电 子 科 技 大 学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

硕士学位论文

MASTER THESIS

论文题目 **基于多核环境的嵌入式操作系统内核设计与实现**

学科专业 **计算机软件理论**

学　　号 **201121060329**

作者姓名 **张岩**

指导教师 **邱会中　　副教授**

分类号 _____ 密级 _____

UDC [注1] _____

学 位 论 文

基于多核环境的嵌入式操作系统内核设计与实现

（题名和副题名）

张 岩

（作者姓名）

指导教师 邱会中 副教授

电子科技大学 成 都

（姓名、职称、单位名称）

申请学位级别 硕士 学科专业 计算机软件理论

提交论文日期 2014.03 论文答辩日期 2014.05.15

学位授予单位和日期 电子科技大学 2014 年 6 月

答辩委员会主席 _____

评阅人 _____

注 1：注明《国际十进分类法 UDC》的类号。

A DESIGN AND IMPLEMENTATION OF EMBEDDED OPERATING SYSTEM KERNEL BASED ON MULTI-CORE ENVIRONMENT

A Master Thesis Submitted to

University of Electronic Science and Technology of China

Major: **Computer Software And Theory**

Author: **Zhang.Yan**

Advisor: **Prof.Qiu Hui Zhong**

School : **Computer Science And Engineering**

独创性声明

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

作者签名：_____　　日期：　　年　　月　　日

论文使用授权

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

（保密的学位论文在解密后应遵守此规定）

作者签名：_____　　导师签名：_____

日期：　　年　　月　　日

摘　要

信息技术的不断飞速发展带动了整个汽车电子软件行业的开发朝着大型化、多元化发展。这会导致汽车的网络技术、通信技术、零部件控制技术等方面的复杂性持续增加。由于处理器的不断升级，这能够导致不同处理器上系统软件移植难度的提高；不同实时操作系统上的软件，应用接口也不大相同，这也会导致应用软件的一致性问题。1993 年德国汽车工业界提出了 OSEK 体系规范，它能够改变这种由移植性和通用性所带来的问题。1994 年推出的汽车分布式执行规范 VDX 与 OSEK 规范合并，从而形成了今天的 OSEK/VDX 规范。2003 年汽车开发系统架构联盟成立（Automotive Open Systems Architecture, AUTOSAR），以 OSEK/VDX 规范为基础并针对汽车电子提出了一整套软件开发方法、架构技术标准。随着单核瓶颈的出现以及多核技术的深化发展，2011 年 11 月 AUTOSAR 第一次引入了针对汽车电子中的多核嵌入式操作系统标准，开启了汽车电子基础软件的多核时代！

对于符合 OSEK/VDX 标准的单核嵌入式实时操作系统国内外均有一定的研究成果，但是针对最新 AUTOSAR 多核嵌入式实时操作系统标准的研究和开发，在国内却少之又少。正因如此，本文基于 AUTOSAR 最新的操作系统规范标准，以 TI 公司的 8 核 DSP 处理器 TMS320C6678 作为底层硬件平台，设计并实现了一款多核嵌入式操作系统——DeCore MOS。该操作系统基于 DeCore OS，并在其基础之上进行了多核化研究与创新，使其不但能够满足 AUTOSAR 多核操作系统标准，并且能够在多核硬件环境流畅运行，提高了应用执行效率。DeCore MOS 采用多级分层结构，提供诸如多核支持层、内核层、配置管理等以满足高剪裁、高可配置的需要。在满足规范的前提下，系统创新地采用了 RPC 机制，使跨核函数的调用者无须考虑核心间复杂的通信问题。

DeCore MOS 提供了多核调度表、堆栈检测、自由软件定时器、计数器接口，并且包含了 OSEK 所有符合级别的配置。可以说系统符合 AUTOSAR 的 SC1 级别。经过模块化测试、集成测试，DeCore MOS 能够满足一般应用需要并在一定程度上提高了算法执行效率。

关键词： DeCore MOS，AUTOSAR，OSEK，多核操作系统，RPC

ABSTRACT

The rapid development of information technology continues to drive the development of the entire automotive electronics software industry towards large-scale and diversification. The result is in the car parts control technology, the complexity of communication and network technology is greatly increased. Due to escalating processor, for instance, it is resulting in escalating cost of software portability between different processors. Another example, different real-time operating system has its own application program interface; it will cause the issue such as portability of applications. In order to change this situation in portability and versatility, the German automotive industry in 1993 made OSEK system. VDX specification launched in 1994 was merged with OSEK specification forming specifications of OSEK/VDX. Automotive Open Systems Architecture (AUTOSAR) found in 2003, it puts forward a set of software development method in automotive electronics, architecture and technical standards based on specifications of OSEK/VDX. With the emergence of single core processor bottlenecks and deepening development of technology, in November 2011, AUTOSAR group had introduced multi-core embedded operating system standards in automobile electronic for first time, opening the era of multi-core automotive electronic infrastructure software.

There are some researches at home and abroad for single -core embedded real-time operating systems which are complianced with OSEK / VDX standard. But for the latest research of the latest AUTOSAR standard about multi-core embedded real-time operating system are rare. For this reason, based on the latest AUTOSAR specification and TI's TMS320C6678 DSP processor as the underlying hardware platform, I designed and implemented a multi-core embedded operating system which is DeCore MOS. This operating system is based on DeCore OS besides the multi-core research and innovation. It will not only be able to meet the standard AUTOSAR multicore operating system and can be run smoothly on multicore hardware environment improving the efficiency of application execution. DeCore MOS adopts multilevel hierarchical structure, such as multi-core support layer, kernel layer, configuration management and so on in order to

meet the needs of reduction and configuration. Under the premise to meet the specification, the system uses innovative RPC mechanism that allows callers across the different kernel without considering the complex inter- core communication problems.

DeCore MOS provides a schedule table, stack monitor, free software timers, counters interface, and includes all eligible OSEK level configuration. You can say the system meets the SC1 level in AUTOSAR. After modular testing and integration testing, DeCore MOS can meet the needs of the general application and to some extent; it can improve the efficiency of the algorithm to perform.

Keywords: DeCore MOS, AUTOSAR, OSEK, Multicore Operating System, RPC

目 录

第一章 绪 论

1.1 课题研究背景和意义

信息电子的飞速发展和日益深化的社会需求，使得汽车工业在提高系统安全性、提高舒适度、降低能耗等方面的要求不断提高。与此同时，市场的激烈竞争也会导致汽车电子软件产品的复杂化和多样化。简单的 ECU（Electronic Control Unit），诸如车身控制 BCM（Body Control Model）、电子动力转向 EPS（Electrical Power Steering），车载音响、仪表灯的使用已经越来越普及。针对这种实时性和安全性要求较高、需求场景和逻辑场景比较复杂的应用，传统的前后台和非实时处理的模式的弊端就显现出来。并且，不同的汽车电子软件开发商所开发的汽车电子软件有所不同，无论是从功能应用角度，还是从软件模块化角度来讲，几乎无法或难以进行移植和复用。为了解决上述移植性和兼容性所带来的问题，德国汽车工业界于 1993 年联合推出了汽车电子的开放式系统及接口软件规范，即 OSEK（Open System and the Corresponding Interfaces for Automotive Electronics）。汽车生产商、汽车供应商、汽车电子软件开发商以及研究机构是该规范的主要使用者。1994 年，汽车分布式执行标准，即 VDX（vehicle distributed executive）纳入到 OSEK 标准体系当中，两者并称 OSEK/VDX。后者 VDX 最初是由法国独自发起的，后加入到 OSEK 组织当中。目前 OSEK/VDX 标准已经成为了汽车电子行业软件开发的通用标准。

在 OSEK/VDX 标准制定和提出的十年后，汽车开放系统架构 AUTOSAR（Automotive Open Systems Architecture）联盟于 2003 年正式成立，该联盟以 OSEK/VDX 规范作为基础，提出了一整套汽车电子软件的开发方法论和软件架构的标准。规范为汽车电子软件开发领域提供了一个开放的标准化的软件架构，以及一个软件运行时环境（Runtime Environment, RTE），这能够使嵌入式软件独立于硬件平台之外，因此能够在很大程度上促进嵌入式软件的集成复用，从而缩短整个开发周期并提升汽车软件行业的效率。从 2005 年 6 月 28 日发布的初始版本到现在的 9 个年头里，AUTOSAR 先后发布了 5 个主版本号，截止 2013 年 10 月 9 日，最新的 AUTOSAR 标准 5.2.0 为版本。在这些版本中，最为重要的是 2011 年提出的 5.0.0 版本，该版本首次将基于多核环境的嵌入式实时操作系统标准纳入到

AUTOSAR 操作系统规范当中，为在多核处理器上开发嵌入式实时操作系统提供了详细、明确的技术标准和体系架构。可以认为该多核标准的出现开启了针对汽车电子嵌入式操作系统的多核时代。

目前，计算机行业中的多核处理器已经取得非常广泛的应用。但是汽车电子中所使用的 ECU 大多仍然还是传统的单核结构。不过多核处理器的出现也必然会在汽车行业的应用中得到快速的应用和发展。汽车上的动力和安全系统目前已经率先采用了多核处理器，其他子系统也正在逐渐朝着多核架构转变。

我们知道在多核处理器下开发嵌入式应用软件，如果没有一个支持多核硬件系统的操作系统，将会是一件非常繁琐和复杂低效的工作，因此为了更好的利用多核硬件资源，研究、设计和开发一款即能够符合 AUTOSAR 操作系统标准，又能够支持多核硬件体系结构，这将会是一件极其必要的工作！

1.2 国内外的发展现状

针对单核嵌入式实时操作系统来说，目前国内外有关单核的嵌入式实时操作系统的研究已经取得了相当满意的成果。无论从强实时调度算法的理论研究[1][2][3]、高效的内存管理[4]到工程实践，甚至到嵌入式实时操作系统的商品化，这早已经形成了一条完整的产业链。就国内外成熟的商用单核嵌入式实时操作系统来讲，主流的有风河公司的 VxWorks、Micrium 公司的μC/OS 系列、QNX、开源 T-Kernel 等，国内的有 HOPEN、Deltaos、Rt-thread 等。这些商用或开源的嵌入式单核实时操作系统已经相当成熟和广泛应用了。

针对多核嵌入式实时操作系统来说，目前国外的研究较为深入和广泛，国内仍处于相对的劣势，甚至处于起步阶段。与此同时国内支持多核的嵌入式实时操作系统更是少之又少。主流的支持多核嵌入式实时操作系统的有风河公司的 VxWorks 6.0+[5]，T-Kernel 多核版，QNX，以及美国军方使用的 LynuxWorks，该操作系统大量用于美国空军（如 F22 日志系统）和海军武器控制系统 WCS 等。开发多核嵌入式操作系统主流的有两种方式，其一是“无中生有”，即不依赖任何开源单核操作系统基础框架，完全重新构建一个支持多核的嵌入式操作系统，比如电子科技大学自主研发的 aCoral[6]；其二是依赖原有单核操作系统，在其既定框架之下进行“多核化改造”，比如依赖单核μC/OS-iii[7]，对其任务调度，消息队列，中断系统等进行改造和翻新，再比如浙江大学的 SmartOSEK OS-M[8]，依赖于单核 SmartOSEK OS[9]，并在其基础之上进行多核化改造。除去使用国外商用多核嵌入

式操作系统之外，目前国内较多采用第二种方式。

针对符合 OSEK/VDX 操作系统规范的单核嵌入式操作系统，在国外已经发展了近 20 几年，自从 1994 年 OSEK/VDX 规范被提出以来，各类开源或商用操作系统层出不穷，大部分汽车生产商也针对该规范定制和开发了自己的嵌入式操作系统，并随着规范的完善而完善。其中德国 3soft 公司推出的全世界最早的 OSEK/VDX 操作系统 ProOSEK，是最为有名的。它为宝马、奥迪、克莱斯勒等提供了基于 OSEK 的软件开发平台。飞思卡尔公司的 OSEKTurbo 是使用最广泛的实时操作系统之一。它几乎在整个业界处于领先地位，并完整地实现了 OSEK/VDX 标准。同时能够支持不同位数的微处理器。稳定性和软件质量表现异常优异。

OSEK Extension for μC/OS-ii 是 Embedded Office 公司开发的。它基于开源的 μC/OS-ii 实时操作系统，通过了一系列官方认证。OSEK Works 是 WindRiver 公司在 VxWorks 基础之上所开发的嵌入式实时操作系统。这两款操作系统都是在原有基础上进行了标准化改造而成。因而具有各家操作系统所独有的特殊机制。

德国 Vector 公司的 osCAN，是一款具有能够结合任意该公司的通信协议、支持多 CAN 协议、运行时堆栈管理、内部跟踪、模板生成器等功能的操作系统。同时它也是目前最具有影响力的实时操作系统之一。

放眼国内，浙江大学嵌入式软件工程中心的 SmartOSEK OS 在 2005 年通过了 OSEK/VDX 组织的官方认证，支持若干国际主流处理器。它具有微内核、高实时性，多种实时调度机制等优点。

普华公司的 ORIENTALS-OS 是一款商用操作系统，它的运行耗费时间很短，配置非常灵活。同时中断处理和屏蔽时间也极短，任务切换速度快。具备良好性能的同时，也不弱于剪裁性和移植性。

北京科银京成公司与电子科技大学 ESE 中心共同开发的 DeltaOSEK 提供了标准的 OS 及 COM 功能部件的应用编程接口。它符合 OSEK/VDX 标准，并于 2009 年获得了 OSEK/VDX 国际认证机构的官方认证。它是一款国内为数不多的在 MPC555/MPC5554 等平台上通过了 OSEK/VDX 测试集的全面测试的操作系统。

上述的符合 OSEK/VDX 标准的嵌入式实时操作系统都是基于单核环境的，而目前针对于多核环境下，并符合最新 AUTOSAR 规范的嵌入式实时操作系统在国内几乎为空白，目前国内只有浙江大学的 SmartOSEK OS-M。而该操作系统是基于 SmartOSEK OS 改造，严格意义上来说其并不符合 AUTOSAR 的多核标准，其上开发的应用并不能方便的移植到 AUTOSAR 平台上，只能算是 OSEK/VDX 的多核化再加工而已。

因此研究和开发一款符合 AUTOSAR 多核标准的嵌入式实时操作系统，是具有一定的理论和实际意义的。本文正是基于这样的目的，以 AUTOSAR 5.2.0 多核嵌入式实时操作系统规范为蓝本；以 TI 公司的 8 核 DSP 处理器 TMS320C6678 作为底层硬件平台。通过改造已获得 OSEK/VDX 认证机构认可，并经过全面测试的 DeCore OS（电子科技大学 ESE 中心自主研发），使该操作系统能够迈入汽车电子多核嵌入式实时操作系统的大门。DeCore MOS 正是对 DeCore OS 的全面改造和升华，在规范之内能够达到 SC1 剪裁级别，在规范之外又具有的一定的创新性，这也正是 AUTOSAR 操作系统规范所带来的灵活性之一。

1.3 课题研究的主要内容

本文主要参考 OSEK/VDX 操作系统标准以及 AUTOSAR 有关多核部分的操作系统标准，并根据其相关内容在 TI 公司的 TMS320C6678 硬件平台上改造并重新设计实现 DeCore OS，使之成为符合 AUTOSAR 多核标准的多核嵌入式操作系统——DeCore MOS，并在一定程度上进行创新。因此本文主要研究内容如下：

（1） OSEK/VDX 有关单核嵌入式实时操作系统的理论与架构设计。

（2） AUTOSAR 最新多核操作系统规范，研究其规范背后的理论意义和实际意义，并进行规范外的创新。

（3） 研究 TMS320C6678 平台关于多核驱动层部分，并设计针对 DeCore OS 的多核支持层。

（4） 研究已经过 OSEK/VDX 机构认可，并经过全面测试的操作系统 DeCore OS 的设计和实现。

（5） 研究多核化改造的关键技术，如多核启动、核间通信、自旋锁等。

1.4 本文的组织结构

第一章为绪论，主要介绍课题的研究背景和意义，以及目前国内外汽车电子嵌入式操作系统发展现状，并给出课题的研究内容和本文的组织结构。

第二章将会介绍 DeCore MOS 的技术基础，其中包括 OSEK/VDX 和 AUTOSAR 规范中针对单核操作系统部分的最主要的内容，以及 DeCore OS 操作系统和多核环境的硬件平台。这些内容正是 DeCore OS 的单核技术架构和实现。

第三章将会介绍 DeCore MOS 的整体设计与创新。从基础软件开发的全局模

块化角度考虑如何改造、设计以及创新。

第四章将会介绍 DeCore MOS 的多核支持层，它包括中断子系统、自旋锁、RPC 机制等底层的设计实现。

第五章将会详细说明 DeCore MOS 各个模块的设计与创新，其中包括任务调度、多核调度表、计数报警器等内容。

第六章讨论 DeCore MOS 的配置与测试。其中包括 DeCore Turbo 配置工具的使用以及对 DeCore MOS 主要模块进行的模块测试。

第七章对目前工作做一个全面的总结，并提出未来的工作计划和目标。

第二章 DeCore MOS 的技术基础

DeCore MOS 是 DeCore OS 的多核化版本，而 DeCore OS 是符合 OSEK/VDX 标准的单核嵌入式实时操作系统，因此本章作为 DeCore MOS 的技术基础，首先介绍 OSEK/VDX 和 AUTOSAR 标准的主要组成部分和技术架构，然后介绍单核环境下的 DeCore OS 的设计与实现，最后我们简单介绍一下 DeCore MOS 的多核硬件环境。

2.1 OSEK/VDX 规范简介

控制装置与机械系统紧密配合使用是属于汽车电子的控制系统部分，比如发动机电子燃油喷射、ABS 防抱死系统、汽车底盘等；车载上网设备，视听娱乐等是属于车载电子部分。以上的分类是按照汽车行驶的机能作用的影响范围。车载电子系统与汽车性能并没有直接上的关系，而控制系统产品是分布在汽车不同位置的电控单元 ECU、智能传感器 SS（Smart Sensor）等功能电子元件组成的。这些器件通过总线连接如 Lin、J1939、CAN，从而组成一个个不同功能的子系统。同时德国汽车工业界联合推出的 OSEK/VDX 标准规范能够解决不同厂商控制模块间的兼容性问题，这使得嵌入式系统软件和应用软件能够满足日益庞大复杂的汽车电控系统的开发需要。

规范标准包括以下四个部分：OSEK/VDX 操作系统规范（OSEK Operating System，OSEK OS）、OSEK/VDX 通信规范（OSEK Communication，OSEK COM）、OSEK/VDX 网络管理规范（OSEK Network Management）以及 OSEK/VDX 实现语言（OSEK Implementation Language，OSEK OIL）。采用符合 OSEK/VDX 标准的嵌入式实时操作系统能够提高代码复用率、降低成本并且缩短开发的时间周期。以上的四个部分分别构成了独立的规范实体，并且它们之间也具有一定的依赖和层次架构关系：其中网络管理规范与通信规范都是依赖于 OSEK/VDX 操作系统 API 规范；通信规范的具体实现也可以直接建立在硬件层的基础之上以提供系统服务接口，关于 OSEK/VDX 的架构图如下图 2-1 所示。

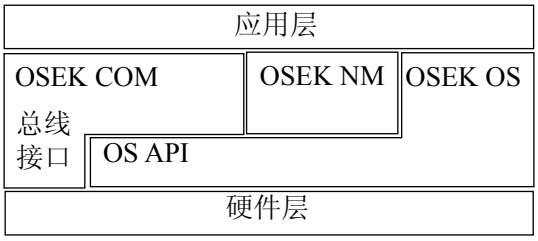

图 2-1 兼容 OSEK/VDX 规范的操作系统应用架构

OSEK COM 规范为汽车 ECU 软件提供了统一的通信环境。为了满足任务与任务之间、ISR 与 ISR 之间以及任务与 ISR 之间的消息发送便捷性，OSEK COM 提供了多种服务。该规范的目的是支持应用软件的一致性、重用性和相互合作性。应用程序接口隐藏了内外部通信区别，也隐藏了不同协议、总线系统和网络。

OSEK COM 通信是基于消息的。消息和消息属性能够通过 OSEK OIL 进行静态配置。消息内容和使用方式与 OSEK COM 无关，并允许长度为 0 的消息存在。在 OSEK COM 规范中交互层 IL（Interaction Layer）起着至关重要的作用，他负责处理内部与外部通信。在内部通信情况下，其使得消息数据立即发送到接受方，而当处于外部通信情况下，IL 层将一个或一批消息压缩成交互层协议数据单元 IPDU（Interaction Protocol Date Unit），并将其传递到下层处理。交互层与下层通信的数据被组织成 IPDUs，包括一个或多个消息。一个消息占据连续的位，不能被分离，也就是构成一个完成的消息数据实体。底层将这些实体封装为协议数据单元，并且同步或异步地传送到消息接收方的消息队列，当然在这个交互过程中，IL 层能够对接受到的数据进行消息提取。

不同厂商的 ECU 产品，能够通过数据交互链接成拓扑网络，OSEK NM 规范则提供了一个该网络连接的规范标准。其不但能够降低成本和缩短开发周期，而且可以提高产品质量和安全可靠性等。规范包括了节点监控算法、与 COM 的接口、协议数据单元等。这些组成部分构成了整个 OSEK NM 基本内容。除此之外，OSEK NM 还能够支持网络诊断功能。

OSEK 操作系统是基于配置的，OSEK OIL 则提供特定 CPU 中配置 OSEK 应用的一种机制。它将 OIL 的描述划分为一组对象集合，其中 CPU 是对象集合的容器。与此同时 OIL 明确定义了每一个 OIL 组件的标准属性。每个 OSEK 应用也能够自定义特殊执行属性以及引用，还可以限制属性的取值范围。处理器、操作系统、应用模式、中断服务、资源、任务、计数器、事件、报警、通信子系统、消息、交互层协议数据单元以及网络管理都能够成为 OIL 的对象。通过 OIL 能够可

靠并方便的进行系统配置和剪裁操作。

通过图 2-1 可知，OSEK OS 规范处于核心地位，应用需要建立在 OSEK 操作系统之上，通过相关 API 来进行网络通信与消息传递，通过 OIL 进行操作系统移植性配置。并且 DeCore MOS 的多核化相关工作也是建立在 OSEK OS 规范的基础之上展开的，因此关于 OSEK OS 规范的详细介绍是必不可少的。为了节约篇幅，在后文将会通过介绍 DeCore OS 的设计与架构来对该规范进行讨论。

2.2 AUTOSAR 规范简介

2.2.1 AUTOSAR 软件架构

在 2003 年，汽车开发系统架构联盟以 OSEK/VDX 规范为基础提出了一整套汽车电子软件开发方法和软件架构标准，它为汽车电子软件开发领域提供开放的、标准化的软件架构。AUTOSAR 操作系统规范以 OSEK/VDX 作为基础，因此符合 OSEK 规应用软件也能够非常轻松地移植到 AUTOSAR 平台上。并且在其基础上增加了调度表模块、堆栈监测模块、OS-应用、安全相关模块等。规范含软件架构（AUTOSAR architecture）[10]、方法论（AUTOSAR methodology）[11]和应用接口（AUTOSAR application interfaces）[12]。自 1.0 版本发布以来，截止目前的最新版本是 5.2.0。其中 AUTOSAR 的操作系统规范属于软件架构部分内容。因此这里重点针对该部分内容做一定的介绍。

为了将应用与具体的 ECU 和微控制器相互隔离，AUTOSAR 软件架构采用了四层架构模型，从上到下分为：Application Layer（应用层）、Runtime Environment(运行时环境，RTE)、Basic Software（基础软件层，BSW）、Microcontroller（微控制器层）。他们的层次关系如图 2-2 所示。

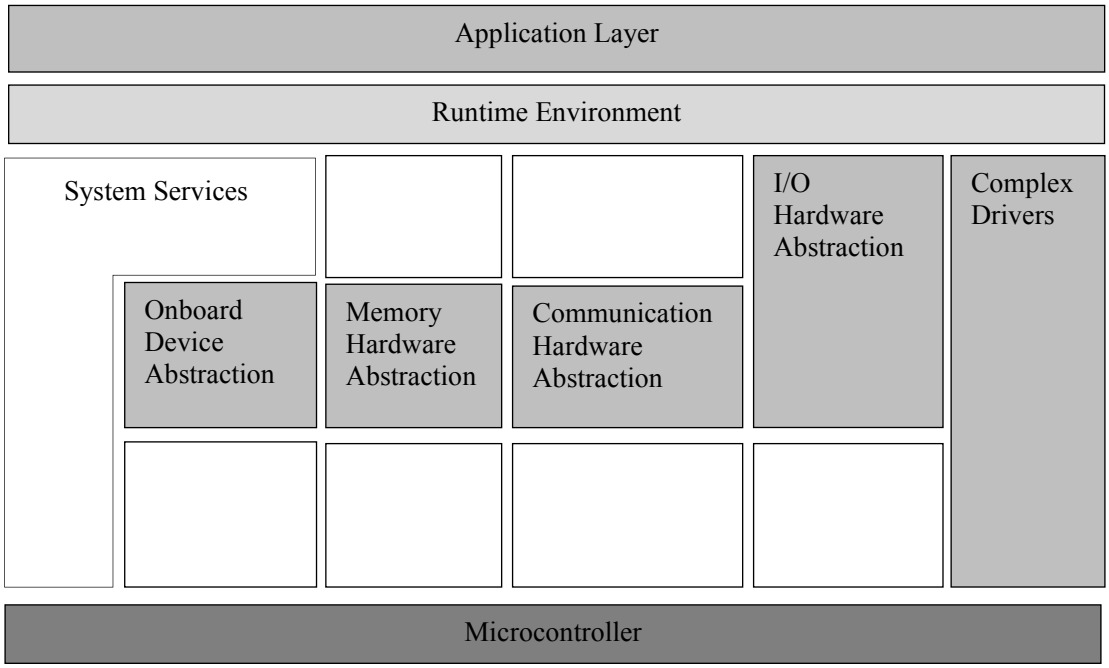

图 2-2 AUTOSAR 软件架构图

处于整个架构最上层位置的是应用层，该层将软件划分为一个原子软件组件（Atomic Software component，ASWC），包括硬件无关的应用软件组件（Application Software Component）、传感器软件组件（Sensor Software Component）等。之所以叫“原子”是因为这些软件组件不可再进行分割。同时这些组件能够通过虚拟功能总线（Virtual Functional Bus，VFB）进行交互与消息通信。该层是最贴近于开发人员和实际需求的一个层次，基于 AUTOSAR 汽车电子应用软件都是在该层开发完成的，一般来讲可以通过建模工具进行算法建模来生成应用，或者直接编写应用代码来完成应用的开发。

第二层为 RTE 层，即 AUTOSAR 运行时环境。RTE 提供基础的通信服务，支持 SW-C 之间和 SW-C 到 BSW 的通信（包括 ECU 内部的程序调用、ECU 外部的总线通信等情况）。AUTOSAR 应用软件之所以能够独立于特定硬件、实现软件复用，RTE 层起到了至关重要的作用。

第三层为基础软件层（Basic Software，BSW），它处于整个架构系统的核心位置。该层分为服务(Services)、ECU 抽象（ECU Abstraction）、微控制器抽象（Microcontroller Abstraction）以及复杂驱动（Complex Drivers）。在该 BSW 层当中，上述四个部分又进行了层次划分。处于外部的服务是 BSW 中的最上层，它包括系统服务、内存服务以及通信服务等。其中的系统服务包括 RTOS、定时器、错

误处理、看门狗、状态管理等。可以看到 DeCore MOS 操作系统正是处于该层；BSW 的第二层为 ECU 抽象层，该层封装了微控制器层以及外围设备的驱动，将微控制器内外设的访问进行了统一，使上层软件应用与 ECU 硬件相剥离。它对上层服务用到的底层设备进行抽象，如内存硬件抽象、通信硬件抽象等；处于 BSW 最下层的是微控制器抽象，该抽象包含了访问微控制器的驱动，分离了上层软件与微控制器，以方便应用的移植；复杂驱动并不属于 BSW 内部的分层结构，可以利用复杂驱动，让应用层通过 RTE 直接访问硬件而跨过操作系统服务。也可以利用复杂驱动的特性封装已有的非分层软件，以实现向 AUTOSAR 软件架构的逐步实施。

采用这种软件分层模型，对于汽车电子应用程序的开发来讲，能够做到应用与 MCU 类型的无关性、与 ECU 类型的无关性、与相互关联的 SW-C 位置的无关性以及 SW-C 实例个数的无关性。可以看到，DeCore MOS 操作系统正是处于核心层 BSW 中的最上层，这也意味着符合 AUTOSAR 软件架构标准的操作系统地位的重要性。

2.2.2 AUTOSAR 对多核硬件的要求和假定

由于针对不同厂商生产的 MCU 不同，操作系统所依赖的多核硬件环境则会有所不同。因此 AUTOSAR OS 规范对其多核环境进行了一定的要求和假定，所有的 AUTOSAR 操作系统都需要遵循这些要求。

SMP 和 AMP

SMP（Symmetric Multi-processing）是对称多处理器的简称[13]，它是指在一个系统中汇集了多个 CPU 处理器，各 CPU 之间共享内存和总线系统等。从操作系统角度来看，所有的 CPU 具有相同的指令集，都是对称的，所有处理器都公用一套操作系统，因此操作系统在 SMP 架构下需要均衡各 CPU 的负载、管理每个 CPU 的工作。AMP（Asynchronous Multi-processing）是非对称多处理器的简称[14]，它是指系统中具有两种或两种以上的 CPU 架构，这些 CPU 的指令集可能不同，也可能不共享总线系统，如 ARM+DSP 架构等。从操作系统的角度来看，由于指令集的不同，可能存在多套操作系统镜像，每个操作系统独立的管理所在的处理器，任务不能在异构核上进行迁移。

目前支持 SMP 架构的操作系统较多，如 Linux、VxWorks、T-Kernel 等。

AUTOSAR 规范没有明确的限定操作系统是否需要支持 AMP 或 SMP，但明确定义了当硬件环境为多核处理器时（SMP），则只需一套操作系统镜像；当硬件环境为多处理器（单处理器可能为多核也可能是单核）的情况下必须使用多套操作系统镜像。

CPU 特性要求

最新的 AUTOSAR OS 标准中关于多核硬件环境中处理器部分，明确定义了 9 条关键性限定，如表 2-1 所列。

表 2-1 AUTOSAR OS 的 CPU 特性要求表

编号	说明
1	同一片处理器中需要有多个核心。
2	硬件环境需要提供获得核心编号的机制。
3	需要支持原子操作。
4	需要提供 Test-And-Set 机制来构建临界区的互斥访问。
5	每个核心必须有相同的指令集。
6	每个核心需要有相同的数据格式，如相同的整形位数。
7	需要提供缓存机制，如清空缓存等操作。
8	每个核心都需要能够处理非法异常，如除 0 异常等。
9	每个核心都应该支持中断触发

通过上表就能够确定 DeCore MOS 所在的硬件环境，它需要提供相同指令集，并且能够独立操作每个核心的缓存、支持异常处理、原子操作等。

内存特性要求

关于内存特性，标准中定义了 5 条限定，如下表 2-2 所示。

表 2-2 AUTOSAR OS 的内存特性要求表

编号	说明
1	共享 RAM 能够被所有的核心访问。
2	Flash（闪存）能够被所有核心共享访问。
3	至少在共享内存部分，需要使用单地址空间。
4	能够在兼容内存保护或不提供能存保护的系统中运行。
5	操作系统不应该提供动态内存管理。

从上表中可以看到，由于使用内存保护机制或使用 VMM（Virtual Memory Management）需要硬件 MMU 的支持，因此规范并没有明确规定是否需要支持，这可以作为系统的扩展以及可选的特性。并且规范明确规定了不应该使用动态内

存管理。为了内核编程需要，DeCore MOS 能够支持多核环境下的静态内存管理（包括私有和共享内存区管理）。

多核限制

在 AUTOSAR 规范中规定：当系统已经启动运行时，不能动态激活没有启动的核心，这也就意味着在多核硬件环境下，启动核心的数目对于应用来讲只能通过静态配置进行指定。并且调度算法不支持动态分配任务到其他核心上运行，因此 DeCore MOS 没有 CPU 亲和性概念和负载均衡的问题。这么做的优点是避免了复杂的负载均衡算法[21]和系统运行中的不确定性。规范中还规定资源算法（Resource Algorithm）不支持跨核使用，因此资源只能在本地使用，诸如多个核心上的不同任务不能使用彼此的中断资源、调度器资源等。DeCore MOS 完全符合以上的 AUTOSAR 的多核限制，但这并不意味着不同核心上的任务之间毫无关系，它们能够通过核间通信、共享内存等彼此进行动态交互。

2.3 嵌入式实时操作系统 DeCore OS

上文分别介绍了 OSEK/VDX 标准和 AUTOSAR 标准的层次结构和基本概念，这些概念和层次结构对于认识嵌入式实时操作系统在汽车电子中所处的位置、功能和特性有着“灯塔”般的作用。DeCore OS 正是参考 OSEK/VDX 规范所开发的单核环境嵌入式实时操作系统，本节将重点介绍 DeCore OS 中各主要功能模块。这些内容对于 DeCore MOS 来说是其最重要的技术基础。

2.3.1 任务管理与符合级别

DeCore OS 的任务管理能力有限，这是因为 OSEK 族操作系统属于静态实时操作系统。它的任务数目、优先级、调度策略等已经在编译期确定了，这能够保证汽车电子软件所要求的实时性和确定性。DeCore OS 的任务分为两种，其一是基本任务；其二是扩展任务。他们的区别如下表 2-3 所示。

表 2-3 基本任务与扩展任务的区别

	基本任务	扩展任务
状态模型	Ready、Running、Suspend	在基本任务上增加 Wait 状态
符合级别	BCC1、BCC2、ECC1、ECC2	ECC1、ECC2
抢占机制	高优先级抢占	支持事件机制

对于其状态模型，基本任务只包括 Ready 状态、Running 状态以及 Suspend 状态。状态转换如下图 2-3。

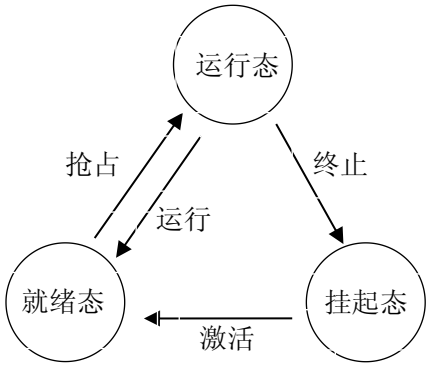

图 2-3 基本任务状态模型

对于扩展任务，它能够支持事件机制，当等待事件发生的时候，任务将会进入等待状态，当所等待的事件发生时，将会切换到就绪状态，并重新等待调度器进行下一轮的任务调度。扩展任务的基本状态如图 2-4 所示。

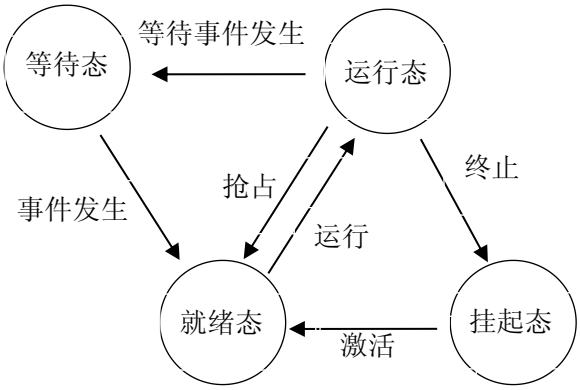

图 2-4 扩展任务状态模型

DeCore OS 采用嵌入式实时操作系统最常用的基于优先级的可抢占式调度策略，每一个任务有且仅有一个在编译期就已经确定的优先级。根据不同的符合类

型按需处理相同类型的优先级，如 BCC2 支持相同优先级下有多个任务，相同优先级的任务进入一个 FIFO 队列等待处理；BCC1 则规定一个优先级仅对应一个任务。不同优先级的任务使用优先级位图表进行组织和管理，这也是实时操作系统中最为广泛使用的一种技巧。DeCore OS 将任务划分为可抢占式任务（Preemptive Task）和不可抢占式任务（Non-preemptive Task），对于可抢占式任务，其低优先级任务可以被高优先级任务所抢占，即任务切换可能在任何时候发生。而对于不可抢占式任务，既使具有更高优先级的任务也不能抢占该任务，只有达到其调度点时才发生调度，而一般程序员能够预知其调度点。需要注意的是，DeCore OS 不允许同一个任务并行的调用，这会导致创建新的任务。当请求调用一个已经激活的进程时，该请求会进入一个请求队列，直到前一个激活进程运行终止(转换为挂起态)，该激活请求才会执行。

由于汽车电子领域的应用非常广泛，不同应用程序软件对操作系统的要求有所不同，而且其硬件也存在很大的差异，这就要求操作系统具有灵活配置的能力。OSEK 规范把不同配置的特点分为两个主要的符合级别（Conformance Classes，CC），包括基本符合级别 BCC 和扩展符合级别 ECC。在 BCC 之下又包含 BCC1、BCC2，同样，ECC 之下也包括 ECC1、ECC2。每一种符合级别都有各自的配置要求和特点，见下表 2-2。

表 2-3 符合类别区别对照表

功能 \ 级别	BCC1	ECC1	BCC2	ECC2
是否支持扩展任务	否	是	否	是
一个优先级是否只有一个任务	否	否	是	是
是否支持多个激活请求	否	否	是	是

2.3.2 同步与消息机制

DeCore OS 的同步机制包括对资源的互斥访问机制以及针对扩展任务的事件机制。消息机制建立于同步机制基础之上，为任务提供通信能力。关于资源管理部分的内容将会在下一小节中讨论，本节主要介绍 DeCore OS 的事件和消息机制。

OSEK/VDX 规定只有扩展任务才支持事件机制。从事件机制本身实现来讲，它包括设置事件（SetEvent）、清除事件（ClearEvent）、等待事件（WatiEvent）、获取事件（GetEvent）。其中 GetEvent 用来获得对应扩展任务的事件集合，本身并不

参与事件状态模型的转换，参考下图 2-5。

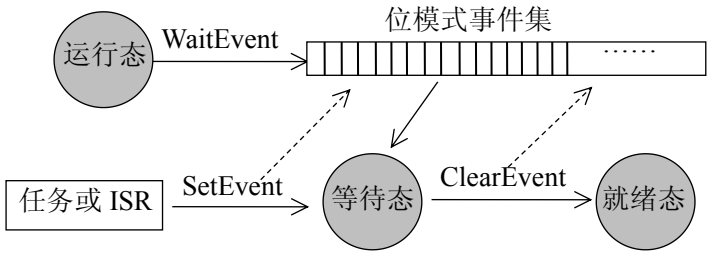

图 2-5 扩展任务事件模型

只有处于运行态的扩展任务才能够调用 WatiEvent 来等待一个事件发生，OSEK 没有规定事件集合的组织方式，因此 DeCore OS 采用最为简单的位模式来组织事件集。目前 DeCore OS 能够支持 32 个事件，即使用一个双字（Double Word）来组织事件。中断服务例程（Interrupt Service Routine，ISR）或任务（包括扩展任务和基本任务）都可以设置事件，如果该任务所等待的一个或多个事件被设置，则任务状态将会转换到就绪态，并且触发重调度。值得注意的是，对于当前正在运行的不可抢占式任务，则不会触发重调度。此时的重调度功能只会发生在该不可抢占任务指定的调度点。ClearEvent 只能由扩展任务本身调用，事件发生时，扩展任务可以清除该事件，也可以重新等待事件的发生。当事件机制与资源管理相互配合使用的时候，必须要保证已获得资源的扩展任务不能等待事件的发生，从而防止死锁（Dead Lock）的出现。

消息机制属于 OSEK COM 规范的内容，该规范已在第一节中做了简单介绍。DeCore OS 中通信部分主要实现操作系统内部的通信，不涉及外部通信，只支持定长消息，符合 OSEK COM 标准的 CCCA 和 CCCB 级别。支持队列和非队列的消息方式，并且支持拷贝和非拷贝的消息使用方式。其中队列方式是指消息不互相覆盖，当消息队列满时到达的消息被丢弃，并且消息的使用方式只支持拷贝方式；非队列方式中新消息覆盖旧消息，同时支持拷贝和非拷贝。为了保证非拷贝方式使用消息时的资源互斥，提供 GetMessageResource 和 ReleaseMessageResource 服务。另外，消息到达的通知机制支持设置事件、激活任务、Callback、设置 flag（标志）等四种方式，其中前面两种需要使用操作系统的服务，后面两种与操作系统无关。一个消息对象只能够使用其中的一种通知方式。

系统服务中不存在创建消息的接口，消息对象都是在系统初始化时根据应用的静态配置确定的，运行过程中不允许创建、删除消息、以及改变消息的属性。

通信管理部分主要包括消息控制结构、队列型消息缓冲区结构以及外部功能

接口等内容。各部分的描述如下：

（1）消息控制结构

消息控制结构包括配置信息部分和运行信息部分，配置信息部分用于获取用户的配置信息，包括消息 ID、指向消息体的指针、消息的大小、消息通知机制的内容、队列型消息的长度等；运行信息用于反映运行过程中消息的信息包括消息状态、队列型消息的读写位置、已有消息数量等。消息控制结构是整个通信管理的核心结构，各种操作都围绕该结构进行。

（2）队列型消息缓冲区结构

主要用于队列型消息对象，采用 FIFO 方式的环形结构。

（3）外部功能接口

外部功能接口主要实现 OSEK COM 标准要求的服务 API，包括发送消息、接收消息、获取消息资源、释放消息资源、获取消息状态、启动通信、初始化通信、关闭通信、停止通信、初始化消息、读取标志和复位标志，这些功能都主要通过对前面的内容进行相关的操作来实现。

2.3.3 资源与中断管理

OSEK 的资源内涵比较丰富，它可以是一段临界区代码、一段共享内存或者是一种数据结构，当然也可以是某种能够共享的硬件设备。系统在多任务环境下对于这些共享资源的访问使用 GetResource，释放资源使用 ReleaseResource，这两个系统服务是 OSEK 规范所规定的标准 API，但如果不进行互斥处理，资源的访问就会出现冲突，导致资源的不一致性。因此 DeCore OS 内部使用信号量机制（该机制属于规范之外的实现）来保证资源的互斥访问，当任务获得某种资源，则对该资源设置信号量值，可以通过信号量初始值来控制资源的互斥访问或多次访问。但是使用信号量机制会产生优先级反转[15]，所谓优先级反转即当一个高优先级任务通过信号量机制访问共享资源时，该信号量已被某一个低优先级任务所占有，而这个低优先级任务在访问共享资源时可能又被其它一些中等优先级任务抢先，因此造成高优先级任务被许多具有较低优先级任务阻塞，实时性难以得到保证。在嵌入式实时操作系统领域中，优先级天花板协议（Priority Ceiling Protocol）[16]和优先级继承协议（Priority Inheritance Protocol）[17]是两种主流的避免优先级反转的算法。OSEK 规范中使用了优先级天花板协议，因此当 DeCore OS 的任务占用某种资源的时候，其任务优先级会被临时提高到该资源所有使用者中最高的优先

级。除了优先级反转问题，使用信号量机制还有可能导致死锁问题[18]。如果使用了优先级天花板协议，那么由于占用资源的任务是最高优先级，试图使用该资源的任务是不会存在的。这也就是说任务试图使用该资源的时候，一定不会有其他任务正在占用资源，自然也就避免了死锁问题。

由于汽车控制系统要求实时性输入能够做出快速反应，在 DeCore OS 中，应用程序开发者编写的中断服务例程与系统封装到一起，这样有利于保护任务和系统的状态。在 OSEK 规范中将中断处理程序分为两类（ISR1 和 ISR2），其一是 ISR 不调用任何系统服务；其二是 ISR 可以调用部分系统服务，如激活进程、设置事件、设置警报等。因此，它可以激活更高优先级的进程。针对这不同类别的中断，DeCore OS 分别提供了 Disable(Enable)AllInterrupts 、Suspend(Resume)AllInterrupts Suspend(Resume)OSInterrupts 来管理所有中断以及第二类中断，并且 DeCore OS 能够支持中断嵌套。

2.3.4 计数报警

一个典型的嵌入式实时操作系统需要有时间管理机制来提供系统的时钟基础。OSEK 系列操作系统都提供了报警服务（Alarm Service）以满足汽车电子控制系统的实时性以及可用性。报警服务依赖于时间机制，在 OSEK 中称为 Counter（计数器）。关于 Counter 的实现，其标准中并没有明确定义，它属于操作系统中的内部实现，也就是说 Counter 的时间源可以是任意的（一般使用底层硬件时钟源），API 也不会提供给用户使用（AUTOSAR 的 SWFRT 提供）。系统中至少具有一个 Counter，被称为 System Counter，其他 Counter 可以由用户自由配置，但所有的 Alarm 服务都要依赖于一个或一组特定的 Counter。DeCore OS 中的 Counter 有两种触发模式，一种是绝对时间触发模式和相对时间触发模式。绝对时间触发模式依赖于系统时钟“tick”的绝对值；相对触发方式依赖于其当前时钟“tick”的相对值。当 Counter 达到触发要求时，则将触发对应 Counter 挂载的 Alarm 线性表中对应警报，使用该警报，用户可以指定需要激活的任务、设置的事件、也可以执行用户编写的报警回调（Alarm Callback）函数。DeCore OS 中除了两种 Counter 触发模式外，也定义了两种报警机制，one-shot 模式和 period 模式。对于 one-shot 模式的报警仅会触发一次；对于 period 模式，报警可以根据用户静态配置的周期数目，循环触发，也可以动态设置其触发周期和周期的时钟值。

对于 Counter 和 Alarm 的架构如下图 2-6 所示。

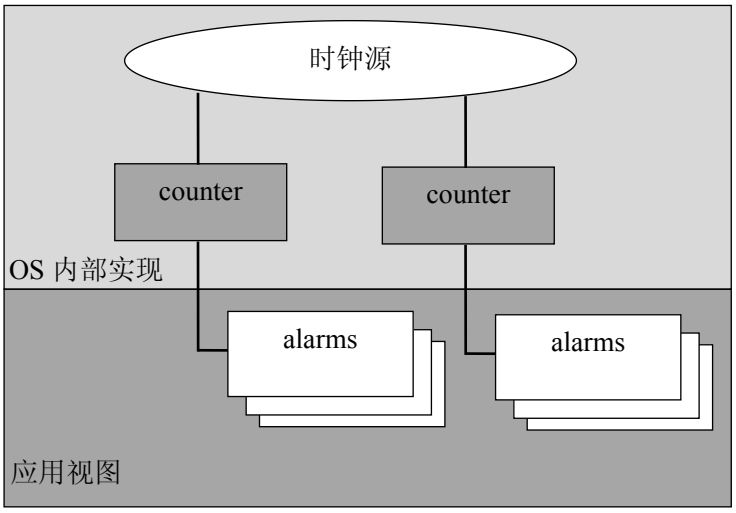

图 2-6 Counter 与 Alarm 架构图

从上图可以很明显的看到，每一个时钟源可以对应多个 Counter，而每一个 Counter 又可以挂载多个 Alarm。并且 Counter 的实现是属于操作系统内部实现。而对于应用视图 Alarms 的实现，则需要符合 OSEK 操作系统 API 规范，以提供标准的编程模型。

2.4 TMS320C6678 多核环境支持

TMS320C6678 是 TI 公司最新的基于 KeyStone[19]多核架构的 DSP（Floating Point Digital Signal Processor ）处理器，它是同时具有支持定点和浮点数处理能力的高级 DSP 内核。TMS32C6678 集成了 8 个 CorePac DSP 核心，每个核的核心频率可达 1.25GHz，并独立拥有 512KB 的 L1-cache 和可共享的 8KB L2-cache。KeyStone 多内核 SoC（System On Chip）架构在业界率先提供了完整的多核性能。KeyStone 可为所有的处理内核、外设、协处理器和 I/O 接口提供非阻塞接入。它具有很多能够充分发挥多内核性能的创新，其中包括：多核导航器（Multicore Navigator）[20]、TeraNet、多内核共享存储器控制器(Multicore Shared Memory Controller，MSMC)[21]和超连结等。

Keystone 的中断系统[22]由两部分组成，芯片中断控制器（INTC）和 DSP 核中断处理控制器（CorePac Interrupt Control）。INTC 共有四个，INTC0 主要负责 Core0 到 Core3，INTC1 主要负责控制 Core4 到 Core7，INTC2 负责控制 EDMA3 的 TPCC1 和 TPCC2，INTC3 负责 EMDA3 的 TPCC0 以及 Hyperlink 的中断。CorePac 内部的中断控制器位于 CorePac 内部，主要负责将外部事件转

换为 CorePac 内部的中断信号，在 DSP 的 Core0 到 Core7 各有自己的核内中断控制器。一般需要关心的是 INTC0 和 INTC1 这两个中断控制器，它们主要将 1024 个系统事件映射为 128 个 CorePac 能够处理的主机事件（Host Interrupt）。DeCore MOS 的驱动层提供了这 128 个中断处理。

每一个 CorePac 都支持三种类型定时器[23]，其一是 64-bit 通用定时器 GP，其二是双 32-bit 定时器，其三是看门狗定时器。DeCore MOS 的配置中所指定的核心都将开启第一种类型的定时器，每一个钟定时器都是通过第 12 号主机事件进行响应，其触发周期也可以通过配置文件静态配置。

CorePac 核之间并不是孤立存在的，它们能够通过共享内存、核间中断（Inter-Processor Interrupt，IPI）机制动态地进行交互。共享内存可以使用硬件信号量作为互斥手段，以一段平坦的内存区作为共享内存工作区，再辅以数据格式协议来实现；当然也可以使用 KeyStone 特有的 MSMC 机制来实现。而 IPI 则提供了更为灵活的机制。所谓 IPI 是一种特殊的中断，该中断是由其他核心发起，以中断指定核心，执行该核心的中断处理器程序。关于 KeyStone 的架构如图 4-7 所示。

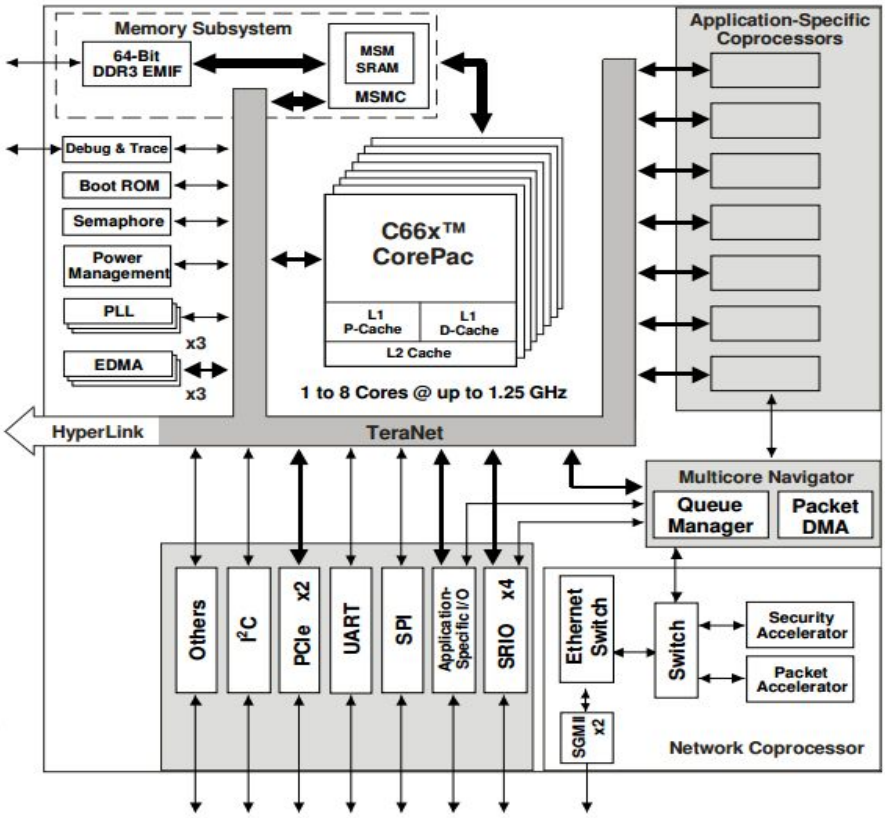

图 2-7 KeyStone 架构图[12]

2.5 本章小结

本章我们重点讨论了 DeCore MOS 的技术基础，其中包括它所遵循的 OSEK/VDX 规范和 AUTOSAR 软件体系结构，因为 DeCore MOS 是符合规范的多核操作系统，我们就需要首先从规范的角度来进行研究和探索。然后我们介绍了 DeCore OS 的各个模块，它作为 DeCore MOS 的单核框架起着重要作用。最后我们又简单介绍了 TMS320C6678 多核硬件支持情况。下一章我们将会介绍 DeCore MOS 的整体设计思路和模块划分架构。

第三章 DeCore MOS 的整体设计

3.1 DeCore MOS 的设计目标

DeCore MOS 的设计目标主要从以下几个系统属性入手：健壮性、可靠性、扩展性、可移植性、可维护性和实时性。

对于健壮性和可靠性来讲，DeCore MOS 的所有 API 都应该进行参数和状态检查，并且为提高效率将检查可分为基本状态和扩展状态检查，其中基本状态主要检查一些必需的内容，而扩展状态则检查更多的内容。对于出错情况，系统应提供一系列 HOOK 机制，使得用户能够干预错误的处理。系统也应该避免运行期资源争用和死锁问题（如持自旋锁的任务不能继续申请资源锁等）。代码编写需要符合规范，未遵循的部分都应有明确合理的文档说明。

DeCore MOS 中对扩展性的考虑主要是针对用户的扩展功能设计，通过 HOOK 机制使用户程序能够介入操作系统的错误处理、任务切换、系统启动和关闭等环节的操作。并且为了支持不同的硬件环境，DeCore MOS 需要采用传统的隔离方式，即增加 HAL 层（Hardware Abstractiong Layer 硬件抽象层），使硬件相关的特性与操作系统内核代码相互隔离。

对于可维护性可以采用约定优先原则。函数、变量、数据结构、宏等定义的地方都采用详细注释的方式进行说明，并且注释风格采用文档注释的形式，能够方便提取参考手册文档。另外，为方便维护，程序各种符号的命名应采用规范的形式。

为了满足 DeCore MOS 的实时性，系统应采用基于优先级位图的可抢占式调度策略，此外 DeCore MOS 还需采用其他一些措施和机制保证实时性，比如在时间管理上采用了时间轮算法，能够减少时钟中断处理中关中断时间。

因此满足上述系统属性的 DeCore MOS 应该具有如下的设计目标：

（1） 良好的代码风格和清晰规范的 API 接口。

（2） 提供可视化配置工具和标准符合级别控制。

（3） 具有硬件抽象层，能够满足一般移植性需求。

（4） 提供高效简捷的多核支持层，提供核间通信机制、自旋锁管理等。

（5） 提供多核环境的计数报警器和事件控制机制。

（6） 提供多核下调度表管理机制。

（7） 满足多核环境的同步启动和同步停止要求。

（8） 提供多核环境下的跨核任务控制机制。

（9） 简捷高效的内存管理机制，以满足多核环境下动态内存分配需求。

3.2 DeCore MOS 的总体架构

DeCore MOS 是满足特定领域的多核嵌入式实时操作系统，它在原有 DeCore OS 的基础之上进行了扩展和多核化改造，它包括硬件抽象层、硬件驱动层、多核支持层、本地基本内核服务层、配置管理层和 API 接口组成。见图 3-1 所示。

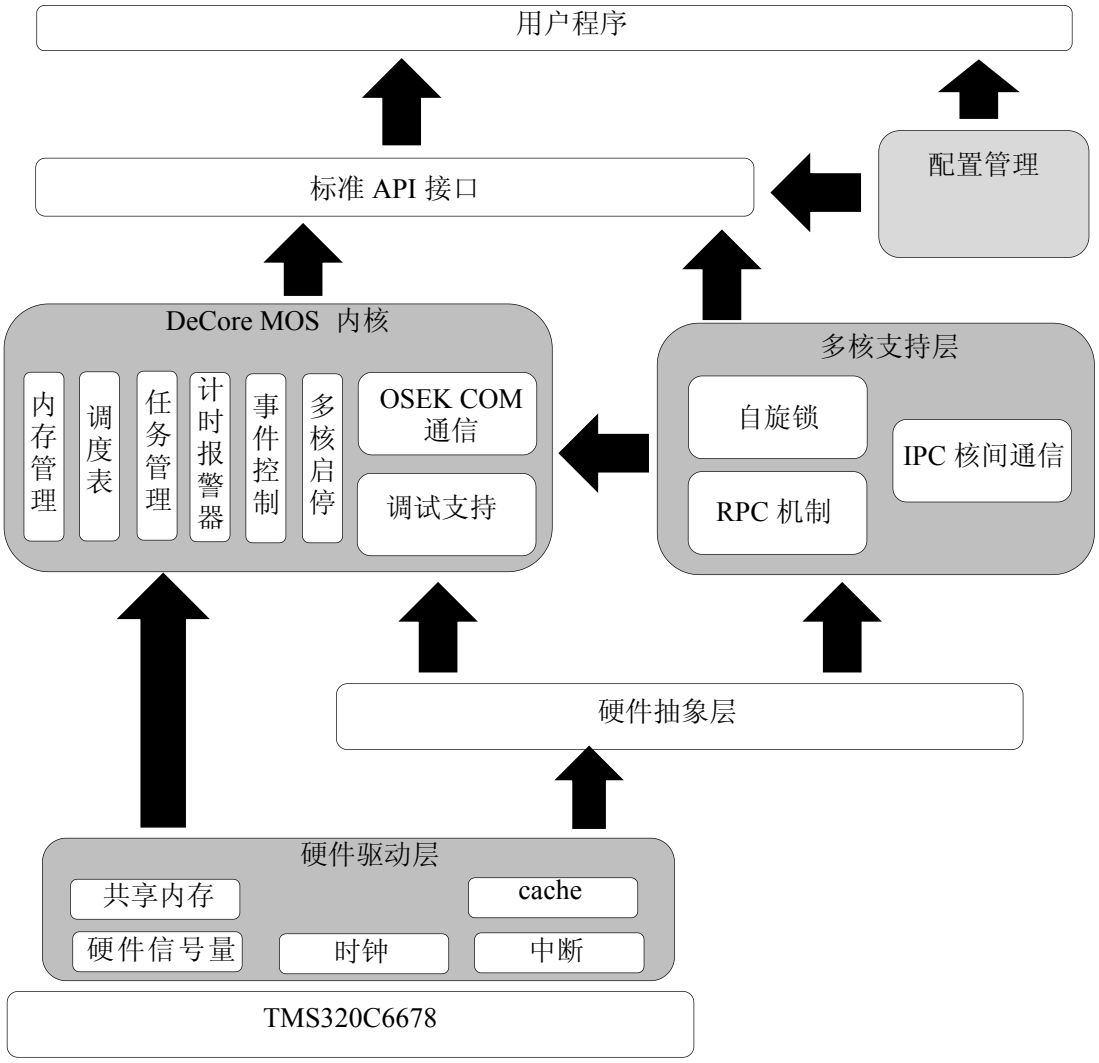

图 3-1 DeCore MOS 系统架构图

DeCore MOS 将底层硬件的功能通过硬件抽象层进行封装。它能够向操作系统内核直接提供硬件机制的使用，如 ISR1 类的中断处理直接使用硬件驱动层接口，并且该层也向硬件抽象层提供必要的操作支持，如将定时器与硬件信号量等进行封装。硬件抽象层的存在主要是为了方便系统移植，DeCore MOS 所使用的硬件抽象层的所有方法或宏定义都使用 BSP_xxx 作为前缀，当硬件体系结构发生了变化，修改的只是硬件驱动层代码，而对整个操作系统并没有任何实质影响。

内核层和多核支持层处于核心地位。由于 DeCore OS 本身只支持单核环境，为了满足多核硬件环境的要求，系统增加了多核支持层。它建立在 IPC 和自旋锁基础之上，自旋锁是多核环境下最为必要的操作之一，因此 DeCore MOS 理所当然也应该支持自旋锁接口。IPC 机制基于核间中断，通过产生核间中断来使得不同核心的任务能够相互交流，使得任务之间不再相互独立。IPC 提供了基本共享内存机制，并且在每个核心单独安排消息队列，当有消息到达该消息队列，则在中断中对该消息进行分发。发送消息方式可以为通知方式和非通知方式，通知方式会产生中断，即中断正在运行的任务。而非通知方式则不产生中断，需要任务单独调用 IPC 的 read 方法来读取本地消息队列内容。RPC 建立在 IPC 机制基础之上，通过指定 RPC 事件、回调代理和服务桩子来模拟函数调用，使得在不破坏原有单核代码的情况下能够支持不同核心之间的跨核函数调用。大多数 DeCore MOS 内核层的机制都是使用了该 RPC 作为实现基础。

内核层是在原有单核 DeCore OS 基础之上通过多核化改造和扩展而形成的。它能够支持调度表、多核技术报警器、事件控制、多核启动停止、任务管理、OSEK COM 通信和简单的固定分区内存管理机制。该层是操作系统最贴近用户代码的一层，因此它的实现效率、错误处理、接口可用性至关重要。除此之外，DeCore MOS 也提供了简单的调试接口和堆栈监测功能。内核层每个机制的设计与实现都离不开多核支持层。无论是多核支持层还是内核层，他们都对用户代码开放了系统调用 API，这些 API 大部分满足 AUTOSAR OS 规范和 OSEK/VDX OSEK COM 所定义的接口规范，有少部分是 DeCore MOS 所特有的，比如 RPC 机制等。

配置管理也是最为重要的一层，用户代码通过配置管理进行配置和剪裁，一般来说配置分为全局配置和本地配置，全局配置可以指定诸如符合级别，功能支持等，而本地配置可以单独配置本地核心内容，比如优先级数目、消息池大小报警器和调度表等。通过配置管理用户能够按照最佳实践和需求剪裁和定制操作系统，再通过操作系统生成器（OS Generator）将用户代码、配置文件、操作系统代

码交叉编译，生成符合需要的操作系统镜像。DeCore MOS 的配置管理提供了一个配置工具 DeCore Turbo，它通过图形化界面满足用户配置剪裁需求。

3.3 本章小结

本章我们介绍了 DeCore MOS 的设计目标，为了多核化 DeCore OS，使其能够满足多核硬件条件并且在一定程度上符合 AUTOSAR 标准，我们就必须对设计目标有一个很清晰的认识。对于 DeCore MOS 来说，最重要的就是要符合代码规范和接口规范，其次是多核功能的满足与实时性的保证。为了方便用户对系统的剪裁，也要提供相应的剪裁工具。

然后我们从整体设计的角度讨论了 DeCore MOS 的系统架构，按照分层的设计，DeCore MOS 由上到下分为内核层，多核支持层，硬件抽象层，硬件驱动层。不同层次负责不同的功能，层次之间又有明确的交互接口。通过分层设计，使得系统能够对下层硬件进行隔离。通过多核支持层的引入，使系统能够在单核操作系统基础上进行多核化；层次模块之间紧内聚，松耦合构成统一的 DeCore MOS 操作系统实体。

从下一章开始，我们将由下层到上层开始，依次介绍每个模块的功能需求和详细设计。

第四章 DeCore MOS 多核支持层详细设计

任何一款操作系统都需要依赖于底层硬件环境。DeCore MOS 是多核嵌入式实时操作系统，因此它的硬件环境又有着其特殊性：既要求支持多核环境（包括自旋锁机制、IPC），又要求严格的实时性（中断）。从上一章我们可以看到 DeCore MOS 的多核支持层属于内核层与硬件层的交互部分，它对上层操作系统内核具体的多核硬件机制进行了抽象，既能够方便内核代码的使用又能够为后期移植工作带来便利。本章我们将会介绍 DeCore MOS 多核支持层的各个模块的设计，其中包括中断、自旋锁、核间通信、RPC 机制。

4.1 中断管理

DeCore MOS 中断服务例程（ISR）分为两种类型。其一是 ISR1，此类中断程序不使用操作系统的资源，中断结束后，处理程序从产生中断的地方继续执行。其对任务的管理没有影响，不要求调用操作系统的 API；其二是 ISR2，此类中断程序是系统生成时，通过用户子程序配置而成，它可以调用操作系统的 API（详细请参考 2.3.3）。在多核环境下，这两种方式依然适用，但对于每一个核心只能触发本地中断（除 IPI 外），也就是说每一个核心都独立地具有一套中断处理函数，其中包括第一类和第二类 ISR。与 DeCore OS 相同的是，所有中断都是可以通过配置表进行配置的。例如如下代码：

```
T_OSEK_TASK_Entry osekConfig_InterruptEntryTable
[CONFIG_OSEK_KERNEL_CORE_NUMS][OCC_ISRLEVELNUM] =
{
    {/***Core 0***/
        0,
        ….
        0,
        BSP_Dolpc,
        BSP_TimeInterruptISR,
        0,
    },
    {/***Core 1***/
        ….
    }
}
```

以上的代码由配置工具自动生成，这里配置了关于核间中断和时钟中断处理函数，这两个函数都是内部函数。其函数内部已经将中断进行了封装。在中断的总入口函数中，通过读取配置表来进行中断的分发。其总体结构图如图 4-2。

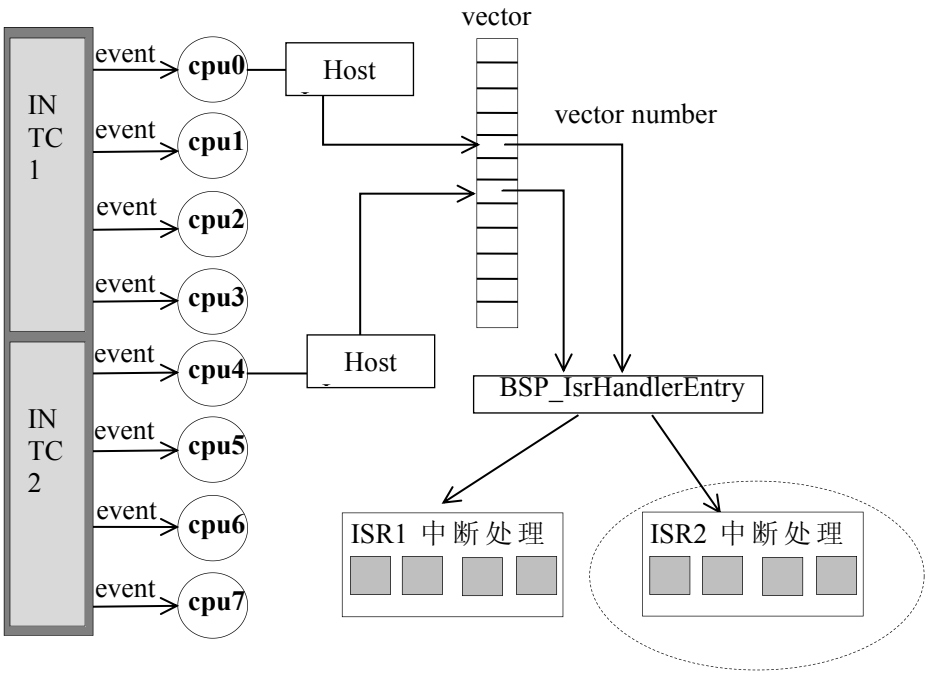

图 4-2 DeCore MOS 中断总体结构图

与中断入口配置表对应的是 osekConfig_InterruptTypeTable 配置表，该表用来配置中断类型，即 ISR1 或 ISR2。通过使用这个两个配置表，就能使用户很方便的进行中断的配置。值得注意的是，DeCore MOS 的一大特点就是中断的处理过程较断，没有进行过度的封装，能够汽车电子领域要求的中断快速响应特性。

4.2 自旋锁管理

TMS320C6678 的 KeyStone 架构没有明确的自旋锁概念，与此相同的是它提供了硬件信号量机制（Hardware Semaphore）[24][25]，使用硬件信号量能够确保多核心互斥的访问共享资源。然而该模块最多只提供 64 个独立硬件信号量（但可以实现多个自旋锁），并支持 3 种请求方式：直接请求、间接请求和组合式请求。使用请求方式类似于异步请求，当请求的资源已经被占用的时候，将会立即返回；而使用间接请求方式，当资源被占用时，程序将会不断的尝试获得该信号量，此时针对该信号量的请求会加入到请求队列，直到资源可用为止；组合式请求介于这两

者之间。针对以上所有的请求方式，程序对硬件信号量的访问是具有原子性的，正因如此，KeyStone 保证了使用硬件信号量能够正确地互斥共享资源。DeCore MOS 使用第一种请求方式，即直接请求。以下的代码片段是 DeCore MOS 中获取自旋锁的代码。

```
static inline int SpinLock(int id)
{
    unsigned long tmp;

    if (id < MIN_SPIN_LOCK || id > MAX_SPIN_LOCK) {                    ①
        DEBUG_DEV("bad spinlock id\n");
        return -1;
    }
    while(!(readl(SEM_DIRECT(id)) & 0x1));                             ②
    return 0;
}
```

代码①中用来判断自旋锁 ID 是否合法，目前 MIN_SPIN_LOCK 配置为 0，MAX_SPIN_LOCK 配置为 64；代码②是属于“自旋”代码，当采用直接方式的硬件信号量的时候，宏定义 SEM_DIRECT 为寄存器地址，通过读写该寄存器（SEM_DIRECTn）的第一位来判断对应 id 的硬件信号量是否被占用。如果被占用则自旋, 不被占用则获得该信号量。

从系统整体角度来看，DeCore MOS 将自旋锁分为系统级自旋锁（System SpinLock）和用户级自旋锁（User SpinLock），对于内核使用的自旋锁为系统级自旋锁，用户程序不应该使用系统级自旋锁，而应该使用用户级自旋锁。系统级自旋锁在内核中的使用方式是通过 HAL 层的 API 直接调用，其数目限定为 20 个，即对应硬件信号量编号 0~19。为了满足用户的使用需要，DeCore MOS 对用户级自旋锁进行了索引映射，将 20~63 映射为 0~32。自旋锁模块的整体架构如下图 4-2 所示。

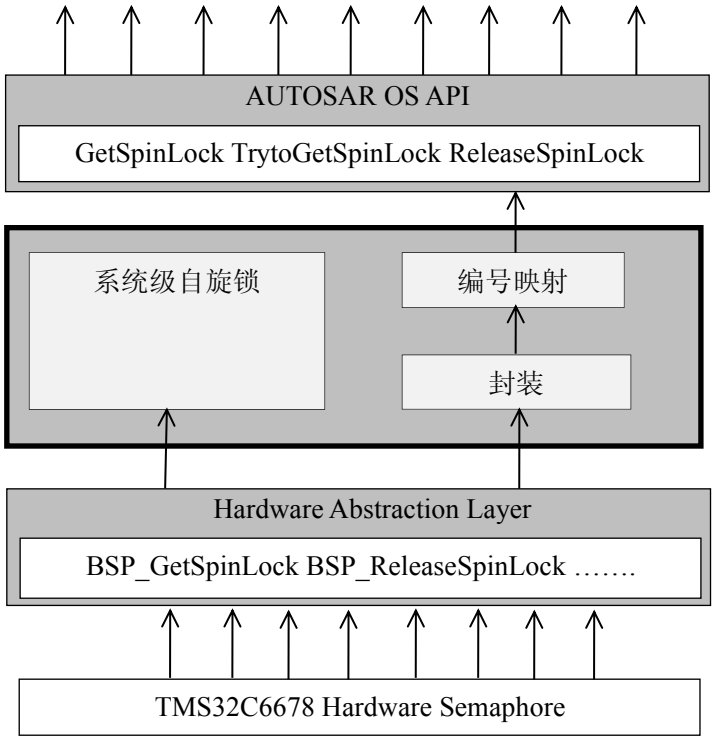

图 4-3 DeCore MOS 自旋锁设计架构

从上图中可以很清晰的看到 DeCore MOS 采用分层的结构来设计自旋锁机制，最上层符合 AUTOSAR 关于自旋锁部分的通用接口；硬件层使用 HAL 进行隔离，并通过 HAL 层直接提供系统级自旋锁所使用的一般方法。用户级自旋锁经过内核的封装（校验、实现、扩展）、编号映射以 AUTOSAR 规范 API 的形式提供给用户。

系统对用户级自旋锁做了一些限定：

（1） 当用户任务申请的自旋锁 id 不合法，则返回 E_OS_ID。

（2） 当用户申请自旋锁与所持有的自旋锁不同，那么函数需要返回 E_OS_NESTING_DEADLOCK，因为这可能会导致死锁。

（3） 当用户任务申请的自旋锁已经被相同核心的其他任务所占用，那么需要返回 E_OS_INTERFERENCE_DEADLOCK，这也可能导致死锁。

（4） 如果自旋锁不能被访问（如未配置自旋锁），则返回 E_OS_ACCESS。

为了满足这些限定，用户级自旋锁的数据结构包括了如下内容。

```
Struct T_OSEK_SPINLOCK_Lock_struct
{
    OSWORD      semID; /*  自旋锁对应的底层硬件编号,由系统初始化时指定  */
    SpinlockIdType lockId; /*  上层应用使用的软件自旋锁 ID */
};
```

第一个字段表示了硬件信号量的编号，用户一般不需要使用这个编号，该标号主要用于调试和作为调用 HAL 自旋锁相关函数的输入。而 lockId 则表示映射之后的自旋锁编号，比如硬件信号量编号为 22，那么 lockId 则为 2。值得注意的是不是所用用户程序都需要使用自旋锁机制，因此 DeCore MOS 提供了自旋锁配置开关，通过配置 OSEK_CONFIG_SPINLOCK_NUM 来指定需要使用的自旋锁数目，配置管理程序将会自动生成对应的用户级自旋锁线性表结构。

由于规范中有着明确的规定，当任务申请自旋锁时，该自旋锁如果已被当前核心的其他任务占用则返回；如果该任务已占有其他自旋锁也将返回。因此 DeCore MOS 并不需要提供类似 VxWorks SMP 中的中断级自旋锁和任务级自旋锁的划分。所谓任务级自旋锁，简单来说就是指自旋锁机制+调度锁；而中断级自旋锁是指在任务级自旋锁基础之上加上本地中断屏蔽。因此 DeCore MOS 的自旋锁没有在 SpinLock 函数里关调度器或关中断。以 GetSpinLock 代码为例，其代码流程图 4-3 如下：

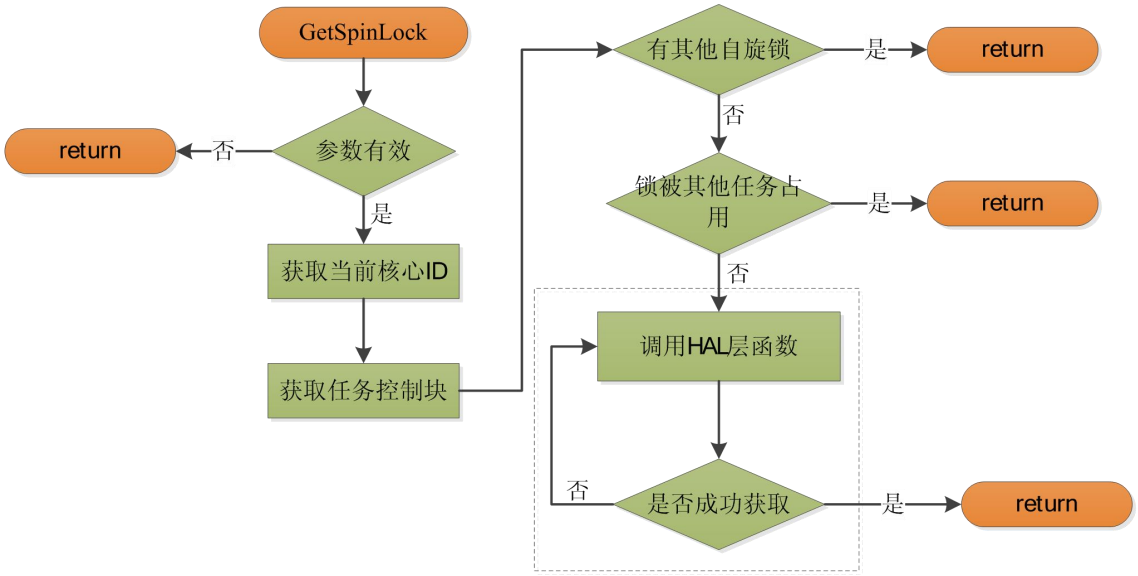

图 4-4 GetSpinLock 代码流程图

4.3 核间通信

在单核环境中的 OSEK 操作系统，可以通过事件机制来进行任务间同步；也可以使用 OSEK COM 规范所定义的消息通信机制来实现本地任务的消息传递，甚至不同 ECU 中的任务也可以通过该机制来进行彼此的数据交换。然而在多核环境

下，针对本地核心的任务通信，能够兼容地使用单核环境的通信机制（事件机制、消息通信），对于分布于不同核心中的任务间通信，由于所属核心不同以及同步方式的差异，也就必须额外地提供一种机制以支持这种多核特殊性。AUTOSAR 将这种机制的实现叫做 IOC（Inter-OS-Application Communicator，IOC）。在第二章介绍过的 AUTOSAR 的软件架构中的 RTE（Runtime Environment）就使用了 IOC 接口以提供统一的多核通信方式，但规范没有明确其技术的实现细节，因此 DeCore MOS 对该部分的底层实现进行一定的创新，通过使用核间中断[26]配合共享内存来实现其底层机制。DeCore MOS 提供了两种类型核间通信：其一是多核邮箱机制（Multi-core Mailbox），即通过多核间发送 Mail 到彼此的邮箱队列中，同步地通知消息的到来或者不进行通知。除了传递消息事件或数据，它同时支持回调函数。其二是远程过程调用（Remote Procedure Call，RPC），该机制能够屏蔽多核之间的底层调用细节，例如激活任务（ActivateTask）函数的调用者无需知道该任务是否在本地核心，也不必关心底层多核间同步交互过程，只需要关注是否任务成功地进行了激活。RPC 机制支持同步调用，并提供代理和回调函数的自动化配置工具。

多核消息邮箱机制为共享内存管理提供了底层支持，它的实现性能较高并且运用灵活，每一个核心都独立地具有一个消息队列，并在每次的核间中断中，一次性读取所有消息，如图 4-5 所示。

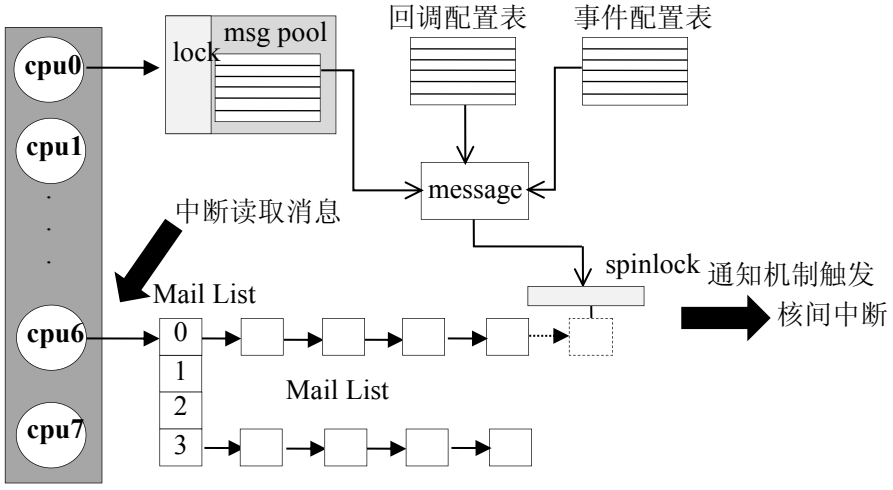

图 4-5 多核消息邮箱机制图示

从上图可以很清楚地看到每一个核心在本地都拥有一个消息邮箱 Mail List，并且消息按照不同的事件类型 event_code 进行划分，比如将 event_code==0 的消息通过单链表结构组织到一起，并采用 FIFO 算法处理每个消息。为防止发生多核间

的同步问题，每一个邮箱需要通过自旋锁机制进行同步管理，在系统中该自旋锁为 CORE_BOX_LOCK。当有一个任务需要发送一个消息给另一个 CPU 中的任务的时候，首先要通过 DeCore MOS 内存管理机制申请一个消息对象，该对象为 struct Message 结构体，该结构体的结构如下代码：

```
struct Message {
        unsigned char event_code;
        unsigned char message_content[MSG_LENGTH_MAX];
        struct list_head message_list;
        void (*callback)(void *data);
        int    from;
};
```

然后程序需要对该结构体进行相应的填充。因为消息通信可能包含了回调函数以及事件码（Event Code），所以 DeCore MOS 采用了静态配置管理，以配置每一个事件码和回调函数，这些事件码的意义和回调函数的功能部分对外提供给用户定义并实现的。消息对象填充完毕以后，就需要同步添加到对应目标核心的消息队列中去，因此需要在投放消息之前必须获得自旋锁，其后触发核间中断。值得注意的是，触发核间中断属于通知机制（Notification），即当消息挂接到消息邮箱之后，通知目标核心消息的到来。DeCore MOS 也支持非通知机制，即只将消息挂接到目标核的消息邮箱当中，而不产生核间中断就返回。上图中采用了通知机制，因此触发了 cpu6 的核间中断，中断总入口函数（BSP_IsrHandlerEntry）将该核间中断进行处理之后，分发给 BSP_IpcEntryHandler 函数进行具体的消息处理，在处理时（必须配合使用自旋锁），将会判断每条消息的消息类型，如回调类型、事件类型。当为回调类型时，将会在核间中断中进行回调函数的调用，此时消息对象将会以回调函数参数传递给该函数；如果是事件类型，则可能会激活处于等待该事件的任务，并将消息数据挂接到对应任务的控制块中；当所有过程都完成之后，静态内存管理系统将会回收该消息对象以继续使用。

4.4 RPC 机制

RPC 机制建立在核间通信的基础之上，为整个操作系统提供了一个跨核函数调用的底层支持。在单核环境下的 DeCore OS 中的绝大部分系统服务都不支持跨核调用。因此对于这些系统服务，其只能运行于本地，为了使本地任务能够跨核调用其他核心的操作，RPC 机制在多核消息邮箱的基础之上进行了一层封装（使用 event_code=1 事件码）。当产生了核间中断，并且当中断处理函数处理到该 RPC 消

息的时候则会调用 BSP_RpcHandler_Entry 函数，将该消息交由 RPC 处理函数进行处理。该函数将邮箱消息(message_content)强制转换为 IpcTransactionData 对象，该对象表示一次 RPC 请求，是整个 RPC 机制的核心部分，其相关结构体定义如下：

```
struct RpcTransactionData
{
    struct RpcTransactionHdr ipc_hdr;
    struct RpcTransactionRpc ipc_rpc_data;
    unsigned long reply;
};
struct FuncSlot
{
    unsigned int handle;
    void (*stub_proxy)();
    void (*stub)();
};
```

```
struct RpcTransactionHdr
{
    unsigned int rpc_type;
    unsigned int src_core;
    unsigned int dst_core;
};
struct RpcTransaction
{
    unsigned int    rpc_code;
    struct parcel   rpc_data;
};
```

从上述的代码中可以看到 IpcTransactionDate 由三部分组成，其一是 RpcTransactionHdr，它用来表示 RPC 的基本信息，如 RPC 类型、发起 RPC 请求的核心编号和请求调用核心编号；其二是返回值 reply，用于在调用者中接收函数的返回值；其三是 RpcTransactionRpc，它用来表示 RPC 事务处理中的数据和过程编号。其中的数据部分使用 parcel 结构体封装并提供了一组字节流操作，它用来表示远程调用函数的参数值；rpc_code 用来与 FuncSlot 做映射关系，因为远程函数调用需要调用已经实现了的相关过程，而这些函数可以通过 FuncSlot 对象进行统一的组织和绑定。每一个 rpc_code 都与一个 FuncSlot 的 handle 值相对应，所有的 FuncSlot 就组成了一个 RPC 配置表，该表在系统中叫做 coreRpcFunc，系统使用该表的部分空间作为内部实现而被占用，其余部分提供给用户，见图 4-6。

```
struct FuncSlot coreRpcFunc[RPC_FUNC_NUM] = {
    RPC_FUNC_SLOT(RPC_FUNC_TEST, func_test_stub_proxy, func_test_stub),
    RPC_FUNC_SLOT(RPC_ACTIVATE_TASK,rpc_activateTask_stub,rpc_activateTask_Server),
    RPC_FUNC_SLOT(RPC_GET_TASK_STATE,rpc_getTaskState_stub,rpc_getTaskState_Server),
    RPC_FUNC_SLOT(RPC_SET_EVENT,rpc_setEvent_stub,rpc_setEvent_Server),
    RPC_FUNC_SLOT(RPC_SET_REL_ALARM,rpc_setRelAlarm_stub,rpc_setRelAlarm_Server),
    RPC_FUNC_SLOT(RPC_SET_ABS_ALARM,rpc_setAbsAlarm_stub,rpc_setAbsAlarm_Server),
    RPC_FUNC_SLOT(RPC_GET_ALARM,rpc_getAlarm_stub,rpc_getAlarm_Server),
    RPC_FUNC_SLOT(RPC_CANCEL_ALARM,rpc_cancelAlarm_stub,rpc_cancelAlarm_Server)
};
```

图 4-6 FuncSlot 配置示例

可以看到，每一个 FuncSlot 都包含有一个 stub 代理和一个 stub。因为 RPC 的底层调用过程是类似的，使用 Stub 代理能够将这些相同点提取出来，减少用户或内核代码的编写工作。而 stub 函数就是跨核需要调用的那个函数。目前 DeCore

MOS 已经提供了 stub 代理生成工具，用户只需要定义函数返回值、参数以及函数名，工具将自动生成这些代理。参考图 4-7 更加清晰的表述了以上的工作过程。

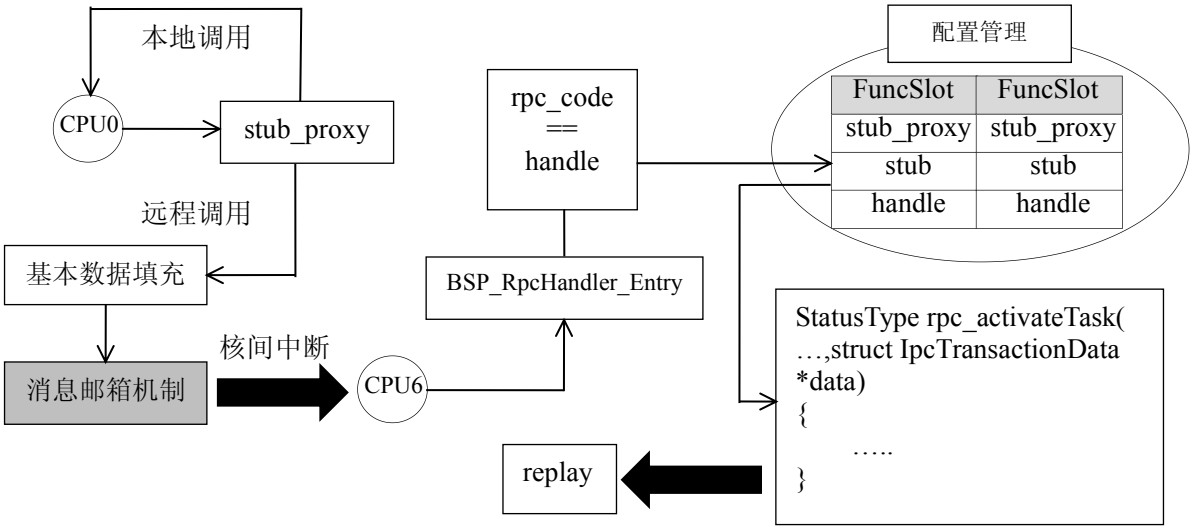

图 4-7 DeCore MOS RPC 机制流程图示

当需要进行跨核调用的时，则调用相应的代理桩子 stub_proxy，它会判断是否是本地调用，如果本地调用则直接调用本地函数。如果是远程调用，则其需要使用消息邮箱机制。因为 DeCore MOS 的 RPC 机制是同步跨核调用，所以上图 4-7 中的 reply 其实是 stub 函数中用于通知调用者，表明函数调用已经完毕，并返回函数执行结果。在 cpu0 中的任务阻塞地等待该远程调用结果，该阻塞不会进入阻塞队列而是类似自旋锁一样的代码“自旋”，但该任务也可以被高优先级任务抢占，这并不影响程序的正确性。需要注意的是当执行 BSP_RpcHandler_Entry 的时候，系统通过 RPC 事务结构体（RpcTransaction）中的 rpc_code 来匹配配置管理中的 FuncSlot 以查询远程调用函数的函数句柄，从而执行该远程函数调用。

4.5 本章小结

本章我们重点讨论了 DeCore MOS 多核驱动层的设计。DeCore MOS 的驱动层与内核层增加了一个硬件抽象层，该层屏蔽了底层硬件的细节，能够方便操作系统的硬件环境移植。DeCore MOS 的中断处理函数分为 ISR1 和 ISR2 两种类型，对于 ISR2 类型的中断处理例程，用户可以通过配置管理进行配置。DeCore MOS 的中断处理虽然较为简单，但是能够应对汽车电子中对中断响应的高实时性要求。在本节的第二部分里我们讨论了 DeCore MOS 自旋锁的实现方式，它分为系统级

自旋锁和用户级自旋锁。系统级自旋锁只针对内核层代码使用，而用户级自旋锁能够进行静态的分配。自旋锁的 TMS320C6678 实现是使用了硬件信号量机制，并且只能提供 64 个信号量，因此在用户使用的时候，系统将用户信号量标号进行了一定的映射，并将信号量的使用进行了封装，满足 AUTOSAR 规范对信号量的使用要求。最后我们详细说明了 DeCore MOS 的关键性技术——核间通信。该技术是整个操作系统的核心技术，它通过核间中断和共享内存机制实现。通过这两个技术 DeCore MOS 实现了消息邮箱机制，它能够支持多核间的消息传递、事件和事件回调。在此技术的基础之上实现了 RPC 机制。使用该机制能够使内核代码的编写无需考虑多核间复杂的通信过程和同步细节。而 DeCore MOS 中所有支持跨核调用的 API 都是通过该机制实现的，并且大部分的 API 都支持跨核调用。因此该技术的实现是整个 DeCore MOS 最为核心的一部分。

以本章作为底层多核技术的实现基础，在下一章里，我们将会讨论 DeCore MOS 内核的详细设计。

第五章 DeCore MOS 内核详细设计

5.1 多核任务管理

DeCore MOS 的任务管理是整个操作系统最为核心的一部分，它包括了多核任务配置、多核任务管理以及多核调度表机制。其中的多核任务管理又包括优先级管理、就绪队列管理、多核任务控制等。有关任务配置部分请参见第六章，调度表参见第 5.5 节。本节将重点讨论优先级管理、队列管理以及跨核任务控制。

5.1.1 任务调度与系统服务

有关任务的调度策略，许多相关文献已经做了很多深入的研究，如采用蚁群算法 ACO 和基本遗传算法 GA 混合实现的多核实时调度算法、多核处理平台的公平调度算法以及一些 Linux 内核[27]中常见的 O(1)、CFS 等。这些算法都经过了很完整的理论研究和工程实践测试。然而在面向汽车电子领域的嵌入式操作系统又有着特殊的要求，既要保证系统的简单性和高可配置性、又要求系统的高实时性保证以及确定性。实用的实时内核在实现时大多采用了简单的调度算法，以确保任务的实时约束特性和可预见性是可以管理的[28]。因此 AUTOSAR OS 规范指出，系统中任务的调度策略必须使用基于优先级驱动的可抢占式调度，即所有任务有且只有唯一不可变优先级，高优先级任务抢占低优先级任务（非 Non-preemptive 任务），相同优先级采用 FIFO 机制而不能采用时间片轮转。因此 AUTOSAR 操作系统的任务调度绝不是公平优先调度策略，也不应该使用复杂的调度算法。依此 DeCore MOS 依然沿用优先级位图的任务优先级管理方式，采用基于优先级的可抢占式调度策略。在多核环境下，每个核心的调度行为都是相对独立的，并且只能根据本地优先级位图表来进行优先级的查询和计算。由于所有的任务都是静态配置，一旦静态指定了任务的 CPU 亲和性，在整个系统运行阶段都不会发生亲和性转变，即任务不会在核心间动态跳跃。因此 DeCore MOS 不提供动态负载均衡算法，所有任务的负载都是用户静态指定的。虽然任务独立运行于本地核心，但这不意味着核心之间的任务就不能进行控制。DeCore MOS 提供了如下的系统服务接口来进行多核环境下的任务管理，见表 5-1。

表 5-1 DeCore MOS 任务管理 API

函数名	功能描述	跨核支持
ActivateTask	用于激活任务	必须
TerminateTask	用于终止任务，任务本身调用	否，但要检测和释放自旋锁
ChainTask	结束当前任务，并且激活指定任务	必须
Schedule	用于任务放弃运行，调度系统激活	否
GetTaskID	获得任务 ID	否
GetTaskState	获得指定的任务状态	必须

TerminateTask 用于将处于非挂起状态的任务加入到就绪队列，以使任务参与调度器调度。如果系统配置了 CONFIG_OSEK_SYSTEM_MACTIVE 宏，则任务能够多次被激活，并且每个任务（扩展任务和基本任务）都有相应的激活次数，一旦达到激活次数的上限，则任务不再允许继续激活。在多核环境中，允许进行跨核激活任务操作，即 Core0 能够激活 Core1 的任务。TerminateTask 用于终止一个任务，它与 ActivateTask 相对，但该函数只能由任务本身调用。ChainTask 是 TerminateTask 和 ActivateTask 的组合操作，它结束当前任务并激活指定任务，由于激活任务能够进行跨核操作，因此该函数也必须支持跨核操作，如 Core0 的 Task1 能够终止自己并且激活 Core2 上的 Task2，但 Core0 终止 Task1 后，需要由本地的调度系统进行任务重调度。Schedule 用于在任务中进行重调度，任务调用该函数能够放弃当前的 CPU 运行权，并进入就绪态，调度系统会选取其他高优先级任务进行调度运行。该函数不能支持跨核操作。GetTaskID 和 GetTaskState 用于获得本地当前正在运行的任务 ID 和指定任务 ID 的任务状态。GetTaskID 只能是本地操作，不支持跨核流程，GetTaskState 能够读取其他核心上的任务状态。

5.1.2 优先级管理

以上介绍了 DeCore MOS 支持的任务管理操作，这些操作都是建立在任务优先级管理的基础之上。DeCore MOS 采用了优先级位图，每个核心配置一个优先级就绪表 osekTask_PriorityBitMap 以表示本地当前处于就绪状态的任务优先级。每一位都表示一个优先级，从低位到高位优先级依次递减。除了优先级就绪表，每个核心同时还配有 osekTask_PriorityBitMapMajor 来表示优先级就绪组。如图 5-1 所示。

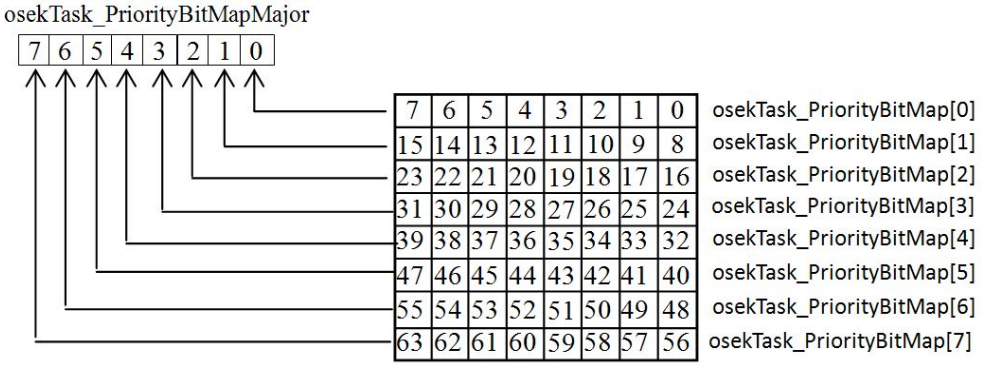

图 5-1 优先级组与优先级表的关系

任务的优先级通过任务控制块的 OSPRIOTYPE curPriority 域表示，它是一个 8 位优先级类型，最多可表示 64 个优先级（排除高 2 位），它的低三位表示优先级表中的横向索引，中间 3 位表示优先级组中的索引。如优先级 16，二进制表示为 00010000，即对应优先级组 2，osekTask_PriorityBitMap[2]的第 0 位。

同时，为了减少左移操作，系统定义了 osekTask_PriorityMapTable[8]，该表表示数字 1 左移的次数所对应的值，如 osekTask_PriorityMapTable[5]表示 1<<5 即二进制的 00100000，对应十进制的 32。该表为所有核心共享，并且为只读表，不存在同步性问题。

为了计算当前最高优先级，系统定义了两种实现方式，其一是使用硬件前导零（Leading Zero）指令，二是使用软件方式。当 DeCore MOS 系统定义了 CONFIG_OSEK_TARGET_PPC 宏则启用硬件前导零，使用硬件前导零指令的执行效率更高。而使用软件方式获得最高优先级，就需要所有核心共享 osekTask_PriorityIndexTable[256]表，该表为优先级索引表，用于快速定位当前最高的优先级。与 osekTask_PriorityMapTable 一样，该表是只读表，因此也不存在同步性问题。

使用以上的数据结构能够实现 O(1)的优先级管理算法。当任务进入就绪态时：

```
high3Bit = priority>>3;
low3Bit = priority&0x7;
osekTask_PriorityBitMapMajor |= osekTask_PriorityMapTable [high3Bit];
osekTask_PriorityBitMap[high3Bit] |= osekTask_PriorityMapTable[low3Bit];
```

以优先级 17 举例，17 对应的二进制为 00010001，它的高 3 位为 010，是十进制的 2，因此查询映射表为 00000100，所以将优先级组的第三位置为 1。00010001

的低三位为 001，对应十进制的 1，因此通过低三位和高三位就能找到对应 osekTask_PriorityBitMap 表中需要置 1 的位。

当任务退出就绪态时，就要将相应的优先级从优先级位图中清除，如下算法能够实现优先级清除：

```
high3Bit = priority>>3;
low3Bit = priority&0x7;
osekTask_PriorityBitMap[high3Bit] &= ~ osekTask_PriorityMapTable[low3Bit];
if(osekTask_PriorityBitMap[high3Bit]==0){
    osekTask_PriorityBitMapMajor &= ~ osekTask_PriorityMapTable[low3Bit];
}
```

算法首先清除优先级位图表中对应指定优先级的位。然后判断，该行是否为 0，如果该行已经没有优先级，那么清除优先级组 osekTask_PriorityBitMap 对应改行的那一位。

当任务发生切换，需要调度最高优先级任务的时候，就需要从优先级位图表中获取最高优先级。其算法如下：

```
high3Bit =osekTask_PriorityIndexTable[osekTask_PriorityBitMapMajor];
low3Bit = osekTask_PriorityIndexTable[osekTask_PriorityBitMap[high3Bit]];
priority = (high3Bit << 3) + low3Bit;
```

通过查询优先级索引表，能够很容易确定优先级的高低 3 位，将它们组合一起，就可以得到当前最高优先级了。

DeCore MOS 就是采用以上的优先级管理方式，并且每个核心也都是使用同一套优先级管理机制。有了优先级管理，还需要提供就绪队列和任务控制块的管理机制。

5.1.3 任务队列管理

DeCore MOS 的任务管理中，包含 4 种类型的队列：事件阻塞队列（Event Blocking Queue，也叫事件等待队列）、待激活任务队列（Suspended Queue）、终止队列（Termination Queue）和就绪队列（Ready Queue）。下面以扩展任务的任务状态结合各队列进行详细说明。见图 5-2 所示。

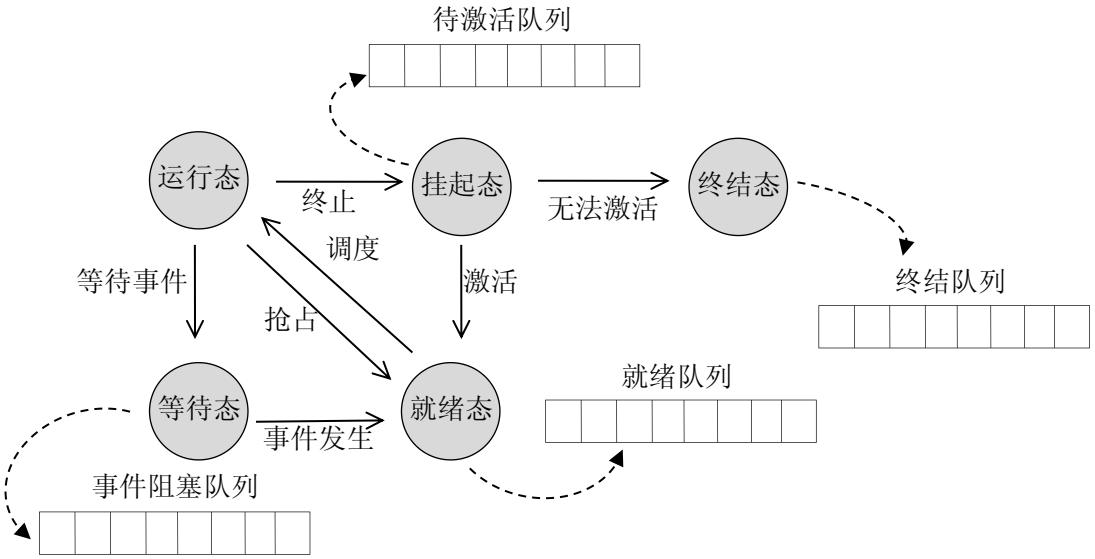

图 5-2 结合队列的任务状态转换图

上图可以看到，不同的任务状态对应进入不同的队列（除了运行态以外）。初始时所有任务都进入待激活队列并进入挂起状态。当任务被激活，则进入就绪态和就绪队列，以提供调度器调度运行所需“材料”。处于运行时的任务等待事件发生时，将会进入等待态，此时也将进入事件阻塞队列。处于运行态的任务除了进入等待态还有可能进入挂起状态（调用 TerminateTask），此时任务进入待激活状态，如果任务已经达到了激活的最大上限，那么就不可能在运行时中再次被激活，此时的任务已经终结，状态切换为终结状态，任务进入终结队列。终结状态和终结队列具有一定的存在意义，一方面能够使任务状态转换模型更加完整，另一方面便于统计和调试。

在 DeCore MOS 中所有的队列都是由 OSEK_QueueBlock 结构体组成的。该结构体由如下域组成。

```
typedef struct OSEK_QUEUEBLOCK{
    OSEK_QueueBlockRef   pre,next,head,tail;
    T_OSEK_TASK_ControlBlock * task;
    OSEK_DEBUG_QUEUE debugQueue;
} OSEK_QueueBlock, OSEK_QueueBlockRef;
```

pre 与 next 分别表示挂载的 OSEK_QueueBlock 前后指针，head 和 tail 分别指向整个队列的头部和尾部。Task 为任务控制块指针。debugQueue 为调试结构体对象，其中包括诸如挂载任务次数，挂接起始时钟数、内存回收次数等用于调试分

析的信息。

OSEK_QueueBlock 结构体对象由内存管理统一进行分配。以就绪队列为例，由于系统能够支持 64 个优先级，指定某一个优先级时就需要能够快速定位该优先级下挂接的任务控制块。因此就绪队列采用优先级与 OSEK_QueueBlock 一一映射的数组。如下图 5-3 所示。

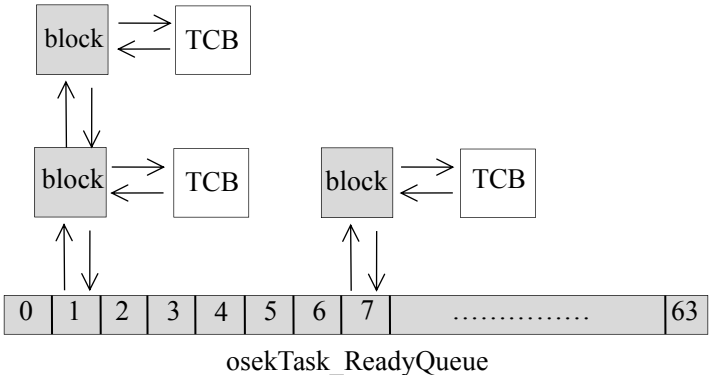

图 5-3 就绪队列与 TCB 关系图

从上图看到，任务控制块需要一个指向 OSEK_QueueBlock 的指针，该指针是一个通用的指针，它的意义随着 OSEK_QueueBlock 用途的不同而不同。当需要某一优先级下的任务控制块时，只需要通过优先级直接查询表中的 block，然后从 block 中可得到指向该任务控制块的指针。

因为除了就绪队列，其他的队列并没有优先级的概念，所以事件阻塞队列、待激活队列、终结队列并不需要使用数组结构进行组合。它们都是直接使用 OSEK_QueueBlock 对象直接链式聚合而成。这些队列的头部节点分别为：osekEvent_BlockingQueue、osekTask_SuspendQueue、osekTask_TerminateQueue。

所有的队列都统一使用一种结构所带来的最大好处之一就是能够统一代码的编写，所有的队列处理都能使用同一套代码。DeCore MOS 为队列操作提供了若干内部函数，由于代码逻辑较为简单，这里不再敷述。需要注意的是，在多核硬件环境下，DeCore MOS 为每一个核心都独立地配置了一套队列系统，它们之间不能直接相互干扰，必须通过跨核任务控制来间接的操作。

5.1.4 跨核任务控制

在多核环境下，DeCore MOS 的任务调度系统独立运行于每一个核心，并且核心之间不能直接进行相互干扰。但这并不是说不同核心上的任务之间毫无关系。

DeCore MOS 提供了多核消息邮箱机制、RPC 机制、多核事件控制等机制来同步跨核任务，以提供跨核任务通信的最基本的系统服务。除此之外，DeCore MOS 的任务管理还提供了单核环境下任务管理系统服务 API 的增强功能。在表 5-1 里已经列出了主要的任务管理 API，其中 ActivateTask、TerminateTask、ChainTask、GetTaskState 系统服务为了支持多核环境，扩展了单核的控制功能，使其能够在多核环境下不但可以控制本地任务也可以用来控制跨核任务。

TerminateTask 只能够用来终止本地正在运行的任务，使其进入挂起状态和待激活队列，但由于多核环境下增了自旋锁机制，因此除了要检查任务是否已经释放了所有资源以外，也应该要检查任务是否持有自旋锁，因此对于该函数的多核化改造只需要增加释放自旋锁相关的操作即可。需要注意的是，任务只能主动放弃运行权限，不能由 ISR 或其他核心任务被动终止。除了 TerminateTask 以外的扩展 API 都需要使用底层的 RPC 机制，该机制已经在第四章详细讨论过，这里不在敷述。下面主要说明 ActivateTask 的具体实现。见图 5-4 所示的跨核任务激活实例。

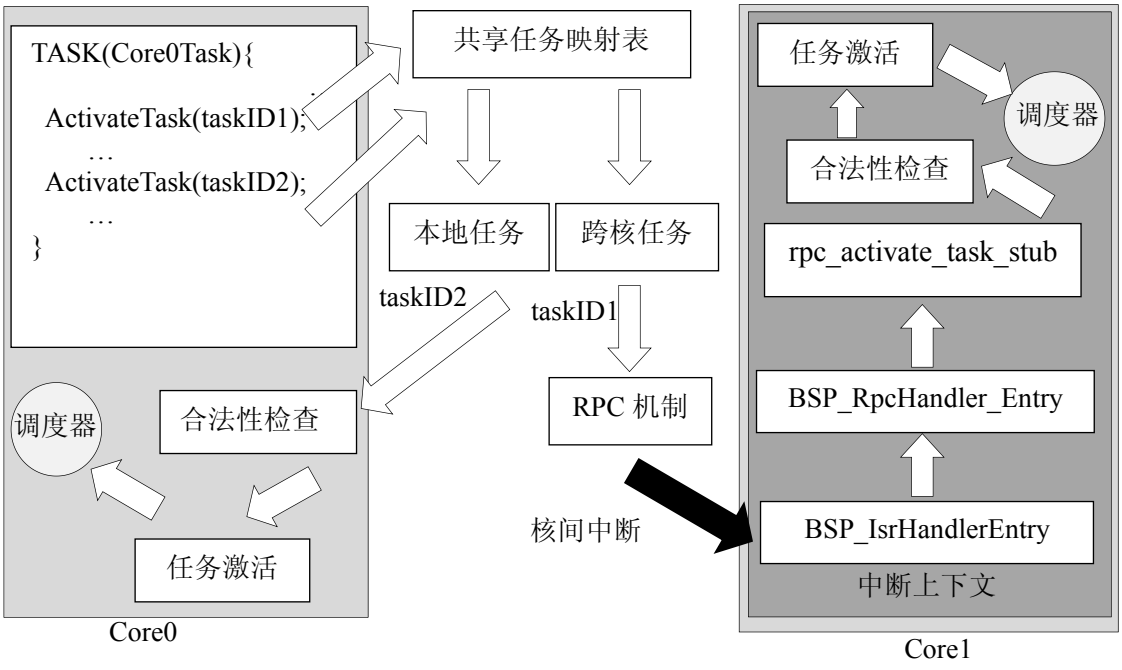

图 5-4 跨核任务激活示例图

图中 Core0 核心的任务 Core0Task 调用了激活函数 ActivateTask(taskID1)，它首先会读取共享任务映射表，由于 DeCore MOS 的任务配置都是静态的，也就是说所有任务在运行时已经确定，无论任务数目还是任务所在核心都是已知的，为了更加方便地查询对应某个 ID 的任务所在的核心编号，系统设置了该共享任务映

射表。该表隶属于 AUTOSAR 可定位实体内涵中的一部分（Locatable Entities）。任务 ID 分为全局任务 ID 和本地任务 ID，本地任务 ID 对应于本地任务实体的索引，不同的任务可能具有相同的本地 ID。全局任务 ID 是系统对外接口所使用的 ID，不同的任务只有唯一的全局 ID。共享映射表将任务的全局 ID 依次映射为核心编号，因此使用全局任务 ID 就可以查询到任务所在核心。上图的 taskID2 是本地任务，因此函数将会在本地进行合法性检查以及任务激活操作（进入就绪队列，设置任务状态等），如果此时满足调度条件，则需要调度器进行调度以运行该被激活的任务。taskID1 通过映射表得知该任务是属于 Core1 的，因此是跨核任务。此时则需要通过 RPC 机制支持，来进行跨核远程函数调用。RPC 机制依赖于 IPC 机制，底层会产生核间中断，以中断 Core1 正在执行的任务，在中断上下文中依次调用相应的中断处理程序，并激活任务判断任务执行条件，最后进行任务的调度。

需要注意的是，图 5-4 省略了核心合法性检查，当查询共享映射表得知该任务属于跨核任务后，还需要判断该任务所在核心是否已经处于停止状态（Shutdown），如果已经处于关闭状态那么该核心已经无法在进行任务调度和中断处理了，它已然不属于 AUTOSAR 操作系统管辖之内。

5.2 内存管理

通常，对于嵌入式实时操作系统在内存管理方面需要考虑几个因素[28]：快速而确定的内存管理、不使用虚拟存储技术、内存保护。一般在强实时系统中的内存管理较为简单，甚至不提供内存管理功能，而页面置换所带来的开销也使得嵌入式实时操作系统一般不适用虚拟存储技术。对于安全性要求较高，实时性要求不高，并且系统有着比较复杂的应用场景下，在内存管理上也就要相对复杂一些，需要实现对操作系统或是任务的保护（如 LynuxWorks）。DeCore OS 本身并不提供任何内存管理机制，然而在多核环境下，由于存在着核间通信，这就要求多核系统至少能够提供一定的内存管理机制以支持共享消息池的动态利用。一般来说，内存分配方式分为静态内存分配和动态内存分配，所谓静态内存分配是系统在启动之前，所有任务都获得了所需要的全部内存，运行过程中将不会有新的内存请求。而动态内存分配一般使用堆进行管理，通过在运行时进行分配或释放操作以使任务能够动态地使用内存。静态内存分配方式只适用于那些强实时、应用比较简单的和任务数量可以静态确定的系统。DeCore MOS 采用固定大小分区管理，并在其基础之上建立了消息池和队列块池。

固定大小分区的直接使用是不支持多核共享方式的，只有本地的任务能够使用该块内存，跨核任务之间如果需要使用共享内存，应该使用消息池和 IPC 机制。因此对于固定大小分区所对应的每一块分区都对应唯一的核心编号。系统使用分区控制块来管理每一个固定大小的分区，参考以下代码。

```
struct MEM_PARTITION {
    /*分区起始地址*/
    void        *addrPtr;
    OSCHAR    *namePtr;
    /*空闲链起始地址*/
    void        *freeListPtr;
    /*核心标号*/
    CoreIDType  targetCore;
    /*内存块大小*/
    OSWORD    blkSize;
    /*分区总数*/
    OSWORD    bbrMax;
    /*自由块总数*/
    OSWORD    bbrFree;
}PartitionType,*PartitionRefType;
```

每一个纳入 DeCore MOS 内存管理的内存块都从属于某一个分区，分区和分区之间相互独立，并且系统提供了 CreatePartition、GetMemBlk 和 PutMemBlk 接口用来操作创建分区、从分区中获取内存以及释放内存操作。创建分区将给定的内存区域划分为相等的部分，并使用前 4 个字节作为指针将这些内存块以链表的形式组织到一起，如图 5-5 所示。

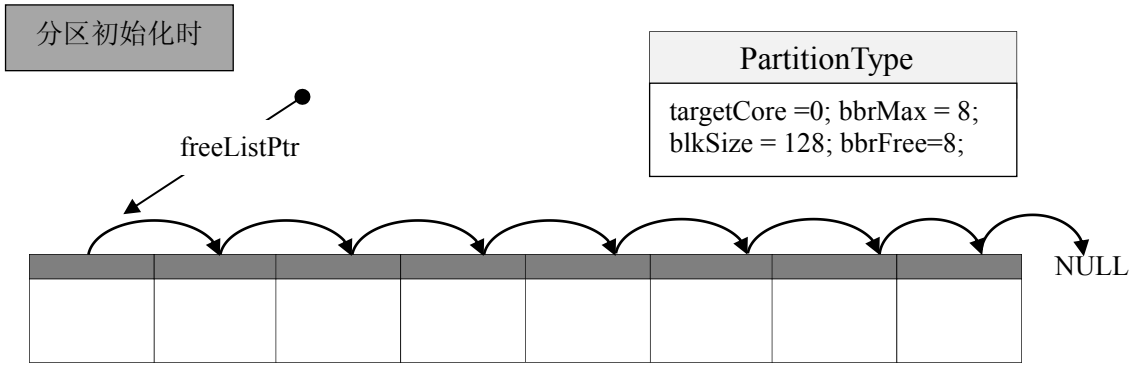

图 5-5 固定大小分区初始化

图中的分区对象所在核心为 0，意味着只有 0 核心上的任务能够使用该分区，每一块内存区域是 128 字节，一共有 8 个内存块并且初始化之后可用块数与总数

相等。当进行空闲内存块分配时，见图 5-6。

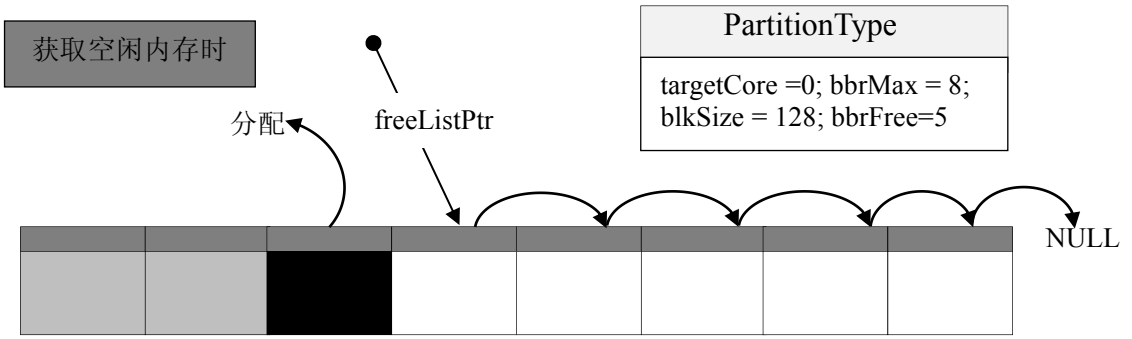

图 5-6 固定大小分区内存分配

从图中所示，分区中前两块内存已经被系统分配出去，所以系统将分配第三块内存。此时空闲指针指向被分配出去的内存块的下一块，即第四块内存。可以看到此时分区所属核心编号依然不变，总块数和每块内存的大小也都不发生变化，但是空闲块数会因内存的分配而自减。

当所需要使用的内存块已经用完，则需要将其回收到对应的分区中。系统将该分区的空闲块指针 freeListPtr 指向该回收内存块地址，并且将该块内存的前四个字节作为指针类型以指向下一个空闲块。此时空闲块的数目也将会相应的自加，见图 5-7 所示。

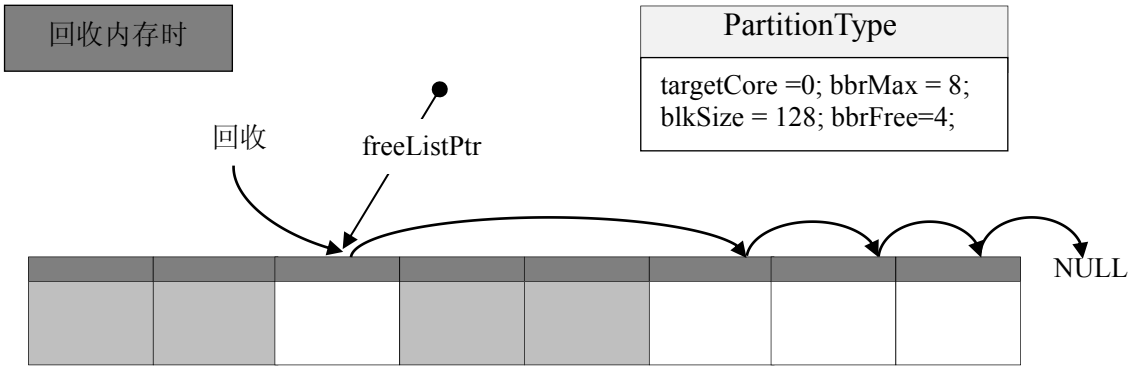

图 5-7 固定大小分区内存回收

由于每个分区表只对应一个核心，所以核心之间不会发生同步性问题，因此无需使用自旋锁进行保护，然而在同一核心中不同的任务（或 ISR）之间有可能会使用同一个分区表，因此在使用时需要关中断以及加调度锁。

消息池与队列块池建立在上述固定大小分区的基础之上。每个核心的消息池和队列块池的容量都可以进行静态配置。在内存管理模块初始化的时候，系统通

过调用 CreatePartition 以创建每个核心的消息池分区和队列块池分区。使用时通过 MSG_POOL_GET 和 MSG_POOL_PUT 宏获得消息对象和释放消息对象。

5.3 多核计数报警器

在汽车电子软件开发中，报警机制尤为重要，其应用场景也比较广泛。如汽车应急闪灯周期性闪烁以警示周围环境；雨刷器周期性扫落风挡玻璃上的雨水；再比如智能车锁保护系统当汽车行驶若干分钟后自动加锁车门。这些都是需要 AUTOSAR 操作系统提供计数器与报警机制支持的。在第二章里，我们已经讲到了在单核环境下 DeCore OS 所支持的计数器与报警机制，我们知道关于计数器 Counter 的实现，在规范中并没有做明确要求，它是属于内部实现。而关于报警机制，在规范中做了明确的定义和要求，也清晰地定义了对外的 API 接口。在 AUTOSAR 操作系统中至少需要提供一个系统计数器，每一个计数器可以挂接一个或多个 Alarm。当时钟中断被触发时，在中断处理函数中循环处理每一个计数器，并从每一个计数器挂接的 Alarm 队列中检查符合触发条件的 Alarm 进行触发，所支持的触发方式如执行回调函数、设置扩展任务事件、激活任务等。在多核的硬件环境下，情况发生了明显的变化。不同核心的报警机制是否能够相互影响、定时器是否能够驱动不同核心上的计数器，以及报警机制能否支持激活其他核心的任务等，这些问题都是要在多核环境下需要进一步的思考的。因此，我们针对计数器和报警做了如下三条要求：

1. 每一个核心至少有一个系统计数器，并且本地定时器只能驱动本地计数器。
2. 本地计数器不能驱动其他核心的报警以及调度表。
3. 本地核心的报警机制能够支持激活其他核心任务，支持对其他核心任务设置事件，但不支持调用其他核心的回调方法。

DeCore MOS 针对如上的三条要求，定义并实现了报警机制在多核环境下的主要特性以及功能。并在 DeCore OS 的基础之上提供了 AUTOSAR 所要求的软件自由运行定时器接口（Software Free Running Timer）。

5.3.1 多核功能

无论是单核环境还是多核环境，报警机制所要完成的任务是不变的，即为整个系统提供时间驱动。DeCore MOS 以低层定时器驱动上层计数器，计数器再驱动

报警器以及调度表，报警器和调度表都支持激活任务和设置事件，报警器还能够执行回调函数。其整体功能架构图如图 5-8 所示。

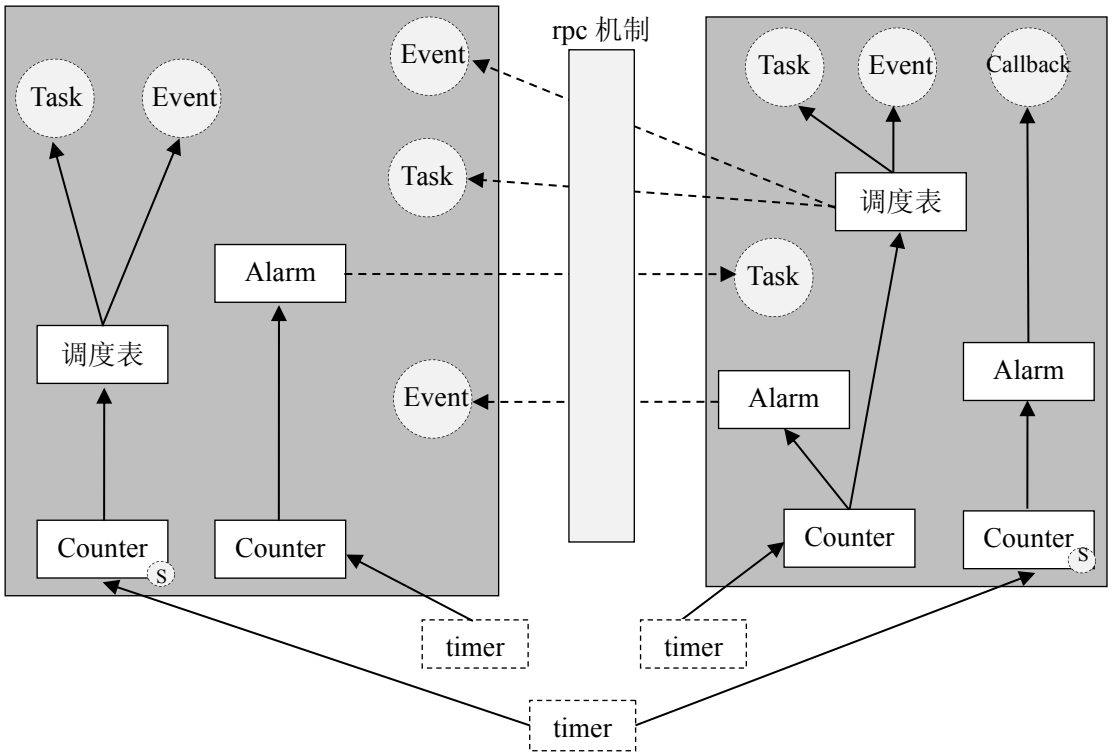

图 5-8 DeCore MOS 报警机制功能设计图

从上图可以清晰的看到，DeCore MOS 支持两种计数器，一种是本地计数器，还有一种是同步计数器。本地计数器由维护本地时钟中断函数进行驱动，同步计数器由主核（MAIN_CORE）统一维护，主核每次响应时钟中断时，会同步的修改每个核心的同步计数器，这些计数器维护了一个统一的时基。本地 Counter 能够驱动本地 Alarm 和调度表，本地 Alarm 和调度表能够激活本地任务、设置本地事件，其使用方式与单核环境是一致的。虽然本地 Counter 不能驱动其他核心的调度表和 Alarm，但这并不意味着多核之间的报警和调度机制是毫无联系的。从上图可以看到，处于两核之间的 RPC 机制提供了跨核函数调用（参考第 4.3.3），也就是说本地的调度表和 Alarm 能够通过 RPC 机制跨核地激活任务和设置事件。需要注意的是，DeCore MOS 的计报警机制不支持远程回调，即本地 Alarm 不支持远程调用其他核心的回调函数。

DeCore MOS 的计时报警功能能够满足规范的要求，并建立在这些要求的基础之上。与此同时，在实现细节方面，DeCore MOS 对 DeCore OS 报警机制做了一定

的创新，它引入了时间轮机制（Time Wheel），该机制能够减少时钟中断的响应时间，尽量避免了在中断处理中采用循环方式查找符合触发条件报警器的 O(N)算法。

5.3.2 时间轮机制

在 DeCore OS 的实现中，时钟中断处理程序每次调用 CounterTrigger 触发指定的计数器，然后调用 osekAlarm_check1 函数循环检查双向队列中的警报对象，以查看报警是否应该进行触发。使用这种方式需要进行一次循环操作，其时间复杂度为 O(N)，N 表示警报对象的挂载数量。由于警报对象是在运行时动态添加的，因此对于某一个时间内，该段代码所执行的时间会随着报警对象数目的增多而增加。如下代码显示了 DeCore OS 使用方式循环检查报警对象。

```
while (almId != (AlarmType)0 ) { //  遍历指定 counter 的 ALARM 链表
        //检查 alarm 是否到期并做相应处理
        status = osekAlarm_Check2(almId, ctrPtr);
        //判断 status 的值，如果需要调度，则设置调度标志为 1
        if( status == OSEK_TASK_NEED_DISPATCH ) {
            dispatchFlag = 1;
        }
        //得到链表中的下一个 ALARM
        almId = almId->nextAlarm;
    }
```

因此为了提高实时性，DeCore MOS 引入了时间轮机制。每一个 counter 对象不再直接对应一组 Alarm，而是对应一个时间轮 OSEK_TimeWheel，并且在每一个时间轮节点 OSEK_TimeSpoke 上挂接报警对象。如图 5-9 所示。

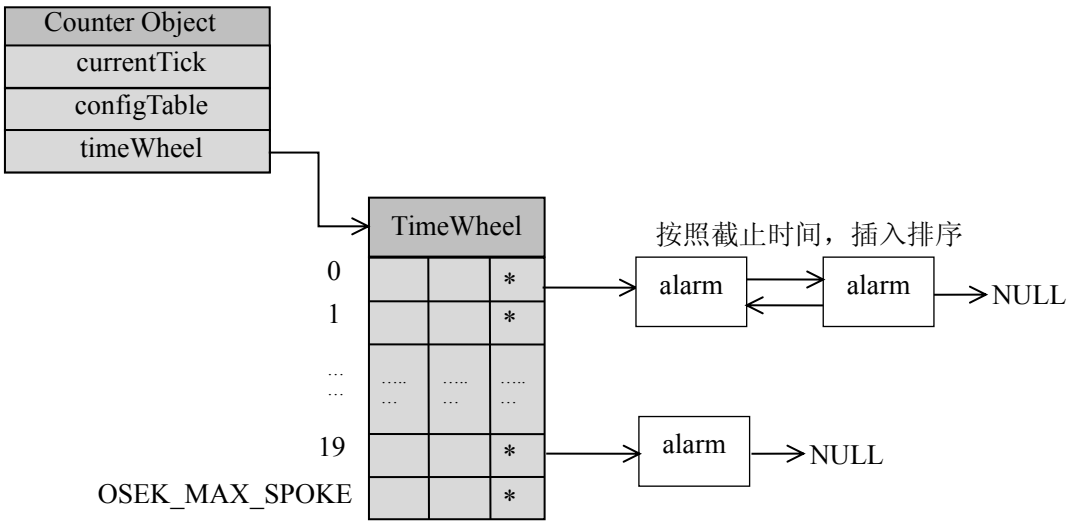

图 5-9 基于时间轮的 Alarm 结构图

从上图可以看到，报警对象 Alarm 组织方式依然采用的是双向链表结构，但是在插入报警对象的时候，系统使用了插入排序[29]将到期时间（expirationTick）从小到大进行序列化。并且每一个 Counter 对象都对应自己的一个时间轮 TimeWheel 对象，同步时钟对应一个同步时间轮（Synchronization TimeWheel），每个核心之间的时间轮机制相互独立，不能进行互操作，因此也就没有多核间同步问题。考虑以下场景：配置 OSEK_MAX_SPOKE 为 20，本地 Counter 的当前时钟为 100，此时任务调用报警机制设置报警为 25 个时钟“tick”后触发。因此,

- expirationTick ：currentTick(100)+25=125
- 插入 spoke 位置：125 % OSEK_MAX_SPOKE(20) = 5

因此该报警对象将会挂接到 OSEK_TimeWheel[5].alarmPtr 的位置上，如果该位置已经挂接了其他报警，那么系统将会进行插入排序，插入到符合条件的位置上。当发生了一次时钟中断，此时 currentTick =currentTick+ 1=121，此时会查询，时间轮的 121%20=1 的位置，但是该位置并没有挂接任何报警，因此时钟中断返回。当又发生了 4 次中断，那么此时的 currentTick 应该为 125，查找 125%20=5 的位置，将会找到刚刚挂接的报警，将其从链表中取出，再进行相应的处理（如设置事件等）。如果该报警是周期性触发类型，系统会重新计算新的 expirationTick 值，再挂接到时间轮上。TimeWheel 除了具有 alarmPtr 指针外，还有为统计和调试准备的 entityCount 和 maxEntity。

采用时间轮的架构方式，当每次发生时钟中断处理时，不需要遍历整个 Alarm 链表，只需与 OSEK_MAX_SPOKE 求余，再找到对应的 spoke 位置即可。因为系统中的 OSEK_MAX_SPOKE 宏能够提供给用户进行静态配置管理，并且用户通过静态分析能够计算出合理 OSEK_MAX_SPOKE 值，从而达到更好的效果，甚至能够做到整个系统在运行时几乎不会发生 spoke 冲突（一个 spoke 挂接多个 alarm）。因此使用时间轮机制能够更好的满足实时性要求。

5.3.3 数据结构与实现

由于在多核环境下采用了时间轮机制，计时报警系统的数据结构发生明显的变化。为了支持时间轮机制，增加了 OSEK_TimeWheel 结构体和 OSEK_TimeSpoke 结构体，并且计数器控制块和报警对象控制块都增加了有关时间轮的指针。而为了支持多核环境的静态配置，配置管理中的结构体也增加了相应的内容。

整个计时报警系统结构体关系图参见图 5-10。

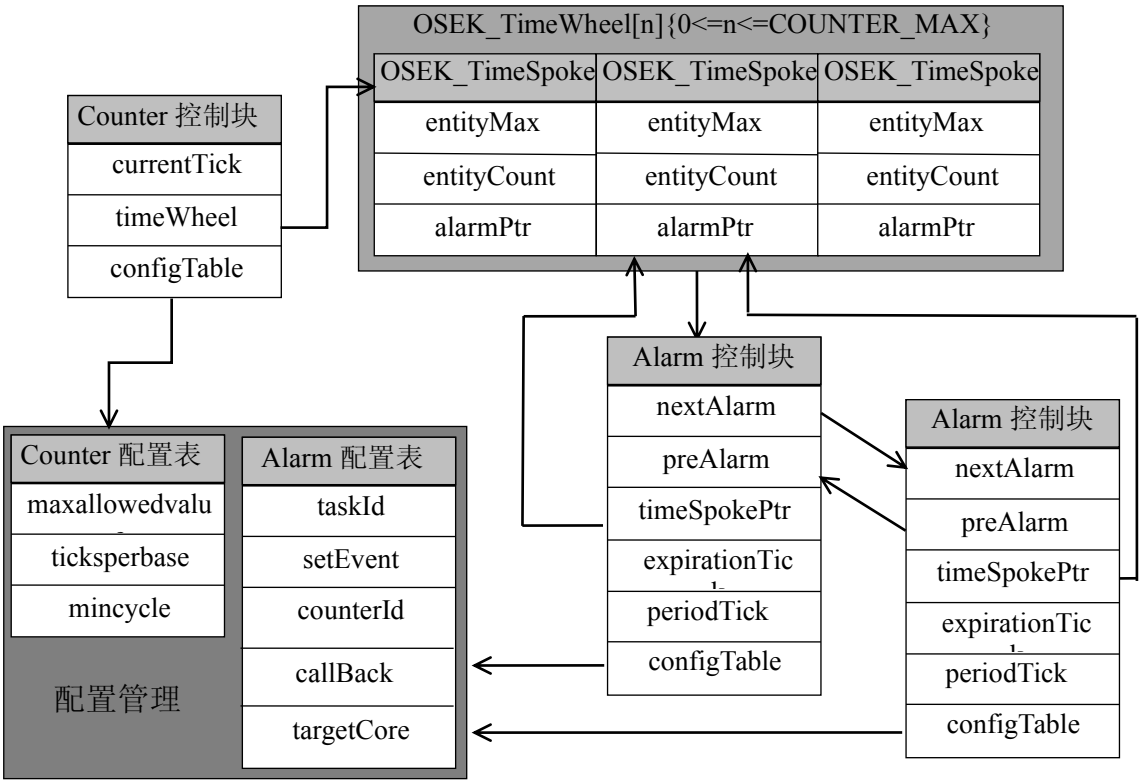

图 5-10 计时报警数据结构关系图

不同核的计时报警机制数据结构关系都是一致的，并且不同核心之间并不能共享 Counter 控制块、Alarm 控制块、TimeWheel 以及配置表。DeCore MOS 的计数报警机制都是依赖于上述的数据结构。

规范定义了 5 个标准 API，下面以 SetRelAlarm 和 CounterTrigger 函数为例，具体说明整个计数报警机制的算法处理流程。SetRelAlarm 用以设定一个报警，它属于系统标准 API，而 CounterTrigger 是 DeCore MOS 的内部实现，并不属于导出 API，它在时钟中断服务例程中被调用，用以执行报警检测算法。

SetRelAlarm 的函数签名如下：

StatusType SetRelAlarm(AlarmType almId,TickType increment,TickType cycle)

almId 对应了一个 Alarm 控制块，increment 就是相对于当前本地时钟的增加值，cycle 是周期执行次数。SetRelAlarm 的返回值是 StatusType 类型。当返回 E_OK 的时候，函数正确执行；当返回 E_OS_STATE 时，表示当前的 Alarm 已经被占用；而当返回 E_OS_ID 的时候表明 Alarm 的 ID 是非法的，也就是说不存在该 Alarm 的 ID，很有可能用户配置出现问题；当返回 E_OS_VALUE 时，设置的相对时间超出了计数器最大值，比如计数器最大值 maxallowedvalue 是 0x6FFFFFFF，而此

时设定 0xFFFFFFFF，此时会返回 E_OS_VALUE。

该函数的算法流程图如下图 5-11 所示。

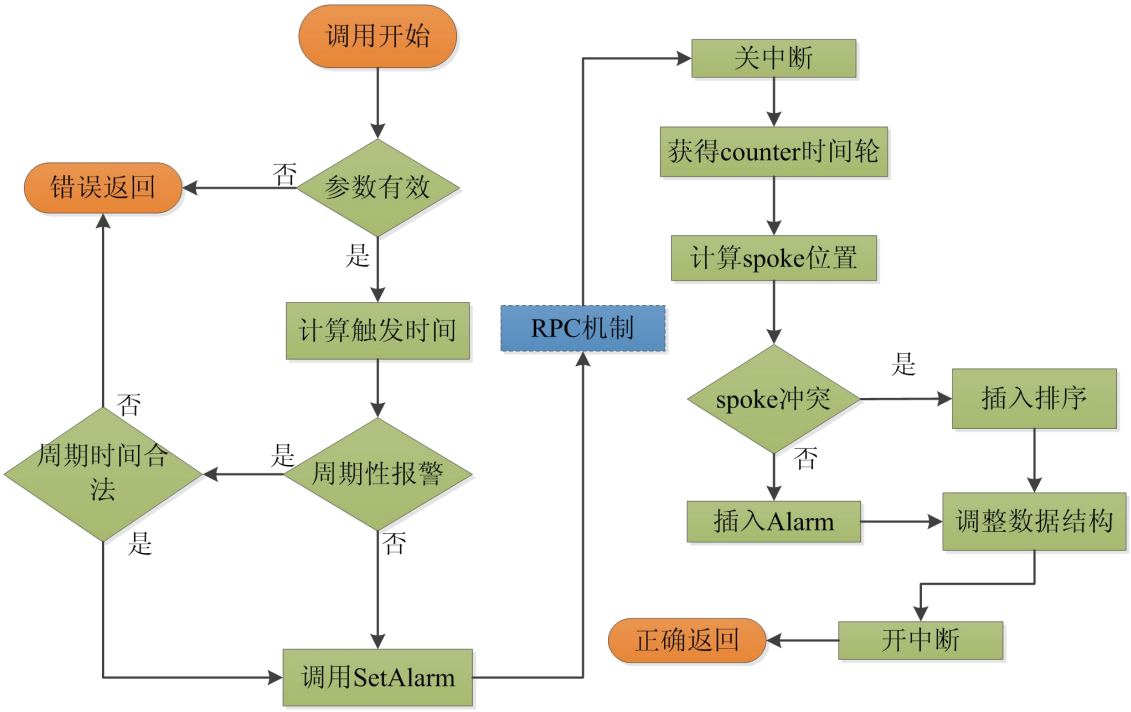

图 5-11 SetRelAlarm 算法流程图

该算法在每个核心的流程都是一致的，其中的 SetAlarm 是操作系统内部函数，它首先需要判断该 Alarm 所对应的核心，通过控制块的 targetCore 域与当前核心 id 做比较，如果它们不相同则表示需要进行跨核调用，即需要将该 Alarm 挂接到其他核心的时间轮上，因此系统会调用 rpc_setRelAlarm_stub 函数，该函数为自动生成的代理，它将远程调用其他核心的 rpc_setRelAlarm_server 函数，以完成真正的 SetRelAlarm 功能； CounterTrigger 是每一个核心的时钟中断处理函数都需要调用的，他用来驱动指定的 Counter。如果用户没有自定义计数器，那么在时钟中断处理中只会调用 CounterTrigger(SYS_COUNTER)，来驱动本地的系统计数器。与 SetRelAlarm 操作相反，CounterTrigger 会从时间轮中查找对应的 spoke，然后找到满足触发条件的 Alarm 进行触发。

CounterTrigger 函数算法流程图参见图 5-12。

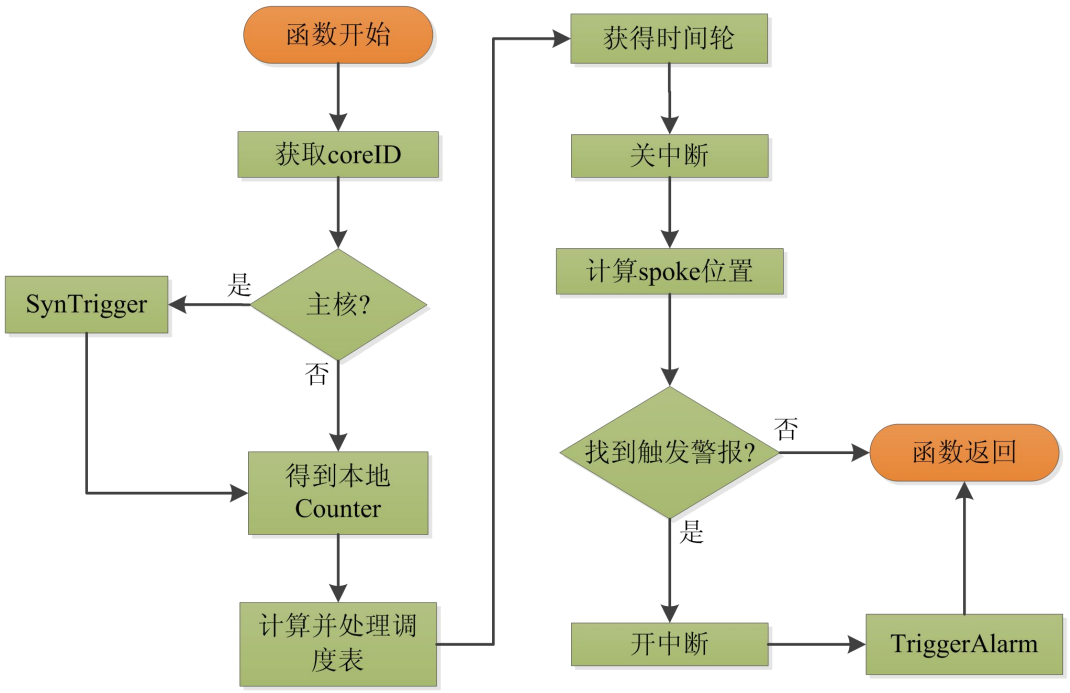

图 5-12 CounterTrigger 算法流程图

从流程图中可以看到函数最开始需要获得调用核的核心编号。由于 DeCore MOS 是支持同步计数器的，因此需要判断当前核心是否为主核心，如果为主核心那么需要调用 SynTrigger，该方法用来同步每个核心的同步计数器。无论是否为主核，算法都要获得本地 Counter 控制块，再更新 Counter 对象的 currentTick 域，同时处理调度表。当调度表处理完后，算法获得 Counter 对象的时间轮数据结构，然后通过上一小节介绍的时间轮算法得到需要触发的报警，如果没有对应的报警那么算法结束；如果具有需要触发的报警，那么算法调用 TriggerAlarm 对报警进行触发。TriggerAlarm 能够根据报警控制块的不同设置来选择是否进行函数回调、或者激活任务、或者设置事件。

5.3.4 自启动 Alarm

前面介绍的报警机制都是用户在系统运行时动态调用系统 API 进行添加的。DeCore MOS 除此之外也支持多核自启动 Alarm（Auto Start Alarm），即在系统启动初始化时，将配置管理生成的自启动 Alarm 数据挂接到相应的 spoke 上。本地时钟中断第一次运行时就能够判断该 Alarm 是否进行触发，甚至如果用户配置了 1 个 tick 的相对时间类型的 Alarm，那么发生第一次时钟中断时就会触发该报警。使

用自启动 Alarm 能够方便用户使用和进行系统调测，比如计算系统初始化到第一次时钟中断发生的时间间隔等。

每一个自启动 Alarm 的配置与上文所述的 Alarm 配置表不同，自启动 Alarm 使用 osekConfig_AutoAlarmTable[][]来保存其配置。每一个配置表结构都包含一个 Alarm 控制块指针，以及若干与时间有关的参数。如 delay 表示自启动的延迟时间；startTick 表示自启动 Alarm 的相对起始时间，与 Alarm 控制块的 expirationTick 相对应；periodTick 表示报警重复执行次数。

为了支持高可配置性，当用户需要使用自启动 Alarm 时，就必须通过配置工具配置 CONFIG_OSEK_ALARM_AUTO_EN 宏以开启自启动支持。

5.4 多核事件控制

DeCore MOS 支持事件控制机制，并且在多核环境下提供了任务间最基本的同步方式。无论是在相同核心的任务之间，抑或是跨不同核的任务之间，都可以使用事件机制进行同步。相同核中的任务的事件控制机制与 DeCore OS 是类似的，而跨核事件控制机制则建立在 RPC 机制的基础之上。DeCore MOS 能够兼容 AUTOSAR 提供的关于事件控制的基本 API，但每一个 API 接口都有相应的使用场景和条件限制。如表 5-2 所示。从表中可以看到，DeCore MOS 的事件机制在 SetEvent 时能够支持跨核调用；而 WatiEvent 与 ClearEvent 只能由扩展任务调用；而 GetEvent 只能得到本地扩展任务的事件，这是因为如果支持获得核外扩展任务的事件，那么就需要使用自旋锁保护任务控制块，这首先会导致代码性能的下降，其次可能会导致死锁；下面两个小结将会讨论这 SetEvent 和 WaitEvent 的实现细节。

表 5-2 事件控制系统服务场景与限制表

函数名	使用场景	支持跨核	限制
SetEvent	调度表、任务、Alarm、中断	是	必须兼容多核系统
WaitEvent	扩展任务	否	只能由扩展任务调用
GetEvent	任务、中断	否	只能获得本地任务事件
ClearEvent	任务	否	只能由事件拥有者调用

5.4.1 WaitEvent 的实现

WaitEvent 只能由扩展任务调用，并且当任务处于等待状态的时，需要确保任

务不占有资源以及自旋锁从而避免死锁的发生。关于 WatiEvent 的函数算法流程图见图 5-13。

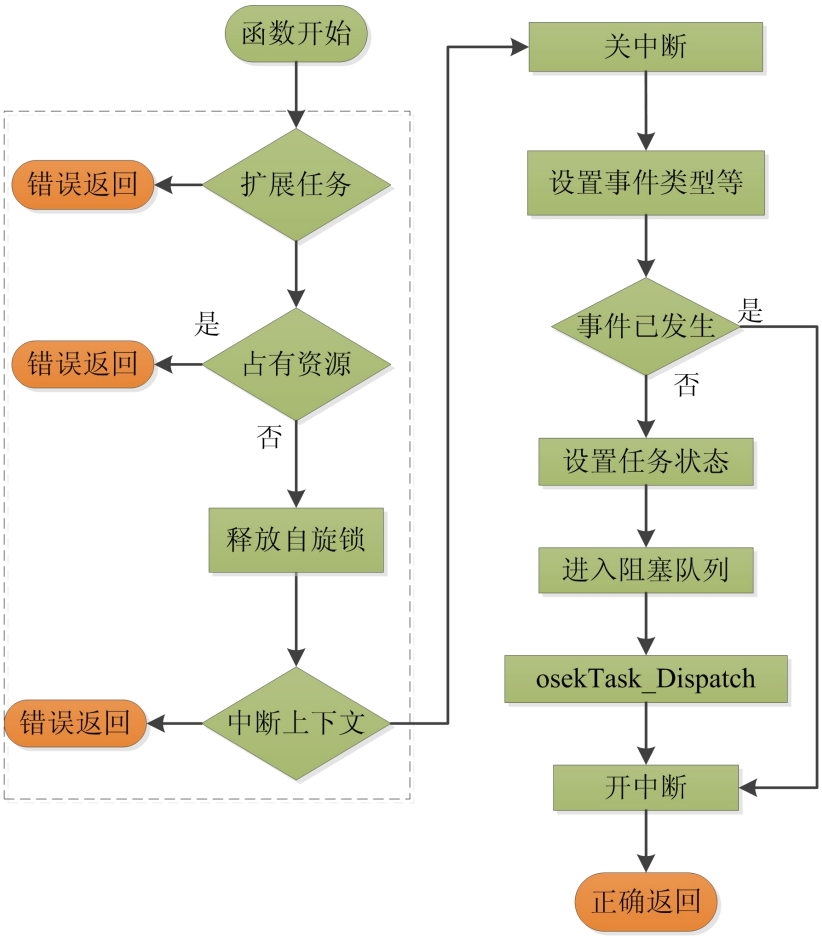

图 5-13 WatiEvent 流程图

上图虚线圈起来的部分是函数的安全性检查代码，我们规定，只有扩展任务在任务上下文中，并且当任务不占有任何资源的情况下才能进行事件等待。这里需要注意自旋锁的释放，当任务持有自旋锁时调用 WatiEvent 时，必须将持有的自旋锁释放以防止死锁的产生。由于是在本地执行的代码，因此无需使用自旋锁进行保护，但为了保证本地同步性，需要关闭中断。如果任务等待的事件已发生，则不会将任务插入到阻塞队列。扩展任务控制块中设置了 setEvent、waitEvent 和 eventType 域，这三个域是多核事件控制机制所必须的。setEvent 表明外部（ISR、任务）所设置的事件；waitEvent 表示任务所期待的事件集；eventTriggerType 表示事件触发类型，它分为两种，一种是单事件触发类型（Single Event Trigger），另一种是全事件触发类型（Whole Event Trigger）。如图 5-14 所示。

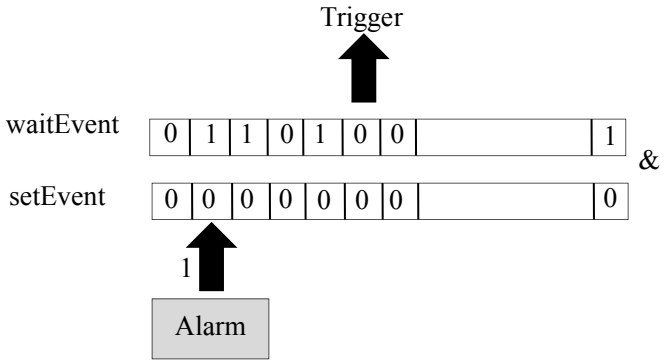

图 5-14 单事件触发模式

扩展任务的事件集为多事件的情况下，无论 Alarm 或者 ISR 还是任务，只要设置了其中一个或多个任务所等待的事件，那么就会发生该任务的状态切换。但是全事件触发方式则有所不同，见图 5-15。

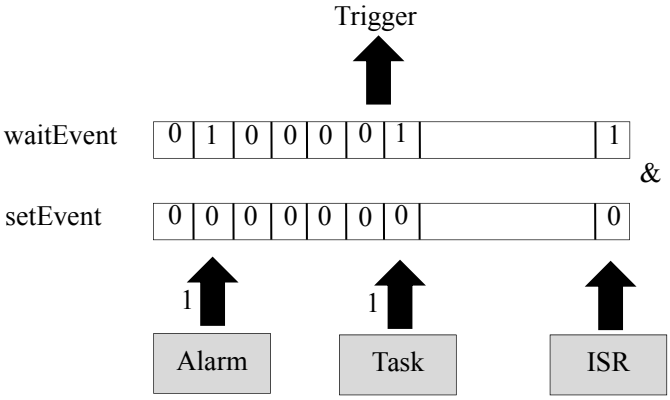

图 5-15 全事件触发模式

全事件触发模式必须保证扩展任务所等待的事件全部发生时才可以发生任务状态的变迁，进入就绪队列。并且每个事件的触发源不一定是相同的，比如上图第一个事件是由报警器设置的，而最后一个事件是由中断设置的。

使用这两种触发模式能够给用户的使用带来一定的灵活性。但无论哪种事件触方式，WatiEvent 都将调用 osekTask_Dispatch，最终完成任务的切换。

5.4.2 SetEvent 的实现

与 WaitEvent 相对应的函数是 SetEvent，当扩展任务状态因等待事件而进入 WAIT 状态时，仅能使用 SetEvent 函数将其从阻塞队列中唤醒。SetEvent 算法流程见图 5-16 所示。

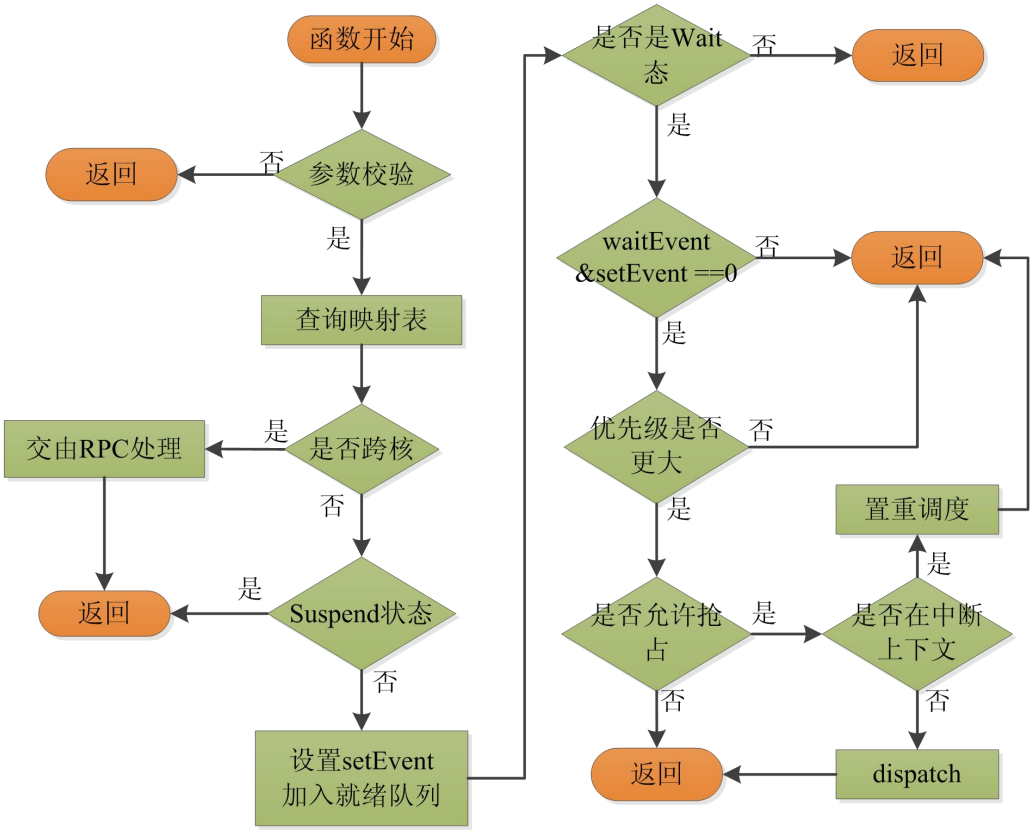

图 5-16 SetEvent 单事件触发模式下流程图

需要首先说明的是，该流程图是在单事件触发模式下的算法流程，因此当判断该事件的触发条件的时候，使用 waitEvent & setEvent == 0 来进行触发条件判断。如果是在全事件触发模式下，应该使用~waitEvent & setEvent == 0 来判断触发条件。为了支持跨核操作，SetEvent 使用了 RPC 机制。它的 RPC 函数代理也是通过配置工具自动生成的，为 rpc_setEvent_stub，远程函数调用服务为 rpc_setEvent_server。在函数完成了一般性校验后，则会检查所设置的事件是否为跨核事件，也即是否需要进行跨核 RPC。是否为跨核事件的判断使用了共享任务映射表，通过查询该表，就能够很方便的获得事件所对应的核心编号，再通过简单的判断就可以知道事件所属的任务是否是本地任务。如果不是本地任务，当然上述已经说明了，此时需要 RPC 机制发送 RPC 请求；如果是本地任务，那么则会检查任务是否处于挂起状态，处于挂起状态的任务是不允许被设置事件的。如果任务不在处于挂起状态，那么将会把事件掩码（Event Mask）与 setEvent 做或操作，即设置事件。

接下来算法判断任务是否处于等待态，如果不处于等待态的任务，那么算法

返回。如果处于等待状态，算法继续判断设置的事件能否进行触发，如果能够触发，则与当前正在运行的任务进行优先级比较。这里需要注意的是，当要唤醒的任务优先级小于正在运行的任务那么就不会发生任务切换，但是如果其优先级更大时则又有两种情况，其一是当前任务是不可抢占式任务，那么就不应该发生任务切换，也即任务的切换只能发生在不可抢占任务指定的调度点上；其二是当前任务可以发生抢占，那么此时的调用如果不处于中端上下文，则会调用 osekTask_Dispatch 进行任务切换。如果处于中断上下文，那么将会把 osekTask_IsrTriggeredTaskSwitchNecessary 至为 1，并在中断处理中根据该值以判断是否应该进行任务的切换等相关操作。

5.5 多核调度表

通过前文的介绍，我们能够使用一个 OSEK 计数器和基于该计数器的自启动报警队列来实现静态定义的任务激活机制，当该计数器值一次达到报警触发值时，触发该报警并调用回调函数、设置事件或者激活任务。然而报警一旦启动，其报警值就不能进行修改了，这无法保证报警之间的相对同步。因此 DeCore MOS 引入了调度表机制。调度表是由一组封装好的并且静态定义的终结点（Expiry Point）组成，这些节点都纳入调度表的管理机制当中。每一个 OSEK 计数器都可以驱动至少一个调度表，但调度表仅能被某一个计数器驱动，并且如同报警器一样，调度表也能够支持自启动模式和运行时启动模型。

如图 5-17 所示，图中定义了具有三个终结点的调度表，最先被调度到的终结点叫做初始终结点（Initial Expiry Point），最后被调度到的终结点叫做最终终结点（Final Expiry Point），每一个终结点都具有一个时钟偏移值，并且规定调度表的计时总是由 0 开始，从 0 到初始终结点的延迟时间叫做初始延迟（Initial Delay），由最终终结点到调度表结束的延迟时间叫做最终延迟（Final Delay）。终结点的调度会按照偏移值（Offset Tick）由小到大一次进行。每一个终结点都有两种操作方式，一种是激活任务，一种是设置事件。需要注意的是，终结点的操作与报警器相比少了回调函数调用功能。

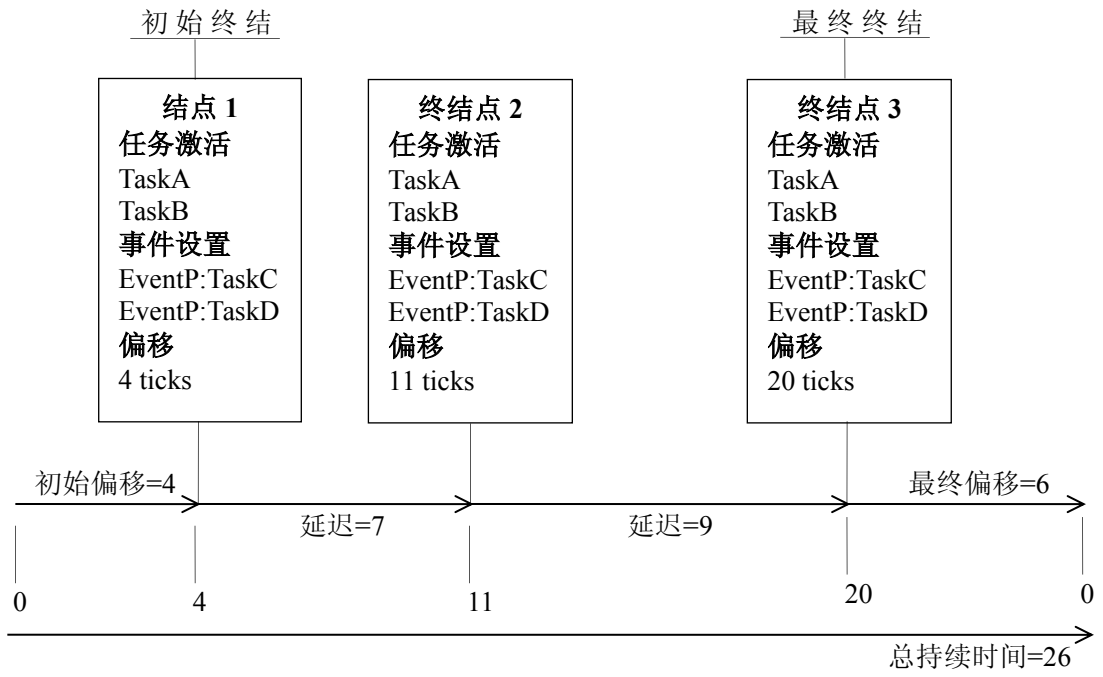

图 5-17 终结点调度时序图

调度表根据运行情况进而能够表现出不同行为，因此调度表具有自身的状态模型。DeCore MOS 的调度表具有 5 种状态，见下表 5-3。

表 5-3 调度表状态集列表

状态宏名称	说明	支持情况
SCHEDULETABLE_STOPPED	调度表停止	支持
SCHEDULETABLE_WAITING	等待同步	暂未支持
SCHEDULETABLE_NEXT	运行下一个调度表	支持
SCHEDULETABLE_RUNNING	调度表运行	支持
SCHEDULETABLE_RUNNING_AND _SYNCHRONOUS	同步运行状态	支持

调度表初始状态为停止态，如果配置调度表的自启动宏则能够在系统初始化时直接启动调度表运行使其进入运行态。当调度表发生切换，即运行下一个调度表时，当前调度表将会处于 SCHEDULETABLE_NEXT 状态，直到下一个调度表执行完毕则恢复到运行状态。如图 5-18 显示了 DeCore MOS 调度表的状态迁移。

系统提供了 StartScheduleTableAbs 和 StartScheduleTableRel 来运行一个指定的调度表，与报警器类似，前者表示使用绝对时钟值，而后者表示使用相对时钟值。

启动一个调度表运行，则调度表进入运行状态，调用 NextScheduleTable 将会使得当前调度表终止调度并进入 SCHEDULETABLE_NEXT 状态。在调度表运行中，如果调用了 StopScheduleTable()或者调度表执行过程已经结束，那么将会导致进入停止状态。需要注意的是此图中没有表现运行同步运行状态，关于此点将在 5.5.3 小节讨论。

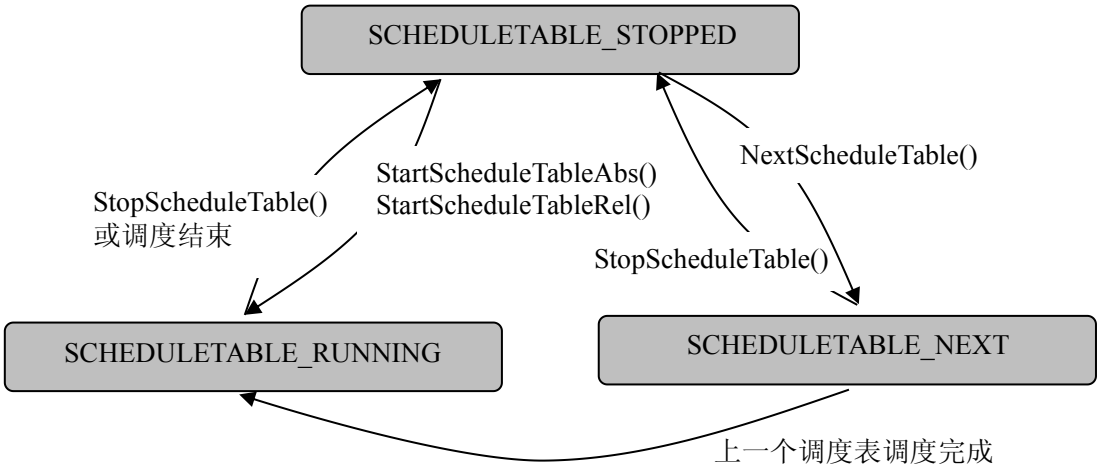

图 5-18 调度表状态迁移图

从图 5-18 可以看到，调度表的运行依赖于计数器，而其数据结构的设计又与报警器有着非常明显的区别。

5.5.1 调度表结构

DeCore MOS 将终结点抽象为 ExpirePoint 结构体。每个终结点对象由 ExpirePointGroup 线性表进行聚合，表中的索引为终结点 ID，并且在系统生成时由用户静态配置每一个终结点对象。以下代码显示了终结点对象域组成结构。

```
typedef struct EXPIRY_POINT{
    ExpiryPointType id; /*  终结点 ID */
    TaskType activatedTasks[2]; /*  激活任务列表  */
    /*  设置事件列表  */
    struct{
        TaskType task;
        EventMaskType eventMask;
    }setEvents[2];
    TickType offset; /*  终结点相对时钟偏移  */
    ExpiryPointRef next; /*  下一个终结点  */
}ExpiryPoint,*ExpiryPointRef;
```

activatedTasks 域表示需要激活的任务 ID 数组，setEvents 域表示需要设置的事件组。由于终结点调度的最终目的就是激活任务或者设置事件，因此每一个终结点都至少配置其中的一个或多个。当某个终结点被调度的时候，系统使用这两个域来作为激活任务或设置事件的目标参数。offset 表示终结点的偏移值，该值为相对于调度表起始 0 值的偏移。

调度表对象由两部分组成，其一是保存调度表运行时信息。另一个是保存调度表静态配置信息。静态配置信息不能在运行时动态改变，而运行时信息能够动态地反应调度表状态变化。如下代码所示。

```
typedef struct SCHEDULETABLE_CONFIG{
    /* 终结点配置组 ID */
    ExpiryPointGroupType epGroupID;
    /* 调度表持续时间 */
    TickType duration;
    /* 初始延迟与最终延迟*/
    TickType initialDelay;
    TickType finalDelay;
    /* 是否重复执行 */
    OSBOOL isRepeat;
    /* 所属核心*/
    CoreIDType targetCore;
    /* 同步策略 */
    ScheduleTableSyncStrategy syncstra;
}ScheduleTableConfig,*ScheduleTableConfigRef;
```

```
typedef struct SCHEDUELTABLE{
    /* 下一个调度表 */
    ScheduleTableRef next;
    /* 上一个调度表 */
    ScheduleTableRef pre;
    /* 当前执行的时钟位置，由 0 起始 */
    TickType currentTick;
    /* 绝对时钟值和相对时钟值 */
    TickType absTick;
    TickType relTick;
    /* 调度表状态 */
    ScheduleTableStatusType status;
    /* 配置表 */
    ScheduleTableConfigRef configTable;
    /* 初始终结点和最终终结点指针 */
    ExpiryPointRef initialEP;
    ExpiryPointRef finalEP;
    /* 下一个要执行的终结点 */
    ExpiryPointRef nextEP;
}ScheduleTable,*ScheduleTableRef;
```

ScheduleTableConfig 保存了调度表配置信息，每一个调度表唯一索引一个配置信息表。ScheduleTable 则保存了调度表运行时信息，其中包括调度表状态，以及下一个要执行的调度点指针等。以上数据结构关系见图 5-19 所示。

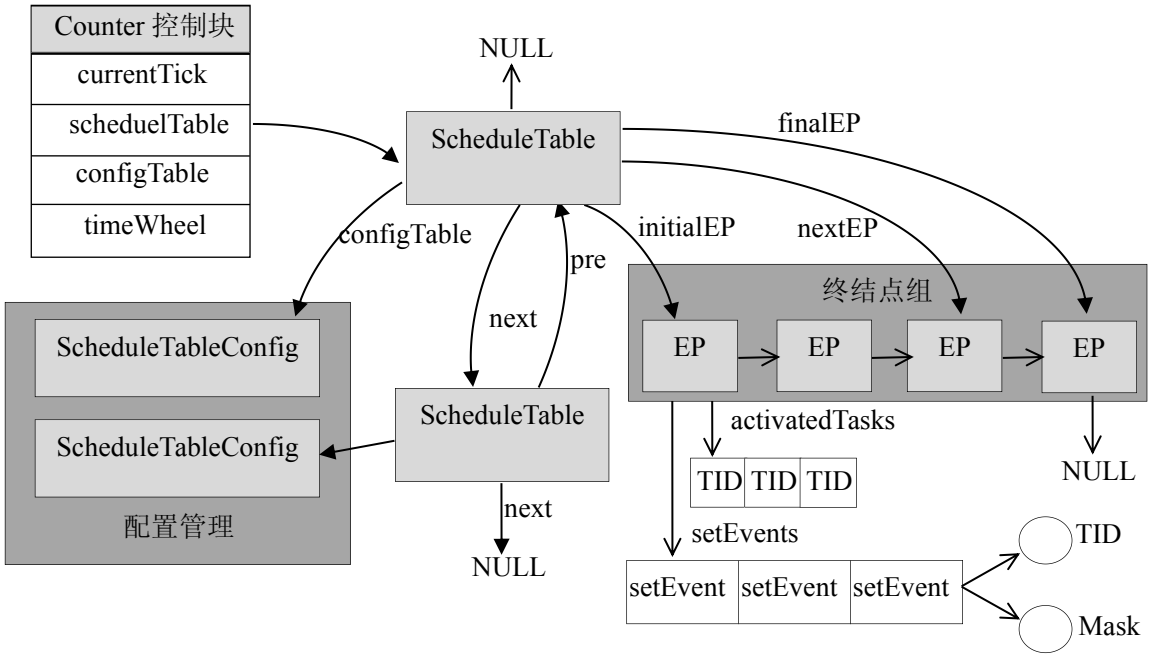

图 5-19 调度表数据结构关系图

图中 EP 表示终结点，它能够挂接一组待激活任务 ID（TID）和一组待激活事件。每个调度表对应一组终结点，该组的终结点成员通过链表结构组织到一起，并且按照偏移时间从小到大排序。ScheduleTable 的 finalEP 和 initialEP 分别指向了终结点组中最后一个终结点和第一个终结点。nextEP 就是下次调度时需要处理的终结点指针。调度表之间是通过 next 和 pre 指针组织成双链表结构。每一个调度表都挂载一个与之对应的调度配置表，该配置表纳入 DeCore MOS 的配置管理中。当调度表模块初始化时，初始化程序（SchduleTable_Initialize）通过读取配置信息以准备调度表运行时环境。当调度表处于运行态时，每次计数器增加的值都会同步更新到调度表的 currentTick 域，当调用 StopScheduleTable 时，将该域清空；调用 NextScheduleTable 时，该域将不会与计数器进行同步，以使调度表能够重新运行时恢复之前抢占点的状态。

5.5.2 调度表同步

根据具体的需求，调度表可配置为单次执行或重复执行。当重复执行时，由

于调度表的持续时间不一定等于驱动计数器的模数（maxallowedvalue 域），所以并不能保证同一个终结点在不同次重复中都在同一个绝对计数值上执行，而在某些情况下是必需的，如校正发动机的角旋转度数。所以需要同步来解决该问题。规范提供了隐式同步（Implicit Synchronization）和显示同步（Explicit Synchronization）。显示同步需要一个同步计数器配合使用，其实现较为复杂和繁琐，目前 DeCore MOS 仅支持隐式同步，但已留有后期扩展接口。见下图 5-20 所示。

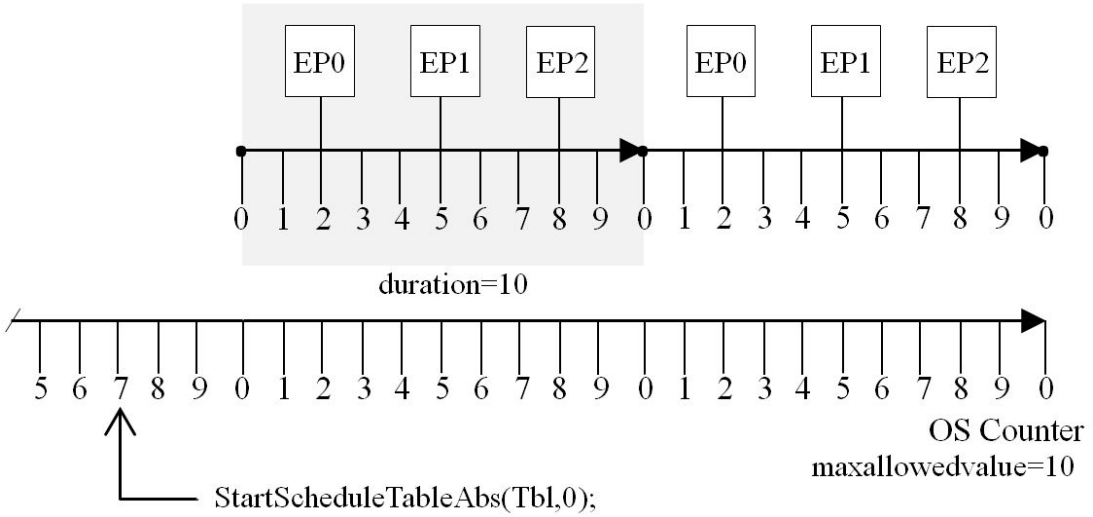

图 5-20 隐式同步时序图

图中可以看到，调度表挂载了 3 个终结点，并且采用循环运行方式。当计数器 currentTick=7 的时候，调用 StartScheduleTableAbs(Tbl,0)函数使得该调度表在绝对时钟 0 处开始执行，初始延迟为 2 个时钟“tick”，之后依次调度 EP0、EP1、EP2。当完成了所有终结点调度之后，调度表当前的时钟数值为 0。如果不采用隐式同步方式，计数器的当前时钟应该为 10。此时再一次执行调度表运行，当又执行到初始终结点的时候，计数器的 currentTick 值应该为 12，这就与第一次执行该终结点时的绝对时钟数值发生了变化（currentTick=2），要求每次重复执行终结点的时候，绝对时钟“tick”应该保持不变。如果采用隐式同步方式，就很容易发现，第一次执行 EP1 时与第二次执行 EP1 的时间差为 12-2=10，刚好为调度表的持续时间 duration，如果计数器的模数（maxallowedvalue）被设置为调度表的持续时间，那么每次当计数器达到最大值得时候，就会重新卷绕，也就能搞保证每次调度相同终结点时的绝对时钟固定不变。如上图，第二次 EP1 的绝对时钟依然为 2，与第一次执行时相同。

隐式同步只作用于调度表的循环执行，只有调度配置表的 isRepeat 被置位时方能生效。采用隐式同步的优点是实现简单，操作性强；缺点是当同一个计数器既挂载了调度表又挂载了报警器时，由于隐式同步的约束条件（模数等于调度表持续时间）使得报警器采用绝对时钟运行方式（SetAbsAlarm）时，容易忽略该模数，导致报警永远不会执行。并且当执行不同调度表时，需要动态设定计数器的模数以保证不同的调度表都能够进行同步。因此隐式同步方式最佳实践是使用在一个计数器只挂在一个调度表的场景中。

5.5.4 多核调度表

在多核环境下，DeCore MOS 依然支持调度表机制。从功能上讲，系统从以下两个方面进行了扩展。

（1） 调度表跨核控制。

（2） 终结点跨核激活任务以及设置跨核事件。

与报警器相同，DeCore MOS 在多核环境下独立地配置每一个调度表，并且只能所属于某一个核心，一旦调度表确定所属核心，也只能在本地进行调度。但调度表管理 API 接口能够支持跨核操作。如表 5-4。

表 5-4 调度表接口列表

函数名	功能	多核支持情况
StartScheduleTableRel	运行调度表	支持
StartScheduleTableAbs	运行调度表	支持
NextScheduleTable	暂停并运行指定调度表	不支持
GetScheduleTableStatus	获得调度表状态	支持
StopScheduleTable	停止调度表运行	支持

从上表中发现除了 NextScheduleTable 不支持跨核操作以外，所有的 API 都能提供跨核操作。每个函数都会在参数检查之后，通过调度配置表的 targetCore 与当前核心编号进行比较，如果该调度表是所属于其他核心，那么将会通过 RPC 机制进行远程跨核调用，其内部函数为 rpc_StopScheduleTable_stub 等。由于 RPC 机制已经在上文有过详细的讨论，这里不在敷述。

报警器能够进行跨核激活任务，设置事件。与其类似，调度表机制除了支持跨核管理控制外，也能够在终结点被调度时，根据终结点配置表来激活其他核心

的任务或者设置其他核心任务的事件。由于其具体实现与报警器类似，这里也不在敷述。需要注意的是，如果使用了调度表的多核特性，那么应该使用同步计数器来驱动该调度表。如果使用本地计数器驱动，由于每个核心的关中断时间是不一样的，因此可能会导致每个核心所面对的绝对时钟“tick”值的不同。

5.6 多核启停

在单核环境下的 DeCore OS 内核的启停，都是按照线性的顺序进行的。比如用户例程中调用 StartOS，则会依次执行关中断、启动通信接口、资源初始化等。这些过程按照指定顺序执行，其中并不涉及多核启停同步。然而在多核环境下，每一个核心的硬件环境、数据结构、软件功能等都是相对独立的，因此这就必须保证在多核环境下内核启停的同步性、可配置性。

5.5.1 同步启动

DeCore MOS 的启动包括板级初始化、内核初始化、内核模块初始化等。本节主要涉及内核的启动初始化。DeCore MOS 的多核启动属于架构中的执行体管理部分（Executor Management），执行体管理主要管理操作系统的运行状态以及启停等相关内容。在其中的多核启动部分，提供了如下的 API 接口，见表 5-5。

表 5-5 DeCore MOS 多核启动 API

相关 API 接口	功能
StartCore	启动指定核心，主从核都可以调用，但启动之后不能调用
StartOS	主要进行模块和数据接口初始化，每个启动的核心必须调用
StartNonAutosarCore	启动非 AUTOSAR 控制的核心

从上表中可以看到，DeCore MOS 为了支持多核的启动，系统增加了 StartCore 和 StartNonAutosarCore 这两个函数。StartCore 是启动一个核心，该核心处于 AUTOSAR 的控制之下，并且每个核心的方法都可以调用该函数，但是一旦当系统初始化完成，进入到运行态（CORE_STATE_RUN）时，就不能再继续调用该函数激活其他核心了，此时会得到 OS_E_STATE 错误码。为了支持非 AUTOSAR 管理的核心，系统提供了 StartNonAutosarCore 这个函数（比如它可以启动μC/OS-iii），但是需要注意的是，非 AUTOSAR 管理的核心是不能调用 DeCore MOS 的相关接口的，否则系统的确定性将会遭到破坏。

DeCore MOS 的多核启动采用主从模式（Master-Slave），即主核用于最初的启动，它能够启动 0 个或多个从核，同时从核也能够启动其他从核。见图 5-21。

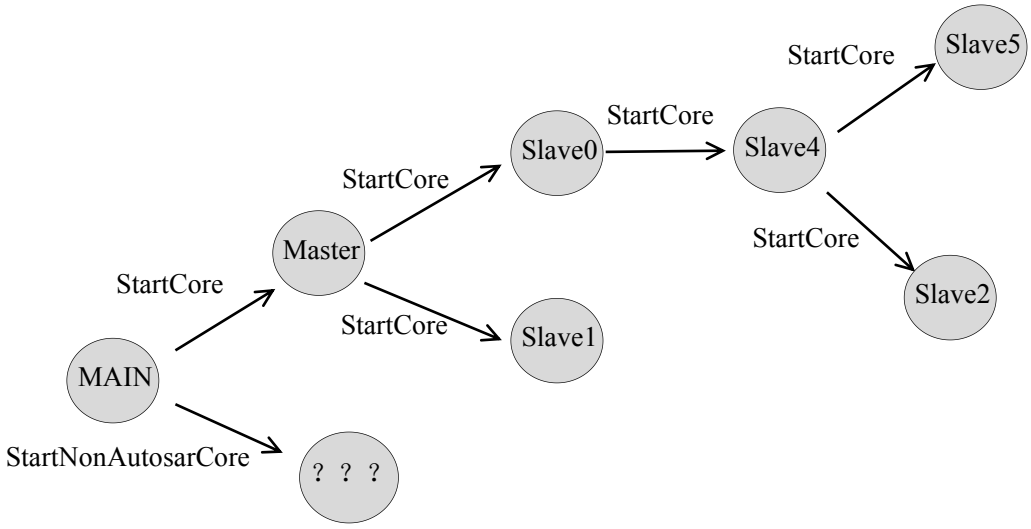

图 5-21 多核启动拓扑图

上图中的 MAIN 是用户程序的入口函数，在该入口函数中，能够启动主核（Master），或者使用 StartNonAutosarCore 来启动非 AUTOSAR 控制之下的其他核心。主核一旦启动，它就能够激活多个从核，如上图的 Slave1 和 Slave0 都是由主核启动的，而 Slave0 也能够激活多个从核，如 Slave4。通过上图能够看到一个问题，上面的拓扑图中启动了 Slave0~Slave2、Slave4~Slave5，并没有发现从核 Slave3。这是因为，DeCore MOS 是基于配置的操作系统，它的多核启动也可以通过配置来指定启动顺序、指定主核以及需要启动的核心集合（Core Set）等。这里并没有配置 Slave3 从核，也就没有启动该从核，一旦系统运行，就不能在运行时动态启动该从核了。

当核心被激活之后，它通过读取启动配置表 OSEKConfig_StartupCore 来判断它需要继续激活哪些核心，当激活完这些核心之后，将会调用 StartOS 来进行本地核心的初始化工作。从上文可以看到，虽然激活核心的顺序可以通过配置表来指定，能够按照既定的初始化轨迹依次执行，但这些核心之间的初始化的先后顺序并不是确定的，比如上图 5-21，主核激活完 Slave0 和 Slave1 后，Slave0 继续激活 Slave4，此时不能断定 Slave1 一定在 Slave4 之前执行内核初始化代码。为了使多核启动能够更加具有确定性，DeCore MOS 采用了两阶段（Two Phases）启动同步方式，即多核心的启动分为两个阶段，以每个阶段结束为同步点，多核心共同推进。见图 5-22。

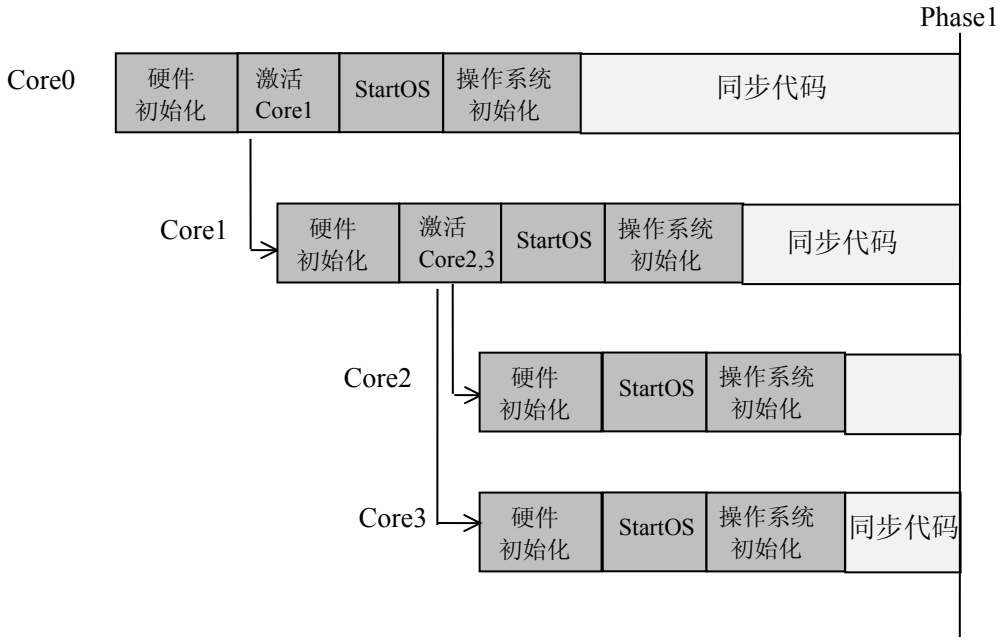

图 5-22 多核同步阶段 1

上图显示了 DeCore MOS 的第一阶段初始化同步，其中主核为 Core0，它首先激活了 Core1，然后 Core1 激活 Core2 和 Core3。由于激活顺序的原因，核心间有着一定的时间间隔，因此当完成操作系统初始化的时候，每个核心都将进入同步代码，同步地等待所有核心执行完第一阶段代码以后，共同推进到第二阶段的执行。在第二阶段主要完成用户和系统的启动钩子函数（HOOK）的调用，见图 5-23。

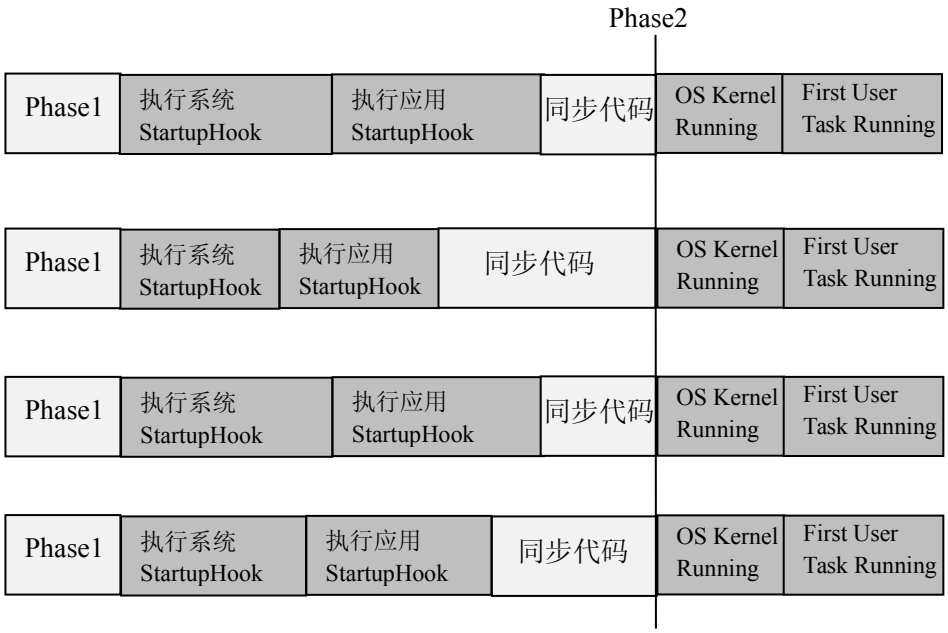

图 5-23 多核同步阶段 2

由于每个核心的启动钩子函数不一定都是相同的，所以他们的执行时间也并不一定一致，因此二阶段同步主要同步 StartupHook 函数的执行，当所有核心都执行完了应用启动钩子则会执行同步代码以等待所有核心完该阶段操作。

经过了 Phase1 和 Phase2 的同步阶段以后，所有核心同时调度第一个用户任务运行，即完成了 DeCore MOS 的多核启动。在这里发现同步代码起着至关重要的作用，它的主要实现机制是使用自旋锁与共享变量（Shared Variable）。系统内部使用 OsekExecution_PhaseSpinCode 来完成相关同步操作，这里以阶段 1 的同步为例。其同步代码如下所示。

```
void OsekExecution_PhaseSpinCode(){
    BSP_GetSpinLock(SPIN_LOCK_OS_SETUP);
    osekExecution_wait1_flag--;
    BSP_ReleaseSpinlock(SPIN_LOCK_OS_SETUP); //去锁
    while(osekExecution_wait_flag!=0) {
        //同步;
    }
}
```

其中 SPIN_LOCK_OS_SETUP 是系统在多核启动时所需要使用的自旋锁编号，osekExecution_wait1_flag 是共享变量，该变量初始值为启动的核心总数（包括主核和从核），每次当某一个核心执行完第一阶段代码时，都将此变量进行自减，在此变量自减之前需要加以自旋锁保护，以免产生同步性问题。之后代码进入自旋阶段，不断的检查该共享变量是否为 0，也即检查是否所有核心都已经完成了第一阶段初始化代码。当所有核心都以完成第一阶段初始化代码，则函数返回，进入第二阶段同步。

5.5.2 同步停止

DeCore MOS 的系统停止概念分为两种，其一是独立停止（Individual Shutdown）；其二是同步停止（Synchronized Shutdown）。独立停止对应于系统 API 的 ShutdownOS，同步停止则为 ShutdownAllCores。无论使用哪种停止的概念，最终都会导致某个或多个核心进入 CORE_STATE_SHUTDOWN 状态，并且进入无限循环（或低功耗模式）。对于独立停止，它只能由本核心调用，并且只能够关闭本核心的系统运行。而同步停止则是可以被任何已经启动的核心调用（主核或从核），它将给所有启动运行的核心发送广播消息（Broadcast Message），每个核心收到广播消息之后，运行独立停止并且运行同步代码以同步每个核心 ShutdownHook 的调用

执行。见图 5-24。

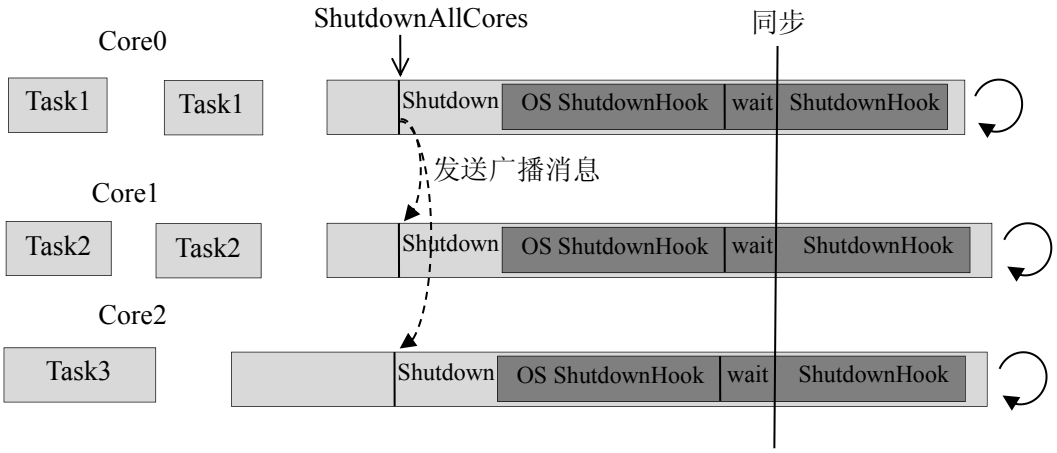

图 5-24 多核停止同步示意图

从上图中可以看到，Core0 的 Task1 任务调用了 ShutdownAllCores 函数，该函数向 Core1 和 Core2 发送广播消息以通知操作系统停止。Core1 和 Core2 接收到该广播事件以后调用 ShutdownOS 已完成独立停止。ShutdownOS 中执行操作系统的停止代码，如关闭 COM 通信接口、释放自旋锁、内核状态转换等。当操作系统停止代码执行完毕则调用同步代码（wait），该同步代码与上一小结中的同步代码类似。所有核心达到同步点之后，开始执行用户的停止钩子函数（ShutdownHook），最后每个核心都将进入到无限循环当中，不再执行任何调度和中断处理。

从上文的分析当中，可知 ShutdownAllCores 最为关键的一个执行步骤是发送广播消息到其他核心，该步骤的实现依赖于 DeCore MOS 的多核消息邮箱机制，这里主要说明 ShutdownAllCores 是如何使用该机制实现的。

```
void ShutdownAllCores(StatusType error){
    ...
    OSDWORD tmp;
    CoreIdType coreID = GetCoreID();
    struct Message * msg;
    for(tmp = 0;tmp < ACTIVE_CORE_NUM;tmp++){
        if(tmp == coreID) continue;
        if(OSEKConfig_StartupCore[i][0] !=CORE_STATE_SHUTDOWN){
            msg = MSG_POOL_GET();
            msg->from = coreID;
            msg->event_code = IPC_EVENT_SHUTDOWN;
            BSP_SendMessage(msg,tmp,IPC_NOTIFICATION);
        }
    }
}
```

以上代码是 ShutdownAllCores 函数的部分实现代码，它能够清楚地说明如何使用多核消息邮箱机制来完成广播消息的发送。代码首先获得本地的核心 ID，然后读取配置表以获得每一个已经启动的核心 ID，通过循环操作，每次从消息池中获得一个消息对象，然后填充消息对象相关的域，其中包括当前的核心 ID 和事件码，系统中将多核停止消息的事件码定义为 IPC_EVENT_SHUTDOWN，最后调用 BSP_SendMessage，将消息发送到指定核心，要注意的是 BSP_SendMessage 的 IPC_NOTIFICATIONS 参数表示使用多核消息邮箱的通知机制，即产生核间中断。

在多核同步停止的过程当中可能会出现其他核心关中断的情况，比如 Core0 给 Core2 发送系统停止事件，但此时 Core2 为了本地的同步要求关闭了本地中断，即 Core0 的事件虽然已经挂接到了对应的消息邮箱当中，但是并没有真正中断 Core2 正在执行的任务。这会导致所有接收到关闭消息的核心一直处于等待同步状态而不能执行用户的钩子程序，此时系统并没有真正的关闭。DeCore MOS 的解决方式是在每次时钟中断处理中都要判断消息邮箱里是否收到关闭事件。如果收到了关闭事件，那么会调用消息的回调函数，它马上激活关闭代理任务（Shutdown Proxy Task），该任务属于系统服务任务（类似 Idle 任务），并且具有最高优先级。当时钟中断处理完成以后，该关闭代理将会执行 ShutdownOS 函数。值得说明的是，即使 Core2 没有关闭中断，在核间中断处理程序中也将会激活该代理任务。

5.7 本章小结

本章我们主要介绍了 DeCore MOS 内核的详细设计，我们从任务管理开始，介绍了 DeCore MOS 采用的基于优先级可抢占式调度方式，并详细说明了它的队列管理，包括优先级队列、事件等待队列、待激活队列以及终结队列。然后我们讨论了 DeCore MOS 所支持的内存管理方式以及在多核环境下的计数报警器机制，在原有计数器基础之上，系统采用了时间轮方式来计算报警触发条件，这种优化能够减少中断的处理时间。在事件控制机制里，我们重点说明了 WaitEvent 和 SetEvent 这两个函数的实现流程。基于计数器和任务管理以及事件控制机制，系统又引入了多核调度表，使用多核调度表能够控制每一个终结点的执行，从而使任务的激活和事件设置更具有确定性。最后本章讨论了在多核环境下操作系统内核的启动和停止，为了保证每个核心的同步性，系统采用了两阶段同步启动方式和广播关闭方式实现多核心的同步启停。

第六章 DeCore MOS 的配置与测试

面向汽车电子行业的嵌入式操作系统的一大特征就是它的高可配置性和高剪裁性。几乎所有的模块都可以进行静态配置，并且由于规范对具体的配置接口信息进行了详细的定义和描述，所以能够很容易做到配置标准化，这也更易于进行代码移植和配置工具的制作。除此之外，配置管理还应该进行一定的代码静态校验、提供操作系统生成器等内容。DeCore MOS 的配置接口完全参考 AUTOSAR OS 5.2.0 和 OSEK OS 2.2.3 规范，为用户提供统一、简捷的配置环境为目标，开发了 DeCore Turbo 配置工具，该工具能够通过图形化界面配置，从而生成 DeCore MOS 所需的实体以及代码。

DeCore MOS 的测试目前没有按照 AUTOSAR 相关测试集[30]进行测试，但是 DeCore MOS 对每一部分模块都进行了独立测试，也进行了一定量集成测试，测试结果表明，DeCore MOS 能够满足一般应用需要。

6.1 DeCore MOS 的配置

DeCore MOS 是基于配置的操作系统，所有的模块和子系统都在一定程度上都支持静态配置。这里需要澄清配置与剪裁的区别。在 DeCore MOS 中，剪裁是属于配置所要达到的其中的一个目的，配置包括了剪裁，但不限于剪裁。比如可以通过配置 OCC_RESOURCE 宏以开启操作资源管理功能，如果不进行配置则将该模块剪裁掉，这是属于模块剪裁范畴。除此之外，对于定义任务、定义调度表、定义报警器等内容，是属于操作系统的实体元素配置，与剪裁有着微妙的区别。

配置管理覆盖了多核支持层以及内核层。其中每个模块都包括若干配置实体和剪裁功能宏，参考下表 6-1 所示内容。

表 6-1 配置管理配置实体一览表

模块	说明	文件位置
任务配置	任务句柄，堆栈空间，优先级，任务类型，所属核心，共享映射表等。	config/cfg_task.c config/cfg_task.h
调用钩子	启动和停止钩子等以及 hook 剪裁相关宏定义。	config/cfg_hook.h config/cfg_hook.c
报警计数器	计数器所属核心，计数器模数，增长值等。报警器所属计数器，执行操作，时间轮 item 数等。自启动等相关内容。	config/cfg_alarm.c config/cfg_alarm.h
调度表	调度表所属核心，终结点数，每个终结点的执行操作和时钟偏移值等。	config/cfg_schedtable.h config/cfg_schedtable.c
启停控制	启动链配置，启动的核心数，非 AUTOSSAR 核心等。	config/cfg_execute.h config/cfg_execute.c
自旋锁	用户自旋锁静态申请配置。	config/cfg_spinlock.h config/cfg_spinlock.c
核间通信	其中包括了核间通信事件的设置，RPC 跨核函数自动生成配置等。	config/cfg_ipc.h cfg_rpc.h config/cfg_ipc.c cfg_rpc.c
资源配置	与 DeCore OS 一致。如调度器资源、中断资源等	config/cfg_resource.h config/cfg_resource.c
内存管理	每个核心的分区最大值，消息块和队列块的分配数量等。	config/cfg_mem.h config/cfg_mem.c
OSEK COM	与 DeCore OS 一致	config/cfg_com.h comfig/cfg_com.c
安全相关	暂未实现	config/cfg_protected.h confg/cfg_protected.c

上表中的文件位置是相对于操作系统根目录而言，于此同时为了便于配置信息的统一管理，定义了 cfg.h 和 cfg.c 两个文件，每个系统模块的头文件都要包含 cfg.h。从上表中也可以看到，各个模块的配置文件划分的非常清楚，这能够使得配置更加的清晰，对后期扩展系统功能和调试都有很大益处。

在第二章中，讨论过 OSEK/VDX 的符合级别，它包括 BCC1、BCC2、ECC1 ECC2，在 AUTOSAR 规范下，也定义了 4 种可调配级别（Scalability Classes），分

别为 SC1、SC2、SC3、SC4。在不同的可调配级别下，操作系统具有不同的功能集。见表 6-2。

表中清晰的显示了对应每一个可调配级别都应该具有哪些功能集合，可以看到无论哪个调配级别都应该至少支持 OSEK/VDX 的所有符合级别，而这些符合级别所能够支持的操作 DeCore MOS 都应该支持，目前 DeCore MOS 仅满足 SC1 级别，因此在配置工具中，只有 SC1 符合级别才有效，这也是默认的配置。

SC1 级别是属于实时内核基础，它往往不考虑安全性，只尽最大可能满足实时性要求。而 SC1 往上，系统首要关注的就是安全性问题，因此它包含了堆栈监测，内存保护等安全性相关的内容。

表 6-2 AUTOSAR 可调配级别功能表

Feature	**Scalability Class 1**	**Scalability Class 2**	**Scalability Class 3**	**Scalability Class 4**
BCC1 BCC2 ECC1 ECC2	√	√	√	√
Counter Interface（计数器接口）	√	√	√	√
SWFRT Interface（自由软件定时器）	√	√	√	√
Schedule Tables（调度表）	√	√	√	√
Stack Monitoring（堆栈监测）	√	√	√	√
Protection Hook（保护钩子）		√		√
Timing Protection（时序保护）		√		√
Global Time（全局时钟）		√		√
Memory Protection（内存保护）			√	√
OS-Applications（OS 应用概念）			√	√
Service Protection（服务保护）			√	√
Call Trusted Function（可信函数调用）			√	√

由于配置管理的内容较多，虽然都是一些实现简单、定义规范的宏、变量、函数等，但却又是整个系统所不可或缺的。为了节省篇幅，以下小节仅介绍 DeCore MOS 的核间通信和调度表的相关配置。

6.1.1 核间通信配置

多核通信的配置分为两个部分，其一是多核消息邮箱的相关配置，其二是 RPC 跨核函数生成配置。RPC 跨核函数的配置使用其他工具自动生成，这里主要介绍消息邮箱的配置。通过前文介绍可知消息邮箱机制能够按照不同的事件进行分类，因此需要支持事件的配置以及对事件作出相应的回调函数，同时邮箱队列最大长度通过 DeCore Turbo 就可以完成。如图 6-1 所示。

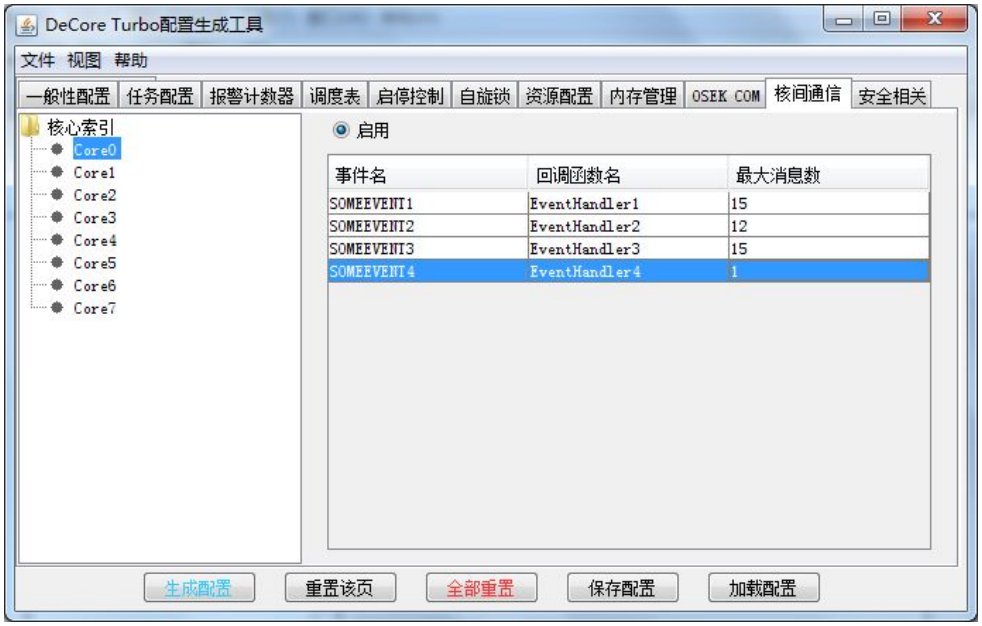

图 6-1 核间通信配置界面

核间通信的配置包括事件名称、回调函数以及最大消息数目。事件名称不同于事件码，事件码用于唯一标识，事件名称是一种更加充满语义的表述，配置管理将根据事件名称生成事件码，由于系统内部已经使用了若干事件码，因此用户事件码是从其他位置开始依次编号的。回调函数也是需要配置函数名，配置管理将会生成标准的函数定义和待用户实现的空函数，用户需要根据具体回调需要自行编写相对应的函数代码。最大消息数是消息队列中对应某一条消息链的最大值，如果超过这个最大值，消息邮箱将不会继续接收任何消息。

消息队列是每个核心都独立拥有的数据结构，因此进行核心的独立配置是至关重要的，上图 6-1 左侧的树形控件列出了 8 个核心，用户可以对每一个核心的核间通信机制进行独立配置。

6.1.2 调度表配置

调度表配置相对消息机制，它更加复杂。每个核心都可以独立拥有若干个调度表，每个调度表又可以进行相互的关联，而调度表又由若干终结点组成。按照调度配置表的数据结构要求，对应的配置应该具有如下图 6-2 所示的内容。

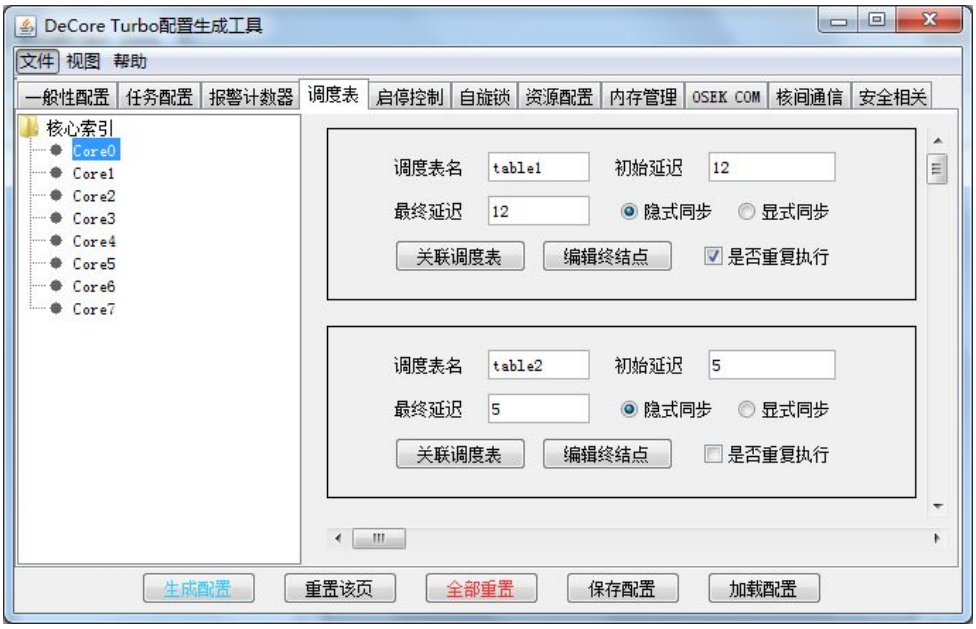

图 6-2 调度表配置界面

与核间通信一样，左侧依然可以对不同核心进行单独配置，右侧可以新建多个调度表，它包括初始延迟和最终延迟的配置，由于通过其他参数可以算出调度表持续时间，因此该项不予以配置。其中显式同步与隐式同步目前只能配置实用隐式同步。重复执行能够配置调度表是否能够按照循环周期重复执行调度。关联调度表能够关联到所有已配置的调度表，使其产生级联关系。编辑终结点用来添加和设定对应表中的终结点，由于篇幅所限，在此不详细说明终结点的配置，但其配置始终对应于 ExpiryPoint 结构体，如偏移值、任务和事件集合等。

6.2 DeCore MOS 的测试

为了测试 DeCore MOS 能否达到预期设计目标，大量的测试用例是必不可少的。DeCore MOS 为每一个模块都设计了一系列独立的模块测试用例，其覆盖任务

管理、核间通信、系统启停、计数报警器等。为了节省篇幅，本节仅重点介绍若干较为重要的测试用例和测试结果。

6.2.1 跨核任务激活测试

任务激活测试主要测试多核环境下，不同的核心的任务是否能够进行激活。主要调用 ActivateTask 函数，该函数相关 API 配置见下表 6-2 所示。

表 6-2 ActivateTask 函数描述

函数原型	StatusType ActivateTask(TaskType taskId)
输入	taskId－任务 ID 号
输出	1) 成功返回 E_OK 2) 超过规定的激活次数返回 E_OS_LIMIT 3) 【无效的任务 ID 返回 E_OS_ID】
功能描述	将指定任务从挂起状态转变为就绪状态
多核支持说明	1. 扩展 2. 支持跨核操作。如果激活其他核心上的任务，必须等待其他核心上的操作完成后才能继续其他操作
补充说明	在任务和 ISR2 中都可以使用该 API

为了测试该函数在多核环境下能否正常调用（主要测试跨核调用该函数，而不是本地调用），准备了如下表 6-3 的测试用例。

表 6-3 任务激活测试用例表

测试目的	测试场景	预期输出
测试 ActivateTask 成功激活其他核心上的任务和测试 GetTaskState 成功获取其他核心上任务的状态	核 0 配置两个任务 C0_Task1 和 C0_Task2。核 1 配置一个任务 C1_Task1。在配置中 C0_Task1 采用延迟 5 毫秒的启动方式。在任务开始之前，在 C0_Task1 上使用 API 分别获取其他任务的状态，然后再使用 ActivateTask 来启动 C0_Task2 和 C1_Task1 。在 C0_Task2 和 C1_Task1 也分别使用 API GetTaskState 来获取三个任务的状态。以判断是否激活。	任务 C0_Task1 可以激活任务 C0_Task2 和任务 C1_Task1。具体表现为在 C0_Task1 中获取任务状态分别为运行，挂起，挂起。任务 C1_Task1 和 C0_Task2 中获取任务状态为挂起，运行，运行。

由测试场景可知，需要在配置管理中生成三个不同核心的任务。Core0 具有两个任务，core1 具有一个任务。其中 Core0 中的两个任务有一个延迟启动，而其他任务都暂且不进行启动，通过自启动任务来激活其他两个任务，并且在激活前调用 GetTaskState 来获得三个任务的状态，激活后也通过该 API 获得任务状态，以前后做对比可以判断是否能够真正的激活任务（包括跨核任务和本地任务），具体配

置细节不在这里展开。在 DeCore MOS 中，每个函数的定义都是使用 TASK 宏，主要代码如下所示。

```
TASK(C0_Task1)
{
	StatusType		status;
	TaskStateType	state1;
	TaskStateType	state2;
	TaskStateType	state3;

	/****start user code******/
	printf("*******C0_Task1 is Running*******    \n");
	GetTaskState(Task1,    &state1 );
	printf("*******the state of C0_Task1 is %d\n",state1);

	GetTaskState(Task2,    &state2 );
	printf("*******the state of C0_Task2 is %d\n",state2);

	GetTaskState(Task3,    &state3 );
	printf("*******the state of C1_Task1 is %d\n",state3);

	printf("*******C0_Task1 will active C0_Task2 and C1_Task1***** \n");
	status= ActivateTask(Task2);
	sleep(10000);
	status= ActivateTask(Task3);
	/****end user code******/
	status = TerminateTask();
}
```

与此类似，C0_Task2 和 C1_Task1 也是调用 GetTaskState 来获得三个任务的状态，这里为了节省篇幅，只给出 C0_Task1 的主要代码。同时为了产生时钟延迟以激活任务 C0_Task1，还需要配置核心 0 上的计数报警器。并且在 ISR 中调用触发计数器的函数，参考如下代码。

```
ISR(TimingInt)
{
	//获取核心 ID
	CoreIDType core=GetCoreID();
	if(core == CORE0){
		CounterTrigger(Core0_SysCounter);
	}
}
```

在配置了 5 毫秒激活的任务 C0_Task1 以后，系统启动后 5 毫秒将会激活该任务运行，并依次激活其他任务。测试结果如下图所示。

```
[C66xx_0]  *******C0_Task1 is Running*******
[C66xx_0]  *******the state of task1 is 0
[C66xx_0]  *******the state of task2 is 3
[C66xx_0]  *******the state of task3 is 3
[C66xx_0]  *******C0_Task1 will active C0_Task2 and C1_Task1*****
[C66xx_0]  *******C0_Task2 is Running*******
[C66xx_0]  *******the state of task1 is 0
[C66xx_0]  *******the state of task2 is 0
[C66xx_0]  *******the state of task3 is 3
[C66xx_1]  *******C1_Task1 is Running*******
[C66xx_1]  *******the state of task1 is 3
[C66xx_1]  *******the state of task2 is 3
[C66xx_1]  *******the state of task3 is 0
```

图 6-3 任务激活测试结果

在这里要注意控制台输出的每行都有一个表示当前运行代码的 CPU 前缀，该前缀并不是程序的输出，而是 CCS5.5 平台自带的功能，它能够使清晰的看到代码执行情况。上图的测试结果表明，ActivateTask 函数能够正确的激活本地核心任务以及不同核心的任务，并且 GetTaskState 也能够正常工作。

6.2.2 跨核计数报警器测试

除了任务激活，计数报警器是内核最重要的模块之一，几乎所有应用都要使用该模块，因此对该模块的测试是至关重要的。其中相关的 API 较多，为了便于测试，这里只给出一个综合测试实例。

测试使用 Core0 和 Core1，每个核心都配置两个任务，其中 Core1 配备 3 个报警器。Core0 的报警器 C0_Alarm1 用来激活 Core1 的任务 C1_Task1，C0_Alarm2 用来激活 Core1 的任务 C1_Task2，前者使用 ActivateTask 直接激活，后者采用 SetEvent 方式。C0_Alarm3 用来激活本地的两个任务 C0_Task1 和 C0_Task2。这样无论报警器进行本地激活任务和设置事件，还是跨核激活其他核心的任务和设置其他核心的事件就都可以覆盖到测试当中。整个测试用例的测试场景图参见 6-4。

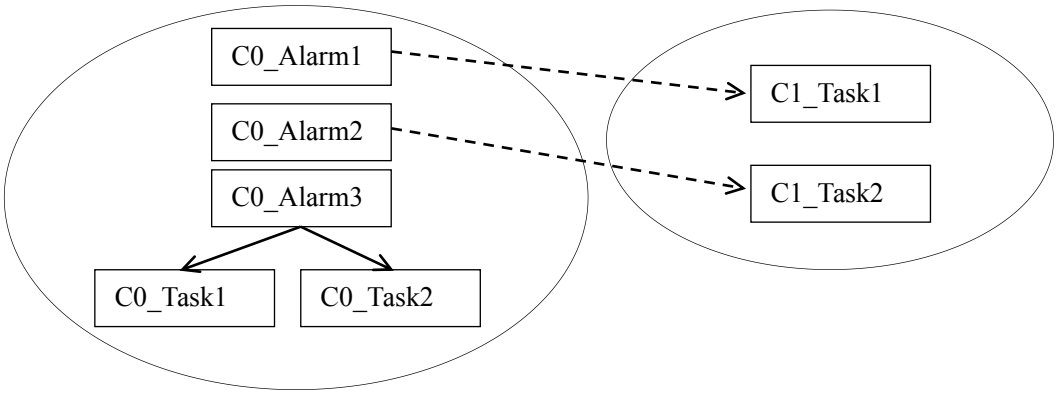

图 6-4 计数报警器测试场景

在系统启动时，C1_Task2 和 C0_Task2 都开始执行，分别启动 C0_Alarm1、C0_Alarm2 和 C1_Alarm3 报警器，之后等待事件 0x00000001 到来。当报警器依次被触发，则会按照图 6-4 所示的触发关系进行激活。主要代码如下所示。

```
TASK(C0_Task1){
	/****start user code******/
	printf("C0_Task1 has been activated\n");
	/****end user code******/
	status = TerminateTask();
}
TASK(C0_Task2){
	/****start user code******/
	printf("C0_Task2 is Running\n");
	SetRelAlarm(C1_Alarm3,1000,0); printf("C0_Task2 waits for event\n");
	WaitEvent(0x00000001,SINGLE_TRIGGER_TYPE);
	printf("C0_Task2 has been set event\n");
	/****end user code******/
	status = TerminateTask();
}
TASK(C1_Task1){
	/****start user code******/
	printf("C1_Task1 has been activated\n");
	/****end user code******/
	status = TerminateTask();
}
TASK(C1_Task2){
	/****start user code******/
	printf("C1_Task2 is Running\n");
	SetRelAlarm(C0_Alarm1,1000,0);
	SetRelAlarm(C0_Alarm2,1000,0); printf("C1_Task2 waits for event\n");
	WaitEvent(0x00000001,SINGLE_TRIGGER_TYPE);
	printf("C1_Task2 has been set event\n");
	/****end user code******/
	status = TerminateTask();
}
```

上述代码省略了报警器和任务的相关配置，其中 SetRelAlarm 用来设定绝对时钟，其第二个参数表示 1000 个计数器“tick”，第三个参数设定为不重复执行。SetEvent 用来等待事件到来，这里使用但事件触发模式，即只有事件到来就触发任务激活。代码执行结果见下图 6-5 所示。

```
[C66xx_0]  *******C0_Task2 is Running*******
[C66xx_0]  C0_Task2 waits for event
[C66xx_1]  *******C1_Task2 is Running*******
[C66xx_1]  C1_Task2 waits for event
[C66xx_0]  C0_Task2 has been set event
[C66xx_0]  C0_Task1 has been activated
[C66xx_1]  C1_Task1 has been activated
[C66xx_1]  C1_Task2 has been set event
```

图 6-5 计数报警器测试用例测试结果

从上述测试结果可以看到，所有任务都按照预期正确的执行了，不过需要注意的是，C1_Task1 和和 C2_Task1 的 执行顺序是不确定的，这是因为他们的报警器的设定是由在两个不同核心上的任务所设定，由于核心彼此独立执行，不可能确定哪个核心一定要先于哪个核心执行（在没有同步机制参与的情况下）。

6.2.3 多核启停测试

测试用例一定是基于操作启停模块之上运行的，不可能操作系统没有启动就能够进行任务管理或者计数报警器等模块的测试。因此以上所介绍的测试也囊括了多核的启动测试。但这里没有调用 ShutdownOS 和 ShutdownAllCores 函数，为了让用户代码正确执行完毕后，使操作系统能够顺利平缓的关闭，所以测试多核的停止功能也是一项必不可少的测试。

在用例准备上，为了保证能够覆盖到每个核心，因此在所有核心都安排了任务，每个任务只做无意义的打印输出，其中 Core4~Core7 的任务最先调用 ShutdownOS，将本地核心关闭。之后 Core0 调用 ShutdownAllCores 将所有核心都进行关闭。

因此用例的预期输出应该是比较明显的，即所有核心都有输出，当依次关闭本地核心时，本地核心的任务不会继续输出，当关闭所有核心的时候，将不会有任何输出行为。具体代码参考如下。

```
TASK(C0_Task)
{
    printf("*******C0_Task is running*******\n");
    sleep(3000);
    printf("*******All cores will be shutdown*******\n");
    shutdownAllCores();
    /****end user code******/
    status = TerminateTask();
}
...
TASK(C3_Task)
{
    while(1){
        printf("*******C3_Task is running*******\n");
        sleep(1000);
    }
    /****理论上不会执行到这里****/
    status = TerminateTask();
}
TASK(C4_Task)
{
    while(1){
        printf("*******C4_Task is running*******\n");
        sleep(1000);
        printf("*******Core4 will be shutdown*******\n");
        shutdownOS();
    }
    /****理论上不会执行到这里****/
    status = TerminateTask();
}
….
TASK(C7_Task)
{
    while(1){
        printf("*******C7_Task is running*******\n");
        sleep(1000);
        printf("*******Core7 will be shutdown*******\n");
        shutdownOS();
    }
    /****理论上不会执行到这里****/
    status = TerminateTask();
}
```

这里同样省略了配置相关的内容，C0_Task 负责调用 ShutdownAllCores，C4_Task~C7_Task 本地调用 ShutdownOS()，这里为了节省篇幅只写了 C7_Task 和 C4_Task 的代码，其他核心的代码都是相同的。下面来看一下测试结果，与前文的用例类似，由于不同核心的执行顺序以及打印输出顺序都有所不同，因此不能期望所有核心的输出都是按照代码编写的顺序进行的。

```
[C66xx_3]  *******C3_Task is Running*******
[C66xx_1]  *******C1_Task is Running*******
[C66xx_2]  *******C2_Task is Running*******
[C66xx_6]  *******C6_Task is Running*******
[C66xx_6]  *******Core6 will be shutdown*******
[C66xx_7]  *******Core7 will be shutdown*******
[C66xx_4]  *******C4_Task is Running*******
[C66xx_7]  *******C7_Task is Running*******
[C66xx_4]  *******Core4 will be shutdown*******
[C66xx_5]  *******C5_Task is Running*******
[C66xx_5]  *******Core5 will be shutdown*******
[C66xx_2]  *******C2_Task is Running*******
[C66xx_3]  *******C3_Task is Running*******
[C66xx_1]  *******C1_Task is Running*******
[C66xx_1]  *******C1_Task is Running*******
[C66xx_3]  *******C3_Task is Running*******
[C66xx_0]  *******All cores will be shutdown*******
[C66xx_2]  *******C2_Task is Running*******
```

图 6-6 多核停止用例测试结果

以上代码每次运行的结果都是不同的，这点无需说明，只专注于系统是否正确关闭。可以看到 Core4~Core7 中某个调用者调用 shutdownOS 后，该核心就会处于关闭状态，因此也就只会在测试中输出一次。“Core5 will be shutdown”是 Core4~Core7 的最后一次输出，此后它们都处于关闭状态。此时的 Core1~Core3 仍然在执行。当 Core0 从 sleep()函数退出之后，Core0 将会马上调用 shutdownAllCores，它会将所有核心关闭（Core4~Core7 除外）。调用此函数之后，整个操作系统将处于停止状态，不再有任何显示。但可以发现在调用 ShutdownAllCores 之后仍然会有 C2_Task 的一次输出，这是 printf 函数内部缓冲的原因，并且因为多核间是异步执行的，调用 ShutdownAllCores 函数不能保证其他核心能够“立即关闭”。

6.3 本章小结

本章我们介绍了 DeCore MOS 的配置和测试。为了便于用户配置，一个好的图形化配置工具是必不可少的，每个模块都独立进行配置并生成单独的配置文件，这能够使配置更加清晰和可控。此外我们介绍了 DeCore MOS 的测试情况，其测试不仅限于任务激活、计数报警器、多核停止这三点。它还包括其他诸多方面，但以上的测试点是能够覆盖整个系统最核心的功能，这些能否正常运行并达到预期结果将会直接影响着整个系统的可用性。

第七章 全文总结与展望

本章对论文现阶段所做的工作和成果进行总结，并对以后需要继续展开的工作进行展望和设想，使其能够更加完善。

7.1 工作总结

DeCore OS 是电子科技大学嵌入式软件工程中心开发的符合 OSEK/VDX 规范认证的嵌入式实时操作系统，它主要针对汽车电子特定领域。随着硬件多核技术的深入发展，在汽车电子领域越来越多的使用多核处理器。为了能够使 DeCore OS 能够支持多核环境，本论文深入 AUTOSAR 操作系统规范的多核部分，对原有单核系统进行了多核化改造，不但能够复用原有单核功能，扩展支持了大部分本地函数，使其能够支持跨核调用，并且增加了调度表，内存管理等新内容。

总的来看论文从以下几个方面取得了一定成果：

（1）引入 RPC 机制，采用 RPC 机制使得整个系统能够使用统一的跨核调用模型。并且为用户提供良好的跨核调用接口。

（2）使用时间哈希方式实现多核计数报警器，以往的实现方式是采用循环扫描链表上的报警器，这会增加时间成本。使用时间轮能够减少这种时间成本。

（3）增加了调度表模块，DeCore OS 没有调度表机制，使用调度表能够静态配置每个终结点的启动顺序和启动时机，使应用更加灵活，更具有可分析性。

（4）增加内存管理功能，由于多核环境下对核间通信有着动态要求，因此必须提供一定的内存管理功能，DeCore MOS 采用固定分区管理方式，并且提供了消息对象的分配器和队列块分配器。能够满足一般情况下的使用。

（5）满足 AUTOSAR 规范要求的多核启停，由于用户每个核心的钩子程序有可能不同，并且不同核心的初始化或停止过程执行时间也一定一致，因此 DeCore MOS 参考并遵循了 AUTOSAR 规范提出的两阶段同步和广播停止。

7.2 工作展望

没有一款嵌入式实时操作系统是完美的，开发一款多核嵌入式实时操作系统

本身就是一件复杂的工程实践。除了 DeCore MOS 目前所完成和支持的功能以外，还具有扩展和优化的余地。下一步工作将会从以下几个方面展开：

（1） 增加调度表显示同步支持，优化调度表结构。

（2） 提供 IOC 机制。

（3） 提供内存保护、时序保护、应用保护等安全性相关的功能。

可以看到后续的工作除了改进和扩展调度表以外，支持 IOC 也是必要的工作，使用 IOC 机制能够给用户提供统一的核间通信接口，使不同核上的 SW-C 之间的通信具有底层透明性。安全性保护也是后续工作所必须的，它包括了时序保护和内存保护等内容。这些保护机制是 AUTOSAR SC1 级别以上所要求的，因此后续工作将主要围绕着系统安全方面展开。此外，不断地测试系统和提高系统稳定性也是后期工作所必不可少的。

致 谢

DeCore MOS 是实验室老师与同学们共同的研究成果，绝不是一个人能够独立完成的。感谢在工作过程中给予指导和帮助的 XX 教授、XX 教授以及 X 老师，老师们的悉心教导和不时的讨论并指点我正确的方向，使我在这些年中获益匪浅。感谢参与该项目的全体同学和学弟学妹们，向你们的无私帮助致以最真诚的致敬。

在我生病期间，我的室友闫龙同学每天都会过来照顾与陪伴，无论什么事情，多大或者多小，他都会不厌其烦的帮助我。同时 X 老师与 X 老师也会经常过来看望，你们的体贴与支持使我感觉到了家的温暖。

感谢施家琪同学这几年来的关心与技术的支持和讨论，每天一同去教研室，一同去食堂吃饭，形影不离，探讨技术，探讨未来。同时也感谢众位学长姐、学弟妹的共同砥砺，你们的陪伴让两年的研究生活变得绚丽多彩。

三年前我只身一人来到这个陌生的城市，没有一位朋友与同学。三年后我不但获得了知识，而且结交了很多优秀的同学与朋友，这使我得人生充满了意义与动力，是她们让我获得了前进的力量。时光转瞬即逝，这三年来有过欢笑，有过悲伤。无论怎样，也无论时间去了哪儿哪里，我都不会觉得后悔与惋惜。即使已然毕业，但前方的路途已然很遥远！

最后我要感谢我的父母和我的亲戚。正因为有了你们，我才能够毫无顾虑地顺利完成学业。

参考文献

[1]王嘉平.多核系统中实施调度算法和研究[D].中国优秀学位论文全文数据库，2012

[2]袁云.基于多核多线程处理器上任务调度技术研究[D].中国科学院研究生院,2006

[3]覃中.基于多核系统的线程调度[D]，电子科技大学，2009

[4]李志军.一种适用嵌入式系统的自适应动态内存管理方案[J].计算机技术与发展，2007

[6]申建晶.嵌入式多核实时操作系统研究及实现[D].电子科技大学，2011

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

[7]Jean J Labrosse.MicroC/OS-II:The Real Time Kernel[M].CMP Books,2002

[8]俞建德.支持异构多核的嵌入式实时操作系统SmartOSEK OS-M[D].中国优秀硕士学位论文全文数据库，2008

[9]电子科技大学ESE中心.DeCore OS技术内部文档，2011年10月

[10]AUTOSAR Administration.V5.2.0.AUTOSAR Specification of Opertion System[S].Sep 10th,2013

[11]AUTOSAR Administration.Specification of Communication[S],2005

[12]AUTOSAR Group.Specification of RTE[EB/OI],2011

[13] 网飙，胡苏太.通用多核产品技术现状.江南计算机技术研究所，计算机世界报，2006

[14] 王庆.面向嵌入式多核系统的并行程序优化技术研究[D].中国优秀博士论文全文数据库，2013

[15] OSEK Group.Version 2.2.3.OSEK/VDX Operation System[S]，Feb 17th,2005

[16] 胡国珍等.嵌入式RTOS优先级天花板协议研究[J].计算机工程与设计，2009

[17] 厉海燕等.优先级继承协议在Linux中的实现[J].计算机工程与设计，2005

[18] 申雪琴.计算机操作系统中死锁问题研究[J].计算机数字与工程，2008年

[19] Texas Instruments.SPRUGW0B.TMS320C66x DSP CorePac (User Guide)[OL].July 2011

[20] Texas Instruments.SPRUGR9E.KeyStone Architecture Multicore Navigator[OL],May 2012

[21] Texas Instruments.SPRUGW7A.KeyStone Architecture Multicore Shared Memory Controller [OL],October 2011

[22] Texas Instruments.SPRUGW4A. KeyStone Architechture Chip Interrupt Controller(CIC User Guide)[OL],March 2012

[23] Texas Instruments.SPRUGS3A.KeyStone Architechture Semaphore2 Hardware Module(User Guide)[OL],April 2012

[24] Texas Instruments.SPRUGV5A.KeyStone Arhitecture TIMER64P [OL].March 2012

[25] 彭正文，徐新爱.基于SMP的Liunx内核自旋锁分析[J].江西教育学院学报.2005

[26] 孔帅帅.基于嵌入式多核处理器的通信和中断问题的研究[D].电子科技大学优秀硕士论文，2011

[27] WolfgangMauerer.深入Linux内核架构[M]（郭旭）.人民邮电出版社，2010

[28] 罗蕾.嵌入式实时操作系统及应用开发[M].北京：北京航空航天大学出版社，2011

[29] 陈黎静.一种新的表插入算法[J].计算机技术与发展，2010

[30] 韩立宏.嵌入式实时操作系统性能测试方法[J].指挥控制与仿真.30(2)，2008

攻读硕士学位期间取得的成果

[1] 参与了总参谋部 57 研究所的基于 x86 VxWorks 协议转换网关项目，项目开发主要人员之一。负责相关代码的编写。
[2] 参与基于多核 DSP 处理器的嵌入式实时操作系统接口与中间件方法研究与技术实现，DeCore MOS 主要开发人员之一。