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 嵌入式多核环境下的板级支持系统设计与实现

学 科 专 业 计算机系统结构

学 号 201121060211

作 者 姓 名 施 家 琪

指 导 教 师 罗 蕾

分类号 _____ 密级 公 开

UDC [注1] _____

学 位 论 文

嵌入式多核环境下的板级支持系统设计与实现

（题名和副题名）

施 家 琪

（作者姓名）

指导教师 罗 蕾 教 授

电子科技大学 成 都

（姓名、职称、单位名称）

申请学位级别 硕士 学科专业 计算机系统结构

提交论文日期 2014.03.25 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 06 月 29 日

答辩委员会主席 _____

评阅人 _____

注 1：注明《国际十进分类法 UDC》的类号。

DESIGN AND IMPLEMENTATION OF A BOARD SUPPORT PACKAGE IN EMBEDDED MULTI-CORE ENVIRONMENT

A Master Thesis Submitted to

University of Electronic Science and Technology of China

Major:	**Computer Systems Organization**
Author:	**Shi Jia Qi**
Advisor:	**Prof. Luo Lei**
School:	**School of Computer Science & Engineering**

独创性声明

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

作者签名：_____ 日期：　　年　　月　　日

论文使用授权

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

（保密的学位论文在解密后应遵守此规定）

作者签名：_____ 导师签名：_____

日期：　　年　　月　　日

摘　要

消费电子产业及制造工业的高速发展对嵌入式系统的性能，规模，安全，系统利用率等方面提出了高要求。针对日益复杂的嵌入式系统，一系列应用于传统桌面及服务器领域的高新技术也开始逐步应用于嵌入式领域。多处理器技术，多核操作系统技术，虚拟化技术是传统桌面及服务器领域针对性能，功耗，系统利用率，安全性等关键指标而诞生的新技术。虽然这些技术在传统桌面及服务器领域已经日益成熟，但是面对嵌入式系统，这些新技术还需要进行许多的修改和优化。由于嵌入式平台的特点以及嵌入式软件及操作系统起步较晚，如何有效的发挥多核计算资源，如何保障多核嵌入式系统的正确性及安全性以及如何在庞大以及复杂的异构环境下协调各系统间的协作使之成为一个有机的整体。这些问题成为了目前在硬件成本不变的情况下如何提高嵌入式系统整体性能的关键。

本论文针对上述问题，以德州仪器最新的高性能 KeyStone 多处理器信号处理器平台为主要硬件研究对象，在深入分析其针对多核多操作系统的硬件技术的基础上，讨论针对该体系架构的多核多操作系统技术，设计并实现一套支持多操作系统同时运行的多核多操作系统板级支撑平台 Scorpius，并以应用两款经典嵌入式实时操作系统为例提出一种在嵌入式多核环境下同时运行多操作系统的方法，为同构或异构多核环境下的嵌入式应用提供系统级的支持。本论文的主要工作包括以下几个方面：

1．对目前嵌入式领域的多核处理器进行总结，并简单介绍一些嵌入式领域内的常见多核操作系统。研究并分析其他主流硬件架构针对多核等技术的实现细节。分析对比了以TMS320C6678为例的TI最新高性能KeyStone体系在多核上的实现。

2．设计针对 TMS320C6678 硬件的可同时运行多个单核或多核操作系统的嵌入式板级支撑平台 Scorpius 系统框架，包括实现基础硬件驱动服务，多核环境下所必要的基础组件以及提供给操作系统运行的操作系统必要组件。

3．采用层次化设计，将多核多操作系统基础平台与操作系统上层服务进行分离，针对 Scorpius 平台移植两款具体的经典嵌入式操作系统。

关键词：多处理器技术，嵌入式操作系统，板级支持平台，KeyStone 体系

ABSTARCT

The rapid development of the consumer electronics industry and the manufacturing industry increase the demand of embedded systems for performance, scalability, security, system utilization and other aspects. For increasingly complex embedded systems, a series of new technologies which are born for the key indicators of performance, power consumption, system utilization, security and so on in traditional desktops and servers environment. Although these technologies have become fully developed in traditional desktops and servers, yet these technologies still need to be modified and optimized for embedded systems. Due to the characteristics of embedded systems and the slow development of software and operating systems in embedded systems, how to effectively use multi-processor computing, how to ensure the correctness and the security of multi-processor embedded systems and how to coordinate the cooperation between systems in a complex heterogeneous environment become critical to how to improve the overall performance of embedded systems with hardware costs remaining unchanged.

To solve issues above, this thesis uses Texas Instruments' latest high-performance KeyStone multi-core digital signal processor platform and discusses the multi-core and heterogeneous multiple operating system architecture technology on the basis of analysis in the technologies supported by KeyStone architecture. It also discusses the design and implementation of a multi-core embedded system framework which supports both asymmetric multiprocessing and symmetric multiprocessing architecture. Finally it gives a method to provide system-level support for running more than one operating system simultaneously and supports embedded applications in homogeneous or heterogeneous multi-core environment by applying two traditional embedded operating systems to the platform called Scorpius. The core work of the thesis can be generalized as following:

1. This thesis firstly discusses the most recent multi-core processors used in embedded systems. Then it gives a simple introduction to some embedded multi-core operating systems. Finally it examines and analyzes the implementation details of multi-processor technology in KeyStone architecture and other CPU architectures and gives a comprehensive analysis of TMS320C6678 which is a member of the KeyStone.

2. This thesis also designs an embedded system platform Scorpius which supports

running more than one operating system simultaneously on cores which can be grouped in asymmetric multiprocessing mode or symmetric multiprocessing mode. The core work includes implementations of hardware drivers for TMS320C6678, basic components which are necessary for multi-core runtime and operating system supporting components for operating system which runs on the Scorpius platform.

3. The Scorpius platform in the thesis designed the “Multi-core Supporting Layer”, which separates the basic platform and the multi-core operating systems which run on it. The thesis also transplanted two traditional embedded operating systems on it.

Keywords: multi-processor technology, embedded operating system, board support platform, KeyStone architecture

目 录

第一章 绪论

随着集成电路技术，微电子技术以及片上系统（System on Chip，简称 SoC）技术的飞速发展，传统嵌入式处理器发生了新的变化。嵌入式应用规模的复杂化以及功能的多元化，嵌入式处理器已经从面向控制为主的 8 位单片机发展成了具有多个核心的面向人工智能，图像音频处理，互联通信等多领域的 32 位甚至 64 位处理器。高速发展的嵌入式多核技术对传统嵌入式软件系统平台提出了巨大的挑战。板级支持包（Board Support Package，简称 BSP）在传统意义上仅仅是某种具体芯片的板级驱动，人们经常将 BSP 归于操作系统的一部分进行设计与实现[1]。为了实现代码重复利用，简化操作系统的移植，服务多核及虚拟化技术，传统板级支持包正向支撑所有硬件技术的综合系统软件平台发展。除此之外，许多成熟的经典嵌入式操作系统仅仅支持单核处理器，而一些支持多核处理器的操作系统又因为实时性和复杂性的原因不适合一些新兴工业领域[2]。然而重新设计，开发嵌入式操作系统具有设计周期长，成本高，灵活性差，测试验证困难等缺点，因此需要探索新的系统架构方法，在操作系统底层设计新的板级支持平台，从底层开始在不影响性能和保障多核运行正确性的基础上使经典嵌入式操作系统适应不同的嵌入式应用需求[3]。因此设计与开发一个支持多核多操作系统的综合性板级支持平台迫在眉睫。

1.1 多核环境下板级支持系统平台研究背景

在 1965 年归纳信息技术和芯片工艺进步速度的基础上，戈登·摩尔（Gordon Moore）提出半导体芯片上集成的晶体管和电阻数量将每年翻一倍[4]；1975 年摩尔又提出了补充和修正，芯片集成的晶体管将每两年翻一倍[5]。摩尔提出的摩尔定律同样适用于当前的嵌入式处理器领域[6]。目前嵌入式多核化技术的发展主要还是集中在与桌面应用相似的消费电子领域，在传统制造业以及对安全性和功耗都有较高要求的医疗电子及汽车电子领域，嵌入式系统的多核化道路还仅仅处于探索验证的阶段[7]。这些产业对嵌入式多核处理器及其相配套的软件资源的要求也十分特殊。如何平衡功耗与性能，如何保障系统的安全性以及如何提高软件的灵活性成为了面向这些产业的嵌入式多核技术发展所必须要解决的一系列关键问题。

目前应用于通用计算的多核嵌入式处理器还不能满足制造业对安全及功耗上的要求。甚至在一些诸如安防，监控，精密控制等对计算强度要求较大的特殊领域通用多核嵌入式处理器在性能方面还不及专业用途的数字信号处理器（Digital

Signal Processor，以下简称 DSP）或现场可编辑逻辑门阵列（Field Programmable Gate Array，以下简称 FPGA）。不仅如此，传统多核技术及多核操作系统的架构方式在这些特殊环境下都存在着许多的问题。面向消费电子的通用多核处理器都采用近似桌面及服务器处理器的架构技术。这些多核处理器往往使用一种被称为同构多核的架构方式。在对称多处理架构下，每个处理器的地位都是平等的，对资源的使用权限相同[8]。这种架构虽然具有高并行性和较低生产设计成本的优点，但是在系统吞吐量要求较高的场合却不实用。除此之外，由于采用同构多核的多核处理器中的每个核心都连接到了同一个共享主内存上，由于系统总线带宽有限，处理器数目将严重受限。为了面向更复杂的嵌入式环境需求，不少半导体厂商开始将眼光转向到曾被视为权宜之计的异构多核架构上。在同构架构出现之前，异构架构就已经应用于通用桌面及服务器平台上。但是最早的异构架构主要通过将多个物理处理器集成到同一块或多块主板上来实现，实际上当今的分布式系统就是这种架构的一种延伸。然而随着片上系统（System on Chip，简称 SoC）的发展，一些半导体厂商开始尝试将不同种类的处理器内核封装在一个硅片上，形成一个片上系统。甚至还出现了将不同用途的多种处理器内核封装在同一块硅片上的特殊领域片上系统。虽然异构架构要早于同构架构，但是针对异构的平台软件却发展缓慢，直到半导体厂商开始推出多款异构架构的嵌入式系统以及主要处理器架构厂商发表异构标准[9]后，这些配套软件资源才开始慢慢得到重视与发展。

另一方面，传统的板级支持包总是和单一嵌入式操作系统以及板级硬件环境紧密关联。这种紧耦合虽然适合传统单核微控制器，但是针对当前多核高频嵌入式处理器来说并不适合。多核处理器在多核技术上存在众多的共性，在底层实现上也大致相同，不仅如此，处理器的外设模块为了驱动能重复使用而大多依照标准来设计。在这种环境下，每次面对新的硬件环境就单独开发其对应的板级支持包实在显得多此一举。另一方面，嵌入式多核操作系统的系统移植往往数倍复杂于以往操作系统，而且虚拟化技术也正以迅猛姿态在嵌入式领域发力。为了支持这些新技术，为了缩短开发周期，减少“重造轮子”[10]，传统的板级支持包必然要脱离上层操作系统和下层具体硬件向高独立性，高可靠性，高可用性的独立系统软件平台发展。

本论文正是在这个背景下，着重归纳与研究现有嵌入式多核技术，然后在此基础上提出一种面向多种嵌入式处理器架构的板级系统软件平台，该平台通过强大的中间层分离多核硬件细节与上层嵌入式操作系统，完成同时在多核上运行多种单核或多核经典嵌入式操作系统的目的，节省了开发专用嵌入式多核操作系统，的成本和时间。

1.2 多核环境下板级支持系统平台研究现状

1.2.1 嵌入式多处理器片上系统发展现状

目前嵌入式多处理器片上系统主要分为，同构多核通用片上系统，异构多核通用片上系统以及其他由诸如 FPGA 或 DSP 等混合构成的多处理器片上系统。这些片上系统的处理器采用精简指令（Reduced Instruction Set Computing，以下简称 RISC）架构并且普遍主频都达到了 800MHZ 以上，由这些嵌入式设备多机组成的分布式集群性能已经接近单台通用服务器并且功耗仅只有后者的十分之一。

1.2.1.1 同构多核通用片上系统

同构多核通用片上系统是嵌入式领域发展最快的多核架构[11]。多数应用于该多核架构的嵌入式处理器采用 RISC 架构，以 ARM，PowerPC 和 MIPS 为主要代表。该多核架构的片上系统主要应用于以消费电子为主的通用计算领域。该架构的多核通用片上系统的特征是多核间除了一级缓存（Level 1 Cache，以下简称 L1 Cache）外共享其他内存资源和总线资源。如图 1-1 为 ARM 的 CortexA9 MP 架构的多核片上系统。

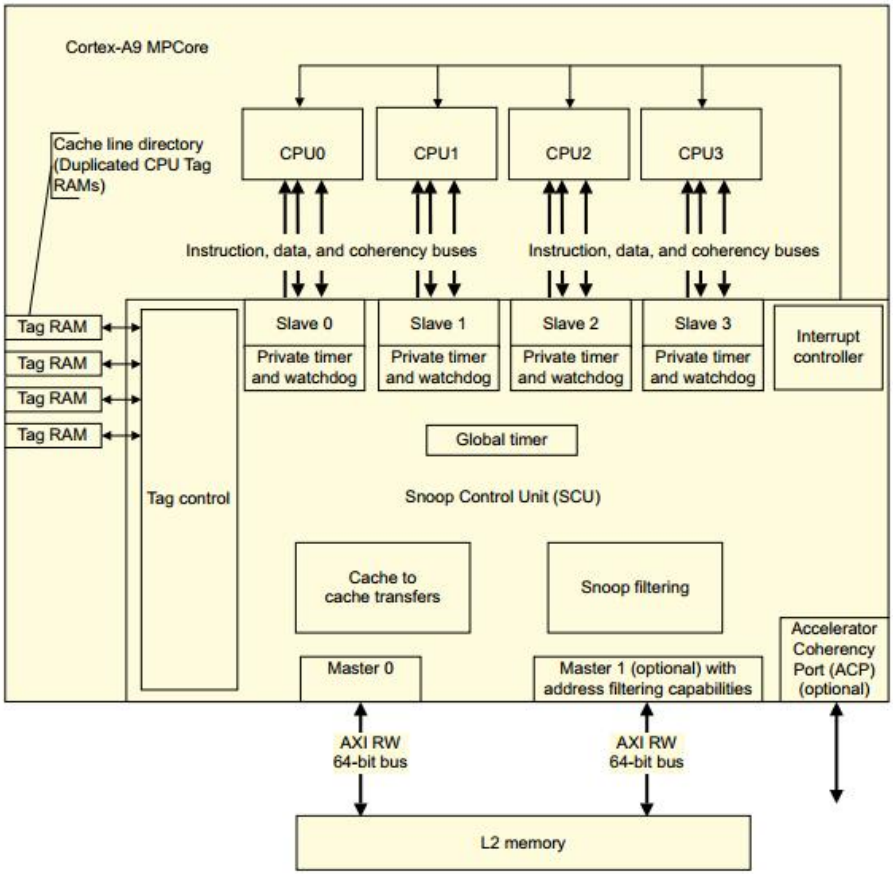

图1-1 ARM CortexA9 MP 架构框图[12]

采用同构多核架构的片上系统的主要优点如下：

1. 同构多核架构在处理器数量较少的情况下是一种吞吐量和成本兼得的优秀架构方案。

2. 对于共享数据，具有可伸缩的灵活特性。

3. 所有数据由所有处理器寻址，并由硬件监视逻辑保持数据的有效性。

4. 针对该架构的并行程序设计较为简单。

虽然采用同构多核架构的片上系统有以上许多优点，但是其架构的特征导致其仍然具有不少的局限性。其中由于所有核心共享总线而总线的带宽有限，因此处理器核心数目是受限的。除此之外，该架构由于多核独享一级缓存而又共享二级缓存（Level 2 Cache，以下简称 L2 Cache），随着核心数目的增多，多核间的缓存一致性（Cache Coherence）问题会严重影响系统吞吐率。

1.2.1.2 异构多核通用片上系统

应用于嵌入式的异构多处理器片上系统的发展是基于应用于传统服务器的异构系统。该架构通常将多个相同指令集架构但不同性能和功耗的内核捆绑在同一块硅片上，形成一种高低搭配的结构。其代表是广泛应用于工业领域的由德克萨斯仪器（Texas Instruments，简称 TI）设计生产的 Omap 平台。在最新一代的 Omap5 平台中，TI 将不同的 ARM 多核处理器 Cortex A15 多核与 Cortex M4 多核封装在一起。这种将相同指令集不同性能和功耗的内核捆绑在一起的做法可以有效的提高系统整体吞吐率。除此之外相较于将同构对称多处理架构那样将多个同种内核捆绑在一起的做法，这种架构方式可以有效降低功耗并降低实现多核间的 Cache 一致性问题的成本。

异构多核通用片上系统通常独享 L1 及 L2 Cache，主存采用非一致性内存访问（Non-uniform Memory Access，简称 NUMA）的设计方式[13]。为了解决多核之间的同步与共享问题，该架构除了指令实现的自旋锁外还设计了硬件实现的特殊自旋锁。有些实现还为多核间添加了共享内存以提高多核间通信的效率。图 1-2 为 ARMv7 中针对 CortexA7 与 CortexA15 设计的 big.LITTLE 技术的框架示意图，该技术预示着未来嵌入式多核架构的发展方向。

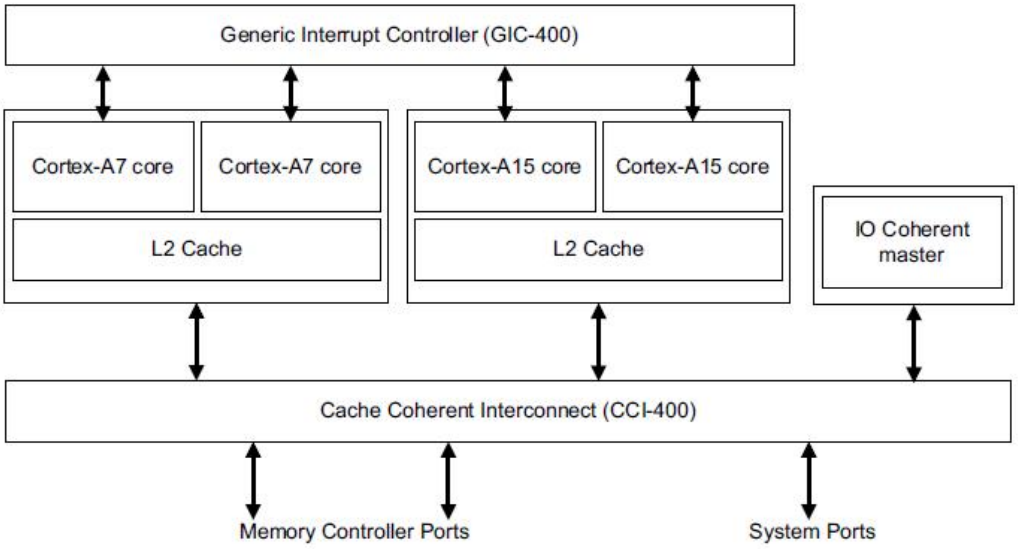

图1-2 ARM big.LITTLE 技术示意图[14]

1.2.1.3 其他多处理器非通用片上系统

随着片上系统技术的发展与日益增长的市场需求，人们在异构多核片上系统的基础上发展出了将多个不同指令集甚至不同功能的处理器芯片集成在同一片硅片上的全新架构。该片上系统通常将 DSP 或 FPGA 等非通用处理器和通用处理器进行混合搭配，将一些特殊应用交给非通用处理器来执行从而整体性提高系统的性能。该架构的典型产品包括德州仪器的达芬奇图像处理平台系列和赛灵思（Xilinx）的 ZYNQ 系列。前者如图 1-3 是将高性能图像处理 DSP 与 ARM 核相结合，后者则是将 FPGA 与 ARM 核相结合。这种架构突破了传统同构多核架构功能单一的局限，通过适才适用的方式在维持功耗不变甚至降低的基础上提高了系统在特殊领域的性能。

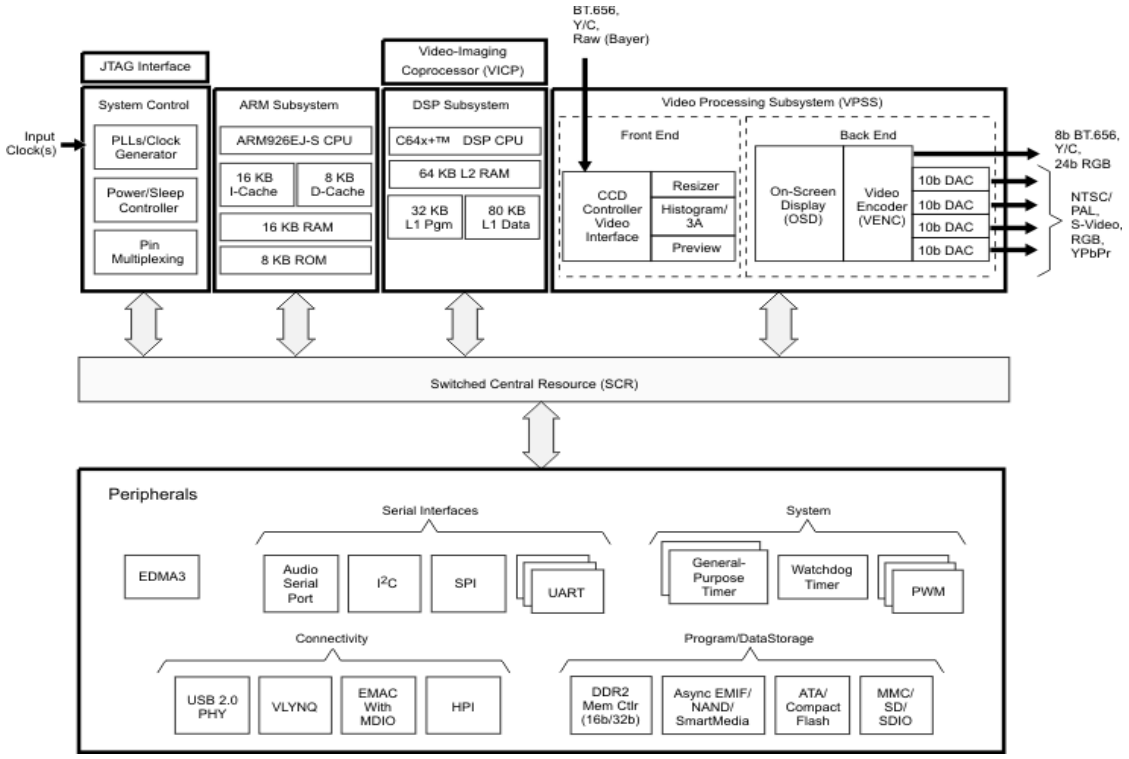

图1-3 TI TMS320DM6446 框架图[15]

1.2.2 多核嵌入式操作系统现状

多核处理器的应用离不开多核操作系统的支持，目前嵌入式领域的多核操作系统主要分为两个方面，一个是面向消费电子为主的通用多核操作系统，另一个是面向高安全，高实时要求行业的实时多核操作系统[16]。其中前者的发展要远比后者成熟。

1.2.2.1 面向通用领域的嵌入式多核操作系统

1. Linux

应用嵌入式的 Linux 是从桌面 Linux 发展而来的。Linux 内核是广泛应用嵌入式领域的支持多核的通用操作系统内核[17]。主线中的 Linux 内核支持多种嵌入式片上系统并针对多种 RISC 架构的处理器进行了深度的优化。其对同构多核的支持十分成熟。但是由于 Linux 最早起源于桌面，实时性较弱的问题制约了 Linux 在工业领域的发展。虽然不少针对 Linux 内核的实时补丁解决了这个问题，但是由于这些补丁对应的内核版本太旧，实时 Linux 内核并没有被主流市场所采用。

2. Windows CE

Windows Embedded Compact（简称 Windows CE）是微软公司开发的面向嵌入式的操作系统。Windows CE 的内核并非从桌面 Windows 内核修改缩减而来，而

是使用一套完全重新设计的核心。最新的 Windows CE7.0 开始支持多核嵌入式系统，主要支持平台为 ARM 的同构对称多处理器架构。由于编程上和传统 Windows 具有一定的相似性，针对 Windows CE 操作系统的开发人员数目庞大。Windows CE 目前主要应用于对实时性要求不高工业控制和部分消费电子领域。

1.2.2.2 面向工业控制领域的实时嵌入式多核操作系统

1. 具有多核扩展的 uCosII/III

uCos 是一种免费开放源代码，结构小巧，具有可剥夺实时内核的嵌入式操作系统。最早出自于 1992 年美国嵌入式专家 Jean J.Labrosse 在《嵌入式系统编程》杂志的文章连载[18]。uCosIII 是在 uCosII 的基础上发展而来的主要针对 32 位微控制器的实时嵌入式操作系统。虽然原生的 uCos 是不支持多核的，但是许多开发者发布了一系列针对 ARM 与 PowerPC 多核平台的 uCos 扩展。由于结构简单，功能不及商业嵌入式系统，目前 uCos 主要还是应用于要求不高的工业生成及实验教育领域。

2. VxWorks 6.x

VxWorks 是美国风河公司（Wind River）设计开发的面向高实时性需求的嵌入式实时操作系统。其良好的开发环境以及高性能，高稳定性的内核在嵌入式操作系统领域占据一席之地。

VxWorks6.x 之后开始支持同构及异构的嵌入式多核处理器。VxWorks 主要应用于通信，军事，航空航天等对实时性要求极高的领域[19]。由于 VxWorks 高额的费用，国内对其的应用与研究相对其他操作系统来说较少的多。

1.2.3 多核处理器与多核操作系统软硬件结合现状

嵌入式系统是操作系统，应用软件和硬件紧密结合的有机整体。目前人们提出了针对多核系统的三种主要系统架构模型。

1.2.3.1 对称多处理架构

对称多处理架构（Symmetric multiprocessing，简称 SMP）主要是指多个处理器同时运行一个操作系统实例。该架构的系统中操作系统内的大量数据被多个处理器共享。操作系统的某个任务可能在一个时间点处于某个核心运行而在另一个时间点处于其他别的核心运行。这种架构的优点是简化了多核操作系统的设计，使并行化粒度实现到进程任务级。但是该架构必须解决多核心在使用共享数据时出现的同步问题。该架构使用核间通信和自旋锁等方式来解决多核对共享数据同

时访问所出现的冲突。

1.2.3.2 非对称多处理架构

非对称多处理架构（Asymmetric Multiprocessing，简称 AMP）给开发者提供了一个与传统单核处理器系统类似的运行环境，这样有利于重复利用原有的软件资源。非对称多处理器架构下的多核处理器可以是同构的也可以是异构，其和对称多处理架构最大的区别是所有处理器内核运行了多个相同或不同的操作系统。多个操作系统之间使用特殊的通信协议进行数据交流，该架构的并行粒度实现在操作系统级别。这种架构的优点是简化了系统模型，减少了同步操作的性能损失。但是该架构相比对称多处理架构增加了系统编程人员的工作量，编程人员需要设计多个运行在不同操作系统上的应用程序。除此之外，对共享资源的访问仍然不可避免的需要锁的保护。

1.2.3.3 混合多处理架构

混合多处理架构（Bound Multiprocessing，简称 BMP）是结合了 SMP 资源管理和 AMP 应用控制的混合架构技术。该架构下的多处理器和 SMP 一样共享一套操作系统代码，但是应用程序却只能运行在某个特定的处理器中。在 BMP 中，锁定了一个处理器的应用是无法自发进行迁移的，这样虽然提高了处理器缓存的利用率，但是由于应用总是在非用户干预的情况下运行在同一个处理器不可避免会出现一定的处理器计算资源的浪费。这种架构的并行粒度定位在应用级别上。

1.3 多核多操作系统平台研究目的与意义

不同于混合多处理器架构在操作系统层面对 SMP 和 AMP 的混合，本论文研究的多核多操作系统板级支持平台从硬件角度出发，在操作系统之下实现一个抽象层从而将 SMP 和 AMP 进行混合。这样做的好处在于既可以重复利用传统的嵌入式单核或多核操作系统，又可以最大化系统整体的利用率。通过抽象的板级支持平台来将传统操作系统与新一代嵌入式多核处理器进行有机结合，做到软硬件协同配合。

1.4 论文主要内容及组织结构

本论文的内容和结构安排如下：

第一章绪论主要从处理器，操作系统和系统架构三个角度对当前嵌入式领域的多核技术进行介绍并简要说明该研究的背景和现状，明确研究的目标。

第二章讨论基于德州仪器 KeyStone 架构的多处理器片上系统，对该架构的多核技术进行深入的分析。然后针对该架构给出对称多处理，非对称多处理和混合多处理三种系统架构模型。

第三章着重讨论多核多操作系统平台 Scorpius 的整体框架设计，给出 Scorpius 所必需的软件模块。

第四章在第三章的基础上详细描述 Scorpius 关键模块的实现细节。

第五章描述移植针对 Scorpius 平台的操作系统移植技术，通过对 uCOSII 和 deCORE MOS 两种嵌入式操作系统的移植讨论了基于 Scorpius 平台的应用开发方法。

第六章为总结与展望，该章节总结了本文的工作并对 Scorpius 平台的不足和改进点进行分析。

第二章 KeyStone 体系的多核技术研究

2.1 KeyStone 多核信号处理器平台简介

德州仪器的 KeyStone 多核架构主要由 C66x 系列信号处理器（Digital Signal Processor，简称 DSP），ARM 通用内核以及其他特殊功能的协处理器组成，其设计最大限度地提高系统的整体吞吐量并消除设备性能瓶颈。KeyStone 架构的计算核心被称为 CorePac，每个 CorePac 内部包括了 DSP 核心，L1 Cache，L2 Cache 以及其他控制器与总线接口等。KeyStone 通过多个 CorePac 组成多核计算平台。每个 CorePac 通过 TeraNet 总线连接到外设接口控制器上。为了实现多个 CorePac 的共享内存访问，避免多 CorePac 访问共享资源时对 TeraNet 总线的占用，KeyStone 单独设计了一个共享内存控制器，该控制器连接到各个 CorePac 上，为内存共享和内存保护机制的实现提供了可能。图 2-21 给出了 KeyStone 架构的典型特征框架图。

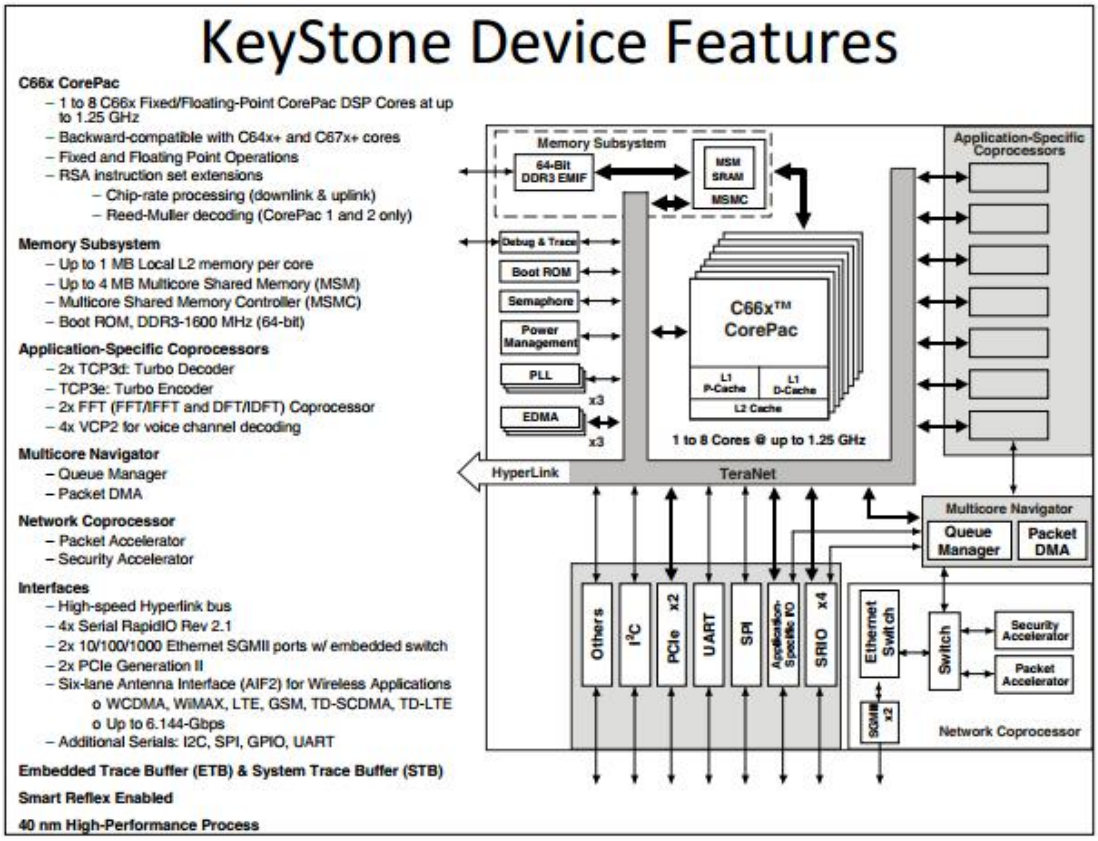

图2-1 TI KeyStone 设备特性[20]

嵌入式平台的多处理器技术的硬件实现大同小异，其基本思想不管是 ARM 还是 MIPS 还是 X86 都是一样的，本文主要讨论基于 KeyStone 架构的 TMS320C6678

的多核及相关技术，这是因为后续的开发工作是在 TMS320C6678 上进行的。

2.2 TMS320C6678 硬件平台多核技术研究

2.2.1 TMS320C6678 平台硬件简介

TMS320C6678 是 KeyStone 架构下的一款高性能多核 DSP 片上系统。其采用的是 TI 最新的 C66x DSP 内核，能以极低功耗运行在 1.25GHz 的主频下。该片上系统主要应用于精密工业控制，医疗图形图像处理以及自动化等对实时性有较高要求的行业[21]。

TMS320C6678 主要由以下部件组成：

1. 8 个 TMS320C66x DSP 组成的 8 个 CorePac 计算核心
2. 用于访问三级内存（后简称 L3 内存）以及外部主存的多核共享内存控制器
3. 用于加速数据在内部多核心间传递而设计的多核导航核心
4. 用于提供安全与包加速功能的网络协处理器
5. 其他外部 I/O 设备

2.2.1.1 内存模型简介

TMS320C6678 内存由多级构成。这些内存从功能上可以分类为缓存（Cache）和通用内存（Memory），从 CorePac 角度可以分为 CorePac 内部内存和 CorePac 外部内存，从离 DSP 的远近可以分为一级，二级，三级和四级内存（后分别简称 L1 内存，L2 内存，L3 内存和 L4 内存）。这些内存有些是 CorePac 独占的，有些是多核共享的。

在 CorePac 内部有两级内存 L1 和 L2，这两级内存都可以配置为静态随机存储器（Static Random-Access Memory，简称 SRAM）或 Cache。L1 在配置为缓存的时候分为指令缓存和数据缓存，分别为 32KB 大小。如果被配置为 SRAM 则总共有 64KB 大小。在默认情况下，L1 内存全部被配置为 Cache，其中指令缓存采用直接映射的方式而数据缓存采用二路组相连的方式，其中缓存行的大小为 32 字节。L1 除了被配置为缓存外还可以被配置为 SRAM 作为内部内存访问，L1 的访问速度是所有内存中速度最快的。其地址空间位于 0xe00000 和 0xf00000 处，长度为 32KB。多核之间是不能互相访问 L1 内存的，每个处理器只能访问自己的 L1 内存。L2 内存和 L1 内存类似可以配置成缓存也可以配置成 SRAM，默认情况下配置成 SRAM。L2 内存大小为每个处理器 512KB，对于拥有 8 个处理器的 TMS320C6678 来说拥有 4096KB 总共大小。L2 内存在多核间是可以交叉访问的，访问不属于本处理器 L2 内存中的数据的性能会明显慢于访问本处理器 L2 内存中的数据。图 2-2

描述了 TMS320C6678 的内存分布框架。

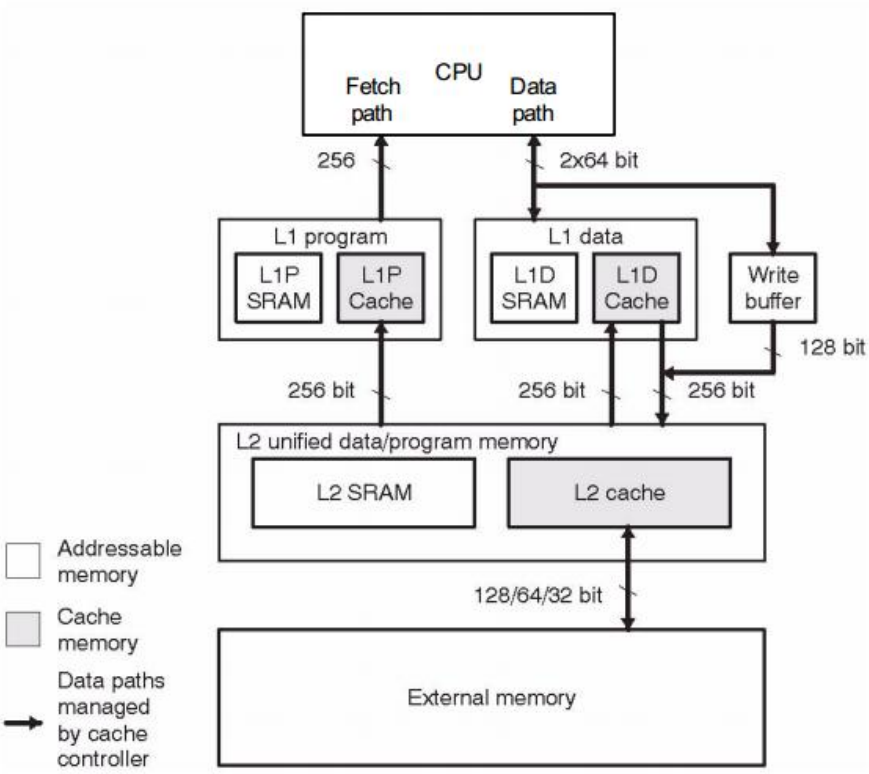

图2-2 CorePac 内存示意图[22]

在 CorePac 外部也有两级内存 L3 和 L4。L3 内存在实现上是 SRAM，其大小为 4096KB，用于多核共享内存管理，又称 MSMC SRAM。不同于 L3 内存，L4 内存位于片外，也叫外部内存。CorePac 通过多核共享内存控制器的 DDR EMIF 来访问外部 DDR。多核共享控制器使用 36 位地址，CorePac 与多核共享控制器的接口会将 36 位地址转换成 DSP 使用的 32 位地址。所有核心对 L3 内存和 L4 内存的访问是根据核心的优先级和总线的配置有关，但是总的来说访问 L3 内存的速度要远远大于 L4 内存。图 2-3 描绘了 TMS320C6678 内部内存互联访问的情况。

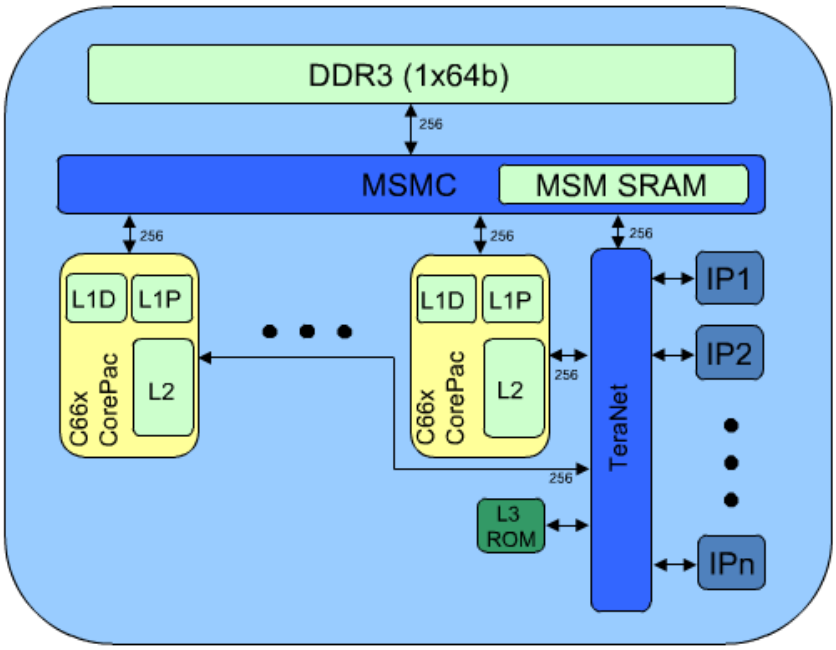

图2-3 TMS320C6678 内存互联[23]

除了上述内存外，TMS320C6678 还拥有一个 128KB 的位于 CorePac 外部的只读存储器（L3 Read Only Memory，简称 L3 ROM），该 ROM 存放的是用于储存将硬件初始化成厂家默认配置的第一阶段加载程序（First Phase Loader，简称 FPL）。

2.2.1.2 片内互联总线简介

TMS320C6678 通过 TeraNet 总线将 C66x CorePac，外部直接内存访问（External Direct Memory Access，简称 EDMA）控制器以及其他外部设备连接在一起。TeraNet 总线允许数据高效的在各主从设备间进行传输。TeraNet 同样支持通过标识优先级的方式在多主设备访问同一从设备时进行总线仲裁。

2.2.2 TMS320C6678 平台多核技术

2.2.2.1 多 CorePac 技术

CorePac 是 TMS320C6678 的计算单元，一个 TMS320C6678 包括了 8 个 CorePac，这是 TMS320C6678 进行多核并行计算的硬件基础。CorePac 并不是不可拆分的模块，其内部包括了许多重要组件，这些组件包括：C66x DSP，L1 内存控制器（包括 L1 指令内存控制器和 L1 数据内存控制器），L2 内存控制器，内部直接内存访问（Internal Direct Memory Access，简称 IDMA）控制器，外部内存控制器，扩展内存控制器，带宽管理单元，中断控制器和节能控制器。

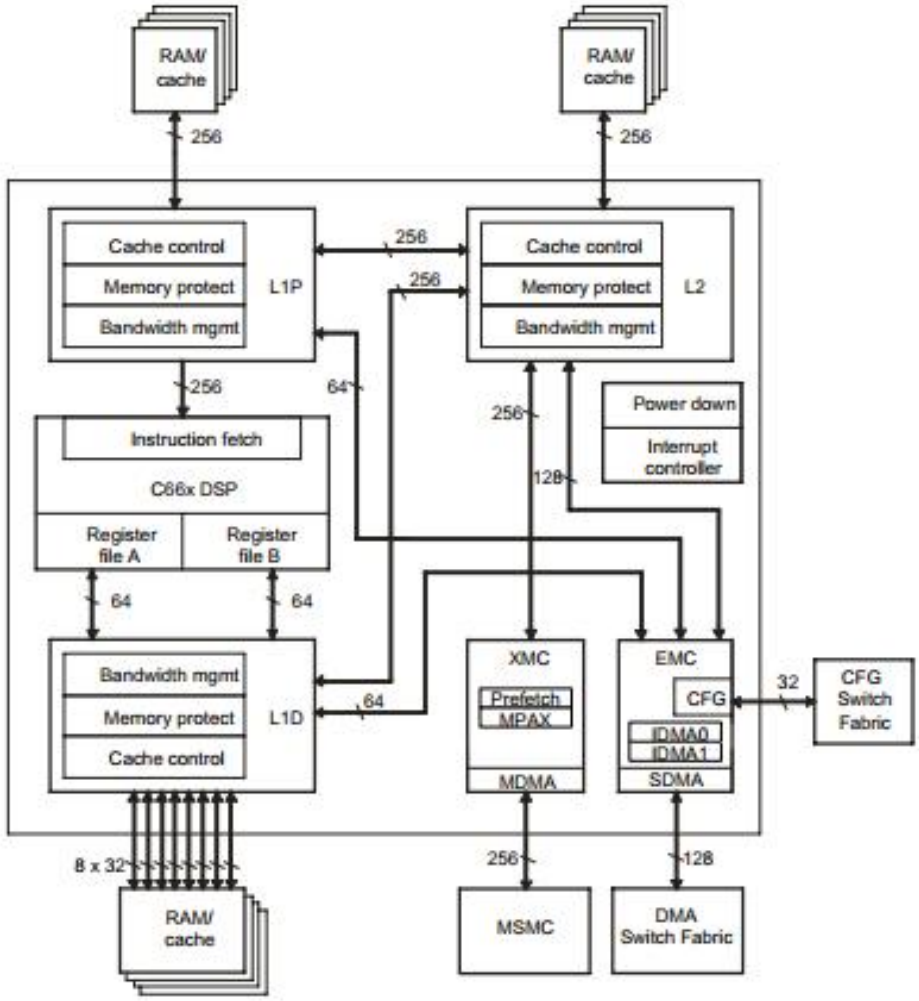

图2-4 CorePac 内部组成示框图[22]

图 2-4 给出了 CorePac 内部的框架组成示意图，接下来简要介绍这些硬件模块的功能：

1. C66x DSP，C66x DSP 是计算的核心单元。C66x DSP 在机器码上兼容 C64x+/C674x DSP。

2. L1 内存控制器，L1 内存控制器包括 L1 指令内存控制器和 L1 数据内存控制器。L1 内存控制器不仅简单支持将 L1 内存配置成 SRAM 或缓存，甚至允许将 L1 内存配置 SRAM 和缓存的混合。除此之外，L1 内存控制器还管理针对 L1 内存的带宽控制，内存保护以及节能模式等功能。

3. L2 内存控制器，L2 内存控制器在功能上和 L1 内存控制器类似。

4. IDMA 内部 DMA 控制器，IDMA 控制器支持将数据在 L1 内存，L2 内存和配置寄存器组间进行快速传输。IDMA 有两个通道，其中通道 1 支持数据在配置寄存器组和 L1，L2 内存间的传输，而通道 2 仅支持数据在 L1 和 L2 内存间的传输。

5. 外部内存控制器，外部内存控制器是 CorePac 通向外部的桥梁。它主要由两个部件组成：配置寄存器组和从直接内存访问控制器（Slave Direct Memory Access，简称 SDMA）。配置寄存器组提供 CorePac 访问外部映射寄存器的功能而 SDMA 提供数据在 CorePac 内存与 CorePac 外部的传输，但是在数据传输的过程中 CorePac 只能处于从模式。

6. 扩展内存控制器，扩展内存控制器是 L2 内存连接多核共享内存控制器的接口。扩展内存控制器的主要功能包括：提供对多核共享内存访问的通路，对外部内存（L3 和 L4 内存）进行内存保护，32 位 CorePac 内部地址到 36 位逻辑地址的扩展，指令预取。

7. 带宽管理单元，带宽管理单元用于管理 CorePac 内各内存访问请求。为了防止某些组件在发出访问内存请求时长期得不到处理，带宽管理单元实现了一种叫做带宽保留的技术来保证发起者的请求能在一定时间内得到处理。

8. 中断控制器，CorePac 内的中断控制器主要负责本地处理器的中断处理。C66x DSP 可以接收 12 个可屏蔽中断，1 个可屏蔽异常和 1 个不可屏蔽中断/异常。CorePac 的中断控制器通过将 128 个系统事件（其中 4 个保留）进行重映射的方式提供一种灵活的中断处理方式。

9. 内存保护单元，CorePac 内的内存保护单元仅用于保护 L1 和 L2 内存。L3 和 L4 内存的保护由扩展内存控制器和外部多核共享内存控制器一起完成。

10. 节能模式控制器，节能模式控制器允许在 CorePac 空闲时降低本 CorePac 的功耗。

2.2.2.2 多核内存共享技术

在 TMS320C6678 中，主要可以通过 L2，L3 和 L4 内存来实现共享内存，在访问这些不同内存的性能会有不同。对于 L2 内存，在配置成 SRAM 的基础上，本地处理器访问本地 L2 内存的速度要远快于访问异地 L2 内存的速度。L3 内存由于位于 CorePac 外部，访问速度要低于 L2 内存。为了防止多个 CorePac 同时访问共享内存而出现阻塞的情况，L3 内存在硬件设计上用了内存分页技术。这个技术允许多个 CorePac 在访问 L3 内存的不同区域的时候能够同时进行。L4 内存通常是一片大容量的双倍速率同步动态随机存储器（Double Data Rate Synchronous Dynamic Random Access Memory，简称 DDR），CorePac 对于该 DDR 的访问速度要慢于访问任何其他内存的速度。TMS320C6678 并没有实现任何保证缓存一致性（Cache Coherence）的协议。多处理器上的缓存一致性问题指的是，在一个系统中，多个处理器共享同一个存储器资源，当同一个资源在不同处理器的缓存上都

有备份时，某个处理器对某个备份的修改会导致该资源的数据在多个处理器间不一致[24]。当 L1 和 L2 内存均被配置为缓存时，由于 L1 和 L2 缓存是 CorePac 独享，所以在 L1 和 L2 内存间不存在缓存一致性问题。但是 TMS320C6678 的 L3 内存被 8 个 CorePac 共享，L3 内存和 L1，L2 内存间就存在出现缓存一致性问题的可能。TMS320C6678 没有实现任何针对 L3 内存的缓存一致性协议，如出现多个 CorePac 同时读写 L3 内存，则需要用户自己通过 CorePac 外部的同步互斥硬件机制或其他软件机制来防止出现缓存一致性问题。

2.2.2.3 多核核间通信技术

TMS320C6678 通过核间中断的方式实现核间通信。在设备状态控制寄存器中有相应的寄存器可以触发核间通信所需要的核间中断。在使用这些寄存器前需要配置中断控制器将核间中断映射到 DSP 的 12 个可屏蔽中断上。

在设备状态控制寄存器中针对核间中断技术设计了 4 组寄存器，这 4 组寄存器分别是核间中断发生寄存器组，核间中断应答寄存器组，主 DSP 核间中断发生寄存器组，主 DSP 核间中观应答寄存器组。其中后面两个是将核间中断信号通过设备的 HOUT 中断线引到外部其他设备上。8 个 CorePac 间的核间中断交互动作只使用前面两组寄存器。对于拥有 8 个 CorePac 的 TMS320C6678 来讲，核间中断发生寄存器组总共有 8 个 32 位寄存器，分别对应 8 个 CorePac。如下图 2-5 所示，向 IPCG 位写 1 即可发送核间中断到对应的 CorePac。

Bit	Field	Description
31-4	SRCSx	Interrupt source indication. Reads return current value of internal register bit. Writes: 0 = No effect 1 = Sets both SRCSx and the corresponding SRCCx.
3-1	Reserved	Reserved
0	IPCG	Inter-DSP interrupt generation. Reads return 0. Writes: 0 = No effect 1 = Creates an Inter-DSP interrupt.

图2-5 核间中断发生寄存器组[22]

SRCSx 用于让用户保存一些核间中断的信息，这些信息可以用来标识核间中断的软件类型。当 SRCSx 写有相应信息并通过向 IPCG 位写 1 发生核间中断后，被中断的 CorePac 可以通过核间中断应答寄存器组来查询这些信息。

2.2.2.4 多核中断管理技术

TMS320C6678 通过多级中断控制器来管理中断。从外到内总共分为三个层次，

这三个层次从外到内分为：片上中断控制器（Chip Interrupt Controller，简称 CIC），CorePac 中断控制器，DSP 中断线。TMS320C6678 总共有四个 CIC，每个 CIC 负责数量不同的外部事件输入。其中 CIC0，CIC1 和 CIC2 个控制 160 个外部事件输入，CIC3 控制 80 个外部事件输入。CIC 将这些外部事件进行重组和合并，最终将被选中的信号送入片内。其中 CIC0 和 CIC1 将外部信号重组后合并成 128 个系统事件发送给 8 个 CorePac，这 128 个信号中包括了 8 个核间中断信号。CIC2 将 64 个系统事件发送给 EDMA3 控制器，CIC3 则将 64 个和 16 个系统事件送给 HyperLink 单元和 EDMA3 CC0 控制器。送给 8 个 CorePac 的 128 个系统事件又通过 CorePac 内部的 CorePac 中断控制器进行重新映射，最终这 128 个系统中断信号会被映射到 DSP 的 CPUINT4 到 CPUINT15 这 12 条中断线上。后文中外部事件指的是从外部引脚引入到 CIC 的中断事件，系统事件指的是从 CIC 引入到 CorePac 中断控制器的中断事件而中断特指从 CorePac 中断控制器到 DSP 的某个中断事件。

2.2.2.5 硬件自旋锁技术

自旋锁（Spinlock）是传统多核平台实现多核间互斥同步的重要技术。自旋锁与互斥锁类似，只是自旋锁不会引起调用者睡眠[25]。如果自旋锁已经被其他的处理器占用，调用者就会一直轮询自旋锁判断持有者是否已经释放了锁。由于自旋锁的使用者一般保持锁的时间非常短，因此选择轮询而非睡眠是非常必要的，自旋锁在较短代码逻辑时效率远远高于互斥锁。

传统多核处理器通过指令的方式实现自旋锁及其他同步机制，但是这种方法只针对指令集相同的多处理器有效。如果一个片上系统中包含有不同体系不同指令集的多个处理器，那么这些处理器间的共享数据同步将无法通过指令来实现。TMS320C6678 为此设计了一个硬件自旋锁组件，该组件提供若干个片内级自旋锁。用户可以使用主动式和被动式这两种方式来利用该组件实现自旋锁语义。其中主动式意味着某个处理器将主动试探某个锁，如果有效则获取该锁，如果无效则继续轮询。被动式则是当某个处理器试探访问失败时，其会直接离开该代码片段继续执行其他程序，当该锁再次有效后锁组件通过中断的形式通知该处理器锁有效。

2.2.3 TMS320C6678 平台的内存保护技术

2.2.3.1 分布式内存保护单元技术

由于 TMS320C6678 多级内存的特殊构造，所以其内存保护单元被设计成了分布式的。这和传统多核处理器集中式的内存保护不同。在 CorePac 中的内存保护单元主要负责本 CorePac 中 DSP 的寻址请求，而 CorePac 外的多核共享内存控制器

则负责保护其他协处理器及 DMA 对 L3，L4 内存的寻址请求。除了针对内存的保护，TMS320C6678 还设计了一个单独的内存保护单元来保护外设寄存器以及其他协处理器。对以上所有内存保护单元统称为分布式内存保护单元。针对平台操作系统而言关键的 L2，L3，L4 内存保护单元皆位于 CorePac 的扩展内存控制器内。

2.2.3.2 内存映射技术

在多核共享内存控制器中，其外部内存可寻址大小为 64GB，也就是 36 位地址空间。CorePac 中的 DSP 可寻址的大小为 4GB，也就是 32 位。为了允许 DSP 可以访问全部 36 位地址空间，TMS320C6678 实现了一种段内存映射技术，该技术是由内存保护与地址扩张单元（Memory Protection and Address Extension，以下简称 MPAX）来实现的。MPAX 定义了一系列映射寄存器，DSP 可以通过修改这些映射寄存器，将外部 36 位地址空间中的一部分映射到当前 DSP 中。图 2-6 给出了默认情况下的内存映射情况。

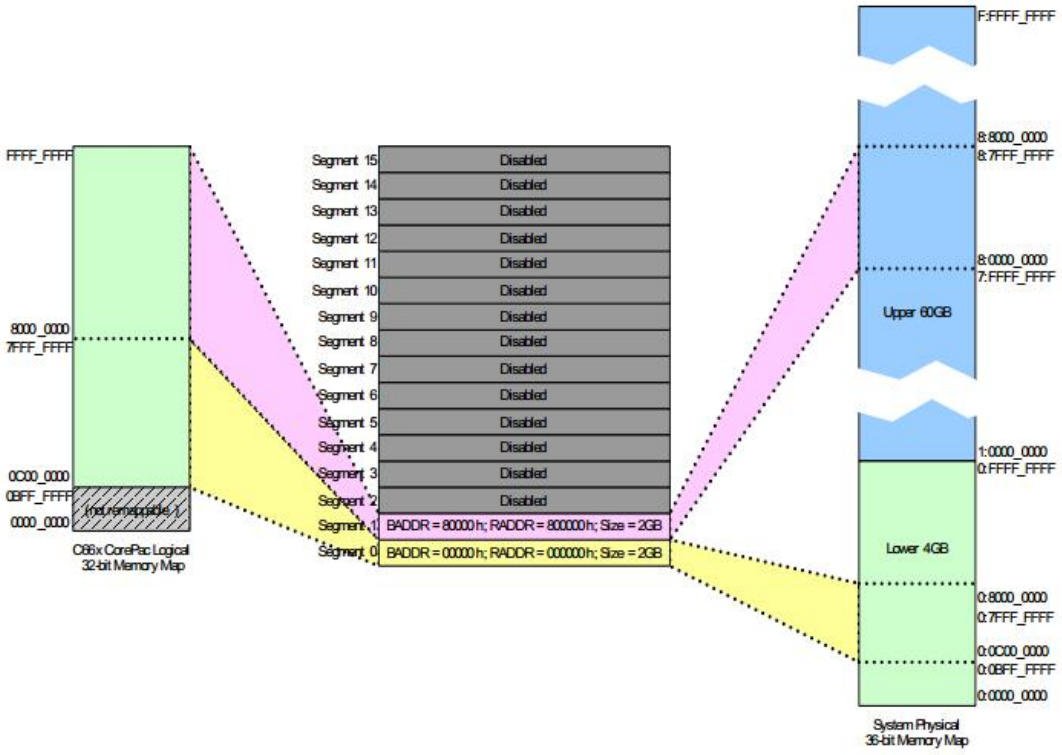

图2-6 默认情况下的内存映射[26]

2.2.3.3 多核访问共享内存时的总线竞争

当系统内多个 CorePac 访问同一个片上资源的时候将会发生针对 TeraNet 总线的总线竞争。TeraNet 总线允许软件来配置多核间的竞争规则。在 TeraNet 总线中，

每个可以发起资源访问请求的设备都拥有一个总线优先级，该优先级随其数字的减小而提高。当多个设备同时访问一个共享资源的时候，TeraNet 总线根据该优先级来判定访问的先后顺序。

2.3 基于 TMS320C6678 的对称多处理架构分析

对称多处理架构要求所有处理器共享一个操作系统副本，多处理器间将出现大量访问共享资源的情况。对于 TMS320C6678 来说使用对称多处理架构是必须对性能和实时性做出一定的妥协的。第一个问题是在 Cache 一致性方面。TMS320C6678 仅在硬件上保证了同一个 CorePac 内 L1 数据 Cache 和 L2 SRAM 间的 Cache 一致性。也就是说，当 L1 被设置 Cache，L2 被设置成 SRAM 时，任何对 L2 的改动（包括其他核对 L2 的改动）都会导致 L1 数据 Cache 数据内容的自动更新。但是硬件不保证以下情况的 Cache 一致性：L1 指令 Cache 和 L2 SRAM，两个不同 CorePac 间的 L1 和 L2 内存，CorePac 内存 L1，L2 内存和外部 L3，L4 内存。TMS320C6678 认为实现任何 Cache 一致性策略会提高系统的整体延迟，针对实时系统应用的多核处理器不应该实现过于复杂的 Cache 一致性策略。在采用对称多处理器架构的操作系统中需要将 L1，L2 内存配置为 SRAM 或将共享内存设置为不可缓存。第二个问题在于多处理间同步互斥方式的实现上。在多 CorePac 访问位于 L3，L4 内存的同一个数据时，为了保证数据的正确性需要锁的保护。TMS320C6678 在指令级别并没实现任何针对外部共享内存的同步指令，通过 TMS320C6678 提供的同步模块虽然能实现自旋锁，但是软件复杂度较高，编程难度较大。综上所述，TMS320C6678 采用单一对称多处理架构对于性能和编程成本上来说是不适合的。

2.4 基于 TMS320C6678 的非对称多处理架构分析

TMS320C6678 的 8 个 CorePac 具有一定的独立性，这种独立性十分适合非对称处理构架。不仅如此，非对称处理架构对共享资源的访问频率远低于对称处理架构，这也有效的解决了锁对性能产生影响的问题。图 2-7 描述了 TMS320C6678 下的非对称多处理器架构的框架图，图中同时运行了 8 个操作系统。

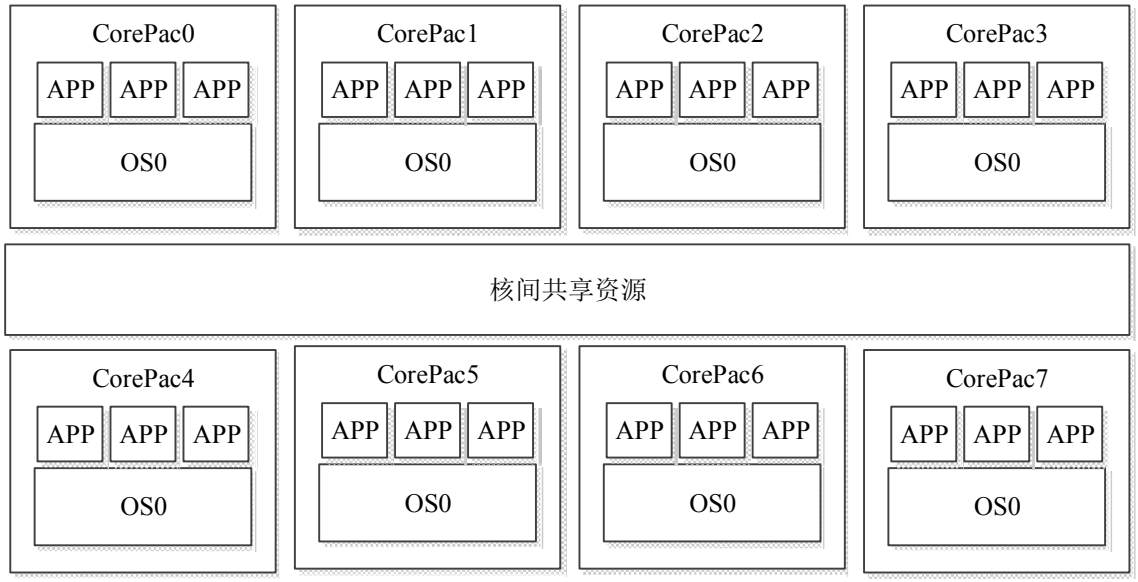

图2-7 TMS320C6678 下的非对称多处理器架构

2.5 基于 TMS320C6678 的混合多处理架构分析

混合多处理架构是对称多处理架构和非对称多处理架构的折中。多个核心虽然运行同一个操作系统，但是每个应用程序却是由用户设定好在哪个处理器上运行。某个任务不会自发的在处理器间发生迁移，只有用户可以控制任务的迁移行为。这种架构不会使用大量锁资源，同时又能最大限度简化系统设计。deCORE MOS 就是应用这种架构的一款运行在 TMS320C6678 上的混合多处理器架构嵌入式操作系统。图 2-8 描述了 deCORE MOS 在 TMS320C6678 上的架构图，deCORE MOS 已经拥有一个在 TMS320C6678 上的移植版本，该操作系统将在后文中介绍。

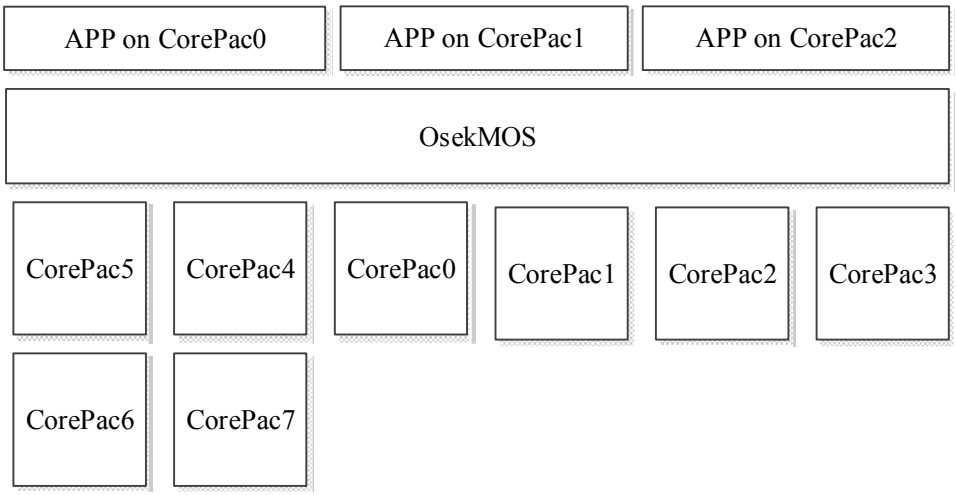

图2-8 deCORE MOS 多核操作系统架构图

2.6 本章小结

本章介绍了本文所选择硬件底层框架和原理，主要包括 TMS320C6678 平台的框架概述，内存总线模型，相关多核技术和内存保护技术。本章还集中讨论了对称多处理架构，非对称多处理架构以及混合多处理架构对于 TMS320C6678 硬件平台的支持情况，对比了 TMS320C6678 下各个架构的优势和劣势，为后续自主设计开发的多核多操作系统平台 Scorpius 提供了参考。

第三章 多核多操作系统板级支持平台 Scorpius 的整体设计

3.1 Scorpius 平台的设计目标与特点

3.1.1 设计目标

本文所研究的多核多操作系统平台 Scorpius 从软件上讲并非是一个操作系统而是一种介于硬件与操作系统之间的中间层。该平台的设计目的在于提供一种多核计算平台，该平台允许用户对底层的处理器进行编组和配置，然后同时运行多个操作系统来完成用户的目标任务。从架构上来讲，该平台在宏观上属于非对称处理架构，但是在微观上该平台融合了对称处理架构。该平台并不涉及用户操作系统的具体如进程调度，应用内存管理等内容。该平台的目标是最大化重复利用市场上现成的优秀嵌入式操作系统，这样做的好处是可以节省用户重复编写应用代码的时间。用户可以现在嵌入式操作系统上编写应用，然后再将操作系统移植到该平台上，从而大幅度减少了应用开发周期从而降低了生成设计成本。

3.1.2 设计特点

本文所研究的多核多操作系统平台 Scorpius 在架构上采用非对称架构模型，支持 8 核处理器并且最大能同时运行 8 个同种或不同种单核嵌入式实时操作系统。除此之外，该平台也支持混合运行多核或单核嵌入式操作系统，由于该平台并未对上层操作系统进行任何性能上的修改，上层操作系统的实时性可以得到尽可能的保留。该平台在设计上采用分层分模块设计的特点，最大限度提升平台可扩展性。该平台定位于工业控制，医疗汽车电子等领域，主要面向的硬件目标是工业控制级微处理器，DSP 以及多处理器微控制器。该平台应具有执行效率高，占用空间小，可扩展性强，可移植性良好等特点。在编码设计方面大部分代码采用 GNU C 语言编写，与平台硬件相关的部分使用汇编语言编写。整体设计采用面向对象的软件思想，将系统的组件抽象成对象，将操作用函数指针的方式保存在对象的数据结构中。

3.2 Scorpius 平台整体设计

3.2.1 整体设计

Scorpius 平台采用分层软件栈的设计思想。如图 3-1，整个完整软硬件体系从上到下分别为应用层，操作系统层，操作系统支持层，功能支持层，硬件抽象层，

基础硬件驱动层，基础硬件驱动层和硬件层组成。其中属于 Scorpius 平台范畴的层次包括操作系统支持层，功能支持层，硬件抽象层和基础硬件驱动层。

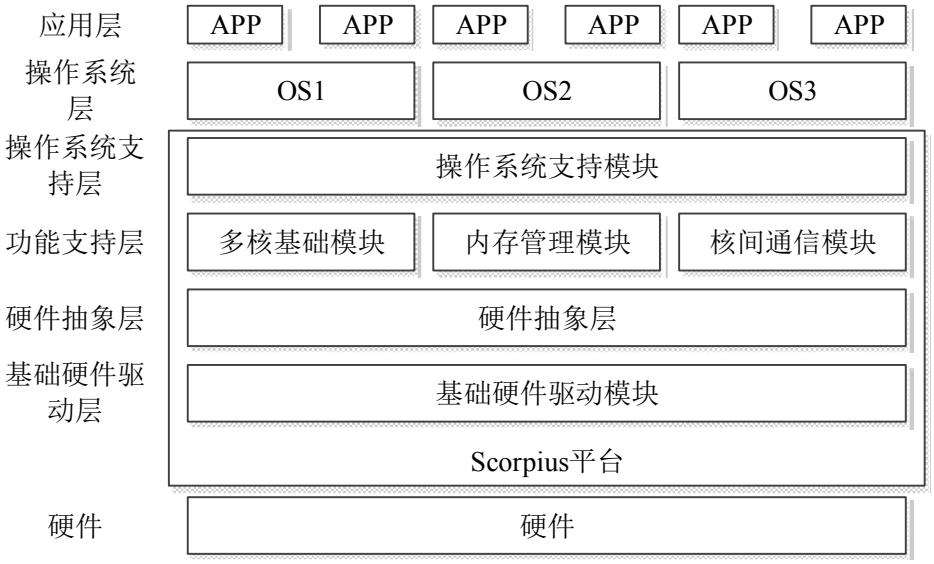

图3-1 Scorpius 生态环境整体设计框图

从下到上，软件环境的各个层次功能如下：

1. 硬件层，硬件层为 Scorpius 平台的硬件基础部分，本文主要以 TI 的 TMS320C6678 为平台，硬件层要求硬件必须是多核处理器。

2. 基础硬件驱动层，该层包含了针对 TMS320C6678 的各种上层必要硬件的驱动程序。为了简单，驱动程序之间互相独立，整个层次并不像 Linux 那样实现复杂的驱动模型框架。

3. 硬件抽象层，该层对基础硬件驱动层提供的具体硬件服务进行抽象。简单来说，就是对 TMS320C6678 的驱动程序进行封装，向上提供接口，屏蔽底层的实现细节。之所需要硬件抽象层，是由于硬件设备特别是嵌入式硬件设备和传统桌面不同。传统桌面统一采用 X86 的架构，而嵌入式设备多是采用片上系统的模式，片上系统内的各个模块可能由不同厂商设计，驱动没有一个固定的模型。为了避免平台直接运行在驱动特定的硬件设备上，必须通过一个隔离层来屏蔽。

4. 功能支持层，功能支持层是纯软件的逻辑层。该层应用硬件抽象层提供的硬件接口实现一些支持上层操作系统所必要的软件逻辑功能。诸如多核支持，核间通信，内存管理等软件功能都是在该层实现。

5. 操作系统支持层，该层将 Scorpius 平台的所有功能进行精炼将这些功能提供给上层操作系统。

6. 操作系统层，该层运行的是多种嵌入式操作系统。虽然 Scorpius 抽象了操作系统所需要的功能，但是诸如上下文切换，系统调用等功能还是需要操作系统针对具体硬件进行具体处理。除此之外由于大多数嵌入式片上系统不支持硬件虚拟化，故 Scorpius 目前尚未实现硬件虚拟化功能，所以每个操作系统至少运行在一个处理器之上，Scorpius 目前还未实现单个处理器并发运行多个操作系统。

7. 应用层，应用层执行在操作系统层之上，该层是用户直接编写的应用程序。

3.2.2 各模块设计

Scorpius 主要由基础硬件驱动层，硬件抽象层，功能支持层和操作系统支持层组成。细分这些层次可以得到基础硬件驱动模块（包含了硬件抽象层），多核基础支持模块，多核同步模块，内存管理模块，核间通信模块和操作系统支持模块。

3.2.2.1 基础硬件驱动模块

基础硬件驱动模块包括了 TMS320C6678 的各个关键硬件组件的驱动程序与相关功能的硬件抽象层。在功能上大致分成以下几个模块：

Bootloader 又称引导加载程序，是系统加电后运行的第一段代码[27]。随着微电子技术的发展片上系统越来越复杂，这种复杂性导致目前应用于嵌入式的 Bootloader 一般分成了两个阶段。第一阶段由 Bootloader 软件和硬件厂商的固件（Firmware）共同组成，该阶段主要任务是初始化硬件并读取第二阶段 Bootloader 代码。第二阶段则由 Bootloader 软件完成，该阶段主要任务则是记录硬件信息，根据用户配置再次初始化硬件到最佳状态，然后加载操作系统内核代码。这里 Bootloader 模块主要指就是第二阶段的加载程序（Second Phase Loader，简称 SPL），TMS320C6678 在复位时将位于 L3 ROM 中的 FPL 加载到内存并执行。FPL 会将 TMS320C6678 的硬件根据用户配置初始化成主机端 GEL 文件所描述的状态。GEL 文件是 TI 定义的一种交互式脚本，该脚本允许用户对设备的初始化情况进行配置。这里的主要任务则是在 FPL 的基础上打印并保存硬件信息，配置多核资源并决定多核间的启动顺序，然后根据用户操作系统的选择重新配置硬件。

锁相环（Phase-Locked Loops，简称 PLL）时钟树模块，PLL 时钟树模块负责记录和配置 TMS320C6678 的各个部件的主频。锁相环是一种利用反馈控制原理实现的频率及相位的同步技术，其作用是将电路输出时钟与外部参考时钟保持同步。参考时钟发生改变时，锁相环会检测到变化并通过反馈电路来调节输出频率[28]。由于嵌入式处理器的日益复杂，内部时钟要求也越来越高。通常多处理器片上系统将时钟需求相近的硬件组件集合在一起构成时钟域，多个时钟域再构成一棵时

钟树。在 Linux 中，设备时钟的管理就是采用时钟树的方式进行。

定时器模块，定时器模块用于产生操作系统所依赖的时间滴答。定时器驱动允许用户通过配置定时器模块的寄存器来调整时钟中断发生的频率。

中断控制器与异常管理模块，TMS320C6678 的多级中断控制器用于控制外部事件到内部 DSP 中断的映射。由于 DSP 中断线有限，必须通过中断映射的方式来将大量外部中断引入 DSP 内。除此之外，中断控制器驱动还提供中断向量表以及某个中断的使能等其他重要的功能。

用户交互模块，用户需要通过一些方式来获取当前系统的状态。嵌入式系统通常使用串口，用户图形化界面等方式提供一些输出给用户。本平台仅提供串口的输出方式，这里用户交互模块也就是针对串口的设备驱动。

其他辅助模块，由于 TMS320C6678 是一个复杂的片上系统。为了完成一个简单功能需要大量硬件模块的支持。这里辅助模块主要是针对功能支持层提供服务。这些模块主要涵盖 DSP 内部寄存器的辅助函数，缓存与写缓冲区（Writer Buffer）控制器驱动等。

3.2.2.2 多核基础支持模块

多核基础支持模块主要针对 CorePac 和片上中断控制器相关的多核功能。该模块主要用于管理 CorePac 并将 CorePac 抽象成资源供上层操作系统使用，该模块提供对多处理器身份的识别，核间共享内存的使用等相关功能。片上中断控制器和 CorePac 内部中断控制器一起实现核间中断。本平台实现一种基于 OpenBinder 模型的扩展式核间通信方式，该方式的底层实现依赖于 CorePac 内部中断控制器和片上中断控制器 CIC。

3.2.2.3 多核同步模块

多核同步模块主要负责实现原子变量，自旋锁及内存屏障，缓存以及预取缓冲区等会发生数据一致性问题组件的配置。硬件同步资源是一种数目有限的硬件模块。平台必须严格记录这些资源的使用，这样做的目的一是防止上层系统大量浪费数量有限的锁资源，一是防止错误使用导致的死锁等问题。在 TMS320C6678 中有两种预取缓冲区，这两个预取缓冲区皆为于 CorePac 内的扩展内存控制器中。一个是数据预取缓冲区，另一个是指令预取缓冲区。预取的作用是旨在减少 CorePac 读入数据流中需要读取数据的未命中情况。由于 CorePac 针对 L3，L4 内存的读取速度要远慢于内部 L1，L2 内存的速度。而程序或数据总是呈现出一定的空间局部性。CorePac 的预取正是利用这种局部性，一次性多读取一些内容到缓冲

区中，这样可以提高系统整体的读取性能。但是这种功能在多核情况下可能会与Cache 互相影响产生一致性问题。该模块提供一些函数通过配置优化预取缓冲区和Cache 策略的方式来解决一致性问题。

3.2.2.4 内存管理模块

内存管理模块是功能支持层的重要模块，该模块主要包括内存管理和内存保护两个方面。由于上层操作系统自身就拥有一些内存管理的算法，为了支持多个操作系统的同时执行，Scorpius 平台必须提供一个统一的块内存分配接口函数给上层操作系统。Scorpius 平台的内存管理分为两个方面，一个是平台动态内存的分配与释放，另一个是操作系统内存的分配与释放。前者内存管理使用伙伴算法配合SLAB 小内存分配器算法的方法实现。这部分内存用于针对 Scorpius 平台使用的内存的分配与释放，后者则是使用用户配置的方式，将大规模共享内存分块分配给操作系统使用，这些内存在操作系统看来是互相隔离的，这是通过内存保护模块实现的。内存保护模块通过段内存映射的方式来保护那些分配给操作系统的块内存。除此之外，内存保护模块利用基础硬件驱动模块中其他辅助模块的内存保护单元驱动提供的功能来保护硬件，防止用户操作系统在运行时修改内存配置从而打破内存管理模块建立起的隔离。

3.2.2.5 核间通信模块

核间通信技术是一种通过核间中断的方式异步向其他处理器上运行的程序发送数据的方式。在硬件上，核间通信模块依赖于多核支持模块中核间中断驱动的相关功能。Scorpius 平台的核间通信在 OpernBinder 方式的基础改进而来。OpenBinder 是一种广泛应用于多种系统的跨进程通信机制。目前广泛应用于消费电子的 Android 操作系统就是使用这种跨进程通信机制。OpenBinder 允许进程将一些接口暴露出来，这些接口将被其他一些进程远程调用。Scorpius 平台在OpenBinder 的基础上将进程间的函数调用扩展到处理器间的函数调用。如图 3-2 给出了这种 OpenBinder 式的跨核函数调用的场景。

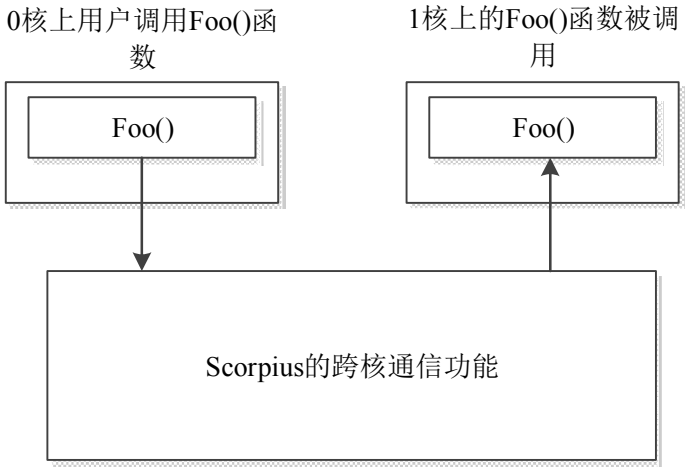

图3-2 基于 OpenBinder 方式的跨核函数调用

Scorpius 平台将一些系统必要的跨核跨 CorePac 组群函数分成本地函数和远程函数两种版本，这些函数拥有相同的接口。当用户调用这些函数时，系统将根据该函数所执行的上下文自动调用这些函数的本地或远程版本，让在远程处理器上执行的函数仿佛是在本地执行一样透明。这样做的好处在于，用户只需要设计自己函数中业务的实现而不需要关心平台底层硬件实现的细节。

Scorpius 平台提供一套名为 OIDL 的接口描述语言脚本，用户在编写跨核函数时只需要根据接口需求编写 OIDL 脚本，然后 Scorpius 平台会自动为用户生成中间代码，用户不需要关心这些代码的功能和实现，只需要将代码复制进自己的工程当中。这样用户可以在不了解底层核间通信实现的基础上编写跨核函数。

核间通信模块还支持核间邮箱方式的核间通信，这种方式较上述跨核函数要简单，适用于多处理器间轻量级数据交互的场景。该功能底层同样依赖于多核支持模块的核间中断驱动。

3.2.2.6 操作系统支持模块

操作系统支持模块主要实现加载运行，时钟滴答，中断处理以及内存管理四个内容。加载运行指的是系统在 Bootloader 之后根据用户配置依次启动上层操作系统。操作系统的时钟滴答通常由底层硬件定时器的时钟中断实现，操作系统支持模块的时钟滴答模块用于为当前操作系统的定时器进行配置，为当前操作系统提供正确的时钟中断发生频率。中断处理则是将底层中断号重新映射到上层各个操作系统对应的中断号上。由于同时运行多个操作系统，多个操作系统之间中断号存在重叠和交叉，必须使用一种中断号空间映射机制来保证底层中断能正确到达上层操作系统并得到处理。内存管理则是在内存管理模块的基础上对从属各个操作系统的内存块进行管理。

3.3 本章小结

本章从多核多操作系统平台 Scorpius 的设计目标和生产需求出发，简要的阐述了 Scorpius 平台的整体设计层次和模块划分的规划，在 3.2 节又详细的针对每个功能层次的具体模块给出具体设计思路。第四章将针对这些模块给出设计实现的具体。

第四章 多核多操作系统板级支持平台 Scorpius 模块设计与实现

4.1 基础硬件驱动模块

4.1.1 启动加载程序

Bootloader 即启动加载程序，是系统开机运行的第一个程序，该程序的目标是检查并初始化必须硬件，根据用户需求配置调整系统状态，记录当前系统资源状态，加载用户操作系统[28]。Bootloader 是任何系统必不可少的一部分。目前 Bootloader 普遍分成了三个阶段，这三个阶段分别是第一阶段加载程序（First Phase Loader，简称 FPL），第二阶段加载程序（Second Phase Loader，SPL）以及操作系统加载程序（Operating System Loader）。

第一阶段加载程序主要由硬件厂商编写，该部分程序通常固化在硬件的非易失存储器当中。在系统启动后，CPU 会将该段程序加载或映射到内存上并执行，这段程序先是进行硬件功能性的自检，检查电源配置，然后根据厂商的默认要求将设备配置到一个简单状态，最后再根据不同的外接引脚配置来选择加载第二阶段加载程序的介质。这些介质通常包括 FLASH 型存储器，以太网络，USB 总线，I2C 总线，SPI 总线以及其他总线等。

第二阶段加载程序则是在第一阶段加载程序的基础上对硬件进行进一步初始化。首先该程序会对时钟进行配置，一般设备在启动后会处于一个低频状态，为了发挥设备的全部功能，第二阶段加载程序会将将设备的时钟配置到一个适合启动的理想状态。然后程序会重新配置内存控制器，这是由于当前系统时钟发生了变化，第一阶段中对内存控制器的配置可能已经失效。然后第二阶段加载程序会初始化其他必要的外部接口设备。最后第二阶段加载程序会初始化一些软件相关的要素，比如设置堆栈指针，配置临时中断向量表，配置临时内存映射表等。在完成所有准备工作后，第二阶段加载程序会将控制权转移给操作系统加载程序。

操作系统加载程序一般运行在 C 语言环境中，该程序会记录一些系统参数，比如当前系统内存的起始地址，结束地址，如果系统包含多块内存地址，那么还会对这些内存的功能进行记录。除此之外，程序还会记录目前输出端口的情况，将这些信息保存在一个固定结构体中。然后将这些结构体根据所需加载的操作系统的不同安排在协议好的地址空间中。最后程序会根据用户配置在存储器中加载操作系统代码并将控制权交于用户操作系统。

TMS320C6678 的第一阶段加载程序位于 DSP 地址 0x20b00000 的位置。为了

适应不同的系统需求，该加载程序支持多种方式来将第二阶段加载程序加载进内存并运行。这些启动方式包括：

EMIF16 加载模式，该方式允许通过 EMIF16 接口来从 FLASH 型存储设备中加载第二阶段加载程序，这种模式较常用。

串行高速 IO（Serial Rapid IO，简称 SRIO）加载模式，该方式通过 SRIO 从远程设备中加载第二阶段加载程序。

以太网加载模式，该方式通过网络从目标主机处加载第二阶段加载程序。

PCIe 加载模式，从外部主机出将第二阶段加载程序通过 PCIE 总线加载到片上主存中，然后再通过主存运行第二阶段加载程序。

I2C 主模式加载模式，将 TMS320C6678 配置成 I2C 主模式，从位于从模式的 I2C EEPROM 中加载第二阶段加载程序，这种模式较常用。

I2C 从模式加载模式，将 TMS320C6678 配置成 I2C 从模式，通过外部主 I2C 设备将第二阶段加载程序加载到设备当中。

SPI 加载模式，通过 SPI 总线将位于串行 FLASH 存储器中的第二阶段加载程序加载到设备当中。

HyperLink 加载模式，其他外部主机通过 HyperLink 协议将配置和第二阶段加载程序加载到设备当中。Scorpius 平台的第一阶段加载程序采用的就是这种启动模式。

Scorpius 平台的开发工具是 TI 的 Code Composer Studio v5.0.3（以下简称 CCS）。CCS 是基于 Eclipse 的一个集成开发环境，工具链采用的是 TI 针对其 DSP 平台特别优化过的 GCC 编译器。Scorpius 平台的第二阶段 Bootloader 是基于 CCS 环境 GEL 配置文件实现的。GEL 配置文件通过脚本的形式让用户对 TMS320C6678 设备的一些主要参数进行配置。设备启动后，设备通过 HyperLink 访问主机 CCS，将位于主机 CCS 上的 GEL 配置文件下载到设备特定位置并运行。通过 GEL 配置文件，用户可以免去复杂而繁琐的寄存器配置，直接跳过第二阶段加载程序，将主要精力集中在操作系统的加载和运行上，图 4-1 描述了 Scorpius 平台的加载流程，操作系统的加载细节将在 4.6.1 小节中描述。

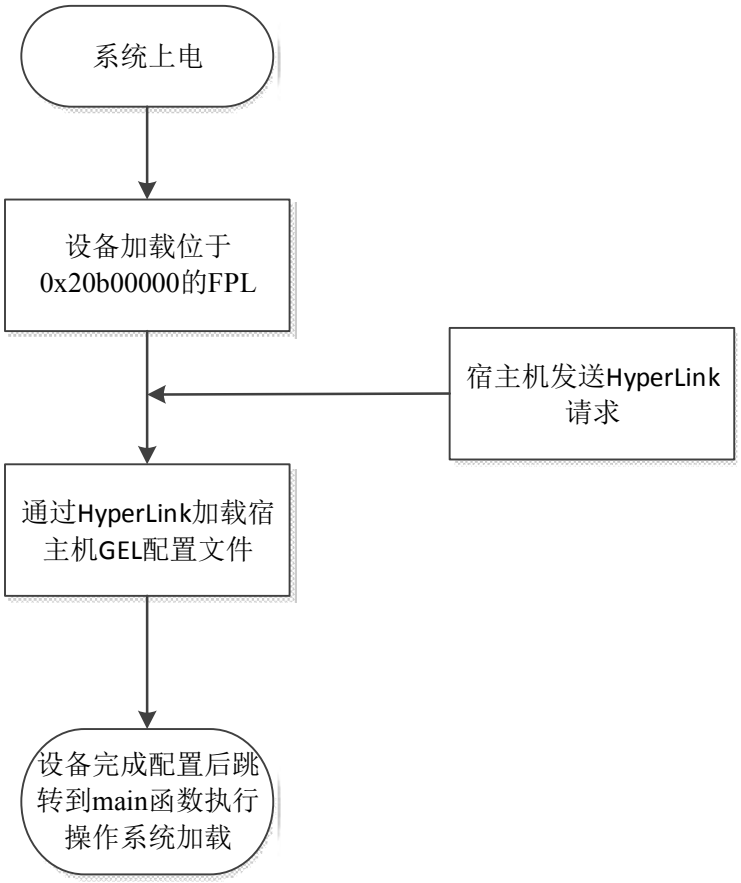

图4-1 Scorpius 平台的加载流程

4.1.2 锁相环与时钟树模块

早期的嵌入式设备时钟主要来自外部晶振引脚的直接分频或倍频，这些设备的时钟配置比较简单，功能单一。但是随着嵌入式片上系统的复杂化发展，单一芯片上包含了各种各样的外设控制器，这些控制器对时钟的需求各不相同。不仅如此，有些时钟信号可能会互相干扰，片上系统内部通过硬件方式将不同时钟信号分成时钟域。这些时钟域有时互相依赖，有时互相隔离，在操作系统角度来看，这些时钟关系构成了一个复杂的时钟树，该树的每个节点都代表了一个可以配置时钟分频倍频比例的锁相环和开关。

Scorpius 平台时钟树模块的主要作用是配置并记录 TMS320C6678 的时钟信息。对时钟树的配置，实质上是对设备锁相环控制器以及开关的配置。如图 4-2 给出了 TMS320C6678 的主要锁相环模块图。

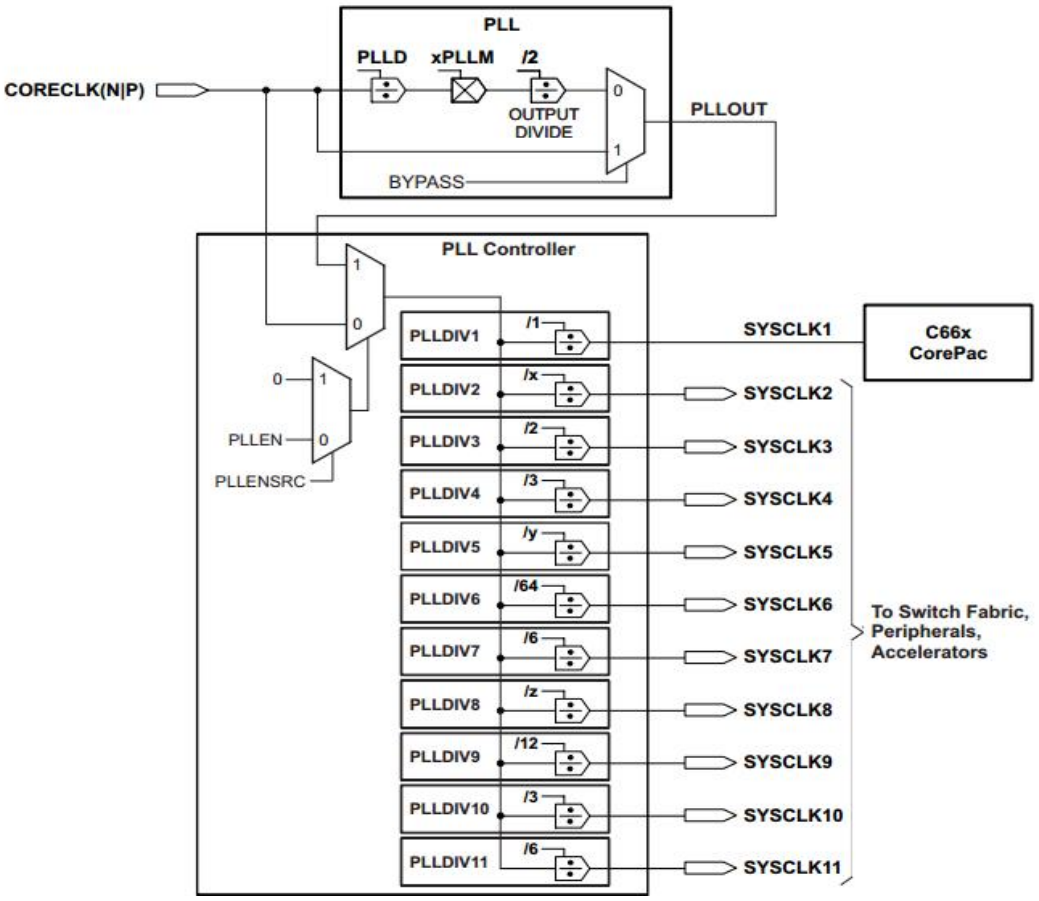

图4-2 TMS320C6678 时钟模块图[29]

TMS320C6678 分成了 11 个时钟域，分别由 11 个锁相环分频器控制，这 11 个时钟域的名字为 SYSCLKx（其中 x 从 1 到 11）。时钟输入来自于一个开关，这个开关为 0 时，时钟源取自 CLKIN 引脚。如果开关为 1 时，时钟来自于 PLL 的 PLLOUT。11 个锁相环分频器可以针对输入时钟进行分频，锁相环分频器支持的分频值从 1 到 256，具体分频值由相关寄存器控制器进行配置。锁相环中的 PLLM 寄存器控制锁相环对输入时钟进行倍频操作。当设备使能了 BYPASS 模式，那么 CLKIN 引脚的输入参考时钟将直接引入 PLLOUT，默认情况下不使用该模式。

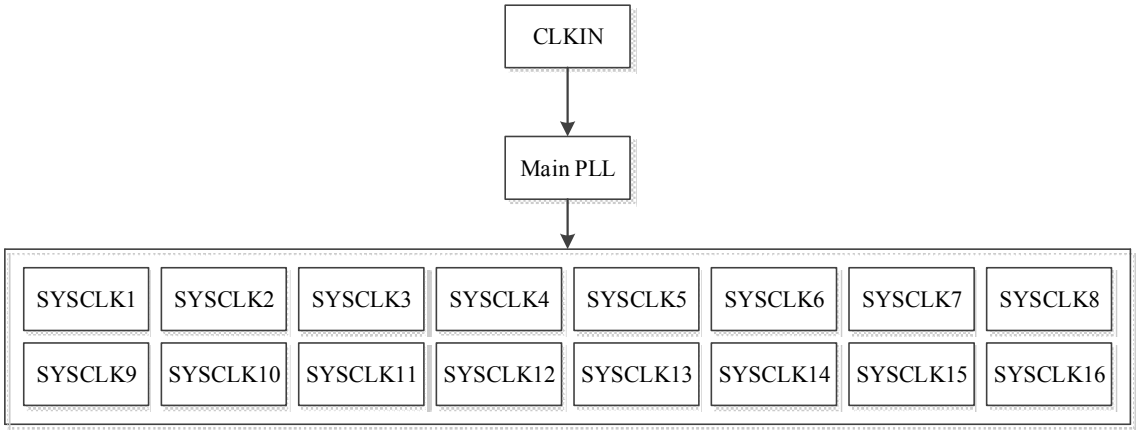

图4-3 TMS320C6678 时钟模块图

根据以上分析，Scorpius 平台设计的时钟树如图 4-3 所示。其中每个 SYSCLKx 都是一个时钟域的根，下面的每个时钟域都用来为一组外设提供时钟：

时钟域 1 负责 8 个 CorePac 的时钟。

时钟域 2 负责在模拟器模式下的 8 个 CorePac 的时钟。

时钟域 3 负责多核共享内存控制器，HyperLink，DDR 和 EDMA 的时钟。

时钟域 4 负责 TeraNet 高速总线，调试模块的时钟。

时钟域 5 负责系统时钟调试模块。

时钟域 6 负责 DDR3 的缓存区。

时钟域 7 负责外部慢速设备及 SYSCLKOUT 引脚的时钟。

时钟域 8 负责 slow_sysclk 引脚的时钟。

时钟域 9 负责 SmartReflex 功能的时钟，这是一种弹性供电功能用于在设备空闲时能反馈给电源管理芯片来到达省电的目的。

时钟域 10 负责串行高速 IO 口的时钟。

时钟域 11 负责 PSC 的时钟。

并不是系统中所有的时钟域都能进行配置，只有 2、5、8 三个时钟域允许在设备启动后进行重配置。在时钟模块驱动内部，为了表示各个时钟对象，Scorpius 平台使用了多个数据结构来表示这些对象。Scorpius 平台使用这些数据结构来记录当前平台的时钟配置，这些配置数据将为其他模块提供时钟参考。一个典型的应用就是，定时器模块将根据这些数据来计算适当的定时器计数值。

pll_div 结构用于表示一个分频器，该数据结构如下所示：

```
struct pll_div {
    int id; //  分频器的编号
    int is_enable; //  该分频器是否有效
```

```
    unsigned long div_ratio; //  该分配器当前分频值
    int (*set_ratio) (struct pll_div *pll_div, unsigned long div_r); //配置分频值的函数指针，返回值判断操作是否成功
    int (*enable_div) (struct pll_div *pll_div);//使能分频器操作的函数指针
    int (*disable_div) (struct pll_div *pll_dir);//失能分频器操作的函数指针
};
```

sysclk 结构用于记录一个时钟域的时钟，该数据结构如下所示：

```
struct sysclk {
    int sysclk_id;   //时钟域 ID
    int sysclk_stat;   //当前时钟域是否
    unsigned long sys_clk;   //当前时钟域时钟
};
```

pll_stat 结构用于记录与配置主锁相环控制器，该结构是时钟树的核心。它决定了整个系统到底运行在什么样的频率上。该数据结构如下表示：

```
struct pll_stat {
    unsigned long input_core_clk;   // CorePac 时钟
    unsigned long input_ddr_clk;   // DDR 时钟
    unsigned long dev_speed;   //设备主频
    int pll_is_enable;   //  是否位于 PLL 模式。
    unsigned long multiplier_rate;   //  主倍频器的倍频值
    unsigned long clk_out;   //  根据 CorePac 时钟，倍频值和分频值计算出的输出时钟
    unsigned long output_div;   //  主分频器的分频值
    struct pll_div pll_divs[11];   //  对应 11 个时钟域的分频器
    struct sysclk sysclk_stats[11];   //  对应 11 个时钟域的状态
};
```

在 Bootloader 的第三阶段操作系统加载程序中，初始化硬件的函数会调用 scan_pll 函数来扫描并构建整个时钟树然后根据用户配置将时钟域 2，时钟域 5 和时钟域 8 配置到用户需求的时钟上来，最后将时钟树信息记录下来根据需要打印出调试信息。

4.1.3 定时器模块

定时器用于在用户的配置下产生周期性的时钟中断，定时器是实现现代分时

操作系统不可或缺的硬件模块之一。对于多操作系统 Scorpius 平台，由于支持同时运行多个操作系统，所有操作系统的时钟滴答配置不一，所以 Scorpius 平台底层必须支持多个定时器模块。

TMS320C6678 有 16 个 64 位定时器，每个 CorePac 拥有其中的 2 个，其中 1 个可以配置为看门狗定时器或通用定时器，另一个只能配置为通用定时器，但是该定时器可以拆分成 2 个 32 位定时器使用。

这些定时器可以配置成看门狗模式，64 位模式，也可以配置成 32 位模式。当配置成看门狗模式时，计数器在外部时钟的驱动下递减，递减到 0 时定时器就会发生一次中断。当配置成 64 位模式时，计数器在时钟的驱动下递增，当计数器的值到达时钟周期寄存器 PRDLO 和 PRDHI 的值时，定时器就会产生一个可屏蔽的 TINTLO 中断。当配置成 32 位模式时，两个 32 位定时器可以配置成连锁模式和非连锁模式。对于 Scorpius 平台来说，定时器采用的是 32 位非连锁模式。该模式下两个 32 位定时器可以一个配置成普通的 32 位定时器另一个配置成一个 4 分频的 32 位低速定时器。

在 Scorpius 平台中，使用 c6x_timer_device 来表示一个定时器，该数据结构主要如下所示：

```
struct c6x_timer_device {
    int id;   // 指明是哪一个定时器
    struct device dev;   // 用于驱动框架记录所有的设备
    int clk_div;   // 指明该定时器的分频值
    int mode;   // 表示该定时器的工作模式
    int init_ready;   // 表示该定时器是否初始化就绪
    unsigned long reload_value;   // 如果定时器使用的是一次填充模式，那么需要在时钟中断中将该值重新加载到时钟周期寄存器中
    void (*init_timer) (struct c6x_timer_device *);   // 该定时器对应的初始化操作的函数指针
    int (*ack_timer_int) (struct c6x_timer_device *);   // 应答中断操作的函数指针
    void (*time_tick) (struct c6x_timer_device *);    // 时钟中断处理函数
};
```

TMS320C6678 的 16 个 64 位定时器在 init_c6x_timer 函数中进行初始化，该函数会根据参数 num 来判断初始化那个对应的 c6x_timer_device，并将初始化好的设备向注册到上层的硬件抽象层当中。图 4-4 描述了整个初始化流程。

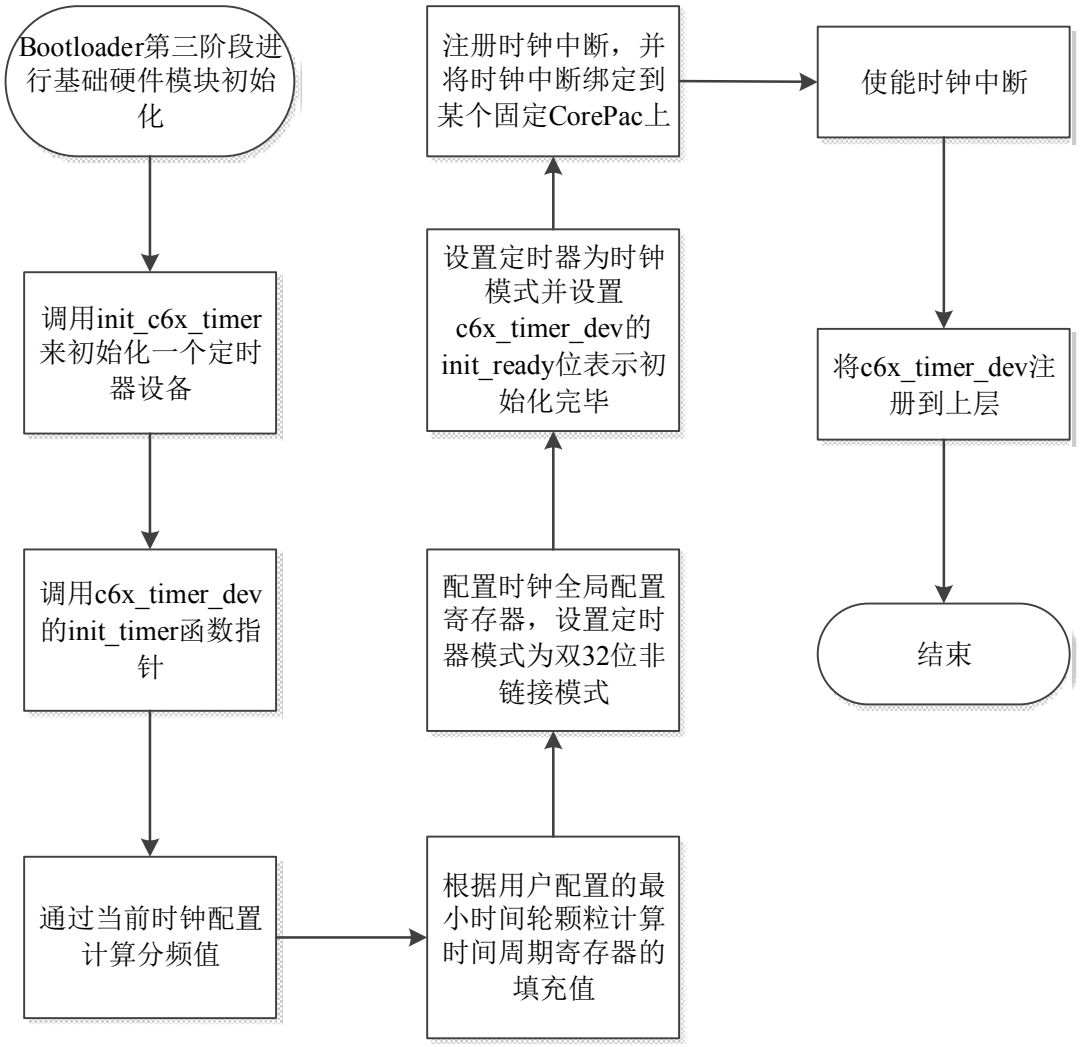

图4-4 Scorpius 平台定时器初始化流程

4.1.4 中断控制器与异常管理模块

中断是指处理器接收到来自硬件或软件的信号，提示发生了某个事件，应该被注意，这种情况就称为中断[30]。位于基础硬件驱动模块的中断控制器与异常管理模块主要负责 CorePac 的中断设备驱动以及 CorePac 中 DSP 内核的中断向量的管理。

DSP 内部响应 12 个可屏蔽中断，1 个不可屏蔽中断，1 个复位异常。其中 12 个可屏蔽中断分别为 INT4，INT5，INT6，INT7，INT8，INT9，INT10，INT11，INT12，INT13，INT14，INT15，这些可屏蔽中断优先级依次递减，总体优先级皆低于不可屏蔽中断 NMI。中断控制器与异常管理模块主要就是围绕着这 12 个可屏蔽中断来工作的。

DSP 内核使用一个中断向量表寄存器来保存中断向量表，中断向量表由 16 个

连续的指令包组成。每个指令包可以包含最多 14 条指令。中断向量表之间相距 32 字节，图 4-5 给出了 TMS320C6678 的中断向量表的示意图。

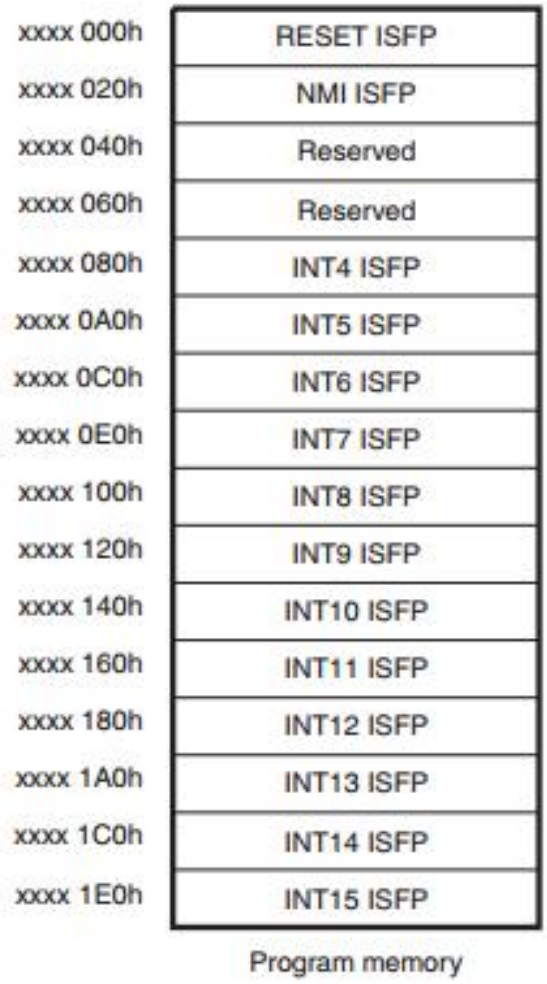

图4-5 TMS320C6678 的中断向量表

每个中断向量表项中的指令集合称为中断服务预取指令包（Interrupt Service Fetch Packet，简称 ISFP），ISFP 的大小是 32 字节。当中断处理完毕，应该使用 BIRP 指令来做中断返回，将程序流程跳转回中断前。由于 ISFP 的大小受限，如果中断处理程序过于庞大，则必须通过跳转的方式在 ISFP 中跳转到对应的子中断处理程序。中断向量表的基地址存储在 DSP 内核的中断向量表基地址寄存器（Interrupt Service Table Point Register，简称 ISTPR）中，该寄存器在 Bootloader 的第三阶段的初始化函数中进行填充。

在 Scorpius 平台中，中断向量表采用汇编编写，中断向量表主要位于代码段的 vecs 段，采用 1024 字节对齐。对应 16 个 ISFP，有 16 个标号的_vector 组成，

每个标号都是一个大小不超过 32 字节的中断处理代码，由于 32 字节长度太少，不足以完成中断处理，Scorpius 平台采用子中断处理程序的方式来进行中断处理。32 字节的处理代码中，仅仅对寄存器进行保存，然后使用跳转指令跳转到子中断处理程序。需要额外注意的是，由于 DSP 的特殊性，如果要等某些指令执行成功，必须要等待一段时间，比如在使用加载指令后，必须要等待 4 个周期加载指令才会执行完毕。中断向量的具体代码如下所示：

```
        .sect ".text:vecs"   // 中断向量表位于代码段的 vecs 段中
        .align 1024   // 中断向量表采用 1K 对齐
_vectors:
_vector0:    VEC_ENTRY _c_int00      ;RESET
_vector1:    VEC_ENTRY _vec_dummy    ;NMI
_vector2:    VEC_ENTRY _vec_dummy    ;RSVD
_vector3:    VEC_ENTRY _vec_dummy
_vector4:    VEC_ENTRY c_int4
_vector5:    VEC_ENTRY c_int5
_vector6:    VEC_ENTRY c_int6
_vector7:    VEC_ENTRY c_int7
_vector8:    VEC_ENTRY c_int8
_vector9:    VEC_ENTRY c_int9
_vector10:   VEC_ENTRY c_int10
_vector11:   VEC_ENTRY c_int11
_vector12:   VEC_ENTRY c_int12
_vector13:   VEC_ENTRY c_int13
_vector14:   VEC_ENTRY c_int14
_vector15:   VEC_ENTRY c_int15
        .end
```

其中每个 VEC_ENTRY 对应一个 ISFP，Scorpius 平台中的 ISFP 使用汇编宏的方式编写。关键代码如下所示：

```
    STW         B0,     *--B15    // 保存 B0 寄存器
    MVKL    addr, B0    // 将子中断处理程序加载到 B0 寄存器中
    MVKH    addr, B0
    B       B0    // 跳转到子中断处理程序
    LDW         *B15++,     B0    // 恢复 B0 寄存器
```

```
    NOP     2   // 在 C6678 中，LDW 操作必须在 4 个 NOP 周期后生效
    NOP
    NOP
```

在 TMS320C6678 中，DSP 使用 B15 通用寄存器作为栈指针，使用 B3 通用寄存器作为函数调用返回地址，使用 A4，A5 作为第一个参数的传递寄存器，使用 B4，B5 作为第二个参数的传递寄存器。虽然 DSP 拥有 B0-B31，A0-A31 总共 64 个 32 位通用寄存器，但是在中断响应过程中不是所有 64 个寄存器都需要保存。在 Scorpius 平台中仅仅对于 B0-B15，A0-A15 这 32 个寄存器进行了保存，根据 TMS320C6678 的嵌入式应用二进制接口规范，操作系统在处理中断时仅需要针对 B0-B15，A0-15 这 32 个寄存器进行保存。下面给出一个具体的子中断处理函数的代码：

```
c_int4:
    isrSaveContext   // 保存上下文
    MVK         0x4, A4   // 将中断号保存在 A4 中，A4 寄存器用于保存第一个参数
    CALLP       isr_handler_entry,B3   // 调用 isr_handler_entry 函数并将返回地址保存在 B3
中
    isrResumeContext   // 恢复上下文
```

上述代码中的 isrSaveContext 和 isrResumeContext 均为宏，这两个宏的主要用途就是保存和恢复上下文。

为了让 DSP 内核能够正确的相应到某个中断，除了中断向量表的编写还需要针对 CorePac 的中断控制器的驱动进行实现。在 TMS320C6678 中，CorePac 外部的片上中断控制器（Chip Interrupt Controller，简称 CIC）将外部事件映射成 128 个系统事件（System Event）。而 DSP 内核只能处理 12 个可屏蔽中断事件和 1 个不可屏蔽中断事件。这两者之间的映射就是依靠 CorePac 的中断控制器单元来实现的。CorePac 的中断控制器主要由事件标志寄存器组（Event Flags Registers，简称 EFR），事件合成器寄存器组（Event Combiner Registers，简称 EvCR），异常合成器寄存器组（Exception Combiner Registers，简称 EcCR）和中断选择子寄存器组（Interrupt Selector Registers，简称 ISR）组成。EFR 用于记录，设置 128 个外部事件的发生情况，如果某个外部事件发生， EFR 中会对于这个事件进行记录。EvCR 可以将事件 4 到事件 31，事件 32 到事件 63，事件 64 到事件 95，事件 96 到事件 127 分别合并成新的组合事件 0，组合事件 1，组合事件 2，组合事件 3。需要特别注意的是，事件 0 到事件 3 在系统中是保留的。在使用 EvCR 的情况下，如果组

合事件发生，程序需要检查 EFR 来确认具体发生的系统事件。EcCR 的功能和 EvCR 类似，EcCR 将外部事件进行组合，合并成一个异常事件，该异常事件会记录到 DSP 内核的 EFR 寄存器的 EXF 位上。当 DSP 内核发生异常时，如果采用了 EcCR，那么必须检查 EcCR 和 EFR 来确认具体发生的异常。ISR 用于将 128 个外部事件映射到 12 个 CPU 内核中断上。ISR 主要由 3 个中断开关寄存器（Interrupt Mux Registers，简称 IMR）组成。这三个 IMR 每个映射 4 个 DSP 中断，其具体原理如下图所示。IMR1 寄存器映射 DSP 中断 4 到中断 7，如果用户需要将外部事件 100 映射到 DSP 中断 4 上，只需要将外部事件号 100 填写到寄存器的 INTSEL4 位域中即可完成，图 4-6 给出了 IMR 寄存器的结构。

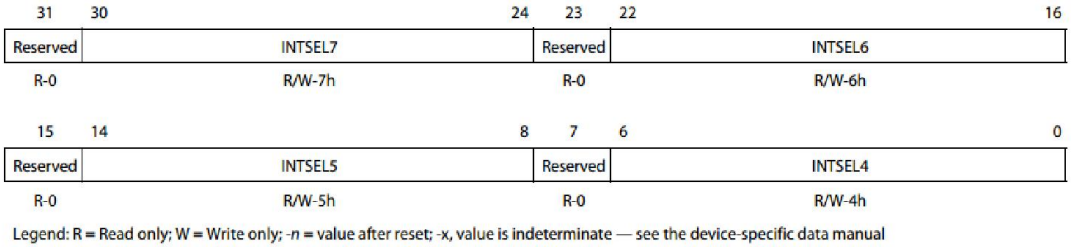

图4-6 IMR1 寄存器

组成图 4-7 描述了 128 个外部系统事件是如何映射到 DSP 的 12 个可屏蔽中断的，注意 DSP 内核中断 INT2 和 INT3 是不使用的，INT1 用于不可屏蔽中断，INT0 用于复位。

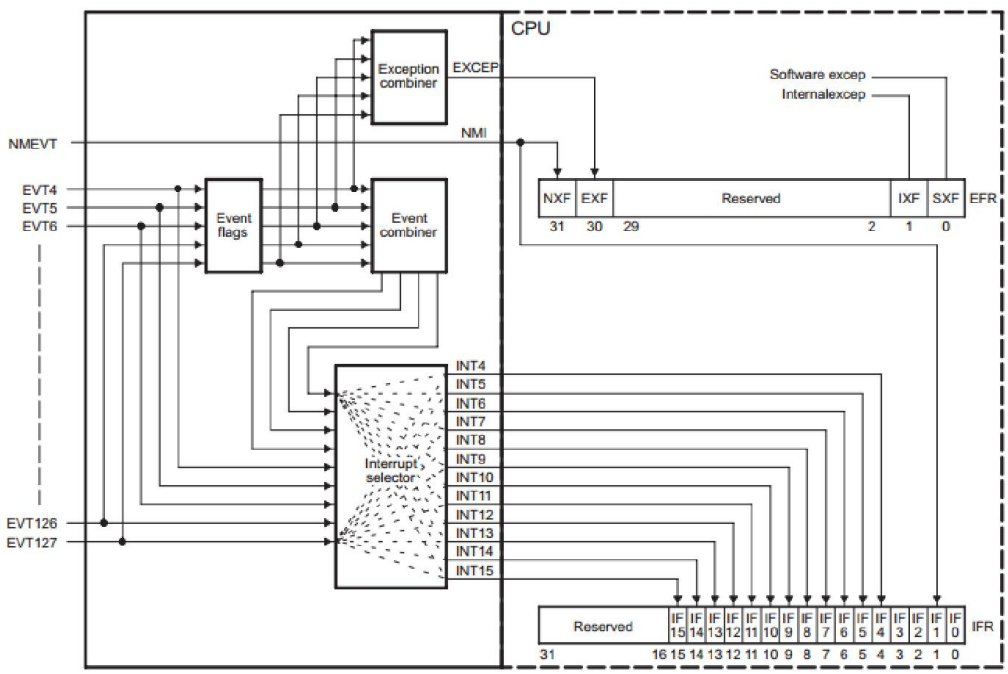

图4-7 外部事件到内部中断的映射

在 Scorpius 平台中，使用 corepac_int_func_sets 数据结构来代表一系列中断映射，使能，失能以及查询的功能集合。这个数据结构由许多函数指针组成，具体代码如下所示：

```
struct corepac_int_func_sets {
    int (*corepac_get_event_flag)(int efnum);   // 判断给定外部事件是否发生
    int (*corepac_set_event_flag)(int efnum);   // 设置某个外部事件，用于调试
    int (*corepac_clr_event_flag)(int efnum);   // 清除某个外部事件，中断中使用
    void (*corepac_mask_combine_event)(int efnum);   // 在组合外部事件中，屏蔽某个外部事件
    void (*corepac_unmask_combine_event)(int efnum);   // 在组合外部事件中，清楚被屏蔽的外部事件
    unsigned long (*corepac_mask_combine_read_event)(int efnum);   // 在组合外部事件中，找到发生的具体外部事件事件号
    int (*corepac_map_dsp_int)(int efnum, int dsp_int);   // 将一个外部事件映射到一个 DSP 内部中断上
    void (*corepac_mapped_dsp_ints_get)(int *array);   // 返回所有 DSP 内部中断对应的外部事件号，该函数用于调试
    int   (*corepac_mapped_event_get)(int dsp_int);   // 返回给定 DSP 内核中断所对应的外部事件号。
};
```

该数据结构保存在 corepac_intc_dev 结构中，corepac_intc_dev 数据结构表示某个 CorePac 的 CorePac 中断控制器设备。在 Bootloader 第三阶段的硬件初始化函数中，系统在初始化 CorePac 时会初始化对应的 CorePac 中断控制器，并将该设备注册到 Scorpius 平台。

4.1.5 用户交互模块

Scorpius 平台目前只实现通过串口的方式进行用户交互。用户可以使用传统的 printf 和 scanf 等输入输出函数来实现对串口的操作。用户交互模块由两个层次组成，底层是具体交互设备的驱动程序，上层是对交互设备进行抽象的输入输出设备层。

TMS320C6678 的通用串口是基于工业标准的 TL16C550 异步通信模块，该模块是 TL16C450 的升级版本，在功能上类似 TL16C450。TM320C6678 的串口支持 FIFO 中断模式和 FIFO 轮询模式。由于串口仅仅负责用户输入输出数据，对于这

些单次传输数据量不大但传输次数频繁的情况，中断方式反而没有轮询方式有效。在 Scorpius 平台上，采用的是 FIFO 轮询模式。Scorpius 平台上的串口初始化流程如下图 4-8 所示。

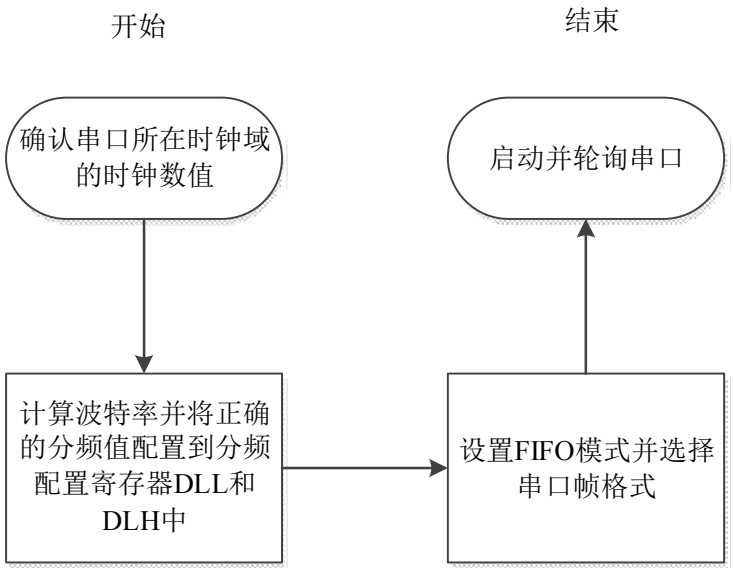

图4-8 串口驱动初始化流程

为了方便管理和扩展，用户交互模块在具体交互设备驱动上面抽象了一个抽象交互设备。该设备将用户交互设备的一系列共性提取出来，每个具体的用户交互设备都包含了一个抽象交互设备。

Scorpius 平台使用 io_device 结构体来表达一个抽象交互设备，所有具体的驱动设备必须包含一个 io_device 成员。当驱动程序初始化具体交互设备时，也同时必须针对抽象交互设备进行初始化。驱动程序需要将该设备通过 register_io_device 函数向上注册到 Scorpius 平台当中以便后续针对交互设备的管理。io_device 的数据结构如下所示：

```
struct io_device dev {
    struct device dev;   // 代表驱动设备的结构体
    struct io_operation ops;   // 交互设备操作集合，定义了一系列用于读写用户数据的函数指针
    int dev_kind;   // 具体交互设备种类
    char *io_buffer;   //   驱动缓冲区
    char *dma_buffer;   // DMA 缓冲区
    struct dma_device *dma_dev; // 对应的 DMA 设备驱动
    struct list_head io_devs;   // 用于将所有交互设备联系在一起的链表
```

```
};
```

4.1.6 其他辅助模块

其他辅助模块主要包括 DSP 内部控制寄存器的直接控制操作和 Cache 及 Writer Buffer 的控制与操作。

DSP 内部包括了以下三组控制寄存器：控制寄存器，扩展控制寄存器，浮点操作扩展控制寄存器。其他辅助模块提供了很多汇编或 C 语言编写的函数来描述这些寄存器的功能。这里简单介绍一些重要的寄存器：

寻址模式寄存器（Addressing Mode Register），用于控制 A4-A7，B4-B7 通用寄存器的线性或循环寻址。

控制状态寄存器（Control Status Register），包含了一系列位来控制 DSP 的当前状态，其中 EN 位控制 DSP 当前字节序，GIE 位控制 DSP 内部中断是否使能。

中断清除寄存器（Interrupt Clear Register），用于手动清除 12 个可屏蔽中断。

中断状态寄存器（Interrupt Flag Register），用于判断 12 个可屏蔽中断是否发生。

中断返回寄存器（Interrupt Return Pointer Register），当中断发生时用于保存中断的返回地址。在结束中断处理的时候使用 B IRP 指令即可退出中断。

中断使能寄存器（Interrupt Enable Register），用于开启或关闭某个特定的可屏蔽中断，设备默认情况下所有 12 个可屏蔽中断都是关闭状态的，

中断向量表基地址寄存器（Interrupt Service Table Pointer Register），用于设置中断向量表的基地址，中断向量表的基地址必须 32 字节对齐。

时间戳计数寄存器（Time Stamp Counter Registers），两个 32 位寄存器构成的 64 位寄存器，用于记录当前 CPU 的时钟数。

其他辅助模块除了负责提供针对 DSP 内部控制寄存器的操作外，还提供了针对 DSP Cache 以及 Writer Buffer 的控制与管理。TMS320C6678 的 L1 Cache 和 L2 Cache 都实现了 SRAM 功能，这些 Cache 可以通过一系列 Cache 控制寄存器来配置成 Cache 或者 SRAM。TMS320CC6678 的 Writer Buffer 宽度为 128 位，Cache 行长度为 32 字节。其他辅助模块提供了针对 L1 Cache，L2 Cache 的写回，无效等操作，除此之外还提供针对 Writer Buffer 的同步操作。针对 Cache 和 Writer Buffer 以及 CorePac 的 Prefetch Buffer 的同步与一致性问题不在本模块的讨论范围内，这些内容属于多核同步模块管理，其他辅助模块仅仅提供一些简单操作函数，并不提供系统平台实现上的策略。

4.2 多核基础支持模块

4.2.1 CorePac 与 CorePac 组群

Scorpius 平台中的每个 CorePac 代表了一个具体的处理器核心。为了支持多核操作系统的运行，Scorpius 平台针对 CorePac 进行分组，一个操作系统下的所有处理器属于同一个 CorePac 组群。如果 CorePac 的 L2 内存被配置成 SRAM，则 CorePac 只能访问同一组的 L2 内存，不能访问其他组内存。外部共享内存同样会根据 CorePac 组群进行划分，CorePac 不能访问其他组的共享内存。Scorpius 平台提供了一个特殊的共享区域，所有 CorePac 均可以访问这个共享区域。下图描述了 CorePac 以及 CorePac 组群之间的内存访问规则，图 4-9 中将 L2 内存全部配置成了 SRAM。

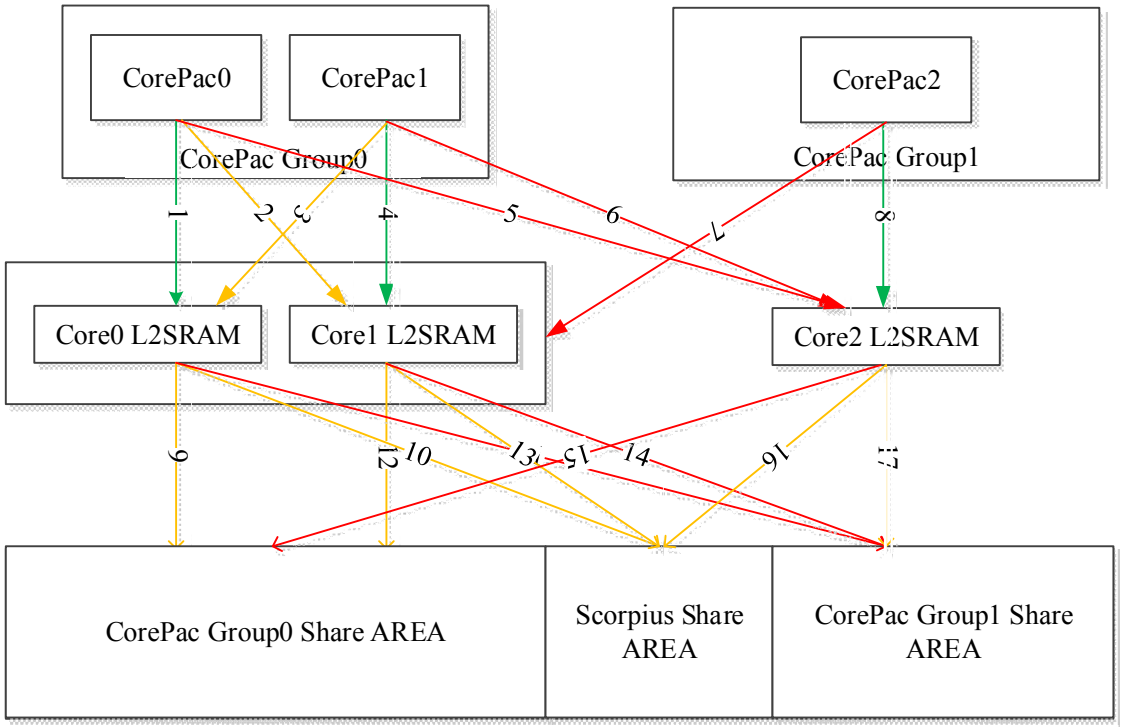

图4-9 CorePac 组之间的内存访问示意

图 4-9 中，1、2、3、4、8、9、10、11、12、16 和 17 的访问路线是合法的，其中 1、4 和 8 的访问效率要远高于其他访问。5、6、7、13、14 和 15 的访问将触发内存保护异常。具体规则描述如下：

情况 A：访存路线 1，访存路线 4，访存路线 8 描述了 CorePac 访问属于自己的 L2 内存，访问速度最快。

情况 B：访存路线 2，访存路线 3 描述了对同一组群下不同 CorePac 的 L2 内存的访问，这种访问效率远低于情况 A 应该避免。

情况 C：访存路线 5，访存路线 6，访存路线 7 描述了对不同组下 L2 内存的访问，这种访问由 L2 内存控制器所禁止，访问将触发内存保护异常。

情况 D：访存路线 9，访存路线 12，访存路线 17 描述了对同组共享内存的访问，这种访问是允许的，但是性能要弱于访问 L2 内存，并且访问 L3 内存的性能要数百倍好于访问 L4 内存的性能。

情况 E：访存路线 10，访存路线 11，访存路线 16 描述了对 Scorpius 平台组间共享内存的访问，这种访问是允许的。

情况 F：访存路线 13，访存路线 14，访存路线 15 描述了对不同组间共享内存的访问，这种访问被 CorePac 扩展内存控制器中的内存保护单元所禁止，访问将触发内存保护异常。

Scorpius 平台使用 corepac 数据结构来表示 CorePac，该数据结构的关键成员如下所示：

```
int id;  // CorePac 编号，对于 TMS320C6678 是从 0 到 7 总共 8 个
int prio_tera_net;  // 该 CorePac 在 TeraNet 总线上的优先级，优先级越高在总线竞争时就越容易获胜
int aid;  // 与内存保护有关的成员，稍后会介绍
struct device dev;  // 对应的设备，用于注册到 Scorpius 平台统一管理
struct corepac_group *cg;  // 所在的 CorePac 组
int cg_id;  // 在 CorePac 分组中的编号，用于随机访问
int bp;  // 表明本 CorePac 是否为启动处理器，如果是该成员为其 CorePac 组的编号，如果不是则为 0。
struct list_head cg_node;  // 在 CorePac 分组中将 CorePac 链接在一起的链表，CorePac 分组使用链表和数组两种方式保存 CorePac
struct corepac *bp_c;  // 指向当前 CorePac 的启动处理器，如果是自己则为 NULL
struct core_cfg *cfg;  // 该 CorePac 的配置信息
struct corepac_int_dev *int_dev;  // 该 CorePac 的中断控制器
struct cache *l1_cache, *l2_cache;  // L1，L2 内存控制器
struct rb_root mem_area_root;  // 本 CorePac 所有内存区域组成的红黑树的根，具体内容见 4.4.2.2 小节
void (*memory_protection_handler) (unsigned long addr);  // 当发生内存保护异常时，该 CorePac 调用的处理函数
```

corepac_group 数据结构用来表示一个 CorePac 分组，该数据结构的关键成员如下所示：

```
int id;  // 该分组编号
int bp_num;  // 当前 CorePac 组群中启动处理器所在序号
char name[CGROUP_NAME_SIZE];  // 分组名
int core_num;  // 该分组所有 CorePac 的数目
struct scorpius_os *os;  // 该分组所运行的操作系统信息
struct list_head cgroup_head;  // 该分组所包含 CorePac 组成的链表
struct corepac **corepac_array;  // 该分组所包含 CorePac 组成的数组，随机访问速度要快于链表
struct cas **allowed_cas_array; // 该分组所可以使用的比较-交换操作资源数组
int cas_num;  // 该分组所被分配的比较-交换操作资源数目
```

corepac 结构和 corepac_group 结构在 Scorpius 平台初始化中根据用户的配置完成初始化和注册。corepac_group 结构中的 allowed_cas_array 和 cas_num 成员的含义见 4.3.1 小节。

4.2.2 片上中断控制器模块

片上中断控制器 CIC 用于将大量外部事件进行分类和合并，从这些外部事件当中选择一些事件发送给 CorePac 或其他片上协处理器。TMS320C6678 包括 4 个 CIC 中断控制器，其中 CIC0 和 CIC1 为 8 个 CorePac 服务，而 CIC2 和 CIC3 则为包括 DMA 控制器在内的其他片上控制器或协处理器服务。

CIC 中断控制器模块主要实现以下几个具体功能：开启或关闭某个外部事件，记录某个外部事件是否发生，将外部事件映射到系统事件，针对外部事件进行优先级排队。这些功能在驱动角度都是通过对寄存器的配置而实现的。

Scorpius 平台为了将外部事件成功传导进 DSP 内部，除了配置 DSP 内部控制器，CorePac 中断控制器外还必须正确配置 CIC 中断控制器的事件映射模块。CIC 中断控制器的事件映射模块使用通道（channel）这一概念来描述事件的映射。CIC0 中断控制器和 CIC1 中断控制器的事件映射模块分别由 40 个通道映射寄存器构成，每个通道映射寄存器负责 4 个通道的映射，总共支持 160 个通道。下图 4-10 给出了一个通道映射寄存器的描述。

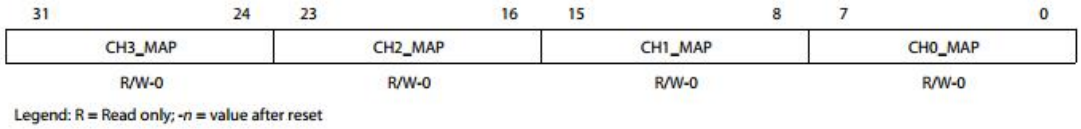

图4-10 通道映射寄存器

CIC 中断控制器模块的映射功能就是将系统事件的编号填写到相应的通道映射寄存器的通道中。CIC0 和 CIC1 中断控制器将 160 个外部事件映射成 128 个系统事件然后将 128 个系统事件派发给每个 CorePac。如果多个外部事件映射到了同一个系统事件上，CorePac 中断控制器中必须查看 CIC 中断控制器的事件标志寄存器来查看具体外部事件的种类。Scorpius 平台使用 cic_device 结构来表示一个 CIC 中断控制器，该数据结构主要包含了一个编号，一个 device 结构和一组 CIC 中断控制器具体操作的函数指针。

4.3 多核同步模块

4.3.1 比较交换指令

传统单核操作系统中，操作系统使用互斥锁或信号量的方式来保护多个进程间共享的数据。这些单处理器操作系统中的互斥锁和信号量是利用开关处理器局部中断来实现的。当操作系统应用在多处理器的环境下，这些互斥锁和信号量将无法保护那些被多处理器间共享的关键数据。图 4-9 中的访存路线 9 和访存路线 12 就描述了一个访问共享数据空间的情况。

多处理器系统中解决多处理器同步与互斥的方式是采用比较与交换指令（Compare-And-Swap，以下简称 CAS）。需要注意的是，CAS 指令并不是一种确定的指令而是一种操作方式，由于历史原因其命名来源于 X86 体系架构。几乎所有架构的处理器都定义了 CAS 指令，但是它们的操作方式各不相同，不过原理相同。CAS 指令定义了一种原子的操作序列，该序列可以是比较-交换（如 X86 体系中的 CMPXCHG 指令），可以是加载-存储（如 ARM 体系中的 STREX 和 LDREX 指令）。各体系通过各自不同的方式来保证该操作序列的原子性。当某个处理器执行序列时，如果此时恰好有另外一个处理器同时也访问同一个序列。那么同一时刻只有一个处理器能成功进入序列，而另一个处理器会显示的得知竞争失败。大多数多核操作系统在 CAS 指令的基础上设计出了针对多处理器系统的解决同步与互斥的方法，最典型的就是自旋锁。当一个处理器试图去访问某个共享资源时，该处理器需要先获取自选锁，如果该自旋锁已被其他处理器获取，那么处理器将在得知获取锁的操作失败后开始轮询该锁是否已经被其他处理器释放。

但是这种方式显然在指令集各异的异构多处理器的系统中无法实现。异构多处理器由于各处理器所采用的架构不同，如果设计统一的 CAS 指令操作则会导致设计成本和处理器性能的严重影响。为了解决这个问题，目前大多数异构多处理器系统都采用硬件模块的方式来模拟 CAS 操作的原子性。目前这种硬件实现方式有两种，一种是开辟一块特殊内存，所有处理器对这块内存的访问都是具有上述 CAS 指令的原子性，即成功则返回，如果被打断则失败。另一种则是提供一些所有处理器都能访问的共享寄存器，这些寄存器和上述内存一样具有 CAS 指令的原子性，即成功则返回，如果被打断则失败。两种实现各有优点和缺点，前者的优点是有近乎无限的资源可供多个处理器同时使用，越多的资源意味着越多的处理器可以做到对不同锁访问的并行性。但是缺点则是硬件设计困难和多个处理器必须承受访问共享特殊内存的缓慢速度。后者实现的优点则是拥有在进行 CAS 指令操作时拥有可预测的实时性速度，缺点则是资源数量有限且硬件成本较大，假设系统中只有一个共享寄存器，那么所有核心在进行 CAS 操作时总是只能有一个顺利成功，其他都会失败，这将直接退化成 X86 架构下的锁总线操作。

TMS320C6678 是一款面向实时性应用的嵌入式异构多处理器片上系统。其 CAS 指令采用的是硬件模块寄存器的方式实现。TMS320C6678 拥有 32 个支持 CAS 指令操作的寄存器，在理想情况下，可以满足同时 32 个 CAS 操作，这对于嵌入式设备来说是绰绰有余了。TMS320C6678 的硬件互斥模块有两种工作模式，一种是直接请求模式，另一种是非直接请求模式。两个模式的区别在于后者在资源空闲时会以中断的形式来通知发出请求的处理器，这种方式适合实现回调函数。

Scorpius 平台的 CAS 操作采用直接请求模式来实现，其关键需要解决以下几个问题：

1. 如何管理数目有限的 32 个共享寄存器。
2. 如何利用共享寄存器实现 CAS 操作。

对于第一个问题，Scorpius 平台设计了一个单独的硬件设备来表示这共享寄存器，该硬件设备用于分配和管理这 32 个有限的共享寄存器资源。Scorpius 平台使用 cas_manager_dev 数据结构来表示该设备，该数据结构的主要成员如下：

```
int cas_bit_maps[ARCH_CAS_GROUP];   // 用于资源分配的位图
int ava_cas_num;   // 可使用的资源数量
int total_cas_num;   // 总共支持的资源数量，数值等于 ARCH_CAS_NUM
struct device dev;   // cas_manager_dev 在上层以设备的形式展现给用户
struct cas cas_set[ARCH_CAS_NUM];   // 共享资源
void (*require) (int id);   // 进入关键区域时进行 CAS 操作所需硬件操作的函数指针
```

```
    void (*release) (int id);   // 退出关键区域时进行 CAS 操作所需硬件操作的函数指针
    int (*try) (int id);   // 尝试进入关键区域时进行 CAS 操作所需硬件操作的函数指针，该操作不同于 require 的地方是 require 在获取失败时将发生自旋。
```

require，release 和 try 函数指针在 cas_manager_dev 被初始化的时候赋值成_dev_cas_require，_dev_cas_release 和_dev_cas_try 函数，这几个函数将根据函数的参数 id 来直接操作寄存器。TMS320C6678 中的 CAS 资源寄存器的结构如下图 4-11 所示：

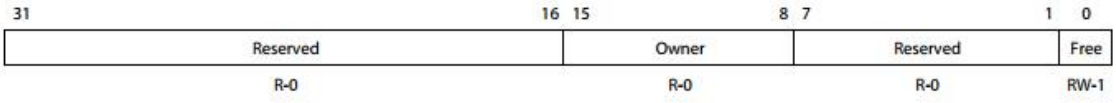

图4-11 CAS 资源寄存器

读写 CAS 资源寄存器的 Free 位将根据 CAS 资源寄存器的当前状态得到不同的值。如果读取 Free 位的值为 1 则说明该 CAS 资源目前没有其他处理器访问，可以被当前处理器获取。如果读取 Free 位的值为 0 则说明该 CAS 资源目前已经被其他处理器获取，不能被当前处理器获取。对 Free 位写 1 将释放该 CAS 资源。在_dev_cas_try 函数中，如果读取 Free 位为 0 则直接返回 0，读取 Free 位为 1 则直接返回 1.在_dev_cas_require 函数中，如果读取 Free 位为 0 则等待 10 个周期（NOP 10）后继续轮询 Free 位，直至 Free 位有效才返回。这三个函数在获取资源失败发生自旋的时候，会调用其他辅助模块提供的 nop 函数，nop 函数内部会调用底层汇编代码提示当前运行代码的处理器进入低功耗模式，低功耗模式将会在下一次外部中断中被唤醒。这三个函数的代码如下所示：

```
//  获取 CAS 资源，失败则轮询
void _dev_cas_require(int id)
{
    lock_irq_cg();      // 关闭整个 CorePac 群组的中断，防止出现死锁
    while(!(readl(SEM_DIRECT(id)) & 0x1))
        nop();
}
//  释放 CAS 资源
void _dev_cas_release(int id)
{
    writel(0x1, SEM_DIRECT(id));
    unlock_irq_cg();      // 恢复整个 CorePac 群组的中断
```

```
}
//   试图获取 CAS 资源，如果失败则返回 0，成功则返回 1
int _dev_cas_try(int id)
{
      if (readl(SEM_DIRECT(id)) & 0x1) {
            lock_irq_cg();       //关闭整个 CorePac 群组的中断，防止出现死锁
            return 1;
      } else
            return 0;
}
```

数据结构 cas 用于表示一个单独的 CAS 操作单元，该数据结构是 Scorpius 平台支持 CAS 操作的核心，其定义如下所示：

```
int id;   // 该 CAS 资源的编号
int kind;   // 用于区别原子操作，快速自旋锁和普通自旋锁
struct cas_manager_dev *dev;   // 指向 CAS 资源管理设备
void (*enter_critical) (struct cas *);   // 进入关键区域的函数指针
void (*exit_critical) (struct cas *);   // 退出关键区域的函数指针
int (*try_critical) (struct cas *);   // 尝试进入关键区域的函数指针
```

cas 结构体中的 id 对应了 cas_manager_dev 中 cas_set 的序号，其取值为 0 到 31。enter_critical，exit_critical，try_critical 是三个函数指针，这三个函数指针在 cas 初始化的时候会赋值为_cas_enter_critical，_cas_exit_critical 和_cas_try_critical。这三个函数会先对参数进行检查，然后直接调用 cas_manager_dev 的 require，release 和 try 函数。Scorpius 平台上层使用三个宏来方便用户使用 cas 资源，这三个宏的代码如下：

```
#define ATOMIC_ENTER(cas)    \
      (cas)->enter_critical((cas)->id)
#define ATOMIC_EXIT(cas)\
      (cas)->exit_critical((cas)->id)
#define ATOMIC_TRY(cas) \
      (cas)->try_critical((cas)->id)
```

对于 TMS320C6678 来说，有 32 个 CAS 硬件资源，这样就总共有 32 个 cas 结构体。理论上可以仅使用 1 个 CAS 硬件资源来实现所有的同步和互斥工作，但

是这种做法的效率十分差，和直接锁总线没有任何区别。在 Scorpius 平台中这些 cas 保留 16 个 CAS 硬件资源用于实现性能较好的快速自旋锁（Fast Spinlock），剩余 16 个 CAS 硬件资源分成 8 组给 CorePac 组群使用，每个组群拥有至少 2 个 CAS 硬件资源，其中 1 个 CAS 硬件资源用于实现自旋锁，另一个 CAS 硬件资源用于实现原子操作。这种将多个 CAS 硬件资源分开使用的方法可以大幅度提高系统的并行性能，减少因为不同锁竞争而引起的处理器阻塞。

4.3.2 原子变量，自旋锁，快速自旋锁

单独依靠 CAS 操作来实现多处理器间的同步与互斥是十分复杂且对用户不友好的。Scorpius 平台在 4.3.1 节的 cas 数据结构的基础上进行扩展，发展出原子变量，自旋锁，快速自旋锁三种基础同步与互斥工具。

原子变量指的是一些类型为 int 或 long 的特殊基础变量，这些变量和基础变量 int 或 long 一样可以进行加减比较等运算，但是和基础变量不同的是这些原子变量的加减比较等运算具有原子性。所谓原子性就是指这些变量的加减比较等操作不能被打断，要么动作执行完毕要么不执行，不存在其他情况。Scorpius 平台使用两种原子变量 atomic_slong 和 atomic_ulong，前者是对 signed long 的包装，后者则是对 unsigned long 的包装，这两个原子变量共享一个 CAS 资源。这两个原子变量的数据结构如下所示：

```
struct atomic_slong {
    long val;   // 原始数据类型
    struct cas *c;   // cas 资源
};
struct atomic_ulong {
    unsigned long val;   // 原始数据类型
    struct cas *c;   // cas  资源
};
```

原子变量提供了一系列加减比较的操作，下面以 atomic_inc_ul 为例描述原始数据类型与 CAS 资源如何协同实现原子变量操作：

```
// 原子变量的自增操作
void atomic_inc_ul(struct atomic_ulong *ul)
{
    ATOMIC_ENTER(&ul->cas);
    ul->val++;
```

```
    ATOMIC_EXIT(&ul->cas);
}
```

自旋锁的基础概念在 4.3.1 节中已经描述了。这里再针对 Scorpius 平台对其实现进行细致描述。Scorpius 平台实现了两类自旋锁，一类是普通自旋锁，使用数据结构 spinlock 代表，另一类是快速自旋锁，使用数据结构 fast_spinlock 代表。前者是基于 cas 数据结构实现的，后者是直接基于 CAS 硬件模块实现的。前者的优点是数目接近无限，缺点是性能较差，后者则刚好相反。在实时性较高，并且系统共享资源较少的情况下，用户应该使用后者而非前者。spinlock 和 fast_spinlock 的数据结构相同，关键成员如下所示：

```
    int lk_val;
    struct cas *c;
```

这两个数据结构虽然形式一模一样，但是对于锁的操作却完全不一样，都使用 cas 数据结构的原因是方便 Scorpius 对他们进行监控和管理。下面给出 spinlock 和 fast_spinlock 的上锁操作函数的关键代码：

```
void spin_lock(struct spinlock *lock)
{
busy:
    ATOMIC_ENTER(&lock->cas);
    if (lock->lk_val) {
        ATOMIC_EXIT(&lock->cas);
        goto busy;      // 自旋操作
    } else
        lock->lk_val = 1;
    MEM_FENCE();      // 内存屏障
    ATOMIC_EXIT(&lock->cas);
}

void fast_spinlock(int id)
{
    if (id < FAST_LOW_BOUND || id > ARCH_CAS_NUM) {
        BUG();
        return ;
    }
```

```
    while(!(readl(SEM_DIRECT(id)) & 0x1));
}
```

从实现可以看出，快速自旋锁直接使用 Scorpius 平台中保留的 CAS 硬件模块资源，这些资源是独占的，所以基本不会和其他处理器产生冲突，故性能较好。而普通自旋锁则是基于 cas 数据结构实现，在 CorePac 组群内容易和其他处理器发生冲突，因此性能较差。

4.3.3 缓存一致性

一般多处理器系统中，缓存一致性是依据总线监视模块所采用的缓存一致性协议而实现的。在 TMS320C6678 中，其总线监视模块并没实现任何多处理器间的缓存一致性协议。TMS320C6678 的总线监视模块仅实现了针对处理器与 DMA 间的缓存一致性协议，该缓存一致性协议仅针对 CorePac 的 L1 内存和 DMA。Scorpius 平台仅在其他辅助模块的基础上提供一系列针对缓存的基本操作，这些操作包括缓存写回，缓存强制失效并写回，缓存强制失效以及改变缓存的缓存策略。运行在 Scorpius 平台上的操作系统应该在系统初始化的时候通过其他辅助模块将缓存设置成写穿（Write-Though）模式或者相关共享内存为不可缓存状态。除此之外，用户操作系统也可以通过其他辅助模块提供的缓存操作函数手动无效或写回缓存。

4.3.4 内存屏障

内存屏障也称内存栅栏，屏障指令等，是一种同步屏障指令，使得处理器或编译器在对内存随机访问的操作存在一个同步点，使得此点之前的所有读写操作都执行后才能开始执行此点之后的操作[31]。TMS320C6678 提供了一个 MFENCE 指令来进行内存同步操作，该指令将阻塞指令预取操作，直到内存系统的忙标志（Busy Flag）被清除。该指令的效果是，该指令之前的所有内存访问操作，在该指令之后都会被其他指令看到。Scorpius 平台利用该指令提供了一个 MEM_FENCE 函数来向用户操作系统提供内存屏障操作。

4.4 内存管理模块

内存管理是任何平台不可或缺的关键组件，Scorpius 平台的内存管理模块主要负责的是平台内部动态内存的管理和保护。而上层用户操作系统的动态内存管理由操作系统自己负责进行，Scorpius 平台在操作系统支持模块中的内存管理模块会

将大块内存分配个 CorePac 组群，每个 CorePac 组群管理一个用户操作系统，该用户操作系统的动态内存就来源于该大块内存。

4.4.1 平台内存管理

4.4.1.1 内存布局

TMS320C6678 根据内存到 DSP 内核的距离将内存分为 4 个级别，这四个级别中 L3 和 L4 内存位于 CorePac 外部，属于共享内存。L1 和 L2 内存位于 CorePac 内部，属于 CorePac 独占内存。对于这些内存的访问速度是随着这些内存级别的提高而减少的。L1 和 L2 内存可以通过 CorePac 的缓存控制器配置成缓存。Scorpius 平台中 L1 内存默认被初始化为 L1 缓存，L2 内存默认初始化为 SRAM。Scorpius 平台需要管理的内存是 L2 SRAM 内存，L3 多核共享内存，L4 DDR 内存。

操作系统或其他系统软件是通过链接脚本来决定系统运行后整个内存布局的。链接脚本是在软件编译链接过程由链接器使用的配置脚本文件。它的作用是根据用户的规定决定如何把输入文件内的段放入输出文件，并控制输出文件内各部分在程序地址空间内的布局。

Scorpius 平台是在 TI 的 CCS 集成开发环境中开发编译链接的，TI 的 CCS 集成开发环境使用的是 TI 编译器专用的链接脚本。Scorpius 平台的内存布局设计也就是规划与编写应用于 CCS 集成环境的链接脚本。在 Scorpius 平台中 L2 内存由于是 CorePac 独享，所以设计成处理器独占内存区，在该区域的内存对象将会在每个 CorePac 中都有一份备份。L2 内存的大小是每个 CorePac 独享 512K 的内存。L3 多核共享内存用于实现多核共享区，稍后章节的核间中断机制就是基于该多核共享区实现的。由于 L3 内存的访问速度远远优于 L4 内存，所以 Scorpius 的小内存对象分配器——SSLAB 分配器中的部分对象会保存在 L3 内存中，这种设计可以提高平台对于这些常用对象的访问效率。除此之外，L3 多核共享内存还为每个 CorePac 群组保留了一个小的内存区域用于存放一些关键数据。L3 多核共享内存的大小是总共 4MB，其中 2MB 用于 SSLAB 分配器分配常用对象，1MB 用于实现 SBinder 设备，SBinder 设备是一种服务于核间中断以及 CorePac 群组通信的虚拟设备。1MB 用于 CorePac 群组保存关键数据。L4 DDR 内存用于保存用户操作系统的代码段，数据段，堆栈等，由于群组之间不能交叉访问彼此的内存，所以 Scorpius 平台将 L4 DDR 内存划分成 CorePac 群组数加 1 个内存空间。每个 CorePac 群组享用一个内存空间，而剩下 1 个内存空间则保留给 Scorpius 平台的伙伴系统和 SSLAB 分配器以及平台自身的代码段，数据段，堆栈等。图 4-12 给出了各个内存区域的布局情况。

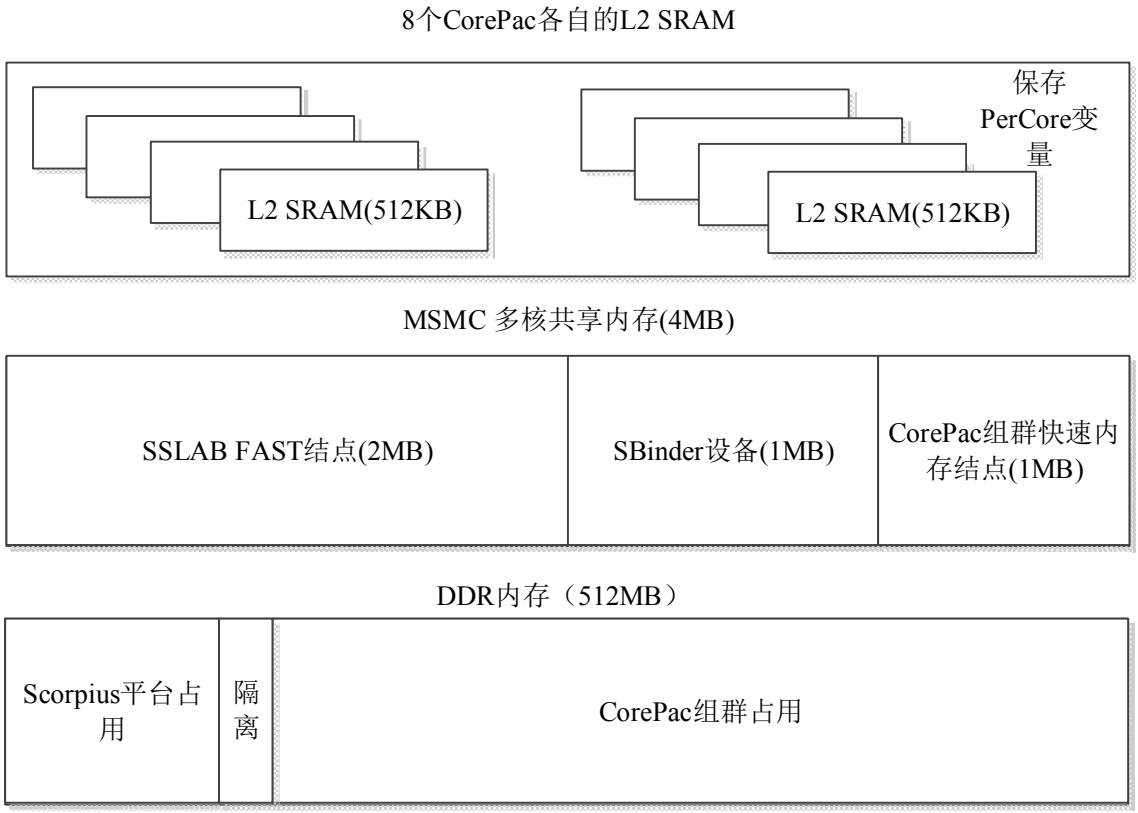

图4-12 Scorpius 平台内存布局

CorePac 组群的快速内存结点采用控制结构头部加内存块的方式进行管理。控制结构头部使用位图的方式记录内存块的使用情况。根据上述内存布局的设计，Scorpius 平台的链接脚本的内存区域实现如下：

```
MEMORY
{
    L2_PER_CORE_SECT: origin = 800000h    length = 80000h     /* 512K */
    SSLAB_FAST_SECT: origin =    c000000h length = 200000h /* 2MB */
    SBIND_SECT: origin = c200000h length = 100000h /* 1MB */
    COREPAC_L3_SECT: origin = c300000h length = 100000h /* 1MB */
    L4_DDR: origin = 80000000h length = 20000000h /* 512MB */
}
```

4.4.1.2 伙伴系统

Scorpius 平台的动态内存管理用于平台内部内存的分配与释放，这些内存全部用于平台内部而不用于上层用户操作系统。Scorpius 平台的动态内存管理采用伙伴算法来实现。伙伴算法是一种经典的内存分配算法，Linux 底层的内存管理使用的

就是伙伴算法。实质上，伙伴算法是一种特殊的“分离适配”，即将内存按照 2 的幂进行划分，相当于分离出若干个块大小相同的空闲链表，搜索该链表并给出同需求最佳匹配的大小，其优点是快速搜索合并以及低外部碎片[32]。

该算法的分配过程大致如下：

1. 寻址大小合适的内存块，如果所要求的内存块不是 2 的幂则分配最接近并且满足要求的内存块。

2. 如果没有找到，则将高阶的内存块分成两个低阶内存块，比较是否满足分配的最小需求。如果不满足则继续进行高阶内存块的分离，直到达到需求。

该算法的释放过程大致如下：

1. 寻址需要释放内存的伙伴，如果伙伴被分配出去则单独释放当前内存块直到伙伴被释放到伙伴系统当中。

2. 如果伙伴被释放到伙伴系统，则和伙伴合并成一个高阶内存块，寻址该高阶内存块的伙伴。如果找到高阶内存块的伙伴，则继续进行合并行为，直到合并到系统允许的最高阶内存块或找不到对应的伙伴为止。

Scorpius 平台伙伴系统最小可分配 4KB 大小的内存块（阶 0），最大允许分配 64MB 大小的内存块（阶 14）。

4.4.1.3 SSLAB 分配器

上节中讨论了 Scorpius 平台用于动态内存管理的伙伴系统，该系统可以分配最小 4KB 大小的内存块。Scorpius 平台内部大量内存分配都是远远小于 4KB 的，如果这些内存的分配与释放都直接使用伙伴系统，那么势必会造成很大的浪费。Scorpius 平台因此改良了一个来自 SunOS 操作系统的小内存缓存与分配算法，该算法同样应用于 Linux 操作系统中，Scorpius 平台将其命名为 SSLAB 分配器（Scorpius Slab Allocator，简称 SSLAB 分配器）。该分配器是围绕对象缓存进行的。Scorpius 平台中，会为有限的对象集中分配大量内存。SSLAB 分配器的思想就是将这些需要大量分配的对象进行集中保存，集中分配。SSLAB 分配器的做法有三个好处，第一是节省了从伙伴系统直接分配内存所花费的时间。第二是由 SSLAB 分配器分配的对象有非常好的缓存特性，这样可以加速程序对常用对象的访问。第三是将部分常用对象缓存到 L3 内存当中，L3 内存拥有远好于 L4 内存的访问速度，这样可以进一步提高访问这些对象的速度。图 4-13 描述了一个具体 SSLAB 实例内部的对象的保存情况，该 SSLAB 实例用于缓存 spinlock 结构体。

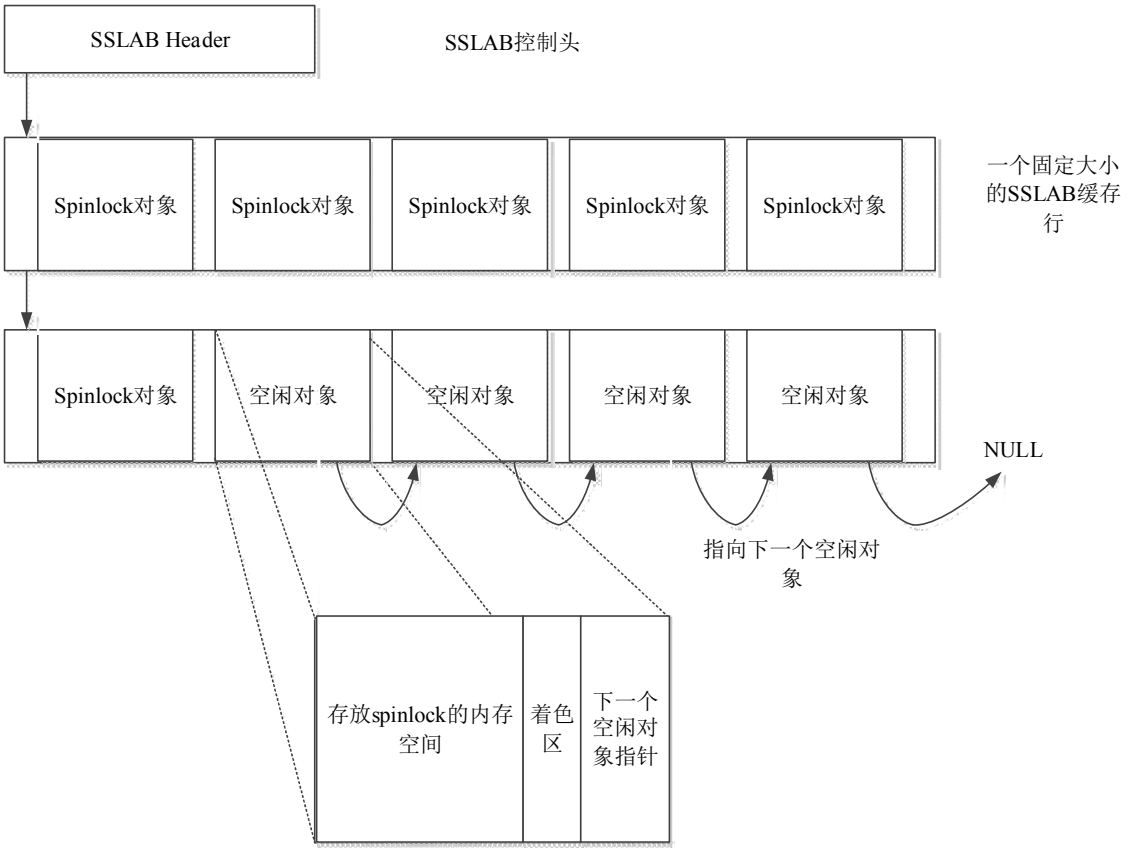

图4-13 SSLAB 构造

SSLAB 分配除了分配常用对象外，还负责小内存（小于 4K）的分配，这些小内存同样以 SSLAB 对象的形式保存在专用的 SSLAB 分配器中。这些 SSLAB 对象的大小分别是 16 字节，32 字节，64 字节，128 字节，256 字节，512 字节，1024 字节，2048 字节，总共 8 种 SSLAB 对象。SSLAB 分配器使用 kmalloc 函数和 kfree 函数来分配和释放这些内存对象，kmalloc 函数会根据申请的内存大小寻找最接近（大于等于）的 SSLAB 对象来进行分配。

4.4.2 分布式内存保护单元

4.4.2.1 内存保护实现原理

由于 Scorpius 平台中 CorePac 组群间是不允许直接访问彼此的内存的，为了实现这个功能必须依靠 TMS320C6678 的分布式内存保护单元的硬件功能。TMS320C6678 的内存保护单元并不像通用处理器中的内存管理单元（Memory Management Unit，简称 MMU）那样全部功能集中在一起，TMS320C6678 的内存保护单元分布在设备各个模块组件的内部，他们在原理和操作上保持一致，这种内存保护单元的架构方式称为分布式内存保护单元。分布式内存保护单元虽然零

零散散，但是在大致结构上可以分为三类：CorePac 内部内存保护单元，CorePac 扩展内存控制器的内存保护单元，其他协处理器，模块的内存保护单元。Scorpius 平台着重关心前两者的底层原理和实现。

对于 CorePac 的内存保护，在 Scorpius 平台上主要就是针对 L2 内存。L2 内存不允许其他 CorePac 组群的 CorePac 访问，仅允许自身或者同组群的 CorePac 的访问。TMS320C6678 的 L2 内存保护单元通过 4 种寄存器实现，这四种寄存器分别是 L2 内存保护页属性寄存器，L2 内存保护无效地址寄存器，L2 内存保护无效地址访问标志设置寄存器和 L2 内存保护无效地址访问标志清除寄存器。其中 L2 内存保护页属性寄存器实现了 32 个，这些寄存器是独立于 CorePac 的，每个 CorePac 只能设置自己的 L2 内存保护页属性寄存器。每个 L2 内存保护页属性寄存器负责一个页（32K）的内存属性保护。

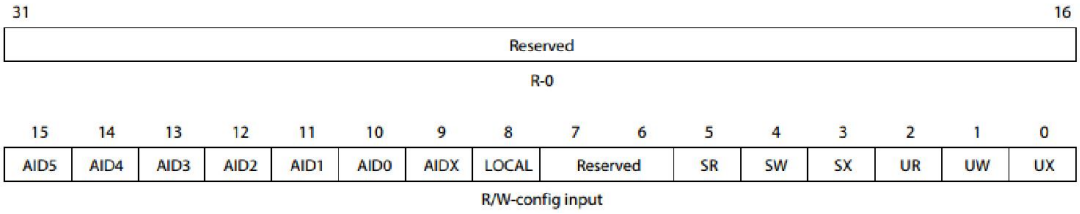

图4-14 L2 内存保护页属性寄存器

图 4-14 给出了 L2 内存保护页属性寄存器的结构，其中最低 6 个位给出了当前页的读写属性。TMS320C6678 中每个可以发起内存访问的模块都有一个 AID 号码，AID 号码在 CorePac 的外部内存控制器的优先级 ID 到 AID 映射寄存器中进行配置，AID 是针对本 CorePac 的，某个 CorePac 的 AID 在不同 CorePac 眼里可以是不同的，但是它们的优先级 ID 是相同不变的。第 8 位 LOCAL 的含义是本地 DSP 内核是否能访问当前 L2 页，第 9 位用来控制 AID 大于 AID6 的模块是否能访问当前 L2 页。这里需要注意的问题是 LOCAL 的权限大于其他 AID0 到 AIDX 位，也就是如果 LOCAL 设置了拒绝访问，不管 AID0 到 AIDX 如何设置，本 CorePac 都将无法访问本 CorePac 的 L2 内存。下面以一个运行了四个 CorePac 群组的用户配置为例，来描述 CorePac0 的 L2 内存保护页属性寄存器配置：

在一个有四个 CorePac 群组的 Scorpius 平台中， CorePac Group0（群组 0）由 CorePac0 和 CorePac1 组成。在该模型下，CorePac0 的 L2 内存允许 CorePac0 和 CorePac1 访问，不允许其他所有 CorePac 的访问。CorePac0 的优先级 ID（以下简称 PrivID）到 AID 映射寄存器配置如下：PrivID1 映射到 AID0，PrivID2 到 PrivID6 映射到 AID1，PrivID7 以上保持系统初始化时默认不变。在 L2 内存保护页属性寄存器中，将 LOCAL 设置成 1（允许），AID0 设置成 1（允许），AID1 设置成 0

（拒绝），其他保持默认不变。那么在这种情况下，CorePac Group0 的两个 CorePac 皆可以访问 CorePac0 的 L2 内存，其他 CorePac Group 的 CorePac 访问 CorePac0 的 L2 内存时皆会触发保护异常。

除了 CorePac 内部内存的保护外，Scorpius 平台还必须实现 CorePac 外 L3 和 L4 内存的内存保护。L3 和 L4 内存的保护是通过 CorePac 内部的扩展内存控制器的内存保护单元来实现的。扩展内存控制器内部的内存保护与地址扩展寄存器组（Extended Memory Controller Memory Protection and Address Extension Registers，简称 MPAXR）来负责针对 L3 和 L4 内存访问的保护与地址扩展工作。MPAXR 使用内存段的概念来进行内存保护和地址扩展，整个 L3 和 L4 地址空间由 16 组 MPAXR 寄存器来表示。每组 MPAXR 寄存器由一对 32 位寄存器组成，这对寄存器被称为 MPAXHx 和 MPAXLx（其中 x 取值为 0 到 15）。图 4-15 和图 4-16 分别给出了 MPAXHx 寄存器和 MPAXLx 寄存器的结构。

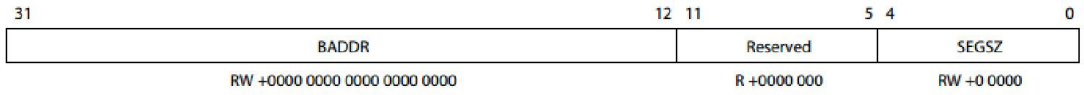

图4-15 MPAXHx 寄存器

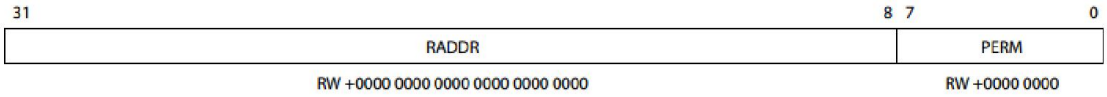

图4-16 MPAXLx 寄存器

这对32位寄存器通过定义了四个重要字段BADDR，RADDR，SEGSZ和PERM来表示一个内存段。BADDR 用来表示 CorePac 内部原生 32 位地址的高 20 位，RADDR 用来表示映射逻辑 36 位扩展地址的 24 位，SEGSZ 用来表示该内存段的段大小，PERM 用来表示本 CorePac 对该内存段的访问属性。BADDR 和 RADDR 的段内存必须是 4K 字节对齐的，也就是 TMS320C6678 的 CorePac 中将 4K 定义为一个段内存最小单位。SEGSZ 可以取 0，4KB，8KB 一直到 4GB 的大小，如果 SEGSZ 取 0 那么当前段是禁止访问的。PERM 字段的含义类似上文 L2 内存保护页属性寄存器的低 6 位，规定了用户或管理员对于该内存段的访问规则——可读，可写以及可执行。需要注意的是，这些访问规则仅对当前 CorePac 有效，其他的 CorePac 是无法看到当前 CorePac 的内存段配置的。

在 Scorpius 平台中，CorePac 组群将会在初始化的时候初始化当前 L2，L3 和 L4 内存的访问规则，Scorpius 平台不使用 36 位扩展地址空间，仅采取一对一线性

映射的平坦 32 位地址空间。Scorpius 通过内存区域（Memory Area）的概念来管理这个线性的 32 位内存区域。

4.4.2.2 线性内存区域管理

上一节中讨论了 Scorpius 平台针对 TMS320C6678 内存保护的底层实现原理。上一节中可以发现 L2 和 L3，L4 内存的保护原理具有一定的区别。Scorpius 平台为了向上层隔离这个区别并且使 L2，L3 以及 L4 的管理更加统一和紧密，Scorpius 单独设计一个线性内存区域管理模块，该模块通过内存区域（Mermoy Area）的概念来统一管理 L2，L3 和 L4 内存段和 CAS 硬件资源的内存访问保护。需要特别注意的是除了保护 L2，L3 和 L4 内存段以外还必须保护 CAS 硬件资源，这是因为在 4.3 小节中已经讨论过不同 CorePac 群组间是不允许访问互相的 CAS 硬件资源，如果不对 CAS 资源加以保护，黑客可以通过恶意访问其他 CorePac 的 CAS 硬件资源的方式来使系统死锁。

Scorpius 平台使用 mem_area 数据结构来表示一个内存段，一个内存段是一片线性内存区域的抽象，用于管理该区域的访问属性。下面给出 mem_area 数据结构的主要成员：

```
struct mem_area {
    unsigned long area_start;   // 内存段开始地址
    unsigned long area_size;   // 内存段大小
    unsigned char perm;   // 内存段访问权限
    union {
        unsigned char aid;   // L2 内存使用的 AID
        unsigned char seg_id;   // L3 或 L4 内存使用的段 ID
    };   /* 匿名 union */
    unsigned char corepac_indx;   // 所属 CorePac 编号
    unsigned char mem_area_state;   // 当前内存段状态
    struct page *page;   // 对应伙伴系统中的页
    struct rb_node rbn;   // 所有内存区域以 CorePac 结构中的 rb_root 为根组成的红黑树
};
```

在 4.2.1 中介绍了 corepac 数据结构，其中有一个成员是 struct rb_root 类型的 mem_area_root，该成员是当前 CorePac 所有线性内存段组成红黑树的根。红黑树是一种常用高级数据结构，它是许多自平衡查找树的一种，它能保证在最坏情况下，基本动态集合操作的时间是 O（lgn）。红黑树是一种自平衡二叉树，在每个

结点上增加一个储存结点颜色的位，每个结点可以是 RED 或 BLACK。通过对任何一条从根到叶子的路径上各个结点着色方式的限制，红黑树确保没有一条路径会比其他路径长出两倍，因而是接近平衡的[33]。使用红黑树来记录内存段允许系统能很快速的对内存段的修改，查询，删除等操作，内存段在红黑树中的键值为内存段的起始内存地址。

4.5 基于 OpenBinder 方式的扩展核间通信

上一节讨论了 Scorpius 平台中如何通过内存保护的方法将多个 CorePac 组群的内存空间隔离出来。但是 Scorpius 平台中的多个操作系统和多个 CorePac 之间并不应该是完全独立，他们必须通过一种系统可控制监视的方式来进行互相交流。Scorpius 平台实现三个模块的通信方式来保证上述交流，这三个模块分别是 CorePac 间的核间通信，CorePac 组群间的跨组通信，核间与组群间的邮箱服务。这些模块虽然功能各异但是实现原理都是基于底层核间中断进行的。

Scorpius 平台采用 OpenBinder 扩展的方式来实现前两个层次，其主要依靠的是位于 L3 内存的 SBinder 虚拟设备，该设备将负责各个 CorePac 和各个 CorePac 组群间信息的交换。OpenBinder 是一种进程间通信框架，该框架最早由 Be 和 Palm 公司开发设计，目前最热门的移动设备操作系统 Android 的进程间通信使用的就是 OpenBinder 框架[34]。OpenBinder 允许进程将其支持的操作以接口函数的形式展现出来，所有这些操作被定义成了两个部分——服务端和客户端。当某个进程要进行远程跨进程函数调用的时候，该进程将根据目标进程的接口协议，调用本地客户端函数，本地客户端函数会将传输数据封装成数据包，并将数据包传递给 Binder 虚拟驱动设备，虚拟驱动设备会根据初始化时目标进程注册的信息将数据包传递给目标进程服务端，最后目标进程解开数据包执行远程调用，并将结果通过异步或同步的方式返还给客户端。这样做的好处是可以将进程间通信的具体实现方式和用户业务逻辑分离开来，用户只需要关心其逻辑代码的实现而不需要关心如何进行进程间通信。在用户代码的角度来看，一个本来在远程进程上执行的代码仿佛就像在本进程上执行一样，这种设计大幅度简化了用户代码的编码复杂性。

Scorpius 平台提供的核间通信对于用户来说只能在框架范围内进行添加，这个框架是独立于用户操作系统的。

4.5.1 核间中断

本节将主要讨论 Scorpius 平台中核间通信的底层实现原理。在 TMS320C6678 中，核间通信是通过核间中断的方式实现的，如果某个 CorePac 想将某个消息告诉

给另外一个 CorePac，该 CorePac 将触发目标 CorePac 的核间中断，在触发中断的同时将消息告诉给目标 CorePac。TMS320C6678 通过 IPCx 发起寄存器和 IPCx 应答寄存器（x 从 0 取到 7）来实现核间中断。图 4-17 和图 4-18 分别给出了 IPCx 发起寄存器和 IPCx 应答寄存器的构造。

31	30	29	28	27 – 8	7	6	5	4	3 – 1	0
SRCS27	SRCS26	SRCS25	SRCS24	SRCS23 – SRCS4	SRCS3	SRCS2	SRCS1	SRCS0	Reserved	IPCG
RW +0	RW +0	RW +0	RW +0	RW +0 (per bit field)	RW +0	RW +0	RW +0	RW +0	R, +000	RW +0

Legend: R = Read only; RW = Read/Write; -*n* = value after reset

图4-17 IPCx 发起寄存器

31	30	29	28	27 – 8	7	6	5	4	3 – 0
SRCC27	SRCC26	SRCC25	SRCC24	SRCC23 – SRCC4	SRCC3	SRCC2	SRCC1	SRCC0	Reserved
RW +0	RW +0	RW +0	RW +0	RW +0 (per bit field)	RW +0	RW +0	RW +0	RW +0	R, +0000

Legend: R = Read only; RW = Read/Write; -*n* = value after reset

图4-18 IPCx 应答寄存器

IPCx 发起寄存器和 IPCx 应答寄存器的高 28 位完全相同，这些位允许核间中断传递一些有用的信息，该 28 位在后文中称为中断信息位。IPCx 发起寄存器的第 0 位 IPCG 用于触发核间中断。IPCx 发起寄存器和 IPC 应答寄存器一共有 8 组，每个 CorePac 对应一组。如果当前代码运行在 CorePac0 上，并且 CorePac0 想要给 CorePac1 发起一次核间中断，那么 CorePac0 的代码应该将相关核间中断消息写到 IPC1 发起寄存器的高 28 位，然后将 IPC1 发起寄存器的第 0 位置成 1。此时如果 CorePac 中断控制器和 CIC 片上中断控制器的设置正确，那么一个核间中断将立即传达到 CorePac1 中。CorePac1 将在其 IPC1 应答寄存器中读到来自 CorePac0 的信息，读完信息后 CorePac1 先写 IPC 应答寄存器来清除核间中断信息，然后再应答 CorePac 中断控制器和 CIC 片上中断控制器的相关中断事件应答寄存器。

4.5.2 CorePac 间通信与 CorePac 组群间通信

上一小节简单讨论了 TMS320C6678 的 CorePac 核间中断的底层实现，这一小节将在上一小节的基础上实现 CorePac 间通信与 CorePac 组群间的通信。

在层次上，CorePac 间通信与 CorePac 组群间通信分为三个层次，底层为 SBinder 驱动层，该层次用来管理 CorePac 和 CorePac 组群在 SBinder 设备中的注册，查询，信息保存以及核间中断的实现。SBinder 驱动层之上为桩函数层，Scorpius 平台中每一个需要实现跨核跨组群功能的函数都会对应两个桩函数，这两个函数一个位于客户端，一个位于服务端。位于客户端的桩函数负责对上层传递下来的跨核跨组群调用请求进行构建 RPC（Remote Processor Communication）数据包，

然后将 RPC 数据包传递到 SBinder 驱动设备中。位于服务端的桩函数负责从 SBinder 驱动设备中读取 RPC 数据包，然后将其解包成具体调用数据然后调用上层服务端函数。在桩函数层以上就是跨核跨组群函数调用层，该层同样有客户端和服务端两个函数，其中客户端函数通过对调用请求进行处理判断出本次请求是否需要进行跨核跨组群函数调用，而服务端则是直接进行用户需要的业务逻辑操作。

图 4-19 描述了 Scorpius 平台为了实现跨核跨组群函数调用的层次关系。一个跨核跨组群函数，用户需要编写一个 OIDL 接口，一个客户端函数，一个服务端函数，一个客户端桩函数，一个服务端桩函数以及将该 OIDL 接口函数注册进 SBinder 驱动。实际上客户端桩函数，服务端桩函数的模式是有规则可循的，在 Scorpius 平台中为了方便用户编写跨核跨组群函数，设计了 OIDL 接口描述语言，该语言通过接口描述脚本的方式为用户直接生成客户端和服务端的桩函数代码，具体操作方式将在 4.5.3 小节中进行讨论。本小节会集中讨论 SBinder 驱动层，桩函数层。

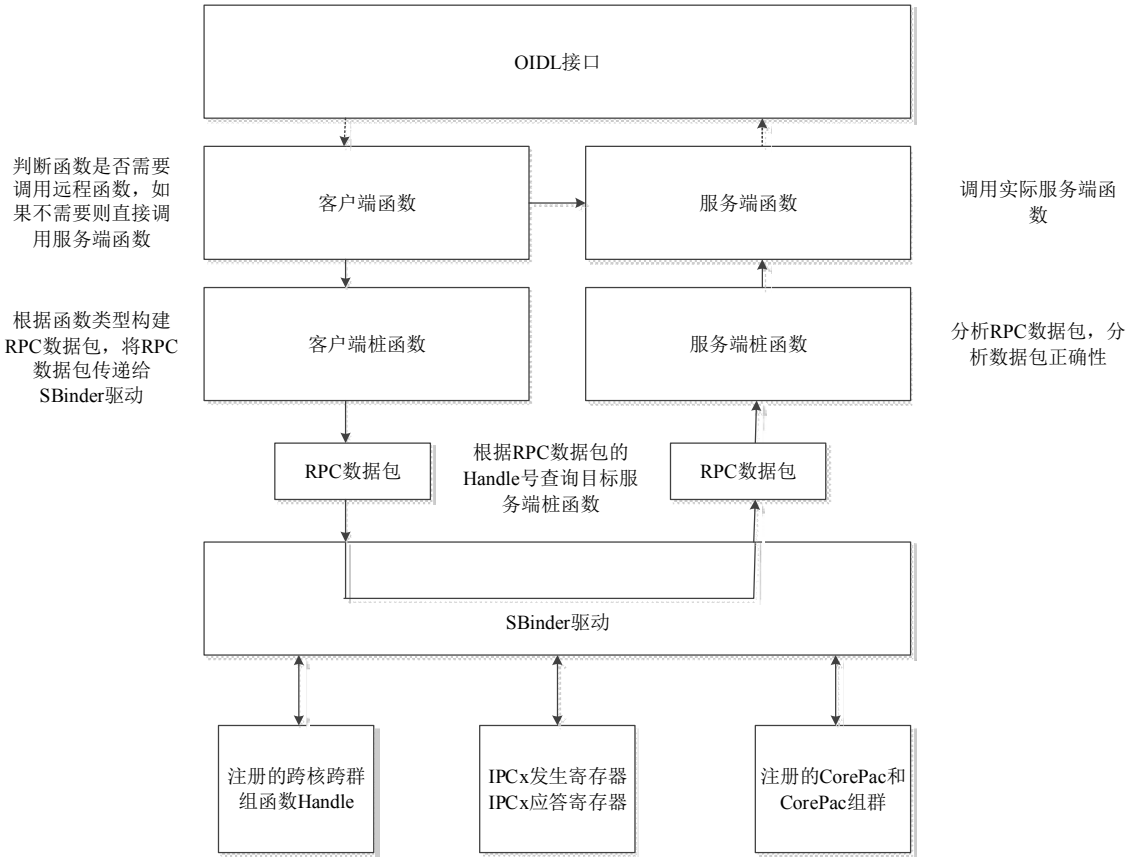

图4-19 各层次调用流与关系

为了实现核间通信与组群间通信，为了区别每个 CorePac，Scorpius 平台定义了一种核间中断发起人逻辑编号，该逻辑编号占用 28 位中断信息位中的 7 位，其中低三位（0 位到 2 位）表示 CorePac 编号，3 位到 5 位表示 CorePac 组群编号，

第 6 位用来标识本次通信是 CorePac 间通信还是 CorePac 组群间通信。图 4-20 给出了该逻辑地址的具体描述情况。

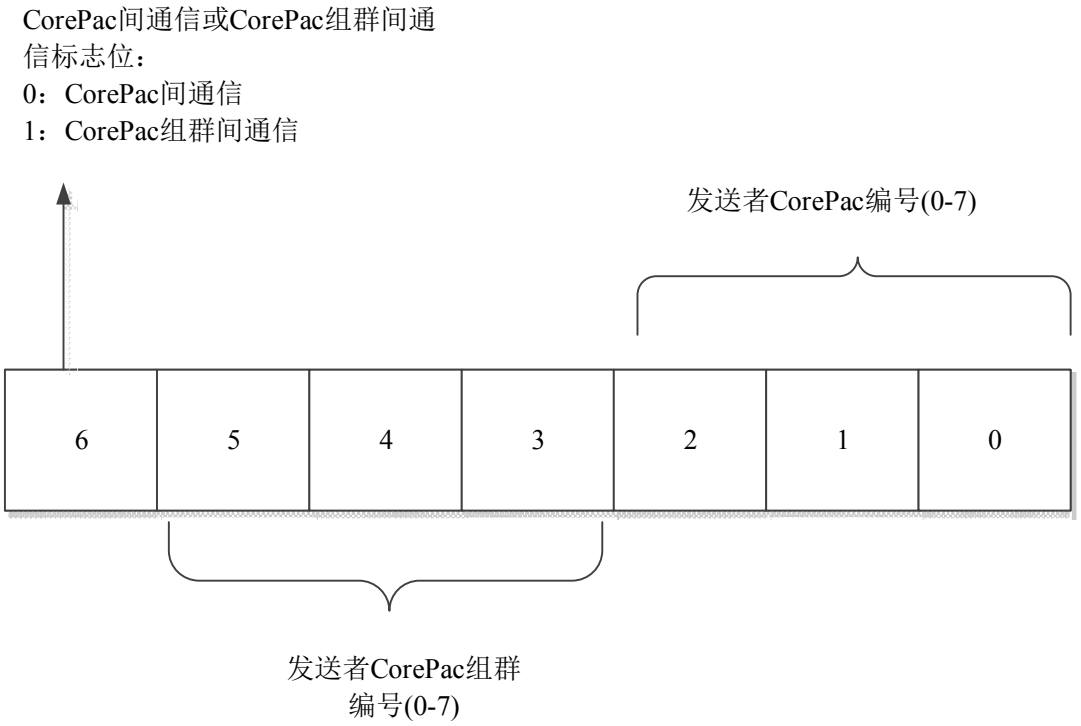

图4-20 逻辑编号

IPCx 发生寄存器和 IPCx 应答寄存器的逻辑编号的高 21 位用于记录跨核跨组群通信所调用函数的句柄 handle，句柄的具体使用方法后文将给出。

Scorpius 平台将跨核跨组群通信的数据封装成 RPC 数据包，RPC 数据包是一种包含了调用者，被调用者，调用函数以及相关参数等数据的字节流，每次跨核跨组群通信必须包括至少一次的 RPC 数据包传递。Scorpius 平台中使用 ipc_rpc_data 数据结构来描述一个 RPC 数据包，该数据结构重要成员如下：

```
unsigned char src_cg;   // rpc 数据来源 CorePac 组群
unsigned char dst_cg;   // rpc 数据目的 CorePac 组群
unsigned int handle;   // 调用函数对应的标识 handle，每一个跨核跨组群函数都拥有一个这样的标识，该标识在系统中唯一确定一个跨核跨组群函数。
unsigned int flag;   // 目前用来标识该次 RPC 数据包是否来自跨组群函数
struct parcel rpc_data;   // parcel 是一种字节流类型的数据包，parcel 提供了一系列操作这些数据包的方法。
unsigned long reply;   // 返回值，当返回值为简单类型时，被远程调用的核心不需要重新
```

申请 RPC 数据包而是直接重复利用当前数据包，提高缓存性能。

struct list_head handle_slot; // SBinder 通过一棵由 handle 为键，function_node 数据结构为数值的红黑树将所有跨核跨组群函数组织起来，其中一个 RPC 数据包的 handle_slot 则是挂载在 function_node 上的队列的结点。

RPC 数据包使用的内存是通过 SBinder 使用 L3 内存中的 SSLAB 分配器进行分配的，这样做的好处是可以做到一次跨核跨组群传输的零拷贝，这部分 L3 内容可以在多个 CorePac 组群中共享，多个 CorePac 可以直接操作 RPC 数据包而不需要将其拷贝到自身的线性内存空间中。

Scorpius 平台使用一个 32 位整形数句柄（handle）来索引一个跨核跨组群函数，所有的跨核跨组群函数在系统初始化的时候通过 register_function_node 函数注册到 SBinder 的红黑树当中，该红黑树使用跨核跨组群函数的句柄（handle）为键。一个跨核跨组群函数在 SBinder 中使用 function_node 数据结构表示，重要成员如下代码所示：

```
unsigned int handle;  // 跨核跨组群函数的句柄
IPC_FUNCTION __numa client_stub_function_pointer;  // 客户端桩函数地址，注意__numa 的意思是标明该地址所处内存地址在较远结点当中。
IPC_FUNCTION __numa server_stub_function_pointer;  // 服务端桩函数地址
struct list_head rpc_node;  // rpc_node 的链表，当同时有多个 CorePac 或 CorePac 组群之间多次调用相同句柄的函数时，所有通信的 RPC 数据包都将在该队列上排队
struct rb_node rbn;  // 所有跨核跨组群函数通过该成员挂接在 SBinder 设备的红黑树上。
struct spinlock lock;  // 自旋锁，用来在多个 CorePac 或 CorePac 组群同时操作时进行保护
void *private;  // 私有数据
```

SBinder 内部整个模型可以用图 4-21 表示。

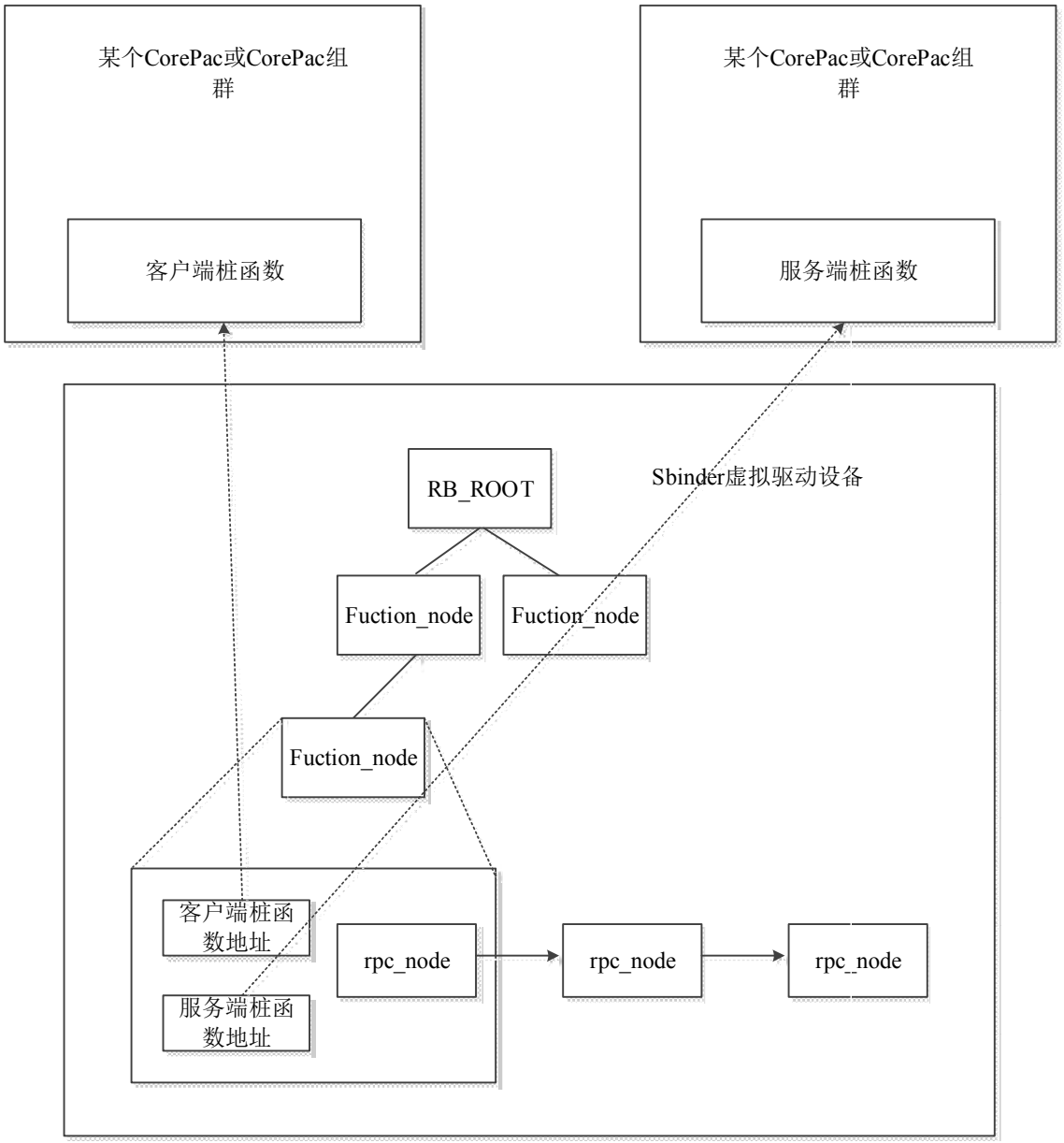

图4-21 SBinder 虚拟设备内部结构示意图

Scorpius 平台规定同 CorePac 组群的 CorePac 间可以进行跨核通信，而不同 CorePac 组群的任意两个 CorePac 间不允许进行任意的核间通信，跨 CorePac 组群的通信必须在本地 CorePac 组群的启动处理器（Bootstrap Processor，简称 BP）与目的 CorePac 组群的启动处理器两者之间进行。SBinder 将拒绝所有不合法的跨核跨组群函数调用行为和 RPC 数据包。下面以一个虚拟的函数 foo 来描述 Scorpius 平台中对于一次跨核跨组群函数的调用过程，该过程是 CorePac 组群 0 上的启动处理器 CorePac0 调用跨核跨组群函数 foo，该函数的实际调用发生在目标 CorePac 组群 1 的启动处理器 CorePac3 上：

这个虚构的 foo 函数的函数原型为 void foo(int a)，该函数在 CorePac 组群 0 的操作系统 A 中被调用。该函数首先根据当前操作系统的上下文，判断该函数是否能在本地被执行。如果可以，则 foo 函数直接调用服务端的处理函数__server_foo 进行处理并将结果返回给 CorePac 组群 0 的操作系统 A。如果 foo 函数判断不能在本地被执行，则调用 foo 函数在桩函数层的客户端代理桩函数 foo_stub_proxy，该函数会首先检查一些边界情况，然后申请并打包 RPC 数据，将调用发起核，调用目标核，调用函数句柄 handle 和其他数据填充到 RPC 数据的头部中，然后将函数的参数打包成 parcel 字节流保存在 RPC 数据包中。在完成上述工作后，客户端代理桩函数会通过函数句柄 handle 查询位于 SBinder 红黑树中的跨核跨组群函数结点，找到后将 RPC 插入该跨核跨组群函数结点的队列中，最后调用 SBinder 驱动的处理函数。SBinder 驱动会根据函数句柄 handle 找到跨核跨组群函数结点并再次确认数据包的正确性，然后将调用底层函数设置 IPCx 发生寄存器中断信息的高 21 位为跨核跨组群函数的句柄，中断信息位的低 6 位设置成逻辑编号，最后将 IPCx 发生寄存器的第 0 位置 1 触发目标 CorePac2 的核间中断。目标 CorePac3 被中断后将检查 IPCx 应答寄存器的中断信息位的逻辑编号，如果逻辑编号没有错误则提取中断信息位中的跨核跨组群函数的句柄，然后调用 SBinder 驱动的处理函数。SBinder 将根据传入的函数句柄索引红黑树上的跨核跨组群函数结点，并将对应的 RPC 数据包取下调用服务端桩函数__foo_stub。服务端桩函数将检查 RPC 数据包的合法性，并提取出所有的参数，然后调用服务端函数，最后服务端函数会进行实际的逻辑处理，将逻辑处理结果上传到当前 CorePac3 所在 CorePac 组群 1 的操作系统 B 上。流程图 4-22 简单描述了上述过程。

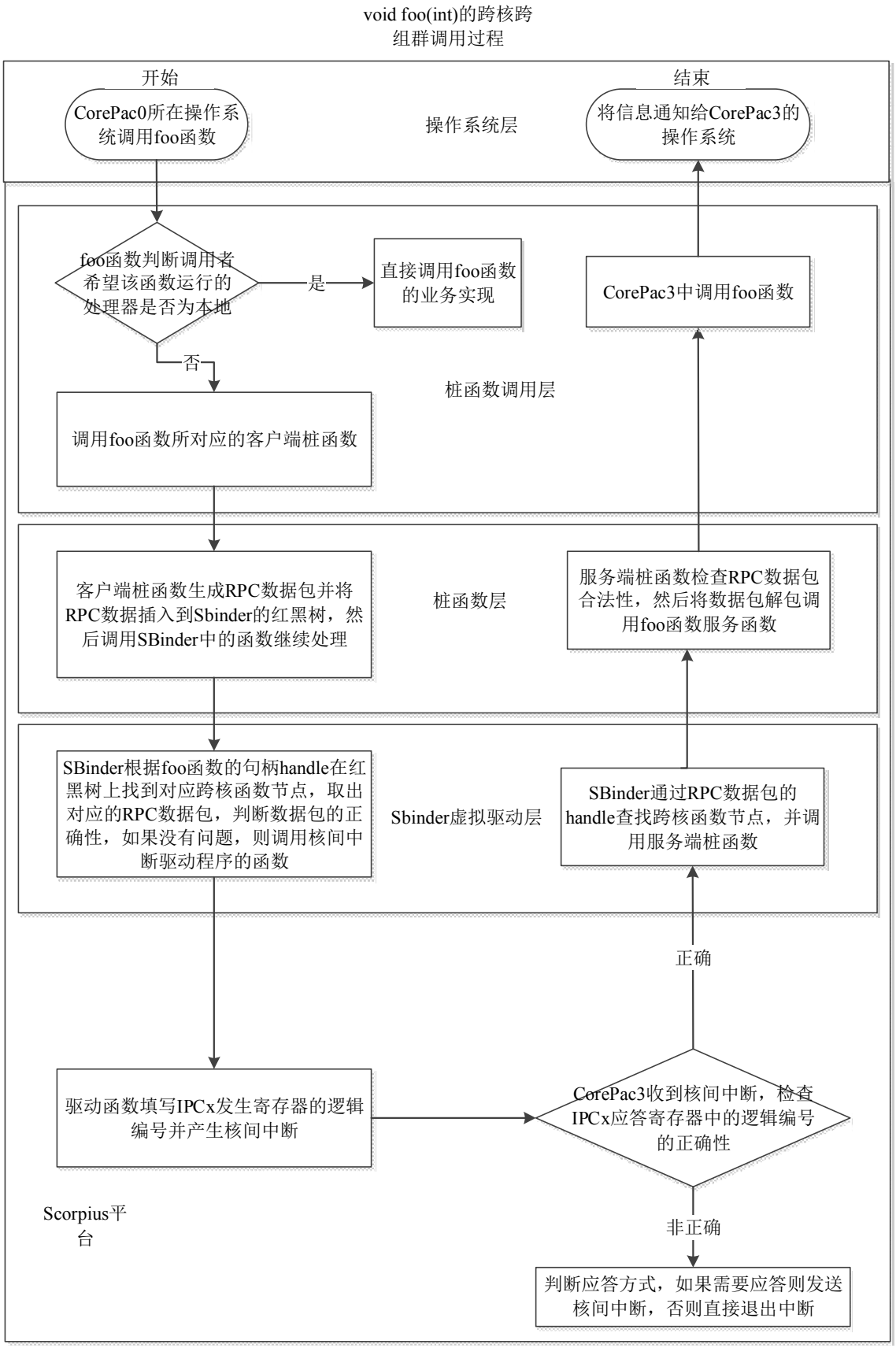

图4-22 foo 函数的跨核跨组群调用流程

上述过程中需要注意的是 CorePac3 在收到核间中断后的处理一直位于中断上下文当中，CorePac3 所在 CorePac 组群运行的操作系统 B 应该单独设计一个任务来处理桩函数层的内容。CorePac3 在处理完 SBinder 虚拟驱动层后应该将 RPC 数据包和相关调用函数插入一个工作队列然后退出中断再处理。下一小节将讨论在 Scorpius 平台上是如何利用 OIDL 接口描述语言自动生成桩函数。

4.5.3 使用 OIDL 接口描述语言自动生成代码

上一小节中讨论了 Scorpius 的跨核跨组群函数调用的具体设计与实现细节。本小节将讨论使用 OIDL 接口描述语言工具来自动生成客户端和服务端的桩函数代码。OIDL 是 OpenBinder Interface Description Language 即 OpenBinder 接口描述语言的缩写，该语言使用 C 语言的函数原型的方式来描述一个接口函数。OIDL 工具是一个 Python 脚本语言编写的简单自动代码生成器，该工具将扫描 OIDL 脚本然后根据用户定义的接口自动生成对应客户端和服务端的桩函数实现代码。Scorpius 平台设计该工具的目的在于简化跨核跨组群函数的编写，上层操作系统用户或者需要扩展 Scorpius 平台的系统软件开发者使用该工具可以将时间集中在上层业务逻辑的编写上而不需要关心跨核跨组群函数调用的实现细节。

图4-23 OIDL 接口描述语言相关工具

如图 4-23 所示，OIDL 工具由 oidl.py，funcs.py，setup.py，template.py 四个 python 脚本文件通过 py2exe 工具编译转换成 Windows 下的 EXE 可执行程序。OIDL 接口描述语言的语法同 C 语言的函数原型。接口中所有类型必须是基本类型，基本类型指针。指针类型必须是基本指针类型且不能是指针的指针，如果需要使用用户自定义类型，需要使用 void 空指针，然后在代码业务逻辑部分进行指针转换。接口中不能出现 typedef 的结构体，以及 define 的其他宏替换。如果需要传递数组，

必须使用指针的方式。接口中的参数以及返回值不能包括函数指针，参数不能省略形参名。下面以上小节的 foo 函数为例给出该函数在 OIDL 接口描述语言中的书写例子：

```
#ifndef   __FOO_TEST_H
#define   __FOO_TEST_H
void foo (int num);
#endif
```

在编写完 OIDL 接口描述脚本后，使用 oidl.exe 转换工具即可生成相对应的桩函数代码，用户只需要自行将生成的桩函数代码拷贝到工程当中。该工具仅仅生成了服务端和客户端的两个桩函数，用户还必须实现服务端的实际业务逻辑同名函数和客户端的判断代理函数，除此之外，用户必须确定该跨核跨组群函数的句柄 handle 号，并且在系统初始化时将该句柄 handle 号记录到 Scorpius 平台 SBinder 驱动的红黑树中。

4.5.4 核间邮箱与组群间邮箱

Scorpius 平台基于 OpenBinder 方式的跨核跨组群函数调用是一种比较重的核间通信方式，该方式虽然简化了用户的逻辑代码，让用户对函数的调用不再关心该函数的实际运行是在本 CorePac 上还是在远程 CorePac 上，但是该方法不适合简单数据的传输。如果仅仅系统只需要相互传递一个字节的数据，那么使用传统的邮箱设计将会更高效。为了满足这种需求，Scorpius 平台设计并实现了简单的核间邮箱与组群间邮箱机制，这两个模块同样是基于核间中断的方式实现的，只不过在上层路径上简化了很多。发起者只需要将数据存放在 L3 内存的共享区的某个位置，然后发起核间中断通知接收者，接收者收到核间中断后，通过 IPCx 应答寄存器中的信息，检索 L3 内存共享区，即可快速获得信息。

4.6 上层操作系统支持模块

Scorpius 平台关键目的就是为上层操作系统提供一个硬件隔离的运行环境。Scorpius 平台的上层操作系统支持模块就是将硬件资源的功能进行抽象，然后直接以接口函数的方式提供给上层操作系统使用。上层操作系统支持模块主要包括加载运行，时钟滴答，中断处理和内存管理四个部分，这四个部分基本都是利用 Scorpius 平台底层提供的功能来实现上层操作系统支持的。

4.6.1 加载运行

Scorpius 平台目前只支持静态加载的方式，用户需要在配置文件中定义好需要支持的操作系统类型，并且设置好该操作系统所运行在哪一个 CorePac 组群上，配置好该操作系统代码入口点和内存布局形式。Scorpius 平台在完成 Bootloader 的初始化后将扫描所有的操作系统配置，根据这些配置的情况为每个操作系统分配硬件资源，在一切初始化完成后就跳转到操作系统配置好的入口地址开始运行用户操作系统。

Scorpius 平台使用 os_cfg 数据结构来表示一个操作系统配置，该数据结构提供给 Scorpius 以下信息：所运行操作系统的 CorePac 组群，操作系统支持多核或单核，该操作系统代码的入口地址，该操作系统的内存空间布局，该操作系统的定时器。一个 os_cfg 和一个 CorePac 组群是一一对应的，一个 CorePac 组群上只能运行一个操作系统，而一个 CorePac 组群则至少由一个 CorePac 组成。下面流程图 4-24 描述了 Scorpius 平台上某个用户操作系统的加载运行过程。

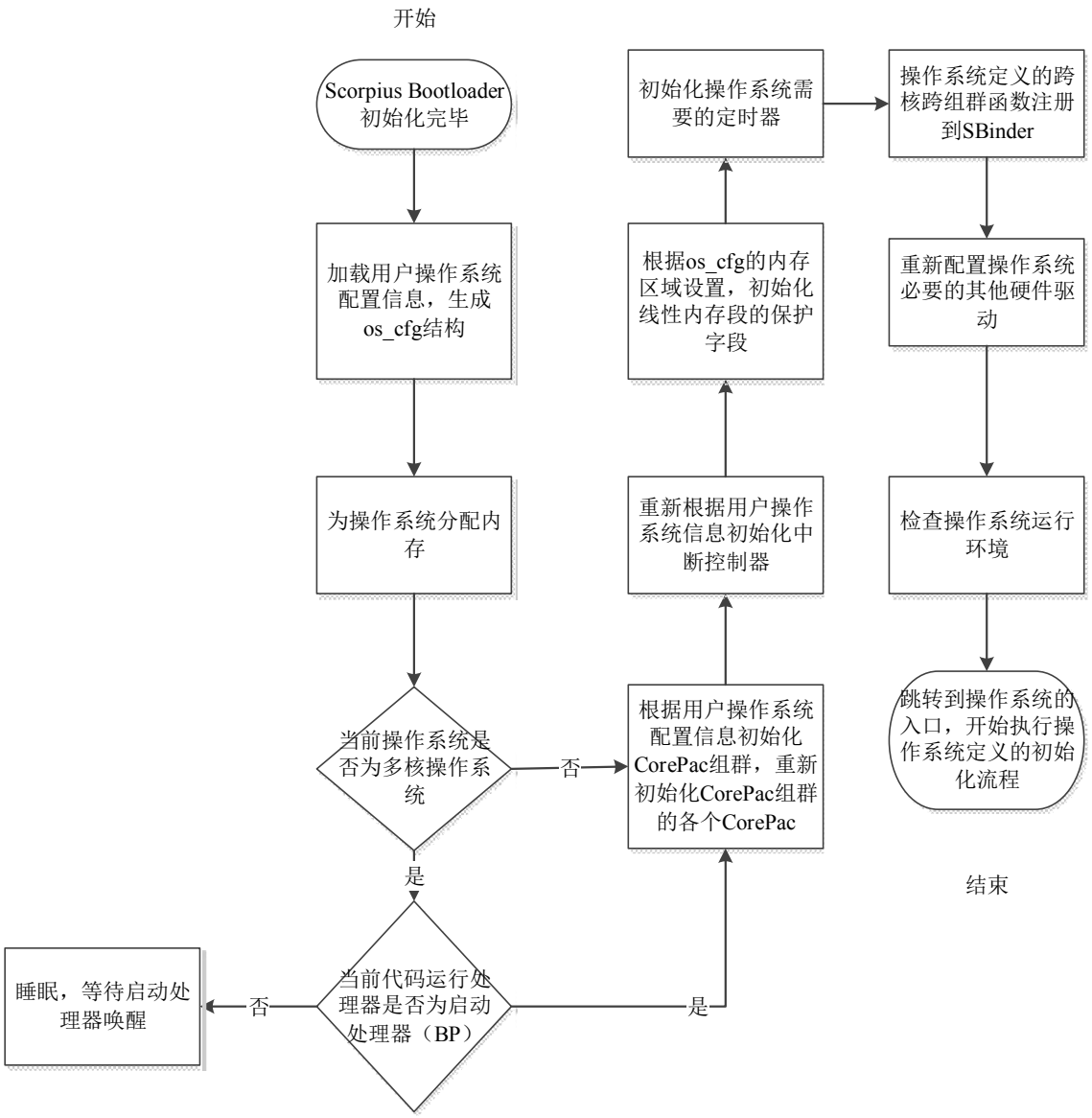

图4-24 用户操作系统加载主要过程

4.6.2 时钟滴答

时钟滴答是现代分时操作系统必不可少的重要组件，时钟滴答的底层依赖于系统硬件定时器所提供的时钟中断。由于 Scorpius 平台上同时运行了多个操作系统，不同操作系统之间采用的时钟中断频率不同，所以不同操作系统需要的定时器配置不同。操作系统和 CorePac 组群是一一对应的，每个 CorePac 组群的时钟中断由启动处理器提供，操作系统的时钟配置就是针对该操作系统所在 CorePac 组群的启动处理器的对应定时器进行配置。

Scorpius 平台使用 os_timer_cfg 数据结构来配置操作系统的定时器，该数据结构提供了以下关键信息：本操作系统的时钟中断理想频率，本操作系统所在

CorePac 组群的启动处理器所对应的硬件定时器驱动，本操作系统的时钟中断对应 DSP 内核中断号。

Scorpius 平台在用户操作系统的初始化时加载时钟配置信息，然后根据配置信息填写 os_timer_cfg 数据结构，根据 os_timer_cfg 数据结构初始化相关 CorePac 的定时器驱动。

4.6.3 中断处理

在 Scorpius 平台中，不同操作系统间的中断配置也是不同的。在每个操作系统所对应的 CorePac 组群中，只有启动处理器才负责对该 CorePac 组群中所有中断的处理。在操作系统的初始化过程中，CorePac 组群的启动处理器会通过 CorePac 中断控制器以及 CIC 片上中断控制器来配置当前 DSP 内核所需要相应的中断。在由于操作系统中需要管理的中断数目远远大于 DSP 支持的中断数目，所以操作系统中的中断号实际上是 CorePac 中断控制器的 128 个系统事件的事件号。当某个中断发生后，操作系统会依靠 Scorpius 平台提供的接口查询 CorePac 中断控制器中的具体外部事件事件号，然后根据外部事件事件号通过 Scorpius 平台中断映射表映射得到操作系统中断号，最后调用该中断号对应的中断处理程序。图 4-25 描述了当一个外部事件发生时，位于某个 CorePac 组群上的操作系统是如何响应该事件的。

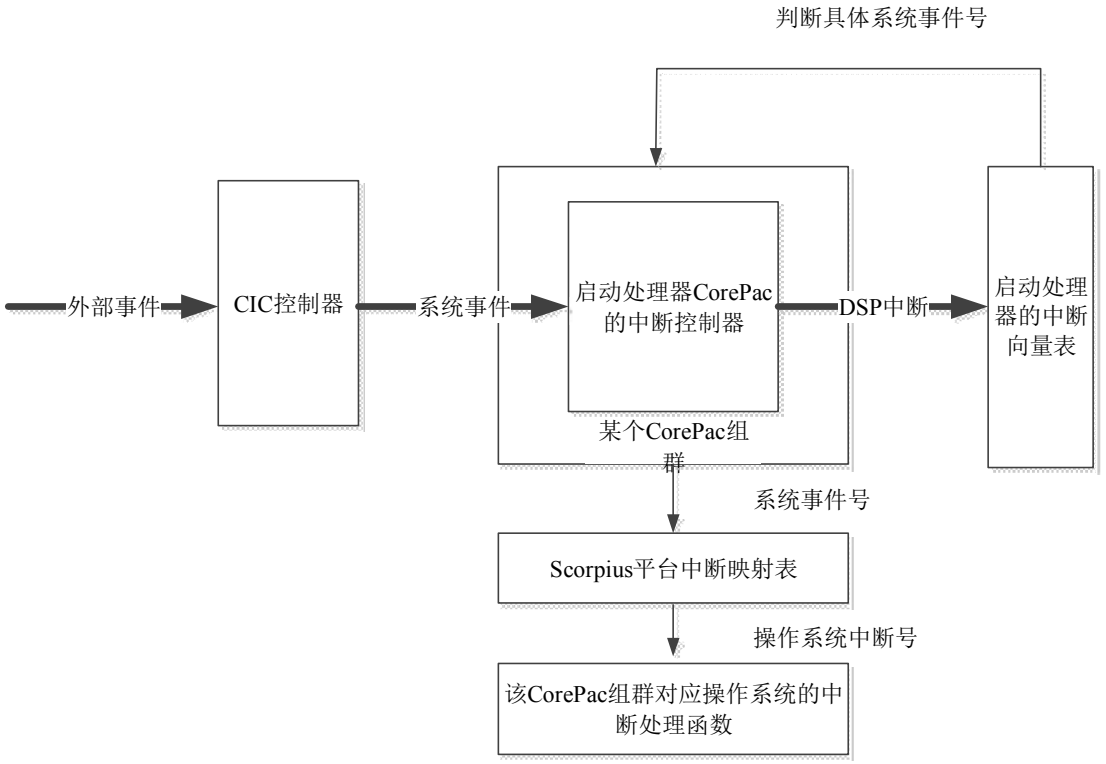

图4-25 用户操作系统的中断处理过程

4.6.4 内存管理

Scorpius 平台并不负责用户操作系统的动态内存管理，不同的用户操作系统的内存管理方式将根据自身的内存管理算法的不同而不同。Scorpius 仅仅根据用户操作系统在加载运行时提供的操作系统配置 os_cfg 中的内存区域描述成员来为用户操作系统分配一大片内存空间。用户操作系统将在该大片内存空间上实现自己的内存管理算法。

4.7 本章小结

本章重点针对多核多操作系统板级支持平台 Scorpius 的各个模块的实现进行讨论，从底层到上层给出了 Scorpius 平台的各个组成部分的实现原理和设计细节。首先，介绍了基于 TMS320C6678 的硬件驱动实现原理和方式，然后大篇幅介绍了多核基础支持模块，多核同步模块，内存管理模块和独创的基于 OpenBinder 方式的扩展核间通信模块。本章的末尾大致介绍了 Scorpius 平台上层操作系统支持模块的设计与实现方式，该部分内容还将在下一章中继续涉及到。下一章将以 deCORE MOS 和 uCosII 为例，简单介绍如何将用户操作系统应用到 Scorpius 平台上。

第五章 多核多操作系统板级支持平台 Scorpius 实例应用

5.1 操作系统的平台移植

上一章讨论了多核多操作系统板级支持平台 Scorpius 关键模块的设计与实现方式，本章将围绕如何移植用户操作系统，如何应用 Scorpius 平台进行讨论。本章的主要工作是移植两款不同的用户级嵌入式实时操作系统到 Scorpius 平台，这两款用户操作系统一个是用于汽车电子机械控制行业满足 AUTOSAR 标准的多核实时操作系统 deCORE MOS，另一个是用于实时领域的经典嵌入式单核操作系统 uCOSII。本章最后将用两个例子展示在 Scorpius 平台同时运行多核操作系统 deCORE MOS 和单核操作系统 uCOSII 以及同时运行 2 个 uCOSII 操作系统。

5.1.1 deCORE MOS 的移植

deCORE MOS 是面向控制领域，符合 OSEK/VDK 标准，AUTOSAR OS 架构的由电子科技大学研发的嵌入式实时多核操作系统，包含 OSEK/VDX 标准及 AUTOSAR OS 架构的任务管理，中断管理，事件管理，资源管理，报警管理，计数器管理，内部消息通信管理，错误处理等基本功能，及支持多核的自旋锁管理，多核系统启动/关闭，核间通信，核间中断管理等扩展功能，并为方便应用的开发提供了调试管理的功能。提供标准的应用编程接口（API），满足面向控制领域的应用软件开发的需要。

该操作系统已经拥有一个开源的基于 TMS320C6678 硬件的移植版本，在硬件兼容性上完全满足 Scorpius 平台的基础要求，只需要修改原来移植版本的启动代码就可以直接将 deCORE MOS 应用到 Scorpius 平台。

5.1.2 uCOSII 的移植

uCOSII（又称 μCOSII）是第二代微控制操作系统（Micro Control Operating System II）的简称，是一个可以基于 ROM 运行的，可裁剪的，抢占式，实时多任务内核，具有高度可移植性，特别适合于微处理器和控制器，是和很多商业操作系统性能相当的嵌入式单核实时操作系统[35]。由于 uCOSII 仅支持单核操作系统，在 Scorpius 平台上将运行在单 CorePac 组成的 CorePac 组群上。由于 uCOSII 具有很高的可移植性，所以针对 Scorpius 平台的移植工作比较简单，主要分为两个步骤，第一个是匹配硬件，即将 Scorpius 操作系统支持模块的时钟滴答功能的 API 直接提供给 uCOSII 使用，然后在 uCOSII 的启动过程中添加对串口设备的配置过

程。第二个是修改上下文切换时的寄存器保存和恢复的汇编代码，由于 Scorpius 平台不关心上下文切换的具体细节，所以在移植时必须考虑如何保存和恢复当前硬件的寄存器。由于第一个步骤十分简单，仅需要修改 uCOSII 的时钟中断配置函数即可，所以本小节集中讨论如何修改 uCOSII 上下文切换中的寄存器恢复与保护。

在 TMS320C6678 硬件平台上，uCOSII 需要保护的寄存器为 CSR，IRP，B0 到 B15 以及 A0 到 A15。这些寄存器用如下图 5-1 的方式进行保存，需要注意的是 TMS320C6678 的应用二进制接口（Application Binary Interface，简称 ABI）标准规定了其堆栈是采用空递减的方式，这和 ARM 不一样。

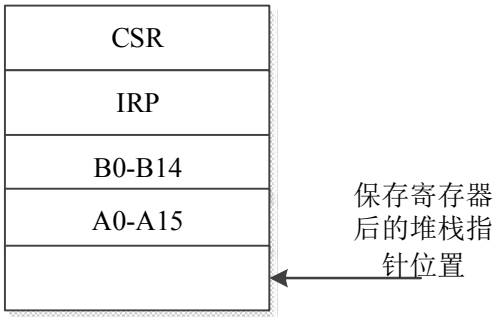

图5-1 TMS320C6678 上下文切换时栈的情况

uCOSII 将保存寄存器后的堆栈指针保存的任务控制块中，在任务恢复运行时 uCOSII 将依照相反的顺序恢复寄存器。为了与 Scorpius 中断时寄存器的恢复保持统一，uCOSII 这里将 IRP 设置在 CSR 的下方，该 IRP 在中断环境下是中断返回地址，在一般上下文切换的环境下则是函数返回地址。在恢复寄存器操作的末尾只需要使用跳转指令即可恢复到程序之前的运行的位置。

5.2 测试实例与运行结果

5.2.1 测试实例设计

为了测试上述移植的正确性以及 Scorpius 平台的正确性，本小节设计了两个十分简单的实例来进行验证。第一个实例使用 deCORE MOS 和 uCOSII 的组合，该实例中配置了两个 CorePac 组群 0 和 1，0 号组群由两个 CorePac 组成，分别是 CorePac0 和 CorePac1，该 CorePac 组群运行 deCORE MOS。1 号组群由一个 CorePac 组成，该 CorePac 组群运行 uCOSII。第二个实例中使用 8 个 uCOSII 的组合，该实例中配置了 2 个 CorePac 组群，每个组群由 1 个 CorePac 组成，用来运行 2 个不同的 uCOSII 操作系统。

5.2.1.1 测试 deCORE MOS+uCOSII 组合

本实例中在 deCORE MOS 上编写了两个简单任务，这两个简单任务分别运行在不同的 CorePac 上，在 uCOSII 上也编写了一个任务。这些任务仅仅将自身的运行信息打印到调试串口。图 5-2 描述了该测试下的 CorePac 组群配置信息。

图5-2 测试 1 的 CorePac 配置信息

5.2.1.2 测试 uCOSII+uCOSII 组合

本实例在每个 uCOSII 上编写了两个任务，这两个任务打印自身的运行信息到调试串口上。

5.2.2 测试结果

图 5-3 给出了测试 1 的运行情况截图。测试结果表明，Scorpius 平台除了可以有效支持多个单核操作系统同时运行。

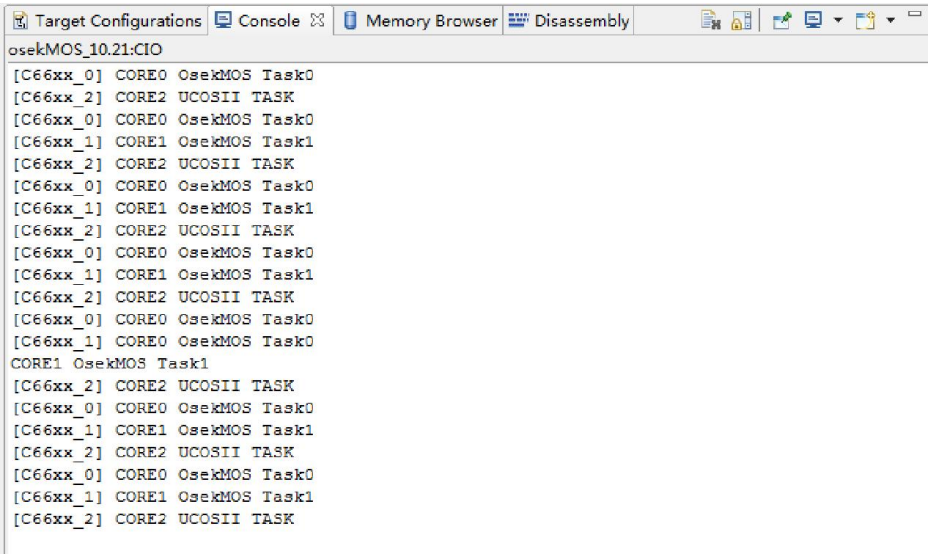

图5-3 测试 1 的运行结果

图 5-4 给出了测试 2 的运行情况截图。测试结果表明 Scorpius 也可以支持多核

操作系统和单核操作系统的混合同时运行。

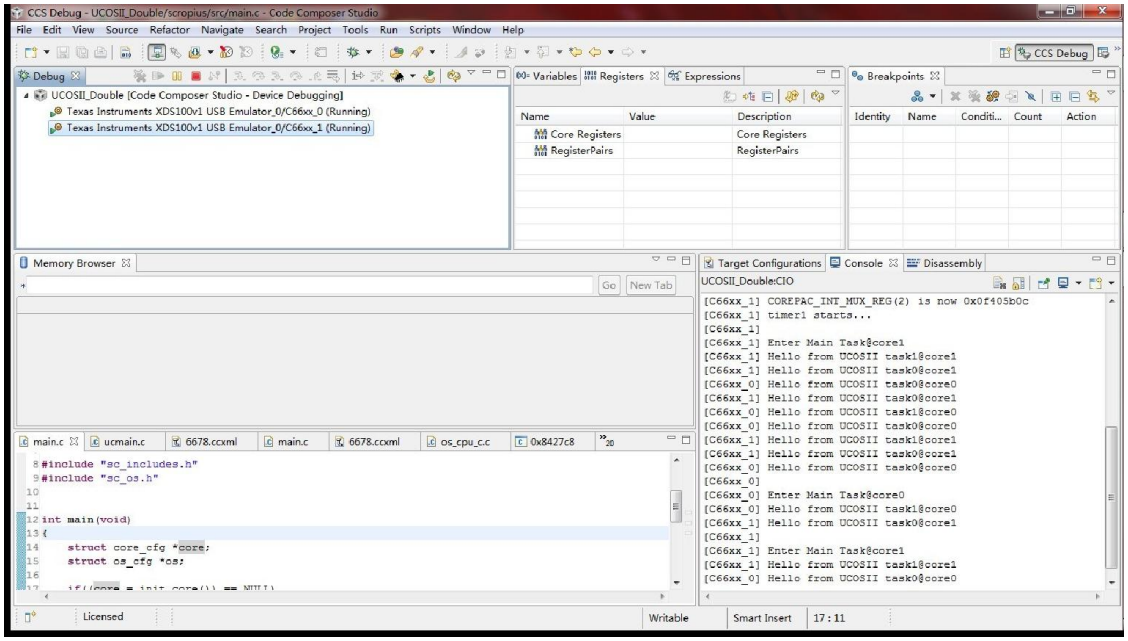

图5-4 测试 2 的运行结果

5.3 本章小结

本章的前半部分简要的描述了两个用户操作系统 deCORE MOS 和 uCOSII 在多核多操作系统板级支持平台 Scorpius 上的移植过程，详细讨论了在 TMS320C6678 硬件平台上的上下文切换的寄存器保存和恢复。本章的后半部分简要介绍了在多核操作系统和单核操作系统混合运行以及多个单核操作系统混合运行的情况下的测试实例以及测试。

第六章 总结与展望

6.1 总结

本文针对多核处理器在嵌入式领域的应用，从硬件，软件以及软硬件结合等三个角度总结了目前多核技术在嵌入式领域的发展，结合 TI 的 KeyStone 平台提供的硬件多核支持，研发了一个支持多核多操作系统的板级平台 Scorpius。本文的主要工作包括了：

1. 从硬件处理器角度分析了多核处理器的发展现状和发展趋势，从软件操作系统的角度描述了目前嵌入式多核操作系统的市场现状，从软硬件结合的角度阐述了三种不同的多核架构。

2. 以 TI 的 KeyStone 平台为硬件基础，详细分析了 KeyStone 平台所提供的多核硬件技术，为设计多核多操作系统板级支持平台 Scorpius 奠定了基础。

3. 设计了一个多核多操作系统板级支持平台 Scorpius 的模块化架构，设计了一种在 Scorpius 平台上应用和操作系统的部署模型。

4. 采用层次化设计方法，将硬件架构和操作系统软件进行隔离，抽象出多处理器架构下必须实现的各个功能模块，从各个层次的各个模块开始进行详细设计与实现。

6.2 展望

设计并开发一个支持多种操作系统，多种体系架构的万能板级系统平台要比设计实现一个操作系统要复杂困难的多，这不仅仅是因为系统平台不仅需要向支持和抽象所有体系架构提供的硬件功能，还需要向上为用户操作系统提供各种各样的服务。用户操作系统的多样性为平台架构的可扩展性，可移植性提出了非常高的要求。在时间有限的情况下，目前 Scorpius 平台还有很多不足的地方，针对目前存在的问题和已经获得的成果，今后还要开展以下工作：

1. 进一步测试目前 Scorpius 平台的模块功能，移植更多的单核或多核嵌入式操作系统到 Scorpius 平台上。

2. 将 Scorpius 平台移植到 ARM 和 X86 等通用计算平台上，从底层入手扩展 Scorpius 平台支持的硬件架构种类。

3. 实现基于 Scorpius 平台的单处理器虚拟化技术，允许在同一个 CorePac 组群中运行多个单核或多核操作系统。

致　谢

研究生生涯和学位论文一起进入了尾声，首先诚挚的感谢指导教授罗蕾老师，李允老师以及电子科技大学嵌入式工程软件中心的其他老师，各位老师悉心的教导使我得以一窥嵌入式多处理器系统软件架构领域的深奥，感谢各位老师不时的指点我正确的方向，使我在这些年中获益匪浅。

本论文的完成还要特别感谢赵焕宇老师，王丽杰老师提供软硬件资源的支持，没有赵老师提供的 TMS320C6678 评估板，本论文的研究仅仅只能停留在理论设计阶段，是赵老师提供的硬件设备让论文的成果能得以实现。没有王老师提供的项目机会，学生也不会涉足嵌入式多核领域。

另外还要感谢德州仪器在线技术社区以及开源社区对我提供的帮助，论文中碰到的很多问题如果没有社区专家的帮助是无法顺利解决的。感谢不厌其烦的指出我研究中所存在问题的各位教研室同学。感谢启发我思考问题方法，研究问题方法的阿里巴巴集团彦军老师及筝心师兄。

最后，谨以此文献给挚爱的父母。

参考文献

[1] 陈军. MPC7448 最小系统的设计与实现[J]. 江汉大学学报（自然科学版），2009, 37(2): 52-56

[2] 都思丹. 前言：嵌入式多核处理器系统及视频信号处理技术研究进程[J] . 南京大学学报（自然科学版），2009, 45(1): 1-4

[3] 吴非. 嵌入式实时操作系统 uC/OS-II 与 eCos 的比较[J]. 单片机与嵌入式系统应用，2004(10): 15-17

[4] Moore, G. E. Cramming More Components onto Integrated Circuits[J]. Electronics, 1965, 38(8): 114-117

[5] Moore, G. E. Progress in digital intergrated electronics[C]. Proceedings of 1975 International Electron Devices Meeting, Las Vegas, 1975: 11-13

[6] 于子萍. 嵌入式计算机系统的发展与展望[J]. 硅谷，2008(16): 53-54

[7] 何立民. 嵌入式系统支柱学科的交叉与融合[J]. 单片机与嵌入式系统应用，2008(5): 5-7

[8] A. Hung, W. Bishop. Symmertic multiprocessing on programmable chips made easy[J]. Design, Automation and Test in Europe, 2005(1): 240-245

[9] Sek Meng Chai, Taha T. M. Heterogeneous architecture models for interconnect-motivated system design[J]. Very Large Scale Intergration(VLSI) Systems, 2000, 8(6): 660-670

[10] John Michael Keogh. Circular transportation facilitation device[M]. Australia: Patent, 2001, 412-413

[11] Namkung, E. Effects of SMP on Biofilm-Reactor Performance[J]. Environ, 1988, 114(1): 199-210

[12] ARM Limited. Cortex-A Series Programmer’s Guide[S]. Cambridge: ARM Limited, 2011: 78-79

[13] Robert A. Alfieri. Operating system for a non-uniform memory access multiprocessor system[P]. U. S. A, CN, US5517648A, 1995

[14] ARM Limited. ARM Architecture Reference Manual v8[S]. Cambridge: ARM Limited, 2013

[15] Texas Instruments. TMS320C6678 Digital Media System-on-Chip[S]. Dallas: Texas Instruments, 2010

[16] 赵懿. 多核嵌入式系统的实时性研究[D]. 浙江：浙江大学，2007, 9-12

[17] 周彩贞. 嵌入式操作系统 uClinux 裁剪技术研究[D]. 武汉理工大学，2007, 8-12

[18] 张新英. STM32 上移植 μC/OS-II 的研究[J]. 商品与质量·理论研究，2010(10): 139

[19] 谭沛. 基于 VxWorks 的环控数据处理计算机软件的设计与实现[D]. 西安电子科技大学，

2011, 8-12

[20] Texas Instruments. TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor Data Manual[S]. Dallas: Texas Instruments, 2012

[21] 单祥茹. TI 多核 DSP 再出新品：高性能计算时代即将到来[J]. 中国电子商情：基础电子，2012(1): 29-30

[22] Texas Instruments. KeyStone Cache User Guide[S]. Dallas: Texas Instruments, 2012

[23] Chuhua hu, David Bell. KeyStone Memory Architecture[M]. Dallas: Texas Instruments, 2012, 5-7

[24] Mark S. Papamarcos, Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories[J]. 84 Proceedings of the 11th annual international symposium on Computer architecture, 1984, 12(3): 348-354

[25] 刘玉利. 线程同步与互斥[J]. 中国科技博览，2010(21): 42-42

[26] Texas Instruments. KeyStone CorePac User Guide[S]. Dallas: Texas Instruments, 2012

[27] Abraham Silberschatz, Peter Baer Galvin. Operating System Concepts Essentials[M]. Danvers: John Wiley & Sons. Inc, 2011, 512-514

[28] Ann Mclver McHoes, Ida M. Flynn. Understanding Operating Systems[M]. Danvers: Course Technology, 2009, 323-334

[29] Texas Instruments. KeyStone PLL User Guide[S]. Dallas: Texas Instruments, 2012

[30] David G. Carson. Multi-processor programmable interrupt controller system[P]. U. S. A, CN, US5283904A, 1993

[31] Wolfgang Mauerer, Professional Linux Kernel Architecture[M]. Indiana: Wiley Publishing, Inc, 2008, 712-720

[32] Andrew S. Tanenbaum, Albert S. Woodhull. Operating Systems Design and Implementation[M]. London: Pearson Education Ltd, 2006, 472-474

[33] Robert Sedgewick, Kevin Wayne. Algorithms[M]. San Francisco: Addison-Wesley, 2011, 612-630

[34] Karim Yaghmour. Embedded Android[M]. Cambridge: O'REILLY, 2012, 312-320

[35] 周航慈. 基于嵌入式实时操作系统的程序设计技术[M]. 北京：北京航空航天大学出版社，2011，192-194

攻读硕士学位期间取得的研究成果

[1] 参与嵌入式安全领域基于 VxWorks 实时操作系统的协议转换网关项目，主要负责 X86 及 PowerPC 设备 VxWorks 驱动编写，基础应用移植及项目代码编写

[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方法研究技术项目，主要负责多核同步机制，多核核间通信，TMS320C6678 硬件底层驱动以及中断控制管理等的设计与编码工作