传感器网络操作系统TinyOS 关键技术分析

李丽娜, 石高涛, 廖明宏

(哈尔滨工业大学 计算机科学与技术学院, 黑龙江 哈尔滨 150001)

摘 要: 在传感器网络中, 作为上层协议和应用程序的运行基础的操作系统有别于传统的嵌入式操作系统, 针对一个典型的传感器网络操作系统 TinyOS, 从其特点出发, 采用与具体的组件代码相结合的方式, 对其体系结构、运行机制和通讯机制等关键技术进行分析, 为更好地了解传感器网络操作系统和开发其应用软件提供了参考和依据.

关键词: 传感器网络; 操作系统; 运行机制; 通讯机制

中图分类号:TP316 文献标识码: A 文章编号: 1672-0946(2005)06-0724-04

Analysis of key techniques for TinyOS in sensor network

LI Li-na, SHI Gao-tao, LIAO Ming-hong

(School of Computer Science and Technology, Harbin Institute of Technology, Harbin 150001, China)

Abstract: In sensor network, sensor networks operating system(SNOS) is the basic for running the high-level protocols and applications and different from traditional embedded OS. TinyOS, a typical operating system for sensor networks, is analyzed from its key techniques, such as architecture, running mechanism and communicating mechanism, in form of combining with its characters and component codes, to provide references and gists for a good understanding of operating system and applications development in sensor networks.

Key words: sensor network; operating system; running mechanism; communicating mechanism

随着传感器技术、嵌入式计算技术和通信技术的日臻成熟, 出现了具有感知、计算和通信能力的微型传感器. 由微型传感器构成的传感器网络可以使人们在任何时间、地点和环境条件下获取大量详实而可靠的信息, 因此被广泛地应用于国防军事、环境监测、交通管理、医疗卫生、反恐抗灾等领域.

传感器网络是由大量形体较小、能源受限并且配置有计算能力和无线通信能力的传感器节点以 Ad Hoc 方式组成, 其目的是协作地感知、采集和处理网络覆盖的地理区域中感知对象的信息, 并发布给观察者[1,2]. 在传感器网络中, 传感器网络操作系统作为其必要的软件支持主要进行较复杂的任务调度与管理, 它运行在每个网络节点上, 是其他上层应用和协议运行的基础. 目前, 已经出现了一些传感器网络操作系统, 如 TinyOS[3]、MagnetOS[4] 和 Bertha[5] 等, TinyOS 是目前应用较广的操作系统. 为了更好地开发基于TinyOS 操作系统的应用软件, 有必要对 TinyOS 的关键技术进行分析. 因此, 本文主要对 TinyOS 的体系结构、程序运行机制和通讯机制进行分析.

1 TinyOS 的特点和体系结构

伯克利大学开发的 TinyOS 采用了组件的结构, 它是一个基于事件的系统. 其设计的主要目标是代码量小、耗能少、并发性高、鲁棒性好, 可以适应不同的应用. 完整的系统由一个调度器和一些组

收稿日期: 2005-07-11.

基金项目: 哈尔滨工业大学校基金(HIT 2002.74).

作者简介: 李丽娜(1978-), 女, 硕士研究生, 研究方向: 无线传感器网络; 廖明宏(1966-), 男, 博士, 博士生导师, 研究方向: 新型操作系统, 实时与嵌入式计算, 对等计算.

件组成, 应用程序与组件一起编译成系统. 组件由下到上可分为硬件抽象组件、综合硬件组件和高层软件组件, 高层组件向底层组件发出命令, 底层组件向高层组件报告事件. 调度器具有两层结构, 第一层维护着命令和事件, 它主要是在硬件中断发生时对组件的状态进行处理; 第二层维护着任务(负责各种计算), 只有当组件状态维护工作完成后, 任务才能被调度. TinyOS 的组件层次结构就如同一个网络协议栈, 底层的组件负责接收和发送最原始的数据位, 而高层的组件对这些位数据进行编码、解码, 更高层的组件则负责数据打包、路由和传输数据. TinyOS 体系结构如图 1 所示.

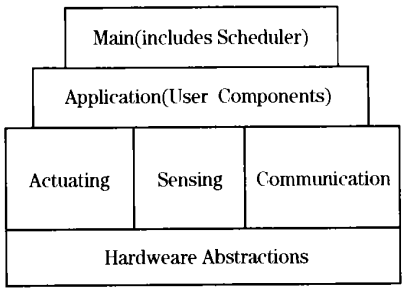

图 1　TinyOS 体系结构

2　TinyOS 程序运行机制分析

TinyOS 程序通过并口导入到节点之上, 节点一旦加电程序就会运行. 实际上在 ROM 空间的零地址上是一个跳转指令, 加电之后的程序首先运行这个跳转指令, 将控制转向程序的真正的代码处, 这之后程序才真正开始执行.

TinyOS 的所有组件都需要一个通用的主要组件即 Main 组件, 在程序的配置文件中将该组件进行绑定. Main 组件提供了一个初始化和运行 TinyOS 组件的接口 StdControl, 但是真正实现这个接口的组件应当是应用程序组件. 下面看看 Main 组件究竟完成哪些功能.

Main 组件本身的实现是另外一个组件 RealMain, 它才是真正的 Main, 我们先不关心其它的代码, 看看 RealMain 做了什么.

```
int main() __attribute __((C, spontaneous))
{ call hardwareInit();
call Pot.init(10);
TOSH __sched __init();
call StdControl.init();
call StdControl.start();
call Interrupt.enable();
while(1) {
TOSH __run __task();
}
}
```

从代码上我们看到 RealMain 完成了硬件初始化, 电位器的初始化, 调度器的初始化, 并且调用了接口 StdControl 中的函数, 这些函数完成了应用程序组件的初始化任务, 此外还要打开中断. 从这里我们也就明白了程序执行后会先进行硬件的初始化, 然后初始化组件. 至于组件的初始化如何完成则取决于用户编写的组件初始化函数, 因为 Main 组件使用的 StdControl 接口需要在用户编写的组件里绑定到其初始化函数中. 最后 RealMain 组件开始运行队列中的任务.

TinyOS 的调度队列是个先进先出的循环队列, 整个队列中至少有一个空闲位置, 应用程序可以调用 Post __Task 将一个新的任务加到队列中. 当没有中断发生时, 调度器会从队列中取出一个任务执行, 任务执行过程中可以被打断. 如果队列中没有了任务, 处理器会进入睡眠状态, 等待被其它事件激活.

3　TinyOS 中的通讯机制分析

TinyOS 中的通讯遵循 AM 通讯模型, 消息中包含有消息类型及消息处理函数, 在消息到达目的地址后, 该消息处理函数被调用. 在 TinyOS 中, 组件是构造程序的基本单元, 通讯功能的实现也是通过由低到高的组件之间的通讯实现的. 大致看来, 在 TinyOS 的通讯经历的组件流程如图 2 所示:

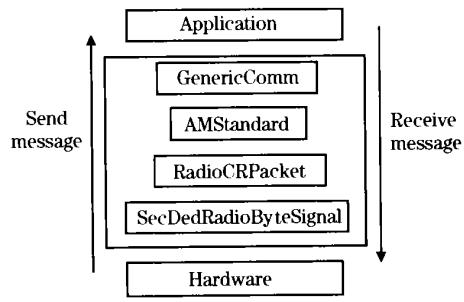

图 2　通讯组件流程图

应用程序直接使用的组件是 GenericComm, 数据将依次经过 AMStandard, RadioCRCPacket, 和 SecDedRadioByteSignal等组件的处理与编码之后通过硬件发送出去. 接受的过程恰好与之相反.

应用程序组件要发送数据需要引用系统组件 GenericComm, GenericComm组件是 TinyOS 的最基本

的网络通讯栈,它是一个配置(configuration)文件,可以在 tos/system/GenericComm.nc 中找到它的绑定实现.该组件提供的接口中有两个最重要的接口即 SendMsg[uint8 __t id]和 ReceiveMsg[uint8 __t id],分别供用户调用来发送和接受消息,并且使用了很多底层的接口来实现通讯.从这个文件中我们可以看到真正实现这些接口的组件由 AMStandard 来完成 Active Message 的发送和接收、UARTNoCRCPacket 来实现了通过串口进行通讯、RadioCRCPacket 来实现了通过无线进行通讯等等.接口 SendMsg 和 ReceiveMsg 都是参数化接口,参数 id 就是前面说的 handler id.接口 SendMsg 中包含 commands send 和 events senddone.通过语句 SendMsg = AMStarrdard.SendMsg; ReceiveMsg = AMStandard.ReceiveMsg;来说明它们的真正实现是由组件 AMStarrdard 中相应接口来完成的.

3.1 消息发送

整个消息的发送涉及组件、接口和事件三方面,当上层组件有消息发送时通过接口调用下层组件来实现,下层组件完成发送后也通过接口向上层组件回送消息.

3.1.1 组件部分

1) GenericComm

在 GenericComm 组件中,用户要发送消息,需要调用 GenericComm 提供的接口 SendMsg,SendMsg 包含以下内容:

```
command result __t send(uint16 __t address, uint8 __t length, TOS __MsgPtr msg);
event result __t sendDone(TOS __MsgPtr msg, result __t success);
```

应用程序调用 SendMsg.Send 来发送消息,同时需要事件 event sendDone 来接受下面的组件 signal 的事件,这样,当消息发送出去后,应用程序就会接到来自下面一层组件(也就是 AMStandard)触发的事件.注意,在 TinyOS 中,基本上涉及到发送消息的接口都是分段操作,发送动作完成后都要伴随着一个事件的产生.在组件 GenericComm 中,接口 SendMsg 是通过语句 SendMsg = AMStandard.SendMsg 被绑定到 AMStandard 组件上实现的.

2) AMStandard

在 AMStandard 组件中,根据消息的目的地址做不同的处理.如果消息地址是 TOS __UART __ADDR,表明消息将要发送到计算机的串口,这个时候将要调用 UARTSend.send 来发送数据.否则就调用 RadioSend.send 来发送数据,在组件 AMStandard 中,RadioSend 就是接口 BareSendMsg.

3) CRCPacket

组件 CRCPacket 完成数据包的 CRC 校验和完整性检查.接口 BareSendMsg 中的 Send 命令的实现就是该组件中的 Send 命令.该组件会再次调用 ByteComm.txByte 来完成进一步的处理和发送任务.同时,该组件也实现和 ByteComm 接口中的几个事件.

4) SecDedRadioByteSignal

组件 SecDedRadioByteSignal 实现了接口 ByteComm,也即实现了 ByteComm.txByte.该组件会调用接口 Radio,由它负责传输位数据.当传输完毕后会向网络栈的上一层组件 CRCPacket 发出信号

```
signal ByteComm.txDone();
signal ByteComm.txByteReady(SUCCESS)
```

3.1.2 接口部分

1) BareSendMsg

BareSendMsg 是发送原始数据包的接口,包含 Commands Send 和 events SendDone,这样,当实现该接口的组件完成 Send 操作之后,就会用 Signal BareSendMsg.sendDone 来通知上一层的组件 Send 已完成.实现该接口的组件有 CRCPacket、NoCRCPacket 和 MicaHighSpeedRadioM.根据不同的应用可以将接口绑定到这三个组件中的任意一个.在无线通讯中,组件 RadioCRCPacket 绑定了 CRCPacket,组件 RadioNoCRCPacket 绑定了 NoCRCPacket,串口通讯组件 UARTNoCRCPacket 则绑定了 NoCRCPacket.在 GenericComm 组件中则是绑定了 RadioCRCPacket,

2) ByteComm

ByteComm 是一个字节级的接口.它会发信号通知收到一个字节等待发送,并且提供了一个分段的字节发送接口.该接口的定义如下:

```
interface ByteComm {
/* Transmits a byte over the radio* /
command result __t txByte(uint8 __t data);
/* Notification that the radio is ready to receive another byte * /
event result __t rxByteReady(uint8 __t data, bool error, uint16 __t strength);
/* Notification that the bus is ready to transmit/queue another byte* /
event result __t txByteReady(bool success);
/* Notification that the transmission has been completed and the transmit queue has been emptied.* /
```

```
event result __t txDone( );
}
```

txByteReady 说明组件可以接收另外一个字节到它的队列中等待发送. txDone 说明整个队列已经全部发送完. 实现该接口的组件是 SecDedRadioByteSignal 组件.

3) Radio

Radio 是一个节点无线通讯的位级接口. Radio 有两个状态即传输态和接收态, 并且可以被设置. 采样和终端频率可以设置成以下之一: 0 (double sampling), 1 (one- and- a= half- sampling) and 2 (single sampling). 该接口直接对硬件抽象, 一些高层组件可以理解的状况在这里可能执行错误, 比如在接收状态下不能调用 txBit. 一个高层接口必须提供这种状况的检查. Radio 接口的定义如下:

```
interface Radio {
/* Start transmitting this bit. Does nothing if in receive mode. */
command result __t txBit(uint8 __t data);
/* Transition into transmit mode. */
command result __t txMode( );
/* Transition into receive mode. */
command result __t rxMode( );
/* Set bit rate to 0 (20Khz), 1 (13 KHz) or 2 (10 KHz). */
command result __t setBitRate( char level);
/* Notification that a bit has been transmitted; */
event result __t txBitDone( );
/* Notification that a bit has been received. */
event result __t rxBit(uint8 __t bit);
}
```

真正实现该接口的组件是 RFM, 并且它向使用 Radio 接口的组件发送事件消息, 如 signal Radio. txBitDone( ); signal Radio. rxBit( ).

(1) RFM

RFM 提供了 Radio 接口, 但是真正实现 txBit 命令的是组件 HPLRFM.

(2) HPLRFMC

这个组件是硬件表示层, 它控制硬件操作进行数据发送.

3.1.3　事件部分

在前面我们多次提到数据发送的动作总要伴随着一个完成事件的触发. 下面我们来看一下在整个数据包发送过程中要触发的事件序列. 这一序列正好和发送数据的方向相反.

组件 HPLRFM 会触发事件 RFM. bitEvent, 实现该事件的组件是 RFM, 这个事件会再次触发事件 Radio. txBitDone, 通知它一位数据已经传输出去. Radio. txBitDone 事件的处理是由组件 SecDedRadioByteSignal 来完成的, 该组件调用了接口 Radio, 这个事件的处理函数会发送数据的下一位给 Radio, 同时, 如果队列中只有一个字节, 处理函数会触发事件 ByteComm. txByteReady 来发送数据并通知可以接收新的字节数据; 如果没有字节缓冲, 它会将转入 Idle 状态并触发事件 ByteComm. txDone 和事件 ByteComm. txByteReady. 当上层组件也即 CRCPacket 接收到事件后表明它可以接收新的字节, 检查是否有数据并发送. 此外, 它也要向上层组件 AMStandard 通知事件即 BareSendMsg. sendDone. 这个事件会继续向上产生事件, 也就是告诉应用程序数据包已经发送完毕. 这个时候用户应用程序调用的 SendMsg 中的 SendDone 事件就会触发, 这个事件的处理则有用户来决定了. 至此, 整个数据包的发送完成.

3.2　消息接收

消息的接收主要是以事件的逐层向上传递来进行的. 当硬件接到一个消息时它会发生中断, 在中断的处理程序中触发事件 RFM. bitEvent, 在该事件的处理函数中又触发了事件 Radio. rxBit, 这个事件的处理函数接收 Radio 的采样数据并试图找到开始标志, 一旦找到了消息的开始标志, 它就会 Post 一个任务将接收到的编码数据进行解码. 数据解码之后该事件发信号给 ByteComm. rxByteReady 事件, 表示可以接收下一字节数据. ByteComm. rxByteReady 事件处理字阶级组件传递的解码数据, 主要是通过 Post 一个任务进行 CRC 检查. 这个任务会发信号给 Receive. receive 事件, Receive. receive 只是简单地将其接到的消息返回. 整个过程涉及到的组件之间的关系如图 3 所示.

4　结　语

传感器网络是一种新的信息获取和处理技术, 传感器网络操作系统是其重要的组成部分. 本文介绍了一种典型传感器网络操作系统 TinyOS 的体系结构, 并对其程序运行机制和通讯机制进行分析. TinyOS 采用组件方式, 结构简单, 任务以先进先出方式进行调度, 数据通讯以事件方式在组件之间进行传递. 该机制非常适合资源受限的传感器网络.

(下转 746 页)

用完后要立刻关闭, 节省系统资源.

4) 在存储过程中使用到事务时, 应当将频繁操作的多个可分割的处理过程放入到多个存储过程中, 这样可以提高系统的相应速度. 但前提是不违背数据的一致性.

8 结 语

由于存储过程保存在数据库中, 当用户调用数据库的存储过程时, 在服务器端执行调用的存储过程, 再把执行结果返回给调用者, 因此使用存储过程有助于提高代码执行效率, 也有助于提高网络使用效率, 同时, 在存储过程中能够把数据经过处理后再返回, 这样可以对数据提供更多的分析和控制. 在开发过程中, 使用存储过程是一个良好的编程习惯, 而存储过程的参数设置和调用方法是使用存储过程的必要手段[6].

参考文献:

[1] 李陶深, 罗翌源. 用存储过程提高应用程序执行效率[J]. 微型电脑应用, 2002(2): 24- 28.

[2] 孙一林. Java 数据库编程实例[M]. 北京: 清华大学出版社, 2003. 168- 195.

[3] JON E, LINDA H. JDBCTM 3. 0 Specification[J]. Sun Microsystemrs Inc, 2001 (3): 35- 68.

[4] 程 凡. 存储过程和触发器在银行报表管理系统中的应用[J]. 合肥工大学报, 2004(27): 278- 281.

[5] 张小波, 成良玉. vs. net 中存储过程使用方法研究[J]. 计算机应用, 2004 (24): 139- 140.

[6] 欧阳江林. JDBC 和 Oracle 的参数设置和调用技术[J]. 浙江工贸职业技术学院学报, 2004 (4): 32- 36.

(上接 727 页)

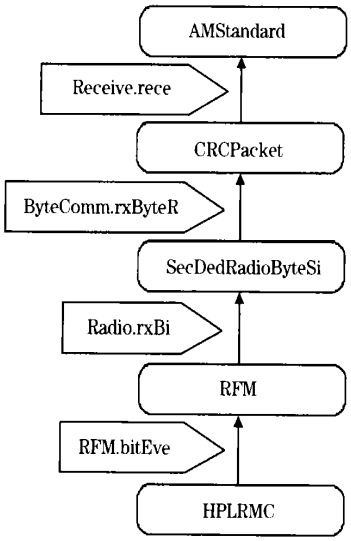

图 3 组件之间的事件传递图

参考文献:

[1] ESTRIN D, GOVINDAN R, HEIDEMANN J, *et al*. Next century challenges: Scalable coordination in sensor networks[A]. Proc. ACM/IEEE International Conference on Mobile Computing and Networks [C]. Seattle, Washington, USA: ACM, 1999. 263- 270.

[2] AKYILDIZ I F, SU W, SANKARASUBRAMANIAM Y, *et al*. Wireless sensor networks: a survey[J]. Computer Networks, 2002, 38 (4): 395- 398.

[3] HILL J, SZEWCYK R, WOO A, *et al*. System Architecture Directions for Networked Sensors[A]. Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems[C]. [s. l]: [s. n.] 2000. 93- 104.

[4] BARR R, BICKET J C, DANTAC D S, *et al*. On the Need for System - Level Support for Ad hoc and Sensor Networks[A]. Operating Systems Review[C]. [s. l.]: ACM, 200. 21- 5.

[5] LIFTON J, SEETHARAM D, BROXTON M, *et al*. Pushpin Computing System Overview: A Platform for Distributed, Embedded, Ubiquitous Sensor Networks[A]. Proceedings of the First International Conference on Pervasive Computing[C]. [s. l]: [s. n.], 2002. 139 - 151.