



浙江大学
Zhejiang University



第三章：物联网操作系统

董 玮

浙江大学

目录

- ❑ 物联网操作系统概述
- ❑ 物联网操作系统构成
- ❑ 关键特性
- ❑ TinyOS
- ❑ Contiki
- ❑ LiteOS
- ❑ AliOS Things
- ❑ 研究进展

物联网操作系统概述

□ 物联网操作系统概念

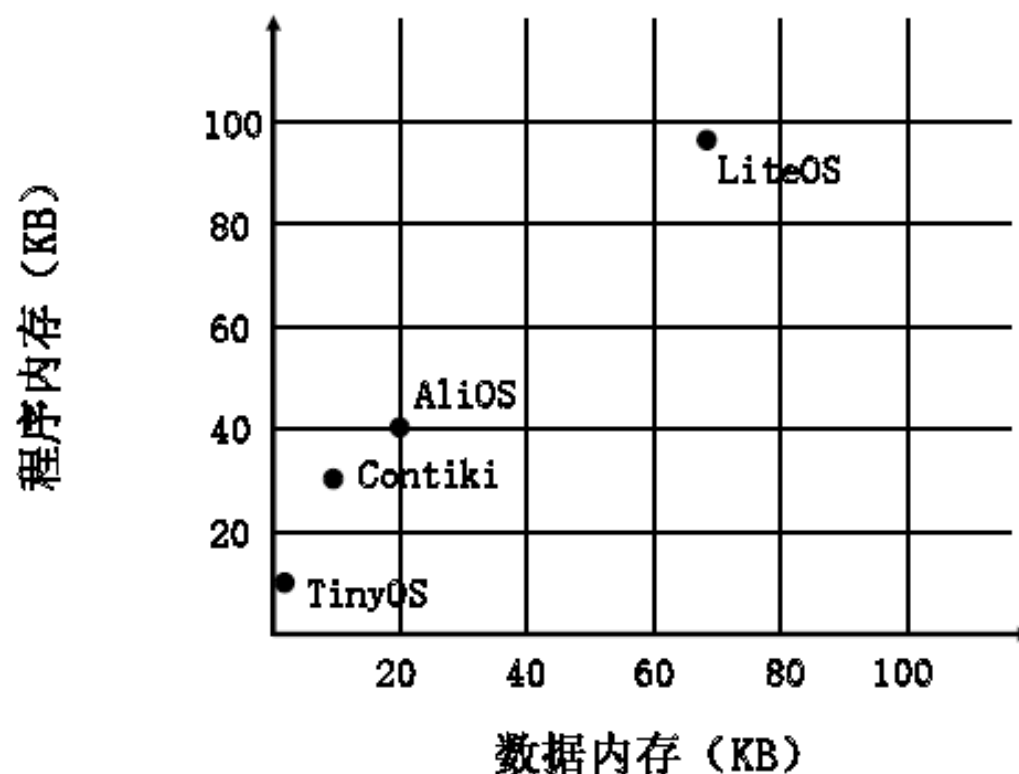
- 物联网操作系统是支撑物联网大规模发展的最核心软件，由操作系统内核、外围功能组件、物联网协同框架、通用智能引擎、集成开发环境等几个大的子系统组成。物联网操作系统屏蔽了物联网碎片化的特征，提供统一的编程接口，降低了物联网应用开发的成本和时间，为物联网统一管理奠定了基础

物联网操作系统概述

□ 与传统操作系统区别

➤ 内存占用

- 物联网操作系统运行在资源受限的节点上，内存占用与传统操作系统相比较小，一般只有几十KB



物联网操作系统概述

□ 与传统操作系统区别

➤ 内存管理

- 传统操作系统一般运行在具有内存管理单元(MMU)的设备上，能够提供地址翻译和内存保护功能。物联网操作系统则因设备而异，当运行在缺乏MMU的设备上时，一般很难提供地址翻译和内存保护功能，当运行在具备MMU的设备上时，可以提供轻量级的地址翻译和内存保护功能

➤ CPU特权模式

- 由于缺乏CPU特权模式，物联网操作系统很难解决控制冒险(Control hazards)的问题，这是因为有Bug的程序可以在不受操作系统的控制下轻松地占用CPU周期，从而导致传感器节点没有响应。传统操作系统由于具备CPU特权模式，就不存在这一问题

物联网操作系统构成

❑ 任务调度

- 抢占、时间片轮转、优先级、FIFO

❑ 动态加载

- 需要的时候进行加载

❑ 内存管理

- 静态分配、动态分配

❑ 资源抽象

- 文件->IO设备, 进程->CPU

❑ 传感接口

- 将数据传递至MCU

❑ 网络协议栈

- BLE、LwIP、LoRaWAN

关键特性

□ 关键特性

- 编程模型
- 调度方式
- IO操作
- 内存分配
- 软件更新
- 网络服务
- 安全机制

编程模型

□ 非模块化 vs. 模块化编程

➤ 非模块编程

- 例如Arduino, setup()和loop()

➤ 模块化编程

- 基于组件的编程模型(TinyOS) & 基于模块的编程模型(SOS)

□ 事件驱动编程 vs. 多线程编程

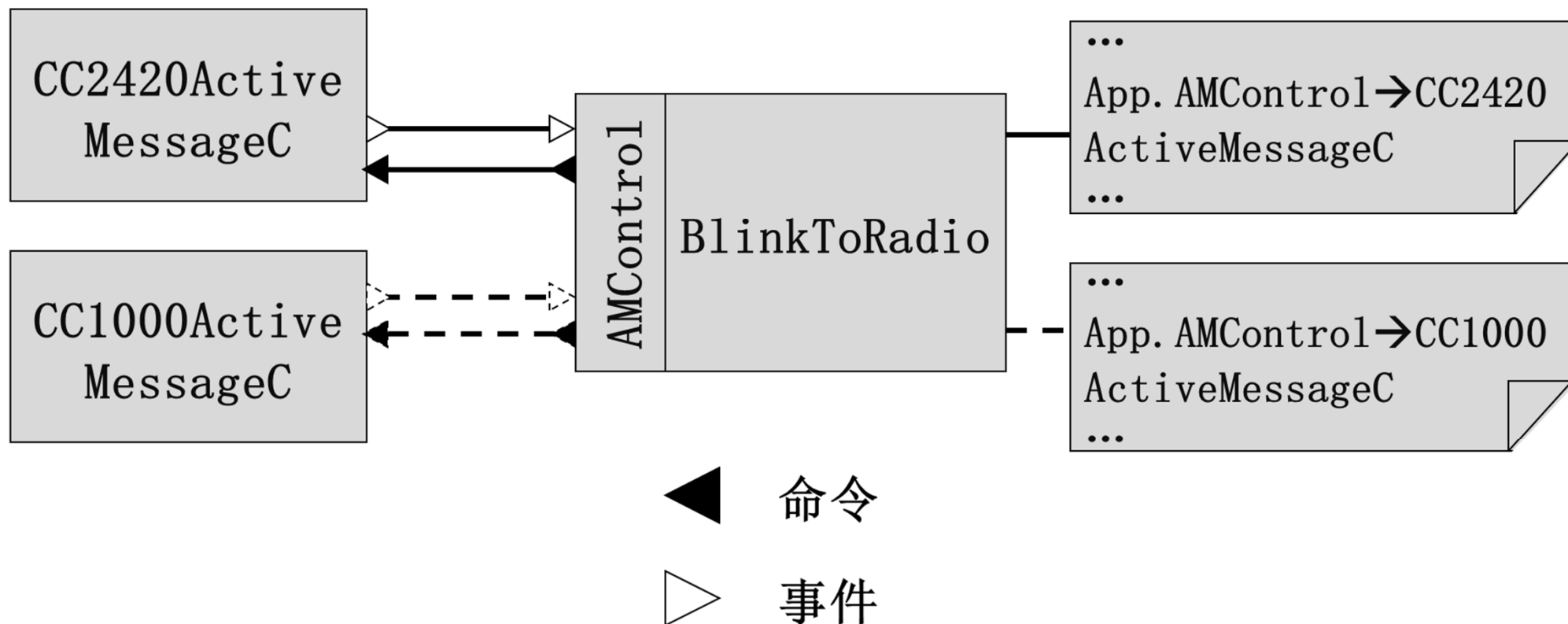
➤ 事件驱动：应用程序由事件处理器构成，一个任务被分解为多个事件处理器

➤ 多线程：一个线程正好对应一个逻辑任务

编程模型：模块化编程

□ 基于组件的编程模型(TinyOS)

- 应用程序由多个组件灵活组合而构成
- 应用与OS一起编译后形成整个二进制程序映像
- 便于整体优化，易于移植，适用于程序改动少的应用



编程模型：模块化编程

□ 基于模块的编程模型(SOS)

- 应用程序编译成可由OS内核加载的单独程序模块
- 便于模块动态修改，适用于程序更新频繁的应用

```
static const mod_header_t mod_header SOS_MODULE_HEADER = {
    .mod_id      = SURGE_MOD_PID,
    .state_size  = sizeof(surge_state_t),
    .num_sub_func = 1,
    .num_prov_func = 0,
    .platform_type = HW_TYPE /* or PLATFORM_ANY */,
    .processor_type = MCU_TYPE,
    .code_id      = ehtons(SURGE_MOD_PID),
    .module_handler = surge_module_handler,
    .funct = {
        [0] = {error_8, "Cvv0", TREE_ROUTING_PID, MOD_GET_HDR_SIZE_FID},
    },
};
```

编程模型：事件驱动编程 vs. 多线程编程

□ 多线程编程

➤ 示例：多线程模型下读取三个传感器后传输的程序代码

```
send_thread:
  radio_pkt
  temp, humidity, light

  while (TRUE) {
    temperature_sensor_read(&temp);
    humidity_sensor_read(&humidity);
    light_sensor_read(&light);
    radio_pkt ← temp | humidity | light;
    radio_send(DEST, radio_pkt);
    cpu_sleep(PERIOD);
  }
```

编程模型：事件驱动编程 vs. 多线程编程

□ 事件驱动编程

➤ 示例：事件驱动模型下读取三个传感器后传输的程序代码

```
event void timer.fired(){
    radio_send(DEST, lastSample); // send the last sample
    temperature_sensor_read(); // request 3 I/Os simultaneously
    humidity_sensor_read();
    light_sensor_read();}

event void radio_sendDone(){}

event void temperature_readDone(){
    store currentSample;}

event void humidity_readDone(){
    store currentSample;}

event void light_readDone(){
    store currentSample;}
```

编程模型

不同操作系统对比

操作系统	模块化	事件驱动	多线程
TinyOS	✓	✓	✓
Contiki	✓		✓
AliOS	✓	✓	✓
LiteOS	✓		✓

调度方式

- ❑ 抢占式、非抢占式、协同式和时间片轮转
- ❑ 使用堆栈的个数
 - 抢占式：需要切换线程上下文，多个堆栈（、时间片轮转）
 - 非抢占式：无需切换线程上下文，单个堆栈
 - 协同式：需要切换线程上下文，一般多个堆栈（Contiki使用单个堆栈实现，利用线程的状态变量切换上下文，详见后文Contiki部分）
 - 时间片轮转：需要切换线程上下文，多个堆栈
- ❑ 线程的原子操作，写入正确的变量x

```
{  
    x++;  
    write(x);  
}
```

非抢占式

```
{  
    x++;  
    Yield(); x  
    write(x);  
    Yield(); ✓  
}
```

协同式

```
mutex m;  
m.lock(); {  
    x++;  
    write(x);  
}  
m.unlock();
```

抢占式或时间片轮转

调度方式

□ 现有的物联网软件系统调度方式对比。

软件系统	TinyOS	Contiki	LiteOS	AliOS Things
调度方式	非抢占式	协同式	抢占式/时间片轮转	抢占式/时间片轮转

I/O操作方式

□ I/O操作方式：阻塞、分阶段

阻塞(LiteOS)	分阶段 (TinyOS)
<pre>if (send() == SUCCESS) { sendCount++; }</pre>	<pre>// start phase call SubSend.send(); //completion phase event void sendDone(error_t err) { if (err == SUCCESS) { sendCount++; } }</pre>

I/O操作方式

□ 两种I/O方式对比

I/O操作方式	阻塞I/O	分阶段I/O
优点	易于编程; 阻塞时自动进行堆栈切换;	不占用堆栈内存; 保持系统的高响应性; 高并发性; 高度可移植性;
缺点	昂贵的上下文切换; 因为栈操作而不可移植;	需要手动切分任务;

I/O操作方式

不同操作系统对比

操作系统	阻塞I/O	分阶段I/O
TinyOS	√ (TOSThreads)	√
Contiki	√	
AliOS	√	√
LiteOS	√	

内存分配

□ 内存分配

- 内存分配是指在软件运行时，操作系统对计算机内存资源进行划分和分配

□ 分配方式

- 常见的内存分配方式是静态内存分配和动态内存分配

内存分配

□ 不同操作系统对比

➤ TinyOS

➤ Contiki

➤ LiteOS

➤ AliOS

操作系统	TinyOS	Contiki	LiteOS	AliOS
内存分配	静态	静态(默认)、动态	静态、动态	静态、动态

软件更新

□ 固件升级（软件更新）

➤ 升级方法

- 将ROM/Flash 中存储程序的扇区内容擦除并写入新文件

➤ 升级步骤

- 上电执行BootLoader（类似于PC的BIOS）
- BootLoader 查找升级文件
- 擦除Flash 中的部分扇区
- 在擦除的扇区写入升级的文件
- 写入完成，读取数据检验是否出错
- 若数据一致，升级成功，删除升级文件
- BootLoader 程序跳转到APP 程序执行

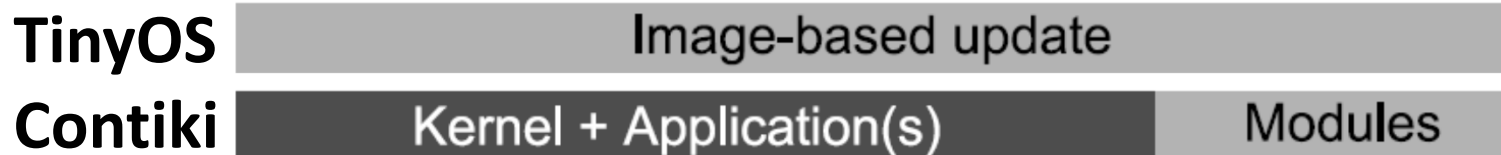
➤ 连接方式

- 有线连接：BSL（MSP430），UART 等
- 无线连接：基于WiFi、ZigBee、LoRa 等方式实现空中升级（FOTA），更加适合物联网的场景

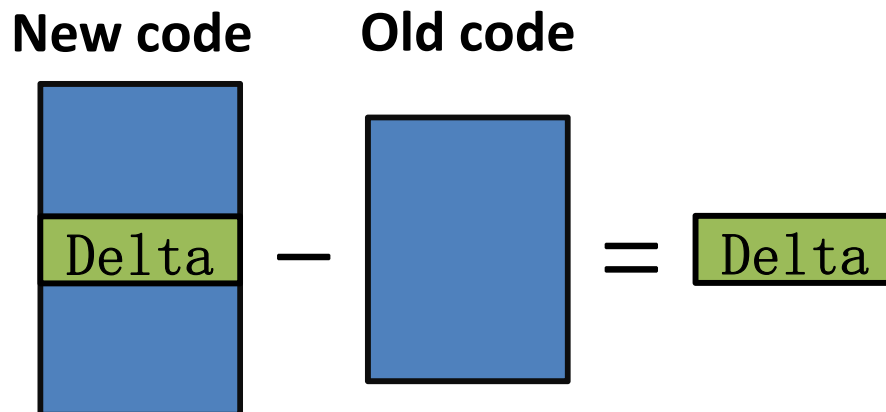
软件更新

□ 升级方式

- 基于整个镜像的软件更新
 - 包含 *Kernel + Application + Module* 的二进制文件
- 基于模块化的软件更新
 - 选择性的更新 *Kernel*、*Application* 和 *Module* 中的部分



- 基于差分的软件更新
 - 对比二进制文件的差异，细粒度的更新二进制文件不同之处

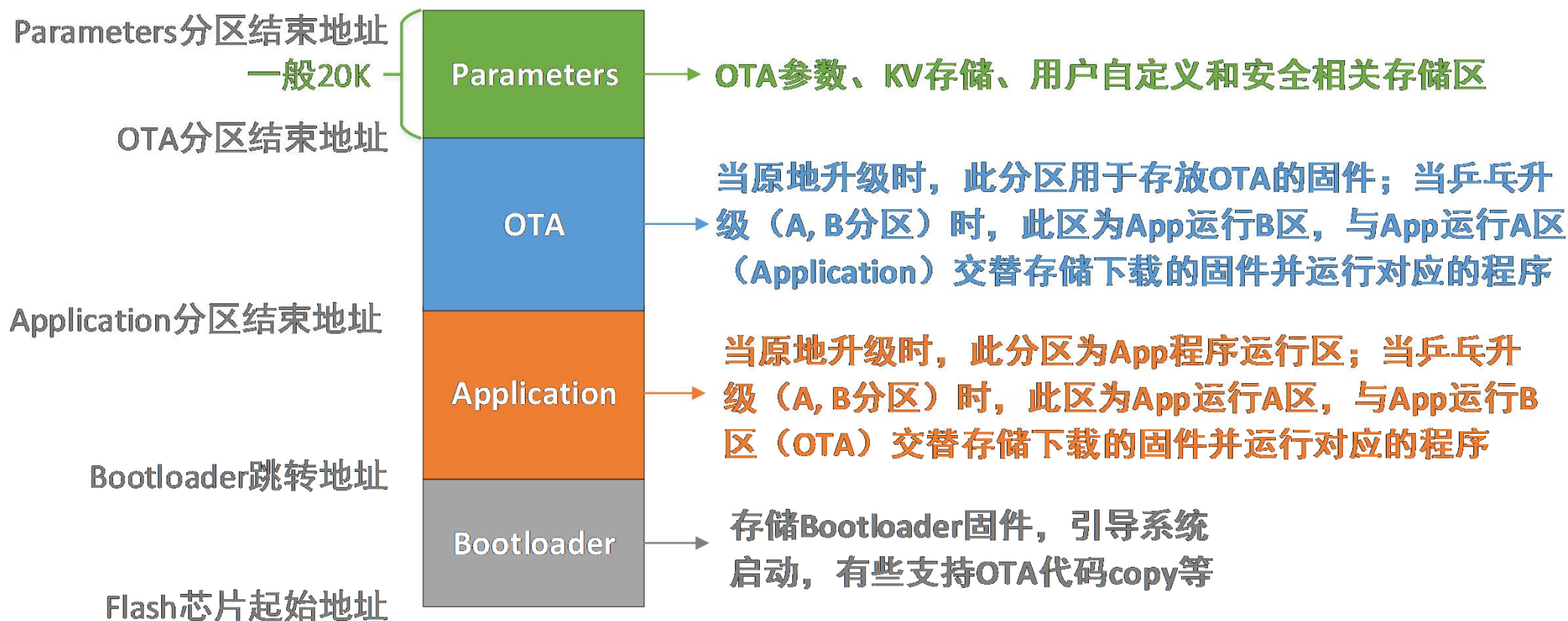


软件更新

❑ FOTA (Firmware Over-The-Air)

- 原地升级
- 乒乓升级（A, B分区，双备份可回滚）

AliOS Things OTA空间划分



软件更新

❑ 对比

	整个镜像升级	模块化升级	差分升级
TinyOS	✓	✓ (TOSThreads)	✗
Contiki	✓	✓	✗
AliOS	✓	✓	✓
LiteOS	✓	✓	✓

网络服务

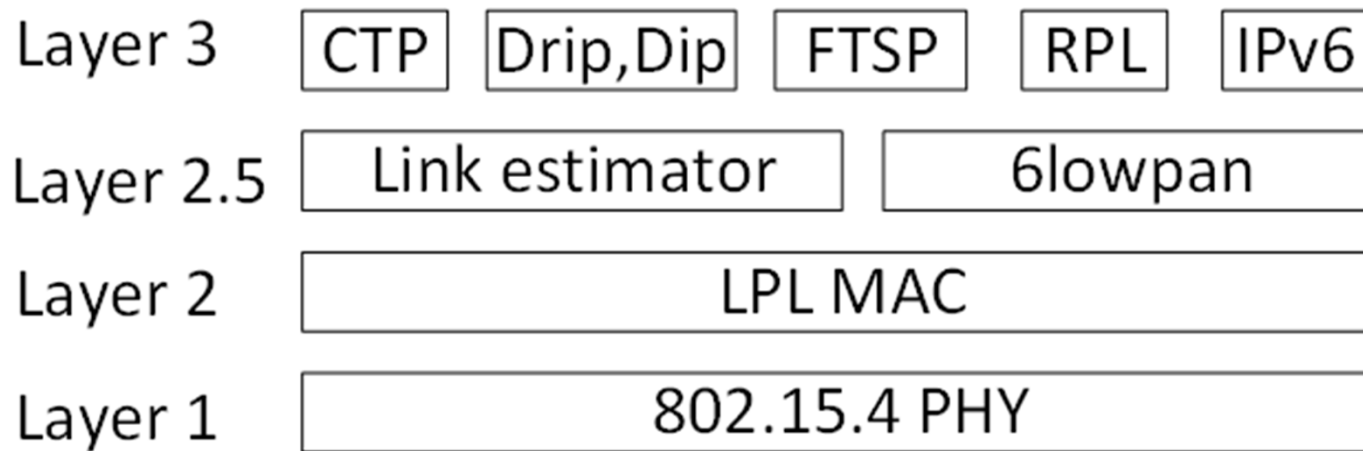
□ 网络服务

- 网络服务是一种服务导向架构的技术，它通过标准的网络协议提供服务，目的是保证不同平台的应用服务可以互操作

□ 类别

- 分为两种，一种是传统无线传感网的网络协议，一种是现代物联网的网络协议

□ TinyOS网络协议栈



➤ CTP

- 支持*many-to-one*的数据传输

➤ RPL

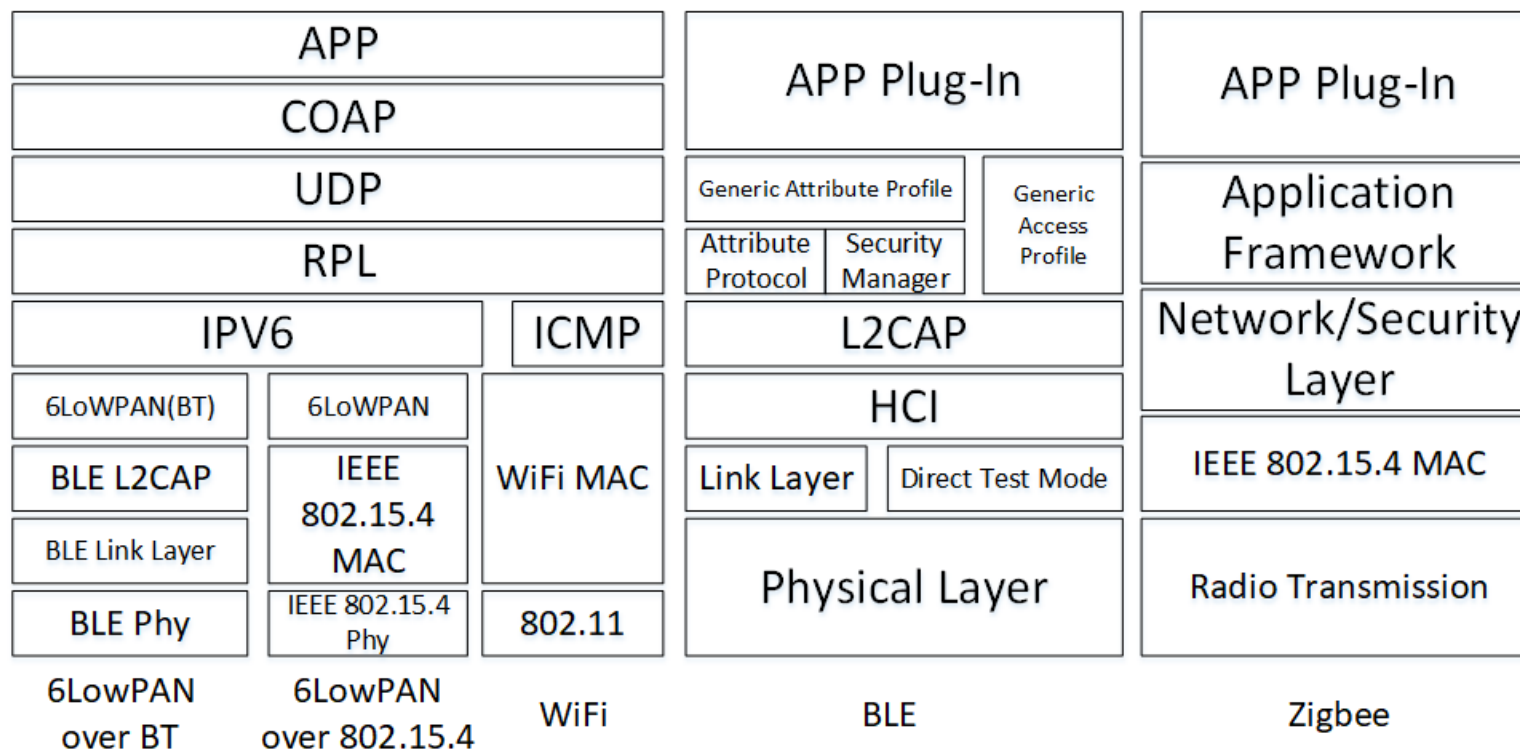
- 支持*many-to-one*、*one-to-many*、*one-to-one*的数据传输

➤ 6LoWPAN

- 实现IPv6的地址压缩、TCP/UDP的包头压缩、ICMP包头压缩

网络服务

□ LiteOS网络协议栈



- LiteOS自带BLE和Zigbee协议栈
- LiteOS支持6LoWPAN over 802.15.4 和6LoWPAN over BT
- 可以发现，学术界专注于对某种特定的通信技术设计网络协议，而工业界更侧重于对不同网络协议的互联互通

安全机制

□ 系统安全

➤ 内存和类型安全性机制 —— Safe TinyOS

- 在代码中插入安全检查，在运行时强行执行类型和内存安全检查
- 这些检查能够在指针/数组错误崩溃程序前捕获他们

/ Buffered high data rate reading, usually from sensor devices */*

```
interface ReadStream<val_t> {  
  
    command error_t postBuffer(val_t* buf, uint16_t n);  
  
    command error_t read(uint32_t usPeriod);  
  
    event void bufferDone(error_t result,  
                           val_t* buf, uint16_t n);  
  
    event void readDone(error_t result, uint32_t usActualPeriod);  
  
}
```

安全机制

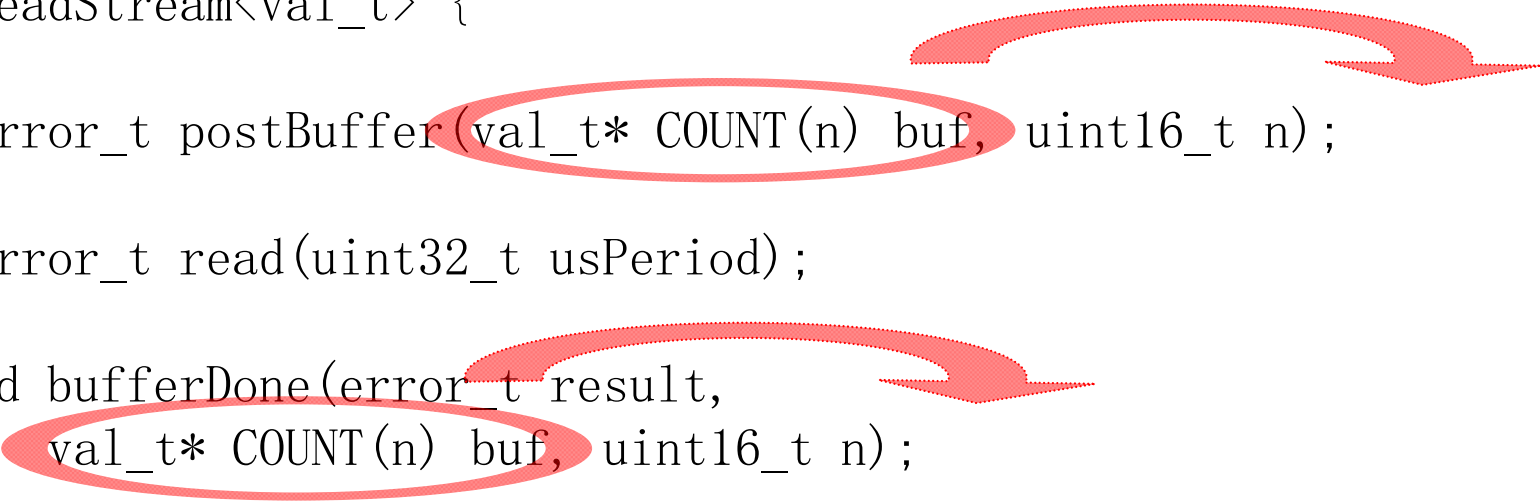
□ 系统安全

➤ 内存和类型安全性机制 —— Safe TinyOS

- 在代码中插入安全检查，在运行时强制执行类型和内存安全检查
- 这些检查能够在指针/数组错误崩溃程序前捕获他们

/ Buffered high data rate reading, usually from sensor devices */*

```
interface ReadStream<val_t> {  
    command error_t postBuffer(val_t* COUNT(n) buf, uint16_t n);  
  
    command error_t read(uint32_t usPeriod);  
  
    event void bufferDone(error_t result,  
        val_t* COUNT(n) buf, uint16_t n);  
  
    event void readDone(error_t result, uint32_t usActualPeriod);  
}
```



COUNT ⇒ Either null or array

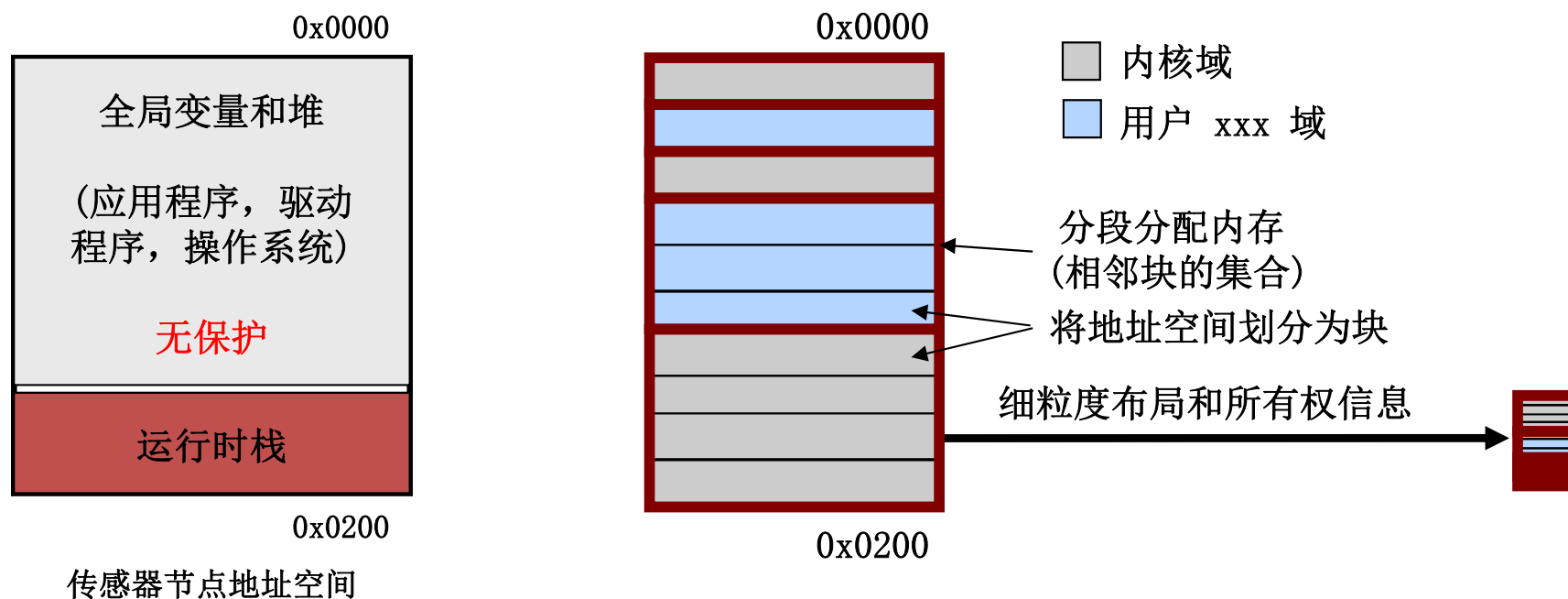
安全机制

□ 系统安全

➤ 沙盒 —— Harbor on SOS

- 用错误隔离技术（沙盒）来限制内存的获取和控制流
- 高效的内存映射（*Memory Map*）技术标记不同内存区域的拥有者并通过校验身份来提供内存写保护
- 将返回地址放置在一个受保护内存区域来提供安全栈（*Safe Stack*），用来保证控制流不受干扰

内存映射



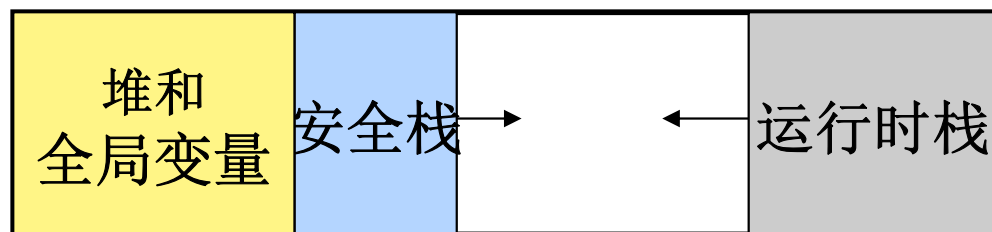
安全机制

□ 系统安全

➤ 沙盒 —— Harbor on SOS

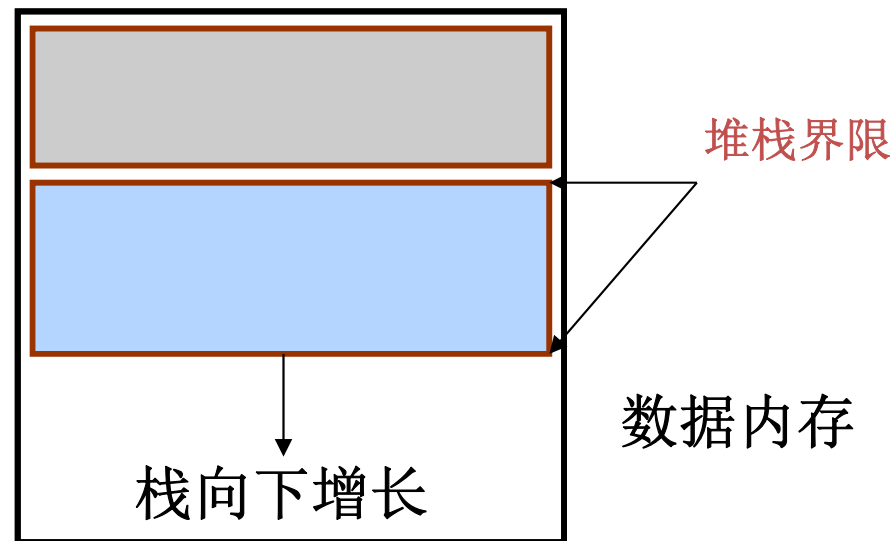
- 用错误隔离技术（沙盒）来限制内存的获取和控制流
- 高效的内存映射（*Memory Map*）技术标记不同内存区域的拥有者并通过校验身份来提供内存写保护
- 将返回地址放置在一个受保护内存区域来提供安全栈（*Safe Stack*），用来保证控制流不受干扰

安全栈



- 存储跨域调用帧
- 存储返回地址

堆栈界限



安全机制

□ 系统安全

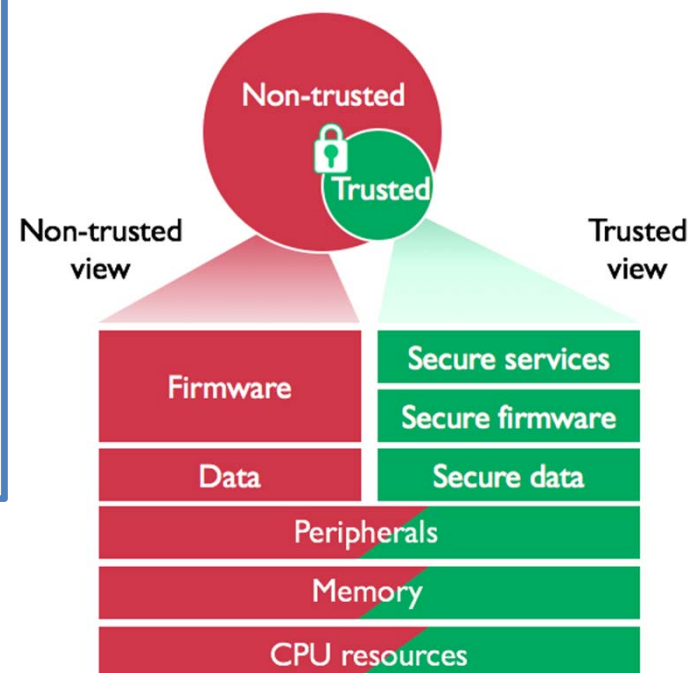
➤ 可信执行环境 —— Link TEE (Trusted Execution Env)

- 支持基于ARM Trustzone或C-SKY的安全扩展技术提供硬件级别的可信根，也提供软件级别的保护方案
- 支持隔离、堆栈保护、地址随机化、防回滚
- 具有高性能（us级切换时间）、低资源（小于百KB存储）特点

ARM TrustZone

- 一种嵌入式安全技术
- 从ARM单核心上同时运行两个环境安全世界和不安全世界
- 同时运行安全操作系统和普通操作系统
- 可保护软件库或操作系统在安全区域中运行
- 禁止非安全软件访问安全环境和其资源

Two worlds - one CPU
Real-time transition*

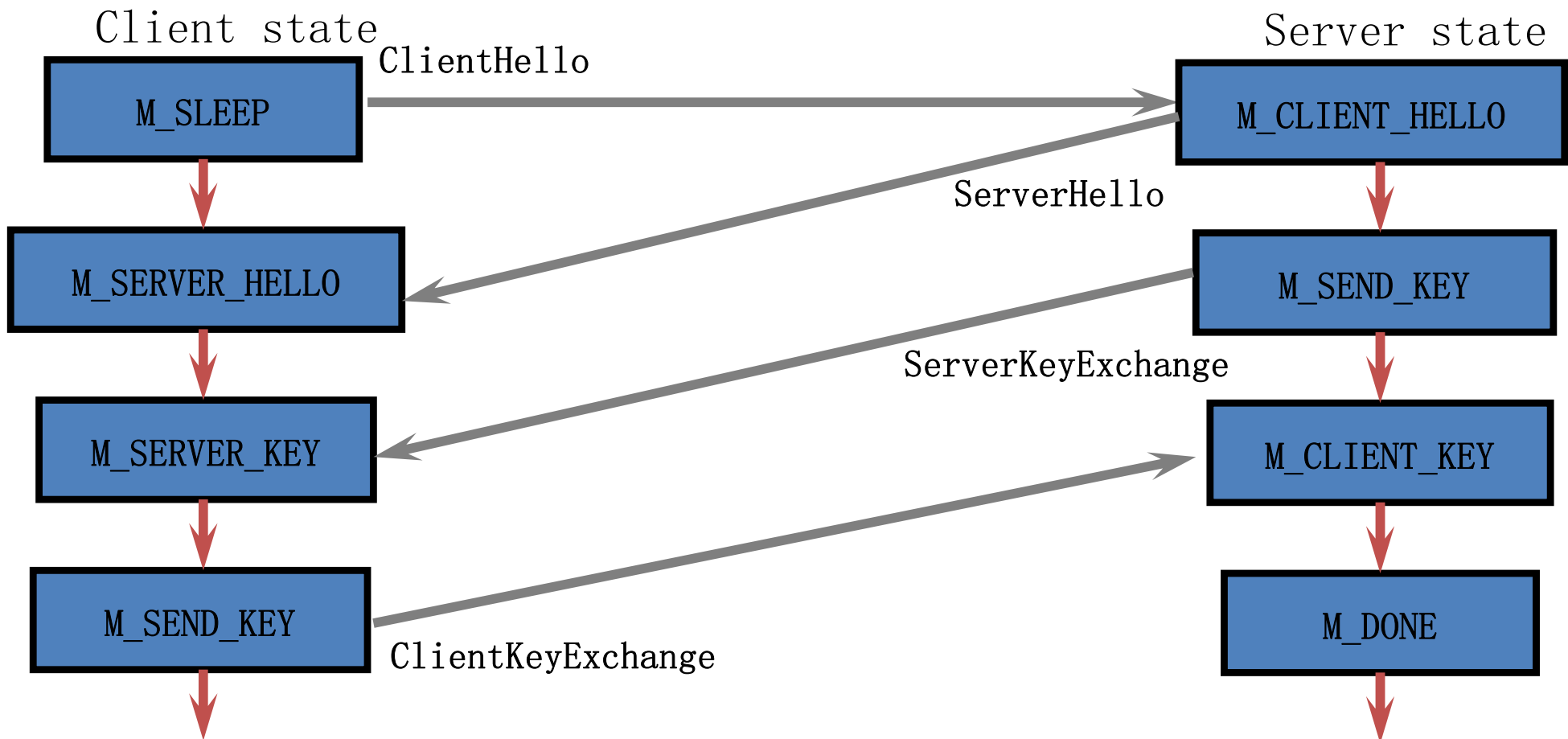


安全机制

□ 传输安全

➤ 安全传输层协议 —— TLS (Transport Layer Security)

- 用于在两个通信应用程序之间提供保密性和数据完整性
- 由两层组成: *TLS 握手协议*和*TLS 记录协议*

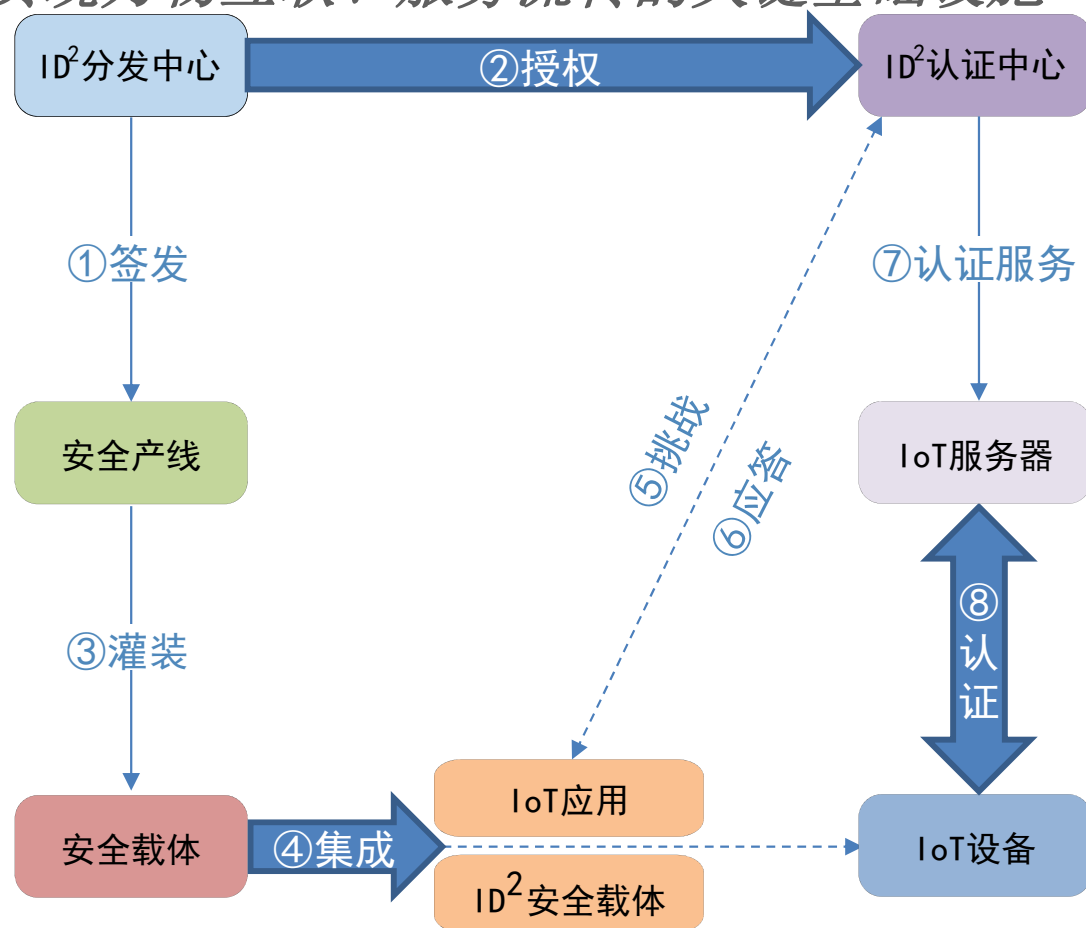


安全机制

□ 传输安全

➤ 可信身份标识 —— 阿里云Link ID²

- 具备不可篡改、不可伪造、全球唯一的安全属性
- 阿里云Link ID²提供密钥分发中心和认证中心两个服务
- 是实现万物互联、服务流转的关键基础设施



安全机制

对比

	系统安全	传输安全
TinyOS	类型和内存安全检查 (Safe TinyOS)	×
Contiki	×	×
AliOS	可信执行环境	安全传输层协议、 可信身份标识
LiteOS	可信应用签名	安全传输层协议

目录

- ❑ 物联网操作系统概述
- ❑ 物联网操作系统构成
- ❑ 关键特性
- ❑ **TinyOS**
- ❑ Contiki
- ❑ LiteOS
- ❑ AliOS Things
- ❑ 研究进展

TinyOS

- TinyOS 简介
- 关键特性
- 编程实例

TinyOS简介

❑ 由加州大学伯克利分校开发，专为无线传感器网络而设计的节点微型操作系统

- 1999年，最早的TinyOS版本出现
 - 在1.0版本前都是用C语言与Perl脚本实现
- 2002年，发布TinyOS1.0，用nesC实现
- 2006年，TinyOS2.0在IPSN会议上发布
- 2012年，最新版本TinyOS2.1.2发布

❑ 支持平台

- Crossbow: TelosB, MicaZ, Mica2, IRIS
- Intel:iMote2 等

❑ 研究项目

- 路易斯安娜州立大学: UcoMS, 用于帮助钻孔、记录操作数据、监控设备、管道管理等
- 清华大学: 绿野干传, 用于森林生态监测

TinyOS支持的硬件列表

平台		<u>EPIC</u> 、 <u>Imote2</u> 、 <u>Shimmer</u> 、 <u>IRIS</u> 、 <u>Telos Rev B</u> 、 <u>MicaZ</u> 、 <u>Mica2</u> 、 <u>Mica2dot</u> 、 <u>NXTMOTE – TinyOS on LEGO</u> <u>MINDSTORMS NXT</u> 、 <u>Mulle</u> 、 <u>TMote Sky</u> , a.k.a. <u>Telos Rev B</u> 、 <u>TinyNode</u> 、 <u>Zolertia Z1</u> 、 <u>UCMote Mini</u>
芯片	微控制器	<u>Atmel ATmega128</u> ，一个8位RISC微控制器 <u>Texas Instruments MSP430</u> ，一款16位低功耗微控制器 <u>Intel XScale PXA271</u> ，一款32位RISC微控制器
	无线电 Radio	<u>CC1000</u> 、 <u>CC1100/CC2500</u> 、 <u>CC2420</u> 、 <u>AT86RF212</u> 、 <u>AT86RF230</u>
	传感器	<u>Sensirion SHT11</u> ，温度和湿度传感器 <u>Hamamatsu S1087</u> ，可见光传感器 <u>Hamamatsu S1087-01</u> ，红外传感器 <u>Intersema MS5534</u> ，温度和气压传感器 <u>Taos TSL2550</u> ，光传感器 <u>Analog Devices ADXL202</u> ，双轴加速度计 <u>LIS3L02DQ</u> ，3轴加速度计 <u>AD5200</u> ，数字电位器

TinyOS设计目标

- **节点资源有限，对操作系统有着低功耗、低复杂度的需求，因此TinyOS设定了以下设计目标**
 - **允许高度的并发性：** 要求执行模式能对事件作出快速的直接响应；
 - **能在有限的资源上运行：** 要求有效利用处理器和内存资源，同时实现低功耗通信；
 - **适应硬件升级：** 要求组件和执行模式能够应对硬件/软件的替换；
 - **支持广泛的应用程序：** 要求能够根据实际需要，方便地裁减或组合操作系统的服务；
 - **鲁棒性强：** 要求系统高度模块化、标准化，通过模块间有限的交互渠道，应对各种复杂情况；
 - **支持一系列平台：** 要求操作系统的服务具有可移植性

TinyOS关键特性

□ 编程模型

- 基于组件的编程模型

□ I/O操作方式

- 分阶段：长时间操作分成请求和完成两个独立阶段

□ 调度机制

- 事件驱动和多线程调度

□ 内存管理

- 静态内存管理

□ 软件更新

- 整个镜像更新(Deluge)、模块化更新(TOSThreads)

□ 网络服务

- 支持汇聚服务、分发服务、时间同步服务等

□ 安全机制

- 系统安全(Safe TinyOS), 提供类型、内存安全检查机制

①基于组件的编程模型： nesC

- ❑ TinyOS及其应用都由nesC语言编写， nesC是C的扩展， 并
有自己的特性
 - 使用nesC编写的应用程序是基于组件的， 组件之间可以
灵活组合， 广泛支持多种应用
 - 组件之间通过接口交互
 - 应用程序要有一个顶层配置文件， 将应用所使用的组件连
接到一起， 定义了组件接口间的连接关系

①基于组件的编程模型：基础编程示例

❑ AntiTheft应用，功能：发现有人偷节点

❑ 案例具体功能描述

➤ 发现小偷

- 假设：小偷将节点放在口袋里，因此“黑暗”的节点就代表被偷了
- 小偷发现算法：每隔N ms检查光照传感器读数是否低于某阈值

➤ 警告小偷

- 假设：闪烁明亮的灯能吓退小偷
- 小偷警告算法：点亮红色的LED灯并持续一段时间

❑ 接下来将看到

- 组件、接口、连接
- 分阶段操作



①基于组件的编程模型：组件

❑ 功能组件文件AntiTheftC.nc

```
module AntiTheftC {// 声明部分
  uses interface Boot;
  uses interface Timer<TMilli> as Check;
  uses interface Read<uint16_t>;
}
implementation {// 实现部分
  event void Boot.booted() {
    call Check.startPeriodic(1000);
  }

  event void Check.fired() {
    call Read.read();
  }

  event void Read.readDone(error_t ok, uint16_t val) {
    if (ok == SUCCESS && val < 200)
      theftLed();
  }
}
```

组件从声明开始，声明：

- provides: 提供的接口
- uses: 要使用的接口

程序功能的实现，实现：

- 提供接口中的命令
- 使用接口中的事件

* **组件**是nesC应用程序的基本组成单元

* 一个组件由两部分组成

- 一个是**声明**，声明其提供及使用的接口；
- 另一部分是它们的**具体实现**。

①基于组件的编程模型：接口

❑ 功能组件文件AntiTheftC.nc

```
module AntiTheftC {  
  // 声明部分  
  uses interface Boot;  
  uses interface Timer<TMilli> as Check;  
  uses interface Read<uint16_t>;  
}
```

```
interface Boot {  
  /* Signaled when OS booted */  
  event void booted();  
}
```

```
implementation {  
  // 实现  
  event void Boot.booted;  
  call Check.startPeriodic(1000);  
}
```

```
interface Timer<tag> {  
  command void startOneShot(uint32_t period);  
  command void startPeriodic(uint32_t period);  
  event void fired();  
}
```

```
event void Check.fired;  
  call Read.read();  
}
```

```
event void Read.readDone(error_t ok, uint16_t val) {  
  if (ok == SUCCESS && val < 200) {  
    theftLed();  
  }  
}
```

*** 接口**是一组相关函数的集合，定义了组件间如何相连
*** 接口是双向的**，提供命令和事件两种交互方式
-命令由接口提供者来实现
-事件由接口使用者A来实现

①基于组件的编程模型：分阶段操作

❑ 功能组件文件AntiTheftC.nc

```
module AntiTheftC {  
  uses interface Boot;  
  uses interface Timer<TMilli> as Check;  
  uses interface Read<uint16_t>;  
}
```

```
implementation {  
  event void Boot.booted() {  
    call Check.startPeriodic(1000);  
  }  
}
```

```
event void Check.fired() {  
  call Read.read(); //启动周期性计时器  
}
```

```
event void Read.readDone(error_t ok, uint16_t val) {  
  if (ok == SUCCESS && val < 200)  
    theftLed();  
}
```

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

分阶段操作:

- 调用命令启动操作: read
- 触发事件标志事件完成: readDone

①基于组件的编程模型：配置文件

❑ 顶层配置文件AntiTheftAppC.nc

```
configuration AntiTheftAppC { }
```

```
implementation
```

```
{
```

```
    components AntiTheftC, MainC, LedsC;  
    components new TimerMilliC() as MyTimer;  
    components new PhotoC();
```

声明所需要用到的组件

TimerMilliC组件是通用组件，在声明时通过关键字“new”来实例化为MyTimer

```
    AntiTheftC.Boot -> MainC.Boot;
```

```
    AntiTheftC.Leds -> LedsC;
```

```
    AntiTheftC.Check -> MyTimer;
```

```
    AntiTheftC.Read -> PhotoC;
```

```
}
```

组件接口间的绑定

指出AntiTheftC组件中使用的接口到底是由哪个组件提供的。

②事件驱动的调度机制：两级调度

- ❑ TinyOS的基本并行运行模型由同步任务和异步中断构成
 - 硬件中断发生时，系统快速调用相关事件处理程序
 - 任务是一个函数，稍后由TinyOS处理器空闲时再运行而不是立即运行，一旦开始就运行直到完成

```
event void Timer0.fired() {  
    uint32_t i;  
    call Leds.led0Toggle();  
    for (i = 0; i < 400001; i++) {}  
}
```

```
task void computeTask() {  
    uint32_t i;  
    for (i = 0; i < 400001; i++) {}  
}  
  
event void Timer0.fired() {  
    call Leds.led0Toggle();  
    post computeTask();  
}
```


②事件驱动的调度机制：两级调度

❑ 任务基于FIFO的非抢占式调度

- 当任务队列为空时，处理器进入休眠，随后由外部事件唤醒CPU进行任务调度

❑ 事件可以抢占任务执行

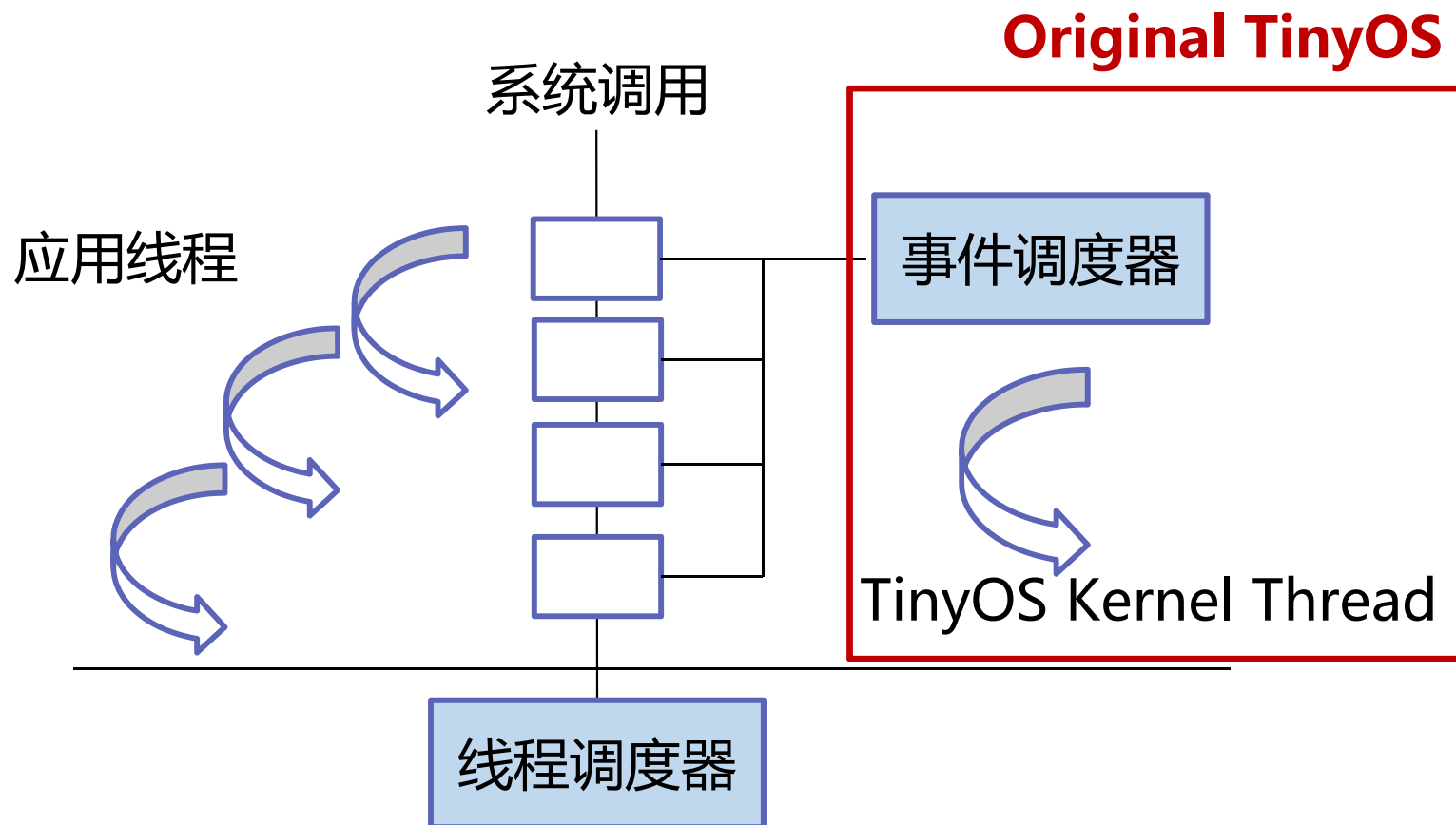
- 当事件对应的硬件中断发生时，系统能够快速调用相关的事件处理程序，具有高度并发性

❑ 适用场景

- 任务一般用在对于时间要求不是很高的应用中，通常要求每一个任务都很短小，能够使系统的负担较轻
- 事件一般用在对于时间的要求很严格的应用中，且它可以占先优于任务和其他事件执行，在TinyOS中一般由硬件中断处理来驱动事件

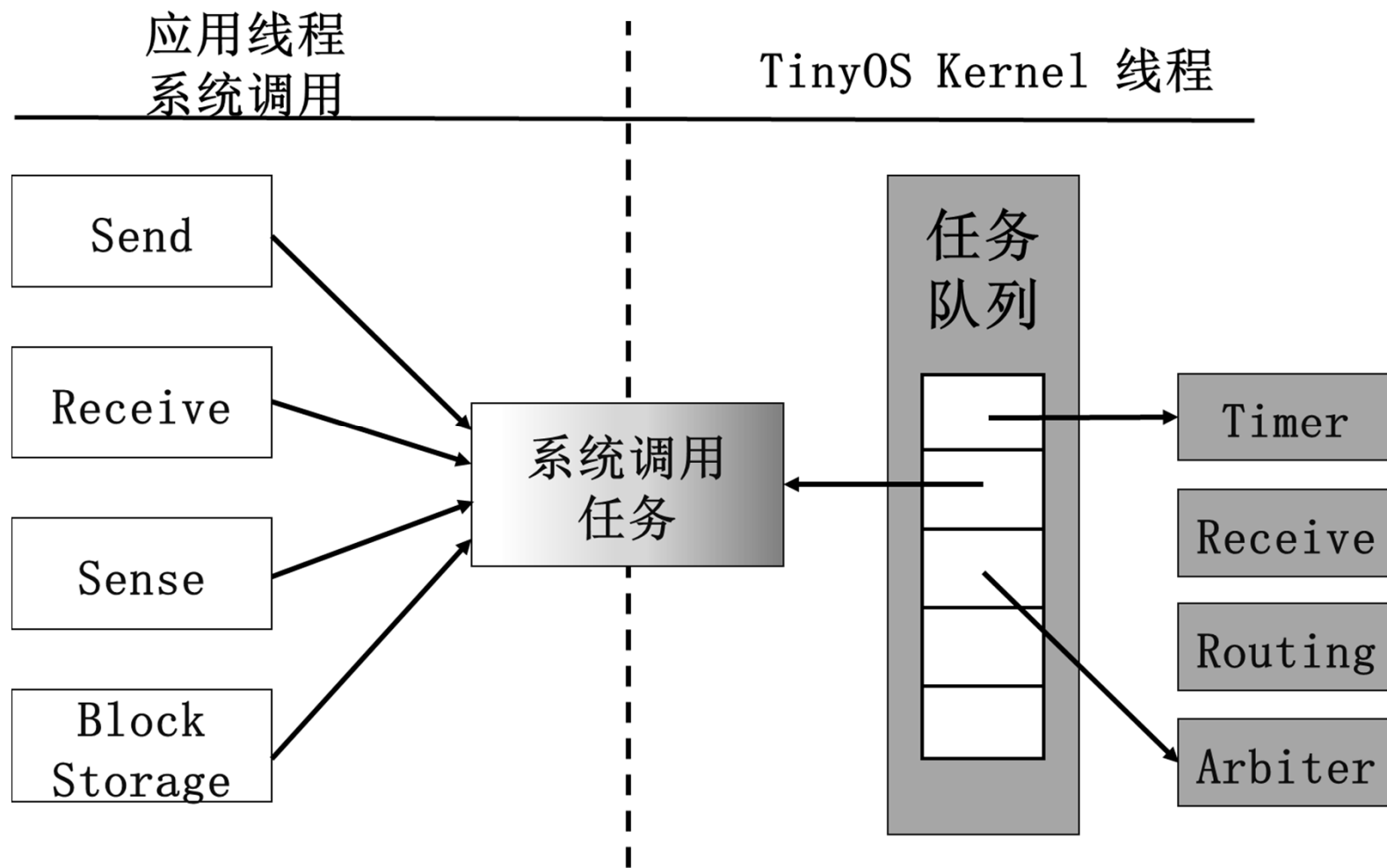
② TOSThread线程调度机制

- ❑ 线程(TOSThreads)是TinyOS提供的第三种并发模型
 - 用户级应用的线程



② TOSThread线程调度机制

- ❑ TOSThreads封装了事件驱动的TinyOS内核服务，提供内核API
- ❑ 这些封装器在单个共享的TinyOS任务中运行各自的系统调用，并与TinyOS自身提交的其它任务交替执行



② TOSThread线程调度机制

❑ ThreadC组件提供Thread接口来创建和操作线程

```
interface Thread {  
    command error_t start(void* arg); //通知线程调度器执行线程  
    command error_t stop();  
    command error_t pause();  
    command error_t resume();  
    command error_t sleep(uint32_t milli);  
    event void run(void* arg); //线程start后触发run事件  
    command error_t join();  
}
```

② TOSThread线程调度机制

❑ TOSThreads示例

➤ 创建三个线程来进行radio收发操作

```
//RadioStressAppC.nc:
```

```
....
```

```
//创建线程0并分配300字节栈空间，同理创建线程1，线程2
```

```
components new ThreadC(300) as RadioStressThread0;
```

```
//对AM Sender的系统调用封装 (AM ID 为 220)
```

```
components new BlockingAMSenderC(220) as BlockingAMSender0;
```

```
//对AM Receiver 的系统调用封装(AM ID is 220)
```

```
components new BlockingAMReceiverC(220) as BlockingAMReceiver0;
```

```
RadioStressC.RadioStressThread0 -> RadioStressThread0;
```

```
RadioStressC.BlockingAMSend0 -> BlockingAMSender0;
```

```
RadioStressC.BlockingReceive0 -> BlockingAMReceiver0;
```

```
.....
```

② TOSThread线程调度机制

```
//RadioStressC.nc:
```

```
.....
```

```
event void Boot.booted() {  
  call RadioStressThread0.start(NULL); //通知线程调度器执行  
  call RadioStressThread1.start(NULL);  
  call RadioStressThread2.start(NULL);  
}
```

```
.....
```

```
event void RadioStressThread0.run(void* arg) { //线程0的主函数  
  call BlockingAMControl.start(); //开启radio, 线程阻塞直到操作完成  
  for(;;) {  
    if(TOS_NODE_ID == 0) {  
      call BlockingReceive0.receive(&m0, 5000); //侦听并接收发过来的数据包, 持续5000ms, 线程阻塞直到操作完成  
      call Leds.led0Toggle();  
    } else {  
      call BlockingAMSend0.send(!TOS_NODE_ID, &m0, 0); //发数据  
      call Leds.led0Toggle();  
    }  
  }  
} } .....
```

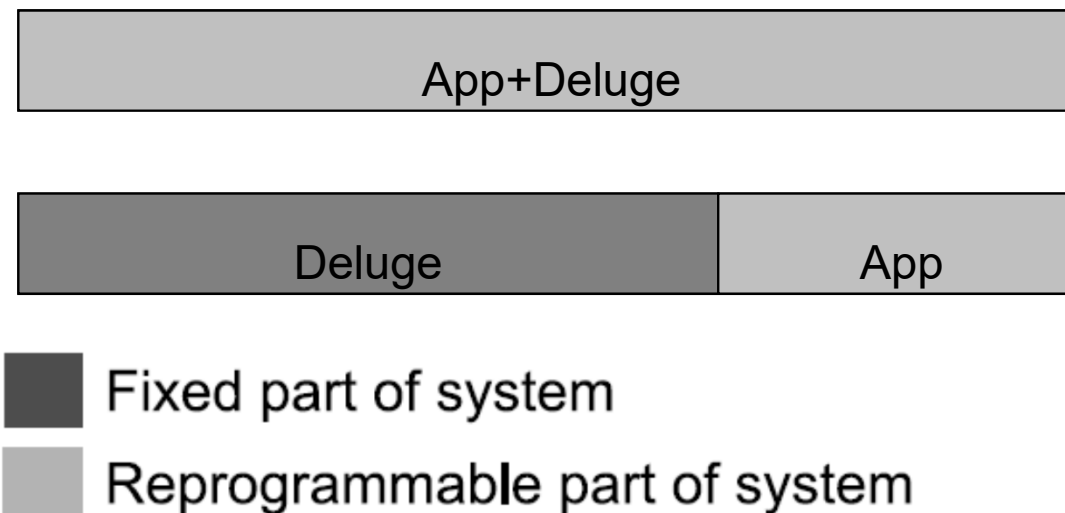
③ 软件更新机制：Deluge

❑ 软件更新在传感网中很重要

- 传感网往往在部署后才能学习到环境变化、网络行为等特征，并对应用做出升级或更新

❑ Deluge是一种可靠的无线重编程协议，适用于传播大型对象，例如程序二进制文件

❑ 在TinyOS中的实现 `/tos/lib/net/deluge`



④丰富的网络服务

网络服务名称（协议）	提供服务
汇聚服务（CTP）	把数据汇聚到基站节点
数据分发服务（Drip,Dip）	分发小段程序、命令及配置信息
时间同步服务（FTSP）	对全网节点进行时间同步

④丰富的网络服务：汇聚服务

□ 协议：汇聚树协议(Collection Tree Protocol, CTP)

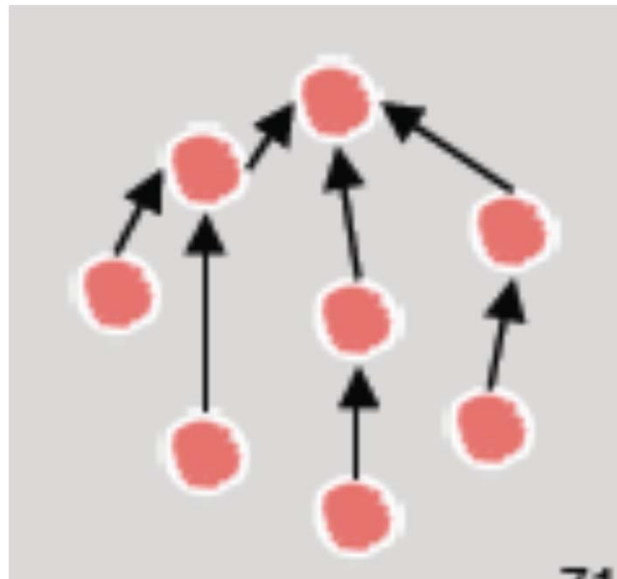
➤ 在TinyOS中的实现 `/tos/lib/net/ctp`

□ 用途：将网络节点产生的数据汇聚到基站节点

□ 方法：网络中的某些节点将自己设为根节点，其他节点形成到这些根节点的路由树

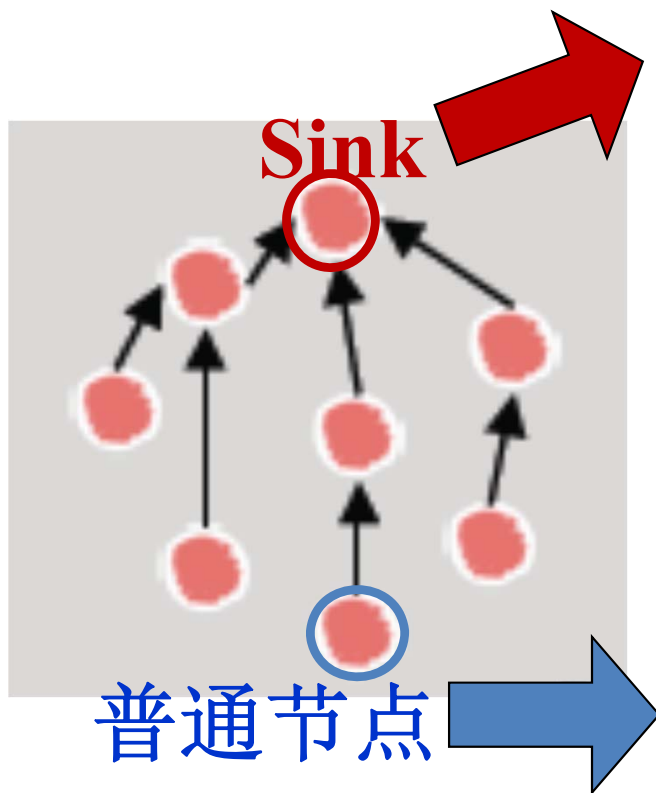
➤ 根节点：设置根节点，接受数据包

➤ 普通节点：发送自己的采集数据，或转发来自其他节点的数据



④丰富的网络服务： 汇聚服务

❑ 汇聚树协议(Collection Tree Protocol, CTP)基本接口



根节点设置

```
interface RootControl {  
    command error_t setRoot();  
    command error_t unsetRoot();  
    command bool isRoot();}
```

数据包接收

```
interface Receive {  
    event message_t* receive(message_t*  
msg, void* payload, uint8_t len);}
```

数据包发送

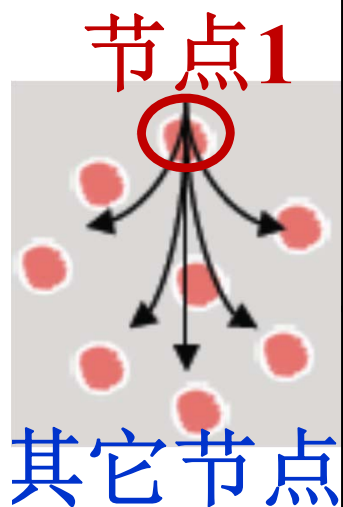
```
interface Send {  
    command error_t send(message_t* msg,  
uint8_t len); ...}
```

数据包处理

```
Interface Intercept {  
    event bool forward(message_t* msg,  
void* payload, uint8_t len);...}
```

④丰富的网络服务：分发服务

- ❑ 协议：Drip库和Dip库 (tos/lib/net/drip, tos/lib/net/dip)
- ❑ 用途：分发协议主要用于实现共享变量的**网络一致性**。
 - 允许管理员向网络注入小段程序、命令以及配置信息。
- ❑ 方法：通知节点变量更改的时间，同时交换数据包以达到整个网络的一致性
- ❑ 分发服务接口及使用



节点1更新数据 Val=5，并将其分 发出去	<pre>uses interface DisseminationUpdate<uint16_t> as Update; uint16_t Val ; Val = 5; call Update.change(Val); //分发 数据</pre>
其它节点获得最新 的数据并更新	<pre>use interface DisseminationValue<uint16_t> as Value; event void Value.changed(){ uint16_t* newVal = call Value.get(); //获取最新数据 Val = * newVal; //更新 }</pre>

④丰富的网络服务：时间同步服务

❑ 协议：泛洪时间同步协议(FTSP)

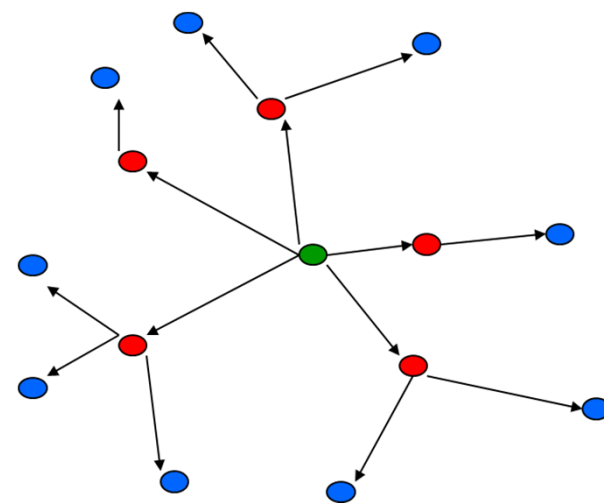
➤ 在TinyOS中的实现 `/tos/lib/ftsp`

❑ 用途：将所有节点同步到全局时间

❑ 方法：从根节点开始单向洪泛广播时间同步报文，广播域内的节点收到报文，进而知道本地时钟与广播源节点的时钟基准之间的差异，从而进行修正

❑ 主要接口：

```
interface GlobalTime<precision_tag>
{
    async command uint32_t getLocalTime();
    async command error_t getGlobalTime(uint32_t *time);
    //将本地时间time同步为全局时间
    async command error_t local2Global(uint32_t *time);
    async command error_t global2Local(uint32_t *time);
}
```



TinyOS

- TinyOS 简介
- 关键特性
- 编程实例

AntiTheft应用编程实例：功能升级

- ❑ 问题：一个聪明的小偷偷节点的时候不装到口袋里，这样刚才的程序就不能判断是否被偷
- ❑ 解法：小偷拿起节点时，节点的加速度会发生突变。
 - 小偷检测算法2：每隔1s, 以100Hz的频率采集加速度读数，检查方差是否超出给定阈值，如果超出，则闪红灯警告
- ❑ 本例包含
 - 安全机制
 - 使用任务来延迟计算密集型操作

AntiTheft应用编程实例：程序编写

□ 编写顶层配置文件AntiTheftAppC.nc

```
configuration AntiTheftAppC { }
```

```
implementation
```

```
{
```

```
  components AntiTheftC, MainC, LedsC;
```

```
  components new TimerMilliC() as MyTimer;
```

```
  components new AccelXStreamC();
```

声明所需要用到的组件

```
  AntiTheftC.Boot -> MainC.Boot;
```

```
  AntiTheftC.Leds -> LedsC;
```

```
  AntiTheftC.Check -> MyTimer;
```

```
  AntiTheftC.ReadStream -> AccelXStreamC;
```

```
}
```

组件接口间的绑定

指出AntiTheftC组件中使用的接口到底是由哪个组件提供的。

AntiTheft应用编程实例：程序编写

□ 编写功能组件文件AntiTheftC.nc

```
module AntiTheftC @safe {  
  uses interface Boot;  
  uses interface Timer<TMilli> as Check;  
  uses interface ReadStream;  
}
```

} 声明组件所使用的接口

```
implementation {  
  uint16_t accelSamples[ACCEL_SAMPLES];
```

```
event void Check.fired() {  
  call ReadStream.postBuffer(accelSamples, ACCEL_SAMPLES);  
  // 采样间隔为10000微秒，即采样频率为100Hz  
  call ReadStream.read(10000);  
}
```

} 添加一个缓冲区，每隔1s以100Hz的频率读传感器，直到缓冲区满

```
event void ReadStream.readDone(error_t ok, uint32_t actualPeriod)  
{  
  if (ok == SUCCESS)  
    post checkAcceleration();  
}
```

} 缓冲区满后触发readDone事件，提交方差计算和报告小偷的任务

```
task void checkAcceleration() {  
  ... check acceleration and report theft... }}
```


AntiTheft应用编程实例：程序编写

```
module AntiTheftC @safe {  
  uses interface Boot;  
  uses interface Timer<TMilli> as Timer;  
  uses interface ReadStream;  
}  
implementation {  
  uint16_t accelSamples[ACCEL_SAMPLES];
```

接口使用：

ReadStream是一个接口，用于将传感器读数定期采样到一个或多个缓冲区中

- postBuffer添加一个大小为ACCEL_SAMPLES的缓冲区
- read开始采样操作
- readDone 在最后一个缓冲区满时被触发

```
event void Timer.fired() {  
  call ReadStream.postBuffer(accelSamples, ACCEL_SAMPLES);  
  call ReadStream.read(10000); // 采样间隔为10000微秒，即采样频率为100Hz  
}
```

```
event void ReadStream.readDone(error_t ok, uint32_t actualPeriod) {
```

安全机制：

//为指针加入COUNT(n)标注，确保其始终指向有效数组；

//模块标注@ safe，表示以安全模式编译

```
interface ReadStream<val_t> {  
  command error_t postBuffer( val_t* COUNT(n) buf, uint16_t count);  
  command error_t read(uint32_t period);  
  event void readDone(error_t ok, uint32_t actualPeriod);  
}
```

AntiTheft应用编程实例：程序编写

```
uint16_t accelSamples[SAMPLES];
event void ReadStream.readDone(error_t ok, uint32_t
actualPeriod) {
    if (ok == SUCCESS)
        post checkAcceleration();
}
```

```
task void checkAcceleration() {
    uint16_t i, avg, var;
```

```
    for (avg = 0, i = 0; i < SAMPLES; i++)
        avg += accelSamples[i];
    avg /= SAMPLES;
```

```
    for (var = 0, i = 0; i < SAMPLES; i++)
    {
        int16_t diff = accelSamples[i] - avg;
        var += diff * diff;
    }
    if (var > 4 * SAMPLES) theftLed();
}
```

任务提交：

在readDone中，需要计算采样的方差。我们将这个计算密集型操作提交为任务

目录

- ❑ 物联网操作系统概述
- ❑ 物联网操作系统构成
- ❑ 关键特性
- ❑ TinyOS
- ❑ **Contiki**
- ❑ LiteOS
- ❑ AliOS Things
- ❑ 研究进展

Contiki

- ❑ 背景概述
- ❑ 关键特性
- ❑ 编程实例

Contiki背景概述

- ❑ 由瑞典计算机科学研究所的Adam Dunkels于2002年创建。
- ❑ Contiki这个名字来自于著名的挪威探险家托尔海尔达尔，当时，这位探险家正是乘坐着一个叫做康提基号（Kon-Tiki）的木筏，从秘鲁卡亚俄港航行到南太平洋图阿莫图岛（4,300海里）而闻名一时的。
- ❑ Contiki专为具有少量内存的MCU而设计。典型的Contiki配置是2,000字节的RAM和40,000字节的ROM。
- ❑ Contiki系统中包括了一个名为Cooja的网络模拟器，它可以模拟基于Contiki系统的网络。
- ❑ 为了在内存有限的系统上高效运行，Contiki的编程模型基于protothreads。protothread是一种可以节省内存的编程框架，它能够让程序员以多线程的风格编写事件驱动程序，同时减少程序使用的内存开销。

Contiki支持的硬件

□ 包括: cc2538dk、micaz、sky、CC2650等17+款。

平台	MCU和SoC	射频芯片
<u>RE-Mote</u>	TI CC2538	Integrated / CC1200
<u>nRF52 DK</u>	nRF52832	Integrated
<u>cc2538dk</u>	TI CC2538	Integrated
<u>exp5438</u> , <u>z1</u>	TI MSP430x	TI CC2420
<u>wismote</u>	TI MSP430x	TI CC2520
avr-raven, avr-rcb, avr-zigbit, iris	Atmel AVR	Atmel RF230
micaz	Atmel AVR	TI CC2420
<u>redbee-dev</u> , <u>redbee-econotag</u>	Freescale MC1322x	Integrated
sky	TI MSP430	TI CC2420
msb430	TI MSP430	TI CC1020
esb	TI MSP430	RFM TR1001
avr-atmega128rfa	Atmel Atmega128 RFA1	Integrated
cc2530dk	TI CC2530	Integrated
native_minimal-net_ccoia	Native	-

Contiki关键特性

- ❑ 编程模型
 - 基于protothreads的编程模型
- ❑ I/O操作方式
 - 阻塞和非阻塞方式
- ❑ 调度机制
 - 协作式调度
- ❑ 内存管理
 - 静态内存管理和动态内存管理
- ❑ 软件更
 - 模块化更新
- ❑ 网络支持
 - uIP、RPL、6LoWPAN

基于protothreads的编程模型

❑ Protothreads是什么

- 它由瑞士计算机学院的Adam Dunkels等人于2006年提出来的编程框架
- 基于协作式的调度模式
- 使用单个堆栈实现
- 不能使用局部变量

基于protothreads的编程模型

❑ 基础编程框架

- 所有的Contiki程序都以宏**PROCESS()**开头
- **AUTOSTART_PROCESSES()**用于在程序启动的时候自动的执行当前进程
- **PROCESS_THREAD()**为当前进程的线程实现
- **PROCESS_BEGIN()**必须在初始化的变量之后调用。其后，用户可以编写具体的程序逻辑。
- **PROCESS_END()**的调用表示当前线程的终止。

```
PROCESS(name, strname); // name:进程结构体; strname:进程名称
AUTOSTART_PROCESSES(struct process &name);
PROCESS_THREAD(name, process_event_t, process_data_t)
{ // process_event_t: 事件; process_data_t: 传递的进程数据
----Initialization of required variables----
PROCESS_BEGIN();
---Set of C statements---
PROCESS_END();
}
```

基于protothreads的编程模型

❑ 两个protothread线程，轮流输出“hello one”和“hello two”。

```
1.PROCESS(hello_one, "hello_one");
2.PROCESS(hello_two, "hello_two");
3.AUTOSTART_PROCESSES(&hello_one,&hello_two);
4.PROCESS_THREAD(hello_one, ev, data){
5.  static struct etimer  wtimer;
6.  PROCESS_BEGIN();
7.  etimer_set(&wtimer,
    CLOCK_CONF_SECOND); /* 1s*/
8.  while (1) {
9.    PROCESS_WAIT_EVENT_UNTIL(etimer_expired (wtimer));
10.    printf("Hello one.\n");
11.    etimer_reset(&wtimer);
12.  }
13.  PROCESS_END();
14.}
```

If time fired
: Yield

```
1.PROCESS_THREAD(hello_two, ev, data){
2.  static struct etimer  wtimer;
3.  PROCESS_BEGIN();
4.  etimer_set(&wtimer,
    CLOCK_CONF_SECOND*2); /* 2s*/
5.  while (1) {
6.    PROCESS_WAIT_EVENT_UNTIL(etimer_expired (wtimer));
7.    printf("Hello two.\n");
8.    etimer_reset(&wtimer);
9.  }
10.  PROCESS_END();
11.}
```

输出：
Hello one.
Hello one.
Hello two.
Hello one.
Hello one.
Hello two

基于protothreads的编程模型

❑ 实战理解protothreads编程。每隔2秒输出“demo”到控制台。

```
1.  PROCESS(contiki_program_process, "Contiki Program Demo");
2.  AUTOSTART_PROCESSES(&contiki_program_process);
3.  PROCESS_THREAD(contiki_program_process, ev, data){
4.      static struct etimer  wtimer;
5.      PROCESS_BEGIN();
6.      etimer_set(&wtimer, CLOCK_CONF_SECOND * 2); /* 2s*/
7.      while (1)  {
8.          PROCESS_WAIT_EVENT_UNTIL(etimer_expired (wtimer));
9.          printf("demo. \n");
10.         etimer_reset(&wtimer);
11.     }
12.     PROCESS_END();
13. }
```

基于protothreads的编程模型

❑ 3-13. protothread主要函数

```
3. PROCESS_THREAD(contiki_program_process, ev, data){
4.     static struct etimer   wtimer;
5.     PROCESS_BEGIN();
6.     etimer_set(&wtimer, CLOCK_CONF_SECOND * 2); /* 2s*/
7.     while (1) {
8.         PROCESS_WAIT_EVENT_UNTIL(etimer_expired (wtimer));
9.         printf("demo. \n");
10.        etimer_reset(&wtimer);
11.    }
12.    PROCESS_END();
13. }
```

基于protothreads的编程模型

❑ 3-13. protothread主要函数的宏转换成C语言代码

- 情况 1 wtimer 没有expired,执行了case 0, case 8, 返回1, yield
- 情况 2 wtimer expired,执行了case 8, 执行了printf

```
3.  PROCESS_THREAD(contiki_program_process, ev, data){
4.      static struct etimer  wtimer;
5.      { char PT_YIELD_FLAG = 1; switch((process_pt)->lc) { case 0;;
6.          etimer_set(&wtimer, CLOCK_CONF_SECOND * 2); /* 2s*/
7.          while (1)  {
8.              PT_YIELD_FLAG  =  0;  (process_pt)->lc  =  8;  case  8;;
          if((PT_YIELD_FLAG == 0) || !(etimer_expired (wtimer))) { return 1; }
9.          printf("demo. \n");
10.         etimer_reset(&wtimer);
11.     }
12. }; PT_YIELD_FLAG = 0; (process_pt)->lc = 0;; return 3; };
13. }
```

基于protothreads的编程模型

- ❑ 支持多线程的库mt_thread. 部分API。
- ❑ 支持**协作式调度**（主动yield）和**抢占式调度**（系统负责抢占）。

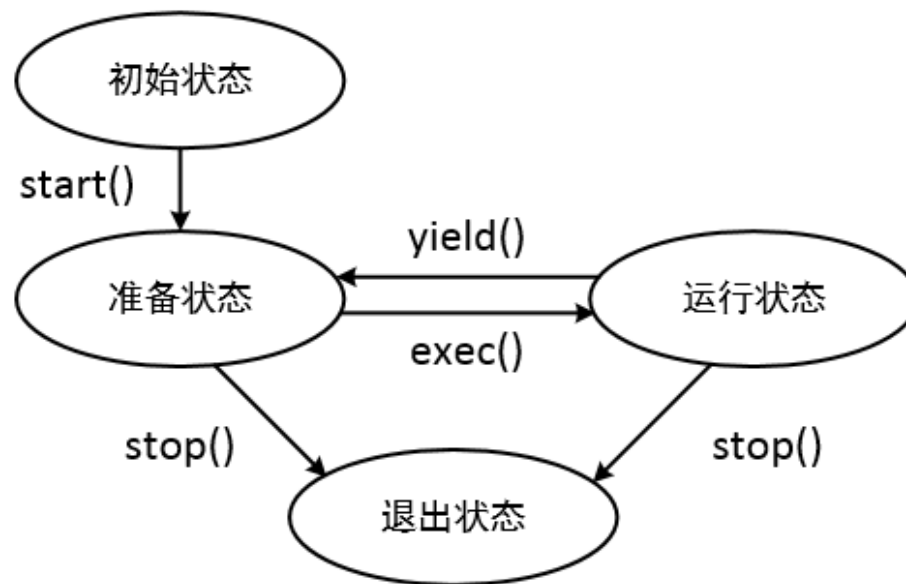
void mt_init (void) : 初始化库。

void mt_start (struct mt_thread * thread , void (* function) (void *) , void * data) : 启动一个线程。

void mt_exit (void) : 退出一个线程。

void mt_exec (struct mt_thread * thread) : 执行线程。

void mt_stop (struct mt_thread * thread) : 终止一个线程。



基于protothreads的编程模型

❑ **mt_thread的样例。创建两个线程，轮流输出one、two.**

```
1. PROCESS_THREAD(multi_threading_process, ev, data){
2.     static struct mt_thread alpha_thread;
3.     static struct mt_thread count_thread;
4.     static int c1, c2;
5.     PROCESS_BEGIN();
6.     mt_init();
7.     mt_start(&alpha_thread, thread_main, "one");
8.     mt_start(&count_thread, thread_main, "two");
9.     mt_exec(&alpha_thread);
10.    mt_exec(&count_thread);
11.    while(1) {
12.        c1++;c2++;
13.        if(c1>5 && c2 > 5){
14.            mt_stop(&alpha_thread);
15.            mt_stop(&count_thread);
16.            break;
17.        }
18.    }
19.    PROCESS_END();
20.}
```

```
static void
thread_main(void *data)
{
    while(1) {
        puts(data);
    }
}
```

网络协议栈的支持

❑ 6LoWPAN

- 实现IPv6的地址压缩、TCP/UDP的包头压缩、ICMP包头压缩
- Contiki6LowPAN使用样例:将128bit的ipv6地址压缩成16bit。

```
compress_addr_64(SICSLOWPAN_IPHC_SAM_BIT,  
                &UIP_IP_BUF->srcipaddr);
```

❑ RPL

- 支持低功耗的路由协议RPL
- 与TinyOS中的数据收集协议CTP相比, Contiki实现的RPL支持更多的数据传输功能。比如**多对一**、**一对多**和**一对一**的数据传输。

网络协议栈的支持

❑ uIP

- 传统的TCP/IP协议栈还包含RIP、OSPF、BGP等协议，相比于它，uIP将TCP/IP协议栈**精简为必需的最小功能集合**，它仅包含了IP、ICMP、UDP和TCP协议。

Server 部分API:

```
uip_listen(); uip_send();
```

Client 部分API:

```
uip_send(); uip_udp_new()
```

- 使用样例。Server监听45端口，client建立连接发送数据。

Server 代码：

```
uip_listen(HTONS(45));  
If(uip_newdata()){  
    uip_send("ok", 2);  
}
```

Client 代码：

```
Uip_ipaddr(ipaddr, 192.168.0.1)  
Uip_connect(ipaddr, HTONS(45))  
If(uip_connected()){  
    uip_send("hello", 5);  
}
```

Contiki环境搭建

❑ 下载Contiki的源代码

- `wget https://github.com/contiki-os/contiki/archive/3.0.zip`
- `unzip 3.0.zip`
- `mv contiki-3.0 contiki`
- `sudo apt-get install build-essential binutils-msp430 gcc-msp430 msp430-libc msp430mcu mspdebug gcc-arm-none-eabi gdb-arm-none-eabi openjdk-8-jdk openjdk-8-jre ant libncurses5-dev`

❑ 进入目录contiki/example/hello-world/测试验证

- `$make TARGET=native`
- `$/hello-world`

编程实例-Antitheft

- ❑ 案例描述：Antitheft，如何检测传感器被偷
- ❑ 问题：一个聪明的小偷偷节点的时候不装到口袋里，这样刚才的程序就不能判断是否被偷
- ❑ 解法：小偷拿起节点放入口袋中时，采集的光照强度急剧减弱。
 - 每隔一秒采集光照数据，检测光照强度是否小于一定阈值

编程实例-Antitheft

❑ 1. 每隔1秒采集光照数据，看是否存在小偷遮挡

```
PROCESS(sensor_acq_process,"Sensor Acquisition");
AUTOSTART_PROCESSES(&sensor_acq_process);
PROCESS_THREAD(sensor_acq_process,ev,data){
    static struct etimer et;
    static int val;
    PROCESS_BEGIN();
    while(1){
        etimer_set(&et, CLOCK_SECOND);    //设置每隔一秒采集光照数据
        SENSORS_ACTIVATE(light_sensor); //激活光照传感器
        PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
        val = light_sensor.value(LIGHT_SENSOR_TOTAL_SOLAR); // 获取光照数据
        if(val != -1 && val < 200) {          // 验证光照数据，判断是否存在遮挡
            leds_on(LED_RED);                // 发现小偷，亮红灯
        }
        etimer_reset(&et);
        SENSORS_DEACTIVATE(light_sensor); // 取消激活光照传感器
    }
    PROCESS_END();
}
```

anti-theft.c

编程实例-Antitheft

❑ 2. 编写MakeFile文件

```
CONTIKI_PROJECT = anti-theft-demo  
  
all: $(CONTIKI_PROJECT)  
  
CONTIKI = ../..  
include $(CONTIKI)/Makefile.include
```

❑ 3. 使用motelist命令发现节点的串口为/dev/ttyUSB0, 使用下面的命令烧写程序

```
$ make TARGET=sky MOTES=/dev/ttyUSB0 anti-theft-demo.upload
```

目录

- ❑ 物联网操作系统概述
- ❑ 物联网操作系统构成
- ❑ 关键特性
- ❑ TinyOS
- ❑ Contiki
- ❑ LiteOS
- ❑ AliOS Things
- ❑ 研究进展

产生及定义

□ 产生背景

- 2015年，华为在HNC网络大会上，正式推出了“1+2+1”物联网战略，即“一个物联网平台，两种接入方式，一个轻量级物联网操作系统”，Huawei LiteOS于是就应运而生。

□ 定义

- Huawei LiteOS是轻量级(典型配置96KB ROM, 64KB RAM)的实时操作系统，遵循BSD-3开源许可协议，能大幅降低设备布置及维护成本，有效降低开发门槛、缩短开发周期。

基本框架

□ LiteOS架构



硬件支持列表

□ LiteOS支持的硬件 (things)

厂商	型号
ST	STM32L476-Nucleo、STM32F429-Discovery、STM32F746-Nucleo、STM32F411-Nucleo
GD	GD32F450i-EVAL、GD32F190R-EVAL
Atmel	ARDUINO M0 PRO、ARDUINO M0 PRO、ARDUINO M0 PRO
NXP	FRDM-KW41Z、FRDM-KL26Z、LPC824Lite、LPC54110 Board
MIDMOTION	MM32F103、MM32L373、MM32L073PF
Silicon Labs	EFM32 GIANT GECKO STARTER KIT、EFM32 HAPPY GECKO STARTER KIT、EFM32 PEARL GECKO STARTER KIT
秉火	秉火指南者STM32F103
联盛德	W500 G2-WM500

LiteOS关键特性

❑ 编程模型

- 基于事件和线程的编程模型

❑ I/O操作方式

- 阻塞和非阻塞方式

❑ 调度机制

- 实时抢占调度、时间片轮转调度

❑ 内存管理

- 静态内存管理、动态内存管理

❑ 软件更新

- 支持整个镜像更新和模块化更新

❑ 网络服务

- 支持BT、WiFi、6LoWPAN、ZigBee、LTE、NB-IOT等

❑ 安全机制

- 可信应用签名、DTLS/TLS加密传输

编程模型

❑ 开发语言：C语言

- 开发者只要懂得C语言的基本语法，就可以基于LiteOS进行物联网应用的开发， LiteOS支持线程和事件

❑ 多线程开发

- 下面的示例将介绍线程的基本操作方法：低优先级线程持续运行，会被高优先级线程打断，直到高优先级线程调用 `LOS_TaskDelay` 后，才将 CPU 释放出来

❑ 创建2个线程: TaskHigh和TaskLow

- TaskHigh为高优先级线程
- TaskLow为低优先级线程

线程

❑ 使用 `LOS_TaskCreate` 创建两个线程，线程优先级分别为 4 和 5

```
1. U_INT32 Example05_Entry(VOID) {
2.     U_INT32 uwRet = LOS_OK;
3.     TSK_INIT_PARAM_S stInitParam = {0};
4.     printf("Example05_Entry\r\n");
5.     stInitParam.pfnTaskEntry = Example_TaskHi; // 创建高优先级线程
6.     stInitParam.usTaskPrio = TASK_PRIO_HI;
7.     stInitParam.pcName = "TaskHi";
8.     stInitParam.uwStackSize = TASK_STK_SIZE;
9.     stInitParam.uwArg = (U_INT32)pcTextForTaskHi;
10.    uwRet = LOS_TaskCreate(&s_uwTskHiID, &stInitParam);
11.    if (uwRet != LOS_OK) {
12.        printf("Example_TaskHi create Failed!\r\n");
13.        return LOS_NOK;
14.    }
```

线程

❑ 使用 LOS_TaskCreate 创建两个线程，线程优先级分别为 4 和 5

```
1.  stInitParam.pfnTaskEntry = Example_TaskLo;//创建低优先级线程
2.  stInitParam.usTaskPrio = TASK_PRIO_LO;
3.  stInitParam.pcName = "TaskLo";
4.  stInitParam.uwStackSize = TASK_STK_SIZE;
5.  stInitParam.uwArg = (UINT32)pcTextForTaskLo;
6.  uwRet = LOS_TaskCreate(&s_uwTskLoID, &stInitParam);
7.  if (uwRet != LOS_OK) {
8.      printf("Example_TaskLo create Failed!\r\n");
9.      return LOS_NOK;
10. }
11. return uwRet;
12.}
```

线程

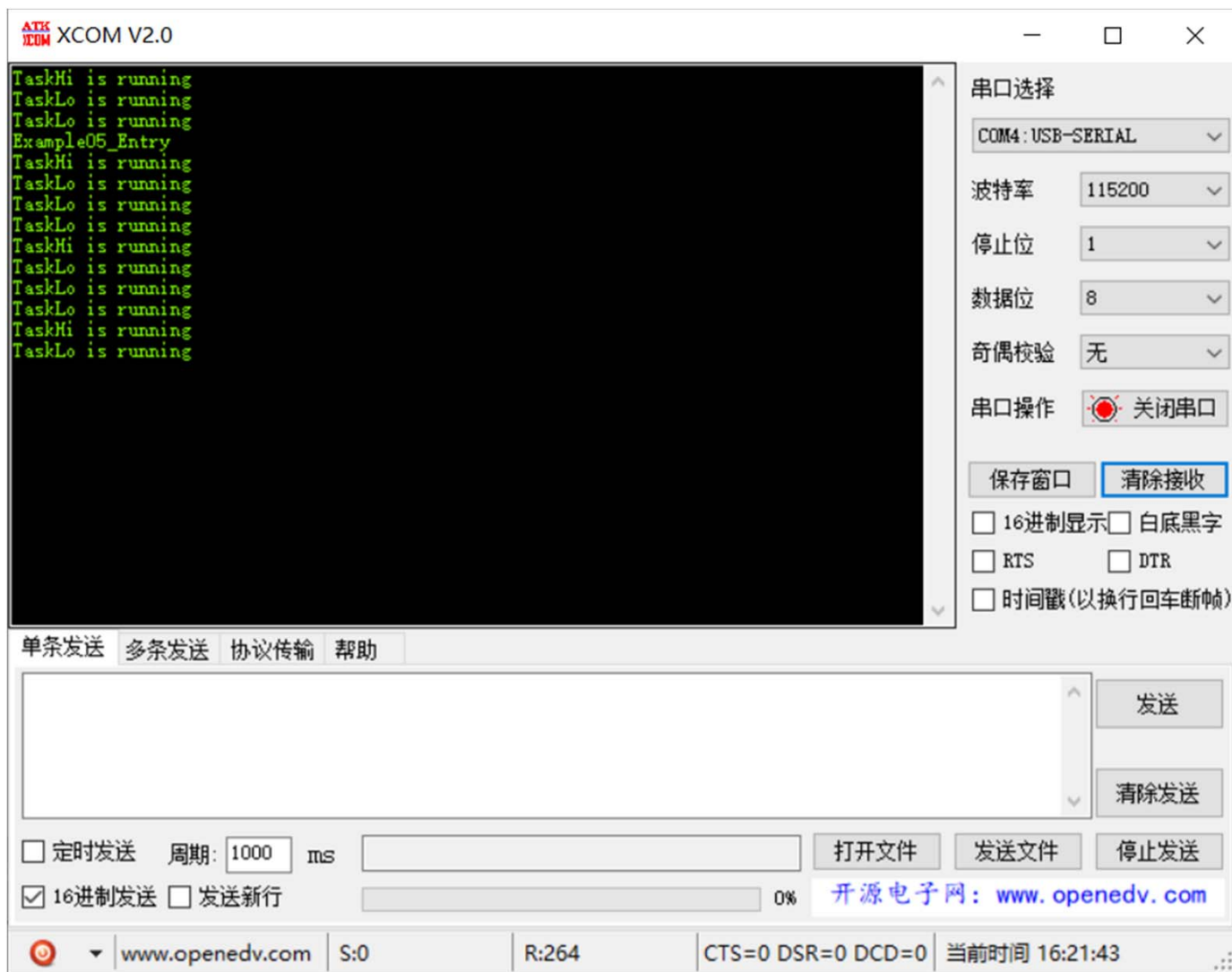
❑ 线程1 和 线程2 周期性打印字符串

```
1.static VOID * Example_TaskHi(UINT32 uwArg) {//高优先级线程
2.    for (;;) {
3.        printf("%s\r\n", (const CHAR *)uwArg);
4.        LOS_TaskDelay(2000);
5.    }
6.}

7.static VOID * Example_TaskLo(UINT32 uwArg) {//低优先级线程
8.    UINT32 i;
9.    for (;;) {
10.        printf("%s\r\n", (const CHAR *)uwArg);
11.        for (i = 0; i < TASK_LOOP_COUNT; i++) {
12.            // 占用CPU耗时运行，过程中会被高优先级线程打断
13.        }
14.    }
15.}
```

线程

运行结果



目录

- ❑ 物联网操作系统概述
- ❑ 物联网操作系统构成
- ❑ 关键特性
- ❑ TinyOS
- ❑ Contiki
- ❑ LiteOS
- ❑ **AliOS Things**
- ❑ 研究进展

AliOS Things概述

- ❑ 以驱动万物智能为目标
- ❑ 面向物联网领域的轻量级物联网嵌入式操作系统
- ❑ 适用于广泛的小型物联网基础设备，可应用在智能家居、智慧城市等领域
- ❑ 具备极致性能，极简开发、云端一体、丰富组件、安全防护等关键能力
- ❑ 为各行各业提供一站式的物联网解决方案，构建物联网云端一体化生态，使物联网终端更加智能



AliOS Things特点



开发简易

支持C和JavaScript语言进行开发
提供IDE，支持编译、调试、内存泄漏检测等
支持GDB/Valgrind/Perf等常用linux工具



服务丰富

连接组件支持设备以不同方式接入物联网
支持自行组织网络和实现设备间本地互连
深度定制和优化的网络协议栈



优异性能

小FootPrint，内核ROM占用小于2KB
低功耗，内核提供了空闲CPU模式
多种调度策略提供较好的实时性



更新便捷

使用FOTA（无线固件升级）更新设备固件
支持独立升级，及多bin、差分和乒乓升级
提供OTA HAL以便轻松移植

关键特性

□ 编程模型

- 基于事件和多线程的编程模型

□ I/O操作方式

- 阻塞和非阻塞方式

□ 调度机制

- 抢占式调度和时间片轮转式调度

□ 内存管理

- 静态内存管理和动态内存管理

□ 软件更新

- 整个镜像更新、模块化更新和差分更新

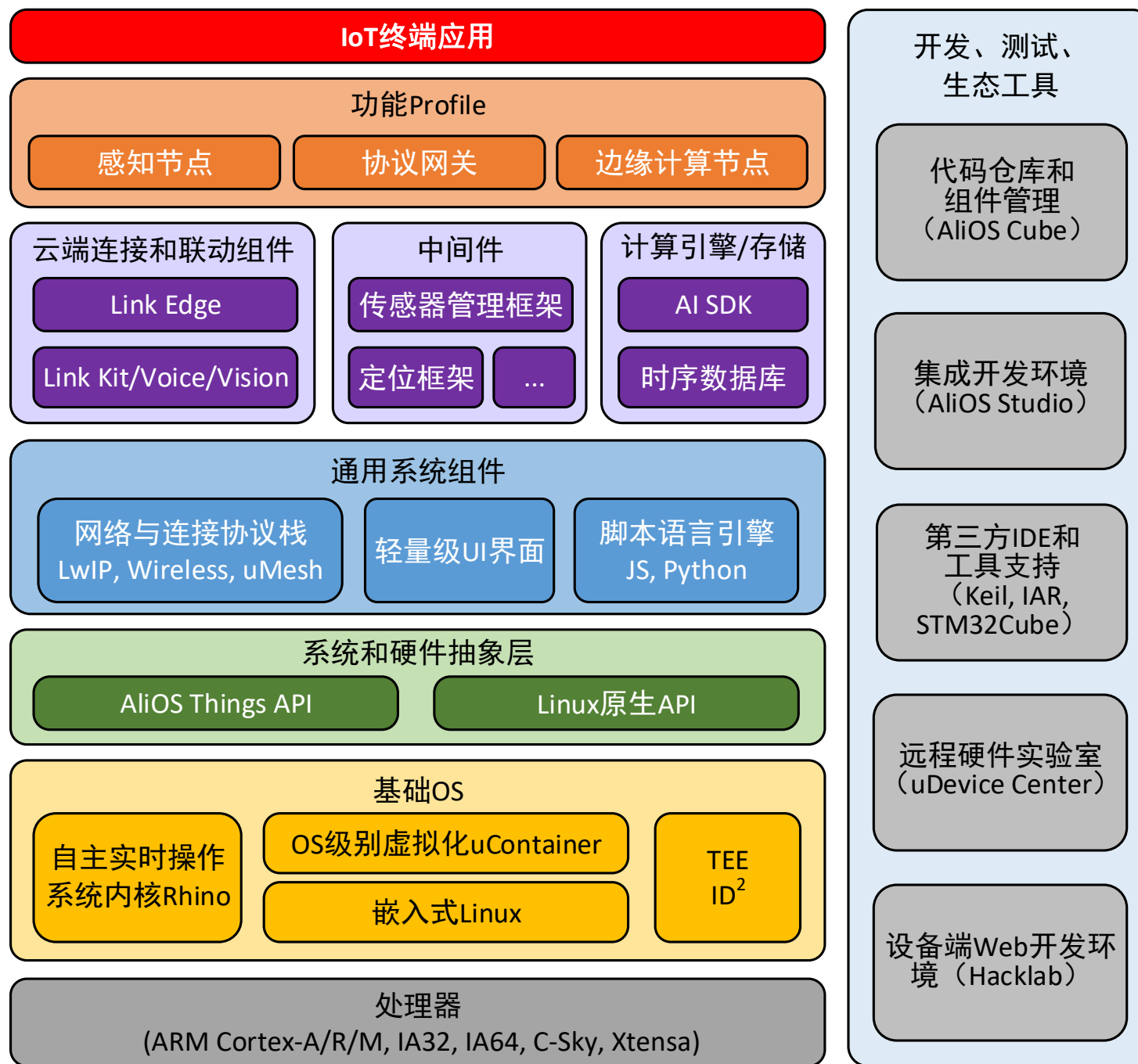
□ 网络支持

- 支持LwIP协议栈、uMesh和LoRaWAN等

□ 安全机制

- 可信执行环境、安全传输协议和可信身份标识等

AliOS Things框架



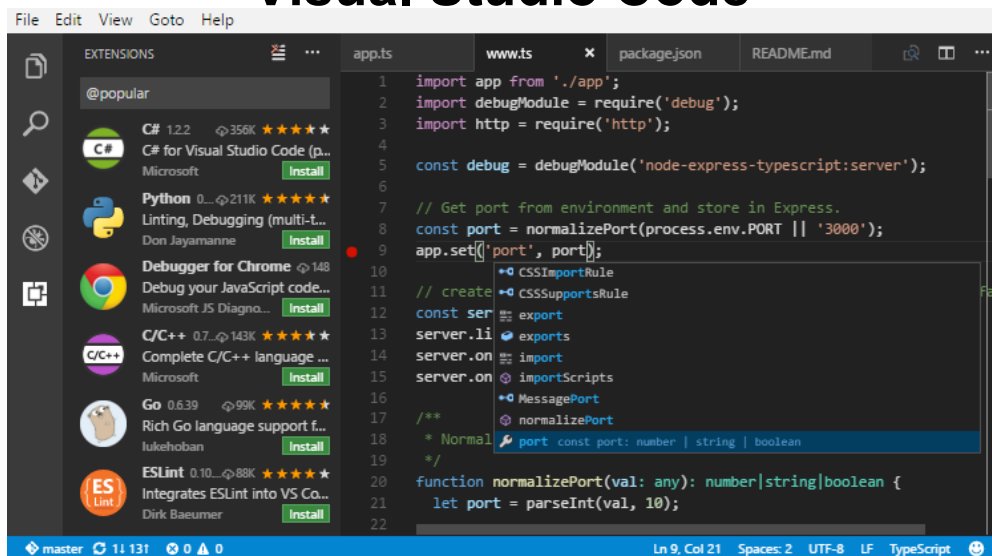
支持开发板

开发板	芯片	主频(MHz)	Flash (KB)	RAM (KB)	无线连接
<u>Developer kit</u>	stm32L496VGTx	80	1024	320	—
<u>Starter kit</u>	stm321433	80	256	64	WiFi
b_1475e	stm321475	80	1024	128	WiFi
stm32f429zi-nucleo	stm32f429zi	180	2048	256	WiFi
mk3060	mc108	120	2048	256	WiFi
esp8266	esp8266	160	4028	50	WiFi
esp32devkitc	esp32	240	448	520	WiFi

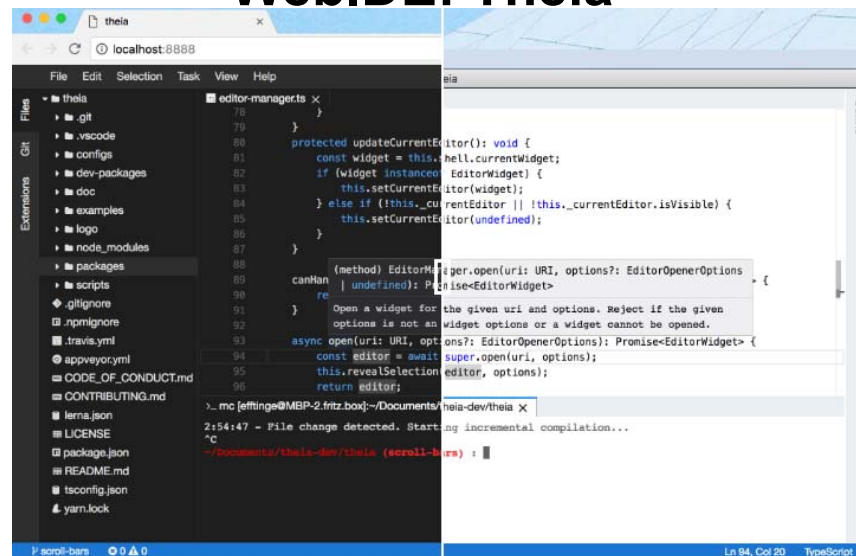
特多语言支持和IDE

- ❑ 多种开发语言：C、JavaScript
- ❑ 强大的IDE：Visual Studio Code, WebIDE
 - 代码编辑与编译、调试、内存泄漏检测
 - 多种常用Linux平台工具
 - 调试工具GDB
 - 测试工具Valgrind与Perf
 - 支持微内核基础上的POSIX 1003实现

Visual Studio Code



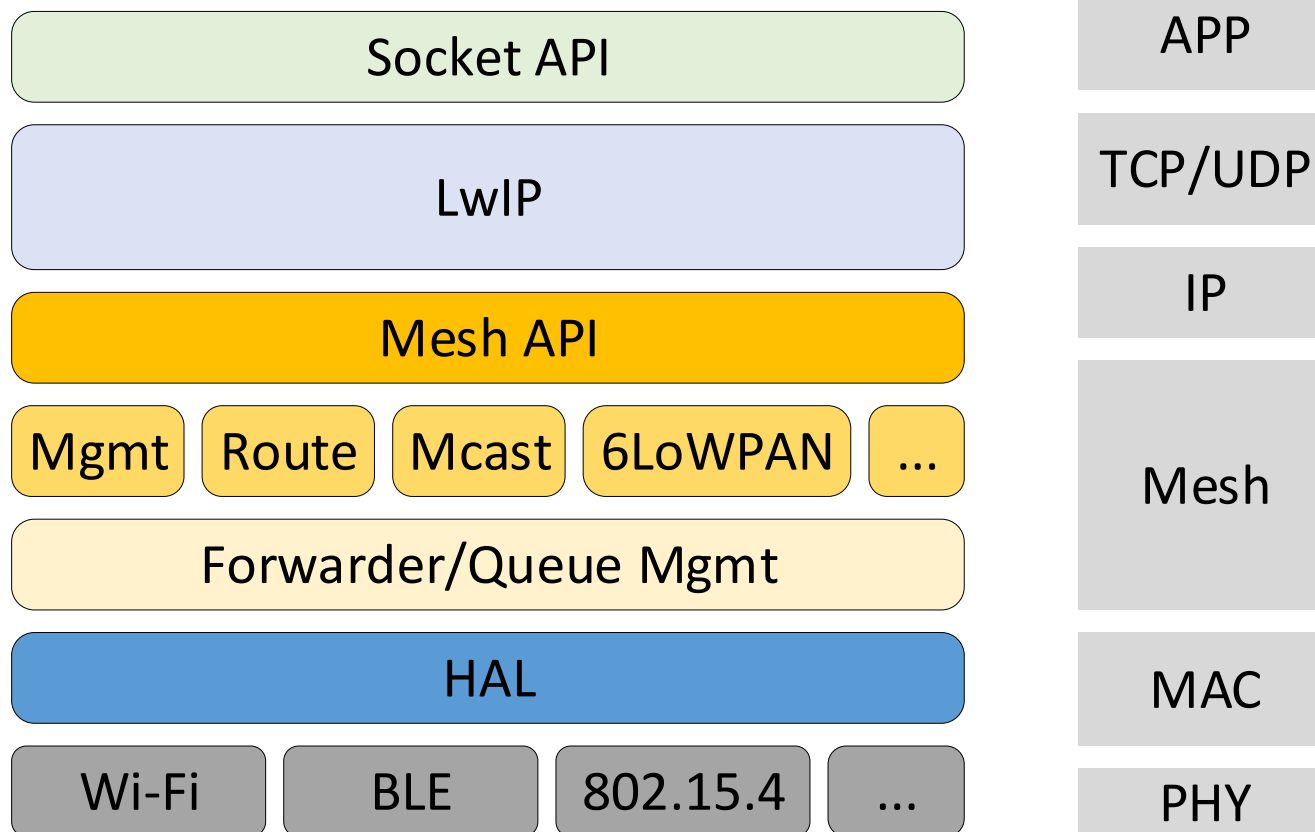
WebIDE: Theia



丰富的网络服务

❑ uMesh即插即用网络组件

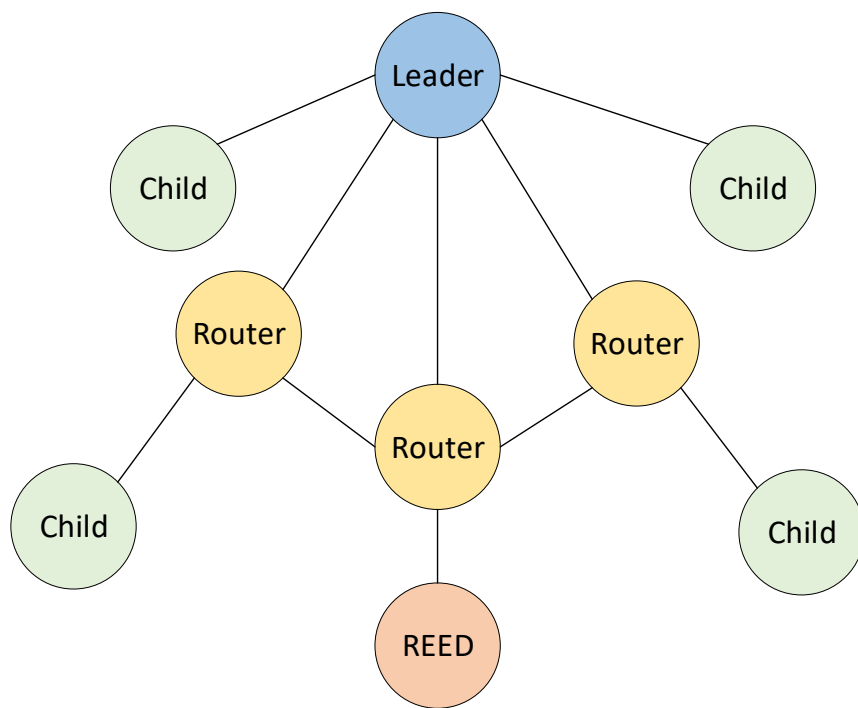
- 自组织，自我修复和多跳
- 适用于需要大规模部署的场景
 - 如室内定位，智能照明和商业环境
- 位于MAC层之上且位于IP层之下



丰富的网络服务

❑ uMesh即插即用网络组件

- 支持IPv4、IPv6
- 支持各种通信媒体，如WiFi、BLE、802.15.4
- 支持树形拓扑和网状拓扑
- 支持低功耗
- 可以使用ID²对设备进行认证，并通过AES-128加密数据



丰富的网络服务

❑ TCP/IP网络协议栈 (LwIP)

- AliOS Things使用深度定制和优化的TCP/IP网络协议栈
- 支持包括如下的协议
 - *IPv4/IPv6、TCP/UDP、ICMP、ARP、DHCP*等
- 相比于uIP，LwIP能提供更多的网络服务
- 对多并发连接和大数据量的场景有着深入的优化

❑ LoRaWAN连接协议

- 支持LoRaWAN的Class A和Class C的两种模式开发
 - *Class A: 低功耗传输协议，仅有数据传输时醒来*
 - *Class C: 节点持续监听信道，不会进入休眠*
- 提供了完整的LoRaWAN开发和测试环境

优异的性能

❑ 基于AliOS Things团队自主研发的Rhino RTOS内核

❑ 小FootPrint

- 在通常状态下，ROM占用约40KB，RAM占用约20KB
- 为内核对象提供静态和动态两种分配方式
- 专为小内存块管理而设计的内存分配器，可以同时支持固定大小块和可变大小块，还可以有多个内存区域
- 可对组件进行自由组装，使得最终镜像尽可能的小

❑ 实时性

- 提供两种调度策略，抢占式调度和时间片轮转式调度
 - 当任务的优先级高于当前运行的任务时，基于优先级的抢占式调度器会抢占CPU，内核会立即保存当前任务的上下文，并切换到优先级较高的任务的上下文。
 - 循环调度器通过将时间分片，在任务之间共享CPU资源。任务逐个运行，都不会抢占处理器，直到自己的时间片结束。

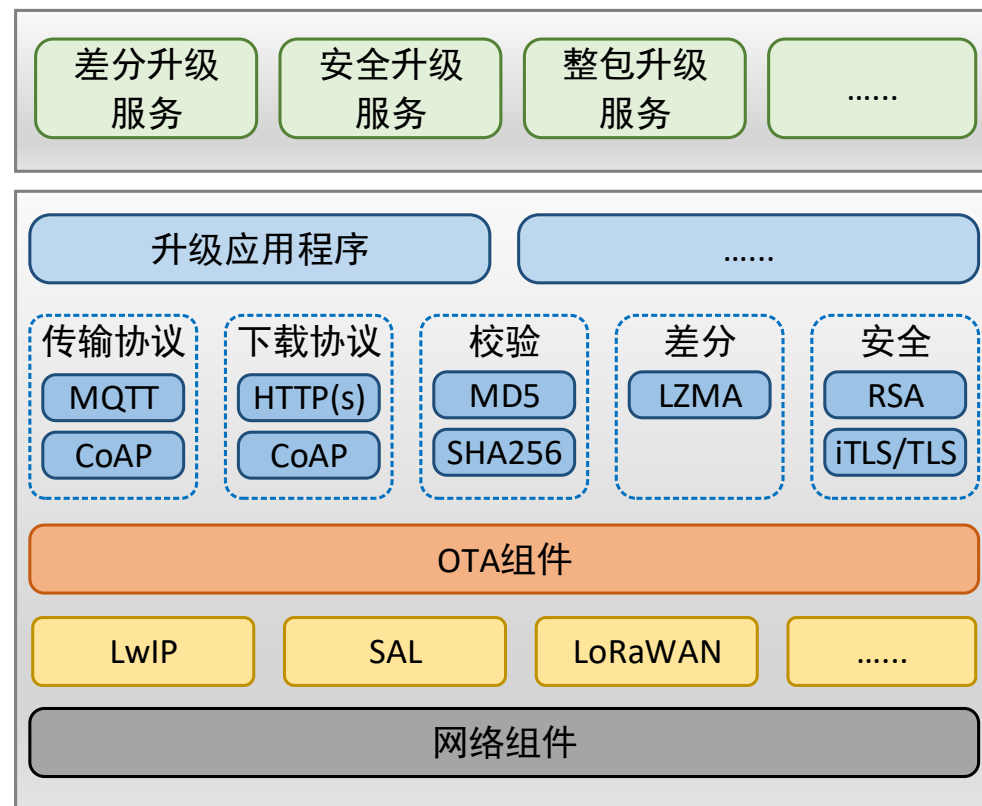
极度优化的性能

❑ 低功耗

- 大多数IoT设备电池资源是有限的，考虑系统功耗是非常重要的。系统消耗的越快，设备的使用时长就越短。
- Rhino内核提供了空闲CPU模式，帮助系统节省电量并延长其使用时间
 - 通常当CPU空闲时将执行处理器的原生指令（ARM处理器的WFI，IA32处理器的HLT）进入低功耗状态，在维持CPU寄存器上下文时，系统时钟中断每嘀嗒一次唤醒CPU。
 - Rhino内核提供了空闲CPU模式，当操作系统检测到固定持续时间内将空闲时，将CPU置于C1状态（ARM处理器为WFI，IA32处理器为HLT），并设置在此时间后触发中断，阻止此时间段内的系统时钟中断，CPU不会唤醒。直到空闲时间过去，再次触发系统时钟中断，将CPU从C1唤醒到C0状态，并且补偿该时间内的系统时钟计数。

细粒度系统更新

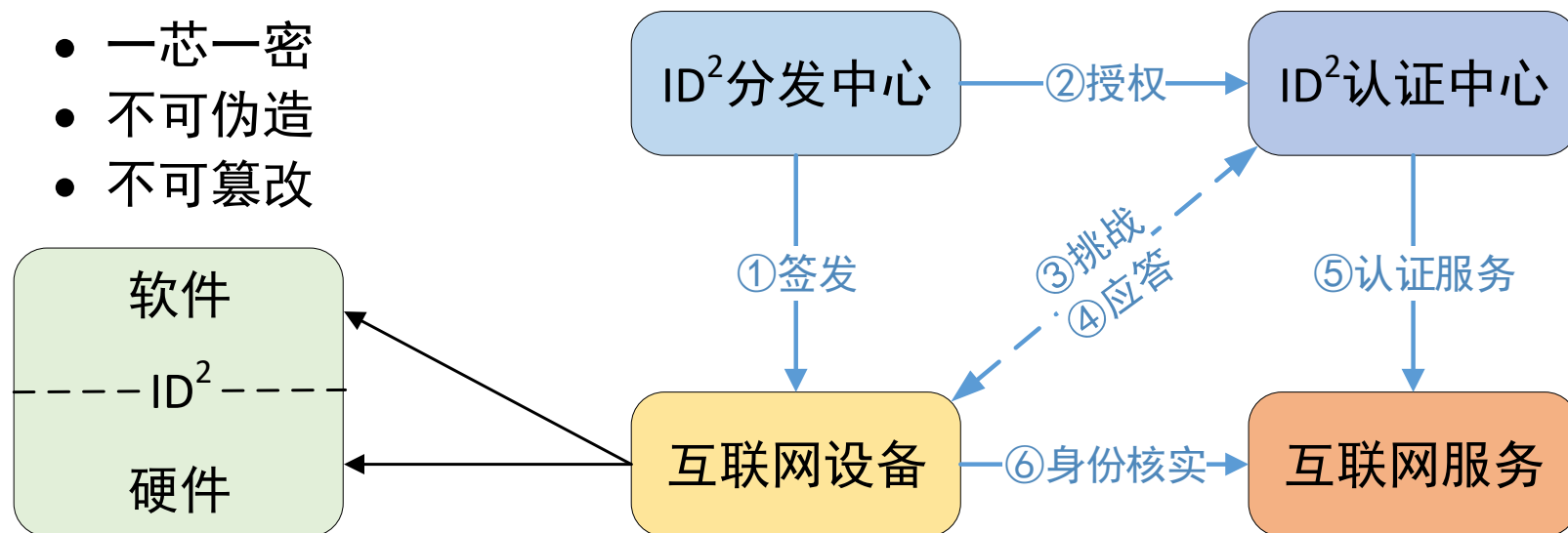
- ❑ FOTA（无线固件升级）使设备固件能够轻松更新
- ❑ AliOS Things根据硬件配置提供定制化的FOTA解决方案
 - AliOS Things可以分为Kernel和Application两部分
 - 支持Application独立升级，减少传输和储存开销
 - 通过细粒度的FOTA服务，支持多Bin或差分升级
- ❑ 支持丰富的物联网协议
 - Alink / MQTT / CoAP
- ❑ 提供OTA HAL以便轻松移植



FOTA软件框架

全方位安全防护

- ❑ AliOS Things既提供系统级别的安全保护，也提供芯片级别的安全保护（需将ID²烧录到安全芯片等载体里）
- ❑ 支持可信运行环境、ID²根身份证和对称/非对称密钥，以及基于此的可信连接和安全服务
- ❑ 阿里云Link ID² (Internet Device ID)
 - 物联网设备的可信身份标识
 - 具备不可篡改、不可伪造、全球唯一的安全属性
 - 是实现万物互联、服务流转的关键基础设施



全方位安全防护

❑ ID²生态中有以下两个角色：烧录者、使用者

- 烧录者负责申请ID²并将ID²烧录到安全芯片等载体里，这个角色通常由SE芯片商/卡商、SIM卡商、Secure MCU芯片商、设备商（采用TEE或软沙盒方案时）承担
- 使用者：将ID²安全载体集成到设备中，并利用ID²构建业务安全。这个角色通常是设备商（OEM）或者方案商（ODM）承担

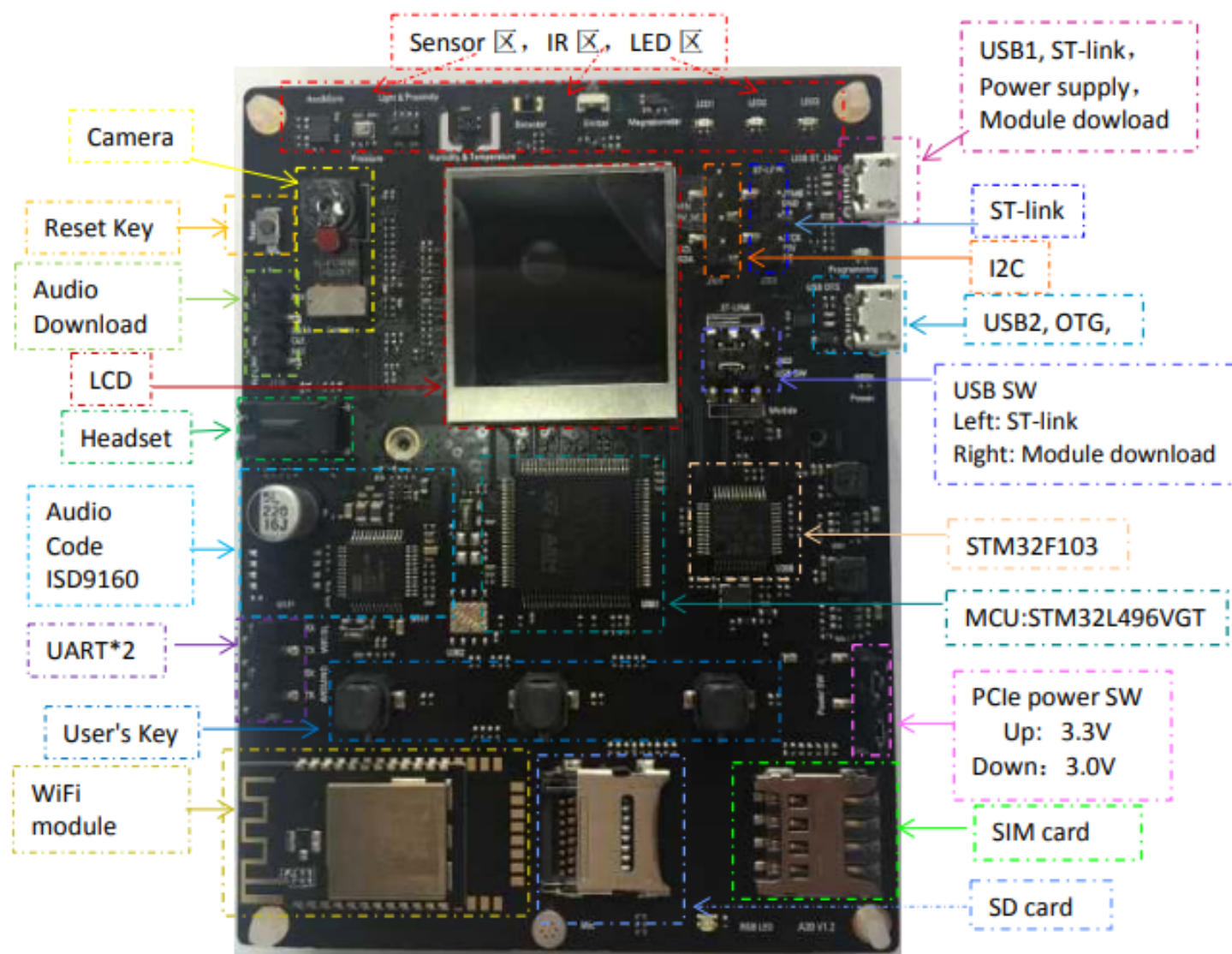
❑ ID²产线烧录SDK是由阿里云IoT开发和维护，目前支持的ID²密钥类型列表如下：

- 非对称密钥
 - *RSA-1024*
- 对称密钥
 - *3DES-112、3DES-168、AES-128、AES-192、AES-256*

案例

□ 搭建软硬件环境

➤ 案例使用AliOS官方淘宝店的AliOS Developer Kit开发板



案例

❑ 搭建软硬件环境

- 案例使用AliOS官方淘宝店的AliOS Developer Kit开发板
- AliOS Things支持命令行工具和开发IDE——VS Code
 - 本案例使用命令行工具，安装方式为先安装Python，然后安装[*aos cube*](#)，命令为

o `pip install aos-cube`

❑ 案例目的

- 完成开发环境搭建后，开始编写一个防小偷程序，功能主要是每隔1秒读一次环境光数据，根据光强判断是否被小偷放入口袋

❑ 工程目录结构

```
example/  
├── anti_theft  
│   ├── anti_theft.cpp           #主要程序  
│   ├── anti_theft.mk           #该工程的Makefile文件  
│   └── k app config.h          #App配置文件
```


案例

❑ Anti_theft.cpp

```
#include <aos/aos.h>
#include <hal/soc/soc.h>
#include <hal/sensor.h>
.....
#define GPIO_LED_IO 18
gpio_dev_t led;          /* 声明LED设备 */
static int fd_als = -1;   /* 声明光照传感器文件标识符 */
aos_timer_t refresh_timer; /* 声明timer */

void app_init() {
    led.port = GPIO_LED_IO;          /* 配置LED灯的GPIO */
    led.config = OUTPUT_PUSH_PULL;   /* 设置为输出模式 */
    hal_gpio_init(&led);              /* 初始化LED灯 */
    fd_als = aos_open(dev_als_path, O_RDWR); /* 打开光照传感器 */
    /* 创建定时器, 每隔1秒执行一次sensor_refresh_task */
    aos_timer_new(&refresh_timer, sensor_refresh_task, NULL, 1000, 1);
}

static void sensor_refresh_task(void *arg) {
    uint32_t lux = 0;
    get_als_data(&lux);              /* 读取光照传感器数据 */
    if (lux <= 200) {
        hal_gpio_output_toggle(&led); /* 亮LED灯提醒用户有小偷 */
    }
}
```

案例

❑ Anti_theft.cpp

```
static int get_als_data(uint32_t *lux) /* 读取光照传感器数据函数 */
{
    als_data_t data = { 0 };
    ssize_t size = 0;
    size = aos_read(fd_als, &data, sizeof(data)); /* 读取光照传感器文件标识符数据 */
    if (size != sizeof(data)) {
        printf("aos_read return error.\n");
        return -1;
    }
    *lux = data.lux;
    return 0;
}

int application_start(int argc, char *argv[]) { /* AliOS Things程序入口 */
    printf("Application starts.\n");
    app_init();
    aos_loop_run();
    return 0;
}
```

总结：IoT OS及其关键特性

操作系统	编程模型			调度方式	I/O操作方式		内存分配	软件更新			安全机制	
	模块化	事件驱动	多线程		阻塞I/O	分阶段I/O		整个镜像升级	模块化升级	差分升级	系统安全	传输安全
TinyOS	√	√	√	非抢占式	√ (TOSThreads)	√	静态	√	√ (TOSThreads)	×	类型和内存安全检查 (Safe TinyOS)	×
Contiki	√		√	协同式	√		静态 (默认)、动态	√	√	×	×	×
AliOS	√	√	√	抢占式/时间片轮转	√	√	静态、动态	√	√	√	可信执行环境	安全传输层协议、可信身份标识
LiteOS	√		√	抢占式/时间片轮转	√		静态、动态	√	√	√	可信应用签名	安全传输层协议

目录

- ❑ 物联网操作系统概述
- ❑ 物联网操作系统构成
- ❑ 关键特性
- ❑ TinyOS
- ❑ Contiki
- ❑ LiteOS
- ❑ AliOS Things
- ❑ 研究进展

研究进展

❑ 高安全性

- 物联网设备处理着用户的隐私数据。对应用安全性要求更高

❑ 高可拓展性

- 对多种语言、多种协议的支持

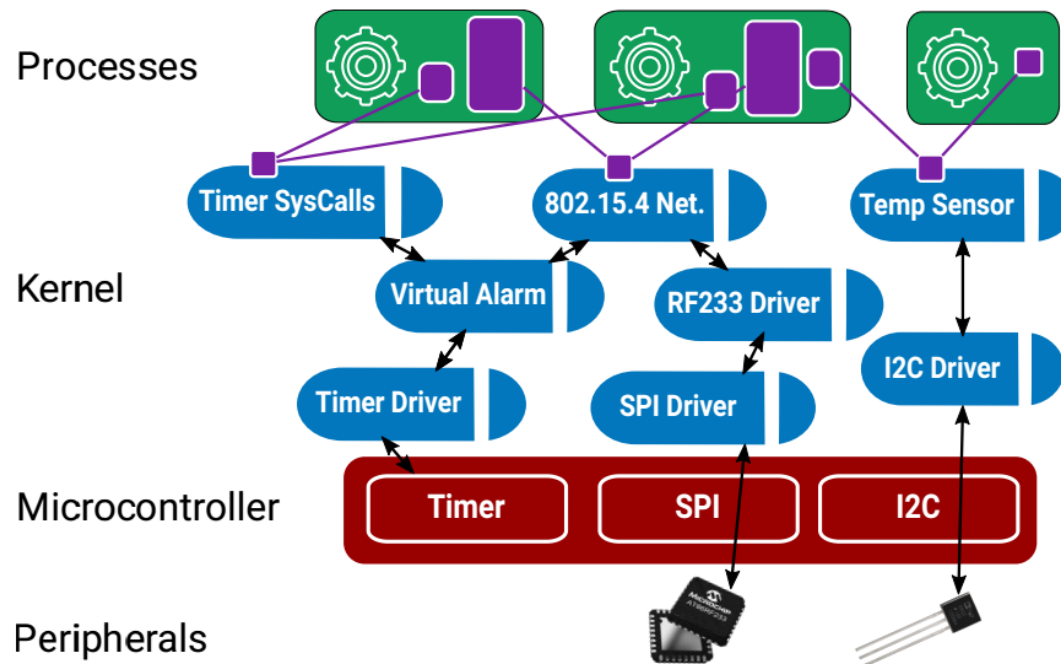
❑ 环境自适应性

- 物联网设备所处环境各异。OS需根据环境做出调度策略的改变。

研究进展

❑ [sosp17] Multiprogramming a 64 kB Computer Safely and Efficiently

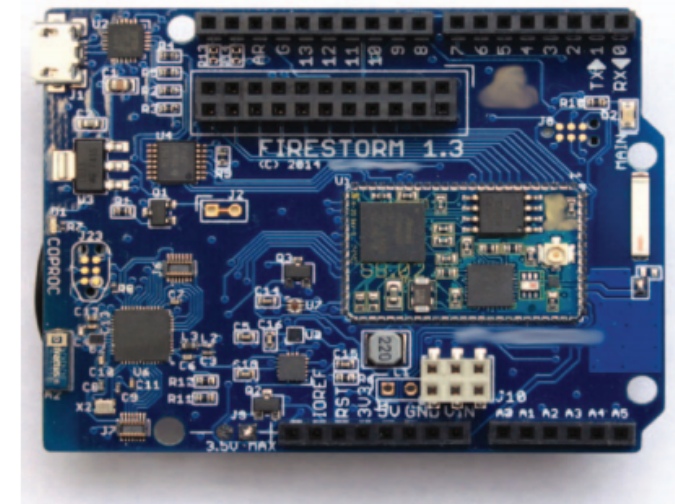
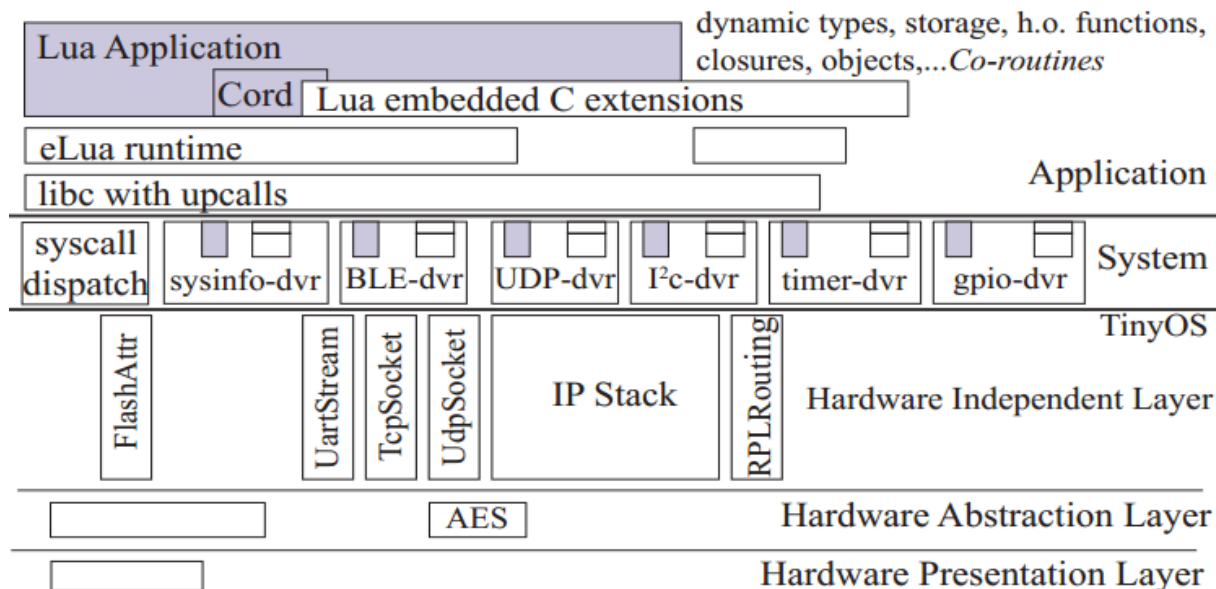
- 为嵌入式设备提供了一个新的软件系统Tock，它能够隔离软件错误、提供内存保护和有效的管理内存。
- 基于类型安全语言Rust编写的内核，能够隔离错误、保护内存



Tock系统架构

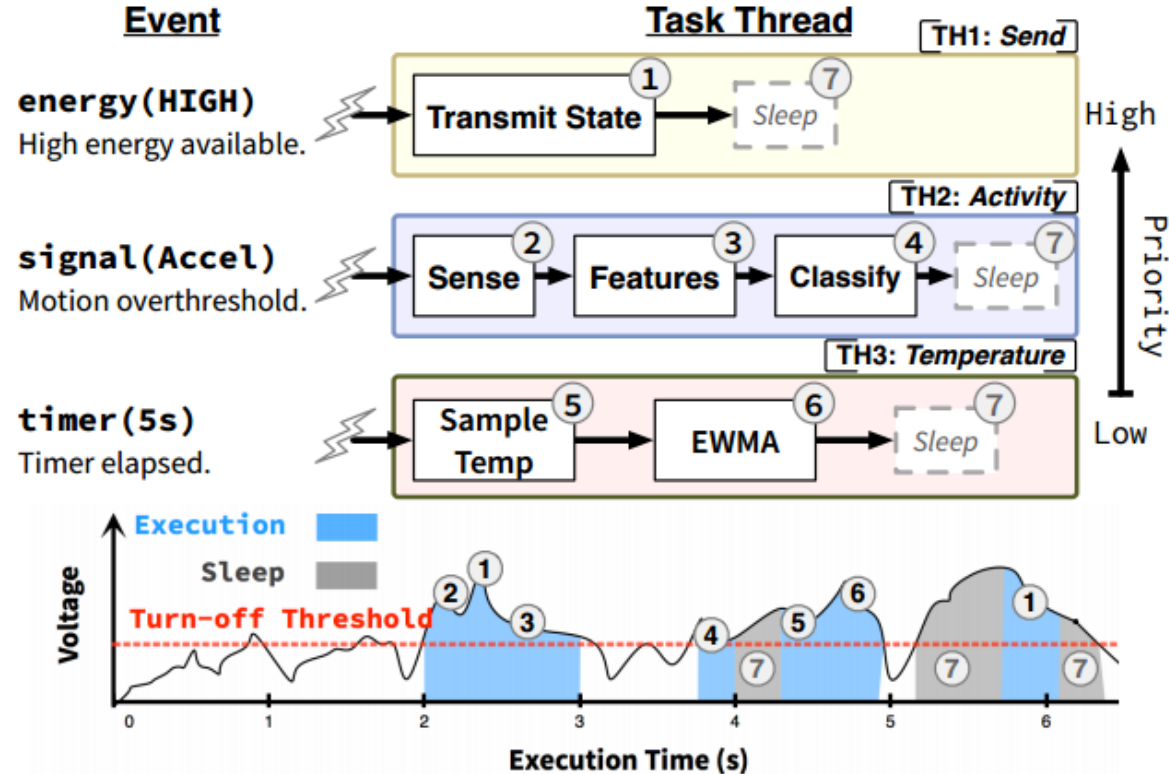
研究进展

- ❑ [IPSN16] System Design for a Synergistic, Low Power Mote/BLE Embedded Platform
 - 设计了基于32位 ARM Cortex M4的低功耗硬件平台 FireStorm, 并基于TinyOS开发了新的系统
 - 利用Lua语言的高可拓展的特性, 支持基于libC的语言的移植, 如C/C++, 支持多种通信协议如BLE, 802.15.4等。



研究进展

- ❑ [sensys18] InK: Reactive Kernel for Tiny Batteryless Sensors
 - 针对通过能量搜集（太阳能）获取能量的物联网设备，设计了操作系统内核InK。
 - InK可以感知系统能量，在能量用尽之前将必要信息保存，确保内存一致性以及程序正常恢复





浙江大学
Zhejiang University



谢谢！



Email: dongw@zju.edu.cn