# 传感器与传感网期末知识点整理

目录

[传感器与传感网期末知识点整理 1](#_Toc185780632)

[ZigBee的OSAL任务调度原理 2](#_Toc185780633)

[通过BDB实现Coordinator和Router的组网 3](#_Toc185780634)

[基于AF完成节点间的通讯 4](#_Toc185780635)

[点对点通讯 4](#_Toc185780636)

[广播通信 6](#_Toc185780637)

[组播通信 8](#_Toc185780638)

[AF层基础概念 9](#_Toc185780639)

[根据应用场景选择合适的传感器 9](#_Toc185780640)

[链路协议与传输协议 10](#_Toc185780641)

[链路协议 10](#_Toc185780642)

[传输协议 11](#_Toc185780643)

[ZigBee协议层次结构与OSI七层结构 11](#_Toc185780644)

[ZigBee协议层次结构 11](#_Toc185780645)

[OSI七层结构 12](#_Toc185780646)

[获取shortAddr和macAddr 13](#_Toc185780647)

[网络地址分配 16](#_Toc185780648)

[传感器数据收发实验 17](#_Toc185780649)

[CC2530外围电路 21](#_Toc185780650)

[传感器连接方式与DHT11 22](#_Toc185780651)

[数字信号传感器连接 22](#_Toc185780652)

[模拟信号传感器连接 23](#_Toc185780653)

[AD口 23](#_Toc185780654)

[DHT11 23](#_Toc185780655)

# ZigBee的OSAL任务调度原理

1. 系统初始化：在系统启动时，会调用osal\_init\_system()函数来初始化OSAL层，包括内存分配、消息队列、定时器和电源管理等。
2. 任务池的建立：通过osalInitTasks()函数为系统的每个任务分配内存空间，并初始化任务事件数组。

/\*任务结构体的定义\*/

typedef struct task\_t

{

bool occupy; // 是否已占用：true表示当前有任务；false表示当前没有任务

uint16\_t task; // 任务内容

uint16\_t expire; // 等待时间

} task\_t;

/\*任务池的定义\*/

#define TASK\_LIST\_SIZE 5

static task\_t taskList\_g[TASK\_LIST\_SIZE];

1. 任务初始化：在任务池建立，会调用每个任务的初始化函数，确保每个任务都正确初始化。

/\*任务池初始化\*/

static void taskListInit()

{

// 把所有任务都设置为未使用

for (uint16\_t i = 0; i < TASK\_LIST\_SIZE; i++)

taskList\_g[i].occupy = false;

}

/\*往任务池中添加任务\*/

static void addTask(uint16\_t expire, uint16\_t task)

{

for (uint16\_t i = 0; i < TASK\_LIST\_SIZE; i++)

{

if (taskList\_g[i].occupy)

continue;

taskList\_g[i].task = task;

taskList\_g[i].expire = expire;

taskList\_g[i].occupy = true;

break;

}

}

1. 事件处理：系统会不断检查任务事件数组，一旦发现有任务触发了事件，就会调用对应的事件处理函数来响应这些事件。

/\*轮询\*/

static void polling()

{

for (uint16\_t i = 0; i < (sizeof(taskList\_g) / sizeof(taskList\_g[0])); i++)

if (taskList\_g[i].occupy && (--taskList\_g[i].expire == 0))

{

// 执行任务

taskHandler(taskList\_g[i].task);

// 释放任务池中的空间

taskList\_g[i].occupy = false;

}

}

1. 系统启动：最后，通过osal\_start\_system()函数启动系统，系统将循环运行，不断处理任务事件。

# 通过BDB实现Coordinator和Router的组网

1. Network Steering：如果设备还没有在ZigBee网络中，那么它们会寻找一个合适的ZigBee网络并加入到其中；特别地，对于路由器类型的设备，在入网成功后，允许其他设备通过本设备加入到这个ZigBee网络中。
2. Network Formation：规定协调器类型的设备需要去建立一个中心信任的安全网络，这种网络的特点是所有需要加入到网络中的设备都需要经过信任中心的同意才能加入，而协调器本身就是这个信任中心。所有的协调器类型的设备都必须要支持Network Formation，而对于路由器类型的设备来说，是可选的模式。
3. Finding & Bingding：ZigBee 3.0是使用Cluster（集群）来描述设备的功能的。每种设备都有各自的功能，都有各自的一系列Cluster。Finding & Bingding是指ZigBee设备的Cluster之间的相互发现、相互绑定。所有的ZigBee设备都必须要支持Finding and Binding。
4. Touchlink：一般用于两个ZigBee设备之间直接进行通信，用得相对比较少，ZigBee设备并不一定都需要支持Touchlink。

/\*协调器创建网络\*/

bdb\_StartCommissioning( // 组建网络

BDB\_COMMISSIONING\_MODE\_NWK\_FORMATION | // 支持Network Formation

BDB\_COMMISSIONING\_MODE\_FINDING\_BINDING // 支持Finding and Binding

);

/\*路由器或终端设备加入网络\*/

bdb\_StartCommissioning( // 设备入网

BDB\_COMMISSIONING\_MODE\_NWK\_STEERING | // 支持Network Steering

BDB\_COMMISSIONING\_MODE\_FINDING\_BINDING // 支持Finding and Binding

);

# 基于AF完成节点间的通讯

/\*AF层数据发送API\*/

/\*

\* @param dstAddr 目标设备地址，包含网络地址和端点号

\* @param srcEP 发送设备的简单描述符

\* @param cID Cluster ID，后续章节将会详细讲解

\* @param len 待发送数据的长度

\* @param buf 待发送的数据

\* @param transID 传输ID，可以用来给每一次发送的数据包编一个号

\* @param options 附加选项，可以用来给这次数据发送添加一些说明

\* @param radius 最大的路由跳转级数

\*/

afStatus\_t AF\_DataRequest(

afAddrType\_t \*dstAddr,

endPointDesc\_t \*srcEP,

uint16 cID,

uint16 len,

uint8 \*buf,

uint8 \*transID,

uint8 options,

uint8 radius);

## 点对点通讯

图示

描述已自动生成

首先，协调器节点根据设定的PAN ID、通道等网络参数进行初始化，启动网络后协调器节点广播网络信息，使得周围的设备可以发现网络。接着，路由器或终端节点发送加入请求给协调器，协调器节点响应加入请求，发送加入响应给路由器或终端节点。路由器或终端节点发送网络密匙给协调器节点。协调器节点确认网络密匙后，发送网络密匙确认给路由器或终端节点。路由器或终端节点开始发送数据给协调器节点。最后，协调器节点发送数据确认给路由器或终端节点，完成入网和数据传输流程。

/\*点对点通信\*/

/\*

\* @param destNwkAddr 目标设备的网络地址

\* @param cid Cluster ID，后续课程将会详细讲解

\* @param len 数据长度

\* @param data 数据内容

\*/

static void zclSampleSw\_AF\_P2P(

uint16 destNwkAddr,

uint16 cid,

uint8 len,

uint8 \*data)

{

afAddrType\_t dstAddr; // 寻址信息配置

static uint8 transferId = 0; // 传输ID，是数据包的标识符

/\* Destination \*/

dstAddr.addrMode = afAddr16Bit; // 设置目标地址模式为16位网络地址，表示使用P2P的通信方式

dstAddr.addr.shortAddr = destNwkAddr; // 目标设备的网络地址

dstAddr.endPoint = SAMPLESW\_ENDPOINT; // 目标设备的端点号

transferId++;

AF\_DataRequest(&dstAddr,

&sampleSw\_TestEp, // 已经创建好的简单描述符

cid,

len,

data,

&transferId,

AF\_DISCV\_ROUTE, // 进行路由扫描操作，用于建立发送数据报文的通信路径。关于这个参数，暂时保持例程默认的代码就可以了

AF\_DEFAULT\_RADIUS); // 指定最大的路由跳转级数

}

## 广播通信

图形用户界面, 图示, 应用程序

描述已自动生成

对于协调器而言，协调器根据设定的PAN ID、通道等网络参数进行初始化。接着，协调器开始定期广播消息到整个网络，这些消息会被网络中的所有设备接收；对于路由器或终端节点而言，首先发送加入请求，协调器响应并允许其加入网络。一旦路由器或终端节点加入网络，它们将开始接受来自协调器的广播消息。

/\*广播通信\*/

/\*

\* @param cid Cluster ID

\* @param len 待发送数据的长度

\* @param \*data 待发送数据的内容

\*/

static void zclSampleSw\_AF\_Broadcast(

uint16 cid,

uint8 len,

uint8 \*data)

{

afAddrType\_t dstAddr;

static uint8 transferId = 0;

/\* Destination \*/

dstAddr.addrMode = afAddrBroadcast; // 使用广播模式

dstAddr.addr.shortAddr = 0xFFFF; // 广播地址

dstAddr.endPoint = SAMPLESW\_ENDPOINT; // 目标设备的端点号

/\* Transfer id \*/

transferId++;

/\* Send \*/

AF\_DataRequest(

&dstAddr,

&sampleSw\_TestEp, // 已经创建好的简单描述符

cid,

len,

data,

&transferId,

AF\_TX\_OPTIONS\_NONE,

AF\_DEFAULT\_RADIUS); // 指定了最大的路由跳转级数

}

## 组播通信

图示

描述已自动生成

对于协调器而言，首先，协调器根据设定的PAN ID、通道等网络参数进行初始化，定义组播表和添加组播组，然后才开始定期广播组播消息到定义的组播组；对于路由器或终端节点而言，首先定义相同的组播组ID，接着发送加入请求，协调器响应后允许其加入网络。在加入网络后，请求加入协调器定义的组播组，协调器确认加入请求，并允许其接收组播消息。最后，路由器或终端节点开始接收来自协调器的组播消息。

/\*组播通信\*/

/\*

\* @param groupId 组ID

\* @param cid ClusterID，后续章节将会详细讲解

\* @param len 待发送数据的长度

\* @param data 待发送数据的内容

\*/

static void zclSampleSw\_AF\_Groupcast(

uint16 groupId,

uint16 cid,

uint8 len,

uint8 \*data)

{

afAddrType\_t dstAddr;

static uint8 transferId = 0;

/\* Destination \*/

dstAddr.addrMode = afAddrGroup; // 使用组播通信模式

dstAddr.addr.shortAddr = groupId;

dstAddr.endPoint = SAMPLESW\_ENDPOINT; // 组中的设备的端点号

/\* Transfer id \*/

transferId++;

/\* Send \*/

AF\_DataRequest(&dstAddr,

&sampleSw\_TestEp, // 已经创建好的简单描述符

cid,

len,

data,

&transferId,

AF\_TX\_OPTIONS\_NONE,

AF\_DEFAULT\_RADIUS); // 指定最大的路由跳转级数，暂时可忽略

}

## AF层基础概念

1. 简单描述符：用于描述设备端点的一个结构体，包含了端点号、应用配置文件ID、设备版本、输入集、输出集等信息。它提供了端点的基本信息，使得其他设备能够理解该端点的功能和应用，有助于网络中的设备发现和通信。
2. 端点描述符：用于描述设备端点特性的一个结构体，包括端点类型、设备ID、设备版本、设备配置等。它提供了端点的详细描述，包括端点支持的集群和功能，有助于网络中的设备了解端点的详细能力，以便进行正确的通信和配置。
3. 端点号：用于区分不同应用或服务的标识符，每个端点号对应一个特定的应用或服务。端点号允许多个应用在同一设备上独立操作，每个应用或服务都可以通过其唯一的端点号进行识别和通信。
4. 短地址：ZigBee网络中分配给每个设备的网络层地址，是一个16位的唯一标识符，用于网络层的通信，用于在ZigBee网络中标识和定位设备。
5. MAC地址：ZigBee设备在物理层和数据链路层的唯一标识符，通常是一个64位的IEEE 802地址，用于在ZigBee网络的低层次（物理层和数据链路层）标识设备，确保数据帧的正确发送和接收。

# 根据应用场景选择合适的传感器

1. 温度传感器：DS18B20数字温度传感器。工业自动化中用于监控和控制系统，确保生产设备正常运行；智能家居系统中用于调节室内温度。
2. 压力传感器：HX711模数转换芯片ADC。电子秤中用于称重，例如人体秤、厨房秤；工业应用中用于测量液体或气体的压力变化。
3. 光电传感器：BH1750数字光强度传感器。自动照明系统控制，如智能路灯根据周围光线自动调整亮度；消费电子产品中用于自动调节屏幕背光亮度。
4. 接近传感器：电感式、电容式、超声波和红外线接近传感器。能检测附近物体的存在，无需物理接触，工业自动化、智能家居、智能手机中有着广泛的应用。
5. 加速度传感器：MEMS加速度传感器。能够测量物体在三个轴向上的加速度变化，惯性导航系统、手机屏幕旋转识别、运动健康监测设备中的核心组件。
6. 磁力计：HL-26SS30霍尔效应开关。无接触式位置检测，如门禁系统中的开闭状态检测。
7. 湿度传感器：DHT22温湿度传感器。农业温室中用于维持适宜的生长环境；空调系统中用于调节室内湿度水平。
8. 气体传感器：MQ-2可燃气体和烟雾传感器。家庭安全系统中用于检测厨房、卧室等区域的烟雾浓度；工业环境中用于监测可燃气体泄漏以预防火灾事故。
9. 生物传感器：血糖传感器、心电图传感器。随着可穿戴技术的发展，越来越多的生物传感器被集成到手表、手环等便携式设备中，帮助用户随时监测身体健康状态。
10. 图像传感器：电荷耦合器件（CCD）、有源像素传感器（CMOS）。通过捕捉光线并将光信号转换为电信号，实现图像的数字化。广泛应用于消费电子、安防监控、医疗成像、机器视觉等领域。

# 链路协议与传输协议

## 链路协议

ZigBee支持网络拓扑结构，使用分布式自组织的网络部署，低功耗，适合大量节点，其传输距离为室内70米至室外300米，功率和成本极低，但速率最高只有250kbps，适用于智能家居设备之间的互联，如智能锁、灯光、恒温器等。

蓝牙支持快速连接和多种拓扑结构，使用中心化或分布式的网络部署，传输距离约100米，功率极低，最新版本的蓝牙的速率可达2Mbps，成本中等，适用于短距离个人区域网（PAN），例如可穿戴设备、医疗保健设备等。

WiFi具有高数据传输速率的特点，支持直接接入互联网，省去额外网关依赖，使用星型拓扑需要路由器接入，传输距离一般为几十到上百米，功率和成本较高，不适用于那些依赖小型电池长时间运行的应用，广泛应用于家庭或企业内部的局域网连接，以及互联网接入。

LoRa具有远程通信能力，具有低功耗的特点，适用于广域网，使用LPWAN架构，需要运营商提供基础设施，传输距离在城市中可达1~2公里，郊区最高可达20公里，其功率和成本较低，上行传输速率最大50kbps，下行传输速率最大115.2kbps，适用于农业监测、城市基础设施管理等领域。

4G/5G具有广泛覆盖、支持高速移动性和大数据量传输的特点，使用蜂窝网络部署，依赖电信运营商基站，传输距离可达几十公里，功率消耗中等，成本较高，传输速率5G理论峰值可达10Gbps，适用于移动宽带服务和车联网等。

星闪具有强抗干扰能力远距离传输的特点，使用专网专用的网络部署，功率适中，成本较高，传输距离优于WiFi，适用于智能电网、交通指挥系统等。

传输距离：

LoRa > 4G/5G > ZigBee > WiFi > 蓝牙 ≥ 星闪

功率：

4G/5G ≥ LoRa > WiFi > ZigBee > 蓝牙 ≥ 星闪

传输速率：

4G/5G > WiFi > 星闪 > ZigBee > 蓝牙 > LoRa

## 传输协议

1. MQTT：一种轻量级的消息传输协议，适用于低带宽、高延迟或不可靠网络环境。MQTT使用发布/订阅模式，主要涉及三个角色：发布者（Publisher）、订阅者（Subscriber）和代理（Broker）。发布者：发送消息到代理；代理：接收来自发布者的消息，并根据订阅关系将消息转发给订阅者；订阅者：订阅感兴趣的主题，并从代理接收消息。
2. CoAP：一种基于UDP的应用层协议，适用于资源受限的设备。CoAP采用客户端/服务器架构，客户端发送请求到服务器，服务器响应请求。客户端：发送请求到服务器；服务器：处理请求并返回响应；确认机制：客户端和服务器之间可以通过确认机制确保消息的可靠传输。
3. LwM2M：一种专门设计用于管理 IoT 设备的标准协议，提供了设备管理和安全功能。LwM2M采用客户端/服务器架构，客户端（即IoT设备）与服务器（即设备管理平台）之间进行通信。客户端：注册到服务器，并定期发送心跳包以保持连接；服务器：管理客户端的状态，发送配置命令、固件更新等指令；数据传输：客户端通过CoAP或其他协议将数据发送到服务器。
4. HTTP：一种基于文本的应用层协议，广泛应用于万维网（World Wide Web）的数据传输。HTTP设计为无状态的客户端/服务器架构，支持多种请求方法来操作资源。客户端（Client）：发起请求的一方，通常是浏览器或其他类型的用户代理，它们向服务器发送请求以获取或提交数据；服务器（Server）：接收来自客户端请求的一方，负责处理请求并返回相应的响应信息，如网页内容、文件、图片等。

# ZigBee协议层次结构与OSI七层结构

## ZigBee协议层次结构

**IEEE 802.14.4 PHY layer**：物理层，其工作内容由IEEE 802.15.4定义，主要作用是将一个设备的数据转换为电磁波信号之后发送到另一个设备，再由另一个设备解读电磁波信号获取数据。

**IEEE 802.14.4 MAC layer**：MAC层，其工作内容由IEEE 802.15.4定义，其主要作用是控制多个网络设备有序地利用物理通信资源电磁波来通信。

**Network（NWK）layer**：网络层，负责多个设备之间的组网、数据传输以及网络安全管理等。

**Application layer**：应用层，可以划分为以下两个层次：  
（1）Application Support（APS）Sub-Layer：应用支持子层，是网络层到应用框架层的过渡，提供数据的收发、安全加密，以及设备地址管理等功能。  
（2）Application Framework：应用框架层，由一个或者多个应用端点（EndPoint）组成。应用端点是不同设备间通信的出入口，同时也是描述设备具备哪些功能的基础。

ZigBee设备进行网络通信时，最终是应用端点间的通信。例如，ZigBee协调器的端点1向智能插座的端点1发送了打开插座命令。

**ZigBee Cluster Library（ZCL）**：ZigBee集群库，由ZigBee联盟定义，是ZigBee 3.0的基础，也是不同厂商设备能够互联互通的基础。

**ZigBee Device Objects（ZDO）**：ZigBee设备对象，也就是应用端点0，它是应用层其他端点与应用子层管理实体交互的中间件，主要提供的功能如下：  
（1）管理ZigBee设备。  
（2）创建、扫描及加入网络。  
（3）应用端点绑定、解绑。  
（4）安全管理。

**Base Device Behavior（BDB）**：ZigBee基础设备行为，定义设备的行为规范，以确保不同厂商设备间的互操作性，基础设备行为规范的范围定义如下：  
（1）基础设备所需的环境。  
（2）基础设备的初始化（initialization）过程。  
（3）基础设备的 Commissioning 过程。  
（4）基础设备的重置（reset）过程。  
（5）基础设备的安全（security）过程。

图示

描述已自动生成

## OSI七层结构



# 获取shortAddr和macAddr

/\*非协调器节点获取地址信息\*/

// P2P Event: 检查当前事件是否包含P2P事件标志

if (events & SAMPLEAPP\_P2P\_EVT)

{

NodeAddressInfo nodeInfo; // 定义一个结构体变量用于存储节点地址信息

// 获取本节点短地址

nodeInfo.shortAddr = NLME\_GetShortAddr(); // 调用NLME\_GetShortAddr()函数获取节点的16位短地址并赋值给nodeInfo.shortAddr

// 注意：这里调用了NLME\_GetExtAddr()，但返回值未被使用。

// 此行可能是为了填充全局变量saveExtAddr，后续会将其复制到nodeInfo.extAddr中。

NLME\_GetExtAddr();

// 将之前保存的扩展地址复制到nodeInfo.extAddr中

memcpy(nodeInfo.extAddr, saveExtAddr, sizeof(nodeInfo.extAddr));

// 获取父节点的短地址（通常是协调器或路由器）

nodeInfo.parentShortAddr = NLME\_GetCoordShortAddr(); // 调用NLME\_GetCoordShortAddr()获取父节点的16位短地址

// 获取父节点的扩展地址

NLME\_GetCoordExtAddr(nodeInfo.parentExtAddr); // 调用NLME\_GetCoordExtAddr()并将结果直接存储在nodeInfo.parentExtAddr中

// 序列化数据：准备将NodeAddressInfo结构体的数据发送出去

uint8 buffer[sizeof(NodeAddressInfo)]; // 定义一个缓冲区用于存放序列化的数据

memcpy(buffer, &nodeInfo, sizeof(NodeAddressInfo)); // 将nodeInfo结构体内容复制到buffer中

// 发送序列化后的数据

zclSampleSw\_AF\_P2P(0x0000, CLUSTER\_P2P, sizeof(buffer), buffer); // 通过zclSampleSw\_AF\_P2P函数发送数据，参数分别为目的地地址、集群ID、数据长度和数据指针

// 设置定时器，使SAMPLEAPP\_P2P\_EVT事件再次触发

osal\_start\_timerEx(zclSampleSw\_TaskID, // 启动定时器，参数为任务ID

SAMPLEAPP\_P2P\_EVT, // 事件类型

SAMPLEAPP\_P2P\_PERIOD); // 定时周期

// 返回剩余的事件，去除已经处理过的P2P事件标志

return (events ^ SAMPLEAPP\_P2P\_EVT);

}

/\*数据接收\*/

static void zclSampleSw\_AF\_RxProc(afIncomingMSGPacket\_t \*MSGpkt)

{

static uint8 p2pCnt = 0; // 记录P2P消息的数量

static uint8 bcCnt = 0; // 记录广播消息的数量

static uint8 gcCnt = 0; // 记录组播消息的数量

switch (MSGpkt->clusterId)

{

case CLUSTER\_P2P: // 如果消息属于P2P集群

p2pCnt++; // 增加P2P消息计数

// 将消息数据指针转换为NodeAddressInfo结构体指针

NodeAddressInfo \*nodeInfo = (NodeAddressInfo \*)MSGpkt->cmd.Data;

// 遍历networkNodes数组，寻找空位或者已存在的节点信息

for (int i = 0; i < sizeof(networkNodes) / sizeof(networkNodes[0]); i++)

{ // 注意：这里应使用正确的数组大小计算方法

if (networkNodes[i] == 0)

{ // 如果找到空位

networkNodes[i] = nodeInfo->shortAddr; // 添加新节点的短地址到数组中

// 打印节点信息

printf("Node Info:\n");

printf("Short Address: 0x%X\n", nodeInfo->shortAddr);

printf("Extended Address: ");

for (int j = 7; j >= 0; j--)

{ // 使用不同的循环变量避免混淆

printf("%02X ", nodeInfo->extAddr[j]); // 格式化输出扩展地址

}

printf("\nParent Short Address: 0x%X\n", nodeInfo->parentShortAddr);

printf("Parent Extended Address: ");

for (int j = 7; j >= 0; j--)

{ // 使用不同的循环变量避免混淆

printf("%02X ", nodeInfo->parentExtAddr[j]); // 格式化输出父节点扩展地址

}

printf("\n");

break;

}

// 如果在数组中找到已存在的节点信息，则不再继续添加

else if (networkNodes[i] == nodeInfo->shortAddr)

{

break;

}

}

break;

// 其他集群ID的处理逻辑...

}

}

# 网络地址分配

图示

中度可信度描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成表格

描述已自动生成

图表, 散点图

描述已自动生成

# 传感器数据收发实验

首先根据电路原理图修改OLED引脚防止冲突。CS引脚用于启用微控制器和LCD之间的通信，DS引脚用于指示通过SPI发送的数据是命令还是显示数据。

图片包含 图形用户界面

描述已自动生成

/\*修改引脚\*/

#define HAL\_LCD\_SPI\_CS\_PORT 1

#define HAL\_LCD\_SPI\_CS\_PIN 3

#define HAL\_LCD\_SPI\_DS\_PORT 0

#define HAL\_LCD\_SPI\_DS\_PIN 0

/\*修改硬件借口\*/

#define HAL\_DHT11\_PORT 0

#define HAL\_DHT11\_PIN 4

/\*获取温湿度和光照\*/

// 获取温湿度并检测数值是否有效

dht11Dat = halDHT11GetData();

if (!(dht11Dat.ok) || (dht11Dat.temp == 0 && dht11Dat.humi == 0))

{

return;

} // 这里的接口需要找到对应的文件hal\_dht11.h，通过原理电路图判断接口号

// 读取P07的ADC值，即光照强度值

lightAdc = HalAdcRead(HAL\_ADC\_CHN\_AIN5, HAL\_ADC\_RESOLUTION\_8); // HAL\_ADC\_CHN\_AIN5是光照硬件接口对应变量，从原理电路图判断此处应选择几号接口

// 读取土壤湿度

wetAdc = HalAdcRead(HAL\_ADC\_CHN\_AIN6, HAL\_ADC\_RESOLUTION\_8);

/\* 优化光照度和土壤湿度值 \*/

#define LIGHT\_ADC\_MAX\_VALUE 120 // 光照度最大值

#define WET\_ADC\_MAX\_VALUE 120 // 土壤湿度最大值

float lightAdcFloat;

float wetAdcFloat;

// 裁剪光照度值

if (lightAdc > LIGHT\_ADC\_MAX\_VALUE)

{

lightAdc = LIGHT\_ADC\_MAX\_VALUE;

}

lightAdcFloat = (float)lightAdc \* 100 / LIGHT\_ADC\_MAX\_VALUE;

// 处理土壤湿度值

if (wetAdc > WET\_ADC\_MAX\_VALUE)

{

wetAdc = WET\_ADC\_MAX\_VALUE;

}

wetAdcFloat = (float)wetAdc \* 100 / WET\_ADC\_MAX\_VALUE;

// 将浮点数转换为整数部分和小数部分

lightAdcFloatInt = (int)lightAdcFloat;

lightAdcFloatDec = (int)((lightAdcFloat - lightAdcFloatInt) \* 10);

wetAdcFloatInt = (int)wetAdcFloat;

wetAdcFloatDec = (int)((wetAdcFloat - wetAdcFloatInt) \* 10);

/\*路由器收集到数据，发送给协调器\*/

/\* 往协调器发送数据 \*/

char msg[64]; // 定义一个长度为64的字符数组来存储消息

sprintf(msg, "D%sT%d.%dH%d.%dL%d.%dW%d.%d#",

zb\_port\_get\_local\_mac(), // 设备本地MAC地址

tempInt, // %d - 温度整数部分

tempDec, // %d - 温度小数部分

humInt, // %d - 湿度整数部分

humDec, // %d - 湿度小数部分

lightAdcFloatInt, // %d - 光照度整数部分

lightAdcFloatDec, // %d - 光照度小数部分

wetAdcFloatInt, // %d - 土壤湿度整数部分

wetAdcFloatDec // %d - 土壤湿度小数部分

);

/\*路由器获取自身物理地址部分\*/

/\*

\* 用于获取本地MAC地址

\* @return MAC地址

\*/

static const char \*zb\_port\_get\_local\_mac(void)

{

uint8 i;

uint8 \*xad;

static uint8 localMacStr[Z\_EXTADDR\_LEN \* 2 + 1];

xad = aExtendedAddress + Z\_EXTADDR\_LEN - 1;

for (i = 0; i < Z\_EXTADDR\_LEN \* 2; xad--)

{

uint8 ch;

ch = (\*xad >> 4) & 0x0F;

localMacStr[i++] = ch + ((ch < 10) ? '0' : 'A' - 10);

ch = \*xad & 0x0F;

localMacStr[i++] = ch + ((ch < 10) ? '0' : 'A' - 10);

}

// 打印自身地址

// printf("%s\n", (const char \*)localMacStr);

return (const char \*)localMacStr;

}

/\*路由器入网后，协调器收到路由器发送的数据\*/

/\*

\* 接收到ZigBee消息事的处理函数

\*

\*@param nwkaddr 发送者的网络地址

\*@param cluster Cluster

\*@param data 数据内容

\*@param len 数据长度

\*/

void zb\_port\_on\_priv\_msg(uint16\_t nwkaddr,

uint16\_t cluster,

const uint8\_t \*data,

uint16\_t len)

{

#if ZG\_BUILD\_COORDINATOR\_TYPE // 协调器

if (cluster == PRIV\_CID\_MESSAGE)

{

HalLcdWriteString("get data", 3);

// 打印接收到的数据（可以在此处添加一个打印语句查看是否正确获取）

}

#elif ZG\_BUILD\_RTRONLY\_TYPE

#else

#endif

}

/\*协调器上传到PC端\*/

#if ZG\_BUILD\_COORDINATOR\_TYPE

/\*

\*从串口中获取到数据时的处理函数

\* @param data 数据内容

\* @param len 数据长度

\*/

void zb\_on\_msg(const uint8\_t \*data, uint8\_t len)

{

// 可加一个打印语句查看数据

}

#endif

# CC2530外围电路

晶振电路：

图示

描述已自动生成

射频电路：

图示

描述已自动生成

外围电路：

图片包含 游戏机, 电路, 站

描述已自动生成

# 传感器连接方式与DHT11

## 数字信号传感器连接

I2C接口：CC2530的GPIO可以配置为I2C的SCL（时钟线）和SDA（数据线），连接到支持I2C协议的传感器。

SPI接口：SPI（串行外设接口）也是一种常见的数字通信接口，适用于许多传感器（如气体传感器、光传感器等）。SPI有四条主要信号线：SCK（时钟线）、MISO（主输入从输出）、MOSI（主输出从输入）和CS（片选）。

## 模拟信号传感器连接

ADC输入：CC2530具有内建的模拟数字转换器（ADC），可以将模拟信号转换为数字信号。模拟传感器（如温度传感器、光敏电阻等）通常输出一个连续变化的电压信号，可以直接连接到CC2530的ADC引脚（如P0.x）。

## AD口

如果需要使用8位的采样精度来采P0\_0的ADC值，可按一下方式调用HalAdcRead：

/\*AD口\*/

uint8 adcVal;

adcVal = HalAdcRead(HAL\_ADC\_CHANNEL\_0, HAL\_ADC\_RESOLUTION\_8);

## DHT11

/\*DHT11\*/

#define dht11\_high GPIO\_SetBits(GPIOB, GPIO\_Pin\_12)

#define dht11\_low GPIO\_ResetBits(GPIOB, GPIO\_Pin\_12)

#define Read\_Data GPIO\_ReadInputDataBit(GPIOB, GPIO\_Pin\_12)

void DHT11\_GPIO\_Init\_OUT(void);

void DHT11\_GPIO\_Init\_IN(void);

void DHT11\_Start(void);

unsigned char DHT11\_REC\_Byte(void);

void DHT11\_REC\_Data(void);

图示, 示意图

描述已自动生成