论文题目　无线传感器网络文件系统与重编程技术研究

学 科 专 业　计算机系统结构

学　　号　201021060111

作 者 姓 名　苏铅坤

指 导 教 师　罗克露　教授

分类号 _____ 密级 _____

UDC [注1] _____

学 位 论 文

无线传感器网络文件系统与重编程技术研究

（题名和副题名）

苏铅坤

（作者姓名）

指导教师 罗克露 教 授

电子科技大学 成 都

（姓名、职称、单位名称）

申请学位级别 硕士 学科专业 计算机系统结构

提交论文日期 _____ 论文答辩日期 _____

学位授予单位和日期 电子科技大学 年 月 日

答辩委员会主席 _____

评阅人 _____

注 1：注明《国际十进分类法 UDC》的类号。

RESEARCH ON FILE SYSTEM AND REPROGRAMMING TECHONOLOGY BASED ON WIRELESS SENSOR NETWORKS

Master Thesis Submitted to

University of Electronic Science and Technology of China

Major: **Computer Architecture**

Author: **Su Qiankun**

Advisor: **Luo Kelu**

School: **School of Computer Science & Engineering**

独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得电子科技大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

作者签名：_____ 日期： 年 月 日

论文使用授权

本学位论文作者完全了解电子科技大学有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权电子科技大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后应遵守此规定）

作者签名：_____ 导师签名：_____

日期： 年 月 日

摘　要

无线传感器网络(Wireless Sensor Networks，WSNs)以低成本、低功耗、分布式和自组织特点带来一场信息感知的变革。随着传感器节点上应用程序复杂性的提升，节点迫切需要操作系统管理多个任务，电子信息技术飞速发展使传感器节点搭建操作系统成为可能。即便如此，相对于 PC 甚至嵌入式系统，传感器节点仍然是资源受限，这就要求操作系统每一部分都需要精心设计。

文件系统和重编程作为 WSNs 操作系统重要组成部分，其设计优劣直接影响整个系统的性能。鉴于文件系统与重编程技术有着紧密联系，本文将两者结合起来考虑，整体设计，以获得全局最优。本研究主要内容：

1.剖析 Contiki 操作系统。本文选取 Contiki 作为研究对象，鉴于 Contiki 可参考资料甚少，研究开展之前，先深入分析 Contiki 源码重现整个系统技术细节，包括 Contiki 内核、文件系统、动态加载、Rime 协议栈，为后续研究打下夯实的基础。

2.改进 Coffee 文件系统。深入源码分析 Coffee 技术细节，修复多处 BUG 并改进代码增强系统健壮性。除此之外，结合重编程技术特点，从格式化、适应 FLASH 类型、采集数据特点、文件元数据组织等方面对 Coffee 进行改进。

3.改进重编程技术。通过分析 WSNs 操作系统 TinyOS、SOS、MantisOS 理解典型重编程实现方式。在此基础上，分析 Contiki 重编程技术并对其进行改进，即重新设计 ELF 文件结构和文件裁剪。

4.建立开发环境并测试。为了便于调试，将 Contiki 从 GCC 平台转移到 IAR+J-Link，包括开发环境搭建、Contiki 系统移植。在此平台基础上，对文件系统和重编程进行功能性测试。并介绍网络仿真器 COOJA 在测试 Contiki 重编程中的应用。

本文深入源码详细分析了 Contiki 内核、Coffee 文件系统、重编程技术、Rime 协议栈，并对 Coffee 和重编程进行改进，具有现实意义。

关键词：无线传感器网络，文件系统，重编程，Contiki，Coffee

ABSTRACT

Wireless Sensor Networks (WSNs) are bringing about a revolution on information perception on account of the advantages of low-cost, low-power, distributed and self-organizing. As the increasing complexity of application on sensor nodes, operating systems is desperately needed for multi-task management and the booming of Electronic information technology makes it possible. Even so, sensor nodes are still resource-constrained, compared with PC even embedded system, which requires that each part of WSNs operating system is well-designed

File system and reprogramming are significant components of WSNs operating system and its design will directly affect the performance of entire system. Since the reprogramming is closely linked with file system, this research refines both of them for global optimum based on consideration of their correlation. The main contents of this study are as follows:

(i)Analyze Contiki operating system deeply. This research selects Contiki operating system as the object of study. However, the references of Contiki are very limited. As a result, analyzing the source code of Contiki deeply is needed to understand its technical details, including Contiki kernel, file system, dynamic loading and Rime stack, which lays a solid foundation for the following research.

(ii)Refine Coffee file system. After in-depth analysis of Coffee, several bugs are fixed and some code styles are improved to enhance the robustness of Coffee. In addition, the Coffee file sytem is improved in various aspects, considering the features of reprogramming which include Coffee format, adapting to the type of FLASH, the characteristics of collected data and file metadata.

(iii)Refine the reprogramming of Contiki. Three WSNs operating system (TinyOS, SOS and MantisOS) are analyzed for a better understanding of typical implementations on reprogramming. After then, the ELF file format is redesigned and the ELF file is tailored to improve the reprogramming of Contiki.

(iv)Build development platform and test. In order to enhance the abilities of debugging, the development platform of Contiki is ported from GCC to IAR+J-Link,

including setting up the development environment and porting Contiki operating system. The function of revised file system and reprogramming is tested on this platform. Furthermore, the way, using network simulator COOJA, is introduced to test its performance.

This study analyzes Contiki kernel, Coffee file system, reprogramming technology and Rime stack by getting into its source code deeply and refines Coffee and reprogramming of Contiki, which has a realistic significance.

Keywords: WSNs, file system, reprogramming, Contiki, Coffee

目 录

图表目录

第一章 绪 论

无线传感器网络(Wireless Sensor Networks，WSNs)以分布式、自组织特点带来一场信息感知的变革，具有低成本、低功耗优势。电子技术飞速发展使无线传感器节点搭载操作系统成为可能，这也迎合了在节点上处理复杂任务的需求。文件系统和重编程技术作为 WSNs 操作系统重要组成部分，其设计的优劣对整个系统的性能具有重要影响。

1.1 课程研究背景及意义

WSNs 是由一组传感器节点以 ad hoc 方式连接而成，能够协同监测、感知网络覆盖的环境并采集数据，传感器节点对这些信息处理后以多跳自组织方式传递给观察者[1]。WSNs 广泛应用于多个领域，包括军事、智能家居、工农业控制、环境检测、生物医疗、交通控制[2-4]。物联网的兴起，更是给 WSNs 发展带来新的生机。

应用程序复杂性的提升给传统传感器网络(没有搭载操作系统)带来新的挑战，复杂的应用程序迫切需要传感器节点能运行多个任务，任务的增加需要操作系统进行有效管理。电子信息技术飞速(如提升运算能力，增加存储空间)发展使传感器节点搭建操作系统成为可能。

即便如此，相对于 PC 或者嵌入式系统，传感器节点仍然是资源受限(如运算能力有限、内存空间有限)，除此之外，传感器节点对功耗非常敏感，加之，WSNs 自身特性(大量节点、自组织、动态)，这些使得原有桌面甚至是嵌入式的一些系统和机制不能很平滑地应用到传感器节点和 WSNs，这就要求开发新的技术、机制、软件来适应这些传感器节点和 WSNs，包括节点操作系统、文件系统、协议栈、重编程技术、应用服务。

文件系统通常是无线传感器网络操作系统必不可少的一部分。传感器节点主要用于采集数据，需要文件系统对采集到的数据进行有效管理。除此之外，以下两种特殊情况需要将采集到的数据进行本地存储：

①数据传递时中断(如网络动态不稳定造成)，需要重传；

②合并多次采集到的数据，打包发出，减少通信，进而降低功耗。

重编程技术是 WSNs 必不可缺的一部分。传感器节点投入运行之后，难免需要对其进行更新(如系统升级、增加功能、BUG 修复)，然而节点一经布署将难以收回(人工去收集大规模节点耗时耗力，况且有些节点不能中途被中断)，这就要求有一种技术实现节点的远程更新，即重编程技术。

事实上，文件系统与重编程技术有着紧密联系，如接收到的更新文件如何在文件系统有效地存储。鉴于此，本文将两者结合起来考虑，整体设计，以获得全局最优。

目前已有多款无线传感器网络操作系统[5]实现了文件系统和重编程技术，最著名的有 TinyOS、Contiki，各有优缺点。TinyOS 是用 nesC 语言编写，除了给阅读源码带来不便外，没有可用的支持 nesC 源码级调试工具，这给开发、测试带来极大不便。而 Contiki 使用标准 C 语言开发，有完整的开发工具支持，故以 Contiki 为载体研究文件系统和重编程技术。

1.2 国内外研究情况

美国国防部于 1998 年提出“智能尘埃”，开启了无线传感器网络研究的先河。此后，WSNs 飞速发展。物联网的兴起，更是给 WSNs 发展带来新的浪潮。标准化上，IEEE 发布低速率、低成本、低功耗专用无线通信协议 802.15.4，ZigBee 联盟制定了传感器节点组网的协议规范；硬件上，专用的 SoC(System on Chip)开始大量涌现；软件上，一些国际著名公司和组织开发了诸如 Z-Stack、uIP 适应 WSNs 协议栈，节点网络操作系统也大量被开发，其中最著名的有 TinyOS、Contiki。在现代意义的 WSNs 研究上，我国几乎与发达国家同时启动[6]。

1.2.1 Contiki 研究现状

Contiki[7]是由瑞典计算机科学研究所开发，具有开源、高度可移植、多线程特性的节点操作系统。Contiki 已经发展成一个完整的操作系统，内核采用事件驱动和 Protothreads[8][9]机制。为了适应大规模数据存储，开发了专用文件系统 Coffee[10]，文献[11]为 Contiki 设计了类 UNIX 的硬件抽象层，方便编程。

Contiki 还提供丰富的通信协议，支持 IPv4 和 IPv6，并实现了适合 8 位微控制器的 TCP/IP 协议栈 uIP，支持 TCP、UDP、ICMP、IP 协议。Contiki 还提供另一种轻量级层次(lightweight layered)协议栈 Rime，Rime 提供单跳单播、单跳多播、多跳传输支持[5]。除此之外，Contiki 还实现了 ContikiRPL[12]，适合低功耗有损(lossy)

网络的 IP 路由协议。文献[13]将 XMPP(Extensible Messaging and Presence Protocol，一种以 XML 为基础的开放式实时通信协议)移植到 Contiki，称为 uXMPP。

在可靠性、安全性方面，Contiki 也得到了完善，文献[14]采用代理(Deputy)的方法，使内存使用更加安全。文献[15]设计了一种安全网络层 ContikiSec，以在安全和功耗间取得平衡。鉴于 WSNs 的不可信，文献[16]提出一种适合 Contiki 的延迟可容忍(Delay Tolerant)传输协议。

尽管 Contiki 发展得比较成熟，一些学者也在 Contiki 上做了不少研究，但能找到的 Contiki 学习资料非常有限(仅有开发者撰写的几篇论文[7-9]及官方 Contiki Wiki[17]，极少的第三方资料，官网[18]则是主要发布一些新闻)，这些资料叙述过于简单，很少深入技术细节。国内外有若干论文[19][20]是将 Contiki 作为其开发平台的操作系统，但他们通常选择官方支持的节点，无须移植，直接使用 Contiki API，并没深入讨论技术细节。

1.2.2 WSNs 文件系统研究现状

闪存存储器作为一种非易失性存储器，由于具有体积小、功耗低、抗震性好等优势，使得 FLASH 在嵌入式系统中得到广泛应用。FLASH 的自身的特性决定了 FLASH 文件系统需要考虑额外问题，如耗损平衡(FLASH 擦除块数限制)、坏块管理、掉电保护、垃圾回收、映射机制等。

已有若干文件系统是针对 FLASH 设计的，其中最有名的是 JFFS2 和 YAFFS。JFFS2(Journaling Flash File System, version 2)是基于日志结构的文件系统，为了支持通用型应用，JFFS2 使用相对庞大的数据结构，文件系统挂载后，需要占用较多的内存，这对于内存受限的存感器节点将难以接受。YAFFS 是针对大容量 NAND FLASH 设计的，其每页通常是 512KB 甚至更大，而传感器节点的容量通常很小。所以 JFFSL2 和 YAFFS 不适合用在传感器节点上[21]。

无线传感器网络文件系统除了考虑上述 FLASH 文件系统的特性外(传感器节点存储器通常也是 FLASH)，还需结合 WSNs 的特点进行设计。传感器节点的数据类型少，模式单一，无需实现标准文件系统所有操作，仅需实现基本的文件操作，并根据传感器网络中的数据特点进行优化[21]。除此之外，还需重点考虑以下两点：

①内存受限，故不能设计很大的数据结构驻留内存，也不能有很大的 cache；

②被采集数据特点，数据量的大小、频率等都会影响整个文件系统的设计。

Contiki 开发者设计了适合内存受限的传感器节点文件系统 Coffee[10]，并且适

合大规模存储。当文件修改时，传统的FLASH日志文件系统是将文件拷贝至RAM，修改后写入新的文件，并将原来文件标记为失效。而 Coffee 采用一种微日志(micro log)文件，减少了整个文件拷入拷出，提升了效率。在 Coffee 文件系统，当文件修改时，在微日志文件写入新的数据，而不是重新创建新的文件。预分配的空间不足时，Coffee 分配新的空间，并将原始文件和微日志文件一起拷入新文件，同时将原始文件标记为失效，以便后续的垃圾回收。但 Coffee 文件系统没有充分考虑重编程技术带来的影响。

LiteOS[22]提供了层次型文件系统 LiteFS，该系统支持文件和目录。LiteFS 被分为 3 个模块，其一，在 RAM 保存打开文件的文件描述符、内存分配位图、FLASH 布局信息；其二，在 EEPROM 保存层次目录结构信息；其三，用 FLASH 存储数据。LiteOS 将所有单跳结点视为文件，挂载到 LiteFS[5]。但 LiteFS 也没充分考虑重编程技术进行设计。

1.2.3 重编程技术

无线传感器节点一经布署，就很难收回(节点数目之多，有些甚至无法再取回了或者节点工作不能被中断)。若要对节点进行更新(如操作系统升级、BUG 修复、增删应用程序)，则需要重编程(reprogramming)技术对节点进行远程更新。

因为传感器节点资源受限(MCU 处理能力、内存、能量)，这使得大数据传输与存储不可能[23]。基于此，大的数据(如系统映像)需要先拆分再传输，而 WSNs 是自组织网络，传输的性能很大程度上影响着功耗。除此之外，重编程还需满足以下要求：便于使用、安全、可靠、可扩展、性能[23]。

通俗地讲，当需要对节点更新时，需要解决两个问题：其一，如何将数据(如系统映像、配置文件、应用程序)传送到传感器节点；其二，传感器节点如何进行更新。前者叫代码分发协议，后者叫节点端设计。

重编程设计思想总体上可以分为三类[24]：整个代码映像替换、仅替换不同的代码、基于模块替换(Loadable modules)。Contiki 借鉴了 Linux 动态模块加载思想，其重编程技术基于模块。代码分发协议研究上，目前已提出 10 多种代码分发协议。

国内，在重编程也做了少许研究，主要集中在代码分发协议的安全认证上。文献[25]提出一种高可靠、低存储，具有一定实时性的设计方法。文献[26][27]提出一种细粒度、低存储开销、可容忍包乱序的安全认证方案。文献[28]提出一种抗污染攻击的安全重编程方法 PRMR，该方法使用组合技术对污染瓒下的编码包进

行正常译码，并通过邻居分类系统隔离污染者。文献[29]提出了一种基于散列链的无线传感器网络重编程安全认证的改进方案 ADV-Data-Hash.，该方案利用散列链和一次数字签名相结合的方法实现对无线重编程轻量认证。文献[30]对无线重编程 DoS 攻击产生危害进行评价和建模，并提供一些关于无线重编程安全性的理论基础。但这些研究，只是研究一个小点，并没有涉及到文件系统的讨论。浙江大学硕士学位论文[31]简要分析了 Contiki、Mantis、SOS 重编程技术，并在浙大自主研发的 WSNs 操作系统 SenSpire 实现重编程技术，采用动态链接和加载技术，并在基站采用了预加载技术，加速了模块在节点端的链接和加载速度，有效减少重编程过程中的代码分发量。但该论文没有讨论文件系统相关细节。

1.2.4 文件系统与重编程技术

文件系统与重编程有着密切关系，将文件系统和重编程技术作为一个整体考虑，有助于系统整体性能的发挥。

上海交通大学硕士学位论文[32]设计文件系统时考虑了重编程的特点。文件系统是为了方便数据组织和管理设计的，传感器节点的数据有其自身的特点，大致可以归为 3 类，即程序镜像数据、配置数据、采集数据[32]。前两者是针对重编程技术而言，论文只是简单将节点更新的数据笼统归为程序镜像数据。事实上，按节点端的替换方式，可以分为：整个映像代码、不同代码、模块代码，这些将会影响着文件系统的设计。此外，论文仅仅分析了现有的基于 FLASH 文件系统 JFFS(针对 NOR FLASH 的文件系统)、YAFFS，而这两种文件系统，由于其数据结构太大已不适合内存受限的传感器节点。

1.3 本文主要工作

本文旨在从系统整体性能出发，对文件系统和重编程进行改进及优化。本文研究对象是 Contiki，论文主要工作如下：

(1)剖析 Contiki 操作系统

本文选取 Contiki 作为研究对象，然而介绍 Contiki 技术细节的资料几乎没有。为了理解 Contiki 文件系统与重编程技术细节，不得不深入分析 Contiki 源码重现其技术细节，包括 Contiki 内核、文件系统、动态加载、Rime 协议栈。

(2)改进 Coffee 文件系统

Coffee 是一种全新的文件系统，改进之前需要深入源码分析其技术细节。在

分析过程中，修复了 Coffee 文件系统多处 BUG 并改进代码使其更具健壮性。除此之外，结合重编程技术特点，从 5 个方面对文件系统进行改进。

(3)改进重编程程技术

通过分析其它 WSNs 操作系统(TinyOS、SOS、MantisOS)来理解典型的重编程技术实现方式。在此基础上，分析 Contiki 重编程技术并对其进行改进。

(4)建立开发环境并测试

Contiki 并没有支持本文选用的开发板(stm32f103+CC2520+J-Link)，Contiki 提供源码是基于 GCC 开发环境，而 Linux 平台对仿真器 J-Link 支持很不完善，给调试带来极大不便，基于这些考虑，本文决定将 Contiki 从 GCC 平台转移到 IAR+J-Link，包括重新配置整个工程项目和移植整个 Contiki 系统到 IAR+J-Link。

1.4 论文组织结构

本论文主要研究无线传感器网络文件系统和重编程技术，并寻求结合两者考虑后的整体最优，共六章。

第一章 绪论。介绍了 WSNs 文件系统与重编程研究背景及意义，并分别介绍了文件系统、重编程技术以及两者的结合研究情况。扼要介绍本文主要工作。

第二章 WSN 文件系统与重编程技术研究。先阐述了选择 Contiki 作为研究对象的原因，而后剖析 Contiki 操作系统。详细阐述 Contiki 运行原理、内核机制、进程调度、事件调度、定时器、Rime 协议栈。

第三章 WSNs 文件系统研究与改进。详细阐述了 Conitki 自带文件系统 Coffee 的技术细节，给出了若干 BUG 并给出解决方法，阐述 Coffee 文件系统 5 个改进方法。

第四章 WSNs 重编程技术研究与实现。介绍了重编程技术原理，以 TinyOS、SOS、MantisOS 为例，介绍了典型的重编程技术实现方式。接着，剖析 Contiki 重编程技术并对其进行改进。

第五章 文件系统与重编程测试。先介绍了本文选用的软硬件平台，阐述 Contiki 移植细节，在此基础上进行测试。并给出对大规模节点测试方法。

第六章 总结。对本文工作进行总结，指出本研究存在的问题并展望。

第二章 WSNs 文件系统与重编程相关技术研究[①]

文件系统和重编程是无线传感器网络重要组成部分，几乎所有的 WSNs 操作系统都含有这两者，其中最著名的是 TinyOS 和 Contiki。鉴于目前没有支持 TinyOS 的源码级(nesC 语言)调试工具，本文选择 Contiki 作为研究对象。本工作开展之初，Contiki 最高版本为 2.5，除了几篇官方发表的论文及少许的介绍性资料外，没有详细的参考资料。为了深入理解无线传感器网络中的文件系统和重编程技术，不得不深入分析源码重现 Contiki 的技术细节。本文关于 Contiki 所有讨论是基于 2.5 版本。

Contiki 是由瑞典计算机科学研究所开发的专用网络节点操作系统，自 2003 年发布 1.0 版本以来，飞速发展，目前已是一个完整的操作系统，包括文件系统 Coffee、网络协议栈 uIP 和 Rime、网络仿真器 COOJA，并于 2012 年发布全新版本 2.6。Contiki 由标准 C 语言开发，具有很强的移植性，目前已被移植到多种平台，包括 8051、MSP430、AVR、ARM，并得到广泛应用。除此之外，Contiki 将 Protothreads 轻量级线程模型和事件机制完美整合到一起，Proththreads 机制使得系统占用内存极小，事件机制保证了低功耗，非常适合资源受限、功耗敏感的传感器网络。

2.1 Contiki 内核

2.1.1 运行原理

嵌入式系统可以视为一个运行着死循环主函数的系统，Contiki 内核是基于事件驱动的，系统的运行可以视为不断处理事件的过程。Contiki 整个运行是通过事件触发完成，一个事件绑定相应的进程。当事件被触发，系统把执行权交给事件所绑定的进程。Contiki 操作系统运行原理如图 2-1 所示。

事实上，图 2-1 几乎是 Contiki 主函数的流程图。通常情况下，应用程序作为一个进程放在自启动的指针数组中，系统启动后，先进行一系列的硬件初始化(包括串口、时钟)，接着初始化进程，启动系统进程(如管理 etimer 的系统进程 etimer_process)和用户指定的自启动进程，然后进入事件处理的死循环(如上图右边框框所

[①] 分析 Contiki 操作系统源码时，已将分析笔记分享在个人博客 (http://jelline.blog.chinaunix.net)，本章大部分内容是对其的重新整理。

示，实际上是 process_run 函数的功能)。通过遍历执行完所有高优先级的进程，而后转去处理事件队列的一个事件，处理该事件(通常对应于执行一个进程)之后，再次执行所有高优先级进程，而后才转去处理下一个事件。

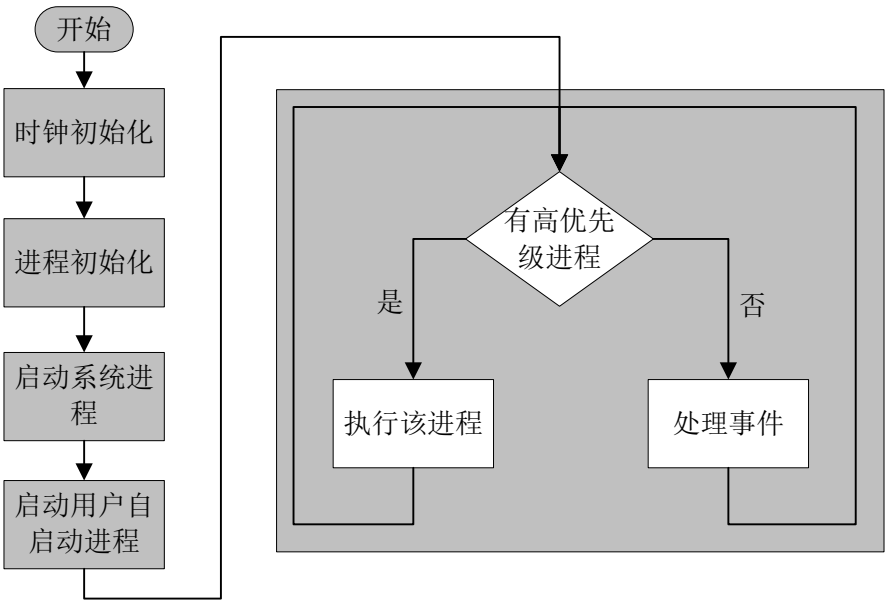

图 2-1 Contiki 运行原理示意图

将 process_run 函数展开，去除无关代码，main 函数关键代码如下：

```
int main()
{
   clock_init();                                  //时钟初始化
   process_init();                                //进程初始化
   process_start(&etimer_process, NULL);      //启动系统进程
   autostart_start(autostart_processes);          //启动用户自启动进程

   while(1)
   {
      /***函数 process_run 的功能***/
      if(poll_requested)
      {
         do_poll();                                  //执行完所有高优先级的进程
      }
      do_event();                                    //仅处理事件队列的一个事件
   }
   return 0;
}
```

2.1.2 进程管理

为确保高优先级任务尽快得到响应，Contiki 采用两级进程调度。进程无疑是一个系统最重要的概念，Contiki 的进程机制是基于 Protothreads[8]轻量级线程模型，在 Protothreads 基础上进行封装。

(1)Protothreads

为了适应内存受限的嵌入式系统，瑞典计算机科学研究所设计了 Protothreads，实际上是一种轻量级无线结构的线程库。传统的桌面操作系统甚至服务器操作系统，每个进程都拥有自己的栈，进行进程切换时，将进程相关的信息(包括局部变量、断点、寄存器值)存储在栈中。然而，对于内存受限的传感器节点几乎不现实，基于这点考虑，Protothreads 巧妙地让所有进程共用一个栈，传统的进程与 Protothreads 机制栈使用对比如图 2-2 所示。

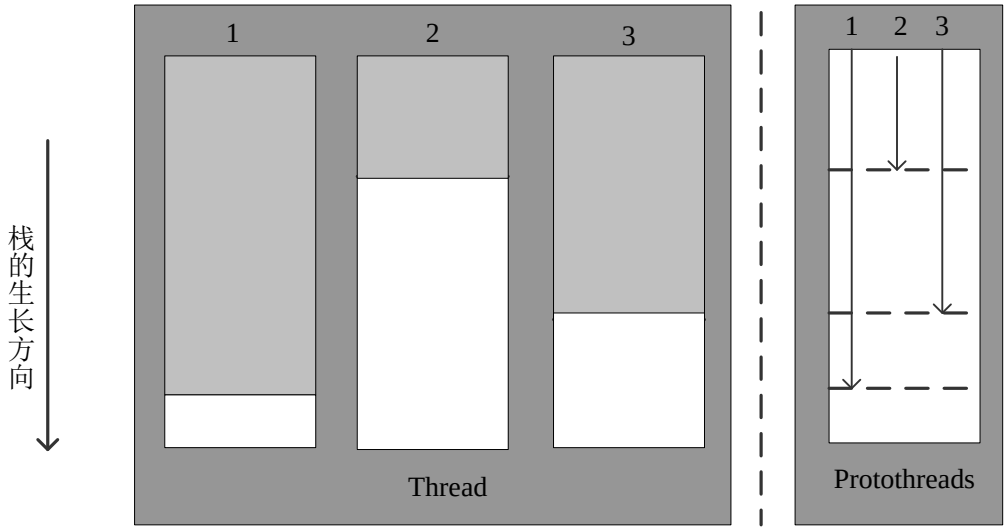

图 2-2 Thread 与 Protothreads 栈使用对比示意图

从图可以看出，原本需要 3 个栈的 Thread 机制，在 Protothreads 只需要一个栈，当进程数量很多的时候，由栈空间省下来的内存是相当可观的。保存程序断点在传统的 Thread 机制很简单，只需保存在私有的栈，然而对于仅有公共栈是行不通的，因为保存在全局栈的变量随时会被覆盖。Protothreads 很巧妙地解决了这个问题，即用一个两字节静态变量存储被中断的行数，因为静态变量不从栈上分配空间，所以即使有任务切换也不会影响到该变量，从而达到保存断点的目的。下一次该进程获得执行权时，进入函数体后就通过 switch 语句跳转到上一次被中断的地方。

①保存断点

保存断点是通过保存行数来完成的，在被中断的地方插入编译器关键字__LINE__，编译器便自动记录所中断的行数。展开那些具有中断功能的宏，可以发现最后保存行数是宏 LC_SET，取宏 PROCESS_WAIT_EVENT()为例，将其展开得到如下代码：

```
#define PROCESS_WAIT_EVENT() PROCESS_YIELD()
#define PROCESS_YIELD() PT_YIELD(process_pt)
#define PT_YIELD(pt) \
do{ \
   PT_YIELD_FLAG = 0; \
   LC_SET((pt)->lc); \
   if(PT_YIELD_FLAG == 0) \
   {
        return PT_YIELDED; \
   } \
}while(0)

#define LC_SET(s) s = __LINE__; case __LINE__:   //保存程序断点，下次再运行,
                                                  //该进程直接跳到__LINE__
```

值得一提的是，宏 LC_SET 展开还包含语句 case __LINE__，用于下次恢复断点，即下次通过 switch 语言便可跳转到 case 的下一语句。

②恢复断点

被中断程序再次获得执行权时，便从该进程的函数执行体进入，按照 Contiki 的编程规范，函数体第一条语句便是 PROCESS_BEGIN 宏，该宏包含一条 switch 语句，用于跳转到上一次被中断的行，从而恢复执行，宏 PROCESS_BEGIN 展开的源代码如下：

```
#define PROCESS_BEGIN() PT_BEGIN(process_pt)
#define PT_BEGIN(pt) { char PT_YIELD_FLAG = 1; LC_RESUME((pt)->lc)
#define LC_RESUME(s) switch(s) { case 0:      //switch 语言跳转到被中断的行
```

(2)进程控制块

和 Linux 一样，Contiki 也用一个结构体描述整个进程的细节，所不同的是，Contiki 进程控制块要简单得多。用链表将系统所有进程组织起来，如图 2-3 所示(将 PT_THREAD 宏展开)。

Contiki 系统定义一个全局变量 process_list 作为进程链表的头，还定义了一个全局变量 process_current 用于指向当前进程。成员变量 next 指向下一个进程，最后一进程的 next 指向空。name 是进程的名称，可以将系统配置成(定义变量 PROCESS_CONF_NO_PROCESS_NAMES 为 0)没有进程名称，此时 name 为空字

符串。变量 state 表示进程的状态，共 3 种，即 PROCESS_STATE_RUNNING、PROCESS_STATE_CALLED、PROCESS_STATE_NONE。变量 needspoll 标识进程优先级，只有两个值 0 和 1，needspoll 为 1 表示进程具有更高的优先级。

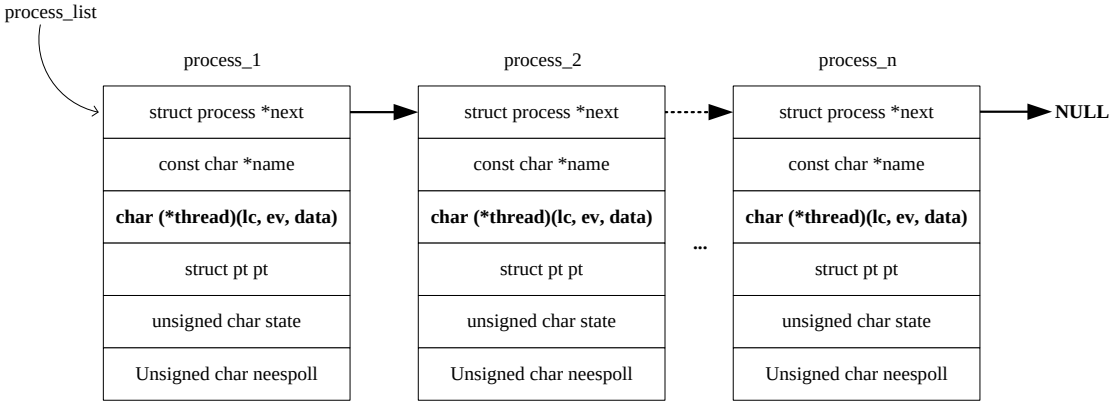

图 2-3 Contiki 进程链表 process_list 示意图

①成员变量 thread

进程的执行体，即进程执行实际上是运行该函数。在实际的进程结构体代码中，该变量由宏 PT_THREAD 封装，展开即为一个函数指针，关键源代码如下：

```
PT_THREAD((*thread)(struct pt *, process_event_t, process_data_t));
#define PT_THREAD(name_args) char name_args

/***宏展开***/
char (*thread)(struct pt *, process_event_t, process_data_t);
```

②成员变量 pt

正如上文所述的一样，Contiki 进程是基于 Protothreads，所以进程控制块需要有个变量记录被中断的行数。结构体 pt 只有一个成员变量 lc(无符号短整型)，可以将 pt 简单理解成保存行数的，相关源代码如下：

```
struct pt
{
   lc_t lc;
};
typedef unsigned short lc_t;
```

(3)进程调度

Contiki 只有两种优先级，用进程控制块中变量 needspoll 标识，默认情况是 0，即普通优先级。想要将某进程设为更高优先级，可以在创建之初指定其 needspoll 为 1，或者运行过程中通过设置该变量动态提升其优先级。实际的调度中，会先运行有高优先级的进程，而后再去处理一个事件，随后又运行所有高优先级的进程。通过遍历整个进程链表，将 needspoll 为 1 的进程投入运行，关键代码如下：

```
/***do_poll()关键代码，由 process_run 调用***/
for(p = process_list; p != NULL; p = p->next)                    //遍历进程链表
{
   if(p->needspoll)
   {
      p->state = PROCESS_STATE_RUNNING;                          //设置进程状态
      p->needspoll = 0;
      call_process(p, PROCESS_EVENT_POLL, NULL);                 //将进程投入运行
   }
}
```

以上是进程的总体调度，具体到单个进程，成员变量 state 标识着进程的状态，共有三个状态 PROCESS_STATE_RUNNING、PROCESS_STATE_CALLED、PROCESS_STATE_NONE。Contiki 进程状态转换如图 2-4 所示。

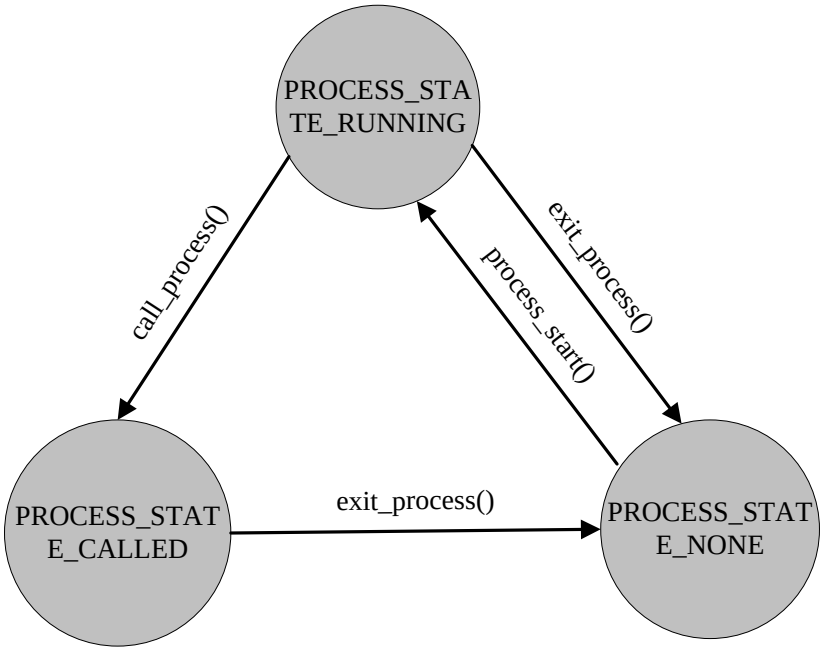

图 2-4 Contiki 进程状态转换图

创建进程(还未投入运行)以及进程退出(但此时还没从进程链表删除)，进程状态都为 PROCESS_STATE_NONE。通过进程启动函数 process_start 将新创建的进程投入运行队列(但未必有执行权)，真正获得执行权的进程状态为 PROCESS_STATE_CALLED，处在运行队列的进程(包括正在运行和等待运行)可以调用 exit_process 函数退出。

①进程初始化

系统启动，需要先将进程初始化，主要工作是事件队列和进程链表初始化(进程链表头与当前进程皆设为空)。process_init 源代码如下：

```
void process_init(void)
{
   /***初始化事件队列***/
   lastevent = PROCESS_EVENT_MAX;
   nevents = fevent = 0;
   process_maxevents = 0;

   /***初始化进程链表***/
   process_current = process_list = NULL;
}
```

②创建进程

创建进程实际上是定义进程控制块和声明进程执行体的函数。宏 PROCESS 的功能包括定义一个结构体，声明进程执行体函数，关键源代码如下(假设进程名称为 Hello world)：

```
PROCESS(hello_world_process, "Hello world");

/***PROCESS 宏展开***/
PROCESS_THREAD(name, ev, data); \
struct process name = { NULL, strname, process_thread_##name }

/***PROCESS_THREAD 宏展开***/
static PT_THREAD(process_thread_##name(struct pt *process_pt, process_event_t
ev, process_data_t data))
#define PT_THREAD(name_args) char name_args

/***将参数代入，PROCESS 宏最后展开结果***/
static char process_thread_hello_world_process(struct pt *process_pt,   \
  process_event_t ev, process_data_t data);
struct process hello_world_process =   \                    //声明进程执行体函数
{NULL, "Hello world", process_thread_hello_world_process };//定义进程结构体
```

可见，PROCESS 宏实际上声明一个函数并定义一个进程控制块，新创建的进程 next 指针指向空，进程名称为“Hello world”，进程执行体函数指针为 process_thread_hello_world_process，保存行数的 pt 为 0，状态为 0(即 PROCESS_STATE_NONE)，优先级变量 needspoll 也为 0(即普通优先级)。

PROCESS 定义结构体并声明函数，还需定义该函数，通过宏 PROCESS_THREAD 实现。值得注意的是，尽管 PROCESS 宏展开包含了宏 PROCESS_THREAD，用于声明函数，而这里却是定义函数，区别在于前者宏展开后面加了个分号。定义函数框架代码如下：

```
PROCESS_THREAD(hello_world_process, ev, data)
//static char process_thread_hello_world_process(struct pt *process_pt,
//                                               process_event_t ev, process_data_t data)
{
  PROCESS_BEGIN();        //函数开头必须有
  /***代码放在这***/
  PROCESS_END();          //函数末尾必须有
}
```

欲实现的代码必须放在宏 PROCESS_BEGIN 与 PROCESS_END 之间，这是因为这两个宏用于辅助保存断点信息(即行数)，宏 PROCESS_BEGIN 包含 switch(process_pt->lc)语句，这样被中断的进程再次获得执行便可通过 switch 语句跳转到相应的__LINE__，即被中断的行。

③启动进程

函数 process_start 用于启动一个进程，首先进行参数验证，即判断该进程是否已在进程链表中，若是则将进程加到链表，并给该进程触发一个初始化事件 PROCESS_EVENT_INIT。process_start 函数流程如图 2-5 所示。

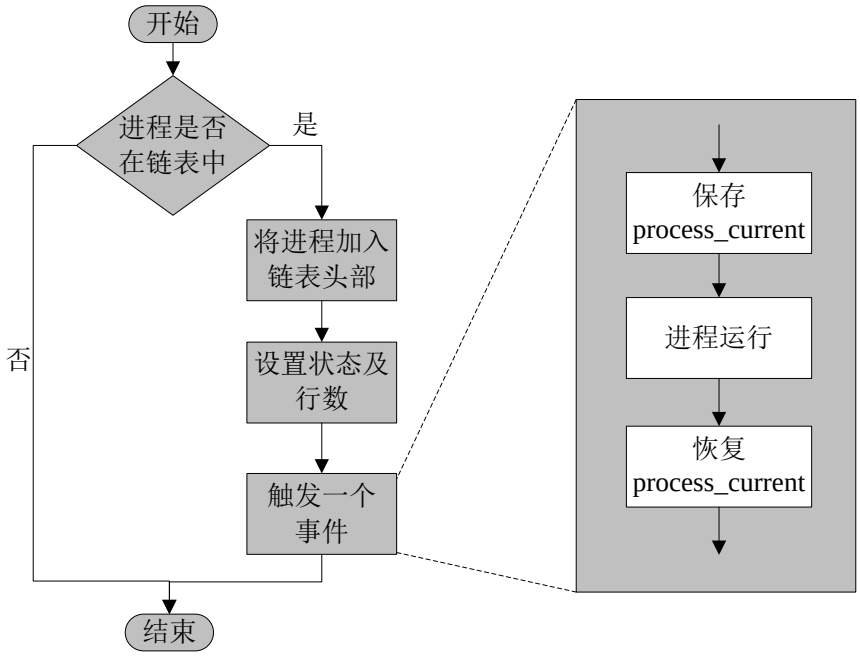

图 2-5 函数 process_start 流程图

process_start 将进程状态设为 PROCESS_STATE_RUNNING，并调用 PT_INIT 宏将保存断点的变量设为 0(即行数为 0)。调用 process_post_synch 给进程触发一个同步事件，事件为 PROCESS_EVENT_INIT。考虑到进程运行过程中可能被打断(比如中断)，所以进程运行前保存当前进程指针保存起来，执行完再恢复(保存断点、

恢复断点)。进程运行由 call_process 函数完成，函数流程如图 2-6 所示。

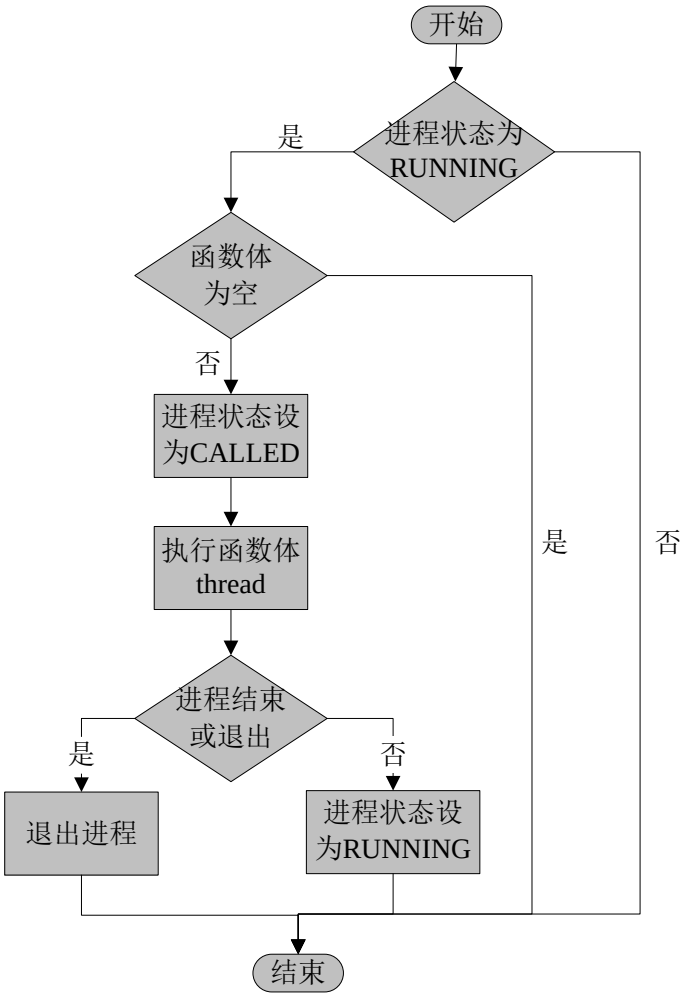

图 2-6 call_process 函数流程图

call_process 首先进行参数验证，即进程处于运行状态(退出但尚未删除的进程，进程状态为 PROCESS_STATE_NONE)并且进程的函数体不为空，接着将进程状态设为 PROCESS_STATE_CALLED，表示该进程拥有执行权。接下来，运行该进程函数体，根据返回值判断进程是否结束(主动的)或者退出(被动的)，若是则调用 exit_process 将退出进程，否则将该进程状态设为 PROCESS_STATE_RUNNING，继续放在进程链表。

④进程退出

进程运行完或者收到退出事件都会导致进程退出。根据 Contiki 编程规则，进程函数体最后一条语句是 PROCESS_END()，该宏包含语句 return PT_ENDED，表示进程运行完毕。系统处理事件时(事件绑定进程，事实上执行进程函数体)，倘若该进程恰好收到退出事件，thread 便返回 PT_EXITED，进程主动退出。还有就是

给该进程传递退出事件 PROCESS_EVENT_EXIT 也会导致进程退出。进程退出函数 exit_process 流程如图 2-7 所示。

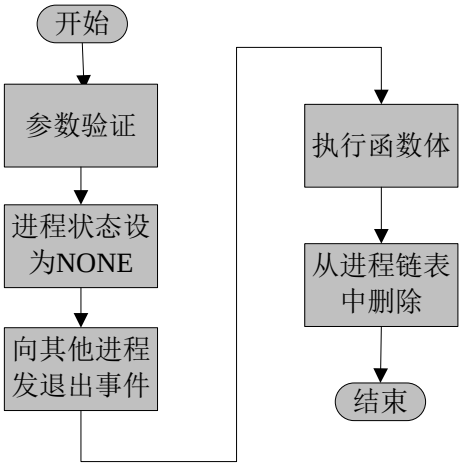

图 2-7 exit_process 函数流程图

进程退出函数 exit_process 首先对传进来的进程 p 进行参数验证，确保该进程在进程链表中并且进程状态为 PROCESS_STATE_CALLED/RUNNING(即不能是 NONE)，接着将进程状态设为 NONE。随后，向进程链表的所有其他进程触发退出事件 PROCESS_EVENT_EXITED，此时其他进程依次执行处理该事件，其中很重要一部分是取消与该进程的关联。进程执行函数体 thread 进行善后工作，最后将该进程从进程链表删除。

2.1.3 事件管理

事件驱动机制广泛应用于嵌入式系统，类似于中断机制，当有事件到来时(如按键、数据到达)，系统响应并处理该事件。相对于轮询机制，事件机制优势很明显，尤其是低功耗(系统处于休眠，当有事件到达时才被唤醒)和 MCU 利用率高。

Contiki 将事件机制融入 Protothreads，每个事件绑定一个进程(广播事件例外)，进程间的消息传递也通过事件传递。用无符号字符型来标识事件，事件结构体 event_data 定义如下：

```
struct event_data
{
   process_event_t ev;
   process_data_t data;
   struct process *p;
};
typedef unsigned char process_event_t;
typedef void *process_data_t;
```

用无符号字符型标识一个事件，Contiki 定义了 10 个事件(0x80~0x8A)，其他的供用户使用。每个事件绑定一个进程，如果 p 为 NULL，表示该事件绑定所有进程(即广播事件 PROCESS_BROADCAST)。除此之外，事件可以携带数据 data，可以利用这点实现进程间的通信(向另一进程传递带数据的事件)。

Contiki 用一个全局的静态数组存放事件，这意味着事件数目在系统运行之前就要指定(用户可以通过 PROCESS_CONF_NUMEVENTS 自选配置大小)，通过数组下标可以快速访问事件。系统还定义另两个全局静态变量 nevents 和 fevent，分别用于记录未处理事件总数及下一个待处理的位置。事件被逻辑组织成环形队列，存储在数组里，如图 2-8 所示。

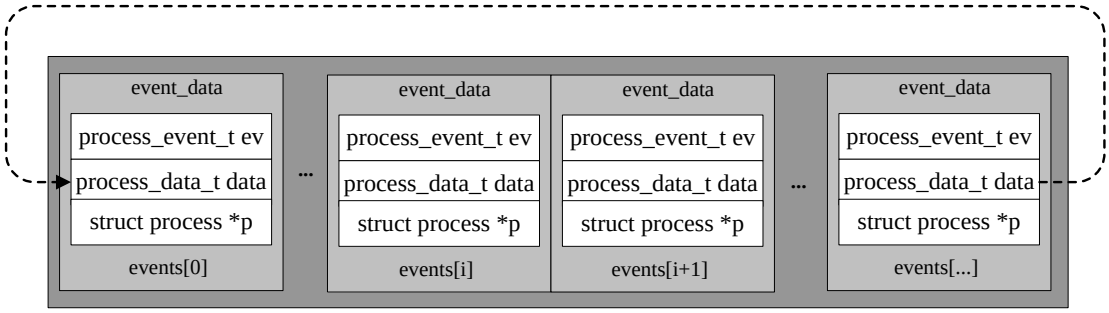

图 2-8 Contiki 事件队列示意图

可见对于 Contiki 系统而言，事件并没有优先级之分，而是先到先服务的策略，全局变量 fevent 记录了下一次待处理的事件。

(1)事件产生

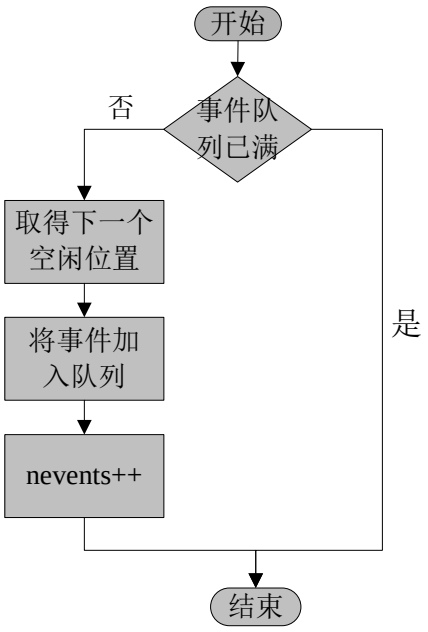

图 2-9 process_post 函数流程图

Conitki 有两种方式产生事件，即同步和异步。同步事件通过 process_post_synch 函数产生，事件触发后直接处理(调用 call_process 函数)。而异步事件产生是由 process_post 产生，并没有及时处理，而是放入事件队列等待处理(由 process_run 函数调用 do_event 处理)，process_post 函数流程如图 2-9 所示。

process_post 首先判断事件队列是否已满，若满返回错误，否则取得下一个空闲位置(因为事件队列被组织成环形，需做余操作)，而后设置该事件并将未处理事件总数加 1。

(2)事件调度

事件没有优先级，采用先到先服务策略，每一次系统轮询(函数 process_run)调用 do_event 只处理一个事件，do_event 函数流程如图 2-10 所示。

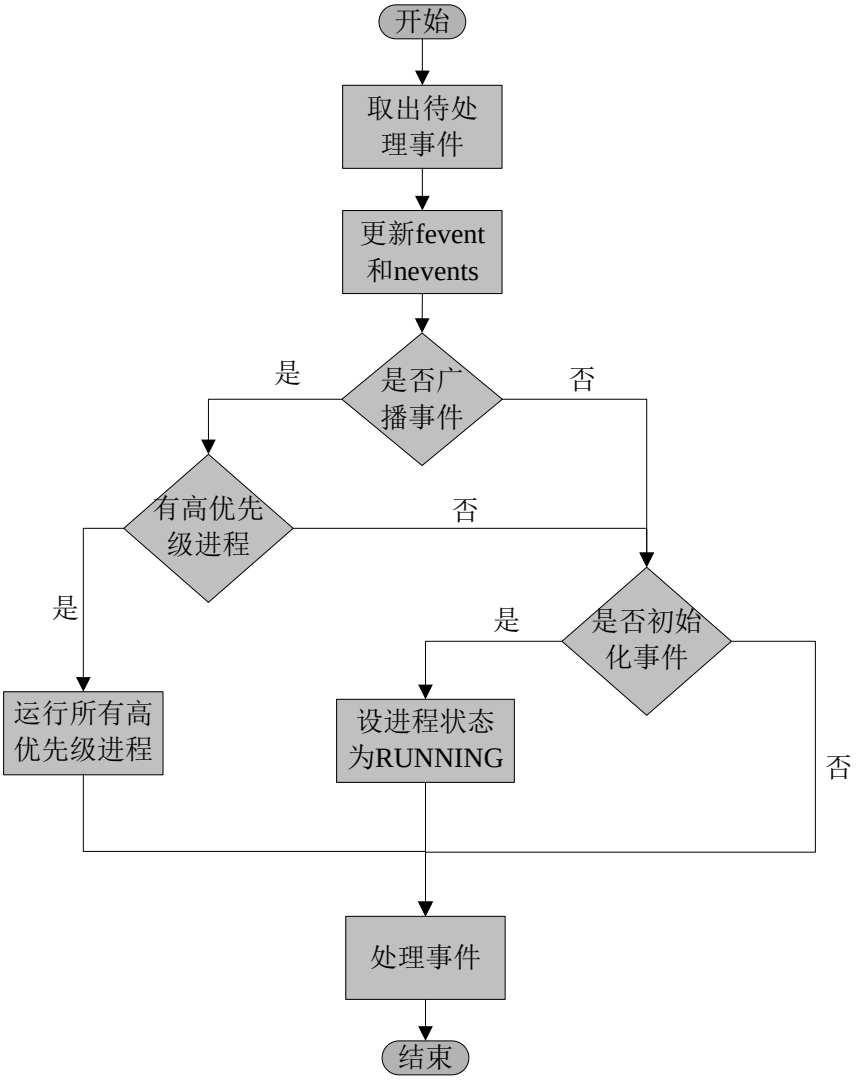

图 2-10 do_event 函数流程图

do_event 首先取出该事件(即将事件的值复制到一个新变量)，更新总的未处理事件总数及下一个待处理事件的数组下标(环形队列，需要取余操作)。接着判断事件是否为广播事件 PROCESS_BROADCAST，若是则先运行高优先级的进程，这样做可以让系统更具实时性(处理广播事件可能需要更多的时间)。而后再去处理事件(调用 call_process 函数)。如果事件是初始化事件 PROCESS_EVENT_INIT (创建进程的时候会触发此事件)，需要将进程状态设为 PROCESS_STATE_RUNNING。

(3)事件处理

事件绑定相应的进程(普通事件绑定一个进程，广播事件绑定所有的进程)，具体的事件处理由进程定义(在 thread 函数体中实现)。处理事件时，将事件作为参数传给绑定的进程，进程获得执行权后(call_process 调用 tread 函数)，进行相应处理。关键代码如下：

```
ret = p->thread(&p->pt, ev, data);
```

(4)定时器

Contiki 内核是基于事件驱动和 Protothreads 机制，事件既可以是外部事件(比如按键，数据到达)，也可以是内部事件(如时钟中断)。定时器的重要性不言而喻，Contiki 提供了 5 种定时器模型，即 timer(描述一段时间，以系统时钟嘀嗒数为单位)、stimer(描述一段时间，以秒为单位)、ctime(定时器到期，调用某函数，用于 Rime 协议栈)、etime(定时器到期，触发一个事件)、rtimer(实时定时器，在一个精确的时间调用函数)。

定时器的重要性不言而喻，而 etimer 是 Contiki 应用最广泛的一类，因此 Contiki 设计一个独立的系统进程 etimer_process 管理系统的所有 etimer。

①etimer 组织结构

etimer 作为一类特殊事件存在，也是跟进程绑定。除此之外，还需变量描述定时器属性，etimer 结构体定义如下：

```
struct etimer
{
  struct timer timer;       //包含起始时刻和间隔两成员变量
  struct etimer *next;      //指向下一个 etimer
  struct process *p;
};
```

成员变量 timer 用于描述定时器属性，包含起始时刻和间隔，将起始时刻与间隔相加与当前时钟对比，便可知道定时器是否到期。变量 p 指向所绑定的进程(p 为 NULL 则表示该定时器与所有进程绑定)。成员变量 next 指向下一个 etimer，系统所有 etimer 被链接成一个链表，如图 2-11 所示。

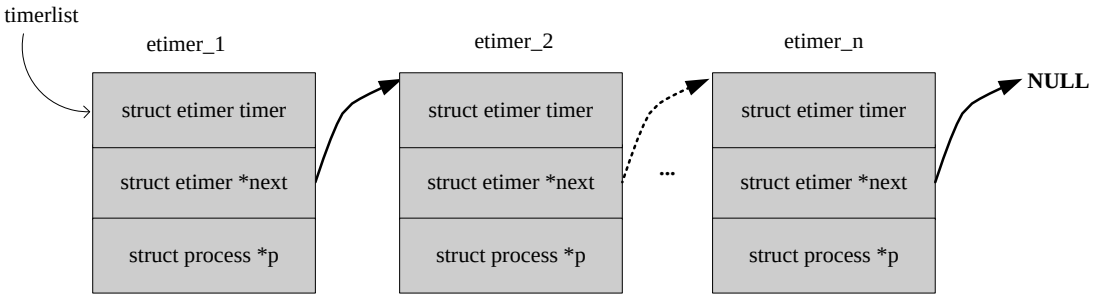

图 2-11 timer 链表 timer_list 示意图

②添加 etimer

欲添加定时器 etimer，首先定义一个 etimer 结构体，并调用 etimer_set 函数将 etimer 添加到 timerlist， etimer_set 函数流程如图 2-12 所示。

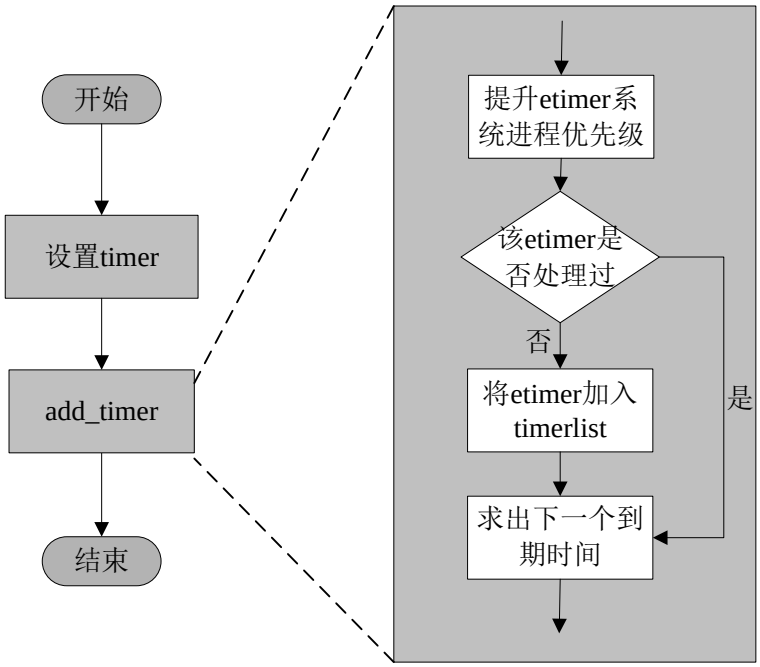

图 2-12 etimer_set 流程图

etimer_set 首先设置 etimer 成员变量 timer 的值(由 timer_set 函数完成)，即用当前时间初始化 start，并设置间隔 interval，接着调用 add_timer 函数，该函数首先将管理 etimer 系统进程 etimer_process 优先级提升，以便定时器到期可以得到更快的响应。接着确保欲加入的 etimer 不在 timerlist 中(通过遍历 timerlist 实现)，若该 etimer 已经在 etimer 链表，则无须将 etimer 加入链表，仅更新时间。否则将该 etimer 插入到 timerlist 链表头位置，并更新时间(update_time)。这里的更新时间是求出还需要多长 next_expiration(全局静态变量)时间，etimer 链表就会有 etimer 到期。

③etimer 管理

Contiki 用一个系统进程 etimer_process 管理所有 etimer 定时器。进程退出时，向所有进程传递事件 PROCESS_EVENT_EXITED，当然也包括 etimer 系统进程 etimer_process。当 etimer_process 拥有执行权的时候，便查看是否有相应的 etimer 绑定到该进程，若有则就删除这些关联的 etimer。除此之外，etimer_process 还会处理到期的 etimer 定时器，etimer_process 的 thread 函数流程如图 2-13 所示。

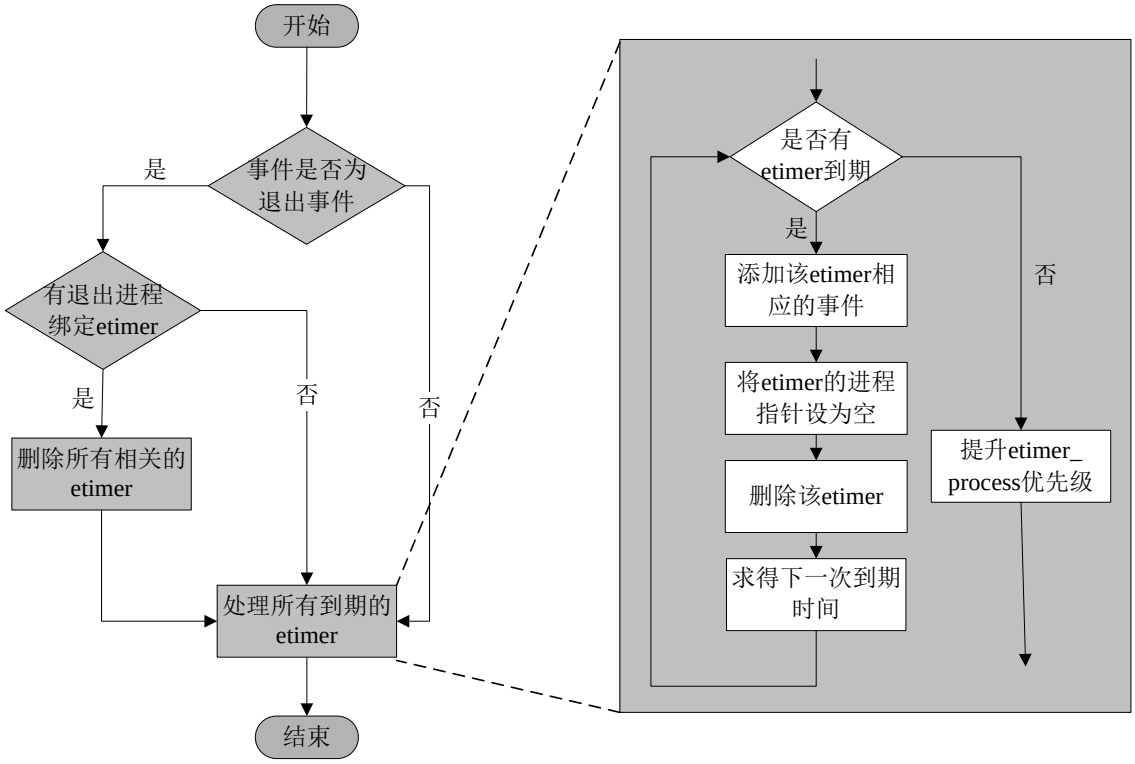

图 2-13 etimer_process 的 thread 函数流程图

etimer_process 获得执行权后，首先判断传递的事件是否为退出事件，若是则先遍历整个 timerlist，将与该进程(通过参数 data 传递)相关联的 etimer 从 timerlist 删除，而后转去处理所有到期的 etimer。通过遍历整个 etimer 查看到期的 etimer，若有到期，向绑定的进程传递事件 PROCESS_EVENT_TIMER，并将 etimer 的进程指针设为空(事件已加入事件队列，定时器处理完毕)，接着删除该 etimer，求出下一次 etimer 到期时间，提升 etimer_process 优先级，而后继续检查是否还有 etimer 到期，若没有 etimer 到期，就退出。总之，遍历 timerlist，只要 etimer 到期，处理之后继续遍历整个链表，直到 timerlist 没有到期的 etimer 才退出。

2.2 Coffee 文件系统

Coffee[10]是 Contiki OS 自带的文件系统，Coffee 每个文件只使用少量 RAM，

这使得内存受限节点管理大文件或者大量文件成为可能。除此之外，Coffee 设计了一种叫微日志(micro log)文件，有效减少了 FLASH 擦除次数。关于 Coffee 详细描述见第三章。

2.3 动态加载

类似于 Linux，Contiki 核心之外的模块或者应用程序，可以通过动态加载机制进行加载、卸载、替换。Contiki 动态加载模块也是实现重编程技术的基础。详情见见第四章。

2.4 Rime 协议栈

传统的分层通信架构(communication architectures)很难满足资源受限的传感器网络，于是研究者转向跨层优化(比如将顶层数据聚合功能放在底层实现)，但这导致系统变得更脆弱以及难以控制(fragile and unmanageable systems)。因此，传统分层通信结构再次得到重视，同时研究发现，传统分层效率几乎可以与跨层优化相媲美[33]。基于此，Rime 也采用分层结构。

2.4.1 Rime 概述

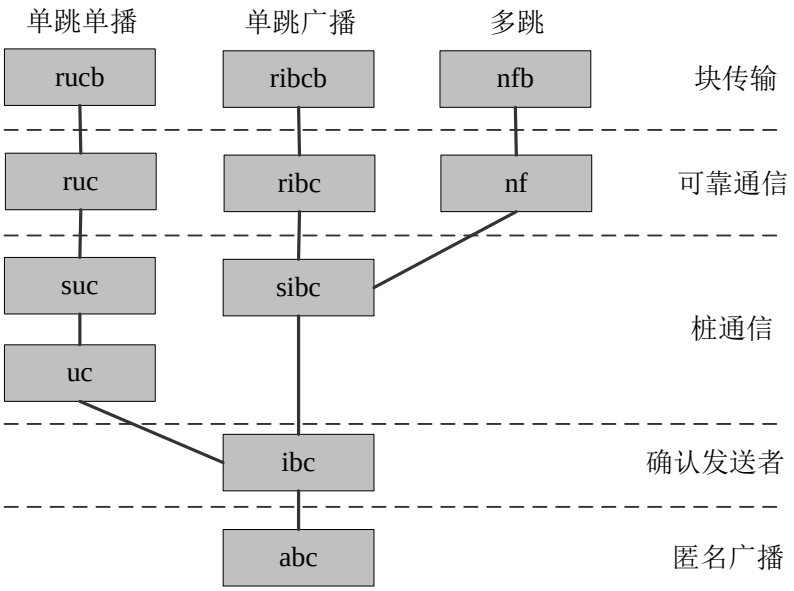

图 2-14 Rime 协议栈结构框图

Rime 是针对传感器网络轻量级、层次型协议栈，也是低功耗、无线网络协议栈，旨在简化传感器网络协议并实现代码重用，属于 Contiki 的一部分(Contiki 还支持 uIPv4、uIPv6、LwIP)。Rime 协议栈结构框图如图 2-14 所示。

上图中单跳单播各个缩写含义如下：

rucb

rucb 是单跳单播的最顶层，将数据以块为单位进行传输(Bulk transfer)。

ruc

ruc 是指 Reliable communication。可靠通信由两层实现：Stubborn transmission、Reliable transmission。该层主要实现确认和序列功能(acknowledgments and sequencing)。

suc

suc 是指 Stubborn transmission，是可靠通信的另一层。suc 这一层在给定的时间间隔不断地重发数据包，直到上层让其停止。为了防止无限重发，需要指定最大重发次数(maximum retransmission number)。

ibc

ibc 是指 identified sender best-effort broadcast，将上层的数据包添加一个发送者身份(sender identity)头部。

uc

uc 是指 unicast abstraction，将上层的数据包添加一个接收者头部。

abc

abc 是指 anonymous broadcast，匿名广播。即将数据包通过无线射频驱动(radio driver)发出去，接收来自无线射频驱动所有的包并交给上层。

2.4.2 连接建立与释放

(1)建立连接

使用 Rime 协议栈进行通信之前，需要建立连接。Rime 协议栈提供单跳单播、单跳广播、多跳三种功能。在此，仅介绍单跳单播(Single-hop unicast)连接建立过程。

建立连接的实质是保存该连接一些信息(如发送者、接收者)，Rime 协议栈用一系列结构体保存这些链接状态信息。Rime 每一层都有相应的连接结构体(以_conn 结尾)，上层嵌套下层，如下：

rucb_conn —> runicast_conn —> stunicast_conn —> unicast_conn —> broadcast_conn —> abc_conn

每个连接结构体都有相应的回调结构体(以_callbacks 后缀结尾)，该结构体的成员变量实为发送、接收函数指针。当接收一个数据报，会调用该结构体相应的函数。回调结构体层次如下：

rucb_callbacks —> runicast_callbacks —> stunicast_callbacks —> unicast_callbacks —> broadcast_callbacks —> abc_callbacks

综上，连接建立_open、连接结构体_conn、回调结构体_callbacks 间的关系如图 2-15 所示。

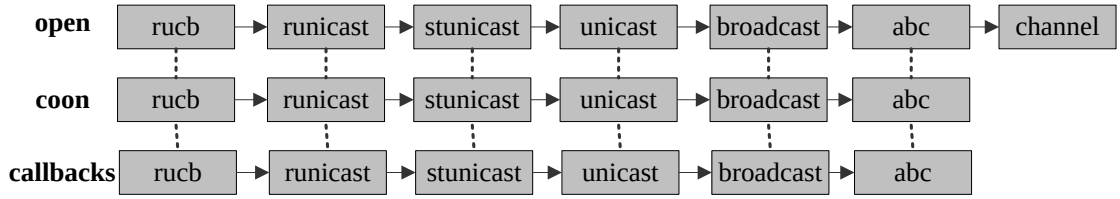

图 2-15 open、coon、callbacks 关系示意图

建立连接，实质是初始化结构体 rucb_conn 各个成员变量，结构体 rucb_conn 定义如下：

```
struct rucb_conn
{
   struct runicast_conn c;
   const struct rucb_callbacks *u;
   rimeaddr_t receiver, sender;
   uint16_t chunk;
   uint8_t last_seqno;
};
```

结构体 rucb_conn 各成员变量含义如下：

c

uc(unicast abstraction)将上层的数据包添加一个接收者头部传递给下一层，这里的 c 指的是下一层连接结构体。

u

结构体 rucb_callbacks 有 3 个函数指针成员变量写数据块 write_chunk、读数据块 read_chunk、超时 timedout，需要用户自己实现。

receiver、sender

用于标识接收者和发送者。这里的 receiver 是指目的节点的接收地址。

chunk

数据块数目。

last_seqno

用于标识数据包传输的结束标记(如果数据包太大，则需要将其划分成多个片段)。目的节点接收到数据时，判断其序列号是否等于最后一个序列号，若是则不再接收(数据包的所有分片都已接收，即完成本次数据包传输)。

(2)释放连接

数据通信完毕之后，需要释放连接，以供其他进程使用。关闭连接实质上是将相应的连接结构体从链表中删除。整个调用过程如下：

rucb_close —> runicast_close —> stunicast_close —> unicast_close —> broadcast_close —> abc_close —> channel_close —> list_remove

2.4.3 数据发送与接收

(1)数据发送

Rime 协议栈建立连接后，就可以进行通信了(发送、接收数据)，Rime 协议栈提供单跳单播、单跳广播、多跳三种功能。在此，仅介绍单跳单播(Single-hop unicast)发送数据的情型。

Rime 是层次型协议栈，整个数据发送过程是通过上层调用下层服务来完成的，具体如下：

rucb_send —> runicast_send —> stunicast_send_stubborn —> unicast_send —> broadcast_send —> abc_send —> rime_output —> NETSTACK_MAC.send

rucb 是块传输(Bulk transfer)层，可以理解成传输层，数据发送函数 rucb_send 源代码如下：

```
int rucb_send(struct rucb_conn *c, const rimeaddr_t *receiver)
{
   c->chunk = 0;
   read_data(c);
   rimeaddr_copy(&c->receiver, receiver);
   rimeaddr_copy(&c->sender, &rimeaddr_node_addr);
   runicast_send(&c->c, receiver, MAX_TRANSMISSIONS);
   return 0;
}
```

c->chunk 将数据块数目初始化为 0，read_data 进行一些 Rime 缓冲区初始化相关工作。rimeaddr_copy 函数设置接收者 receiver 和发送者 sender 的 Rime 地址，rimeaddr_node_addr 用于标识本节点的 Rime 地址。接下来，调用下一层的发送函

数 runicast_send 完成数据发送。

(2)数据接收

Rime 协议栈建立连接后，就可以调用数据接收函数 recv 接收数据，整个接收数据过程是通过上层调用下层服务来完成的，具体如下：

recv —> recv_from_stunicast —> recv_from_uc —> recv_from_broadcast —> recv_from_abc

函数 recv 首先判断该数据包是不是最后一个序列(数据包传输的结束标记)，若不是，将收到的数据写入物理存储介质。recv 函数流程如图 2-16 所示。

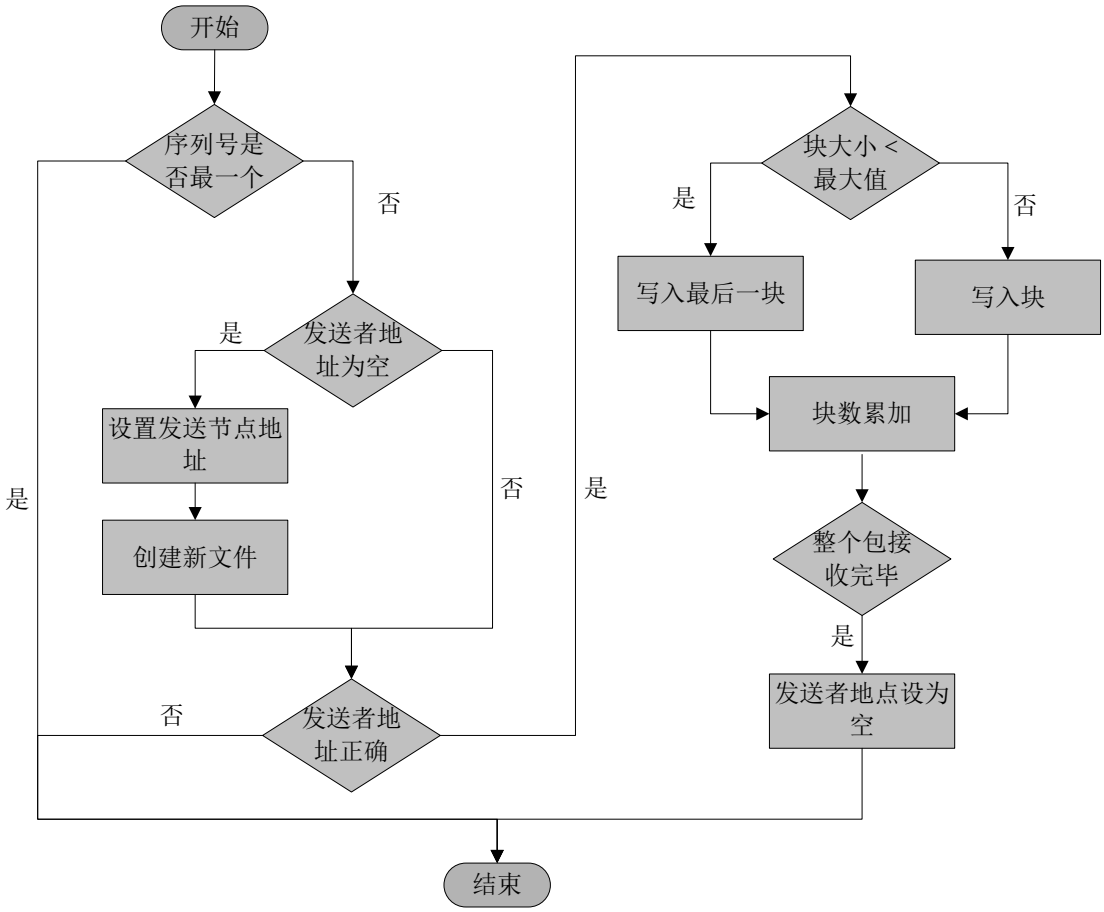

图 2-16 recv 函数流程图

函数 recv 首先判断接收到的包是不是最后一个序列，若是则返回(用最后序列号标识包传递完毕)，否则意味着还有数据要接收。接着判断发送者地址是否为空，若是，说明节点未曾接收该包的任何序列，则建立文件以存放数据。确保发送地址无误之后，若块小于块的最大值(RUCB_DATASIZE)，即这是数据包最后的一块，写入最后一块，否则正常写入这块的数据。把块的数目累加，接着判断该数据块是否是最后一块，若是则将发送者地址设为空，否则返回。

2.5 本章小结

本章首先阐述了选择 Contiki 作为研究对象的原因，而后深入浅出介绍 Contiki 操作系统内核和 Rime 协议栈的技术细节。首先，从全局视角描述了整个系统是如何运行的，即反复执行所有高优先级进程并逐一处理事件。接着，循序渐进对 Contiki 两个核心机制进行剖析。先是介绍了 Protothreads 原理以及如何做到减少内存使用，分析了进程控制块和进程调度，包括总体调度策略、进程状态转换、进程初始化、创建进程、启动进程、进程退出。随后介绍了事件机制，包括事件产生、调度、处理。除此之外，还分析了定时器这类特殊事件，包括创建定时器以及系统如何管理这些定时器。最后，剖析了 Rime 协议栈，先给出整体结构，而后分别介绍连接建立与释放、数据发送与接收。

第三章 WSNs 文件系统研究与改进①

FLASH 由于体积小、功耗低、抗震性好等优势，成为嵌入式系统首选的存储器。作为嵌入式系统的一个分支，无线传感器网络节点的存储器通常也是 FLASH，相对于桌面系统(通常是磁盘)，FLASH 的鲜明特性(如先擦后写，寿命)决定了设计 FLASH 文件系统需要考虑额外因素，如耗损平衡(FLASH 擦除块数限制)、坏块管理、掉电保护、垃圾回收、映射机制。除此之外，WSNs 节点资源受限(尤其是内存)更是给设计 WSNs 文件系统提出了挑战。

本文选取 Contiki 作为研究对象，Coffee[10]是 Contiki 自带的文件系统，Coffee 每个文件只使用少量 RAM，这使得内存受限节点管理大文件或者大量文件成为可能。Coffee 采用微日志(micro log)结构，即文件修改时，创建新的微日志文件，并链接到原始文件，从而减少 FLASH 擦除次数，提升性能。当预分配的空间不足时，Coffee 分配新的空间，并将原始文件和微日志文件一起拷入新文件，同时将原文件标记为失效，以便后续的垃圾回收。

3.1 Coffee 文件系统

3.1.1 文件组织

Coffee 采用简单顺序页结构，FLASH 上文件组织如图 3-1 所示，原始文件包括文件头、数据区、保留空间。当修改一个文件时，不是在原文件修改，而是创建一个微日志文件(micro log file)，并链接到原始文件[34]。

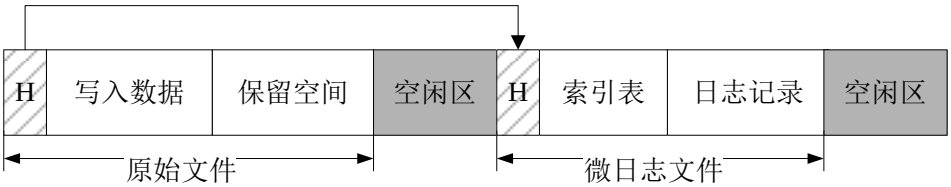

图 3-1 Coffee 文件物理组织示意图

描述文件的 3 个主要结构体是 file_header、file_desc、file。文件头 file_header 描述文件基本信息(在物理介质上)，文件 file 描述已打开文件的信息(在内存上，打

①本章部分成果已在《计算机技术与发展》杂志发表。

开文件时，从 file_header 获得这些信息)，文件描述符 file_desc 描述文件使用情况(与文件 file 一一对应)，文件描述符 file_desc、文件 file 分别组织成全局数组 coffee_fd_set[COFFEE_FD_SET_SIZE]、coffee_files[COFFEE_MAX_OPEN_FILES]. Coffee 文件系统在内存映射如图 3-2 所示。

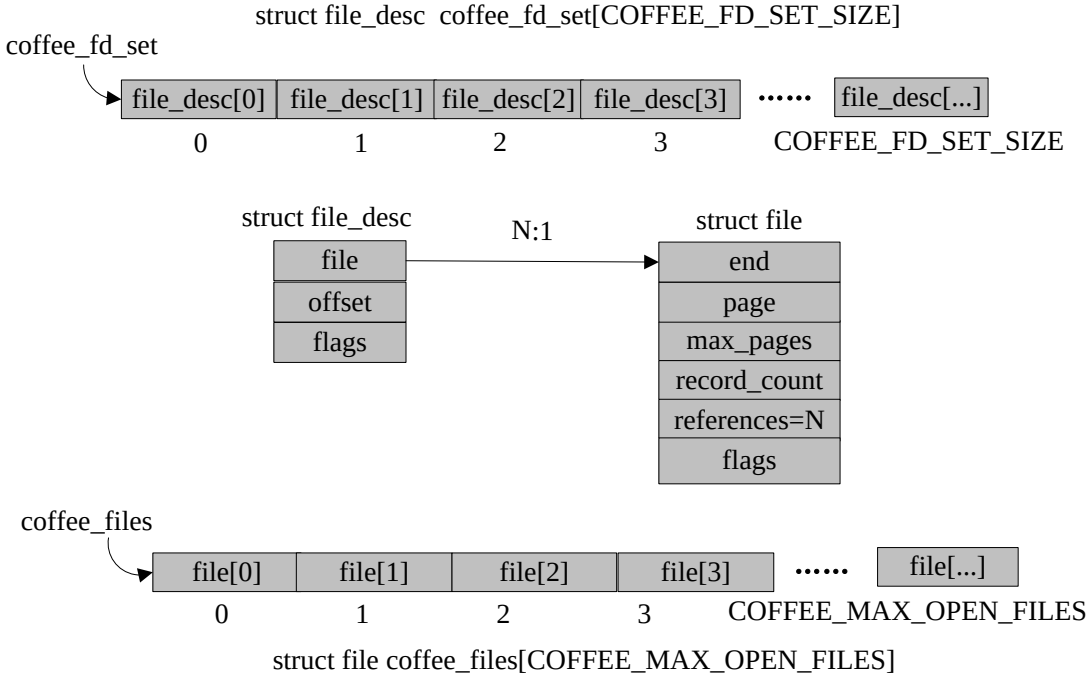

图 3-2 Coffee 文件内存映射示意图

(1)文件元数据 file_header

文件的元数据描述一个文件的基本信息(比如占用页情况)，Coffee 文件元数据由结构体 file_header 描述，file_header 位于文件(log file 或 micro log file)第一页的开始处。结构体 file_header 定义及各成员变量含义如下：

```
struct file_header
{
   coffee_page_t log_page;                    //指向微日志第一页(如果有微日志)
   uint16_t log_records;                      //微日志数量
   uint16_t log_record_size;                  //微日志大小
   coffee_page_t max_pages;                   //为文件保留的页面数
   uint8_t deprecated_eof_hint;               //指示文件最后一个字节
   uint8_t flags;                             //文件标志，反映文件状态
   char name[COFFEE_NAME_LENGTH]; //文件名
};
```

文件标志 flags 记录了整个文件的状态，只使用低 6 位，分别表示 ILMOAV，。V(valid)标记文件头是否完整的；A(allocated)标识空间是否已被分配，为 0 表示当

前页及所有保留页(直到下一个逻辑区的边界)是空闲的；M(modified)标识文件是否已被修改，为 1 表示微日志存在；L(log)标识微日志文件是否已被修改；I(isolated)标识是否为孤立页。

(2)文件描述符 file_desc

为提高性能，Coffee 用文件描述符 file_desc 缓存文件元数据 file_header，并与之建立一一映射关系，即一个文件描述符对应一个文件元数据，可以配置通过 COFFEE_FD_SET_SIZE 配置缓存文件元数据的数量，Coffee 定义了 file_desc 结构体类型的数组 coffee_fd_set 将这些文件描述符组织起来。结构体 file_desc 定义(去除非关键信息)及各成员变量含义如下：

```
struct file_desc
{
  cfs_offset_t offset;      //文件偏移量
  struct file *file;        //指向文件
  uint8_t flags;            //标志
};
```

偏移量 offset 记录光标位置(类似于 Word 光标所在位置)，标志 flags 描述文件的状态或读写权限，共 4 个，分别是 COFFEE_FD_FREE(该文件描述符未使用)、_READ(可读)、_WRITE(可写)、_APPEND (从文件末尾写)。

(3)文件 file

进程打开一个文件，系统会给该文件分配一个结构体 file，用来描述该文件基本信息(取自文件元数据)，结构体 file 及各成员变量含义如下：

```
struct file
{
  cfs_offset_t end;              //存放文件最后一个字节的偏移量
  coffee_page_t page;            //文件第一页
  coffee_page_t max_pages;   //文件保留的页面数
  int16_t record_count;          //微日志数量
  uint8_t references;            //文件被引用次数(被多少进程打开)
  uint8_t flags;                 //标志
};
```

file 标志 flags 只有两种取值：0 和 COFFEE_FILE_MODIFIED，即标识文件是否被修改。

3.1.2 实现细节

Coffee 是基于 FLASH 文件系统，被组织成层次型结构，支持目录和文件操作。

其总体框图如图 3-3 所示。

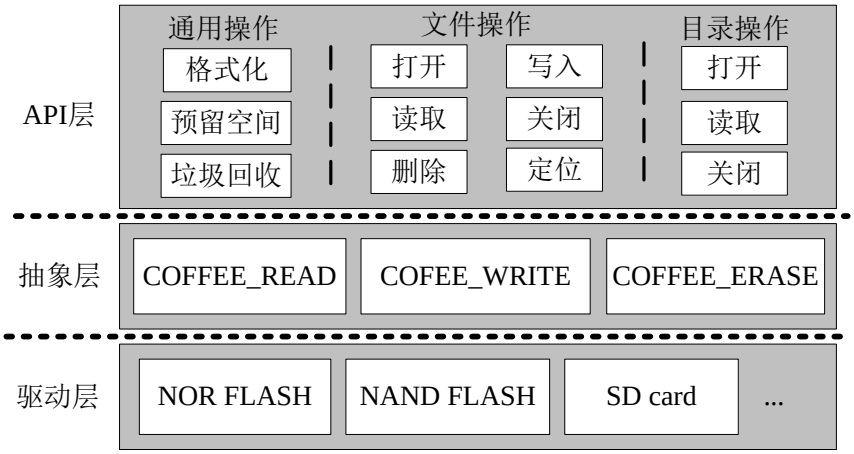

图 3-3 Coffee 文件系统总体框图

(1)格式化 cfs_coffee_format

第一次使用 Coffee 文件系统，需将其格式化，即将 Coffee 管理的 FLASH 区域全部写入“0”。Coffee 格式化是按区擦除，可以把区大小 COFFEE_SECTOR_SIZE 设成几倍的块大小(片外 FLASH，如 NAND FLASH，按块擦除)或者几倍的页大小(片上 FLASH，如 NOR FLASH，按页擦除或整个 FLASH 擦除)，这将有利于容量大的存储设备(加快顺序扫描速度)。

Coffee 文件系统格式化函数调用 COFFEE_ERASE 宏(需映射到 FLASH 具体的底层函数)将所有区擦除，并将结构体 protected_mem(Coffee 文件系统的一些全局变量)所有成员变量初始化为 0，函数 cfs_coffee_format 关键代码如下：

```
for(i = 0; i < COFFEE_SECTOR_COUNT; i++)
{
  COFFEE_ERASE(i);
}

memset(&protected_mem, 0, sizeof(protected_mem));
```

(2)创建打开 cfs_open

cfs_open 用于打开一个文件，成功打开返回文件描述符 fd，否则返回-1。如果文件不存在，则试图创建新文件。cfs_open 首先获得最小未使用的文件描述符 fd(若没有可用的 fd，则返回-1)，而后查看文件是否已缓存并且对应物理文件有效，若是则打开成功。若物理上没有该文件，则试图创建。值得注意的是，即便文件创建成功也可能返回-1(当 coffee_files[COFFEE_MAX_OPEN_FILES]数组没有可用项时)。cfs_open 函数流程如图 3-4 所示。

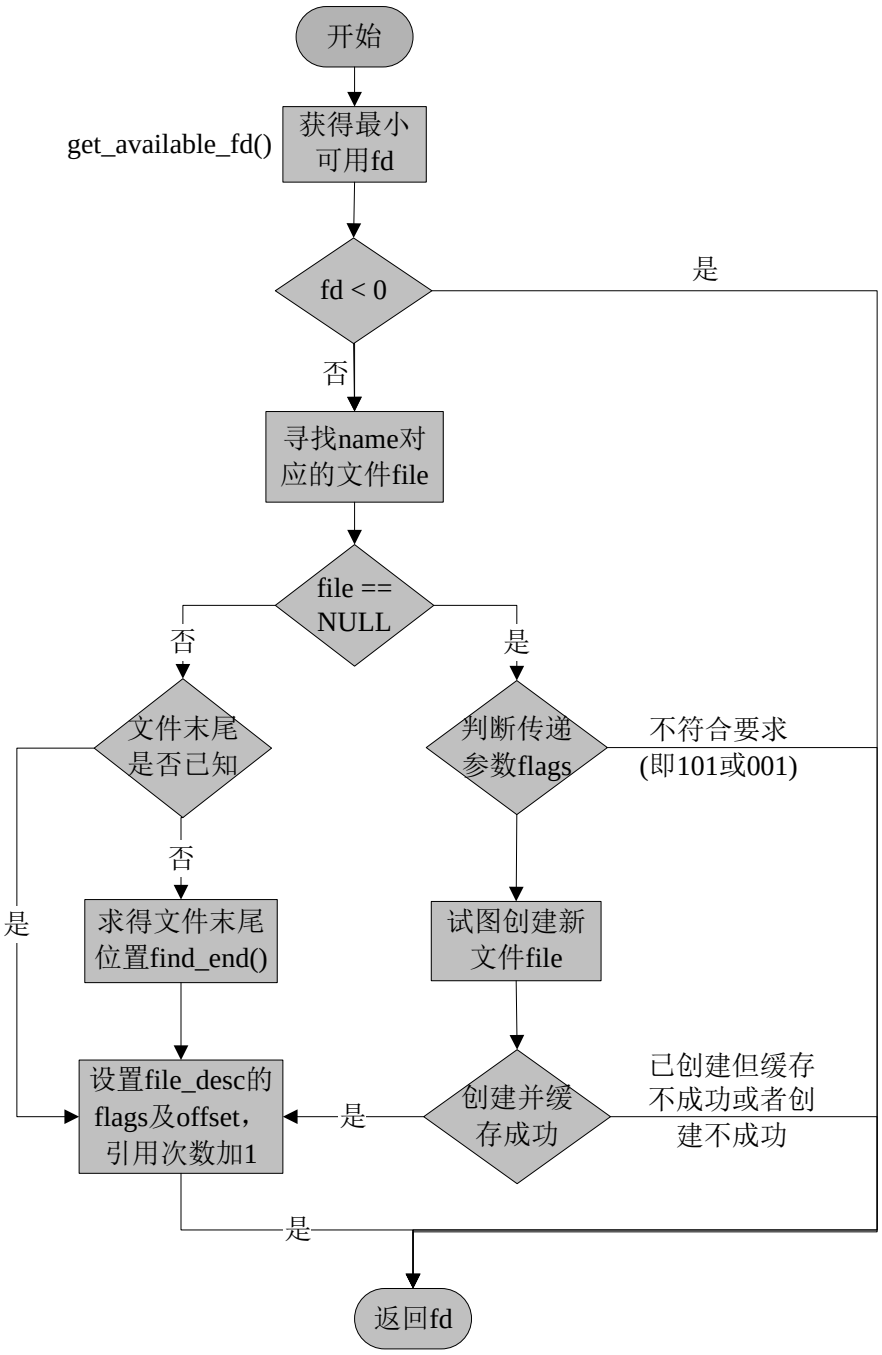

图 3-4 cfs_open 函数流程图

(3)读取 cfs_read

cfs_read 从打开文件 fd 读取数据，存放在 buf，读取的字节数是 size，返回实际读取的字节数。cfs_read 先进行参数验证(fd 有效性、读取权限、size)，而后再读取。读取时总体上分为两种情况：配置了微日志与没有配置微日志。前者，根据文件偏移量求得文件在 Coffee 文件系统的偏移量，直接读取。后者，根据文件

元数据计算微日志偏移量再读取。cfs_read 函数流程如图 3-5 所示。

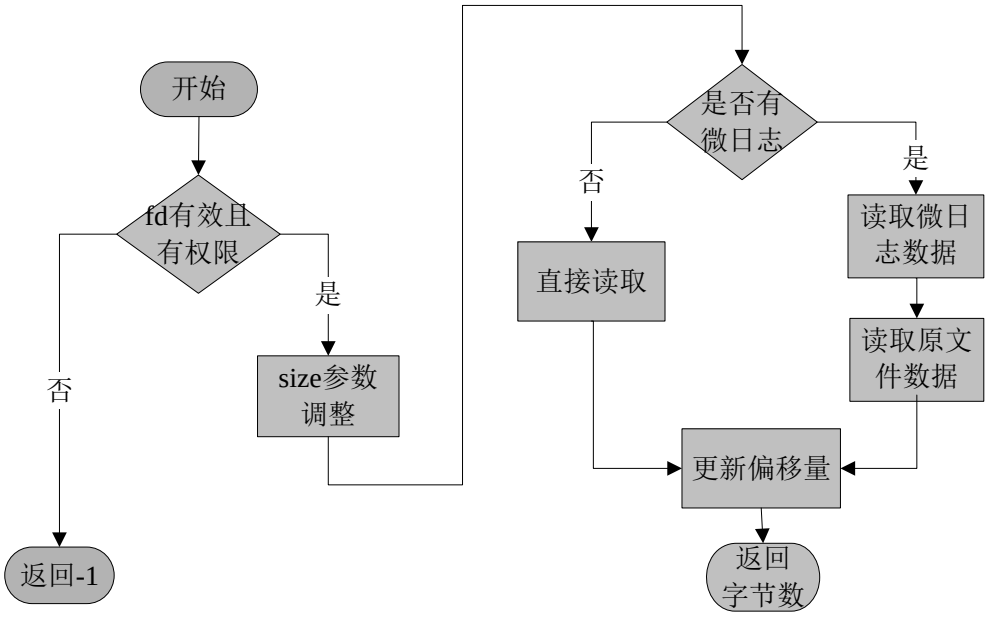

图 3-5 cfs_read 函数流程图

(3)写入 cfs_write

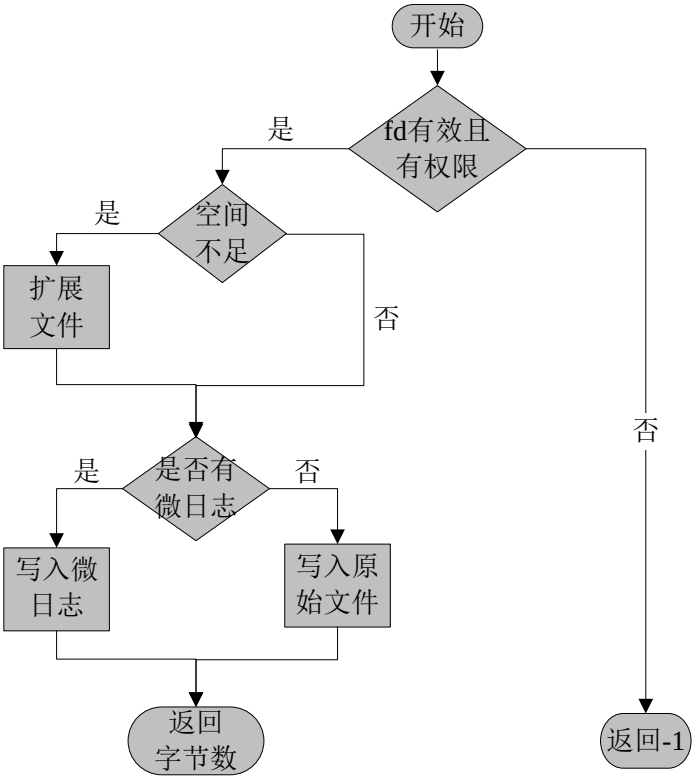

图 3-6 cfs_write 函数流程图

cfs_write 首先检查现有空间是否足以写入数据，若不足则扩展文件，即将原

始文件和日志文件拷贝到一个新的文件。而后进入实际写阶段，如果不存在微日志文件且原始文件足以容纳新的数据，则直接写入。否则创建新的微日志文件，将内容写入。cfs_write 函数流程如图 3-6 所示。

创建微日志文件 create_log 首先调整日志记录大小和日志记录数量，reserve 创建并加载微日志文件，若成功，对 file_header 及 file 相关成员变量进行一些设置，返回微日志文件的第一页，否则返回 INVALID_PAGE。成功创建后的文件总体示意图如图 3-7 所示。

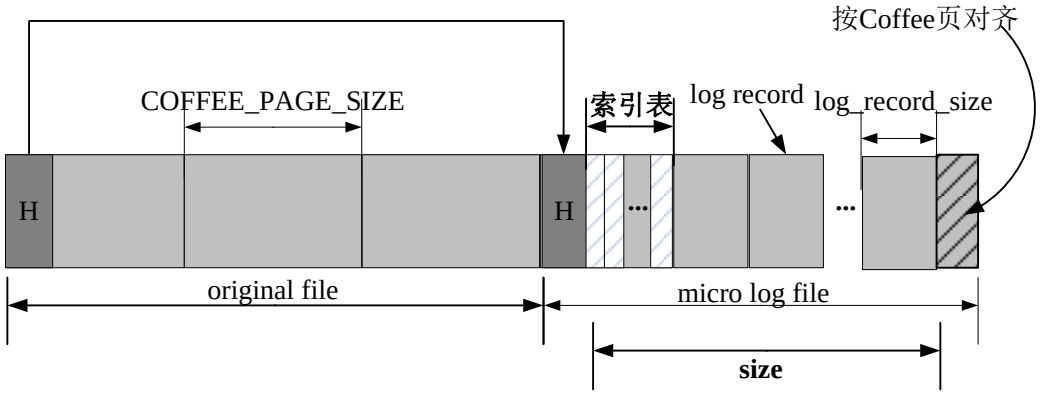

图 3-7 创建微日志文件示意图

(5)关闭 cfs_close

关闭不使用的文件，释放文件占有的内部资源(如 fd)，腾出缓存供他用。将 fd 对应的 file_desc 成员变量 flags 设为 COFFEE_FD_FREE，以便下次 cfs_open 函数可以找到可用的 fd，而后将与该 file_desc 关联文件 file 的引用次数减 1，最后将 file_desc 指向的 file 设为空。

(6)删除 cfs_remove

cfs_remove 首先找到 name 对应的文件 file 指针(find_file 函数)，先删除微日志文件(如果有的话)，将文件头 file_header 的 flags 中的 O 位(isolated，孤立)置 1，更新 file_header，将该文件关联的 file_desc 的 flags 设为 COFFEE_FD_FREE 和初始化 file 结构体，必要时(COFFEE_EXTENDED_WEAR_LEVELLING 为 0 且 gc_allowed 为 1)进行垃圾回收。cfs_remove 函数流程如图 3-8 所示。

值得一提的是，将文件头 file_header 的 flags 中的 O 位置 1，虽表示文件已删除，但此时，文件占用的空间还不能被使用(还没擦除)，只有垃圾回收处理后才能使用，而垃圾回收只有当存储空间不足时才执行。

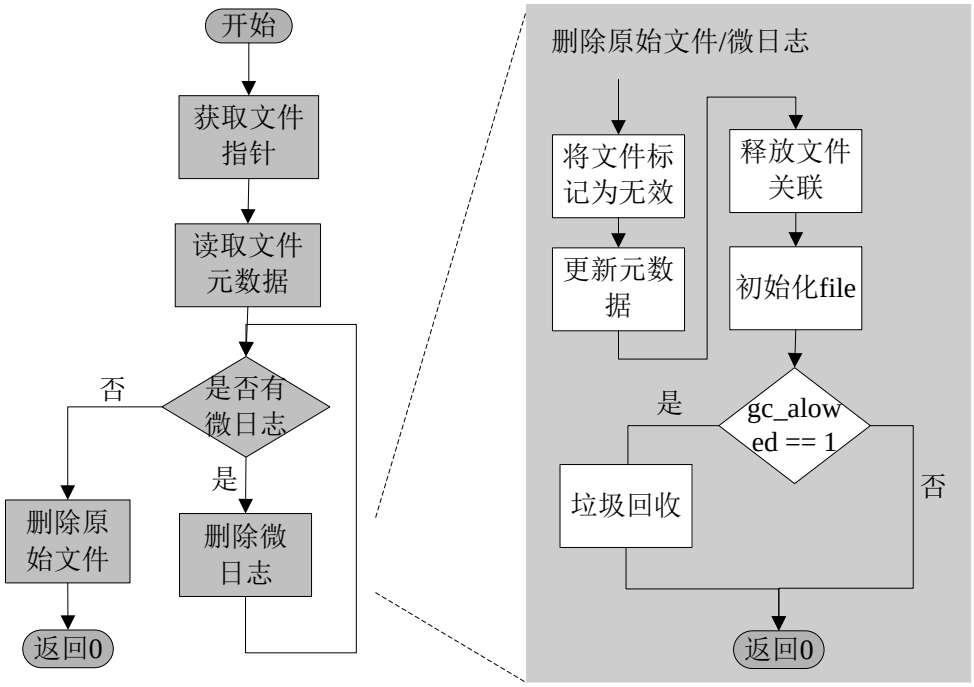

图 3-8 cfs_remove 函数流程图

(7)存储回收

FLASH 先擦后写的特性，也就不可能每次把失效的页及时擦除(效率低下，而且，对于按块擦除，可能有些页没失效，若此时擦除则需要转移这些未失效页的数据，才能擦除)。Coffee 只有在适当时候才擦除，即垃圾回收。Coffee 文件系统策略是文件删除并没有立即进行垃圾回收，而是待到没有可用空间时再回收，这样做有一个明显的缺点，垃圾回收会占用比较长的时间。

Coffee 提供了两种垃圾回收机制：GC_GREEDY 和 GC_RELUCTANT，前者垃圾回收过程中，擦除尽可能多的区(即贪心回收)，后者擦除一个区后就停止。

3.2 修复 Coffee 的 BUG

3.2.1 cfs_write 函数

Coffee 文件系统写入函数 cfs_write, 极端情况下，会把参数 size(写入字节数)直接返回而没有做任何写入操作。常规编程中，返回的 size 通常作为文件写入成功与否的判断，显然这是一个 BUG。去除与 BUG 无关的 cfs_write 源代码如下：

```
int cfs_write(int fd, const void *buf, unsigned size)
{
   ……
```

```
    #if COFFEE_IO_SEMANTICS
    ……
    #endif

    #if COFFEE_MICRO_LOGS
    ……
    #endif

    if(fdp->offset > file->end)
    {
        file->end = fdp->offset;
    }
    return size;
}
```

由此可见，假设系统没有配置 COFFEE_IO_SEMANTICS 且是原始文件(即没有微日志)，就会直接跳到 if(fdp->offset > file->end)执行，不论 offset 与 end 关系如何，程序原封不动地返回 size。修改这个 BUG，只需在程序开始前增加一个判断，排除这种极端情况：

```
if((~FILE_MODIFIED(file)) && COFFEE_IO_SEMANTICS)
{
    return    - 1;
}
```

3.2.2 宏 FD_WRITABLE

Coffee 的文件操作可以读、写、文件末尾写，习惯上(Linux 或 Windows 编程)，从文件末尾写的时候只需指定 APPEND 标志即可向文件写入。然而，对于 Coffee 文件系统，想要从文件末尾写需要同时指定 WRITE 和 APPEND 标记位，不符合开发者所熟悉的编程风格，带来诸多麻烦(由此引起错误甚至需要单步调试)。为修正这个问题，只需修改宏 FD_WRITABLE(判断文件是否有写的权限)的定义，如下：

```
/***修改前代码***/
#define FD_WRITABLE(fd) (coffee_fd_set[(fd)].flags & CFS_WRITE)

/***修改后代码***/
#define FD_WRITABLE(fd) \
(coffee_fd_set[(fd)].flags & CFS_WRITE & CFS_APPEND)
```

3.2.3 提升程序健壮性

据国外一项统计，编写函数时，返回相应的运行状态(即加入 return 语句)，可

以大大提升系统可靠性。这样做不仅可以有效防止程序跑飞，而且也便于调试。文件系统 Coffee 有不少函数没有返回值或者返回值不全面。本文对这些函数做了一一修改，以文件关闭函数 cfs_close 为例，加入相应返回语句后的源代码如下：

```
int cfs_close(int fd)                    //改为带有返回值
{
   if(FD_VALID(fd))
   {
      return 0;                          //添加返回值，操作成功返回 0
   }
   return -1;                            //添加返回值，操作失败返回-1
}
```

3.3 Coffee 改进及实现

尽管 Coffee 内存使用量少，也考虑了擦除平衡、擦写次数，但还是有改进的空间。

3.3.1 Coffee 格式化

FLASH 的物理特性(写入数据时，只能从“1”变为“0”,反之则不行)决定了修改文件时需要先擦除。官方发布的 Coffee 文件系统格式化是将整个 FLASH 写入“0”(先擦除后写入“0”)。这样的设计显然效率低下，格式化后的文件系统，当有新数据写入的时候，需要再次擦除才能写入。本文对其进行改进，Coffee 格式化直接是对整个 FLASH 擦除，下一次写入数据到 FLASH 就可以直接写入了，无须再擦除，这不仅提高了效率，而且减少擦除次数，也就延长了 FLASH 寿命。

Coffee 格式化函数会调用 Coffee 宏 COFFEE_ERASE 将所有区擦除，而宏 COFFEE_ERASE 会被映射到 FLASH 底层擦除函数。基于上述的改进，将 COFFEE_ERASE 直接映射到 FLASH 擦除函数 stm32_flash_erase，源代码如下(注：本文假定 Coffee 逻辑区等于 FLASH 页大小)：

```
void stm32_flash_erase(u8_t sector)
{
   //求得对应物理 FLASH 页首地址
   u32_t addr = COFFEE_START + (sector) *COFFEE_SECTOR_SIZE;
   FLASH_Unlock();              //打开 FLASH 擦除控制器
   FLASH_ErasePage(addr);       //擦除页面
   FLASH_Lock();                //关闭 FLASH 擦除控制器
}
```

经过上述改进，Coffee 格式化后的 FLASH 全部为“1”，包括文件元信息 file_header，这意味着文件的所有标志位都初始化为“1”。当需要修改标志位时，将相应标志位写入“0”，这样可以减少擦除次数(有些片内 FLASH 支持按位写)，从而延长 FLASH 寿命。Coffee 定义了一系列变量辅助标志位判断，这些变量定义也应做相应改变，修改后的源码如下：

```
#define HDR_FLAG_VALID          0xFE    //第 1 位为 0
#define HDR_FLAG_ALLOCATED      0xFD    //第 2 位为 0
#define HDR_FLAG_OBSOLETE       0xFB    //第 3 位为 0
#define HDR_FLAG_MODIFIED       0xF7    //第 4 位为 0
#define HDR_FLAG_LOG            0xEF    //第 5 位为 0
#define HDR_FLAG_ISOLATED       0xDF    //第 6 位为 0
```

Coffee 的其他函数也应按此逻辑进行相应修改，再此不一一赘述。

3.3.2 依 FLASH 类型定制

FLASH 分为 NOR 型和 NAND 型，前者通常按页擦除，按字节、字、双字甚至是位写，后者通常是按块擦除，按页写。基于此特性，便可以根据 FLASH 类型来定制文件系统，从而减少擦除次数。本文所用平台上的 FLASH 是片内的，将区大小固定为页大小。源代码如下：

```
#define   COFFEE_SECTOR_SIZE     FLASH_PAGE_SIZE
```

3.3.3 依采集数据特点定制

根据采集数据的频率、大小(文本、图像、音视频)来定制文件系统，可以将频率很快并且数据量很小的合并成单个文件，为采集音视频分配更多的页。同时，鉴于音视频极少修改，可以考虑不为其配置微日志文件。本研究隶属于实验室预研项目电力监测，因电力监测采集到的数据频率高、文件小，这意味着需要频繁更改原文件，所以配置微日志文件，并将微日志记录设大一些。相关源代码如下：

```
#define COFFEE_MICRO_LOGS 1              //配置微日志
#define COFFEE_LOG_TABLE_LIMIT 128   //微日志索引个数
#define COFFEE_LOG_SIZE 64               //微日志大小
```

3.3.4 文件元数据组织

Coffee 文件系统将元数据置于文件开始处，零散分布，不便于查找，而且元数据经常需要被修改，意味着频繁擦除。一种可行的改进办法是将元数据统一组

织起来，这不仅便于查找，也更具安全性(便于备份元数据)。将所有文件元数据存放在 Coffee 文件系统开头处，改进后的 Coffee 文件组织如图 3-9 所示。

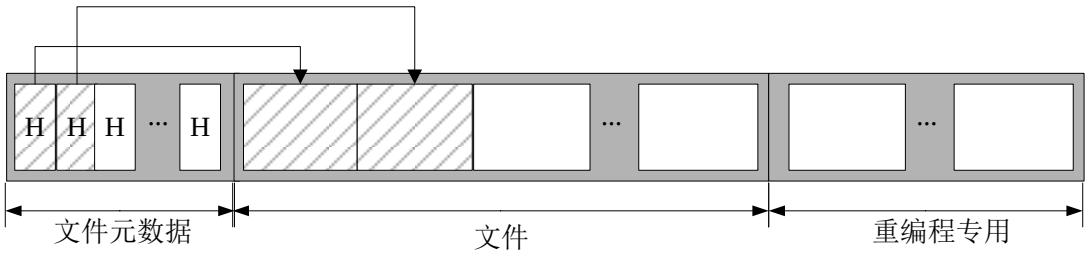

图 3-9 改进后的 Coffee 文件组织示意图

3.3.5 结合重编程优化

对节点更新时(重编程)，需要先将待更新的模块、镜像或者配置文件通过无线传感器网络传输到节点并存储，考虑到重编程数据的特殊性，在文件系统划出一块区域供其专用，通过变量 COFFEE_REPRO_START(新增的)可以快速找到重编程文件，修改变量如下：

```
#define COFFEE_HEADER_PAGES     \
COFFEE_HEADER_CONF_PAGES          //文件元数据页数，可配置
#define COFFEE_REPRO_PAGES        \
COFFEE_REPRO_CONF_PAGES           //重编程页数，可配置

#define COFFEE_PAGES \                    //Coffee 页数
((FLASH_PAGES*FLASH_PAGE_SIZE-(COFFEE_ADDRESS-FLASH_START))
/COFFEE_PAGE_SIZE)          -          COFFEE_REPRO_PAGES          -
COFFEE_HEADER_PAGES
#define COFFEE_START     (COFFEE_ADDRESS & ~(COFFEE_PAGE_SIZE-1))
#define COFFEE_SIZE      (COFFEE_PAGES*COFFEE_PAGE_SIZE)

#define COFFEE_REPRO_START \
(COFFEE_START + COFFEE_PAGES*COFFEE_PAGE_SIZE) //重编程开始位置
```

3.4 本章小结

本章剖析了 Contiki 自带的文件系统 Coffee，并在此基础上提出若干改进方法。首先，深入源码分析 Coffee，从全局着眼介绍 Coffee 文件系统是如何组织文件，包括物理介质 FLASH 上的组织以及在内存上的组织。紧接着，呈现 Coffee 实现细节，包括格式化、创建、打开、读取、写入、关闭、删除，还分析了存储回收机

制。在此基础上，结合重编程技术进行改进，先是修复 Coffee 一些 BUG，而后从格式化、适应 FLASH 类型定制、结合采集数据特点、文件元数据组织等方面进行改进。

第四章 WSNs 重编程技术研究与改进

重编程是无线传感器网络重要组成部分，很多情况是必不可少的一部分。大量的传感器节点一经部署，随着时间推移，难免需要对节点做一些变更(如升级，BUG 修复)，人工地去更新这些节点或者将这些节点收回集中处理将是非常耗力、耗时的工作。更糟糕的是，有些应用环境节点一经布置，便无法再次接触到这些节点(如布置在敌军战场的节点)或者节点工作不能被中止(即无法回收节点)。而重编程技术很好地解决了该问题。

4.1 重编程技术概述

重编程技术旨在对已部署的传感器节点进行远程 BUG 修复，软件升级、任务重分配、系统配置，甚至是替换整个系统。显然，需要解决两个问题：其一，如何将数据传送到传感器节点，即代码分发协议；其二，节点接收到数据如何更新，即节点端设计。除此之外，需要尽可能减少分发代码的大小以减少数据传输，从而降低功耗(传感器节点的能量消耗主要来源于数据通信)，这就要求更好的代码压缩技术[35]。

4.1.1 代码分发协议

更新传感器节点时，首先将待替换的数据先传送到结点。最容易想到的方法就是将整个系统映像传送到节点，而后进行更新。另一方法就是仅将差异的代码传送到节点。不同数据类型的传送，对应着不同的代码分发协议。

(1)替换整个系统映像

替换整个代码映像(Entire code image)，即将节点上所有代码替换掉，典型的代码分发协议有 XNP[36]、Deluge[37]、MOAP[38]、GARUDA[39]、MOS[40]。这种方法固然简单，但缺点也是致命的，需要传输大量数据，功耗极大，这对于功耗敏感的 WSNs 几乎不现实。除此之外，对于资源受限的传感器节点，整个映像需要占用节点的大量的存储空间。更糟糕的是，每次更新都需要重启节点，无法满足那些不能中断工作的应用场景。

(2)仅替换差异的代码

基于差异代码(Dirrerence-based)替换，即对比原映射和待更新映射，只传输不同的代码，进而替换，典型的代码分发协议有 code distribution[41]、Modecule[42]、Difference-based RMTD[43]、Zephyr[44]、Incremental linking[45]。这种方法最大优点是数据传输量达到最小，因此功耗极低。缺点也很明显，需要一个精确的算法找出原映像和新映射的不同之处，除此之外，还需要极复杂的节点端设计。

(3)基于模块替换

基于模块替换(Loadable modules)，是前两种方法的折中，对于模块化设计的操作系统，可以以模块为单位进行替换，类似于 Linux 动态模块加载，典型的代码分发协议有 FlexCup[46]、Module-linking[47][48]。

4.1.2 节点端设计

传感器节点接收到待更新的镜像、模块、配置文件后，需要合理的策略将其更新。不同的更新需求对应着不同的策略，如果是替换整个系统镜像，通常是通过 boot loader 将新的系统镜像替换旧的系统镜像，而后重新启动节点。对于模块或者配置文件，通常可以直接更新而无须重启动节点。

4.1.3 重编程实例

传感器节点资源受限、对功耗的苛刻要求、动态变化的网络拓扑以及无线网络链接的不稳定性，这些都增加了 WSNs 重编程设计的难度。因此，WSNs 重编程需要综合考虑存储空间的使用、能耗的有效性、传输的可靠性、更新的实时性[25]。

重编程作为 WSNs 操作系统的重要组成部分，几乎所有的节点操作系统都实现了重编程。本小结以 TinyOS、SOS、MantisOS 为例阐述典型的 WSNs 操作系统重编程技术，为改进 Contiki 重编程技术奠定基础。

(1)TinyOS

TinyOS 是加州大学伯克利分校(UC Berkeley)为无线传感器网络开发的基于组件/模块化架构专用操作系统[31]。如今，已演变成一个国际合作项目，即 TinyOS 联盟。TinyOS 是用 C 扩展语言 nesC(对内存使用方面进行优化)开发，旨在将组件化/模块化思想和基于事件驱动机制结合起来[31]。

TinyOS 1.x 重编程采用 MNP[49](Multi-hop Network Reprogramming)，整个过程是这样的：首先，将整个二进制镜像通过 WSNs 发送到相应的传感器节点(通常存储在 FLASH 中)；接着，通过 boot loader 将新镜像覆盖旧镜像；最后，重启节点，

重编程结束。

可见，TinyOS 重编程是采用替换整个系统镜像的策略来实现的，正如上文所分析的一样，这种方式需要传输的数据量巨大(数据冗余度高)，效率低下且功耗大。很多学者在改进 TinyOS 重编程做了很多努力，典型的有基于 Diff 的解决方案[50]、FlexCup[46]。然而 TinyOS 无法实现模块的动态加载、替换和删除，这是因为 TinyOS 基于组件/模块化设计仅限于编译阶段。

(2)SOS

SOS(Share Operating System)是于 1959 年由加州大学洛杉矶分校 NESL 开发的 WSNs 操作系统，用 C 语言编写。其最大特点是引入消息模式实现操作系统内核与用户应用程序间的绑定。SOS 通用内核包含了动态模块加载功能，整个系统采用动态编程思想，不需要的模块可以从内核卸载，节省内存，非常迎合资源受限的传感器节点。

SOS 重编程是基于模块替换的。为了让 ELF 文件更加适合资源受限的节点，SOS 通过裁减 ELF 文件头达到这一目的，并将新的 ELF 文件定义为 minielf。值得一提的是，SOS 模块加载是位置无关的，也就是说，模块无须重定位便可加载到任意地址。因为是基于位置无关，数据下载到节点无须携带与重定位相关的数据，也就减轻了网络负担。

(3)MantisOS

MantisOS[51]是美国科罗拉多大学开发的一款 WSNs 操作系统，整个系统用 C 语言开发，并且开发环境支持 Windows 和 Linux 双平台[52]，非常有利于跨平台开发及代码重用。MantisOS 支持多线程(抢占式调度)，并且利用节点循环休眠机制来降低功耗[31]。MantisOS 重编程是基于标准的 ELF 模块动态加载，这点与 Contiki 类似。

4.2 Contiki 重编程技术

可以把 Contiki[7]简单看成由两部分组成[15]：核心、可加载的程序(loaded program)。核心包括 Contiki 内核、程序加载器、语言运行库和通信服务。Contiki 代码映像如图 4-1 所示。

作为一款流行的 WSNs 操作系统，Contiki 自然实现了重编程。事实上，Contiki 在完全替换整个镜像与仅替换差异代码取得了平衡，既可以更新整个系统，也可以仅更新若干模块，实现了不同粒度的替换。

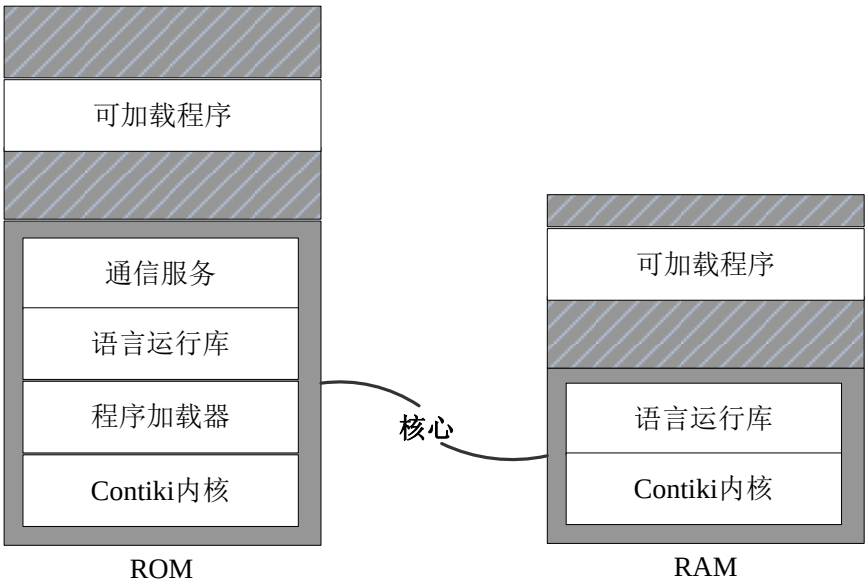

图 4-1 Contiki 代码映像示意图

核心被编译成一个二进制镜像，通常不被修改。若需要修改则通过 boot loader 替换整个镜像。所以，更新核心中的任一部分都需要通过 boot loader 替换整个系统镜像。

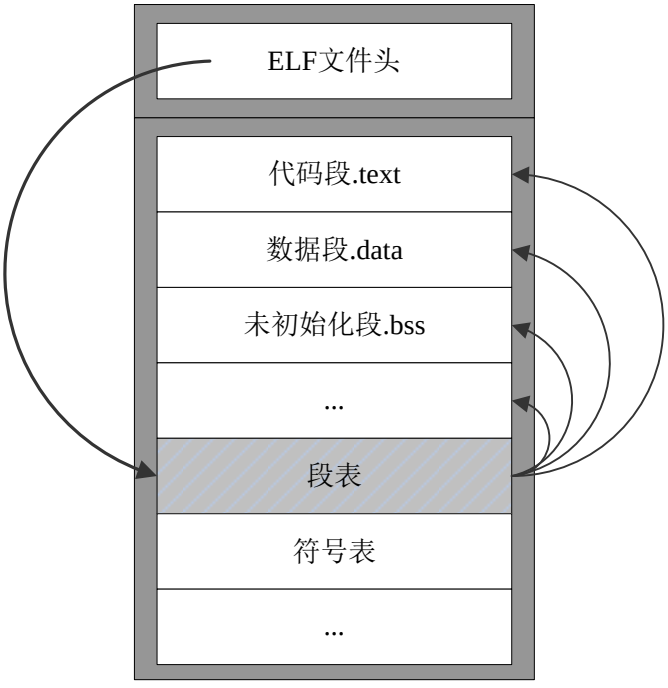

图 4-2 ELF 文件结构图

而核心之外的模块或者应用程序，可以通过动态加载机制进行加载、卸载、替换。可加载程序包含重定位信息和重定位函数。当一个程序被加载到系统，程序加载器首先为其分配足够的空间，若成功，程序被加载到内存，程序加载器调

用该程序的初始化函数，初始化函数启动或者替换一个或多个进程[7]，从而实现更新模块的功能。

(1)节点端设计

Contiki 重编程节点端设计是基于模块动态加载，用的是动态链接文件 ELF(Executable and Linking Format)中的可重定位文件(ELF 划分为 4 类：可重定位文件、可执行文件、共享目标文件、核心转储文件)。该文件在编译期并不进行链接，直到运行才解析文件在预链接过程无法确定的符号。当有新的模块或程序被传输到节点时，系统触发一个中断，利用动态加载机制将模块加载、替换甚至是卸载。Contiki 使用的 ELF 文件格式与 Linux 类似，ELF 文件头描述整个文件，再加上若干段组成完整的 ELF 文件，ELF 文件结构如图 4-2 所示。

文件由若干段组成，而 ELF 文件头描述整个文件信息，文件头成员变量 e_shoff 指向段表，而段表描述各个段的详细信息。ELF 文件头结构体源代码如下：

```
struct elf32_ehdr
{
  unsigned char e_ident[EI_NIDENT];   //包含魔数、文件类、字节序等
  elf32_half e_type;                  //ELF 文件类型
  elf32_half e_machine;               //CPU 平台属性
  elf32_word e_version;               //ELF 版本号
  elf32_addr e_entry;                 //程序入口地址
  elf32_off e_phoff;
  elf32_off e_shoff;                  //段表在文件中的偏移
  elf32_word e_flags;                 //ELF 标志位
  elf32_half e_ehsize;                //文件头大小
  elf32_half e_phentsize;
  elf32_half e_phnum;
  elf32_half e_shentsize;             //段表描述符大小
  elf32_half e_shnum;                 //段表描述符数量
  elf32_half e_shstrndx;              //字符串表所在的段在段表中的下标
};
```

(2)代码分发协议

Contiki 重编程代码分发是利用 Rime 或者 uIP 协议栈的广播通信完成，将待更新的配置文件、模块甚至是系统映像通过协议栈传送到各个传感结点。

4.3 重编程改进与实现

基于上述分析，可以从节点端和代码分发协议两方面对 Contiki 重编程进行改

进。本设计将重心放在节点端，Contiki 节点端采用动态模块加载机制，ELF 文件作为动态加载模块对象，对其优化，以适应内存受限的传感器节点。

为了通用性，Linux 定义的 ELF 相关结构体有许多成员变量，需要占用较多内存空间。Contiki 动态加载所用到的 ELF 文件几乎是没有经过裁减的(如 ELF 头和段表)，给资源受限节点带来极大的挑战。对 ELF 文件裁剪将有效减少内存的使用，给内存受限的节点带来极大的便利。

4.3.1 ELF 文件设计

ELF 文件由多个段加上文件头组成，除了.txt、.data、.bss 最常用的三个段外，标准的 ELF 文件还定义了很多段，如只读数据段.rodata、程序初始化段.init、程序终结段.fini。而这些段使用的特别少，为适应资源受限的传感器节点，可以将其裁剪，并将字符串表合并到符号表，重新设计的 ELF 文件组织结构如图 4-3 所示。

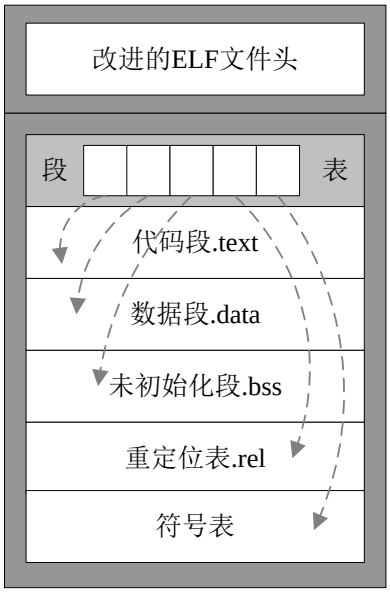

图 4-3 改进后的 ELF 文件组织结构图

裁剪后的 ELF 文件只保留必须的段，即代码段、数据段、未初始化段、重定位表、符号表。

4.3.2 ELF 文件裁剪

标准的 ELF 文件头有 52 个字节之多。成员变量 e_ident(包含魔数、文件类、字节序等)有 16 个字节，其中预留了 9 个字节供扩展，可将其裁剪为 1 个字节，ELF 文件头成员变量 e_ident 裁剪前后对比如图 4-4 所示。

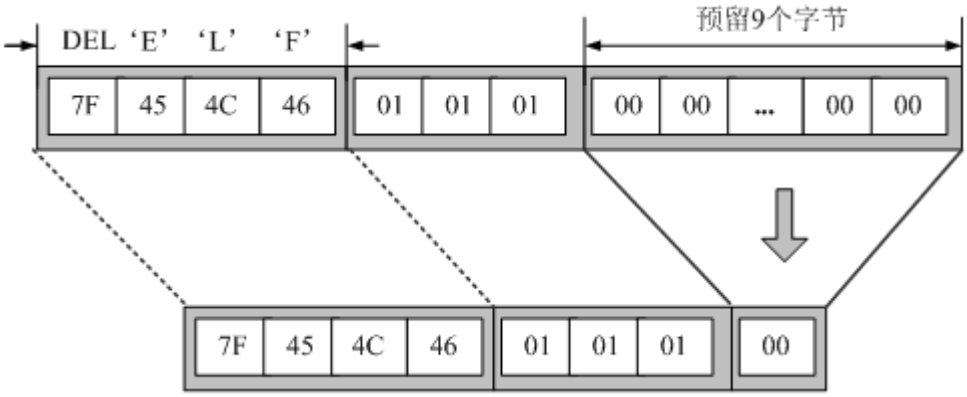

图 4-4 ELF 文件头变量 e_ident 裁剪前后对比示意图

除此之外，标准 ELF 文件头包括成员变量 e_shoff(段表在文件中的偏移)、e_shentsize(段表描述符大小)、elf32_half e_shnum(段表描述符数量)、elf32_half e_shstrndx(字符串表所在的段在段表中的下标)，重新组织 ELF 文件之后，便无须这些变量来标识。例如，段表在文件中的偏移即文件头大小 e_ehsize。裁剪后的 ELF 文件头只有 22 个字节，elf32_ehdr 源代码如下：

```
struct elf32_ehdr
{
   unsigned char e_ident[EI_NIDENT];     //包含魔数、文件类、字节序等
   elf32_half e_type;                     //ELF 文件类型
   elf32_half e_machine;                  //CPU 平台属性
   elf32_word e_version;                  //ELF 版本号
   elf32_addr e_entry;                    //程序入口地址
   elf32_half e_ehsize;                   //文件头大小
};
```

(3)ELF 其他结构体裁剪

段表实际上是一个数组，数组类型是 elf32_shdr 结构体，标准的 elf32_shdr 结构体有 38 个字节之多，裁剪掉一些非关键变量，得到 14 字节 elf32_shdr 结构体，源代码如下：

```
struct elf32_shdr
{
   elf32_word sh_name;                    //段名
   elf32_word sh_type;                    //段的类型
   elf32_off sh_offset;                   //段的偏移
   elf32_word sh_size;                    //段的长度
};
```

符号表通常是独立的一个段，存储为一个数组，第一个数组元素即为一个符号,由结构体 elf32_sym 描述。标准的 elf32_sym 结构体有 16 个字节，裁剪后为 10

字节，源代码如下：

```
struct elf32_sym
{
   elf32_half st_shndx;                  //符号所在的段
   elf32_word st_name;                   //符号名
   elf32_word st_size;                   //符号大小
};
```

4.4 本章小结

本章主要介绍了重编程技术原理及应用实例，并在此基础上对 Contiki 重编程技术进行改进。首先，阐述了重编程技术在 WSNs 必要性，接着，分析重编程关键技术，即代码分发协议和节点端设计。随后，以 3 个典型的 WSNs 操作系统(TinyOS、SOS、MantisOS)为例介绍了典型重编程技术实现。最后，分析了 Contiki 重编程技术，并在基础上进行改进，包括重新设计整个 ELF 文件结构和文件裁剪。

第五章 文件系统与重编程测试

为了验证文件系统与重编程结合优化的效果，需要对其进行一系列测试，包括模块测试、集成测试、功能测试、性能测试。先在开发板上进行测试，首先单独测试文件系统模块，接着将重编程模块加入一起测试。鉴于开发板节点数目太少，难以完成性能测试，值得庆幸的是，可以在 Contiki 配套仿真环境 COOJA 做大规模仿真和性能测试。

5.1 开发平台简介

本文选择 Contiki 作为研究文件系统和重编程的平台，但 Contiki 相关资料甚少，前期工作不得不深入源码分析 Contiki 操作系统。一个好的开发环境将非常有利于源码分析，出于调试方便性和完整性考虑，本文选择 IAR+J-Link 作为软件开发环境，硬件平台采用实验室自主设计的开发板 ACme[①]。除此之外，在改进文件系统和重编程过程中，也可以用该套开发环境进行调试，测试。

(1)硬件平台

开发板 ACme 是专为无线传感器网络设计的，功能强大。采用基于 Cortex-M3 的微处理器 STM32F103RBT6，无线射频模块 CC2520，开发板外观如图 5-1 所示。

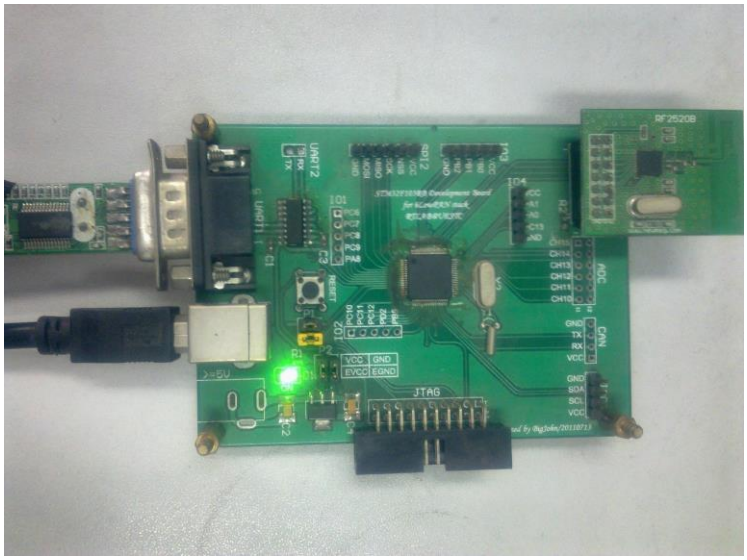

图 5-1 ACme 开发板示意图

[①] 设计 ACme 开发板最初目的是用来智能电网用户端用电监测方面的研究。

意法半导体(STMicroelectronics)公司推出的 STM32 是基于 Cortex-M3(ARM 公司 V7 架构)核心的 32 位 SoC 微控制器，具有高代码密度、优异性能、低功耗等优点。STM32F103RBT6(SRAM 大小为 20KB)集成了许多常用外设，包括 128KB FLASH、3 个通用 16 位定时器、PWM 定时器、I2C 总线、SPI 总线、USART 串口、CAN 总线接口等。这些特性使得该处理器非常适合用在低成本、低功耗、较为复杂的嵌入式应用中。

CC2520 是 TI 公司推出的第二代 ZigBee/IEEE 802.15.4 无线射频收发器，数据传输速率最高可达 250kbps，对数据缓冲、数据加密、数据认证、空闲信道检测等提供良好的支持。通过 SPI 总线将 CC2520 无线射频模块与 STM32F103RBT6 微处理器相连。

开发板还引出了丰富的接口，将 J-Link 与板上 JTAG 口连接，串口与 PC 连接，再结合 IAR 集成开发环境，便可以很方便地进行调试。

(2)软件平台

官方发布的 Contiki 源码只支持 Linux 开发环境，然而 Linux 对 J-Link 调试器支持很不完善，给开发调试带来极大不便。而 Windows 下的 IAR 集成开发环境可以与 J-Link 无缝集成，极大方便调试。基于此，本文采用 IAR+J-Link 作为软件开发平台。

IAR Systems 是世界领先的嵌入式开发工具和服务提供商，旗舰产品 IAR Embedded Workbench 无疑是当今最好用的嵌入式集成开发环境之一，目前已支持世界范围内绝大多数微处理器，除此之外，IAR 对主流调试器也支持得很完善，如 J-Link、ST-Link。

J-Link 硬件仿真器是 SEGGER 公司为 ARM 芯片推出的 JTAG 调试下载工具，目前几乎支持所有 ARM 芯片。除此之外，J-Link 通过 RDI 接口与主流嵌入式集成开发环境(如 IAR，KEIL，ADS,RealView)无缝连接，极大方便调试。本例中，用 JTAG 线将 J-Link 与开发板 JTAG 口连接。

5.2 Contiki 移植

准备好硬件和软件平台后，为了更好地剖析 Contiki 相关源代码及后续的改进测试，需要将 Contiki 操作系统从 Linux 环境移植到 IAR，主要包括开发工具配置及系统移植，整个移植流程如图 5-2 所示。

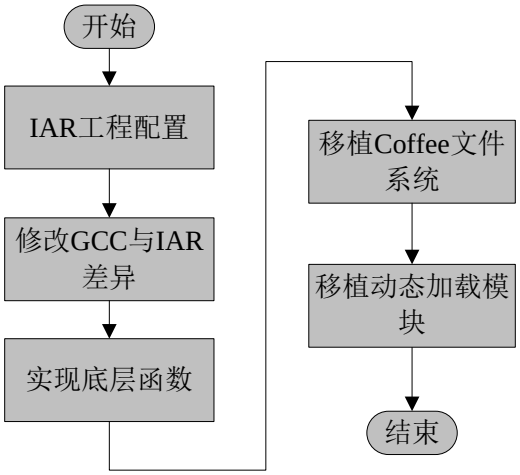

图 5-2 Contiki 移植流程图

5.2.1 IAR 配置[②]

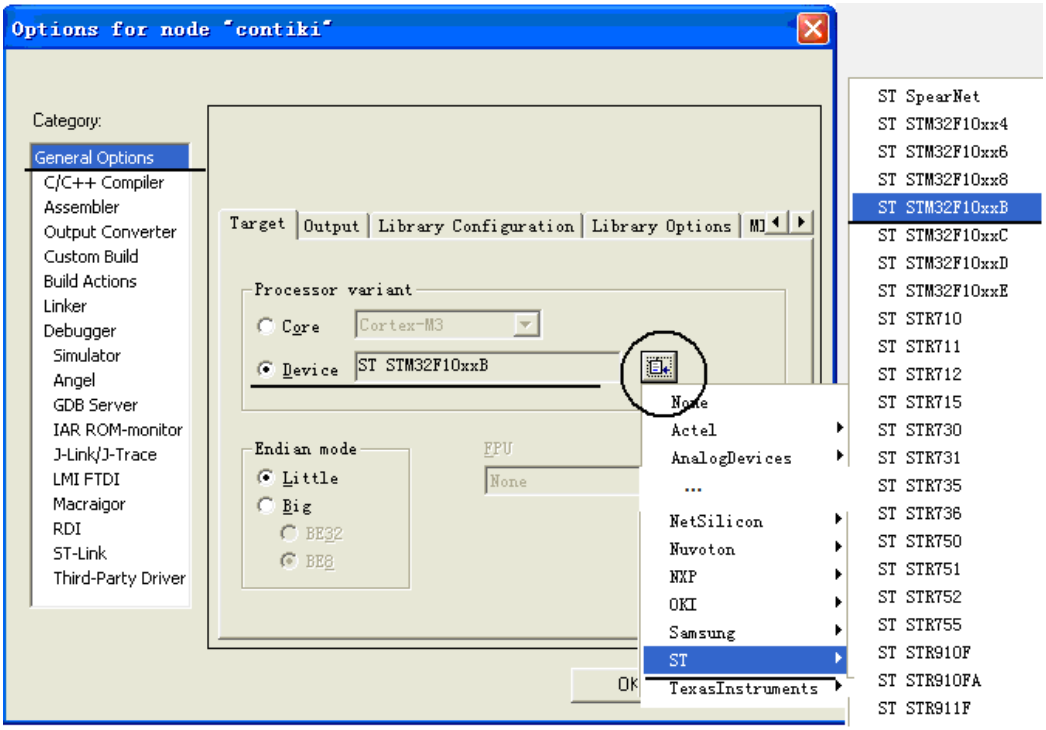

图 5-3 IAR 工程项目属性配置示意图

不同于 Linux 用 Makefile 组织整个工程项目，IAR 是通过配置工程项目和设置文件层次关系实现整个项目管理。首先，建立工作区(File->New->Workspace)和项目(Project–>Create New Project)，接下来配置项目属性(右击项目->options)。选择正确的微处理器型号(General Options->Target->Device，这里选择 ST STM32F10xxB)

[②]本文使用的 IAR 版本是 IAR Embedded Workbench for ARM 5.30

及调试器类型(Debugger->Driver,这里选择 J-Link/J-Trace)。IAR 工程项目属性配置如图 5-3 所示。

将 Contiki 源码按层次添加到工作区(右击项目->add),并新建 cpu/arm /stm32f103 和 platform/stm32f103 组。将源代码文件路径加到预编译路径，否则会提示不能打开源文件错误。工程目录及预编译路径配置如图 5-4 所示。

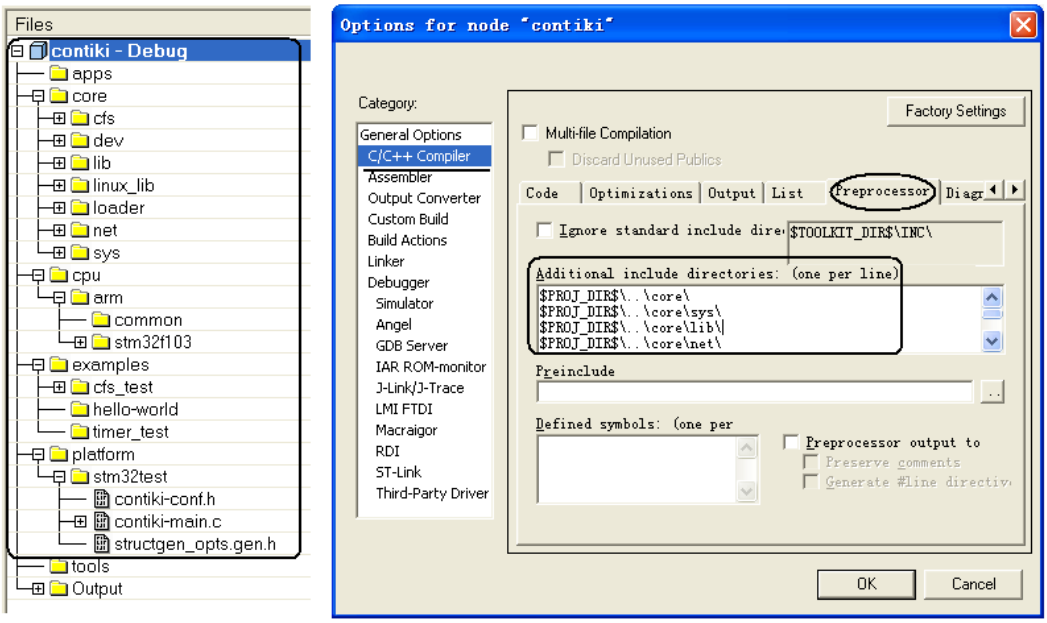

图 5-4 Contiki 工程目录和预编译路径配置示意图

5.2.2 GCC 到 IAR

设置好 IAR 工程项目后，就可以编译整个项目了，根据提示的错误信息进行修改。编译链接遇到的主要问题及对应的解决方法如下：

(1)头文件名称差异

不同开发环境对板级库文件命名不一样，比如 MSP430 平台的基础头文件在 Linux 为 io.h，而在 IAR 则为 io430.h。为了让代码更具移植性，加入预编译指令解决这一差异，源代码如下：

```
#ifdef __GNUC__
  #include <io.h>
#elseif __IAR_SYSTEMS_ICC__       /*将适应 IAR 头文件添加进去*/
  #include <stm32f10x.h>
#endif
```

(2)外部符号未定义

符号 autostart_processes 未定义，原因是 autostart_processes 指针数组是由宏 AUTOSTART_PROCESSES 定义，而该宏又取决于条件编译变量 AUTOSTART_ENABLE。解决该问题只需把变量 AUTOSTART_ENABLE 定义为 1，在 contiki-conf.h 文件添加如下语句：

```
#define     AUTOSTART_ENABLE      1
```

符号 dint 和 eint 未定义，这是因为 Linux 开发环境用 eint()和 dint()实现开、关中断，但 IAR 使用__enable_interrupt()和__disable_interrupt()。解决方法是将 eint()和 dint()替换掉，这里采用更具移植性的方法，即加入预编译指令，部分源代码如下(添加在 platform-conf.h 文件)：

```
#ifdef __IAR_SYSTEMS_ICC__
   #define dint() __disable_interrupt()
   #define eint() __enable_interrupt()
#endif
```

(3)定义格式不同

Linux 与 IAR 对底层函数的定义有些不同，例如，Linux 下中断服务处理程序 ISR 定义格式，将 GCC 定义的中断服务处理程序改成符合 IAR 格式，如下：

```
/***原始版本(GCC 格式)***/
interrupt(PORT2_VECTOR)
irq_p2(void)

/***需要改成符合 IAR 格式***/
#pragma vector= PORT2_VECTOR
__interrupt void irq_p2(void)
```

(4)嵌入式汇编

IAR 汇编格式不同于 GCC，自然也包括嵌入式汇编。这就要求对 Contiki 所有嵌入式汇编进行重写。GCC 与 IAR 的内嵌汇编并非一一对应，有时需要读懂 GCC 内嵌汇编的源码含义，才能改成 IAR 格式的嵌入式汇编。比如，中断使能代码：

```
/***GCC 下中断使能源码***/
asmv("mov r2, %0" : "=r" (sr));
asmv("bic %0, r2" : : "i" (GIE));

/***IAR 下中断使能源码***/
asmv("EINT");
```

再比如，获取堆栈指针，GCC 用嵌入式汇编实现，但 IAR 已对该功能进行封装，源代码如下：

```
//asmv("mov r1, %0" : "=r" (stack_pointer));                  //GCC 嵌入式汇编
*stack_pointer = (unsigned short)__get_SP_register();     //IAR 实现方式
```

5.2.3 时钟驱动

时钟驱动是与硬件相关，不同的处理器时钟驱动不同。时钟驱动包括时钟初始化、时钟延迟、返回当前时钟、时钟中断处理程序。而时钟中断处理程序无疑是最重要的一个，其源代码如下：

```
void SysTick_Handler(void)
{
   SCB->ICSR = SCB_ICSR_PENDSTCLR;
   current_clock++;                          //时钟嘀嗒
   if(etimer_pending() && etimer_next_expiration_time() <= current_clock)
   {
      etimer_request_poll();            //提升 etimer 系统进程优先级
   }
   if(--second_countdown == 0)      //秒数
   {
      current_seconds++;
      second_countdown = CLOCK_SECOND;
   }
}
```

Contiki 系统维护两个时间：时钟嘀嗒数、秒数，这样有利于不同粒度定时器设定。时钟中断处理程序 ISR 完成时钟嘀嗒数累加，秒数累加(达到 1s 才累加)，提升 etimer 系统进程 etimer_process 优先级(etimer 列表不为空且还有 etimer 未到期)。

5.2.4 Coffee 移植

Coffee 文件系统移植工作主要是根据开发板实际情况配置一些参数和实现底层函数。本开发板 FLASH 是片内的，故只需查看微处理器数据手册和参考手册，设置相应变量和实现底层函数。

(1)参数配置

需要配置的参数包括 FLASH 起始地址 FLASH_START、FLASH 页大小 FLASH_PAGE_SIZE、页数 FLASH_PAGES、Coffee 所管理的起始地址 COFFEE_ADDRESS、Coffee 区大小 COFFEE_SECTOR_SIZE。还有一些额外的配置参数，比如文件名长度、最大打开文件数。

查看微处理器数据手册和参考手册(本实验版的 MCU 型号是 STM32F103RBT6)，可以得到 FLASH 大小为 128KB 、FLASH 起始地址为 0x08000000、FLASH 页大小为 1KB、FLASH 页数为 128。值得一提的是，Coffee

将 FLASH 最后 3 页留给 NVM。为了适应不同类型的 FLASH(块大小或页大小不同，接块或按页擦除)，Coffee 文件系统提供了逻辑区和逻辑页，以区为单位进行擦除。这样做的目的是：一方面，对付大的存储设备，如 SD 卡，将逻辑区设为页大小的几倍，可加快 FLASH 顺序扫描速度；另一方面，便于大数据存储(如采集数据是音频、视频)。然而，ContikiWiki 文档中建议将 COFFEE_PAGE_SIZE 和 COFFEE_SECTOR_SIZE 设小一点，以获得更合理的性能(reasonable performance)。基于这些宏，便可定义 Coffee 页面总数 COFFEE_PAGES 、Coffee 起始地址 COFFEE_START、Coffee 总大小 COFFEE_SIZE。部分源代码如下：

```
#define COFFEE_PAGE_SIZE (FLASH_PAGE_SIZE/4)//Coffee逻辑页大小，256B
#define COFFEE_SECTOR_SIZE FLASH_PAGE_SIZE //Coffee 逻辑区大小, 1KB

#define COFFEE_PAGES    \
((FLASH_PAGES*FLASH_PAGE_SIZE-(COFFEE_ADDRESS-FLASH_START))
/COFFEE_PAGE_SIZE)                                    //Coffee 页数

#define COFFEE_START   \
(COFFEE_ADDRESS & ~(COFFEE_PAGE_SIZE-1))      //起始地址

#define COFFEE_SIZE      \
(COFFEE_PAGES*COFFEE_PAGE_SIZE)                 //Coffee 文件系统大小
```

更直观显示上述宏的值如图 5-5 所示。

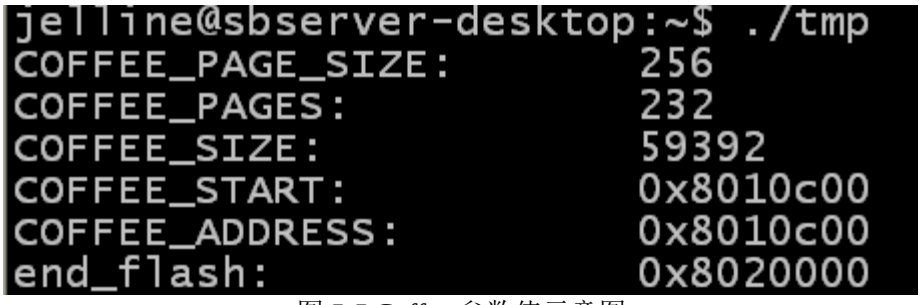

图 5-5 Coffee 参数值示意图

除此之外，结合本文的实际应用，配置 Coffee 其他参数，如下：

```
#define COFFEE_NAME_LENGTH 20                    //文件名长度
#define COFFEE_MAX_OPEN_FILES 50                 //最大打开文件数
#define COFFEE_FD_SET_SIZE 20                    //文件缓冲数目
#define COFFEE_DYN_SIZE (COFFEE_PAGE_SIZE*1)
#define COFFEE_MICRO_LOGS 1                      //配置微日志
#define COFFEE_LOG_TABLE_LIMIT 128               //微日志索引个数
#define COFFEE_LOG_SIZE 64                       //微日志大小
```

(2)宏映射

将 Coffee 文件系统读 COFFEE_READ、写 COFFEE_WRITE、擦除 COFEE_ERASE 宏映射到平台相关的底层函数，如下：

```
#define COFFEE_WRITE(buf, size, offset)     \
stm32_flash_write(COFFEE_START + offset, buf, size)

#define COFFEE_READ(buf, size, offset)      \
stm32_flash_read(COFFEE_START + offset, buf, size)

#define COFFEE_ERASE(sector)    stm32_flash_erase(sector)
```

Coffee 擦除函数的实现见 3.4.1 小节的 Coffee 格式化。本文使用的 FLASH 是片内的，可以直接寻址，读取数据直接调用库函数 memcpy 即可。FLASH 写入函数，因为 FLASH 固有特性(写的时候只能从 1 变成 0)，所以需要先把数据以页为单位拷贝到缓冲区，直接在缓冲区修改数据，而后擦除该页，最后才将缓冲区数据写入 FLASH。FLASH 底层写函数 stm32_flash_write 关键代码如下：

```
stm32_flash_read(curr_page, buf, FLASH_PAGE_SIZE); //1.将数据拷贝到缓冲区
memcpy(buf + offset, data, next_page - i);          //2.修改缓冲区的数据
FLASH_ErasePage(i);                                 //3.擦除 FLASH 页
FLASH_ProgramWord(curr_page + j, *(u32_t*) &buf[j]); //4.写入缓冲区内容
```

5.2.5 动态加载模块移植

Contiki 动态加载模块是实现重编程技术的基础，然而 Contiki 开发环境是 Linux，其源代码引用了 Linux 库头文件，加之，GCC 与 IAR 的编码格式不尽相同。移植过程中出现的问题及解决方法如下：

(1)不能打开源文件

Contiki 仅仅只是引用了 Linux 库的一些头文件，并没有调用 Linux 库函数，事实上，Contiki 实现了相关库函数(如 dlfcn.h 的 dlopen 函数)。将 Contiki 放到 IAR，编译的时候会提示不能打开源文件，解决这个问题只需将这些头文件从 Linux 找出来并加到 IAR 工程目录，Linux 库头文件可能又包含其他库头文件，也将这些头文件包括进来，涉及到文件如下：

```
./include/malloc.h dlfcn.h features.h unistd.h
./include/i386-linux-gnu/bits/predefs.h wordsize.h dlfcn.h environments.h esizes.h confname.h types.h posix_opt.h
./include/i386-linux-gnu/sys/cdefs.h
./include/i386-linux-gnu/gnu/stubs.h stubs-32.h
./include/getopt.h
```

(2)类型不匹配

GCC 与 IAR 使用的编译器不同，在类型转换方面存在差异，总体上，IAR 在类型隐性转换上要求比 GCC 高得多。例如 core\loader\cmod.c 文件中 h.bss = h.data + h.datasize 语句，h 类型是 cle_info 结构体，bss、data、datasize 类型分别为 void *、void *、cle_word(实质是 u16_t)，在 GCC 可以正常运行，但在 IAR 编译时会提示"expression must be a pointer to a complete object type"错误。主要问题在于类似的隐式转换，解决该问题只需将类型转换显式化，如下：

```
h.bss = h.data + h.datasize
/*将隐性的类型转改成显性*/
h.bss = (void *)((u32_t)h.data + h.datasize);
```

GCC 可以隐性地将指针转换成函数指针(都是地址)，然而在 IAR 行不通。需要将指针显性转换成函数指针。这种情况在 Contiki 多次出现，部分源代码如下：

```
cmod_module[imod].fini = cle_lookup(&h, pread, off, "_fini");
//将指针显性转换成函数指针
cmod_module[imod].fini = (void (*)(void))(cle_lookup(&h, pread, off, "_fini"));

init = cle_lookup(&h, pread, off, "_init");
//将指针显性转成函数指针
init = (void (*)(void))(cle_lookup(&h, pread, off, "_init"));

elfloader_fini = cle_lookup(&h, xmem_pread, eepromaddr, "_fini");
//将指针显性转换成函数指针
elfloader_fini = (void (*)(void))(cle_lookup(&h, xmem_pread, eepromaddr, "_fini"));

elfloader_init = cle_lookup(&h, xmem_pread, eepromaddr, "_init");
//将指针显性转换成函数指针
elfloader_init = (void (*)(void))(cle_lookup(&h, xmem_pread, eepromaddr, "_init"));
```

(3)重定义错误

GCC 环境下，通过 Makefile 文件组织工程文件，Makefile 决定将哪些文件加入工程，IAR 则不同，IAR 是通过工程目录层次关系管理整个项目。将动态加载模块从 GCC 移到 IAR，编译器会提示某些变量重定义错误。多数情况是变量在 Contiki 定义了，同时也在 Linux 库文件定义(动态加载模块使用了库文件中的部分变量)，例如 off_t 在 contiki-conf.h 定义，同时也在库文件 unistd.h 定义。解决方法很简单，找出分别定义的地方，通过逻辑分析，注释掉其中一个。

链接时也会提示符号重定义错误，这是因为函数在不同地方实现了两次。符号 elfloader_arch_relocate 、 elfloader_arch_write_rom 以及 elfloader_arch_allocate_ram 在文件 elfloader-stub.c、elfloader-stm32f10x.c 都实现了。通过逻辑，

注释其中一个实现，这里注释 elfloader-stub.c 中相应的实现。符号 elfloader_load、elfloader_unknow 在文件 elfloader.c 和 elfloader_compat.c 都实现了，这里注释 elfloader_compat.c 中相应的实现。

(4)未定义错误

编译时，在 elfloader_compat.c 文件中提示_etext、_edata、__data_start 未定义错误，这些符号(被称为特殊符号)被定义在链接脚本中，程序使用需先声明，链接器会在链接成可执行文件时解析成正确的值。在 elfloader_compat.c 加入这些变量的外部引用，如下：

```
extern unsigned long _etext;          //代码段结束地址 _stext 为代码段开始地址
extern unsigned long __data_start;    //初始化的数据开始地址
extern unsigned long _edata;          //初始化的数据结束地址
```

文件 elfloader_compat.c 提示 ROM_ERASE_UNIT_SIZE 未定义错误，Contiki 是按 sector 擦除的，这里将 ROM_ERASE_UNIT_SIZE 定义为 COFFEE_SECTOR_SIZE，在 elfloader_compat.c 加入如下代码：

```
#include "cfs-coffee-arch.h"
#define ROM_ERASE_UNIT_SIZE COFFEE_SECTOR_SIZE
```

5.3 测试

完成了上述工作，便可以在开发板测试文件系统和重编程。基于软件工程思想，先功能测试，后性能测试。先模块测试，后集成测试。测试用例直接放在自启动的指针数组(通过宏 AUTOSTART_PROCESSES 加入)，在主函数中，只要根据这个数组启动进程，通用的主函数 main 如下：

```
int main()
{
   dbg_setup_uart();                                  //串口初始化
   FLASH_ClearFlag(FLASH_FLAG_BSY | FLASH_FLAG_EOP   \
| FLASH_FLAG_PGERR | FLASH_FLAG_WRPRTERR);   //清除 FLASH 标志位
   clock_init();                                      //时钟初始化
   process_init();                                    //进程初始化
   process_start(&etimer_process, NULL);      //启动系统进程
   autostart_start(autostart_processes);          //启动指针数组所有进程
   /*系统进入死循环*/
   do { }while(process_run() > 0);         //运行所有高优先级任务及处理事件

   return 0;
}
```

5.3.1 文件系统

基于上述的改进及实现，对文件系统进行测试，首先是功能性测试，通过串口输出测试 Coffee 格式化、创建、读取、写入等函数。用宏 PROCESS_THREAD 声明一个进程，并将其加入自启动指针数组(通过 AUTOSTART_PROCESSES 宏)。文件系统测试函数流程如图 5-6 所示。

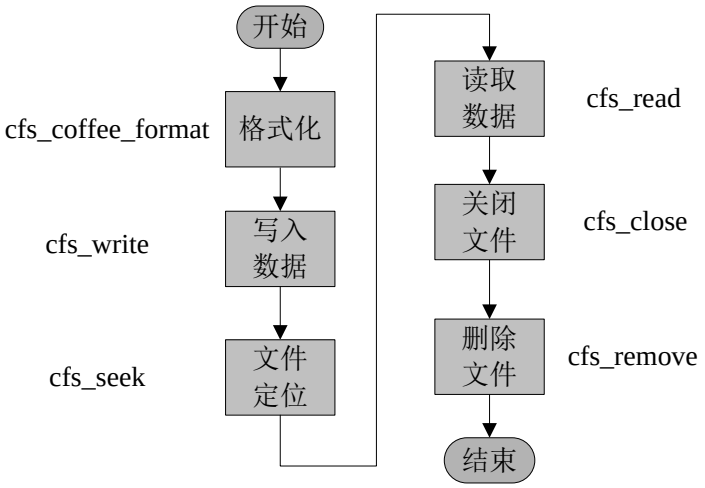

图 5-6 文件系统测试代码流程图

首先对 Coffee 文件系统进行格式化，即将整块 FLASH 擦除(从“0”变成“1”)，然后创建一个文件(指定标志位 CFS_WRITE | CFS_READ)，接着向新建的文件写入数据，将文件定位到开始处(cfs_seek)，读取数据。文件不使用时，将其关闭，释放资源供他用，最后将其删除。文件系统功能测试函数源代码如下：

```
#include "contiki.h"
#include "debug-uart.h"
#include "cfs/cfs.h"
#include "cfs-coffee-arch.h"

PROCESS(cfs_test_process, "cfs test");
AUTOSTART_PROCESSES(&cfs_test_process);

PROCESS_THREAD(cfs_test_process, ev, data)
{
  PROCESS_BEGIN();
  usart_puts("***cfs test process start***\n");

  if(cfs_coffee_format() == - 1)                    //Coffee 文件系统格式化
  {
    usart_puts("coffee format error.");
```

```
    return - 1;
  }

  int fd = cfs_open("CoffeeTest", CFS_WRITE | CFS_READ);   //新建一个文件
  if(fd == - 1)
  {
    usart_puts("First time open error.");
    return - 1;
  }

  char buf1[] = "Hello, World!";
  char buf2[32] = "Orignal!";

  usart_puts("The orignal buf1 and buf2 is : ");
  usart_puts(buf1);
  usart_puts(buf2);

  int size_write = cfs_write(fd, buf1, sizeof(buf1));      //写入数据
  cfs_seek(fd, 0, CFS_SEEK_SET);                           //设置文件光标位置
  int size_read = cfs_read(fd, buf2, sizeof(buf1));        //读取数据

  usart_puts("The update buf1 and buf2 is : ");
  usart_puts(buf1);
  usart_puts(buf2);

  cfs_close(fd);                                           //关闭文件
  if (cfs_remove("CoffeeTest") == -1)                      //删除文件
  {
    usart_puts("Remove file error.");
  }

  PROCESS_END();
}
```

串口打印结果如图 5-7 所示。

```
Coffee format has been done.
systick isr
The orignal buf1 and buf2 is : Hello, World!Orignal!
The update buf1 and buf2 is :  Hello, World! Hello, World!
systick isr
```

图 5-7 Coffee 功能性测试结果示意图

当然，这样的测试远远不够，结果正确并不能保证逻辑准确。本文通过单步调试的方式对改进后的文件系统进行逻辑分析，确保逻辑正确。J-Link 调试器与 IAR 无缝结合使得调试非常方便，调试窗口可以查看 FLASH、寄存器、变量的值，

如图 5-8 所示。

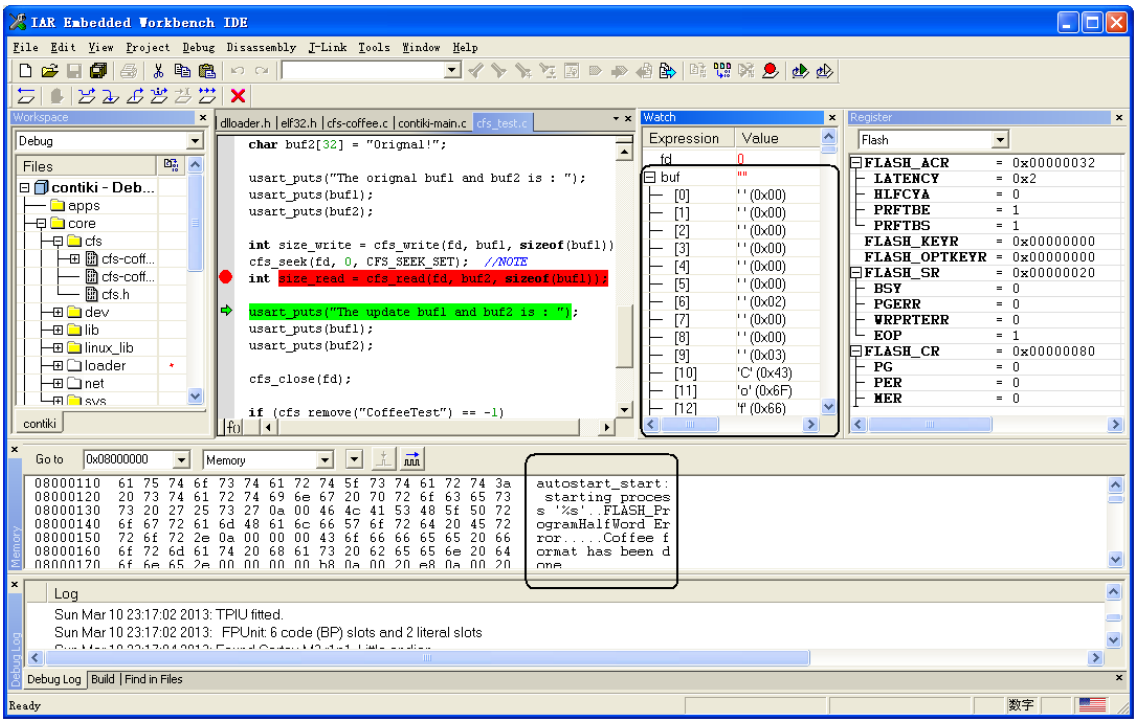

图 5-8 文件系统单步调试示意图

5.3.2 重编程

本文对重编程的改进集中在对 ELF 文件进行重新设计，进而对 ELF 文件裁剪，取若干典型变量的裁剪前后对比如表 5-1 所示，可见裁剪后的 ELF 文件要小得多。

表 5-1 ELF 文件裁剪前后对比

名称	更进前(字节)	更进后(字节)
ELF 文件头	52	22
e_ident	16	9
段表	38	14
符号表	16	10

首先进行功能性测试，将改进后的代码烧入开发板，通过串口输出观察重编程效果，比如节点收到更新数据后向串口打印信息，节点更新完毕再向串口输出信息，通过这样方式可以对重编程进行功能性测试。

但因板子数量太少(只有 5 个)，无法完成重编程的大规模功能性测试及性能测试。庆幸的是，Contiki 提供一套完整的开发平台 InstantContiki，该平台基于 Ubuntu，可以装在虚拟机上(如 VMware)。InstantContiki 附带仿真器 COOJA[53][54]，该仿真

器几乎可以完全仿真 Contiki 官方支持的典型传感器节点 MicaZ、Sky、ESB。本文使用 COOJA 对大规模传感器节点重编程技术进行仿真并对其进行性能评估。

启动 InstandContiki，打开终端，进入到 contiki/tools/cooja 目录，运行命令 ant run，启动 COOJA 仿真器。打开相应的 csc 格式文件 (File->Open simulation->Browse)，点击 Start 开始仿真，通过仿真可视化窗口，可以很直观看到数据传递过程。除此之外，可以选择显示的条目，如节点编号、节点地址、位置、输出信息、传输路径等。可视化仿真窗口 4 个不同阶段截图如图 5-9 所示。

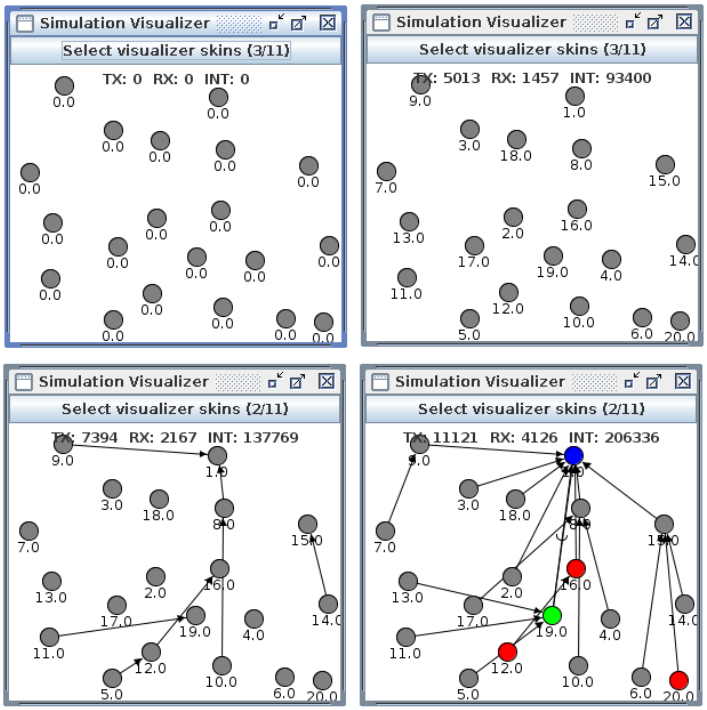

图 5-9 COOJA 仿真可视化窗口示意图

以上只是便于感观认识，为了分析其性能，还需对数据进行分析。将无线收发日志及监听日志窗口的数据保存(右击窗口选择 Save to file)为文件，以便进行性能分析。无线收发日志及监听日志窗口如图 5-11 所示。

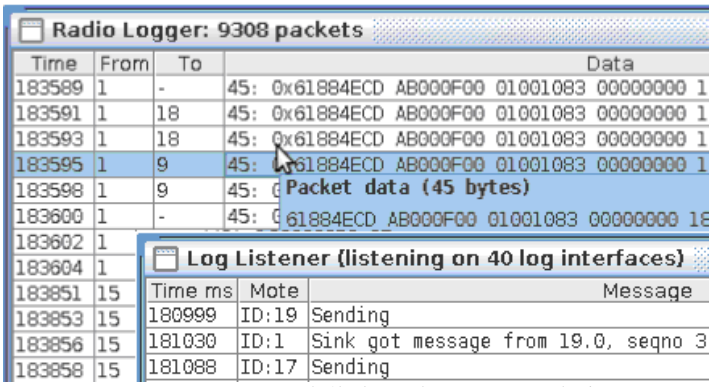

图 5-10 COOJA 无线收发日志及监听日志窗口

5.4 本章小结

本章对改进的文件系统和重编程进行测试。首先介绍开发平台，包括硬件平台(STM32F103RBT6+CC2520)和软件平台(IAR+J-Link)。接着详细分析 Contiki 从 GCC 移植到 IAR 细节。在此平台基础上，对文件系统和重编程进行功能性测试。最后，讨论网络仿真器 COOJA 在大规模传感器网络测试中的应用。

第六章 总结

6.1 论文工作总结

鉴于文件系统与重编程技术在无线传感器网络的重要性，而且两者有着紧密的联系，本文将文件系统和重编程技术结合起来考虑，综合优化，提升总体性能。工作总结如下：

(1)剖析 Contiki 操作系统

鉴于介绍 Contiki 技术细节资料甚少，在开展研究之前，深入 Contiki 源码分析整个系统，还原技术细节，这有助于对文件系统和重编程技术的理解，为后续研究打下夯实的基础。

(2)改进 Coffee 文件系统

Coffee 设计和实现与桌面甚至是嵌入式文件系统差别甚大，深入源码分析其技术细节，对存在 BUG 进行修复。结合重编程技术，对文件系统进行改进。

(3)改进重编程程技术

分析无线传感器网络操作系统典型的重编程技术实现方式，在此基础上，分析 Contiki 重编程实现方式并对其进行改进。

(4)建立开发环境并测试

为了便于调试，本文将 Contiki 整个开发环境从 GCC 迁到 IAR，并将 Contiki 移植到尚未支持平台，极大方便代码调试及测试。

6.2 存在问题及展望

尽管本研究取得了一定成果，但还存在一些不足。存在问题及接下来工作如下：

(1)代码分发协议

重编程技术包含代码分发协议和节点端设计，本文仅对节点端设计进行改进。Contiki 是用协议栈广播功能实现代码分发的，没有专用的代码分发协议。下一步工作可以尝试将代码协议在 Contiki 实现，考虑协议安全性，并测其性能。

(2)测试不充分

因实验节点数目少，无法在实际环境中对重编程进行功能性测试。在性能测试上也不充分。下一步工作，可以进一步测试改进前后的性能，在此基础，进一步对文件系统和重编程进行改进。

致 谢

时光荏苒，在本论文完成之际，谨向所有给予我热忱关心和帮助的人，表达最诚挚的谢意。

感谢我导师罗克露教授应允我提前进入教研室(推免，提前半年进入)，为我创造了优越的科研和学习环境。您的包容让我们得以按照最有利于自己的方式去发展，并向我们提供最大限度的支持。您严谨的治学态度、一丝不苟的工作作风和和蔼可亲的生活作风都深深地影响了我。总之，您关心我们的方方面面，从学习到生活，从研究到发展，这一切都远远超出您所应付的职责。

感谢指导老师廖勇副教授三年来对我的悉心指导。在我初进实验室时，您多次组织项目例会、课外活动、英语角，极大地帮助了我更快的融入其中。是您带我步入丰富多彩的嵌入式世界，您给我呈现了整个行业形态，开拓了我的视野。您对问题的深入见解，常常引发我新的认知。您时常鼓励我们要有自己的想法和见解，并在适当的时候提出建议，修正我们的想法。

感谢我本科班主任张春元老师在我研究方向和个人发展给予的极大帮助。您无私地将世界范围内研究最新进展与我分享，并协助我找到自己感兴趣的研究方向。您通过向我分析各种实例让我深切地明白了自己的处境和现状，让我少走了很多弯路，您的远见卓识在我人生规划中起到了至关重要的作用。

感谢电子科技大学教导过我以及给予过我帮助的所有老师，尤其是雷航、谢莉钧、吴准、龚心愿老师。感谢项目组所有成员，尤其是钟太聪师姐、周强、许斌。感谢我身边的每一位朋友，尤其是殷诗润、吴美珠、蔡平。因为有你们，我的学习生活更加精彩。

感谢那些在我申请出国攻博道路上帮助过我的人，尤其是赵亮、张海宽，你们给我讲解出国申请的一切细节并给我提供最新资讯，还帮我完善我的申请资料。

感谢评阅本论文的老师，您们建设性的建议使得本文更加完善。

最后，还要感谢我的亲人，是你们们在我漫长求学生涯中一直给予默默的支持和无微不至的关心。

参考文献

[1] 刘国兴, 余镇危. 无线传感器网络综述[J]. International Conference on Internet Technology and Applications, 2011:1-8

[2] 杨卓静，孙宏志，任晨虹.无线传感器网络应用技术综述[J]. 中国科技信息，2010(13):127-129

[3] Jennifer Yick, Biswanath Mukherjee, Dipak Ghosal. Wireless sensor network survey[J]. Computer Network, 2008, 52(12): 2292-2330

[4] 贺慧琳，肖强华. 无线传感器网络数据收集研究综述[J]. 电脑知识与技术，2011(30): 7370-7371

[5] Muhammad Omer Farooq, Thomas Kunz. Operating Systems for Wireless Sensor Networks: A Survey[J]. Sensors, 2011: 5900-5928

[6] 余向阳. 无线传感器网络研究综述[J]. 单片机与嵌入式系统应用, 2008, 9208:8-12

[7] Dunkels, A. Gronvall, B. Voigt, T. Contiki-a lightweight and flexible operating system for tiny networked sensors[C]. 29th Annual IEEE International Conference on Local Computer Networks, 2004: 455-462

[8] Dunkels Adam, Schmidt Oliver, Voigt Thiemo. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems[C]. Proceedings of the Fourth International Conference on Embedded Networked Sensor Systems, 2006:29-42

[9] Adam Dunkels, Oliver Schmidt. Protothreads Lightweight, Stackless Threads in C[R]. Sweden: SICS, 2005

[10] Tsiftes Nicolas, Dunkels Adam, He Zhitao. Enabling large-scale storage in sensor networks with the coffee file system[C]. International Conference on Information Processing in Sensor Networks, 2009:349-360

[11] Hewage Kasun, Keppitiyagama Chamath, Thilakarathna Kenneth. TikiriDev: A UNIX-like device abstraction for Contiki[J], Real-World Wireless Sensor Networks, 2010, 6511:74-81

[12] Tsiftes Nicolas, Eriksson Joakim, Dunkels Adam. Low-power wireless IPv6 routing with ContikiRPL[C]. Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks, 2010:406-407

[13] Hornsby, A. Bail, E. μXMPP: Lightweight implementation for low power operating system

Contiki[C]. International Conference on Ultra Modern Telecommunications & Workshops, 2009:1-5

[14] Paul Tomsy, Kumar G., Santhosh. Safe contiki OS: Type and memory safety for contiki OS[C]. International Conference on Advances in Recent Technologies in Communication and Computing, 2009:169-171

[15] Casado Lander, Tsigas Philippas. ContikiSec: A secure network layer for wireless sensor networks under the Contiki Operating System[C]. Nordic Conference on Secure IT Systems, 2009:133-147

[16] Bruno, L. Franceschinis, M. Pastrone, C. Tomasi, R. Spirito, M. 6LoWDTN: IPv6-enabled Delay-Tolerant WSNs for Contiki[C]. International Conference on Distributed Computing in Sensor Systems and Workshops, 2011:1-6

[17] Adam Dunkels. ContikiWiki[EB/OL]. http://www.sics.se/contiki/wiki/index.php/Main_Page, 2012-06-01

[18] Adam Dunkels. The Contiki OS[EB/OL]. http://www.contiki-os.org/, 2012-12-01

[19] 周皓. IP 传感器网络下的普适计算系统设计[D].上海：上海交通大学, 2010.

[20] Djenouri Djamel, Balasingham Ilangko. Power-aware QoS geographical routing for wireless sensor networks -Implementation using Contiki[C]. International Conference on Distributed Computing in Sensor Systems, 2010

[21] 俄立波. 文件系统在无线传感器网络中的应用研究[D].上海：上海交通大学, 2009

[22] Cao, Q. Abdelzaher, T. Stankovic, J. He, T. The LiteOS Operating System: Towards Unix Like Abstraction for Wireless Sensor Networks[C]. In Proceedings of the 7th International Conference on Information Processing in Sensor Networks (IPSN 2008), St. Louis, MO, USA, 2008: 22–24.

[23] Sun Jun-Zhao. Dissemination protocols for reprogramming wireless sensor networks: A literature survey[C]. International Conference on Sensor Technologies and Applications, 2010:151-156

[24] Sun Jun-Zhao. OS-based reprogramming techniques in wireless sensor networks: A survey[C]. IEEE International Conference on Ubi-Media Computing,2010:17-23

[25] 张集文，李士宁，贾军博，周涛. 无线传感器网络重编程技术的研究与设计[J].计算机测量与控制, 2009, 17(07):1441-1444

[26] 张羽，周兴社，蒋泽军，王丽芳. 传感器网络远程网络重编程服务安全认证机制研究[J].计算机科学, 2008, 35(10):44-47

[27] 蒋泽军, 张英, 王丽芳, 方智毅. 无线传感器网络重编程安全认证机制研究[J].西北工业大学学报, 2009, 27(06):867-870

[28] 张羽, 周兴社, Yee WeiLaw, Marimuthu Palaniswami. 一种抗污染攻击的传感器网络重编程方法[J]. 西北工业大学计算机学院, 2011, 29(03):443-446

[29] 任超,张羽,方智毅.一种基于散列链的无线网络重编程安全认证机制[J].自然科学进展,2009, 19(10):1100-1104

[30] 方智毅, 王丽芳, 张羽, 蒋长清. 无线传感器网络重编程中 DoS 攻击初探[J]. 西北工业大学学报, 2009, 27(05):725-729

[31] 孟硕培. 无线传感器网络节点重编程研究与设计[D].浙江: 浙江大学, 2008

[32] 俄立波. 文件系统在无线传感器网络中的应用研究[D].上海: 上海交通大学, 2009

[33] Adam Dunkels, Fredrik Österlind, Zhitao He. An adaptive communication architecture for wireless sensor networks[C]. Proceedings of the Fifth ACM Conference on Networked Embedded Sensor Systems, 2007:335-349

[34] 苏铅坤, 罗克露, 周强. 基于无线传感器网络文件系统 Coffee 的研究[J]. 计算机技术与发展, 2013, 23(1):67-70

[35] 董玮. 面向无线传感网络的嵌入系操作系统设计[D].浙江：浙江大学, 2010.

[36] Crossbow Technology, Inc. Mote In-Network Programming User Reference Version 20030315[EB/OL]. http://webs.cs.berkeley.edu/tos/tinyosl.x/doc/Xnp.pdf, 2003

[37] Jonathan W. Hui, David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale[C]. The 2nd international conference on Embedded networked sensor systems, Baltimore, Maryland, USA, 2004:81-89

[38] Thanos Stathopoulos, John Heidemann, Deborah Estrin. A Remote Code Update Mechanism for Wireless Sensor Networks[R]. CENS-TR-30, University of California, Los Angeles, Center for Embedded Networked Computing, 2003

[39] S-J. Park, R. Vedantham, R. Sivakumar, I. F.Akyildiz. A scalable approach for reliable downstream data delivery in wireless sensor networks[C]. The ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc), 2004: 78-89.

[40] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, Richard Han. MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms[J]. ACMKluwer Mobile Networks & Applications (MONET), Special Issue on Wireless Sensor Networks, 2005, 10(4):563-579

[41] Niels Reijers, Koen Langendoen. Efficient Code Distribution in Wireless Sensor Networks[C]. The 2nd ACM Int. Workshop on Wireless Sensor Networks and Applications, San Diego, CA, 2003:60-67.

[42] S. Vi, H. Min, Y. Cho, J. Hong. Molecule: An adaptive dynamic reconfiguration scheme for sensor operating systems[J]. Computer Communications, 2008, 31(4):699-707

[43] Jingtong Hu, Chun Jason Xue, Yi He, Edwin H.M.Sha. Reprogramming with Minimal Transferred Data on Wireless Sensor Network[C]. The 6th IEEE International Conference on Mobile Ad Hoc and Sensor Systems (MASS 2009), Macau SAR, P.R.C., 2009:160-167

[44] Rajesh Krishna Panta, Saurabh Bagchi, Samuel P. Midkiff. Zephyr: Efficient Incremental Reprogramming of Sensor Nodes using Function Call Indirections and Difference Computation[C]. Annual Technical Conference (USENIX), San Diego, CA, USA, 2009

[45] J. Koshy, R. Pandey. Remote incremental linking for energy-efficient reprogramming of sensor networks[C]. The second European Workshop on Wireless Sensor Networks, 2005:354-365.

[46] Pedro Jose Marron, Matthias Gauger, Andreas Lachenmann, Daniel Minder, Olga Saukh, Kurt Rothermel. FlexCup: A Flexible and Efficient Code Update Mechanism for Sensor Networks[C]. European Conference on Wireless Sensor Networks, 2006: 212-227

[47] Seung-Ku Kim, Jae-Ho Lee, Kyeong Hur, DooSeop Eom. Tiny Function-Linking for Energy Efficient Reprogramming in Wireless Sensor Networks[C].The Third International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies, 2009: 208-213

[48] Seung-Ku Kim, Jae-Ho Lee, Kyeong Hur, DooSeop Eom. Tiny Function-Linking for Energy Efficient Reprogramming in Wireless Sensor Networks[C]. The Third International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies, 2009: 208-213

[49] S.S. Kulkarni, L. Wang. MNP: Multihop network reprogramming service for sensor networks[R]. Technical Report, Michigan State University, 2004

[50] Jaein Jeong, David E. Culler. Incremental Network Programming for Wireless Sensors[J]. IEEE Sensor and Ad Hoc Communications and Networks(SECON), 2004:25-33

[51] Abrach H, Bhatti S, Carlson J, et al. MANTIS：System Support for Multimodal Networks of In-situ Sensors[C]. The 2nd ACM InternationalWorkshop on Wireless Sensor Networks and Applications, 2003: 50-59

[52] 刘莉，黄海平. 基于 MantisOS 的无线传感器网络应用开发模型[J]. 信息技术，2010, 34(6):127-129.

[53] Österlind Fredrik, Dunkels Adam, Eriksson Joakim, et al. Cross-Level Sensor Network

Simulation with COOJA[C]. The 31st IEEE Conference on Local Computer Networks. Piscataway, NJ: IEEE, 2006: 641-648

[54] 徐顶鑫, 易卫东. 基于 Contiki/COOJA 平台的 Deluge 协议性能测试[J]. 计算机研究与发展,2011, 48(Suppl.):290-294

[55] 潘爱民，俞甲子，石凡. 程序员的自我修养——链接、装载与库[M]. 北京：电子工业出版社. 2009

[56] John R.Levine. 链接器和加载器[M](李勇译). 北京：北京航空航天大学出版社.2009

[57] Jean-Philippe Vasseur, Adam Dunkels. 基于 IP 的物联网架构、技术与应用[M](田辉等译). 北京：人民邮电出版社. 2011

攻硕期间取得的研究成果

[1] 苏铅坤, 罗克露, 周强. 基于无线传感器网络文件系统 Coffee 的研究[J]. 计算机技术与发展, 2013, 23(1):67-70

[2] 周强, 苏铅坤. 基于无线传感器网络的电能监测节点设计[C]. 电子科技大学计算机科学与工程学院硕士生学术论坛. 2012

参与项目：

基于无线传感器网络智能电网用户端电器监控系统

嵌入式多核实时操作系统 aCoral(http://acoral.org/)