



信息科学与技术丛书

TinyOS 实用编程

——面向无线传感网节点软件开发

李 鸥 张效义 王晓梅 张大龙 编著



- 结合长期的TinyOS系统**开发经验**，根据读者的不同需求提供不同层次的知识
- 通过引用研发中的**应用实例**予以辅助说明，具有较强的实用性和指导性
- 结合具体应用场景，给出了一个完整的应用实例



机械工业出版社
CHINA MACHINE PRESS

信息科学与技术丛书

TinyOS 实用编程

——面向无线传感网节点软件开发

李 鸥 张效义 王晓梅 张大龙 编著



机械工业出版社

本书结合 TinyOS 2.x 系统具体的工程应用, 面向程序开发的需求, 介绍了利用 TinyOS 开发无线传感器网络应用系统应具备的基础知识, 包括 TinyOS 系统的特点、体系结构、安装与常用命令、简单实例等; 详细介绍了 TinyOS 的编程语言 nesC (包括组件、接口、模块、配件与连接、参数化接口、通用组件等), TinyOS 系统并发执行模型, 驱动程序与硬件抽象, 系统主要功能模块, TinyOS 典型应用; 剖析了应用程序运行过程, 对应用程序的仿真、调试和编程提示进行了描述; 最后结合应用实例进行系统分析以提高读者对于 TinyOS 的系统认识。

各章节引用具体的应用实例予以辅助说明, 具有较强的实用性和指导性。

本书既可作为无线传感器网络开发人员的技术参考书, 也可作为高等学校嵌入式操作系统和通信类专业参考教材。

图书在版编目 (CIP) 数据

TinyOS 实用编程: 面向无线传感网节点软件开发 / 李鸥等编著. —北京: 机械工业出版社, 2012.11

(信息科学与技术丛书)

ISBN 978-7-111-40722-5

I. ①T… II. ①李… III. ①无线电通信—传感器—网络操作系统—程序设计 IV. ①TP212②TP316.8

中国版本图书馆 CIP 数据核字 (2012) 第 293307 号

机械工业出版社 (北京市百万庄大街 22 号 邮政编码 100037)

策划编辑: 郝建伟

责任编辑: 郝建伟 王 凯

责任印制: 杨 曦

北京中兴印刷有限公司印刷

2013 年 7 月·第 1 版第 1 次印刷

184mm×260mm·12.75 印张·314 千字

0001—3000 册

标准书号: ISBN 978-7-111-40722-5

定价: 36.00 元

购本书, 如有缺页、倒页、脱页, 由本社发行部调换

电话服务

网络服务

社服务中心: (010) 88361066

教材网: <http://www.cmpedu.com>

销售一部: (010) 68326294

机工官网: <http://www.cmpbook.com>

销售二部: (010) 88379649

机工官博: <http://weibo.com/cmp1952>

读者购书热线: (010) 88379203

封面无防伪标均为盗版

出版说明

随着信息科学技术的迅速发展，人类每时每刻都会面对层出不穷的新技术和新概念。毫无疑问，在节奏越来越快的工作和生活中，人们需要通过阅读和学习大量信息丰富、具备实践指导意义的图书来获取新知识和新技能，从而不断提高自身素质，紧跟信息化时代发展的步伐。

众所周知，在计算机硬件方面，高性价比的解决方案和新型技术的应用一直备受青睐；在软件技术方面，随着计算机软件的规模和复杂性与日俱增，软件技术不断地受到挑战，人们一直在为寻求更先进的软件技术而奋斗不止。目前，计算机在社会生活中日益普及，随着 **Internet** 延伸到人类世界的方方面面，掌握计算机网络技术和理论已成为大众的文化需求。由于信息科学与技术 在电工、电子、通信、工业控制、智能建筑、工业产品设计与制造等专业领域中已经得到充分、广泛的应用，所以这些专业领域中的研究人员和工程技术人员越来越迫切需要汲取自身领域信息化所带来的新理念和新方法。

针对人们对了解和掌握新知识、新技能的热切期待以及由此促成的人们对语言简洁、内容充实、融合实践经验的图书迫切需要的现状，机械工业出版社适时推出了“信息科学与技术丛书”。这套丛书涉及计算机软件、硬件、网络和工程应用等内容，注重理论与实践的结合，内容实用、层次分明、语言流畅，是信息科学与技术领域专业人员不可或缺的参考书。

目前，信息科学与技术的发展可谓一日千里，机械工业出版社欢迎从事信息技术方面工作的科研人员、工程技术人员积极参与我们的工作，为推进我国的信息化建设做出贡献。

机械工业出版社

前 言

无线传感器网络（Wireless Sensor Networks, WSN）作为人类社会和物理世界沟通的桥梁和纽带，近年来取得了飞速发展。TinyOS 是目前国内外使用最为广泛的无线传感器网络嵌入式操作系统之一。了解并熟练使用 TinyOS 是开发人员进行无线传感器网络软件开发的前提和基础。本书主要介绍 TinyOS 2.x，帮助学生和工程开发人员快速熟悉 TinyOS 操作系统，熟练掌握基于 TinyOS 的开发方法。

本书从 TinyOS 系统的特点、安装和常用命令等基础知识入手，由浅入深，逐步介绍 TinyOS 的编程语言 nesC、系统并发模型、硬件抽象，一直到应用程序的仿真、调试以及编程提示，可帮助读者建立对 TinyOS 系统的整体概念，快速掌握 TinyOS 应用程度的开发技巧。书中各章节通过引用研发中的应用实例予以辅助说明，具有较强的实用性和指导性。

本书作者结合 TinyOS 系统长期开发经验，根据读者的不同需求提供不同层次的知识。对于初学者，本书提供简单实用的入门知识，方便其快速上手；对于有一定 TinyOS 使用经验的读者，则通过对 TinyOS 进行提炼、切块、总结和细化，提供了宏观性的整体介绍和更为深入的细节介绍，有助于读者更深入地理解和运用 TinyOS。本书既可作为高等学校嵌入式操作系统和通信类专业参考教材，也可作为无线传感器网络软件开发人员的技术手册。

本书第 10 章结合煤矿井下定位系统的具体应用场景，给出了一个完整的应用实例，以帮助读者顺利地完成任务。

本书由李鸥、张效义、王晓梅、张大龙编写，参加本书程序调试及辅助工作的人员还有刘广怡、鲁旭阳、张瑞琴、宋涛、孙武剑等。本书的顺利出版，要感谢解放军信息工程大学的领导和老师给予的大力支持和帮助。

由于时间仓促，书中难免存在不妥之处，请读者批评指正，并提出宝贵意见。

编 者

目 录

出版说明

序

前言

第 1 章 无线传感网与 TinyOS 1

1.1 无线传感网 1

1.1.1 无线传感网的提出与发展 2

1.1.2 无线传感网的特点与优势 4

1.1.3 无线传感网的应用 7

1.2 无线传感网节点软件技术 8

1.3 TinyOS 10

1.3.1 TinyOS 的特点 11

1.3.2 TinyOS 体系结构 12

第 2 章 TinyOS 安装与常用命令 13

2.1 TinyOS 安装 13

2.1.1 在 Windows 下安装 TinyOS 13

2.1.2 在 Linux 下安装 TinyOS 19

2.2 目录结构 21

2.2.1 Cygwin 的目录结构 21

2.2.2 TinyOS 的目录结构 21

2.3 常用命令与快捷键 22

2.3.1 常用命令 22

2.3.2 快捷键 24

2.4 一个简单的应用分析 25

2.4.1 分析 Powerup 应用程序 25

2.4.2 Powerup 应用程序的图形化
说明 26

2.4.3 编译 Powerup 应用程序 26

2.4.4 Powerup 应用程序中 nesC 到 C 的
映射 27

2.4.5 仿真 Powerup 应用程序 27

2.4.6 下载应用程序 28

第 3 章 TinyOS 编程语言 nesC 30

3.1 C 与 nesC 的比较 30

3.1.1 C 与 C++ 30

3.1.2 nesC 32

3.2 组件定义 36

3.2.1 组件规范 36

3.2.2 关键字 as 37

3.2.3 将接口分成组 38

3.3 接口 39

3.3.1 一般接口 39

3.3.2 分阶段操作 40

3.3.3 通用接口 41

3.4 模块 42

3.4.1 接口函数 42

3.4.2 任务 44

3.4.3 模块数据 46

3.5 配件与连接 50

3.5.1 配件实现 50

3.5.2 多连接 57

3.6 参数化接口 59

3.6.1 为什么需要参数化接口 60

3.6.2 基于参数化接口的两个典型
例子 61

3.6.3 基于参数化接口的连接 62

3.6.4 参数化接口的实现 64

3.6.5 unique()和 uniqueCount()
函数 65

3.6.6 Default 函数 66

3.7 通用化组件 66

3.7.1 为什么需要通用组件 67

3.7.2 通用组件的定义与实例化 67

3.7.3 通用化模块 69

3.7.4 通用化配件 71

3.7.5 属性 attributes 73

第 4 章 并发执行模型	75	6.1.2 资源的仲裁	105
4.1 任务	75	6.1.3 跨组件预约	109
4.1.1 任务的执行	77	6.1.4 执行	110
4.1.2 执行和事件处理	78	6.2 微控制器能量管理	112
4.2 任务和分阶段调用	78	6.2.1 TinyOS 1.x 中的微控制器能量 管理	112
4.2.1 硬件与软件	78	6.2.2 TinyOS 2.x 中微控制器能量 管理	113
4.2.2 任务和调用循环	78	6.2.3 外围设备和子系统	115
4.3 异步代码	80	6.3 外围设备能量管理	115
4.3.1 关键字 <code>async</code>	80	6.3.1 能量管理模型	115
4.3.2 异步的代价	81	6.3.2 显式能量管理	116
4.3.3 原子语句和关键词 <code>atomic</code>	82	6.3.3 隐式能量管理	118
4.3.4 状态切换的管理	84	第 7 章 TinyOS 典型应用	122
4.3.5 CC2420ControlP 组件的工作	85	7.1 传感	122
4.3.6 任务的再次调用	86	7.1.1 传感简介	122
4.4 功率锁	86	7.1.2 Sense 实例	122
4.4.1 功率锁需求实例——链路层 确认	87	7.1.3 Oscilloscope 实例	125
4.4.2 分阶段锁	87	7.2 存储	127
4.4.3 锁的内部结构	88	7.2.1 存储简介	127
4.4.4 能量管理	88	7.2.2 配置数据的存储	128
4.4.5 硬件配置	89	7.2.3 日志数据的存储	133
4.4.6 MSP430 USART 配置	89	7.2.4 大数据块的存储	135
4.4.7 功率锁库	90	7.3 节点与 PC 的通信	135
第 5 章 TinyOS 驱动程序与平台硬件 抽象	91	7.3.1 串口通信堆栈	136
5.1 硬件抽象结构简介	91	7.3.2 MIG	140
5.1.1 什么是硬件抽象	91	7.3.3 数据包源	142
5.1.2 硬件表示层	92	7.3.4 串口通信测试	143
5.1.3 硬件适配层	93	7.4 节点之间通信	144
5.1.4 硬件接口层	93	7.4.1 AM 通信	144
5.1.5 不同层次抽象的结合	94	7.4.2 单跳发送数据包	150
5.1.6 横向分解	94	7.5 定时器 (Timer)	154
5.2 TinyOS 平台	96	7.5.1 接口	154
5.2.1 TinyOS 平台简介	96	7.5.2 定时器 HAL 方案	158
5.2.2 新建 TinyOS 平台	97	7.5.3 定时器 HIL 需求	158
第 6 章 TinyOS 其他主要功能模块 分析	103	7.5.4 用到的其他组件	159
6.1 资源仲裁	103	7.5.5 实现	161
6.1.1 资源的分类	104	第 8 章 TinyOS 应用程序运行过程 剖析	163

8.1 系统初始化	163	9.1.3 在 Python 下运行 TOSSIM	177
8.1.1 任务调度器初始化	165	9.2 GDB/ddd 调试	179
8.1.2 组件初始化	167	9.3 编程提示	182
8.1.3 触发 Boot 接口中的事件	168	9.3.1 Packages 包	182
8.2 Blink 执行过程	169	9.3.2 语法约定	182
8.2.1 配置文件	169	第 10 章 基于 WSN 的煤矿井下定位	
8.2.2 模块文件	172	系统	185
8.2.3 Blink 应用程序的图形化		10.1 功能需求	185
说明	173	10.2 系统设计	186
8.2.4 程序运行过程	173	10.3 硬件组成	187
8.3 Blink 应用程序中 nesC 到 C 的		10.4 TinyOS 实现	187
映射	174	10.4.1 软件结构框架	187
8.4 仿真 Blink 应用程序	174	10.4.2 模块介绍	188
第 9 章 仿真、调试与编程提示	176	10.4.3 主要接口与连通情况介绍	189
9.1 TinyOS 自带仿真工具——		10.4.4 节点运行流程	190
TOSSIM	176	附录 缩略语	193
9.1.1 概述	176	参考文献	194
9.1.2 编译 TOSSIM	177		

第1章 无线传感网与 TinyOS

现代微电子技术、无线通信技术、计算机技术和信号处理技术的进步以及互联网的迅猛发展极大地推动了低功耗多功能传感器的快速发展。无线传感器网络（Wireless Sensor Network, WSN）作为信息获取技术新的发展方向的代表应运而生。无线传感器网络的诸多特点使其应用范围非常广泛，涉及军事应用、工业监视与控制、环境监测、医疗监护、智能家居或建筑、交通控制管理等诸多领域，具有广阔的市场和产业前景。美国《商业周刊》（Business Week）预测：WSN 和其他三项信息技术将会在不远的将来掀起新的产业浪潮；美国橡树岭国家实验室（Oak Ridge National Laboratory, ORNL）提出了“网络就是传感器”（Network is Sensor）的论断；美国《今日防务》杂志甚至认为 WSN 的应用和发展，将引起一场划时代的军事技术革命和未来战争的变革。在国内，各大科研院所也纷纷展开了该领域的研究工作，一份有关我国未来 20 年预见技术的调查报告显示，信息领域的 157 项技术课题中有 7 项与传感器网络直接相关，由此可见人们对 WSN 的重视程度。

无线传感器网络节点是一个资源受限的嵌入式系统，其自身的一些特点决定了现有的一些嵌入式操作系统都不能很好地适用于该网络的节点，因此它需要拥有适应其自身特点的专属操作系统。近年来，无线传感器网络作为国内外研究的热点领域，已经吸引了大量优秀的科研团队参与研究，许多国内外的知名大学和研究机构都纷纷开发出了具有自主知识产权的 WSN 操作系统，比如：美国加州大学伯克利分校开发的 TinyOS、首尔大学开发的 SenOS 以及我国中科院计算机研究所开发的 GOS 操作系统等。本书以 TinyOS 操作系统为研究对象，重点介绍了 TinyOS 编程语言的特点；深入分析了该系统中各个模块的功能；详细剖析了 TinyOS 应用程序的运行流程；最后，给出一个具体的开发实例，使读者对该操作平台的运行规律有一个更加清晰的认识。

1.1 无线传感网

无线传感器网络是由部署在监测区域内的大量低成本、低功耗的传感器节点组成，这些传感器节点协作地感知、采集和处理网络覆盖区域中被感知对象的信息，并通过一个多跳的、自组织的无线网络将信息传输到汇聚节点，然后再利用 Internet、卫星等通信方式将信息发送给用户。WSN 比较典型的工作方式：将大量传感器节点抛撒到感兴趣区域内，节点之间通过自组织的形式快速形成一个无线网络。随机分布的集成有传感器、数据处理单元和通信模块的微小节点借助内置的传感器测量周围环境中的热、红外、雷达和地震波等信号，从而探测包括温度、湿度、噪声、光强度、压力、土壤成分、移动物体的大小、速度和方向等众多部署者感兴趣的物理参数。在无线传感器网络中，节点既是信息的采集和发出者，也充当信息的路由，采集的数据通过多跳路由到达网关。网关是其中一个特殊的节点，可以通

过互联网、移动通信网络、卫星等与监控中心进行通信，也可利用无人机飞越网络上空，通过网关节点来采集数据。

1.1.1 无线传感网的提出与发展

1. 无线传感网的提出

无线传感器网络的研究起源于 20 世纪 70 年代，最早应用于军事领域。1978 年，美国国防部高级项目研究署（Defense Advanced Research Projects Agency, DARPA）的分布式传感器网络（Distributed Sensor Networks, DSN）项目开启了现代传感器网络研究的先河，如图 1-1 所示。

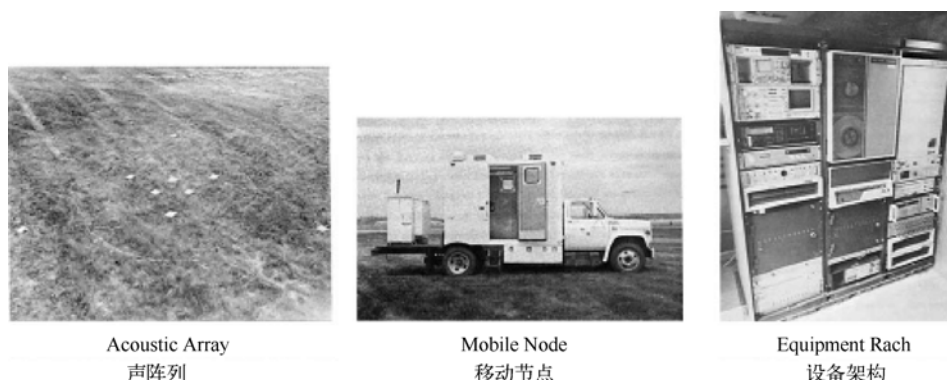


图 1-1 分布式目标跟踪试验床构成

通过卡耐基梅隆大学、麻省理工学院等大学研究人员的努力，DSN 项目在信号处理、节点实验平台等很多方面取得了一定的进展。20 世纪 80~90 年代，WSN 的研究依然主要集中在军事领域，并成为了网络中心战中的关键技术，其中比较著名的系统包括：美国海军研制的协同交战能力系统（Cooperative Engagement Capability, CEC）、远程战场传感器网络系统（Remote Battlefield Sensor System, REMBASS）和战术远程传感器系统（Tactical Remote Sensor System, TRSS）等。1994 年加州大学洛杉矶分校的 William J. Kaiser 教授向 DARPA 提交了研究建议书“Low Power Wireless Integrated Microsensors”，这对推动 WSN 的研究具有里程碑式的意义。1998 年 G. J. Pottie 从网络研究的角度重新阐释了 WSN 的科学意义。由于 WSN 巨大的应用价值，无线传感器网络的学术研究在世界范围内逐渐展开，其中最为著名的项目有：MIT 的 uAMPS 和 NMS 项目；UC Berkeley 的 PicoRadio 和 Smart Dust 项目；哈佛大学的 Code Blue 项目；南加州大学的 SCADDS 项目等。美国自然科学基金委员会分别于 2003 年和 2004 年设立并推出了“嵌入式智能传感器研究中心”（Center Of Embedded Networked Sensing, CENS）和“网络技术与系统”（Networking Technology And Systems, NeTS）研究项目，取得了一系列杰出的研究成果。

在我国，国家自然科学基金委员会、科技部、工业和信息化部等对无线传感器网络的研究极为重视，国家自然科学基金委员会于 2003 年开始对 WSN 的研究进行资助，并于 2004 年将其列为重点科研项目。国务院在制定《国家中长期科学和技术发展规划纲要（2006-2020 年）》时将“传感器网络及智能信息处理”列为未来信息产业及现代服务业的重点方向

之一。在国家的大力支持下,包括清华大学、北京邮电大学、中科院计算机研究所、沈阳自动化所等国内多所高校和科研机构积极地展开了对 WSN 各方面的研究工作。

2. 无线传感网的研究进展

无线传感网络的研究起于美国军方。20 世纪 90 年代末期,在美国军方资助下,现代意义上的无线传感网络得以产生并逐步发展。从 2000 年起,国际上开始出现一些有关无线传感网络研究成果的报道。由于无线传感网络的巨大应用价值,许多国家的军事界、工业界和学术界等业界已经开始对其广泛关注,并相继启动了许多无线传感网络的研究计划。

根据研究侧重点的不同,可以将无线传感网络技术从产生到现在大致划分为两个阶段。第一个阶段主要致力于小型化、低功耗、传感器节点的开发和研制,出现了大量的传感节点平台和初级应用示范系统,其中 Motes 硬件平台及其配套的 TinyOS 操作系统应用最为广泛,已为全球 400 多家研究机构所采用。第二个阶段主要致力于开展大量针对无线传感网络通信协议及其支撑技术的深入研究工作,如 MAC 协议、路由协议、传输层协议、定位技术、时间同步等。同时,美国各大高校和研究机构及其他各国的相关研究机构也都开始了对无线传感网络中的信息处理、数据查询等问题的细化研究。

(1) 军事领域

1978 年美国国防部高等计划研究署在宾夕法尼亚州匹兹堡的卡耐基梅隆大学主办的分布式传感器网络研讨会上,提出对传感器网络展开研究。各军事部门和研究机构从 20 世纪 90 年代开始了对无线传感网络的研究,并有针对性地设立了一系列军事传感器网络研究项目。1998 年由包括加州大学伯克利分校、麻省理工、哈佛在内的 25 个研究机构共同承担致力于研究大规模分布式军事传感系统的无线专用网络的 SensIT (Sensor Information Technology) 项目。1999 年开始并于 2001 年完成的 Smart Dust 项目,研制出了体积不超过 1 立方毫米、使用太阳能电池供电、具有光通信能力、可悬浮于空中的自治传感器节点。该节点可用于隐蔽式战场监视,将重要的情报秘密发送给中央指挥控制中心。

2001 年美国陆军提出了“灵巧传感器网络通信”计划,很大程度上提高了对战场态势的感知能力。2002 年 5 月,美国 Sandia 国家实验室与美国能源部合作,共同研究防范生化武器袭击的系统,这属于美国能源部恐怖对策项目的重要环节。2005 年美军试验成功的“狼群”地面无线传感器网络标志着电子战领域的最新技术突破。美国海军最近确立了“传感器组网系统”研究项目,该系统是一套实时数据库管理系统,能够部署到各级指挥单位。美国空军展开了对 WSN 相关方面的研究,旨在为部队官兵提供战场威胁的实时信息、增强其战场状态感知能力。

(2) 工业领域

在工业领域,1995 年美国交通部提出了“国家智能交通系统项目规划”,试图有效集成先进的信息技术、数据通信技术、传感技术、控制技术 & 计算机处理技术,建立大范围全方位实时高效的综合交通运输管理系统,并预计到 2025 年全面投入使用。

作为信息业巨头,Intel 和 Microsoft 也开始了对 WSN 应用方面的研究。Intel 公司于 2002 年 10 月 24 日发布了“基于微型传感器网络的新型计算发展规划”,致力于微型传感器网络在预防医学、环境监测、行星探测等领域的应用,预计用三个阶段来实现该计划,即物理阶段、实现阶段和应用阶段。物理阶段主要用于开发集成感知、计算和通信功能的超微型

传感器；实现阶段将在实际商务中使用来自传感网的感知数据；应用阶段将无线传感网络用于预防医学、环境监测及灾害对策等领域。

2002~2005 年的欧洲研究计划——EYES (Energy efficient sensor networks) 项目中，包括来自荷兰、法国、意大利和德国的研究机构，主要研究了能量有效的自组织协同传感网络的问题，解决了无线分布式信息采集、处理和传输的问题，提出了体系结构和相关技术，使新一代的传感器节点能够有效组网，并为大量异构的移动传感网络提供平台。2005 年 Microsoft 在印度设立了传感器研发中心，旨在地理信息系统、多语言系统和传感器网络领域开展研究，为新兴的市场提供技术。

(3) 学术领域

在学术领域，2003 年美国自然科学基金委员会 (NSF) 制定了无线传感器网络研究计划，并一期资助 4000 万美元在加州大学洛杉矶分校成立了传感器网络研究中心 (CESN)，联合周边大学开展“嵌入式智能网络传感器”研究项目。在 NSF 的推动和帮助下，加州大学伯克利分校、麻省理工、波士顿大学等研究机构率先开展了无线传感网络的关键技术和基础理论的研究工作，以求利用传感器网络对我们生活的物理世界实现全方位的测试与控制，支持相关基础理论的研究。同时，这也是美国国情咨文中有关 Internet2 最主要的远景规划之一。几乎美国所有著名院校都有研究小组在从事传感器网络相关技术的研究，德国、英国、芬兰和意大利等国家的研究机构也开始了传感器网络的研究工作。

我国的无线传感网络及其应用研究起于 1999 年中国科学院《知识创新工程试点领域方向研究》的信息与自动化领域研究报告，并成为该领域提出的五个重大项目之一，是我国科技领域内少数位于世界前列的方向之一。中国科学院上海微系统研究所、软件研究所、电子所和自动化所等科研机构，清华大学、上海交通大学、北京邮电大学和天津大学等院校，中国移动、华为等大型企业都较早开展了无线传感网络的研究。《国家中长期科学与技术发展规划》的重大专项、优先发展主题、科技前沿等部分，都将无线传感网络作为重点研究项目。目前，国家重大基础研究发展计划 973 计划、国家自然科学基金和国家 863 高技术计划也都有项目资助无线传感网络领域的基础理论和关键技术研究。

1.1.2 无线传感网的特点与优势

无线传感器网络与传统的无线网络 (如 Ad hoc 网络、Wi Fi、Wimax 和 GSM 等)有着不同的设计理念，后者在高度移动的环境中通过优化路由和资源管理策略最大化带宽的利用率，同时为用户提供一定的服务质量保证；而无线传感器网络中除少数节点需要移动外，大部分节点都是静止的。研究表明：无线传感器网络与传统无线网络有着明显不同的技术要求，前者以数据为中心，后者以传输数据为目的。一些为自组织的 Ad hoc 网络设计的协议和算法并不适合传感器网络的特点和应用的要求。所以说无线传感器网络是一种自身特点明显、不同于传统网络体系结构的新型网络。

1. 无线传感网的特点

(1) 网络中的节点数量多、规模大

由于无线传感网络节点的通信能力、感知范围和能量有限，且实际工作中为了节能，大部分节点是处于休眠状态的，所以往往需要布设大量的传感器节点来保证网络的容错性和抗毁性，特别是军事应用时，通常在监测区域布设大量的传感器节点，节点数以千计且分布非

常密集。大量密集的传感器节点可以通过不同空间视角获得具有大信噪比的信息；分布式处理大量的采集信息可以降低对单个传感器节点的精度要求；能够增大监测范围，减少盲区；通过节点的密集布设对其节点进行合理的休眠调度，可以有效延长网络的工作时间。

(2) 微型化、集成度高、价格低廉的传感器节点

微机电系统 (MEMS) 技术和低功耗电子技术的发展，使开发功耗低、体积小、价格低廉并集成有微传感器或执行器、微处理器和无线通信等多种功能部件的无线传感器网络节点成为可能，为无线传感器网络的诞生提供了重要的技术条件。上述的节点特征也成为了无线传感器节点的一大特色。

(3) 协作式网络

协作是无线传感器网络执行任务的基本工作方式，一般包括协作式信息采集、协作式信息处理、协作式信息存储、协作式信息传输等。通过协作，多个相同或不同类型的传感器节点可以从不同的空间位置或不同的现象角度（如物理现象、化学现象等）共同对感知对象进行感知，获得更为精确、完整的信息；通过协作，传感器节点可以克服自身处理和存储能力受限的不利因素，在多个节点的共同协作下完成对复杂任务的感知功能；通过协作，传感器节点之间可以通过多跳中继转发实现远距离通信，也可以通过多节点协作发射、多节点协作接收等机制，实现延伸通信距离、改善通信质量、降低网络能耗等目的。

(4) 网络自组、动态拓扑

无线传感器节点通常采用随机布设的方式被放置于缺乏基础通信设施的环境中，如通过飞机播撒等方式将传感器节点布设到人类不可到达或危险的区域中。传感网络中节点的位置不能预先精确设定，节点之间的相邻关系也不能预先确定。另外，传感器节点可能由于能量耗尽或受到环境因素影响而失效，而其他一些节点又可能为了弥补失效节点或增加监测精度而被补充进来，再加上节点可能移动以及采用休眠调度机制，网络拓扑往往处于动态变化之中。而且，无线通信环境的时变特征和节能需求又决定了传感网络的高度时空复杂性。鉴于上述因素，无线传感网络必须能够通过节点之间的相互协调、协商与协同，自动进行配置、管理及调度，来适应不断变化的内、外部环境，保持自身工作的连续性和高效性。无线传感器网络的自组织主要包括自组织通信、网络功能的自调度以及网络的自主管理等。

(5) 以数据为中心、应用相关的网络

传统网络通常以传输数据为目的，所有的功能处理都在网络终端进行，网络的中间节点主要用于分组的存储转发；网络设备使用唯一的 IP 地址进行标识，资源定位和信息传输依赖于终端、路由器、服务器等网络设备的 IP 地址，所以称传统网络是一个以地址为中心的网络。而无线传感器网络是一个任务型的网络，以完成对环境的感知、监测和反馈为目的。网络中的节点采用节点编号标识，但节点标号与节点的位置之间没有必然的联系。用户使用传感器网络查询事件时，直接将所关心的事件通知给网络，而不是去访问具有某个地址标识的节点，网络在获得指定事件的信息之后汇报给用户，这是一种以数据本身作为查询或传输线索的方式，更加接近于自然语言交流的习惯，所以称无线传感器网络是一个以数据为中心的网络。

无线传感器网络通过感知客观世界来获取数据信息，不同的传感器网络应用关心不同的物理量，因此不同的应用背景对传感器网络的要求不同，其硬件平台、软件系统和网络协议之间也存在着很大的差别，所以无线传感器网络不像互联网那样有统一的通信协议平台，针

对每一个具体应用来设计传感器网络，是该网络不同于传统网络的显著特征。

(6) 多跳、对等的通信方式

由于大量传感器节点是密集布设的，且节点之间的通信范围有限，因此，多跳（Multi-Hop）、对等（Peer to Peer）的通信方式比传统的单跳、主从通信方式更适合在无线传感器网络中使用。由于每跳的距离较短，无线收发器可以在较低的能量级别上工作。另外，多跳通信方式可以有效地避免在长距离无线信号传播过程中遇到的信号衰减和干扰等各种问题，提高了数据传输的有效性。

2. 无线传感网络的优势

无线传感网络是由大量微型传感器节点以无线、多跳、自组织方式构成的分布式协作网络，能够实时协同地感知、采集和处理监测区域内各种环境和对象的信息，并通过卫星或互联网发送给用户，使得人们可以在任何时间、地点以及任何环境下实现对物理世界的动态、智能、协同泛在的感知。与传统的无线网络相比，具有以下几个比较明显的优势。

(1) 布设灵活

无线传感网络自组织、协作通信等特点使网络的布设方式灵活便捷，可以通过人工布设、飞机播撒等方式将传感器节点随机部署到监测区域，尤其适合于一些不易到达、危险的应用场合。所以，无线传感网络在布设、使用上的灵活性要远远强于传统的无线网络。

(2) 容错能力强、可靠性高

大量廉价、微型的传感器节点对监测区域进行密集空间抽样，或者对监测目标进行近距离密集监测，可以获取非常高的信息感知精度，这是传统单一大型传感器所难以达到的；大量冗余节点的存在，使得系统具有很强的容错能力；通过不同空间视角获得的信息具有更大的信噪比；由于无线传感网络具有自组织特点，当某一节点发生故障或失效时，其功能可迅速由其他节点替代承担。因此，虽然单一传感器节点的可靠性比较低，但作为一个系统整体，无线传感网络可以获取非常高的可靠性。

(3) 普适泛在

无线传感网络中部署大量集成有传感器、数据处理单元和通信模块的微小节点，它们通过自组织的方式构成网络，利用节点中内置的功能各异、形式多样的传感器对周边环境中的热、红外、雷达等信号进行实时的协作感知，从而探测到光强、湿度、温度和噪声等许多物理现象。这些特点决定了无线传感网络将会是泛在普适环境的核心组成部分，成为普适计算中“无所不在的感知”的基础平台。能够与物理世界紧密耦合是无线传感网络的一大优势。

(4) 高度自治

与传统的无线网络不同，无线传感网络没有固定的基础设施，网络中节点之间的地位是平等的，都具备数据采集和无线路由的功能，而且节点之间自组成网，相互协作。在实际应用中，由于节点的高故障率、休眠机制等因素会导致无线传感网络的拓扑动态变化。因此，为了使网络高效有序地运行，无线传感网络应具有很强的自适应和自治能力。

(5) 经济性好

无线传感网络利用无线传输代替了传统的线缆传输，从很大程度上降低了系统的建设成本。其次，随着半导体技术、计算机技术和微系统技术的迅猛发展以及无线传感网络的广泛应用，单一传感器节点的造价将会越来越低。因此，与传统的无线网络相比，无线传感网络

在经济适用上的优势将越来越明显。

1.1.3 无线传感网的应用

无线传感器网络具有十分广阔的应用前景，在军事、空间探索和灾难拯救等特殊领域有其得天独厚的技术优势，能够广泛应用于环境监测、智能家居、仓库管理、城市交通、健康护理等领域。随着无线传感器网络的不断深入研究和探索，传感器网络将逐渐延伸到人类生活的各个方面。

1. 军事应用

无线传感器网络自身具有部署迅速、隐蔽性高、容错性强的特点，其自组织性和容错能力不会使整个系统因为个别节点的损坏而崩溃，这一特点使得传感器网络非常适合应用于恶劣的战场环境中，包括监视冲突区域、侦察敌方地形和布防、定位攻击目标、评估损失、侦察和探测核、生物、化学攻击等。美国海军开展的网状传感器系统使用战船或飞机战斗群携带的计算机进行感知数据的处理，每艘战船不但依赖自身的雷达系统，还依靠其他战船或者装载有传感器的战机来共享感知数据，利用这些多方面的测量数据可以更加精确地击中目标。无线传感器网络已经成为 C⁴ISRT (Command, Control, Communication, Computing, Intelligence, Surveillance, Reconnaissance and Targeting) 系统不可或缺的一部分。C⁴ISRT 系统的目标是利用先进的高科技技术，为未来的现代化战争设计一个集命令、控制、通信、计算、智能、监视、侦察和定位于一体的战场指挥系统，受到了军事发达国家的普遍重视。在战争中，对冲突区和军事要地的监视也是至关重要的，通过铺设传感器网络，以更隐蔽的方式近距离地观察敌方的布防。当然，也可以直接将传感器节点撤向敌方阵地，在敌方还未来得及反应时迅速收集利于作战的信息，传感器网络也可以为火控和制导系统提供准确的目标定位信息。在生物和化学战中，利用传感器网络及时、准确地探测爆炸中心将为军方提供宝贵的反应时间，从而最大限度地减小伤亡。

2. 环境科学

近年来，人们对生态环境的保护意识越来越高，环境科学所涉及的范围也越来越广，无线传感器网络为环境研究数据的获取提供了极大的方便，可用于洪水和地表监测、气象和地理研究、监视农作物的灌溉和土壤空气情况等。基于传感器网络的 ALERT 系统中就有数种传感器可监测降雨量、河水水位和土壤水分，并据此来推测山洪暴发的可能性。除此之外，无线传感器网络还有一个十分重要的应用就是对生态多样性的描述，能够监测动物栖息地的生态情况，并将信息数据及时地传送给相关机构。

2002 年 10 月，Intel 公司发布了“基于微型传感器网络的新型计算的发展规划”，致力于在环境监测、森林灭火、海底板块调查、行星探查等领域的应用。Intel 研究中心伯克利实验室和大西洋学院的研究人员计划使用无线传感器网络来研究海岛上的生态环境。

3. 医疗健康

无线传感器网络在医疗系统和健康护理方面的应用越来越成熟，包括实时监测人体的各项生理数据、医院的药物管理、远程医疗等。中国台湾国立大学开展了涵盖生物技术、网络通信技术、信息技术以及纳米技术等多项前沿科学技术的人体生理数据监测项目研究，通过自主设计的临床体温计和血压、脉搏测量仪来监测患者的体温、血压等常规数据，并将相关信息发送给医疗中心。Intel 公司正在研究通过压力检测来预测初期溃疡的“Smart Socks”，

以及根据伤口化脓情况来确定有效抗生物物质的“智能绷带”，为未来智能医疗的实现提供了良好的技术支持。无线传感器网络的应用为有视力疾病的患者带来了福音，在 SSIM（Smart Sensors and Integrated Microsystems）计划中，用 100 个微型传感器来代替视网膜并将它置入人眼，这样可以使失明者或视力极差者恢复到一个较为正常的视力水平。

4. 商业应用

无线传感器网络可用于实现家居环境、工作环境等的智能化。欧洲的“DELTA 计划”在汽车出厂之前就把车载信标集成到车载的计算机中，这是车辆识别和车辆与路测设备通信的必选器件，而且还可以构成自动定位和汽车导航等更加完整的应用体系。通过在家电和家具中嵌入传感器节点，并利用无线网络和互联网连接在一起，将会为人们提供更加舒适、方便和更具人性化的智能家居环境。除此之外，还可以利用无线传感器网络来建立智能交通系统、智能幼儿园等。

无线传感器网络重要的科学意义和应用价值，已经引起了世界发达国家学术界、军事部门和工业界的极大关注。2003 年 2 月，美国《技术评论》杂志评选出对人类未来生活产生深远影响的十大新兴技术，无线传感器网络位列第一。2003 年 8 月，美国《商业周刊》的技术评论将无线传感器网络定位成 21 世纪高技术领域的四大支柱型产业之一。

1.2 无线传感网节点软件技术

无线传感器网络的网络体系结构，也可以称为传感器网络的结构组成，如图 1-2 所示。

无线传感器网络系统通常包括传感器节点（Sensor Node）、Sink 网关节点（Sink Node）和管理节点。大量传感器节点随机部署在监测区域（Sensor Field）内部或附近，能够通过自组织方式构成网络。传感器节点监测到的数据经过多跳路由到达 Sink 网关节点，再利用互联网或卫星等传输到管理节点，最后用户通过管理节点对无线传感器网络进行配置和管理，发布监测任务以及收集监测数据。

传感器节点通常是一个微型的嵌入式系统，在不同应用中传感器节点的结构也不尽相同，但一般都由数据采集单元、数据处理单元、数据存储单元、数据传输单元、电源和嵌入式操作系统等部分组成，如图 1-3 所示。其主要特点介绍如下。

1) 节点的设计面向应用，不同的应用领域对数据采集、处理、发送的要求有很大的差别，所以对传感节点的要求也不尽相同。比如，环境污染监测系统和智能仓库系统在网络拓扑结构、通信模型、数据传送模型等方面都有很大的区别。所以，传感节点的设计必须与应用领域密切结合。

2) 节点的资源受限，具体表现在电池能源有限、计算及存储能力低、通信能力差等方面。由于节点都趋于微型化，因此必然导致节点所带电池的容量受到很大的限制，所以节点

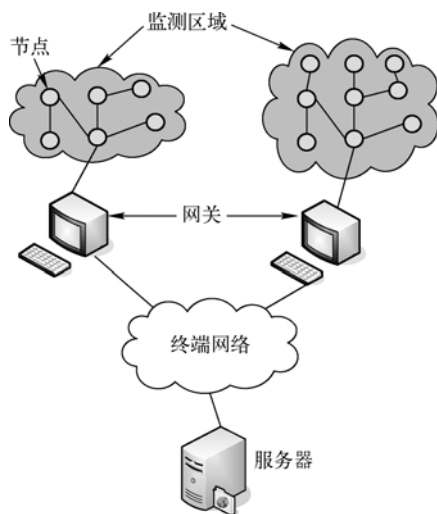


图 1-2 无线传感器网络的体系结构

的节能设计是关键；节点的低成本导致了其处理和存储能力的不足；而节点的通信速度则很大程度上受到了无线通信带宽的影响。

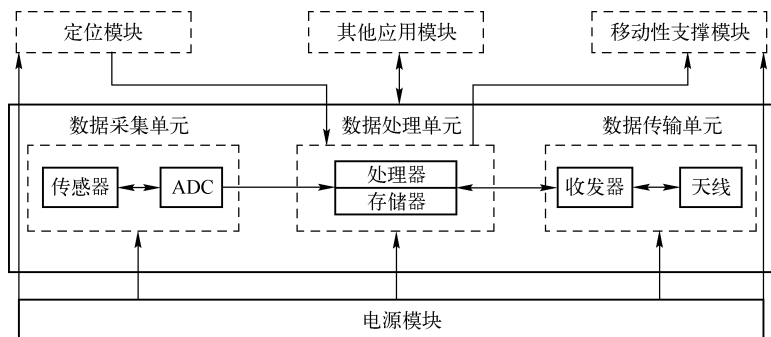


图 1-3 无线传感器网络节点的基本组成

传感器节点的设计与实现包括硬件技术和软件技术两个方面。无线传感器网络节点软件实现技术与硬件平台密切相关，要充分利用硬件平台的处理能力，以尽量小的处理（能量）开销实现节点功能。相对于硬件技术而言，软件开发与实现更具有特色，本书将重点介绍无线传感器网络节点的软件实现技术。

无线传感器网络节点上软件应实现的功能主要包括：采集并处理传感器数据、组网通信（完成数据的传输或转发）、网内数据处理、全网时间同步、节点定位及支持上述功能的操作系统内核（管理硬件、多个并发执行任务的调度与同步）等。对于不同的无线传感器网络应用领域，某些软件功能可能不同，即便某些软件功能相同，其具体实现也可能不同。另外，无线传感器网络节点上通常没有用户接口部分。

无线传感器网络节点的软件部分包括嵌入式操作系统和应用，其中嵌入式操作系统是软件的重要组成部分，负责管理硬件与传感器、调度与同步、组网通信等，为应用及软件中间件提供安全的硬件资源和方便的硬件抽象描述，支撑应用软件的研发。图 1-4 给出了比较常用的无线传感器网络的软件功能体系结构示意图。

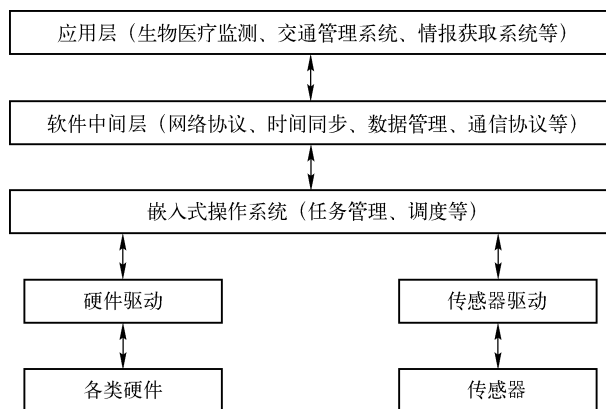


图 1-4 无线传感器网络节点软件功能体系结构

在无线传感器网络节点上实现软件功能时，要充分考虑无线传感器网络的能量有限、计

算能力有限、网络节点数量庞大、分布范围广、网络动态性强以及网络数据量大等特点。嵌入式操作系统是无线传感器网络节点软件中最重要的组成部分,从很大程度上来讲,整个无线传感器网络节点的软件开发都是围绕着嵌入式操作系统来进行的。目前,几种典型的应用于无线传感器网络的操作系统包括:美国加利福尼亚大学伯克利分校研发的 TinyOS (Tiny Micro Threading Operating System)、美国科罗拉多大学开发的 MOS (MANTIS OS) 等,也有研究者试图将传统的嵌入式操作系统如 μ COS、 μ Clinux 进行改进并应用于无线传感器网络。目前,研究和应用比较多的是 TinyOS。TinyOS 具有微型化、支持轻量级并发操作、灵活、低功耗等优点,已经被世界上成千上万的研发人员所采纳。

1.3 TinyOS

无线传感器网络操作系统是无线传感器网络系统的基本软件环境,是绝大部分无线传感器网络应用软件开发的基础平台,但其并没有特定的系统用户界面,也不是一系列特定的系统服务,而是定义了一套通用的界面框架,允许应用程序选择服务和实现;它提供框架的模块化,以便适应硬件的多样性,同时允许应用程序重用通用的软件服务和抽象。对无线传感器网络操作系统的研究涉及的内容很多,如任务管理、内存管理、低功耗通信协议、电源管理、文件系统等。归纳起来,无线传感器网络的特点决定了其操作系统的特点有如下几方面。

1) 传感器节点大量而密集的使用要求其只能使用廉价的 CPU 和存储器,所以节点的存储资源极其有限,同时由于采用电池供电,电能难以及时补充,因此要求无线传感器网络操作系统应该可以根据不同的应用系统进行裁剪和扩充,只实现必要的功能,从而以最小的代码量满足系统需求,最大程度上降低计算、存储以及通信方面的能量开销。

2) 鲁棒性好,传感器节点通常是一次性使用的,只要部署下去就需要在无人值守的环境下连续工作数月甚至数年的时间,且运行期间无法进行维护和检修,因此要求操作系统可以在能源不足、处理及存储能力有限和通信不畅的情况之下依然能够很好地工作,特别是当某部分软、硬件出现故障时,系统可以及时有效地调整整个系统继续运行。

3) 系统要能够适应硬件的多样性,以便硬件的升级。操作系统的设计方案必须方便程序移植到新的硬件平台上,而且还要能够适应不同应用的差异性,因为不同的应用对网络的生存周期、通信方式等有不同程度的要求。

4) 程序模块化,无线传感器网络上的应用和硬件平台很多,采用模块化编程方式可对上层应用模块屏蔽底层硬件的操作,使得应用程序员不需要对硬件编程,只要重写底层模块就可以使得操作系统应用于不同的平台上,增加了软件的重用性,提高了开发效率。

5) 应具有可重构能力,当节点工作失常时,要求能自恢复、自组织。无线传感器网络是一种反应系统,通常被应用于有很强实时要求的领域,所采集的数据需要及时地传给观测点,并通过执行器对环境的变化做出快速的反应,对于不同的事件所要求的响应周期也不同,所以要求操作系统必须内置可靠高效的实时任务调度机制。同时,节点之间通过协作的工作方式来处理大量的数据,这就要求操作系统可以快速高效地使用资源,并能够及时处理好大量并发的执行流程。

美国加利福尼亚大学伯克利分校研发的 TinyOS 正是具备了上述特点的一套无线网络传感器操作系统。TinyOS 是一个由事件驱动的、基于组件的、开放源代码的无线传感器网络

操作系统，是加利福尼亚大学伯克利分校的 David Culler 领导的研究小组为无线传感网量身定做的嵌入式操作系统。目前，由于绝大多数的无线传感网研究项目都采用了 TinyOS 操作系统，因此该操作系统已经成为无线传感网研究领域事实上的标准平台。TinyOS 操作系统已经不再由 UCB 的研发小组单独开发和升级，而是成为了 SourceForge 的一个项目，由众多研发小组共同开发和维护。同时，TinyOS 联盟论坛成立，以共同探讨和制订 TinyOS 的发展规划。

1.3.1 TinyOS 的特点

TinyOS 操作系统的特点主要体现在以下几个方面。

(1) 轻量级线程技术

线程和轻量级线程的提出都是为了提高操作系统的并发性。线程是为了实现 CPU 上的并发，将程序中若干控制流的功能进行模块化，是进程中能够独立运行的更小的单位。线程可以共享进程的内存空间，线程之间进行切换时，CPU 会用一部分时间来将寄存器状态的信息存入线程控制块 (Thread Control Block, TCB) 中。但传感器节点的处理能力有限，线程间的切换会占用处理器很多时间，这也就是嵌入式操作系统难以在无线传感器节点上取得较好运行效果的原因。所以，TinyOS 提供了任务和硬件事件处理两级调度体系。轻量级线程比一般的线程更为简单，此种线程按照 FCFS 的方式进行调度，轻量级线程之间不允许抢占；而当进行硬件处理线程时，中断处理线程可以打断用户的轻量级线程和低优先级的中断处理线程，对硬件中断进行快速响应。

(2) 主动消息通信模式

主动消息通信是一种早期就应用于并行计算中的高性能通信模式，是基于事件驱动的高性能通信方式。在主动消息通信方式中，每个消息都由相应的应用层句柄进行维护，当目标节点收到消息后，就会把该消息中的数据作为参数，传递给应用层的处理器进行处理。句柄首先从无线传感器网络中获取消息，然后将数据合并到计算中或返回一个应答消息给消息发送者，实现了通信和计算的重叠，从而很大程度上降低了通信的代价。

(3) 组件化编程

组件技术是各类软件重用方法中最重要的一种，也是分布式计算和 Web 服务的基础。基于组件的编程可以将独立的组件组合到需要的地方，且不需要知道其具体的实现过程；不同来源的组件结合在一起可以快速构成一个实用的应用程序。TinyOS 仅支持 nesC 语言开发的组件，并采用自定义的组件机制。

(4) 硬件抽象层

美国微软公司首先提出了将底层与硬件相关的部分单独设计成硬件抽象层的概念，使得嵌入式操作系统在不同的硬件平台上可以方便地进行移植，并增加了程序的移植性，简化了程序的开发过程。TinyOS 中的硬件抽象层分别由硬件表示层、硬件适应层及硬件接口层来实现，它隐藏了特定硬件平台的硬件接口实现细节，为嵌入式操作系统提供了一个虚拟的、统一硬件平台接口，使其具有硬件无关性并可在多种硬件平台上进行移植。

(5) 事件驱动模式

在事件驱动的程序设计中，代码执行路径是事先不可预知的，响应不同的事件时会执行不同区域的代码，事件的发生顺序决定了代码的执行顺序，所以应用程序每次所执行的代码

路径都是不同的。而无线传感器网络中，事件的发生顺序是无法得知的，但事件的种类是可以预测的。因此，事件驱动模式的应用就增强了传感器节点的灵活性，降低了传感器节点的编程难度。TinyOS 中有两种事件：硬件事件和软件事件。前者是硬件中断处理程序，后者是通过 signal 来触发的。

1.3.2 TinyOS 体系结构

TinyOS 操作系统采用了组件层次结构，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，以便适应不同的应用环境。完整的系统由一个调度器和一些组件组成，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。TinyOS 的体系结构如图 1-5 所示。



图 1-5 TinyOS 的体系结构

第 2 章 TinyOS 安装与常用命令

通过前一章节的介绍，我们了解到无线传感网络是有大量的具有通信和计算能力的微小传感器节点，布设成能够根据环境自主完成任务的智能自组织网络系统。TinyOS 是由加州大学伯克利分校专门为无线传感器网络开发的一种微型操作系统，是目前主流的无线传感器网络的操作系统。目前，基于 TinyOS 2.x 的研究正在国内外进行。本章我们主要介绍 TinyOS 的安装、目录结构及一些相关的使用命令。在本章结尾处，我们简单地介绍一个 PowerUp 应用程序例子。

2.1 TinyOS 安装

目前 TinyOS 最新的版本是：TinyOS 2.1.1。TinyOS 的安装可以在 TinyOS 1.x 的基础上升级，也可以使用传统的方法安装。使用传统的方法可以安装一个拥有完整的 TinyOS 的虚拟环境或者手动安装 TinyOS 到主操作系统中。这里主要介绍 TinyOS 2.1.1 在 Windows 和 Linux 下的安装步骤。

2.1.1 在 Windows 下安装 TinyOS

在 Windows 下安装 TinyOS 2.x 可以按照 [http://docs.tinyos.net/index.php/Installing_TinyOS_2.1.1#Manual installation on your host OS with RPMs](http://docs.tinyos.net/index.php/Installing_TinyOS_2.1.1#Manual_installation_on_your_host_OS_with_RPMs) 的安装步骤进行。这里需要注意的是，Cygwin 不支持 Windows Vista 操作系统。安装过程共需六个步骤。

1. 安装 Java 1.6 JDK

在 <http://java.sun.com> 上下载 Java 1.6 JDK。需要注意的是，若安装 Java 1.5 或者 Java 1.4，进行 `tos-check-env` 测试环境会显示无错误安装成功；安装 Java 1.6 版本在进行 `tos-check-env` 时会有一个有关 Java 的警告，但这并不影响系统环境测试，这里我们安装 Java 1.6 版本。安装完成之后，需要设置计算机的环境变量，新建两个环境变量，以便使用 JDK。右击我的电脑→属性→高级→环境变量→系统变量栏→新建→新建系统变量，弹出新建系统变量对话框。

变量名：CLASSPATH

变量值：.;C:\Program Files\Java\jdk1.6.0_24\bin;C:\Program Files\Java\jdk1.6.0_24\lib\tools.jar

作用：新建名为 CLASSPATH 的环境变量是为了提供应用程序在运行期间寻找所需资源的路径，比如类、文件等。值得注意的是，最前面应加上“.”，意为首先在当前目录中查找。

变量名：JAVA_HOME

变量值：C:\Program Files\Java\jdk1.6.0_24

作用：新建名为 JAVA_HOME 的环境变量指向 JDK 的安装路径，在该路径下可以找到

bin、lib 等目录。值得注意的是，JAVA_HOME 必须全部大写，中间应有下划线隔开。

编辑变量名：PATH

添加变量值：C:\ProgramFiles\Java\jdk1.6.0_24\bin;C:\ProgramFiles\Java\jdk1.6.0_24\jre\bin
到原有 PATH 变量值的前面，中间用分号隔开。

作用：设置该环境变量的目的是为了指向 JDK 的 bin 目录，因为在 bin 目录里存放的是各种编译执行命令。值得注意的是，系统本身就有多种 PATH 环境变量，只需把上述变量值直接放到原变量值的前面即可，中间用“；”隔开。

此时，环境变量已经设置完毕。

现在应检测 Java 环境是否配置成功。可以编个 Java 小程序简单地测试一下。打开记事本，输入程序代码为

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

将此程序另存为 HelloWorld.java 文件，需要注意的是，文件的扩展名应当是“java”，表示这是一个 Java 程序。然后单击“开始”菜单，选择“运行”命令，在之后弹出的“运行”对话框中输入“cmd”，在之后弹出的 DOS 窗口中，切换到文件 HelloWorld.java 所在目录，输入命令

```
javac HelloWorld.java
java HelloWorld
```

如果能正确输出“Hello, World!”，则说明环境变量设置成功。

2. 安装 Cygwin

在 Windows 下运行 TinyOS 是基于 Cygwin 这个软件平台，Cygwin 提供了一个在 Windows 下的 shell 环境和开发 TinyOS 时用到的大多数 UNIX 工具，比如 perl 和 shell 脚本。首先从 http://docs.tinyos.net/index.php/Installing_TinyOS_2.1.1 上下载 cygwin-files.zip 到 C 盘的 cygwin-files 目录下，解压后双击 setup.exe 开始安装。setup.exe 文件还可以用于软件的重新安装，能够添加、修改或升级 Cygwin 配置。具体的安装过程说明如下。

1) 选择安装方式：当出现"Choose A download source"界面时，选择"Install from Local Directory"。也可选择从网络安装，但网络安装速度较慢。这里我们选择从本地安装。

2) 设置安装目录：当出现"Select Root Install Director"界面时，Root Directory 选择“c:\cygwin”，这是默认的安装目录。也可以对其进行更改，安装在自己所需要的目录下。Install for 选择“All Users”；Default Text file type 选择“Unix\Binary”。

3) 选择本地安装包源文件的本地存储路径：当出现"Select local Package directory"界面时，选择“c:\cygwin-files”。这里应注意，这一步的依据为压缩文件 cygwin-file.zip 存在的位置。

4) 选择安装策略：安装策略有“Keep”、“Prev”、“Curr”、“Exp”等选项。

- Keep: 指保持目前已经安装的版本不动，不替换计算机目前的版本，升级时比较方便。

- Prev: 指安装上一个版本。
- Curr: 指安装最新版本, 这是默认选项。
- Exp: 指安装试验版。

安装方式共有以下四类。

- Default: 表示是默认选择, 可单击包左边的+号展开, 看看默认安装究竟是些什么内容。
- Install: 表示要下载或安装该包的全部内容。
- Reinstall: 表示重新下载或安装该包的内容。
- Uninstall: 卸载。

这里我们选择 Install, 安装该软件的全部内容。然后进行“下一步”直至完成安装。最后, 安装好 Cygwin 后如图 2-1 所示。

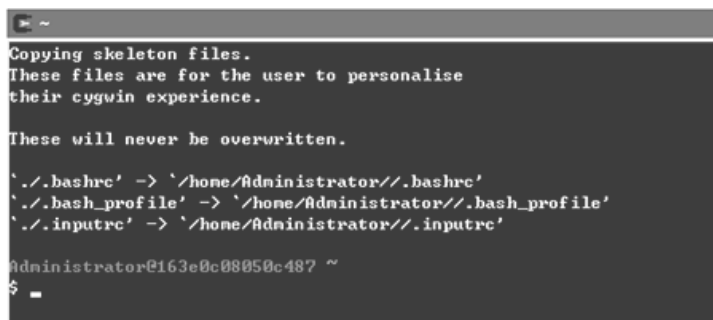


图 2-1 Cygwin 安装完成图

3. 安装平台交叉编译器

平台交叉编译器是用于将 C 代码交叉编译成硬件可以运行的二进制文件。在编译 TinyOS 程序时, 会产生对特定微控制器芯片的 C 代码和汇编代码, 这就需要有相应的平台的编译器支持。如编译 Mica 系列节点上的应用程序, 需要安装 Atmega128 单片机的 AVR 工具包, Telos 系列则需要安装 MSP430 工具包。

从 http://docs.tinyos.net/index.php/Installing_TinyOS_2.1.1 上下载 Atmel AVR 工具, MSP430。打开 Cygwin 后默认的目录是“\home\用户名”, 这个可以用“pwd”命令查看。在命令行输入“rpm-qa”可以查看系统目前已经安装了的程序。刚安装完 Cygwin 时, 系统还没有任何软件。可以把下载好的 AVR 工具和 MSP430 的 RPM 包文件复制到“home\用户名”的文件夹下, 这样可以直接安装, 不用再考虑路径。由于这些文件名都挺长, 输入起来较麻烦。可以输入前几个字母, 按〈table〉键, 系统会自动载入名字。如安装第一个文件 avr-binutils-2.17tinyos-3.cygwin.i386.rpm, 只需输入“rpm -ivh avr-b”, 然后按〈table〉键, 这个文件名就会自动载入。

1) 安装 AVR 工具包:

```
avr-binutils-2.17tinyos-3.cygwin.i386.rpm
avr-gcc-4.1.2-1.cygwin.i386.rpm
avr-libc-1.4.7-1.cygwin.i386.rpm
avarice-2.4-1.cygwin.i386.rpm
avr-insight-6.3-1.cygwin.i386.rpm
```

```
avrdude-tinyos-5.6cvs-1.cygwin.i386
```

2) 安装 PXA27x 工具包:

```
xscale-elf-binutils-2.15tinyos-1.cygwin.i386.rpm  
xscale-elf-gcc-3.4.3-1.cygwin.i386.rpm  
xscale-elf-newlib-1.11.0tinyos-1.cygwin.i386.rpm
```

3) 安装 MSP430 工具包:

```
msp430tools-base-0.1-20050607.cygwin.i386.rpm  
msp430tools-python-tools-1.0-1.cygwin.noarch.rpm  
msp430tools-binutils-2.16-20050607.cygwin.i386.rpm  
map430tools-gcc-3.2.3-20050607.cygwin.i386.rpm  
msp430tools-libc-20080808-1.cygwin.i386.rpm
```

在安装的过程中, 标准的“rpm”安装包用命令“rpm -ivh 文件名”安装。如下是一些错误处理。

- “the rpm was intended for a cygwin_nt-5.1 operating system”说明当前的 Cygwin 版本与 TinyOS 系统的安装包不兼容, 可能需要升级 Cygwin。可使用命令“rpm -ivh --ignoreos”安装;
- “you have a newer version already installed”, 则可使用命令“rpm -Uvh -force”安装;
- “rpm error that indicates that you are missing/bin/sh”, 可使用命令“rpm -Uvh -force --nodeps”安装。

这里值得注意的是, 由于每个安装者所使用的系统可能不同, 会出现不同的问题, 在安装的过程中可以参考官网 www.tinyos.net 上的安装方法。

4. 安装 TinyOS 源码与工具包

TinyOS 工具包包括 nesC 编译器和一系列开发 TinyOS 源代码工具。TinyOS 操作系统的内核和应用程序都是由 nesC 语言编写的, 所以必须安装 nesC 工具才能将 TinyOS 应用程序编译成可交给平台交叉编译器的 C 代码。从 http://docs.tinyos.net/index.php/Installing_TinyOS_2.1.1 上下载 rpm 包, 具体安装方法与前面相同。

```
nesc-1.3.1-1.cygwin.i386.rpm  
tinyos-deputy-1.1-1.cygwin.i386.rpm  
tinyos-tools-1.4.0-3.cygwin.i386.rpm
```

5. 安装 TinyOS 2.1 源代码包

TinyOS 2.1 源代码包的作用是编译和下载 TinyOS 程序。从 http://docs.tinyos.net/index.php/Installing_TinyOS_2.1.1 上下载 tinyos-2.1.1-3.cygwin.noarch.rpm, 安装方法与前面相同, TinyOS 2.x 系统的源代码默认安装到 C:\cygwin\opt\tinyos-2.x 路径。

然后设置环境变量。TinyOS 在 Windows 环境变量的设置应对照如下信息。

TOSROOT	Opt/tinyos-2.x
TOSDIR	\$TOSROOT/tos
CLASSPATH	C:\cygwin\opt\tinyos-2.x\support\sdk\java\tinyos.jar

(续)

MAKERULES	\$TOSROOT/support/make/Makerules
PATH	/opt/msp430/bin:\$PATH

这里，我们使用 UltraEdit 在 C:\cygwin\etc\profile.d 目录下创建文件 tinyos.sh，输入以下内容，这里特别要注意的地方是：必须使用 UNIX 换行符-LF 保存此文件。

```
# script for profile.d for bash shells, adjusted for each users
# installation by substituting /opt for the actual tinyos tree
# installation point.
export TOSROOT="/opt/tinyos-2.x"
export TOSDIR="$TOSROOT/tos"
export CLASSPATH="C:\cygwin\opt\tinyos-2.x\support\sdk\java\tinyos.jar"
export CLASSPATH="$CLASSPATH;."
export MAKERULES="$TOSROOT/support/make/Makerules"
export PATH="/opt/msp430/bin:$PATH"
export PATH="/cygdrive/c/Program Files/Java/jdk1.6.0_24/bin:$PATH"
# Extend path for java
type java >/dev/null 2>/dev/null || PATH=`/usr/local/bin/locate-jre --java`:$PATH
type javac >/dev/null 2>/dev/null || PATH=`/usr/local/bin/locate-jre --javac`:$PATH
echo $PATH | grep -q /usr/local/bin || PATH=/usr/local/bin:$PATH
```

然后在 Cygwin 中执行命令 “tos-install-jni”，若出现错误：

```
Installing 32-bit Java JNI code in /cygdrive/c/Program Files/Java/jdk1.6.0_24/jre/bin ...
install: cannot stat `/usr/lib/tinyos/*-32.dll': No such file or directory
```

说明缺少*-32.dll 文件，则需要将目录 “C:\cygwin\lib\tinyos” 和目录 “C:\Program Files\Java\jdk1.6.0_24\jre\bin” 以下目录中的 toscomm.dll，重命名为 “toscomm-32.dll”。然后再重新执行命令 “tos-install-jni”，若出现：

```
Installing 32-bit Java JNI code in /cygdrive/c/Progrm Files /Java/jdk1.6.0_24/jre/bin...
done.
```

说明文件加载成功。

最后安装好各 rpm 包后用命令 “rpm -qa” 查看如图 2-2 所示 Cygwin 系统已安装的软件。

6. 安装 Graphviz

TinyOS 环境包含 nesdoc 工具，该工具可以自动生成可视化的组件关系图表。Graphviz 是 nesdoc 用来画图的一个开源工具。从 http://docs.tinyos.net/index.php/Installing_TinyOS_2.1.1 下载 graphviz-1.10.exe 文件，双击运行直接安装在 Windows 中就可以了。安装完后退出 Cygwin 再重新运行。

到现在 TinyOS 2.x 在 Windows 下的安装已经完毕。下面我们进行环境测试。环境测试可分为以下几步进行：

7. 环境测试

运行 Cygwin，在命令行输入命令：

```
Administrator@23zhangyanjun1 ~
$ rpm -qa
avr-gcc-4.1.2-1
avarice-2.4-1
avrdude-tinyos-5.6cvs-1
xscale-elf-gcc-3.4.3-1
msp430tools-base-0.1-20050607
msp430tools-binutils-2.16-20050607
msp430tools-libc-20080808-1
deputy-1.1-1
tinyos-2.1.0-2
avr-binutils-2.17tinyos-3
avr-libc-1.4.7-1
avr-insight-6.3-1
xscale-elf-binutils-2.15-1
xscale-elf-newlib-1.11.0tinyos-1
msp430tools-python-tools-1.0-1
msp430tools-gcc-3.2.3-20050607
nesc-1.3.0-1
tinyos-tools-1.3.0-1
```

图 2-2 查看 Cygwin 已安装的软件

1) \$ tos-check-env

tos-check-env completed with errors:

WARNING: The JAVA version found first by tos-check-env may not be version 1.4 or version 1.5 one of which is required by TOS. Please ensure that the located Java version is 1.4 or 1.5

Depending on your PATH environment variable, there is often another version of java.exe in c:\windows\system32 that is “seen” first. Check that this is version 1.4 or 1.5 or reconfigure your PATH environment variable if this is the case.

该命令能够检测出 TinyOS 环境中绝大多数的配置情况。前面在安装过程的第一步中我们已经提到，若安装的是 Java 1.5 或者 Java 1.4，则进行 tos-check-env 时如果出现 “tos-check-env completed without error” 则说明安装成功。现在我们安装的是 Java 1.6 版本，进行 tos-check-env 时出现以上内容，这只是 Java 的版本问题所引起的，并不影响 TinyOS 系统的运行及 nesC 语言的开发，所以说明安装成功。

2) 测试 Java 版本，在命令行输入：\$ which java，若出现：

```
/cygdrive/c/WINDOWS/system/java
```

则在前面所建的 tinyos.sh 文件中加入：

```
export PATH="/cygdrive/c/Program Files/Java/jdk1.6.0_24/bin:$PATH"
```

这会导入 Java 路径到 PATH 变量。然后再次执行 “\$ which java”，若出现：

```
/cygdrive/c/Program Files/Java/jdk1.6.0_24/bin/java
```

则说明 Java 环境可用。

3) 检查 TinyOS 编译系统环境是否可运行。

运行 Cygwin，在命令行输入命令 “\$ printenv MAKERULES”，如果可以看到：

```
/opt/tinyos-2.x/support/make/Makerules
```

则说明 TinyOS 编译系统环境是可运行的。

4) 运行 make 命令 (可以是 make mica2, make telosb, or make micaz sim...)。

首先进入 opt/tinyos-2.x/apps/Blink 文件夹, 然后在命令行输入命令:

```
$ make telosb
```

则可以看到如下的输出:

```
mkdir -p build/telosb
compiling BlinkAppC to a telosb binary
ncc -o build/telosb/main.exe -Os -O -mdisable-hwmmul -Wall -Wshadow -Wnesc-all -
target=telosb -fnesc-cfile=build/telosb/app.c -board= -DDEFINED_TOS_AM_GROUP=0x2
2 -DIDENT_APPNAME="BlinkAppC" -DIDENT_USERNAME="Administator" -DIDENT_
HOSTNAME="s
cy\" -DIDENT_USERHASH=0x71b2a242L -DIDENT_TIMESTAMP=0x4937fa3dL -DIDENT_
UIDHASH=
0xf51978b3L BlinkAppC.nc -lm
compiled BlinkAppC to build/telosb/main.exe
2650 bytes in ROM
55 bytes in RAM
msp430-objcopy --output-target=ihex build/telosb/main.exe build/telosb/main.ihex
writing TOS image
```

这是因为在安装好 TinyOS 后, 在文件夹 opt/tinyos-2.x/apps/Blink 下有一个 Blink 应用程序 BlinkC.nc, make telosb 命令是把这个 Blink 应用程序编译成可在 TelosB 平台上运行的代码。所以 make telosb 命令就把文件夹 opt/tinyos-2.x/apps/Blink 里面的 Blink 应用程序编译成了可在 TelosB 平台下运行的代码。当编译通过后, 就可以直接把程序代码下载到节点上, 我们将会在后面介绍。

至此, Cygwin 和 TinyOS-2.x 环境配置完毕。

2.1.2 在 Linux 下安装 TinyOS

在 Linux 下安装 TinyOS 2.x, 可以参考网站 http://docs.tinyos.net/index.php/Installing_TinyOS 2.1.1 的安装步骤进行, 但是在安装的过程中, 可能会碰到各种问题。这里我们在 Red Hat 9.0 下安装 TinyOS 2.1.1, 如同在 Windows 下安装 TinyOS 2.1.1, 在 Linux 下安装 TinyOS 2.1.1 也需以下几个步骤。

1) 安装 Java。首先从网站 https://dct.sun.com/dct/forms/reg_us_2704_620_0.jsp 上注册一个账号, 然后会在注册时所填写的邮箱里有个网站链接, 点开这个链接进入该官网就可以下载 JDK。这里我们所下载的是 jdk_1_5_0_19-linux-i586-rpm.bin, 这在 Red Hat 下并不能直接安装, 先把该文件放在所要安装的文件夹 usr/java 下, 此时新建终端, 输入命令:

```
#chmod a+x jdk_1_5_0_19-linux-i586-rpm.bin
```

此命令的作用是更改此包的权限, 将其设置为可行, 然后输入命令:

```
#!/jdk_1_5_0_19-linux-i586-rpm.bin
```

此命令可得到一个能在 Linux 下安装的 rpm 包，然后安装 rpm 包，输入命令：

```
#rpm -ivh jdk_1_5_0_19-linux-i586-rpm
```

这样 JDK 便会自动运行直至安装完成。

现在来对 Java 配置环境变量。对 Java 配置环境变量，即在 `etc/profile` 文件后添加 `JAVA_HOME`、`PATH` 和 `CLASSPATH` 的值。输入命令：

```
#vi /etc/profile
```

此命令是打开 `profile` 文件，在此文件末尾添加如下语句：

```
export JAVA_HOME=/usr/java/jdk1.5.0
export PATH=$JAVA_HOME:$JAVA_HOME/bin:$PATH
export CLASSPATH=.:$JAVA_HOME/lib/tools.jar:$JAVA_HOME/lib/rt.jar
```

这样，环境变量就已设置完毕。但是环境变量一般在系统重启后才会生效，由于 Linux 系统启动较慢，我们可以使用命令 `#source etc/profile` 使其环境变量的配置生效，但此命令只会让环境变量的配置本次会话生效，重启系统才能使环境变量的配置永久生效。然后输入命令 `#Java -version` 查看版本，若显示正确版本号则表示安装成功。

2) 在网站上下载其他的 rpm 包，分别安装平台交叉编译器、TinyOS 源码与工具包和 TinyOS 2.x 源代码。安装方法类似于在 Windows 下安装各 rpm 包，参考网站 www.tinyos.net 的安装方法，使用命令 `rpm -ivh` 逐个地安装 rpm 包。在安装完 TinyOS 2.x 源代码时，需要配置环境变量。需要在 `etc` 文件夹下建立一个 `tinyos.sh` 文件，最好是复制一个其他.sh 文件，用 VI 编辑器打开，然后把里面的内容用以下代码替换。

```
export TOSROOT=/opt/tinyos-2.x
export TOSDIR=$TOSROOT/tos
export PATH="/opt/msp430/bin:$PATH"
export CLASSPATH=.:$TOSROOT/support/sdk/java/tinyos.jar:$JAVA_HOME/lib/rt.jar:$JAVA_HOME/lib/tools.jar
export MAKERULES=$TOSROOT/support/make/Makerules
```

然后保存。

现在更改并口属性，在任意路径下输入：

```
chmod 666 /dev/<devicename>
```

最后，安装 Graphviz。

在 [http://download.chinaunix.net/download.php?id=8163&ResourceID=4119\[url\]](http://download.chinaunix.net/download.php?id=8163&ResourceID=4119[url]) 上下载压缩包 `graphviz-1.10.tar.gz`，然后完成以下内容。

```
#tar xzf graphviz-2.24.0.tar.gz
#cd graphviz-2.24.0
#./configure
#make
#make install
```

现在已安装完毕，输入“#tos-check-env”，检测 TinyOS 有没有安装成功，若出现“tos-check-env completed without error”，说明安装成功。

2.2 目录结构

由前面知，TinyOS 在 Windows 里是安装在 Cygwin 下。Cygwin 提供了一个在 Windows 里的 shell 环境和开发 TinyOS 时用到的大多数 UNIX 工具。这里我们主要介绍 Cygwin 的目录结构和 tinyos-2.x 的目录结构。

2.2.1 Cygwin 的目录结构

Cygwin 的目录结构与 Linux 的目录结构相似，它们的目录结构均是采用级层式的树状目录结构。安装 TinyOS 时会自动创建很多默认的目录，这些目录都具有特殊的功能。下面我们先简略地介绍 TinyOS 安装环境（Cygwin 和 Linux）的目录结构，然后再详细地介绍 TinyOS 的目录结构。Cygwin 的目录结构如图 2-3 所示。



图 2-3 Cygwin 目录结构图

1) bin: binary 的缩写。此目录里面存放着引导启动所需的命令或普通用户的标准命令。

2) dev: device（设备）的缩写。此目录是接口设备目录。

3) etc: 用来存放系统的配置文件和子目录。

4) home: 用户的主目录。比如用户名为 Administrator，则用户的主目录为 home\Administrator。

5) lib: 存放着系统最基本的动态链接共享库，几乎所有的文件都会用到此目录的这些共享库。

6) opt: 用来存放 Cygwin 下安装的软件。

7) sbin: 存放系统启动时所需执行的程序。

8) tmp: 用来存放不同程序执行时所产生的应用程序。

9) usr: 存放用户使用的系统命令和应用程序等信息。

10) var: 具有变动性质的相关程序目录，例如 log 文件。

2.2.2 TinyOS 的目录结构

TinyOS 安装在 opt 里面，TinyOS 主要包括 apps、supports 和 tos 三个文件夹，TinyOS 的结构目录如图 2-4 所示。

1) apps: 主要包含 tinyos-2.x 下应用性文件。

2) supports: 主要由以下几个部分构成。

- make: 包含构建系统的一些 Makefile 的脚本文件，这些文件主要是作用于 Mica2、TelosB 等平台。

- sdk: 包含支持 TOSSIM 仿真的 C 和 pytohn，支持 Java 应用程序的 Java 文件。要使用此文件夹里的功能需设置

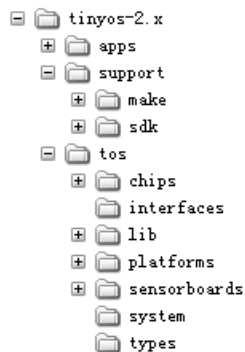


图 2-4 tinyos-2.x 目录结构

PYTHONPATH=/opt/tinyos-2.x/rupport/sdk/python。

3) tos: 主要由以下几个部分构成。

- chips: 包含对不同 ICS 的函数模块的 nc 代码, 如 microcontrollers、radios、flash memory 等。
- interfaces: 包含 TOS 核的接口。如 Boot 接口、StdControl 接口等。
- lib: 包含更多复杂的设备, 协议级支持: 如 timer, serial 等。
- platforms: 包含传感器平台的代码。这些平台有 Mica、Mica2、Micaz、TelosB 等。
- system: 包含一些保持接口实现的文件。
- types: 包含一些.h 类型的文件。

2.3 常用命令与快捷键

本节我们主要介绍在 Cygwin 环境下的常用操作命令, 这类似于 Linux 操作系统下的常用命令。其次我们简单地介绍 Cygwin 的快捷键, 以供读者了解。

2.3.1 常用命令

1. 基本命令

- pwd: 显示当前路径。
- cd: 改变当前路径, 无参数是进入对应用户的 home 目录。
- cd dir: 切换到当前目录下的 dir 目录。
- cd ..: 切换到上一级目录。
- cd ../../: 切换到上二级目录。
- cd ~: 切换到用户目录下。
- ls: 显示当前目录文件列表。
- ls -a: 显示所有文件包括隐藏文件。
- ls -l: 显示文件属性, 包括大型、日期、符号连接、是否可读/写及是否可执行。
- cp: 复制文件或目录。
- cp file target: 将文件 file 复制为 target。
- cp /root/file: 将/root 下的文件 file 复制到当前目录。
- mkdir: 建立目录。
- rmdir: 删除目录。
- rm: 删除文件。
- rm file: 删除某一个文件。
- rm -fr dir: 删除当前目录下名为 dir 的整个目录。
- mv: 文件改名或目录改名。
- mv file target: 将文件 file 更名为 target。
- diff dir1 dir2: 比较目录 1 与目录 2 的文件列表是否相同, 但不比较文件的实际内容, 不同则列出。
- diff file1 file2: 比较文件 1 与文件 2 的内容是否相同, 如果是文本格式的文件, 则将

不相同的内容显示，如果是二进制代码则只显示两个文件是不同的。

- `comm file1 file2`: 比较文件，显示两个文件不相同的内容。
- `echo message`: 显示一串字符 `message`。
- `echo "message message2"`: 显示不连续的字符串 `message`、`message2`。
- `uname`: 显示操作系统的类型。
- `whoami`: 显示登录的用户账号。
- `echo $*`: 显示环境变量的值，如 `echo $PATH`，则可显示环境路径 `PATH`。
- `ps`: 显示当前系统进程。
- `kill`: 终止某个进程。
- `help`: 显示该命令的辅助说明。例如 `ls -help`，显示 `ls` 命令的使用说明。

2. 安装 TinyOS 时使用的基本命令

在 Cygwin 中安装 TinyOS 时用到的命令与在 Linux 中安装 TinyOS 用到的命令基本相同。

- `rpm -ivh`: 安装 `rpm` 文件时使用此命令。
 - i: 表示安装指定的 RPM 包；
 - v: 表示在安装期间以 “#” 来表示安装进度；
 - h: 显示安装的详细信息。
 - `rpm -ivh --replacepkgs`: 需要重新安装 `rpm` 文件时使用此命令。
 - replacepkgs: 表示所需安装的软件包再安装一次。
 - `rpm --ignoreos -ivh`: 安装 `rpm` 文件出错时使用此命令。
 - ignoreos: 表示忽略错误强制安装。
 - `rpm -ivh -force`: 安装 `cygwin` 软件包冲突时使用此命令。
 - force: 表示当所安装的软件包与已经安装的软件包存在冲突时，使用此参数进行强制安装，但不保证所安装的软件包可以正常使用。
 - `rpm -ivh --nodeps`: 安装 `rpm` 文件遇到关联性问题时使用此命令。
 - nodeps: 表示当在安装此软件时，必须要先安装某个软件版才能正常安装，否则会出现提示信息。可以使用此参数进行强制安装。
 - `rpm -Uvh`: 升级已安装的 `rpm` 文件
 - U: 表示升级所需安装的软件包。
 - `rpm -e *`: 卸载已安装的 `rpm` 软件包。
 - e: 表示要卸载已安装的 `cygwin` 软件包。注意在删除软件包时，输入软件包名即可，加上软件包的版本号，也可以进行卸载。不可以用完整的软件包名。
- 例如，`nesc-1.3.0-1.cygwin.i386.rpm` 的卸载方式：

```
rpm -e nesc
rpm -e nesc-1.3.0
rpm -e nesc-1.3.0-1
```

后面不能加 “.cygwin”、“i386” 和 “.rpm”，所以说不可以用完整的软件包名。

- `rpm -q`: 后面直接加 `rpm` 文件名称，查看 `rpm` 文件是否安装。

- rpm -qa: 查看系统已安装的软件包。

3. 操作 TinyOS 应用程序的基本命令

- make [platform]: 将 nesC 代码编译成可在某平台运行的代码。在执行前要切换到代码所在的目录。实现此命令的前提是保证 nesC 代码没有任何语法错误。
- make [platform] reinstall: 可以将某平台的可执行代码下载到对应的平台。例如, make telosb reinstall 就可以将编译好的可在 TelosB 平台上运行的代码下载到 telosb 硬件平台。
- make [platform] install: 相当于先执行命令 make [platform], 再执行命令 make [platform] reinstall。当然, 如果执行 make [platform] 命令时, 发现程序有错误, 则不会执行下载动作, 即不会执行 make [platform] reinstall 命令。
- make [platform] sim: 在平台上仿真应用程序。
- make [platform] install.n: 不仅为平台节点编译应用程序, 同时还为该节点设置一个值为 n 的 ID 号。
- make clean: 删除 make [platform] install 等编译命令所产生的文件及文件夹。
- make micaz docs: 用图表显示应用程序中组件、接口的连接方式, 此连接方式在目录 opt/tinyos-2.x\doc\nesdoc\micaz\html 下。
- motelist: 可以查询当前设备。
- printenv MAKERULES: 检测 TinyOS build system 是否可运行。
- java TestSerial: 测试串口是否可用。
- python blink.py: 显示 blink 应用程序的结果。
- tos-check-env: 检测 Cygwin 环境是否可行。

2.3.2 快捷键

这里简单地介绍 Cygwin 环境下最基本的快捷键。

- <Ctrl+W>: 向左删除一个字, 用来删除刚刚输入错误的字。
- <Alt+D>: 向右删除一个字或删除从光标当前位置至当前字的结尾位置之间的所有字符。
- <Ctrl + U>: 删除从光标当前位置至命令行首之间的所有字符。
- <Ctrl + K>: 删除从光标当前位置至命令行尾之间的所有字符。
- <Ctrl + D>: 删除光标当前位置的字符。
- <Ctrl+A>: 光标跳转至命令行首。
- <Ctrl+E>: 光标跳转至命令行尾。
- <←>: 光标跳转至上一个字符。
- <→>: 光标跳转至下一个字符。
- <Tab>: 自动补齐命令或文件(文件夹)名。
- <↑> / <↓>: 显示前一个或后一个历史命令。
- <Ctrl+C>: 取消本次命令的执行或输入。

2.4 一个简单的应用分析

本节以 `tinys-2.x\apps\Powerup` 目录下的一个简单的应用程序为例。此例的作用开启 `Led0` 灯。通过这个例子我们可以先简单地了解 TinyOS 环境中使用 nesC 语言编写应用程序应注意的结构和细节，了解 TinyOS 下的程序执行过程。

`Powerup` 应用程序由两个组件组成：一个名为“`PowerupC.nc`”的模块和一个名为“`PowerupAppC.nc`”的配件。在一个 TinyOS 应用中，可能存在多个配件，各个配件间应有一个层次关系。所以应用中需要一个顶层配件，它定义了这个应用中的最上层组件（即 `Main` 组件）与其他组件接口的连接方式，同时也确定了这个应用所需要的最上层组件和其他组件之间的调用关系。这里，“`PowerupAppC.nc`”是 `Powerup` 应用程序的顶层配件，主要负责连接应用所依赖的其他组件，也是 nesC 编译器用来生成可执行程序文件的源文件。“`PowerupC.nc`”是提供 `Powerup` 应用程序的代码实现。典型的 TinyOS 应用程序都有一个标准的 `Makefile` 文件和 `README` 文件。`Makefile` 文件中可以设置硬件平台选项、顶层配件名称等 nesC 编译器（`ncc`）编译选项。

2.4.1 分析 `Powerup` 应用程序

- 应用配件文件：`PowerupAppC.nc`

配件文件如下：

```
configuration PowerupAppC{
}
implementation {
    components MainC, PowerupC, LedsC;
    MainC.Boot <- PowerupC;
    PowerupC -> LedsC.Leds;
}
```

配件文件分析：关键字 `configuration` 表明这是一个配件文件。前两行

```
configuration PowerupAppC{
}
```

表明这是一个名为 `PowerupAppC` 的配件。在此例中，配件 `PowerupAppC` 是顶层配件，不提供和使用任何接口。当配件不是顶层配件是，在关键字 `configuration` 后面的花括号之间可以包括如下语句：关键字 `provides` 指出配件要提供给其他组件调用的接口；关键字 `uses` 指出配件要使用的其他组件。具体的配件实现是在关键字 `implementation` 后的花括号中完成的，关键字 `components` 指出配件所用到的组件。此例中是 `MainC`，`PowerupC` 和 `LedsC`。组件间接口的联系（或称为调用）用配件实现内容中的“`—>`”、“`=`”等符号表示，“`—>`”表示位于“`—>`”左边的组件接口要调用位于“`—>`”右边的组件接口。配件文件最后两行把模块 `PowerupC` 使用的接口与其他组件提供的相应接口连接起来，即实现“连接”作用。

● 应用模块文件：PowerupC.nc

应用模块文件如下：

```
module PowerupC{
    uses interface Boot;
    uses interface Leds;
}
implementation{
    event void Boot.booted() {
        call Leds.led0On();
    }
}
```

模块文件分析：关键字 `module` 表明这是一个模块文件。模块的规范部分表明这是一个名为“PowerupC”的模块，并声明了模块所使用的接口，分别为 `Leds` 和 `Boot`。接口 `Boot` 中只有一个函数，即事件 `booted()`。当该事件到达时，表示系统中所有的组件都已经初始化完毕，系统已经启动。模块 `PowerupC` 对该事件的处理程序为调用 `Leds` 接口中的 `led0On` 命令。

2.4.2 Powerup 应用程序的图形化说明

TinyOS 系统中有一种名为“nesdoc”的文档生成方法，可自动地根据源代码产生图形化的说明文档，还能以图形方式显示整个应用程序的框架。对 `Powerup` 应用程序，在 `Cygwin` 中将目录切换到 `opt\tinyos-2.x\apps\Powerup` 下，输入 `make micaz docs` 命令，生成的文档说明位于 `opt\tinyos-2.x\doc\nesdoc\micaz\html` 目录下。打开 `Powerup AppC`，可看到如图 2-5 所示图形化说明。

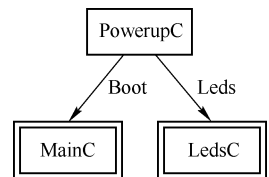


图 2-5 Powerup 应用程序的图形化说明

在 `nesdoc` 图表中，单一的矩形表示模块；双层的矩形框表示配件；虚拟的边界框表示组件是通用的，在实际使用时，需要将通用组件实例化；带阴影的椭圆表示接口；带有箭头的线条表示接口的绑定，由接口的使用者指向接口的提供者，连线上的文字表示绑定的接口。在本应用程序中，`PowerupC` 表示模块，`MainC` 和 `LedsC` 表示配件。模块 `PowerupC` 使用配件 `MainC` 中的 `Boot` 接口和配件 `LedsC` 中的 `Leds` 接口。

2.4.3 编译 Powerup 应用程序

在前面我们已经讲过，要将一个由 `nesC` 语言编写的应用程序下载到节点上，必须先把程序的 `nesC` 代码编译成可下载到平台节点上的映像文件。TinyOS 支持多种平台，每个平台在 `opt\tinyos-2.x\tos\platform` 下都有自己的目录。编译 `nesC` 代码使用命令 `make [platform]`，其中，“platform”是所使用的平台，可以是 `TelosB` 平台、`MicaZ` 平台等。本例我们使用 `MicaZ` 平台。将 `Cygwin shell` 切换到 `opt\Blink\apps\Powerup` 目录下，输入“`make micaz`”。如果没有任何的语法错误，则我们可以看到如图 2-6 所示的编译信息。

```
Administrator@23zhangyanjun1 /opt/tinyos-2.x/apps/Powerup
$ make micaz
mkdir -p build/micaz
compiling PowerupAppC to a micaz binary
ncc -o build/micaz/main.exe -Os -Wall -Wshadow -Wnesc-all -target=micaz -fnesc-
cfile=build/micaz/app.c -board=micasb -DDEFINED_TOS_AM_GROUP=0x22 --param max-in
line-insns=1000000 -DIDENT_APPNAME="PowerupAppC" -DIDENT_USERNAME="Admi
nistrator" -DIDENT_HOSTNAME="23zhangyanjun1" -DIDENT_USERHASH=0x3778f25bL -DI
DENT_TIMESTAMP=0x4e92b898L -DIDENT_UIDHASH=0x350fd01eL -fnesc-dump=wiring -fnesc
-dump='interfaces{!abstract<>}' -fnesc-dump='referenced(interfacedefs, component
s)' -fnesc-dumpfile=build/micaz/wiring-check.xml PowerupAppC.nc -lm
compiled PowerupAppC to build/micaz/main.exe
630 bytes in ROM
4 bytes in RAM
avr-objcopy --output-target=srec build/micaz/main.exe build/micaz/main.srec
avr-objcopy --output-target=ihex build/micaz/main.exe build/micaz/main.ihex
writing TOS image
```

图 2-6 编译 Powerup 应用程序

2.4.4 Powerup 应用程序中 nesC 到 C 的映射

nesC 是对 C 的扩展和修改，ncc 是 nesC 的编译器，它是 gcc 的修改和补充。ncc 先把 nesC 预编译成 C 文件，再通过交叉编译器把 C 文件编译成可执行文件。这就存在一个 nesC 语言编译成 C 语言时 nesC 中的变量、函数名和产生的 C 语言中的变量、函数名的对应关系，即 nesC 到 C 的映射。在 Powerup 中这种映射就是指 Powerup 的接口、组件、C 文件中的变量以及函数名与 ncc 编译成 app.c 对应后的变量、函数名之间的映射。

应用程序中用到的 C 文件中包含的类型、变量、函数名在 ncc 处理过程中保持不变。如果 C 文件出现了与 nesC 的关键字相同的变量，则在编译时自动在该变量前加上前缀 `_nesC_keyword_`。具体的映射关系如下。

- 模块 M 中的变量 X 映射成：M\$X。
- 模块 M 中的函数 F 映射成：M\$F。
- 模块 M 中的命令或事件 C 映射成：M\$C。
- 模块 M 中的接口 I 中的命令或事件 C 映射成：M\$I\$C。

此外，nesC 使用 ncc（确切地说是 gcc）的一些属性来声明一些函数和变量的属性。其中，`spontaneous` 属性表示该函数可以在其他文件中被访问，而不是仅仅限于本文件。如果函数没有声明 `spontaneous` 属性，则 ncc 在编译成 C 语言时，就在函数前加上 `static`。具体的属性在后续章节介绍。

对于本例 Powerup 应用程序，编译此应用程序时，会在 `opt/tinyos-2.x/apps/Powerup/build/micaz` 目录下生成一个 `app.c` 文件。这就是 ncc 把 nesC 预编译成的 C 文件。在 `app.c` 中，我们可看到如下映射：

```
static inline void PowerupC$Boot$booted(void )
{
    PowerupC$Leds$ledOn();
}
```

2.4.5 仿真 Powerup 应用程序

TOSSIM 是 TinyOS 操作系统自带的一种仿真器，它可以仿真完整的 TinyOS 应用程序，

它是 TinyOS 系统的一个程序库，其核心代码位于 `tos/lib/tossim` 目录。TOSSIM 唯一支持的平台是 MicaZ 平台。由于 TOSSIM 运行和传感器硬件相同的代码，所以仿真编译器能直接从 TinyOS 应用的组件表编译仿真程序。对应用程序进行仿真时，需编写一个 .py 文件。如对本例进行仿真时，首先进入目录 `\cygwin\opt\tinyos-2.x\apps\Powerup`。然后编写文件 `C:\cygwin\opt\tinyos-2.x\apps\Powerup\Powerup.py`。具体如何仿真应用程序将会在第 9 章做详细介绍。最后在命令行输入仿真编译命令：

```
$ make micaz sim
```

将会看到很长的编译信息，最后会显示模拟结果如下：

```
***Successfully built micaz TOSSIM library.
```

若出现如上的显示结果，则可用命令 `$ python Powerup.py`（或 `./Powerup.py`）显示仿真结果。

2.4.6 下载应用程序

1. 安装 USB 串口驱动

TinyOS 开发环境支持多种不同的编程器。USB 是常用的程序烧录端口，有不少编程器会把节点上的串口转换为 USB，以方便与 PC 的连接。

Mica 和 TelosB 平台使用 FTDI 公司的串口转 USB 芯片 FT232B，该芯片的驱动程序可以到 FTDI 公司网站下载。这里，我们所用的操作系统是 Windows XP，故选择 XP 版本的驱动。

第一次使用硬件节点时，需要安装驱动程序。将 Mica 系统编程器 MIB520 插入到 PC 的 USB 接口，如果找不到驱动程序，则 Windows 会弹出“找到新的硬件向导”对话框。选择“从列表或指定位置安装”按钮，然后单击“下一步”按钮。

选择“在搜索中包括这个位置”复选框，然后单击“浏览”按钮，选择驱动程序所在目录，单击“下一步”按钮，Windows 就会安装该驱动程序。安装完成后，硬件节点就变成一个 USB 串口设备。此时，可以检查设备管理器里是否有“USB Serial Port”，若有，则说明安装成功；若没有，则可重试此过程。

2. 下载 Powerup 应用程序

编译完成后，可以把编译的二进制映像文件下载到节点上。这里我们主要以 MicaZ 平台和 Telos 平台为例来介绍在 Windows 系统中下载程序的方法。

1) 在 MicaZ 平台上下载程序时，可以使用与平台配套的串口编程板 MIB510。关闭节点的电池供电，将编程板的两端分别连接到节点串口和计算机 USB 口上，然后，使用命令 `cd` 切换到 Powerup 应用程序所在路径下，并输入如下命令：

```
$ make micaz reinstall.<addr>MIB410,<serialport>
```

命令的功能如下。

- **reinstall**: 告诉编译系统下载当前已编译的二进制映像文件到节点的微控制器芯片上。此命令与 `install` 命令的区别在于前者没有应用程序的编译过程，只设置节点

地址，并下载程序；后者为目标平台编译程序，设置节点地址以及下载程序到节点上。

- **<addr>**: 分配给该硬件节点在整个网络中的身份标识序号，即节点在整个网络中的地址。这里需要注意的是，TOS_BCAST_ADDR (0xFFFF) 和 TOS_UART_ADDR (0x007E) 两个地址是保留地址，不可使用。
- **<serialport>**: 串口设备的名字，也就是编程板与 PC 连接的串口端口。这里需要注意的是，在 Windows 系统中节点通过编程板连接到串口 COMn，在 Cygwin 里需使用 /dev/ttySn-1 作为程序下载的串口。

2) 在 Telos 平台上下载程序时，由于 Telos 系列节点是 USB 类型的设备，可以直接插入到 USB 端口。当 Telos 节点直接插入到 PC 上时，PC 的操作系统就会发现。输入“motelist”命令则会显示节点当前插入的 COM 端口号。此时所显示的 COM 端口号必须减 1 后作为程序下载的端口命令参数。

当然，不同类型的硬件平台，其程序下载命令可能有所不同，具体的下载方法请参考节点平台的说明手册。

第 3 章 TinyOS 编程语言 nesC

本章着重介绍 TinyOS 的编程语言 nesC。

在具体介绍 nesC 之前，先以 C 及 C++ 作为参照物，讨论 C (C++) 与 nesC 之间的不同，这将有利于读者尤其是 nesC 的初学者更好地理解 nesC 的语法、用法。

nesC (Network Embedded Systems C) 采用了特有的基于组件 (Component) 的编程模型，即通过连接组件来创建应用。实际上，可以将 nesC 程序看做若干组件的集合。组件是一个个单独的代码块，为输入和 (/或) 输出定义了明确的接口 (interface)。接口是一组被称为命令 (command) 和 (/或) 事件 (event) 的函数集合，是组件的访问点。接口是双向的，是组件间相互联系的通道。组件实现的功能或函数必须在其声明的接口中说明，否则不能被其他组件使用。组件可以分为两种类型：模块 (Module) 和配件 (Configuration)。模块负责定义状态以及可执行逻辑，配件通过接口将组件连接起来，形成一个新的抽象 (也可以称为服务) 或者一个完整的顶层应用。

本章按照从简单到复杂的顺序，依次介绍组件、接口、模块实现、配件实现、参数化接口与通用化组件等编程知识。

3.1 C 与 nesC 的比较

nesC 与 C 之间的本质区别在于程序的连接 (linking) 模型。C 程序由变量、函数和类型等程序元素组成。这些程序元素在文件中定义，分别被编译并连接到一起构建成应用。nesC 程序由组件组成，这些组件通过连接语句显式地组织起来。nesC 编译器将这些组件编译并连接成为一个整体 (即一个应用)。站在编程者的角度来看，nesC 编程的困难与复杂性不在于如何写组件，而在于如何把组件组织成一个可以工作的应用。

本节试图从编程者的角度出发，先介绍 C (C++) 程序如何组织，再进一步讨论 nesC 程序如何组织及原因。

3.1.1 C 与 C++

C 程序元素包括变量、类型和函数。为了叙述方便，这里大多时候将函数与变量统称为变量。在一些具体情况下，再明确指出是变量还是函数。

C 程序命名变量有三种方式：声明 (declaration)，定义 (definition) 和引用 (reference)。

声明变量是告知编译器变量的名称和类型。例如，声明函数是把函数名、函数参数的个数和参数类型等信息通知编译系统，以便在调用函数时，编译系统能够正确识别函数并检查调用是否合适。只要保持一致，可以多次声明变量。声明变量不是实现，仅表明变量存在，其他变量可以使用它。

引用变量包括调用函数、赋值及取地址等。通常情况下，C 编译器要求声明变量之后才

能引用它。使用未声明的变量是不良的 C 编程习惯。

定义变量是最后一种变量命名方式。声明变量表明变量是存在的，引用变量是使用变量，而定义变量实际上是创建变量。具体地说，定义函数是实现函数，定义变量是为变量分配存储区域。变量可以被声明多次，引用多次，但只能定义一次。

定义变量将变量名字引入到了程序的命名范围。C 程序有多个命名范围，可以将其看成一个范围树，在最顶层的根是全局范围。任何代码都可以引用全局范围内的变量。全局变量和非静态（non-static）函数都在全局范围内。大括号{ }定义了子范围，该范围内的代码只能引用树中该范围以上范围内命名的变量。另外，还有一个变量范围级别是文件范围，由 static 关键字来声明，也就是说，带有关键字 static 标识的变量只能被同文件内的代码引用，不能在全局范围内被引用。

如果一个 C 程序由多个 C 文件构成，为了引用别的文件中的变量，需要使用声明变量的方式，将所要使用的变量名字引到全局范围。在下面的例 3.1 中，文件 f2.c 定义了函数 g()，在文件 f1.c 中要使用这个函数，需要使用语句 extern void g()先声明它。

例 3.1:

f1.c 包含的 C 代码如下：

```
extern void g()
int main () {
    g();g();
}
```

f2.c 包含的代码如下：

```
void g() {
    printf("hello world!");
}
```

在全局范围内组织变量要采用诸如头文件和命名约定等技巧。头文件也是一种 C 源文件。它将变量声明组织在一起，这些声明与相应实现文件或目标文件的定义相匹配。使用头文件可以避免大量重复输入可能带来的差错。命名约定可以在一定程度上避免使用不同的符号为相同的变量命名。例如，类型通常有_t 的前缀，保证类型和函数不会有相同的名字。有些库在其变量前使用相同的前缀，如 gkt_用在 GKT+图形工具的库中。这些前缀提醒用户避免与其他库中的符号名字冲突。但是这样的做法使得编程显得繁冗。

如果某个目标文件引用一个变量，在连接阶段，就会被连接到定义这个变量的目标文件（可能是库文件或标准的目标文件）。以函数为例，由于声明函数仅对应一个函数定义，这表明代码引用一个函数，实际是引用了它的实现。如果两个源文件引用相同的函数名字，则实际上引用的是同一个函数，这就在两块代码间引入了潜在的依赖性。因为如果想改变一个文件使用的函数实现，那么也会改变另一个源文件代码使用的函数实现。这种情况显然不是我们想要看到的。函数指针是避免这种间接绑定的一个常用办法。在代码中不引用特定的函数，而是引用一个存放函数指针的变量，这个指针可以指向任意一个函数。这就使得代码可以在运行时再解析绑定，也就是说，通过选择在变量中存放什么函数地址值（指针）确定绑定。通过使用函数指针，C 程序可以调用那些在编译时还没有或不能命名的代码块。这对于

具有回调（callback）的系统是非常重要的。

例如，GUI 工具箱需要调用函数对用户事件进行反应，函数是应用代码的一部分，但是在 GUI 工具箱预编译时需要调用它。具体地说，应用创建一个按钮，给这个按钮一个函数指针，当按钮被单击时，调用函数指针所指的函数。C 中没有其他方式可以高效地做到这一点，因为工具箱不能命名这个函数。所以必须使用函数指针，而该指针必须在运行时分配。

UNIX 或 Linux 内核使用的虚拟文件系统也是一个典型的例子。操作系统具有访问任意数目的文件系统的需求，将所有文件系统静态编译到内核是很麻烦的。如果为了支持一个新的文件系统，如 USB 存储设备，就需要重新编译操作系统，这很烦琐，是不应该的。为了避免这一点，操作系统使用虚拟文件系统（Virtual file system, VFS）接口与文件系统进行交互。这是一套基本的操作，比如 `read()` 和 `write()` 等，存放在一个数据结构中。每个文件系统实现都使用自己的函数来实现这个数据结构，然后把这个数据结构传给内核。比如，当用户程序要读 CD 时，OS 读取与 CD 文件相关的 VFS 结构，调用 `read()` 函数指针，从而使得内核与特定的文件系统保持独立。这个基本方法与 GUI 工具箱相同。总之，正是由于 C 的命名方法，操作系统必须使用运行时分配的函数指针来达到灵活的程序连接与扩展。

C++除了具有更为丰富的命名范围和继承性之外，其他与 C 相似。C++使用双冒号::表明命名范围的层次性。虽然类可以在其域和方法上使用不同的访问描述符，但这与 C 中关键字 `static` 声明的变量不同，它不是在命名范围的级别上操作。`static` 表明变量和函数不会出现在全局范围，因此也不会在全局范围内被命名，其他文件想要访问这样的变量，系统会报“no such variable exists”的错误。相对的，C++类中使用 `private` 指定变量为私有的，意味着如果其他代码引用私有变量，就会产生一个“access violation”的编译错误。

C++的继承性提供了比 C 更容易扩展功能的方式，程序使用类而不是函数指针来表示可扩展抽象。实践中，每个对象都有一个指向自己的函数表的指针。该函数表是只读的，不需要程序员维护它。基于 C++的系统使用对象而非函数指针来扩展功能。与 C 一样，在运行时建立变量名字之间的绑定关系。

C++中，变量名字的绑定使得工厂函数（Factory）成为软件工程实践中的通用对象。工程将对象与其初始化分开。比如，Tetris 游戏可能需要创建游戏片段。一种简单的方式是让游戏引擎直接分配各个片段。将游戏引擎绑定到一个具体类比如 `JPiece`，也同时绑定了 `JPiece` 构造函数的参数列表。这种绑定是不必要的，而且，如果更改类的名字，比如改成 `JTetrisPiece`，会产生问题。如果不将游戏引擎绑定到一组特定类的集合，而是定义一个抽象的类 `Piece`，仅仅要求一个工程来为它创建类，这样，游戏引擎与 `Piece` 的名字及实现就相互独立了。也就是说，改变一组 `Piece` 仅仅要求改变工程，因为仅有一个分配点，构造函数的变化都可以局部到工程中来。工程的设计模式使得命名具有一定程度的间接性，从而打断了那些不想要的特定类之间的依赖关系。

3.1.2 nesC

C 与 nesC 之间最本质的区别是如何组织程序。下面以两个简单应用为例，结合其 C 实现和 nesC 实现进行讨论。

1. 一个例子的 C 实现与 nesC 实现

例 3.2: Powerup 应用

C 实现代码:

```
#include "mote.h"
main()
{
    mote_init();
    Led0_on();
    sleep();
}
```

Powerup 应用的 C 实现被编译并连接到一个“mote”库。该库提供了硬件初始化、控制二极管以及设置节点低功耗状态等函数。节点启动后，main 函数自动被调用。

Powerup 应用的 nesC 实现代码见 2.4.1 节，主要分为两个组成部分：一是模块 PowerupC，包含应用 Powerup 的执行逻辑；二是配件 PowerupAppC。

在模块 PowerupC 的代码中，可以看出模块 PowerupC 基于两个接口 Boot 和 Leds 与系统的其他部分（服务）交互。配件 PowerupAppC 指定了模块 PowerupC 如何连接到 TinyOS 的服务，显式地指定了接口之间的连接关系。MainC 启动系统，它通知 Boot 接口的 booted 事件，在配件中将 Boot 接口连接到 PowerupC，因此该事件被通知到组件 PowerupC 中。在这个事件的实现中调用了接口 Leds 的 led0On 命令，因此就可以打开 LED0。组件组织结构可以由文档生成工具 nesdoc 生成，具体见 2.4.2 节

2. nesC 与 C 的主要区别

由上述应用 Powerup 的 C 实现版本与 nesC 实现版本可以得出 C 与 nesC 的不同。

1) C 程序由函数组成；相对的，nesC 程序由组件组成，组件实现特定的服务。

2) C 函数通过直接彼此调用来进行交互；组件之间交互由接口指定。接口的使用者对接口的提供者提出请求，提供者回调（通知事件）接口的使用者。接口是一组相关函数的集合，其中的命令与事件类似常规函数，包含标准的 C 代码。调用命令与通知事件就是函数调用。PowerupC 是 Boot 和 Leds 的使用者，启动事件是一个回调函数，通知系统启动。Led0On 是命令，请求打开 LED0。nesC 接口与 Java 的接口类似，添加了关键字 command 和 event 来区分请求与回调。

3) PowerupAppC 是 C 与 nesC 之间最大的不同。换句话说，nesC 程序使用连接显式地指定组件之间如何交互。C 程序中调用的是一个全局名字 Led0_on。而 nesC 程序中由编程者来显示选择使用哪个组件的函数实现，比如，组件 PowerupAppC 中的连接语句显式地指定了 PowerupC 的 Leds 接口连接到组件 LedsC 上。将组件之间的连接交给编程者决定可以带来 2 个好处。

- 在编程过程中往往需要切换调用的功能模块，由编程者来决定组件之间的连接关系，会简化具体实现代码。

我们尝试改变下面两行代码：

```
Components PowerupC, NoLedsC;
PowerupC.LedsC->NoLedsC.Leds;
```

这种更改将连接切换到组件 NoLedsC 上，没有影响 nesC 程序的其他部分。但是，对于

C 的实现代码而言，如果使用别的版本替换掉 Powerup 中使用的 mote 库，则会影响所有的 LED 用户，不仅仅是应用 Powerup 自己。

● 提供了一个支持回调的有效机制

考虑计时器的例子。在计时器到时时，要回调与该计时器相关的触发函数。C 实现中为了从计时器函数回到主调函数，需要使用函数传递运行时的参数。在 nesC 实现中，计时器组件与 PowerupC 组件之间的连接是在编译时指定的，因此不需要函数指针，既节省了 RAM，也允许 nesC 编译器执行跨组件的回调函数的优化。

总而言之，nesC 与 C 的主要不同在于组件、接口和连接。这些都与程序元素（变量、函数与类型等）的命名和组织有关。

C 与 C++ 的全局命名范围需要动态组织程序。对于函数调用，连接程序的工作就是对应上全局命名范围中那个唯一的函数名字。如果多个程序块引用同一个函数，在这些程序块之间会引入潜在的依赖性。针对这个问题，C 和 C++ 分别使用了函数指针或工程等间接方式，将函数的不同实现区分。

nesC 中采用了不同的方式——静态连接。nesC 是基于组件的 C 语言。组件是一段段具有一定功能的代码块。在某些方面，nesC 组件与对象相似，比如，封装状态、将状态与功能合到一起。它们的主要区别在于命名范围，C++ 对象引用在全局命名范围的函数和变量；而 nesC 的组件名字是全局的，但组件内部的操作是局部的。组件仅能引用自己局部范围（组件范围）内的函数和变量，不能引用别的组件范围内的函数和变量。为此，组件不仅要声明它提供（实现的可供其他组件使用）的函数，也要声明它要使用的函数。由于组件用来调用函数的名字是完全局部的，它引用函数使用的名字不必与实现函数的名字相同。组件 A 声明提供函数 b，实际上是将 A.b 引入到全局命名范围，另一个组件 C 提供函数 b 即将 C.b 引入到全局命名范围。也就是说，即便某个组件 D 想要引用函数 b，也可能引用完全不同的实现。

表 3-1 中总结了 C、C++ 与 nesC 在组织程序方面的不同。

表 3-1 C、C++ 与 nesC 的比较

结构元素	C	C++	nesC
编程单元	文件	类	组件
编程单元规范	头文件	类声明	组件规范
编程规范模式	—	抽象类	接口
编程单元组织	名字匹配	名字匹配	连接
延迟组织	函数指针	虚拟方法	连接

C 语言中的编程单位是文件，由相关头文件来规范文件行为。连接程序通过匹配文件内的全局元素名字来构建应用。用这种连接方式来构建程序是不够的，因此，编程者还需使用函数指针的方式支持延迟调用的函数。

C++ 提供了组织程序的显示机制。编程单位是类，类将相关函数组成组，程序由一组交互的对象（类的实例）组成。抽象类用于定义通用的类模式，类从抽象类中继承并实现它的方法。连接程序通过匹配类和函数名字来构建应用。相对于 C 中的函数指针，虚拟方法提供了更加便利和结构化的方式。

nesC 程序由相互协作的组件组成。每个组件使用接口来指定它提供和使用的服务。编程者编写连接语句来连接组件以构建应用。由于使用显式连接语句，不依赖于隐式的名字匹配，因此，不需要使用如函数指针、虚拟方法的动态机制。

3. nesC 编程模式成因的简要分析

实际上，编写 nesC 程序主要包括两个部分：编写组件，连接函数的使用者与提供者。

每个组件有一个规范（说明），它是一个代码块，用来声明它提供的函数和使用的函数。对于提供的函数，组件必须定义这个函数，其他组件可以调用它。相反的，组件要调用某个函数，需要由其他组件来定义这个函数。组件可以调用自己提供（定义）的函数。

组件通常由一组接口（Leds、Boot、SplitControl 和 AMSend 等）来指定。实际编程中，在组件规范中很少声明单独的函数，更多是使用接口（interfaces）。接口是一组相关函数的集合。例如，管理和配置方面的应用经常需要启动和停止系统抽象（也就是某种系统服务），如打开传感器读取数据，打开无线协议栈接收分组等。StdControl 接口是表示这种功能的常用方式。被打开和关闭的抽象或服务的组件要提供 stdControl，同时，需要打开或关闭的其他服务或抽象的组件要使用 stdControl 接口。接口简化了代码，并且使得代码更加清晰。因为组件之间的相互操作遵循标准的模式。

将提供者与使用者连接起来称为“wiring”。例如，PowerupC 中的代码具有调用 Leds.start() 的函数的内容，如果 Leds 不连接到提供者，这些函数就是未定义的符号，即没有绑定到任何实际的代码上。如果 Leds 被连接到 LedsC 上，那么当 PoewerupC 调用 Leds.start() 时，就调用了 LedsC 中的 Leds.start()。PoewerupC 和 LedsC 是完全分离的，仅当在连接了的时候才绑定起来。这种连接在编译时进行，不需要运行时分配或在 RAM 中存储函数指针。而且，由于没有间接引用方式，nesC 编译器清楚完整的函数调用关系可以进行一些跨越调用边界的优化。

为什么在 TinyOS 或 nesC 中采用这种编程模式呢？

这主要是由 TinyOS 面向应用的网络节点的特点与需求所决定的。TinyOS 主要是面向无线传感网等低功率网络节点而设计的。这类网络节点与我们熟悉的功能强大、应用繁杂的互联网端用户节点存在很大不同。端节点上存在大量与用户即时交互的应用，比如使用浏览器上网，使用 word 编辑文档，使用邮件客户端下载邮件，使用 QQ 与朋友聊天等等，这些使得用户与节点的交互操作极其频繁。另外，由于商业等多方面的因素，节点上运行的软件程序功能也需要动态的维护更新。

相对的，无线传感网一般具有如下几个特点。

- 1) 一般面向监测、感知等方面的应用，网络节点功能相对特定。通常工作在无人看管的状态下，节点周期性或触发性地向控制中心发回感知数据，控制中心主动与节点的交互非常少。
- 2) 网络通常布设在远端，网络功能在一段较长的时间内不会发生变化，也就是说，网络节点功能的更新周期很长，通常会几个月或是几年。
- 3) 网络节点能力微小，包括处理能力、存储能力以及通信能力等。

由此可见，该类节点上的软件程序采用静态组织的方法，在满足节点能力微小限制的同时，又可以满足网络节点的应用需求。

3.2 组件定义

每个组件都有自己的源文件，组件与其源文件名字之间是一一映射的。例如，文件 `LedsC.nc` 包含组件 `LedsC` 的 `nesC` 代码，而组件 `PowerupC` 的 `nesC` 代码包含在文件 `PowerupC.nc` 中。组件名字位于全局范围，也就是说，每个组件只有一个定义，相应的，`nesC` 编译器只加载一个名字为 `LedsC.nc` 的文件。

组件分为配件和模块两类。

<pre> module 模块名{ //组件规范 } implementation{ // 组件实现 } </pre>	<pre> configuration 配件名{ //组件规范 } implementation{ // 组件实现 } </pre>
---	--

无论模块还是配件，其定义都包括两个组成部分：规范与实现，如上所示。模块使用关键字 `module` 来声明。配件使用关键字 `configuration` 来声明。关键字 `module` 或 `configuration` 后面跟着模块或配件的名字。组件名字后加上“.nc”的扩展名即是存放该组件的源文件名字。组件名后面大括号中的语句描述了该组件的规范。关键字 `implementation` 后面大括号中的语句是该组件的实现代码。

TinyOS 代码有一个非强制性的编码约定：组件名字皆以 `C` 或 `P` 结尾。`C` 代表“Common”，即共有的，表明该组件是可以被其他组件自由使用的。`P` 代表“Private”，即私有的，表明在该组件所在目录之外的组件不能使用它，但一般会在所在目录下有一个以 `C` 结尾的组件来封装它，以便其他组件使用。例如，对于组件 `LedsC`，任意组件都可以使用它；相对的，其组件 `LedsP` 只有与同目录下的文件才能使用它。`LedsP` 被 `LedsC` 封装后，便可以供所有其他组件使用。

模块和配件的规范部分具有相同的语法，可以声明一个或多个接口，也可以为空（不声明接口）。模块与配件的主要区别在于实现部分。模块实现部分由类似 `C` 的 `nesC` 代码组成。模块代码包括声明变量和函数以及调用函数等。配件实现部分由 `nesC` 连接（`wiring`）代码组成。这些连接代码将组件组织起来。这是一种新的编程模式，也使得配件成为 `nesC` 与 `C` 的主要区别。将模块和配件区别开来的主要原因在于允许用户在系统现有实现代码基础上构建自己的应用程序。比如，通过配件把一个或多个组件封装起来，其中没有一个是自己设计的。其他开发者也可提供一组新的组件库，供应用程序开发者使用。读者将在 `nesC` 编程实践的过程中逐渐理解这一点。

模块与配件的实现分别在 3.4 节与 3.5 节中着重介绍。本节主要介绍组件的规范部分。

3.2.1 组件规范

大多数情况下，组件需要声明自己提供的服务（或功能）和使用的服务（或功能）。这些通过在组件规范中声明提供接口（`provides interface`）或使用接口（`uses interface`）的 `nesC` 语句来实现。

- 使用关键字 **provides** 声明组件提供的接口，表明组件能够提供的服务。
- 使用关键字 **uses** 声明组件使用的接口，表明组件需要使用的服务。

接口是为某个服务或抽象声明一组相关的函数。例如，为了控制节点上的二极管(LED)，定义接口 **Leds** 来声明关闭、打开二极管等相关函数；为了通知应用节点已经启动起来，定义接口 **Boot** 来声明一个通知函数 (**booted**)。接口相关说明后面将会详述。

例 3.3:

```
1): module PowerupC{
    uses interface Boot;
    uses interface Leds;
}
2): configuration Leds{
    provides interface Leds;
}
```

在例 3.3 中，模块 **PowerupC** 的规范中包含了 2 条语句，关键字 **uses** 表明接口 **Boot** 和接口 **Leds** 是两个使用的接口。**PowerupC** 使用 **Boot** 接口来接收节点启动的通知事件，使用 **Leds** 接口来打开节点的二极管。相对的，配件 **LedsC** 提供了一个接口 **Leds**，用于控制 3 个二极管的抽象。

组件规范中可以既提供接口也使用接口。在例 3.4 中，配件 **MainC** 的规范部分提供接口 **Boot**，使用接口 **Init**。配件 **MainC** 实现了节点的启动顺序。节点启动顺序中的一项主要工作是 **MainC** 使用 **Init** 接口来初始化软件。例如某模块中有一个状态需要在系统启动前进行初始化，那么该模块需要提供 **Init** 接口。当节点启动起来时，**MainC** 通过 **Boot** 接口中的事件 **booted** 通知其他组件，如模块 **PowerupC**。

例 3.4:

```
configuration MainC{
    provides interface Boot;
    uses interface Init as SoftwareInit;
}
```

组件的规范部分可以为空，表明组件是顶层应用配件，如 **PowerupAppC**。

3.2.2 关键字 as

细心的读者会发现，上面 **MainC** 配件规范部分的第二条语句中引入了一个新的关键字 **as**。关键字 **as** 的作用是为接口定义一个别名。这里是为接口 **Init** 定义了一个别名 **SoftwareInit**，表明使用该接口对软件进行初始化。从这个角度来看，使用 **as** 可以提高程序的可读性。

除了上述情况外，还有一种情况要求在组件规范中必须使用关键字 **as**。

在组件规范中可能需要多次提供或使用同一接口，此时必须使用关键字 **as** 来区分同一类型接口的多个实例。例如，配件 **LedsC** 通过 **Leds** 接口提供了三个二极管的抽象。**LedsC** 是一个配件，没有可执行代码，它将二极管模块 **LedsP** 连接到更为底层的能够激励二极管的组件上。模块 **LedsP** 的规范见例 3.5，通过使用关键字 **as**，为接口 **GeneralIO** 的三个实例定

义了不同的别名。总之，在组件规范中，必须确保每个接口实例都具有唯一的名字，才能保证配件实现组件连接的过程中不至于产生混乱。在模块 LedsP 后的 “@” 代表属性，详见 3.7.5 小节。

例 3.5: 模块 LedsP 的规范

```
module LedsP @safe(){
  provides{
    interface Init;
    interface Leds;
  }
  uses{
    interface GeneralIO as Led0;
    interface GeneralIO as Led1;
    interface GeneralIO as Led2;
  }
}
```

事实上，接口声明中已经隐性地使用了 as。下述两条 nesC 语句中，第一条语句实际上是第二条语句的缩写。

```
uses interface Leds;
uses interface Leds as Leds;
```

关键字 as 不仅可以应用于组件规范中为接口设定别名，还可以应用于配件实现部分，为声明的组件设定别名。他们的作用类似于变量名字和函数名字。有效使用关键字 as，会大大提高程序代码的可读性。

3.2.3 将接口分成组

如果组件规范中提供多个接口或者使用多个接口，可以将提供或者使用的多个接口分成组。上面提到的模块 LedsP 是一个典型的例子，使用大括号将提供的 Init 接口和 Leds 接口括起来，并用关键字 provides 标识，将使用的 Led0、Led1 和 Led2 接口括起来，用关键字 uses 标识。这只是一种编程风格而已，目的是使得代码更加清晰易读。也就是说，例 3.6 中的两个版本是等价的，这两种写法在语法和代码执行效率方面没有差异。

例 3.6:

```
module LedsP @safe(){
  provides interface Init;
  provides interface Leds;
  uses interface GeneralIO as Led0;
  uses interface GeneralIO as Led1;
  uses interface GeneralIO as Led2;
}
```

另外，组件规范中通常会先声明提供的接口，再声明使用的接口。这样便于读者了解组件的可用服务和所依赖的服务（需要别的组件提供的服务）。但是在某些具有多个功能的较

为复杂的组件中，就不必遵循这种风格了。相反的，将功能相关的接口放到一块，往往使得代码更具有可读性。

3.3 接口

接口用于描述组件之间在功能方面的相互关系。在这种相互关系中，组件扮演的角色依赖于它是提供接口还是使用接口。与组件类似，接口名字与其源文件名字之间也是一一映射的关系。例如，文件 `Boot.nc` 中描述的是接口 `Boot`，接口 `Init` 包含在文件 `Init.nc` 中。接口名字位于全局范围。

3.3.1 一般接口

在语法上，接口与组件完全不同，它只有一个组成部分，其中包括一个或多个函数。接口的定义如下所示，关键字 `interface` 用来表明这是一个接口定义。接口中的函数可以分为 2 类：命令和事件。关键字 `command` 和 `event` 分别标识它后面的函数是命令和事件。由此看来，接口实际上是事件和（/或）命令的集合。根据具体设计情况，接口中的命令或事件可以是一个或多个，也可以只有命令或事件。

```
interface 接口名{
    command 类型标识符 函数名（形式参数列表）；
    ...
    event 类型标识符 函数名（形式参数列表）；
    ...
}
```

接口 `Init` 和接口 `Boot` 的定义如下所示。这两个接口很简单，接口 `Init` 中只有一个命令 `init`，接口 `Boot` 只有一个事件 `Booted`。

例 3.7: 接口 `Init` 和接口 `Boot`

```
1): interface Init{
    command error_t init();
}
2): interface Boot{
    event void booted();
}
```

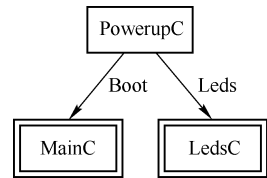
为了便于描述，将在规范中具有提供某个接口语句的组件称为该接口的提供者，具有使用某个接口语句的组件称为该接口的使用者。

接口的使用者调用接口中的命令，接口的提供者调用接口中的事件。相反的，由使用者实现或定义事件，由提供者实现或定义命令。在应用 `Powerup` 中，组件 `PowerupC` 是接口 `Boot` 的使用者，组件 `MainC` 是接口 `Boot` 的提供者。当系统启动起来后，由组件 `MainC` 通过接口 `Boot` 通知该系统已启动的事件。这里的通知事件实际上就是函数调用，即由组件 `MainC`（实际上是由 `MainC` 封装的模块 `RealMainP`）调用组件 `PowerupC` 所使用接口 `Boot` 中的 `Boot.booted()` 函数，实现通知事件的目的。实现事件的工作在 `PowerC` 中进行。相对的，

对于命令的实现与调用，与事件正好相反。组件 PowerupC 使用 Leds 接口（定义如下所示），该接口由 LedsC 提供。PowerupC 调用组件 LedsC（实际是模块 LedsP）中实现的命令 Leds.Leds0on()。

读者读到这里，也许会有些模糊。不用着急，阅读后续对模块和配件实现的介绍后，就会更加清楚了。

Leds 接口代码如下：



```

interface Leds{
    async command void led0On();
    async command void led0Off();
    async command void led0Toggle();
    ...
    async command uint8_t get();
    async command void set(uint8_t val);
}
  
```

前面接触的几个接口，要么只有命令，要么只有事件。还有一些接口中既包含命令也包含事件。例如，下一节将要介绍的分阶段接口就同时包含命令和事件。

3.3.2 分阶段操作

通常来说，硬件是分阶段（split-phase）操作的，即软件对硬件提出请求，硬件完成操作后发送信号（比如中断）给软件系统。在硬件中断时间较长的情况下，如果软件提出请求后一直处于阻塞状态，CPU 和能量都会产生较大浪费。对于这个问题，传统的解决方法是使用多个线程（thread）等待硬件信号。某应用需要使用硬件时，操作系统建立请求，将线程放入等待队列，然后调度其他线程工作。当硬件中断到达，再恢复等待线程，并将其放在就绪队列中。由于每个线程都具有自己的堆栈，嵌入式系统中使用线程会带来较大的 RAM 开销。对于 RAM 资源非常有限的无线传感器网络节点等来说，等待状态的线程造成了堆栈空间的严重浪费。而且在系统中分配合适大小的堆栈也是非常棘手的问题。

不同于传统的解决办法，TinyOS 采用的方法是软件也使用分阶段操作，并使用分阶段接口来达到这个目的。分阶段接口是双向的，具有向下的函数调用（命令）来启动操作，具有向上的函数调用（事件）来通知操作完成。接口 Read 就是一个基本的分阶段接口，大部分传感器驱动程序都提供该接口。接口 Read 中包括一个命令 read() 和一个事件 readDone()。当调用者调用命令 Read.read()，接口 Read 的提供者直接返回成功，并在将来读取到传感器数据后，再调用 Read.readDone() 来通知 Read 接口的使用者。Read 接口代码如下：

```

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

下面结合模块 PeriodicReaderC 来说明分阶段接口 Read 的应用。在模块 PeriodicReaderC 的实现代码中，接口函数 StdControl.start() 启动了计时器。计时器时间到时，Timer.fired() 被

通知，该事件的处理程序调用 `Read.read()` 并返回（这是分阶段操作的第一阶段）。`Read.read()` 由负责读取传感器数据的组件定义并提供。之后的某个时刻，读取传感器数据的组件通知 `Read.readDone()` 事件，实参 `val` 为传感器数据。`PeriodicReaderC` 对于该事件的处理操作是将采集的传感器数据存储在模块变量 `lastVal` 中（对事件进行处理，是分阶段操作的第二阶段）。

这种同时包含命令和事件的接口也称为双向接口。双向接口使得调用者不需要使用函数指针，就能够注册回调函数。`PeriodicReaderC` 模块代码如下：

```
module PeriodicReaderC{
    provides interface StdControl;
    uses interface Timer<TMilli>;
    uses interface Read<uint16_t>;
}
implementation{
    //模块代码实现开始
    uint16_t lastVal=0;           //定义局部变量
    command error_t StdControl.start(){ //定义 StdControl 接口中的 start 命令
        return call Timer.startPeriodic(1024);
    }
    command error_t StdControl.stop(){ //定义 StdControl 接口中的 stop 命令
        return call Timer.startPeriodic(1024);
    }
    event void Timer.fired(){ //定义 Timer 接口中的 fired 是将处理程序
        call Read.read();
    }
    event void Read.readDone(error_t err, uint16_t val){ //定义接口 Read 的事件 readDone 的处理程序
        if(err==SUCCESS)
            lastVal=val;
    }
}
//implementation 代码结束
```

3.3.3 通用接口

前一小节中介绍的接口 `Boot`、`Init` 和 `Leds` 都是无类型的。本小节介绍一类新的将一个或多个类型作为参数的接口，如上面提到的接口 `Read`，这类接口称为通用接口。接口的类型参数用尖括号括起来。接口定义中的事件和命令可以引用接口的参数，具体地说，可以在命令或事件的形参或者函数（指命令或事件）返回值类型等处使用。例如，在接口 `Read` 中，参数 `val_t` 定义了读取数据的类型，并将其作为事件 `readDone` 第二个形参的类型。在接口 `Get` 中，参数 `val_t` 定义获取数据的类型，将类型 `val_t` 作为命令 `get()` 返回值的类型。`Get` 接口代码如下：

```
interface Get<val_t>{
    command val_t get();
}
```

对于通用接口，有以下三点说明。

- 1) 通用接口可以支持多个参数类型，在各参数类型之间使用逗号分隔。
- 2) 组件声明通用接口时必须指定参数。
- 3) 通用接口的类型也可能不被接口中的任意一个命令或事件使用。但是接口的提供者和使用之间进行连接时，接口的类型参数一定要匹配。

例 3.8: 通用接口 TimeSyncPacket

```
interface TimeSyncPacket<precision_tag, size_type>{
    command bool isValid(message_t* msg);
    command size_type eventTime(message_t* msg);
}
```

通用接口 TimeSyncPacket 具有两个参数类型，precision_tag 和 size_type。参数 precision_tag 标识计时器的精确度类型，接口 TimeSyncPacket 中的命令和事件没有使用这个参数。参数 size_type 标识了命令 eventTime 的返回值类型。

磁力计组件的规范中提供的接口 Read 也是一个通用接口，接口 Read 的参数类型为 uint16_t。如果将使用 Timer<TMilli>接口的组件与提供 Timer<TMicro>接口的组件连接起来，则会产生编译错误。

例 3.9: MagnetometerC 组件

```
module MagnetometerC{
    provides interface StdControl;
    provides interface Read<uint16_t>
}
```

3.4 模块

在介绍了组件和接口等概念的基础之上，本节着重介绍模块的实现。

模块实现包括分配状态和实现执行逻辑，主要包括两类代码：数据定义或声明和函数定义。

在模块实现代码部分的数据可以分为变量和常量两种类型。模块变量是静态存储变量。除特别声明外，函数内部的变量都是动态存储变量。为了屏蔽多平台的边界对齐差异以及同一平台上 MCU 与无线通信芯片字节顺序差异，nesC 支持一种新的平台独立类型的数据。模块实现中使用的常量可以通过常值变量、宏定义和枚举常量等多种方式来定义。

函数可以细分为接口函数、任务和一般的内部函数。接口函数主要对接口中的命令或事件进行定义。任务是 TinyOS 并发执行模型的主要组成部分，是轻量级的可以延后执行的过程调用，可以在某时刻提交并被调度器按照一定的调度策略调度执行。

总的来说，模块实现部分的大部分 nesC 代码与 C 非常相似，主要的区别在于：接口函数的定义与调用、任务定义与提交以及模块变量的命名范围等。

3.4.1 接口函数

1. 接口函数的定义与调用

我们知道，接口由一组被称为命令或事件的函数组成，组件必须实现它提供接口中的命

令以及使用接口中的事件。因此，在模块实现部分就要定义接口函数。不同的接口中可能存在相同的命令名字或事件名字，为了保证函数定义的唯一性，在模块实现中定义接口函数时，函数名字由接口名字与接口中的命令或事件名字组合而成，中间使用“.”连接到一起，避免了不同接口中可能存在的同名命令或事件带来的混乱。下面的程序是模块 `PowerupC` 和 `RealmainP` 的定义，关键字 `implementation` 后面大括号中的代码是模块实现。由于 `PowerupC` 使用 `Boot` 接口，`Boot` 接口中有一个事件 `booted`，因此，模块 `PowerupC` 中必须定义接口函数 `event void Boot.booted() {...}`。接口函数名字由接口名字 `Boot` 与事件名字 `booted` 联合组成。当接口 `Boot` 的提供者 `RealmainP` 向模块 `PowerupC` 通知系统已经启动的事件（`signal Boot.booted()`）时，就调用了模块 `PowerupC` 中的 `Boot.booted()` 函数，运行模块 `PowerupC` 中 `Boot.booted()` 函数，会进一步调用接口命令 `Leds.Leds0on()`，打开相应的二极管。从模块 `PowerupC` 以及 `RealmainP` 中的例子可以看到，模块实现部分除了接口函数的定义外，接口函数调用也与 C 中函数调用的用法不同，需要使用新的关键字。

- 调用接口命令函数时使用关键字 `call`，如模块 `PowerupC` 中的调用接口命令语句：

```
call Leds.led0on();
```

- 调用接口事件函数使用关键字 `signal`，如模块 `RealmainP` 中的调用接口事件语句：

```
signal Boot.booted();
```

在上述的描述中，读者自然会产生这样一个疑问：为什么在模块 `RealMainP` 中运行 `signal Boot.booted()` 语句，会去调用模块 `PoweupC` 中定义的接口函数 `Boot.booted()` 呢？这些函数名字的调用关系是如何绑定的呢？答案是：只需要将接口（比如这里的 `Boot`）的提供者和使用者的连接上就可以了。连接接口的使用者和提供者是配件的工作，我们将在下一节讨论。模块 `PowerupC` 和 `RealmainP` 的定义代码如下：

```
module PowerupC{
    uses interface Boot;
    uses interface Leds;
}
implementation{ //模块实现部分开始
    event void Boot.booted()
    {
        call Leds.led0On();
    }
}

module RealMainP@safe(){
    provides interface Boot;
    ...
}
implementation{
    int main()@spontaneous()//组件的内部函数
    {
        ...
        signal Boot.booted();//通知系统已经启动的事件
    }
}
```

2. 通过接口函数传递参数

组件间交互的唯一方式是通过函数调用，这里的函数通常是指接口函数。通过接口函数在组件之间传递参数的方式有两种：赋值和引用（指针）。第一种方式是将数据复制到栈。在这种方式下，被调用函数可以自由存放和修改数据。第二种方式是在调用函数和被调用函数之间共享指针变量。在这种方式下，为了避免内存数据损坏和内存空间泄露，两个组件都要谨慎管理对数据的访问，因此应该尽量减少数据共享情况的出现。

减少数据共享的一种简单的方法是在模块变量中不存放指针。在一些抽象数据组件中就使用了这种方法。抽象数据组件主要包括通用模块和带有指针参数命令的通用接口。将数据封装在组件中，使用接口中提供的命令来访问这些数据。由于使用指针作为参数的命令或者事件的持续时间是有限的，因此数据共享的时间也很短暂，减少了数据共享带来的问题，但仍然会出现数据不一致的问题。

这种方法在分阶段调用中不适用。考虑下面的接口 Send 中的函数 send 和 sendDone。

```
interface Send {
    command error_t send(message_t* msg, uint8_t len);
    command error_t cancel(message_t* msg);
    event void sendDone(message_t* msg, error_t error);
    command uint8_t maxPayloadLength();
    command void* getPayload(message_t* msg, uint8_t len);
}
```

接口 Send 也是一个分阶段接口。发送分组的 Send 接口中具有 4 个命令和 1 个事件。Send 接口的使用者调用 Send.send() 命令，即调用下层组件发送分组，当下层组件发送完分组后，会调用该接口中的 sendDone() 事件通知上层组件。更具体地说，为了发送一个数据分组，接口 Send 的使用者调用命令 Send.send()，返回值是 SUCCESS，表明已经将指向数据分组的指针 (msg) 传递给了接口的提供者 (被调用者)。被调用者将在一个变量中存储该指针，改变状态并直接返回。如果接口 Send 的使用者在将数据分组传递给接口的提供者后修改了数据包，则数据分组可能会被损坏。例如，接口提供者中的调用函数在计算完整个数据分组的校验和后发送该分组，此后又修改了该数据分组中某些字段的值，导致数据分组的校验和不再与该分组匹配，在这种情况下，接收该数据分组的节点会认为收到一个错误的分组而丢弃它，造成了资源的浪费。

为了避免上述问题，在 TinyOS 中提出了“所有权”规则：任何时间存放变量的内存块应归唯一的某个模块所有。在上述例子中，接口 Send 的使用者调用了命令 Send.send()，实际上就将指针 msg 所指的内存块的所有权从调用者 (接口的使用者) 交给了被调用者 (接口的提供者)，因此，在该分组真正发送完毕之前，即收到 Send.sendDone() 事件之前，接口 Send 的使用者不是该内存块的所有者，所以不要对该内存块进行操作，这样可以避免共享指针带来的损坏内存数据的问题。为了充分利用内存，避免内存泄露问题，还应在 Send.sendDone() 事件中将相应的内存块指针作为参数传递回接口的使用者，将内存块的所有权归还给它原来的所有者，使得原来的所有者可以继续使用该内存块。

3.4.2 任务

在模块实现中有一类推后调用的函数形式，被称为任务 (task)。任务可以使某个计算逻辑推后一段时间执行。任务相对于其他任务具有原子性，在运行过程中，不必担心其他任务破坏自己的数据。任务应尽量短，否则会影响其他任务的执行，降低系统响应的灵敏性。

定义任务 (定义函数) 和提交任务 (调用函数) 都在模块实现部分完成。可以使用两种方式来定义任务和提交任务。对于同一个任务，两种方式不能混用。

1. 第一种定义任务和提交任务的方式

```

//定义任务
task void 任务名(){
    ...
}

//提交任务
post 任务名();

```

task 是定义任务时使用的关键字，表明后面的函数是一个任务，任务没有返回值，没有参数。

post 是提交任务时使用的关键字。执行该语句，实际上是将任务名所指的任务挂入任务队列后立即返回。任务提交成功返回 **SUCCESS**，否则返回 **EBUSY**。

2. 第二种定义任务和提交任务的方式

在 TinyOS 中提供了一个任务接口 **TaskBasic**，该接口中包括一个提交任务的异步命令和一个通知运行任务的事件。使用该接口来定义任务和提交任务与第一种方式是等价的。

```

interface TaskBasic{
    async command error_t postTask();    //提交任务
    event void runTask();                //运行任务
}

```

当组件用关键字 “**task**” 定义一个任务时，实际上就表示它使用了接口 **TaskBasic** 的一个实例。任务的主体就是事件 “**runTask**”，当组件使用关键字 “**post**” 时，实际上是调用了命令 **postTask**。每个任务接口都有唯一的标识作为参数来与调度器组件连接，而这个参数是通过以 “**TinySchedulerC.TaskBasic**” 字符串为实参的编译函数 **unique()** 来获得的。编译函数 **unique()** 将在第五章中介绍。

例 3.10: BlinkTaskC 组件

```

#include "Timer.h"
module BlinkTaskC{
    uses interface Timer<TMilli> as Timer0;
    uses interface Leds;
    uses interface Boot;
}
implementation{
    task void toggle(){
        call Leds.led0Toggle();
    }
    event void Boot.booted(){
        call Timer0.startPeriodic(1000);
    }
    event void Timer0.fired(){
        post toggle();
    }
}

```

BlinkTaskC 组件使用了第二种定义和提交任务的方式。

此任务是以 1000ms 为周期来触发 Led0 灯，切换其亮灭状态。在模块的实现部分，定义了一个名为 toggle 的任务，此任务调用 Leds 接口中的命令 led0Toggle()来切换 Led0 的亮灭状态。在事件 Boot.booted()中，设定定时器 Timer0 的值为 1000ms，当定时器 Timer0 的时间到达 1000ms 时，重新初始化，使定时器重新开始计时。在最后事件 Timer0.fired()中，当定时器 Timer0 被触发时，提交一个任务。

3.4.3 模块数据

1. 模块变量

一般来说，nesC 中不提倡使用 malloc 或者其他 C 库函数来动态分配内存。这是因为大多数嵌入式微控制器没有内存保护，动态分配内存会带来一定的风险。合适的做法是在模块中通过变量来静态分配内存。模块变量定义位于模块实现代码的开始部分。

例：模块 PeriodicReaderC 的工作是周期性采集传感器数据。在模块 PeriodicReaderC 的实现部分，除了定义 StdControl 接口中的命令和 Timer、Read 接口中的事件外，在开始位置还定义一个类型为 uint_16 的变量 lastVal，用于存放传感器数据。uint_16 的关键字，后面会详细说明。

对于模块变量有三点说明。

1) 模块变量定义位于模块实现部分的开始位置。模块变量定义与 C 变量是一致的，在定义的同时可以初始化（如下面第一行），也可以将定义与初始化分开（如下面第二行和第三行）。

```
uint8_t count=1;
message_t packet;
message_t* packetPtr =&packet;
```

2) 模块变量的命名范围是组件。本模块文件范围内可以引用该变量，而其他组件不能直接引用该变量，其他组件引用 lastVal 的唯一方式是通过接口函数。在这个例子中，其他组件要引用 lastVal，可以调用函数 Read.readDone()来返回该变量的值。

3) 模块变量是静态存储变量，类似于 C 中使用关键字 static 标识的变量。

除了在模块实现代码的开始位置定义模块变量外，在各函数内部也可以根据需要在内部变量，与 C 函数内部变量的用法一致，这里不再赘述。

2. 平台独立数据

TinyOS 具有支持多种硬件平台的能力。微控制器和无线通信芯片是硬件平台的主要组成部分。无线芯片与微控制器对数据的布局（字节顺序 big-endian/little-endian）可能有所不同。不同微控制器上处理器的地址总线宽度也可能不同，比如 MSP430 的地址总线是 16 位，Atmag128 的地址总线是 8 位，这往往使得同一个数据结构在有的平台上能够对准边界，有的平台上却不能对准边界。而没有边界对准的字是不能加载使用的。比如 TelosB 平台上，针对无线通信芯片 CC2420 的通信分组控制头的数据结构如下所示，该数据结构的字节长度是 11，为奇数。

```
typedef struct cc2420_header_t{
    uint8_t length;
```

```
uint16_t fcf;
uint8_t dsn;
uint16_t destpan;
uint16_t dest;
uint16_t src;
uint8_t type;
}cc2420_header_t;
```

该平台的微控制器采用的是 MSP430F1611 (MSP430 系列中的一种), 由于其地址总线宽度是 16 位, 字节的变量必须与 2 字节边界对准。为了解决这个问题, MSP430 编译器会在后面填充一个字节, 从而实现边界对齐。但这样做也带了新的问题, 即为针对 CC2420 设计的这个数据结构在不同平台 (无线通信芯片相同, 微控制器不同) 上的数据结构是不相容的。如何使得 TinyOS 程序中的数据结构独立于硬件平台, 以便于能够对数据自由访问与使用呢?

一种常用的解决无线芯片与处理器之间不同字节顺序的方法是调用宏指令来转换微控制器和无线芯片的字节顺序, 如 UNIX 中的宏指令 `htons`, `ntohl` 等。但是这种方法容易出错, 也不能解决边界对准的问题。

在 TinyOS 1.x 中, 一些程序试图通过使用 gcc 的 `packed` 属性来解决数据结构独立于硬件平台。如例 3.11 所示, `packed` 告知 gcc 忽略平台的结构对齐要求来紧凑封装一个数据结构。这种做法使得运行于 ATmega128 和 x86 上的代码数据结构格式保持一致, 然而也存在一些问题。首先, MSP430 系列的 gcc 版本 (用于 Telos 系列节点) 不能正确处理使用 `packed` 属性标识的数据结构; 其次, `packed` 属性是 gcc 特有的, 因此代码可移植性不好; 最后, 这种方法能够解决边界对准问题, 但是不能解决不同的字节顺序问题, 也就是说, 在存在不同字节顺序的平台上, 仍然需要使用 `htons`、`ntohl` 等的宏转换。

例 3.11:

```
typedef struct RPEstEntry {
    uint16_t id;
    uint8_t receiveEst;
} __attribute__((packed)) RPEstEntry;
```

在 TinyOS 2.x 中的做法是使用了 nesC 1.2 中的平台独立类型。对于简单数据类型而言, 除了在关键字之前加上 `nx_` (表示高字节顺序布局) 或者 `nxle_` (低字节顺序布局) 前缀外, 平台独立类型数据的定义 (声明) 与正常类型数据一致。如例 3.12。

例 3.12:

```
nx_uint16_t val; // A big -endian 16-bit value
nxle_uint32_t otherVal; // A little -endian 32-bit value
```

除了简单数据类型, 也有独立于平台的结构体 (`struct`) 和共用体 (`union`), 分别用 `nx_struct` 和 `nx_union` 声明。这些数据类型中的每个域也必须是平台独立类型。非位的域以字节为单位对准边界。例如, 针对无线通信芯片 CC2420 的通信分组控制头的平台独立的结构体如下。在所有硬件平台上编译这个结构体时都使用相同的内存布局, 对结构体中的各个域都使用相同的字节顺序。这就使得平台代码可以封装和解封装该结构体, 而不用使用

htons、ntohl 等宏指令。

```
typedef nx_struct cc2420_header_t{
    nxle_uint8_t length;
    nxle_uint16_t fcf;
    nxle_uint8_t dsn;
    nxle_uint16_t destpan;
    nxle_uint16_t dest;
    nxle_uint16_t src;
    nxle_uint8_t type;
}cc2420_header_t;
```

对大多数 nesC 代码，可以忽略运行时间的开销。如例 3.13。

例 3.13:

```
nx_uint16_t x = 5;
uint16_t y = x;
```

上述代码需要几个处理器时钟周期将 x 的字节重新排序，按照本地芯片的字节顺序形成 y。需要注意的是，系统处理开销与转化次数成正比。因此，在需要对某个平台独立类型进行大量计算或者多次访问时，需要先将其转化为本地数据格式来处理。例如，要对多字节数组执行计算，应在计算前把这些数组复制到本地格式，在计算结束后再将这些数组复制回平台独立的数据中。

3. 常值变量

在 TinyOS 程序实现中往往需要一些常量，比如，重传次数或阈值。可以采用常值变量、枚举以及#define 三种方式实现，这三种方式可以达到相同的效果，但是内存分配及其他方面稍有不同。例如，定义一个最大重传次数，这三种方式声明（定义）分别如下。

方式一：常值变量

```
const int MAX_RETRANSMIT = 5;
```

方式二：MAX_RETRANSMIT 5 枚举类型的常量

```
enum {
    MAX_RETRANSMIT = 5
};
```

方式三：宏定义 #define

```
#define MAX_RETRANSMIT 5
```

在 C 中，常值变量是一种常用方式，它属于静态存储分配，编译器会为其分配一块 RAM 或 ROM 区域，即便程序运行完，这部分存储区域也不释放。使用枚举常量（枚举元素）来定义常量也可以达到相同的目的。枚举常量在节省内存方面优于前一种方式，属于动态存储变量。相应程序函数运行结束，这部分区域可以重新使用。枚举常量在某些方面也优于第三种方式宏定义，因为枚举常量存在调试符号表和应用的元数据中。然而，枚举常量也有一些局限性，只能声明整数常量，而使用宏定义还可以定义浮点和字符串常量等。

需要注意的是，由于枚举类型默认为整数宽度，使用枚举类型声明变量可能浪费内存。例如枚举类型变量 `state_t` 的定义如例 3.14。

例 3.14:

```
typedef enum {  
    STATE_OFF = 0,  
    STATE_STARTING = 1,  
    STATE_ON = 2,  
    STATE_STOPPING = 3  
} state_t;
```

程序中需要定义变量 `state`，该变量的有效值范围是 0~3。我们可以采用下述两种不同的方式来定义变量 `state`。

方式一:

```
state_t state; // platform int size (e.g., 2-4 bytes)
```

方式二:

```
uint8_t state; // one byte
```

在第一种方式下，将变量 `state` 定义为枚举类型。由于该类型变量的存储单位是本地整数宽度，在微控制器上可能是 2~4 个字节（依据不同微控制器而不同）。在第二个方式下，由于使用类型 `uint8_t`，因此只分配单个字节。所以当变量所需长度比较短时，应该避免使用枚举类型声明变量。

另外，模块中的数据是局部的，模块之间通过接口函数实现数据（状态）的交互。除了这些局部数据之外，nesC 程序中还需要一些全局可用的数据结构、数据类型及函数，比如 `message_t`、`error_t` 以及 `ecombine(...)`。为了说明方便，这些统称为全局名字。与 C 一致，nesC 中的这些全局名字通常在头文件中定义。例如，TinyOS 的 `error_t` 类型和错误常量在 `\opt\tinyos-2.x\tos\types\TinyError.h` 中定义。如例 3.15。

例 3.15:

```
...  
enum {  
    SUCCESS = 0,  
    FAIL = 1, // Generic condition: backwards compatible  
    ESIZE = 2, // Parameter passed in was too big.  
    ECANCEL = 3, // Operation cancelled by a call.  
    EOFF = 4, // Subsystem is not active  
    EBUSY = 5, // The underlying system is busy; retry later  
    EINVAL = 6, // An invalid parameter was passed  
    ERETRY = 7, // A rare and transient failure: can retry  
    ERESERVE = 8, // Reservation required before usage  
    EALREADY = 9, // The device state you are requesting is already set  
};  
typedef uint8_t error_t;
```

需要使用相应的全局名字时，只需要在模块、接口或配件等源文件中的适当位置（通常是关键字 `module`、`interface` 和 `configuration` 之前）使用文件包含（`#include`）来声明即可。另外，在 `nesC` 中还可以使用文件包含，使得在模块等内部使用的是标准 C 库或者是用户私有 C 代码库。

3.5 配件与连接

前一节介绍了模块实现。模块实现代码负责分配状态和定义可执行逻辑。在模块命名范围内的函数与变量仅能供模块自己访问，其他组件不能直接访问。如果模块 A 要调用模块 B 的函数，需要将模块 B 中的函数名字映射到模块 A 中。在 `nesC` 中，这种映射是通过配件连接（`wiring`）两个组件的接口来实现的。

基于连接，配件能够将现有组件组织成一个新的抽象，这是 `nesC` 区别于 C 的一个主要方面。尽管配件的语法非常简单（仅有几个操作符：`->`、`<-` 和 `=`），但是，对于熟悉了诸如 C 这样的全局范围编程模式的编程人员来说，这种编程模式是较为陌生的，应该作为重点来学习。

配件也包括规范和实现两个组成部分。有一类配件位于最上层，将其他组件连接成为一个完整的应用，称为顶层配件。顶层配件在其规范中既不提供接口也不使用接口。每个应用都对应着一个顶层配件。在 `\opt\tinyos-2.x\apps\` 目录下的每一个应用中，都有一个顶层配件的源文件。对于顶层配件的命名，有一个非强制性约定：顶层配件名字的最后几个字母是 `AppC`，表明这是一个完整应用的顶层配件。顶层配件源文件的扩展名仍然是“`.nc`”。

除了顶层配件外，TinyOS 下的其他配件都是系统或其他组件能够使用的抽象。在这样的配件规范中要声明提供的接口、使用的接口。对于这些自己声明的接口，配件还需要通过输出接口方式来实现。

配件的规范在上一节中已经详细描述，本节主要讨论配件的实现。

3.5.1 配件实现

配件定义的一般形式如下：

```
configuration 配件名 {
    //组件规范，即声明提供的接口、使用的接口，也可以为空
}
implementation {
    //声明组件语句
    components 组件名 1, 组件名 2, ..., 组件名 n;
    //连接语句
    连接组件语句;
    输出接口语句;
}
```

在关键字 `implementation` 后面的大括号内是配件实现代码。

配件的实现代码由两部分组成。第一部分是声明组件语句，用于声明配件将要使用的组件。第二部分是连接语句。在连接语句中，有一类是使用操作符“`->`”或“`<-`”将前面声

明的组件连接起来，另一类是使用操作符“=”将自己声明的接口与前面某个声明组件的接口连接起来，从而实现配件声明的接口。对于顶层配件而言，其实现代码中的连接语句只有第一类。

例 3.16: 应用 Powerup 的顶层配件 PowerupAppC

```
configuration PowerupAppC {}
implementation {
    components MainC, PowerupC, LedsC;           // (1) 声明组件
    MainC.Boot <- PowerupC;                       // (2) 连接组件
    PowerupC -> LedsC.Leds;                       // (3) 连接组件
}
```

以上是应用 Powerup 的顶层配件 PowerupAppC 的代码。配件 PowerupAppC 的规范为空。配件 PowerupAppC 的实现代码中：行（1）是声明组件语句，声明了本配件使用的三个组件；行（2）和行（3）是第一类连接语句，将行（1）中声明的组件按照需要连接起来。具体地说：使用操作符“<-”将组件 MainC 与组件 PowerupC 通过接口 Boot 连接起来；使用操作符“->”将 PowerupC 与 LedsC 通过接口 Leds 连接起来。配件 LedsC 的定义如例 3.18。

例 3.17: 配件 LedsC

```
configuration LedsC {
    provides interface Init;
    provides interface Leds;
}
implementation {
    components LedsP, PlatformLedsC;             // (1)
    Init = LedsP;                                 // (2)
    Leds = LedsP;                                 // (3)
    LedsP.Led0 -> PlatformLedsC.Led0;             // (4)
    //将模块 LedsP 通过接口 Led0 连接到组件 PlatformLedsC
    LedsP.Led1 -> PlatformLedsC.Led1;             // (5)
}
```

配件 LedsC 的规范中提供了接口 Init 和接口 Leds。在配件 LedsC 实现代码中，除了使用操作符“->”连接组件 LedsP 与组件 PlatformLedsC 外（行（4）、行（5）），还使用操作符“=”，将自己提供的接口 Init（Leds）与 LedsP 提供的接口 Init（Leds）连接起来（行（2）、行（3））。后面会解释这几个操作符的作用。

在配件的实现代码中，声明组件与 C 中声明某个变量类似，难点主要在于使用操作符来实现连接。

1. 声明组件

在配件实现代码中声明组件时，使用关键字 components，后面跟着需要声明的组件名字，如果要声明多个组件，可以使用多条语句分别声明，也可以使用一条语句一起声明，对于组件名字的先后顺序没有要求，但是要用逗号隔开，声明组件语句以分号结尾。

例如在顶层配件 PowerupAppC 中，行（1）使用一条语句声明了三个组件 MainC、PowerupC 和 LedsC。使用多条语句分别声明也是可以的。

```
components MainC;
components PowerupC;
components LedsC;
```

在声明组件语句中，可以使用关键字 `as` 为组件设定一个别名。在 3.2.2 小节中我们已经介绍过关键字 `as`，主要讨论了如何使用关键字 `as` 为接口设定别名。这里继续介绍一下在配件实现代码中声明组件时关键字 `as` 的主要应用场合。

1) `nesC` 组件名字位于全局范围，为了增强名字的描述性，组件名往往较长。例如，ATmega128 上 SPI 总线的硬件表示层组件名字是 `HplAtm128SpiC`。在配件中使用组件时，较长的名字往往为书写带来麻烦，因此，在声明时可以使用 `as` 为其重新命名。例如，在封装 `HplAtm128SpiC` 的稍上一级的配件如 `Atm128SpiC` 中，将 `HplAtm128C` 重新命名如下：

```
components HplAtm128C as HplSpi;
```

实际上，在配件中声明组件时都隐性的使用了关键字 `as`。这与在组件规范中声明接口是一致的。上述声明组件 `MainC` 的语句实际上是下述语句的缩写。

```
components MainC as MainC;
```

2) 关键字 `as` 的一个重要应用场景是：在多次实例化同一通用组件（通用组件在后面会介绍）时，需要不同的名字分别与各实例对应。这种场景下，可以实现区分的唯一方式是使用关键字 `as` 为其设置别名。例如，在配件 `BlinkAppC` 的实现代码中实例化通用组件 `TimerMilliC()` 得到三个实例，为了区分这三个实例，使用关键字 `as` 分别为其命名为 `Timer0`、`Timer1` 和 `Timer2`。

例 3.18: `BlinkAppC` 配件

```
configuration BlinkAppC
{
}
implementation
{
  components MainC, BlinkC, LedsC;
  components new TimerMilliC() as Timer0;
  components new TimerMilliC() as Timer1;
  components new TimerMilliC() as Timer2;
  BlinkC -> MainC.Boot;
  BlinkC.Timer0 -> Timer0;
  BlinkC.Timer1 -> Timer1;
  BlinkC.Timer2 -> Timer2;
  BlinkC.Leds -> LedsC;
}
```

3) 关键字 `as` 还有一个重要的作用，就是带有一定的间接性。具体地说就是，使用关键字 `as` 为某个组件设定一个别名，当需要替换这个组件时，只需要替换这一行，而不必在所有该组件名出现的地方都进行替换操作，减少了差错出现的可能。

例如，在配件 `ActiveMessageC` 的实现部分，使用关键字 `as` 把组件 `CC2420Active`

MessageC 设定别名为 AM。然后通过操作符 “=” 将组件 ActiveMessageC 提供的接口与组件 AM（即组件 CC2420ActiveMessageC）提供的接口连接起来。这样，若需要替换该组件的使用，则可以直接替换实现部分的第一行替换该组件即可。

2. 连接

在 nesC 中用两类操作符来实现连接语句：一类是操作符 “->” 或 “<-”，用来将接口的使用者与接口的提供者连接起来；另一类是操作符 “=”，用来将配件自身提供或使用的接口与某个声明组件的接口连接起来。下面分别介绍这两类操作符。

（1）操作符“->”和“<-”

操作符“->”和“<-”是两个基本的连接操作符，用于将接口的提供者与使用者连接起来，实现调用组件和被调用组件之间的绑定。箭头从使用接口的组件（接口的使用者）指向提供接口的组件（接口的提供者）。不必区分连接的这些组件是配件还是模块。

在配件 PowerupAppC 中，行（2）的连接语句使用操作符 “<-” 将组件 MainC 和组件 PowerupC 通过接口 Boot 连接起来。其中，PowerupC 是接口 Boot 的使用者，位于箭头的尾部，MainC 是接口 Boot 的提供者，位于箭头的头部。值得注意的是，只有 PowerupC 和 MainC 在其规范中分别真正地声明了使用接口 Boot 和提供接口 Boot，才能保证这种连接不出现错误。

行（3）使用操作符 “->” 将组件 PowerupC 和 LedsC 通过接口 Leds 连接起来，PowerupC 是接口 Leds 的使用者，LedsC 是接口 Leds 的提供者。

只要保证箭头指向符合从接口使用者到接口提供者，操作符“->”和“<-”具有相同的连接效果。例如，下面两行是等价的。

```
MainC.Boot <- PowerupC;  
PowerupC-> MainC.Boot;
```

1) 快捷连接

在操作符 “->” 的两侧是两个端点。通常情况下，端点由声明的组件实例及其接口实例组成，如配件 LedsC 中的行（4）：LedsP.Led0 -> PlatformLedsC.Led0。操作符左侧的端点是 LedsP.Led0，LedsP 是模块 LedsP 的实例，Led0 是 LedsP 规范中声明的类型为 Leds 的实例，名字为 Led0。操作符右侧的端点是 PlatformLedsC.Led0，由组件类型为 PlatformLedsC 的实例与接口类型 Leds 的实例组成。在连接语句中，要求两个端点的接口实例是同类型的。不能将不同类型的接口实例连接在一起，比如，不能将接口 Boot 连接到接口 Leds 上。对于参数化接口而言，接口类型及参数都应匹配，比如，不能将 Read<uint8_t> 与接口 Read<int16_t> 连接到一起。否则会产生编译错误。

在操作符两侧端点中组件的接口实例类型一致的前提下，如果某侧端点的接口实例不存在连接歧义性，那么可以将该端点的接口名字省略。这是一种快捷连接的写法。例如，在配件 PowerupAppC 中，行（2）与行（3）都使用了快捷连接写法。在操作符的左侧，端点 MainC.Boot 中使用的是接口 Boot 的一个实例。在操作符的右侧端点只写了组件名字 PowerupC，没有接口 Boot 的名字。这是因为组件 PowerupC 仅提供了接口 Boot 的一个实例，nesC 认为连接时就应该连接到这个实例。行（3）的情况类似。行（2）与行（3）是与下面两条语句等价的快捷连接写法。

```
MainC.Boot <- PowerupC.Boot;      //与配件 PowerupAppC 中行（2）的语句等价
PowerupC.Leds -> LedsC.Leds;      //与配件 PowerupAppC 中行（3）的语句等价
```

类似的，这种快捷连接写法也可以应用到使用操作符“=”的输出连接语句中，如配件 LedsC 实现代码的行（2）和行（3）。LedsC 的接口 Leds 和接口 Init 被连接到组件 LedsP 上。

在输出接口的连接语句中，不能省去配件声明的输出接口的名字。例如在配件 LedsC 中的如下写法是错误的。

```
=LedsP.init
```

2) 使用快捷连接时是接口类型而不是名字。

例 3.19:

● 组件 BlinkC 的规范

```
module BlinkC @safe()
{
    uses interface Timer<TMilli> as Timer0;
    uses interface Timer<TMilli> as Timer1;
    uses interface Timer<TMilli> as Timer2;
    uses interface Leds;
    uses interface Boot;
}
```

● 组件 TimerMilliC 的规范

```
generic configuration TimerMilliC() {
    provides interface Timer<TMilli>;
}
```

下面的 nesC 连接语句是合法的。

```
Components new TimerMilliC() as Timer0;
BlinkC.Timer0->Timer0;
```

TimerMilliC()是一个通用配件，生成实例的方法与其应用场景后面会详细介绍，这里只需将其看成一个普通组件即可。BlinkC.Timer0 的类型是 Timer<TMilli>，连接时搜索组件 Timer0 中类型为 Timer<TMilli>的接口，发现有确定的类型匹配，因此能够正确连接。

如果连接中存在不确定性，即模糊性，那么就不能完成连接，会产生错误报告。例如，下面的连接语句是不合法的。由于组件 BlinkC 的规范声明了三个 Timer 接口实例，在连解语句中没有指出到底是哪个接口实例，存在不确定性，因此是无法实现连接的。

```
BlinkC->Timer0.Timer;
```

3) 连接如何工作

为了更好地理解连接是如何工作，下面以接口 Leds 为例继续分析配件 LedsC。配件 LedsC 将 LedsC.Leds 连接到 LedsP.Leds 上（行（3），后续会解释操作符“=”的作用）。组

件 PowerupAppC 实现代码的行 (2) 和组件 LedsC 实现代码的行 (3)，将组件 PowerupC 中接口函数名字 (局部变量) Leds.led0On 与 LedsP 中的接口函数名字 (局部变量) Leds.led0On 匹配到一起。也就是说，在 PowerupC 组件中调用接口函数 Leds.led0On()，实际上是调用了 LedsP 组件中的接口函数 Leds.led0On()。阅读 nesC 编译器生成的部分中间代码 (见 apps/Powerup/build/ 相应平台下的 app.c) 有利于我们更好地理解这一点。另外，为了减少函数调用带来的开销，nesC 编译器广泛使用内联函数 (inline) 来生成这些中间代码，最后编译成目标可执行代码时将这些函数的代码复制到被调用的地方。

```
static inline void LedsP.Leds.led0On(void )
{
    LedsP.Led0.clr();
}

inline static void PowerupC.Leds.led0On(void ){
    LedsP.Leds.led0On();
}

static inline void PowerupC.Boot.booted(void )
{
    PowerupC.Leds.Led0On();
}
```

(2) 操作符 “=”

除了顶层配件之外，其他的配件与模块相似，在其规范中也提供或使用接口。但是在配件实现中没有诸如模块那样的执行逻辑代码，也就是说，没有定义提供接口中命令或使用接口中事件的函数。因此，配件规范声明的接口必须由其他组件来定义。也就是说，用其他组件的接口为自己规范中声明的接口重命名。具体的实现方式是：使用关键字 **components** 声明一个组件；使用操作符 “=” 将其接口与自己规范中声明的接口连接起来。例如，在配件 LedsC 的规范中声明了提供的接口 Leds。由于配件实现代码没有接口函数的定义，那么如何使得 LedsC 能够提供接口 Leds 呢？来看看其实现代码，先声明了组件 LedsP (注意 LedsP 的规范中也提供接口 Leds)，再使用操作符 “=” 将自己提供的接口 Leds 与 LedsP 提供的接口 Leds 连接起来，也就是说，配件 LedsC 通过输出 LedsP 提供的接口 Leds，以实现自己提供接口 Leds。

站在编程者实现配件的角度来看，操作符 “=” 的作用是通过类似于重命名的方式，实现了配件规范中声明的接口。站在提供功能 (或服务) 的角度，操作符 “=” 的作用是将配件封装的某个组件的功能 (或服务) 输出。

(3) 操作符 “->” 和 “<-” 与操作符 “=” 的比较

操作符 “->”、“<-” 和 “=” 都起到了连接的作用。只要操作符两侧的组件的接口类型一致，而且连接的组件的接口实例不存在模糊性，都可以使用快捷连接的写法。但是，在操作符的作用、连接对象以及语法细节方面仍然存在差异，具体见表 3-2。

表 3-2 操作符 “->” / “<-” 与 “=” 的对比

	操作符 “->” 与 “<-”	操作符 “=”
作 用	连接接口的提供者与使用者	定义配件声明的接口如何实现，或者说，将自己封装的组件的接口（服务）输出
连 接 对 象	用关键字 <code>components</code> 声明的两个组件，且分别是同一个接口的提供者和使用者的	规范中声明的接口与关键字 <code>components</code> 声明的组件接口；要求操作符两端都是提供的接口（使用的接口）
语 法 要 求	箭头由接口的使用者指向提供者	没有左右的次序

3. 连接规则

在连接过程中，有两条规则需要遵循。

1) 规则 1：如果组件中使用关键字 `as` 为接口设置了别名，那么在配件实现代码的连接语句中必须使用这个别名。

例如，组件 `MainC` 与 `LedsP` 的规范，代码可参考例 3.4 和例 3.5。

基于组件 `MainC` 与组件 `LedsP` 的规范，下面两行连接语句中第一行是合法的，第二行是不合法的。

```
MainC.SoftwareInit->LedsP.Init;    //合法的连接语句
MainC.Init->LedsP.Init;            //不合法的连接语句，因为没有使用声明时设定的别名
```

2) 规则 2：配件在连接一个组件之前必须先声明它。

前面列举的例子中，配件的实现代码都是声明组件语句放在前面，连接语句放在后面。事实上，声明组件语句与连接语句之间可以有交叉，但是必须在连接语句之前声明该组件。

例 3.20:

一种合法的配件 `PowerupAppC` 的写法如下：

```
configuration PowerupAppC{}
implementation {
    components MainC, PowerupC;    // (1) 声明组件 MainC 和 PowerupC
    MainC.Boot <- PowerupC;        // (2) 通过接口 Boot 连接组件 MainC 和组件 PowerupC
    components LedsC;              // (3) 声明组件 LedsC
    PowerupC -> LedsC.Leds;        // (4) 通过接口 Leds 连接组件 PowerupC 和 LedsC
}
```

相对的，一种错误的配件 `PowerupAppC` 的写法如下。这是因为在行（2）中连接语句使用了组件 `PowerupC`，而在该连接语句之前没有声明 `PowerupC`。

```
configuration PowerupAppC{}
implementation {
    components MainC;              // (1) 声明组件 MainC
    MainC.Boot <- PowerupC;        // (2) 在本连接语句之前没有声明组件 PowerupC
    components PowerupC, LedsC;    // (3)
    PowerupC -> LedsC.Leds;        // (4)
}
```

3.5.2 多连接

1. 扇入与扇出

我们先前接触的配件 PowerupAppC 中的两条连接语句分别形成了两个一对一映射关系。如图 3-1 所示。

除了一对一映射，TinyOS 程序中还存在其他的映射关系。

例如，配件 CC2420TRansmitC 组织了一个针对无线通信芯片 CC2420 的发送分组抽象。在该配件中，前两条连接语句将本配件的 Init 接口分别连接到了 Alarm.Init 和 CC2420TransmitP.Init，形成了一个端点连接到两个不同端点的一对多映射关系。这是一种典型的多连接。在这种一对多连接模式下，只有一个调用点，即组件 CC2420TransmitC 的 Init.init()。该调用点将扇出两个被调用者，分别是组件 Alarm 中的 Init.init()和组件 CC2420TransmitP 中的 Init.init()。这种端点间形成的一对多的映射关系称之为扇出（fan-out）。如图 3-2 所示。

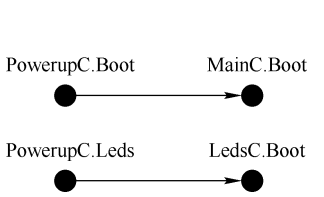


图 3-1 一对一映射关系

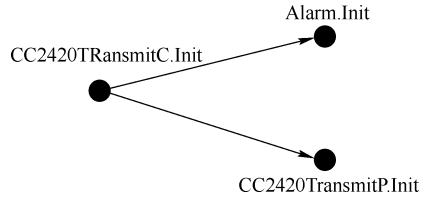


图 3-2 一对多映射关系

扇出是一个组件通过某类型的接口与多个组件相连。与扇出相反，当多个不同组件通过某类型接口同时连接到同一组件上时，连接语句会在端点之间形成多对一的映射关系，这种情形称为扇入（fan-in）。如图 3-3 所示。

例 3.21:

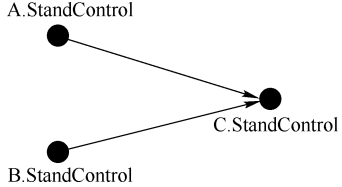


图 3-3 多对一映射关系

```
interface StandCtrol
{
    command error_t start();
    command error_t stop();
}
```

接口 StandCtrol 中有两个命令：start()和 stop()。下面的连接语句基于 StandCrol 接口形成了扇入。

```
components A, B, C;
A.StandCtrol->C;
B.StandCtrol->C;
```

扇入实际上就是多个调用者使用同一个被调用者（函数）。这种情形在 C 中也会遇到，比如在一处调用某个系统函数，因此不难理解。

我们前面介绍过分阶段接口，接口中有命令也有与命令相对应的完成事件，比如接

口 Send 中的命令 `send()` 及相应的完成事件 `sendDone`；再如接口 `SpiltControl` 中的命令 `start()` (`stop()`) 及相应的完成事件 `startDone()` (`stopDone()`)。基于分阶段接口形成的连接，如果在某命令上形成的连接是扇入（多个调用者）的。那么，在通知接口事件时，就会形成扇出（即多个被调用者）。这一点与基于单阶段接口（如 `StandCtrl`）形成的映射关系有所不同。

接口 `StandCtrl` 是单阶段的。基于该接口的服务或功能，在接口命令 `start` (`stop`) 返回时就会启动（停止）。上面例子中的语句在 `StandCtrl` 接口形成了扇入。相对的，基于 `SplitCtrl` 接口的服务，只有通知了 `startDone` (`stopDone`) 事件后才能保证启动（停止）。下面语句形成的连接下，不仅有启动命令也有完成事件，因此既形成了扇入也形成了扇出。

```
components A, B, C;
A.SplitCtrl->C;
B.SplitCtrl->C;
```

假如 A 或 B 中只有一个调用了 `SplitCrol.start()`，当 C 通知 `SplitCrol.startDone` 事件时，由于 A 和 B 都连接到 `SplitCtrl` 接口上了，因此，组件 A 的 `SplitCtrl.startDone()` 和 B 的 `SplitCtrl.startDone()` 都会被调用，实现中不能确定 A 还是 B 调用了命令 `SplitCrol.start()`。

推广开来，根据不同应用中的组件组织情况，导致组件之间的连接关系不是简单的一对一，而会存在 n 对 k 的映射关系。设 n 是接口的使用者， k 是接口的提供者，任意一个接口的提供者通知事件都会调用 n 个使用者的事件处理程序，任意一个接口的使用者调用命令，都会调用 k 个提供者中的命令。

2. 多连接带来的好处

多连接使得组件的一个接口函数实现是独立于它所依赖的组件的数目的。例如，组件 `MainC` 将启动顺序抽象为两个接口：`SoftwareInit` 和 `Boot`。在启动时，组件 `MainC` 调用 `SoftwareInit.init()`，使得启动开始前能够初始化软件组件。在启动顺序结束时，`MainC` 使用 `Boot.booted()` 通知节点已启动的事件。由于在一个实际应用中会有多个软件组件需要初始化，比如即便是非常简单的应用 `RadioCountToLeds` 都有 10 个软件组件需要初始化，所以一个合适的做法就是，将这些需要初始化的软件组件通过接口 `Init` 连接到 `MainC.SoftwareInit`。这样仅需要在组件 `MainC` 中调用 `SoftwareInit` 一次，就可以基于扇出调用这些组件的初始化代码，以实现相应的初始化。

还有一种做法是在 `MainC` 组件中使用多个 `Init` 接口，将这些接口分别与那些软件组件连接上，然后再按照某个次序分别调用它们。显然，相对于扇出的做法，这种做法不仅麻烦而且容易出错。

3. 组合函数

扇出带来一个有趣的问题：执行 `Call SoftwareInit.init()`，实际上调用了多个不同的函数，那么返回值应该是什么呢？

`nesC` 提供了组合函数来确定返回值。数据类型可以具有其相应的组合函数。扇出会调用 N 个函数，这 N 个函数具有相同类型的返回值，通过 $N-1$ 次调用组合函数将这 N 个返回值组合为一个。`nesC` 编译器编译应用时，自动生成使用了组合函数的扇出函数。例如，TinyOS 下的错误类型 `error_t` 就是一个常用返回值类型，它的组合函数是 `ecombine`。该组合

函数定义入下：

```
error_t @combine(error_t e1, error_t e2){
    return (e1==e2) ? e1 : FAIL;
}
```

如果 `e1` 和 `e2` 的值相同，`ecommbine` 就返回那个值，否则返回 `FAIL`。因此，如果组合了两个 `SUCCESS`，组合函数返回值是 `SUCCESS`，如果组合了两个 `EXXX`，返回值是 `EXXX`。其他的组合返回 `FAIL`。

使用 `nesC` 属性（后面将会介绍）将组合函数绑定到 `error_t`。

```
typedef uint8_t error_t @combine("ecommbine");
```

例 3.22：编译如下代码：

```
configuration InitExample{
implementation{
components MainC;
    components AppA, AppB, AppC;
    MainC.SoftwareInit->AppA;
    MainC.SoftwareInit->AppB;
    MainC.SoftwareInit->AppC;
}
```

编译器生成如下代码：

```
error_t MainC__SoftwareInit__init(){
    error_t result ;
    result = AppA__SoftwareInit__init() ;
    result = ecombine(result, AppB__SoftwareInit__init());
    result = ecombine(result, AppC__SoftwareInit__init());
    return result;
}
```

组合函数应具有组合性和可交换性，从而确保扇出调用的返回值不会依赖执行命令（事件）的顺序。

有些返回值没有组合函数，或者是由于编程者的疏忽，或者是由于数据类型的语义。后一种情况的例子包括诸如数据指针等，如果两个调用都返回到某个分组的指针，没有办法能将其组合成为一个单一指针。如果程序中扇出的调用的返回值不能被组合，`nesC` 编译器会生成一个没有组合或者没有组合函数的警告。编程者一定要重视这样的警告。

3.6 参数化接口

在 `nesC` 编程中，参数化接口是一个很重要的用法，能够为组件提供同类型接口的多个实例。本节将会介绍为什么需要参数化接口、如何基于参数化接口连接组件。在使用参数接口时，使用系统提供的编译函数 `unique()` 和 `uniqueCount()`，可以减少 Bug。

3.6.1 为什么需要参数化接口

有些情形需要一个组件能够提供某种接口给多个使用者（这里也将这些使用者称为用户）。比如，为了同时支持多种网络协议的发送和接收功能，TinyOS 下的主动消息（AM）通信组件需要提供接口 `Send` 和接口 `Receive` 给多个客户。再如，提供一项重要功能——计时。TinyOS 下每个硬件平台通常都有计时器组件 `HilTimerMilliC`，在平台硬件计时器上实现多个逻辑计时器（软计时器）。应用中有多少个软计时器，就需要 `HilTimerMilliC` 提供多少个接口 `Timer` 的实例。那么，如何在组件中提供同类型接口的多个实例呢？

第一种方式是在通用模块（在第六章中会介绍通用模块）中提供该接口，再将该通用模块实例化多次，从而达到支持同一类型接口的多个实例。这种做法存在两个缺点：一是浪费代码空间；二是在大多数情况下，这多个实例的实现代码做不到相互独立，相反的，会形成某种协作（依赖性）。例如，在计时器组件 `HilTimerMilliC` 上支持的多个逻辑计时器的实现中需要依据共同的策略（比如最早时间到达）。

第二种方式是在组件规范中直接声明多个接口实例，比如，在组件 `HilTimerMilliC` 的规范中提供 100 个类型为接口 `Timer` 的实例，如例 3.24。

例 3.23:

```
configuration HilTimerMilliC {
    provides interface Timer<TMilli> as Timer0;
    provides interface Timer<TMilli> as Timer1;
    provides interface Timer<TMilli> as Timer2;
    provides interface Timer<TMilli> as Timer3;
    ...
    provides interface Timer<TMilli> as Timer100;
}
```

这种方式能够工作，但是每个实例接口中的函数都要有相应的实现。比如，上述组件 `HilTimerMilliC`（实际是 `HilTimerMilliC` 封装的某个模块）的实现代码中，要实现 100 个 `Timer<TMilli>` 接口中的各个函数，而这些函数实现几乎完全一样。这种方式不仅比较麻烦，而且产生了大量重复代码。

第三种方式是使用一个参数用来指定在用哪个接口，这类似于 POSIX 文件系统的文件描述符。在这种方式下，应该生成唯一的计时器标识符，并在调用接口时传递该标识符。这种方式也可以工作，而且不会像第二种方式那样生成多个实现。但这种方式是在运行时传递参数，而通常来说这是不必要的。因为组件通常分配一定数目的计时器，而且仅使用这些计时器。或者说，通常在编译时就可以知道组件使用的计时器集合以及集合的大小，因而没有必要让调用者在运行时传递这个参数，继而引入可能的 Bug。

从编程者角度来看，重要的问题不在于使用者调用了哪些组件，而在于需要明确调用从那些组件而来。例如，对于计时器而言，重要的问题是如何区分调用者，以便向正确的组件通知 `fired()` 事件。上面介绍的第三种方式基于标识符可以实现区分，但是需要在模块中管理标识符。因为标识符是一个运行时参数，计时器服务组件中通知 `Timer.fired()` 事件时，扇出的各调用函数中必须检查这个标识符参数。基于这些考虑，为了支持抽象提供一组同类型的

接口实例，nesC 提供了参数化接口。参数化接口实质是接口数组，数组的索引是接口参数，标识了调用组件。对于多个接口实例仅需一个实现，接口参数是常量，在编译时确定，避免了运行时传递参数的弊端，既简化了代码，也提高了函数调用的效率。

3.6.2 基于参数化接口的两个典型例子

参数化接口既可以在配件的规范中声明，也可以在模块的规范中声明。

一个典型的使用参数化接口的例子是平台特定的主动消息通信组件 `ActiveMessageC`。

对于分组通信，在基于 TCP/IP 网络协议的处理过程中，节点接收到协议号（protocol 字段）为 6 的 IP 分组，认为这是一个 TCP 分组，会将其提交给 TCP 协议栈进一步处理。TinyOS 中提供的单跳分组通信的主动消息（AM）组件也采用了类似做法，只是没有遵循 IANA 的标准。

TinyOS 中单跳通信协议层（主动消息层，AM 层）以上的网络应用或协议具有各自的功能与分组格式，需要 AM 层能够将收到的分组正确地向上递交对应协议，为此，在 AM 消息中有一个长度为 8 比特的类型字段，用于承载上层协议的标识符。对于上层协议来说，需要它们注册自己的 AM 类型，以便发送和接收该类型的分组。

从实现 AM 通信的角度来看，AM 层上面有多少个协议，就需要 AM 组件中提供多少个类型为发送或接收接口的实例。更为具体地说，在 TinyOS 系统中，通常在每个硬件平台的目录中都会有一个名字为 `ActiveMessageC` 的配件，这是平台特定的 AM 通信组件。其中，配件 `ActiveMessageC` 规范中的接口 `AMSend`、接口 `Receive` 和接口 `Snoop` 都是参数化接口，这些接口的参数是主动消息的类型（协议标识符）。

例 3.24: 配件 `ActiveMessageC` 的规范部分

```
configuration ActiveMessageC {
  provides {
    interface SplitControl;

    interface AMSend[am_id_t id];
    interface Receive[am_id_t id];
    interface Receive as Snoop[am_id_t id];

    interface Packet;
    interface AMPacket;
    interface PacketAcknowledgements;
    interface PacketTimeStamp<T32khz, uint32_t> as PacketTimeStamp32khz;
    interface PacketTimeStamp<TMilli, uint32_t> as PacketTimeStampMilli;
  }
}
```

另一个使用参数化接口的典型例子是组件 `HilTimerMilliC`，其规范如例 3.26。它是一个平台特定的支持毫秒计时粒度的配件。该配件提供了类型为通用接口 `Timer<Tmilli>` 的接口数据 `TimerMilli[]`（参数化接口），通过提供 256（数组参数是 `unit8_t`）个单独 `Timer<Tmilli>` 接口实例，该配件支持多达 256 个独立的计时器。某组件在 `Timer<TMilli>` 接口上调用命令

startPeriodic(), 周期性地通知事件 Timer.fired()。正常情况下, 应用组件不直接连接 HilTimerMilliC。相反, 他们使用 TimerMilliC, 而 TimerMilliC 提供单一接口。

例 3.25: 配件 HilTimerMilliC 的规范部分

```
configuration HilTimerMilliC
{
    provides interface Init;
    provides interface Timer<TMilli> as TimerMilli[ uint8_t num ];
    provides interface LocalTime<TMilli>;
}
```

上述 2 个例子都是使用参数化接口的配件, 最后这个例子是使用参数化接口的模块。事实上, 为了实现配件的参数化接口, 需要具有相应参数化接口的模块。下面是针对 CC2420 无线通信芯片的模块, 该模块提供了多个参数化接口, 如 AMSend 和 Recive 等。

例 3.26: CC2420 无线通信芯片的模块

```
module CC2420ActiveMessageP @safe() {
    provides {
        interface AMSend[am_id_t id];
        interface Receive[am_id_t id];
        interface Receive as Snoop[am_id_t id];
        interface AMPacket;
        interface Packet;
        interface SendNotifier[am_id_t id];
        interface RadioBackoff[am_id_t id];
    }
    ...
}
```

3.6.3 基于参数化接口的连接

配件的作用是通过接口将声明的组件连接起来, 或者实现自己提供的接口。如果连接端点包括参数化接口实例, 那么连接不同, 语句会有什么不同么? 根据操作符的两个端点是否包括参数化接口, 可以分成下面两种情况。

第一种情况: 两个端点都包括参数化接口。

在这种情况下, 虽然两个端点都包含了参数化接口, 但是对应连接语句的写法与两端都不包括参数化接口的写法一致。

例 3.27:

```
1) : configuration CC2420ActiveMessageC{
    provides interface AMSend[am_id_t id];
    }{...}
2) : configuration ActiveMessageC{
    provides interface AMSend[am_id_t id];
    }
```

```
implementation{
    components CC2420ActiveMessageC;
    AMSend = CC2420ActiveMessageC;
}
```

在组件 `ActiveMessageC` 和组件 `CC2420ActiveMessageC` 的规范中都申明提供了参数化接口 `AMSend`。在组件 `ActiveMessageC` 的实现代码中输出组件 `CC2420ActiveMessageC` 的参数化接口 `AMSend` 的连接语句如上所示。在连接语句中不需要附加索引。

第二种情况：一个端点中包括参数化接口。

这种情况下，配件中的连接语句写法有所变化，在参数化接口中要填充确定的参数（索引）。这里仍然以组件 `ActiveMessageC` 为例来介绍。在正常情况下，最上层的应用并不直接连接组件 `ActiveMessageC`，而是使用组件 `AMSenderC` 和组件 `AMReceiver`，这些组件提供了简单的接口。这些提供简单接口的组件最后一定会连接到具有参数化接口的 `ActiveMessageC` 上。为了将焦点聚集到组件 `ActiveMessageC` 上，我们尝试跨越中间的组件组织关系，考虑有一个 AM 层的测试应用 `TestAMent`。该应用直接使用组件 `ActiveMessageC`。`TestAMAppC` 是顶层应用配件，连接模块 `TestAMC` 的接口 `Send (Receive)` 到组件 `ActiveMessageC` 的接口参数为 240 的参数化接口 `AMSend (Receive)` 上，如例 3.29 行（4）、行（5）语句所示。

例 3.28:

1) TestAMAppC 配件

```
Configuration TestAMAppC { }
implementation {
    components MainC, TestAMC as App;           //(1)
    components ActiveMessageC;                 //(2)
    MainC.SoftwareInit->ActiveMessageC;        //(3)
    App.Receive->ActiveMessageC.Receive[240];  //(4)
    App.AMSend->ActiveMessageC.AMSend[240];    //(5)
    ...
}
```

2) TestAMC 模块

```
module TestAMC{
    uses{
        ...
        interface Receive;
        interface AMSend;
        ....
    }
}
```

模块 `TestAMC` 负责收发包；配件 `TestAMAppC` 把 `TestAMC` 模块和 `MainC` 组件连接起来，该配件表明当 `TestAMC` 调用 `AMSend.send()` 时，它调用 `ActiveMessageC` 中编号为 240 的 `ActiveMessageC.AMSend` 命令（发送分组的网络协议号为 240）。类似的，下层组件收到协议类型为 240 的分组时，也会将该分组交给 `TestAMC`。由于这些常量在配件中指定，因此

不受限于模块。从模块的角度看，这些常量甚至不存在。也就是说从 TestAMC 的角度看，下面两行语句是等价的。

```
TestAMC.AMSend -> ActiveMessageC.AMSend240;           //不是真正的 TinyOS 代码
TestAMC.AMSend -> ActiveMessageC.AMSend[240];
```

它们的不同之处在于后面一行使用了参数化接口，参数实际上是接口中函数的一个参数。当调用 send()时，参数就是传递给它的编号并且在通知 sendDone 时也必须指定这个常量。参数化接口存在两个关键点：

- 当 TestAMC 调用 send 时自动填充了参数；
- 当 ActiveMessageC 触发 sendDone 时，会自动地根据参数提交分组。

3.6.4 参数化接口的实现

ActiveMessageC 是个配件，封装了特定无线通信芯片的通信协议栈，如 CC2420 无线通信芯片的通信组件 CC2420ActiveMessageC，而后者也是一个配件，它继续封装了一个模块 CC2420ActiveMessageP。也就是说，在模块 CC2420ActiveMessageP 中，真正定义了参数化接口 AMSend 的命令。相对于前面介绍的接口函数而言，参数化接口中函数的定义与调用稍有不同，需要增加接口参数部分。接口参数在函数形参列表之前，程序实现在处理接口参数时会像其他变量一样。

例如，在模块 CC2420ActiveMessageP 中的参数化接口的函数 AMSend.send 定义如下面代码所示，其中，at_id_t 指定了接口参数 id 的数据类型，而且接口参数位于函数（命令 send）形式参数列表之前。这实际上是个函数参数，nesC 编译器在编译时即可确定该值。当 CC2420 ActiveMessageP 要通知 sendDone 事件时，它取出分组的协议标识符（类型字段），并将其作为接口参数。如果分组 AM 类型字段值为 240，分发代码将通知连接到 ActiveMessageC.AMSend[240]的 sendDone 事件，并最终会调用到模块 TestAMC 的事件 AMSend.send Done()。接收过程类似，不再赘述。

参数化接口可以提供多个实例，但只有一个实现。相对的，上述对于基于名字的方法（比如 send240）而言，每个接口实例都要对应一个实现，即便这些实现是完全一样的，这样会造成代码空间的浪费。另外，由于参数是数值，它可以随意设置或更改，比如，下面这种写法。

```
#define ROUTING_TYPE
#define ROUTING_TYPE 201
#endif
Router.AMSend->PacketSenderC.AMSend[ROUTING_TYPE];
command error_t AMSend.send[at_id_t](at_id_t addr, message_t* msg, uint*_t len){
    cc2420_header_t* header =getHeader(msg);
    header->type =id;
}

event void SubSend.sendDone (message_t* msg, uint*_t result){
    signal AMSend.sendDone[call AMPacket.type(msg)](msg, result);
}
```

由于参数是在编译时确定的，nesC 能自动填写并以此参数为基础提交分组，因此，参数化接口既简化了代码，又提高了函数调用的效率。

由于每个使用该服务的客户连接了不同的参数标识符，因此，对于分阶段接口而言，这样能够避免完成事件的扇出情形。

考虑如下两种情况：

情况 1

```
components RouterC, SourceAC, SouceBC;
SourceAC.Send->RouterC;
SourceBC.Send->RouterC;
```

情况 2

```
components RouterC, SourceAC, SouceBC;
SourceAC.Send->RouterC[0];
SourceBC.Send->RouterC[1];
```

对于情况 1，如果 RouterC 通知 Send.sendDone，会把这个完成事件扇出给 SourceAC 和 SourceBC。因此，SourceAC 和 SourceBC 必须作进一步的判断才能了解该事件是否是发给自己的。对于情况 2，由于 RouterC 中保持了 Send[...]中传递的参数标识符，它能够向正确的组件通知完成事件。例如，SourceBC 调用了 Send.send，它实际是调用了 RouterC[1].Send.send()。由于 RouterC 存放了参数标识符 1，因此，它通知完成事件时，就会通过 Send.sendDone[1]()，即直接通知给 SourceBC，避免了扇出。

参数化接口与通用配件的结合能够在编译时虚拟抽象，不会造成空间浪费，也不需要使用函数指针。

3.6.5 unique()和 uniqueCount()函数

参数代表了唯一的客户标识符。为了避免 Bug 和冲突，nesC 提供了编译函数：unique() 和 uniqueCount()。

当 nesC 编译器编译程序时，它把所有 unique()函数调用转换为一个整数标识符。unique()函数以字符串作为参数，具有相同参数的 unique()函数调用返回互不相同的整数值。例如，两个组件 AppOneC 和 AppTwoC 都想调用参数化接口 Timer，编译函数 unique()的参数相同，皆为字符串“tmpTimer”，因此就会得到参数化接口 Timer 不同的下标参数。如果有 n 个 unique 调用，那么 unique 的数值范围是 0~(n-1)。

```
AppOneC.Timer->HilTimerMilliC.Timer[unique("tmpTimer")];
AppTwoC.Timer->HilTimerMilliC.Timer[unique("tmpTimer")];
```

参数不同的 unique()函数调用可能返回相同的整数值，也可能返回不同的整数值。例如：

```
AppOneC.Timer->HilTimerMilliC.Timer[unique("tmpTimer")];
AppTwoC.Timer->HilTimerMilliC.Timer[unique("myTimer")];
```

这两行中 unique()函数的参数分别为字符串“tmpTimer”和“myTimer”，因此可能返回相同

的参数下标。

`uniqueCount()`也以字符串作为参数。如果以相同字符串为参数调用了 n 次 `unique()`函数，则 `uniqueCount()`返回的值是 n 。

有了 `unique()`和 `uniqueCount()`函数，用户就不用预先确定参数化接口所需要的参数个数，这样既不必担心参数个数不够，也不必担心参数个数过多而浪费存储空间。

3.6.6 Default 函数

配件的作用是将组件连接起来，因此确定了模块之间的调用点。如果在配件中由于疏忽少写了某个连接语句，那么就会带来编译错误。这相当于在 C 中忘记包含一个库。另外，如果在程序中存在一个悬着的连接，那它将成为一个 Bug。例如，组件 `ActiveMessageC` 的 `Receive` 接口。大多数应用中只包括少数几个 AM 类型，一般来说最多 15 个，也就是说，在该程序下，不是对每个协议标识号都有相应的网络协议的，例如，没有相应的上层应用发送 AM 类型字段值为 144 的分组。但是，这个分组有可能是一个具有合法 AM 值的分组发生错误而形成的，而巧合的是这种错误也通过了分组的校验。对于这样的分组没有对应的处理程序。

为了避免上述问题，`nesC` 提供了 `default` 处理程序（函数）。当某个接口函数没有相应的实现时，就使用这个函数。如果某个组件连接到该接口，就使用该组件的实现；否则就会调用 `default` 处理程序。

例 3.29: `CC2420ActiveMessageP` 具有如下的 `default` 处理程序

```
default event message_t* Receive.receive[am_id_t id](message_t* msg, void* p, uint8_t len){
    return msg;
}
default event message_t* Snoop.receive[am_id_t id](message_t* msg, void* p, uint8_t len){
    return msg;
}
default event message_t* AMSend.sendDone[uint8_t id](message_t* msg, error_t err){
    return msg;
}
```

值得说明的是，使用 `default` 函数也可能会带来 Bug。因为对于 `default` 函数，编译程序不检查组件接口的连接情况。比如，组件 `CC2420ActiveMessageP` 中有默认函数 `Send.send Done`，在 `TestAMC` 中不应有 `Send.send` 的默认处理程序，否则，如果忘记连接 `TestAMC.Send`，整个应用编译是无错的，但运行时并不发送分组。

`default` 函数应该仅用于这样的场合：对于正常执行的组件来说，某接口是多余的。这几种情况几乎总是与参数化接口相关，因为接口的所有参数值被使用的概率都非常小，通常会有一些参数值的接口没有被使用，对于这样的接口，需要 `default` 函数来处理。

3.7 通用化组件

`nesC 1.2` 中增加了通用组件和通用接口，这是 `TinyOS 2.x` 与 `TinyOS 1.x` 的主要区别。通

用接口是在普通的接口上引入了类型参数，在 3.3.3 小节已经介绍过。现在我们来了解一下通用组件。通过本小节的学习，我们将会了解到为什么需要通用组件，通用组件的作用和好处，通用组件的定义以及在编程过程中如何使用通用组件。

3.7.1 为什么需要通用组件

默认情况下，组件是单一的，即在全局范围内只有一个，任意配件都可以直接使用。某单一组件（如 `LedsC`）在不同的配件中声明，但对应的都是 `LedsC`。具体的，如果两个配件都连接了 `LedsC`，他们连接的是相同的代码，访问的是相同的变量。

`nesC 1.0` 和 `1.1` 中没有支持通用组件。任何时候组件都需要一个通用的数据结构，编程者必须复制数据结构组件，给它一个新的名字；或者是分割所需通用数据结构的功能，并通过本地分配数据结构和使用库程序来分配功能。

除了单一组件，`nesC 1.2` 还支持通用组件。通用组件可以多次实例化。例如，硬件的底层软抽象是一个单一组件，即只有硬件寄存器的一个备份。但是软件数据结构是可以实例化的。实例化使得组件代码可以重用。以组件 `BitvectorC` 为例，`BitvectorC` 组件提供了比特向量的抽象，在多个组件中都需要使用比特这个抽象。如果它是单一组件，每个比特向量都是一个具有不同名字的组件，也就是说，需要多次定义、多次实现。如果比特向量是通用组件，就可以对它进行一次定义、多次使用。

为了实现通用组件的重用，通用组件实例化使用代码复制的机制。代码复制可以带来 2 个好处：简单和匹配类型。如果通用模块没有使用代码复制的方法，则会有一个单独的代码复制，这个单独的代码复制作用于所有组件实例。`nesC` 仅仅创建代码的副本。所有的副本都是从一个单一的源文件得来，所以它们是一致的，不会出现多个源文件所导致的维护问题。但复制代码的方法降低了减小代码大小的可能性。

为了实现代码重用，通用组件实例化使用代码复制的机制。这不仅简化了实例化工作，还可以为通用组件提供类型等参数。

首先，如果通用组件没有使用代码复制的方法，那么在组件的所有实例中都会对应一个单一代码复制。如果通用组件采用了类型参数，如分配大小、偏移量等，将会使得每个实例的单一代码复制方法变得更加复杂。即便通用组件采用非类型参数时也会导致这样的问题，这是因为非类型参数可能会用来指定数组大小，转换 `case values` 等。

其次，如果采用通用组件的各个实例可共享代码的方式，就需要给各实例的函数添加一个类似于指针作用的参数，指明在执行哪个实例，但相应的各实例中的变量访问均需避开这个参数，因此，执行时间和函数成本会有大的改变。

综上，为了提供简单、易于理解和高效运行的组件，`nesC` 在通用组件实例化时使用代码复制的机制。

3.7.2 通用组件的定义与实例化

1. 通用组件的定义

组件分为模块和配件，相应的，通用组件也分为通用模块和通用配件。通用组件的定义中要使用关键字 `generic`。另外，在语法方面与单一组件不同的地方是，通用组件有一个参数列表。参数列表跟在组件名后面，用圆括号括起来。一般来说，通用组件支持三种类型的参

数：类型、数字常量和字符串常量。如果参数是类型，则使用关键字 `typedef` 声明。这些参数可以在多处使用，比如，组件规范中的通用接口的类型、接口函数中的参数或返回值类型，模块变量的定义或声明等。参数列表也可以为空。

通用模块与通用配件的一般形式：

<pre>generic module 模块名{ //声明接口 } implementation{ // 通用模块实现 }</pre>	<pre>generic configuration 配件名{ //声明接口 } implementation{ // 通用配件实现 }</pre>
---	--

例 3.30：有参数的通用组件

```
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 {
    ...
}
```

组件 `VirtualizeTimerC` 是一个通用模块。它有两个参数：第一个参数是计时器精度的类型，是用关键字 `typedef` 声明的。这种精度类型提供了额外的对接口类型的检测，因为这个参数在提供和使用的通用接口 `Timer` 中被用于指定接口的精度类型；另一个定义了组件所提供的最大的虚拟定时器的数量，它的值通常由编译函数 `uniqueCount()` 计算得到。

例 3.31：参数列表为空的通用配件

```
generic configuration TimerMilliC() {
    provides interface Timer<TMilli>;
}
implementation {
    ...
}
```

组件 `TimerMilliC` 是一个通用配件，没有携带参数，也就是说，参数列表为空。注意：即便通用配件没有参数，配件名字后面的圆括号也不能省略。

2. 通用组件的实例化

使用通用组件需要先实例化它。实例化通用组件实际是复制组件代码。组件代码复制就是逐字地创建源文件的副本。代码复制作用于通用模块时，是将可执行代码和变量拷贝到最后的应用中。代码复制作用于通用配件时，表现为实例化其他组件并创建连接。也就是说，通用模块定义了重复的可执行逻辑，通用配件定义了组件组织关系的重复模式。

实例化通用组件使用关键字 `new`，并需填写传递的实参。

例 3.32:

```
generic module BitVectorC( uint16_t max_bits ) {
    provides interface Init;
    provides interface BitVector;
}
configuration ExampleVectorC {}
implementation {
    components new BitVectorC(10);
}
```

以上代码创建一个向量长度大小为 10bit 的通用模块 `BitVectorC`，该语句实际上是创建 `BitVectorC` 代码的副本。在创建的通用模块 `BitVectorC` 的新实例的代码中，由 10 代替参数 `max_bits`。

通用组件只有在实例化它的配件中被命名，对于实例化该通用组件的配件而言，它是私有的，其他组件不可以直接使用它。上述例子中，在配件 `ExampleVectorC` 中实例化了通用化模块 `BitVectorC`，因此，`BitVectorC(10)` 对于配件 `ExampleVectorC` 是私有的，其他组件不能访问它。那么如何使用实例化的通用组件呢？可以使用我们先前介绍的方法，即输出通用组件实例的接口。

例 3.33：一个位向量负责跟踪系统是否正在运行，多个组件都需要访问这个位向量。`BitVectorC` 是通用组件，配件 `SystemServiceVectorC` 实例化 `BitVectorC` 后，将其接口输出，以便于其他组件使用。

```
configuration SystemServiceVectorC {
    provides interface BitVector;
}
implementation {
    components MainC, new BitVectorC(uniqueCount(UQ_SYSTEM_SERVICE));
    MainC.SoftwareInit -> BitVectorC;
    BitVector = BitVectorC;
}
```

上述 2 个例子都只对通用组件 `BitVectorC` 实例化 1 次，即只需要一个实例，这种情况可以使用关键字 `as` 设置别名，也可以不用。但是，如果在配件中需要通用组件的多个实例，实例化时需使用关键字 `as` 设置别名。

例 3.34：在 `Blink` 应用中，使用通用组件 `TimerMilliC` 的三个实例，为了对这三个进行区分，需要使用关键字 `as` 设定别名。其代码可参考例 3.18。

3.7.3 通用化模块

通用模块是可以使用诸如类型、常量等通用参数的模块。比如下面 `QueueC` 的例子中，通用模块的形参类型 `queue_t` 像标准的 C 类型一样，可以用于声明数组（队列）的类型、接口函数返回值的类型等。参数 `QUEUE_SIZE` 就像 C 常量一样，定义模块中的数组 `queue`（队列）的大小。

例 3.35: QueueC

```
generic module QueueC(typedef queue_t, uint8_t QUEUE_SIZE) {
    provides interface Queue<queue_t>;
}
implementation {
    queue_t ONE_NOK queue[QUEUE_SIZE];
    ...
    command queue_t Queue.head() {
        return queue[head];
    }
}
```

1. 类型参数

声明通用组件要检查类型。默认情况下，允许对类型参数的操作只有声明变量和赋值。

例：类型参数的操作。

下面两条语句是合法的。

```
queue_t x, y;
x=y;
```

如下的操作是不合法的。

```
queue_t x, y;
x=y+1;
```

然而，如果需要的话，可以在类型参数后添加后缀@integer()。属性@integer()允许对该类型参数进行整数操作。

例：在类型参数后添加后缀@integer()以允许整数操作。

在通用模块 ConstantSensorC 中，在类型参数 width_t 后使用了属性@integer()，表明可以进行整数方面的操作。该类型参数用于指定通用接口 Read 的类型，进一步的，在通用模块的实现代码中，该类型参数在接口 Read 的通知事件 Read.readDone (SUCCESS, (width_t) val) 中，用该类型参数对读取的感知数据 val 进行了类型强制转换，即 val 可以被强制转换为整数类型。

例 3.36:

```
generic module ConstantSensorC(typedef width_t @integer(), uint32_t val) {
    provides interface Read<width_t>;
}
implementation
{
    task void senseResult() {
        signal Read.readDone(SUCCESS, val);
    }
    command error_t Read.read() {
```

```

        return post senseResult();
    }
}

```

实例化通用组件 `ConstantSensorC` 时, `width_t` 必须作为整型类型使用, 下面两条语句中, 第一条是合法的, 而第二条是不合法的。

```

components new ConstantSensorC( uint16_t, 23) as C1;      // 合法
components new ConstantSensorC( struct X, 39) as C2;      // 编译出错

```

另外, 也有属性 `@number()` 可以作为后缀, 这表示类型参数是一个整型或浮点型。对该类型参数只可进行整型操作或浮点型操作。 `@number()` 和 `@integer()` 表示 nesC 语句的属性。

2. 通用抽象数据类型

在 TinyOS 中, 有些数据类型, 比如队列、比特向量以及通信消息等都是较为通用的, 也就是说, 往往多个组件都要使用。这些组件对这些通用数据类型可能的需求也不尽一致, 比如, 对于队列来说, 不同的组件可能需要使用不同类型、不同最大长度的队列。如何以一种较为方便的方式来使用这些通用数据类型呢?

通用模块是一种有效的方式, 具体的做法是: 使用模块变量来存放通用数据状态; 使用接口中的命令来访问这些通用数据状态。通过上述的通用模块 `QueueC` 例子, 不难理解这一点。在通用模块的变量 `queue` 中存放队列的状态, 为了对队列进行访问, 可以使用通用接口 `Queue<queue_t>`, 该接口中提供了一系列命令 (访问操作), 如取队列头元素 `head()`、入队 `enqueue()`、出队 `dequeue()` 等等。

3.7.4 通用化配件

连接时, 组件实例不用带参数。

模块是一个包含可执行代码的组件, 配件定义了组件之间的关系, 形成更高层次的抽象; 通用模块是可重复使用的执行代码, 因此, 通用配件是组件之间的重复模式, 形成了一个更高级别的抽象。nesC 也有通用配件, 这个通用配件是在建设 TinyOS 抽象和服务上的一个非常强大的工具。通用配件也是用关键字 `generic` 声明的。

在深入研究通用配件之前, 我们来看一下没有通用配件时的代码。

例 3.37: HilTimerMilliC。这是一个非常重要的组件, 但程序中从来没有直接命令它, 但是定时器实例化通用组件的一个应用被命名为 `TimerMilliC`, 后面我们将会讲到 `TimerMilliC`。 `HilTimerMilliC` 是定时器服务的核心部分。许多组件需要定时器, `HilTimerMilliC` 通过它的参数化接口提供了这种能力。 `HilTimerMilliC` 封装 `VirtualizeTimerC`, 其入参取值是 `unique(UQ_TIMER_MILLI)` 的返回值。这意味着如果一个组件 `AppP` 需要一个定时器, 则它的配件 `AppC` 需要这样写:

```

configuration AppP {...}
implementation {
    components AppP, HilTimerMilliC;
    AppP.Timer -> HilTimerMilliC.TimerMilli[unique(UQ_TIMER_MILLI)];
}

```

而实际中 AppP 需要三个定时器，所以代码应为：

```
configuration AppP {...}
implementation {
    components AppP, HilTimerMilliC;
    AppP.Timer1 -> HilTimerMilliC.TimerMilli[unique(UQ_TIMER_MILLI)];
    AppP.Timer2 -> HilTimerMilliC.TimerMilli[unique(UQ_TIMER_MILLI)];
    AppP.Timer3 -> HilTimerMilliC.TimerMilli[unique(UQ_TIMER_MILLI)];
}
```

这种方法可以很好地了解 TimerC 在 TinyOS 1.x 中是如何工作的。但它存在一些问题。首先，有 UQ_TIMER_MILLI 散置在系统中的许多个组件中，导致使用的标识符不太可能改变，因而调用一个错误的但有效的参数 unique() 将不会返回 error。例如，如下组件：

```
AppP.Timer1 -> HilTimerMilliC.TimerMilli[unique(UQ_TIMER_MICRO)];
```

其次，有时会有两个组件都连接到定时器的同一个实例，这会导致使用 unique 的定时器系统内部执行暴露给其他组件。通常情况下，所有的组件都想分配一个新的计时器，但它不关心是如何实现的。

通用配件通过定义一个可以实例化的连接结构来简化这个问题。例如，所有定时器的使用者都想连接一个毫秒定时器，而且它们不必担心 unique 函数的入参到底是什么。

下面是一个通用配件的例子：TimerMilliC。

例 3.38: TimerMilliC。 TimerMilliC 提供了毫秒定时器这种抽象。TimerMilliC 是一个提供单一定时器接口的通用配件。它的实现语句完成的工作是：将提供的定时器接口连接到一个实例，这个实例使用关键词“UQ_TIMER_MILLI”作为函数 unique 的入参，并以函数 unique(UQ_TIMER_MILLI) 的结果来参数化定时器接口 TimerMilli 的参数。这意味着 unique() 只在一个文件中被调用，只要所有的组件用 TimerMilliC 分配定时器，就不会存在错误。TimerMilliC 的执行非常简单：

```
generic configuration TimerMilliC() {
    provides interface Timer<TMilli>;
}
implementation {
    components TimerMilliP;
    Timer = TimerMilliP.TimerMilli[unique(UQ_TIMER_MILLI)];
}
```

其中，TimerMilliP 是一个单独的配件，自动连接 HilTimerMilliC 到启动顺序，并输出 HilTimerMilliC 的参数化接口：

```
configuration TimerMilliP {
    provides interface Timer<TMilli> as TimerMilli[uint8_t id];
}
implementation {
    components HilTimerMilliC, MainC;
```

```

MainC.SoftwareInit -> HilTimerMilliC;
TimerMilli = HilTimerMilliC;
}

```

TimerMilliC 封装了一个连接结构给其他组件使用，这个连接结构调用 unique wiring 到定时器服务。当一个组件实例化 TimerMilliC 时，它创建一个 TimerMilliC 代码的副本，这个副本包括调用 unique(UQ_TIMER_MILLI)。实例化 TimerMilliC 的代码如下：

```

components X, new TimerMilliC();
X.Timer -> TimerMilliC;

```

在本质上是：

```

components X, TimerMilliP;
X.Timer -> TimerMilliP.TimerMilli[unique(UQ_TIMER_MILLI)];

```

TimerMilliP 在本质上是一个配件，连接到 HilTimerMilliC 配件。当一个组件在 TimerMilliC 上调用 Timer.start() 时，它调用的实际功能是 VirtualizeTimerC 上的 Timer.start()。

下面我们来看一个使用通用化配件 TimerMilliC() 的应用实例 BlinkAppC。

例 3.39: BlinkAppC。BlinkAppC 连接 BlinkC 模块到它的三个定时器。其代码可参考例 3.19。

连接 Blink.Timer0 到 Timer0，建立如下连接链：

```

BlinkC.Timer0 -> Timer0.Timer
Timer0.Timer = TimerMilliP.TimerMilli[unique(UQ_TIMER_MILLI)]
TimerMilliP.TimerMilli[unique(UQ_TIMER_MILLI)] = HilTimerMilliC[unique(UQ_TIMER_MILLI)]
HilTimerMilliC[unique(UQ_TIMER_MILLI)] = VirtualizeTimerC.Timer[unique(UQ_TIMER_MILLI)]

```

BlinkC 和 VirtualizeTimerC 是两个模块，中间的组件都是配件。当 nesC 编译这段代码时，所有的中间层将被优化，BlinkC.Timer().start 将会在 VirtualizeTimerC.Timer[...].start 上开始直接函数调用。

3.7.5 属性 attributes

属性（attributes）是将元数据与程序语句关联的一种方式。nesC 的属性基于 Java 注释中采取的方法。这里主要介绍最常用的属性及如何使用这些属性，完整的细节不再详述。

属性声明是指一个数据结构名字前有 “@” 的声明。属性具有域，域可以被初始化。前面已经接触了属性 @combine。

```

struct @combine {
    char *function_name;
}
typedef uint8_t error_t @combine("encombine");

```

组合属性将字符串作为参数，error_t 的声明由 ecombine 来注释，指定了 error_t 的组合函数名字。圆括号中的参数实际使用了与 C 初始化属性定义相同的语法。

用户定义的属性可以附加到 nesC 程序中的大多数名字（组件、接口、typedef、变量和

函数等) 上。

例 3.40:

```
struct @atleastonce{};
configuration LedsC{
    provides interface Init @atleastonce();
    provides interface Leds;
}
```

这个例子声明了 `atleastonce` 属性, 该属性中没有域。配件 `LedsC` 使用这个属性注释其接口 `Init`。默认情况下, `@atleastonce` 什么都不做。但是 `nesC` 编译器具有工具可以将应用的信息输出, 包括属性信息。TinyOS 编译过程中有一部分工作是运行一个脚本, 检查连接限制, `atleastonce` 是其中之一。编译中检查有 `atleastonce` 注释的接口被连接至少一次。

属性没有引入新的关键字提供了扩展 `nesC` 语言的方式。目前系统中的通用属性包括如下几项。

@spontaneous: 表明函数可以从 `nesC` 程序外部调用, 因此, 由 `nesC` 死代码删除中不应该剪除这个函数。一般在中断处理程序或者连接二进制库 (比如, `TOSSIM`) 时要使用该属性。

@C: 表明函数为 C 函数, 而不是 `nesC` 函数。特殊的, 使用该属性定义的函数不是组件内部私有的函数。在 C 代码需要调用 `nesC` 代码时使用该属性。该属性隐含 `@spontaneous`。

@hwevent: 表明函数是硬件中断调用的结果。

@atomic_hwevent: 表明函数是中断调用的结果, 且函数运行具有原子性。该属性隐含 `@spontaneous`。在本属性与 `@hwevent` 的区分是必需的, 这样 `nesC` 可以知道是否需要额外的原子语句。

@atmostonce: 表明接口必须连接 0 或 1 次;

@atleastonce: 表明接口必须连接 1 或多次;

@exactlyonce: 表明接口必须连接 1 次, 不能多也不能少。

@integer: 表明通用组件类型参数必须是整数。这个属性允许通用组件对该类型参数使用整数操作。

@number: 表明通用组件类型参数必须是整数或者浮点类型。该属性允许通用组件对该类型参数使用算术操作。

@combine: 声明类型时, 用于指定类型的组合函数。它有 1 个参数, 即组合函数名字的字符串。

第 4 章 并发执行模型

TinyOS 中的 I/O 操作都是分阶段的，不都通过线程。TinyOS 程序在单个程序栈上执行。代替线程，TinyOS 程序的所有代码是由一个任务或者中断处理程序执行的。TinyOS 中的任务实际上是一个轻量级的延期调用：TinyOS 调度程序可在任意时间发布任务，所发布的任务将会在一定时间后被执行。TinyOS 中的中断可发生在任何时间，中断任务或者中断其他处理程序（当中断被禁止时除外）。

在一个特定的模块中声明一个任务或中断处理程序时，当调用一个命令或者给事件一个信号，这个任务或中断处理程序可能会跨组件执行。如图 4-1 所示。

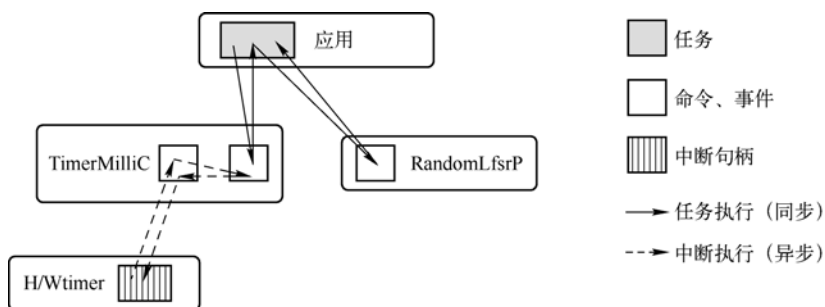


图 4-1 TinyOS 执行模型：任务和中断处理程序跨组件执行

由于中断处理程序执行的代码可以更加明确地意识到并发问题（因为它可能在任何时候被调用），nesC 中区分同步（sync）代码和异步（async）代码，同步（sync）代码仅由任务执行，异步（async）代码可被任务和中断处理程序执行，并且异步执行的命令或事件需用关键字 `async` 声明。

4.1 任务

任务是由 nesC 提供的一个简单的延期计算机制，任务有一个 `void` 的返回值，不带参数（不带参数可以保持任务调度的简单性），并且用关键字 `task` 声明。

```
task void setupTask () {  
    // task code  
}
```

使用关键字 `post` 为稍后的执行调度一个任务：

```
event void Boot.booted () {
```

```

    call Timer.startPeriodic (1024);
    post setupTask ();
}

```

在上例中，一旦 TinyOS 完成初始化序列（信号初始化，初始化所有组件），它将运行 setupTask()。

和其他模块的代码类似，任务可以访问变量，调用命令，给事件发送信号，调用内部或全局的 C 函数。任务的调度操作就如同调用一个返回值为 `error_t` 的函数。发布一个任务总是返回 `SUCCESS`，除非该任务被挂起。从另一个角度说，一个组件不能同时调度同一个任务的多个备份。但是，一个任务一旦开始运行，它可以再次调度它自己。

下面的代码是在系统初始化后运行的 TinyOS 核心调度循环，这个函数没有返回值。它执行任务直到任务队列为空，然后让微控制器进入睡眠状态。

```

command void Scheduler.taskLoop () {
    for (;;) {
        uint8_t nextTask;
        atomic {
            while ((nextTask == popTask ()) == NO_TASK) {
                call McuSleep.sleep ();
            }
            signal TaskBasic.runTask[nextTask ]();
        }
    }
}

```

这段代码的伪代码是：

```

一直执行：
    当没有任务时：
        睡眠
    执行下一任务

```

TinyOS 一直处于睡眠状态直到中断唤醒处理器为止。如果中断处理程序发布一个或者多个任务，TinyOS 将会一直执行到没有任务可执行，然后进入睡眠状态。任务是一个接一个执行的，不可抢先占用，直到当前任务执行完成，下一个任务才可执行。任务的这种属性极大地简化了代码编写的任务量，所有的同步代码不需要任何机制来保护共享变量。同步例程假设直到它完成前，它已经完全控制处理器。

以 BaseStation 应用为例。BaseStation 连接 UART 串口和无线通信模块。BaseStation 把所有通过无线信道接收到的数据包发送给 UART，并把所有接收到的 UART 数据包发送到无线通信模块。因为无线和 UART 可能有不同的吞吐量，BaseStation 在每个方向都有一个发送队列。当它从无线信道接收到一个数据包时，它把该数据包置于无线模块到 UART 的发送队列中。当它接收到一个 UART 数据包时，它把该数据包置于 UART 到无线模块的发送队列中。

BaseStationP 是执行 BaseStation 应用的模块，它使用任务从队列中提取数据包并发送数

据包。接收处理器把数据包放入队列，并在应用不发送数据包时调度 send 任务。其伪代码如下。

```
接收数据包：
  若队列非空：
    将数据包放入队列
  若当前没有要发送的数据
    提交任务 send Task
```

任务不带参数。如果一个组件需要传递一个数据给任务，则这个数据必须存储在这个组件的变量中。例如，BaseStationP 不能将一个指向消息的指针作为参数传递给任务 sendTask。相反，sendTask 必须直接从队列中提取数据包。

4.1.1 任务的执行

TinyOS 通常都是以“先进先出”的顺序运行任务。一个任务经调度后，只有当前面的所有任务都运行完成后该任务才会运行。这意味着每个任务应该很短。如果一个组件有一个很长的计算，则应该把它分成多个任务。由于任务运行后就不会再存在于队列中，因此为反复运行一个任务，任务可以调度自身。

在当前的 mote 平台上，大约需要 80 个微控制器时钟周期来调度和执行一个任务。一般来说，应保持任务的运行时间最多为几个毫秒。因为一个任务运行结束后，一个长时运行的任务或者大量非长时运行的任务可导致任务调度和执行之间的较大的时延（几十毫秒）。

现考虑两种情况，这两种情况都有五个处理组件和一个无线协议栈。mote 处理器以 8MHz 的速率运行。每个处理组件需要占用大量的 CPU 时间。情况一，处理组件调度任务运行 5ms 后重新调度自己来继续工作。情况二，处理组件调度任务运行 0.5m 后重新调度自己来继续工作。

在第一种情况下，任务调度开销是 0.02%：执行 40 000 个周期中有 80 个周期是用做开销。在第二种情况下，任务调度的开销是 0.2%：执行 4000 个周期中有 80 个周期是用做开销。所以执行完成的时间是没有明显差异的。然而，考虑到任务排队时延，在第一种情况下，当无线协议栈调度一个任务，通知已接收到一个数据包，预计它将大约等待 25ms（5 个处理任务×每个任务 5ms 时延）时，就限制了系统每秒最多处理 40 个数据包。在第二种情况下，当无线协议栈调度一个任务，预计它将大约等待 2.5ms（5 个处理任务×每个任务 0.5m 时延）时，就使得系统每秒可处理 400 个数据包。任务调度代价低，使用大量的短运行时间的任务可提高系统响应，而不会增加 CPU 的开销。

当然，这里还必须对短运行时间任务的数量和组件中状态数量进行折中。例如，假设要加密一个数据块。如果加密操作需要一段时间（如 10ms），可把它分裂成多个任务来执行，以提高整个系统的响应。然而，如果按一个任务来执行，要为任务分配所需的所有状态和暂存空间。相反，分裂任务需要在组件中保持任务的状态和暂存空间。这里没有折中的硬性规定，但总体来说，长时间运行的任务可导致操作系统的其他部分出现问题，因此应尽可能地避免。

4.1.2 执行和事件处理

短运行时间任务的需求直接影响组件的实现，特别是事件处理程序。**BaseStationP** 在它的接收事件处理程序中不直接发送数据包，而是通过调度任务来发送数据包。通过调度任务来发送数据包需要很多周期，在这个发送完成前，底层无线组件都不会从应用中得到一个新的缓冲区。一般来说，如果接收处理程序中有大量的计算，无线模块必须等待这个接收完成，才会获得接收下一个数据包的缓冲区。

一个简单的事件处理程序不可能完成复杂的处理，但是，一个事件处理程序实际上可能跨多个软件层运行。例如，一个网络组件可以处理一个基于数据包的计算量较小的接收事件，然后给下一层发送一个信号。因此，任何一个给定的组件，可能只是一个长事件处理程序的一部分。所以，如果处理程序需要进行大量的计算，最好的方法就是调度任务。

这就是 **BaseStationP** 为什么使用任务而不是直接在事件处理程序中发送数据包的原因。虽然在理论上任务可能要等待一段时间才运行，但在实践中任务往往很短，所以在调度和执行之间有一点很小的延迟。

4.2 任务和分阶段调用

任务不仅仅为保持系统的响应提供了一种方法，还使 **nesC** 程序有了一个灵活的硬件、软件分界线，软件组件可以有和硬件组件类似的行为。

4.2.1 硬件与软件

分阶段调用描述了大量外围设备是如何工作的。软件给设备发出命令，并在一段时间后用一个中断表明设备操作完成。设备驱动程序的中断处理程序发送一个信号给操作完成事件。当设备忙时，处理程序可以继续发出指示并完成其他工作。因此，启动操作的命令可立即返回，并允许程序继续运行。

对于硬件实现，关键点是中断处理程序调用了驱动程序代码。但是，当这种实现是一种纯软件形式时，应如何操作呢？**SineSensorC** 是一个纯软件传感器，它用于计算正弦值。当我们将 **SineSensorC** 和一个硬件传感器互换时，它需要提供相同的分阶段接口。当一个组件调用 **Read** 命令时，**SineSensorC** 需要通知 **readDone** 执行下一个读操作。

4.2.2 任务和调用循环

回顾 **Read** 操作，大多数传感器接口可为用户提供生成的传感器读数。**Read** 操作有一个命令 **read** 和一个单独的事件（**readDone**）。若一个传感器有噪声，为了尽量滤掉一些噪声，应用程序需要一个简单的滤波器组件，指数加权移动平均（EWMA）平滑原始读数。

```
module FilterMagC {
  provides interface StdControl;
  provides interface Read <uint16_t>;
  uses interface Timer <TMilli>;
```

```

    uses interface Read <uint16_t> as RawRead;
}
implementation {
    uint16_t filterVal = 0;
    uint16_t lastVal = 0;
    error_t StdControl.start () {
        return call Timer.startPeriodic (10);
    }
    command error_t StdControl.stop () {
        return call Timer.stop ();
    }
    event void Timer.fired () {
        call RawRead.read ();
    }
    event void RawRead.readDone(error_t err, uint16_t val) {
        if (err == SUCCESS) {
            lastVal = val;
            filterVal *= 9;
            filterVal /= 10;
            filterVal += lastVal / 10;
        }
    }
    command error_t Read.read () {
        signal Read.readDone(SUCCESS, filterVal);
    }
}

```

驱动程序以 10ms 周期采样磁力计，并利用 EWMA 平滑读数。当应用通过调用 `Read.read` 来采样已滤波的值时，`FilterMagC` 仅发送给 `Read.readDone` 缓存的已滤波的值。

一方面，这种方法非常简单和快速；但另一方面，它也可导致堆栈中的重大问题。例如，如果 `FastSamplerC` 组件想要多次快速地采样传感器数据，它将调用 `Read.readDone` 处理程序中的 `Read.read`。

```

event void Read.readDone(error_t err, uint16_t val) {
    buffer[index] = val;
    index ++;
    if (index < BUFFER_SIZE) {
        call Read.read ();
    }
}

```

如果因为某种原因，应用程序将 `FastSamplerC` 连接到 `FliterMagC`，将会在 `read` 和 `readDone` 中直接有一个循环调用。如果编译器不能优化函数调用，将会引起堆栈显著增长。给定的 `motes` 仅有有限的 RAM 并且没有硬件的内存保护，快速增长的堆栈会损坏数据内存，并导致程序崩溃。

当然，利用上述例子中 `Read` 操作采集高频率的信号是不合理的。对于一个缓存值，

多于一次的采样是没有任何帮助的。在请求完成通知事情中发出新的请求是一种常见的调用模式。

在 FilterMagC 中该如何给 readDone 事件发送一个信号呢？它需要对随后调用的函数进行调度，正确的方式是使用任务。下面是数据滤波器组件的代码。

```
module FilterMagC {
  provides interface StdControl;
  provides interface Read <uint16_t>;
  uses interface Timer <TMilli>;
  uses interface Read <uint16_t> as RawRead;
}
implementation {
  uint16_t filterVal = 0;
  ... unchanged ...
  task void readDoneTask () {
    signal Read.readDone(SUCCESS, filterVal);
  }
  command error_t Read.read () {
    post readDoneTask ();
    return SUCCESS;
  }
}
```

当 FilterMagC 的 Read.read 被调用时，FilterMagC 调度 readDoneTask 并直接返回。一段时间后，TinyOS 运行该任务，该任务用滤波器值作为参数发送给 Read.readDone。

同步代码几乎可以满足所有的应用层程序编写的需要。但是，在高性能应用程序和底层驱动程序中，有时会要求额外的功能和并发模式。在下一部分将主要介绍两种额外的机制：异步代码和资源锁。异步代码是 nesC 语言的一个特点，资源锁是 TinyOS 的一组组件和机制。

4.3 异步代码

任务允许软件组件来模拟硬件的分阶段行为，但是它们的作用不仅仅如此，它还提供了一种管理系统优先权的机制。由于任务相互之间在运行上的原子性，如果程序代码仅在任务中运行，程序代码就会十分简单，在一个任务运行时，没有另一个任务突发执行和修改数据的风险。然而，中断的存在破坏了任务运行上的原子性，它打断当前的执行，并以更高的优先级开始运行。

4.3.1 关键字 async

在 nesC 语言中区分同步（sync）和异步（async）代码。中断处理程序中的命令和事件具有更高的优先级，因此相对于任务来说是异步的。它们必须用关键字 async 在接口和模块中进行声明。

命令和事件可以在中断处理程序中抢先运行，但这必须要求同时在声明命令和事件的接

口、执行命令和事件的模块中用关键字 `async` 声明。基于此，异步的命令或事件，或者他们所指向的函数，只能调用或通知异步的命令和事件（如果打破了该规则，`nesC` 会有提示）。此规则使我们可以清楚地看到，在模块或接口中哪个代码是同步的，哪个是异步的。

例 4.1: `Send` 接口是同步的，它的命令和事件没有用关键字 `async` 声明

```
interface Send {
    command error_t send(message_t* msg, uint8_t len);
    event void sendDone(message_t* msg, error_t error);
    command error_t cancel(message_t* msg);
    command void* getPayload(message_t* msg);
    command uint8_t maxPayloadLength(message_t* msg);
}
```

例 4.2: `Leds` 接口是异步的

```
interface Leds {
    async command void ledOn ();
    async command void ledOff ();
    async command void ledToggle ();
    ... more commands declared with async ...
}
```

所有中断处理程序都是异步的。因此，在调用时它们不能包含任何同步函数。在中断处理程序中执行一个同步函数的唯一方法是发布（`post`）一个任务。在异步代码中发布任务是允许的，但是发布的任务应在同步代码中执行。

例如，考虑 `UART` 之上的数据分组层。当 `UART` 接收到一个字节，它发出一个中断信号。在中断处理程序中，软件从数据寄存器中读取字节并把它置于缓冲器中。当接收到数据分组的最后一个字节时，软件需要通知接收数据分组。但 `Receive` 接口中的 `receive` 事件是同步的。因此在最后一个字节的中断处理程序中，组件发布一个任务通知接收数据分组。

4.3.2 异步的代价

先考虑一个问题：如果任务引入延迟，为什么使用它们？为什么不一直使用异步代码？原因很简单：竞争，尤其是数据竞争。执行优先级的基本问题是它能修改当前运算的状态，这可能导致系统进入不稳定的状态。

例 4.3: `toggle` 命令，它翻转状态比特并返回该值

```
bool state;
async command bool toggle () {
    if (state == 0) {
        state = 1;
        return 1;
    }
    if (state == 1) {
        state = 0;
    }
}
```

```

    return 0;
}
}

```

现在来看这个执行，从 `state=0` 开始：

```

toggle ()
state = 1;
-> interrupt
toggle ()
state = 0
return 0;
return 1;

```

在这个执行中，当第一个 `toggle` 返回，调用组件认为 `state=1`。但最后在中断后 `state=0`。

如果允许对单个语句执行中断，这个问题更为严重。例如，在 Micaz 或者 Telos 节点上，读或写 32 位的数字需要多个指令。当在两个指令之间中断语句时，这样所读到的数据一部分是新值，而另一部分是旧值。

数据竞争中最突出的问题是状态变量问题。举个例子，设想 AMStandard 中的一小段代码，在这里状态变量指示组件是否忙。

例 4.4:

```

command result_t SendMsg.send ... {
    if (! state) {
        state = TRUE;
        // send a packet
        return SUCCESS;
    }
    return FAIL;
}

```

如果这个命令是异步的，在条件 `if(!state)` 和 `state=TRUE` 间可能插入另一个组件，同样试图发送数据分组。第二个调用会看到 `state` 是 `FALSE`，然后会设定 `state` 为 `TRUE`，并开始发送数据，返回 `SUCCESS`。但第一个调用者的结果会使 `state` 再次设定为 `TRUE`，同时开始发送数据，返回 `SUCCESS`。这两个数据包中只有一个能发送成功，除非在调用路径上有额外的错误检测，否则很难发现是哪个出现错误，这可能给调用的组件带来各种各样的错误。为了避免这个问题，发送命令不是异步的。

注意：编程时应尽量使用同步代码。只有当程序代码对运行时机的掌控有较高要求或者应用在对运行时机的掌控有较高要求的场景时，才使用异步代码。

4.3.3 原子语句和关键词 `atomic`

中断引入了一个问题：程序需要一种方法来执行代码片段并使代码片段不会被抢占。nesC 语言通过 `atomic` 语句提供了这种功能。

例 4.5:

```
command bool increment () {
    atomic {
        a++;
        b = a + 1;
    }
}
```

`atomic` 程序块允许变量可以在原子状态下进行读和写。实际上，在当前所有的 `mote` 平台上，`atomic` 语句是通过禁止中断来实现原子性的。理论上，一个 `nesC` 的执行允许 `incrementA` 和 `incrementC` 在代码片段中互相抢占对方，这是因为它们访问不同的变量。

例 4.6:

1)

```
async command bool incrementA () {
    atomic {
        a++;
        b = a + 1;
    }
}
```

2)

```
async command bool incrementC {
    atomic {
        c++;
        d = c + 1;
    }
}
```

然而，在实际执行时 `nesC` 是棘手的，因为它开销大、与 I/O 设备交互困难。因此，应假设 `atomic` 语句禁止中断。

`nesC` 中 `atomic` 语句用于检查变量是否有合适的保护，并在没有合适保护的情况下发出警告。例如，在例 4.6 中若 `b` 和 `c` 没有在 `atomic` 语句中，`nesC` 将会因为可能的自我抢占而发出警告。如果一个变量是由异步函数访问的，那这个变量必须受 `atomic` 语句保护。如果一个不包含 `atomic` 语句的函数经常被有 `atomic` 语句的函数调用，则编译器不会发出警告。

虽然可以通过在代码中使用 `atomic` 语句使数据竞争不被发出警告，但这么做时应考虑以下两个方面。一方面，围绕 `atomic` 语句禁止中断和使能中断会带来 CPU 开销，因此应最小化禁止和使能中断的次数。另一方面，较短的 `atomic` 语句使中断有更少的时延，这样可改善系统的并发性。确定 `atomic` 语句的运行时间是一个棘手的问题，特别是在一个组件不得不调用另一个组件时。

例 4.7:

```

atomic {
    if (! call ArbiterInfo.inUse () ) {
        stopSpi ();
    }
}

```

ATmega128 SPI 总线的实现有一个资源仲裁器对总线访问进行管理。仲裁器允许不同的用户请求资源（总线）并在所请求的资源被授予时通知它们。然而，SPI 总线不希望明确仲裁策略（如先到先服务或者基于优先级），所以它必须绑定一个仲裁器。这种分解涉及能量管理。SPI 在没有用户时自行关闭，不需要调用仲裁器。这意味着 SPI 必须用 `atomic` 语句来查看是否调用仲裁器，若没有，则自行关闭。

这种情况下，调用 `inUse` 所需时间应该非常短（实际上它可能只是读一个状态变量）。如果某一个应用绑定了仲裁器，它的 `inUse` 命令耗时 `1ms`，那么就可能带来一些问题。实现过程中，通常假定调用 `inUse` 命令几乎不耗时，在例 4.7 中就做了这样的假定。

4.3.4 状态切换的管理

在一个组件中，`atomic` 语句最基本的用途是状态切换。通常，一个状态切换包括两部分：第一是转换为一种新状态，第二是采取某种动作，二者都取决于现有状态和调用。让我们再来看看 `AMStandard`：

```

if (! state) {
    state = TRUE;
    // send a packet
    return SUCCESS;
}
else {
    return FAIL;
}

```

如果状态涉及异步函数，则需使用 `atomic` 来切换状态。但是，在编程时不希望整块代码都变成 `atomic` 语句，因为这样发送一个数据分组可能需要较长的时间，从而导致系统错过中断。所以应对代码进行如下修改。

```

uint8_t oldState;
atomic {
    oldState = state;
    state = TRUE;
}
if (! oldState) {
    // send a packet
    return SUCCESS;
}
else {

```

```

    return FAIL;
}

```

若 `state` 已经是 `TRUE`，上述程序不影响已设定好的 `TRUE`。上述语句比 “`if (state != TRUE) {state = TRUE ;}`” 节省 CPU 时间。

在这个例子中，状态切换发生在 `atomic` 块中，但实际处理过程发生在 `atomic` 块外。

4.3.5 CC2420ControlP 组件的工作

组件 `CC2420ControlP` 是 `CC2420` 射频堆栈的一部分。`CC2420ControlP` 负责配置射频的各种 I/O 选项以及射频的打开和关闭。打开 `CC2420` 射频分四步：

- 1) 打开稳压器 (0.6ms)；
- 2) 获取 SPI 总线到射频 (时间取决于竞争总线的时间)；
- 3) 总线上发送命令来开启射频的振荡器 (0.86ms)；
- 4) 将射频置于 `RX mode` (0.2ms)。

需要时间的一些步骤 (特别是 1) 和 3)) 是分阶段的，并有异步执行事件。但开始这一系列事件实际上是调用同步的 `SplitControl.start`。实施这些步骤的一种方法是给每一步分配一个状态，并使用一个状态变量来跟踪目前位置。然而，射频关闭过程不是必要的，一旦启动顺序开始，它会一直继续直到完成。所以在这里所需要的状态变量仅仅是是否开始。开始之后，每个完成事件都隐含为该状态的一部分。例如，`startOscillatorDone` 事件暗示射频处于状态 3)。因为 `SplitControl.start` 是同步的，状态变量可以不使用 `atomic` 语句来修改。

```

command error_t SplitControl.start () {
    if ( m_state != S_STOPPED )
        return FAIL;
    m_state = S_STARTING;
    m_dsn = call Random.rand16 ();
    call CC2420Config.startVReg ();
    return SUCCESS;
}

```

`startVReg` 命令启动稳压器。这是一个异步命令。事件完成后，射频将竞争 SPI 总线。

```

async event void CC2420Config.startVRegDone () {
    call Resource.request ();
}

```

在事件完成 (占用了总线) 后，它会发送一个命令来启动振荡器。

```

event void Resource.granted () {
    call CC2420Config.startOscillator ();
}

```

最后，振荡器接收到完成信号，组件告诉射频进入 `RX` 模式，同时发布一个任务来通知 `startDone` 事件。因为 `oscillatorDone` 是异步的，所以它必须发布一个任务。注意：组件此时释放了总线。

```

async event void CC2420Config.startOscillatorDone () {
    call SubControl.start ();
    call CC2420Config.rxOn ();
    call Resource.release ();
    post startDone_task ();
}

```

最后，任务改变射频的状态从 STARTING 到 STARTED。

```

task void startDone_task () {
    m_state = S_STARTED;
    signal SplitControl.startDone( SUCCESS );
}

```

另一种执行可能已经把以下代码放入 startOscillatorDone 事件中。

```

atomic {
    m_state = S_STARTED;
}

```

这样做唯一可能的好处就是射频可以在理论上更早地接收到请求。但是由于组件在 startDone 事件接收到信号前不应该调用射频，所以这可能带来一点问题。当改变状态和发送信号给事件均在 startDone_task 中时，没有另一个任务可以介入这二者之间。

4.3.6 任务的再次调用

在执行模型中介绍了任务是推迟运算并保持 nesC 程序响应的一种方法。然而，任务另一个额外的关键作用是关于异步代码的。任务是一个组件可以从异步转换为同步的唯一方法。在接口中可编写任务，其代码如下。

```

interface TaskBasic {
    async command error_t post ();
    event void run ();
}

```

所有包含一个异步命令和一个同步完成事件的分阶段接口必须使用一个任务。

在分阶段接口中包含一个同步命令和一个异步事件的情况非常少。这样的接口仅可被任务调用，但是这要求用户处理中断。由于事件是异步的，任何命令和事件之间共享的状态必须用 atomic 语句来保护。因此，将一个命令设为异步不如在事件中直接调用命令。

4.4 功率锁

atomic 语句允许组件在没有中断干扰的情况下执行代码块。然而，底层驱动程序需要在没有系统其他部分干扰的情况下在单个硬件资源上执行一系列分阶段的操作。这些驱动程序不能使用 atomic 语句，因为驱动程序的许多执行步骤需要在不同函数中完成，这显然不能在一个 atomic 语句中完成。此外，长时间地禁止中断以执行分阶段操作不是一种好做法。

在这些情况下，原子属性不应在处理器上而应该在硬件资源本身上实现。为了编写这样的驱动程序，TinyOS 采用了功率锁，它可被用来获取一个特定的硬件资源的独占访问。除了管理并发性，功率锁也可用于能量管理和帮助配置硬件。虽然在应用程序中不可见，功率锁是 TinyOS 内部构成的重要组成部分，并对大多数底层系统的工作非常重要。

4.4.1 功率锁需求实例——链路层确认

链路层的数据分组通信就是功率锁的一个例子。TinyOS 链路层协议栈通过 Packet Acknowledgements 接口支持确认。当一个节点接收到一个发送给该节点的数据分组时，它会立即发送一个小数据分组来进行响应，对接收进行确认。在发送端，堆栈在一小段时间内等待链路层确认，并报告是否接收确认信号。最大限度地减少发送端的等待时间是提高性能的关键，所以接收机要尽快发送确认信号。

从编程的角度来看，射频堆栈的接收部分需要经过以下几步：

- 1) 读取无线信道上的数据分组；
- 2) 检查数据分组，看它是否应该发送确认；
- 3) 转变射频为发送模式；
- 4) 发送确认；
- 5) 返回射频接收模式。

以上每个步骤都是单独的分阶段操作。例如，在 CC2420 射频上，每个操作需要通过 SPI 总线发送一个命令来读取或写入数据。然而，这个 SPI 总线是多个芯片和子系统共享的，在节点接收数据分组时，闪存驱动器可能需要在同一时间使用 SPI 总线来读取数据。我们不能依赖于 SPI 总线组件来调度多个外部请求，因为它可能试图在两个射频堆栈操作之间插入一个较大的 flash 写操作。我们需要一个方法来使射频堆栈请求独占访问总线，这样它可以快速地执行这五个操作，然后释放总线以供给其他部分使用。

4.4.2 分阶段锁

TinyOS 通过分阶段锁支持对资源的独占访问。传统上，诸如互斥器和信号量的锁采用封锁结构，用于保护关键部分或共享数据结构。然而，由于 TinyOS 没有封锁调用，所以 TinyOS 的锁必须是分阶段的。组件调用一个命令来请求功率锁，并在获取锁的时候接收一个事件。锁接口被命名为 Resource。

```
interface Resource {  
    async command error_t request ();  
    async command error_t immediateRequest ();  
    event void granted ();  
    async command void release ();  
    async command uint8_t getId ();  
}
```

为了获得一个锁，组件通常调用 request。一段时间以后，它将接收到一个 granted 事件，通知它能够使用锁保护下的所有东西。命令 immediateRequest 是一个优化，它允许一个组件在单个阶段的操作中获取空闲锁。如果 immediateRequest 返回 FAIL，那么该锁已经被

保留，组件必须调用 request。组件在释放命令时释放锁。

对锁提出需求的抽象提供 Resource 接口。未拥有锁的情况下调用功能接口是被禁止的，因为这可能导致程序错误和系统问题。例如，这里有一个抽象节点，在该节点上 MSP430 微处理器提供 SPI 总线。

```
generic configuration Msp430Spi0C () {
    provides interface Resource;
    provides interface SpiByte;
    provides interface SpiPacket;
    uses interface Msp430SpiConfigure;
}
```

同样，下面是 ADC 的抽象：

```
generic configuration Msp430Adc12ClientC () {
    provides {
        interface Resource;
        interface Msp430Adc12SingleChannel;
        interface Msp430Adc12MultiChannel;
        interface Msp430Adc12Overflow;
    }
}
```

4.4.3 锁的内部结构

TinyOS 功率锁管理并发性、能量和共享硬件资源配置。功率锁有三个子组件：

- 一个仲裁器，控制锁策略；
- 一个能量管理器，控制能量策略；
- 一个或多个配置器，配置客户端硬件。

图 4-2 显示了三部分的结合的方式。仲裁器是控制中心。一个仲裁器接收客户端的锁请求，并基于请求调用能量管理器和配置器。锁使用参数化接口和 unique() 维持一个锁的等待请求队列。每个锁客户端有唯一的锁 ID，锁在执行时使用 uniqueCount() 来维护一个适当长度的队列。

4.4.4 能量管理

TinyOS 不仅使用锁建立独占访问，同时也使用锁来决定何时开关外围设备。以 SPI 总线为例，微控制器通常有数个低功耗状态，这些低功耗状态的区别在于芯片子系统和中断资源的不同激活状态。典型的最低功耗状态是一个节点只能响应一个特定的硬件计数器或外部中断资源的唤醒，系统中诸如总线和 ADCs 处于休眠状态，不进行操作。因此，如果 TinyOS 可以在不使用 SPI 时关闭 SPI 总线，即可达到节省能量的目的。

锁提供了一种简单的方法来进行能量的自动管理，减轻了程序员的负担。当锁被释放掉并且没有等待处理的请求时处于空闲状态，此时锁可以关闭被它保护的系统的电源。当锁在空闲状态下接收一个请求时，锁切换到忙状态，此时可以打开系统电源。根据底层硬件的性

质，锁可以在断电前等待一小段时间或者立即关闭电源。

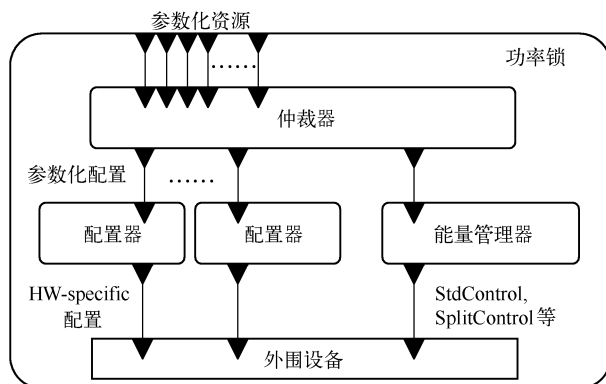


图 4-2 功率锁的子组件的结合方式

功率锁支持存在一个“默认所有者”，这个“默认所有者”是一个在锁空闲时占用锁的组件。功率锁的能量管理策略由默认所有者负责。由于默认所有者从不请求锁，它使用一个与 Resource 稍有不同的接口。当有等待处理的请求时，仲裁器总是授予请求者一个空闲锁，并通知默认所有者。响应请求时，默认所有者给硬件加电，一旦加电成功便释放锁。

```
interface ResourceDefaultOwner {
    async event void granted ();
    async command error_t release ();
    async command bool isOwner ();
    async event void requested ();
    async event void immediateRequested ();
}
```

4.4.5 硬件配置

除了控制能量状态，功率锁还可以配置硬件。以微控制器的 ADC 为例，ADCs 通常有数个配置参数，如测量脚管电压并且与参考电压比较。这些参数可由每个用户单独配置，也可以自动进行配置，以简化用户的工作。当一个配置实例化 ADC 用户时，它传送其配置参数给用户组件。然后这些参数传送给绑定仲裁器的配置器。在仲裁器给用户分配锁之前，它调用相应的配置程序来建立所需要的硬件。当用户释放该锁时，仲裁器反向配置硬件并返回到就绪状态。

```
interface ResourceConfigure {
    async command void configure ();
    async command void unconfigure ();
}
```

4.4.6 MSP430 USART 配置

MSP430 USART0（通用同步/异步接收器/发送器）是一个通用的总线，基于多种应用，软件可对其进行配置。USART 可被作为 UART（串口），SPI（串行外围接口），或者

是 I2C（内部集成电路）总线。这三个协议共享一组相同的硬件引脚，并通过软件配置 MSP430 的时序。

软件驱动程序通过实例化一个通用组件（如 Msp430Spi0C，M2p430Uart0C，Msp430I2C0C）来使用总线协议。每个通用组件提供功能接口和一个 Resource 接口，并使用一个特定的硬件配置接口。例如在 SPI 中，用户需要配置总线速度，这允许用户处于配置程序路径中。图 4-3 展示了 MSP430 USART 组件结构。

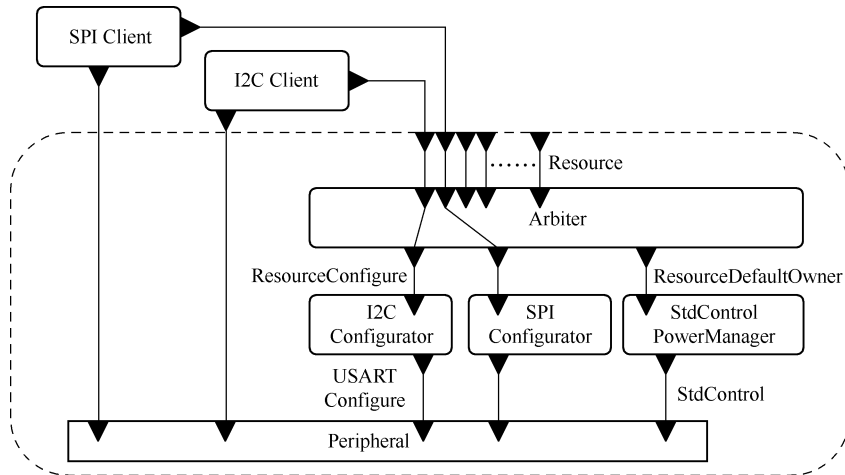


图 4-3 MSP430 USART 组件结构

4.4.7 功率锁库

TinyOS 有一个小的功率锁组件库，在目录 `tos/system` 下，它包括如下内容。

- 两个不同调度策略的仲裁器：FcfArbiterC（先到先服务）和 RoundRobinArbiterC（循环复用）。
- 两个功率管理器：ImmediatePowerManagerC 和 DeferredPowerManagerC。
- 多个配置器。由于配置器基于硬件的不同而有所差别，因此它们可通过基本代码的扩展来获得，大部分配置器可在 `tos/chips` 下找到。

第 5 章 TinyOS 驱动程序与平台硬件抽象

TinyOS 2.x 与 TinyOS 1.x 相比在硬件抽象结构方面有了更加清晰的设计,通过采用 3 层结构的抽象,使得一方面提高了代码的可重用性和可移植性,另一方面也实现了代码执行效率与能耗之间的折衷优化。本章将针对 TinyOS 2.0 具体介绍其硬件抽象架构。此外, TinyOS 独有的集成平台为传感器网络快速开发提供了强大的帮助,本章的第二部分将介绍 TinyOS 平台的概念。

5.1 硬件抽象结构简介

5.1.1 什么是硬件抽象

通常所说的硬件抽象是位于操作系统内核与硬件电路之间的接口,其目的在于将硬件抽象化。它隐藏了特定平台的硬件接口细节,为操作系统提供虚拟硬件平台,使其具有硬件无关性,可在多种平台上进行移植,并简化应用程序的开发过程。举个简单的例子,假设我们希望控制节点上的 LED 亮起,对于顶层应用开发的程序员来说,可能只是简单的调用一个命令 `LightUp()` (该命令是假设的),但是想要通过微控制器控制 LED 就需要了解微控制器的管脚特性,这对于不同的微控制器是不同的,同时还需要知道 LED 与微控制器的哪个管脚相连接等,才能通过编写指令完成 LED 的控制。具体的完成对 LED 控制的代码集一般称之为硬件抽象(驱动程序),这种抽象向上层应用提供一种屏蔽了内部操作细节的可操作指令(在不同的编程语言中有不同的名字,如命令、函数等),这种分层开发的方式使得程序员可以专注于本层功能的实现,减少开发工作量,同时在复杂的系统中也为系统安全性提供了一定的保障。

以上提到的硬件抽象可以封装底层的操作细节,并向上提供可供操作硬件的精简的指令,但是,对于应用开发者而言,依然有一个令人头疼的问题,那就是不同的硬件其抽象出的指令完全不同,编程时需要时常查阅帮助文档。面对这样的问题时,用户希望底层的抽象可以提供一种可读性、通用性非常强的操作指令,甚至是“一键式”的操作。但是,有编程经验的程序员可能都会意识到这会带来另一方面的问题,那就是效率的下降。正所谓鱼和熊掌不能兼得,高集成度的封装会使上层开发变得简单,但同时由于隐藏了过多的细节,使得本来可以很简单解决的问题,由于对底层操作性的降低,执行起来反而很复杂。另一方面,要

对种类繁多的不同硬件提供高集成度的统一的硬件驱动，其工作量也是十分巨大的，这同样会延长开发的周期。

从以上的讨论我们可以看出，硬件抽象应该存在着这样一条线——线以下关注的是硬件操作细节，线以上关注的是应用开发，而这条线究竟应该画在哪里是一个需要认真考虑的问题。这条所谓的线就是本节要讲述的硬件抽象结构。

对于传感器网络的应用而言，硬件抽象结构还需要面对另一个问题，那就是性能和能耗的冲突。因此，如何隐藏硬件复杂的特性，增加代码的移植性，简化应用程序开发的同时在性能和能耗中找到平衡点是作为专为传感器网络设计的嵌入式操作系统 TinyOS 需要解决的问题。

事实上，之前我们提到的那条线在 TinyOS 2.0 中不是一条，而是 3 条。TinyOS 2.0 使用了具有 3 层结构的硬件抽象架构。顶层抽象提供与平台无关的硬件接口，便于代码移植；中间抽象带有相对完善的与硬件相关的接口，有助于提高效率；而底层抽象则与硬件的寄存器和中断紧密相关。

在图 5-1 所示的组件框架里，硬件抽象架构可以分为 3 个不同的组件层。每一层都清楚地定义了各自的功能范围，并依赖于其下层提供的接口。底层硬件的功能自下而上逐渐扩展为操作系统和应用程序之间具有平台无关性的接口。从底层硬件到顶层接口，组件的依赖性越来越弱，从而在设计和实现代码可重用的应用程序时，开发者拥有更多的自由。

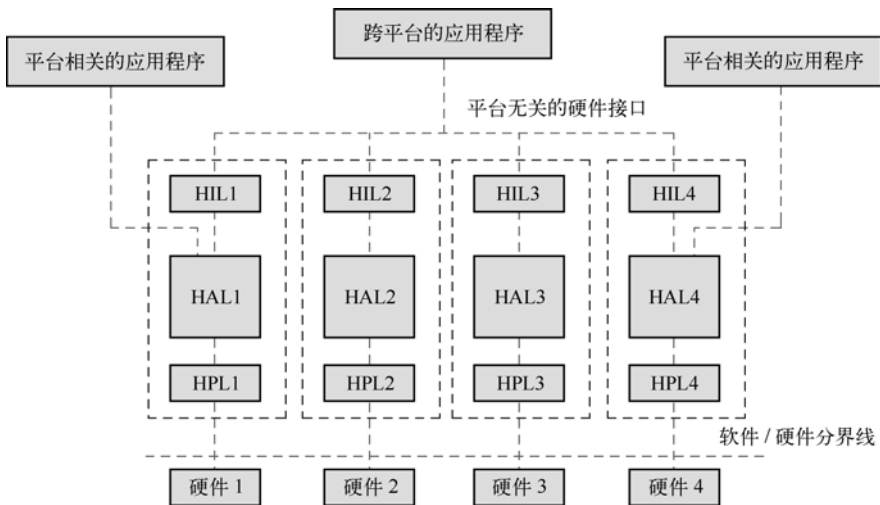


图 5-1 硬件抽象架构

5.1.2 硬件表示层

硬件表示层（Hardware Presentation Layer, HPL）位于硬件/软件的接口之上，也就是说它位于程序开发的最底层，其主要任务就是完成对硬件功能的代码形式的表示。TinyOS 中，组件对硬件进行访问的一般方法是通过内存或者 I/O 映射。另一方面，硬件可以发出中断信号请求服务。通过这些内部的交流渠道，HPL 层隐藏了复杂的硬件接口，并提供了可读性更强的接口。

作为硬件的第一层抽象，HPL 抽象应该为硬件模块向外提供一个功能充分定义的接口，同时，HPL 组件应该与硬件紧密的绑定。这种绑定几乎不会给设计和实现组件留下自由的空间。但即使每个 HPL 组件相对于底层的硬件都是独立的，它们还是有一个大体相似的架构。为了与系统的其他部分优化整合，每个 HPL 应该具有以下功能：

- 1) 提供一组能量管理策略所必须的命令，包括初始化、启动、停止硬件模块。
- 2) 读取和设置硬件寄存器的命令。
- 3) 经常用到的标志位设置和查看的命令。
- 4) 使能和禁用硬件中断的命令。
- 5) 处理硬件中断。

HPL 中的中断服务程序只处理对时间有严格要求的操作，例如拷贝一个值或清除一个标识位等，而将其余的处理交给上层的组件去做。经过硬件抽象之后，上层的程序员就只需要通过一些类似的组件接口就可以访问硬件，而不用再去搞明白那些种类繁多的寄存器名了。

除了自动操作常用的命令序列，HPL 层没有提供任何实质性的硬件抽象。事实上，HPL 并不提供任何对硬件的整体抽象，而只是将相对独立的硬件代码封装起来，向更高一层的抽象组件提供一组可以操作硬件的方法。这些上层的抽象组件在被使用时可能会使用属于同一种类型的不同的 HPL 硬件模块。例如，目前在传感器网节点中普遍使用的微控制器都包括两个用于串口通信的 USART 模块，他们具有相同的功能但却分属不同的寄存器来控制，并产生不同的中断向量。HPL 组件将这些细小的差别封装在统一的接口内部，这样对于上层的编程者来说，切换不同的 USART 只是简单的与 HPL 组件重新连接的问题了，而不用再去对实现代码做任何的改变。

5.1.3 硬件适配层

硬件适配层（Hardware Adaptation Layer，HAL）组件是硬件抽象结构的核心部分，它使用由 HPL 组件提供的初级接口建立相对完善的硬件功能抽象实体，并进一步封装复杂的硬件资源代码。与 HPL 不同的是，HAL 允许维护状态变量，这对于资源控制和资源仲裁而言是很重要的。鉴于传感器网对能量有效性有较高的要求，HAL 的抽象要根据硬件实体进行相应的调整。HAL 并不是将硬件的所有特征都封装进一个通用的接口，而是将一些硬件独有的特征通过接口向外表达，并提供在有限资源状态下的可能最佳的抽象方案。

由于传感器网络对执行效率的要求较高，因此 HAL 层的硬件抽象必须针对于具体的设备类型和平台特征进行设计，而不应该使用隐藏各种硬件特点的单一的抽象模型，虽然这种单一的抽象模型可以进一步简化上层程序开发的复杂度，但同时也会一定程度上降低操作硬件的执行效率。通常建议使用特定领域的硬件抽象模型（如 Alarm、ADC 和 EEPROM 等），而不是对所有设备都使用单一的硬件抽象模型。根据特定的模型，为了实现对硬件抽象的访问，HAL 组件应当使用丰富的、定制的接口，而不是那种通过重载命令隐藏所有功能的标准接口。这样的设计方式使得编译时对接口错误的检查效率更高。

5.1.4 硬件接口层

硬件抽象架构的最上层是硬件接口层（Hardware Interface Layer，HIL）。HIL 组件使用由 HAL 层提供的平台相关的硬件抽象，并将它们整合为可跨平台使用的独立接口。这些独

立接口提供了与平台无关的硬件抽象，从而进一步隐藏了硬件之间的差异，简化了应用软件的开发。

HIL 组件的复杂性主要取决于被抽象化的硬件的功能。具体的来说，当进行传感器网络系统设计时，对底层抽象出的 API 功能会有一个事先的约定，HIL 层就是要满足这一约定，当硬件的功能超过了限度，在 HIL 层就要把 HAL 层的平台相关的硬件抽象“降级”，而当底层硬件的功能比较有限时，HIL 层就可能需要通过软件模拟出缺少的硬件。通过 HIL 层的调整，当底层硬件变动不大的情况下可以较好的维持其向上提供的 API 不变，当然，从另一方面来看，当硬件平台发生较大变化需要修改 API 约定时，反而可能会付出更大的代价。

HIL 接口随着新平台的设计虽然可以逐步演化，但必须确保在演化过程中程序的开销不会超出可承受的范围。为此，TinyOS 设计者引进了 HIL 接口的版本管理机制，给每一代 HIL 接口指定一个版本号。在设计应用程序时，针对先前设备可以使用保留的兼容接口。无线传感器网络往往需要工作很长时间，可能是几年，这就要求有非常合理的版本管理机制来确保程序的性能。此外，HIL 层也可能会出现不同的发展，提供多个功能级别的 HIL 接口。

5.1.5 不同层次抽象的结合

应当指出的是，三层硬件抽象结构是 TinyOS 2.x 的一种建议而非强制执行标准。在 TinyOS 2.0 提供的代码中较好地执行了这种建议，但也有例外。在某些情况下，系统设计者和程序员可能需要为应用程序提供不同级别的硬件抽象，如提供 HIL 层和 HAL 层的硬件抽象。此时，硬件资源可能有多个级别的并行访问。

例如，在 TinyOS 2.0 中的 Oscilloscope 应用（tinyos-2.x/apps/Oscilloscope）就属于这样一种情况。Oscilloscope 使用 ADC 转换传感器采样到的数值，然后把消息通过无线发送出去，考虑到不同平台的兼容性，应用程序使用标准 Read 接口，它由模数转化 HIL 组件提供，并经 DemoSensorC 组件转出，当消息缓冲区收集了足够的采样值，应用程序就把消息传递到无线电堆栈（radio stack）。MAC 协议根据 CCA 信道评估状况确定何时可以安全地发送消息，这就涉及到无线模块发出的 RSSI 信号中提取几个 ADC 模数转换值的问题。由于这是个临界操作，采样值的连续性对这类操作有着重大的影响，所以编写 MAC 协议时可以直接使用 HAL 组件的硬件相关接口，HAL 组件对模数转换过程具有更好的控制能力。图 5-2 显示了如何在 HIL 级别和 HAL 级别并行地访问 MSP430 微控制器的 ADC 硬件。

为了支持这种“纵向”的灵活性，模数转换器的 HAL 组件包含了更复杂的资源仲裁和控制功能。这样，就可以保证对 HPL 层资源实现安全地共享访问。

5.1.6 横向分解

硬件抽象架构除了有纵向的分解之外还有横向的分解。横向分解可以使得不同平台上硬件抽象资源重新利用率更高。因此，TinyOS 2.0 引入了芯片的概念，即为给定芯片设计独立的完整的硬件资源，如微控制器芯片、射频芯片以及闪存芯片等。每个芯片都可以遵循硬件抽象架构的模型进行分解，提供 HIL 的实现部分作为顶层组件。然后，一个平台由各种芯片组件组合在一起。如图 5-3 所示，CC2420 软件依赖于专用的物理定时器，micaz 平台代码就将其映射到 Atmega128 微控制器的定时器上。

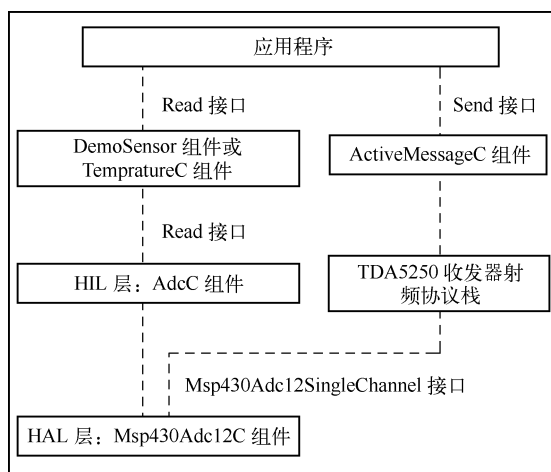


图 5-2 通过 HIL 和 HAL 并行访问 MSP430 的 ADC 硬件抽象

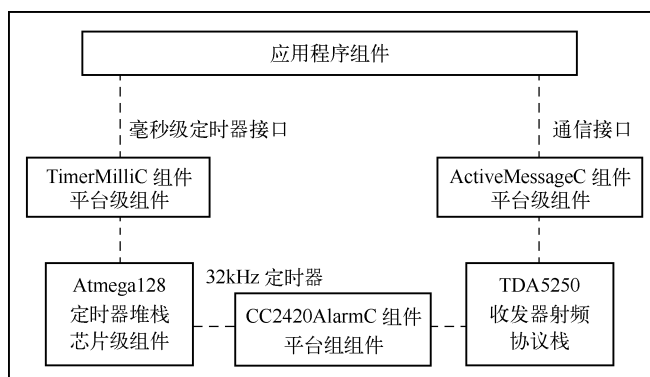


图 5-3 micaz 平台把 CC2420 所需的定时功能连接到 Atmega128 定时器

一些共用的硬件模块是通过标准的总线接口连接到微控制器的，如 SPI 总线、I2C 总线以及 UART 接口等。为了实现硬件驱动的平台兼容性，就需要实现这些互连总线的硬件抽象。显然，类似 NetBSD (<http://www.netbsd.org>) 的通用总线抽象可以最大程度实现这些硬件驱动的可移植性和再利用性。这种模型是从不同的总线协议中提取出一种通用的总线接入方案。以这种方式，把芯片抽象和互连抽象分离开来，这样就可以在一定程度上实现“同一芯片抽象以不同的连接协议引入不同的平台”。然而，这种通用化的实现方式会导致较高的性能开销。这样的做法对于桌面操作系统能够接受，但对于根据特定应用设计的传感器网络平台来说，这种实现方式绝不是最理想的。

TinyOS 2.0 采用较少的通用化方法，提供了一些主要的总线协议的硬件抽象，如 I2C、SPI、UART 以及 I/O 引脚。例如，SPI 总线抽象不必了解用户的地址，而 I2C 总线抽象就需要知道占有 I2C 总线的用户的地址。此外，还可以把这些互连抽象直接接入特定芯片的 HAL 组件，根据芯片的配置选项作细微调整，从而进一步提高性能。

TinyOS 2.0 的总线抽象是与底层 I/O 引脚和引脚中断的抽象相结合的，这样就提高了芯片的硬件抽象在支持该总线协议的平台上的重新利用率。例如，CC2420 射频芯片可以同时

在 Telos 平台和 MicaZ 平台上使用，这是因为 MSP430 和 Atmega128 微控芯片的串行模块的硬件抽象都可以支持标准的 SPI 总线抽象，而 CC2420 芯片使用的就是 SPI 总线。

跨平台的芯片共享也会带来一些问题。当多个芯片设备连接到总线上时，就会导致总线资源的竞争冲突。例如，MicaZ 平台上的 CC2420 芯片连接到专用的 SPI 总线，而 Telos 平台上的一条 SPI 总线被 CC2420 芯片和闪存芯片共用。为了化解冲突，就需要引入资源仲裁机制；总线上的每个芯片抽象为了获得总线资源，都必须使用 Resource 接口。无论是专用总线，还是多个芯片共用的总线，只要基于这种资源仲裁机制，就都可以安全访问芯片。

5.2 TinyOS 平台

5.2.1 TinyOS 平台简介

在 TinyOS 中，无论 1.x 还是 2.x，平台都是一个非常重要的概念。在本书的很多地方读者可以看到类似的编译命令“make telosb”和“make micaz”。这里的“telosb”和“micaz”就是平台的名称。TinyOS 中的平台指的是在确定的硬件设备之上的功能抽象的程序集合。通俗的说，就是加州大学伯克利分校在设计 TinyOS 系统的同时设计了一套硬件设备，该硬件设备包括微处理器、射频处理器、存储器以及传感器等无线传感器网络节点必须的一些功能，并对这些硬件的功能进行了很好的抽象，形成了完整的底层驱动代码，并将这些代码作为 TinyOS 的一个子集打包发布。在完整地安装 TinyOS 之后，用户就获得了这些代码的使用和修改权。

Telosb 和 micaz 就是这样一种平台。编译命令“make telosb”和“make micaz”指明了在当前应用的目录下，该应用底层承接的平台环境。例如读者可以在命令窗口中使用 cd 命令进入“tinyos-2.x\apps\Blink\”目录，并输入命令“make telosb”完成程序编译。

这里的 Blink 是一个与平台无关的应用，它的功能是使节点的 LED 定时闪烁。编译命令指明了该应用承接在 telosb 平台之上。当然，也可以输入“make micaz”等其他命令完成编译。需要指出的是，TinyOS 不仅给出了可以直接使用的程序以简化开发周期，同时也给出了良好的编程示例，通过这些示例可以清晰地看到健壮完整的硬件抽象为上层应用的开发带来的便利，也可以帮助读者更好的理解 TinyOS 的硬件抽象结构。

在“tinyos-2.x\tos\platform\”目录下可以看到“telos”、“telosb”、“mica”等目录，这是硬件平台抽象代码的入口。所谓入口，是指当执行编译命令“make ****”时，编译器会自动根据 make 命令后的平台名称进入对应的平台目录进行进一步编译，但并不是所有与平台相关的代码都存放在平台目录下。这样的作法可能会使读者迷惑：既然是平台代码目录，如果不包含所有与平台相关的代码，那其他的代码存放在哪里？事实上，这就涉及到上节内容谈到的硬件抽象的横向分解。TinyOS 将传感器网络节点适用的微处理器、射频芯片以及传感器等进行了抽象，分别存在“tinyos-2.x\tos\chips”和“tinyos-2.x\tos\sensorboards”目录下。这样的分解模式使得当用户需要在一个硬件平台的基础上扩展一个新的硬件平台时，那些保留的器件的驱动程序可以最大限度地被重用，这大大减小了开发的工作量。

当然，还需要说明的一点是，如果读者在互联网上使用搜索引擎搜索诸如“telosb”、“mica”以及“micaz”等关键词时，得到的搜索结果很可能多数是产品的推广信息。这是因

为以上这些平台对应的硬件节点多数已经由 CrossBow（柯思博）公司进行生产并面向全球发售。这意味着“telosb”这样的最初的实验室概念已经完成了商品化的转换，这当然是一件好事，但可能会让 TinyOS 的初学者产生一个错觉，那就是“telosb”和“mica”指的是实实在在的硬件产品。这样理解固然没有错，但就本书而言，讨论的是 TinyOS 系统中这种硬件节点抽象出的软件平台。之所以强调这一点，是因为市面上销售的例如“telosb”硬件产品对其编程不一定使用 TinyOS，但这种节点依然可以被叫做“telosb”。

5.2.2 新建 TinyOS 平台

上一节简单说明了 TinyOS 系统中的平台的概念，利用集成的平台通常可以快速的搭建针对具体应用的无线传感器网络。当然，由于 TinyOS 中包含的平台功能毕竟有限，很可能不能满足现实应用的需求，此时，当需要一种新的硬件节点时，就相应地需要建立新的平台抽象。本节通过一个简单的示例说明如何新建一个 TinyOS 平台。

在建立新的平台之前，需要给平台取一个名字，并创建存放平台代码的目录。这里将新的平台取名为 **yamp**，在上节中已经介绍过，平台的代码存放在“tinyos-2.x/tos/platforms”目录下，因此，用户可以在命令窗口中输入以下命令：

```
$ cd tinyos-2.x/tos/platforms
$ mkdir yamp
```

1) .platform 文件

每个平台目录下都应该包含一个名为“.platform”，新建的 **yamp** 的目录下也不例外。该文件中包含了该平台的基本编译信息，这个以“.”作为前缀的文件中包含以下内容：

```
push( @includes, qw(

    %T/chips/cc2420
    %T/chips/msp430
    %T/chips/msp430/adc12
    %T/chips/msp430/dma
    %T/chips/msp430/pins
    %T/chips/msp430/timer
    %T/chips/msp430/usart
    %T/chips/msp430/sensors
    %T/lib/timer
    %T/lib/serial
    %T/lib/power
));

@opts = qw(

    -gcc=msp430-gcc
    -mmcu=msp430x1611
    -fnesc-target=msp430
    -fnesc-no-debug
```

```
-fnesc-scheduler=TinySchedulerC,TinySchedulerC.TaskBasic,TaskBasic,TaskBasic,runTask,postTask
);
```

以上代码中的第一条语句是把该平台需要用到的其他代码目录添加到“include”路径下。其中，%T 指代 TOSDIR 环境变量。前文已经提到，平台目录是底层编译的入口目录，因此，需要在这个入口处进一步指明其他需要进行编译的文件的具体路径。

上述代码中的第二条语句使用@opt 包含了传递给 nesC 编译器的一些具体编译参数。

2) hardware.h 文件

除了“.platform”之外，每个平台目录下还需要一个名为“hardware.h”的文件。该文件中定义一些平台相关的常量、管脚名称以及其他头文件等，例如，这个例子中包含了msp430hardware.h。当对该平台的应用进行编译时，该文件会被默认进行编译，无须程序员额外说明。通过输入以下内容建立 hardware.h。

```
#ifndef _H_hardware_h
#define _H_hardware_h

#include "msp430hardware.h"

// LEDs
TOSH_ASSIGN_PIN(RED_LED, 5, 4);
TOSH_ASSIGN_PIN(GREEN_LED, 5, 5);
TOSH_ASSIGN_PIN(YELLOW_LED, 5, 6);

// UART pins
TOSH_ASSIGN_PIN(SOMI0, 3, 2);
TOSH_ASSIGN_PIN(SIMO0, 3, 1);
TOSH_ASSIGN_PIN(UCLK0, 3, 3);
TOSH_ASSIGN_PIN(UTXD0, 3, 4);
TOSH_ASSIGN_PIN(URXD0, 3, 5);
TOSH_ASSIGN_PIN(UTXD1, 3, 6);
TOSH_ASSIGN_PIN(URXD1, 3, 7);
TOSH_ASSIGN_PIN(UCLK1, 5, 3);
TOSH_ASSIGN_PIN(SOMI1, 5, 2);
TOSH_ASSIGN_PIN(SIMO1, 5, 1);

#endif // _H_hardware_h
```

该文件将位于“tos/chips/msp430”下的“msp430hardware.h”中。由于在.platform 文件中已经增加了上述目录，因此此处只需要给出文件名称即可。hardware.h 中还定义了一些管脚的名称。例如说明了红色 Led 与 5.4 号通用 I/O 接口绑定在一起。

该头文件还实现了一些其他的重要功能，例如通过包含 msp430hardware.h，实现了原子语句的中断禁用（nesC 语句中原子语句包括的代码段本质上等同于__nesc_atomic_start() and __nesc_atomic_end()），微处理器的休眠功能等。

此外，每个平台还都必须有一个名为“platform.h”的文件，即使该文件没有任何内容。

3) 配置编译环境

在“tinyos-2.x/apps/null”目录下有一个名为“null”的应用，顾名思义，null 应用是一个不进行任何操作的空应用，它的目的就是为了让平台进行测试。在命令窗口输入以下命令：

```
$ cd tinyos-2.x/apps/Null
$ make yamp
```

会出现以下提示信息：

```
/tinyos-2.x/support/make/Makerules:166: ***

Usage:  make
        make  help

Valid targets: all bnode3 clean eyesIFX eyesIFXv1 eyesIFXv2 intelmote2 mica2 mica2dot
micaz null telos telosa telosb tinynode tmote
Valid extras: docs ident_flags nescDecls nowiring rpc sim sim-cygwin sim-fast tos_image
verbose wiring

Welcome to the TinyOS make system!

You must specify one of the valid targets and possibly some combination of
the extra options.  Many targets have custom extras and extended help, so be
sure to try "make  help" to learn of all the available features.

Global extras:

docs      : compile additional nescdoc documentation
tinysec   : compile with TinySec secure communication

ERROR, "yamp tos-ident-flags tos_image" does not specify a valid target.  Stop.
```

出现以上问题的原因是用户还没有在 TinyOS 系统中定义这个新建的平台，通俗点说就是 TinyOS 系统还不认识这个名为 yamp 的平台。要使系统认识 yamp 这个名字，需要在“tinyos-2.x/support/make/”目录下创建名为“yamp.target”的文件，并输入以下内容：

```
PLATFORM = yamp
$(call TOSMake_include_platform,msp)
yamp: $(BUILD_DEPS)
@:
```

上述代码中第一行指出平台名称为“yamp”，第二行使 yamp 平台包含 msp 处理器的文件（make/msp/msp.rules），最后两行为编译 yamp 平台上的应用程序提供编译规则。

在创建了 yamp.target 文件后回到“tinyos-2.x/apps/null”目录下重新键入 make null，这次将会得到如下的编译信息。

```
~/tinyos-2.x/apps/Null] make yamp
mkdir -p build/yamp
    compiling NullAppC to a yamp binary
ncc -o build/yamp/main.exe -Os -Wall -Wshadow -DDEF_TOS_AM_GROUP=0x7d -Wnesc-all -
target=yamp -fnesc-cfile=build/yamp/app.c -board=    NullAppC.nc -lm
In file included from NullAppC.nc:42:
In component `MainC':
~/tinyos-2.x/tos/system/MainC.nc:50: component PlatformC not found
~/tinyos-2.x/tos/system/MainC.nc:53: no match
make: *** [exe0] Error 1
```

此时，终于得到了一个真正意义上的编译错误，这说明系统已经识别了 yamp 平台。发生错误的原因是由于编译器没有找到“PlatformC”这个组件。由于特定硬件的初始化是由每个平台的 PlatformC 提供的，这是每个平台不能缺少的组件。因此，创建“tos/platforms/yamp/PlatformP.nc”并输入以下内容：

```
#include "hardware.h"

module PlatformP{
    provides interface Init;
    uses interface Init as Msp430ClockInit;
    uses interface Init as LedsInit;
}
implementation {
    command error_t Init.init() {
        call Msp430ClockInit.init();
        call LedsInit.init();
        return SUCCESS;
    }

    default command error_t LedsInit.init() { return SUCCESS; }
}
```

接着创建“tos/platforms/yamp/PlatformC.nc”，并输入以下内容：

```
#include "hardware.h"

configuration PlatformC
{
    provides interface Init;
}
implementation
{
    components PlatformP
        , Msp430ClockC
        ;

    Init = PlatformP;
```

```
PlatformP.Msp430ClockInit -> Msp430ClockC.Init;
}
```

现在，再次执行 `make yamp`，如果依然得到如下的编译错误信息：

```
mkdir -p build/yamp
    compiling NullAppC to a yamp binary
ncc -o build/yamp/main.exe -Os -fnesc-separator=__ -Wall -Wshadow -Wnesc-all -target=yamp -fnesc-
cfile=build/yamp/app.c
-board= -DDEFINED_TOS_AM_GROUP=0x22 -DIDENT_APPNAME="\NullAppC" -DIDENT_
USERNAME="\naoshi\" -DIDENT_HOSTNAME="\ubuntu\"
-DIDENT_USERHASH=0x41acd239L -
DIDENT_TIMESTAMP=0x4d7f00a0L -DIDENT_
UIDHASH=0x5f8c3a6dL NullAppC.nc -lm
/opt/tinyos-2.x/tos/system/tos.h:41:22: error: platform.h: No such file or directory
make: *** [exe0] Error 1
```

通过输入以下命令：

```
$ touch /opt/tinyos-2.x/tos/platforms/yamp/platform.h
```

这次将会得到编译正确通过的信息：

```
mkdir -p build/yamp
    compiling NullAppC to a yamp binary
ncc -o build/yamp/main.exe -Os -Wall -Wshadow -DDEF_TOS_AM_GROUP=0x7d -Wnesc-all -
target=yamp
-fnesc-cfile=build/yamp/app.c -board= NullAppC.nc -lm
    compiled NullAppC to build/yamp/main.exe
        1216 bytes in ROM
        6 bytes in RAM
msp430-objcopy --output-target=ihex build/yamp/main.exe build/yamp/main.ihex
writing TOS image
```

4) 让 Blink 在新建的平台上运行

通过以上步骤，已经完成了新建 `yamp` 平台的基本操作，但更为重要的是，我们需要在该平台上完成应用程序的开发。这里通过掩饰如何让 `Blink` 应用在 `yamp` 平台上工作来说明如何在新建平台上进行应用的开发。

`Blink` 应用是非常简单的程序，只是控制 LED 灯的闪烁。下面进入 `Blink` 所在的目录并运行命令：`make yamp`，此时会得到如下的编译错误信息：

```
    compiling BlinkAppC to a yamp binary
ncc -o build/yamp/main.exe -Os -Wall -Wshadow -DDEF_TOS_AM_GROUP=0x7d -Wnesc-all -
target=yamp
-fnesc-cfile=build/yamp/app.c -board= BlinkAppC.nc -lm
In file included from BlinkAppC.nc:45:
In component `LedsC':
~//tinyos-2.x/tos/system/LedsC.nc:38: component PlatformLedsC not found
```

```

~tinyos-2.x/tos/system/LedsC.nc:42: cannot find `Init'
~tinyos-2.x/tos/system/LedsC.nc:43: cannot find `Led0'

~tinyos-2.x/tos/system/LedsC.nc:44: cannot find `Led1'
~tinyos-2.x/tos/system/LedsC.nc:45: cannot find `Led2'
make: *** [exe0] Error 1

```

从以上信息中可以看出编译器无法在编译组件 LedsC 时找到 PlatformLedsC 组件。通过该组件的名字可以看出这是一个与平台有关的组件，因此需要在 yamp 平台中定义该组件。

为什么一定需要一个与平台相关的组件来链接 Led 呢？这是因为在最底层，led 灯的驱动操作在不同的平台是不一样的，它取决于具体连接的管脚以及执行点亮/关闭操作的输出电平等等。而在 Blink 应用中，程序员并不关注具体的平台，因此，就需要在平台中隐藏底层的操作差异，从而向上提供统一的 Led 操作命令。PlatformLedsC 组件中相关代码如下：

```

#include "hardware.h"

configuration PlatformLedsC {
  provides interface GeneralIO as Led0;
  provides interface GeneralIO as Led1;
  provides interface GeneralIO as Led2;
  uses interface Init;
}
implementation
{
  components
    HplMsp430GeneralIO as GeneralIO
    , new Msp430GpioC() as Led0Impl
    , new Msp430GpioC() as Led1Impl
    , new Msp430GpioC() as Led2Impl
    ;
  components PlatformP;

  Init = PlatformP.LedsInit;

  Led0 = Led0Impl;
  Led0Impl -> GeneralIO.Port54;

  Led1 = Led1Impl;
  Led1Impl -> GeneralIO.Port55;

  Led2 = Led2Impl;
  Led2Impl -> GeneralIO.Port56;
}

```

接下来就可以在 Blink 目录下编译 yamp 平台了，虽然 yamp 是一个虚拟的平台，但可以确定的是这种平台需要工作在具有 MSP430x1611 微处理器的节点上，并且 Led 连接到微处理器的 5.4-5.6 管脚，这与 telos 和 Tmote 节点的连接方式是一致的。

第 6 章 TinyOS 其他主要功能模块分析

TinyOS 中用户通过资源抽象实现对各类资源的访问。TinyOS 中定义了三种类型的资源抽象：专用、虚拟和共享。根据用户需求的不同，不同类型的抽象提供了不同的资源共享机制和能量管理能力。对每种类型的抽象用户必须明确两个问题：如何访问抽象提供的资源和如何控制资源的能量状态。

6.1 资源仲裁

TinyOS 1.x 使用两种机制管理共享资源：虚拟化和完成事件（completion events）。一个虚拟的资源以抽象的一个独立实例出现，例如 TimerC 的 Timer 接口。一个 Timer 实例的用户可独立使用该 Timer 实例，TimerC 将底层硬件时钟虚拟为 N 个独立的定时器。

然而，一些抽象不适合进行虚拟化，程序可能需要使用物理抽象提供的控制。举个例子，TinyOS 1.x 中的组件共享同一个通信协议栈 GenericComm。GenericComm 一个时间只能处理一个发送包。如果一个组件在 GenericComm 处于忙状态时试图发送一个包，那么调用将返回 FAIL。此时，有包需要发送的组件需要采用一种方法获知 GenericComm 什么时候处于空闲状态，以便再次进行发送。TinyOS 1.x 中提供一个全局完成事件，当一个包发送完成后，系统启动完成事件。基于此希望调用 GenericComm 的组件可以处理这个事件并且进行重发。

这种方法对于物理抽象（不是虚拟抽象）来说有几个缺陷。

1) 如果需要提出多个请求，必须对每次提出请求时可能返回的 FAIL 进行处理。这将增加程序的状态，使得程序复杂化。

2) 不能对一个操作序列的运行时机进行控制。这种情况可能会对时机敏感的应用（如 A/D 转换器）产生影响。例如，你需要采取一些措施对 ADC 的使用进行预约，以实现你在期望的时刻你对 ADC 进行操作。

3) 即使一个硬件资源支持预约，也不能通过软件接口完成硬件资源的预约。例如，I2C 总线进行多总线处理时有个“重复开始”的概念，但是在 TinyOS 1.x 的 I2C 抽象中不知道如何使用它。

4) 多数 TinyOS 1.x 服务既没有提供一个非常便利的方法来监测抽象的可用性以便于再次使用抽象，也没有一个文档清楚地说明如何同时提出多个请求。

综上所述，采用一种方式来实现资源共享不能满足所有情况。比如，明确的资源预约可以使程序在访问 A/D 转换器时更好地控制访问时机，然而，当程序不需要严格地控制访问时

机时（如在生物监测应用中测量温度时），资源预约不是一个必需的步骤，并且它将使程序代码更加复杂。此时使用虚拟化的方法更好一些。接下来将介绍资源的分类，一个资源抽象所使用的共享策略是依据该资源的分类确定的。

6.1.1 资源的分类

TinyOS 中定义了三种资源抽象：专用、虚拟和共享。根据抽象的目标和等级，组件提供不同的资源共享机制。注意：HAA 的 Hardware Presentation Layer (HPL) 组件不可能是虚拟的，依据它们的使用情况，HPL 抽象或者是专用抽象或者是共享抽象。

1. 专用抽象

如果子系统需要在整个运行周期中独占一个资源，那么这个资源抽象定义为专用的。由于仅被一个组件占用，这类资源不使用共享策略。中断和计数器是专用抽象的典型例子。专用抽象具有 nesC 语言的 `@atmostonce` 或 `@exactlyonce` 属性，以保证它们在调用时不被干扰。

这种类型的资源通过 `AsyncStdControl`、`StdControl` 或者 `SplitControl` 接口控制能量状态。每个接口的定义可参考 `tinyos-2.x/tos/interfaces`，如下所示。

```
interface AsyncStdControl {
    async command error_t start();
    async command error_t stop();
}
interface StdControl {
    command error_t start();
    command error_t stop();
}
interface SplitControl {
    command error_t start();
    command void startDone(error_t error);
    command error_t stop();
    command void stopDone(error_t error);
}
```

所有专用资源的能量状态都是由这三个接口之一控制的。不管硬件平台提供的物理能量状态有几种，逻辑资源能量状态只有“开”或“关”两个。特定的资源根据用户在不同时间，对资源的物理能量状态所处位置（开或关）的不同需求，对这些接口提供支持。

2. 虚拟抽象

虚拟资源抽象利用软件虚拟化，在多个用户间进行相互屏蔽。每一个虚拟用户都认为自己是一个专用用户，所有虚拟实例复用同一个底层资源。由于虚拟化是由软件完成的，如果不考虑存储和效率的制约，使用同一个资源抽象的用户数可以是无穷。虚拟化通常要求保持资源的状态，因此限制了虚拟实例的数量。虚拟化通常提供一个非常简单的接口给用户。这种简单化付出的代价是降低了效率，同时使得用户不能对底层资源进行精确控制。举个例子，一个虚拟的定时器将带来 CPU 开销，这开销主要用于调度和保持每一个独立的虚拟定时器，同时当两个定时器在同一时间开始计时时还将带来时间抖动。

3. 共享抽象

当资源是仅被一个组件控制时，可使用专用抽象。如果用户愿意付出一点代价并且牺牲一些控制能力，以换取利用简单方法共享一个资源时，可使用虚拟抽象。然而，当多个用户需要准确控制一个资源时，显然这些用户不可能在同一时间同时实现对资源的准确控制，这时就需要一定程度的复用。

共享资源的典型例子是总线。一条总线可连接相当于不同子系统的多个外围设备。例如，在 Telos 平台上，闪存芯片（存储）和射频（网络）共享一个总线。存储和网络协议栈访问总线时，需要排他性地接入到总线，但它们又需要与其他子系统共享总线。在这种情况下，虚拟化是存在问题的。射频协议栈使用总线时需要连续完成一系列的操作，而不是在每次操作前都要重新提出申请。总线作为一个共享资源允许射频协议栈给射频设备自动地发送一系列的操作，而不需要在内存中预先缓存这些操作。

在 TinyOS 2.x 中，由仲裁器负责解决一个共享资源在不同用户间的复用问题。它决定哪个用户在什么时间访问资源。当一个用户占用资源时，它可对资源进行彻底的、不受干扰的占用。仲裁器假设用户之间是相互协作的，用户只有在必要的时候才占用资源。用户完成使用后必须明确释放资源，仲裁器没有办法强制收回资源。

6.1.2 资源的仲裁

每一个共享资源都有一个仲裁器来管理哪个用户能够在给定时间使用这一资源。由于一个仲裁器就是一个控制中心，它能够知道资源是否在使用，因此它总是能够为许多其他的服务提供信息，例如为能量管理提供帮助。如图 6-1 所示，一个仲裁器必须提供一个参数化的 Resource 接口，同时提供一个 ArbiterInfo 接口的实例。Resource 接口由希望访问资源的各个用户进行实例化。ArbiterInfo 接口由希望检索资源状态全局信息（诸如资源是否在使用、谁在使用它，等等）的组件调用。一个仲裁器应该提供一个参数化的 ResourceRequested 接口，同时使用一个参数化的 ResourceConfigure 接口。它还可能提供一个 ResourceDefaultOwner 接口的实例或者任何用于执行特殊仲裁策略的附加接口。对于这些接口更加详细的解释如下：

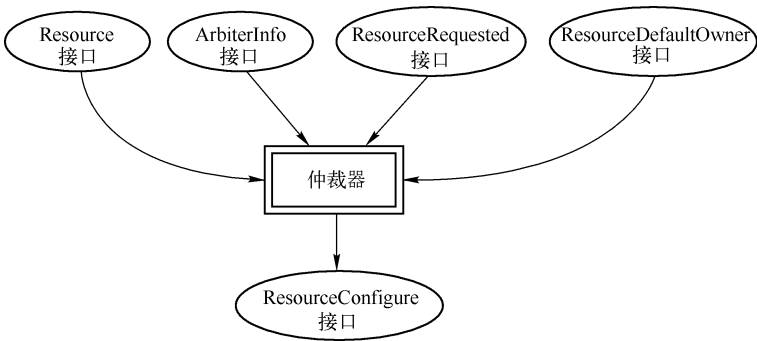


图 6-1 仲裁器接口示意图

1. Resource 接口

仲裁器用户使用 Resource 接口访问共享资源。Resource 接口代码参见本书第 87 页。

用户通过调用 `request()` 通知仲裁器它希望访问某个资源。如果这个资源是空闲的，将返回 `SUCCESS`，同时利用事件 `granted()` 通知用户。如果资源被占用，则仍然会返回 `SUCCESS`，但是该请求将根据仲裁器的排队策略加入队列。用户使用资源完毕后，将调用 `release()` 命令，排在请求队列中的下一个用户通过 `granted()` 事件获得通知，获准访问资源。如果一个用户在接收到 `granted` 事件前提出了多个请求，将返回一个 `EBUSY` 值，同时这些请求不能加入队列。通过使用这种策略，一个用户不能通过提出多个请求的方式独占资源队列，但是，用户仍可以通过不释放资源的方式独占资源的使用权。

用户还可以通过 `immediateRequest()` 命令提出资源使用请求。调用 `immediateRequest()` 可能返回 `SUCCESS` 或者 `FAIL`，利用该命令提出的资源使用请求不会加入请求队列。如果调用 `immediateRequest()` 时返回 `SUCCESS`，则用户在调用返回后可立即访问资源，不用根据 `granted` 事件的通知进行访问。如果返回 `FAIL`，则用户将不被获准访问资源，并且请求也不会进入队列。为使用资源，用户将不得不再次提出请求。

用户可利用 `Resource` 接口的 `isOwner` 命令检测自己是否是资源的当前使用者。该命令通常被用于运行检测，以确保没有占用资源的用户不能够使用该资源。如果调用 `isOwner` 失败，将不能调用资源提供的其他命令。

图 6-2 展示了通过正确地使用仲裁器提供的 `Resource` 接口，利用一个专有资源构建一个简单的共享资源。

一个仲裁器必须提供一个参数化的 `Resource` 接口，其参数应该是一个用户 ID。一个仲裁组件 `SomeNameP` 一定包含 `#define SOME_NAME_RESOURCE`，它对应一个字符串，该字符串能够传给 `unique()` 以获取一个用户的 ID。基于 `nesC` 文件的预处理方式，该 `#define` 必须放在一个单独的文件中。在 `nesC` 中，包括 `SomeNameP` 组件在内的所有组件均不能确保 `SomeNameP` 中的宏定义 `#define` 在所涉及的组件中可见。

2. ArbiterInfo 接口

如图 6-3 所示，仲裁器必须提供一个 `ArbiterInfo` 接口的实例，该接口用于组件查询一个仲裁器的当前状态。

```
interface ArbiterInfo {
    async command bool inUse();
    async command uint8_t clientId();
}
```

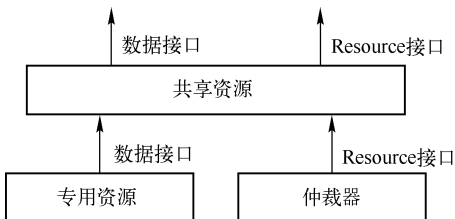


图 6-2 Resource 接口示意图

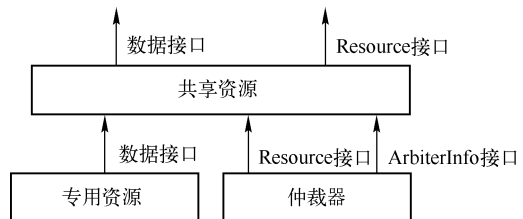


图 6-3 ArbiterInfo 接口示意图

与仲裁器提供的参数化的 `Resource` 接口形成对照，`ArbiterInfo` 接口仅有一种形式。它的作用是查明如下信息。

1) 仲裁资源当前是否被使用。

2) 哪个用户正在使用它。

一个接口希望获得资源的本地访问权时，需要对资源状态进行观察。同时可通过观察 `ArbiterInfo` 获取资源使用情况的全局信息。

`ArbiterInfo` 接口的首要作用是，当用户并非正在访问资源时，允许共享资源拒绝用户通过数据接口对资源进行的调用。

3. ResourceRequested 接口

且如图 6-4 所示，`ResourceRequested` 接口可使一个用户长期占用一个资源，直到有其他用户需要使用该资源。使用 `ResourceRequested` 接口，`ResourceRequested` 的信息对于资源的当前占用者可见。

```
interface ResourceRequested {
    async event void requested();
    async event void immediateRequested();
}
```

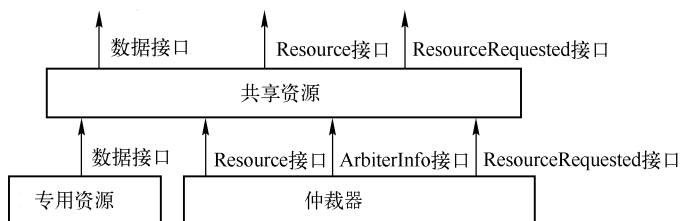


图 6-4 ResourceRequested 接口示意图

如果其他用户通过它的 `Resource` 接口的 `request()` 命令提出了资源使用请求，则资源的当前占用者将通过 `requested` 事件获得通知。如果请求通过 `immediateRequest()` 命令提出，则通过 `immediateRequested()` 事件获得通知。

一个仲裁器应该提供一个参数化的 `ResourceRequested` 接口给它的用户，但并不是必须提供。参数化的 `ResourceRequested` 接口的用户 ID 将与 `Resource` 接口的用户 ID 匹配，以确保所有事件都能通知到正确的用户。

4. ResourceConfigure 接口

如图 6-5 所示，`ResourceConfigure` 接口的存在允许一个用户在获准访问资源之前，对资源进行自动配置。提供 `ResourceConfigure` 接口的组件使用底层专用资源提供的接口，按照运行期望的模式对资源进行配置。一个用户连接它的共享资源抽象到该组件执行期望的配置。用户获准访问资源的同时配置命令被调用，并且反配置命令在资源释放前调用。

```
interface ResourceConfigure {
    async command void configure();
    async command void unconfigure();
}
```

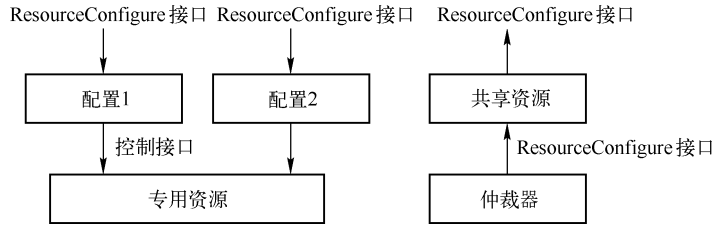


图 6-5 ResourceConfigure 接口示意图

仲裁器将使用一个参数化的 ResourceConfigure 接口，该接口的用户 ID 参数应与参数化的 Resource 接口的用户 ID 一致。如果一个仲裁器使用了 ResourceConfigure 接口，则在执行 Resource.granted() 事件前，它必须根据赋予的用户 ID 调用 ResourceConfigure.configure()。类似的，在正确调用 Resource.release()之后，它必须调用 ResourceConfigure.unconfigure()释放资源。在准许用户访问资源之前调用 ResourceConfigure.configure()，在释放资源之前调用 ResourceConfigure.unconfigure()，可以确保对资源进行重新配置时，资源总是处于未配置状态。

ResourceConfigure 接口中的命令可以作为标准 Resource 接口的一部分（并且转变为返回事件），但是这样做要比保持它们的独立性付出更多代价。在 Resource 接口中引入新的命令将导致大量的用户程序中存在冗余的配置代码。与之对应的是，如果将其作为一个独立的组件，那么用户只有在需要时才对这些代码进行实例化。

5. ResourceDefaultOwner 接口

常规的 Resource 接口用于所有用户以平等的方式共享资源。当一个用户需要拥有其他用户不使用资源时的资源控制权时，该用户使用 ResourceDefaultOwner 接口。一个仲裁器一定不会提供多个 ResourceDefaultOwner 接口实例，它只需要提供一个 ResourceDefaultOwner 接口实例。

```

interface ResourceDefaultOwner {
    async event void granted();
    async command error_t release();
    async command bool isOwner();
    async event void requested();
    async event void immediateRequested();
}
    
```

仲裁器必须保证在 boot 初始化完成前，ResourceDefaultOwner 接口的用户是资源的拥有者。当一个普通资源用户提出资源使用情况时，ResourceDefaultOwner 将接收到 requested() 或者 immediateRequested()事件。然后，它必须决定是否和什么时候释放资源。一旦资源释放，所有提出请求的用户将基于仲裁器使用的排队策略依次访问资源。一旦所有的请求都进行了处理（包括在其他用户访问资源时提出的请求），ResourceDefaultOwner 将自动获得资源控制权，并处理 granted()事件。ResourceDefaultOwner 接口中包含和普通 Resource 接口中相同的 isOwner()命令，并且它们的用法完全相同。

虽然 ResourceDefaultOwner 接口看起来像是普通 Resource 接口和 ResourceRequested 接口的联合体，但是它的用法是完全不同的。ResourceDefaultOwner 接口仅被希望在其他用户

不访问资源时一直占有资源的用户使用。这些用户不主动寻求访问资源，而是利用资源空闲的时间使用资源执行操作。

定义 ResourceDefaultOwner 接口的首要动机是，对于一个资源及其仲裁策略，允许对其能量管理进行简便整合。如果一个仲裁器希望能够通过精确的能量管理策略对资源进行控制，它可以提供 ResourceDefaultOwner 接口给组件，以执行上述能量管理策略。只要资源处于空闲，能量管理组件就将接收 granted()事件，并且将资源的能量状态置于关。当其他用户请求使用资源时，能量管理器将通过 requested()或者 immediateRequested()事件获得通知。然后，它将资源的能量状态置为开，并且在执行上述步骤后释放资源。注意，如果能量打开是一个独立操作阶段（执行需要一些时间），那么在资源能量处于关状态时，用户调用 immediateRequest()将返回 FAIL。

6.1.3 跨组件预约

在一些情形下，可能需要实现跨组件资源预约。例如，在 TI 的 MSP430 中，一个 USART 组件可作为一个 I2C 总线、UART 或者 SPI 线路使用。非常明显，在该芯片上，I2C 总线的预约暗含着对 UART 和 SPI 服务的限制。使用跨组件预约策略需要满足以下条件。

- 1) 为使用共享资源的每一个服务生成一个唯一的 ID。
- 2) 将这些 ID 映射到底层资源的 ID 上。

这种映射对于使用服务的用户来说是透明的。只有在两个不同的用户使用同一个服务时，才进行仲裁。例如，在 MSP430 中，一个 I2C 总线用户可能与一个 SPI 连接的用户竞争资源，但是它们或许有相同的服务级用户 ID。为了共享 USART 组件，这两个服务级用户 ID 映射为两个唯一的资源 ID。完成该映射的正确方法是使用通用组件。下面给出的例子展示了在 MSP430 中 SPI 组件是如何执行该映射的。UART 和 I2C 总线的做法类似。

例 6.1:

```
#include "Msp430Usart.h"
generic configuration Msp430Spi0C() {
    provides interface Resource;
    provides interface SpiByte;
    provides interface SpiPacket;
}
implementation {
    enum { CLIENT_ID = unique(MSP430_SPIO_BUS) };
    components Msp430Spi0P as SpiP;
    Resource = SpiP.Resource[ CLIENT_ID ];
    SpiByte = SpiP.SpiByte;
    SpiPacket = SpiP.SpiPacket[ CLIENT_ID ];
    components new Msp430Usart0C() as UsartC;
    SpiP.UsartResource[ CLIENT_ID ] -> UsartC.Resource;
    SpiP.UsartInterrupts -> UsartC.HplMsp430UsartInterrupts;
}
```

MSP430_SPIO_BUS 字符串在 Msp430Usart.h 中定义。利用该字符串在每次一个新的 Msp430Spi0C 组件实例化时生成一个唯一的 ID。这个 ID 作为 Msp430Spi0P 组件提供的参数化 Resource 接口的参数使用。Msp430Spi0P 组件在提供一个参数化 Resource 接口 (Msp430Spi0P.Resource) 的同时, 还使用一个参数化的 Resource 接口 (Msp430Spi0P.UsartResource)。用户利用 ID 号 CLIENT_ID 调用组件提供的 Resource 接口时, 相当于利用相同的 ID 实现了 Msp430Spi0P.Resource 接口的底层调用。通过 ID 号 CLIENT_ID 连接 Msp430Spi0P.UsartResource 接口到一个实例化的 Msp430Usart0C 组件提供的 Resource 接口可完成映射。任何针对由实例化 Msp430Spi0C 组件提供的 Resource 接口的调用都将通过一个唯一的由底层 Msp430Usart0C 组件提供的 Resource 接口实现。

当服务需要明确知道当前它所提供服务用户数量时, 使用上述映射就显得十分重要, 这些服务不受底层共享资源所拥有的用户总量的影响。举个例子, 一个使用底层 ADC 资源的传感器实现可能希望在它自身没有用户时关闭电源, 而不是等到整个 ADC 都空闲时才关闭电源。上述映射允许 TEP115 中描述的潜在的能量管理组件连接到两个等级的抽象上, 它们相互之间不会产生干扰。采用这种方式, 组件的执行存在许多相似之处, 并且可进行代码重用。

6.1.4 执行

因为大多数组件使用有限的几个仲裁策略中的一个, tinyos-2.x 中包括了一定数量的默认资源仲裁器。在 tinyos-2.x/tos/system 中可找到这些仲裁器, 并且所有的通用组件使用如下两个模块中的一个, 其代码如下:

1)

```
generic module SimpleArbiter {
    provides interface Resource[uint8_t id];
    provides interface ResourceRequested[uint8_t id];
    provides interface ArbiterInfo;
    uses interface ResourceConfigure[uint8_t id];
}
```

2)

```
generic module Arbiter {
    provides interface Resource[uint8_t id];
    provides interface ResourceRequested[uint8_t id];
    provides interface ResourceDefaultOwner;
    provides interface ArbiterInfo;
    uses interface ResourceConfigure[uint8_t id];
}
```

“简单”仲裁器主要由不要求额外开销的资源使用, 额外开销主要是由于提供 ResourceDefaultOwner 接口而产生的。

在许多情况下, 改变仲裁策略主要是改变排队策略, 排队策略用于决定资源请求的获准

顺序。在这种方式下，将排队策略实现代码从现行的仲裁实现代码中分离出来可提高代码重用度。通过使用 ResourceQueue 接口，SimpleArbiterP 和 ArbiterP 组件可被连接到提供具体排队策略的组件上去。

```
interface ResourceQueue {
    async command bool isEmpty();
    async command bool isEnqueued(resource_client_id_t id);
    async command resource_client_id_t dequeue();
    async command error_t enqueue(resource_client_id_t id);
}
```

下面是使用 ResourceQueue 接口连接一个“先进先出（FCFS）”排队策略到 SimpleArbiterP 组件的例子。

例 6.2:

```
generic configuration SimpleFcfsArbiterC(char resourceName[]) {
    provides {
        interface Resource[uint8_t id];
        interface ResourceRequested[uint8_t id];
        interface ArbiterInfo;
    }
    uses interface ResourceConfigure[uint8_t id];
}
implementation {
    components MainC;
    components new FcfsResourceQueueC(uniqueCount(resourceName)) as Queue;
    components new SimpleArbiterP() as Arbiter;
    MainC.SoftwareInit -> Queue;
    Resource = Arbiter;
    ResourceRequested = Arbiter;
    ArbiterInfo = Arbiter;
    ResourceConfigure = Arbiter;
    Arbiter.Queue -> Queue;
}
```

当一个资源的用户希望以 FCFS 方式处理资源请求时，可以实例化这个通用配置。下面是 tinyos-2.x 中提供的所有默认排队策略以及仲裁组件。

排队策略：

- FcfsResourceQueueC。
- RoundRobinResourceQueueC。

仲裁器：

- SimpleFcfsArbiterC。
- FcfsArbiterC。
- SimpleRoundRobinArbiterC。
- RoundRobinArbiterC。

仲裁器仅仅提供了一个管理用户 ID 的标准方法,以达到共享资源不必在每次执行时都复制它自身的功能。为了限制不提出请求而使用资源的用户,共享资源必须使用 `ArbiterInfo` 接口提供的功能,对资源的当前所有者执行运行时间检测。

6.2 微控制器能量管理

微控制器通常有多种能量状态,每个能量状态的能量消耗、唤醒延迟和支持的外围设备都不相同。理想情况下,微控制器应尽可能地处于能满足应用需求的最低能耗状态。为了准确地确定微控制器所处的能量状态,需要明确微控制器各个子系统和外围设备的能耗状态。此外,微控制器各能量状态之间的切换是经常发生的。微控制器每次处理中断,就从低能耗的休眠状态转到工作状态,并且只要 TinyOS 调度器发现任务队列为空,它就将微控制器置于低能耗状态。在 TinyOS 2.x 中,微控制器的能量状态的切换由三种机制来决定:状态寄存器和控制寄存器、dirty 位和能量状态重载 (override)。

6.2.1 TinyOS 1.x 中的微控制器能量管理

当任务队列为空时, TinyOS 调度器将处理器置于休眠状态。然而,处理器通常有多个能量状态。例如, MSP430 有一个激活模式和五个低能耗模式。其低能耗模式从只禁用 CPU 和主系统时钟的 LPM0 模式到禁用 CPU、所有的时钟和振荡器的 LPM4 模式。各个低能耗模式下的能耗可以相差 350 倍甚至更多 (电源电压为 3V 时, LPM0 模式所需电流为 75 μ A, LPM4 为 0.2 μ A)。所以正确选择微控制器的低能耗模式可以大大提高系统的使用寿命。

TinyOS 1.x 的平台有几种不同的管理微控制器能量状态的方式,这几种方式之间有共性。例如, Mica 平台有一个名为 `HPLPowerManagement` 的组件,此组件有一个启用和禁用低能耗模式的命令,还有一个 `adjustPower()` 命令,此命令能根据各种控制寄存器和状态寄存器来计算低能耗模式,并把计算的值存储在 `Atmega128` 微控制器的控制寄存器中。当 TinyOS 调度系统使微控制器进入休眠状态时,它使用控制寄存器来决定进入哪种低能耗模式。相反,基于 MSP430 微控制器的平台则不同,如 Telos 平台和 eyes 平台,这些平台会自动计算低能耗模式,并使微控制器进入所计算出的低能耗模式。

这两种方法各有优缺点。在 TinyOS 1.x 中 Mica 平台使用方法的效率较高,此方法仅计算被告知的低能耗模式。然而,此方法由其他组件决定何时计算低能耗模式,这会引入漏洞。另外, TinyOS 1.x 缺少对硬件抽象架构的定义,这会使情况更加恶化。相反, MSP430 的方法相对简单,此方法在没有任何外部激励下,系统就可进入正确的能量状态。然而,它的开销较大,在用于唤醒系统的中断中引入了 40~60 个时钟周期的开销,限制了系统中断处理速度的提高。

这两种方法都假定 TinyOS 的调度系统可以通过检查控制寄存器和状态寄存器状态来确定正确的低能耗模式。例如, MSP430 微控制器默认低能耗模式是 LPM3,但在检测到 Timer A、USARTs 或者 ADC 为激活状态时,它使用 LPM1 低能耗模式。当微控制器进入休眠状态时,外围设备和子系统可唤醒节点。然而,能量模式间的切换需要一定的唤醒延迟 (由低能耗模式向更高能耗模式切换时),在使用更高层组件时需要注意这一点。虽

然，唤醒延迟在如 Atmega128 和 MSP430 的低能耗微控制器上不是一个重要的问题，但是，对于一些功能强大的处理器，如 Xsale 系列的处理器（用于 imote2 等平台），它们的最大唤醒时延可达 5ms。在一些应用场景中，这种延迟能够带来严重的问题。因此，更高层组件需要一种方法，为 TinyOS 微控制器能量管理提供所需的信息，用于计算合适的低能耗模式。

6.2.2 TinyOS 2.x 中微控制器能量管理

TinyOS 2.x 使用三种基本机制来管理和控制微控制器能量状态：dirty 位、基于芯片的低能耗状态计算函数和能量状态重载函数。dirty 位决定 TinyOS 何时计算新的低能耗状态，低能耗状态计算函数完成新低能耗状态的计算，能量状态重载函数允许高层组件根据需要对能量状态提出额外需求。

这三种机制都运行在 TinyOS 的核心调度循环中。核心调度循环在系统启动时调用。微控制器休眠时，Scheduler.taskLoop()命令可用。如果调用 Scheduler.taskLoop()命令时，任务队列为空，则 TinyOS 调度器会通过 MsuSleep 接口使微控制器进入休眠状态。

```
interface McuSleep {
    async command void sleep();
}
```

McuSleep.sleep()命令将微控制器置于低能耗休眠状态，此时微控制器可通过中断唤醒。此命令与 TinyOS 1.x 中的 _nesc_atomic_sleep()不同。注意，在 TinyOS 1.x 中使微控制器进入休眠模式的操作必须是原子属性的。_nesc_atomic_sleep()的调用必须在 atomic 中进行，必须原子性的重新使能中断并进入睡眠。如果系统在重新使能中断后、进入休眠前，系统处理了一个中断，这会带来一个问题：中断可能会提交一个任务，但是任务不会被运行，直到微控制器从休眠中唤醒。微控制器通常使用硬件机制来解决这个问题。例如，在 Atmega128 上，sei 指令在发出该指令两个周期后才会重新使能中断。

McuSleepC 组件提供了 McuSleep 接口，TinySchedulerC 组件必须自动地连接 McuSleep 接口到调度器执行。MsuSleepC 组件与使用的芯片或平台有关系，其属性声明需包括以下接口：

```
component McuSleepC {
    provides interface McuSleep;
    provides interface PowerState;
    uses interface PowerOverride;
}

interface McuPowerState {
    async command void update();
}

interface McuPowerOverride {
    async command mcu_power_t lowestState();
}
```

1. Dirty 位

当硬件表示层组件更改硬件配置时，这种更改可能会改变微控制器的低能耗状态，因此

必须调用 `McuPowerstate.update()` 命令。Dirty 位通过检测代码在不同存储位置的一致性发现这种更改，从而决定是否计算新的低能耗状态。如果 `McuPowerState.update()` 命令被调用，`McuSleep` 接口必须在调用 `McuSleep.sleep()` 命令再次进入休眠模式之前重新计算低能耗状态。

2. 低能耗状态计算

`McuSleepC` 组件负责计算在不干扰 TinyOS 各子系统操作的情况下，微控制器可安全进入的最节能低能耗模式。`McuSleepC` 组件具有原子属性，因此应避免频繁使用。因为频繁地使用 `McuSleepC` 组件（比如在每次中断时）会导致较大的开销和时延抖动。

微控制器的能量状态必须在标准芯片的头文件中用 `enum` 定义。这个文件同时需定义一个 `mcu_power_t` 的类型和一个 `combine` 函数。`combine` 函数可将两个能量状态值合并为一个返回。

例如，假设一个微控制器有三个低能耗模式 LPM0、LPM1 和 LPM2；两个硬件时钟资源 HR0 和 HR1。在 LPM0 模式，HR0 和 HR1 都处于激活状态。在 LMP1 模式，HR0 处于非激活状态，而 HR1 为激活状态。在 LPM2 模式，HR0 和 HR1 均处于非激活状态。组合函数如表 6-1 所示。

表 6-1 LPM2 模式下，HR0 和 HR1 均处于非激活状态时组合函数表

	LPM0	LPM1	LPM2
LPM0	LPM0	LPM0	LPM0
LPM1	LPM0	LPM1	LPM1
LPM2	LPM0	LPM1	LPM2

如果在 LPM2 模式，HR0 处于激活状态而 HR1 处于非激活状态，则组合函数如表 6-2 所示。

表 6-2 LPM2 模式下，HR0 激活态、HR1 非激活态时组合函数表

	LPM0	LPM1	LPM2
LPM0	LPM0	LPM0	LPM0
LPM1	LPM0	LPM1	LPM0
LPM2	LPM0	LPM0	LPM2

3. 能量状态重载

`McuSleepC` 组件计算最佳的低能耗状态时需调用 `PowerOverride.lowestState()` 命令。`McuSleepC` 组件应当有该命令的默认实现，该命令返回微控制器能达到的最节能的低能耗状态。该命令的返回值是一个 `mcu_power_t` 类型的状态变量。`McuSleepC` 组件必须将该返回值与计算出的低能耗状态相对应。

高层组件对能量状态有一些特别需求，如系统允许的最大唤醒时延，这些需求是不能从硬件状态和配置寄存器中获取的，所以就有了能量状态重载函数。由于能量状态重载函数可以涉及所有的微控制器能量保持机制，所以使用时必须谨慎。在核心调度循环中，能量状态重载函数 `PowerOverride.lowestState()` 具有原子属性，因此该函数的实现要求尽可能的高效，执行时间不应长于 20 或 30 个周期，并返回一个默认值。TinyOS 中不可随意地使用这个函

数。mcu_power_t 组合函数的存在意味着这个命令函数是一个扇出调用。

6.2.3 外围设备和子系统

在硬件接口层，TinyOS 子系统通常有一个简单的强制的能量管理接口。根据延迟时间的不同，这个接口可以是 StdControl 接口、SplitControl 接口或者 AsyncStdControl 接口。当任一组件调用另一个组件的 stop()命令时，接口强制性地使该组件所代表的子系统进入非激活的低能耗状态。从微控制器能量管理的角度来看，能量状态的切换可引起状态寄存器和控制寄存器的改变。微控制器能量管理子系统就会被告知这些变化，并在系统进入休眠状态之前做出适当的响应。

6.3 外围设备能量管理

TinyOS 平台的能量有限。由于激活时间、能量配置和操作时延不同，所有设备很难使用统一的能量管理策略。有些设备，如微控制器，可以很快计算出所允许的最低能耗状态，而其他设备，如需激活时间的传感器，需要有额外的信息才可以计算出所允许的最低能耗状态。

在 TinyOS 1.x 中，应用程序负责所有的能量管理。低层子系统，如 SPI 总线，由高层抽象控制其电源的开启与关闭。该方法深层次调用了 StdControl.start 命令和 StdControl.stop 命令，但是会带来一些奇怪的问题，并且不利于节能。例如，在 Telos 平台上，射频模块在关闭的情况下，关闭 SPI 总线会导致 flash 驱动器无法继续工作。此外，即使 SPI 总线上没有激活设备，为了监控 SPI 总线微控制器仍需处于高能耗状态。

为了进行能量管理，TinyOS 2.x 定义了两类设备：微控制器和外围设备。微控制器可以使用几个不同的能量状态。但是，外围设备只有两个截然不同的能量状态：开启或关闭。本节主要介绍 TinyOS 2.x 如何控制外围设备的能量状态。

6.3.1 能量管理模型

如图 6-6 所示，TinyOS 中管理外围设备的能量状态有两种不同的模型：显式能量管理模型和隐式能量管理模型。

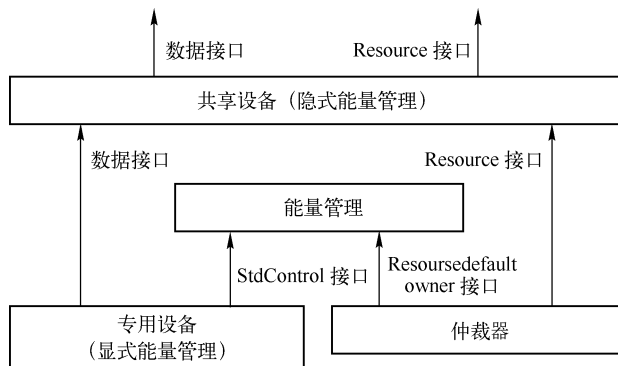


图 6-6 能量管理结构示意图

显式能量管理模型提供了一种单个用户组件控制专用物理设备能量状态的方法。当用户要求打开或关闭设备的能量状态时，设备会立即执行。当控制信息依靠高层组件中的外部逻

辑来控制时，该模型非常有效。

隐式能量管理模型提供了一种通过设备内部的驱动器控制设备的能量状态的方法。遵循这种模型的设备不能由外部的用户控制其电源状态的开启或关闭。设备驱动器需要定义一些内部策略，准确判定何时切换设备的能量状态。使用何种策略由物理设备自身的硬件条件决定，依附于显式能量管理模型，这些策略在设备的低层次抽象之上实现。

共享设备根据它们提供给用户的接口来推断是否需要开启或关闭电源。例如，当用户请求 ADC 时，需开启 ADC；若没有请求，则需关闭 ADC。因此共享设备不需要提供电源控制接口，可使用隐式的能量管理策略。

6.3.2 显式能量管理

为了使用显式能量管理模型控制外围设备电源状态的开启与关闭，与 TinyOS 1.x 类似，TinyOS 2.x 中提供了 StdControl 接口和 SplitControl 接口。此外，TinyOS 2.x 引入了第三个接口：AsyncStdControl 接口。如果一个组件需要对硬件设备的能量状态进行开启或关闭操作，则该组件必须提供上述三个接口中的一个。具体使用哪个接口，可根据能量状态切换时延和接口命令中的代码是同步还是异步来决定。

1. 基于 StdControl 接口的能量管理

如果开启或关闭设备所花费的时间可忽略不计，则可使用 StdControl 接口。

```
interface StdControl {
    command error_t start();
    command error_t stop();
}
```

外部组件必须调用 StdControl.start()命令来开启设备，调用 StdControl.stop()命令来关闭设备。这两个命令都必须返回 SUCCESS 或 FAIL。

成功调用 StdControl.start()命令后，设备必须处于完全开启状态，从而允许调用只有在该设备返回成功时才可调用的其他接口命令。成功调用 StdControl.stop()后，设备必须处于完全关闭状态，任何基于该设备实现的其他接口命令被调用时，必须返回 FAIL 或 EOFF。如果设备不能完成 StdControl.start()命令或 StdControl.stop()命令，则必须返回 FAIL。

基于上述规范，表 6-3 描述了设备处于不同能量状态时，调用 StdControl 接口的合法返回值。

表 6-3 StdControl 接口的合法返回值表

调 用	设 备 开	设 备 关
StdControl.start()	SUCCESS	SUCCESS or FAIL
StdControl.stop()	SUCCESS or FAIL	SUCCESS
其他接口命令	——	FAIL or EOFF

设备组件中提供的 StdControl 接口代码如下：

```
configuration DeviceC {
    provides {
```

```

interface Init;
interface StdControl; //For Power Management
....
}
}

```

2. 基于 SplitControl 接口的能量管理

当开启或关闭设备所花费时间不可忽略时，应当使用 SplitControl 接口代替 StdControl 接口。SplitControl 接口的定义如下：

```

interface SplitControl {
    command error_t start();
    event void startDone(error_t error);
    command error_t stop();
    event void stopDone(error_t error);
}

```

外部设备必须调用 SplitControl.start()命令来开启设备，调用 SplitControl.stop()命令来关闭设备。调用上述命令将返回 SUCCESS、FAIL、EBUSY 或者 EALREADY。SUCCESS 表示正在切换能量状态，随后会触发一个相应的完成事件。EBUSY 表示设备正在进行其他操作（如在开启设备时调用 stop 命令，在关闭设备时调用 start 命令），随后不会触发事件。EALREADY 表示设备已处于该能量状态，该命令的调用是错误的，此时完成事件不会触发。FAIL 表示没有成功切换设备的能量状态。

成功的调用 SplitControl.start()命令必须触发 SplitControl.startDone(SUCCESS)事件或 SplitControl.startDone(FAIL)事件。

成功地调用 SplitControl.stop()命令必须触发 SplitControl.stopDone(SUCCESS)事件或 SplitControl.stopDone(FAIL)事件。

在触发 SplitControl.startDone(SUCCESS)事件之前，设备必须完全开启，并且调用该设备其他接口命令的操作请求可能会成功。

在触发 SplitControl.stopDone(SUCCESS)事件之前，设备必须完全关闭，并且随后调用该设备其他接口命令后一定返回 EOFF 或 FAIL。

如果设备处于开启状态，并在成功的调用 SplitControl.stop()命令后触发 SplitControl.stopDone(FAIL)事件，设备必须仍然处于完全开启状态，并且调用该设备其他接口命令的操作请求可能会成功。

如果设备处于关闭状态，并在成功地调用 SplitControl.start()命令后触发 SplitControl.startDone(FAIL)事件，则设备必须仍然处于完全关闭状态，并且调用该设备其他接口命令后一定返回 EOFF 或 FAIL。

如果设备不能完成 SplitControl.start()或 SplitControl.stop()请求，则必须返回 FAIL。

当设备正在开启时调用 SplitControl.start()命令或者当设备正在关闭状态时调用 SplitControl.stop()命令必须返回 SUCCESS，随后会触发相应的 SplitControl.startDone()事件或 SplitControl.stopDone()事件。

当设备处于开启状态时调用 SplitControl.start()命令或者当设备处于关闭状态时调用 Split-

Control.stop()命令必须返回 EALREADY，表示设备已处于该状态。相应的完成事件不能被触发。

当设备正在关闭时调用 SplitControl.start() 命令或者当设备正在开启时调用 SplitControl.stop()命令必须返回 EBUSY，表示设备忙于执行一个不同的操作。相应的完成事件不能被触发。

表 6-4 是在设备处于不同状态时，SplitControl 接口命令的返回值：

表 6-4 SplitControl 接口的合法返回值表

调 用	设 备 开	设 备 关	设备正在开启	设备正在关闭
SplitControl.start()	EALREADY	SUCCESS FAIL	SUCCESS	EBUSY
SplitControl.stop()	SUCCESS FAIL	EALREADY	EBUSY	SUCCESS
其他接口命令	— —	FAIL EOFF EOFF	FAIL EOFF SUCCESS	FAIL EOFF

设备组件中提供的 SplitControl 接口代码如下：

```
configuration DeviceC {
    provides {
        interface Init;
        interface SplitControl; \\ For Power Management
        ....
    }
}
```

3. 基于 AsyncStdControl 接口的能量管理

StdControl 接口和 SplitControl 接口中的命令和事件都是同步的，不能在异步代码中被调用。当需异步代码控制设备的电源状态时，需要使用 AsyncStdControl 接口，其定义如下：

```
interface AsyncStdControl {
    async command error_t start();
    async command error_t stop();
}
```

所有 StdControl 设备提供的语义对于 AsyncStdControl 接口同样适用，设备组件中提供的 AsyncStdControl 接口代码如下：

```
configuration DeviceC {
    provides {
        interface Init;
        interface AsyncStdControl;
        ....
    }
}
```

6.3.3 隐式能量管理

虽然显式能量管理模型提供了改变能量状态的机制，但它没有指定策略。这在简单情形

下,如专用设备的能量管理,不会有大的问题。但是设备被多个用户控制时,它们之间的依赖关系复杂,此时显式能量管理模型就存在很严重的问题。

例如,如果组件 A 是组件 B 和组件 C 的用户,当组件 A 调用 `StdControl.stop()` 命令时,组件 B 和 C 中将会发生什么?组件 B 和 C 也会被停止吗?相反,如果组件 B 和 C 同时是一个共享组件 A 的用户,又会有什么不同?如果组件 B 和 C 被关闭,那么组件 A 也应被关闭吗?怎样才能恰当地处理这种级联的电源开关问题?

TinyOS 1.x 中使用 `StdControl` 接口会产生许多未知行为。在一些平台上,射频模块和 flash 设备共享 SPI 总线。在射频模块上调用 `StdControl.stop()` 命令会导致一系列的调用,最终导致 SPI 总线不可用,从而无法与 flash 设备通信。当然,正确的策略将会跟踪 SPI 总线的用户,并且在射频模块和 flash 设备均不使用 SPI 总线时关闭 SPI 总线。相反的,只要有用户,就应开启 SPI 总线。

选择正确的能量管理策略是一个复杂的任务,这取决于所使用设备的固有属性、依赖关系和特定的应用需求。因为有些特征可以预先得知或者是受一定限制,所以系统可强制使用一种默认的能量管理策略。

1. 能量管理策略

正如 TinyOS 2.x 使用通用仲裁器来提供共享资源的仲裁功能, TinyOS 2.x 也提供了通用能量管理策略来自动控制外围设备的能量管理。

通过仲裁器组件以共享资源形式实现的设备驱动,提供了资源依赖关系,从而实现默认的能量管理策略。当多个用户共享单个资源时,共享资源提供了组件间的依赖关系。这种关系可使默认的能量管理策略能自动地切换资源的能量状态。

能量管理者组件作为共享资源设备的默认所有者,通过使用 `ResourceDefaultOwner` 接口与该设备进行交互。`ResourceDefaultOwner` 接口代码见本书第 89 页。

作为默认所有者, `ResourceDefaultOwner.granted()` 事件触发后,能量管理组件可获得资源设备的所有权。

一旦获得了设备,能量管理组件就可使用底层资源提供的类似于 `StdControl` 的接口,执行自己的能量管理策略。不同的能量管理组件可执行不同的策略。最简单的策略就是通过调用 `stop()` 命令来直接关闭电源。当能量状态切换在唤醒时延或者能耗方面必须付出一定开销时,能量管理组件可能使用更加智能的策略来减小开销。该策略可能使用定时器来推后设备的关闭时间,使资源用户有机会在设备关闭之前发出请求。

无论能量管理策略是否正在使用,只要没有用户请求使用该资源,能量管理组件就一直保持资源的所有权。当用户提出请求时,能量管理组件将会从相关的仲裁器中接收一个 `ResourceDefaultOwner.requested()` 或者 `immediateRequested()` 事件。一旦接收到该事件,能量管理组件必须通过物理设备低层抽象提供的类似于 `StdControl` 的接口来开启设备。能量管理组件使用 `ResourceDefaultOwner.release()` 命令释放资源的所有权,但是它必须等到设备已经完全开启后才能执行该操作。

能量管理策略解决了如何关闭设备的问题,而能量管理策略与 `granted()` 事件和 `ResourceDefaultOwner.requested()` 事件的结合解决了何时关闭设备的问题。只有当一组级联资源用户的底层资源被完全释放时,能量管理者才能合适地关闭资源。

采样隐式能量管理模型的资源代码如下:

```

module MyFlashP {
    provides {
        interface Init;
        interface SplitControl;
        interface Resource;
        interface FlashCommands;
        ...
    }
}

implementation {
    ...
}

generic module PowerManagerC(uint8_t POWERDOWN_DELAY) {
    provides {
        interface Init;
    }
    uses {
        interface SplitControl;
        interface ResourceDefaultOwner;
    }
}

implementation {
    ...
}

#define MYFLASH_RESOURCE "MyFlash.resource"
#define MYFLASH_POWERDOWN_DELAY 1000
configuration MyFlashC {
    provides {
        interface Init;
        interface Resource;
        interface FlashCommands;
    }
}

implementation {
    components new PowerManagerC(MYFLASH_POWERDOWN_DELAY)
    FcfsArbiter(MYFLASH_RESOURCE), MyFlashP;
    Init = MyFlashP;
    Resource = FcfsArbiter;
    FlashCommands = MyFlashP;
    PowerManagerC.ResourceDefaultUser -> FcfsArbiter;
    PowerManagerC.SplitControl -> MyFlashP;
}

```

此例的实现由三个组件构成。第一个组件（MyFlashP）遵循显式能量管理模型来给 flash 设备定义接口。第二个组件（PowerManagerC）是一个能量管理通用组件，执行设备特定的能量管理策略。第三个组件（MyFlashC）是一个配件，它遵循隐式能量管理

模型，连接设备执行所需要的所有组件。它包含了 MyflashP 组件和 PowerManagerC 组件以及用于管理共享设备的仲裁器组件。这里要注意能量管理组件是如何连接到由仲裁器提供的 ResourceDefaultUser 接口和 flash 提供的 SplitControl 接口的。该 flash 设备的所有用户都直接连接到由仲裁器提供的资源接口上。PowerManagerC 组件通过 ResourceDefaultUser 接口触发事件，从而决定何时通过 SplitControl 接口开启和关闭设备的电源。

2. 能量管理组件实例：PowerManagerC 和 DeferredPowerManagerC

TinyOS 2.x 提供了两种默认的能量管理策略。这两种策略是由位于 tinyos-2.x/lib/power 下的各种组件来实现的。第一种策略使用直接电源控制方案，即设备在被请求和释放时，分别直接开启和关闭。第二种策略使用延时电源控制方案，即设备被请求时直接开启设备，设备被释放一段时间后才关闭设备。

每个策略都是由 StdControl 接口、SplitControl 接口或者 AsyncStdControl 接口中的一个来实现的。

直接电源控制组件：

- StdControlPowerManagerC;
- SplitControlPowerManagerC;
- AsyncStdControlPowerManagerC。

延时电源控制组件：

- StdControlDeferredPowerManagerC;
- SplitControlDeferredPowerManagerC;
- AsyncStdControlDeferredPowerManagerC。

第 7 章 TinyOS 典型应用

本章就无线传感器网络中的基本应用，结合 TinyOS 的编程实践，详细介绍包括传感、存储、节点与 PC 通信、节点之间通信以及定时器等编程操作。这些功能集成在 TinyOS 系统中，希望通过本章的介绍，使读者对 TinyOS 编程有更全面的认识。

7.1 传感

本节主要介绍 TinyOS 系统中数据采集的工作方式，并将演示两个关于应用的例子：一个是简单的 Sense 实例，它周期性地读取传感数据，并显示在 LED 上；另一个实例 Oscilloscope 相对复杂些，节点周期性地读取传感器的数值并发送到基站节点，再由基站转发传感器数据到计算机串口，并在计算机上以图形界面显示出来。

7.1.1 传感简介

利用传感器进行感知是无线传感器网络的主要功能之一。在 TinyOS 1.x 中，如 Oscilloscope 或 Sense 之类的应用程序都使用 ADCCControl 接口来获取传感器数据。当带有传感器的新平台出现，可以通过串行接口来读取传感器数据，为此需要引进诸如 ADCError 之类的额外接口，但是这并不一定适合于带有 ADC 的传感器。

通常，采集传感器数据可以分成两个步骤：配置并启动传感器和读取传感器数据。第 1 步的工作非常复杂，包括传感器的配置，以及与其链接的硬件模块的配置，例如当与处理器通过模数转换器（Analog to Digital Converter, ADC）或串行外设接口（Serial Peripheral Interface, SPI）相连接时，就需要配置相应的 ADC 模块和 SPI 模块。考虑到平台不同，传感器也不同，各自的配置要求（如 ADC 输入通道、参考电压等）也就完全不一样。第 2 步的工作是读取传感器的数据，其实现较为容易。

在 TinyOS 2.x 中，如 Oscilloscope、Sense 以及 RadioSenseToLeds 之类的应用都具有平台无关性，它们都没有使用如 ADCCControl 之类的配置接口，而是使用了标准的数据获取接口，如 Read 接口、ReadStream 接口和 ReadNow 接口。所有配置详情在应用程序里都被隐藏了，所以即使实际的传感器以及硬件连接情况都完全不一样，Sense 应用也可以直接编译并下载到 TelosB 平台或 MicaZ 平台上。

7.1.2 Sense 实例

Sense 是一个简单的传感器数据采集程序示例。它周期性地采集传感器数据，并显示读取数值的低 3 位。Sense 应用可以在/opt/tinyos-2.x/apps/Sense 里找到，其配件 SenseAppC 的内容如下：

```

configuration SenseAppC
{
}
implementation {
  components SenseC, MainC, LedsC, new TimerMilliC();
  components new DemoSensorC() as Sensor;

  SenseC.Boot -> MainC;
  SenseC.Leds -> LedsC;
  SenseC.Timer -> TimerMilliC;
  SenseC.Read -> Sensor;
}

```

Sense 应用的配件和 Blink 应用的 BlinkAppC 配件非常相似。为了更好地理解这些接口的绑定情况，查看 SenseC 模块中使用到的接口：

```

module SenseC
{
  uses {
    interface Boot;
    interface Leds;
    interface Timer<TMilli>;
    interface Read<uint16_t>;
  }
}

```

和 BlinkC 一样，SenseC 模块使用了 Boot、Leds 和 Timer<TMilli>接口。此外，它还使用了 Read<uint16_t>接口。SenseC 模块的具体操作步骤是：Boot 接口在系统初始化后启动了一个周期性的定时器，每隔一定时间触发定时器事件，并在该事件的响应中通过 Read<uint16_t>接口来读取数据。读取数据由 Read.read 命令的调用和 Read.readDone 事件的触发两部分构成。每次定时器到时都会调用 Read.read 命令，然后等待 Read.readDone 事件的触发。在 Read.readDone 事件中获取采样数据，并把数据的低 3 位显示在 LED 上。

Read 接口一次只能读取一个数据，其定义位于“tinyos-2.x/tos/interfaces”目录：

```

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

```

上述代码定义了一个通用 Read 接口。需要注意的是，该通用接口必须带有一个参数，当且仅当提供的接口和使用的接口具有一样的参数时，两个组件才可以连接起来。还须注意，readDone 事件传递了一个<val_t>类型的参数，其实这只是代替实际参数的占位符。SenseC 模块为 Read 接口使用了一个 unit16_t 的变量，它只能绑定到提供 Read<unit16_t>接口的组件，这表明 SenseC 模块希望读取 16 位无符号整型数据。如果将该接口连接到提供 Read<unit8_t>接口的组件，就会出现编译错误。

现在再来看一下 SenseC 模块使用的 Read<unit16_t>接口的语句:

```
components new DemoSensorC() as Sensor;
```

和

```
SenseC.Read -> Sensor;
```

通用组件 DemoSensorC 提供 Read<unit16_t>接口给 SenseC 组件,但是我们还不知道 SenseC 模块到底被连接到哪个传感器。事实上, SenseC 组件并不定义连接的传感器类型,它甚至也不关心是否能从传感器成功读取数据。这是因为具体的读取数据功能有提供 read 接口的组件来完成。了解这一点对更好的理解 TinyOS 系统是很有帮助的。由于每个平台都是不一样的,在不同的平台上将由不同的 DemoSensorC 组件来提供 read 接口供 SenseC 使用,这种做法很好的屏蔽了平台差异对上层开发的影响。例如,在一个既没有内置传感器也没有附带传感板的平台上(如 MicaZ 平台),DemoSensorC 组件可能只是简单地返回一个常量。此外,传感板也可能附带有相应的 DemoSensorC 组件(比如在 Mica 系列的传感板上,针对板载的光照传感器定义了 demoSensorC 组件)。

接下来,让我们来看一下 DemoSensorC 组件如何定义采样的传感器。

1. DemoSensorC 组件

每一个 DemoSensorC 组件都采用如下方式进行声明:

```
generic configuration DemoSensorC()
{
    Provides interface Read<unit16_t>;
}
```

在不同的节点平台上, DemoSensorC 组件的实现部分是不同的。例如,在 TelosB 平台, DemoSensorC 组件里实例化一个 VoltageC 组件,它可以从 MCU 内部的电压传感器获得节点的电池电压。而 micaz 平台没有内置的传感器,它的 DemosensorC 组件使用系统库组件,比如 ConstantSensorC 组件或 SineSensorC 组件,这些组件返回虚假的传感器数据。也就是说, DemoSensorC 组件间接地从相关的传感器组件(如 VoltageC 组件)获得传感器数据,并提供给应用程序。通常,传感器的配置工作由 DemoSensorC 组件内部实例化的传感器组件完成。

那么,该怎样修改 Sense 应用程序,使其不再采用默认的传感器?通常的做法是修改 DemoSensorC,即将第一行代码修改为第二行代码:

```
components new VoltageC() as DemoSensor;
components new ConstantSensorC(unit16_t,OxbeeF) as Demosensor;
```

至于有哪些传感器可用,这就由具体平台决定。一般,传感器组件可能位于相应平台的子目录(/tos/platforms),或者传感板的子目录(/tos/sensorboards);如果是处理器内部的传感器,就可能位于各自芯片的子目录(/tos/chips)。ConstantSensorC 组件和 SineSensorC 组件分别返回常量值和正弦值作为传感器数据,它们可以在系统组件库(/tos/system)找到。

2. 运行 Sense 应用

在 apps/Sense 目录下，根据当前的硬件平台，输入以下编译命令，如果碰到如下错误：

```
$ make telosb install
SenseAppC.nc:50: component DemoSensorC not found
SenseAppC.nc:50: component `DemoSensorC` is not generic
SenseAppC.nc:50: no match
```

表明当前平台还没有 DemoSensorC 组件。可以从/tos/platform/micaz 目录复制 DemoSensorC.nc 文件到当前的平台目录。

如果有一个 Mica 系列的节点和一个基本传感板（mda100），可以做一个更有趣的测试，输入以下命令告诉节点采集基本传感板 mda100 上的光照传感器数据：

```
SENSORBOARD=basicsb make platform install
```

下载运行该程序，传感数据的低 3 位就会显示在节点的 LED 上。之所以是低 3 位，是因为 Sense 应用不知道返回值的精确范围。如果是 unit_16 类型值的高 3 位，当数值经过 12 位 ADC 转换后就毫无意义（除非数值是左对齐）。如果 DemoSensorC 组件从传感器那里获得的数值在上下波动，将会看到 LED 灯在不停地闪烁。

7.1.3 Oscilloscope 实例

Oscilloscope 应用是一个可以在计算机上显示数据曲线的应用程序。节点周期性地启动传感器采集数据，当传感数据达到 10 个后将它们通过无线发送出去。另一个运行着 BaseStation 应用的节点通过无线接收这些数据并通过串口转发至 PC 机。因此要运行 Oscilloscope 应用，至少需要两个节点：一个基站节点，通过串口连接到 PC（BaseStation 应用位于“tinyos-2.x/apps/BsaeStation”目录），另一个节点运行 Oscilloscope 应用。

OscilloscopeAppC 配件的代码如下：

```
configuration OscilloscopeAppC
{
}
implementation
{
  components OscilloscopeC, MainC, ActiveMessageC, LedsC,
    new TimerMilliC(), new DemoSensorC() as Sensor,
    new AMSenderC(AM_OSCILLOSCOPE),
    new AMReceiverC(AM_OSCILLOSCOPE);

  OscilloscopeC.Boot -> MainC;
  OscilloscopeC.RadioControl -> ActiveMessageC;
  OscilloscopeC.AMSend -> AMSenderC;
  OscilloscopeC.Receive -> AMReceiverC;
  OscilloscopeC.Timer -> TimerMilliC;
  OscilloscopeC.Read -> Sensor;
```

```
OscilloscopeC.Leds -> LedsC;
}
```

OscilloscopeC 模块的部分代码如下：

```
module OscilloscopeC
{
  uses {
    interface Boot;
    interface SplitControl as RadioControl;
    interface AMSend;
    interface Receive;
    interface Timer;
    interface Read;
    interface Leds;
  }
}
```

Oscilloscope 应用与 Sense 应用一样，使用了 DemoSensorC 组件和一个周期性的定时器。当它收集了 10 个传感器数据，就把它们放到一个消息包里，然后通过 AMSend 接口发送出去。在 OscilloscopeC 组件中，SplitControl 接口开启无线服务，通过 Receive 接口接收控制信号，从而同步采样速率。

1. 运行 Oscilloscope 应用

进入 tinyos-2.x/apps/Oscilloscope 目录，根据具体的平台，输入“make micaz install.1”，将 Oscilloscope 应用下载到节点。需要说明一点的是，“install”之后的“.1”是指定节点的 ID 号，分配不同的节点 ID 号有助于在图形用户界面中分辨不同节点。所以，应当确保给所有的 Oscilloscope 节点分配了不同的 ID 号（例如，下载 Oscilloscope 应用到第 2 个节点可以输入“make micaz install.2”命令，以此类推）。Oscilloscope 节点会在每次发送信息时切换 LED2 的显示状态，接收消息时切换 LED3 的状态，而在无线电连接出现问题时，就会切换 LED1 的状态。Oscilloscope 节点能够接收到包含有控制信号的消息，从而实现不同节点的采样同步，还可以改变采样速率。

下载 BaseStation 应用程序到另一个节点，并将其插到计算机的串口，每当有信息从无线网络桥接到串口时，LED2 就会闪烁。

2. 运行 Java 图形界面

在“/apps/Oscilloscope/java”目录输入“make”命令，然后启动 SerialForwarder 工具，确认其连接到基站节点串口；接着就可以运行 GUI 工具，在“/apps/Oscilloscope/java”目录输入“./run”命令，得到如图 7-1 所示的图形窗口。

每一个节点用不同颜色的线条表示，横轴表示接收到数据包的序号，而纵轴表示数据包里包含的传感器采样值。为了改变采样速率，可修改“sample period(ms)”栏。当修改采样周期并确认后，包含采样数率的消息将被发送到 BaseStation 节点，由其广播到网络中的所有节点。

Oscilloscope（或 Sense）应用程序显示的都是 Read.readDone 事件中获得的原始数据，从图形界面的纵轴可以大概了解到数据的波动范围。

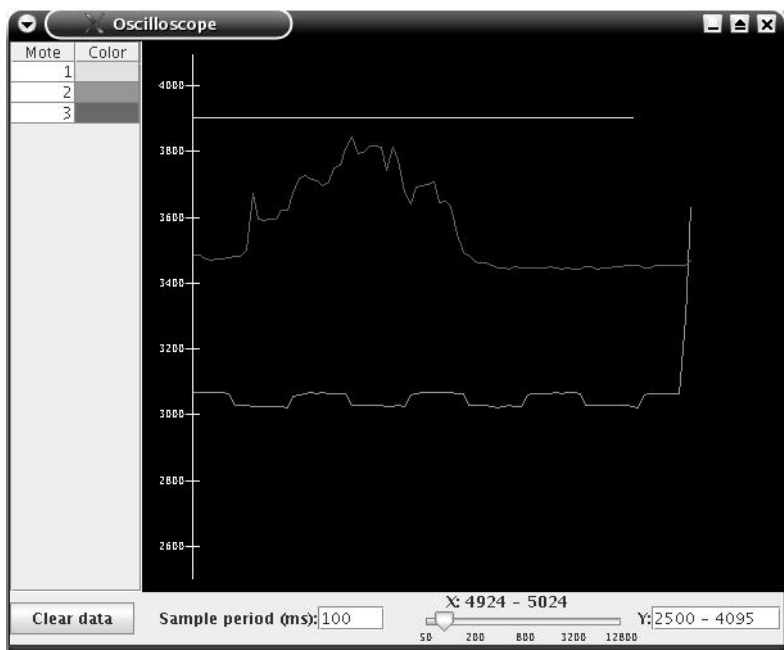


图 7-1 Oscilloscope 图形界面

7.2 存储

本节将对 TinyOS 系统中永久性（或非易失性）数据存储操作进行介绍。具体的来讲就是对 flash 芯片上进行存储的操作，这种数据存储方式允许节点在即使断电或者重新编程的情况下也能够保持原来的数据。针对 TinyOS 系统不同的数据存储类型，本节将就一下问题展开具体的介绍：

- 1) 如何把 flash 芯片划分为卷，允许多种不同类型数据的存储；
- 2) 如何使配置数据保存时间超过一个电源开/关周期（Power Cycle）；
- 3) 如何使用日志存储方式。

7.2.1 存储简介

TinyOS 2.x 提供了 3 种基本的存储类型：小数据对象存储、循环的日志存储以及大数据对象存储。TinyOS 2.x 也提供了相应的接口与组件来对底层的存储服务进行抽象。

通过一些相关接口（/tos/interfaces）与类型定义（/tos/types），我们可以大致了解到存储系统的功能。几个重要的相关文件包括：BlockRead、BlockWrite、Mount、ConfigStorage、LogRead、LogWrite 和 Storage.h，而组件 ConfigStorageC、LogStorageC 和 BlockStorageC 为上述接口提供了具体实现。

在不同的硬件平台，同一存储抽象的组件虽然名字相同，但实际的实现代码却可能不同。这看起来虽然繁琐，不过不用担心，因为 TinyOS 的编译系统会自动包含相关芯片的正确驱动。在应用开发环节，程序员不需要担心这些相关驱动的文件位置，只需知道这些组件

的名称即可。

如果读者对这些特定芯片的存储驱动比较好奇，以 Mica2/MicaZ 节点上的 Atmel AT45DB 系列 flash 存储为例，可以试着去了解它们的具体实现：

```
\tos\chips\at45db\ConfigStorageC
\tos\chips\at45db\LogStorageC
\tos\chips\at45db\BlockStorageC
```

下面是 M25Pxx 系列 flash 存储和 Intel imote2 节点 flash 存储的具体实现。

M25Pxx 系列 flash:

```
\tos\chips\stm25p\ConfigStorageC
\tos\chips\stm25p\LogStorageC
\tos\chips\stm25p\BlockStorageC
```

Intel imote2 节点的 flash:

```
\tos\platforms\intelmote2\ConfigStorageC
\tos\platforms\intelmote2\LogStorageC
\tos\platforms\intelmote2\BlockStorageC
```

在 TinyOS 2.x 系统中，编译时可以用 XML 文件来说明 flash 芯片的分卷情况。这个 XML 文件，称作卷表（volume table），它指明了卷的名称、大小以及在 flash 中的起始地址。每个卷只能提供一种存储类型（如配置存储、日志存储或块存储）。存储类型定义了 flash 存储体上数据的物理布局。以下是一个卷表实例：

```
<volume_table>
  <volume name="CONFIGLOG" size="65536"/>
  <volume name="PACKETLOG" size="65536"/>
  <volume name="SENSORLOG" size="131072"/>
  <volume name="CAMERALOG" size="524288"/>
</volume_table>
```

该卷表定义了 CONFIGLOG 卷，其大小为 65536B，起始位置为 0B；而 PACKETLOG 卷的大小为 65536B，起始位置则为 65536B，因为前面的 65536B 属于 CONFIGLOG 卷。此外，该卷表还定义了 131072B 的 SENSORLOG 卷和 524288B 的 CAMERALOG 卷。

具体应用程序的卷表必须放在该程序的目录下，且必须命名为 volumesCHIPNAME.xml，其中 CHIPNAME 为 flash 存储芯片的名称。例如，Twlos 节点使用的是 ST 微电子公司的 M25P 系列 flash 存储体，其芯片驱动可以在“tos/chips/stm25p”目录中找到。因此，一个基于 Telos 平台且使用存储功能的应用程序都需要一个名为 volumes-stm25p.xml 文件。

注意：卷表中的 size 参数必须是 flash 芯片可擦写单元的倍数，而不同 flash 芯片可擦写单元的字节数可能各有不同。

7.2.2 配置数据的存储

配置数据是指用于配置节点属性的一组参数。它们的字节数是不确定的，从几十字节到

几百字节，一般都属于小数据对象。另外，它们的值也不确定，在各个节点可能都不一样，甚至是未知的。本节主要介绍配置数据如何保存以及如何从存储中读取。

配置数据要求必须能够在复位、能量循环（关闭电源，再重开电源，算作一个能量循环）或者重编程时保存下来。在很多场合，完整保存配置数据的能力是非常必要的。

对于传感器网络节点而言，配置数据中一个很重要的部分是刻度校正。一般情况下，传感器的校准系数在节点出厂时就已配置好，并存储下来，所以它们不会在节点掉电或者重编程时丢失。例如，温度传感器利用校准系数将输出电压信号转换成直观的温度数据。传感器校准系数的结构定义代码可参考如下：

```
typedef struct calibration_config_t {
    int16_t temp_offset;
    int16_t temp_gain;
} calibration_config_t;
```

身份信息，即设备辨认信息，也是配置数据的重要组成部分。如 IEEE 兼容的 MAC 地址或 TinyOS 的 TOSNODE_ID 参数在各个节点都是不一样的，一旦被分配到节点上，这些值就会被保存，即使碰到复位、能量循环和重编程，也不会丢失。其结构定义代码如下：

```
typedef struct radio_config_t {
    ieee_mac_addr_t mac;
    uint16_t tos_node_id;
} radio_config_t;
```

除了校正信息和身份信息之外，配置数据中一般还包括位置信息和传感参数等。节点的位置数据可能在编译时是未知的，只有在部署节点时才确定。例如，一个应用程序可以按下面的数据形式存储节点的坐标信息。

```
typedef struct coord_config_t {
    uint16_t x;
    uint16_t y;
    uint16_t z;
} coord_config_t;
```

传感参数与信号检测和发送有关的参数，如采样周期、滤波系数以及可调的报警值。这些配置数据的结构形式如下：

```
typedef struct sense_config_t {
    uint16_t temp_sample_period_milli;
    uint16_t temp_ema_alpha_numerator;
    uint16_t temp_ema_alpha_denominator;
    uint16_t temp_high_threshold;
    uint16_t temp_low_threshold;
} sense_config_t;
```

接下来我们将用一个简单的例子程序来说明如何存储和读取配置数据，这个示例程序的功能是节点周期性地从 flash 中读出配置数据，处理后再写回 flash。

卷在第 1 次使用之前不包含任何有效数据。因此，应当首先检测卷是否是初次使用，并作出适当的行为（例如预加载默认值）。同样，当卷中的数据发生变化时（例如应用程序需要新的或不同的配置变量），应用程序应当检测到变更，并作出相应的响应（例如擦写卷，并重新加载默认值）。类似这些情况，建议跟踪卷的版本。为此，在配置数据的结构中，引入了版本控制变量。BlinkConfig 应用中配置数据的字段包括卷的版本和读取周期。其结构形式如下：

```
typedef struct config_t {
    uint16_t version;
    uint16_t period;
} config_t;
```

下面将分析 BlinkConfig 例子程序的设计要点和具体的操作步骤：

创建 volumes-CHIPNAME.xml 文件，即建立卷表，并保存在应用程序所在目录。注意，CHIPNAME 是目标平台上的 flash 芯片名字。例如，Telos 平台上的 flash 是 stm25p，Micaz 平台的 flash 则是 at45db。卷表的具体内容如下：

```
<volume_table>
  <volume name="LOGTEST" size="262144"/>
  <volume name="CONFIGTEST" size="131072"/>
</volume_table>
```

编译工具会利用该卷表自动创建一个 StorageVolumes.h 头文件。然而，在有 ConfigStorageC 组件声明的配件(如 BlinkConfigAppC 组件)里，需要手动添加该头文件：

```
#include "StorageVolumes.h"
```

(2) 在 BlinkConfigC 模块的规范说明里，声明 Mount 接口和 ConfigStorage 接口。注意，ConfigStorage 接口被重命名为 Config 接口。其代码如下：

```
module BlinkConfigC {
    uses {
        ...
        interface ConfigStorage as Config;
        interface Mount;
        ...
    }
}
```

(3) 每个接口必须绑定到提供该接口的组件。相应的代码如下：

```
configuration BlinkConfigAppC {
}
implementation {
    components BlinkConfigC as App;
    components new ConfigStorageC(VOLUME_CONFIGTEST);
    /NOLUME_CONFIGTEST 定义在自动生成的 StorageVolumes.h 头文件中
```

```

...

App.Config -> ConfigStorageC.ConfigStorage;
App.Mount -> ConfigStorageC.Mount;

...
}

```

(4) 在使用 flash 存储芯片前,必须先挂载卷。在 Boot.booted 事件处理函数中调用 Mount.mount 命令,启动挂载,其代码如下:

```

event void Boot.booted() {
    conf.period = DEFAULT_PERIOD;

    if (call Mount.mount() != SUCCESS) {
        //失败情况的处理
    }
}

```

(5) 卷挂载成功,就会触发 Mount.mountDone 事件。接下来,需要检查挂载的卷是否有效。如果卷有效,使用 Config.read 命令读取卷中的数据即可。否则,调用 Config.commit 命令使其有效(这个命令也可用来书信缓冲数据,类似于 UNIX 系统中调用 fsync 命令可以刷新写入到磁盘上的缓冲),检查挂载的卷的代码如下:

```

event void Mount.mountDone(error_t error) {
    if (error == SUCCESS) {
        if (call Config.valid() == TRUE) { //检查挂载的卷是否有效
            if (call Config.read(CONFIG_ADDR, &conf, sizeof(conf)) != SUCCESS) {
                // 失败情况的处理
            }
        }
        else {
            // 分卷无效,发送 commit 命令使其有效
            call Leds.led1On();
            if (call Config.commit() == SUCCESS) {
                call Leds.led0On();
            }
            else {
                // 失败情况的处理
            }
        }
    }
    else {
        // 失败情况的处理
    }
}

```

(6) Config.read 命令执行成功, 就会触发 Config.readDone 事件。这时, 先检查是否成功读取了卷中的内容, 如果数据读取成功, 再检查版本号。如果版本号与预期的一样, 就可以复制配置数据到本地的配置变量, 并调节变量 period 的值; 如果版本号不对, 就将配置变量设为默认值。最后, 调用 Config.write 函数将新的配置变量写回到 flash 存储, 具体实现代码如下:

```
event void Config.readDone(storage_addr_t addr, void* buf,
    storage_len_t len, error_t err) __attribute__((noinline)) {

    if (err == SUCCESS) {
        memcpy(&conf, buf, len);
        if (conf.version == CONFIG_VERSION) {
            conf.period = conf.period/2;
            conf.period = conf.period > MAX_PERIOD ? MAX_PERIOD : conf.period;
            conf.period = conf.period < MIN_PERIOD ? MAX_PERIOD : conf.period;
        }
        else {
            call Leds.led1On();
            conf.version = CONFIG_VERSION;
            conf.period = DEFAULT_PERIOD;
        }
        call Leds.led0On();
        call Config.write(CONFIG_ADDR, &conf, sizeof(conf));
    }
    else {
    }
}
```

(7) 调用 Config.commit 命令。当调用 ConfigStore.write 命令并触发 Config.writeDone 事件时, 并不能保证数据一定写入了 flash。为了确保数据被保存到 flash, 就要调用 Config.commit 命令, 调用 Config.commit 命令的代码如下:

```
event void Config.writeDone(storage_addr_t addr, void *buf,
    storage_len_t len, error_t err) {
    // Verify addr and len
    if (err == SUCCESS) {
        if (call Config.commit() != SUCCESS) {
            // Handle failure
        }
    }
    else {
        // Handle failure
    }
}
```

(8) 将数据写入 flash 中，当 Config.commitDone 事件触发时，数据就将被永久地写入到 flash 中，并能够存活一个能量循环，具体实现代码如下：

```
Event void Config.commitDone(error_t err) {
    call Leds.led0Off();
    call Timer().startPeriodic(conf.period);
    if (err == SUCCESS) {
        //失败情况的处理
    }
}
```

7.2.3 日志数据的存储

通常，应用程序需要可靠地记录事件和小数据项，因此 TinyOS 提供一种日志形式的记录方式，这种记录方式记录的数据即使在系统崩溃时也不会丢失。日志记录可以是顺序录入（卷满就停止录入）也可以是循环录入（卷满就从该卷的起始地址开始写入）。

TinyOS 系统的日志存储抽象具有这些特点：

- 1) 日志是基于记录的，每次调用 LogWrite.append 命令就创建一条新纪录；
- 2) 节点断电时（崩溃或能量循环），日志只丢失最近的记录；
- 3) 一旦循环日志绕了一圈，日志就覆盖最早的记录。

示例程序 PacketParrot (tinyos-2.x/apps/tutorials/PacketParrot/) 演示了如何使用 LogWrite 和 LogRead 接口记录日志。该应用程序的主要功能是：节点将无线接收到的数据记录到 flash 中；如果读取失败，则擦除该日志，然后继续接收并记录数据包，如此循环运行。为了便于观察和调试，我们在程序中设定：当日志被擦除时，点亮红色 LED 灯 (Led0)；接收到数据包时则点亮黄色 LED 灯 (Led1)，成功录入后熄灭；如果数据包被接收但没有被录入（因为日志正在被擦除），黄色 LED 灯保持点亮状态；在一个能量循环后，如果记录的数据包被取出并发送出去，绿色 LED 灯 (Led2) 快速闪烁。下面将对 PacketParrot 的设计要点和具体操作步骤进行介绍。

(1) 使用日志存储使用日志存储的第 1 步是要确定存储什么样的数据到日志里。PacketParrot 程序声明了如下的结构体：

```
Typedef nx_struct logentry_t{
    nx_uint8_t len;
    message_t msg;
}logentry_t;
```

(2) 传递缓冲区指针和读取字节数。与配置存储不同，日志存储不需要挂载卷。一个简单的 read 命令就可以实现日志的读取，LogRead.read 命令传递缓冲区指针和读取字节数，代码如下：

```
event void AMControl.startDone(error_t err) {
    if (err == SUCCESS) {
        if (call LogRead.read(&m_entry, sizeof(logentry_t)) != SUCCESS) {
```

```

        //Handle error
    }
}
else {
    call AMControl.start();
}
}

```

(3) 检查返回的数据长度。如果 `LogRead.read` 命令返回 `SUCCESS`，那 `LogRead.readDone` 事件就会马上被触发。在 `LogRead.readDone` 事件中，首先检查返回的数据长度是否和预期长度一样。如果一致，可以将该数据无线发送出去；否则，可以认为日志是空的，或已失去了同步，那么日志就需要被擦除，代码如下：

```

Event void Logread.readDone(void* buf, storage_len_t len, error_t err) {
    If( (len == sizeof(logentry_t)) && (buf == &m_entry) ) {
        call Send.send(&m_entry.msg, m_entry.len);
        call Leds.led1On();
    }
    else {
        if (call LogWrite.erase() != SUCCESS) {
            //Handle error
        }
        call Leds.led0ON();
    }
}

```

(4) 接收数据包并将其中的数据写入 flash。接收到来自无线电的数据包，并调用 `LogWrite.append` 命令将数据包中的数据写入到 flash，具体实现代码如下：

```

event message_t* Receive.receive(message_t* msg, void* payload, uint*_t len) {
    call Leds.led2On();
    if (!m_busy) {
        m_busy = TRUE;
        m_entry.len = len;
        m_entry.msg = *msg;
        if (call LogWrite.append(&m_entry, sizeof(logentry_t)) != SUCCESS) {
            m_busy = FALSE;
        }
    }
    return msg;
}

```

(5) 触发 `LogWrite.appendDone` 事件如果 `LogWrite.append` 返回 `SUCCESS`，就会触发 `LogWrite.appendDone` 事件。这个事件返回了日志写入的详情，如缓冲区指针、写入数据的长度、是否有记录丢失（如果是循环缓冲）以及错误提示。如果没有错误发生，数据就在原子性（每条记录不可被分隔开保存）、连续性（前后条的记录是连续的）和耐用性（能经受

节点崩溃和重启)的保证下写入到 flash, 具体实现代码如下:

```
event void LogWrite.appendDone(void* buf, storage_len_t len,
    bool recordsLost, error_t err) {
    m_busy = FALSE;
    call Leds.led2Off();
}
```

7.2.4 大数据块的存储

块存储 (Block Storage) 通常用于保存那些不能存于 RAM 的大数据对象。块是一个低级的系统接口, 使用时需要小心, 因为其本质是一次性写入的存储模型。重写就需要擦除, 而擦除比较耗费时间, 且必须以块为单位 (如 256B~64KB) 擦除, 以及只能擦除有限次 (一般是 10000~100000 次)。TinyOS 的重编程下载系统就是在节点上使用块存储来保存程序镜像的。

在 tinynos-2.x/apps/tests/storage/Block/目录下, 有一个块存储的例子程序供读者学习和参考。下面将简单介绍下相关的两个接口: BlockRead 接口和 BlockWrite 接口。

1) BlockRead 接口

```
interface BlockRead {
    command error_t read(storage_addr_t addr, void* buf, storage_len_t len);
    event void readDone(storage_addr_t addr, void* buf, storage_len_t len, error_t error);
    command error_t computeCrc(storage_addr_t addr, storage_len_t len, uint16_t crc);
    event void computeCrcDone(storage_addr_t addr, storage_len_t len, uint16_t crc, error_t error);
    command storage_len_t getSize();
}
```

2) BlockWrite 接口

```
interface BlockWrite {
    command error_t write(storage_addr_t addr, void* buf, storage_len_t len);
    event void writeDone(storage_addr_t addr, void* buf, storage_len_t len, error_t error);
    command error_t erase();
    event void eraseDone(error_t error);
    command error_t sync();
    event void syncDone(error_t error);
}
```

7.3 节点与 PC 的通信

在无线传感器网络中传感器节点一般有两种通信方式: 节点之间通过无线通信, 节点与 PC 之间通常是有线通信, 通过 PC 串口或者 USB 口与传感器节点连接, 交互数据。前面已经讲过节点与节点无线通信方式的实现原理和操作步骤, 本节将重点介绍节点与 PC 之间的串口通信。

7.3.1 串口通信堆栈

在 TinyOS 2.x 串口协议栈分为 4 个功能组件，自底向上分别是：原始 UART、编码器/装帧器、传输协议、分派器。

堆栈结构如图 7-2 所示：

1. 原始 UART

协议栈的最下一层是原始 UART 组件，HIL 层组件提供了配置 UART（速率，停止位等）、收发字节、刷新 UART 缓冲区的功能。

UART 的 HIL 组件为 UartC，它为串口通信提供了字节级的接口 SerialByteComm。SerialByteComm 接口代码如下：

```
interface SerialByteComm {
    async command error_t put(uint8_t data);
    async event void get(uint8_t data);
    async event void putDone();
}
```

另外，UartC 提供接口以便在 UART 空闲时发出通知信号。有时我们需要知道发送到串口的数据是否已传输完毕（比如无线收发机需要从发送转换到接收状态时），这时我们就可以通过这个接口来进行判断。代码如下：

```
SerialFlush {
    command void flush();
    event void flushDone();
}
```

但是如果 MCU 使用的是双缓冲 UART 通信机制，那么 putDone 事件只意味着它已经准备好接收下一字节，而不是 UART 空闲。

2. 编码器/装帧器

编码器/装帧器组件建于原始 UART 层之上，将原始字节数据转换为串口协议数据包格式。编码/装帧器假定有两种类型的字节：分隔符和数据字节，分别用不同的事件向上层组件发出信号。

HdlcTranslatorC 是串口协议栈中的编码和解码组件，它使用 SerialByteComm 接口，提供 SerialFrameComm 接口。SerialFrameComm 接口代码如下：

```
interface SerialFrameComm {
    async command error_t putDelimiter();
    async command error_t putData(uint8_t data);
    async command void resetSend();
    async command void resetReceive();
    async event void delimiterReceived();
    async event void dataReceived(uint8_t data);
    async event void putDone();
}
```

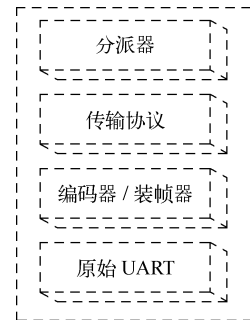


图 7-2 堆栈结构

编码器/装帧器使用 HDLC 协议作为编码方式, 0x7e 作为帧分隔符, 0x7d 作为转义符。HdlcTranslatorC 中维护了 10 位的状态码。在接收和发送通路上各使用了一位状态码用于表示是否使用了转义符, 另外在传输路径上还有一个字节用于决定何时发送转义符。

当 HdlcTranslatorC 收到分隔符时, 它设置 receiveEscape 位为 true, 当它接收到其他字节时, 先检测 receiveEscape 位是否已设置。若是, 则数据字节与 0x20 异或并清除 receiveEscape 位, 通过触发 dataReceived()完成数据字节接收。

HdlcTranslatorC 在发送端的行为与接收端类似。当需要发送一个与分隔符或转义符相同的字节时, 它设置 transmitEscape 标识位为 true, 同时存储数据字节与 0x20 异或的值, 然后发送转义符。当转义符发送完成后, 接着发送刚才存储的数据字节。

3. 传输协议

协议组件负责读入并发送所有的协议控制包。如果协议组件开始接收数据包, 则它会向分派器组件发送信号并附上接收到的数据字节作为参数。当数据包接收完毕时, 协议组件通知分派器组件, 数据包已接收完成, 同时也告诉它 CRC 校验是否通过。

SerialP 组件用类似于 PPP/HDLC 的帧结构实现了串口协议 (具体参考 RFC1662)。消息分派和缓冲区管理就留给了上层去处理, 主机到节点的通信使用的停等的方式, 而节点到主机用尽力而为 (best effort) 的传输方式。

SerialP 提供了两个面向字节的接口用于收发数据包, 分别是 SendBytePacket 和 ReceiveBytePacket。

在发送方, SerialP 负责封装上层的数据包。上层组件如 SerialDispatcherC 调用 startSend()初始化包的发送并传递第一个要发送的字节。SerialP 通过触发 nextByte()来收集接下来要发送的数据。在 nextByte 运行期间或者在调用 nextByte()的间隙, 协议上层调用 completeSend()来表明已经到数据包的结尾。如果 completeSend 是在 nextByte()运行期间被调用的, SerialP 会忽略调用 nextByte()的返回值。SendBytePacket 接口代码如下:

```
interface SendBytePacket {
    async command error_t startSend(uint8_t first_byte);
    async command error_t completeSend();
    async event uint8_t nextByte();
    async event void sendCompleted(error_t error);
}
```

SerialP 用一个小的窗口来存放上层接收的字节以及还没有被发送到 UART 的字节。窗口的大小取决于 UART 的时间需求, 同时通过反复的调用 nextByte()来填充窗口。SerialP 使用 SerialFrameComm 来发送帧间的分隔符、数据包字节和两字节的循环冗余校验(CRC)码。一帧数据发送结束并且接收到底层的最后一个 putDone()事件后, SerialP 调用 sendCompleted()表明此帧发送是否成功。SerialP 还负责数据包的接收工作, 并且向高层提供 ReceiveBytePacket 接口。ReceiveBytePacket 接口代码如下:

```
interface ReceiveBytePacket {
    async event error_t startPacket();
    async event void byteReceived(uint8_t b);
}
```

```

        async event void endPacket(error_t result);
    }

```

一旦接收到帧间分隔符和一个新帧的帧头，SerialP 向高层发出信息表明一个数据包已经到达。每收到一个字节，SerialP 调用一次 `byteReceived()`。当收到一个完整的帧时，SerialP 调用 `endPacket` 并返回 `SUCCESS`。反之，若在接收过程中失去同步，SerialP 调用 `endPacket` 并返回 `FAIL`。SerialP 会对收到的帧发出确认信息。确认信息的优先级比数据传输高，因此，数据帧会被延迟发送。然而，确认信息存储在与数据缓冲区不同的队列中，因此当 SerialP 正在发送确认信息时，其他待发送的数据包进入后台运行。仅有 PC 到 mote 方向的通信才支持确认功能。SerialP 不需要获得 PC 的确认信息，原因有两个。首先，确认机制不是很可靠，在 PC 到节点的通信中使用确认机制是为了将可靠性提高到可用级别。在 PC 到节点方向的通信路径中，典型的 UART 接收缓存仅有一个字节大小，因此高负荷中断很容易丢失字节。PC 的接收缓存则要大得多，并且不用处理溢出问题。其次，增加确认机制会增加串口通信栈的代码长度和复杂度。这样的结果就是以消耗宝贵的代码空间来换取很少需要的功能。当然，任何对可靠性要求较高的应用程序都会将其任务计划基于串口协议栈。确认协议采用停等机制以最少占用节点的缓存资源。在 PC 到节点方向的通信过程中，利用停等机制节省缓存，在存储受限的设备上要比增加吞吐量更加重要，绝大多数的应用程序仅偶尔进行传输控制的时候才使用后者。

4. 分派器

分派器组件处理数据包字节与分隔符。它负责将数据读入 `message_t` 并告知上层组件数据包已接收完毕。Dispatcher 组件支持多种包格式，基于 `message_t` 的工作方式需要知道数据包的包头大小以计算出数据的偏移。

SerialDispatcherC 负责处理协议组件接收的数据包。它使用 `SendBytePacket` 和 `ReceiveBytePacket` 接口，提供带参数的 `Send` 和 `Receive` 接口。`Send` 和 `Receive` 接口的参数 (`uart_id_t`) 决定了 `message_t` 中数据包的格式。SerialDispatcherC 在通过 SerialP 发送和接收的数据包中放置了一字节的包头和包格式标识符。通过使用带参数的 `SerialPacketInfo` 接口，SerialDispatcherC 能够处理各种数据包格式。SerialPacketInfo 接口代码如下：

```

interface SerialPacketInfo {
    async command uint8_t offset();
    async command uint8_t dataLinkLength(message_t* msg, uint8_t upperLen);
    async command uint8_t upperLength(message_t* msg, uint8_t dataLinkLen);
}

```

SerialDispatcherC 收到由 SerialP 发出的第一个字节后，先将其存储为数据包类型，同时调用 `offset()` 以在 `message_t` 中确定偏移量。然后将数据字节导入并填充到 `message_t` 的缓冲区中。发送过程与此类似，SerialDispatcherC 先发送类型字节并导出数据字节，数据字节的起始位置由调用 `offset()` 函数得到的索引值确定。SerialDispatcherC 使用了两个长度命令：`dataLinkLength()` 和 `upperLength()` 以在两种数据包类型中进行转换。向上，长度为不含数据包头部的有效载荷长度；向下，长度则为包含了数据包头的长度。在使用了串口端口参数：`uart_id_t U` 提供通信服务的组件中，必须在组件实现中通过 `uart_id_t U` 将 `SerialPacketInfo` 连接到 `SerialDispatcherC`。目前，仅

有平台独立的活动消息（TOS_SERIAL_ACTIVE_MESSAGE_ID，在 3.5 节中有详细描述）、802.15.4 的活动消息（TOS_SERIAL_802_15_4_ID）、mica2 平台的 CC1000 数据包（TOS_SERIAL_CC1000_ID）和错误代码 TOS_SERIAL_UNKNOWN_ID 是保留标识符，新的数据包格式绝对不可以使用任何保留标识符。

5. SerialActiveMessageC

SerialActiveMessageC 是平台独立的活动消息层组件，工作于串口通信栈顶端的配置文件，它将 SerialActiveMessageP 连接到带 uart_id_t 参数的 SerialDispatcherC，同时将 SerialPacketInfoActiveMessageP 连接到带参数 uart_id_t TOS_SERIAL_ACTIVE_MESSAGE_ID 的 SerialDispatcherC。

```

includes Serial;
configuration SerialActiveMessageC {
  provides {
    interface Init;
    interface AMSend[am_id_t id];
    interface Receive[am_id_t id];
    interface Packet;
    interface AMPacket;
  }
  uses interface Leds;
}
implementation {
  components new SerialActiveMessageP() as AM, SerialDispatcherC;
  components SerialPacketInfoActiveMessageP as Info;

  Init = SerialDispatcherC;
  Leds = SerialDispatcherC;

  AMSend = AM;
  Receive = AM;
  Packet = AM;
  AMPacket = AM;

  AM.SubSend -> SerialDispatcherC.Send[TOS_SERIAL_ACTIVE_MESSAGE_ID];
  AM.SubReceive -> SerialDispatcherC.Receive[TOS_SERIAL_ACTIVE_MESSAGE_ID];

  SerialDispatcherC.SerialPacketInfo[TOS_SERIAL_ACTIVE_MESSAGE_ID] -> Info;
}

```

SerialActiveMessageP 是一个通用组件，因此可用于各种数据包级的通信层顶端。它不以目的地址或组号过滤数据包，而是假设从串口端口接收的数据包目的地址就是本节点。这样 PC 端的工具就不用再寻找和考虑节点的 ID 和组号了。平台无关的活动消息没有 CRC（通常假设其串口栈的 CRC 是足够的），其头部格式如下：

```

typedef nx_struct SerialAMHeader {
  nx_am_addr_t addr;
}

```

```

    nx_uint8_t length;
    nx_am_group_t group;
    nx_am_id_t type;
} SerialAMHeader

```

6. 数据包格式

TinyOS 2.x 的串口栈里，数据包在连接过程中有如下的格式。各协议字段分别和特定的组件对应，如图 7-3 所示。



图 7-3 数据包格式示意图

F=帧字节: HdlcTranslateC

P=协议字节: SerialP

S=序列字节: SerialP

D=数据包格式分派字节: SerialDispatcherC

Payload=有效负载: SerialDispatcherC

CR=两字节的 CRC 校验: SerialP

F=帧字节: HdlcTranslateC

Payload 是由组件 SerialDispatcherC 读入的连续数据包。需要注意: payload 中任何值为 0x7e 和 0x7d 的字节都将被相应转化为 0x7d 0x5e 或者 0x7d 0x5d。例如, 一个目标地址为 0xbeef, 类型为 6, 组号为 0x7d, 长度为 5 的平台独立活动数据包为: 7e 40 09 00 be ef 05 7d 5d 06 01 02 03 04 05 7e 注意: 组号 0x7d 被转化为 0x7d 0x5d。协议字段 (P) 值为 0x40 (64), 与 SERIAL_PROTO_ACK 一致 (SERIAL_PROTO_ACK 在 Serial.h 文件中有说明)。

7.3.2 MIG

Listen 应用程序是传感器节点大多数通信方式中最基本的一种, 它仅仅把数据包的二进制内容输出在屏幕上, 明显, 这样并不能很清晰的理解数据包所传递的内容。基站节点将从无线传感器网络中接收到的数据包通过串口等方式发送给 PC, 为了清晰、形象的呈现出数据包承载的内容, TinyOS 有一些应用程序来完成数据包的解析和显示, 还支持用户根据自身需求创建一些新的应用实现对数据包的处理。

MIG (Message Interface Generator)即消息接口产生器, 是一个用于自动产生 Java 类的工具, 对应于微粒应用程序中使用的活动消息 (Active Message) 类型。MIG 读取在微粒应用程序中使用的消息类型的 nesC 结构定义, 并为每个消息类型产生一个 Java 类, 以处理消息字节格式中的各字段打包、拆包等细节问题。使用 MIG 可以避免在 Java 应用程序中解析消息格式的麻烦。

MIG 用来与 net.tinyos.message 包联合起来, 并通过 MIG 产生的消息类提供大量发送和接收消息的例程。NCG(nesC Constant Generator) 即 nesC 常数产生器, 它从 nesC 文件中提取常数以供其他应用程序使用, 用于与 MIG 连接。

TestSerial 应用程序就使用到了 MIG，能够很方便的解析和显示通过串口上传的数据。在 Cycwin 界面中进入 TestSerial 所在路径，输入：make clean；make telosb，应该有如图 7-4 所示内容。

```

yang@PC2011010811gyu /opt/tinyos-2.x/apps/tests/TestSerial
$ make clean
rm -rf build *.class TestSerialMsg.java pp
rm -rf _TOSSIMmodule.so TOSSIM.pyc TOSSIM.py app.xml simbuild
rm -rf VolumeMapC.nc

yang@PC2011010811gyu /opt/tinyos-2.x/apps/tests/TestSerial
$ make telosb
mkdir -p build/telosb
mig java -target=null -I/opt/tinyos-2.x/tos/lib/T2Hack -DIDENT_APPNAME="TestSerialAppC" -DIDENT_USERNAME="yang" -DIDENT_HOSTNAME="PC2011010811gyu" -DIDENT_USERHASH=0x33c2c2a2L -DIDENT_TIMESTAMP=0x4e7a9316L -DIDENT_UIDHASH=0x24be6afcL -java-classname=TestSerialMsg TestSerial.h test_serial_msg -o TestSerialMsg.java

javac -target 1.4 -source 1.4 *.java
    compiling TestSerialAppC to a telosb binary
ncc -o build/telosb/main.exe -Os -O -mdisable-hwmul -Wall -Wshadow -Wnec-all -target=telosb -fnesc-cfile=build/telosb/app.c -board= -DDEFINED_TOS_AM_GROUP=0x22 -I/opt/tinyos-2.x/tos/lib/T2Hack -DIDENT_APPNAME="TestSerialAppC" -DIDENT_USERNAME="yang" -DIDENT_HOSTNAME="PC2011010811gyu" -DIDENT_USERHASH=0x33c2c2a2L -DIDENT_TIMESTAMP=0x4e7a9316L -DIDENT_UIDHASH=0x24be6afcL TestSerialAppC.nc -ln
    compiled TestSerialAppC to build/telosb/main.exe
        6372 bytes in ROM
        304 bytes in RAM
msp430-objcopy --output-target=ihex build/telosb/main.exe build/telosb/main.ihex

writing TOS image
搜狗拼音 半:

```

图 7-4 编译 TestSerial 应用程序

在 TestSerial 的程序中，利用了 MIG 来生成对于 BlinkToRadio 数据包的 MSg 代码，当编译是会出现如下信息：

```

mig java -target=telosb -I%/lib/oski -java-classname=TestSerialMsg TestSerial.h TestSerialMsg -o TestSerialMsg.java

```

此段即为生成 TestSerialMSg.java 编译时的代码，实现此段的代码在 Makefile 中：

```

COMPONENT=TestSerialAppC
BUILD_EXTRA_DEPS += TestSerial.class
CLEAN_EXTRA = *.class TestSerialMsg.java
TestSerial.class: $(wildcard *.java) TestSerialMsg.java
javac *.java
TestSerialMsg.java:
mig java -target=null -java-classname=TestSerialMsg TestSerial.h TestSerialMsg -o $@
include $(MAKERULES)

```

BUILD_EXTRA_DEPS += TestSerial.class: 代表在编译程序之前需要先编译 TestSerial.class。

CLEAN_EXTRA = *.class TestSerialMsg.java:指定在用户执行 make clean 时清除的代码。

TestSerial.class: \$(wildcard *.java) TestSerialMsg.java: TestSerial.class 的生成需要本目录中的所有 java 文件，执行完后进行编译。

mig: 使用 mig。

java: 建立 java 的 class。

-target=null: 表示针对 null 平台。

-java-classname=TestSerialMsg: java 类的名字。

TestSerial.h: 定义结构体的文件。

TestSerialMsg: 结构体的名字。

-o \$@ : 写文件到\$@, 也就是 TestSerialMsg.java。

7.3.3 数据包源

数据包源是传感器节点与 PC 之间不同通信媒介的抽象，可以用来接收节点上传的数据包并能向节点发送数据包。主要包括直接串口连接、通过以太网远程串口连接、通过模拟节点连接、通过串口代理连接（SerialForwarder）等方式。

数据包源的格式是 connection@arguments，其中，connection 就是上面介绍的几种连接方式的其中之一，arguments 就是在该连接方式下的具体参数配置，比如 serial@COM1:115200 就表示通过串口直接连接并且使用串口端口 COM1，波特率 115200B。执行命令：

```
java net.tinyos.packet.BuildSource
```

可以输出所有的数据包源和配置，如图 7-5 所示。

```
$ java net.tinyos.packet.BuildSource
- sf@HOSTNAME:PORTNUMBER
  A serial forwarder.
- serial@SERIALPORT:BAUDRATE
  A mote connected to a serial port using the TinyOS 2.0 serial protocol.
  BAUDRATE is either a number or a platform name (selects platform's
  default baud rate).
- network@HOSTNAME:PORTNUMBER
  A mote whose serial port is accessed over the network.
- tossim-serial[CHOSTNAME]
  The serial port of tossim node 0.
- tossim-radio[CHOSTNAME]
  The radios of tossim nodes.

Examples: serial@COM1:mica2, serial@dev/ttyUSB2:19200, sf@localhost:9000
```

图 7-5 命令执行界面

SerialForwarder 程序用来从串口读取数据包的数据并将其在互联网上转发，这样就可以写一些其他的程序通过互联网来与传感器网络进行通信。

SerialForwarder 是一种更为重要的数据包源。SerialForwarder 是一个 Java 程序，它与节点连接并且可以与多个客户端交互数据，其中这些客户端通过 TCP/IP 协议与 SerialForwarder 连接。使用 SerialForwarder 有如下好处：

1. 可以远程接入 Sink 节点。比如远端的 GUI 可以通过 SerialForwarder 方便的接入传感器网络。

2. 可以使普通的应用程序和 debug 系统同时与一个 Sink 通信。比如利用 SerialForwarder 显示接收到某个节点发出的的所有数据包，而不必频繁的改动代码程序。

3. 可以将一个应用程序分割成几个独立的部分，各个部分都可以和同一节点通信。比如 TestSerial.java 应用程序可以通过 SerialForwarder 分割成单独的接受和发送两个部分分别与节点通信。

前面提到数据包源有其特有的格式，SerialForwarder 的数据包源格式是~@HOST:PORT，其中 HOST 和 PORT 是可选的，默认值分别为当地主机名和 9002，如图 7-6 所示。

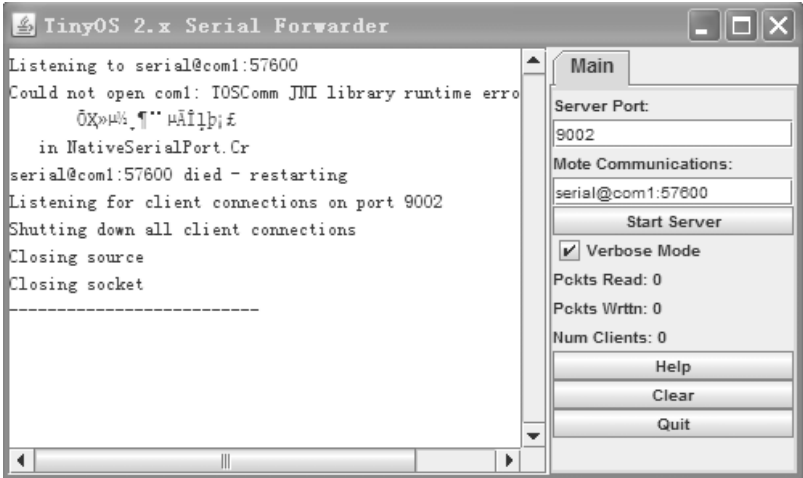


图 7-6 数据包源格式

7.3.4 串口通信测试

TestSerial 应用程序对节点和 PC 之间是否可以正常通信进行测试。节点和 PC 每秒会通过串口互发一个数据包，当节点收到串口发送的数据包时其数据部分会通过节点的 LED 灯相应的显示，而 PC 收到串口上传的数据包时也会将其内容打印出来。TestSerial 使用的扩展数据类型定义在 TestSerial.h 头文件中，如图 7-7 所示。

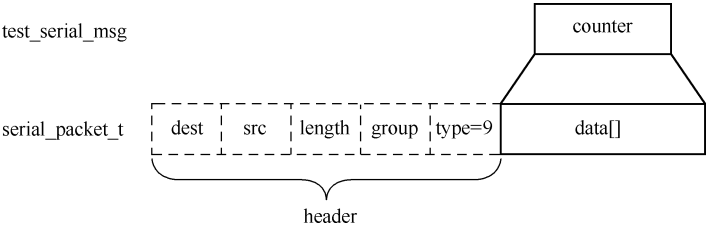


图 7-7 TestSerial 数据类型

```
TestSerial.h

typedef nx_struct test_serial_msg {
    nx_uint16_t counter;
} test_serial_msg_t;
```

1. 设置串口

```
$ java net.tinyos.tools.Listen -comm serial@COM1:telos
```

在 windows 的机器上设置监听工具使用 COM1 串口以 telos 节点的码率通信。

```
$ java net.tinyos.tools.Listen -comm serial@/dev/ttyS0:micaz
```

在 UNIX 机器上设置监听工具使用 COM1 串口以 micaz 节点的码率通信。

在安装 TinyOS 的 windows 电脑上找到 Cycwin 文件夹下 apps/tests/TestSerial, 将其编译下载到节点。如果不设定数据包源, 则默认的数据包源是 SerialForwarder, 由于不是 SerialForwarder 的连接方式, 因此会报错无法连接, 退出 TestSerial 应用。在这里我们设置数据包源为 Serial, 具体格式如前所述: serial@<PORT>: <SPEED>, 其中 PORT 是端口号, SPEED 是数据传输波特率。

```
class BaudRate {
    static void init() throws Exception {
        /* The Platform.x argument is there for when this code is #include'd
           into C */
        Platform.add(Platform.x, "mica",      19200);
        Platform.add(Platform.x, "mica2",     57600);
        Platform.add(Platform.x, "mica2dot",  19200);
        Platform.add(Platform.x, "telos",     115200);
        Platform.add(Platform.x, "telosb",    115200);
        Platform.add(Platform.x, "tinynode",   115200);
        Platform.add(Platform.x, "tmote",     115200);
        Platform.add(Platform.x, "micaz",     57600);
        Platform.add(Platform.x, "eyesIFX",   57600);
        Platform.add(Platform.x, "intelmote2", 115200);
        Platform.add(Platform.x, "iris",      57600);
        Platform.add(Platform.x, "shimmer",   115200);
    }
}
```

7.4 节点之间通信

在无线传感器网络中传感器节点上一般都配置有无线射频模块, 节点之间的通信主要是通过无线方式来完成, TinyOS 作为一个比较完善的操作系统, 实现节点之间的通信是其必不可少的功能之一, 本节将详细为读者分析和介绍使用 TinyOS 的相应接口和组件来实现无线通信。

7.4.1 AM 通信

在节点之间无线通信过程中通常有多种服务都要接入无线射频模块, TinyOS 提供了 Active Message (AM)层对无线射频模块的复用, 每个数据包被定义成 AM 类型, 并通过一个

字节来辨别该数据包类型。由于这种类型往往自动将接收到的数据包分发给适当的处理者，就像是信息可以主动地寻找并到达目的地，因此这种类型的通信方式被称为 Active Message。以下是两种 AM 接口，在目录 tos/interface 中：

AMPacket: 对 message_t 抽象数据类型进行基本 AM 存取。这个接口提供了获得一个节点 AM 地址、AM 包的目的地和 AM 包类型的命令，也提供了设置 AM 包目的地和类型，并检查目的地是否为本地节点。

AMSend: 相似于 Send 接口，提供了基本的 AM 发送接口。这二者的关键差别是 AMSend 是在 send 命令中获得目的地的 AM 地址。

另外很多组件实现了基本通信和 active message 接口，以下是在 tos/system 目录下的组件。

AMReceiverC: 提供了接口：Receive, Packet, AMPacket。

AMSenderC: 提供了接口：AMSend, Packet, AMPacket, PacketAcknowledgements as Acks。

AMSnooperC: 提供了接口：Receive, Packet, AMPacket。

AMSnoopingReceiverC: 提供了接口：Receive, Packet, AMPacket。

ActiveMessageAddressC: 提供了获得和设置 AM 地址的命令。这个接口一般不怎么使用，因为改变节点的 AM 地址可能会破坏网络堆栈。

1. ActiveMessageC

由于 TinyOS 支持多种硬件平台，每个平台都有其自己的无线射频芯片驱动，ActiveMessageC 是 AM 接口和下层实现的桥梁，提供了大多数 AM 通信所需的接口。ActiveMessageC 根据其平台的不同进行不同的命名，主要根据其射频芯片。其中

eyesIFX 平台的 ActiveMessageC，由 Tda5250ActiveMessageC 实现。

intelmote2, micaz, telosa 和 telosb 平台的 ActiveMessageC，由 CC2220ActiveMessageC 实现。

mica2 平台的 ActiveMessageC 由 CC1000ActiveMessageC 实现。

以下是 CC1000ActiveMessageC 的代码实现如下：

```
configuration ActiveMessageC {
  provides {
    interface SplitControl;
    interface AMSend[uint8_t id];
    interface Receive[uint8_t id];
    interface Receive as Snoop[uint8_t id];
    interface Packet;
    interface AMPacket;
    interface PacketAcknowledgements;
  }
}

implementation {
  components CC1000ActiveMessageC as AM;
  SplitControl = AM;
```

```

    AMSend = AM;
    Receive = AM.Receive;
    Snoop = AM.Snoop;
    Packet = AM;
    AMPacket = AM;
    PacketAcknowledgements = AM;
}

```

2. message_t

TinyOS 提供了许多组件，这些组件提供了许多接口来抽象底层通信服务。这些接口和组件均使用一个共同的消息缓冲器抽象，被称为 `message_t`。`message_t` 代替了 TinyOS1.x 中的 `TOS_Msg` 提取。`message_t` 是不透明的，因此它是不可以直接访问的。"message_t"类型的主要目的是允许报文作为内存的一个连续存储区域以零拷贝的方式在不同的链路层传输。

在 TinyOS2.x 中，标准的消息缓存是 `message_t`。`message_t` 结构在"tos/types/message.h"中定义。`message_t` 的代码实现如下：

```

typedef nx_struct message_t {
    nx_uint8_t header[sizeof(message_header_t)];
    nx_uint8_t data[TOSH_DATA_LENGTH];
    nx_uint8_t footer[sizeof(message_footer_t)];
    nx_uint8_t metadata[sizeof(message_metadata_t)];
} message_t;

```

此格式将数据字段保持在一个固定的偏移量上，这对于在两个不同的链路层之间传送数据是非常重要的。如果数据负载字段对于不同的链路层有不同的偏移量，那么将一个包从一个链路层发送到另一个链路层就需要使用"memmove"操作（需要使用拷贝）。在 TinyOS 1.x 中 `TOS_Msg` 是一个明确的 active messaging 报文，而 `message_t` 是一个更泛化的数据链路缓存。实际上，大多数 TinyOS 2.x 数据链路层都提供 active messaging，同时也可以由 AM 协议栈向非 AM 协议栈传输 `message_t` 报文。

`message_t` 的头部，尾部，元数据对于上层协议来说都是不透明的，不能直接存取这些域。而数据链路层提供从 nesC 接口，通过它们来实现访问结构内部的字段。

每个不同的链路层都会定义它们自己的头，尾，元数据字段。这些字段必须是 `external` 结构("nx_struct")，这些结构的所有成员必须是 `external` 类型("nx_")。这样做有两个原因：一、`external` 类型能确保跨平台的兼容；二，它会强制结构在字节边界上对齐，从而避免缓冲区及其内部的字段的对齐问题。在整个包被发往串口被记录通信日志时，元数据字段也必须是 `nx_structs` 的。

例 7.1：CC1000 的射频实现在其"CC1000Msg.h"中定义 `message_t` 结构

```

typedef nx_struct cc1000_header {
    nx_am_addr_t addr;
    nx_uint8_t length;
    nx_am_group_t group;
    nx_am_id_t type;
} cc1000_header_t;

```

```

typedef nx_struct cc1000_footer {
    nxle_uint16_t crc;
} cc1000_footer_t;

typedef nx_struct cc1000_metadata {
    nx_uint16_t strength;
    nx_uint8_t ack;
    nx_uint16_t time;
    nx_uint8_t sendSecurityMode;
    nx_uint8_t receiveSecurityMode;
} cc1000_metadata_t;

```

每个数据链路层都要定义它自己的结构体，而一个平台要负责定义其 `message_header_t`, `message_footer_t` 和 `message_metadata_t`。这是因为一个平台可能有多个数据链路层，通过以上三个部分来决定有哪些结构体是需要的。这些必须定义在平台目录下的 `platform_message.h` 文件中。

例 7.2：在 `mica2` 平台下有两个数据链路层：CC1000 射频模块和 TinyOS 串口协议栈，在 `tos/platforms/mica2/platform_message.h` 文件中如下代码：

```

typedef union message_header {
    cc1000_header_t cc1k;
    serial_header_t serial;
} message_header_t;

typedef union message_footer {
    cc1000_footer_t cc1k;
} message_footer_t;

typedef union message_metadata {
    cc1000_metadata_t cc1k;
} message_metadata_t;

```

一般来说串口数据包的格式中没有 `footer` 或者 `metadata` 部分，在代码中可以看到 `message_footer` 中只有 `cc1000_footer_t cc1k`。

例 7.3：假如一个名为“megamica”的平台同时有 CC1000 和 CC2420 射频，那么它的“`platform_message.h`”就应该如下：

```

typedef union mega_mica_header {
    cc1000_header_t cc1k;
    cc2420_header_t cc2420;
    serial_header_t serial;
} message_header_t;

typedef union mega_mica_footer {

```

```

        cc1000_footer_t cc1k;
        cc2420_footer_t cc2420;
    } message_footer_t;

    typedef union mega_mica_metadata {
        cc1000_metadata_t cc1k;
        cc2420_metadata_t cc2420;
    } message_metadata_t;

```

如果一个平台有多个链路层，那么它应该将 `message_t` 的每一个 `header`, `footer`, `metadata` 字段分别定义成链路层结构的联合字段，这样能保证在结构中为所有的链路层都分配足够的空间。

3. The message fields

TinyOS 2.x 组件将包看做是一个抽象数据类型(ADTs)，而非一个 C 语言结构体，这样做有比较明显的优势，链路层的客户并不依赖于结构中特定的字段名字和位置，这就使其能够选择包的结构，增强了操作的自主性，另外还在其他方面还作出了各种各样的优化。

在数据链路层之上的组件必须从接口中存取包的字段。一个引入新的包字段的组件应该要为其其他与其交互的组件提供一个访问接口。这些结构应当实现存取数据的 `get/set` 操作，而不是给出包结构中相关字段的偏移量。

例如在 `active message` 中的 `AMPacket` 接口，它能提供接入 AM 字段的命令。在 TinyOS 1.x 中，组件能直接访问 `TOS_Msg.addr`，在 TinyOS 2.x 中，组件需要调用 `AMPacket.getAddress(msg)`。这些接口中最基础的是 `Packet` 接口，它能提供存取一个包的数据负荷的命令。

链路层组件在存取包字段时可能不同于其他组件，它们关心具体的包格式，因此它们要实现负责为其他组件实现访问这些字段的接口。

`message_t` 的头部是一个字节数组，它的长度是一个硬件平台的数据链路头部组合的长度。这些射频协议栈倾向于将数据包连续地存储起来，因此包在内存中的布局并不一定需要反映其 `nesC` 结构的布局。

一个包的头部字段可能会在 `message_t` 的某个位置开始，而并非一定在 `message_t` 的首个字节处。

例 7.4：在 Telos 平台中：

```

typedef union message_header {
    cc2420_header_t cc2420;
    serial_header_t serial;
} message_header_t;

```

CC2420 头部有 11 个字节长，而串口头部才 5 个字节。串口头部结尾处紧挨着数据字段的开始处，因此需要在串口头部前填充 6 个字节。下图显示了 `message_t` 的布局：一个 10 字节的 CC2420 包和一个 12 字节的串口包。

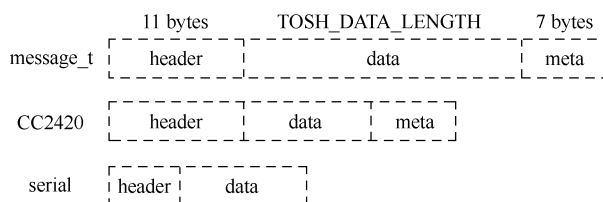


图 7-8 message_t 的布局

CC2420 包和串口包都没有 footer 字段，串口包没有任何元数据。

数据链路层包头字段并不需要在 message_t 的开始处，它在数据字段开始处的一个负偏移位置上。当一个链路层组件需要读写包头部字段时，它必须从数据字段开始处减去一个偏移量来计算头部字段所在位置。例如，串口协议栈头部有一个 active message 字段，是 AM 类型，返回 AM 类型的“AMPacket.type”命令如下：

```

serial_header_t* getHeader(message_t* msg) {
    return (serial_header_t*)(msg->data - sizeof(serial_header_t));
}

command am_id_t AMPacket.type(message_t* msg) {
    serial_header_t* hdr = getheader(msg);
    return hdr->type;
}

```

直接计算负的偏移位置是不方便的，所以串口协议栈使用内部的 getHeader 函数来帮助完成此事。很多其他单跳协议也采用了此方法，nesC 编译器会将此函数内联化，以降低开销，大多数情况下 C 编译器会将其编译成一个偏移寻址。

在 message_t 的数据字段中存储了单跳的包负荷。它的长度是 TOSH_DATA_LENGTH，默认值 28 字节。TinyOS 应用能在编译时重新定义 TOSH_DATA_LENGTH，重新定义使用 ncc 的一个选项：-DTOSH_DATA_LENGTH=x。因为这个值是能重新配置的，所以有可能存在 2 个不同版本的应用，它们的 MTU 是不同的。如果接收到的包数据长度大于 TOSH_DATA_LENGTH，那么它必须将包丢弃。因为头部正好在数据字段之前，所以一个平台上所有的链路层数据字段都有一个相对 message_t 缓冲区并且都有一个固定的偏移量。

message_footer_t 域确保 message_t 为所有的底层链路层提供足够的空间来存储尾部。就像 header 一样，footer 并不必像 C 结构所表示的那样存储在内存中，相反，它们是依赖于特定实现的。一个单跳的层次可能将 footer 紧接在数据字段后。对于长度短的包而言，这就意味着 footer 字段实际上是被存储在数据域中的。

message_t 的元数据字段用于收集单跳协议栈所需要的元数据信息，而非用于传输。这种机制使得基于包的协议层能为每个包存储 RSSI，时间戳等信息。

例 7.5：CC2420 元数据类型结构：

```

typedef nx_struct cc2420_metadata_t {
    nx_uint8_t tx_power;
    nx_uint8_t rssi;
    nx_uint8_t lqi;
    nx_bool crc;
}

```

```

    nx_bool ack;
    nx_uint16_t time;
} cc2420_metadata_t;

```

`message_t` 结构为固定长度的 `header` 和 `footer` 优化过，可变长的 `footers` 通常是容易实现的。可变长的 `header` 则稍微困难。有三种通用方法能使用。

如果链路硬件是基于字节的，那么头部可以存储在 `message_t` 结构的开始处，这样有一个可知的偏移量。头部和数据字段之间可能会填充其他值。

如果链路硬件是基于包的，那么协议栈既能包含指示头部开始的元数据，或者把头部放在固定位置，也能在接收和发送端使用 `memmove(3)`。在接收端和发送端使用 `memmove` 的情况下，接收端接收包，将其持续地读入 `message_t` 的 `header` 处，一旦包被完整地接收了，头部就会被解码，以获得数据字段的长度，然后此包的数据字段就能被拷入 `message_t` 的数据域中。要注意的是，一旦传输完成，它们就要被拷贝回来。可选的方法是射频协议栈在底层保存一个包的拷贝。

7.4.2 单跳发送数据包

1. 重新编写 Blink

我们将创建一个简单的应用，实现计数的增加，将计数的三个最低有效位在三个 LED 灯上显示，并通过无线方式将计数消息发出。

```

$ cd tinyos-2.x/apps //进入应用程序 apps 目录
$ mkdir BlinkToRadio //创建 BlinkToRadio 文件夹

```

创建 `BlinkToRadio.nc` 文件，以下是其具体代码实现。

```

#include <Timer.h> //本行是告诉编译器用 Timer.h 中的全部内容完全代替 directive。编译器在一个标准的地方来寻找 Timer.h 文件，这种情况下，标准文件一般位于 tos 或它的子目录下。也可以在文件 Makefile 中使用-I 标记来告诉编译器标准目录。

```

```

#include "BlinkToRadio.h" //引号告诉预处理程序在寻找合法的文件时先在本目录下寻找，在这种情况下，BlinkToRadio.h 文件位于相同的目录下并且定义了一些在本程序中使用的常量。

```

```

module BlinkToRadioC {
    uses interface Boot;
    uses interface Leds;
    uses interface Timer<TMilli> as Timer0;
}

implementation {
    uint16_t counter = 0;
    event void Boot.booted() {
        call Timer0.startPeriodic(TIMER_PERIOD_MILLI);
    }
    event void Timer0.fired() {
        counter++;
        call Leds.set(counter);
    }
}

```

创建 BlinkToRadio.h 头文件，在文件中定义用到的常量和数据结构。

```
#ifndef BLINKTORADIO_H
#define BLINKTORADIO_H
enum {
    TIMER_PERIOD_MILLI = 250;
};
#endif
```

注释：BlinkToRadio.h 是一个标准的头文件，但这里应注意两点：第一：注意 ifndef、define 和 endif 的使用，它们会确保每个头文件中的定义不被多次定义，因为编译器会对多次定义提出警告。

第二：使用 enum 定义了常量 TIMER_PERIOD_MILLT。使用 enum 定义常量优于使用 define，因为 enum 不会轻易地代替已定义的每个常量值。

接下来，要编写配置文件 BlinkToRadioAppC.nc，将用到的组件导通：

```
#include <Timer.h>
#include "BlinkToRadio.h"
configuration BlinkToRadioAppC {
}
implementation {
    components MainC;
    components LedsC;
    components BlinkToRadioC as App;
    components new TimerMilliC() as Timer0;
    App.Boot -> MainC;
    App.Leds -> LedsC;
    App.Timer0 -> Timer0;
}
```

另外还需一个 Makefile 文件：

```
COMPONENT=BlinkToRadioAppC //告诉 TinyOS 编译系统顶层应用组件是 BlinkToRadioAppC
include $(MAKERULES)
```

2. Dissemination 分发

分发是一种基本的传感器网络协议，主要用于实现基于共享变量的网络一致性。网络中的每一个节点都有该变量的一个备份，这种小数据分发服务会通知节点该变量值更改的时间，同时交换数据包以达到整个网络的一致性。在任意给定时段，也许会有两个节点不同意交换数据包，但是随着时间的流逝，不同意的节点数会越来越少，最终整个网络完全依赖于一个变量值(converge on a single value)。

网络的高度一致性能有效避免临时性的链接失效以及高丢包率。与泛洪协议(flooding protocols)不同，泛洪协议主要是离散性的工作(节点与节点之间不受某一变量值约束)，它能够终止并且不再达成网络的一致性，小数据分发机制确保网络内部在有可靠连接的情况下能够达到基于单个变量值的一致性。

小数据分发协议会因为数据项(data item)的不同而不同：相比于分发两个字节的配置字到网络中，有效的分发几十个 KB 的二进制数据流时需要另外的协议。但是，深入研究一下，会发现二者其实还是有相同点的，两种协议并非完全不同。把小数据分发协议分成两个部分--分别是控制量部分(control traffic)和数据量部分(data traffic)。其中数据通信量(data traffic)协议依赖于数据项的大小，而控制通信量(control traffic)协议大致相同。例如，Deluge-这种二进制重编程服务以二进制形式分发 metadata。当网络中的节点发现收到的 metadata 与自身的 metadata 不同时，他们就会意识到自身原来的二进制信息已经失效了，需要一个新的二进制信息。

Noverlty 协议在小数据分发一致性模型上做了很周到的考虑，致力于让网络中的每一个节点都接纳分发到网络中的变量值的最新版本。按照这种方式，节点可以通过告诉网络变量值被更新从而促进网络达成一致。如果有好几个节点决定更新变量值，小数据分发协议就可以确保网络只需要一次更新便可以确保达成一致。

网络的这种一致性并不意味着每一个节点都能够接收到变量值，这种一致性仅仅表示网络最终会在哪个最新变量值这个问题上达成一致。如果有一个节点从网络中断开并且此后网络经过 8 次更新才得到共享变量，则当该节点重新加入网络后，它所接收到的变量值只会是最后一个更新所得到的变量值。

能够将小数据分发到整个网络中，这对于传感网应用而言是重要的组成部分。它允许管理员向网络中插入小段程序，命令以及配置字。比如，安装一小段程序到整个网络中就相当于建立网络一致性的问题，该一致性通过包含这段程序的一个变量来建立。

小数据分发协议有两个接口：DisseminationValue 和 DisseminationUpdate。前者适用于分发数据的消费者，后者便适用于生产者。详细代码如下：

1)

```
interface DisseminationValue<T> {
    command const T* get();
    command void set(const T*);
    event void changed();
}
```

2)

```
interface DisseminationUpdate<T> {
    command void change(T* newVal);
}
```

这些接口假定在分发服务中分配空间来储存变量值。这样一来，多个组件都可以访问该变量并且还能共享分发服务建立网络一致性所基于的相同的变量。消费者能够通过 DisseminationValue.get()指向的数据区域获取 CONST 类型的指针，但它不能存储这个指针，因为网络更新过程中可能会导致指针值的改变。另外，由于该指针可以很容易就被获取，这么做会浪费 RAM,DisseminationValue.get()只要分发的变量值改变的时候就会触发 change()事件，因此在这种情形下消费者需要执行一些运算或者采取相应的举措。

DisseminationValue 接口有一个命令"set"，该命令允许节点在不以建立网络一致性的情况

下改变其当前的变量值，这个命令的存在可以使节点为变量分配一个初始值。节点不能在它已经处理了"changed"事件之后再调用"set"函数，否则整个网络就可能会变得不一致。如果一个节点已经收到了一个更新信息或者一个客户机已经调用了"change"函数，那么"set"命令不能用来设定新的变量值。

DisseminationUpdate 有个单独的命令"change",该命令函数有一个参数，并且是指针类型。该指针不能被存起来，因为提供 DisseminationUpdate 接口的组件必须将接收到的信息存入自己分配的内存里。

小数据分发协议必须在整个网络中以最新的变量值为桥梁从而达成网络的一致性。间接调用 change 函数使得被传输的数据项得以更新，这样它就可以被分发给网络中的每一个节点。然而，这个 change 函数是局部的，对于一个过时的节点也调用了 change 函数的话，新的变量值可能不会被分发，因为其他的节点或许都有了一个更新的变量值。如果两个节点同时调用了 change 函数但是却在分发不同的变量值，在网络中节点变量值仍旧不同的情况下，整个网络也许可能会达成一致！但是小数据分发协议必须能有一种可以打破僵局的机制，以至每个节点最终都能有相同的变量值。

小数据分发服务必须提供一个组件，DisseminatorC,该组件的声明如下：

```
generic configuration DisseminatorC(typedef t, uint16_t key) {
    provides interface DisseminationValue <t>;
    provides interface DisseminationUpdate <t>;
}
```

考虑到小数据分发协议所介绍的帧头格式，参数 t 必须能够适用于单个 message_t 格式。如果使用了一个更大容量的类型，在实现小数据分发过程中编译器是会报错的。

由于每一个 DisseminatorC 的实例都可能分配存储空间或者产生代码，如果有更多的组件希望共享一个分发的变量值时，那么他们应当在一个不通用但又能被共享的组件中对该变量值进行封装，代码示例如下：

```
configuration DisseminateTxPowerC {
    provides interface DisseminationValue<uint8_t>;
}
implementation {
    components new DisseminatorC(uint8_t, DIS_TX_POWER);
    DisseminationValue = DisseminatorC;
}
```

两个 DisseminatorC 的不同实例不能共用相同变量值的关键参数。

在使用上述接口的过程中会出现这样一个问题，即如何选择变量 key 的值。一方面，采用 unique()很简单，但是这意味着对于同一个程序的两个不同的汇编的参数 key 所占用的空间可能有所不同，且没有办法支持一个以上的二进制网络。另一方面，组件在其内部声明自己的参数 key 意味着会碰到无法解决的参数 key 碰撞问题。因为在这期间，应用程序可以选择其他组件的参数 key。

一般而言，小数据分发的变量参数 key 可独自产生或手动产生。这些定义好的参数 key 可以被应用程序特定的头文件所覆盖。单独的命名空间和静态命名空间被其最重要的二进制

位所分隔。一个组件作者可能会写代码如下：

```
#include <disseminate_keys.h>
configuration SomeComponentC {
    ...
}
implementation {
    #ifndef DIS_SOME_COMPONENT_KEY
        enum {
            DIS_SOME_COMPONENT_KEY = unique(DISSEMINATE_KEY) + 1 << 15;
        };
    #endif
    components SomeComponentP;
    components new DisseminatorC(uint8_t, DIS_SOME_COMPONENT_KEY);
    SomeComponentP.ConfigVal -> DisseminatorC;
}
```

你可以在应用程序目录下的头文件 disseminate_keys.h 中看到：

```
#define DIS_SOME_COMPONENT_KEY 32
```

即使仔细选择参数 key，两个不兼容的参数 key 的二进制存储空间最终在同一个网络以碰撞的形式出现。如果发生这种情况，一个 GUID，参数 key 独有的二进制可包括在该协议内。利用 GUID 可以使节点检测其他二进制版本，而不是储存它们。此 GUID 不会成为外部接口的一部分，但将在内部被使用。

7.5 定时器（Timer）

大部分微控制器都提供了丰富的定时器系统，具有如下特点：

- 多个计数器，具有不同的位数和多种时钟选择
- 每个计数器都有一个或多个比较寄存器，输出管脚变化和计数器值变化可触发中断
- 获取输入管脚变化的时间

TinyOS 不会捕获平台之间的所有差异。相反，根据 HAA 原理，每个微控制器应通过 HPL 层（适当的，也包括 HAL 层）的组件和接口来展现其所有功能。然而，定时器在两个方面即测量时间和特定时间的触发（可能重复）事件，需要很好的定义。

7.5.1 接口

定时器的三个基本属性是精度（precision）、位数（width）和准确性（accuracy）。在讨论接口前，我们先简单的看一下定时器接口中对这三种基本属性的描述。

精度的单位可以是毫秒，32kHz 时钟的一个时钟周期或者微秒。本节中所有相对秒的精度都以二进制表示。也就是说，1 秒包括 1024 二进制毫秒，32768 32kHz ticks，或者是 1048576 微秒。为了合理兼容其他精度，本节主要介绍毫秒和 32kHz ticks 这两个精度。位数即 8 位、16 位、32 位和 64 位。定时器接口和组件的位数应为 32 位。本节主要介绍 32 位。

考虑到其他位数的平台，特定平台会使用其他位数。

准确度反映了组件与它提供的精度的相适应性。准确度受时钟漂移（相对晶振而言，内部时钟更受影响）和硬件受限影响。硬件限制的例子有：mica2 平台中 7.37MHz 的时钟不能提供准确的 binary 毫秒计数器，它能提供的最接近的是 7.37MHz/8。相对过剩精度，则应优先选择最接近所提供的时钟精度。

本节中所有接口带的参数都有精度，某些接口的参数还有时钟位数。这使具有不同精度或位数的定时器接口互不兼容。用户可根据给定的定时器接口，编写所需的精度和位数。

精度被表示为一种虚拟的类型：TMilli，T32khz，TMicro，它们通常在标准头文件 Timer.h 中定义：

```
typedef struct { int notUsed; } TMilli; // 1024 ticks per second
typedef struct { int notUsed; } T32khz; // 32768 ticks per second
typedef struct { int notUsed; } TMicro; // 1048576 ticks per second
```

注意：精度可根据需求命名，表示为频率或周期。

1. 定时器接口

定时器接口如下：

```
interface Counter<precision_tag, size_type>
interface Alarm<precision_tag, size_type>
interface BusyWait<precision_tag, size_type>
interface LocalTime<precision_tag>
interface Timer<precision_tag>
```

LocalTimer 接口和 Timer 接口使用固定的位数即 32 位，主要被用户的应用程序使用。Alarm 接口，BusyWait 接口和 Counter 接口主要被 TinyOS 定时器系统和高级用户组件使用。

1) Counter 接口

Counter 接口返回当时时间，并提供命令和一个事件函数来管理溢出。这些溢出命令和事件对从低位数计数器转变为高位数计数器是非常必要的。

```
interface Counter<precision_tag, size_type>
{
    async command size_type get();
    async command bool isOverflowPending();
    async command void clearOverflow();
    async event void overflow();
}
```

get() 返回当前时间。

isOverflowPending() 当本计数器的溢出标志位置位时返回 TRUE。例，当最外层的原子操作退出后会发生溢出事件，此时返回 TRUE。否则返回 FALSE。此命令仅返回溢出标志位的状态，此外并无其他作用。

clearOverflow()取消溢出事件，并清空溢出标志位。

overflow()通知现在发生溢出。也就是说，当前时间已经从它的最大值变为零（溢出发生状态）。

2) Alarm 接口

Alarm 组件是 Counter 的扩展，当 Alarm 组件的比较寄存器检测到定时器到时的时候，触发某事件。Alarm 接口中所有的命令和事件都是异步的。Alarm 接口为通常应用提供了基本命令集，为高级应用提供了扩展命令集。

```
interface Alarm<precision_tag,size_type>
{
    // basic interface
    async command void start( size_type dt );
    async command void stop();
    async event void fired();
    // extended interface
    async command bool isRunning();
    async command void startAt( size_type t0, size_type dt );
    async command size_type getNow();
    async command size_type getAlarm();
}
```

start(dt) 取消所有在此之前运行的警钟。设置从现在算起 dt 个时间单位的警钟。此警钟仅运行一次然后停止。

stop() 取消所有在此之前运行的警钟。

fired() 触发一个事件通知警报时间到。

isRunning() 当警钟已经开始运行，并且还没用被取消或者运行时间还未结束时返回 TRUE。否则返回 FALSE。

startAt(t0, dt) 取消所有在此之前运行的警钟，并且设置触发时间为 $t1=t0+dt$ 。即允许在调用 startAt 前有一个时延 t0。定时器子系统在内部使用这种方式，这可充分使用警钟位数，也可检测短警钟过早发生。

getNow() 返回当时时间，包含警钟精度和位数。

getAlarm() 返回现在正在运行的警钟的期满时间或者是先去运行警钟的期满时间。getAlarm 可于 startAt 结合来为警钟设置新的警钟时间。例如，startAt(getAlarm(),dt) 用于构建周期性警钟，这模式在 fire() 事件中使用。

3) BusyWait 接口

BusyWait 接口允许有一个非常短的同步延迟。为了确保警钟能合理并且高效使用，BusyWait 应保守使用。BusyWait 接口代替了 TinyOS1.x 中的 TOSH_uwait 宏。

BusyWait blocks 不少于某个特定的时间。延迟时间没有一个固定的上限。

```
interface BusyWait<precision_tag,size_type>
{
    async command void wait( size_type dt );
}
```

wait(dt)至少等待 dt 个时间单位。

4) LocalTime 接口

LocalTime 接口展现了一个无溢出的 32 位计数器。这主要是为不关心溢出条件的应用程序所设计的。

```
interface LocalTime<precision_tag>
{
    async command uint32_t get();
}
```

get()返回当前时间。

5) Timer 接口

Timer 接口中所有的命令和事件都是同步的。Timer 接口为基本应用提供了基本命令集，为高级应用提供了扩展命令集。Timer 接口支持周期性事件发生。

```
interface Timer<precision_tag>
{
    // basic interface
    command void startPeriodic( uint32_t dt );
    command void startOneShot( uint32_t dt );
    command void stop();
    event void fired();
    // extended interface
    command bool isRunning();
    command bool isOneShot();
    command void startPeriodicAt( uint32_t t0, uint32_t dt );
    command void startOneShotAt( uint32_t t0, uint32_t dt );
    command uint32_t getNow();
    command uint32_t gett0();
    command uint32_t getdt();
}
```

startPeriodic(dt)取消之前运行的定时器，并设置从调用时钟开始的 dt 时间单位后触发 fire。定时器每隔 dt 个时间单位周期性触发直到 stop()事件发生。

startOneShot(dt) 取消之前运行的定时器，并设置在调用时钟开始的 dt 时间单位后触发 fire。定时器触发一次后停止。

stop()取消之前所有运行的定时器。

fired()发出信号表示定时器期满（一次）或者重复触发（周期性）

isRunning() 对于运行一次定时器，如果定时器已经开始，但没有取消或时间还没满则返回 TURE。对于周期性运行定时器，isRunning()会返回 TURE，直到取消该周期性定时器。

isOneShot()如果定时器是一次运行定时器，则返回 TRUE。若定时器是一个周期性的定时器，则返回 FALSE。

startPeriodicAt(t0,dt)取消之前运行的定时器，并且设置期满时间为 $t1=t0+dt$ 。定时器每隔 dt 的时间单位，将周期性触发 fire()，直到停止。

`startOneShotAt(t0,dt)` 取消之前运行的定时器，并且设置期满时间为 $t1=t0+dt$ 。定时器将周期性触发一次 `fire()`，然后停止。

`getNow()` 返回当时时间，包含警钟的精度和位数。

`get0()` 返回时间锚，该时间锚针对先前启动的定时器或周期性定时器的前个事件时间。

`getdt()` 返回前个启动的定时器的延迟或周期。

7.5.2 定时器 HAL 方案

平台应该通过使用标准的 `Alarm` 接口和 `Counter` 接口来实现其时钟功能。在这一层的接口设计应该满足两种需求：首先应当独立于底层特定的定时器和报警器，从而对上层屏蔽硬件的差异；其次应该能够在定时器或报警器并不存在时报错，从而提高程序的健壮性。

平台在 `Counter` 和 `Alarm` 接口中指定硬件定时器参数 `precision ${P}` 和 `width ${W}`。

```
configuration Counter${P}${W}C
{
    provides interface Counter< T${P}, uint${W}_t >;
}
generic configuration Alarm${P}${W}C()
{
    provides interface Alarm< T${P}, uint${W}_t >;
}
```

实例 `Alarm${P}${W}C` 组件提供了新的独立的 `Alarm`。如果平台呈现出有限个 `Alarm` 资源，则在应用中分配的 `Alarm` 比在平台中呈现出的 `Alarm` 多时，将产生编译错误。

例如，如果某平台有一个 8 位的 32KHz 计数器和三个 8 位的 32KHz 的 `alarm`，则对 `${P}=32khz` 和 `${W}=16` 的 `Counter` 和 `Alarm` 接口是：

```
configuration Counter32khz8C
{
    provides interface Counter< T32khz, uint8_t >;
}
generic configuration Alarm32khz8C()
{
    provides interface Alarm< T32khz, uint8_t >;
}
```

这种结构用于定义平台组件，这些组件在单个应用中互不兼容。不兼容的组件在一起编译时将产生编译错误。

7.5.3 定时器 HIL 需求

所有的平台必须提供以下组件：

```
HilTimerMilliC
BusyWaitMicroC
```

这两个组件都使用 `binary` 单位，例，`HilTimerMilliC` 使用 `1/1024s`，`BusyWaitMicroC` 使用

1/1048576s。组件也可以使用其他精度。

1. HilTimerMilliC

```
configuration HilTimerMilliC
{
    provides interface Init;
    provides interface Timer<TMilli> as TimerMilli[ uint8_t num ];
    provides interface LocalTime<TMilli>;
}
```

分配一个新的定时器时，使用 `unique` (`UQ_TIMER_MILLI`) 来获取一个新的唯一的定时器数值。这个定时器数值用来索引 `TimerMilli` 的参数接口。`UQ_TIMER_MILLI` 在 `Timer.h` 中定义。`HilTimerMilliC` 被 `LocalTimeMilliC` 组件和 `TimerMilliC` 组件使用，位于 `tos/system` 目录下。

2. BusyWaitMicroC

```
configuration BusyWaitMicroC
{
    provides interface BusyWait<TMicro,uint16_t>;
}
```

`BusyWaitMicroC` 允许应用有数微秒的忙等待。当延迟很短时，`BusyWaitMicroC` 组件的使用是受限的。

7.5.4 用到的其他组件

许多独立于平台的通用组件用来帮助实现和提高 TinyOS 定时器系统。

- `AlarmToTimerC`
- `BusyWaitCounterC`
- `CounterToLocalTimeC`
- `TransformAlarmC`
- `TransformCounterC`
- `VirtualizeTimerC`

1. AlarmToTimerC 组件

`AlarmToTimerC` 组件将 32 位的 `Alarm` 转换成定时器。

```
generic component AlarmToTimerC( typedef precision_tag )
{
    provides interface Timer<precision_tag>;
    uses interface Alarm<precision_tag,uint32_t>;
}
```

2. BusyWaitCounterC 组件

`BusyWaitCounterC` 组件使用计数器来使程序等待一个特定的时间。

```
generic component BusyWaitC( typedef precision_tag, typedef size_type @integer() )
{
```

```

    provides interface BusyWait<precision_tag,size_type>;
    uses interface Counter<precision_tag,size_type>;
}

```

3. CounterToLocalTimeC 组件

CounterToLocalTimeC 组件将 32 为计数器转换成 LocalTime。

```

generic component CounterToLocalTimeC( precision_tag )
{
    provides interface LocalTime<precision_tag>;
    uses interface Counter<precision_tag,uint32_t>;
}

```

4. TransformAlarmC 组件

TransformAlarmC 组件减少 Alarm 的精度和（或）位数。

```

generic component TransformAlarmC(
    typedef to_precision_tag,
    typedef to_size_type @integer(),
    typedef from_precision_tag,
    typedef from_size_type @integer(),
    uint8_t bit_shift_right )
{
    provides interface Alarm<to_precision_tag,to_size_type> as Alarm;
    uses interface Counter<to_precision_tag,to_size_type> as Counter;
    uses interface Alarm<from_precision_tag,from_size_type> as AlarmFrom;
}

```

to_precision_tag 和 to_size_type 表示 Alarm 的最终精度和最终位数。from_precision_tag 和 from_size_type 表示 AlarmFrom 的精度和位数。bit_shift_right 表示 bit_shift 将使用过的精度转换为所提供的精度。

例，将 Alarm<T32khz,uint13_t> 转换为 Alarm<TMilli,uint32_t>, 则需建立 TransformAlarmC 如下：

```
new TransformAlarmC( TMilli, uint32_t, T32khz, uint16_t, 5 )
```

开发者使用 TransformAlarmC 来确保所有的 5 个参数都是自给的。当传递给 TransformAlarmC 的参数不一致时，不会发生编译错误。

5. TransformCounterC 组件

TransformCounterC 组件减少计数器的精度和（或）位数。

```

generic component TransformCounterC(
    typedef to_precision_tag,
    typedef to_size_type @integer(),
    typedef from_precision_tag,
    typedef from_size_type @integer(),
    uint8_t bit_shift_right,
    typedef upper_count_type @integer() )

```

```

{
    provides interface Counter<to_precision_tag,to_size_type> as Counter;
    uses interface Counter<from_precision_tag,from_size_type> as CounterFrom;
}

```

to_precision_tag 和 to_size_type 表示 Counter 的最终精度和最终位数。from_precision_tag 和 from_size_type 表示 CounterFrom 的精度和位数。bit_shift_right 表示 bit_shift 将使用过的精度转换为所提供的精度。upper_count_type 表示存数附加计数器位的数字类型。upper_count_type 位数必须大于等于附加位数即 to_size_type 加 bit_shift_right。

例如，将 Counter<T32khz,uint13_t>转换为 Counter<TMilli,uint32_t>，则需建立 TransformCounterC 如下：

```
new TransformCounterC( TMilli, uint32_t, T32khz, uint16_t, 5, uint32_t)
```

6. VirtualizeTimerC 组件

VirtualizeTimerC 组件使用单一的定时器来建立 255 个虚拟定时器。

```

generic component VirtualizeTimerC( typedef precision_tag, int max_timers )
{
    provides interface Init;
    provides interface Timer<precision_tag> as Timer[ uint8_t num ];
    uses interface Timer<precision_tag> as TimerFrom;
}

```

7.5.5 实现

HIL 接口定义位于 tinyos-2.x/tos/lib/timer 目录下：

- Alarm.nc
- BusyWait.nc
- Counter.nc
- LocalTime.nc
- Timer.h defines precision tags and strings for unique()
- Timer.nc

有效组件的实现位于 tinyos-2.x/tos/lib/timer 目录下：

- AlarmToTimerC.nc
- BusyWaitCounterC.nc
- CounterToLocalTimeC.nc
- TransformAlarmC.nc
- TransformCounterC.nc
- VirtualizeAlarmC.nc
- VirtualizeTimerC.nc

MSP430 定时器的实现位于 tinyos-2.x/tos/chips/msp430/timer 目录下：

- Alarm32khz16C.nc is generic and provides a new Alarm<T32khz,uint16_t>

- Alarm32khz32C.nc is generic and provides a new Alarm<T32khz,uint32_t>
- AlarmMilli16C.nc is generic and provides a new Alarm<TMilli,uint16_t>
- AlarmMilli32C.nc is generic and provides a new Alarm<TMilli,uint32_t>
- BusyWait32khzC.nc provides BusyWait<T32khz,uint16_t>
- BusyWaitMicroC.nc provides BusyWait<TMicro,uint16_t>
- Counter32khz16C.nc provides Counter<T32khz,uint16_t>
- Counter32khz32C.nc provides Counter<T32khz,uint32_t>
- CounterMilli16C.nc provides Counter<TMilli,uint16_t>
- CounterMilli32C.nc provides Counter<TMilli,uint32_t>
- GpioCaptureC.nc
- HilTimerMilliC.nc provides LocalTime<TMilli> and Timer<TMilli> as TimerMilli[uint8_t num]
- Msp430AlarmC.nc is generic and converts an MSP430 timer to a 16-bit Alarm
- Msp430Capture.nc HPL interface definition for MSP430 timer captures
- Msp430ClockC.nc exposes MSP430 hardware clock initialization
- Msp430ClockInit.nc HPL interface definition for hardware clock initialization
- Msp430ClockP.nc implements MSP430 hardware clock initialization and calibration and startup
- Msp430Compare.nc HPL interface definition for MSP430 timer compares
- Msp430Counter32khzC.nc provides Counter<T32khz,uint16_t> based onMSP430 TimerB
- Msp430CounterC.nc is generic and converts an Msp430Timer to a Counter
- Msp430CounterMicroC.nc provides Counter<TMicro,uint16_t> based onMSP430 TimerA
- Msp430Timer.h defines additional MSP430 timer bitmasks and structs
- Msp430Timer.nc HPL interface definition
- Msp430Timer32khzC.nc is generic and allocates a new 32khz hardware timer
- Msp430Timer32khzMapC.nc exposes the MSP430 hardware timers as a parameterized interface allocatable using Msp430Timer32khzC
- Msp430TimerC.nc exposes the MSP430 hardware timers
- Msp430TimerCapComP.nc is generic and implements the HPL forMSP430 capture/compare special function registers
- Msp430TimerCommonP.nc maps the MSP430 timer interrupts to Msp430TimerEvents
- Msp430TimerControl.nc HPL interface definition
- Msp430TimerEvent.nc HPL interface definition
- Msp430TimerP.nc is generic and implements the HPL for MSP430 timer special function registers

ATmega128 和 PXA27x 定时器的实现分别位于“tinyos-2.x/tos/chips/atm128/timer”和“tinyos-2.x/tos/chips/pxa27x/timer”目录下。

第 8 章 TinyOS 应用程序

运行过程剖析

本章详细介绍了 Blink 应用实例的运行过程，从系统初始化到具体的功能实现，对程序中每条语句的意义功能做了详细分析，有助于读者对 TinyOS 进行更深入的了解。

8.1 系统初始化

每个应用实例中都会有系统初始化，只是实例中不同的功能使得系统初始化也不同。系统初始化的功能主要是通过对硬件平台、软件代码进行基本配置操作，使程序可执行。系统初始化在模块文件中体现为 `event void Boot.booted()`，表示系统初始化完毕，触发启动事件，由此开始下面代码的执行。Boot 接口是由 MainC 组件提供，因此每个应用实例的顶层配置文件中都会包含 MainC 组件。

系统初始化分为三个阶段：

- 1) 任务调度器初始化。
- 2) 硬件平台和软件组件初始化。
- 3) 触发 Boot 接口中的事件。

系统初始化主要是由组件 MainC 来完成，在组件 MainC 中提供了接口 Boot，使用了接口 Init，用来初始化组件。以下是 MainC 文件的具体代码，可以发现 Boot 接口实际上是由组件 RealMainP 提供的。

```
#include "hardware.h"
configuration MainC {
    provides interface Boot;
    uses interface Init as SoftwareInit;
}
implementation {
    components PlatformC, RealMainP, TinySchedulerC;

    RealMainP.Scheduler -> TinySchedulerC;
    RealMainP.PlatformInit -> PlatformC;
    SoftwareInit = RealMainP.SoftwareInit;
    Boot = RealMainP;
}
```

系统初始化的具体实现是在其模块文件 RealMainP 中进行，在文件夹 C:\cygwin\opt\tinyos-

2.x\tos\system 中可以找到。

```

module RealMainP @safe() {
  provides interface Boot;
  uses interface Scheduler;
  uses interface Init as PlatformInit;
  uses interface Init as SoftwareInit;
}
implementation {
  int main() @C() @spontaneous() {
    atomic
    {
      platform_bootstrap();
      call Scheduler.init();
      call PlatformInit.init();
      while (call Scheduler.runNextTask());
      call SoftwareInit.init();
      while (call Scheduler.runNextTask());
    }
    __nesc_enable_interrupt();
    signal Boot.booted();
    call Scheduler.taskLoop();    //使调度程序进入无穷的循环，队列里
                                //没有任务时把 MCU 调至低功耗状态
    return -1;                  //这里不是 Void 函数，如果没有返回值，一些版本的编译器会警告
  }
  default command error_t PlatformInit.init() { return SUCCESS; }
  default command error_t SoftwareInit.init() { return SUCCESS; }
  default event void Boot.booted() { }
}

```

在系统初始化过程中涉及三个接口，分别是：init 接口用来初始化组件或者硬件状态；Scheduler 接口用来初始化任务调度器和执行任务；Boot 接口是系统完成初始化后程序运行的起始位置，每个应用实例中都会从这个接口的事件触发开始。

int main() @C() @spontaneous()的@C()表示 main()应该出现在全局域内，而不是出现在 RealMain 域内，main()就是一个全局函数。@spontaneous()表示 main()可以被其他文件访问，不仅限于本文件。

因为任务和硬件事件句柄可能被其他异步代码抢占，所以 nesC 程序易于受到特定竞争条件的影响，导致产生不一致或不正确的数据。避免竞争的办法通常是在任务内排他地访问共享数据，或访问所有数据都使用原子语句。这里用到了原子语句，原子语句用 atomic 作为关键字，表示在执行以下语句时不受硬件中断的影响，直至语句执行完毕。

```

atomic
{
  platform_bootstrap();
  call Scheduler.init();
  call PlatformInit.init();
}

```

```

while (call Scheduler.runNextTask());
call SoftwareInit.init();
while (call Scheduler.runNextTask());
}

```

在模块 RealMainP 中首先执行的是 platform_bootstrap()函数,使系统进入可执行状态,这个函数仅包括为以后编码所必需的操作,一般是空函数,也可对其编辑。然后是任务调度器初始化 call Scheduler.init(),使任务可投递、可执行。如果任务调度器初始化没有完成,随后的其他组件初始化时就不能提交任务,以至于程序不能正常运行。

8.1.1 任务调度器初始化

TinyOS 2.x 中,任务调度器是一个组件。调度组件必须协调不同的任务类型。基本类型的任务仍然是无参数、先入先出的。这种任务类型支持 TinyOS 1.x 中 task 和 post 的用法。调度器提供参数化的任务接口,每个与该接口连接的任务要使用函数 unique()来获得一个唯一的标识,用该标识来提交任务。TinyOS 调度组件位于 system\TinySchedulerC 文件中,默认的调度实现模块是 SchedulerBasicP。如果要采用新的调度算法,可以将配置文件 TinySchedulerC 放入应用中,且使用新的调度实现模块(比如 SchedulerEdfP)来替换 SchedulerBasicP。调度实现模块必须提供参数化接口 TaskBasic 和 Scheduler,接口 TaskBasic 用于支持 TinyOS 1.x 中 post 和 task 的用法。

```

configuration TinySchedulerC {
    provides interface Scheduler;
    provides interface TaskBasic[uint8_t id];
}
implementation {
    components SchedulerBasicP as Sched;
    components McuSleepC as Sleep;
    Scheduler = Sched;
    TaskBasic = Sched;
    Sched.McuSleep -> Sleep;
}
module SchedulerBasicP {
    provides interface Scheduler;
    provides interface TaskBasic[uint8_t id];
    uses interface McuSleep;
}
implementation
{
    ...
}

```

在 TinySchedulerC 中提供两个接口: Scheduler 和 TaskBasic。这两个接口实际上是 SchedulerBasicP 提供的。接口 TaskBasic 用于其他组件提交任务和任务调度器通知执行任务。Scheduler 接口用来初始化和运行任务。基本类型的任务队列是一个数组,存放任务标识

号（即参数化接口 TaskBasic 的下标）。这与 TinyOS 1.x 不同。在 TinyOS 1.x 中，任务队列中存放的是指向任务函数的指针。

```
interface Scheduler {
    command void init();
    command bool runNextTask();
    command void taskLoop();
}
```

1) Scheduler.init(): 该命令是系统初始化时执行的第一条语句，初始化任务队列结构和队列头尾指针。

```
command void Scheduler.init()
{
    atomic {
        memset( (void *)m_next, NO_TASK, sizeof(m_next) );
        m_head = NO_TASK;
        m_tail = NO_TASK;
    }
}
```

2) Scheduler.runNextTask(): 从任务队列中取出下一个将要执行任务的标识号，通过 TaskBasic.runTask 事件通知相应任务。

```
command bool Scheduler.runNextTask()
{
    uint8_t nextTask;
    atomic {
        nextTask = popTask();           //从任务队列中取下一个要执行的任务
        if( nextTask == NO_TASK )
            return FALSE;
    }
    signal TaskBasic.runTask[nextTask](); //参数是队列下标，即任务标识号
    return TRUE;
}
```

3) Scheduler.taskLoop(): 这是一个无限循环。该命令是 main()函数的最后一条语句，在系统初始化完成后执行。该语句进入任务调度无限循环，即从任务队列中调度任务，执行任务，再调度任务，如此循环下去。当任务队列中没有任务时，可以将 MCU 置为低功耗状态。

```
command void Scheduler.taskLoop()
{
    for (;;)
    {
        uint8_t nextTask;
        atomic
```

```

    {
        while ((nextTask = popTask()) == NO_TASK)
        {
            call McuSleep.sleep();
        }
        signal TaskBasic.runTask[nextTask]();
    }
}

```

如果替换调度来实现一个特殊的应用，则需要将配置文件 TinySchedulerC 放入该应用中。所有的调度实现模块都要像 SchedulerBasicP 模块一样提供一个参数化的接口 TaskBasic，这是为了支持 post 和 task 的用法。所有的调度实现都必须提供接口 Scheduler。例如，用调度模块 SchedulerEdfP 来替换模块 SchedulerBasicP，SchedulerEdfP 提供了三个接口：

```

module SchedulerEdfP {
    provides interface Scheduler;
    provides interface TaskBasic[uint8_t taskID];
    provides interface TaskEdf[uint8_t taskID];
}

```

其中，接口 TaskEdf 如下：

```

interface TaskEdf {
    async command error_t postTask(uint16_t deadlineMs);
    event void runTask();
}

```

配件 TinySchedulerC 为：

```

configuration TinySchedulerC {
    provides interface Scheduler;
    provides interface TaskBasic[uint8_t taskID];
    provides interface TaskEdf[uint8_t taskID];
}
implementation {
    components SchedulerEdfP;
    Scheduler = SchedulerEdf;
    TaskBasic = SchedulerEdfP;
    TaskEDF = SchedulerEdfP;
}

```

8.1.2 组件初始化

TinyOS 的启动顺序将组件的初始化分为硬件平台初始化 PlatformInit 和软件初始化 SoftWareInit，硬件平台的初始化在软件初始化之前。

硬件平台进行初始化的命令是 `call PlatformInit.init()`，选用的硬件平台不一样，其初始化内容也不相同，并且硬件平台的初始化要按照特定的顺序。不依赖于硬件资源的其他组件软件在初始化时都通过配置文件 `MainC` 连接到接口 `SoftwareInit`，在软件初始化时没有特定的顺序。

```
#include "hardware.h"
configuration MainC {
    provides interface Boot;
    uses interface Init as SoftwareInit;
}
implementation {
    components PlatformC, RealMainP, TinySchedulerC;
    RealMainP.Scheduler -> TinySchedulerC;
    RealMainP.PlatformInit -> PlatformC;
    // Export the SoftwareInit and Booted for applications
    SoftwareInit = RealMainP.SoftwareInit;
    Boot = RealMainP;
}
```

硬件平台初始化的具体初始化功能由 `PlatformC`（或其连接的组件）完成，每个硬件平台都会有组件 `PlatformC`，平台的不同使得其内容不同，但功能是一致的，都是对硬件平台完成初始化。初始化过程可能会提交任务，因此，随后要调度执行任务队列中的任务直至任务队列为空。硬件初始化提交的任务执行完毕后才能执行软件的初始化，在代码中显示为：

```
call PlatformInit.init();
while (call Scheduler.runNextTask());
call SoftwareInit.init();
while (call Scheduler.runNextTask());
```

用一个 `While` 循环来判断任务调度器队列中的任务是否已经执行完毕，如果队列中为空则跳出循环，执行接下来的代码。

当软件的初始化完毕后，说明整个系统已经准备完毕，进行中断初始化，使硬件中断可用，为了保证前期的初始化顺利进行，在这期间是不能执行硬件中断的。`__nesc_enable_interrupt()`的实现比较简单，是一段汇编语言，完成开启中断的功能，其代码如下：

```
Static__inline void __nesc_enable_interrupt(void){
    __asm volatile( " sei " );
}
```

8.1.3 触发 Boot 接口中的事件

进入第三个阶段，触发 `Boot` 接口中的事件。在组件 `RealMainP` 中提供了 `Boot` 接口，在程序的最后定义了 `Boot` 接口的事件，`default event void Boot.booted() { }`，在具体的应用实例中要对这个事件进行完善，执行系统初始化后立即触发该事件，比如在 `Blink` 应用实例中，初始化完毕后马上开始各个计时器的计时。

```

event void Boot.booted()
{
    call Timer0.startPeriodic( 250 );
    call Timer1.startPeriodic( 500 );
    call Timer2.startPeriodic( 1000 );
}

```

8.2 Blink 执行过程

以 apps\Blink 目录下的一个简单应用 Blink 为例, 说明 TinyOS 下的程序的执行过程。该应用的功能是周期性开关 LED 灯。该目录下共有四个文件, BlinkAppC.nc、BlinkC.nc、Makefile 和 Readme。其中文件 BlinkAppC.nc 中是顶层配置组件, 负责连接应用所依赖的其他组件。文件 BlinkC.nc 中是模块组件, 是该应用功能的具体实现。典型的 TinyOS 应用程序都有一个 Makefile 文件, 文件中可以设置硬件平台选项、顶层配件名称等 nesC 编译器 (ncc) 编译选项。

8.2.1 配置文件

代码可参考例 3.19:

```

//应用配置文件 BlinkAppC.nc
configuration BlinkAppC
{
    //顶层配置文件, 不提供和使用任何接口
}
implementation
{
    //整个应用实例中所涉及的组件
    components MainC, BlinkC, LedsC;
    components new TimerMilliC() as Timer0;
    components new TimerMilliC() as Timer1;
    components new TimerMilliC() as Timer2;
    //组件间的接口连接
    BlinkC -> MainC.Boot;

    BlinkC.Timer0 -> Timer0; //组件 BlinkC 的 Timer<TMilli>接口和组件 TimerMilliC()连接,表示
    组件 TimerMilliC()实现了组件 BlinkC 要用到的 Timer<TMilli>接口
    BlinkC.Timer1 -> Timer1;
    BlinkC.Timer2 -> Timer2;
    BlinkC.Leds -> LedsC;
}

```

在编译期间, 为了创建把组件连接在一起的逻辑关系, 会预先处理配置文件, 这是由 nesC 编译器自动完成的。nesC 编译器的输出是一个标准 C 文件, 包含应用程序中的所有组件, 也包含所有必需的连接信息。

配件 BlinkAppC 是顶层配件, 不提供和使用任何接口。当配件不是顶层配件时, 在关键字 configuration 后面的花括号之间可以加入 uses 和 provides 语句来使用和提供接口。配件

实现是由关键字 `implementation` 中的语句完成的。用关键字 `components` 声明了配件所用到的组件。这里是 `MainC`、`BlinkC`、`LedsC` 以及 `Timer0`、`Timer1`、`Timer2`。在 `nesC` 中可以使用 `as` 来为某种类型的组件或接口的实例命名。这里 `Timer0`、`Timer1`、`Timer2` 分别是 `TimerMilliC` 组件的一个实例的名字。把模块 `BlinkC` 使用的接口 `Boot`、`Timer0` 和 `Leds` 分别与其他组件提供的相应接口连接起来。

在上一节系统的初始化中已经就组件 `MainC` 进行了详细介绍，在此主要对组件 `LedsC`、`TimerMilliC` 进行分析。在配置文件 `BlinkAppC` 中可以看到“`BlinkC.Leds -> LedsC`”，说明模块文件中要使用到 `Leds` 接口，并且这个接口是由组件 `LedsC` 来实现的。

```
configuration LedsC {
    provides interface Leds;
}
implementation {
    components LedsP, PlatformLedsC;

    Leds = LedsP;

    LedsP.Init <- PlatformLedsC.Init;
    LedsP.Led0 -> PlatformLedsC.Led0;
    LedsP.Led1 -> PlatformLedsC.Led1;
    LedsP.Led2 -> PlatformLedsC.Led2;
}
```

在组件 `LedsC` 中提供了接口 `Leds`，在 `implementation` 中提到“`Leds = LedsP`”，这就指出了接口 `Leds` 是由组件 `LedsP` 真正提供和定义的。下面来看模块文件 `LedsP`。

```
{
    .....
    async command void Leds.led0On() {
        call Led0.clr();
    }
    async command void Leds.led0Off() {
        call Led0.set();
    }
    async command void Leds.led0Toggle() {
        call Led0.toggle();
    }
    async command void Leds.led1On() {
        call Led1.clr();
    }
    async command void Leds.led1Off() {
        call Led1.set();
    }
    async command void Leds.led1Toggle() {
        call Led1.toggle();
    }
    async command void Leds.led2On() {
```

```

        call Led2.clr();
    }
    async command void Leds.led2Off() {
        call Led2.set();
    }
    async command void Leds.led2Toggle() {
        call Led2.toggle();
    }
    .....
}

```

在模块文件 LedsP 中提供了接口 Leds，定义了一组命令 `async command void Leds.led0On()`、`led0Off()`、`led0Toggle()`等，这些命令控制着硬件 Led 灯的开、关、置换等功能。请注意在这里使用了关键字 `async`，表示所声明的命令和事件为异步代码。异步代码是可以对硬件中断予以及时响应的代码。下面以 `led0` 的开关为例说明。

```

    async command void Leds.led0On() {
        call Led0.clr();
    }
    async command void Leds.led0Off() {
        call Led0.set();
    }
}

```

两个命令的不同在于 `call Led0.clr()`和 `call Led0.set()`分别调用了接口 `Led0` 中的两个函数，在 `configuration LedsC` 中看到接口 `Led0` 由组件 `PlatformLedsC` 提供，`LedsP.Led0 -> PlatformLedsC.Led0`。不同的硬件平台有不同的 `PlatformLedsC`，在这里以 `Mica` 节点为例进行说明。

```

configuration PlatformLedsC
{
    provides interface GeneralIO as Led0;
    provides interface GeneralIO as Led1;
    provides interface GeneralIO as Led2;
    uses interface Init;
}
implementation
{
    components HplAtm128GeneralIOC as IO;
    components PlatformP;

    Init = PlatformP.MoteInit;

    Led0 = IO.PortA2; // Pin A2 = Red LED
    Led1 = IO.PortA1; // Pin A1 = Green LED
    Led2 = IO.PortA0; // Pin A0 = Yellow LED
}

```

从 PlatformLedsC 可以知道, Led0、Led1、Led2 是 GeneralIO 的不同实例, 并且它们分别对应着 MCU 芯片的不同引脚, 表示为红灯、绿灯和黄灯。因此可以理解 call Led0.clr()是将 PinA2 引脚电平置低, 使红灯亮, 而 call Led0.set()就是使红灯灭。

8.2.2 模块文件

应用模块文件 BlinkC.nc 具体的功能主要是实现 Led 灯周期性开关, 包括两个部分: 声明模块文件中使用到的接口和提供的接口、功能的实现和执行过程。

```
#include "Timer.h"
module BlinkC @safe()
{
    uses interface Timer<TMilli> as Timer0; //声明所使用的接口, 也可以提供接口
    uses interface Timer<TMilli> as Timer1;
    uses interface Timer<TMilli> as Timer2;
    uses interface Leds;
    uses interface Boot;
}
implementation
{
    event void Boot.booted() //系统初始化, 完成后触发事件
    {
        call Timer0.startPeriodic( 250 ); //Timer0 计时每隔 250ms 触发 Timer0 中事件
        call Timer1.startPeriodic( 500 );
        call Timer2.startPeriodic( 1000 );
    }
    /**通过配置文件可以找到接口 Timer<TMilli>最终在 tos\lib\timer\VirtualizeTimerC.nc 文件中定义, 里面给出了 command void Timer.startPeriodic[uint8_t num](uint32_t dt)以及 default event void Timer.fired[uint8_t num]() **/
    event void Timer0.fired()
    {
        call Leds.led0Toggle();
    }
    event void Timer1.fired()
    {
        call Leds.led1Toggle();
    }
    event void Timer2.fired()
    {
        call Leds.led2Toggle();
    }
}
```

在模块 BlinkC 的规范部分声明了使用的接口, 分别为 Timer0、Timer1、Timer2 以及 Leds 和 Boot。其中 Timer0、Timer1、Timer2 是接口类型为 Timer<TMilli>的实例, 通过 as 命名。值得注意的是, 虽然这里的 Timer0 和前面配件 BlinkAppC 中的 Timer0 名字相同, 但

是代表的意义却不同。配件中声明的 `Timer0` 代表的是类型为 `TimerMilliC()` 的组件实例。在组件 `TimerMilliC()` 中提供了接口 `Timer<TMilli>`，通过配置文件查看组件之间的连接，接口 `Timer<TMilli>` 最终是由 generic module `VirtualizeTimerC` 实现和定义的。

结合前面配件中描述的连接情况，可以在模块 `BlinkC` 中调用这些接口中的任何命令（具体实现由提供接口的组件完成，比如 `LedsC`），同时必须实现这些接口中的所有事件。

8.2.3 Blink 应用程序的图形化说明

对 `Blink` 应用程序，在 `Cygwin` 中将目录切换到 `opt\tinyos-2.x\apps\Blink` 下，输入 `make micaz docs` 命令，生成的文档说明位于 `opt\tinyos-2.x/doc/nedoc/micaz/chtml` 目录下。打开 `BlinkAppC`，可看到如图 8-1 所示的说明。

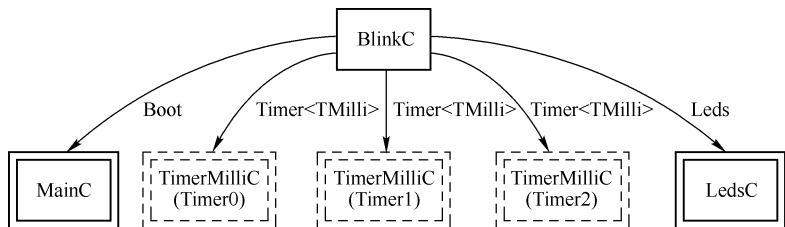


图 8-1 Blink 应用程序的说明

在 `nedsoc` 图表中，单一矩形的 `BlinkC` 表示模块；双层矩形框的 `MainC` 和 `LedsC` 表示配件；虚拟边界框的 `TimerMilliC` 表示组件是通用的。`BlinkC` 模块使用了 `MainC` 配件中的 `Boot` 接口、`LedsC` 配件中的 `Leds` 接口、`TimerMilliC` 中的 `Timer<TMilli>` 接口。

8.2.4 程序运行过程

系统初始化完毕后，模块 `BlinkC` 收到 `booted()` 事件，其处理程序为启动周期性模式的计时器。计时器组件使用 MCU 上的硬件计时器来计时。接口 `Boot` 中只有一个函数，即事件 `booted()`。该事件到达时，表明系统已经启动（即所有组件已经初始化完毕）。模块 `BlinkC` 对该事件的处理程序为调用计时器接口 `Timer0` 的 `startPeriodic(1000)` 命令，该命令表明设置并启动一个周期为 1s 的计时器。

当时间到达时，计时器组件通知事件 `Timer0.fired()`。在模块 `BlinkC` 的 `Timer0.fired()` 事件的实现代码中仅使用了一条语句：`post toggle()`，即提交一个任务 `toggle()`，当任务调度器调度该任务执行时，该任务触发 LED（开关转换）。

```
event void Timer0.fired()
{
    call Leds.led0Toggle();
}
```

在 `TinyOS` 中，当一个组件用关键词 “`task`” 声明一个任务时，实际上就表示它使用了接口 `TaskBasic` 的一个实例。当组件使用关键词 “`post`” 时，实际上是调用了 `TaskBasic.postTask()` 命令。在 `Scheduler.taskLoop()` 中会调度到该任务，并通过 `runTask()` 事件通知任务执行，即切换 `led0` 开关。

由于计时器组件启动的计时器是周期性的，因此，时间间隔到达后会继续重复上述工作。事实上，在整个应用的过程中，还有其他任务，比如计时器组件在计时到达时是在任务中向上层通知事件的。

Leds 接口定义了许多命令，诸如 led0On()、led0Off()、led0Toggle()等，控制节点上的 LED0 开、关、置换。这些命令由提供 Leds 接口的 LedsC（实际上是 LedsP）实现。

8.3 Blink 应用程序中 nesC 到 C 的映射

对于本例的 Blink 应用程序，编译时，会在 opt\tinyos-2.x\apps\Blink\build\micaz 目录下生成一个 app.c 文件，也就是 ncc 把 nesC 预编译成的 C 文件。在 app.c 中可看到如下映射。

```
static inline void BlinkC$Boot$booted(void)
{
    BlinkC$Timer0$startPeriodic(250);
    BlinkC$Timer1$startPeriodic(500);
    BlinkC$Timer2$startPeriodic(1000);
}
static inline void BlinkC$Timer0$fired(void)
{
    static void BlinkC$Leds$led0Toggle(void);
}
static inline void BlinkC$Timer1$fired(void)
{
    static void BlinkC$Leds$led1Toggle(void);
}
static inline void BlinkC$Timer2$fired(void)
{
    static void BlinkC$Leds$led2Toggle(void);
}
```

8.4 仿真 Blink 应用程序

对本例进行仿真时，首先进入目录：\cygwin\opt\tinyos-2.x\apps\Blink。然后编写“C:\cygwin\opt\tinyos-2.x\apps\Blink\Blink.py”的内容如下。

```
#!/usr/bin/python
from TOSSIM import *
import sys
t=Tossim([]);
t.addChannel("BlinkC",sys.stdout)
t.getNode(1).bootAtTime(10000);
for i in range (0,100):
    t.runNextEvent()
```

最后在命令行输入命令： `$ make micaz sim`，显示模拟结果。

```
***Successfully built micaz TOSSIM library.
```

执行命令： `$ python Blink.py`（也可以用命令“`./Blink.py`”），显示仿真结果。

如图 8-2 所示，Led0、Led1 和 Led2 分别以 250ms、500ms 和 1000ms 的周期来转换亮灭状态。

```
Administrator@23zhangyanjun1 /opt/tinyos-2.x/apps/Blink
$ ./Blink.py
DEBUG (1): Timer 0 fired @ 0:0:0.244141635.
DEBUG (1): Timer 0 fired @ 0:0:0.488282260.
DEBUG (1): Timer 1 fired @ 0:0:0.488282270.
DEBUG (1): Timer 0 fired @ 0:0:0.732422885.
DEBUG (1): Timer 0 fired @ 0:0:0.976563510.
DEBUG (1): Timer 1 fired @ 0:0:0.976563520.
DEBUG (1): Timer 2 fired @ 0:0:0.976563530.
DEBUG (1): Timer 0 fired @ 0:0:1.220704135.
DEBUG (1): Timer 0 fired @ 0:0:1.464844760.
DEBUG (1): Timer 1 fired @ 0:0:1.464844770.
DEBUG (1): Timer 0 fired @ 0:0:1.708985385.
DEBUG (1): Timer 0 fired @ 0:0:1.953126010.
DEBUG (1): Timer 1 fired @ 0:0:1.953126020.
DEBUG (1): Timer 2 fired @ 0:0:1.953126030.
DEBUG (1): Timer 0 fired @ 0:0:2.197266635.
DEBUG (1): Timer 0 fired @ 0:0:2.441407260.
DEBUG (1): Timer 1 fired @ 0:0:2.441407270.
```

图 8-2 仿真 Blink 应用程序

第9章 仿真、调试与编程提示

与所有嵌入式开发一样，TinyOS 也离不开仿真工具，所谓仿真就是通过软件模拟传感器节点硬件，从而在不需要进行程序实际下载的情况下观察我们编写的应用程序的运行情况。本章将对 TinyOS 自带的仿真工具 TOSSIM 进行介绍。此外，由于 TinyOS 本身没有提供调试工具，本章将针对一种第三方的调试工具如何在 TinyOS 中使用进行介绍。最后针对 TinyOS 编程中容易出现的一些问题进行一些提示。本章内容部分来自对国际互联网公开内容的收集和整理，在此对所有免费公开内容提供者表示崇高的敬意。

9.1 TinyOS 自带仿真工具——TOSSIM

9.1.1 概述

TOSSIM (TinyOS simulator) 是 TinyOS 自带的一个仿真工具，可以支持大规模的网络仿真。由于 TOSSIM 运行和传感器硬件相同的代码，所以仿真编译器能直接从 TinyOS 应用的组件表编译仿真程序。通过替换 TinyOS 下层部分硬件相关的组件，TOSSIM 把硬件中断换成离散仿真事件，由仿真器事件抛出的中断来驱动上层应用，其他的 TinyOS 组件尤其是上层的应用组件都无需更改。

1. 编译器的支持

TOSSIM 改进了 nesC 编译器，通过使用不同的选项，用户可以把在硬件节点上运行的代码编译成仿真程序。

2. 执行模型

TOSSIM 的核心是一个仿真事件队列。与 TinyOS 不同的是，硬件中断被模拟成仿真事件插入队列，仿真事件调用中断处理程序，中断处理程序又可以调用 TinyOS 的命令或触发 TinyOS 的事件，这些 TinyOS 的事件和命令处理程序又可以生成新的任务，并将新的仿真事件插入队列，重复此过程直到仿真结束。

3. 硬件模型

TinyOS 把节点的硬件资源抽象成组件。通过将硬件中断转换成离散仿真事件，替换硬件资源组件，TOSSIM 模仿了硬件资源组件行为，为上层提供了与硬件相同的标准接口。硬件模拟为仿真物理环境提供了接入点，通过修改硬件模拟组件，可以为用户提供各种性能的硬件环境，满足不同用户的需求。

4. 无线模型

TOSSIM 允许用户选择不同精确度和复杂度的无线模型，该模型独立于仿真器之外，从而保证了仿真器的简单和高效。用户可以通过一个有向图指定不同节点对之间通信的误码率，表示在该链路上发送一个比特数据时可能出错的概率。对同一个节点来说，双向误码率

是独立的，可以模拟不对称链路。

5. 仿真监控

用户可以自行开发应用软件来监控 TOSSIM 仿真的执行过程，二者通过 TCP/IP 通信。TOSSIM 为监控软件提供实时仿真数据，包括 TinyOS 源代码加入的 Debug 信息、各种数据包和传感器的采样值等，监控程序可以根据这些数据显示仿真的执行情况。同时允许监控程序以命令调用的方式更改仿真程序的内部状态来控制仿真程序。TinyViz (TinyOS Visualizer) 是 TinyOS 提供的一个仿真监控程序。

9.1.2 编译 TOSSIM

TOSSIM 包含在 TinyOS 的库文件中，其核心代码位于 `tos/lib/tossim`，所有 TinyOS 的源文件都包含一个可供选择的子目录，这个目录中包含仿真实现的开发包。例如 `tos/chips/atm128/timer/sim` 就包含着 Atmega128 的定时器抽象的 TOSSIM 实现代码。

在命令窗中输入如下命令来对 TOSSIM 进行编译。

```
$ cd apps/Blink
$ make micaz sim
```

目前，Micaz 是 TOSSIM 唯一支持的平台，执行上述命令会看到如下的显示结果：

```
mkdir -p build/micaz
placing object files in build/micaz
writing XML schema to app.xml
compiling BlinkAppC to object file sim.o
ncc -c -fPIC -o build/micaz/sim.o -g -O0 -tossim -fnesc-nido-tosnodes=1000
-fnesc-simulate -fnesc-nido-motenummer=sim_node\(\) -finline-limit=100000 -Wall
-Wshadow -DDEF_TOS_AM_GROUP=0x7d -Wnesc-all -target=micaz -fnesc-cfile=build/micaz/app.c
-board=micasb -Wno-nesc-data-race BlinkAppC.nc -fnesc-dump=components -fnesc-dump=variables
-fnesc-dump=constants -fnesc-dump=typedefs -fnesc-dump=interfacedefs -fnesc-dump=tags -fnesc-
dumpfile=app.xml
compiling Python support into pytosim.o and tossim.o
g++ -c -shared -fPIC -o build/micaz/pytosim.o -g -O0 /home/pal/src/tinyos-2.x/tos/lib/tossim/tossim_
wrap.cxx
-I/usr/include/python2.3 -I/home/pal/src/tinyos-2.x/tos/lib/tossim -DHAVE_CONFIG_H
g++ -c -shared -fPIC -o build/micaz/tossim.o -g -O0 /home/pal/src/tinyos-2.x/tos/lib/tossim/tossim.c
-I/usr/include/python2.3 -I/home/pal/src/tinyos-2.x/tos/lib/tossim
linking into shared object ./_TOSSIMmodule.so
g++ -shared build/micaz/pytosim.o build/micaz/sim.o build/micaz/tossim.o -lstdc++ -o
_TOSSIMmodule.so
copying Python script interface TOSSIM.py from lib/tossim to local directory
```

9.1.3 在 Python 下运行 TOSSIM

进入 RadioCountToLeds application 的目录下编译 TOSSIM。

```
$ cd tinyos-2.x/apps/RadioCountToLeds
$ make micaz sim
```

在交互模式下启动 Python。键入如下命令：

```
$ python
```

将看到如下内容：

```
Python 2.3.4 (#1, Nov  4 2004, 14:13:38)
[GCC 3.4.2 20041017 (Red Hat 3.4.2-6.fc3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
```

根据读者安装的情况不同，版本号会有所区别。

第一件事是导入 TOSSIM 并创建一个 TOSSIM 对象，键入：

```
>>> from TOSSIM import *
>>> t = Tossim([])
```

方括号内是可选参数，可以在仿真时对变量进行跟踪，为空时表示无需跟踪变量。TOSSIM 的运行是通过使用 `runNextEvent` 来实现的，例如：

```
>>> t.runNextEvent()
0
```

当 TOSSIM 被告知运行下一事件时，若返回值是 0，表示没有事件可以运行。没有事件的原因是没有启动任何的节点，下面这段代码将会使 32 号节点在 45654 时间（仿真定时器时间）启动并运行。

```
>>> m = t.getNode(32)
>>> m.bootAtTime(45654)
>>> t.runNextEvent()
1
```

此时，`runNextEvent` 返回值为 1，但是我们并不知道节点是否如预期那样启动，我们可以通过以下方法来获知这一事件。

```
>>> m.isOn()
1
>>> m.turnOff()
>>> m.isOn()
0
>>> m.bootAtTime(560000)
>>> t.runNextEvent()
0
>>> t.runNextEvent()
1
```

注意这里第一个 `runNextEvent` 返回值为 0，这是因为当我们关掉节点时，队列里还有事件没有被执行。当节点被关闭时事件被挂起，`runNextEvent` 返回 0。第二次 `runNextEvent` 返回值为 1 是因为我们在仿真时间 560000 时启动了节点。

如果显示节点依然没有开启，以下代码会一直执行直到节点启动为止。

```
>>> while not m.isOn():
>>> t.runNextEvent()
```

Tossim 对象具有一些非常实用的功能，包括如下几项：

- `currentNode()`: 返回当前节点的 ID。
- `getNode(id)`: returns an object representing a specific mote。
- `runNextEvent()`: 启动一个仿真事件。
- `time()`: 以长整型返回当前仿真时间。
- `timeStr()`: 以 string 型返回当前仿真时间。
- `init()`: 初始化 TOSSIM。
- `mac()`: return the object representing the media access layer。
- `radio()`: return the object representing the radio model。
- `addChannel(ch, output)`: add output as an output to channel ch。
- `removeChannel(ch, output)`: remove output as an output to channel ch。
- `ticksPerSecond()`: return how many simulation ticks there are in a simulated second。

9.2 GDB/ddd 调试

由于 TinyOS 并不自带调试工具，因此在 TinyOS 下进行程序开发时需要借助第三方的调试工具进行程序调试。GDB 是 GNU 开源组织发布的一个强大的 UNIX 下的程序调试工具。与 VC、BCB 等 IDE 的调试方式不同，GDB 虽然没有图形化的调试界面，却有着更强大的功能。

Ddd (Data Display Debugger) 是命令行调试程序，如 GDB、DBX、WDB、Ladebug、JDB、XDB、Perl Debugger 或 Python Debugger 的可视化图形前端。它特有的图形数据显示功能 (Graphical Data Display) 可以把数据结构按照图形的方式显示出来。接下来，将具体讲解如何结合命令行调试工具 GDB 和可视化前端 ddd 这两个工具来进行 TinyOS 程序的调试工作。

(1) nesC 名称到 C 名称的映射关系

ncc 将 nesC 代码转换为 C 代码后，nesC 中的类型、变量和函数名称是不会发生变化的，除非和 nesC 的关键字相同。在这种情况下需要在名字前面加上前缀 “_nesc_keyword_”，比如，组件 AA 就变为 “_nesc_keyword_AA”。nesC 代码与 C 代码的具体映射关系如下：

模块 M 中的变量 X 映射为 M\$X，变量的名字不发生改变；

模块 M 中的函数 F 映射为 M\$F；

模块 M 中的命令或者事件 C 映射为 M\$C；

模块 M 中的接口 I 的命令或者事件 C 被映射为 M\$I\$C。

(2) 用 gdb/ddd 调试 nesC 代码

`gdb` 现在还没有特定的 nesC 模式。但是，在 `gdb` 中可以有效地调试通过 `ncc` 编译器产生的 C 代码，C 代码中包含 `#line`，所以对 nesC 代码的单步跟踪可以显示正确的 nesC 源代码，可以根据 nesC 组件的名称和行号设置断点。但是变量、函数、命令和事件的名称就不是这么简单了。如果你想要引用他们中的任何一个就必须使用他们在产生的 C 代码中的名称，在调试中读者要注意这一点。

`gdb` 是命令行调试工具，要输入的字符比较多，调试起来比较麻烦。为了方便，可以采

用 ddd 辅助调试。它是命令行调试工具的图形化前端，能够提供更友好的用户功能和界面，而且更容易设定调试会话。

我们在 Linux 环境下安装了 TinyOS。使用 gdb/ddd 调试 BlinkTask 应用的步骤如下。

1) 编译欲调试的 TinyOS 应用程序，把将目标映像下载到节点上。

```
$ cd apps/tutorials/BlinkTask
$ make telosb install telosb debug
```

注意，调试时上面的命令中要加 debug 参数。

输出信息：

```
mkdir -p build/telosb
  compiling BlinkTaskAppC to a telosb binary
ncc -o build/telosb/main.exe -O1 -g -fnesc-no-inline -mdisable-hwmul -Wall
-Wshadow -DDEF_TOS_AM_GROUP=0x7d -Wnesc-all -target=telosb
-fnesc-cfile=build/telosb/app.c -board=    BlinkAppC.nc -lm
  compiled BlinkTaskAppC to build/telosb/main.exe
      4496 bytes in ROM
      61 bytes in RAM
msp430-objcopy --output-target=ihex build/telosb/main.exe
build/telosb/main.ihex    writing TOS image
cp build/telosb/main.ihex build/telosb/main.ihex.out
installing telosb binary using the parallel port jtag adapter
msp430-jtag -Iepr build/telosb/main.ihex.out
MSP430 parallel JTAG programmer Version: 2.0
Mass Erase...
Program...
4528 bytes programmed.
Reset device...
Reset and release device...
```

2) 从互联网上下载 msp430-gdbproxy。它是 JTAG 和 gdb 之间连接的桥梁。在 msp430-gdbproxy 文件存放的路径下输入：

```
$ ./msp430-gdbproxy --port=2000 msp430
```

输出信息：

```
Remote proxy for GDB,  v0.7.1,  Copyright (C) 1999 Quality Quorum Inc.
MSP430 adaption Copyright (C) 2002 Chris Liechti and Steve Underwood

GDBproxy comes with ABSOLUTELY NO WARRANTY; for details
use '--warranty' option. This is Open Source software. You are
welcome to redistribute it under certain conditions. Use the
'--copying' option for details.

debug: MSP430_Initialize()
About to init '/dev/parport0'
debug: MSP430_VCC(3000)
```

```
debug: MSP430_Identify()
info: msp430: Target device is a 'MSP430F1611' (type 42)
debug: MSP430_Configure()
notice: msp430-gdbproxy: waiting on TCP port 2000
```

3) 建立新的终端, 在应用程序二进制文件所在的目录下 (apps/Tutorials/BlinkTask/build/telosb) 输入:

```
$ ddd --debugger msp430-gdb main.exe
```

该命令启动 ddd, 使用 msp430-gdb 作为调试软件。

ddd 由三个主要的窗口构成。

- 数据窗口: 显示被调试程序的当前数据。
- 源代码窗口: 显示调试程序的源代码。
- 调试控制台: 输入调试命令, 显示调试信息。

除了三个主要的窗口外, 还有其他一些可选的窗口。

- 命令工具: 提供了经常使用的命令。
- 机器码窗口: 显示当前的机器码, 通常在源代码窗口之下。
- 执行窗口: 显示调试程序的输入和输出。

启动 ddd 后, 通过菜单栏的 file→open source, 打开 BlinkTaskC 文件, BlinkTaskC 的源代码会显示在源代码窗口中。要在 Timer0.fired 处设置断点, 只需要把鼠标移到源代码窗口中 Timer0.fired 所在的行, 单击右键选择 set breakpoint, 就会在对应的行看到一个红色的 stop 标志, 表示断点设置成功。要使程序运行到断点处, 单击浮动命令工具的 cont 按键, 经过一小段时间, 在断点所在行出现一个小箭头, 表示程序已运行到断点处。在工具栏中输入 BlinkTaskC\$state, 单击 Display 就会在数据窗口中显示变量 state 的值。如图 9-1 所示。

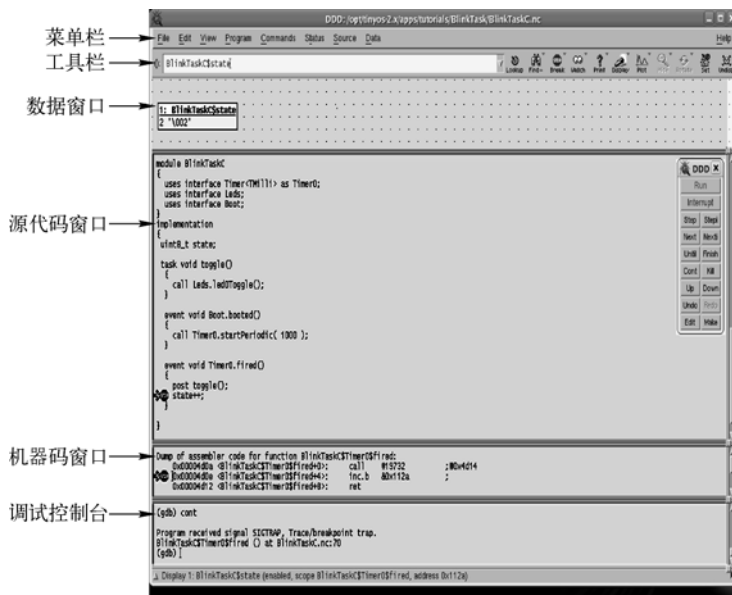


图 9-1 ddd 调试截图

(4) 仿真环境下调试

TOSSIM 是 TinyOS 传感器网络的离散事件仿真器，它为用户提供了一个可控制和可重复的调试、检验和分析代码的仿真环境。TOSSIM 在 PC 环境上运行，用户可以通过 gdb 和其他的相关工具调试 TinyOS 代码。

TinyViz 是 TOSSIM 的图形用户调试接口，它能够可视化地与 TinyOS 应用程序进行交互，更方便地跟踪 TinyOS 应用的执行。有关 TOSSIM 和 TinyViz，有兴趣的读者可以阅读参考文献。

9.3 编程提示

本节的目的在于使读者在进行 TinyOS 程序开发时，尽可能避免因冲突导致编译错误，TinyOS 中最重要的地方是接口和组件名字的冲突。为了使阅读者和编程者能根据名字区分组别以及定义在哪个包中，请记住阅读有用代码的时间远比写它要花的时间多，如果你添加了新的约定，应在自述文件中注明。

一般性建议：

- General 不要使用不常见的简写和缩写。
- 简称的首字母写应该大写。
- 缩写要保持一致性。
- 所有的代码应该有 nesdoc /Doxygen /Javadoc 说明文件，并在代码文件的第一个文档块添加@author 标签。

9.3.1 Packages 包

包是一个相关源文件和其他文件的集合。包是一个逻辑上的分组。其在物理上有可能但不一定是一个单独的文件夹。TinyOS 中包通常是一个包含零个或者多个子目录的目录。nesC 和 C 目前不提供任何包支持，所以在不同的包中类型和组件的名字有可能发生冲突。为了尽可能减少这种冲突，在相关的文件使用组前缀时一种明智的方法在一个包中区分公共组件（可以在包外部被使用和连接）和私有组件（只能在包内部使用和连接）。这个区分在 nesC 中不是强制的。

9.3.2 语法约定

1. nesC 约定

所有的 nesC 文件名都必须以.nc 扩展名结束，nesC 编译器要求文件名和接口或组件名匹配。目录名使用小写字母。接口或者组件名以大写字母开头的混合形式。所有的组件都带有扩展名“C”。所有的私有组件必须带有扩展名“P”。接口不能以“C”或“P”结尾。如果接口和组件相关，推荐使用除了扩展名其他都相同的命名。命令、事件、任务和函数应该使用小写字符开头的混合形式。事件处理命令名称应该和产生该事件的起因命令相关。推荐命名为 xxxDone。常量应该全部使用大写形式，词语间使用“_”（下划线连接符）连接。推荐使用枚举类型定义整型常量，而不是使用#define 方式。通用组件的参数类型同 C 类型，全部使用小写字符，以“_t”结束。模块变量应该使用以小写字符开始的混合形式。

每一个包名都使用其组件接口和全局 C 名称前缀。这个前缀可能在多个包中保持一致。比如所有的 HPL 命名都以 Hpl 开头（这是一个公用前缀示例）。具体芯片 HAL 组件和接口命名以芯片名字开头，如 Atm128 代表 ATmega128。所有 Mate 虚拟机相关的文件都使用 Mate 前缀。

TinyOS 的核心（例如 timer 组件、Init 接口）命名不使用前缀。某些包可能使用多个前缀，比如 ATmega128 芯片（使用 Atm128 前缀）包中的硬件表象层组件使用 Hpl 前缀。

2. 预处理

不要使用 nesC 风格的 include 语句。它不能正确地处理宏，应使用#include 代替。nc 文件中的宏定义#define 必须用在模块或者配置关键字之后，以便明确限定其使用范围。被多个.nc 文件使用的宏定义应该在头文件中定义，并使用#include 包含进去。尽量少使用宏，只有在不能使用 enum 和 inline 的情况下才使用#define。

unique()参数应该使用#define 字符串常量，这样可以减少编译器无法捕获的编译错误。所有的 C 文件都有一个.h（头文件）或者一个.c（源文件）扩展名。和某个相关组件的文件名也应该保持一致。包中的文件名应该包含其包名前缀（如果有）。不属于任何组件的文件名使用小写形式。

C 不提供任何形式的名称保护，如果包使用前缀，那么所有的类型、标识符、函数、变量、常量和宏都应该保持一致，这样自然使最小化的 C 代码在 nesC 文件之外。即许多在 TinyOS 1.x 中用于特定硬件的宏在 TinyOS 2.x 中被 nesC 的组件所取代。

C 类型名（用 typedef 重定义的）应该使用小写形式，词语之间使用下划线分割，并以“_t”结束。C 标识符名（如结构体、联合、枚举）应该用小写形式。这些类型应该提供一个 typedef 类型定义。

C 中无类型指针（如参数中使用的），应该如同其他类型命名，但是以“_ptr_t”结束。函数名应该小写，词之间以下划线分割。宏函数应该使用大写形式，词间使用下划线连接。不推荐使用宏函数，建议使用 inline 函数。常量应该全部大写，词间用下划线。不推荐使用#define 定义整型常量，建议使用枚举类型。全局变量应该使用混合形式，以小写字母开始。

3. 编程风格提示

- 尽量避免在命令中通知事件。
- 任务应尽量短。
- 尽量让代码同步。
- 尽量减少使用原子操作次数，且原子操作应尽量短。在原子操作中调用其他组件函数时应小心。
- 任何时刻应只有一个组件修改指针数据。理想的情况是，在任何时候应该只有一个组件来存储指针变量。
- 在组件中为所有的状态分配存储空间。如果一个应用程序需要动态的存储空间，应在组件中对其进行封装，并设法限制使用者的集合。
- 应通过 enum 来保存存储空间，而不是把变量限制为整常数。
- 在一个软件抽象的配件中，应把 Init 接口和 MainC 连接起来。这会减轻编程器连接 Init 的负担，在启动节点时消除不必要的工作，减少由于忘记连接带来的错误。
- 如果组件是一个独立的可用抽象，那它的名称应该以 C 结尾。如果它是一个抽象的内

部私有组件，名称应该以 P 结尾。永远不要从组件的外部连接一个以 P 结尾的组件。

- 要多使用关键词 `as`。
- 不要忽略结合警告（combine warning）。
- 如果函数的某个参数是一些常数中的一个，应考虑把这个函数定义为许多分开的函数来避免出现错误。如果接口函数的某个参数是一个很大范围内的常数，应考虑使用参数化的接口来节约代码空间。
- 如果组件依赖于“unique”，那么就需要在头文件中 `#define` 一个字符串来避免错误。
- 在任何时候都不要使用“packed”标志。
- 在定义消息格式的时候要使用与平台无关的数据类型。
- 如果不得不对一个与平台无关的数据类型进行有效的运算或者多次访问，那么应考虑把它复制到本地数据变量中。

第 10 章 基于 WSN 的煤矿

井下定位系统

煤矿的自然环境非常复杂多变，这增加了煤矿开采过程中的不安全性，经常导致恶性事故。目前，我国的煤矿安全生产形势非常严峻，特别是近年来矿难时常发生，造成巨大的财产和生命损失。现有的煤矿安全监控系统因为存在种种不足，已经无法满足日益严格的安全需求，迫切需要改善。当煤矿发生塌方、爆炸等事故时，快速确定受困矿工的位置并有效解救是关键。在此种背景下，产生了一种以无线传感器网络为基础的煤矿井下人员定位系统，实现了对现有有线监控系统无法监控且存在事故危险的特殊地段的监控，能有效地提高煤矿安全生产水平。

煤矿井下人员定位系统监控区域是煤矿井下的巷道，监控目标上携有特定的传感器节点，通过对这些传感器节点的实时定位，实现对监控目标的实时定位。基于无线传感器网络的煤矿井下定位系统的设计目标是实现对煤矿井下工作人员所处位置的跟踪与定位，使井上人员可以方便地通过监控系统了解到巷道内与工作面上人员分布状况，并可根据需求精确地跟踪到具体人员的行进路线与当前井下所处位置，以保证生产顺利进行，并为发生紧急状况时的救援工作提供信息支持。

在无线传感器网络中，运用区域定位技术，通过定位网络中的参考节点接收目标节点（矿工随身携带）的无线信号强度 RSSI 和无线信号质量等信息与后台 GIS 系统相结合，对传感器节点进行跟踪与定位，从而随时了解矿工的工作位置与行进路线，在发生紧急状况时可以为救援提供帮助。该系统还能应用于人员调度、监控、考勤等，提高矿山及其他高危行业的管理水平和工作效率。

10.1 功能需求

基于无线传感器网络的煤矿井下定位系统是煤矿井下安全监测和无线定位系统。该系统的设计目标是实现对井下人员的监控、定位和危机报警等功能。系统具体功能如下。

(1) 人员监控

该系统实现了对井下人员上、下班的考勤记录以及工作期间对井下人员工作状态的监控功能。该系统可帮助控制人员对井下工作人员实施有效管理，提高工作效率。

(2) 定位功能

根据对定位精度要求的不同，系统实现的定位功能又分为两种：区域定位和精确定位。

1) 区域定位, 区域定位对定位精度要求低, 仅需要系统能够知晓移动节点所经过的区域。在本系统中, 要求 Sink 节点在移动节点经过该 Sink 节点的通信覆盖区域时能够完成对移动节点的通过检测。

2) 精确定位, 精确定位对定位精度的要求较高。在本系统中, 要求对移动节点的定位精度小于等于 3 米。

(3) 报警功能

在报警功能中, 报警信号可由移动节点的佩戴者触发, 也可由控制中心触发。由移动节点的佩戴者触发的报警称为危机报警; 由控制中心触发的报警称为疏散警报。

(4) 危机报警

危机报警由佩戴移动节点的矿工人员触发。当煤矿井下出现瓦斯浓度过高、矿井透水和矿井坍塌等异常情况时, 矿工通过移动节点上的按键, 人工发出危机报警信号, 向控制中心报警。

(5) 疏散警报

疏散警报由控制中心触发。当井下存在风险、需要井下人员疏散时, 控制中心触发疏散警报, 通过移动节点上频繁闪烁的指示灯通知井下人员撤离。控制中心可根据井下具体情况通知某一区域或者是整个矿井的矿工疏散。

10.2 系统设计

整个系统由两个部分组成: 井上部分和井下部分, 如图 10-1 所示。

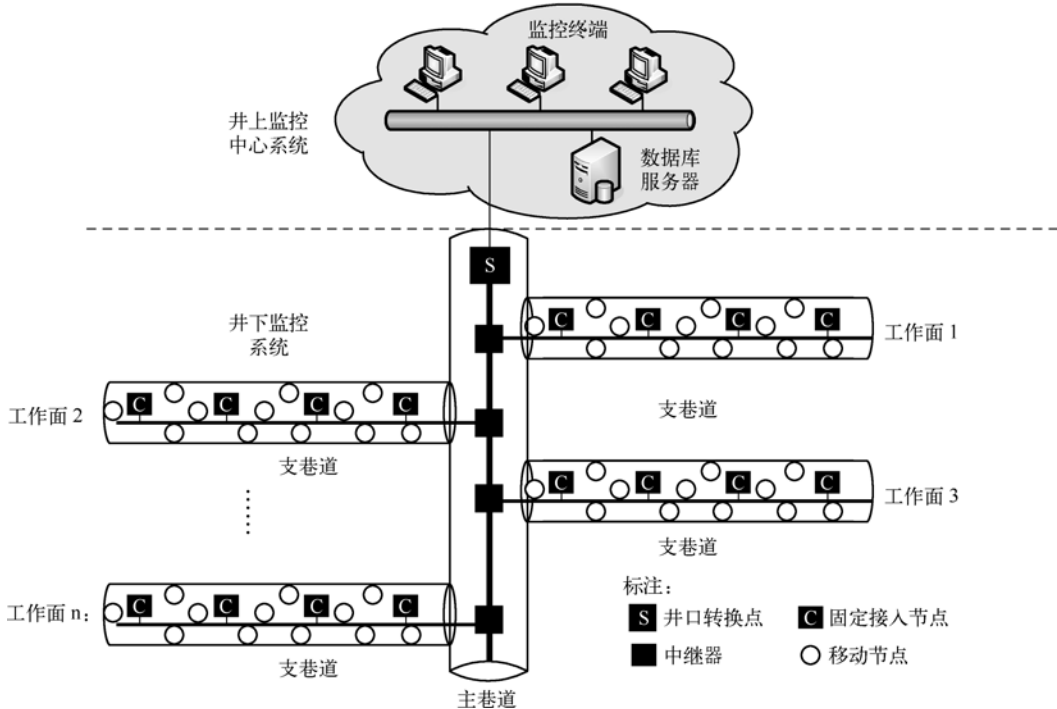


图 10-1 煤矿系统组成

井下部分是整个系统的核心, 由有线传输网络和无线定位网络两部分组成。无线定位网

络主要包括两类节点：固定接入节点和移动节点。移动节点向固定接入节点发送定位信标，使系统可以对其进行实时定位与跟踪；固定接入节点主要负责采集移动节点发送的定位信息。有线传输网络负责将固定接入节点采集的信息传送到井上监控系统。固定接入节点所获取的定位信息通过 485 总线回传至控制器，控制器汇聚多个固定接入节点采集的信息，并通过以太网发送到井上监控中心。

井上监控中心系统由数据库服务器和监控终端等设备组成，它们之间通过以太网连接，为系统提供数据存储、处理和显示功能。

10.3 硬件组成

在硬件方面，系统自主搭建了基于 MSP430F1611-CC1100 硬件平台方案，利用 CC1100 无线通信芯片提供的强大的硬件功能来实现目标功能。

无线通信芯片 CC1100 是一款功能强大，操作简便并且原理简单的芯片。CC1100 不仅可以完成数据接收与发送的基本功能，并且可以自动进行位同步、分组同步、CRC 校验，对数据进行前向纠错（Forward Error Correction, FEC）、白化（whitening）以及休眠与唤醒^[29]。所以，在实现上，只需对相应的寄存器进行配置即可，而不像 TR3000 通信芯片，需要利用软件来实现各个功能。

无线通信芯片 CC1100 是通过简单的 4 线 SPI 接口（SI，SO，SCLK，CSn）来进行配置的，并且一般作为从机。其中，SI 引脚为数据输入引脚；SO 引脚为数据输出引脚；SCLK 引脚为时钟输入引脚；CSn 引脚为片选引脚，在地址和数据传输时，该引脚必须保持为低。简单电路连接如图 10-2 所示。

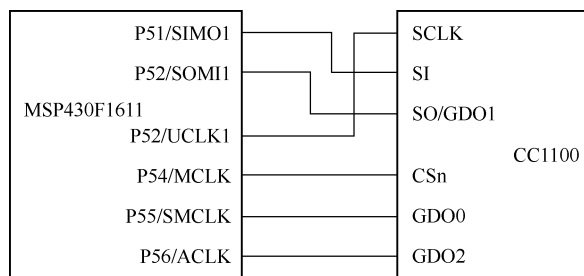


图 10-2 MSP430F1611 与 CC1100 的简单电路连接图

10.4 TinyOS 实现

10.4.1 软件结构框架

在基于 TinyOS 操作系统的具体代码实现过程中，需要把目标功能层次化和模块化，把功能分为上层功能和下层功能，不同功能用不同模块表示，模块间用接口连接，接口具体功能的实现是在下层模块中完成的，上层模块通过接口来调用下层模块的功能。模块的连接关系用配件文件表示。如图 10-3 和图 10-4 所示，分别为项目组设计出的 Sink

节点和移动节点基本功能结构图。图中显示了程序中各组件和接口及他们相互连接调用的关系。其中实线框表示模块，虚线框表示配件。箭头左边的为上层模块，箭头右边的为下层模块。

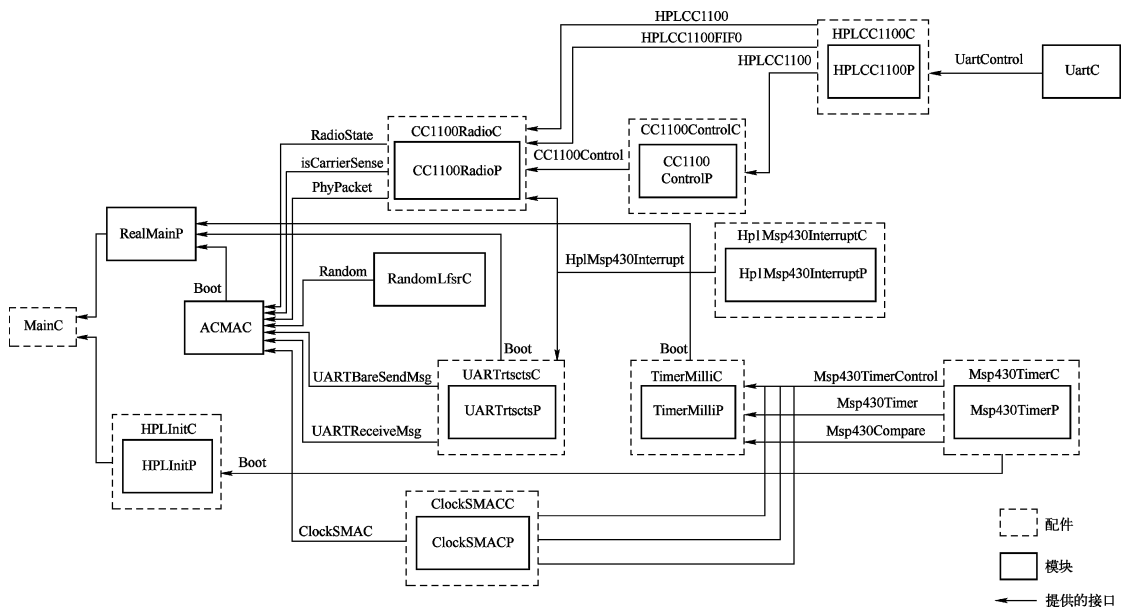


图 10-3 Sink 节点软件结构

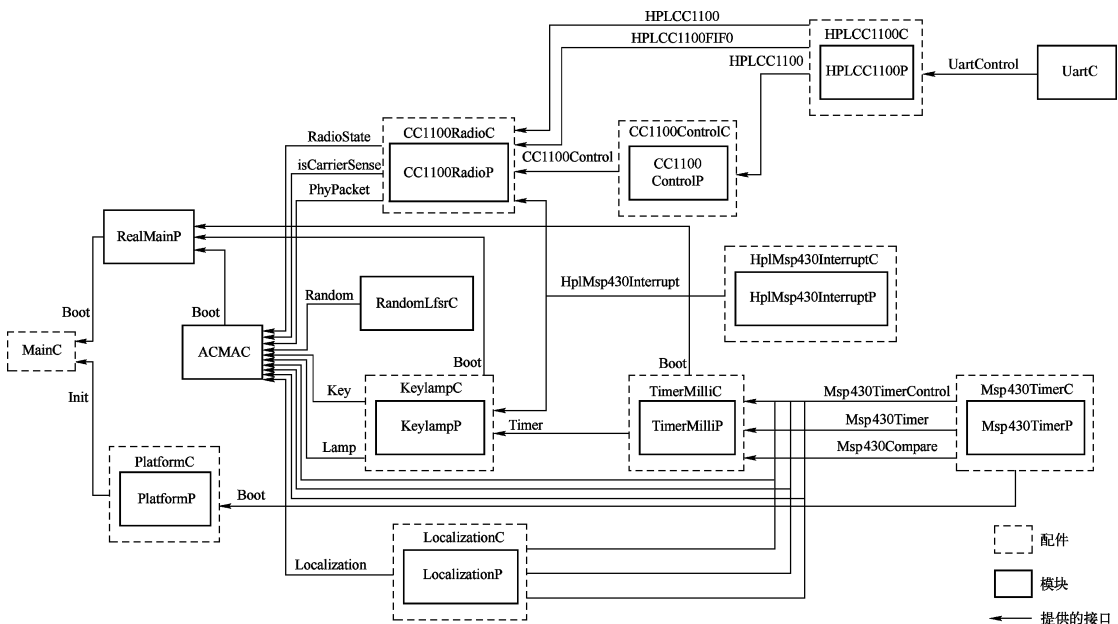


图 10-4 移动节点软件结构

10.4.2 模块介绍

1) 有些模块是 TinyOS 系统自带的, 如表 10-1 所示。主要用于对节点处理器 MSP430

的初始化和配置,在编写 nesC 程序时,需根据功能需求改变其中一些函数或寄存器的配置,然后再调用它们来完成对 MSP430 的控制。

表 10-1 用到的 TinyOS 系统自带模块列表

组 件	功 能	应 用 说 明
MainC	应用程序入口	程序自带文件,编程时不用改变
PlatformC	硬件初始化	
RandomLfsrC	产生随机数,用于随机退避	编程时需对它们进行连接和调用来实现其功能
VartC	用于 MSP430 的串口通信和 SPI 通信	
TimerMillip	提供 1 秒 1 次到 15 分钟 1 次的时钟中断	
Hpl MSP430InterruptP	产生 Msp430 的 P1、P2 端口中断	
Msp430TimerP	产生 Msp430 时钟中断	用于晶振选择、MSP430 时钟系统的配置和选择、TimerA 和 TimerB 模块的配置等
Msp430ClockP	用于 Msp430 系统时钟的初始化	

2) 另外在自主设计的硬件平台基础上更新了 TelosB 软件平台,主要用于对节点的传感器模块和射频模块进行配置和管理等。在具体实施过程中,需改变、连接和调用这些模块。本系统在实现时,主要用到了如表 10-2 所示的三个模块。

表 10-2 用到的 TelosB 平台下模块列表

组 件	功 能	应 用 说 明
CC1100ControlP	初始化收发器 CC1100	根据不同应用调用它来对 CC1100 的发送频率、调制方式、载波门限、收发数据包结构进行配置
HPLCC1100P	MSP430 通过 SPI 收发器 CC1100 的 FIFO 写数据或从 FIFO 中读数据	在实现数据收发功能时需调用它
UARTTrtsctsC	为 Sink 节点存储并转发聚合数据	Sink 节点有线通信时调用它

3) 在软件实现部分最重要的是根据 WSN 的不同应用开发相应的功能模块,本系统主要设计开发了如表 10-3 所示的模块。

表 10-3 节点功能模块列表

组 件	功 能	说 明
ACMAC	节点流程模块,包括节点 MAC 层流程处理器,以及射频、晶振不同步的休眠和唤醒	最上层的模块文件,负责程序的运行顺序
CC1100Radio	数据收发	控制射频模块的数据收发
LocalizationC	节点定位	主要运用 RSSP 测距技术实现定位
ClockSMACC	时钟,可以提供毫秒级的时钟	通过配置表 7 中 MSP430ClockP,并调用 MSP430TimerP 来完成,实现略

10.4.3 主要接口与连通情况介绍

在顶层配置文件 ACMAAppC 中,使用到了组件 MainC、ACMAC、CC1100RadioC、

RandomLfsrC、ClockSMACC、UARTTrtsctsC，在组件 CC1100RadioC 中提供了接口 PhyPacket、RadioState、isCarrierSense，它们的功能分别是物理层数据包分组、射频占用状态、载波监听，与顶层配置文件的连通情况如下：

```
ACMAC.PhyState -> CC1100RadioC;
ACMAC.isCarrierSense -> CC1100RadioC;
ACMAC.PhyPacket -> CC1100RadioC;
```

在 Sink 节点的应用程序中需要对串口通信进行实现，在这里顶层配置文件通过组件 UARTTrtsctsC 提供的 Send、Receive 接口完成 Sink 节点的串口通信功能。

```
ACMAC.Send -> UARTTrtsctsC;
ACMAC.Receive -> UARTTrtsctsC;
```

另外，还用到了 RandomLfsrC 的用于产生随机数的 Random 接口等等。

```
ACMAC.Random -> RandomLfsrC;
ACMAC.Clock -> ClockSMACC;
```

10.4.4 节点运行流程

1. 移动节点流程

如图 10-5 所示，移动节点流程如下。

1) 节点初始化，包括 MSP430 的各端口、寄存器、时钟、CC1100 工作状态以及各个变量。

2) 判断节点周期定时器 t1 是否到期，如果到期则进行第 3) 步。

3) 节点周期定时器重新赋值，t1=3000ms。

4) 判断标志位 flag 是否等于 0。如果不等于 0，表示节点在上一轮数据收发中收到了区域定位节点的 ACK 信号，处于 15000~20000ms 的休眠中；如果等于 0，表示节点在上一轮休眠中没有收到 ACK 信号或收到 ACK 信号并且已完成休眠，这样可以进行第 5) 步。

5) 在 1000ms 内随机选择一个时间点 t2，并休眠。

6) 时间 t2 到期，最大退避次数设为 6。

7) 判断信道忙闲。如果信道忙，则退避数减 1，并在 11~20ms 内选择一个时间 t3，并休眠，t3 到期后重复步骤 7)；如果信道闲，则节点从休眠模式中激活，并发送数据包，且设置最大的 ACK 接收时间 t4 为 11ms。

8) 判断是否收到 ACK。如果在 t4 到期内都没有收到 ACK 信号，表明没有来自控制中心的控制命令，则节点休眠，等待下一次周期定时器到期，即进入步骤 2)；如果在 t4 到期前收到了 ACK 信号，就提取其中的控制信息并执行。则在 15000~20000ms 内随机选择一个时间点 t5 作为休眠时间，并把标志位 flag 置 1，这样即使周期定时器到期，也不会发送定位分组。

9) t5 到期，把标志位 flag 复位。并重复以上步骤。

2. Sink 节点流程

Sink 节点流程较为简单，具体如下。

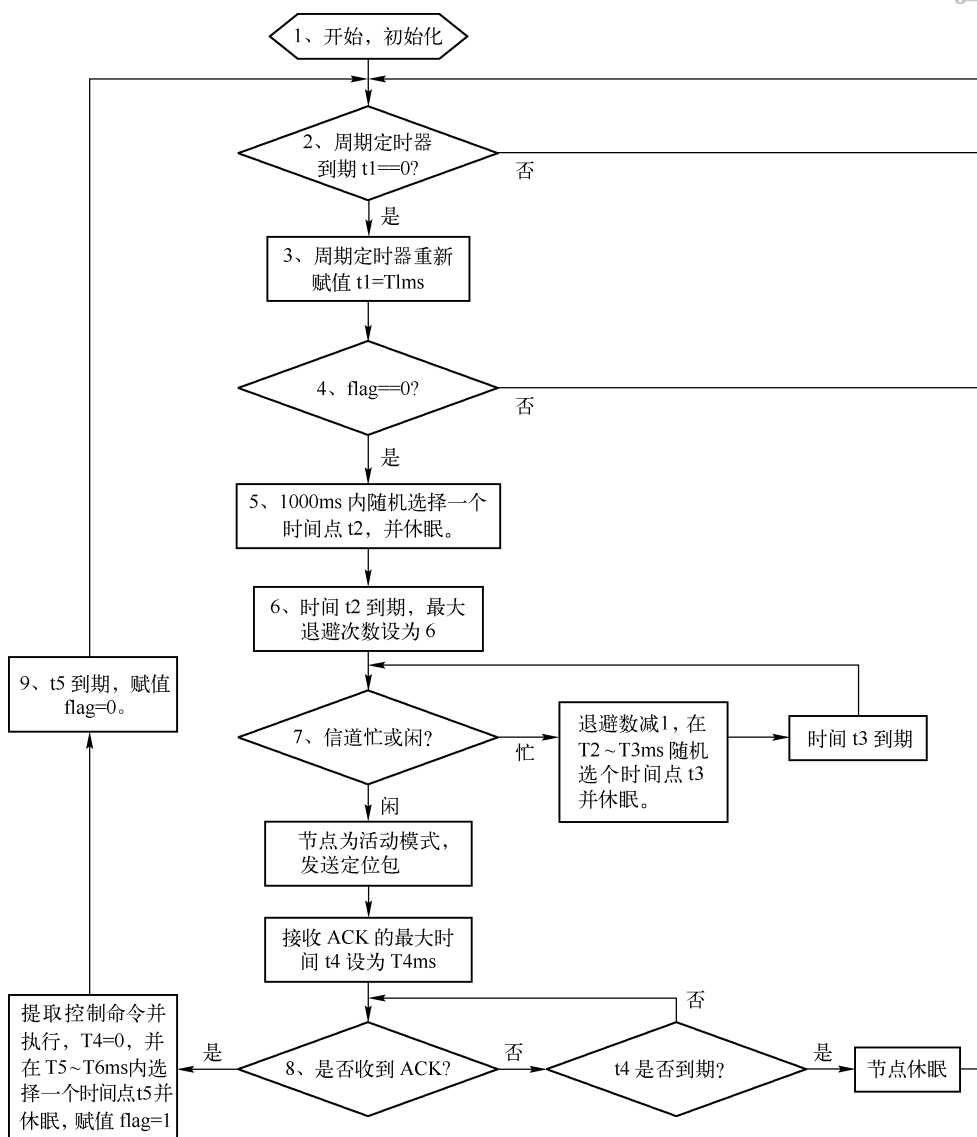


图 10-5 移动节点流程图

1) 节点初始化。

2) 侦听信道，判断有无信号需要接收。

3) 当侦听到空间有信号时，接收并判断包类型号，如为移动节点发来的数据报，则调用 UARTTrtsctsC 模块，并回复嵌有控制命令的 ACK 包。

3. 无线通信模块在 TinyOS 系统下的实现

在节点流程模块在 TinyOS 平台下实现时生成的组件连接图中，ACMAC 为主程序，它通过调用 CC1100RadioC 来收发数据包；通过调用 RadomLfsrC 来产生随机退避数；通过调用 Msp430TimerC 来提供 ms 级时钟；通过调用 UartC 来进行串口通信；通过调用 KeyLampC 来完成电量指示和报警按键驱动。如图 10-6 所示。

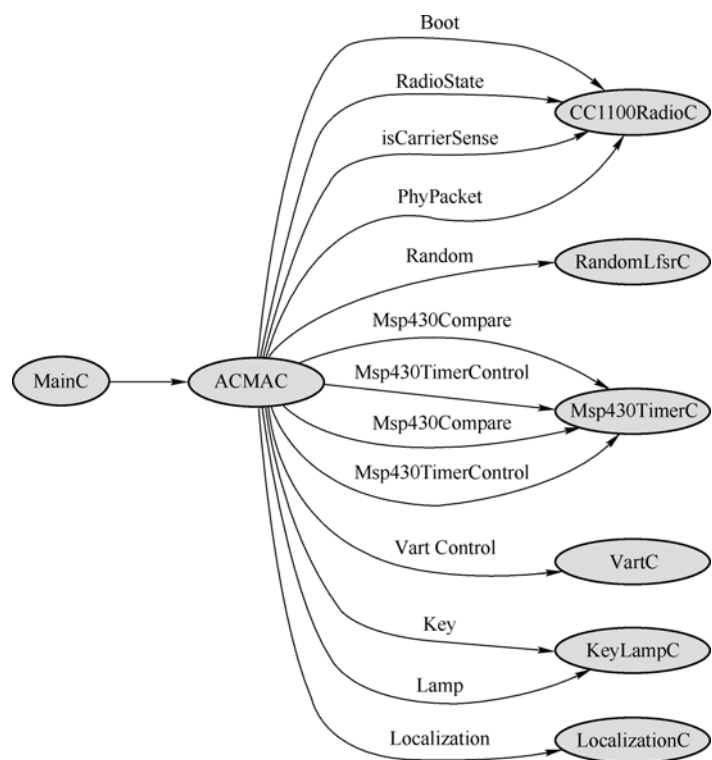


图 10-6 无线通信模块在 TinyOS 平台下的实现

附录 缩 略 语

WSN: Wireless Sensor Network	无线传感器网络
DSN: Distributed Sensor Networks	分布式传感器网络项目
DARPA: Defense Advanced Research Projects Agency	美国国防部高级项目研究署
CEC: Cooperative Engagement Capability	协同交战能力系统
REMBASS: Remote Battlefield Sensor System	远程战场传感器网络系统
TRSS: Tactical Remote Sensor System	战术远程传感器系统
CENS: center of embedded networked sensing	嵌入式智能传感器研究中心
NeTS: networking technology and systems	网络技术与系统
VFS: Virtual file system	虚拟文件系统
nesC: network embedded system C	网络嵌入式系统 C 语言
AM: Active Message	主动消息
HAA: Hardware Abstraction Architecture	硬件抽象结构
HPL: Hardware Presentation Layer	硬件表示层
HAL: Hardware Adaptation Layer	硬件适配层
HIL: Hardware Interface Layer	硬件接口层
HII: Hardware Independent Interfaces	硬件无关接口
ADC: Analog to Digital Converter	模数转换器
SPI: Serial Peripheral Interface	串行外设接口

参 考 文 献

- [1] Business Week Online. Tech wave 2: The sensor revolution[EB/OL]. http://www.businessweek.com/magazine/content/03_34/b3846622.htm.
- [2] 李晓维, 许勇军, 任丰原. 无线传感器网络技术[M]. 北京: 北京理工大学出版社, 2007.
- [3] 孙利民, 李建中. 无线传感器网络[M]. 北京: 清华大学出版社, 2005.
- [4] Lacoss R, Walton R. Strawman design of a DSN (distributed sensors networks) to detect and track low flying aircraft. In: Proc. of Distributed Sensor Nets Conference, Pittsburgh, PA, 1978: 41-52.
- [5] C-Y Chong. Sensor Networks: Evolution, Opportunities, and Challenges[EB/OL]. IEEE, 0018-9219/03.
- [6] William J. Kaiser. Low Power Wireless Integrated Microsensors[EB/OL]. [http://www. Janet. Ucla.edu/WINS/lwim-innovative.htm](http://www.Janet.Ucla.edu/WINS/lwim-innovative.htm).
- [7] G J Pottie. Wireless Sensor Networks. In: IEEE Information Theory Workshop, 1998: 139-140.
- [8] MIT uAMPS project[EB/OL]. <http://www-mtl.mit.edu/research/icesystems/uamps/>.
- [9] NMS project[EB/OL]. <http://nms.csail.mit.edu/>.
- [10] Kahn J M, Katz R H, Pister K S J. Next century challenges: mobile networking for smart dust. Proc. of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'99), ACM SIGMOBILE, Seattle, USA, 1999:483-492.
- [11] Codeblue project. <http://www.eecs.harvard.edu/~mdw/proj/codeblue/>.
- [12] SCADDS project. <http://www.isi.edu/scadds/>.
- [13] CENS: Center for Embedded Networked Sensing. <http://research.cens.ucla.edu/>.
- [14] Networking Technology and Systems (NeTS)[OL]. [http:// www. Nsf .gov/ pubs/2007/nsf07507/nsf07507.htm](http://www.Nsf.gov/pubs/2007/nsf07507/nsf07507.htm).
- [15] 国家中长期科学和技术发展规划纲要(2006-2020)[OL]. [http:// www. gov. cn/jrzg/2006-02/ 09/content_183787.htm](http://www.gov.cn/jrzg/2006-02/09/content_183787.htm).
- [16] LF Akyildiz, W Su, Y Sankarasubramaniam, E Cayirci. Wireless sensor network: a survey. Computer Networks[J], 2002, 38(12):393-422.
- [17] Estrin D, Govindan R, Heidemann J, Kumar S. Next century challenges: Scalable coordinate in sensor network[C]. Proc. of the 5th ACM/IEEE International Conference on Mobile Computing and Networking, 1999: 263-270.
- [18] 于宏毅, 李鸥, 张效义. 无线传感器网理论、技术与实现[M]. 北京: 国防工业出版社, 2009.
- [19] 任丰原, 黄海宁, 林闯. 无线传感器网络[J]. 软件学报, 2003, 14(7).
- [20] 马祖长, 孙怡宁, 梅涛. 无线传感器网络综述[J]. 通讯学报, 2004, 25(4): 114-124.
- [21] Philip Levis, San Madden, David Gay, Joseph Polastre, Robert Szewczyk, Alec Woo, Eric Brewer, David Culler. The Emergence of Networking Abstractions and Techniques in TinyOS. In: Proceedings of the First Symposium on Networking Systems Design and Implementation, 2004.
- [22] S Bhatti, J Carlson, H Dai, et al. MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms. ACM/Kluwer Mobile Networks & Application (MONET)Journal, Special Issue on Wireless Sensor Networks, vol.10, no.4. August 2005.

精品教材推荐目录

序号	书号	书 名	作者	定价	配套资源
1	23989	新编计算机导论	周 苏	32	电子教案
2	33365	C++程序设计教程——化难为易地学习 C++	黄品梅	35	电子教案
3	36806	C++程序设计 ——北京高等教育精品教材立项项目	郑 莉	39.8	电子教案、源代码、习题答案
4	23357	数据结构与算法	张晓莉	29	电子教案、配套教材、习题答案
5	08257	计算机网络应用教程(第3版) ——北京高等教育精品教材	王 洪	32	电子教案
6	30641	计算机网络——原理、技术与应用	王相林	39	电子教案、教学网站、超星教学录像
7	20898	TCP/IP 协议分析及应用 ——北京高等教育精品教材	杨延双	29	电子教案
8	36023	无线移动互联网：原理、技术与应用 ——北京高等教育精品教材立项项目	崔 勇	52	电子教案
9	24502	计算机网络安全教程(第2版)	梁亚声	34	电子教案
10	25930	网络安全技术及应用	贾铁军	41	电子教案
11	33323	物联网技术概论	马 建	36	电子教案
12	34147	物联网实验教程	徐勇军	43	配光盘
13	37795	无线传感器网络技术	郑 军	39.8	电子教案
14	39540	物联网概论	韩毅刚	45	电子教案、教学建议
15	26532	软件开发技术基础(第2版) ——“十二五”普通高等教育本科国家级规划教材	赵英良	34	电子教案
16	28382	软件工程导论	陈 明	33	电子教案
17	33949	软件工程(第2版)	瞿 中	42	电子教案
18	37759	软件工程实践教程 (第2版)	刘 冰	49	电子教案
19	08968	数值计算方法(第2版)	马东升	25	电子教案、配套教材
20	28922	离散数学(第2版) ——“十一五”国家级规划教材	王元元	34	电子教案
21	41926	数字逻辑(第2版)	武庆生	36	电子教案
22	35895	Linux 应用基础教程 ——Red Hat Enterprise Linux/CentOS 5	梁如军	58	电子教案
23	40995	单片机原理及应用教程(第3版)	赵全利	39	电子教案、习题答案、源代码
24	23424	嵌入式系统原理及应用开发 ——北京高等教育精品教材	陈 渝	38	电子教案
25	19984	计算机专业英语	张强华	32	电子教案、素材、实验实训指导、配光盘
26	28837	人工智能导论	鲍军鹏	39	电子教案
27	31266	人工神经网络原理 ——北京高等教育精品教材	马 锐	25	电子教案
28	26103	信息安全概论	李 剑	28	电子教案
29	40967	计算机系统安全原理与技术(第3版)	陈 波	49	电子教案
30	33288	网络信息对抗(第2版) ——“十一五”国家级规划教材	肖军模	42	电子教案、配套教材
31	37234	网络攻防原理	吴礼发	38	电子教案
32	40081	防火墙技术与应用	陈 波	29	电子教案

机工出版社·计算机分社书友会邀请卡

尊敬的读者朋友：

感谢您选择我们出版的图书！我们愿以书为媒与您做朋友！我们诚挚地邀请您加入：

“机工出版社·计算机分社书友会”

以书结缘，以书会友

加入“书友会”，您将：

- ★ 第一时间获知新书信息、了解作者动态；
- ★ 与书友们在线品书评书，谈天说地；
- ★ 受邀参与我社组织的各种沙龙活动，会员联谊；
- ★ 受邀参与我社作者和合作伙伴组织的各种技术培训和讲座；
- ★ 获得“书友达人”资格（积极参与互动交流活动的书友），参与每月5个名额的“书友试读赠阅”活动，获得最新出版精品图书1本。

如何加入“机工出版社·计算机分社书友会”

两步操作轻松加入书友会

Step1

访问以下任一网址：

- ★ 新浪官方微博：<http://weibo.com/cmpjsj>
- ★ 新浪官方博客：<http://blog.sina.com.cn/cmpbookjsj>
- ★ 腾讯官方微博：<http://t.qq.com/jigongchubanshe>
- ★ 腾讯官方博客：<http://2399929378.qzone.qq.com>

Step2

找到并点击调查问卷链接地址（通常位于置顶位置或公告栏），完整填写调查问卷即可。

联系方式

通信地址：北京市西城区百万庄大街22号

机械工业出版社计算机分社

邮政编码：100037

联系电话：010-88379750

传 真：010-88379736

电子邮件：cmp_itbook@163.com

敬请关注我社官方微博：<http://weibo.com/cmpjsj>

第一时间了解新书动态，获知书友会活动信息，与读者、作者、编辑们互动交流！

在线互动交流平台

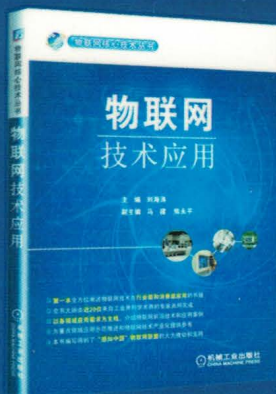
官方微博: <http://weibo.com/cmpjsj>

豆瓣网: <http://site.douban.com/139085/>

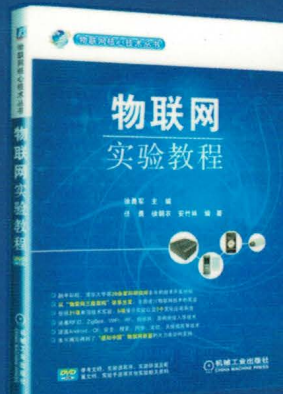
读者信箱: cmp_itbook@163.com



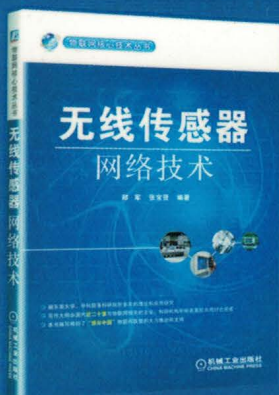
物联网技术概论
ISBN 978-7-111-33323-4



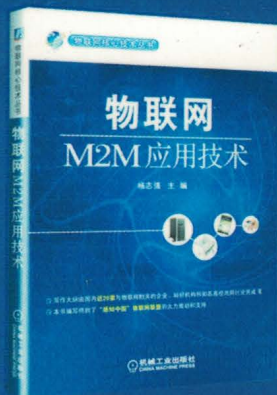
物联网技术应用
ISBN 978-7-111-34168-0



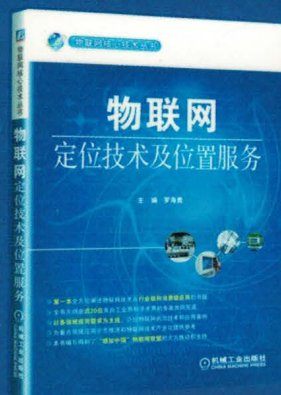
物联网实验教程
ISBN 978-7-111-34147-5



无线传感器网络技术
ISBN 978-7-111-37795-5



物联网M2M应用技术
即将出版



物联网定位技术及位置服务
即将出版

地址: 北京市百万庄大街22号

邮政编码: 100037

电话服务

社服务中心: 010-88361066

销售一部: 010-68326294

销售二部: 010-88379649

读者购书热线: 010-88379203

网络服务

教材网: <http://www.cmpedu.com>

机工官网: <http://www.cmpbook.com>

机工官博: <http://weibo.com/cmp1952>

封面无防伪标均为盗版

上架指导 计算机/物联网

ISBN 978-7-111-40722-5

策划编辑◎ 郝建伟 / 封面设计◎



子时文化
Zishi Culture

ISBN 978-7-111-40722-5



9 787111 407225 >

定价: 36.00 元