



# 《无线传感器网络技术》讲义

宁波中科无线通信事业部  
<http://www.wsn.org.cn>

宁波中科无线通信事业部  
<http://www.wsn.org.cn>

## 第十二章、操作系统

2007年8月20日



中国科学院计算技术研究所

Institute of Computing Technology, Chinese Academy of Sciences



# 内容提要

1. WSN 操作系统概述
2. TINYOS 操作系统
3. MANTIS 操作系统
4. SOS 操作系统
5. 三种操作系统的设计实现比较
6. WSN OS最新研究进展
7. 主要参考文献

1. WSN 操作系统概述
2. TINYOS 操作系统
3. MANTIS 操作系统
4. SOS 操作系统
5. 三种操作系统的设计实现比较
6. WSN OS最新研究进展
7. 主要参考文献

## WSN操作系统的设计目标

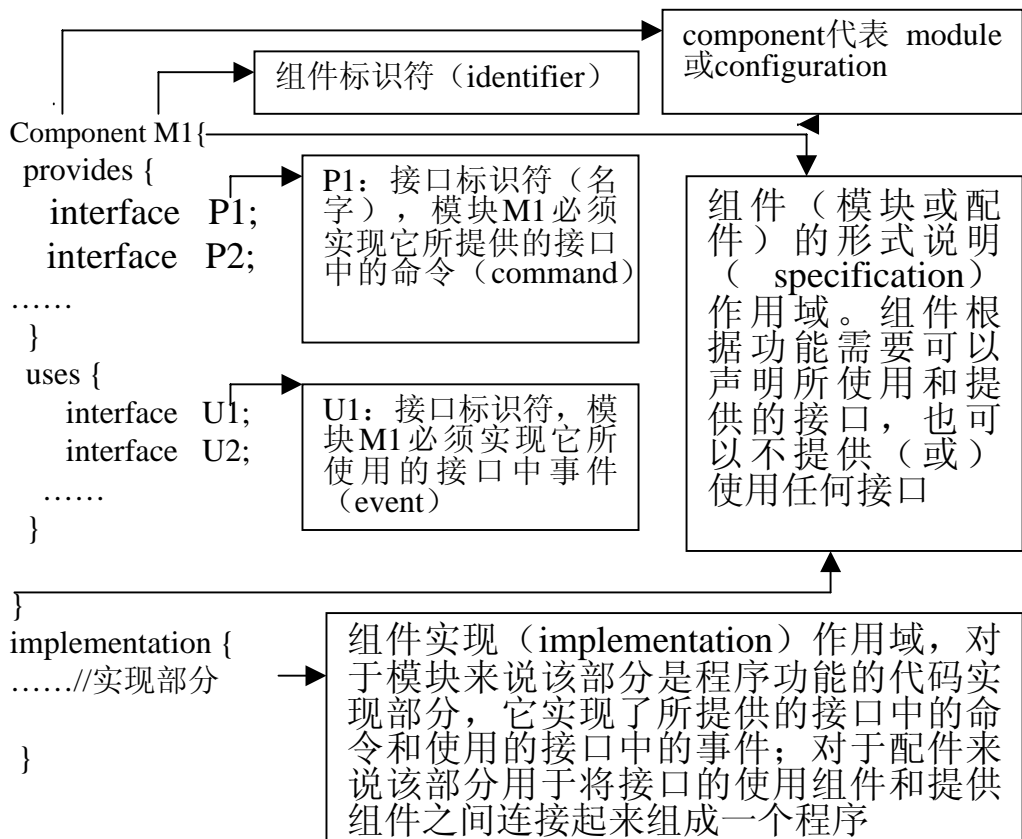
- **WSN是应用相关的网络**，其硬件节点功能、结构、组织方式会随应用而不同。由此要求WSN操作系统能够良好的模块化设计，使应用/协议/服务与硬件资源之间可以随意搭配
- **WSN节点资源非常有限**（通信带宽资源/能量资源/计算资源），操作系统必须能够高效地使用各种资源
- **WSN是一个网络系统**，其操作系统也必然是面向网络化开发的。网络化系统要求操作系统必须为应用提供高效的组网和通信机制

- 本章讲解三个有代表性的开源的无线传感器网络操作系统：
  - **Tiny OS 2.0**: 美国加州大学伯克利分校开发
  - **Mantis OS 0.9.5** (Multimodal Networks of In-situ Sensors): 美国克罗拉多大学开发
  - **SOS 1.7**: 美国加州大学洛杉矶分校开发

1. WSN操作系统概述
2. TINYOS 操作系统
3. MANTIS 操作系统
4. SOS 操作系统
5. 三种操作系统的设计实现比较
6. WSN OS最新研究进展
7. 主要参考文献

- 事件驱动，基于组件
- 使用nesC编写
- 支持的平台：eyesIFXv2、intelmote2、mica2、mica2dot、micaZ、telosb、tinynode、btnode3
- nesC:使用C作为其基础语言，支持所有的C语言词法和语法
  - 增加了**组件**（component）和**接口**（interface）的关键字定义
  - 定义了接口及如何使用接口表达组件之间关系的方法
  - 目前只支持组件的静态连接，不能实现动态连接和配置

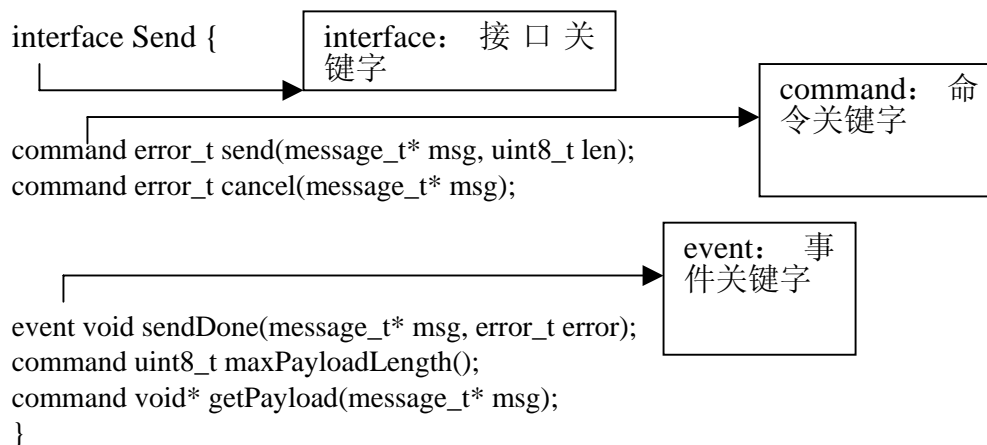
# 组件模型



- 组件（component）
  - Module组件（模件）
  - Configuration组件（配件）
- 组件特征
  - 组件内变量、函数可以自由访问，但组件之间不能访问和调用
- 组件可以提供（provide）和使用（use）接口
  - 接口是一组相关函数的集合，它是双向的并且是组件间的唯一访问点。接口声明了一组函数，称为命令（command），接口的提供者必须实现它们；接口还声明了另外一组函数，称为事件（event），接口的使用者必须实现它们

## 接口的特点

- Provides未必一定有组件使用，但uses一定要有人提供，否则编译会提示出错。在动态组件配置语言中uses也可以动态配置
- 接口可以连接多个同样的接口，叫做多扇入/扇出
- 一个module可以同时提供一组相同的接口，又称参数化接口，表明该Module可提供多份同类资源，能够同时给多个组件分享



## 组件命名规则

- **C和P的命名规则**：TinyOS 所有的终端程序组件都以字母C或P为结尾。以C结尾所命名的组件表示它是一个可用的抽象，而以P结尾的组件则表示它是私有的。以P结尾的组件不能被直接连接，但可以对它做一些封装以使它变成可用（变成名字以C结束的）
- **硬件平台抽象命名规则**：TinyOS 2.0中的硬件抽象通常是三级抽象架构，称作HAA(Hardware Abstraction Architecture)。
  - HAA的最底层是HPL( Hardware Platform Layer)
  - HAA的中间层是HAL (Hardware Abstraction Layer)
  - HAA的最高层是HIL (Hardware Independent Layer)

- 基本任务模型：基本任务模型中任务的原型声明如下：

```
task void taskname () {.....}
```

用户使用post关键字抛出任务，调用方式如下：

```
result_t ret = post taskname ()
```

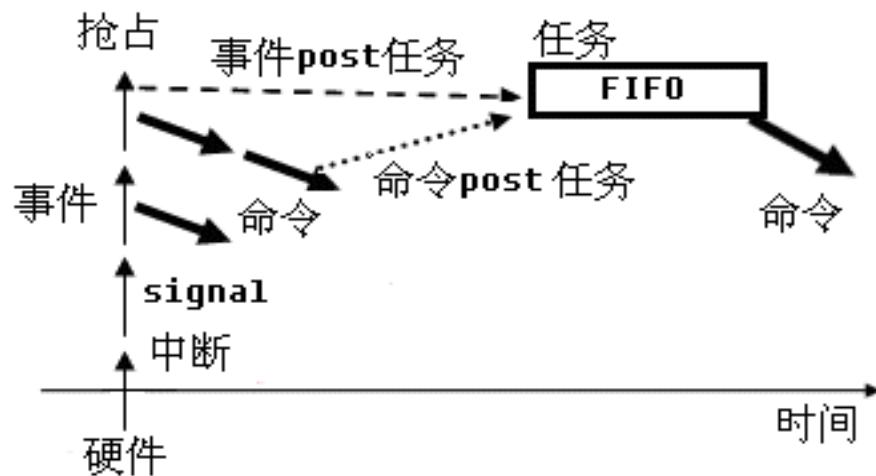
- 任务接口模型：任务接口扩展了任务的语法和语义。通常情况下，任务接口包含一个异步（async）的post命令和一个run事件，这些函数的具体声明由接口决定。实例：

```
Interface TaskParameter {  
  async error_t command postTask(uint16_t param);  
  event void runTask(uint16_t param);  
}
```

调用方式：

```
call TaskParameter.postTask(34); //抛出任务
```

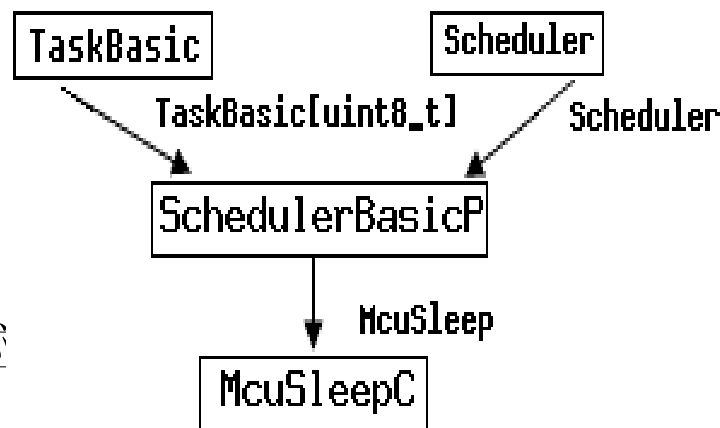
- TinyOS的调度器实现了任务和事件的两级调度
- 任务之间不能互相抢占，底层硬件中断触发事件，事件能抢占任务，事件之间也能互相抢占。命令和事件都可以post任务。任务中也可以调用命令。
- TinyOS 2.x调度器被实现为一个TinyOS组件。调度器既支持最基本的任务模型，也支持任务接口模型，并且由调度器负责协调不同的任务类型。



- TinyOS调度器的形式说明如下:

```
module SchedulerBasicP {  
    provides interface Scheduler;  
    provides interface TaskBasic[uint8_t taskID];  
    uses interface McuSleep;  
}
```

- 配件TinySchedulerC封装了组件SchedulerBasicP
- 调度器必须提供参数化的TaskBasic接口
- 调度器还必须提供Scheduler接口
- TinyOS2.0允许用户使用自己定义的应用程序（组件）取代系统调度器

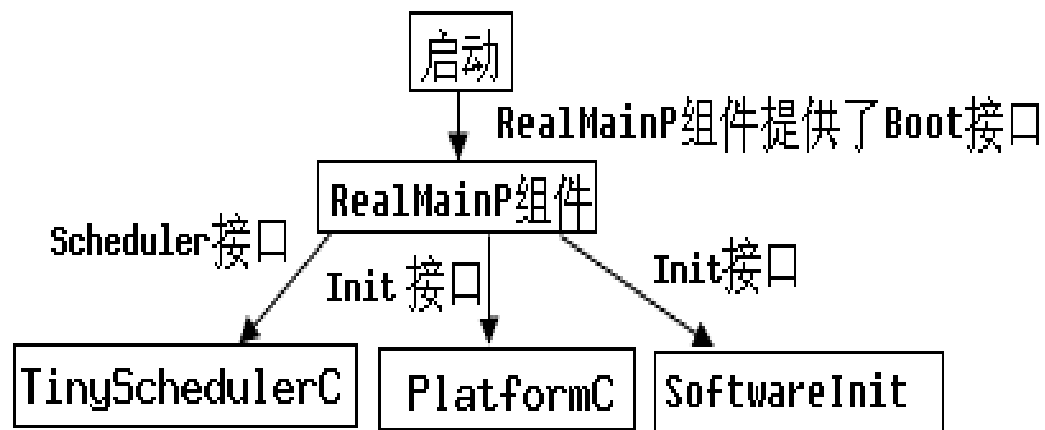


- TinyOS使用了静态的内存分配和管理机制。  
TinyOS中的组件在编译时分配所需要的内存
- 组件之间能共享状态（state）的唯一方法是通过函数调用。
- 组件传递参数使用了两种方法：值传送和指针传送。
  - 指针传递参数比较危险
  - 推荐做法：在任何时候，每一个指针都有一个明确的所有者，并且只有所有者才能修改相应的存储区。

- 消息缓冲区：TinyOS 2.0中的消息缓冲区类型是 `message_t`（与1.x不同），并且仍采用了静态包缓冲策略。缓冲区大小可以适合任何节点的通信接口，组件不能直接访问结构的各域，所有缓冲区的访问必须通过接口 `AMPacket` 和 `Packet`（定义在 `opt\tinyos-2.x\tos\intefaces` 目录）实现。
- 通信组件：用户可以使用如下四个主动消息通信组件实现无线消息的收发（定义在 `tos\tinyos-2.x\tos\system`）：  
`AMSenderC`, `AMReceiverC`, `AMSnooperC`,  
`AMSnoopingReceiverC`

- **TinyOS** 一次仅执行一个程序。程序运行时，有两个执行线程：任务和事件。事件是由硬件中断触发的，事件之间可以互相抢占
- 任务之间不互相抢占，事件可抢占任务，事件也可互相抢占。可抢占运行的函数用`async`标识，同步运行的函数用`sync`
- **nesC**的规则是：异步函数调用的命令和事件也必须是异步的。一个函数（命令或事件）不是异步就是同步（缺省）。接口的定义指明了命令和事件是异步还是同步
- 中断（异步函数）可以执行同步函数的唯一方法就是`post`一个任务
- 使用原子语句块来实现对临界数据的访问

- TinyOS 2.x的启动序列使用了3个接口：
  - **Init:** 初始化组件和硬件，它执行的是顺序的，同步的操作，在初始化完成前不会启动任何组件
  - **Scheduler:** 初始化和运行任务，用于初始化和控制任务的执行
  - **Boot:** 通知系统已经成功的启动。它定义了一个事件booted（），用它通知系统已经被成功的启动
- TinyOS 2.x中的RealMainP（定义opt\tinyos-2.x\tos\system中）实现了标准的启动序列，MainC封装了组件RealMainP





# 系统启动和初始化.....

http://www.wsn.org.cn  
宁波中科无线通信事业部

http://www.wsn.org.cn  
宁波中科无线通信事业部

- 模块RealMainP 的定义

```
module RealMainP {  
    provides interface Boot;  
    uses interface Scheduler;  
    uses interface Init as PlatformInit;  
    uses interface Init as SoftwareInit;  
}  
implementation {...}
```

- 系统的启动过程分为3个独立的初始化过程：调度器初始化（Scheduler）,平台初始化（ PlatformInit）,和软件初始化（ SoftwareInit）。
- MainC导出Boot和SoftwareInit接口，用于应用程序连接
- 不直接依赖于硬件资源的组件初始化都应该连接到 MainC.SoftwareInit

- TinyOS 2.x中的能量管理分为处理器能量管理和设备能量管理：
  - 微处理器能量管理，TinyOS 2.x使用了一个dirty位，一个芯片相关的能量状态计算函数和一个能量状态重载函数来管理和控制微处理器的能量状态
  - 外设能量管理，TinyOS 2.0定义了2种不同的能量管理模型：显式能量管理模型和隐式能量管理模型
- 如下接口用于实现设备能量管理：
  - **StdControl**: 若一个设备的开启或关闭所花费的时间可以忽略，那么它应该提供这个接口；
  - **SplitControl**: 若一个设备的开启或关闭所花费的时间不可以被忽略，那么它应该提供这个接口；
  - **AsyncStdControl**: 由于上述2个接口都是同步接口，所以若想在异步代码中控制一个设备的能量状态，那么就必须使用该接口

- TOSSIM是TinyOS的模拟器，它是一个程序库
- TOSSIM是一个离散的时间模拟器，当它运行的时候，会从时间队列中依次取出事件（以时间排序）并且执行它们
- 模拟事件可以是硬件中断也可以是高层的系统事件（例如包接收事件），任务也可以成为模拟事件
- 支持两种编程接口：Python 和C++，这两种编程接口各有优缺点，而且这两种代码之间的转换比较简单。
- TOSSIM支持的唯一平台是micaz，而且还不支持能量检测。

- 回顾组件和接口的概念
  - 组件， nesC组件使用的是一个纯局部的命名空间，这就是说一个组件除了要声明它将执行的函数外，还要声明它所调用的函数。每一个组件都有一个形式说明（specification），这个形式说明是一段代码，它声明了组件所提供（执行）接口（函数）和所使用（调用）的接口（函数）
  - 接口， 接口（interface）是相关函数的一个集合，用于可以根据功能的需要定义自己的接口，但在定义接口中的函数时，必须使用command或event关键字声明该函数是命令或是事件，否则编译时会报错

- nesC有两种组件：
  - 配件（configurations），用于将组件连接在一起从而形成一个新的组件
  - 模块（modules），提供了接口代码的实现并且分配组件内部状态，是组件内部行为的具体实现
- 模块相关概念：
  - 分段操作，分段接口的一个重要的特征就是两个阶段的调用是相反的：向下调用是开始操作，向上的signal操作是完成操作。在nesC中，向下调用的是命令，而向上调用的是事件，接口指定了这种关系的两个方面

分段接口实例:

```
interface Read<val_t>
{
    command error_t read();
    event void readDone( error_t result, val_t val );
}
```

- Read接口的提供者需要定义Read函数和通知Readdone事件，而Rend接口的使用者则需要定义Readdone事件，而且能够调用Read命令。

- 接口可以带有类型参数，接口的类型参数放在一对尖括号里，如果提供者和使用者的接口都带有类型参数，那么在连接时，它们的类型必须匹配，
- 例：  
LocalTime接口带有一个precision\_tag参数，定义在opt\tinysos-2.x\tos\lib\timer目录中。如下：  

```
interface LocalTime<precision_tag> {  
    async command uint32_t get();  
}
```
- 参数precision\_tag虽然没有在接口的命令中出现，但它在连接时会被用于类型检查：该参数指明了最小的时间间隔



## 模块实现... 形式说明

- module SenseC
- {
- uses {
- interface Boot;
- interface Leds;
- interface Timer<TMilli>;
- interface Read<uint16\_t>;//使用接口的声明
- }
- }



# 模块.....实现

```
• Implementation // 以下是实现部分
• { #define SAMPLING_FREQUENCY 100
• event void Boot.booted() {
• call Timer.startPeriodic(SAMPLING_FREQUENCY);//调用了Timer接口的命令
• }
• event void Timer.fired() //因为使用了Timer接口，所以必须实现Timer接口中定义的事件
• { call Read.read();//调用了Read接口的命令}
• event void Read.readDone(error_t result, uint16_t data) //因为使用了Read接口，所以必须
• { //实现 Read接口中定义的事件
• if (result == SUCCESS){
• if (data & 0x0004)
• call Leds.led2On();
• else
• call Leds.led2Off();
• if (data & 0x0002)
• call Leds.led1On();
• else
• call Leds.led1Off();
• if (data & 0x0001)
• call Leds.led0On();
• else
• call Leds.led0Off();
• }
• }
• }
```

- 任务
- 原子语句：在nesC中通过使用原子语句（atomic statements）的方式实现了对临界数据保护，例  
command bool increment() {  
    atomic {  
        a++;  
        b = a + 1;}  
}
- 原子代码块保证了这些变量可以被原子的写和读。注意，这里并没有保证这个原子的代码块不能被抢占。即使使用了原子代码块，两个不涉及任何相同变量的代码段还是可能互相抢占的。
- 灵活使用enum可节省存储空间

- 组件之间是完全独立的，只有通过连接才能绑定到一起，配件用于实现此功能。配件的定义与模块类似
- 使用了三个操作->,<-和=实现连接。前面两个是最基本的连接操作：箭头从使用者指向提供者。例如，下面两行是等同的：

```
Sched.McuSleep -> Sleep;
```

```
Sleep<-Sched.McuSleep;
```

- 一个直接的连接总是从使用者指向提供者，箭头的方向决定了调用关系
- 和模块一样，配件可以提供和使用接口。但是由于配件没有代码实现，所以这些接口的实现必须依赖其他的组件。

- 导通连接是指一个配件将两个组件连接到一起，并且必须使用“=”操作符把使用者连接到提供者操作符。例：

```
generic configuration AMReceiverC(am_id_t amId) {
  provides {
    interface Receive;
    interface Packet;
    interface AMPacket;
  }
  implementation {
    components ActiveMessageC;
    Receive = ActiveMessageC.Receive[amId];
    Packet = ActiveMessageC;
    AMPacket = ActiveMessageC;
  }
}
```

- 接口之间可以是n对k的关系，这里n是使用者数，k是提供者数
- 扇出（fan-out）同一个调用者一次调用了多个函数
- 扇入（fan-ins）用来描述多个人调用同一个函数。
- 接口之间是一个n对k的关系，任何提供者的signal将会引起n个使用者的事件处理函数，并且任何使用者调用一个命令将会调用k个提供者的命令
- combine函数，它将多个返回值组合后只返回一个值。一个数据类型可以有一个相关的combine函数。因为一个fan-out总是涉及到调用N个相同的函数，调用者最终得到的返回值是对所有的被调用者的返回值使用combine函数之后得到的返回值。

- 参数化接口用于提供同一接口的多个实例。如：

```
configuration ActiveMessageC {
  provides {
    interface SplitControl;
    interface AMSend[uint8_t id];
    interface Receive[uint8_t id];
    .....
  }
}
```

- 参数化接口本质上是一个接口数组，数组的索引就是接口的参数。
- nesC还提供了一个unique函数用于保证参数不重复

## 缺省连接，unique()和uniqueCount()函数

- nesC提供了缺省连接处理。如果一个组件连接到了某个接口，那么就按照该连接调用接口中的函数。若没有，则命令（或事件）会执行缺省的处理函数（使用default关键字标识），尽量不要使用缺省连接处理
- unique(): 保证每一个相同的接口必须有不同的参数ID，它将把所有的对unique()调用变换成整数标志符。unique函数需要一个字符串关键字作为参数。例  
AMQueueEntryP.Send ->  
AMQueueP.Send[unique(UQ\_AMQUEUE\_SEND)]
- uniqueCount(): 该函数是用来计算调用unique的客户总数目，这样就可以使组件能够分配正确的状态总数

通用组件不是单一实例的，它在配件内能被实例化  
通用组件与非通用组件原型定义的最大差别有两点：  
➤ 在关键字component（表示module或configuration）之前有一个generic关键字，它表示该组件是通用组件  
➤ 通用组件在组件名字后必须带有参数列表，从这方面来看类似于函数的定义。若该通用组件不需要参数，那么该参数列表为空

目前通用组件支持如下三种类型的参数：

- 类型（types）参数：这些参数可以作为类型化接口的参数，声明时使用typedef
- 数值常数(numeric constants)参数
- 字符串常数（constant strings）参数

- 使用通用组件时需要在配件中使用关键字new实例化一个通用组件，这个实例是配件所私有的。用户每使用一次new便会被会创建一个实例。在使用关键字new实例化通用组件的时候，系统使用了代码拷贝的方式生成新的实例。例如：

例如：

```
{  
n {  
0);  
}
```

为10的BitVectorC组件

- 通用模块带有三种参数，如果参数是一个类型，那么必须用typedef关键字声明。例如通用模块 VirtualizeTimerC (opt\tinyos-2.x\ tos\lib\ timer) ，如下：

```
generic module VirtualizeTimerC(typedef precision_tag, int  
max_timers)
```

```
{  
  provides interface Timer<precision_tag> as Timer[uint8_t num];  
  uses interface Timer<precision_tag> as TimerFrom;  
}  
implementation {.....}
```

在上面这个例子中生成了一个VirtualizeTimerC代码的拷贝，并且分配了uniqueCount(UQ\_TIMER\_MILLI)个毫秒精度的定时器。

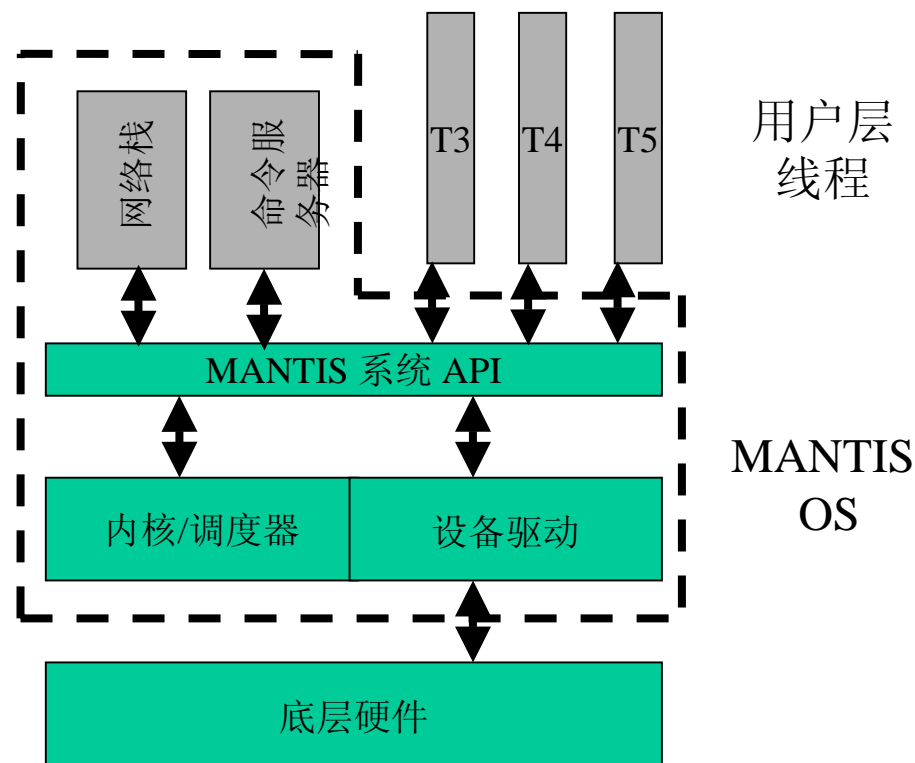
- 通用配件构成了更高层次的虚拟化和抽象。使用通用配件与使用通用模块的方法是一样的

1. WSN操作系统概述
2. TINYOS 操作系统
3. MANTIS 操作系统
4. SOS 操作系统
5. 三种操作系统的设计实现比较
6. WSN OS最新研究进展
7. 主要参考文献

- 轻量级的基于抢占的多线程无线传感器网络操作系统
- 类Unix编程环境，编程语言为c语言
- 整个内核占用的RAM小于500个字节
- 适合于无线传感器网络中处理复杂任务（例如加密解密，数据融合，定位，时间同步等）的需求
- 目前支持MICA系列的节点和MANTIS课题组研发的namph节点。

# Mantis OS系统介绍

- 应用程序线程和底层操作系统API相互独立，所以MOS通过提供不同平台的API就可以实现对多个平台的支持
- MOS系统由内核（kernel/scheduler），网络栈，通信层（COMM）以及其他组件构成（如dev驱动层）



## Mantis OS内核和调度器

- 提供了类似于UNIX风格的调度器和POSIX服务
- 基于优先级的多线程调度和在同一优先级中进行轮转调度的服务
- 时间片可配置，缺省为10ms
- 支持互斥信号量和计数信号量
- 逻辑上把RAM可以分成两部分：一部分在编译时分配给全局变量，其他部分以堆的形式管理。
- 内核的主要全局数据结构是线程表，每一个线程占一个条目
- 系统在以下情况下会引发上下文切换：调度器接收到一个来自硬件定时器的中断，系统调用和信号量操作。定时器中断是唯一一个被内核处理的硬件中断
- 没有软件中断

- 内核的主要全局数据结构是线程表，每一个线程占一个条目

```
typedef struct thread_s {  
    stackval_t *sp;           线程当前的堆栈指针  
    stackval_t *stack;       堆栈基指针  
    uint16_t  stackSize;     堆栈的大小  
    void (*func)(void);     指向线程开始函数的指针  
    uint8_t  state;         目前线程的状态  
    uint8_t  suspend_state; 当前线程的挂起状态  
    uint8_t  priority;      线程的优先级  
    uint32_t st;           线程的睡眠时间  
    uint8_t  port;         端口号（只用于网络接收）  
    struct thread_s *next;   指向线程列表中的下一个线程  
} thread_t;
```

- 线程的状态有五种：空闲状态，运行状态，就绪状态，阻塞状态，休眠状态
- 阻塞状态又分为二种：阻塞空闲状态和阻塞休眠状态
- 线程设置了5个优先级：内核优先级，休眠优先级，高优先级（能抢占所有其他的优先级的线程），正常优先级，空闲优先级。系统缺省定义的最大线程个数为6
- 内核为每一个优先级维护一个含有头指针和尾指针的就绪列表
- Idle线程是空闲优先级的线程，所以只有当其他的所有线程都处于阻塞状态时才被运行。idle线程可以检测cpu的使用情况，并且调整内核参数以节省能量。

# 内核相关数据结构定义

- 内核为每一个优先级维护一个含有头指针和尾指针的就绪列表

```
typedef struct {  
    thread_t *head;  
    thread_t *tail;  
} tlist_t;
```

- 信号量数据结构定义如下：

```
typedef struct {  
    int8_t val; //该字节用于作为互斥信号量或计数信号量  
    tlist_t q; //头指针和尾指针  
} mos_sem_t;
```

- 在任何时刻，任何一个线程或是属于就绪列表，或是属于信号量列表
- 指示当前正在运行线程的全局数据结构，定义如下：  
thread\_t\* \_current\_thread

# 系统启动过程

- 整个MOS系统是从主函数main（）开始运行的
- Main（）函数中生成了一个新线程：pre\_start（）。这个新生成线程主要的任务是初始化系统，例如串口，传感器等
- pre\_start（）中又调用了start（）函数，而start（）是用户程序的开始执行函数，所有的用户程序都应该以start（）函数开始。
- 调度器启动函数mos\_sched\_start()的定义如下：

```
void mos_sched_start(void)
{
    running = TRUE;
    kernel_timer_init();//初始化硬件定时器，这些定时器用于控制休眠时间和时间片大小的
    sleep_timer_init();
    ENABLE_INTS();//一旦中断使能了，调度器便会开始调度线程和把时间分片
    dispatcher(); //开始调度第一个线程
    idle_loop(); //若无线程可调度，调度器进入空闲状态
}
```

- 整个网络栈有四层组成 - 应用层，网络层，MAC 层和物理层
- 第三层及三层以上 由网络协议栈提供，它可被作为一个或多个用户级线程执行，提供零拷贝服务
- 网络协议的不同层可以执行在不同的线程中，或者网络协议栈中所有的层可以执行在同一个线程中
- MAC 和物理层由通信层（COMM层）提供
- COMM层为通信设备的驱动程序定义了统一接口，这些设备包括串口，无线通信设备等，它实现了异步的I/O操作，也管理数据包缓冲和同步设备驱动程序
- COMM层也实现了零拷贝操作和零轮询，高层的网络协议或用户线程可以使用COMM层提供的统一接口函数与通信设备交互

# 设备驱动层（DEV层）

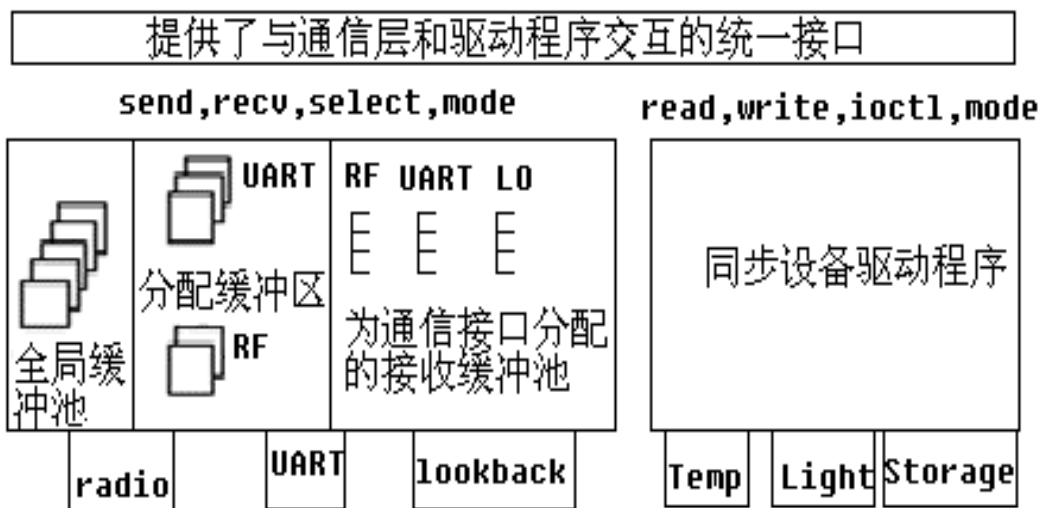
设备驱动层（dev层）涵盖了同步IO设备的驱动程序（如传感器，外部存储器等）和异步通信设备的驱动程序（如无线电，串口等），每一个设备都为上层用户提供了 **POSIX**风格的系统调用函数

**MOS**使用了一个静态表来存储每一个设备的函数指针

每一个设备都使用一个互斥信号量，帮助多个使用者互斥地使用设备

设备的状态：

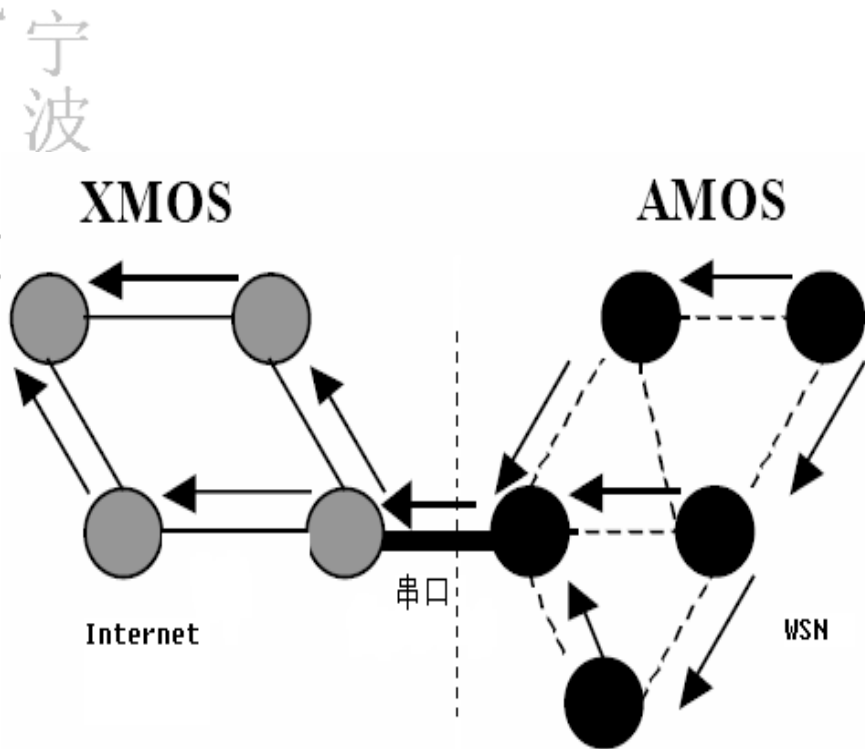
- DEV\_MODE\_OFF,
- DEV\_MODE\_IDLE,
- DEV\_MODE\_ON



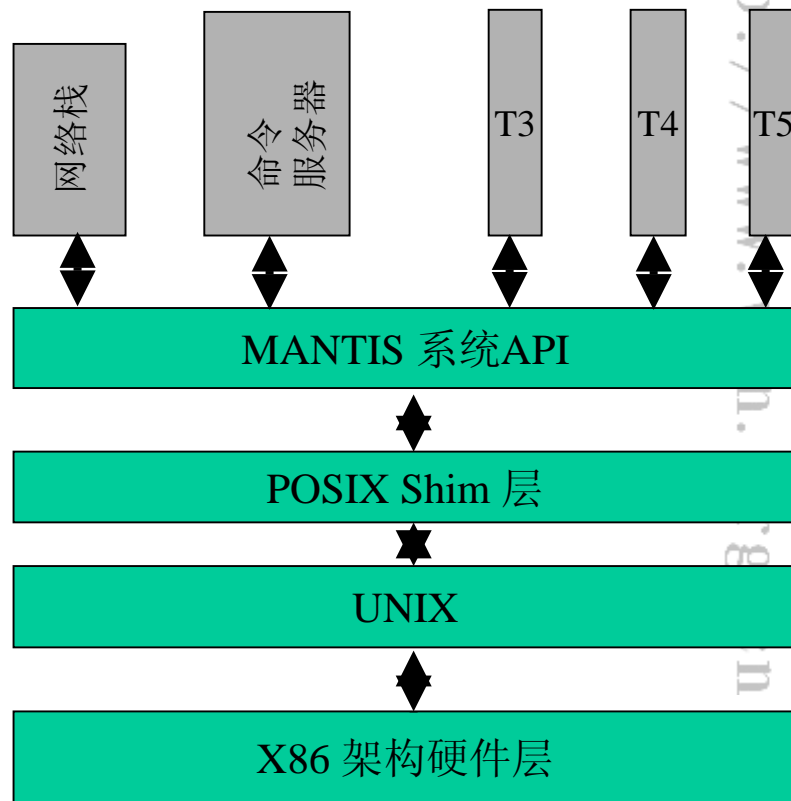
- 提供了标准的接口用来控制外部设备的能量状态，可以使用dev\_mode()函数来设定底层设备的能量状态，三种不同的能量设备状态：开启，关闭和空闲
- MOS的微处理器能量管理与线程调度是紧密结合的，并且它支持2个级别的节能：
  - 当调度器就绪队列空闲了，调度器会隐式的使微处理器进入空闲状态，此时消耗更少的能量
  - 当系统中所有的线程都通过调用了函数mos\_thread\_sleep()要求进入休眠状态时，调度器就会使处理器进入更节能的休眠状态，此状态下，只有一个定时器处于运行状态以便在休眠周期到的时候能唤醒相应的进程。

- 提供了原型环境，用于在真正配置网络前测试传感器网络应用程序
- MANTIS允许程序员在虚拟传感器节点和真实的传感器节点上测试同一个程序
- MANTIS把虚拟环境和真实的网络结合在一起，例如允许虚拟节点和物理节点共存，并且在原型环境中还可以互相通信
- MANTIS的虚拟节点可以使用MANTIS API以外的API
- MOS可以在X86（Linux 或Windows）平台上作为一个应用程序运行 (称为 XMOS)
- XMOS 不能完美的模拟传感器节点

# Mantis OS模拟器



## X86 PC





## Mantis OS 编程

- C语言作为编程语言
- MOS系统在初始化完毕后会自动调用start（）函数从而启动应用程序运行
- 可以使用MOS\_thread\_new（）生成新的子线程

## 使用无交互后台程序命令

- 提供了一种与连接在串口上的传感器节点交互的方式，用户也可以定义自己的命令
- 用来生成新线程的函数，其原形如下：  
`uint8_t mos_thread_new (void(*function_start)(void),  
uint16_t stack_size, uint8_t priority)`
- `start`函数里生成一个新的`command daemon`线程
- 使用`MOS_register_function`函数可以注册用户定制的函数：

```
bool mos_register_function (char *name,  
void(*func_pointer)(void))
```

## 设备层 ( Device Layer)

- 设备层提供了与硬件设备交互的通用接口
- 设备的驱动被定义在MANTIS/src/MOS/dev中，设备列表定义在MANTIS/src/MOS/dev /include/dev.h中
- 通过调用以下函数使用设备：

`dev_read (DEV_NAME, (uint8_t *)buf, bytes_to_read)`

`dev_write (DEV_NAME, (uint8_t *)buf, bytes_to_write)`

`dev_ioctl (DEV_NAME, REQUEST,  
[PARAMETERS...])`

`dev_open(DEV_NAME)`

`dev_close(DEV_NAME)`

- 为了使用新设备，需要做以下工作：
  1. 把设备的驱动程序（.c文件）放到 MANTIS/src/MOS/dev中，并且将它的头文件放在相应的子目录中
  2. 创建新设备所需要的每一个函数
  3. 为dev\_open()， dev\_close()这两个函数增加一个信号量
  4. 最后，把init调用放在main.c中，init应该能够初始化该设备的mutex信号量，并且执行与该设备相关的其它初始化

- **COMM**掩盖了创建缓冲区和底层不同硬件通信设备接口的细节，以方便用户使用
- 调用`comm_send`发送数据包时，需要把指向包缓冲区的指针作为参数，**COMM**层将阻塞这个发送进程，并把这个指针传递给具体的设备驱动程序，由低层的驱动程序完成数据的发送，发送完成后，被阻塞的线程会继续运行。
- 接收缓冲区是由**COMM**层自己管理的，设备的驱动程序可以申请缓冲区，并把收到的数据包放入该缓冲区。
- 应用程序线程调用`com_recv`时，它将会被阻塞，直到设备填满了缓冲区，同时会返回指向填满数据缓冲区的指针，使用完该缓冲区后必需调用`com_free_buf`（声明为静态存储的除外）来释放缓冲区



## 使用通信层.....使用comBuf

- 接收数据包时按下列方法使用comBuf:
  1. 包含源文件: `#include "com.h"`
  2. 创建一个comBuf指针, 该指针指向的缓冲区将在com\_recv()函数中分配, 如下: `static comBuf *recv_pkt;`
  3. 使设备进入监听模式, 这里需要使能接收中断以便开始接收数据, 如下: `com_mode(IFACE_NAME, IF_LISTEN);`
  4. 最后将接收到的数据放到缓冲区的负载部分, 从而完成接收, 如下: `recv_pkt = com_recv(IFACE_NAME);`
- 发送操作中, 除了需分配静态comBuf\* (或使用刚使用完的com\_recv()中) 之外, 其他方面与接收操作相同

- 休眠—使用`mos_thread_sleep`函数使线程进入休眠状态，其原型如下：  
`void mos_thread_sleep (uint32_t sleeptime)`
- 使用计时器时钟，可以使用`realtimer`来实现计时操作，它是底层硬件定时器的抽象  
`#include "realtime.h" //包含头文件`  
`real_timer_init(); //初始化为CTC模式`  
`uint32_t time;`  
`time = *real_timer_get_ticks(); //获得计时值`
- 调用`real_timer_clear()`可以重启计时器，将运行的值重新设为0即可

## 使用定时器Alarms

- Alarm用来实现节点上的定时器功能，使用alarm过程如下：
  1. 提供alarm的头文件，如下：`#include "clock.h"`
  2. 下一步需要实例化alarm——实例化`mos_alarm_t`结构（windows平台或linux平台）
  3. 下一步需要设定回调函数，当alarm触发（溢出）时该函数将被调用
  4. 设定`mos_alarm_t`的其他参数，不同平台的使用方法也不同
- 使用`mos_remove_alarm`函数移除一个alarm：  
`mos_remove_alarm (&myalarm)`

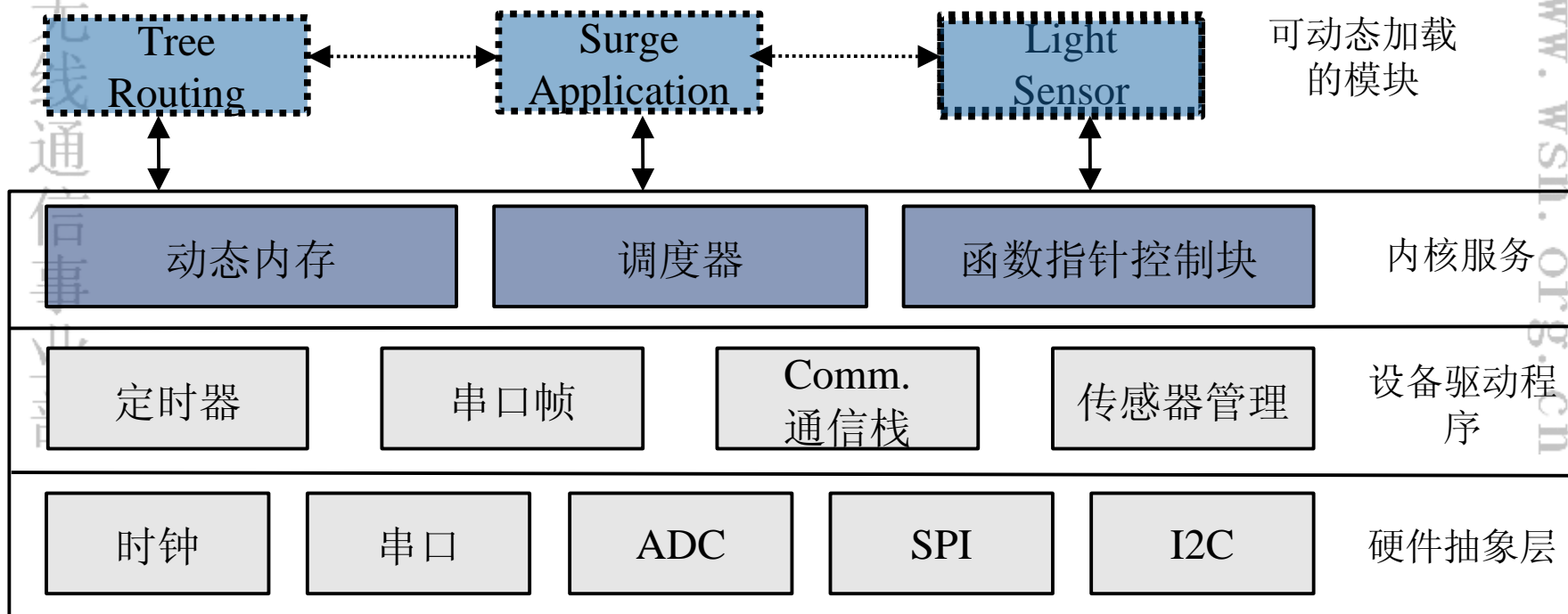
- `printf`的参数和标准系统中也有所不同：
  - `%s` – 字符串
  - `%c` – 字符
  - `%C` – 8位十进制数
  - `%d` – 16位十进制数
  - `%l` – 32位十进制数
  - `%x` – 16位的十六进制数
  - `%o` – 16位八进制数
  - `%b` – 16位二进制数
  - `%%` – `%` 字符
- MOS中也可以使用`puts`函数和`putchar`函数

1. WSN OS概述
2. TINYOS 操作系统
3. MANTIS 操作系统
4. SOS 操作系统
5. 三种操作系统的设计实现比较
6. WSN OS最新研究进展
7. 主要参考文献

- 事件驱动
- 提供了很好的动态增加和删除模块的功能
- 内核和应用程序模块中都使用动态存储
- 实现了优先级调度
- 使用标准C语言和编译器
- 目前支持Mica2, MicaZ节点和Yale的XYZ节点

# 系统架构

- 由可以动态加载的模块和静态内核组成
- 静态内核实现了最基本的服务，包括底层硬件抽象，灵活的优先级消息调度器，动态内存分配等功能
- 模块实现了系统大多数的功能，包括驱动程序，协议，应用程序等。这些模块都是独立的



- 每一个应用程序都包含一个或多个模块
- 模块本身是一个独立的代码实体，实现一项具体的任务或功能
- 只有在需要改变底层的硬件资源管理功能时才会修改SOS内核
- 在模块内部，使用了函数调用实现交互
- 模块间通过传递消息实现交互
- 模块可以维护状态，分配或回收内存



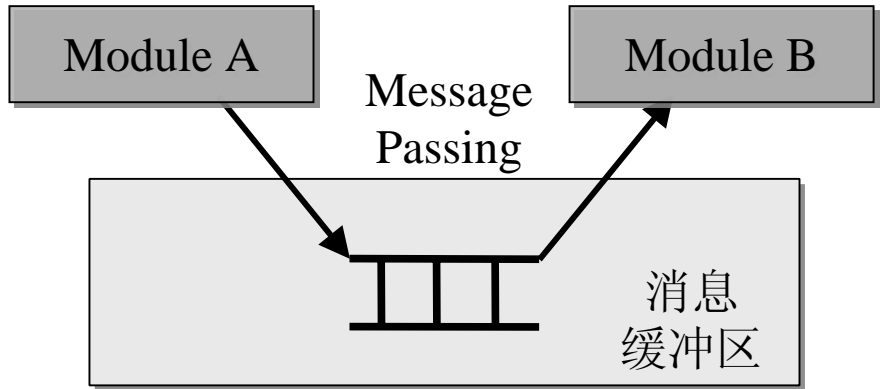
# 模块结构

```
static int8_t module(void *state, Message *msg)//消息处理函数
{
    app_state_t *s = (app_state_t *) state;
    switch (msg->type){ //下面实现了对不同消息的处理
    case MSG_INIT:
        {
            s->pid = msg->did; // 初始化模块具体的状态，如定时器等
            return SOS_OK;
        }
        .....//其他消息的处理
    case MSG_FINAL: // 释放模块占用的资源
        {
            return SOS_OK;
        }
    default:
        return -EINVAL;
    }
    return SOS_OK;
}
```

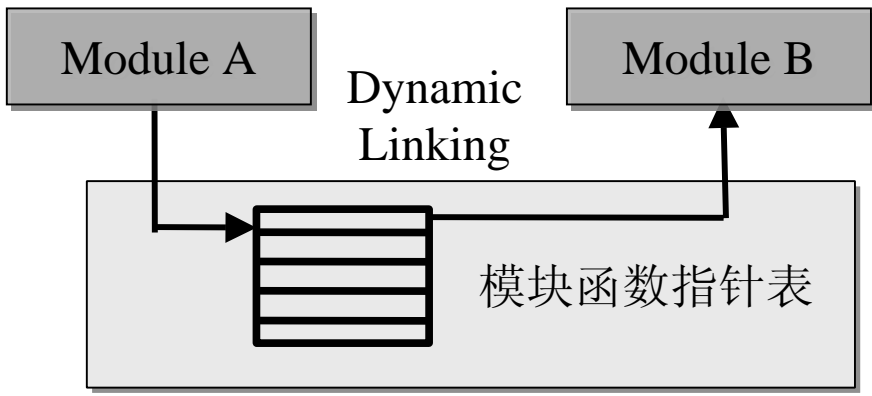
- 在下面的几种情形下，模块间会发生交互：
  1. 模块间消息传递。
  2. 对一个函数的直接调用，这个函数被另一个模块所注册。
  3. 模块调用内核函数ker-\*实现对系统内核的调用
- SOS中的消息都是异步的，模块发出一个消息后，该消息便被放入了消息队列。SOS主调度循环从优先级队列中取得消息并将消息传递给目标模块的消息处理程序
- 模块之间的直接函数调用用于实现模块间需要同步运行的操作。SOS使用了函数注册和订阅机制实现了直接函数调用

# 模块交互...

- 模块A调用通过消息调用模块B



- 模块A直接调用模块B



```
static mod_header_t mod_header SOS_MODULE_HEADER = {
    .mod_id      = TREE_ROUTING_PID, //模块的id号
    .state_size  = sizeof(tree_route_state_t), //模块状态占有的空间
    .num_timers  = 2,                //使用了2个定时器
    .num_sub_func = 0,                //没有订阅其他函数
    .num_prov_func = 1,              //提供了一个函数供其他函数订阅
    .code_id     = ehtons(TREE_ROUTING_PID), //代码ID
    .platform_type = HW_TYPE,
    .processor_type = MCU_TYPE,
    .module_handler = tree_routing_module, //模块的消息处理函数
    .funct = {
        [0] = {tr_get_hdr_size, "Cvv0", TREE_ROUTING_PID,
                MOD_GET_HDR_SIZE_FID},
    },
};
```

- 模块头中的.funct指明了Tree\_routing模块提供给其他模块的函数的信息
- 第一个参数是该模块提供给其他模块的函数名
- 第二个参数是函数的原型信息
- 第三个参数和第四个参数分别是模块ID和函数ID
- 原型信息编码了基类型信息和是否含有动态内存的参数

- 两种方法：一种是静态的，另一种是动态的
  - 静态订阅是指模块在编译时指定它要调用的函数
- ```
static mod_header_t mod_header
SOS_MODULE_HEADER = {
    .....//省略
    .num_sub_func = 1, //订阅了一个函数
    .num_prov_func = 0, //没有提供给其他模块可订阅的
    函数
    .....//省略
    .funct = {
        [0] = {error_8, "Cvv0", TREE_ROUTING_PID,
        MOD_GET_HDR_SIZE_FID},
    }
};
```

- 动态函数订阅：使用动态函数订阅时，只需在模块头中指明原型信息，并且模块ID和函数ID均使用系统预定义的运行时ID

```
static mod_header_t mod_header SOS_MODULE_HEADER =
{
    .....//省略
    .num_timers    = 1,
    .num_sub_func  = 2,
    .num_prov_func = 0,
    .....//其他部分的定义
    .funct = {
        [0] = {error_16, "Svv0", RUNTIME_PID, RUNTIME_FID},
        [1] = {error_8, "cCC2", RUNTIME_PID, RUNTIME_FID},
        },
    };
其中： #define RUNTIME_FID    255
        #define RUNTIME_PID    NULL_PID
```

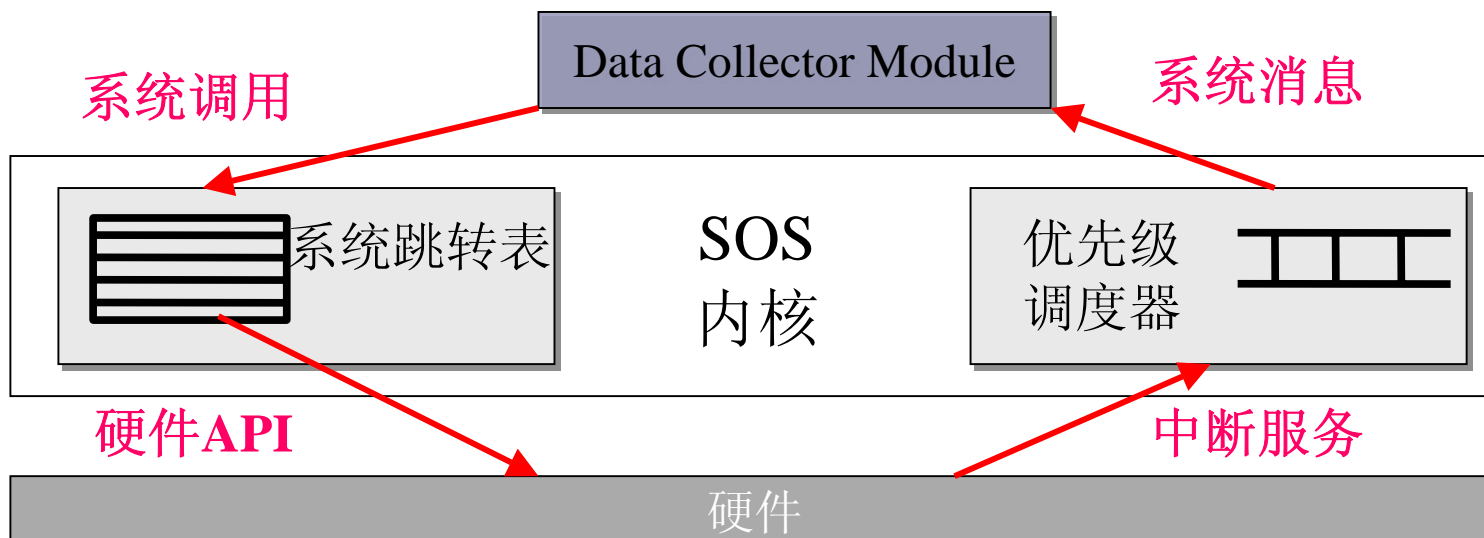
- 用户还需要在程序中显式的使用内核函数 `ker_fntable_subscribe()`实现对函数的订阅，得到函数指针，该函数的原型如下：  

```
int8_t ker_fntable_subscribe(sos_pid_t sub_pid,  
sos_pid_t pub_pid, uint8_t fid, uint8_t table_index)
```

第一个参数是订阅者模块ID，  
第二个参数是提供者模块ID，  
第三个参数是被订阅函数的ID，  
第四个参数是函数记录的索引
- 成功订阅函数后，用户使用 `SOS_CALL(fnptrptr,type, args...)`来调用被订阅到的函数

# 模块与内核交互

- 通过系统调用获得内核服务
- 跳转表吧系统调用重定向到服务提供者
- 内核的升级独立于模块
- 系统调用开销- 12 个时钟周期



- 模块插入过程如下：
  1. 运行于节点上的分布协议监听网络中的新模块。
  2. 当监听到一个模块的广播后，它检查该模块的版本是否比运行在自己节点上的模块更新，或者节点对这个模块感兴趣并且有空闲的程序空间可以容纳这个模块。
  3. 当上述的2种情况之一成立，则分布协议开始下载模块并立即检查数据包头中的元数据。元数据中包含了模块ID，所需的内存空间大小（用于存储模块的局部状态），模块版本信息（用于区分模块的新旧）。
  4. 若SOS内核不能为模块分配用于存储模块局部状态的内存，模块插入立即失败退出。
  5. 在模块插入期间会创建一个内核数据结构，该结构以模块ID（包含在元数据中）作为索引。这个数据结构存储了模块消息处理函数的绝对地址，指向用于存储模块状态的动态内存的指针和模块的标志。
- 内核通过给模块调度一个final消息来启动模块删除，收到final消息后，内核将通过释放动态分配的存储空间，定时器，传感器驱动程序和被模块所拥有的其他的资源

- 非抢占的优先级调度策略
- 优先级队列有两种：高优先级和低优先级
  1. 高优先级队列用于实时性比较高的事件，包括硬件中断和敏感定时器等
  2. 低优先级队列用于调度大多数普通的事件
- 优点：改善了系统的中断响应服务性能
- 消息也能够传递参数
- 使用post\_short和post\_long这两个函数把消息放入相应的队列
- 系统的主调度函数定义在sos-1.7\kernel\sched.c

- SOS中消息结构定义如下:

```
typedef struct Message{
    sos_pid_t did;           //消息目标模块的ID
    sos_pid_t sid;           //源模块的ID
    uint16_t daddr;          //节点的目的地地址
    uint16_t saddr;          //节点的源地址
    uint8_t type;            //模块消息类型
    uint8_t len;             //负载长度
    uint8_t *data;           //指向实际的数据负载
    uint16_t flag;           //表示消息的状态
    uint8_t payload[SOS_MSG_PAYLOAD_LENGTH]; //静态分配的负载
    struct Message *next;    //指针，用于连接下一个消息
} PACK_STRUCT Message
```

- 优先级消息队列定义如下:

```
typedef struct {  
    uint8_t msg_cnt; //队列中的消息总数  
    uint8_t lm_cnt; //低优先级消息数  
    uint8_t sm_cnt; //系统消息数  
    uint8_t hm_cnt; //高优先级消息数  
    Message *hq_head; //高优先级队列  
    Message *hq_tail;  
    Message *sq_head; //系统队列  
    Message *sq_tail;  
    Message *lq_head; //低优先级队列  
    Message *lq_tail;  
} mq_t;
```

- 系统的主调度函数（定义在sos-1.7\kernel\sched.c）：

```
void sched(void)
{
ENABLE_GLOBAL_INTERRUPTS();    //使能全局中断
for(;;){
    SOS_MEASUREMENT_IDLE_END();
    DISABLE_GLOBAL_INTERRUPTS(); //关闭全局中断
    if (int_ready != 0) {
        ENABLE_GLOBAL_INTERRUPTS();
        if (true == sched_stalled) continue;
        handle_callback();
    } else if( schedpq.msg_cnt != 0 ) {
        ENABLE_GLOBAL_INTERRUPTS();
        if (true == sched_stalled) continue;
        do_dispatch();           //这个函数实现了真正的消息调度，它定
                                //义在sos-1.7\kernel\sched.c文件中
    } else {
        SOS_MEASUREMENT_IDLE_START();
        atomic_hardware_sleep();
    }
    watchdog_reset();
}
}
```

- 模块需要内存存储状态信息
- 采用了简单的最佳适应固定块内存分配策略，块的大小有三种
- 空闲的内存按照块的大小组成空闲链表——常数时间的内存分配和回收开销
- 提供了一种用于请求数据所有权改变的机制
- 动态内存块还带有少量的注释数据，用于检测内存溢出
- 运行时动态分配和回收内存的方法提高了内存的利用率

```
pkt = (uint8_t*)ker_malloc(hdr_size + sizeof(SurgeMsg),  
SURGE_MOD_PID)
```

- 源代码级的网络模拟
  - UDP很好的模拟了无线信道
  - 模拟框架提供了一种简单的方式用来支持SOS应用程序开发
  - 无需安装交叉编译器
  - 需要定制内核
  - 支持用户定义的拓扑和异构软件配置
  - 用于验证功能的正确性
- 使用Avrora实现了指令级的模拟
  - 指令周期精确模拟
  - 用于验证定时信息
  - 参考<http://compilers.cs.ucla.edu/avrora/>

- 编程语言：C，减小了学习新的编程语言的难度
- 可以充分利用C的编译器，开发环境，调试器和其他的为C所设计工具
- 代码实例请参考课本

1. WSN操作系统概述
2. TINYOS 操作系统
3. MANTIS 操作系统
4. SOS 操作系统
5. 三种操作系统的设计实现比较
6. WSN OS最新研究进展
7. 主要参考文献

# TinyOS, MOS和SOS的设计实现比较

- 下面从设计操作系统必须考虑的几个方面列举了这三个系统的设计考虑

| 系统特征           | TinyOS   | MOS      | SOS      |
|----------------|----------|----------|----------|
| 事件驱动           | √        |          | √        |
| 线程驱动           |          | √        |          |
| 处理器能量管理        | √        | √        | √        |
| 外设能量管理         | √        | √        |          |
| 优先级调度          |          | √        | √        |
| 实时服务           |          | √        |          |
| 远程动态重编程服务      | √        |          | √        |
| 外设管理           | √        | √        |          |
| 模拟服务           | √        | √        | √        |
| 内存管理<br>系统执行模型 | 静态<br>组件 | 静态<br>线程 | 动态<br>模块 |

1. WSN操作系统概述
2. TINYOS 操作系统
3. MANTIS 操作系统
4. SOS 操作系统
5. 三种操作系统的设计实现比较
6. WSN OS最新研究进展
7. 主要参考文献

在我们书籍的编写过程中，操作系统的设计者也正在努力的完善自己的系统，到目前为止，进展如下：

- TinyOS: 2007-7-31 TinyOS 2.0.2 发布  
<http://www.tinyos.net/>
- MOS: 更改了**License**，现在的不稳定版本采用了BSD license，原来的0.9.5版本仍采用eCOS-style license  
<http://mantis.cs.colorado.edu/index.php/tiki-index.php>
- SOS:2007-06-29 SOS 2.0.1 版本发布  
<https://projects.nesl.ucla.edu/public/sos-2x/doc/>

1. WSN操作系统概述
2. TINYOS 操作系统
3. MANTIS 操作系统
4. SOS 操作系统
5. 三种操作系统的设计实现比较
6. WSN OS最新研究进展
7. 主要参考文献

## 主要参考文献

- [1] Philip Levis et al. The Emergence of Networking Abstractions and Techniques in TinyOS. In Proceedings of the First Symposium on Networked Systems Design and Implementation. 2004.
- [2] Chih-Chieh Han, Ram Kumar Rengaswamy, Roy Shea, Eddie Kohler and Mani Srivastava. SOS: A dynamic operating system for sensor networks. Proceedings of the Third International Conference on Mobile Systems, Applications, And Services (Mobisys), 2005.
- [3] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, R. Han. MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms. ACM/Kluwer Mobile Networks & Applications (MONET), Special Issue on Wireless Sensor Networks, vol. 10, no. 4. August 2005.
- [4] David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. Proceedings of Programming Language Design and Implementation (PLDI). June 2003.
- [5] Philip Levis and Nelson Lee. TOSSIM: A Simulator for TinyOS Networks. In proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys), 2003.
- [6] Philip Levis, Cory Sharp. Schedulers and Tasks. <http://www.tinyos.net/tinyos-2.x/doc/html/tep106.html>. December 2006.
- [7] Roy Shea, Chih-Chieh Han, and Ram Kumar Rengaswamy. Motivations Behind SOS. NESL Technical Report. 2004.
- [8] Philip Levis. TinyOS Programming. [www.tinyos.net](http://www.tinyos.net). June 28, 2006.
- [9] Ram Kumar Rengaswamy. sos-dynamic operating system for sensor networks. Proceedings of the Third International Conference on Mobile Systems, Applications, And Services (Mobisys), 2005. June, 2005.

- [10] Kevin Klues, Vlado Handziski, David Culler, David Gay, Phillip Levis, Chenyang Lu, Adam Wolisz. Dynamic Resource Management in a Static Network Operating System. Department of Computer Science & Engineering. September 2006.
- [11] David Gay, Philip Levis, David Culler, Eric Brewer. nesC 1.1 Language Reference Manual. May 2003.
- [12] 孙利民, 李建中, 陈渝, 朱红松. 无线传感器网络. 北京: 清华大学出版社, 2005.
- [13] 黄光燕, 李晓维. 无线传感器网络操作系统. 信息技术快报. 2005, Vol 3 No 3.
- [14] 陈喜贞, 王书茂, 徐勇军. 无线传感器网络操作系统调度策略研究. 中国科技论文在线.
- [15] TinyOS: [www.tinyos.net](http://www.tinyos.net).
- [16] MagnetOS: <http://www.cs.cornell.edu/People/egs/magnetos/overview.html>.
- [17] Mos: <http://mantis.cs.colorado.edu/index.php>
- [18] SenOS: [http://redwood.snu.ac.kr/bbs/zboard.php?id=Technical\\_reports](http://redwood.snu.ac.kr/bbs/zboard.php?id=Technical_reports).
- [19] SOS: <http://nesl.ee.ucla.edu/projects/SOS/release/>.
- [20] <http://www.sics.se/contiki/>.



宁波中科无线通信事业部  
<http://www.wsn.org.cn>

谢谢!

宁波中科无线通信事业部  
<http://www.wsn.org.cn>