GAINTS 系列开发套件
MAC433 库用户使用手册

资料版本 **V1.0**

归档时间 **2006** 年 **3** 月 **8** 日

宁波中科集成电路有限公司WSN事业部为您提供无线传感器网路方面全方位的技术支持，包括自主开发的 GAINS 系列节点和各种配套的后台软件以及 TestBed 无线网络测试平台，希望我们的产品为您的学习和研究带来方便。

公司地址：浙江省宁波市科技园区创业大厦 6 层

邮编： 315040

产品主页：http://www.wsn.net.cn/wsnnb/wsn.htm

客户服务热线：0574-87910141

E_mail: jp@nbicc.com

目录

引言 3
提醒 3
注意 3
免责申明 4
GAINTS系列节点软件开发套件介绍 4
GAINTS系列节点体系结构 2
软件协议栈对节点资源占用 3
源文件构成 4
全局变量的说明以及MAC库的一些配置 5
数据包格式声明 7
MAC433 库 8
辅助模块 21
开发模板及其架构 33
winavr编译器的makefile文件的设置 38

引言

无线传感器网络是一个全新的科学研究方向，它是一种短距离、低速率无线网络技术，它在工业控制，交通监控，仓储物流，环境和构筑物监测、抢险救灾以及军事等多个方面有着广泛的应用前景。

本文旨在引导用户使用 GAINTS 系列节点的软件开发套件，使您对套件的软件机构和硬件系统有更深入的了解。手册提供对物理层和 MAC 层协议栈函数库中所有接口函数的详细的说明，使得用户可以方便的调用并且进行网络层等更高层次应用的开发。

提醒

协议栈使用 C 语言编程，本文中对函数的解释均以 C 语言的形式出现。GAINTS 系列 MAC433 协议栈的设计考虑了多方面的因素，在发布该文档时，协议栈具有以下的特性：

- ◆ 部分功能参照了 IEEE 802.11 无线局域网协议
- ◆ 使用 Chipcon CC1000 RF 收发器支持 433MHz 频带
- ◆ 支持广播和单播方式
- ◆ 支持 ACK 和 NACK 两种方式
- ◆ 采用 CSMA-CA 机制实现信道的访问
- ◆ 支持睡眠机制
- ◆ 模块化设计使得协议栈的结构十分清晰，易于删除和修改
- ◆ 支持 AVR-GCC 4.0.7 编译器
- ◆ 可以在大多数 ATmega 系列单片机之间进行移植
- ◆ 提供了 ATmega 系列单片机的一些功能模块的支持（这些在辅助模块中）

注意

在使用 GAINTS 系列节点 MAC 协议栈时请注意一些局限：

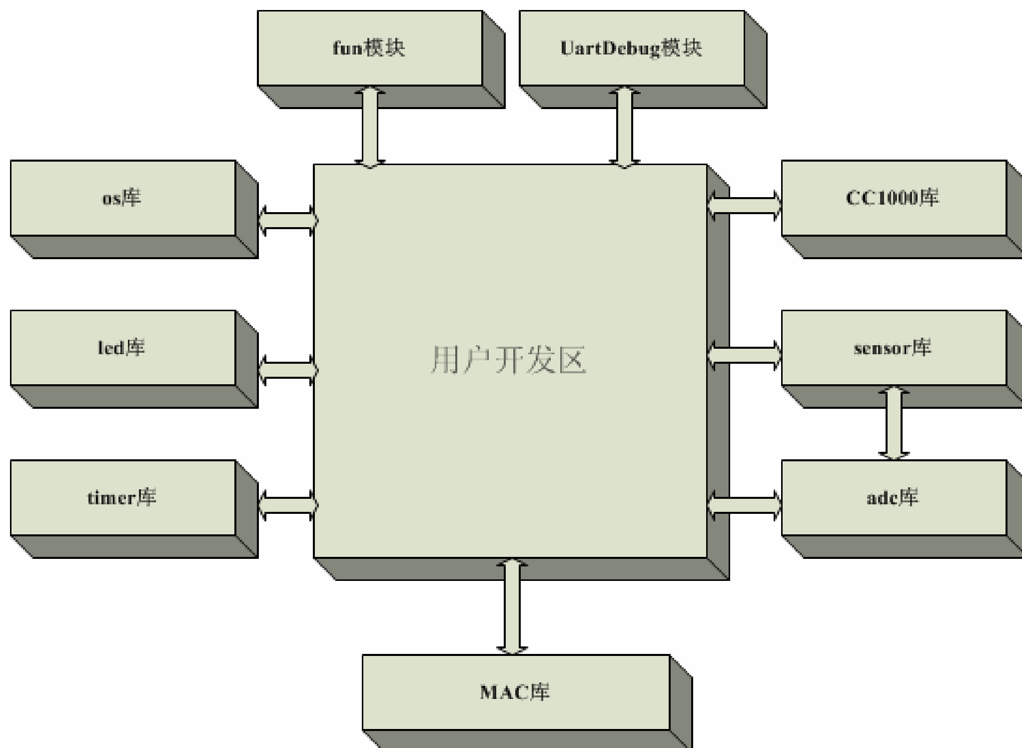
- ◆ 协议还没有实现 RTS/CTS 机制
- ◆ 没有对缓存队列的管理
- ◆ 没有加入安全机制
- ◆ 本协议栈仅提供 MAC 层功能相关的库文件，不提供协议源代码，但是对所有的接口函数做了详细的说明。
- ◆ 本公司会继续对该版本的 MAC 协议栈升级

免责申明

本公司提供该版本协议栈仅用于各科研机构学术研究用途，不确保商业用途性能稳定，用户使用本协议栈做商业用途产生任何后果，本公司不负任何相关责任。

GAINTS 系列节点软件开发套件介绍

为了方便大家的研究和开发，本公司在开发出了兼容 MICA 系列的无线传感器节点硬件平台 GAINTS 系列之后，制作了相关的软件开发套件。这些套件包括基于 433Mhz 载频的无线传感器网络 MAC 层协议栈的库，以及相关的辅助开发模块。MAC 协议栈的库主要提供透明的传输数据包的能力，并且支持 ACK 机制以及睡眠机制。辅助模块包括控制 LED 的库，控制传感器的（现阶段只提供光传感器的库）库，控制 ADC 的库，控制 0 号计数器的库，射频驱动的库，任务调度库，功能函数模块以及 Uart 驱动模块，使用这些库和模块基本上可以实现现有的一些需求，而且我们会提供一个开发模板，使用该模板可以让你的开发更方便，而且用户可以随意添加自己需要的功能。下面是我们提供的模版的整个系统的一个架构，用户可以按照自己的需要裁减，也可以方便的添加自己的模块。

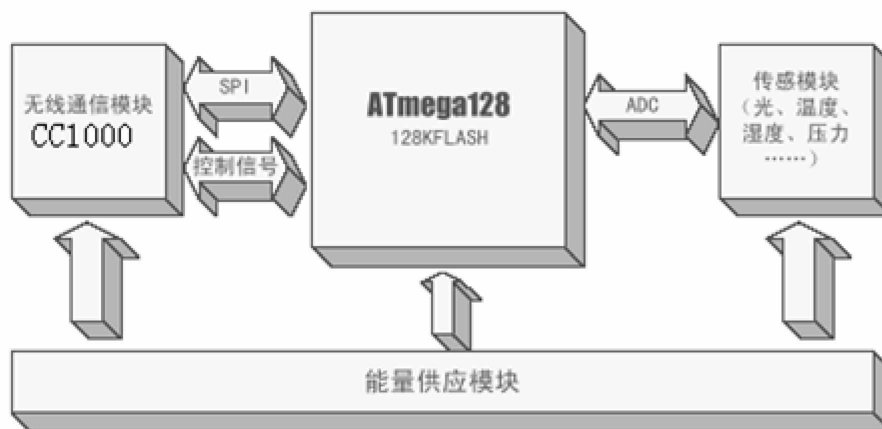


我们提供的 MAC 协议栈的库，给用户提供了丰富的接口，便于用户自己开发和改进 MAC 层的协议。现阶段我们提供的 1.0 版本的 MAC 提供了 ACK 机制以及透明的数据包传输能力，提供了支持射频睡眠的接口，采用了 CSMA/CA 技术，使用二进制退避算法实现冲

突避免。

GAINTS 系列节点体系结构

GAINTS 系列节点的硬件由传感器模块、处理模块、无线通信模块和能量供应模块四个部分组成，如下图所示。传感器模块负责监测区域内信息的采集和数据转换，处理模块负责控制整个传感器节点的处理操作、存储和处理本身采集的数据和其它节点发来的数据，包括了数据安全、通信协议、同步定位、功耗管理、任务管理等等；无线通信模块负责与其它传感器节点进行无线通信，交换控制消息和收发采集数据；电源供应模块为传感器节点提供运行所需的所有电源。



GAINTS系列节点体系结构

- ◆ *处理器模块:* GAINTS 系列节点使用 ATMEGA128 单片机作为控制器和处理核心，相对于其它通用的 8 位微控制器来说，它具有非常丰富的资源和更低的功耗。ATMEGA128 具有 128K 字节的片内程序存储器（Flash），4K 字节的数据存储器（SRAM，可外扩到 64K）和 4K 字节的 E2PROM 以及丰富的对外接口。在 GAINTS 系列节点中，ATMEGA128 的驱动时钟使用频率为 7.3728M 的晶振，另外配置一个频率为 32.768k 的晶振作为计数器的外部时钟（相关内容可以参考 ATMEGA128 的手册）。
- ◆ *无线通信模块:*无线通信模块核心采用工作在 433MHz 在的单芯片低电压 CC1000 收发器。该射频芯片具有工作电压低（2.1v~3.6v 均可工作）、能耗低、体积小等非常适合于集成的特点。它采用 FSK 调制方式，外部采用 SPI 的接口，可以和微控制器直接相联。CC1000 使用频率为 14.745M 的晶振作为驱动，在该驱动下面 CC1000 可以提供的最大数据传输率为 19.2kbps，也就是说每毫秒不到 3 个字节，这个数据对 MAC 层的协议是

很有用的，在设置 ACK 等待时间和 RTS—CTS 等待时间的时候这是需要考虑的重要参数。

◆ *传感器模块:*考虑到各种不同的应用场合中需要采集的模拟量千差万别,我们选用了一些当今应用最为常见的传感器,设计了拥有通用接口的传感器子模块.传感器电路部分设计采用 power gating 技术在无采集数据任务时降低功耗。

软件协议栈对节点资源占用

◆ *软件协议栈对节点 I/O 资源的占用要求*

软件对节点硬件 I/O 端口的占用用于对外部的传感器，LED，UART，JTAG，外部 Flash 等的接口以及与射频芯片之间的通信。

ATMEGA128 与 CC1000 的引脚连接如下表（详细的情况可以参考我们给出的原理图）：

ATMEGA128I/O 引脚	CC1000 RF 收发器引脚
PB3（输入输出）	DIO
PB1（输出）	DCLK
PD6（输出）	PCLK
PD7（输入输出）	PDATA
PD4（输出）	PALE
PF0 （输入）	RSSI/IF

ATMEGA128 与传感器以及 ADC 的接口：

ATMEGA128I/O 引脚	作用
PE5（输出）	给传感器供电
PF0（输入）	作为 ADC0 的通道用于载波监听
PF1（输入）	作为 ADC1，是传感器数据的输入通道

我们现阶段只使用了 ADC0 和 ADC1。

ATMEGA128 与其它功能部件的口：

ATMEGA128 定时器	作用
UART0	用于串口通讯
UART1	用于外部的串行 Flash
ADC4~ADC 7	用于 JTAG
PA0~PA2	用于对 LED 的控制

参照原理图 ATMEGA128 与其它功能部件的的接口可以很容易的掌握。

◆ *软件协议栈对节点定时资源的占用要求*

ATMEGA128 定时器	作用
定时器 0	提供相应的驱动供用户使用
定时器 1	使用了 A 类比较中断，用于 MAC 的超时计数

源文件构成

源文件的构成以我们给定的用户开发模版为原型，介绍我们提供的功能，包括 MAC 层的功能以及其它一些模块的功能，详细的情况参考下面的源代码列表：

<table>
<tr><td rowspan="14">硬件驱动</td><td rowspan="12">其余驱动</td><td>led.h</td><td rowspan="2">实现对 LED 的控制</td><td rowspan="12">辅助模块</td></tr>
<tr><td>led.c</td></tr>
<tr><td>adc.h</td><td rowspan="2">实现对128单片机ADC转换器的控制</td></tr>
<tr><td>adc.c</td></tr>
<tr><td>sensor.h</td><td rowspan="2">实现对外部传感器的控制，包括对其的电源管理和数据的提取。</td></tr>
<tr><td>sensor.c</td></tr>
<tr><td>os.h</td><td rowspan="2">提供基本的任务调度，可以看作是一个小型的操作系统</td></tr>
<tr><td>os.c</td></tr>
<tr><td>fun.h</td><td rowspan="2">实现对 128 单片机的一些寄存器的配置和访问功能函数</td></tr>
<tr><td>fun.c</td></tr>
<tr><td>timer.h</td><td rowspan="2">实现对 128 单片机中 0 号计数器的控制，并且虚拟出两个计数器</td></tr>
<tr><td>timer.c</td></tr>
<tr><td rowspan="2">射频驱动</td><td>cc1000.h</td><td rowspan="2">提供射频芯片的底层驱动和配置，以及 128 单片机与 cc1000 之间的通讯</td><td rowspan="10">MAC 库</td></tr>
<tr><td>cc1000.c</td></tr>
<tr><td rowspan="4">PHY代码</td><td rowspan="4">物理层</td><td>radiocontrol.h</td><td rowspan="2">调用cc1000实现数据传输的功能模块，可以实现 RF 的睡眠</td></tr>
<tr><td>radiocontrol.c</td></tr>
<tr><td>physical.h</td><td rowspan="2">负责将数据帧以字节为单位提供给 radiocontrol，并且负责将收到的数据打包发送给 MAC 层</td></tr>
<tr><td>physical.c</td></tr>
<tr><td rowspan="4">MAC代码</td><td rowspan="4">MAC 层</td><td>macClock.h</td><td rowspan="2">1 号计数器的控制模块，主要用于 MAC 层的超时计数</td></tr>
<tr><td>macClock.c</td></tr>
<tr><td>mac.h</td><td rowspan="2">MAC 层模块，负责 mac 层的功能和调度，包括 RTS—CTS 机制，ACK 机制，透明的传输数据包的功能</td></tr>
<tr><td>mac.c</td></tr>
</table>

全局变量的说明以及 MAC 库的一些配置

一些重要的全局变量放在 global.h 文件中，该文件主要是放置一些和 MAC 的配置有关的全局变量，下面是该文件中一些变量的意义：

选项名称　POWERLEVEL

用途　　　用来设定 CC1000 射频芯片的输出功率，下面是 CC1000 的手册提供的数据，需要按照要求设定，具体的可以参考 CC1000 的用户手册，需要注意的是要使用 16 进制。

示例　　　下面的行将 CC1000 的输出功耗设定为-20dBm

POWERLEVEL = 0X01; //射频的功率

Output power [dBm]	RF frequency 433 MHz		RF frequency 868 MHz	
	PA_POW [hex]	Current consumption, typ. [mA]	PA_POW [hex]	Current consumption, typ. [mA]
-20	01	5.3	02	8.6
-19	01	6.9	02	8.8
-18	02	7.1	03	9.0
-17	02	7.1	03	9.0
-16	02	7.1	04	9.1
-15	03	7.4	05	9.3
-14	03	7.4	05	9.3
-13	03	7.4	06	9.5
-12	04	7.6	07	9.7
-11	04	7.6	08	9.9
-10	05	7.9	09	10.1
-9	05	7.9	0B	10.4
-8	06	8.2	0C	10.6
-7	07	8.4	0D	10.8
-6	08	8.7	0F	11.1
-5	09	8.9	40	13.8
-4	0A	9.6	50	14.5
-3	0B	9.4	50	14.5
-2	0C	9.7	60	15.1
-1	0E	10.2	70	15.8
0	0F	10.4	80	16.8
1	40	11.8	90	17.2
2	50	12.8	B0	18.5
3	50	12.8	C0	19.2
4	60	13.8	F0	21.3
5	70	14.8	FF	25.4
6	80	15.8		
7	90	16.8		
8	C0	20.0		
9	E0	22.1		
10	FF	26.7		

选项名称　SINKNODE

用途　　　设置 sink 节点的 ID 号

示例　　　下面的行将 sink 节点 ID 号设置为 0

#define SINKNODE 0 //SINK 节点为 0 号节点

选项名称　OS_LOCAL_ADDRESS

用途　　　设置本节点的 ID 号

示例　　　下面的行将本节点的 ID 号设置为 6

OS_LOCAL_ADDRESS = 6; //本地节点的地址，如果该ID号和SINKNODE相同，该节点被设置为 sink 节点

选项名称　OS_BCAST_ADDR

用途　设置广播地址，用于广播时使用

示例　下面的行将广播地址设置为 0xEE

OS_BCAST_ADDR = 0xEE;　//广播地址

选项名称　ACK_ON 和 ACKTIME

用途　这两个是一对，配套使用，ACK_ON 表示是否使用 ACK 机制，如果设置为 1 表示使用 ACK 机制，为 0 表示不使用 ACK 机制。如果选择使用 ACK 机制，那么 ACKTIME 的参数就有效了，ACKTIME 代表的时间槽的长度为 16ms，提供这个时间槽的是 1 号计数器的 A 匹配中断。这里提供配置 1 号计数器的源代码：

```
#define SCALE_1ms 3
#define INTERVAL_1ms 15000    // 7650
void ClockStart(void)
{
    uint8_t intEnabled, scale = SCALE_1ms;
        uint16_t interval = INTERVAL_1ms;
        scale |= 0x8;
        cbi(TIMSK, OCIE1A);                     // 关比较器 A 的输出中断
        intEnabled = bit_is_set(SREG, 7);
        cli();                                  // 访问寄存器前先关中断
        outw(TCNT1L, 0);                        // 清空寄存器
        outw(OCR1AL, interval);                 // 设置比较匹配寄存器
        if (intEnabled)
          sei();
        outp(scale, TCCR1B);                    // 设置时钟以及分频系数
        sbi(TIMSK, OCIE1A);                     // 开中断
}
```

顺便说明一下对 0 号 8 位计数器的配置采用 32 分频，并且采用外部的 32.768k 的晶振作为时钟源，对 0 号计数器采用软件虚拟出了两个 32 位的计数器。

示例　下面的行设定 MAC 协议采用 ACK 机制，并且等待 ACK 的时间设定为 16*5ms

ACK_ON = 0;　//使用 ack 机制，建议不要使用，1 表示开启，0 表示关闭

ACKTIME = 5;　//ack 等待时间=ACKTIME*16ms

选项名称　RTS_CTS_ON 和 RTS_CTSTIME

用途 和上面的相似，这两个也是配套使用的，RTS_CTS_ON 设置为 1 表示启用 RTS/CTS 机制，为 0 表示不启动。如果 RTS_CTS_ON 被设置为 1，RTS_CTSTIME 的值就有效了，它表示发出 RTS 后等待 CTS 的时间，同上面一样它的时间槽也是 16ms。

示例 下面的行设定 MAC 协议采用 RTS/CTS 机制，并且设定等待时间为 16*4ms

```
RTS_CTS_ON = 0; //使用 RTS，CTS 机制，1 表示开启，0 表示关闭
RTS_CTSTIME = 4; //等待 CTS 时间
```

数据包格式声明

下面是 message.h 中的部分代码，相关的数据包的格式在该文件中定义。

```
#define MAX_PKT_LEN 100
#define MIN_PKT_LEN    7
#define MAC_HEADER_LEN 7
#define OS_DEFAULT_GROUP 1
#define MAC_CTRL_LEN 7
typedef struct{
    uint8_t length;                              // 包总长度
    uint8_t type;                                // 包类型
    uint8_t toAddr;                              // 目的地址
    uint8_t fromAddr;                            // 源地址
    uint8_t group;                               // 群号
    int8_t    data[MAX_PKT_LEN - MAC_HEADER_LEN];        // MAC 包数据部分
    int16_t crc;                                 // CRC16
} OSMACMsg;
typedef OSMACMsg* OSMACMsgPtr;
// MAC 数据包类型
```

length,type,toAddr,fromAddr,group 这 5 个字节是数据包的 MAC 头，length 是个比较重要的部分，它表示包的长度，由#define MAC_HEADER_LEN 7 这段代码可以看出我们在设计的时候 crc 的 2 个字节是计算在 MAC 头中的，对于 length 的计算需要先说明一下 data 数组。data[MAX_PKT_LEN - MAC_HEADER_LEN]才是用户可以自由发挥的地方，在这里用户可以定义自己的数据包格式，可以放心的是我们的 MAC433 库对数据包是透明的，数据包的解析和添加过程是在外部完成的，不过数据包要放在 data 数组中。比如我定义了一个数据包（这是我们给的一个 Demo 的数据包格式）格式：

```
#define BUFFER_LEN     10
typedef struct{
    uint8_t seqNo;
```

```
    int16_t data[BUFFER_LEN];
}SensorMsg;
```

那么我们在添充 data 的时候是这样做的，下面是代码：

```
SensorMsg *pack;
pack = (SensorMsg *)LLCmsg[LLCcurrentMsg].data;    //LLCmsg 是 OSMACMsg 类型
pack->seqNo = LLCPacketNumber;
pack->data[LLCpacketReadingNumber++] = data;
```

对于这个数据包来说，length 的计算应该是 sizeof(SensorMsg) + MAC_HEADER_LEN，这里有一点要强调的就是 length 不是一次就算出来的，只有在 MAC 层中才会加入 MAC_HEADER_LEN，在 MAC 层外面 length 只代表 data 数组中的有效数据的长度。

type 字节表示数据包的类型，这都是在 MAC 库中定义的，分为下面几种，在 MAC 中用枚举类型进行了定义：enum { RTS_PKT, CTS_PKT, DATA_PKT, ACK_PKT };

toAddr 和 fromAddr 字段表示源地址和目的地址，group 字段表示组号，MAC 中设定为 0x02，一般来说 group 字段是用不到的，可以不管，crc 字段是用来做检校用的。

在 message.h 文件中还定义了控制包的格式：

```
typedef struct{
    uint8_t length;
    uint8_t type;
    uint8_t toAddr;
    uint8_t fromAddr;
    uint8_t ack;
    int16_t crc;
} OSMACCtrlMsg;                                   // RTS/CTS/ACK 帧
typedef OSMACCtrlMsg* OSMACCtrlMsgPtr;
```

不过现阶段我们提供的 MAC 库还没有很好的完善它，而且 ACK 机制主要是和 RTS/CTS 机制配套使用的，由于现阶段我们提供的库还没有支持 RTS/CTS 机制，建议大家不要使用，在下面的版本中我们会实现和强化这方面的功能。

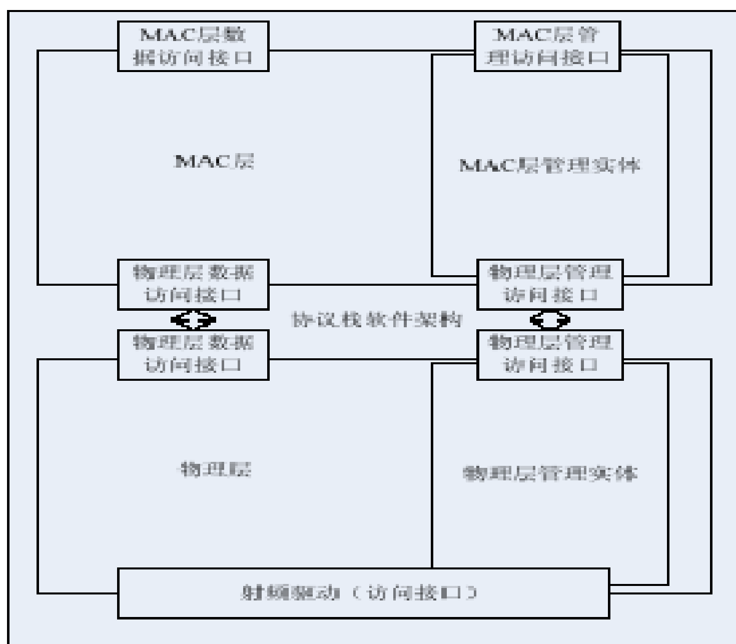
MAC433 库

◆ *MAC 库的结构*

参考 IEEE 802.11 规范，协议栈使用下面所示的架构：各层定义独立的数据访问接口与管理访问接口，层与层之间通过接口互连。其中数据访问接口提供高层发送和接收数据的能力，管理访问接口提供高层管理和设置物理层属性的能力。

物理层定义了物理无线信道和 MAC 子层之间的接口，提供物理层数据服务和物理层管理服务。物理层数据服务从无线物理信道上收发数据，物理层管理服务维护一个由物理层

相关状态数据组成的数据表。 MAC 子层使用物理层提供的服务实现设备之间的数据帧传输。MAC 子层提供两种服务：MAC 层数据服务和 MAC 层管理服务（MAC sublayer management entity, MLME)。前者保证 MAC 协议数据单元在物理层数据服务中的正确收发，后者维护一个存储 MAC 子层协议状态相关信息的数据表。



我们提供的 MAC433 库主要包含了三个模块，分别是 mac 模块，physical 模块和 radiocontrol 模块，mac 模块完成 MAC 层的功能，提供对上层的支持；physical 模块和 radiocontrol 模块共同完成物理层的功能。

MAC433 库对外接口主要由 mac 模块提供，physical 不对外提供任何接口，只是被 mac 和 radiocontrol 模块调用。radiocontrol 模块在初始化的时候会调用 cc1000 模块完成射频芯片的一些配置工作，在 tx 和 rx 切换的时候也会调用 cc1000 模块，还有一点需要说明的是 radiocontrol 模块在载波监听的时候会调用 adc 模块，adc 转换结束的时候会回调 radiocontrol 模块中的 RadiocontrolRSSIADCDataReady 函数，在该函数中会根据 adc 转换获得的数据来判断信道是否空闲。

下面主要是说明 MAC433 库的接口规范。

MAC 层函数说明

MACInit

MAC 层初始化函数，主要是对一些数据结构和控制变量的初始化

语法

result_t MACInit(void);

参数

无

返回值

SUCCESS

前提

无

副作用

无

注

在该过程中会实现物理层的初始化

示例

```
……
result_t Init(void) {
    ……
    uartDebug_init();
    MACInit();
    ……
}
```

库中定义了两个入口函数，分别为发送单播数据包的函数：

MACUnicastMsg

发送数据包到指定的下一个节点（需要路由支持获得下一跳的地址）。

语法

result_t MACUnicastMsg(void* data, uint8_t length, uint8_t toAddr);

参数

data　　需要传送的数据包 OSMACMsg 的首地址

length　数据包中有效数据的长度，可以参考数据包格式声明部分的讲解

toAddr　数据包的下一跳地址

返回值

SUCCESS

前提

需要路由提供下一跳的地址

副作用

无

注

该函数由上层调用，实现数据包的定向发送

示例

```
……
MACUnicastMsg(&Queuemsgqueue[dequeue_next].Msg,
Queuemsgqueue[dequeue_next].length,Queuemsgqueue[dequeue_next].address);
……
```

和发送广播数据包的函数

MACBroadcastMsg

将数据包广播

语法

result_t MACBroadcastMsg(void* data, uint8_t length);

参数

data 需要传送的数据包 OSMACMsg 的首地址

length 数据包中有效数据的长度，可以参考数据包格式声明部分的讲解

返回值

SUCCESS

前提

无

副作用

无

注

该函数由上层调用，实现数据包的广播发送

示例

```
……
MACBroadcastMsg(&Queuemsgqueue[dequeue_next].Msg,
Queuemsgqueue[dequeue_next].length);
……
```

上面是输入流函数，接下来要介绍的输出流函数，也就是需要用户实现的回调函数：

ReceiveDone

在接收数据包完成时通知上层处理接收到的数据

语法

result_t　ReceiveDone(OSMACMsgPtr packet);

参数

packet　　接收缓存的指针

返回值

用户定义

前提

无

副作用

无

注

该函数由上层实现

示例

```
……
if (packet->toAddr == OS_LOCAL_ADDRESS) {
……
ReceiveDone(packet);
……
```

TransmitDone

通知上层数据包发送完成

语法

result_t TransmitDone (OSMACMsgPtr packet);

参数

packet 发送缓存的指针

返回值

用户定义

前提

无

副作用

无

注

该函数由上层实现

示例

……

MACTxReset();

……

TransmitDone(dataPkt);

……

SendFail

通知上层发送数据包失败

语法

void SendFail(void);

参数

无

返回值

无

前提

无

副作用

无

注

该函数由上层实现

示例

```
……
if (state == IDLE) {
……
} else {
……
SendFail()；
```

需要注意的是这三个函数由用户自己来定义，当然由于 mac 模块要调用这三个函数，声明这几个函数的头文件需要包含在 mac.h 中。

物理层的功能

物理层包括 physical 模块和 radiocontrol 模块，这两个模块控制相关的硬件资源一起实现物理层的功能，下面介绍一下物理层的功能函数。

PhysicalInit

初始化 physical 模块，在初始化该模块的时候会实现 radiocontrol 模块的初始化

语法

result_t PhysicalInit(void);

参数

无

返回值

SUCCESS

前提

无

副作用

无

注

该函数由 MAC 层的初始化函数调用

示例

```
result_t MACInit()
{
……
RandomInit();
//mac 层 tick 时钟初始化
ClockStart();
    // 初始化物理层
PhysicalInit();
……
}
```

RadiocontrolInit

初始化 radiocontrol 模块，在初始化 radiocontrol 模块的时候会配置 CC1000 射频芯片。

语法

result_t RadiocontrolInit()；

参数

无

返回值

SUCCESS

前提

无

副作用

无

注

该函数由 physical 模块的初始化函数调用

示例

```
result_t PhysicalInit(void){
……
    recvBufState = FREE;
    procBufState = FREE;
    RadiocontrolInit();
    return SUCCESS;
}
```

PhysicalTxPkt

物理层发送数据包接口函数，进入发送状态，启动整个发送流程。

语法

result_t PhysicalTxPkt(void* packet, uint8_t length);

参数

packet　发送缓存区指针

length　数据包长度，包括 MAC 数据包的包头和包尾以及有效数据的长度

返回值

SUCCESS　成功

FAIL　　失败

前提

无

副作用

无

注

该函数由 MAC 层调用，当监听到信道空闲的时候由 MAC 层调用该函数启动一次数据包的传输过程

示例

```
static void startBcast()
{
    PhysicalTxPkt(dataPkt, txPktLen);      // 调用物理层发送接口发送数据包
    radioState = RADIO_TX;
    state = TX_PKT;

}
```

一般来说物理层的函数都是内部函数，用户不能够直接调用，但是为了支持射频休眠方式，我们的库对外提供了直接对射频进行控制的接口。

RadiocontrolIdle

射频控制进入空闲状态，CC1000、SPI 进入接收状态，射频部分处于工作状态。

语法

result_t RadiocontrolIdle(void);

参数

无

返回值

SUCCESS　成功

前提

无

副作用

无

注

启动射频部分

示例

```
result_t RadiocontrolIdle()
{
    ……
    if (state == SLEEP) {                                          // 打开 CC1000
        CC1000ControlStdControlStart();
    } else {
        * (volatile unsigned char *)(0x0D + 0x20) = 0x00;      // 关 SPI 和 SPI 中断
    }
    CC1000ControlRxMode();                                     // CC1000 设为接收模式
        // 配置 SPI 为输入
    OSH_MAKE_MISO_INPUT();
    OSH_MAKE_MOSI_INPUT();
    * (volatile unsigned char *)(0x0D + 0x20) = 0xc0;          // 开 SPI 和 SPI 中断
……
}
```

RadiocontrolSleep

射频控制进入睡眠状态，CC1000、SPI 进入关闭状态

语法

result_t RadiocontrolSleep();

参数

无

返回值

SUCCESS 成功

前提

无

副作用

无

注

关闭射频部分

示例

```
result_t RadiocontrolSleep()
{
    if (state == SLEEP)
        return SUCCESS;
    * (volatile unsigned char *)(0x0D + 0x20) = 0x00;      // 关 SPI 和 SPI 中断
    CC1000ControlStdControlStop();                          // 关 CC1000

    state = SLEEP;
    return SUCCESS;
}
```

辅助模块

辅助模块包括控制灯的模块led，控制0号计数器的模块timer，控制传感器的模块sensor，控制模数转化的模块 adc，对单片机寄存器的操作模块 fun 以及任务调度模块 os，下面逐个介绍。

1) led 模块

led.h 中声明的函数接口：

```
result_t LedInit(void);
result_t LedRedOn(void);
result_t LedRedOff(void);
result_t LedRedToggle(void);
result_t LedGreenOn(void);
result_t LedGreenOff(void);
result_t LedGreenToggle(void);
result_t LedYellowOn(void);
result_t LedYellowOff(void);
result_t LedYellowToggle(void);
```

我们提供的硬件平台上面有红，绿和黄三种颜色的指示灯各一个，函数 Led***On 表示让某种颜色的灯亮，Led***Off 表示让某种颜色的灯灭，Led***Toggle 表示让某一颜色的灯交替亮灭，即对亮着的灯调用一次会让它灭，对没有亮着的灯调用一次会让它亮，LedInit 是对整个模块的初始化，主要是一些控制变量的初始化。

2) timer 模块

该模块通过软件对 8 位的 0 号寄存器进行控制，用软件虚拟化出 2 个 32 位的计数器，它们的中断处理函数为，Timer0_0_Fired()和 Timer0_1_Fired()，响应函数如下：

```
static result_t TimerTimerFired(uint8_t id)
{
    unsigned char result;
    switch (id) {
    case 1:     //hladd
        result = Timer0_1_Fired();
```

```
            break;
        case 0:
            result = Timer0_0_Fired() ;
            break;
        default:
            result = TimerDefaultFired(id);
        }
        return result;
}
```

Timer0_0_Fired()和 Timer0_1_Fired()函数也是需要用户自己定义的，用户通过调用 TimerTimerStart(uint8_t id,char type, uint32_t interval)函数来启动相关的虚拟计数器，参数 id 表示是那一个计数器，如果 id 为 0，其响应函数是 Timer0_0_Fired()，如果是 1，其响应函数是 Timer0_1_Fired()。type 是指计数器的类型，timer.h 文件中的一个枚举类型给出了它的选项，type 可以选择 TIMER_REPEAT 和 TIMER_ONE_SHOT 两种类型，TIMER_REPEA 表示该计数器的计数值到达以后会重新开始计数，TIMER_ONE_SHOT 表示该计数器计数值到达后就结束了不会从来，顺便提一下的就是枚举变量中的 NUM_TIMERS 表示虚拟计数器的个数。Interval 是计数值，尽管 0 号计数器是一个 8 位的计数器，但是通过软件的虚拟，其计数值可以达到 32 位。

```
enum {
  TIMER_REPEAT = 0,
  TIMER_ONE_SHOT = 1,
  NUM_TIMERS = 2
};
```

timer 模块的函数接口：

TimerStdControlInit

初始化 0 号计数器，主要是对一些控制变量以及寄存器的初始化

语法

result_t TimerStdControlInit(void);

参数

无

返回值

SUCCESS　成功

前提

无

副作用

无

注

启动 0 号计数器

示例

```
result_t Init(void) {
……
    SensorPhoOStdControlInit();
    TimerStdControlInit();
    // 初始化 UART debugging
    uartDebug_init();
    MACInit();
……
}
```

TimerTimerStop

停止指定的虚拟计数器，id 的意义参考 timer 模块的介绍

语法

result_t TimerTimerStop(uint8_t id);

参数

id　参考 timer 模块的介绍解释

返回值

SUCCESS　成功

前提

无

副作用

无

注

无

示例

```
result_t TimerTimerStop(uint8_t id)
{

    if (id >= NUM_TIMERS)
    {
        return FAIL;
    }
    if (TimerM_mState & (0x1 << id)) {
        { uint8_t atomicState = AtomicStart();
        TimerM_mState &= ~(0x1 << id);
        AtomicEnd(atomicState); }
        if (!TimerM_mState) {
            TimerM_setIntervalFlag = 1;
        }
        return SUCCESS;
    }
    return FAIL;
}
```

TimerTimerStart

启动某个虚拟计数器

语法

result_t TimerTimerStart(uint8_t id, char type, uint32_t interval);

参数

id　　参考 timer 模块的介绍

type　参考 timer 模块的介绍

interval　计数器跳数

返回值

SUCCESS　成功

前提

无

副作用

无

注

无

示例

……

TimerTimerStart(0, TIMER_REPEAT, 2000);

……

3)　sensor 模块和 adc 模块

这两个模块的联系比较紧密，adc 模块的初始化是由 sensor 的初始化过程主动调用的。sensor 模块提供温度传感器和光强传感器的调用接口，由于温度传感器的变化不是很敏感，这里我们只提供光强传感器的调用接口，温度传感器的接口不起作用。下面给出温度传感器的调用接口：

SensorPhoOStdControlInit

光传感初始化,由上层调用

语法

result_t SensorPhoOStdControlInit(void);

参数

无

返回值

SUCCESS　成功

前提

无

副作用

无

注

无

示例

```
result_t Init(void) {
……
SensorPhoOStdControlInit();
……
}
```

SensorPhoOStdControlStart

启动光传感,由上层调用

语法

result_t SensorPhoOStdControlStart(void);

参数

无

返回值

SUCCESS　成功

前提

无

副作用

无

注

无

示例

result_t Start(void) {

……

SensorPhoOStdControlStart();

……

}

SensorInternalPhotoADCDataReady

接收采集数据，由 adc 模块调用，当 ADC 模块采集传感器数据结束后调用该函数

语法

result_t SensorPhoOStdControlStart(void);

参数

无

返回值

SUCCESS　成功

前提

无

副作用

无

注

无

示例

```
static result_t ADCDataReady(uint8_t port, uint16_t value){
……
    case OS_ADC_PHOTO_PORT:
        result = SensorInternalPhotoADCDataReady(value);
        break;
……
}
```

SensorExternalPhotoADCDataReady

调用上层数据接收函数 SensordataReady 将 adc 采集到的数据交给外部模块

语法

result_t SensorExternalPhotoADCDataReady(uint16_t data);

参数

无

返回值

SUCCESS　成功

前提

无

副作用

无

注

在该函数中需要调用外部函数

示例

```
result_t SensorExternalPhotoADCDataReady(uint16_t data)
{
    unsigned char result;
```

```
    result =   SensordataReady(data);
    return result;
}
```

SensorExternalPhotoADCGetData

启动一次数据采集，由上层调用

语法

result_t SensorExternalPhotoADCGetData(void);

参数

无

返回值

SUCCESS　成功

前提

无

副作用

无

注

无

示例

```
result_t Timer0_1_Fired(void) {
    //采集光数据
    SensorExternalPhotoADCGetData();
    return SUCCESS;
}
```

SensorInternalPhotoADCGetData

调用 ADC 开始采集数据

语法

result_t SensorInternalPhotoADCGetData(void);

参数

无

返回值

SUCCESS　成功

前提

无

副作用

无

注

非对外接口函数

示例

```
result_t SensorInternalPhotoADCGetData(void)
{
    unsigned char result;
    result = ADCGetData(OS_ADC_PHOTO_PORT);

    return result;
}
```

SensorPhoOStdControlInit 初始化主要是 sensor 一些全局变量的初始化，并且在初始化的过程中调用 adc 模块的初始化函数和端口绑定函数，下面是它的源代码：

```
  result_t SensorPhoOStdControlInit(void)
  {
    ADCBindPort(OS_ADC_PHOTO_PORT, OSH_ACTUAL_PHOTO_PORT);
    { uint8_t atomicState = AtomicStart();
      {
```

```
        PhotoTempM_state = PhotoTempM_IDLE;
      }
      AtomicEnd(atomicState); }
    return ADCControlInit();
  }
```

SensorPhoOStdControlStart 函数负责一些和传感器相关的一些引脚的配置(指的是单片机的引脚)，配置完成后传感器就开始工作了。

下面的 4 个函数是用来实现传感器数据的采集的，传感器数据要通过 ADC 通道进入单片机的 A/D 转换器，这样完成一次数据的采集。首先由外部调用 SensorExternalPhotoADCGetData 函数，该函数调用 SensorInternalPhotoADCGetData 函数，SensorInternalPhotoADCGetData 会调用 adc 模块中的函数 ADCGetData，并且将函数参数设置为 OS_ADC_PHOTO_PORT，adc 模块被启动，开始数据的采集过程，采集完成后发生 adc 转换完成中断，中断任务中调用 ADCDataReady 函数，在该函数中将采集到的数据 value 按照调用时的端口号发送给外部的不同函数进行处理。下面是 ADCDataReady 函数的源代码。

```
  result_t ADCDataReady(uint8_t port, uint16_t value){
    unsigned char result;
    switch (port) {
        case OS_ADC_CC_RSSI_PORT:
        result = RadiocontrolRSSIADCDataReady(value);
        break;
        case OS_ADC_PHOTO_PORT:
        result = SensorInternalPhotoADCDataReady(value);
        break;
        case OS_ADC_TEMP_PORT:
        result = SensorInternalTempADCDataReady(value);
        break;
        default:
        result = 0;
      }
    return result;
```

}

根据调用 adc 模块时传递的端口参数传感器模块的相应函数被调用，这里是 SensorInternalPhotoADCDataReady 函数，该函数调用 SensorExternalPhotoADCDataReady 函数，下面是 SensorExternalPhotoADCDataReady 函数的代码，该函数调用外部处理函数将 adc 采样的数据交给了外部处理函数 SensordataReady。

```
result_t SensorExternalPhotoADCDataReady(uint16_t data)
{
    unsigned char result;
    result =   SensordataReady(data);
    return result;
}
```

上面是整个数据采样的过程，需要注意的一点是 adc 模块对外的接口函数(这里指的是和数据采样相关的函数)是 ADCGetData(uint8_t port)，它的参数就是端口号，在载波监听的时候会调用该函数实现载波监听，这是日 radiocontrlo 模块中调用该函数实现物理信道的采样的语句 ADCGetData(OS_ADC_CC_RSSI_PORT)，其过程和传感器的采样过程差不多，只不过相应的回调函数不同， RadiocontrolRSSIADCDataReady(value)是监听过程的回调函数，该函数在 radiocontrol 模块中定义。

4) fun 模块

该模块主要提供对单片机 atmega128 一些寄存器的操作以及随机数的产生以及获得电源的一些信息的函数(有关电源的这些函数基本上没有用到)。

5) os 模块

该模块实现的是一个小型的操作系统，基本功能是提供一个任务队列，用户可以通过函数 OSPostTask 来提交一个任务，该函数的参数是一个返回值和参数都为 void 的函数名。另为 os 模块还提供一些原子操作的支持，通过一对函数来实现该功能，下面是它们的源代码。

```
void   AtomicEnd(uint8_t oldSreg)
{
    * (volatile unsigned char *)(0x3F + 0x20) = oldSreg;
}
uint8_t   AtomicStart(void )
{
```

```
        uint8_t result = * (volatile unsigned char *)(0x3F + 0x20);
        __asm volatile ("cli");
        return result;
    }
```

一次原子操作的过程如下：

```
intEnabled = AtomicStart();
其它代码
AtomicEnd（intEnabled）;
```

此外 os 模块还有开全局中断的功能函数 EnableInterrupt，对这个函数需要提一下的就是单片机在开机后的全局中断是关闭的，用户如果要使用中断的话需要在初始化的时候开全局中断。关于 os 的跟进一步的讨论留在下一部分。

开发模板及其架构

为了方便用户的开发我们提供给用户一个开发模板，通过开发模板，用户可以很快地掌握开发的流程，把握系统的整体架构，随意的增加自己的功能，实现自己的协议。下面介绍一下开发模板的架构，让用户有一个对系统的整体架构的了解。我们从 main 函数开始。

```
int main(void)
{
    MainHardwareInit();
    OSSchedInit();

    MainStdControlInit();
    MainStdControlStart();

    EnableInterrupt();

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

```
    return 1;
  }
```

MainHardwareInit()函数是在 os 中定义的，作用是定义芯片的输入输出引脚，相关的源代码如下：

```
result_t MainHardwareInit(void)
{
    OSH_SET_PIN_DIRECTIONS();
    return SUCCESS;
}

void OSH_SET_PIN_DIRECTIONS(void )
{
    OSH_MAKE_RED_LED_OUTPUT();
    OSH_MAKE_YELLOW_LED_OUTPUT();
    OSH_MAKE_GREEN_LED_OUTPUT();

    OSH_MAKE_CC_CHP_OUT_INPUT();

    OSH_MAKE_PW7_OUTPUT();
    OSH_MAKE_PW6_OUTPUT();
    OSH_MAKE_PW5_OUTPUT();
    OSH_MAKE_PW4_OUTPUT();
    OSH_MAKE_PW3_OUTPUT();
    OSH_MAKE_PW2_OUTPUT();
    OSH_MAKE_PW1_OUTPUT();
    OSH_MAKE_PW0_OUTPUT();

    OSH_MAKE_CC_PALE_OUTPUT();
    OSH_MAKE_CC_PDATA_OUTPUT();
```

```
    OSH_MAKE_CC_PCLK_OUTPUT();
    OSH_MAKE_MISO_INPUT();
    OSH_MAKE_SPI_OC1C_INPUT();

    OSH_MAKE_SERIAL_ID_INPUT();
    OSH_CLR_SERIAL_ID_PIN();

    OSH_MAKE_FLASH_SELECT_OUTPUT();
    OSH_MAKE_FLASH_OUT_OUTPUT();
    OSH_MAKE_FLASH_CLK_OUTPUT();
    OSH_SET_FLASH_SELECT_PIN();

    OSH_SET_RED_LED_PIN();
    OSH_SET_YELLOW_LED_PIN();
    OSH_SET_GREEN_LED_PIN();
}
```

以上关于引脚配置的函数的原形在 fun 模块中，参考我们给的节点的原理图会有更好的了解。

OSSchedInit()函数初始化操作系统得任务队列。

MainStdControlInit()函数会调用外部的 Init 函数完成一些模块的初始化工作，一般来说这些模块是：

- LedInit();

  LedRedOff();

  LedGreenOff();

  LedYellowOff();

  灯的初始化
- SensorPhoOStdControlInit();

  光强传感器的配置初始化，主要是端口的绑定工作还没有实现对传感器的启动
- TimerStdControlInit();

  0 号计数器的配置初始化

- uartDebug_init();

  串口的初始化

- MACInit();

  mac 层的初始化

用户可以根据自己的功能要求选择相应得模块初始化。

MainStdControlStart()函数会会调用外部的函数 Start 来启动一些模块的功能，一般来说这些模块是：

- SensorPhoOStdControlStart();

  实现了光强传感器的启动

- TimerTimerStart(0, TIMER_REPEAT, 1000);

  实现 0 号计数器虚拟的 0 号 id 计数器的启动

- TimerTimerStart(1, TIMER_REPEAT, 2000);

实现 0 号计数器虚拟的 1 号 id 计数器的启动

这些也是可选的。

EnableInterrupt()函数开全局中断，下面的 while 循环是操作系统的死循环，如果用户不想使用操作系统的话，这一部分可以修改，但是 main 函数上面的部分不要有任何的改动，因为 main 函数中上面的部分和硬件的初始化密切相关，不能改动。如果采用操作系统的方式的话就需要一些任务的种子，这个一般是由中断来产生的，如果采用非操作系统的方式就和普通的程序一样一条一条语句执行。下面以我们提供的操作系统方式下模版来实现一个定时采集传感器数据，并且发送给 sink 节点的单跳组网的例子，sink 节点可以将各个节点采集的数据通过 Uart0 口传给主机。

首先我们必须声明 OS_LOCAL_ADDRESS（本地地址 8 位）和 OS_BCAST_ADDR（广播地址 8 位），因为这两个变量在 fun 模块和 mac 模块中会被调用，另外各个模块对外调用的几个函数也需要声明：

- result_t Init(void);

  result_t Start(void);

上面的两个函数由 os 模块调用，这两个函数的一些情况在上面已经详细的讨论过了，这些函数声明在我们定义的 app.h 中（以下相同），为了让 os 模块识别这两个函数，需要在 os.h 中添加 app.h 文件的声明。

- result_t Timer0_0_Fired(void);

result_t Timer0_1_Fired(void);

上面的两个函数由 timer 模块调用，需要在 timer.h 头文件中添加 app.h 文件

✧ result_t SensordataReady(uint16_t data);

上面的函数由 sensor 模块调用，需要在 sensor.h 头文件中声明

✧ result_t ReceiveDone(OSMACMsgPtr packet);

result_t TransmitDone(OSMACMsgPtr msg);

void SendFail(void);

上面的函数由 mac 模块调用，需要在 mac.h 头文件中声明

当然这些函数的原形还是需要在 app.c 文件中定义的，在我们所给的例子中用到了指示灯模块，因此调用了 led 模块的 LedInit 函数对其进行初始化；用到了传感器模块，因此调用了 SensorPhoOStdControlInit 函数对其进行初始化，sensor 初始化的时候会调用 adc 的初始化过程，因此实际上 adc 也被初始化了；需要用到 0 号计数器实现定时采集，因此调用 TimerStdControlInit 函数实现初始化过程；需要用到无线发射模块，因此需要调用 MACInit 函数，提供射频的功能；另外可能需要 Uart 口同主机通讯，因此调用 uartDebug_init 函数初始化 Uart 口。

在 Start 函数中调用 SensorPhoOStdControlStart 模块启动光强传感器，调用 TimerTimerStart 函数启动计数器实现定时采样，该函数的参数为(0, TIMER_REPEAT, 125)，即虚拟的 0 号计数器，重复计数，由于采用的是 32 分频而且 0 号计数器的驱动时钟为外部的 32.768k 的晶振，差不多是 100ms 采样一次，我们的数据包格式为：

```
#define BUFFER_LEN    10
typedef struct{
    uint8_t seqNo;
    int16_t data[BUFFER_LEN];
}SensorMsg;
```

一个 BUFFER 放一个采样数据，每个数据包可以放 10 个，也就是说基本上是 1 秒钟完成一个数据包的采集，并且通过无线模块发送(sink 节点除外)，数据包格式在 message.h 中定义。中断每产生一次，中断响应调用 Timer0_0_Fired 函数，该函数会调用 SensorExternalPhotoADCGetData() 函数开始数据采集，数据采集结束后会调用 SensordataReady(uint16_t data)函数，将采集到的数据 data 传送出来交给外部处理。在我们提供的例子中，根据节点是否为 sink 节点决定抛弃还是处理该数据，通过源代码大家可以很

好的理解这一过程。当数据包被填满了后，通过语句 OSPostTask(LLCdataTask)在操作系统的任务队列中加入一个任务，当操作系统执行该任务的时候就会调用 LLCdataTask 函数，下面的过程大家可以从源代码中清除的看出来，最后在 LLCDataMsgSend 函数中通过语句 MACBroadcastMsg(data, length)调用 mac 模块的接口实现传送，如果该数据包被发送出去了，mac 模块就会回调 TransmitDone 函数，通知包发送完成，如果发送失败则回调 SendFail 函数。如果接收到一个数据包，如果是 sink 节点就启动 Uart 口将数据包传送给主机，否则什么都不作。关于 Uart 口的控制，我们的源码是公开的，用户可以自己配制。

winavr 编译器的 makefile 文件的设置

其它的部分和平时使用的是一样的，只是在链接参数设置一项需要将我们的库加入。

```
# 链接参数设置 Linker flags.
#  -Wl,...:     tell GCC to pass this to linker.
#    -Map:      create map file
#    --cref:    add cross reference to  map file
LDFLAGS = -Wl,-Map=$(TARGET).map,--cref
LDFLAGS += $(EXTMEMOPTS)
LDFLAGS += $(PRINTF_LIB) $(SCANF_LIB) $(MATH_LIB)
LDFLAGS += -LE:/lib223 -l433mac –lall
```

//需要加入的部分，这里假设我们的库放在 E 盘的 lib223 目录下，库的名字为 lib433mac.a（mac 部分的库）和 liball.a（辅助模块的库），按照上面的设置就 ok 了，需要注意的是-L 后面紧接着的是库的目录，中间没有空格，-l（小写的 L）后面紧跟着库名但是前缀 lib 和后缀.a 不需要。

上面我们公开的简体版的库，我们会继续升级我们的库，希望大家使用愉快，对开发中的一些问题和建议请及时反馈给我们,谢谢！

E_Mail：jiangwenfeng@ict.ac.cn