分类号：**TP316.2**
密　级：

单位代码：**10433**
学　　号：**Y0705121**

山东理工大学

硕士学位论文

面向 C8051F 的嵌入式操作系统内核设计

DESIGN OF AN EMBEDDED OS KERNEL FACING C8051F

研　　究　　生：丁　帅

指　导　教　师：张景元（教授）

申请学位门类级别：工　学　硕　士

学 科 专 业 名 称：计算机应用技术

研　究　方　向：计算机测控技术

论 文 完 成 日 期：2010 年 4 月 20 日

独 创 性 声 明

本人声明所呈交的论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，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，也不包含为获得山东理工大学或其他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了谢意。

研究生签名：　　时间：　　年　月　日

关于论文使用授权的说明

本人完全了解山东理工大学有关保留、使用学位论文的规定，即：学校有权保留送交论文的复印件和磁盘，允许论文被查阅和借阅；学校可以用不同方式在不同媒体上发表、传播学位论文的全部或部分内容，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协议)

研究生签名：　　时间：　　年　月　日

导师签名：　　时间：　　年　月　日

摘 要

嵌入式操作系统出现于20世纪80年代，它的出现使嵌入式计算机系统在国防、民用等各个领域的应用日益广泛。嵌入式操作系统大大降低了系统的开发强度，缩短了开发周期，便于系统维护和二次开发，同时也提高了嵌入式系统的稳定性和可靠性。

本文的研究内容是设计应用于具体硬件环境的嵌入式操作系统。研究、分析了 μC/OS-Ⅱ实时操作系统的内核结构并对其加以改进，设计实现了一种面向C8051F的嵌入式操作系统内核micro-K。micro-K适用于中小型测控系统，结构精巧，兼容性强，可移植，可扩展。

在 micro-K 的设计过程中，详细剖析了 μC/OS-Ⅱ实时内核的任务管理，优化了优先级位图调度算法，在保证内核已有的特性和优点不受影响的前提下，将 μC/OS-Ⅱ支持的最大优先级数由原有的64个扩展到micro-K内核所需的128个。在分析原有任务调度算法和RTOS中经典调度策略的基础上，对内核的调度机制进行改进，优化了 μC/OS-Ⅱ任务优先级的分配策略，提出了micro-K内核的 HP 算法，这种算法支持多个任务具有相同的优先级，不同级的任务采用原有的静态任务调度方式，而同级的任务则采用动态调度机制来处理。这样既解决了优先级反转的问题，又有效地改善了任务的调度性能。

micro-K仅仅是一个抢占式内核，只有一些基本的功能，如进程调度、任务管理以及内存管理等，不具备用户接口、文件管理和网络通信功能，因此需要对内核进行功能扩展。设计了micro-K内核的外设驱动程序管理层PDML，利用 PDML 对设备进行统一管理，并在C8051F120硬件平台上对 micro-K 内核进行测试。实验证明这种实时内核除了具有抢占式内核的优点之外，功能更完善，内核更灵活、高效。

关键词：嵌入式操作系统；micro-K；HP 算法；C8051F120

Abstract

Embedded operating system appeares in the 1980s,it makes the embedded computer systems applied in various fields such as national defense and civil use.The embedded operating system(EOS) has been used widely.EOS reduces the intensity of system development significantly and shortens the development cycle.It is advantageous not only for system maintenance and secondary development,but for improving the stability and reliability of the embedded system.

The research contents of this dissertation is to design an embedded operating system for specific hardware environment.It researched and analyzed the structure of real-time operating system kernel μC/OS-Ⅱ,and also improved the kernel structure.The dissertation designed and implemented an embedded operating system kernel named micro-K facing C8051F Single Chip Microcomputer.The micro-K OS which had compact structure could be applied in small or medium control system.It was compatible,portable and extensional.

In the design process of micro-K,the dissertation analyzed the task management of μC/OS-Ⅱ real-time kernel in detail,and optimized the priority bitmap scheduling algorithm.Under the premise of ensuring the features and advantages of μC/OS-Ⅱ kernel was not affected,the dissertation extended the kernel's priority numbers from 64 supporting by μC/OS-Ⅱ kernel to 128 required by micro- K kernel.Based on the analysis of the original task scheduling algorithm and the classical scheduling strategy,the dissertation improved μC/OS-Ⅱ kernel's scheduling mechanism,optimized the task priority allocation strategy of μC/OS-Ⅱ,proposed HP algorithm for micro-K kernel.HP algorithm allowed multiple tasks with the same priority,and the different level of tasks used the original static scheduling while at the same level of tasks was to be handled adopting dynamic dispatching mechanism.The algorithm not only solved the priority inversion,but also improved task scheduling performance.

Micro-K was just a preemptive kernel,and it only owned some basic functions such as process scheduling,memory management and task management.As the kernel did not have user interface,document management and network communication,there would be the need for kernel function expansion.The dissertation designed peripheral driver management (PDML) for micro-K kernel to

realize the unified equipment management,and tested micro-K kernel on C8051F120 hardware platform.The experiments showed that in addition to possessing the advantages of preemptive kernel,micro-K kernel's function was more perfect,and the kernel was more flexible and efficient.

Key words： embedded operating system; micro-K; HP algorithm; C8051F120

目 录

摘 要 I
Abstract II
目 录 V
第一章 绪论 1
1.1 论文选题背景及意义 1
1.2 国内外研究现状 2
1.3 发展历程与趋势 3
1.4 论文主要内容及结构安排 3
第二章 嵌入式操作系统体系架构 5
2.1 嵌入式实时操作系统 5
2.2 嵌入式操作系统体系架构 6
2.2.1 单块结构 6
2.2.2 层次结构 6
2.2.3 微内核结构(客户/服务器) 7
2.3 嵌入式操作系统的设计 10
2.4 本章小结 10
第三章 micro-K 嵌入式操作系统内核 11
3.1 硬件环境 11
3.2 μC/OS-Ⅱ简介 12
3.2.1 μC/OS-Ⅱ内核特点 12
3.2.2 μC/OS-Ⅱ功能结构 13
3.2.3 μC/OS-Ⅱ工作原理 14
3.2.4 任务优先级及其分配 14
3.2.5 任务调度 15
3.3 micro-K 操作系统 16
3.3.1 micro-K 的体系架构 16
3.3.2 micro-K 的多任务管理 17
3.4 本章小结 20
第四章 micro-K 任务调度算法的设计与实现 21
4.1 优先级位图调度算法 21
4.1.1 任务就绪表 OSRdyTbl[] 21

4.1.2 任务优先级到 OSRdyGrp 的优先级映射表 OSMapTbl[8] ... 23
4.1.3 优先级判定表 OSUnMapTbl[] ... 23
4.1.4 优先级判定表具体的算法分析 ... 24
4.1.5 任务进入就绪态 ... 25
4.1.6 任务退出就绪态 ... 25
4.1.7 获取进入就绪态的最高优先级 ... 25
4.1.8 根据优先级获取相应的任务 ... 26
4.2 深度优先级调度算法 ... 26
4.2.1 设计方案比较 ... 26
4.2.2 深度优先级调度算法 ... 29
4.3 深度优先级调度算法的性能分析 ... 33
4.4 本章小结 ... 34
第五章 micro-K 任务分配策略的设计与实现 ... 35
5.1 动态任务调度算法 ... 35
5.1.1 EDF(最早时限优先)调度算法 ... 36
5.1.2 LSF(最短松弛时间优先)调度算法 ... 37
5.1.3 HVF(价值最高优先)调度算法 ... 37
5.1.4 HVDF(价值密度最大最优先)调度算法 ... 37
5.2 调度策略的改进 ... 37
5.2.1 调度算法的选择 ... 38
5.2.2 任务模型 ... 38
5.2.3 任务优先级的设计 ... 39
5.2.4 任务优先级的分配 ... 40
5.2.5 任务优先级的获取 ... 41
5.2.6 调度实现 ... 41
5.3 算法评估 ... 46
5.4 本章小结 ... 46
第六章 设备驱动管理设计 ... 47
6.1 外设驱动管理层设计 ... 47
6.1.1 PDML 架构 ... 47
6.1.2 外设驱动索引表 ... 48
6.1.3 驱动程序表 ... 48
6.1.4 PDML 中的主要函数 ... 50
6.1.5 PDML 工作原理 ... 51
6.2 底层接口的具体实现 ... 52

6.2.1 C8051F120 单片机中的 A/D 转换器 52
6.2.2 micro-K 下 A/D 驱动 53
6.2.3 micro-K 下串口驱动 55
6.3 设备驱动程序的编写原则 59
6.4 本章小结 60
第七章 针对 micro-K 内核的性能测试 61
7.1 测试环境 61
7.2 micro-K 在测试平台上的加载过程 62
7.2.1 mK_CPU.H 文件的修改 62
7.2.2 mK_CPU_C.C 文件的修改 63
7.2.3 mK_CPU_A.ASM 文件的修改 64
7.3 micro-K 内核的启动与调试 65
7.4 测试结果和性能分析 65
7.4.1 micro-K 内核任务数的测试 66
7.4.2 micro-K 实时性能的测试 67
7.4.3 HP 调度算法性能测试 68
7.5 本章小结 71
第八章 总结与展望 73
8.1 总结 73
8.2 展望 73
参考文献 75
致谢 79
在学期间公开发表论文及科研情况 80

第一章 绪论

1.1 论文选题背景及意义

自嵌入式操作系统出现后，各种嵌入式产品越来越广泛地应用于民用以及工业生产中。迄今为止，嵌入式操作系统的设计与开发依然是 IT 行业中的技术热点。早期没有嵌入操作系统的控制系统是通过工作人员编写的代码来控制硬件，这种系统的结构和功能都相对单一，处理效率较低，存储容量较小，而且几乎没有用户接口。当将操作系统应用于控制系统之后，就可以调用操作系统的 API 来控制软、硬件资源，并能使多条线程同时工作。较之无操作系统的单线程，使用操作系统能够有效管理日趋复杂的系统资源；能够把硬件虚拟化，使得开发人员从繁忙的驱动程序移植和维护中解脱出来；能够提供库函数、驱动程序、工具集以及应用程序；能使任务的设计变得更加简单、方便，提高应用系统的可靠性。

伴随着内置处理器性能的不断提高，嵌入式控制系统逐渐趋向于高端化，中高端的单片机中广泛使用了嵌入式操作系统。然而实际中大量使用的低端单片机（以 8 位单片机为主），由于其内存小，需要进行内存扩展，所以设计者使用嵌入式 OS 的意识不强，也很少有人进行这类单片机的嵌入式开发。从全球范围来看，内置低端单片机的测控系统应用非常广泛，占据着很大的市场份额，但是国内外绝大多数的嵌入式操作系统大多不能应用于低端单片机，使得这类系统的稳定性能和可靠性能都不高，这在某种程度上严重制约了嵌入式软件产业的发展，因此设计实现面向低端测控系统的嵌入式操作系统具有重要的意义。

目前，市面上有很多优秀的嵌入式操作系统，如 VxWorks、PalmOS、Windows CE、Linux、VRTX 等[1,2]。这些操作系统均属于商品化产品，价格昂贵且源代码不公开，这些因素导致了对设备的支持、应用软件的移植等一系列的问题。虽然商业操作系统的功能十分强大，但是其中的许多功能并不是低端应用所需要的，所以在中小型系统中应用这些商用操作系统并不划算。在实际应用中，越来越多的人将目光投向于开源的操作系统，相对于商用操作系统，开源操作系统无论在成本还是技术上都有其特有的优势。

设计构建一个完整的实时操作系统，如果从底层开始，就需要做大量的重复性劳动，这样做不但工作量大、效率低，且由于系统的功能没有经过严格的验证而缺乏稳定性。由于开源性的操作系统其性能都经过大量的检验，具备了

很高的可靠性和稳定性，因此可利用现有的开源嵌入式操作系统做参照，通过修改操作系统的源代码来改善内核的功能以满足不同系统的要求[3]。在开发过程中，用户只需要编写各个任务的程序，不必记住所有任务运行的各种情况，这既减轻了程序编写的工作量，也减少了出错的可能性。因此开发面向特定应用的嵌入式操作系统将具有重要的意义。

1.2 国内外研究现状

从世界上第 1 个商业嵌入式实时内核(VRTX32)到今天，嵌入式操作系统已有近 30 年的历史。最初的嵌入式操作系统还只支持一些 16 位的微处理器，如 8086 等。近几年来，嵌入式操作系统的发展趋势更加强劲，出现了适用于不同应用领域的产品。目前，全球嵌入式操作系统的发展空间更是随着互联网、通讯和计算机市场的飞速增长而不断的扩大。

国外的嵌入式操作系统已经从简单走向成熟，对嵌入式实时操作系统的研究主要倾向于：多处理器结构、分布式实时操作系统和实时网络的研究，集成、开放式的实时系统开发环境研究以及规范化研究。国外的嵌入式操作系统总体上可以分为两类，一类以 WindRiverSystem 公司的 VxWorks 为代表的商用嵌入式操作系统，其他的还有 Windows CE、MacOS X、Symbian OS、QNX 和 PalmOS 等。另一类则是以嵌入式 Linux 为代表的开源系统，包括 T-Kernel、eCos、μC/OS-Ⅱ、RT-Linux 和 Thread X 等等[4]。其中 μC/OS-Ⅱ发布于 1999 年，并于 2000 年被美国航空航天管理局(FAA)认证，μC/OS-Ⅱ的代码是开源的，是面向低端应用的最经济的嵌入式实时操作系统。

我国的嵌入式操作系统研发基础薄弱，研究开发的嵌入式操作系统主要有两种：一种是具有完全自主知识产权的商用嵌入式操作系统，如女娲 Hopen、Delta OS、夏桑 3000 等等；另一种是基于 Linux 的嵌入式操作系统，如红旗嵌入式 Linux、中软 Linux、东方 Linux 等等。然而我国的软件产业却大而不强，它的工业增加值和利润均不高。2009 年我国软件行业系统集成收入为 2202.9 亿元，嵌入式软件收入仅为 1673.6 亿元。由于缺乏核心技术和知识产权，大多数中国软件企业只是在重复简单的装配和加工工作，因此中国的软件行业亟需掌握核心技术和知识产权。

中国近几年来的嵌入式系统软硬件市场增长迅速。《国家中长期科学和技术发展规划纲要(2006-2020 年)》专项中加强了对嵌入式操作系统/嵌入式软件的支持。随着微电子技术和嵌入式操作系统的发展，在未来的几年里，中国乃至世界的电子消费市场将越来越有活力，嵌入式系统将在人们的日常生活中占据更为广阔的领域。

1.3 发展历程与趋势

从早期的监控式操作系统到各类通用/专用的嵌入式操作系统，再到特定应用的嵌入式操作系统ASOS(application specific operating systems)，嵌入式操作系统的应用领域一直在不断细分，运用嵌入式操作系统的产品丰富多样，嵌入式操作系统在今后仍有广阔的研发空间。

随着嵌入式系统软硬件趋向复杂化、多样化和智能化，嵌入式操作系统多应用于高性价比的高端硬件平台。而低端单片(8位)机从20世纪70年代初期诞生至今，其市场占有率仍然很大，中小型控制系统的市场需求仍然会在未来相当长的时间里居高不下，而嵌入式操作系统在产业发展过程中将会发挥越来越重要的作用。今后也可能出现性能更高的低端单片机，而且将出现基于这类单片机的“通用”嵌入式操作系统。

针对具体的CPU，嵌入式操作系统的商业版本有很多种。目前各种开源的嵌入式操作系统可运行在通用机上，今后还将研发出更多的基于各种低端型号单片机的具备图形，文件网络功能的商业化产品，也会将更多的操作系统嵌入低端单片机，开发出更加可靠的嵌入式系统。

1.4 论文主要内容及结构安排

本文的研究内容主要是通过分析、研究μC/OS-Ⅱ实时操作系统内核结构，提出一种新的嵌入式操作系统内核的设计方案，使其能运行于硬件平台(本文采用基于C8051F120单片机的硬件平台)。通过进一步深入研究，掌握嵌入式操作系统的设计方法，进而研究设计适用于测控领域低端的MCU的嵌入式实时多任务微内核。

本文的研究内容主要是：

全文共分八章，各章研究内容安排如下：

第一章　绪论。提出课题研究目的和意义，对嵌入式操作系统的发展和现状做简明介绍，明确课题目标和所要完成的工作。

第二章　嵌入式操作系统体系架构。介绍了嵌入式实时操作系统的系统架构，更深层次了解实时嵌入式多任务微内核的整体概念。

第三章　micro-K嵌入式操作系统内核。结合μC/OS-Ⅱ内核，提出micro-K嵌入式操作系统微内核的设计方案。

第四章　micro-K任务调度算法的设计与实现。通过分析μC/OS-Ⅱ内核的调度算法，给出了micro-K任务调度算法的设计方案，将任务数从μC/OS-Ⅱ所支持的64个扩充到micro-K所需的128个。

第五章　micro-K任务分配策略的设计与实现。将静态优先级调度算法改进为支持动静态优先级的调度算法，优化micro-K中任务优先级的分配策略。

第六章　设备驱动管理设计。针对目标硬件，添加操作系统的外围设备驱动功能。

第七章　操作系统的性能测试。介绍实验的硬、软件平台，并针对部分功能进行系统性能实验。

第八章　总结与展望。对所研究课题进行总结，展望以后的研究工作。

第二章 嵌入式操作系统体系架构

嵌入式产品是信息产业中一个强劲的增长点，伴随着人们的需求日益增多，嵌入式系统也越来越复杂，实时操作系统(RTOS)在嵌入式设备中的使用也随之增多。利用实时操作系统可以将一个复杂的工程任务分解成多个分任务系统，并有效管理越来越复杂的系统资源；提供库函数、驱动程序、工具集以及应用程序，使硬件虚拟化。操作系统使开发人员不必从事繁忙的驱动程序移植和系统维护，大大提高了系统的可靠性和稳定性。

2.1 嵌入式实时操作系统

嵌入式操作系统即应用于嵌入式系统的操作系统。与通用操作系统相比，其优势在于系统实时高效性、系统稳定性、硬件的相关依赖性、软件固态化以及应用的专用性[5,6]。早期嵌入式操作系统的应用领域很有限，品种也不多。随着应用的扩展，目前已出现了针对不同领域的嵌入式操作系统。

实时系统(real-time system)[7,8]是指有一定时间约束的计算机系统。实时系统在嵌入式系统中占有相当大的比例，一般的嵌入式系统都是实时系统。嵌入式实时操作系统(Embedded Real Time Operating System)[9,10]出现于20世纪80年代初，嵌入式系统一般都采用实时操作系统，有时嵌入式实时操作系统又称为嵌入式操作系统。

嵌入式实时操作系统属于系统软件，在嵌入式系统结构中处于应用软件与计算机硬件之间，负责下层各种硬件资源进行控制和管理，对操作系统和应用提供所需驱动的支持，并为上层应用提供服务，其特点主要有以下几个方面：

1. 使系统的执行时间可确定

使用实时操作系统的目的就是提高计算机的执行效率。设计实时操作系统必须遵循的原则就是采用各种算法和策略，始终保证系统能迅速地对外界做出反应[11]。

2. 为上层应用提供多种支持

嵌入式实时操作系统一般都提供了丰富的系统调用，方便了应用程序的开发。

3. 可剪裁、可移植

用户可以根据应用的需要对操作系统灵活地进行裁剪及扩充，将操作系统移植到形态各异的硬件平台上。

4. 虚拟化硬件

嵌入式实时操作系统为底层硬件和用户提供了一些驱动支持和系统服务函数，使开发人员不用对驱动程序进行移植和维护，提高了开发效率。

当用户调用应用程序时，若在其他外围设备的工作期间CPU没有其他的任务，那么就会延长计算机运行应用程序的时间，这种系统的实时性就会较差。若能采用多任务系统进一步细分设备需求，则所有的需求都会被系统中的任务关注，使系统的实时性发挥到最大化[12]。多任务的架构能显著提高CPU的利用率，并使应用程序模块化，减少系统执行任务所花的时间，同时使应用程序更容易设计、维护。

在多任务系统中，任务只处理相关的事件，因此有很好的实时性和可维护性。但是缺点是操作系统设计相对复杂，需要高效的调度算法。多任务的系统架构是嵌入式操作系统的必然选择，也是最佳选择，当前流行的嵌入式操作系统大都选择了这种系统架构。

2.2 嵌入式操作系统体系架构

体系架构是操作系统的基础，它定义了软硬件的界限、内核与操作系统其他组件的组织关系以及系统与硬件的接口。操作系统的体系架构关系到系统的实时性、可靠性、稳定性、可移植性和可扩展性，因此选择哪种操作系统的体系架构实现预期要求是设计嵌入式系统时首先应该考虑的问题。目前，操作系统的体系架构主要有：单块结构、层次结构和微内核(客户/服务器)结构。

2.2.1 单块结构

单块结构的操作系统由许多模块组成，这些模块通过相互调用共同完成系统的各项功能。单块结构的操作系统结构紧密，模块之间的接口比较简单，系统的效率不是很低。但是由于各模块互相关联，独立性差。一旦某一个模块被添加或修改，其他的模块都会受到影响。且随着模块数量的增加和模块之间连接(特别是多重关联)的增多，系统在调用时也会变得更加混乱[13]。单块的架构模型如图2.1所示。

2.2.2 层次结构

层次结构的操作系统对各模块进行分层，减少了模块之间的互相调用及互相依赖的关系，使模块间的调用变得有规则。各模块间的调用、组织和依赖关

系比较清晰，每个系统调用都可以直接到达每一层，甚至直接到达硬件层。当对某一层进行操作时，最多会影响到邻近的两层，不会对其他层造成任何影响，系统的安全性、稳定性以及可靠性都大大增强。

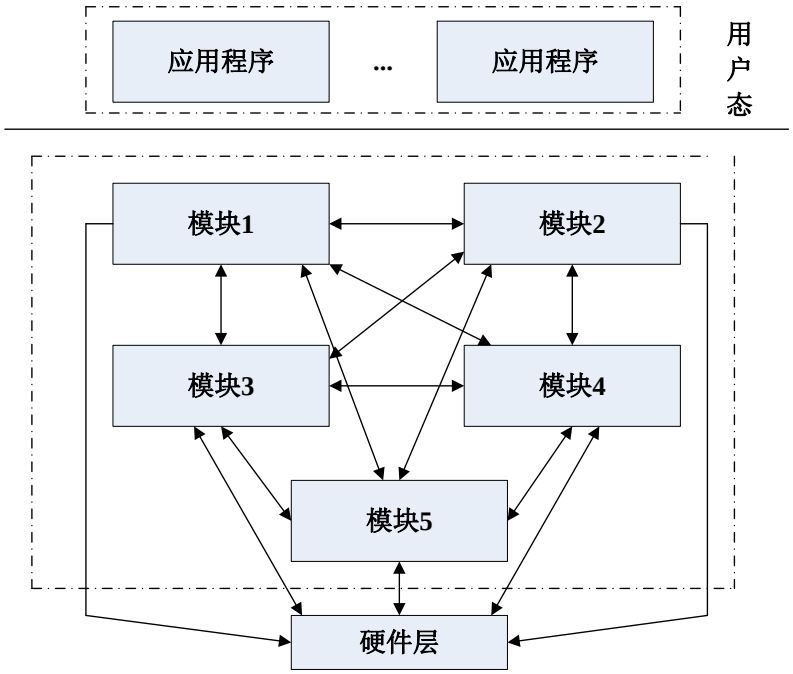

图2.1 操作系统的单块结构

在层次结构中，通常把与硬件底层联系紧密相关的软件(如硬件驱动、中断处理、I/O管理)放在结构的最底层，将最常用的系统操作放在最里层，而将那些由于系统操作方式而改变的部分放在最外层。其架构模型如图2.2所示。

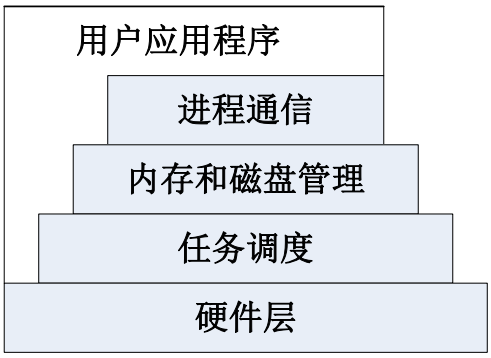

图2.2 操作系统的层次结构

2.2.3 微内核结构(客户/服务器)

1. 微内核结构

RTOS微内核的思想出现于20世纪90年代初期。所谓微内核技术是指将传统操作系统中最核心的、必不可少的功能集合，例如任务管理、中断处理、进程调度、虚拟存储、消息传递、设备驱动以及内核的原语操作集等抽象出来，

构成操作系统的公共基础即内核，而操作系统中那些扩展的、非必要功能和服务则作为可配置部分放到内核之外，具体的操作系统功能则由构造在微内核之外的服务器实现，这样使用户程序及上层操作系统组件对系统设备透明，也避免了出现系统故障。

基于微内核结构的操作系统分为两大部分：核心态的微内核和用户态的客户/服务器。它的架构模型如图2.3所示。

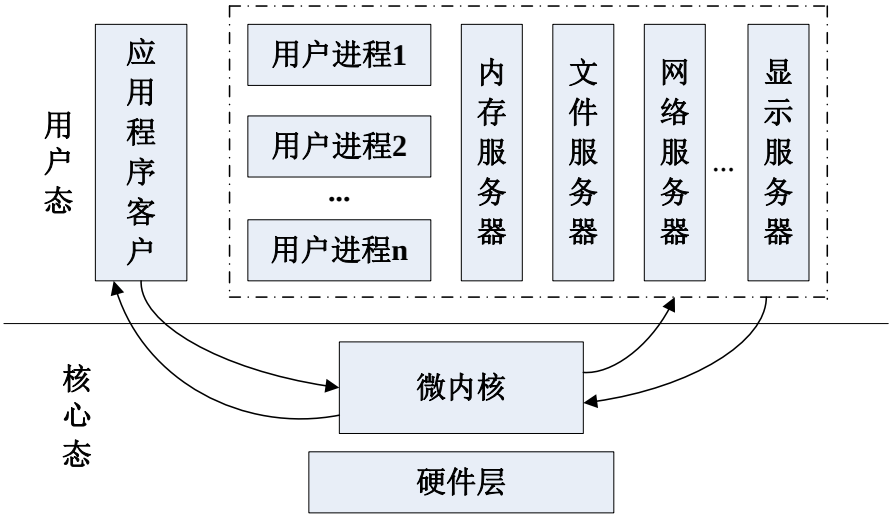

图2.3 操作系统的微内核结构

微内核运行在核心态，其功能组合通常采用层次结构，它的主要工作是由内核将系统的各个系统调用和服务发消息到不同的服务进程，处理进程调用与服务器进程之间的通信，服务进程执行完相应的操作后以消息的方式返回内核。除微内核之外，操作系统的外层部分被分成若干个相对独立的进程，每一个进程实现一组服务，这些进程可以提供各种系统功能，如文件系统、网络系统、设备管理等。操作系统中的每个进程之间不能进行直接通信，客户进程必须通过内核发送消息才能与服务器进程通信。

2. 微内核的功能与特点

微内核是操作系统的核心部分，内核不能建立新的任务，内核为每个任务分配CPU时间，并负责任务之间的通信。微内核需要完成的基本功能[14]有：

(1) 任务管理

嵌入式内核的核心部分是任务的管理。实时操作系统支持多任务管理，任务管理负责任务的创建、调度、挂起、恢复、删除以及优先级的设置与改变等。

(2) 存储器管理

虚拟存储管理功能为系统中的进程分配了必要的运行空间，从逻辑上扩充了内存的容量。内存管理还包括对内存的优化分配，减少整个系统的内存占有量。

(3) 任务间通信、同步和互斥

实时操作系统任务间的通信、同步和互斥机制包括消息、事件、信号量、管道、异步信号、邮箱等。这几种方式相互补充、相互配合，协同完成任务。

(4) 设备管理

微内核为每个外设提供相应的设备驱动程序，用以实现设备的I/O处理。

(5) 时钟管理

时钟管理提供高精度、灵活的系统时钟，还负责与时间相关的任务管理工作，包括系统时钟管理、周期时钟处理、报警时钟处理等。

(6) 中断管理

中断管理服务包括设置中断处理的优先级、定义系统中断的处理过程、向系统申请、注册中断处理函数等。

采用微内核方式结构的操作系统一般只有几十到几百KB，便于直接存放于ROM中，可灵活剪裁。微内核结构降低了操作系统的复杂程度，使操作系统的结构更清晰，提高了操作系统的可靠性。嵌入式操作系统只需稍加修改微内核中与硬件相关的部分就能稳定地运行在新的硬件平台上，容易实现不同平台间的移植。嵌入式系统通常是面向某个特定应用的，操作系统的微内核结构使得用户可以根据自己的实际需要，增删服务功能，在内核外围扩展相应的功能模块，且修改服务器的代码不会影响系统的其他部分，维护起来相当方便。

3. 嵌入式微内核和嵌入式操作系统的关系

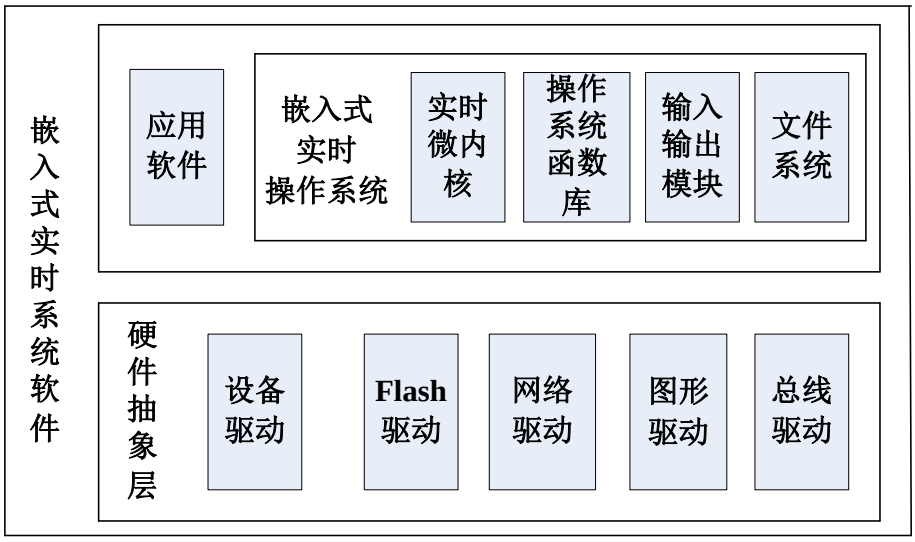

图2.4 嵌入式实时内核与嵌入式实时操作系统的关系

嵌入式微内核和嵌入式操作系统的关系[15]如图 2.4 所示。由图 2.4 可以得知：微内核是嵌入式操作系统的核心，它是模块化扩展的基础。系统整体上采用分层结构，层与层之间相互独立，功能清楚，下层为上层提供功能支持，上层依靠下层提供的功能完成所需的操作。在每一个独立的层中，有很多系统模块，其功能也是相对独立的。微内核处于核心层，它负责实现最基本的功能。

微内核并不是一个密不可分的整体，其内部的设计也是采用模块化，各个功能模块相对独立，可以根据应用需要对其进行裁减、修改与扩充。这种微内核的体系结构充分结合了层次结构和模块结构的可取之处，使整个系统的体系结构更加完整、清晰，对于提高系统的可靠性、实时性和确定性都是至关重要的。

2.3 嵌入式操作系统的设计

系统效率问题是微内核结构首先应该考虑的问题。一方面，由于微内核维系着所有用户进程之间的消息通信，这使得微内核本身就成为系统的瓶颈。当系统进程之间频繁进行通信时，系统的效率往往是比较低的。另一方面，消息机制本身也需要很大的系统开销。由于微内核和服务进程分别处于不同的地址空间，当微内核和服务进程间进行任务切换时，频繁的操作将引起上下文的切换，所以在微内核结构操作系统中其通信开销往往是传统整体结构的数十倍。

如何确定微内核的功能范围也是微内核结构设计中较为关键的问题。是否选择将某个功能写入微内核，取决于用户想要设计实现的特定的操作系统。如果某项功能在应用系统中被频繁使用到(比如中断处理、任务调度、设备驱动等等)，则可应将其写入微内核中。

2.4 本章小结

本章介绍了嵌入式实时操作系统的系统架构，引出了嵌入式实时多任务微内核，并对它和操作系统的关系进行了分析，对实时嵌入式多任务微内核的整体概念有了更深层次的了解，最后探讨了嵌入式系统设计时应该考虑的因素。这些理论依据是设计嵌入式操作系统的基础。

第三章 micro-K嵌入式操作系统内核

micro-K 是面向 C8051F 平台的嵌入式操作系统内核，在其设计过程中需要考虑 C8051F 单片机的硬件条件以及具体的实现技术等相关因素。本章通过对 μC/OS-Ⅱ嵌入式实时操作系统的简要分析，提出了 micro-K 嵌入式操作系统内核的体系架构和设计方案。

3.1 硬件环境

利用 Silicon Laboratories 公司研发的 C8051F 系列高速 SoC 单片机作为控制核心，选用同系列产品中性价比较高的 C8051F120。C8051F120 是真正能独立工作的混合信号片上系统级 MCU 器件，片内集成了数据采集和控制系统中常用的模拟、数字外设及其它功能部件，C8051F120 的片内资源主要有[16]:

- 具有与 MCS-51 内核及指令集完全兼容的高速、流水线结构，单周期指令速度可提高到 MCS-51 的 12 倍，最高处理能力为 100MIPS
- 100ksps 逐次逼近型(SAR)8 通道 ADC
- 两路 12Bit DAC，可同步输出，用于产生无抖动波形
- 128KB 的 FLASH 程序存储器，支持在系统编程
- 8448(8K XRAM+256 RAM)字节的片内数据 RAM
- 64 个 I/O 引脚
- 2 路全双工增强型 UART 接口
- 4 个通用的 16 位定时/计数器
- 2 个指令周期的 16×16 的硬件乘法、加法器
- 6 路 PWM 通道

另外片上还集成了内部电压基准，片内电源监视、降压检测、看门狗定时器和温度传感器、SMBus/I^2C、SPI 串行接口等。C8051F120 采用 JTAG 调试方式，通过 JTAG 接口实现对单片机闪存的读写操作以及在系统调试，不需要仿真器。图 3.1 为 C8051F120 的内部结构图。

C8051F120 功能强大，集成的外设较多，且是 51 指令集的单片机中主频最高的，CPU 的运行速度较快。片内有足够的存储空间来存储数据和程序，其 128KB 的 Flash 程序存储器对于大部分应用系统而言，容量基本足够，不需要外扩内存就能运行操作系统。

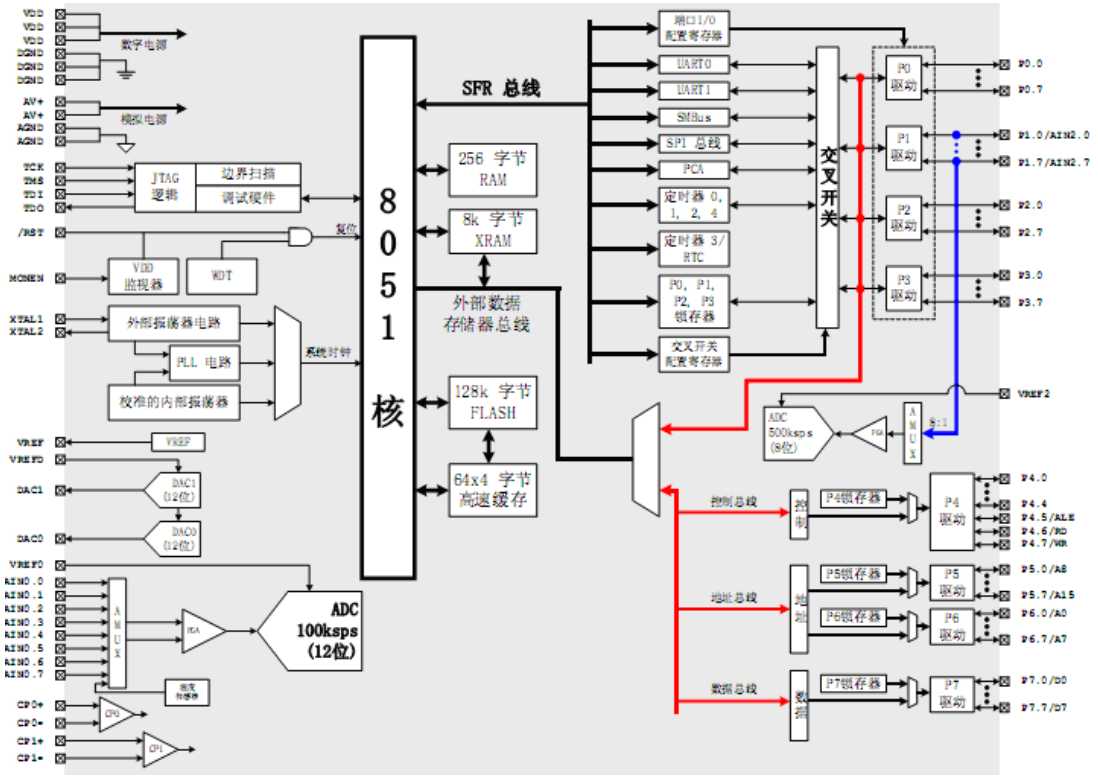

图 3.1 C8051F120 内部结构图

3.2 μC/OS-Ⅱ简介

μC/OS-Ⅱ前身是μC/OS，由美国嵌入式系统专家Jean J.Labrosse在1992年发布。此后又对第一版进行改进与升级，推出μC/OS的第二版μC/OS-Ⅱ[17,18]。μC/OS-Ⅱ精巧、实用，一经推出便受到业界的普遍欢迎。

μC/OS-Ⅱ是一种免费公开源代码、结构小巧、采用可剥夺型实时内核的实时操作系统，它提供了60多个系统调用，目标是实现一个基于优先级调度的抢占式实时内核，并在这个内核之上提供最基本的系统服务，包括信号量，邮箱队列，事件标志组，任务调度与管理，时间管理，中断服务和内存管理等。

μC/OS-Ⅱ是专门为计算机嵌入式应用而设计的，主要面向于中小型控制系统。绝大部分代码用C语言编写，只有那些与处理器硬件相关的代码(约200行)用汇编语言编写，便于移植到其它任意的CPU上。严格地说，μC/OS-Ⅱ只是一个实时内核，仅仅由一些基本功能组成。但由于μC/OS-Ⅱ良好的可扩展性，I/O管理，文件系统，网络等额外的服务功能完全可以由用户自己根据需要各自实现。

3.2.1 μC/OS-Ⅱ内核特点

作为一个实时内核，μC/OS-Ⅱ有以下几个显著特征[19]：

- 免费，源代码公开。
- 实时性强。由于在μC/OS-Ⅱ中所有的任务都有唯一的优先级，因此适合于实时性要求较强的系统。
- 精简性。μC/OS-Ⅱ是一个源码公开的实时嵌入式操作系统，它提供了实时系统所需要的必备功能，其包含全部功能的核心代码只有8.3K字节。由于μC/OS-Ⅱ是可裁剪的，所以实际系统中的代码最少可达2.7KB。
- 可确定性。μC/OS-Ⅱ中全部服务的调度/执行时间是可知的，即μC/OS-Ⅱ系统服务的执行时间不依赖于应用程序任务的数量。
- 多任务。μC/OS-Ⅱ可以管理64个任务，应用程序最多可以使用56个任务。
- 可裁剪、可固化。μC/OS-Ⅱ由多个相对独立的功能模块组成，用户可以根据需求对这些模块裁剪。此外用户还可以针对具体的硬件系统来优化代码，以获得更好的性能。
- 可移植性。μC/OS-Ⅱ的大部分代码是用ANSI C编写，可以运行在大多数8位、16位、32位的微处理器和数字信号处理器上。

μC/OS-Ⅱ最主要的特点就是源码公开,用户可以根据自己的需要对它进行裁剪、修改或优化。利用 μC/OS-Ⅱ的提供任务管理，事件管理等应用程序接口，可将 μC/OS-Ⅱ直接应用到具体系统程序中。同时以 μC/OS-Ⅱ为基础，针对实时操作系统的要求来扩充设计其他的功能模块，使系统具有较高的实时性。然而 μC/OS-Ⅱ缺乏必要的支持，没有功能强大的软件包，用户通常需要自己编写硬件的驱动程序及移植程序[20]。

μC/OS-Ⅱ内核采用抢占式多任务方式，支持信号量，消息队列等任务间通讯和同步方式，具有执行效率高、占用空间小和可扩展性强等特点。

3.2.2 μC/OS-Ⅱ功能结构

μC/OS-Ⅱ的实时内核大致包括：核心功能、任务管理、时间处理、事件管理、时间管理、任务同步与通信及CPU的移植等功能块。

1. 核心功能OS_CORE. C：是操作系统的核心功能，由一组反映系统状态的全局变量和发挥核心功能的函数构成，包括操作系统的任务调度、事件处理初始化及运行等部分,这些程序保证了系统维持基本的工作。
2. 任务处理OS_TASK. C：包括任务堆栈以及任务的创建、删除、挂起、恢复等操作。可调用OS_EVENT. C中的相关操作实现任务间的通信。

3．事件管理OS_EVENT. C：μC/OS-Ⅱ中事件分三种：信号量、邮箱和队列，当事件进行申请或释放时要改变核心功能中任务就绪表的相应值。
4．时间管理OS_TIME. C：以一组时间控制函数为基础构成，完成任务延时等的操作。其中的延时函数执行时将改变CORE. C中的任务就绪表。
5．任务同步和通信：包括信号量、邮箱、邮箱队列、事件标志等部分，主要用于任务间的互相联系和对临界资源的访问。
6．CPU的移值部分：μC/OS-Ⅱ是一个可以运行在PC通用机上的操作系统，应用到嵌入式系统中时需要根据硬件平台中CPU的具体内容和要求作相应的移植。

3.2.3 μC/OS-Ⅱ工作原理

μC/OS-Ⅱ的核心工作原理是：总是运行最高优先级的就绪任务。系统首先初始化MCU，然后初始化操作系统。操作系统的初始化主要包括初始化任务控制块，任务控制块优先级表，任务控制块链表，事件控制块（ECB）以及空任务的创建等。初始化完毕开始创建新任务，并可在新创建的任务中衍生其他的新任务。最后，调用OSStart()函数启动多任务调度。开始多任务调度之后，时钟节拍源也开始计时，它提供周期性的时钟中断信号给系统，用来实现系统延时和超时确认。

3.2.4 任务优先级及其分配

μC/OS-Ⅱ采用静态优先级，最多可以支持 64 个任务，每个任务被赋予不同的优先级。进程中的任务数由“#define constant”语句定义。由于 4 个最高优先级的任务和 4 个最低优先级的任务留出供以后的版本使用，所以用户可以定义 56 个任务使用。在 μC/OS-Ⅱ中，任务的优先级编号就是任务编号 ID。

μC/OS-Ⅱ采用单调执行率调度法RMS（Rate Monotonic Scheduling）分配任务优先级。RMS算法优先执行系统中周期性的、可抢先的任务，有较小的运行开销[21]。RMS算法可调度性的充要条件是：对于系统中n个不同任务，要使所有的任务满足硬实时条件，必须使不等式3.1成立。

$$\sum_{i}\frac{Ei}{Ti}\le n(2^{1/n}-1) \tag{3.1}$$

这里 Ei 是任务 i 最长执行时间，Ti 是任务 i 的执行周期，Ei/Ti 是任务 i 所需的 CPU 时间。RMS 算法为进程分配的优先级与事件的发生频率成正比，即任务的周期越小，优先级越大[22]。

3.2.5 任务调度

μC/OS-Ⅱ内核只支持基于优先级的抢占式调度算法，不支持时间片轮转。任务调度不支持优先级逆转，每个任务的优先级都不相同且是静态的。任务调度采用位图调度算法，算法简单、效率高。其调度算法的复杂度为O(1)，任务调度的时间是可知的，用户程序所建立的任务总数(OS_MAX_TASKS)与任务调度无关。

在任务调度的过程中，内核根据任务的参数可动态修改优先级。任务调度支持时钟节拍，支持信号量、消息队列、事件控制块、事件标志组和消息邮箱任务通讯机制并支持中断嵌套。中断嵌套可达255层，中断使用当前任务的堆栈保存上下文。每个任务有自己的堆栈，堆栈的大小用户自己设定，任务堆栈由用户静态或者动态创建，任务的创建本身不进行动态内存分配。

在 μC/OS-Ⅱ中由任务调度器来完成任务调度。μC/OS-Ⅱ内核任务调度的主要工作分为两部分：一是查找进入就绪态的具有最高优先级的任务；二是实现任务间的切换。利用就绪任务表对任务就绪表中优先级最高的任务进行查找。

μC/OS-Ⅱ内核调度流程(图3.2)为：

1. 通过OSTaskCreate()函数建立任务。
2. 初始化任务控制块TCB，及其该任务的堆栈。
3. 将优先级索引表上的指针指向与其对应的任务控制块，用OSTabHighRdy变量记录当前优先级最高的就绪任务的任务控制块地址，系统判断优先级最高的就绪任务并进行调度。

查找到最高优先级的就绪任务后，μC/OS-Ⅱ调用任务切换宏OS_TASK_SW()进行任务切换。μC/OS-Ⅱ调度器在查到最高优先级就绪任务堆栈之后，只要恢复处理器的各个寄存器的值，特别是PC寄存器的值，任务就可从原来被中断的地方继续执行。

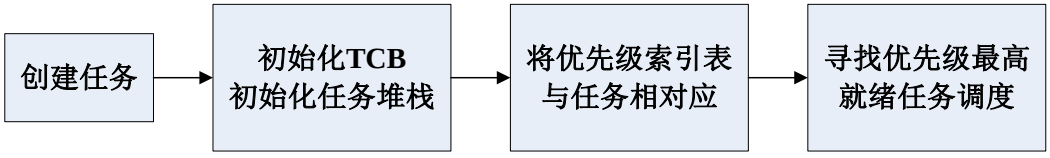

图3.2 μC/OS-Ⅱ调度流程

任务切换的相关函数与CPU体系相关，由汇编完成。相关函数如下：

1. OSStartHighRdy()：执行优先级最高的任务
2. OSCtxSw()：完成任务的上下文切换
3. OSIntCtxSw()：中断后的任务上下文切换
4. OSTickISR()：启动中断服务程序

3.3 micro-K 操作系统

3.3.1 micro-K 的体系架构

本文的重点是设计一种面向 C8051F120 的微型嵌入式操作系统内核 micro-K，因此在设计内核时，只需具备内核最基本的几项功能，力求内核体积的最小化，确保 micro-K 内核能正常运行于 Flash 存储器为 128KB 的 C8051F 单片机中。

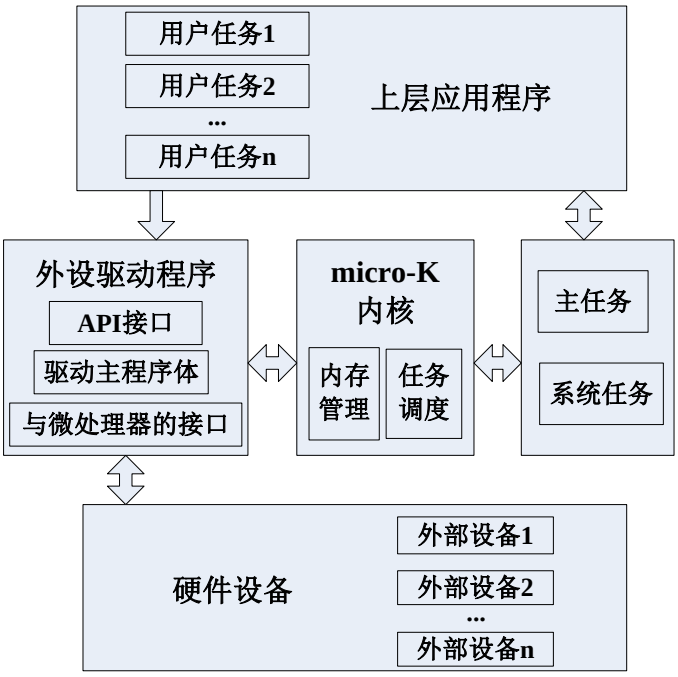

图 3.3 总体框架图

micro-K 实时多任务操作系统的总体框架图如图 3.3 所示。根据任务需要，可将 micro-K 操作系统划分成如下几个功能模块：

1. micro-K内核

micro-K的主要任务就是完成多任务之间的调度和同步，为用户提供标准的API接口函数。

2. 驱动程序模块

驱动程序部分是连接上层API函数和底层硬件的纽带。驱动程序模块将API函数和底层硬件分离开来。任何底层硬件的改变、删除或者添加，只需要对提供给操作系统的硬件驱动程序随之进行相关的操作即可，不会影响到API函数和用户应用软件。

3. 用户应用程序模块

用户根据实际的需求创建用户应用程序，其任务建立在系统的主任务基础之上。用户应用程序通过调用micro-K提供的接口函数对系统进行操作，实现

系统功能。

在应用过程中，若要运行某一外设，则内核只需调用相应的外设驱动程序，而不必对内核及运行在其中的软件进行修改，可以加快系统开发，同时也使系统维护起来更加容易。

3.3.2 micro-K的多任务管理

1. 任务

任务在操作系统和并发程序设计中非常重要。在操作系统中，任务(线程)是一个简单的程序，此程序可认为自己完全占有 CPU 的控制权。设计实时应用程序时，首要考虑的就是如何把问题分解为多个任务，并建立起任务之间的关系[23]。在实时应用中，操作系统的多任务化使得开发人员可以将很复杂的应用程序层次化，同时使程序更容易设计与维护。

在micro-K中，实时应用程序设计的关键就是如何把实际问题分割成多个子任务，为每个子任务赋予一定的优先级，分配一套独立的CPU寄存器和栈空间(图3.4)。

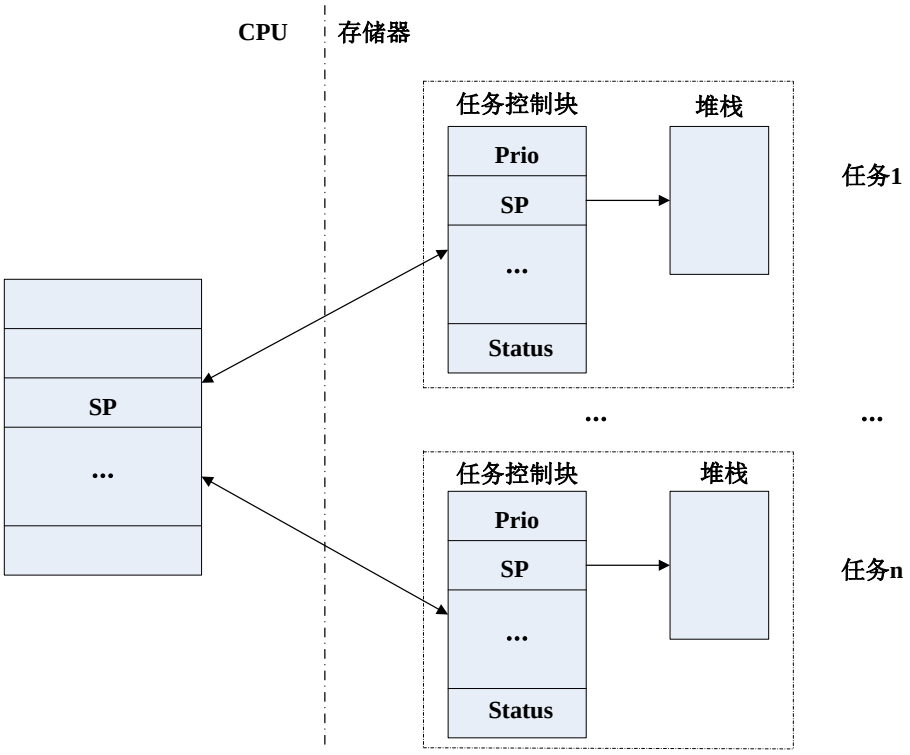

图3.4 micro-K的多任务堆栈

2. micro-K内核中的任务状态

micro-K中的任务主要有睡眠态(DORMANT)、就绪态(READY)、运行态(RUNNING)、挂起态(PENDING)和中断服务状态(ISR RUNNING)五种状态，

利用变换条件，任务可以在任意两种状态间相互转换。micro-K下的任务自建立到删除就一直处于无限的循环之中，且在这一过程中，任务的状态必定是这五种状态之一。micro-K中任务的状态转换图见图3.5。

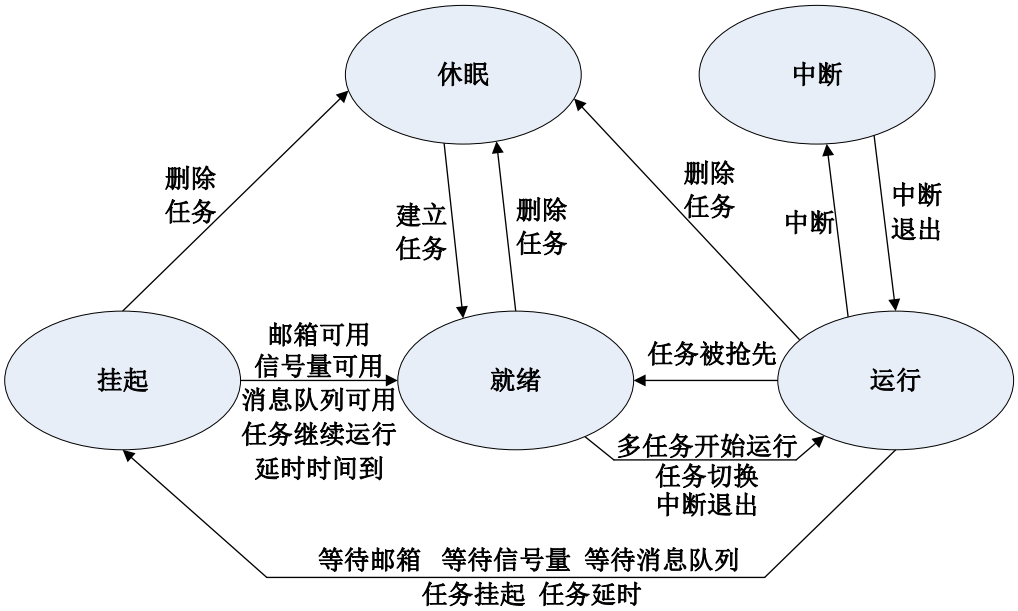

图3.5 任务的状态变化

3. 任务优先级

micro-K 是一个基于优先级的抢占式调度内核，最多可以有 128 个优先级，分别对应优先级 0-127，优先级越高，优先级数值越低，0 代表最高的优先级。在 micro-K 中，任务的优先级编号就是任务编号 ID。优先级可以在任务初始化设定，也可以在任务运行过程中通过提高任务的特征参数由系统动态分配。一些内核功能函数也可调用优先级号，如改变优先级函数，任务删除函数等等。

4. 任务控制块及任务控制块链表

任务控制块是任务的核心数据结构，用来描述任务的属性，结构简单，可裁剪。内核根据优先级序号进行任务的添加，查找，删除等功能。

任务控制块为静态数组，它在任务建立时被初始化，并被赋予初值。当任务被撤销时，内核将任务的状态全部保存在任务控制块中，当任务再次处于运行态时，任务控制块使任务从原中断处继续执行。任务控制块是任务与操作系统之间以及任务与任务之间的纽带[24]。所有的任务控制块都放在任务控制块列表数组OSTCBTbl[]中。

任务控制块中的相关变量如下：

- . *mKTCBStkPtr：指向当前任务栈顶的指针，是任务控制块中唯一需要用汇编语言来处置的变量。. mKTCBStkPtr被放在第一个字段，是任务运行的关键。
- . *mKTCBExtPtr：在用户定义的任务扩展模块中使用。用户可以扩展

任务控制块而不必修改μC/OS-Ⅱ的源代码。

- .*mKTCBStkBtm：指向任务栈底的指针。变量mKTCBStkBtm在函数mKTaskStkChk()中使用，在运行中检验栈空间的使用情况。
- .mKTCBStkSize：任务堆栈的长度。
- .mKTCBId：用于存储任务的识别码。
- .mKTCBEventPtr：指向事件控制块的指针。
- .mKTCBMsg：指向传给任务的消息的指针。
- .mKTCBStat：任务的状态字。当.mKTCBStat为0，任务进入就绪态。也可以为.mKTCBStat赋予其它的值。
- .mKTCBPrio：任务优先级。.mKTCBPrio值越小，任务的优先级越高。
- .mKTCBDelReq：用于表示该任务是否需要删除。
- .D：任务的时限距离(相对截止期)。采用HP优先级策略分配动态优先级时使用该变量。
- .V：任务的价值，即任务的重要程度。
- .R：任务的运行时间，即任务在不被中断的情况下，从开始到运行完毕所需的处理器时间。
- .ExPrio：优先级交换前的任务优先级。若优先级未发生过交换，则将其置为127。
- .mKTCBX，.mKTCBY，.mKTCBBitX和.mKTCBBitY：用于加速任务进入就绪态。
- .mKTCBNext和.mKTCBPrev：用于任务控制块的双重链接。双重连接的链表使得任一成员都能被快速插入或删除。
- .mKTCBDly：用于任务延时若干时钟节拍，或者将任务挂起一段时间以等待某事件的发生(这种等待是有超时限制的)。

micro-K利用空进程块链表free_mK_TCBs和进程块链表mKTCBTbl两条链表来管理任务控制块。当micro-K初始化时，所有的任务控制块被链接起来，形成空进程块链表，如图3.6所示。

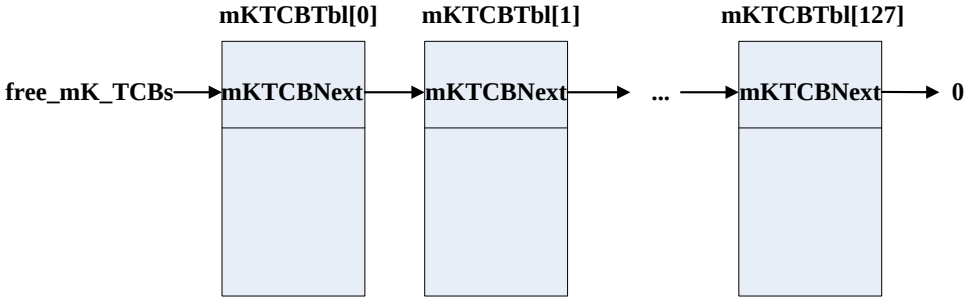

图3.6 任务控制块链表

任务控制块链表是系统在调用函数mKTaskCreate()或mKTaskCreateExt()

创建任务时建立的。任务建立后便获取空任务控制块指针mKTCBFreeList所指向的任务控制块，而mKTCBFreeList的指针则指向链表中下一个空的任务控制块。从空任务控制块链表中摘取一个空的任务控制块，对这个控制块的任务属性进行赋值，这样就形成了任务控制块链表。mKTCBTbl[]数组，用来存放以任务的优先级别为顺序的各个任务控制块的指针。这种方法避免了系统在访问任务时遍历整个任务控制块。变量mKTCBCur存放当前正在运行的任务的任务控制块指针。

5. 建立和删除任务

应用程序通过调用mKTaskCreate()或mKTaskCreateExt()函数创建任务。建立一个任务就是要建立相应的任务控制块，通过任务控制块为任务建立代码和堆栈，并指定任务的优先级。若在micro-K内核中采用HP优先级分配算法，则由任务的特征参数动态提供任务优先级。建立任务时，首先要判断任务优先级是否合法且未用，随后初始化任务的堆栈和任务控制块。初始化工作完成后，将任务计数器mKTCounter加1，当判断micro-K内核处于运行态时就进行任务调度。

应用程序调用mKTaskDel()函数删除任务。删除任务就是把该任务的任务控制块从任务控制块链表中清除，放回到空任务控制块链表，然后将任务置于休眠状态，系统不再管理任务。

6. 挂起和恢复任务

应用程序调用mKTaskSuspend()函数来完成任务的挂起，停止运行任务。如果调用mKTaskSuspend()函数的任务要挂起自身，则删除任务的就绪标志并做挂起记录，触发任务调度，使运行其他的就绪任务。若调用mKTaskSuspend()函数的是其他任务，则只需删除任务的就绪标志并做挂起记录。

mKTaskAwake()函数负责指定任务的解挂。当函数在判断该任务是已存在的挂起任务且不在等待其他资源后，将任务的状态从挂起态变为就绪态，进行任务调度。

3.4 本章小结

本章研究、分析了实时抢占式操作系统μC/OS-Ⅱ的内核结构，重点介绍了micro-K内核的体系架构和主要功能。在后续的章节里将具体说明内核中重要功能的设计与实现并着重介绍调度算法、优先级分配策略以及设备驱动管理的设计与实现。

第四章 micro-K 任务调度算法的设计与实现

嵌入式操作系统的核心在其调度算法。μC/OS-Ⅱ使用的调度算法是优先级位图调度算法[25]，这种算法简洁明了，效率很高，且其全部的函数调用与服务时间都是可知的。但是 micro-K 操作系统要面对多个任务且对时限要求较高，因此需要更高效的优先级调度算法。micro-K 内核采用改进的优先级位图调度算法对任务进行管理，并融合动态调度策略提高系统的实时性能和资源利用率。本章主要是设计并实现 micro-K 的任务调度算法—深度优先级调度算法。

4.1 优先级位图调度算法

在操作系统中，任务队列通过任务控制块实现对系统中所有任务的管理，比较简单的管理方式就是把任务组织为就绪队列和等待队列。在基于优先级的调度处理中，要获得系统处于就绪态中的最高优先级任务，通常采用下面的处理方式[26]：

1. 任务就绪时，把就绪任务的TCB放在就绪队列的末尾。调度程序需要从头到尾遍历就绪队列，才能获得优先级最高的任务。
2. 按照优先级从高到低的顺序排列就绪队列。当有新的就绪任务到达时，将其插到就绪队列中合适的位置，保证就绪对列中的优先级仍按照从高到低排列。

这两种方法所花费的处理时间都与任务数量有紧密的关系，具有不确定性。为了提高内核的确定性，μC/OS-Ⅱ采用优先级位图调度算法。

4.1.1 任务就绪表 OSRdyTbl[]

优先级位图调度算法共涉及到三个表两个变量，分别是就绪表(OSRdyTbl)，映射表(OSMapTbl)，反映射表(OSUnMapTbl)，变量OSRdyGrp以及相关的任务优先级prio。其中映射表和反映射表是两个常数表，用于查表算法。优先级位图算法是以空间来换取时间，这两个常数表使就绪算法在任务查找和状态修改时能够在常数时间内完成。

1. OSRdyGrp 与 OSRdyTbl[]之间的关系

任务就绪表是一个位矩阵。μC/OS-Ⅱ中的每个就绪任务都放在就绪表中。就绪表有两个变量：OSRdyGrp 和 OSRdyTbl[]，通过定义这两个变量就可标识

所有的就绪任务。OSRdyGrp 和 OSRdyTbl[]用 8 位分别表示任务优先级所在的行和列的位置。

内核任务就绪表 OSRdyTbl 是一个 8bit 的字符型数组，每一位对应 64 个优先级中的 8 个优先级，当处于某一优先级的任务已经就绪时，就将该位置为“1”。字符型全局变量 OSRdyGrp 共有 8 位，每一位表示其中一组，每组又包含 8 个优先级。当 OSRdyTbl[]中的任何一位为 1 时，OSRdyGrp 中的对应位也应置 1，表示 8 组任务对应的优先级组中至少有 1 个优先级分配给进入就绪态的任务。为了确定下一个需运行的就绪态任务，μC/OS-Ⅱ总是把优先级别最低的任务在 OSRdyTbl[]中相应字节的相应位置“1”。检索时，内核从优先级最高的 OSRdyTbl[0]开始扫描整个任务就绪表，再经过计算，就可找到具有最高优先级的就绪任务[27]。

2. 任务优先级与 OSRdyGrp 和 OSRdyTbl 的关系

任务优先级 prio 与 6 个二进制位相对应，其中高 3 位为优先级在 OSRdyGrp 的二进制位位置，与 OSRdyTbl 的数组元素下标相对应；低 3 位表示对应 OSRdyTbl 的数组元素的值。任务优先级与 OSRdyGrp 和 OSRdyTbl 的关系见图 4.1。

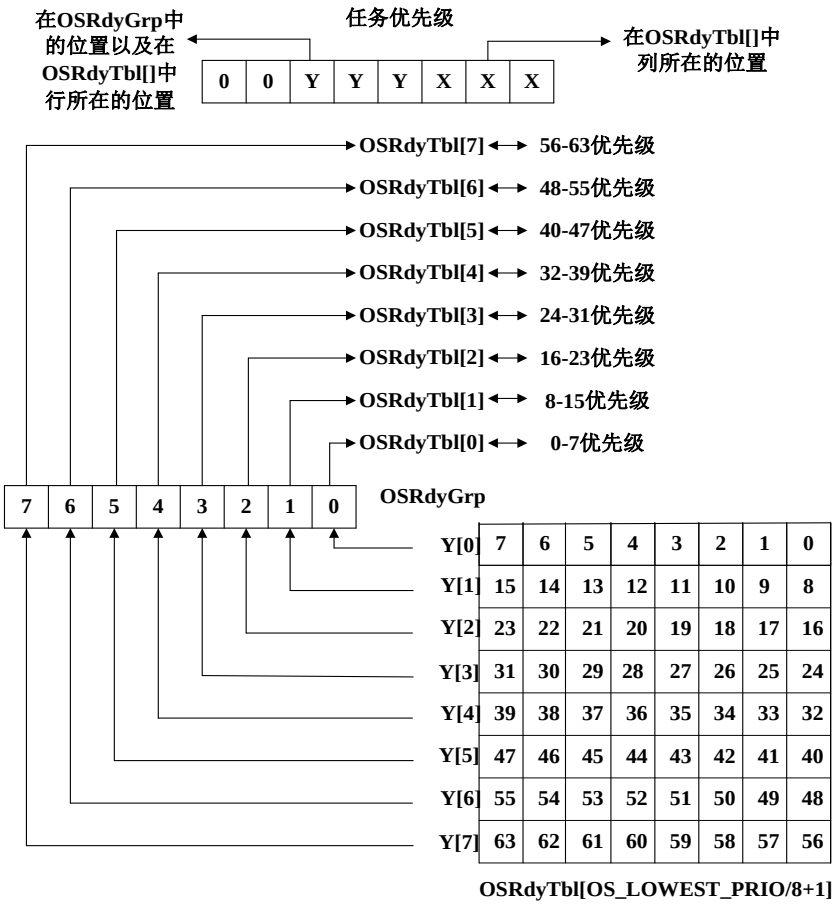

图 4.1 任务优先级与 **OSRdyGrp** 和 **OSRdyTbl** 的关系

4.1.2 任务优先级到 OSRdyGrp 的优先级映射表 OSMapTbl[8]

表 4.1 优先级映射表

下标	位掩码
0	**00000001**
1	**00000010**
2	**00000100**
3	**00001000**
4	**00010000**
5	**00100000**
6	**01000000**
7	**10000000**

OSMapTbl[]是 ROM 中的屏蔽字，用于限制 OSRdyTbl[]数组的元素下标(0-7)。OSMapTbl[]数组元素的下标对应于任务优先级的高三位。OSMapTbl 的数组元素对应的二进制值中，位为“1”的位表示 OSRdyGrp 的对应位也为“1”。优先级映射表的内容如表 4.1 所列。

4.1.3 优先级判定表 OSUnMapTbl[]

在操作系统中，顺次检索是寻找优先级最高就绪任务的一种最简单的方法。然而当系统中的任务逐渐越多时，优先级最高的就绪任务就越来越靠后，查找的速度也越来越慢。μC/OS-Ⅱ是通过查找 OSUnMapTbl[]表避免以上情况的，这种方式不仅可以提高查找速度，而且能够保证优先级不同的任务的查找时间为常数。优先级判定表 OSUnMapTbl[]为 16*16 维的数组，这是因为 OSRdyGrp 用 8 位存储，因此可能存在 2^8=256 种的优先级组的组合。而且 OSRdyTbl[i]也为 8 位，同样存在 2^8=256 种的不同的优先级的组合[28]。

OSUnMapTbl[]中每个数组元素的值是固定的，当使用该表格时，以 OSRdyGrp 和OSRdyTbl数组元素的值为索引，数组的下标转换为相应的八位二进制，其中低四位对应数组的列，高四位对应数组的行，由此查表就可获取优先级最高的就绪任务所在的位置序号(0-7)。如OSRdyGrp的值为00010010(0x12)，以0x12为下标，对应OSUnMapTbl的数组元素的值为1，则表示OSRdyGrp对应二进制表示中1出现的最低二进制位的序号为1。检索时，内

核从优先级最高的OSRdyTbl[0]开始扫描整个任务就绪表，再经过计算，就可找到具有最高优先级的就绪任务。OSUnMapTbl[]优先级判定表的内容如图4.2所示。

```
OSUnMapTbl[] = {
    0，0，1，0，2，0，1，0，3，0，1，0，2，0，1，0
    4，0，1，0，2，0，1，0，3，0，1，0，2，0，1，0
    5，0，1，0，2，0，1，0，3，0，1，0，2，0，1，0
    4，0，1，0，2，0，1，0，3，0，1，0，2，0，1，0
    6，0，1，0，2，0，1，0，3，0，1，0，2，0，1，0
    4，0，1，0，2，0，1，0，3，0，1，0，2，0，1，0
    5，0，1，0，2，0，1，0，3，0，1，0，2，0，1，0
    4，0，1，0，2，0，1，0，3，0，1，0，2，0，1，0
    7，0，1，0，2，0，1，0，3，0，1，0，2，0，1，0
    4，0，1，0，2，0，1，0，3，0，1，0，2，0，1，0
    5，0，1，0，2，0，1，0，3，0，1，0，2，0，1，0
    4，0，1，0，2，0，1，0，3，0，1，0，2，0，1，0
    6，0，1，0，2，0，1，0，3，0，1，0，2，0，1，0
    4，0，1，0，2，0，1，0，3，0，1，0，2，0，1，0
    5，0，1，0，2，0，1，0，3，0，1，0，2，0，1，0
    4，0，1，0，2，0，1，0，3，0，1，0，2，0，1，0
};
```

图 4.2 优先级判定表 OSUnMapTbl[]

OSUnMapTbl[256]这个预先计算好的矩阵可以简化为横、纵两个数组，直接用矩阵数组就可查出对应的坐标值。这个矩阵的作用就是用空间换时间效率，虽然运算步骤略微多几步，但是避免了计算，空间占用也可以减少了很多。

4.1.4 优先级判定表具体的算法分析

算法真值表如表 4.2 所示，令 OSRdyGrp 和 OSRdyTbl[i]的各位从右到左依次为 bit0，bit1，…，bit7，算法规定：

bit0=bit1=…=bit7= =0，即 00000000，偏移量为 0；

当 bit0=1 时，即*******1，偏移量为 0；

当 bit0=0、bit1=1 时，即******10，偏移量为 1；

……

当 bit0=bit1=…=bit6=0、bit7=1，即 1000000，偏移量为 7。

根据以上算法，可得到 OSUnMapTbl[]优先级判定表，要快速进入当前就绪态中优先级最高的任务只需查找 OSUnMapTbl[]即可。

表 4.2 算法的真值表

OSRdyGrp/RdyTbl[i]								OSUnMapTbl[]
bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	
0	0	0	0	0	0	0	0	0
*	*	*	*	*	*	*	1	0
*	*	*	*	*	*	1	0	1
*	*	*	*	*	1	0	0	2
*	*	*	*	1	0	0	0	3
*	*	*	1	0	0	0	0	4
*	*	1	0	0	0	0	0	5
*	1	0	0	0	0	0	0	6
1	0	0	0	0	0	0	0	7

4.1.5 任务进入就绪态

当某一优先级为 Prio 的任务准备就绪时，μC/OS-Ⅱ将位图中的相应位置位，任务进入就绪态。将任务放入就绪表的代码如下：

```
OSRdyGrp |= OSMapTbl[prio>>3];
OSRdyTbl[prio>>3] |= OSMapTbl[prio&0x07];
```

4.1.6 任务退出就绪态

当该优先级任务由于等待信号量、消息队列、邮箱或延时而进入等待状态时，则将相应位复位，任务退出就绪态，使任务脱离就绪态的代码如下：

```
OSRdyGrp &= ~OSMapTbl[prio>>3];
OSRdyTbl[prio>>3] &= ~OSMapTbl[prio&0x07];
```

4.1.7 获取进入就绪态的最高优先级

找出进入就绪态的优先级最高的任务程序代码如下：

```
y = OSUnMapTbl[OSRdyGrp];
x = OSUnMapTbl[OSRdyTbl[y]];
prio = (y<<3)+x;
```

确定优先级最高的任务以后，用 μC/OS-Ⅱ中的调度器来调度运行任务。μC/OS-Ⅱ的任务调度与应用程序建立了多少任务没有关系。随后只需要先将被

挂起任务的处理器寄存器推入堆栈，然后将较高优先级任务的寄存器值从堆栈恢复到寄存器中就可完成任务的切换。

4.1.8 根据优先级获取相应的任务

μC/OS-Ⅱ中任务按优先级进行组织，以优先级为下标，即可得到相应任务的任务控制块。

μC/OS-Ⅱ采用固定优先级调度的最大问题是对时限的要求具有易忘性，也就是说这种优先级调度对硬实时不敏感。μC/OS-Ⅱ中可供用户使用的任务只有56个，且不能同时存在多个优先级相同的任务，这往往满足不了应用系统的需求。另外由于内核采用信号量机制实现任务同步并防止多个任务同时访问共享资源，也会出现优先级反转[29,30]的情况。

4.2 深度优先级调度算法

μC/OS-Ⅱ是专门为实时系统所设计的多任务操作系统，尽管能管理64个任务，但若想让多任务在C8051F120微处理器上并发运行，同时又能实时响应系统需求，在运行过程中也会有出现一些问题。

micro-K内核最基本的调度算法为优先级位图算法的改进算法—深度优先级调度算法。这种算法可以管理128个任务，这对于基于C8051F120的嵌入式系统来说任务数是足够的，即便应用于当前一般的系统，也不会成为开发过程中的瓶颈资源。嵌入式操作系统内核micro-K在保证μC/OS-Ⅱ内核已有特性和优点不受影响的前提下，对其核心算法进行改进，使其可用的任务数增加一倍，达到自身所需的128个。

4.2.1 设计方案比较

μC/OS-Ⅱ的任务调度是通过任务就绪表 OSRdyTbl[]来完成的。调度函数采用查找优先级判定表 OSUnMapTbl[]的方式来确定最高优先级的就绪任务。这种方法的本质是用任务优先级的低 3 位和次低 3 位来确定其在表 OSUnMapTbl 中的 X 坐标和 Y 坐标的位置。

设计方案至少必须考虑到下面几点：

1. 尽可能保持μC/OS-Ⅱ原有的系统结构，对内核代码的修改应该越少越好，以便于内核移植和软件复用。
2. 由于嵌入式系统中任务调度比较频繁，且占用的多是系统的绝对优势

资源，所以对应的代码不能占用大量的存储空间，涉及的修改变量也要尽量少。

3. 在实际应用中，要求嵌入式系统具有良好的实时性，那么系统的相应时间就变得很重要，所以应考虑如何将任务的响应时间最优化。

方案一：

在保持 μC/OS-Ⅱ原有结构的前提下，当要增加可用的任务数时，任务就绪表 OSRdyTbl[]的位数或下标必须增加，OSMapTbl[]的下标也应随之增长，这种增长关系是线性的。对 OSRdyTbl[]的扩展应该首先利用位数。这里选择将位数扩到 16 位，下标保持不变。将 OSMapTbl[]改为 mKMapTbl[]，mKMapTbl[]的下标和位数都是 16，其他变量基本不需变动，文件 μCOS_Ⅱ.H 中的改动代码如下所示。

```
#define mK_EVENT_TBL_SIZE   ((mK_LOWEST_PRIO)/16+1)
#define mK_RDY_TBL_SIZE   ((mK_LOWEST_PRIO)/16+1)
INT16U mKTCBBitX;
INT16U mKTCBBitY;
mK_EXT INT16U mKRdyTbl[mK_RDY_TBL_SIZE];
extern INT16U const mKMapTbl[];
#if mK_MAX_TASK>127
#error “mK_CFG. H，mK_MAX_TASKS must be<=127”
#endif
```

原先的变量 OSRdyTbl[]为 8 位，故原先的优先级判定表 OSUnMapTbl[]是个有 2^8=256 个元素的数组。但是当变量 mKRdyTbl[]为 16 位时，不能够直接从 OSUnMapTbl[]中查出当前优先级最高的任务，因为它本来是为 8 位数据设计的。如果将 OSUnMapTbl[]改为 16 位数据的查找表，则要求 OSUnMapTbl[]变成 2^{16}=65536 维的大数组。很明显，这样的方案即增加了代码的空间，又不符合嵌入式系统代码精简的要求，因此这种指数级的空间增长在嵌入式系统中是不可行的。

对文件 mK_CORE. C 中的优先级位图调度算法进行下面的调整后，系统便可运行 128 个任务。

```
INT16U const mKMapTbl[]={0x0001,0x0002,0x0004,0x0008,0x0010,0x0020,
0x0040,0x0080,0x0100,0x0200,0x0400,0x0800,0x1000,0x2000,0x4000,
0x8000};
mKIntExit()
{   …
  if((mKRdyTbl[mKIntExitY]&0xFF) = = 0)
```

```
            x=mKUnMapTbl[mKRdyTbl[mKIntExitY]>>8]+8;
    else
            x=mKUnMapTbl[mKRdyTbl[mKIntExitY]&0xFF];
    mKPrioHighRdy=(INT8U)((mKIntExitY<<4)+X));
     …
}
mKStart()
{   …
    if((mKRdyTbl[y]&0xFF) = = 0)
     mKPrioHighRdy=(INT8U)((y<<4)+(mKUnMapTbl[mKRdyTbl[y]>>8]+8));
    else
     mKPrioHighRdy=(INT8U)((y<<4)+mKUnMapTbl[mKRdyTbl[y] &0xFF]);
     …
}
mK_Sched()
{   …
    if((mKRdyTbl[y]&0xFF) = = 0)
     mKPrioHighRdy=(INT8U)((y<<4)+(mKUnMapTbl[mKRdyTbl[y]>>8]+8));
    else
     mKPrioHighRdy=(INT8U)((y<<4)+mKUnMapTbl[mKRdyTbl[y] &0xFF]);
     …
}
mK_TCBInit()
{   …
    ptcb->mKTCBY=prio>>4;
    ptcb->mKTCBX=prio>>0x0F;
     …
}
```

但这样做也增加了查找时间，原来的查找操作为：两次查表、一次移位和一次加法。改进后的查找操作为：两次位与，两次比较，两次查表，两次移位，两次加法。改进后任务管理方法使系统的响应时间一定程度上有所延迟，因此这种方案不可取。

既要扩充任务数，又要不增加代码的空间复杂度，在对任务管理算法进行优化时，就只能以运算能力来换取对相关资源的需求，从而解决实际中的问题。为了增加 μC/OS-Ⅱ内核可以管理的任务数目，需要对 μC/OS-Ⅱ中任务的相关

结构和算法进行改进。选取的具体任务管理方案如下。

方案二：

任务数的扩充可利用 μC/OS-Ⅱ有的优先级判定表，重新定义存放任务优先级变量的字节，并建立深度任务就绪表，把 64 个任务扩充到 128 个任务，μC/OS-Ⅱ原有的调度算法使用的查表法使得查找优先级的时间是常数。在深度优先级调度算法中，这个查找过程所用的时间仍然是一个常量。这样就可以保证系统的实时性。

4.2.2 深度优先级调度算法

1. 设计思路

最行而有效的做法是在 μC/OS-Ⅱ原有的结构上进行扩展，即扩展原来 6 位的优先级定义字节。分析 μC/OS-Ⅱ内核发现，原有的基于 64 个任务调度的优先级调度算法中任务优先级变量 prio 只利用了其中的 6 位，分别用 3 个比特位来定位任务优先级在任务就绪表中的行和列，即 0-2 位标识该任务在任务就绪表中的横向信息，3-5 位标识该任务在就绪表中的纵向信息。因此，存放任务优先级字节中的 8 个比特位只用到了 6 个，扩展任务数就可利用任务优先级变量 prio 中的一位空闲位，这样将原先的 64 个任务扩展为 64 个任务集合，每个集合包括 2 个深度任务，即在同一优先级下扩展 2 个深度任务，使任务能够管理 128 个任务。

micro-K 内核将就绪任务划分为若干个集合，把紧迫程度相近的任务放在同一集合内，用户可根据实际需要在集合内定制任务调度策略，同时保留集合间优先级差异。在设定优先级时，重要度相差很大的任务应避免放在同一就绪组中。算法中保留优先级位图算法中的优先级设置作为静态优先级。本文主要实现任务集合间的深度优先级位图调度算法和任务集合内的动态调度机制，在以后的实际应用中还可根据需要填充其他的动态调度机制。

2. 对任务控制块的设计

当用户程序调用函数 mKTaskCreate()创建一个用户任务时，mKTaskCreate()就会为待运行的任务建立任务控制块，这样便于系统对任务进行管理。micro-K 在进行任务调度时最主要的依据是任务就绪表，而任务优先级则决定了任务在就绪表中的位置。mK_TCB 内核用 mKTCBPrio 来记录任务在系统中运行时所具有的优先级，即任务执行优先级。增加 mKTCBTaskID 字段来标识各优先级任务集合的 ID 号，启用任务控制块中的 mKTCBId 字段来标识深度优先级任务的 ID 号。将变量 prio（任务的优先级别，保存在 mKTCBPrio 中）高两位空闲位中的一位标识为深度任务号。保持 mKTCBPrio 变量不变，将 mKTCBId 变

量的值设置为 0 和 1，便可以实现同一优先级集合下有 2 个深度任务。

将 mK_LOWEST_PRIO 的值设为 127，确定任务就绪表 mKRdyTbl[]数组的大小。为确定下次运行就绪任务集合中的任务，内核调度器总是将 mK_LOWEST_PRIO 在就绪表中相应字节的相应位置 1。深度优先级调度算法中任务优先级字节的定义以及 mKRdyGrp 和 mKRdyTbl[]之间的关系规定如下图 4.3 所示。

增加深度任务就绪表 mKRdyDTbl[]来标识深度任务就绪情况，这样便于在调度时快速查找深度任务优先级，实现深度任务管理与调度。这样即使任务数得到了扩展，保证了系统实时性，又实现了多任务间的并发执行。mKRdyDTbl[]的示意图如图 4.4 所示。

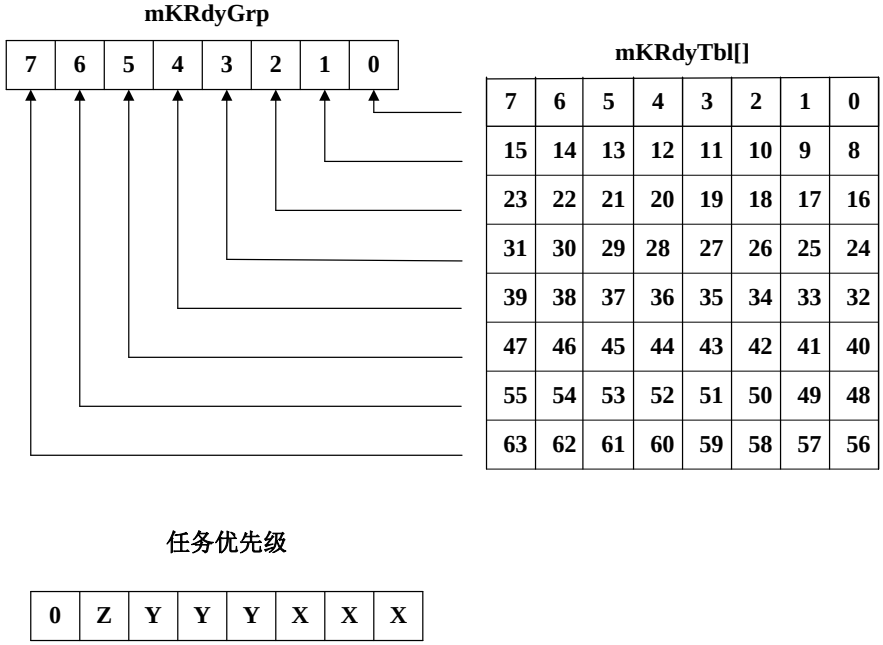

图4.3 **micro-K就绪表**

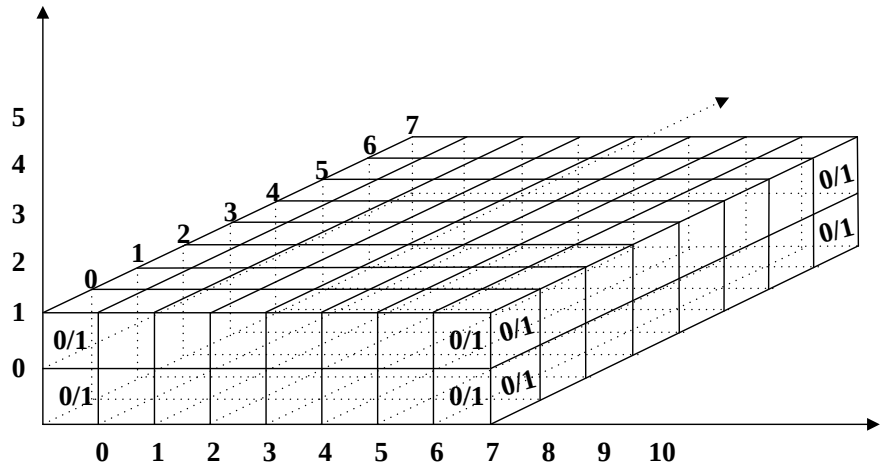

图 4.4 深度任务就绪表 **mKRdyDTbl[]**

3. 对相关函数及操作代码的设计

以下函数就为任务进入就绪状态的操作与任务脱离就绪状态的操作：

◆ 创建任务

micro-K内核通过mKTaskCreateExt()函数来建立任务。mKTaskCreateExt()函数内容包括：

首先检测分配给任务的优先级是否有效。任务优先级的取值范围必须为0到mK_LOWEST_PRIO之间。但在应用程序中不能调用mKTaskCreateExt()函数来创建优先级为mK_LOWEST_PRIO的任务。

其次需要确认指定的优先级上是否有已经存在的任务。如果不存在则开始创建任务，并为待创建的任务分配一个任务控制块mK_TCB。

接下来对任务堆栈进行初始化操作。mKTaskCreateExt()调用mK_TCBInit()，从空闲的mK_TCB缓冲池中获得并初始化一个任务控制块mK_TCB。同时对任务堆栈进行检查，如果没有错误则将任务计数器mKTCounter加1。

最后判断系统是否处在运行状态。如果mKTaskCreateExt()是在某个任务执行过程中被调用，则进行任务调用，以判断新建立的任务是否比原来任务的优先级高。如果在多任务调度之前建立新的任务，那么任务调度函数不会起作用。

其他有关任务的函数如任务启动、挂起、恢复、删除函数中代码的相关修改不多，在此不做过多讨论。

◆ 任务进入就绪状态

prio是任务的优先级，也是任务的识别号，则将任务放入就绪表，即使任务进入就绪态的方法是：

```
mKRdyGrp |= mKMapTbl[prio>>3];
mKRdyTbl[ptcb->mKTCBY] |= mKMapTbl[prio&0x0F];
mKRdyDTbl[prio] |= mKMapTbl[ptcb->mKTCBId&0x07];
```

◆ 任务脱离就绪状态

从就绪表中删除优先级为 prio 的任务也比较简单，只要清除 mKRdyGrp 和 mKRdyTbl[]中该任务的对应位即可。代码如下：

```
if((mKRdyDTbl[prio] &= ~mKMapTbl[DtaskID&0x07]) = = 0)
if((mKRdyTbl[prio>>3] &= ~mKMapTbl[prio&0x07]) = = 0)
mKRdyGrp &= ~mKMapTbl[prio>>3];
```

◆ 查找最高优先级的就绪任务

在 μC/OS-Ⅱ中，调度程序总是选择优先级最高的就绪任务运行。内核运行时将多次、频繁地进行任务调度，任务在进行调度时需要独占 CPU，不允许外部中断和任务切换。所以在 μC/OS-Ⅱ任务数扩充问题上，最大的难点就

是在任务调度机制中，如何快速有效地找出当前当前就绪的最高优先级任务，否则会影响整个系统的响应速度和处理能力。

任务级的调度是由函数mKSched()完成的。mKSched()中的核心代码是利用就绪表和深度任务就绪表计算最高优先级任务集合的ID号和最高优先级深度任务的ID号。

$$mKPrioHighRdy = (mKPrioHighRdy << 2) + id \tag{4.1}$$

由公式4.1可得：当前就绪的最高优先级任务号=就绪表中最高优先级任务集合的ID号左移2位+就绪表中最高优先级深度任务的ID号。

◆ 任务切换

任务切换由两步完成：将被挂起任务的处理器寄存器推入堆栈；然后将较高优先级任务的寄存器值从栈中恢复到寄存器中。为实现任务切换，mKTCBHighRdy必须指向优先级最高的那个任务控制块mK_TCB，还需让当前任务控制块mKTCBCur指向通过优先级最高的那个任务。将以mKPrioHighRdy为下标的mKTCBPrioTbl[]数组中的元素赋给mKTCBHighRdy来实现，并且统计任务切换次数的计数器mKCtxSwCtr加l，以跟踪任务切换次数。最后宏调用mK_TASK_SW()来完成实际上的任务切换。而这一操作是在临界区完成，它不允许任何操作将其中断。具体代码如下：

```
if(mKPrioHighRdy != mKPrioCur)
{
    mKTCBHighRdy = mKTCBPrioTbl[mKPrioHighRdy];
    mKCtxSwCtr++;
    mK_TASK_SW();
}
```

◆ 任务间的通讯与同步

在micro-K内核中，保护任务之间的共享数据和提供任务之间的通讯的方法有以下几种：

方法一：是通过mK_ENTER_CRITICAL()和mK_EXIT_CRITlCAL()保护临界段中的数据；

方法二：利用函数mKSchedLock()和mKSchekUnlock()实现数据的共享；

方法三：通过信号量、邮箱和消息队列实现数据共享和任务通讯(所有的信号都被看成是事件)。

为了实现子任务间的通讯，内核事件控制块(ECB)是核心数据，需要进行扩展。修改后事件控制块的数据结构：

```
typedef struct{
```

```
    void      *mKEventptr;                          //指向消息的指针
    INT8U     mKEventTbl[mK_EVENT_TBL_SIZE];//等待任务列表
    INT8U     mKSubTaskEventTbl[mK_EVENT_TBL_SIZE];
                                                    //对深度任务的事件等待表
    INT8U     mKEventCnt;                           //事件是信号量时的计数器
    INT8U     mKEventType;                          //事件类型
    INT8U     mKEventGrp;                           //等待任务所在的组
}mK_EVENT;
```

. mKEventGrp中的8位分别对应所有任务的8组优先级，当某组中有任务处于事件等待状态时，. mKEventGrp中的对应位就被置“1”，. mKEventTbl[]中的相应位也被置“1”，同时在深度任务的事件等待表mKSubTaskEventTbl[]中将其对应的任务也置为“1”。当一个事件发生后，调用mKEventTaskRdy()函数实现功能。mKEventTaskRdy()修改后的主要代码如下：

```
pevent->mKEventTbl[prio>>3] |= mkMapTbl[prio&0x0F];
pevent->mKEventGrp |= mkMapTbl[prio>>3];
pevent->mKSubTaskEventTbl[mKTCBCur->mKTCBMasterID] |=
mkMapTbl[[mKTCBCur->mKTCBId&0x07];
if((mKRdySubTask[mKTCBCur->mKTCBMasterID] &=
~ mKMapTbl[mKTCBCur->mKTCBId&0x07]) = = 0)
{
 if((mKRdyTbl[prio>>3] &= ~mKMapTbl[prio&0x0F]) = = 0x00)
  {
   mKRdyGrp &= ~mKMapTbl[prio>>3];
  }
}
```

上述代码除了增加对深度任务事件操作外，其他工作原理与原系统并无差异。其他相关函数主要是针对事件控制块的操作进行修改，其原理与mKEventTaskRdy()是相似的。邮箱和消息队列都是通过缓冲区来装载任务之间的消息，故不需进行修改。

4.3 深度优先级调度算法的性能分析

相比原来的调度程序，深度优先级调度算法对数据结构的改动较小，代码的增加量也不多。算法的耗用时间由集合内外调度两部分时间组成，其时间复杂度为 O(n)，n 为同集合任务数。深度优先级调度算法为线性调度，没有增加

系统消耗。

优先级位图算法使得每一个优先级只能对应一个任务，不能同时存在多个相同优先级的任务[31]。深度优先级调度将相同优先级的任务放在一个任务集合中，通过优化的调度策略便可支持多个相同优先级任务同时存在。

4.4 本章小结

本章介绍了 μC/OS-Ⅱ的优先级位图调度算法，通过对 μC/OS-Ⅱ操作系统的研究与分析，设计实现了 micro-K 操作系统的任务调度算法。深度优先级调度算法在保留原有系统完整和统一的基础上，以最小的改动，将 μC/OS-Ⅱ的最大任务数目由 64 个扩展到了 micro-K 所需的 128 个，满足了嵌入式实时系统的代码精简、执行时间可预测的要求。

第五章 micro-K 任务分配策略的设计与实现

设计多任务操作系统的调度算法应根据系统的具体需求来确定调度策略。μC/OS-Ⅱ采用的是静态优先级分配策略—RMS 算法，用户程序为每个任务指定优先级。虽然 μC/OS-Ⅱ可用 OSTaskChangePrio()函数改变任务的优先级，但该函数功能简单，这在执行实时任务时有一定局限性。当 micro-K 内核采用静态优先级调度算法时会使系统中的任务优先级在改变时显得无能为力，系统的实时性随之减弱，若能在内核上融合动态调度策略，在任务调度中将会带来更好的效果。

本章在 micro-K 内核深度优先级调度算法的基础上融合了动态调度算法加以优化，提出了一种新的任务分配算法—HP(hybrid priority 混合优先级)算法，既使系统满足了多任务时限的能力，又解决了优先级反转的问题，极大地提高了 micro-K 的实时性。

5.1 动态任务调度算法

在实时操作系统中，实时调度算法为实时任务合理地调配运行资源，以确保能使任务在其时限到来前获得运行结果。实时调度算法的选择与设计是跟应用系统的结构和性能密切相关的。实时任务调度算法的分类如图 5.l 所示。

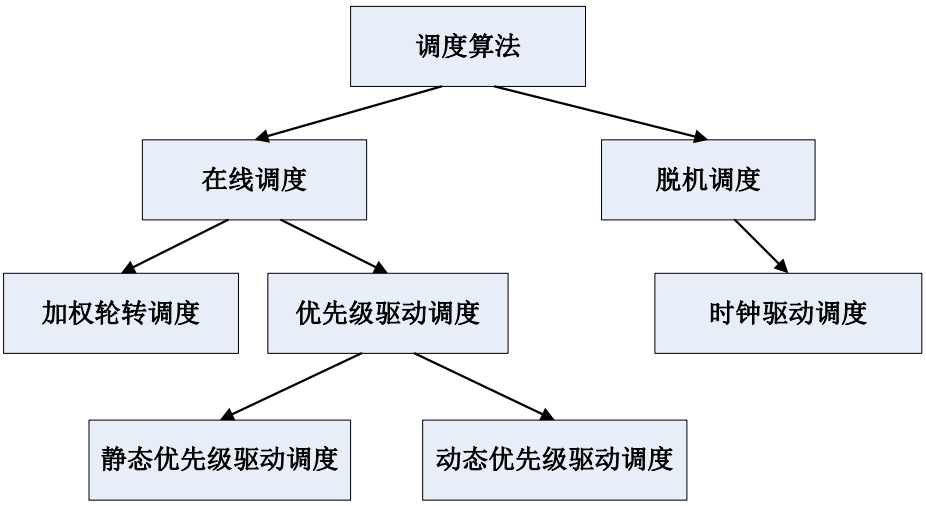

图5.1 调度算法分类

多数实时内核是基于优先级调度算法的。在实时嵌入式系统中，任务调度分为静态调度和动态调度。动态调度在于任务优先级依据不同的特征参数(如到达时间、截止时间、空闲时间、任务的重要程度或价值等)进行动态分配。

动态调度可以是抢占式的或非抢占式的，有很宽的适用范围。如果任务组中任务较多且 CPU 利用率大于 0.69 而小于 1 时，就必须使用动态调度。

任务调度算法的好坏以及执行效率直接关系到嵌入式内核的应用范围及实时性程度。动态任务调度算法的调度可行性判断是在任务执行过程中完成的，需要较少的任务信息，且在调度的过程中，任务的数量可变化，它的适应性更强。动态调度算法的调度开销较大，但它运用灵活，CPU 的利用率较高。

动态调度算法主要有最早时限优先法(Earliest Deadline First，EDF)[32,33]、最短空闲时间优先(Least Slack Time First，LSF)[34]、价值最高优先 HVF(Highest Value First)[35]、价值密度最大优先 HVDF(Highest Value Density First)[36]等算法。下面简单介绍一下这些经典的动态调度算法。

5.1.1 EDF(最早时限优先)调度算法

EDF 算法由 Liu 和 Layland 提出,是一种动态可抢占优先级实时调度算法，实时任务集的模型的表述如下：存在任务集 S(S1,S2,…,Sn)，任务 Si 可以表述为 Si={Ti,Ri,Di}；其中 Ti 为任务 Si 的周期；Ri 为任务 Si 的执行时间；Di 任务 Si 的截止时间或时限距离。任务在周期的起点释放，高优先级任务可以抢占低优先级任务的执行。所有实时任务必须满足以下限制条件：

- ◆ 所有实时任务均为周期任务，且周期大于或等于时限；
- ◆ 所有实时任务必须在其时限到来前结束；
- ◆ 所有实时任务相互独立；
- ◆ 所有实时任务均具有恒定的运行时间；

任务按周期由小到大排列为 S1≤S2≤...≤Sn。任务的时限距离 D 定义为：

$$D = d - t \tag{5.1}$$

d 表示时限，t 代表当前执行时刻。在调度时刻，任务根据任务的时限距离动态分配优先级，时间期限最短，任务优先级越高。因此，EDF 总是选择当前最迫切需要完成的任务获得处理器[37]。当有一个新的任务处于就绪状态时，各个任务的优先级就有可能要进行调整，选择截止时间最近的任务去运行。

EDF 的充分可调度条件如下：

定理 1　对于由 n 个周期任务组成的实时任务集，当且仅当满足公式 5.2 时，该任务集能够由 EDF 调度。

$$U = \sum_{i=1}^{n} \frac{Ri}{Ti} = \frac{R1}{T1} + \frac{R2}{T2} + ... + \frac{Rn}{Tn} \le 1 \tag{5.2}$$

定理 1 的意义是，对于任何给定的任务集，只要处理器的利用率不超过

100%，就必能由 EDF 调度。EDF 是单处理器实时系统的最优动态可抢占优先级调度，具有较小的调度开销，对于任何实时任务集，只要存在可行的动态调度算法，则必可由 EDF 调度。

5.1.2 LSF（最短松弛时间优先）调度算法

在动态可抢占实时调度中还可以任务的松弛时间作为判断任务优先级的标准。LSF 调度算法优先级的分配基于每个任务的有效松弛时间，一个任务的松弛时间被定义为在其错过截止期之前能够承受的最大时间延迟。LSF 算法就是根据任务的空闲时间动态分配任务的优先级，空闲时间越短，优先级越高。

任务的松弛时间 L[38]定义为：

$$L = D - b \tag{5.3}$$

D 表示时限距离（相对截止期），b 代表剩余执行时间。该算法在任意时刻把最高优先级分配给具有最小松弛时间的任务，以此来保证紧急任务的优先执行。LSF 算法的可调度条件与 EDF 算法相同。

5.1.3 HVF（价值最高优先）调度算法

对于一个任务集合 S，任务 Si 的优先级与任务的实现价值相关，实现价值[39]越高的任务其优先级也就越高，任务的重要程度由价值来表示。

5.1.4 HVDF（价值密度最大最优先）调度算法

这种算法使用价值密度作为参数去衡量任务的重要程度，价值密度等于任务的实现价值/任务的最坏运行时间，价值密度越高任务的优先级也就越高。事实上这是从某种程度上将任务的价值与任务的最坏完成时间结合的一种调度算法，等同于 HVF 的改进。GButtazzo 等人的实验结果表明当系统过载时 HVDF 算法性能更佳[40]。

5.2 调度策略的改进

HP 调度算法是一种动静结合的优先级调度机制，任务优先级由静态优先级和动态优先级两部分组成，将深度优先级调度算法中的优先级设为静态优先级（包括就绪组和组内优先级两部分），程序员将优先级重要度相差不大的任务分配到同一就绪组中；动态优先级则是处理器根据相关规则在运行期间计算分

配。这样改进后的任务调度就划分为三个阶段：查找最高优先级的任务时首先查找优先级最高的任务集合，在任务集合相同的情况下，查找最高动态优先级任务集合，最后找到集合内最高优先级的任务进行调度。

5.2.1 调度算法的选择

RMS 算法有较小的运行开销，但在系统中执行周期最短的任务不一定是最重要的。EDF 和 LSF 算法都是单处理器下的最优调度算法。EDF 算法有着较小的调度开销，但算法在调度时需要分析整个任务链表，这会消耗大量的系统时间及资源，而且 Katcher 等人也证明 EDF 算法在资源受限的系统中性能会明显下降[41,42]。LSF 算法在调度时不断变化空闲时间，造成任务切换次数增多，产生抖动现象，系统开销增大，其应用范围受到一定的限制。另外当两个或多个任务具有相似的松弛时间，LSF 调度情况也将变坏。

μC/OS-Ⅱ实时调度任务时，一般使用单一的调度算法分配任务优先级。虽然单参数的优先级调度算法在某些情况下比较理想，但有时也会出现系统总体调度性能不高、算法适应能力不强的状况。而各种改进优化型的算法虽能在某种程度上改善任务的调度性能，但并不能从根本上改变这些经典算法的先天不足[43]。本文对多种调度算法综合考虑，利用多种特征参数来分配 micro-K 操作系统中的任务优先级别，得到一个优先级分配的优化算法，记为 HP(hybrid priority 混合优先级)调度算法，从而更好地进行任务调度。

5.2.2 任务模型

对 μC/OS-Ⅱ任务模型中存在任务集 S(S1,S2…,Sn)，任务 Si 重新描述如下：

Si={Ti,Pi,Ri,ai,bi,di,Vi,Di,Li}，其中 1≤i≤n，i 可取任意值。

- T 表示任务的周期。
- P 表示任务的优先级。
- R 表示任务的最坏运行时间，即任务在不被中断的情况下，从任务开始到运行完毕所需的处理器时间。
- a 表示任务的到达时间，即任务被启动并准备执行的时间(任务由其他状态转变为就绪态的那个时刻)。
- b 表示任务的剩余执行时间，即任务最坏执行时间与任务已经执行时间的差。
- t 表示任务的当前执行时刻。

◆ d 表示任务的时限(绝对截止期)，即任务完成指定操作所需的时间。

◆ V 表示任务的价值，即任务的重要程度，该值越小表示重要程度越高。

◆ D 表示任务的时限距离(相对截止期)，D=d−t。

◆ L 表示空闲时间，L=D-b。

对于第 i 个任务 Si，必须满足 Ri≤Di≤Ti，ai≤ti≤Di。

5.2.3 任务优先级的设计

在 HP 算法中，任务的优先级由价值参数，EDF 算法中的时限距离，LSF 算法中的空闲时间，RMS 算法中的周期综合确定，系统按照这 4 个参数的重要性由高到低的顺序来分配优先级。根据任务参数重要性的不同，可为任务分配不同的优先级。当两个或多个任务具有同一优先级时，HP 算法的优势更加明显。

本章首先讨论当任务具有特定的特征参数时优先级的设计思想。

设含有 n 个任务的集合 S={S1,S2,...,Sn}，对于任一任务 Si，将其价值、时限距离、空闲时间和周期的取值范围分成若干个不同的区间，从各个区间选择一个典型值来表示这个区间。

V 有 m 个典型值，分别是 V1，V2，...，Vm(V1<V2<...<Vm)；

D 的 n 个典型值为 D1，D2，...，Dn(D1<D2<...<Dn)；

L 的 p 个典型值为 L1，L2，…，Lp(L1<L2<...<Lp)；

T 的 q 个典型值为 T1，T2，…，Tq(T1<T2<...<Tq)。

任务 Sxyzσ的优先级值为

P(Sxyzσ,Vx,Dy,Lz,Tσ)(x=1,..,m;y=1,..,n;z=1,..,p;σ=1,..,q)，记为 $P_{xyz\sigma}$。改进后的任务优先级 Pxyzσ按照公式 5.4 进行动态更新。

$$P_{xyz\sigma} = \alpha Vx + \beta Dy + \varphi Lz + \omega T\sigma \tag{5.4}$$

在公式 5.4 中，α、β、φ、ω均不小于 0。当价值越小，截止期越短，空闲时间越短，周期越短所代表的 P 值越小，则所对应的任务其优先级别越高。

下面通过图示来说明 HP 算法的设计思想，为了便于理解，将四维空间图形投影到二维空间上，下面以这 6 个投影图中的 V-D 关系图(图 5.2)为例来说明。在同一优先级等级的情况下，任务优先级值沿着箭头方向递增(优先级别降低)。

对于任一任务 Si，当其价值或其他参数不是特定的特征值，可通过对事先确定的优先级关系进行线性插值来确定任务优先级，假设 V1≤V′≤Vm，D1≤D′≤Dn，L1≤L′≤Lp，T1≤T′≤Tq。在各参数的典型值中分别找到离它们最接近的三个典型值，分别记为：

Vx（x=m,m+1,m+2）

Dy（y=n,n+1,n+2）

Lz（z=p,p+1,p+2）

Tσ（σ=q,q+1,q+2）

得到点（V′,D′,L′,T′）周围的 3^4=81 个点（Vx,Dy,Lz,Tσ），每个点对应一个典型的任务 Sxyzσ，每个任务 Sxyzσ 具有自己惟一的优先级值 P（Sxyzσ,Vx,Dy,Lz,Tσ），然后利用这 81 个点对应的任务优先级值进行 Lagrange 插值[44]计算 P（Sxyzσ,V′,D′,L′,T′）。P（Sxyzσ,V′,D′,L′,T′）的计算公式见公式 5.5。

$$P(S_{xyz\sigma},V',D',L',T')= \sum_{x=m}^{m+2}\sum_{y=n}^{n+2}\sum_{z=p}^{p+2}\sum_{\sigma=q}^{q+2}\left[\prod_{\substack{k=m\\k\neq x}}^{m+2}\frac{V'-Vk}{Vx-Vk}\prod_{\substack{l=n\\l\neq y}}^{n+2}\frac{D'-Dl}{Dy-Dl}\prod_{\substack{r=p\\r\neq z}}^{p+2}\frac{L'-Lr}{Lz-Lr}\prod_{\substack{u=q\\u\neq\sigma}}^{q+2}\frac{T'-Tu}{T\sigma-Tu}\right]P(S_{xyz},Vx,Dy,Lz,T\sigma) \tag{5.5}$$

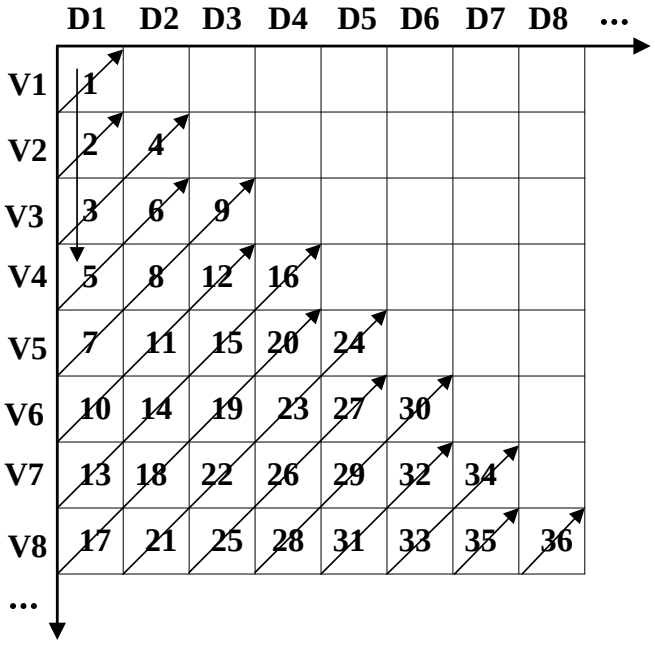

图5.2 V-D优先级关系

5.2.4 任务优先级的分配

将任务优先级字节的第 0-5 位标识为静态优先级，空闲的高两位用来标识任务的动态优先级。根据实际系统需要，选择使用高两位可以管理 128 或 256 个任务，按照 HP 算法可将同一就绪组中的 8 个任务划分成 2 个或 4 个级别的动态优先级。

HP 算法在分配任务优先级时，优先级的价值、时限距离、空闲时间、任务周期均按升序排列。具有同一优先级别的任务，由于优先级分配时各参数的比重不同，仍然可以使每个任务具有不同的优先级。

5.2.5 任务优先级的获取

动态获取优先级函数和优先级改变函数的结合使用使得任务可以动态改变优先级，并且这由 micro-K 内核负责。

获取新优先级应遵守的原则是：

- 新的优先级要高于就绪态中所有任务的优先级。
- 新的优先级要高于原先最低的优先级，衡量的标准要看比原优先级高的优先级是否被占用。

当任务时限到来，触发中断服务子程序执行优先级改变函数。在这个函数中首先检测任务 Si 是否处于运行态，若是，则计算 Si 的时限到来之前的运行时间，然后中断返回。在这种情况下，如果任务没有运行完，而被挂起，则在挂起时刻计算下次时限到来时刻(下次到来的时刻=被挂起的时刻+本次时限前运行的时间)。如果不是则继续运行中断服务程序。中断过程见下图 5.3。

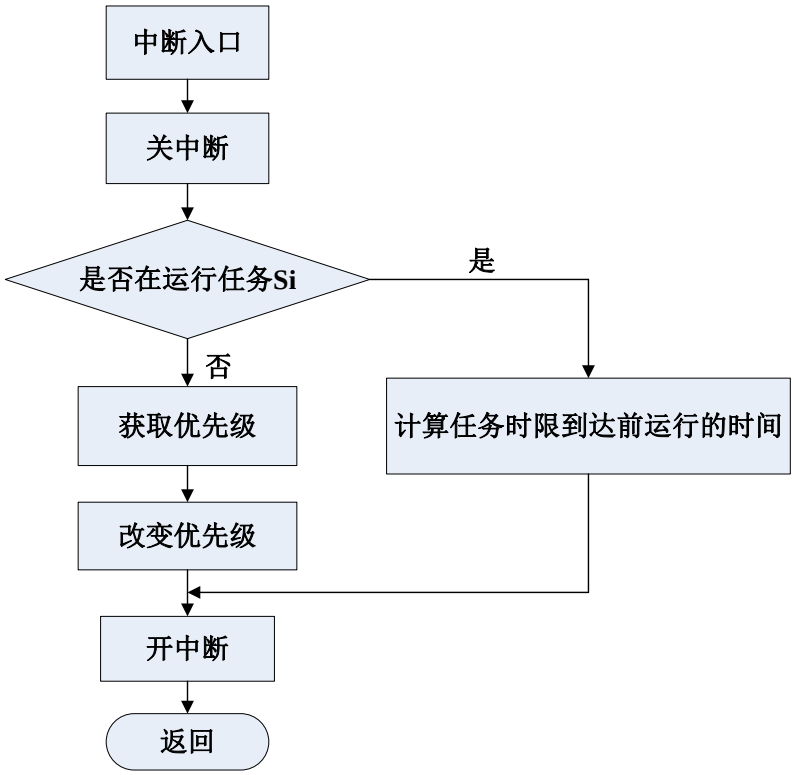

图5.3 中断服务程序流程图

5.2.6 调度实现

micro-K 内核在调度时，需要对系统中任务的优先级作出判断，寻找优先级最高的任务并调度其运行。当某一新任务进入就绪态，根据其 4 个维度的特征参数参数确定任务的优先级，并将其插入就绪队列。任务完成后，直接从就绪队列中删除此任务。由于任务调度过程中其时限距离、空闲时间等参数都在发生变化，所以需要定期扫描就绪队列，重新计算并确定系统当前具有最高优先级的任务。HP 算法采用抢占式任务调度策略。无论任务 Si 何时进入就绪队列，都要重新进行调度。HP 算法的调度原则是：

- 选择价值最大的任务优先执行。
- 如果两个或多个任务的价值相同，则选择截止期最短的任务。
- 对于价值和截止期都相同的任务，选择时间最紧迫(空闲时间小)的任务。
- 对于价值、截止期、空闲时间相同的任务，选择执行最频繁的任务。
- 如果两个或多个任务的价值、截止期、空闲时间、周期都相同，则按先来先服务的方式选择任务。

将各特征参数添加到 micro-K 内核的任务控制块中。在调度过程中，创建基于价值的任务链表 L_V、基于截止期的任务链表 L_D、基于空闲时间的任务链表 L_L 和基于周期的任务链表 L_T。这些链表都是双向的，这样可快速地插入与删除任务结点。这些操作的时间复杂度均为 O(4n)。增加参数后的任务控制块结构如下：

```
typedef struct mK_TCB{
    …
    INT8U V;                    //任务价值
    INT8U D;                    //任务时限
    INT8U L;                    //任务空闲时间
    INT8U T;                    //任务周期
    struct mK_TCB *L_V_Prev,*L_V_Next;   // L_V 任务链表前驱及后继指针
    struct mK_TCB *L_D_Prev,*L_D_Next;   // L_D 任务链表前驱及后继指针
    struct mK_TCB *L_L_Prev,*L_L_Next;   // L_L 任务链表前驱及后继指针
    struct mK_TCB *L_T_Prev,*L_T_Next;   // L_T 任务链表前驱及后继指针
    INT8U PRIO;                          //任务的优先级
    …
}mK_TCB;
mK_TCB *mK_L_V_LINK;                  //各任务链表的头结点
mK_TCB *mK_L_D_LINK;
mK_TCB *mK_L_L_LINK;
```

mK_TCB *mK_L_T_LINK;

任务各个特征参数的提取算法描述如下：

◆ 价值参数 V 的提取

(1)L_V 任务链表中任务按重要程度升序排列。

(2)新任务到达时，从头结点开始。

mK_PL_V=mK_L_V_LINK->L_V_Next;

(3)将其价值与链表中其他任务的价值进行比较，确定新任务位置。

while(mK_NewTask->V>mK_PL_V->V)mK_PL_V=mK_PL_V->L_V_Next;

(4)从 mK_PL_V 指向的任务结点开始，向后搜索，直至尾结点。

mK_NewTask->L_V_Next=mK_PL_V->L_V_Next;

(5)将新任务插入 L_V 任务链表。

◆ EDF 算法中时限参数 D 的提取

(1)L_D 任务链表中任务按时限升序排列。

(2)新任务到达时，从头结点开始。

mK_PL_D=mK_L_D_LINK->L_D_Next;

(3)将其时限与链表中其他任务的时限进行比较，确定新任务位置。

while(mK_NewTask->D>mK_PL_D->D)mK_PL_D=mK_PL_D->L_D_Next;

(4)从 mK_PL_D 指向的任务结点开始，向后搜索，直至尾结点。

mK_NewTask->L_D_Next=mK_PL_D->L_D_Next;

(5)将新任务插入 L_D 任务链表。

◆ LSF 算法中空闲时间 L 的提取

(1)L_L 任务链表中任务按空闲时间升序排列。

(2)新任务到达时，从头结点开始。

mK_PL_L=mK_L_L_LINK->L_L_Next;

(3)将其空闲时间与链表中其他任务的空闲时间进行比较，确定新任务的位置。

while(mK_NewTask->L>mK_PL_L->L)mK_PL_L=mK_PL_L->L_L_Next;

(4)从 mK_PL_L 指向的任务结点开始，向后搜索，直至尾结点。

mK_NewTask->L_L_Next=mK_PL_L->L_L_Next;

(5)将新任务插入 L_L 任务链表。

◆ RMS 算法中周期参数 T 的提取

(1)L_T 任务链表中任务按周期升序排列。

(2)新任务到达时，从头结点开始。

mK_PL_T=mK_L_T_LINK->L_T_Next;

(3)将其周期与链表中任务的周期进行比较，确定新任务位置。

while(mK_NewTask->T>mK_PL$_T$->T)mK_PL$_T$=mK_PL$_T$->L$_T$_Next;

(4)从 mK_PL$_T$ 指向的任务结点开始，向后搜索，直至尾结点。

mK_NewTask->L$_T$_Next=mK_PL$_T$->L$_T$_Next;

(5)将新任务插入 L$_T$ 任务链表。

将任务的特征参数提取后，建立一个优先级判定函数，将特征参数传入任务优先级判定函数以确定最终的优先级，之后系统就可以根据任务的优先级进行调度了。优先级判定函数如下：

```
INT8U mKCurTskPrio(INT8U mK_LV,INT8U mK_LD,INT8U mK_LL,INT8U
mK_LT)
{
    INT8U mK_PRIO;
    mK_PRIO=mK_LV+mK_LD+mK_LL+mK_LT;
    …
    return mK_PRIO;
}
```

micro-K 内核配置深度优先级调度算法和 HP 优先级分配算法后，其任务创建和任务调度以及其他函数的伪代码如下所示。

任务创建伪代码：

```
…
建立任务控制块;
建立任务堆栈mKTaskStkInit();
if(在该优先级上已有任务存在)
{
    比较任务的优先级;
    将该TCB按降序插入到优先级队列中;
}
else
{
    新任务是优先级队列上的第一个任务;
}
…
```

任务调度伪代码：

```
…
if(在该优先级上有任务存在)
```

```
  {
    if(任务数大于1)
      {
        在该优先级的队列中找到优先级别最高的任务进行调度;
      }
    else{执行任务调度;}
  }
  else{
        返回错误，任务不存在;
  }
  …
```

改进后的部分函数：

```
#defined   mKEXTASK   x          //x为一个整数，代表扩展的任务个数，
                                  若x=0表示没有扩展任务
TaskType   mKTaskReady(void)
{
  if defined(mKMIXPREEMPT)                  //混合抢占性调度
    if((mKTaskProperty[mKRunning]&mKTSKNOPREEMPTIVE) = = 0
    {
       找出就绪态中优先级最高的任务，并将优先级赋给prio;
       prioz=mKCurTskPrio[y];
       返回prioz的值；
    }
  …
}
TaskType   mKTaskReadyrun(viod)
{
   INT8U u, w, z;
   z=(u<<2)+w;                          //找出就绪态中优先级最高的任务
   …
   prioz=mKCurTskPrio[z];
   return(prioz);
 }
```

mKTaskReady()在系统创建任务时被调用，而 mKTaskReadyrun()在系统启动代码 StartmK()中和扩展任务中被调用，此时任务从运行态转到等待态。

5.3 算法评估

HP 调度算法由于优先级中包含了多种参数信息，因此对各个任务的时限情况都很了解，因此当多个任务处于相同就绪组时，通过计算把紧急、重要的任务放在前面执行，从而避免了因执行运行期长的任务而导致其他大部分任务出现阻塞导致它们超出时限的情况，增强了系统的时限控制能力。此外，在优先级反转情况出现时，系统可以提升占有系统临界资源的任务的动态优先级，使之与当前任务的动态优先级相同，实现了优先级的继承，从而解决了同组优先级的反转。

抢占式优先级调度策略要求绝大多数服务的执行时间具有确定性，并且任务的优先级必须不同，这些要求虽然简化了任务管理，但也限制一些大规模系统的应用。而 HP 优先级分配优化算法突破了原系统对任务数量的限制，并使多个任务可以有相同的优先级，无论是在正常负载还是过载情况下，都可以保证任务具有较高的截止期保证率，从而保证了 CPU 运行系统中最重要，最紧急的任务。

5.4 本章小结

本文在分析各种任务调度算法的基础上，采用了 micro-K 嵌入式操作系统内核的多参数优先级分配算法—HP 算法。该算法是一种动静结合的优先级实时调度算法，综合了两种优先级调度算法的优点，既可提高系统满足多任务时限的能力，又能解决优先级反转问题，提高了 micro-K 的实时性。

在面对大量周期性、时限要求高的任务时，HP 算法在很大程度上优化了任务优先级的分配策略，大大提高了系统对任务调度的成功率，有效地改善了任务的调度性能。

第六章 设备驱动管理设计

鉴于嵌入式系统内微处理器或微控制器的多样性，一般将嵌入式操作系统的内核分为两层，其上层称为“内核”，下层则称为“硬件抽象层”或“设备驱动层”。设备驱动层提供了各种设备的驱动程序，这些驱动程序以库函数的形式对硬件进行管理和控制。操作系统使用一组定义好的编程接口调用设备驱动层，从而访问真正的硬件[45]。micro-K 只提供了核心内核，操作系统的其他部分，例如 API 函数接口，都没有向用户提供，因此需要自行建立 micro-K 的外设驱动程序，进一步完善其功能。但是在对 micro-K 进行驱动开发时，外设的访问接口没有统一的标准，因此需要设计一个规范的外设驱动体系结构。

6.1 外设驱动管理层设计

6.1.1 PDML 架构

在 micro-K 实时内核下，应用程序直接调用外设的底层驱动函数，但没有统一、规范的外设访问接口，这种状况导致研发者开发出的设备驱动结构差异较大，非常不利于代码的重用及软件工程化。另一方面，研发者在系统扩展时还需要了解不同的微控制器或微处理器信息，若系统中硬件设备较多，需要的硬件驱动程序就增多，这样研发者需要对驱动层增加额外的管理工作，这种情况会使应用程序的编写复杂化，同时也增加了应用程序的维护难度。若为 micro-K 操作系统增加设备驱动管理层 PDML(peripheral driver management layer)，外设驱动程序就成为内核代码的一部分，二者不再分别独立，系统通过统一的接口函数就可访问底层，统一管理设备驱动程序。

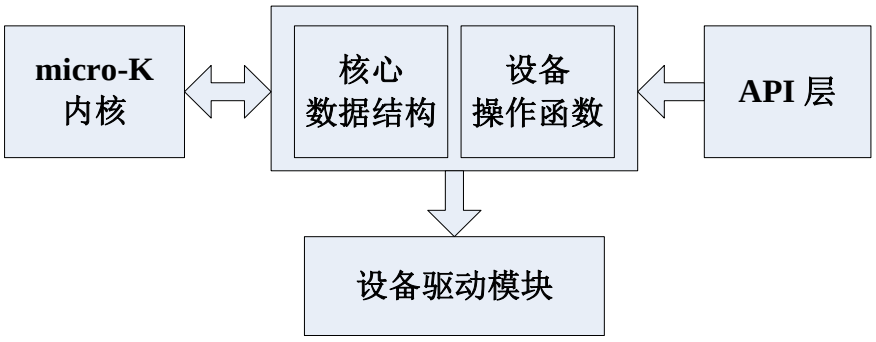

图6.1 设备驱动管理层PDML

设备驱动管理层框架如图 6.1 所示。PDML 共分为 3 部分，分别是：

- API 层：为上层应用程序提供 3 个 API 接口函数，这些函数分别是：PDML_Open()、PDML_Close()和 Create_Perp()。
- 驱动管理核心层：为每个硬件设备分配唯一的设备名，设备属性、设备状态、设备操作表等等，是 PDML 的核心，主要涉及到外设驱动索引表，驱动程序表和设备操作表等核心数据结构。
- 设备驱动模块层：硬件设备驱动功能的实现层。各个硬件设备驱动模块一般需要实现 perp_open()、perp_close()、perp_read()、perp_write()和 perp_control()等相关函数。

6.1.2 外设驱动索引表

外设驱动索引表设计成双向链表，表的大小可动态变化，可根据需要在表中添加新的设备。应用程序通过设备名访问外设驱动索引表，以此找到该设备在驱动程序表对应的索引号，再从设备操作表里找到该设备的底层操作函数。外设驱动索引表定义如下：

```
typedef struct perpnode
{struct perpnode *perpnext;
 struct perpnode *perpprev;
 unsigned short perpNum;
 PDLINK *head;
 INT8U (*PERP_GET)(INT8U p,INT8U *buffer,INT8U blen,INT8U
                   lenToRead);                    //查找设备
 INT8U (*PERP_ADD)(char *perpName,void *hd);      //添加设备
 INT8U (*PERP_DEL)(INT8U p,void *hd);             //卸载设备
}PERP_NODE;
```

应用程序研发者在查找设备函数 PERP_GET(unsigned char perpName)中，通过设备名来查找设备索引号。当调用添加设备函数 PERP_ADD(unsigned char perpName)向链表中添加新的设备节点时，需要向 micro-K 申请操作该设备所需的信号量资源，以及存储数据所用的缓冲区。卸载设备函数 PERP_DEL(unsigned char perpName)负责在链表中删除某个设备节点，并释放该设备所申请的各种资源。

6.1.3 驱动程序表

驱动程序表 PERP_DRV PerpTbl[MAX_PERP]是设备的驱动程序底层函

数，它在 micro-K 操作系统中有固定的存储空间，表的大小是可变的，最大值可由开发人员根据应用程序中使用的设备总数 MAX_PERP 变量的值来确定。在驱动程序表中，PerpTbl[i]所表示的设备其索引号为 i。驱动程序表定义如下：

```
typedef struct
 { unsigned char perpName;                          //设备名
   INT8U perpType;                                  //设备类型
   unsigned char perp_run;
   mK_EVENT *perpsem;                               //设备信号量
   PERP_OPT op;                                     //设备操作表
   INT8U PerpPrio;        //设备链表中优先级，相应的位置 1，表示该
                            优先级任务请求对此设备进行操作
   PERP_REQ *rq;                                    //设备请求链表
   PERP_OPT *opt_handle;                            //设备操作句柄
 }PERP_DRV;
```

设备操作表 PERP_OPT 结构定义如下：

```
typedef struct
 { INT8U (*perp_open )(void*hd);                    //设备打开函数
   INT8U (*perp_read)(INT8U *buffer,INT8U blength);  //设备读函数
   INT8U (*perp_write)(INT8U *buffer,INT8U blength); //设备写函数
   INT8U (*perp_control)(INT8U t,void*hd);          //设备控制函数
   INT8U (*perp_close)(void*hd);                    //设备关闭函数
   unsigned char PDRV_Install(char i,PERP_OPT ho,PERP_OPT hcls,
   PERP_OPT hr,PERP_OPT hw,PERP_OPT hc);            //设备安装函数
 }PERP_OPT;
```

设备请求链表 PERP_REQ 的结构定义如下：

```
typedef struct perpreq
 {struct perpreq *drnext;
  DRLINK *drhead;
  INT8U *buffer;
  INT8U blen;
  INT8U lenToRead;
  INT8U prio;                                       //发出读请求的任务的优先级
}PERP_REQ;
```

驱动程序表结构描述了系统设备的各种属性，并通过设备名字段 perpName 定位设备操作表 PERP_OPT。设备操作表结构定义了设备的操作函

数表，perp_open、perp_read、perp_write、perp_control 和 perp_close 分别完成设备的打开、读、写、控制和关闭，这些操作函数都是在设备驱动模块中实现的。

6.1.4 PDML 中的主要函数

驱动管理层为上层应用提供了 PDML_create()、PDML_open()和 PDML_close()共 3 个 API 函数。PDML_open()函数的伪代码如下所示。

```
INT8U PDML_open(char *perpName,void *pd)
{
 在PDML核心数据结构中查找设备名为perpName的设备i;
 if(找到指定的设备i){
  if(设备i已处于打开状态){
     if(设备i为共享设备){
          设备打开次数统计量加1;
                        }
     else{
          根据发起请求的任务将PERP_REQ链表中优先级的相应位置1;
          执行任务调度函数，使任务进入等待状态;
         if(等待队列中任务的优先级高于占用设备的任务优先级)
            将占用设备的任务的优先级调整到更高级别;
              报错并返回;
         }
     }
     else{
         从PDML中得到该设备的函数操作表并向操作系统审请打开设备;
         调用perp_open()函数对设备进行初始化;
         将设备索引i作为句柄传递回上层应用程序;
         }
}
else{
    报错“未安装此设备驱动”并返回;
   }
}
```

驱动程序表为设备提供了安装、删除、关闭及显示信息等函数。在实际操

作中，用户应用程序通过驱动程序表中的设备索引号 i 找到设备 i 及其底层入口函数。系统通过调用设备驱动安装函数将设备驱动模块植入到 PDML 中。设备安装函数如下：

```
unsigned char PDRV_Install(unsigned char i,PERP_OPT hopen,PERP_OPT
hclose,PERP_OPT hread,PERP_OPT hwrite,PERP_OPT hcontrol) //传入的
                                                    参数为底层函数
{unsigned char*err;
 unsigned char i;
 PERP_OPT *HPERP_DRV=NULL;
 mKSemPend(hperptbl_event,WAIT_FOREVER,err);          //信号量在
                                          PERP_ADD函数中定义并初始化
 if(*err = = mK_NO_ERR)
    return(FALSE);
 else{
 HPERP_DRV=&PerpTbl[i];                    //查找与索引号对应的设备
 HPERP_DRV->perp_open=hopen;               //将底层入口函数的放入表中
 HPERP_DRV->perp_close=hclose;
 HPERP_DRV->perp_read=hread;
 HPERP_DRV->perp_write=hwrite;
 HPERP_DRV->perp_control=hcontrol;
 HPERP_DRV->perp_ran=TURE;
 mKSemPost(hperptbl_event);                //释放该信号量
 if(*HPERP_DRV!=NULL)
    return(FALSE);
 else
    return(TRUE);
   }
}
```

6.1.5 PDML 工作原理

当用户应用程序要访问某一硬件设备时，首先以要打开的设备的设备名 perpName 变量作为参数，调用 API 层的 PDML_Open()函数，从而定位要操作的设备。PDML_Open()通过 perpName 参数找到相应设备的核心管理数据结构，进而得到相应设备的操作函数表。系统定位到相应设备的驱动模块后，先要调用设备驱动模块的 perp_open()函数完成设备的初始化。当初始化完成后，

PDML_Open()函数将一个句柄传递回应用程序，便于应用程序调用相应的设备驱动模块进行后续的操作，这样通过 API 接口函数实现对设备的统一访问控制。

驱动设备管理层为千差万别的设备抽象出一个统一的接口，为上层操作系统控制设备驱动程序带来了方便，它不局限于某个特定的开发平台，具有通用性。在具体的开发环境中，底层开发人员还需要根据不同的硬件平台编写自己的底层操作函数。

由于嵌入式系统有很强的针对性，且其操作系统的内存空间比较有限，因此一般将驱动程序直接编译到内核中，使其成为内核中的一个模块。当应用程序在内核外调用设备驱动时，micro-K 内核以库函数的形式对其提供设备驱动，但由于 micro-K 的内核空间是一个整体，不区分为系统空间和用户空间，因此驱动程序仍然运行于内核模式[46]。

6.2 底层接口的具体实现

本文采用 C8051F120 单片机硬件环境，介绍在 micro-K 上编写外设驱动程序的一般思路。

工业实践中的系统大都具备数据采集功能，A/D 转换是数据采集的重要组成部分，下面分析在 micro-K 实时内核下 A/D 驱动程序的设计和实现。

6.2.1 C8051F120 单片机中的 A/D 转换器

C8051F120 提供了一个 8 路的 8 位 ADC 和一个 8 路的 12 位 ADC，共有 16 路 A/D 采样通道，按照单片机的 A/D 转换时序要求，就能将模拟量信号转换成 8 位或 l2 位的数字量，能够满足绝大多数系统通道数目和精度的要求。

典型的 A/D 转换电路一般由模拟多路复用器(MUX)、放大器和模数转换器(ADC)三部分组成。C8051F120 片内 12 位的逐次逼近寄存器型 ADC(ADC0)包括一个 9 通道的可编程模拟多路选择器(AMUX0)，一个可编程增益放大器(PGA0)，ADC0 中还集成了跟踪保持电路和可编程窗口检测器。当最大采样速率达到 100ksps 时，ADC0 可提供 12 位分辨率。而片内的 8 位 ADC(ADC2)带有一个 8 通道输入多路选择器和可编程增益放大器。当最大采样速率达到 500ksps 时，ADC2 可提供 8 位分辨率[47]。

ADC 主要有两种触发方式：内部触发方式(软件命令、定时器 2 溢出、定时器 3 溢出)及外部触发方式(外部信号输入)。这样用软件事件、外部硬件信号或周期性的定时器溢出信号就可进行触发转换，触发方式非常灵活。可以使用

查询状态位或者产生中断的方法结束转换。

ADC0 的最高转换速度为 100ksps，其转换时钟(SAR0)来源于系统时钟分频(100MHz)。ADC0 有 4 种启动方式，由 ADC0CN 中的 ADC0 启动转换方式位(AD0CM1，AD0CM0)的状态决定。当进行 A/D 转换时，将 AD0BUSY 位置“1”，转换结束后置“0”。ADC0 转换时钟周期及 ADC0 内部放大器增益由 ADC0 配置寄存器 ADC0CF 来控制的。ADC2 的最高转换速度为 500ksps，转换时钟频率最大为 6MHz。ADC2 有 5 种 A/D 转换启动方式，由 ADC2CN 中的 ADC2 启动转换方式位(AD2CM2-0)的编程状态决定。

6.2.2 micro-K 下 A/D 驱动

1. A/D 的读取方法

在 micro-K 内核下，应该首先考虑驱动程序采用哪种方法读取 A/D 采样数据，因为系统中的许多因素都将影响 A/D 的读取，比如说 A/D 的输入通道数、转换时间及模拟值的转换频率等，但在实时内核下 A/D 驱动程序的实现过程主要取决于 A/D 转换器的转换时间。

C8051Fxxx 系列单片机中 ADC 的速率都可由程序员在编程时根据需要而设置[48]。在 C8051F120 单片机中，ADC 的转换时钟应不大于 2.5MHz。ADC 完成一次转换需要用 16 个系统时钟，而 ADC 在启动之前一直处于跟踪方式，这样在转换之前还要加上 3 个系统时钟的跟踪/保持捕获时间，因此 ADC 完成一次转换最多需要需 19 个系统时钟(7.6μs)。对于 ADC 是选择查询方式还是中断方式各有利弊。micro-K 实时操作系统是基于占先式调度方式的，它对中断的处理和一般的前后台系统是不一样的。micro-K 在运行完中断服务程序之后，并不一定返回到被中断的任务上去，而是要进行一次任务调度来决定是返回被中断的任务还是运行一个具有更高优先级别的就绪任务。任务切换的时间以 ms 来计量，而 ADC 完成一次转换的时间是 7.6μs，所以为了减小 CPU 的开销，系统应该选择查询方式。

图 6.2 是一种典型的读取方法—轮询驱动法。应用程序调用图 6.2 所示的驱动程序，并传递要读取的通道。轮询驱动法的步骤如下：

① 驱动程序通过 MUX 选择要读取的模拟通道开始读取信号。

② ADC 被驱动程序触发开始进行转换(在 A/D 转换前需要延时几 μs 以便稳定信号)。

③ A/D 转换结束后，驱动程序进行循环等待状态。

④ 驱动程序循环检测 ADC 的状态信号。若检测到 ADC 转换结束信号时，驱动程序读取转换结果；若等待超时时，则结束等待循环。

⑤ 读取转换结果。

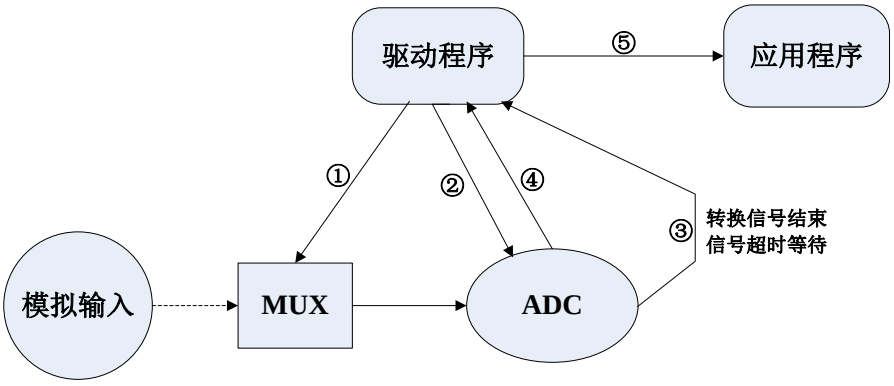

图 6.2 A/D 读取方法

轮询驱动方法的优点表现在：A/D 的转换速度比较快，可获得较短的转换时间，不需要增加额外的 ISR，信号转换时的改变时间较短，微处理器的系统开销小，因此对 C8051F120 单片机而言比较适合采用这种 A/D 读取方法。

2. A/D 驱动程序的编写

将 micro-K 移植到 C8051F120 微控制器上后，就要在这个实时内核之上编写接口驱动程序。利用 C8051F120 的 12 位 ADC 来进行采集，采集前需要进行相应的配置。A/D 驱动程序主要是对设备的 A/D 接口进行控制，使其能够与通讯另一方进行数据交换，在 micro-K 内核中，ADTask()任务负责控制 A/D 转换器。管理模拟输入。通过驱动程序读取模拟输入通道可将实现的细节隐藏在程序里，在用户使用时可以使用驱动程序而无需改变应用程序。micro-K 内核使 A/D 的使用变得更加可靠、有效。

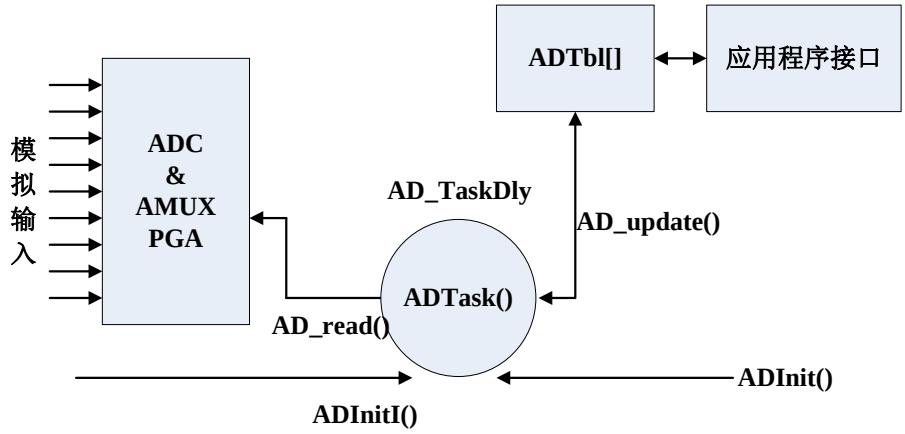

图6.3 A/D驱动程序模块流程图

A/D 驱动程序模块流程如图 6.3 所示。A/D 驱动程序可分为 5 个主要部分：A/D 接口的初始化，A/D 接口的关闭，选择合适的模拟输入，读取 A/D 接口的模拟输入通道，A/D 接口的通讯参数设定，这 5 个部分的功能分别对应 C8051F120 平台下的 AD_init()，AD_close()，AD_read()，AD_update()，AD_ctrl()

底层通用接口函数。

各数据结构、常量和 A/D 驱动程序接口函数的定义如下：

AD ADTb1[AD_MaxNum]：AD 类型的数组，由 AD_MaxNum 的值确定数组的大小。

AD AD_Prio：ADTask()转换任务的优先级别。

AD AD_ChnNum：AMUX 的模拟输入通道数。

AD AD_ChnDly：更新模拟通道的时间间隔。

AD AD_StkSize：分配给 ADTask()转换任务的堆栈大小。

ADTb1[]：模拟输入通道信息、ADC 硬件状态参数配置及转换结果存储表。

void AD_init(void)函数：对所有的模拟输入通道、A/D 控制器以及应用程序的调用参数进行初始化，同时创建 ADTask()任务。在使用任何其他 A/D 转换模块的函数之前，必须先调用该函数。

INT16U AD_read(INT8U chn)：定义如何读取 A/D 以及返回 ADC 转换结果到 AD_update()。

void AD_update(INT8U n)：在 AD_ChnDly 时间内更新输入通道。

void AD_inithw(void)：初始化硬件 A/D 控制器。

void ADTask(void data)：由 AD_init()创建，调用 AD_update()函数更新输入通道。

3. 外设驱动程序管理层对 A/D 设备的访问

micro-K 中 A/D 驱动程序的植入由底层开发人员完成。首先要初始化外设驱动索引表，通过 PERP_ADD(AD)函数在表中添加一个名为 AD 的设备，然后返回该设备的索引号，并将驱动程序表中相应的 perp_ran 参数置为 TURE，这样就在外设驱动索引表、驱动程序表以及设备操作表中建立了该设备的相关参数和操作。

PDRV_Install(AD,AD_init,AD_close,AD_read,AD_update,AD_ctrl) 函数最后被调用执行，该函数将 A/D 驱动程序函数的入口加入到驱动程序表中。最后通过中断调用将 A/D 驱动程序加入到 micro-K 操作系统中。

当应用程序启动 A/D 驱动程序进行数据采集时，通过设备名 AD 执行 PERP_GET(AD)函数得到设备在驱动程序表中的索引号 i。再根据索引号执行 PDrvFind(i)函数，返回驱动程序表与该设备对应的设备入口结构体 PERP_DRV。PERP_DRV 结构体中有该设备的底层函数入口，应用程序研发者调用这些函数就可执行设备相应的操作。完成以上步骤后，应用程序便可通过 PDML 访问设备，这对研发者来说是既方便又有效。

6.2.3 micro-K 下串口驱动

C8051F 系列单片机为用户提供了大量的外围接口，可以根据实际需要进行功能扩展[49]。C8051F120 单片机片内的串行通信接口主要包括 2 个异步串行(UART)接口，1 个系统管理总线(SMBus)接口以及 1 个串行外设(SPI)总线接口。UART(Universal Asynchronous Receiver/Transmitter)是独立的、可编程的全双工异步串行通信接口，还可作为同步移位寄存器使用。每个 UART 接口内部有 2 个相互独立、地址相同的数据接收和数据发送缓冲器 SBUF，可以工作在查询模式或中断模式下，其波特率的大小可根据微处理器的时钟振荡频率设定。

1. 串口操作模式及原理

micro-K 利用信号量来完成任务间的通信和同步。每个串口都配有发送与接收两个环状缓冲队列，通过定义的数据接收与发送两个信号量，使缓冲区两端的操作得以同步进行。串口的缓冲队列定义如下：

```
typedef struct{
INT16U RBuf_RXCnt;              //接收缓冲区中的字符数
INT16U RBuf_TXCnt;              //发送缓冲区中的字符数
mK_EVENT *RBuf_RXSem;           //任务接收信号量
mK_EVENT *RBuf_TXSem;           //任务发送信号量
INT8U RBuf_RX[RX_SIZE];         //接收环形缓冲区
INT8U *RBut_RIPtr;              //接收缓冲中下一个写入字符的位置
INT8U *RBuf_ROPtr;              //接收缓冲中下一个待取出的字符的位置
INT8U RBuf_TX[TX_SIZE];         //发送环形缓冲区
INT8U *RBuf_TIPtr;              //发送缓冲中下一个写入字符的位置
INT8U *RBut_TOPtr;              //发送缓冲中下一个待取出的字符的位置
}UART_RBUF;
```

micro-K 中的 UART 大多工作在中断模式下，串口驱动以中断的形式实现。在中断方式下，串口的接收与发送向上层提供两个接口函数：发送函数 UART_TXChr()和接收函数 UART_RXChr()。当用户任务要发送数据，但发送缓冲队列已满，这时需要将任务挂起，将 CPU 的使用权交给别的任务，待缓冲队列中的数据被发送 ISR 读走之后，唤醒处于睡眠态的任务；同理，当用户任务要接收数据，但接收缓冲队列为空，这时也需要使任务处于睡眠态，待接收缓冲队列中有外部设备的相关数据时再唤醒该任务。

在 micro-K 嵌入式操作系统下，串口驱动的原理是：在 UART 发送/接收完数据的过程中，通过发送/接收信号量获取发送/接收环形缓冲区中的可用数据。当发送/接收操作完成之后，调用中断服务子程序，发送/接收 ISR 向串口驱动程序发出信号量，通知发送/接收任务已经完成。若发送/接收信号量超时，

则转而执行其他任务[50]。

2. 串口发送/接收的相关函数

(1)与操作系统相关的接口函数

在 C8051F120 平台下，串口驱动程序通过调用相关的函数来访问和控制 UART，与 micro-K 相关的底层接口函数有：

void UART_Init(void);

INT8U UART_RXChr(INT8U cha,INT16U tda,INT8U err);

INT8U UART_TXChr(INT8U cha,INT8U s,INT16U tda);

BOOLEAN UART_RXEpt(INT8U cha);

BOOLEAN UART_TXFul(INT8U cha);

UART_Init()函数对整个串口软件模块进行初始化，包括环形缓冲区以及接收/发送信号量。

利用 UART_RXEpt()和 UART_TXFul()函数来查询缓冲器的状态。其中 UART_RXEpt()函数允许应用程序检查接收缓冲区是否为空。UART_TXFul()函数允许应用程序检查发送缓冲区是否已满。

应用程序通过调用 UART_RXChr()函数读取接收缓冲区中的数据，通过调用 UART_TXChr()函数将数据写入发送缓冲区。

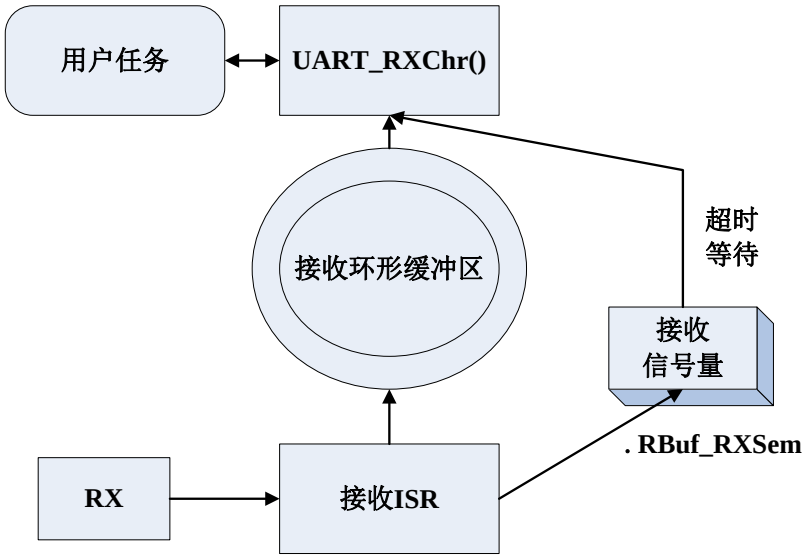

图6.4 串口接收数据示意图

当串口要读取数据时，用户任务调用 UART_RXChr()，触发接收中断，接收数据的 ISR 读出 UART 控制器中的字节，并将其放入环形接收缓冲，接着用接收信号量. RBuf_RXSem 将用户任务端的读操作唤醒。在接收过程中，通过调用 UART_RXEpt()函数可避免因缓冲区中没有数据而挂起应用任务。UART 接收数据的示意图如图 6.4 所示。

当用户任务要向串口发送数据时，首先要让任务发送信号量处于等待态，

检测发送缓冲区是否已满。只要发送缓冲区有空间，就能进行写入操作。写入过程是按照写入单个字节，触发中断，发送 ISR 取出此字节并输出至 UART，再次触发中断的步骤循环进行，直到发送缓冲为空[51,52]。图 6.5 是 UART 发送数据的示意图。

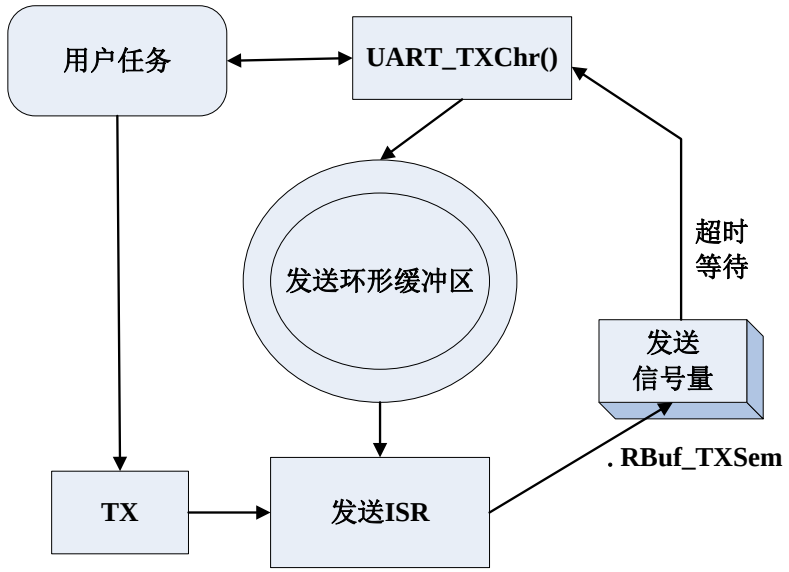

图6.5 串口发送数据示意图

(2)与 C8051F120 硬件平台相关的接口函数

```
void UART_SetINT(INT8U cha);      //设置 UART 的中断向量
void UART_MppINT(INT8U cha);      //还原 UART 的中断向量
```

(3)对 UART 进行控制、访问的操作函数

```
INT8U UART_Port(INT8U cha,INT8U bd,INT8U bt,INT8U p,INT8U s);
//对串口控制器进行初始化，设置端口的波特率、奇偶校验和停止位
void UART_RX_Empty(INT8U cha);              //清空接收寄存器
void UART_RXINT_Close(INT8U cha);           //关闭接收中断
void UART_RXINT_Open(INT8U cha);            //开启接收中断
void UART_TXINT_Close(INT8U cha);           //关闭发送中断
void UART_TXINT_Open(INT8U cha);            //开启发送中断
```

3. 中断服务程序

(1)发送中断服务子程序(ISR)的伪代码如下：

```
UART_TX ISR()
{
    保存 C8051F120 的全部寄存器，保存现场;
    mKIntEnter();                          //通知 micro-K 进入 ISR
    if(接收环形缓冲区为空){
    关闭中断，返回;
```

```
        }else{
            发送 ISR 从环形缓冲中读出一个字节并输出到 UART 中;
            }
        mKSemPost(UART_RBUF. RBuf_TXSem);  //发出发送完成信号量
        mKIntExit();                        //通知 micro-K 退出 ISR
        恢复 C8051F120 的现场;
        IERT;
    }
```

(2) 接收中断服务子程序(ISR)的伪代码如下：

```
  UART_RX ISR()
  {
        保存 C8051F120 的全部寄存器，保存现场;
        mKIntEnter();                       //通知 micro-K 进入 ISR
        接收 ISR 从 UART 寄存器读出一个字节;
        if(接收环形缓冲区已满){
        舍弃收到的字符;
        }else{
        将字节放入环形接收缓冲区;
        }
        mKSemPost(UART_RBUF. RBuf_RXSem);  //发出接收完成信号量
        mKIntExit();                        //通知 micro-K 退出 ISR
        恢复 C8051F120 的现场;
        IERT;
  }
```

6.3 设备驱动程序的编写原则

对于 micro-K 下的其他的设备驱动程序，其编写的基本原则主要有以下几点：

- ◆ 对于自身具有中断功能的设备(例如串口)，可将用户需要的服务缓存于就绪队列中，运行时通过中断调度依次获取服务。
- ◆ 对于周期性的设备(例如键盘和 LCD)，需要单独建立任务，运行时通过任务间通信，与其他驱动程序同步调度。
- ◆ 对于不具有周期性，且本身又不具备中断的设备，若运行过程中速度较慢，则需用信号量来独占该设备。

PDML 在 micro-K 嵌入式操作系统下具有通用性，并不依赖于特定的开发环境，在实际的开发过程中，还需要研发人员根据具体的硬件平台编写各种底层操作函数。

6.4 本章小结

本章为 micro-K 设计了一个对设备进行统一管理的外设驱动程序管理层，在这个通用驱动层下，可以不针对系统硬件的差异，将底层驱动程序模块化，同时为上层的应用提供统一的 API 接口函数，从而实现了对系统设备的有效管理。其次，在 micro-K 内核移植到 8 位单片机 C8051F120 的基础上，以 A/D 驱动程序和串口驱动为例对底层驱动的编写进行了研究。

在 micro-K 上扩充设备驱动程序管理层这种方式既可简化应用程序编程，有效缩短了应用软件程序的开发周期，也可使应用程序的维护变得更加简易。

第七章 针对 micro-K 内核的性能测试

7.1 测试环境

本文利用 C851F120 单片机工控板(图 7.1)和 JTAG 仿真器作为测试工具，将操作系统直接加载到工控板的存储器进行测试。适配器使用 SiliconLab 公司的 EC2。

本实验系统采用集成有 Keil uVision3 开发工具的 Silicon Laboratories IDE 进行软件开发与调试。Silicon Laboratories IDE 是 Silicon Laboratories 公司免费提供的一套完整独立的软件程序，是专为 C8051F 系列单片机提供的开发环境。Silicon Laboratories IDE 包括了设计者可以用于开发和测试项目的所有工具。该集成开发环境以项目为单元进行开发和管理，Silicon 配置向导可为指定的目标环境产生配置代码。支持第三方开发工具，通过工具链接集成可以提供与第三方的汇编器、编译器以及链接器进行有机集成。支持观察和修改存储器和寄存器，支持断点、观察点、堆栈指示器、单步、运行和停止命令。调试时不需要额外的目标 RAM、程序存储器、定时器或是通信通道，所有的模拟和数字外设都正常工作[53]。Silicon Laboratories IDE 开发环境见图 7.2。

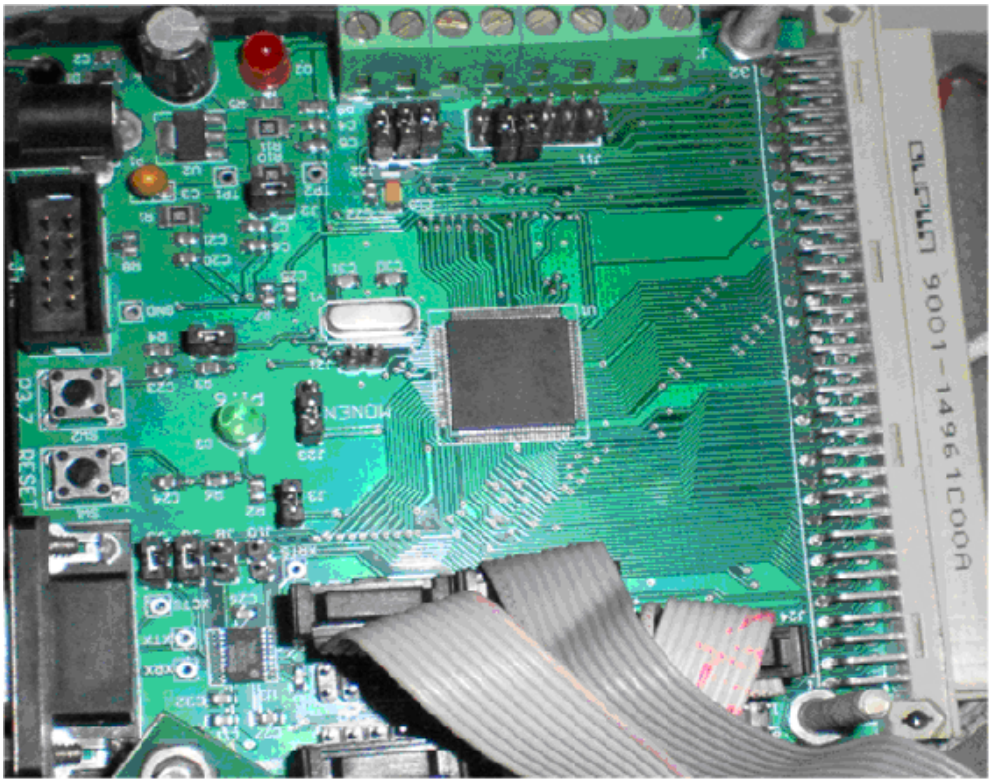

图 7.1 **C851F120 单片机工控板**

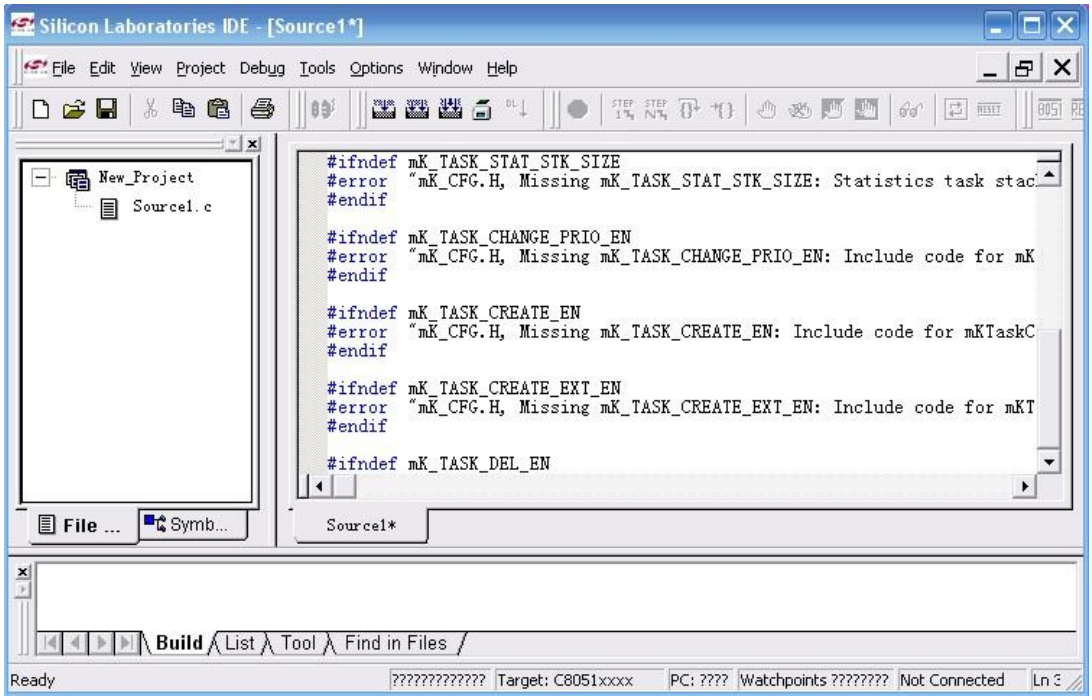

图 7.2 **Silicon Laboratories IDE 开发环境**

7.2 micro-K 在测试平台上的加载过程

测试 micro-K 嵌入式操作系统的性能，首先要完成内核在 C8051F120 微控制器上的加载工作，这部分工作主要是修改与 CPU 相关的文件，与 CPU 无关的内核文件可直接嵌套调用[54]。micro-K 内核与 CPU 相关的文件有汇编文件 mK_CPU_A.ASM，头文件 mK_CPU.H 以及 C 文件 mK_CPU_C.C。

7.2.1 mK_CPU.H 文件的修改

mK_CPU.H 文件声明了各种与处理器相关的数据类型、宏以及常量等基本信息。

1. C8051F120 单片机采用与 MCS-51 兼容的 CIP-51 内核，其堆栈按字节操作，且堆栈从低地址向高地址的方向增长(0=向上，1=向下)，故堆栈 mK_STK 的相关声明如下：

```
typedef   INT8U   mK_STK              //堆栈入口的宽度为 8 位
#define   mK_STK_GROWTH   0           //堆栈从下往上增长
```

2. 宏的定义

C8051F120 对临界区采用直接开关中断的处理方法，EA(中断允许寄存器 IE 的第 7 位)为中断允许控制位，对临界区的设定语句如下：

```
#define  mK_ENTER_CRITICAL  EA=0   //关中断，进入临界区
#define  mK_EXIT_CRITICAL  EA=1    //开中断，退出临界区
```

micro-K 的任务切换以中断的形式完成，由于 C8051F120 没有软中断机制，所以需要用函数模拟中断来实现任务切换，对任务切换宏的定义语句为：

```
#define  mK_TASK_SW  mKCtxSw()     //mKCtxSw()实现任务切换
```

3. 数据类型的定义。

数据类型和长度应根据 C8051F120 微处理器的特点来定义。

7.2.2 mK_CPU_C.C 文件的修改

mK_CPU_C.C 包含 1 个 mKTaskStkInit()函数和 mKTaskCreateHook()、mKTaskDe1Hook()、mKTaskSwHook()、mKTaskStatHook()、mKTimeTickHook()等 9 个 Hook 函数。Hook 函数扩展 micro-K 的功能，可不包含代码，但必须声明。mKTaskStkInit()函数用来初始化任务的堆栈结构，由 mKTaskCreate()和 mKTaskCreateExt()函数调用使用。

1．mKTaskStkInit()任务堆栈初始化函数的设计

mK_CPU_C.C 文件中最重要的函数是任务堆栈初始化函数，该函数返回堆栈的最低地址以及堆栈的长度，其伪代码如下所示：

```
void *mKTaskStkInit(void(*task)(void*pd),void*ppd,void*pt,INT16U opt)
reentrant
 {
   mK_STK   *stk;                    //堆栈指针stk
   ppd=ppd;                          //栈的大小由ppd的值确定
   opt=opt;
   stk=(mK_STK *)pt;                 //堆栈最低有效地址
   任务地址task入栈;
   将PC、PSW、ACC、B、DPL、DPH、R0-R7寄存器依次入栈;
   将栈顶指针SP返回给调用该函数的函数;
  }
```

2．系统时钟初始化程序的设计

将 micro-K 内核移植到 C8051F120 硬件平台时需要在 mK_CPU_C.C 文件中添加系统时钟初始化函数 mKInitTimer0()，使用 C8051F120 单片机的定时器 0 作为系统时钟中断源，mKInitTimer0()时钟初始化函数需要设定时钟的工作模式、时间初值以及使能中断，其代码如下：

```
void  mKInitTimer0(void)  reentrant
```

```
{
  SFRPAGE=0x00;            //切换寄存器页
  TMOD=TMOD&0xF0;
  TMOD=TMOD|0x01;          //模式1(16位定时器)，仅受TR0控制
  TH0=0x2C;                //定义Tick=20次/秒
  TL0=0x12;
  ET0=1;                   //允许T0中断
  TR0=1;
}
```

7.2.3 mK_CPU_A.ASM 文件的修改

mK_CPU_A. ASM 文件用于对 CPU 寄存器进行读写，主要包括 4 个汇编函数：mKStartHighRdy()、mKCtxSw()、mKIntCtxSw()和 mKTickISR()，这些函数都是不可重入的。

1. mKStartHighRdy()函数：运行就绪队列中优先级最高的任务，该函数由 mKStart()多任务启动函数调用。mKStartHighRdy()先将系统运行标志位 mKRunning 置为 TRUE，然后保存优先级最高就绪任务的现场(主要是寄存器的值)，并将该任务的栈顶指针装入 SP 中，最后中断返回，系统运行优先级最高的就绪任务。

2. mKCtxSw()函数：任务间切换函数，由任务级调度函数 mKSched()通过宏定义“#define mK_TASK_SW mKCtxSw()”调用。mKCtxSw()函数的作用是保存被中止运行任务的现场，恢复当前优先级最高的任务的现场并将这个任务启动。函数的关键代码如下：

```
MOV A,SP
SUBB A,#mKSStartptr     ;mKSStartptr 为系统堆栈起始地址
MOV R3,A                ;获得堆栈的长度
mKOS_TO_mKSTK:          ;将系统堆栈中的数据复制到任务堆栈中
INC DPTR
INC R0
MOV A,@R0
MOVX @DPTR,A
DJNZ R3,mKOS_TO_mKSTK
```

3. mKIntCtxSw()函数：中断服务程序中的任务切换函数。mKIntCtxSw()除了不需保存寄存器的内容之外，其他的大部分代码类似于 mKCtxSw()。

4. mKTickISR()函数：时钟节拍程序，实现时间的延迟、超时功能。该函数需要完成的工作是保存寄存器，调用 mKIntEnter()、mKTimeTick()以及 mKIntExit()函数，恢复寄存器，中断返回。mKTickISR()时钟节拍中断服务子程序的源代码为：

```
LJMP  mKTickISR      ;使用定时器 0
mKTickISR:
USING 0
CLR EA               ;屏蔽中断
PUSHREG
LCALL mKIntEnter     ;中断入口
CLR T0               ;清零
MOV TH0, #0x2C       ;每 50ms 产生一次定时中断
MOV TL0, #0x12
SETB TR0
LCALL mKTimeTick
LCALL mKIntExit
POPREG
RETI
```

7.3 micro-K 内核的启动与调试

在实际应用中，计算机测控系统中的微处理器都集成了片内 ROM 或 FLASH 存储器，操作系统一般在 ROM 或 FLASH 中运行。C8051F120 中 FLASH 存储器的容量为 128K，而 micro-K 内核的全部代码却不足 10K，经裁减后的核心代码可压缩到 3K 左右，因此完全可将 micro-K 内核加载到 C8051F120 的 FLASH 中运行。micro-K 内核与应用程序共同编译成一个文件，将其烧写到固化的 FLASH 存储器中，系统上电后 C8051F120 微处理器便可直接从 FLASH 中启动操作系统。

基于 micro-K 的嵌入式系统采用 EC2 仿真器进行在线调试。但是作为一个多任务的实时操作系统，micro-K 中的程序并不是顺序执行每个循环，它采取的是频繁地、大范围的跳跃执行方式。因此当在进行系统调试时，对于模块内部的调试，利用了 Silicon Laboratories IDE 和 EC2 提供的调试手段，如断点、单步执行等。

7.4 测试结果和性能分析

7.4.1 micro-K 内核任务数的测试

1. 对系统任务总数的测试

对 micro-K 内核中的任务总数进行测试应一直创建任务直到失败为止，最终显示出在测试过程中创建成功的任务数。对 micro-K 内核进行测试的代码如下所示。

```
void   main(void)
{
  INT8U   i=0;
  INT8U   ret=mK_NO_ERR;
  …
  while(ret = = mK_NO_ERR)
  {
  ret=mKTaskCreate(task_test,(void*)0, (void*)TaskStack[i++],0);
  }
  printf(“i=%d”,i);
}
void   task_test(void *data)
{ while(1){
            mKTimeDly(500);
            }
}
```

测试结果表明，该程序打印输出“i=120”，由于系统保留了最高 4 个和最低 4 个任务优先级，因此说明任务数扩充成功，micro-K 内核支持的任务数为 128。

2. 与 μC/OS-Ⅱ内核的比较

表 7.1 列出了 micro-K 的项目测试数据，并与 μC/OS-Ⅱ内核的测试结果进行对比。

表 7.1 中的数据说明 micro-K 内核在进行任务调度时数值会有所下降(测试 2)，这是因为 micro-K 内核中的优先级位图调度算法需要进行深度调度处理，增加了运算处理时间。如果能在汇编层面上对运算进一步优化，可以得到更理想的结果。另外，测试 3 的数值也有所下降，因为事件就绪表中的数据结构也进行了相应的改进。

表 7.1 测试数据

测试项目	迭代值	
	micro-K	μC/OS-Ⅱ
测试1—基准测试	1483	/
测试2—抢占式的任务调度测试	210558	214055
测试3—任务同步处理测试	492580	529713

7.4.2 micro-K 实时性能的测试

本章选择任务切换时间和中断响应时间来对 micro-K 内核的实时性能进行测试与分析。

任务切换时间(Task Switch)是指 CPU 的控制权由正在运行的任务转移到另一个处于激活态的任务这一过程所需要的时间，是评估嵌入式操作系统实时性能的重要指标之一。任务切换时间包括保存当前任务上下文的时间、选择新的运行任务的调度时间以及新任务上下文的恢复时间。任务切换时间不仅取决于中央处理器的结构和指令集，也与操作系统所采用的调度算法以及任务上下文切换时处理数据结构的效率有关[55]。

中断响应时间指从中断发生到开始执行用户中断服务程序的第一条指令所用的时间，是衡量嵌入式操作系统实时性能的、最具代表性的指标之一。中断响应时间包括中断延迟、保存 CPU 内部寄存器的时间以及内核中断服务程序的处理时间。

加载到 C8051F120 硬件测试平台上的 micro-K 内核与 μC/OS-Ⅱ内核的任务切换时间和中断响应时间的比较如表 7.2 所示。

表 7.2 内核实时性能比较

内核	任务切换时间	中断响应时间
micro-K	6.5μs	2.4μs
μC/OS-Ⅱ	6.3μs	2.4μs

从表 7.2 可知，micro-K 内核的任务切换所花的时间和 μC/OS-Ⅱ内核相当，与系统中建立的任务数目无关。micro-K 内核的中断响应时间与 μC/OS-Ⅱ内核一致，这个结果满足一般中小型嵌入式应用系统的要求。由此可知：micro-K 内核整体性能与 μC/OS-Ⅱ内核基本一致，基于 μC/OS-Ⅱ内核所做的开发程序

可以不作修改就能在 micro-K 内核下运行。

7.4.3 HP 调度算法性能测试

1. micro-K 采用 HP 调度算法后的结果

μC/OS-Ⅱ原有的实时优先级抢占调度算法开销小，也比较简单灵活，而 micro-K 内核的 HP 算法则比较复杂，会使 CPU 的一部分资源都用于调度器上，但是 HP 算法能保证 CPU 运行系统中最重要、最紧急的任务。

系统测试时，测试 3 可以较好地反应调度算法的速度，因此，测试以抢占式的任务调度测试作为基础，将测试任务数分别设为 l，2，5，8，14，19，24，通过这 7 组测试对 micro-K 内核的 HP 算法与 μC/OS-Ⅱ内核的静态算法进行全面对比。测试结果见表 7.3。

表 7.3 两种不同任务调度算法的测试结果

任务数	静态算法	HP算法
1	**156291**	**159430**
2	**285037**	**161835**
5	**611430**	**595300**
8	**376427**	**360511**
14	**542610**	**540039**
19	**173405**	**175462**
24	**812514**	**871125**

从表 7.3 中可以发现，对于任务数较少的简单应用，原有的静态调度算法比较令人满意，当系统中的任务数增多后，相比较而言，micro-K 内核的 HP 算法则拥有更好的性能。当系统调用多个任务时，HP 调度算法能使任务的超时概率比改进前有半数左右的降低，并且不会发生优先级反转的情况。

2. 差分截止期错失率

实时任务在调度时总是优先调度价值较大，相对重要的任务。差分截止期错失率 DMDR(difference missed deadline rate)[56]是描述系统运行中任务的实时性能是否满足需求，衡量调度策略好坏的一个重要性能指标。本章采用差分截止期错失率比较不同算法对于重要程度不同的任务所提供的差分服务能力。

设在一定时间内，系统中错过截止期未被调度成功的任务数为 p，正常完成调度的任务数为 q，则差分截止期错过率的计算公式 7.1 如下：

$$DMDR = \frac{p}{p+q} \tag{7.1}$$

差分截止期错失率与调度成功率成反比，差分截止期错失率越高，任务调度成功率就越低。

根据 micro-K 内核的任务模型，随机建立 20 个任务，任务的参数产生的方法为：

- 任务的执行时间 R 的范围为 10ms—20ms，服从均匀分配；
- 任务的到达时间 a 的取值范围为 0ms—15ms；
- 任务的松弛时间 S 在 15ms—25ms 范围内，服从均匀分布；
- 截止期 D 范围为 20ms—30ms，服从均匀分布；
- 周期 T 的范围为 30ms—45ms，服从均匀分布；
- 任务的价值 V 服从 10—100 之间的均匀分布，并且所有任务被划分到关键程度不同的 k(1≤k≤10)个类别中，其中 k 越小表示该类别任务的关键程度越高，任务 Ti 属于第 k 类当且仅当 k*10<Vi≤(k+1)*10。

这 20 个任务分别运用 RMS、EDF、LSF 以及多参数的 HP 调度算法进行任务调度，任务实际完成时间由 micro-K 内核提供的函数 mKTaskStat()来统计计算。计算出任务的 DMDR，并将 mKTaskStat()计算的完成时间与任务的截止期比较。各种调度算法的 DMDR 比较如图 7.3 所示，其中横坐标是实验次数，纵坐标是调度算法的差分截止期错失率 DMDR。

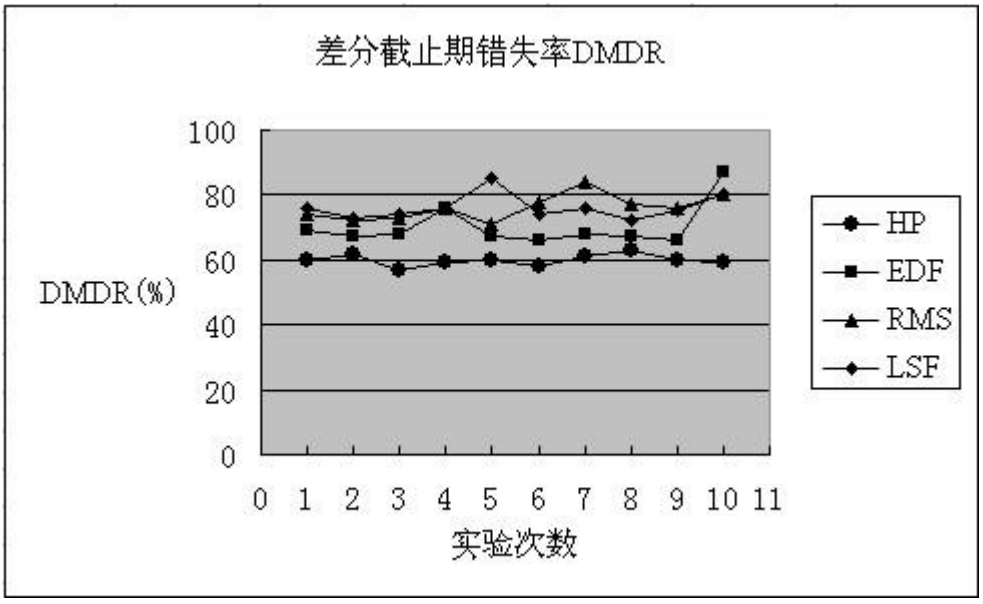

图 7.3 差分截止期错失率 DMDR

从曲线图 7.3 可以看出，在 micro-K 内核中，使用 HP 调度算法的任务截止期的错过率要比单一使用 EDF，LSF，RMS 调度算法的任务截止期错过率低。

对于 EDF 算法来说，它只考虑任务的时限，这样所有任务都会有十分相似的截止期错失率，缺乏差分服务能力，随着应用过程中负载的增加，系统性能就会急剧降低。而 HP 算法综合各种参数，优先执行系统中最重要的任务，因而不同类别任务的差分截止期错失率明显不同，且随着重要类别的降低而显著降低。无论是在正常运行还是在过载的情况下，HP 算法都可以保证较高价值的任务具有较低的截止期错失率，具备一定的差分服务能力，在很大程度上优化了 micro-K 操作系统的实时调度性能。

3. 最高价值完成率 AHVR

在系统中设定任务的价值数值与任务的实际价值成反比。在实验中任务的实际价值采用 Vi′(100-Vi)来表示，以便使最高价值完成率 AHVR(achieved highest value rate)更加直观[57]。AHVR 的计算公式如下式 7.2：

$$AHVR = 100 \times \frac{\sum_{j \in TD} Vj}{\sum_{i=1}^{100} Vi} \tag{7.2}$$

TD 代表满足差分截止期的所有任务的任务序号集合。

图 7.4 给出了 EDF 算法、HVF(价值最高最优先)算法和 HP 算法在不同负载下 AHVR 的变化曲线图，其中横坐标是系统负载，纵坐标是调度算法的最高价值完成率 AHVR。

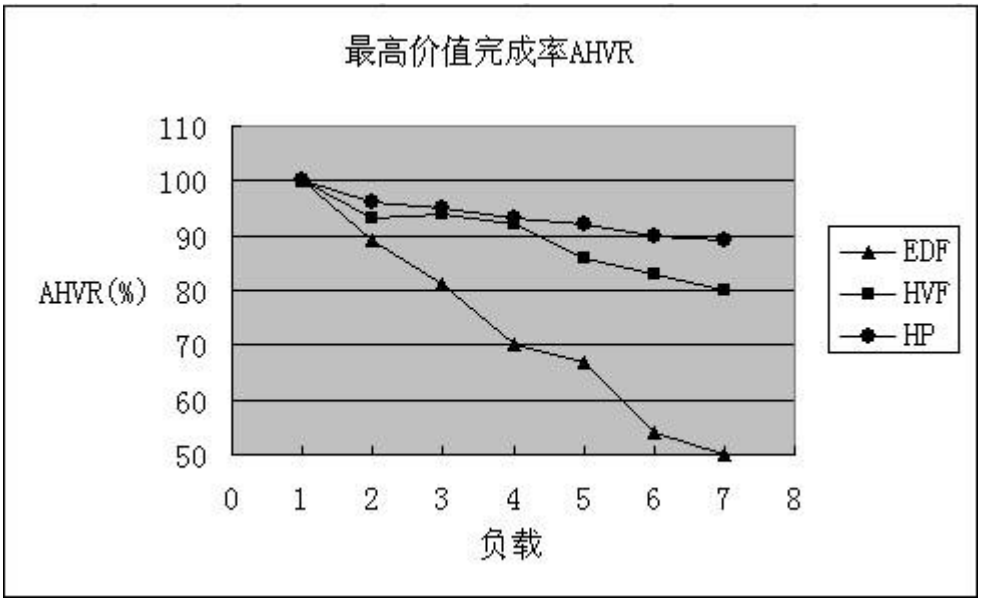

图 7.4 最高价值完成率 AHVR

由图 7.4 可以看出，EDF 算法在系统负载较少时表现最优，但是当系统运行过程中不断增加负载后，EDF 的性能就会急剧下降。在负载较低的情形下，

HVF 算法表现最差。HVF 算法选择价值最高的任务优先调度运行，忽略了任务的空闲时间和剩余执行时间等影响因素，所以比较容易错过任务的截止期。但是随着负载的增多，其调度性能要好过 EDF，不过 HVF 算法总体执行性能还是偏低的。HP 算法在负载较少时，它的调度性能与 EDF 算法非常接近。随着负载的不断增加，其调度性能要明显好于 EDF 算法，并且在总体执行性能方面也比 HVF 算法有明显的优越性，并能在系统过载的情况下实现平缓的降级。因此综合考虑各方面因素，HP 算法总体执行性能令人满意。

7.5 本章小结

本章阐述了测试的软硬件平台，并将 micro-K 内核移植到 C8051F120 平台上进行了相关的测试工作。在测试中首先针对 micro-K 内核任务调度算法的性能测试进行测试，并与 μC/OS-Ⅱ内核进行相关比较。其次，从差分截止期错失率 DMDR 和最高价值完成率 AHVR 两方面对 HP 算法进行了实验与分析。实验结果表明，HP 算法相对于 RMS、EDF 算法、HVF 算法和 LSF 算法都有比较明显的性能改进。

第八章 总结与展望

8.1 总结

本课题的任务是设计及实现一种嵌入式操作系统。本文研究的主要内容和创新之处在于：

1. 设计了一种面向 8 位微处理器的嵌入式实时操作系统 micro-K，并在目标硬件开发平台下运行成功。micro-K 嵌入式操作系统简单、小巧，占用空间小，执行效率高，可裁剪，可移植。系统经裁减后的实际代码只有 3KB，拥有实时嵌入式操作系统所需要的各种功能。
2. 改进了优先级位图调度算法，增加了其支持的优先级数。内核采用了深度优先级位图算法作为调度的基本算法。该算法将 μC/OS-Ⅱ中支持的优先级数增加一倍，扩充到 micro-K 所需的 128 个优先级。算法在增加任务数的同时，基本保留了原有的数据结构，仅需增加有限次的运算就能达到目的，实验证明算法性能未受到影响。
3. 在 micro-K 内核中引入动态调度算法，提出了一种新的任务优先级分配策略—HP 调度算法。该算法配合静态优先级管理的同时也提供了对动态优先级的支持，综合了两种优先级调度算法的优点，提高了内核的灵活性及实用性，减少了系统的开销。
4. 针对目标硬件，添加了一些外围设备驱动功能。针对硬件平台 C8051F120 单片机的特点，为 micro-K 操作系统设计了一个外设驱动程序管理层 PDML，对设备进行统一管理，并利用 C8051F120 丰富的片上资源来提高系统的资源利用率。

8.2 展望

本课题的研究也存在许多不足之处，还需要进一步研究和改进，具体来说可以从以下方面入手：

1. 继续完善 micro-K 的调度框架以及 HP 混合优先级调度算法，对内核服务进行扩充，并对其进行标准商业化的封装。
2. 通常使用的实时调度算法，在超载情况下性能会发生退化。HP 算法虽然改善了超载下的性能，但是增加了一些额外的系统开销，还需要进一步研究如何处理开销这个复杂的问题。

3. 在实时系统中还要考虑与调度相关的其它关键性的技术，如支持容错、具有资源回收的调度等等。
4. 建立功能更完备的设备驱动程序管理层独立于内核工作。
5. 嵌入式操作系统的开发设计还需要考虑硬件资源，要根据不同的硬件环境进行内核代码移植，根据硬件的特点改进 micro-K 的内核，减少内核的使用量，如对 micro-K 进行中断高效处理等等。

参考文献

[1] Ganssle Jack G. The Art of Programming Embedded Systems. San Diego: Academic Press, 1992: 20-22.

[2] Laplante Phillip A. Real-Time Systems Design and Analysis, An Engineer's Handbook. Piscataway, New Jersey: IEEE Computer Society Press, 1992: 30-33.

[3] 张坤. 一种实时嵌入式多任务微内核的分析与改进[D]. 西安: 西安电子科技大学, 2006.

[4] 高伟华, 杨子军. 嵌入式操作系统的研究现状及发展趋势[J]. 黑龙江电力, 2002, 24(5): 383-386.

[5] Wayne Wolf著, 孙玉芳等译. 嵌入式计算系统设计原理[M]. 北京: 机械工业出版社, 2002. 2.

[6] 罗蕾. 嵌入式实时操作系统及应用[M]. 北京: 北京航空航天大学出版社, 2005. 1.

[7] Jejurikar R., Pereira C., Gupta R.. Leakage aware dynamic voltage scaling for real-time embedded systems[C]. Proceedings of 41st Conference on Design Automation, 2004: 275-280.

[8] Jupyung Lee, Kyu-Ho Park. Delayed locking technique for improving the real-time performance of embedded Linux by prediction of timer interrupt[C]. Proceedings of the 11th IEEE Real Time and Embedded Technology and Applications Symposium, 2005: 487-496.

[9] Baynes K, Collins,C, Fiterman E, Brinda Ganesh, Kohout P, Smit C, Zhang T, Jacob B. The performance and energy consumption of embedded real time operating ystems[J]. IEEE Transactions on Computers, 2003, 52(11): 1454-1469.

[10] Gang Quan, Linwei Niu, Hu X.S, Mochocki B. Fixed priority scheduling for reducing overall energy on variable voltage processors[C]. Proceedings of 25th IEEE International Real-Time Systems Symposium, 2004: 309-318.

[11] Qing Li. 嵌入式系统的实时概念[M]. 北京: 北京航空航天大学出版社, 2004. 1.

[12] 李仕涌, 谭南林. 多任务操作系统在嵌入式系统开发中的应用[J]. 北方交通大学学报. 2002, 26(4): 79-82.

[13] 赵良, 叶俊民, 罗景等. 操作系统体系结构风格的比较研究[J]. 计算机应用研究, 2005, 22(5): 50-52.

[14] 许海燕, 付炎. 嵌入式系统技术与应用[M]. 北京: 机械工业出版社, 2002. 4.

[15] 王田苗. 嵌入式系统设计与实例开发—基于ARM微处理器与μCOS-Ⅱ实时操作系统

(第2版)[M]. 北京:清华大学出版社, 2003. 10.

[16]潘琢金. C8051F120/1/2/3/4/5/6/7C8051F130/l/2/3 系列混合信号 ISP FLASH 微控制器数据手册[R/OL]. Rev 1. 3, 2004. 12.

[17]Labrosse Jean. μC/OS-II The Real-Time Kernel. Second Edition. CMP Books. 2003.

[18]Jean J. Labrosse著, 邵贝贝等译. 嵌入式实时操作系统μCOS-Ⅱ(第2版)[M]. 北京:北京航空航天大学出版社, 2003. 5.

[19]任哲. 嵌入式实时操作系统 μC/OS-II 原理及应用[M]. 北京:北京航空航天大学出版社, 2005. 8.

[20]Michael M.Swift, Brian N.Bershad, Henry M.Levy. Improving the reliability of commodity operating systems[J]. ACM Transactions on Computer Systems, 2005, 23(1): 77-110.

[21]刘军祥, 王永吉, MatthewCartmell. 一种改进的RM可调度判断算法[J]. 软件学报, 2005, 16(1):89-100.

[22]王永吉, 陈秋萍. 单调速率及其扩展算法的可调度性判定[J]. 软件学报, 2004, 15(6): 800-811.

[23]刘怀, 胡继峰. 实时系统的多任务调度[J]. 计算机工程, 2002, 28(3):43-44.

[24]冉全. 单片机中基于多线程机制的实时多任务研究[J]. 微型机与应用, 2003, 22(8): 13-15.

[25]万柳, 郭玉东. 嵌入式 RTOS 中就绪任务查找算法和优先级反转的解决方案[J]. 计算机应用, 2003, 23(6):49-51.

[26]沈胜庆. 嵌入式操作系统的内核研究[J]. 微计算机信息, 2006, 22(5):72-74.

[27]王海燕. μC/OS-Ⅱ任务调度的改进与实现[J]. 现代电子技术, 2006, 29(14):41-43.

[28]熊玉梅, 陈一民. μC/OS-Ⅱ任务调度算法的改进[J]. 计算机应用与软件, 2008, 25(6):84-86.

[29]林游, 韩志科. 在 μC/OS-Ⅱ上实现优先级天花板[J]. 单片机与嵌入式系统应用, 2005, 5(4):77-79.

[30]刘智臣, 孟益民. 嵌入式操作系统 μC/OS-II 中优先级反转问题及其解决方案[J]. 科学技术与工程, 2005, 5(1):23-27.

[31]郭长国, 周明辉, 王怀民. 一个基于多线程的优先级继承协议锁的算法研究[J]. 计算机研究与发展, 2002, 39(12):1550-1555.

[32]LIU C L, LAYLAND J M. Scheduling algorithms for multiprogramming in a hard real-time ACM, 1973, 20(1):46-61.

[33]R Abbott, H Garcia-Molina. Scheduling real-time transactions [J]. ACMSIGMOD Record, 1988, 17(1):71-81.

[34]DERTOUZOS M L, MOK A K. Multiprocessor on-line scheduling of hard-real-time

tasks Engineering, 1989, 15(12):1497-1506.

[35]SPRUNT B, SHA L, LEHOCZKY J. Aperiodic task scheduling for hard-real-time systems[J]. Real Time Systems, 1989, 1(1):27-60.

[36]G Buttazzo, MSpuri, F Sensini. Value vs deadline scheduling in over-load conditions [A]. Proceedings of the 16thIEEE Real-Time Systems Symposium [C]. Los Alamitos, California:IEEE Computer Society, 1995:90-99.

[37]Lu C, Stankovic J, Tao Getal. Design and evaluation of a feed-back control EDF scheduling algorithm[C]. Proceedings of the 20th IEEE Real-Time Systems Symposium. Washington, DC, USA:IEEE, 1999:56-67.

[38]金宏,王宏安,王强等.改进的最小空闲时间优先调度算法[J].软件学报,2004, 15(8):1117-1123.

[39]Burns A, Prasad D, Bondavalli A, Giandomenico FD, Ramamritham K, Stankovic J, Strigini L. The meaning and role of value in scheduling flexible real-time systems. Journal of Systems Architecture, 2000, 46(4):305-325.

[40]GButtazzo, MSpuri, Fsensini, Valuevs. deadline scheduling inover-load conditions. Proceedings of the 16th IEEE Real-Time Systems Symposium[C]. LosAlamitos, California. IEEE Computer Society. 1995:90-99.

[41]Katcher D I. Engineering and analysis of real-time operating systems[D]. Ph.D.Dissertation, Dept.of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA.1994.

[42]Kettler K A, Katcher D I, Strosnider J K. A modeling methodology for real-time/multimedia operating systems. Proceedings of the Real-Time Technolog and Applications Symposium[C]. Chicago, Illinois, IEEE Computer Society Press. 1995.

[43]Jin H, Wang HA, Wang Q, Dai GZ. An integrated design method of task priority. Journal of Software, 2003, 14(3):376-382.

[44]盛中平,王晓辉,李冰玉.多点多重 Lagrange 型插值公式[J].东北师大学报(自然科学版),2007,39(2):136-137.

[45]于渊.自己动手写操作系统[M].北京:电子工业出版社,2005.8.

[46]魏洪兴,周亦敏.嵌入式系统设计与实例开发实验教材 I—基于 ARM 微处理器与 μC/OS-Ⅱ实时操作系统[M].北京:清华大学出版社,2005.9.

[47]鲍可进.C8051F 单片机原理及应用[M].北京:中国电力出版社,2005.9.

[48]李刚,林凌.与 8051 兼容的高性能高速单片机 C8051Fxxx[M].北京:北京航空航天大学出版社,2002.5.

[49]Labrosse J.Jean 著,袁勤勇等译.嵌入式系统构件(第 2 版)[M].北京:机械工业出版社,2002.2.

[50]刘歌群.单片机系统多串行口设计技术研究[J].航空精密制造技术,2006,42(2):60-62.

[51]蔡宁果,何晓琼. 用 8 位单片机实现串口-以太网转换器[J]. 电子技术应用,2002,28(2):14-16.

[52]胡立坤,王庆超.基于 UART 的可靠通信与性能分析[J].计算机工程,2006,32(10):15-17.

[53]阳富民,柯滔,涂刚.基于 JTAG 技术的嵌入式交叉调试软件[J].计算机工程与设计,2005,26(10):2817-2819.

[54]李新龙. 嵌入式实时操作系统 μC/OS-Ⅱ在自动杂交仪上的应用[J]. 中国西部科技,2009,8(4):38-40.

[55]张谦,竹利平.μC/OS-Ⅱ实时嵌入式操作系统的实时性分析与测试[J].计算机工程与设计,2005,26(9):2422-2424.

[56]邢群科,郝红卫,温天江.两种经典实时调度算法的研究与实现[J]. 计算机工程与设计,2006,27(1):118-119.

[57]宋宝燕,李巍,李志强等. 一种基于优先级的数据流查询实时调度策略[J]. 计算机工程,2007,33(9):107-108.

致谢

在山东理工大学大学计算机学院的三年中，对于我来说，既是生活在幸福的大家庭里，又是置身于知识的海洋中，是一次难得的学习机会。借此论文完成之际，我由衷地感谢所有给予我关怀、指导、帮助、支持和鼓励的所有老师、同学和亲人。

首先，我向我的导师张景元教授表示衷心感谢和敬意，感谢张老师为我提供了研究环境和方向，为我的论文工作奠定了基础。在三年的学习中，张老师言传身教，在专业学习上给予我精心的指导和极大的帮助，引导着我在科学研究的道路不断前进。张老师科学的思维方式、活跃的学术思想、高深的学术造诣、对事业的执著追求和诲人不倦的师长风范都使我受益终生。本论文是在张老师的悉心指导和关怀下才得以顺利完成的，从论文的选题、收集资料到论文的撰写都凝聚着导师的心血。在此我谨向敬爱的导师致以最诚挚的谢意。

求学期间，计算机学院的领导和老师们给予了我莫大的帮助和关怀，使我的知识面得到拓展，认识程度得到加深。研究生处的领导和老师们给予了我工作和生活上的关照，在此对他们表示由衷的感谢。

深深的感谢我的父母！你们一直在默默地关心着我。你们给予了我战胜一切艰难挫折的勇气和力量，你们的鼓励永远是我前进的动力。永远感谢您们！

衷心的感谢各位支持我，关心我的人，论文的成功有你们的功劳。谨以此文献给我的恩师，我的家人和所有帮助过我的人！

感谢在百忙之中为本文审稿的各位老师，谢谢！

在学期间公开发表论文及科研情况

文章名称	发表刊物(出版社)	刊发时间	刊物级别	第几作者
基于 C8051F 的智能清洁车控制器设计	山东理工大学学报	2010 年第 1 期	普通期刊	3 排 1