浙江大学

硕士学位论文

SmartOSEK OS3.0的设计与实现

姓名：郁利吉

申请学位级别：硕士

专业：计算机科学与技术

指导教师：吴朝晖;赵民德

20070501

摘要

实时操作系统（RTOS）是一种特殊的能满足实时系统实时性需求的操作系统，实时操作系统是各种嵌入式实时系统的功能基础和开发平台。SmartOSEK OS是浙江大学ESE工程中心开发的符合OSEK/VDX标准的嵌入式实时操作系统，该系统已经推出到2.0版本，是一个高效可靠的实时内核。但是由于起初设计上的缺陷和OSEK/VDX标准本身的局限性，SmartOSEK OS 2.0依然存在一些问题和不足，这些问题和不足影响了该系统的可用性。

SmartOSEK OS 3.0在继承SmartOSEK OS 2.0优点的同时对其进行改良和扩展。与SmartOSEK OS 2.0不同，其整体采用层次化结构，分为硬件抽象层、实时内核层以及系统服务层三层。硬件抽象层实现硬件相关操作，向上层提供平台无关的接口，便于系统移植。实时内核层分为超核以及内核组件。SmartOSEK OS 2.0相当于SmartOSEK OS 3.0的超核部分。超核针对SmartOSEK OS 2.0的不足进行扩展，实现最基本的内核功能，其中扩展的任务队列为实现复杂的任务调度机制和同步机制奠定了基础；动态内存管理实现了动态的内存分配操作；动态对象管理实现了动态创建和释放内核对象的功能。内核组件基于超核实现不同标准规定的内核功能，并提供标准化的应用程序编程接口。AUTOSAR内核组件实现了AUTOSAR内核功能，最主要的是实现了调度表从而引入了基于时间的调度机制。POSIX内核组件提供了符合POSIX标准的实时线程模型以及基本的线程同步机制，如：互斥量、信号量以及条件变量。系统服务层提供各种扩展的系统服务：ECU状态管理器管理ECU的运行、停止和睡眠状态并负责在这些状态间的切换；看门狗管理器提供基于硬件看门狗的应用程序监控机制；AUTOSAR通信服务提供基于消息的统一通信平台；TCP/IP协议栈提供基于TCP/IP的通信能力；图形用户界面提供基于LCD的图形交互功能。

SmartOSEK OS 3.0对现有的SmartOSEK OS 2.0进行改进和扩展，并在一定程度上进行革新，是一个高效、可靠、功能丰富的嵌入式实时操作系统。

关键词 嵌入式实时操作系统、OSEK/VDX、AUTOSAR、POSIX

Abstract

Real-time operating system (RTOS) is a special kind of real-time system which can meet the needs of real-time systems. RTOS is the function base and the development platform of embedded real-time systems. SmartOSEK OS is an OSEK/VDX complaint embedded real-time operating system developed by ESE Center, Zhejiang University. It is a highly efficient and reliable real-time kernel. However, because of the initial design flaws and OSEK/VDX standards' own limitations, SmartOSEK OS 2.0 still has some problems and shortcomings. These problems and shortcomings are affecting the usability of the system.

SmartOSEK OS 3.0 improves and expands SmartOSEK OS 2.0. SmartOSEK OS 3.0 uses a level based structure. It is divided into three layers: hardware abstraction layer, real-time kernel layer and the service layer. Hardware Abstraction Layer encapsulates hardware-related operation and improves the portability of the system. Real-time kernel layer is divided into super core and kernel components. The super core is based on SmartOSEK OS 2.0 and expands some functions including Task Queue, Dynamic Memory Management and Dynamic Object Management. Task Queue makes complex task scheduling possible. Dynamic Object Management realized the dynamic creation and release of the kernel object. Kernel components support different standards and provide kernel APIs. AUTOSAR Component provides AUTOSAR kernel function, including the schedule table used to bring time-triggered scheduling mechanism. POSIX component provides the real-time POSIX thread model and the basic thread synchronization mechanisms, such as: mutexes, semaphores and condition variables. System services layer provides various services to expand the system. ECU state manager manages ECU states. Watchdog manager offers supervision base on hardware watchdog. AUTOSAR communications service provides a uniform mechanism to exchange messages. TCP/IP Stack provides TCP/IP communication services and GUI provides services based on LCD.

SmartOSEK OS 3.0 improves and extends the existing SmartOSEK OS 2.0. SmartOSEK OS 3.0 is an efficient, reliable, feature-rich embedded real-time operating system.

Keywords Embedded RTOS, OSEK/VDX, AUTOSAR, POSIX

图目录

表目录

第1章 绪论

实时系统是一种对数据或 IO 处理有严格时间约束的计算机系统。在实时系统中特定的处理如果在一定的时间范围内没有正确完成将会导致系统错误。实时系统一般被分为两类：硬实时系统和软实时系统。硬实时系统保证关键的任务能按时完成。软实时系统要求关键任务有较高优先级以便于其按时完成。

实时操作系统（RTOS）是一种特殊的能满足实时系统实时性需求的操作系统，常被用于对数据处理或 IO 操作有严格时间限制的场合。实时操作系统是嵌入式实时系统软件的基础和开发平台，实时操作系统的使用可以使系统开发人员避开繁琐的硬件操作，而把主要精力集中于具体问题的解决和实际应用程序的设计上。实时操作系统的应用有助于提高系统的模块化程度，增强嵌入式实时系统的可移植性和可维护性。目前实时操作系统已经在各种科学试验设备、医疗设备、工业控制设备、汽车电子设备、家用电器、武器系统中被广泛应用。

本章介绍嵌入式实时操作系统 SmartOSEK OS 3.0 的项目背景、研究基础、研究现状和具体研究目标并简要说明了本文的主要内容、贡献和整体结构。

1.1 项目背景

SmartOSEK OS 是浙江大学 ESE 工程中心开发的符合 OSEK/VDX 标准的超微型、强实时的嵌入式实时操作系统。SmartOSEK OS 是一种适用于嵌入式设备的实时操作系统，作为嵌入式操作系统，其具有可裁剪、可配置、占用资源少的优点；作为实时操作系统，其具有高效、可确定性的特点。

SmartOSEK OS 的 1.0 版本是在参考了一些开源实时操作系统，并将其进行符合 OSEK 标准的改造后推出的，该版本作为原型系统存在着诸多问题。浙江大学 ESE 工程中心在 SmartOSEK OS 1.0 的基础上，通过反复测试和改进，优化了内核实现并扩展了调试功能后推出了 SmartOSEK OS 2.0。SmartOSEK OS 2.0 获得了国家软件著作版权版，还是国内唯一通过国际 OSEK 组织认证的嵌入式实时操作系统，并且已经在一汽、奇瑞和其它一些单位的项目中得到实际的应用。经过实践的考验，SmartOSEK OS 2.0 被认为是一个高效、安全、可靠的多任务实时操作系统。

但是由于 OSEK/VDX 标准仅仅是针对汽车电子领域的简单实时内核的标准，

其标准本身就有相当的局限性，不能满足稍微复杂一点的嵌入式实时系统对实时操作系统的需求，而且 SmartOSEK OS 在起初设计的时候对于整体架构和可扩展性的考虑不足。本课题就是针对这些局限和不足，在广泛调研参考现有各种实时操作系统的基础上，对原有 SmartOSEK OS 2.0 进行改进、扩展，在许多方面对原有系统进行突破和革新，设计并实现性能更优良、功能更完备的嵌入式实时操作系统 SmartOSEK OS 3.0。

1.2 研究基础和现状

1.2.1 实时操作系统比较

目前市场上有许多实时操作系统，其中比较著名和常见的有以下这些：

1) μC/OS

μC/OS[4]是开源的实时操作系统，其最新版本是 μC/OS –II。μC/OS 和 μC/OS -II 是专门为计算机的嵌入式应用设计的实时操作系统，绝大部分代码是用 C 语言编写的。CPU 硬件相关部分是用汇编语言编写的、总量约 200 行的汇编语言部分被压缩到最低限度，为的是便于移植到任何一种其它的 CPU 上。许多移植的范例可以从网站上得到。用户只要有标准的 ANSI 的 C 交叉编译器，有汇编器、连接器等软件工具，就可以将 μC/OS 嵌入到开发的产品中。

μC/OS 具有执行效率高、占用空间小、实时性能优良和可扩展性能等特点，最小内核可编译至 2KB。μC/OS-II 已经移植到了几乎所有知名的 CPU 上。SmartOSEK OS 的最初版本就是参考了 μC/OS-II。

2) eCos

eCos(embedded Configurable operating system)，即嵌入式可配置操作系统，最初起源于美国的 Cygnus Solutions 公司[5]。1999 年 11 月，RedHat 公司以 6.74 亿美元收购了 Cygnus 公司。在此后的几年里，eCos 成为其嵌入式领域的关键产品，得到了迅速的发展，并于 2003 年 5 月正式发布了 eCos2.0。

eCos 弥补了 Linux 在嵌入式应用领域的不足。目前，一个最小配置的 Linux 内核大概有 500KB，需要占用 1.5MB 的内存空间，这还不包括应用程序和其它所需的服务；eCos 可以提供实时嵌入式应用所需的基本运行基件，而只占用几十 KB 或几百 KB 的内存空间。eCos 是一个源码开放的可配置、可移植、无版税、面向深嵌入式应用的实时操作系统。从 eCos 的名称可以看出，它最大的特点在于它是一个配置灵活的系统。eCos 的核心部分是由不同的组件组成的，包括内核、C 语言库和底层运行包等。每个组件以能提供大量的可配置选项，利用 eCos 提供

的配置工具可以很方便地进行配置。通过不同的配置使得 eCos 能够满足不同的嵌入式应用。

eCos 的硬件抽象层向上层提供了硬件平台的抽象，大大增强了系统的可移植性，器硬件抽象层的设计是在广泛分析比较了各种硬件体系架构的异同后得出的，十分具有参考价值。

3) VxWorks

VxWorks[6]可以说是最著名的商用实时操作系统。VxWorks 实时操作系统拥有良好的持续发展能力、高性能的内核以及友好的用户开发环境，在嵌入式实时操作系统领域占据一席之地。它以其良好的可靠性和卓越的实时性被广泛地应用在通信、军事、航空、航天等高精尖技术及实时性要求极高的领域中，如卫星通讯、军事演习、弹道制导、飞机导航等。在美国的 F-16、FA-18 战斗机、B-2 隐形轰炸机和爱国者导弹上，甚至连 1997 年 4 月在火星表面登陆的火星探测器上也使用到了 VxWorks。

4) RTEMS

RTEMS，即实时多处理器系统(Real Time Executive for Multiprocessor Systems)，是一个开源的无版税实时嵌入操作系统 RTOS[7]。它最早用于美国国防系统，早期的名称为实时导弹系统（Real Time Executive for Missile Systems），后来改名为实时军用系统（Real Time Executive for Military Systems），现在由 OAR 公司负责版本的升级与维护。目前无论是航空航天、军工，还是民用领域 RTEMS 都有着极为广泛的应用。

5) Nucleus

Nucleus 是由 Accelerated Technology 推出的多任务实时操作系统[8]，其核心是 Nucleus PLUS 内核，在其基础上扩展为多种接口的版本，包括 C/C++接口、OSEK 接口、μITRON 接口及 POSIX 接口。Nucleus 同时可提供 Web 支持，网络，图形包，文件系统等模块。Nucleus 最大的特点是全部提供源代码，免去用户购买使用许可的费用。此外用户只需通过 DLL 动态连接库便可进行任务级调试。

6) QNX[9]

QNX Neutrino 是一个真正的微内核操作系统。驱动程序，应用程序，协议栈和文件系统都在内核外作为进程运行，从而几乎所有组件都可以失败再重启，而不影响内核或其他组件。QNX 使用消息传递作为进程间通信（IPC）的基本方式，因此其系统可扩展性强，模块化程度高和易于使用。QNX 提供高级图形用户接口、网络支持、Java 环境和驱动程序开发工具。QNX 已经被成功地应用在成千上万的关键系统中，包括： 911 报警系统、高端路由器、医疗仪器、车内电子系统平台、

过程控制系统、航空航天系统，等等。

7) OSE[10]

OSE 是一个紧凑的，可抢占的，提供存储器保护的多处理器实时操作系统。该系统宣称针对分布式容错应用进行优化，达到到最大可靠性、安全性和可用性。OSE 实时操作系统在世界范围内已用于数百万产品，包括分布式通讯系统，如通讯基础设施，移动电话，航空电子，医药业，汽车和工业控制。

OSE RTOS 提供三个版本的内核，分别是 OSE, OSE Epsilon 和 OSEck for DSPs。OSE 面向复杂系统和要求容错与安全的项目，功能丰富，双冗余，代码动态升级和多个通讯协议。OSE Epsilon 是低成本版本，占用内存空间小，速度快，面向深度嵌入式和 SoC 开发的实时操作系统。OSEck 是面向 DSP 的压缩内核。

OSE 的实时内核采用消息机制，提供全抢占调度，支持动态运行时配置，只有 8 个关键 API，易于使用。

8) LynxOS

LynxOS 是一个与 UNIX 兼容的实时操作系统[11]。它符合 POSIX 标准，支持多进程和多线程等特性，是专为要求快速和确定性的复杂实时应用设计的操作系统。LynxOS 内核尤其为硬实时应用作了专门设计。LynxOS 采用模块化设计，具有可伸缩性和配置性。最小应用时，LynxOS 可仅使用内核和应用相连接形成 ROM 映象文件。完全应用时，LynxOS 具有由一系列软件开发工具组成完整的自主调试环境，包含 UNIX 兼容的软件工具，工业标准的网络，图形用户接口和与 UNIX 相似的分级文件系统。

LynxOS 符合 POSIX 的 1003.1 的系统调用接口标准，同时实现 POSIX 的 1003.1b 标准规定的实时扩展和 1003.1c 标准规定的线程扩展。LynxOS 包括 4.4 BSD 系统调用接口和库函数，能为任何一个 UNIX 的平台上的应用提供相当高程度的源代码水平上的兼容。

上述列举的各种实时操作系统各具特色，但概括来说都是为嵌入式实时系统的实现提供了一些基础的服务，主要包括：提供实时多任务的运行环境、提供基本的资源管理、提供基本的系统服务，如，网络通信服务、存储服务等。而这些服务往往都会依据一定的标准体系，所以其功能上实际上是大同小异，主要的不同是在各自的实现方式上。

1.2.2 SmartOSEK OS 及其现状

浙江大学计算机学院 ESE 工程中心针对我国汽车电子领域的自主国产需求，研制一种面向汽车电子控制设备的嵌入式软件平台，并参照欧洲 OSEK 标准,实现

了超微型实时安全的嵌入式实时操作系统 SmartOSEK OS。SmartOSEK OS 的实现提升了我国汽车电子领域嵌入式软件自主开发能力，取得显著的经济效益和社会效益。

SmartOSEK OS 是一套面向汽车电子的、参照欧洲 OSEK 标准的、支持多种国际主流处理器的、超微型实时安全的嵌入式实时操作系统。该操作系统实现可抢占式内核以及多种实时调度机制，适用于实时性要求高的汽车电子产品。SmartOSEK OS 以 Motorola MPC555 和 HCS12 系列芯片为基础运行平台,并可以移植到多种国际主流处理器，满足不同的硬件需求。

SmartOSEK OS 的主要特点是：

1) 符合 OSEK OS 标准 BCC1、ECC1、CCCA、CCCB
2) 静态、微内核、实时嵌入式内核
3) 支持基于优先级的抢占式调度，满足高实时性要求
4) 支持三种调度方式：全抢占式、非抢占式、混合调度方式，提供不同应用调度解决方案
5) 统一封装中断管理，提高中断处理效率以及操作系统的稳定性
6) 提供以事件为主的任务间同步机制，具有高效性和方便性
7) 提供以优先级置顶协议为主的资源管理方法，避免优先级反转
8) 利用定时器实现周期性激活任务、触发事件等功能。
9) 实现基于消息对象的任务、中断间通信机制，具有灵活、方便、高效、多功能等优点
10) 提供灵活的错误处理机制，方便用户实时跟踪调试
11) 满足严格的片内存储器要求，内核小于 9K 字节（在 32 位 PowerPC 平台上）
12) 满足高实时性要求，上下文切换迅速
13) 支持 8 位、16 位和 32 位的处理器
14) 兼容 OSEK OIL 标准，支持用户通过 OIL 对平台进行配置

SmartOSEK OS 1.0 在 2005 年通过了国际 OSEK/VDX 组织的官方认证，是国内唯一通过该组织认证的实时操作系统。

在 1.0 版本的基础上 ESE 工程中心通过反复测试，优化了内核的部分实现并扩展了用户调试机制推出了 2.0 版本并在 2006 年获得了软件著作版权。

SmartOSEK OS 2.0 是一个安全、可靠的多任务实时操作系统，但是其本身依然有一定的局限性和不足。

首先，OSEK/VDX 的实时内核标准是针对汽车电子中的车控系统的特定应用环境制定。OSEK/VDX 操作系统是一个静态的超小型的实时内核，其提供的功能十分简单，并且有许多的限制。比如，OSEK/VDX 操作系统是静态配置的，如任务数量、任务栈大小、任务优先级、资源数量、内存分配等参数都是在系统生成前由用户指定并保证正确性，系统运行过程中不能动态改变。这些限制在特定应用中可以提高系统的效率，但是大多数情况下给用户的使用带来了不便。

其次，OSEK 标准的编程接口对除汽车电子领域外的大多数开发者来说并不熟知，一些习惯使用传统系统接口开发程序的开发人员不太愿意学习和使用这种“陌生”的操作系统。

第三，OSEK/VDX 标准本身也在不断发展。目前许多著名汽车生产厂商联合制定的 AUTOSAR 标准就扩展了 OSEK OS 的功能，并在此基础上提出了全新的汽车电子应用体系。

最后， SmartOSEK OS 只提供了实时内核，对其它一些基本的系统服务功能没有提供支持，大大局限了其使用范围。

综上所述，有必要对 SmartOSEK OS 进行进一步的改进和革新。

1.3 研究目的

本项目的主要目的是对现有的 SmartOSEK OS 2.0 进行改进和扩展，设计实现一个高效、可靠、功能丰富并且能够满足未来嵌入式系统，尤其是能够满足未来汽车电子系统需求的嵌入式实时操作系统 SmartOSEK OS 3.0。其具体目标有：

1) 继承原来 SmartOSEK OS 2.0 高效、可靠的优点。
2) 设计一种易于移植，易于扩展的整体架构。
3) 扩展 SmartOSEK OS 2.0 内核功能使其符合 AUTOSAR 标准对于实时操作系统的要求。
4) 扩展 SmartOSEK OS 2.0 内核功能使其提供符合 POSIX 标准的编程接口。
5) 设计实现与 AUTOSAR 兼容的系统服务和通信服务模块。
6) 扩展 SmartOSEK OS 使其支持 TCP/IP 通信。
7) 扩展 SmartOSEK OS 使其支图形用户界面。

1.4 论文主要内容和贡献

本文在调研总结嵌入式实时操作系统各种基本技术的基础上，提出了对于 SmartOSEK OS 3.0 的整体设计方案，并详细说明了实时内核层以及系统服务层的

设计与实现。其中实时内核层的设计主要说明如何利用层次化内核技术扩展OSEK操作系统使其符合AUTOSAR标准以及POSIX标准;系统服务层的设计主要包括ECU状态管理器、看门狗管理器、基于CAN的AUTOSAR通信服务、对TCP/IP栈的支持以及对图形用户界面的支持。

本论文的主要贡献是:

- 为SmartOSEK OS 3.0设计了一种易于移植、易于扩展的整体架构。
- 提出利用层次化内核技术扩展OSEK操作系统使其支持多种标准化系统编程接口的方法。
- 实现了基于OSEK OS并部分兼容POSIX标准的实时内核。
- 对于AUTOSAR操作系统、系统服务以及通信系统的分析、设计与实现,处于国内领先水平。

1.5 论文结构

第一章为绪论。该章节说明了论文的项目背景,简单说明了相关的研究基础和现状,明确项目目标。

第二章介绍SmartOSEK OS的技术基础。该章节是在调研目前市场上常见实时操作系统的基础上,对实时操作系统基本技术的总结。

第三章为整体设计,说明了SmartOSEK OS 3.0的整体设计方案以及各个模块的主要功能。

第四章为实时内核层设计与实现,说明了如何利用层次化内核技术扩展OSEK OS使其支持AUTOSAR OS标准和POSIX标准。

第五章为系统服务层设计与实现,说明了ECU状态管理器、看门狗管理器以及AUTOSAR通信服务的设计。

第六章为总结与展望,总结了本文的工作,并对未来工作进行展望。

第2章 SmartOSEK OS 3.0的技术基础

实时操作系统与分时操作系统不同，它更关注于系统的响应时间而不是平均性能。因此，实时操作系统一种普遍采用的设计原则是：采用各种算法和策略，始终保证系统行为的可预测性。可预测性是指能够确切断定一个任务的完成时间的能力。也就是说即使在最坏的情况下，任务的运行时间也是可以确定。这就要求实时操作系统的资源调配策略都能为争夺资源(包括CPU、内存等)的多个实时任务合理地分配资源，使每个实时任务的实时性要求都能得到满足。因此，实时操作系统的设计与实现与通用操作系统有所不同。本章从实时操作系统标准化、实时内核技术、嵌入式通信技术以及嵌入式图形用户界面这几方面介绍了SmartOSEK OS 3.0设计与实现的技术基础。

2.1 实时操作系统的标准化

市场上各种不同类型的嵌入式实时操作系统可谓是纷繁复杂，不同的系统往往提供各种不同的功能，这使得用户常常觉得难以选择，而且这种差异性大大增加了嵌入式实时系统在不同操作系统间移植的难度。为了解决这一问题，一些有影响的组织往往会制定一些针对实时操作系统的标准，这些标准规定了实时操作系统的一些基本行为和功能，而且规定了实时操作系统所提供的应用程序接口标准。这样，不同的实时操作系统只要遵循同一标准，那么在某一实时操作系统上开发的应用就能方便得被移植到另一种实时操作系统上。为了增强在不同系统上开发的应用软件的可移植性，提高代码重用性，实时操作系统一般会提供对某些被广泛使用的标准的兼容。目前市场上已经有一部分得到广泛应用和支持的系统接口标准，本项目主要参照的标准有：OSEK/ VDX 标准、AUTOSAR 标准和POSIX 标准。

表 2-1 部分常见实时操作系统支持的接口标准

RTOS	μC/OS	ECOS	NUCLEUS	OSEK OS	RTEMS
编程接口	专用	专用 μITRON POSIX	专用 POSIX OSEK	OSEK/VDX	专用 μITRON POSIX

2.1.1 OSEK/VDX标准

为了满足日益庞大复杂的汽车电子控制软件的开发需要，实现应用软件的可

移植性和不同厂商的控制模块间的可兼容性，1993年德国汽车工业界联合推出了汽车电子的开放式系统及接口－OSEK(Open Systems and the Corresponding Interfaces For Automotive Electronics)规范，旨在为汽车上的分布的控制单元提供一个开放结构的工业标准。1994年，法国汽车行业的一个类似项目VDX（Vehicle Distributed eXecutive）加入OSEK，形成了OSEK/VDX。OSEK/VDX规范从实时操作系统、软件接口、通讯和网络管理等方面对汽车的电子控制软件开发平台作了较为全面的定义与规定。[2]

OSEK/VDX标准是从欧洲发展而来的针对汽车工业的嵌入式实时操作系统标准。目前，OSEK/VDX标准在欧美汽车行业已经被广泛采用，一些公司推出了符合OSEK/VDX标准的操作系统并得到了OSEK委员会的认证，如OSEK Works、OSEK-OS、OSEK-Turbo等。

OSEK/VDX标准包括以下四部分：OSEK操作系统规范（OSEK Operating System，*OSEK OS*）[26]，*OSEK*通讯规范（OSEK Communication，*OSEK COM*）[27]，OSEK网络管理规范（OSEK Network Management，OSEK NM）[28]以及OSEK实现语言（OSEK Implementation Language，OSEK OIL）[29][30]。采用符合OSEK/VDX标准的嵌入式实时操作系统可以提高产品代码的复用率、降低开发成本、缩短产品开发周期。

SmartOSEK OS 3.0就是以符合OSEK/VDX标准的嵌入式实时操作系统SmartOSEK OS1.0/2.0基础上设计开发的。

2.1.2 AUTOSAR标准

AUTOSAR（汽车开放系统标准，AUTomotive Open System ARchitecture）是由一些汽车制造厂商、供应商和开发商联合制定的开放的汽车电子软件体系结构标准[3]。该标准的目的是促进汽车电子的技术创新，提高汽车电子产品的性能、可靠性、安全性以及环境友好性。其宗旨是使得全球汽车生产商“在标准上合作，在实现上竞争”。该标准旨在提供一种管理汽车电子/电气产品复杂性的关键技术并为汽车电子领域未来的技术革新和进步做好准备。根据AUTOSAR的设计理念，不同的厂商开发系统的一个部分，然后应用者根据具体的需求使用标准化的方法和工具整合各个部分构成整体系统。

AUTOSAR标准的部分内容是基于OSEK标准的，AUTOSAR标准涵盖了汽车电子软件的多个领域，包括：硬件驱动、资源管理、操作系统、实用标准程序库、系统服务、通信服务以及系统开发方法和整合技术等各个方面。

AUTOSAR 标准的主要目标是[3]：

1) 管理由于功能范围扩大而导致的嵌入式电子设备日益增长的复杂性
2) 增强产品更新升级的灵活性
3) 增强解决方案在产品线内和产品线间的可扩展性
4) 增强嵌入式电子系统的质量和可靠性
5) 在开发的早期发现错误

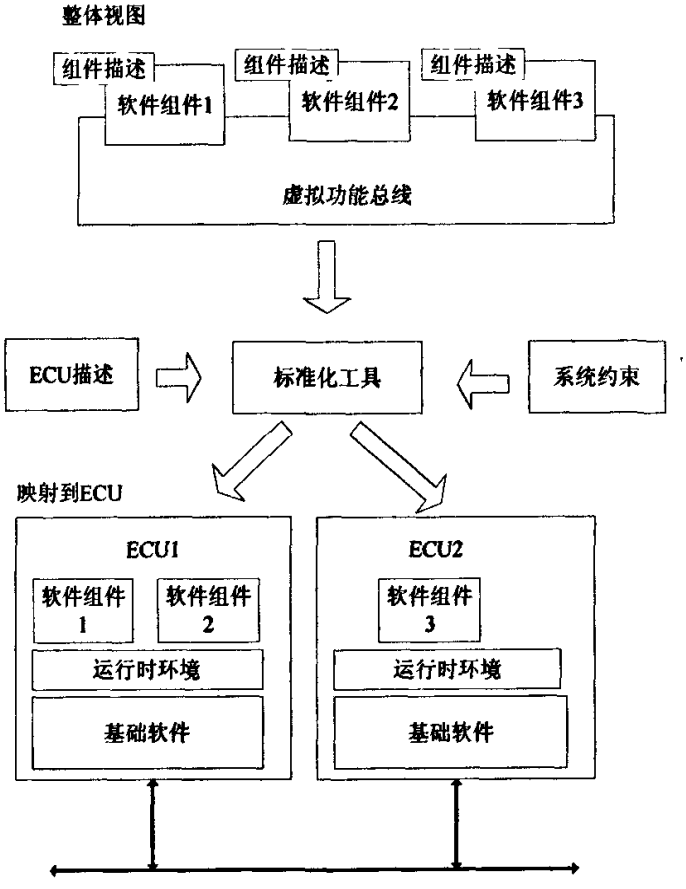

图 2-1 AUTOSAR 系统方案示意图

如图，AUTOSAR 系统主要由如下模块组成：

- AUTOSAR 软件组件（AUTOSAR SW-C）

AUTOSAR 软件组件是对在 AUTOSAR 体系中运行的应用程序的封装。AUTOSAR 软件组件通过特定的 AUTOSAR 接口互相联系。一个 AUTOSAR 软件组件封装了特定的应用。AUTOSAR 软件组件的实现是和底层基础软件以及硬件无关的。

为了描述软件组件的接口以便于各部分的整合，AUTOSAR 定义了统一的格式来对软件组件进行描述，这就是软件组件描述。软件组件描述一般描述该软件组件提供的数据和操作、需求的数据和操作、需求的资源以及其它一些和实现相关的特定的信息。

- 虚拟功能总线（Virtual Functional Bus，VFB）

虚拟功能总线是在一个抽象层面上对 AUTOSAR 提供的所有通信机制的概括。AUTOSAR 软件组件的硬件无关性就是靠虚拟功能总线实现的。虚拟功能总线提供了一个虚拟的硬件环境，当对应于一个特定系统的 AUTOSAR 软件组件被开发出来后，虚拟功能总线允许在开发的早期对这些组件进行虚拟的整合。

- 系统约束和 ECU 描述（System Constraint and ECU Description）

为了把 AUTOSAR 软件组件整合到 ECU 网络中去，AUTOSAR 提供了标准化的格式来描述单个 ECU 的资源和配置。这类描述和软件组件的描述是相互独立的。

- ECU 映射（Mapping on ECU）

对于如何整合不同元素来构建整个系统，AUTOSAR 定义了一系列的方法和工具。其中特别包括了 ECU 上的运行时环境和基础软件的配置和生成。

- 运行时环境（Runtime Environment，RTE）

从 AUTOSAR 软件组件的角度来看，运行时环境是对虚拟功能总线在特定 ECU 上的实现。从系统设计层面看运行时环境是 ECU 内或 ECU 间信息交换的中心。运行时环境提供了对于通信的抽象，屏蔽了实际通信的细节，AUTOSAR 软件组件不必知道它们之间的通信是通过不同的通信信道（CAN、LIN 等）还是仅仅局限在单个 ECU 内部。

- 基础软件（Basic Software）

基础软件提供在单个 ECU 上运行 AUTOSAR 系统的基本功能。它是一个标准软件层，提供了运行 AUTOSAR 软件组件基本服务，包括标准组件和 ECU 相关组件。标准组件包括系统服务、通信、操作系统、微控制器抽象。ECU 相关组件包括 ECU 抽象和复杂驱动程序。

AUTOSAR 标准是各大汽车生产厂商总结多年开发经验提出的最新标准，2006 年开始陆续发布一些标准的说明。该标准目前还处于初步指定和推广阶段。从内容上看，AUTOSAR 标准是对现有汽车电子软件技术的总结、继承和发展；从参与该标准的厂商实力上来判断，该标准很可能成为未来汽车电子发展软件发展的重要方向。

2.1.3 POSIX 标准

POSIX 标准则是从传统 UNIX 发展而来的一种开放系统标准。POSIX 在传统的通用计算机领域可以说是最广为使用的标准了，由于 POSIX 标准有着极强的传统优势，如果实时操作系统提供 POSIX 标准兼容的应用编程接口，则应用 POSIX 接口风格编写的应用软件可以较为方便地移植到这些实时操作系统之间。所以 POSIX 标准应用范围广泛，支持或部分支持 POSIX 标准的嵌入式操作系统也比较多。

POSIX 是 Portable Operating System Interface（可移植操作系统界面）的缩写，而 X 则表明其 API 是继承了 Portable Operating System for UNIX(TM)的缩写。其正式称呼为 IEEE 1003，而国际标准名称为 ISO/IEC 9945。

POSIX 其实是一系列的标准，其中最重要的是 POSIX 1003.1[22][23]，它定义了一套 C 语言应用编程接口（API），此外和实时操作系统有关的标准还有 POSIX 1003.1b[24]和 POSIX 1003.13[25]。

POSIX 1003.1b 提供了 POSIX 1003.1 标准的实时扩展。

POSIX 把操作系统的实时性定义为“操作系统在一定的反应时间内提供特定层次服务的能力”。该扩展定义了用于支持实时系统的可选系统接口集合，主要包括：信号量、进程内存锁、内存映射文件和共享内存、基于优先级的调度、异步信号的实时扩展、定时器、处理器内部通信、同步 I/O 和异步 I/O。

POSIX 1003.13 定义了基于 POSIX 1003.1 的实时嵌入式应用环境。它定义了四种实时嵌入式运行环境：最小实时系统、实时控制器系统、专用实时系统、多用途实时系统。该标准为每种运行环境指明必须支持的 POSIX 系统调用。

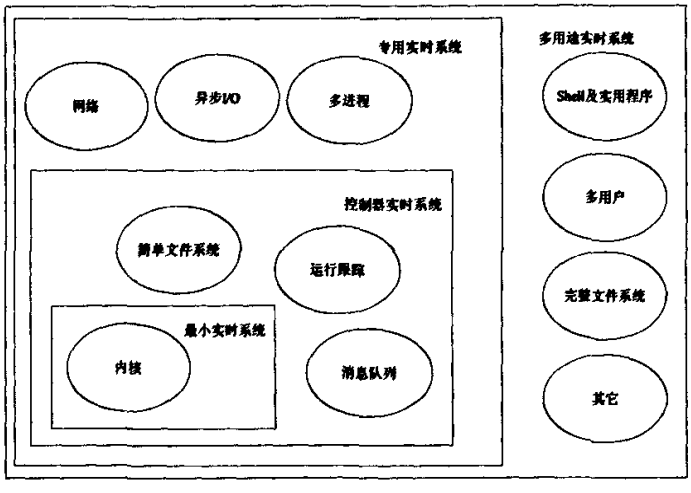

图 2-2 POSIX1003.13 定义的四种实时系统运行环境

2.2 实时内核相关技术

在嵌入式领域，广为流传着这样一种定义：

实时内核＝实时操作系统[15]。

虽然很多人不同意这种观点，但实际上在大多数嵌入式实时操作系统的应用场合，开发人员只需要一些最基本的操作系统功能，如：任务管理、同步、任务通信等。由于用户在使用实时操作系统时往往会把一些额外的功能如：TCP/IP、文件系统裁剪掉而只保留基本的内核部分。而且许多操作系统提供商，如μC/OS、NUCLEUS[8]，往往会把实时内核作为独立产品推出，所以久而久之在人们的观念中就认为实时内核就是实时操作系统了。

2.2.1 实时任务管理

实时操作系统中的一个任务是指一个特定的应用程序模块，实时操作系统以任务为单位分配处理器资源。系统设计者把系统划分成不同的任务，每个任务完成一个特定的功能，各个任务间通过同步与通信机制相互协调共同完成系统的实际功能。实时内核的一个核心的功能就是提供一个实时的多任务运行环境，因此任务管理是实时内核最基本也是最重要的一项功能。CPU在许多任务之间进行转和调度，一个CPU轮番服务于一系列任务中的一个。多任务运行提高了程序间的并发性，使得CPU的利用率得到最大发挥。多任务模型的最大好处是使得开发人员可以将复杂的程序层次化、模块化。开发人员只需要关注任务的设计，具体的任务调度以及切换由操作系统完成。

任务调度是实时内核的主要职责之一，简单说任务调度就是决定在某一时刻该运行哪个任务。任务调度算法实质就是为了满足特定需求，而对处理器资源进行时间上的分配的方法。由于对于实时系统的一个重要要求就是任务必须在特定的时间限制内完成，所以任务调度算法成为实时内核的一项重要技术。

实时系统中常采用两种调度策略：时间驱动策略和事件驱动策略。

时间驱动策略就是系统设计者把任务的启动时间事先设计好，实时内核只需要在特定的时间启动任务就可以了。这种方式的优点是任务的实时性要求可以得到保证而且任务调度的实现简单；缺点是系统的灵活性差，不能适应运行条件的迅速变化。

事件驱动就是由事件来激活任务。事件有事先安排好的，如编写软件通过系统调用激活任务；也有随机发生的，如中断等。实时内核负责根据特定的调度算法分配处理器资源。

最基本的实时调度方法是基于优先级的调度算法。在这种算法中，每个任务都有一个优先级，当决定运行哪个任务是总是选取当前就绪任务中优先级最高的任务运行。

基于优先级的调度算法分为不可抢占调度方式和可抢占调度方式。使用不可抢占调度方式时，当高优先级任务就绪时并不立即占用调度器资源，而是要等正在运行的任务主动放弃处理器资源时才运行；而使用可抢占调度方式是一旦高优先级任务就绪就立即抢占处理器资源运行。不可抢占调度的优点是运行的任务不用担心被抢占，所以不需要额外操作来保护共享数据，但其相应时间较长，而且是不确定的；可抢占内核响应迅速，但是由于任务随时有可能被抢占所以有较高的互斥需求。而实际系统常采用两种方式的混合：每个任务指定自己是否可抢占，如果可抢占则采用可抢占调度，如果不可抢占则采用不可抢占的调度方式。

当两个或两个以上任务具有相同优先级时常使用先来先服务或时间片轮转算法。采用先来先服务方法时，多个优先级相同的就绪任务中，时间上先进入就绪状态的任务先运行，当其运行完毕后再运行同优先级的后进入就绪状态的任务。当使用时间片轮转调度时，一个任务运行事先确定的一段时间，然后切换给另一个相同优先级的就绪任务。

使用基于优先级的调度方式时，用户将高优先级分配给关键任务，使得关键任务及时运行，从而满足实时约束。但是如何分配任务优先级使得整个系统能够满足实时约束需要通过特定的可调度性分析方法来确定。基于优先级的调度算法分为静态调度算法和动态调度算法。采用静态调度算法的系统中，任务的优先级在运行前被确定；采用动态调度算法时任务的优先级在系统运行时确定。

最常用的静态调度算法是单调速率算法[16]，单调速率算法把高优先级赋予执行频率高的任务。如果一个系统可以用固定优先级的调度方式调度，那它一定可以用单调速率方法调度。理论证明，应用单调速率算法的系统最高 CPU 利用率为 69.3%。最早截止期优先算法既可以用于静态调度又可以用于动态调度，它的最高 CPU 利用率为 100%。其变种有最小富裕度优先和最大紧急度优先等。

目前为止在大多数使用的实时操作系统中被广泛采用的还是静态调度算法。

2.2.2 任务间的同步和通信

实时任务需要利用同步机制来处理互斥资源的访问和任务相关性问题。同步机制的基本原理是让实时任务在某种情况下放弃执行进入等待状态，当某种条件满足时再使得该任务进入就绪态然后在得到处理器资源后继续运行。不同的实时

操作系统提供各种不同的同步机制，常用的同步机制有：互斥量、信号量、条件变量、事件、异步信号等。常见实时操作系统提供的同步机制如下所示。

表 2-2 部分常见实时操作系统提供的同步机制

实时操作系统	ECOS	μITRON	NUCLEUS	OSEK OS	RTEMS
同步机制	互斥量 信号量 事件 条件变量	互斥量 信号量 事件 异步信号	信号量 事件 异步信号	事件	互斥量 条件变量 信号量 事件 异步信号

运行在同一处理器上的任务通过使用任务间通信机制进行数据交换，实时操作系统中常用的任务通信机制是邮箱和消息队列。其它一些传统的任务间通信机制，如：管道、FIFO、套接字等在实时操作系统中应用较少。常见实时操作系统提供的任务间通信机制如下所示。

表 2-3 部分常见实时操作系统提供的任务间通信机制

实时操作系统	μCOS	μiTRON	NUCLEUS	OSEK COM	RTEMS
任务间通信机制	邮箱 消息队列	邮箱 消息队列 消息缓冲	邮箱 消息队列 管道	邮箱 消息队列	消息队列

2.2.3 内存管理

设计实时操作系统时对于内存管理方式的选择主要面临两个问题：是否支持动态内存分配以及是否支持虚拟存储器管理。

静态分配内存的系统中，所有程序需要用到的内存在系统创建时就已经被分配好，系统运行时不可以动态的申请和释放内存。静态分配内存的系统省去了动态分配内存的开销，所以时间性能比较好。但是这种方式限制了程序设计的灵活性，而且内存的利用率也相对较低。比如，在静态分配内存的实时操作系统中系统对象控制块（如任务控制块等）的数量必须在编译时确定，而不能在系统运行时动态创建和删除。这样即使某任务在整个系统中仅仅运行了很小一段时间，其任务控制块也将在整个系统运行期间占用内存。而且如何为每个任务预留合适的内存空间大小也是一个设计上的难点。比如，任务运行需要栈空间，如果为一个任务预留过小的栈空间则其运行过程中可能发生栈溢出而导致系统崩溃；相反如果为任务分配太多栈空间则无疑是对内存空间的一种浪费。

动态内存分配增加了程序设计的灵活性，提高了内存的利用率，但是增加了系统分配回收内存的开销。所以实时操作系统往往支持有限的动态内存分配机制。常用的动态内存管理方式有固定大小的内存池和可变大小的内存池[1]。

固定大小内存池是指内存的分配是以固定大小的内存块为单位的。比如内存块大小为 4 字节的话，即使只需要 5 字节内存，系统也要分配两个内存块即 8 字节内存。这种管理方式实现简单，而且其时间开销是可确定的，但是其缺点是有一定的内存浪费，比如上述例子中就有 3 字节的内存被浪费了。

可变大小内存池是指操作系统按用户实际申请的大小分配内存。这种管理方式的内存利用率较高，但是实现比较复杂。

由于较早的嵌入式微控制器没有内存管理单元（MMU），而且虚拟存储管理会带来一定的不确定性，所以传统的实时操作系统都不支持虚拟存储管理。但是现在越来越多现代处理器内建了内存管理单元，不支持 MMU 的实时操作系统似乎是浪费了硬件资源，而且利用虚拟存储技术可以提供内存空间的保护，所以是否提供虚拟内存管理成为了一种两难的选择。一些实时操作系统（如，VxWorks）处理这一问题的方法是把虚拟存储管理作为可选的软件包提供给用户，由用户自己决定是否使用虚拟存储管理。

2.2.4 时间管理

实时操作系统一般都提供良好的时间管理。

计算机系统中由硬件时钟周期性产生时钟中断，操作系统负责利用这些时钟中断时限较为复杂的时间相关功能。不同操作系统提供的时间相关功能各不相同，但一般会包括如下内容：

1) 维护日期和时间

日期和时间维护的实现并不困难，一般通过记录时钟中断的次数来记录时间。在 UNIX 系统中，时间被转化为通用标准定时（UTC）1970 年 1 月 1 日中午 12 点开始的时钟滴答数。

实时操作系统似乎更注重诸如运行时间以及运行时限这样的相对时间的控制而不是类似日期和时间这样的绝对时间的管理，所以不同的实时操作系统为了节省记录时间所用的内存或为了解约维护时间的代价会而简化日期和时间的维护，比如减少精度等，有的甚至不提供日期和时间维护功能。

2) 维护任务时间片

对于使用时间片轮转调度方式的实时操作系统，内核需要记录任务运行的时

间，如果该任务运行的时间超出一定的时间片则有可能需要利用调度器把处理器资源分配给其它任务

3) 提供定时器功能

由于实时操作系统的重要特性是保证任务的实时约束，所以定时器功能对实时操作系统十分重要。任务可以请求操作系统在一定时间后执行某特定操作，一次进行时间相关的操作。操作系统一般会使用软件方法利用单一硬件时钟模仿多个虚拟时钟，以此支持多个定时操作。

4) 对 CPU 进行记帐

CPU 记帐就是记录任务占用 CPU 的时间。实时操作系统跟分时系统比更关心系统反应时间而不太关心 CPU 的公平分配和利用率，但是有时为了进行性能分析和调度性分析还是会需要 CPU 记帐功能。

2.3 控制器局域网通信协议

CAN（Controller Area Network 控制器局域网）是一种有效支持分布式控制或实时控制的串行通信网络。为了达到设计透明度以及实现柔韧性 CAN 被细分为以下不同的层次[19]：

- CAN 对象层 the object layer
- CAN 传输层 the transfer layer
- 物理层 the physical layer

对象层和传输层包括所有由 ISO/OSI 模型定义的数据链路层的服务和功能。对象层的功能是报文滤波以及状态和报文的处理。对象层的作用范围包括：查找被发送的报文；确定由实际要使用的传输层接收哪一个报文；为应用层相关硬件提供接口。

传输层是 CAN 协议的核心，它把接收到的报文提供给对象层以及接收来自对象层的报文。传输层的作用主要是传送规则，也就是控制帧结构、执行仲裁、错误检测、出错标定、故障界定。总线上什么时候开始发送新报文以及什么时候开始接收报文均在传输层里确定。

物理层定义实际信号的传输方法，其作用是在不同节点之间根据所有的电气属性进行位信息的实际传输。同一网络内物理层对于所有的节点必须是相同的。但是 CAN 协议没有具体规定物理层以便允许用户根据各自的需求对发送媒体和信号电平进行优化。其通信介质可以是双绞线、同轴电缆和光导纤维等。

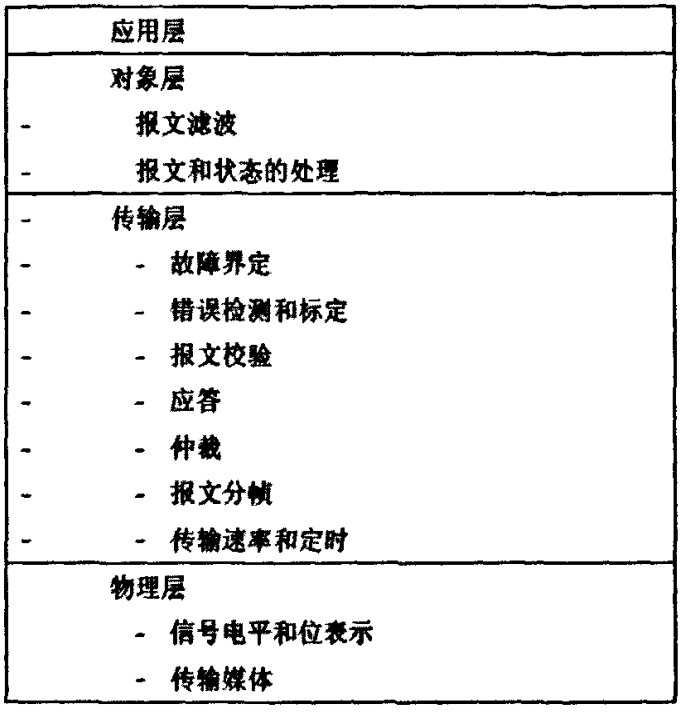

图 2-3 CAN 节点的层次结构[19]

CAN 协议的一个最大特点是废除了传统的站地址编码，而代之以对通信数据块进行编码。采用这种方法的优点可使网络内的节点个数在理论上不受限制，数据块的标识码可由 11 位或 29 位二进制数组成，因此可以定义 211 或 229 个不同的数据块，这种按数据块编码的方式，还可使不同的节点同时接收到相同的数据，这一点在分布式控制系统中非常有用。

CAN 中数据段长度最多为 8 个字节，可满足通常工业领域中控制命令、工作状态及测试数据的一般要求。同时，8 个字节不会占用总线时间过长，从而保证了通信的实时性。CAN 协议采用 CRC 检验并可提供相应的错误处理功能，保证了数据通信的可靠性。

另外，CAN 总线采用了多主竞争式总线结构，具有多主站运行和分散仲裁的串行总线以及广播通信的特点。CAN 总线空闲时，任意节点都可主动地向网络上其它节点发送信息而不分主次，因此可在各节点之间实现自由通信。

CAN 总线协议已被国际标准化组织认证，技术比较成熟，控制的芯片已经商品化，性价比高，特别适用于分布式测控系统之间的数通讯。CAN 卓越的特性、极高的可靠性和独特的设计，特别适合工业过程监控设备的互连，因此，越来越受到工业界的重视，并已公认为最有前途的现场总线之一。

2.4 嵌入式 TCP/IP 协议栈

目前，随着 Internet 网络技术的成熟和普及，许多嵌入式设备实现网络化。所以，许多嵌入式实时操作系统中引入了 TCP/IP 协议栈，以支持嵌入式设备接入网络。但是由于 TCP/IP 协议比较复杂，而目前嵌入式系统中大量应用低速处理器，受内存和速度限制，所以目前嵌入式实时系统常提供简化的 TCP/IP 实现，并尽可能做到代码精简、降低存储开销。TCP/IP 参考开放系统互连（OSI）模型，采用一种简化的四层模型，分别为：应用层、传输层、网络层、链路层。[17]

- 应用层

该层包含所有的高层协议，如：虚拟终端协议（TELNET）、文件传输协议(FTP)、电子邮件协议(SMTP)、域名系统服务(DNS)及超文本传输协议(HTTP)等。

- 传输层

传输层的功能是源端和目标端主机上的对等实体可以进行会话。传输层使得网络程序可以通过明确定义的通道及某些特性获取数据，如定义网络连接的端口号等。这里定义了两个端到端的协议：传输控制协议（TCP）和用户数据报协议（UDP）。

由于 UDP 协议是一种不可靠的传输协议，目前多用于语音和视频传输等流媒体应用中，考虑到嵌入式设备在工业控制等专用领域中在网络传输的是重要的数据和命令，其传输的可行性可以通过 TCP 协议来保证，因此一些嵌入式 TCP/IP 的实现不提供 UDP 协议。

- 网络层

网络层是整个 TCP/IP 的关键部分，它的功能是使得信息可以发送到相邻的 TCP/IP 网络上的任一主机上。该层定义了正式的分组格式和协议，即 IP 协议。IP 协议就是该层中传送数据的机制，它为不同网络的主机之间发送数据报的操作序列提供无连接服务。

ICMP 协议用于报告在数据报过程中的错误，IP 协议是无连接的，他无法将报文和错误信息传到最初的主机，ICMP 将状态信息和错误信息发送到发报文的主机。因此嵌入式系统中常提供 ICMP 的实现。

此外，由于嵌入式设备中大多使用以太网网卡进行互连，为了建立网络间的互连，还提供了 ARP 地址解析协议，实现从 IP 地址到数据链路物理地址的映射。

- 主机至网络层

TCP/IP 体系中并没有描述网络层下有什么，只是指出主机必须使用某种协议与网络相连接。

由于商业嵌入式 TCP/IP 协议栈比较昂贵，所以一些源代码公开的免费协议栈被广泛使用。目前较为著名的两个免费协议栈是：lwIP(Light weight TCP / IP Stack)和 uIP0+9(TCP / IP stack for micro)[18]。lwIP 是一个简化的 TCP/IP 实现，包括 IP、ICMP、UDP 和 TCP 协议的实现，它是模块化的，可以容易地扩展成传统的协议。lwIP 支持多个本地网络接口，并且配置方便，所以适用于许多嵌入式设备。uIP 只包含 TCP/IP 协议栈中的最小功能子集。它只能处理一个网络接口，它仅实现了 IP、ICMP 和 TCP，没有实现 UDP。

uIP 实现了 TCP/IP 协议栈所需的最小功能，包括 IP 协议、ICMP 协议和 TCP 协议的实现，但是没有实现 UDP 协议，且仅能处理单个网络接口。

从高层看，uIP 协议栈就像一个黑盒，在接受端它从底层网络接口接到数据包，TCP 协议对数据包进行排序盒重组，然后把数据提交给上层应用。在发送端，uIP 协议栈从应用程序收集数据并通过底层网络接口发送数据。数据到达 uIP 协议栈是异步的，由于收到的包可能是乱序的，所以 uIP 协议栈需要收集重组数据包；此外 uIP 协议栈需要控制网络的数据流量，也就是不是所有数据都被立即发送，所以 uIP 协议栈需要为数据包提供数据队列。

完整的 TCP/IP 协议栈包括许多协议，包括从底层的 ARP 协议到高层的 SMTP 协议。uIP 主要实现 TCP 以及 IP 协议，所有上层协议被看成是应用程序。底层协议由网络驱动控制，被认为是网络设备。

由于在嵌入式系统中，内存容量十分有限，TCP/IP 协议栈只能使用有限的内存空间，所以需要特殊的内存管理方式。

uIP 不使用动态内存分配，而是使用一个全局的包缓存来存放数据包并有一个固定大小的表格来维护连接状态。全局包缓存能够放下一个最大的数据包。当数据包从网络到达时，驱动程序把它放入全局缓存并调用 uIP 协议栈。如果包中含有数据，uIP 通知上层应用。上层应用程序决定时立即处理数据还是把数据存放到其它的位置，在上层应用处理完数据之前，包缓存的数据不会被新数据覆盖。这时新来的数据必须被保持在队列中，该队列由硬件或驱动实现。许多以太网控制器有片内缓存可以暂时存放一定量的数据。如果缓冲满了，新来的数据被丢弃。

传输数据时，全局缓存被用来存放 TCP/IP 包头。发送数据时，应用程序把数据长度以及数据指针发送给 uIP 协议栈。协议栈把包头写入全局缓冲，一旦包头生成，协议栈就利用网络设备发送数据包。协议栈不为重发保存数据，如果数据需要重传，则数据会被重新生成。

应用程序接口定义了应用程序使用 TCP/IP 的方式，最常用的 TCP/IP 接口是

BSD套接字。该套接字使用了停等方式，所以需要底层复杂的多任务操作系统的支持。uIP 协议栈提供一种事件驱动的接口。数据到达、连接请求、轮询都是事件，这种事件机制能够和实时内核很好的配合，而且事件机制提供了短的响应时间。

使用基于事件的接口时，上层应用作为特定事件的处理程序被 uIP 协议栈调用。当数据到达，或当数据被发送到另一个通信端，或当新连接被建立或当数据被重新传输完毕时 TCP/IP 协议栈调用上层应用程序。由于当特定事件发生时，上层应用可以被立即调用所以可以获得低的延迟时间。uIP 协议栈不保存要发送的数据，数据重传依靠应用程序完成。

uIP 的 IP 层利用一个独立缓存实现了分段重组。当一个分段进入时，该分段被复制到该缓存的正确位置，然后用一个位图记录分段是否已经被接收。当所有分段被重组，完整的 IP 包被提交给传输层，如果在一定时间内数据包没有重组成功，整个包被丢弃。目前只支持单个包的重组。

uIP 的 ICMP 协议被用来报告网络错误或查询主机参数，但是其最常用的是利用回声（echo）机制实现“ping”程序。对于该消息的应答就是简单的交换源地址与目标地址，并把包头改为 ICMP 的回声应答类型。其它的消息目前不支持。

uIP 协议栈中的传输协议是事件驱动的。TCP 层检查输入的数据包，如果包含给应用程序的数据，则 TCP 层把数据提交给应用程序。如果进入的数据包是对发送数据的应答则更新连接状态并通知应用程序，这样应用程序可以发送新数据。

uIP 的 TCP 层为了计算合适的重发时限不停的为每个活跃的连接计算当前的来回时间（RTT）。TCP 层用周期性定时器计算 RTT。每次周期定时器到期时，TCP 层为每个连接增加一个计数器值，当确认被收到时，当前的计数器值作为一个 RTT 样本，然后使用 Jacobson 的算法计算 TCP 的 RTT[17]。

数据重发由周期定时器触发。每次周期性定时器过期时，重发定时器被减一。当定时器为 0 时进行一次数据重发。TCP 层依靠应用程序实现数据重发，当 TCP 层要求数据重发时它调用应用程序，并设置一个标志位表明需要重发数据，应用程序重新生成需要重发的数据。实际的重发机制由应用程序实现，协议栈的任务决定是否需要重发数据。

2.5 嵌入式图形用户界面

在桌面计算领域，基于图形界面的操作系统由于操作方便被广泛采用。在嵌

入式实时系统中有时也需要使用图形用户界面进行人机交互，随着液晶屏幕成本的降低，各种带有图形界面的嵌入式系统和设备取得了广泛的应用，因此图形用户界面成为了许多嵌入式实时系统提供的重要功能。

由于嵌入式实时系统的硬件架构千差万别，同时嵌入式实时系统的硬件资源十分有限，因此高可移植性、可扩展性和简单灵活性成了设计嵌入式 GUI 的主要设计目标。目前市场上出很多嵌入式 GUI 产品，可谓花样繁多，其中应用比较广泛的嵌入式图形用户界面系统有：

- MicroWindows[20]

MicroWindows 采用了分层的设计思想。最低层是面向输入输出设备的驱动程序；中间层实现了图形引擎，并进行窗口管理；最高层提供兼容于 X Windows 和 Windows CE 的 API。MicroWindows 提供了相对完善的图形功能，但是普遍认为其代码质量不高，并存在一些低效算法。

- OpenGUI

OpenGUI 也分为三层。底层是出汇编语言编写的快速图形引擎；中间层提供了图形绘制功能；第三层用 C++编写，提供了完整的 GUI 对象集合。但是 OpenGUI 目前只支持 X86 平台。

- Qt/Embedded

Qt/Embedded 是著名的 Qt 库开发商发布的面向嵌入式系统的 Qt 版本。应为 Qt 是 KDE 等项目的 GUI 支持库，所以 Qt/Embedded 的使用方便将多种 X Windows 程序移植到嵌入式平台上。但是 Qt/Embedded 缺乏对硬件加速的支持，而且 Qt/Embedded 结构过于复杂，难以定制、扩充和裁剪。

- MiniGUI[21]

MiniGUI 是一种面向嵌入式系统或者实时系统的图形用户界面支持系统。它主要运行于 Linux 控制台，实际可以运行在任何一种具有 POSIX 线程支持的 POSIX 兼容系统上。MiniGUI 同时也是国内最早出现的几个自由软件项目之一，MiniGUI 的一大特点是有良好的中文支持。

μC/GUI 是 Micrium 公司开发的通用的嵌入式用户图形界面软件。它给任何使用图形 LCD 的应用程序提供独立于处理器和 LCD 控制器之外的有效的图形用户接口。可以应用于单一任务环境，也可以应用于多任务环境中。μC /GUI 能够应用于任何 LCD 控制器和 CPU 的任何尺寸的物理显示或者模拟显示中。

μC/GUI 的体系结构如图所示，其函数库为用户程序提供 GUI 接口，函数支持文本、数值、二维图形、输入设备以及各种窗口对象。输入设备可以是键盘、鼠标或触摸屏；二维图形包括图片、直线、多边形、圆、椭圆、圆弧等；窗口对

象包括按钮、编辑框、进度条、复选框等。

μC /GUI 支持的主要输出设备是 LCD，它支持多屏幕和多层次的液晶屏。支持触摸屏和鼠标作为点输入设备，支持多种类型的键盘。

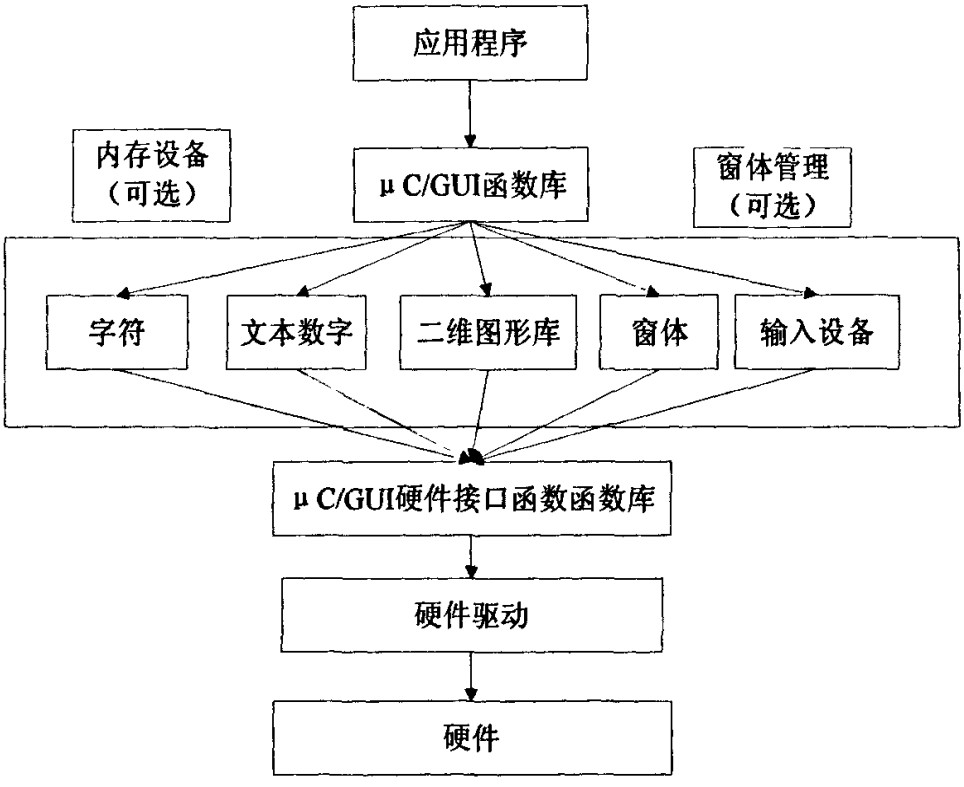

图 2-4 μC/GUI 的体系结构

μC/GUI 提供系统调用用于显示字符。μC/GUI 支持多种不同字体，而且字体可以在显示屏的任何位置显示。μC/GUI 支持两种控制字符 LF 及 CR。LF 表示换行，即当前字符位置为下一个行的开始。CR 表示回车，即当前字符位置为当前行的开头。其中 LF 支持使得可以通过一个函数调用显示多行文字。μC/GUI 同时支持利用系统调用把文字位置转移到特定坐标。μC/GUI 支持各种二进制、十进制以及十六进制数据的输出。

μC /GUI 提供了完整的二维图形库，该图形库基于快速高效的算法。μC /GUI 支持两种绘图模式：普通模式和异或模式。普通模式中显示内容会被图形覆盖，而异或模式中显示内容会在重绘时被翻转。其支持的二维图形库包括二维图形包括图片、直线、多边形、圆、椭圆、圆弧。

μC /GUI 提供两种显示图片文件的方式。一种方式是实现把图片文件转换成 C 代码数据结构，然后把该 C 文件加入工程，然后根据该 C 数据结构显示图片。

另一种方式是直接显示位于存储器中的图片文件。目前支持的图形文件有：BMP、JPEG 和 GIF。对于需要多次显示的图片，可以把它转换为 C 文件，这样的显示方式比较快速。直接显示建议只对于需要经常变换的图片采用。

μC /GUI 利用内存设备技术防止显示重叠图像时的闪烁。不使用内存设备时，图形的绘制是实时体现在显示屏上的，这样在绘制多个图形时由于屏幕不停更新从而导致闪烁。使用内存设备时，所有的绘制操作先在内存中进行，当所有绘制操作完毕后再一次性在屏幕上显示。其本质区别时，不使用内存设备时，复杂图形的绘制过程是在屏幕上实时显示出来的，用户其实时看到了整个图形的绘制过程，因此会有图形闪烁的感觉；而使用内存设备时，复杂图形的绘制是一次性显示出来的，这样就不会有闪烁感觉。内存设备的使用会占用一定量的内存，其实质是用空间换时间，所以是否使用内存设备是可选的，用户根据各自应用的不同需求确定。

当使用窗体管理器时所有显示在屏幕上的内容都包含在窗体内。窗体可以时任意大小，而且可以在屏幕上显示多个窗体，可以显示局部窗体，可以层叠窗体。窗体管理器提供创建、移动、缩放等窗体操作。同时提供底层支持用于通知用程序改变窗体。窗体管理器是一个可选的组件。窗体管理通过回调机制提供事件驱动的系统。回调机制提供从图形界面到应用程序的控制流。当事件发生时，GUI 向特定窗体发送一个消息，该窗体的回调函数根据接收到的消息的类型进行特定的操作。

μC /GUI 提供丰富的控件，控件根据自身属性重绘自身，实际的重绘操作是由一个特定的子程序实现的，用户程序不停调用该子程序，该子程序根据 GUI 控件的属性重绘控件。

用户可以用对话框包含多个控件，对话框被用来向用户获得输入。窗体管理器会记住被最后选择的窗体，从外部获得的输入将被转化为消息发送到该窗体。大多数传递给对话框的消息被对话框处理，其它一些消息被发送给特定的回调函数。

2.6 本章小结

本章是在作者调研和分析目前市场上出现的多种嵌入式实时操作系统后对嵌入式实时操作系统技术的概括和总结。

为了提高嵌入式实时操作系统的可移植性和通用性，嵌入式实时操作系统需要合一定的标准。POSIX 标准是最常用的标准，其提供了对实时系统的支持。

OSEK/VDX是汽车电子领域被广泛使用的实时分布式系统的标准。AUTOSAR是最新的汽车电子软件的标准体系结构，为未来汽车电子软件的发展指明了方向。

实时内核是实时操作系统最核心的模块，其主要功能是提供高效的实时多任务模型，一般采用基于优先级的抢占式调度算法以保证实时性。实时内核同时需要提供任务间的同步和通信功能。除此之外，实时内核还提供时间管理和内存管理。在内存管理方面，实时内核主要提供高效简单的动态内存分配功能，有时也提供对虚拟内存技术的支持。

嵌入式通信最常用的是TCP/IP协议和CAN通信协议，其中TCP/IP是最常用的传统通信方式，和传统计算机系统的兼容性好。uIP是一个著名的开源嵌入式TCP/IP协议栈，其包含TCP/IP协议栈中的最小功能子集。uIP只能处理一个网络接口，它仅实现了IP、ICMP和TCP，没有实现UDP。CAN协议是适合实时控制领域的线程总线通信协议，越来越受到工业界的重视，并已公认为最有前途的现场总线之一。

随着液晶屏成本降低，嵌入式图形用户界面已经被越来越多的嵌入式设备使用，许多嵌入式实时操作系统都提供对于嵌入式图形界面的支持。μC/GUI是Micrium公司开发的通用的嵌入式用户图形界面软件。它给任何使用图形LCD的应用程序提供独立于处理器和LCD控制器之外的有效的图形用户接口。

本章中提到的技术是SmartOSEK OS 3.0设计与实现的基础和依据，在以后的章节中都会有所体现。

第3章 SmartOSEK OS 3.0的整体设计

SmartOSEK OS 2.0提供有限的系统功能，规模较小，因此其采用了整体式结构以提高系统运行效率。SmartOSEK OS 3.0在SmartOSEK OS 2.0基础上扩展了许多功能模块，由于系统复杂性的提高，整体式结构不利于扩展和移植的缺点逐渐显露出来，因此SmartOSEK OS 3.0采用层次化结构以提高系统的可移植性和可扩展性。本章详细说明了SmartOSEK OS 3.0层次式结构的设计。

3.1 SmartOSEK OS 3.0的层次化结构

实时操作系统与通用操作系统往往采用类似的系统结构，其中最常用的是整体式、层次式和微内核式[13]。整体式和层次式结构注重程序的执行效率，适合规模不大的嵌入式实时操作系统；微内核结构注重安全性、稳定性及可扩展性，适合较为复杂的实时操作系统。

许多比较大型的实时操作系统采用微内核结构，微内核操作系统的设计理念是仅仅在操作系统内核中实现最关键的核心功能，如基本进程管理、和基本通信机制，而大多数操作系统功能由用户进程实现[13][14]。为了获取某项服务，用户进程将请求发给一个服务进程，服务进程随后完成此操作并将结果返回。此时，核心的全部工作是处理客户与服务器间的通信。操作系统被分割成许多小部分，每一部分更小更易于管理。微内核操作系统的优点是系统提供较好的保护，比较稳定，而且易于扩展。但微内核操作系统的一个明显的缺点是性能问题，由于消息传递的效率没有直接调用子程序高，所以以消息传递为基础的微内核操作系统的运行性能会有所降低。由于SmartOSEK OS 3.0是面向嵌入式领域的嵌入式实时操作系统，该系统十分重视系统的运行性能所以不采用微内核结构。但是在逻辑上SmartOSEK OS 3.0限定了实时内核的功能，使其只实现必要的内核功能。

SmartOSEK OS 1.0/2.0采用的是整体式结构。SmartOSEK OS 1.0/2.0是一些系统调用函数的集合，每个函数可以任意调用其它函数，函数的互相调用受约束很少。SmartOSEK OS 1.0/2.0是可裁剪的，在设计时采用模块化的思想，对各模块的功能进行封装，明确定义模块间的接口，并把不同的系统功能划分到不同的逻辑模块中。不同的模块分别编译，实际用户只需要选择自己需要的模块，并把这些模块和用户程序链接到一起组成运行的目标程序。这样对用户提供了灵活性

以满足实时系统中常见的苛刻的片上存储的限制。

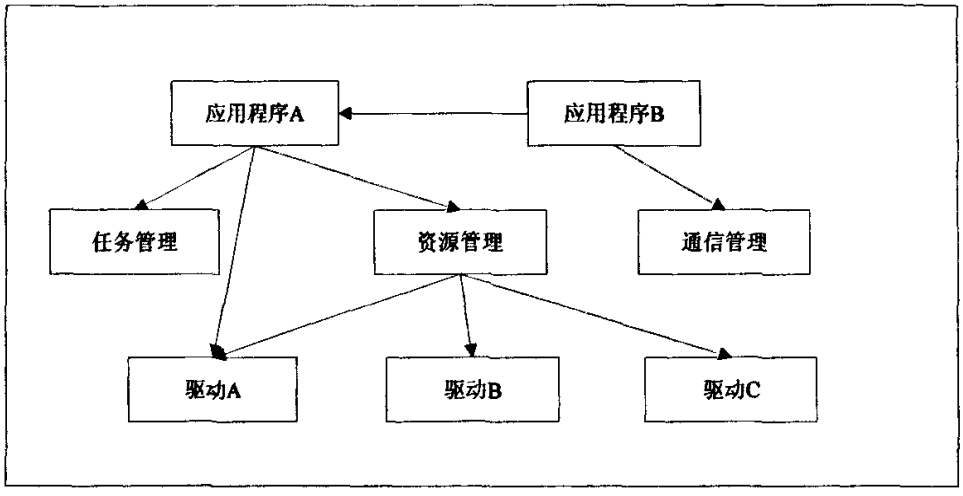

图 3-1 SmartOSEK OS 2.0 的整体式结构

SmartOSEK OS 3.0 采用非严格的层次式结构。所谓非严格的层次结构，其最终程序实际上还是被链接成单个目标，而且过程间调用没有或有很少的限制。但是在系统设计时采用层次化的思想，把特定功能封装在特定层中。SmartOSEK OS 3.0 在逻辑上由许多层组成。其中的每一层是一组数据和操作的抽象，它包括一组数据和对这些数据的操作，它提供的操作可以被上层调用，同时每一层也可以调用下层提供的操作。[13][14] 这种结构的一个优点是系统的调试和验证比较方便。比如，当第一层验证完毕后，开发人员可以认为该层功能完全正确，当调试第二层的时候可以集中于第二层本身。而且高层实现只需要了解底层的接口而不需要了解其实现的细节。总体上 SmartOSEK OS 3.0 分为硬件抽象层、实时内核层以及系统服务层三层。

硬件抽象层实现了操作系统的硬件相关操作，为上层提供了统一的硬件平台无关的接口。当对 SmartOSEK OS 3.0 进行移植的时候只需要重新实现硬件抽象层而不需要修改其它代码。

实时内核层实现了一个实时内核，向上层提供了标准的应用程序编程接口（API）。为了支持多种标准。SmartOSEK OS 3.0 的内核又细分为超核以及内核组件两层。超核以 SmartOSEK OS 2.0 为基础实现了实时内核最基本的功能。内核组件利用超核提供的功能实现不同内核标准定义的机制，提供不同的标准化的应用程序编程接口。SmartOSEK OS 2.0 提供的编程接口有 AUTOSAR OS 编程接口以及 POSIX 编程接口。

外围系统服务层提供可选的系统服务。主要包括：ECU 状态管理器、看门狗管理器、AUTOSAR 通信服务、TCP/IP 协议栈以及图形用户界面。其中 ECU 状态管理器、看门狗管理器、AUTOSAR 通信服务的实现依赖于 AUTOSAR OS 提供的功能。TCP/IP 协议栈以及图形用户界面可以不依赖实时内核。

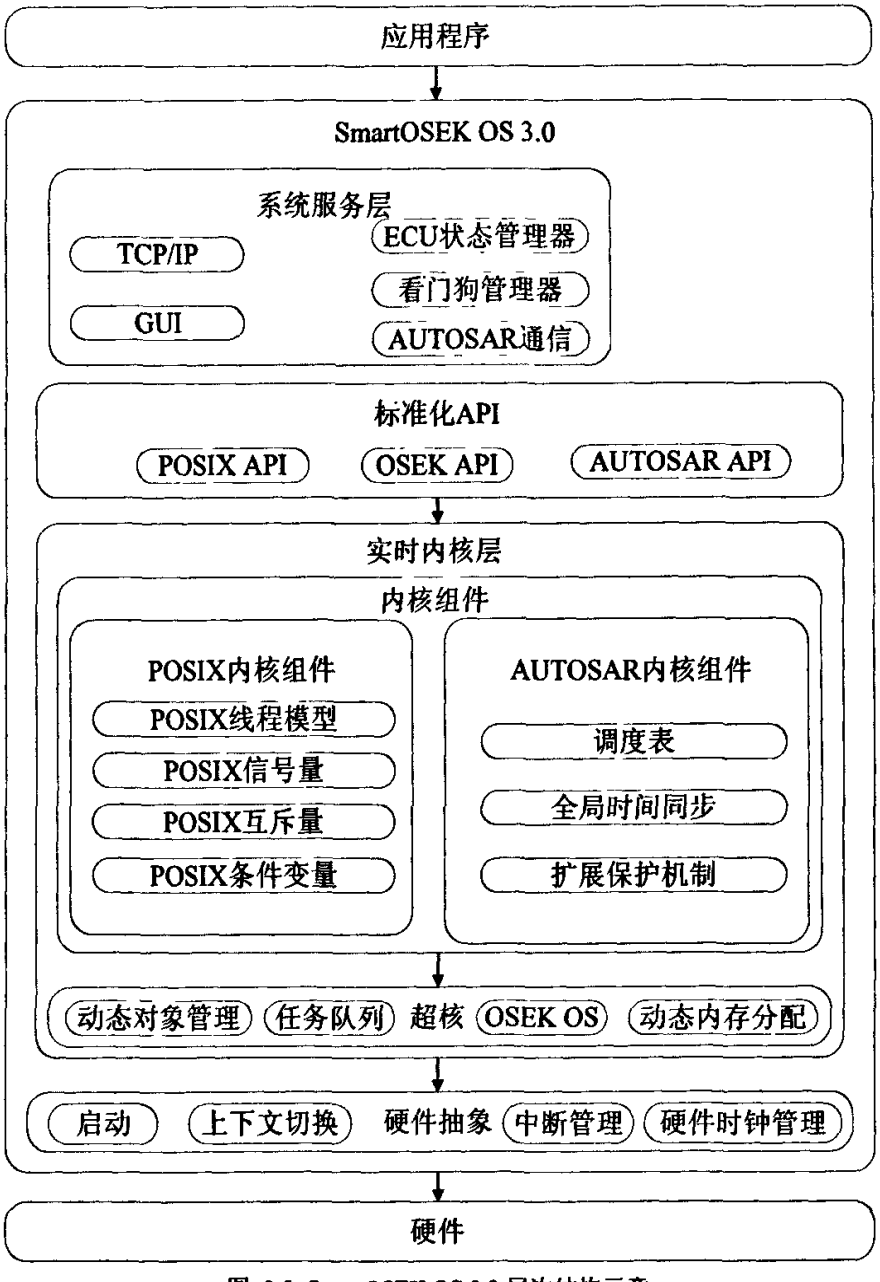

图 3-2 SmartOSEK OS 3.0 层次结构示意

3.2 硬件抽象层

SmartOSEK OS 2.0 在设计和实现时没有明确区分硬件相关代码和硬件无关代码，其在各个硬件平台上的实现有比较明显的差异。当从一个平台移植到另一个平台时需要修改许多分散在各处的代码，因此其可移植性比较差。

SmartOSEK OS 3.0 利用硬件抽象层为实现内核功能必须的硬件相关操作提供了统一的封装。当 SmartOSEK OS 3.0 的从一个平台移植到另一个平台时只需要重写硬件抽象层便可以完成移植。硬件抽象层只负责实现内核必须的平台无关抽象，与内核无关的外围模块驱动的平台无关抽象不包括在硬件抽象层中，这样做是为了保持硬件抽象层的简洁和单一，同时防止过分抽象导致的系统性能下降。

SmartOSEK OS 3.0 的硬件抽象层功能主要包括：系统启动、上下文切换、硬件时钟管理以及中断管理。

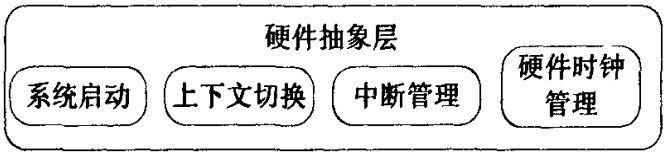

图 3-3 SmartOSEK OS 3.0 硬件抽象层示意

3.2.1 系统启动

系统启动负责操作系统启动前的准备工作。不同平台的启动过程会略有不同，主要的工作是为操作系统的启动准备运行环境，比如：必要的硬件初始化、栈的准备、全局数据的空间分配和初始化等。简单说系统启动从硬件启动开始直至第一个实时任务开始运行为止。不同硬件平台的启动机制各部相同，SmartOSEK OS 3.0 不规定系统启动时的具体步骤，用户根据不同硬件平台选择合适的实现方式。

3.2.2 上下文切换

任务上下文切换是实时内核在进行任务切换时保存和恢复任务运行时的寄存器的值的操作。该部分操作一般由不同体系结构的汇编代码实现。简单的实现方式是把所有寄存器的值都保存和恢复，但是这样做的效率不高，尤其是目前流行的 RISC 架构的处理器有很多寄存器，全部保存和恢复需要一定的时间。一种优化的做法是根据不同寄存器的用途选择性地保存和恢复其值，不过这样做的一个风险是如果该保存的寄存器没有保存则系统会发生错误。因此这里有一个性能

和稳定性之间的矛盾，在SmartOSEK OS 3.0的设计实现中我们的原则是稳定性优先，在保证稳定的前提下追求性能。

SmartOSEK OS 3.0的硬件抽象层规定了以下上下文切换接口：

1) HAL_THREAD_INIT_CONTEXT
2) HAL_THREAD_LOAD_CONTEXT
3) HAL_THREAD_SWITCH_CONTEXT

HAL_THREAD_LOAD_CONTEXT 负责读取任务的上下文恢复其运行，HAL_THREAD_SWITCH_CONTEXT 负责保存当前任务上下文并读取其它任务的上下文然后恢复其运行。对于第一次运行的任务，理论上没有上下文，但是在某些平台的实现中（如，HCS12利用软中断的上下文切换实现方式）可以事先调用 HAL_THREAD_INIT_CONTEXT 人为初始化上下文，并通过HAL_THREAD_LOAD_CONTEXT启动该任务。

3.2.3 硬件时钟管理

实时内核的时间管理是以硬件时钟为基础的，硬件时钟每隔一定时间间隔产生一个时钟中断，实时内核在时钟中断处理程序内进行时间管理的操作。实时内核对于硬件时钟的管理主要是硬件时钟的初始化、停止以及时钟频率的控制。

SmartOSEK OS 3.0的硬件抽象层规定了以下硬件时钟管理接口：

1) HAL_CLOCK_INITIALIZE
2) HAL_CLOCK_RESET
3) HAL_CLOCK_READ

HAL_CLOCK_INITIALIZE 负责硬件时钟的出世化并设定硬件中断的频率，HAL_CLOCK_RESET 负责在硬件时钟中断时进行必要的操作以保证正确触发下一中断，HAL_CLOCK_READ 读取硬件时钟的值。

3.2.4 中断管理

中断管理是硬件抽象层中最复杂的部分，其原因是不同体系结构的中断机制差别很大，所以提供统一的管理接口比较困难。中断管理主要功能是中断控制器的操作、中断上下文的保存和恢复以及中断向量表的管理。

中断控制器操作主要是中断的开关以及中断优先级的操作，一般通过读写专门的寄存器来实现。中断上下文的保存和恢复和任务上下文的保存恢复类似，不同的是中断常常嵌套发生，上一个中断没有处理完就有可能发生下一个中断，其结果是保存的上下问会逐渐占用更多内存。如果把中断上下文保存在任务栈中则

可能导致栈溢出。

一种解决办法是增大任务栈，这样的一个缺点是如果中断频率不高的话会导致内存浪费。另一种解决方法是设置一个较大的内存栈，当首次中断发生时把当前栈切换到中断栈，这样嵌套的中断上下文保存到中断栈中。当返回任务时再切会任务栈。这种方式的好处是任务栈可以相对较小，内存利用率较高，但是栈切换会对性能有一定影响。

SmartOSEK OS 3.0的做法是根据不同体系结构采用不同做法。对于寄存器数量较少的处理器采用前一种方式，对于寄存器数量较多的处理器采用后一种方式。

中断向量表管理的目的是使得用于可以方便的将自己编写的函数绑定到中断向量表中。SmartOSEK OS 3.0利用软中断向量表来实现这一功能。

各种处理器都会有一个中断向量表，这里称为硬中断向量表。不同体系结构对中断向量表的使用有所不同，用户直接操作不方便，所以硬中断向量表由超核管理。

此外硬件抽象层定义一个软中断向量表，软中断向量表是函数指针数组，用户通过系统服务把其中的定项指向自己写的中断处理函数。硬中断向量表指向的是一段超核的代码，当中断发生时，这段代码负责保存中断上下文，然后判定实际中断源，最后根据中断源在软中断向量表中找到用户定义的函数并执行之。该函数返回后超核负责中断上下文的恢复工作。此外，在这段代码中还可以进行一些内核需要的操作。

SmartOSEK OS 3.0的硬件抽象层规定了以下中断管理接口：

1) HAL_ENABLE_INTERRUPTS：使能中断
2) HAL_DISABLE_INTERRUPTS：关中断并保留中断响应状态
3) HAL_RESTORE_INTERRUPTS：恢复中断响应状态
4) HAL_QUERY_INTERRUPTS：查询中断响应状态
5) HAL_INTERRUPT_IN_USE：查询中断向量是否使用
6) HAL_INTERRUPT_ATTACH：用户程序与中断向量的绑定
7) HAL_INTERRUPT_DETACH：用户程序与中断向量的分离

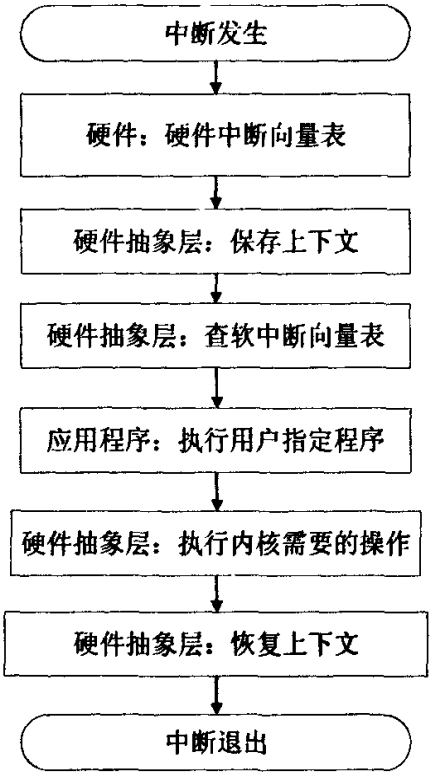

图 3-4 SmartOSEK OS 3.0的中断处理流程

3.3 实时内核层

为了能够兼容多种接口标准，SmartOSEK OS 3.0的内核又细分为超核及内核组件两层。

超核是实时内核的基础，即内核的内核。符合不同标准的实时内核的机制各不相同，但是其中有部分功能是共同的，超核就是实现这些共同的基本功能。SmartOSEK OS 3.0的超核是以OSEK OS（SmartOSEK OS 2.0）为基础，并扩展了部分功能。超核的功能包括：基本任务管理、基本同步机制、资源互斥访问、基本定时器、动态对象管理、任务队列管理以及内存管理。

SmartOSEK OS 3.0使用内核组件来提供具体的内核API。AUTOSAR内核组件扩展了SmartOSEK OS 2.0，增加了AUTOSAR OS要求的功能，使其符合AUTOSAR标准对实时操作系统的要求。POSIX内核组件以SmartOSEK OS 2.0为基础，增加了动态创建和释放内核对象的功能，提供了符合POSIX线程模型的应用程序编程接口。

关于实时内核层的详细情况将在第4章叙述。

3.4 系统服务层

系统服务层为嵌入式实时系统的开发提供了丰富的基础服务。SmartOSEK OS 3.0的系统服务层包括：AUTOSAR兼容的ECU状态管理器、 AUTOSAR兼容的看门狗管理器、AUTOSAR兼容的通信服务、TCP/IP协议栈以及图形用户界面。其中AUTOSAR兼容的ECU状态管理器、 AUTOSAR兼容的看门狗管理器以及AUTOSAR兼容的通信服务依赖与AUTOSAR OS提供的服务，而TCP/IP协议栈以及图形用户界面可以不依赖于操作系统。

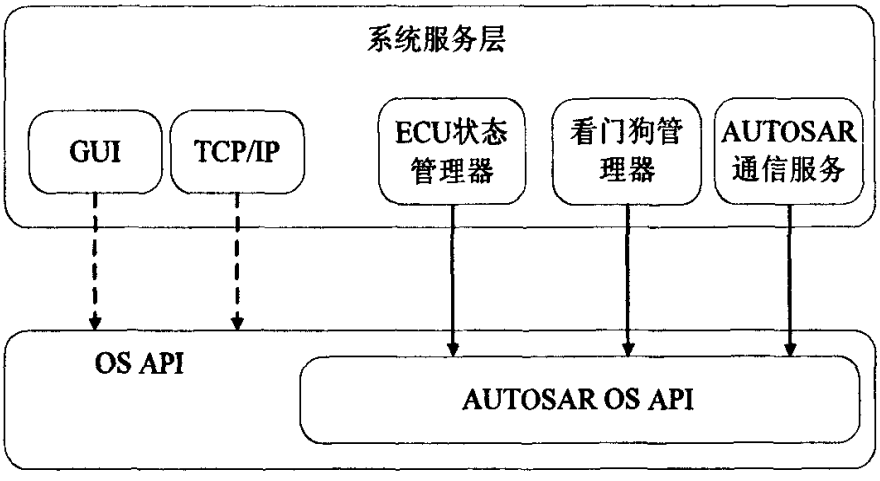

图 3-5 SmartOSEK OS 3.0系统服务层

ECU状态管理器负责维护ECU的停止、运行和睡眠三个状态以及负责在三个状态间进行切换，也就是通常所说的启动和关闭ECU的操作。ECU状态管理器负责所有硬件驱动程序和基础软件模块的启动、初始化和关闭。当应用程序都不需要ECU处于运行状态时，ECU状态管理器负责把ECU切换到睡眠状态并且在合适的时候唤醒ECU。ECU状态管理器提供了管理ECU状态的机制，通过在适当的时候使得ECU进入睡眠状态达到节能的目的。各个应用程序只需要向ECU状态管理器请求运行状态或释放运行状态，ECU状态管理器决定对ECU状态的转换并进行硬件操作。当唤醒事件发生时，ECU状态管理器判断是否真的需要唤醒ECU，从而避免不必要的唤醒操作。

看门狗管理器负责对应用程序的运行进行监控。各个应用程序把各自的时间约束条件通知看门狗管理器，然后在运行期间向看门狗管理器进行运行提示，看门狗管理器根据运行提示的状况以及事先得到的约束条件判断系统的运行状况。如果系统运行正常则根据硬件看门狗的要求按时触发硬件看门狗，如果系统运行超时则停止触发硬件看门狗，从而引发看门狗的报错机制。

AUTOSAR 的通信服务系统提供了基于消息的通信机制。在发送端，应用层向通信服务发送消息，通信服务根据事先配置的情况把消息封装在协议数据单元（PDU）中并选择合适的网络信道发送该 PDU；在接收端，通信服务从网络中接收 PDU，提取其中的消息并提交给应用程序。SmartOSEK OS 3.0 实现了基于 CAN 的 AUTOSAR 通信服务，但其架构可以方便的扩展到其它通信协议。

SmartOSEK OS 3.0 利用开源 TCP/IP 协议栈 uIP 实现 TCP/IP 通信。

uIP 协议栈本身不依赖于任何操作系统，其主体是一个循环，主循环不断的做两件事：检查是否有包到达、检查周期定时是否到期。

当有数据包到达时，主循环调用输入处理函数处理输入的数据包。根据获得数据包的不同类型，输入处理函数将其交给应用程序或使用协议栈中的协议处理该包，当其返回后可能需要发送一个或多个应答包，如果这样，协议栈调用网络设备驱动程序发送该包。周期定时用于处理需要定时服务的 TCP 机制，比如延迟确认、重传数据等。当主循环发现需要触发超时操作时，它将调用超时处理函数。

SmartOSEK OS 3.0 将 uIP 的主循环封装在一个周期性运行的任务中，该任务周期性启动检测是否有底层数据包到达，如果有则调用响应的函数处理之。该任务同时负责检查周期定时是否到期，并进行相应的处理。

μC /GUI 支持三种运行环境：单任务、多任务中单任务调用 GUI、多任务中多任务调用 GUI。

单任务环境实际是一个无限循环，各个组件被循环调用并利用中断机制处理实时部分。由于不使用实时内核所以代码量小，而且只需要一个栈空间所以内存需求也小，而且没有抢占同步问题。但是这种环境只适用于小型系统，而且其实时性能比较差。在单任务环境中使用 GUI 的方式是在循环前初始化 GUI 并在循环中不停进行 GUI 操作。

多任务中单任务调用 GUI，是指在实时内核的支持下，系统应用被划分为多个任务，其中的一个任务负责进行 GUI 操作，该任务往往有十分低的优先级。这种运行环境的好处是实时性能比较好，由于任务的实时性只被高优先级任务影响，所以 GUI 的操作不会影响高实时性要求的任务，但是实时内核的使用增加了一定的系统消耗。由于只有一个任务进行了 GUI 操作，所以对 GUI 来说不需要为多任务提供任何机制上的改变，所以可以被用于多种实时内核。

多任务中单任务调用 GUI，是指在实时内核的支持下，系统应用被划分为多个任务，其中的多个任务负责进行 GUI 操作。用户使用一个任务更新窗体，用其它任务进行 GUI 操作。由于多个任务调用 GUI 所以 GUI 需要负责多个任务间的

互斥和同步。

SmartOSEK OS 3.0中对μC/GUI的使用对应于第二种情况，所有GUI操作都在一个周期性运行的GUI任务中完成。

关于ECU状态管理器、看门狗管理器和AUTOSAR通信服务的详细情况将在第5章叙述。

3.5 本章小结

SmartOSEK OS 3.0采用层次化结构，大体分为硬件抽象层、实时内核层以及系统服务层三层。硬件抽象层实现硬件相关操作，向上层提供平台无关的接口。实时内核层分为超核及内核组件。超核以SmartOSEK OS 2.0为基础实现内核的基本功能，内核组件基于超核实现不同标准规定的内核功能，并提供标准化的应用程序编程接口。SmartOSEK OS 3.0除了提供OSEK兼容接口外还提供AUTOSAR兼容接口及POSIX兼容接口。系统服务层提供各种扩展的系统服务：ECU状态管理器管理ECU的运行、停止和睡眠状态并负责在这些状态间的切换；看门狗管理器提供基于硬件看门狗的应用程序监控机制；AUTOSAR通信服务提供基于消息的统一通信平台；TCP/IP协议栈以uIP为基础提供基于TCP/IP的通信能力；图形用户界面以μC/GUI为基础提供基于LCD的图形交互功能。

第4章 SmartOSEK OS 3.0实时内核层设计与实现

SmartOSEK OS 3.0为了兼容多种实时操作系统标准，把实时内核又细分成两部分：超核以及内核组件。SmartOSEK OS 3.0的超核以SmartOSEK OS 2.0为基础实现各种内核组件需要的基本功能。而内核组件基于超核实现不同标准规定的功能。SmartOSEK OS 3.0的实时内核层能够兼容OSEK/VDX OS标准、AUTOSAR OS标准以及POSIX标准。本章说明SmartOSEK OS 3.0实时内核层的设计与实现。

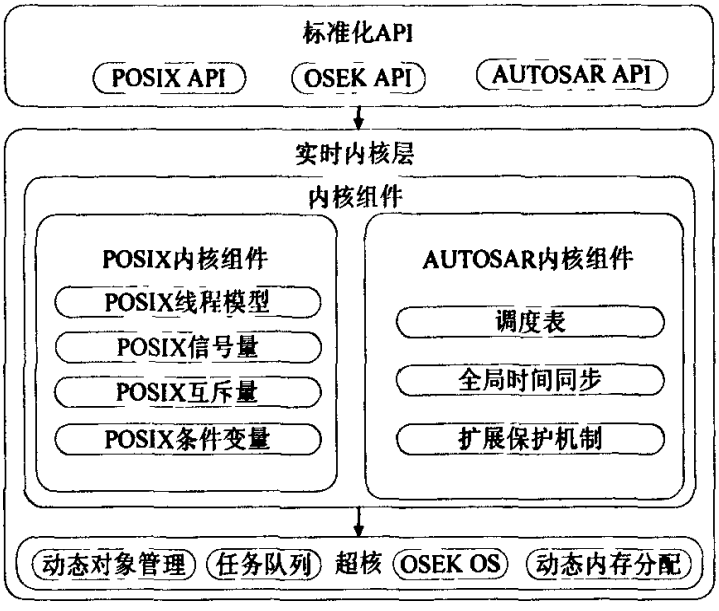

图 4-1 SmartOSEK OS 3.0实时内核层

4.1 SmartOSEK OS 3.0超核

SmartOSEK OS 3.0的超核以SmartOSEK OS 2.0为基础并扩展了动态对象管理、任务队列管理以及基本内存管理功能。

4.1.1 超核的基本功能

SmartOSEK OS 2.0已经实现了基本任务管理、基本同步机制、互斥资源访问以及基本定时器功能，这些功能是SmartOSEK OS 3.0超核的基础。

基本任务管理实现了运行、就绪、挂起和等待四个基本任务状态以及在这四

个状态间的切换。包括任务激活、任务终止、任务调度、任务等待和任务恢复。

表 4-1 SmartOSEK OS 2.0 实现的基本任务管理功能

系统调用	实现功能
ActivateTask	任务激活
TerminateTask	任务终止
WaitEvent	任务等待
SetEvent	任务恢复
Schedule	任务调度

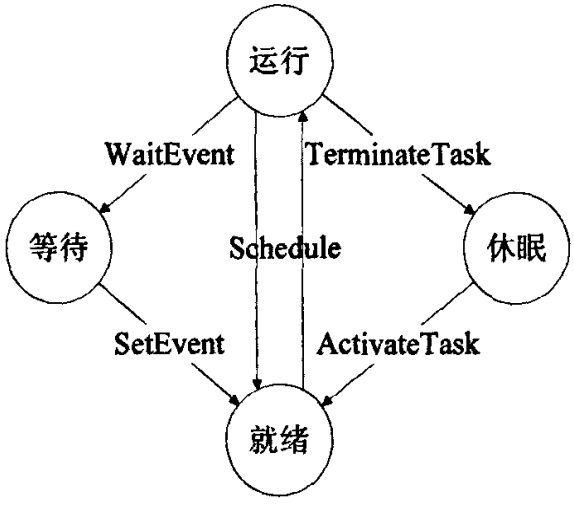

图 4-2 SmartOSEK OS 2.0 任务状态模型

处于休眠状态的任务存在于系统中但是并不准备运行，ActivateTask 使得任务进入就绪状态并参与对 CPU 资源的竞争。SmartOSEK OS 2.0 采用基于优先级的抢占式调度方式，高优先级任务将自动抢占可以被抢占的低优先级的任务进入运行状态。对于不可抢占的低优先级任务用户调用 Schedule 强制进行调度。任务运行完毕后调用 TerminateTask 返回休眠状态。SmartOSEK OS 2.0 任务管理机制提供最多 64 个优先级，但是一个优先级只能有一个任务，也就是系统最多只能有 64 个任务。而且任务的数量和属性是编译时确定的，在运行时不能动态创建和删除任务，不能改变任务的优先级和其它一些属性。

SmartOSEK OS 2.0 任务之间的同步通过事件机制实现。任务调用 WaitEvent 等待事件时进入等待状态，当其它任务对该任务设置事件时调用 SetEvent 恢复该

任务至就绪状态。SmartOSEK OS 2.0实现的同步机制是十分简单的。比如，任务A由于等待事件a而进入等待状态之后必须由其它任务对任务A设置事件a使得其恢复运行。内核不维护等待队列，即内核不统计哪些任务正在等待a事件，当a事件发生时，由程序员负责对所有等待a事件的任务恢复运行。这样做的好处时内核的实现简单高效，但是程序的灵活性已经编程的难度有所增加。

在基于优先级调度的实时操作系统中使用传统同步机制带来的一个问题是优先级反转。当高优先级任务运行需要的资源被低优先级任务占有时就发生了优先级反转。此时如果其它中等优先级的任务不断抢占低优先级任务使其不能立即完成操作并释放资源，则被阻塞的高优先级任务将会长期无法运行。

SmartOSEK OS 2.0利用优先级置顶算法防止优先级反转。优先级置顶算法给资源分配一个优先级，该优先级一般是所有可能占用该资源的任务中优先级最高的任务的优先级。在任务占用资源时把任务优先级提高到这个资源的优先级。其主要的功能是防止优先级反转，对于互斥的实际只是简单的当任务试图占用已经被占用的资源时进行报错处理。此外SmartOSEK OS 2.0以占用资源的形式提供了锁调度器的操作。

SmartOSEK OS 2.0的定时器是为处理循环事件提供的服务机制。当定时器的计数值等于设置的数值时，实时内核就调用服务程序来设置事件或者激活任务。计数器可以设置单个定时器或者周期性定时器。定时器计数值可以是时间中断的个数也可以是角度位置等。SmartOSEK OS 2.0可以通过调用API来取消定时器并得到定时器目前的状态，可以将几个定时器以同一个计数器为基准，但是定时器的参数都是静态配置的。

综上所述，SmartOSEK OS 2.0实现基本的内核功能，所以可以作为超核的基础。但是由于其设计是针对汽车电子特殊领域的需求的，其原则是简单高效，也就是把许多责任推给了程序员，要求程序员保证各种参数的正确性。这就使得其在一些稍微负责一点的应用环境中显得过于简单了，因此有必要对其进行一定的扩充。

4.1.2 通用链表

超核中的许多功能（动态对象管理、任务队列、内存管理）都需要用到链表，因此有必要实现一个可复用的链表操作集合。由于SmartOSEK OS 3.0是用C语言实现的，传统的做法是在每一种需要用到链表结构的对象定义一种链表数据结构，这样的实现显然效率不高。

SmartOSEK OS 3.0 的做法是实现一个通用的链表，该链表中的节点只有两个指针域分别指向前一个节点和后一个节点，通用链表操作仅针对于这样的节点操作。需要使用链表的结构体把第一个数据域定义为通用链表的节点类型，这样指向该节点的指针实际上也就指向了该数据类型。例如，当需要从任务链表中取得第一个任务控制块时，只需要用通用链表操作取得第一个通用节点的位置，由于该位置实际上就是第一个任务控制块的位置，所以只需要进行简单的指针类型转换就能操纵任务控制块了。

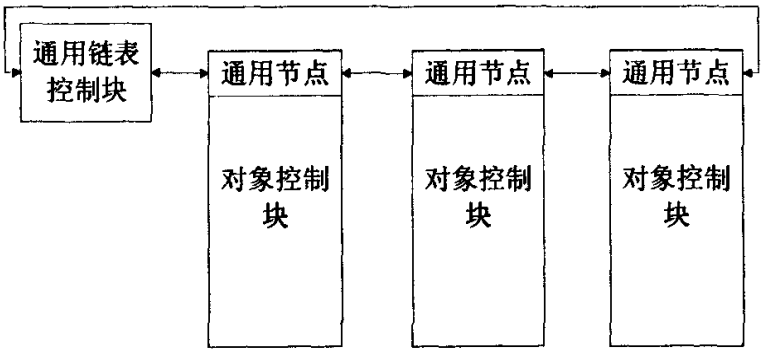

图 4-3 通用链表示意图

4.1.3 动态对象管理

动态对象管理提供了动态创建和释放对象的能力。不同的实时内核对于内核对象控制块的分配策略是不同的。OSEK 和 AUTOSAR 内核各种对象控制块是静态分配的，用户在系统生成是指定内核各种对象控制块数目，系统运行时控制块的数目不能动态改变。比如，用户指定系统中有 5 个任务，则系统在编译是就分配了 5 个任务控制块，这 5 个任务在系统的运行过程中都存在者，数量不会增多和减少，而且其参数是确定不变的。这种内核基本不需要对象的管理操作。

但类似 POSIX 这样的内核支持各种内核对象的动态创建，内核各种对象控制块的数目会在运行时动态改变。比如，系统在运行过程中会动态创建或删除任务，则系统需要在任务创建时负责分配一个空闲的控制块并在任务被删除时回收这个控制块。这就是说一个任务控制块在系统运行过程中可能代表不同的任务，其属性及也会不停的改变。

SmartOSEK OS 3.0 超核是这样实现内核各种对象控制块的动态分配和释放。系统在初始化时分配一定数量的控制块，并用双向链表把这些控制块链接起来，这个双向链表就是空闲链表。当创建对象时就从该链表中分配一个控制块，当删

除对象时就把控制块插回空闲链表。

在简单实现下，当空闲链表为空时，分配新对象的操作就会失败。根据用户配置，SmartOSEK OS 3.0超核可以自动扩展可用对象数。使用自动扩展时，超核将利用动态分配内存功能再多分配一定数量的空闲块并将它们插入空闲队列。

4.1.4 任务队列

任务队列是实时内核任务管理的基本功能之一，任务队列主要用于就绪队列和各种同步机制中的等待队列的实现。SmartOSEK OS 3.0的超核中用双链表实现任务队列，对于任务队列的操作主要是基本的入队和出队操作。

实时内核的一个重要概念是任务的优先级，高优先级任务比低优先级任务更容易获得处理器资源，所以实时内核从任务队列选取任务时有时要求取出的是最高优先级的任务而不是最先进入任务队列的任务。为了满足实时内核的特殊需求，SmartOSEK OS 3.0超核实现两种任务队列，先进先出队列和优先级队列。

先进先出队列就是单个任务控制块组成的双链表。任务进入队列时加入链表尾部，任务出队列时从队列头取出。

优先级队列其实是多个双链表。每个链表中的任务具有相同优先级。任务进入队列是插入任务优先级相应的链表尾部，当从队列中获取任务时，超核从优先级最高的非空链表头取出任务。

就绪队列是记录系统中所有就绪任务的数据结构，它是一种重要的优先级队列。就绪表要求能够迅速找到下一个要运行的任务。在基于优先级抢占的实时系统中运行的任务一般是优先级最高的就绪任务。同一优先级的任务采用先来先服务或时间片轮转方式调度。

为了满足上述需求，SmartOSEK OS 3.0的超核实现了基于优先级的就绪表。该就绪表由优先级队列和优先级位图组成。

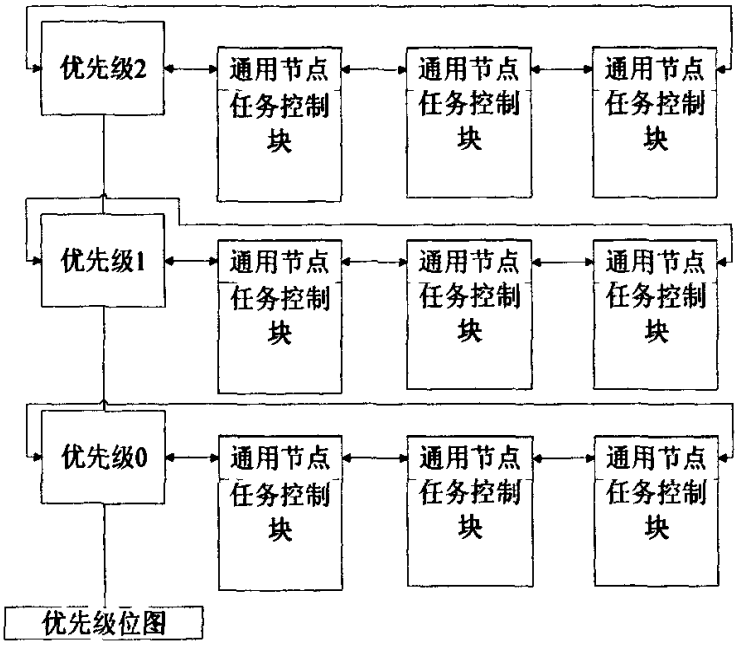

图 4-4 就绪表示意图

优先级队列是多个双链表，每一个链表由具有相同优先级的任务的控制块组成，任务就绪时插入相应队列的队尾，任务运行时从队头出队。

优先级位图表示相应优先级是否有任务就绪。优先级位图中的每一位表示一个优先级的就绪状态，如果该位为 1 表示该优先级有任务就绪，如果为 0 表示该优先级没有任务就绪。任务就绪时把各自优先级对应的位置 1，退出就绪态时置 0。任务调度时在优先级位图中找到最高的就绪优先级，最后在相应的优先级队列中找到就绪的任务。

等待队列实现和优先级队列类似，但是等待队列可以采用先进先出和优先级队列两种方式。先进先出的优先级队列实现简单但是实时性较差，优先级队列能保证较好的实时性但实现较为复杂。

4.1.5 动态内存分配

SmartOSEK OS 3.0 超核实现了内存的动态分配，其提供的主要功能包括：包括内存堆的创建、内存的分配和内存的释放。

嵌入式实时系统中往往内存的数量十分有限，用户往往要求十分严格的控制每一个字节的使用。SmartOSEK OS 3.0 中内存堆由用户创建，用户指明堆的起始地址和大小。这样提高了用户对于动态内存管理的可控制性。

SmartOSEK OS 3.0 的动态内存管理采用边界标识法[38]。系统将所有空闲内存块链接在一个双链表结构中。每个内存区的头部和底部边界上都设有标识以表示该区域为占用块或空闲快，这样在回收用户释放的空闲块时易于判别物理位置上相邻的内存区域是否被占用，以便于合并。

内存分配时采用首次适配法，系统找到第一个符合用户要求的内存块，然后把高地址的内存分配给用户。

当用户释放内存时，根据其相邻内存块情况处理。如果其前后内存块均为占用块则把空闲块插入空闲链表；如果前内存块空闲则与前内存块合并一个空闲块；如果后内存块空闲则与后内存块合并一个空闲块；如果前后内存块都空闲则三个内存块合并一个空闲块。

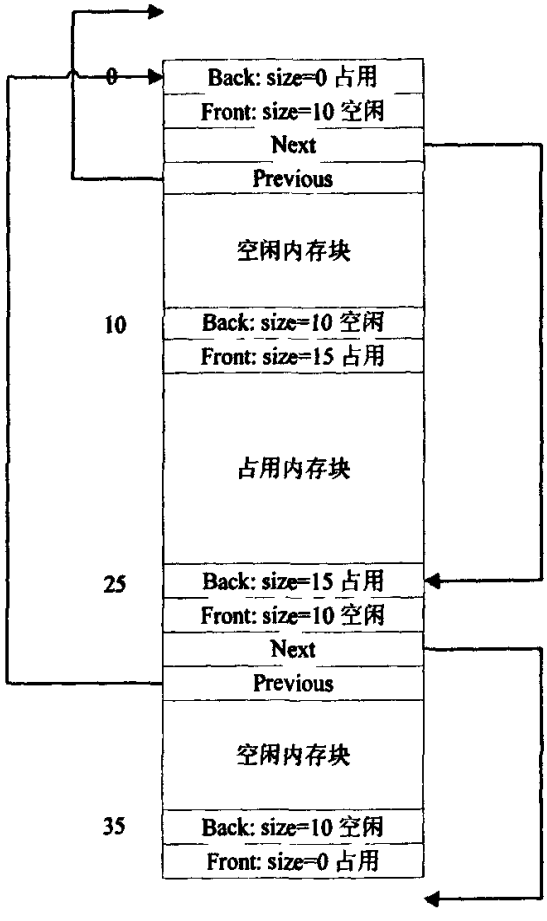

图 4-5 内存分配算法示意图

4.2 AUTOSAR 内核组件

根据 AUTOSAR 标准 AUTOSAR OS 是基于 OSEK OS 的，所以这里就在 SmartOSEK OS 2.0 的基础上根据 AUTOSAR 标准进行扩充而成为 AUTOSAR OS[37]。AUTOSAR OS 对 OSEK OS 的扩展体现在如下部分。

4.2.1 调度表

调度表是一种根据事先配置好的时间点激活一系列任务的机制，该机制在基于事件触发的 OSEK 操作系统中引入了基于时间触发的调度机制。其实际上通过定义一系列 OSEK 定时器来实现[26]。这些定时器的参数在启动后不能被独立地改变，调度表机制提供了对一系列这样的定时器的封装。

一个调度表中包含若干调度点，每一个调度点和一个或多个操作相关联。比如：在 0ms 激活任务 1，在 2ms 激活任务 2，在 8ms 激活任务 3。一个调度点类似于一个 OSEK 定时器，但是在一个调度表中的一系列调度点是静态配置的，操作系统不提供接口对其进行单个的操作。

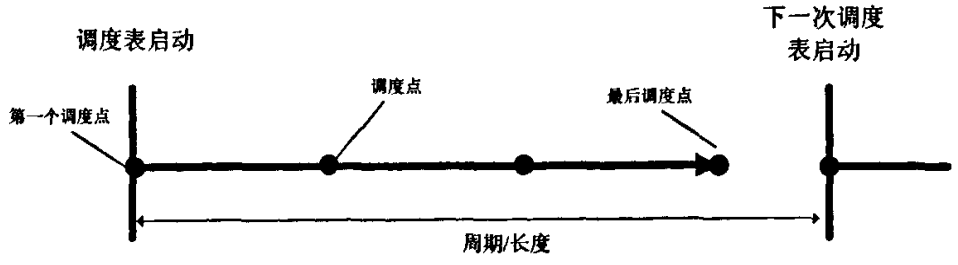

图 4-6 AUTOSAR 调度表模型

调度表分为周期性启动和非周期性启动。非周期性启动的调度表启动后只运行一次，以后需要显示调用系统调用让其再次启动。周期性启动的调度表有个周期表示其连续两次启动的时间间隔，它在后将按照这个周期不断启动。

AUTOSAR 调度表的时间间隔是以 OSEK 计数器为参照的，调度表的启动有两种方式。可以在一个绝对的计数器值时启动也可以在相对当前计数器值经过一个给定的偏移后启动。在启动点，该调度器的第一个调度点被执行，然后根据事先的配置依次执行其它调度点。

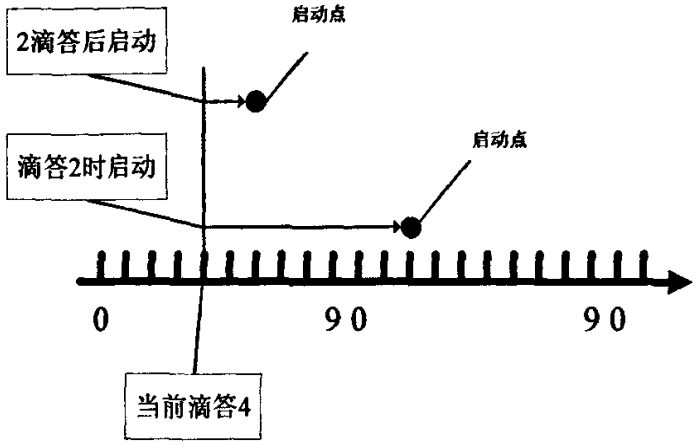

图 4-7 AUTOSAR 调度表启动说明

AUTOSAR 调度表都在系统编译前静态配置，操作系统提供系统调用启动、停止调度表或在多个调度表之间切换。

4.2.2 全局时间同步

在系统开发时有时候需要保证不同处理器上的操作能够在时间上同步，一种常用的方法就是用一个全局的时钟源来同步各个处理器。操作系统利用调度表机制安排任务激活和事件触发，并通过保证本地调度表和全局时间的一致性来保证同步。

一种简单的方法是直接用全局时间来驱动调度表，但是这种方式下如果全局时间丢失，则本地操作也无法正确执行了。另一种更好的方式是用本地时间驱动调度表，然后告诉调度表全局时间，操作系统的职责就是保证本地时间和全局时间的偏差在一个可以接收的范围之内。

当使用第二种方式时，调度表可以同步启动也可以异步启动。同步启动就是仅当本地时间和全局时间同步后调度表才会启动，异步启动不需要这种同步。异步启动后可以通过两种方式进行重新同步：硬同步和平滑同步。所谓硬同步就是当得知全局时间后，在调度表结束的时候把本地时间设为全局时间，在调度表下次启动时获得同步。平滑同步就是在被告知全局时间后在每个调度点之间根据事先设定的量进行时间调整直到完成同步。

4.2.3 保护机制

AUTOSAR OS 还规定了一系列保护机制。

即使在没有内存保护支持的处理器上有时也需要提供一定的软件内存保护。由于 AUTOSAR OS 的任务在任务切换时会把上下文保存在栈中，而且基于 C 语言的运行环境也会占用栈空间，所以一个常见的错误就是栈的溢出，由于 AUTOSAR OS 不支持内存动态分配所以如果没有为任务分配足够的栈空间则栈溢出很有可能发生，所以 AUTOSAR OS 需要提供栈检测。

在 AUTOSAR 内核组件中通过如下方式实现栈检测。在系统初始化阶段在分配的栈区中预先填充一个特定的模式，在系统运行时如果该栈区被使用则相应内存区的模式会被破坏，这样就能统计栈区的利用情况。在上下文保存或中断发生时通过检测栈区边界的模式是否被破坏来检测栈是否溢出。

AUTOSAR OS 通过监控如下时间元素提供时间保护：

1) 任务和二类中断的运行时间
2) 任务和二类中断占用资源和关闭中断的时间
3) 任务和二类中断的发生速率。

AUTOSAR OS 把任务和二类中断的运行时间限定在一定时间内，同时 AUTOSAR OS 还限定了对象在一定时间内运行的次数。当上述限定被打破时，AUTOSAR OS 会调用特定的钩子点进行处理。

AUTOSAR OS 还提供了系统服务的保护，这种保护处理诸如非法服务调用、非法参数等错误。如果硬件支持，AUTOSAR OS 还提供处理器特权级别的控制。

AUTOSAR OS 利用操作系统应用的概念组织各种操作系统对象，如：任务、中断、定时器、钩子点。用户把一系列这样的对象定义为一个操作系统应用，操作系统根据操作系统应用来分配系统资源。属于同一个操作系统应用的各个对象可以互相访问，属于不同操作系统应用的对象是否可互相访问根据配置而不同。AUTOSAR OS 区分两种操作系统应用，可信的操作系统应用和非可信的操作系统应用。

可信的操作系统应用对于系统资源（内存、操作系统 API）的访问没有限制，如果处理器支持特权模式，则它可以在特权模式下运行。

非可信的操作系统应用对于系统资源（内存、操作系统 API）的访问收到限制。如果处理器支持特权模式，则它可以在特权模式下运行。

AUTOSAR OS 提供系统调用用于查询对象的访问权限和其所属的操作系统应用。

4.3 POSIX内核组件

SmartOSEK OS 3.0的POSIX内核组件，以超核为基础实现了POSIX1003.13标准规定的最小实时系统中的部分内核功能。POSIX1003.13认为最小实时系统不需要用户交互以及文件系统，其编程模型是一个单POSIX进程，该进程对应于处理器的硬件地址空间，并包括多个线程，即传统实时操作系统的任务。该系统需要提供线程间的同步机制并支持单处理器。

在研究了一些商用微型实时内核后发现，POSIX的线程模型比较好的反应了实际应用领域队嵌入式实时操作系统的要求。这种应用环境下一般不需要文件系统的支持，而且一般从硬件和软件上都不支持进程的独立性，所以一般不支持多进程。

SmartOSEK OS 3.0的POSIX内核组件以超核为基础实现POSIX多线程模型和基本同步机制。

4.3.1 POSIX多线程模型的实现

SmartOSEK OS 3.0的POSIX内核组件实现了POSIX的单进程多线程模型。其中的线程相当于传统实时系统中的任务的概念。POSIX内核组件实现了基本的线程管理功能，主要包括：各种线程属性的定义、线程的创建、线程汇合以及线程终止。SmartOSEK OS 3.0的POSIX内核组件把线程对应为OSEK OS中的任务，并利用OSEK OS的功能实现POSIX线程模型。

4.3.1.1 POSIX线程属性

POSIX的线程属性主要包括：线程栈的位置与大小、竞争范围、调度策略、调度参数以及分离状态。

线程栈的位置与大小表明用户指定的线程栈。如果用户定义了线程栈则使用用户定义的栈，如果用户没有定义线程栈则使用系统分配的栈。

竞争范围表明线程竞争CPU的范围，分为系统范围（PTHREAD_SCOPE_SYSTEM）和进程范围（PTHREAD_SCOPE_PROCESS），由于SmartOSEK OS 3.0的POSIX内核组件采用单进程多线程模型，所以只支持进程范围的竞争。

调度策略表明线程使用的调度策略。SmartOSEK OS 3.0的POSIX内核组件支持两种调度策略：先进先出（SCHED_FIFO）以及时间片轮转（SCHED_RR）。这两种方式都是基于优先级的可抢占调度。主要区别是当具有相同优先级的任务

就绪时，先进先出方式首先运行先进入就绪状态的任务，当该任务运行完毕或阻塞后再运行后进入就绪状态的任务。时间片轮转调度方式让每个任务运行一段时间然后切换到下一个任务。

调度参数指明线程的优先级。SmartOSEK OS 3.0 的 POSIX 内核组件提供最多 64 个优先级，其中 0 优先级最低，63 优先级最高。0 优先级系统保留为空闲任务，1～31 用于使用时间片轮转的任务，32～83 用于使用先进先出方式的任务。

分离状态表明线程是否分离，分离的线程不能调用 pthread_join 对其进行等待。

POSIX 模型扩展了原有任务的属性集，POSIX 内核组件需要实现 POSIX 的线程模型需要对原有 SmartOSEK OS 2.0 任务控制结构的扩展。传统的做法是在原有任务控制块中增加所有需要的数据域，并用条件编译提供配置选项，但这种做法的可扩展差。这里的做法是在原有的任务控制块中增加一个指针指向一个 POSIX 线程控制块。POSIX 线程控制块有 POSIX 内核组件负责维护。这样做的好处是，如果以后需要扩展超核使其支持其它标准只需要将指针指向不同的扩展块即可而不需要修改超核的实现。

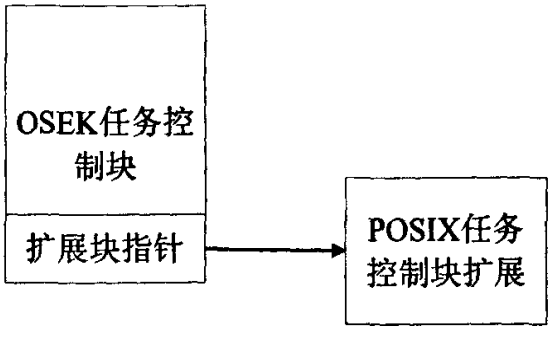

图 4-8 POSIX 任务扩展块示意

4.3.1.2 POSIX 线程创建

POSIX 线程模型使用 pthread_create 系统调用根据线程属性创建线程。线程一旦创建就参与 CPU 的竞争。POSIX 内核组件利用 OSEK OS 的任务管理机制实现任务就绪操作，并利用超核的动态对象管理功能复用 OSEK 的任务控制块实现动态任务创建。

使用 OSEK 任务模型实现 POSIX 任务模型的一个问题是 POSIX 线程的执行体和 OSEK 任务体的函数原型不同。POSIX 任务执行体为有 void *类型的参数和返回值，而 OSEK 任务体的单数和返回值。SmartOSEK OS 3.0 的做法是有 POSIX

内核组件中的 POSIX 任务控制块指向 POSIX 原型的执行体，OSEK OS 的任务控制块指向 POSIX 内核组件实现的统一入口函数，该入口函数负责根据 POSIX 任务控制块中的信息调用实际的线程执行体。

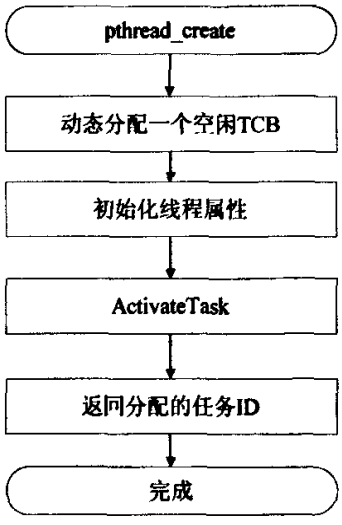

图 4-9 创建 POSIX 线程的实现

上图说明了 POSIX 线程创建的过程。首先调用超核的动态对象分配操作分配一个空闲的任务控制块，并对其中的数据域进行初始化。然后调用 ActivateTask 激活该线程，最后返回获得的任务的 ID。这时新的线程已经存在于系统中并参与 CPU 的竞争。如果新的线程优先级高于创建它的线程，那它将抢占当前线程。否则将根据当前线程的调度策略再适当的时候获得运行机会。

4.3.1.3 POSIX 线程汇合

pthread_join 系统调用将阻塞当前线程直到指定线程退出运行。这里的实现利用 OSEK OS 的事件机制实现线程的阻塞和恢复，并利用任务队列实现等待队列。等待线程完成的实现方式就是先判断线程是否可调度，然后在锁调度器的前提下调用 WaitEvent 使得当前线程退出就绪状态进入等待状态，然后把当前任务加入目标线程的 join 线程队列，最后开调度器并调用 Schedule 进行调度。这时其它线程将占用 CPU 运行，而调用该系统调用的线程将一直处于等待状态直至目标线程退出运行。

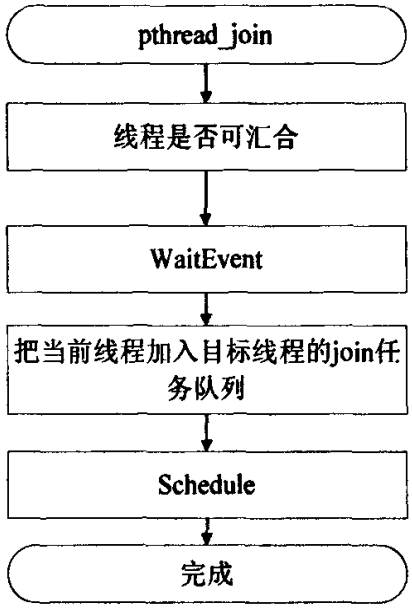

图 4-10 POSIX 线程汇合的实现

4.3.1.4 POSIX 线程退出

当 POSIX 线程执行完毕时调用 pthread_exit 系统调用退出运行。

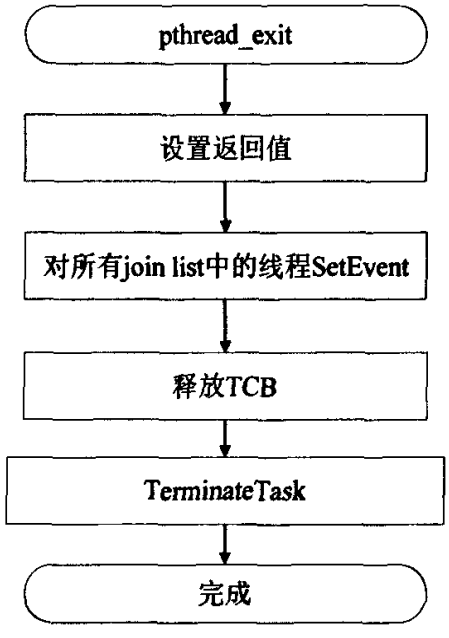

图 4-11 POSIX 线程退出的实现

这里利用 OSEK OS 的事件机制恢复汇合线程的运行并利用动态对象管理释放线程控制块。具体实现过程是：首先记录结束线程的返回值，然后对关调度器的前提下把所有在当前线程 join 队列中的线程执行 SetEvent 使其返回就绪状态，然后释放任务控制块使其处于空闲状态，最后调用 TerminateTask 真正释放任务控制块，并选择下一个运行的线程并调度。

4.3.2 POSIX 线程同步机制的实现

POSIX 内核组件的线程同步机制主要包括互斥量、信号量以及条件变量。

4.3.2.1 POSIX 互斥量

互斥量提供了对资源的互斥访问机制，主要操作为锁互斥量和解锁互斥量。一个互斥量只能被一个线程锁住，该线程就占用了该互斥量，其它试图锁该互斥量的线程将被阻塞，当占用互斥量的线程解锁互斥量后，互斥量等待队列中的第一个线程占用该互斥量。

POSIX 互斥量的属性包括：共享范围、互斥策略以及递归互斥。

共享范围表明互斥量的共享范围，PTHREAD_PROCESS_PRIVATE 范围表明该互斥量只在进程范围内有效，PTHREAD_PROCESS_SHARED 在整个系统范围内有效。

互斥策略表明防止优先级反转的策略。PTHREAD_PRIO_INHERIT 表明使用优先级继承，PTHREAD_PRIO_NONE 表示不使用任何策略，PTHREAD_PRIO_PROTECT 表示使用优先级置顶。

递归互斥表明互斥量是否可以被一个线程多次占用。可递归互斥的互斥量运行被一个线程多次占用，每一次占用将一个计数器加一，每一次释放减去一，当计数器为 0 时真正释放互斥量。

锁互斥量的过程如下，首先判断是否可以占用互斥量，如果可以则判断是否是优先级置顶协议，如果是且资源优先级大于线程优先级，则利用 OSEK 的 GetResource 实现的优先级置顶协议提升线程的优先级，然后成功返回。如果无法占用资源则判断是否使用优先级继承，如果是且占用互斥量的线程优先级低于当前线程优先级则需要提升占用线程的优先级到当前线程的优先级。然后阻塞当前线程，即调用 WaitEvent 退出就绪状态，然后加入互斥变量的等待队列，最后调度执行其它线程。

释放互斥量首先判断当前线程是否有优先级提升，如果有则恢复优先级，然取出等待队列中的第一个线程，使其占用互斥量，然后调用 SetEvent 使其恢复就

绪状态，并引发调度。

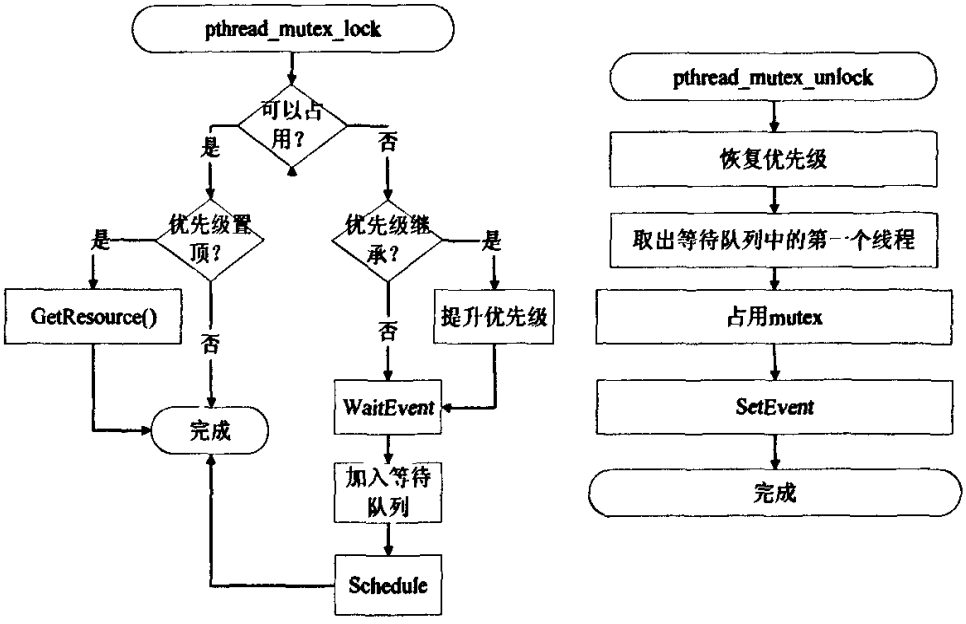

图 4-12 POSIX 互斥量的实现

4.3.2.2 POSIX 信号量

POSIX 信号量是 POSIX 的基本同步机制。信号量的操作为锁信号量和信号量，信号量有个非负数的计数器，每次锁信号量时如果计数器值为正数则计数器值减去一；如果锁信号量时计数器的值为 0 则阻塞该线程。解锁信号量时将计数器值加一，如果之前计数器为正值则继续操作，如果是 0 则恢复一个等待该信号量的线程。

POSIX 内核组件实现了 POSIX 的非命名信号量，并通过动态对象管理机制支持信号量的动态创建和删除。

锁信号量时，判断信号量值是否为 0，如果不为 0 则将值减去一并返回，如果为 0 则调用 WaitEvent 将当前线程退出就绪状态，然后当前线程加入信号量的等待队列，最后调用 Schedule 进行调度。

解锁信号量时判断当前值是否为 0，如果不是则将值加一退出，如果是则取出等待队列中的第一个线程，对其调用 SetEvent 恢复就绪状态。

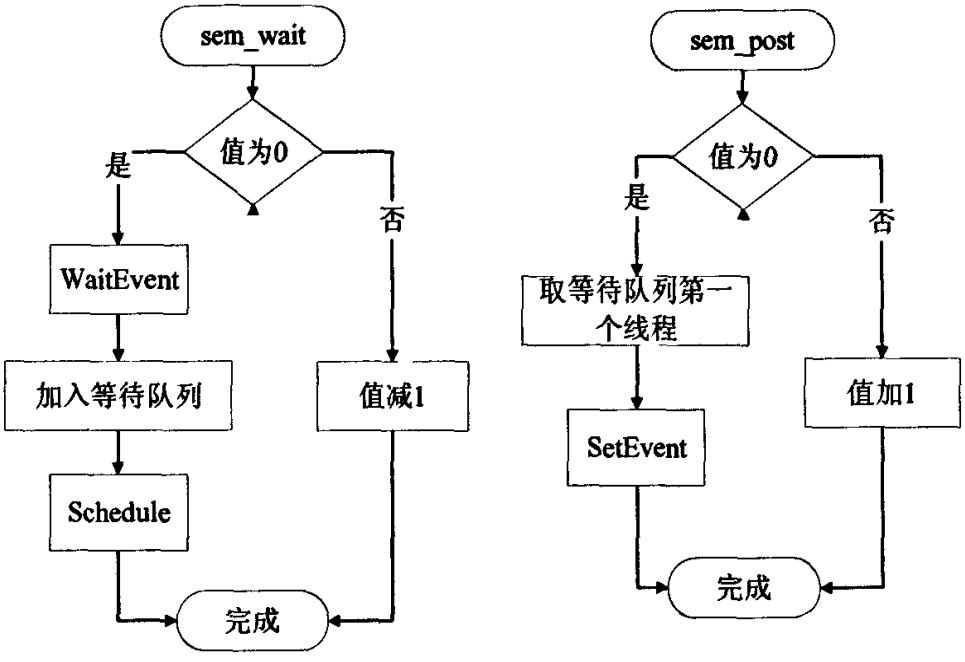

图 4-13 POSIX 信号量的实现

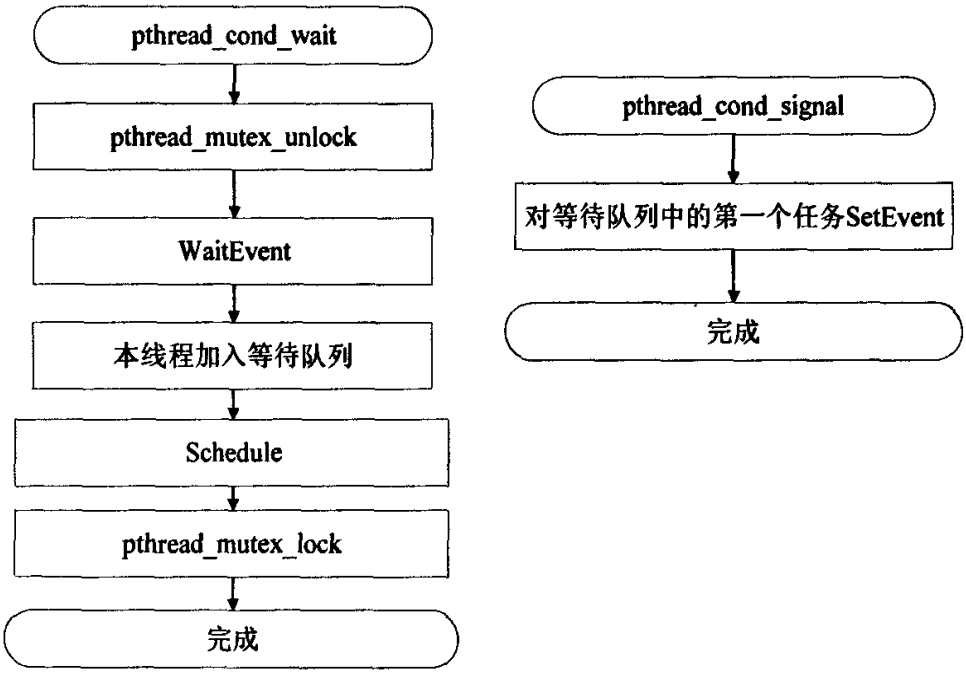

图 4-14 POSIX 条件变量的实现

4.3.2.3 POSIX 条件变量

POSIX 条件变量提供的同步机制主要操作为等待条件变量、触发条件变量和广播条件变量。线程等待条件变量而进入等待状态，当其它线程触发该条件变量后才恢复运行。一个条件变量对应于一个互斥量以防止优先级反转。

POSIX 内核组件利用动态对象管理实现条件变量的动态创建和删除。

等待条件变量时先解锁对应的互斥量，然后调用 WaitEvent 使得当前任务进入等待状态，接着将本线程加入条件变量的等待队列，最后进行调度。

触发条件变量时对等待队列中的第一个任务 SetEvent 使其恢复就绪状态。广播条件变量就是对等待队列中的所有任务 SetEvent。

4.4 本章小结

SmartOSEK OS 3.0 的内核采用层次化设计。最低层的是超核，超核之上是内核组件。

超核实现以 SmartOSEK OS 2.0 为基础，并针对其不足进行扩展，实现最基本的内核功能，包括基本任务管理、基本同步机制、资源互斥访问、基本定时器机制、动态内存管理、任务队列管理和动态对象管理。其中任务队列为实现复杂的任务调度机制和同步机制奠定了基础。动态内存管理实现了动态的内存分配操作。动态对象管理实现了动态创建和释放内核对象的功能。

内核组件利用超核提供的服务实现符合不同标准的实时内核。SmartOSEK OS 3.0 在 OSEK OS 范围内继承了原先版本的设计和实现。AUTOSAR 内核组件实现了 AUTOSAR 内核功能，最主要的是实现了调度表从而引入了基于时间的调度机制。POSIX 内核组件提供了符合 POSIX 标准的实时线程模型以及基本的线程同步机制，如：互斥量、信号量以及条件变量。

第5章 SmartOSEK OS 3.0 系统服务层设计与实现

AUTOSAR 的系统服务和通信服务属于 AUTOSAR 的基础软件模块。其提供在单个电子控制单元（ECU）上运行 AUTOSAR 系统的基本功能，提供了运行 AUTOSAR 软件组件的基本服务。SmartOSEK OS 3.0 系统服务的一大特点是提供与 AUTOSAR 标准兼容的系统服务和通信服务。本章描述了 SmartOSEK OS 3.0 系统服务层提供的 ECU 状态管理器、看门狗管理器以及 AUTOSAR 通信服务。对于 TCP/IP 以及 GUI 的支持已经在第 3 章中有所说明。

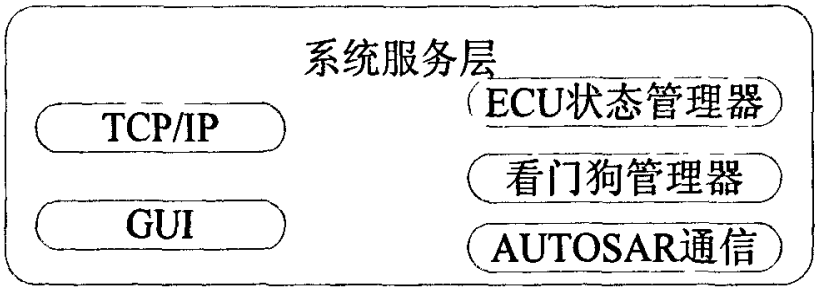

图 5-1 SmartOSEK OS 3.0 系统服务层

5.1 ECU 状态管理器

ECU 即电子控制单元，ECU 状态管理器负责维护 ECU 的运行、关闭和睡眠状态以及在这些状态间的转换[35]。该模块的任务是：

1) 负责操作系统基础软件模块的初始化
2) 管理唤醒事件维护 ECU 的睡眠状态
3) 提供运行请求协议：决定 ECU 是保持运行状态还是关闭

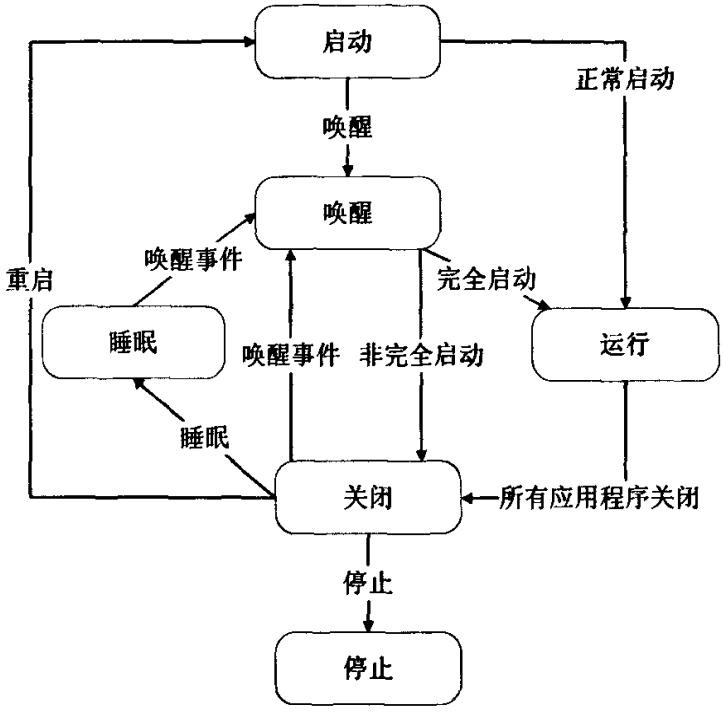

图 5-2 ECU 状态管理器的状态转换图

5.1.1 启动状态

启动状态就是初始化各个基础软件模块和硬件驱动程序的状态。启动状态分为两个部分：操作系统启动前和操作系统启动后。前一个状态的主要任务是为 OS 的启动做准备，该过程尽可能的短；第二个阶段进行需要 OS 支持的初始化。

ECU 的启动过程如下：

1) 硬件启动，BSP 进行基本初始化操作，ECU 状态管理器被启动。
2) ECU 状态管理器进行第一阶段初始化操作。
3) 启动操作系统。
4) 操作系统调用 ECU 状态管理器的系统服务进行第二阶段初始化操作。
5) 进入运行状态或唤醒状态。

第一阶段初始化操作主要是为操作系统的启动做准备，该过程要尽可能简短。该过程将调用一个特定回调函数，该函数具体操作由用户定义，一般在该函数中进行硬件驱动的初始化。

第二阶段运行时，ECU 状态管理器已经处在操作系统的调度之下了。该阶段 ECU 状态管理器负责启动其余需要启动的硬件驱动程序以及其它基础模块，比

如，看门狗管理器。

在两阶段初始化完毕后，ECU 状态管理器判断本次启动的原因，如果是正常启动则进入运行状态，如果是一次唤醒操作则进入唤醒状态。

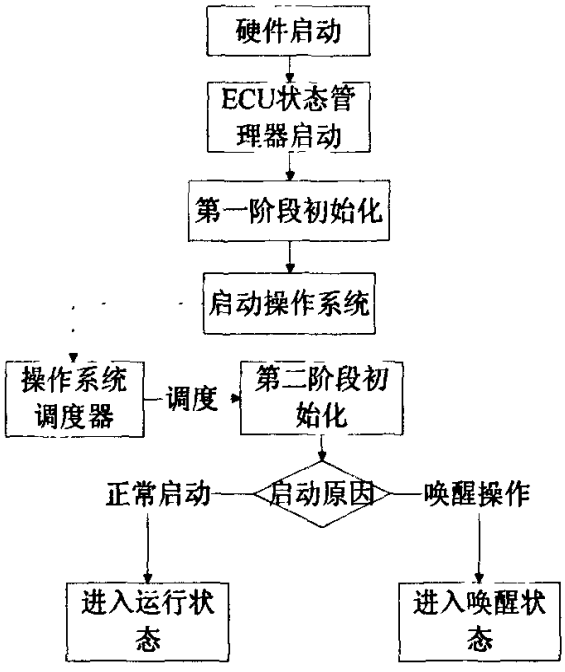

图 5-3 启动状态处理流程

5.1.2 运行状态

当所有基础软件模块被初始化后，ECU 进入运行状态。运行状态表示应用程序开始执行。ECU 状态管理器在运行状态的工作都由操作系统调度 ECU 状态管理器的系统服务完成。所有运行状态详细分为正常运行状态和运行后两个子状态。

正常运行状态中应用程序正常执行各自功能。进入运行状态后，ECU 状态管理器将自己请求运行状态，该请求将持续一段预先配置的时间，然后 ECU 状态管理器释放运行请求。然后在操作系统每次调度 ECU 状态管理器时判断是否有持续的运行请求，如果有则保持在运行状态，如果所有运行请求都被释放则进入运行后状态。

运行后状态在应用程序结束运行和关闭之间提供一段时间的缓冲，使得应用程序可以保存关键数据并关闭外设。这种机制提供了同步关闭的功能。在操作系统每次调度 ECU 状态管理器时如果是处于运行后状态则判断是否有应用程序请

求运行后状态，如果有则保持在运行后状态，如果没有则进入关闭状态。

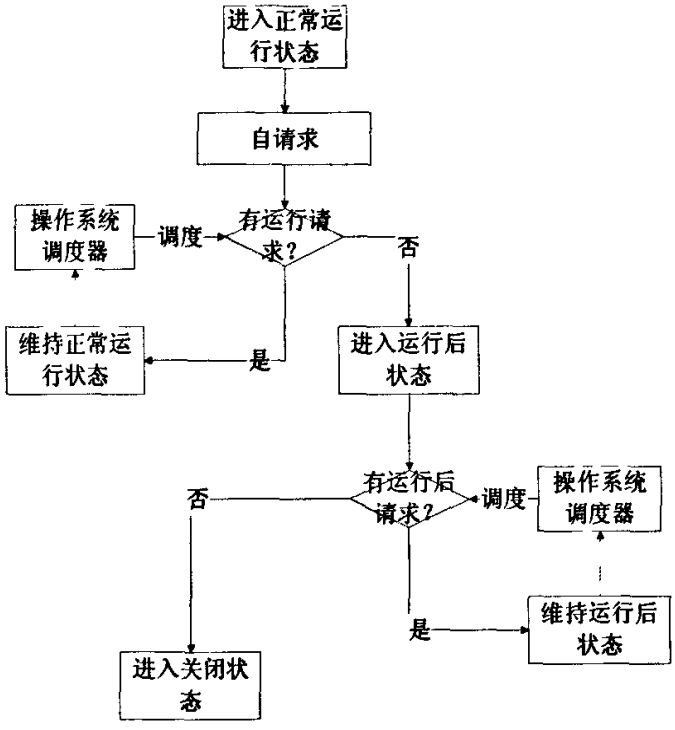

图 5-4 运行状态处理流程

5.1.3 关闭状态

关闭状态执行基础软件的关闭操作。关闭状态只是一个中间过度状态，最后将进入睡眠状态或停止状态或进行重启。如果目标是进入睡眠状态，则关闭过程中如果有唤醒操作则立即进入唤醒状态。如果目标是停止或重启且发生唤醒操作，则关闭操作将被进行完毕，但是之后立即进行重启。

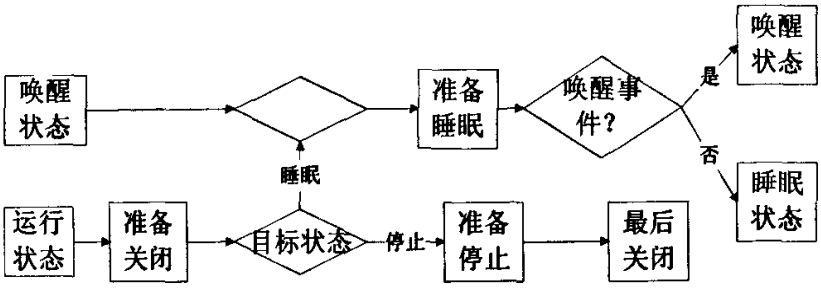

图 5-5 关闭状态处理流程

关闭状态的操作根据进入关闭前的状态以及关闭的目标状态的不同而不同。

如果是从运行状态进入关闭状态则先进行准备关闭操作，该操作将通知应用程序系统将关闭。进行完该操作后将判断目标状态是停止还是睡眠，如果是从唤醒状态进入关闭状态，则直接判断目标状态而不进行准备关闭操作。

如果目标是睡眠状态则进行准备睡眠操作，该操作判断是否有未处理的唤醒请求，如果有则立即进入唤醒状态，如果没有则调用硬件驱动程序把硬件设备转入睡眠状态，然后系统进入睡眠状态。

如果目标是停止状态，则 ECU 状态管理器进行准备停止操作，关闭所有基础软件模块，最后关闭操作系统。操作系统关闭后调用回调函数进行最后关闭操作，该操作判断是否重启，如果重启则进行重启操作重新进入启动状态，如果确实停止则最后关闭微控制器进入停止状态。

5.1.4 睡眠状态

睡眠状态是一种节能状态。该状态中外围硬件设备被关闭或休眠，该状态不执行任何代码。根据配置，ECU 可以从这个状态被唤醒。

5.1.5 唤醒状态

ECU 从睡眠状态被唤醒或重启后会进入唤醒状态，其处理过程如下。

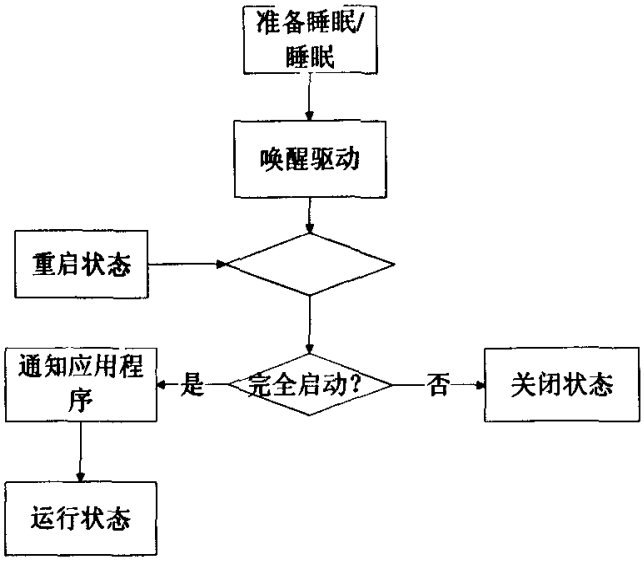

图 5-6 唤醒状态处理流程

如果是由于唤醒事件从睡眠状态被唤醒，则首先唤醒硬件驱动。如果是由于重启进入唤醒状态则已经在启动状态中完成了基本的启动操作。然后根据唤醒事件不同进行不同操作。如果是完整的启动过程，则通知应用程序唤醒操作已经完毕，进入运行状态。如果是在睡眠时被请求了关闭操作，则进入关闭状态。

5.2 看门狗管理器

看门狗管理器是通过周期新检测应用程序运行状态来监控应用程序可靠性的一种系统服务。看门狗控制器采用了层次化架构，以松耦合的形式整合看门狗硬件和应用程序，向应用层提供了一种灵活的可靠性监控机制。[36]

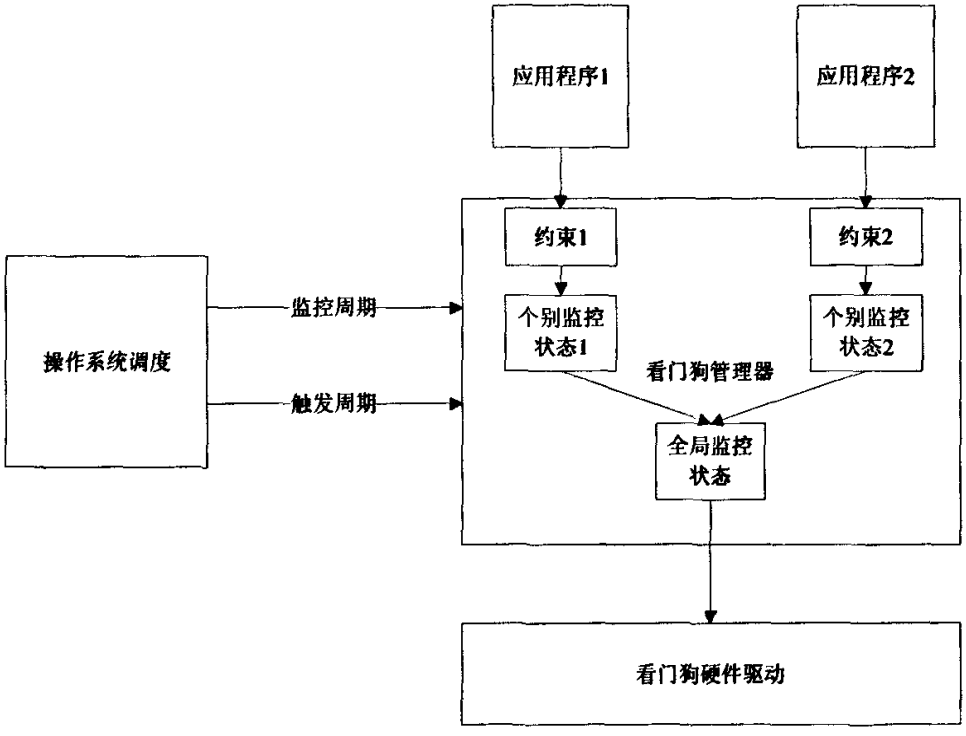

图 5-7 看门狗管理器服务

如图，多个应用程序把各自的时间约束条件以配置的参数的形式提供给看门狗管理器。在运行过程中应用程序使用特定的系统调用向看门狗管理器表明自身的运行状态，看门狗管理器根据各个应用程序报告来的情况分别判断每个应用程序的运行状况，并根据所有应用程序的运行状况判断整个系统的运行状态。

看门狗管理器根据整个系统的状态进行硬件看门狗的触发操作。当所有应用程序运行正确时，看门狗管理器负责按照硬件看门狗的特性正确触发硬件看门

狗。当轻微错误发生是，看门狗管理器将继续触发硬件看门狗，防止立即引发看门狗的出错机制，并在这段时间内提供容错能力，允许系统返回正确运行的状态。当系统错误无法修复时，看门狗管理器会提供一段时间，使得应用程序有时间进行出错处理，然后停止触发硬件看门狗，这将导致引发硬件看门狗的出错机制。

看门狗管理器通过底层硬件驱动进行看门狗的硬件操作，要求此驱动程序符合 AUTOSAR 标准，其具体实现不在本文讨论范围内。

5.2.1 运行监控

看门狗管理器的运行监控机制分别监控各个应用程序，并判断其是否满足各自的周期性时间约束，其判断状态被称为个别监控状态。图 5-8 说明了看门狗管理器的监控算法。看门狗管理器根据所有其监控的应用程序的个别监控状态判断整体系统的运行状态，即全局监控状态，最后根据全局监控状态判断是否触发硬件看门狗。

应用程序在运行过程中周期性调用看门狗管理器提供的系统服务以表明自己的运行状态，这种操作称为运行提示，看门狗管理器会记录每个应用程序进行运行提示的次数，并周期性检测各个应用程序进行运行提示的次数是否符合其运行约束，一次判断其监控状态，这个检测的周期就称为监控周期。该周期由操作系统周期性调度看门狗管理器的特定系统服务实现。

被看门狗管理器监控的应用程序以配置参数的告诉看门狗管理器在一定监控周期内需要进行多少次运行提示，以此明确其周期性运行时间约束。

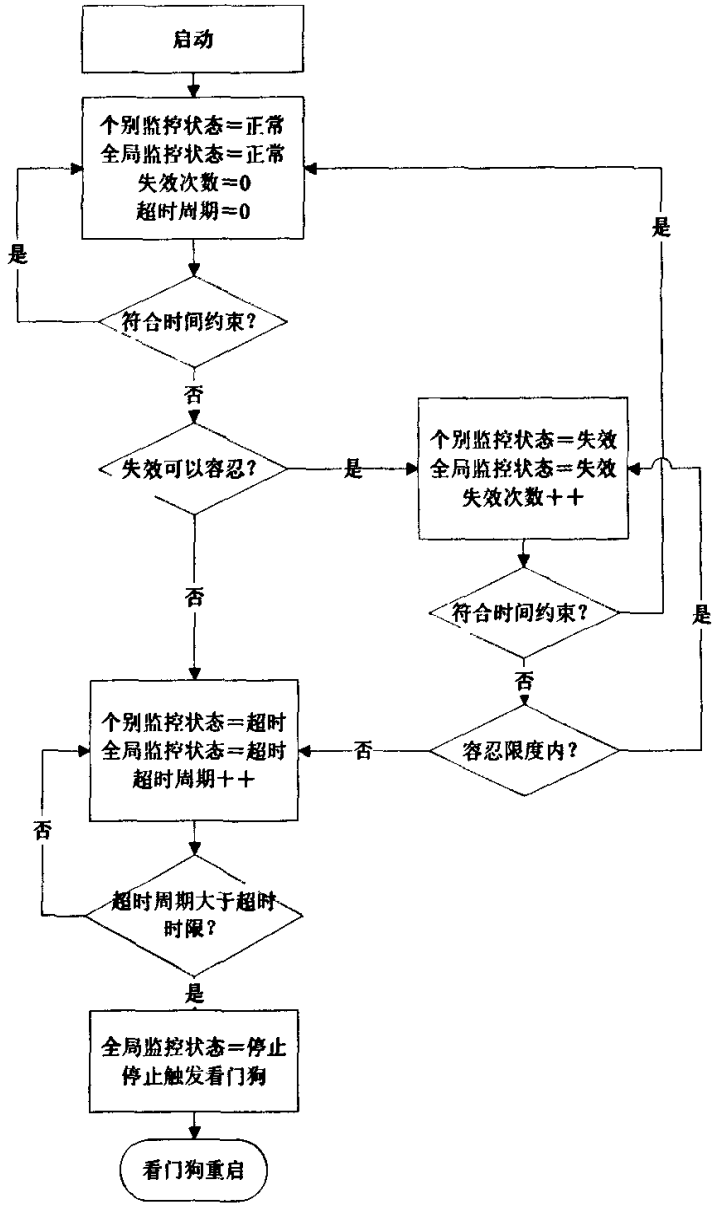

图 5-8 看门狗管理器监控算法

被监控的应用程序指定自己的参考监控周期，参考监控周期为监控周期的倍数，看门狗管理器按照参考监控周期检测应用程序的状态，其步骤是：

首先判断进行运行提示的次数是否满足其事先设定的约束，如果不满足，该次监控失败，则发生一次监控失效，看门狗管理器记录每个应用程序监控失效的次数。

然后，判断监控失效次数是否超过该应用程序事先配置的限度，如果超过，则发生一次监控超时，看门狗管理器记录每个应用程序发生监控超时的个数。

接着，看门狗管理器根据各个应用程序监控失效的次数判断其个别监控状态。共有三种状态：

- 正常

如果运行提示次数符合实现设定的约束，则该应用程序处于正常状态。

- 失效

发生了监控失效，但次数在可以容忍的限度之内，则该应用程序处于失效状态。当失效次数没有超过一个事先设定的容忍度之前，如果在某此参考监控周期内符合时间约束，则其个别监控状态可以立即返回正常，以此提供一定的容错能力。

- 超时

监控失效的次数超过容忍限度，则该应用程序处于超时状态。

如果应用程序在处于失效状态使得某个参考监控周期内符合时间约束，则其可以立即返回正常状态，一次提供一定的错误容忍能力，但是应用程序一旦进入超时状态，其状态不能被改变。

在每个监控周期中，看门狗管理器根据个别监控状态判断全局监控状态：

- 正常

如果所有应用程序的个别监控状态为正常，则全局监控状态为正常。

- 失效

如果至少一个应用程序的个别监控状态为失效，则全局监控状态为失效。

- 超时

如果至少一个应用程序的个别监控状态为超时，则全局监控状态为超时。用户必须指定一个超时时限，看门狗管理器处在超时状态的时间不会超出该时限。在这段时间内，应用程序处理一些系统失败的善后工作。

- 停止

当看门狗管理器处在超时状态的时间不超出该超时时限，全局监控状态为停止。

5.2.2 硬件看门狗的触发

传统应用中，应用程序根据硬件看门狗的时间约束周期性触发看门狗以防止其超时重启。这里触发硬件看门狗的工作由看门狗管理器完成，以此实现应用程

序监控和硬件操作的松耦合。

看门狗管理器的一个功能是按照一定周期触发硬件看门狗，由于单个 ECU 可能会有多个硬件看门狗，而每个硬件看门狗可能有不同的时间约束，看门狗管理要以一个合适的周期，进行硬件看门狗的触发。这里看门狗管理器通过最大公约数方法决定触发硬件看门狗的周期。

如图，硬件看门狗 1 要求每 80ms 触发一次，硬件看门狗 2 要求每 60ms 触发一次，侧看门狗管理器触发硬件看门狗的周期为 80 与 60 的最大公约数，为 20ms。每 20ms 看门狗管理器判断是触发硬件看门狗 1 与硬件看门狗 2 中的哪个。

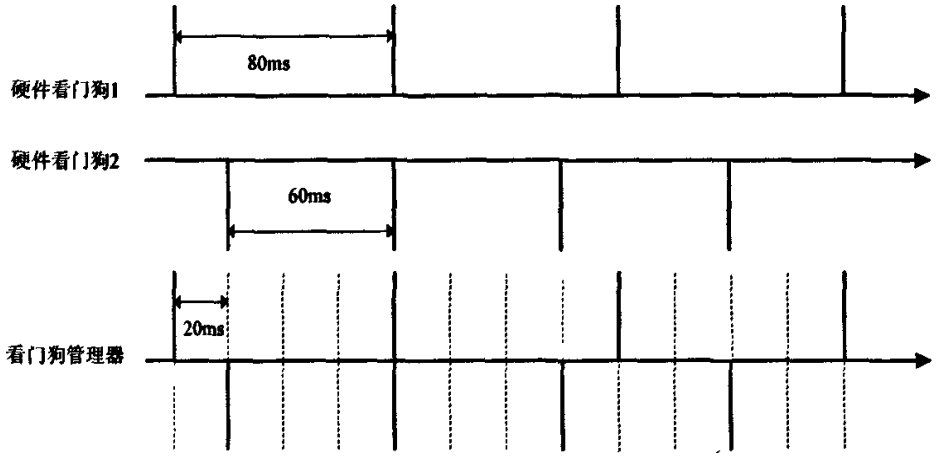

图 5-9 硬件看门狗触发示意

看门狗管理器根据全局监控状态判断是否触发硬件看门狗。仅当全局监控状态为正常、失效和超时状态是看门狗管理器开触发硬件看门狗，当全局监控状态为停止时，看门狗管理器停止触发硬件看门狗。硬件看门狗的周期性触发由操作系统周期性调用看门狗管理器的特定系统服务实现。

5.3 AUTOSAR 通信服务

通讯服务利用通信硬件抽象层提供的通信驱动程序提供车载网络通信服务(包括：CAN, LIN, FlexRay 和 MOST)，该层的任务有：

- 为不同应用程序利用车载网络进行通信提供统一的接口
- 为网络管理提供统一的服务接口
- 为诊断通信提供统一的车载网络接口
- 对应用程序隐藏不同协议和消息的差异

SmartOSEK OS 3.0 的 AUTOSAR 通信服务包括了 AUTOSAR 通信服务的

CAN 部分。包括通信、通信管理、PDU 路由、IPDU 多路复用器和 CAN 传输协议。

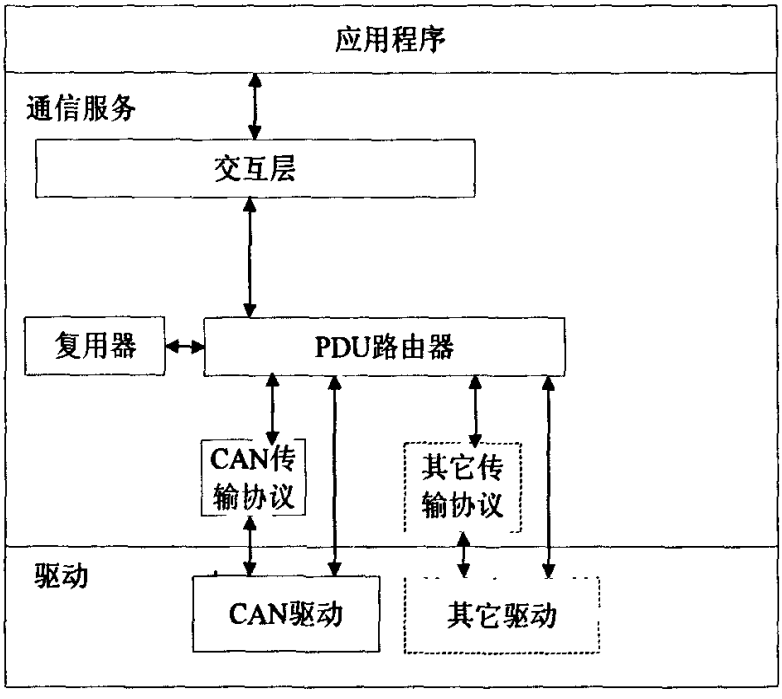

图 5-10 AUTOSAR 通信服务

5.3.1 AUTOSAR 通信交互层

交互层向应用程序提供基于消息的通信服务[31]，AUTOSAR 通信交互层以 OSEK 通信[27]为基础。AUTOSAR 通信定义了一个交互层，交互层根据用户配置把消息封装在交互层协议数据单元（I-PDU）中，然后以 I-PDU 为单位进行消息的收发。在发送端，交互层把应用程序发送的一个或多个消息写入一个 I-PDU，交给底层 PDU 路由器进行发送；在接受端，交互层从 PDU 路由器接受协议数据单元并把其中的消息分发给上层应用程序。

交互层提供时间监控功能，分为接收监控和发送监控。接收监控用于在接收端验证消息是否在允许的时间间隔内被接受。发送监控用于在发送端验证发送动作在发出发送请求后给定的时间间隔内是否成功。用户可以根据需求制定各个消息的发送和接收是否使用功能，并可以定义不同的时限长度。

交互层提供了一种灵活配置的通知机制。根据用户的配置，交互层可以向上层应用发出如下几种通知：

- 消息接收成功。该通知发生在交互层正确接收到消息时。
- 消息发送成功。该通知发生在包含该消息的 I-PDU 被成功发送时。
- 消息接收错误。该通知发生在消息接收发生错误时。比如当接收监控发生定时器超时或底层传输发生错误时。
- 消息发送错误。该通知发生在消息发送发生错误时。比如当发送监控发生定时器超时或底层传输发生错误时。

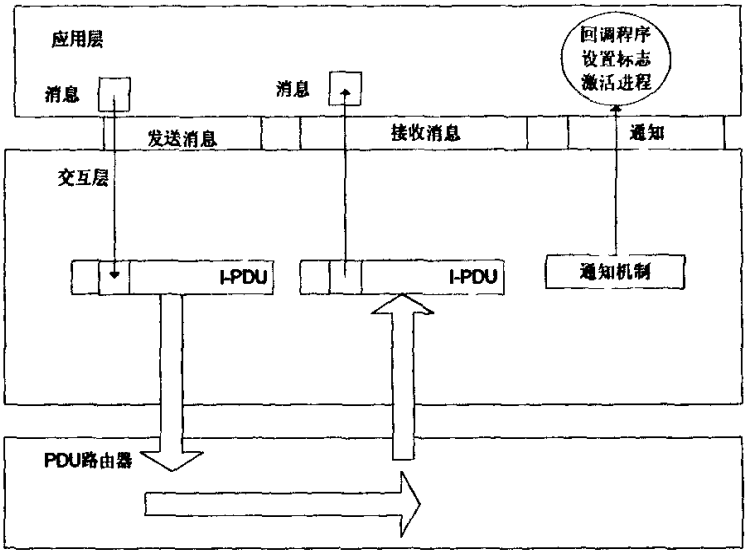

图 5-11 AUTOSAR 通信的交互层模型

5.3.2 PDU 路由器

PDU 路由器[32]在如下模块之间进行 I-PDU 的路由操作。

- 通信驱动模块
- 传输层
- AUTOSAR 通信交互层
- I-PDU 复用器

每一个 I-PDU 由 I-PDU 的 ID 标示，PDU 路由器根据 I-PDU ID 和一张用户配置的表格判断 I-PDU 的目标。I-PDU 路由层之上的信息用 I-PDU 进行交互。I-PDU 路由器不改变 I-PDU，它只是把 I-PDU 发送到指定的模块。PDU 路由器向它的上层和下层都提供了 API。

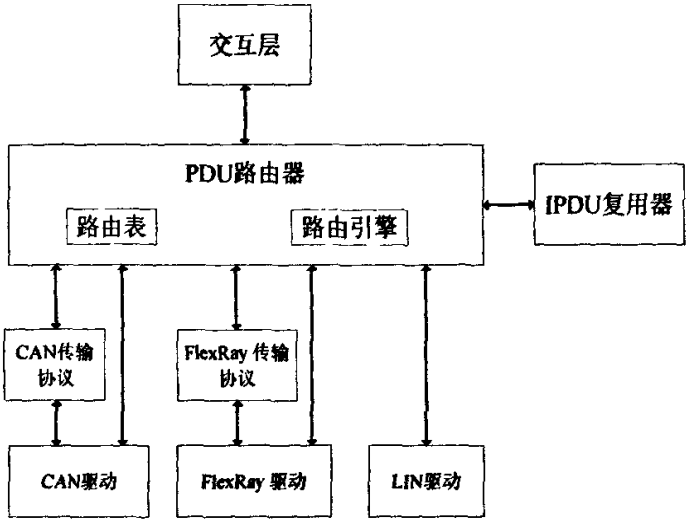

图 5-12 PDU 路由结构

如图 PDU 路由器由路由表和路由引擎两部分组成。I-PDU 路由表是一个静态配置的表，描述了每个 PDU 的路由信息。PDU 路由引擎根据路由表进行路由操作。

5.3.3 I-PDU 复用器

I-PDU 的复用指把多个 I-PDU 中的数据放到同一个 I-PDU 中，使用相同的协议控制信息(PCI)发送。在发送端 I-PDU 多路复用器把从交互层收到的多个 I-PDU 组合成复用的 I-PDU 然后送回 PDU 路由器[33]。在接受端，该模块负责解析复用的 I-PDU 然后向交互层提供独立的 I-PDU。

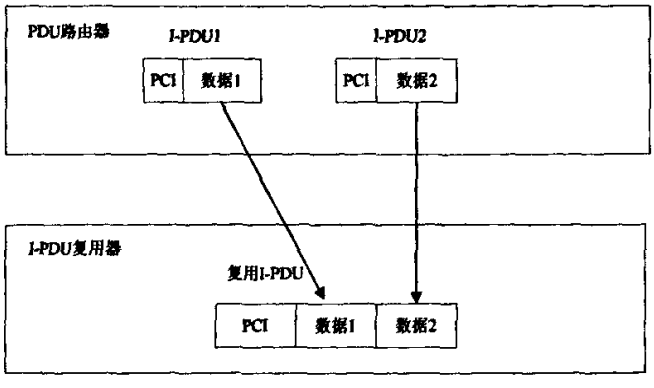

图 5-13 I-PDU 复用示意图

5.3.4 CAN 传输层

CAN 传输层[34]在 PDU 路由器和 CAN 通信驱动之间。CAN 传输层的主要作用是对大于 8 字节的数据进行分段和重组，检测分段的错误并控制数据流。PDU 路由器把 I-PDU 分配到不同的通信协议。CAN 通信驱动提供对于 CAN 控制器的访问。

5.4 本章小结

SmartOSEK OS 3.0 的系统服务层提供内核功能范围以外的系统服务，主要包括符合 AUTOSAR 标准的基础软件模块中的 ECU 状态管理器、看门狗管理器、AUTOSAR 通信服务、TCP/IP 协议栈以及图形用户界面。SmartOSEK OS 3.0 的系统服务层利用开源 TCP/IP 协议栈 uIP 提供 TCP/IP 通信服务，利用开源软件 μC/GUI 提供嵌入式图形用户界面服务。

ECU 状态管理器维护 ECU 的运行、睡眠以及停止状态，并负责在这三个状态间的状态转换。看门狗管理器用来监控应用程序的可靠性。由于采用了层次化架构，该模块使得应用程序的时间约束和硬件看门狗实现松耦合。看门狗管理器基于这种松耦合对硬件看门狗的激活进行抽象，从而提供应用程序的运行监控。

AUTOSAR 通信服务提供了基于多种通信硬件的统一通信服务，该通信服务提供了基于消息的通信模型。主要由 AUTOSAR 交互层、PDU 路由器、CAN 传输层组成。

第6章 工作总结与展望

6.1 工作总结

本文针对 SmartOSEK OS 2.0 的缺点与不足，对其进行改进和扩展，设计实现了 SmartOSEK OS 3.0，其中的主要工作有。

1) 通过调研学习，总结了实时操作系统的基本技术。
2) 突破了 SmartOSEK OS 2.0 的整体式结构设计，使用层次化结构使得 SmartOSEK OS 3.0 易于移植和扩展。
3) 使用层次化内核技术，扩展原有 SmartOSEK OS 使其支持 AUTOSAR 标准和 POSIX 标准。
4) 分析、设计和实现了最新的 AUTOSAR 标准中的系统服务（ECU 状态管理器以及看门狗管理器）和通信服务中的部分功能。

表 6-1 总结了 SmartOSEK OS 3.0 与 SmartOSEK OS 2.0 的不同之处。

6.2 工作展望

由于 SmartOSEK OS 3.0 项目进行的时日有限，所以目前只做了一些基本的工作，难免有一些不足，今后主要还有以下工作可以做：

1) 目前 SmartOSEK OS 3.0 的配置是手工进行的，今后要提供软件工具进行自动化配置。
2) 进一步实现 POSIX1003.13 中规定实时操作系统功能，提高 POSIX 兼容性。
3) 进一步跟踪 AUTOSAR 标准的进展，实现其它模块。
4) 对 SmartOSEK OS 3.0 进行全面功能和性能测试，提高可靠性和稳定性。

表 6-1 SmartOSEK OS 版本比较

	SmartOSEK OS 1.0/2.0	SmartOSEK OS 3.0
支持标准	OSEK/VDX	OSEK/VDX AUTOSAR POSIX
硬件抽象	无	有
调度算法	基于优先级抢占	基于优先级抢占 先进先出 时间片轮转 调度表
同步机制	OSEK 事件	OSEK 事件 POSIX 信号量 POSIX 条件变量 POSIX 互斥量
动态对象管理	不支持	支持
动态内存分配	不支持	支持
系统服务	OSEK 通信	ECU 状态管理器 看门狗管理器 AUTOSAR 通信 TCP/IP 协议栈 图形用户界面

参考文献

[1] Hiroaki Takada. μITRON4.0 Specification Ver. 4.00.00. TRON Association. 2002.

[2] OSEK. OSEK/VDX Binding Specification Version1.4.1. OSEK/VDX Organization. 2003

[3] AUTOSAR GbR. AUTOSAR Technical Overview. AUTOSAR. 2006

[4] Jean J Labrosse, 邵贝贝译. μC/OS 源代码公开的实时嵌入式操作系统. 北京：电力工业出版社,，2001.

[5] Nick Garnett, Jonathan Larmour. eCos Reference Manual. Red Hat, Inc. 2003

[6] 孔祥营, 柏桂枝. 嵌入式实时操作系统 VxWorks 及其开发环境 Tornado. 中国电力出版社. 2002

[7] On-Line Applications Research Corporation. RTEMS C User's Guide Edition 4.6.6. http://www.rtems.com/onlinedocs/releases/rtemsdocs-4.6.6/share/rtems/html/. 2002

[8] Mentor Graphics. NUCLEUS RTOS. http://www.mentor.com/. 2005

[9] QNX Software Systems. QNX Neutrino Realtime Operating System, the embedded industry RTOS standard for building reliable, innovative systems. www.qnx.com. 2006

[10]Enea Embedded Technology. OSE Real-Time Kernel (Data Sheet). www.enea.com

[11]LYNUXWORKS. LynxOS RTOS, the world's most powerful, open-standards real-time OS. http://www.lynuxworks.com/

[12]ESE 工程中心. SmartOSEK OS 简介. ESE 工程中心内部文档

[13]Andrew S Tanenbaum, Albert S Woodhull, 王鹏, 尤晋元等译. 操作系统设计与实现（第二版）. 北京：电子工业出版社. 1998: 27-32

[14]Abraham Silberschatz, Peter B Galvin, Greg Grgne. Operating System Concepts 6th Edition. John Wiley & Sons. 2002: 74-84

[15]Tom Barrett. HOW TO WRITE AN RTOS, A Guide for Anyone Pondering the "Make-or-Buy" Decision. Real-Time Magazine 97-3: 43-45

[16]C.W.Liu and J.W.Layland. Scheduling algorithms for multiprogramming in a hard real time environment. Journal of the ACM, 20(1):46-61, 1973

[17]Andrew S. Tanenbaum, 熊桂喜等译. Computer Networks Third Edition. 清华大学出版社. 1998: 26-28, 158

[18]Adam Dunkels, "Full TCP/IP for 8-Bit Architectures". In Proceedings of the first international conference on mobile applications, systems and services (MOBISYS 2003), San Francisco, May 2003

[19]BOSCH. CAN Specification Version 2.0. Robert Bosch GmbH. 1991:4-6

[20]Greg Haerr. The Nano-X Window System Architecture. http://www.microwindows.org/microwindows_architecture.html. 2000

[21]Feynman. MiniGUI Technology White Paper . Beijing: Feynman Software Technology Co., Ltd. 2006

[22]Portable Applications Standards Committee, The Open Group. Standard for Information Technology—Portable Operating System Interface (POSIX) Base Definitions. Institute of Electrical and Electronics Engineers, Inc. and The Open Group. 2004

[23]Portable Applications Standards Committee, The Open Group. Standard for Information Technology—Portable Operating System Interface (POSIX) System Interfaces. Institute of Electrical and Electronics Engineers, Inc. and The Open Group. 2004

[24]Portable Applications Standards Committee, The Open Group. IEEE Standard for Information Technology-Portable Operating System Interface (POSIX)-Part 1: System Application Program Interface (MI)-Amendment 1: Realtime Extension [C Language]. Institute of Electrical and Electronics Engineers, Inc. and The Open Group. 1994

[25]Portable Applications Standards Committee, The Open Group. IEEE Standard for Information Technology—Standardized Application Environment Profile (AEP)—POSIX® Realtime and Embedded Application Support. Institute of Electrical and Electronics Engineers, Inc. and The Open Group. 2004

[26]OSEK/VDX. OSEK/VDX Operation System Specification 2.2.1. OSEK Group. 2003

[27]OSEK/VDX. OSEK/VDX Communication Specification 3.0.1. OSEK Group. 2003

[28]OSEK/VDX. OSEK/VDX Network Management Concept and Application Programming Interface 2.5.2. OSEK Group. 2003

[29]OSEK/VDX. OSEK/VDX OSEK Run Time Interface(ORTI) Part A: Language Specification 2.2.1. OSEK Group. 2002

[30]OSEK/VDX. OSEK/VDX OSEK Run Time Interface(ORTI) Part B: OSEK Objects and Attributes 2.1. OSEK Group. 2002

[31]AUTOSAR GbR. AUTOSAR Specification of Communication2.0.1. AUTOSAR Administration. 2006

[32]AUTOSAR GbR. AUTOSAR Specification of PDU Router2.0.1. AUTOSAR Administration. 2006

[33]AUTOSAR GbR. AUTOSAR Specification of I-PDU Muliplexer2.0.1. AUTOSAR Administration. 2006

[34]AUTOSAR GbR. AUTOSAR Specification of CAN Transport Layer2.0.1. AUTOSAR Administration. 2006

[35]AUTOSAR GbR. AUTOSAR Specification of ECU State Manager1.0.0. AUTOSAR Administration. 2006

[36]AUTOSAR GbR. AUTOSAR Specification of Watchdog Manager1.0.1. AUTOSAR Administration. 2006

[37]AUTOSAR GbR. AUTOSAR Specification of Operating System2.0.1. AUTOSAR Administration. 2006

[38]严蔚敏, 吴伟民编著. 数据结构：C 语言版. 北京：清华大学出版社, 1996: 198-201

攻读硕士学位期间主要的研究成果

本人攻读硕士学位期间主要在浙江大学 ESE 工程中心内核组参与嵌入式实时操作系统及通信系统的工程开发，并担任内核组组长由一年多时间，期间主要参与了如下工程实践。

1) SmartOSEK OS 2.0 的功能测试和调试，修正了该版本系统中的多处隐藏的错误，极大提高了系统的稳定性。
2) 对 SmartOSEK OS 2.0 在 MPC555 版本上任务管理机制的实现进行优化，极大提高了系统的实时性；增加了栈检测功能。申请了软件著作版权。
3) 实现 OSEK OS 的 CC2 功能，使其一个优先级支持多个任务。
4) 在 SmartOSEK OS 2.0 的基础上实现了 AUTOSAR OS 的调度表功能。
5) SmartOSEK OS 3.0 的架构设计。
6) SmartOSEK OS 3.0 中硬件抽象层以及实时内核层的设计与实现。
7) 组织符合 J1939 标准的 CAN 通信系统 SmartJ1939 开发。
8) 组织 AUTOSAR 系统服务及通信服务的研究、分析以及开发。
9) 组织 TCP/IP 协议栈以及图形用户界面的开发。

本人参与如下专利和软件著作版权的申请：

1) 嵌入式实时操作系统中基于消息对象的任务间通信方法，申请号：200510060732，已受理
2) 基于 PowerPC 体系结构的嵌入式操作系统的中断管理方法，申请号：200510060734.X，已受理
3) 一种基于 CAN 总线的支持三种传输模式的实时通信方法，已提交
4) 一种基于系统描述语言的嵌入式实时操作系统的静态配置方法，已提交
5) 符合 OSEK 标准的嵌入式实时微型操作系统软件 V2.0[简称 SmartOSEK 操作系统] 软著登字第 058021 号 登记号：2006SR10355，已获得软件著作版权。

致谢

本研究成果是在浙江大学计算机学院吴朝晖教授的 CCNT 实验室下属 ESE 工程中心中开发完成的，它属于整个 ESE 工程中心项目的一个重要组成部分。2 年以来，ESE 工程中心为我提供了良好的研究开发环境，而我的工作更是受到吴朝晖教授、赵民德副教授的悉心指导。

再此特别感谢 ESE 工程中心负责人杨国青博士对我的严格要求和热情鼓励，没有他，我不可能取得如此大的进步。

作为内核组的项目组长，我一个人并不能完成如此庞大的研发课题，在这里特别要感谢的是内核项目组的开发人员：苏晓龙、俞建德、周海娟、魏城炯、彭巍、张吕红、张培锋、程云、汪志成，感谢你们一直以来对我工作的不断支持。

还要感谢 ESE 的各位博士，高志刚、李红、郑能干，他们在我的开发过程中给了我很多宝贵的建议。

还有很多实验室同事们都曾经对此项目提出过建议，在此一并感谢。

总之，SmartOSEK OS 3.0 是 ESE 工程中心全体成员共同劳动的成果，希望下一任项目组长俞建德能够继续保持这个成果，让他为整个实验室，乃至浙江大学贡献学术和技术的力量。

郁利吉 于 求是园

2007 年 5 月 17 日