件接口

ZigBee(CC2530)模块硬件上设计有 2 个 LED 灯，用来编程调试使用。分别连接 CC2530 的 P1_0、P、1_1 两个 IO 引脚。从原理图上可以看出，2 个 LED 灯共阳极，当 P1_0、P1_1 引脚为低电平时候，LED 灯点亮。

系统框图如下：

图 3.1.2 系统框图

本实验实现上位机通过串口控制命令，发送数据到 ZigBee 协调器节点，协调器通过无线网络控制节点端 LED 灯的开关状态。

3.2 SampleApp 实验分析

SampleApp 实验是协议栈自带的 ZigBee 无线网络自启动(组网)样例，该实验实现的功能主要是协调器自启动(组网)，节点设备自动入网。之后两者建立无线通讯，数据的发送主要有 2 种方式，一种为周期定时发送信息(本次实验采用该方法测试)，另一种需要通过按键事件触发发送 FLASH 信息。由于实验配套 ZigBee 模块硬件上与 TI 公司的 ZigBee 样板有差异，因此本次实验没有采用按键触发方式。

接下来我们分析发送 periodic 信息流程(发送按键事件 flash 流程略)

Periodic 消息是通过系统定时器开启并定时广播到 group1 出去的,因此在 SampleApp_ProcessEvent 事件处理函数中有如下定时器代码：

```
            case ZDO_STATE_CHANGE:
            SampleApp_NwkState = (devStates_t)(MSGpkt->hdr.status);
            if ( (SampleApp_NwkState == DEV_ZB_COORD)
                || (SampleApp_NwkState == DEV_ROUTER)
                || (SampleApp_NwkState == DEV_END_DEVICE) )
                {
                  // Start sending the periodic message in a regular interval.
                  HalLedSet(HAL_LED_1, HAL_LED_MODE_ON);
                  if(SampleApp_NwkState != DEV_ZB_COORD)
                    {
                        SampleApp_ConnectToParent();//端点获取自己地址发送给协调器
                    }
                  osal_start_timerEx( SampleApp_TaskID,
SAMPLEAPP_SEND_PERIODIC_MSG_EVT,
SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT );
                }
                else
                {
                    // Device is no longer in the network
                }
              break;
```

当设备加入到网络后，其状态就会变化，对所有任务触发 ZDO_STATE_CHANGE 事件，开启一个定时器。当定时时间一到，就触发广播 periodic 消息事件，触发事件 SAMPLEAPP_SEND_PERIODIC_MSG_ EVT,相应任务为 SampleApp_TaskID,于是再次调用 SampleApp_ProcessEvent()处理 SAMPLEAPP_SEND _PERIODIC_MSG_EVT 事件，该事件处理函数调用 SampleApp_SendPeriodicMessage()来发送周期信息。

```
if ( events & SAMPLEAPP_SEND_PERIODIC_MSG_EVT )
    {
      SampleApp_SendPeriodicMessage();      // Send the periodic message
      // Setup to send message again in normal period (+ a little jitter)
      osal_start_timerEx( SampleApp_TaskID, SAMPLEAPP_SEND_PERIODIC_MSG_EVT,
          (SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT + (osal_rand() & 0x00FF)) );
      return (events ^ SAMPLEAPP_SEND_PERIODIC_MSG_EVT);        // return unprocessed events
```

```
    }
```

♦ **MT 层串口通讯**

协议栈将串口通讯部分放到了 MT 层的 MT 任务中去处理了，因此我们在使用串口通讯的时候要在编译工程（通常是协调器工程）时候在编译选项中加入 MT 层相关任务的支持：MT_TASK,ZTOOL_P1 或 ZAPP_P1。

关于串口控制 LED 关键代码分析

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

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

              case SPI_INCOMING_ZAPP_DATA:
                SampleApp_ProcessMTMessage(MSGpkt);
                MT_UartAppFlowControl (MT_UART_ZAPP_RX_READY);
                break;

              // Received whenever the device changes state in the network
              case ZDO_STATE_CHANGE:
                SampleApp_NwkState = (devStates_t)(MSGpkt->hdr.status);
                if ( (SampleApp_NwkState == DEV_ZB_COORD)
                    || (SampleApp_NwkState == DEV_ROUTER)
                    || (SampleApp_NwkState == DEV_END_DEVICE) )
                {
                  // Start sending the periodic message in a regular interval.
                  HalLedSet(HAL_LED_1, HAL_LED_MODE_ON);
```

```
                    if(SampleApp_NwkState != DEV_ZB_COORD)
                      {
                          SampleApp_ConnectToParent();//端点获取自己地址发送给协调器
                      }
                      osal_start_timerEx( SampleApp_TaskID,
          SAMPLEAPP_SEND_PERIODIC_MSG_EVT,
          SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT );
                    }
                    else
                    {
                      // Device is no longer in the network
                    }
                    break;

                  default:
                    break;
                }
                osal_msg_deallocate( (uint8 *)MSGpkt );         // Release the memory
                MSGpkt = (afIncomingMSGPacket_t *)osal_msg_receive( SampleApp_TaskID );     // Next - if
one is available
          }

          return (events ^ SYS_EVENT_MSG);       // return unprocessed events
        }

        // Send a message out - This event is generated by a timer
        //   (setup in SampleApp_Init()).
        if ( events & SAMPLEAPP_SEND_PERIODIC_MSG_EVT )
        {
          SampleApp_SendPeriodicMessage();      // Send the periodic message
          // Setup to send message again in normal period (+ a little jitter)
          osal_start_timerEx( SampleApp_TaskID, SAMPLEAPP_SEND_PERIODIC_MSG_EVT,
              (SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT + (osal_rand() & 0x00FF)) );
          return (events ^ SAMPLEAPP_SEND_PERIODIC_MSG_EVT);        // return unprocessed events
        }

        return 0;     // Discard unknown events
}
```

SampleApp_ProcessEvent() 函数为应用层事件处理函数，从上面的代码可知，当应用层接收到串口数据（即发生 SPI_INCOMING_ZAPP_DATA:事件）时，会调用 SampleApp_ProcessMTMessage(MSGpkt); 串口处理函数，当接收到网络数据（即发生 AF_INCOMING_MSG_CMD 事件）时，会调用 SampleApp_MessageMSGCB(MSGpkt);处理函数，现在来分析这两个函数。

```
void SampleApp_ProcessMTMessage(afIncomingMSGPacket_t *msg)
{
```

```
    //byte len = msg->hdr.status;
    const char *msgPtr = ((const char *)msg+2);
    //HalUARTWrite ( 0, msgPtr, len);
    uint8 status;

    if(strncmp(msgPtr, "on", 2) == 0){
            status = 0x01;
            HalUARTWrite ( 0, "\rset led on\r", 12);
    }
    else if(strncmp(msgPtr, "off", 3) == 0){
            status = 0x00;
            HalUARTWrite ( 0, "\rset led off\r", 13);
    }
    /*发送消息给终端节点*/
    if ( AF_DataRequest( &SampleApp_Addr, &SampleApp_epDesc,
                         SAMPLEAPP_LEDCTL_CLUSTERID,
                         1,
                         &status,
                         &SampleApp_TransID,
                         AF_DISCV_ROUTE,
                         AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
    {
    }
    else
    {
      // Error occurred in request to send.
    }

}
```

这个函数是协调器要完成的工作，当串口接收到字符串“on”时会向串口回发“set led on”，并向终端节点发送 0x01，当串口接收到字符串“off”时会向串口回发“set led off”,并向终端节点发送 0x00。

```
void SampleApp_MessageMSGCB( afIncomingMSGPacket_t *pkt )
{
    uint16 flashTime;

    switch ( pkt->clusterId )
    {
        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;
```

```
            case SAMPLEAPP_LEDCTL_CLUSTERID:
              SetLedStatus(pkt->cmd.Data[0]);
              break;
            case SAMPLEAPP_CONNECTREQ_CLUSTERID:
              SampleApp_ConnectReqProcess((uint8*)pkt->cmd.Data);
        }
}
```

协调器处理 SAMPLEAPP_CONNECTREQ_CLUSTERID 这个消息类型，此消息携带着终端节点的地址，协调器通过这个消息获取终端节点的地址。

终端节点处理 SAMPLEAPP_LEDCTL_CLUSTERID，当终端节点收到协调 SAMPLEAPP_LEDCTL_CLUSTERID 簇 ID 发送过来的一字节命令（保存在 cmd.Data[0]）时，会根据这个命令来设置 LED 状态。

更详细的处理流程，具体见工程源代码。

4. 实验步骤

1）使用 ZigBee Debuger USB 仿真器连接 PC 机和 ZigBee(CC2530)模块，打开 ZigBee 模块开关供电。

2）打开产品光盘资料里 Compnents\ZigBee\TI\exp\zigbee\基于 ZStack 的上位机串口控制 LED 实验\Projects\zstack\Samples\SampleApp\CC2530DB 里的工程。

选择 CoordinatorEB 工程，编译下载到 ZigBee COORDINATOR 模块中

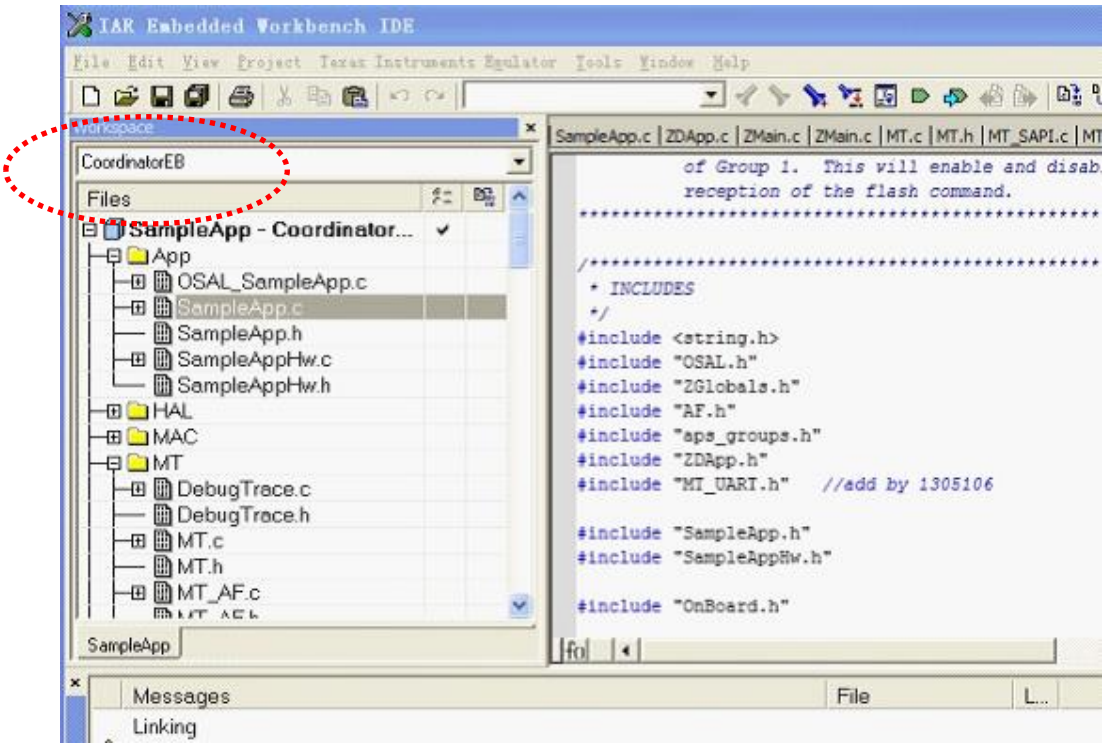


图 4.1 CoordinatorEB 工程

3）选择 EndDeviceEB 工程，编译下载到终端节点

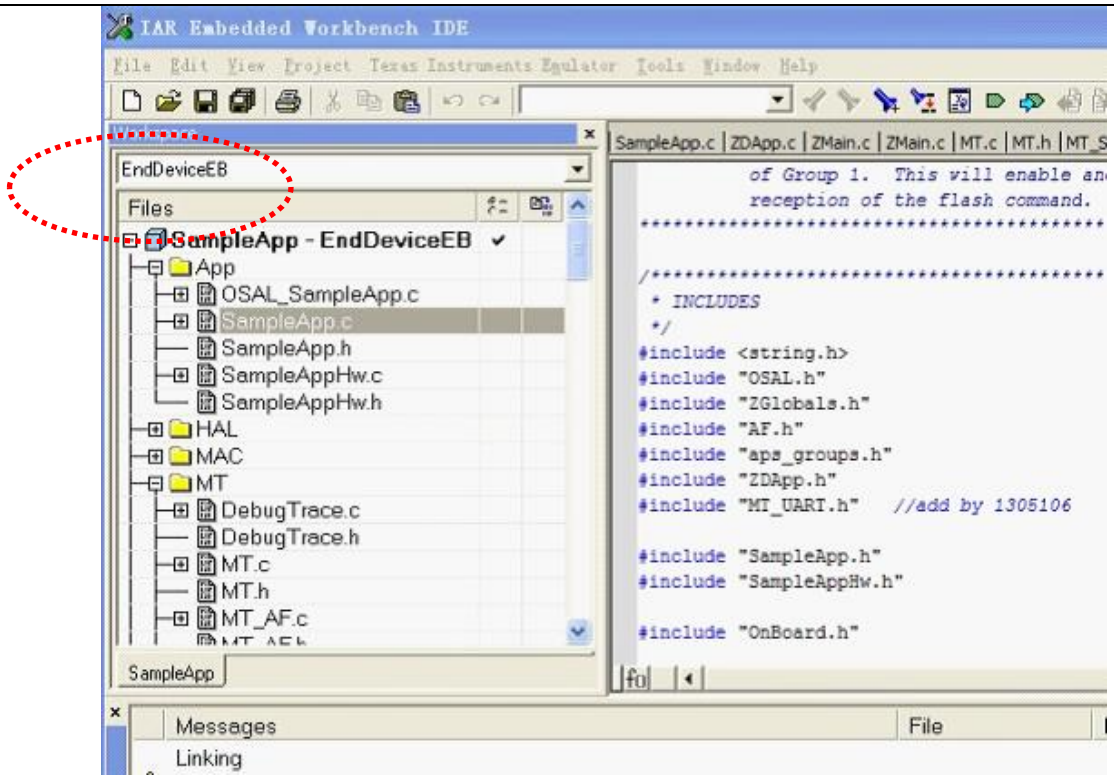


图 4.2 EndDeviceEB 工程

4）启动设备测试，首先启动协调器模块，建立网络时 LED1 闪烁，成功后 LED1 点亮停止闪烁，再启动节点端 ZigBee 模块，入网成功后 LED1 点亮停止闪烁，网络组建成功后，将 PC 机串口线连接到 ZigBee 协调器模块对应的串口上，打开串口终端，设置波特率为 38400、8 位、无奇偶校验、无硬件流模式。即可在串口终端中输入“on”或者“off”来发送串口控制命令至协调器模块，协调器通过串口接收到命令后，无线控制远程节点模块上 LED 灯开关状态。

- **实验截图**

图 4.3 串口终端显示

备注：如果多套实验设备同时在运行此工程实验(局域网中存在多个相同工程编译出来运行的协调器模块)，为避免相同工程的 ZigBee 网络间的组网冲突，需要用户手动更改本工程下的 Tools 目录下的 f8wConfig.cfg 文件，将其中默认的 DZDAPP_CONFIG_PAN_ID=0xFFFF 宏 更改为唯一的特定值(0-0x3FFF 之间)，重新编译下载相应工程，运行。这样可以避免各个 ZigBee 网络(协调器)的冲突。

若出现程序编译出现……has no prototype 作如下设置 project-->option-->c/c++compiler 去掉 Require prototype 前的勾。

实验六. 无线温度检测实验

1. 实验环境

- 硬件：ZigBee(CC2530)模块(至少两个)，ZigBee 下载调试板，USB 仿真器，PC 机。
- 软件：IAR Embedded Workbench for MCS-51　ZStack-2.3.0-1.4.0 协议栈。

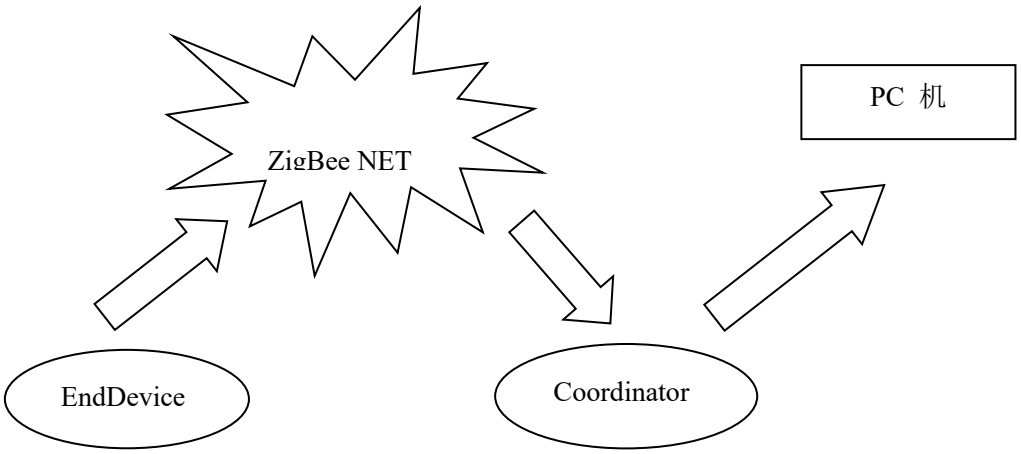
2. 实验内容

- 学习 TI ZStack2007 协议栈内容，掌握 CC2530 模块数据传输的实现过程。
- 学习协议栈中关于串口的基本设置和操作。

3. 实验原理

3.1 系统流程图

协调器分立 ZigBee 无线网络，终端节点自动加入该网络中，然后终端节点周期性的采集温度数据并将其发送给协调器，协调器接收到温度数据后，通过串口将其输出到 PC 机。如图 3.3.1 所示。



3.3.1 无线温度检测实验效果图

- **协调器流程图**

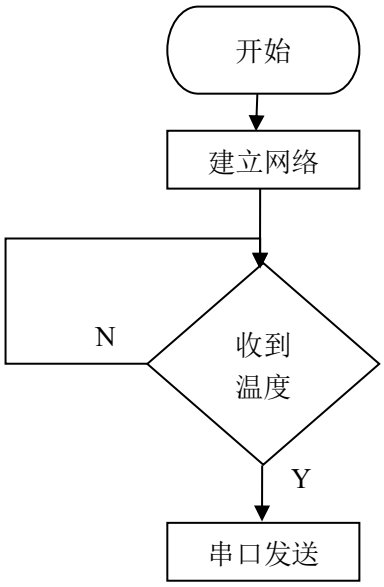


图 3.3.2 无线温度检测实验协调器流程图

- **终端节点流程图**

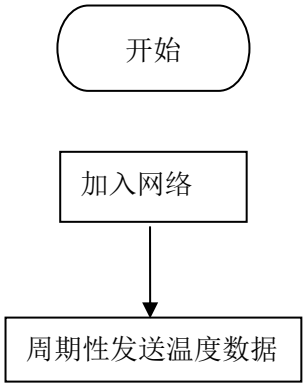


图 3.3.3 无线温度检测实验终端节点流程图

3.2 关键代码分析

对于协调器而言，只需要将收集到的温度数据通过串口发送到 PC 机即可；对于终端节点而言，需要周期性地采集温度数据，采集温度数据可以通过读取温度传感器的数据得到。

协调器编程。

温度数据包结构设计。

数据包	数据头	温度数据十位	温度数个位	数据尾
长度、字节	1	1	1	1
默认值	‘&’	0	0	‘C’

该数据包结构体定义如下。

```
typedef union h
{
    uint8 TEMP[4];
    struct RFRXBUF
    {
       unsigned char Head;
       unsigned char value[2];
       unsigned char Tail;
    }BUF;
}TEMPRETURE;
```

使用一个共用体来表示整个数据包，里面有两个成员变量，一个是数组 TEMP，该数组有 4 元素；另一个是结构体，该结构体具体实现了数据包的数据头、温度数据、数据尾。结构体所占的存储空间也是 4 个字节。

- **协调器代码：**

```
void GenericApp_MessageMSGCB(afIncomingMSGPacket_t *pckt);
void GenericApp_SendTheMessage(void);
/*static void rxCB(uint8 port, uint8 event);
static void rxCB(uint8 port, uint8 event)
{
  HalUARTRead(0, uartbuf, 16);
  if(osal_memcmp(uartbuf,"www.wlwmaker.com",16))
        {
            HalUARTWrite(0, uartbuf,16);
        }
}*/
void GenericApp_Init(byte task_id)
{
  halUARTCfg_t uartConfig;

  GenericApp_TaskID          =task_id;
  GenericApp_TransID         =0;
  GenericApp_epDesc.endPoint=GENERICAPP_ENDPOINT;
  GenericApp_epDesc.task_id=&GenericApp_TaskID;
  GenericApp_epDesc.simpleDesc=(SimpleDescriptionFormat_t *)&GenericApp_SimpleDesc;
  GenericApp_epDesc.latencyReq=noLatencyReqs;
  afRegister(&GenericApp_epDesc);

  uartConfig.configured      =TRUE;
  uartConfig.baudRate        =HAL_UART_BR_115200;
  uartConfig.flowControl     =FALSE;
  uartConfig.callBackFunc        =NULL;
  HalUARTOpen(0,&uartConfig);
```

```
}
UINT16 GenericApp_ProcessEvent(byte tadk_id,UINT16 events)
{
  afIncomingMSGPacket_t *MSGpkt;
  if(events&SYS_EVENT_MSG)
       {
            MSGpkt=(afIncomingMSGPacket_t *)osal_msg_receive(GenericApp_TaskID);
            while(MSGpkt)
                 {
                      switch(MSGpkt->hdr.event)
                           {
                                case AF_INCOMING_MSG_CMD:
                                     GenericApp_MessageMSGCB(MSGpkt);
                                     break;
                                     default:
                                          break;
                           }
                      osal_msg_deallocate((uint8 *) MSGpkt);
                      MSGpkt=(afIncomingMSGPacket_t *)osal_msg_receive(GenericApp_TaskID);
                 }
            return (events ^SYS_EVENT_MSG);

       }
  return 0;
}
void GenericApp_MessageMSGCB(afIncomingMSGPacket_t * pkt)
{
  unsigned char buffer[2]={0x0A,0x0D};
  TEMPRETURE    tempreture;
  switch(pkt->clusterId)
       {
            case GENERICAPP_CLUSTERID:
                 osal_memcpy(&tempreture,pkt->cmd.Data,sizeof(tempreture));
                 HalUARTWrite(0,(uint8*)&tempreture,sizeof(tempreture));
                 HalUARTWrite(0,buffer,2);
                 break;
       }
}
```

- **终端节点编程**

```
//读取温度

int8 readTemp(void)
{
    static uint16 reference_voltage;
```

```
    static uint8 bCalibrate=TRUE;
    uint16 value;
    int8 temp;

    ATEST=0x01;
    TR0|=0x01;
    ADCIF=0;
    ADCCON3=(HAL_ADC_REF_115V|HAL_ADC_DEC_256|HAL_ADC_CHN_TEMP);
    while(!ADCIF);
  ADCIF=0;
  value=ADCL;
  value|=((uint16)ADCH)<<8;
  value>>=4;
  if(bCalibrate)
      {
          reference_voltage=value;
          bCalibrate=FALSE;
      }
  temp=22+((value-reference_voltage)/4);
  return 22;

}

//终端节点事件处理与无线数据发送
UINT16 GenericApp_ProcessEvent(byte tadk_id,UINT16 events)
{
  afIncomingMSGPacket_t *MSGpkt;
  if(events&SYS_EVENT_MSG)
      {
          MSGpkt=(afIncomingMSGPacket_t *)osal_msg_receive(GenericApp_TaskID);
          while(MSGpkt)
              {
                  switch(MSGpkt->hdr.event)
                      {
                          case ZDO_STATE_CHANGE:
                              GenericApp_NwkState=(devStates_t)(MSGpkt->hdr.status);
                              if(GenericApp_NwkState==DEV_END_DEVICE)
                                  {
                                      //GenericApp_SendTheMessage();

osal_set_event(GenericApp_TaskID,SEND_DATA_EVENT);
                                  }
                              default:
                                  break;
                      }
                  osal_msg_deallocate((uint8 *) MSGpkt);
```

```
                    MSGpkt=(afIncomingMSGPacket_t *)osal_msg_receive(GenericApp_TaskID);
                }
            return (events ^SYS_EVENT_MSG);

        }
    if(events&SEND_DATA_EVENT)
        {
            GenericApp_SendTheMessage();
            osal_start_timerEx(GenericApp_TaskID,SEND_DATA_EVENT,1000);
            return (events^SEND_DATA_EVENT);
        }
    return 0;
}


void GenericApp_SendTheMessage(void)
{
    //unsigned char theMessageData[10]="EndDevice";
    int8 tvalue;
    TEMPRETURE    tempreture;
    tempreture.BUF.Head='&';
    tvalue=readTemp();
    tempreture.BUF.value[0]=tvalue/10+'0';
    tempreture.BUF.value[1]=tvalue%10+'0';
    tempreture.BUF.Tail='C';


    afAddrType_t my_DstAddr;
    my_DstAddr.addrMode=(afAddrMode_t)Addr16Bit;
    my_DstAddr.endPoint=GENERICAPP_ENDPOINT;
    my_DstAddr.addr.shortAddr=0x0000;
    AF_DataRequest(&my_DstAddr, &GenericApp_epDesc, GENERICAPP_CLUSTERID,
                    sizeof(tempreture),
                    (uint8 *)&tempreture,
                    &GenericApp_TransID,
                    AF_DISCV_ROUTE,
                    AF_DEFAULT_RADIUS);
    HalLedBlink(HAL_LED_2,0,50,500);
}
```

4. 实验步骤

1）打开 ZigBee2530 部分\exp\zigbee\无线温度检测实验\Projects\zstack\Samples\无线温度检测实验。

2）将程序下载到 cc2530 开发板，打开串口调试助手，波特率设为 115200，打开协调

器、终端节点电源，用手放在终端节点 cc2530 单片机上（这样片内集成的温度传感器就可以感应到温度变化），无线温度检测实验测试效果如图所示.。

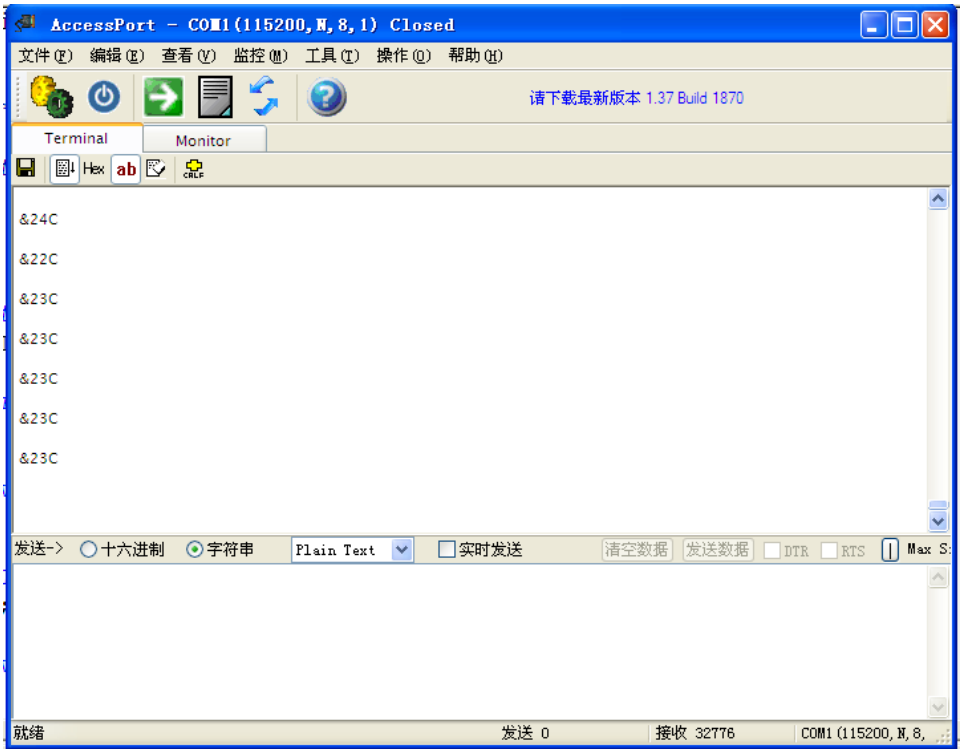


图 4.1 温度数据

实验七. 无线透传实验

1. 实验环境

- 硬件：ZigBee(CC2530)模块(两个)，ZigBee 下载调试板，USB 仿真器，串口调试工具、PC 机。
- 软件：IAR Embedded Workbench for MCS-51　ZStack-2.3.0-1.4.0 协议栈。

2. 实验内容

- 学习 ZigBee（CC2530）串口操作。
- 掌握数据收发原理。

3. 实验原理

3.1 串口初始化

```
halUARTCfg_t uartConfig;
uartConfig.configured                = TRUE;                    // 2x30 don't care - see uart driver.
uartConfig.baudRate                  = SERIAL_APP_BAUD;
uartConfig.flowControl               = FALSE;
uartConfig.flowControlThreshold = SERIAL_APP_THRESH; // 2x30 don't care - see uart driver.
uartConfig.rx.maxBufSize             = SERIAL_APP_RX_SZ;    // 2x30 don't care - see uart driver.
uartConfig.tx.maxBufSize             = SERIAL_APP_TX_SZ;    // 2x30 don't care - see uart driver.
uartConfig.idleTimeout               = SERIAL_APP_IDLE;     // 2x30 don't care - see uart driver.
uartConfig.intEnable                 = TRUE;                    // 2x30 don't care - see uart driver.
uartConfig.callBackFunc              = SerialApp_CallBack;
HalUARTOpen (SERIAL_APP_PORT, &uartConfig);
```

3.2 串口回调函数

```
static void SerialApp_CallBack(uint8 port, uint8 event)
{
  (void)port;

  if ((event & (HAL_UART_RX_FULL | HAL_UART_RX_ABOUT_FULL |
HAL_UART_RX_TIMEOUT)) &&
#if SERIAL_APP_LOOPBACK
      (SerialApp_TxLen < SERIAL_APP_TX_MAX))
#else
```

```
        !SerialApp_TxLen)
#endif
  {
    SerialApp_Send();
  }
}
```

3.3 串口数据发送函数

```
static void SerialApp_Send(void)
{
#if SERIAL_APP_LOOPBACK
    if (SerialApp_TxLen < SERIAL_APP_TX_MAX)
    {
        SerialApp_TxLen += HalUARTRead(SERIAL_APP_PORT,
SerialApp_TxBuf+SerialApp_TxLen+1,
                                       SERIAL_APP_TX_MAX-
SerialApp_TxLen);
    }

    if (SerialApp_TxLen)
    {
      (void)SerialApp_TxAddr;
      if (HalUARTWrite(SERIAL_APP_PORT, SerialApp_TxBuf+1, SerialApp_TxLen))
      {
        SerialApp_TxLen = 0;
      }
      else
      {
        osal_set_event(SerialApp_TaskID, SERIALAPP_SEND_EVT);
      }
    }
#else
    if (!SerialApp_TxLen &&
        (SerialApp_TxLen = HalUARTRead(SERIAL_APP_PORT, SerialApp_TxBuf+1,
SERIAL_APP_TX_MAX)))
    {
      // Pre-pend sequence number to the Tx message.
      SerialApp_TxBuf[0] = ++SerialApp_TxSeq;
    }

    if (SerialApp_TxLen)
    {
      if (afStatus_SUCCESS != AF_DataRequest(&SerialApp_TxAddr,
                                             (endPointDesc_t *)&SerialApp_epDesc,
```

```
                                                    SERIALAPP_CLUSTERID1,
                                                    SerialApp_TxLen+1, SerialApp_TxBuf,
                                                    &SerialApp_MsgID, 0, AF_DEFAULT_RADIUS))
        {
          osal_set_event(SerialApp_TaskID, SERIALAPP_SEND_EVT);
        }
      }
#endif
}
```

3.4 数据接收与消息处理

```
void SerialApp_ProcessMSGCmd( afIncomingMSGPacket_t *pkt )
{
  uint8 stat;
  uint8 seqnb;
  uint8 delay;

  switch ( pkt->clusterId )
  {
  // A message with a serial data block to be transmitted on the serial port.
  case SERIALAPP_CLUSTERID1:
    // Store the address for sending and retrying.
    osal_memcpy(&SerialApp_RxAddr, &(pkt->srcAddr), sizeof( afAddrType_t ));

    seqnb = pkt->cmd.Data[0];

    // Keep message if not a repeat packet
    if ( (seqnb > SerialApp_RxSeq) ||                    // Normal
         ((seqnb < 0x80 ) && ( SerialApp_RxSeq > 0x80)) ) // Wrap-around
    {
      // Transmit the data on the serial port.
      if ( HalUARTWrite( SERIAL_APP_PORT, pkt->cmd.Data+1, (pkt->cmd.DataLength-1) ) )
      {
        // Save for next incoming message
        SerialApp_RxSeq = seqnb;
        stat = OTA_SUCCESS;
      }
      else
      {
        stat = OTA_SER_BUSY;
      }
    }
    else
```

```
      {
          stat = OTA_DUP_MSG;
      }

      // Select approproiate OTA flow-control delay.
      delay = (stat == OTA_SER_BUSY) ? SERIALAPP_NAK_DELAY : SERIALAPP_ACK_DELAY;

      // Build & send OTA response message.
      SerialApp_RspBuf[0] = stat;
      SerialApp_RspBuf[1] = seqnb;
      SerialApp_RspBuf[2] = LO_UINT16( delay );
      SerialApp_RspBuf[3] = HI_UINT16( delay );
      osal_set_event( SerialApp_TaskID, SERIALAPP_RESP_EVT );
      osal_stop_timerEx(SerialApp_TaskID, SERIALAPP_RESP_EVT);
      break;

    // A response to a received serial data block.
    case SERIALAPP_CLUSTERID2:
      if ((pkt->cmd.Data[1] == SerialApp_TxSeq) &&
          ((pkt->cmd.Data[0] == OTA_SUCCESS) || (pkt->cmd.Data[0] == OTA_DUP_MSG)))
      {
        SerialApp_TxLen = 0;
        osal_stop_timerEx(SerialApp_TaskID, SERIALAPP_SEND_EVT);
      }
      else
      {
        // Re-start timeout according to delay sent from other device.
        delay = BUILD_UINT16( pkt->cmd.Data[2], pkt->cmd.Data[3] );
        osal_start_timerEx( SerialApp_TaskID, SERIALAPP_SEND_EVT, delay );
      }
      break;

      case SERIALAPP_CONNECTREQ_CLUSTER:
        SerialApp_ConnectReqProcess((uint8*)pkt->cmd.Data);

      case SERIALAPP_CONNECTRSP_CLUSTER:
        SerialApp_DeviceConnectRsp((uint8*)pkt->cmd.Data);

      default:
        break;
    }
}
```

3.5 事件处理

```
UINT16 SerialApp_ProcessEvent( uint8 task_id, UINT16 events )
{
  (void)task_id;  // Intentionally unreferenced parameter

  if ( events & SYS_EVENT_MSG )
  {
    afIncomingMSGPacket_t *MSGpkt;

    while ( (MSGpkt = (afIncomingMSGPacket_t *)osal_msg_receive( SerialApp_TaskID )) )
    {
      switch ( MSGpkt->hdr.event )
      {
      case AF_INCOMING_MSG_CMD:
        SerialApp_ProcessMSGCmd( MSGpkt );
        break;

      case ZDO_STATE_CHANGE:
        SampleApp_NwkState = (devStates_t)(MSGpkt->hdr.status);
        if ( (SampleApp_NwkState == DEV_ZB_COORD)
            || (SampleApp_NwkState == DEV_ROUTER)
            || (SampleApp_NwkState == DEV_END_DEVICE) )
        {
            // Start sending the periodic message in a regular interval.
            HalLedSet(HAL_LED_1, HAL_LED_MODE_ON);

            if(SampleApp_NwkState != DEV_ZB_COORD)
              SerialApp_DeviceConnect();                    //add by 1305106
        }
        else
        {
          // Device is no longer in the network
        }
        break;

      default:
        break;
      }

      osal_msg_deallocate( (uint8 *)MSGpkt );
    }

    return ( events ^ SYS_EVENT_MSG );
  }
```

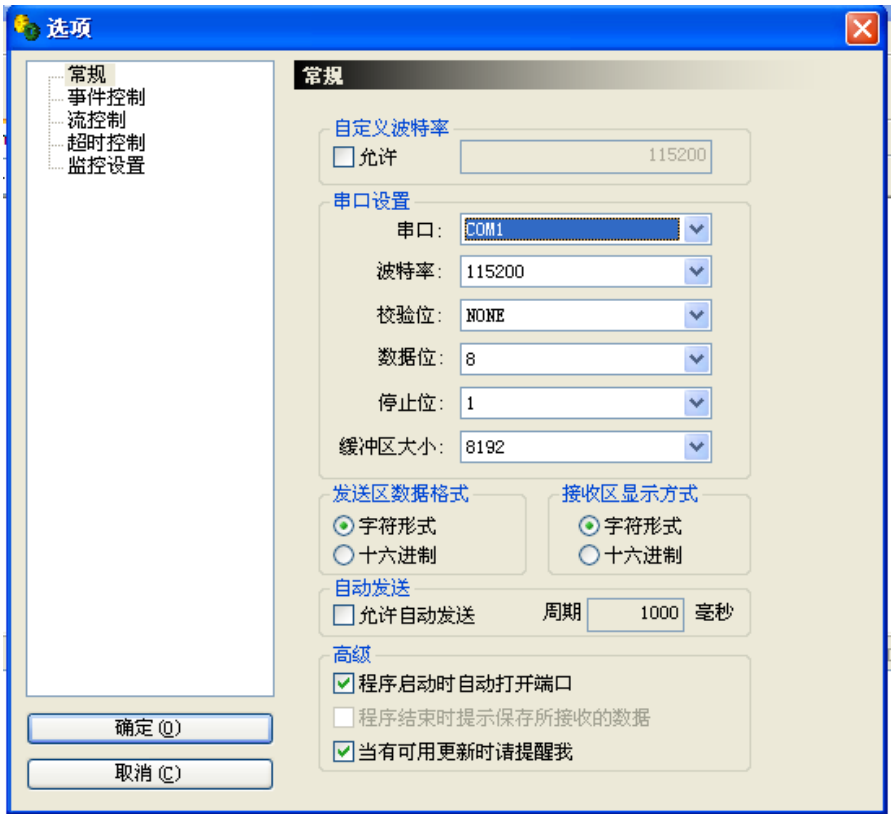
```
  if ( events & SERIALAPP_SEND_EVT )
  {
    SerialApp_Send();
    return ( events ^ SERIALAPP_SEND_EVT );
  }

  if ( events & SERIALAPP_RESP_EVT )
  {
    SerialApp_Resp();
    return ( events ^ SERIALAPP_RESP_EVT );
  }

  return ( 0 );  // Discard unknown events.
}
```

4. 实验步骤

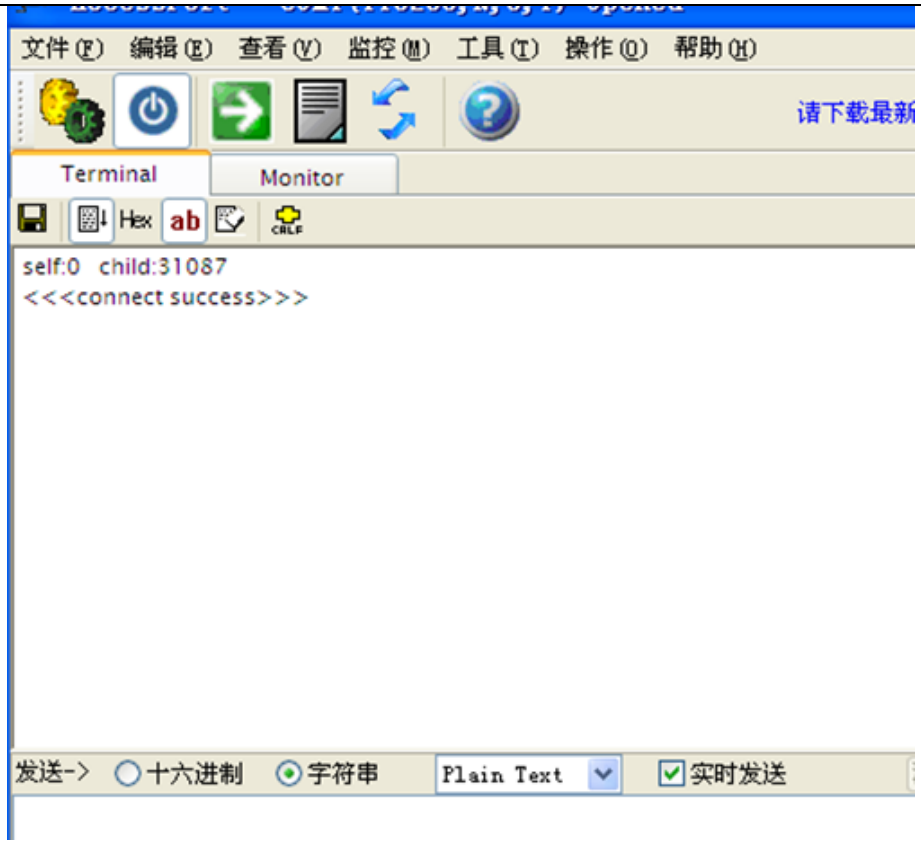
1）打开 cc2530 实验目录\exp\zigbee\基于 Z-Stack 协议栈的数据透传模型实验\.Projects\zstack\Utilities\SerialApp\CC2530DB

2）下载协调器与端点工程到两块开发板上。

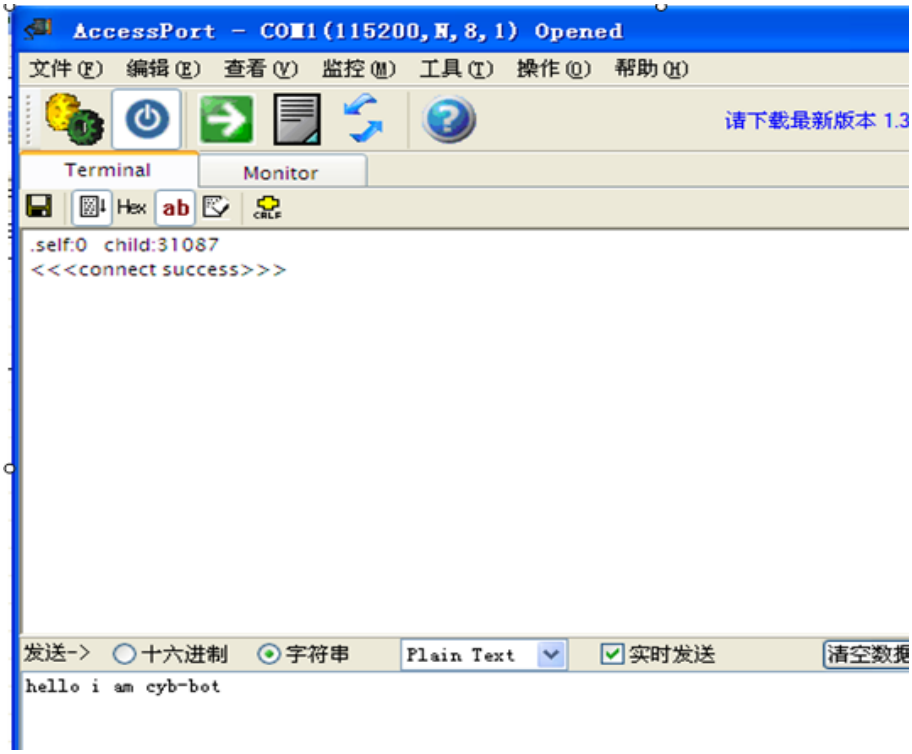
3）打开两个串口调试窗口，分别连到协调器与端点。串口配置如图所示，端口号根据实际情况进行选择。



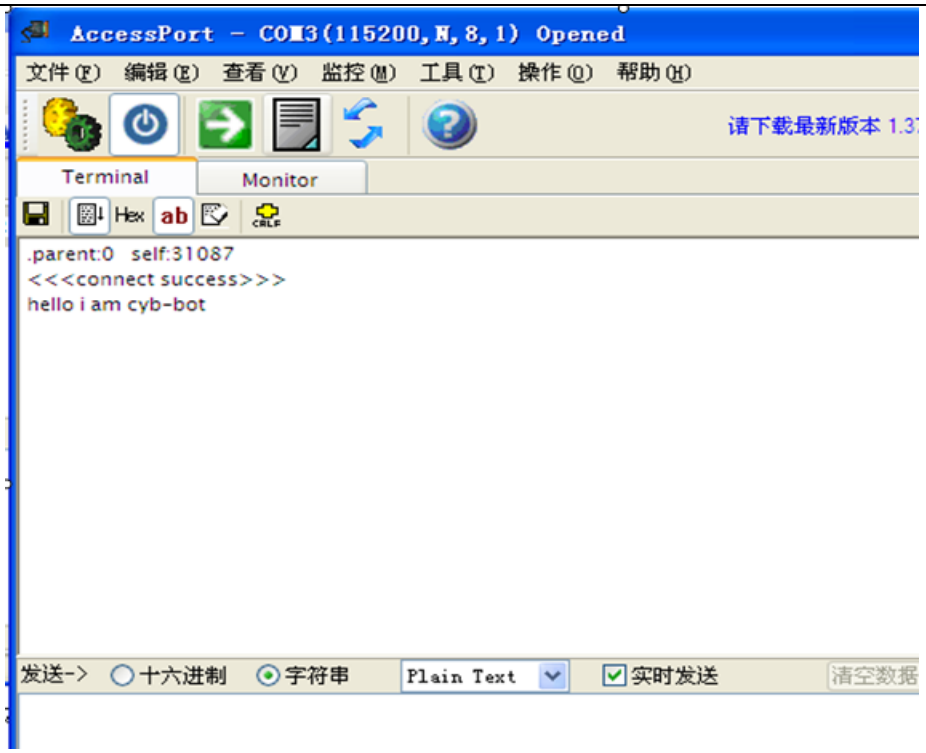
4）当协调器端显示下图所示时，表明连接成功。在端点端也是如此。



5）此时在协调器端输入数据，如图所示。



6）在端点端会实时的收到如下信息，如图所示。



7）反过来，也可以从端点端输入信息，从协调器端查看信息。

实验八. 无线传感网络演示实验

1. 实验环境

- 硬件：ZigBee(CC2530)模块，ZigBee 下载调试板，USB 仿真器，PC 机。
- 软件：IAR Embedded Workbench for MCS-51　ZStack-2.3.0-1.4.0 协议栈。

2. 实验内容

- 掌握 ZigBee（cc2530）与传感器节点的串口通信协议。
- 掌握传感器数据的采集与传输过程。
- 学习无线传感网络的搭建。

3. 实验原理

3.1 传感器说明

传感器名称	传感器类型编号	传感器输出数据说明
磁检测传感器	0x01	1-有磁场；0-无磁场
光照传感器	0x02	1-有光照；0-无光照
红外对射传感器	0x03	1-有障碍；0-无障碍
红外反射传感器	0x04	1-有障碍；0-无障碍
结露传感器	0x05	1-有结露；0-无结露
酒精传感器	0x06	1-有酒精；0-无酒精
人体检测传感器	0x07	1-有人；0-无人
三轴加速度传感器	0x08	XH, XL, YH, YL, ZH, ZL
声响检测传感器	0x09	1-有声音；0-无声音
温湿度传感器	0x0A	HH, HL, TH, TL
烟雾传感器	0x0B	1-有烟雾；0-无烟雾
振动检测传感器	0x0C	1-有振动；0-无振动
传感器扩展板	0xFF	用户自定义

表 4.1.1 传感器状态串口协议

3.2 串口设置

波特率 115200，数据位 8，停止位 1，无校验位

- **三轴加速度**

X 轴加速度值 = XH*256+XL

Y 轴加速度值 = YH*256+YL

Z 轴加速度值 = ZH*256+ZL

- **温湿度传感器**

湿度值 = (HH*256+HL) / 10，以%为单位。

温度值 = (TH*256+TL) / 10，以℃为单位。

3.3 传感器底层协议

传感器模块	发送	返回	意义
磁检测传感器	CC EE 01 NO 01 00 00 FF 查询是否有磁场	EE CC 01 NO 01 00 00 00 00 00 00 00 FF	无人
		EE CC 01 NO 01 00 00 00 00 00 01 00 00 FF	有人
光照传感器	CC EE 02 NO 01 00 00 FF 查询是否有光照	EE CC 02 NO 01 00 00 00 00 00 00 00 00 FF	无光照
		EE CC 02 NO 01 00 00 00 00 00 01 00 00 FF	有光照
红外对射传感器	CC EE 03 NO 01 00 00 FF 查询红外对射传感器是否有障碍	EE CC 03 NO 01 00 00 00 00 00 00 00 00 FF	无障碍
		EE CC 03 NO 01 00 00 00 00 00 01 00 00 FF	有障碍
红外反射传感器	CC EE 04 NO 01 00 00 FF 查询红外反射传感器是否有障碍	EE CC 04 NO 01 00 00 00 00 00 00 00 00 FF	无障碍
		EE CC 04 NO 01 00 00 00 00 00 01 00 00 FF	有障碍
结露传感器	CC EE 05 NO 01 00 00 FF 查询是否有结露	EE CC 05 NO 01 00 00 00 00 00 00 00 00 FF	无结露
		EE CC 05 NO 01 00 00 00 00 00 01 00 00 FF	有结露
酒精传感器	CC EE 06 NO 01 00 00 FF 查询是否检测到酒精	EE CC 06 NO 01 00 00 00 00 00 00 00 00 FF	无酒精
		EE CC 06 NO 01 00 00 00 00 00 01 00 00 FF	有酒精
人体检测传感器	CC EE 07 NO 01 00 00 FF 查询是否检测到人	EE CC 07 NO 01 00 00 00 00 00 00 00 00 FF	无人
		EE CC 07 NO 01 00 00 00 00 00 01 00 00 FF	有人
三轴加速度传感器	CC EE 08 NO 01 00 00 FF 查询 XYZ 轴加速度	EE CC 08 NO 01 XH XL YH YL ZH ZL 00 00 FF	XYZ 轴加速度
声响检测传感	CC EE 09 NO 01 00 00 FF	EE CC 09 NO 01 00 00 00 00 00	无声响

器	查询是否有声响	00 00 00 FF	
		EE CC 09 NO 01 00 00 00 00 00 01 00 00 FF	有声响
温湿度传感器	CC EE 0A NO 01 00 00 FF 查询湿度和温度	EE CC 0A NO 01 00 00 HH HL TH TL 00 00 FF	湿度和温度值
烟雾传感器	CC EE 0B NO 01 00 00 FF 查询是否检测到烟雾	EE CC 0B NO 01 00 00 00 00 00 00 00 00 FF	无烟雾
		EE CC 0B NO 01 00 00 00 00 00 01 00 00 FF	有烟雾
振动检测传感器	CC EE 0C NO 01 00 00 FF 查询是否检测到振动	EE CC 0C NO 01 00 00 00 00 00 00 00 00 FF	无振动
		EE CC 0C NO 01 00 00 00 00 00 01 00 00 FF	有振动

表 4.1.2 传感器串口通讯协议

3.4 协调器程序

```
void GenericApp_MessageMSGCB(afIncomingMSGPacket_t *pckt);
void GenericApp_SendTheMessage(void);
void GenericApp_Init(byte task_id)
{
  halUARTCfg_t uartConfig;

  GenericApp_TaskID           =task_id;
  GenericApp_TransID          =0;
  GenericApp_epDesc.endPoint=GENERICAPP_ENDPOINT;
  GenericApp_epDesc.task_id=&GenericApp_TaskID;
  GenericApp_epDesc.simpleDesc=(SimpleDescriptionFormat_t *)&GenericApp_SimpleDesc;
  GenericApp_epDesc.latencyReq=noLatencyReqs;
  afRegister(&GenericApp_epDesc);

  uartConfig.configured       =TRUE;
  uartConfig.baudRate         =HAL_UART_BR_115200;
  uartConfig.flowControl      =FALSE;
  uartConfig.callBackFunc           =NULL;
  HalUARTOpen(0,&uartConfig);

}
UINT16 GenericApp_ProcessEvent(byte tadk_id,UINT16 events)
{
  afIncomingMSGPacket_t *MSGpkt;
  if(events&SYS_EVENT_MSG)
      {
          MSGpkt=(afIncomingMSGPacket_t *)osal_msg_receive(GenericApp_TaskID);
          while(MSGpkt)
```

```
                {
                    switch(MSGpkt->hdr.event)
                        {
                            case ZDO_STATE_CHANGE:
                                GenericApp_NwkState=(devStates_t)(MSGpkt->hdr.status);
                               if(GenericApp_NwkState==DEV_ZB_COORD)
                              HalLedSet(HAL_LED_1, HAL_LED_MODE_ON);
                              case AF_INCOMING_MSG_CMD:
                                    GenericApp_MessageMSGCB(MSGpkt);
                                    break;
                                    default:
                                        break;
                        }
                    osal_msg_deallocate((uint8 *) MSGpkt);
                    MSGpkt=(afIncomingMSGPacket_t *)osal_msg_receive(GenericApp_TaskID);
                }
            return (events ^SYS_EVENT_MSG);
        }
   return 0;
}
void GenericApp_MessageMSGCB(afIncomingMSGPacket_t * pkt)
{
   unsigned char buffer[14];
   int i=0;
   switch(pkt->clusterId)
        {
            case GENERICAPP_CLUSTERID:
                osal_memcpy(buffer, pkt->cmd.Data, 14);
                                    uartbuf[0]=0xee;
                                    uartbuf[1]=0xcc;
                                    uartbuf[2]=0x00;
                                    uartbuf[3]=0x00;
                                    uartbuf[4]=0x00;
                                    uartbuf[5]=HI_UINT16(pkt->srcAddr.addr.shortAddr);
                                    uartbuf[6]=LO_UINT16(pkt->srcAddr.addr.shortAddr);
                                    uartbuf[7]=0x00;
                                    uartbuf[8]=0x00;
                                    uartbuf[9]=HI_UINT16(NLME_GetCoordShortAddr());
                                    uartbuf[10]=LO_UINT16(NLME_GetCoordShortAddr());
                                    uartbuf[11]=0x01;//state
                                    uartbuf[12]=0x0B;//chanel
                                    uartbuf[13]=pkt->endPoint;
                                    for(i=14;i<=26;i++)
                                    {
                                            uartbuf[i]=buffer[i-12];
                                    }
```

```
                                    HalUARTWrite(0,uartbuf,26);
                                    HalLedBlink(HAL_LED_2,0,50,500);
                    break;
            }
}
```

3.5 端点端程序

```
void GenericApp_MessageMSGCB(afIncomingMSGPacket_t *pckt);
void GenericApp_SendTheMessage(void);
static void rxCB(uint8 port, uint8 event);
static void rxCB(uint8 port, uint8 event)
{
  HalUARTRead(0, uartbuf, 14);
          osal_set_event(GenericApp_TaskID,SEND_DATA_EVENT);
}
void GenericApp_Init(byte task_id)
{
  GenericApp_TaskID          =task_id;
  GenericApp_NwkState  =DEV_INIT;
  GenericApp_TransID         =0;
  GenericApp_epDesc.endPoint=GENERICAPP_ENDPOINT;
  GenericApp_epDesc.task_id=&GenericApp_TaskID;
  GenericApp_epDesc.simpleDesc=(SimpleDescriptionFormat_t *)&GenericApp_SimpleDesc;
  GenericApp_epDesc.latencyReq=noLatencyReqs;
  afRegister(&GenericApp_epDesc);
  halUARTCfg_t uartConfig;
  uartConfig.configured        =TRUE;
  uartConfig.baudRate          =HAL_UART_BR_115200;
  uartConfig.flowControl       =FALSE;
  uartConfig.callBackFunc            =rxCB;
  HalUARTOpen(0,&uartConfig);

}
UINT16 GenericApp_ProcessEvent(byte tadk_id,UINT16 events)
{
  afIncomingMSGPacket_t *MSGpkt;
  if(events&SYS_EVENT_MSG)
      {
          MSGpkt=(afIncomingMSGPacket_t *)osal_msg_receive(GenericApp_TaskID);
          while(MSGpkt)
              {
                  switch(MSGpkt->hdr.event)
                      {
                          case ZDO_STATE_CHANGE:
```

```
                                    GenericApp_NwkState=(devStates_t)(MSGpkt->hdr.status);

  if((GenericApp_NwkState==DEV_END_DEVICE)||(GenericApp_NwkState==DEV_ROUTER))
{
    HalLedSet(HAL_LED_1, HAL_LED_MODE_ON);
    osal_set_event(GenericApp_TaskID,SEND_DATA_EVENT);
}
                        default:
                                    break;
                        }
                    osal_msg_deallocate((uint8 *) MSGpkt);
                    MSGpkt=(afIncomingMSGPacket_t *)osal_msg_receive(GenericApp_TaskID);
                }
            return (events ^SYS_EVENT_MSG);

        }
  if(events&SEND_DATA_EVENT)
        {
            GenericApp_SendTheMessage();
            osal_start_timerEx(GenericApp_TaskID,SEND_DATA_EVENT,1000);
            return (events^SEND_DATA_EVENT);
        }
  return 0;
}
void GenericApp_SendTheMessage(void)
{

  afAddrType_t my_DstAddr;
  my_DstAddr.addrMode=(afAddrMode_t)Addr16Bit;
  my_DstAddr.endPoint=GENERICAPP_ENDPOINT;
  my_DstAddr.addr.shortAddr=0x0000;

if(afStatus_SUCCESS!=AF_DataRequest(&my_DstAddr,
                                    &GenericApp_epDesc, GENERICAPP_CLUSTERID,
                                    14,
                                    uartbuf,
                                    &GenericApp_TransID,
                                    AF_DISCV_ROUTE,
                                    AF_DEFAULT_RADIUS))
        {
          osal_set_event(GenericApp_TaskID, SEND_DATA_EVENT);
        }
  HalLedBlink(HAL_LED_2,0,50,500);
}
```

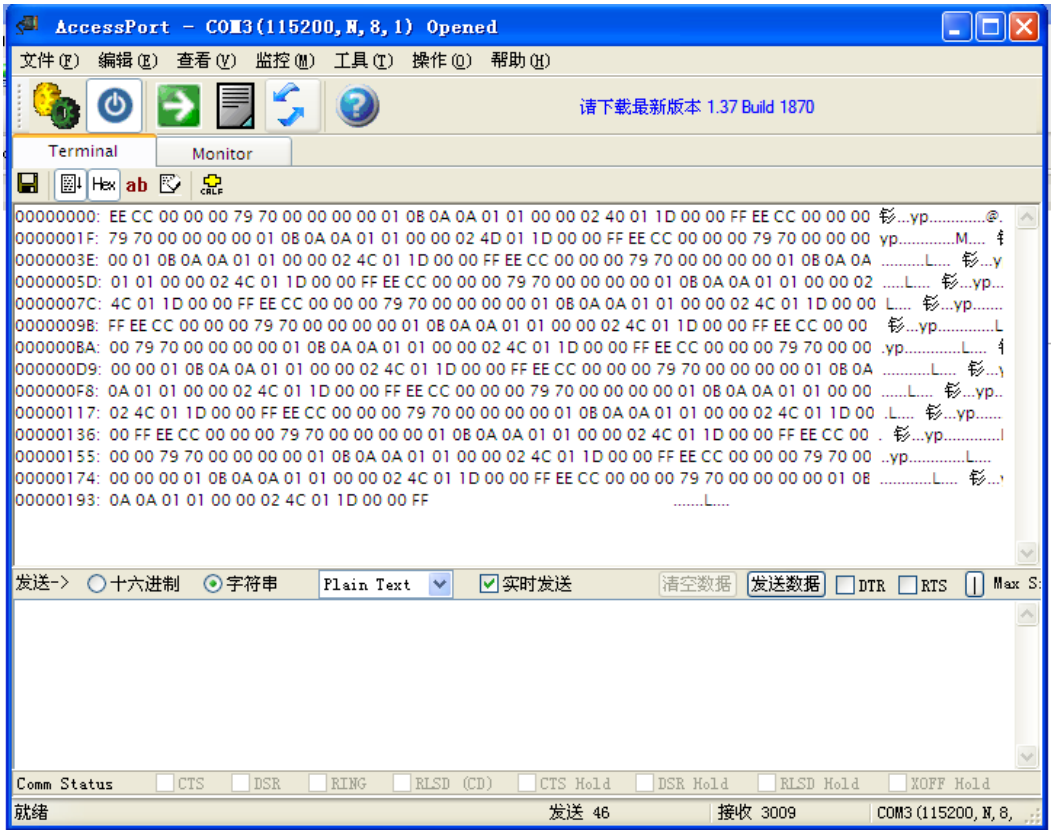
4. 实验步骤

1）打开\cc2530 实验目录\exp\zigbee\ticc2530_demo\CC2530D。

2）将 EndDeviceEB 工程下载到带有传感器的 zigbee 节点。将 CoordinatorEB 工程下载到根节点。

3）将根节点用串口与 pc 机连接。下面以温湿度传感器与光照传感器进行示范。

♦ **只打开温湿度传感器节点。**



对收到的数据进行解析：

说明	数据	字节数
包头	EE CC	2
ZigBee 网络标识	00	1
节点地址	00 0079 70	4
根节点地址	00 00 00 00	4
节点状态	01 已发现	1
节点通道	0B	1
通信端口	0A	1
传感器类型编号	0A	1
相同类型传感器 ID	01	1

节点命令序号	01	1
节点数据	00 00 02 4C 01 1D	6
保留字节	00 00	2
包尾	FF	1

湿度值 = (HH*256+HL) / 10，以%为单位。

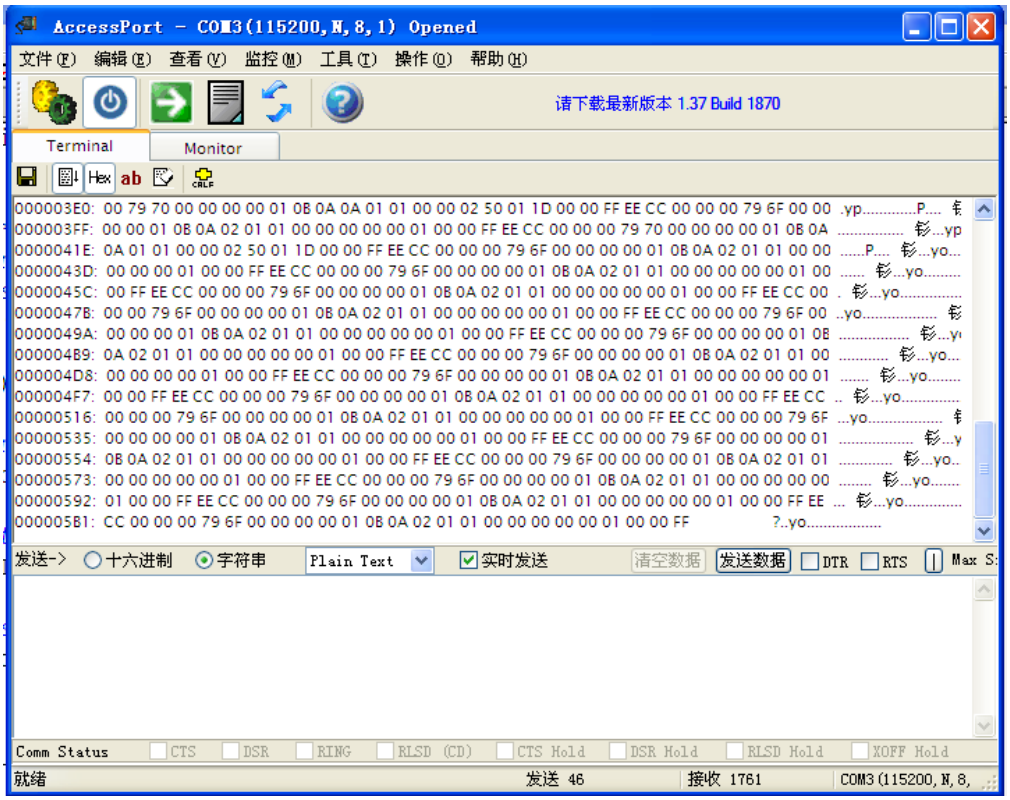
温度值 = (TH*256+TL) / 10，以℃为单位。

代入公式得：

当前湿度=58.8%

当前温度=28.5℃

- **只打开光照传感器节点。**



对收到的数据进行解析：

说明	数据	字节数
包头	EE CC	2
ZigBee 网络标识	00	1
节点地址	00 00 79 6F	4
根节点地址	00 00 00 00	4
节点状态	01 已发现	1
节点通道	0B	1
通信端口	0A	1
传感器类型编号	02	1

相同类型传感器 ID	01	1
节点命令序号	01	1
节点数据	00 00 00 00 00 01 有光照	6
保留字节	00 00	2
包尾	FF	1

更详细的协议说明，请用户参见《模块通讯协议 V2.6.pdf》文档。