论文摘要

I/O 系统的性能决定着服务器，工作站等网络中心设备的整体表现。而 PC 服务器和传统的工作站的 I/O 体系源于单用户的 PC 台式机，而不是为处理大吞吐量任务的专用服务器而设计的，一旦成为网络中心设备后，数据传输量大大增加，因而 I/O 数据传输经常会成为整个系统的瓶颈。智能输入/输出技术（Intelligent I/O，简称 I2O）即是为了解决这一问题而提出的一种可靠、廉价的解决方案。I2O 智能输入/输出技术把任务分配给智能 I/O 系统，在这些子系统中，专用的 I/O 处理器将负责中断处理、缓冲存取以及数据传输等烦琐任务，这样系统的吞吐能力就得到了提高，服务器的主处理器也能被解放出来去处理更为重要的任务。因此，依据 I2O 技术规范实现的 PC 服务器在硬件规模不变的情况下能处理更多的任务，作为中小型网络核心的低端 PC 服务器可以从中获得更多的性能提高。

本课题研究目前已经广泛应用于各种中小型网络核心的低端 PC 服务器和工作站的 I2O 通信协议及其实现，主要侧重它的协议规范和具体实现。

本论文按照以下顺序介绍 I2O 协议以及它在某项目中的具体实现：

首先，介绍 PCI 总线，因为 I2O 到目前为止主要是基于 PCI 总线，而且本文相关的实现部分也是以 PCI 总线为基础的，同时 PCI 也是当前使用最为广泛的局部总线，所以有必要做相关说明。

其次，作为本文的重点，介绍 I2O 技术出现的背景，从它的协议看它的优势和对以往 I/O 技术的改进，并结合后续将要介绍的具体实现，说明 I2O 协议中定义的消息通信过程。在介绍 I2O 通信环境的同时，也结合本课题介绍了 I2O 规范是如何基于 PCI 总线的，两者是如何结合在一起实现 I2O 通信的。

最后，介绍 I2O 协议在我所做项目中的具体实现，使我们对 I2O 有更清晰的理解。主要介绍本项目实现的总体情况，并结合项目所使用的具体硬件环境详细介绍 I2O 通信的实现，且对比了我们的实现与 I2O 规范间的区别。

关键词： I2O，PCI，I/O

ABSTRACT

The throughput of the Input/Output system determine the network equipment's throughput, such as network server and workstation.Troditional design of servers and workstations are derived from Personal Computer, which is accessed by only one people, not intended to process huge throughput, so, once these equipment become a center of the network, data transfer of the Input/Output system will become bottleneck of the system. Intelligent Input/Output(I2O) protocol has been handed in to solve this problem,and it is a reliable.I2O technology assign data processing to intelligent I/O subsystem.In the subsystem,dedicated processor will process the incoming interrupt, buffer access,data transfer and so on,in this way,the throughput of the entire system will be improved greatly and the main processor of the server can do many more important work.

In this paper,it focuses on I2O communication protocol and its implementation in a project.Because I2O protocol has been mostly implemented on the PCI(Peripheral Component Interconnect)bus,so first,it gives you the impression of the PCI bus.Secondly,it will introduce the I2O protocol and its Advantages,and it also tells you the combination of the PCI bus and I2O protocol.At last, it introduces how I2O protocol has been implemented in our project,it introduces you a entire implementation of I2O communication,the implementation is based on the hardware and software of our project,so there are some differences between this implementation and the I2O protocol,the differences are introduced at the end of this paper.

Keywords: Intelligent I/O, Peripheral Component Interconnect, Input/Output

南京邮电大学
硕士学位论文摘要

学科、专业：工 学 通信与信息系统
研究方向：网络技术与应用

作 者：2003 级研究生 王守林 指导教师 郑少仁

题 目：基于 PCI 的 I2O 通信

英文题目：Communication on the PCI-based intelligent I/O platform

主题词：I2O PCI I/O
Keywords: Intelligent I/O Peripheral Component Interconnect Input/Output

南京邮电大学学位论文独创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得南京邮电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了谢意。

研究生签名：王守林 日期：2006.4.12

南京邮电大学学位论文使用授权声明

南京邮电大学、中国科学技术信息研究所、国家图书馆有权保留本人所送交学位论文的复印件和电子文档，可以采用影印、缩印或其他复制手段保存论文。本人电子文档的内容和纸质论文的内容相一致。除在保密期内的保密论文外，允许论文被查阅和借阅，可以公布（包括刊登）论文的全部或部分内容。论文的公布（包括刊登）授权南京邮电大学研究生部办理。

研究生签名：王守林 导师签名：郑少仁 日期：2006.4.12

前 言

要实现不同的应用和功能，任何一个微处理器都要与一定数量的部件和外围设备连接，但如果将各部件和每一种外围设备都分别用一组线路与 CPU 直接连接，那么连线将会错综复杂，甚至难以实现。为了简化硬件电路设计、简化系统结构，常用一组线路，配置以适当的接口电路，与各部件和外围设备连接，这组共用的连接线路被称为总线。采用总线结构便于部件和设备的扩充，尤其是制定了统一的总线标准，更容易使不同设备间实现互连。

I/O 系统的性能决定着服务器，工作站等网络中心设备的整体表现，而通常 I/O 系统又是通过总线将外围设备与系统互连起来的，所以提高系统的 I/O 性能离不开一个性能优越的总线。总线通常是一个硬件连接标准，但是一个完整的系统是由硬件和软件协调工作的，所以要想提高系统的整体性能，需要将一个性能优越的总线和一个能够充分发挥该总线特点的软件模型结合，形成系统高效的运行环境。智能输入/输出技术（Intelligent I/O，简称 I2O）不但针对目前典型的硬件环境提出了扩展建议，而且提出了一个基于典型系统的软件分层模型，通过软硬件两方面的改进，极大的提高 I/O 系统的性能。

本文讨论基于 PCI 总线的 I2O 通信及其在一个具体环境中的实现，共分为四章，具体安排如下：

第一章， 介绍总线概念，总线发展回顾。

第二章， 详细介绍 PCI 总线，分析讨论它的特点以及 PCI 总线上驱动的特点和共性。

第三章， 详细介绍 I2O 通信规范，分析制定该规范的初衷，分析它相对于以往的 I/O 通信的改进。在介绍 I2O 通信环境的同时，结合本课题介绍 I2O 规范是如何基于 PCI 总线的，两者是如何结合在一起实现 I2O 通信的。

第四章， 具体阐述在我所做的路由器项目中，使用 I2O 通信协议的考虑和具体的实现。主要介绍本项目实现的总体情况，并结合项目所使用的具体硬件环境详细介绍 I2O 通信的实现，且对比了我们的实现与 I2O 规范间的区别。

结束语，对全文进行总结。

第一章 总线及其发展

1.1 总线出现的背景

任何一种外围没有挂接设备的微处理器都是没有价值的，因为它什么都不能做，对我们毫无用处。所以必须在微处理器周围挂接上适应各种不同应用的设备，但如果单纯的将各部件或外围设备都分别用一组线路与CPU直接连接，那么连线将会错综复杂，很多控制功能甚至难以实现，而且也不方便整个系统的标准化。采用总线结构便于部件和设备的扩充，尤其制定了统一的总线标准更容易使不同设备间实现互连。

总线的出现基于以下的一些要求：

1. 模块化，计算机部件要具有通用性，应能尽量适应不同系统与不同用户的需求，设计必须模块化。

2. 外设的多样性，计算机部件产品模块供应出现多元化。

3. 兼容性，模块之间的联接关系要标准化，使模块具有通用性，有利于不同系统之间的兼容。

4. 标准化，模块设计必须基于一种大多数厂商认可的模块联接关系，即一种总线标准，这样才能推广。

1.2 总线及其分类

1.2.1 总线定义和分类

总线是按一定的传输规则组织起来的信号线集合，它是模块间传输信息的公共通道，计算机各部件间通过它可进行各种数据和命令的传送。总线是为了简化硬件电路设计、简化系统结构所使用的一组线路，并且配置以适当的接口电路，将CPU和各部件及外围设备进行连接。

总线按其所承担的功能可分为三种类型:

1. 数据总线（Data Bus）：其功能是传输数据和指令信息 。
2. 地址总线（Address Bus）：其功能是传输内存或I/O设备地址 。
3. 控制总线（Control Bus）：其功能是给出总线周期类型、I/O操作完成的时刻、DMA周期、中断等有关控制信号。

总线按其相对于CPU和其它外围芯片的位置通常可以分为内部总线和外部总线：

1. 内部总线是微处理器片上内部各外围芯片与微处理器之间的总线，用于芯片一级

的互连。直观地可以理解为，在 CPU 内部，寄存器之间和算术逻辑部件 ALU 与控制部件之间传输数据所用的总线。

2. 外部总线是微机和外部设备之间的总线，微机作为一种设备，通过该总线和其他设备进行信息与数据交换，用于设备一级的互连。直观地可以理解为，是指 CPU 与内存 RAM、ROM 和输入 / 输出设备接口之间进行通讯的通路。

另外，从广义上说，计算机通信方式可以分为并行通信和串行通信，相应的通信总线被称为并行总线和串行总线。并行通信速度快、实时性好，但由于占用的信号线多，封装成本较高，且不适于小型化产品；而串行通信速率虽低，但在数据通信吞吐量不是很大的微处理电路中则显得更加简易、方便、灵活。

1.2.2 总线的基本概念

上文中提到总线是按一定的传输规则组织起来的信号线集合，为了更好地描述总线上的相关传输规则，定义了以下术语：

1. 总线周期：是通过总线完成一次内存读写操作或完成一次输入/输出设备的读写操作所必须的时间。通常由地址时间和数据时间组成。

a) 地址时间：用于 CPU 向内存或 I / O 设备送地址到地址总线的时间。

b) 数据时间：用于 CPU 向内存或 I / O 设备送数据的时间。

2. 总线的等待状态：若设备的读写速度慢，不能在一个总线的数据时间完成读写操作，必须再增加一到几个数据时间，这段时间称总线的等待状态。

3. 总线周期分类：依据具体操作的性质，可把一个总线周期分为内存读周期、内存写周期、I/O 读周期和 I/O 写周期。依据数据传输方式，可以把一个总线周期分为正常总线周期和 BURST 总线周期。

a) 正常总线周期（normal bus cycle）：若每次数据传输都要用地址时间和数据时间组成的完整的总线周期来完成读写，则称这种总线周期为正常总线周期。

b) BURST 总线周期（burst mode）：若给出一次地址信息（一个地址时间）后，接着用连续多个数据时间依次传输多个数据，则称这种运行方式为总线的突发传输方式，又称 BURST 总线周期

4. 总线仲裁：由于总线在某个具体时刻只能由一个设备占用，所以当总线上连接多个设备时，必须有分配总线控制权的机构（总线仲裁器）。一次总线操作中通常会涉及以下几个设备：

a) 总线主设备(bus master)：首先发起总线操作并启动传输过程，即申请总线使用权并发出命令控制总线运行的一方称为总线主设备。

b) 总线从设备(bus slave)：响应由主设备发出的命令并执行读写操作的设备称为总线从设备。

c) 总线仲裁器(bus arbiter)：当有多个总线主设备都发出申请总线的请求时，能决定哪一个申请者能取得总线的使用权的专用部件称为总线仲裁器。

5.数据传送控制（总线通信控制）：即同步问题，常用的有同步和异步通信两种方式。

a) 同步通信：是指在总线上传送数据时，通信双方使用同一个时钟信号进行同步，该时钟称为总线时钟。

b) 异步通信：是指在总线上传送数据时，允许通信双方各自使用自己的时钟信号，采用“应答方式”（握手方式）解决数据传输过程中的时间配合问题，而不是共同使用同一个时钟。

上述术语将有助于我们理解下文中的 PCI 总线规范。

1.2.3 总线发展介绍

随着微电子技术和计算机技术的发展，总线技术也在不断地发展和完善，使计算机总线技术种类繁多，各具特色。下面仅对在微机发展中，各种应用较为广泛的总线加以介绍。

1.2.3.1 内部总线

1. I2C 总线

I2C（Inter-IC）总线十多年前由 Philips 公司推出，是近年来在微电子通信控制领域广泛采用的一种新型总线标准。它是同步通信的一种特殊形式，具有接口线少，控制方式简化，器件封装尺寸小，通信速率较高等优点。在主从通信中，可以有多个 I2C 总线器件同时接到 I2C 总线上，通过地址来识别通信对象。

2. SPI 总线

串行外围设备接口 SPI（serial peripheral interface）总线技术是 Motorola 公司推出的一种同步串行接口。Motorola 公司生产的绝大多数 MCU（微控制器）都配有 SPI 硬件接口，如 68 系列 MCU。SPI 总线是一种三线同步总线，因其硬件功能很强，所以，与 SPI 有关的软件就相当简单，使 CPU 有更多的时间处理其他事务。

3. SCI 总线

串行通信接口 SCI（serial communication interface）也是由 Motorola 公司推出的。它是一种通用异步通信接口 UART，与 MCS-51 的异步通信功能基本相同。

1.2.3.2 外部总线

1. ISA 总线

ISA（industrial standard architecture）总线标准是 IBM 公司 1984 年为推出 PC/AT 机而建立的系统总线标准，所以也叫 AT 总线。它是对 XT 总线的扩展，以适应 8/16 位数据总线要求。它在 80286 至 80486 时代应用非常广泛，以至于现在奔腾机中还保留有 ISA 总线插槽。ISA 总线有 98 只引脚。

2. EISA 总线

EISA 总线是 1988 年由 Compaq 等 9 家公司联合推出的总线标准。它是在 ISA 总线的基础上使用双层插座，在原来 ISA 总线的 98 条信号线上又增加了 98 条信号线，也就是在两条 ISA 信号线之间添加一条 EISA 信号线。在实用中，EISA 总线完全兼容 ISA 总线信号。

3. VESA 总线

VESA（video electronics standard association）总线是 1992 年由 60 家附件卡制造商联合推出的一种局部总线，简称为 VL(VESA local bus)总线。它的推出为微机系统总线体系结构的革新奠定了基础。该总线系统考虑到 CPU 与主存和 Cache 的直接相连，通常把这部分总线称为 CPU 总线或主总线，其他设备通过 VL 总线与 CPU 总线相连，所以 VL 总线被称为局部总线。它定义了 32 位数据线，且可通过扩展槽扩展到 64 位，使用 33MHz 时钟频率，最大传输率达 132MB/s，可与 CPU 同步工作。是一种高速、高效的局部总线，可支持 386SX、386DX、486SX、486DX 及奔腾微处理器。

4. PCI 总线

PCI（peripheral component interconnect）总线是当前最流行的总线之一，它是由 Intel 公司推出的一种局部总线。它定义了 32 位数据总线，且可扩展为 64 位。PCI 总线主板插槽的体积比原 ISA 总线插槽还小，其功能比 VESA、ISA 有极大的改善，支持突发读写操作，33MHz、32bit 总线最大传输速率可达 132MB/s，可同时支持多组外围设备，支持 PCI－PCI 桥，大大提高了系统的可扩展性。PCI 局部总线不能兼容现有的 ISA、EISA、MCA（micro channel architecture）总线，但它不依赖于处理器，是基于奔腾等新一代微处理器而发展的总线。关于 PCI 的详细介绍见第二章。

5. PCI-X 总线

这是目前服务器网卡经常采用的总线接口，它与原来的 PCI 相比在 I/O 速度方面提高了一倍，比 PCI 接口具有更快的数据传输速度（2.0 版本最高可达到 266MB/s 的传输速率）。PCI-X 总线接口的网卡一般为 32 位总线宽度，也有的是用 64 位数据宽度的。

6. PCI Express 总线

PCI Express 采用了点对点串行连接，比起 PCI 以及更早期的计算机总线的共享并行架构，每个设备都有自己的专用连接，不需要向整个总线请求带宽，而且可以把数据传输率提高到一个很高的水平，达到 PCI 所不能提供的高带宽。相对于传统 PCI 总线在单一时间周期内只能实现单向传输，PCI Express 的双单工连接能提供更高的传输速率和质量，它们之间的差异跟半双工和全双工类似。

PCI Express 的接口根据总线位宽不同而有所差异，包括 X1、X4、X8 以及 X16（X2 模式将用于内部接口而非插槽模式）几种模式。PCI Express 也支持高阶电源管理，支持热插拔，支持数据同步传输，为优先传输数据进行带宽优化。

7. RS-232-C 总线

RS-232-C 是美国电子工业协会 EIA（Electronic Industry Association）制定的一种串行物理接口标准。RS 是英文"推荐标准"的缩写，232 为标识号，C 表示修改次数。RS-232-C 总线标准设有 25 条信号线，包括一个主通道和一个辅助通道，在多数情况下主要使用主通道，对于一般双工通信，仅需几条信号线就可实现，如一条发送线、一条接收线及一条地线。RS-232-C 标准规定的数据传输速率为每秒 50、75、100、150、300、600、1200、2400、4800、9600、19200 波特。RS-232-C 标准规定，驱动器允许有 2500pF 的电容负载，通信距离将受此电容限制，例如，采用 150pF/m 的通信电缆时，最大通信距离为 15m；若每米电缆的电容量减小，通信距离可以增加。传输距离短的另一原因是 RS-232 属单端信号传送，存在共地噪声和不能抑制共模干扰等问题，因此一般用于 20m 以内的通信。

8. USB 总线

通用串行总线 USB（universal serial bus）是由 Intel、Compaq、Digital、IBM、Microsoft、NEC、Northern Telecom 等 7 家世界著名的计算机和通信公司共同推出的一种新型接口标准。它基于通用连接技术，实现外设的简单快速连接，达到方便用户、降低成本、扩展 PC 连接外设范围的目的。它可以为外设提供电源，而不像普通的使用串、并口的设备需要单独的供电系统。另外，快速是 USB 技术的突出特点之一。USB1.1 标准的最高传输率可达 12Mbps，比串口快 100 倍，比并口快近 10 倍，而且 USB 还能支持多媒体；而 USB2.0 标准的传输速率可以高达 480Mbps。

9. IEEE 1394 总线

IEEE 1394 是为了增强外部多媒体设备与电脑连接性能而设计的高速串行总线，传输速率可以达到 400Mbps，利用 IEE1394 技术我们可以轻易地把电脑和摄像机，高速硬盘，音响设备等设备中存储的数据倒入到 PC 电脑中。它具有两种数据传输模式：同步(Isochonous)传输与非同步(Asynchronous)传输，同步传输模式可确保某一连线的频宽，对

于即时影像而言这是相当重要的。因为影音数据都有时间上的限制，无法接受过久的延迟。

IEEE 1394 支持热插拔，可以自动侦测设备的加入与移出动作，并对系统做重新整合，无须人工干预。

第二章 PCI 总线

第一章介绍了计算机总线的定义、类别、发展情况和目前流行的几种总线标准。本章将具体介绍本文重点关注的 PCI 总线。从 1992 年创立规范至今，PCI 总线已成为事实上的计算机工业总线标准。这主要归功于 PCI 本身的很多独创性的优点。下面将分节介绍 PCI 总线出现的背景，PCI 总线优势，并结合它的各种优点详细介绍与我们设计、实现 PCI 总线和设备驱动相关的内容。

2.1 PCI 总线的提出

1992 年以前，PC 机中流行的是 ISA 总线（8/16bit 的系统总线，最大传输速率仅为 8MB/s），该总线是以 IBM 的 PC/AT 总线为基础发展起来的工业标准总线。在 386 出现后，随着 CPU 频率的提升，ISA 总线速度就成了整个系统性能的瓶颈，影响 CPU 效率。伴随着 CPU 频率的大幅提升和各种应用对 I/O 性能要求的提高，为了提高 PC 机的整体性能，1992 年，Intel 在发布 486 处理器的同时，提出了 32bit 数据位宽的 PCI（Peripheral Component Interconnect,周边组件互连）总线。

最早提出的 PCI 总线工作在 33MHz 频率之下，传输带宽达到 133MB/s（33MHz *32bit/8），比 ISA 总线有了极大的改善，基本上满足了当时处理器的发展需要。随着对更高性能的要求，1993 年提出了数据位宽为 64bit 的 PCI 总线，后来又提出把 PCI 总线的工作频率提升到 66MHz 。目前广泛采用的是 32bit、33MHz 的 PCI 总线。PCI 总线是独立于 CPU 的系统总线，采用了独特的中间缓冲器设计，可将声卡、网卡、硬盘控制器等高速的外围设备直接挂在 CPU 总线上，使得 CPU 的性能得到充分的发挥。

2.2 PCI 总线

PCI 总线是一种不依赖于某个具体处理器的局部总线。从结构上看，PCI 为 CPU 提供了一级设备扩展总线，并由一个桥接电路实现对这一层的管理，实现 CPU 和外部设备之间的接口以协调数据的传送。管理器提供了信号缓冲，使之能支持 10 种外设，并能在高时钟频率下保持高性能。

相对于 ISA 这些早期总线来说，PCI 总线有很多优点，比如，工作频率提高，总线带宽提高并适应了 CPU 总线的要求，‘即插即用’功能，中断共享等；另外 PCI 总线具有严格的标准和规范，这就保证了它具有良好的兼容性，符合 PCI 规范的扩展卡可插入任何 PCI 系统可靠地工作；PCI 总线可以提供高数据传送速率（132MB/s）或（264MB/s）；PCI 总

线与CPU无关，与时钟频率亦无关，可适用于各种平台，支持多处理器和并发工作；PCI总线还具有良好的扩展性，通过PCI-PCI桥路，可进行多级扩展。

PCI总线的自动配置功能使其应用更为简单、方便。由于该总线标准为元件和插件分配了相应的配置寄存器，对于某个系统只要有嵌入的自动配置软件，就可以在系统加电时自动配置PCI总线上的设备，为用户提供了极大的方便，以上特点使得PCI总线成为目前PC机上最通用的一种总线。

2.2.1 基本概念

从数据宽度上看，PCI总线有32bit、64bit之分；从总线工作频率上看，有33MHz、66MHz两种。目前流行的是32bit位宽，工作在33MHz频率下的PCI总线（32bit*33MHz）。改良的PCI系统—PCI-X，带宽最高可以达到64bit*133MHz，这样就可以得到超过1GB/s的数据传输速率。如果没有特殊说明，以下的讨论以32bit*33MHz为例。

不同于ISA总线，PCI总线的地址总线与数据总线是分时复用的，这样做的优点是可以节省接插件的管脚数。在进行数据传输时，由一个PCI设备作为发起者（主控，Initiator或Master），而另一个PCI设备作为目标（从设备，Target或Slave）。总线上的所有时序的产生与控制，都由Master发起。由于PCI总线在同一时刻只能供一对设备完成传输，所以当一个PCI总线上挂接多个PCI设备时，就要求有一个仲裁者（Arbiter）来决定谁有权力拿到总线的主控权。PCI规范中有专门关于总线仲裁的说明和定义。

作为一个总线标准，引脚功能及操作时序是非常重要的，下面以PCI基本读操作时序为例说明PCI操作的时序要求，首先介绍32bit位宽的PCI总线管脚分类：

1. 系统控制：

1) CLK：时钟信号线，提供PCI操作时钟，除了RST#、INTA#、INTB#、INTC#和INTD#之外的PCI操作信号都是在CLK信号的上升沿采样，它由外部时钟源提供。

2) RST#：复位信号线，该信号可以和CLK时钟信号异步。

2. 传输控制：

1) FRAME#：Cycle Frame，由PCI操作主控设备发出，标志操作开始与结束。

2) IRDY#：Initiator Ready，表示PCI主控设备可以完成当前操作。

3) TRDY#：Target Ready，表示PCI从设备可以完成当前操作。TRDY#和IRDY#同时生效表明后续的数据传输可以进行。

4) DEVSEL#：Device Select，当某个从设备发现自己被寻址时置低应答。

5) IDSEL：Initialization Device Select，在配置空间读写时作为片选信号线。

6) STOP#：从设备主动结束传输数据的信号。

3. 地址与数据总线：

1) AD[31:0]：地址/数据分时复用总线。

2) C/BE#[3:0]：Bus Command and Byte Enable，命令/字节使能信号。当地址信号有效时，这四个信号线定义的是 Bus Command；当数据信号有效时，这四个信号线定义的是 Byte Enable。

3) PAR：奇偶校验信号。

4. 仲裁信号：

1) REQ#：Request，主设备用来向仲裁器请求总线使用权的信号。

2) GNT#：Grant，仲裁器允许主设备得到总线使用权的信号。

5. 错误报告：

1) PERR#:Parity Error，数据奇偶校验错。

2) SERR#:System Error，系统奇偶校验错。

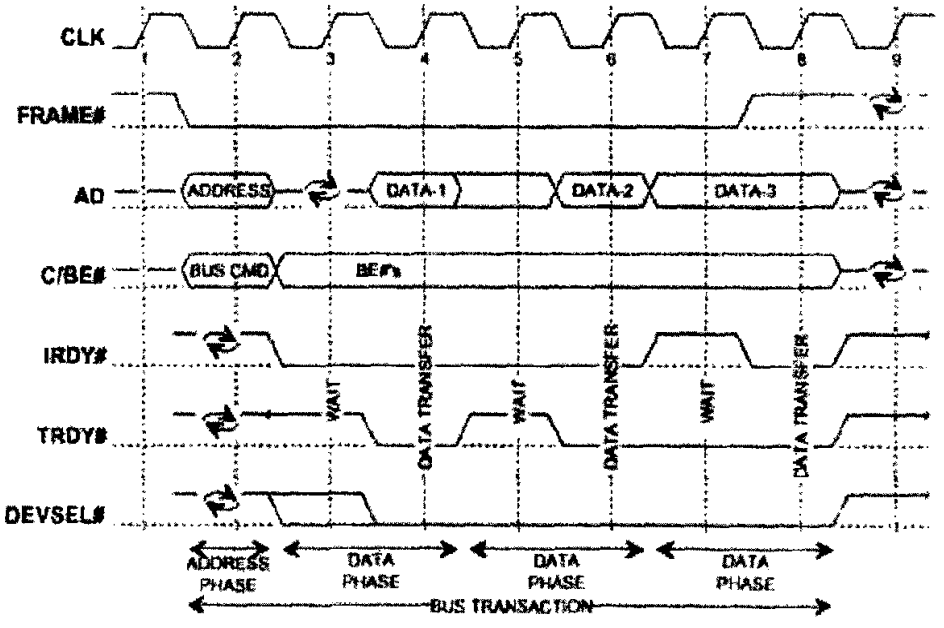

图 1 PCI 基本读操作时序

上图是 PCI 基本读操作时序，当 PCI 总线进行操作时，发起者(Master)先置 REQ#，当得到仲裁器(Arbiter)的许可时(GNT#)，会将 FRAME#置低，并在 AD 总线上放置 Slave 地址，同时 C/BE#放置命令信号，说明接下来的传输类型。所有 PCI 总线上设备都需对此地址译码，被选中的设备要置 DEVSEL#以声明自己被选中。然后当 IRDY#与 TRDY#都置低

时，可以传输数据。当 Master 数据传输结束前，将 FRAME#置高以标明只剩最后一组数据要传输，并在传完数据后放开 IRDY#以释放总线控制权。

这里我们可以看出，PCI 总线的传输是很高效的，发出一组地址后，理想状态下可以连续发数据，峰值速率为 132MB/s。实际上，目前台式机支持 33MHz*32bit 标准 PCI 的桥芯片一般可以做到 100MB/s 的连续传输。

2.2.2 PCI 总线优点

PCI 总线之所以能够迅速的取代 ISA 成为流行，而且到目前为止还是应用得最广的总线标准，它的以下几大优点功不可没：

1. PCI 总线的实现独立于具体 CPU 类型（不同类型的 CPU 都可以实现 PCI 标准），且 PCI 总线与 CPU 时钟频率无关，可适用于各种平台，支持多处理器。通常系统中实现 PCI 总线标准有两种方式：

1) 通过挂接在 CPU 上的桥芯片实现 PCI 控制器，进而实现 PCI 总线功能。

2) 通过 CPU 上集成的 PCI 总线控制器实现 PCI 总线功能。

2. PCI 总线具有良好的扩展性，通过 PCI-PCI 桥路，可进行多级扩展，详见下文说明。

3. 自动配置，‘即插即用’功能，详见下文说明。

4. 中断共享，PCI 规范定义了四条中断引脚（INTA#、INTB#、INTC#和 INTD#），设备制造商可以根据产品特点任意组合这些中断引脚形成一个中断线引到系统中断控制器或者 CPU，参见下文 Interrupt Pin 说明。

下面将着重介绍“即插即用”和可扩展性。

2.2.2.1 自动配置

所谓“即插即用”，是指当 PCI 设备插入系统时，系统会自动对该设备所需资源进行分配，如基地址、中断号等，并自动寻找相应的驱动程序。而不象旧的 ISA 板卡，需要进行复杂的手动配置。

实际的实现远比说起来要复杂。为了实现“即插即用”功能，在所有 PCI 设备中（除了直接挂接在 CPU 上的主桥，其上实现 PCI 控制器），必须包含一组寄存器，叫“配置空间（Configuration Space）”。PCI 规范中将配置空间大小定义为 256 字节，且将它分为预定义头（Predefined header region）和与设备相关的部分（Device dependent region）。

预定义头部分定义的是设备标识和设备控制相关的寄存器。预定义头部分又可分为两个部分，前面 16 字节和剩余字节。之所以分成这两部分是由于，前面 16 字节对于所有 PCI

设备具有相同的定义，而后面剩余字节定义的寄存器格式，不同的设备可能实现情况不同，它们是利用 Header Type 域（偏移为 0x0E）进行区分。下面将介绍我们通常所用的配置空间结构（Header Type＝0x00）：

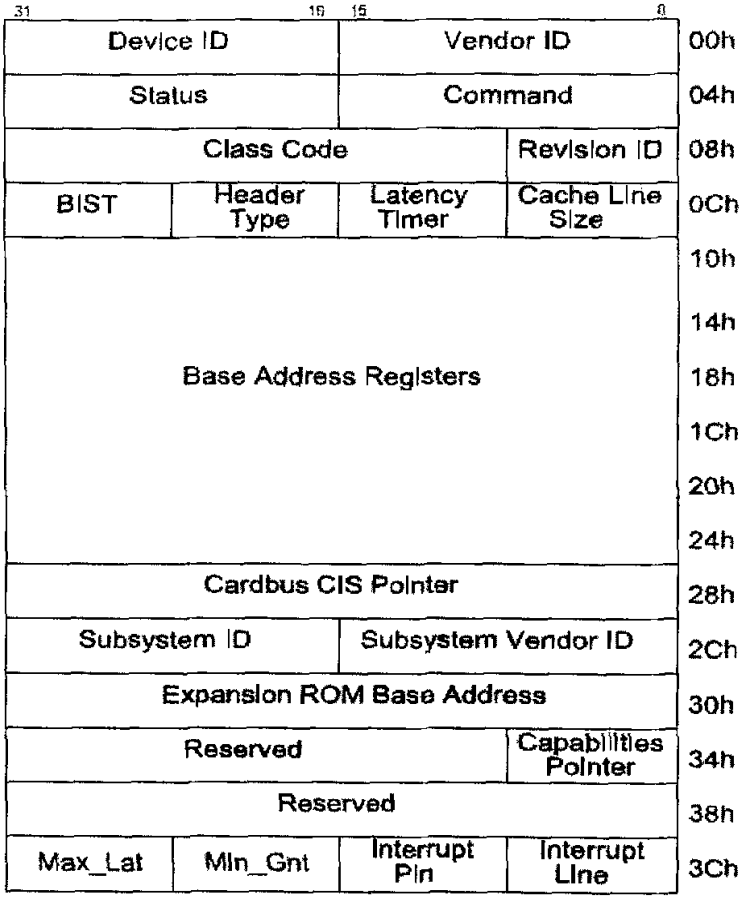

图 2 PCI 配置空间定义（Type:00h）

下面介绍常用的几个域，其它域的详细说明参见 PCI 规范。

Base Address Register：系统分配给 PCI 设备的 I/O 映射空间或 Memory 空间基地址。上电时，系统的初始化代码（通常是存放在 ROM 或 EEPROM 中的启动代码）将按照 PCI 规范配置 PCI 设备的 I/O 映射空间、Mem 映射空间等寄存器，使系统运行后能够正常访问 PCI 设备，同时也方便后续启动代码为整个系统分配地址空间。PCI 设备占用的空间分配完成后，操作系统将记录 PCI 设备占用空间的情况，这样就不必手工设置开关来分配内存或基地址了。后续系统将根据这个空间来访问 PCI 设备的相关寄存器。

Interrupt Pin：该域的值分别对应 INTA#、INTB#、INTC#和 INTD#中的一个。PCI 标准中同一个 PCI 设备中可以包含多个具有不同功能的 PCI 子设备，所以当一个 PCI 设备是

多功能设备时，可以把不同的中断引脚分配给不同的子设备；对于单功能的 PCI 设备只使用 INTA#引脚。

Interrupt Line：上电时，系统的初始化代码将为 PCI 设备分配一个系统中断线，用于 PCI 设备向系统发出中断请求。驱动和操作系统通常根据这个域为 PCI 设备挂接中断处理程序并知道中断优先级的分配。对于中断线的分配通常是根据系统中断的整体分配情况而定，如果系统中只有固定个数的 PCI 设备，则系统的初始化代码通常会分配一个固定的中断线，而不需要扫描 PCI 设备，这样可以节省系统初始化时间。

2.2.2.2 扩展性

通常，PCI 设备挂接在一个 PCI 桥上(通常 PCI 桥是由系统桥实现的一个 PCI 控制器)，PCI 的可扩展性主要表现在两个方面：

1. PCI 桥既可以直接挂在 CPU 总线上，又可以挂在上一级的 PCI 桥上。
2. PCI 规范在配置空间定义了很多可扩展字段，允许厂商根据具体需要扩展自己的系统，完善相关功能。

以上两个优点使得系统厂商有更大的自由空间来定制自己的系统，也是 PCI 总线迅速流行的一个重要原因。下面就从上面的两个方面具体介绍。

PCI 总线作为 CPU 总线的扩展，通常直接挂在 CPU 总线上，但是如果系统中有很多 PCI 设备时，由于总线驱动能力有限，可能一级 PCI 桥已经不能够满足要求，此时就可以利用 PCI－PCI 桥扩展 PCI 总线，这样可以挂接更多的设备，而不会引起总线共享上的问题。

当挂接的 PCI 设备功能非常丰富或者该 PCI 设备需要额外的初始化时，PCI 配置空间也提供了相关的支持。这就是 Expansion ROM Base Address 字段，它可以指向扩展 ROM 的基地址，使得 ROM 中的扩展程序可以执行，实现设备的扩展功能；而且，配置空间也实现了 Capability Pointer 域用于配置空间的扩展。

第三章 I2O 通信规范

由于 CPU 主频速度越来越快，I/O 速度就成为系统性能提升的瓶颈，而且很多高端应用场合对 I/O 通信要求越来越高，传统 I/O 已经不能满足要求，I2O 协议就是在这样的背景下出现的。该协议最初是由 Intel 的一个开发小组开发，后来随着 Microsoft 等很多公司的加入，组成了一个开发组织 I2O SIG（I2O Special Interest Group）。I2O 并没有在 I/O 通信总线上有本质的创新，它的目标是基于目前流行的总线，提供一种可靠而且廉价的提高 I/O 通信效率的方式。它的目的是提高服务器、工作站等的 I/O 通信性能，为缓解目前大型计算机系统的 I/O 瓶颈问题提供一种可行的方案。I2O SIG 于 1997 年提出了 I2O v1.5 规范，并成功应用于各种服务器、工作站等大型系统，I2O 是一个独立于具体物理总线的通信协议，目前主要基于 PCI 总线。

3.1 设计初衷及其优点

I2O 结构规范描述了一个开放的在网络系统环境下开发设备驱动的架构。这个架构独立于操作系统（OS）、处理器和系统 I/O 总线。该规范努力使 I/O 驱动开发标准化，并使得设备驱动能在不同的处理器平台上运行。

当前高端网络和存储技术的趋势是：将很多功能推向下层，由驱动实现，同时又要求驱动的高效率。为了满足这些要求，硬件厂商开始生产包含一些智能的产品，它们包含自己的 I/O 控制器，用来专门处理 I/O 事务，比如存储中的 RAID 控制器和网络中的 ATM 控制器。

在硬件级加入智能有很多好处。使用专门的处理器完成 I/O 事务减少了对 CPU 资源的消耗，它将 I/O 中断交给能够更高效处理 I/O 事务的处理器完成，而这些 I/O 中断以前都是由主 CPU 完成处理的（这将打断主 CPU 上的其他应用程序的运行）。I2O 规范不只是向硬件中加入智能，它的目标是标准化 I/O 智能平台，使其具有以下优势：

1. 精简的驱动接口。目前硬件厂商针对每个硬件必须提供多个驱动，不同驱动工作在不同的 OS 下，OS 和硬件厂商都需要测试不同版本的驱动。I2O 的目标是使 OS 厂商针对每一类设备只需要提供一个驱动，而且只需要提供 OS 部分的驱动。硬件厂商也是一样，只需要提供硬件部分的驱动，并且针对一种设备只需要一个版本的驱动。这就使得硬件和 OS 厂商都能够专注自己部分的开发，使其更加优化，I2O 规范提供它们之间的标准接口。

2. 扩展的经济性。除了为 I2O 设备提供系统接口外，I2O 规范还为 I/O 子系统定义了一个操作环境。这个努力使得系统提供商可以建立一个 I/O 平台能够支持非 I2O 设备，这一个操作环境。这个努力使得系统提供商可以建立一个 I/O 平台能够支持非 I2O 设备，这

就提供了比较经济的可扩展性。当然单个的 I/O 设备肯定比一个 I/O 处理器便宜，但是，多个 I/O 设备在同一个 I/O 平台下运行就可以建立一个经济的智能解决方案。

3. 驱动的分层结构。为了实现精简的驱动接口，方便驱动和 OS 开发人员完成软件开发，I2O 规范将原有驱动实现的功能进行分层，详细说明见下面 I2O 通信规范的具体介绍。

4. I2O 协议是基于目前已经存在的总线标准，不需要额外定义规范，方便硬件厂商实现 I2O 产品。

3.2 基本概念

I2O 结构定义了创建设备驱动的环境，这些驱动功能上被分成了驻留在主机操作系统中的与主机操作系统相关的部分和驻留在 I2O 子系统中的与 I2O 通信相关的部分。I2O 规范中描述的通信模型是基于消息传递协议（message-passing protocol）的,这个协议和网络中的面向连接的通信方式类似（TCP 握手）。具体的传输层提供 I/O 子系统的硬件抽象，使得消息传递独立于硬件系统。

这一节总体描述设备驱动并且解释它的分层结构（达到独立于硬件平台的目的）。然后介绍消息传输层，它使得分层成为可能。从驱动的角度看，介绍了下面两种接口：

1. 将驱动分层所产生的接口
2. 提供分层的各单元之间通信的消息层接口

3.2.1 硬件结构

I2O 操作是针对一个包括单主机、单个智能 I/O 子系统（这个系统又包括多个 I/O 处理器）的系统进行优化的。主机是指一个或多个应用处理器和它们运行所需的相关资源，这些处理器允许统一的 OS。下图是一个典型的 I2O 系统的硬件结构，一个主机实体和多个嵌入式 I/O 处理实体。我们下面涉及的嵌入式 I/O 处理实体都被称作 IOP（I/O Platform，专门用来处理 I/O 操作，由处理器、内存和 I/O 设备组成）。

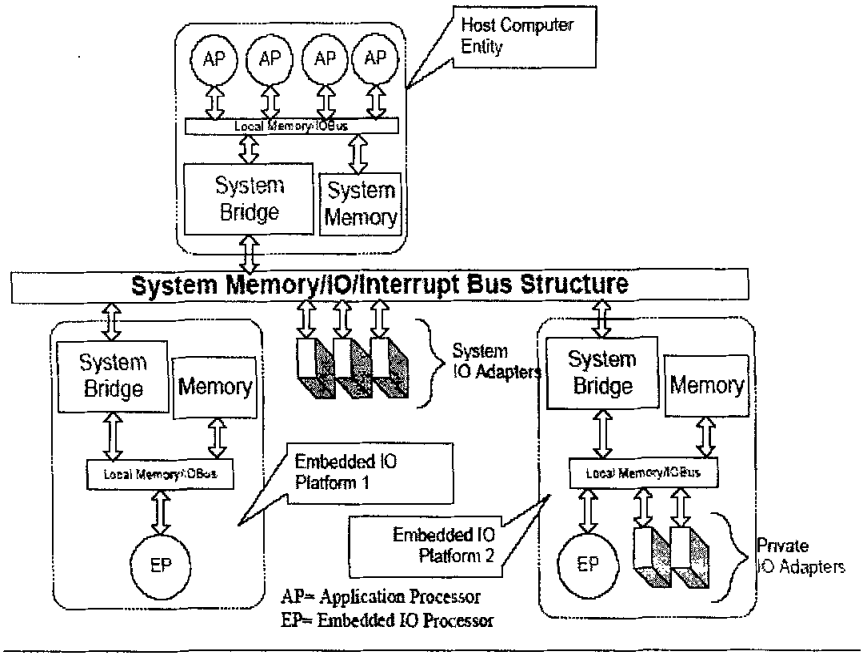

图 3 I2O 典型硬件结构

系统 I/O 适配器可以从系统总线访问，所以可以由 IOP 或主机控制。这种适配器可以直接安装在主板上也可以以卡的形式插在主板上。以前的系统 I/O 适配器通常是由运行在主机上的驱动控制的，IOP 要想控制它的话，必须做额外的工作，起码要使中断信号连接到 IOP 上。为此，设置一个专用的 I/O 适配器（Private I/O adapter）和 IOP 绑定，直接由 IOP 管理，Private I/O adapter 对于主机来说是看不见的。

本文将要说明 I2O 通信在一个具体项目中的实现，其硬件结构与 I2O 规范的定义有所区别。本课题中主处理器（Main Processor，简称 MP）侧软硬件环境对应于 I2O 规范中的主机实体，从处理器（Bridge Processor，简称 BP）侧软硬件环境对应于 I2O 规范中的 IOP。详细介绍可以参见第四章。

3.2.2 驱动分层模型

将驱动分成不同层，并且在它们之间定义标准的消息传递接口就可以将它们从实现上分开。不同的层可以运行在不同的处理器上甚至可以运行在不同的 OS 下。下图说明了驱动的分层结构，

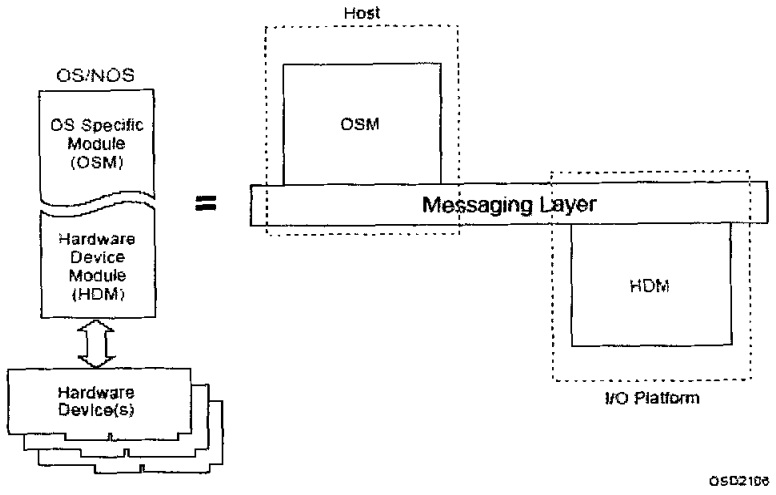

图 4 I2O 驱动分层模型

驱动分层后使得原来的整个系统分成了两块：

1. 操作系统相关模块（OS-Specific module，简称 OSM），通常是由 OS 厂商提供这个模块，这个模块通常也不包括硬件相关的代码。

2. 设备驱动模块（Device driver module，简称 DDM）。这一层为 I/O 适配器和挂在它下面的设备提供接口，通常是硬件厂商提供这个模块，不包含 OS 相关的代码。上图中的 HDM 加上一些中间服务软件构成 DDM，下文中无特殊说明均用 DDM 表示 HDM 和这些中间软件构成的整体。

通过消息层（Message layer），OSM 可以和任何一个 DDM 通信，也就是说，一个 OSM 可以和多个 DDM 同时通信。

本文主要讨论基于 I2O 的通信，所以将重点介绍 IOP（I/O Platform）和消息层的消息传递机制。I2O 目前主要用于基于 PCI 总线环境的各种应用，本课题实现的 I2O 通信也基于 PCI 总线，所以本文默认情况是针对 PCI 环境下的 I2O 通信。

3.3 通信原理

I2O 规范制定的目标是使主机和 IOP、各 IOP 之间的通信标准化，且提高通信效率。而要想高效的传递消息必须指定消息的存储介质和传递的一般流程。下面将具体介绍 I2O 规定的内存访问方式和利用这些内存存放消息时的消息传递机制。

3.3.1 通信环境

为了更加清晰的描述 I2O 通信环境，我们将 I2O 的各种应用抽象成一个通用的应用环境 I2O segment（通常由一个主机和多个 IOP 组成），下图是 I2O segment 内部各组件之间

的关系图：

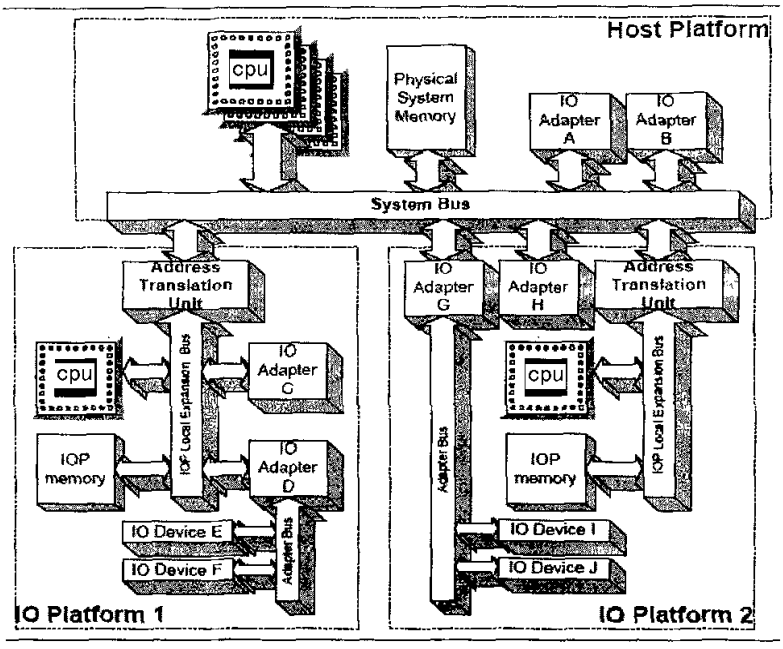

图 5 I2O Segment 示例

3.3.1.1 I/O 设备域

I/O 适配器 A 和 B 直接被 Host OS 控制，而 I/O 适配器 G 和 H 挂在系统总线上但是被 IOP2 控制，以上四个适配器都可以在系统总线上直接寻址；而 I/O 适配器 C 和 D 可以在 IOP1 的局部总线上直接访问，I/O 适配器 D 和 G 下面还挂接了通过它们自己的本地总线访问的设备。除了适配器 A 和 B 以外的其他适配器都由 IOP 控制，都会被登记在 I2O 配置表中，并且被抽象成各种服务，在 Host 或其他 IOP 看来它们只提供不同的服务，IOP 只为 Host 或其它 IOP 提供服务接口。

3.3.1.2 地址域

典型的系统是由一个 Host 和一个 IOP 或多个 IOP 构成的。每个 IOP 包含并管理了独立于其他 IOP 的、它自己本地的内存和 I/O 系统。Host 和 IOP 所看到的内存是不同的：Host 认为系统内存和适配器的内部空间是同一地址域，而 IOP 的地址域就是它自己本地总线的空间，不能直接访问系统空间。所以 IOP 必须将本地的一部分内存映射到系统空间，Host 才能通过系统空间访问 IOP 的内存，这部分内存叫做物理共享内存（我们在后续实现中将用到 I2O 规范的这部分内容）。首先，介绍一下典型 I2O 系统中的三类内存。

三类内存：

1. 系统内存：这段内存空间只能通过系统总线访问，所以只能通过系统内存地址指

定相应的内存位置。IOP 可以使用 DMA 机制在系统内存和自己的本地内存间传递数据。

2. IOP 私有内存：这段内存只能通过 IOP 的本地总线进行访问，所以只能通过本地地址指定相应的内存位置。

3. 共享内存：这是一段 IOP 的本地内存，但是它既可以通过系统总线访问，也可以通过本地总线访问，每个内存位置都对应一个系统总线地址和一个本地总线地址（比如：我们后续实现中在 Host 本地内存中划分的一段 Agent 和 Host 共享内存）。

由于很多模块运行在 IOP 本地，所以 IOP 必须提供本地地址和系统地址的相互转换机制，也即地址翻译。

3.3.1.3 地址翻译单元

共享内存是 IOP 本地的一段内存，但是系统可以通过地址翻译单元 ATU（Address Translation Unit）访问。ATU 映射 IOP 本地的一段内存到系统内存域中，所以相应系统内存的访问将被翻译成对该段 IOP 本地内存的访问。这种翻译机制使得每个 IOP 有自己独立于系统空间的地址分空间，但是其他 IOP 和系统仍然能够访问它的一段本地内存空间（共享内存）。ATU 就是实现将 IOP 的一段内存映射到系统空间，在系统看来这段空间就是系统空间的一部分。ATU 转换示意图：

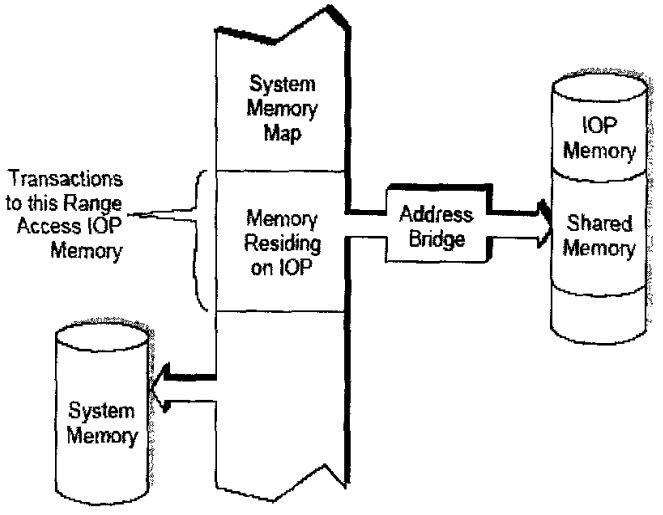

图 6 内存地址空间翻译

ATU 并不将系统内存空间映射到 IOP 本地内存空间，也就是说，IOP 不能通过访问自己的空间达到传递数据给系统的目的（而系统可以，通过 ATU），所以 IOP 必须提供将数据从本地内存（或本地设备）传递到系统内存的机制（通常是 DMA），而反方向可以通过共享内存实现，当然也可以通过该机制实现。从系统的角度看，共享内存是系统可以直接

访问的 IOP 物理内存的一部分；从 IOP 角度看，共享内存是系统和其他 IOP 可以访问的本地内存的一部分。对于每个 IOP 来说，共享内存的本地地址和系统地址的差值是个常数，加或减这个差值后就是相应的本地地址或系统地址。

在本课题中，考虑到具体软硬件环境，我们在从处理器侧（IOP 侧）内存空间中划分一段空间作为主从处理器间通信的共享空间，所以本课题实现中系统内存是主处理器侧内存空间，共享内存是在从处理器侧划分出来的那部分内存，IOP 私有内存就是从处理器侧除了共享内存以外的内存空间。

本课题实现中，从处理器侧的桥芯片将作为通信的 ATU（下文中提到 ATU 指的就是从处理器侧的桥芯片），所以这是单向地址翻译，此时从处理器直接访问共享内存（就像访问它的其它本地内存一样），而主处理器通过 PCI 总线空间访问共享内存。当主处理器通过 PCI 访问共享内存时，ATU（具体是桥中的 PCI 控制器）作为 PCI 从设备，所以此时主处理器发出的访问地址将由 ATU 进行地址译码决定访问哪个设备（通常可以访问内存和桥上挂接的各种设备），通过通信初始化阶段的配置，此时 ATU 将地址译码到共享内存，这样就实现了规范中的地址翻译。

3.3.2 两种传递方式

图 7 显示了数据在模块间传递时的两种方式：推（pushing）和拉（pulling）。

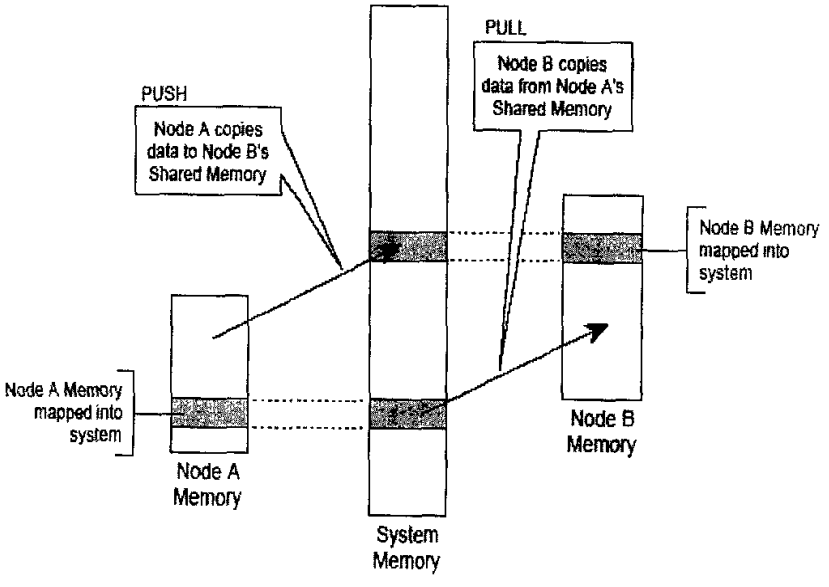

图 7 数据传递机制

1. Pushing 和 pulling 的主要区别是：pushing 是发送方主动将本地的数据送到接收方的映射的系统空间，对方直接访问自己的本地空间；pulling 是接收方将发送方的映射的系

统空间中的数据取到本地，发送方只需将数据放在其本地的共享空间就行了。

2. Node A 用 pushing 方式将存放在本地内存中的数据移到一段系统内存中，由于这段系统内存是 Node B 的本地内存映射的一段空间，所以 Node B 可以直接访问这些数据。

3. Node B 用 pulling 方式将数据从一段系统内存空间（映射到 Node A 的一段本地空间）移到本地空间。

4. 对于入队列 Inbound message queue 来说，IOP 在自己的本地内存空间分配消息空间 MFA（Message Frame Area，用于存放消息），Host 或者其他 IOP 将自己本地的数据拷贝（pushing）到这些 MFA。对于出队列 Outbound message queue 来说，Host 在系统内存空间分配 MFA，IOP 将本地数据拷贝到 MFA。在我们后续的实现中，由于只有一个 IOP（I2O Host），且只在 IOP 中分配了共享内存，在 I2O Agent 系统内存中并没有与这部分共享内存对应的内存空间，所以在后续实现中，对于 Inbound message queue 来说，I2O Host 在自己的本地内存空间分配 MFA，I2O Agent 将自己本地的数据拷贝到这些 MFA。对于 Outbound message queue 来说，I2O Agent 将本地数据拷贝到 I2O Host 在自己的本地内存空间分配 MFA（通过 ATU）。

5. 本课题实现的 I2O 通信使用的是 pushing 方式。主处理器将待发送消息拷贝到共享内存，然后通过硬件机制通知从处理器去处理；从处理器在共享内存中准备好消息后，通过硬件机制通知主处理器处理。这里提到的硬件机制可以参见第四章说明。

3.3.3 环境配置

在使用 I2O 进行消息通信前必须将 I2O 通信的基本环境准备好，所以下面我们将说明 I2O 环境初始化和相关配置。

3.3.3.1 I2O 系统初始化

I2O 环境的配置和初始化目的是提供一个供上层 OSM 调用完成配置和初始化 I2O 环境的标准接口。每个 IOP 负责其本身的初始化并为 Host 向其发送消息初始化好它的入队列。Host 负责启动系统中的所有 IOP 并将它们添加到系统配置表中，接着初始化自己的出队列，然后 Host 将为每个 IOP 提供系统中存在的所有 IOP 列表和它们的入队列的物理位置。当一个 IOP 想与另外一个 IOP 通信时，它可以将连接请求消息发送到该 IOP 的入队列中，连接请求和回复将使得两个 IOP 能够建立起直接通信的通道。

通常 Host 启动后，将下载系统所有 IOP 的可执行代码并运行，然后发送初始化消息给每个 IOP 进行下一步初始化。

3.3.3.2 IOP 初始化

IOP 初始化主要是为 DDM 和各模块间（本地 DDM 间，OSM 与各 IOP 的 DDM 间，各 IOP 的 DDM 间）的通信提供运行环境。

IOP 下载完自己的运行环境后，将下载并初始化 DDM。每个 DDM 都会有一张模块描述符表，该表描述了该 DDM 和挂在它下面的设备列表，使用这些信息 IOP 可以决定是否有必要下载并初始化该 DDM。

IOP 需要扫描本地的物理适配器并装载和初始化相应的 DDM（DDM 创建并管理 I2O 设备），建立一张逻辑配置表，这个配置表既可以扫描硬件建立，也可以 IOP 软件的配置数据结构建立。DDM 将调用 IOP 的登记函数把它所管理的设备属性和入口函数登记到 IOP 的逻辑配置表中。逻辑配置表是为了后续 IOP 间通信和 OSM 与 IOP 间通信作准备的，通过它可以查到每个 IOP 下的可用的服务。

IOP 初始化主要包括 DDM 安装和初始化，下面将分别介绍。

1. DDM 安装：与 DDM 相关的所有数据都是存放在 IOP 本地永久存储介质中，包括 DDM 可执行代码、模块描述符表和模块参数表，其中模块参数表可能在 DDM 运行过程中被修改，所以必须在随时保存更新。

2. DDM 初始化：IOP 初始化 DDM 时，将该 DDM 的模块参数表作为参数传给 DDM 初始化代码。初始化时，DDM 将自己作为设备登记到 IOP 上，在登记的同时，IOP 将为该 DDM 创建一个事件队列（event queue）并分配一个目标标识符 TID（Target Identifier，是 I/O 设备以及 DDM 在消息传递过程中的逻辑地址，它标识一个 I/O 设备或某个消息发送主体）给该 DDM，当某个消息的目的 TID 域的值等于该 TID 时，该消息将发送到这个事件队列中。IOP 将扫描扩展总线上的设备，如果某个设备登记在该 DDM 的描述符表中，IOP 将通知该 DDM 初始化这个设备并创建 I2O 设备，然后在 IOP 中登记这个 I2O 设备。其间 DDM 将给每个设备分配 TID 并在 IOP 中登记，所以最终 IOP 将知道它可以管理的所有 I2O 设备列表，这些信息全部登记在 IOP 的逻辑配置表中。

3.3.4 通信过程

IOP 初始化完成后，我们就可以利用初始化建立起来的基本环境进行 I2O 消息通信了，I2O 通信主要分为 Host 与各 IOP 间通信和各 IOP 间的通信，下面将分别介绍。

3.3.4.1 建立 IOP 间连接

主机（Host）通过发送一个消息给 IOP 来初始化 IOP 间的连接（peer connection），该消息分配一个在远端 IOP 登记的 I/O 设备的设备号给本地的 DDM，进而本地 IOP 将这个

设备挂接到本地 DDM（本讨论中也叫：用户 DDM）。

每个 I2O 设备登记时，IOP 都会给每个设备分配一个 TID，这个 TID 就成了操作这个设备的句柄(handle)。设备的 TID 只在 IOP 的本地有效，在 DDM 能够给处于另外一个 IOP 上的设备发消息前，收发设备的 TID 必须已经建立起来了，这样请求和响应才会正确的到达对方。下面是 IOP 建立连接的过程：

1. 本地 IOP 给远端需要通信的设备分配本地的 TID，本地的 DDM 使用这个 TID 来区分不同的远端设备。

2. 本地 IOP 创建带有建立连接所需信息的连接请求消息并发送给目标 IOP 的消息处理器（messenger）。该消息包含本地设备的 TID，目标 IOP 将记住这个 TID，以便回复消息时将该 TID 替换到目标地址（Target Address）域中。

3. 目标 IOP 将验证连接的合法性并回复源 IOP。

至此，IOP 间连接建立。

上面是建立连接的大概过程，总之，两个 IOP 要想能够互相通信，必须通过消息建立连接，使得双方知道通信对端的是哪个 IOP 管理下的哪个 I/O 设备，在 I2O 系统中是用 TID 来区分不同的 IOP、IOP 中不同的 DDM 和主机 OSM 的。TID 或者是全局的（IOP 可执行程序和主机的 OSM 是固定分配的 TID）或者是双方在发消息过程中带给对方的。

DDM 只认识本地 TID，因为 TID 不是整个系统唯一的（可能存在重复），IOP 发送消息时需要将源地址（Initiator Address）和目标地址（Target Address）域转换成目标 IOP 定义的 TID。也就是说，所有发送的消息中这两个域表示的是目标 IOP 定义的 TID，发送之前都需要进行转换。

下图是两个 IOP 通信时的消息通信过程：

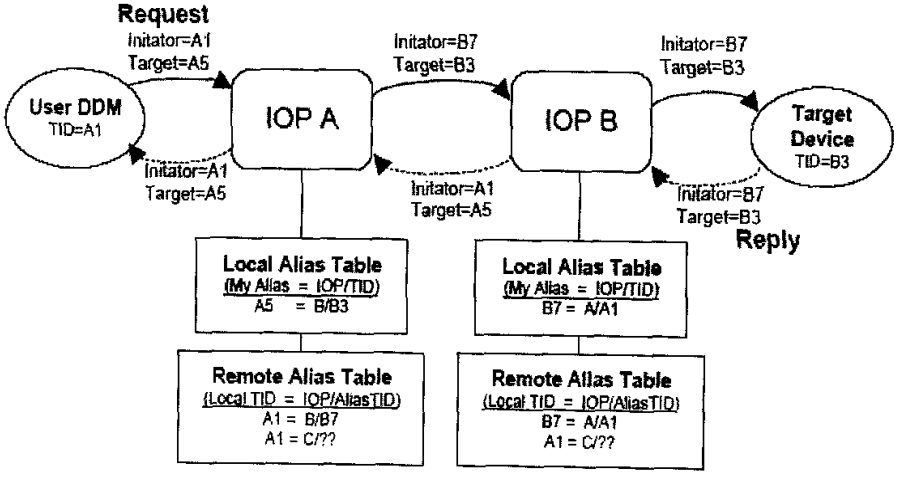

图 8 IOP 间消息通信示例

值为 0，1 的 TID 分别保留给 IOP 可执行程序（IOP executive）和主机的 OSM。因为主机 OSM 预先分配了全主机唯一的 TID，所以在 IOP 和主机间通信时别名 TID（主机上各 IOP 拥有的设备都可以分配的 TID，可以重复，所以称“别名 TID”，以前说的设备 TID 都是别名 TID；而这里 IOP 和主机的 TID 是全主机唯一的，不存在重复）就不需要了。当然处于同一个 IOP 内的不同设备间通信也就不需要别名 TID，因为它们间的通信只是 IOP 本地事务，不存在重复的 TID。

3.3.4.2 消息传递

当通信双方的 IOP 建立连接后，双方都可以收发消息，消息发送是异步非阻塞的。

1. 主机与某个 DDM 通信的过程：

a) OSM 建立请求并调用主机的消息服务，通知其想要通信的目标 IOP。

b) 主机的消息例程通过读目标 IOP 的 inbound message port 从 IOP 的空闲列表中获得一个空闲的 MFA。

c) 主机将消息放入 MFA，并通过将 MFA 地址写入 IOP 的 Inbound port 来通知 IOP 处理消息。

d) IOP 将检查请求消息的 Target Address 域，将消息传给相应的 DDM。

e) DDM 处理完后将 MFA 释放，IOP 就会将该 MFA 重新放回到空闲队列中（free list）。

f) 如果需要回复，DDM 将建立一个回复消息，将请求消息中的 initiator address、target address 和 Initiator Context 域填入回复消息中并调用本地的消息服务。

g) IOP 消息服务发现 Initiator Address 值为 001h（Host OSM）将从主机空闲 MFA 中分配一个，将消息拷贝到 MFA 指定的地址中并将消息存放地址放入自己的 outbound message queue 中。

h) 主机通过读 IOP 的 outbound message port 获得回复消息，并根据消息中的 Initiator Context 域分发消息。

i) 当 OSM 处理完回复消息并释放 MFA 后，主机消息例程将通过写 MFA 地址到 IOP 的 outbound message port 回收该 MFA 给 IOP。

2. IOP 间通信过程：

a) 源 DDM 建立请求消息并将本地 IOP 定义的别名 TID 放入 Target Address 域，再调用本地消息服务。

b) 源 IOP 将检查 Target Address 决定目标 IOP 和目标 DDM 对应的目标 IOP 实际定义的 TID，并用实际的 TID 替换当前的 Target Address。

c) 源 IOP 将查找 Initiator Address，找到目标 IOP 定义的源 DDM 对应的目标 IOP 实

际定义的别名 TID，并将它放入 Initiator Address。

d) 源 IOP 通过读目标 IOP 的 inbound message port 获得一个空闲 MFA 并将消息放入其指定的空间中，然后写 inbound message port 通知目标 IOP 处理消息。

e) 目标 IOP 将检查请求消息的 Target Address 域，将消息传给相应的 DDM。

f) 目标 DDM 释放 MFA，IOP 将其返回给空闲队列。

g) 当需要回复时，目标 DDM 将组织一个 Initiator Address、Target Address 和 Initiator Context 域和请求消息中对应域完全一样的回复消息并调用它的本地消息服务。

h) 源 IOP 将检查回复中 Initiator Address 域（不是 001h，它已经分配给主机的 OSM）。

i) 目标 IOP 检查回复消息中的 Initiator Address 域，并使用源 IOP 定义的 TID 替换 Initiator Address。

j) 目标 IOP 检查回复消息中的 Target Address 域，并使用源 IOP 定义的 TID 替换 Target Address。

k) 目标 IOP 通过读源 IOP 的 inbound message port 获得一个空闲的 MFA，将回复消息放入 MFA 指定的空间中并通过写 inbound message port 通知源 IOP 处理该消息。

l) 源 IOP 将检查回复消息的 Initiator Context 域并把该消息传给相应的 DDM。

说明：

1. Initiator Address：发送消息的 DDM 定义在接收 IOP 本地的 TID。

2. Target Address：接收消息的 DDM 定义在接收 IOP 本地的 TID。

3. Initiator Context：接收 IOP 为本地多个 DDM 定义的不同的标识符，以区分消息是发送给哪个 DDM 的。

4. 主机的操作系统发出 I/O 请求。

5. OSM 接收请求并将其转换成可寻址到目标 DDM 的消息，并激活通信层发送消息。

6. 主机消息例程通过将消息拷贝到目标 IOP 的空闲消息队列而将消息排队。

7. 目标 IOP 将消息放入对应 DDM 的事务队列（event queue）。DDM 处理该消息。

8. 处理完消息后，DDM 将建立一个回复消息，将请求消息的相关域内容拷贝到回复消息中，并调用消息服务发送。

9. IOP 消息服务将回复消息拷贝到主机消息例程的队列 buffer（也就是 IOP 的 outbound queue）中排队等待处理。

10. IOP 通知主机消息例程准备处理，驱动将消息传给 OS 处理。

第四章 I2O 通信实现

上面分别介绍了 PCI 和 I2O 通信的规范，规范是一个框架，由于两个规范在制定时都考虑了可扩展性和适应性问题，所以也就给厂商相应的自主设计空间。本章介绍的 I2O 通信实现来源于本人在实习期间所做的某型号路由器项目（考虑到保密，本文中将不提及该路由器具体型号），是基于具体厂商产品之上的驱动实现，因而可能与规范内容有一些不一致，但是这是在规范允许范围内的。

4.1 本课题情况介绍

随着 Internet 业务迅速发展，网络数据流量急剧增加，骨干层和汇聚层的路由器的转发性能成为网络业务发展的瓶颈。因此，转发性能和可管理性是路由器能否适应市场需求的关键因素。该型号路由器市场定位主要针对城域网、金融网、政府网、军网，以填充该公司路由器产品线提供接入和边缘层的产品空白，可覆盖 Cisco75xx 系列以下的大部分路由器组网应用，提供各类业务和协议实现平台。为了能在成本允许前提下尽量提高系统的性能，该型号路由器采用了双 MIPS CPU 架构，其中一个 MIPS 处理器称为主处理器（Main Processor，简称 MP），它是系统的控制管理和协议处理的核心；另一个 MIPS 处理器称为桥处理器（Bridge Processor，简称 BP），作为主处理器的代理，完成主处理器对桥处理器侧设备和资源的控制和管理，它们之间通过桥芯片 GT64120 上自带的 PCI 总线进行通信。由于在路由器运行中有大量控制管理信息在 BP、MP 间传递，所以 MP、BP 间的可靠通信将非常重要。

下文主要说明此项目中使用的基于 PCI 总线的 I2O 通信的原理，设计与实现。

4.1.1 本项目的总体情况

为了方便后续内容展开，先介绍该型号路由器产品的总体情况。下图是该产品机架的主控板硬件结构图：

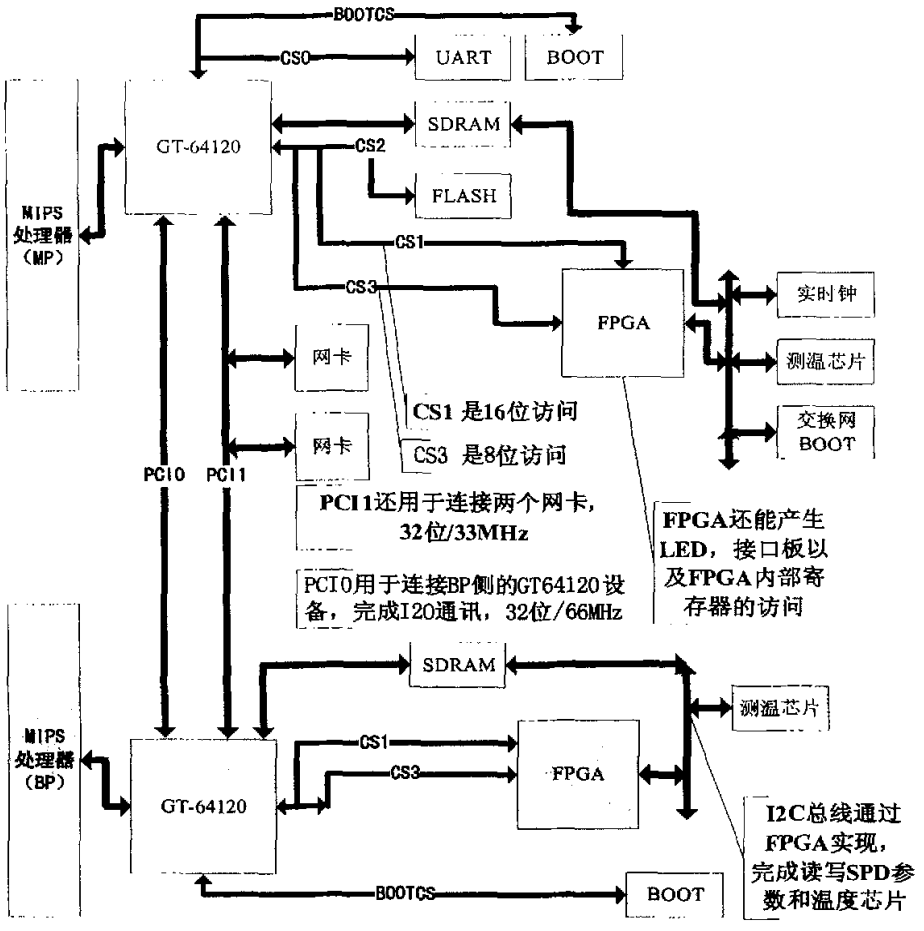

图 9 系统主控板硬件结构图

为了保证系统最多可以支持 4 个网络处理模块，采用两套高速 MIPS 处理器 RM7065A（PMC 公司）实现中央处理系统。RM7065A 内部主频为 400MHz，内嵌 16Kbyte 的指令 Cache、16Kbyte 数据 Cach 及 256Kbyte 的二级 Cache。MIPS CPU 通过主机桥 GT64120A 与其它外部设备（如 SDRAM、Boot ROM 等）相连，两套 MIPS CPU 之间的通信也通过 GT64120A 的 PCI 接口实现。两个 CPU 通过 PCI 组成一个对称双 CPU 处理系统，运行协议处理软件和系统控制管理软件；其中主处理器 MP 是系统的控制管理和协议处理的核心；桥处理器 BP 作为主处理器的桥，代理主处理器控制管理相应的模块。另外，选择 MIPS CPU 作为系统中央处理器的一个主要原因是 MIPS 总线与本系统选择的网络处理器模块之间可以无缝连接，大大降低系统的硬件成本和开发难度。

作为 CPU 桥芯片的 GT64120 是专门与 MIPS 处理器配合使用的控制器芯片，可以支持各种 64bits MIPS 总线接口，同时内嵌了 SDRAM 控制器、DMA 控制器及两个 32bit 或一个 64bit PCI 接口。两个 MIPS 控制器通信时，即可通过主 PCI 接口的 I2O 单元操作。

系统提供两条 PCI 总线，一条 PCI 总线上连接两个处理器的主机桥用作处理器之间的通信（这就是本文将要介绍和实现的部分）；另一条 PCI 总线连接两个处理器的主机桥以及两个 PCI 网卡，两个网卡分别实现主备中央处理系统之间的通信和系统的外部专用网管接口，其中主备中央控制系统通信的网卡可以使用本公司产品线申请的专利技术在两个处理器之间动态切换；系统正常运行时主备通信网卡被中央处理系统的 MP 使用，在主备中央控制系统之间交换数据。系统开发人员可以使用网卡切换命令将其切换给 BP 使用，通过网卡调试 BP 上的软件。

中央处理系统可以配置 128M/256M/512M 字节的内存，512K 字节的 BOOTROM，64M 字节的 FLASH 内存；提供 4 个网络处理模块接口。

4.2 通信模块描述

处理器（MP/BP）间I2O通信模块是通过基于PCI的I2O通信机制实现的。这里的I2O通信机制主要依靠I2O消息单元实现。I2O消息单元是为多处理器间通信提供的一种硬件技术，通过I2O进行通信可以在只使用I2O Host的内存而不使用I2O Agent的内存资源的前提下，保证通信的效率和可靠性。

4.2.1 I2O 通信原理

本模块是为该型号路由器主控板的双 MIPS CPU 通信所设计的。通信的硬件基础由 CPU 所连接的系统控制器 GT64120 支持的基于 PCI 的 I2O 通信机制提供。两个 MIPS CPU 分别命名为 MP（主控处理器）和 BP（桥接处理器）。

MP 作为 I2O 通信的 I2O Agent，通过 Inbound Queue 向 MIPS BP 发消息，通过 Outbound Queue 接收来自 MIPS BP 的消息。MIPS BP 作为 I2O 通信的 I2O Host，需要从自己内存空间中划分出一个固定的部分用于存放 Inbound Messages（发自 I2O Agent 的消息）与 Outbound Messages（发向 I2O Agent 的消息）。通过操作 I2O 中断以及 I2O 寄存器进行通信。

在本课题实现中，I2O Host 侧和 I2OAgent 内存具体分配情况如下（总体是静态分配，但是各段内存内部是动态分配的）：

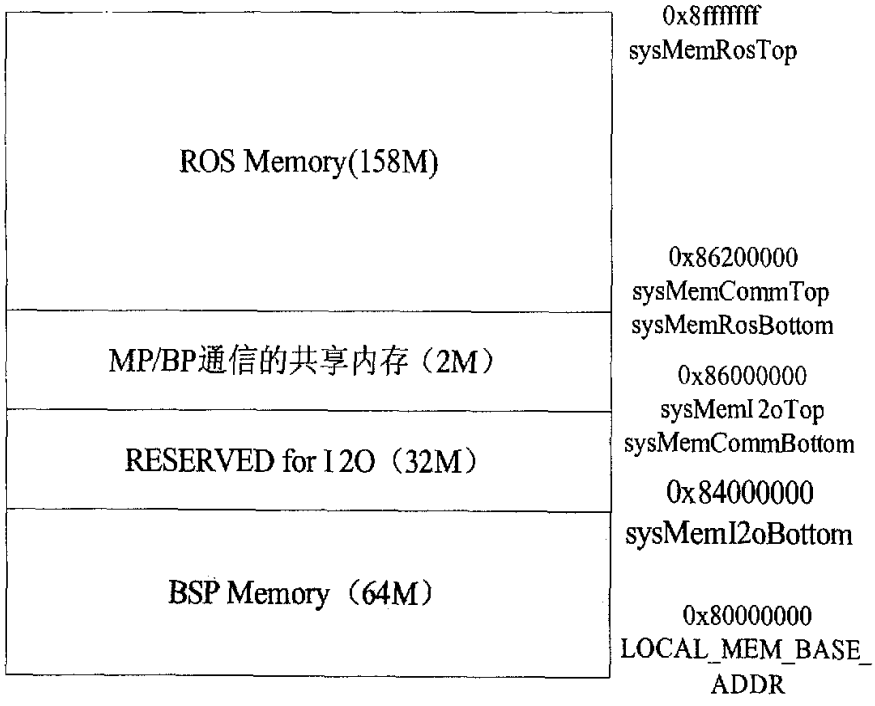

图 10 BP 侧内存空间分配

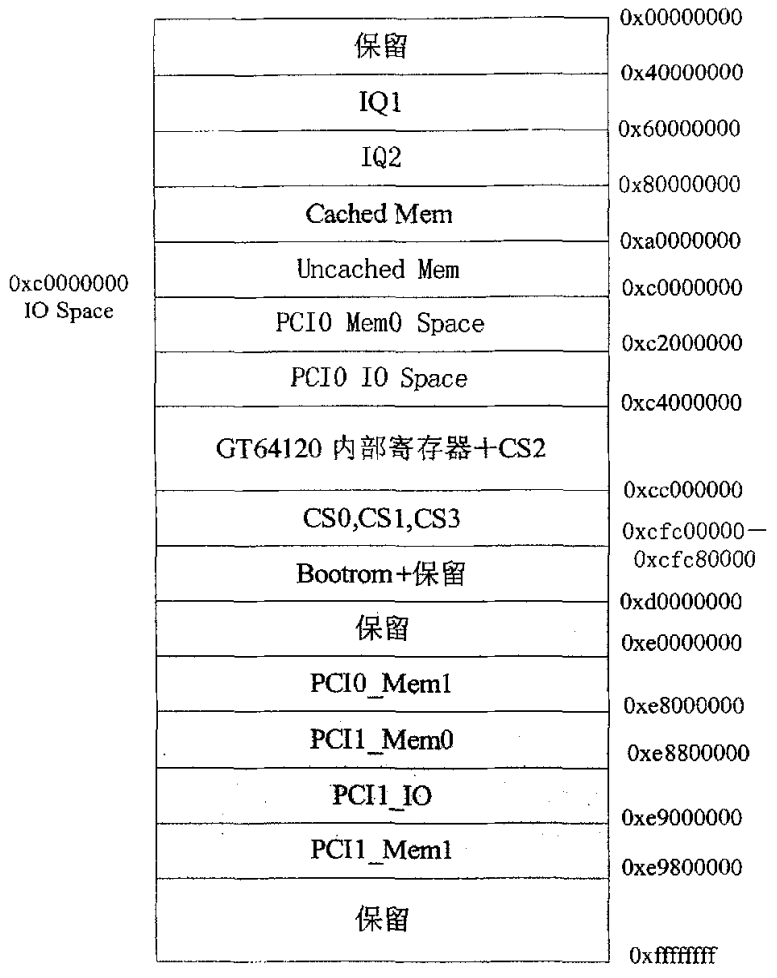

图 11　MP 侧内存空间分配

另外由于 GT64120 的 I2O 有这样的访问要求：PCI Agent 访问 BP GT64120 的 I2O 寄存器的基地址是由 PCI0 SCS[1:0]基地址寄存器所指定的，该空间前 4K 被定向到 I2O 寄存器，后面的才是 SDRAM。

由于上述原因，MIPS MP 为了访问 I2O 寄存器以及 I2O 的 MFA，需要将这部分空间映射到 MP 可访问的虚拟地址空间内。所以在 MIPS MP 建立 TLB（Translation Lookaside Buffer，虚拟地址到物理地址快速转换表缓存）表时，还应考虑这部分空间。设计上将这部分空间定义到 PCI0 Memory1 上，大小为 128Mbytes（其中 I2O MFA 空间为 0xE4000000～0xE6000000，大小为 32Mbytes）。I2O 保留内存大小包括 4 个 I2O 队列以及所有的 MFA。同时将 PCI0 SCS[1:0]基地址寄存器值设置为 PCI0 Memory1 的基地址。

注：

1. I2O Host：在本课题实现中指的是MIPS BP 侧处理器。
2. I2O Agent：在本课题实现中指的是MIPS MP侧处理器。
3. BSP Memory：是保留给Vxworks的代码段、BSS段、数据段和可分配内存段。
4. MP/BP 通信共享内存：主要用来在线升级BP侧BOOTROM时存放BP侧BOOTROM代码。
5. ROS Memory：是提供本项目的操作系统使用的，主要是为各种应用提供可以动态申请和释放的内存空间。
6. RESERVED for I2O：就是提供给I2O通信时存放消息（MF：message frame）的。

4.2.1.1 I2O 寄存器

GT64120 为实现 I2O 提供了下面的寄存器（见图 12）。这些寄存器的访问方式为：PCI 方（MP 通过 PCI 访问，I2O Agent）通过 PCI0 Memory1 空间基地址寄存器指定的前 4k 字节空间进行访问。CPU（BP 侧 CPU，I2O Host）通过 CPU internal space base 寄存器的偏移量进行访问。I2O 是通过硬件使能打开的，如果 I2O 被关闭，从 PCI 方相应地址处读出的为 I2O Host SDRAM 中的内容。

GT64120 提供的 I2O 寄存器访问方式在 MIPS 下增加了编程的复杂度，I2O Agent 需通过 TLB 表将 I2O Host 的 SDRAM 映射到可访问的虚拟地址。而 I2O Host 也需要将 I2O 的空间建立在 SDRAM 的非 Cache 虚拟地址下。

I_2O Register	PCI SIDE[1]	CPU Side[2]
Inbound Message Register 0	0x10	0x1c10
Inbound Message Register 1	0x14	0x1c14
Outbound Message Register 0	0x18	0x1c18
Outbound Message Register 1	0x1c	0x1c1c
Inbound Doorbell Register	0x20	0x1c20
Inbound Interrupt Cause Register	0x24	0x1c24
Inbound Interrupt Mask Register	0x28	0x1c28
Outbound Doorbell Register	0x2c	0x1c2c
Outbound Interrupt Cause Register	0x30	0x1c30
Outbound Interrupt Mask Register	0x34	0x1c34
Inbound Queue Port Virtual Register	0x40	0x1c40
Outbound Queue Port Virtual Register	0x44	0x1c44
Queue Control Register	0x50	0x1c50
Queue Base Address Register	0x54	0x1c54
Inbound Free Head Pointer Register	0x60	0x1c60
Inbound Free Tail Pointer Register	0x64	0x1c64
Inbound Post Head Pointer Register	0x68	0x1c68
Inbound Post Tail Pointer Register	0x6c	0x1c6c
Outbound Free Head Pointer Register	0x70	0x1c70
Outbound Free Tail Pointer Register	0x74	0x1c74
Outbound Post Head Pointer Register	0x78	0x1c78
Outbound Post Tail Pointer Register	0x7c	0x1c7c
Reserved	0x80 to 4K	-

图 12 I2O 寄存器列表

GT64120 提供的寄存器按功能可分为如下几类：

消息寄存器

消息寄存器（message register）接收和发送短消息，不需要通过内存来传递数据。当进行写操作时，消息寄存器会向 PCI 总线触发一次中断，也就是可以将中断线直接连到 MIPS CPU 上或者是连接到 GT 6412，0 从而间接产生中断。有两种类型的 message registers：

1. 入队列消息寄存器（Inbound messages register）：由 I2O Agent 通过 GT64120 向 I2O Host 的 GT64120 发送消息。

2. 出队列消息寄存器（Outbound messages register）：由 I2O Host GT64120 向 I2O Agent GT64120 发送消息。

入队列消息（Inbound message）的中断状态记录在 Inbound Interrupt Cause 寄存器中。

出队列消息（Outbound message）的中断状态记录在 Outbound Interrupt Cause 寄存器中。

GT64120 有两个入队列消息寄存器 IMR（Inbound Message registers0, Inbound Message registers1）。当 I2O Agent 完成对 IMR 写操作时，便会在 I2O Host 的入队列中断状态寄存器 IISR （Inbound Interrupt Status register）中产生一个可屏蔽的中断请求。如果中断未被屏蔽，那么 I2O Host CPU 便会收到一个中断请求。CPU 可以通过向 IISR 的入消息中断位写 1 来清该中断。中断可以通过入队列中断屏蔽寄存器 IIMR（Inbound Interrupt Mask register）相应的屏蔽位屏蔽。

GT64120 有两个出队列消息寄存器 OMR（Outbound Message registers0,Outbound Message registers1）。当 I2O Host 完成对 OMR 写操作时，便会在出队列消息中断状态寄存器 OISR （Outbound Interrupt Status register）中产生一个可屏蔽的中断。如果中断未被屏蔽 ，I2O Agent 便会收到一个中断请求。该中断可以由 I2O Agent 向 OISR 的出消息中断位写入 1 来清除。该中断还可以通过出队列中断屏蔽寄存器 OIMR（Outbound Interrupt Mask register）中相应的屏蔽位屏蔽。

Doorbell 寄存器

GT64120 使用 Doorbell 寄存器在 PCI 和 CPU 总线上产生中断请求。共有两种类型的 doorbell 寄存器。

1. Inbound doorbells：由 I2O Agent 向 I2O Host 请求中断服务。
2. Outbound doorbells：由 I2O Host 向 I2O Agent 请求中断服务。

I2O Host 处理器通过设置出门铃寄存器 ODR（Outbound Doorbell register）产生一个中断请求。该中断可以通过设置出队列中断屏蔽寄存器 OIMR（Outbound Interrupt Mask register）屏蔽，但是屏蔽该中断并不能阻止 ODR 相应的中断位设置。I2O Agent 通过向 ODR 相应位写 1 来清除该中断。

I2O Agent 总线可以通过设置入队列门铃寄存器 IDR（Inbound Doorbell register）向 I2O Host 产生一个中断请求。中断可以通过设置入队列中断屏蔽寄存器 IIMR（Inbound Interrupt Mask register）屏蔽，但屏蔽该中断不能阻止 IDR 的中断请求置位。I2O Host 通过向 IDR 相应位写 1 来清除该中断。

环形队列操作寄存器

环形队列是 I2O 消息传递机制的核心部分，也是消息 MU（Messaging Unit）功能最强

的部分。在 MU 中共有四个环形队列：入空闲队列（Inbound Free queue）、入消息队列（Inbound Post queue）、出空闲队列（Outbound Free queue）和出消息队列（Outbound Post queue）。所以相应的有管理这四个队列的寄存器（详细定义可以参见上文的 I2O 寄存器定义），每个队列分别有一个头指针（Head Pointer）和一个尾指针（Tail Pointer）寄存器，分别管理这个队列的头尾指针。

在本课题实现中，针对这几个寄存器的操作方式，I2O Host 直接通过桥的 CPU 内部寄存器基地址（CPU Internal Space Base） + 寄存器偏移的方式访问，而 I2O Agent 是通过入队列虚端口寄存器（Inbound Queue Port Virtual Register）和出队列虚端口寄存器（Outbound Queue Port Virtual Register）分别访问入空闲队列、入消息队列、出空闲队列和出消息队列。

I2O Agent 对 Inbound Queue Port Virtual Register 的读操作最终转换成对 Inbound Free Queue Tail Pointer 的读操作，对 Inbound Queue Port Virtual Register 的写操作最终转换成对 Inbound Post Queue Head Pointer 的写操作。

I2O Agent 对 Outbound Queue Port Virtual Register 的读操作最终转换成对 Outbound Post Queue Tail Pointer 的读操作，对 Outbound Queue Port Virtual Register 的写操作最终转换成对 Outbound Free Queue Head Pointer 的写操作。

4.2.1.2 I2O 环形队列

消息的传递有两条路径：一是 Inbound queue 用于 I2O Host 接收从 I2O Agent 发过来的消息，另一条 Outbound queue 是用于 I2O Host 向 I2O Agent 发送消息。每一个队列都由一对 FIFO 实现。Inbound queue 和 Outbound queue 都是由 free FIFO 和 post FIFO 组成。

两个入队列分别为：Inbound Posts message queue 和 Inbound Free message queue。Inbound Posts message queue 用来存放 I2O Agent 发送给 I2O Host 处理的消息。而 Inbound Free message queue 用来存放 I2O Host 处理完之后的消息（由 I2O Agent 向 I2O Host 发送的消息）。

两个入队列的用法为： I2O Agent 向 I2O Host 的 Inbound post message queue 队列发送 Inbound messages，而 I2O Host 处理之后返回释放的消息存在 Inbound free message queue 队列。处理过程如下：

1. I2O Agent 发送一个 Inbound message。
2. I2O Host 接收并处理该消息。
3. 处理完成后，I2O Host 将该消息返回到 Inbound free queue 中，通知 I2O Agent 已经完成消息处理。

两个 Outbound message queue 的用法为： I2O Host 向 I2O Agent 的 Outbound post message queue 队列发送 Outbound messages，而 I2O Agent 处理之后返回释放的消息存在 Outbound free message queue 队列。处理过程如下：

1. I2O Host 发送一个 Outbound message。
2. I2O Agent 接收并处理该消息。
3. 处理完成后，I2O Agent 将该消息返回到 Outbound free queue 中，通知 I2O Host 已经完成消息处理。

可以看出 Outbound message queue 消息处理过程基本上是入队列消息处理的逆过程。

环形队列位于 SDRAM 中分配的数据存储区内。每个队列项为 32-bit 长。环形队列的空间大小为 4K（16K bytes）～256K 项（1024bytes）,也就是内存分配大小为 64Kbytes ～1Mbytes。四个队列空间大小必须相同，并且必须连续。队列基地址和空间大小分别通过队列基地址寄存器 QBAR（Queue Base Address register）和队列控制寄存器（Queue Control register）设置。

每个队列基地址都是基于 QBAR 和队列空间大小，具体如下表：

Queue	Starting Address
Inbound Free	QBAR
Inbound Post	QBAR + Queue Size
Outbound Post	QBAR + 2*Queue Size
Outbound Free	QBAR + 3*Queue Size

图 13 环形队列基地址定义

以上四个队列的每个队列都有一个头指针（head pointer）和一个尾指针（tail pointer）进行消息空间的定位，这些指针的值是相对于 QBAR 的偏移且都存放在 GT64120 的内部寄存器中。对队列的写操作使用的是头指针，对队列的读操作使用的是尾指针，头尾指针是由软件或硬件自动更新的，具体可以参考下面章节的说明。I2O 消息传递可以用下面的图示说明：

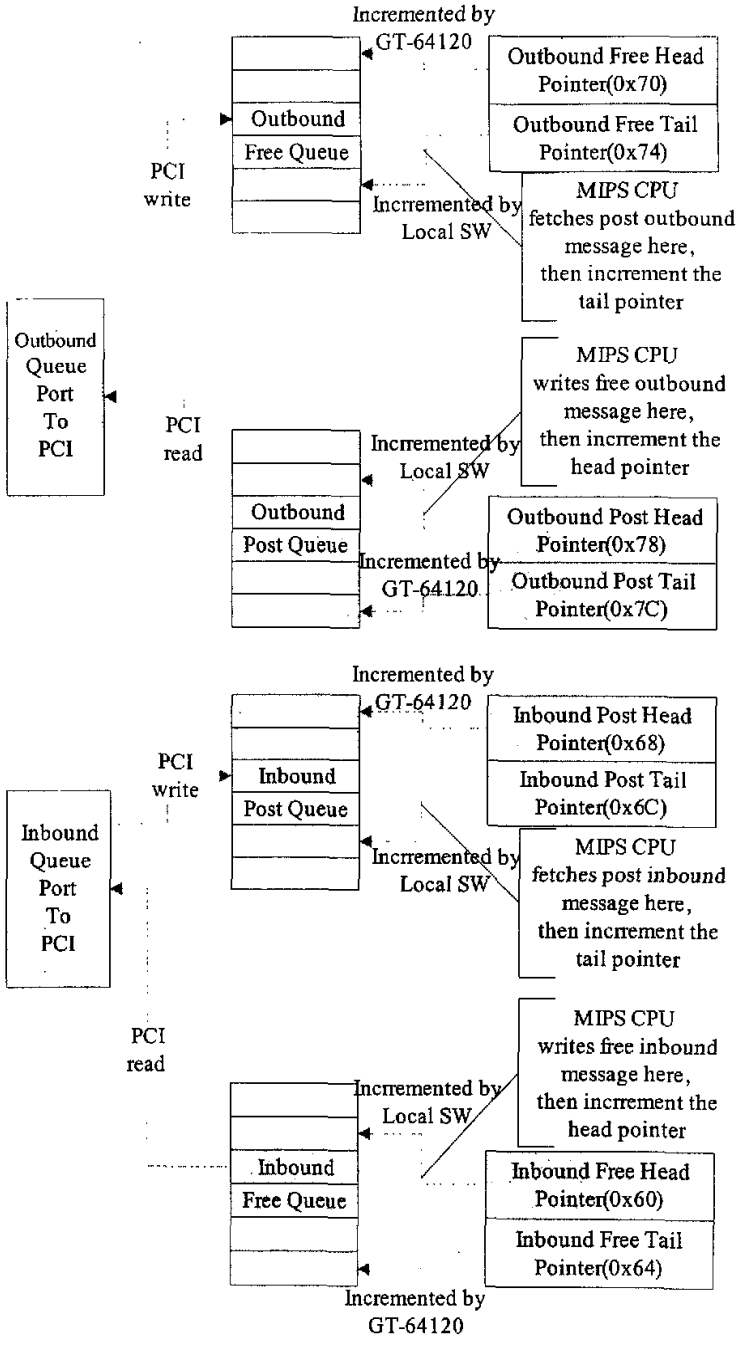

图 14 环形队列示意图

Inbound Queue

GT64120 I2O Circular Queue 有两个入队列（Inbound Queue），分别是入消息队列（Inbound Post Queue）和入空闲队列（Inbound Free Queue）。

Inbound Free Queue 中存放可供 I2O Agent 使用的 Inbound free messages。I2O Host 将释放的 free messages 放在队列头部，而 I2O Agent 则从队列尾部取出 free messages。队列的头指针由 I2O Host 的软件来更新，而尾指针是在 I2O Agent 取出一个新的 free message 时由 GT64120 硬件自动更新（如果不发生错误时，错误情况见下面说明）。

由于硬件实现上的原因，I2O Agent 访问 Inbound queue 和 Outbound queue 分别是通过 Inbound Queue Port virtual register 和 Outbound Queue Port virtual register 间接访问的。通过 Inbound Queue Port virtual register，I2O Agent 向 Inbound Queue Port 发出读操作时：

1. 如果 Inbound Free Queue 不空（可以通过头指针和尾指针是否相等判别），那么 QBAR + Inbound Free Tail pointer 指向的数据返回。

2. 如果队列为空（头尾指针相同），那么将返回 0xFFFFFFFF（错误条件）。

Inbound Post Queue 存放 I2O Agent 发送给 I2O Host 的消息。I2O Host 从该队列的尾部取出消息进行处理。I2O Agent 发送的消息置于队列的头部。尾指针由 I2O Host 软件更新。发送一条新的消息后，头指针由 GT64120 自动更新。如果 Inbound Post Queue 满，则 I2O Agent 想要继续往里面写时，将导致 PCI 的 RETRY 操作，直到有队列有空间为止。

向 IQP（Inbound Queue Port）做的 PCI 写入会传递到 QBAR + Inbound Post 头指针所指向的位置。写操作完成后，GT64120 将 Inbound Post Head Pointer 加 4 字节（1 个字）；头指针将指向下一个可用的 Inbound 消息指针。同时，还会向 I2O Host 发送一个中断表示新的消息需要处理。

Outbound Queue

GT64120 I2O Circular Queue 有两个出队列（Outbound Queue），分别是出消息队列（Outbound Post Queue）和出空闲队列（Outbound Free Queue）。

Outbound Post Queue 用于 I2O Host 向 I2O Agent 发送消息。当 I2O Agent 向 Outbound Queue Port 发出读操作时，可能出现如下情况：

1. 如果 Outbound Post Queue 不空（可以通过头指针和尾指针是否相等判别），那么 QBAR +Outbound Pos Tail pointer 指向的数据返回。

2. 如果队列为空（头尾指针相同），那么将返回 0xFFFFFFFF。

Outbound Free Queue 存放 I2O Agent 发送给 I2O Host 的处理完消息。I2O Host 从该队列的尾部取出消息进行处理。I2O Agent 发送的消息置于队列的头部。尾指针由 I2O Host 软件更新。发送一条新的消息后，头指针由 GT64120 自动更新。如果 Outbound Free Queue 满，且 I2O Agent 继续往里面写时，将导致 PCI 的 RETRY 操作，直到有队列有空间为止。

4.2.1.3 本课题 Circular Queue 设计实现

根据 I2O 规范并结合本系统的特点，将环形队列设计如下：

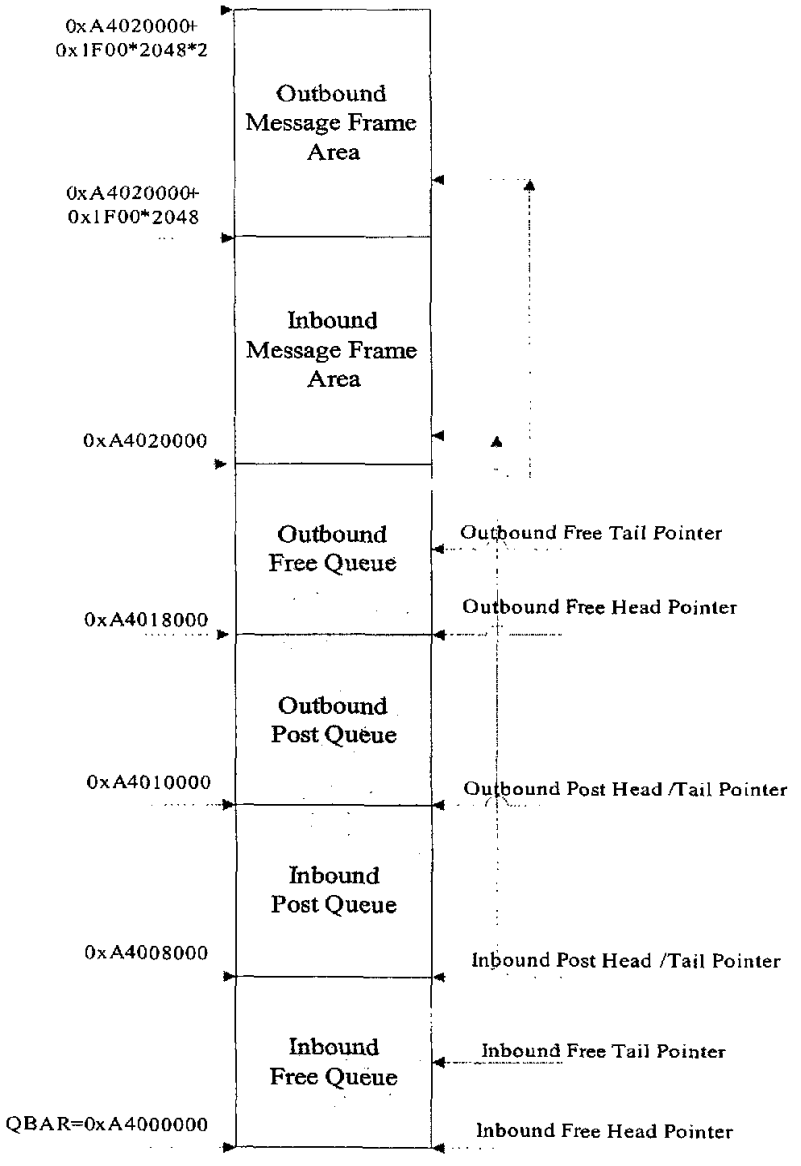

图 15 环形队列空间分配

我们在 I2O 初始化时完成以上环形队列空间和 MFA（Message Frame Area）空间的初始化，具体步骤如下：

1. 初始化环形队列的基地址和每个队列空间的大小，并将如上图所示的各队列的指针写入相应的寄存器中。

2. 将 MFA 空间划分成大小为 2048bytes 的 0x3E00 个 buffer 分别作为 Inbound 和

Outbound 消息空间，并初始化 Inbound/Outbound Free Queue Tail Pointer 所指空间，方便后续发送消息。

注意：我们为每个环形队列分配 32Kbytes 空间（也就是 8K 条指针），通常情况下，我们需要分别分配 8K 个 MFA 给 Inbound queue 和 Outbound queue。但是根据我们产品空间分配情况和 MP、BP 通信特点，在空间分配时并没有将分配 8K 个 MFA 给 Inbound queue 和 Outbound queue，只是分别分配了 0x1F00 个 MFA。之所以分配 0x1F00 个 MFA，首先是由于我们根据系统空间分配，只分配了 32M 空间作为 I2O 通信空间，但是如果分配 8K 个 MFA 的话，将超出 32M 空间，加上对于分配 0x1F00 个 MFA 时 I2O 通信带宽和可靠性的测试，得出这样分配是可以满足我们系统要求的。

4.3 通信模块设计

4.3.1 运行环境

本模块设计基于 WindRiver 的 Tornado 集成开发环境，采用 C 语言开发实现。本模块属于 VxWorks 的 BSP（Board Support Package，板级支持包）范畴，但是最终将编译链接进 VxWorks 映象中运行。

由于本模块的实现是基于 VxWorks 操作系统，所以下面的很多内容涉及到具体 VxWorks 操作系统的一些特性，我们用到的特性在此加以说明：

1. 信号量：是进程间通信的重要机制，是互斥和同步的主要手段，在 VxWorks 中提供二进制信号量、互斥信号量和计数信号量，下面我们着重介绍前两种：

1) 二进制信号量：是 VxWorks 中最快最通用的信号量，适用于进程间同步（此时信号量可作为任务等待的一个状态或事件，本文中 semIn 和 semOut 就是这样使用的）和互斥。二进制需要的系统开销最小，因而特别适用于高性能的场合。关于二进制信号量的创建、使用、删除等内容可参见 VxWorks 相关文档。

2) 互斥信号量：是一种解决内在互斥问题的特殊的二进制信号量，它包括优先级翻转、删除安全等高级特性。互斥信号量与二进制信号量的主要区别在于它仅用于互斥，它仅能由获取（semTake）它的任务释放，而且它不能在中断处理程序中释放。所以下文中信号量 semApp 定义为互斥信号量。

2. 中断处理程序和任务：在 VxWorks 系统中，为了提高中断处理速度，所有的中断处理程序共用一个独立于其它任务的上下文和中断堆栈，且由于中断处理优先级高，所以在编写中断处理程序时首先要求中断处理程序尽量简短（如果确实需要做大量处理工作可以交由任务级进程来做），其次要求不使用导致调用者阻塞的函数（比如：printf，malloc

和 semTake 等)。下文叙述的 I2O 通信实现中，我们是通过信号量在中断处理程序和消息处理任务间通信的，此时信号量作为消息处理任务等待的一个事件。

4.3.2 I2O 通信设计实现

本课题的 I2O 通信实现中，主要包括 I2O 通信初始化、消息发送、消息接收处理进程相关代码的实现。

4.3.2.1 I2O 通信初始化

I2O 通信初始化是在真正进行 I2O 通信前做的一些准备工作，主要是搭建 I2O 通信所需的基本环境。

I2O Host 的 I2O 相关初始化包括：

1. 关闭所有中断（包括 Doorbell，message register，message queue 等），这是由于此时整个 I2O 子系统还没有初始化，不能正常处理这些中断。

2. 在内存中为 Message Frame 以及 4 个 Queue 分配空间(Message Frame 必须 32 位对齐，队列的起始地址以 1M 字节对齐)。

3. 初始化 Inbound Free Queue 和 Outbound Free Queue 的各项为相应的 MFA。

4. 初始化 I2O 中断服务程序、任务以及相关的信号量。

5. 使能 I2O。

6. 打开 I2O 的 Inbound Post Queue 中断和 Doorbell 中断。

流程如下图：

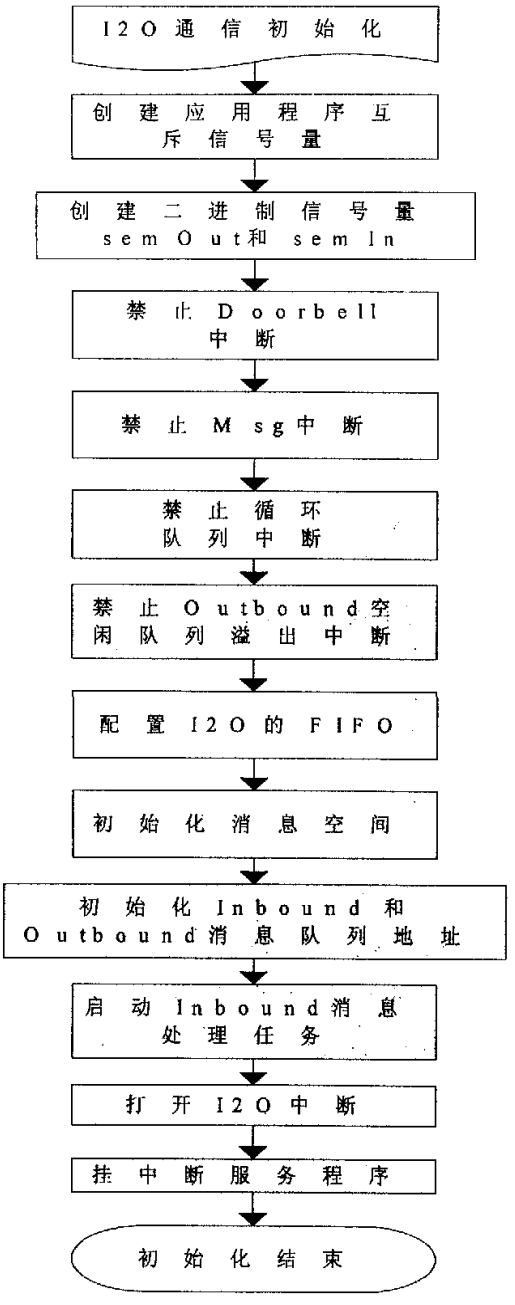

图 16 I2O Host I2O 通信初始化流程图

I2O Agent 的 I2O 初始化包括：

1. 初始化 I2O 中断服务程序、任务以及相关的信号量。
2. 打开 I2O 的 Inbound Post Queue 中断和 Doorbell 中断。

流程如下图：

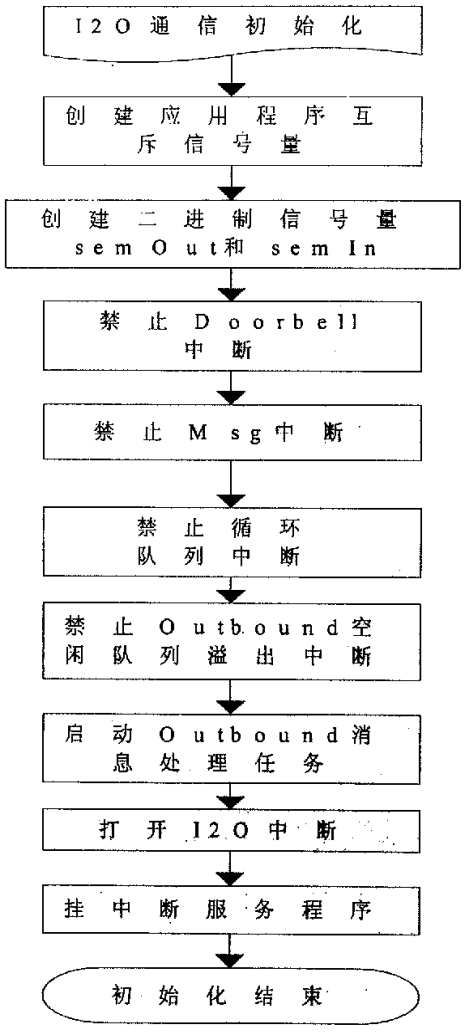

图 17 I2O Agent I2O 通信初始化流程图

4.3.2.2 MP 消息发送实现说明

为了防止多任务发送导致 I2O 通信异常，建立一个任务信号量 semApp。另外，当一方发送完消息后为了确保消息发送成功，建立了一个通知接收消息的信号量 semIn，当一方处理完一条消息后，通过 semOut 通知对方有空闲空间。具体发送过程如下：

1. 发送程序首先获取该任务信号量。如果获取成功，说明此时没有其他进程使用 I2O 发消息，则进入发送过程。注意此处针对不同的消息类型，获取信号量时会有不同的处理过程，是为了各种消息的不同要求，详见消息发送章节叙述。

2. 使能消息寄存器中断（message register interrupt），防止在发送消息时获取不到空闲 MFA（Message Frame Area），此时针对高优先级消息，本地将等待对方释放 MFA，而对方

释放 MFA 后是通过 message register interrupt 通知本地已经有空闲 MFA 使用，详见消息发送章节叙述

3. 从 Inbound Free queue 中取得一个空 MFA 的指针。会导致三种结果：获取成功、队列空以及 I2O 不正常。

4. 如果 I2O 不正常，则释放任务信号量，返回消息发送错误。（这一点如果 I2O 初始化正确，将不会出现）

5. 如果获取成功，则将待发送的消息拷贝到该指针所指向的 MFA 中，并通过将该指针写到 Inbound Post queue 中，通知对方有消息需要处理（这里包括将指针地址转换为 PCI Memory 空间地址并加以判断）。

6. 如果 Inbound Free queue 为空，则根据不同的消息类型进行处理。如果是高优先级消息，我们将一直获取 Free queue 不空的信号量（由对方在处理完一条消息后，触发一次 message interrupt，中断服务程序读出中断状态寄存器值为 message interrupt 时释放该信号量（semOut））；如果是普通消息，我们将释放 semApp 并返回，即不再继续发送该消息。如果获取仍旧不正常，说明 I2O 不正常，释放任务信号量 semApp 后，返回发送消息错。

4.3.2.3 BP 消息发送实现说明

BP 和 MP 消息发送过程类似，主要不同在于 MP 侧在获取 MFA 后需要对它的正确性进行判断，防止地址没有正确转换。

和 MP 一样，BP 为了防止多任务发送导致 I2O 通信异常，也建立一个任务信号量 semApp。另外，当一方发送完消息后为了确保消息发送成功，建立了一个通知接收消息的信号量 semIn，当一方处理完一条消息后，通过 semOut 通知对方有空闲空间。具体发送过程如下：

1. 发送程序首先获取该任务信号量 semApp。如果获取成功，说明此时没有其他进程使用 I2O 发消息，则进入发送过程。注意此处针对不同的消息类型，获取信号量时会有不同的处理过程，是为了各种消息的不同要求，详见消息发送章节叙述。

2. 使能 message register interrupt，防止在发送消息时获取不到空闲 MFA，此时针对高优先级消息，本地将等待对方释放 MFA，而对方释放 MFA 后是通过 message register interrupt 通知本地已经有空闲 MFA 使用，详见消息发送章节叙述

3. 从 Outbound Free queue 中取得一个空 MFA 的指针。会导致三种结果：获取成功、队列空以及 I2O 不正常。

4. 如果 I2O 不正常，则释放任务信号量，返回消息发送错误。（这一点如果 I2O 初始化正确，将不会出现）

5. 如果获取成功，则将待发送的消息拷贝到该指针所指向的 MFA 中，并通过将该指针写到 Outbound Post queue 中，通知对方有消息需要处理。

6. 如果 Outbound Free queue 为空，则根据不同的消息类型进行处理。如果是高优先级消息，我们将一直获取 Free queue 不空的信号量（由对方在处理完一条消息后，触发一次 message interrupt ，中断服务程序读出中断状态寄存器值为 message interrupt 时释放该信号量（semOut））；如果是普通消息，我们将释放 semApp 并返回，即不再继续发送该消息。如果获取仍旧不正常，说明 I2O 不正常，释放任务信号量后，返回发送消息错。

I2O Host 和 I2O Agent 发送消息的公共流程图如下：

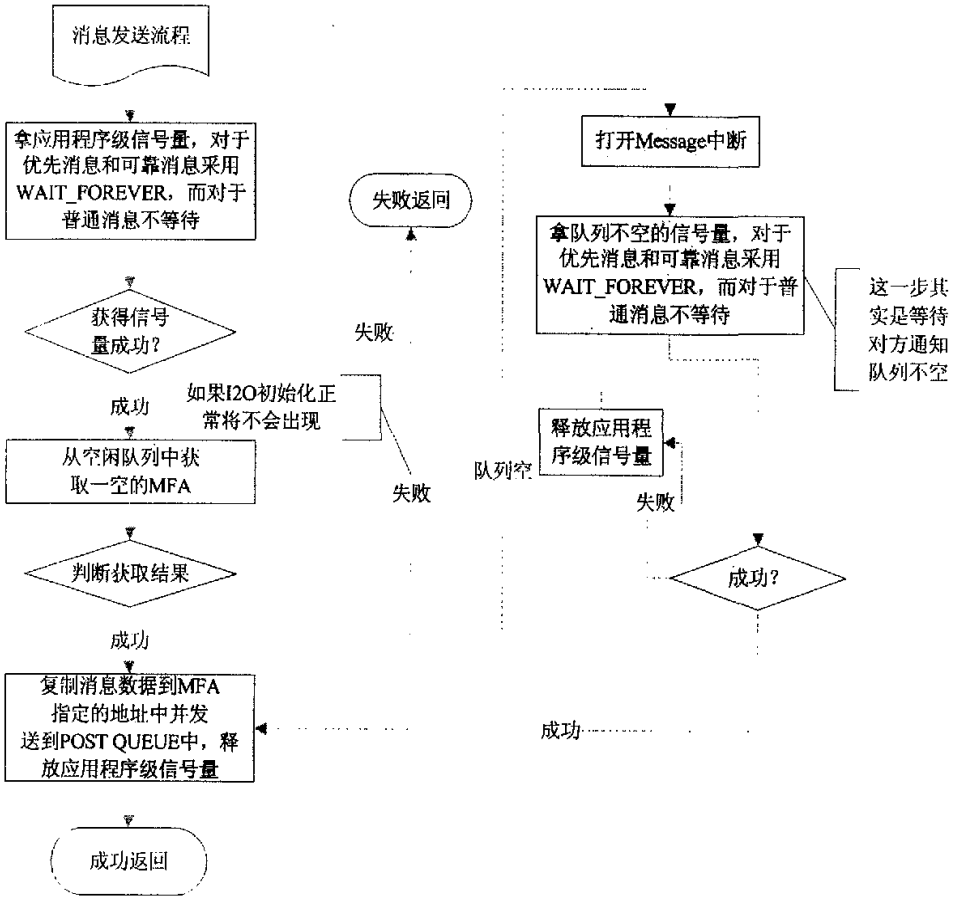

图 18 发送消息的公共流程图

4.3.2.4 MP 消息接收实现说明

I2O 消息的接收涉及到 I2O 接收中断处理程序以及为了优化消息处理生成的 I2O 接收任务。

MIPS MP（I2O Agent）I2O 中断处理程序的处理过程如下：

读取出队列中断状态寄存器（Outbound Interrupt Cause Register）和出队列中断屏蔽寄存器（Outbound Interrupt Mask Register），并从中获取当前中断类型。有两种类型的中断需要处理：

1. message register interrupt，由于我们不用 Message register 收发信息，所以如果产生该类中断，表示 BP 侧成功处理了一条消息（表明有空闲的 MFA 供 MP 分配使用），发送队列肯定未满，所以释放信号量通知当前因未拿到该信号量而等待（PEND）的进程继续执行下去，后续过程即是 MP 的发送消息过程。

2. Outbound post queue 中断，通知接收消息任务收消息并处理消息。MP 的处理过程如下：

1) 中断处理程序处理：Disable FIFO outbound post queue interrupt，防止处理中断过程中又有新的 outbound post queue interrupt（也就是有新的 outbound 消息）产生，影响前面消息的处理（所以在每条消息处理完成后需注意及时将 FIFO outbound post queue interrupt 打开）；同时释放 semIn 信号量，使中断处理任务得以继续执行。

2) 中断处理任务进程：获取存放消息的 MFA 指针，如果获取成功，则将该指针由 BP 侧内存空间地址转换为 MP 可见的 PCI Memory 空间地址，并判断地址是否合法，以便 MP 读取并处理该消息；将该消息上送给上层程序处理；将收到的消息的指针写入 outbound free queue head pointer，即释放该消息空间，以便 BP 使用；通过 inbound message register 0 产生中断通知 BP 侧 MP 侧已经处理完一条消息；最后打开 outbound post queue interrupt，方便消息中断上报处理。

4.3.2.5 BP 消息接收实现说明

MIPS BP（I2O Host） I2O 中断处理程序的处理过程如下（MP 与之类似）：

读取 Inbound Interrupt Cause Register 和 Inbound Interrupt Mask Register。并从中获取当前中断类型。有两种类型的中断需要处理：

1. message register interrupt，由于我们不用 Message register 收发信息，所以如果产生该类中断，表示 MP 侧成功处理了一条消息，发送队列肯定未满，所以送信号量通知当前因未拿到该信号量而 PEND 的进程继续执行下去。(因为 MP 每处理完一条消息都向 BP 触发一次 message register 中断)。

2. Inbound post queue 中断，通知接收消息任务收消息并处理消息。BP 侧的处理过程如下：

1) 中断处理程序处理：Disable FIFO inbound post queue interrupt，防止处理中断过程中又有新的 inbound post queue interrupt 产生，影响前面消息的处理（所以在每条消息

处理完成后需注意及时将 FIFO inbound post queue interrupt 打开）；同时释放 semIn 信号量，使中断处理任务得以继续执行。

2) 中断处理任务进程：获取存放消息的 MFA 指针，如果获取成功，则将该消息上送给上层程序处理；将收到的消息的指针写入 inbound free queue head pointer，即释放该消息空间，以便 MP 使用；通过 outbound message register 0 产生中断通知 BP 侧 MP 侧已经处理完一条消息；最后打开 inbound post queue interrupt，方便消息中断上报处理。

I2O Host 和 I2O Agent 消息接收中断处理程序公共流程如下：

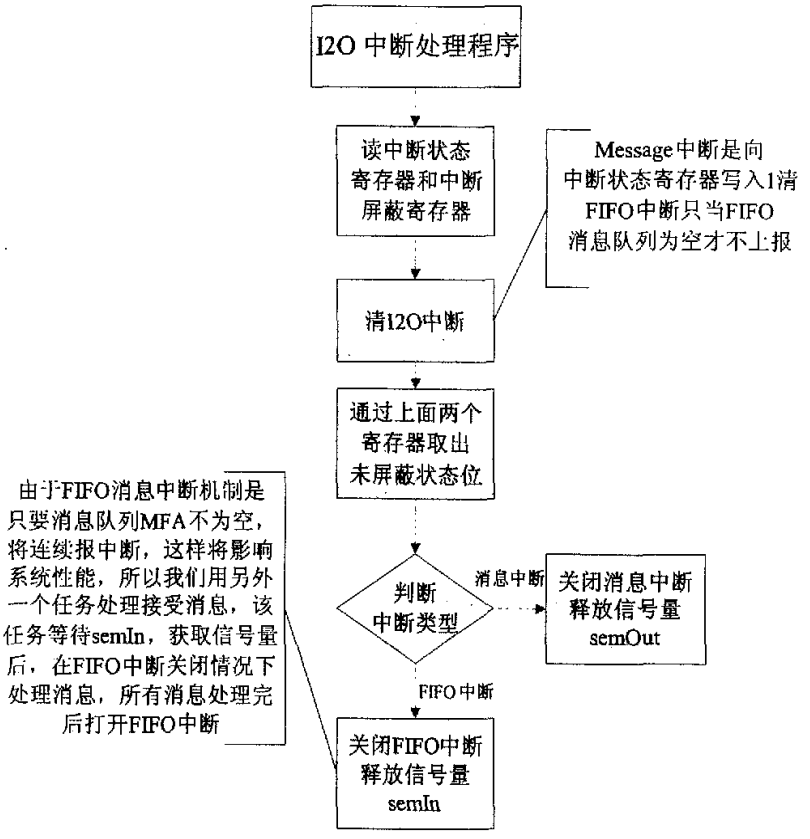

图 19 中断处理程序公共流程图

I2O Host 和 I2O Agent 消息接收的处理任务公共流程如下：

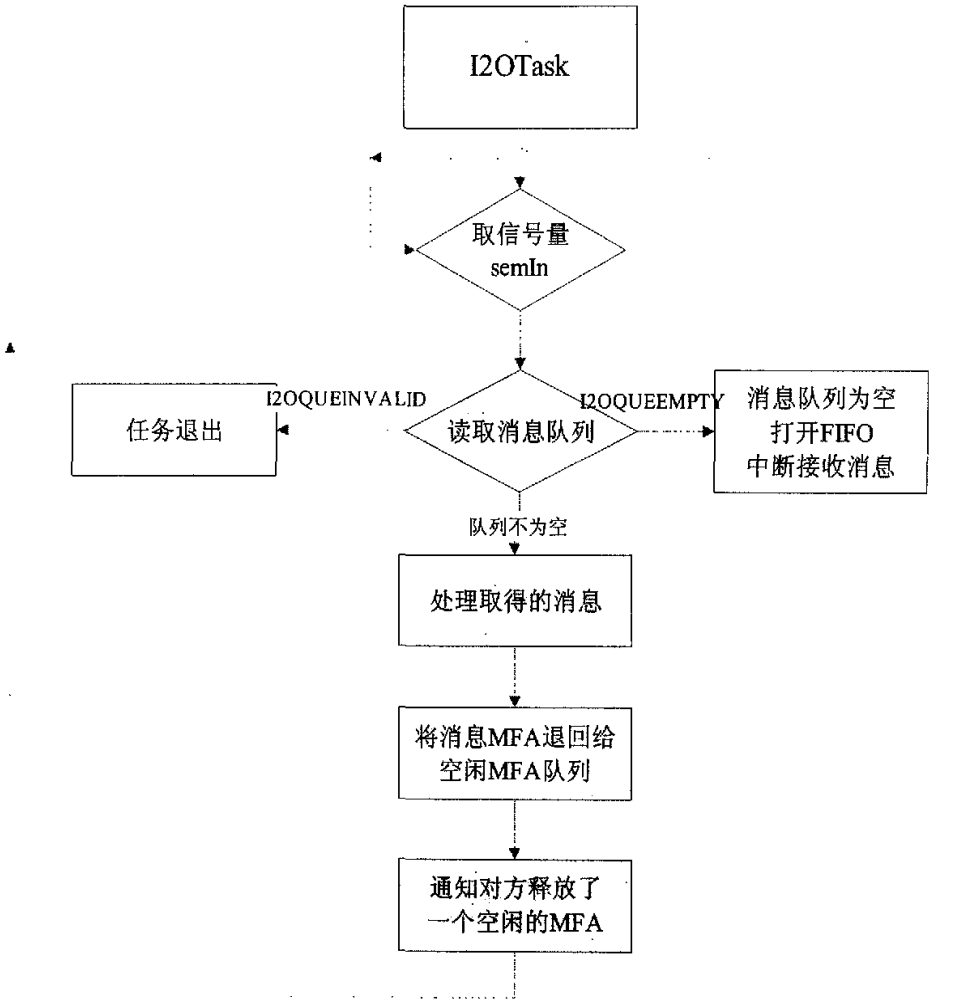

图 20 消息接收的处理任务公共流程图

4.4 I2O 通信的可靠性和效率

本产品中 I2O 通信通道是 MP 和 BP 通信的唯一手段，且 MP 和 BP 间有大量的数据需要传递，所以 I2O 通信的效率问题就尤为突出，而且本产品定位于电信级路由器产品，所以可靠性也是非常重要的。针对以上要求，在本产品中为了提高 I2O 通信效率和可靠性，我们做了如下设计：

1. 使用缓冲区方式，且消息发送方在完成消息发送后，并不等待对方处理状态提前返回。也即，我们在 BP 侧（I2O Host）内存中划分出一段空间专门用于 MP 和 BP 间的 I2O 通信，由于可以根据需要的大小分配每块缓冲区的大小（与 message register 方式相比性能有大幅提高，每次 message register 中断只能传递 4bytes 数据，是远远不能够满足要求的），

所以能够保证一次发送过程可以发送长度很长的消息（甚至是多个短消息的组合），提高了消息传递效率。之所以消息发送后不需要等待对方处理状态返回，是由于我们在 BP 侧划分的内存空间是 MP 和 BP 共享的，MP 通过 PCI 的 memory 空间访问这段空间（详见第三章 ATU 说明），这是由硬件地址翻译机制保证的，所以可以确保消息已经正确发送，不需要等待返回的状态。

2. 定义不同的消息类型，确保性能和可靠性达到要求。根据消息的不同性质和传递要求，我们定义了紧急消息类型、可靠消息类型、普通消息类型等，前两种类型消息是高优先级消息，后一种是低优先级消息。在发送消息时，我们将根据不同的消息类型做相应的处理，保证高优先级消息能够可靠传递，低优先级消息在 I2O 通道忙的情况下有可能没有正常传递。由于 I2O 通信带宽有限（由于 CPU 处理能力和 PCI 带宽的限制），这一点确保了性能和可靠性。

3. 采用中断处理程序和消息处理任务结合方式处理消息接收过程，在中断处理程序中只完成中断类型的判断、清中断和关中断工作，其它工作交给低优先级的消息处理任务处理。由于中断处理程序具有较高的优先级，如果在其中执行时间很长的话，将影响其它进程执行，所以考虑到整体性能，我们在消息处理任务中对消息做进一步的处理，这样做减少了接收消息过程 CPU 的占用率，提高 CPU 利用率和路由器整体性能。

4. 采用多进程访问 I2O 控制，保证了多进程访问 I2O 时的可靠性。为了确保 I2O 通信的可靠性，每个使用 I2O 发送消息的进程在发送消息前都必须获取发送互斥信号量 semApp，该信号量在初始化时完成创建，用于使用 I2O 发送消息的进程间的互斥，这样就保证了多进程访问 I2O 时的可靠性。

通过对 I2O 通信的长时间测试（CPU 在仅有 I2O 消息处理进程运行的情况下收发消息），我们所做的 I2O 通信设计是可以达到系统对双 CPU 间通信要求的。

4.5 数据结构和内部函数

4.5.1 数据结构说明

以下是 I2O 模块中定义的重要数据结构：

```
/* 定义消息相关属性，作为收发消息时不同处理的依据 */
typedef struct tagI2OMSG
{
    short      MsgLen;       /* 消息长度（消息头长+数据长） */
```

```
    char    Version;        /* 消息结构的版本                    */
    char    Type;           /* 消息类型:可靠消息,紧急消息,普通消息  */
}I2OMSG;
/* FIFO 状态，用于初始化 FIFO，提供 FIFO 基地址和每个 FIFO 的大小 */
typedef struct _fifo_stat
{    unsigned long   qsz;  /* 每个 FIFO 的大小 */
    unsigned long   qba;  /* 提供 FIFO 基地址 */
} FIFOSTAT;
/*  按照 Outbound Interrupt Cause register 相应位定义如下结构，
用于中断处理 */
typedef struct _i2o_om_stat
{
   UINT rsvd   : 28;  /* reserved，保留位 */
   UINT opqi   : 1;   /* Outbound post queue interrupt */
   UINT odi    : 1;   /* Outbound doorbell interrupt */
   UINT om1i   : 1;   /* Outbound message_1 interrupt */
   UINT om0i   : 1;   /* Outbound message_0 interrupt */
} I2OOMSTAT;
/*  按照 Inbound Interrupt Cause register 相应位定义如下结构，
用于中断处理 */
typedef struct _i2o_im_stat
{
   UINT rsvd0 : 26;  /* reserved，保留位 */
   UINT ofoi   : 1;  /* Outbound free queue overflow interrupt */
   UINT ipqi   : 1;  /* Inbound post queue interrupt */
   UINT rsvd1 : 1;   /* reserved，保留位 */
   UINT idi    : 1;  /* Inbound doorbell interrpt */
   UINT im1i   : 1;  /* Inbound message_1 interrupt */
   UINT im0i   : 1;  /* Inbound message_0 interrupt */
} I2OIMSTAT;
/* I2O 通信处理过程中，返回值定义，用于判断返回状态 */
typedef enum _i2o_status
{
```

```
    I2OSUCCESS = 0,    /* 操作成功返回 */
    I2OINVALID,        /* 非法操作 */
    I2OMSGINVALID,     /* 需要发送的消息不合法 */
    I2ODBINVALID,      /* Doorbell 不正常 */
    I2OQUEINVALID,     /* 获取 MFA 空间时，不正常 */
    I2OQUEEMPTY,       /* 没有空闲的 MFA 供使用 */
    I2OQUEFULL,        /* 所有 MFA 都被占用 */
    } I2OSTATUS;
/* 定义该结构区分 MP 侧和 BP 侧操作 */
typedef enum _location
{
    IOP = 0,        /* indicate the I2O Host(=BP) operation */
    REMOTE,         /* indicate the I2O Agent(=MP) operation */
} LOCATION;
```

4.5.2 全局变量说明

本模块中定义的全局变量主要用于调试目的，BP 侧定义了如下全局变量：

```
UINT32 i2oMsgAllocEmpty = 0;/* 暂时无法获取空闲 MFA 次数 */
UINT32 i2oMsgAllocSuc = 0;   /* 成功分配空闲 MFA 数 */
UINT32 bpMsgRecvNum = 0;     /* BP 侧接收到的消息数 */
UINT32 bpMsgPostNum = 0;     /* BP 侧发送的消息数 */
UINT32 bpMsgExceedMaxSize = 0;/* BP 侧发送的长度超出范围的消息数 */
```

MP 侧定义了如下全局变量：

```
UINT32 i2oMsgAllocEmpty = 0; /* 暂时无法获取空闲 MFA 次数 */
UINT32 i2oMsgAllocSuc = 0;   /* 成功分配空闲 MFA 数 */
UINT32 mpMsgRecvNum = 0;     /* MP 侧接收到的消息数 */
UINT32 mpMsgPostNum = 0;     /* MP 侧发送的消息数 */
UINT32 mpMsgExceedMaxSize = 0; /* MP 侧发送长度超出范围的消息数*/
```

4.5.3 函数说明

下面将就 I2O 通信处理中几个关键函数做出说明：

- 原型：void I2OInit(LOCATION loc)

说明：完成 I2O 初始化

参数：Loc：区分是 MP 还是 BP 操作

返回值：无

- 原型：I2OSTATUS I2OFIFOInit(QUEUE_SIZE sz,UINT qba)

说明：完成 I2OFIFO（MFA 空间）初始化

参数：sz：每个 FIFO 空间大小

Qba：FIFO 空间的基地址

返回值：I2OSTATUS（具体可参见上面数据结构定义）

- 原型：void I2OHostISR (void)

说明：I2O Host 中断处理程序

参数：无

返回值：无

- 原型：void I2OHostTask (void)

说明：I2O Host 接收消息处理任务入口程序

参数：无

返回值：无

- 原型：void I2OAgentISR (void)

说明：I2O Agent 中断处理程序

参数：无

返回值：无

- 原型：void I2OAgentTask (void)

说明：I2O Agent 接收消息处理任务入口程序

参数：无

返回值：无

- 原型：STATUS I2OPostMsg(LOCATION loc , void * pData)

说明：I2O 消息发送程序

参数：loc：区分是 MP 侧还是 BP 侧操作

pData：上层需要发送的消息指针

返回值：STATUS（OK：发送成功；ERROR：发送失败）

- 原型：I2OSTATUS I2OFIFOAlloc(LOCATION loc,void **pMsg)

说明：发送消息时，FIFO（MFA）分配函数，也就是获取空闲 MFA

参数：loc：区分是 MP 侧还是 BP 侧操作

pMsg：Inbound/Outbound free queue tail pointer 寄存器值

返回值：I2OSTATUS（具体可参见上面数据结构定义）

- 原型：I2OSTATUS I2OFIFOFree(LOCATION loc,void *pMsg)

说明：消息处理完之后，释放 MFA 空间，方便对方发送消息时使用

参数：loc：区分是 MP 侧还是 BP 侧操作

pMsg：Inbound/Outbound free queue head pointer 寄存器值

返回值：I2OSTATUS（具体可参见上面数据结构定义）

- 原型：I2OSTATUS I2OFIFOPost(LOCATION loc,void *pMsg)

说明：通知对方有消息需要处理

参数：loc：区分是 MP 侧还是 BP 侧操作

pMsg：写入 Inbound/Outbound post queue head pointer 寄存器值

返回值：I2OSTATUS（具体可参见上面数据结构定义）

- 原型：I2OSTATUS I2OFIFOGet(LOCATION loc,void **pMsg)

说明：获取对方发送过来的消息指针

参数：loc：区分是 MP 侧还是 BP 侧操作

pMsg：Inbound/Outbound post queue tail pointer 寄存器值

返回值：I2OSTATUS（具体可参见上面数据结构定义）

4.6 I2O 规范与本文实现

上文中分别说明了 I2O 规范和一个 I2O 通信过程的具体实现，可以看出它们是有区别的。本文 I2O 通信实现和 I2O 规范最主要的不同在于以下几个方面：

1. I2O 规范中每个 IOP 都有自己的入队列，主机在自己内存中分配 Outbound message queue；而在我们的实现中将 Inbound message queue 和 Outbound message queue 都在 I2O Host 的内存中分配。

2. I2O 规范中，消息通信过程中，消息队列指针更新是由通信双方软件完成的；而在我们的实现中，由于桥芯片提供了硬件指针更新维护功能，所以我们将 I2O Agent 侧的所有消息指针更新工作全部交给硬件完成，这样也相应的提高了 I2O Agent 侧 CPU 的效率。

3. I2O 规范中，定义了 IOP 间的通信和 Host 与各 IOP 间的通信；我们实现中的环境比较简单，只有两个通信实体，所以我们可以认为系统中只存在一个 IOP，是 IOP 和 Host 间的通信，本实现中可以认为 I2O Host 是 IOP，I2O Agent 是协议中的 Host。

4. I2O 规范中，还有大量关于 OSM 和 OSM 与 DDM 间消息传递的说明；由于我们的实现是基于 VxWorks 操作系统的，所以 OSM 和消息传递部分主要由 VxWorks 操作系统实现，我们在实现中并没有多加关注。

5. I2O 规范中，定义了 I2O 驱动的分层结构并作为 I2O 的一个重要优点；但是我们这里是在 VxWorks 操作系统环境下实现的，所以 I2O 驱动的分层结构并不明显。我们在实现时，将供上层调用完成 I2O 初始化和发送消息的代码放在一起形成一个 API 文件；将具体操作硬件部分放在一起形成一个底层驱动文件。如果需要划分的话我们这部分实现的代码可以认为是 I2O 规范中说明的 DDM 层。

6. I2O 规范中，对具体硬件和软件系统如何实现 I2O 通信并没有严格的规定，而我们的实现是基于具体硬件和 VxWorks 操作系统的，所以在更多的是根据特定的硬件和操作系统来优化我们的代码和整个 I2O 的通信效率。

7. I2O 规范中，并没有对 I2O 通信所基于的底层硬件作出具体规定，但是鉴于 PCI 总线的很多优点，目前 I2O 通信规范的实现多数是基于 PCI 总线的，本课题实现的 I2O 通信也是基于 PCI 总线。本项目硬件通过 PCI 总线实现了 I2O 规范定义的地址域划分、地址翻译和 pushing 方式的 I2O 消息传递功能。

结束语

本文介绍了计算机中总线的由来、发展，并重点介绍了与本文相关的 PCI 总线规范，随后介绍了 I2O 规范（主要是基于 PCI 的）和实现，后面两部分是本文的重点部分。

其中第二章重点介绍了 PCI 相对于过去总线的两个独特优点：软件自动配置和扩展性，并逐一展开说明。第三章结合 I2O 规范，重点介绍 I2O 规范设计的初衷和优点、I2O 硬件环境、I2O 规范建议的驱动分层结构、I2O 环境初始化与配置和 I2O 消息传递的典型过程。第四章重点介绍在一个路由器项目中，针对具体的硬件环境并结合 I2O 规范，我们所实现的 I2O 通信代码。由于是具体实现，所以我们也介绍了相应的软硬件环境和实现中对 I2O 通信可靠性和通信效率的考虑，并且对比了 I2O 规范和我们这个具体实现之间的异同。同时也介绍了本实现中主要的数据结构和函数。

本文是基于目前成熟的 PCI 总线上的 I2O 通信实现，这是一个具体实现性的课题，所以关注的是成熟性，对目前比较新的总线规范涉及不多。当然目前成熟的标准也会有其缺点和不足，随着技术的进步和各种应用的需求，总线和基于总线之上的通信协议也将不断的向前发展，不断会有全新的标准出现替代老的标准，并且将广泛应用于各种系统中。

对于本文我所做的工作包括：

1. 结合 PCI 总线规范介绍 PCI 总线及其优点，从 PCI 设备驱动实现的角度介绍了 PCI 的自动配置和可扩展性，并且对其中关键部分进行了详细说明。其中关于 PCI 设备配置空间的说明也方便读者理解 I2O 和 PCI 的结合。
2. 结合本课题实现部分理解 I2O 规范，在介绍 I2O 规范的同时，说明了 I2O 规范与 PCI 总线规范是如何结合的，当 I2O 通信基于 PCI 总线时，PCI 总线实现了哪些 I2O 规范规定的功能。
3. I2O 通信在本课题中的实现是我所做的主要工作，结合本课题实现的具体软硬件环境，实现 MP 和 BP 间的 I2O 通信，包括以下几个方面：
   1) 利用硬件提供的 PCI 总线实现地址翻译和给对方发出中断等功能，实现 I2O 规范定义的共享内存（本课题在 BP 内存中实现），并使得 MP 可以通过 PCI 总线访问共享内存。
   2) 结合 MIPS CPU 和系统实现的特点调整整个系统的空间分配，为 PCI 总线分配合适的 Memory 空间以访问 BP 侧的共享内存。
   3) 根据 MIPS CPU 和 VxWorks 操作系统对中断的具体要求，将消息处理和中断处理分开实现 I2O 通信中断的快速处理。

4) 本课题实现的是 I2O 通信的底层驱动部分，为上层消息传递提供 API 接口，所以必须考虑整个系统的多任务特点，所以必须考虑到多个进程同时发送消息时 I2O 通信的互斥处理。

5) 为了提高系统高优先级消息传递的可靠性，本课题定义了多种消息类型，在实现中将针对不同的消息类型完成消息传递。

6) 本课题是实现性课题，所以对于实现的代码在系统中的调试和性能测试将尤为重要，在调试中我们模拟了多任务、多优先级消息的高速收发（双方 CPU 只有处理消息和收发消息的进程在运行，这样可以测试出高速收发情况下是否有收发不正常），并且进行了长时间的测试，结果说明我的实现是可以满足实际环境要求的。

I2O 通信是本课题实现中重要的组成部分，对于系统整体性能和稳定性都至关重要，虽然我们做了很多优化，但是具体实现中可能还有不足，希望能够提出改进建议，以进一步提高系统的设计水平。

致谢

首先要向我的导师郑少仁教授致以最衷心的感谢！感谢他在我近三年的研究生学习过程中对我的指导和帮助。从我论文的选题、研究方案的确定、资料的收集、论文的书写和审阅，每一点一滴的进展都有郑教授付出的心血和汗水。郑教授治学严谨、待人宽厚、心胸开阔、处事达观，他在为人上的表率将使我终身受益。同时还要感谢我的师母顾朝霞老师，感谢她在近三年的学习生活中在各方面给我的关心和教导。

衷心感谢田畅教授、张磊副教授在学习和工作上的关心帮助，还有赵志峰、罗和谭两位师兄在学习和工作上的帮助。

衷心感谢我的所有老师、同学和朋友在学习、生活各方面的关心和帮助，感谢在实习期间的领导和同事在工作上的帮助，他们使我有了一个宽松和谐的工作环境，使论文得以顺利完成。

感谢我的父亲和母亲，他们含辛茹苦把我抚养长大，并竭尽所能为我创造宽松的学习环境。我的每一步成长都凝结了他们的汗水、鼓励和关怀。

最后，我要向所有的评审老师和参加答辩的老师在百忙之中抽出时间评阅论文和参加我的答辩表示衷心的感谢！

参考文献

1. Galileo Technology, Inc. System Controller For RM526X/527X/7000 CPUs, Datasheet Revision 1.1 , JAN 10, 2001

2. PMC-Sierra, Inc. *RM7000™ Family User Manual*, Issue1,, May 2001

3. The I2O Special Interest Group, Intelligent I/O (I2O) Architecture Specification, Draft Revision 1.5 , March 1997

4. The PCI Special Interest Group, PCI Local Bus Specification, Revision 2.2 , December 18, 1998

5. 路由器项目组，处理器间通信软件设计方案，2004.9

6. Wind River System Inc，BSP Reference，2001

7. Wind River System Inc，GNU Make，2001

8. Wind River System Inc，Tornado API Guide，2001

9. 路由器项目组，通用路由器系统方案，2003.12

10. 路由器项目组，主控单板硬件设计说明，2003.12

11. Wind River System Inc，Tornado BSP Training Workshop, Version 1.0.2，April 1998

12. Wind River System Inc，Tornado API Reference，2001

13. Wind River System Inc，Tornado User's Guide，2001

14. Philips Semiconductors，THE I 2C-BUS SPECIFICATION, VERSION 2.1，JANUARY 2000

15. Wind River System Inc，TrueFFS for Tornado Programmer's Guide，2001

16. 路由器项目组，通用路由器系统原理，2004.8

17. Wind River System Inc，VxWorks Network Programmer's Guide，2001

18. Wind River System Inc，VxWorks Programmer's Guide，2001

19. Wind River System Inc，VxWorks Reference Manual，2001

20. Wind River System Inc，Tornado Device Driver Workshop, Version 3.0，February 1999

21. Richard Herveille,I2C-Master Core Specification, Rev. 0.3，March 2001

22. Intel Corporation,PC SDRAM Serial Presence Detect (SPD) Specification REVISION 1.2A，December 1997