VxWorks 项目开发实践系列

VxWorks 驱动及分布式编程

孔祥营 张保山 俞烈彬 编著

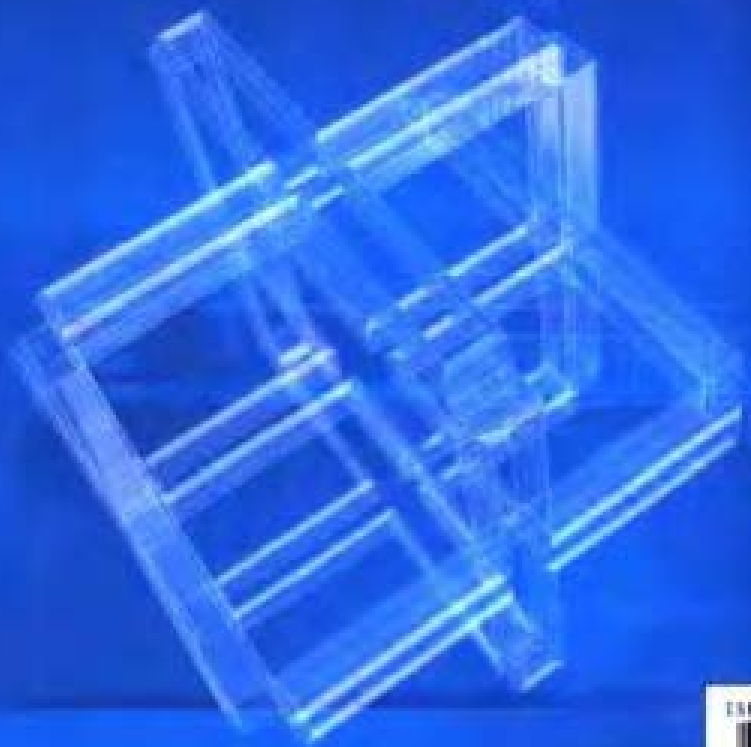
先进性 ■ VxWorks版本不断提高，其功能越来越强，应用范围也越来越广，本书介绍的驱动开发和基于构件的分布式编程知识是目前研究的焦点。

实用性 ■ 全书图文并茂，理论分析和工程开发并重，结合作者多年丰富实践开发经验，给出了详尽的编程实例，具有很强的参考价值。

针对性 ■ 本书能够有针对性地提高读者的编程开发能力，是从事嵌入式开发技术人员的理想参考书。

中国电力出版社
www.infopower.com.cn

VxWorks
驱动及分布式编程



责任编辑 刘 江
封面设计 方 静

上架建议：计算机技术 单片机

ISBN 978-7-5083-5645-7
9 787508 356457 >
定价：39.80元

VxWorks
驱动及分布式编程

孔祥营　张保山　俞烈彬　编著

中国电力出版社
www.infopower.com.cn

内容简介

本书在内容上分为两部分：驱动篇和分布式编程篇。驱动篇主要介绍了字符设备驱动、增强型网络设备驱动（END）以及 WindML 中文字库的设计和 MicroWindows 向 VxWorks 平台上的移植过程；分布式编程篇介绍了分布式构件对象模型（DCOM）和公共对象请求代理体系结构（CORBA）。

本书主要针对从事以 VxWorks 操作系统为基础内核的嵌入式系统开发人员，可以作为广大从事嵌入式技术相关工作的工程技术人员的参考书。

图书在版编目（CIP）数据

VxWorks 驱动及分布式编程 / 孔祥营，张保山，俞烈彬编著. —北京：中国电力出版社，2007.7

ISBN 978-7-5083-5645-7

（VxWorks 项目开发实践系列）

Ⅰ. V… Ⅱ. ①孔…②张…③俞… Ⅲ. 实时操作系统，VxWorks-程序设计 Ⅳ. TP316.2

中国版本图书馆 CIP 数据核字（2007）第 068185 号

丛 书 名：VxWorks 项目开发实践系列

书　　名：VxWorks 驱动及分布式编程

出版发行：中国电力出版社

地　　址：北京市三里河路 6 号　　邮政编码：100044

电　　话：（010）68362602　　传　　真：（010）68316497，88383619

本书如有印装质量问题，我社负责退换

服务电话：（010）58383411（总机）　　传　　真：（010）58383267

E-mail：infopower@cepp.com.cn

印　　刷：汇鑫印务有限公司

开本尺寸：185×260　　印　　张：23.75　　字　　数：580 千字

书　　号：ISBN 978-7-5083-5645-7

版　　次：2007 年 7 月北京第 1 版

印　　次：2007 年 7 月第 1 次印刷

印　　数：0001—4000 册

定　　价：39.80 元

知识改变命运，技术成就梦想！

嵌入式系统开发技术丛书

- **ARM系统开发从实践到提高**
  书号：ISBN 978-7-5083-5371-5　定价：29.80元
- **ASIC芯片设计从实践到提高**
  书号：ISBN 978-7-5083-5378-4　定价：36.00元
- **DSP应用开发从实践到提高**
  书号：ISBN 978-7-5083-5385-4　定价：32.00元
- **FPGA应用开发从实践到提高**
  书号：ISBN 978-7-5083-5377-7　定价：27.00元
- **SoC系统开发从实践到提高**
  书号：ISBN 978-7-5083-5379-1　定价：26.00元
- **嵌入式无线互连系统开发从实践到提高**
  书号：ISBN 978-7-5083-5662-6　定价：35.00元

8051单片机技术应用系列丛书

- **8051单片机彻底研究——入门篇**
  书号：ISBN 978-7-5083-5154-8　定价：26.00元
- **8051单片机彻底研究——实习篇**
  书号：ISBN 978-7-5083-4665-4　定价：26.00元
- **8051单片机彻底研究——基础篇**
  书号：ISBN 978-7-5083-4666-1　定价：36.00元
- **8051单片机彻底研究——经验篇**
  书号：ISBN 978-7-5083-5153-7　定价：32.00元
- **嵌入式Linux程序设计与应用案例**
  书号：ISBN 978-7-5083-5386-3　定价：36.00元
- **电机控制型单片机SPMC75应用基础**
  书号：ISBN 978-7-5083-4668-5　定价：35.00元
- **模块化软件在AVR单片机及教学机器人上的应用**
  书号：ISBN 7-5083-4666-1　定价：19.80元（含1CD）

中国电力出版社
www.infopower.com.cn

中国电力出版社读者服务卡

非常感谢您选择中国电力出版社计算机与自动化类图书，您的支持是对我们工作最大的肯定！请对我们的图书提出宝贵的意见和建议，以帮助我们不断提升图书质量，继续推出更符合读者需求、更实用、品质更高的图书。

返回此服务卡后，您将成为我们的正式读者会员，能更快捷地了解到最新的图书出版信息和优惠购书信息，谢谢！

姓名 _____（必填）性别 _____ 学历 _____

职业 _____ 职称 _____

年龄 □ 10~20　□ 20~30　□ 30~40　□ 40 以上

工作单位 _____

电子邮件 _____（必填）联系电话 _____（必填）

通信地址 _____

邮政编码 _____

您经常阅读哪种类型的图书：

□操作系统 □数据库 □网络 / 通信 □ Web 设计 □程序设计 / 软件开发 □ 图形图像与多媒体

□单片机 / 嵌入式系统 □机电一体化 □自动化 □电子技术 □其他 _____

您对中国电力出版社计算机与自动化类图书印象最深的几本图书是：

_____

_____

您对本书的评价：

_____

_____

您认为此类图书的价格定位在多少合适？ _____

您最希望我们出版本方向哪些内容的图书？

_____

您希望成为我们的作 / 译者吗？

您准备编写的图书名称：_____

您可以翻译的图书类型（从事的专业成研究方向）_____

您推荐引进出版的 _____

您的其他建议 _____

地址：北京市西城区三里河路 6 号中国电力出版社（100044）
电话：010-58383411　**传真**：010-58383267
E-mail: infopower@cepp.com.cn
敬请访问 **www.infopower.com.cn**

前　言

VxWorks 是美国风河（WRS）公司推出的面向嵌入式领域的强实时操作系统，其集成开发环境为 Tornado，随着发布版本的不断提高，其功能越来越强大，应用范围也越来越广，目前已广泛应用于通信、银行、航空、航天、国防等领域，已引起了众多嵌入式从业人员的关注。在基于 VxWorks 操作系统的应用与开发中，硬件驱动程序开发和基于构件的分布式编程是目前研究的焦点。本书针对这两方面内容，主要介绍了 VxWorks5.5 嵌入式实时操作系统下的驱动程序开发和分布式编程技术。通过将理论分析和工程开发的有机结合，在说明基本原理的基础上，给出了详尽的编程样例。

本书在内容上分为两大部分，即硬件驱动篇和分布式编程篇。

硬件驱动篇主要介绍了字符设备驱动程序、增强型网络设备驱动程序（END）以及 WindML 中文字库的设计和 Microwindows 向 VxWorks 平台上的移植过程。字符设备驱动与 END 是两类最常见的设备驱动，第 3 章和第 4 章首先介绍了两种类型驱动框架体系，并分别以 i8250 和 ns83820 芯片为例，给出了相应驱动的详细实现过程和代码；第 5 章在分析 WindML BMF 字库结构的基础上，给出了由 XWindows 的 BDF 字库转换得到 BMF 字库的代码实现；图形是嵌入式操作系统的弱项，Mircowindows 是嵌入式系统广泛使用的开源图形包，第 6 章详细介绍了 Mircowindows 向 VxWorks 平台的移植步骤，并给出了几个底层驱动的实现。

分布式编程篇介绍了分布式构件对象模型（DCOM）和公共对象请求代理体系结构（CORBA），分别是由微软（Microsoft）和对象管理组织（OMG）推出的两种组件规范，提供了 VxCOM/VxDCOM 的分布式组件对象模型，同时第三方的软件开发商也提供了 CORBA，用于解决分布式编程，这其中包括宝兰（Borland）公司的 RT-VisiBorker、中创的 InforBuf 以及华盛顿大学开源的 TAO。

COM/DCOM、CORBA 是软件构件的两种不同规范，它们是解决分布式系统编程的主要方法、嵌入式实时系统下的构件技术。随着结构化设计、面向对象技术和基于构件的软件开发应用的增多，软件设计方法学的变化也改变着软件架构的设计，促进软件构件在更高层次上的重用，从源代码重用、二进制目标重用到架构重用，构件的开发方便了软件的独立升级和并行开发。

本书第 1～5 章由孔祥营编写，第 7～13 章由张保山编写，俞烈彬编写了第 6 章并提供了关于 VxWorks 操作系统下 TAO 有关内容。希望本书的出版，能够对有志于在嵌入式实时系统的领域进行探索的朋友有所借鉴和帮助。由于时间仓促和水平所限，书中难免存在错误和疏漏，敬请广大读者批评指正。

编　者

2007 年 3 月

目　　录

第1章 建立开发环境

我们已经习惯了在 Windows、UNIX/Linux 等桌面操作系统下的软件开发，在这些环境下，有丰富的资源支持开发工作，CPU 足够快，存储器足够大，键盘鼠标配置齐全，各种软件开发工具足够友好，开发的程序同样运行在相同的平台（计算机、操作系统）之上。例如，使用 MicroSoft Visual C++开发一个 Windows 程序，从编码、编译、连接、调试到发布，整个开发流程都可以在 MicroSoft Visual Studio 中完成。

而嵌入式系统软件开发则不同，因为嵌入式系统大多数是应用于特定领域的定制系统，受系统资源限制，多数本身不具备提供运行开发工具的能力，因而需要采用主机目标机开发方式，即软件编辑和编译都连接在主机（通常是 Windows、Unix、Linux）上进行，由交叉编译工具生成的嵌入式软件目标码通过网口、串口等下载到目标机（待开发的嵌入式硬件系统）上。嵌入式软件运行在嵌入式硬件上，调试器运行在开发主机上，通过调试口控制嵌入式软件的运行，并获取相应信息，完成应用调试。这种开发方式称作主机目标机远程开发模式，如图 1-1 所示。

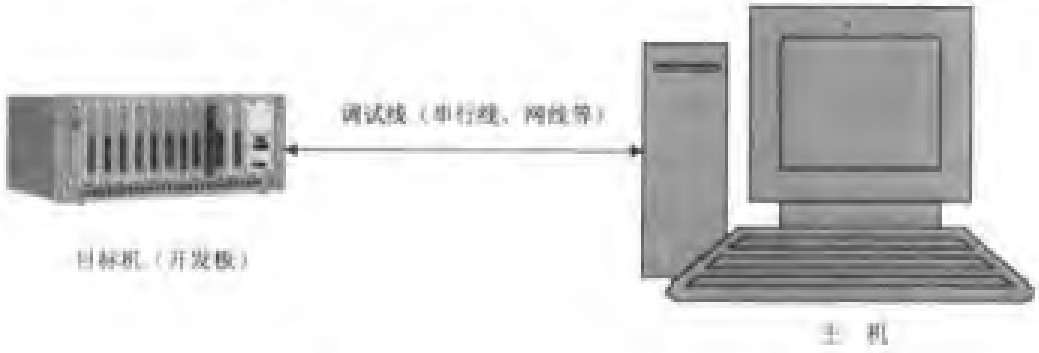


图 1-1 嵌入式系统软件开发方式示意图

VxWorks 是目前应用最多的嵌入式实时操作系统之一，广泛应用于工业控制、医疗器械、通信、航空航天以及国防电子等领域。VxWorks 软件开发也是采用主机目标机开发方式的。开发环境的建立是进行软件开发的第一步，本章介绍与 VxWorks 开发环境建立相关的问题，包括相关文件的介绍、串口下载调试方式、VxWorks 可引导优盘制作、VxWorks 与 Windows 文件互拷贝等内容。

1.1 引导行详解

VxWorks 支持多种引导下载方式，包括软盘、硬盘（ATA、SCSI）、DOC(DiskOnChip)、网络、串口等等，此外，通过修改相关文件还可以增加其他引导下载方式，如 NandFlash、优盘等等。这些引导下载方式配置定义的有关代码放在 BSP 目录下的 config.h 文件中，与网络之间的引导还涉及到 configNet.h 等文件，具体过程实现则在 target\all 目录下的 bootConfig.c 中。

config.h 中与引导有关的内容主要有：

（1）包含 configall.h 文件，configall.h 定义了 VxWorks 所有默认的设置。

（2）定义了引导行（bootline），引导行约定了引导设备、影像路径和网络 IP 地址等。

（3）文件系统及存储设备配置。

（4）定义了系统包含的网络设备（END）驱动程序。

（5）修订了 configall.h 文件中的一些设置，如硬件设备配置、WDB 调试方式等。

现以 X86 体系下 Pentium BSP 目录下的 config.h 为例，进行说明详细解释如图 1-2 所示。

该文件中，默认引导行为：

```
"fd=0,0(0,0)host:/fd0/vxWorks.st h=90.0.0.3 e=90.0.0.50 u=target"
```

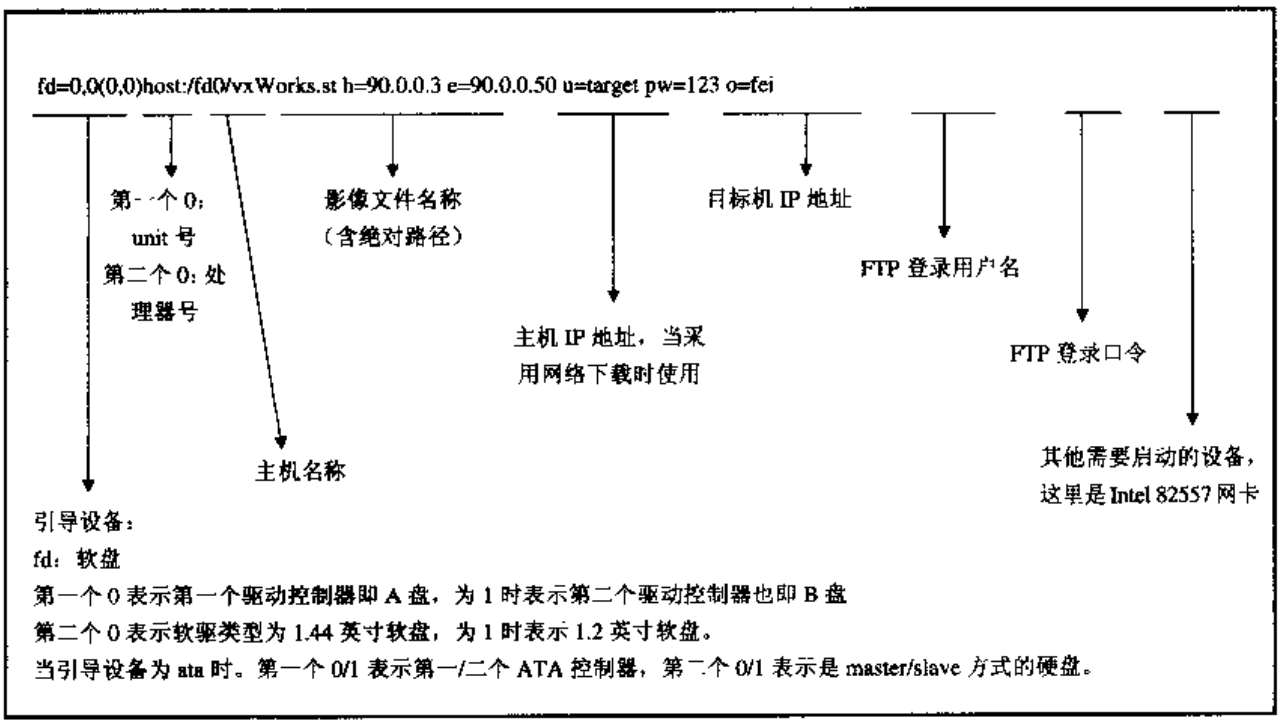


图 1-2　引导行详细解释

引导行中经常变化的是 host 之前的部分。当采用网络引导下载时，应为 nicName(0,0)，其中，nicName 为网卡设备的名称，常用网卡设备名称如表 1-1 所示，括号中的第一个"0"指明是系统中相同类型网卡的第 1 块（使用第二块应为"1"），第二个"0"是指处理器编号。采用网卡引导"时，影像文件名称绝对路径应为相应文件在主机 FTP 根目录下的相对路径，例如，如果主机 FTP 根目录为 D:\tornado\target\proj\project0\default\，VxWorks 影像即放在该目录下，那么，host:后应为 VxWorks。主机 IP 与目标机 IP 地址根据具体情况而定，二者必须在同一个子网段内。FTP 用户名与口令应该与 FTP server 分配的用户名和口令一致。

表 1-1　常 用 网 卡 设 备 名 称

名称	设备宏定义	网卡芯片	驱动程序
fei	INCLUDE_FEI	Intel 82557/9 10/100 芯片	Fei82557end.c
elPci	INCLUDE_EL_3C90X_END	3COM 3c905B 芯片	El3c90xend.c

续表

名称	设备宏定义	网卡芯片	驱动程序
ln	INCLUDE_LN_97X_END	AMD 79C97x 芯片	Ln97xend.c
dc	INCLUDE_DEC21X40_END	Intel/DEC 21x4x 芯片	Dec21x40end.c
ene	INCLUDE_ENE_END	Ne2000 ISA 网卡	Ne2000end.c
elt	INCLUDE_ELT_3C509_END	3COM 3c509b ISA 网卡	elt3c509End.c

每种网卡驱动对应的引导设备名称可从其头文件中找到，这些头文件放在 target\h\drv\end 目录下，每个文件中通常有一个宏 XXX_DRV_NAME 来定义其设备名称，例如，在 el3c90cend.h 中，有如下定义：

```
#define EL3C90X_DEV_NAME    "elPci" /* device name */
```

当用相应设备引导时，必须在 config.h 中定义对应的宏，网卡如表 1-1 所示，其他设备如表 1-2 所示。

表 1-2 引导设备及宏定义

名 称	设备宏定义	设 备	影像文件名	备 注
fd	INCLUDE_FD	软 盘	/fd0/vxWorks	还需要定义：INCLUDE_DOSFS
ata	INCLUDE_ATA	ATA 硬盘	/ide0/vxWorks	
ide	INCLUDE_IDE	IDE 硬盘	/ata0/vxWorks	
scsi	INCLUDE_SCSI	SCSI 硬盘	/sd0/vxWorks	
tffs	INCLUDE_TFFS	DOC（DiskOnChip）	/tffs0/vxWorks	
tsfs	INCLUDE_TSFS	串 口	/tgtsrv/vxWorks	见 1.2 节
pcmcia	INCLUDE_PCMCIA	PCMCIA 卡	/pcmcia0/vxWorks	

当系统中使用两块网卡时，如果是相同的网卡，则需要修改 configNet.h 文件，在数组 endDevTbl 中增加其定义。例如，当系统中有两块 intel82557 网卡时，需要增加以下内容：

```
#ifdef INCLUDE_FEI_END
    {0, FEI82557_LOAD_FUNC, FEI82557_LOAD_STRING, FEI82557_BUFF_LOAN,
    NULL, FALSE},
    {1, FEI82557_LOAD_FUNC, FEI82557_LOAD_STRING, FEI82557_BUFF_LOAN,
    NULL, FALSE},
#endif /* INCLUDE_FEI_END */
```

其中，斜体内容是针对第二块网卡而增加的，1 表示 unit 号，从 0 开始编号，1 即为第二块网卡。

如果系统中存在两块网卡，又需要通过其中一块进行调试（该网卡支持 WDB 协议），则必须把该块网卡的配置项移到 endDevTbl 数组的最前面，无论两块网卡是否相同。例如，在上面的例子中，如果采用第二块 intel82557 作为调试口，则需要改为如下形式：

```
#ifdef INCLUDE_FEI_END
    {1, FEI82557_LOAD_FUNC, FEI82557_LOAD_STRING, FEI82557_BUFF_LOAN,
    NULL, FALSE},
    {0, FEI82557_LOAD_FUNC, FEI82557_LOAD_STRING, FEI82557_BUFF_LOAN,
    NULL, FALSE},
#endif /* INCLUDE_FEI_END */
```

当采用不同网卡时，也要将调试网卡放在前面，例如当系统中同时存在 AMDLn97X 和 3COM3c905B 两种网卡时（各一块），configNet.h 中数组 endDevTbl 默认顺序为：

```
#ifdef INCLUDE_EL_3C90X_END
    {0, EL_3C90X_LOAD_FUNC, EL_3C90X_LOAD_STR, EL_3C90X_BUFF_LOAN,
    NULL, FALSE},
#endif /* INCLUDE_EL_3C90X_END */

#ifdef INCLUDE_LN_97X_END
    {0, LN_97X_LOAD_FUNC, LN_97X_LOAD_STR, LN_97X_BUFF_LOAN,
    NULL, FALSE},
#endif /* INCLUDE_LN_97X_END */
```

当使用 AMDLn97X 作为调试口，应改为：

```
#ifdef INCLUDE_LN_97X_END
    {0, LN_97X_LOAD_FUNC, LN_97X_LOAD_STR, LN_97X_BUFF_LOAN,
    NULL, FALSE},
#endif /* INCLUDE_LN_97X_END */

#ifdef INCLUDE_EL_3C90X_END
    {0, EL_3C90X_LOAD_FUNC, EL_3C90X_LOAD_STR, EL_3C90X_BUFF_LOAN,
    NULL, FALSE},
#endif /* INCLUDE_EL_3C90X_END */
```

引导行中，影像文件后的部分可根据情况删减，并且先后次序也可调整，每一项的意义由等号前面的字符确定，例如，h 表示等号后的是主机 IP 地址；e 则表示等号后的是目标机 IP 地址；u 表示 FTP 用户；pw 表示 FTP 口令；o 表示还需要启动的设备。除上述几个选项外，还有 tn、s 和 f 经常用到。

- tn=后可跟目标机名称，系统启动后可用 hostShow 看到。
- s=后通常是一个文本文件名(含路径)，该文件是一个 shell 可识别的文本文件。VxWorks 启动后，targetShell 将打开该文件，逐行解释并执行。下面是一个简单脚本文件的例子。

```
ifMaskSet("fei0",0xffffff00);
ifAddrSet("fei0", "192.0.1.12");
ld </ata0/app.out
sp appInit
```

该脚本文件首先设置 IP 掩码和 IP 地址，然后从硬盘中加载应用程序 app.out 文件，最后发起一个任务执行应用程序入口函数 appInit。在上面的例子中，采用脚本文件以及动态加载的方式，避免应用程序与 VxWorks 内核连接在一起，有利于应用程序单独升级，可大大提高系统配置的灵活性。

- f=后跟系统启动标示，是一个双字节十六进制整数，定义如表 1-3 所示。

表1-3 引导标示定义

flag 位	为 1 时的意义
0	不作为系统控制器
1	加载本地系统符号
2	不自动启动
3	快速自动启动，不作数字递减
5	禁止登录安全
6	如果定义了 DHCP，则使用 dhcp 得到启动参数；如果定义了 BOOTP，则使用 bootp 得到启动参数
7	使用 tftp 得到引导影像
8	使用代理 ARP

1.2 采用串口连接的开发调试环境

很多驱动软件的开发是在不具备网络的环境前提下进行的，包括网络驱动的调试。在这种情形下，由于串口（驱动）相对简单，通常采用串口作为调试口。本节介绍如何建立以串行口作为调试口的过程。以 X86 平台为例。

准备工作：除了开发主机（已安装 Tornado）外，还要一台有软驱的 PC 机，至少有一个串口，串行电缆，空白软盘。如果没有 PC 机也没关系，就可采用 VMWARE 软件在开发主机上虚拟一台 PC 机，要求开发主机至少要有两个串口，一个分配给 VMWARE；一个作为 target Server 文件系统的输出口。

（1）修改引导行。

前面已经介绍过，采用串口作为引导下载设备时，引导行设备名称为“tsfs”，当主机 target Server 文件系统设置为 VxWorks 影像所在路径时，引导行如下：

```
"tsfs(0,0)host:vxWorks"
```

按照上面的引导行形式修改 BSP 目录下的 config.h 文件中的 DEFAULT_BOOT_LINE 定义：

```
#define DEFAULT_BOOT_LINE  "tsfs(0,0)host:vxWorks"
```

（2）向 config.h 增加以下内容。

```
#define     INCLUDE_SERIAL
#undef      CONSOLE_TTY
#define     CONSOLE_TTY         NONE
#undef      WDB_TTY_CHANNEL
#define     WDB_TTY_CHANNEL     0              /* 使用第一个串口进行调试 */
#undef      WDB_COMM_TYPE
#define     WDB_COMM_TYPE  WDB_COMM_SERIAL  /* 使用串口作为调试口 */
```

```
#undef      WDB_TTY_BAUD
#define     WDB_TTY_BAUD    115200      /* 串口通信波特率，建议采用 38400bps */
#define     INCLUDE_TSFS_BOOT           /* 使用第一个串口下载影像 */
```

（3）修改 bootconfig.c 文件。

找到以下内容：

```
#define  INCLUDE_TSFS_BOOT_VIO_CONSOLE  /* needed for Target Server Console */
```

改为：

```
#undef  INCLUDE_TSFS_BOOT_VIO_CONSOLE  /* needed for Target Server Console */
```

（4）重新编译该 BSP 的 bootrom 或 bootrom_uncmp。

（5）用 mkboot.bat 制作引导盘。

（6）用串行线物理连接主机与目标机串口。

（7）配置一个可引导影像工程，其中 VxWorks 配置如图 1-3 所示。

```
WDB agent connection
```

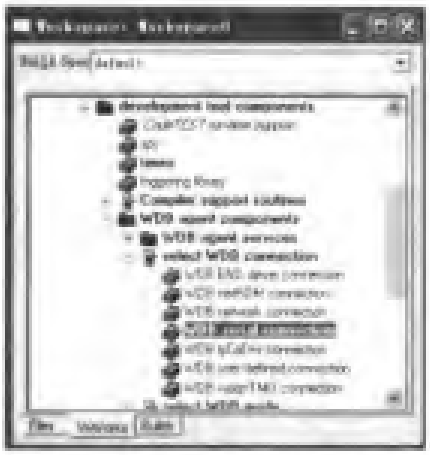


图 1-3　VxWorks 影像 WDB 配置

在 WDB serial connection 上单击鼠标右键，修改其参数值，保持与主机设置一致，如图 1-4 所示。

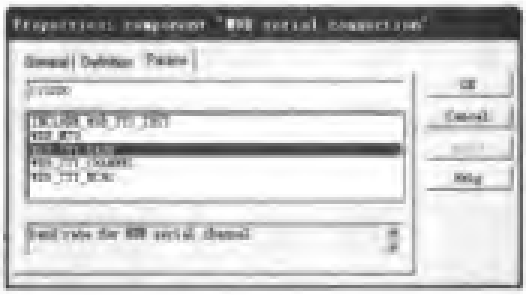


图 1-4　串口 WDB 参数

```
WDB_TTY_BAUD 为 115200bps
WDB_TTY_CHANNEL 为 0
```

（8）设置主机 target Server。

在 Tornado 环境工具栏中，依次点击 tools、target Server、configure，新建一个连接配置，如图 1-5 所示。

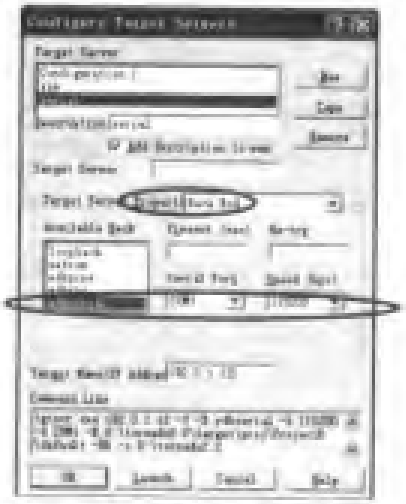


图 1-5 Target Server 串口连接配置图一

图中，Back End 方式选用 wdbserial，使用主机第一个串口（COM1），波特率为 115200bps（与目标机一致）。然后在 Target Server Project 下拉选框中，选中 Target Server File System，设置其根路径为 VxWorks 影像所在目录，并允许读写，如图 1-6 所示。最后点击 OK 保存。

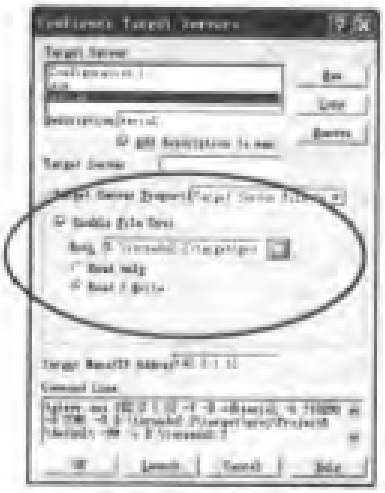


图 1-6 Target Server 串口连接配置图二

（9）用制作的软盘启动目标机，并启动上面配置的 Target Server（在主机 Tornado 中）。目标机启动在自动计数结束后屏幕显示如图 1-7 所示。

```
Booting using TSFS...
Make sure that your Target Server is started with -R[root] option.
Waiting for Target Server connection... Done

Loading /tgtsvr/vxWorks...
857696
```

图 1-7 串口引导目标机下载过程示意图

最后一行数字是 VxWorks 影像大小，不同配置的影像大小也不同。

Target Server 窗口显示内容如图 1-8 所示。

图 1-8　影像下载过程 Target Server 截图

当下载结束时，target server 显示 TSFS 文件关闭。

下载时间可以初步估算，例如当 VxWorks 影像大小为 size，串口波特率为 baud 时，那么下载传输时间 t（单位秒）约为：

```
t=size*10/baud
```

上述例子中，VxWorks 的大小为 2543328，波特率为 115200，传输时间约为：

```
857696*10/115200=75 秒
```

大约 2min，由于文件传输还有一些控制信息，因此，实际时间要大于计算时间。

上述配置中，PC CONSOLE 和网络都可以保留，便于信息浏览和调试网络驱动。当保留网络时，VxWorks 启动后可能会提示一些错误信息，是因为它错误地把引导设备 tsfs 作为 END 设备启动但不成功造成，这时可以通过修改文件 target\config\comps\src\net\usrNetBoot.c 来除掉这一错误。

（1）找到函数：void usrNetDevNameGet (void)

（2）在语句：

```
if ( !strncmp (sysBootParams.bootDev, "scsi", 4) == 0) ||
```

之后增加：

```
!strncmp (sysBootParams.bootDev, "tsfs", 4) == 0) ||
```

即可。

当连接不成功时，应考虑以下情况是否存在：

（1）主机方 target server 是否启动？bootrom 启动后，会等待主机 target server 启动。

（2）双方波特率是否一致？系统默认连接时使用 9600 bps 或 38400bps。

（3）波特率是否过高？降低波特率到 38400bps，波特率过高增加了传输出错概率。低波特率虽然慢些，但稳定可靠。115200b/s 传输虽然快但对线路质量要求较高，容易出错，可能会导致欲速则不达，建议使用 38400bps。

（4）串行线是否正确？可以用两台 PC 机超级终端进行线路通信验证。

1.3　BootConfig 文件分析

嵌入式系统各不相同，在有些情况下可能需要从特殊设备引导加载 VxWorks，例如希望能从优盘加载 VxWorks，或系统使用 NAND/NOR FLASH 作为存储器，这两种 FLASH 通常不使用 DOSFS，而 VxWorks 默认配置中不支持引导这些方式，就需要修改相应文件达到目的。

包含引导设备的选择与初始化代码的文件 bootconfig.c 是以源码方式提供的，放在

target\config\all 目录下，通过分析该文件，可以找到增加新的引导设备的方法。

Bootconfig.c 文件与引导设备相关的部分放在函数 bootLoad 中。该函数以引导行和影像入口点地址作为参数。bootLoad 函数根据引导设备名称，调用相应的加载函数，如配置串口，且引导设备名称为“tsfs”时，调用 tsfsLoad 完成影像加载。

以下以系统配置引导设备为软盘 fd 为例，分析 bootLoad 的流程，ata、ide 等设备的情况与此类似，如图 1-9 所示。

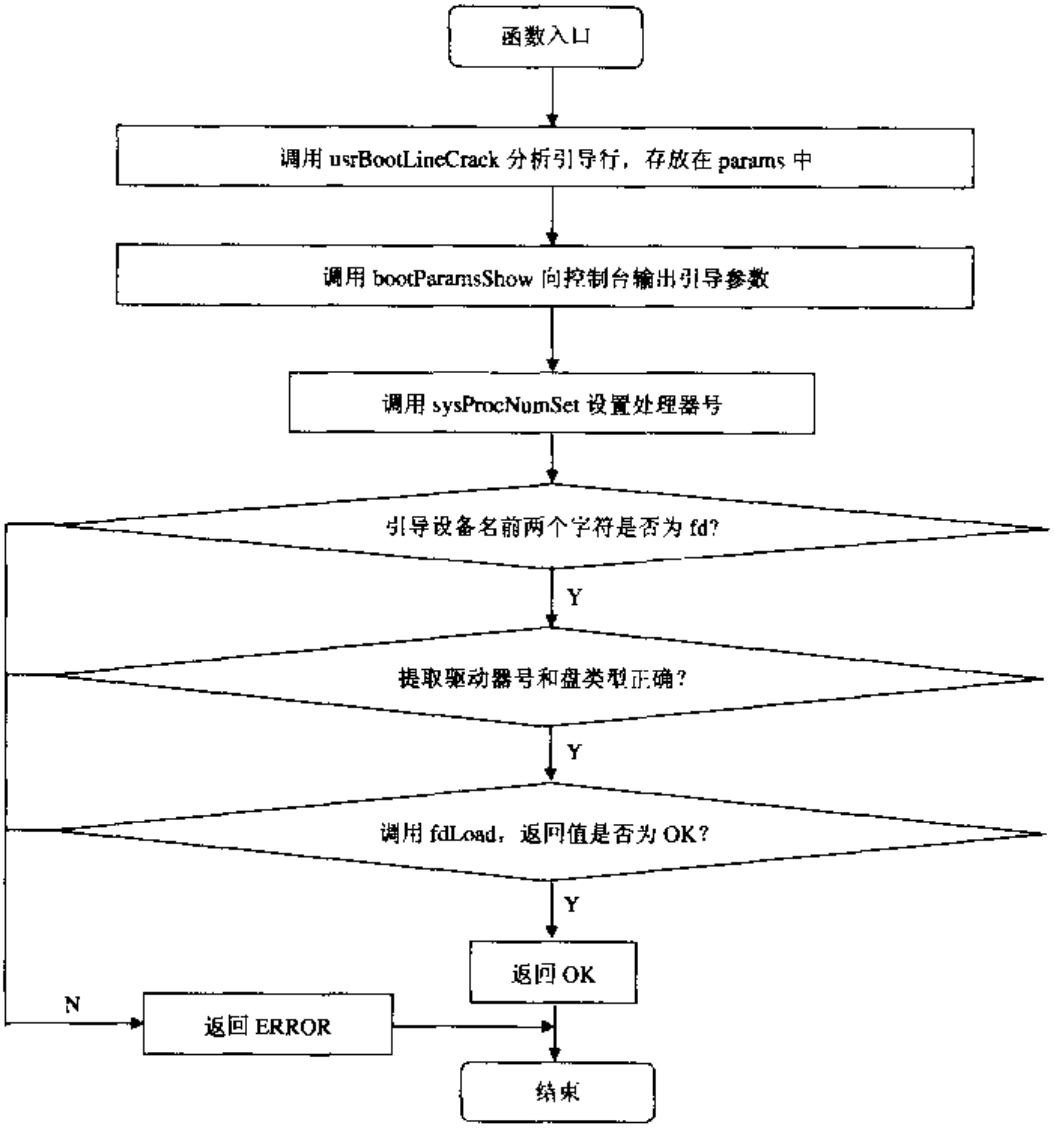


图 1-9　bootLoad 流程图

fdLoad 函数流程如图 1-10 所示。

从上述流程可知，从一个引导设备加载影像，实际上是根据引导设备名称判明是哪种类型设备，然后初始化相应的设备，如果是存储类，则用文件操作，调用 open 打开存放在相应设备上的影像文件，并返回文件句柄；如果是网络设备，则通过 FTP 或 TFTP 操作，同样也能获得文件句柄。将该文件句柄传递给 bootLoadModule 函数，bootLoadModule 负责将影像文件加载到系统内存（分析影像文件格式，将文件的 text、data、bss 等段放置到内存指定的位置）。

因此，当增加一种新的引导设备（非网络设备）时，可以仿照 fd 设备的引导流程，修改

bootLoad 函数，增加相应设备的分支处理，并实现对应的 xxLoad 函数，当然前提是该设备的驱动程序已实现，如果是存储类设备还需实现相应的文件系统支持。下一节以优盘为例介绍具体过程。

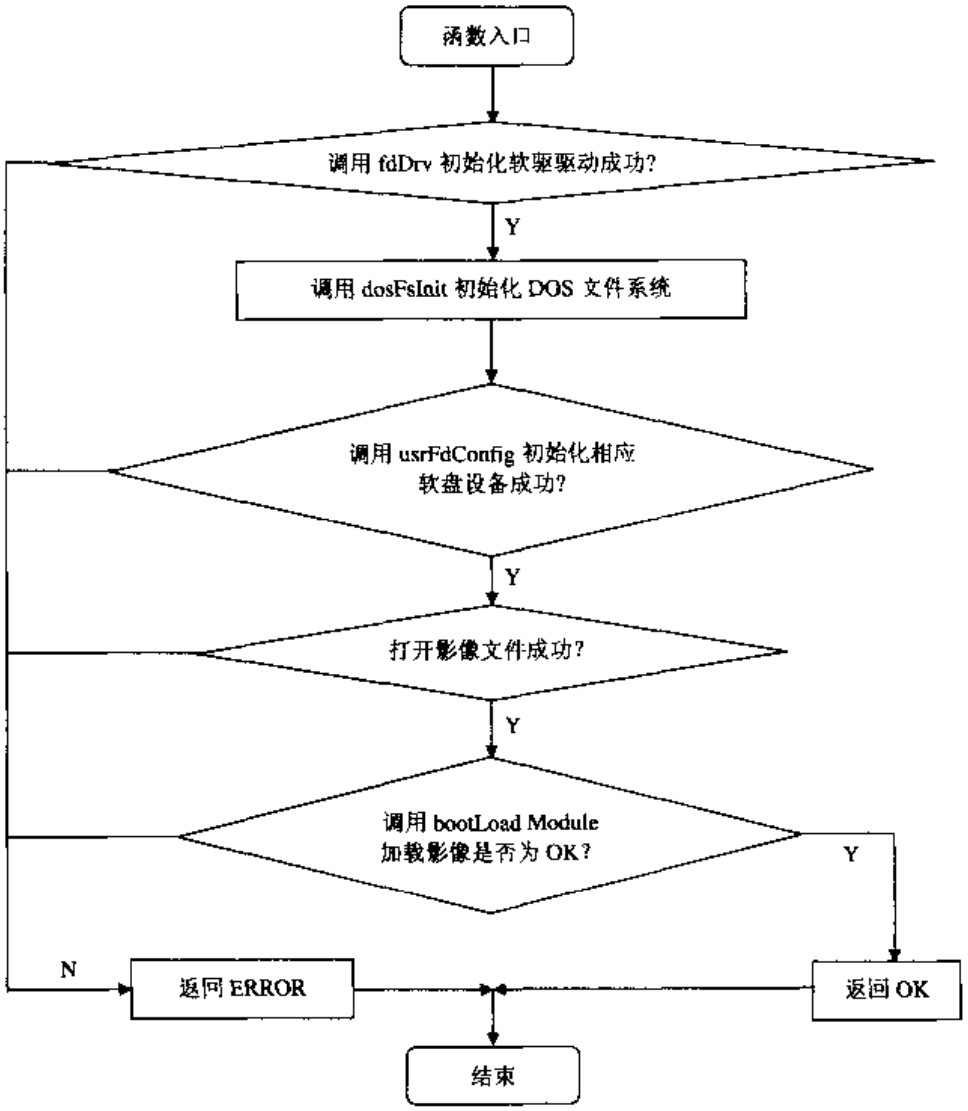


图 1-10　fdLoad 流程图

1.4　VxWorks 可引导优盘制作

优盘引导 VxWorks 就是影像文件放在优盘上，系统启动后将从优盘加载 VxWorks。由于软盘极不稳定，很容易出错，而且容量有限，而硬盘更新 VxWorks 影像又极不方便，优盘则没有这些缺点。

1.4.1　bootrom 制作

解决从优盘加载 VxWorks 的关键是要有优盘驱动，VxWorks5.5 提供的 USB 协议栈支持大容量存储类设备，也就是支持优盘，只要在 bootrom 中配置 USB 协议栈及大容量存储类设备驱动，bootrom 启动后就能识别优盘，然后在其上创建 DOSFS 文件系统，就能从优盘上读取

影像文件。

假定优盘引导设备名为“usb”，在USB协议栈找到优盘后，就会创建一个名称为“/usb0”的块设备。

具体修改如下：

（1）在函数boorLoad函数中，找到：

```
#ifdef  INCLUDE_FD
```

在它的前增加以下代码：

```
#ifdef  INCLUDE_USB
    dosFsInit (NUM_DOSFS_FILES);          /* initialize DOS-FS */

                    usbDevInit();         /* usb初始化函数 */
    taskDelay(sysClkRateGet()*2);
                    /* 由于USB设备枚举需要一定的时间，加入延时，等待优盘初始化完成 */

    if (strncmp (params.bootDev, "usb", 3) == 0)
    {

    if (strlen (params.bootDev) == 3)
        return (ERROR);
    else
        sscanf (params.bootDev, "%*3s%*c%d%*c%d", 0, 0);

    if (usbLoad (params.bootFile, pEntry) != OK)
        {
        printErr ("\nError loading file: errno = 0x%x.\n", errno);
        return (ERROR);
        }

    return (OK);
    }

#endif  /* INCLUDE_USB */
```

（2）找到以下代码：

```
#ifdef  INCLUDE_FD

#include "../../src/config/usrFd.c"

/*******************************************************************************
*
* fdLoad - load a vxWorks image from a local floppy disk
*
* RETURNS: OK, or ERROR if file can not be loaded.
*/

LOCAL STATUS fdLoad
```

在其前增加以下代码：

```
#ifdef  INCLUDE_USB

/*******************************************************************************
*
* usbLoad - load a vxWorks image from a local USB disk.
*
* RETURNS: OK, or ERROR if file can not be loaded.
*/

LOCAL STATUS usbLoad
    (
    char    *fileName,
    FUNCPTR *pEntry
    )
    {
    int fd;

    /* load the boot file */

    printErr ("Loading %s...", fileName);

    if ((fd = open (fileName, O_RDONLY, 0)) == ERROR)
    {
        /* 因为优盘初始化可能需要较长时间，出错时重试一次*/
        taskDelay(sysClkRateGet()*2);
        if ((fd = open (fileName, O_RDONLY, 0)) == ERROR)
        {
          printErr ("\nCannot open \"%s\".\n", fileName);
          return (ERROR);
        }
    }


    if (bootLoadModule (fd, pEntry) != OK)
        goto usbLoadErr;

    close (fd);
    return (OK);

usbLoadErr:

    printErr ("\nerror loading file: status = 0x%x.\n", errno);
    close (fd);
    return (ERROR);
    }

#endif  /* INCLUDE_USB */
```

(3) config.h 修改。

将 config.h 中的引导行改为：

```
"usb0.0(0,0)host:/usb0/vxworks"
```

增加

```
#define INCLUDE_USB
```

(4) makefile 修改。

在 makefile 中，增加链接 USB 驱动模块，重新制作 bootrom，即可支持从优盘加载 VxWorks 影像。

1.4.2 USB 设备初始化

在 1.4.1 小节的修改中，函数 usbDevInit 定义如下：

```
#define FIND_UHCI 1
#define FIND_OHCI 2

    STATUS usbInit (void)
        {
        UINT16 verStatus;
        UINT16 usbdVersion;
        char   usbdMfg [USBD_NAME_LEN+1];


        if (!usbdInitByKernel)
            {
            if (usbdInitialize() != OK)
                {
                printf(" Failed to initialize USBD\n");
                return ERROR;
                }
            else
                {
                usbdInitByKernel = TRUE;
                printf("USBD Initialized.\n");
                }
            }
        else
            {
            printf("USDB Already Initialized\n");
            }

        if ((verStatus = usbdVersionGet (&usbdVersion, usbdMfg)) != OK)
            {
            printf ("usbdVersionGet() failed..returned %d\n", verStatus);
        return ERROR;
            }
        return OK;
        }

    STATUS usbDevInit()
```

```
    {
        int hci;
        hci=usbFindHCI();  /*  用来识别 UHCI 还是 OHCI*/
        if((hci>0)&&(hci<3)){
            usbInit();
            switch (hci){
                case FIND_UHCI:
                    usrUsbHcdUhciAttach();
                    break;

                case FIND_OHCI:
                    usrUsbHcdOhciAttach();
                    break;
                default:
                    return ERROR;

            }
            usrUsbBulkDevInit();
            return OK;
        }else
            return ERROR
}
```

其他函数可从 VxWorks USB 协议栈中找到。

1.4.3 可引导优盘制作

对于 X86 系统，如果 BIOS 支持 USB 引导，bootrom 就可以放在优盘上，下面介绍制作可引导优盘的步骤。

（1）文件组成。需要以下工具，可从 Internet 上下载。

vfloppy.zip	虚拟软驱工具
msdos71f.zip	DOS7.1 软盘影像
usboot167.rar	引导 U 盘制作工具
wrar350sc.exe	文件压缩解压工具

（2）制作步骤。

1）解压上面的三个压缩包到相应目录。

2）安装并运行虚拟软驱工具 WinVF.exe（在 vfloppy 安装后相应目录下），影像文件选 DOS71_1.IMG（在 MSDOS71f 解压后的相应目录下），文件大小选“1.44MB”，设置选“B:”，最后点击按钮“加载”，在“我的电脑”会出现一个“5.25 软盘”，盘符为 B。

3）运行 USBoot.exe，将出现警告信息，此时不用理会。选择对应的 U 盘，工作模式选“ZIP 模式”（带分区 32 扇区），最后点击开始“按钮”，根据提示完成后续操作。

4）在 U 盘上创建一个 config.sys 文本文件，内容如下：

```
DOS=HIGH,UMB
DEVICE=HIMEM.SYS
shell=vxload.com bootrom.dat
```

5）首先把文件 vxload.com（在 tornado\host\X86-win32\bin 目录下）、himem.sys（在解压

后的 DOS7.1 目录下）拷贝到 U 盘，然后再将 BSP 目录下的 bootrom 或 bootrom_uncmp 拷贝到 U 盘，并改名为 bootrom.dat。

6）为了使 Ctlr+X 和 reboot 命令能够正常复位目标机，需要修改 bsp 目录下的 config.h 和 syslib.c。

①在 config.h 中，增加

```
#define INCLUDE_USB
```

②在 sysLib.c 中，函数 sysToMonitor 中在变量声明后增加

```
#ifdef INCLUDE_USB
    sysReboot();
    return OK;
#endif
```

7）重新编译操作系统 VxWorks 影像，并拷贝到 U 盘中即可。

1.5 从文件中读引导行

在开发过程中，有些情况下可能需要改变主机或目标的 IP 地址，传统的做法是在启动过程中手工修改引导参数，或者修改 config.h 中的引导行参数，然后再制作 bootrom，这样就需要重新制作引导软盘或对 FLASH 编程。我们知道。这样做不仅麻烦而且由于软盘质量较差很容易写坏，当引导扇区变坏或 bootrom 文件不连续时，就不能引导成功。下面以引导软盘为例介绍采用引导行与 bootrom 分离的方法，从而改变传统的操作方法。

1.5.1 bootline 获取过程

bootline 的获取是在 bootconfig.c 中完成的，通常放在指定内存地址中，宏 BOOT_LINE_ADRS 指向该地址，如 X86 体系中，默认存放在内存地址 0x1200 处。

在 config.h 中 BOOT_LINE_ADRS 定义如下：

```
#define LOCAL_MEM_LOCAL_ADRS    0x00000000  /* fixed */
/* The bootroms put the boot line at the following address */

#define BOOT_LINE_ADRS  ((char *) (LOCAL_MEM_LOCAL_ADRS+BOOT_LINE_OFFSET))
#define BOOT_LINE_SIZE  255                 /* use 255 bytes for bootline */
```

Bootrom 在启动过程中，函数 usrBoot 在完成内存初始化、时钟初始化、IO 子系统初始化、控制台初始化和 MUX 初始化后，将调用 bootCmdLoop，然后会发起一个名为“tRoot”的任务，该任务的函数体是 bootCmdLoop。bootCmdLoop 的流程如图 1-11 所示。

其中最关键的是函数 usrBootLineInit，该函数将 DEFAULT_BOOT_LINE 拷贝到 BOOT_LINE_ADRS 处，我们可以通过修改该函数，用自己定义的 bootline 替换 DEFAULT_BOOT_LINE。基本思想是：假定 bootrom 放在软盘上（其他介质类似），可以将自己定义的 bootline 以文本文件方式放在软盘上，然后在 usrBootLineInit 中，从软盘上读取该文件，并且将其内容拷贝到 BOOT_LINE_ADRS 处。

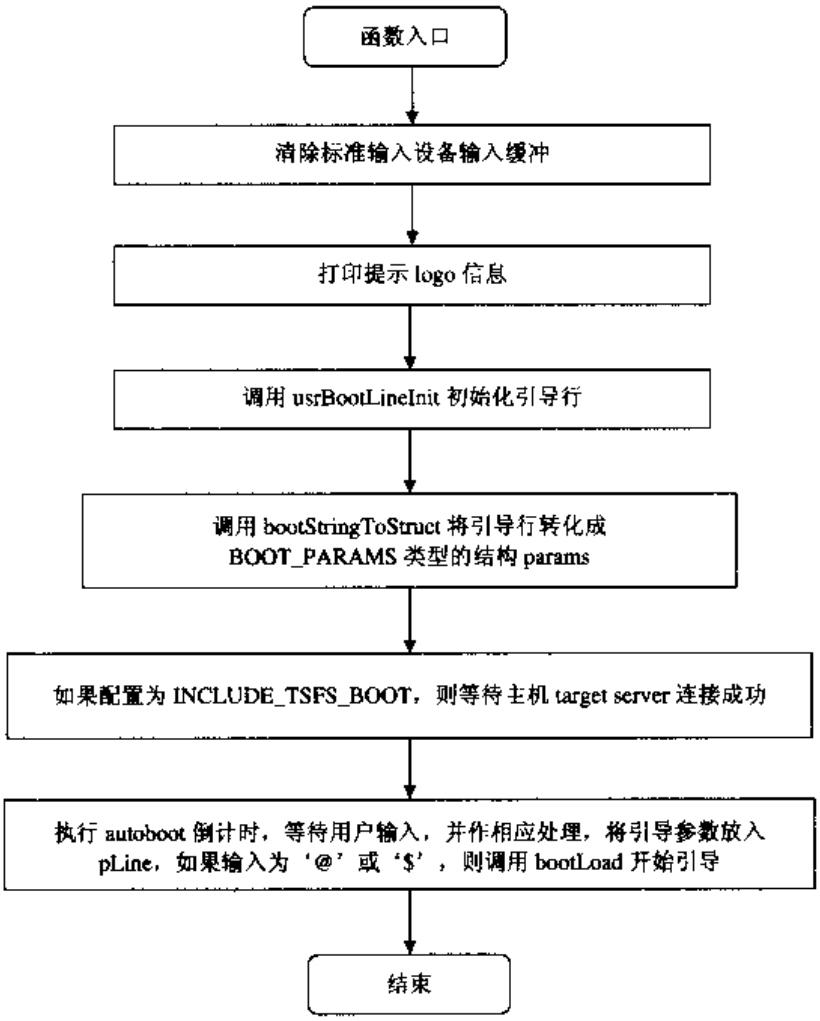


图 1-11　bootCmdLoop 流程图

1.5.2　usrBootLineInit 修改

由于在 bootCmdLoop 调用之前，IO 子系统已初始化完成，多任务环境也已经建立，因此，可以通过初始化文件系统，搭接软盘设备来访问软盘，具体操作如下：

（1）假定 bootline 以文本文件方式放在软盘（位于第一个驱动器）上，名称为 bootline.txt，内容如下：

```
fei(0,0)host:vxworks h=xxx.xxx.xxx.xxx e=xxx.xxx.xxx.xxx u=my pw=123
```

（2）修改 usrBootLineInit 的实现。

```
LOCAL void usrBootLineInit
    (
    int startType
    )
    {
    if (startType & BOOT_CLEAR)
    {
    /* this is a cold boot so get the default boot line */
```

```
    if ((sysNvRamGet (BOOT_LINE_ADRS, BOOT_LINE_SIZE, 0) == ERROR) ||
        (*BOOT_LINE_ADRS == EOS))
        {
        /* either no non-volatile RAM or empty boot line */
#if 0
        strcpy (BOOT_LINE_ADRS, DEFAULT_BOOT_LINE);
#else
    /* 初始化 dosFs 文件系统 */
    char bline[256];
    int  blLen;
    dosFsInit (NUM_DOSFS_FILES);        /* initialize DOS-FS */

    /* 搭接软盘设备 */
    if (usrFdConfig (0, 0, "/fd0") == ERROR)
    {
        printErr ("usrFdConfig failed.\n");
        return (ERROR);
    }

    /* 读引导行文件 */
    if ((fd = open ("/fd0/bootline.txt", O_RDONLY, 0)) == ERROR)
    {
        printErr ("\nCannot open /fd0/bootline.txt \n");
        return (ERROR);
    }
    if((blLen =read(fd,&bline,256))>0)
        strncpy (BOOT_LINE_ADRS, &bline, blLen);
    else
        printErr ("read bootline error!!!\n");
#endif
        }
    }
    }
```

重新生成 bootrom，然后制作引导软盘，当我们需要修改引导参数时，仅需在 Windows 环境下重新编辑 bootline.txt 文件，并将其保存到软盘即可。当然 bootline 内容仍要遵循引导行的格式。

1.6 VxWorks 与 Windows 系统文件互拷贝

在一些项目的实际应用中，应用系统的运行需要很多数据资源，这些资源都存放在存储器，如硬盘等介质上，并且需要不断地从外部更新。一般情况下可能想到多种方法：将存储介质挂在 Windows 系统下完成数据拷贝，或通过编写网络通信程序完成数据传输，或采用 FTP/TFTP 服务进行拷贝。以上方法各有缺点，第一种对于非硬盘设备不适用；第二种不够灵活（编写一个通用程序难度很大）；第三种需要对每个文件逐个执行拷贝操作。本节介绍一种利用 NFS（网络文件系统）进行文件拷贝的方法，可以避免上述缺点。

VxWorks 网络协议栈支持丰富的网络服务，包括网络文件系统（NFS）。NFS 允许通过网

络与他人共享目录与文件，使用 NFS 用户和程序可以像访问本地文件一样访问远端系统上的文件。同其他网络服务类似，NFS 也由服务器和客户端两部分组成，提供访问服务的是服务端，而执行访问的是客户端。当 VxWorks 与 Windows 系统进行文件拷贝时，由于 Windows 桌面人机操作方便友好，通常 Windows 作为 NFS 的客户端，VxWorks 作为 NFS 服务器，在 Windows 上运行 NFS 客户端工具，可以将 VxWorks 系统上的 NFS 服务设备影射成本地的磁盘设备（盘符）。下面介绍其详细过程。

（1）配置 VxWorks 影像，包含 NFS Server，如图 1-12 所示。

图 1-12　将 NFS 配置到 VxWorks 影像中

（2）在 usrAppInit.c 中，增加以下代码，并将对应的设备作为 NFS root 路径。

本例以软盘根目录下的 nfsroot 目录为例。

```
#define NFS_ROOT "/fd0/nfsroot"
    nfsExport(NFS_ROOT,0,0,0);
```

（3）重新 build VxWorks 影像，启动系统，与 Windows 主机通过网络连接，并且在同一个子网中。

（4）在 Windows 系统中安装一个支持 NFS 客户端的工具软件，以 omni-NFS 为例。安装后启动其 NFS Client 工具，如图 1-13 所示。

图 1-13　omni-NFS Client 界面

（5）点击 HOSTEDIT，对主机表进行编辑，增加一个新的 host 名称与 IP 地址的对应关系，如图 1-14 所示。其中 localhost 是系统中原有的，vx 则是后加的。点击菜单 New，出现如图 1-15

所示的界面，在 host 中键入 VxWorks 目标机的名称（可任意，此处假定为 vx），在 Host IP 中键入 VxWorks 目标机的 IP 地址。点击 Next，在新出现的界面中点击 Finish，即可完成。

图 1-14 HOSTEDIT 主界面

图 1-15 主机名及 IP 地址编辑界面

（6）在 NFS Client 主界面中，双击一个空盘符，就会出现如图 1-16 所示的界面，然后在 Server Name 中，键入上面命名的主机名（vx），并在登录路径中键入 NFS 服务器定义的根路径名称（/fd0/nfsroot）。点击下一步，出现如图 1-17 所示的界面，Authenticated by 选择 "UID and GID"。最后点击下一步，在新出现的界面中点击完成。这时 NFS Client 主界面如图 1-18 所示。

图 1-16 NFS 网络驱动器映射编辑界面

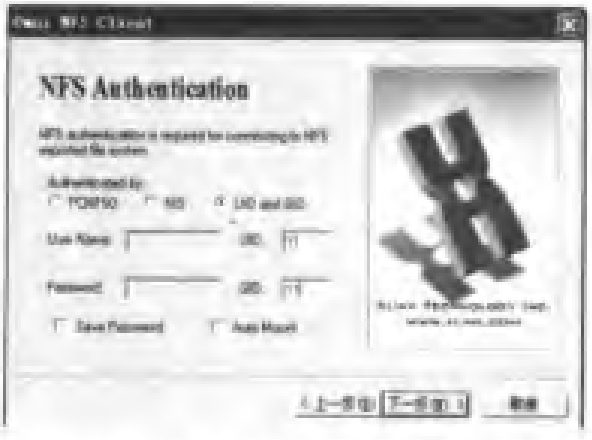


图 1-17 NFS 验证方式配置界面

（7）如图 1-18 所示，选中第一行，点击 MOUNT 按钮，这时目标机会访问软盘，访问结束，图 1-18 中的第一行盘符会变为绿色。这时打开资源管理器，会看到多出一个新的盘符 I，如图 1-19 所示。这样就可像访问本地其他磁盘一样，对该目录进行读写。

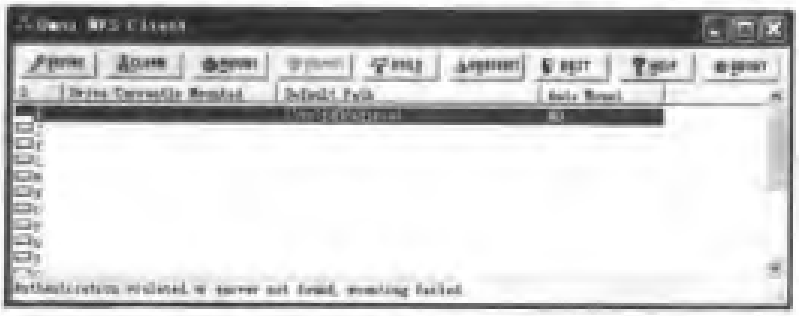


图 1-18　增加 NFS 网络驱动器映射后的 Omni NFS 客户端界面

图 1-19　网络驱动器映射完成后的资源管理器

为了增加安全性，VxWorks NFS 也提供了安全验证机制，具体内容请参见函数库（nfsdlib、nfslib 等）。

第2章 驱动开发基础

硬件设备驱动程序可以看作硬件的软件抽象，编写驱动程序必须具备一定的基础知识，包括：

- 编程语言：汇编语言和C语言，这是最基本的要求。
- 硬件设备的一些基本概念，如I/O、存储器、DMA、中断、寄存器和总线等。
- 硬件的体系结构及工作原理：中断如何产生、数据如何交换等。
- 操作系统提供访问与控制硬件资源的手段：函数和数据结构。
- 写一个标准设备驱动，必须了解操作系统相应的设备驱动体系。
- 驱动的开发调试工具与手段。

2.1 硬件基础

编写外部设备的驱动程序，首先要对计算机组成结构及外部设备具备一定的了解：

- 什么是寄存器、如何编址、如何访问与控制。
- CPU与外设的交互机制，是中断还是查询。
- 外设数据传送机制，采用程序控制I/O还是DMA方式。

2.1.1 计算机组成结构

简单地说，计算机可以看成是由中央处理器（CPU）、存储器、与外部设备相连接的系统总线和扩展系统功能的外部设备组成的，如图2-1所示。

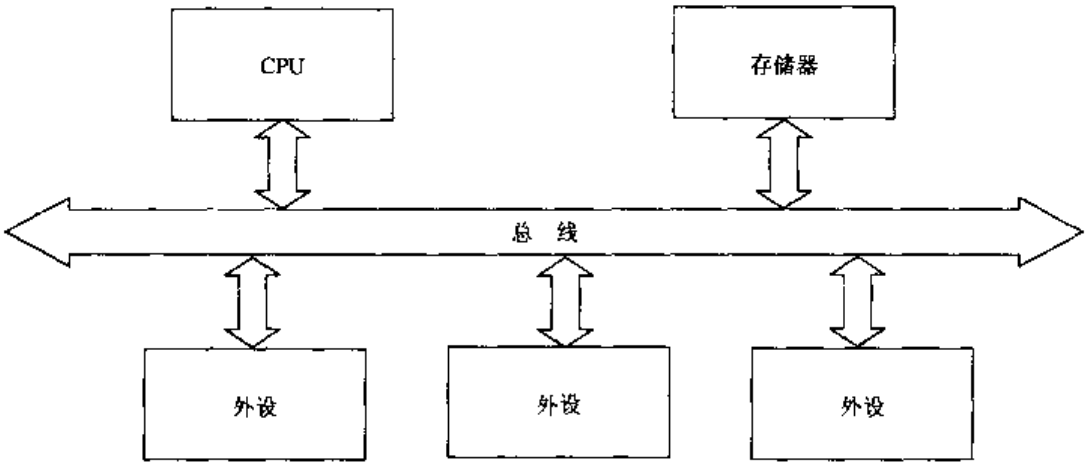


图2-1 计算机结构框图

1. CPU

CPU也称微处理器，是计算机系统的大脑，负责系统的计算和控制核心。

2. 存储器

存储器是 CPU 的工作空间，用来存储程序和数据。所有计算机系统都有一个由不同速度与大小的存储器组成的层次结构。最快的存储器是高速缓存，它被用来暂存主存中的内容。这种存储器速度非常快但非常昂贵，又可分为数据缓存（D-Cache）和指令缓存（I - Cache）。速度仅次于高速缓存的是内存（RAM），是程序代码和数据运行时的载体。内存存储是暂时的，数据和程序只有在计算机通电和未被重启的情况下才保留在这里，因此，系统往往还有由 FLASH 或硬盘等扩充的外存，用来存放永久性的数据。

3. 总线

总线是 CPU 与外部设备互联，用于交换（接收或发送）数据的一组线（引脚），是数据传输的途径，总线不仅是一组传输线而且还包含一套管理信息传输的规则，可以看成计算机系统中一个具有独立功能的组成部件。总线的功能在逻辑上可划分为三部分：地址总线、数据总线和控制总线。地址总线为数据传输指明内存位置（地址），决定了直接寻址的范围。数据总线包含传输的数据，它是双向的，允许数据读入 CPU 也支持从 CPU 读出来，决定了数据传输的宽度。控制总线则包含几条总线分时和方向的控制信号，决定了总线功能的强弱和适应性的好坏。系统中的设备都连接到总线上。PC（个人计算机）系统中，常见的总线有 ISA（工业标准体系结构）和 PCI（外部部件互联）。ISA 是 8MHz 16 位的低速总线，用于早期的机器，目前较新的 PC 主板不再包含这类总线。PCI 是 33MHz 32 位的总线，最新的系统都支持 66MHz 64 位的版本。多数网卡、声卡和显卡都是采用 PCI 总线的。

4. I/O 接口与外部设备

外部设备是一些实现具体功能的物理设备，如终端、打印机、磁盘和网卡等，能够接收 CPU 发来的信息或向 CPU 提供输出。输入/输出（I/O）接口是计算机与外部设备进行信息交换时不可或缺的手段，在整个计算机系统中占有极其重要的地位，没有输入/输出，计算机几乎失去意义。I/O 接口通常是一些控制芯片，有的集成在主板上，如串口、并口和 IDE 控制器；有的则与外部设备集成在一起，如网卡等。这些控制器通过各种总线连接到 CPU 上或相互间互连。控制器是一些类似 CPU 的处理器，它们可以看做简单的处理器，辅助 CPU 完成特定的功能。各种 I/O 接口基本功能主要有：

（1）数据缓冲。

（2）提供联络信息。

（3）信号与信息格式的转换。

（4）设备选择。

（5）中断管理。

（6）可编程功能等。

I/O 接口与外部设备的基本结构如图 2-2 所示。

驱动程序的开发主要是针对 I/O 接口与外部设备进行的，驱动程序可以看成是这些硬件设备的软件抽象。

2.1.2 寄存器、地址空间

1. 寄存器

寄存器是一种由（多个）锁存器和触发器组成的时序逻辑电路，可以看作硬件提供的软件

编程接口，是驱动软件与硬件打交道的主要途径。驱动软件通过读写与设备关联的一组寄存器中的相关位来与外设通信。寄存器通常可分为三类：

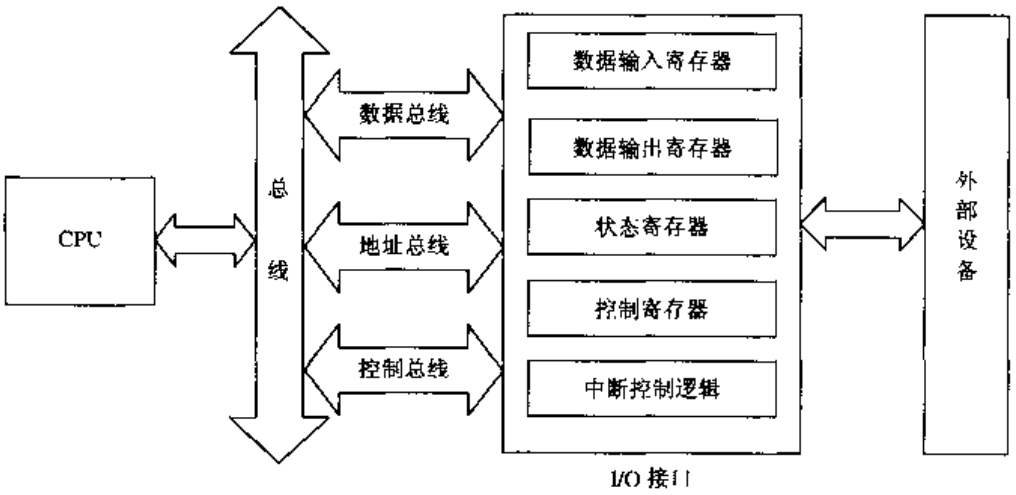


图 2-2 I/O 接口与外部设备的基本结构图

（1）控制寄存器，也称命令寄存器，设置或清除该类寄存器中的位将引起设备开始一个操作，或以某种方式改变硬件的状态。

（2）状态寄存器，这类寄存器包含设备当前的状态信息，驱动程序通过这些寄存器获取设备的状态。

（3）数据寄存器，是设备与处理器交换数据的通通，当数据写到输出缓冲寄存器时，输出设备接收要传送的数据。来自输入设备的数据则存放在设备的输入缓冲寄存器中。

硬件设备的寄存器由硬件设计人员决定，由于大部分硬件没有统一的工业标准，硬件设计人员往往依据硬件的功能、行业惯例和个人喜好进行寄存器的设计。寄存器的多少和复杂程度主要由硬件功能决定，简单的设备仅有几个寄存器，如 8042 键盘/鼠标接口（如表 2-1 所示）、并行口，复杂的设备需要一组数目很大的寄存器，如网卡（如表 4-2 所示）、显卡等等。例如，在 PC 机中，intel 8042 键盘接口芯片包含 4 个 8 位寄存器，占用两个 I/O 端口，分别为 0x60 和 0x64。

表 2-1 intel 8042 键盘接口芯片寄存器定义

寄存器	位	访问	说明
状态寄存器 0x64	0	R	1 表示输出缓冲器满；0 表示空。CPU 读后置 0
	1		1 表示输入缓冲器满；0 表示空。8042 读取后置 0
	2		系统标志，加电启动后置 0，自检通过后置 1
	3		1 表示最后写入的是端口 0x60；0 表示最后写入的是端口 0x60
	4		1 表示键盘未被禁止；0 表示键盘被禁止
	5		1 表示发送超时；0 表示发送未超时
	6		1 表示接收超时；0 表示接收未超时
	7		1 表示从键盘获取的数据奇偶校验错；0 表示未出错

续表

寄存器	位	访问	说明
控制寄存器 0x64	0	W	1 表示使能键盘中断；0 表示禁止中断
	1		1 表示使能鼠标中断；0 表示禁止中断
	2		设置状态寄存器的位 2
	3		1 表示忽略状态寄存器的位 4
	4		1 表示禁止键盘
	5		1 表示禁止鼠标
	6		将第二套扫描码翻译成第一套
	7		保留，应为 0
输入缓冲寄存器 0x60	0~7	R	CPU 从该端口地址中读数据
输出缓冲寄存器 0x60	0~7	W	缓冲 CPU 准备发送到 8048（键盘控制器）的数据

从表中可以看出，状态和控制两个寄存器共用一个 I/O 端口，输入、输出缓冲寄存器共享一个 I/O 端口 0x60，寄存器的功能由读写操作决定，为节省设备所占 I/O 空间，很多硬件都采用这种方式。实际上，intel 8042 并不直接控制键盘，控制键盘的芯片是 intel 8048，它与键盘集成在一起，8042 是与之相匹配的接口芯片，驱动程序与 8048 打交道时需要通过 8042。

2. 地址空间

端口是接口电路中能被 CPU 直接访问的寄存器地址，硬件在设计时将按照计算机约定，为每个寄存器设定相应的端口地址。

计算机系统有两种外部设备接口（寄存器）地址编址方式。一种是外部设备与内存地址统一编址，也称存储器映射编址，这种方式中，将外部设备接口地址与内部存储器地址统一安排在内存的地址空间中，即部分内存地址分配给外部设备，存储器不再使用，外部设备可以看成部分存储单元，访问外部设备占用的地址空间与其他存储器采用相同的指令，即可采用访问正常存储器的装入和存储指令，不需要专门的 I/O 指令，如图 2-3 所示。

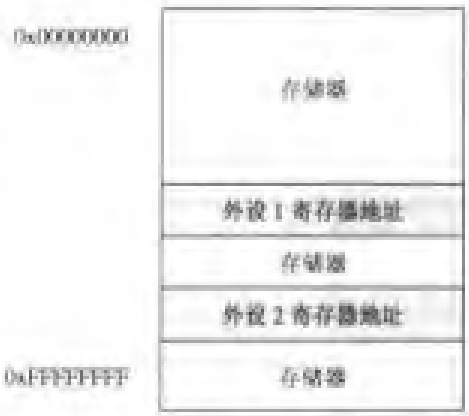


图 2-3 外设与内存统一编址

另一种编址方式是外部设备与内存独立编址，内存地址空间与外部设备地址空间相互独立，CPU 将外部设备寄存器地址映射到称为 I/O 空间的一组地址，这些 I/O 地址也称为端口，不是 CPU 看到的内存空间的一部分，只能通过特定的指令访问。例如，在 X86 体系中，I/O 地址范围从 0x0000 到 0xFFFF，内存地址从 0x00000000 到 0xFFFFFFFF，二者相互独立，互不影响，各有各的访问指令。PC 机 ISA 设备都是采用 I/O

映射方式。

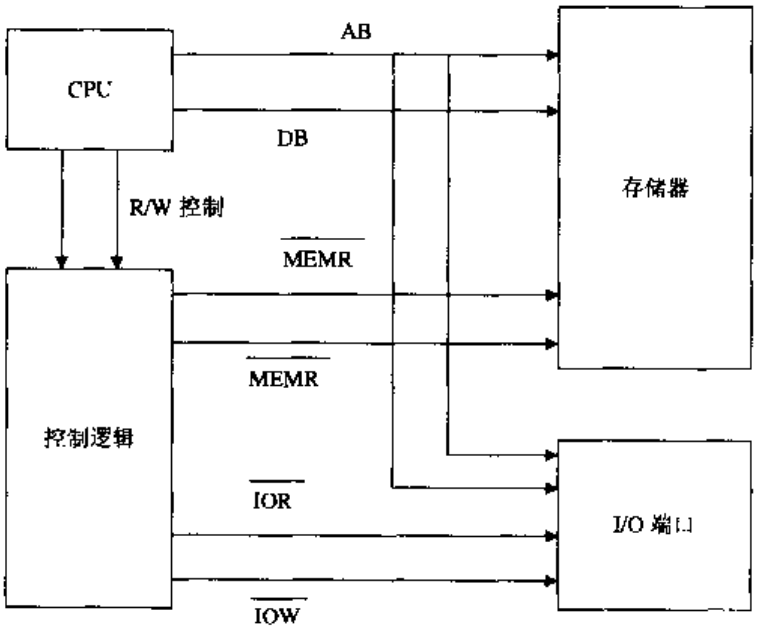


图 2-4 外部设备与内存独立编址

即使在分离 I/O 空间的 CPU 体系上，一些外部设备也采用内存影射寄存器，这样有助于提高具有大寄存器组的高速设备性能。例如，PCI 设备往往支持这种方式，如图 2-5 所示。第四章介绍的 ns83820 芯片就支持这种方式。

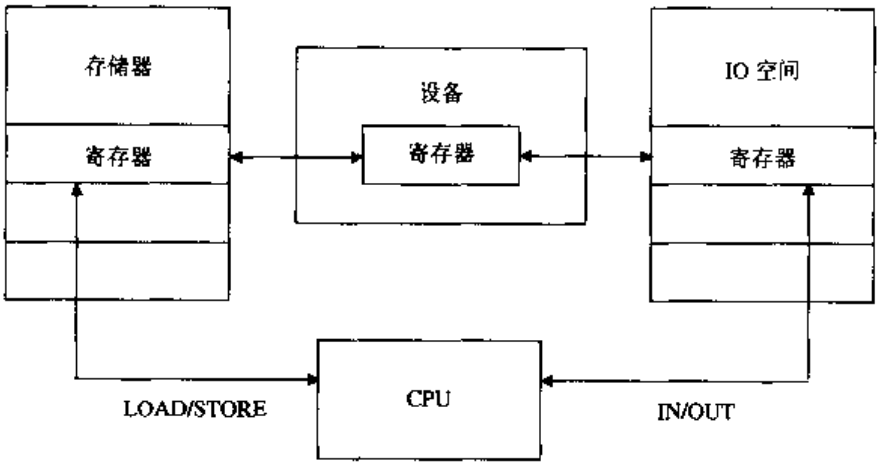


图 2-5 两种映射方式并存

2.1.3 外部设备数据传送方式

外部设备的主要输入/输出方式有四种：

（1）无条件传送。

（2）查询方式。

（3）中断方式。

（4）直接存储器存取（MDA）方式。

本节解释前两种方式。这两种方式也称为程序控制 I/O，数据交换由 CPU 控制完成。

（1）无条件传送方式主要用于一些简单的外部设备，如 GPIO 控制的开关、指示灯（发光二极管）等，这些设备随时都准备接收 CPU 的输出或它的数据都是就绪的，CPU 随时都可读取它的数据。数据交换和指令执行是同步的。这类设备通常只有数据寄存器，没有状态寄存器。

（2）查询方式用于那些数据收发需要一定条件的外部设备，当需要数据传送时，CPU 首先需要查询设备的状态，看看设备是否就绪/空闲，必须在设备条件满足（就绪/空闲）后，才能进行数据传送。这类设备不仅有数据寄存器，还有一个或多个状态寄存器，如图 2-6 所示。

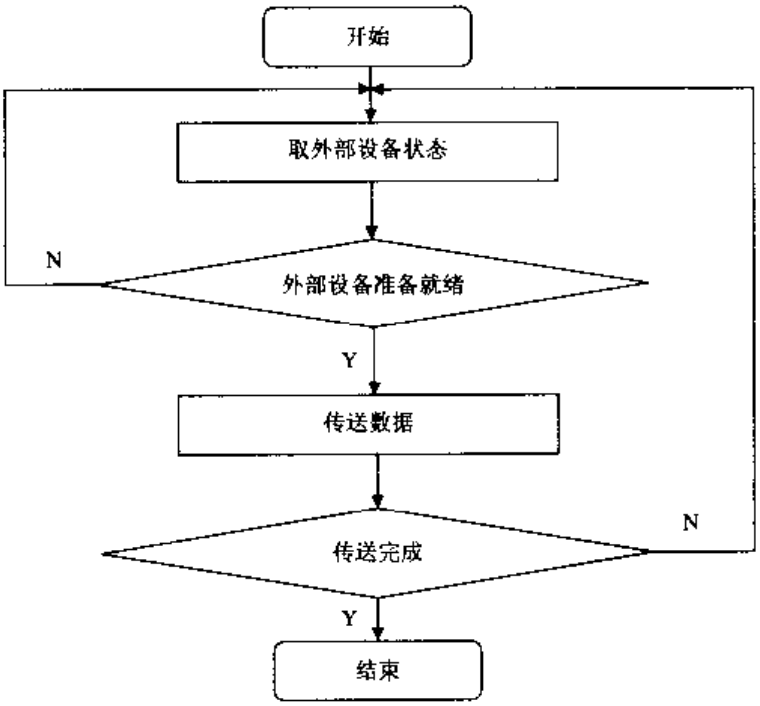


图 2-6　查询方式数据传送流程图

图 2-6 描述的是一个设备工作状态下的查询流程，对于多个设备一起工作在查询模式下，CPU 需要依次查询每个设备的状态，然后进行数据传送，如图 2-7 所示。在查询过程中 CPU 不能再做别的事，这就大大降低了 CPU 的效率。而且由于设备间的查询采用轮循，假如某一外部设备刚好在查询后马上处于就绪状态（有事务需要处理），那么它也必须等到其他设备查询并处理完成，CPU 再次查询该设备时才能发现它已处于就绪态，并对其进行服务。这样对于一些实时性要求较高的外设而言，事务不能及时（实时）响应，可能造成不可预知的后果。这种方式在现代计算机系统中已很少应用。

处理器的高速和输入/输出设备的低速是一对矛盾，是设备管理要解决的一个重要问题。为了提高整体效率，减少在程序直接控制方式中 CPU 之间的数据传送，是很有必要的。因此就需要引入其他机制（中断、DMA）来解决这种问题。

2.1.4　设备中断

大多数硬件在一些事件产生需要 CPU 注意时，都会生成一个中断请求，向 CPU 申请服务。如果系统中断允许，中断发生时，将强迫 CPU 暂停当前程序的执行，转而对产生中断的设备进行处理。这些事件称为中断源，中断源通常是：

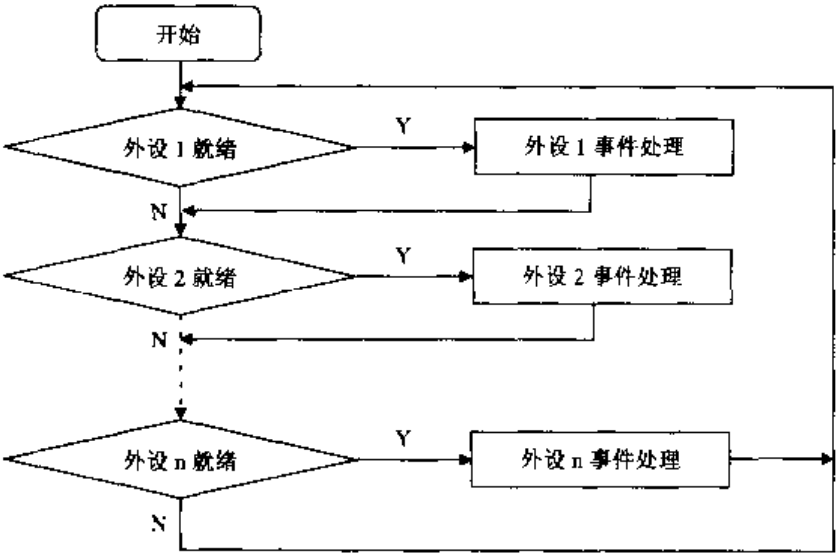


图 2-7 多个外设查询方式工作流程

（1）定时器到达。

（2）数据有效，例如：对于通信类设备是接收数据到达、FIFO 满；对于 A/D 采集类设备则是数据转换完成。

（3）设备完成了以前的请求现在空闲。例如：对于通信类设备，发送状态机变为空闲或发送 FIFO 已有足够空闲；对于 D/A 转换类设备数据转换完成，可进行下一次操作。

（4）设备与外界的连接状态发生变化，如网卡链路连接发生变化、USB 设备插入或移出、设备复位完成。

（5）设备在工作过程中出现的某种错误，如循环校验错等。

上述状态出现时，设备将产生中断（实质是将与总线连接的中断线电气信号改变），CPU 中断当前程序执行，保存现场，跳转到与该中断相连接的中断处理程序，然后执行该中断处理程序，处理该事件，处理完成恢复原被中断程序，并继续执行。为支持中断传输机制，设备需要一组完整的包含数据寄存器、状态寄存器和控制寄存器在内的寄存器组。

在设备驱动程序初始化时，系统将中断处理程序与指定的中断向量相关联。中断向量是允许 CPU 标示中断源和调用合适中断服务的唯一总线相关的数字，当 CPU 接受一个中断请求时，中断控制器通常将这个向量传递给 CPU，CPU 使用这个向量作为索引，查找含有该中断服务程序的表。

中断方式数据传输步骤如下：

（1）在某个进程需要数据时，发出指令启动输入/输出设备准备数据。

（2）在进程发出指令启动设备之后，该进程放弃处理器，等待相关 I/O 操作完成。此时，进程调度程序会调度其他就绪进程使用处理器。

（3）当 I/O 操作完成时，输入/输出设备控制器通过中断请求线向处理器发出中断信号，处理器收到中断信号之后，转向预先设计好的中断处理程序，对数据传送工作进行相应的处理。

（4）得到了数据的进程，转入就绪状态。在随后的某个时刻，进程调度程序会选中该进程继续工作。

采用中断方式，I/O 设备中断方式使处理器的利用率提高，且能支持多个程序和 I/O 设备

的并行操作。不过，中断方式仍然存在一些问题。首先，现代计算机系统通常配置有各种各样的输入/输出设备。如果这些 I/O 设备都通过中断处理方式进行并行操作，那么中断次数的急剧增加就会造成 CPU 无法响应中断和出现数据丢失现象。其次，如果 I/O 控制器的数据缓冲区比较小，在缓冲区装满数据之后将会发生中断。那么，在数据传送过程中，发生中断的机会就较多，这将耗去大量的 CPU 处理时间。

2.1.5 DMA

直接内存存取（DMA）技术是指数据在内存与 I/O 设备间直接进行成块传输。适用于高速外设与存储器进行高速数据交换，中间过程不需要 CPU 干涉，只需要 CPU 在过程开始时向设备发出"传送块数据"的命令，然后在中断到来时，判断传送过程是否结束和下次操作是否准备就绪。

DMA 工作过程描述如下：

（1）当进程要求设备输入数据时，CPU 把准备存放输入数据的内存起始地址，以及要传送的字节数分别送入 DMA 控制器中的内存地址寄存器和传送字节计数器。

（2）发出数据传输要求的进程进入等待状态，此时正在执行的 CPU 指令被暂时挂起。进程调度程序调度其他进程占据 CPU。

（3）输入设备不断地窃取 CPU 工作周期，将数据缓冲寄存器中的数据源源不断地写入内存，直到所要求的字节全部传送完毕。

（4）DMA 控制器在传送完所有字节时，通过中断请求线发出中断信号。CPU 在接收到中断信号后，转入中断处理程序进行后续处理。

（5）中断处理结束后，CPU 返回到被中断的进程中，或切换到新的进程上下文环境中，继续执行。

从上述过程可以看出，中断方式是在数据缓冲寄存器满之后发出中断，要求 CPU 进行中断处理的，而 DMA 方式则是在所要求传送的数据块全部传送结束时才要求 CPU 进行中断处理，这就大大减少了 CPU 进行中断处理的次数。中断方式的数据传送是在中断处理时由 CPU 控制完成的，而 DMA 方式则是在 DMA 控制器的控制下，不需经过 CPU 控制完成的。这就避免了 CPU 因并行设备过多而来不及处理以及因速度不匹配而造成数据丢失等现象。

在 DMA 方式中，由于 I/O 设备直接同内存发生成块的数据交换，因此 I/O 效率比较高。DMA 技术可以大大提高 I/O 效率，因此在现代计算机系统中得到了广泛的应用。许多输入/输出设备的控制器，特别是块设备的控制器、高速通信卡（网卡、SCC 控制器），都支持 DMA 传送方式。

通过上述分析可以看出，DMA 控制器功能的强弱，是决定 DMA 效率的关键因素。DMA 控制器需要为每次数据传送做大量的工作，数据传送单位的增大意味着传送次数的减少。另外，DMA 方式窃取了时钟周期，CPU 处理效率降低了，要想尽量少地窃取时钟周期，就要设法提高 DMA 控制器的性能，这样可以更少地影响 CPU 处理效率。

2.1.6 PCI 总线与设备

总线是数据线、地址线和控制线的集合，是外围设备与 CPU 通信的通道。总线的规范定义物理信号的大小和形状、每条线的功能，以及连接到总线上的设备使用的时序与信号规范。

最初由 Intel 设计的外围部件互连（PCI）总线，由于其相对处理器中立，经过十几年的发展，已成为局部总线的新标准。一个 PCI 系统的典型结构如图 2-8 所示。

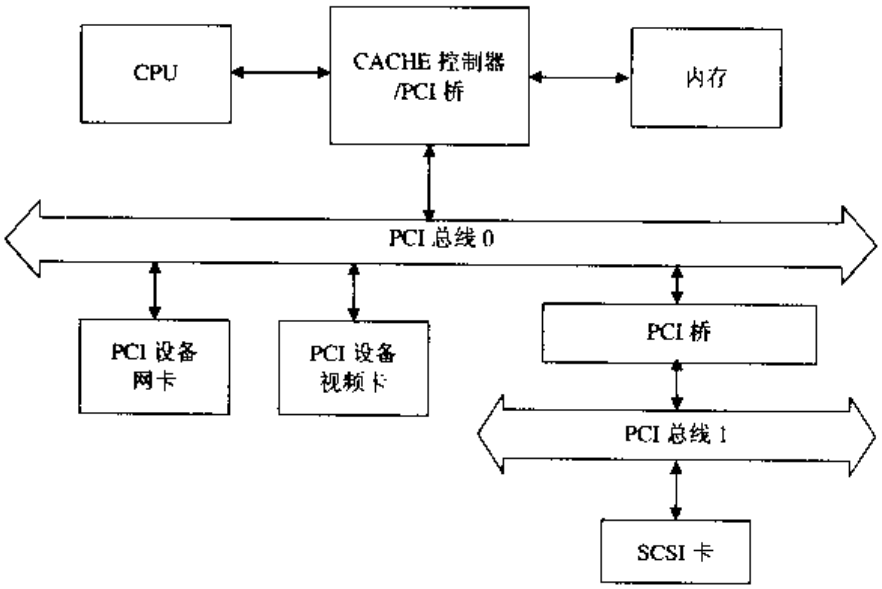


图 2-8 PCI 总线典型结构

从图中可以看出，一个 PCI 系统主要由处理器、存储单元、主机-PCI 桥（主桥）、PCI-PCI 桥、PCI-扩展总线、各种 PCI 卡及附属电路组成。PCI 总线是一种不依附于某个具体处理器的局部总线。从结构上看，PCI 是在 CPU 和原来的系统总线之间插入的一级总线，具体由一个桥接电路实现对这一层的管理，并实现上下之间的接口以协调数据的传送。管理器提供了信号缓冲，使之能支持多种外部设备，并能在高时钟频率下保持高性能。PCI 总线也支持总线主控（master）技术，允许智能设备在需要时取得总线控制权，以加速数据传送。

PCI 总线是以 ISA/EISA 等 PC 用低速总线的替代者的身份出现的，它之所以成为局部总线的主流，是由其所具有的一些显著的特点决定的，具体如下：

（1）运行速度快，可扩展性好。PCI 总线典型的工作频率为 33 MHz，支持 66 MHz 扩展。它的总线宽度为 32 位，并可以扩展到 64 位。当较多外部设备接到 CPU 总线上而使总线驱动能力不足时，可采用多条 PCI 总线，这些总线可以并发工作，每条总线上最大可以接多个 PCI 设备。

（2）兼容性好，稳定可靠。PCI 总线可与 ISA、EISA、VESA 等总线兼容，由于 PCI 规范与 CPU 及时钟无关，也就是说，PCI 的插卡是通用的，可插到任何一个有 PCI 总线的系统上去（一般对同一类型 CPU 的系统而言）。PCI 卡的通用性大大简化了系统设计，同时保证了 PCI 设备工作状态的稳定。

（3）从硬件上保证即插即用功能的实现。所谓即插即用，就是要求各种插卡插入系统就能工作，而不必手动设置开关或跳线，即有自动配置功能。PCI 总线为每个 PCI 插槽定义了相应的配置空间，一旦 PCI 卡插入系统，系统 BIOS 能根据从配置空间读到的关于该扩展卡的信息，结合系统实际情况为插卡分配内存或 I/O 地址、中断和某些定时信息，实现自动配置功能，从根本上免除了人工配置。

（4）规范标准严格。PCI 总线对协议、时序、负载、电气性能和机械性能等指标都有严格

的规定，这正是 ISA、VESA 这类总线所不及的地方，从而保证了它的可靠性和兼容性。

2.1.6.1 PCI 地址空间

PCI 地址空间是 CPU 和 PCI 设备交换信息的共享内存空间。驱动程序使用这块内存区域控制 PCI 设备并在 CPU 与 PCI 设备之间传递信息。共享内存通常包括设备的各种寄存器（控制、状态、数据寄存器）。驱动程序使用这些寄存器用来控制设备并读取其信息。

PCI 设备需要访问三种地址空间：PCI I/O、PCI 内存和 PCI 配置空间。CPU 则可以访问所有这些地址空间。PCI I/O 和 PCI 内存由设备驱动程序使用，PCI 配置空间由系统在 PCI 初始化时使用。系统在 PCI 初始化时将确定设备影射的 I/O 空间和内存空间的地址范围，如图 2-9 所示。

PCI 规范约定总线上的每个设备拥有自己的 256 个字节大小的配置空间，配置空间用于存放配置数据。配置空间的前 64 个字节可以作为一个以定义好的数据结构（称为 PCI 配置头），剩余的 192 个字节可以由板卡设计者定义。

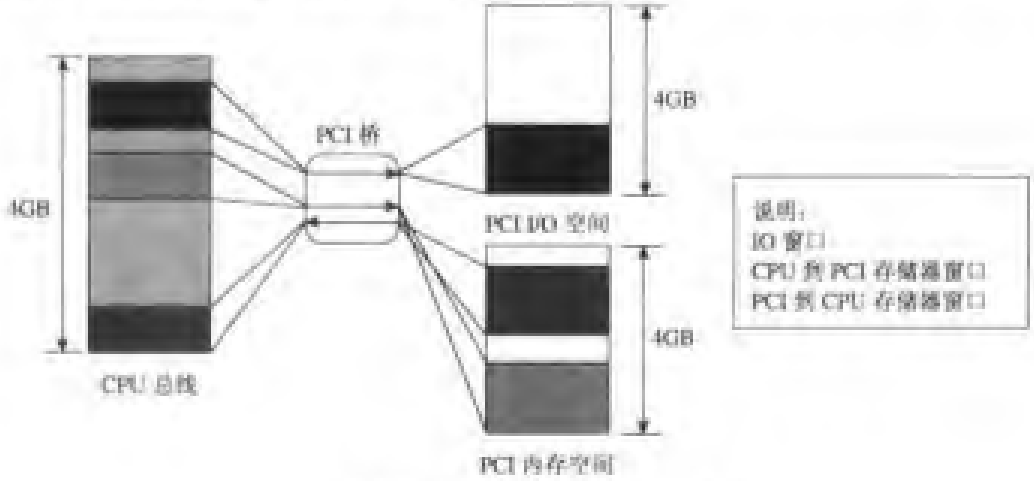


图 2-9 PCI 配置空间影射示意图

2.1.6.2 PCI 配置头

系统中每个 PCI 设备，包括 PCI-PCI 桥在内，都有一个配置数据结构，它通常位于 PCI 配置地址空间中。PCI 配置头允许系统来标识与控制设备。配置头在 PCI 配置空间的位置取决于系统中 PCI 设备的拓扑结构。通常与 PCI 槽有关，但不论配置头位置怎么变化，对系统都没什么影响，系统将找到每个 PCI 设备与桥并使用它们配置头中的信息来确定其寄存器。PCI 配置头信息定义如表 2-2 所示。

表 2-2 PCI 配置头信息定义

偏移	定义			
	31-24	23-16	15-8	7-0
0x00	DEVICE ID		VENDOR ID	
0x04	状态		命令	
0x08	类代码			版本
0x0c	BIST	Header Type	PCI Latency Timer	Cache Line Size

续表

偏移	定义			
	31-24	23-16	15-8	7-0
0x10	BAR0			
0x14	BAR1			
0x18	BAR2			
0x1c	BAR3			
0x20	BAR4			
0x24	BAR5			
0x28				
0x2c	Subsystem ID		Subsystem Vendor ID	
0x30	PCI Base Address for Local Expansion ROM			
0x34				
0x38				
0x3c			Interrupt Pin	Interrupt Line

相应域解释如下：

- 厂商标识（Vendor Identification）

由 PCI 组织分配，用来唯一标识 PCI 设备生产厂家的数值。例如，Intel 的厂商标识为 0x8086，National Semiconductor 为 0x100B。

- 设备标识（Device Identification）

用来唯一标识设备的数值。例如，ns83820 千兆以太设备的设备标识为 0x0022。

- 状态（Status）

此域提供 PCI 标准定义中此设备的状态信息。

- 命令（Command）

通过对此域的写可以控制此设备。例如，允许设备访问 PCI I/O 或内存等。

- 分类代码（Class Code）

此域标识本设备的类型。对于每种类型板卡，如显卡、网卡，SCSI 等设备都有标准的分类代码。例如，网卡设备分类代码为 0x020000。

- 基地址寄存器（Base Address Registers）

PCI 卡内有存储器、以存储器编址的寄存器和 I/O 空间，为使驱动程序能访问它们，需要申请一段存储区域将它们定位，基地址寄存器组用来决定和分配此设备可以使用的 PCI I/O 与 PCI 内存空间的类型、数量及位置。

- 中断引脚（Interrupt Pin）

PCI 卡上的四个物理引脚可以将中断信号从插卡上带到 PCI 总线上。这四个引脚标准的标记分别为 A、B、C 及 D。中断引脚域描述此 PCI 设备使用的引脚号，通常特定设备都是采用硬连接方式的。这也是在系统启动时设备总使用相同中断引脚的原因。中断处理子系统用它来管理来自该设备的中断。

- 中断连线（Interrupt Line）

设备配置头中的中断连线域用来在 PCI 初始化代码、设备驱动以及中断处理子系统间传递中断处理过程。虽然本域中记录的这个数值对于设备驱动毫无意义，但是它可以将中断处理过程从 PCI 卡上正确路由到操作系统中相应的设备驱动中断处理代码中。

2.1.6.3 PCI I/O 和 PCI 内存地址

这两个地址空间用来实现 PCI 设备与设备驱动程序之间的通信，设备驱动可以通过对这些寄存器的读写来控制此设备。在 PCI 系统建立并通过用 PCI 配置头中的命令域来打开这些地址空间前，系统不允许对它们进行存取。通常设备驱动只读写 PCI I/O 和 PCI 内存地址。BSP 相关代码可通过设置配置空间的一些寄存器配置硬件的功能。

2.2 VxWorks 设备驱动概述

设备驱动是用于控制一种设备的程序集合，它将通用控制操作转换为设备能够识别的特定操作（寄存器或 I/O 操作序列），是硬件设备的软件抽象。

2.2.1 VxWorks 驱动体系

VxWorks 驱动在系统中的位置如图 2-10 所示。

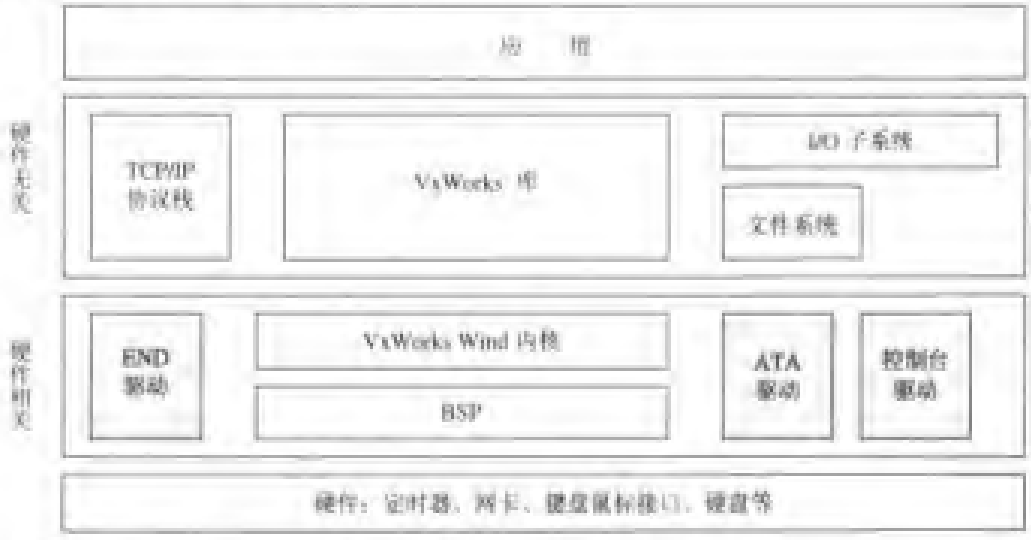


图 2-10 VxWorks 驱动体系

注意：在 VxWorks 驱动软件体系中也经常提到“设备”的概念，这是指驱动软件中用于描述硬件设备的设备描述结构（device descriptor structure）。此外，一个驱动可以支持（驱动）多个相同类型的硬件设备，对应每个硬件设备，在 VxWorks 驱动软件体系都有一个软设备。例如，通常 PC 机提供两个 i8250 串口，VxWorks 只提供一个驱动程序 i8250Serial.c，它可以驱动两个串口，当它加载成功时，I/O 系统中将出现两个设备：“/tyCo/0”和“/tyCo/1”，在 Shell 下用 devs 命令可以看到这两个设备，如图 2-11 所示。

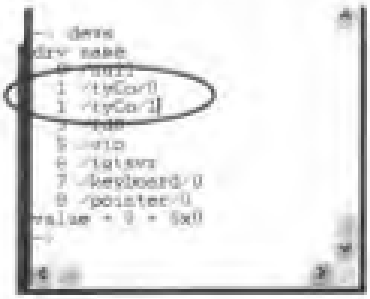


图 2-11 两个串口设备

在 VxWorks 系统中，与 Linux 相似，设备是以文件方式管

理的，也就是说，可以通过文件操作 *creat*()、*open*()、*close*()、*read*()、*write*()和 *ioctl*()等函数访问设备。硬件驱动程序统一由 I/O 子系统管理，向应用提供统一的标准。网络驱动也不例外，除了支持 SOCKET 接口外，还支持 *read*()、*write*() 等操作。

2.2.2 VxWorks 驱动分类

VxWorks 提供了一些驱动程序的示例代码，位于 target\src\drv 目录下，如图 2-12 所示。

图 2-12 target\src\drv 目录

其中比较重要的目录有：

- dma：DMA 控制器驱动。
- end：增强型网络驱动。
- hdisk：硬盘驱动。
- intrCtrl：中断控制器驱动。
- mem：存储类设备驱动。
- netif：BSD 风格网络驱动。
- pci：PCI 总线驱动及相应库函数。
- pcmcia：PCMCIA 驱动。
- sio：串行 I/O 驱动。
- timer：定时器驱动。
- vme：VME 总线驱动。

VxWorks 标准设备驱动分为以下几类：

（1）SIO_CHAN：串行设备驱动。

（2）END_OBJ：增强型网络驱动。

（3）BLK_DEV：块设备驱动。

VxWorks 提供了与这三类设备相联系的库函数，驱动可以利用对应的库函数创建相应的

设备描述符，并向系统注册自己。

本章介绍编写设备驱动需要的软件基础，第 3、4 章分别介绍串口设备驱动和增强型网络设备驱动的具体实现。

2.3 VxWorks 与驱动相关的函数

VxWorks 为方便驱动开发提供访问硬件资源和驱动管理的函数库，定义在 sysLib、intArchLib、pciConfigLib 等库中。sysLib 定义了访问与系统相关的函数，驱动中要用到的主要有端口读写、延时和中断控制器某级中断允许/禁止函数。intArchLib 定义了与体系相关的中断管理函数，驱动要用到的主要有中断向量连接管理、系统中断允许与禁止等。pciConfigLib 定义了 PCI 设备驱动要用到的 PCI 配置空间的访问函数，主要有设备查找、设备端口读写等函数。

2.3.1 寄存器读写函数

寄存器端口可以分为两类：一类是影射到 IO 空间；另一类影射到内存地址空间。两类寄存器采用不同的读写函数。影射到内存地址空间的寄存器可以直接按照存储器进行访问，映射到 I/O 空间的则要调用 sysLib 库提供的函数来访问寄存器。

2.3.1.1 存储器寄存器操作

假定一块设备寄存器组映射到内存地址范围为 0xEDAFF000～0xEDAFFFFF，大小为 0x1000。寄存器全部为 32 位。以下函数可用于读写相关寄存器：

1. 寄存器读

```
unsigned int  regMemRead(unsigned int devAdrs,int reg)
{
    ULONG *  csrReg;
    ULONG    csrData;

    csrReg = (ULONG *)(devAdrs + reg);
    csrData = *csrReg;

    return (PCISWAP (csrData));
}
```

该函数以寄存器组基地址（devAdrs）及寄存器编号（reg）为参数，返回寄存器内容。其中，PCISWAP 用于交换高低字节顺序，对于 big endian 机器，PCISWAP 定义为

```
#define PCISWAP (x) ((LLSB(x) << 24) | \
                (LNLSB(x) << 16)| \
                (LNMSB(x) << 8) | \
                (LMSB(x)))
```

对于 little endian 机器，PCISWAP 定义为

```
#define PCISWAP (x) (x)
```

2. 寄存器写

```
void regMemWrite
```

```
(
unsigned int devAdrs,
int      reg,
unsigned int value
)
{
unsigned int *  csrReg;

csrReg = (unsigned int *)(devAdrs + reg);
*csrReg = PCISWAP (value);

return;
}
```

该函数以寄存器组基地址（devAdrs）、寄存器编号（reg）和准备写到寄存器的值（value）为参数。

3. 寄存器位操作

操作寄存器时有时需要对某一位单独进行操作（设为 1 或 0），其他位保持不变。函数 regBitSet 将完成这种操作。

```
    void regBitSet (unsigned int regBase,int reg,int bit, int value)
{
    unsigned int data;
    data= regMemRead(regBase,reg);
    if(value==1)          /* 将位设置为 1 */
        data=data|(1<<bit);
    else                  /* 将改位清除 */
        data=data&(~((1<<bit)));
    regMemWrite(regBase,reg,data);
}
```

2.3.1.2 I/O 空间寄存器读写操作

I/O 空间操作需要使用系统提供的 I/O 指令，VxWorks 将这些 I/O 指令封装成 C 语言可调用接口，放在 sysLib 库中。具体介绍如下：

1. 寄存器读函数

```
UCHAR sysInByte(int port)
USHORT sysInWord(int port)
ULONG sysInLong(int port)
```

这三个函数都以 I/O 地址作为参数，分别用于读 8 位、16 位和 32 位 I/O 寄存器的值。

2. 寄存器写函数

```
void sysOutByte(int port, UCHAR data)
void sysOutWord(int port, UWORD data)
void sysOutLong(int port, ULONG data)
```

这三个函数都以 I/O 地址和准备写到寄存器的值作为参数，分别用于向 8 位、16 位和 32 位 I/O 寄存器写数据。

一些硬件寄存器连续访问时需要一定的总线恢复时间，VxWorks 提供了函数 sysDelay()可

用于连续读写寄存器操作时短暂的延时。

2.3.2 中断管理函数

与中断相关的函数主要有中断连接、中断允许和中断禁止等，具体介绍如下：

2.3.2.1 中断处理程序连接函数

```
STATUS intConnect
    (
    VOIDFUNCPTR * vector,    /* interrupt vector to attach to */
    VOIDFUNCPTR   routine,   /* routine to be called */
    int           parameter  /* parameter to be passed to routine */
    )
```

中断连接函数，将一个中断处理程序与一个中断向量相连接，vector 是中断向量，routine 是中断处理程序，parameter 是传给中断处理程序的参数。中断向量存放中断处理程序的入口。VxWorks 提供宏 INUM_TO_IVEC（irq）用于在中断号与中断向量间的转化，其中 irq 为中断号，该宏返回 irq 对应的中断向量。注意：这里的中断号与我们平时提到的硬件设备的使用的中断号（例如在 PCI 头中查到的中断号）并不一致，通常要加上一个常量。

例如对于 X86 体系，这个常量为 0x20，第一个串口中断为 4，中断号与中断向量间的转化为 INUM_TO_IVEC（4+0x20）。假定其中断处理程序名称为 comIntHandle，没有参数，其中断连接语句为：

```
intConnect (INUM_TO_IVEC(4+0x20), comIntHandle,0);
```

由于中断处理程序执行时，禁止中断，也就不允许发生任务的调度，因此，中断服务程序在设计时必须尽可能地短，通常不允许调用可能引起阻塞的函数，例如获取信号量 semTake、内存申请分配 malloc、格式化输出 printf 等函数。中断信息输出可以调用 logMsg。

典型的中断处理程序流程是：进中断后读中断状态寄存器，然后判断中断状态，并清中断标志，接着调用 semGive 释放信号量，最后激活相应任务处理后续工作。

中断服务程序还可以采用共享内存、消息队列等与任务通信。

2.3.2.2 中断允许与禁止函数

硬件设备在初始化过程中，可能会产生中断，系统将进入中断处理程序。为避免这种情况发生，通常要求禁止该级别的中断产生，这时可以调用 sysIntDisablePIC 和 sysIntEnablePIC 来完成。通常要求二者成对出现。

二者分别用于禁止和使能指定级别的中断，以中断的级别作为参数。例如，如下代码用于 PC 机第一个串口芯片（中断号为 4）初始化过程中禁止中断和允许中断。

```
sysIntDisablePIC(4);
/* 以下为串口硬件初始化 */
...

/* 初始化完成，允许中断 */
sysIntEnablePIC(4);
```

此外，为防止硬件初始化时产生中断，驱动程序在设计时，要求先调用 intConnect 连接中断处理程序，然后再进行硬件初始化操作。

当程序访问临界区时，要求排它访问，不允许被打断，要求禁止系统中所有中断，可以调用 intLock、intUnlock 完成。intLock 禁止系统中断（封中），intUnlock 则重新允许系统中断。intLock、intUnlock 要求成对出现，后者以前者的返回值作为参数，示例代码如下：

```
int ss;
ss=intLock();
/* 以下是封中操作 */
...
/* 封中操作结束，允许中断 */
intUnlock(ss);
```

2.3.3 PCI 配置空间访问函数

前面已经提到 PCI 设备使用的硬件资源信息由系统 PCI 总线初始化时分配，并将其存放在设备的配置空间内。因此 PCI 设备驱动程序首先要定位设备的配置空间，然后从中获取其资源信息。VxWorks 的 pciConfigLib 库提供定位与访问 PCI 设备配置空间的函数。

2.3.3.1 设备查找函数

由于每个 PCI 设备都有唯一的 vendorID 和 deviceID，因此可以以此作为参数来定位 PCI 设备。函数 pciFindDevice 就是使用 vendorID 和 deviceID 来查找 PCI 设备的。其原型如下：

```
STATUS pciFindDevice
    (
    int   vendorId,          /* vendor ID */
    int   deviceId,          /* device ID */
    int   index,             /* desired instance of device */
    int * pBusNo,            /* bus number */
    int * pDeviceNo,         /* device number */
    int * pFuncNo            /* function number */
    )
```

其中：index 是说明需要查找的设备序号的。这是用于当系统中可能同时存在多个相同的 PCI 设备时，PCI 在初始化时从 0 开始顺序分配给每个设备一个索引号。

该函数返回对应设备的 bus、device、Func 号（总线号、设备号、功能号），这三个参数将唯一标识一个 PCI 设备。驱动程序可以使用 bus、device、Func 号来读写该 PCI 设备的配置空间寄存器。

此外，PCI 规范将 PCI 设备按功能分类，如网卡、显卡等等，每类设备指定一个唯一的编号，驱动程序也可根据设备所属类号来查找设备。pciFindClass 可以实现依据类号来查找设备，USB 主机控制驱动就是采用这种方式来查找设备的。函数原型如下。

```
STATUS pciFindClass
    (
    int   classCode,         /* 24-bit class code */
    int   index,             /* desired instance of device */
    int * pBusNo,            /* bus number */
    int * pDeviceNo,         /* device number */
    int * pFuncNo            /* function number */
    )
```

2.3.3.2 PCI 配置空间寄存器访问函数

以下函数用来访问 PCI 设备的配置空间寄存器：

pciConfigInByte( )：从 PCI 配置空间指定偏移位置读一个字节。

pciConfigInWord( )：从 PCI 配置空间指定偏移位置读一个短字（双字节）。

pciConfigInLong( )：从 PCI 配置空间指定偏移位置读一个长字（四字节）。

pciConfigOutByte( )：向 PCI 配置空间指定偏移位置写一个字节。

pciConfigOutWord( )：向 PCI 配置空间指定偏移位置写一个短字（双字节）。

pciConfigOutLong( )：向 PCI 配置空间指定偏移位置写一个长字（四字节）。

例如，函数 pciConfigOutLong 原型为：

```
STATUS pciConfigOutLong
    (
    int    busNo,            /* bus number */
    int    deviceNo,         /* device number */
    int    funcNo,           /* function number */
    int    offset,           /* offset into the configuration space */
    UINT32 data              /* data written to the offset */
    )
```

其中前三个参数分别为总线号、设备号、功能号（用以定位 PCI 设备），offset 为寄存器在配置空间的偏移，data 是要写入的数据。

函数 pciConfigInLong 的原型为：

```
STATUS pciConfigInLong
    (
    int      busNo,          /* bus number */
    int      deviceNo,       /* device number */
    int      funcNo,         /* function number */
    int      offset,         /* offset into the configuration space */
    UINT32 * pData           /* data read from the offset */
    )
```

其中前四个参数与 pciConfigOutLong 意义相同，pData 是一个整形指针，存放读回的数据。

2.3.3.3 PCI 设备 BSP 初始化

以下代码查找系统中第一个 PCI 接口的 Intel82557 网卡设备，获得配置空间数据（IO 地址、PCI 存储器地址、中断号），并映射 PCI 内存，使能 I/O、存储器访问和总线 master 方式，可用作大部分 PCI 设备 BSP 初始化。

```
#include <vxworks.h>
#include "vmLib.h"
#include "sysLib.h"
#include "drv/pci/pciIntLib.h"

#define VENDORID  0x8086
#define DEVICEID  0x8086
#define DEVICEUNIT 0

STATUS i557PciInit()
```

```
{
    UINT32  pciBus,          /* store a PCI bus number */
    UINT32  pciDevice,       /* store a PCI device number */
    UINT32  pciFunc,         /* store a PCI function number */

    UINT32  memIo;           /* memory-mapped IO address (BAR 0) */
    UINT32  ioBase;          /* IO base address (BAR 1) */
    UINT8   irq;             /* interrupt line number (IRQ) for device */

    if (pciFindDevice (VENDORID, DEVICEID, DEVICEUNIT,
                &pciBus, &pciDevice, &pciFunc) == OK){
        pciConfigInLong  (pciBus, pciDevice, pciFunc,
                    PCI_CFG_BASE_ADDRESS_0, &memIo);
        pciConfigInLong  (pciBus, pciDevice, pciFunc,
                    PCI_CFG_BASE_ADDRESS_1, &ioBase);

        memIo &= PCI_MEMBASE_MASK;
        ioBase  &= PCI_IOBASE_MASK;

    /* map a 4kb 32-bit non-prefetchable memory IO address decoder */

    if (sysMmuMapAdd ((void *)(memIo & PCI_DEV_MMU_MSK),
        PCI_DEV_ADRS_SIZE, VM_STATE_MASK_FOR_ALL, VM_STATE_FOR_PCI) == ERROR)
        {
        return (ERROR);
        }
    /* read the IRQ number and vector and save to the resource table */
        pciConfigInByte (pciBus, pciDevice, pciFunc,
                    PCI_CFG_DEV_INT_LINE, &irq);

    /* enable mapped memory and IO decoders */

    pciConfigOutWord (pciBus, pciDevice, pciFunc, PCI_CFG_COMMAND,
                    PCI_CMD_MEM_ENABLE | PCI_CMD_IO_ENABLE |
                    PCI_CMD_MASTER_ENABLE);

    /* disable sleep mode */

    pciConfigOutByte (pciBus, pciDevice, pciFunc, PCI_CFG_MODE,
                    SLEEP_MODE_DIS);
    }
}
```

第3章 字符驱动程序设计

串行接口是一种相对简单而又较为常用的通信接口。本章以i8250串口为例介绍VxWorks字符类驱动程序的设计。首先介绍i8250硬件原理，然后是一个非标准驱动的设计过程，最后是标准字符驱动程序设计与实现。

3.1 i8250芯片简介

3.1.1 i8250工作原理

串行接口适配器由地址译码器、时钟电路、16550/i8250 UART、数据缓冲器、发送接收驱动器和RS-232插座等组成，如图3-1所示。

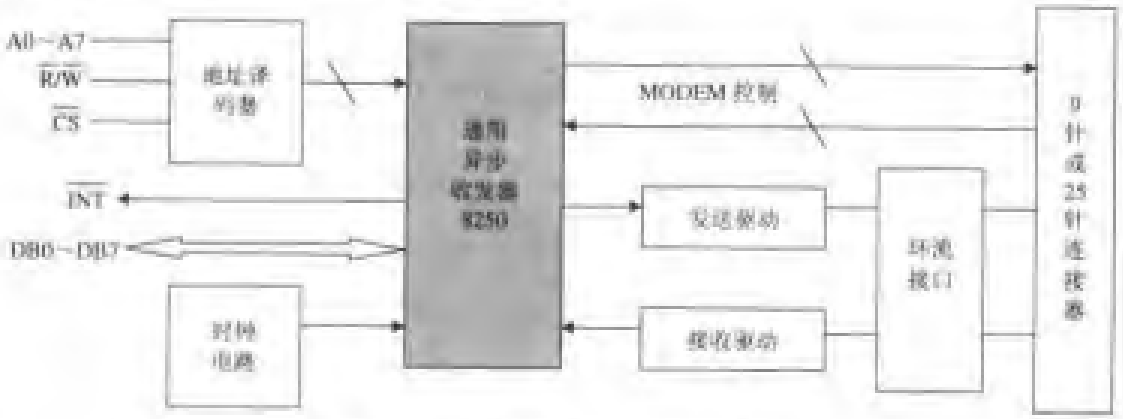


图3-1 串行接口适配器组成示意图

地址译码器接收系统发出的寻址信息，并使能有关的端口寄存器，进行读写操作。

i8250是用于串行数据通信的通用异步收发器（UART），它与外界接口的线驱动芯片耦合，完成异步通信功能，在发送时将并行数据转换成串行数据，接收时将串行数据转换成并行数据。i8250隐藏了用START/STOP位组成的数据帧，产生奇偶校验位，保证所有数据以合适的速率移出和移入。软件可以通过读写数据寄存器，完成数据的输入和输出。通常可处理速率在1200~115200bps的串行数据。

（1）输出过程：把数据写入发送数据寄存器，i8250将该数据送到一个移位寄存器，由该寄存器逐位将数据发送到串行线上。当移位寄存器为空时，i8250将其TBE（发送缓冲区空）标志置位。

（2）输入过程：i8250接收器一直监视串行输入线状态。当数据位出现时，它们被送入一个移位寄存器，该寄存器将这些数据位组装为一个完整的字节数据，并置数据寄存器RxRDY（接收数据有效）标志，指示数据有效。软件就可通过读取数据寄存器获得该字节数据，该标志清除。

数据发送或接收完成，如果对应的中断使能位允许，i8250 将产生中断，通知 CPU。中断处理程序可以读取中断状态寄存器，判断芯片状态，并做出相应处理。

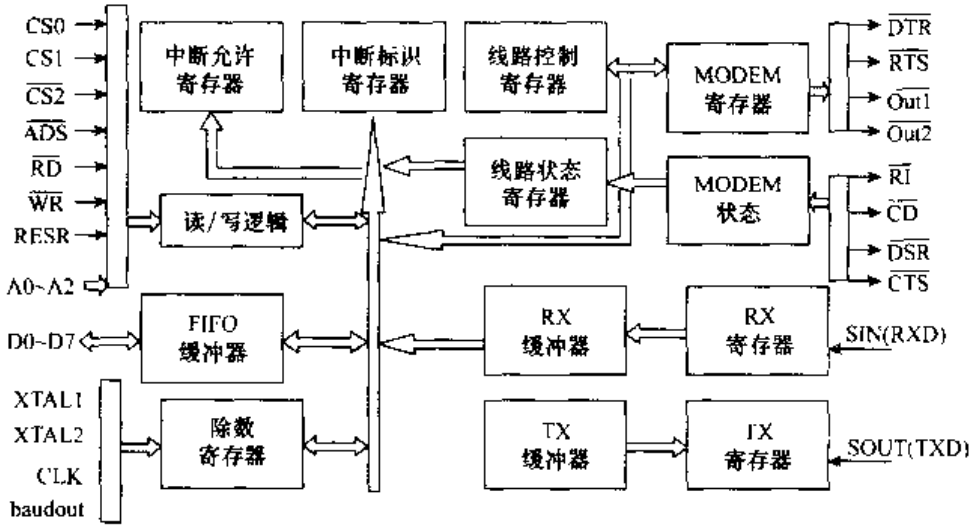


图 3-2 i8250 结构框图

3.1.2 i8250 寄存器定义

i8250 内部有 12 个单字节寄存器，占用 8 个字节的 I/O 地址空间分别是数据发送寄存器、数据接收寄存器、波特率除数锁存器寄存器低字节（LSB）、波特率除数锁存器寄存器高字节（MSB）、中断使能寄存器、中断识别寄存器、FIFO 控制寄存器、线控制寄存器、Modem 控制寄存器、线状态寄存器、Modem 状态寄存器和标签寄存器。地址分配如表 3-1 所示。

表 3-1 i8250 寄 存 器

偏移	名　　称	读/写	备　　注
0	数据发送寄存器	写	线控制寄存器最高位为 0 时
0	数据接收寄存器	读	线控制寄存器最高位为 0 时
0	波特率除数锁存器低字节（LSB）	写	线控制寄存器最高位为 1 时
1	波特率除数锁存器高字节（MSB）	写	线控制寄存器最高位为 1 时
1	中断使能寄存器	写	
2	中断识别寄存器	读	
2	FIFO 控制寄存器	写	
3	线控制寄存器	写	
4	Modem 控制寄存器	写	
5	线状态寄存器	读	
6	Modem 状态寄存器	读	
7	标签寄存器（暂存器）	读写	

下面简单介绍每个寄存器的定义，具体的说明请参考相应的数据手册。

（1）数据发送寄存器。在线控制寄存器最高位（除数锁存位）为 0 时，向该寄存器写入数据，将串行输出。

（2）数据接收寄存器。在线控制寄存器最高位（除数锁存位）为 0 时，从该寄存器读，将读入 i8250 接收到的字节数据。

（3）波特率除数锁存器低字节（LSB）。当线控制寄存器最高位（除数锁存位）为 1 时，向该寄存器写入数据，将作为除数锁存器的低 8 位。

（4）波特率除数锁存器高字节（MSB）。当线控制寄存器最高位（除数锁存位）为 1 时，向该寄存器写入数据，将作为除数锁存器的高 8 位。

（5）中断使能寄存器。该寄存器控制 i8250 的中断行为。

表 3-2　i8250 中断使能寄存器位定义

位	意　　义	备　　注
0	1 表示接收中断允许；0 表示接收中断禁止	
1	1 表示发送中断允许；0 表示发送中断禁止	
2	1 表示线路中断允许；0 表示线路中断禁止	
3	1 表示 Modem 中断允许；0 表示 Modem 中断禁止	
4	0	
5	0	
6	0	
7	0	

（6）中断识别寄存器。该寄存器由于判断中断源，Bit0 指示有无中断悬挂，中断源由 bit1～3 决定。i8250 支持四种中断源，按照优先级从高到低分别是：

1）线路状态中断。

2）接收数据可用中断。

3）发送缓冲区空中断。

4）Modem 状态中断。

表 3-3　i8250 中断识别寄存器位定义

中断识别寄存器				说　　明		
Bit3	Bit2	Bit1	Bit0	优先级	中断源	清中断条件
0	0	0	1		无中断	
0	1	1	0	1	接收线状态	读线状态
0	1	0	0	2	接收数据可用	读数据接收寄存器
1	1	0	0	2	接收数据可用（超时）	读数据接收寄存器
0	0	1	0	3	发送缓冲区空	读中断识别寄存器或向数据发送寄存器写
0	0	0	0	4	Modem 状态中断	读 Modem 状态寄存器

（7）线控制寄存器。该寄存器控制数据在线路上串行传输，决定数据位、停止位、校验位的长度和校验方式等。

表 3-4 i8250 线控制寄存器位定义

位	意　义	备　注
0，1	这两位控制数据位数 00 表示五位数据位；01 表示六位数据位； 10 表示七位数据位；11 表示八位数据位	
2	0 表示 1 位停止位； 1 表示当五位数据位时，表示 1.5 位停止位； 当六或七或八位数据位时，表示 2 位停止位	
3	1 表示奇偶校验允许；0 表示奇偶校验禁止	
4	1 表示偶校验；0 表示奇校验	
5	0	
6	0	
7	1 表示除数锁存器允许；0 表示除数锁存器禁止	

（8）Modem 控制寄存器。MODEM 控制寄存器可以用程序的方式设置信号的电平，如数据终端准备就绪信号 DTR、请求传送信号 RTS 的电平高低，都可以通过置位或复位该寄存器来实现。另外，MODEM 控制寄存器的位 2 和位 3 直接控制 OUT1、OUT2 引脚上的电平高低。MODEM 控制寄存器位 4 用于选择是否对来自通信设备的信号进行测试。

表 3-5 i8250Modem 控制寄存器位定义

位	意　义	备　注
0	数据终端就绪 0 表示 DTR 引脚为 1；1 表示 DTR 引脚为 0	
1	请求发送引脚 0 表示 RTS 引脚为 1；1 表示 RTS 引脚为 0	
2	OUT1 引脚 0 表示 OUT1 引脚为 1；1 表示 OUT1 引脚为 0	
3	OUT2 引脚 0 表示 OUT2 引脚为 1；1 表示 OUT2 引脚为 0	
4	1 表示自绕回模式使能；0 表示自绕回模式禁止	
5		
6		
7		

（9）线状态寄存器。该寄存器提供线路状态指示。

表 3-6 i8250 线状态寄存器位定义

位	意　义	备　注
0	数据就绪位 0 表示无接收数据；1 表示接收数据有效	
1	Overrun 状态 0 表示无 Overrun 错误；1 表示 Overrun 错误	
2	奇偶校验 0 表示无奇偶错误；1 表示奇偶错误	
3	帧错误 0 表示无帧错误；1 表示帧出错	
4	数字间隔允许 1 表示数字间隔；0 表示无数字间隔	
5	发送缓冲区空 1 表示空；0 表示非空	
6	发送移位寄存器空 1 表示空；0 表示非空	
7	接收 FIFO 状态 0 表示无错误；1 表示出错	

（10）Modem 状态寄存器。MODEM 状态寄存器用于检测通信设备发出的信号状态。

3.1.3 i8250 工作过程

为使 i8250 能正确工作，首先要对其硬件进行初始化，如果采用中断方式，在产生中断时就要判断中断源，作相应处理；如果采用查询方式，则要周期性地或必要时查询硬件状态，作相应处理。

初始化步骤：

（1）需要根据应用具体需求，根据波特率写对应的值到除数锁存器。

（2）根据通信协议（数据位、停止位和校验方式）写对应的值到线控制字寄存器。

（3）写 Modem 控制寄存器。

（4）根据工作方式是否采用中断模式，写相应的值到中断使能寄存器。

采用中断模式，当中断产生时，读中断识别寄存器，并判断中断源，转入相应处理，然后清中断源。

对于发送，第一个字符需要在发送处理中直接写到发送数据寄存器，发送空中断的产生条件是发送缓冲区空或发送移位寄存器空，当该中断产生时，如果还有数据等待发送，则将待发送数据写到发送数据寄存器，否则退出（读中断识别寄存器即清除了发送中断）。对于接收中断，读接收数据寄存器，读取接收到的字符，送给上层应用，并读线状态寄存器，判断是否还有接收数据可用，如有重复上述接收过程，如无则退出（读线状态和接收数据寄存器即清除了接收中断）。如果是线状态错误中断，则读线状态寄存器清除线状态错误中断。Modem 中断则读 Modem 状态寄存器清除相应中断。

查询模式下，需要循环读线状态寄存器和 Modem 状态寄存器，判断收发状态和 Modem 状态，并做相应处理。

3.2 非标串口驱动设计

3.2.1 非标驱动概述

非标驱动的设计类似于 DOS 下的驱动，不受驱动体系约束，根据上层应用的需求提供接口；另一方面，由于 VxWorks 提供了良好的中断机制和缓冲区管理手段，其非标驱动开发要比在 DOS 下开发方便一些。

一个非标驱动程序通常可分为包括三部分：初始化、中断处理和与应用接口函数。

（1）初始化需要完成的工作主要有驱动软件必要的数据初始化、连接中断处理程序、硬件的工作方式初始化等。

（2）中断处理程序完成对硬件中的响应，判断中断原因，清除中断，中断处理自身通常不做过多的事情，而是启动相应的任务完成后续的处理，类似于 Windows 中的中断延期处理机制。

（3）与应用的接口函数提供应用与硬件进行交互的接口。对于通信类硬件，通常包括数据接收、数据发送、硬件状态查询与设置等等。

下面首先介绍通信类驱动通常要用到的 VxWork 提供的环形缓冲区库 rngLib，然后以 i8250 串口芯片驱动为例，介绍非标驱动的设计过程。

3.2.2 环形缓冲区与 rngLib 库

应用程序可以通过通信类设备与外部交换数据。对于 i8250 等串口芯片，数据口一次只能写一个字节，而通信应用通常以报文形式进行，报文通常由长度不等的多个字节组成，如果让应用程序一个字节一个字节地逐字节方式发送显然不可取。为实现应用层的多字节报文方式收发，就需要驱动层提供一个缓冲，存放已接收或待发送的报文。这种缓冲区通常采用环形缓冲区。

所谓环形缓冲区就是一个 FIFO（先进先出）环形缓冲，如图 3-3 所示。

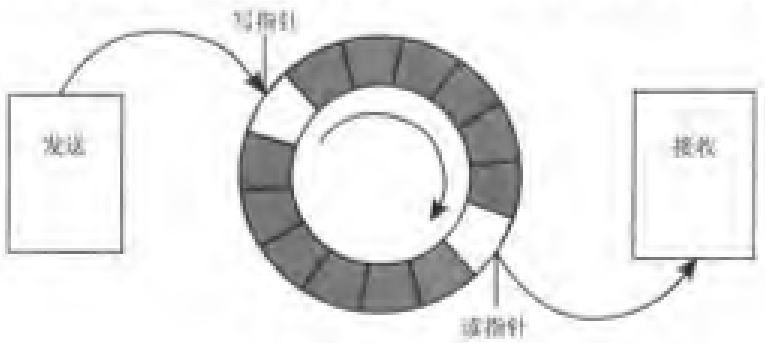


图 3-3 环形缓冲区示意图

采用 C 语言实现时，可用一个一定长度的数组和两个分别指定“读”、“写”指针位置的整形变量，提供相应的函数操作时还需要考虑到多个读写操作互斥的情形，具体实现也有一定

的难度。为方便编程，VxWorks 操作系统自身提供了一个环形缓冲管理的接口库 rngLib，为开发字符设备驱动提供了方便。

RING_ID 是 VxWorks 提供的一种环形缓冲区数据结构，相关的定义在 rngLib.h 中，定义如下：

```
typedef struct      /* RING - ring buffer */
    {
    int pToBuf;     /* offset from start of buffer where to write next */
    int pFromBuf;   /* offset from start of buffer where to read next */
    int bufSize;    /* size of ring in bytes */
    char *buf;      /* pointer to start of buffer */
    } RING;
typedef RING *RING_ID;
```

rngLib 提供的函数说明如表 3-7 所示。

表 3-7 rngLib 提供的函数

函　数	说　明
rngCreate	创建一个空的环形缓冲区
rngDelete()	删除一个环形缓冲区
rngFlush()	清空一个环形缓冲区
rngBufGet()	从一个环形缓冲区中读取一定长度的字符
rngBufPut()	向一个环形缓冲区中写入一定长度的字符
rngIsEmpty()	测试一个环形缓冲区是否为空（没有可用数据）
rngIsFull()	测试一个环形缓冲区是否已满（没有空间）
rngFreeBytes()	获得一个环形缓冲区可用的自由空间字节数
rngNBytes()	获得一个环形缓冲区可用数据的字节数
rngPutAhead()	将一个字节数据放置到环形缓冲区队头，而不移动对写指针
rngMoveAhead()	将写指针向前移动 n 个字节

函数原型及说明参见相应的帮助手册。

3.2.3 i8250 非标串口驱动实现

本节以 PC 机常用的 i8250 串口芯片为例介绍非标驱动程序的设计，将用到上面提到的 rngLib 相关函数。需要注意的是，作为例子仅仅介绍一个用于数据通信的串口驱动，并非一个完整的终端驱动。

该程序包括以下五个函数：

- i8250Init()　　驱动程序初始化
- i8250Int()　　中断处理
- i8250Recv()　　应用接收数据函数
- i8250Send()　　应用发送数据函数

- i8250IoCtrl()　　设置i8250工作方式或读取i8250信息

采用两个环形缓冲区管理接收和发送数据。

```
RING_ID         recvRing
RING_ID         sendRing
```

i8250驱动的控制结构如下：

```
typedef struct      /* i8250_CTRL */
{
    int             baseAddr;
    int             intLevle;
    RING_ID         recvRing;
    RING_ID         sendRing;
    int             bateRate;
    UCHAR           lcr;
    int             sendFlag;
} I8250_CTRL;
```

其中：

baseAddr：标记串口芯片占用的基地址，对于PC机，串口的基地址可在CMOS中设置。默认设置第一个串口基地址为0x3f8；第二个串口基地址为0x2f8。

intLevel：标记串口占用的中断级，对于PC机，串口的中断级同样可在CMOS中设置。默认设置第一个串口使用4号中断；第二个串口使用3号中断。

recvRing：RING_ID类型，接收缓冲区指针。

sendRing：RING_ID类型，发送缓冲区指针。

bateRate：记录当前串口设置的波特率。

lcr：记录当前串口设置的线控制字。

sendFlag：正在发送标志，表示中断处理程序正在发送数据。

驱动的工作原理如图3-4所示。

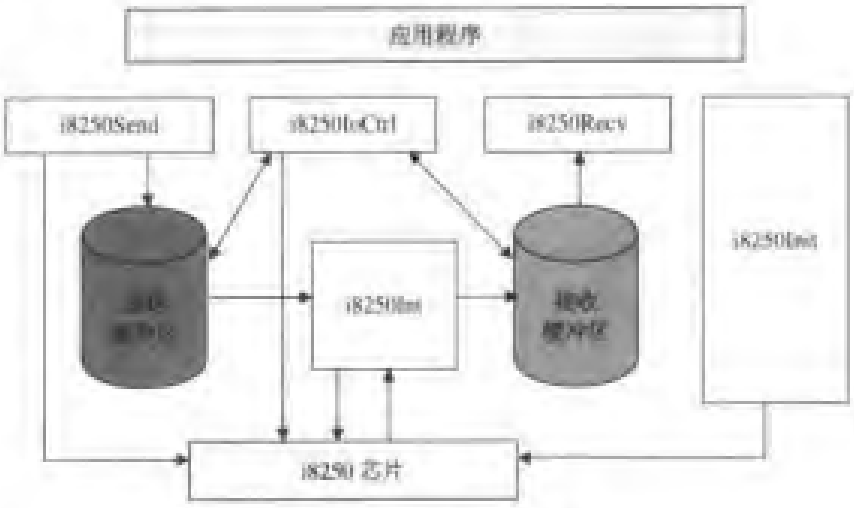


图3-4　环形缓冲区示意图

应用程序调用i8250Init对驱动程序进行初始化，i8250Init完成驱动用到的数据结构初始化，连接硬件中断，完成对i8250硬件的初始化工作，允许接送与发送空中断。

当 i8250 收到数据时，产生接收中断，操作系统中断机制调用中断处理程序 i8250Int，i8250Int 判断中断是接收中断，读数据寄存器，将接收到的数据送到接收缓冲区。

应用程序需要接收数据时，调用 i8250Recv 从接收缓冲区中读取数据；对于发送，应用程序调用 i8250Send，判断发送中断处理标志是否已置，如果发送中断处理标志已置，则将需要发送的字符串送到发送缓冲区；如果未置，i8250Send 将需要发送的字符串，除第一个字节外送到发送缓冲区，然后将第一个字节写入数据寄存器，该字节发送完成将触发发送空中断。中断发生时，操作系统中断机制调用中断处理程序 i8250Int，i8250Int 判断是否发送空中断，如果发送缓冲区非空，置发送中断处理标志，并从发送缓冲区读一个字符送到数据寄存器；如果发送缓冲区为空，清发送空中断和送中断处理标志。对于其他中断，中断处理程序简单地清除相应中断。

系统运行过程中，应用程序可以根据需要调用 i8250IoCtrl 来改变 i8250 的工作方式，如设置新的波特率、新的方式字等，也可调用 i8250IoCtrl 获得当前接收或发送缓冲区的信息。

下面详细介绍代码实现。

3.2.3.1 头文件 i8250.h

头文件 i8250.h 包含以下内容：

（1）串口设备控制信息类型定义。

（2）寄存器偏移地址及每个寄存器的详细组成定义。

（3）驱动和应用程序用到的一些常量宏定义。

```
/* i8250.h */
#ifndef __INCi8250h
#define __INCi8250h

#ifdef __cpluspius
extern "C" {
#endif

#include "vxWorks.h"
#include "rnglib.h"
/* 数据结构：串口设备控制信息 */

typedef struct  /* I8250_CTRL */
{
    int         baseAddr;               /* 串口基地址：0x3f8，0x2f8 */
    int         intLevel;               /* 中断级：4 or 3 */
    RING_ID     recvRng;                /* 接收缓冲区指针 */
    RING_ID     sendRng;                /* 发送缓冲区指针 */
    int         baudRate;               /* 波特率 */
    UCHAR       lcr;                    /* 线控制字 */
    int         sendFlag;               /* 正在发送标志 */
} I8250_CTRL;

/* 寄存器定义 */

#define UART_THR                0x00    /* 数据发送寄存器 */
```

```
#define UART_RDR            0x00    /* 数据接收寄存器 */
#define UART_BRDL           0x00    /* 波特率除数锁存器器低字节(LSB) */
#define UART_BRDH           0x01    /* 波特率除数锁存器高字节(MSB) */
#define UART_IER            0x01    /* 中断使能寄存器 */
#define UART_IID            0x02    /* 中断识别寄存器*/
#define UART_LCR            0x03    /* 线控制寄存器 */
#define UART_MDC            0x04    /* modem控制寄存器 */
#define UART_LST            0x05    /* 线状态寄存器 */
#define UART_MSR            0x06    /* Modem状态寄存器 */

/* 中断使能寄存器位定义 */

#define I8250_IER_RXRDY     0x01    /* 接收数据就绪 */
#define I8250_IER_TBE       0x02    /* 发送缓冲区空 */
#define I8250_IER_LST       0x04    /* 线状态中断 */
#define I8250_IER_MSI       0x08    /* modem 状态中断 */

/* 中断识别寄存器位定义 */

#define I8250_IIR_IP        0x01    /* 是否有中断，1：无中断，0：有中断 */
#define I8250_IIR_MASK      0x07    /* 识别位屏蔽 */
#define I8250_IIR_MSTAT     0x00    /* moder 状态中断 */
#define I8250_IIR_THRE      0X02    /* 发送缓冲区空 */
#define I8250_IIR_RBRF      0x04    /* 接收缓冲区空中断 */
#define I8250_IIR_SEOB      0x06    /* 出错或break */

/* 线控制寄存器位定义 */

#define I8250_LCR_CS5       0x00    /* 5 位数据位 */
#define I8250_LCR_CS6       0x01    /* 6位数据位 */
#define I8250_LCR_CS7       0x02    /* 7位数据位 */
#define I8250_LCR_CS8       0x03    /* 8位数据位 */
#define I8250_LCR_2_STB     0x04    /* 2 停止位 */
#define I8250_LCR_1_STB     0x00    /* 1停止位*/
#define I8250_LCR_PEN       0x08    /* 校验使能 */
#define I8250_LCR_PDIS      0x00    /* 校验禁止 */
#define I8250_LCR_EPS       0x10    /* 奇偶校验：1：偶，0：奇 odd */
#define I8250_LCR_SP        0x20    /* stick parity select */
#define I8250_LCR_SBRK      0x40    /* break 控制位 */
#define I8250_LCR_DLAB      0x80    /* 除数锁存访问使能位 */

/* modem 控制寄存器位定义 */

#define I8250_MCR_DTR       0x01    /* DTR输出 */
#define I8250_MCR_RTS       0x02    /* RTS 输出 */
#define I8250_MCR_OUT1      0x04    /* output #1 */
#define I8250_MCR_OUT2      0x08    /* output #2 */
#define I8250_MCR_LOOP      0x10    /* loop back 方式使能(自绕回方式)*/

/* 线状态存器位定义 r */
```

```
#define I8250_LSR_RXRDY         0x01    /* 接收数据可用 */
#define I8250_LSR_OE            0x02    /* overrun 错 */
#define I8250_LSR_PE            0x04    /* 校验 error */
#define I8250_LSR_FE            0x08    /* 帧错 */
#define I8250_LSR_BI            0x10    /* break 中断 */
#define I8250_LSR_THRE          0x20    /* 发送寄存器空 */
#define I8250_LSR_TEMT          0x40    /* 发送空 */

/* modem 状态寄存器位定义 */

#define I8250_MSR_DCTS          0x01    /* cts 改变 */
#define I8250_MSR_DDSR          0x02    /* dsr 改变*/
#define I8250_MSR_DRI           0x04    /* ring 改变 */
#define I8250_MSR_DDCD          0x08    /* data carrier 改变*/
#define I8250_MSR_CTS           0x10    /* complement of cts */
#define I8250_MSR_DSR           0x20    /* complement of dsr */
#define I8250_MSR_RI            0x40    /* complement of ring signal */
#define I8250_MSR_DCD           0x80    /* complement of dcd */

/* i8250IoCtrl 用到的一些常量的宏定义 */
#define I8250_SET_BAUDRATE      1       /* 设置波特率 */
#define I8250_SET_LINECTRL      2       /* 设置线控制字 */
#define I8250_GET_NREAD         3       /* 获得接收缓冲区已接收字符数    */
#define I8250_GET_NWRITE        4       /* 获得发送缓冲区未发送字符数 */
#define I8250_RECVBUF_FLUSH     5       /* 清接收缓冲区 */
#define I8250_SENDBUF_FLUSH     6       /* 清发送缓冲区 */

#define I8250_CSIZE             0xc     /* bits 3 and 4 数据位长度 */
#define I8250_CS5               0x0     /* 5 位数据位 */
#define I8250_CS6               0x4     /* 6 位数据位 */
#define I8250_CS7               0x8     /* 7 位数据位 */
#define I8250_CS8               0xc     /* 8 位数据位 */

#define I8250_STOPB             0x20    /* 2 位停止位，其他表示 1 位 */
#define I8250_PARENB            0x40    /* 奇偶校验使能，否则禁止 */
#define I8250_PARODD            0x80    /* 奇校验，否则为偶校验 */

/* 向应用层提供的接口函数声明 */
#if defined(__STDC__) || defined(__cplusplus)
STATUS i8250Init(int ioBase,int intLevel);
int i8250Send( char *  buffer, int  nbytes );
int i8250Recv (char *  buffer, int  nbytes );
int i8250IoCtrl ( int function,  int arg );
#else
STATUS i8250Init();
int i8250Send();
int i8250Recv();
int i8250IoCtrl ();

#endif  /* __STDC__ */
```

```
#ifdef __cplusplus
}
#endif

#endif  /* __INCi8250h */
```

3.2.3.2 程序 i8250.c

首先是包含必要的头文件，其中：

（1）VxWorks.h 是所用 VxWorks 程序应该包含的。

（2）error.h 定义了一些错误信息用到的宏。

（3）intLib.h 定义了中断处理相关的宏和函数声明。

（4）iv.h 定义了中断向量。

```
#include <vxWorks.h>
#include <intLib.h>
#include <errno.h>
#include <stdio.h>
#include <logLib.h>
#include <iv.h>
#include <ioLib.h>
#include <string.h>
#include "i8250.h"

#define COM0_IO_BASE 0x3f8
#define COM1_IO_BASE 0x2f8
#define COM0_INTLEVEL   4
#define COM0_INTLEVEL   3
```

串口芯片晶振频率为 1843200，采用 16 倍分频，波特率计算公式为：

波特率=晶振频率/（16×除数锁存寄存器值）

换算得到

除数锁存寄存器值= 1843200/（16×波特率）

```
#define FREQ 1843200
#define BAUDRATE(x) (1843200/16/x)
#define BAUD_LSB(x) (BAUDRATE(x)&0xff)
#define BAUD_MSB(x) ((BAUDRATE(x)&0xff00)>>8)
#define DEFAULT_BAUD    9600
#define DEFAULT_CTRL    (I8250_LCR_CS8 | I8250_LCR_1_STB)
```

为方便程序移植，定义寄存器访问宏。

```
#define WRITE_I8250_REG(reg,value)  sysOutByte(reg,value)
#define READ_I8250_REG(reg)  sysInByte(reg)
```

定义接送发送环形缓冲区大小。

```
#define RING_SIZE 512
```

局部函数声明。

```
LOCAL void i8250Int(I8250_CTRL *pI8250Ctrl);
LOCAL void i8250HwInit(I8250_CTRL *pI8250Ctrl,int baderate, int ctrl);
LOCAL STATUS  i8250BaudSet(I8250_CTRL *pI8250Ctrl, UINT baud);
```

全局变量声明，串口控制结构定义，采用这种方式可以使驱动支持多个串口设备。本文没有实现。实现支持多个设备功能仅需对每个函数增加一个串口设备序号，程序中根据不同设备序号，使用不同的串口控制结构指针即可。

```
LOCAL I8250_CTRL *pI8250Ctrl;
```

第一个函数是驱动初始化 i8250Init，要求传入芯片的基地址，中断级。主要流程：为 pI8250Ctrl 分配内存空间，并对成员变量初始化；调用 rngCreate 分配接收和发送环形缓冲区；调用连接 intConnect 中断处理程序，其中中断向量通过宏 INUM_TO_IVEC 转换得到，调用 i8250HwInit 初始化硬件。

```
STATUS i8250Init(int ioBase,int intLevel)
{
    pI8250Ctrl=( I8250_CTRL *)malloc(sizeof(I8250_CTRL));
    if(pI8250Ctrl==NULL)
        return ERROR;
    pI8250Ctrl-> baseAddr = ioBase;
    pI8250Ctrl-> intLevel= intLevel;
    /* 调用 rngCreate 分配接收和发送环形缓冲区 */
    if((pI8250Ctrl->recvRng=rngCreate(RING_SIZE))==NULL){
        free(pI8250Ctrl);
        return ERROR;
    }
    if((pI8250Ctrl->sendRng=rngCreate(RING_SIZE))==NULL){
        free(pI8250Ctrl->recvRng);
        free(pI8250Ctrl);
        return ERROR;
    }
    /* 连接中断处理程序 */
    intConnect(INUM_TO_IVEC(0x20+pI8250Ctrl->intLevel),
        (VOIDFUNCPTR )i8250Int, (int)pI8250Ctrl);
    /* 初始化 i8250 芯片，默认波特率 9600，8 位数据位，1 位数据位，无校验 */
    pI8250Ctrl->sendFlag = 0;          /* 未处于发送数据状态标志 */
    i8250HwInit(pI8250Ctrl, DEFAULT_BAUD, DEFAULT_CTRL);
    sysIntEnablePIC (devParas[i].intLevel);
    return OK;
}
```

函数 i8250BaudSet 用来设置波特率。先向线控制寄存器除数锁存访问使能位置 1，允许访问除数锁存寄存器（即偏移为 0 和 1 的两个寄存器），计算对应波特率的除数值，低字节写入偏移为 0 的寄存器，高字节写入偏移为 1 的寄存器。注意：当设置波特率时，需要在禁止系统中断（简称封中）条件下完成，封中操作由 intLock 函数完成，设置完成后调用 intUnlock 恢复系统中断设置。

```
/*******************************************************************************
*
* i8250BaudSet - 设置波特率
*
* 该函数设置串口的波特率，在芯片访问时，将禁止系统中断
*
* 返回：OK 为设置成功，否则返回 ERROR
*/
LOCAL STATUS  i8250BaudSet
    (
    I8250_CTRL *pI8250Ctrl,      /* 串口设备控制结构指针 */
    UINT       baud              /* 想要设置的波特率 */
    )
    {
    int          oldlevel;
    STATUS       status;
    FAST   int   ix;
    UINT8        lcr;

    /* 禁止系统中断 */
    oldlevel = intLock ();

    status = ERROR;

    if((baud<=1200)||(baud>=115200)){
        intUnlock(oldlevel);
        return status;
    }
    WRITE_I8250_REG(pI8250Ctrl->baseAddr + UART_LCR, I8250_LCR_DLAB);
    sysDelay();
    WRITE_I8250_REG(pI8250Ctrl->baseAddr + UART_BRDL, BAUD_LSB(baud));
    sysDelay();
    WRITE_I8250_REG(pI8250Ctrl->baseAddr + UART_BRDH, BAUD_MSB(baud));
    sysDelay();
    WRITE_I8250_REG(pI8250Ctrl->baseAddr + UART_LCR, pI8250Ctrl->lcr);
    sysDelay();
    status = OK;
    intUnlock(oldlevel);
    return (status);
}
```

函数 i8250HwInit 完成 i8250 芯片初始化。i8250 过程为首先设置波特率，然后设置线控制字，最后清数据口和使能中断。初始化操作也是在封中状态下完成。

```
/*******************************************************************************
* i8250HwInit i8250 硬件初始化
*
* 初始化 i8250 硬件，设置波特率，数据位、停止位、校验方式等
*
* 返回值：无。
*/
LOCAL void i8250HwInit(I8250_CTRL *pI8250Ctrl,int baudrate, int ctrl)
```

```
{
    int oldLevel;
    pI8250Ctrl->baudRate= baudrate;
    pI8250Ctrl->lcr = ctrl;
    /* 设置波特率 */

    i8250BaudSet(pI8250Ctrl, baudrate);
    oldLevel = intLock ();

    WRITE_I8250_REG (pI8250Ctrl->baseAddr + UART_LCR, pI8250Ctrl->lcr);
    sysDelay();
    WRITE_I8250_REG (pI8250Ctrl->baseAddr + UART_MDC,
        (I8250_MCR_RTS | I8250_MCR_DTR | I8250_MCR_OUT2));

    /* 清数据口 */
    sysDelay();

    READ_I8250_REG (pI8250Ctrl->baseAddr + UART_RDR);

    /* 允许接收发送中断 */
    WRITE_I8250_REG (pI8250Ctrl->baseAddr + UART_IER,
    I8250_IER_RXRDY| I8250_IER_TBE);
    intUnlock (oldLevel);
}
```

函数 i8250Int 包含驱动程序的中断服务代码，当 i8250 芯片产生中断时，X86 的中断处理机制将跳转到对应的中断向量地址，该中断向量地址在完成保存现场后，执行该函数，执行完毕恢复被中断的现场。整个过程是在封中条件完成，所以必须要使中断过程尽可能地短。中断处理过程如下：读中断使能寄存器，读中断识别寄存器，判断是哪种中断：

（1）接收中断。读接收数据寄存器，调用 rngBufPut，写到接收环形缓冲区。

（2）发送中断。判断发送缓冲区是否还有未发送数据，如有，则调用 rngBufGet 得到一个字符，然后写到数据口。发送缓冲区空，则清正在发送标志。

（3）其他中断处理，也都需要做清除相应的中断操作。

由于 i8250 可以同时产生多个中断，即发送和接收中断可能同时产生，为了防止频繁进出中断和长时间陷入该中断（造成其他中断无法得到服务），中断处理中需要连续查询多次，直至没有中断或达到一定次数。

```
#define I8250_IIR_READ_MAX    40

LOCAL void i8250Int(I8250_CTRL *pI8250Ctrl)
{
    char outChar;        /* store a byte for transmission */
    char interruptID;    /* store contents of interrupt ID register */
    char lineStatus;     /* store contents of line status register */
    char ier;            /* store contents of interrupt enable register */
    int  ix = 0;         /* allows us to return just in case IP never clears */

    ier = READ_I8250_REG(pI8250Ctrl->baseAddr + UART_IER);
```

```
        /* service UART interrupts until IP bit set or read counter expires */

        while(1){
            interruptID  =(READ_I8250_REG(pI8250Ctrl->baseAddr  +  UART_IID)  &
I8250_IIR_MASK);

            if ((interruptID == I8250_IIR_IP)  . (--ix > I8250_IIR_READ_MAX))
                {
                break;  /* interrupt cleared or loop counter expired */
                }

        /* filter possible anomalous interrupt ID from PC87307VUL (SPR 26117) */

            interruptID &= 0x06;  /* clear odd-bit to find interrupt ID */

            if (interruptID == I8250_IIR_SEOB)
                {
                lineStatus = READ_I8250_REG(pI8250Ctrl->baseAddr + UART_LST);
                }
            else if (interruptID == I8250_IIR_RBRF)
                {
                if (pI8250Ctrl->recvRng != NULL){
                    outChar=READ_I8250_REG(pI8250Ctrl->baseAddr + UART_RDR );
                    rngBufPut(pI8250Ctrl->recvRng,&outChar,1);
                }
                else
                    READ_I8250_REG(pI8250Ctrl->baseAddr + UART_RDR );
                }
            else if (interruptID == I8250_IIR_THRE)
                {
                if (pI8250Ctrl->sendRng != NULL)
                    if(rngBufGet(pI8250Ctrl->sendRng, &outChar,1) == 1)
                    {
                      WRITE_I8250_REG(pI8250Ctrl->baseAddr + UART_THR, outChar);
                    }else
                       pI8250Ctrl->sendFlag=0;

                 /* There are no bytes available for transmission.  Reading
                  * the IIR at the top of this loop will clear the interrupt.
                  */

                }
            else if (interruptID == I8250_IIR_MSTAT) /* modem status register */
                {
                char msr = READ_I8250_REG(pI8250Ctrl->baseAddr + UART_MSR);

                /* (|=) ... DON'T CLOBBER BITS ALREADY SET IN THE IER */

                ier |= (I8250_IER_RXRDY | I8250_IER_MSI);
```

```
            /* if CTS is asserted by modem, enable tx interrupt */

            if ((msr & I8250_MSR_CTS) && (msr & I8250_MSR_DCTS))
                {
                WRITE_I8250_REG(pI8250Ctrl->baseAddr + UART_IER, (I8250_IER_TBE
| ier));
                }
            else  /* turn off TBE interrupt until CTS from modem */
                {
                WRITE_I8250_REG(pI8250Ctrl->baseAddr   +   UART_IER,   (ier   &
(~I8250_IER_TBE)));
                }
            }
        }  /* while(1) */
    }
```

下面是三个应用编程频繁使用的函数：数据发送、数据接收和状态控制函数。

发送数据函数 i8250Send 在判断 pI8250Ctrl->sendFlag 为 1 时，也即发送环形缓冲区有数据，中断处理程序正在逐个发送数据，则直接将待发送数据写入发送环形缓冲区。当判断 pI8250Ctrl->sendFlag 为 0 时，表明发送环形缓冲区没有数据，需要主动向 i8250 数据发送寄存器写入第一个待发送字符，以启动发送中断，再写数据口之前，需要将除第一个字符外的发送报文写到发送环形缓冲区中。

```
/*******************************************************************
* i8250Send  串口发送数据函数
*
* 向串口发送一定字节长度的数据，由于串口发送缓冲区实际可用大小限制，实际能* 发送的字节数可
* 能小于等于待发送的字节数。对 sendRng 环形缓冲区，i8250Send 是写操作，仅改变写指针，中断
* 处理程序是读操作，仅改变读指针，因此，不存在互斥问题。
*
* 返回：实际发送的长度。
*/

int i8250Send(
    char *  buffer,          /* 待发送的数据 */
    int     nbytes           /*待发送的数据长度 */
 )
{
    int len;
    int oldLevel;
    oldLevel = intLock ();
    if(nbytes<1)
        return 0;
    if(pI8250Ctrl->sendFlag==1){
        len= rngBufPut (pI8250Ctrl->sendRng,buffer,nbytes);
        intUnlock(oldLevel);
    }
    else{
        len = rngBufPut (pI8250Ctrl->sendRng,&buffer[1],nbytes-1);
```

```
        intUnlock(oldLevel);
        pI8250Ctrl->sendFlag=1;
        WRITE_I8250_REG(pI8250Ctrl->baseAddr + UART_THR, buffer[0]);
        len=len-1;
    }
    return len;
}
```

数据的接收函数比较简单，直接将接收环形缓冲区的数据搬到用户缓冲区中即可。

```
/*******************************************************************
* i8250Recv  串口接收数据函数
*
* 由串口接收一定字节长度的数据，由于串口接收缓冲区实际已经接收字符的数目限* 制，实际接收的
* 字节数可能小于或等于希望接收的字节数。对 recvRng 环形缓冲区， i8250Recv 是读操作，仅改
* 变读指针，中断处理程序是写操作，仅改变写指针，因此，不存在互斥问题。
*
* 返回：实际接收数据的长度。
*/
int i8250Recv  (
    char *  buffer,      /* 存放接收数据的缓冲区 */
    int     nbytes       /* 希望接收的字节数 */
 )
{
    if(buffer==NULL)
        return 0;
    return rngBufGet (pI8250Ctrl->recvRng,buffer,nbytes);
}
```

i8250IoCtrl 函数是一个比较复杂的程序，允许应用根据需要设置串口工作方式，获得接收或发送缓冲区的数据长度，或清空接收或发送缓冲区。波特率的设置比较简单，线控制字的设置需要较多的判断。

```
/*******************************************************************
* i8250IoCtrl 串口设置或信息读取函数
*
* i8250IoCtrl 用来设置串口的波特率、方式字(数据位、停止位、校验方式等)
* 也可用来获取当前串口接收到的字符数目
* 获得当前未发送字符数目
* 清空接收缓冲区
* 清空发送缓冲区
*
* 返回：操作成功，返回 OK，否则，返回失败。
*/
int i8250IoCtrl (
    int function,        /*  function code */
    int arg              /*  arbitrary argument */
)
{
    int status=ERROR;
    int oldlevel;
```

```
UCHAR lcr_value;
switch (function){
        case  I8250_SET_BAUDRATE:
              i8250BaudSet(pI8250Ctrl, arg);
              status = OK;
              break;
        case  I8250_SET_LINECTRL:
              oldlevel = intLock();
              /*data length*/
              lcr_value= pI8250Ctrl->lcr;

        switch (arg&I8250_CSIZE)
        {
            case I8250_CS6:
                lcr_value|= I8250_LCR_CS6;
            break;
            case I8250_CS7:
                lcr_value|= I8250_LCR_CS7;
            break;
            case I8250_CS8:
                lcr_value|= I8250_LCR_CS8;
            break;
        }

        if (arg & I8250_STOPB)
            lcr_value |= I8250_LCR_2_STB;          /* select 2 stop bits */
        else
            lcr_value |= I8250_LCR_1_STB;          /* select one stop bit */

        switch (arg & (I8250_PARENB | I8250_PARODD))
        {
            case I8250_PARENB| I8250_PARODD:       /*ODD_PARITY*/
                lcr_value &= 0xef;
                lcr_value |= I8250_LCR_PEN;
            break;
            case I8250_PARENB:                     /*EVEN_PARITY*/
                lcr_value |= I8250_LCR_PEN | I8250_LCR_EPS;
            break;
            case 0:
            default:
                lcr_value &= 0xf7;                 /*NO_PARITY*/
            break;
        }
        pI8250Ctrl->lcr= lcr_value;
        WRITE_I8250_REG (pI8250Ctrl->baseAddr + UART_LCR,
            pI8250Ctrl->lcr);
        intUnlock(oldlevel);
        status=OK;
        break;
    case I8250_GET_NREAD:
```

```
            *((int *) arg) = rngNBytes (pI8250Ctrl->recvRng);
            status=OK;
            break;
        case I8250_GET_NWRITE:
            *((int *) arg) = rngNBytes (pI8250Ctrl->sendRng);
            status=OK;
            break;
        case I8250_RECVBUF_FLUSH:
            rngFlush (pI8250Ctrl->recvRng);
            status=OK;
            break;
        case I8250_SENDBUF_FLUSH:
            rngFlush (pI8250Ctrl->sendRng);
            status=OK;
            break;
        default:
            status=ERROR;
    }
    return status;
}
```

3.2.3.3 应用编程示例

上两节介绍了一个非标串口驱动的实现过程，本节通过一个实际的例子说明驱动的使用方法。

在本例中，首先调用 i8250Init 初始化串口设备，使用的是第一个串口，然后调用 i8250IoCtrl 设置串口的波特率和线控制字，并清空收发缓冲区，然后执行一个循环操作。在循环中，首先调用 i8250IoCtrl 获得串口当前接收字符数目，如果还没接收到数据，则稍作等待继续查询；如有可用数据，则调用 i8250Recv 读取串口接收到的数据，并通过该串口回送出去。

```
int i8250Test()
{
    int i,n;
    UCHAR buf[512];
    i8250Init(0x3f8,4);
    i8250IoCtrl(I8250_SET_BAUDRATE,38400);
    i8250IoCtrl(I8250_SET_LINECTRL, I8250_CS8| I8250_PARENB | I8250_PARODD);
    i8250IoCtrl(I8250_RECVBUF_FLUSH, 0);
    i8250IoCtrl(I8250_SENDBUF_FLUSH, 0);
    i=0;
    while(i++<5000){
        do{
            taskDelay(1);
            i8250IoCtrl(I8250_GET_NREAD, (int)&n);
        }while(n<=0);
        i8250Recv(buf,n);
        i8250Send("#Recv:",6);
        i8250Send(buf,n);
        if((i%100)==0)printf("\n %d",i);
    }
}
```

3.3 字符设备 TTY 驱动设计

上面介绍的串口非标准驱动，直接向应用系统提供一系列特定的接口函数，供应用程序调用。对于特定系统而言似乎没什么问题，但仔细考虑，这种驱动设计方式存在一些不利因素：

- 当系统有多个这种非标设备时，将有大量非标准编程接口存在，给应用编程带来不便。
- 采用非标准接口开发的应用程序，当硬件发生变动需要移植时会遇到很大困难。
- 多个非标设备的存在，不利于系统对设备的维护管理，不符合现代操作系统设备管理的趋势。

正因为存在上面提到的多个不利因素，现代操作系统通常提供一个 I/O 子系统，用来管理所用的输入/输出设备（网络除外），包括字符设备（串、并口）、块设备（盘设备）等，设备驱动程序按照一定规则编写，挂接在 I/O 子系统之下，由 I/O 子系统统一管理，向上层提供统一的文件操作接口。VxWorks 也采用这种思想，提供 I/O 子系统管理所用字符设备和块设备。

3.3.1 VxWorks I/O 体系

这一部分的阅读请结合《VxWorks programer's guide》一书第 3.9 节进行。

VxWorks I/O 子系统实际上是由一些数据结构和对应的操作函数来实现的。对于 VxWorks 系统，一个 I/O 设备驱动在完成初始化工作后，将向系统注册一个“设备”，这个“设备”是一个软件设备，在 shell 下调用 devs 或 iosDevShow 可以看到 I/O 子系统中所有的设备，如图 3-5 所示（网络设备信息可调用 ifshow 和 muxShow 查看）。同 UNIX 类似，这些设备可以看成文件，应用程序可以以文件方式操作这些设备。

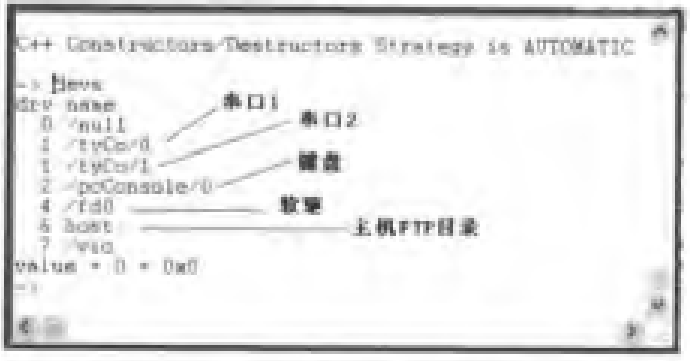


图 3-5 devs 命令显示的系统设备

其中“/tyCo/0”是第一个串口驱动对应的设备名，当应用程序使用该串口时，首先调用 open 打开该设备，获得一个整型文件句柄，然后可以以该文件句柄为参数调用 read 或 write 对串口进行读写操作。具体情况如下：

```
int fd;
char buf[64];
fd=open("/tyCo/0",2,0);          /* 第二个参数 2 表示打开后允许读写 */
write(fd, "abcdefg",7);
read(fd,buff,64);
```

在 VxWorks 系统中，文件句柄存放在一个 FD_ENTRY （在 ioslibp.h 中定义）类型的结构

指针数组 fdTable 中，数组的大小由 I/O 系统的参数 NUM_FILES 决定，默认为 50，数组的初始化在系统初始化时调用 iosInit 函数中完成，“null”就是在这时创建的。可以调用文件操作设备工作。

FD_ENTRY 类型定义如下：

```
typedef struct                  /* FD_ENTRY - entries in file table */
    {
    DEV_HDR *    pDevHdr;       /* device header for this file */
    int          value;         /* driver's id for this file */
    char *       name;          /* actual file name */
    int          taskId;        /* task to receive SIGIO when enabled */
    BOOL         inuse;         /* active entry */
    BOOL         obsolete;      /* underlying driver has been deleted */
    void *       auxValue;      /* driver specific ptr, e.g. socket type */
    void *       reserved;      /* reserved for driver use */
    } FD_ENTRY;
```

FD_ENTRY 结构的第一个成员 pDevHdr 指向对应的设备，类型为 DEV_HDR，定义在 ioslib.h 中。

```
typedef struct      /* DEV_HDR - device header for all device structures */
    {
    DL_NODE      node;          /* device linked list node */
    short        drvNum;        /* driver number for this device */
    char *       name;          /* device name */
    } DEV_HDR;
```

FD_ENTRY 结构的第二个成员 value，用来索引设备驱动函数表描述指针，等于设备驱动函数表数组 drvTable 中的一项，inuse 表示该链表项是否已使用。已经注册到系统中的所有设备的控制结构中都有一个类型为 DEV_HDR 的成员变量，DEV_HDR 类型变量的第一个成员变量 node 类型是 DL_NODE，它是一个双向链节点，I/O 系统通过 node 将所有注册到系统中的设备组成一个双向链，I/O 系统使用一个类型为 DL_LIST 的全局变量 iosDvList 作为这个双向链的链头和链尾，便于在搜索设备双向链时使用。

每个 I/O 设备驱动函数都提供一套函数（create、open、close、read、write、ioctl、remove），该函数表类型为 DRV_ENTRY（也在 ioslibp.h 中定义），定义如下：

```
typedef struct                  /* DRV_ENTRY - entries in driver jump table */
    {
    FUNCPTR  de_create;
    FUNCPTR  de_delete;
    FUNCPTR  de_open;
    FUNCPTR  de_close;
    FUNCPTR  de_read;
    FUNCPTR  de_write;
    FUNCPTR  de_ioctl;
    BOOL     de_inuse;
    } DRV_ENTRY;
```

所有设备的函数表在系统中组成一个函数表数组 drvTable，也由 I/O 系统维护，数组的大

小由 I/O 系统的参数 NUM_DRIVERS 决定，默认为 20。设备驱动调用 iosDrvInstall 将该设备驱动的函数表添加到 drvTable 中的一项，并将对应的 drvTable 数组下标返回。该返回值同设备名称一起将作为对应设备驱动调用 iosDevAdd 向 I/O 系统增加设备时的参数，分别赋值给该设备控制结构类型为 DEV_HDR 成员变量的 drvNum 和 name 成员。

这样，当应用程序调用 open 或 create 函数时（例如 fd=open("/tyCo/0",2,0)），根据设备名称，I/O 系统首先遍历 iosDvList 设备驱动链，找到对应名称的设备驱动控制结构，获得对应的 drvNum 值（也即驱动函数表数组的下标），再在 fdTable 中找到一个空白项，将该设备驱动控制结构指针赋值给 pDevHdr，drvNum 赋值给 value，设备名称赋值给 name，inuse 赋值为 TRUE，将该项在 fdTable 中的序号作为 open 函数的返回值，返回给应用程序，作为该设备的文件句柄 fd 值。

当应用程序调用 read（或 write、ioctl、close、remove）等函数操作设备时，需要将该设备的文件句柄 fd 作为第一个参数，I/O 系统将 fd 值作为 fdTable 数组的下标，找到对应的设备句柄结构，以 value 作为驱动函数表数组的下标，找到该设备驱动对应的函数表，执行对应的函数调用，完成对设备的操作。

上述过程的示意图如图 3-6 所示。

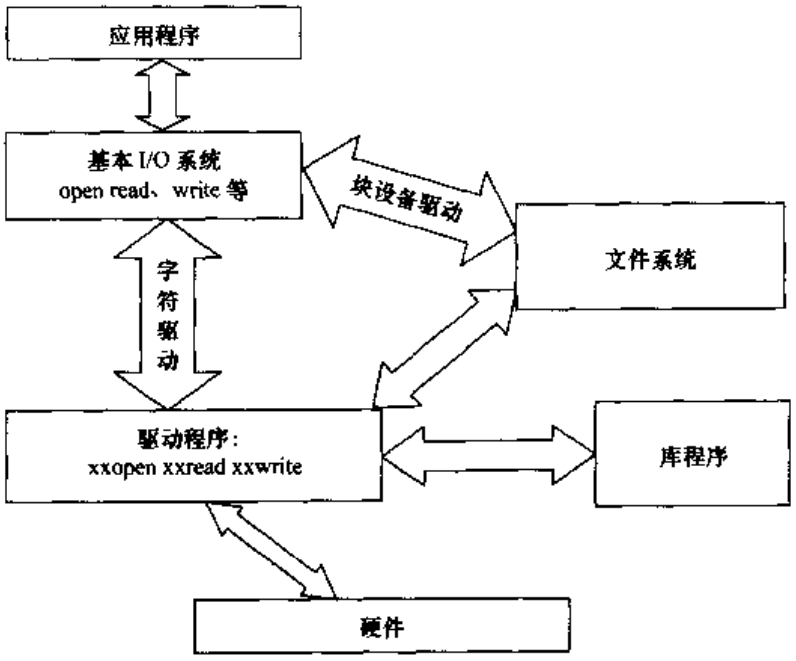


图 3-6　VxWorks I/O 子系统体系结构图

3.3.2　字符设备 TTY 驱动框架

对于 I/O 系统来说，一个 VxWorks 字符设备驱动可以看成一个对象，有自己的成员数据以及操作设备的功能函数。

按照 VxWorks 字符设备驱动标准要求，字符设备驱动的成员数据需要设计成如下形式的结构，通常称为设备描述子：

```
typedef struct          /* TY_CO_DEV */
{
    TY_DEV tyDev;
```

```
    BOOL created;        /* true if this device has really been created */
                         /* 以下成员根据驱动的具体需要而定 */
    int ioaddr;
}XX_DEV;
```

其中：

（1）第一个成员 tyDev 是必须的，这样可以直接利用 I/O 系统提供的 ttyLib 库，为驱动设计提供方便。TY_DEV 类型定义在 tyLib.h 中。

（2）第二个成员 created 标记该设备是否以创建，避免重复创建。

（3）其他的成员根据驱动设备的具体情况而定，通常包括设备硬件信息，如 IO 基地址、中断级等等。

同样一个字符设备驱动可以支持多个采用相同芯片设计的设备，如一个 i8250 串口驱动可以支持两个以上的采用 i8250 芯片设计的串口设备，因此，驱动程序通常声明一个 XX_TY_CO_DEV 类型的结构数组，如：

```
#define N_XXX_DEV   4    /* 该驱动可以最多支持四个相同的设备 */
XX_ DEV xxTyDev[N_XXX_DEV];
```

字符设备驱动需要提供的操作设备的功能函数包括 xx_creat()、xx_remove()、xx_open()、xx_close()、xx_read()、xx_write()、xx_ioctl()等 7 个函数，分别对应于驱动的创建、移出（I/O 系统）、打开、关闭、读、写和设置控制等功能。每个函数的原型如下：

```
LOCAL STATUS xx_create(
    XX_DEV* pXXDev,
    char* additionalInfo,
    int mode)
{

}

LOCAL STATUS xx_ remove (
    XX_DEV* pXXDev)
{

}

LOCAL STATUS xx_open(
    XX_DEV* pXXDev,
    char* additionalInfo,
    int mode)
{

}

LOCAL STATUS xx_ close (
    XX_DEV* pXXDev)
{

}
```

```
LOCAL STATUS xx_read(
    XX_DEV* pXXDev,
    char* buffer,
    int nBytes)
{
/* read nBytes from the device and put them into the buffer*/
}

LOCAL STATUS xx_write(
    XX_DEV* pXXDev,
    char* buffer,
    int nBytes)
{
/* write to the device from buffer if the device has room*/
}

LOCAL STATUS xx_ioctl(
    XX_DEV* pXXDev,
    int request,
    int arg)
{

}
```

对于大部分字符设备而言，xx_create 函数功能等同于 xx_open，通常可用 xx_open 函数替代，对字符设备一般不调用 xx_create 操作；xx_open 函数中通常是启动设备的接收中断；xx_close 函数也不需要做其他工作，该函数可以为空。xx_remove 函数调用 iosDevDelete 删除该设备；如没有特殊要求，可直接用 tyRead、tyWrite 替代 xx_read、xx_write 函数；一些与设备相关的设置或获取信息操作可放在 xx_ioctl 中实现，如波特率设置、线控制设置等。

驱动程序调用 **iosDrvInstall** 将这些函数注册到 I/O 系统的驱动函数表 drvTable（参见上节内容）中，代码如下。

```
int xxDrvNum;
xxDrvNum = iosDrvInstall(    xx_create       /*create*/,
                             xx_ remove      /*remove*/,
                             xx_ open        /*open()*/,
                             xx_ close       /*close()*/,
                             xx_ read        /*read()*/,
                             xx_ write       /*write()*/,
                             xx_ ioctl       /*ioctl()*/
                         );
```

为在 I/O 系统中注册一个设备（一个驱动可支持多个同类设备），驱动程序可以调用 iosDevAdd 来完成。例如，对第 i 个设备，代码如下。

```
XX_DEV* pXXDevice= (XX_DEV*)&xxTyDev[i];
status = iosDevAdd( &pXXDevice->tyDev.devHdre,
                            /* pointer to XX_DEV device */
    name,                   /* input param */
```

```
        xxDrvNum                    /* return value from    iosDrvInstall */
        );
```

其中 name 为对应的设备名称，命名方式通常为“/tyXX/i”，其中，i 表示该驱动支持的第 i 个设备，从 0 开始编号。

为了充分利用 I/O 系统提供的 TTY 设备读写缓冲区管理功能，在调用 iosDevAdd 创建之前，需要调用 tyDevInit 对 pXXDevice 中的 tyDev 成员进行初始化。例如，下述代码初始化 tty 设备描述子，接收缓冲区大小为 512，发送缓冲区大小为 256，xxStartup 为发送启动函数，后面将进一步介绍该函数的作用。

```
int rdBufSize=512, wrtBufSize=256;
tyDevInit (&pXXDevice->tyDev, rdBufSize, wrtBufSize, (FUNCPTR)tyCoStartup);
```

如上所述，设计一个字符设备分为以下几步：

（1）设计设备驱动的设备描述子。

（2）调用 iosDrvInstall 安装驱动函数表。

（3）连接中断，初始化硬件，使能中断。

（4）调用 tyDevInit 初始化设备描述子中的 tyDev 成员。

（5）调用 iosDevAdd 向 I/O 系统注册该设备。

（6）实现每个必须的子函数，如 create、open、close、read、write、remove、ioctl、xxStartup 和中断处理函数等。

xxStartup 函数用于启动串口发送中断，前面已经提到，串口的发送空中断必须在第一个字符发送完成后产生，因此必要的情况下需要在该函数体内发送第一个字符。

由于一个驱动可以支持多个设备，因此通常把设备创建函数独立出来，这样提供一个驱动初始化函数 xxDrvInstall 和一个设备创建函数 xxDevCreate。

其中，步骤 2 在 xxDevCreate 中完成，如果多个设备共享一个中断，则中断连接工作也要在该函数中完成，否则放在 xxDevCreate 中。步骤 3、4、5 在 xxDevCreate 中实现，形式如下：

```
STATUS xxDrvInstall()
{
if (xxDrvNum > 0)
return (OK);
    /*调用 iosDrvInstall 安装驱动函数表*/
    xxDrvNum = iosDrvInstall (  xx_create,      xx_ remove, xx_ open,
                                xx_ close , xx_ read,  xx_ write, xx_ ioctl);

    return (xxDrvNum == ERROR ? ERROR : OK);
}

STATUS xxDevCreate(char *name, int channel, int rdBufSize, int wrtBufSize)
{
    XX_DEV* pXXDevice= (XX_DEV*)&xxTyDev[channel];

    if (xxDrvNum <= 0)
    {
    errnoSet (S_ioLib_NO_DRIVER);
    return (ERROR);
```

```
    }

    /* if this device already exists, don't create it */

    if (pXXDevice ->created)
    return (ERROR);
    /* 初始化设备描述子中的 tyDev 成员 */
    if (tyDevInit (&pXXDevice ->tyDev, rdBufSize, wrtBufSize, (FUNCPTR)
xxStartup) != OK)
    {
    return (ERROR);
    }

    /* 连接中断处理程序 */
       intConnect(INUM_TO_IVEC(0x20+ pXXDevice ->intLevel), (VOIDFUNCPTR)
i8250DevInt, (int) pXXDevice);
    /* 初始化该通道硬件 */
       hwInit(pXXDevice);
    /* 使能中断 */
       sysIntEnablepic(pXXDevice ->intLevel);
    /* 标记设备已创建，增加该设备到 I/O 系统 */
    pXXDevice ->created = TRUE;
    return (iosDevAdd (&pXXDevice ->tyDev.devHdr, name, xxDrvNum));
}
```

3.3.3 串口设备驱动实现

本节以 PC 串口 I8250 芯片驱动为例介绍标准字符设备驱动程序的设计实现过程。

3.3.3.1 i8250tty.h 设计

i8250tty.h 借用前面介绍 i8250.h，增加串口设备的设备描述子结构定义：

```
#include "tyLib.h"
#include "i8250.h "

   typedef struct          /* I8250_DEV */
   {
      TY_DEV   tyDev;
      BOOL created;        /* true if this device has really been created */
      int      ioaddr;     /* 串口 IO 基地址 */
      int      intLevel;   /* 中断级 */
      UCHARlcr;            /* 线控制字 */
   }I8250_DEV;
```

3.3.3.2 i8250tty.c 设计

具体情况如下：

```
#include "vxWorks.h"
#include "iv.h"
#include "ioLib.h"
#include "iosLib.h"
#include "tyLib.h"
```

```
#include "intLib.h"
#include "errnoLib.h"
#include "sysLib.h"
#include "i8250tty.h"

#define COM0_IO_BASE    0x3f8
#define COM1_IO_BASE    0x2f8
#define COM0_INTLEVEL   4
#define COM1_INTLEVEL   3

#define N_I8250_CHANNELS    2

/* 全局变量 */
I8250_DEV i8250Dev [N_I8250_CHANNELS] =
    {
    {{{{NULL}}}, FALSE,
     COM0_IO_BASE,
     COM0_INTLEVEL,
     0x03},
    {{{{NULL}}}, FALSE,
     COM1_IO_BASE,
     COM1_INTLEVEL,
     0x03},
    };
```

同 3.2.3.2 节介绍的一样，以下用于对应波特率除数除数锁存寄存器值计算。

```
#define FREQ 1843200
#define BAUDRATE(x) (1843200/16/x)
#define BAUD_LSB(x) (BAUDRATE(x)&0xff)
#define BAUD_MSB(x) ((BAUDRATE(x)&0xff00)>>8)
#define DEFAULT_BAUD    9600
#define DEFAULT_CTRL    (I8250_LCR_CS8 | I8250_LCR_1_STB)
```

定义寄存器访问宏。

```
#define WRITE_I8250_REG(reg,value)  sysOutByte(reg,value)
#define READ_I8250_REG(reg)  sysInByte(reg)
```

i8250DrvNum 表示 I/O 系统分配给本驱动的驱动号。

```
LOCAL int i8250DrvNum;
```

局部函数提前声明。

```
LOCAL void      i8250Startup    (I8250_DEV *pI8250Dev);
LOCAL int       i8250Open       (I8250_DEV * pI8250Dev, char *name, int mode);
LOCAL STATUS    i8250Ioctl      (I8250_DEV * pI8250Dev, int request, int arg);
LOCAL void      i8250HwInit     (I8250_DEV *pI8250Dev);
LOCAL void      i8250DevInt     (I8250_DEV *pI8250Dev);
```

i8250DrvInstall：串口字符设备驱动安装程序，注意本函数只能调用一次，应该在 i8250DevCreate 之前调用。返回值：OK 表示驱动安装成功，ERROR 表示安装失败。该函数首先检查驱动是否已经安装，以避免重复安装，然后调用 iosDrvInstall 安装驱动函数表。由于 PC

机主板上两个串口虽然机制相同但使用不同的中断号和 I/O 地址空间，因此可以分开控制，因而初始化放在设备创建时完成。

```
STATUS i8250DrvInstall (void)
{
    /* 检查驱动是否已安装 */

    if (i8250DrvNum  > 0)
    return (OK);

    /* 安装驱动函数表 */
    i8250DrvNum = iosDrvInstall (i8250Open, (FUNCPTR) NULL, i8250Open,
              (FUNCPTR) NULL, tyRead, tyWrite, i8250Ioctl);

    return (i8250DrvNum == ERROR ? ERROR : OK);
    }
```

函数 i8250DevCreate：创建一个串口设备，其中：

（1）参数 name 指明串口设备的名称。

（2）channel 表示是该驱动支持的第几个串口设备，应该小于驱动最大支持的支持设备数。

（3）rdBufSize、wrtBufSize 分别是接收缓冲区和发送缓冲区的大小。

例如，i8250DevCreate ("/i8250/0", 0, 512, 512); 表示创建一个第 0 个通道，设备名为"/i8250/0"的串口设备。注意设备名称通常要以‘/’开始，如果驱动支持多个设备，最后一个‘/’后用数字表示是第几个设备。

返回：①OK：设备创建成功；②ERROR：安装不成功，驱动未安装或通道无效或设备已存在。

该函数首先判断对应的通道号是否没有超出驱动支持的最大设备数，然后判断驱动函数表是否安装。如果上述条件成立，调用 tyDevInit 初始化设备描述子中的 tyDev 成员；连接中断处理程序，初始化硬件，使能对应的中断级，然后调用 iosDevAdd 向 I/O 系统注册该设备。

该函数应该在 i8250DrvInstall 之后调用，执行成功后，应用程序可以调用 open、read、write、ioctl、close 等函数操作该设备。

```
STATUS i8250DevCreate
    (
    char *name,
    FAST int channel,
    int rdBufSize,
    int wrtBufSize
    )
    {
    I8250_DEV *pI8250Dev = &i8250Dev[channel];

    if (channel >= N_I8250_CHANNELS)
        return (ERROR);

    if (i8250DrvNum  <= 0)
    {
```

```
        errnoSet (S_ioLib_NO_DRIVER);
        return (ERROR);
    }

    /* 判断设备是否已创建 */

    if (pI8250Dev->created)
        return (ERROR);

    if (tyDevInit (&pI8250Dev->tyDev, rdBufSize, wrtBufSize,
    (FUNCPTR)i8250Startup) != OK)
    {
        return (ERROR);
    }

    /* 连接中断处理程序 */
    intConnect(INUM_TO_IVEC(0x20-  pI8250Dev  ->intLevel),  (VOIDFUNCPTR)
i8250DevInt, (int) pI8250Dev);

    /* 初始化该通道硬件 */
    i8250HwInit(pI8250Dev);

    /* 使能中断 */
    sysIntEnablePIC(pI8250Dev ->intLevel);

    /* 标示设备已创建，并增加到 I/O 系统 */

    pI8250Dev->created = TRUE;
    return (iosDevAdd (&pI8250Dev->tyDev.devHdr, name, i8250DrvNum ));
    }
```

函数 tyCoHrdInit，初始化指定通道硬件：设置默认波特率、线控制字，允许接收中断。

```
LOCAL void i8250HwInit (I8250_DEV *pI8250Dev)

    {
    int oldLevel = intLock ();
    int ix;

    WRITE_I8250_REG (pI8250Dev->ioaddr + UART_LCR, I8250_LCR_DLAB);
    WRITE_I8250_REG (pI8250Dev->ioaddr + UART_BRDL, BAUD_LSB(9600));
    WRITE_I8250_REG (pI8250Dev->ioaddr + UART_BRDH, BAUD_MSB(9600));
    WRITE_I8250_REG (pI8250Dev->ioaddr + UART_LCR, pI8250Dev->lcr);
    WRITE_I8250_REG  (pI8250Dev->ioaddr  +  UART_MDC,  (I8250_MCR_OUT2  |
I8250_MCR_RTS | I8250_MCR_DTR));

    /* 清数据口 */

    READ_I8250_REG (pI8250Dev->ioaddr + UART_RDR);

    /* 使能接收 */
```

```
    WRITE_I8250_REG (pI8250Dev->ioaddr + UART_IER,  I8250_IER_RXRDY);

    intUnlock (oldLevel);

    }
```

i8250Open：打开串口设备函数。该函数将设备描述子指针地址返回给上层调用者。

```
LOCAL int i8250Open
    (
    I8250_DEV *pI8250Dev,
    char *name,
    int mode
    )
    {
    return ((int) pI8250Dev);
    }
```

i8250Ioctl：设备特定的控制功能函数。处理如波特率设置、线控制字等设备设置功能，其他设置功能传递给 tyIoctl 处理。

返回：OK 表示调用成功，ERROR 表示设置无效。

```
LOCAL STATUS i8250Ioctl
    (
    I8250_DEV *pI8250Dev,    /* device to control */
    int request,             /* request code */
    int arg                  /* some argument */
    )
    {
    int ix;
    int status;
    UCHAR lcr_value;
    int oldLevel;
    switch (request)
    {

    /* 设置波特率 */
    case FIOBAUDRATE:
        if((arg>=1200)&&(arg<=115200))
        {
            oldLevel = intLock ();
            WRITE_I8250_REG (pI8250Dev->ioaddr + UART_LCR, I8250_LCR_DLAB);
            WRITE_I8250_REG (pI8250Dev->ioaddr + UART_BRDL, BAUD_LSB(9600));
            WRITE_I8250_REG (pI8250Dev->ioaddr + UART_BRDH, BAUD_MSB(9600));
            WRITE_I8250_REG (pI8250Dev->ioaddr + UART_LCR, pI8250Dev->lcr);
            intUnlock (oldLevel);
            status=OK;
        }else
            status=ERROR;
        break;
```

```
/* 设置线控制字 */
case SIO_HW_OPTS_SET:

    oldLevel = intLock();

    lcr_value =READ_I8250_REG (pI8250Dev->ioaddr + UART_LCR);

    lcr_value &=0xe0;


    /*data length*/
    switch (arg&CSIZE)
    {
        case CS6:
            lcr_value|=I8250_LCR_CS6;
        break;
        case CS7:
            lcr_value|=I8250_LCR_CS7;
        break;
        case CS8:
            lcr_value|=I8250_LCR_CS8;
        break;
    }

    if (arg & STOPB)
        lcr_value |= I8250_LCR_2_STB;      /* select 2 stop bits */
    else
        lcr_value |= I8250_LCR_1_STB;      /* select one stop bit */


    switch (arg & (PARENB | PARODD))
    {
        case PARENB|PARODD:                /*ODD_PARITY*/
            lcr_value &= 0xef;
            lcr_value |= I8250_LCR_PEN;
        break;
        case PARENB:                       /*EVEN_PARITY*/
            lcr_value |= I8250_LCR_PEN | I8250_LCR_EPS;
        break;
        case 0:
        default:
            lcr_value &= 0xf7;             /*NO_PARITY*/
        break;
    }

    WRITE_I8250_REG (pI8250Dev->ioaddr + UART_LCR,lcr_value);
    intUnlock (oldLevel);
    pI8250Dev->lcr=lcr_value;
    break;
```

```
        /* 其他请求传递给 tyIoctl 处理 */
        default:
            status = tyIoctl (&pI8250Dev->tyDev, request, arg);
            break;
        }

        return (status);
        }
```

i8250DevInt：串口中断处理函数。处理接收发送及线状态中断。

```
    LOCAL void i8250DevInt
        (
        I8250_DEV *pI8250Dev
        )
        {
        char outChar;
        char interruptID;
        char lineStatus;
        int ix = 0;

        interruptID = READ_I8250_REG (pI8250Dev->ioaddr + UART_IID);
        do {

        interruptID &= I8250_IIR_SEOB;

            if (interruptID == I8250_IIR_SEOB)
                lineStatus = READ_I8250_REG (pI8250Dev->ioaddr + UART_LST);
            else if (interruptID == I8250_IIR_RBRF)
            {
                if (pI8250Dev->created)
                tyIRd (&pI8250Dev->tyDev, READ_I8250_REG (pI8250Dev->ioaddr +
UART_RDR));
            else
                READ_I8250_REG (pI8250Dev->ioaddr + UART_RDR);
            }
            else if (interruptID == I8250_IIR_THRE)
            {
                if ((pI8250Dev->created && tyITx (&pI8250Dev->tyDev, &outChar)) ==
OK)
                    WRITE_I8250_REG (pI8250Dev->ioaddr + UART_THR, outChar);
                else
                    WRITE_I8250_REG (pI8250Dev->ioaddr + UART_IER, I8250_IER_RXRDY);
            }

            interruptID = READ_I8250_REG (pI8250Dev->ioaddr + UART_IID);

        } while (((interruptID & I8250_IIR_MASK) == I8250_IIR_IP) &&
             (ix++ < 10));
        }
```

函数 i8250Startup：发送启动函数。该函数从发送缓冲区去取一个字节数据，写到 8250 的数据寄存器，然后允许发送空中断。

```
LOCAL void i8250Startup
    (
    I8250_DEV *pI8250Dev    /* tty device to start up */
    )
    {
    char    outChar;
    char    lsrReg;
    static int  charCount = 0;

    /* enable the transmitter and it should interrupt to write the next char */

    if (tyITx (&pI8250Dev->tyDev, &outChar) == OK){

        WRITE_I8250_REG (pI8250Dev->ioaddr + UART_THR, outChar);

        WRITE_I8250_REG  (pI8250Dev->ioaddr  +  UART_IER,  (I8250_IER_TBE |
I8250_IER_RXRDY));
        }

    }
```

3.3.3.3 字符设备编程

以上面介绍的 i8250 串口字符设备驱动为例，介绍应用如何使用该驱动进行编程操控该串口设备。

在应用访问设备之前，必须调用 i8250DrvInstall 安装设备驱动，然后调用 i8250DevCreate 创建一个串口设备，之后就可以调用 open 打开该设备，返回一个文件描述符，就可以像读写文件一样利用该串口进行数据通信。

下述代码首先调用 i8250DrvInstall 安装设备驱动，然后调用 i8250DevCreate 创建一个串口设备"/i8250/0"，用 open 打开该设备，返回一个文件描述子 fd，然后循环 50 次：首先从串口读数据到 buff 中，并返回接收的数据字节数目，然后先回送字符串“recv:”，然后回送接收到的数据。循环结束后，关闭文件，退出。

```
void ttyTest()
{
    int fd,i,len;
    char buff[512];
```

第一步：安装设备驱动。

```
i8250DrvInstall();
```

第二步：创建字符设备，串口 0，名称为"/i8250/0"。

```
i8250DevCreate("/i8250/0",0,512,512);
```

第三步：打开该串口设备。

```
fd=open("/i8250/0",2,0);
```

第四步：可以调用 ioctl 进行设备设置，或 read/write 进行读写。

```
while(i++<50){
    len=read(fd,buff,32);
    if(len>0){
        buff[len]=0;
        printf("\n recv %d %s",len,buff);
        write(fd,"recv: ",6);
        write(fd,buff,len);
    }
}
```

第五步：操作完成关闭设备文件。

```
    close(fd);
}
```

第 4 章　END 网络驱动设计

网络已成为操作系统不可或缺的组成部分，嵌入式系统也不例外，但是不像桌面操作系统，嵌入式操作系统自身支持的网卡设备不是太多，网卡设备厂商通常也不为嵌入式操作系统提供相应的驱动程序，因此开发者往往需要自己完成网卡驱动程序的设计。

写一个网络驱动需要掌握以下几方面的知识：

- 现代操作系通常采用 BSD TCP/IP 网络协议，因此要对 TCP/IP 网络体系有一个基本了解。
- 对操作系统的网络架构，特别是数据链路层与上层协议的接口、存储器管理等要非常清楚。
- 需要了解该操作系统驱动开发的一些基础知识，如 PCI 设备查找、访问、端口读写，中断连接、处理等，如何调试等。
- 清楚了解网卡硬件工作原理，寄存器定义，接收/发送过程和中断处理等。

本章从上述几方面，介绍 VxWorks 网络驱动程序编写相关的知识，并以 ns83820 网络驱动为例介绍驱动设计与实现过程。

4.1　END 驱 动 概 述

4.1.1　MUX 与 END

VxWorks 网络协议栈遵循工业标准 BSD4.4，可以分为以下几层：应用层、传输层、IP 层、MUX 层、数据链路层和物理层。如图 4-1 所示。

从图中可知，与其他 TCP/IP 协议相比，VxWorks 网络协议栈在数据链路层与 IP 层之间增加了一层，也就是 MUX 层。MUX 层是 VxWorks 为方便在网络接口硬件上实现多种协议而增加的一层，它用于管理底层的多个（种）硬件（以太网、PPP 等）设备驱动（END），向上层不同协议提供统一的接口，降低了上层协议与底层物理硬件的耦合，使得网络驱动和上层协议彼此保持独立，既方便在现有硬件基础上实现新的上层协议，也利于用新的硬件支持原有的上层协议。

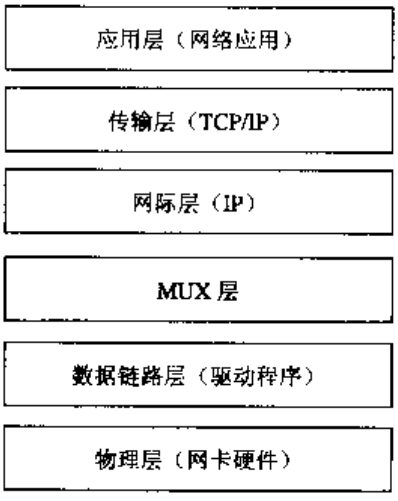


图 4-1　VxWorks 网络协议栈分层结构示意图

与其他几层不同，MUX 不提供任何的协议处理。MUX 在网络协议中的作用如图 4-2 所示。

VxWorks5.4 及以后的版本采用所谓 END（Enhanced Network Driver，增强型网络驱动）

网络驱动。END 位于数据链路层，实际上数据链路层是一个抽象的概念，也称网络接口层。网络接口驱动程序是该层的具体代码实现，是上层协议通过物理硬件与外部进行数据交互的通道。

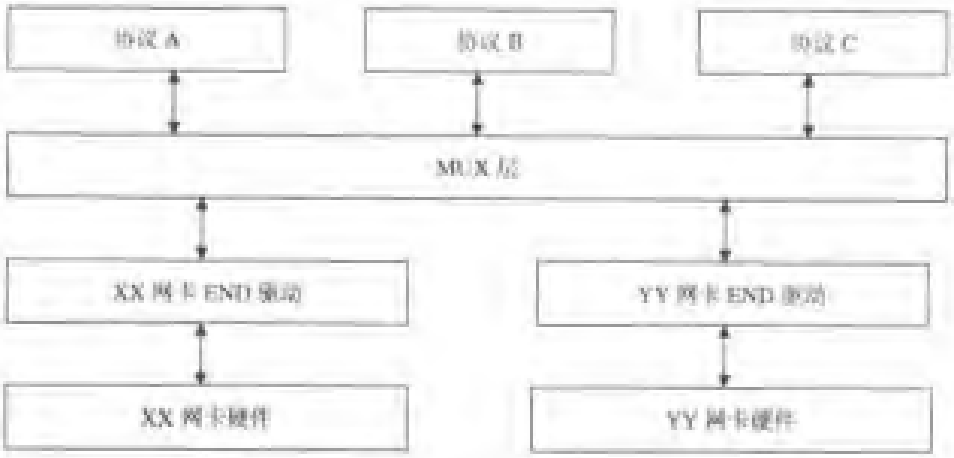


图 4-2　MUX 在网络协议中的作用

VxWorks 中，MUX 与 END 合称 SENS 驱动（Scalable Enabced Networkd Drives，可裁减增强型网络驱动）。

MUX 与 END 的交互是通过提供一套可供底层调用接口服务来实现的，实现一个 END 驱动必须遵循这套接口关系。二者的接口关系如图 4-3 所示。

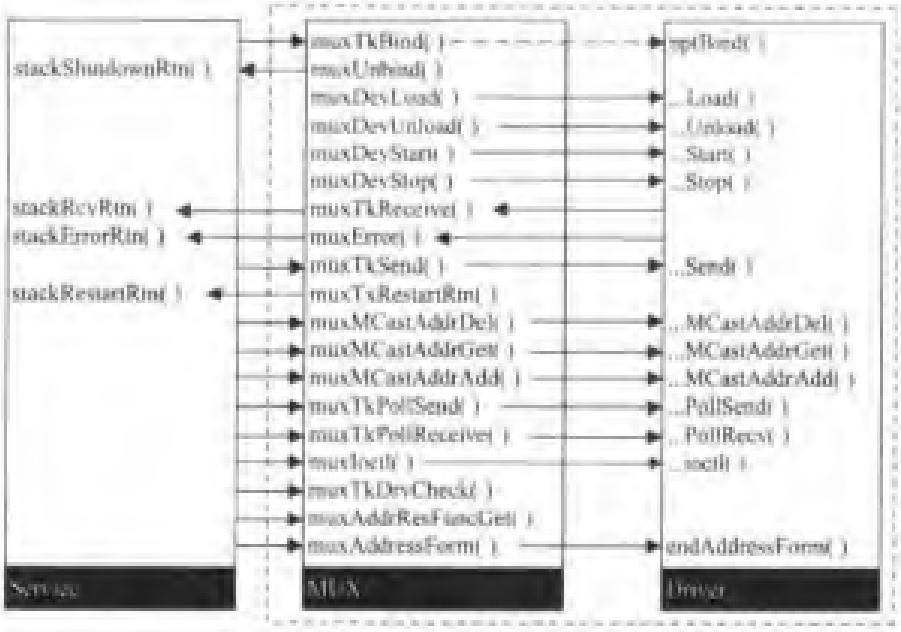


图 4-3　MUX 与 END 的接口关系

图 4-3 中右边框中列出的函数是驱动需要实现的函数，供 MUX 层调用在必要的时候调用，如当上层使用该网卡发送数据时，MUX 会调用该网卡 END 驱动提供的 Send 函数，将数据提交给网卡芯片硬件。

VxWorks 提供了一套函数库和相应的数据结构用于驱动与 MUX 进行数据交换，下面分别介绍。

4.1.2　缓冲池数据结构

网络设备驱动与上层协议进行数据交换需要相应的内存缓冲，并且管理这些缓冲也需要相应的函数。VxWorks 提供了 netBufLib 函数库用于创建和管理网络设备用到的内存缓冲池，网络设备驱动可以直接使用也可在此基础上设计自己特定的内存缓冲池。数据以簇的形式保存，数据结构 mBlks(内存块)和 clBlks（簇块）形成的数据链结构（如图 4-4 所示）则用于指定各个簇。

从图 4-4 中可以看出，mBlk 是用于管理 clBlk 的数据结构，一个 mBlk 管理一个 clBlk，也就是说一个 mBlk 仅指向一个 clBlk，但是一个 clBlk 可以被多个 mBlk 同时指向。mBlk 中也包含了 clBlk 所指向的簇中所包含的数据信息。多个 mBlk 可链在一起，由这些 mBlk 索引到的 cluster 数据共同形成一个网络报文。

clBlk 是用于管理 cluster 的数据结构，一个 clBlk 仅能指向一个簇，一个簇也仅能由一个 clBlk 所指向，也即 clBlk 和簇是一一对应的关系。

一个簇就是一个固定大小缓冲，然而并非所有的簇都有相同的大小。

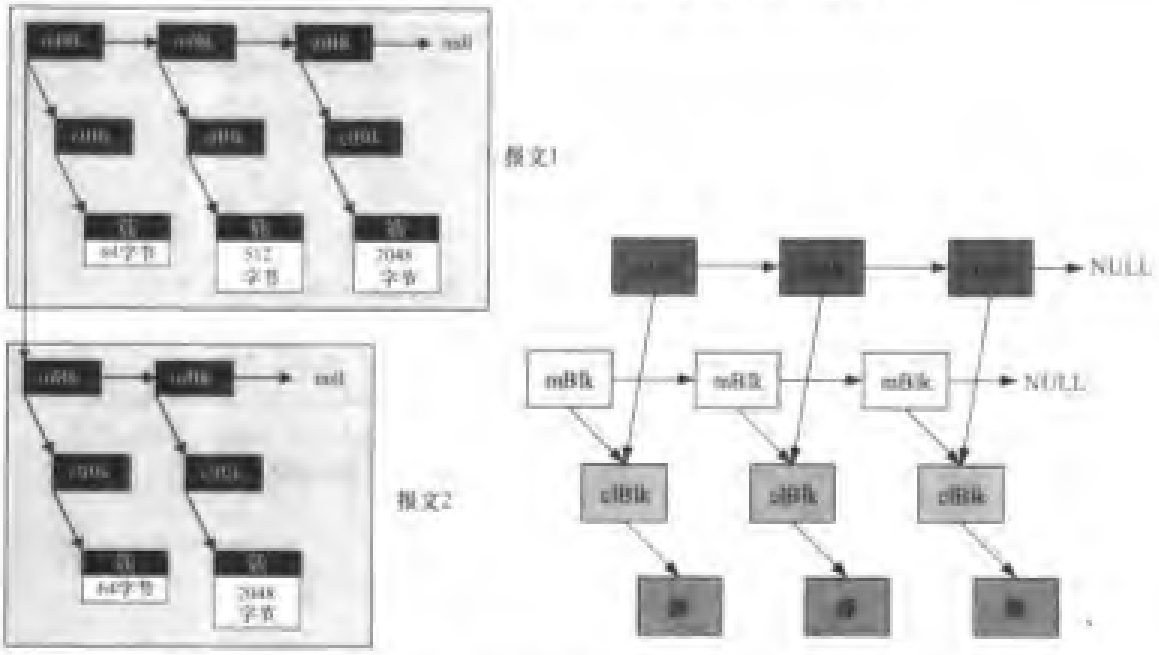


图 4-4　mBlks 和 clBlks 数据结构

图 4-4 中，当多个 mBlk 指向同一个 clBlk 时，通常发生在报文被复制时，这种方法避免了缓冲区数据拷贝。

对于网络设备驱动程序设计，为简化处理，各簇的尺寸可取相同的字节数，因此整个缓冲池的分配可如图 4-5 所示。

mBlks 和 clBlks 定义在 netBufLib.h 中，如下：

```
typedef union clBlkList
    {
    struct clBlk *  pClBlkNext;     /* pointer to the next clBlk */
    char *          pClBuf;         /* pointer to the buffer */
    } CL_BLK_LIST;
```

图 4-5 设备驱动程序缓冲池分配

其中，成员 pClBuf 指向簇缓冲。

```
typedef struct clBlk
    {
    CL_BLK_LIST     clNode;         /* union of next clBlk, buffer ptr */
    UINT            clSize;         /* cluster size */
    int             clRefCnt;       /* reference count of the cluster */
    FUNCPTR         pClFreeRtn;     /* pointer to cluster free routine */
    int             clFreeArg1;     /* free routine arg 1 */
    int             clFreeArg2;     /* free routine arg 2 */
    int             clFreeArg3;     /* free routine arg 3 */
    struct netPool *pNetPool;       /* pointer to the netPool */
    } CL_BLK;
```

其中，clSize 表示 clNode. pClBuf 对应缓冲的大小。

```
typedef struct mHdr
    {
    struct mBlk *   mNext;          /* next buffer in chain */
    struct mBlk *   mNextPkt;       /* next chain in queue/record */
    char *          mData;          /* location of data */
    int             mLen;           /* amount of data in this mBlk */
    UCHAR           mType;          /* type of data in this mBlk */
    UCHAR           mFlags;         /* flags; see below */
    USHORT          reserved;
    } M_BLK_HDR;

typedef struct  pktHdr
    {
    struct ifnet *  rcvif;          /* rcv interface */
    int             len;            /* total packet length */
    } M_PKT_HDR;

typedef struct mBlk
    {
    M_BLK_HDR       mBlkHdr;        /* header */
    M_PKT_HDR       mBlkPktHdr;     /* pkthdr */
    CL_BLK *        pClBlk;         /* pointer to cluster blk */
    } M_BLK;
```

其中，成员 pClBlk 指向对应的 CL_BLK，mBlkHdr->mData 指向 pClBlk ->clNode. pClBuf，mBlkHdr->mLen 等于 pClBlk->clSize。其中 mBlkHdr->mData 在报文发送中可能需要拷贝到驱动缓冲中。

4.1.3　设备描述数据结构

MUX 通过 END_OBJ 数据结构来确定设备名及控制结构，END_OBJ 数据结构定义在 end.h 中，如图 4-6 所示，包括设备名称、设备编号和设备描述等。

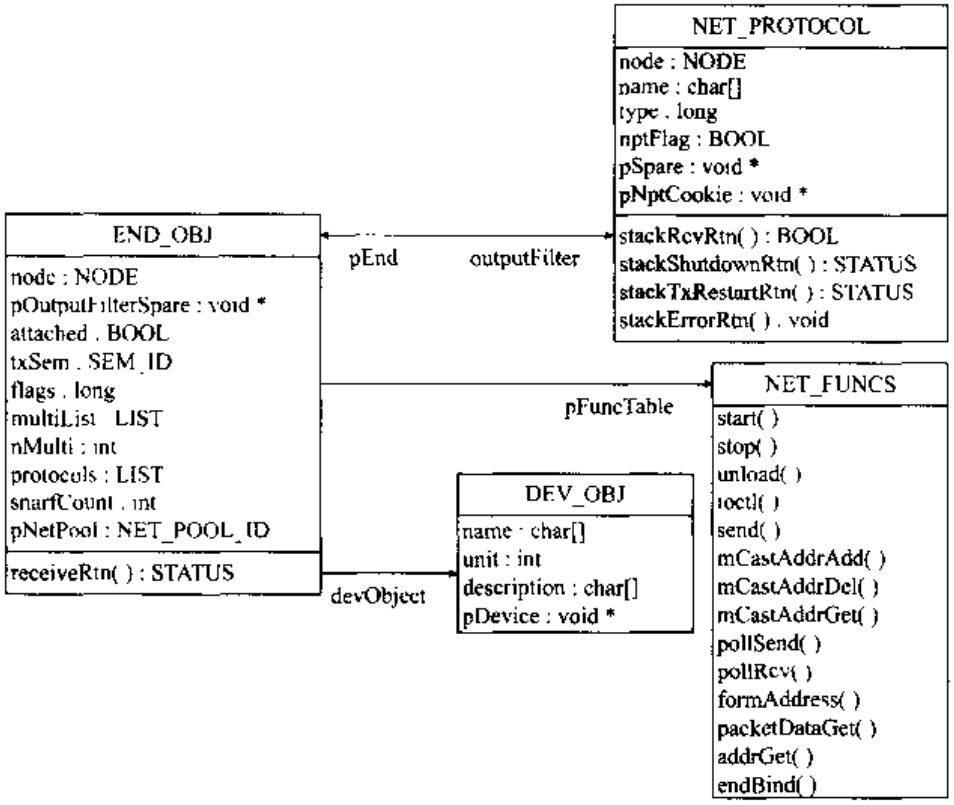


图 4-6　设备控制数据结构示意图

设备驱动程序通常定义一个设备控制结构，形式如下：

```
typedef struct XXXDevice
    {
    END_OBJ          end;           /* The class we inherit from */
    int              unit;          /* unit number of the device */
    UINT32           memAddr;       /* device structure address */
    UINT32           ioAdrs;        /* device structure address */
    int              ivec;          /* interrupt vector */
    int              ilevel;        /* interrupt level */
    UINT8            enetAddr[6];   /* ethernet address */
    CACHE_FUNCS      cacheFuncs;    /* cache function pointers */
    BOOL             txBlocked;     /* transmit flow control */
    CL_POOL_ID       pClPoolId;     /* cluster pool Id */
    } XXX_END_DEVICE;
    XXX_END_DEVICE *pDrvCtrl;
```

其中，第一个成员就是 END_OBJ 类型，成员 unit 表示是该驱动支持的第几块网卡。其他可能还包含设备硬件的一些信息，如网卡物理地址、PCI 影射的内存地址、IO 地址和中断号等。

驱动程序在加载时，可以直接调用 endObjInit 对成员 end 初始化。

```
endObjInit( &pDrvCtrl->end, (DEV_OBJ *)pDrvCtrl, DEV_NAME,
```

```
pDrvCtrl->unit, &xxxFuncTable,
                "XXX Enhanced Network Driver");
```

END_OBJ 结构中有一个类型为 DEV_OBJ 的成员变量 devObject，将指向 endObjInit 调用中的第二个参数 pDrvCtrl。调用中第三个参数 DEV_NAME 应该是一个指向该网卡名称的字符串指针，如 3COM 3C905B 网卡的名称为“elPci”，也就是网络引导时引导行中引导设备的名称。第五个参数 xxxFuncTable 是一个函数指针数组，指向设备驱动提供的一系列函数调用。

XXX_END_DEVICE、END_OBJ 与 DEV_OBJ 的关系如图 4-7 所示。

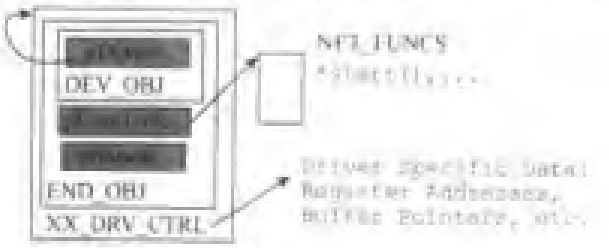


图 4-7　XXX_END_DEVICE、END_OBJ 与 DEV_OBJ 的关系

4.1.4　END 驱动框架

一个 END 驱动程序可以分为两大部分：BSP 部分和驱动主体。例如，对 3COM 3C905B 网卡，其 BSP 部分对应的文件是放在 BSP 目录下的 sysEl3c90xEnd.c，驱动主体则是放在源文件目录 target\src\drv\end\elPci3c90xend.c。

4.1.4.1　BSP 部分主要的工作

（1）PCI 设备初始化。

1）依据 PCI 设备号和厂商号查找设备，确定并记录该设备使用的中断、IO、存储器等硬件资源。

2）向系统申请设备需要使用资源，主要是影射 PCI 内存空间。

3）使能存储器和 IO 访问。

（2）实现 sysXXXEndLoad 函数，供 muxDevLoad 函数调用。

muxDevLoad 将连续两次调用 sysXXXEndLoad，sysXXXEndLoad 调用驱动主体函数提供的 XXXEndLoad 函数；第一次调用将返回该驱动对应的设备名称，如“elPci”，muxDevLoad 检查对应的设备是否已存在，避免重复加载驱动。如果不存在，muxDevLoad 再次调用 sysXXXEndLoad，并将 unit 值传递给 sysXXXEndLoad 的参数串中，sysXXXEndLoad 依据 unit 值，定位对应编号的网卡硬件资源（中断、IO、存储器等），将这些以格式化的字符串形式作为参数，传递给 XXXEndLoad，后者完成对将要使用设备的初始化和驱动注册功能。

4.1.4.2　驱动主体

这部分部较复杂，主要有如下工作：

（1）实现一个与 BSP 部分的接口函数 XXXEndLoad，该函数供 sysXXXEndLoad 调用。主要完成设备存储器初始化、硬件的初始化和向 MUX 层注册该 END 驱动功能。

该函数主要完成以下工作：

- 初始化设备和接口。
- 分配并填充 END_OBJ 结构。

- 初始化必要的私有结构。
- 分析处理初始化串。
- 创建并初始化私有内存池。
- 调用 netBufLib API 分配并初始化私有内存池。
- 创建 MIB-II 接口表。
- 填充 NET_FUNCS 表。

（2）实现 END 驱动要求的设备函数，如表 4-1 所示。

表 4-1　END 驱动要求的设备函数

函　数	功　能
xxxStart	启动函数：硬件初始化，连接中断，设置设备状态 Flag 为 UP
xxxStop	禁止设备中断，断开设备中断，设置设备状态 Flag 为 Down
xxxUnload	调用 endObjectUnload，卸载设备驱动，释放资源
xxxIoctl	设备控制函数
xxxSend	数据报文发送函数
xxxMCastAddrAdd	组播地址增加函数
xxxMCastAddrDel	组播地址删除函数
xxxMCastAddrGet	获得组播地址函数
xxxPollSend	查询方式发送函数
xxxPollReceive	查询方式接收函数
endEtherAddressForm	将给定的源地址和目标地址，格式化成网络缓冲要求的格式，返回 M_BLK_ID 指针。该函数和后面的两个函数，直接使用 endlib 库函数
endEtherPacketDataGet	从给定的网络缓冲中得到报文数据指针
endEtherPacketAddrGet	从给定的网络缓冲中得到地址信息

这些函数将作为一个 NET_FUNCS 类型结构的成员，该结构指针被赋值给 end 对象的成员 pFuncTable。这些函数的具体功能及函数原型本章的后续部分结合驱动程序的具体实现将作详细介绍。

（3）实现中断处理程序，完成硬件中断的处理，主要有：报文接收中断、发送中断和状态中断等。

4.2　ns83820 芯片简介

当前大部分 PCI 以太网控制器芯片通常由以下几部分组成：PCI 总线接口、传输 FIFO、接收 FIFO、E^2PROM 和 MAC 层接口等，如图 4-8 所示。

PCI 总线接口部分提供网卡芯片与 CPU 通过 PCI 总线软件控制和交换数据的接口。控制功能通过一组 I/O 寄存器实现的，这组寄存器通常包括：控制寄存器、状态寄存器等；数据交换通常采用 DMA 方式，这是因为网卡通信与低速异步串口不同，与 CPU 数据交换量大，DMA 功能由 PCI 网卡内部实现，控制 CPU 内存与片内存储器的数据交换。E^2PROM 存储以太网地

址和硬件配置信息，PCI 总线接口通过 I/O 寄存器提供对 E^2PROM 串行访问能力。FIFO 是用于芯片收发的临时缓冲。MAC 层提供报文收发的媒介控制功能，如 10/100M、半双工/全双工等。此外，芯片还提供中断控制功能，指示报文接收、报文发送、错误状态等事件发生，可以通过中断状态寄存器判别事件的类型。

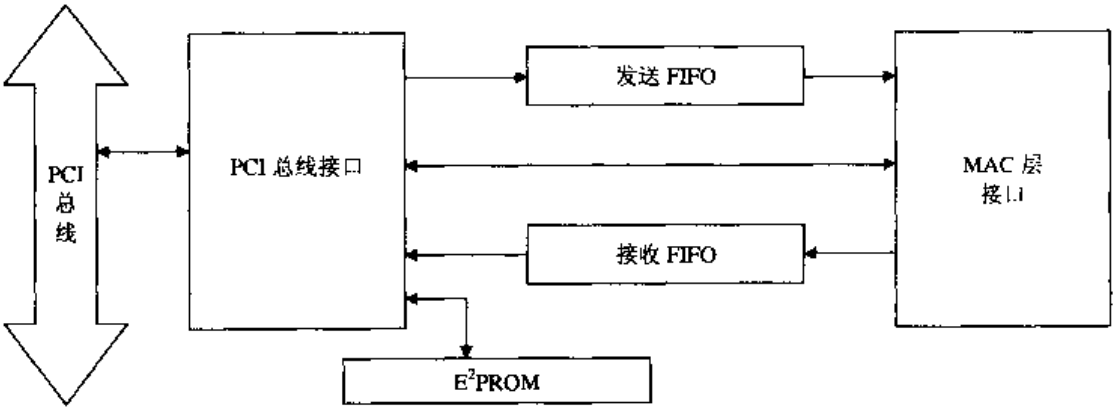


图 4-8　PCI 网卡体系结构

4.2.1　概述

ns83820 芯片功能块如图 4-9 所示。摘自 National Semiconductor《DP83820 10/100/1000 Mb/s PCI Ethernet Network Interface Controller》数据手册，详细信息请参见该手册，这里仅根据编写驱动基本需要摘要介绍有关内容。

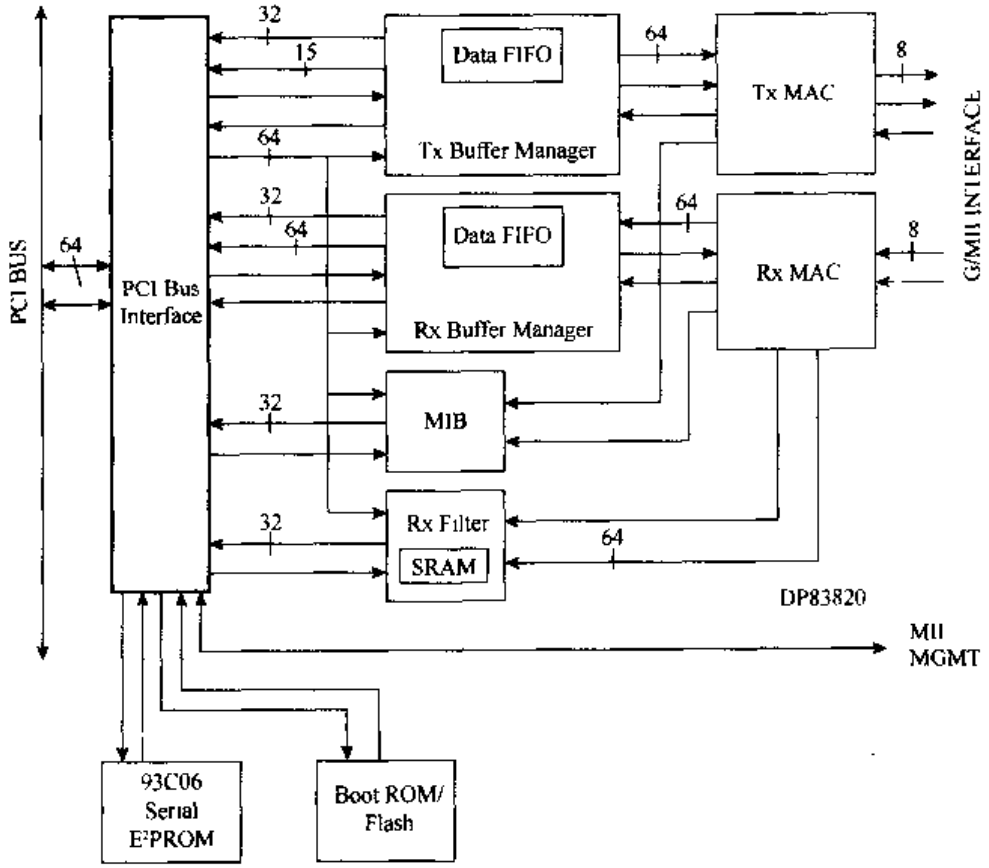


图 4-9　ns83820 芯片功能块图

图中可以看出，该网卡类似于大部分 PCI 网卡，有 PCI 总线接口、接收缓冲管理器、发送数据管理器、MAC、E^2PROM，此外还有一个 MIB 接口和接收过滤器。

该芯片 PCI 配置寄存器同其他 PCI 设备，其中偏移为 0x10 和 0x14 的寄存器分别包含芯片映射的 I/O 空间和存储器空间地址。

- 分离的接收和发送缓冲管理提供 FIFO 和 DMA 控制。
- MIB 是管理信息基本统计寄存器。
- E^2PROM 接口提供串行访问 NMC93C06 E^2PROM 的接口。
- 接收过滤器实现控制接收报文过滤能力。
- MAC 接口提供与 10/100/1000Mb/s 媒介访问功能。
- 物理层接口实现对 MII（Media Independent Interface）、GMI（Gigabit Media Independent Interface）、TBI（Ten-Bit Interface）等接口的支持。

4.2.2　ns83820 寄存器简介

当映射到 I/O 空间，ns83820 占用 256 个字节的 I/O 窗口；当映射到存储器空间，ns83820 占用 4KB 的内存窗口。主要的寄存器有：

表 4-2　ns83820 寄　存　器

偏移	简　称	说　　明	读/写
00H	CR	命令寄存器	R/W
04H	CFG	配置寄存器	R/W
08H	MEAR	MII/E^2PROM 访问寄存器	R/W
10H	ISR	中断状态寄存器	RO
14H	IMR	中断屏蔽寄存器	R/W
18H	IER	中断使能寄存器	R/W
20H	TXDP	传输描述符指针寄存器（低 32 位）	R/W
24H	TXDP_HI	传输描述符指针寄存器（高 32 位）	R/W
28H	TXCFG	传输配置寄存器	R/W
30H	RXDP	接收描述符指针寄存器（低 32 位）	R/W
34H	RXDP_HI	接收描述符指针寄存器（高 32 位）	R/W
38H	RXCFG	接收配置寄存器	R/W
3CH	PQCR	优先级排队控制寄存器	R/W
48H	RFCR	接收过滤/匹配控制寄存器	R/W
4CH	RFDR	接收过滤/匹配数据寄存器	R/W

1. 命令寄存器

表 4-3 命 令 寄 存 器

位	简 称	说 明	备 注
0	TXE	置 1 时，发送使能，激活发送状态机	当与 TXD 同为 1 时，忽略 TXE
1	TXD	置 1 时，发送禁止，停止发送状态机	
2	RXE	置 1 时，接收使能，激活接收状态机	当与 RXD 同为 1 时，忽略 RXE
3	RXD	置 1 时，接收禁止，停止接收状态机	
4	TXR	发送复位，清 FIFO 和发送数据	
5	RXR	接收复位，清 FIFO 和接收数据	
7	SWI	产生一个软件中断	
8	RST	复位 ns83820	

2. 配置和媒介访问寄存器

表 4-4 配置和媒介访问寄存器

位	简 称	说 明	备 注
1	BEM	Big Endian 方式	
18～20	PINT_CTL	物理中断状态： 1xx: DUPSTS 状态改变 x1x: LNKSTS 状态改变 xx1: SPDSTS 状态改变	
28	DUPSTS	全双工方式，1：全双工 0：半双工	
29～30	SPDSTS[1:0]	速度状态，1X：1000M X1：100M 00：10M	
31	LNKSTS	链路状态，1：链路 OK 0：链路 ERROR	

3. MII/E^2PROM 访问寄存器

表 4-5 MII/E^2PROM 访问寄存器

位	简 称	说 明	备 注
0	EEDI	E^2PROM 读	
1	EEDO	E^2PROM 写	
2	EECLK	E^2PROM 串行时钟	
3	EESEL	E^2PROM 芯片选择	
4	MDIO	MII 管理数据	
5	MDDIR	MII 管理方向	
6	MDC	MII 管理时钟	

4. 中断状态寄存器

表 4-6　中 断 状 态 寄 存 器

位	简　称	说　　明	备　　注
0	RXOK	指示接收状态机已经完成最后一个接收描述子的修改	
1	RXDESC	当一个接收描述子的 CMDSTS 的 INTR 位置 1 时，该位为 1	
4	RXIDL	接收状态机进入空闲状态	
6	TXOK	当发送状态机完成最后一个发送描述子的修改。	
7	TXDESC	当一个发送描述子的 CMDSTS 的 INTR 位置 1 时，该位为 1	
9	TXIDL	当发送状态机进入空闲状态	
12	SWI	当 CR 的 SWI 设置为 1 时，该位置 1	
14	PHY	该位置 1 表示物理状态改变	
21	RXRCMP	接收复位完成	
22	TXRCMP	发送复位完成	

5. 中断屏蔽寄存器

中断屏蔽寄存器的每一位控制相应的中断状态寄存器每位对应的事件是否能触发相应的中断。例如，bit1 为 1 时，允许接收状态机进入空闲状态时产生的相应的中断，为 0 时禁止。

6. 中断使能寄存器

该寄存器位 0 为 1 时允许硬件中断产生，为 0 时禁止。该位不影响 ISR 和 IMR。

7. 发送描述符指针寄存器（64 位）

该寄存器指向当前发送描述子。发送描述子地址必须是长字对齐（低八位为 0）。

8. 发送配置寄存器

定义了 ns83820 的发送配置，控制 loopback、心跳等。

表 4-7　发 送 配 置 寄 存 器

位	简　称	说　　明	备　　注
7～0	DRTH	32 字节的倍数，说明当 FIFO 中字节数超过该值时（或有一个完整报文），发送状态机将启动发送	
15～8	FLTH	32 字节的倍数，说明当 FIFO 中字节数超过该值时，发送状态机将请求 PCI 总线传输	
22～20	MXDMA	发送时，最大 DMA Burst 值 000 = 256 32-bit words (1024 bytes) 001 = 2 32-bit words (8 bytes) 010 = 4 32-bit words (16 bytes) 011 = 8 32-bit words (32 bytes) 100 = 16 32-bit words (64 bytes) 101 = 32 32-bit words (128 bytes) 110 = 64 32-bit words (256 bytes) 111 = 128 32-bit word (512 bytes)	
29	MLB	MAC 层 LoopBack 方式	
30	HBI	忽略 HeartBeat	

9. 接收描述符指针寄存器（64 位）

该寄存器指向当前接收描述子，接收描述子地址必须是 64 位对齐。

10. 接收配置寄存器

定义了 ns83820 的接收配置，控制是否接收错误报文、短报文等等。

表 4-8 接收配置寄存器

位	简 称	说 明	备 注
5～1	DRTH	8 字节的倍数，说明当 FIFO 中字节数超过该值时（或有一个完整报文），发送状态机将启动接收，从 FIFO 传输到主机内存	
22～20	MXDMA	接收时，最大 DMA Burst 值 000 = 256 32-bit words (1024 bytes) 001 = 2 32-bit words (8 bytes) 010 = 4 32-bit words (16 bytes) 011 = 8 32-bit words (32 bytes) 100 = 16 32-bit words (64 bytes) 101 = 32 32-bit words (128 bytes) 110 = 64 32-bit words (256 bytes) 111 = 128 32-bit word (512 bytes)	
28	RX_FD	接收时允许发送	
29	STRIPCRC	为 1 时，接收时硬件自动去掉 crc，并调整接收字节数	
30	ARP	为 1 时，接收短报文	
31	AEP	为 1 时，接收错误报文	

11. 优先级队列控制寄存器

表 4-9 优先级队列控制寄存器

位	简 称	说 明	备 注
0	TXPQEN	为 1 时，使能发送优先级队列	
1	TXFAIR	为 1 时，使能发送优先级队列公平调度	
3～2	RXPQ	接收优先级队列使能 00 - Disabled (one queue) 01 - Two queues (0,1) 10 - Three queues (0,1,2) 11 - Four queues (0,1,2,3)	

12. 接收过滤/匹配控制寄存器

用于控制 ns83820 的接收过滤逻辑。

表 4-10 接收过滤/匹配控制寄存器

位	简 称	说 明	备 注
9～0	RFADDR	当访问接收过滤/匹配数据寄存器时，选择内部的接收过滤寄存器 000h - PMATCH octets 1-0 002h - PMATCH octets 3-2 004h - PMATCH octets 5-4	

续表

位	简　称	说　　明	备　　注
9～0	RFADDR	Pattern Count Registers (PCOUNT) 006h - PCOUNT1, PCOUNT0 008h - PCOUNT3, PCOUNT2 SecureOn Password Register (SOPAS) 00Ah - SOPAS octets 1-0 00Ch - SOPAS octets 3-2 00Eh - SOPAS octets 5-4 Filter Memory 100h-3FEh - Rx filter memory (Hash table/pattern buffers)	
22	AARP	接收 ARP 报文	
29	AAM	接收所有 Multicast 报文	
30	AAB	接收时所有广播报文	
31	RFEN	接收过滤使能	

13. 接收过滤/匹配数据寄存器

用于读写 ns83820 的内部的接收过滤寄存器。

表 4-11　接收过滤/匹配数据寄存器

位	简　称	说　　明	备　　注
15～0	RFDATA	接收过滤数据	
17～16	BMASK	字节掩码	

4.2.3　ns83820 工作原理

ns83820 提供收发报文缓冲管理功能，驱动软件按照要求组织缓冲区，将收发缓冲区地址分别填写到接收/发送描述符指针寄存器，ns83820 将自动完成收发过程，并置相应的中断状态。这种功能既加快收发过程，最大可能利用 PCI 总线和最可能小地占用 CPU 时间，又方便软件编程。

4.2.3.1　描述子结构

描述子结构如表 4-12 所示（或表示 64 位总线时偏移）。

表 4-12　描 述 子 结 构 位 定 义

偏　　移	简　称	描　　述
0	LINK	连接到下一个描述子，BIT2-0 必须是 0
4 或 8	BUFPTR	数据缓冲，对于发送，可以是任意地址，对于接收必须 64 为对齐
8 或 0X10	CMDSTS	命令和状态
0X0c 或 0X14	EXSTS	可选的 32 位扩展状态

CMDSTS 的一些位对于接收和发送定义相同，如表 4-13 所示。

表 4-13 CMDSTS 的一些位对于接收和发送相同位定义

位	简 称	说 明	备 注
15～0	SIZE	描述子字节数目	
26～16		接收和发送定义不同，见表 4-14 和表 4-15	
27	OK	报文状态 OK	
28	SUPCRC INCCRC	禁止 CRC	
29	INTR	软件设置该位用于请求“描述符中断”	
30	MORE	为 1 时，表示该描述符非报文最后一个描述符。为 0 时，该描述符是报文的最后一个描述符	
31	OWN	为 1 时，数据生产者将描述符所有关系转移给消费者，为 0 时，消费者将描述符所有关系转移给生产者。发送时，驱动软件是生产者，ns83820 为消费者；接收时，ns83820 是生产者，驱动软件是消费	

发送 CMDSTS 的位 26～16 定义如表 4-14 所示。

表 4-14 CMDSTS 的发送位 26～16 位定义

位	简 称	说 明	备 注
19～16	CCNT	碰撞计数	
25	TFU	发送 FIFO 下溢	
26	TXA	发送废弃	

接收 CMDSTS 的位 26～16 定义如表 4-15 所示。

表 4-15 CMDSTS 的接收位 26～16 位定义

位	简 称	说 明	备 注
22～16		各类错误报文	
24～23	DEST	目的地址分类 00 - Packet was rejected 01 - Destination matched the Receive Filter Node Address Register 10 - Destination is a multicast (but not broadcast) 11 - Destination is a broadcast address	
25	RXO	接收 FIFO 溢出	
26	RXA	接收废弃	

单个描述符的示意图如图 4-10 所示。单个描述符 CMDSTS 的 MORE 位为 0。

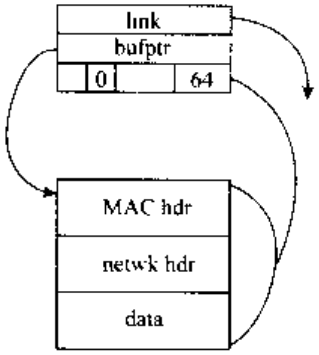


图 4-10　单个描述符的示意图

一个报文可以由多个描述符 LINK 域连接组成链表，图 4-11 是描述符链表示意图。

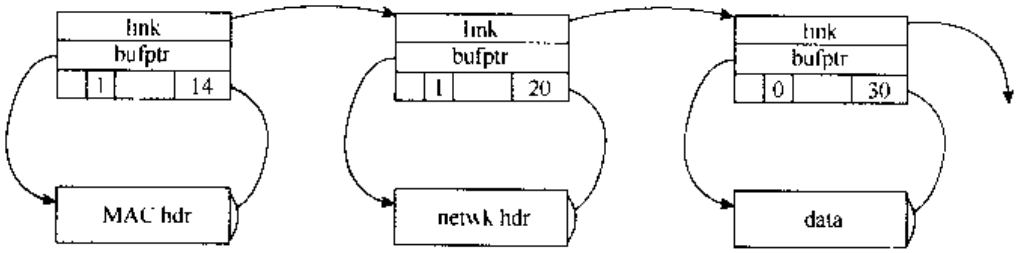


图 4-11　描述符链表示意图

图 4-12 是一个环状组织的描述符链表实例。网络驱动程序在设备初始化时，按照下图组织好描述子链表，把头指针地址写入接收/发送描述符指针寄存器。

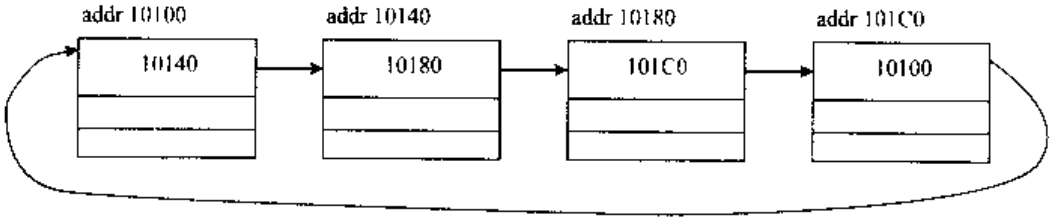


图 4-12　环状组织的描述符链表

4.2.3.2　发送过程

ns83820 支持两种发送队列：基于优先级的和无优先级的。简单起见，仅讨论无优先级的发送方式。无优先级队列的发送体系如图 4-13 所示，这时系统中仅有一个描述子队列。

发送描述子链的初始化工作在驱动初始化时完成，发送描述子链表的首地址已写入 TXDP 寄存器。

（1）驱动程序接收上层传递下来的数据包。

（2）驱动程序找到一个可用的发送描述子，并将数据包数据复制到该描述子中，置 cmdsts 的 CMDSTS_OWN 为 1，将数据包字节长度写入 cmdsts。

（3）置 CR 寄存器的 TXE 位为 1，激活发送状态机，启动发送。

（4）当数据包发送完成，发送状态机设置该描述子的 cmdsts 相应位，清 CMDSTS_OWN 为 0，产生发送完成中断。

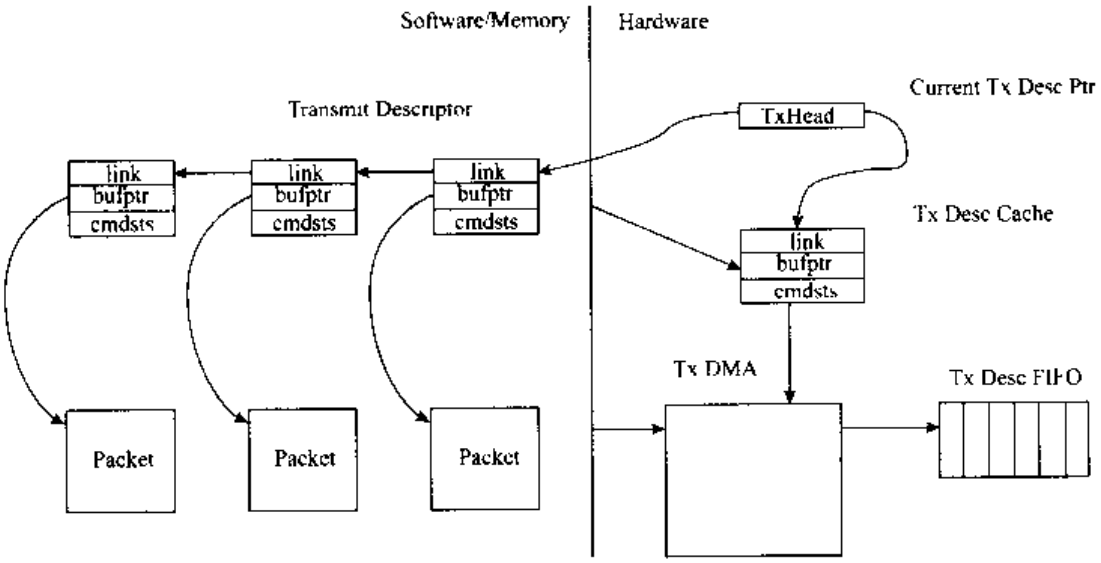


图 4-13　无优先级队列的发送体系

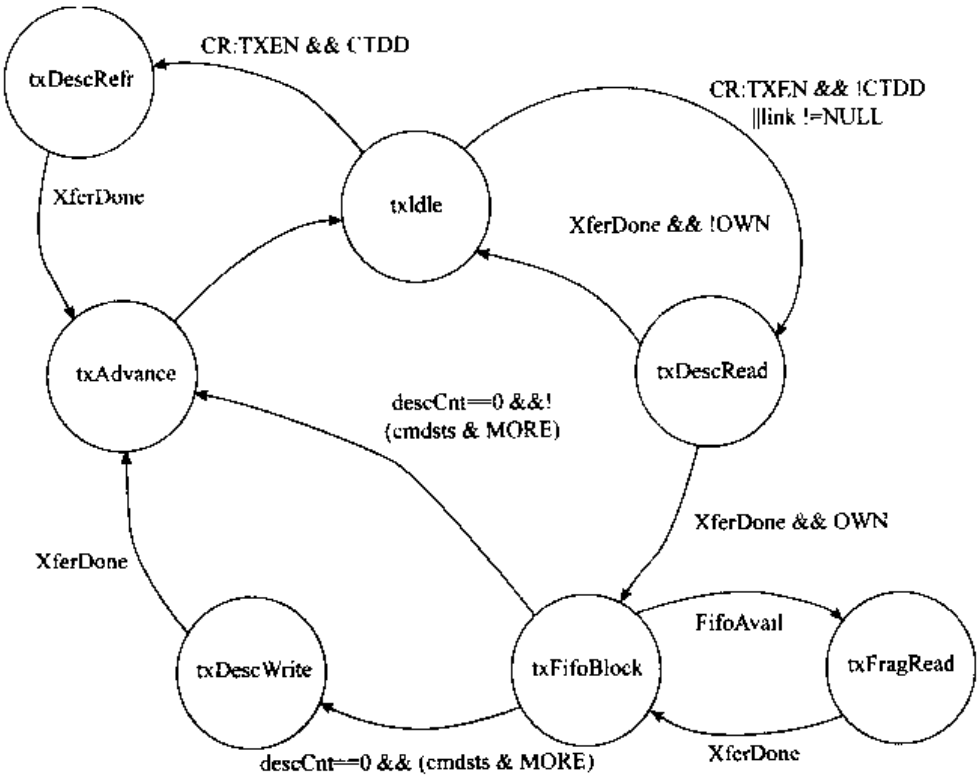


图 4-14　发送状态机状态变迁

4.2.3.3　接收过程

接收描述子链的初始化工作在驱动初始化时完成，CMDSTS_OWN 置 0，接收描述子链表的首地址已写入 RXDP 寄存器。

（1）83820 自动完成数据包接收工作，将数据填充到当前描述子的数据缓冲中，并置 cmdsts 的相应位，CMDSTS_OWN 置 1，如果允许产生接收中断，则触发接收中断，系统调用驱动程序中断处理程序。

（2）驱动软件可以采用查询或中断方式进行接收，本文讨论中断方式，驱动中断处理程序

判断中断状态，如果是接收中断，定位到相应描述子，并将数据拷贝到驱动接收缓冲区，提交给上层，最后清该描述子的 cmdsts 的相应位，CMDSTS_OWN 置 0。

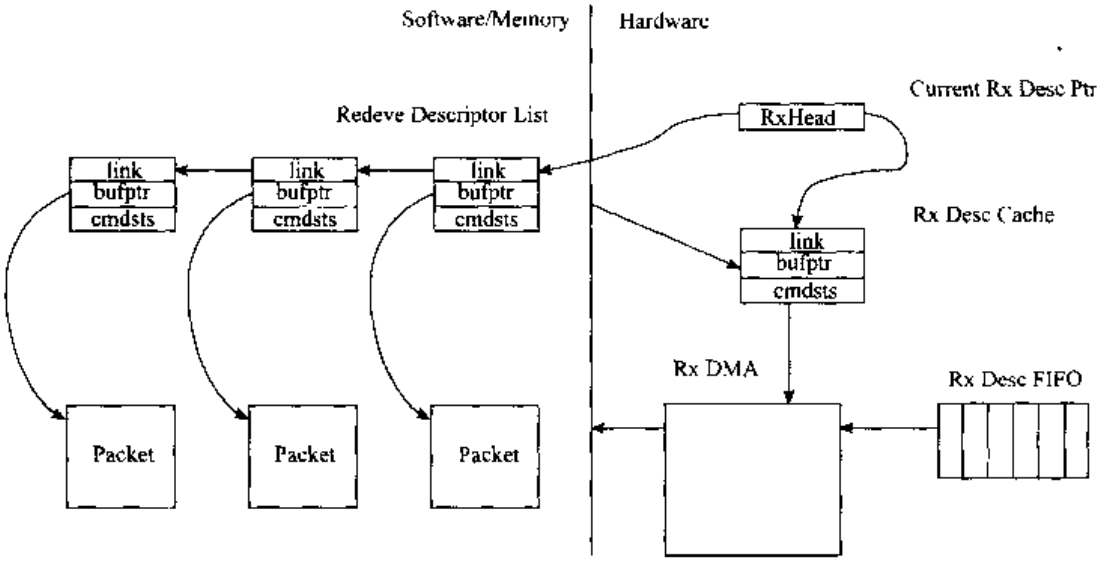


图 4-15　无优先级队列的接收体系

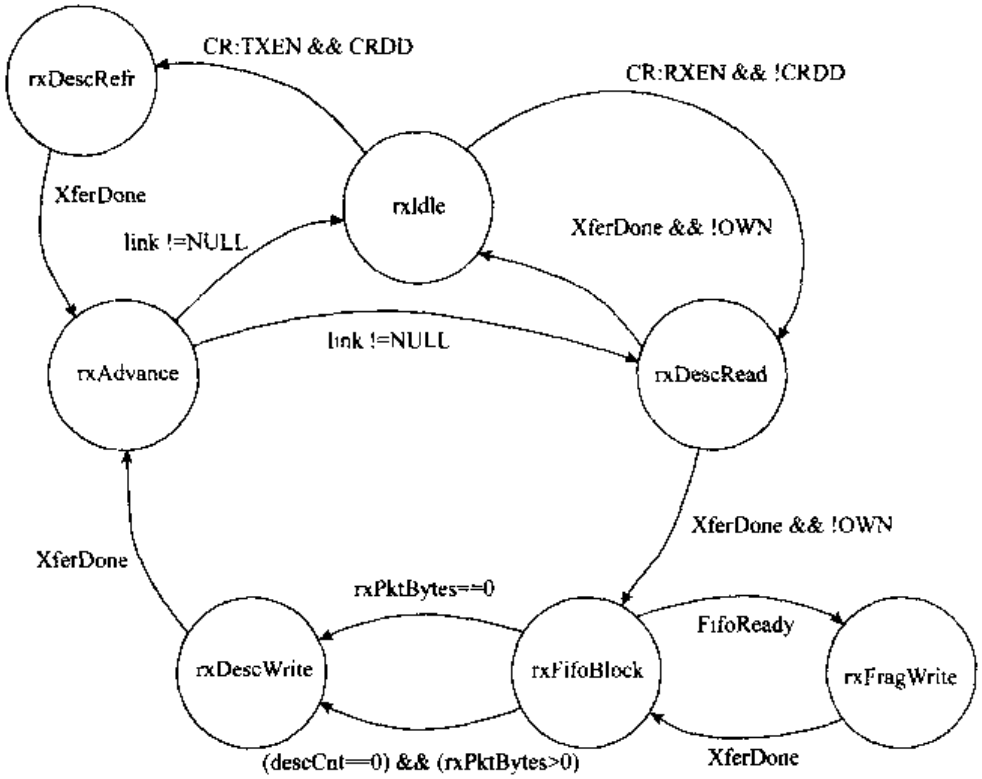


图 4-16　接收状态机状态变迁

4.3　ns83820 END 驱动实现

本节介绍驱动的实现过程，该驱动侧重于介绍几个主要的函数，可以正常通信，但没实现组播等功能。驱动程序是在 VxWorks 提供的 DEC21140 驱动框架基础上，移植了 Linux ns83820

驱动部分代码（主要是初始化、MDIO 访问、中断处理等）实现的。这是因为 ns83820 描述子存储器结构与 dec21140 芯片描述子类似。dec21140 芯片资料可由 Intel 网站上下载。

4.3.1 BSP 部分 sysNs83830End.c

4.3.1.1 数据结构

记录 ns83820PCI 配置头信息：

```
typedef struct ns83820PciResource       /* NS83820_PCI_RESOURCES */
    {
    UINT32      iobaseCsr;              /* Base Address Register 0 */
    UINT32      membaseCsr;             /* Base Address Register 1 */
    char        irq;                    /* Interrupt Request Level */
    UINT32      irqvec;                 /* Interrupt Request vector */
    UINT32      configType;             /* type of configuration */
    UINT32      boardType;              /* type of LAN board for this unit */
    UINT32      pciBus;                 /* PCI Bus number */
    UINT32      pciDevice;              /* PCI Device number */
    UINT32      pciFunc;                /* PCI Function number */
    } NS83820_PCI_RESOURCES;
```

4.3.1.2 网络设备查找及 PCI 初始化函数

该函数在 BSP 中调用，根据 PCI ID 号查找硬件设备，获得 PCI 配置头信息（IO 和 PCI 内存影射空间地址、中断级），调用 sysMmuMapAdd 将 PCI 内存空间地址影射插入到 sysPhysMemDesc 表中，并对该 PCI 设备简单初始化，使能 I/O、存储器访问，使能 PCI 设备总线 Master 方式。该函数是 ns83820 驱动所有函数中最早被调用的函数。

```
STATUS sysNs83820PciInit ()
    {
    UINT32      membaseCsr;
    UINT32      iobaseCsr;
    char        irq;
    int         pciBus, pciDevice,  pciFunc;
    int         ix,  unit,  found = 0;
    int         netEndUnits = 0;
    UINT32      boardType = NONE;       /* board type detected */

    /* 以下根据 NS83820 芯片的 VENDOR ID 和 DEVICE_ID 查找 PCI 设备 */

    for (unit = 0; unit < NS83820_MAX_DEV; unit++){
        if (pciFindDevice (  PCI_VENDOR_ID_NS, PCI_DEVICE_ID_NS,
                             unit, &pciBus, &pciDevice, &pciFunc) == OK) {
            found = TRUE;
            ns83820CardNum++;
            /* load up the PCI device table */
            pNs83820Rsrc [unit] = ns83820PciResrcs + netEndUnits;
            pNs83820Rsrc [unit]->pciBus    = pciBus;
            pNs83820Rsrc [unit]->pciDevice = pciDevice;
            pNs83820Rsrc [unit]->pciFunc   = pciFunc;
            pNs83820Rsrc [unit]->boardType = boardType;
```

```
            netEndUnits++;                  /* number of units found */
            break;
        }
    }

    if (found != TRUE)
    return (NULL);

    /* Now initialize all the units we found */

    for (unit = 0; ((unit < netEndUnits) && (unit < NELEMENTS (pNs83820Rsrc)));
unit++){
    /* Fill in the resource entry */

    pNs83820Rsrc [unit] = ns83820PciResrcs + unit;

    /* 以下代码获得设备的内存基地址、IO 基地址和中断级。本驱动是通过 PCI 存储器影射空间访问
寄存器的。*/
        pciConfigInLong   (pNs83820Rsrc[unit]->pciBus,   pNs83820Rsrc[unit]->
pciDevice,
            pNs83820Rsrc[unit]->pciFunc,PCI_CFG_BASE_ADDRESS_0, &iobaseCsr);

        pciConfigInLong   (pNs83820Rsrc[unit]->pciBus,   pNs83820Rsrc[unit]->
pciDevice,
            pNs83820Rsrc[unit]->pciFunc,PCI_CFG_BASE_ADDRESS_1, &membaseCsr);

        pciConfigInByte   (pNs83820Rsrc[unit]->pciBus,   pNs83820Rsrc[unit]->
pciDevice,
            pNs83820Rsrc[unit]->pciFunc,PCI_CFG_DEV_INT_LINE, &irq);

         /*
          * mask off registers. IO base needs to be masked off because bit0
          * will always be set to 1
          */

        membaseCsr   &= PCI_MEMBASE_MASK;
        iobaseCsr    &= PCI_IOBASE_MASK;

    /* 调用 sysMmuMapAdd 将 PCI 内存空间地址影射插入到 sysPhysMemDesc 表中*/
        if (sysMmuMapAdd ((void *)(membaseCsr & PCI_DEV_MMU_MSK),
                PCI_DEV_ADRS_SIZE,
                VM_STATE_MASK_VALID | VM_STATE_MASK_WRITABLE |
                VM_STATE_MASK_CACHEABLE,
                VM_STATE_VALID      | VM_STATE_WRITABLE |
                VM_STATE_CACHEABLE_NOT
                ) == ERROR)
            {
            /* for now, exit, but later break and stop where we're at when we're
detecting multiple units */
```

```
            return (ERROR);

            }

        /* 将对应的信息赋值给设备资源数据结构 */

        pNs83820Rsrc[unit]->membaseCsr   = membaseCsr;
        pNs83820Rsrc[unit]->iobaseCsr    = membaseCsr;   /* only for X86 */
                                                         /*iobaseCsr;*/
        pNs83820Rsrc[unit]->irq          = irq;

        /* 对 X86 中断向量为中断级加 0x20*/
        pNs83820Rsrc[unit]->irqvec       = irq + 0x20;
        /* enable mapped memory and IO addresses */
        /* 使能 I/O、存储器访问，使能 PCI 设备总线 Master 方式 */
        pciConfigOutWord (   pNs83820Rsrc[unit]->pciBus,
                             pNs83820Rsrc[unit]->pciDevice,
                             pNs83820Rsrc[unit]->pciFunc,
    PCI_CFG_COMMAND,      PCI_CMD_IO_ENABLE      |      PCI_CMD_MEM_ENABLE      |
PCI_CMD_MASTER_ENABLE);

        return OK;
    }
```

4.3.1.3 驱动加载函数

函数 sysNs83820EndLoad 是由 muxDevLoad 来调用，sysNs83820EndLoad 调用后面将提到的函数 ns83820EndLoad，并把网卡对应的信息：PCI 内存基地址、IO 基地址、中断级、中断向量等信息，以一个格式化串的形式传递给 ns83820EndLoad。

格式化串的形式如下：

```
"<unit number>:<device addr>:<PCI addr>:<ivec>:<ilevel>:<mem base>:<mem
size>:<user flags>:<offset>"
```

分别是：网卡编号、IO 基地址、PCI 内存基地址、中断向量、中断级、使用的内存地址、内存大小、用户标示、偏移。

该函数基本采用 dec21140 驱动相应的函数。实际上对 83820 驱动而言，<mem base>: <mem size>:<user flags>:<offset>等参数都没用到。

该函数正常返回一个 END_OBJ 对象指针。

对该函数的理解应同步参考 ns83820 EndLoad 及相应的参数分析函数 ns83820InitParse。

代码如下：

```
END_OBJ * sysNs83820EndLoad
    (
    char * pParamStr,    /* ptr to initialization parameter string */
    void * unused        /* unused optional argument */
    )
    {
    /*
     * The ns83820 driver END_LOAD_STRING should be:
```

```
 "<unit number>:<device addr>:<PCI addr>:<ivec>:<ilevel>:<mem base>:\
<mem size>:<user flags>:<offset>"
 * Note that unit string is prepended by the mux, so we
 * don't put it here.
 */
int unit_number;
char * pStr = NULL;
char paramStr [200];
static char ns83820ParamTemplate [] =
       "0x%x:0x%x:0x%x:0x%x:0x%x:0x%x:0x%x:0x%x:0x%x:0x%x";
END_OBJ * pEnd;

if(ns83820CardNum<0)
    return NULL;

if (strlen (pParamStr) == 0)
    {
   /*
    * muxDevLoad() 将调用本函数两次，第一次传递进来的参数 pParamStr 长度为 0，这时
      调用 ns83820EndLoad ，将网卡设备驱动的名字反填到 pParamStr，返回给
      muxDevLoad。
    */
    pEnd = (END_OBJ *) ns83820EndLoad (pParamStr);
    }
else
    {
   /*
    * 当 pParamStr 长度非零时，表示 muxDevLoad() 第二次调用本函数，应该将对应 unit
      网卡的资源信息格式化成要求的串，传递给 ns83820EndLoad 。
    */

/* Work out the Unit Number to initialise */

    unit_number = atoi (pParamStr);
    if ((unit_number < 0) || (unit_number > NELEMENTS (ns83820Brds)))
        {
        return NULL;
        }
    else
        {
        pStr = strcpy (paramStr, pParamStr);

        /* Now, we get to the end of the string */

        pStr += strlen (paramStr);

        /* 完成参数串的初始化 */
        sprintf (pStr, ns83820ParamTemplate,
                  /* device Io base */
                  (UINT)     pNs83820Rsrc[unit_number]->iobaseCsr        +
```

```
CPU_PCI_IO_ADRS,
                        /* device memory IO base */
                        (UINT)      pNs83820Rsrc[unit_number]->membaseCsr      +
CPU_PCI_MEM_ADRS,
                    /*  PCI2DRAM_BASE_ADRS,     */      /* pciMemBase */
                        pNs83820Rsrc[unit_number]->irqvec,
                                                    /* interrupt IRQ vector */
                        pNs83820Rsrc[unit_number]->irq,
                                                    /* interrupt irq number */
                        NONE,
                        NONE,
                        0 /*pNs83820Rsrc [unit_number]->boardType*/,
                                                    /* user flags */
                        0,                          /* offset */
                        NS83820_BUFF_MTPLR          /* flags */
                        );
                                                    /* 调用 ns83820EndLoad */
          if ((pEnd = (END_OBJ *) ns83820EndLoad (paramStr)) == (END_OBJ *)ERROR)
                {
                logMsg ("Error: ns83820EndLoad failed to load driver\n",
                          0, 0, 0, 0, 0, 0);
                ;
              }
            }

        return (pEnd);
        }
```

4.3.2 驱动主体 ns83820End.c

该程序是驱动主体，完成驱动加载，硬件初始化，实现收发过程及中断处理等功能。

4.3.2.1 数据结构

结构 ns83820_drv_ctrl 定义了该网络驱动设备需要的所有信息。

```
typedef struct ns83820_drv_ctrl
    {
    END_OBJ      endObj;            /* END 对象，是每个 END 网络驱动程序必须有的。*/
    int          flags;             /* 驱动标志 */
    int          unit;              /* 指示是第几块网卡 */
    ULONG        devAdrs;           /* IO 基地址 */
    int          ivec;              /* 中断向量 */
    int          ilevel;            /* 中断级 */
    char *       memBase;           /* 分配给驱动的描述符内存基地址 */
    ULONG        memSize;           /* 描述符内存池大小 */
    ULONG        pciMemBase;        /* PCI 映射内存基地址 */
    ULONG        usrFlags;          /* 用户标示 */
    int          offset;            /* 偏移量，本驱动为 0 */
                                    /* 以下是记录驱动硬件初始化信息 */
    ULONG        CFG_cache;         /* 配置寄存器值 */
    ULONG        IMR_cache;         /* 中断屏蔽 */
```

```
    ULONG       linkstate;          /* 连路状态：10/100/1000M、双工/半双工 */
    struct eeprom   ee;             /* 保存 EEPROM 内容 */

    int         numRds;             /* 接收描述子数目 */
    int         rxIndex;            /* 接收描述子索引 */
    int         rxFreeIndex;        /* 自由接收描述子索引*/
    int         rxIdle;             /* 接收状态机是否空闲 */
    NS_DESC *   rxRing;             /* 接收描述子链指针 */
    ULONG       rxAddr;             /* 分配给接收内存地址 */

    int         numTds;             /* 发送描述子数目 */
    int         txIndex;            /* 发送描述子索引 */
    int         txDiIndex;          /*  需要释放的发送描述子索引 */
    int         txIdle;             /* 发送状态机是否空闲 */
    NS_DESC *   txRing;             /* 发送描述子链指针 */
    ULONG       txAddr;

    BOOL        rxHandling;         /* 接收包处理任务正在运行标志 */
    BOOL        txCleaning;         /* 发送队列清除任务运行标示 */

    CACHE_FUNCS cacheFuncs;         /* cache 函数指针 */
    BOOL        txBlocked;          /* 发送阻塞标志，当发送描述子不够用时置该标志 */
    FREE_BUF    freeBuf[128];       /* 待释放的发送描述子 */
    CL_POOL_ID  clPoolId;           /* 缓冲池簇指针 */
    UCHAR       eAdrs[6];           /* 以太网 MAC 地址 */
} DRV_CTRL ;
```

驱动程序将使用一个 DRV_CTRL 类型的指针 pDrvCtrl，作为函数参数，获取或记录驱动设备信息。每个函数通过 pDrvCtrl 获得设备寄存器基地址，来访问寄存器。

4.3.2.2 寄存器访问函数

读寄存器函数：ns83820CsrRead。该函数读取命令或状态寄存器，返回其值。

这里采用 PCI 影射内存来访问寄存器。

```
LOCAL ULONG ns83820CsrRead
    (
    ULONG    devAdrs,               /* 设备寄存器基地址 */
    int      reg                    /* 寄存器偏移 */
    )
    {
    ULONG *  csrReg, csrData;

    csrReg = (ULONG *)(devAdrs + reg);
    csrData = *csrReg;
    return (PCISWAP (csrData));
    }
```

写寄存器函数：ns83820CsrWrite。该函数向寄存器写入特定的值。

同样采用 PCI 影射内存来访问寄存器。

```
LOCAL void ns83820CsrWrite
    (
```

```
    ULONG    devAdrs,                    /* 设备寄存器基地址 */
    int      reg,                        /* 寄存器偏移 */
    ULONG    value                       /* 待写入的值 */
    )
    {
    ULONG * csrReg = (ULONG *)(devAdrs + reg);
    *csrReg = PCISWAP (value);
    return;
    }
```

4.3.2.3 驱动初始化函数

驱动初始化函数 ns83820EndLoad 是 ns83820End.c 向系统提供唯一全局接口函数，由前面提到的驱动加载函数 sysNs83820EndLoad 调用。该函数对输入参数进行分析，如果参数串第一个字符位 '\0'，则将驱动名称拷贝到参数串返回；否则对输入参数进一步分析，初始化 END 对象、初始化内存池、获取 MAC 地址、初始化 MIB-II、置驱动对象 ready 标志、返回 END 对象指针。MIB 是管理信息数据库的缩写（Management Information Base, MIB），相关信息请参见 RFC 1156 和 RFC 1213。

代码如下：

```
END_OBJ * ns83820EndLoad
    (
    char *       initStr                 /* 格式化参数串 */
    )
    {
    DRV_CTRL *   pDrvCtrl;
    char eAdrs[EADDR_LEN];               /* ethernet address */
    int          retVal;
    char bucket[sizeof(ULONG)];
/* 开始的几行代码所有 END 驱动通用，基本不用修改，理解则需要结合 sysNs83820EndLoad 和 muxDevLoad 函数。*/
    if (initStr == NULL)
        return (NULL);
    if (initStr[0] == '\0')              /* 第一次调用返回驱动名称，这里是“ns” */
        {
        memcpy(initStr, (char *) DRV_NAME, DRV_NAME_LEN);
        return (0);
        }

    /* 分配驱动设备控制结构 */
    pDrvCtrl = calloc (sizeof(DRV_CTRL), 1);
    if (pDrvCtrl == NULL)
        {
        LOG_MSG ("%s - Failed to allocate control structure\n",
                (int) DRV_NAME, 0, 0, 0, 0, 0);
        return (NULL);
        }

#ifdef DRV_DEBUG
    pDrvCtrlDbg = pDrvCtrl;              /* 调试时使用 */
```

```
#endif

    pDrvCtrl->flags = 0;
    pDrvCtrl->txCleaning = FALSE;
    pDrvCtrl->rxHandling = FALSE;
```

调用 ns83820InitParse 分析初始化参数串，大部分 END 驱动类似，只要参数串格式化形式一致，基本可借用。

```
    if (ns83820InitParse (pDrvCtrl, initStr) == ERROR)
        {
        LOG_MSG ("%s - Failed to parse initialization parameters\n",
               (int) DRV_NAME, 0, 0, 0, 0, 0);
        return (NULL);
        }
    /* 检测寄存器所在内存是否可访问，PCI 影射内存是在 PCI 初始化时完成。。*/
    if (vxMemProbe ((char *) pDrvCtrl->devAdrs, VX_READ,
        sizeof(ULONG), &bucket[0]) != OK)
    {
    LOG_MSG ("%s%d - need MMU mapping for address %#x\n",
         (int)DRV_NAME, pDrvCtrl->unit, (int)pDrvCtrl->devAdrs, 0, 0, 0);
    return (NULL);
    }

    /*  调用宏 END_OBJ_INIT 初始化 END 对象*/

    if (END_OBJ_INIT (&pDrvCtrl->endObj, (DEV_OBJ*)pDrvCtrl, DRV_NAME,
                    pDrvCtrl->unit, &netFuncs,
                    "ns83820 10/100/1000m Enhanced Network Driver") == ERROR)
        {
        LOG_MSG ("%s%d - Failed to initialize END object\n",
               (int) DRV_NAME, pDrvCtrl->unit, 0, 0, 0, 0);
        return (NULL);
        }

    /* 内存池初始化，后面做详细分析 */
    if (ns83820InitMem (pDrvCtrl) == ERROR)
        goto error;

    /* 读 MAC 地址，必须放在 END_MIB_INIT 之前。 */
    ns83820_getmac(pDrvCtrl,eAdrs);

    /* 初始化 MIB-II entries */

    if (END_MIB_INIT (&pDrvCtrl->endObj, M2_ifType_ethernet_csmacd,
                    (UINT8*) eAdrs, EADDR_LEN,
                     ETHERMTU, NS_SPEED_DEF) == ERROR)
        {
        LOG_MSG ("%s%d - MIB-II initializations failed\n",
               (int) DRV_NAME, pDrvCtrl->unit, 0, 0, 0, 0);
        goto error;
```

```
        }

    /* 置驱动缺省标志 */

    END_OBJ_READY (&pDrvCtrl->endObj,
                   IFF_NOTRAILERS | IFF_MULTICAST | IFF_BROADCAST);

    /* 返回 */

    return (&pDrvCtrl->endObj);

    /* 出错处理，出错时调用 ns83820Unload，释放申请的资源*/
error:
        {
        ns83820Unload (pDrvCtrl);
        return (NULL);
        }
    }
```

4.3.2.4 内存初始化

内存初始化 ns83820InitMem 完成驱动内存初始化工作，主要是收发描述子内存的初始化：申请与构造。ns83820 与 DEC21140 芯片基本类似，区别在于 83820 支持 64 位总线，DMA 操作要求内存 64 位对齐，也就是描述子内存地址的低三位必须为 0。因此，该函数基本与 dec21140end.c 中的 dec21140InitMem 类似。

```
LOCAL STATUS ns83820InitMem
    (
    DRV_CTRL *  pDrvCtrl
    )
{
/* pRxD 、pTxD 分别指向接收和发送描述子链 */
    NS_DESC *    pRxD = pDrvCtrl->rxRing;
    NS_DESC *    pTxD = pDrvCtrl->txRing;
    M_CL_CONFIG dcMclBlkConfig;
    CL_DESC clDesc;                          /* cluster description */
    char *   pBuf;
    int      ix;
    int      sz;
    char *   pShMem;

    /* 构建驱动需要的共享内存 */
    /* 首先判断是否已指定共享内存地址，本例采用驱动申请方式 */
    if ((int)pDrvCtrl->memBase != NONE)     /* specified memory pool */
    {
    sz  = ((pDrvCtrl->memSize - (RD_SIZ + TD_SIZ)) /
              (((2 + NUM_LOAN) * NS_BUFSIZ) + RD_SIZ + TD_SIZ));
    pDrvCtrl->numRds = max (sz, MIN_RDS);
    pDrvCtrl->numTds = max (sz, MIN_TDS);
    }
else
```

```
/* 在本函数中申请，NUM_RDS_DEF和NUM_TDS_DEF分别是接收和发送描述子的数目 */
        {
        pDrvCtrl->numRds = NUM_RDS_DEF;
        pDrvCtrl->numTds = NUM_TDS_DEF;
        }


    switch ((int)pDrvCtrl->memBase)
        {
        default :               /* caller provided memory */
            break;
        case NONE :             /* get our own memory */

/* 首先计算共享内存的大小sz，然后调用cacheDmaMalloc分配DMA需要的地址连续的内存块 */

            if (!CACHE_DMA_IS_WRITE_COHERENT ())
                {
                LOG_MSG ( "%s%d: device requires cache coherent memory\n",
            (int) DRV_NAME, pDrvCtrl->unit, 0, 0, 0, 0);
                return (ERROR);
                }

            sz = (((pDrvCtrl->numRds + 1) * RD_SIZ) +
                  ((pDrvCtrl->numTds + 1) * TD_SIZ));

            pDrvCtrl->memBase =
                pShMem = (char *) cacheDmaMalloc ( sz );

            if (pShMem == NULL)
                {
                LOG_MSG ( "%s%d - system memory unavailable\n",
                        (int) DRV_NAME, pDrvCtrl->unit, 0, 0, 0, 0);
                return (ERROR);
                }

            pDrvCtrl->memSize = sz;

            DRV_FLAGS_SET (NS_MEMOWN);

                            /* copy the DMA structure */

            pDrvCtrl->cacheFuncs = cacheDmaFuncs;

            break;
        }
/* 共享内存初始化：首先清零 */
    memset (pShMem, 0, (int) sz);
```

/* 构造接收DMA使用内存，也即接收描述子基地址，注意ns83820支持64位地址总线，要求地址的低三位为0，该地址将写入接收描述子地址寄存器 */

```
    pDrvCtrl->rxAddr = ((int)pShMem + 0x07) & ~0x07;
    pRxD =pDrvCtrl->rxRing = (NS_DESC *) (((int)pShMem + 0x07) & ~0x07);

/* 构造发送 DMA 使用内存，也即发送描述子基地址，该地址将写入接收描述子地址寄存器 */
    pDrvCtrl->txAddr = (int)(pDrvCtrl->rxRing + pDrvCtrl->numRds);
    pTxD =pDrvCtrl->txRing = (NS_DESC *) (pDrvCtrl->rxRing + pDrvCtrl->numRds);

/* 初始化网络接收发送内存池，这部分代码借用 dec21140 相关代码 */

    memset ((char *)&dcMclBlkConfig, 0, sizeof(dcMclBlkConfig));
    memset ((char *)&clDesc, 0, sizeof(clDesc));

    dcMclBlkConfig.mBlkNum  = pDrvCtrl->numRds * 4;
    clDesc.clNum            = pDrvCtrl->numRds + pDrvCtrl->numTds + NUM_LOAN;
    dcMclBlkConfig.clBlkNum = clDesc.clNum;

    /*
     * mBlk and cluster configuration memory size initialization
     * memory size adjusted to hold the netPool pointer at the head
     */

    dcMclBlkConfig.memSize  = ((dcMclBlkConfig.mBlkNum *
                              (MSIZE + sizeof (long))) +
                              (dcMclBlkConfig.clBlkNum *
                              (CL_BLK_SZ + sizeof (long))));

    if ((dcMclBlkConfig.memArea = (char *)memalign(sizeof (long),
                                  dcMclBlkConfig.memSize)) == NULL)
        return (ERROR);

    clDesc.clSize   = NS_BUFSIZ;
    clDesc.memSize  = ((clDesc.clNum * (clDesc.clSize + 8)) + 8);

    if (DRV_FLAGS_ISSET(NS_MEMOWN))
        {
        clDesc.memArea = (char *) cacheDmaMalloc (clDesc.memSize);
        if (clDesc.memArea == NULL)
            {
            LOG_MSG ( "%s%d - system memory unavailable\n",
                    (int) DRV_NAME, pDrvCtrl->unit, 0, 0, 0, 0);
            return (ERROR);
            }
        }
    else
        clDesc.memArea = (char *) (pDrvCtrl->txRing + pDrvCtrl->numTds);

    if ((pDrvCtrl->endObj.pNetPool = malloc (sizeof(NET_POOL))) == NULL)
        return (ERROR);

    /* 初始化发送网络缓冲池，借用 21140 相关代码 */
```

```
    if (netPoolInit (pDrvCtrl->endObj.pNetPool, &dcMclBlkConfig,
                &clDesc, 1, NULL) == ERROR)
        {
        LOG_MSG ("%s%d - netPoolInit failed\n",
        (int) DRV_NAME, pDrvCtrl->unit, 0, 0, 0, 0);
        return (ERROR);
        }

    /* 保存 ID 号 */

    pDrvCtrl->clPoolId = clPoolIdGet (pDrvCtrl->endObj.pNetPool,
                                      NS_BUFSIZ, FALSE);
    /* 以下几个是接收和发送描述子索引，初始为 0 */

    pDrvCtrl->rxIndex=0;
    pDrvCtrl->txIndex=0;
    pDrvCtrl->rxFreeIndex=0;
    pDrvCtrl->txDiIndex=0;

    /* 建立接收描述子链 */

    for (ix = 0; ix < pDrvCtrl->numRds; ix++, pRxD++)
    {
        pBuf = (char *)malloc(1548);
        pBuf = (char *)(((int)pBuf+ 0x07)&~0x07);
        if (pBuf == NULL)
            {
            LOG_MSG ("%s%d - netClusterGet failed\n",
            (int) DRV_NAME, pDrvCtrl->unit, 0, 0, 0, 0);
            return (ERROR);
            }

/* 缓冲区指针 */
            pRxD->buf = PCISWAP (/*NS_VIRT_TO_PCI*/ ((ULONG)pBuf));
            pRxD->link = (ULONG)(pRxD+1);              /* 下一个描述子 */

            /* 命令与状态 */
            /* 芯片所有，缓冲大小，请求中断*/
            pRxD->cmdsts = PCISWAP ( REAL_RX_BUF_SIZE | CMDSTS_INTR);
            pRxD->vlansts = 0;    /* 扩展状态 */
    }
/* 最后一个 */
    pDrvCtrl->rxRing[pDrvCtrl->numRds-1].link=(int)&(pDrvCtrl->rxRing[0]);

    /* 建立发送描述子链 */

    for (ix = 0; ix < pDrvCtrl->numTds; ix++, pTxD++)
    {
        pTxD->link = (ULONG)(pTxD+1);
```

```
        /* empty -- no buffers at this time */
        pTxD->buf = 0;
        pTxD->cmdsts = PCISWAP (CMDSTS_INTR);        /* 驱动所有，请求中断 */
        pTxD->vlansts = 0;                           /* owner is host */
    }

/* 最后一个 */
    pDrvCtrl->txRing[pDrvCtrl->numTds-1].link=(int)&(pDrvCtrl->txRing[0]);

    /* Flush the write pipe */

    CACHE_PIPE_FLUSH ();

    return (OK);
    }
```

4.3.2.5 设备启动函数

设备启动函数 ns83820Start，连接中断处理程序，初始化硬件，置中断允许位，设置驱动状态为 UP。硬件初始化代码由 Linux ns83820 驱动移植而来。

```
LOCAL STATUS ns83820Start
    (
    DRV_CTRL *  pDrvCtrl
    )
    {
    int      retVal;
    int using_dac = 0;

    /* 在初始化硬件之前，首先连接中断处理程序 */

    SYS_INT_CONNECT (pDrvCtrl, ns83820Int, pDrvCtrl, &retVal);
    if (retVal == ERROR)
        return (ERROR);

    /* 禁止中断 */
    NS_CSR_WRITE( IMR, 0);
    NS_CSR_WRITE( IER,0 );
    NS_CSR_READ(IER);
    pDrvCtrl->IMR_cache = 0;

    setup_ee_mem_bitbanger(pDrvCtrl,&pDrvCtrl->ee, MEAR, 3, 2, 1, 0,   0);

    /* Reset the device */

    ns83820ChipReset (pDrvCtrl, CR_RST);

    NS_CSR_WRITE(PTSCR, PTSCR_EEBIST_EN);
    do {
        taskDelay(1);
    } while (NS_CSR_READ(PTSCR) & PTSCR_EEBIST_EN);
```

```
        NS_CSR_WRITE(PTSCR, PTSCR_EELOAD_EN);
        do {
            taskDelay(1);
        } while (NS_CSR_READ(PTSCR) & PTSCR_EELOAD_EN);

        /* 读配置寄存器，然后初始化配置寄存器 */
        pDrvCtrl->CFG_cache = NS_CSR_READ(CFG);

        if ((pDrvCtrl->CFG_cache & CFG_PCI64_DET)) {
            printf ("%s: detected 64 bit PCI data bus.\n",
                pDrvCtrl->endObj.devObject.name);
            /*pDrvCtrl->CFG_cache |= CFG_DATA64_EN;*/
            if (!(pDrvCtrl->CFG_cache & CFG_DATA64_EN))
                printf ("%s: EEPROM did not enable 64 bit bus.  Disabled.\n",
                    pDrvCtrl->endObj.devObject.name);
        } else
            pDrvCtrl->CFG_cache &= ~(CFG_DATA64_EN);

        pDrvCtrl->CFG_cache &= (CFG_TBI_EN  | CFG_MRM_DIS   | CFG_MWI_DIS |
                  CFG_T64ADDR | CFG_DATA64_EN | CFG_EXT_125 |
                  CFG_M64ADDR);
        pDrvCtrl->CFG_cache |= CFG_PINT_DUPSTS | CFG_PINT_LNKSTS | CFG_PINT_SPDSTS |
                 CFG_EXTSTS_EN | CFG_EXD | CFG_PESEL;
        pDrvCtrl->CFG_cache |= CFG_REQALG;
        pDrvCtrl->CFG_cache |= CFG_POW;
    #ifdef USE_64BIT_ADDR
        pDrvCtrl->CFG_cache |= CFG_M64ADDR;
    #endif
        if (using_dac)      /* 64 bit addr */
            pDrvCtrl->CFG_cache |= CFG_T64ADDR;

        /* Big endian mode does not seem to do what the docs suggest */
        pDrvCtrl->CFG_cache &= ~CFG_BEM;

        /* setup optical transceiver if we have one */
        if (pDrvCtrl->CFG_cache & CFG_TBI_EN) {
            printf      ("%s:      enabling      optical      transceiver\n",
pDrvCtrl->endObj.devObject.name);

            NS_CSR_WRITE(GPIOR, NS_CSR_READ(GPIOR) | 0x3e8 );

            /* setup auto negotiation feature advertisement */
            NS_CSR_WRITE(TANAR,   NS_CSR_READ(TANAR)   |   TANAR_HALF_DUP   |
TANAR_FULL_DUP );

            /* 启动自动协商 */
            NS_CSR_WRITE(TBICR, TBICR_MR_AN_ENABLE | TBICR_MR_RESTART_AN );
            NS_CSR_WRITE(TBICR, TBICR_MR_AN_ENABLE);
```

```
            pDrvCtrl->linkstate = LINK_AUTONEGOTIATE;

            pDrvCtrl->CFG_cache |= CFG_MODE_1000;
        }

        NS_CSR_WRITE( CFG, pDrvCtrl->CFG_cache);
        DRV_LOG (DRV_DEBUG_LOAD,"CFG: %08x\n", pDrvCtrl->CFG_cache,0,0,0,0,0);

    /* Note!  The DMA burst size interacts with packet transmission, such that the
largest packet that can be transmitted is 8192 - FLTH - burst size. If only the
transmit fifo was larger... */
        NS_CSR_WRITE(TXCFG, TXCFG_CSI | TXCFG_HBI | TXCFG_ATP | TXCFG_MXDMA1024
            | ((1600 / 32) * 0x100));

        /* Flush the interrupt holdoff timer */
        NS_CSR_WRITE(IHR,0x000);
        NS_CSR_WRITE(IHR,0x100);
        NS_CSR_WRITE(IHR,0x000);

        /* Set Rx to full duplex, don't accept runt, errored, long or length range
errored packets.  Set MXDMA to 0 => 1024 word burst
         */
        NS_CSR_WRITE(RXCFG, RXCFG_AEP | RXCFG_ARP | RXCFG_AIRL | RXCFG_RX_FD
            | RXCFG_STRIPCRC
            | RXCFG_ALP
            | (RXCFG_MXDMA0 * 0) | 0 );

        /* Disable priority queueing */
        NS_CSR_WRITE(PQCR, 0);

        /* Enable IP checksum validation and detetion of VLAN headers.
         * Note: do not set the reject options as at least the 0x102
         * revision of the chip does not properly accept IP fragments
         * at least for UDP.
         */
        NS_CSR_WRITE(VRCR, VRCR_IPEN | VRCR_VTDEN);

        /* Enable per-packet TCP/UDP/IP checksumming */
        NS_CSR_WRITE(VTCR, VTCR_PPCHK);

        /* Disable Pause frames */
        NS_CSR_WRITE( PCR, 0 );

        /* Disable Wake On Lan */
        NS_CSR_WRITE( WCSR, 0);

    if (using_dac) {
            printf ("%s: using 64 bit addressing.\n",
                pDrvCtrl->endObj.devObject.name);
        }
```

```
/* 禁止优先级队列 */
   NS_CSR_WRITE(PQCR, 0);

/* 接收描述子地址 */
   NS_CSR_WRITE(RXDP_HI,0) ;
   NS_CSR_WRITE(RXDP,/*NS_VIRT_TO_PCI*/(pDrvCtrl->rxAddr));


/* 发送描述子地址 */
   NS_CSR_WRITE(TXDP_HI,0);
   NS_CSR_WRITE(TXDP,/*NS_VIRT_TO_PCI*/(pDrvCtrl->txAddr));

/* 启动接收 */
 {
       DRV_LOG (DRV_DEBUG_LOAD,"starting receiver\n",0,0,0,0,0,0);

       NS_CSR_WRITE(CCSR,0x0001);
       NS_CSR_WRITE( RFCR,0 );
       NS_CSR_WRITE( RFCR,0x7fc00000);
       NS_CSR_WRITE( RFCR,0xffc00000);

       phy_intr(pDrvCtrl);

       pDrvCtrl->IMR_cache |= ISR_PHY;
       pDrvCtrl->IMR_cache |= ISR_RXRCMP;
       pDrvCtrl->IMR_cache |= ISR_TXRCMP;
       pDrvCtrl->IMR_cache |= ISR_RXERR;
       pDrvCtrl->IMR_cache |= ISR_RXOK;
       pDrvCtrl->IMR_cache |= ISR_TXOK;
       pDrvCtrl->IMR_cache |= ISR_TXERR;
       pDrvCtrl->IMR_cache |= ISR_RXORN;
       pDrvCtrl->IMR_cache |= ISR_RXSOVR;
       pDrvCtrl->IMR_cache |= ISR_RXDESC;
       pDrvCtrl->IMR_cache |= ISR_RXIDLE;
       pDrvCtrl->IMR_cache |= ISR_TXDESC;
       pDrvCtrl->IMR_cache |= ISR_TXIDLE;

       NS_CSR_WRITE(IMR,pDrvCtrl->IMR_cache);/* write interrupter mask reg */

       NS_CSR_WRITE(IER,1); /* enable interrupter*/

       kick_rx(pDrvCtrl);

   }

/* 置 txBlocked 标志为 FALSE */
   pDrvCtrl->txBlocked = FALSE;

   /* mark the interface -- up */
```

```
    END_FLAGS_SET (&pDrvCtrl->endObj, (IFF_UP | IFF_RUNNING));

    /* 允许该网卡中断 */

    SYS_INT_ENABLE (pDrvCtrl);

    return (OK);
    }
```

4.3.2.6 设备停止函数

设备停止函数也是 END 驱动要求的实现的函数之一，一般情况下不会调用。主要的工作是设置接口为 DOWN 状态，并复位硬件设备：禁止设备中断，停止接收和发送。所有 END 驱动的停止函数基本一致，不同的是复位寄存器操作不同。

```
LOCAL STATUS ns83820Stop
    (
    DRV_CTRL *  pDrvCtrl
    )
    {
    int     retVal=OK;

    DRV_LOG (DRV_DEBUG_LOAD, "Stop\n", 0, 0, 0, 0, 0, 0);

    /* 设置接口为 DOWN 状态 */

    END_FLAGS_CLR (&pDrvCtrl->endObj, IFF_UP | IFF_RUNNING);

    /* 复位设备 */

    ns83820ChipReset (pDrvCtrl,CR_RST);

    /* 禁止网卡中断 */
    SYS_INT_DISABLE (pDrvCtrl);

    /* 断开中断连接*/
    SYS_INT_DISCONNECT (pDrvCtrl, ns83820Int, (int)pDrvCtrl, &retVal);

    return (retVal);
    }
```

4.3.2.7 接口操作函数

接口操作函数 ns83820Ioctl 实现网络接口控制功能，处理 EIOCSADDR、EIOCGADDR、EIOCSFLAGS、EIOCGFLAGS、EIOCMULTIADD、EIOCMULTIDEL、EIOCMULTIGET、EIOCPOLLSTART、EIOCPOLLSTOP、EIOCGMIB2 等命令。该函数基本照搬 21140 相关代码。实际上没有实现组播和查询收发模式。

这些命令分别对应：

- EIOCSADDR：设置网卡硬件（MAC）地址。

- EIOCGADDR：得到网卡硬件（MAC）地址。
- EIOCSFLAGS：设置设备的 flag 标示。
- EIOCOFLAGS：得到设备的 flag 标示。
- EIOCMULTIADD：增加组播地址。
- EIOCMULTIDEL：删除组播地址。
- EIOCMULTIGET：得到组播地址。
- EIOCPOLLSTART：启动查询收发模式。
- EIOCPOLLSTOP：停止查询收发模式。
- EIOCGMIB2：MibII 功能。

```
LOCAL int ns83820Ioctl
    (
    DRV_CTRL *  pDrvCtrl,
    int         cmd,
    caddr_t     data
    )
    {
    int         error=0;
    long        value;
    int         savedFlags;
    END_OBJ *   pEndObj=&pDrvCtrl->endObj;

    switch (cmd)
        {
        case EIOCSADDR:
            if (data == NULL)
                error = EINVAL;
            else
                {
                                    /* Copy and install the new address */
                memcpy ((char *)END_HADDR(pEndObj), (char *)data,
                      END_HADDR_LEN(pEndObj));
                ns83820IASetup (pDrvCtrl);
                }
            break;

        case EIOCGADDR:             /* HELP: move to mux */
            if (data == NULL)
                error = EINVAL;
            else
                memcpy ((char *)data, (char *)END_HADDR(pEndObj),
                      END_HADDR_LEN(pEndObj));
            break;

        case EIOCSFLAGS:
            value = (long) data;
            if (value < 0)
                {
```

```
                value = -value;
                value--;
                END_FLAGS_CLR (pEndObj, value);
                }
            else
                END_FLAGS_SET (pEndObj, value);

            /* handle IIF_PROMISC and IFF_ALLMULTI */

            savedFlags = DRV_FLAGS_GET();
            if (END_FLAGS_ISSET (pEndObj, IFF_PROMISC))
                DRV_FLAGS_SET (NS_PROMISC);
            else
                DRV_FLAGS_CLR (NS_PROMISC);

            if (END_FLAGS_GET (pEndObj) & (IFF_ALLMULTI | IFF_MULTICAST))
                DRV_FLAGS_SET (NS_MCAST);
            else
                DRV_FLAGS_CLR (NS_MCAST);

            if ((DRV_FLAGS_GET() != savedFlags) &&(END_FLAGS_GET(pEndObj)
& IFF_UP))
                ns83820ModeSet (pDrvCtrl);
            break;

        case EIOCGFLAGS:                    /* move to mux */
            if (data == NULL)
                error = EINVAL;
            else
                *(long *)data = END_FLAGS_GET(pEndObj);
            break;

        case EIOCMULTIADD:                  /* move to mux */
            error = ns83820MCastAddrAdd (pDrvCtrl, (char *)data);
            break;

        case EIOCMULTIDEL:                  /* move to mux */
            error = ns83820MCastAddrDel (pDrvCtrl, (char *)data);
            break;

        case EIOCMULTIGET:                  /* move to mux */
            error = ns83820MCastAddrGet (pDrvCtrl, (MULTI_TABLE *)data);
            break;

        case EIOCPOLLSTART:                 /* move to mux */
            ns83820PollStart (pDrvCtrl);
            break;

        case EIOCPOLLSTOP:                  /* move to mux */
            ns83820PollStop (pDrvCtrl);
```

```
            break;

        case EIOCGMIB2:                    /* move to mux */
            if (data == NULL)
                error=EINVAL;
            else
                memcpy((char *)data, (char *)&pEndObj->mib2Tbl,
                       sizeof(pEndObj->mib2Tbl));
            break;

        default:
            error = EINVAL;
        }

    return (error);
    }
```

4.3.2.8　组播操作函数

MUX 要求 END 驱动实现一套组播函数，主要有：组播地址增加、组播地址删除、组播地址得到等。函数原型如下：

```
LOCAL STATUS ns83820MCastAddrAdd
    (
    DRV_CTRL *   pDrvCtrl,
    char *       pAddr
    )
    {
    /* 与设备组播地址增加操作相关代码 */
    return  操作状态：OK 或 ERROR;
    }

LOCAL STATUS ns83820MCastAddrDel
    (
    DRV_CTRL *   pDrvCtrl,
    char *       pAddr
    )
    {
    /* 与设备组播地址删除操作相关代码 */
    return  操作状态：OK 或 ERROR;
    }

LOCAL STATUS ns83820MCastAddrGet
    (
    DRV_CTRL *       pDrvCtrl,
    MULTI_TABLE *    pTable
    )
    {
    /* 获得设备组播地址表 */
    return  操作状态：OK 或 ERROR;
}
```

4.3.2.9 报文发送函数

报文发送函数 ns83820Send，完成一帧以太网报文传输功能。发送报文传递给协议层，协议层完成协议报文封装操作，形成一帧以太网报文，以 M_BLK 结构指针的形式作为参数，调用 ns83820Send，ns83820Send 从 M_BLK 结构中获得以太网报文数据，填充到一个空闲的发送描述子中，设置该描述子的相应状态，然后启动 ns83820 发送状态机，ns83820 硬件完成后续的发送工作，发送完成将产生中断，发送中断处理将在中断处理函数中介绍。

在 VxWorks 协议栈中，muxSend()调用该程序。发送是主动的，当上层软件需要发送数据时，将逐层调用底层发送函数，完成报文发送。

```
LOCAL STATUS ns83820Send
    (
    DRV_CTRL *   pDrvCtrl,
    M_BLK *      pMblk    /* 包含将要发送的数据 */
    )
    {
    NS_DESC *    pTxD;
    char *       pBuf;
    int          len;
    int          s;

    /* 首先判断发送状态机是否已被阻塞，当发送描述子或系统存储器不可用时，置阻塞状态，发送中断处理程序解除阻塞。阻塞时返回 END_ERR_BLOCK */
        if (pDrvCtrl->txBlocked){
            return (END_ERR_BLOCK); /* transmitter not ready */
        }

        /* 互斥操作，获取信号量，不允许重入。*/

        END_TX_SEM_TAKE (&pDrvCtrl->endObj, WAIT_FOREVER);

        /* 调用 ns83820TxDGet 获得下一个可用发送描述子 */
        pTxD = ns83820TxDGet (pDrvCtrl);
        调用 NET_BUF_ALLOC()申请发送数据缓冲区。
        pBuf = NET_BUF_ALLOC();

        /* 判断发送描述子和发送数据缓冲区是否有效。*/
        if ((pTxD == NULL) || (pBuf == NULL))
            {
        /* 无效：置错误状态，并释放互斥信号量。 */
            END_ERR_ADD (&pDrvCtrl->endObj, MIB2_OUT_ERRS, +1);
            END_TX_SEM_GIVE (&pDrvCtrl->endObj);

            if (pBuf)
                NET_BUF_FREE (pBuf);

            if (!pDrvCtrl->txCleaning && !pDrvCtrl->txBlocked)
                ns83820TxRingClean (pDrvCtrl);
```

置发送阻塞标示。

```
    s = intLock();
    pDrvCtrl->txBlocked = TRUE;     /* transmitter not ready */
    intUnlock(s);
```

返回 END_ERR_BLOCK。

```
    return (END_ERR_BLOCK);         /* just return without freeing mBlk chain */
    }
```

缓冲区申请操作有效，将 pMblk 中的数据拷贝到 pBuf 中。拷贝操作可直接调用 netMblkToBufCopy 完成。

```
    len = netMblkToBufCopy (pMblk, pBuf, NULL);
```

释放该 pMblk。

```
       NET_MBLK_CHAIN_FREE (pMblk);

    /* 将 pBuf 赋值给发送描述符 pTxD 的 buf 域，将长度 len 填充到 cmdsts 域，置 cmdsts 的 own
位 */

       pTxD->buf = PCISWAP (/*NS_VIRT_TO_PCI*/(ULONG) (pBuf));
       pTxD->cmdsts = PCISWAP (CMDSTS_OWN | CMDSTS_INTR | len);

       /* Flush the write pipe */

       CACHE_PIPE_FLUSH();

       /* 保存该 pBuf 指针，pBuf 将在发送中断处理函数中释放。*/
       pDrvCtrl->freeBuf[pDrvCtrl->txIndex].pClBuf = pBuf;

       /* 调整，待释放缓冲区链索引 */

       pDrvCtrl->txIndex = (pDrvCtrl->txIndex + 1) % pDrvCtrl->numTds;

        /* 置命令寄存器的发送使能位为 1，启动发送 */
       NS_CSR_WRITE(CR, CR_TXE);
```

释放互斥信号量。

```
    END_TX_SEM_GIVE (&pDrvCtrl->endObj);
```

修改统计数据。

```
    END_ERR_ADD (&pDrvCtrl->endObj, MIB2_OUT_UCAST, +1);
```

正常返回。

```
    return (OK);
    }
```

函数 ns83820TxDGet 用来得到下一个可用的发送描述子，返回给调用者。

```
    LOCAL NS_DESC * ns83820TxDGet
       (
       DRV_CTRL *  pDrvCtrl
       )
```

```
    {
```

由发送描述子索引，从描述子链表中定位下一个可用的发送描述子。

```
NS_DESC *   pTxD = pDrvCtrl->txRing + pDrvCtrl->txIndex;

NS_CACHE_INVALIDATE (pTxD, TD_SIZ);
```

检查该描述子是否为主机所有或是否越界。

```
if ((pTxD->cmdsts & PCISWAP(CMDSTS_OWN)) ||
    (((pDrvCtrl->txIndex + 1) % pDrvCtrl->numTds) == pDrvCtrl->txDiIndex)){
return (NULL);
}

    return (pTxD);
    }
```

函数 ns83820TxRingClean 由中断处理程序调用，用来处理刚刚完成发送操作的发送描述子，释放发送缓冲区，将发送描述子所有权转交给主机。

```
LOCAL void ns83820TxRingClean
    (
    DRV_CTRL *  pDrvCtrl
    )
    {
    NS_DESC *   pTxD;
```

置 pDrvCtrl->txCleaning 为 TRUE，避免中断处理程序重复调用。

```
pDrvCtrl->txCleaning = TRUE;

while (pDrvCtrl->txDiIndex != pDrvCtrl->txIndex)
    {
    pTxD = pDrvCtrl->txRing + pDrvCtrl->txDiIndex;
    NS_CACHE_INVALIDATE (pTxD, TD_SIZ);
```

检查描述子的所有位，如为芯片所有，这表示该描述子尚未完成发送操作，则中止操作。

```
if (pTxD->cmdsts & PCISWAP(CMDSTS_OWN))
    break;
```

释放发送数据缓冲，并调整待释放缓冲区数组下标指针，见前面 ns83820Send 函数。

```
if (pDrvCtrl->freeBuf[pDrvCtrl->txDiIndex].pClBuf != NULL)
    {
    NET_BUF_FREE(pDrvCtrl->freeBuf[pDrvCtrl->txDiIndex].pClBuf);
    pDrvCtrl->freeBuf[pDrvCtrl->txDiIndex].pClBuf = NULL;
    }
pDrvCtrl->txDiIndex =  (pDrvCtrl->txDiIndex + 1) % pDrvCtrl->numTds;
```

置该描述符的 cmdsts 为驱动所有，请求中断。

```
    pTxD->cmdsts = PCISWAP (CMDSTS_INTR);
}
```

置 pDrvCtrl->txCleaning 为 FALSE，允许中断处理程序再次调用。

```
pDrvCtrl->txCleaning = FALSE;

return;
}
```

4.3.2.10　接收报文处理程序

与发送报文处理函数相对立的是接收报文处理函数 ns83820RxIntHandle，接收与发送不同，接收是被动的，在中断方式下接收过程不直接由上层协议调用，而是由接收中断触发，当接收中断产生时，中断处理函数将调用该函数，完成接收处理。接收过程如下：

调用 ns83820RxDGet 遍历接收描述符链表，根据接收描述子的 cmdsts 的 OWN 位，判断是否有新接收的报文，如果是，则调用 ns83820Recv 处理该报文。

为缩短中断处理时间，中断处理程序将接收处理过程放在 tNetTask 任务中运行。

```
LOCAL void ns83820RxIntHandle
    (
    DRV_CTRL *   pDrvCtrl
    )
    {
    NS_DESC *    pRxD;
```

置 pDrvCtrl->rxHandling 为 TRUE，避免中断处理程序重复调用（向 netTask 任务中添加）本函数。

```
pDrvCtrl->rxHandling = TRUE;
```

遍历调用 ns83820RxDGet 遍历接收描述符链表。

```
while ((pRxD = ns83820RxDGet (pDrvCtrl)))
```

对有效的接收描述子，调用 ns83820Recv 处理该接收描述子。

```
ns83820Recv (pDrvCtrl, pRxD);
```

置 pDrvCtrl->rxHandling 为 FALSE。

```
pDrvCtrl->rxHandling = FALSE;
}
```

函数 ns83820RxDGet 用来获得一个有效的接收描述子，并返回给调用程序，类似于 ns83820TxDGet。

```
LOCAL NS_DESC * ns83820RxDGet
    (
    DRV_CTRL *   pDrvCtrl
    )
    {
```

由接收描述子索引，从描述子链表中定位下一个可用的接收描述子。

```
NS_DESC *    pRxD = pDrvCtrl->rxRing + pDrvCtrl->rxIndex;

NS_CACHE_INVALIDATE (pRxD, RD_SIZ);
```

检查描述子的有效性，是否为主机所有。

```
if (!(pRxD->cmdsts & PCISWAP (CMDSTS_OWN))){
    return (NULL);
}
```

返回该描述子指针。

```
return (pRxD);
}
```

ns83820Recv 完成一帧报文的接收处理，将报文数据组织成 M_BLK，调用宏 END_RCV_RTN_CALL 提交给上层协议。

```
LOCAL STATUS ns83820Recv
    (
    DRV_CTRL *   pDrvCtrl,
    NS_DESC *    pRxD
    )
    {
    END_OBJ *    pEndObj = &pDrvCtrl->endObj;
    M_BLK_ID     pMblk;      /* MBLK to send upstream */
    CL_BLK_ID    pClBlk;     /* pointer to clBlk */
    char *       pBuf;       /* A replacement buffer for the current RxD */
    char *       pData;      /* Data pointer for the current RxD */
    char *       pTmp;
    int          len;        /* Len of the current data */

    /* 检查接收描述子的出错状态 */

    if (pRxD->cmdsts & PCISWAP(CMDSTS_ERR))
        {
     /* 出错：更新出错统计数据，转出错处理。*/
        END_ERR_ADD (pEndObj, MIB2_IN_ERRS, +1);
        goto cleanRxD;
        }
```

分配一个 MBLK 结构。

```
pMblk = NET_MBLK_ALLOC();
pBuf  = NET_BUF_ALLOC();
pClBlk = NET_CL_BLK_ALLOC();

/* 判断缓冲分配是否成功 */
if ((pMblk == NULL) || (pBuf == NULL) || (pClBlk == NULL))
    {
```

出错处理：更新统计数据，释放分配成功的缓冲内存，转出错处理。

```
        END_ERR_ADD (pEndObj, MIB2_IN_ERRS, +1);
        if (pMblk)
            NET_MBLK_FREE (pMblk);
        if (pBuf)
            NET_BUF_FREE (pBuf);
```

```
        if (pClBlk)
            NET_CL_BLK_FREE (pClBlk);
        goto cleanRxD;
        }
```

从当前接收描述子中获得报文数据及长度

```
    len   = (PCISWAP (pRxD->cmdsts&0xffff));     /* - ETH_CRC_LEN; */
    pData = (char *) (PCISWAP (pRxD->buf));
```

下面的代码部分借鉴 21140，对于 83820 驱动 pDrvCtrl->offset 为 0，可以不考虑。

```
    if (pDrvCtrl->offset != 0)
    {
    /* exchange buffer addresses */

    pTmp = pData;
    pData = pBuf;
    pBuf = pTmp;

    /* copy data in new unaligned buffer */

    pData += pDrvCtrl->offset;
    memcpy(pData, pBuf, len);
    }
```

将接收数据拷贝到 pBuf 中，这里没有直接利用零拷贝技术，是因为 ns83820 接收 DMA 要求接收缓冲是低 8 位对齐，NET_BUF_ALLOC 分配得到的 pBuf 不能保证是低 8 位对齐，也就不能作为接收缓冲区。大多数 END 驱动可以采用零拷贝技术，直接将接收描述子的数据缓冲指针提交给上层协议，而不需要缓冲间的数据拷贝。零拷贝技术示意图如图 4-17 和图 4-18 所示：

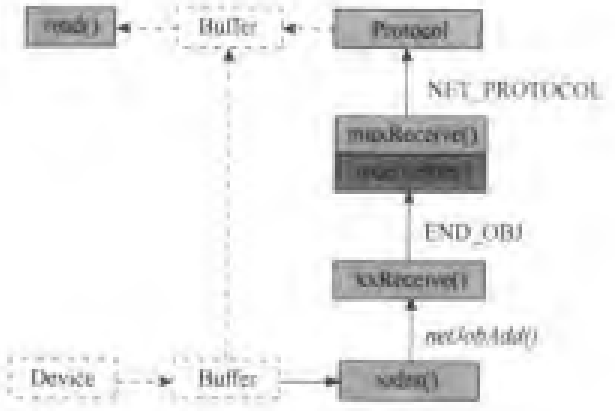


图 4-17　零拷贝技术接收过程示意图

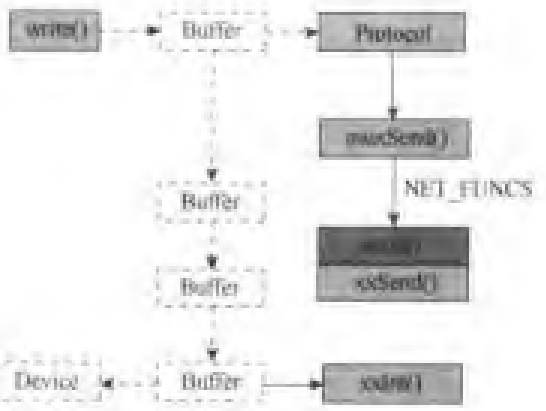


图 4-18　零拷贝技术发送过程示意图

```
    pData -= pDrvCtrl->offset;
    memcpy(pBuf,pData,len);
```

将 pBuf 赋值给 pMblk 中的数据指针，这一操作可以直接调用宏 NET_CL_BLK_JOIN 和 NET_MBLK_CL_JOIN 完成。

```
    NET_CL_BLK_JOIN (pClBlk, pBuf /*pData*/, NS_BUFSIZ);
```

```
    /* Associate the data pointer with the MBLK */

    NET_MBLK_CL_JOIN (pMblk, pClBlk);

    pMblk->mBlkHdr.mData     += pDrvCtrl->offset;
    pMblk->mBlkHdr.mFlags    |= M_PKTHDR;     /* set the packet header */
    pMblk->mBlkHdr.mLen      = len;           /* set the data len */
    pMblk->mBlkPktHdr.len    = len;           /* set the total len */
```

cache 一致性操作。

```
    NS_CACHE_INVALIDATE (pBuf, len);
```

正如前面提到的，本驱动中没有使用零拷贝技术，接收描述子的数据缓冲也就不需要更新。采用零拷贝技术，接收描述子的数据缓冲指针直接提交给上层协议，上层协议完成数据处理后将释放该指针，因此，发送描述子的数据缓冲需要重新赋值，由下述操作完成。

```
/* pBuf = (char *)(((int)pBuf+ 0x07)&~0x07);
   pRxD->buf = (ULONG)PCISWAP ((pBuf));
*/
    /* 将该描述子所有权交给芯片所有，即清除 cmdsts 的 OWN 位。*/

    pRxD->cmdsts = PCISWAP ( REAL_RX_BUF_SIZE   CMDSTS_INTR);

    /* 更新统计数据 */

    END_ERR_ADD (pEndObj, MIB2_IN_UCAST, +1);

    /* 修改接收描述子链表索引值 */

    pDrvCtrl->rxIndex = (pDrvCtrl->rxIndex + 1) % pDrvCtrl->numRds;

    /* 清写通道 */

    CACHE_PIPE_FLUSH();
```

调用宏将该帧增数据提交给上层协议。

```
    END_RCV_RTN_CALL (pEndObj, pMblk);

    return OK;
```

以下是出错时的错误处理。

```
cleanRxD:

    /* 将该描述子返还给芯片*/

    pRxD->cmdsts = PCISWAP ( REAL_RX_BUF_SIZE | CMDSTS_INTR);

    /* 更新统计数据 */
```

```
DRV_LOG (DRV_DEBUG_RX, "+ ", 0, 0, 0, 0, 0, 0);
END_ERR_ADD (pEndObj, MIB2_IN_UCAST, +1);

/* Flush the write pipe */

CACHE_PIPE_FLUSH();

/* 修改接收描述子链表索引值 */

pDrvCtrl->rxIndex = (pDrvCtrl->rxIndex + 1) % pDrvCtrl->numRds;
return (OK);
}
```

4.3.2.11　中断处理函数

中断处理函数 ns83820Int 处理设备中断。过程如下：读中断状态寄存器，清中断事件，根据中断状态，调用相应的中断处理程序，如接收中断处理程序、发送中断处理程序等等。

```
LOCAL void ns83820Int
    (
    DRV_CTRL *  pDrvCtrl
    )
    {
    ULONG       isr;
```

读中断状态寄存器。

```
    isr = NS_CSR_READ(ISR);

    if(isr!=0x18000)
        if ((ISR_RXIDLE | ISR_RXOK) & isr) {
        pDrvCtrl->rxIdle = 1;
        }
```

判断是否为接收描述符中断，如是，则将接收中断处理程序添加到 tNetTask 任务的工作队列中。tNetTask 是 VxWorks 网络协议栈处理任务，为避免长时间处理器处于封中状态，提高系统中断处理能力，中断处理程序在设计时应该把处理时间较长的事务启动一个延迟任务处理（类似于 Windows 中的延期调用），对于网络驱动，如果收发处理时间过长，应将处理函数调用 netJobAdd 添加到 tNetTask 任务的工作队列中，netJobAdd 调用将发送信号量激活该任务，在系统调度 tNetTask 任务时处理。tNetTask 任务被调度时，将遍历工作队列，处理所有排队任务，直至队列为空。在 shell 下执行 i 命令可看到 tNetTask 的状态，如图 4-19 所示。

```
-> i
  NAME        ENTRY       TID    PRI   STATUS      PC       SP     ERRNO  DELAY
---------- ------------ -------- --- ---------- -------- -------- ------- -----
tExcTask   excTask      f4f63f0    0 PEND       6b4f23   f4f6338       0     0
tLogTask   logTask      f4f3a50    0 PEND       6b4f23   f4f3988       0     0
tShell     shell        f4262a8    1 PEND       610f7b   f425f0c       0     0
tWdbTask   wdbTask      f4274dc    3 READY      610f7b   f4273d0       0     0
tNetTask   netTask      f493144   50 READY      610ee1   f492f88       0     0
tDcacheUpd dcacheUpd    f4d225c  250 DELAY      6a9d11   f4d21e4       0     4
value = 0 = 0x0
->
```

图 4-19　shell 下察看 tNetTask 任务状态

```
    if ((ISR_RXDESC ) & isr) {
        NS_CSR_WRITE(IMR, pDrvCtrl->IMR_cache & ~(ISR_RXDESC | ISR_RXOK));
        if(!pDrvCtrl->rxHandling){
            pDrvCtrl->rxHandling=TRUE;
            netJobAdd ((FUNCPTR)ns83820RxIntHandle, (int)pDrvCtrl, 0, 0, 0, 0);
        }
```

清中断状态。

```
        NS_CSR_WRITE(IMR,pDrvCtrl->IMR_cache | ISR_RXDESC|ISR_RXOK);
    }

    if (ISR_RXRCMP & isr)
        NS_CSR_WRITE(CR, CR_RXE);
```

判断是否为发送描述符中断，如是，则处理发送中断。

```
    if ((ISR_TXDESC | ISR_TXIDLE | ISR_TXOK) & isr) {
        ULONG txdp,txdp1;
        int tiIndex;
        txdp=txdp1 = NS_CSR_READ( TXDP);
        txdp -= /*NS_VIRT_TO_PCI*/(ULONG)(pDrvCtrl->txRing);
        tiIndex = txdp / 16;
        if (tiIndex >= pDrvCtrl->numTds) {
            tiIndex = 0;
        }
        if (tiIndex != pDrvCtrl->txIndex)
            NS_CSR_WRITE(CR,CR_TXE);
        else
        {
        if(!pDrvCtrl->txCleaning){
            pDrvCtrl->txCleaning=TRUE;
            ns83820TxRingClean(pDrvCtrl);
        }

        pDrvCtrl->txIdle = 1;
```

如果发送状态为阻塞，则置 pDrvCtrl->txBlocked 为 FALSE，并调用 muxTxRestart 重新启动发送。

```
        if (pDrvCtrl->txBlocked)
        {
        pDrvCtrl->txBlocked = FALSE;
        netJobAdd ((FUNCPTR)muxTxRestart, (int)&pDrvCtrl->endObj, 0, 0, 0, 0);
        }

        }
    }
```

处理 mib 中断。

```
    if (ISR_MIB & isr){
        logMsg("mib\n",0,0,0,0,0,0);
/*      ns83820_mib_isr(dev);
```

```
*/
    }
```

处理物理层中断。

```
if (ISR_PHY & isr){
        logMsg("phy\n",0,0,0,0,0,0);
/*      phy_intr(pDrvCtrl);
*/
    }
    return;
    }
```

4.3.2.12　查询方式函数

查询方式需要提供如下几个函数：这几个函数本例中没有实现。

（1）启动查询工作方式：ns83820PollStart。

```
LOCAL STATUS ns83820PollStart
    (
    DRV_CTRL *   pDrvCtrl
    )
    {
    return (OK);
    }
```

（2）停止查询工作方式：ns83820PollStop。

```
LOCAL STATUS ns83820PollStop
    (
    DRV_CTRL *   pDrvCtrl
    )
    {
    return (OK);
    }
```

（3）查询方式发送：ns83820PollSend。

```
LOCAL STATUS ns83820PollSend
    (
    DRV_CTRL *   pDrvCtrl,
    M_BLK *      pMblk
    )
    {
    return (OK);
    }
```

（4）查询方式接收：ns83820PollReceive。

```
LOCAL STATUS ns83820PollReceive
    (
    DRV_CTRL *   pDrvCtrl,
    M_BLK  *     pMblk
    )
    {
```

```
    return OK;
    }
```

4.3.2.13 其他函数

以下几个函数由 Linux 83820 驱动移植而来。

（1）MAC 地址获取：ns83820_getmac。

```
LOCAL STATUS ns83820_getmac(
    DRV_CTRL *   pDrvCtrl,
    UCHAR *      mac
    )
{
    unsigned i;
    NS_CSR_WRITE(PTSCR,PTSCR_EELOAD_EN);
    taskDelay(sysClkRateGet()/10);

    for (i=0; i<3; i++) {
        ULONG data;
        /* Read from the perfect match memory: this is loaded by
         * the chip from the EEPROM via the EELOAD self test.
         */
        NS_CSR_WRITE(RFCR,i*2);
        taskDelay(1);
        data = NS_CSR_READ(RFDR);
        *mac++ = data>>8;
        *mac++ = data;
    }
    return OK;
}
```

（2）物理层中断处理函数：phy_intr。

```
LOCAL void phy_intr( DRV_CTRL *pDrvCtrl )
{
    ULONG cfg, new_cfg;
    ULONG tbisr, tanar, tanlpar;
    int  speed,fullduplex, newlinkstate;

    cfg = NS_CSR_READ(CFG) ^ SPDSTS_POLARITY;

    if (pDrvCtrl->CFG_cache & CFG_TBI_EN) {
        /* we have an optical transceiver */
        tbisr = NS_CSR_READ(TBISR);
        tanar = NS_CSR_READ(TANAR);
        tanlpar = NS_CSR_READ(TANLPAR);

        if ( (fullduplex = (tanlpar & TANAR_FULL_DUP)
             && (tanar & TANAR_FULL_DUP)) ) {

            /* both of us are full duplex */
            NS_CSR_WRITE(TXCFG, NS_CSR_READ(TXCFG)
```

```
                    | TXCFG_CSI | TXCFG_HBI | TXCFG_ATP);
                NS_CSR_WRITE( RXCFG, NS_CSR_READ( RXCFG) | RXCFG_RX_FD);
                /* Light up full duplex LED */
                NS_CSR_WRITE(GPIOR, NS_CSR_READ(GPIOR) | GPIOR_GP1_OUT );

            } else if(((tanlpar & TANAR_HALF_DUP)
                  && (tanar & TANAR_HALF_DUP))
                || ((tanlpar & TANAR_FULL_DUP)
                  && (tanar & TANAR_HALF_DUP))
                || ((tanlpar & TANAR_HALF_DUP)
                  && (tanar & TANAR_FULL_DUP))) {

                /* one or both of us are half duplex */
                NS_CSR_WRITE( TXCFG, (NS_CSR_READ(TXCFG) & ~(TXCFG_CSI | TXCFG_HBI))
| TXCFG_ATP);
                NS_CSR_WRITE( RXCFG, NS_CSR_READ(RXCFG) & ~RXCFG_RX_FD);
                /* Turn off full duplex LED */
                NS_CSR_WRITE(GPIOR, NS_CSR_READ(GPIOR) & ~GPIOR_GP1_OUT);
            }

            speed = 4; /* 1000F */

        } else {
            /* we have a copper transceiver */
            new_cfg = pDrvCtrl->CFG_cache & ~(CFG_SB | CFG_MODE_1000 | CFG_SPDSTS);

            if (cfg & CFG_SPDSTS1)
                new_cfg |= CFG_MODE_1000;
            else
                new_cfg &= ~CFG_MODE_1000;

            speed = ((cfg / CFG_SPDSTS0) & 3);
            fullduplex = (cfg & CFG_DUPSTS);

            if (fullduplex)
                new_cfg |= CFG_SB;

            if ((cfg & CFG_LNKSTS) &&
                ((new_cfg ^ pDrvCtrl->CFG_cache) & CFG_MODE_1000)) {
                NS_CSR_WRITE(CFG ,new_cfg);
                pDrvCtrl->CFG_cache = new_cfg;
            }

            pDrvCtrl->CFG_cache &= ~CFG_SPDSTS;
            pDrvCtrl->CFG_cache |= cfg & CFG_SPDSTS;
        }
        newlinkstate = (cfg & CFG_LNKSTS) ? LINK_UP : LINK_DOWN;

        pDrvCtrl->linkstate = newlinkstate;
    }
```

其他有关 MII 读写的函数也由 Linux 移植而来。

4.4 网络驱动程序调试

调试是驱动开发工作的一个重要组成部分。VxWorks 其他软件（含其他类型的驱动软件）都可以采用网络连接方式进行调试，网络驱动同样也可如此，只是需要用另一块不同类型的网卡。

网络驱动是协议栈的一部分，网络驱动的正确性必须要放到协议栈中验证。因此为调试网络驱动必须把它放到系统中，下一节将介绍如何将新的网络驱动添加到系统中，这一过程也较为繁琐，而且不利于调试。这里介绍方便调试的方法，对于理解 MUX 层如何加载 END 驱动也有帮助。

（1）新建一个可引导工程，并把 sysNs83820End.c、ns83820End.c 添加到工程中。

（2）配置 VxWorks，将其 IP_MAX_UNITS 参数改为 2，如图 4-20 所示。这样 VxWorks 协议栈支持两个 IP。

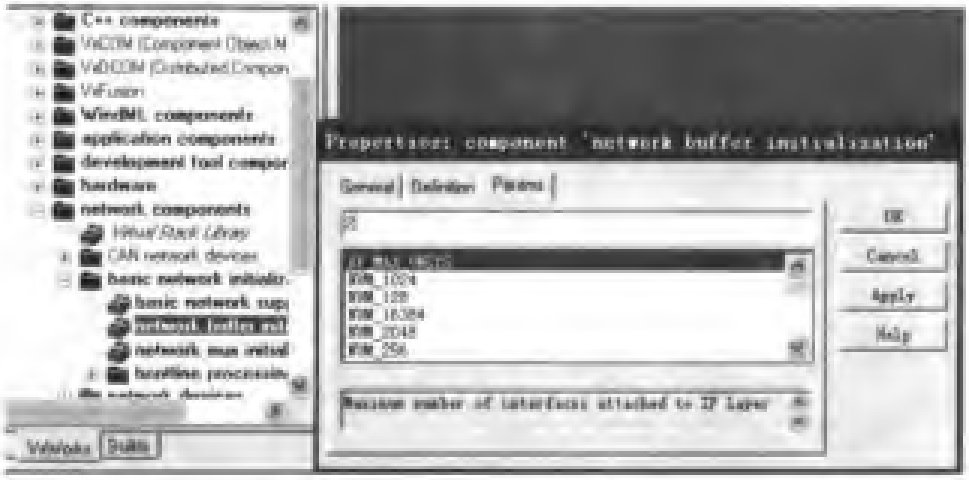


图 4-20 IP_MAX_UNITS 参数修改

（3）修改工程中的 sysLib.c 文件，找到函数 sysHwInit (void)，在该函数调用 sysSerialHwInit 之前，添加如下内容：sysNs83820PciInit();。部分代码如下：

```
    sysNs83820PciInit();

    /* initializes the serial devices */

    sysSerialHwInit ();         /* initialize serial data structure */
```

（4）将下面一段代码，添加到 ns83820end.c 文件的最后。

```
void*     pNs83820Cookie =NULL;
extern END_OBJ *  sysNs83820EndLoad();
STATUS nsEndDrv(
)
{

    /* 检查网卡驱动是否已加载 */
```

```
    if(!muxDevExists("ns",0)){
        /* 否则，调用 muxDevLoad 加载该驱动 */
        pNs83820Cookie = muxDevLoad(0, sysNs83820EndLoad,"", TRUE, NULL);

        if (pNs83820Cookie == NULL) {
            printf("muxDevLoad failed for device Ns83820!\n");
            return ERROR;
        }
    }

    if(ifunit("ns0")==NULL)
```

调用 muxDevStart 启动该 MUX 设备，将调用 ns83820Start，初始化硬件。

```
        if (muxDevStart(pNs83820Cookie) == ERROR){
          printf("muxDevStart failed for device ns0!\n");
            return ERROR;
        }
```

将 IP 协议搭接到该设备驱动上。

```
    ipAttach(0, "ns");
```

设置 IP 掩码。

```
    ifMaskSet("ns0", 0xffffff00);
```

设置 IP 地址。

```
    ifAddrSet("ns0", "172.16.21.233");
    return (OK);
    }
```

（5）build 该工程影像，并用另一块网卡作为引导设备下载该影像。

（6）在开发主机中连接该目标机，启动 debug，用 debug 的运行对话框，在 nsEndDrv 函数入口设置断点，执行 nsEndDrv，即可开始网络驱动的调试，如图 4-21 所示。

图 4-21　用 DEBUG 运行 nsEndDrv

注意，当调试网卡与引导网卡使用相同的中断向量时，不能调试中断处理程序。

在 muxDevLoad 语句未执行前，在 shell 中调用 muxShow，看到的只有调试使用的网卡 mux 设备，如图 4-22 所示。

```
-> muxShow
Current mode  NORMAL
Device  fea Unit  0
Description  Intel 82557 Ethernet Enhanced Network Driver
Protocol  Wind Debug Agent  Type  257   Recv 0x69b830   Shutdown 0x0
Protocol  IP 4 & ARP    Type  2054  Recv 0x636b00   Shutdown 0x6361f0
Protocol  IP 4 & TCP/IP Type  2048  Recv 0x636b00   Shutdown 0x636e10
value = 0 = 0x0
->
```

图 4-22　muxDevLoad 语句未执行前 muxShow 显示结果

调试过程屏幕截图如图 4-23 所示。

muxDevLoad 语句执行完成，在 shell 下执行 muxShow，将能看到该 mux 设备，如图 4-24

所示。

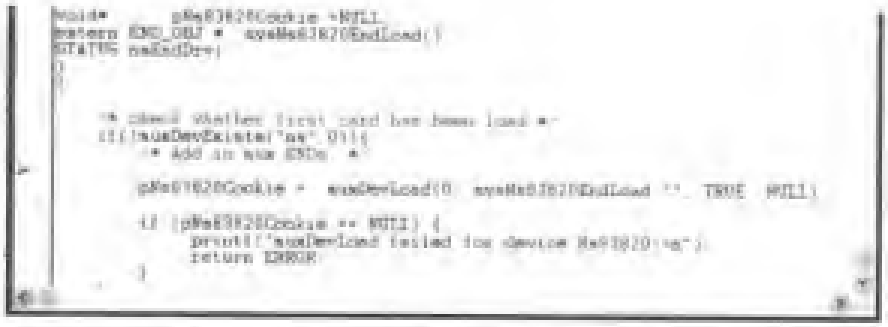


图 4-23　调试过程屏幕截图

```
-> muxShow
Current mode: NORMAL
Device: fei Unit: 0
Description: Intel 82557 Ethernet Enhanced Network Driver
Protocol: Wind Debug Agent  Type: 257    Recv 0x69bf30   Shutdown 0x0
Protocol: IP 4 & ARP    Type: 2054  Recv 0x636b00   Shutdown 0x636f50
Protocol: IP 4 & TCP/IP Type: 2048  Recv 0x636b00   Shutdown 0x636e10
Device: ns Unit: 0
Description: ns83820 10/100/1000m Enhanced Network Driver
value = 0 = 0x0
->
```

图 4-24　muxDevLoad 语句执行结束 muxShow 显示结果

muxDevStart 语句执行完成，在 shell 下执行 muxShow，该 mux 设备描述如图 4-25 所示。

```
-> muxShow
Current mode: NORMAL
Device: fei Unit: 0
Description: Intel 82557 Ethernet Enhanced Network Driver
Protocol: Wind Debug Agent  Type: 257    Recv 0x69bf30   Shutdown 0x0
Protocol: IP 4 & ARP    Type: 2054  Recv 0x636b00   Shutdown 0x636f50
Protocol: IP 4 & TCP/IP Type: 2048  Recv 0x636b00   Shutdown 0x636e10
Device: ns Unit: 0
Description: ns83820 10/100/1000m Enhanced Network Driver
Protocol: IP 4 & ARP    Type: 2054  Recv 0x636b00   Shutdown 0x636f50
Protocol: IP 4 & TCP/IP Type: 2048  Recv 0x636b00   Shutdown 0x636e10
value = 0 = 0x0
->
```

图 4-25　muxDevStart 语句执行结束 muxShow 显示结果

在 ifAddrSet，语句执行完成，在 shell 下执行 ifShow "ns"，该网络接口描述如图 4-26 所示。

```
-> ifShow "ns0"
ns0 network interface not found
value = -1 = 0xffffffff
-> ifShow "ns"
ns (unit number 0):
     Flags: (0x8063) UP BROADCAST RUNNING ARP MULTICAST
     Type: ETHERNET_CSMACD
     Internet address: 172.16.21.233
     Broadcast address: 172.16.21.255
     Netmask 0xffff0000 Subnetmask 0xffffff00
     Ethernet address is 00:08:00:17:06:00
     Metric is 0
     Maximum Transfer Unit size is 1500
     0 packets received; 1 packets sent
     0 multicast packets received
     0 multicast packets sent
     0 input errors; 0 output errors
     0 collisions; 0 dropped
value = 0 = 0x0
->
```

图 4-26　ifShow "ns" 显示结果

如果程序正确，在开发主机可以 ping 通该 IP 地址。可以用 ping -l 选项验证长报文的通过能力。如图 4-27 所示（在 DOS 命令行下，执行 ping 172.16.21.233 -l -l 60000）。

图 4-27　用 ping 验证驱动正确性

多开几个 ping 窗口，对驱动长时间考核，验证缓冲区申请与释放的正确性，不断在 shell 下调用 memShow 观察系统内存资源的使用情况，判断是否存在内存泄露。

4.5　将新 END 驱动添加到 VxWorks 网络体系中

4.5.1　VxWorks5.5 版本

在 VxWorks5.5 版本中，需要修改以下文件：

①config.h，②configNet.h，③sysNet.h。

具体过程如下。

4.5.1.1　config.h 文件修改

找到

```
#undef  INCLUDE_DLTRA_END              /* (END) SMC Elite16 Ultra interface */
```

在其后添加：

```
#define  INCLUDE_NS83820_END           /*  (END)  NS83820 PCI interface */
```

找到

```
#if defined (INCLUDE_LN_97X_END)    || defined (INCLUDE_EL_3C90X_END) || \
    defined (INCLUDE_FEI_END)       || defined (INCLUDE_DEC21X40_END) || \
    defined (INCLUDE_GEI8254X_END)  || defined (INCLUDE_AIC_7880)     || \
    defined (INCLUDE_WINDML)        || defined (INCLUDE_USB)
```

修改为：

```
#if defined (INCLUDE_LN_97X_END)    || defined (INCLUDE_EL_3C90X_END) || \
    defined (INCLUDE_FEI_END)       || defined (INCLUDE_DEC21X40_END) || \
    defined (INCLUDE_GEI8254X_END)  || defined (INCLUDE_AIC_7880)     || \
    defined (INCLUDE_WINDML)        || defined (INCLUDE_USB) || \
    defined (INCLUDE_ NS83820_END)
```

4.5.1.2　configNet.h 文件修改

找到

```
#ifdef INCLUDE_ULTRA_END
```

在其前添加如下代码：

```
/* ns838203 END driver defines */

#ifdef INCLUDE_83820_END

#define END_NS83820_LOAD_FUNC   sysNs83820EndLoad
#define END_ NS83820_BUFF_LOAN   TRUE
#define END_ NS83820_LOAD_STRING ""

IMPORT END_OBJ * END_ NS83820_LOAD_FUNC (char *, void *);

#endif /* INCLUDE_ NS83820_END */
```

找到

```
END_TBL_ENTRY endDevTbl [] =
    {
```

增加以下代码：

```
#ifdef INCLUDE_ NS83820_END
    {0, NS83820_LOAD_FUNC, NS83820_LOAD_STR, NS83820_BUFF_LOAN,
    NULL, FALSE},
#endif /* INCLUDE_ NS83820_END */
```

4.5.1.3 sysNet.c 文件修改

找到

```
#   include "sysUltraEnd.c"          /* ultraEnd support routines */
```

增加以下代码：

```
#   include "sysNs83820End.c"       /* ns83820End support routines */
```

找到

```
LOCAL VEND_ID_DESC vendorIdEnet [] =
    {
```

在其后增加：

```
    #if defined(INCLUDE_NS83820_END)

    { PCI_VENDOR_ID_NS,      sysNs83820PciInit },
    #endif /* INCLUDE_ NS83820_END */
```

4.5.2 VxWorks5.4 版本

在 VxWorks5.4 版本中，需要修改的文件有所不同，其中 config.h 和 configNet.h 修改基本一致，5.4 中没有 sysNet.c，PCI 的初始化放在 sysLib.c 中，修改如下：

找到

```
#ifdef INCLUDE_LN_97X_END
IMPORT STATUS sysLan97xPciInit (void);
#endif /* INCLUDE_LN_97X_END */
```

在其后添加：

```
/* include dec21x4xEnd driver support routines */
#ifdef INCLUDE_NS83820_END
IMPORT STATUS sysNs83820PciInit ();
#endif /* INCLUDE_NS83820_END */
```

找到

```
#ifdef INCLUDE_LN_97X_END
#   include "sysLn97xEnd.c"
#endif /* INCLUDE_LN_97X_END */
```

在其后添加：

```
#ifdef INCLUDE_NS83820_END
#   include "sysNs83820End.c"
#endif /* INCLUDE_NS83820_END */
```

找到

```
#ifdef INCLUDE_LN_97X_END
    sysLan97xPciInit ();
#endif /* INCLUDE_LN_97X_END */
```

在其后添加：

```
#ifdef INCLUDE_NS83820_END
   sysNs83820PciInit ();
#endif /* INCLUDE_NS83820_END */
```

将 sysNs83820End.c 拷贝到 BSP 目录下，修改 config.h 中的引导行，将引导设备改为“ns”，制作引导盘，即可通过 83820 网卡引导 VxWorks。

注意，如果系统配置了多块网卡，那么文件 configNet.h 中，作为调试的网卡应放在数组 endDevTbl 的最前面。

第 5 章　Zinc/WindML 本地化

Zinc 是一个面向嵌入式应用、面向对象的图形用户接口开发平台，支持所见即所得的可视化图形用户界面开发。在 VxWorks 操作系统环境中，Zinc 基于 WindML 图形库。WindML 是为支持嵌入式系统开发多媒体应用而设计的媒体库，提供基本图形、视频和音频函数库，并为开发客户化设备驱动程序提供了一个框架。作为 WindML 的一部分，字体引擎实现字符屏幕显示输出功能。WindML 字体引擎在设计时考虑到中文等双字节字符集的输出问题，可以说基本上支持点阵汉字的显示，但是 WindML 自身没有提供汉字字库，并且字符输出函数也没考虑到单字节字符和双字节字符混合串输出的问题，因此当系统需要显示中文时，需要对其修改。

5.1　WindML 字体字符显示原理

字符输出是图形系统的必要组成部分之一。字符屏幕输出需要借助于图形系统作图函数，如点阵字体通常采用画点函数向屏幕打点方式实现，矢量字体则采用多边形填充函数向屏幕输出字形信息。中文输出显示应该遵循 WindML 原有的字体驱动体系。

5.1.1　WindML 字体驱动体系

WindML 字体驱动提供一个在字体引擎专有 API 之上的公用的、可移植的透明抽象层。它隔离不同字体引擎的差异，向应用层提供一套公用的字体 API。其体系结构如图 5-1 所示。

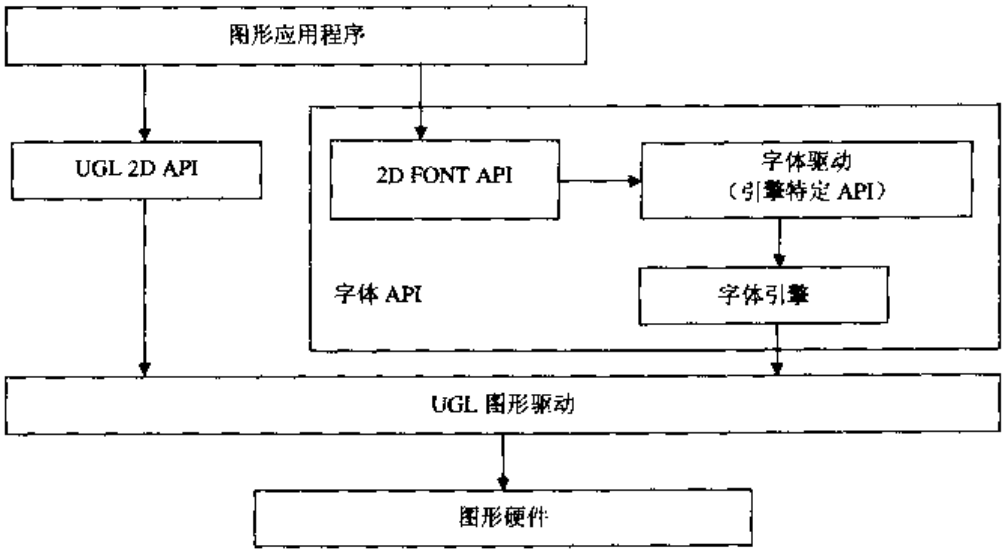


图 5-1　WindML 字体驱动体系示意图

WindML 自身支持两种字体，分别是 AGFA 和 BMF。AGFA 是一种矢量字体，WindML2.0 曾提供相关字体驱动，但没有提供相应的字库，在 WindML3.0 中，这种字体不再支持。BMF

是 Bitmap Font 的缩写，是一种位图字体，是目前 WindML 唯一支持的字体。本章介绍的中文字库就是采用这种字体。

5.1.2　WindML 字符输出函数

WindML 支持双字节编码字库，它提供两套字符输出函数，分别支持单字节字符和双字节字符的输出。WindML 字符输出过程如图 5-2 所示。

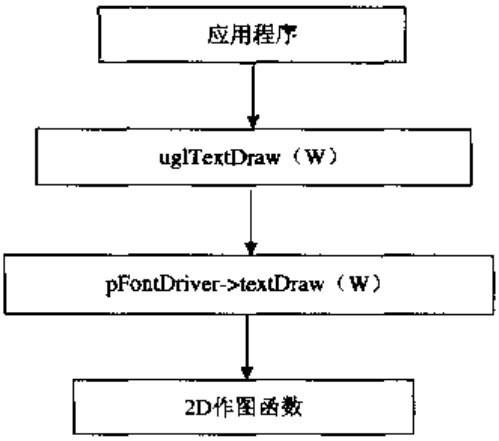


图 5-2　WindML 字符输出过程

应用程序调用字符串显示接口函数 uglTextDraw/uglTextDrawW，并传入要显示的字符串及字体、位置、GC（图形上下文）等信息。uglTextDraw/uglTextDrawW 根据传入的参数调用对应字体驱动的 textDraw/textDrawW 函数，完成字符的位图显示。其中 uglTextDraw 与 uglTextDrawW 分别用于单字节字符和双字节字符串的输出。定义如下：

```
UGL_STATUS uglTextDraw          /*单字节文本串显示函数*/
    (
    UGL_GC_ID gc,               /* 图形上下文 */
    UGL_POS x,                  /* 文本串的位置：X */
    UGL_POS y,                  /* 文本串的位置：Y */
    UGL_SIZE length,            /* 文本串的长度 */
    const UGL_CHAR *text        /* 单字节字符串 */
    )
UGL_STATUS uglTextDrawW         /*双字节文本串显示函数*/
    (
    UGL_GC_ID gc,               /* 图形上下文 */
    UGL_POS x,                  /* 文本串的位置：X */
    UGL_POS y,                  /* 文本串的位置：Y */
    UGL_SIZE length,            /* 文本串的长度 */
    const UGL_WCHAR *text       /* 双字节字符串 */
    )
```

而在我们在 VC 或其他常用编辑器下编辑 C 代码时，输入了一个含中文和西文的混合文本字符串，例如输入：

```
char text[32]="abc 中国 123";
uglTextDraw(gc, 0, 0, -1, text);
```

通常保存为 ANSI 编码方式的文本文件，这时文件存放方式为单双字节混合存储。例如上述代码保存后，用 VC 或 UltraEdit 等支持二进制方式的工具以二进制方式打开，如图 5-3 所示，左边的红色方框部分为字符串对应的编码，可以看到西文字符以 ASCII 码方式存放，中文字符则以其区位码存放。

当含有该代码的文件编译、链接后，在执行 uglTextDraw 或 uglTextDrawW 时，text 参数包含的内容就是左边的红色方框中的内容（0x61（a）、0x62（b）、0x63（c）、0xD6D0（中）、0xB9FA（国）、0x31（1）、0x32（2）、0x33（3））。即使包含了中文字库，uglTextDraw 与 uglTextDrawW

都不能正确处理这种中西文混合字符串的显示。需要在调用底层字体驱动对应的字符显示函数（pFontDriver->textDraw/ textDraw W）之前，先作预处理正确区分中文字符和西文字符。

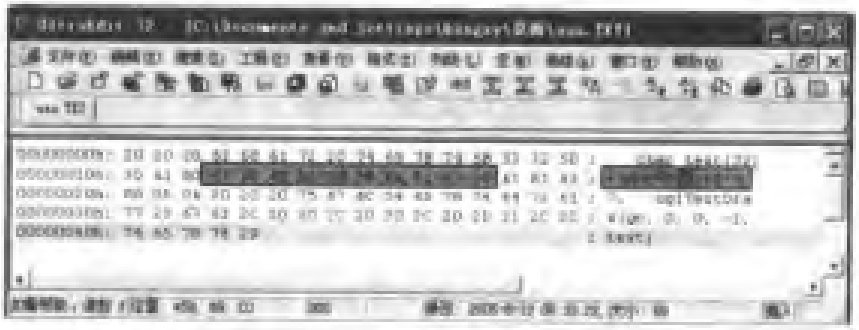


图 5-3　文本文件中 WindML 字符输出过程

5.2 BMF 字 体

BMF 字体是一种位图字体，即字体信息是以位图形式存放。WindML BMF 字体存放在 target\src\ugl\fonts\bmf 目录下，对应的字体驱动存放在 target\src\ugl\driver\font\bmf 目录下。

5.2.1 BMF 字体结构

BMF 是一种双字节字体。一个字符可由页、页内索引来定位，页类似 GB2312 汉字库中的区码、页内索引类似汉字的位码。每种 BMF 字体由一个 UGL_BMF_FONT_DESC 结构描述其属性。

UGL_BMF_FONT_DESC 结构定义如下：

```
typedef struct ugl_bmf_font_desc    {
    /* Standard Header */
    UGL_FONT_DESC header;
    UGL_SIZE leading;
    UGL_SIZE maxAscent;
    UGL_SIZE maxDescent;
    UGL_SIZE maxAdvance;
    const UGL_UINT8 * const * pageData;          //位图数据
    } UGL_BMF_FONT_DESC;
```

其中，pageData 是 char 型指针，指向对应字库的图形信息数据，这些数据包括每个字符的页码、页内索引、大小（宽度和高度）及位图信息。

Header 是一个 UGL_FONT_DESC 类型结构，用以描述该字体的相关信息，如像素大小、是否倾斜、字符间间距、字符集等信息。

UGL_FONT_DESC 定义如下：

```
typedef struct ugl_font_desc    {
    UGL_RANGE pixelSize;          /* average size of font in pixels */
    UGL_RANGE weight;             /* weight is a bold setting from 0 - 100 */
    UGL_ORD italic;               /* Italic is usually either on or off */
    UGL_ORD spacing;              /* Mono spaced or proportional */
```

```
    UGL_ORD charSet;              /* ISO 8859-1, Unicode, etc */
    char faceName[UGL_FONT_FACE_NAME_MAX_LENGTH];  /* face name of font */
    char familyName[UGL_FONT_FAMILY_NAME_MAX_LENGTH];
    } UGL_FONT_DESC;
```

下面以 ufcbi12.c 为例分析字体文件结构。ufcbi12.c 是 Courier Bold Oblique（粗、斜体）12 点阵字体。

该文件的第一行为：

```
/* ufcbi12.c - Courier_Bold_Oblique_12 font for BMF font driver */
```

其中，“-”后面的“Courier_Bold_Oblique_12”，将由 WindML Configuration 解释为“Courier Bold Oblique 12”字体，如图 5-4 所示。即第一个单词中的下划线替换为空格。WindML Configuration 忽略其余的注释内容。

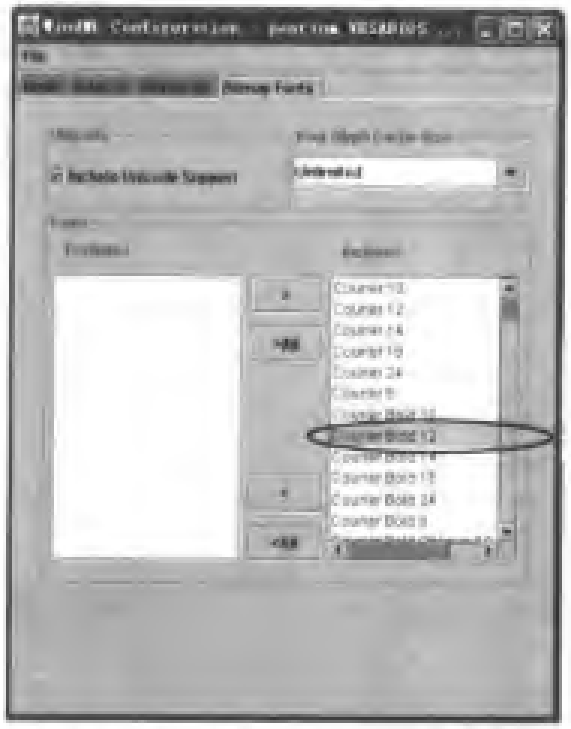


图 5-4　WindML Configuration 屏幕截图

注释后面是头文件的引用：

```
#include <ugl/driver/font/udbmfnt.h>
```

紧接着是位图字体的每页的信息：

```
UGL_LOCAL const unsigned char page0Data[] =
    {
                /* 0x0020 (' ') */      /* 空格 */
     0,         /* page */              /* 该字符页编号 */
    32,         /* index */             /* 该字符在页内的编号 */
     0,         /* size (MSB) */        /* 位图信息字节数（高字节）*/
     5,         /* size (LSB) */        /* 位图信息字节数（低字节），位图的信息字节数
等于从本字节开始到字体位图所占用的字节数，本例为 5（4+1）*/
     7,         /* width */             /* 宽度像素数 */
```

```
        1,          /* height */            /* 高度像素数 */
        1,          /* ascent */            /* 上升像素数 */
      0x00,                                 /* 字体位图，空格字符只有一个字节 */

                    /* 0x0021 ('!') */      /* 感叹号 */
        0,          /* page */              /* 该字符页编号 */

       33,          /* index */             /* 该字符在页内的编号 */
        0,          /* size (MSB) */        /* 位图信息字节数（高字节） */
       12,          /* size (LSB) */        /* 位图信息字节数(低字节)，位图的信息字节数
等于从本字节开始到字体位图所占用的字节数，本例为 12(4+8)*/
        7,          /* width */
        9,          /* height */
        9,          /* ascent */
      0x0c, 0x18, 0x20, 0x41, 0x82, 0x00, 0x18, 0x30,
                                            /* 字体位图，感叹号有 8 个字节 */
    ...
    ...
    ...
                    /* 0x00ff */
        0,          /* page */
      255,          /* index */
        0,          /* size (MSB) */
       17,          /* size (LSB) */        /* 17(4+13) */
        9,          /* width */
       11,          /* height */
        8,          /* ascent */
      0x33, 0x00, 0x1e, 0xe6, 0x63, 0x20, 0xe0, 0x60, 0x30, 0x30, 0x30, 0x38, 0x00,
/* 字体位图，本字符有 13 个字节 */

                    /* End of page */       /* 本页结束，用四个 0 表示 */
      0, 0, 0, 0
      };
```

由于 Courier 是西文字符集，西文字符可见字符小于 256 个，所以仅有第 0 页，其他页为空。

在所有页信息结束时的代码为：

```
UGL_LOCAL const unsigned char * const pageArray[] =
    {
    page0Data,
    UGL_NULL
    };
```

其中数组 pageArray 由所有页信息组成的位图数据，并以 UGL_NULL 作为结束。

最后是字体的数据结构：

```
const UGL_BMF_FONT_DESC uglBMFFont_Courier_Bold_Oblique_12 =
    {
    /* UGL_FONT_DESC structure */
```

```
{
{ 12, 12},                              /* pixelSize */
{UGL_FONT_BOLD, UGL_FONT_BOLD },        /* weight */
UGL_FONT_ITALIC,                        /* italic */
UGL_FONT_MONO_SPACED,                   /* spacing */
UGL_FONT_ISO_8859_1,                    /* char set */
"Courier Bold Oblique",                 /* face name */
"Courier"                               /* family name */
},

/* UGL_BMF_FONT_DESC structure */

 1,                                     /* leading */
11,                                     /* maxAscent */
 3,                                     /* maxDescent */
 7,                                     /* maxAdvance */
pageArray                               /* glyph pages */
};
```

包含了该字体的 face name 和 family name，以及所有字符的位图数据 pageArray。

5.2.2　BMF 字体驱动

udbmffnt.c 是 BMF 字体驱动程序源代码。

5.2.2.1　BMF 字体驱动主要函数

按照 WindML 字体驱动要求，udbmffnt.c 提供以下函数：

（1）uglBMFFontDriverCreate：　创建 BMF 字体驱动。
（2）uglBMFFontCreate：　创建一个 BMF 字体。
（3）uglBMFFontDestroy：　注销一个 BMF 字体。
（4）uglBMFFontDriverDestroy：　注销 BMF 字体驱动。
（5）uglBMFFontFindFirst：　在 BMF 字体链中查找第一个字体。
（6）uglBMFFontFindNext：　在 BMF 字体链中查找下一个字体。
（7）uglBMFFontFindClose：　结束 BMF 字体链中查找。
（8）uglBMFFontDriverInfo：　获取 BMF 字体驱动信息。
（9）uglBMFFontInfo：　获取 BMF 字体信息。
（10）uglBMFFontMetricsGet：　获取 BMF 字体的度量信息。
（11）uglBMFTextDraw：　用指定 BMF 字体输出单字节字符串。
（12）uglBMFStyledTextDraw：　该函数没有实现。
（13）uglBMFTextSizeGet：　获得采用指定 BMF 字体的单字节字符串大小。
（14）uglBMFTextDrawW：　用指定 BMF 字体输出双字节字符串。
（15）uglBMFStyledTextDrawW：　该函数没有实现。
（16）uglBMFTextSizeGetW：　获得采用指定 BMF 字体的双字节字符串大小。

5.2.2.2　BMF 字体使用流程

在 WindML 中，使用一种 BMF 字体，通常需要以下几步：

（1）获取已注册字体驱动信息。

```
pRegistryData = uglRegistryFind (UGL_FONT_ENGINE_TYPE, 0, 0, 0);
```

（2）取已注册字体驱动的 ID 号。

```
fontDrvId = (UGL_FONT_DRIVER_ID)pRegistryData->id;
```

（3）找到对应名称和大小的字体定义。

```
uglFontFindString(fontDrvId, "familyName= Courier; pixelSize = 12",
&systemFontDef);
```

（4）创建该字体。

```
uglFontCreate(fontDrvId, &systemFontDef);
```

（5）得到该字体文本串大小，调用 **uglTextDraw** 显示文本串。

```
uglFontSet(gc, systemFontDef);
uglTextSizeGet(systemFontDef, UGL_NULL, &tmp, -1, fontTestText);
uglTextDraw(gc, 0, y, -1, fontTestText);
```

在调用 uglFontCreate 创建一个 BMF 字体时，将调用函数 uglBMFFontCreate。uglBMFFontCreate 将对应的 uglBMFFont_Courier_Bold_Oblique_12 结构中的相应信息转化为一个类型为 UGL_BMF_FONT 的结构，该结构包含了所有字符的位图信息。

函数 uglTextDraw/uglTextDrawW 将调用对应的 uglBMFTextDraw/uglBMFTextDrawW，uglBMFTextDraw/uglBMFTextDrawW 根据传入的（需要显示）文本串的每个字符所在页号和页内索引，定位其位图信息，使用 gc 中指定的颜色将位图显示到屏幕上。

5.3 汉字 BMF 字库构造

根据前面分析的 BMF 字库结构以及字体驱动的原理，只要按照 BMF 字库结构来构造汉字库，WindML 就可支持汉字显示。

5.3.1 转换方法

汉字库构造有三种方法：

（1）WRS 提供了一个名为 UGL Font Editor 工具，可以从 TTF 字库转化得到 BMF 字库，可以得到所有 TTF 字体对应的 BMF 字体，该工具可将转换结果保存为 BMF 字库结构的 C 文件。但是该工具在转换得到中文字库每页缺少的第一个字。该软件运行时屏幕截图如图 5-5 所示。

该工具转换得到的字库是 UNICODE 编码，每个汉字的页码及页内索引是由该汉字对应的 UNICODE 编码决定（高八位为页号，低八位为页内索引）。

（2）编写相应的工具，由 gb2312 点阵字库转化，需要从头开始编写完整代码。

（3）利用工具 bdf2bmf，由 Linnx bdf 字库转化成 BMF 字库。从 Internet 上可下载相应工具，适当修改，即可完成相应字库的转化，BFD 格式的中文字库也可从 Internet 上下载。WindML 自身提供的西文字体即是由 BDF 字库转化得到。

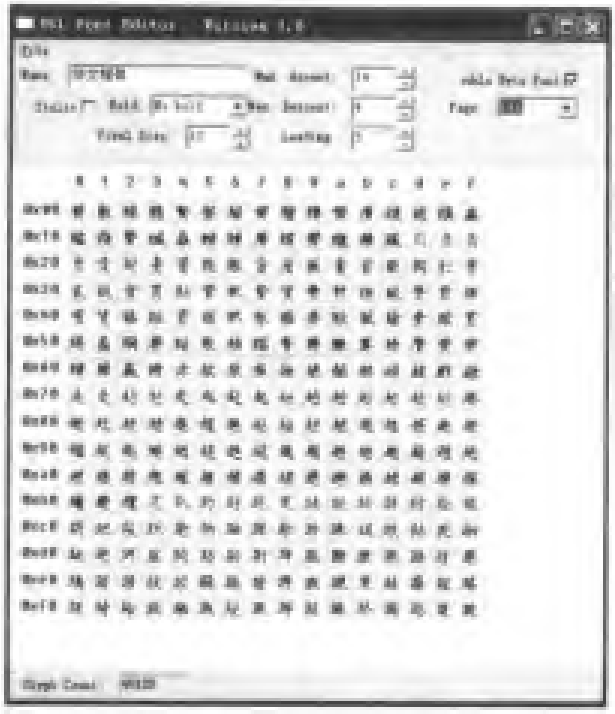


图 5-5　UGL Font Editor 工具界面

5.3.2　BDF 字库结构

BDF 字库是用于 X Window 系统的位图字体。通常结构及解释如下，其中由/**/括起来的部分是相应的解释。

```
STARTFONT 2.1          /* 标示字体开始，并与最后的 ENDFONT 相呼应*/
COMMENT                /* 以 COMMENT 开始的行是注释行 */
COMMENT  Copyright (C) 1988  The Institute of Software, Academia Sinica.
COMMENT
...(一系列注释)
COMMENT
/* 以下是字体属性信息：名称、大小等 */
FONT -ISAS-Fangsong ti-Medium-R-Normal--16-160-72-72-c-160-GB2312.1980-0
SIZE 16 72 72
FONTBOUNDINGBOX 16 16 0 -2
STARTPROPERTIES 19
FONTNAME_REGISTRY ""
FOUNDRY "ISAS"
FAMILY_NAME "Fangsong ti"
WEIGHT_NAME "Medium"
SLANT "R"
SETWIDTH_NAME "Normal"
ADD_STYLE_NAME ""
PIXEL_SIZE 16
POINT_SIZE 160
RESOLUTION_X 72
RESOLUTION_Y 72
SPACING "C"
```

```
AVERAGE_WIDTH 160
CHARSET_REGISTRY "GB2312.1980"
CHARSET_ENCODING "0"
FONT_ASCENT 14
FONT_DESCENT 2
DEFAULT_CHAR 8481
COPYRIGHT "Copyright (c) 1988  The Institute of Software, Academia Sinica."
ENDPROPERTIES           /* 字体属性信息结束 */
CHARS 7614              /* 字符数目 */
STARTCHAR 0x2121        /* 标示一个字符编码开始，其后数字为其编码 */
ENCODING 8481
SWIDTH 1000 0
DWIDTH 16 0             /* 字符宽度 */
BBX 16 16 0 -2          /* 边界 */
BITMAP                  /* 以下为位图信息 */
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
ENDCHAR                 /*  标示一个字符编码结束 */
/* 以下是后续字符的信息 */
STARTCHAR 0x2122
ENCODING 8482
SWIDTH 1000 0
DWIDTH 16 0
BBX 16 16 0 -2
BITMAP
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
```

```
6000
3000
1000
0000
0000
ENDCHAR
...
/* 最后一个字符信息 */
STARTCHAR 0x777e
ENCODING 30590
SWIDTH 1000 0
DWIDTH 16 0
BBX 16 16 0 -2
BITMAP
1010
3e14
32fe
2a10
3e38
0056
7f90
497c
7f44
497c
7f44
007c
ff44
2200
22fe
4200
ENDCHAR
ENDFONT                    /* 字体文件结束 */
```

事实上对于大部分中文 BDF 字库文件，例如 16 点阵的字库，每个汉字的宽度、高度和位图信息的长度都是相同的。

5.3.3　从 BDF 字库转换得到 BMF 字库

了解了 BDF 字库和 BMF 字库文件的结构，实现二者间的转换就比较简单。二者文件头部分都是固定格式，需要转换的是将每个字符的信息按照相应字库要求的格式重新格式化。例如，对于 16 点阵 BDF 字库中的编码为 0x777e 的字符（见上一节），它在相应的 BMF 字库文件描述信息应为：

```
UGL_LOCAL const unsigned char page119Data[] =
  {
    ...
                    /* 0x00fe */
        247,        /* page */
        254,        /* index */
         0,         /* size (MSB) */
```

```
            36,          /* size (LSB) */
            16,          /* width */
            16,          /* height */
            14,          /* ascent */
            0x10, 0x10, 0x3e, 0x14, 0x32, 0xfe, 0x2a, 0x10, 0x3e, 0x38, 0x00, 0x56,
0x7f, 0x90, 0x49, 0x7c, 0x7f, 0x44, 0x49, 0x7c, 0x7f, 0x44, 0x00, 0x7c, 0xff, 0x44,
0x22, 0x00, 0x22, 0xfe, 0x42, 0x00,
        ...
        }
```

下面给出简单的转换代码：

```
int readline(FILE *fp,unsigned char * buff,unsigned long location);
void Usage();

int main(int argc, char* argv[])
{
    FILE *ifp,*ofp;
    int first=0,i=0,j,k;
    unsigned char pageno=-1;
    unsigned long location=0;
    char *ifn,*ofn,*fontname;
    unsigned char buff[LINEMAXLENG],pageExist[256];
    unsigned char fontSize[3];
    unsigned char fontFamilyNmae[40];
    unsigned short fontAscent;
    unsigned short fontDescent;
    unsigned short fontCharNum;

    unsigned short StartChar;
    unsigned short Encoding;
    unsigned short Swidth[2];
    unsigned short Dwidth[2];
    unsigned short Bbx[4];
    unsigned int Bitmap[256];
    unsigned char BitmapSize;
    unsigned char firstLine[80];
    int aa;
    if(argc<4)
        return 0;
    ifn=argv[1];
    ofn=argv[2];
    fontname=argv[3];

    if((ifp=fopen(ifn,"rb"))==NULL){
        printf("\ncannot open file:%s\n",ifn);
        return 0;
    }
    if((ofp=fopen(ofn,"w"))==NULL){
        printf("\ncannot open file:%s\n",ofn);
        return 0;
```

```
        }

        for(j=0;j<80;j++)
            firstLine[j]=0x20;
        firstLine[79]=0;

        for(j=0;j<256;j++)
            pageExist[j]=0;

        fprintf(ofp,"%s\n",firstLine);

        fprintf(ofp,"/*\n *\n");
    /* 以下代码转换所有以 "COMMENT" 开头的注释 */
        location+=readline(ifp,buff,location);
        do{
            location+=readline(ifp,buff,location);
            if(strncmp((const char *)buff,"COMMENT",7)==0){ // COMMENT line
                printf("**");
                if(&buff[8]!=0)
                    fprintf(ofp," * %s",&buff[8]);
                else
                    fprintf(ofp," * \n");
            }else{                                          // COMMENT end
                fprintf(ofp," *\n*/\n");
                break;
            }
        }while(1);

    /* 以下代码，提取字体属性信息*/

        do{
            if(strncmp((const char *)buff,"FAMILY_NAME",11)==0){
                                                            // FAMILY_NAME
                strncpy((char  *)fontFamilyNmae,(const  char  *)&buff[13],strlen
((const char *)buff)-15);
                fontFamilyNmae[strlen((const char *)buff)-15]=0;
                goto readNext;
            }

            if(strncmp((const char *)buff,"SIZE",4)==0){    // SIZE
                sscanf((const char *)&buff[5],"%d %d %d",&fontSize[0],&fontSize[1],
&fontSize[2]);
                goto readNext;
            }

            if(strncmp((const char *)buff,"FONT_ASCENT",11)==0){
                                                            // FONT_ASCENT
                sscanf((const char *)&buff[12],"%d",&fontAscent);
                goto readNext;
            }
```

```
        if(strncmp((const char *)buff,"FONT_DESCENT",12)==0){
                                                    // FONT_DESCENT
            sscanf((const char *)&buff[13],"%d",&fontDescent);
            goto readNext;
        }

        if(strncmp((const char *)buff,"ENDPROPERTIES",13)==0)
                                                    // ENDPROPERTIES
            break;

    readNext:
        location+=readline(ifp,buff,location);
      }while(1);

      fprintf(ofp," /* Copyright 2001 kongxiangying .*/\n");
      fprintf(ofp," /* This file was converted to C from %s using the b2u utility.
*/\n\n", ifn);
      fprintf(ofp," #include <ugl/driver/font/udbmffnt.h>\n\n");

      location+=readline(ifp,buff,location);

      if(strncmp((const char *)buff,"CHARS",5)==0)          // CHARS
          sscanf((const char *)&buff[6],"%d",&fontCharNum);

    /* 以下代码，逐个转换每个字符的位图信息*/

      do{
          BitmapSize=0;
          location+=readline(ifp,buff,location);
          if(strncmp((const char *)buff,"STARTCHAR",9)==0)    // CHARS
              sscanf((const char *)&buff[10],"%04x",&StartChar);
          location+=readline(ifp,buff,location);
          if(strncmp((const char *)buff,"ENCODING",8)==0)     // CHARS
              sscanf((const char *)&buff[9],"%d",&Encoding);
          location+=readline(ifp,buff,location);
          if(strncmp((const char *)buff,"SWIDTH",6)==0)       // CHARS
              sscanf((const char *)&buff[7],"%d %d",&Swidth[0],&Swidth[1]);
          location+=readline(ifp,buff,location);
          if(strncmp((const char *)buff,"DWIDTH",6)==0)       // CHARS
              sscanf((const char *)&buff[7],"%d %d",&Dwidth[0],&Dwidth[1]);
          aa=Dwidth[0];
          printf("\n %d %d",Dwidth[0],aa);

          location+=readline(ifp,buff,location);
          if(strncmp((const char *)buff,"BBX",3)==0)          // CHARS
              sscanf((const  char  *)&buff[4],"%d  %d  %d  %d",&Bbx[0],&Bbx[1],
&Bbx[2],&Bbx[3]);
          location+=readline(ifp,buff,location);
          if(strncmp((const char *)buff,"BITMAP",6)==0)       // CHARS
```

```
            do{
                location+=readline(ifp,buff,location);
                if(strncmp((const char *)buff,"ENDCHAR",7)==0)        // CHARS
                    break;
                else{
                    if(strncmp((const char *)buff,"ENDFONT",7)!=0)    // CHARS
                        sscanf((const char *)buff,"%x",&Bitmap[BitmapSize++]);
                    else
                        break;
                }
            }while(1);

        if(pageno!=((StartChar)&0xff00)>>8){
            if(i!=0){
                fprintf(ofp," /* End of page */ \n    0, 0, 0, 0\n };\n\n");
                if(i==indexSize)
                    break;
            }
            pageno=((StartChar)&0xff00)>>8;
            pageExist[pageno]=1;
            fprintf(ofp," UGL_LOCAL const unsigned char page%dData[] = \n
{\n",pageno);
        }
        fprintf(ofp,"            /* 0x%04x */\n", (StartChar)&0x00ff);
        fprintf(ofp,"    %d,     /* page */ \n", pageno );
        fprintf(ofp,"    %d,     /* index */ \n", (StartChar)&0x00ff);
        fprintf(ofp,"     0,     /* size (MSB) */ \n");
        fprintf(ofp,"    %d,
                /* size (LSB) */ \n",4+BitmapSize* (fontSize [0]/8));
        fprintf(ofp,"    %2d,    /* width */ \n",aa);
        printf(" %d",aa);
        fprintf(ofp,"    %2d,    /* height */ \n",aa);
        fprintf(ofp,"    %d,     /* ascent */ \n",aa/*fontAscent*/);

        fprintf(ofp,"    ");
        for(j=0;j<BitmapSize;j++){
            for(k=(fontSize[0]/8)-1;k>=0;k--)
                fprintf(ofp,"0x%02x, ",(Bitmap[j]>>(8*k))&0xff);
        }

        fprintf(ofp,"\n\n ");

        i++;
    }while(!feof(ifp));
    fprintf(ofp,"           /* End of page */ \n   0, 0, 0, 0\n };\n\n");

    fprintf(ofp,"   UGL_LOCAL const unsigned char * const pageArray[] =\n
{\n");
    for(j=0;j<256;j++){
        if(pageExist[j]==1)
```

```
            fprintf(ofp,"    page%dData,\n",j);
        }
        fprintf(ofp,"       UGL_NULL\n   };\n\n");

        fprintf(ofp,"   const   UGL_BMF_FONT_DESC   uglBMFFont_%s_%02d   =\n{\n",
fontFamilyNmae,fontSize[0]);
        fprintf(ofp,"        /* UGL_FONT_DESC structure */\n");
        fprintf(ofp,"       {\n");
        fprintf(ofp,"       {  %02d,  %02d},
                             /* pixelSize */\n",fontSize [0],fontSize[0]);
        fprintf(ofp,"       {UGL_FONT_BOLD_OFF, UGL_FONT_BOLD_OFF},
                                        /* weight */\n");
        fprintf(ofp,"       UGL_FONT_UPRIGHT,
                                        /* italic */\n");
        fprintf(ofp,"       UGL_FONT_MONO_SPACED,
                                        /* spacing */\n");
        fprintf(ofp,"       UGL_FONT_UNICODE,
                                        /* char set */\n");
        fprintf(ofp,"       \"%s %02d\",
                             /* face name */\n",fontFamilyNmae,fontSize[0]);
        fprintf(ofp,"       \"%s\"
                             /* family name */\n",fontFamilyNmae);
        fprintf(ofp,"       },\n\n");

        fprintf(ofp,"        /* UGL_BMF_FONT_DESC structure */\n\n");
        fprintf(ofp,"       1,                  /* leading */\n");
        fprintf(ofp,"       %02d,               /* maxAscent */\n",fontAscent);
        fprintf(ofp,"       %02d,               /* maxDescent */\n",fontDescent);
        fprintf(ofp,"       2,                  /* maxAdvance */\n");
        fprintf(ofp,"       pageArray           /* glyph pages */\n");

        fprintf(ofp,"   };\n");

        fprintf(ofp,"\n                         /*  End!!! */\n");

        fseek(ofp,0,SEEK_SET);

        fprintf(ofp,"/* %s - %s_%02d font from %s
                            */\n",ofn,fontFamilyNmae, fontSize[0],ifn);

        fclose(ifp);
        fclose(ofp);
        printf("\nBye!\n");
        return 1;
    }

    /* 读一行代码函数 */
    int readline(FILE *fp,unsigned char * buff,unsigned long location)
    {
        unsigned long i=0;
```

```
    unsigned char cc;
    fseek(fp,location,SEEK_SET);
    do{
        fread(&cc,1,1,fp);
        buff[i++]=cc;
    }while((cc!=0x0a)&&(i<LINEMAXLENG));
    buff[i]=0;
    return i;
}
```

最后转化得到 C 文件需要作适应性修改。仔细阅读 BDF 字库文件，可以发现每个汉字在 BDF 字库中的编码与区位码还有区别，对应关系为：

BDF 编码+0x8080=区位码

由于汉字在 GB2312 编码文本文件中的存放是按照区位码存放的，为了取得一致，上述转换代码中，每个汉字的页号和页内索引，都应该加上 0x80。

此外，BDF 字库不包括西文字符，可以把大小相近的西文字库中的第 0 页，拷贝过来作为相应的中文字库文件的第 0 页。

5.3.4　中西文混合输出支持

字库文件已经得到，将相应的文件添加到 WindML BMF 字库目录下，然后重新打开 Tornado，运行 WindML 字体配置工具，可以看到新增加的字体文件，如图 5-6 所示。

图 5-6　新添加的中文字体

现在有了中文字库，可以像使用西文字符那样使用中文字体，需要的时候调用 uglBMFTextDrawW 函数就可以输出中文。但是由于 C 代码是以 ANSI 编码的文本文件存放(如前面所述)，中英文分别以单双字节存放，UGL 中 uglBMFTextDraw 与 uglBMFTextDrawW 分别显示单字节和双字节字符输出，然而不能处理混合输出，需要对其修改才能支持中西文的混

合显示。下面介绍修改的过程。

基本思想是在 uglBMFTextDrawW 函数中，对中西文混合字符串进行预处理，将其中单字节西文扩展成双字节，这种扩展可以通过在西文字符编码之前增加 00 实现。例如，当下述代码输出中西文混合字符串时：

```
char text[32]="abc 中国 123";
uglTextDrawW(gc, 0, 0, -1, text);
```

text 的原始内容是：

```
0x61、0x62、0x63、0xD6、D0、0xB9、0xFA、0x31、0x32、0x33
```

扩展后的内容是：

```
0x00、0x61、0x00、0x62、0x00、0x63、0xD6、D0、0xB9、0xFA、0x00、0x31、0x00、0x32、
0x00、0x33
```

由于西文字符的编码在 0～0x7f 之间，中文字符两个字节的编码均大于 0x80，所以很容易区分混合串中的中西文字符。

uglBMFTextDrawW 在文件 uglFont1.c 中，修改后的代码为：

```
UGL_STATUS uglTextDrawW
    (
    UGL_GC_ID gc,                   /* graphics context */
    UGL_POS x,                      /* left position of the text */
    UGL_POS y,                      /* top position of the text */
    UGL_SIZE length,                /* number of characters in text */
    const UGL_WCHAR *text           /* double-byte text array */
    )
    {
    UGL_STATUS status;
    /* 以下加黑代码，将 text 中的西文字符扩展成双字节。*/
    UCHAR *strTemp;
    int leng;
    leng=(int)strlen((const char * )text)+1;
    if((strTemp=malloc(2*leng))==NULL)
        return UGL_STATUS_ERROR;

    leng=myHzStrInit((UCHAR *)text,strTemp,leng-1);

    /* Lock GC and graphics driver */
    if ((status = uglBatchStart(gc)) == UGL_STATUS_OK)
    {
    UGL_FONT_DRIVER * pFontDriver = UGL_NULL;

    if (gc->pFont)
        {
        pFontDriver = gc->pFont->pFontDriver;
        }

    if (pFontDriver != UGL_NULL && pFontDriver->textDrawW != UGL_NULL)
        {
```

```
            status = (*pFontDriver->textDrawW)(gc, x, y, -1, (const UGL_WCHAR
*)strTemp);
            }
        else
            {
            status = UGL_STATUS_ERROR;
            }
        uglBatchEnd(gc);
        }

        free(strTemp);
        return (status);
        }
```

其中，函数 myHzStrInit 完成西文字符的双字节扩展。代码如下：

```
LOCAL int myHzStrInit(UCHAR *inStr,UCHAR *outStr,int leng)
{
    int i=0,j=0;

    while((i<leng) && (inStr[i]!=0) ){
        if(inStr[i]<0x80){
            outStr[j+1]=0x00;
            outStr[j]=inStr[i];
            i+=1;
            j+=2;
        }
        else{
            if((i+1)<leng){
                if(inStr[i+1]>=0x80){
                    outStr[j+1]=inStr[i];
                    outStr[j]=inStr[i+1];
                    i+=2;
                    j+=2;
                }else{
                    i++;
                }
            }else{
                i++;
            }
        }
    }
    outStr[j+1]=0;
    outStr[j]=0;
    return j+2;
}
```

最后，更为完善的修改还要修改函数 uglTextSizeGetW，以获得真正的中西文混合文本串的大小，修改内容与方式与函数 uglTextDrawW 类似。

5.4 Zinc 中文字体支持

Zinc 默认提供五种字体，如表 5-1 所示。

表 5-1 Zinc 默认提供的五种字体

定　　义	字体名称	大　小	备　注
ZAF_FNT_SMALL	Lucida Sans	8	
ZAF_FNT_DIALOG	Lucida Sans	12	
ZAF_FNT_APPLICATION	Helvetica	12	
ZAF_FNT_SYSTEM	Helvetica	12	
ZAF_FNT_FIXED	Courier	12	

其中，ZAF_FNT_SYSTEM 与 ZAF_FNT_APPLICATION 实际上是相同的字体。这些字体的创建在文件 target\src\zinc\generic\ i_ugldsp.cpp 中。假定 WindML 已经增加的四种中文字体，分别是宋体 12、宋体 16、宋体 24，仿宋 16。如下代码将 Zinc 系统五种字体分别定义为：宋体 12、仿宋 16、宋体 16、宋体 16、宋体 24。

```
    uglFontFindString(fontDrvId, "familyName=Songti; pixelSize = 12", &fontDef);
    fontTable[ZAF_FNT_SMALL] = uglFontCreate(fontDrvId, &fontDef);

    uglFontFindString(fontDrvId, "familyName=Fangsong_ti_16; pixelSize = 16",
&fontDef);
    fontTable[ZAF_FNT_APPLICATION] = uglFontCreate(fontDrvId, &fontDef);

    uglFontFindString(fontDrvId, "familyName= Song_ti_16; pixelSize = 16",
&fontDef);
    fontTable[ZAF_FNT_SYSTEM] =uglFontCreate(fontDrvId, &fontDef);

    uglFontFindString(fontDrvId, "familyName=Song_ti_16; pixelSize = 16",
&fontDef);
    fontTable[ZAF_FNT_DIALOG] = uglFontCreate(fontDrvId, &fontDef);

    uglFontFindString(fontDrvId, "familyName=Song_ti_24; pixelSize = 24",
&fontDef);
    fontTable[ZAF_FNT_FIXED] =uglFontCreate(fontDrvId, &fontDef);
```

将上述代码增加到 i_ugldsp.cpp 中，替换原有字体的创建函数，即可将 Zinc 的五种字体改为中文字体。

第 6 章　移植 Microwindows

在 VxWorks 操作系统环境中，除了使用 zinc/WindML 图形用户接口开发平台构建图形用户界面开发之外，还可以用 Microwindows 来开发图形用户界面。与 zinc/WindML 相比较，Microwindows 具有源代码开放、免费等优点。

本章介绍了 Microwindows，以及如何在 VxWorks 操作系统下使用 Microwindows 创建图形用户接口开发平台。

6.1　Microwindows 介绍

Microwindows 是由 Gregory Haerr 组织的一个开放源码项目，起源于 NanoGUI 项目的开放源码的嵌入式 GUI 软件，是嵌入式系统中广泛应用的一种图形用户接口（GUI），该项目的目标是在嵌入式 Linux 平台上提供与普通个人电脑类似的图形用户界面。作为 X_Windows 的替代品，Microwindows 提供了和 X_Windows 类似的功能，但是却占用很少的内存，根据用户的配置，Microwindows 占用的内存资源只有 100～600KB 左右。Microwindows 的核心是基于显示设备接口的，可移植性较强，其本身提供了多种嵌入式系统常见的显示设备驱动程序。目前新版本的 Microwindows 已经内建了 FrameBuffer，因此可以不局限于 Linux 的开发平台，在 eCos、FreeBSD、MINIX、VxWorks 等操作系统上都可以运行。Microwindows 可以使用 FrameBuffer 机制直接读写显存，也可以调用 SVGALib 库。在基于 FrameBuffer 机制中，Microwindows 支持每像素 1 位、2 位、4 位、8 位、16 位、32 位的色彩/灰度，并通过调色板技术将 RGB 格式的颜色空间转换为目标机上的颜色进行显示。Microwindows 的图形引擎能够运行在任何支持 readpixel，writepixel，drawhorzline, drawvertline 和 setpalette 的系统之上。在底层函数的支持之下，上层实现了位图、字体、光标以及颜色的支持。系统使用了优化的绘制函数，这样当用户在移动窗口时可以提供更好的响应。内存图形绘制和移动的实现使得屏幕画图显得很平滑，这点特别在显示动画、多边形绘制、任意区域填充、剪切时有用。Microwindows 系统可以图形方式支持在主机平台上的仿真目标平台开发，因此 Microwindows 应用程序就可以直接在台式机上进行编写和开发，通过交叉编译就可在目标平台上运行。Microwindows 允许设计者轻松加入各种显示设备、鼠标、触摸屏和键盘等。Microwindows 支持窗口覆盖和子窗口概念、完全的窗口和客户区剪切、比例和固定字体，还提供了字体和位图文件处理工具。

Microwindows 目前支持两套接口：Win32 API 接口称为 Microwindows，类 Xlib API 接口称为 Nano-X。Microwindows API 接口采用消息机制实现，提供了 Win32 API 的子集，还实现了一些 Win32 用户模块功能。Nano-X API 接口采用类 X 机制实现，用 X API 编写的应用程序经过较少的修改就能移植到 Nano-X 环境下。

6.2 建立一个基本的 Nano-X 程序

在本节中，我们通过一个简单的例子来说明如何编写一个基本的 Nano-X 程序 arcdemo.c，使读者对 Nano-X 能有一个初步的认识。

```
/*
 * Arc drawing demo for Nano-X
 *
 * Copyright (C) 2002 Alex Holden <alex@alexholden.net>
 * Modified by G Haerr
 */
#include <stdlib.h>
#define MWINCLUDECOLORS
#include "Nano-X.h"

static void
draw(GR_EVENT *ep)
{
    GR_WINDOW_ID wid = ((GR_EVENT_EXPOSURE *)ep)->wid;
    GR_GC_ID gc = GrNewGC();
    int x = 40;
    int y = 40;
    int rx = 30;
    int ry = 30;
    int xoff = (rx + 10) * 2;

    GrSetGCForeground(gc, GREEN);

    /* filled arc*/
    GrArc(wid, gc, x, y, rx, ry, 0, -30, -30, 0, GR_PIE);
    GrArc(wid, gc, x+5, y, rx, ry, 30, 0, 0, -30, GR_PIE);
    GrArc(wid, gc, x, y+5, rx, ry, -30, 0, 0, 30, GR_PIE);
    GrArc(wid, gc, x+5, y+5, rx, ry, 0, 30, 30, 0, GR_PIE);

    /* outlined arc*/
    x += xoff;
    GrArc(wid, gc, x, y, rx, ry, 0, -30, -30, 0, GR_ARCOUTLINE);
    GrArc(wid, gc, x+5, y, rx, ry, 30, 0, 0, -30, GR_ARCOUTLINE);
    GrArc(wid, gc, x, y+5, rx, ry, -30, 0, 0, 30, GR_ARCOUTLINE);
    GrArc(wid, gc, x+5, y+5, rx, ry, 0, 30, 30, 0, GR_ARCOUTLINE);

    /* arc only*/
    x += xoff;
    GrArc(wid, gc, x, y, rx, ry, 0, -30, -30, 0, GR_ARC);
    GrArc(wid, gc, x+5, y, rx, ry, 30, 0, 0, -30, GR_ARC);
    GrArc(wid, gc, x, y+5, rx, ry, -30, 0, 0, 30, GR_ARC);
    GrArc(wid, gc, x+5, y+5, rx, ry, 0, 30, 30, 0, GR_ARC);
```

```
    GrDestroyGC(gc);
}

int
main(int ac, char **av)
{
    GR_EVENT ev;
    GR_WINDOW_ID wid;

    if (GrOpen() < 0)
        exit(-1);

    wid = GrNewWindowEx(GR_WM_PROPS_BORDER|GR_WM_PROPS_CAPTION|
        GR_WM_PROPS_CLOSEBOX, "arcdemo",
        GR_ROOT_WINDOW_ID, 0, 0, 250, 90, WHITE);

    GrSelectEvents(wid, GR_EVENT_MASK_EXPOSURE | GR_EVENT_MASK_CLOSE_REQ);
    GrMapWindow(wid);

    while (1) {
        GrGetNextEvent(&ev);

        if (ev.type == GR_EVENT_TYPE_CLOSE_REQ)
            break;
        if (ev.type == GR_EVENT_TYPE_EXPOSURE)
            draw(&ev);
    }

    GrClose();

    return 0;
}
```

6.2.1 头文件

所有 Nano-X 的程序都要包含"Nano-X.h"头文件。头文件"Nano-X.h"中定义了 Nano-X 中需要用到的基本的结构类型定义及所有提供给用户的函数的声明。

下面对 Nano-X.h 中定义的数据结构做简要说明。

（1）GR_WM_PROPERTIES：描述窗口的管理属性，常由 GrGetWMProperties()或 GrSetWMProperties()函数调用。

（2）GR_WINDOW_INFO：描述窗口属性，由 GrGetWindowInfo ()函数调用。

（3）GR_GC_INFO：图形上下文信息，包含画图的若干信息。

（4）GR_EVENT：所有事件的联合结构，常由 GrGetNextEvent()函数返回。

6.2.2 连接服务器

将应用程序与 Nano-X 服务器相连是一个 Nano-X 程序首先要做的事情，对应用程序来说，这相当于打开显示器，可以采用 GrOpen()函数完成这一工作。当连接服务器成功以后，应用

程序就可以调用 GrGetScreenInfo()函数获取屏幕信息。

6.2.3 创建窗口

在 Nano-X 中，创建窗口的基本的函数是 GrNewWindow，它的用法是：

```
GR_WINDOW_ID GrNewWindow(GR_WINDOW_ID parent, GR_COORD x,
                         GR_COORD y, GR_SIZE width, GR_SIZE height,
                         GR_SIZE bordersize, GR_COLOR background,
                         GR_COLOR bordercolor);
```

其中，GR_WINDGW_ID 结构用来表示窗口的 ID 号，从 1 开始计数。系统的根窗口表示为 GR_ROOT_WINDOW_ID，它的值为 1；parent 为窗口的父窗口，对于程序的最上级窗口来说，它的父窗口是系统的根窗口；x,y,width,height 分别表示窗口的横坐标、纵坐标、宽度和高度；Bordersize 表示窗口边框的大小；Background 和 bordercolor 分别表示窗口的背景色和边框的颜色。

在 Nano-X 中，颜色是由对应的像素值代表和实现的，像素值本身实际上是一个无符号短整形（unsigned short），从它的取值并不能看出所对应的颜色，因此应用程序一般不直接通过选择像素值来选择颜色，而使用 MWRGB 宏来表示。MWRGB 宏用颜色的红绿蓝三色成分来定义颜色。在 include\nxcolor.h 文件中，使用 MWRGB 宏定义了 657 种颜色所对应的像素值，编写应用程序时，可以从中选取需要的颜色。如：黑色可以用 GR_COLGR_BLACK 表示。

在调用 GrNewWindow 函数创建窗口之后，如果需要设置窗口的属性，如标题等，则可以调用 GrSetWMProperties 设置窗口的属性。

GrSetWMProperties 的用法是：

```
void GrSetWMProperties(GR_WINDOW_ID wid, GR_WM_PROPERTIES *props);
```

其中，Wid 表示窗口的 ID 号，结构 GR_WM_PROPERTIES 结构描述窗口的管理属性，它的定义是：

```
typedef struct {
  GR_WM_PROPS flags;        /**< Which properties valid in struct for set*/
  GR_WM_PROPS props;        /**< Window property bits*/
  GR_CHAR *title;           /**< Window title*/
  GR_COLOR background;      /**< Window background color*/
  GR_SIZE bordersize;       /**< Window border size*/
  GR_COLOR bordercolor;     /**< Window border color*/
} GR_WM_PROPERTIES;
```

flags：当用 GrSetWMProperties() 函数设置窗口的管理属性时，根据 flag 域确定 GR_WM_PROPERTIES 结构中其他属性的有效性，可能的取值如下。

```
#define GR_WM_FLAGS_PROPS        0x0001     /* Properties*/
#define GR_WM_FLAGS_TITLE        0x0002     /* Title*/
#define GR_WM_FLAGS_BACKGROUND   0x0004     /* Background color*/
#define GR_WM_FLAGS_BORDERSIZE   0x0008     /* Border size*/
#define GR_WM_FLAGS_BORDERCOLOR  0x0010     /* Border color*/
```

props：包含两类。

一是窗口自身的属性：

GR_WM_PROPS_NOBACKGROUND	不画窗口的背景色
GR_WM_PROPS_NOFOCUS	不聚集本窗口
GR_WM_PROPS_NOMOVE	用户不能移动窗口
GR_WM_PROPS_NORAISE	用户不能提升窗口
GR_WM_PROPS_NODECORATE	不重新装饰窗口
GR_WM_PROPS_NOAUTOMOVE	在第一次映射窗口时不移动窗口
GR_WM_PROPS_NOAUTORESIZE	在第一次映射窗口时不提升窗口

二是与窗口管理器相关的属性：

GR_WM_PROPS_APPWINDOW	不使用窗口管理器提供的边框
GR_WM_PROPS_BORDER	窗口的边宽为一个像素
GR_WM_PROPS_APPFRAME	3D 类型的边框
GR_WM_PROPS_CAPTION	使用标题栏
GR_WM_PROPS_CLOSEBOX	使用关闭按钮

窗口创建以后，如果需要获取窗口信息，可以调用 GrGetWindowInfo 函数。它的用法是：

```
void GrGetWindowInfo(GR_WINDOW_ID wid, GR_WINDOW_INFO *infoptr);
```

结构 GR_WINDOW_INFO 描述窗口信息，它的定义是：

```
typedef struct {
  GR_WINDOW_ID wid;          /**< window id (or 0 if no such window) */
  GR_WINDOW_ID parent;       /**< parent window id */
  GR_WINDOW_ID child;        /**< first child window id (or 0) */
  GR_WINDOW_ID sibling;      /**< next sibling window id (or 0) */
  GR_BOOL inputonly;         /**< TRUE if window is input only */
  GR_BOOL mapped;            /**< TRUE if window is mapped */
  GR_BOOL realized;          /**< TRUE if window is mapped and visible */
  GR_COORD x;                /**< parent-relative x position of window */
  GR_COORD y;                /**< parent-relative  y position of window */
  GR_SIZE width;             /**< width of window */
  GR_SIZE height;            /**< height of window */
  GR_SIZE bordersize;        /**< size of border */
  GR_COLOR bordercolor;      /**< color of border */
  GR_COLOR background;       /**< background color */
  GR_EVENT_MASK eventmask;   /**< current event mask for this client */
  GR_WM_PROPS props;         /**< window properties */
  GR_CURSOR_ID cursor;       /**< cursor id*/
  unsigned long processid;   /**< process id of owner*/
} GR_WINDOW_INFO;
```

其中，wid 表示窗口的 ID 号；Parent、child、sibling 分别表示父窗口、子窗口和下一个兄弟窗口的 ID；mapped 表示是否被映射，当一个窗口被创建时，默认是没有映射的。所以在调用 GrNewWindow 之后，需要调用 GrMapWindow 映射窗口。GrMapWindow 函数递归调用指定的窗口及它的子窗口。

eventmask：窗口的事件屏蔽类型。在 6.2.5 节中描述。

另外一种创建窗口的函数是 GrNewWindowEx，它的用法是：

```
GR_WINDOW_ID GrNewWindowEx(GR_WM_PROPS props, GR_CHAR * title, GR_WINDOW_ID
parent, GR_COORD x, GR_COORD y, GR_SIZE width, GR_SIZE height,GR_COLOR background)
```

GrNewWindowEx 在 GrNewWindow 的基础上增加了 props 和 title 两个参数。使用 GrNewWindowEx 函数可以直接指定窗口的属性和窗口的标题，而不需要调用 GrSetWMProperties 函数重新设置窗口的属性。

6.2.4 图形上下文（GC）

在 Nano-X 中，图形的绘制是通过一些图形基元的来完成，如画点、画线、画圆等。这些图元决定了要绘制的图形的形状，至于具体如何绘制，则由图形上下文（GC）来完成。GC 确定了要绘制图形的前景色、背景色、线型等内容。

结构 GR_GC_INFO 描述了 GC 所包含的信息：

```
typedef struct {
  GR_GC_ID gcid;          /**< GC id (or 0 if no such GC) */
  int mode;               /**< drawing mode */
  GR_REGION_ID region;    /**< user region */
  int xoff;               /**< x offset of user region */
  int yoff;               /**< y offset of user region */
  GR_FONT_ID font;        /**< font number */
  GR_COLOR foreground;    /**< foreground RGB color or pixel value */
  GR_COLOR background;    /**< background RGB color or pixel value */
  GR_BOOL fgispixelval; /**< TRUE if 'foreground' is actually a GR_PIXELVAL */
  GR_BOOL bgispixelval; /**< TRUE if 'background' is actually a GR_PIXELVAL */
  GR_BOOL usebackground;/**< use background in bitmaps */
  GR_BOOL exposure;       /**< send exposure events on GrCopyArea */
} GR_GC_INFO;
```

其中，gcid 表示 gc 的 ID 号，mode 表示绘制模式，如表 6-1 所示。

表 6-1 GC 的绘制模式

模　式	值	操　作	说　明
GR_MODE_COPY	0	src	拷　贝
GR_MODE_SET	1	src	拷　贝
GR_MODE_XOR	2	src ^ dst	异　或
GR_MODE_OR	3	src | dst	或
GR_MODE_AND	4	src & dst	与
GR_MODE_CLEAR	5	0	置 0
GR_MODE_SETTO1	6	11111111	置 1
GR_MODE_EQUIV	7	~(src ^ dst)	反异或
GR_MODE_NOR	8	~(src | dst)	反与非
GR_MODE_NAND	9	~(src & dst)	反或非
GR_MODE_INVERT	10	~dst	取　反

续表

模　式	值	操　作	说　明
GR_MODE_COPYINVERTED	11	src	反拷贝
GR_MODE_ORINVERTED	12	src | dst	反　或
GR_MODE_ANDINVERTED	13	src & dst	反　与
GR_MODE_ORREVERSE	14	src |~dst	或　非
GR_MODE_ANDREVERSE	15	src &~dst	与　非
GR_MODE_NOOP	16	dst	不操作

在这 16 种模式中，最常用的是 GR_MODE_COPY 和 GR_MODE_XOR。

（1）GR_MODE_COPY 表示将要画的像素画到指定的点，不受该点原来的像素值影响。GR_MODE_COPY 模式是系统默认的绘画模式。

（2）GR_MODE_XOR 表示异或操作，即要画的像素与原来的像素值先进行异或操作，然后再画到指定的点。

GrSetGCMode 函数用来设置 GC 的绘制模式。

6.2.5　颜色信息

颜色在 Nano-X 中起着非常重要的作用，它由一个无符号的长整形值（unsigned long）表示，0~7 位表示红色，8~15 位表示绿色，16~23 位表示蓝色，颜色的像素信息如图 6-1 所示。宏 MWRGB(r,g,b) 根据颜色的红、绿、蓝索引计算出颜色的像素值。

图 6-1　颜色像素信息

Nano-X 中定义了一些通用颜色，编程时可以直接使用这些宏表示颜色：

```
#define BLACK          MWRGB( 0   , 0   , 0   )
#define BLUE           MWRGB( 0   , 0   , 128 )
#define GREEN          MWRGB( 0   , 128 , 0   )
#define CYAN           MWRGB( 0   , 128 , 128 )
#define RED            MWRGB( 128 , 0   , 0   )
#define MAGENTA        MWRGB( 128 , 0   , 128 )
#define BROWN          MWRGB( 128 , 64  , 0   )
#define LTGRAY         MWRGB( 192 , 192 , 192 )
#define GRAY           MWRGB( 128 , 128 , 128 )
#define LTBLUE         MWRGB( 0   , 0   , 255 )
#define LTGREEN        MWRGB( 0   , 255 , 0   )
#define LTCYAN         MWRGB( 0   , 255 , 255 )
#define LTRED          MWRGB( 255 , 0   , 0   )
#define LTMAGENTA      MWRGB( 255 , 0   , 255 )
#define YELLOW         MWRGB( 255 , 255 , 0   )
#define WHITE          MWRGB( 255 , 255 , 255 )
/* other common colors*/
#define DKGRAY         MWRGB( 32  , 32  , 32  )
```

6.2.6 事件和选取事件

事件是当鼠标、键盘、屏幕状态改变时，Nano-X 通知各户程序的方法。当鼠标、键盘、屏幕状态改变时，Nano-X 产生事件，各户程序捕获事件并对事件进行相应的处理。

Nano-X 的事件可以分为两类：

（1）与硬件相关的事件，包括鼠标和键盘事件。当键盘中的某一键按下（释放）时，产生键盘事件，键盘事件中包含导致事件产生的字符。当鼠标的按键按下（释放）或鼠标移动时，产生鼠标事件，鼠标事件中包含了鼠标的按键状态及鼠标的位置。

（2）与硬件无关的事件，该类事件类型很多，如鼠标进入或退出某一窗口等。

Nano-X 支持的事件有 22 种，它们是：

GR_EVENT_TYPE_EXPOSURE、GR_EVENT_TYPE_BUTTON_DOWN、GR_EVENT_TYPE_BUTTON_UP、GR_EVENT_TYPE_MOUSE_ENTER、GR_EVENT_TYPE_MOUSE_EXIT、GR_EVENT_TYPE_MOUSE_MOTION、GR_EVENT_TYPE_MOUSE_POSITION、GR_EVENT_TYPE_KEY_DOWN、GR_EVENT_TYPE_KEY_UP、GR_EVENT_TYPE_FOCUS_IN、GR_EVENT_TYPE_FOCUS_OUT、GR_EVENT_TYPE_FDINPUT、GR_EVENT_TYPE_UPDATE、GR_EVENT_TYPE_CHLD_UPDATE、GR_EVENT_TYPE_CLOSE_REQ、GR_EVENT_TYPE_TIMEOUT、GR_EVENT_TYPE_SCREENSAVER、GR_EVENT_TYPE_CLIENT_DATA_REQ、GR_EVENT_TYPE_CLIENT_DATA、GR_EVENT_TYPE_SELECTION_CHANGED、GR_EVENT_TYPE_TIMER、GR_EVENT_TYPE_PORTRAIT_CHANGED。

GR_EVENT_TYPE_EXPOSURE 表示窗口被其他窗口屏蔽之后，Nano-X 剪辑该窗口内容并保存下，当被遮挡部分需要重新显示时产生的事件。EXPOSURE 事件产生之后，需要重新绘制窗口的内容。Nano-X 在窗口第一次显示时就会发出 EXPOSURE 事件。

GR_EVENT_TYPE_MOUSE_ENTER 和 GR_EVENT_TYPE_MOUSE_EXIT 分别表示鼠标进入和退出窗口时产生的事件。

GR_EVENT_TYPE_CLOSE_REQ 表示关闭窗口事件，当鼠标按下窗口关闭按钮时产生。

GR_EVENT_TYPE_UPDATE 表示窗口更新事件，当窗口 map、unmap、改变大小及改变父窗口时产生。

在 Nano-X 中，默认的是不接受这些事件，即对它们不予理睬，应用程序应根据自己的实际需要选择有关的事件，以使程序可以接受这些事件并对它们进行必要的处理。选取事件的函数是 GrSelectEvents()，它的调用方式是：

```
GrSelectEvents(w, GR_EVENT_MASK_EXPOSURE);
```

其中，GR_EVENT_MASK_EXPOSURE 是对要选择的事件的屏蔽，应用程序通过它来设置需要选择的事件，如 GR_EVENT_TYPE_EXPOSURE| GR_EVENT_TYPE_BUTTON_UP| GR_EVENT_TYPE_KEY_DOWN 等。

6.2.7 设置事件循环

Nano-X 是事件驱动的，这就意味着在创建了窗口及选择了相应的事件之后，程序就需要依靠事件执行下一步的命令。大部分应用程序在完成初始化后，就进入了一个无限的事件接收循环，接收到一个事件后调用相应的处理程序进行处理，处理完之后返回事件循环，等待新的事件。比如在本例中，在处理完 GR_EVENT_TYPE_EXPOSURE 事件之后，就接着进入事件循环，等待事件的产生。当 GR_EVENT_TYPE_CLOSE_REQ 事件产生时，程序退出事件等待

循环，然后关闭应用程序。

6.2.8　编译运行 arcdemo

前面我们介绍了如何编写一个基本的 Nano-X 应用程序。在本节中，我们将介绍如何在 linux 平台下编译 arcdemo.c，并运行它。

（1）配置 Nano-X。

Nano-X 的大多数的设置选项在配置文件中，配置文件 config 在 src 目录下。在 configs 目录下放置了许多 config 文件的样本，我们可以将 config.x11 复制到 src 目录下。

（2）编译 Nano-X。

在 microwin/src 下运行

```
make
```

对 Nano-X 进行编译。编译生成的可执行程序放在 microwin/src/bin 目录下，库程序放在 microwin/src/lib 目录下。

（3）运行 Nano-X 服务器。

```
bin/Nano-X & sleep 1
```

（4）运行资源管理器。

```
bin/nanowm&
```

（5）编译 arcdemo.c。

先将文件 arcdemo.c 拷贝到 microwin/src 目录下

```
gcc arcdemo.c -o arcdemo -Iinclude -Llib -lNano-X
```

（6）运行 arcdemo 程序。

```
./arcdemo&
```

运行的结果如图 6-2 所示。

图 6-2　arcdemo 的运行结果

6.3 深 入 Nano-X

6.3.1 Nano-X 的目录树与库

Nano-X 的目录树结构如图 6-3 所示。

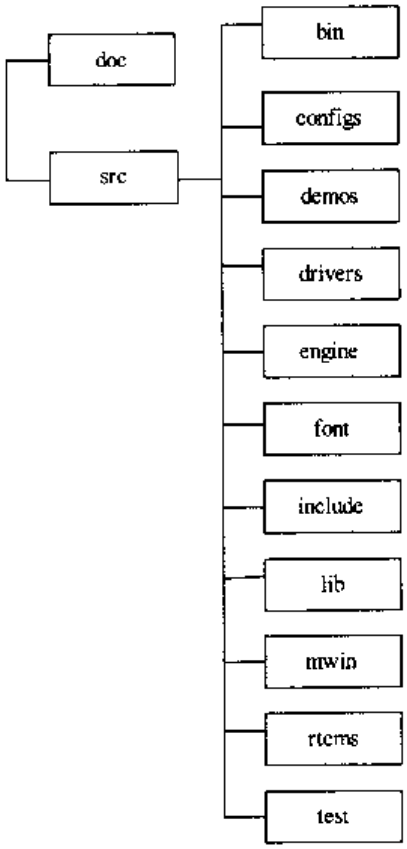


图 6-3 Nano-X 的目录树

（1）doc 目录存放着 Nano-X 的帮助文档。

（2）src 目录存放着 Nano-X 的源代码。具体描述如下：

1）bin：编译后的可执行程序。

2）configs：编译时所需的配置文件。

3）demos：样例程序。

4）drivers：各种驱动程序，包括触摸屏驱动、鼠标驱动、键盘驱动、屏幕驱动等。

5）ecos：在 ecos 系统上运行时所需的支撑程序。

6）engine：图形引擎程序。

7）fonts：字体。

8）include：头文件。

9）lib：编译后的库。

10）mwin：Win32 接口程序。

11）nanox：Nano-X 接口程序。

12）rtems：在 rtems 系统上运行时所需的支撑程序。

13）test：测试程序。

Nano-X 编译成功后，需要的库文件都放在 src/lib 目录下，其中服务器需要的库有 libNano-X.a、libengine.a、libfonts.a、libmdrivers.a，客户程序需要的库有 libnanox.a。

6.3.2 Nano-X 体系结构

Nano-X 与 X_Windows 类似，都采用支持远程连接的客户机/服务器设计模式。采用客户端/服务器的体系结构，服务器的程序在本地工作站上运行，负责实际的窗口绘图工作以及协调不同程序的访问要求。每个程序窗口都被称为客户端，并且与在同一个机器上运行的服务器程序以客户机/服务器关系进行交互。服务器处理所有客户端的作图请求，以及通过消息传递系统和其他客户交互工作。

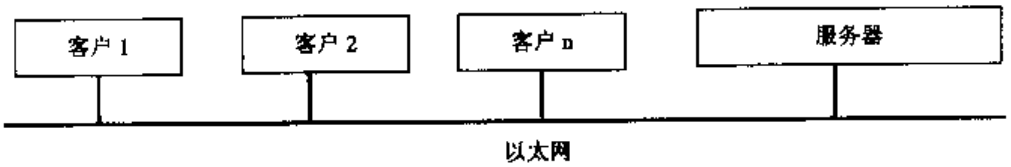


图 6-4 客户端和服务器

服务器使用分层机构的设计方法，允许改变不同的层来适应实际的应用。最底层为驱动层，提供了屏幕、鼠标、触摸屏和键盘底驱动，使程序能访问实际的硬件设备和其他用户定制设备。

中间层是图形引擎层，提供了绘制线条、区域填充、绘制多边形、裁剪和使用颜色模式的方法。最上一层是 API 层，提供了不同的 API 给图形应用程序使用。

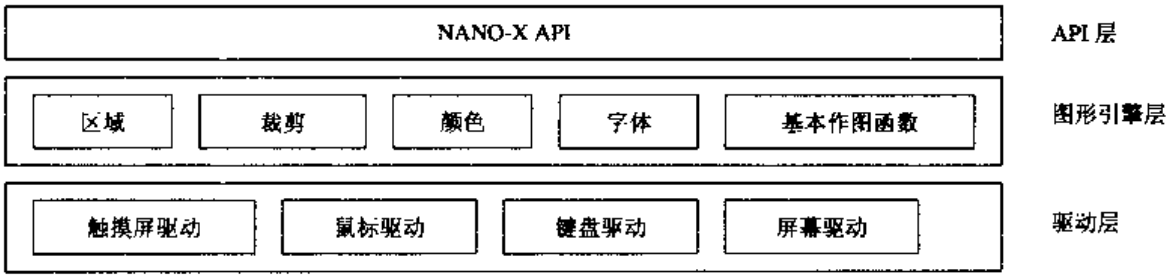


图 6-5　Nano-X 体系结构

与服务器相比，客户端的实现则较为简单，它提供了接口函数 API 给客户程序调用。客户程序通过调用这些 API 给服务器程序发送发送请求，同时通过这些 API 接收服务器程序发来的用户事件。

这种基本的客户端—服务器的体系结构能完成基本的绘制功能，包括窗口及各种图形与字符的绘制等功能。但它不具备窗口管理的功能，如：窗口移动、窗口重叠等。

Nano-X 提供了一个特殊的客户端程序来实现窗口管理的功能，这个窗口管理器称为 nanowm。服务器程序将事件递交给窗口管理器，然后窗口管理器根据事件的类型，完成窗口拖动、窗口重叠、改变窗口大小等功能。

6.4　Nano-X 在 VxWorks 下的实现

前面我们介绍了 Nano-X 的基本原理及体系结构，本节将介绍如何将 Nano-X 移植到 VxWorks 操作系统，在 VxWorks 下移植 Nano-X 需要以下几个环节：

（1）生成库文件。

（2）移植驱动程序，如鼠标驱动、键盘驱动、显示驱动等。

（3）解决服务器与客户端程序函数重名。

（4）处理多个客户端并存的问题。

（5）利用虚拟串口，提高客户端与服务器的通信效率。

6.4.1　生成库文件

Nano-X 中需要生成的库有 libengine.a、libfonts.a、libmdrivers.a、libnanox.a。这些库的生成过程基本一致。

（1）新建可下载工程。

（2）将相应的文件添加到工程中。

（3）修改对应的文件。

（4）修改工程的 C/C++ compiler 选项，使用-I 选项增加 Nano-X 的头文件路径。

（5）将生成目标改成 archive。

（6）编译生成库。

不同的只是第三步和第四步中添加和修改的文件不同，生成库所需要的文件如表 6-2

所示。

表 6-2 生成库需要的文件

库　名	添加的文件	添加的文件所在的目录	描　述
libengine.a	devarc.c、devclip.c、devdraw.c、devdraw.c、devfont.c、devimage.c、devimage_stretch.c、devkbd.c、devlist.c、devmouse.c、devopen.c、devpal1.c、devpal2.c、devpal4.c、devpal8.c、devpalgray4.c、devpoly.c、devrgn2.c、devrgn.c、devstipple.c、devtimer.c、error.c、font_dbcs.c、font_fnt.c、image_bmp.c、image_gif.c、image_jpeg.c、image_png.c、image_pnm.c、image_tiff.c、image_xpm.c、obsolete.c、selfont.c	src\engine	图形引擎库
libfonts.a	obsolete.c、rom8x8.c、rom8x16.c、winFreeSansSerif11x13.c、winFreeSystem14x16.c	src\fonts	字体库
libmdrivers.a	fb.c、fblin1.c、fblin2.c、fblin4.c、fblin8.c、fblin12.c、fblin16.c、fblin24.c、fblin32.c、fblin32alpha.c、fbportrait_down.c、fbportrait_left.c、fbportrait_right.c、genfont.c、genmem.c、kbd_tty.c、mou_tty.c、scr_fb.c	src\drivers	驱动库
libnanox.a	nxutil.c、client.c、clientfb.c、nxproto.c、nxtransform.c	src\nanox	应用程序库

6.4.2 鼠标驱动

6.4.2.1 PS/2 鼠标通信协议

标准的鼠标有两个计数器保持位移的跟踪：X 位移计数器和 Y 位移计数器。可存放 9 位的 2 进制补码并且每个计数器都有相关的溢出标志，它们的内容连同三个鼠标按钮的状态一起以三字节移动数据包的形式发送给主机。位移计数器表示从最后一次位移数据包被送往主机后有位移量发生。

PS/2 鼠标发送位移和按键信息给主机采用如下的 3 字节数据包格式：

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 1	Y overflow	Xoverflow	Y sign bit	X sign bit	Always 1	Middle Btn	Right Btn	Left Btn
Byte 2	X Movement							
Byte 3	Y Movement							

图 6-6 PS/2 鼠标协议报文

（1）第一个字节表示鼠标的状态信息。

表 6-3 鼠标协议第一个字节

位	信　息
0	左键状态：0 为抬起，1 为按下
1	右键状态：0 为抬起，1 为按下
2	中键状态：0 为抬起，1 为按下
3	一直为 1
4	X 方向标记：0 为正向位移，1 为反向位移
5	Y 方向标记：0 为正向位移，1 为反向位移
6	X 溢出标记：0 为未溢出，1 为溢出
7	Y 溢出标记：0 为未溢出，1 为溢出

（2）第二个字节表示鼠标的 X 位移。X 位移表示距上次取值时 X 方向的偏移。当第一个字节的第 4 位为 0 时取值为正；第一个字节的第 4 位为 1 时取值为负。

（3）第三个字节表示鼠标的 Y 位移。Y 位移表示距上次取值时 Y 方向的偏移。当第一个字节的第 5 位为 0 时取值为正；第一个字节的第 5 位为 1 时取值为负。

6.4.2.2　鼠标驱动分析

Nano-X 下的鼠标驱动程序可以分成两层，如图 6-7 所示，上层为协议层，下层为链路层。链路层完成鼠标设备的初始化和挂接中断处理函数，当鼠标移动或按钮按下时，将接收到的数据放到缓冲区中。协议层负责将解释链路层接收到的数据解释成对应的鼠标状态及位置信息，然后提交给 Nano-X 服务器。

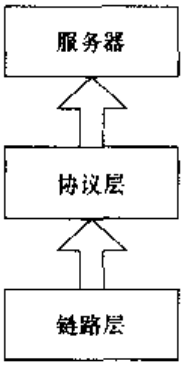


图 6-7　Nano-X 鼠标驱动结构

协议层与服务器间的数据传递是通过数据结构 mousedev 完成。mousedev 结构实现如下：

```
MOUSEDEVICE mousedev = {
    MOU_Open,
    MOU_Close,
    MOU_GetButtonInfo,
    MOU_GetDefaultAccel,
    MOU_Read,
    NULL
};
```

其中，MOU_Open 函数用于打开鼠标设备，然后完成对鼠标设备的初始化，并挂接鼠标的中断处理函数。服务器初始化时调用 MOU_Open，从而完成鼠标设备初始化。

```
static int  MOU_Open(MOUSEDEVICE *pmd)
{
    /* PS/2 mouse*/
    ps2DevCreate("/ps2mse");
    left     = PS2_LEFT_BUTTON;
    right    = PS2_RIGHT_BUTTON;
    middle   = 0;
    parse    = ParsePS2;

    /* open mouse port*/
    mouse_fd = open("/ps2mse", O_NONBLOCK,0);
    if (mouse_fd < 0) {
        EPRINTF("Error %d opening serial mouse type %s on port %s.\n", errno,
type, port);
        return -1;
    }
    /* initialize data*/
    availbuttons = left | middle | right;
    state = IDLE;
    nbytes = 0;
    buttons = 0;
```

```
    xd = 0;
    yd = 0;
    return mouse_fd;
err:
    close(mouse_fd);
    mouse_fd = 0;
    return -1;
}
```

MOU_Open 的调用过程如下：

（1）调用链路层的 ps2DevCreate 函数，ps2DevCreate 完成鼠标驱动的初始化，挂接相应的中断处理函数 i8042MseInt，将鼠标设备注册到系统中。

（2）将函数指针 parse 赋值成 ParsePS2 函数，函数指针 parse 在 MOU_Read 中被调用，它将鼠标数据口的数据解释成鼠标的相应状态。

（3）打开鼠标设备，并置鼠标的初始状态、起始位置等。

```
int ps2DevCreate
    (
    char        *name           /* name to be associated with device   */
    )
    {
    int         mseDrvNum;      /* device number for this driver */
    DEV_HDR     *  pHdr;
    MSE_DEVICE  *  pMseDevice;  /* device descriptors */

    static istrue=0;
    if(istrue==1)
       return OK;
    else
       istrue=1;

    /* 安装驱动*/
    mseDrvNum = iosDrvInstall(mseOpen, (FUNCPTR) NULL, mseOpen,
                              (FUNCPTR) mseClose, tyRead, tyWrite, tyIoctl);

    /* 分配设备结构*/
    pMseDevice = (MSE_DEVICE *)malloc (sizeof(MSE_DEVICE));

    /* 创建设备通道*/
    if (tyDevInit(&pMseDevice->ty_dev, 512, 512, (FUNCPTR)mseTxStart) != OK)
        return (ERROR);

     /* 搭接鼠标中断*/
    (void)  intConnect  (INUM_TO_IVEC  (MSE_INT_VEC),  i8042MseInt,  (int)
pMseDevice);

    /* 初始化 8042 以支持鼠标*/
    mseHwInit ();
```

```
    /* 打开中断*/
    sysIntEnablePIC (MSE_INT_LVL);

    /*把鼠标设备添加到 I/O 系统中*/
    if (iosDevAdd(&pMseDevice->ty_dev.devHdr,name,mseDrvNum) == ERROR)
        return (ERROR);

    return (mseDrvNum);
    }
```

与第 3 章的 i8250tty.c 类似，ps2DevCreate 先将鼠标驱动数据结构注册到系统中，然后分配鼠标设备，挂接相应的中断处理函数 i8042MseInt 并初始化 8042 以支持鼠标，最后把鼠标设备添加到 I/O 系统中。

MOU_Read 函数用于读取鼠标的状态信息，它将链路层接收的数据转换成鼠标的位置信息和状态信息。

```
static int MOU_Read(MWCOORD *dx, MWCOORD *dy, MWCOORD *dz, int *bptr)
{
    int b;
    /*
     *缓冲区没有更多的数据时,读取新到达的数据
     */
    if (nbytes <= 0) {
        bp = buffer;
        if(ioctl(mouse_fd,FIONREAD,&nbytes)==ERROR)
        {
            printf ("ioctl error (errno = %#x)\n", errnoGet ());
            return -1;
        }
        if(nbytes>0)
        {
            nbytes = read(mouse_fd, bp, MAX_BYTES);
            if (nbytes < 0) {
                if (errno == EINTR || errno == EAGAIN)
                    return 0;
#if _MINIX
                return 0;
#else
                return -1;
#endif
            }
        }
    }
    /*
     *从缓冲区中循环读取数据，解析这些数据，
     *直到读取到某一完整的状态，返回。
     *把剩余的字节留给下次调用。
     */
    while (nbytes-- > 0) {
        if ((*parse)((int) *bp++)) {
```

```
            *dx = xd;
            *dy = yd;
            *dz = 0;
            b = 0;
            if(buttons & left)
                b |= MWBUTTON_L;
            if(buttons & right)
                b |= MWBUTTON_R;
            if(buttons & middle)
                b |= MWBUTTON_M;
            *bptr = b;
            return 1;
        }
    }
    return 0;
}
```

真正将数据解析成鼠标位置状态信息的函数是 ParsePS2，ParsePS2 的代码如下：

```
static int ParsePS2(int byte)
{
    switch (state) {
        case IDLE:
            if (byte & PS2_CTRL_BYTE) {
                buttons = byte &
                    (PS2_LEFT_BUTTON|PS2_RIGHT_BUTTON);
                state = XSET;
            }
            break;

        case XSET:
            if(byte > 127)
                byte -= 256;
            xd = byte;
            state = YSET;
            break;

        case YSET:
            if(byte > 127)
                byte -= 256;
            yd = -byte;
            state = IDLE;
            return 1;
    }
    return 0;
}
```

当鼠标发来一帧数据时，MOU_Read 需要调用 ParsePS2 函数 3 次，ParsePS2 函数第一次判断鼠标的按键状态，并将读取状态置为 XSET 状态。第二次计算 X 方向的偏移，并将读取状态置为 YSET 状态。第三次计算 Y 方向的偏移，并将读取状态置为 IDLE 状态，完成一次鼠标状态和偏移的读取。

6.4.3　服务器与客户端函数重名

Nano-X 设计是基于 unix 平台来设计的，服务器与客户端运行在独立的进程空间，因此不会产生符号冲突的问题。但对于 VxWorks 操作系统来说，它运行在任务模式下，如果服务器与客户端使用相同的函数名，则会产生符号冲突的问题。Nano-X 的设计者为了简化设计，在服务器程序和客户端程序中使用了大量的同名函数。要解决此问题，则需将服务器或客户端程序的同名函数改名。但这样将导致大量的重复劳动，并且不能保证修改一定正确。所幸的是，Nano-X 的设计者们已经考虑到此问题，他们使用宏定义将同名函数定义成另一名称，在编译时加予区别。在文件 serv.h 中有如下宏定义：

```
#if defined(__ECOS) && !defined(_NO_SVR_MAPPING)
/*
 * Since eCos is a single task, multi-threaded environment, the server and the
client code share the same namespace.  This means that server functions which are
represented by dispatchers in the client code need to have unique names, thus this
remapping.
 */
#define nxErrorStrings      SVR_nxErrorStrings
#define GrArcAngle          SVR_GrArcAngle
#define GrArc               SVR_GrArc
#define GrArea              SVR_GrArea
...
#define GrSetWindowRegion   SVR_GrSetWindowRegion
#define GrStretchArea       SVR_GrStretchArea
#define GrRedrawncarea      SVR_GrRedrawncarea
#endif
```

我们只要将

```
#if defined(__ECOS) && !defined(_NO_SVR_MAPPING)
```

修改成#if defined(__ECOS) && !defined(_NO_SVR_MAPPING) || SERVER，并在编译服务器工程时修改编译选项，加上-D SERVER 就可以将服务器中所有的同名函数编译成为 SVR_xxx。

6.4.4　多个客户端与服务器

Nano-X 的每一个应用程序首先需要调用 GrOpen 与服务器建立连接，GrOpen 在与服务器建立连接时，使用到全局的变量 nxSocket。nxSocket 表示客户端与服务器通信时 socket 的值。对于 unix 或 linux 操作系统来说，nxSocket 只对本进程有效的，其他进程是无法获取 nxSocket 的值。但对于 VxWorks 系统来说，nxSocket 对所有的任务都是可以访问的。这就导致了一个问题，如果系统中有多个客户端需要与服务器通信，则所有的客户端只能使用一个 Socket 与服务器通信，而服务器无法区分是哪一个客户端发来的请求。当然同样的问题也发生在其他使用的全局变量上。

解决此问题的方法是将 nxSocket 等全局变量由系统中的所有客户端共用改成一个客户端独有。在 VxWorks 系统中，一个任务发起一个客户端，多个客户端对应着多个任务。

我们将所有的这类全局变量组织成数据结构 VxWorks_nanox_client_data。一个任务拥有一个 VxWorks_nanox_client_data 结构，所有任务的 VxWorks_nanox_client_data 通过 next 域构成链表。

```
typedef struct {                                    /* Init to: */
    int                 _nxSocket;                  /*  -1 */
    int                 _storedevent;               /* 0 */
    GR_EVENT            _storedevent_data;          /* no init(0) */
    int                 _regfdmax;                  /* -1 */
    fd_set              _regfdset;                  /* FD_ZERO */
    GR_FNCALLBACKEVENT  _GrErrorFunc;               /* GrDefaultErrorHandler */
    REQBUF              _reqbuf;
    EVENT_LIST          *_evlist;
    int                  taskId;                    //任务的 ID 号
    VXWROKS_NANOX_CLIENT_DATA* next;
                                        //指向下一个 vxworks_nanox_client_data 结构
} vxworks_nanox_client_data;
```

当某一个应用程序调用 GrOpen 与服务器建立连接时，GrOpen 先调用函数 init_per_task_data 分配任务所需的 VxWorks_nanox_client_data 结构，并赋予相应的初值。

```
void init_per_task_data()
{
        dptr=(vxworks_nanox_client_data*) malloc(sizeof(vxworks_nanox_client_
data));

        dptr->_nxSocket = -1;
        dptr->_storedevent = 0;
        dptr->_regfdmax = -1;
        FD_ZERO(&dptr->_regfdset);
        dptr->_GrErrorFunc = GrDefaultErrorHandler;
        dptr->_reqbuf.bufptr = NULL;
        dptr->_reqbuf.bufmax = NULL;
        dptr->_reqbuf.buffer = NULL;
        dptr->_evlist = NULL;
        dptr->taskId=getpid();
        if(data_head==NULL)
        {
           dptr->next=NULL;
           data_head=dptr;

        }
        else
        {
           dptr->next=(VXWROKS_NANOX_CLIENT_DATA*)data_head;
           data_head=dptr;
        }
}
```

当应用程序调用其他绘制函数与服务器通信时，它首先调用 VxWorks_task_get_data 函数获取该任务所需的 VxWorks_nanox_client_data 结构。

```
vxworks_nanox_client_data* vxworks_task_get_data(void)
   {
        int taskId;
        vxworks_nanox_client_data* data_ptr;
        /*获取当前任务的 id 号*/
        taskId=getpid();
        data_ptr=data_head;

        /*遍历 vxworks_nanox_client_data 结构链表，取出当前任务的 vxworks_nanox_
client_data 结构*/
        while(data_ptr->taskId!=taskId)
        {
            if(data_ptr->next==NULL)
                return NULL;
            data_ptr=(vxworks_nanox_client_data*)data_ptr->next;
        }
        return data_ptr;
   }
```

6.4.5　虚拟串口

Nano-X 的客户端与服务器之间的通信使用 TCP 网络协议，由于 VxWorks 网络协议的效率不是很高，导致当窗口拖动和窗口切换时速度很慢。对于目前许多的应用来说，往往客户端与服务器放在同一台机器上，因此我们需要修改客户端与服务器间的通信方式，提高客户端与服务器间的通信速度，从而解决窗口拖动和窗口切换时速度慢的问题。

6.4.5.1　客户端与服务器间的连接过程

首先服务器方在服务器初始化时首先创建基于 TCP 的 socket，并等待客户程序发出连接请求。

```
int GsOpenSocket(void)
{
...
    /* Create the socket */
    if((un_sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        return -1;

    /* Bind to any/all local IP addresses */
    memset( &sckt, '\0', sizeof(sckt) );
    sckt.sin_family = AF_INET;
    sckt.sin_len = sizeof(sckt);
    sckt.sin_port = htons(6600);
    sckt.sin_addr.s_addr = INADDR_ANY;
    if(bind(un_sock, (struct sockaddr *) &sckt, SUN_LEN(&sckt)) < 0)
        return -1;

    /* Start listening on the socket: */
    if(listen(un_sock, 5) == -1)
        return -1;
    return 1;
}
```

然后客户端程序调用 GrOpen 向服务器程序发送连接请求。

```
int GrOpen(void)
{
    size_t      size;
    nxOpenReq   req;
    int         tries;
    int         ret = 0;
...
    ACCESS_PER_THREAD_DATA()

    /* 如果已经建立连接，则直接返回*/
    if (nxSocket >= 0)
        return nxSocket;

    /* 创建用于连接 socket*/
    if ((nxSocket = socket(ADDR_FAM, SOCK_STREAM, 0)) == -1)
        return -1;

    /* initialize global critical section lock*/
    LOCK_INIT(&nxGlobalLock);
    ...
    name.sin_family = AF_INET;
    name.sin_port = htons(GR_NUM_SOCKET);   /* AF_INET socket 6600*/
    if (!(he = gethostbyname(sockaddr))) {
        EPRINTF("nxclient: Can't resolve address for server %s\n", sockaddr);
        close(nxSocket);
        nxSocket = -1;
        return -1;
    }
    name.sin_addr = *(struct in_addr *)he->h_addr_list[0];
    size = sizeof(struct sockaddr_in);
    /*
     * 向服务器发送连接请求。如果失败，则重发，最多发送 10 次
     * 每次尝试的时间间隔为 0.1 秒或 2 秒
     */
    for (tries=1; tries<=10; ++tries) {
        struct timespec req;

        ret = connect(nxSocket, (struct sockaddr *) &name, size);
        if (ret >= 0)
            break;
#if ADDR_FAM == AF_INET
        req.tv_sec = 0;
        req.tv_nsec = 100000000L;
#else
        req.tv_sec = 2;
        req.tv_nsec = 0;
#endif
        nanosleep(&req, NULL);
```

```
            EPRINTF("nxclient: retry connect attempt %d\n", tries);
        }
        /*如果连接失败，则错误返回*/
        if (ret == -1) {
            close(nxSocket);
            nxSocket = -1;
            return -1;
        }

        setbuf(stdout, NULL);
        setbuf(stderr, NULL);
        /*连接成功，发送类型为 GrNumOpen 的请求*/
        req.reqType = GrNumOpen;
        req.hilength = 0;
        req.length = sizeof(req);
        /* associate the process ID with the client*/
        req.pid = getpid();

        nxWriteSocket((char *)&req,sizeof(req));
        return nxSocket;
    }
```

当有客户端的请求到来时，服务器方的 GsSelect 函数捕获到这种请求，并调用 GsAcceptClient 处理客户请求。

函数 GsAcceptClient 负责接收请求并进行处理。

```
    void GsAcceptClient(void)
    {
        int i;
    #if ELKS
        struct sockaddr_na sckt;
    #elif __ECOS
        struct sockaddr_in sckt;
    #else
        struct sockaddr_un sckt;
    #endif
        socklen_t size = sizeof(sckt);
    /*接收客户端发来的连接请求，建立客户端与服务器间的连接，i 为客户程序与服务器通信的 socket
描述符*/
        if((i = accept(un_sock, (struct sockaddr *) &sckt, &size)) == -1) {
            EPRINTF("Nano-X: Error accept failed (%d)\n", errno);
            return;
        }
    /* GsAcceptClientFd 函数负责生成客户数据结构，并放置在客户列表中*/
       GsAcceptClientFd(i);
    }

    void GsAcceptClientFd(int i)
    {
        GR_CLIENT *client, *cl;
```

```
/*分配并填写客户的相关数据结构*/
    if(!(client = malloc(sizeof(GR_CLIENT)))) {
        close(-);
        return;
    }
/*客户的 id 用与客户端程序通信的 socket 描述符表示*/
    client->id = i;
    client->eventhead = NULL;
    client->eventtail = NULL;
    /*client->errorevent.type = GR_EVENT_TYPE_NONE;*/
    client->next = NULL;
    client->prev = NULL;
    client->waiting_for_event = FALSE;
    client->shm_cmds = 0;
/*把客户放到客户列表中*/
    if(connectcount++ == 0)
        root_client = client;
    else {
        cl = root_client;
            while(cl->next)
                cl = cl->next;
        client->prev = cl;
        cl->next = client;
    }
}
```

整个建立连接的过程如图 6-8 所示。

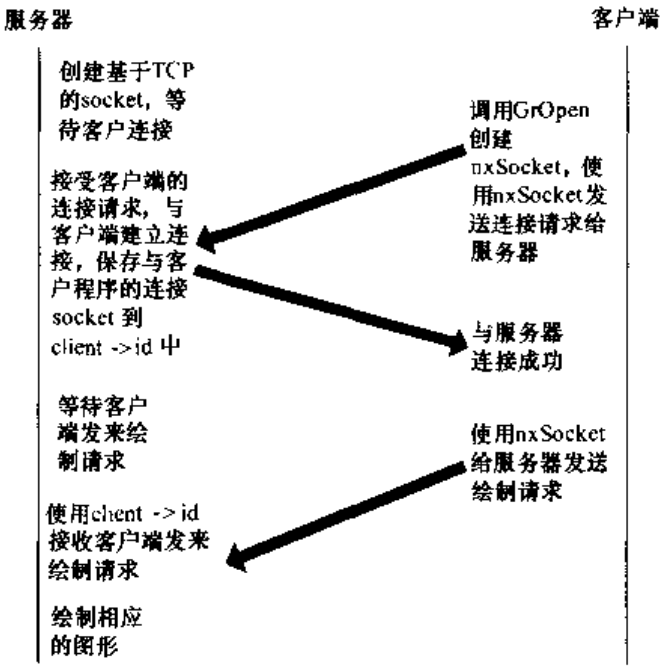


图 6-8　改进前客户端与服务器的连接过程

当客户端与服务连接成功以后，客户端就会利用已连接成功的 TCP 连接发送请求给服务器，而服务器也会利用这一连接发送事件给客户端。

6.4.5.2　客户端与服务器间连接过程改进

这种使用 TCP 连接的通信方式在 VxWorks 下存在着通信速度慢的弊端，这将导致窗口移动及切换时间很长，不能满足窗口移动及切换的要求。为了解决使用 TCP 连接的通信方式在 VxWorks 下存在着通信速度慢的问题，需要改进客户端与服务器端的通信方式。

解决方法是先使用网络建立起一次连接，服务器接收到客户端发来的连接请求之后。在内存中建立起一对虚拟串口 tyMm，然后客户端和服务器就用虚拟串口 tyMm 进行通信，由于虚拟串口不经过网络协议层，没有缓冲区的多次拷贝，因此运行速度比原来使用网络要快，可以满足窗口拖动和窗口切换的需要。

函数 GsAcceptClient 接收客户端程序使用 socket 发来的连接请求。

```
Void GsAcceptClient(void)
{
    int i;
    if((i = accept(un_sock, (struct sockaddr *) &sckt, &size)) == -1) {
        EPRINTF("Nano-X: Error accept failed (%d)\n", errno);
        return;
    }
    GsAcceptClientFd(i);
}

Void GsAcceptClientFd(int i)
{
    int tymmfd0,tymmfd1;
    char tyMmname0[10],tyMmname1[10];

    GR_CLIENT *client, *cl;

    if(!(client = malloc(sizeof(GR_CLIENT)))) {
        close(i);
        return;
    }
    /*创建一对虚拟串口 tyMm*/
    sprintf(tyMmname0,"/tyMm%d",channo);
    tyMmDevCreate(tyMmname0,channo,4096,4096);
    channo++;
    sprintf(tyMmname1,"/tyMm%d",channo);
    tyMmDevCreate(tyMmname1,channo,4096,4096);
    channo++;

    /*打开虚拟串口 tyMm*/
    tymmfd0=open(tyMmname0,2,0);
    tymmfd1=open(tyMmname1,2,0);
    /*客户的 id 用 tymmfd0 表示*/
    client >id = tymmfd0;
    /*将 tymmfd1 的值发给客户端*/
    write(i,(char*)&tymmfd1,2);
    /*关闭与客户端的网络连接*/
    close(i);

    client->eventhead = NULL;
```

```
        client->eventtail = NULL;
        client->next = NULL;
        client->prev = NULL;
        client->waiting_for_event = FALSE;
        client->shm_cmds = 0;

        if(connectcount++ == 0)
            root_client = client;
        else {
            cl = root_client;
                while(cl->next)
                    cl = cl->next;
            client->prev = cl;
            cl->next = client;
        }
    }
```

改进后的连接过程如图 6-9 所示。

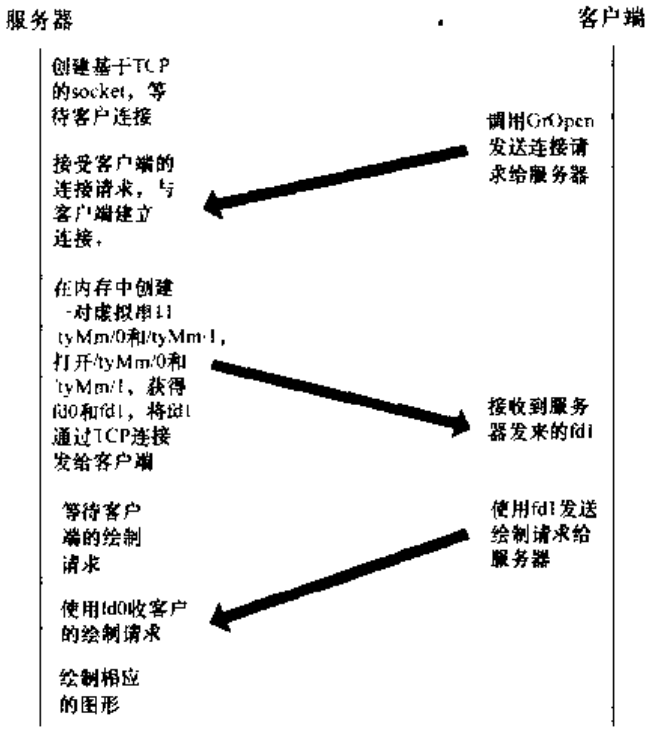


图 6-9　改进后客户端—服务器的连接过程

6.5.5.3　tyMmDrv.c 设计

虚拟串口驱动与真实的串口驱动程序类似，在内存中虚拟了一对使用串口通信方法进行通信的串口。与真实串口不同的是，虚拟串口没有串口的连接线，它们之间的连接线是虚拟的。所以一般是同时创建两个虚拟串口，并且虚拟连接这两个虚拟串口。

文件 tyMmDrv.c 是虚拟串口的驱动程序。

```
#include "vxWorks.h"
#include "iv.h"
#include "ioLib.h"
#include "iosLib.h"
```

```
#include "tyLib.h"
#include "intLib.h"
#include "errnoLib.h"

/* ty_mm_dev 结构，表示虚拟串口*/
struct ty_mm_dev            /* TY_CO_DEV */
    {
    TY_DEV tyDev;
    BOOL created;           /* true if this device has really been created */
    struct ty_mm_dev* pair; //与本虚拟串口连接的另一虚拟串口
    };
typedef  struct ty_mm_dev TY_MM_DEV;
/*系统中最多有 100 对虚拟串口*/
TY_MM_DEV tyMmDv[200];      /* device descriptors */

/*局部函数申明*/
LOCAL int tyMmDrvNum;       /* driver number assigned to this driver */
LOCAL int tyMmOpen (TY_MM_DEV *, char *, int);
LOCAL int tyMmRead (TY_MM_DEV *, char *, int);
LOCAL int tyMmWrite (TY_MM_DEV *, char *, int);
LOCAL int tyMmStartup (TY_MM_DEV *);
```

函数 tyMmDrv 初始化虚拟串口驱动，调用函数 iosDrvInstall 将虚拟串口驱动装载进系统中。

```
STATUS tyMmDrv (void)
    {
    /* 检测驱动是否已经被装载，如果已经装载，则返回*/

    if (tyMmDrvNum > 0)
    return (OK);

    tyMmDrvNum = iosDrvInstall (tyMmOpen, (FUNCPTR) NULL, tyMmOpen,
                (FUNCPTR) NULL, tyMmRead, tyMmWrite, tyIoctl);

    return (tyMmDrvNum == ERROR ? ERROR : OK);
    }
```

函数 tyMmDevCreate 完成虚拟串口设备初始化，并将虚拟串口虚拟地连接起来。

```
STATUS tyMmDevCreate
    (
    char *      name,           /* name to use for this device */
    FAST int    channel,        /* physical channel for this device */
    int         rdBufSize,      /* read buffer size, in bytes */
    int         wrtBufSize      /* write buffer size, in bytes */
    )
    {
    FAST TY_MM_DEV *pTyMmDv;

    if (tyMmDrvNum <= 0)
    {
    errnoSet (S_ioLib_NO_DRIVER);
    return (ERROR);
```

```
    }

    /* if this doesn't represent a valid channel, don't do it */

    if (channel < 0 )
    return (ERROR);

    pTyMmDv = &tyMmDv [channel];

    /* if there is a device already on this channel, don't do it */

    if (pTyMmDv->created)
    return (ERROR);

    /* initialize the ty descriptor */

    if  (tyDevInit  (&pTyMmDv->tyDev,  rdBufSize,  wrtBufSize,  (FUNCPTR)
tyMmStartup) != OK)
    {
    return (ERROR);
    }

    /* mark the device as created, and add the device to the I/O system */

    pTyMmDv->created = TRUE;
    /*将一对虚拟串口建立虚拟连接*/
    if(channel%2==0)
       pTyMmDv->pair=&tyMmDv[channel+1];
    else
       pTyMmDv->pair=&tyMmDv[channel-1];
    /*添加虚拟串口到*/
    return (iosDevAdd (&pTyMmDv->tyDev.devHdr, name, tyMmDrvNum));
    }
```

tyMmOpen：打开虚拟串口设备函数，该函数将设备描述子指针地址返回给上层调用者。

```
LOCAL int tyMmOpen
    (
    TY_MM_DEV *pTyMmDv,
    char      *name,
    int        mode
    )
    {
    return ((int) pTyMmDv);
    }
LOCAL int tyMmRead (TY_MM_DEV *pTyMmDv, char *buffer, int maxbytes)
    {
    return (tyRead ((TY_DEV_ID) pTyMmDv, buffer, maxbytes));
    }
LOCAL int tyMmWrite (TY_MM_DEV *pTyMmDv, char *buffer, int nbytes)
    {
```

```
        return (tyWrite ((TY_DEV_ID) pTyMmDv, buffer, nbytes));
    }
LOCAL int tyMmStartup (TY_MM_DEV *pTyMmDv)
    {
    char outchar;

    /* any character to send ? */
    while (tyITx ((TY_DEV_ID) pTyMmDv, &outchar) == OK)
    {
        if(pTyMmDv->pair->created==TRUE)
            tyIRd ((TY_DEV_ID) pTyMmDv->pair, outchar);
    }
    return (OK);
    }
```

第 7 章　VxCOM/VxDCOM 软件开发

7.1　嵌入式实时系统软件开发的现状

嵌入式实时系统环境下应用软件功能变得越来越复杂，规模变得越来越大，工程化维护、升级特别困难，耗资巨大，增加新的功能往往意味着要冒更大的风险，同时在程序中带来了更多的漏洞。很多应用程序集成了不同开发商的应用程序，效果并不好，操作系统方面也有类似的问题。

传统的软件开发方式是基于过程的，其特点是以数据为中心，即源码的开发方法。如果多个应用软件要重用一个模块，要把该模块的产生的目标代码与应用软件一起链接产生可执行文件，如果该模块要进行升级，需要将每个应用软件同时升级，该模块的耦合性大大增加。嵌入式实时系统中应用软件的一个示意图如图 7-1 所示，有三个应用软件系统均要用到模块 a(a.obj)，当模块 a 升级后，需要三个应用软件重新进行升级。

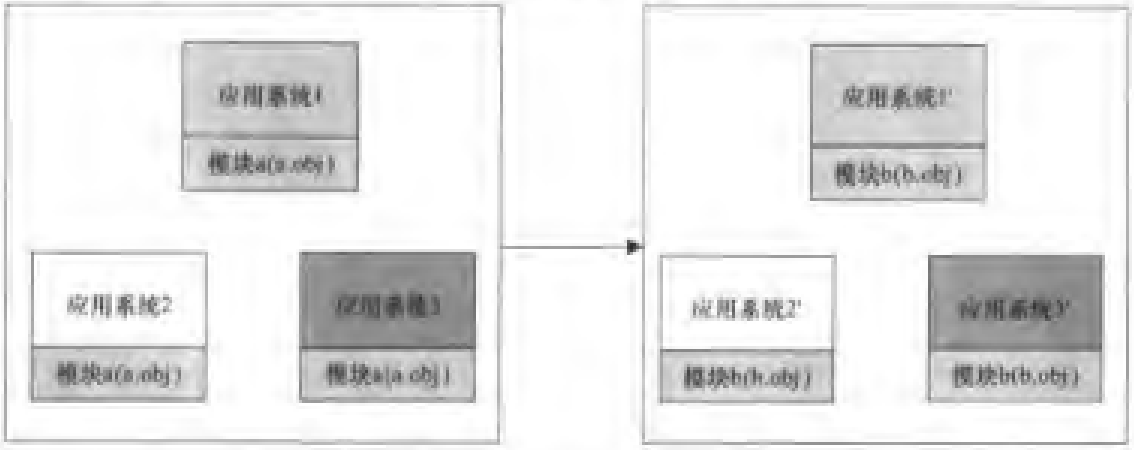


图 7-1　嵌入式系统软件传统开发模式

随着 Windows 操作系统的推广，出现了动态链接库（DLL）的形式，在一定程序上解决了应用软件与公用程序的耦合问题，如图 7-2 所示。

面向对象技术对如上问题提供了满意的方案，类是一组属性（数据元素）和行为（操作函数）的集合，类是源代码级软件复用技术。随着面向对象技术的发展，迫切需要系统能够集成不同开发商用不同编程语言编写的软件，软件的开发可以像硬件一样实现“搭集木”组合。软件构件技术应允而生，在单个实体内封装数据和函数，通过单个引用、指针，使集成开发的应用程序分割成功能模块，构件技术具有即插即用的功能，通过继承（inheritance）、多态（polymorphism）、聚合（aggregation）方式可以充分利用已有对象的功能。软件构件技术是建立在面向对象技术基础上的，构件设计可以突破进程、机器和网络的限制。基于构件的软件开发模式的示意图如图 7-3 所示。

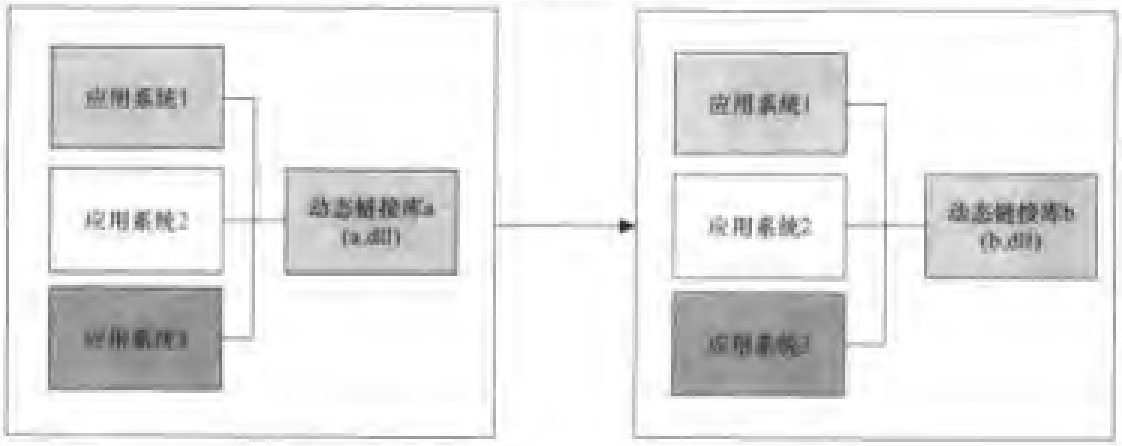


图 7-2　采用动态链接库技术的软件开发方式

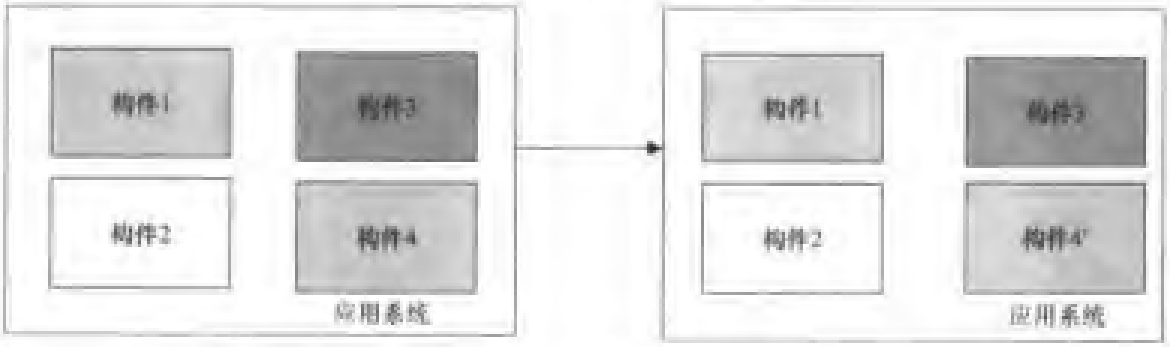


图 7-3　基于构件的软件开发模式

7.1.1　嵌入式系统定义及特点

以应用为中心，以计算机技术为基础，软、硬件可剪裁，适应应用系统对功能、可靠性、成本、体积、功耗严格要求的专用计算机系统称之为嵌入式系统。嵌入式系统具有如下特点：

（1）软件要求能够固态化存储。

（2）软件代码质量高、可靠性高。

（3）满足实时性要求。

（4）运行在多任务操作系统中。

7.1.2　软件构件定义

软件构件是一组软件包，该软件包能够提供某些内聚功能，可作为一个独立单元进行开发和发布，并且可通过预先定义好的接口为用户提供服务。具有如下特点：

（1）编程语言无关性。

（2）位置透明。

（3）二进制规范。

COM（组件对象模型，Component Object Model），是基于构件对象通信的二进制规范。

VxDCOM 在 VxWorks 操作系统上实现了 COM 和分布式 COM（DCOM）的技术，VxDCOM 是指该项技术及其可选的产品的总称，为基本 COM 支持加入了 DCOM 特性。Wind River 的 VxDCOM 技术用于构建嵌入式实时操作系统 VxWorks 下 COM 和 DCOM 组件，能够帮助用

户方便地编写实时嵌入式系统软件下的分布式对象应用程序。

VxDCOM 具有如下主要特性：

（1）无缝 PC 整合。

（2）实时扩展。

（3）实时 DCOM 连线协议。

（4）OLE 自动代理。

（5）支持 OPC 定制接口集。

（6）根据客户环境的高度可配置性。

（7）应用灵活性。

（8）创建可重用对象。

（9）语言独立。

（10）紧凑的 280KB 内存占用。

（11）由运行时库组成（独立的 COM 和 DCOMAPIs）。

（12）目标方头文件。

（13）主机方组件（样例 makefiles、代码生成工具、样例代码）。

（14）文档。

系统需求：

（1）Tornado II。

（2）MSVC 5.0 或 VisualStudio 6.0（用于 MIDL）。

支持的主机：

Windows NT/Windows 2000/Windows XP。

支持的处理器：

ARM、Power PC、X86、MIPS、68K、CPU32、SPARC、i960 和 SimNT 体系。

Wind River 系统的 VxDCOM™实现了 DCOM 标准——面向嵌入世界，在 Tornado™ II 开发台上。使用 VxDCOM，运行 VxWorks®实时操作系统的内嵌设备可以无缝地相互连接，或连接到基于 PC 的管理控制台上。VxDCOM 支持工业标准，并提供分布环境中的灵活的、实时的方案。

COM（组件对象模型）与 CORBA 对象处理标准相似，并且是微软实现许多 Windows 系统接口的方法。例如，ActiveX 组件之间使用 COM 协议直接通信。DCOM（分布组件对象模型）是为操作跨计算机的此对象模型接口的一个自然扩展。

（1）与 PC 控制台无缝集成。

越来越多的内嵌计算产品的用户要求与其他设备，特别是与基于 PC 的管理控制台和基于 Windows 的应用程序和工具的简单的、无缝的整合。在 PC 世界中，DCOM 已经成为对象处理的公认标准。由于 VxWorks 广泛使用在工业中需要非常小、非常快和非常稳定的应用方面，对于 OEM 来说，VxDCOM 将是一个重要的产品。例如，在工业测量和控制工业，符合开放标准的、紧凑的、强健的方案将越来越重要。使用 VxDCOM，制造商可以立即创建与远程 PC 无缝交互的、快速和紧凑的嵌入应用。另外，VxDCOM 允许一个 Windows NT 工作站与生产机器人通过图形化监控包或基于 VxWorks 的传感器互相通信，把数据直接传递到 PC 的电子表单中。VxWorks 设备之间也可以实用标准对象接口通信。

（2）兼容性。

VxDCOM 遵守带有“即插即用”的二进制 COM 协议和 DCOM 连接协议，当向远程系统传输数据类型时可确保兼容性，组件询问通过标准的 IUnknown 接口提供。为了与 VxWorks 目标服务器通信，NT 主机不需要进行任何修改。例如，工业软件生产商已经定义了一套标准接口作为框架进行开放分布式控制。VxDCOM 中的 OPC 支持包括所有数据访问、警报、事件和客户接口集中的常用元素，因此支持对目标所在的 OPC 服务器的快速和有效开发。提供 Windows 平台的用户端工具和开发人员工具箱的销售商确保其应用程序是支持 COM 的。

（3）语言独立。

COM 是一种独立语言标准。一旦创建组件后，它可能影响 Java 语言、Visual Basic、C++ 或由任何可以产生 ActiveX 组件的语言编写的应用程序。语言独立性是通过接口定义语言（IDL）保证的，这种语言可以直接在 Tornado II 中产生。

（4）实时扩展。

线程池用于运行 VxWorks 上的 VxDCOM 任务，它可通过把新线程动态加入线程池来实现对线程活动峰值的处理。为了保证实时性能，需要为任务分配优先权。优先权可以在线程运行前固定，或者在线程运行时修改，以配合客户的优先权。

（5）易实现性。

Tornado II 是直接设计用于创建嵌入应用程序的。由于它与 Tornado II 平台紧密相联，VxDCOM 是实现为嵌入系统创建 COM 对象最容易的途径。

（6）对象重用性。

COM 鼓励代码重用，因为他提供了一个标准的框架，通过这些框架，由不同销售商和不同嵌入部分创建的软件对象之间可以互相通信。VxDCOM 属于此框架，所以允许 VxWorks 组件的再利用。

（7）紧凑内存占用。

VxDCOM 方案针对嵌入世界进行了特定剪裁，可确保大约 280KB 的紧凑内存占用。为了最小化对象尺寸，Tornado 的项目工具允许 VxWorks 在构造时带有对 COM 或 DCOM 的支持。

VxDCOM 体系结构如图 7-4 所示。

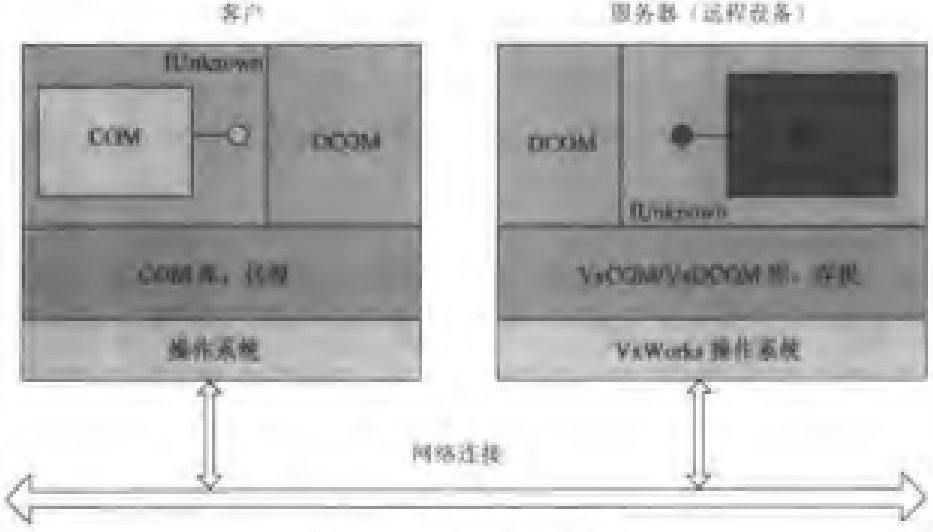


图 7-4　VxDCOM 体系结构图

本节主要参考了 VxWorks 的在线文档，其中大多数为作者在对其资料进行整理的基础上进行归纳，包括如下章节：

（1）VxDCOM 技术简介。

（2）Wind 对象模板库类（WOTL，Wind Object Template Library）。

（3）创建 VxDCOM 应用程序。

7.2 VxDCOM 技术简介

COM 是对象之间通信协议的规范，即 COM 组件。COM 组件是 COM 客户机/服务器应用程序的基本构架，通过 COM 接口向客户端应用程序提供服务。COM 接口是一组方法原型的集合，该集合描述了一个连续、完备定义的功能或者向 COM 客户提供的服务。

7.2.1 COM 组件及软件重用

COM 组件是 COM 类的例示，COM 类定义包括（单个或多个）COM 接口的继承，而 COM 类则需要完成其所派生接口的方法执行。接口本身严格定义其所提供的服务，执行细节则完全封装在 COM 类执行代码的服务器端，客户感觉不到 COM 组件，仅通过 COM 接口交互，这是 COM 技术设计的要点之一，促使软件开发人员可以不考虑服务器端程序更新和重用组件。

1. COM 接口（COM Interface）

接口被命名为纯粹方法原型的集合。接口名表征其方法的功能，通常以大写字母“I”开头。例如：IMalloc 可以代表分配、释放和管理内存的接口。同样，ISem 可以表示封装了信号量（semaphore）功能的接口。作为 COM 技术的一部分，基本的接口服务已定义在 COM 库中。VxDCOM 的 COM 和 DCOM 库执行了基本的接口，需要 VxDCOM 支持 COM 技术的方方面面。这些 COM 和 DCOM 库使用 C++模板类库，同时提供 COM 访问的 API 函数的定义可参见 comLib.h 和 dcomlib.h。

作为开发者，传统的方法是定义用户自己的接口。接口定义必须与 COM 规范一致，包括接口、接口的方法、方法所带参数、返回类型的描述属性和规范。为严格执行该规范，COM 接口成了客户和服务器的约定，可以保证通信协议运行得更好。

接口定义是纯粹原型，与包括纯虚方法的抽象 C++类相似。实际上，Wind 对象模板库（WOTL）用于编写 VxDCOM 的应用程序，正是通过这种方式执行接口。

客户端和服务器的约定是提供服务，但没必要保证服务是如何执行的，执行的细节封装在 COM 类中。

2. COM 类（CoClasse）

COM 类是声明接口和执行接口方法的类。COM 类定义包括支持的所有接口的声明，COM 类执行代码必须执行 COM 类声明的所有接口的全部方法。当 COM 类实例化后，变成了 COM 组件，即服务器。客户应用程序查询所需的接口服务，如果服务器支持这些接口，就可通过接口与客户应用程序建立通信。

3. 接口指针（Interface Pointer）

COM 模型中，在 COM 客户端和服务器之间的通信协议通过指向 COM 接口的指针实现。接口指针在它们之间传递数据。客户端应用程序使用 COM 接口指针查询指定接口的 COM 服

务器，获得访问这些接口的权限，调用接口的方法。由于 COM 规范和通信协议建立在这些指针上，可以认为是二进制标准。

COM 技术使用二进制标准，理论上 COM 组件可以用任何语言编写，运行在任意操作系统上，还能建立通信。用这种方法，COM 技术为软件开发者特别是那些想要移植应用程序到不同操作系统的人提供了极大的灵活性和组件重用。但是目前 VxDCOM 下支持 COM 的语言只有 C 和 C++；对 DCOM 服务器，目前仅支持 C++。

7.2.2　VxDCOM 工具

定义了接口和 COM 类，按照 COM 规范，可以细化和扩充。然而，由于规范遵循标准的准则，VxDCOM 工具可以自动产生大部分的代码来实现这一过程。例如，使用 VxDCOM wizard——comwizard，可简单命名用户接口，从预先定义的列表中选择方法参数类型和属性，comwizard 自动产生正确返回类型的方法，接口和 COM 类定义。当编译应用程序时，Wind IDL（接口定义语言）编译器——widl，产生用于服务器注册和调度（marshaling）的代码，为了开发 COM 应用程序，用户所需的是使用这些工具，编写客户端和服务器的执行代码并编译工程。

7.2.3　VxDCOM 和实时分布式技术

COM 提供了描述软件构件（software components）行为和可访问性的通用框架，将基本 COM 技术突破进程和机器限制扩展到分布式对象需要网络 RPC 协议的支持。DCOM 使用的对象 RPC（ORPC，Object RPC），是微软 DCE-RPC 规范的扩充。ORPC 使用调度接口指针作为 COM 组件之间的通信协议，指定了访问组件接口的方式。COM 最初接口中提供了该项功能，同样在标准 COM 库中定义了这些接口。

支持基本 COM 接口的 VxDCOM 技术，作为标准 VxWorks 的功能的一部分，DCOM 网络协议作为 VxDCOM 可选件的一部分。这些执行记录在 VxDCOM 的 COM 和 DCOM 库，包括与嵌入式系统开发的目标机的 COM 和 DCOM 有关的接口集，同样支持 ORPC。详细描述接口，可参见相应的.idl 文件，例如 vxidl.idl。API 文档可参见 comlib.h 和 dcomlib.h。

VxDCOM 支持进程内和远程服务器模式。可在 VxWorks 目标机上编写服务器程序，为运行在其他 VxWorks 目标机或者运行 Windows NT 的 PC 机的客户程序提供服务。嵌入式智能系统（例如：电信设备、工业控制器和办公设备等）使用 DCOM 协议允许开发人员在不关心网络问题的情况下，扩展 COM 的编程环境到局域网（LAN），甚至 Internet。

7.3　Wind 对 象 模 板 库

Wind 对象模板库（WOTL）是 C++模板类库，设计用来编写 VxDCOM 客户和服务器程序。对应于 ATL（微软 COM 模板类库）模型的基础上，源代码兼容的子集。WOTL 是用户编写客户和服务器程序的架构。

运行 VxDCOM wizard 建立 COM 组件，comwizard 产生输出文件，包括 WOTL 方式下的头文件和执行源文件的框架代码，使用已经产生的框架，用户需完成服务器方的执行细节，还可编写其他的客户端程序。

7.3.1 WOTL 模板类目录

有三种 VxDCOM 模板类用户可以定义使用的 WOTL。这些类型由是否使用 CLSID 来区分，是否是类的一个实例。CLSID 是类的唯一的标识值，允许类通过类厂（类厂是对立在 CLSID 基础上 CoClass 的实例的对象）实现外部实现。三种 WOTL 模板类如下：

（1）纯 COM 模板类（True CoClasses）。

意味着它们拥有注册的 CLSID，通过类厂获得实例。CComCoClass 模板类用于声明这些类，COM 组件使用 VxDCOM wizard 为 CoClass 定义产生框架代码。

（2）单实例模板类（Singleton classes）。

它们也属于 true CoClasses，但仅有一个实例，每次调用 IClassFactory::CreateInstance( )返回相同的实例。欲声明这样的类，在类定义中要使用宏定义 DECLARE_CLASSFACTORY_SINGLETON。

（3）轻量模板类（Lightweight classes）。

它们是简单的类，技术上讲并不是 true COM 类。模板类没有相关的 CLSID（类标识值），通过使用默认的类厂建立机制实例化对象。它们永远不是 DCOM 类，欲声明这样的类，使用 CComObject 类，常用这些类建立类型内部对象以增强用户代码的面向对象功能。

7.3.2 纯 COM 模板类

VxDCOM wizard 为用户 CoClass 自动产生模板类定义，含相应的基类派生，想要的接口以及 CoClass 的实现。

1. CComObjectRoot——IUnknown 实现支持类

CComObjectRoot 是所有 VxDCOM 类的基类，提供任务安全模式下 IUnknown 实现，另外支持聚合对象。声明这样的类提供一个或者多个 COM 接口的“坚固”执行，该类必须继承 CComObjectRoot 及执行的接口。下例声明了一个 CExample 类，支持 IExample 接口，实现其方法。

```
class CExample: public CComObjectRoot, public IExample
{
        // Private instance data
    public:
        // Implementation, including IExample methods...
};
```

作为 IUnknown 类的实现，所有声明 WOTL 实现类必须包括 CComObjectRoot 作为最初的基类。

2. CComCoClass——CoClass 类模板

CComCoClass 提供 CoClass 执行的模板类，即包括一个众所周知的类 CLSID 和接口。从该类创建的对象认为是真正的 COM 对象，通过使用 CLSID 和调用 COM/DCOM API 函数 CoCreateInstance( )和 CoCreateInstanceEx( )进行外部实例化。

CComCoClass 包装了类厂类和注册机制所需的功能，从而派生类可以继承该项功能。CComClassFactory 是类厂类模板执行标准的 IClassFactory COM 接口。该接口允许创建的对象在运行时使用 CLSID。声明这样的类遵循标准的 WOTL 格式，继承 CComObjectRoot 及执

行的接口。

3. CoClass 定义

CoClass 定义由 wizard 自动产生，派生的 CoClass 如下：

（1）最初的 WOTL 基类，CComObjectRoot。

（2）所有 CoClass 的 WOTL 基类模板，CComCoClass。

（3）原始接口，由 CoClass 执行。

（4）由 CoClass 执行的其他所有接口。

例子定义：

下例定义了 CoMathDemoImpl.h 头文件的继承 CoClass 的定义。

```
class CoMathDemoImpl
    : public CComObjectRoot,
      public CComCoClass<CoMathDemoImpl, &CLSID_CoMathDemo>
     ,public IMathDemo
     ,public IEvalDemo
{
       // 定义体部分
};
```

为 CoClass 产生的服务器应用程序执行头文件简单派生于 CComObjectRoot、CComCoClass、用户最初接口、其他接口。WOTL 支持多重继承，包括继承多个接口，也从执行的接口派生用户定义的 CoClass。

4. 使用 CLSID 实例

CComCoClass 类模板建立基于 CoClass 和 CLSID 的类实例，上例中&CLSID_CoMathDemo 参数，代表 GUID（全局唯一标识符，Globally unique identifier）用来标识 CoClass。用户 CoClass 的 CLSID 为 CLSID_basename，可在由 widl 编译产生的 basename_i.c 文件中找到。CLSID_basename（类 ID）被声明为常量，用在服务方的头文件和执行文件中，也可用在客户端的执行文件中。

7.3.3 轻量模板类

Lightweight Class 建立 COM 对象时无需 CLSID，因为这一点，它们被认为不是真正意义上的 CoClass。轻量类主要内部使用，用以优化程序面向对象设计的接口实现。

对 WOTL 而言，CComObject 是轻量对象类模板。CComObject 是一包装模板类，用于建立轻量对象类。真正的执行类使用由 CComObjectRoot 派生的 CComObject 类的一个参数模板化，目的是继承 IUnknown 接口支持。

下面是 CComObject 模板类实例，CExample 上例中已经定义，它从 CComObjectRoot 派生，执行 IExample 接口：CComObject<CExample>。

轻量类没有 CLSID，因此不能使用正常类厂方法使外部实例化。基于此，CComObject 提供了默认的类厂执行。这些类通过调用函数 CComObject<CExample>::CreateInstance()可以实例化，而不用调用函数 CoCreateInstance()，函数的入口参数同 IClassFactory::CreateInstance()。VxDCOM wizard 不能为这类模板类产生定义，欲创建轻量类对象，用户需在头文件和源文件中简要加入定义和执行代码。

7.3.4 单实例类宏

定义一个类为 singleton，即一个类仅有一个实例，必须在类定义中包括 DECLARE_CLASSFACTORY_SINGLETON 语句。例如：

```
class CExample
    : public CComObjectRoot,
      public CComCoClass<CExample, &CLSID_Example>,
      public IExample
{
        // Private instance data
    public:
        DECLARE_CLASSFACTORY_SINGLETON();
        // Implementation, including IExample methods...
};
```

使用该宏声明一个类后，所有调用 IClassFactory::CreateInstance()函数均返回同一实例。VxDCOM wizard 也不为这样的模板类产生定义，欲创建单实例类对象，用户需在头文件和源文件中简要加入定义和执行代码。

7.3.5 Wind 对象模板库（WOTL）

WOTL 可以自动产生的 WOTL 框架代码。Wind IDL 编译器和命令行选项则有：

（1）自动产生文件中 IDL 定义的结构及意义。

（2）VxWorks 下支持 DCOM 的配置参数。

（3）实时扩展和 OPC 接口。

（4）编写执行代码的提示和例子。

（5）VxDCOM 和 ATL 的比较。

7.4 创建 VxDCOM 应用程序

7.4.1 简介

本节主要描述如何在 Tornado2.2 开发环境下，一步一步建立和编译 VxDCOM 应用程序的过程。首先介绍了配置操作系统支持 VxCOM/VxDCOM,然后使用 Tornado 提供的应用程序向导产生服务方和客户方源程序代码，最后介绍如何构建工程文件产生执行代码。

为了更加方便地建立 VxDCOM 程序，Tornado 提供了基本的 VxDCOM 支持工具，主要包括：

（1）应用程序向导。

无需在接口定义语言（IDL）中定义 COM 类和接口的情况下，可让用户方便产生基本 VxDCOM 程序的框架代码。类似 Visual C++5.0 中的提供了许多 AppWizard，主要为用户产生框架代码。

（2）IDL 编译器。

编译 IDL 文件，产生必要的存根（proxy）/代理（stub）代码和 VxDCOM 所需的头文件。

（3）C++模板类库。

方便编写客户端和服务器方执行代码。

7.4.2　VxDCOM 开发过程

VxDCOM 开发的基本流程如图 7-5 所示。

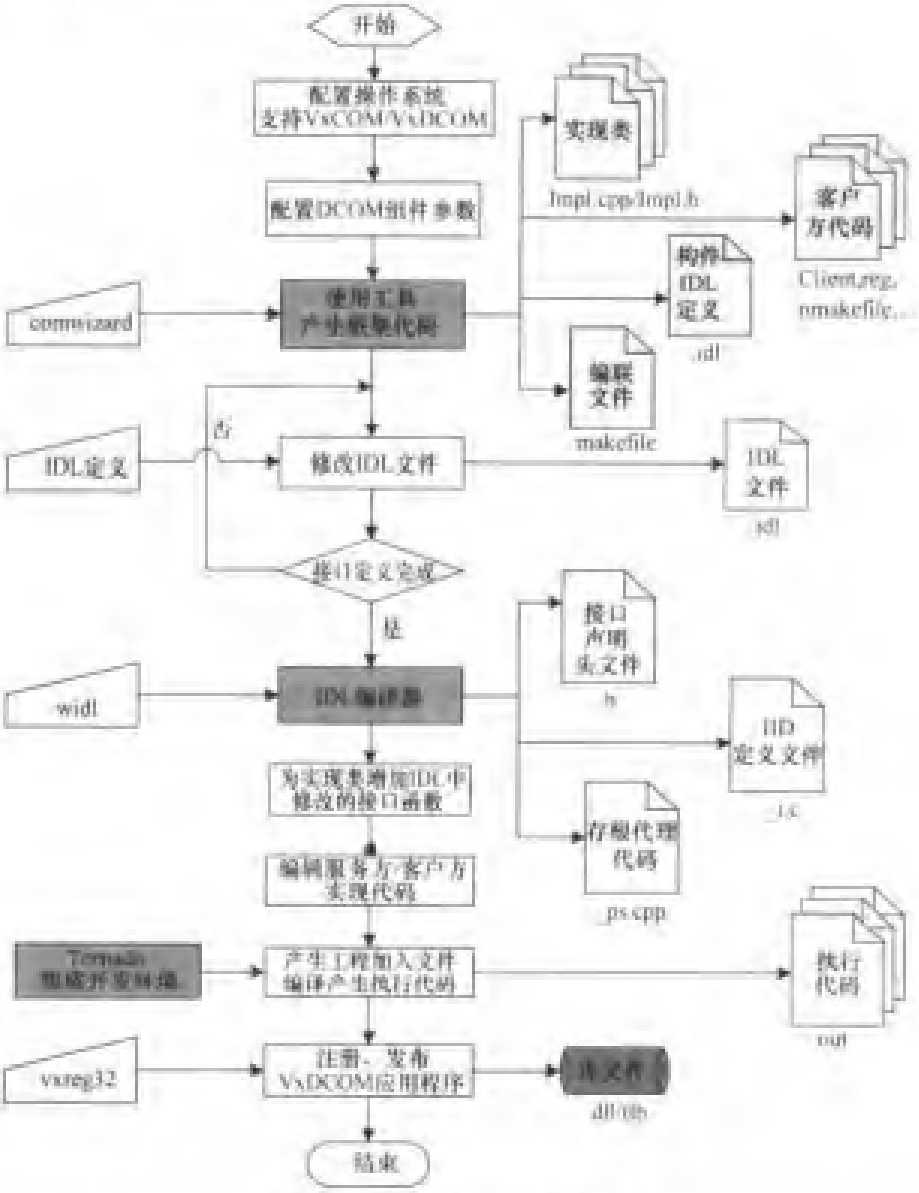


图 7-5　VxDCOM 构件开发流程

VxWorks 环境下的 VxDCOM 客户和服务器程序可以创建引导程序或者是下载到目标机的程序，总的步骤如下：

（1）建立支持 VxDCOM 的可引导镜象文件。

（2）配置 DCOM 组件参数。

（3）用 VxDCOM 应用程序向导产生框架工程文件。

（4）根据用户要求更改接口定义头文件，使用 IDL 编译器产生代理/存根代码及声明文件。

（5）完成服务器和客户端实现程序编写。

（6）在工程文件中加入文件并编译产生目标代码。

（7）注册并发布 VxDCOM 应用程序。

7.4.3 配置 VxDCOM 可引导镜象

无论是创建可引导操作系统，或者是开发客户方或者服务器方下载程序，首先需要一个内核能够支持 VxDCOM 技术的 VxWorks 引导镜像。所有 VxDCOM 程序都需要这样一个映像文件，创建可引导程序时，应向可引导系统映像中加入 VxDCOM 应用程序文件。建立下载程序时，将 VxDCOM 文件加入到可下载模块中，然后下载模块到支持 VxDCOM 的引导系统中。

启动 Tornado，建立一个可引导 VxWorks 操作系统映像，配置的操作系统工程名称为 jarios，目标机开发环境为 Pentium。工程创建成功后，点击工作空间 VxWorks 表单，用户会看到初始并没有包括 VxCOM/VXDCOM 的操作系统选项，如图 7-6 所示。

图 7-6 VxWorks 操作系统的初始选项

7.4.3.1 增加 VxCOM 组件支持

单击 VxCOM 组件的鼠标右键，选择“Include ‘VxCOM(Component Object Model)’”，会弹出配置 VxCOM 的对话框，如图 7-7 所示。VxCOM 组件主要包括：

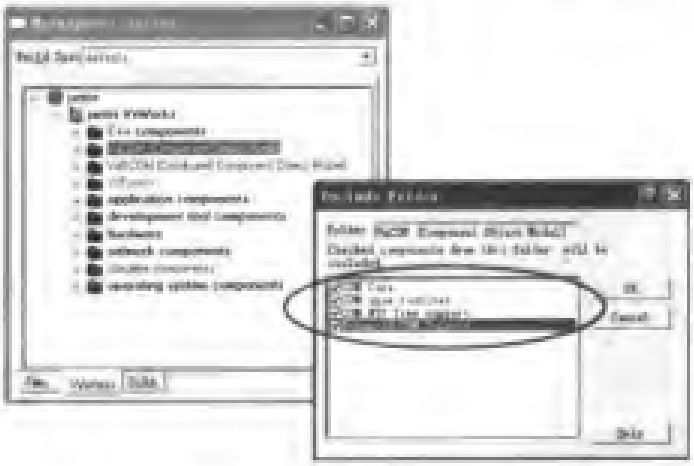


图 7-7 操作系统下加入 VxCOM 组件对话框

（1）COM 核组件。

也称做 COM_CORE，支持 C COM 工程和 C++ COM 及 DCOM 工程。为必选项。

（2）COM 支持组件。

COM 支持组件针对所有 C++ COM 和 DCOM 程序。为可选项。

（3）ComShow 函数支持。

COM_SHOW 支持组件在使用 COM 显示函数时有用，包括该组件为 VxWorks 运行的注册代理提供了诊断函数。为可选项。

选定配置的选项，按 OK 会弹出一个关于代码空间的对话框，再按 OK 键完成 VxCOM 组件的配置，如图 7-8 所示。

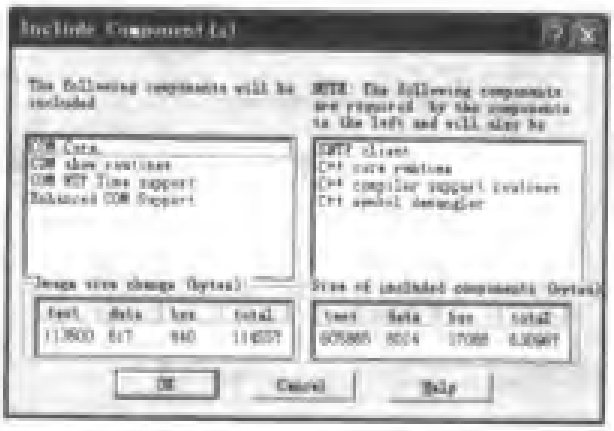


图 7-8　包括 VxCOM 组件的代码空间对话框

设定 VxCOM 的参数。单击 VxCOM 组件的鼠标右键，选择“Params for'VxCOM(Component Object Model)'”就会弹出配置 VxCOM 参数的对话框，如图 7-9 所示。

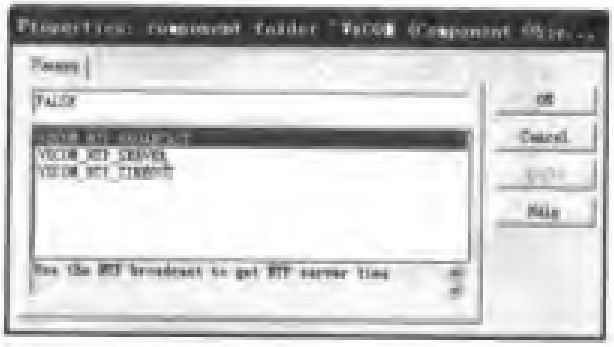


图 7-9　VxCOM 参数设定

主要包括 3 个选项，这 3 个选项为可选选项，使用网络时间协议（NTP）可将本机网络协议通信时间对准到系统时间，仅在“COM NTP Time support”有效时才用。

（1）VXCOM_NTP_BROADCAST。

使用 NTP 广播方式得到服务器系统时间。

（2）VXCOM_NTP_SERVER。

在使用广播方式时服务器是否使用或者忽略接收到的系统时间，布尔型变量：是（TRUE），忽略（FALSE）。

（3）VXCOM_NTP_TIMEOUT。

NTP 超时设置，默认为 5 秒。

7.4.3.1 增加 VxDCOM 组件支持

加入相应的 VxDCOM 支持组件，某些组件适用于所有 VxDCOM 程序，另外一些组件仅适用于 DCOM，还有一些组件是可选组件，提供不同功能。

本节描述 VxDCOM 支持以及何时加入内核中去。

单击 VxDCOM 组件选项，选择“Include 'VxDCOM(Distributed Component Object Model)'”选项，如图 7-10 所示。

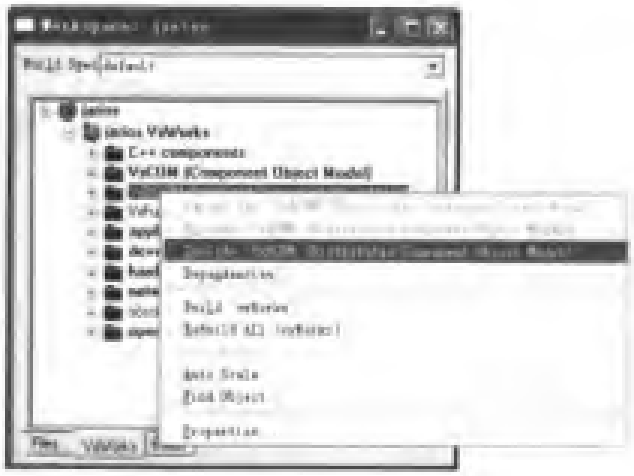


图 7-10 操作系统加入 VxDCOM 组件

弹出对话框，可选定 VxDCOM 选项，如图 7-11 所示。

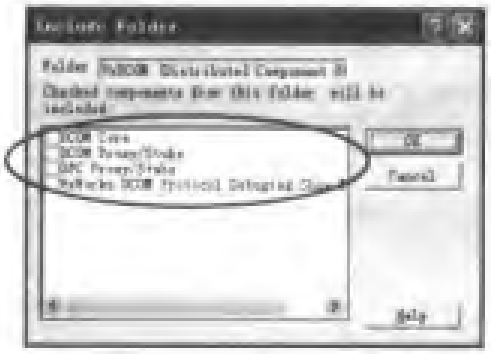


图 7-11 VxDCOM 组件选项

（1）DCOM 核组件。

DCOM 核对 DCOM 程序是必须的，DCOM 组件提供了分布式 COM 的支持。

（2）DCOM 支持组件。

DCOM_PROXY 支持组件对 DCOM 程序是必须的，DCOM 组件提供了分布式 COM 的支持，DCOM_PROXY 组件提供存根(proxy)/代理(stub)代码的支持，DCOM 核与 DCOM_PROXY 组件提供了对基本 COM 的支持。

（3）OPC 程序支持。

DCOM_OPC 支持组件在编写自己的 OPC 服务器是非常有用，如果使用 VxOPC 产品就没有必要了。

（4）DCOMShow 函数支持。

DCOM_SHOW 组件在调试 VxDCOM 协议时有用，它作为 Wind River 客户支持提供调试信息，主要是客户在访问远程服务器时在服务器端显示网络通信的协议内容，建议用户最好不要包装到最终的产品中。

按 OK 键后弹出 VxDCOM 代码的详细情况，如图 7-12 所示。按 OK 键可把 VxDCOM 组件加入到操作系统中去。

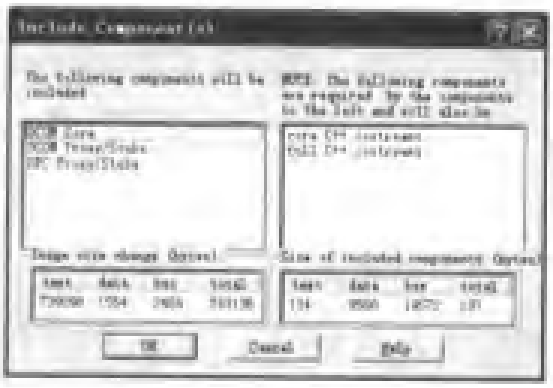


图 7-12　VxDCOM 组件代码对话框

7.4.3.2　配置 DCOM 参数

如果程序中包括 DCOM，可为该组件加入配置参数，方法是选择“Params'VxDCOM (Distributed Component Object Model)'”选项，如图 7-13 所示。

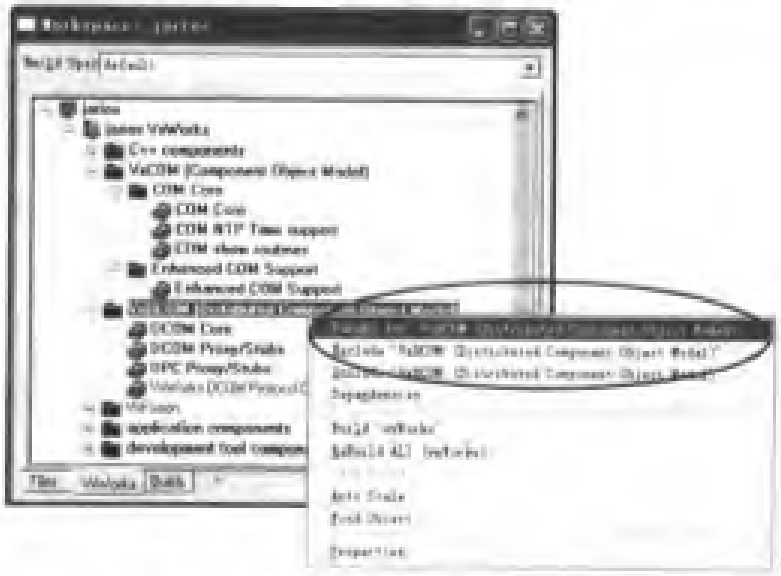


图 7-13　配置 VxDCOM 参数的弹出菜单

会弹出 VxDCOM 所用到一些参数的设置情况，用户可以修改这些参数，以满足系统的要求，如图 7-14 所示。

DCOM 的参数主要有 9 个，主要进行安全验证、优先级配置以及任务分配等。下面详细介绍。

（1）VXDCOM_AUTHN_LEVEL。

指定服务器方组件所需的验证级别。通常 Windows 平台有 7 个不同值，范围为 0~7，其含义如表 7-1 所示。通常情况下，数字越高，表示其验证级别越严格。

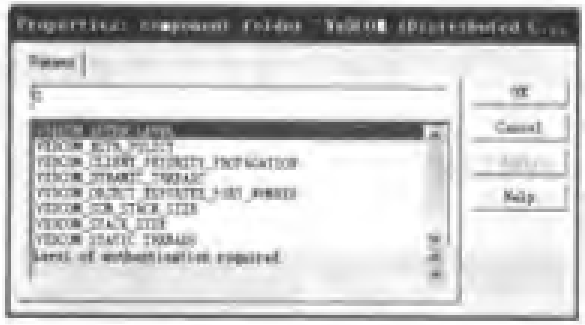


图 7-14　配置 VxDCOM 参数的对话框

表 7-1　远程访问验证级别说明

值	身份验证级别	标　志	含　义
0	默认	RPC_C_AUTHN_LEVEL_DEFAULT	使用正常的安全协商算法
1	无	RPC_C_AUTHN_LEVEL_NONE	无验证级别
2	连接	RPC_C_AUTHN_LEVEL_CONNECT	客户与服务器建立连接时需要客户验证
3	调用	RPC_C_AUTHN_LEVEL_CALL	每一次进行远程调用前进行客户验证
4	数据包	RPC_C_AUTHN_LEVEL_PKT	验证所有来自期望客户的数据
5	数据包完整性	RPC_C_AUTHN_LEVEL_PKT_INTEGRITY	验证服务器客户之间传送过程中数据是否有被更改过
6	数据包保密性	RPC_C_AUTHN_LEVEL_PKT_PRIVACY	对以上所有级别进行验证，同时对每一次远程调用参数值进行加密处理

为了与 Windows 平台兼容，VxWorks 也定义了 7 个值，宏定义在 dcomlib.h 中，但仅使用了 0～2，即：

1）0：默认，不需要验证。

2）1：不需要验证级别。

3)2:RPC_C_AUTHN_LEVEL_CONNECT，表示用户需要使用 API 函数 vxdcomUserAdd( ) 创建一个用户 ID 和密码组合，在登录到服务器时需要进行验证。

默认值：0。

（2）VXDCOM_BSTR_POLICY。

设定 BSTR 字符列集的配置属性。布尔型值，其含义为：

TRUE：BSTRs 按照可计数的字符串（字节顺序）进行列集，第 1 个字节表示字符串的长度，余下字节代表 ASCII 字符。

FALSE：按照 BSTR 进行列集，采用宽字符串形式，使用两个字节的 unicode。

默认值：FALSE。

（3）VXDCOM_CLIENT_PRIORITY_PROPAGATION。

为基本 DCOM 功能增加优先级模式和实时扩充优先配置。布尔型值，其含义为：

TRUE：使用客户的优先级，能够使服务器与客房运行在相同的优先级。

FALSE：服务器使用自己的优先级，与客户优先级独立。

默认值：TRUE。

（4）VXDCOM_DYNAMIC_THREADS。

指定服务器方动态线程的数目。范围为 0～32。

默认值：30。

（5）VXDCOM_OBJECT_EXPORTER_PORT_NUMBER。

系统重新启动后，配置验证 ObjectExporter 的端口号，如果该值为 0，表示是可以动态分配端口号。其值建立范围为 1025～65535。

默认值：65000。

（6）VXDCOM_SCM_STACK_SIZE。

设定编译时服务控制管理器（SCM）任务（tScmTask）的堆栈大小，该参数主要用于配置 PPC60x 体系的目标机系统，相比其他目标机系统而言，可以创建较大的堆栈空间，大多数目标机系统，可使用默认值。如果 tSCM 任务（tScmTask）出现堆栈或者数据异常，可以使用 browser 检查堆栈是否用完，如果是需要增大该值。服务控制管理器用于访问许可方面的控制。

默认值：30K。

（7）VXDCOM_STACK_SIZE。

指定服务器方线程池中线程堆栈大小，即每个 tComTask 堆栈大小。建议值范围：16K～128K。如果 tComTask 出现堆栈或者数据异常，可以使用 browser 检查堆栈是否用完，如果是需要增大该值。

默认值：16K。

（8）VXDCOM_STATIC_THREADS。

指定服务器方线程池中预先分配的线程数目，即系统在重新启动后创建任务 tComTask 的数目，范围为 1～32。VxDCOM 启动几个初始化线程以加速 COM 类运行时间，其大小可以设定为系统中正在运行 COM 类的平均数。

默认值：5。

（9）VXDCOM_THREAD_PRIORITY。

指定服务器线程池中线程优先级，其任务 tComTask 的优先级，其范围与 VxWorks 任务优先级范围相同。

默认值：150。

配置完操作系统，编译并联接成 VxWorks 操作系统目标代码，并产生 bootrom 引导程序，引导成功后，打开 jarios 的 shell，可以看到操作系统下增加了两类任务，1 个 tScmTask，多个 tComTask 任务（默认为 5 个，可以根据用户参数进行修改），如图 7-15 所示。

至此，我们已经完成了操作系统的配置，使其能够支持 VxCOM/VxDCOM 选项。

7.4.4　使用 VxDCOM 向导

VxDCOM 向导也称为 VxDCOM wizard，能够让用户产生完整的 VxDCOM 工程，或者导入现有文件到一个新工程中。在运行 wizard 前，须在命令行运行 Tornado 开发环境所提供的批处理文件 torvars.bat，设定 tornado 的相关参数，并把 tornado 执行文件目录加入到系统路径中。该文件主要信息如下：

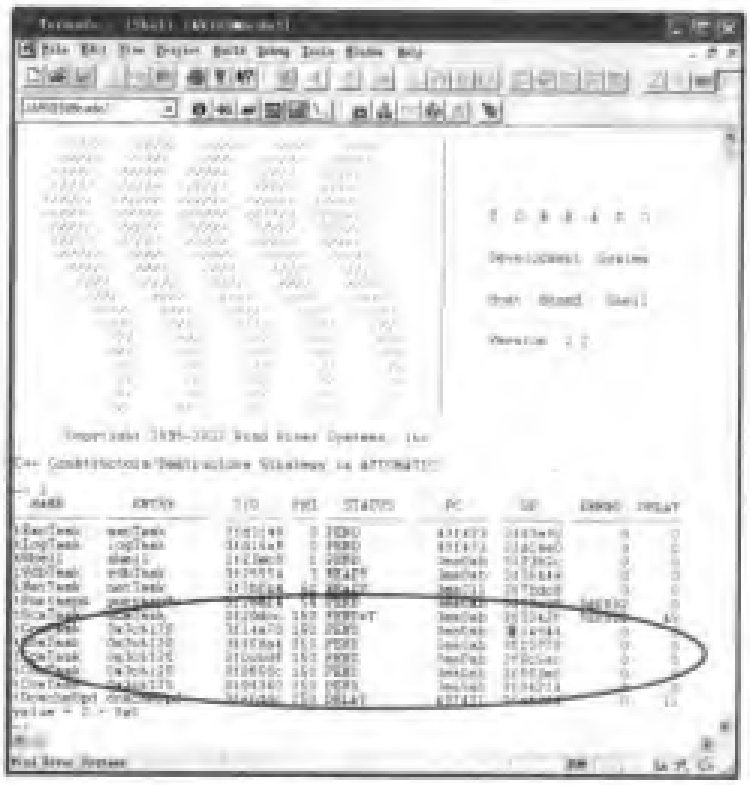


图 7-15　成功配置 COM 组件的 shell

```
installDir/host/hostType/bin/torVars.bat
```

对 X86 平台，上述的路径为“tornado2.2/host/X86-win32/bin”。

```
comwizard projDir
```

其中，comwizard 是批处理文件，projDir 是用户工程的目录，由 wizard 产生的文件放在该目录下，目录名作为 CoClass 新工程的默认名，可以在 wizard 中修改它。

创建新工程时，需要指定服务器类型和编程语言，使用 wizard 图形用户界面方式定义用户 CoClass 和接口，用户不必编写 IDL，也不必编写 proxy/stub 代码。输入信息后，wizard 产生最原始的 IDL 定义文件和其他与服务器模式和客户端程序有关的文件。

下面我们以 COM 库 dcomdemo 为例，说明如何 comwizard 的使用方法。dcomdemo 有两个接口 IMath 和 IAccount，如图 7-16 所示。

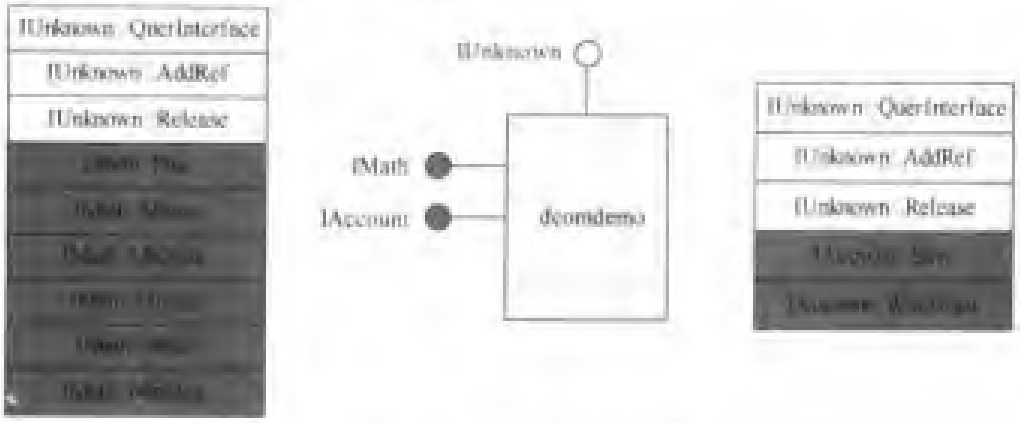


图 7-16　dcomdemo 库的两个接口

7.4.4.1　建立 COM/DCOM 框架工程

7.4.4.2　选择工程类型

VxDCOM 首页显示的选择工程类型是建立 COM/DCOM 框架工程，还是导入现有文件到新工程中，如图 7-17 所示。

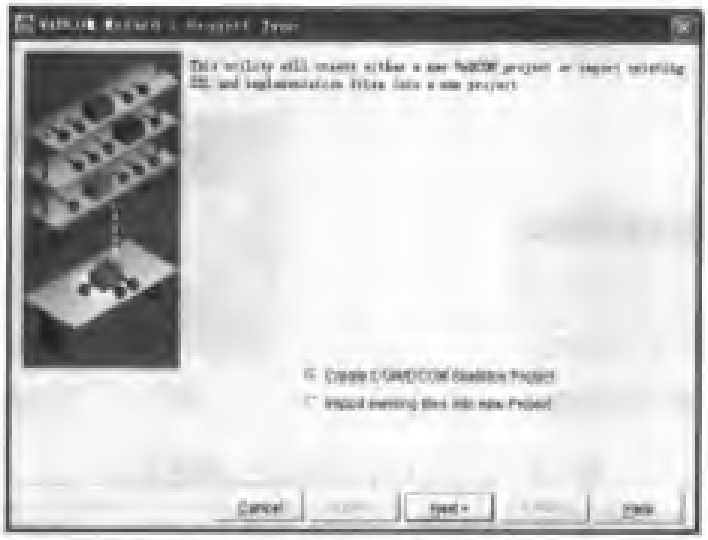


图 7-17　选择工程类型

导入现有文件到新工程中，主要导入的是接口定义文件和实现类的声明及实现源程序。选择“创建 COM/DCOM 框架工程”，并单击“next”按钮。

7.4.4.3　定义 COM 类

图 7-18 是 COM 类对话框，从该对话框定义 COM 类，并为 COM 类增加 1 到多个接口方法，为每个接口增加多个方法，默认的 COM 类名是用户传给 comwizard 的参数，如果用户想改变 COM 类的类名，可以单击 Edit 按钮进行修正。

图 7-18　COM 类对话框

欲定义 COM 类，需要增加接口和接口方法，指定这些方法的参数类型，常规步骤如下：

（1）为 COM 类添加一个或多个接口（见图 7-19）。

（2）为每一个接口添加一个或多个方法。

（3）为每一个方法添加一个或多个参数，并指定每个参数类型和属性。

定义完 COM 类后，单击 Next 按钮，为 COM 类添加接口项，高亮度所选项，并选择 Add 菜单，弹出 Add 对话框，该对话框让用户输入新接口的名字、接口方法和接口方法参数。

当添加接口方法参数时，必须指定每一参数的属性和类型，默认参数的属性和类型已经在编辑框中，可从下拉列表中选择正确的选项，按下 Apply 后，用户的设定就显示在 CoClass 对话框中，如图 7-20 所示。

图 7-19　为 COM 类添加接口

图 7-20　为方法添加参数

表 7-2　VxCOM/VxDCOM 默认参数类型

BOOL	BOOL*	FLOAT	FLOAT*	SHORT	SHORT*
BSTR	BSTR*	HRESULT	HRESULT*	UINT	UINT*
BYTE	BYTE*	INT	INT*	ULONG	ULONG*
CHAR	CHAR*	IUnknown*	IUnknown**	USHORT	USHORT*
DOUBLE	DOUBLE*	LONG	LONG*	VARIANT	VARIANT*
DWORD	DWORD*	LPWSTR	LPWSTR*	WORD	WORD*

基本数据类型定义在 target\h\comCoreTypes.h 头文件中。

方法参数必须指明其方向属性，主要有：

（1）in。

（2）out。

（3）in，out。

（4）out，retval。

方法参数属性中，所有“out”参数及“out”参数组合必须是指针型的。

目前 comwizard 仅支持自动数据类型，对于非自动数据类型，即用户自定义的数据类型，需要用户手工修改 IDL 文件和执行文件。我们将在第 6 章说明如何手工增加这些操作，这里描述的全部为自动数据类型。

定义好的 COM 类如图 7-21 所示。

7.4.4.4　设定 COM 类选项

一旦 COM 类定义完成后，按“next”按钮弹出设定 COM 类选项的对话框，用以指定 COM 类服务器模型、编程语言、其他可选的客户程序的框架代码，如图 7-22 所示。

（1）服务器模型。

选择服务器类型是 COM 还是 DCOM。

COM—COM 服务器模型在 VxWorks 系统全部使用 COM 技术。在同样的 VxWorks 目标机上，COM 服务器受限于 COM C 或者 C++的通信方式，使用 COM 组件设计是基于内部使用

COM 接口的面向对象代码。

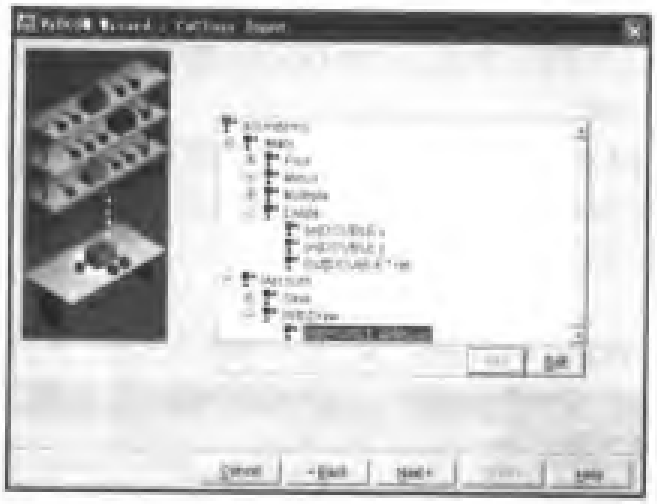


图 7-21　使用 comwizard 定义的用户接口

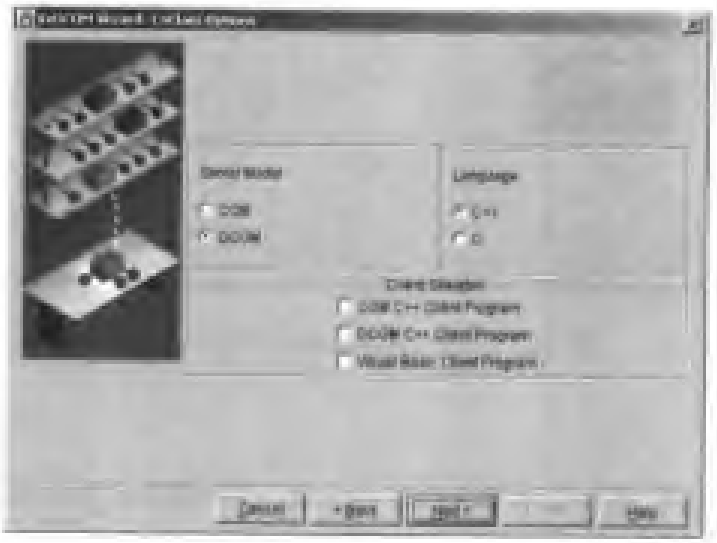


图 7-22　设定 COM 类选项

DCOM—DCOM 服务器模型用于基于分布式组件的系统（分布式 COM）。DCOM 扩充了基本 COM 技术。例如，可用 VxDCOM 服务器连接网络上一个分布对象的 PC 机。

（2）语言。

选择执行服务器 CoClass 的语言：

1）C++——该语言可用于 COM 或者 DCOM 程序。

2）C——仅用于 COM，不适合 DCOM 程序。

（3）客户端框架程序。

依赖于服务器模型，用户可以从用户程序类型中选择。如果有一个 COM 服务器，则仅有一个 C++ COM 客户端工程；如果有一个 DCOM 服务器，可以选择一个或者多个任意类型的客户端程序。客户端程序是可选的。

1）C++ COM 客户端程序——客户端必须与 VxWorks COM 服务器同时存在且位于同一 VxWorks 通信服务器内。

2）C++ DCOM 客户端程序——DCOM 客户可驻留 VxWorks 目标机上或者运行 Windows NT 的 PC 机上，通信服务器必须是 VxWorks 目标服务器，当客户方是 PC 或者其他目标机上，须使用 DCE 网络层；当客户方是目标机，尽管 DCOM 调用完成，它还要检测客户方是否位于同一主机，此时则使用 COM。

3）Visual Basic DCOM 客户端程序——VB 客户端程序（例如：Excel Basic，Word Basic，Access Basic 等）必须驻留在运行 Windows NT 的 PC 机上，客户方用 Visual Basic 编写，通信服务器则是 DCOM VxWorks 目标机服务器。

7.4.4.5 产生框架文件

最后一个对话框——Project Creation，按 Finish 键产生工程，如图 7-23 所示。

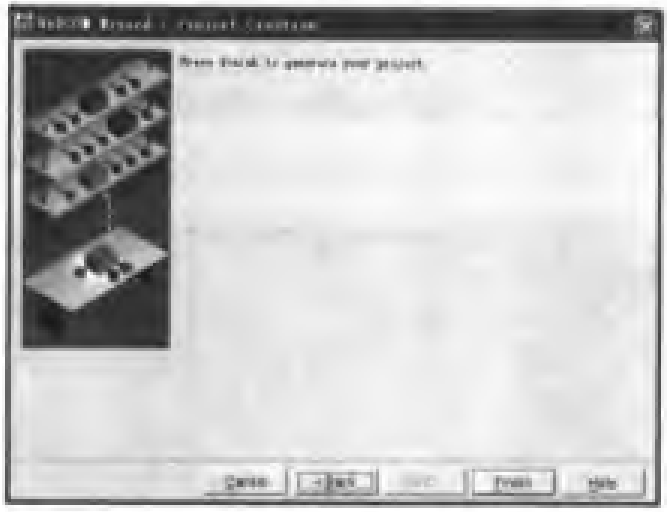


图 7-23 完成 comwizard 人机交互过程

7.4.5 产生的输出

wizard 产生输出文件，放在工程的几个子目录内，这与用户的选项有关。我们以 component 构件为例进行说明。

（1）输出目录。

当 wizard 完成运行后，产生输出文件，建立 2 个目录，分别为：

1）component。

2）component/client。

（2）工程文件。

使用这些文件可以编写、编译、编辑代码等。

Makefile：用于从命令行编译工程，编译下载模块时的默认方法。

component.idl：IDL 文件，由文件名.idl 标识，当编译 Tornado 工程时，该文件自动由 widl 编译。

componentImpl.cpp：服务器 COM 类执行文件，由 Impl.cpp 扩展名标识，是执行服务器接口方法的文件。

componentImpl.h：服务器 COM 类执行文件头文件，由 Impl.h 扩展名标识，是执行服务器接口方法类说明的文件。

Client/componentDCOMClient.cpp：DCOM 客户端程序的框架执行文件，由 DCOMClient.cpp 扩展名标识，编辑该文件可以加入客户端代码，激活 DCOM 服务器。

Client/component.rgs（仅 DCOM）：注册表文件，由.rgs 扩展名标识，使用 VxDCOM 工具注册 COM 类，当选择 DCOM 服务器时产生，编写 DCOM 程序时可以编辑该文件。

Client/nmakefile(仅 C++客户方)：这是主机方（客户方）的 makefile，使用(nmake)为 Win32 或者 VxWorks 建立 MFC 客户端应用程序。

（3）服务器输出文件。

COM 和 DCOM 服务器文件由于调度的不同，仅在 proxy/stub 代码中不同。

（4）COM 服务器输出。

1）component.h。

2）component.idl。

3）componentImpl.cpp。

4）componentImpl.h。

5）component_i.c。

（5）DCOM 服务器输出。

1）component.h。

2）component.idl。

3）componentImpl.cpp。

4）componentImpl.h。

5）component_i.c。

6）component_ps.cpp。

在 component/client 目录下的文件

component.rgs

服务器端的一些文件初始化长度为 0（例如：component_i.c、component_ps.cpp 和 component.h），必须运行 widl.bat 为这些文件产生代码，widl 的帮助如下：

```
WIDL Version 2.2.1 Copyright (c) 2000 Wind River Systems, Inc.
Parser generated by ANTLR Parser Generator Version 2.7.0 1989-1999 MageLang Institute

usage: widl [<options>] -I<include-dir> inputfile

  <options> ::= <option> | <option> [<options>]

   <option> ::= -noh | -h <headerDir> |
                -nops | -ps <psDir> |
                -noi | -i <guidDir> |
                -x | -dep | -b <backendClass>[:<arg>]
```

相关的选项说明：

1）inputFile：IDL 源文件的名字或者是一类型库。

2）options：可选参数，是如下参数的组合。

-I<include-dir>：使用外部 include 目录。

-noh：不产生头文件。

-h<headerDir>：在 headDir 目录下产生头文件。

-nops：不生代理/存根代码。

-ps <psDir>：在 psDir 目录下产生代理/存根代码。

-noi：不产生_i.c 文件。

-i<guidDir>：在<guidDir>目录下产生包括 GUID、CLSID 标识的_i.c 文件。

-dep：产生 GNU 风格 dependencies。

（6）客户端输出文件。

所有客户端程序均产生在目录 client 下，对于 DCOM 客户端程序而言，component.rgs 主要是进行类型库注册用。

（7）COM 客户端输出。

```
componentClient.cpp
```

（8）DCOM C++客户端输出。

```
componentDCOMClient.cpp
nmakefile
```

例如，用 Visual C++6.0 时可以使用 nmake 产生类型库，方法是：nmake –f nmakefile。当然要保证 Visual C++6.0 的路径已经包含在系统路径中，否则需要运行 vcvars32.bat 批处理文件完成设定。

（9）DCOM Visual Basic 客户端输出。

```
componentClient.bas
```

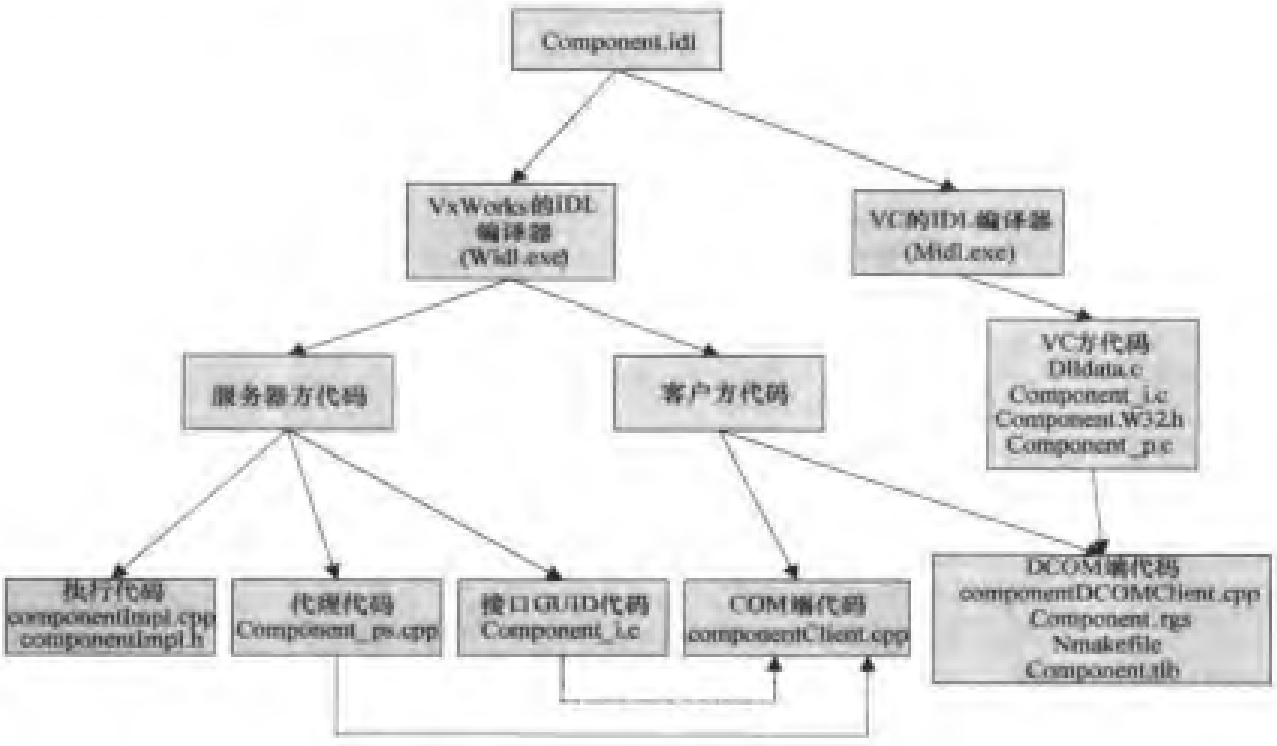


图 7-24　comwizard 产生的输出文件树

7.4.6　添加非自动类型

在使用非自动数据类型时，简单的方式就是使用 wizard 首先为接口产生自动数据类型，然后手工编辑 IDL 文件和服务器执行文件，添加 wizard 不能产生的接口方法，这些接口方法使用非自动数据类型。

使用 comwizard 产生工程文件框架，同时产生的 IDL 文件 dcomdemo.idl 如下：

```
#ifdef _WIN32
import "unknwn.idl";
#else
import "vxidl.idl";
#endif

[
    object,
    oleautomation,
    uuid(85deaae0-16d1-11db-0057-000000000000),
    pointer_default(unique)
]
interface IMath : IUnknown
{
    HRESULT Plus ([in]DOUBLE x, [in]DOUBLE y, [out]DOUBLE * ret);
    HRESULT Minus ([in]DOUBLE x, [in]DOUBLE y, [out]DOUBLE * ret);
    HRESULT Multiple ([in]DOUBLE x, [in]DOUBLE y, [out]DOUBLE * ret);
    HRESULT Divide ([in]DOUBLE x, [in]DOUBLE y, [out]DOUBLE * ret);
};

[
    object,
    oleautomation,
    uuid(883767f0-16d1-11db-000b-000000000000),
    pointer_default(unique)
]
interface IAccount : IUnknown
{
    HRESULT Save ([in]DOUBLE dbCount);
    HRESULT WithDraw ([in]DOUBLE dbMoney);

};

[
    uuid(8f151b70-16d2-11db-002a-000000000000),
    version(1.0),
    helpstring("dcomdemo Type Library")
]
library dcomdemoLib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");
```

```
    [
    uuid(8f12aa70-16d2-11db-002a-000000000000),
    helpstring("dcomdemo Class")
    ]
    coclass dcomdemo
    {
        [default] interface IMath;
                  interface IAccount;
    };
};
```

使用 comwizard 不能定义用户自定义数据类型，例如枚举、结构、数组等复合数据变量的定义。需要用户手工编辑 IDL 文件。通常情况下，需要完成以下三部分的工作：

（1）在 IDL 接口定义文件中增加数据结构的定义和接口的声明。

定义的数据结构类型可参见微软关于 IDL 语言定义的接口规范，但需要在所有接口方法的返回值指定为 HRESULT 型，并为接口参数指定自动数据类型和方向属性。声明的接口类型应具有如下形式：

```
HRESULT userMethod([方向属性]类型 参数, ...);
```

例如，在上述产生的 IDL 文件 dcomdemo.idl 的 ISum 接口中增加统计一组数组均值和极值的接口声明。在 Plus 接口前声明如下数据结构：

```
typedef struct tagARRAY_STRUCT
{
    int nLen;
    [size_is(nLen)]double *pdbArray;
}ARRAY_STRUCT;
```

在 Divide 接口后面增加两个接口声明：

```
HRESULT Mean ([in]ARRAY_STRUCT *pArray,[out]DOUBLE * ret);
HRESULT MinMax ([in]ARRAY_STRUCT *pArray,[out]DOUBLE * dbMin,[out]DOUBLE *
dbMax);
```

（2）修改服务器执行类声明头文件，增加接口的实现声明。

编辑 dcomdemoImpl.h 文件，在 dcomdemoImpl 类声明中增加如下代码：

```
STDMETHOD (Mean)(ARRAY_STRUCT* pArray,DOUBLE* ret);
STDMETHOD (MinMax)(ARRAY_STRUCT* pArray,DOUBLE* dbMin,DOUBLE* dbMax);
```

（3）修改服务器执行类源程序，增加接口的实现部分。

编辑 dcomdemoImpl.cpp 文件，增加上述操作的实现代码：

```
STDMETHODIMP dcomdemoImpl::Mean(ARRAY_STRUCT* pArray,DOUBLE* ret)
{
    return S_OK;
}
STDMETHODIMP dcomdemoImpl::MinMax(ARRAY_STRUCT* pArray,DOUBLE* dbMin,DOUBLE*
dbMax)
{
```

```
    return S_OK;
}
```

至此，已经全部完成了用户自定义数据类型及接口的修改工作。注意在使用非自动数据类型定义接口方法时，不能在接口定义中使用[oleautomation]属性。

7.4.7　完成服务器代码

一旦系统建立完成，准备为服务器组件执行 COM 类的方法编写实现程序。COM 服务器代码位于工程目录下的执行文件 componentImpl.cpp。本例中要声明该 COM 类为一单实例类，需要在服务器实现类头文件中手工增加宏定义。同时增加私有变量 dbAccount，用以存放用户账户的存款数目。

dcomdemoImpl.h 文件如下（注意黑体部分是增加的代码）：

```
/* dcomdemoImpl.h -- auto-generated COM class header */

#ifndef _dcomdemoImpl_h
#define _dcomdemoImpl_h

#include "comObjLib.h"          // COM-object template lib
#include "dcomdemo.h"           // IDL-output interface defs

class dcomdemoImpl
    : public CComObjectRoot,
      public CComCoClass<dcomdemoImpl, &CLSID_dcomdemo>,
      public IMath,
      public IAccount
{
  public:
    DECLARE_CLASSFACTORY_SINGLETON();

    dcomdemoImpl () {}
    ~dcomdemoImpl () {}

    // Interface methods go here...
    STDMETHOD (Plus) (DOUBLE x, DOUBLE y, DOUBLE * ret);
    STDMETHOD (Minus) (DOUBLE x, DOUBLE y, DOUBLE * ret);
    STDMETHOD (Multiple) (DOUBLE x, DOUBLE y, DOUBLE * ret);
    STDMETHOD (Divide) (DOUBLE x, DOUBLE y, DOUBLE * ret);
    STDMETHOD (Mean) (ARRAY_STRUCT* pArray, DOUBLE* ret);
    STDMETHOD (MinMax) (ARRAY_STRUCT* pArray, DOUBLE* dbMin, DOUBLE* dbMax);
    STDMETHOD (Save) (DOUBLE dbCount);
    STDMETHOD (Withdraw) (DOUBLE dbMoney);

    // COM Interface map
    BEGIN_COM_MAP(dcomdemoImpl)
        COM_INTERFACE_ENTRY(IMath)
        COM_INTERFACE_ENTRY(IAccount)
```

```
    END_COM_MAP()

  private:
    // Private instance variables go here...
    double dbAccount;
};

typedef CComObject<dcomdemoImpl> dcomdemoClass;
#endif
```

dcomdemoImpl.cpp 文件如下：

```
#include "dcomdemoImpl.h"       // Class Definition

AUTOREGISTER_COCLASS (dcomdemoImpl, PS_DEFAULT, 0);

// Methods for dcomdemoImpl go here...

STDMETHODIMP dcomdemoImpl :: Plus (DOUBLE x, DOUBLE y, DOUBLE * ret)
{
    *ret = x + y;
    return S_OK;
}

STDMETHODIMP dcomdemoImpl :: Minus (DOUBLE x, DOUBLE y, DOUBLE * ret)
{
    *ret = x - y;
     return S_OK;
}

STDMETHODIMP dcomdemoImpl :: Multiple (DOUBLE x, DOUBLE y, DOUBLE * ret)
{
    *ret = x * y;
     return S_OK;
}

STDMETHODIMP dcomdemoImpl :: Divide (DOUBLE x, DOUBLE y, DOUBLE * ret)
{
    *ret = x / y;
     return S_OK;
}

STDMETHODIMP dcomdemoImpl :: Mean (ARRAY_STRUCT* pArray, DOUBLE* ret)
{
    int i;

    *ret = 0.0;

    for ( i=0; i<pArray->nLen;i++)
    {
```

```
            *ret += pArray->pdbArray[i];
        }

        *ret /= pArray->nLen;

        return S_OK;
    }

    STDMETHODIMP dcomdemoImpl :: MinMax (ARRAY_STRUCT* pArray, DOUBLE* dbMin,
DOUBLE* dbMax)
    {
        int i;

        *dbMin = pArray->pdbArray[0];
        *dbMax = *dbMin;

        for ( i=1; i<pArray->nLen;i++)
        {
            if( *dbMax < pArray->pdbArray[i] )
            {
                *dbMax = pArray->pdbArray[i];
            }

            if( *dbMin > pArray->pdbArray[i] )
            {
                *dbMin = pArray->pdbArray[i];
            }
        }

        return S_OK;
    }

    STDMETHODIMP dcomdemoImpl :: Save (DOUBLE dbCount)
    {
        dbAccount += dbCount;
        return S_OK;
    }

    STDMETHODIMP dcomdemoImpl :: WithDraw (DOUBLE dbMoney)
    {
        if ( dbMoney <= dbAccount)
        {
            dbAccount -= dbMoney;
            return S_OK;
        }
        else
        {
            return S_FALSE;
        }
    }
```

7.4.8 编译并连接应用程序

创建基于 pentium 的服务器工程文件 server 和客户端工程文件 client，首先把 dcomdemoImpl.cpp 文件和 dcomdemo_i.c 文件添加到 server 工程文件中，然后再把 dcomdemo_i.c 和 client 目录下的 dcomdemoClient.cpp 添加到 client 工程文件中。

编译上述工程，分别下载 server.out 和 client.out 到目标机，并在 shell 下运行 dcomdemoClient 可以看到成功完成了 VxCOM 的演示。当然也可将由 wizard 产生的相应文件加入到工程中，这些文件包括服务器和客户方的执行代码，如果建立的是引导程序，则加入文件至引导的系统映像中，如果创建的是可下载的应用程序，则加入文件至模块中，并确保该模块下载到的引导系统包含支持应用程序的 DCOM，如图 7-25 所示。

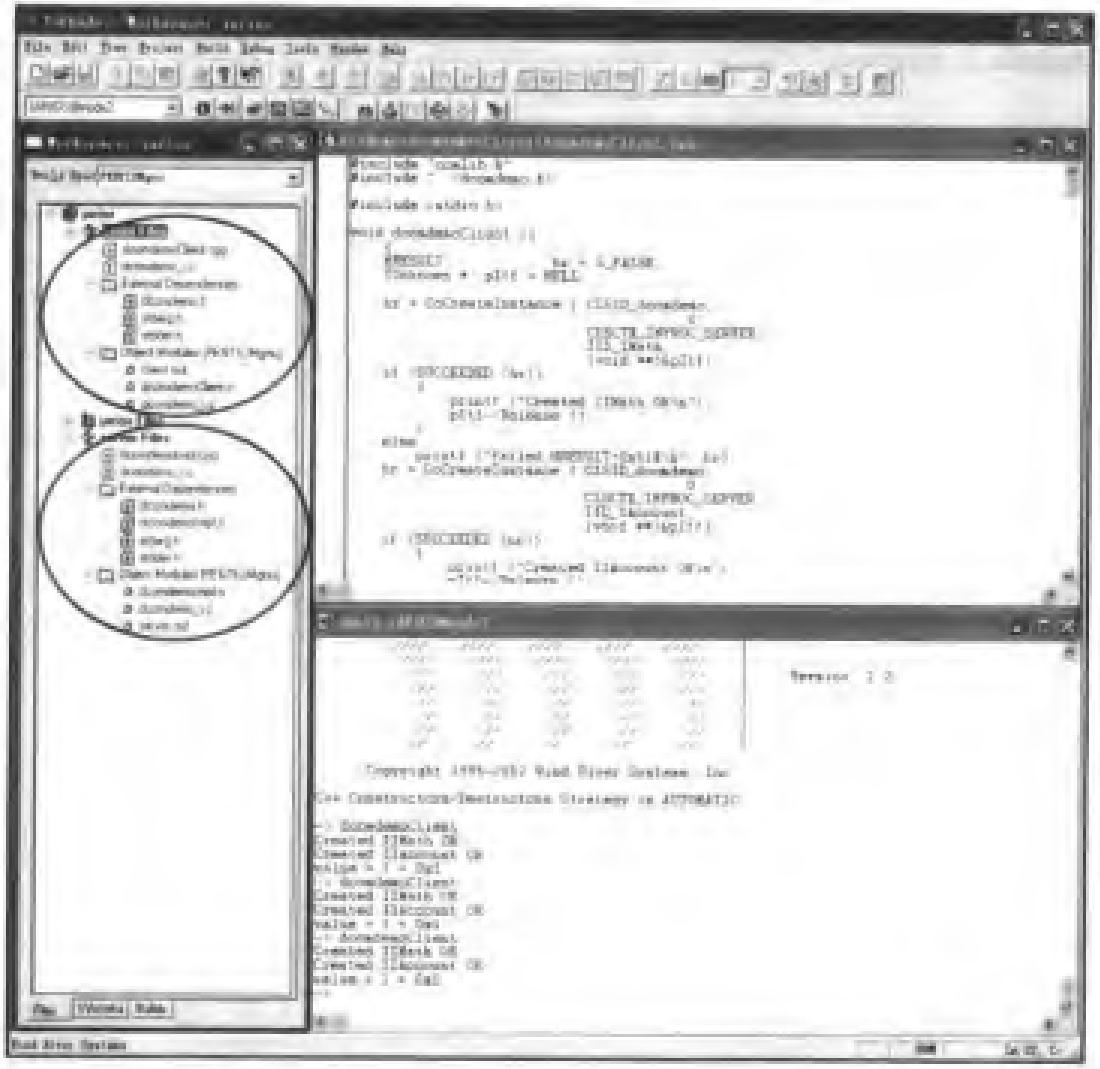


图 7-25 编译、链接、运行应用程序的效果图

第 8 章　VxCOM/VxDCOM 客户程序设计

8.1　VxWorks COM/DCOM 库组成

VxWorks 操作系统下 COM/DCOM 库主要由 COM 库和 DCOM 库组成，其中 COM 库可由核心 COM 库、COM 库以及 COM 显示函数库组成，DCOM 库则由 DCOM 库和 DCOM 扩展显示函数组成，如表 8-1 所示。

表 8-1　VxWorks COM/DCOM 库组成

名　称	函数原型	函数说明
COM 库	ComCoreLib	核心 COM 库支持
	ComLib	COM 库（VxDCOM）
	ComShow	COM 显示函数库模块
DCOM 库	DcomLib	DCOM 库（VxDCOM）
	DcomShow	DCOM 扩展显示函数

8.2　COM 库 组 成

8.2.1　comCoreLib——核心 COM 库

核心 COM 库包括的主要函数，如表 8-2 所示。

表 8-2　核心 COM 库函数说明

函数原型	函数说明
comCoreLibInit()	初始化 comCore 库
comInstanceCreate()	创建一个 coclass 实例
comClassObjectGet()	返回类对象实例
comRegistryAdd()	向 COM 核加入注册实例
comClassRegister()	为 CLSID 注册一个 get-class-object 函数
comCoreGUID2String()	将 GUID 转换为可打印的字符串
comGuidCmp()	比较两个 GUID
comSafeInc()	安全模式下变量加 1
comSafeDec()	安全模式下变量减 1

该库提供了 VxWorks 下基本的 COM 支持。通过类对象实现可扩展注册模式，提供建立类实例的函数。目前仅支持用 C/C++构造的基本 COM 接口 IUnknown 和 IClassFactory。用于 COM 接口 vtable 格式由 C/C++编译器定义，详见 comBase.h，该文件枚举了现有已知编译器和宏定义，可以产生 vtable 的结构并填充 vtable 实例。

Coclass 的注册通过库内的注册实例表处理。当 comCoreLib 请求建立实例时，首先搜索注册表，返回请求的 COM 注册类（通过 CLSID），也可以请求注册表建立一个 COM 类的实例。

8.2.2 comLib——COM 库（VxCOM）

COM 库包括如下函数，如表 8-3 所示。

表 8-3 COM 库函数说明

函数原型	函数说明
comCoreLibInit( )	初始化 comCore 库
comInstanceCreate( )	创建一个 coclass 实例
comClassObjectGet( )	返回类对象实例
comRegistryAdd( )	向 COM 核加入注册实例
comClassRegister( )	为 CLSID 注册一个 get-class-object 函数
comCoreGUID2String( )	将 GUID 转换为可打印的字符串
comGuidCmp( )	比较两个 GUID
comSafeInc( )	安全模式下变量加 1
comSafeDec( )	安全模式下变量减 1
comLibInit( )	VxWorks COM 库初始化函数
CoCreateInstance( )	创建对象类的进程内实例
CoInitialize( )	初始化支持 APARTMENTTHREADED 运行的 COM
CoInitializeEx( )	初始化 COM 运行支持
CoUninitialize( )	结束 COM 运行支持
CoGetCurrentProcess( )	返回当前进程唯一的值
CLSIDFromAscii( )	将 ASCII 字符格式的 CLSID 转化为真实的 CLSID 结构
CoGetMalloc( )	得到当前任务分配符的指针
CoTaskMemAlloc( )	为 VxDCOM 函数分配一块内存
CoTaskMemRealloc( )	改变内存块的大小
CoTaskMemFree( )	释放任务内存块
CoCreateGuid( )	创建 GUID
WriteClassStm( )	将 CLSID 写到文件
ReadClassStm( )	从文件中读入 CLSID

续表

函 数 原 型	函　数　说　明
CLSIDFromString( )	将一个宽字符格式的 CLSID 转化为 CLSID 结构
StringFromGUID2( )	基本 GUID -> 字符转换函数
StringFromCLSID( )	将 CLSID 转化为宽字符格式
StringFromIID( )	将 IID 转化为宽字符格式
IIDFromString( )	将宽字符格式的 IID 转化 IID 结构
IsEqualGUID( )	测试两个 GUID 是否相等
IsEqualCLSID( )	测试两个 CLSID 是否相等
IsEqualIID( )	测试两个 IID 是否相等
SysAllocString( )	将现有的字符串复制到新分配的字符串
SysAllocStringLen( )	创建给定长度的字符串，并用传递的字符串初始化
SysAllocStringByteLen( )	创建给定长度的字符串，并用传递的字符串初始化
SysFreeString( )	释放分配 BSTR 字符串的内存
SysStringByteLen( )	计算 BSTR 字符串中的字节数
SysStringLen( )	计算 BSTR 字符串的长度
comStreamCreate( )	创建基于内存的 IStream 接口
comWideToAscii( )	转化宽字符至字符串（ascii）
comAsciiToWide( )	转化字符串（ascii）至宽字符串
comWideStrLen( )	计算宽字符串的长度
comWideStrCopy( )	完成宽字符串的复制
VariantInit( )	初始化 VARIANT 变量
VariantClear( )	清除 VARIANT 变量
VariantCopy( )	复制 VARIANT 型变量
VariantChangeType( )	将变量从一种类型转换为另一种类型
SafeArrayCreate( )	创建新的 SAFEARRAY 数组
SafeArrayDestroy( )	删除已有数组的描述符
SafeArrayLock( )	增加 SAFEARRAY 数组的锁计数器
SafeArrayUnlock( )	减少 SAFEARRAY 数组锁计数器
SafeArrayPutElement( )	在给定位置存储数据
SafeArrayGetElement( )	查找数组中的一个元素
SafeArrayAccessData( )	锁 SAFEARRAY 数组
SafeArrayUnaccessData( )	完成数组锁计数器控制

该库提供了 Win32 COM API 调用的子集，目的是支持 VxWorks 环境下的 COM 实现。CoInitialize/CoInitialize 函数，其中 CoInitialize 函数完成本机系统的 COM 库初始化工作。

（1）CoCreateInstance 函数说明。

在完成 COM 库初始化工作后，必须为用户提供一种手段能够访问到 COM 类，CoCreateInstance 就是实现它的方法，主要是建立对象类的进程内实例。其函数原型：

```
HRESULT CoCreateInstance
    (
    REFCLSID  rclsid,              /* 对象 CLSID */
    IUnknown* pUnkOuter,           /* 指向聚合对象的指针 */
    DWORD     dwClsContext,        /* 上下文*/
    REFIID    riid,                /* 用户期望接口的 IID */
    void* *   ppv                  /* 指向接口指针的指针型变量*/
    )
```

CoCreateInstance 的第一个参数是类标识符（CLSID），用于关联要实例化的 COM 类库的 GUID。第二个参数指定对象是否为聚集对象的一部分，实例不使用聚集的情况下应将该参数设定为 NULL，否则为指向聚集 IUnkown 对象的指针。第三个参数用于指定组件将要运行的上下文。Windows 平台可以指定组件位于进程内、进程外或者远程的机器上，VxWorks 仅有进程内和远程两种类型。Windows 平台定义的数据结构定义如下：

```
typedef enum tagCLSCTX
{
    CLSCTX_INPROC_SERVER   = 1,
    CLSCTX_INPROC_HANDLER  = 2,
    CLSCTX_LOCAL_SERVER    = 4
    CLSCTX_REMOTE_SERVER   = 16
} CLSCTX;
#define CLSCTX_SERVER   (CLSCTX_INPROC_SERVER    |    CLSCTX_LOCAL_SERVER    |
CLSCTX_REMOTE_SERVER)
#define CLSCTX_ALL      (CLSCTX_INPROC_HANDLER | CLSCTX_SERVER)
```

VxWorks 在 comCoreTypes.h 文件中定义了上下文的定义：

```
enum tagCLSCTX
{
  CLSCTX_INPROC_SERVER = 1,
  CLSCTX_INPROC_HANDLER = 2,
  CLSCTX_LOCAL_SERVER = 4,
  CLSCTX_INPROC_SERVER16 = 8,
  CLSCTX_REMOTE_SERVER = 16,
  CLSCTX_INPROC_HANDLER16 = 32,
  CLSCTX_INPROC_SERVERX86 = 64,
  CLSCTX_INPROC_HANDLERX86 = 128,
};
typedef enum tagCLSCTX CLSCTX;
```

第四个参数指定所要接口的接口标识符 IID。第五个参数为返回标识符为 IID 的接口指针。例如，可以直接将接口 IID_IMath 值直接传递给该函数，使用 comwizard 产生 COM 客户端代码即采用这种方法。但建议用户应该首先获得 IUnknown 接口的指针，然后再通过调用 IUnknown::QueryInterface 来定位其他接口。

该函数用于在 VxDCOM 下创建一个局部对象（进程内服务器）的实例，dwClsContext 唯一有效值是 CLSCTX_INPROC_SERVER。成功时返回 S_OK，如果类不支持所请求的接口时返回 E_NOINTERFACE，如果 dwClsContext 无效或者类未在注册表中注册时返回 REG_E_CLASSNOTREG。

（2）CoInitialize( )函数说明。

初始化 COM 运行时支持 APARTMENTTHREADED，函数原型如下：

```
HRESULT CoInitialize
    (
    void* pv                    /* 保留。必须为 NULL */
    )
```

描述：该函数提供与 COM 标准兼容，由于 VxDCOM 不支持公寓线程模式，该函数总返回 E_INVALIDARG。

返回值：任何条件下均返回 E_INVALIDARO。

（3）CoInitializeEx()函数说明。

运行时初始化 COM 支持，函数原型如下：

```
HRESULT CoInitializeEx
    (
    void* pv,               /* 保留。必须为 NULL */
    DWORD dwCoInit          /* 必须 COINIT_MULTITHREADED */
    )
```

描述：该函数调用时 dwColnit 参数必须设置为 COINIT_MULTITHREADED，VxWorks COM 执行时没有公寓线程模式。

返回值：成功时返回 S_OK，如果调用者调用公寓线程时返回 E_INVALIDARG。

（4）CoUninitialize() 函数说明。

任务运行时不初始化 COM，函数原型如下：void CoUninitialize ()

描述：仅提供与 COM 标准兼容。

返回值：无。

8.2.3　comShow——主要 COM 显示函数库模块

comshow 的函数如下：

（1）comShowInit()——VxWorks COM 显示库初始化函数。

（2）comRegShow()——显示 VxCOM 注册表。

（3）comTrackShow()——显示所有 CoClass 的实例。

描述：该库执行一些有用的显示函数，用于显示用户 CoClass 的相关信息。

8.3　DCOM　库　组　成

8.3.1　dcomLib——DCOM 库（VxDCOM）

DCOM 库提供了 Win32 COM/DCOM API 调用的子集，目的是支持 VxWorks 环境下的 COM

和 DCOM 执行。主要库函数如表 8-4 所示。

表 8-4 DCOM 库函数说明

函 数 原 型	函 数 说 明
dcomLibInit( )	dcomLib 初始化函数
dcomLibTerm( )	dcomLib 结束函数
CoGetClassObject( )	返回 CoClass 类对象的实例
CoRegisterClassObject( )	用全局类厂注册类对象
CoRevokeClassObject( )	取消注册类对象
CoCreateInstanceEx( )	创建对象的单个实例
CoGetStandardMarshal( )	返回标准调度的实例
CoMarshalInterface( )	调度接口指针例流
CoUnmarshalInterface( )	取消来自流调度的接口指针
CoGetPSClsid( )	返回给定接口 ID 的 proxy/stub CLSID
CoGetMarshalSizeMax( )	返回调度接口的字节数的上界
vxdcomUserAdd( )	为 NTLMSSP 用户表增加一个用户
CoInitializeSecurity( )	为整个应用程序建立安全
CoDisconnectObject( )	断开对象的所有远程连接

（1）CoCreateInstanceEx( )函数说明。

创建对象的单个实例，其函数原型：

```
HRESULT CoCreateInstanceEx
    (
    REFCLSID      rclsid,          /* 对象 CLSID */
    IUnknown*     pUnkOuter,       /* 指向聚合对象的指针 */
    DWORD         dwClsCtx,        /* CLSCTX 值的一种 */
    COSERVERINFO* pServerInfo,     /* 创建对象的服务器 */
    ULONG         nInterfaces,     /* MULTI_QI 结构数 */
    MULTI_QI*     pResults         /* MULTI_QI 结构指针 */
    )
```

函数描述：该函数不同于 Win32，当建立支持 DCOM 是有效，建立 COM 时无效。

返回值：一个 HRESULT 的值。

（2）CoInitializeSecurity( ) 函数说明。

为整个应用程序建立安全，其函数原型：

```
HRESULT CoInitializeSecurity
    (
    void* psd,                     /* 安全描述- MBZ */
    long  cAuths,                  /* 必须为 1 */
    void* asAuths,                 /* array of services - MBZ */
    void* pReserved1,              /* 保留 MBZ */
    DWORD dwAuthnLevel,            /* 默认验证级别 */
    DWORD dwImpLevel,              /* 默认模拟级别 */
```

```
    void* pAuthList,            /* per-service info - MBZ */
    DWORD dwCapabilities,       /* 性能，必须为 EOAC_NONE */
    void* pReserved3            /* 保留 MBZ */
    )
```

函数描述：本函数与 Win32 平台对应函数不同，仅支持取消安全设置，原因在于 VxDCOM 不完全支持 NTLM 安全子系统，该 API 仅用于代码兼容性。为了防止在 VxDCOM 下安全行为出错，该 API 函数在试图建立非空安全设置时将报错。

返回值：在禁止安全设置请求时返回 S_OK，否则返回 E_INVALIDARG。

8.3.2　dcomShow——DCOM 扩充显示函数

```
dcomShowInit( ) - 初始化 DCOM 显示函数
```

函数描述：该函数完成 VxDCOM 网络流量的显示及报文解析机制。网络数据在压包/解包前可以捕获，数据可以通过单独的算法进行解析。所有系统级的 GUID 解析到符号，用户 com 类的 GUID 和 COM 追踪机制必须得到允许，对每一个 COM 类需增加如下的宏定义：

```
-DVXDCOM_COMTRACK_LEVEL=1
```

该函数中没有用户调用函数，输出的网络报文解析默认时输出到用户终端。如果配置参数 VXDCOM_DCOM_SHOW_PORT 设置有效，telnet 类型的客户可以使用无用的端口号完成相应的连接，输出结果可以发给客户，同时只能有一个客户，如果端口号为 0，这项功能将被禁止，所有的输出直接输出到用户终端。该功能会带来很高的性能负载，目的在于提供调试信息，否则不应包括进来。

8.4　客户端程序设计

8.4.1　VxCOM 客户端程序设计

如果 COM 服务器与客户程序位于同一目标机上，此时应建立进程内服务器模型，在建立工程文件时不需要加入代理/存根代码。建立 server 和 client 工程文件（可参见第 7 章）。

此时客户方程序的框架代码如下：

（1）声明接口指针。

```
IUnknow *pItf=NULL;
IMath *pMathCom=NULL;
```

（2）初始化 COM 库。

```
CoInitialize(NULL);
```

（3）创建类实例。

```
CoCreateInstance(类名 CLSID,
        NULL,
        公寓模型,
        接口 IID,一般为 IID_IUnknown,
        接口指针(void **)&pItf
        );
```

例：hr=CoCreateInstance(CLSID_dcomdemo, NULL, CLSCTX_INPROC_SERVER, IID_IUnknown, (void**)&pUnknown);

得到用户接口

```
pItf->QueryInterface(接口 IID，用户接口指针 pUserCom);
```

例：hr=pUnknown->QueryInterface(IID_IMath, (void**)&pMathCom);

释放接口

```
pItf->Release();
```

调用用户接口

```
pUserCom->Method(...);
```

释放用户接口

```
pUserCom->Release();
```

COM 库释放

```
CoUnitialize();
```

重新编写的客户端程序如下：

```
#include "comLib.h"
#include "../dcomdemo.h"

#include <stdio.h>

void vxClient ()
{
 HRESULT         hr = S_FALSE;
 IUnknown *      pItf = NULL;
 IMath *         pMathCom = NULL;
 IAccount *      pAccountCom = NULL;

    //初始化 COM 库
    CoInitialize(NULL);

    //创建类实例
    hr = CoCreateInstance ( CLSID_dcomdemo,
        0,
        CLSCTX_INPROC_SERVER,
        IID_IUnknown,
         (void **)&pItf);
    if (SUCCEEDED (hr))
    {
        printf ("Created IID_IUnknown OK\n");

        //得到用户接口
        hr = pItf->QueryInterface(IID_IMath,(void **)&pMathCom);
        if (SUCCEEDED (hr))
        {
            printf ("Created IID_IMath OK\n");
        }
```

```
        hr = pItf->QueryInterface(IID_IAccount,(void **)&pAccountCom);
        if (SUCCEEDED (hr))
        {
            printf ("Created IID_IAccount OK\n");
        }

        //释放接口
        pItf->Release ();

        //调用用户接口
        double x=1.0,y=3.3,ret;

        hr = pMathCom->Plus(x,y,&ret);
        printf("%f + %f = %f\n",x,y,ret);

        hr = pMathCom->Minus(x,y,&ret);
        printf("%f - %f = %f\n",x,y,ret);

        hr = pMathCom->Multiple(x,y,&ret);
        printf("%f * %f = %f\n",x,y,ret);

        hr = pMathCom->Divide(x,y,&ret);
        printf("%f / %f = %f\n",x,y,ret);

        double xx[]={1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0};
        ARRAY_STRUCT array;

        array.nLen = sizeof(xx)/sizeof(double);
        array.pdbArray=xx;

        double dbMin,dbMax;

        hr = pMathCom->Mean(&array,&ret);
        printf("array xx mean is %f\n",ret);

        hr = pMathCom->MinMax(&array,&dbMin,&dbMax);
        printf(" max is %f \n min is %f\n",dbMax,dbMin);

        hr = pAccountCom->Save(100.0);
        hr = pAccountCom->WithDraw(70.0);
        hr = pAccountCom->WithDraw(80.0);
        if(hr == S_FALSE)
        {
            printf("could not withdraw enough money\n");
        }

        //释放用户接口
        pMathCom->Release();
        pAccountCom->Release();
    }
    else
    {
        printf ("Failed:HRESULT=0x%lX\n", hr);
```

```
    }

    //COM 库释放
    CoUninitialize();
}
```

将服务器目标代码与客户方目标代码下载到同一台目标机上，并在 shell 下运行 vxClient，其输出如下：

```
-> vxClient
Created IID_IUnknown OK
Created IID_IMath OK
Created IID_IAccount OK
1.000000 + 3.300000 = 4.300000
1.000000 - 3.300000 = -2.300000
1.000000 * 3.300000 = 3.300000
1.000000 / 3.300000 = 0.303030
array xx mean is 5.500000
 max is 10.000000
 min is 1.000000
could not withdraw enough money
value = 1 = 0x1
->
```

8.4.2 VxDCOM 客户方程序设计

建立工程文件 remoteclient 和 remoteserver，其中 remoteserver 工程文件包括文件 dcomdemoImpl.cpp、dcomdemo_i.c 和 dcomdemo_ps.cpp；remoteclient 工程文件包括 VxDCOM 客户端程序 dcomdemoDCOMClient.cpp 以及接口 IID 文件 dcomdemo_i.c，代理文件 dcomdemo_ps.cpp，如图 8-1 所示。

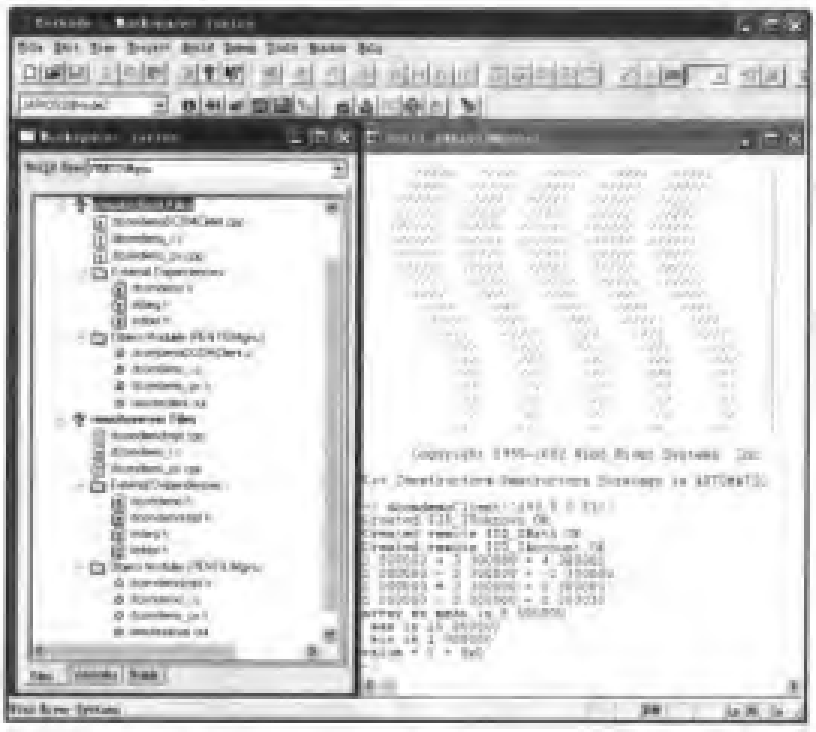


图 8-1 VxWorks DCOM 客户端工程文件

（1）声明接口指针。

```
IID_IMath *pMathCom=NULL;
```

（2）初始化 COM 库。

```
hr = CoInitializeEx (0, COINIT_MULTITHREADED);
```

（3）创建类实例。

```
OLECHAR     *wszServerName = L"194.0.0.91";

MULTI_QI mqi [] = {
    {&IID_IUnknown,0,S_OK}
                };

COAUTHINFO authInfo = {
                    RPC_C_AUTHN_WINNT,
                RPC_C_AUTHZ_NONE, 0,
                RPC_C_AUTHN_LEVEL_NONE,
                RPC_C_IMP_LEVEL_IMPERSONATE,
                0,
                EOAC_NONE
                };

COSERVERINFO serverInfo = { 0, wszServerName, &authInfo, 0 };

hr = CoCreateInstanceEx (CLSID_dcomdemo,
                    0,
                           CLSCTX_REMOTE_SERVER,
                    &serverInfo,
                    1,
                    mqi);
```

（4）得到用户接口。

```
pItf->QueryInterface(接口 IID，用户接口指针 pUserCom);
```

例：

```
mqi[0].pItf->QueryInterface(IID_IMath,(void **)&pMathCom);
```

（5）释放接口。

```
mqi [0].pItf->Release ();
```

（6）调用用户接口。

```
pUserCom->Method(...);
```

（7）释放用户接口。

```
pUserCom->Release();
```

（8）COM 库释放。

```
CoUnitialize();
```

编写的 VxDCOM 客户端程序如下：

```
#ifdef _WIN32
#define _WIN32_WINNT 0x0400
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include "dcomdemo.w32.h"
#else
#include "dcomLib.h"
#include "../dcomdemo.h"
#define mbstowcs comAsciiToWide
#endif

#include <stdio.h>

int dcomdemoClient (const char* serverName)
{
   OLECHAR      wszServerName [128];
   HRESULT      hr = S_OK;
   //声明接口指针
 IMath * pMathCom = NULL;
 IAccount * pAccountCom = NULL;

   mbstowcs (wszServerName, serverName, strlen (serverName) + 1);

   //初始化 COM 库
   hr = CoInitializeEx (0, COINIT_MULTITHREADED);
   if (FAILED (hr))
      return hr;
   //创建类实例
   MULTI_QI mqi [] = {
    {&IID_IUnknown, 0, S_OK},
 };

   COAUTHINFO authInfo = {
    RPC_C_AUTHN_WINNT,
        RPC_C_AUTHZ_NONE, 0,
        RPC_C_AUTHN_LEVEL_NONE,
        RPC_C_IMP_LEVEL_IMPERSONATE,
        0,
        EOAC_NONE
 };

   COSERVERINFO serverInfo = { 0, wszServerName, &authInfo, 0 };

   hr = CoCreateInstanceEx ( CLSID_dcomdemo,
    0,
    CLSCTX_REMOTE_SERVER,
    &serverInfo,
    1,
```

```
    mqi);

   if (SUCCEEDED (hr))
{
      if (SUCCEEDED (mqi [0].hr))
    {
        printf ("Created IID_IUnknown OK\n");

        //得到用户接口
        mqi [0].pItf->QueryInterface(IID_IMath,(void **)&pMathCom);
        if (SUCCEEDED (hr))
        {
           printf ("Created remote IID_IMath OK\n");
        }

        hr = mqi [0].pItf->QueryInterface(IID_IAccount,(void **)&pAccountCom);
        if (SUCCEEDED (hr))
        {
           printf ("Created remote IID_IAccount OK\n");
        }

        //释放接口
          mqi [0].pItf->Release ();

        //调用用户接口
        double x=1.0,y=3.3,ret;

        hr = pMathCom->Plus(x,y,&ret);
        printf("%f + %f = %f\n",x,y,ret);

        hr = pMathCom->Minus(x,y,&ret);
        printf("%f - %f = %f\n",x,y,ret);

        hr = pMathCom->Multiple(x,y,&ret);
        printf("%f * %f = %f\n",x,y,ret);

        hr = pMathCom->Divide(x,y,&ret);
        printf("%f / %f = %f\n",x,y,ret);

        double xx[]={1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0};
        ARRAY_STRUCT array;

        array.nLen = sizeof(xx)/sizeof(double);
        array.pdbArray=xx;

        double dbMin,dbMax;

        hr = pMathCom->Mean(&array,&ret);
        printf("array xx mean is %f\n",ret);

        hr = pMathCom->MinMax(&array,&dbMin,&dbMax);
        printf(" max is %f \n min is %f\n",dbMax,dbMin);
```

```
        hr = pAccountCom->Save(100.0);
        hr = pAccountCom->WithDraw(70.0);
        hr = pAccountCom->WithDraw(80.0);
        if(hr == S_FALSE)
        {
            printf("could not withdraw enough money\n");
        }

        //释放用户接口
        pMathCom->Release();
        pAccountCom->Release();

    }
      else
      {
       printf ("Failed:HRESULT=0x%lX\n", mqi [0].hr);
    }
}
  else
{
      printf ("Failed:HRESULT=0x%lX\n", hr);
}

  CoUninitialize ();

  return hr;
}

#ifdef _WIN32
int main (int argc, char* argv [])
{
  if (argc != 2)
{
      puts ("usage: dcomdemoClient <server>");
      exit (1);
}

  return dcomdemoClient (argv [1]);
}
#endif
```

将产生的目标代码下载到不同的目标机上。例如：将产生的 remoteserver.out 目标码下载到目标机 1（其 IP 地址为“193.0.0.51”），产生 remoteclient.out 目标码下载到目标机 2（其 IP 地址为“193.0.0.52”），并在目标机 2 的 shell 下运行：dcomdemoClient("193.0.0.51")，其输出为：

```
-> dcomdemoClient("193.0.0.51")
Created IID_IUnknown OK
Created remote IID_IMath OK
Created remote IID_IAccount OK
1.000000 + 3.300000 = 4.300000
```

```
1.000000 - 3.300000 = -2.300000
1.000000 * 3.300000 = 3.300000
1.000000 / 3.300000 = 0.303030
array xx mean is 5.500000
 max is 10.000000
 min is 1.000000
value = 0 = 0x0
->
```

8.4.3　Visual C++6.0 客户程序设计

在 Windows 平台上，访问 VxDCOM 服务器采用何种存根/代理机制，一般可根据具体情况进行分析，主要有三种形式可供用户选择：

（1）类型库。

（2）标准调度（DLL 动态链接库）。

（3）用户定制调度策略。

建议用户使用标准调度和类型库完成远程构件的注册。

本节描述注册代理（proxy）动态链接库（DLL），类型库，服务器以及授权服务器和激活服务器。下面我们首先介绍动态链接库的方法，然后再介绍类型库的方法。

8.4.3.1　注册、发布和运行应用程序

在 Windows 注册代理动态链接库（Proxy DLL）

在 Windows 机器上，对自动数据类型则不需要 proxy/stub。通常，使用自动数据类型来定义接口，这些类型选择是从 VxDCOM wizard 的接口方法参数产生的，选择这些类型，参数类型被定义为[oleautomation]属性，表示它们为自动类型，Win32 Automation Marshaler 表明不需多余的 Win32 proxy/stub 动态链接库，因为这些类型的调度是自动完成的。如果使用了非自动类型，必须在 Windows 平台上注册代理动态链接库。另一方面，如果工程需要非自动类型，就不能指定[oleautomation]属性，也不能自动完成参数类型的调度，对使用非自动类型的接口，必须产生并安装 Win32 proxy/stub 动态链接库，该动态链接库包含使用新接口的客户方程序分发的接口代理。

widl 能够编译指定的非自动类型。

8.4.3.2　在 VC 环境下为标准调度创建代理/存根动态链接库

按以下步骤创建一个代理（proxy）/存根（stub）动态链接库（DLL）：

（1）使用 MIDL 编译用户 IDL 文件（例如 dcomdemo.idl）。

由 IDL 编译器 midl 产生的客户端代理/注册的主要代码如表 8-5 所示。

表 8-5　midl 编译器产生的代码列表

文件名称	描　　述	对应的 dcomdemo.idl 输出
idlname.h	接口定义头文件	dcomdemo.h
idlname_i.c	定义 IID 和 CLSID 常量的文件	dcomdemo_i.c
idlname_p.c	编码文件	dcomdemo_p.c
idlname.tlb	类型库	dcomdemo.tlb
dlldata.c	编码程序的入口	dlldata.c

（2）打开 Visual C++选择 File/New 菜单。

（3）选择工程标签，并选择 Win32 Dynamic-Link Library。

（4）在 Project Name 文本框中，输入 DDL 名字（例如：dcomdemo），然后按 OK 键。

（5）当询问创建何种 DLL 库，选择使用空的 DLL 工程文件，选择 Finish 按键，并按 OK 键。

（6）在 Project/FileView 命令框中选择 Project/Add 菜单。

（7）选择由 MIDL 产生的文件 dlldata.c，dcomdemo_i.c 和 dcomdemo_p.c，并按 OK 键。

（8）选择 File/New 命令，然后选择 Files 标签，并选择文本文件。

（9）在文件名字输入框中，输入 dcomdemo.def，并按 OK 键。

在定义文件中输入如下模块定义：

```
;dcomdemo.def
LIBRARY dcomdemo.dll
DESCRIPTIO dcomdemo.DLL'
EXPORTS
   DllGetClassObject   @1 PRIVATE
   DllCanUnloadNow     @2 PRIVATE
   DllRegisterServer   @3 PRIVATE
   DllUnregisterServer @4 PRIVATE
```

（10）选择 File/Save 命令，然后选择 Project/Settings。

（11）在 Settings 的列表框中，选择 All Configurations。

（12）选择 C/C++标签，并在目录列表框中选择 General。

（13）在预定义（Preprocessor）目录框中，增加 REGISTER_PROXY_DLL 和_WIN32_DCOM 宏定义，中间由逗号分开，如图 8-2 所示。

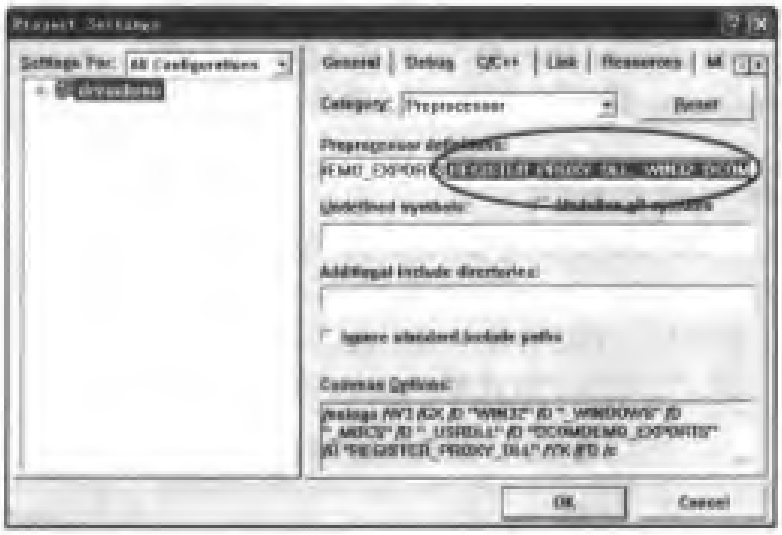


图 8-2 增加预定义宏

（14）选择 Link 标签，并在其目录列表中选择 General。

（15）在 Object/Library Modules 框中，增加 rpcndr.lib、rpcns4.lib 和 rpcrt4.lib 用空格分开，如图 8-3 所示，都设定完成后按 OK 键。

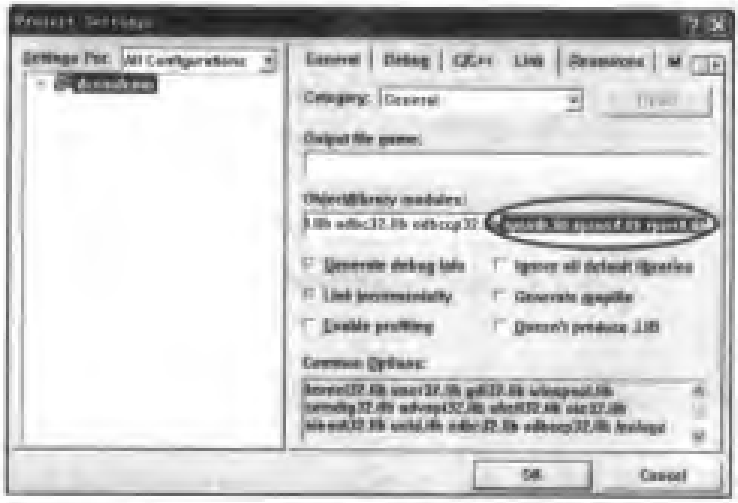


图 8-3　增加远程构件 RPC 库支持

或者在应用程序中加入如下代码：

```
#pragma comment(lib, "rpcndr.lib")
#pragma comment(lib, "rpcns4.lib")
#pragma comment(lib, "rpcrt4.lib")
```

（16）选择 Build/Build component.dll 菜单。

（17）确保上述工作完成后，选择 Tools/Register Control 菜单进行构件的注册，如图 8-4 所示。

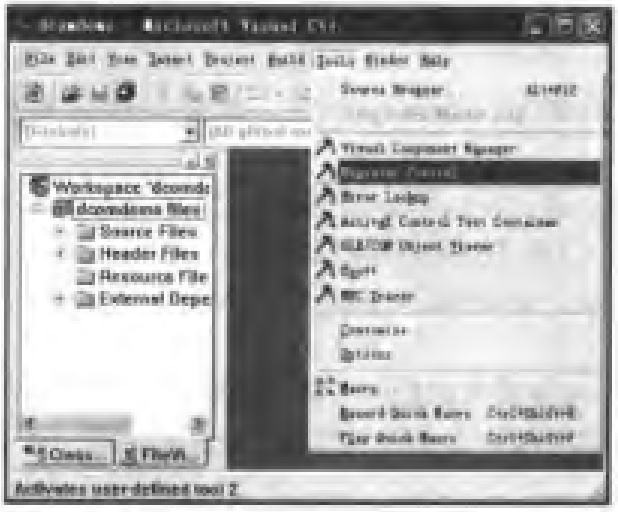


图 8-4　完成构件注册

成功后弹出一个对话框，如图 8-5 所示。

图 8-5　构件注册成功后弹出对话框

用户也可以手工进行注册，方法是使用微软的 regsvr32，如图 8-6 所示。

图 8-6　采用手工注册时的对话框提示

```
Regsvr32 dcomdemo.dll
```

注册动态链接库时，仅把该库注册到系统的注册表中。

8.4.3.3　注册类型库

类型库是二进制文件，扩展名为.tlb，存储关于 DCOM 对象属性和方法的信息，可以在运行时访问其他应用程序。Windows 客户方程序使用类型以决定对象支持的接口和调用的接口方法。为了保证 midl 能够正确产生类型库，必须将关键字 librayr 添加到 IDL 文件中，IDL 文件中的 library 代码段中还有统一的唯一标识符（UUID）。可以使用注册文件（.reg）来创建注册入口，但是对类型库注册的最好方法是采用编程方式，调用 LoadTypeLibEx 函数来完成类型库注册工作，使用 UnRegisterTypeLib 取消类型库的注册。基于此，类型库必须注册，欲注册类型库，运行 installDir\host\X86-win32 \bin\vxreg32.exe，该命令向 Windows 注册表中加入类型库，从而对象可以从 Windows 主机上得到访问，如果向 IDL 文件中加入接口，类型库必须重新注册，这样接口才能被 Windows Automation Marshaler 知道。

注册包括名叫 'vxworks.target' 的实体，可以修改指定目标机，有两种方式改变目标机实体注册类型库：

（1）编辑由 wizard 产生的 basenaem.rgs 文件，改变'vxworks.target'实体为目标机 IP 地址，然后运行 vxreg32。

（2）首先运行 vxreg32，改变 Windows 注册表中目标机的 DCOMCNFG，使用工具改变它。

说明：vxreg32 是 VxDCOM 的 DCOM 注册工具，可以在 NT 客户端注册 VxDCOM 控件。为了保证控件能够正确注册，必须保证<ProjectName>.tlb 和<ProjectName>.rgs 位于同一目录下。

VxWorks 提供的 VxReg32 源代码程序，该程序提供了类型库和注册文件的注册，并没有取消类型库的注册代码，有兴趣的读者可以自己编写这些代码。

```
/* vxreg32.cpp - DCOM registration utility for VxDCOM */

/* Copyright (c) 1998-1999 Wind River Systems, Inc. */

 /*
modification history
--------------------
01a,09Aug00,nel  Written (cribbed from TypeLib reader)

*/

/*
  DESCRIPTION:
```

```
    Command line utillity to register a VxDCOM control on
    an NT client.

    To register the control the files <ProjectName>.tlb and
    <ProjectName>.rgs must be present in the same directory.
    The project basename is supplied to the utility and it
    registers the interface.

    e.g.
        If the project basename is CoMathDemo then typing

        vxreg32 CoMathDemo

        in the directory containing both the files
        CoMathDemo.rgs and CoMathDemo.tlb will register
        the CoClass.

    CoMathDemo.tlb is generated by midl, whilst CoMathDemo.rgs
    is generated by the COM Wizard.

NOMANUAL
*/

#include <iostream.h>
#include <stdio.h>
#include <atlbase.h>
#include <comdef.h>
#include <atlimpl.cpp>

/*****************************************************************
*
* RegisterTLBFile - Register the interfaces contained in the TLB file
*
* This function is used to register the CoClass interfaces contained in
* the TLB file created by midl.
*
* RETURNS: S_OK, or and error code if the TLB can't be registered.
*
* NOMANUAL
*/

HRESULT RegisterTLBFile
    (
    const char*     pszFileName
    )
    {
    HRESULT         hr = S_OK;
    ITypeLib*       pTLB;
```

```
    _bstr_t       bsFileName (pszFileName);

    bsFileName += ".tlb";

    hr = LoadTypeLibEx (bsFileName, REGKIND_REGISTER, &pTLB);

    if (FAILED (hr))
        {
        cerr << "ERROR: file " << bsFileName << " could not be registered, " <<
            "HRESULT=0x" << cerr.width (8) << hex << hr << "\n";
        return hr;
        }

    pTLB->Release ();
    return S_OK;
    }


/*****************************************************************
*
* RegisterRGSFile - Register the CoClass contained in an RGS file.
*
* This function is used to register the CoClass contained in
* the RGS file created by the COM Wizard.
*
* RETURNS: S_OK, or and error code if the RGS can't be registered.
*
* NOMANUAL
*/

int RegisterRGSFile
    (
    const char*       pszFileName
    )
    {
    IRegistrar*       pRegistrar;
    HRESULT           hr = S_OK;
    _bstr_t           bsFileName (pszFileName);

    bsFileName += ".rgs";

    hr = CoCreateInstance (CLSID_Registrar,
                           NULL,
                           CLSCTX_INPROC_SERVER,
                           IID_IRegistrar,
                           (void**) &pRegistrar);
    if (FAILED (hr))
        {
        cerr << "ERROR (HRESULT=0x" << cerr.width (8) << hex << hr << ") : " <<
```

```
            "the Registrar could not be created.\n" <<
            "Please ensure that ATL.DLL is registered on this system.\n";
        return E_FAIL;
        }

    hr = pRegistrar->FileRegister (bsFileName);
    pRegistrar->Release ();

    if (FAILED (hr))
        {
        cerr << "ERROR: file " << bsFileName << " could not be registered" <<
            ", HRESULT=0x" << cerr.width (8) << hex << hr << "\n";
        return hr;
        }

    return S_OK;
    }

void main (int argc, char * argv [])
    {
    if (argc < 2)
        {
        cout << "Usage: " << argv [0] << " <Project Name>\n";
        }
    else
        {
        CoInitialize (NULL);

        if (RegisterTLBFile (argv [1]) == S_OK)
            {
            RegisterRGSFile (argv [1]);
            }
        }
    }
```

例如，如果工程名称为 dcomdemo，则注册时：

```
vxreg32 dcomdemo
```

注意：在当前目录下应包括 dcomdemo.rgs 和 dcomdemo.tlb。前者由 comwizard 向导产生，后者由 midl 产生。

注册类型库后，可在注册表（使用 regedit.exe）和组件服务（使用 dcomcnfg.exe）中找到 dcomdemo 类型库，如图 8-7 所示。用户也可修改其 DCOM 配置，也可使用 oleview.exe 查看类型库的详细信息，如图 8-8 所示。

欲产生类型库，最方便的方法是使用 VC 编译的 nmake 批处理命令，编译 Win32 客户方程序，确保安装了 Visual C++，nmake 文件所在的路径已经指定（如果未指定，请在命令行运行 vcvars32.bat 命令），并从命令行运行：

```
nmake -f nmakefile
```

图 8-7　组件服务中用户组件属性

图 8-8　类型库信息

（1）注册服务器。

需在 VxWorks COM 注册表中注册 DCOM 服务器类，完成客户程序的定位，每一个目标模块仅有一个服务器类，目标模块包括服务器类代码，proxy/stub 代码，注册目标码。注册目

标码在 VxWorks 注册服务器类，不管预连接或下载的，能确保模块自动注册。这种自动注册过程不依赖于构造的次序，在系统初始化安全运行。

在加载时或者系统启动时自动注册 DCOM 服务器，需要在 COM 类的执行文件中包括 AUTOREGISTER_COCLASS 宏定义。这为 VxWorks COM 注册表中创建类实体，AUTOREGISTER_COCLASS 将 COM 类与其类标识符（CLSIDS）相关，在 VxRegistry 中注册模块名。

当产生框架执行文件时该宏自动被包括进来，用户不必增加宏定义（除非文件不是自动产生的）。例如产生的 dcomdemoImpl.cpp 中有如下代码：

```
AUTOREGISTER_COCLASS (dcomdemoImpl, PS_DEFAULT, 0);
```

AUTOREGISTER_COCLASS 宏定义有三个参数：

1）服务器执行 COM 类。

2）优先级模式。

3）默认的优先级。

优先级参数作为实时扩充优先级模式的一部分。

（2）授权服务器。

VxWorks 下的 DCOM 注册表并不复杂，Win32 支持的多用途注册表，专门设计目的是：

1）允许 COM 服务器类注册 CLSID（类 ID）。

2）在给定 CLSID 条件下，提供实例过程的连接。

3）在给定 IID 条件下，对给定的接口决定正确的 proxy/stub 配置。

VxWorks 注册工作在简单相关查询方式，以接口和类标识为关键词，存储在注册项的值是简单的字符串，用于不同内部函数查询 proxy/stub 类，系统提供的对象（例如标准调度），以及这些值所标识的其他对象。

VxWorks 的非 Win32 兼容的 API 函数调用访问注册表，用户程序代码不需要这些 API，因为 widl 自动为服务器类产生注册代码。

VxDCOM 充当服务器时可以参与基本 NTLM 授权过程，作为客户方则没有，可以识别来自 NT 的授权需求，正确完成请求/应答处理，默认方式下并不对这些处理采取任何措施。未来的 VxDCOM 可以完全支持 NTLM 安全系统，这要依赖于 NT 的域安全、网络安全等等。目前最安全最有效的方式是使用注册/DCOMCNFO 工具或者调用 CoInitializeSecurity( )禁止客户方安全。

（3）激活服务器。

完成上述步骤后，用户就可以运行程序，所有 VxDCOM 线程创建时必须有 VX_FP_TASK 属性。

（4）创建客户程序。

创建 VC 控制台程序，将 VxDCOM 客户端程序 dcomdemoDCOMClient.cpp 加入到工程文件中，使用 VC 的 IDL 编译器（midl.exe）编译 dcomdemo.idl 文件，将包括 COM 类标识符，接口标志符的文件 dcomdemo_i.c 也加入到工程文件中，编译控制台程序，产生执行文件，在完成标准调度动态链接库(regsvr32 dcomdemo.dll)或者类型库（vxreg32 dcomdemo）注册后，设定第一个入口参数为“193.0.0.51”，运行该程序，可以看到 VC 客户端成功访问到 VxDCOM 服务器程序，如图 8-9 所示。

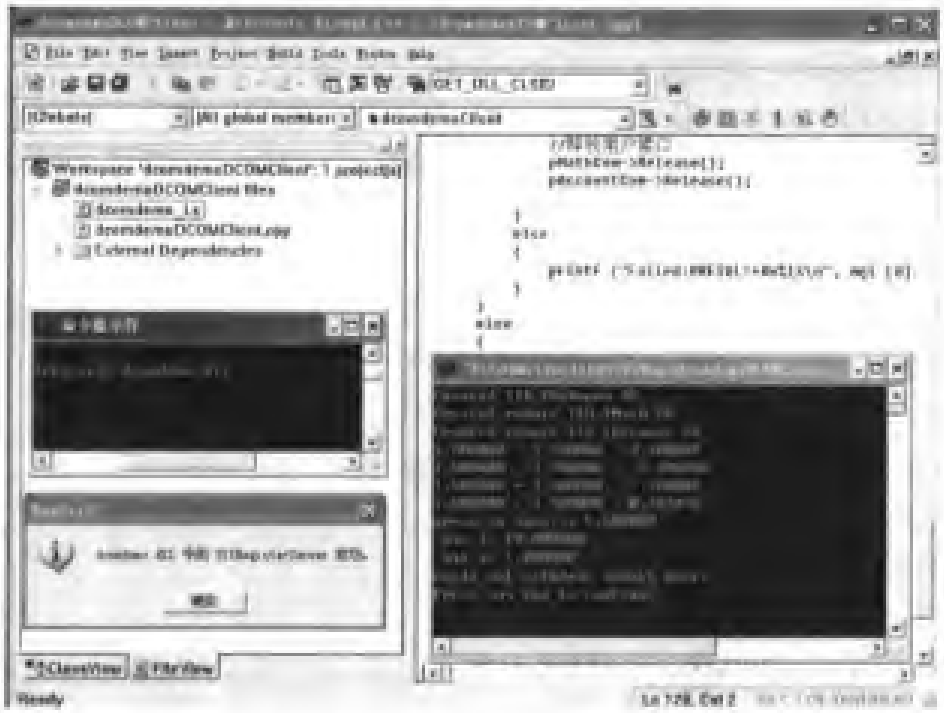


图 8-9 Visual C++客户端程序运行结果

8.4.4 Delphi 客户程序设计

在本章中，我们在 VxWorks 操作系统中建立了一个 VxDCOM 组件，该组件产生的 client 目录下可以使用 VC 的 nmake 产生类型库。这样，就可以很方便地在其他编程环境下使用该组件。

本例中由于 dephi 对结构编译时为压缩格式（packed record），因此需要修改 IDL 文件的数据结构 ARRAY_STRUCT 的定义，将其修改为：

```
typedef struct tagARRAY_STRUCT
{
    long nLen;
    float pdbArray[10];
}ARRAY_STRUCT;
```

重新使用 widl 编译该 IDL 文件，并在 client 目录下重新产生类型库文件，完成类型库的注册，并在 tornado 开发环境下重新编译服务器程序，产生目标代码下载到目标机。

在 dephi 中，使用 CreateComObject 实现对本地组件对象的访问， CreateRemoteObject 函数是用来完成远程对象的初始化工作，其函数原型为：

```
function CreateRemoteComObject(const MachineName: WideString; const ClassID:
TGUID): IUnknown;
```

CreateRemoteObject 函数的第一个参数 MachineName，可以设置为远程计算机的名字或者 IP 地址，第二个参数 ClassID 指明远程要实例化的 COM 类标识符，如果导入类型库产生单元文件，则该文件对这些接口进行了封装。

这里介绍使用 Delphi7 开发环境建立客户程序相关步骤：

（1）打开 Delphi7.0 开发环境。

（2）选择 File->New->Application，创建工程。

（3）选择 Project->Import Type Library...，弹出类型库对话框，如图 8-10 所示。

选择 dcomdemo Type Library(Version 1.0)选项，如果该类型库不在列表中，按“Add...”按钮，并选择“Create Unit”产生类型库的单元源程序，这里产生是 dcomdemoLib_TLB.pas 文件。以下为 dephi 导出的数据结构 ARRAY_STRUCT 的定义：

```
PUserType1 = ^tagARRAY_STRUCT; {*}

tagARRAY_STRUCT = packed record
  nLen: Integer;
  pdbArray: array[0..9] of Single;
end;
```

（4）在 Form1 放置相应控件。

9 个 TEdit 控件，9 个 TLabel 控件和 1 个 TButton 控件，其外形如图 8-11 所示，属性如表 8-6 所示。

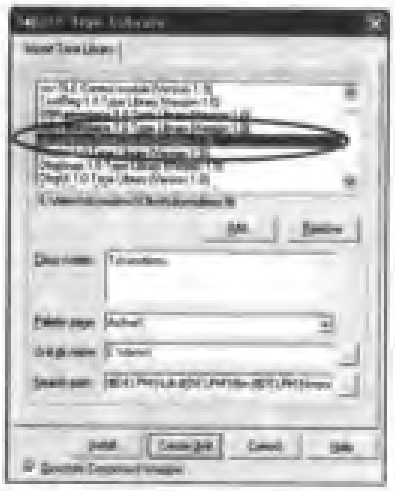


图 8-10　Delphi 导入类型库

图 8-11　Delphi 窗体布局

表 8-6　控 件 的 属 性

控件名称	控件类型	相关属性	值
Form1	TForm1t	Caption	“Delphi 客户端程序”
Edit1	TEdit		
Edit2	TEdit		
Edit3	TEdit	Enabled	False
Edit4	TEdit	Enabled	False
Edit5	TEdit	Enabled	False
Edit6	TEdit	Enabled	False
Edit7	TEdit	Enabled	False
Edit8	TEdit	Enabled	False
Edit9	TEdit	Enabled	False

续表

控件名称	控件类型	相关属性	值
Label1	TLabel	Caption	"X="
Label2	TLabel	Caption	"Y="
Label3	TLabel	Caption	"X+Y="
Label4	TLabel	Caption	"X-Y="
Label5	TLabel	Caption	"X*Y="
Label6	TLabel	Caption	"X/Y="
Label7	TLabel	Caption	"Mean"
Label8	TLabel	Caption	"Min"
Label9	TLabel	Caption	"Max"
Button1	TButton	Caption	"&Compute"

（5）选择 Object Inspector，在 events 中增加用户事件。

首先需要为 Button1 增加 OnClick 事件，还为 Form1 增加 OnCreate 事件。在使用的单元中增加 dcomdemoLib_TLB 项，编辑 Button1Click 和 FormCreate 实现代码。Button1Click 主要完成 DCOM 远程访问，而 FormCreate 主要完成 9 个 TEdit 控件的初始化，保证其数据初始有意义。

源程序如下（黑体部分需要手工输入）：

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs,dcomdemoLib_TLB, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Edit4: TEdit;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    Edit5: TEdit;
    Edit6: TEdit;
    Label7: TLabel;
```

```
    Label8: TLabel;
    Label9: TLabel;
    Edit7: TEdit;
    Edit8: TEdit;
    Edit9: TEdit;
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
var
  client : IMath;
  x      : double;
  y      : double;
  ret    : double;
  minRet : double;
  maxRet : double;
  index  : Integer;
  myStru : tagARRAY_STRUCT;
begin
  client := Codcomdemo.CreateRemote('193.0.0.51');

  x:=strtofloat(Edit1.Text);
  y:=strtofloat(Edit2.Text);

  client.Plus(x,y,ret);
  Edit3.Text := floattostr(ret);

  client.Minus(x,y,ret);
  Edit4.Text := floattostr(ret);

  client.Multiple(x,y,ret);
  Edit5.Text := floattostr(ret);

  client.Divide(x,y,ret);
  Edit6.Text := floattostr(ret);

  for index:=0 to 9 do
  begin
```

```
    myStru.pdbArray[index] := 1.0+index;
  end;

  myStru.nLen := 10;
  client.Mean(myStru,ret);
  Edit7.Text := floattostr(ret);

  client.MinMax(myStru,minRet,maxRet);
  Edit8.Text := floattostr(minRet);
  Edit9.Text := floattostr(maxRet);

  client:=nil;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Edit1.Text := '1.0';
  Edit2.Text := '2.0';
  Edit3.Text := '';
  Edit4.Text := '';
  Edit5.Text := '';
  Edit6.Text := '';
  Edit7.Text := '';
  Edit8.Text := '';
  Edit9.Text := '';
end;

end.
```

选择 Run->Run，然后单击 Compute 按钮测试该组件。输出结果如图 8-12 所示。用户可以修改 X、Y 的值来观察输出结果。

图 8-12　Delphi 调用 VxDCOM 服务器运行结果

第 9 章　VxCOM/VxDCOM 数据类型及接口

9.1　自 动 化 数 据 类 型

自动化数据类型包括简单类型、接口指针、VARIANT 和 BSTR。其中简单数据类型指 long, long *, int, int *, short, short *, char, char *, float, float *, double, double *。接口数据类型指 IUnknown *, IUnknown **。

9.1.1　VARIANT 变量

VxWorks 下定义了 VARIANT 结构，目的是为了与 Windows 平台兼容，VARIANT 结构是一个所有与自动化兼容类型的联合，其定义在 comAutomaiton.idl 文件中，其变量可以包括字符串、标量、对象引象或者数组。每个变量有一个 VARIANT 类型 vt，用以表明当前存贮变量的数据类型(详细数据类型定义可参见文件 comCoreTypes.idl)，数据类型的定义见 VARENUM。

```
typedef enum tagVARENUM
{
    VT_EMPTY     = 0,        // nothing
    VT_NULL      = 1,        // NULL value
    VT_I2        = 2,        // 2-byte signed int
    VT_I4        = 3,        // 4-byte signed int
    VT_R4        = 4,        // 4-byte real
    VT_R8        = 5,        // 8-byte real
    VT_CY        = 6,        // currency
    VT_DATE      = 7,        // date
    VT_BSTR      = 8,        // BSTR counted-string
    VT_DISPATCH  = 9,        // IDispatch* - not supported
    VT_ERROR     = 10,       // SCODE value
    VT_BOOL      = 11,       // true=-1, false=0
    VT_VARIANT   = 12,       // VARIANT*
    VT_UNKNOWN   = 13,       // IUnknown*
    VT_DECIMAL   = 14,       // decimal - not supported
    VT_I1        = 16,       // signed char
    VT_UI1       = 17,       // unsigned char
    VT_UI2       = 18,       // unsigned short
    VT_UI4       = 19,       // unsigned long
    VT_I8        = 20,       // 64-bit signed int
    VT_UI8       = 21,       // 64-bit unsigned int
    VT_INT       = 22,       // signed machine int
    VT_UINT      = 23,       // unsigned machine int
    VT_VOID      = 24,       // C-like void
    VT_HRESULT   = 25,       // std COM return type
    VT_PTR       = 26,       // pointer to some type
```

```
    VT_SAFEARRAY        = 27,       // not supported
    VT_CARRAY           = 28,       // not supported
    VT_USERDEFINED      = 29,       // not supported
    VT_LPSTR            = 30,       // not supported
    VT_LPWSTR           = 31,       // not supported
    VT_FILETIME         = 64,       // not supported
    VT_BLOB             = 65,       // not supported
    VT_STREAM           = 66,       // not supported
    VT_STORAGE          = 67,       // not supported
    VT_STREAMED_OBJECT  = 68,       // not supported
    VT_STORED_OBJECT    = 69,       // not supported
    VT_BLOB_OBJECT      = 70,       // not supported
    VT_CF               = 71,       // not supported
    VT_CLSID            = 72,       // not supported
    VT_VECTOR           = 0x1000,   // not supported
    VT_ARRAY            = 0x2000,   // SAFEARRAY*
    VT_BYREF            = 0x4000,   // pointer
    VT_RESERVED         = 0x8000,
    VT_ILLEGAL          = 0xffff,
    VT_ILLEGALMASKED    = 0xfff,
    VT_TYPEMASK         = 0xfff
} VARENUM;
```

VARIANT 数据结构及表示的数据类型：

```
typedef [wire_marshal(variant)] struct tagVARIANT
{
    VARTYPE vt;
    WORD wReserved1;
    WORD wReserved2;
    WORD wReserved3;
    union
    {
        LONG          lVal;           //VT_I4
        BYTE          bVal;           //VT_UI1
        SHORT         iVal;           //VT_I2
        FLOAT         fltVal;         //VT_R4
        DOUBLE        dblVal;         //VT_R8
        VARIANT_BOOL  boolVal;        //VT_BOOL
        SCODE         scode;          //VT_ERROR
        CY            cyVal;          //VT_CY
        DATE          date;           //VT_DATE
        BSTR          bstrVal;        //VT_BSTR
        IUnknown*     punkVal;        //VT_UNKNOWN
        //IDispatch*  pdispVal;       //VT_SAFEARRAY
        struct tagSAFEARRAY* parray; // no need to fwd declare
        BYTE*         pbVal;          //VT_BYREF|VT_UI1
        SHORT*        piVal;          // VT_BYREF|VT_I2
        LONG*         plVal;          //VT_BYREF|VT_I4
        FLOAT*        pfltVal;        //VT_BYREF|VT_R4
        DOUBLE*       pdblVal;        //VT_BYREF|VT_R8
        VARIANT_BOOL* pboolVal;       //VT_BYREF|VT_BOOL
        SCODE*        pscode;         //VT_BYREF|VT_ERROR
        CY*           pcyVal;         //VT_BYREF|VT_CY
```

```
        DATE*           pdate;          //VT_BYREF|VT_DATE
        BSTR*           pbstrVal;       //VT_BYREF|VT_BST
        IUnknown**      ppunkVal;       //VT_BYREF|VT_UNKNOWN
        //IDispatch**   ppdispVal;      //VT_BYREF|VT_VARIANT
        //SAFEARRAY**   pparray;
        struct tagVARIANT* pvarVal;     // no need to fwd declare
        void*           byref;          //VT_BYREF
        char            cVal;           //VT_I1
        USHORT          uiVal;          //VT_UI2
        ULONG           ulVal;          //VT_UI4
        INT             intVal;         //VT_INT
        UINT            uintVal;        //VT_UINT
        //DECIMAL*      pdecVal;
        char*           pcVal;          //VT_BYREF|VT_I1
        USHORT*         puiVal;         //VT_BYREF|VT_UI2
        ULONG*          pulVal;         //VT_BYREF|VT_UI4
        INT*            pintVal;        //VT_BYREF|VT_INT
        UINT*           puintVal;       //VT_BYREF|VT_UINT
    };
} VARIANT;

typedef VARIANT* LPVARIANT;
typedef VARIANT VARIANTARG;
typedef VARIANT* LPVARIANTARG;
```

从中可以看出，VxWorks 下定义的 VRAIANT 数据结构定义与 VC++中 IDispatch 接口访问所使用的 VARIANT 有所不同，主要是简化了 VARIANT 的定义并删除了不支持的一些数据类型。同时 VxWorks 也提供了一些 COM 库函数实现对 VARIANT 类型变量的操作。

（1）初始化函数。

```
void VariantInit (VARIANT* v);
```

初始化 VARIANT 变量，无论当前存储何数据，均将其类型置为 VT_EMPTY。

（2）清除变量函数。

```
HRESULT VariantClear (VARIANT* v);
```

清除 VARIANT 变量内容，返回到 VT_EMPTY 状态，考虑到当前值，如果 VARIANT 表示是指向 IUnknown 的指针，本函数将调用 Release()函数进行释放；如果是 BSTR 和 SAFEARRAY 会释放其所占用的内存空间。成功时返回 S_OK，v 为空或者无效时返回 E_INVALIDARG，如果 VARIANT 变量类型不是如下类型时返回 DISP_E_BADVARTYPE。

VT_EMPTY, VT_NULL, VT_I2, VT_I4, VT_R4, VT_R8, VT_CY, VT_DATE, VT_BSTR, VT_ERROR, VT_BOOL, VT_UNKNOWN, VT_UI1.

（3）复制变量函数。

```
HRESULT VariantCopy (VARIANT* d, VARIANT* s);
```

该函数将释放目标变量，复制到源变量。如果源变量是 BSTR，则产生 BSTR 的一个复制，不支持 SAFEARRAY；如果是试图复制该类型会产生 E_INVALIDARG 错误。成功时返回 S_OK，如果 s 和 d 有一个为 NULL 或者无效时返回 E_POINTER；如果没有足够内存空间分配新的 VARIANT 变量时返回 E_OUTOFMEMORY。

（4）变量类型转换函数。

```
HRESULT VariantChangeType(VARIANT* d,VARIANT* s,USHORT wFlags,VARTYPE vt);
```

该函数仅支持如下类型的转换：

```
VT_BOOL, VT_UI1, VT_I2, VT_I4, VT_R4, VT_R8, VT_BSTR.
```

成功时返回 S_OK，源和目标不是有效 VARIANT 变量时返回 E_INVALIDARG；如果 VARTYPE 不是所支持转换类型时返回 DISP_E_BADVARTYPE。

VxWorks 下的 VARIANT 类

为了更好地对 VARIANT 类型的数据进行操作，VxWorks 下提供了 CComVariant 类，该类由 tagVARIANT 派生，详细可参见 comObjLib.h 文件。

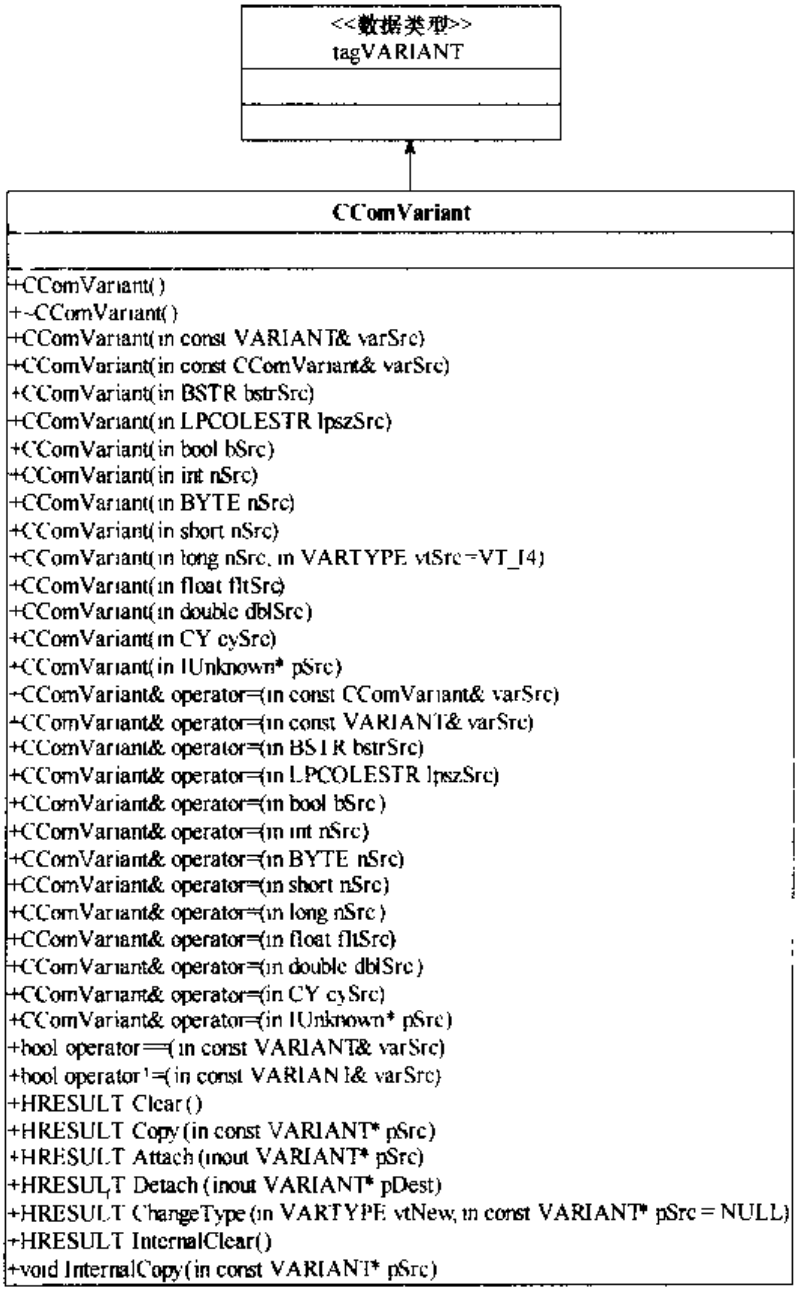


图 9-1　VARIANT 变量类

构造函数：

```
CComVariant()
CComVariant(const VARIANT& varSrc)
CComVariant(const CComVariant& varSrc)
CComVariant(BSTR bstrSrc)
CComVariant(LPCOLESTR lpszSrc)
CComVariant(bool bSrc)
CComVariant(int nSrc)
CComVariant(BYTE nSrc)
CComVariant(short nSrc)
CComVariant(long nSrc, VARTYPE vtSrc = VT_I4)
CComVariant(float fltSrc)
CComVariant(double dblSrc)
CComVariant(CY cySrc)
CComVariant(IUnknown* pSrc)
```

析构函数：

```
~CComVariant()
```

运算符重载函数：

```
CComVariant& operator=(const CComVariant& varSrc)
CComVariant& operator=(const VARIANT& varSrc)
CComVariant& operator=(BSTR bstrSrc)
CComVariant& operator=(LPCOLESTR lpszSrc)
CComVariant& operator=(bool bSrc)
CComVariant& operator=(int nSrc)
CComVariant& operator=(BYTE nSrc)
CComVariant& operator=(short nSrc)
CComVariant& operator=(long nSrc)
CComVariant& operator=(float fltSrc)
CComVariant& operator=(double dblSrc)
CComVariant& operator=(CY cySrc)
CComVariant& operator=(IUnknown* pSrc)
bool operator==(const VARIANT& varSrc)
bool operator!=(const VARIANT& varSrc)
```

清除函数（Clear）：

```
HRESULT Clear()
```

复制函数（Copy）：

```
HRESULT Copy(const VARIANT* pSrc)
```

Attach 函数：

```
HRESULT Attach(VARIANT* pSrc)
```

Detach 函数：

```
HRESULT Detach(VARIANT* pDest)
```

改变类型函数：

```
HRESULT ChangeType(VARTYPE vtNew,const VARIANT* pSrc = NULL)
```

9.1.2 BSTR 串（BSTR, BSTR *）

在 C++中字符串通常以 NULL（'\0'）结束符结尾的 ASCII 或 UNICODE 字符数组形式进行存储的。为了使不同语言生成的组件能够交换字符串，微软定义了一种新的字符串，即 BASIC 串（BSTR），BSTR 的规则是以长度为前缀（4 个字节），以 NULL 结束的 UNICODE 字符数组。

BSTR 是宽字符串，使用 16 位无符号整数来创建字符串，COM 的 API 函数使用宽字符串做为参数，字符串的传递通常采用 BSTR。可通过修改操作系统 DCOM 的 BSTR 配置选项来设定 BSTR 是使用宽字符串还是使用 8 位 ASCII 字符串来访问。如果需要实现两种类型字符串之间的转换，VxWorks 的 COM 库提供了以下两个转换函数：

（1）将宽字符转换为 ASCII。

```
comWideToAscii()
```

（2）将 ASCII 转换为宽字符串。

```
comAsciiToWide()
```

VxWorks 提供了关于 BSTR 的 COM 库 API 函数实现对 BSTR 的操作，它们分别是：

（1）SysAllocString 函数。

SysAllocString 函数分配一个新的字符串并将现有字符串拷入到新字符串中。

```
BSTR SysAllocString(const OLECHAR* psz)
```

指向新字符串 BSTR 的指针，当 psz 为 NULL 或者没有空间可以分配时返回空字符串（NULL）。

（2）SysAllocStringLen 函数。

SysAllocStringLen 函数创建给定长度的字符串并将输入的字符串拷入到新分配的字符串 BSTR 中。

```
BSTR SysAllocStringLen(const OLECHAR* psz,unsigned long  nLen)
```

创建指定长度的 BSTR 串的 API 函数，分配后，长度为 nLen 的字符从 psz 中复制到分配的 BSTR 串中，并在 BSTR 串后添加 NULL 字符，指向新字符串的指针或者无法分配内存时返回 NULL。

（3）SysAllocStringByteLen 函数。

SysAllocStringByteLen 函数分配指定长度的字符串并用传递的字符串初始化 BSTR。

```
BSTR SysAllocStringByteLen(const char *psz,unsigned long nBytes)
```

该 API 函数用于创建指定长度的 BSTR 串，包括 8 位数据。参数 bytes 表示从 psz 复制的字节数目，随后添加 2 个 NULL 字符或者 1 个 NULL OLECHAR，分配空间为 bytes+2 个字节。指向新字符串的指针，无法分配空间时返回 NULL。

（4）SysFreeString 函数。

SysFreeString 函数释放分配给 BSTR 字符串的内存。

```
HRESULT SysFreeString(BSTR bstr)
```

释放 BSTR 的 API 函数，输入参数 bstr 可以为 NULL.

（5）SysStringByteLen 函数。

SysStringByteLen 函数用于计算 BSTR 字符串中的字节数。

```
DWORD SysStringByteLen (BSTR bstr)
```

计算 BSTR 字符串中字节数（不是宽字符的数目），当 BSTR 为 NULL 时返回 0。

（6）SysStringLen 函数。

SysStringLen 函数计算 BSTR 字符串的长度。

```
DWORD SysStringLen( BSTR bstr)
```

计算 BSTR 中 OLECHAR 的数目。

为了实现对 BSTR 类型的操作，VxWorks 提供了类 CComBSTR（comObjLib.h 文件中）和类 VxComBSTR（comObjLibEx.h 文件中）。

（1）CComBSTR 提供基本 BSTR 的操作，例如：创建、赋值、转换和释放。

1）构造函数。

```
CComBSTR ()
explicit CComBSTR (int nSize, LPCOLESTR sz = 0)
explicit CComBSTR (LPCOLESTR psz)
explicit CComBSTR (const CComBSTR& src)
```

2）析构函数。

```
~CComBSTR ()
```

3）运算符重载函数。

```
CComBSTR& operator= (const CComBSTR& cbs)
CComBSTR& operator= (LPCOLESTR pSrc)
operator BSTR () const
BSTR * operator& ()
bool operator! () const
CComBSTR& operator+= (const CComBSTR& cbs)
```

4）计算长度函数（Length）。

```
unsigned int Length () const
```

5）复制函数（Copy）。

```
BSTR Copy() const
```

6）添加函数。

```
void Append (const CComBSTR& cbs)
void Append  (LPCOLESTR lpsz)
void AppendBSTR (BSTR bs)
void Append (LPCOLESTR lpsz, int nLen)
```

7）空函数（Empty）。

```
void Empty ()
```

8）Attach 函数。

```
void Attach (BSTR src)
```

9）Detach 函数。

```
BSTR Detach ()
```

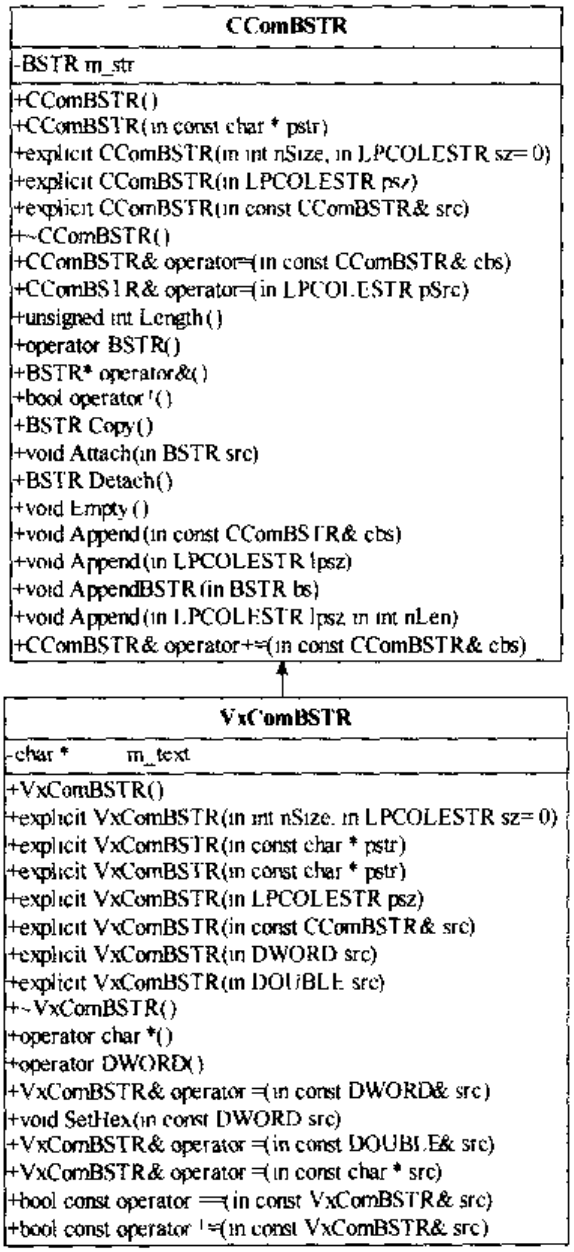


图 9-2　BSTR 字符串类

（2）VxComBSTR 类。VxComBSTR 由类 CComBSTR 派生并扩充了其功能。

1）构造函数。

除了派生 CComBSTR 构造函数，还有：

```
VxComBSTR ()
explicit VxComBSTR (int nSize, LPCOLESTR sz = 0)
explicit VxComBSTR (const char * pstr)
explicit VxComBSTR (LPCOLESTR psz)
explicit VxComBSTR (const CComBSTR& src)
explicit VxComBSTR (DWORD src)
explicit VxComBSTR (DOUBLE src)
```

2）析构函数。

```
~VxComBSTR ()
```

3）运算符重载函数。

```
operator char * ()
operator DWORD ()
VxComBSTR& operator = (const DWORD& src)
VxComBSTR& operator = (const DOUBLE& src)
VxComBSTR& operator = (const char * src)
bool const operator == (const VxComBSTR& src)
bool const operator != (const VxComBSTR& src)
```

4）SetHex 函数。

```
void SetHex (const DWORD src)
```

9.1.3　安全数组 SAFEARRAY

```
COM Support
```

（1）在 COM 下使用 SAFEARRAY，仅支持如下特性的多维安全数组：

VT_UI1，VT_I2，VT_I4，VT_R4，VT_R8，VT_ERROR，VT_CY，VT_DATE，VT_BOOL，VT_UNKNOWN，VT_VARIANT，VT_BSTR。

（2）在 DCOM 下使用 SAFEARRAY，仅支持一维安全数组，其支持的类型如下：

VT_UI1，VT_I2，VT_I4，VT_R4，VT_R8，VT_ERROR，VT_CY，VT_DATE，VT_BOOL，VT_BSTR。

（3）VxDCOM 支持 SAFEARRAY 的长度不超过 16KB。

微软 DCOM 数据包长度可超过 4 KB 字节，从微软平台发送 SAFEARRAY 时，其发送的长度应不超过 4KB 字节。

```
typedef enum tagFADF_TYPE
{
    FADF_AUTO           = 0x0001,
    FADF_STATIC         = 0x0002,
    FADF_EMBEDDED       = 0x0004,
    FADF_FIXEDSIZE      = 0x0010,
    FADF_RECORD         = 0x0020,
    FADF_HAVEIID        = 0x0040,
    FADF_HAVEVARTYPE    = 0x0080,
    FADF_BSTR           = 0x0100,
    FADF_UNKNOWN        = 0x0200,
    FADF_DISPATCH       = 0x0400,
    FADF_VARIANT        = 0x0800,
    FADF_RESERVED       = 0xF008
} FADF_TYPE;

typedef struct tagSAFEARRAYBOUND
{
    ULONG cElements;
```

```
    LONG  lLbound;
} SAFEARRAYBOUND;

typedef SAFEARRAYBOUND * LPSAFEARRAYBOUND;

typedef struct tagSAFEARRAY
{
    USHORT          cDims;
    USHORT          fFeatures;
    ULONG           cbElements;
    ULONG           cLocks;
    PVOID           pvData;
    [size_is(cDims)]SAFEARRAYBOUND  rgsabound[];
} SAFEARRAY;
```

（1）创建新的 SAFEARRAY 变量。

```
SAFEARRAY *SafeArrayCreate
(
    VARTYPE vt,
    UINT cDims,
    SAFEARRAYBOUND* rgsabound
);
```

类型 vt 是数组的基类型，不能有 VT_ARRAY 和 VT_BYREF 标志，VT_EMPTY 和 VT_NULL 也是无效的，其他类型均可有效。该函数创建一个新数组，为该数组分配内存并完成初始化，返回指向新数组的指针。数组在创建时应具有 FADF_HAVEVARTYPE|FADF_FIXEDSIZE|FADF_AUTO 特征，并设定相应的 BSTR，VARIANT 或者 IUNKNOWN 位。

（2）删除 SAFEARRAY 变量。

```
HRESULT SafeArrayDestroy(SAFEARRAY *psa);
```

该函数删除现有 SAFEARRAY 变量以及数组中的数据，如果数组中存储的是复杂数据类型，会调用相应的内存释放函数。成功时返回 S_OK，如果 SAFEARRAY 处于锁定状态返回 DISP_E_ARRAYISLOCKED，参数无效时则返回 E_INVALIDARG，psa 由 SafeArrayCreate 函数创建。

（3）SAFEARRAY 加锁函数。

```
HRESULT SafeArrayLock (SAFEARRAY * psa);
```

该函数完成 SAFEARRAY 锁计数器加 1 操作。成功时返回 S_OK，SAFEARRAY 无效时返回 E_INVALIDARG。

（4）SAFEARRAY 解锁函数。

```
HRESULT SafeArrayUnlock (SAFEARRAY * psa);
```

该函数完成 SAFEARRAY 锁计数器减 1 操作。成功时返回 S_GK，SAFEARRAY 无效时返回 E_INVALIDARG，计数器为 0 时如果试图执行该操作时返回 E_UNEXPECTED。

（5）设定安全数组指定位置数据。

```
HRESULT SafeArrayPutElement
(
    SAFEARRAY * psa,
    long *      rgIndicies,
    void *      pv
)
```

该函数将一个数据元素复制到安全数组指定的位置，它在赋值前后调用 SafeArrayLock 和 SafeArrayUnlock 函数。数据元素是字符串、对象或者 VARIANT 变量时，函数也可完成正确复制，如果安全数组当前元素为字符串，对象或者 VARIANT 变量时可以完成正确清除工作。成功时返回 S_OK，指定索引无效时返回 DISP_E_BADINDEX，任一参数无效时返回 E_INVALIDARG，无法为元素分配内存时返回 E_OUTOFMEMORY。

（6）获取安全数组指定位置数据。

```
HRESULT SafeArrayGetElement
(
    SAFEARRAY * psa,
    long * rgIndicies,
    void *    pv
)
```

遍历查找安全数组中的一个元素。该函数在查找元素前后自动调用 SafeArrayLock 和 SafeArrayUnlock 函数，调用者必须提供所要查找数据的正确大小存储空间。如果数据元素为字符串、对象或者 VARIANT 变量，该函数能够正确复制元素。成功时返回 S_OK，指定索引无效时返回 DISP_E_BADINDEX，参数无效时返回 E_INVALIDARG，无法分配内存空间时返回 E_OUTOFMEMORY。

（7）获得安全数组中的全部数据。

```
HRESULT SafeArrayAccessData(SAFEARRAY *psa,void **ppvData);
```

该函数对安全数组的锁计数器加 1，并返回指向数组数据的指针。数组返回时的格式如下：

```
p=(i[0]+(d[0]*i[1])+(d[0]*d[1]*i[2])...)*元素大小
```

其中：

i[n]——下标；

d[n]——维数；

p——存储位置的指针。

成功时返回 S_OK，参数无效时返回 E_INVALIDARO。

（8）释放 SafeArrayAccessData 函数调用锁计数器。

```
HRESULT SafeArrayUnaccessData(SAFEARRAY * psa);
```

该函数减少由函数 SafeArrayAccessData 的锁计数器。成功时返回 S_OK，参数无效时返回 E_INVALIDARG。

（9）拷贝安全数组。

```
HRESULT SafeArrayCopy(SAFEARRAY *psa,SAFEARRAY **ppsaOut);
```

该函数复制现有安全数组 psa，并产生当前数组的拷贝 ppsaOut。成功时返回 S_OK，参数

无效时返回 E_INVALIDARG，无法分配内存空间时返回 E_OUTOFMEMGRY。

（10）获得安全数组指定维的下限。

```
HRESULT SafeArrayGetLBound(SAFEARRAY *psa,unsigned intnDim,long *plLbound);
```

得到安全数组指定维的下限值。成功时返回 S_OK，指定的维数越界时返回 DISP_E_BADINDEX，参数无效时返回 E_INVALIDARG。

（11）获得安全数组指定维的上限。

```
HRESULT SafeArrayGetUBound(SAFEARRAY *psa,unsigned int nDim,long *plUbound);
```

得到安全数组指定维的上限值。成功时返回 S_OK，指定的维数越界时返回 DISP_E_BADINDEX，参数无效时返回 E_INVALIDARG。

（12）返回安全数组的维数。

```
UINT SafeArrayGetDim(SAFEARRAY *psa);
```

（13）返回安全数组某一元素的大小（字节数）。

```
UINT SafeArrayGetElemsize(SAFEARRAY *psa);
```

（14）返回存储在安全数组中元素的类型。

```
HRESULT SafeArrayGetVartype(SAFEARRAY *psa,VARTYPE *pvt);
```

成功时返回 S_OK，参数无效时返回 E_INVALIDARG。

SafeArrayPutElement 和 SafeArrayGetElement 函数的局限性在于它们每次只能访问安全数组中的一个元素，在处理大量安全数组的元素时为降低性能，这时，可采用 SafeArrayAccessData 和 SafeArrayUnaccessData 函数。

9.2 非自动化数据类型

枚举类型定义可参见 SAFEARRAY 中的相关类型。widl 也支持非自动化类型，它们是：

（1）简单数组（simple array）。

数组有固定长度，例如：

IDL 定义：

```
HRESULT Sum([in]double dbMyValue[10],[out]double* dbRet);
```

Client 代码：

```
double xx[10]={1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0};
double ret;
pClient->Sum(xx,&ret);
```

server 代码：

```
HRESULT XXXX::Sum(double dbMyValue[10],double *dbRet);
{
    ...
}
```

（2）固定长度结构（fixed-size structure）。

结构的成员是自动化类型、简单数组或者其他固定长度结构。

例如，如下函数实现计算平面坐标上三个点所组成的三角形的面积。

IDL 定义：

```
typedef struct tagPOINT
{
  double x;
  double y;
}POINT;
HRESULT TriangleArea([in]POINT sPoint[3],[out]double *dbRet);
```

Client 代码：

```
POINT xx[3]={{0.0,0.0},{1.0,0.0},{0.0,1.0}};
double ret;
pClient->(xx,&ret);
```

server 代码：

```
HRESULT XXXX::Sum(POINT sPoint[3],double *dbRet);
{
    ...
}
```

（3）一致性数组（conformant array）。

数组的长度在运行时由另一参数、结构域或者表达式决定，使用 size_is 属性。

表 9-1　接口定义中一致性数组的属性

IDL 中的属性	说　　明
first_is	第一个传送数组元素的索引
last_is	最后一个传送数组元素的索引
length_is	传送数组元素的总数
max_is	传送数组元素的最大有效值
min_is	传送数组元素的最小有效值
size_is	传送数组元素的分配的总数目

例如：

IDL 定义：

```
HRESULT  Sum([in]int  nLen,[in,size_is(nLen)]double  *dbMyValue,[out]double*
dbRet);
```

Client 代码：

```
double xx[10]={1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0};
double ret;
pClient->Sum(10,xx,&ret);
```

server 代码：

```
HRESULT XXXX::Sum(int nLen,double *dbMyValue,double *dbRet);
```

```
{
    ...
}
```

（4）一致性结构（conformant structure）。

结构字节可变，最终成员是一致性数组，长度由结构的另一成员决定。

IDL 定义：

```
typedef struct tagSUM_STRUCT
{
   int nLen;
   [size_is(nLen)]double *dbVal;
}SUM_STRUCT;
    HRESULT Sum([in]SUM_STRUCT str,[out]double* dbRet);
```

Client 代码：

```
double xx[10]={1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0};
SUM_STRUCT str;
str.nLen = sizeof(xx)/sizeof(double);
str.dbVal=xx;
double ret;
pClient->Sum(str,&ret);
```

server 代码：

```
HRESULT XXXX::Sum(SUM_STRUCT str,double *dbRet);
{
    ...
}
```

（5）string。

指向有效字符类型（单字节或者对字节类型），使用[string]属性。

```
typedef struct tagSTRING_STRUCT
{
     [string] LPWSTR szSource;
     [string] LPWSTR szDest;
}STRING_STRUCT;
HRESULT StringOp1(
     [in, string] LPWSTR szComment,
     [in, size_is(dwCount)] LPWSTR* pszSource);
HRESULT StringOp2([in]STRING_STRUCT str);
HRESULT StringOp3(
    [in] DWORD dwCount,
    [out, string, size_is(,dwCount)] LPWSTR ** ppszNewItemIDs);
```

9.3 HRESULT 说 明

VxWorks 在文件 comCoreTypes.idl 定义了 HRESULT 类型，是一个 32 位的 long 型。通常情况下 COM 接口方法返回值类型为 HRESULT 类型，HRESULT 主要包括三部分组成：

（1）严重性代码（severity）。

（2）设备代码（facility）。

（3）状态代码（status code）。

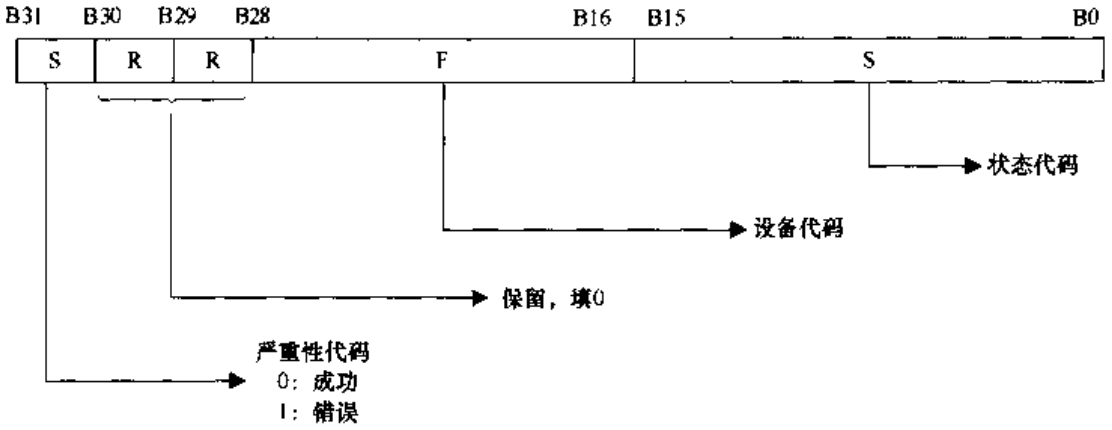


图 9-3　HRESULT 位定义说明

最高位 B31 定义了 HRESULT 的严重性，该位为 0 表示成功，为 1 表示有错误。B30 位还可用做用户自定义错误代码标志，为 1 表示是用户自定义错误代码。VxWorks 在 comErr.h 文件中定义了错误代码及相关的宏定义。

设备代码占用低 16 位，如表 9-2 所示。

表 9-2　HRESULT 设备代码的值及其含义

定　义	值	含　义
FACILITY_NULL	0	用于一般性错误，例如 S_OK
FACILITY_RPC	1	远程程序调用（RPC）返回的错误
FACILITY_DISPATCH	2	IDispatch 接口返回的错误
FACILITY_STORAGE	3	IStorage 或者 IStream 接口返回的错误
FACILITY_ITF	4	用户自定义接口返回的错误
FACILITY_WIN32	7	来自 Win32 API 的错误
FACILITY_WINDOWS	8	来自标准接口的错误
FACILITY_SSPI	9	来自 SSPI 的错误
FACILITY_CONTROL	10	用于 OLE 控件有关错误代码
FACILITY_CERT	11	
FACILITY_INTERNET	12	

VxWorks 定义：

```
#define FACILITY_NULL           0
#define FACILITY_RPC            1
#define FACILITY_DISPATCH       2
#define FACILITY_STORAGE        3
#define FACILITY_ITF            4
```

```
#define FACILITY_WIN32          7
#define FACILITY_WINDOWS        8
#define FACILITY_SSPI           9
#define FACILITY_CONTROL        10
#define FACILITY_CERT           11
#define FACILITY_INTERNET       12
```

状态代码占用高 16 位的低 13 位，主要用来说明错误对应的代码。

关于 HRESULT 的一些宏定义

```
#define SUCCEEDED(hr)           ((HRESULT) (hr) >= 0)
#define FAILED(hr)              ((HRESULT) (hr) < 0)
#define HRESULT_CODE(hr)        ((hr) & 0xFFFF)
#define HRESULT_FACILITY(hr)    (((hr) >> 16) & 0x1FFF)
#define HRESULT_SEVERITY(hr)    (((hr) >> 31) & 0x01)

/* How to make your own HRESULT */
#define MAKE_HRESULT(sev,fac,code) \
    ((HRESULT)  (((unsigned long)(sev)<<31) |   \
                ((unsigned long)(fac)<<16) |    \
                ((unsigned long)(code))) )
```

对 HRESULT 的判断建议用户使用 SUCCEEDED 和 FAILED，不要直接判断是否为 S_OK 或者 S_FALSE。状态代码、设备代码和严重性代码都可通过宏 HRESULT_CODE、HRESULT_FACILITY 和 HRESULT_SEVERITY 获得。

9.4 VxWorks 接 口

在 comCoreTypes.idl 文件中 VxWorks 定义了一些基本接口，微软定义了 200 多个接口，但 VxCOM/VxDCOM 仅定义了有限的接口，目的是为了兼容 COM/DCOM。每一个接口都提供一个全局唯一标识符（UUID，unique identifier），UUID 是一个 128 位的数字，以 16 进制表示，格式为“8-4-4-12”。

9.4.1 IUnknown 接口

VxWorks 用户接口通常派生于 IUnknown 接口，IUnknown 接口的 UUID 是：00000000-0000-0000-C000-000000000046，IUnknown 接口定义如下：

```
[
  local,
  object,
  uuid(00000000-0000-0000-C000-000000000046),
  pointer_default(unique)
]
interface IUnknown
{
    typedef [unique] IUnknown *LPUNKNOWN;
    HRESULT QueryInterface(
        [in] REFIID riid,
```

```
        [out, iid_is(riid)] void **ppvObject);
    ULONG AddRef();
    ULONG Release();
}
```

IUnknown 接口一共提供了 3 个方法：QueryInterface、AddRef 和 Release。QueryInterface 确定对象是否支持用户所需要的接口，如果是则指向用户接口的指针，否则返回错误码 HRESULT。AddRef 和 Release 方法完成计数工作，确定对象何时可以从内存中释放，对每个接口指针，必须在调用其他方法之间调用 AddRef，在使用完接口指针后必须调用 Release，Release 并不是消毁对象，而是减少引用计数器，当该计数器减到 0 时才消毁对象。

9.4.2　IClassFactory 接口

类厂（IClassFactory）的 UUID 为：00000001-0000-0000-C000-000000000046，其定义如下：

```
[
    object,
    local,
    uuid(00000001-0000-0000-C000-000000000046),
    pointer_default(unique)
]

interface IClassFactory : IUnknown
{
    typedef [unique] IClassFactory * LPCLASSFACTORY;

    HRESULT CreateInstance
    (
        [in, unique] IUnknown * pUnkOuter,
        [in] REFIID riid,
        [out, iid_is(riid)] void **ppvObject
    );

    HRESULT LockServer ([in] BOOL fLock);
}
```

类厂主要实现对象实例化工作，用户调用 CoCreateInstance 时，该函数内部调用了 IClassFactory 接口的方法，实现对象的创建。IClassFactory 包括两个方法：CreateInstance 和 LockServer。CreateInstance 实现指定类标识符（CLSID）的初始化工作，LockServer 则锁定内存中的对象服务器，允许快速创建新的对象。

9.4.3　IMultiQI 接口

IMultiQI 接口允许客户在同一时间内完成服务器多个接口远程 DCOM 访问操作，这样用户不必要每次调用 IUnknown 的 QueryInterface 得到一个接口指针，减少了访问 RPC 的次数和时间，其定义如下：

```
[
    object,
    local,
```

```
    uuid(00000020-0000-0000-C000-000000000046)
]

interface IMultiQI : IUnknown
{
    typedef [unique] IMultiQI* LPMULTIQI;

    typedef struct tagMULTI_QI
    {
        const IID  *pIID;
        IUnknown   *pItf;
        HRESULT     hr;
    } MULTI_QI;

    HRESULT QueryMultipleInterfaces
     (
        [in]      ULONG      cMQIs,
        [in,out] MULTI_QI  *pMQIs
    );
}
```

IMultiQI 接口只有一个方法 QueryMultipleInterfaces，第一个参数说明了接口的数目；第二个参数为一致性结构，每个结构对应一个接口的数据结构，其输入为接口标识符（IID），输出为指向该接口的指针及其返回的 HRESULT 值。

9.4.4 IRegistry 接口

IRegistry 接口主要是内部使用，由 comCoreLib 和 VxCOM 实现，主要是完成 COM 类在 VxWorks 操作系统下的注册工作。

```
[
    uuid(a01ccc02-cfeb-4311-8ebc-d7edb4a8de9c),
    pointer_default(unique),
    local,
    object
]
interface IRegistry : IUnknown
{
    HRESULT RegisterClass
        (
        [in] REFCLSID       clsid,
        [in] void *         pfnGetClassObject
        );

    HRESULT IsClassRegistered
        (
        [in] REFCLSID       clsid
        );

    HRESULT CreateInstance
```

```
        (
        [in] REFCLSID        clsid,
        [in] IUnknown *      pUnkOuter,
        [in] DWORD           dwClsContext,
        [in] const char *    hint,
        [in] ULONG           cMQIs,
        [in,out] MULTI_QI *  pMQIs
        );

    HRESULT GetClassObject
        (
        [in] REFCLSID        clsid,
        [in] REFIID          iid,
        [in] DWORD           dwClsContext,
        [in] const char *    hint,
        [out, iid_is(iid)] IUnknown ** ppClsObj
        );

    HRESULT GetClassID
        (
        [in] DWORD           dwIndex,
        [out] LPCLSID        pclsid
        );

};
```

IRegistry 接口共有 5 个方法：①方法 RegisterClass 完成由 GetClassObject()函数返回的类对象指针和指定类标识符（CLSID）在注册表中的注册工作。②方法 IsClassRegistered 判定类标识符是否在注册表中，是就返回 S_OK，否则返回 S_FALSE。③方法 CreateInstance 为指定类标识符、指定接口标识符，必要的参数的类进行实例化工作，并返回新对象的接口指针。④方法 GetClassObject 指定类标识符的类对象一个实例，参数中包括接口标识符和返回的类对象，接口标识符通常情况下可设定为 IClassFactory。⑤方法 GetClassID 允许客户得到注册表中指定位置的类标识符（CLSID），参数 dwIndex 标识注册表的位置，超出范围时返回 E_FAIL。

9.4.5　枚举器 IEnumXXXX 接口

枚举器接口实现对象序列的遍历操作。在 VxWorks 下可以使用枚举器对一组对象进行枚举，根据所要枚举对象类型的不同，VxWorks 下提供了不同的接口，主要包括：IEnumGUID、IEnumUnkown 和 IEnumString。每一个枚举器均有相同的 4 个方法：Next、Skip、Reset 和 Clone。方法 Next 遍历枚举序列中下一指定数据的数据项，它是所有枚举器的核心，第一个参数是要获取的项的总数目，实际获得项的总数目放在第三个参数中，第二个参数用来存放枚举器存放枚举项的数据。①方法 Skip 略过指定数目的枚举序列，成功时返回 S_OK，否则返回 S_FALSE；②方法 Reset 使枚举序列指向开始第一个元素；③方法 Clone 创建一个枚举器，与当前枚举器包括相同枚举状态，使用该函数，客户方能够记录枚举序列中特定点，新的枚举器与原来枚举器支持相同的接口。

```
[
    object,
    uuid(0002E000-0000-0000-C000-000000000046),
    pointer_default(unique)
]
interface IEnumGUID : IUnknown
{
    typedef [unique] IEnumGUID *LPENUMGUID;

    HRESULT Next
    (
    [in] ULONG celt,
    [out, size_is(celt), length_is(*pceltFetched)] GUID *rgelt,
    [out] ULONG *pceltFetched
    );

    HRESULT Skip ([in] ULONG celt);

    HRESULT Reset ();

    HRESULT Clone ([out] IEnumGUID **ppenum);
}


[
    object,
    uuid(00000101-0000-0000-C000-000000000046),
    pointer_default(unique)
]

interface IEnumString : IUnknown
{

    typedef [unique] IEnumString *LPENUMSTRING;

    HRESULT Next
    (
        [in] ULONG celt,
        [out, size_is(celt), length_is(*pceltFetched)] LPOLESTR *rgelt,
        [out] ULONG *pceltFetched
    );

    HRESULT Skip ([in] ULONG celt);

    HRESULT Reset ();

    HRESULT Clone ([out] IEnumString **ppenum);
}

[
```

```
        object,
        uuid(00000100-0000-0000-C000-000000000046),
        pointer_default(unique)
    ]
    interface IEnumUnknown : IUnknown
    {

        typedef [unique] IEnumUnknown *LPENUMUNKNOWN;

        HRESULT Next(
            [in] ULONG celt,
            [out, size_is(celt), length_is(*pceltFetched)]IUnknown **rgelt,
            [out] ULONG *pceltFetched);

        HRESULT Skip ([in] ULONG celt);

        HRESULT Reset ();

        HRESULT Clone ([out] IEnumUnknown **ppenum);
}
```

9.4.6　IMalloc 接口

IMalloc 接口主要功能是分配、释放和管理内存。可通过调用 CoGetMalloc()函数返回来指向 IMalloc 接口的指针。

```
[
    object,
    local,
    uuid(00000002-0000-0000-C000-000000000046),
    pointer_default(unique)
]
interface IMalloc : IUnknown
{
    void* Alloc ([in] ULONG cb);
    void* Realloc ([in] void *pv, [in] ULONG cb);
    void  Free ([in] void *pv);
    ULONG GetSize ([in] void *pv);
    int   DidAlloc ([in] void *pv);
    void  HeapMinimize ();
};
```

9.5　虚　函　数　表

用户定义完 IDL 文件后，需将接口定义头文件映射到编程语言，其产生的存根和代理代码是一个虚函数表，对于 C++编程语言则是纯虚类，用户需要继承该类并在派生类中实现纯虚类的函数，对于标准 C，IDL 编译器将 IDL 接口转换成结构（struct），结构的成员则是指向函数的指针，VxWorks 还提供了一些宏定义用来实现这些转换。

欲定义虚函数表 vtable 结构，需要使用宏 COM 虚函数表开始（COM_VTBL_BEGIN）、COM 虚函数表结束（COM_VTBL_END）以及 COM 虚函数表项（COM_VTBL_ENTRY）。例如定义 IFooVtbl 的虚函数结构使用如下方法：

```
typedef struct
{
   COM_VTBL_BEGIN
   COM_VTBL_ENTRY (HRESULT, Method1, (int x));
   COM_VTBL_ENTRY (HRESULT, Method2, (long x));
   COM_VTBL_END
} IFooVtbl;
```

由于虚函数表是一个数据结构，因此其不占用内在空间。

创建常虚函数表，可采用如下方法：

```
COM_VTABLE(IFoo) IFoo_proxy_vtbl = {
   COM_VTBL_HEADER
   COM_VTBL_METHOD (&IFoo_Method1_proxy),
   COM_VTBL_METHOD (&IFoo_Method2_proxy)
};
```

VxWorks 定义了一些宏，实现虚函数到实现函数的映射。

```
#define STDMETHODCALLTYPE       STDCALL
#define STDMETHODIMP            HRESULT STDMETHODCALLTYPE
#define STDMETHODIMP_(type)     type STDMETHODCALLTYPE

#define STDMETHOD(method) virtual HRESULT STDMETHODCALLTYPE method
#define STDMETHOD_(type,method) virtual type STDMETHODCALLTYPE method
```

标准 C 不支持面向对象技术，将接口映射为结构，以 Iunknown 接口为例，其定义的数据结构为：

```
typedef struct
{
   COM_VTBL_BEGIN
   COM_VTBL_ENTRY (HRESULT, QueryInterface, (IUnknown* pThis, REFIID riid,
void** ppvObject));

#define IUnknown_QueryInterface(pThis, riid, ppvObject) pThis->lpVtbl->
QueryInterface(COM_ADJUST_THIS(pThis), riid, ppvObject)

   COM_VTBL_ENTRY (ULONG, AddRef, (IUnknown* pThis));

#define IUnknown_AddRef(pThis) pThis->lpVtbl->AddRef(COM_ADJUST_THIS(pThis))

   COM_VTBL_ENTRY (ULONG, Release, (IUnknown* pThis));

#define IUnknown_Release(pThis) pThis->lpVtbl->Release(COM_ADJUST_THIS(pThis))

   COM_VTBL_END
} IUnknownVtbl;
```

如果使用 C++，则映射为纯虚类，需要用户继承该类。

```
interface IUnknown
{
virtual HRESULT QueryInterface (REFIID riid, void** ppvObject) =0;

virtual ULONG AddRef () =0;

virtual ULONG Release () =0;

};
```

第 10 章　VxDCOM 网络协议剖析

10.1　概　　述

开放软件基金组织（OSF：Open Software Foundation）定义了用于分布式计算的接口规范，制定了基于分布式系统的分布式计算环境（DCE）。远程过程调用（RPC：Remote Procedure Call）是开放组织关于 CAE（公共应用环境）的一组规范，主要用于计算机之间的通信规范的定义。

Sniffer 是 NAI 公司开发的一类功能强大的网络协议分析软件。具有如下基本功能：

（1）捕获网络流量进行详细分析。

（2）使用专家分析系统诊断网络问题。

（3）实时监听网络活动。

（4）收集网络利用率和错误统计。

（5）保存以往历史信息和错误情况。

（6）产生实时报警，故障发生时通知网络管理员。

（7）可使用工具对网络进行监测。

Sniffer 能够充分利用 Windows 32 位多任务操作系统的特性，可以同时运行多个实例，其可视化界面使得 Sniffer 简单易学。

10.1.1　IP 报的定义

IP 是 TCP/IP 协议族中最重要的协议，这是因为 TCP、UDP、ICMP、IGMP 数据报均是通过 IP 数据报格式进行传输的。目前使用的是 IPv4，IP 报通常由 IP 首部和数据部分组成，如表 10-1 所示。

表 10-1　IP 数据报格式

<table>
<tr><td colspan="4">0　　15　16　　31</td><td rowspan="6">20 字节</td></tr>
<tr><td>4 位版本
Version</td><td>4 位头长度
Header Length</td><td>8 位服务类型
Type Of Service</td><td colspan="2">16 位总长度（单位：字节）
Total Length</td></tr>
<tr><td colspan="3">16 位标识
Identification</td><td>3 位标志
Flag</td><td>13 位段偏移地址
Fragment Offset</td></tr>
<tr><td colspan="2">8 位生存时间 TTL
Time To Live</td><td>8 位协议
Protocol</td><td colspan="2">16 位头校验和
Header Checksum</td></tr>
<tr><td colspan="5">32 位源 IP 地址（Source IP Address）</td></tr>
<tr><td colspan="5">32 位目的 IP 地址（Destination IP Address）</td></tr>
<tr><td colspan="6">选项（如果有）</td></tr>
<tr><td colspan="6">数　　据</td></tr>
</table>

其中前 20 个字节为 IP 头定义。IP 的版本为 4，故称为 IPv4。4 位头长度填 5，表示 20 个字节。

IP 的 8 位协议域（Protocol field），几种常用网络报的含义如下：

（1）0x01：表示 ICMP（Internet Control Message Protocol，网间控制报文协议）报文。

（2）0x02：表示 IGMP（Internet Group Management Protocol，网间组管理报文协议）报文。

（3）0x06：表示 TCP（Transmission Control Protocol，传输控制协议）报文。

（4）0x11：表示 UDP（User Datagram Protocol，用户数据报文协议）报文。

16 位头校验和 VxWorks 下可以不填，其网络协议栈将自动计算头校验和。

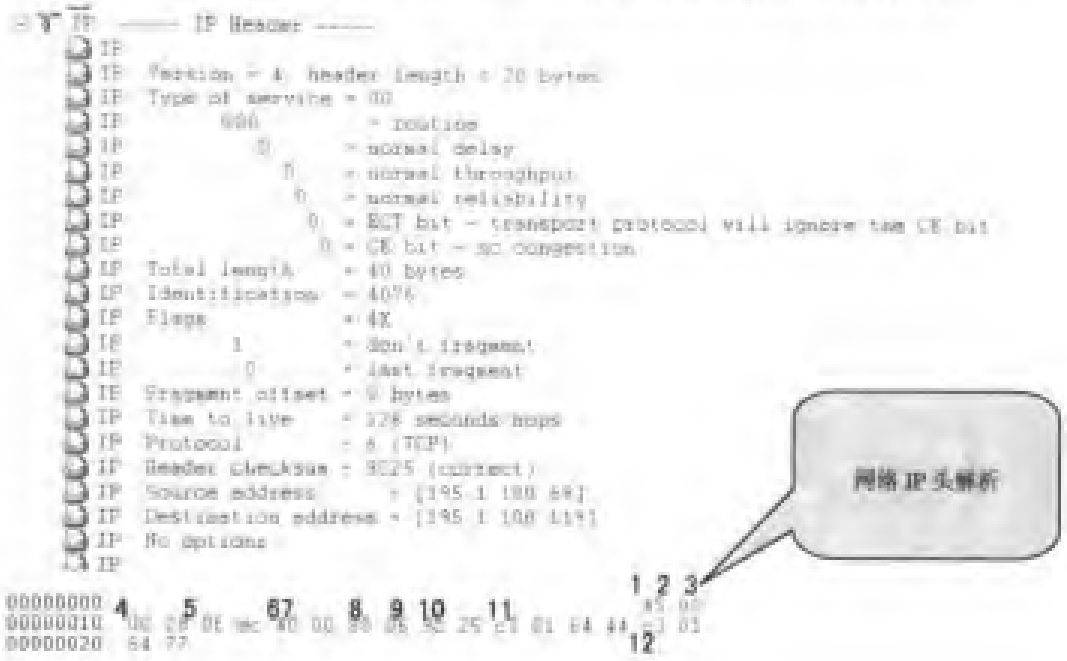


图 10-1　IP 头报文解析

1—版本（IPv4）；2—头长度（20 字节）；3—服务类型；4—总长度；5—标识；6—3 位标志；
7—13 位段偏移地址；8—生存时间（128）；9—协议（TCP）；10—头校验和；
11—源 IP 地址（195.1.100.119）；12—目的 IP 地址（195.1.100.68）

注意：IP 头中字节定义是高字节结束的，即是 Big-endian 方式，这种方式与 unix 的字节表示方式相同，但是与 Windows 平台则有所不同。

10.1.2　TCP 头定义

TCP 是面向连接的，可靠的传输协议，TCP 数据封装在一个 IP 数据报中，如图 10-2 所示。TCP 报详细定义如表 10-2 所示。

图 10-2　TCP 报定义

表 10-2 TCP 报详细定义

0		15 16						31
16 位源端口号（Source Port Number）								16 位目标端口号（Destination Port Number）
32 位序列号（Sequence Number）								
32 位确认码（Acknowledgment Number）								
4 位首部 Header Length	6 位保留 Reserved	U R G	A C K	P S H	R S T	S Y N	F I N	16 位窗口大小 Window Size
16 位校验和 TCP Checksum								16 位紧急指针 Urgent Pointer
选项（如果有）								
数据								

20 字节

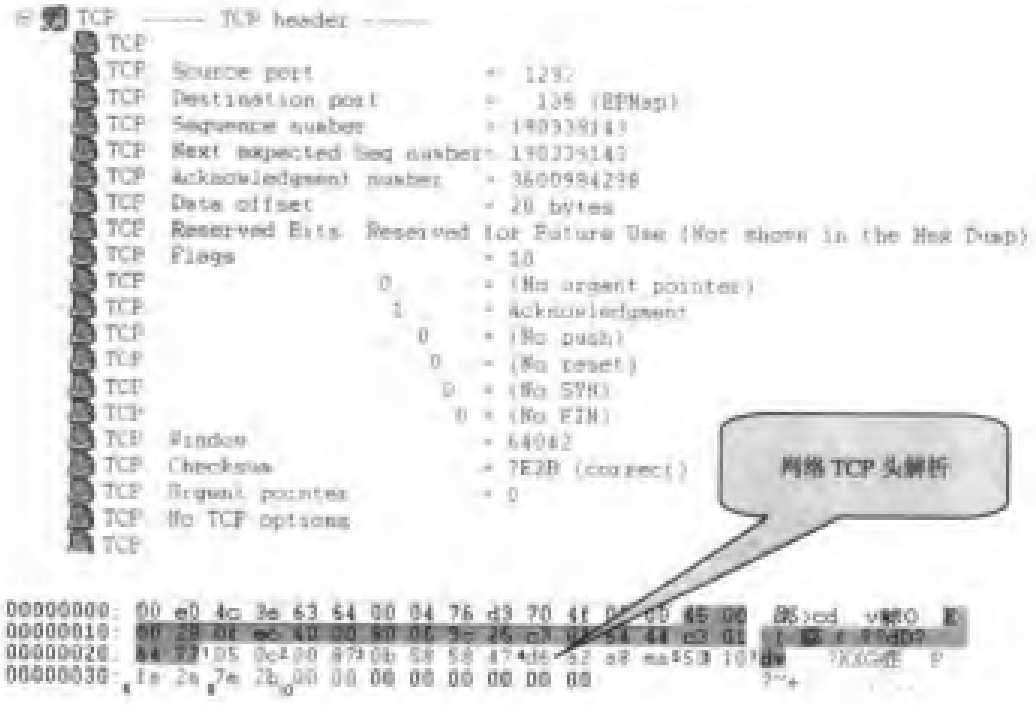


图 10-3 TCP 头网络报文解析

1—源端口号；2—目的端口号；3—序列号；4—确认号；5—TCP 头长度；6—保留；7—TCP 标志；8—窗口大小；9—TCP 校验和；10—紧急指针

10.1.3 数据报文解码

Sniffer 捕获的网络数据报采用分层方式进行处理，针对 DCOM 通信协议，主要有 4 层协议，即数据链路层（DLC）、网络层（IP）、传输层（TCP/UDP）和应用层（RPC），如图 10-4 所示。RPC 分层的表示如图 10-5 所示。

图 10-4　Sniffer 监听到 RPC 网络协议

图 10-5　RPC 网络分层

RPC 与国际标准化组织所建议的开放系统互联 OSI/RM 7 层参考模型的对应关系如图 10-6 所示。

图 10-6　RPC 4 层网络协议与 OSI/RM 7 层网络协议对照图

10.2　RPC 网络通信报文分析

在 DCOM 远程调用时，主要采用 RPC 协议，它是微软和 DEC 公司开发的一种基于 TCP/UDP 之上的网络协议，主要的 PDU 类型主要有 19 种，如表 10-3 所示。协议数据单元（PDU）

可适用于 DCOM 通信的面向连接的（CO）或者无连接（CL）。

表 10-3 DCOM 通信的 PDU 协议类型

序 号	PDU 类型	协议	类型值	传送方向
1	request	CO/CL	0	客户→服务器
2	ping	CL	1	客户→服务器
3	response	CO/CL	2	服务器→客户
4	fault	CO/CL	3	服务器→客户
5	working	CL	4	服务器→客户
6	nocall	CL	5	服务器→客户
7	reject	CL	6	服务器→客户
8	ack	CL	7	客户→服务器
9	cl_cancel	CL	8	客户→服务器
10	fack	CL	9	客户←→服务器
11	cancel_ack	CL	10	服务器→客户
12	bind	CO	11	客户→服务器
13	bind_ack	CO	12	服务器→客户
14	bind_nak	CO	13	服务器→客户
15	alter_context	CO	14	客户→服务器
16	alter_context_resp	CO	15	服务器→客户
17	shutdown	CO	17	服务器→客户
18	co_cancel	CO	18	客户→服务器
19	orphaned	CO	19	客户→服务器

无论是面向连接的或者无连接的 RPC，其 PDU 通常由 3 部分组成：

（1）PDU 头部分，主要包括协议控制信息，为必需项。

（2）PDU 体部分，主要包括数据项，仅存在于某些类型的 PDU 中，为可选项。

（3）审核验证部分，主要包括身份认证协议，依赖于 PDU 类型及其使用的审核方式，为可选项。

本章重点介绍有连接的 RPC 协议。

10.2.1 无连接 RPC PDU

无连接 PDU 同样包括 PDU 头、PDU 体以及可选的审核验证数据。①PDU 头的长度是固

定的，为 80 个字节；②PDU 体可定义为接口定义语言中 byte 类型的数组，其长度在 PDU 头中指定；③审核认证部分定义为 byte 类型的数组，其长度由 PDU 体中字段 auth_proto 决定，当该字段非 0 时表示有审查认证代码。

无连接 PDU 的 IDL 接口定义如下：

```
typedef struct {
    unsigned small rpc_vers = 4;     /* RPC 协议主版本(低 4 位有效)*/
    unsigned small ptype;            /* 报文类型(低 5 位有效) */
    unsigned small flags1;           /* 报文标志 */
    unsigned small flags2;           /* 报文标志 */
    byte          drep[3];           /* 数据表示格式*/
    unsigned small serial_hi;        /* 序列码的高字节 */
    uuid_t        object;            /* 对象标识 */
    uuid_t        if_id;             /* 接口标识 */
    uuid_t        act_id;            /* 活动标识 */
    unsigned long  server_boot;      /* 服务器引导时间 */
    unsigned long  if_vers;          /* 接口版本 */
    unsigned long  seqnum;           /* 序列码 */
    unsigned short opnum;            /* 操作码 */
    unsigned short ihint;            /* 接口提示 */
    unsigned short ahint;            /* 激活提示 */
    unsigned short len;              /* 报文长度  */
    unsigned short fragnum;          /* 帧编码 */
    unsigned small auth_proto;       /* 鉴定协议标识*/
    unsigned small serial_lo;        /* 序列码低字节 */
} dc_rpc_cl_pkt_hdr_t;
```

（1）协议版本号 rpc_vers：协议版本号低 4 位有效，无连接 PDU 时为 4。

（2）PDU 类型 ptype：字段 ptype 的低 5 位用来表示 PDU 类型，具体类型如表 10-1 所示，理论上共有 32 种 PDU 类型,但其值为表 10-1 的 19 中之一。

（3）标志（Flags）：标志字段 flags1 和 flags2 用来说明 PDU 报文的相关标志。

（4）数据表示格式：如图 10-7 所示。

（5）序列号（Serial Number）：序列号用来标志帧的传输标志。

（6）对象标识 Object Identifier：对象标识是 UUID 类型，用来唯一标识远程正在运行过程，通常该操作发生在一个对象上，如果不是对对象进行操作，该字段将为空（NIL）。

（7）接口标识 Interface Identifier：接口标志也是一个 UUID 类型，用来唯一标志调用的接口。

（8）激活标识 Activity Identifier：激活标识也是一个 UUID，用来唯一标识客户激活一个远程调用，服务器方可以使用激活 UUID 做为服务器与客户端的通信密钥。

（9）服务引导时间（Server Boot Time）：服务引导时间是一个 32 位无符号整数，用于标识服务器启动的时间，该时间为服务器运行进程创建的时间，其格式是从 1970 年 1 月 1 日起的时间，单位为秒。

（10）接口版本（Interface Version）：接口版本是一个 32 位无符号整数，用于标识所调用

接口听版本号，该字段允许服务器执行一个接口的多个版本。服务方使用对象 UUID、接口 UUID、接口版本号和操作码以执行来自客户的请求。

（11）序列号（Sequence Number）：序列号是一个 32 位无符号数，用于标识正在执行的远程过程调用。每一个远程过程激活通过一个唯一序列号所引起的请求调用，该序列号初始化时进行赋值，所有 RPC 无连接的 PDU 报文中代表发送请求的报文具有相同的序列号，无论它们是从客户到服务器还是服务器到客户。当激活一个新的远程过程调用时，将该序列号加 1，这样活动 UUID 和序列码唯一地标识远程过程调用。

（12）操作码 Operation Number：操作码是一个 16 位无符号整数，用来标识调用接口的特殊操作。

（13）接口提示（Interface Hint）：接口提示是一个 16 位无符号整数，目的是让服务器优化其接口的信息查询。

（14）激活提示（Activity Hint）：激活提示是一个 16 位无符号整数。尽管实现时可用于任何用途，但其最初的意图是让服务器优化与活动通信状态的信息查询。

（15）PDU 体长度（PDU Body Length）：指定 PDU 体的长度，最大为 65528 个字节。

（16）帧编码（Fragment Number）：帧编码是一个 16 位无符号整型，用于标识网络中的多个 PDU 传输。例如在请求（request）和应答（response）PDU 多包传输时，帧的号码表示正在发送网络报文，其序号依次递增，第 1 帧为 0，第 2 帧为 1，以依类推。在 fack PDU 和 nocall PDU 报文中，如果有报文体，帧编号则表示用户接收序列；如果帧 0 到帧 n 已经接收，而第 n+1 帧报文尚未接收，则帧的号码应为 n；如果帧编号 0 未收到，则帧的编号应为 16 进制的 FFFF。

（17）鉴定协议标识：认证协议标识是一个 8 位无符号数，用于标识认证的协议。

10.2.2 面向连接的 RPC 协议分析

客户端与服务器通信时，通常采用远程过程调用（RPC）方式，RPC 在运行时通常采用面向连接网络传输模式。根据用户服务器之间协议状态机不同，可以分为两类：调用（CALL）和相关（ASSOCIATION）协议状态机，其协议类型也有所不同。

相关网络协议状态机主要有如下类型的 RPC：

（1）bind。

（2）bind_ack。

（3）bind_nak。

（4）alter_context。

（5）alter_context_response。

调用网络协议状态机主要有如下类型的 RPC：

（1）request。

（2）response。

（3）fault。

（4）shutdown。

（5）cancel。

（6）orphaned。

面向连接的远程过程调用 PDU 的头是可变的，每个头包括公共定义域。bind、bind_ack、alter_context 和 alter_context_response 报文类型中有审核验证部分，而 bind_nak 和 shutdown 网络报文中则没有认证代码，在其他类型报文中则是可选的，这依赖于安全协议。

所有面向连接的 PDU 的报文均有一个公共报文域，即 PDU 头部分，占用 16 个字节，其定义如下：

```
u_int8  rpc_vers = 5;          /* RPC 版本 */
u_int8  rpc_vers_minor;        /* 副版本 */
u_int8  PTYPE;                 /* 报文类型 */
u_int8  pfc_flags;             /* 标志(参见 PFC_... ) */
byte    packed_drep[4];        /* NDR 数据表示格式 */
u_int16 frag_length;           /* 帧的总长度 */
u_int16 auth_length;           /* 认证长度 */
u_int32 call_id;               /* 调用标识 */
```

面向连接的 RPC 主版本号为 5，副版本号可为 0 或者 1，报文类型如表 10-1 所示的定义，例如：如果是 bind 类型，该值为 0X0B。标志字段定义了 PDU 标志位域的不同含义：

```
#define PFC_FIRST_FRAG          0x01/* 第 1 帧 */
#define PFC_LAST_FRAG           0x02/* 最后帧 */
#define PFC_PENDING_CANCEL      0x04/* 发送挂起取消 */
#define PFC_RESERVED_1          0x08
#define PFC_CONC_MPX            0x10/* 支持单独连接的同时多个调用*/
#define PFC_DID_NOT_EXECUTE     0x20/* 仅对错误报文有意义，置位时表示调用未得到执行*/
#define PFC_MAYBE               0x40/* 需要调用语义一致 */
#define PFC_OBJECT_UUID         0x80/* 指定了非空对象的 UUID，存在于可选的对象域中，
否则，删除对象域*/
```

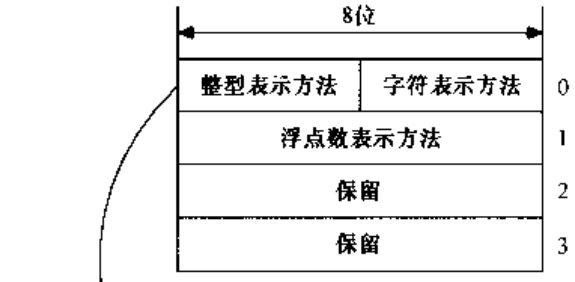
NDR 代表网络数据表示的格式，用于说明采用的字符集、整数的表示方法和浮点数的表示方法等。

字段 frag_length 表示帧的总长度，指全部 PDU 的长度，包括头长度、可选头字段、存根代码和可选身份验证代码的长度。auth_length 仅用于指定身份验证代码的长度，当该值为 0 时，表示没有身份验证代码。字段 call_id 用于服务器/客户方之间请求/应答的匹配，客户向服务器发送一个 call_id 的消息，服务器返回相应的 call_id 的对应消息，该方法还适用于 bind_ack、bind_nak 和 alter_context_response 报文类型。

表示层符号标识符 p_syntax_id_t 定义如下：

```
typedef   struct   {
   uuid_t   if_uuid;
   u_int32  if_version;
} p_syntax_id_t;
```

变量 if_uuid 可设定为接口 UUID，if_version 设定为接口版本，其中主版本号在其低 16 位，副版本在其高 16 位字段中。表示层上下文列表的一个元素（element）定义如下：



数据类型	值	含　义
字符表示方法	0	ASCII
	1	EBCDIC
整型数表示方法	0	低字节在前，高字节在后（Big-endian）
	1	高字节在前，低字节在后（Little-endian）
浮点数表示方法	0	IEEE浮点格式
	1	VAX浮点格式
	2	Cray浮点格式
	3	IBM浮点格式

图 10-7　NDR 数据表示格式

```
typedef   u_int16   p_context_id_t;
typedef   struct {
     p_context_id_t   p_cont_id;
     u_int8           n_transfer_syn;        /* 项数目 */
     u_int8           reserved;              /* 用于对齐，保留 */
     p_syntax_id_t    abstract_syntax;       /* 传输语法列表*/
     p_syntax_id_t [size_is(n_transfer_syn)] transfer_syntaxes[];
} p_cont_elem_t;
```

整个列表由多个元素组成，其数据结构定义如下：

```
typedef   struct {
     u_int8          n_context_elem;         /*项数目*/
     u_int8          reserved;               /*用于对齐，保留*/
     u_short         reserved2;              /*用于对齐，保留*/
     p_cont_elem_t [size_is(n_cont_elem)] p_cont_elem[];
} p_cont_list_t;
```

表示层上下文协商返回的结果定义如下：

```
typedef  short enum {
     acceptance, user_rejection, provider_rejection
} p_cont_def_result_t;
```

上下文元素拒绝原因定义：

```
typedef   short enum {
     reason_not_specified,
     abstract_syntax_not_supported,
```

```
    proposed_transfer_syntaxes_not_supported,
    local_limit_exceeded
} p_provider_reason_t;
```

结果列表的列表元素声明：

```
typedef struct {
   p_cont_def_result_t   result;
   p_provider_reason_t   reason;
   p_syntax_id_t         transfer_syntax;
} p_result_t;
```

整个列表定义顺序及元素数目和 bind 请求相同：

```
typedef struct {
   u_int8   n_results;          /* 数目 */
   u_int8   reserved;           /* 用于对齐，保留*/
   u_int16 reserved2;           /* 用于对齐，保留*/
   p_result_t [size_is(n_results)] p_results[];
} p_result_list_t;

typedef struct{
      /* 恢复到 4 字节对齐方式 */
      u_int8  [size_is(auth_pad_length)] auth_pad[];  /* 对齐 */
      u_int8  auth_type;                              /* 认证服务类型 */
      u_int8  auth_level;                             /* 服务内的级别 */
      u_int8  auth_pad_length;                        /* 无用的字节长度 */
      u_int8  auth_reserved;                          /* 保留，仅用于字节对齐 */
      u_int32 auth_context_id;                        /* 认证上下文标识 */
      u_int8  [size_is(auth_length)] auth_value[]; /* 认证内容*/
} auth_verifier_co_t;
```

字段 auth_pad f 仅用于在存根数据后恢复到 4 字节对齐，否则，它将凶手 0～3 个空字节。auth_type 字段定义使用的审核认证服务，目前仅支持如下类型的值：

（1）rpc_c_authn_none。

（2）rpc_c_authn_dce_secret。

（3）rpc_c_authn_default。

字段 auth_level 定义保护的级别，支持如下类型的值：

（1）rpc_c_protect_level_default。

（2）rpc_c_protect_level_none。

（3）rpc_c_protect_level_connect。

（4）rpc_c_protect_level_call。

（5）rpc_c_protect_level_pkt。

（6）rpc_c_protect_level_pkt_integrity。

（7）rpc_c_protect_level_pkt_privacy。

字段 auth_pad_length 用来说明 pad 字节的长度，附加在头和存根数据的后边，位于认证审

核代码的前面。字段 auth_reserved 保留以供将来用，发送时必须置 0，接收方应忽略。字段 auth_context_id 表示先前已经创建的相应的安全上下文。字段 auth_value 包括与安全相关的数据，对 bind、bind_ack、alter_context 和 alter_context_response PDU，该字段 encodes credentials，其他 PDU，该字段代表校验和，与保护级别有关。

10.2.2.1 bind PDU 报文类型

bind PDU 用于初始化网络表示层的协商，与 OSI 网络分层模型不同的是，该 PDU 包括抽象层、传输层和上下文标识，而抽象层和传输层由 UUID 和其版本标识组成。其 IDL 接口定义如下：

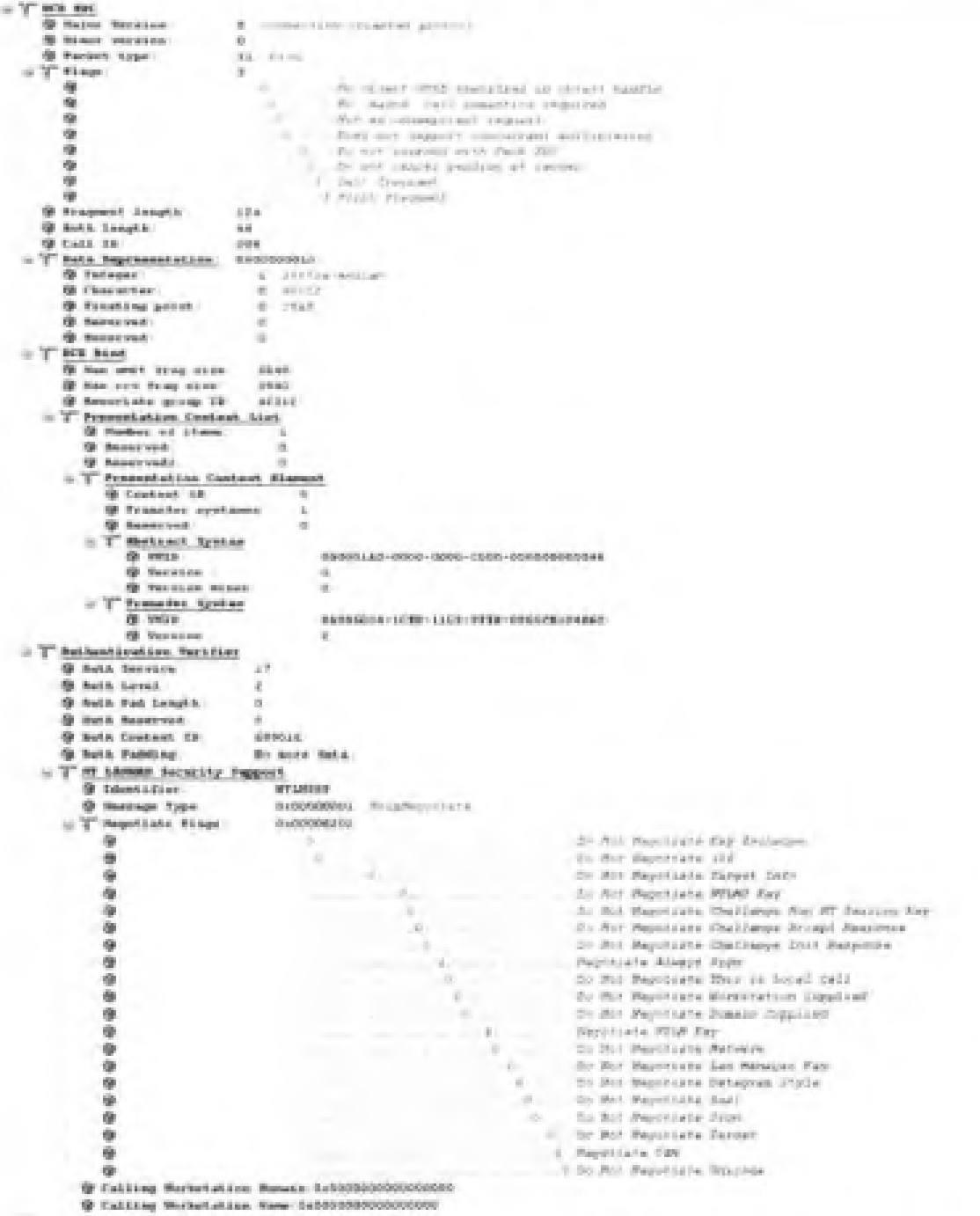
```
typedef struct {
    /* 16 字节公用报文头 */
    u_int8  rpc_vers = 5;           /* RPC 版本号 */
    u_int8  rpc_vers_minor;         /* 副版本号 */
    u_int8  PTYPE;                  /* 报文类型，见表 10-1 */
    u_int8  pfc_flags;              /* 标志字(可参见 PFC_... ) */
    byte    packed_drep[4];         /* NDR 数据表示层格式 */
    u_int16 frag_length;            /* 帧报文总长度 */
    u_int16 auth_length;            /* 身份认证的长度 */
    u_int32 call_id;                /* 调用标识 */

    u_int16 max_xmit_frag;          /* 最大传输帧大小，单位：字节 */
    u_int16 max_recv_frag;          /* 最大接收帧大小，单位：字节 */
    u_int32 assoc_group_id;         /* 客户-服务器相关组表示 */
    /* 表示层上下文列表 */
    p_cont_list_t  p_context_elem;  /* 字节可变 */
    /* 可选的身份验证仅当公共报文头中字段 auth_length 不为 0 时有效 */
    auth_verifier_co_t   auth_verifier;
} rpcconn_bind_hdr_t;
```

字段 max_xmit_frag 用于定义最大传输帧的长度；字段 max_recv_frag 用于定义最大接收帧的长度；字段 assoc_group_id 为 0 时表示一组新的请求，否则表示先前已经建立绑定连接协商的相关组的标识符。该 PDU 的长度不能超过 MustRecvFragSize(1432)，如果 p_context_elem 太长，首先传输主要部分，在得到一个 bind_ack 应答报文后，随后再传输一帧 alter_context PDU 报文。

监听到的 bind 网络报文如图 10-8 所示。根据其 PDU 报文中有无身份验证代码，其报文协议略有不同。无身份验证代码时的 bind PDU 网络协议剖析如图 10-9 所示，有身份验证代码时的 bind PDU 网络协议剖析如图 10-10 所示。

字段 1～8 为 PDU RPC 公用头部分；字段 9～11 为 bind 报文附加的头部分；字段 12～21 为 PDU 体部分，其中 15～21 为一个元素上下文 p_cont_elem_t 所对应的内容，18～19 为抽象层语法的内容，20～21 为传输层语法的内容。



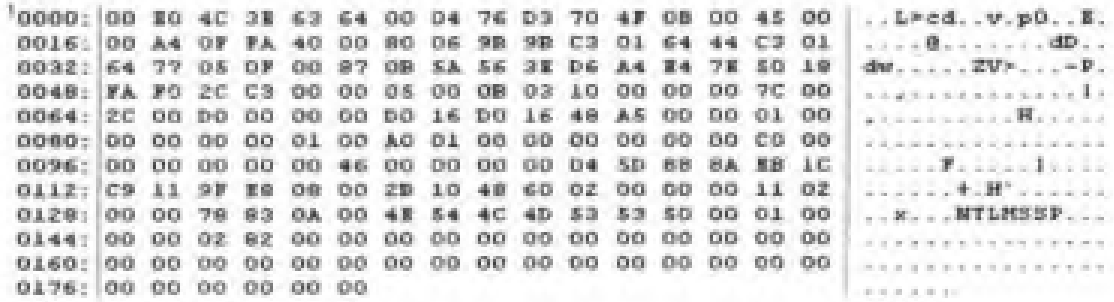
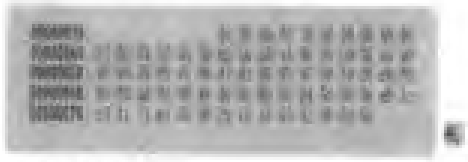


图 10-8　bind 网络报文



左边方框为监听的网络 RPC PDU 报文

报文类型为 bind

解析代码见放大部分

```
00000030:                    1 05 2 00 3 0b 4 03 5 10 00 00 00 6 48 00
00000040: 7 00 00 8 01 00 00 00 9 d0 16 10 d0 16 11 00 00 00 00 12 01 13 00
00000050: 14 00 00 15 00 00 16 01 17 00 18 43 01 00 00 00 00 00 00 c0 00
00000060: 00 00 00 00 00 46 19 00 00 00 00 20 04 5d 88 8a eb 1c
00000070: c9 11 9f e8 08 00 2b 10 48 60 21 02 00 00 00
```

图 10-9　无身份验证代码时的 bind PDU 网络协议剖析

1—RPC 版本号（5）；2—副版本号（0）；3—报文类型（0X0B,bind）；4—标志字（3）；5—NDR 网络数据表示格式；6—帧报文总长度（0x48）；7—身份认证的长度（0，无身份认证数据）；8—调用标识（1）；9—最大传输帧大小（0x16d0）；10—最大接收帧大小（0x16d0）；11—相关组标识（0，表示新的连接）；12—列表中上下文元素数（1）；13—保留（0，1 字节）；14—保留（0，2 字节）；15—元素上下文 ID；16—元素数目；17—保留，用于字节对齐；18—抽象层 UUID（00000143-0000-0000-C000-000000000046）；19—版本号（0.0）；20—传输层 UUID（8A885D04-1CEB-11C9-9FE8-08002B104860）；21—版本号（0.2）

左边方框为监听的网络 RPC PDU 报文

报文类型为 bind，增加了审核认证部分

解析代码见放大部分

```
00000040  28 00 cd 00 00 00 d0 16 d0 16 48 a5 00 00 01 00
00000050  00 00 00 00 01 00 a0 01 00 00 00 00 00 00 c0 00
00000060  00 00 00 00 00 46 00 00 00 00 04 5d 88 8a eb 1c
00000070  c9 11 9f e8 08 00 2b 10 48 60 02 00 00 00 0a 02
00000080  00 00 78 83 0a 00 4e 54 4c 4d 53 53 50 00 01 00
00000090  00 00 07 b2 08 a0 04 00 04 00 24 00 00 00 04 00
000000a0  04 00 20 00 00 00 43 41 49 51 53 4f 46 54
```

图 10-10　有身份验证代码时的 bind PDU 网络协议剖析

1—21—说明同图 10-9，但在字段 7 中身份认证的长度为 40 字节（0x28），表示有验证数据；22—认证服务类型（0x0a）；

23—服务级别（2）；24—无用的字节长度（0）；25—保留，仅用于字节对齐；

26—认证上下文 ID（0x000a8378）；27—40 个字节的身份验证

字段 22～27 是身份验证部分代码。

10.2.2.2 bind_ack PDU 报文类型

bind_ack PDU 用于服务器应答客户方来的 bind 请求，其 IDL 定义如下：

```
typedef struct {
    /* 16 字节公用报文头 略 */
    ...

    u_int16 max_xmit_frag;                  /* 最大传输帧大小 */
    u_int16 max_recv_frag;                  /* 最大接收帧字节 */
    u_int32 assoc_group_id;                 /* 返回的相关组 ID */
    port_any_t sec_addr;                    /* 可选第 2 个地址项*/
    /*保证 4 字节对齐*/
    u_int8 [size_is(align(4))] pad2;
    /* 表示层上下文结果列表 */
    p_result_list_t     p_result_list;      /*字节可变*/
    /* 可选的身份验证仅当公共报文头中字段 auth_length 不为 0 时有效 */
    auth_verifier_co_t   auth_verifier;
} rpcconn_bind_ack_hdr_t;
```

使用 bind 请求时返回第二网络地址 port_any_t，是一个字符串表示本地地址的端口号，字符的长度应将 C 语言的空（NULL）字符结束。其数据结构定义如下：

```
typedef struct {
    u_int16 length;
    char   [size_is(length)] port_spec;     /* port string spec */
} port_any_t;
```

监听到网络 bind_ack 网络 PDU 报文如图 10-11 所示，bind_ack 网络 PDU 报文解析如图 10-12 所示。

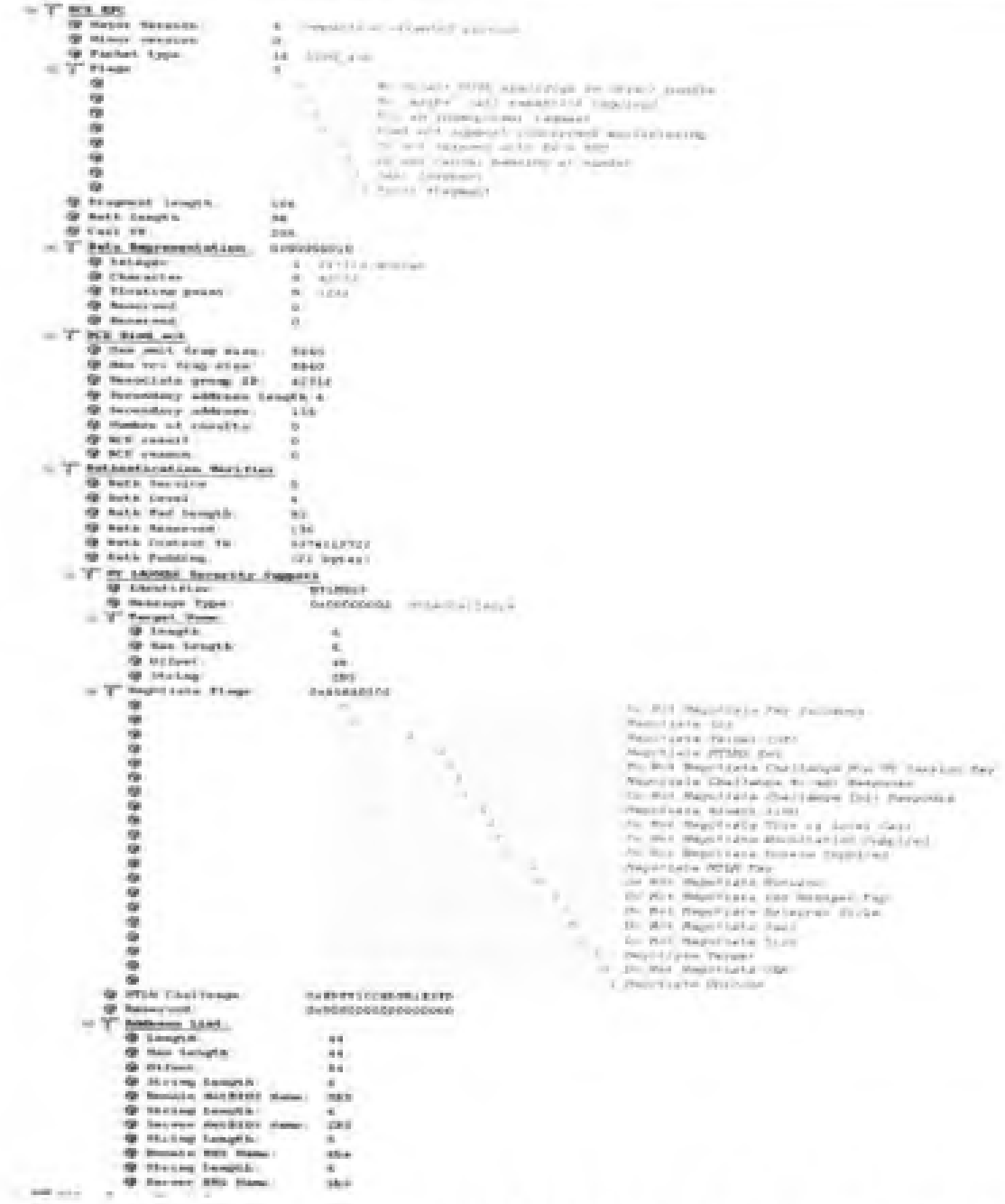
10.2.2.3 bind_ack PDU 报文类型

报文 bind_ack 的 PDU 的 IDL 定义如下：

```
typedef struct {
    /* 16 字节公用报文头 略 */
    ...

    u_int16 max_xmit_frag;      /* 最大传输帧大小 */
    u_int16 max_recv_frag;      /* 最大接收帧字节 */
    u_int32 assoc_group_id;     /* 返回的相关组 ID */
    port_any_t sec_addr;        /* 可选第 2 个地址项*/
    /* 保证 4 字节对齐 */
    u_int8 [size_is(align(4))] pad2;
    /* 表示层上下文结果列表 */
    p_result_list_t     p_result_list;   /*字节可变*/
    /* 可选的身份验证仅当公共报文头中字段 auth_length 不为 0 时有效 */
    auth_verifier_co_t  auth_verifier;
} rpcconn_bind_ack_hdr_t;
```

服务器返回的 bind_ack 报文对应于客户方发送的 bind 报文。



```
0000: 00 04 76 D3 70 4F 00 B0 4C 3E 63 64 08 00 45 00  ..v.pO..L>cd..E.
0016: 00 CE 05 5C 40 00 80 06 A6 0F C3 01 64 77 C3 01  ...\@.......dw..
0032: 64 44 00 87 05 0C D6 A2 A8 23 0B 58 54 70 50 18  dD.......#.XTpP.
0048: 43 F8 14 18 00 00 05 00 0C 03 10 00 00 00 A6 00  C...........
0064: 62 00 CD 00 00 00 D0 16 D0 16 48 A5 00 00 04 00  b.........H.....
0080: 31 33 35 00 00 00 01 00 00 00 00 00 00 00 04 5D  135.............]
0096: 88 8A EB 1C C9 11 9F E8 08 00 2B 10 48 60 02 00  ..........+.H`..
0112: 00 00 0A 02 00 00 78 83 0A 00 4E 54 4C 4D 53 53  ......x...NTLMSS
0128: 50 00 02 00 00 00 06 00 06 00 30 00 00 00 05 82  P.........0.....
0144: 8A A0 ED 9F 1C C8 D9 B1 E3 FD 00 00 00 00 00 00  ................
0160: 00 00 2C 00 2C 00 36 00 00 00 5A 00 42 00 53 00  ..,.,.6...Z.B.S.
0176: 02 00 06 00 5A 00 42 00 53 00 01 00 06 00 5A 00  ....Z.B.S.....Z.
0192: 42 00 53 00 04 00 06 00 7A 00 62 00 73 00 03 00  B.S.....z.b.s...
0208: 06 00 7A 00 62 00 73 00 00 00 00 00 00 00 00 00  ..z.b.s.........
```

图 10-11　监听到网络 bind_ack 网络 PDU 报文


```
00000030: 43 f8 b2 57 00 00 05 00 0c 03 10 00 00 00 a6 00
00000040: 62 00 d9 00 00 00 d0 16 d0 16 4b a5 00 00 04 00
00000050: 31 33 35 00 00 00 01 00 00 00 00 00 00 00 04 5d
00000060: 88 8a eb 1c c9 11 9f e8 08 00 2b 10 48 60 02 00
00000070: 00 00 0a 02 00 00 78 83 0a 00 4e 54 4c 4d 53 53
00000080: 50 00 02 00 00 00 06 00 06 00 30 00 00 00 05 82
00000090: 8a e0 ee 6a 26 ef c8 27 eb f0 00 00 00 00 00 00
000000a0: 00 00 2c 00 2c 00 36 00 00 00 5a 00 42 00 53 00
000000b0: 02 00 06 00 5a 00 42 00 53 00 01 00 06 00 5a 00
000000c0: 42 00 53 00 04 00 06 00 7a 00 62 00 73 00 03 00
000000d0: 06 00 7a 00 62 00 73 00 00 00 00 00
```

图 10-12　bind_ack 网络 PDU 报文解析

1—RPC 版本号（5）；2—副版本号（0）；3—报文类型（0X0c,bind ack）；4—标志字（3）；5—NDR 网络数据表示格式；
6—帧报文总长度（0xa6）；7—身份认证的长度（0x62，有身份认证数据）；8—调用标识（0xd9）；9—最大传输帧大小（0x16d0）；
10—最大接收帧大小(0x16d0)；11—相关组标识（0xa54b，表示已经存在的连接）；12—第二个网络地址的长度；
13—端口字符串，以 NULL 结束；14—保留，用于 4 字节对齐；15—列表中上下文元素数（1）；
16—保留（0，1 字节）；17—保留（0，2 字节）；18—元素上下文 ID；19—元素数目；
20—UUID（8A885D04-1CEB-11C9-9FE8-08002B104860）；21—版本号（0.2）；
22—认证服务类型（0x0a）；23—服务级别（2）；24—无用的字节长度（0）；
25—保留，仅用于字节对齐；26—认证上下文 ID（0x000a8378）；
27—98 个字节的身份验证数据

10.2.2.4　orphaned PDU 报文类型

orphaned PDU 主要用于客户方通知服务器放弃由客户发出的请求，该请求可能未完全传输完或者正在进行传送中。orphaned PDU 的 IDL 声明如下：

```
typedef struct {
    /* 16 字节公用报文头 略 */
    ...
    /* 可选的身份验证仅当公共报文头中字段 auth_length 不为 0 时有效 */
    auth_verifier_co_t   auth_verifier;
} rpcconn_orphaned_hdr_t;
```

10.2.2.5　cancel PDU 报文类型

cancel PDU 用于传输取消操作，其 IDL 声明如下：

```
typedef struct {
    /* 16 字节公用报文头 略 */
    ...
    /* 可选的身份验证仅当公共报文头中字段 auth_length 不为 0 时有效 */
    auth_verifier_co_t   auth_verifier;
} rpcconn_cancel_hdr_t;
```

10.2.2.6　bind_nak PDU 报文类型

bind_nak PDU 用于服务器向客户方发送一个禁止 bind 的相关请求，其 IDL 声明如下：

```
typedef struct {
    /* 16 字节公用报文头 略 */
    ...
    p_reject_reason_t provider_reject_reason;
    p_rt_versions_supported_t versions;
} rpcconn_bind_nak_hdr_t;
```

字段 provider_reject_reason 代表 rejection 原因代码，当该字段为 protocol_version_not_supported，其中字段 versions 指明了服务器所支持的运行时的协议。bind_nak PDU 不包括身份验证信息。p_rt_versions_supported_t 数据结构定义：

```
typedef struct {
    u_int8   n_protocols;   /* count */
    p_rt_version_t  [size_is(n_protocols)] p_protocols[];
} p_rt_versions_supported_t;
```

p_rt_version_t 的数据结构定义如下：

```
typedef struct {
    u_int8   major;
    u_int8   minor;
    } version_t;
typedef   version_t   p_rt_version_t;
```

bind nack 网络 PDU 协议剖析如图 10-13 所示。

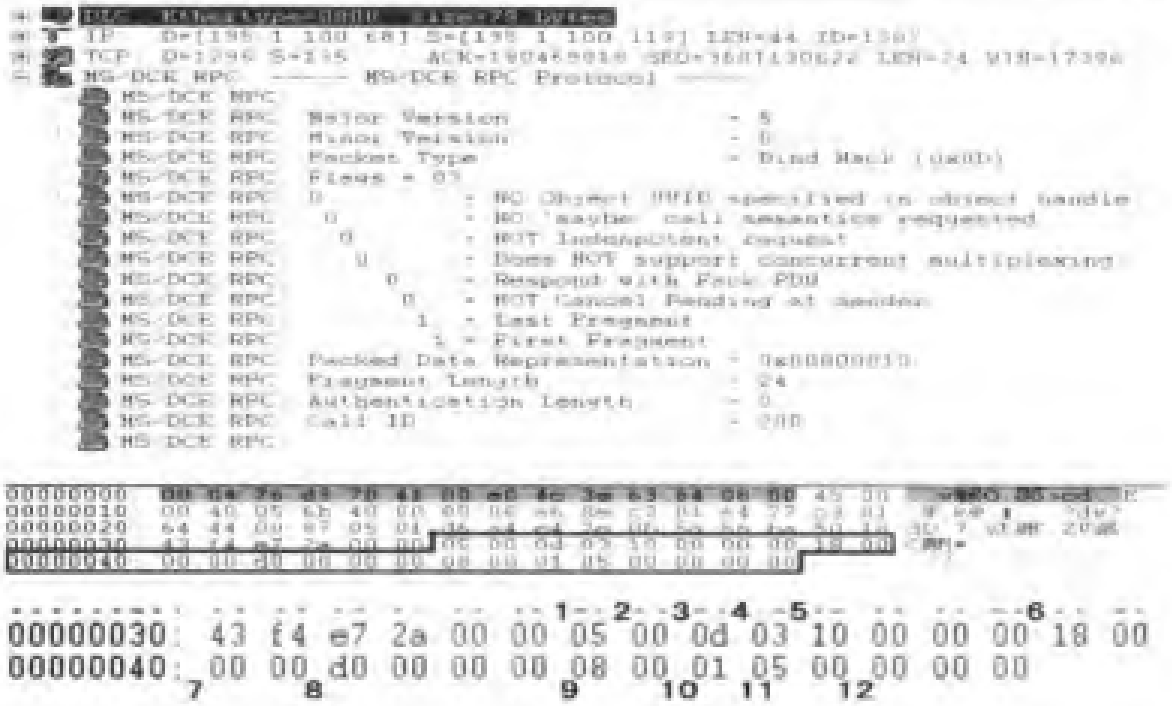


图 10-13　bind nack 网络 PDU 协议剖析

1—RPC 版本号（5）；2—副版本号（0）；3—报文类型（0X0d,bind nack）；
4—标志字（3）；5—NDR 网络数据表示格式；6—帧报文总长度（0x18）；
7—身份认证的长度（0，无身份认证数据）；8—调用标识（0xd0）；
9—绑定请求被拒绝原因代码；10—协议数目；11—版本（5.0）；
12—保留，3 个字节，仅用于 4 字节对齐

10.2.2.7　alter_context　PDU 报文类型

alter_context PDU 用于请求更多的网络表示层的协商，也可用于协商安全上下文，其 IDL 接口定义如下：

```
typedef struct {
    /* 16 字节公用报文头 略 */
    ...
    u_int16 max_xmit_frag;          /* 忽略 */
    u_int16 max_recv_frag;          /* 忽略 */
    u_int32 assoc_group_id;         /* 忽略 */
    /* 数据表示层上下文列表 */
    p_cont_list_t  p_context_elem;  /* 字节可变 */
    /* 可选的身份验证仅当公共报文头中字段 auth_length 不为 0 时有效 */
    auth_verifier_co_t   auth_verifier;
} rpcconn_alter_context_hdr_t;
```

可以看出，alter_context PDU 数据结构定义基本与 bind PDU 定义相同，不同的是其报文类型 PTYPE 不同，同时字段 max_xmit_frag、max_recv_frag 和 assoc_group_id 保留未用，alter_context PDU 网络报文解析如图 10-14、图 10-15 所示。

图 10-14　alter_context PDU 网络报文解析

1—RPC 版本号（5）；2—副版本号（0）；3—报文类型（0X0E）；4—标志字；5—NDR 数据表示层格式；6—帧报文总长度；7—身份认证字段长度；8—调用标识；9—最大发送帧长度（保留）；10—最大接收帧长度（保留）；11—相关分组标志；12—列表中上下文元素数（1）；13—保留（0，1 字节）；14—保留（0，2 字节）；15—元素上下文 ID；16—元素数目；17—保留，用于字节对齐；18—抽象层 UUID（1CA5C413-8A8C-40CE-BBE5-5DDA3ED2D446）；19—版本号（0.0）；20—传输层 UUID（8A885D04-1CEB-11C9-9FE8-08002B104860）；21—版本号（0.2）

10.2.2.8　alter_context_resp PDU 报文类型

alter_context_resp PDU 表示服务器对 alter_context PDU 请求的应答，其 IDL 接口定义如下：

```
typedef struct {
    /* 16 字节公用报文头 略 */
    ...
    u_int16 max_xmit_frag;        /* 忽略 */
    u_int16 max_recv_frag;        /* 忽略 */
    u_int32 assoc_group_id;       /* 忽略 */
    port_any_t sec_addr;          /* 忽略 */
    /* 以下字段保证 4 字节对齐 */
    u_int8 [size_is(align(4))] pad2;
    /* 表示层上下文结果列表，包括提示信息 */
    p_result_list_t     p_result_list;   /* 字节可变 */
    /* 可选的身份验证仅当公共报文头中字段 auth_length 不为 0 时有效 */
    auth_verifier_co_t   auth_verifier;
} rpcconn_alter_context_response_hdr_t;
```

```
Info:                 P=0x00 S=0x00 L=130 T=16:13:36.695995000 02/21/2004
Ethernet:             B=Realtek Semi:3E:67:64 S=3 Com:D2:70:4F
IP:                   S=195.1.100.68 B=195.1.100.119
TCP:                  S=epmap(135) D=nsa SEQ=190676967 ACK=3601232267 W=64093
DCE RPC
  Major Version:      5  connection-oriented protocol
  Minor version:      0
  Packet type:        14  alter_context
  Flags:              3
                      1... .... No object UUID specified in object handle
                      .0.. .... No maybe call semantics required
                      ..0. .... Not an independent request
                      ...0 .... Does not support concurrent multiplexing
                      .... 0... Do not respond with Fack PDU
                      .... .0.. Do not cancel pending at sender
                      .... ..1. Last fragment
                      .... ...1 First fragment
  Fragment length:    72
  Auth length:        0
  Call ID:            2
  Data Representation:  0x00000010
    Integer:            1  little-endian
    Character:          0  ASCII
    Floating point:     0  IEEE
    Reserved:           0
    Reserved:           0
  DCE Alter_context
    Max transmit frag size:5840
    Max receive frag size:5840
    Associate group ID:   4740005
    Presentation Context List
      Number of items:    1
      Reserved:           0
      Reserved2:          0
      Presentation Context Element
        Context ID            1
        Transfer syntaxes:    1
        Reserved:             0
        Abstract Syntax
          UUID                1CA5C413-8A8C-40CE-BB85-5DDA3BD2D446
          Version :           0
          Version Minor:      0
        Transfer Syntax
          UUID                8A885D04-1CEB-11C9-9FE8-08002B104860
          Version:            2
FCS - Frame Check Sequence
  FCS:                0xAA2C742D  Calculated
```

```
0000: 00 B0 4C 3E 63 64 00 04 76 D3 70 4F 08 00 45 00 00 70 10 07  ..L>cd..v.pO..E..p..
0020: 40 00 80 06 9B C2 C3 01 64 44 C3 01 64 77 05 11 04 1A 0B 5D  @.......dD..dw.....]
0040: 7F B7 D6 A7 F8 2B 50 18 FA 58 66 B0 00 00 05 00 0B 03 10 00  .....+P..Xf.........
0060: 00 00 48 00 00 00 02 00 00 00 D0 16 D0 16 A5 53 4B 00 01 00  ..H..........SK...
0080: 00 00 01 00 01 00 13 C4 A5 1C 8C 8A CE 40 BB B5 5D DA 3B D2  ...........@..].;.
0100: D4 46 00 00 00 00 04 5D 88 8A EB 1C C9 11 9F E8 08 00 2B 10  .F.....]..........+.
0120: 48 60 02 00 00 00 00 00 00 00                                H`........
```

图 10-15　alter_context PDU 网络报文

可以看出，alter_context_resp PDU 的格式同 bind_ack PDU，不同的是其报文类型 PTYPE 不同，字段 max_xmit_frag、max_recv_frag、assoc_group_id 和 sec_addr 保留未用。

图 10-16　alter_context_resp PDU 网络报文解析

1—RPC 版本号（5）；2—副版本号（0）；3—报文类型（0X0F）；4—标志字；5—NDR 数据表示层格式；6—帧报文总长度；7—身份认证字段长度；8—调用标识；9—最大发送帧长度（保留）；10—最大接收帧长度（保留）；11—相关分组标志；12—第二个网络地址的长度；13—保留（用于 4 字节对齐）；14—结果列表元素数（1）；15—保留（0，1 字节）；16—保留（0，2 字节）；17—结果代码（acceptance）；18—拒绝原因代码，此处无效；19—传输层 UUID（8A885D04-1CEB-11C9-9FE8-08002B104860）；20—版本号（0.2）

10.2.2.9　shutdown PDU 报文类型

shutdown 网络报文由服务器发向客户端要求客户终止连接，并且释放资源，该类 PDU 报文不含审核认证部分，因此 shutdown 网络报文 PDU 仅包括文件头域说明。其 IDL 接口定义如下：

```
typedef struct {
   /* 16 字节公用报文头 略 */
   ...
} rpcconn_shutdown_hdr_t;
```

10.2.2.10　request PDU 报文类型

请求 PDU 用于初始化一次调用请求，request PDU 的 IDL 定义如下：

```
typedef struct {
  /* 16 字节公用报文头 略 */
  ...
  u_int32  alloc_hint;              /* 内存分配提示 */
  p_context_id_t p_cont_id;         /* 表示层上下文，数据表示层 */
  u_int16 opnum;                    /* 接口内部操作符 */
  /* 仅有请求（request）时可选域，如果字段 PFC_OBJECT_UUID 非 0 时有用*/
```

```
    uuid_t  object;                    /* 对象 UUID */
    /* 存根数据，8 字节对齐 */
...
    /* 可选的身份验证仅当公共报文头中字段 auth_length 不为 0 时有效 */
    auth_verifier_co_t   auth_verifier;
} rpcconn_request_hdr_t;
```

```
⊞ Info:          F=0x00 S=0x00 L=114 T=16:13:36.696197000 02/21/2004
⊞ Ethernet:      D=3 Com:D3:70:4F S=Realtek Semi:9E:63:64
⊞ IP:            S=195.1.100.119 D=195.1.100.68
⊞ TCP:           S=cmc D=sqlproxy SEQ=3601332267 ACK=190677039 W=17200
⊟ DCE RPC
    Major Version:          5  connection-oriented protocol
    Minor version:          0
    Packet type:            15  alter_context_resp
  ⊟ Flags:                  3
                              0... .... No object UUID specified in object handle
                              .0.. .... No 'maybe' call semantics required
                              ..0. .... Not an indempotent request
                              ...0 .... Does not support concurrent multiplexing
                              .... 0... Do not respond with Fack PDU
                              .... .0.. Do not cancel pending at sender
                              .... ..1. Last fragment
                              .... ...1 First fragment
    Fragment length:        56
    Auth length:            0
    Call ID:                2
  ⊟ Data Representation:    0x00000010
      Integer:                1  little-endian
      Character:              0  ASCII
      Floating point:         0  IEEE
      Reserved:               0
      Reserved:               0
  ⊟ DCE Alter_context_resp
      Max xmit frag size:     5840
      Max rcv frag size:      5840
      Associate group ID:     4740005
      Secondary address length:0
      Number of results:      1
      ACK result:             0
      ACK reason:             0
    ⊟ Transfer Syntax
        UUID                    8A885D04-1CEB-11C9-9FE8-08002B104860
        Version:                2
⊟ FCS - Frame Check Sequence
    FCS:                    0x88E9DC66  Calculated
```

```
0000: 00 04 76 D3 70 4F 00 E0 4C 3E 63 64 08 00 45 00 00 60 05 76  ..v.pO..L>cd..E..`.v
0020: 40 00 80 06 A6 63 C3 01 64 77 C3 01 64 44 04 1A 05 11 D6 A7  @....c..dw..dD......
0040: F8 2E 0B 5D 80 2F 50 18 43 30 3D F6 00 00 05 00 0F 03 10 00  .+.]./P.C0=.........
0060: 00 00 38 00 00 00 02 00 00 00 D0 16 D0 16 A5 53 48 00 00 00  ..8............SH...
0080: 31 30 01 00 00 00 00 00 00 00 04 5D 88 8A EB 1C C9 11 9F E8  10.........].......
0100: 08 00 2B 10 48 60 02 00 00 00 00 00 00 00                    ..+.H`........
```

图 10-17　alter_context_resp PDU 网络报文

字段 p_cont_id 代表了表示层上下文标识符用于标识数据表示层，opnum 字段表示接口内部相应的操作，API 操作共有 7 类：

（1）0——绑定（binding）相关操作。

（2）1——命名服务操作（name service）。

（3）2——终端（endpoint）操作。

（4）3——安全（security）操作。

（5）4——存根内存管理操作。

（6）5——管理操作。

（7）6——全局唯一标识符（UUID）操作。

在公共报文头中字段 pfc_flags 定义 PFC_OBJECT_UUID 属性时，该 PDU 可包括对象 UUID，object 字段表示对象 UUID。alloc_hint 字段为可选项，表示客户方以提示方式通知服务器应分配连接的帧接收缓冲区的大小，正确接收发送的报文，为 0 表示无任何提示信息。rpcconn_request_hdr_t 数据结构最小为 24 字节，此时将不包括对象 UUID、身份验证字段或者使用服务完整性和安全性验证。存根数据长度按以下公式计算：帧报文总长度减去固定的请求报文头长度（24 字节，16 字节 PDU 头加随后的 8 个字节）和身份验证字段的长度，如果包括对象的 UUID，还应减去 UUID（16 字节）的长度，即：

```
stub_data_length = frag_length - fixed_header_length - auth_length;
if pfc_flags & PFC_OBJECT_UUID
{
    stub_data_length = stub_data_length - sizeof(uuid_t);
}
```

10.2.2.11　response PDU 报文类型

报文 response 的 IDL 定义如下：

```
typedef struct {
  /* 16 字节公用报文头 略 */
  ...
  u_int32  alloc_hint;            /* 内存分配提示 */
  p_context_id_t p_cont_id        /* 表示层上下文，数据表示层 */
  u_int8  cancel_count            /* 取消计数 */
  u_int8  reserved;               /* 保留 */
  /* 存根数据，8 字节对齐  */
  ...
  /* 可选的身份验证仅当公共报文头中字段 auth_length 不为 0 时有效 */
  auth_verifier_co_t   auth_verifier;
} rpcconn_response_hdr_t;
```

报文 response PDU 用于应答一个主动调用。字段 p_cont_id 包括上下文标识用于表示数据表示层；字段 cancel_count 指示了所接收取消的次数；字段 alloc_hint 是可选的，发送者可向接收者提供发送帧请求的连接分配的缓冲区的大小，为 0 时表示发送方不提供任何信息。


```
00000030: fa 20 c7 fa 00 00 05 00 00 00 83 10 00 00 00 58 00
00000040: 00 00 02 00 00 00 30 00 00 00 01 00 07 00 03 00
00000050: 00 00 d8 07 00 00 d4 07 00 00 03 00 00 00 05 00
00000060: 06 00 00 00 00 00 00 00 00 00 2b 37 96 14 e6 d7
00000070: 19 44 bb 90 6a c9 8e bb 7b be 00 00 00 00 06 00
00000080: 00 00 07 00 00 00 d4 07 00 00 03 00 00 00
```

图 10-18　request 报文解析

1—RPC 版本号（5）；2—副版本号（0）；3—报文类型（0X00）；4—标志字；5—NDR 数据表示层格式；
6—帧报文总长度（0X58）；7—身份认证字段长度；8—调用标识；
9—内存分配提示符；10—数据表示层 ID；11—操作（7）；
12—64 字节的存根代码（以 8 字节对齐）

图 10-19　response 报文解析

1—RPC 版本号（5）；2—副版本号（0）；3—报文类型（0X02）；4—标志字；5—NDR 数据表示层格式；
6—帧报文总长度（0X58）；7—身份认证字段长度；8—调用标识；9—内存分配提示符；
10—数据表示层 ID；11—取消数目；12—保留；
13—16 字节的存根代码（以 8 字节对齐）

10.2.2.12　fault PDU 报文类型

fault PDU 用于向客户发送一个异常信息，这个异常信息可能是 RPC 运行错误、RPC 存根错误或者特定 RPC 异常。其 IDL 定义如下：

```
typedef struct {
  /* 16 字节公用报文头 略 */
  ...
  u_int32  alloc_hint;          /* 内存分配提示 */
  p_context_id_t p_cont_id      /* 表示层上下文，数据表示层 */
  u_int8  cancel_count          /* 取消计数 */
  u_int8  reserved;             /* 保留 */
  /* 错误代码 */
  u_int32  status               /* 运行时的错误代码 */
  /* 补钉，用于以下 8 字节对齐 */
  u_int8  reserved32[4];        /* 保留，用于字节对齐 */
  /* 存根数据，8 字节对齐 */
```

```
    ...
    /* 可选的身份验证仅当公共报文头中字段 auth_length 不为 0 时有效 */
    auth_verifier_co_t   auth_verifier;
} rpcconn_fault_hdr_t;
```

字段 p_cont_id 包括上下文标识；字段 alloc_hint 使用时可选，提示客户方帧请求连续分配的缓冲区大小；字段 status 用于说明运行时的状态，不为 0 时表示是一个运行是的错误代码。例如，接口版本不匹配，为 0 时则表示存根代码中所定义的异常，某些代码还表明调用未得到执行，这需要将某些位置位如可将 PFC_DID_NOT_EXECUTE 标志置为 TRUE 等。

```
Info:          F=0x00 S=0x00 L=90 T=16:19:96.258559000 02/21/2004
Ethernet:      D=3 Com:D3:70:4F S=Realtek Semi:98:68:64
IP:            S=195.1.100.119 D=195.1.100.68
TCP:           S=epmap D=2000 SEQ=3600984265 ACK=150335142 W=16418
DCE RPC
  Major Version:          5   connection-oriented protocol
  Minor version:          0
  Packet type:            3   fault
  Flags:                  3
                          0... .... No object UUID specified in object handle
                          .0.. .... No 'maybe' call semantics required
                          ..0. .... Not an idempotent request
                          ...0 .... Does not support concurrent multiplexing
                          .... 0... Do not respond with Fack PDU
                          .... .0.. Do not cancel pending at sender
                          .... ..1. Last Fragment
                          .... ...1 First Fragment
  Fragment length:        32
  Auth length:            0
  Call ID:                205
  Data Representation:    0x00000010
    Integer:              1   little-endian
    Character:            0   ASCII
    Floating point:       0   IEEE
    Reserved:             0
    Reserved:             0
  DCE Stub
ECS - Frame Check Sequence
  FCS:                    0xA50887BD   Calculated
```

```
0000: 00 04 76 D3 70 4F 00 E0 4C 3E 63 64 08 00 45 00 00 49 05 5E  ..v.pO..L>cd..E..I.
0020: 40 00 80 06 A6 93 C3 01 64 77 C3 01 64 44 00 87 05 DC D6 A2  @.......dw..dD......
0040: A6 C9 0B 58 59 46 50 18 40 22 2E 2A 00 00 05 00 03 03 10 00  ...XYFP.@".*........
0060: 00 00 20 00 00 00 CD 00 00 00 00 00 00 00 00 00 00 00 05 00  .. ...............
0080: 00 00 00 00 00 00 00 00 00 00                               ..........
```

图 10-20　监听到 Fault PDU 网络报文

```
MS/DCE RPC  ------ MS/DCE RPC Protocol ------
    MS/DCE RPC
    MS/DCE RPC  Major Version                   = 5
    MS/DCE RPC  Minor Version                   = 0
    MS/DCE RPC  Packet Type                     = Fault (0x03)
    MS/DCE RPC  Flags = 03
    MS/DCE RPC  0.          = NO Object UUID specified in object handle
    MS/DCE RPC  .0          = NO 'maybe' call semantics requested
    MS/DCE RPC  ..0         = NOT Indempotent request
    MS/DCE RPC  ...0        = Does NOT support concurrent multiplexing
    MS/DCE RPC  ....0       = Respond with Pack PDU
    MS/DCE RPC  .....0      = NOT Cancel Pending at sender
    MS/DCE RPC  ......1     = Last Fragment
    MS/DCE RPC  .......1    = First Fragment
    MS/DCE RPC  Packed Data Representation = 0x00000010
    MS/DCE RPC  Fragment Length             = 32
    MS/DCE RPC  Authentication Length       = 0
    MS/DCE RPC  Call ID                     = 207
    MS/DCE RPC
```

```
00000000: 00 04 76 d0 70 81 00 e0 4e 2e 63 64 08 00 45 00   ..v.p...N.cd..E.
00000010: 00 48 05 67 40 00 80 06 ab 8a c1 01 64 77 c3 01   .H.g@.......dw..
00000020: 64 44 00 87 05 0e d6 a4 30 21 0b 53 fd 18 50 18   dD......0!.S..P.
00000030: 40 22 3f ed 00 00 05 00 03 03 10 00 00 00 20 00   @"?.............
00000040: 00 00 cf 00 00 00 00 00 00 00 00 00 00 00 05 00   ................
00000050: 00 00 00 00 00 00                                 ......
```

左边方框为监听的网络 RPC PDU 报文

报文类型为 fault

解析代码见放大部分

```
00000030: 40 22 3f ed 00 00 05 00 03 03 10 00 00 00 20 00
00000040: 00 00 cf 00 00 00 00 00 00 00 00 00 00 00 05 00
00000050: 00 00 00 00 00 00
```

图 10-21　Fault PDU 网络报文解析

1—RPC 版本号（5）；2—副版本号（0）；3—报文类型（0X02）；4—标志字；5—NDR 数据表示层格式；
6—帧报文总长度（0X20）；7—身份认证字段长度；8—调用标识；9—内存分配提示符；
10—数据表示层 ID；11—取消数目；12—保留；13—运行时的错误代码（5）；
14—保留，用于 8 字节对齐

10.3　VxDCOM 网络协议分析实例

本章结合一个具体实例 RPCDemo 来分析 VxWorks 操作系统下的 DCOM 网络协议，首先接口 RPCDemo.idl 的定义如下：

```
#ifdef _WIN32
import "unknwn.idl";
#else
import "vxidl.idl";
#endif

[
    object,
    oleautomation,
    uuid(B5d1ce80-9a37-11db-00d4-000000000000),
    pointer_default(unique)
```

```
]
interface IRPCDemo : IUnknown
    {

    typedef struct tagARRAY_STRUCT
    {
        int nLen;
        [size_is(nLen)]long *pdbArray;
    }ARRAY_STRUCT;

    HRESULT Sum ([in]LONG x, [in]LONG y, [out]LONG * ret);
    HRESULT SumArray ([in]ARRAY_STRUCT arr,[out]LONG * ret);

    };


[
    uuid(b1f93660-9a37-11db-005a-000000000000),
    version(1.0),
    helpstring("RPCDemo Type Library")
]
library RPCDemoLib
    {
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    [
    uuid(b1f6c560-9a37-11db-005a-000000000000),
    helpstring("RPCDemo Class")
    ]
    coclass RPCDemo
    {
        [default] interface IRPCDemo;
    };
};
```

服务器端的代码比较简单，这里不再重复叙述，仅列出客户方的主要程序，其代码如下：

```
#include "dcomLib.h"
#include "../RPCDemo.h"
#define mbstowcs comAsciiToWide

#include <stdio.h>

int dcomdemoClient (const char* serverName)
{
    OLECHAR      wszServerName [128];
    HRESULT      hr = S_OK;
    //声明接口指针
    IRPCDemo * pDemo = NULL;

    mbstowcs (wszServerName, serverName, strlen (serverName) + 1);
```

```
//初始化 COM 库
hr = CoInitializeEx (0, COINIT_MULTITHREADED);
if (FAILED (hr))
    return hr;
//创建类实例
MULTI_QI mqi [] = {
    {&IID_IUnknown, 0, S_OK},
};

COAUTHINFO authInfo = {
    RPC_C_AUTHN_WINNT,
        RPC_C_AUTHZ_NONE, 0,
        RPC_C_AUTHN_LEVEL_NONE,
        RPC_C_IMP_LEVEL_IMPERSONATE,
        0,
        EOAC_NONE
};

COSERVERINFO serverInfo = { 0, wszServerName, &authInfo, 0 };

hr = CoCreateInstanceEx ( CLSID_RPCDemo,
    0,
    CLSCTX_REMOTE_SERVER,
    &serverInfo,
    1,
    mqi);

if (SUCCEEDED (hr))
{
    if (SUCCEEDED (mqi [0].hr))
    {
        printf ("Created IID_IUnknown OK\n");

        //得到用户接口
        mqi [0].pItf->QueryInterface(IID_IRPCDemo,(void **)&pDemo);
        if (SUCCEEDED (hr))
        {
            printf ("Created remote IID_IRPCDemo OK\n");
        }

        //释放接口
        mqi [0].pItf->Release ();

        //调用用户接口
        long x=1,y=3,ret;

        hr = pDemo->Sum(x,y,&ret);
        printf("%d + %d = %d\n",x,y,ret);

        long xx[]={1,2,3,4,5,6,7,8,9,10};
        ARRAY_STRUCT array;
        long retarr;
```

```
            array.nLen = sizeof(xx)/sizeof(long);
            array.pdbArray=xx;
            hr = pDemo->SumArray(array,&retarr);
            printf("array xx sum is %d\n",retarr);

            //释放用户接口
            pDemo->Release();
        }
        else
        {
            printf ("Failed:HRESULT=0x%lX\n", mqi [0].hr);
        }
    }
    else
    {
        printf ("Failed:HRESULT=0x%lX\n", hr);
    }

    CoUninitialize ();

    return hr;
}
```

通过 sniffer 监听到的网络报文如图 10-22 所示。

5		[194.0 0.100]	[194.0.0 101]	TCP D=65000 S=1032 SYN SEQ=2399329	74
6		[194.0 0.101]	[194 0 0 100]	TCP D=1032 S=65000 SYN ACK=2399329	74
7		[194 0 0 100]	[194 0 0 101]	TCP D=65000 S=1032 ACK=2412754	66
8		[194.0.0 100]	[194 0 0 101]	MS/DCE RPC(V5.0) Bind	138
9		[194 0 0.101]	[194 0 0 100]	TCP D=1032 S=65000 ACK=2399329	66
10		[194 0.0 101]	[194 0 0 100]	MS/DCE RPC(V5 0) Bind Ack	126
11		[194 0.0 100]	[194 0 0 101]	TCP D=65000 S=1032 ACK=2412754	66
12		[194.0 0 100]	[194 0 0.101]	MS/DCE RPC(V5 0) Request	182
13		[194.0 0 101]	[194 0.0 100]	TCP D=1032 S=65000 ACK=2399329	66
14		[194 0 0 101]	[194.0 0 100]	MS/DCE RPC(V5 0) Response	158
15		[194 0 0 100]	[194 0 0 101]	TCP D=65000 S=1032 ACK=2412754	66
16		[194.0 0.100]	[194.0 0 101]	MS/DCE RPC(V5 0) Alter Context	138
17		[194 0 0 101]	[194 0 0 100]	TCP D=1032 S=65000 ACK=2399329	66
18		[194.0.0.101]	[194 0 0.100]	MS/DCE RPC(V5 0) Alter Contest Res	122
19		[194 0 0 100]	[194 0 0 101]	TCP D=65000 S=1032 ACK=2412754	66
20		[194.0 0.100]	[194.0 0 101]	MS/DCE RPC(V5 0) Request	146
21		[194 0 0.101]	[194 0.0.100]	TCP D=1032 S=65000 ACK=2399329	66
22		[194 0 0 101]	[194.0.0 100]	MS/DCE RPC(V5 0) Response	106
23		[194.0 0 100]	[194 0 0 101]	TCP D=65000 S=1032 ACK=2412754	66
24		[194.0.0.100]	[194 0 0 101]	MS/DCE RPC(V5 0) Request	190
25		[194 0 0 101]	[194.0.0.100]	TCP D=1032 S=65000 ACK=2399329	66
26		[194.0 0 101]	[194.0.0 100]	MS/DCE RPC(V5 0) Response	106
27		[194 0.0 100]	[194 0 0 101]	TCP D=65000 S=1032 ACK=2412754	66
28		[194 0 0 100]	[194 0 0.101]	MS/DCE RPC(V5 0) Alter Context	138
29		[194 0 0 101]	[194 0.0 100]	TCP D=1032 S=65000 ACK=2399330	66
30		[194 0 0 101]	[194.0 0 100]	MS/DCE RPC(V5 0) Alter Contest Res	122
31		[194.0 0.100]	[194 0 0 101]	TCP D=65000 S=1032 ACK=2412754	66
32		[194 0.0 100]	[194 0.0 101]	MS/DCE RPC(V5 0) Request	194
33		[194.0 0 101]	[194 0.0 100]	TCP D=1032 S=65000 ACK=2399330	66
34		[194.0 0.101]	[194.0 0.100]	MS/DCE RPC(V5.0) Response	102
35		[194 0 0.100]	[194 0 0 101]	TCP D=65000 S=1032 ACK=2412754	66
36		[194 0 0 100]	[194.0.0 101]	TCP D=65000 S=1032 FIN ACK=2412754	66
37		[194 0 0 101]	[194 0 0 100]	TCP D=1032 S=65000 ACK=2399330	66
38		[194.0 0 101]	[194 0 0 100]	TCP D=1032 S=65000 FIN ACK=2399330	66
39		[194.0 0 100]	[194 0 0.101]	TCP D=65000 S=1032 ACK=2412754	66

图 10-22 监听的客户方—服务器之间的网络报文

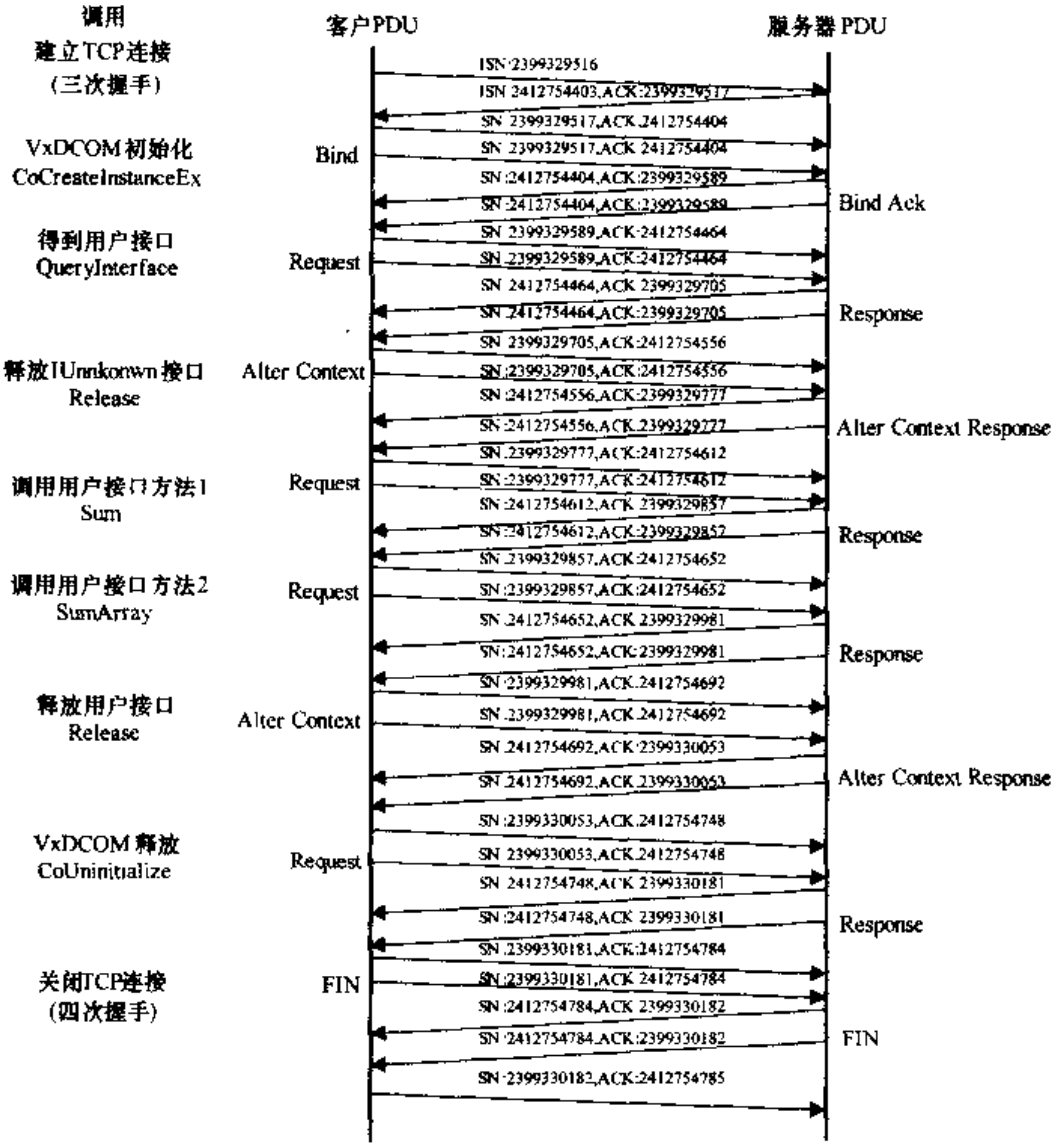


图 10-23　客户方—服务器之间 DCOM 访问时的网络报文序列

从图 10-23 可以看出，客户在调用 VxDCOM 库初始化 CoCreateInstanceEx 函数时，首先建立 TCP 连接，客户向服务器发送 Bind 类型的 RPC 报文，由于 Bind 报文是面向连接的 TCP 报文，服务器需要向客户回送确认 ACK 报文，对应 Bind 报文，服务器需要向客户方发送 Bind Ack 报文，客户方在收到服务器发送的 Bind Ack 报文，需要向服务器发送确认 ACK 报文，至此，完成了 VxDCOM 的初始化过程。访问远程客户程序时，需要建立面向连接（TCP）的通信过程，通常情况下要经过三个过程，又称为三次握手过程。

（1）客户方向服务器发送一个 SYN 报文指明客户方欲连接的端口号，该报文中有一个初始序列号 ISN（2399329516）。

（2）服务器回送包含服务器初始序列号 ISN（2412754403）的 SYN 报文进行应答，同时需将应答报文中确认报文序号设置为客户的 ISN 加 1。

（3）客户必须将确认序号设置为服务器的 ISN 加 1 进行回送。

上述三个过程完成后即表明成功建立了客户—服务器之间的 TCP 连接。

客户在访问指定接口时需要调用 QueryInterface 函数，此时首先由客户向服务器发送

Request 类型的 RPC 报文，服务器在收到客户报文后，由于该报文是面向连接的，服务器需要向客户发送确认 ACK 报文，对应 Request 报文，服务器需要回送 Response 报文，客户方在收到该报文后，需要向服务器回送确认 ACK 报文。

客户在释放接口时，需要调用 Release 函数，此时首先由客户向服务器发送 Alter Context 类型的 RPC 报文，服务器在收到客户报文，由于该报文是面向连接的，服务器需要向客户发送确认 ACK 报文，对应 Alter Context 报文，服务器则需要回送 Alter Context Response 报文，客户方在收到该报文，需要向服务器回送确认 ACK 报文。

在客户方释放 VxDCOM 时，首先向服务器发送 Request 类型的 RPC 报文，服务器回送确认 ACK 报文，服务器再回送 Response 类型的 RPC 报文，客户在收到服务器回送的 Response 报文，回送确认 ACK 报文，此后双方终止 TCP 连接。终止一个 TCP 连接需要经过四个过程，又称为四次握手，其基本流程如下：

（1）客户向服务器发送一个 FIN 报文。

（2）服务器向客户回送确认 ACK 报文。

（3）服务器向客户发送 FIN 报文。

（4）客户向服务器发送确认 ACK 报文。

在调用服务器所提供接口的方法时，客户需要向服务器发送 Request 报文，服务器在收到 Request 报文时，需要回送一帧确认 ACK 报文，对应客户的 Request 报文，服务器需要回送 Response 报文，客户在收到服务器的 Response 报文时，需要向服务器回送确认 ACK 报文，到此，完成了一次接口的调用。

这里，具体分析一下接口调用的网络协议，其他协议在前面已经概述。我们分析一下用户在调用方法 Sum 时的 Request 报文和 Response 的情况以及调用方法 SumArray 时 Request 报文和 Response 的情况。

在调用 Sum 方法时，由于不需要进行内在分配，其操作码定义为 3 则表示安全操作，服务方可以安全使用存根数据。要求服务器返回的调用标识应为 79（0x0000004f），如图 10-24 所示。存根代码类型为 RPC_PROXY_MSG，后部分为方法 Sum 的两个输入参数 1 和 3，如图 10-25 所示。

在图 10-25 中需要说明的是在存根数据中，包含了对象 UUID 的 16 个字节，存根数据的后部分应为用户调用接口方法的输入“in”或者输入/输出参数“inout”，这里是客户要求服务器计算整数 1 和整数 3 相加。

图 10-26 则是服务器返回的客户 request RPC PDU 的对应 Response RPC PDU 报文，其对应的调用标识（call ID）应回填客户的调用标识，即 49（0X0000004F），以说明两者之间的对应关系。图 10-27 解析了应答报文中的数据项及其内容，字段 3 说明了接口 Sum 的输出值“out”为 4，即 1+3 的值，返回值为 HRESULT 类型，其值为 0，标识 S_OK。

图 10-24　用户方法 Sum 的 Request 网络 RPC 报文

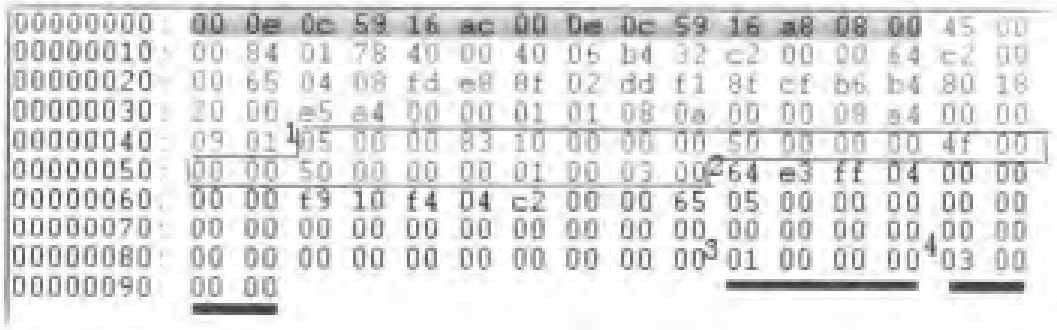


图 10-25　用户方法 Sum 的 Request 网络报文输入参数

1—RPC 公用报文头和 Request PDU 报文头；2—存根数据（包括字段 3 和 4）；3—输入参数 1；4—参数输入 3

图 10-26　服务器对应方法 Sum 的 Response 网络 RPC 报文

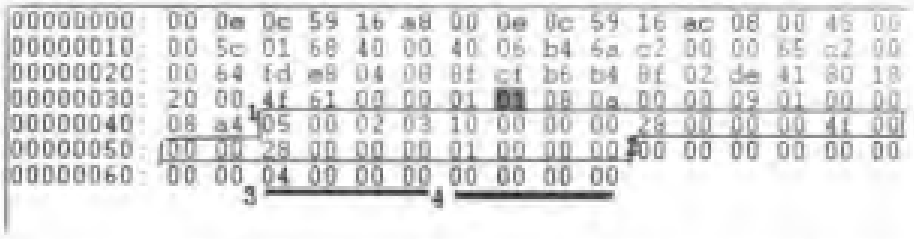


图 10-27　服务器对应方法 Sum 的 Response 网络报文输出参数及其返回值

1—RPC 公用报文头和 Request PDU 报文头；2—存根数据（包括字段 3 和 4）；
3—输出参数 4；4—返回的 HRESULT 值 0，表示 S_OK

上述说明的客户在调用接口方法时的静态使用，方法 SumArray 则说明了在调用接口时存根数据的动态变化，在计算一组整型数的和时，需要指定数组的个数及其值，数组个数的变化会导致存根数据的长度有所变化。其操作码为 4 表示该操作为存根内存管理操作，调用标识为 80(0x00000050)，存根代码长度为 100 个字节，如图 10-28 所示。图 10-29 解析了方法 SumArray 客户方请求报文的数据项，字段 3 说明了数组的长度为 10（0x0000000a），字段 4~13 为 10 个数组的值，对应 1~10，要求服务器计算这十个数据的和。

图 10-28　用户方法 SumArray 的 Request 网络 RPC 报文

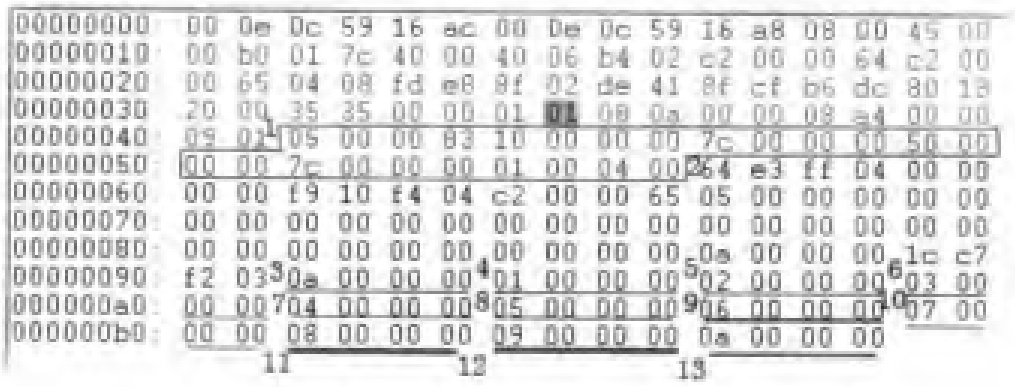


图 10-29　用户方法 SumArray 的 Request 网络报文输入参数

1—RPC 公用报文头和 Response PDU 报文头；2—存根数据（包括字段 3～13）；3—数组的长度，为 10(0x0000000a)；4—数组的第 1 个值，为 1(0x00000001)；5—数组的第 2 个值，为 2(0x00000002)；6—数组的第 3 个值，为 3(0x00000003)；7—数组的第 4 个值，为 4(0x00000004)；8—数组的第 5 个值，为 5(0x00000005)；9—数组的第 6 个值，为 6(0x00000006)；10—数组的第 7 个值，为 7(0x00000007)；11—数组的第 8 个值，为 8(0x00000008)；12—数组的第 9 个值，为 9(0x00000009)；13—数组的第 10 个值，为 10(0x0000000a)

服务器在收到客户方的请求报文后，计算数组的和，并将计算的 $\sum_{i=1}^{10} i$ 值返回给客户端，其过程与 Sum 的应答报文类似。图 10-30 为监听到的对应方法 SumArray 的服务器方的应答报文，其返回的调用标识与客户方请求调用标识一致，在存根数据中存储了返回给客户方的数组计算结果，如图 10-31 所示。

图 10-30　用户方法 SumArray 的 Response 网络 RPC 报文

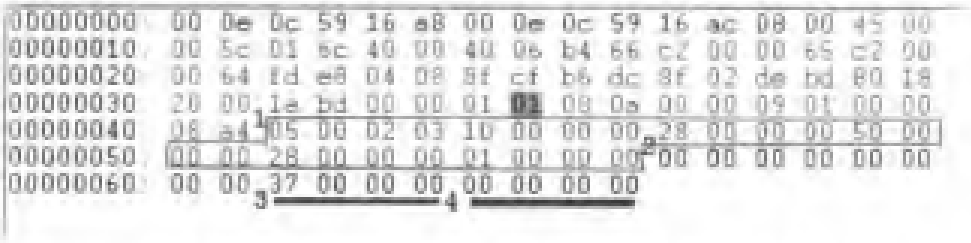


图 10-31　服务器对应方法 SumArray 的 Response 网络报文输出参数及其返回值

1—RPC 公用报文头和 Request PDU 报文头；2—存根数据（包括字段 3 和 4）；
3—输出参数 55（0x00000037）；4—返回的 HRESULT 值 0，表示 S_OK

第 11 章　嵌入式实时系统 CORBA 技术

CORBA（公共对象请求代理体系结构，Common Object Request Broker Architecture）是一组标准，用来定义“分布式对象系统”互操作的一组规范，由 OMG（对象管理组织，Object Menagement Group）作为发起和标准的制定单位。OMG 成立于 1989 年，到目前为至，由 800 多家公司和单位组成，几乎包括了所有有影响的信息系统提供商、软件开发商和用户，其宗旨是促进面向对象技术在软件开发中的应用。

CORBA 的目的是定义一套协议，符合这个协议的对象可以互相交互，不论它们是用什么样的语言写的，不论它们运行于什么样的机器和操作系统。为了达到这个目标，CORBA 制定了一套对象间通信的协议，通信介质被称为 ORB（对象请求代理，Object Request Broker），它负责在对象之间传递消息。如果对象在同一台机器上，ORB 可以采用一些 IPC 技术来优化消息的传递；如果在不同的机器上，则使用 IIOP 或 GIOP 协议（可以建立在任何网络通信协议之上）。IIOP（互联网内部对象请求代理协议，Internet Inter-ORB Protocol）就是基于 IP 的网络通信协议，是为方便在 Internet 上的 CORBA 应用而设计的。

CORBA 提供了 IDL 到 C、C++、Java 和 COBOL 等语言的映射机制——IDL 编译器（见图 11-1）。IDL 编译器可以生成服务器方的框架代码（Skelton）和客户端的存根代码（Stub），通过分别与客户端和服务端程序的联编，即可得到相应的服务器和客户方执行代码。

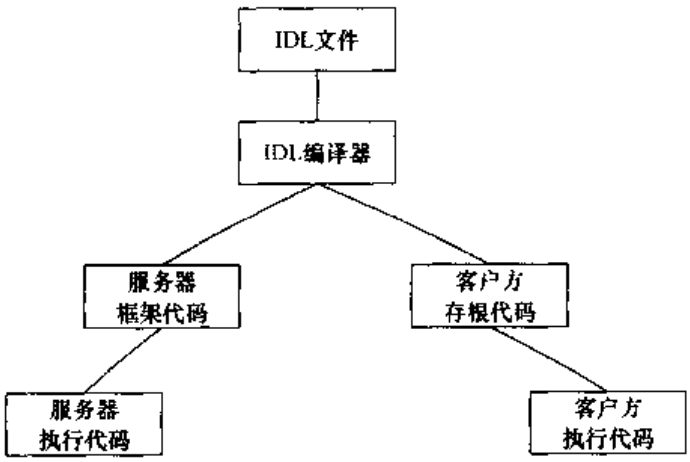


图 11-1　CORBA 的 IDL 编译器体系结构

CORBA 同时提供了一系列的公共服务规范——COSS，其中包括名字服务、永久对象服务、生命周期服务、事务处理服务、对象事件服务和安全服务等，它们相当于一类用于企业级计算的公共构件。COBRA 是对象通信的中间件构件，COBRA 的角色定义在 OMG 的对象管理框架（Object Management Architecture，OMA）中。

目前 OMG 主要开发了：

（1）参考结构（Reference Architecture，OMA）。

（2）概念模型（Conceptual Model，Core Object Model）。

OMA 定义了分布式对象计算中的高层次的抽象，多种特性和必备服务。OMA 包括 4 个主要结构构件（元素），如图 11-2 所示。

（1）对象请求代理（Object Request Broker，ORB）。

（2）对象服务（Object Services，OS）。

（3）公共服务（Common Facilities，CF）。

（4）应用程序对象（Application Objects，AO）。

CORBA 是一种语言中性的软件构件模型，可以跨越不同的网络、不同的机器和不同的操作系统，实现分布对象之间的互操作。

对象：封装的实体，可为客户提供一个或者多个服务。

请求：对来自客户方的服务简称为请求，或者客户请求。与请求相关的还有操作、对象、参数和请求的上下文。

操作：所请求的服务。

接口：一组操作的集合，通过这些接口，客户可以完成对象的请求。

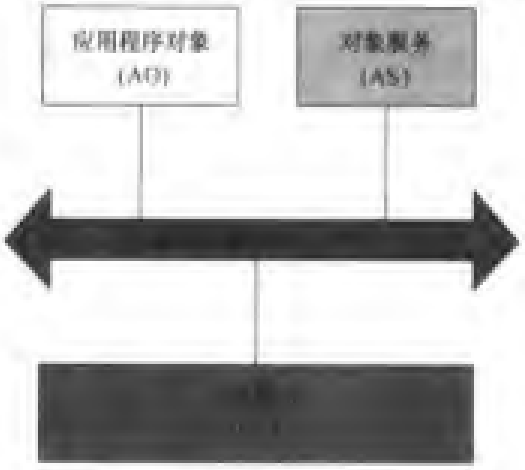


图 11-2　CORBA 元素组成及关系图

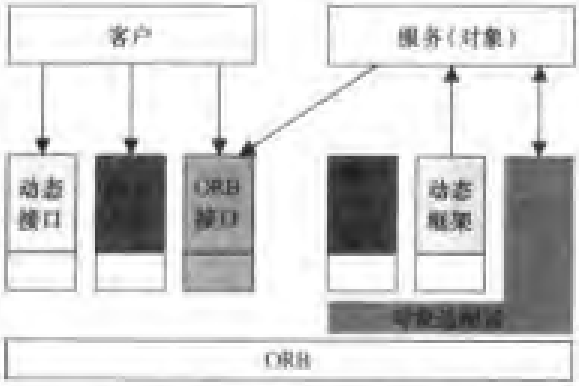


图 11-3　对象请求接口体系结构

本章主要对嵌入式实时系统下用到的 CORBA 技术进行概述。嵌入式实时系统下常用的主要有最小 CORBA（Minimum CORBA）和实时 CORBA（RT-CORBA），最后对 VxWorks 嵌入式实时操作系统下的 CORBA 技术进行了说明。

11.1　最小 CORBA（Minimum CORBA）

Minimum CORBA 是 CORBA 的一个子集，其主要目的是满足有限资源的系统。某些应用系统中 CORBA 过于庞大，难以满足空间和性能的要求，这种情况需要对 CORBA 进行剪裁，剪裁的 CORBA 版本称之为“Minimum CORBA”，即最小 CORBA。最小 CORBA 的服务和 CORBA 安全是 CORBA 规范的扩充选项。

11.1.1　IDL 的兼容性

Minimum CORBA 规范支持 OMG 定义的所有 IDL，包括 IDL 语法和语义，这为 Minimum

CORBA 提供了与标准 CORBA 的最大兼容性。

11.1.2　ORB 接口剪裁说明

最小 CORBA 在 ORB 接口方面做了一些删除。

（1）删除了 create_list 和 create_operation_list 操作，因为其主要目的是支持动态接口调用（DII）。

（2）删除了 Context 对象，因为其为 DII 的一部分。同样删除了与 context 对象相关的操作 get_default_context。

（3）操作 get_current 删除，该操作从 CORBA2.2 才有。

（4）删除了 work_pending、perform_work 和 shutdown 操作，因为这些操作仅用在某些类型的 CORBA，但对于基本 ORB 操作显得多余。需要注意的是仍保留了 run 操作，在单线程模型中有必要为服务器提供初始化代码，在多线程模式下，对 run 操作进行了封装。

详细的 IDL 说明如下：

```
module CORBA {
    typedef unsigned short ServiceType;
    typedef unsigned long ServiceOption;
    typedef unsigned long ServiceDetailType;

    const ServiceType Security = 1;

    struct ServiceDetail {
        ServiceDetailType service_detail_type;
        sequence <octet> service_detail;
    };

    struct ServiceInformation {
        sequence <ServiceOption> service_options;
        sequence <ServiceDetail> service_details;
    };

    interface ORB {
        string object_to_string (in Object obj);
        Object string_to_object (in string str);

        Status create_list(in long count,out NVList new_list);
        Status create_operation_list (in OperationDef oper,out NVList
new_list);

        Status get_default_context (out Context ctx);

        boolean get_service_information (in ServiceType service_type,out
ServiceInformation service_information);
```

删除的 create_list 和 create_operation_list 操作

删除 Context 对象，删除与之相关的操作 get_default_context

```
        // get_current deprecated operation - should not be used by new code
        // new code should use resolve_initial_reference operation instead
        Current get_current();

        //Obtaining Initial Object References
        typedef string ObjectId;
        typedef sequence <ObjectId> ObjectIdList;

        exception InvalidName {};
        ObjectIdList list_initial_services ();
        Object resolve_initial_references (in ObjectId identifier)raises
(InvalidName);

        boolean work_pending();
        void perform_work();
        void shutdown( in boolean wait_for_completion );
        void run();
    };
    ...
};
```

删除操作 get_current

删除操作 work_pending perform_work shutdown

11.1.3 对象（object）剪裁说明

由于删除了接口池（IR），所以以下操作也被删除。

（1）操作 get_interface 删除。

（2）操作 get_implementation 删除，该操作从 CORBA2.2 才有。

（3）操作 is_a、non_existent 删除，create_request 删除，主要是不支持 DII。

（4）删除 ConstructionPolicy 接口及其支持的 SecConstruction。

```
module CORBA {
    ...
    interface Object {
        ImplementationDef get_implementation ();
        InterfaceDef get_interface ();
        boolean is_nil();
        Object duplicate ();
        void release ();
        boolean is_a (in string logical_type_id);
        boolean non_existent();
        boolean is_equivalent (in Object other_object);
        unsigned long hash(in unsigned long maximum);

        Status create_request (in Context ctx,in Identifier operation,in NVList
arg_list,
                inout NamedValue result,out Request request,in Flags req_flags);

        Policy get_policy (in PolicyType policy_type);
        DomainManagersList get_domain_managers ();
```

删除的操作 get_interface 和 get_implementation

删除的操作 is_a 和 non_existent

删除的操作 create_request

```
        };

        //ORB Initialization
        typedef string ORBid;
        typedef sequence <string> arg_list;
        ORB ORB_init (inout arg_list argv, in ORBid orb_identifier);

        //Current Object
        interface Current {};

        //Policy Object
        typedef unsigned long PolicyType;

        // Basic IDL definition
        interface Policy {
            readonly attribute PolicyType policy_type;
            Policy copy();
            void destroy();
        };

        typedef sequence <Policy> PolicyList;

        //Domain management operations
        interface DomainManager {
            Policy get_domain_policy (in PolicyType policy_type);
        };

        const PolicyType SecConstruction = 11;
        interface ConstructionPolicy: Policy {
        void make_domain_manager(in CORBA::InterfaceDef object_type,in boolean
constr_policy);
        };

        typedef sequence <DomainManager> DomainManagerList;
        ...
    };
```

删除 ConstructionPolicy 接口及其支持的 SecConstruction

11.1.4　DII 及 DSI 接口剪裁说明

（1）DII（动态接口调用）。由于不支持 DII，删除了动态请求接口（Dynamic Invocation Interface），同样 NamedValue 类型及 NVList 也删除。

（2）DSI（动态框架调用）。由于不支持 DSI，删除了动态框架调用（Dynamic Skeleton Interface）。

11.1.5　动态 Any 剪裁说明

删除 Any 值的动态管理。

11.1.6 接口池（IR）剪裁说明

因为没有动态类型编程模型，删除接口池（IR），但保留了 RepositoryId 和 TypeCode 部分接口。

接口池对象（IRObject）时基类，接口池的删除导致派生与基类的其他接口也将删除，删除的接口主要有：

- Contained 接口
- Repository 接口
- Container 接口
- ModuleDef 接口
- ConstantDef 接口
- IDLType 接口
- StructDef 接口
- UnionDef 接口
- EnumDef 接口
- AliasDef 接口
- InterfaceDef 接口
- PrimitiveDef 接口
- StringDef 接口
- WstringDef 接口
- FixedDef 接口
- SequenceDef 接口
- ArrayDef 接口
- TypedefDef 接口
- ExceptionDef 接口
- AttributeDef 接口
- OperationDef 接口
- InterfaceDef 接口

它们与接口池（IRObject）对象的关系图如图 11-4 所示。

TypeCode 接口仅保留了 id，kind 和 name 操作，其他操作予以删除，其删除的操作主要有：

member_count 、 member_name 、 member_type 、 member_label 、 discriminator_type 、default_index、length、content_type、fixed_digits、fixed_scale、param_count、parameter、BOUNDS 异常。

所有与 TypeCode 相关的创建操作也被删除，包括：

create_struct_tc、create_union_tc、create_enum_tc、create_alias_tc、create_exception_tc、create_interface_tc、create_string_tc、create_wstring_tc、create_sequence_tc、create_recursive_sequence_tc、create_array_tc。

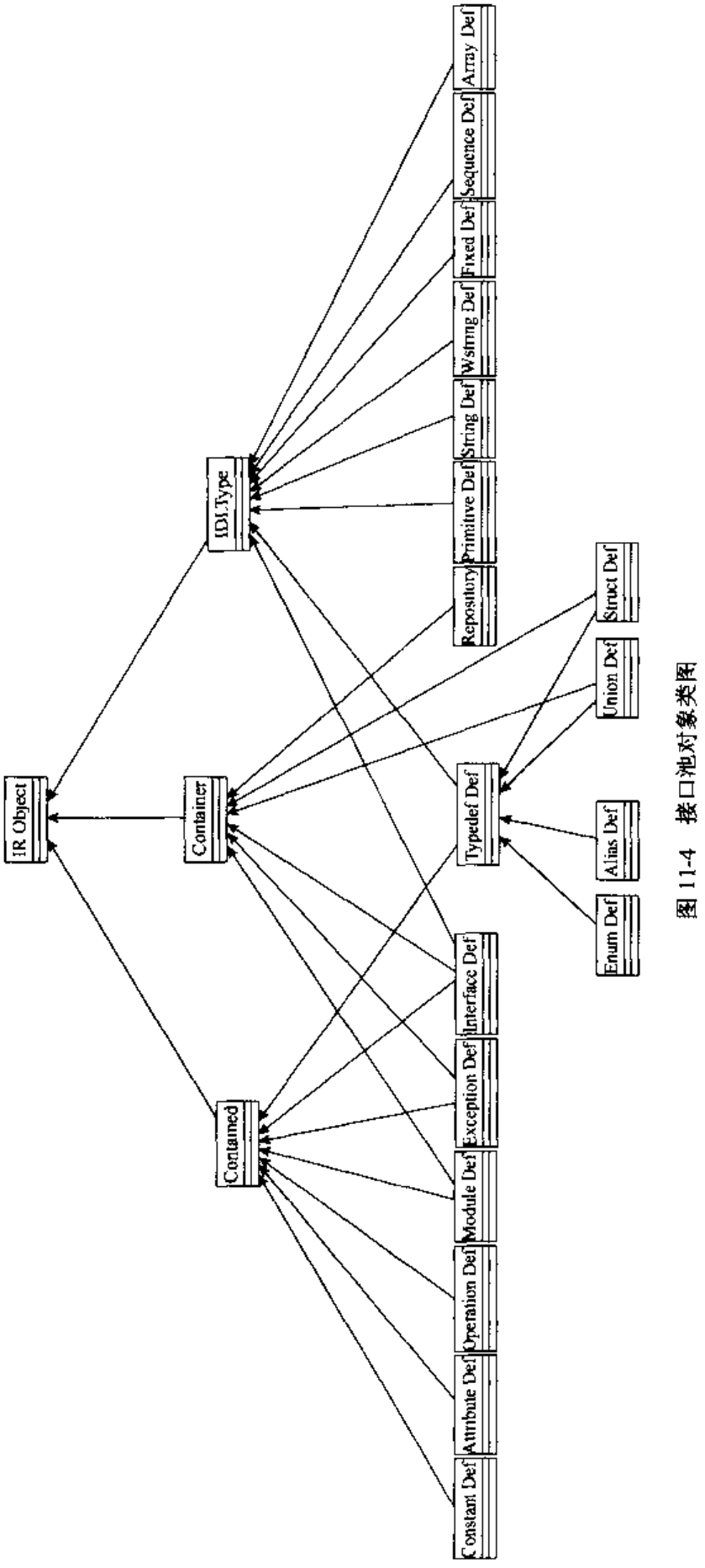


图 11-4　接口池对象类图

其 IDL 定义如下：

```
module CORBA {
    typedef string Identifier;
    typedef string ScopedName;
    typedef string RepositoryId;

    enum DefinitionKind {
    dk_none, dk_all,dk_Attribute, dk_Constant, dk_Exception, dk_Interface,
    dk_Module, dk_Operation, dk_Typedef,dk_Alias, dk_Struct, dk_Union,
dk_Enum,
    dk_Primitive, dk_String, dk_Sequence, dk_Array,dk_Repository,
    dk_Wstring, dk_Fixed
    };

    interface IRObject {
    // read interface
    readonly attribute DefinitionKind def_kind;
    // write interface
    void destroy ();
    };

    typedef string VersionSpec;

    interface Contained;
    interface Repository;
    interface Container;

    interface Contained : IRObject {
    ...//为了简化省略了接口内容，下同
    };

    interface ModuleDef;
    interface ConstantDef;
    interface IDLType;
    interface StructDef;
    interface UnionDef;
    interface EnumDef;
    interface AliasDef;
    interface InterfaceDef;
    typedef sequence <InterfaceDef> InterfaceDefSeq;
    typedef sequence <Contained> ContainedSeq;

    struct StructMember {
    Identifier name;
    TypeCode type;
    IDLType type_def;
    };

    typedef sequence <StructMember> StructMemberSeq;
```

枚举 DefinitionKind 用在 IRObject 中，删除了接口池，此枚举类型属于多余声明

删除接口池对象

删除接口池对象

```
struct UnionMember {
    Identifier name;
    any label;
    TypeCode type;
    IDLType type_def;
};

typedef sequence <UnionMember> UnionMemberSeq;
typedef sequence <Identifier> EnumMemberSeq;

interface Container : IRObject {
...
};

interface IDLType : IRObject {
    readonly attribute TypeCode type;
};

interface PrimitiveDef;
interface StringDef;
interface SequenceDef;
interface ArrayDef;

enum PrimitiveKind {
    pk_null, pk_void, pk_short, pk_long, pk_ushort, pk_ulong,
    pk_float, pk_double, pk_boolean, pk_char, pk_octet,
    pk_any, pk_TypeCode, pk_Principal, pk_string, pk_objref,
    pk_longlong, pk_ulonglong, pk_longdouble, pk_wchar, pk_wstring
};

interface Repository : Container {
...
};

interface ModuleDef : Container, Contained {
};

struct ModuleDescription {
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
};

interface ConstantDef : Contained {
    readonly attribute TypeCode type;
    attribute IDLType type_def;
    attribute any value;
};
```

```
struct ConstantDescription {
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    TypeCode type;
    any value;
};

interface TypedefDef : Contained, IDLType {
};

struct TypeDescription {
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    TypeCode type;
};

interface StructDef : TypedefDef, Container {
    attribute StructMemberSeq members;
};

interface UnionDef : TypedefDef, Container {
    readonly attribute TypeCode discriminator_type;
    attribute IDLType discriminator_type_def;
    attribute UnionMemberSeq members;
};

interface EnumDef : TypedefDef {
    attribute EnumMemberSeq members;
};

interface AliasDef : TypedefDef {
    attribute IDLType original_type_def;
};

interface PrimitiveDef: IDLType {
    readonly attribute PrimitiveKind kind;
};

interface StringDef : IDLType {
    attribute unsigned long bound;
};

interface WstringDef : IDLType {
    attribute unsigned long bound;
};
```

```
interface FixedDef : IDLType {
    attribute unsigned short digits;
    attribute short scale;
};

interface SequenceDef : IDLType {
    attribute unsigned long bound;
    readonly attribute TypeCode element_type;
    attribute IDLType element_type_def;
};

interface ArrayDef : IDLType {
    attribute unsigned long length;
    readonly attribute TypeCode element_type;
    attribute IDLType element_type_def;
};

interface ExceptionDef : Contained, Container {
    readonly attribute TypeCode type;
    attribute StructMemberSeq members;
};

struct ExceptionDescription {
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    TypeCode type;
};
enum AttributeMode {ATTR_NORMAL, ATTR_READONLY};

interface AttributeDef : Contained {
    readonly attribute TypeCode type;
    attribute IDLType type_def;
    attribute AttributeMode mode;
};

struct AttributeDescription {
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    TypeCode type;
    AttributeMode mode;
};

enum OperationMode {OP_NORMAL, OP_ONEWAY};
enum ParameterMode {PARAM_IN, PARAM_OUT, PARAM_INOUT};
struct ParameterDescription {
```

```
    Identifier name;
    TypeCode type;
    IDLType type_def;
    ParameterMode mode;
};
typedef sequence <ParameterDescription> ParDescriptionSeq;
typedef Identifier ContentIdentifier;
typedef sequence <ContentIdentifier> ContentIdSeq;
typedef sequence <ExceptionDef> ExceptionDefSeq;
typedef sequence <ExceptionDescription> ExcDescriptionSeq;

interface OperationDef : Contained {
...
};

struct OperationDescription {
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    TypeCode result;
    OperationMode mode;
    ContentIdSeq contexts;
    ParDescriptionSeq parameters;
    ExcDescriptionSeq exceptions;
};

typedef sequence <RepositoryId> RepositoryIdSeq;
typedef sequence <OperationDescription> OpDescriptionSeq;
typedef sequence <AttributeDescription> AttrDescriptionSeq;

interface InterfaceDef : Container, Contained, IDLType {
...
};

enum TCKind {
    tk_null, tk_void,tk_short, tk_long, tk_ushort, tk_ulong,
    tk_float, tk_double, tk_boolean, tk_char,
    tk_octet, tk_any, tk_TypeCode, tk_Principal, tk_objref,
    tk_struct, tk_union, tk_enum, tk_string,
    tk_sequence, tk_array, tk_alias, tk_except
    tk_longlong, tk_ulonglong, tk_longdouble,
    tk_wchar, tk_wstring, tk_fixed
};

interface TypeCode {
    exception Bounds {};
    exception BadKind {};

    // for all TypeCode kinds
```

```
            boolean equal (in TypeCode tc);
            TCKind kind ();

            // for tk_objref, tk_struct, tk_union, tk_enum, tk_alias, and tk_except
            RepositoryId id () raises (BadKind);
            // for tk_objref, tk_struct, tk_union, tk_enum, tk_alias, and tk_except
            Identifier name () raises (BadKind);

            // for tk_struct, tk_union, tk_enum, and tk_except
            unsigned long member_count () raises (BadKind);
            Identifier member_name (in unsigned long index) raises (BadKind,Bounds);
            // for tk_struct, tk_union, and tk_except
            TypeCode member_type (in unsigned long index) raises (BadKind,Bounds);
            // for tk_union
            any member_label (in unsigned long index) raises (BadKind, Bounds);
            TypeCode discriminator_type () raises (BadKind);
            long default_index () raises (BadKind);
            // for tk_string, tk_sequence, and tk_array
            unsigned long length () raises (BadKind);
            // for tk_sequence, tk_array, and tk_alias
            TypeCode content_type () raises (BadKind);
            // for tk_fixed
            unsigned short fixed_digits() raises (BadKind);
            short fixed_scale() raises (BadKind);
            // deprecated interface
            long param_count ();
            any parameter (in long index) raises (Bounds);
        };

        interface ORB {
            // other operations ...
        TypeCode create_struct_tc (in RepositoryId id,in Identifier name,in
StructMemberSeq members);
        TypeCode create_union_tc (in RepositoryId id,in Identifier name,in TypeCode
discriminator_type,in UnionMemberSeq members);
        TypeCode create_enum_tc (in RepositoryId id,in Identifier name,in
EnumMemberSeq members);
        TypeCode create_alias_tc (in RepositoryId id,in Identifier name,in TypeCode
original_type);
        TypeCode create_exception_tc (in RepositoryId id,in Identifier name,in
StructMemberSeq members);
        TypeCode create_interface_tc (in RepositoryId id,in Identifier name);
        TypeCode create_string_tc (in unsigned long bound);
        TypeCode create_wstring_tc (in unsigned long bound);
        TypeCode create_fixed_tc (in unsigned short digits,in short scale);
        TypeCode create_sequence_tc (in unsigned long bound,in TypeCode element_type);
        TypeCode create_recursive_sequence_tc(in unsigned long bound,in unsigned
long offset);
        TypeCode create_array_tc (in unsigned long length,in TypeCode element_type);
        ...
    };
```

11.1.7 可移植对象适配器（POA）剪裁说明

Minimum CORBA 支持 POA 接口和策略的子集，所保留的可以支持不同 minimum CORBA 以及 minimum CORBA 与标准 CORBA 之间的可移植性和互操作性。

删除 Policy 对象及其相关策略。删除的策略对象如下：

（1）ThreadPolicy。

（2）ImplicitActivationPolicy。

（3）ServantRetentionPolicy。

（4）RequestProcessingPolicy。

从策略类厂中主要删除的操作有：

（1）create_thread_policy。

（2）create_implicit_activation_policy。

（3）create_servant_retention_policy。

（4）create_request_processing_policy。

POAManager 对象保留，主要是考虑到操作 create_POA，唯一保留没有删除的是 activate 操作。

AdapterActivator 对象删除。

删除了 ServantManagers，操作 get_servant_manager 和 set_servant_manager 也被删除。由于 RequestProcessingPolicy 不支持 USE_DEFAULT_SERVANT 选项，删除了操作 get_servant 和 set_servant。

删除了 the_activator 属性。

完全支持 PortableServer::Current 对象，主要考虑的是移植性和互操作性。

ServantManagers 对象删除，由此派生的接口 ServantActivator 和 ServantLocator 也删除。

1. 策略

支持 CORBA 默认的策略值，最小 CORBA 的 RootPOA 是 CORBA 的 RootPOA 的子集。

2. 线程策略（ThreadPolicy）

唯一的线程策略是 ORB_CTRL_MODEL，删除了 SINGLE_THREAD_MODEL 策略因为对基本的 ORB 操作是不需要的。

3. 生命期策略（LifespanPolicy）

支持 LifespanPolicy：TRANSIENT 和 PERSISTENT。PERSISTENT 策略保留，但不做任何事情。

4. 对象 ID 唯一策略（ObjectIdUniquenessPolicy）

支持 UNIQUE_ID 和 MULTIPLE_ID。

5. ID 产生策略（IdAssignmentPolicy）

支持 SYSTEM_ID 和 USER_ID。

6. 服务器驻留策略（ServantRetentionPolicy）

仅支持 RETAIN ServantRetentionPolicy，删除 NON_RETAIN。

7. 请求出路策略（RequestProcessingPolicy）

仅支持 USE_ACTIVE_OBJECT_MAP_ONLY RequestProcessingPolicy，USE_DEPAULT_

SERVANT 和 USE_SERVANT_MANAGER 策略被删除。

8. 隐式激活策略（ImplicitActivationPolicy）

仅支持 NO_IMPLICIT_ACTIVATION 策略，IMPLICIT_ACTIVATION 删除。

Minimum CORBA OMG IDL 定义：

```
module PortableServer{
    // forward reference
    interface POA;
    native Servant;
    typedef sequence<octet> ObjectId;
    exception ForwardRequest{Object forward_reference;};

    // ***********************************************
    //
    // Policy interfaces
    //
    // ***********************************************
    enum ThreadPolicyValue {ORB_CTRL_MODEL,SINGLE_THREAD_MODEL};
    interface ThreadPolicy : CORBA::Policy{
        readonly attribute ThreadPolicyValue value;
    };

    enum LifespanPolicyValue {TRANSIENT,PERSISTENT};
    interface LifespanPolicy : CORBA::Policy    {
        readonly attribute LifespanPolicyValue value;
    };
    enum IdUniquenessPolicyValue {UNIQUE_ID,MULTIPLE_ID};
    interface IdUniquenessPolicy : CORBA::Policy    {
        readonly attribute IdUniquenessPolicyValue value;
    };
    enum IdAssignmentPolicyValue {USER_ID,SYSTEM_ID};
    interface IdAssignmentPolicy : CORBA::Policy    {
        readonly attribute IdAssignmentPolicyValue value;
    };

    enum ImplicitActivationPolicyValue {IMPLICIT_ACTIVATION,NO_IMPLICIT_ACTIVATION};
    interface ImplicitActivationPolicy : CORBA::Policy{
        readonly attribute ImplicitActivationPolicyValue value;
    };

    enum ServantRetentionPolicyValue {RETAIN,NON_RETAIN};
    interface ServantRetentionPolicy : CORBA::Policy{
        readonly attribute ServantRetentionPolicyValue value;
    };
    Enum RequestProcessingPolicyValue {USE_ACTIVE_OBJECT_MAP_ONLY,USE_DEFAULT_SERVANT,
USE_SERVANT_MANAGER};
    interface RequestProcessingPolicy : CORBA::Policy{
        readonly attribute RequestProcessingPolicyValue value;
    };
    // ***********************************************
    //
```

删除 ThreadPolicy 策略对象

删除 ImplicitActivationPolicy 策略对象

删除 ServantRetentionPolicy 策略对象

删除 RequestProcessingPolicy 策略对象

```
        // POAManager interface
        //
        // ***********************************
        interface POAManager{
            exception AdapterInactive{ };
            void activate( )raises( AdapterInactive );

            void hold_requests( in boolean wait_for_completion )raises( AdapterInactive );
            void discard_requests( in boolean wait_for_completion )raises( AdapterInactive );
            void deactivate( in boolean etherealize_objects,in boolean wait_for_completion )
                raises( AdapterInactive );
        };

        // ***********************************************
        //
        // AdapterActivator interface
        //
        // ***********************************************
        interface AdapterActivator{
            boolean unknown_adapter( in POA parent, in string name );
        };

        // ***********************************************
        //
        // ServantManager interface
        //
        // ***********************************************
        interface ServantManager{ };
        interface ServantActivator : ServantManager {
            Servant incarnate (in ObjectId oid,in POA adapter ) raises
( ForwardRequest );
            void etherealize (in ObjectId oid,in POA adapter,in Servant serv,
                in boolean cleanup_in_progress,in boolean remaining_activations );
        };

        interface ServantLocator : ServantManager {
            native Cookie;
            Servant preinvoke(in ObjectId oid,in POA adapter,in CORBA::Identifier
operation,
                out Cookie the_cookie ) raises ( ForwardRequest );
            void postinvoke(in ObjectId oid,in POA adapter,in CORBA::Identifier
operation,
                in Cookie the_cookie,in Servant the_servant );
        };

        // ***************************************************
        //
        // POA interface
        //
        // ***************************************************
        interface POA
        {
```

POAManager 接口仅保留了 activate 操作，删除了其他操作

删除 AdapterActivator 对象

ServantManagers 对象删除，由此派生的接口 ServantActivator 和 ServantLocator 也删除

```
        exception AdapterAlreadyExists {};
        exception AdapterInactive { };
        exception AdapterNonExistent { };
        exception InvalidPolicy { unsigned short index; };
        exception NoServant { };
        exception ObjectAlreadyActive { };
        exception ObjectNotActive { };
        exception ServantAlreadyActive { };
        exception ServantNotActive { };
        exception WrongAdapter { };
        exception WrongPolicy { };

        //--------------------------------------------------
        //
        // POA creation and destruction
        //
        //--------------------------------------------------
        POA create_POA( in string adapter_name,in POAManager a_POAManager,
            in CORBA::PolicyList policies )raises ( AdapterAlreadyExists,
InvalidPolicy );
        POA find_POA(in string adapter_name,in boolean activate_it)
            raises (AdapterNonExistent );
        void destroy( in boolean etherealize_objects,in boolean
wait_for_completion );

        // ****************************************************
        //
        // Factories for Policy objects
        //
        // ****************************************************
        ThreadPolicy create_thread_policy( in ThreadPolicyValue value );
        LifespanPolicy create_lifespan_policy( in LifespanPolicyValue value );
        IdUniquenessPolicy create_id_uniqueness_policy(in
IdUniquenessPolicyValue value );
        IdAssignmentPolicy create_id_assignment_policy( in IdAssignment-
PolicyValue value );

        ImplicitActivationPolicy
            create_implicit_activation_policy(in ImplicitActivationPolicyValue value );

        ServantRetentionPolicy
            create_servant_retention_policy( in ServantRetentionPolicyValue
value );
        RequestProcessingPolicy
            create_request_processing_policy( in
```

删除了 ThreadPolicy 策略对象，与之相关的 create_thread_policy 操作也被删除

删除了 ImplicitActivationPolicy 策略对象，与之相关的 create_implicit_activation_policy 操作也被删除

删除了 ServantRetentionPolicy 策略对象，与之相关的 create_servant_retention_policy 操作也被删除

~~RequestProcessingPolicyValue value);~~

删除了 RequestProcessingPolicy 策略对象，与之相关的 create_request_processing_policy 操作也被删除

```
        //--------------------------------------------------
        //
        // POA attributes
        //
        //--------------------------------------------------
        readonly attribute string the_name;
        readonly attribute POA the_parent;
        readonly attribute POAManager the_POAManager;
        ~~attribute AdapterActivator the_activator;~~
        //--------------------------------------------------
        //
        // Servant Manager registration:
        //
        //--------------------------------------------------
        ~~ServantManager get_servant_manager( ) raises ( WrongPolicy );~~
        ~~void set_servant_manager( in ServantManager imgr )raises (WrongPolicy );~~

        //--------------------------------------------------
        //
        // operations for the USE_DEFAULT_SERVANT policy
        //
        //--------------------------------------------------
        ~~Servant get_servant( )raises ( NoServant, WrongPolicy );~~
        ~~void set_servant( in Servant p_servant )raises ( WrongPolicy );~~

        // ****************************************************
        //
        // object activation and deactivation
        //
        // ****************************************************
    ObjectId activate_object(in Servant p_servant)raises(ServantAlreadyActive,
WrongPolicy );
    void activate_object_with_id(in ObjectId id,in Servant p_servant )
          raises    (    ServantAlreadyActive,    ObjectAlreadyActive,
  WrongPolicy );
    void deactivate_object( in ObjectId oid )raises ( ObjectNotActive,
WrongPolicy );

        // ****************************************************
        //
        // reference creation operations
        //
        // ****************************************************
    Object create_reference (in CORBA::RepositoryId intf )raises
( WrongPolicy );
    Object create_reference_with_id (in ObjectId oid,in CORBA::RepositoryId
intf )
          raises ( WrongPolicy );

        //--------------------------------------------------
```

由于删除了 ServantManagers 对象，操作 get_servant_manager 和 set_servant_manager 也被删除

删除了操作 get_servant 和 set_servant

删除了 the_activator 属性

```
            //
            // Identity mapping operations:
            //
            //--------------------------------------------------
        ObjectId servant_to_id( in Servant p_servant )raises ( ServantNotActive,
WrongPolicy );
        Object servant_to_reference(in Servant p_servant )raises (ServantNotActive,
WrongPolicy );
        Servant reference_to_servant( in Object reference )
              raises ( ObjectNotActive, WrongAdapter, WrongPolicy );
        ObjectId reference_to_id( in Object reference )raises ( WrongAdapter,
WrongPolicy );
        Servant id_to_servant( in ObjectId oid )raises ( ObjectNotActive,
WrongPolicy );
        Object id_to_reference( in ObjectId oid )raises ( ObjectNotActive,
WrongPolicy );
    };

    // ****************************************************
    //
    // Current interface
    //
    // ****************************************************
    interface Current : CORBA::Current
    {
        exception NoContext { };
        POA get_POA( ) raises ( NoContext );
        bjectId get_object_id( ) raises ( NoContext );
    };
    };
```

11.2　实时 CORBA（RT-CORBA）

实时 CORBA 主要关注的是实时系统的开发。在实时系统中，一般有“硬实时”和“软实时”之分，具有不同的资源控制和调度算法。实时 CORBA 是 CORBA2.2 的扩充。满足实时性要求，并且要求分布式线程，所有实时 CORBA IDL 定义包括在新的模块 RTCORBA 和 RTPortableServer 中。实时 CORBA 的体系结构如图 11-5 所示。

1. RT-ORB

实时 CORBA 定义了 ORB 接口的扩充，RTCORBA::RTORB，主要处理实时 ORB 的配置，管理实时 CORBA IDL 接口的创建和注销工作。

2. 线程调度

实时 CORBA 使用线程作为调度的基本实体，主要包括线程池(ThreadPool)和实时 CORBA Current 接口。

3. 实时 CORBA 优先级

实时 CORBA 优先级定义了通用的、与平台无关的优先级模式。本地优先级及优先级映射定义了 NativePriority 和 PriorityMapping 接口。

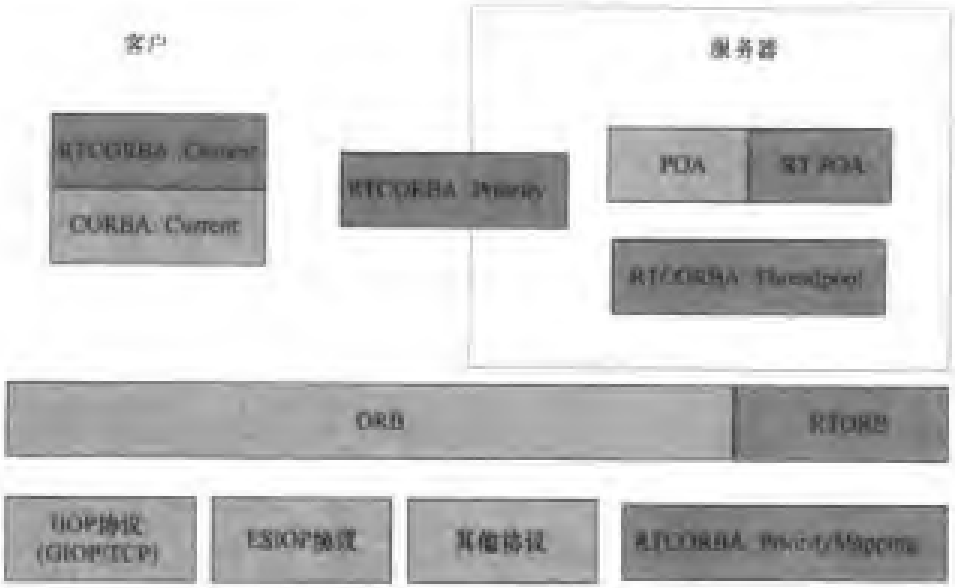


图 11-5　实时 CORBA 模型集

4. 实时 CORBA Current

提供了访问线程优先级的接口。

5. 线程模型

线程模型主要有两种：

（1）客户繁殖优先级模型。服务器使用客户方设定的优先级，使用 PriorityMapping。非实时 CORBA 的 ORB，不使用此方式，用服务器方决定优先级。

（2）服务方指定优先级模型。

6. 实时 CORBA 互斥及优先级继承

Mutex 接口提供了系统资源共享访问的机制，指定了 RTCORBA::Mutex。

```
local interface Mutex {
   void lock( );
   void unlock( );
   boolean try_lock ( in TimeBase::TimeT max_wait );
   // if max_wait = 0 then return immediately
};
local interface RTORB {
   Mutex create_mutex( );
   void destroy_mutex( in Mutex the_mutex );
...
};
```

7. 线程池

实时 CORBA 使用线程池，创建线程池后可以设定其特性，具有如下特性：

（1）预分配线程。

（2）线程分片。

（3）请求缓冲。

8. 实时 CORBA 配置

新的策略需要定义以配置服务方 RT-CORBA：

（1）服务方线程配置（使用线程池 ThreadPools）。

（2）优先级模型（客户繁殖或者服务方指定）。

（3）协议选择。

（4）协议配置。

客户方应采用的一组策略为：

（1）创建客户与服务方绑定优先级。

（2）创建服务方非多方联接。

（3）客户方协议选择和配置。

9. 实时 CORBA 模块

所有由实时 CORBA 定义的 CORBA IDL 均包括在新增模块 RTCORBA 和 RTPortableServer 中。

10. 实时 ORB

实时 CORBA 定义 CORBA::ORB 接口的扩充为 RTCORBA::RTORB，该接口并不派生于 CORBA::ORB，也没有实现继承，只是概念上为 ORB 接口的扩充。对每一个 CORBA::ORB 仅有一个 RTCORBA::RTORB 的实例，欲得到 RTORB 的对象引用须通过调用操作 ORB::resolve_initial_references，其入口参数为对象 ID（ObjectId）“RTORB”。实时 CORBA 定义了 RTCORBA::RTORB 是个局部接口，其声明如下：

```
module RTCORBA {
  local interface RTORB {
  ...
  };
};
```

代码：

```
CORBA::ORB_var orb = CORBA::ORB_init(argc,argv);
CORBA::Ojbect_var obj = orb->resolve_initial_references("RTORB");
RTCORBA::RTORB_var rtorb = RTCORBA::RTORB::_narrow(obj);
```

11. 实时 ORB 初始化

实时 ORB 初始化在 CORBA::ORB_init 操作中进行，也就是说，实时 ORB 的实现是通过 CORBA::ORB_init 来完成的，该操作进行实时 ORB 性能的必备配置。为了保证实时 ORB 的使用优先级，ORB_init 操作其参数 argv 应该具有如下格式：

```
-ORBRTpriorityrange<optional-white-space><short>,<short>
```

其中<short>是字符编码，范围为 0～32767，第 1 个数字应小于第 2 个数字，否则会出现 BAD_PARAM 系统异常。这两个数字代表用户可用 CORBA 优先级范围，该优先级将与 ORB 内部线程优先级相对应，如果 ORB 不能映射这些数字到本地的优先级模式，会产生一个 DATA_CONVERSION 系统异常；如果 ORB 认为优先级范围太小而不能实现正常功能，则会返回一个 INITIALIZE 系统异常。

12. 实时 CORBA 系统异常

实时 CORBA 提供了系统异常，使用与标准 CORBA 相同意义的系统异常，称之为标准系统异常。

（1）INITIALIZE 异常。

映射码：1，ORB 优先级范围太小时产生的异常。

（2）DATA_CONVERSION 异常。

映射码：2，优先级映射对象错误时产生。

（3）NO_RESOURCES 异常。

映射码：2，与请求优先级无连接时产生的异常。

（4）MARSHAL 异常。

映射码：4，试图调度本地对象产生异常。

（5）BAD_INV_ORDER 异常。

映射码：18，试图重新分配优先级时产生的异常。

13. 实时 POA(RTPortableServer::POA)

实时 CORBA 定义了 POA 的扩展，其定义在接口 PortableServer::POA 中。

```
// IDL
module RTPortableServer {
local interface POA : PortableServer::POA {
    Object create_reference_with_priority (in CORBA::RepositoryId intf,
        in RTCORBA::Priority priority )raises ( WrongPolicy );
    Object create_reference_with_id_and_priority (in PortableServer::ObjectId oid,
        in CORBA::RepositoryId intf,in RTCORBA::Priority priority )raises
( WrongPolicy );
    PortableServer::ObjectId activate_object_with_priority (
        in PortableServer::Servant p_servant,in RTCORBA::Priority priority )
        raises ( ServantAlreadyActive, WrongPolicy );
    void activate_object_with_id_and_priority (in PortableServer::ObjectId oid,
        in PortableServer::Servant p_servant,in RTCORBA::Priority priority )
        raises ( ServantAlreadyActive,ObjectAlreadyActive, WrongPolicy );
    };
};
```

通过调用 ORB::resolve_initial_references（“RootPOA”）返回一个接口类型为 RTPortableServer::POA 的对象，实时 POA 与 POA 有两点不同，一方面实时 POA 可以提供较多操作以支持对象级优先级设定；另一方面其实现更加了解扩展中定义的实时策略，注意的是实时 POA 派生于 POA 接口，其同样也支持所有 POA 定义的语法语义。

14. 本地线程优先级

本地线程优先级主要指定优先级值的范围及其变化方向。

```
// IDL
module RTCORBA {
typedef short NativePriority;
};
```

15. CORBA 优先级

由于不同的实时操作系统（RTOS）存在不同于本地线程优先级模式，实时 CORBA 定义了 CORBA 优先级，以统一定义，其类型为 RTCORBA::Priority。可以映射到不同 RTOS 的本地优先级模式，CORBA 优先级提供了不同 RTOS 之间的一个通用的优先级表示方法。也就是

说，CORBA 定义了与操作系统无关的优先级定义，其范围为 0～32768。用户可以将这一优先级映射到本地的优先级。

```
//IDL
module RTCORBA {
    typedef short Priority;
    const Priority minPriority = 0;
    const Priority maxPriority = 32767;
};
```

16. CORBA 优先级映射

在 CORBA 优先级与本地优先级之间，实时 CORBA 定义了优先级映射（PriorityMapping）概念，其 IDL 定义中使用了本地类型，优先级映射机制可以对用户透明。实时 ORB 应提供所支持的每一平台的默认映射，同时还能提供一种机制允许用户定制自己的优先级映射来替代默认映射方式。优先级映射提供了本地优先级与 CORBA 优先级之间的对应关系，操作 to_native 与 to_CORBA 负责实施完成它们之间的转换。

```
// IDL
module RTCORBA {
    native PriorityMapping;
    native PriorityTransform;
};
```

例如：默认方式下，VisiBroker 采用的优先级映射如图 11-6 所示。

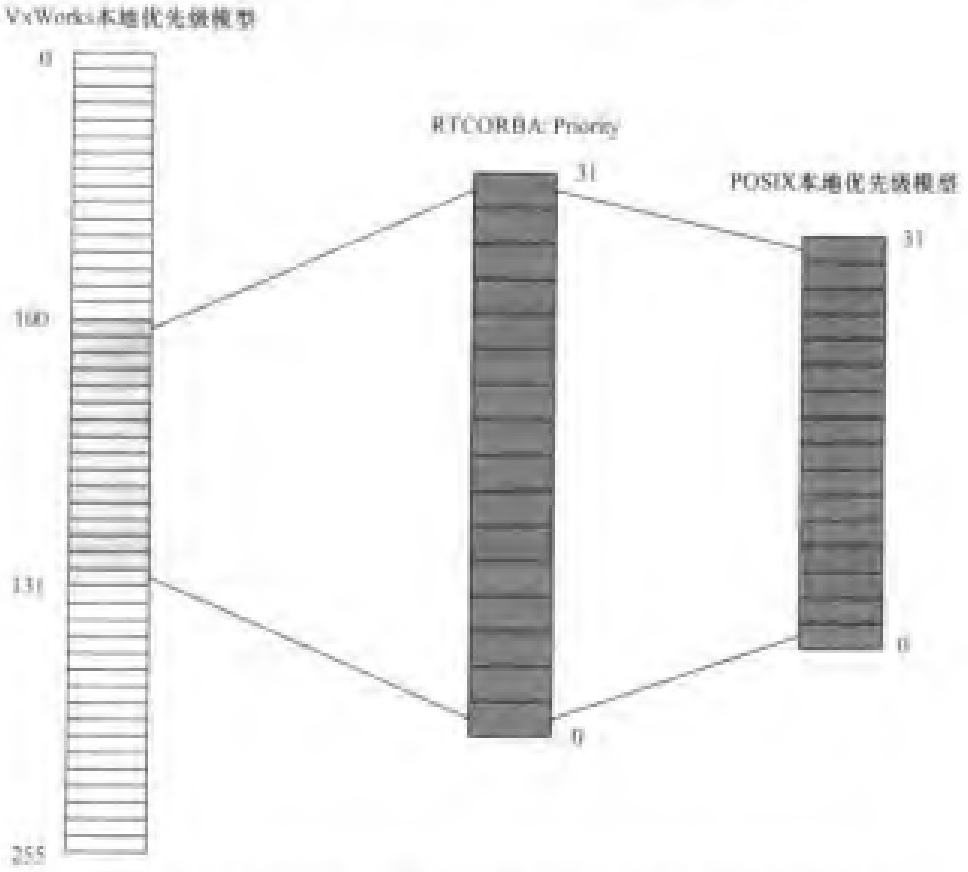


图 11-6　CORBA 优先级与 VxWorks 和 POSIX 本地优先级模型映射关系

17. 实时流 Current

RTCORBA::Current 接口派生于 CORBA::Current，提供访问当前线程的 CORBA 优先级方式，可以通过调用 CORBA::ORB::resolve_initial_references("RTCurrent")操作得到 Current 的一个实例。通过设定 RTCORBA::Current 对象的优先级属性，实时 CORBA 优先级可以对当前线程相关联。如果优先级的值不在 0~32767 之间，会返回 BAD_PARAM 系统异常。一旦线程有一个与其相关的 CORBA 优先级值，通过 CORBA 对象调用的行为依赖于目标对象的优先级模型策略（PriorityModelPolicy）。

```
//IDL
module RTCORBA {
   local interface Current : CORBA::Current {
      attribute Priority base_priority;
   };
};
```

代码：

```
obj = orb->resolve_initial_references("RTCurrent");
RTCORBA::Current_var rt_current = RTCORBA::Current::_narrow(obj);
rt_current->base_priority(20);
```

18. 实时 CORBA 优先级模型

实时 CORBA 支持两种与系统优先级相协调的模型：客户繁殖优先级模型和服务器声明优先级模型。

一般在优先级模型策略中进行声明。

（1）服务器声明优先级模型：对象创建时由服务器负责请求调用的优先级声明。

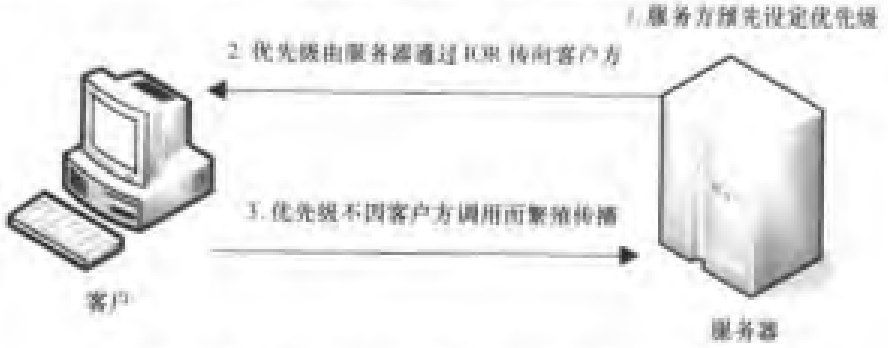


图 11-7 实时 CORBA 的服务器声明优先级模型

（2）客户繁殖优先级模型：请求调用时的优先级由客户方设定，优先级封装在客户的请求报文中，在服务器中设定优先级。

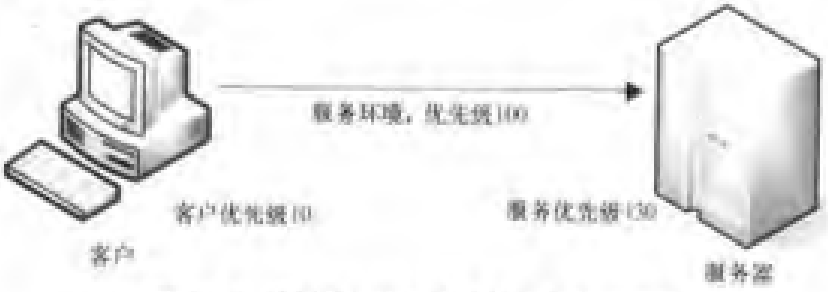


图 11-8 实时 CORBA 客户繁殖优先级模型

19. 优先级模型策略（PriorityModelPolicy）

```
//IDL
module RTCORBA {
    // Priority Model Policy
    const CORBA::PolicyType PRIORITY_MODEL_POLICY_TYPE = 40;
    enum PriorityModel {CLIENT_PROPAGATED,SERVER_DECLARED};
    local interface PriorityModelPolicy : CORBA::Policy {
    readonly attribute PriorityModel priority_model;
    readonly attribute Priority server_priority;
 };
};
```

对指定 POA，当采用服务器声明模型时，server_priority 属性表示由 POA 管理的 CORBA 对象设定优先级，该优先级可以被每一个对象引用所覆盖。当使用客户繁殖模型时，server_priority 属性表示优先级由来自非实时 CORBA ORB 的请求调用来设定。

20. 优先级模型策略范围

在 POA 创建的同时，可以将优先级模型策略（PriorityModelPolicy）应用于实时 POA，主要有两种方式：一种是通过默认 ORB 级来设定；另外一种就是使用 create_POA 操作中的策略参数进行设定。调用 create_priority_model_policy 操作时可以创建优先级模型策略（PriorityModelPolicy）的一个实例。

```
//IDL
module RTCORBA{
    local interface RTORB{
    ...
    PriorityModelPolicy    create_priority_model_policy(in    PriorityModel
priority_mode,
        in Priority server_priority);
    };
};
```

21. 实时 CORBA 线程池管理

```
//IDL
module RTCORBA {
    // Threadpool types
    typedef unsigned long ThreadpoolId;
    struct ThreadpoolLane {
        Priority lane_priority;
        unsigned long static_threads;
        unsigned long dynamic_threads;
    };

    typedef sequence <ThreadpoolLane> ThreadpoolLanes;
    // Threadpool Policy
    const CORBA::PolicyType THREADPOOL_POLICY_TYPE = 41;
    local interface ThreadpoolPolicy : CORBA::Policy {
        readonly attribute ThreadpoolId threadpool;
    };

    local interface RTORB {
```

```
            ...
            ThreadpoolPolicy create_threadpool_policy (in ThreadpoolId threadpool);
            exception InvalidThreadpool {};
            ThreadpoolId create_threadpool (in unsigned long stacksize,in unsigned
long    static_threads,in    unsigned    long    dynamic_threads,in    Priority
default_priority,in    boolean    allow_request_buffering,in    unsigned    long
max_buffered_requests,in unsigned long max_request_buffer_size );
            ThreadpoolId    create_threadpool_with_lanes    (in    unsigned    long
stacksize,in  ThreadpoolLanes  lanes,in  boolean  allow_borrowing,in  boolean
allow_request_buffering,in unsigned long max_buffered_requests,in unsigned long
max_request_buffer_size );
            void  destroy_threadpool  (  in  ThreadpoolId  threadpool  )raises
(InvalidThreadpool);
        };
    };
```

create_threadpool 和 create_threadpool_with_lanes 操作提供了两种不同的线程池创建的方法。可以使用 destroy_threadpool 操作注销创建的线程池，也可以调用 create_threadpool_policy 以创建线程池策略，再使用 POA_create 将创建的线程池策略与不同的 POA 相关联。

22. 网络协议配置

（1）服务方协议策略（ServerProtocolPolicy）。

ServerProtocolPolicy 的创建是通过 create_server_protocol_policy 完成的。可指定使用不同的协议，ProtocolList 指明了其不同协议的顺序。

```
// IDL
module RTCORBA {
    local interface ProtocolProperties {};
    struct Protocol {
        IOP::ProfileId protocol_type;
        ProtocolProperties orb_protocol_properties;
        ProtocolProperties transport_protocol_properties;
    };
    typedef sequence <Protocol> ProtocolList;
    // Server Protocol Policy
    const CORBA::PolicyType SERVER_PROTOCOL_POLICY_TYPE = 42;
    local interface ServerProtocolPolicy : CORBA::Policy {
        readonly attribute ProtocolList protocols;
    };
    local interface RTORB {
        ...
        ServerProtocolPolicy create_server_protocol_policy (in ProtocolList
protocols);
    };
};
```

ServerProtocolPolicy 的创建是通过 create_server_protocol_policy 完成的。可指定使用不同的协议，ProtocolList 指明了其不同协议的顺序。

```
//IDL
module RTCORBA {
    local interface TCPProtocolProperties : ProtocolProperties {
```

```
            attribute long send_buffer_size;
            attribute long recv_buffer_size;
            attribute boolean keep_alive;
            attribute boolean dont_route;
            attribute boolean no_delay;
        };
        local interface RTORB {
        ...
        TCPProtocolProperties    create_tcp_protocol_properties    (in    long
send_buffer_size,
        in long recv_buffer_size,in boolean keep_alive,in boolean dont_route,in
boolean no_delay );
        };
    };
```

TCPProtocolProperties 通过 RTORB 的 create_tcp_protocol_properties 创建。

（2）客户方协议策略（ClientProtocolPolicy）。

```
    // IDL
    module RTCORBA {
        // Client Protocol Policy
        const CORBA::PolicyType CLIENT_PROTOCOL_POLICY_TYPE = 43;
        local interface ClientProtocolPolicy : CORBA::Policy {
            readonly attribute ProtocolList protocols;
        };
        local interface RTORB {
            ...
            ClientProtocolPolicy create_client_protocol_policy (in ProtocolList
protocols);
        };
    };
```

11.3　VisiBroker-RT 编程说明

VisiBroker 是 Borland 公司的产品，VisiBroker RT 是其针对嵌入式应用而推出的支持实时 CORBA 的产品，支持优先级映射，完全符合 CORBA2.6 ORB 核和 GIOP/IIOP1.2 规范，POA 支持多种网络接口，采用动态接口(DII、DSI、POA 策略)，支持 QoS，而且支持事件服务，采用了 SmartAgent 技术，提供先进的 CORBA 命名服务(Naming Service)功能，具备集群(Clustering)、负载均衡（Load Balancing)、容错（Fault Tolerance）以及错误接管（Fail-Over）能力，最大可能地保证分布式应用系统安全而稳定运行。VisiBroker 市场份额的四分之一是在军事和航空航天领域。

1. 硬件开发环境

目标机（TARGET）和主机（HOST）通过以太网相连，建议目标机配置有软盘驱动器，网卡最好为 intel 82557/8/9 或者 3com3c905B 类型的 PCI 网卡。

2. 软件开发环境

（1）主机操作系统建议为 Microsoft Windows XP Professional（Service Pack 2)。选用 Windows 2000 操作系统时需要安装 tornado2.2 的补丁程序。

（2）Tornado Version2.2 开发环境及其 VxWorks5.5 操作系统。

（3）安装 Borland VisiBroker-RT6.0。

3. 设定 Windows 系统中的环境变量

打开控制面板，选择“性能和维护”下的“系统”、“高级”，设定“环境变量”。

（1）增加 VBRT 环境变量，并设定其值为 VisiBroker RT6.0 的根目录，这里设定为 C:\VisiBrokerRT60。

（2）增加 VBROKERDIR 环境变量，并设定其值为%VBRT%。

（3）增加 VBROKER_ADM 环境变量，并设定其值为%VBRT%\var\。

（4）将%VBRT%\bin;%VBRT%\jdk\jdk1.4.2\bin 添加到 path 环境变量中。

（5）如果没有设定 WIND_BASE 环境变量，那么设定其值为 Tornado2.2 的根目录，这里设定为 c:\tornado2.2。

4. 建立开发环境

我们在 X86 平台上，目标机配置 intel 82557/8/9 网卡和软盘驱动器为例，以软盘为引导，建立调试环境，主要的步骤如下。

（1）建立可引导的 VxWorks 映像。

选择 File->New Project 菜单，创建一个 bootable VxWorks Image 的工程，选择板级支持包（BSP）为 PENTIUM，完成操作系统工程文件的创建过程。

（2）修改 BSPNAME/config.h 文件。修改 DEFAULT_BOOT_LINE，找到默认引导行，CPU 类型为 PENTIUM 的引导行改为：

```
"fei(0,0)host:vxworks h=193.0.0.99 e=193.0.0.100 u=wrs pw=wrswrs"
```

其说明如下：

参　数	值	说　明
Boot device	fei	Intel 82557/8/9 网卡，默认网卡
Processor number	(0,0)	
Host name	host	主机名
File name	VxWorks	VxWorks 操作系统文件
Host inet(h)	193.0.0.99	主机网络接口地址（IP）
Inet on Ethernet(e)	193.0.0.100	目标机网络接口地址（IP）
User(u)	wrs	FTP 用户名
Password(pw)	wrswrs	FTP 密码

同时应确保宏可用，即需要定义：

```
#define INCLUDE_FEI_END
```

为了保证键盘和标准输入输出可用，需要将如下代码：

```
#undef INCLUDE_PC_CONSOLE          /* PC keyboard and VGA console */
```

改为

```
#define INCLUDE_PC_CONSOLE         /* PC keyboard and VGA console */
```

（3）打开 VxWorks 标签页，选择 hardware->serial->PC console，单击鼠标右键，将所改选项加入到操作系统中。

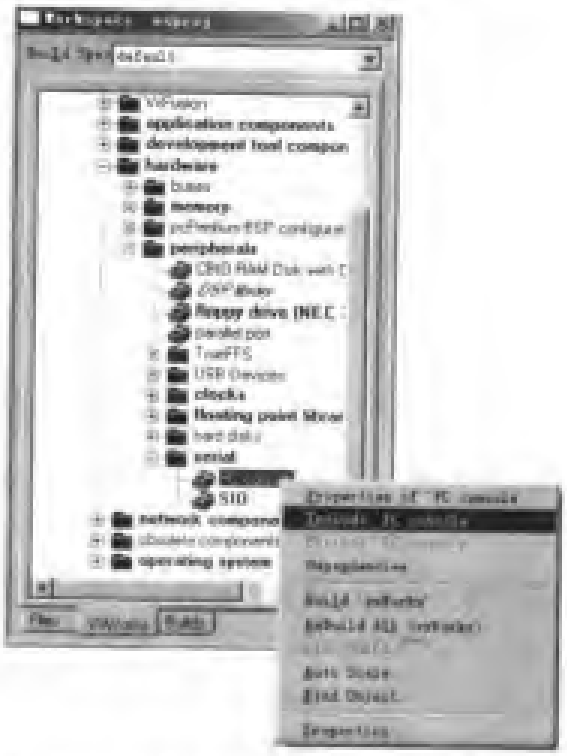


图 11-9　为 VxWorks 操作系统加入终端选项

5. 加入 CORBA 组件

为 VxWorks 操作系统加入 CORBA 组件支持，打开 VxWorks 标签页，可以看到 VisiBroker C++ Components，单击鼠标右键，将该组件加入到操作系统中。

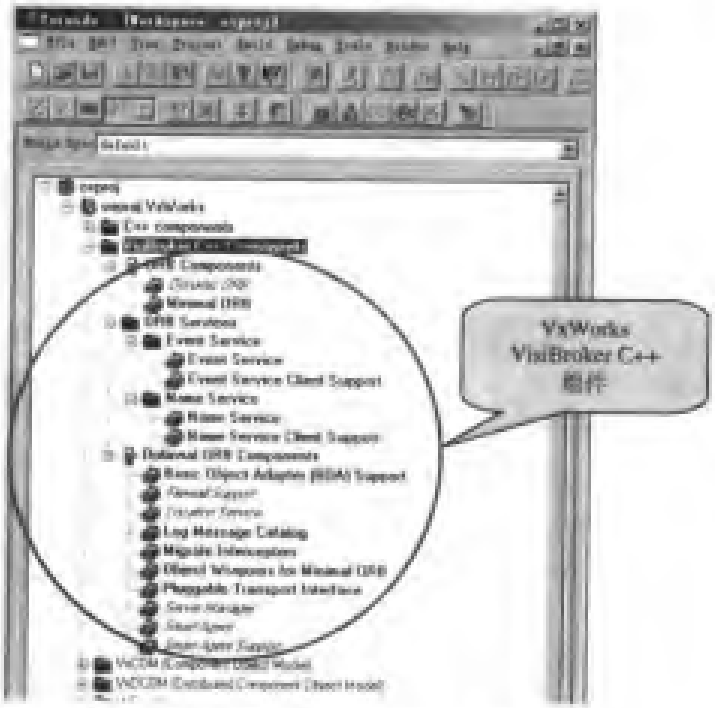


图 11-10　VxWorks 的 VisiBroker C++组件

VisiBroker C++组件主要包括 Minimal ORB、BOA、事件服务和命名服务等。

成功后会弹出用户所加入的 CORBA 组件的目标代码情况。

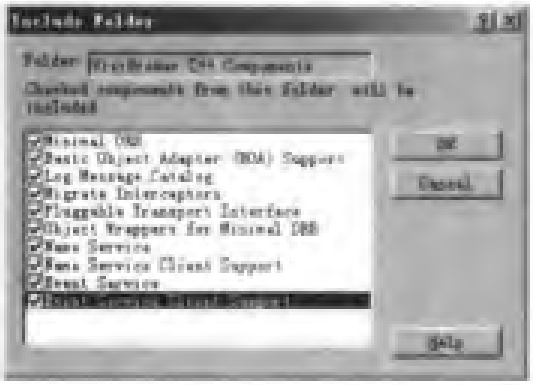


图 11-11　加入的 VisiBroker C++组件

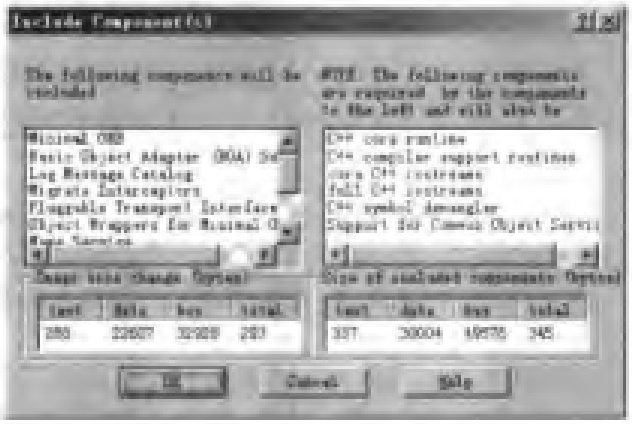


图 11-12　VxWorks 操作系统下加入 VisiBroker C++ CORBA 目标码

必要时，用户需要将相应的库文件加入到操作系统中。打开 Builds 标签页，选择 default 的 Properties，会弹出如下对话框，选择 Macros 标签页，设定宏 EXTRA_MODULES 的值为操作系统所用的 CORBA 库。这里仅加入了 osagent 相应的库，如图 11-13 所示。

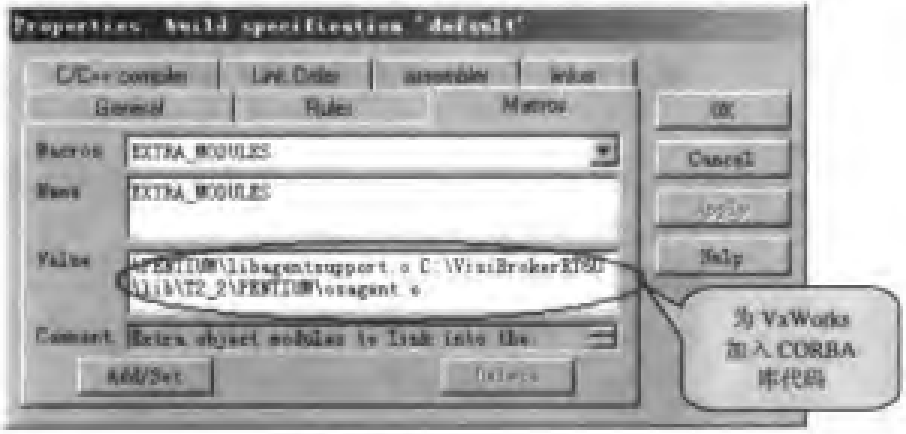


图 11-13　VxWorks 加入的 VisiBroker C++ CORBA 库代码

6. 产生 VxWorks 操作系统

我们以 osproj 为例，完成上述设定后，选择 Build->Rebuid All，产生操作系统。

7. 制作启动软盘的方法

产生引导文件。按 Build->Build Boot ROM…弹出启动文件的设置对话框，设置 BSP 为 pcPentium，然后设置映像文件为 bootrom_uncmp，工具为 gnu，最后按 OK 键，如图 11-14 所示。

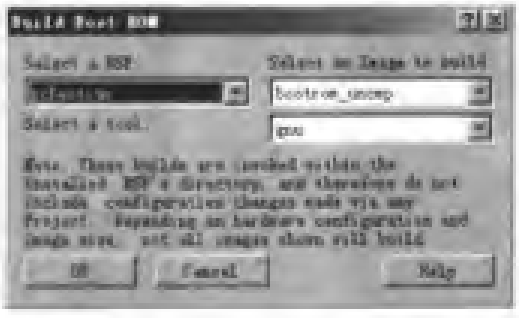


图 11-14　制作 bootrom 引导程序

复制 bootrom_uncmp 文件到\tornado2.2\host\X86-win32\bin 目录下，该目录下包括执行文件和批处理命令。

运行“mkboot a: bootrom_uncmp”将会在软盘产

生 bootrom.sys。

8. 配置文件传输协议（ftp）

Tornado2.2 有 FTP 服务器软件 WFTPD。使用了 FTP，目标机就可以通过网络完成从主机下载 VxWorks 操作系统映像文件。

运行程序->Tornado2.2->FTP Server，弹出 Tornado 的 FTP 设置环境。

选择 Security->User/rights...，弹出如图 11-15 所示的对话框。

按 New User...设置用户名及密码，并设置路径，按 Done 完成设置。单击 New User...，设定用户名为 wrs，密码为 wrswrs，同时将 Home Directory 设定为 VxWorks 操作系统所在的目录。

可以通过 Logging->Log Options...设定 FTP 文件传输协议的选项。

9. 配置 Target Server

在 Tornado2.2 开发环境中，选择菜单 Tools->Target Server->Configure…，弹出 Configure Target Servers 对话框，并设定参数如下：

```
Description: os
Target Server Properties: Back End, wdbrpc, Timeout(10sec), Re-try(10)
Target Name/IP Address: 193.0.0.100
Core File and Symbols: File"C:\zbs\os\default\vxWorks"
```

如图 11-16 所示。

图 11-15　开发机设置 ftp 选项

图 11-16　配置 Target Server

10. 创建用户工程

选择 File->New Project...，选择工程类型为 Create downloadable application modules for VxWorks，在设定完工程文件的名称路径后，第二步需设定工程文件的工具链为 PENTIUMVisiBrokerRT，如图 11-17 所示。

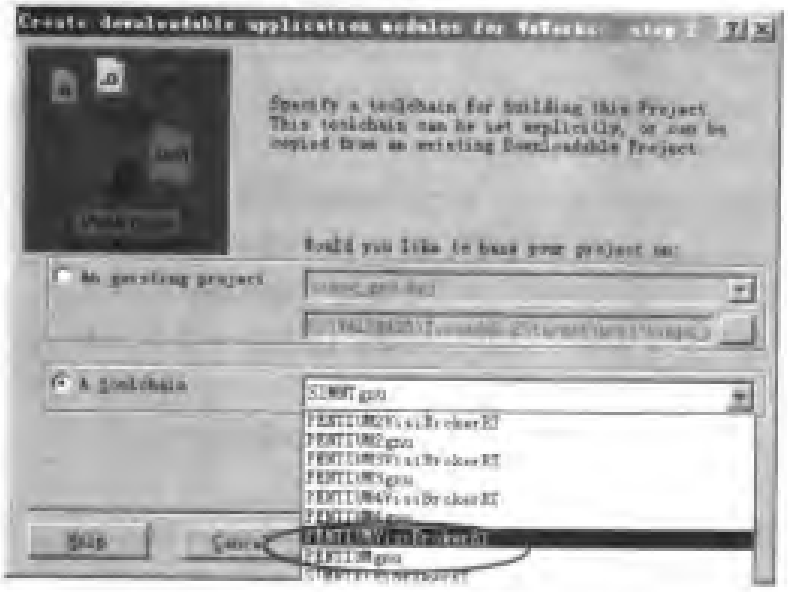


图 11-17　创建用户工程文件第二步

例如，欲创建一个 server 用户工程，其工具链应设定为 PENTIUMVisiBrokerRT，如图 11-18 所示。

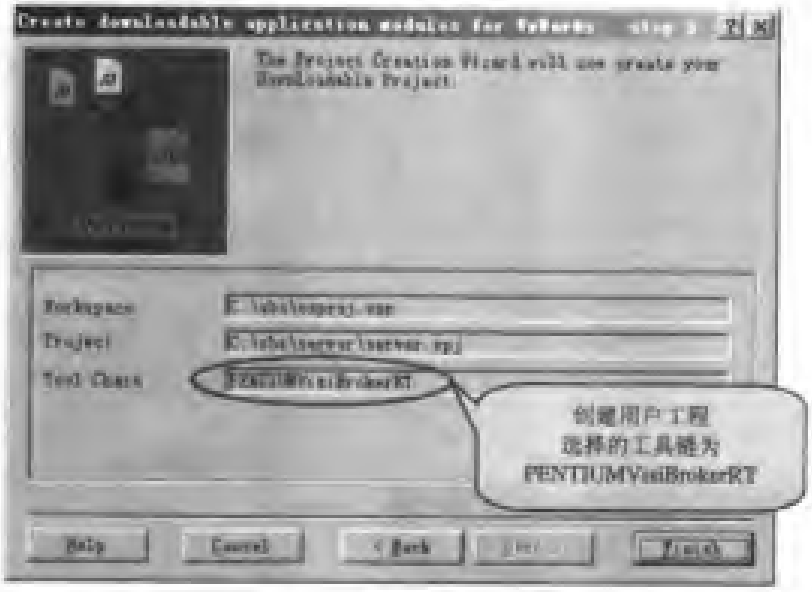


图 11-18　创建用户工程文件设定工具链为 PENTIUMVisiBrokerRT

11.4　VxWorks 操作系统下的 TAO 技术

TAO 是最著名的开源 CORBA 实现，由华盛顿大学的 DOC 小组研究开发，OCI 公司提供商业支持。DOC 小组不仅将 CORBA 标准以开源方式实现并发布，而且还做了许多关于协议和 DRE 中间件优化的研究工作。研究的重点集中在优化 TAO ORB 的性能以及可预言性（predictability）上，以期望满足端到端（end to end）应用时的服务质量（QoS）要求，主要通过集成中间件以及 OS I/O 子系统、通信协议以及网络接口来实现。TAO 经过精心地设计和优

化，达到了高效性、可预言性（predictability）以及可扩展性。TAO 支持 CORBA 实时扩展和安全规范，TAO 的实时 CORBA 模型集如图 11-19 所示。

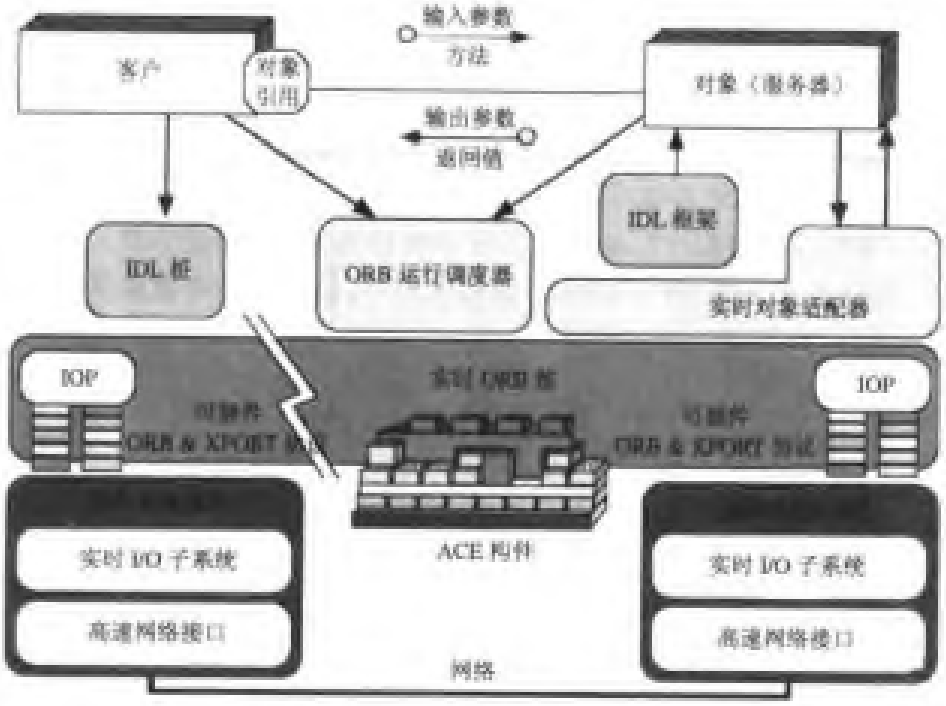


图 11-19　TAO 的实时 CORBA 模型集

第 12 章　嵌入式系统 CORBA 编程技术

本章基于 VisiBrokerRT C++，介绍嵌入式系统下的 CORBA 编程技术。首先通过一个简单的例子来说明如何在 VxWorks5.5 操作系统下建立一个基于 CORBA 对象的分布式应用程序的开发过程，以初步帮助读者掌握一个嵌入式实时系统 CORBA 编程的方法，然后介绍命名服务的 CORBA 例子，最后介绍一个实时 CORBA 编程的例子，以帮助读者更加清晰地了解 VxWorks 下的 CORBA 编程。

12.1　简　　介

我们建立一个关于 Bank 的构件，该构件由两个接口 Account 和 AccountManager 组成，Account 定义了一个 balance 操作，AccountManager 定义了一个 open 操作。应用程序包括两部分，即服务器程序和客户程序。服务器程序的功能是实现构件中定义的两个接口，并为该构件提供一个实现对象，能够为客户提供服务。

建立一个 CORBA 构件的过程如下：

（1）使用 IDL 语言描述对象的接口定义。

（2）使用 IDL 编译器产生框架代码。

（3）实现服务对象。

（4）初始化 ORB 和 POA。

（5）创建服务器对象。

（6）产生服务对象的引用。

（7）保存对象引用。

（8）获得服务对象的引用。

（9）异常处理。

（10）实现服务器程序。

（11）实现客户程序。

（12）编译和链接应用程序。

（13）运行服务器。

（14）运行客户程序。

该例子包含的文件如表 12-1 所示。

表 12-1　例子 1 所包括的文件

文　　件	说　　明
Bank.idl	使用接口定义语言（IDL）定义的 Bank 对象公共接口
BankImpl.h	Bank 对象实现类的声明头文件
BankImpl.cpp	Bank 对象实现类的实现源程序

续表

文　件	说　明
Client.c	客户方主程序，调用 Bank 对象中的操作
Server.c	服务器方主程序，启动 Bank 服务对象，为客户方提供服务
Corba_init.c	客户方与服务器方的 CORBA 初始化程序
Makefile	Makefile 文件

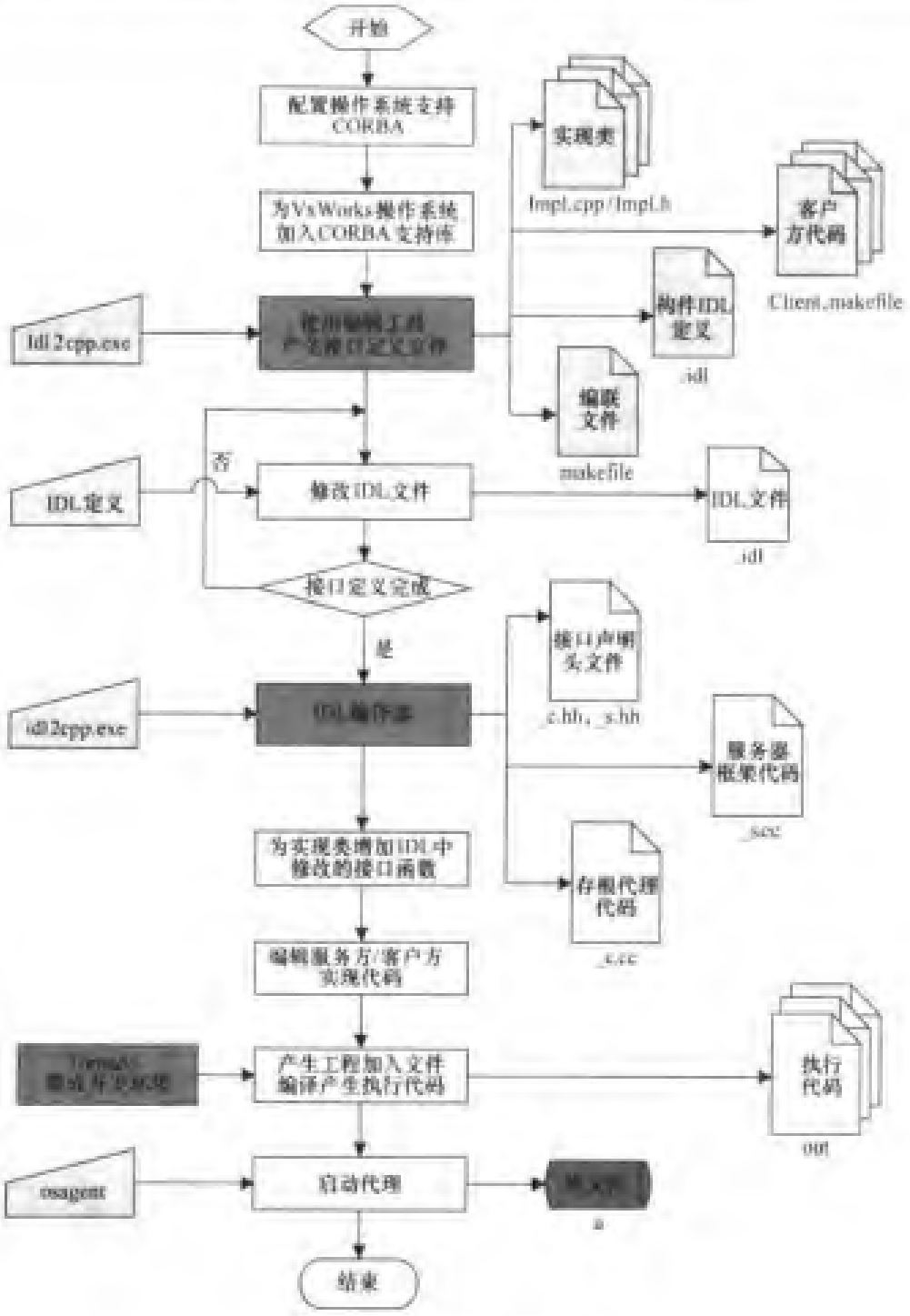


图 12-1　CORBA 开发过程图

1. 使用接口定义语言（IDL）定义接口

IDL 的语法与 C++极为相似，但需要注意的是它并不是一种编程语言，IDL 编译器能够将接口定义语言映射到不同的编程语言。一般情况下支持 IDL 语言到 C、C++和 JAVA 语言的映射，这部分工作由 IDL 编译器完成。定义接口可以由一个 IDL 文件实现（其文件名后缀为.idl）。以下代码定义了 Bank 接口的具体定义，包括 Account 接口，它定义了 balance 方法；AccountManager 接口，它定义了 open 方法。代码在文件 bank.idl 中：

```
// Bank.idl
module Bank {
  interface Account {
    float balance();
  };
  interface AccountManager {
    Account open(in string name);
  };
};
```

2. 使用 IDL 编译器产生客户方与服务器框架代码

创建 IDL 文件后，可以使用 IDL 编译器产生所需要针对具体编译环境的编程语言的框架代码，由于 VxWorks 支持 C、C++代码，这里需要使用 IDL 编译器产生 C++框架代码，产生的代码有两部分：客户方框架代码（也称为存根（stub）代码）和服务器框架代码（也称为代理代码、框架（skeleton）代码）。VisiBroker 的 C++的 IDL 编译器为 idl2cpp，可以使用如下命令编译 bank.idl 文件：

```
idl2cpp bank.idl
```

IDL 编译器 idl2cpp 编译 bank.idl 文件后，将产生以下四个文件：

（1）bank_c.hh——客户方框架代码类的声明。

（2）bank_c.cc——客户方框架代码类的实现，即 stub 代码。

（3）bank_s.hh——服务器框架代码类的声明。

（4）bank_s.cc——服务器框架代码类的实现，即 skeleton 代码。

使用后缀.hh 和.cc 分别表示 C/C++头文件和源程序，以区别于.cpp 和.b 文件，用户也可以使用 idl2cpp 的参数来设定其输出文件的后缀，格式如下所示：

```
idl2cpp [参数] IDL文件
```

idl2cpp 的参数说明如表 12-2 所示。

表 12-2 idl2cpp 的参数说明

参　数	含　义
-D, -define foo[=bar]	预处理宏定义，值可选
-I, -include <dir>	指定#include 搜索时的路径
-P, -no_line_directives	不从预定义中删除#line directives（默认为 off）
-H, -list_includes	显示 IDL 编译器编译时遇到所包括的#included 文件名（默认 off）
-C, -retain_comments	保留预处理后输出的注释（默认 off）
-U, -undefine foo	取消预处理宏定义

续表

参　数	含　义
-[no_]idl_strict	严格按照 OMG 标准编译 IDL 源文件（默认 off）
-[no_]builtin (TypeCode|Principal)	创建内置类型"::TypeCode"或者"::Principal"（默认 on）
-[no_]warn_unrecognized_pragmas	#pragma 宏不识别时告警（默认 on）
-[no_]back_compat_mapping	使用与 Visibroker 3.x 兼容的映射模式（默认 off）
-[no_]preprocess	Preprocess the input file before parsing（默认 on）
-[no_]preprocess_only	Stop parsing the input file after preprocessing（默认 off）
-[no_]warn_all	同时打开/关闭全部警告（默认 off）
-[no_]boa	产生 BOA 兼容代码（默认 off）
-[no_]comments	产生代码中存在注释（默认 on）
-gen_included_files	产生#include 文件代码（默认 off）
-list_files	代码产生时输出列表文件（默认 off）
-[no_]obj_wrapper	Generate support for object wrappers（默认 off）
-root_dir <path>	Directory in which generated files should reside
-[no_]servant	产生服务器方代码（默认 on）
-[no_]tie	产生 tie 类（默认 on）
-[no_]warn_missing_define	前向声明名称从未定义时输出告警信息（默认 on）
-client_ext <string>	客户端文件扩展名（_c）
-server_ext <string>	服务器端文件扩展名（_s）
-corba_inc <file>	系统包括指定文件#include（"corba.h"）
-excep_spec	产生异常说明（throw 列表）（off）
-export <tag>	类输出标志
-export_skel <tag>	框架代码中的类输出标志
-hdr_suffix <string>	头文件名后缀（.hh）
-src_suffix <string>	源程序文件名后缀（.cc）
-namespace <ns>	为所产生的代码指定根命名空间
-impl_base_object <C++ type>	指定执行类基类（"CORBA::Object"）
-[no_]pretty_print	产生_pretty_print 方法（默认 on）
-[no_]stdstream	产生输入输出流操作（默认 on）
-target <compiler>	产生<compiler>（"solaris"）目标代码
-corba_style_tie	需要"-tie"标志。在框架代码中产生 tie 类（默认 off）
-type_code_info	产生类型代码信息（默认 off）
-corba_style	需要"-type_code_info"标志。为 CORBA::Any 类产生指针插入/删除操作（默认 off）
-impl_inherit	产生执行类继承（默认 off）
-map_keyword <kwd> <replacement>	指定关键词及其等价类
-h, -help, -usage, -?	打印使用帮助信息
-version	显示软件版本号

需要特别注意的是，通常情况下这些产生的文件不需要用户进行编辑和修改。如果接口发生了改变，即 idl 文件发生更改，则需要重新编译 idl 文件产生新的输出。

3. 初始化 ORB

ORB 的初始化通过 CORBA::ORB_init 完成，ORB_init 参数可选如下两种方式：

（1）指定参数进行初始化，例：start_corba("-ORBagentport 19000")。

（2）使用默认方式进行初始化。

初始化代码如下：

```
#include <vxWorks.h>
#include "corba.h"
#include <taskLib.h>
#include "vutil.h"

#define OSAGENT_PORT "14000"

/* 用户函数说明 */
extern "C" void start_corba(char * ORB_options_string);
static void do_corba(char * ORB_options_string);

CORBA::ORB_var orb;                    /* 全局变量,由ORB_init初如化得到 */

/* 产生的VxWorks 任务,完成ORB 必要的初始化工作*/
void start_corba(char *opts)
{
    taskSpawn("CORBA_INIT",100,VX_FP_TASK,20000,(FUNCPTR)do_corba,(int)opts,
0,0,0,0,0,0,0,0,0,0);
}

/*函数:do_corba 功能:主要完成ORB 初始化，调用ORB_init*/
void do_corba(char * ORB_options)
{
   int default_argc = 2;
   char  *default_argv[] = {"-ORBagentport", OSAGENT_PORT};
   char **new_argv;
   int new_argc = VISUtil::stringToArgv(&new_argv, default_argv, default_argc,
ORB_options);

   VISTRY                              /* 调用 ORB_init */
   {
      orb = CORBA::ORB_init(new_argc, new_argv);          // 初始化ORB
      VISUtil::freeArgv(new_argc, new_argv);
   }
   VISCATCH(CORBA::Exception, e)
   {
      cerr << e << endl;
      taskSuspend(0);
   }
   VISEND_CATCH
   return;
}
```

4. 实现服务器程序

服务器方执行代码需要派生 AccountPOA 和 AccountManagerPOA 类，完成对象方法的实现代码。

AccountRegistry 类主要功能是存储客户名称和储蓄金额的信息，能够进行对指定客户的存取操作。

AccountImpl 类的主要功能为帐户分配金额，并执行 balance()操作，该操作得到指定客户的账户余额信息。

AccountManagerImpl 类则主要用于的得到和设定帐户信息，并执行 open()操作，该操作为新客户开一个银行账户，并设定其初始的存款余额。

AccountImpl 和 AccountManagerImpl 类派生于 Bank_POA 类，其关系如图 12-2 所示。

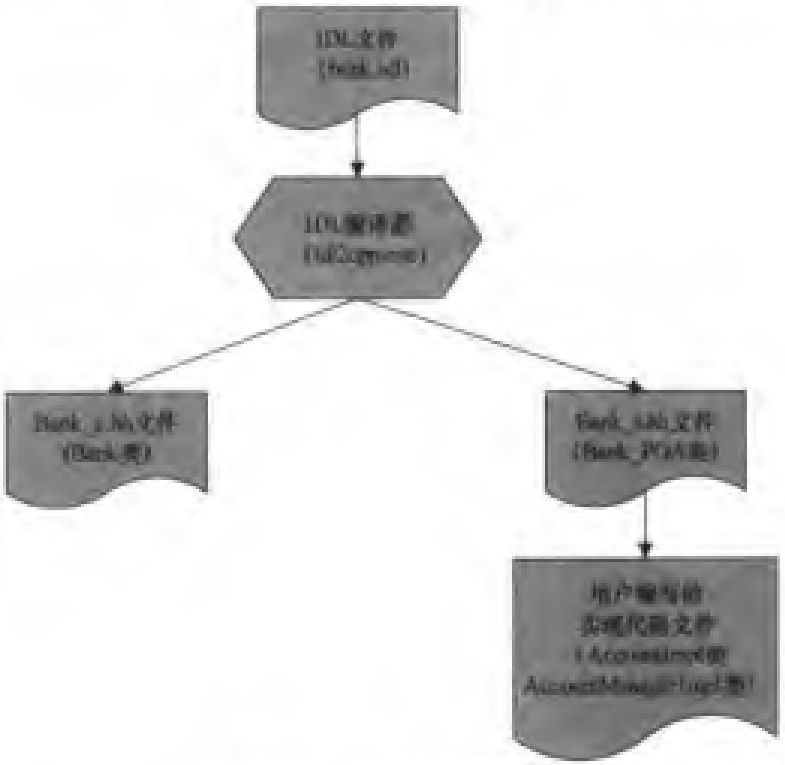


图 12-2　用户对象类关系

5. AccountRegistry 类

```
#define _MAX_SIZE  256
#define _TYPE_SIZE 32
// The AccountRegistry is a holder of Bank account implementations
class AccountRegistry
{
 public:
   AccountRegistry() : _count(0), _max(16), _data((Data*)NULL)
   {
     _data = new Data[16];
   }
   ~AccountRegistry() { delete[] _data; }
   void put(const char* name, PortableServer::ServantBase_ptr servant) {
```

```
    VISMutex_var lock(_lock);
    if (_count + 1 == _max) {
      Data* oldData = _data;
      _max += 16;
      _data = new Data[_max];
      for (CORBA::ULong i = 0; i < _count; i++)
        _data[i] = oldData[i];
      delete[] oldData;
    }
        _data[_count].name = name;
    servant->_add_ref();
    _data[_count].account   = servant;
    _count++;
  }
  PortableServer::ServantBase_ptr get(const char* name) {
    VISMutex_var lock(_lock);
    for (CORBA::ULong i = 0; i < _count; i++) {
      if (strcmp(name, _data[i].name) == 0) {
        _data[i].account->_add_ref();
        return _data[i].account;
      }
    }
    return PortableServer::ServantBase::_nil();
  }
 private:
  struct Data {
    CORBA::String_var          name;
    PortableServer::ServantBase_var  account;
  };

  CORBA::ULong _count;
  CORBA::ULong _max;
  Data*        _data;
  VISMutex   _lock;            // Lock for synchronization
};
```

6. AccountImpl 类

AccountImpl 类派生于 POA_Bank 和 PortableServer::RefCountServantBase，其代码如下：

```
class AccountImpl : public virtual POA_Bank::Account,
                    public virtual PortableServer::RefCountServantBase
{
 public:
  AccountImpl(CORBA::Float balance) : _balance(balance)
  {}

  CORBA::Float balance() { return _balance; }

 private:
  CORBA::Float _balance;
};
```

7. AccountManagerImpl 类

AccountManagerImpl 类派生于 POA_Bank 和 PortableServer::RefCountServantBase，其代码如下：

```
class AccountManagerImpl: public POA_Bank::AccountManager,
                          public virtual PortableServer::RefCountServantBase
{
  public:
    AccountManagerImpl() {}
      Bank::Account_ptr open(const char* name) {
      // Lookup the account in the account dictionary.
      PortableServer::ServantBase_var servant = _accounts.get(name);
       if (servant == PortableServer::ServantBase::_nil()) {
        // Seed the random number generator
        srand((unsigned)tickGet());
        // Make up the account's balance, between 0 and 1000 dollars.
        CORBA::Float balance = abs(rand()) % 100000 / 100.0;
        // Create the account implementation, given the balance.
        servant = new AccountImpl(balance);
        // Print out the new account
        cout << "Created " << name << "'s account." << endl;
        // Save the account in the account dictionary.
        _accounts.put(name, servant);
      }
      VISTRY {
        // Activate it on the default POA which is root POA for this servant
        PortableServer::POA_var default_poa = _default_POA();
        CORBA::Object_var ref;
        VISIFNOT_EXCEP
          ref = default_poa->servant_to_reference(servant);
        VISEND_IFNOT_EXCEP
        Bank::Account_var account;
        VISIFNOT_EXCEP
          account = Bank::Account::_narrow(ref);
        VISEND_IFNOT_EXCEP
        VISIFNOT_EXCEP
          // Print out the new account
          cout << "Returning " << name << "'s account: " << account << endl;
          // Return the account
          return Bank::Account::_duplicate(account);
        VISEND_IFNOT_EXCEP
      }
      VISCATCH(CORBA::Exception, e) {
        cerr << "_narrow caught exception: " << e << endl;
        taskSuspend(0);
      }
      VISEND_CATCH
      return Bank::Account::_nil();
    }
```

```
    private:
    static AccountRegistry _accounts;
};
```

服务器方需要经过如下步骤才能完成服务器方的启动过程。

(1) 得到 root POA 的引用。

(2) 得到 POA Manager。

(3) 设置策略并使用策略创建 POA。

(4) 创建 AccountManager 服务器对象。

(5) 使用 POA 激活服务器。

(6) 激活 POA Manager。

(7) 得到对象引用。

(8) 使用 CORBA::ORB::object_to_string()将创建字符对象。

(9) 等待客户请求。

(10) 异常处理。

服务器方代码如下:

```
//bank_account 服务器方程序
#include <vxWorks.h>
#include "corba.h"
#include "bankImpl.h"

/* 用户函数说明 */
extern "C" void start_bank_server(void);
static void bank_server(void);

extern CORBA::ORB_var        orb;          /* 全局变量,由 ORB_init 初始化得到 */
AccountRegistry AccountManagerImpl::_accounts;     // 声明全局对象

void start_bank_server(void)
{
  taskSpawn("BANK_SRVR",100,VX_FP_TASK,20000,(FUNCPTR)bank_server,0,0,0,0,
0,0,0,0,0,0);
}

void bank_server()
{
  PortableServer::POA_var rootPOA;

  VISTRY {
     //得到根 POA 的引用
     CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
     rootPOA = PortableServer::POA::_narrow(obj);

     // 得到 POA Manager
     PortableServer::POAManager_var poa_manager = rootPOA->the_POAManager();

     //设置策略
```

```
        CORBA::PolicyList policies;
        policies.length(1);
                policies[(CORBA::ULong)0]=rootPOA->create_lifespan_policy
(PortableServer::PERSISTENT);
        //使用策略创建用户 POA
        PortableServer::POA_var myPOA;
        myPOA = rootPOA->create_POA("bank_account_poa", poa_manager, policies);

        // 创建用户对象
        AccountManagerImpl *managerServant = new AccountManagerImpl;

        // 创建用户对象 ID
        PortableServer::ObjectId_var managerId;
        managerId = PortableServer::string_to_ObjectId("BankManager");
        // 在用户 POA 上使用对象 ID 激活服务器对象
        myPOA->activate_object_with_id((CORBA_OctetSequence        &)managerId,
managerServant);
        poa_manager->activate();                    // 激活 POA Manager

         //得到对象 IOR,并输出到用户终端
         CORBA::Object_var ref = myPOA->servant_to_reference(managerServant);
         CORBA::String_var string_ref = orb->object_to_string(ref.in());
         cout<<endl<<"CORBA  Object  ==>  "<<endl<<ref<<endl<<string_ref<<"  is
ready"<<endl ;

         orb->run();
      }
      VISCATCH(CORBA::Exception, e) {
        cerr << e << endl;
        taskSuspend(0);
      }
      VISEND_CATCH
      return;
    }
```

8. 实现客户程序

客户方需要初始化 VisiBroker 的 ORB，绑定 Account 和 AccountManager 对象，调用对象方法，并打印对象方法所返回的值。

（1）通过 OSAgent（例如使用_bind()方法）定位 CORBA 对象。

（2）调用 CORBA 对象的方法。

（3）使用 CORBA::ORB::string_to_object()方法从字符串创建对象引用。

以下为客户方代码，在完成 ORB 初始化后，绑定 AccountManager 对象，调用 open()方法，然后调用 Account 对象引用的 balance()方法。

客户通过 OSAgent，使用_bind 方法（默认）定位服务器对象，也可以使用服务器方字符串的 IOR。如果使用 IOR，将服务器方 IOR 字符串代替空的字符串即可，该字符串只要服务器方激活后将显示到服务器所处的显示终端。

客户方代码如下：

```
    //bank_account 客户端程序
    #include <vxWorks.h>
    #include "corba.h"
    #include <vport.h>
    #include "bank_c.hh"

    /* 用户函数说明 */
    extern "C" void start_bank_client(const char* name);
    static void bank_client(const char* name);

    extern CORBA::ORB_var orb;                    /* 全局变量,由 ORB_init 初如化得到 */

    void start_bank_client(const char* name)
    {
    taskSpawn("BANK_CLNT",100,VX_FP_TASK,20000,(FUNCPTR)bank_client,(int)name,0,
0,0,0,0,0,0,0,0);
    }

    void bank_client(const char* name)
    {
      VISTRY {
        Bank::AccountManager_var manager;    //使用 POA 名字和服务器 ID 定位对象
       PortableServer::ObjectId_var managerId=PortableServer::string_to_ObjectId
("BankManager");        manager=Bank::AccountManager::_bind("/bank_account_poa",
(CORBA_OctetSequence &)managerId);

        Bank::Account_var account;
        if (name==NULL) {
          name = "JARI ZBS";
        }
        account = manager->open(name);

        // 调用接口
        CORBA::Float balance;
        balance = account->balance();
        cout << "The balance in " << name << "'s account is $" << balance << endl;
      }
      VISCATCH(CORBA::Exception, e) {
        cerr << e << endl;
      }
      VISEND_CATCH

      return;
    }
```

绑定用户对象：在客户程序调用 open()成员函数前，首先必须使用 bind()函数建立与服务器的连接。执行 bind()成员函数自动由 idl2cpp 完成，该函数使用 VisiBroker ORB 定位对象，建立与服务器之间的连接。如果服务器成功建立连接，就创建代理对象，向客户方返回对象指针。

9. 编译程序

（1）makefile 文件。

```
include $(VBROKERDIR)/examples/stdmk

EXE = corba_init server client

exe: $(EXE)

all: $(EXE)

clean:
    $(RM) *.hh *.cc core *.o $(EXE)

#
# "bank_account" specific make rules
#

bank_c.cc: bank.idl
    $(ORBCC) bank.idl

bank_s.cc: bank.idl
    $(ORBCC) bank.idl

corba_init: bank_c.o bank_s.o corba_init.o

client: client.o

server: server.o
```

其中 stdmk 文件定义：

```
## Multi-threaded vxWorks Makefile definitions

## Default TOOL to gnu
ifeq ($(TOOL),)
  TOOL=gnu
endif

## include VxWorks rules
include $(WIND_BASE)/target/h/make/defs.bsp
include $(WIND_BASE)/target/h/make/make.$(CPU)$(TOOL)
include $(WIND_BASE)/target/h/make/defs.$(WIND_HOST_TYPE)

ifeq ($(CPU),SIMNT)
DEBUG =
else
DEBUG = -g
endif
STDCC_LIBS =
```

```
## VisiBroker IDL compiler location in installation
ORBCC      = $(VBROKERDIR)/bin/idl2cpp

## header file location
CCINCLUDES = -I. -I$(VBROKERDIR)/include

# VBCPP_BASE declared for backward compatibility with 3.x
ifeq ($(VBROKERDIR),)
  VBROKERDIR = $(VBCPP_BASE)
endif

############################################################
#    compile switches for Tornado
############################################################
#

ifeq ($(ORB_TYPE),)
  ORB_TYPE = min
endif

#tln--issue203                  -D_VIS_NO_SRVRMGR -D_VIS_NO_ORBMGR \
# VisiBroker Tuneflags common to both full and min ORB
COMMON_TUNEFLAGS  = -D_VIS_NO_LOADLIB -D_VIS_LONG_LONG -DRT_CORBA \
                    -D_VIS_NOEXCEPTIONS -D_VIS_NO_TRANSACTIONS \
                    -D_VIS_FULL_POA -DTHREAD

# VisiBroker Tuneflags just for ORB
FULL_TUNEFLAGS = -D_VIS_FULL_ORB

# VisiBroker Tuneflags just for min ORB
MIN_TUNEFLAGS = -D_VIS_NO_DII -D_VIS_NO_DSI -D_VIS_NO_IR -D_VIS_NO_DYNANY

# Setup final option variables
ifeq ($(ORB_TYPE), dyn)
    TUNEFLAGS = $(COMMON_TUNEFLAGS) $(FULL_TUNEFLAGS)
    EXCEPT_FLAGS = -fno-exceptions
endif
ifeq ($(ORB_TYPE), min)
    TUNEFLAGS = $(COMMON_TUNEFLAGS) $(MIN_TUNEFLAGS) -D_VIS_NOEXCEPTIONS
    ORB_TYPE_FLAGS = -DORB_TYPE=min
    EXCEPT_FLAGS = -fno-exceptions
endif


## Compiler flags for VxWorks
COMMONDEFS =  -x c++ -DVXWORKS -Dgcc272 -fno-builtin -fno-inline \
            $(TUNEFLAGS) $(ORB_TYPE_FLAGS) $(EXCEPT_FLAGS) $(TOR_VER_FLAGS)

ifeq (SIMNT,$(CPU))
ADDED_C++FLAGS    = $(CCINCLUDES) $(DEBUG) $(COMMONDEFS) -O0
```

```
else
ADDED_C++FLAGS    = $(CCINCLUDES) $(DEBUG) $(COMMONDEFS)
endif

FLAGS_WARNINGS_REMOVE = -Wall
FLAGS_STRIPPED = $(filer-out $(FLAGS_WARNINGS_REMOVE),$(C++FLAGS))
CFLAGS := $(FLAGS_STRIPPED)

.SUFFIXES: .C .o .h .hh .cc

#    $(CC) -traditional $(CFLAGS) -c ctdt.c

.o:
    $(LD) $(LD_PARTIAL_FLAGS) -o $@.tmp  $^
    $(NM) $@.tmp | $(MUNCH) > ctdt.c
ifeq ($(CPU), MIPS64)
    $(CC) -ansi -mips3 $(CFLAGS) -c ctdt.c
else
    $(CC) -ansi $(CFLAGS) -c ctdt.c
endif
    $(LD) $(LD_PARTIAL_FLAGS) -o $@ $@.tmp ctdt.o
    @ $(RM) *.tmp ctdt.*

.C.o:
    $(CC) $(C++FLAGS) -c -o $@ $<

.c.o:
    $(CC) $(C++FLAGS) -c -o $@ $<

.cc.o:
    $(CC) $(C++FLAGS) -c -o $@ $<
```

（2）使用工程进行编译和链接。

按以下步骤建立工程文件：

第一步：创建工程。

在 Tornado2.2 中打开 File->New Project 菜单，选择一个 downloadable 应用。设定服务器方工程名为 server，客户方工程名为 client，注意在选择工程的工具链是须设定为 PENTIUMVisiBrokerRT。

第二步：为工程加入文件。

将 IDL 编译器产生的源文件、头文件、IDL 文件以及服务器或者客户方代码分别加入相对应的工程文件中。在工作空间管理窗口中的项目节点上单击鼠标右键，选择 Add Files 菜单，将 IDL 文件 bank.idl、编译 IDL 文件所产生的客户方代码 bank_c.cc 和 bank_c.hh、ORB 初始化代码 corba_init.c 以及客户方代码 client.c 加入到 client 工程文件中，将 IDL 文件 bank.idl、编译 IDL 文件所产生的客户方代码 bank_c.cc 和 bank_c.hh、编译 IDL 文件所产生的服务器代码 bank_s.cc 和 bank_s.hh、ORB 初始化代码 corba_init.c 以及服务方代码 server.c 加入到 server 工程文件中，如图 12-3 所示。

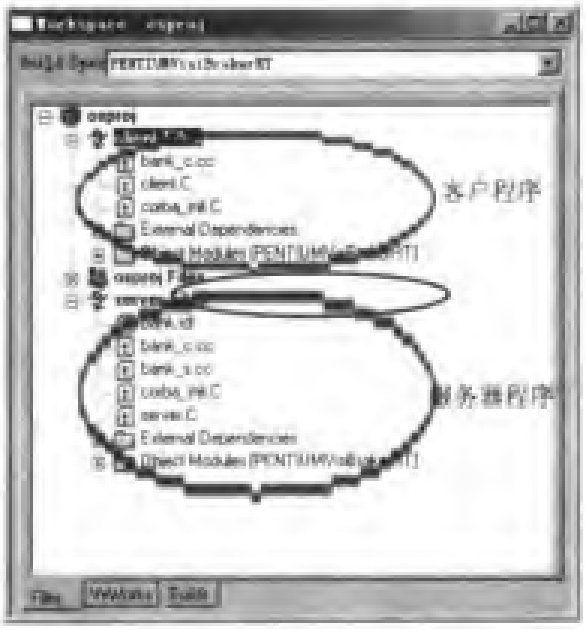


图 12-3　向工程添加文件示意图

第三步：完成编译和链接。

在 Files 标签页，选择用户的工程项目，点击鼠标右键，选择 Dependency，处理文件之间的依赖性。选择 Rebuild All，开始工程文件的编译和链接。

10. 启动 VisiBroker 的智能代理（OSAgent）

（1）Windows 或者 Unix 开发环境。

首先应保证 Smart Agent 的 PATH 环境变量已经成功更新到 VisiBroker for C++的 Tornado 的"bin"目录中。

Windows 平台，在 DOS 提示符下输入如下命令：

```
prompt -> osagent
```

UNIX 平台，输入如下命令：

```
prompt -> osagent &
```

（2）VxWorks 开发节点。

在 VxWorks 节点上启动 VisiBroker 智能代理（OSagent），应保证 osagent 库（osagent.o）可用。加入 osagent.o 的方法有两种：

1）将 osagent.o 与 VxWorks 操作系统链接到一起。

2）将 osagent_munched.o 下载到 VxWorks 操作系统中。

在 Tornado WindShell 启动智能代理：

```
--> startOsagent()
```

（3）加载 ORB。

对应该例子的 CORBA 系统库和 IDL 编译产生的代码应首先加载到目标机上：

```
prompt-> ld < corba_init
```

（4）初始化 ORB。

可以采用如下两种方式初始化 ORB：

1）用默认方式初始化 ORB。

```
prompt-> start_corba
```

2）用指定的初始化选项初始化 ORB。例如，将 ORBtcpnodelay 选项打开方式初始化 ORB。

```
prompt-> start_corba ("-ORBtcpnodelay 1")
```

（5）加载并运行例程。

在运行例程前，应确保所有节点的 ORB 完成初始化工作。

1）运行服务器（节点 1）。

```
prompt-> ld < server
prompt->start_bank_server
```

其对应的输出为：

```
CORBA Object ==>
Repository ID: IDL:Bank/AccountManager:1.0
 Object name: NONE
IOR:010000001c00000049444c3a42616e6b2f4163636f756e744d616e616765723a312e300
0010000000000000
005d000000010102000c0000003139332e302e302e3130300000400002f00000001504d430
400000012000000
2f62616e6b5f6163636f756e745f706f610000000b00000042616e6b4d616e6167657200010
000000353495605
0000000104070102 is ready
```

2）运行客户端（节点 2 或者与节点 1 运行在同一机器上）。

```
prompt-> ld < client
prompt-> start_bank_client "john"
```

客户方	服务器
	Created jari's account. Returning jari's account: Repository ID: IDL:Bank/Account:1.0 Object name: NONE
The balance in jari's account is $19.79	

也可以采用如下方式：

```
prompt-> ld < client
prompt-> start_bank_client
```

（使用默认名字）

客户方	服务器
	Created JARI ZBS's account. Returning JARI ZBS's account: Repository ID: IDL:Bank/Account:1.0 Object name: NONE
The balance in JARI ZBS's account is $191.3	

12.2 命名服务例程

命名服务是 CORBA 中最基本的对象服务，它实现了逻辑名和对象应用之间的关联。服务器方应使用命名服务将对象引用注册到名字服务中，并向客户发布逻辑名。客户方在得到逻辑命名后，通过逻辑名到名字服务的映射解析出对应的对象引用，从而调用对象所支持的方法。本节描述了命名服务的基本概念和方法，以及如何实现编程。

命名服务将对象引用与名字相对应，实现“名字|引用”二元组的映射，称之为名字绑定(name binding)。同一个对象引用可以使用不同的命名，但是每个名字只能对应到唯一的一个对象引用。命名上下文（naming context）能够注册名字绑定的对象引用，实现了名字到对象引用的映射，命名上下文也是对象用于存放名字绑定。使用命名服务具有如下优点：

（1）赋值功能，实现对象与名字的对应。

（2）查找功能，可以通过名字查找对象。

（3）构造命名图以清晰表示对象之间的关系。

（4）在绑定名字与对象引用之间提供了一种兼容方式。

名字服务中，命名定义为序列类型，其序列为命名部件（NameComponent）。一个命名部件有两部分组成：标识属性 id 和类型属性 kind。VisiBroker 命名服务时可以使用可打印字符，其长度不能超过 255 字节，不包括 NULL 空字符。当比较两个命名部件是否等价时，需要判断这两个命名部件的 id 和 kind 是否完全一致，并且有大小写区分。

```
// CosNaming.idl
module CosNaming {
    typedef string Istring;
    struct NameComponent {
        Istring id;
        Istring kind;
    };
    typedef sequence<NameComponent> Name;
};
```

命名服务提供了如下的接口：

```
// CosNaming.idl
module CosNaming {
    ...
    interface NamingContext{...};
    interface BindingInterator{...};
    interface NamingContextExt{...};
};
```

12.2.1 命名上下文返回的异常

调用命名服务的操作可能会出现如下异常：

（1）NotFound 异常。

当某一操作需要查找一个逻辑名，而此逻辑名不能正确地被解析为一个现有的绑定时，

NotFound 包括两个数据项：why 和 rest_of_name。why 指明了 Name 或者命名部件的一部分无法找到。

（2）CannotProceed 异常。

该异常在无法执行所请求的操作时出现。

（3）InvalidName 异常。

指定 Name 为空命名部件或者空的 id 域，即它们是空的字符串时将产生 InvalidName 异常。

（4）AlreadyBound 异常。

表示一个对象已经绑定到指定的名字。在命名上下文中，一个对象只能绑定到一个特定的名字。

（5）NotEmpty 异常。

当 NamingContext 包括绑定时调用 destroy 操作时，所删除的命名上下文应确保为空，否则会产生 NotEmpty 异常。

异常的定义：

```
interface NamingContext {
    enum NotFoundReason { missing_node, not_context, not_object };
    exception NotFound {
        NotFoundReason  why;
        Name            rest_of_name;
    };

    exception CannotProceed {
        NamingContext   cxt;
        Name            rest_of_name;
    };

    exception InvalidName{};
    exception AlreadyBound {};
    exception NotEmpty{};
};
```

12.2.2　命名上下文的操作

接口 NamingContext 操作包括绑定和生命期管理，其 IDL 定义如下：

```
interface NamingContext {
    void bind(in Name n, in Object obj)raises(NotFound, CannotProceed,
InvalidName, AlreadyBound);
    void rebind(in Name n, in Object obj)raises(NotFound,CannotProceed,
InvalidName);
    void bind_context(in Name n, in NamingContext nc) raises(NotFound,
CannotProceed, InvalidName, AlreadyBound);
    void rebind_context(in Name n, in NamingContext nc) raises(NotFound,
CannotProceed, InvalidName);
    Object resolve (in Name n) raises(NotFound, CannotProceed, InvalidName);
    void unbind(in Name n) raises(NotFound, CannotProceed, InvalidName);
    NamingContext new_context();
```

```
        NamingContext bind_new_context(in Name n) raises(NotFound, AlreadyBound,
CannotProceed, InvalidName);
        void destroy() raises(NotEmpty);
        void list(in unsigned long how_many,out BindingList bl,out BindingIterator
bi);
    };
```

（1）方法 bind 实现了创建绑定操作。在命名上下文中创建对象绑定，该方法试图将指定对象绑定到指定命名，使用第一个 NameComponent 类型的参数解析命名上下文，然后绑定对象到新的命名上下文中。bind 操作的参数 Name 用来指明对象所绑定的名字，参数 obj 则为所命名的对象。

（2）方法 rebind 的功能同 bind，不同的是 rebind 不会出现 AlreadyBound 异常。如果指定的命名已经绑定到另一对象，该绑定将被新的绑定所替代。

（3）命名上下文对象的绑定使用 bind_context()，该方法与 bind 方法类似，不同的是其输入的名字与 NamingContext 相关联，而 bind()方法则是一个任意的 CORBA 对象。

（4）rebind_context 方法与 bind_context 方法相一致，不同的是产生的异常中没有 AlreadyBound 异常。如果指定名字已绑定到一个命名上下文，该绑定将被新的绑定所替代。

（5）方法 resolve 在给定的上下文中执行查找绑定名字对象的操作，给定的名字必须与绑定的名字相一致，命名服务不会返回对象类型，客户方必须使用 narrow 将相应的对象转化为用户所需要的类型。

（6）方法 unbind 从命名上下文中删除命名绑定，它执行的操作与 bind 方法正好相反。

（7）方法 new_context 创建并返回一个新的命名上下文，但并未绑定。

（8）方法 bind_new_context 创建一个新的命名上下文，并绑定到指定的名字。

（9）方法 destroy 用于删除命名上下文，如果命名上下文包括有相关的绑定将会产生 NotEmpty 异常，否则删除以后，任何在该命名上下文上执行该操作会产生 OBJECT_NOT_EXIST 运行错误。

（10）方法 list 返回命名上下文中所有的绑定，允许客户迭代命名上下文中的绑定。该方法返回给定数量绑定，其值在 BindingList，如果命名上下文中包括更多的绑定，则该方法返回一个 BindingIterator 以包括多余的绑定。

12.2.3 命名迭代接口

命名迭代接口的 IDL 定义如下：

```
module CosNaming {
    ...
    enum BindingType { nobject, ncontext };

    struct Binding {
        Name        binding_name;
        BindingType binding_type;
    };
    typedef sequence <Binding> BindingList;

    interface BindingIterator {
```

```
        boolean next_one(out Binding b);
        boolean next_n(in unsigned long how_many, out BindingList bl);
        void    destroy();
      };
    ...
    };
```

Binding、BindingList 和 BindingIterator 接口用于描述名字－对象之间的绑定。

Binding 结构封装了两个变量，binding_name 字段表示名字（Name）；字段 binding_type 表示名字是绑定到 ORB 对象，还是绑定到 NamingContext 对象。当使用 bind_context、rebind_context 和 bind_new_context 方法时，binding_type 值为 ncontext，其他方法时 binding_type 值为 nobject。

BindingList 是 Binding 结构的序列，该序列包括命名上下文（NamingContext）对象。

BindingIterator 接口允许客户方迭代绑定，其使用的方法是 next_one 或者 next_n。方法 next_one 遍历列集中的下一个绑定，如果输出的是一个有效的绑定，则其返回值为 true，否则没有绑定时返回 false。方法 next_n 返回 BindingList 变量，其数量为用户所需的列表中的绑定对象。next_n 返回的参数是调用 next_n 或者 next_one 尚未遍历的的列表，如果返回列表的长度不为 0 则返回 true，返回 false 则表示没有绑定或者列表的长度为 0。方法 destroy 用于删除迭代器，删除后用户再调用命名迭代器的任何操作，该操作均将返回一个 OBJECT_NOT_EXIST 异常。

12.2.4　命名上下文扩展接口

命名上下文扩展 NamingContextExt 接口，派生于命名上下文 NamingContext，提供了关于使用 URL 和字符串命名所需的操作，主要功能是实现字符串形式的名字到命名部件之间的转换。其 IDL 定义：

```
module CosNaming {
  interface NamingContextExt:NamingContext{
    typedef string StringName;
    typedef string Address;
    typedef string URLString;
    StringName to_string(in Name n) raises(InvalidName);
    Name to_name(in StringName sn)  raises(InvalidName);
    exception InvalidAddress {};
    URLString to_url(in Address addr, in StringName sn)raises(InvalidAddress,
InvalidName);
    Object  resolve_str(in  StringName  n)  raises(NotFound,  CannotProceed,
InvalidName, AlreadyBound);
  };
};
to_name()
```

转换字符串形式的名字到命名部件。如果字符串形式的名字语法错误或者范围越界，会产生 InvalidName 异常。

```
to_string()
```

转换命名部件到字符串形式。如果参数 Name 无效时，会产生 InvalidName 异常。

```
resolve_str()
```

使用字符串形式的名字从命名上下文中解析出所对应的对象引用，常用方法的执行与 NamingContext::resolve 相同操作。

```
to_url()
```

该方法在给定 URL 组件的情况下，返回字符格式 URL，格式不对时会产生异常。地址为空时产生 InvalidAddress 异常，第二个参数为空时产生 InvalidName 异常。

12.2.5 编程实例

本节通过一个简单的例子来说明在 VxWorks 操作系统下，服务器与客户方如何通过命名服务进行编程。可以将服务器程序与与客户程序下载到不同目标机或者在同一目标机上运行程序，以观察运行结果。

该例子包含的文件如表 12-3 所示。

表 12-3 命名例程所包括的文件

文　件	说　明
Bank.idl	使用接口定义语言（IDL）定义的 Bank 对象公共接口
BankImpl.h	Bank 对象实现类的声明头文件
BankImpl.cpp	Bank 对象实现类的实现源程序
Client.c	客户方主程序，调用 Bank 对象中的操作
Server.c	服务器方主程序，启动 Bank 服务对象，为客户方提供服务
Corba_init.c	客户方与服务器方的 CORBA 初始化程序
Makefile	Makefile 文件

12.2.5.1 服务方程序

本节通过一个简单的例子来说明在 VxWorks 操作系统下，服务器与客户方如何通过命名服务进行编程。

```
#include <vxWorks.h>
#include "corba.h"
#include "CosNaming_c.hh"
#include "bankImpl.h"

// 外部函数声明
extern "C" void start_bank_server(void);
static void bank_server(void);
extern CORBA::ORB_var orb;
AccountRegistry AccountManagerImpl::_accounts;      // 静态变量
void start_bank_server(void)
{
taskSpawn("BANK_SRVR",100,VX_FP_TASK,20000,(FUNCPTR)bank_server,0,0,0,0,0,0
```

```
,0,0,0,0);
    }

    void bank_server()
    {
        PortableServer::POA_var     rootPOA;
        CosNaming::NamingContext_var rootContext;

        //得到 root POA 引用
        CORBA::Object_var obj;
        obj = orb->resolve_initial_references("RootPOA");
        rootPOA = PortableServer::POA::_narrow(obj);

        // 得到命名服务上下文的引用
        obj=CosNaming::NamingContext::_narrow(orb->resolve_initial_references
("NameService"));
        rootContext = CosNaming::NamingContext::_narrow(obj);

        // 显示命名服务上下文的字符 IOR
        CORBA::String_var str;
        str = orb->object_to_string(obj);
        cout << "NamingContext object IOR is \n" << str << endl;

        //设置 POA 策略
        CORBA::PolicyList policies;
        policies.length(1);
        policies[(CORBA::ULong)0] = rootPOA->create_lifespan_policy (Portable-
Server::PERSISTENT);

        //得到 POA Manager
        PortableServer::POAManager_var poa_manager;
        poa_manager = rootPOA->the_POAManager();

        //使用策略创建用户 POA
        PortableServer::POA_var myPOA;
        myPOA = rootPOA->create_POA("bank_account_poa", poa_manager, policies);

        //创建服务器
        AccountManagerImpl *managerServant = new AccountManagerImpl;

        //设置服务方对象 ID
        PortableServer::ObjectId_var managerId;
        managerId = PortableServer::string_to_ObjectId("BankManager");

        //使用创建的 ID 和 POA 激活服务器
        myPOA->activate_object_with_id((CORBA_OctetSequence          &)managerId,
managerServant);

        //激活 POA Manager
        poa_manager->activate();
```

```
        // 得到对象引用
        CORBA::Object_var reference;
        reference = myPOA->servant_to_reference(managerServant);

        //将对象引用与命名上下文中的名字进行绑定
        CosNaming::Name name;
        name.length(1);
        name[0].id = (const char *) "BankManager";
        name[0].kind = (const char *) "";
        rootContext->rebind(name, reference);

        cout << reference << " is ready" << endl;
        return;
    }
```

12.2.5.2 客户方代码

客户方代码说明了如何通过方法 resolve 得到对象引用，然后调用对象方法的过程。其代码如下：

```
    #include <vxWorks.h>
    #include "corba.h"
    #include "CosNaming_c.hh"
    #include "bank_c.hh"

    //外部声明
    extern "C" void start_bank_client(void);
    static void bank_client(void);

    extern CORBA::ORB_var orb;

    void start_bank_client(void)
    {
    taskSpawn("BANK_CLNT",100,VX_FP_TASK,20000,(FUNCPTR)bank_client,0,0,0,0,0,0
,0,0,0,0);
    }

    void bank_client(void)
    {
        CosNaming::NamingContext_var rootContext;
        //得到命名服务上下文的引用
        CORBA::Object_var obj;
        obj = CosNaming::NamingContext::_narrow(orb->resolve_initial_references
("NameService"));
        rootContext = CosNaming::NamingContext::_narrow(obj);

        //显示命名服务上下文的字符 IOR
        CORBA::String_var str;
        str = orb->object_to_string(obj);
```

```
        cout << "NamingContext object IOR is \n" << str << endl;

        //构造命名
        CosNaming::Name name;
        name.length(1);
        name[0].id = (const char *) "BankManager";
        name[0].kind = (const char *) "";

        //通过命名服务定位对象
        Bank::AccountManager_var manager;
        manager = Bank::AccountManager::_narrow(rootContext->resolve(name));

        const char* account_name = "JARI";
        Bank::Account_var account;
        //调用方法
        account = manager->open(account_name);

        CORBA::Float balance;
        balance = account->balance();
        cout<<"The balance in "<< account_name << "'s account is $"<< balance <<
endl;

        return;
    }
```

12.3　RT-CORBA 例　程

我们在第 11 章中已经介绍了 RT-CORBA 的相关知识，本节仅通过一个实例来讲解实时 CORBA 的应用，需要注意的是在实时 CORBA 编程中对相关的编程进行了简化。

该例子包含的文件如表 12-4 所示。

表 12-4　实时 CORBA 所包括的文件

文　件	说　　明
Test.idl	使用接口定义语言（IDL）定义的对象公共接口
Test_Client.c	客户方主程序，调用 Bank 对象中的操作
Test_Server.c	服务器方主程序，启动 Bank 服务对象，为客户方提供服务
Corba_init.c	客户方与服务器方的 CORBA 初始化程序
Makefile	Makefile 文件

IDL 定义：

```
Test.idl
// Model.idl

interface Test
{
```

```
        void sum(in double x,in double y,out double ret);
    };
```

12.3.1 服务器端程序

```
    //测试服务器代码
    #include <vxWorks.h>
    #include <math.h>
    #include <stdlib.h>

    #include "test_s.hh"

    #include "rtcorba.h"

    extern "C" void start_test_server(void);
    static void test_server(void);

    extern CORBA::ORB_var orb;

    void start_test_server(void)
    {
    taskSpawn("SERVER",100,VX_FP_TASK,20000,(FUNCPTR)test_server,0,0,0,0,0,0,0,
0,0,0);
    }

    class TestImpl : public POA_Test
    {
    public:
        TestImpl() {}

        void sum(::CORBA::Double _x, ::CORBA::Double _y,::CORBA::Double_out _ret)
        {
            _ret = _x + _y;
            printf("%f + %f = %f\n",_x,_y,_ret);
        }
    };

    void test_server(void)
    {
        PortableServer::POA_ptr poa;

        // 得到根 POA 的引用
        CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
        PortableServer::POA_var rootPOA = PortableServer::POA::_narrow(obj);

        //得到 RT ORB 引用
        CORBA::Object_var objref = orb->resolve_initial_references("RTORB");
        RTCORBA::RTORB_var rtorb = RTCORBA::RTORB::_narrow(objref);

        //创建线程池
```

```
        RTCORBA::ThreadpoolId test_id;
        test_id = rtorb->create_threadpool( 30000,      // 堆栈大小
                                            5,          // 静态线程数
                                            0,          // 动态线程数
                                            5,          // 默认实时 CORBA 优先级
                                            0, 0, 0 );

        PortableServer::POAManager_var poa_manager;
        // 创建线程池策略对象
        RTCORBA::ThreadpoolPolicy_ptr test_policy =rtorb->create_threadpool_policy
(test_id);

         // 为 POA 初始化设定服务器模型策略
         RTCORBA::PriorityModelPolicy_ptr server_model_policy;
         server_model_policy        =        rtorb->create_priority_model_policy
( RTCORBA::SERVER_DECLARED, 25 );

        // 为将创建的 POA 指定永久生命期策略属性
        CORBA::PolicyList policies;
        policies.length(2);
        policies[0] = rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);
        policies[1] = server_model_policy;

        // 创建用户 POA
        poa_manager = rootPOA->the_POAManager();

        // 使用线程池创建用户 POA
        poa = rootPOA->create_POA("POA", poa_manager, policies);

        //创建服务器对象
        TestImpl *test = new TestImpl;

        // 激活对象
        PortableServer::ObjectId_var
oid=PortableServer::string_to_ObjectId("Test");
        poa->activate_object_with_id((CORBA_OctetSequence &)oid, test);

        // 激活 POA
        poa_manager->activate();

        cout << "Test Object is ready." <<endl;
        return;
    }
```

12.3.2　客户方程序

```
    //测试客户端代码
    #include <vxWorks.h>
    #include <fstream.h>
```

```
#include "test_c.hh"

#include "poa_c.hh"
#include "rtcorba.h"

extern "C" void start_test_client(void);
static void test_client(void);

extern CORBA::ORB_var orb;                    /* 全局变量,由ORB_init初始化得到 */

CORBA::Double x=4.0,y=5.0,ret;

void start_test_client(void)
{
    taskSpawn("CLIENT",100,VX_FP_TASK,20000,(FUNCPTR)test_client,0,0,0,0,0,
0,0,0,0,0);
}

void test_client(void)
{
    Test_var test;

    PortableServer::ObjectId_var testId;
    CORBA::Object_var rtorb_obj = orb->resolve_initial_references("RTORB");
    RTCORBA::RTORB_var rtorb = RTCORBA::RTORB::_narrow(rtorb_obj);

    // 创建 PolicyManager
    CORBA::Object_var    pol_man_obj    =    orb->resolve_initial_references
("ORBPolicyManager");
    CORBA::PolicyManager_var policy_manager = CORBA::PolicyManager::_narrow
(pol_man_obj);
    // 绑定对象
    testId = PortableServer::string_to_ObjectId("Test");
    test = Test::_bind("/POA",(CORBA_OctetSequence &)testId);

    // 调用对象的方法
    test->sum(x,y,ret);
    x+=1.0;y+=1.0;

    return;
}
```

第 13 章　GIOP 网络协议剖析

13.1　概　　述

本章主要说明 GIOP（通用内部对象请求代理协议，General Inter-ORB Protocol）协议，主要用于 ORB 之间的可交互性，该协议将其映射到面向连接的网络传输协议。OMG 定义了一类特定 GIOP 映射，直接基于 TCP/IP 连接，称这为 IIOP（互联网内部对象请求代理协议，Internet Inter-ORB Protocol）。

GIOP 和 IIOP 以一种通用低成本的方式支持协议级的 ORB 的可交互性。

GIOP 规范包括下列元素：

（1）公用数据表示定义（CDR）。CDR 是一个传输的语法，用于将 OMG IDL 映射到低级别的表示，用于 ORB 和内部 ORB 桥之间的传输。

（2）GIOP 消息格式。GIOP 消息用于在对象之间交互信息。

（3）GIOP 传输假定。GIOP 规范定义了与网络传输层有关的通用假定以发送 GIOP 消息，另外，GIOP 规范还定义了如何管理连接和 GIOP 消息序列上的约束等。GIOP 是抽象的协议。而 IIOP 则实现了协议的映射。

IIOP 规范在 GIOP 规范的基础上还增加了如下元素：

互联网 IOP 消息传输。IIOP 规范描述如何打开 TCP/IP 连接，并传输 GIOP 消息。

IIOP 并不是一个独立的规范，它是一种映射规范，是将 GIOP 映射到特定的传输协议（TCP/IP）。换句话说，IIOP 实现了网络传输层之间的映射，它将应用传输控制协议（TCP）借助因特网的传输层来传递请求或者接收应答。也可以简单地认为：GIOP+TCP/IP=IIOP。当 IIOP 使用 TCP/IP 协议时，其端口号为 535。

IIOP1.0 基于 GIOP1.0；IIOP1.1 基于 GIOP1.0 或者 GIOP1.1；IIOP1.2 基于 GIOP1.0、GIOP1.1 或者 GIOP1.2；IIOP1.3 基于 GIOP1.0、GIOP1.1、GIOP1.2 或者 GIOP1.3。

GIOP 的网络模型与 OSI/RM 7 层网络协议模型的对照如图 13-1 所示。

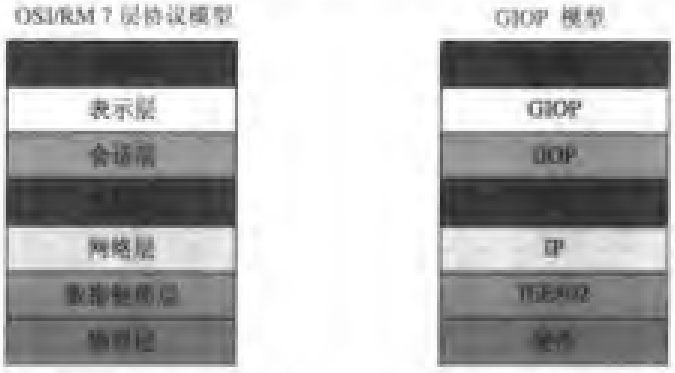


图 13-1　GIOP 的网络模型与 OSI/RM 7 层网络协议模型

GIOP 对数据编码的格式，传输信息的格式和传输协议的要求进行了详细的要求。

公共数据表示（CDR）是传输语法，它将由 OMG IDL 定义的数据类型映射到对象之间传输表示上，它具有如下特征：

（1）变量字节顺序。

主要有 big-endian 和 little-endian 两种。

（2）原始类型字节对齐。

表 13-1 原始类型字节定义

类　　型	占用字节
Char	1
Octet	1
Boolean	1
Short	2
Unsigned short	2
Long	4
Unsigned long	4
Float	4
Enum	4
Long long	8
Unsigned long long	8
Double	8
Long double	8

（3）全部 OMG IDL 映射。

13.2 GIOP 消息类型

GIOP 消息类型如表 13-2 所示。

表 13-2 GIOP 消息类型

消息类型	值	调用者
请求（request）	0	客　户
应答（reply）	1	服务器
撤消应答（cancelRequest）	2	客　户
本地请求（LocateRequest）	3	客　户
本地应答（LocateReplay）	4	服务器
关闭连接（CloseConnect）	5	服务器
消息错误（MessageError）	6	客户/服务器
帧（Fragment）	7	客户/服务器

13.2.1　GIOP 消息头

IDL 定义：

```
module GIOP{                        //IDL extended for version 1.1,1.2, and 1.3
    struct Version{
        octet major;
        octet minor;
    };
  #ifndef GIO_1_1
  //GIOP 1.0
  enum MsgType_1_0 {                //Renamed from MsgType
    Request,Reply,CancelRequest,
    LocateRequest,LocateReply,
    CloseConnect,MessageError
  };
  #else
  //GIOP 1.1
  enum MsgType_1_1 {
    Request,Reply,CancelRequest,
    LocateRequest,LocateReply,
    CloseConnect,MessageError,
    Fragment                        //GIOP 1.1 addition
  };
  #endif                            //GIOP_1_1

  //GIOP 1.0
  struct MessageHeader_1_0{//       //Renamed from MessageHeader
    char magic[4];
    Version GIOP_version;
    boolean byte_order;
    octet message_type;
    unsigned long message_size;
  };

 //GIOP 1.1
  struct MessageHeader_1_1{
    char magic[4];
    Version GIOP_version;
    octet flags;                    //GIOP 1.1 change
    octet message_type;
    unsigned long message_size;
  };

    //GIOP 1.2 , 1.3
    typedef MessageHeader_1_1 MessageHeader_1_2;
    typedef MessageHeader_1_1 MessageHeader_1_3;
};
```

消息定义在结构 MessageHeader 中：

- char magic[4]

用于标识 GIOP 消息，四个大写字母“GIOP”。

- GIOP_Version

GIOP 协议的版本号，主要版本，次要版本。主版本可为 1，副版本可为 1、2 或者 3。

- byte_order

字节顺序，FALSE（0）表示高字节在后（big_endian），TRUE（1）表示低字节在后（little_endian）。

注意：该字段仅适用于 GIOP1.0。

- flags

8 位字节，B0 位用于说明字节顺序，FALSE（0）表示高字节在后（big_endian），TRUE（1）表示低字节在后（little_endian）。B1 位用于说明是否还有后继帧。FALSE（0）表示该消息为最后帧，TRUE（1）表示还有后继帧。B2～B6 保留，必须置 0。

注意：该字段仅适用于 GIGP1.1，1.2 和 1.3。

- message_type

GIOP 消息类型，如表 13-2 所示。

- message_size

消息报文的长度，这里指消息头之后的字节数，该长度不包括消息头长度（12 字节）。报文长度为 0，且消息类型为 Request、LocateRequest、Reply 或者 LocateReply 用于保留今后使用。对于 GIOP1.2 和 1.3 版本，如果 flags 的 B2 为 1，消息报文的长度与消息头长度（12）的和应被 8 整除。

‘G’	‘I’	‘O’	‘P’
消息类型	字节顺序	GIOP_Version	
消息报文长度			

13.2.2 请求消息

请求消息封装了 CORBA 对象调用请求，包括属性访问操作、CORBA::Object 操作 get_interface 和 get_implementation，其流向是从客户向服务器传送。

请求消息由三部分组成：

（1）GIOP 消息头。

（2）请求报文头。

（3）请求报文。

请求头的定义：

```
module GIOP{//IDL extended for version 1.1,1.2, and 1.3
   //GIOP 1.0
   struct RequestHeader_1_0{   //Renamed from RequestHeader
      IOP::ServiceContentList service_context;
      unsigned long request_id;
      boolean response_expected;
```

```
        sequence<octet> object_key;
        string  operation;
        CORBA::OctetSeq requesting_principal;
    };

  //GIOP 1.1
    struct RequestHeader_1_1{   //Renamed from RequestHeader
        IOP::ServiceContentList service_context;
        unsigned long request_id;
        boolean response_expected;
        octet   reserved[3];     //Added in GIOP 1.1
        sequence<octet> object_key;
        string  operation;
        CORBA::OctetSeq requesting_principal;
    };

    //GIOP 1.2 , 1.3
    typedef short AddressingDisposition;
    const short KeyAddr = 0;
    const short ProfileAddr = 1;
    const short ReferenceAddr = 2;

    struct IORAddressingInfo{
        unsigned long selected_profile_index;
        IOP::IOR  ior;
    };

    union TargetAddress switch(AddressingDisposition){
        case KeyAddr: sequence<octet>object_key;
        case ProfileAddr: IOP::TaggedProfile profile;
        case ReferenceAddr: IORAddressingInfo ior;
    };

    struct RequestHeader_1_2{
        unsigned long request_id;
        octet response_flags;
        octet reserved[3];
        TargetAddress target;
        string  operation;
        IOP::ServiceContentList service_context;
        //requesting_principal no in GIOP 1.2 and 1.3
    };

    typedef RequestHeader_1_2 RequestHeader_1_3;
};
```

请求头字段的说明如下：

- request_id

该字段用于将应答（reply）消息与请求消息（request）消息相关，以确保请求与应答报文

相对应。

- response_flags

0x0：表示 NONE 和 WITH__TRANSPORT 设定为 SynaScope。

0x1：表示 WITH__SERVER 设定为 SynaScope。

0x3：表示 WITH__TARGET 设定为 SynaScope。

对于 GIOP1.0，GIOP1.1，response_expected 值为 TRUE 等价于 response_flags 为 0x3，值为 FALSE 等价于 response_flags 为 0x0。

- reserved

保留值，设定为 0，主要保留字段为今后用。

- object_key

用于 GIOP1.0 和 1.1 版本，用于标识调用目标对象，仅对服务器方有意义。

- target

用于 GIOP1.2 和 1.3 版本，用于标识调用目标对象，是个联合结构，其定义的值如下：

（1）KeyAddr：等价于 object_key 域。

（2）ProfileAddr：由客户方 ORB 所选目标 IOR 特定传输 GIOP 结构。

（3）IGRAddressingInfo：目标对象的 IOR。

- operation

IDL 标识命名，限于接口内部上下文，调用操作。

- service_context

包括 ORB 服务信息，由客户向服务器传递。

- requesting_principal

标识请求 principal，支持 BOA::get_principal 操作，改字段仅用于 GIOP1.0 和 1.1，而不适用于 GIOP1.2 和 1.3。

请求体

GIOP1.0 和 1.1 中，请求体紧跟请求头，以 CDR 形式封装了信息。GIOP1.2 和 1.3 中，请求头需按 8 字节对齐，也就是说，如果请求头的长度不是 8 的倍数，将会在请求头补充字段保证请求头长度满足能够被 8 整除，随后才是请求体的数据。

请求体的数据主要有：从左向右所有 in、in out 参数和可选的 context 伪对象。

例如：

```
double example(in short m,out string str,inout long p);
```

等价于如下数据结构

```
struct example_body
{
    short m;          //最左边的输入(in)或者输入输出(in out)参数
    long  p;          //最右边的参数
};
```

13.2.3 应答消息

应答（Reply）消息对应于请求（Request）消息，此时要求请求报文中的应答标志设定为

TRUE。应答报文主要包括输入输出（in out）和输出（out）参数，操作的返回值，以及可能的异常值。另外，应答报文还可包括对象定位信息。

应答消息由如下三部分组成：

（1）GIOP 消息头。

（2）应答报文头。

（3）应答报文（应答体）。

```
module GIOP{                        //IDL extended for 1.2 and 1.3
   #ifndef GIOP_1_2
   //GIOP 1.0 and 1.1
   enum ReplyStatusType_1_0{        //Renamed from ReplyStatusType
      NO_EXCEPTION,
      USER_EXCEPTION,
      SYSTEM_EXCEPTION,
      LOCATION_FORWARD
   };

   //GIOP 1.0
   struct ReplyHeader_1_0{          //Renamed from ReplyHeader
      IOP::ServiceContextList service_context;
      unsigned long request_id;
      ReplyStatusType_1_0 reply_status;
   };

   //GIOP 1.1
   typedef ReplyHeader_1_0 ReplyHeader_1_1;
   //same header contents for 1.0 and 1.1
#else
//GIOP 1.2 , 1.3
   enum ReplyStatusType_1_2{
      NO_EXCEPTION,
      USER_EXCEPTION,
      SYSTEM_EXCEPTION,
      LOCATION_FORWARD,
      LOCATION_FORWARD_PERM,        //new value for 1.2
      NEEDS_ADDRESSING_MODE         //new value for 1.2
   };

   struct ReplyHeader_1_2{
      unsigned long request_id;
      ReplyStatusType_1_2 reply_status;
      IOP::ServiceContextList service_context;
   };

   typedef ReplyHeader_1_2 ReplyHeader_1_3;
#endif //GIOP_1_2
};
```

字段定义如下：

- request_id

与请求相关的应答 ID，与对应请求报文中的 request_id 相一致。

- reply_status

标识相关请求的状态，同时也决定着应答体的内容。如果无异常并且操作成功，其值为 NO_EXCEPTION，应答体包括返回的值。否则应答体会包括：一个异常，直接通知客户重新启动一个请求；或者直接要求客户提供更多的地址信息。

- service_context

由服务器传输给客户的 ORB 服务信息。

应答体

GIOP1.0 和 1.1 中，应答体紧跟应答报文头，以 CDR 形式封装了信息。GIOP1.2 和 1.3 中，应答体需按 8 字节对齐，也就是说，如果应答报文头的长度不是 8 的倍数，将会在应答报文头补充字段保证应答头长度满足能够被 8 整除，随后才是应答体的数据。应答体的内容由 reply_status 状态决定，主要由如下 5 种情形：

- 当 reply_status 值为 NO_EXCEPTION

应答体封装了类似结构的信息，第一个字段是操作的返回值，随后是操作的自左到右的输入输出（inout）和输出（out）参数。

针对请求报文中的例子，其应答的数据接口如下：

```
struct example_reply{
   double return_value;          //返回值
   string str;
   long p;                       //最右的输入输出(inout)参数
};
```

- 当 reply_status 值为 USER_EXCEPTION 或者 SYSTEM_EXCEPTION

此时表明是用户异常或者是系统异常。应答体封装了由操作调用引起的异常。当是系统异常（SYSTEM_EXCEPTION）时，应答体的数据结构格式如下：

```
module GIOP{
    struct SystemExceptionReplyBody{
     string exception_id;
     unsigned long minor_code_value;
     unsigned long completion_status;
    };
};
```

minor_code_value 的高 20 表示供应商次要代码 ID（VMCID，Vendor Minor Codeset ID），低 12 位表示其代码。

- 当 reply_status 值为 LOCATION_FORWARD 时

应答体包括对象引用（IOR），客户 ORB 负责需要再次发送最初的请求，对客户方程序设计应是透明的。

- 当 reply_status 值为 LOCATION_FORWARD_PERM 时

其行为类似 LOCATION_FORWARD，由服务器提供标识给客户方，服务器将使用新的有效的 IOR 代替旧的 IOR。改值现在已经取消。

- 当 reply_status 值为 NEEDS_ADDRESSING_MODE 时

此时应答体包括 GIOP::AddressingDisposition。客户 ORB 使用编址方式重新发送最初的请求，重新发送应对用户是透明的，但在今后 GIOP 版本中可能会取消。

```
unsigned long request_id;
   ReplyStatusType_1_0;
};
```

13.2.4　撤消请求消息

撤消请求（CancelRequest）消息从客户端向服务器发送，该消息将通知服务器，客户方不再需要由请求（Request）或者本地请求（LocateRequest）的应答报文。

撤消请求消息由两部分组成：

（1）GIOP 消息头。

（2）撤消请求头。

撤消请求头的定义如下：

```
module GIOP{
  struct CancelRequestHeader{
    unsigned long request_id;
  };
};
```

request_id 成员字段用于标识由 Request 或者 LocateRequest 消息的 ID。

13.2.5　本地请求消息

本地请求消息（LocateRequest）由客户向服务器传送。主要由两部分组成：

（1）GIOP 消息头。

（2）本地请求头。

本地请求头（LocateRequestHeader）的定义如下：

```
module GIOP{
  //GIOP 1.0
  struct LocateRequestHeader_1_0{//Ranamed LocationRequestHeader
    unsigned long request_id;
    sequence<octet>  object_key;
   };
   //GIOP 1.1
   typedef LocateRequestHeader_1_0 LocateRequestHeader_1_1;

   //GIOP 1.2,1.3
   struct LocateRequestHeader_1_2{
      unsigned long  request_id;
      TargetAddress target;
   };
   typedef LocateRequestHeader_1_2 LocateRequestHeader_1_3;
};
```

request_id、object_key 和 target 字段含义同 Request 请求头说明。

13.2.6 本地应答消息

本地应答（LocateReply）消息与本地请求报文相对应，由三部分组成：

（1）GIOP 消息头。

（2）本地应答头（LocateReplyHeader）。

（3）本地应答体（LocateReplyBody）。

本地应答头的定义如下：

```
module GIOP{
#ifndef GIOP_1_2
//GIOP 1.0 and 1.1
enum LocateStatusType_1_0{      //Renamed form LocateStatusType
   UNKNOWN_OBJECT,
   OBJECT_HERE,
   OBJECT_FORWARD
};

//GIOP 1.0
struct LocateReplyHeader_1_0{  //Renamed from LocateReplyHeader
   unsigned long request_id;
   LocateStatusType_1_0 locate_status;
};
//GIOP 1.1
typedef LocateReplyHeader_1_0 LocateReplyHeader_1_1;

#else
//GIOP 1.2,1.3
enum LocateStatusType_1_2{
   UNKNOWN_OBJECT,
    OBJECT_HERE,
    OBJECT_FORWARD,
    OBJECT_FOWARD_PERM,          //new value for GIOP 1.2
    LOC_SYSTEM_EXCEPTION,        //new value for GIOP 1.2
    LOC_NEEDS_ADDRESSING_MODE    //new value for GIOP 1.2
};

struct LocateReplyHeader_1_2{
   unsigned long request_id;
   LocateStatusType_1_2 locate_status;
};
typedef LocateReplyHeader_1_2 LocateReplyHeader_1_3;
#endif //GIOP_1_2
};
```

locate_status 字段决定了本地应答体的内容：

（1）当 locate_status 值为 UNKNOWN_OBJECT 或者 OBJECT_HERE 时将无本地应答体。

（2）当 locate_status 值为 OBJECT_FORWARD、OBJECT_FOWARD_PERM、LOC_SYSTEM_EXCEPTION 或者 LOC_NEEDS_ADDRESSING_MODE 时，会有本地应答体。

（3）当 locate_status 值为 OBJECT_FORWARD 或者 OBJECT_FOWARD_PERM 时，本地应答体的内容包括对象引用（IOR）。

（4）当 locate_status 值为 LOC_SYSTEM_EXCEPTION 时，本地应答体的内容包括系统异常，其定义为 GIOP::SystemExceptionReplyBody。

（5）当 locate_status 值为 LOC_NEEDS_ADDRESSING_MODE 时，本地应答体的内容为 GIOP::AddressingDisposition。

本地应答体紧随本地应答头。

13.2.7　关闭连接消息

GIOP1.0 和 1.1 中，关闭连接（CloseConnection）消息由服务器向客户方发送，表示服务器将关闭已经建立的网络连接。GIOP1.2 和 1.3 中，服务器和客户方都可发送关闭连接消息。关闭连接消息仅有 GIOP 头，用于标识该报文是关闭消息类型。

13.2.8　消息错误消息

消息错误(MessageError)消息以应答任何 GIOP 消息表明有误，一种是接收者收到的 GIOP 版本号或者是消息类型有错误；另外一种是收到的消息的报文头不正确。消息错误消息仅包含 GIOP 消息头，用于标识其 OIOP 报文类型。

13.3　基于 CORBA 规范的网络协议剖析

下面我们以 Borland 公司（现改名为 Inprise 公司）实时 CORBA（RT-CORBA）的 VisiBroker-RT 为例进行网络协议的剖析，完全支持 CORBA2.6 和 GIOP/IIOP1.2。

通过一个简单的例子，说明嵌入式系统中基于 CORBA 的客户/服务器网络协议，主要分析请求和应答报文。

13.3.1　IDL 定义

用 IDL 编写的 NetSniffInterface 接口定义。

NetSniff.IDL 文件：

```
// NetSniff.idl

typedef short Short8[8];
interface NetSniffInterface
{
   short Sum1(in short x,in short y);
   boolean Sum2(in Short8 x,inout short y);
};
```

13.3.2　服务器方代码

服务器方主要代码部分由接口类实现部分和服务器主控程序两部分组成。

接口类实现部分使用 C++类实现了接口 NetSniffInterface 的方法 Sum1 和 Sum2。类

NetSniffImpl 派生于 POA_NetSniffInterface，并实现了其定义的虚函数。

```
class NetSniffImpl: public virtual POA_NetSniffInterface
//,
//                  public virtual PortableServer::RefCountServantBase
{
 public:

  ::CORBA::Short Sum1(::CORBA::Short _x, ::CORBA::Short _y)
  {
     return (::CORBA::Short)(_x + _y);
  }

   ::CORBA::Boolean Sum2(const _VISanon_arr_8_short _x,::CORBA::Short& _y)
   {
      ::CORBA::Boolean flag=0;
      _y = 0;
      for(::CORBA::ULong i = 0; i < 8;i++)
      {
         _y += _x[i];
      }

      return (flag);
   }

};
```

服务器主控程序实现了如何在 VxWorks 操作系统下完成 NetSniffImpl 实现类。其主要步骤如下：

（1）初始化 ORB。

（2）得到根 POA 引用。

（3）得到 POA Manager。

（4）设置策略。

（5）使用策略创建用户 POA。

（6）创建用户对象。

（7）使用用户 POA 激活用户对象。

（8）激活 POA Manager。

（9）运行 ORB，等待客户请求。

```
//服务器方程序
#include <vxWorks.h>
#include "corba.h"
#include "NetSniff_s.hh"
#include "NetSniffImpl.h"

/* 用户函数说明 */
extern "C" void start_NetSniff_server(void);
static void NetSniff_server(void);
```

```
    extern CORBA::ORB_var     orb;       /* 全局变量,由 ORB_init 初始化得到 */

    void start_NetSniff_server(void)
    {
       taskSpawn("NETSNIFF_SRVR",100,VX_FP_TASK,20000,(FUNCPTR)NetSniff_server,
0,0,0,0,0,0,0,0,0,0);
    }

    void NetSniff_server()
    {
       PortableServer::POA_var rootPOA;
       FILE *fp;

       VISTRY
       {
          //得到根 POA 的引用
          CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
          rootPOA = PortableServer::POA::_narrow(obj);

          //得到 POA Manager
          PortableServer::POAManager_var poa_manager = rootPOA->the_POAManager();

          //设置策略
          CORBA::PolicyList policies;
          policies.length(1);
          policies[(CORBA::ULong)0]=rootPOA->create_lifespan_policy
(PortableServer::PERSISTENT);
          //使用策略创建用户 POA
          PortableServer::POA_var myPOA;
          myPOA   =   rootPOA->create_POA("NetSniff_account_poa",   poa_manager,
policies);

          //创建用户对象
          NetSniffImpl *managerServant = new NetSniffImpl;

          //创建用户对象 ID
          PortableServer::ObjectId_var managerId;
          managerId = PortableServer::string_to_ObjectId("NetSniffManager");

          //在用户 POA 上使用对象 ID 激活服务器对象
          myPOA->activate_object_with_id((CORBA_OctetSequence      &)managerId,
managerServant);
          poa_manager->activate();     // 激活 POA Manager

          //得到对象 IOR,并输出到用户终端
          CORBA::Object_var ref = myPOA->servant_to_reference(managerServant);
```

```
            CORBA::String_var string_ref = orb->object_to_string(ref.in());
            cout<<endl<<"CORBA Object ==> "<<endl<<ref<<endl<<string_ref<<" is
ready"<<endl ;
            fp=fopen("host:demolior.ref","w");
            fprintf(fp,"%s",(char *)string_ref);
            fclose(fp);

            orb->run();
        }
        VISCATCH(CORBA::Exception, e)
        {
            cerr << e << endl;
            taskSuspend(0);
        }
        VISEND_CATCH
            return;
    }
```

13.3.3 客户方代码

客户方程序实现了如何在 VxWorks 操作系统下调用 NetSniffInterface 接口的方法。其主要步骤如下：

(1) 初始化 ORB。

(2) 得到对象标识。

(3) 绑定对象。

(4) 调用对象方法。

```
//客户端程序
#include <vxWorks.h>
#include "corba.h"
#include <vport.h>
#include "NetSniff_c.hh"

/* 用户函数说明 */
extern "C" void start_NetSniff_client(const char* name);
static void NetSniff_client(const char* name,int flag);

extern CORBA::ORB_var orb;              /* 全局变量,由 ORB_init 初始化得到 */

void start_NetSniff_client(const char* name,int flag)
{
    taskSpawn("NETSNIFF_CLNT",100,VX_FP_TASK,20000,(FUNCPTR)NetSniff_client,
(int)name,flag,0,0,0,0,0,0,0,0);
}

void NetSniff_client(const char* name,int flag)
{
```

```
        VISTRY
        {
            NetSniffInterface_var client;    //使用 POA 名字和服务器 ID 定位对象.

            PortableServer::ObjectId_var managerId=PortableServer::string_to_ObjectId
("NetSniffManager");
            client=NetSniffInterface::_bind("/NetSniff_account_poa",
(CORBA_OctetSequence &)managerId);

            ::CORBA::Short x=10,y=20;
            ::CORBA::Short ret;
            ret = client->Sum1(x,y);
            printf("%d + %d = %d\n",x,y,ret);

            ::CORBA::Short xx[8]={1,2,3,4,5,6,7,8};
            ::CORBA::Short retxx;
            ::CORBA::Boolean bFlag;
            bFlag = client->Sum2(xx,retxx);
            printf("%d",retxx);

        }
        VISCATCH(CORBA::Exception, e)
        {
            cerr << e << endl;
        }
        VISEND_CATCH
            return;
    }
```

13.3.4　监听的网络报文解析

客户调用接口 NetSniffInterface 方法 Sum1 时发送的请求报文如图 13-2 所示，在 GIOP 消息头和请求报文头之后的请求报文体中的数据结构是方法 Sum1 的输入参数 x 和 y，其对应的值为 10（0x000A）和 20（0x0014），请求服务器计算 x+y 的值。该 GIOP 采用的版本为 GIOP1.2，请求的标识为 2，需要返回的应答标识应为 2。需要注意的是在对象键（object key）中包括用户 POA 信息和 POA Manager 的信息。

图 13-3 说明了 Sum1 在服务器上被运行之后，产生的应答报文，并发送给客户的情形，可以明显地看到应答报文的 GIOP 消息头和应答报文头之后的应答体中，Sum1 的输出结果为 30（10+20），对应的 16 进制为 0x001E。应答报文头中表明本次调用成功，无异常，返回请求标识为 2 的应答报文。

图 13-4 说明了客户调用接口 NetSniffInterface 方法 Sum2 时发送的请求报文，在 GIOP 消息头和请求报文头之后的请求报文体中的数据结构是方法 Sum2 的 8 输入数组参数，其对应的值为 1（0x0001）、2（0x0002）、3（0x0003）、4（0x0004）、5（0x0005）、6（0x0006）、7（0x0007）、8（0x0008），以及输入输出的参数的值，由于程序未对该变量进行初始化，其对应的初始值与编译器有关，这里为 0XEEEE。该请求报文的请求标识为 4，要求应答报文的请求标识也为 4，以表明请求应答报文相对应。

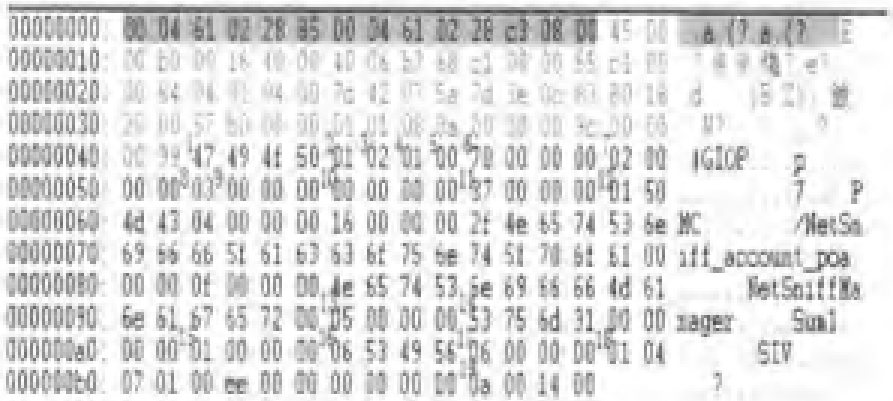


图 13-2　请求报文方法 Sum1 的网络报文解析

1～6 为 GIOP 消息头：

1—GIOP 头，4 个大写字母“GIOP”；2—主版本为 1；3—副版本为 2；4—标志字，字节顺序为低字节在后，无后续帧；

5—报文类型为请求报文；6—消息报文长度为 112 字节；

7～18 为请求报文头：

7—请求标志为 2；8—应答标志字为 3；9—保留，3 个字节；10—GIOP1.2 的 KeyAddr 的值；

11—对象键长度（55 字节）；12—对象键值；13—客户调用方法名的长度；

14—客户调用方法的名称；15～18—服务上下文

19 为请求体

19—输入的参数数据为 10，20，向服务器请求计算 10+20 的值

```
IP   D=[193.0.0.101] S=[193.0.0.100] LEN=58 ID=68
TCP  D=1025 S=1024     ACK=2101479362 SEQ=2101218435 LEN=26 WIN=8192
GIOP  ------ CORBA IIOP ------
  GIOP
  GIOP  GIOP String              = 'GIOP'
  GIOP  Major Version            = 1
  GIOP  Minor Version            = 2
  GIOP  Flags                    = 01
  GIOP                0000 00    = Reserved
  GIOP                     0     = (No More Fragments)
  GIOP                      1    = Little Endian
  GIOP  Message Type             = 1 (Reply)
  GIOP  Message Size             = 14
  GIOP  Request ID               = 2
  GIOP  Reply Status             = 0 (NO_EXCEPTION)
  GIOP  Service Ctxt Entries     = 0
  GIOP  Out & InOut Parameters     (2 bytes)
  GIOP
```

```
00000000  00 04 61 02 29 c3 00 04 61 02 29 85 08 00 45 00  ..a.).. a.)...E.
00000010  00 4e 00 44 40 00 40 06 b7 9c c1 00 00 64 c1 00  .N.D@.@. .....d..
00000020  00 65 04 00 04 01 7d 3e 0c 93 7d 42 07 d6 50 18  .e....}> ..}B..P.
00000030  20 00 f5 24 00 00 01 01 08 0a 00 00 00 99 00 00  ..$..... ........
00000040  00 9c 47 49 4f 50 01 02 01 00 00 00 02 00 47 49  ..GIOP.. ......GI
00000050  00 00 00 00 00 00 00 00 00 00 00 00 1e 00        ..... ........
```

图 13-3　应答报文方法 Sum1 的网络报文解析

1~6 为 GIOP 消息头

1—GIOP 头，4 个大写字母 "GIOP"；2—主版本为 1；3—副版本为 2；4—标志字，字节顺序为低字节在后，无后继帧；

5—报文类型为应答报文；6—消息报文长度为 112 字节

7~9 为应答报文头

7—对应的请求标志；8—应答标志（无异常）；9—服务上下文

10 为应答体

10—向客户方反回的输出结果，服务器计算 10+20 的值为 30

图 13-5 说明了 Sum2 在服务器上被运行之后，产生的应答报文，并发送给客户的情形。可以明显地看到应答报文的 GIOP 消息头和应答报文头之后的应答体中，Sum2 的输出结果为 36（1+2+3+4+5+6+7+8），对应的 16 进制为 0x0024，返回的标志为 0。应答报文头中表明本次调用成功，无异常，返回请求标识为 4 的应答报文。

图 13-4　请求报文方法 Sum2 的网络报文解析

1～6 为 GIOP 消息头

1—GIOP 头，4 个大写字母“GIOP”；2—主版本为 1；3—副版本为 2；4—标志字：字节顺序为低字节在后，无后继帧；

5—报文类型为请求报文；6—消息报文长度为 112 字节

7～18 为请求报文头

7—请求标志为 4；8—应答标志字为 3；9—保留，3 个字节；10—GIOP1.2 的 KeyAddr 的值；

11—对象键长度（55 字节）；12—对象键值；13—客户调用方法名的长度；

14—客户调用方法的名称；15～18—服务上下文

19 为请求体

19—输入的参数数据为 1、2、3、4、5、6、7、8 及 0xeeee，请求服务器计算 8 个数的和

图 13-5　应答报文方法 Sum2 的网络报文解析

1~6 为 GIOP 消息头

1—GIOP 头，4 个大写字母 "GIOP"；2—主版本为 1；3—副版本为 2；4—标志字：字节顺序为低字节在后，无后继帧；

5—报文类型为应答报文；6—消息报文长度为 112 字节

7~9 为应答报文头

7—对应的请求标志；8—应答标志（无异常）；9—服务上下文

10 为应答体

10—向客户方反回的输出结果，返回值为 0，计算数组之和为 36