信息科学与技术丛书

嵌入式可配置实时操作系统
eCos 开发与应用

蒋句平　编著

机 械 工 业 出 版 社

嵌入式可配置实时操作系统 eCos 是一种完全免费的开放源代码的软件，适合于深度嵌入式应用。本书全面介绍 eCos 系统的基本结构，详细描述 eCos 开发环境的建立及其配置方法，从各个层次对 eCos 的各个组成部分进行阐述和分析，并通过实例说明如何开发基于 eCos 的嵌入式应用。本书附带光盘包含了最新版本的 eCos 2.0 源代码、联机说明、开发工具源代码以及应用程序 eCos 嵌入式 Web 服务器，可以直接使用光盘内容建立完整的 eCos 开发环境。

本书可作为广大嵌入式系统研发人员及其他相关科研人员的技术参考书，也可作为在校学生学习嵌入式系统的参考教材。

图书在版编目（CIP）数据

嵌入式可配置实时操作系统 eCos 开发与应用/蒋句平编著. —北京：机械工业出版社，2004.1
（信息科学与技术丛书）
ISBN 7-111-13242-4

Ⅰ.嵌… Ⅱ.蒋… Ⅲ.实时操作系统，eCos Ⅳ.TP316.2

中国版本图书馆 CIP 数据核字（2003）第 094918 号

机械工业出版社（北京市百万庄大街 22 号 邮政编码 100037）
策 划：胡毓坚
责任编辑：时 静
责任印制：

·新华书店北京发行所发行

2004 年 1 月第 1 版·第 1 次印刷
787mm×1092mm 1/16·23 印张·569 千字
0001—5000 册
定价：34.00 元

出 版 说 明

随着信息科学与技术的迅速发展，人类每时每刻都会面对层出不穷的新技术、新概念。毫无疑问，在节奏越来越快的工作和生活中，人们需要通过阅读和学习大量信息丰富、具备实践指导意义的图书，来获取新知识和新技能，从而不断提高自身素质，紧跟信息化时代发展的步伐。

众所周知，在计算机硬件方面，高性价比的解决方案和新型技术的应用一直备受青睐；在软件技术方面，随着计算机软件的规模和复杂性与日俱增，软件技术受到不断挑战，人们一直在为寻求更先进的软件技术而奋斗不止。目前，计算机在社会生活中日益普及，随着因特网延伸到人类世界的层层面面，掌握计算机网络技术和理论已成为大众的文化需求。也正是由于信息科学与技术在电工、电子、通信、工业控制、智能建筑、工业产品设计与制造等专业领域中已经得到充分、广泛的应用，所以这些专业领域中的研究人员和工程技术人员将越来越迫切需要汲取自身领域信息化所带来的新理念和新方法。

针对人们对了解和掌握新知识、新技能的热切期待以及由此促成的人们对语言简洁、内容充实、融合实践经验的图书迫切需要的现状，机械工业出版社适时推出了“信息科学与技术丛书”。这套丛书涉及计算机软件、硬件、网络、工程应用等内容，注重理论与实践相结合，内容实用，层次分明，语言流畅，是信息科学与技术领域专业人员不可或缺的图书。

现今，信息科学与技术的发展可谓一日千里，机械工业出版社欢迎从事信息技术方面工作的科研人员、工程技术人员积极参与我们的工作，为推进我国的信息化建设作出贡献。

机械工业出版社

前　言

嵌入式产品是一类充满巨大商机的产业。在开发嵌入式产品的时候，开发人员将面临选择哪种嵌入式操作系统的问题。有许多因素值得考虑：软件的价格问题、版税问题、开发工具的好坏、是否提供源代码、所提供的实时操作系统具有哪些特性等等。在许多情况下，价格和版税是首先考虑的因素，低成本的解决方案是一种最好的选择，降低成本价格是提高产品竞争力的一个重要举措。另一个必须考虑的问题是前期投资问题，将现有软件代码移植到新的硬件平台上时这一问题显得尤为突出，软件的移植能力在很大程度上会直接影响产品的开发周期。另外，软件的可重用性和配置能力也是开发嵌入式产品必须考虑的因素。采用嵌入式可配置实时操作系统 eCos 便是针对所有这些问题的一个很好的解决方案。

eCos 于 1997 年起源于 Cygnus 公司，后来成为 RedHat 的一个非 Linux 嵌入式操作系统。在短短的几年时间内，eCos 的发展极为迅速，它已逐渐被人们接受，并受到越来越多的嵌入式产品开发人员的青睐，目前市场上已经有了许多成功应用 eCos 的嵌入式产品。eCos 是一种适合于深度嵌入式应用的实时操作系统，它是一种免费的、无版权限制（无版税）的开放源代码的软件。eCos 的独特之处是它的可配置能力和配置机制，这是其他嵌入式操作系统无法比拟的。此外，eCos 还具有良好的开放性、兼容性和可扩展性，可移植能力强，目前它能支持十余种市场上流行的嵌入式处理器。开发人员在不同平台上进行移植时，几乎不用修改或稍加修改就可以完成应用程序的移植工作。

eCos 不仅是开发嵌入式产品的一个很好的选择，还是学习嵌入式操作系统原理与方法的一个很好的实例教材。eCos 除了提供了嵌入式实时操作系统所必须具备的全部功能外，它的配置机制、组织结构以及软件源码都值得其他软件借鉴。与其他嵌入式操作系统相比，eCos 简单、易学、易于操作，利用普通微机就可以很容易地建立一个完整的开发环境，而且不需要特殊的设备。加之它又是一种免费软件，因此特别适合国内学生将其作为学习和研究嵌入式系统的理想平台。为此，本书大部分操作和例子都以普通微机作为目标系统。本书光盘提供了 eCos 的全部源码、开发工具和应用程序，读者可以直接使用光盘提供的内容建立完整的开发平台。

全书共分十三章，为兼顾初学者和有经验的开发人员，采用从简到繁、由浅入深的方式对 eCos 进行全面阐述，读者按照本书内容的编排可以逐步了解和掌握 eCos，并且可以着手开发自己的 eCos 应用。

本书读者对象包括广大嵌入式系统研发人员、在校学生以及其他嵌入式系统爱好者，要求具有一定的软件（C 语言）编程经验和基本的硬件知识。

本书的编写得到了我的同事及家人的帮助和支持，窦强博士对本书初稿进行了细致的审阅，在此谨向他们表示衷心的感谢！

由于编者水平有限，书中难免存在一些错误和缺点，殷切希望读者批评指正。

编　者

目　录

第1章 概 述

随着信息技术的飞速发展和互联网的迅速普及，人们已经步入了数字化时代。形式多样的数字化产品已经开始成为继 PC 机之后信息处理的主要工具，消费电子、计算机、通信（3C）一体化已经成为数字化产品的一种趋势，并且正在逐步形成一个充满商机的巨大产业。在这样一种数字化潮流下，嵌入式系统已成为当前研究和应用的热点之一。嵌入式产品已经涉及到人们日常生活和工作的各个方面，手机、个人数字助理 PDA、MP3 播放器、机顶盒、VCD、智能外设、数码相机、数码电视、网络家电、GPS 设备、数控机床等各种各样的数字设备都是广泛使用的嵌入式产品。

嵌入式系统是一种以应用为中心、软硬件可裁减的专用计算机系统。它具有很强的灵活性，可以适应应用系统对功能、可靠性、成本、体积、功耗等方面的严格要求。嵌入式系统主要由嵌入式处理器、相关支撑硬件、嵌入式操作系统以及应用软件系统等组成，它是一种集软、硬件于一体的可独立工作的设备或组件。嵌入式操作系统是一种实时的、支持嵌入式系统应用的操作系统软件，它是嵌入式系统极为重要的组成部分，通常包括与硬件相关的底层驱动软件、系统内核、设备驱动接口、通信协议、图形界面等。与通用操作系统相比较，嵌入式操作系统在系统实时高效性、硬件的相关依赖性、软件固态化以及应用的专用性等方面具有较为突出的特点。

目前，嵌入式操作系统的种类较多，其中比较流行的有 VxWorks、Windows CE、pSOS、Palm OS、嵌入式 Linux、QNX 和 Nuclear 等产品。这些嵌入式操作系统在开放性、实用性以及性能等方面各有千秋，而且大多数为商用产品。除了商用产品外，另外还有一些免费的嵌入式操作系统，eCos 就是这些产品的一个典型代表。嵌入式可配置操作系统 eCos（Embedded Configurable Operating System）是一种开放源代码软件，它是一种免费、无版权限制的适合于深度嵌入式应用的实时操作系统。eCos 最为显著的特点是它的可配置性、可裁剪性、可移植性和实时性，它的一个主要技术创新是其功能强大的配置系统，可以在源码级实现对系统的配置和裁剪。正是由于这些特性，eCos 已引起越来越多的关注，同时也吸引了越来越多的厂家使用 eCos 开发其新一代嵌入式产品。

1.1 什么是 eCos

eCos 是一种嵌入式可配置实时操作系统，适合于深度嵌入式应用，主要应用对象包括消费电子、电讯、车载设备、手持设备以及其他一些低成本和便携式应用。eCos 是一种开放源代码软件，无任何版权费用。eCos 具有很强的可配置能力，而且它的代码量很小，通常为几十到几百 KB。它的最小配置形式是它的硬件抽象层 HAL 所提供的引导程序 RedBoot，可以支持很大范围内许多不同的处理器和平台。它的最大配置形式是一个完整的实时操作系统，所提供的服务和支持能与其他大多数商用实时操作系统相媲美。eCos 为开发人员提供了一个能涵盖大范围内各种不同嵌入式产品的公共软件基础结构，使得嵌入式软件开发人员可以集中

精力去开发更好的嵌入式产品,而不是停留在对实时操作系统的开发、维护和配置上。

1.1.1 起源与历史

eCos 最初起源于 Cygnus 公司。Cygnus 创建于 1989 年,创始人为 Michael Tiemann、David Henkel-Wallace 和 John Gilmore,其目的是为开源软件提供高质量的开发和支持。经过几年的艰苦努力,最后推出了今天被人们广泛使用的 GNUPro 开发工具包,包括 GCC (ANSI-C 编译器)、G++ (C++ 编译器)、GDB (源码级和汇编级调试工具)、GAS (GNU 汇编器)、LD (GNU 链接器)、Cygwin (Windows 下的 UNIX 环境)、Insight (GDB 图形界面 GUI) 等。

Cygnus 对 eCos 的设计始于 1997 年春季,其主要目的是为市场提供一种低成本、高效率、高质量的嵌入式软件解决方案,同时要求该软件所占系统资源极少。eCos 和 GNUPro 相辅相成,扩大了 Cygnus 的产品线。eCos 从设计之初就考虑到了嵌入式系统中内存资源的限制以及嵌入式硬件平台的多样性。通过与其他许多半导体公司的协作,Cygnus 成功构造了一个可以对硬件层进行抽象的实时操作系统 (RTOS),并且具有高度可配置性。这些特性使得 RTOS 可以适合于各种各样的嵌入式系统,这种 RTOS 就是 eCos。eCos 的高度可配置性可以显著缩短嵌入式产品的开发周期。

Cygnus 对 eCos 的另一个设计目标是降低嵌入式产品的成本。低成本是嵌入式系统开发中必须考虑的一个重要因素。通过使用开放源代码的形式,eCos 基本上不需要任何费用。它是一种完全免费的软件,任何公司和个人都可以直接从 Internet 上下载其源码和相应的开发工具,并且可以自由地进行修改和扩展,eCos 产品的发布也无需交纳任何版权费用。用户可以自由使用 eCos,但是要求公布对 eCos 的改动,这是为了提高或促进 eCos 发展的一种措施。当然,用户的应用程序不必公开。

Cygnus 于 1998 年 11 月发布了第一个 eCos 版本 (eCos 1.1)。它只支持有限的几种处理器结构:Matsushita MN10300、Toshiba TX39 和 PowerPC。1999 年 5 月,Cygnus 发布了 eCos 的第二个版本 eCos 1.2.1,它在 eCos 1.1 的基础上增加了许多新的特性,并扩大了对处理器的支持范围,包括 ARM7、SPARClite MB8683x 系列处理器、PowerPC (MPC860、MPC850、MPC823)、VR4300 和 SH3 等。

1999 年 11 月,Red Hat 收购了 Cygnus 公司。在此后的几年里,eCos 作为 Red Hat 的一个嵌入式产品得到了迅速的发展。2000 年 3 月,Red Hat 发布了 eCos 的第三个版本 eCos 1.3.1。eCos 1.3.1 在 1.2.1 版本的基础上又增加了许多新的特性 (TCP/IP 协议栈、PCI 支持等),并扩充了对处理器平台的支持,包括 ARM Thumb、ARM9、StrongARM、AM33、PowerPC、VR4300、SH3、x86 等。2000 年 8 月,eCos 增加了对目标系统引导和调试固件 (Firmware) 的支持,即 RedBoot,它是 Red Hat 的一个标准嵌入式系统引导和 Debug 环境。RedBoot 目前已被许多嵌入式产品所采用。

2002 年,Red Hat 由于财务方面的原因,裁减了 eCos 开发队伍,这在一定程度上影响了人们对 eCos 的信心。但 eCos 的发展并没有因此而停止,RedHat 随后宣称将继续支持 eCos 的发展,而原来的 eCos 主要开发人员组建了一个新的 eCosCentric 公司,继续进行 eCos 的开发和技术支持。2002 年 4 月,eCosCentric 发布了 eCos 2.0 alpha 版,2003 年 3 月,又发布了 eCos 2.0 beta 版,2003 年 5 月,正式发布了 eCos 2.0。它所支持的处理器包括:ARM、StrongARM、XScale、SuperH、Intel x86 (IA32)、PowerPC、MIPS、AM3x、Motorola 68K/Coldfire、SPARC、

Hitachi H8/300H 和 NEC V850 等，能支持近百种当时市场上广泛使用的嵌入式系统开发平台和评估版。eCos 2.0 与它的前一个版本 eCos 1.3.1 相比，增加和改进的功能包括：

1）RedBoot，基于 eCos 的引导和调试 Firmware。

2）TCP/IP 协议栈，支持 BOOTP/DHCP、DNS、TFTP/FTP、IPv6 和 HTTPD。

3）RAM、ROM 和 Flash 文件系统。

4）电源管理。

5）USB 支持。

6）POSIX 兼容 API。

7）对称多处理器 SMP 支持。

目前，许多公司都在使用 eCos，并先后成功推出了使用 eCos 的嵌入式产品。部分产品有：Brother HL-2400 CeN 网络彩色激光打印机、Delphi Communiport 车载信息处理系统（MPU）、Iomega Hip Zip 数字音频播放器、Ikendi 指纹识别系统、3G LAB 移动电话、GPS 卫星地面设备、MP3 播放器、CrosStor RAID 系统等等。

1.1.2 功能与特性

eCos 是一个适合于深度嵌入式应用的开放源代码实时操作系统。它能满足嵌入式 Linux 难以满足的对嵌入空间的需求，Linux 目前内核最小约 500KB，占用 1.5 MB 内存，而 eCos 只占用几十到几百 KB。eCos 使用了多任务抢占机制，具有最小的中断延迟，支持嵌入式系统所需的所有同步原语，并拥有灵活的调度策略和中断处理机制。eCos 还提供了普通嵌入式应用中所需要的全部功能，包括设备驱动程序、内存管理、例外处理、标准 C、数学库等等。除了这些对系统运行时的支持外，eCos 所提供的支持还包括开发嵌入式应用所需的所有工具，如 eCos 配置和编译工具、基于 GNU 的编译器、汇编器、链接器、调试器和模拟器。

eCos 提供的基本功能如下：

1）硬件抽象层（HAL）。

2）实时内核：

① 中断处理。

② 例外处理。

③ 可选择的调度器。

④ 多线程支持。

⑤ 一组丰富的同步原语。

⑥ 定时器、计数器、告警器。

⑦ 内存分配算法选择。

⑧ 调试和测试支持。

3）μITRON 3.0 兼容 API。

4）POSIX 兼容 API。

5）ISO C 和数学库。

6）串口、以太网、墙上时钟和看门狗设备驱动程序。

7）USB 支持。

8）TCP/IP 网络栈，包括：

① BOOTP/DHCP。
② DNS。
③ TFTP/FTP。
④ SNMP。
⑤ IPv6。
⑥ HTTPD。
9）文件系统：
① JFFS2 Flash 文件系统。
② RAM 文件系统。
③ ROM 文件系统。
10）电源管理。
11）GDB debug 支持。
eCos 的主要特性包括：
① 开放源代码。
② 免费软件，无版权费用。
③ 高度可配置性。
④ 易于移植。
⑤ 实时系统。
⑥ 代码量小。
⑦ 符合标准协议。
⑧ 网络支持。

1.1.3 eCos 核心组件

实时嵌入式操作系统通常应该提供一些标准功能，这些功能包括线程同步机制、调度机制、中断处理、例外和错误处理、定时机制以及设备驱动程序等。eCos 通过提供以实时内核为核心的一些核心组件来实现这些标准功能。这些核心组件包括：

⑴ 硬件抽象层（HAL）：对硬件平台进行抽象，为上层软件对硬件的控制和访问提供一个标准接口。硬件抽象层的实现保证了 eCos 系统具有良好的可移植性。

⑵ 内核：内核包含了中断和例外处理机制、多线程机制、同步机制、可供选择的多种调度机制、定时机制、计数器等。

⑶ ISO C 和数学库：提供与标准兼容的函数和调用。

⑷ 设备驱动程序：对一些典型的设备提供驱动程序支持，包括串口驱动程序、以太网驱动程序、Flash ROM 驱动程序、USB 驱动程序、PCMCIA 驱动程序等等。

⑸ GNU debugger（GDB）支持：为目标平台上的软件与 GDB 主机之间提供通信机制，实现对目标平台硬件和软件的调试。

eCos 系统及其应用程序以特权方式运行，没有用户方式和内核方式之分。eCos 还提供了一些对系统基本结构进行测试的测试程序，这些测试程序也可以与系统一样进行类似的配置，使其能精确地对系统所进行的配置进行测试。

1.1.4 对硬件的支持

eCos 支持当前流行的大部分嵌入式处理器。eCos 具有很好的可移植特性，它可以在 16 位、32 位和 64 位等不同体系结构之间以及它们的各种不同平台之间进行移植。eCos 的内核、库以及运行组件位于硬件抽象层（HAL）之上，只要将硬件抽象层和相关的设备驱动程序进行移植，eCos 及其应用程序就可以在新的目标平台上运行。因此，厂家在进行产品开发时，在硬件结构的选型方面具有很大的选择余地。

eCos 目前支持十几种处理器，包括这些处理器的多种变体和多种典型开发板。eCos 源码支持当前市场上最为流行的各种不同处理器结构的标准商用评估板。下面是 eCos 目前所支持的主要处理器（对硬件的详细支持情况见“附录 A”）：

① ARM。
② Fujitsu FR-V。
③ Hitachi H8/300。
④ Intel x86。
⑤ Matsushita AM3x。
⑥ MIPS。
⑦ NEC V8xx。
⑧ PowerPC。
⑨ Samsung CalmRISC16/32。
⑩ SPARC。
⑪ SPARClite。
⑫ SuperH。

读者可以在 eCos 网站上查阅 http://sources.redhat.com/ecos/hardware.html，了解 eCos 对硬件支持的最新情况。

1.1.5 eCos 资源

eCos 是一种开放源代码的软件，它的源代码和开发工具、配置工具都可以从 eCos 网站上免费下载。相关网站和主要网址如下：

eCos 网址：http://ecos.sourceware.org/或 http://sources.redhat.com/ecos。
RedHat 网址：http://www.redhat.com。
eCosCentric 网址：http://www.ecoscentric.com。
GNU 网址：http://www.gnu.org。
Cygwin 网址：http://www.cygwin.com。
eCos 源码下载地址：ftp://sources.redhat.com/pub/ecos。
GNU 开发工具包下载地址：ftp://ftp.gnu.org/gnu/。
图形配置工具下载地址：http://sources.redhat.com/ecos/ct2.html。

eCos 源码采用 CVS 系统进行版本管理，它的 CVS 服务器上具有最新的 eCos 源码。用户可以登录该 CVS 服务器及时更新 eCos 源码。eCos 的 CVS 服务器的登录方式（Linux 或 Cygwin 环境下）为：

cvs -d :pserver:anoncvs@sources.redhat.com:/cvs/ecos login

对于一些不能使用 CVS 服务的用户，eCosCentric 提供了一个及时下载 eCos 源码的方法，它使用 eCos 源码“快照（snapshot）”的形式，定期将最新的 eCos 源码打包并放置到其网址上，用户可以到下面的网址通过 FTP 或 HTTP 将其下载：

http://www.ecoscentric.com/snapshots

在进行嵌入式产品开发时，开发人员最为关心的是开发过程中的技术支持与服务。对于 eCos 来说，可以根据具体情况采用不同的方式来获取技术支持。eCos 提供了六个邮件列表（见表 1-1），用户可以在这些邮件列表中查找、订阅以及提交需要咨询的各种问题。用户在使用邮件列表时，可以先在其主页上查找相关问题的的答案，然后再通过 email 寻求更多的帮助，通常在两到三天内便会得到技术支持。订阅这些邮件列表的网址是：

http://sources.redhat.com/ecos/intouch.html

表 1-1　eCos 邮件列表

邮件列表	描　述	网　址	
Discussion List	开发人员对 eCos 的技术讨论、技术支持和帮助	主页	http://sources.redhat.com/ml/ecos-discuss
		email	ecos-discuss@sources.redhat.com
Patches List	用于发布 eCos 补丁	主页	http://sources.redhat.com/ml/ecos-patches
		email	ecos-patches@sources.redhat.com
Development List	eCos 当前开发进展情况，包括新特性、新的硬件支持	主页	http://sources.redhat.com/ml/ecos-devel
		email	ecos-devel@sources.redhat.com
Announcement List	eCos 重要新闻、新版本的发布、新特性	主页	http://sources.redhat.com/ml/ecos-announce
		email	ecos-announce@sources.redhat.com
CVS Web Pages List	由 CVS 系统维护的 eCos 主页变化通知（只读）	主页	http://sources.redhat.com/ml/ecos-webpages-cvs
CVS List	由 CVS 系统维护的 eCos 源码库变化通知（只读）	主页	http://sources.redhat.com/ml/ecos-cvs

除了这些对 eCos 的免费支持外，eCosCentric 公司还提供商业支持。它为客户提供技术咨询、技术帮助、解决方案、为客户硬件定制软件以及测试等专业支持。

eCos 还提供了一个 Bug 报告和跟踪的方法。Bug 的查询可以在下面的网址内进行：

http://bugzilla.redhat.com/bugzilla/query.cgi?product=eCos

报告新的 Bug 可以进入下面的网址：

http://bugzilla.redhat.com/bugzilla/enter_bug.cgi?product=eCos

1.2　eCos 的可配置性

eCos 在嵌入式系统软件方面最具创新意义的贡献是它的配置结构和配置方法。以往，大多数嵌入式系统的开发需要开发人员自己对应用进行适当的调整和改变来适应底层的操作系

统,但 eCos 改变了这种情况。eCos 为开发人员提供了大范围的可选项来对底层操作系统进行配置,使其能够更好地满足应用的需求,这样就变成了调整操作系统来适应应用。当前,eCos 提供了 200 多个配置选项,典型的配置选项包括调度器的类型和任务优先级别数目等。目前,已经有一些软件公司正在借鉴 eCos 的这种配置方式。

1.2.1 配置的必要性

在了解 eCos 结构时,首先必须对构成 eCos 系统的组件框架 (Component Framework) 有所了解。eCos 组件框架的主要目的是为了满足嵌入式系统的设计需求,它是对系统进行配置的一个主要途径。通过组件框架的使用,可以利用具有可重用性的软件组件和软件模块来建造具有众多功能的应用。eCos 的组件框架可以用来对组件进行控制,减少对内存的使用。它还允许用户对与时间相关的行为进行控制,从而满足实时系统的需要。采用组件框架后,用户在编程时可以使用普通的编程语言,包括 C、C++ 以及汇编语言 (硬件抽象层中的某些功能的实现必须使用汇编语言)。

目前大多数的嵌入式系统软件都有可能包含了在具体应用中不会被使用的一些功能。这些嵌入式软件由于包含了一些实际系统不需要的功能,从而产生了多余的代码。这些多余代码增加了软件的复杂性,并且造成了系统资源的浪费。多余代码越多,系统的可靠性也越低。举例来说,即使是一个简单的仅仅是输出“Hello World”的应用程序,在大多数的实时操作系统中也要包含诸如互斥以及任务调度机制等这样一些实际应用根本不需要的功能。而 eCos 却不同,它将系统运行组件的最后控制权交给了开发人员,从而可以非常容易地删除那些实际应用中不需要的功能和代码。根据具体应用的不同需要,eCos 系统可以被裁剪和扩充,其代码量可能是几十 KB,也可能是几百 KB。

通过 eCos 组件框架的使用,开发人员可以选择满足应用需要的组件,针对应用的实际需要对某些组件进行配置。这种配置可以是使能或禁止某个特性,也可以是选择组件的某种实现。以 eCos 内核调度器的配置为例,eCos 为开发人员提供了一些选项,如优先级数目的选择、决定是否使用时间片等等。通过对这些选项进行配置,任何不需要的代码都将从最后所产生的映像中剔除掉。

eCos 的可配置性还使得建立一个可重用的软件组件库成为可能。由于具有很强的可移植性和广泛的适用性,可重用软件组件的使用可以显著减少产品的开发周期。eCos 的组件框架还鼓励第三方软件的开发,从而扩展 eCos 核心组件的特性和功能。正是由于越来越多的开发人员朝着这一目标的努力并将他们的成果返回给 eCos,因此可以说 eCos 在功能方面的发展将是无限的。

1.2.2 配置方法

在嵌入式系统向更小、更快、更便宜和更复杂的方向发展时,需要对系统中的软件组件进行控制。对一个应用映像中所包含的组件进行控制具有多种不同方法。eCos 对组件控制的原则是尽量减少系统程序代码体积、节省系统资源、降低系统崩溃的可能性。采用这种原则设计的系统将不会包含那些只有比其更复杂的系统才使用的代码。

对软件组件进行控制的一种方法是在运行时进行控制。这种方法不需要对组件进行预先配置,它所产生的程序代码相当大,可能包含了大量实际应用不需要的代码。桌面系统通常使

用这种方法(如动态链接库)。另一种方法是在链接时进行控制,这种方法只使用需要的组件函数,不需要的组件函数将被剔除掉。许多链接器(如 GNU 链接器)都支持这种功能。但是,链接时的控制只能在函数级进行控制,仍然难以满足嵌入式系统的需求。第三种方法是在编译时进行控制,这种方式可以使开发人员在最初阶段就实现对组件的控制,并且可以根据实际应用的需要对软件组件本身进行配置。编译阶段的控制方式可以在源码级实现对单条语句进行控制,而不是在函数级或目标代码级。采用编译阶段的控制方式可以有效地降低程序代码的大小,它非常适合于嵌入式系统。

eCos 主要使用编译阶段的控制方法来控制软件组件,同时也使用 GNU 链接器提供的链接时控制方法。编译阶段的控制方法也就是源码级的配置将由 C 预处理程序实现,所采用的主要机制就是 C 预处理表达式 # ifdef。下面是源码级配置的一个例子:

```
1 # ifdef CYGDBG _ USE _ ASSERTS
2 action1
3 . . .
4
5 # else
6 action2
7 . . .
8
9 # endif
```

例子中的 CYGDBG _ USE _ ASSERTS 标志(配置选项)的使能或禁止由开发人员进行控制,当对该段程序进行编译时,只有实际应用真正需要的代码才进入最后的映像中,action1 和 action2 不可能出现在同一个配置中。当系统中这样的配置选项较多时,选项之间可能会出现相互依赖关系,这就有可能导致冲突的出现。eCos 提供的配置工具可以很好地解决这种冲突问题。

在嵌入式软件开发中,采用源码级的配置方式除了可以减少代码量外,还具有以下一些好处:

⑴ 系统在运行阶段决定下一步采取什么行为时不需要对变量进行检查,因此运行速度将会更快。

⑵ 降低了系统延迟,响应速度将会更快,这在实时系统中尤为重要。

⑶ 简化了程序代码,使得调试和测试更加容易。

⑷ 根据实际应用需求对程序代码进行裁减,形成针对具体应用的专用 RTOS。

⑸ 资源的使用受到优化,处理器使用效率被提高,从而降低了硬件成本。

使用 eCos 提供的配置工具可以很方便地对软件组件进行选择和配置,它还可以调用 GNU 交叉编译工具对所选择的软件组件进行编译,形成最后与应用程序进行链接的库。eCos 配置工具具有命令行配置工具和图形配置工具两种形式,可运行在 Windows 和 Linux 平台上。

1.3 eCos 的组织结构

eCos 的一个主要设计目标是实现系统的高度可配置能力。为实现这一目标,它将系统分

成不同的软件组件。这些软件组件具有可重用性，根据目标硬件平台的实际需要，通过其独特的配置工具可以选择使用相应的组件，从而实现完整的嵌入式系统。对于每一个被选择的组件，还可以对它的各个选项进行更细致的配置，可以增加和删减组件的某些功能，也可以修改某些配置选项的值，对组件的配置完全可以根据实际应用的具体需求进行。使用这种方式最后产生可执行的 eCos 映像文件非常紧凑，只包含应用所需的一些功能。相对那些没有经过这种配置的其他嵌入式软件来说，由于不包含实际应用不需要的多余代码，eCos 具有更快的运行速度，稳定性也更好。

除了高度可配置能力外，eCos 所追求的还包括移植性和兼容性。它使用硬件抽象层的形式将上层软件与底层硬件进行隔离，这种特性可以很容易地实现 eCos 系统及其应用在不同平台和不同体系结构之间的移植。eCos 提供了一些标准库，并实现了与 μITRON 和 POSIX 标准的兼容，这种兼容性为第三方软件迅速移植到 eCos 系统提供了保障。

eCos 的可配置性、移植性和兼容性主要得力于它所具备的层次结构，这种层次结构还实现了它的易扩展性，新的组件可以很容易地加入到 eCos 中来。

1.3.1 eCos 的层次结构

eCos 采用模块化设计，将不同功能的软件分成不同的组件，这些组件具有可重用性，分别位于系统的不同层次。这种层次结构实现了 eCos 的可配置性、可移植性、兼容性和可扩展性。图 1-1 是 eCos 系统的层次结构图。

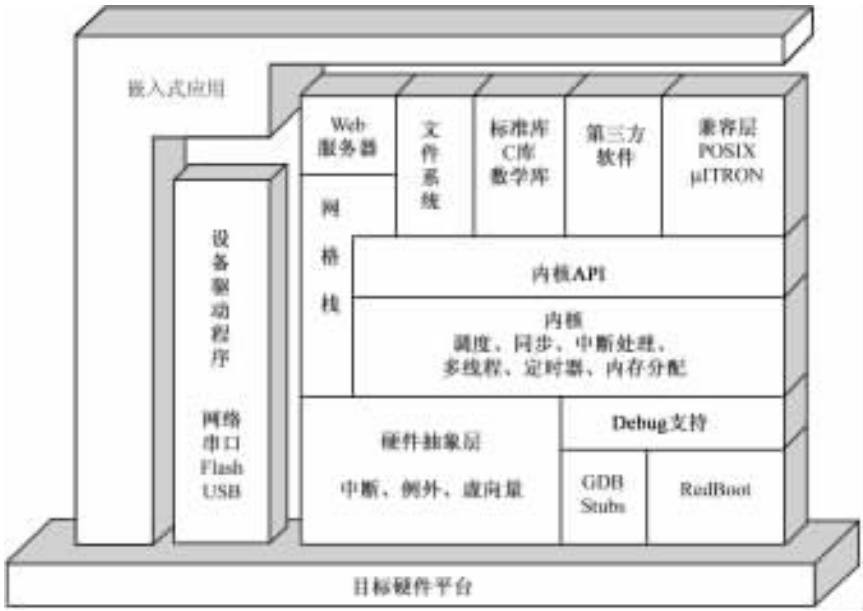


图 1-1 eCos 结构图

这种层次结构的最底层是硬件抽象层，它负责对目标系统硬件平台进行操作和控制，包括对中断和例外的处理，它为上层软件提供硬件操作接口。只需对硬件抽象层进行适当的修改就可以将整个 eCos 系统包括基于 eCos 的应用移植到新的硬件平台上。

RedBoot 是一个无内核的系统引导程序，它是 eCos 的一个特殊应用。RedBoot 可以加载

eCos 应用程序，并提供 Debug 支持，通过 RedBoot 还可以对目标系统环境参数进行管理。RedBoot 不仅可以引导 eCos 系统，也可以作为其他嵌入式系统的引导程序，Intel IQ80310 等许多厂家提供的开发板都使用了 RedBoot。RedBoot 具有一个用于 Debug 目的的 Stub 程序，为 GDB 工具的使用提供支持。

设备驱动程序负责对硬件设备进行控制和管理，并完成设备数据的读写操作。设备驱动程序模块自身也采用层次结构，上层驱动程序（相当于一个虚设备）可以调用下层驱动程序（物理设备），并且可以增加下层驱动程序未能提供的一些功能。驱动程序为上层软件提供标准的 API 函数，应用程序可以使用这些 API 函数对设备进行访问，完成对设备的初始化配置、获取配置信息以及数据传输等操作。

内核是 eCos 的一个核心组件，也是系统的一个可选组件，一些较为复杂的应用需要使用内核。内核提供了多个可供选择的调度器，它还具有一个多线程机制，支持多任务处理，同时还支持对称多处理器（SMP）系统。eCos 内核提供了一组丰富的同步原语，完全满足各种嵌入式应用的需求。内核还负责对中断和例外进行处理，它的中断滞后处理机制保证了系统的实时性。此外，内核还具有内存分配机制和定时机制，并提供多线程 GDB 调试支持。内核为上层软件和应用程序提供了丰富的 API 接口函数。

eCos 包含的网络支持包支持完整的 TCP/IP 网络协议栈，它提供了基于 OpenBSD 和 FreeBSD 的两种实现方式。eCos 目前支持的网络服务包括 FTP、TFTP、SNMP、DNS、HTTPD 等等。

在 eCos 内核和驱动程序之上的软件组件还包括标准库（ANSI C 库和数学库）、兼容层（POSIX 兼容和 μITRON 兼容）、文件系统（RAM 文件系统、FLASH 文件系统、ROM 文件系统）等。作为一种开放软件，eCos 还可以很方便地容纳第三方软件。

eCos 的这种层次结构不仅体现在构成它的各个软件模块上，而且还体现在其配置方法上。eCos 具有一个图形配置工具用于对系统的各个软件组件以及它们的配置选项进行配置。图 1-2 是其对 eCos 内核进行配置的一个片段。eCos 将软件的各种组成模块封装成一个个独立而完整的包，而这些包又可以进行分层配置。从图中可以看出这种从包（"eCos kernel"）到组件（"kernel schedulers"、"Exception handling"等）、从组件到配置选项（"Scheduler timesilicing"）、再从配置选项到子选项（"Number of clock ticks between timeslices"）的多个配置层次。

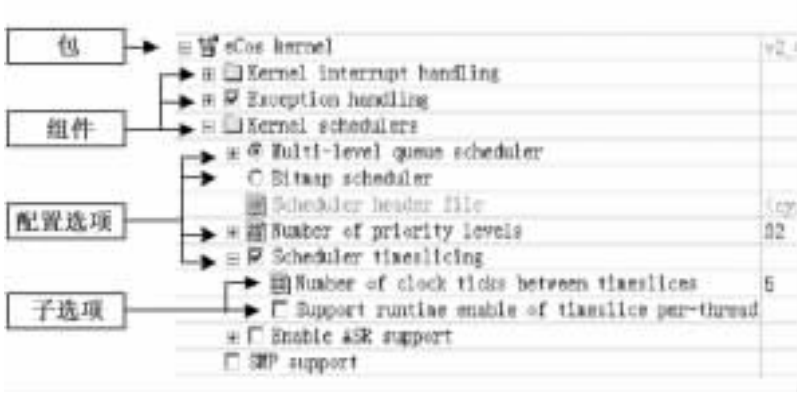


图 1-2　eCos 的配置层次

1.3.2 eCos 源码结构

eCos 源码所提供的内容包括构筑 eCos 系统的所有软件组件、测试程序源码、配置工具及源码、应用例子程序和说明文档等。其主要目录内容包括：

⑴ packages：包含 eCos 系统的所有软件包。

⑵ tools：包含 eCos 配置工具（图形配置工具和命令行配置工具），包括可执行文件和源码。

⑶ examples：包含一些从简单到复杂的应用程序例子。

⑷ doc：联机说明文档。

eCos 使用组件仓库（component repository）的形式对所有软件包和组件进行管理，packages 子目录就是包含所有这些软件包及组件的组件仓库。组件框架包含的管理工具可以用来对 eCos 源码进行管理，可以在组件仓库中增加新的软件包、更新当前使用的包以及删除旧的软件包。packages 子目录下包含一个数据库文件 ecos.db，由管理工具对它进行维护。该数据库文件包含了组件仓库内所有软件包的信息。

数据库文件 ecos.db 通常不需要修改，但将 eCos 移植到新的平台时需要对它进行编辑，以便配置工具可以识别并控制新的硬件抽象层 HAL。由于 eCos 还处于不断发展之中，任何时候都有可能增加新的平台支持以及增加新的组件，eCos 的组件仓库也因此处于不断变化之中。开发人员可以通过 eCos 的 CVS 服务器或其他方式来更新组件仓库内容。

在针对某一项目进行 eCos 开发时，配置工具将根据具体配置情况从组件仓库获取所需要的软件组件，并将具体配置保存为配置文件，形成编译树（build tree）目录和安装树（install tree）目录。编译树包含用于编译的 makefile 文件和编译时产生的中间文件，编译完成后将在安装树目录下产生 eCos 应用程序所需要的库文件。应用程序在编译时将与已经生成的库文件进行链接，并形成最终所需的可运行在目标系统上的可执行文件。

表 1-2 为 eCos 的组件仓库内容。

表 1-2 eCos 组件仓库内容

目　录	说　明
compat	POSIX（IEEE 1003.1）和 μITRON 3.0 兼容软件包
cygmon	CygMon 程序包
devs	设备驱动程序，不同硬件结构所支持的设备有所不同
error	错误和状态处理程序，用于所有包的错误和状态报告
fs	包含 RAM 文件系统、ROM 文件系统、JFFS2 Flash 文件系统
hal	硬件抽象层
infra	eCos 的基础结构包，内容包括有公共类型、宏、声称、启动选项等
io	基本 I/O 系统支持软件包，如文件 IO、PCI、以太网、Flash、串口、USB 等，是系统设备驱动程序的基础
isoinfra	ISO C 库和 POSIX 支持包
kernel	eCos 内核，包括调度器、同步原语、线程支持、中断和例外处理、时钟等
language	C 库和数学库

(续)

目　录	说　明
net	基本网络支持包,包括 TCP/IP、UDP、SNMP 等协议支持,还提供 DNS、FTP、HTTPD 支持
pkgconf	包的配置信息和编译规则 (makefile 生成规则)
redboot	RedBoot 软件包
services	包含内存分配、压缩算法、CPU 负载、电源管理、CRC 校验算法、Microwindows 图形窗口、实时性能测试等程序
templates	eCos 提供的各种模板,开发人员可以根据硬件平台的实际情况选择相应的模板 (见表 1-3)

1.4　eCos 的一些基本概念

eCos 一开始就被设计成一个具有可配置性的组件结构。它的核心部分由许多不同组件组成,如内核、C 库、基础结构包 (infrastructure package) 等。每一个组件又提供了大量的配置选项,允许开发人员根据开发项目的需要进行不同的配置。为了对繁多而复杂的组件及其众多的配置选项进行管理,eCos 提供了一个组件框架,这是专门用于支持对多个组件进行配置的一组工具。组件框架工具具有可扩展性,可以在任何时候增加其他组件。这一节主要介绍 eCos 的一些基本概念,在研究和开发 eCos 之前应该对这些基本概念有所了解。

1.4.1　组件框架

组件框架 (Component Framework) 是用于对目标系统进行配置以及对组件仓库进行管理的一组工具。包括 ecosconfig 命令行配置工具、图形配置工具、组件包管理工具。命令行配置工具和图形配置工具都建立在一个用于对组件进行描述的 CDL 库的基础之上。

1.4.2　配置选项

配置选项 (Configuration Options) 是进行配置的基本单位。每一个选项都对应于用户的一个选择。例如,有一个选项用于控制是否进行声称和使能,内核有一个选项用于设置系统调度优先级数目。选项可以对一些代码很小的函数进行控制,例如选择是否内嵌 C 库函数 strtok。也可以对一些代码相当大的函数进行控制,例如对 printf 是否支持浮点转换的选择。

大多数选项都比较简单,用户只需选择使能或禁止。有些选项却较为复杂,例如调度优先级数目必须在指定范围内进行选择。每个选项都具有一个合理的默认值,在开始开发一个系统时不必对所有选项都进行设置和选择。开发人员可以在基本系统正常运行之后,再根据实际应用的特殊性进行相应的调整。

有些选项用户是不可以修改的,如处理器的字节排列方式。一些处理器只能采用单一的 big-endian 或 little-endian 方式,而有些处理器则可以对此进行选择。根据目标系统硬件的不同,字节排列方式选项有时是可以修改的,有时是不能修改的。

1.4.3　组件

组件 (Component) 是一个功能单位,如一个特殊的内核调度器、一个指定设备的设备驱动

程序等等。组件也是一个配置选项,用户可以对组件的所有功能的使能和禁止进行选择。例如,如果目标系统中的某个设备在实际应用中不被使用,就不需要该设备的驱动程序。禁止该设备的驱动程序可以降低程序代码和数据对内存的需求。

组件可以包含更多的配置选项。例如,一个设备驱动程序可能具有某些选项对它的功能进行控制。当然如果整个设备驱动程序被禁止,这些选项就没有任何意义了。一般来说,大多数的选项和组件具有一种层次结构,任何组件都可以包含一些该组件的选项和一些子组件。可以把 eCos 内核看作是一个大的组件,它包含一些如调度器、例外处理、同步元素等等这样一些子组件,而同步元素组件又包含更多的子组件,如互斥、信号、条件变量、事件标志等等。互斥组件又包含一些配置选项,如是否支持优先级倒置等。

1.4.4 包

包 (Packages) 是一种特殊类型的组件,它是组件的发布单位。对于一个包含了所有源代码、头文件、说明文档和其他一些相关文件的包,可以形成一个发布文件。使用适当的管理工具可以对该发布文件进行安装,也可以卸载该包,或更新包的版本。eCos 源码已经提供了许多包,如内核和基础结构包等。其他第三方软件包可以通过不同途径获得,并且可以与 eCos 安装在一起。

在进行 eCos 配置时,包可以被使能或禁止。使能一个包时,它的配置数据将被加载,而禁止一个包则意味着对该包进行卸载操作。除了可以禁止和使能外,还可以对包的版本进行选择。

一般来说,包所包含的内容如下:

⑴ 用于建库的源文件。应用程序将与该库进行链接,形成可执行文件。某些源文件可能用于其他目的,如提供链接脚本。

⑵ 用于对包的接口进行定义的头文件。

⑶ 在线说明。

⑷ 用于测试的源码程序。

⑸ 描述该包的一个或多个 CDL 脚本。

并不是所有的包都必须包含上述全部内容。举例来说,设备驱动程序可以不提供新的包接口。但是,所有的包都必须具有对其进行描述的 CDL 脚本。

1.4.5 组件仓库

eCos 具有一个组件仓库 (Component Repository),它是一个目录结构,所有的包都被安装在这里。组件框架提供了一个组件管理工具用于对包进行安装和删除等操作。组件仓库包含一个简单的数据库,该数据库包含了每个包的详细信息,用管理工具进行维护。

除了可以用管理工具对组件仓库进行操作外,用户进行开发时一般不需要对组件仓库进行修改,eCos 将用户的开发工作与组件仓库分别放在不同的目录下。可以把组件仓库当作只读资源,多个用户或多个设计项目可以共享组件仓库。

1.4.6 配置

配置 (Configuration) 是用户所做选择的一个集合。组件框架的各种工具负责对整个配置

进行处理，使用这些工具可以产生新的配置并保存配置（默认为 ecos.ecc）。还可以对配置进行操作，在编译 eCos 和其他软件包之前使用配置来产生一个编译树。配置包含了哪些包被选择以及这些包中的哪些选项被用户使能或禁止等详细信息。

1.4.7 目标系统

目标系统（Target）是指将要运行 eCos 应用程序的硬件平台。它可以是真实的一个硬件平台，也可以是一个模拟器。在创建一个新的配置的时候，需要指定相应的目标系统。组件框架将根据指定的目标系统选择一组包来产生相应的配置，如硬件抽象层 HAL 和设备驱动程序等等。此外，它还有可能根据指定目标系统的具体特点来对某些选项进行设置。

1.4.8 模板

模板（Templates）是一种局部性配置，其目的是为用户的开发工作提供一个适当的起点。eCos 源码中提供了一些模板。其中有一个只提供很小功能的模板（RedBoot），它只为硬件提供一个引导程序并直接跳转到应用程序。默认模板（default）在此基础上增加了其他一些功能，如内核和 C 库等。Uitron 模板以 μITRON 兼容层的形式增加了更多的功能。创建一个新的配置时，通常要指定模板和目标系统，应用程序在这种配置下进行编译和链接，并最终在硬件平台上运行。使用配置工具可以对配置选项进行适当的设置，使其可以更好地满足应用需求。表 1-3 为 eCos 2.0 所提供的模板。

表 1-3 eCos 模板

模板名称	描　述
all	包含所有的软件包
cygmon	用于生成 CygMon 的配置
cygmon_no_kernel	用于生成 CygMon 的配置，不包括内核
default	eCos 的默认配置，包含基础结构包、内核、C 和数学库，以及其他软件包
kernel	包含 HAL、基础结构包、内核
minimal	eCos 的最小配置，只包含 HAL 和基础结构包
net	提供网络支持，包含 TCP/IP 协议栈（FreeBSD）
old_net	提供网络支持，包含 TCP/IP 协议栈（OpenBSD）
posix	包含 HAL、基础结构包、内核、POSIX 包
redboot	建立 RedBoot 环境时的配置，生成 RedBoot 映像文件
stubs	eCos 的 stubs 配置，用于生成 eCos 的 GDB stubs 程序
uitron	与 μITRON 3.02 完全兼容的配置

1.4.9 属性

对于每一个配置选项，组件框架都需要了解它的一些相关信息。例如，它需要知道什么样的选项值是合法的、选项的默认值是什么、当用户需要对其进行选择时相应的联机说明文档位

置在哪里,等等。每一个选项(包括组件和包)都由选项名字和选项的一组属性(Properties)组成。

1.4.10 约束条件

配置选项的选择并不是独立的,一个配置选项可能会受到其他配置选项的限制,并可能影响到其他的配置选项。例如,只有在内核对每线程数据(pre-thread data)提供支持的条件下,C 库才会提供一些像 rand 这样具有线程安全性(C 库选项 thread-safety)的函数。这就存在一个约束条件(Constraints):C 库选项对内核有要求。一个典型的配置包含数量相当可观而且比较复杂的约束条件。许多约束条件比较直观,如:选项 A 要求有选项 B,或者选项 C 排斥选项 D。而有些约束条件却较为复杂,例如:选项 E 可能要求具有内核调度器,但它并不关心具体选择的是哪一个调度器。

另外一种类型的约束条件涉及到选项的具体值。例如,一个与调度优先级数目相关的内核选项就存在这样一个合法值的约束条件:调度优先级数目为 0 或负值是非法的。

1.4.11 冲突

用户对配置选项进行操作时可能会出现非法配置,可能有一个或多个约束条件得不到满足。例如,如果内核中的"pre-thread data"选项被禁止,而 C 库中的"thread-safety"选项仍然是使能的,这样就存在一个没有被满足的约束条件,这就是冲突(Conflicts)。配置工具将会报告这样的冲突。冲突的出现并不妨碍 eCos 的编译,但其结果难以预料,有可能在编译时报错,也有可能在链接时报错,或者应用程序根本就不能运行,或者应用程序虽然能运行但在某些时候将会出现错误。在进行 eCos 配置的时候,如果配置工具报告有冲突发生,在继续进行配置之前通常要先解决这些冲突。

为使用户操作更容易,配置工具提供了一个推理机。该推理机能够自动检测到当前配置中所存在的冲突,并能指出解决这些冲突的方法。推理机既可以自动解决这些冲突,也可以在人工干预下进行解决。

1.4.12 组件定义语言 CDL

配置工具需要由每个包提供各种选项信息,如结果信息、约束条件、在线说明文档位置等。这些信息是以 CDL(组件定义语言 Component Definition Language)脚本的形式提供的。CDL 主要用于对配置选项进行描述。

第 2 章　eCos 开发环境的建立

eCos 作为一种开放源代码，它的开发工具和源码都是免费提供的。在学习和开发 eCos 系统时，所需的开发条件非常简单，读者利用现有的学习和工作条件可以很方便地建立一个完整的 eCos 开发环境。所有的开发工具和 eCos 源代码都可以从 Internet 上下载，也可以直接使用本书所附光盘中提供的开发工具和源码。

本章首先介绍了建立 eCos 开发环境所必备的一些工具，并具体讲述了如何利用这些工具建立一个完整的 eCos 开发环境。读者可以通过本章循序渐进地学习如何建立 eCos 开发环境，在此基础上再进行后续章节的学习。

2.1　系统需求

学习和开发 eCos 系统时，首先要准备所需的开发环境。eCos 开发环境对系统的要求不高，最简单的配置只要一台微机就可以满足需要。因此对于 eCos 爱好者来说，这是极为简捷方便的。

2.1.1　主机系统需求

一般来说，eCos 开发环境对主机系统有下述要求：

1）标准 Intel 结构的 PC 机，运行 Winsows XP、Windows 2000 或 Windows NT 4.0 (SP3) 操作系统，也可以使用 RedHat Linux 6.0 以上版本的操作系统，或其他 Linux 操作系统。

2）64MB 以上内存，300MHz 以上的奔腾处理器。

3）Windows NT 必须安装 4.0 以上版本的 IE。

4）也可以使用基于 SPARC 处理器的 Sun 工作站，运行 Solaris2.5.1 以上版本的操作系统。

5）足够的硬盘空间。eCos 的所有源码以及它的开发过程所产生的所有文件和目录实际所需容量不到 200MB，但交叉编译工具 (GCC 等) 的编译还需要一定的硬盘空间。建议为 eCos 保留 2GB 以上的空间。

6）一个 RS232 串口。

7）网卡，可选。

8）如果希望能及时下载最新版本的 eCos 源码或其他开发工具，则应该与 Internet 相连。

2.1.2　目标系统需求

除了对主机系统的这些要求外，对运行 eCos 的目标系统也有一些要求。根据开发板的不同需要，还需要一些相应的配置。一般需要目标系统提供下述支持：

1）串口。用于与主机通信，通过 GDB 调试工具进行 eCos 的加载，对目标系统软件进行调试。如果目标系统没有串口，则必须使用网卡与主机系统进行连接和通信，这时主机系统必

须具有网卡。

2）网卡。不同目标系统对网卡的支持不一样。如果 eCos 源码不支持所使用的网卡，则必须自行开发网卡驱动程序。网卡的支持是可选的，如果所开发的 eCos 系统不需要网络支持，则不需要网卡，但必须提供串口与主机系统进行通信。

3）开发板一般需要带有 GDB stub 小程序（用于与主机系统的 GDB 建立通信连接）的 ROM 或 Flash ROM，或者带有 RedBoot、CygMon 的 ROM 或 Flash ROM。

eCos 源码支持多种处理器，并对全球范围内许多知名厂家提供的多种典型嵌入式系统开发板（评估板）提供支持。对于不同的开发板，eCos 对其要求也可能会有所不同，读者可以从 eCos 网站上了解其详细内容。即使 eCos 不支持某种新的目标平台，也可以将 eCos 移植到新的平台上。

在 eCos 的学习阶段，可以直接使用基于 x86 的标准 PC 主板作为目标系统开发平台，eCos 配置工具使用“i386 pc target”来表示这种硬件平台。对于这种目标系统的要求是：

1）带有 i386 或更好处理器的 PC 主板。

2）3.5in 软驱（用于引导 RedBoot）。

3）显卡与显示器。

4）标准键盘。

5）如果使用串口进行调试，则需要使用简单的三线交叉串口电缆将其与主机系统串口相连。

6）如果使用网络进行调试，则主机需要一块网卡，通过网线与目标系统相连。目标平台需要一块基于 Intel i82559 的 PCI 以太网卡。

作为 eCos 开发环境的一个例子，笔者所使用的典型配置是：

1）标准 PC 主板（具有串口、网卡）。

2）2.4GHz 奔腾 4 处理器。

3）256MB 内存。

4）60GB 硬盘。

5）Windows 2000 操作系统、RedHat Linux 8.0 操作系统。

6）目标系统：标准 PC 主板、奔腾 2 处理器、RS232 串口、基于 Intel i82559 的 PCI 网卡 Intel EtherExpress Pro 10/100，软驱。

使用基于 x86 标准主板的原因是易于进行 eCos 的学习和开发。本书将以此为基础介绍 eCos 系统的详细开发过程。

2.2 开发工具

eCos 具有两种可选择的开发环境：一种是建立在 Windows 下的开发环境；另一种是基于 Linux 或 UNIX 下的开发环境。本篇重点讲述 Windows 环境下所需的开发工具。Linux 与 UNIX 的开发环境类似，这里只简单介绍 Linux 环境下所需的开发工具。

在 Windows 环境下（包括 Windows XP、Windows 2000 和 Windows NT），eCos 开发环境的建立需要三种工具和环境，分别是 Cygwin、GNU 交叉编译工具和 eCos 配置工具。Linux 环境下只需要 GNU 交叉编译工具和 eCos 配置工具。

2.2.1 Cygwin

eCos 是一种开放源代码的软件，它的开发环境所依赖的基础是 Linux。在 Windows 环境下，开发 eCos 系统需要使用 Linux 模拟环境 Cygwin。Cygwin 是运行在 Windows 平台上的一个 UNIX/Linux 模拟环境。它由两部分组成：

(1) 动态链接库 Cygwin1.dll。它是 Windows 环境下的 UNIX/Linux 模拟层，提供标准的 UNIX/Linux API 函数。

(2) 一组 UNIX/Linux 工具集。使得用户可以在 Windows 环境下使用 UNIX/Linux 环境所提供的各种工具。

Cygwin 提供的一组功能强大的工具可以帮助开发人员将应用程序从 UNIX/Linux 移植到 Windows 平台上，它提供了标准自由软件 GNU GCC 编译器和 GDB 调试工具。除此之外，它还为 Windows 提供一个包括 API 和命令 SHELL 在内的标准 UNIX/Linux 开发环境。通过使用 Cygwin，开发人员可以对不同的系统环境进行一致而有效的管理。

Cygwin 可以从其网站 http://www.cygwin.com 上下载并安装最新版本 (网络安装)，也可以直接使用本书光盘提供的 Cygwin (本地安装)。

2.2.2 交叉编译工具

在进行 eCos 开发时，在开发主机上需要使用交叉编译工具对 eCos 系统和应用程序进行编译，生成可在目标系统上运行的执行代码。eCos 所使用的是开放源代码 GNU 编译工具，包括三个部分：GNU 编译器 GCC、GNU 调试工具 GDB 和包括 GNU 汇编器和链接器在内的 GNU 二进制工具 (GNU Binary Utility)。

这些交叉编译工具源码可以从 GNU 网站上下载：http://gcc.gnu.org/。

(1) GCC。GCC 核心编译器和 C++ 编译器，2.95.2 以上版本。

(2) Newlib。为 C++ 运行库的编译提供支持，可以使用最新的 Newlib 版本。

(3) GDB。Insight 或 GDB，5.0 以上版本。Insight 是 GDB 的扩充，它是一个具有图形界面的 GDB。在使用 Insight 的时候，仍然可以使用“-nw”选项以命令行的形式运行 GDB。

(4) GNU 二进制工具。binutils 2.10.1 以上版本。

本书光盘提供下述版本的 GNU 编译工具：

① gcc-core-3.2.1.tar.gz。

② gcc-g++-3.2.1.tar.gz。

③ newlib-1.11.0.tar.gz。

④ insight-5.3.tar.gz。

⑤ gdb-5.3.tar.gz。

⑥ binutils-2.13.2.1.tar.gz。

另外还提供相应的补丁程序：

① binutils-2.13.2-2.13.2.1.patch。

② gcc-3.2.1-arm-multilib.patch。

③ insight-5.3-tcl_win_encoding.patch。

2.2.3 eCos 配置工具

eCos 是一个可配置的嵌入式操作系统，根据实际目标系统的具体情况，可以使用 eCos 配置工具对系统进行裁减，从而在满足系统需求的条件下实现代码量的最小化。

eCos 配置工具包括图形配置工具和命令行配置工具，通常使用图形配置工具。在 Windows 和 Linux 下都可以使用 eCos 图形配置工具。命令行配置工具可以在 Windows 的 Cygwin 环境下运行，也可以在 Linux 下直接使用。

eCos 的配置工具可以直接从其网站上下载。也可以使用其源码中提供的配置工具，这些工具位于 ecos-2.0/tools/bin 目录内。目前较为常用的版本是 eCos Configuration Tool 2.11。命令行配置工具目前常用的版本是 ecosconfig 1314。

eCos 配置工具的下载地址：http://sources.redhat.com/ecos/。

2.3 Cygwin 的安装与设置

在准备好上述工具之后，就可以进行安装了。如果是在 Windows 环境，首先必须安装 Cygwin。如果是 Linux 环境，则不需要使用 Cygwin。安装工作可以选择网络安装（通过 http://www.cygwin.com）和本地安装（使用本书光盘）。下面介绍具体的安装过程：

1）运行 Cygwin 安装程序 setup.exe（光盘目录 Cygwin 内）。如果是网络安装，选择“Install from Internet”，如果是从光盘安装，则选择“Insatll from Local Directory”，选择“下一步”，如图 2-1 所示。

图 2-1 Cygwin 安装

2）选择 Cygwin 的安装根目录，并选择 DOS 文本文件类型。选择“下一步”，如图 2-2 所示。

3）选择软件包、下载存放目录。在使用网络安装时，首先要从 Cygwin 服务器下载即将安装的软件包，这些软件包将存放在指定的位置。如果指定目录不存在，则自动创建指定目录。如果是本地安装，则将其指定为本地 Cygwin 存放目录（如：光盘目录/cygwin/cygwin）。选择

“下一步”,如图 2-3 所示。

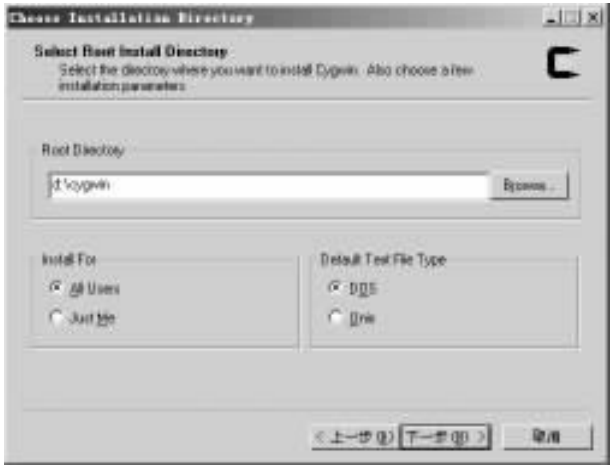


图 2-2　选择安装目录

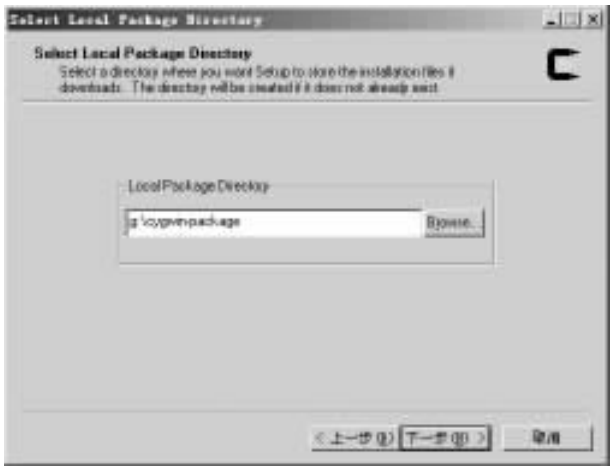


图 2-3　下载软件包存放目录

4）进行 Internet 设置。如果使用网络安装,则必须对 Internet 的连接方式进行指定 如果通过代理服务器与 Internet 相连,则选择“Use HTTP/FTP Proxy”,并指定代理服务器名字和端口 如果是直接与 Internet 相连,则选择“Direct Connection”;也可以使用 IE 的 Internet 设置。选择“下一步”,如图 2-4 所示。如果是本地安装,则可跳过这一步。

5）选择安装软件包。根据需要在图 2-5 所示的对话框中选择所要安装的软件包。如果硬盘容量足够大,可以选择安装所有的软件包以及相应的源代码。对于 eCos 来说,不需要选择所有的软件包,也不需要这些软件包的源代码。但仅仅使用安装程序的默认选择是不够的,需要仔细为 eCos 开发环境选择软件包。在选择完成后,点击“下一步”将开始软件包的下载和安装工作。除了安装默认的软件包外,还必须安装 gcc, make, sharutils, tcltk, wget 等软件包。

一般来说,应该选择下述软件包：

图 2-4　Internet **连接设置**

图 2-5　Cygwin **软件包的选择**

① Archive **:全选。**
② Base **:全选。**
③ Devel **:全选。**
④ Doc **:默认选择。**
⑤ Editors **:全选。**
⑥ Graphics **:默认选择。**
⑦ Interpreters **:全选。**
⑧ Libs **:全选。**
⑨ Mail **:不选。**
⑩ Math **:不选。**
⑪ Net **:不选。**
⑫ Publishing **:选择。**
⑬ Shells **:全选。**

⑭ System 默认选择。

⑮ 选择 expat、libkpathsea3、libxml2、libxslt、texinfo。

⑯ utils 全选。

⑰ web 不选。

⑱ _ PostInstallLast 选择。

软件包安装完成后，根据提示进行其余的操作，有些系统需要重新启动。此后就可以运行 Cygwin 了。

6) 为 eCos 设置 Cygwin 安装点。eCos 使用“/c”、“/d”等表示硬盘“c:”和“d:”，因此必须进行下面的操作：

① 启动 Cygwin，进入 Cygwin 的 Shell 界面，执行下述命令：

```
$ mount -f c:/c
$ mount -f d:/d
$ mount -f e:/e
```

② 有多少个硬盘分区就进行多少次这样的操作，所有的盘符都应该进行这种 mount 操作。此时，可以使用下述命令进入相应的硬盘：

```
$ cd /c (进入 C:\ 盘)
$ cd /d (进入 D:\ 盘)
```

③ 为保证能正确运行图形配置工具，应该在系统环境变量中使路径名包括 cygwin/bin 目录。

2.4 GNU 交叉编译工具的编译与配置

在开发 eCos 系统时，要根据不同体系结构的 CPU 选择不同的编译工具。eCos 网站上提供了一些已编译好的交叉编译工具，可以直接使用。如果没有所需要的工具，则需要自行编译。GNU 提供了交叉编译工具的源代码，必须对它们进行编译来生成所需要的编译工具。从 GNU 网站上下载源代码（或者直接使用本书光盘提供的源代码）后，可以将它们解压缩到一个 src 源目录。必须注意的是应该有足够的硬盘空间（大约 6 倍于源码压缩文件的大小）用于存放源码文件和中间文件。为方便读者，本书光盘中提供了一些已编译好的 Cygwin 环境下交叉编译工具（提供 GDB 图形界面）。

下面以 Intel x86 为例介绍交叉编译工具的编译过程。Windows 环境和 Linux 环境下的编译过程基本相似。

在 Windows 环境下，首先要运行 Cygwin，在 Cygwin 环境下进行工具的编译。Linux 环境下可以在命令行窗口下进行编译。其步骤如下：

1) 创建目录/src：

```
$ mkdir -p /src
$ cd /src
```

将下述源码文件复制到该目录下：

- gcc-core-3.2.1.tar.bz2
- gcc-g++-3.2.1.tar.bz2
- newlib-1.11.0.tar.gz
- insight-5.3.tar.bz2
- gdb-5.3.tar.bz2
- binutils-2.13.2.1.tar.bz2
- binutils-2.13.2-2.13.2.1.patch
- gcc-3.2.1-arm-multilib.patch
- insight-5.3-tcl_win_encoding.patch

其中 gdb **和** insight **可以根据是否使用图形界面** GDB **选择其中的一个,下面的步骤使用的是** insight**。**

2)对压缩文件解压缩。

如果是 bzip2 **压缩文件,使用下述命令解压缩:**

```
$ bunzip2 < binutils-2.13.2.1.tar.bz2 | tar xvf -
$ bunzip2 < gcc-core-3.2.1.tar.bz2 | tar xvf -
$ bunzip2 < gcc-g++-3.2.1.tar.bz2 | tar xvf -
$ bunzip2 < insight-5.3.tar.bz2 | tar xvf -
$ bunzip2 < newlib-1.11.0.tar.bz2 | tar xvf -
```

如果是 gzip **压缩文件,则使用下述命令解压:**

```
$ gunzip < binutils-2.13.2.1.tar.gz | tar xvf -
$ gunzip < gcc-core-3.2.1.tar.gz | tar xvf -
$ gunzip < gcc-g++-3.2.1.tar.gz | tar xvf -
$ gunzip < insight-5.3.tar.gz | tar xvf -
$ gunzip < newlib-1.11.0.tar.gz | tar xvf -
```

将补丁程序加入到源码文件中:

```
$ patch -p0 < binutils-2.13.2-2.13.2.1.patch
$ patch -p0 < gcc-3.2.1-arm-multilib.patch
$ patch -p0 < insight-5.3-tcl_win_encoding.patch
```

如果源码中已经包含了这些补丁,将出现下面的信息:

```
Reversed (or previously applied) patch detected! Assume -R? [n]
```

此时直接按<n>和回车键即可。

至此,产生了下述源码目录:

```
/src/binutils-2.13.2.1
/src/gcc-3.2.1
/src/insight-5.3
/src/newlib-1.11.0
```

如果使用 GDB (gdb-5.3.tar.gz 或 gdb-5.3.tar.bz2),而不是使用 Insight,则产生目录/src/gdb-5.3。

然后再将 newlib-1.11.0 的内容移至 gcc-3.2.1 目录下:

```
$ mv newlib-1.11.0/newlib gcc-3.2.1
$ mv newlib-1.11.0/libgloss gcc-3.2.1
```

3) 验证环境变量 PATH 是否包含了 Cygwin 的编译工具 (如 gcc、make 等) 的路径名。注意路径名不能以“.”开头。

后面步骤所进行的编译过程将使用临时编译目录/tmp/build 和编译产生的工具最后安装目录/gnutools,也可以指定其他目录。编译时需要较大的磁盘空间,/tmp/build 目录在编译完成后可以将其删除。

需要注意的是,这里介绍的是以 i386 为目标系统的编译过程,使用 i386-elf 来表示。如果是针对其他体系结构的目标系统,则应该替换下述步骤中的 i386-elf。表 2-1 为不同体系结构的表示方式。

表 2-1 不同体系结构的表示方式

体系结构	表示方式
ARM (包括 StrongARM 和 XScale)	arm-elf
Intel x86	i386-elf
Matsushita AM3x	mn10300-elf
Motorola 68K/ColdFire	m68k-elf
MIPS32	mipsisa32-elf
NEC V850	v850-elf
PowerPC	powerpc-eabi
Renesas H8/300H	h8300-elf
Renesas SuperH	sh-elf
Toshiba TX39	mips-tx39-elf
Toshiba TX49	mips-tx49-elf

4) 编译和安装 GNU Binary Utilities。

首先使用下述命令对 GNU Binary Utilities 的编译进行配置:

```
$ mkdir -p /tmp/build/binutils
$ cd /tmp/build/binutils
$ /src/binutils/binutils-2.13.2.1/configure --target=i386-elf \
        --prefix=/gnutools -v 2>&1 | tee configure.out
```

如果配置过程出现错误,可以查看 configure.out 文件中的详细信息。

配置成功后,使用下面的命令进行 GNU Binary Utilities 的编译,并将其安装到目录/gnutools 下:

```
$ make -w all install 2>&1 | tee make.out
```

详细编译信息存放在 make.out 文件内，如果编译出现错误，可以查看该文件。编译产生的二进制应用程序位于/gnutools/bin 目录下。

5）编译和安装 GCC。

在编译 GCC 之前，应该将前面生成的 GNU Binary Utilities 工具路径名包含在环境变量 PATH 内：

```
$ PATH=/gnutools/bin:$ PATH ;export PATH
```

然后再对 GCC 进行配置，命令如下：

```
$ mkdir -p /tmp/build/gcc
$ cd /tmp/build/gcc
$ /src/gcc/gcc-3.2.1/configure --target=i386-elf \
        --prefix=/gnutools -enable-languages=c,c++ \
        --with-gnu-as --with-gnu-ld --with-newlib \
        --with-gxx-include-dir=/gnutools/i386-elf/include \
        -v 2>&1 | tee configure.out
```

如果配置过程出现错误，可以查看 configure.out 文件中详细信息。

配置成功后，使用下面的命令进行 GCC 的编译，并将其安装到目录/gnutools 下：

```
$ make -w all install 2>&1 | tee make.out
```

详细编译信息存放在 make.out 文件内，如果编译出现错误，可以查看该文件。

6）编译 Insight（或 GDB）。

首先对 Insight 的编译进行配置：

```
$ mkdir -p /tmp/build/gdb
$ cd /tmp/build/gdb
$ /src/gdb/insight-5.3/configure --target=i386-elf \
        --prefix=/gnutools -v 2>&1 | tee configure.out
```

如果有错误，则可以查看 configure.out 文件存放的信息。如果配置成功，则使用下面的命令对其进行编译与安装：

```
$ make -w all install 2>&1 | tee make.out
```

详细编译信息存放在 make.out 文件内，如果编译出现错误，可以查看该文件。

至此，基于 Intel x86 的开发工具全部编译完毕，所有的这些工具存放的目录是/gnutools/bin。可以将该目录添加到环境变量 PATH 上。使用 eCos 配置工具也可以设置这些工具的路径名。其他体系结构的交叉编译工具的编译过程与此相似。

2.5 eCos 源码与配置工具的安装

eCos 源代码和配置工具都可以从 eCos 网站上下载（http://sources.redhat.com/ecos）。本

书光盘提供了 eCos 2.0 的所有源代码和配置工具，用户可以直接使用。

2.5.1 eCos 的安装

有两种方法可以从 Internet 安装 eCos：一种方法是直接执行 eCos 安装程序；另一种方法是通过 FTP 或 HTTP 获取每一个单独的软件包分别进行安装。另外也可以直接使用本书光盘附带的 eCos 源代码，可参照网络安装的方式将其解压缩到相应的目录就可以使用。这里主要介绍如何通过 Internet 安装 eCos。

1. 通过 Internet 安装 eCos

这种安装方法可以一次性地安装所有的 eCos 源码和相应的 GNU 交叉编译工具。由于代码量大，因此要求有较高的网络速度。

1）进入 Cygwin，在 Cygwin 环境下运行下面的 wget 命令获取 eCos 安装文件 ecos-install.tcl：

```
$ wget --passive-ftp ftp://sources.redhat.com/pub/ecos/ecos-install.tcl
```

该命令执行时将连接到 eCos FTP 服务器 sources.redhat.com，下载文件 ecos-install.tcl，其过程如图 2-6 所示。

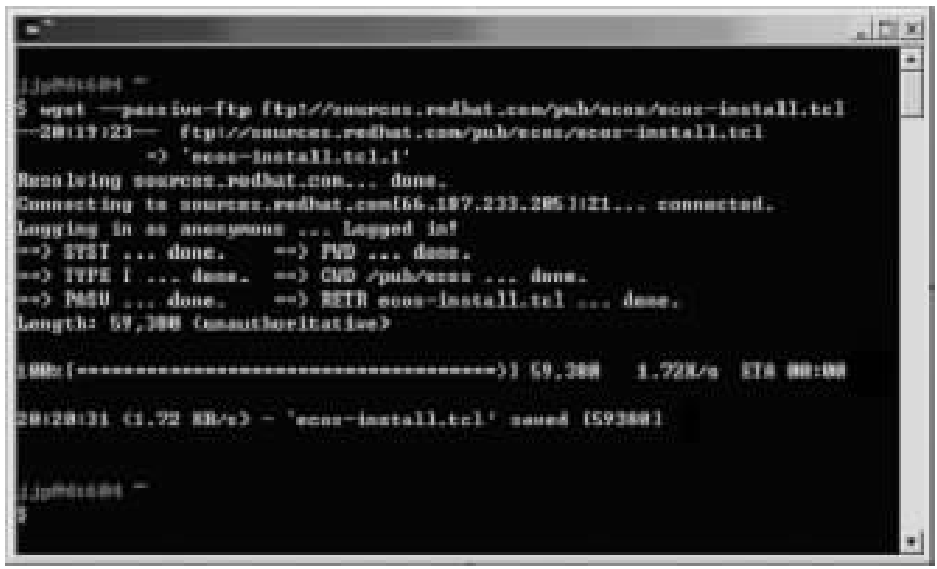


图 2-6 eCos 的网络安装

2）运行安装文件 ecos-install.tcl，开始 eCos 和编译工具的下载与安装。命令格式如下：

```
$ sh ecos-install.tcl
```

该命令执行时首先下载安装数据文件 ecos-install.db，然后出现一个可供下载 eCos 的镜像服务器列表，从列表中选择一个速度最快的服务器进行下载和安装。

3）指定安装目录。在选择服务器后，将提示输入 eCos 的安装目录。默认目录是/opt/ecos。

4）选择交叉编译工具。在指定安装目录后，安装程序将出现一个选择交叉编译工具的列

表,根据实际需要选择相应的工具。这些工具是已经编译好的可以直接使用的可执行文件,如果不需要这些编译工具,则可以不选择。

5) 安装程序自动下载并安装 eCos 源码和编译工具。安装完成后,在安装目录下将产生 shell 文件 ecosenv.sh,运行该文件对 eCos 的环境进行设置。例如:

```
$ ./opt/ecos/ecosenv.sh
```

可以在 shell 启动文件 (如 $HOME/.profile) 的末尾增加此行。

至此,在安装目录下将出现两个子目录:一个是 eCos 源码目录 ecos-2.0;另一个是交叉编译工具目录 gnutools。源码目录 (ecos-2.0/tools/bin) 中包含了 eCos 配置工具,可以在桌面上设置其快捷方式。

上面介绍的是在 Windows 环境下的安装方法,Linux 环境下的安装与此基本类似。

2. 使用单个软件包进行安装

可以从 eCos 网站上下载单个软件包,或者使用本书光盘提供的软件包,这些软件包为:

① ecos-2.0.cygwin.tar.bz2 (eCos 源码,包含配置工具和联机说明)。

② ecoscentric-gnutools-i386-elf-1.4-2.cygwin.tar.bz2 (不同的目标系统选用不同的编译工具,i386-elf 指的是 i386 系统)。

如果是在 Linux 下,则相应的软件包为:

① ecos-2.0.i386linux.tar.bz2。

② ecoscentric-gnutools-i386-elf-1.4-2.linux.tar.bz2。

交叉编译工具也可以使用自行编译的 GNU 工具。

得到上述软件包后,将它们在指定的安装目录下进行解压缩操作,将分别产生子目录 ecos-2.0 和 gnutools。

接下来的操作是设置环境变量。主要有两个环境变量:一个是组件仓库的位置 ECOS_REPOSITRY,另一个是编译工具的路径。假设安装目录是/opt/ecos,组件仓库的位置将是/opt/ecos/ecos-2.0/packages,编译工具路径为/opt/ecos/gnutools/bin。如果是在 Windows 环境下,也可以在系统环境变量中设置这两个参数,还可以在图形配置工具中对它们进行设置。如果使用 Cygwin 命令行配置工具,或者是在 Linux 环境下,则可以使用下面的命令对它们进行设置:

对于 sh、ksh、bash 用户,使用下述命令进行设置:

```
$ ECOS_REPOSITORY=/opt/ecos/ecos-2.0/packages ;export ECOS_REPOSITORY
$ PATH=$PATH:/opt/ecos/gnutools/bin ;export PATH
```

对于 csh、tcsh 用户,使用下面的命令进行设置:

```
$ setenv ECOS_REPOSITORY /opt/ecos/ecos-2.0/packages
$ setenv PATH $PATH:/opt/ecos/gnutools/bin
```

3. 使用 eCos 的 CVS 服务器

eCos 的最新源代码可以通过访问 eCos 匿名 CVS 服务器获取。在访问 CVS 服务器之前,必须安装有 CVS 客户端程序,并且必须直接与 Internet 相连。对于位于防火墙内的用户,必须具有相应的访问权限。在安装 Cygwin 时,如果选择了 CVS,则可以在 Cygwin 环境下直接

使用 CVS 客户端程序。在 Linux 环境下,也可以直接使用 Linux 的 CVS 客户端程序。

在第一次访问 CVS 服务器时,需要使用下述命令进行登录:

```
$ cvs -d :pserver:anoncvs@sources.redhat.com:/cvs/ecos login
```

这里使用了用户名 anoncvs,可以使用任意口令 (Password)。

在下载 eCos 源码之前,如果要将下载的最新 eCos 源码存放在一个新的目录,则首先应该进入到新的目录,然后再执行下面的命令:

```
$ cvs -z3 -d :pserver:anoncvs@sources.redhat.com:/cvs/ecos co -P ecos
```

这一命令与 CVS 服务器建立连接,检查和下载 eCos 的最新版本,下载的源码存放在 ecos 子目录下。

如果要将现有的 eCos 更新为最新版本,则可以在 eCos 的根目录下执行下面的命令:

```
$ cvs -z3 update -d -P
```

该命令检查 CVS 服务器上的最新版本的 eCos 源码,并与本地 eCos 源码版本进行对照,从 CVS 服务器上下载最新版本的源码,更新本地 eCos 源码。

一旦下载了新的 eCos 源码,如果使用命令行方式进行配置,则应该设置 ECOS_REPOSITRY 环境变量,使其指向新的 ecos/packages 目录:

对于 sh、ksh、bash 用户,使用下述命令进行设置:

```
$ ECOS_REPOSITORY=/ecoscvs/ecos/packages ;export ECOS_REPOSITORY
```

对于 csh、tcsh 用户,使用下面的命令进行设置:

```
$ setenv ECOS_REPOSITORY /ecoscvs/ecos/packages
```

如果使用 eCos 图形配置工具,则可以使用菜单选项“Build→Repository”进行设置。

2.5.2 eCos 配置工具

eCos 有两种形式的配置工具:一种是命令行配置工具;另一种是图形配置工具。这些配置工具分别具有 Windows 和 Linux 的不同版本。

Windows 环境下的命令行配置工具是 ecosconfig.exe,该命令行配置工具必须在 Cygwin 环境下运行。它位于 eCos 源码安装目录的 ecos-2.0/tools/bin 子目录下,也可以位于其他目录下,但 PATH 环境变量应该包含它的路径名。

Windows 环境下的图形配置工具是 Configtool.exe,位于 eCos 源码安装目录的 ecos-2.0/tools/bin 子目录下。eCos 图形配置工具的界面如图 2-7 所示。

Linux 环境下的命令行配置工具是 ecosconfig,位于 eCos 源码安装目录的 ecos-2.0/tools/bin 子目录下。如果单独从 eCos 网站上下载,则可以使用下述命令进行安装:

```
gzip -N -d ecosconfig1314-linux.gz
chmod +x ecosconfig
```

然后再将 ecosconfig 移至 eCos 安装目录的 ecos-2.0/tools/bin 子目录下。

Linux 环境下也有一个图形配置工具,文件名为 configtool,位于 eCos 源码安装目录的

ecos-2.0/tools/bin 子目录下。运行该程序，将出现 Linux 下的图形配置工具界面，其界面如图 2-8 所示。

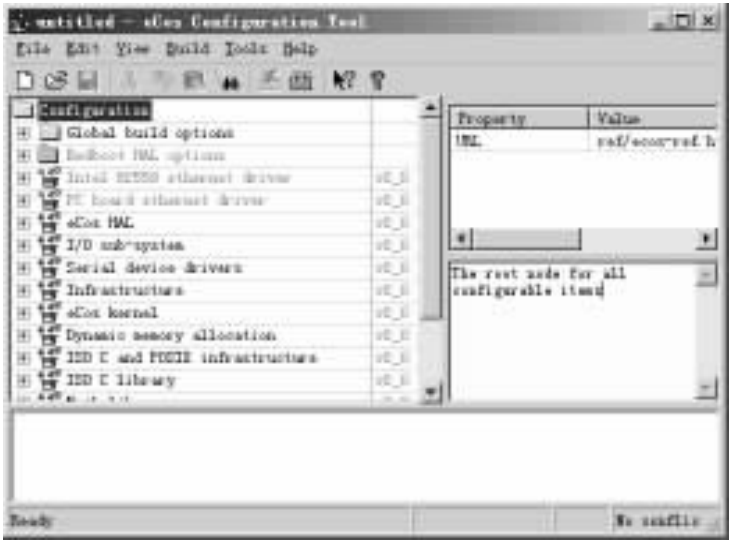


图 2-7　Windows 环境下 eCos 图形配置工具

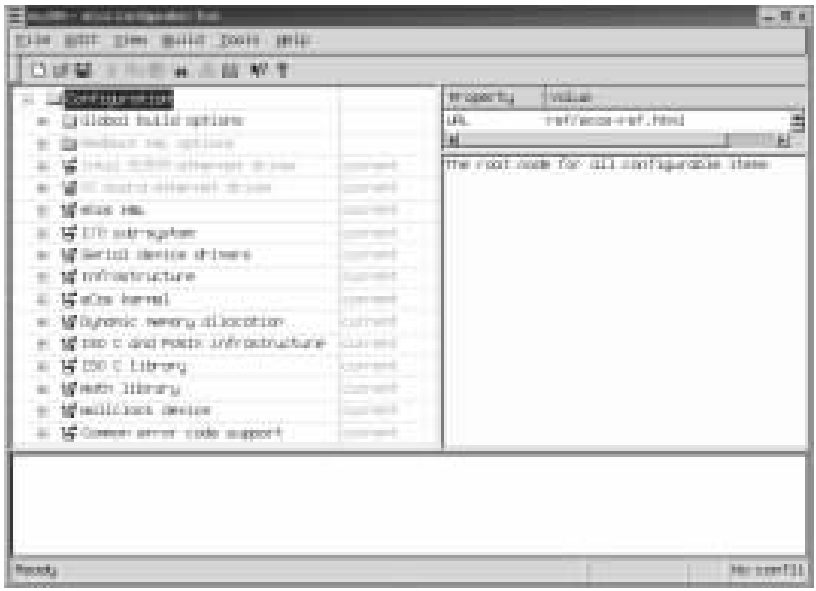


图 2-8　Linux 环境下 eCos 图形配置工具

eCos 图形配置工具安装完成后，必须对 eCos 组件仓库路径名和编译工具（包括交叉编译工具）路径名进行设置。eCos 组件仓库路径名可以使用菜单选项“Build→Repository”进行设置，它是 eCos 源码中包含 ecos.db 数据库文件的路径名。编译工具包括用户工具（包含 cat、ls 等命令的目录）和交叉编译工具（包含 xxx-elf-gcc 如 i386-elf-gcc 等命令的目录），分别使用菜单选项“Tools→Paths→Users Tools”和“Tools→Paths→Build Tools”来指定相应的工具路径名。用户工具路径可以指定为 Cygwin 的 bin 目录，交叉编译工具路径可以使用按前面介绍方法建

立的交叉编译工具目录 opt \ ecos \ gnutools \ bin (以 i386 为例)。

2.6 建立 eCos 开发环境

在完成所有开发工具的编译和安装后,可以开始建立 eCos 开发环境。根据目标系统的不同,eCos 开发环境的建立也有一定的不同之处,但它们的过程基本类似。这一节将以 i386 PC 为例,讲述如何建立 eCos 开发环境。

2.6.1 基于 x86 的 eCos 开发平台

图 2-9 所示为基于 x86 的 eCos 开发环境。它包括两台 PC 机,一台用作 eCos 的开发主机 Host,另外一台为运行 eCos 的目标系统 Target。Host 和 Target 使用串口连接,串口电缆的连接在图中已有说明,串口电缆两端分别连接到 Host 和 Target 的 DB9 串口插座。

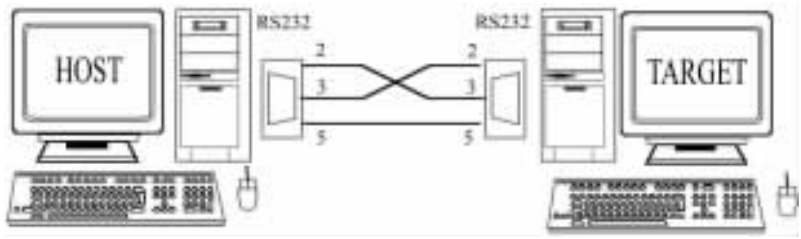


图 2-9 基于 x86 的 eCos 开发平台

Host 主机运行 Windows 2000 操作系统,也可以是 Windows XP,它作为 eCos 开发环境的主开发平台,负责 eCos 系统的配置、编译和链接、eCos 应用程序的编译和链接、目标系统 eCos 应用的加载、调试等任务。它安装有 eCos 开发环境所需的全部软件和工具,主要有:

① Cygwin。
② GCC。
③ GDB (Insight)。
④ Binutils。
⑤ eCos 图形配置工具。
⑥ eCos 系统源码。
⑦ eCos 应用程序源码。

Target 目标系统是运行 eCos 系统的目标平台。在 Host 主机上开发的 eCos 系统将通过 RS232 串口被加载到目标平台上,并在目标平台上运行。目标平台上运行的 eCos 系统可以使用本地控制台 (显示器、键盘),也可以使用 Host 主机上的串口超级终端作为其控制台。Target 系统首先要运行 RedBoot 才能与 Host 主机建立通信连接,建立连接时必须注意 Host 和 Target 两端串口属性参数的设置。Target 上运行下述软件:

① RedBoot。
② eCos 应用。

上述方法采用了 RS232 串口作为 Host 和 Target 的通信接口。除了采用串口外,还可以使用以太网作为它们之间的通信接口。使用以太网时要求 Host 和 Target 都有以太网卡,

Target 上的网卡必须是 i82559 兼容网卡，如 Intel EtherPress Pro 10/100 以太网卡。如果使用其他网卡，必须提供其 eCos 驱动程序。

2.6.2 建立 RedBoot 引导环境

Target 在与 Host 建立连接时以及在运行 eCos 应用的时候，必须具备 RedBoot 引导环境。Target 在 RedBoot 阶段负责与 Host 建立通信连接，并协助完成 eCos 应用程序的加载和运行。一旦 eCos 应用程序开始运行，RedBoot 将把系统控制权全部交给 eCos 系统。目标系统可以采用软盘引导的方式来运行 RedBoot。

在制作 RedBoot 引导盘之前，首先要生成 RedBoot。RedBoot 实际上是 eCos 的一个特殊的应用程序。如果 eCos 源码中包含了预编译好的 RedBoot，可以直接使用，但应该注意其目标系统类型是否是 i386 PC，以及是否是软盘引导方式。如果没有，则必须使用 eCos 配置工具对其进行配置，并生成所需的软盘引导 RedBoot。下面介绍软盘引导 RedBoot 的生成方法。

1）软盘引导 RedBoot 的配置。首先运行 eCos 图形配置工具，选择“RedBoot”模板（“Build →Templates”），出现图 2-10 所示对话框。在对话框中选择硬件平台“i386 PC target”并选择“redboot”软件包。

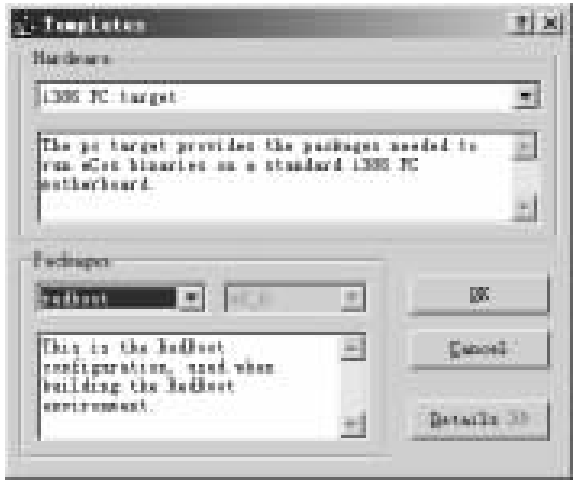


图 2-10 RedBoot 模板选择

选择“RedBoot”模板（如果出现冲突对话框，点按继续“Continue”）后，在 eCos 配置窗口的“eCos HAL”选项内找到“i386 PC Target”配置选项，将其启动类型“Startup type”设置为“FLOPPY”，如图 2-11 所示。与此同时，还应该对目标系统的串口进行设置，包括串口波特率（默认值为 38400bit/s）、串口数目、指定 GDB 串口、诊断串口以及控制台串口（如果使用同一个串口，则其波特率应该相同）。另外还可以选择是否在目标系统显示器上显示输出信息。串口属性如下：

① 波特率：38400（注意 Host 上的串口也必须使用相同的波特率）。

② 数据位：8。

③ 奇偶位：0。

④ 停止位：1。

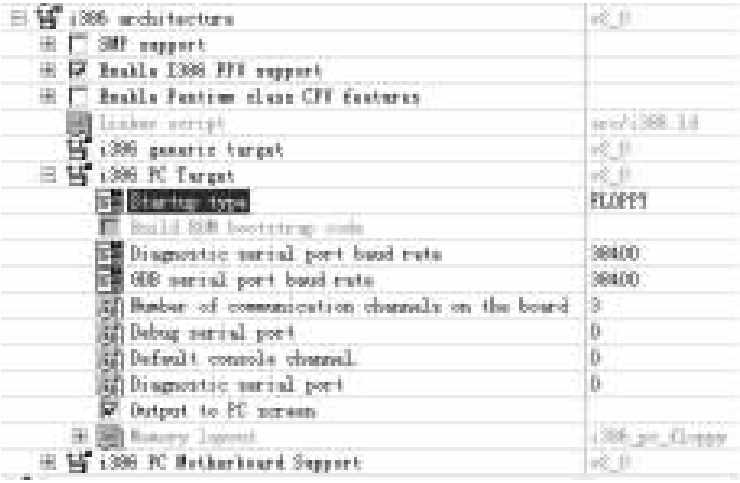


图 2-11　目标系统软盘引导 RedBoot 的配置

⑤ 流控制 :无。

如果使用以太网与 Host 主机进行通信,则必须在 RedBoot 的配置中包含网络支持。

2) RedBoot 的编译。在对软盘引导 RedBoot 进行配置后,便可以开始编译 RedBoot。首先要保存所进行的配置,用菜单选项"File→Save As",将其保存为 my _ redboot. ecc,然后选择"Build→Library",eCos 配置工具将开始编译新的 RedBoot。编译时要求已经设置好了编译工具和用户工具的路径名。编译完成后,在 RedBoot 的安装目录 my _ redboot _ install \ bin 下将生成 redboot. bin 文件,该文件就是软盘引导 RedBoot 所需要的二进制文件。

3) RedBoot 引导盘的制作。在 Host 主机完成 RedBoot 的编译并产生新的 RedBoot. bin 文件后,在 Cygwin 环境下通过下面的步骤可以生成 Target 的 RedBoot 引导软盘:

首先用 mount 命令检查是否已经安装了软驱,如果出现了下面的信息,则可以跳过软驱安装,直接生成引导盘:

```
\ \ . \ a :/dev/fd0 user binmode
```

如果没有出现说明这样的信息,则说明还没有安装软驱。使用下面的命令安装软驱:

```
$ mount -f -b //./a :/dev/fd0
```

在确认软驱已经被安装后,将软盘插入软驱中,使用下面的命令生成 RedBoot 引导盘:

```
$ dd conv = sync if = my _ redboot _ install/bin/redboot. bin of = /dev/fd0
```

如果是在 Linux 环境下,则使用下面命令生成引导盘:

```
$ dd conv = sync if = my _ redboot _ install/bin/redboot. bin of = /dev/fd0H1440
```

在实际使用过程中,要注意 redboot. bin 的路径名。

4) 运行 RedBoot 并与 Host 连接。生成 RedBoot 引导盘后,将引导盘插入目标系统的软驱内,重启系统。目标平台将从软盘引导,运行 RedBoot。如果此时 Host 主机的串口超级终端或者其他串口终端连接到目标平台的串口上,则这些终端将显示 RedBoot 的引导界面。

RedBoot 引导完成后,Host 主机可以通过 GDB 与其建立通信连接,并且可以向 Target 加

载和运行 eCos 程序。在使用 GDB 进行连接时,应该注意超级终端是否已经与 Target 脱连(当使用同一个串口时)。

在 Cygwin 环境下,使用下述命令运行 GDB,与 Target 的 RedBoot 进行连接:

```
$ gdb
```

此时将运行 Insight(图形界面 GDB),并出现 Insight 的操作界面。在其界面上选择菜单"File→Target Settings"对串口进行如下设置:

```
Target:Remote/Serial
Baud Rate:38400
Port:com1  (如果是 Linux 环境,则用/dev/ttyS0)
```

然后再回到 GDB 主界面选择菜单"Run→Connect to target"与目标系统建立连接。

如果不使用 Insight 图形界面,可以使用下面的命令进入 GDB 的命令行操作界面:

```
$ gdb -nw
```

此时再使用下述命令与目标系统进行连接:

```
(gdb) set remotebaud 38400
(gdb) target remote com1  (如果是 Linux 环境,则用/dev/ttyS0)
```

至此,一个完整的基于 x86 的 eCos 开发环境已经建立完成。下一章将介绍如何使用 eCos 配置工具开发 eCos 应用程序,并用简单的例子来说明整个过程。

第 3 章　eCos 配置工具与编程实例

eCos 的一个最显著的特性是其配置功能，实现其配置功能的主要途径是使用 eCos 配置工具。eCos 配置工具在源码级对系统进行配置和裁剪。这种配置操作在系统编译之前进行，它提供一个配置文件以及其他一些文件用于生成用户应用程序。eCos 具有一个组件仓库，组件仓库内包含了用于生成 eCos 系统所需的源码及其他相关文件，eCos 配置工具在启动时将加载这一组件仓库。组件仓库内包含了一些用于描述其各组件的文件，称为 CDL (组件定义语言，将在第 12 章介绍) 文件，eCos 配置工具将依靠这些 CDL 文件对组件进行管理和配置。

在开发 eCos 系统时，首先必须熟悉其配置工具的使用。本章重点介绍 eCos 的图形配置工具的使用方法，同时简要说明如何使用 eCos 的命令行配置工具。另外还举例说明了如何使用配置工具生成 eCos 系统，包括应用程序的编译、加载、运行和调试。通过本章的学习，读者完全可以自己动手配置和建造简单的 eCos 系统，并可以开发简单的 eCos 应用程序。

3.1　eCos 图形配置工具

eCos 具有两种类型的配置工具：一种是图形配置工具；另一种是命令行配置工具。Windows 和 Linux 都支持这两种配置工具，它们在这两种环境下的使用方法基本相同。因此这里主要介绍 Windows 环境下 eCos 配置工具的使用。

eCos 图形配置工具具有操作简单、使用方便、配置灵活的特点，而且具有很强的配置能力，它集配置、编译、调试、测试、运行等功能于一体。eCos 这种灵活而强大的配置能力已经被其他一些嵌入式操作系统所借鉴。

eCos 图形配置工具界面如图 3-1 所示。它包括几个窗口 配置窗口、属性窗口、描述窗口、输出窗口、冲突窗口和内存布局窗口。其中冲突窗口和内存布局窗口必须使用菜单选项“View”才能显示。在某些版本的图形配置工具中，没有内存布局窗口。

配置窗口是对 eCos 进行配置的主要窗口，大部分的配置工作都在此窗口下进行。它采用树型结构显示当前配置中所包含的所有配置选项。这些配置选项可以选择使用宏名或者其简要描述名来显示该选项。在配置窗口中对可配置的选项进行设置，可以使能或禁止该选项，或者对选项的值进行修改。选中某一配置选项并点击鼠标右键可以查看该选项的简要说明、对应头文件、选项的属性以及浏览联机帮助文件，还可以选择对包进行卸载操作。

配置窗口中的某个配置选项被选中时 (鼠标点击该选项)，属性窗口将显示该配置选项的属性。属性中包含有该选项的宏名、头文件以及相应的参考文档。用鼠标双击头文件和参考文档可以查看这些文件。关于选项的属性将在第 12 章介绍。

描述窗口显示在配置窗口中被选中的配置选项的简要描述信息。用户在使用时可以从这里得到简单的帮助信息。

输出窗口用于显示配置、编译和链接时所产生的各种提示、警告和错误信息。在该窗口点击鼠标右键可以选择保存或清除所显示的输出信息。

冲突窗口显示当前配置所存在的冲突。eCos 的许多配置选项之间存在相互依存关系，当对它们所进行的配置出现冲突时，eCos 配置工具将自动检测这些配置冲突，并在冲突窗口显示这些冲突。

内存布局窗口显示的是当前配置所使用的内存分布方案。包括向量地址、程序代码地址与大小、数据段地址与大小等等。点击鼠标右键可以对它们进行修改。有些版本的图形配置工具没有内存布局窗口，如果需要，则必须在相应的安装树目录中的内存布局文件 install/include/pkgconf/mlt_*.* (例如：pkgconf/mlt_i386_pc_floppy.ldi 文件) 中进行修改。

图 3-1　eCos 图形配置工具窗口

eCos 图形配置工具中的菜单工具提供了各种操作功能。它可以创建新的配置、打开一个现有配置，可以对配置进行导入导出操作，也可以在当前配置中寻找某个配置选项。它可以对配置工具进行设置，如显示字体、配置冲突检查点、运行测试程序时与目标系统的连接方式与属性等等。它还可以进行系统编译操作和选项设置、提供对 eCos 软件包的管理和联机帮助。

3.2　图形配置工具的使用

eCos 图形配置工具为 eCos 的配置、组件管理、编译以及执行提供了一个方便而灵活的操作环境。本节将从这些方面简要介绍在 Windows 环境下如何使用图形配置工具，Linux 环境下的使用方法基本相同。

3.2.1　组件仓库位置

eCos 图形配置工具在被启动时将访问包含只读配置信息的组件仓库。组件仓库包含了

用于产生一个配置所需要的所有源码文件和其他一些文件，它是一个多层次的目录结构。在第一次使用图形配置工具的时候，必须指定组件仓库的位置。以后每次运行图形配置工具时将使用上一次运行时所使用的组件仓库。

组件仓库位置的指定可以通过菜单选项来完成。点击“Build→Repository”将出现图 3-2 所示的对话框，在对话框内输入组件仓库所在目录位置。

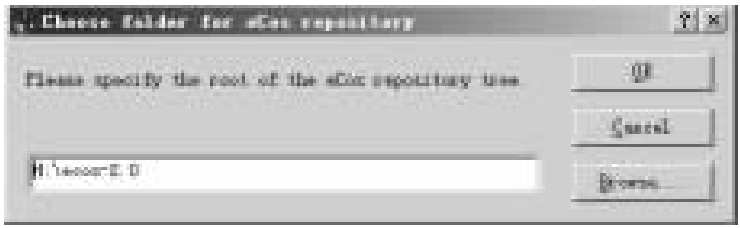


图 3-2　组件仓库位置的指定

当改变 eCos 源码目录时，需要使用这种操作来指定新的组件仓库位置。如果指定的位置不正确，图形配置工具将提示重新输入有效的位置。

点击菜单选项“Help→Repository Information...”，可以检查当前所使用的组件仓库位置，同时还可以显示当前 eCos 配置保存文件位置、当前硬件平台和默认包等信息。

3.2.2　配置文件的管理

使用 eCos 配置工具对配置选项所进行的设置以及其他一些信息可以用文件的形式保存下来。在进行一项新的配置时，首先必须产生一个新的配置文件（*.ecc）。配置文件类似于其他软件开发环境中的项目（Project）。

点击菜单选项“File→New”将创建一个新的配置文件。一旦创建完成，就可以使用“File→Save as”或“File→Save”将其保存。下次再使用该配置文件时，可以使用“File→Open”打开对应的配置文件。

创建并保存一个新的配置文件后，eCos 配置工具将自动产生三个与该配置相关的目录：

① 编译树（Build Tree）：对 eCos 进行编译时所生成的文件都存放于此，包括 makefile 文件、目标文件和其他一些文件。

② 安装树（Install Tree）：存放用于编译 eCos 应用程序的所有文件，包括库文件 libtrget.a 和头文件。eCos 编译后的输出结果文件都存放在安装树目录内。

③ 内存布局目录（mlt 目录）：存放内存布局文件。

例如，如果新的配置文件被保存为 my386.ecc，那么将产生下述文件和目录：

① 配置文件：my386.ecc。

② 编译树目录：my386_build。

③ 安装树目录：my386_install。

④ 内存布局目录：my386_mlt。

3.2.3　模板选择

在创建新的配置时，需要指定目标平台的类型以及所需的软件包。eCos 源码中提供了许

多可以选择的各种模板。这些模板分别对应不同处理器的各种开发板,对于新的平台可以选择与其最相近的模板。

模板的选择可以通过点击菜单“Build→Templates”进行。此时将出现图 3-3 所示的对话框。

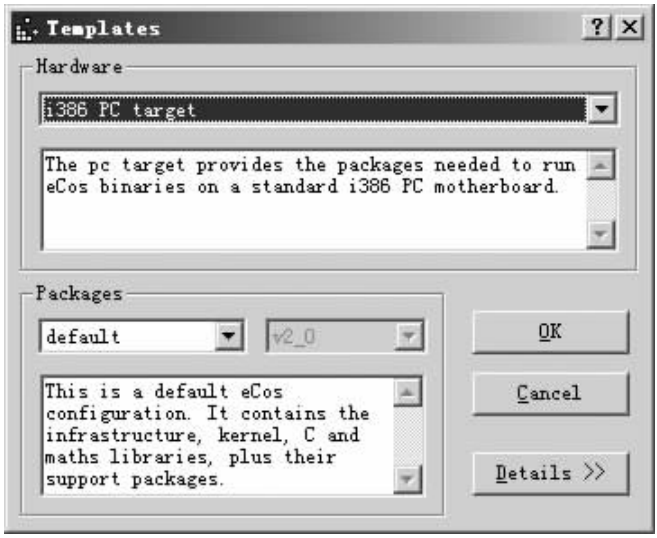


图 3-3　选择平台模板

假设以 i386 微机作为目标开发平台,可以选择硬件为“i386 PC target”,同时选择默认软件包(“default”)。点击“Details”按钮可以查看具体包括了哪些软件包。

eCos 图形配置工具提供了源码级的裁剪能力,用户可以根据目标系统的实际需要来增加或删除某些软件包。在主菜单中选择“Build→Packages”将出现图 3-4 所示的对话框,在该对话框中可以实现包的增加和删除。

图 3-4 的左上方窗口显示的是可以被增加的软件包,右上方窗口显示的是当前配置中已经存在的软件包。中间窗口显示的是被选软件包的简要描述,“Version”栏内显示该软件包的版本号。如果存在多个版本,则可以在此选择指定版本的软件包。在“Keywords”栏内输入软件包的关键词(包的名字、宏名、描述)可以迅速地在包的窗口内显示相应的软件包。如果选择了“Match exactly”,则只显示与关键词完全匹配的软件包。如果选择了“Omit hardware packages”,则只显示与硬件无关的软件包。

3.2.4　选项配置

eCos 提供了许多配置选项,用户可以根据实际需要对这些选项进行配置。选项的配置操作主要在配置窗口内进行。配置窗口以树型结构表示当前所使用软件包中的各种配置选项。图 3-5 为配置窗口。

对于那些选项值可以被修改的配置选项,可以使用复选框、单选按钮、单元格的方式进行配置。单元格位于配置窗口右边,用鼠标点击单元格便可以选择或修改其值。单元格的值有三种数据类型 整型(十进制或十六进制)、浮点、字符串。

某些配置选项处于禁止状态,处于禁止状态的配置选项的标号以及与其相关的控制和单

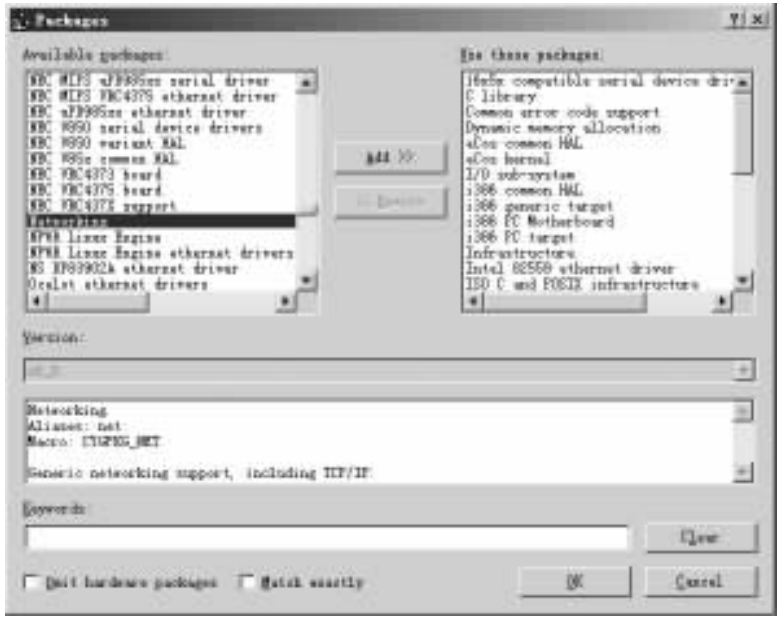


图 3-4　软件包的增加与删除

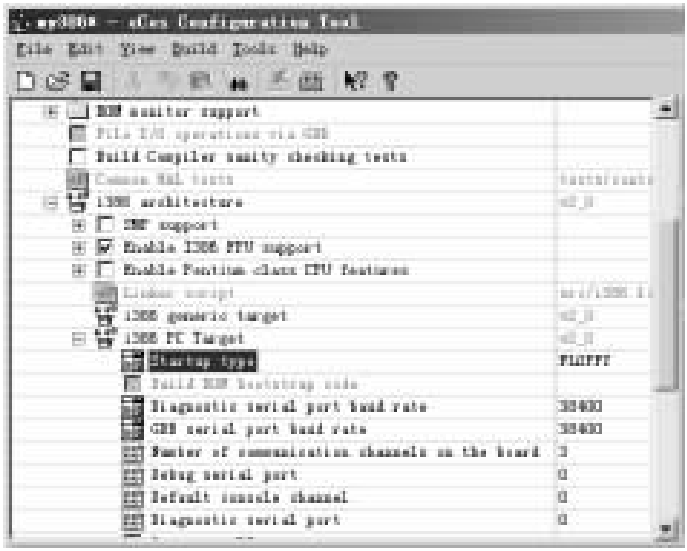


图 3-5　eCos 配置窗口

元格呈灰色状态，这些配置选项的值不能被修改。

值得注意的是对启动类型（Startup type）的选择配置。对于一个可执行的映像文件，有多种方法将其加载到目标平台上。根据加载方式的不同，必须使用不同的启动方式来生成可执行文件。eCos 的硬件抽象层 HAL 提供了一个配置选项“Startup type”用于支持不同的加载方法。该配置选项的典型值可以设置为 ROM 启动、RAM 启动、软盘启动、Grub 引导等。表 3-1 列举了不同加载方式应该配置的启动方式。

表 3-1　启动方式的配置

加 载 方 式	HAL 配 置
程序位于 ROM 内	ROM 启动
加载到 ROM 模拟器	ROM 启动
使用 RedBoot 加载到 RAM	RAM 启动
从软盘加载	软盘启动
通过 Grub 加载程序引导	Grub 启动

3.2.5　冲突的解决

eCos 具有各种配置选项，不同选项之间可能存在某些依赖关系。在对 eCos 进行配置的过程中，对某些配置选项的配置可能引起不同配置选项之间的相互冲突。图形配置工具的冲突窗口以及配置工具下边的状态栏将显示当前配置中的所有冲突和冲突数目。这些冲突的解决需要相当长的时间。为此，eCos 配置工具提供了一个冲突自动解决机制。

在主菜单中选择“View→Settings...”，在出现的设置对话框内选择“Conflict Resolution”项，将出现如图 3-6 所示的冲突检查点设置框。

图 3-6　冲突检查点设置框

根据需要，可以选择在每次改变一个配置选项的时候进行冲突检查，也可以在保存配置文件的时候再进行冲突检查。另外还可以选择不检查冲突。值得注意的是，如果选择每一次改变配置选项时进行冲突检查，则只显示新出现的冲突 如果选择在保存配置文件的时候进行检查，那么将显示所有的冲突。

如果选择了“Automatically suggest fixes”，那么一旦出现新的冲突，就将出现一个图 3-7 所示的冲突解决对话框。使用菜单项“Tools→Resolve Conflicts”也将出现同样的对话框。

冲突解决对话框具有两个窗口。上边的窗口包含了所出现的冲突，其显示格式与配置工具中的冲突窗口一致。下边的窗口包含了解决这些冲突的建议方法。如果点击“Continue”按钮，将采用该窗口中被选择的冲突解决方案。

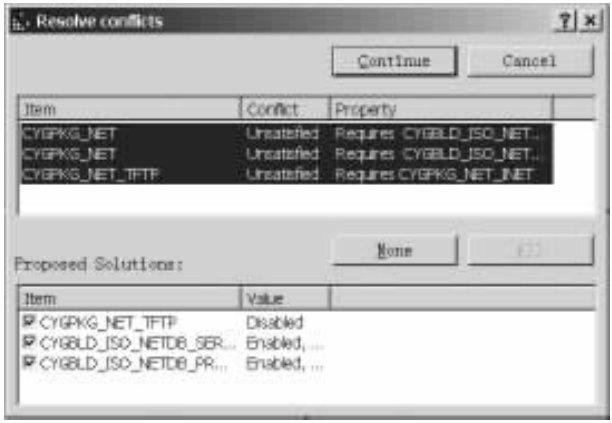


图 3-7　冲突解决对话框

3.2.6　配置选项的查找

eCos 图形配置工具的配置窗口包含了许多配置选项，这些配置选项可以使用宏名来显示，也可以使用简要描述来显示。当配置选项较多时，要查找某个选项是较为困难的。为此，eCos 图形配置工具提供了一个配置选项的查找机制。

选择菜单项“Edit→Find”，将出现图 3-8 所示的选项查找对话框。

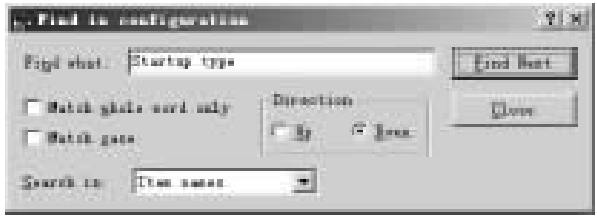


图 3-8　选项查找对话框

使用这一对话框可以以下面几种形式（在“Search in”下拉菜单中选择）在配置窗口中查找字符串：

① 宏名（Macro names）根据输入的配置选项宏名进行查找。

② 选项名（Item names）根据输入的配置选项的描述名进行查找。

③ 简要描述（Short descriptions）根据输入的配置选项的简要描述进行查找。

④ 当前值（Current values）根据配置选项的当前值进行查找。

⑤ 默认值（Default values）根据配置选项的默认值进行查找。

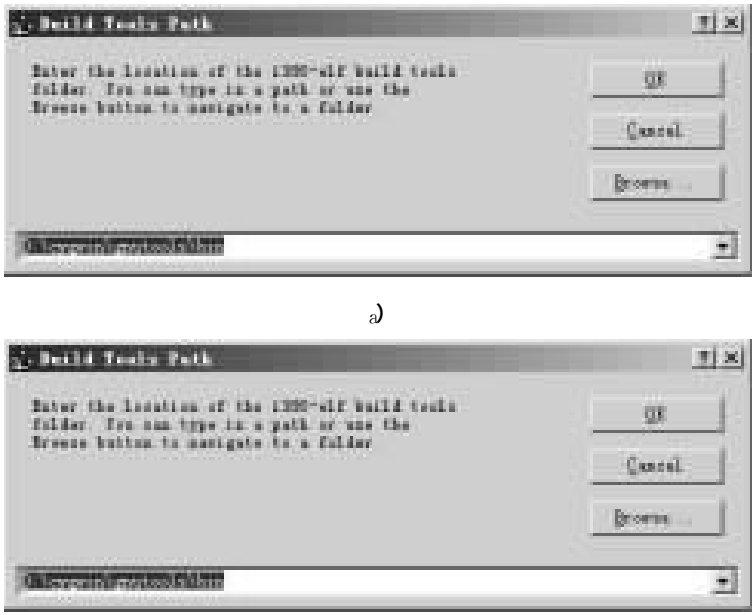
如果查找成功，还可以使用“Find Next”按钮查找下一个相匹配的配置选项。

3.2.7　编译

利用 eCos 配置工具可以对已经配置好的 eCos 进行编译。对 eCos 进行编译时要用到针对目标平台的交叉编译工具，如编译器、链接器等，另外还需要使用一些用户工具（cat、ls 等）。编译工具的选择通常只需进行一次，可以通过菜单“Tools→Paths→Build Tools”来设置编译工

具所在的目录路径，用户工具所在目录路径可以使用菜单“Tools→Paths→User Tools”进行设置。如果目标系统采用不同的 CPU，则需要重新指定相应的编译工具目录路径名。

图 3-9 是指定 i386 PC 的编译工具的对话框。其交叉编译工具路径名指的是包含 i386-elf-gcc 等 i386-elf 编译工具的目录，用户工具指的是主机系统包含 cat 和 ls 文件的目录。



a)

b)

图 3-9　编译工具和用户工具的选择

a) 编译工具路径设置　b) 用户工具路径设置

eCos 图形配置工具提供了一个查看编译选项的功能。使用菜单项“Build→Options”将出现一个有关编译选项的对话框 (图 3-10)，该对话框包含一个下拉菜单和两个窗口。下拉菜单

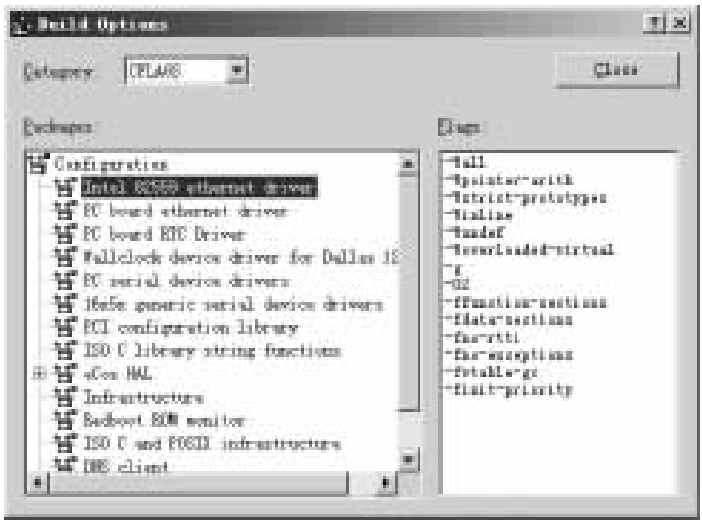


图 3-10　编译选项对话框

“Category”用于选择显示全局编译标志(“CFLAGS”)或链接标志(“LDFLAGS”)。左边窗口显示当前配置中的软件包,右边窗口显示的是当前所选择软件包的编译或链接选项。编译选项对话框只提供选项的只读功能。

对 eCos 进行编译时,可以使用图形配置工具的菜单项“Build”进行。Build 菜单有四个与编译相关的选项,它们分别是:

① Library:对 eCos 进行编译和链接。编译结果是产生一个库文件,用户应用程序将与该库进行链接。

② Tests:对 eCos 测试程序进行编译,并与 eCos 库进行链接。

③ Clean:删除所有中间文件,使得所有文件可以被重新编译。

④ Stop:中止当前正在进行的编译过程。

3.2.8 执行

对测试程序进行编译后生成的可执行测试程序,可以通过 eCos 图形配置工具将其加载到目标平台上执行。测试程序的执行通过 GDB 进行控制。使用菜单项“Tools→Run Tests”可以实现测试程序的执行。程序的执行通常使用单独的 GDB 命令进行控制。

在配置工具上选择运行测试程序时,配置工具查找与当前配置的硬件目标相对应的平台名。如果没有相应的平台,则出现一个对话框用于对该平台进行定义,如图 3-11 所示。增加新的平台可以使用菜单“Tools→Platforms”实现。对话框中的“Command Prefix”栏填入命令前缀(如 i386-elf-gdb.exe 的前缀 i386-elf),GDB 参数栏填入传送给 GDB 的命令与参数。

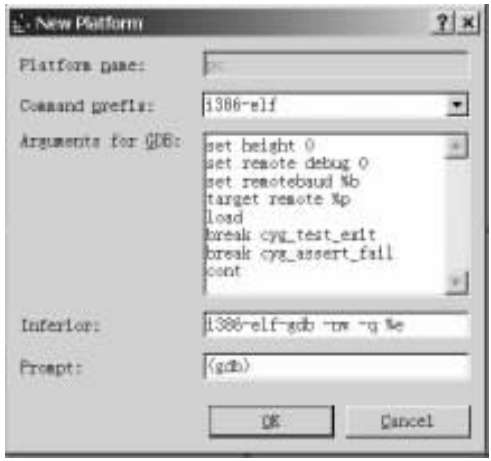


图 3-11 新平台的定义

当找到相应的平台时,或者对新平台进行设置后,将出现一个运行测试程序的对话框,如图 3-12 所示,它包含三个表单窗口:

①“Executables”:列举被选择的测试程序,可以选择执行一个或多个程序。

②“Output”:程序执行时的输出信息。

③“Summary”:程序执行的概要情况。

测试程序运行对话框的下部有三个按钮,其中“Run/Stop”按钮用于运行测试程序或停止测试程序。“Properties”按钮用于对目标平台的连接属性进行设置,如图 3-13 所示。在连接属

性设置对话框中可以对程序加载的超时时间、程序运行的超时时间、串口连接参数、TCP/IP连接参数等等进行设置。

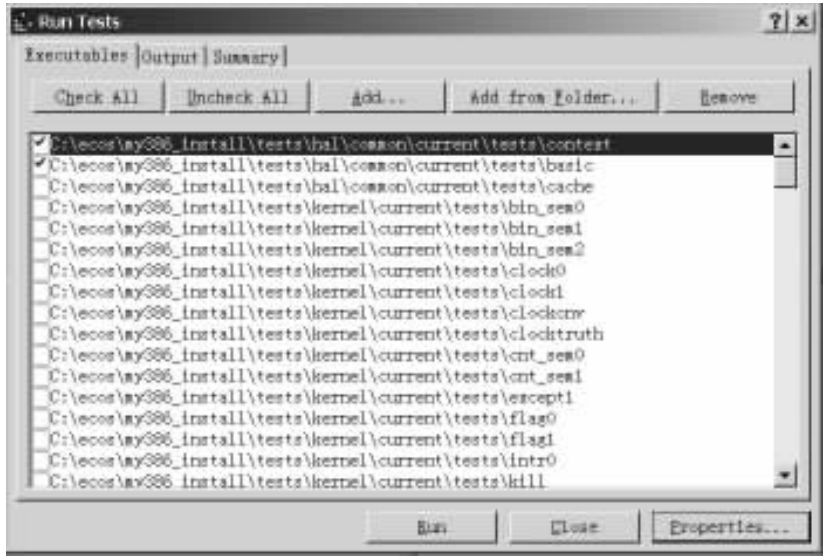


图 3-12　执行测试程序

测试程序的执行情况显示在“Summary”窗口内。其显示内容包括执行日期与时间、测试程序名、测试结果状态、程序加载时间、程序执行时间等。

图 3-13　与目标平台的连接属性设置

3.3 命令行配置工具

在某些环境不能使用图形配置工具的情况下（如 UNIX），需要使用命令行配置工具 ecosconfig 对 eCos 进行手工配置。

在使用 ecosconfig 配置工具之前，必须对环境变量 PATH 和 ECOS _ REPOSITORY 进行设置。PATH 环境变量应该包含编译 eCos 所需的交叉编译工具路径和用户工具路径，ECOS _ REPOSITORY 环境变量为 eCos 组件仓库路径。例如：

```
$ PATH=/bin:/opt/ecos/gnutools/bin:$ PATH
$ export PATH
$ ECOS _ REPOSITORY=/opt/ecos/packages
$ export ECOS _ REPOSITORY
```

上面例子中用户工具目录为/bin，交叉编译工具为/opt/ecos/gnutools/bin，eCos 组件仓库路径为/opt/ecos/packages。

3.3.1 ecosconfig 配置工具

使用 ecosconfig 命令时，可以用 ecosconfig --help 来查看该命令的使用方法。图 3-14 为该命令的使用方法。

```
$ ecosconfig --help
Usage: ecosconfig [ qualifier ... ] [ command ]
  commands are:
    list                                    : list repository contents
    new TARGET | TEMPLATE [ VERSION ] ]     : create a configuration
    target TARGET                           : change the target hardware
    template TEMPLATE [ VERSION ]           : change the template
    add PACKAGE [ PACKAGE ... ]             : add package(s)
    remove PACKAGE [ PACKAGE ... ]          : remove package(s)
    version VERSION PACKAGE [ PACKAGE ... ] : change version of package(s)
    export FILE                             : export minimal config info
    import FILE                             : import additional config info
    check                                   : check the configuration
    resolve                                 : resolve conflicts
    tree                                    : create a build tree
  qualifiers are:
    --config=FILE                           : the configuration file
    --prefix=DIRECTORY                      : the install prefix
    --srcdir=DIRECTORY                      : the source repository
    --no-resolve                            : disable conflict resolution
    --version                               : show version and copyright
    -q, --quiet                             : reduce verbosity
    -v, --verbose                           : increase verbosity
    -i, --ignore-errors                     : ignore unresolved conflicts
    --help                                  : display this message
```

图 3-14 ecosconfig 命令使用方法

图 3-14 已列举了 ecosconfig 命令的使用方法：

ecosconfig［qualifier . . . ］［command ］

其中命令（command）为：

① list 列举组件仓库内容，包括软件包、目标平台和模板、别名、版本信息。

② new TARGET［TEMPLATE［VERSION］］：产生一个新的 eCos 配置，可以指定目标系统硬件和模板，默认模板为“default”。如果未指定版本号，则使用最新版本。

③ target TARGET：改变 eCos 的目标平台硬件。

④ template TEMPLATE［VERSION］：改变 eCos 模板的选择。如果未指定版本号，则使用最新版本。

⑤ add PACKAGE［PACKAGE ...］：在当前配置中增加指定的软件包。

⑥ remove PACKAGE［PACKAGE ...］：在当前配置中删除指定的软件包。

⑦ version VERSION PACKAGE［PACKAGE ...］：选择指定版本的软件包。

⑧ export FILE：导出并保存 eCos 的最小配置信息到指定文件。

⑨ import FILE：将指定文件所保存的最小配置信息导入到当前配置。

⑩ check：检查并显示当前配置的相关信息，包括硬件平台的选择、模板的选择、增加的软件包、删除的软件包、非最新版本软件包的版本号、当前配置中的冲突等。

⑪ resolve：解决当前配置中所出现的冲突。

⑫ tree：生成当前配置的编译树（Build tree）。

限定符（qualifiers）包括：

① --config = FILE：指定配置工具所使用的 eCos 配置保存文件，默认为当前目录下的 ecos.ecc 文件。

② --prefix = DIRECTORY：指定安装树（Install tree）目录，默认为当前目录下的 install 目录。

③ --srcdir = DIRECTORY：指定组件仓库位置，默认为 ECOS_REPOSITORY 环境变量指定的位置。

④ --no-resolve：禁止冲突解决，可以对 eCos 配置保存文件进行编辑对冲突进行手工解决。

⑤ --version：显示配置工具的版本和版权信息。

⑥ -q，--quiet：显示少量信息。

⑦ -v，--verbose：显示较多的信息。

⑧ -i，--ignore-errors：忽略未解决的冲突，默认情况下 ecosconfig 一旦遇到冲突将自动退出。

⑨ --help：显示帮助信息，提供基本的配置工具使用指导。

3.3.2 使用 ecosconfig 配置 eCos

在使用 ecosconfig 对 eCos 进行配置之前必须选择一个工作目录。假设工作目录为/ecos-work，如果该目录不存在则可以创建这个目录：

```
$ mkdir /ecos-work
$ cd /ecos-work
```

此时可以使用下述命令创建一个新的基于 i386 PC 的配置：

```
$ ecosconfig new pc
```

该命令将在当前目录（/ecos-work）下产生一个配置文件 ecos.ecc。如果要对其进行具体配置，可以手工编辑 ecos.ecc 文件，对配置选项进行修改。

使用下面的命令产生编译树 Build Tree：

```
$ ecosconfig tree
```

该命令将生成 build 目录结构，同时还生成一个安装树目录 install。在当前目录下还生成了一个用于编译的 makefile 文件。此时可以使用下述命令对 eCos 进行编译：

```
$ make
```

编译产生的库文件位于安装树目录 install/lib 内。如果要支持多处理器，可以使用下面命令进行编译，其中 n 表示处理器的个数：

```
$ make -j n
```

如果要对测试程序进行编译，可以使用下面的命令：

```
$ make tests
```

该命令将所有的测试程序进行编译，并与 eCos 编译产生的库文件进行链接，生成可在目标系统上运行的可执行程序。这些可执行测试程序位于安装树目录的 test 子目录下。

3.4 eCos 应用程序

在完成对 eCos 的配置工作后，再对其进行编译和链接，最后生成库文件。eCos 应用程序将与库文件进行链接，生成最终所需的可在目标系统上运行的可执行文件。eCos 的编译采用 GNU C 和 C++ 交叉编译工具。这一节首先介绍 eCos 应用程序的编译方法，然后举例说明如何编写 eCos 应用程序以及如何编译、加载和运行。

3.4.1 使用编译工具

eCos 应用程序可以使用 C 或 C++ 进行编程，对它们的编译所使用的工具也是 GNU C 和 C++ 交叉编译工具。在使用 GCC 编译工具时，需要使用某些选项。下面的例子是对用 C 语言编写的 eCos 程序进行编译和链接时所使用的一组最小选项示例：

```
$ GCC -c -IINSTALL_DIR/include file.c
$ GCC -o program file.o -LINSTALL_DIR/lib -Ttarget.ld -nostdlib
```

注意上面例子中的 GCC 表示的是目标平台相应的交叉编译工具。如：i386-elf-gcc，mn10300-elf-gcc，mips-tx39-elf-gcc，powerpc-eabi-gcc，sparclite-elf-gcc，arm-elf-gcc，mips64vr4300-elf-gcc，sh-elf-gcc 等等。INSTALL_DIR 指的是 eCos 编译后所产生的安装树目录，该目录包含了编译 eCos 应用程序所需要的包含文件和库文件。

除了上面例子中的选项外，还可以使用其他一些编译选项，如：

```
$ GCC -c -IINSTALL_DIR/include -I. -ffunction-sections -fdata-sections
      -g -O2 file.c
$ GCC -o program file.o -ffunction-sections -fdata-sections
      -Wl,--gc-sections -g -O2 -LINSTALL_DIR/lib -Ttarget.ld -nostdlib
```

对于使用 C++ 编写的 eCos 应用程序,可以使用 G++ 交叉编译工具。下面的例子是对 C++ 的 eCos 程序进行编译和链接所使用的一组最小选项示例:

```
$ G++ -c -IINSTALL_DIR/include -fno-rtti -fno-exceptions file.cxx
$ G++ -o program file.o -LINSTALL_DIR/lib -Ttarget.ld -nostdlib
```

注意上面例子中的 G++ 表示的是目标平台相应的交叉编译工具。如:i386-elf-g++,mn10300-elf-g++,mips-tx39-elf-g++,powerpc-eabi-g++,sparclite-elf-g++,arm-elf-g++,mips64vr4300-elf-g++,sh-elf-g++ 等等。INSTALL_DIR 指的是 eCos 编译后所产生的安装树目录。在编译 C++ 程序时,还可以使用其他编译和链接选项,如:

```
$ G++ -c -IINSTALL_DIR/include -I. -ffunction-sections -fdata-sections
    fno-rtti -fno-exceptions -fvtable-gc -finit-priority -g -O2 file.cxx
$ G++ -o program file.o -W1,--gc-sections -g -O2 -LINSTALL_DIR/lib
    -Ttarget.ld -nostdlib
```

3.4.2 简单的 hello 程序

eCos 源码中包含了一些例子程序,位于 examples 目录内。这些例子程序包含了从简单到复杂的程序实例,参考这些例子程序有助于了解如何编写 eCos 应用程序。

最简单的一个例子程序是 hello.c 应用程序:

```
/* this is a simple hello world program */
#include <stdio.h>
int main(void)
{
printf("Hello, eCos world! \n");
return 0;
}
```

以普通 PC 作为目标平台为例,假设已经对 eCos 进行配置并完成了编译(前面产生的配置文件 my386.ecc),编译后产生了安装树目录 my386_install(d:\ecos-work\my386_install),该目录包含了 include 和 lib 两个子目录。include 目录包含了 eCos 应用程序引用的头文件,lib 目录包含了 eCos 库文件和链接文件。

在 Cygwin 环境下,可以使用下述命令对 hello.c 进行编译和链接:

```
$ i386-elf-gcc -g -I/d/ecos-work/my386_install/include hello.c
  -L/d/ecos-work/my386_install/lib -Ttarget.ld -nostdlib
```

其中-g 选项使能 debug,-I 指定头文件路径/d/ecos-work/my386_install/include,-L 为链接器指定链接脚本文件 target.ld 的位置。编译后产生的可执行文件为 a.out。

此时,可以使用 GDB 将执行文件 a.out 加载到目标平台上,并启动其在目标系统上的执行。在加载程序之前,目标系统首先要运行 RedBoot。如果以普通 PC 作为目标平台,可以使用软盘引导进入 RedBoot。RedBoot 实际上也是 eCos 应用的一个特例,也可以使用 eCos 对其进行配置和编译。RedBoot 的配置和编译以及引导软盘的制作方法将在第 4 章中介绍。

目标系统进入 RedBoot 状态后，如果使用 GDB 命令行方式，在 Host 主机上执行下面的命令：

```
$ i386-elf-gdb -nw a.out
```

该命令将进入 GDB（出现提示符“(gdb)”）。然后按下述步骤可以连接到目标平台（串口 COM1，波特率 38400）并加载程序：

```
(gdb) set remotebaud 38400
(gdb) target remote COM1
(gdb) load
```

如果是 Linux 系统，将 COM1 换为/dev/ttyS0。

程序 a.out 加载到目标平台上后，可以使用下述命令启动程序的运行：

```
(gdb) continue
```

或者

```
(gdb) run
```

此时，程序开始运行，并输出程序执行结果：

```
Hello，eCos World！
```

上面的操作均基于命令行方式。如果想使用 GDB 图形界面（Insight），则可以使用不带-nw 选项的 gdb 命令：

```
$ i386-elf-gdb a.out
```

该命令将调用 Insight，出现 GDB 图形界面，如图 3-15 所示。

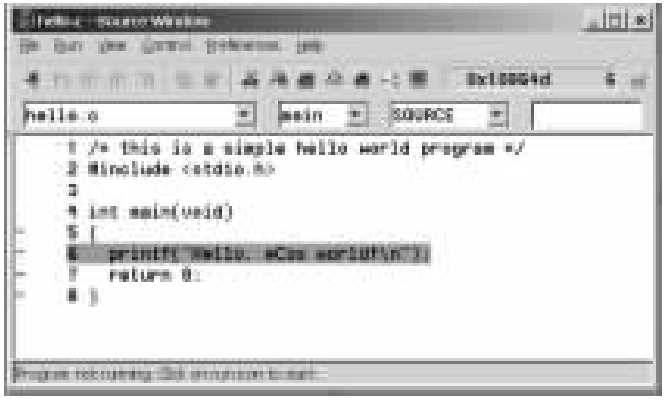


图 3-15 GDB 图形界面

在 GDB 界面的窗口菜单选择“File→Target Settings”，如图 3-16 所示，填入串口连接参数：

```
Target：Remote/Serial
Baud Rate：38400
Port：com1
```

然后再选择主窗口菜单选项“Run→Connect to target”与目标平台连接，选择“Run→Download”可以将执行文件 a.out 加载到目标系统上，选择“Run→Run”启动目标平台执行 a.out，并在 GDB 的控制台窗口显示执行结果“Hello，eCos World !”。

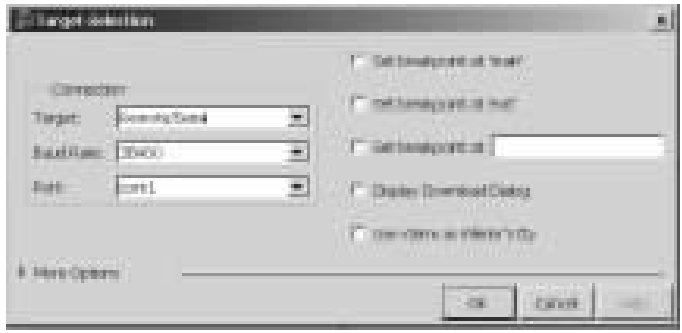


图 3-16　设置连接参数

如果要对程序进行调试，可以利用 GDB 图形界面进行设置，并执行单步、断点等操作，同时还可以查看目标平台的现场信息（寄存器、内存、变量等）。

3.4.3　多线程编程例子

上一节介绍的 hello.c 是一个非常简单的 eCos 应用程序例子，它仅仅是作为一个演示程序来说明 eCos 应用程序的编译、加载、运行以及调试过程。然而，实际应用要比它复杂得多。eCos 是一个支持多线程的系统，本节将介绍一个具有两个线程的 eCos 应用程序。这个例子程序将使用正常的 eCos 程序入口函数 cyg_user_start()，而不是使用 hello.c 中的 main()。虽然可以使用 main() 函数作为应用程序的入口点，但如果 eCos 在配置时没有选择 ISO C 软件包，main() 函数将不存在。

该例子程序位于 eCos 源码中的 examples 目录，程序名为 twothreads.c：

```
#include <cyg/kernel/kapi.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
/* now declare (and allocate space for) some kernel objects, like the two threads we will use */
cyg_thread thread_s[2];/* space for two thread objects */
char stack[2][4096];        /* space for two 4K stacks */
/* now the handles for the threads */
cyg_handle_t simple_threadA, simple_threadB;
/* and now variables for the procedure which is the thread */
cyg_thread_entry_t simple_program;
/* and now a mutex to protect calls to the C library */
cyg_mutex_t cliblock;
/* we install our own startup routine which sets up threads */
void cyg_user_start(void)
{
```

```
    printf ("Entering twothreads' cyg_user_start() function\n");
    cyg_mutex_init (&cliblock);
    cyg_thread_create (4, simple_program, (cyg_addrword_t) 0,
                       "Thread A", (void *) stack[0], 4096,
                       &simple_threadA, &thread_s[0]);
    cyg_thread_create (4, simple_program, (cyg_addrword_t) 1,
                       "Thread B", (void *) stack[1], 4096,
                       &simple_threadB, &thread_s[1]);
    cyg_thread_resume (simple_threadA);
    cyg_thread_resume (simple_threadB);
}
/* this is a simple program which runs in a thread */
void simple_program (cyg_addrword_t data)
{
    int message = (int) data;
    int delay;
    printf ("Beginning execution; thread data is %d\n", message);
    cyg_thread_delay (200);
    for (;;)
    {
      delay = 200 + (rand() % 50);
      /* note: printf() must be protected by a
         call to cyg_mutex_lock() */
      cyg_mutex_lock (&cliblock); {
          printf ("Thread %d: and now a delay of %d clock ticks\n",
                  message, delay);
      }
      cyg_mutex_unlock (&cliblock);
      cyg_thread_delay (delay);
    }
}
```

该程序产生两个线程 A 和 B，两个线程都进入循环状态。在循环体内，线程使用 cyg_thread_delay() **函数处于睡眠状态，睡眠时间为随机数，其流程图如图 3-17 所示。**

按照上一节 hello **例子程序的方法对** twothread.c **进行编译，使用** GDB **将执行文件加载到目标平台并启动该程序的执行，其输出结果如下：**

```
Entering twothreads' cyg_user_start()
    function
    Beginning execution; thread data is 0
    Beginning execution; thread data is 1
    Thread 0: and now a delay of 240 clock ticks
    Thread 1: and now a delay of 225 clock ticks
    Thread 1: and now a delay of 234 clock ticks
```

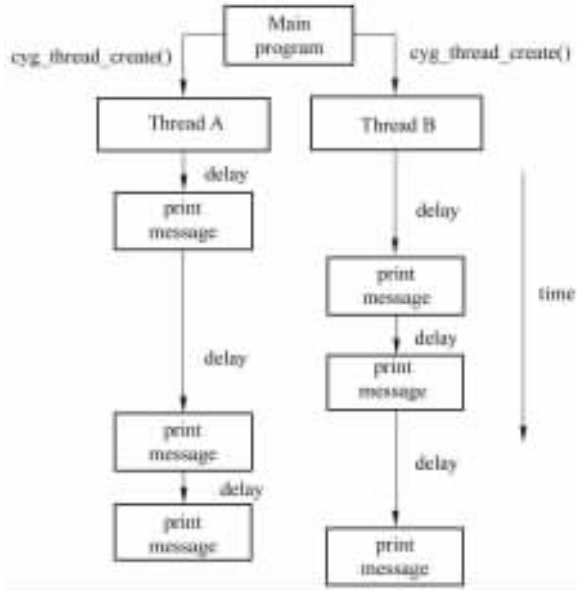


图 3-17　两线程例子程序流程图

```
Thread 0 : and now a delay of 231 clock ticks
Thread 1 : and now a delay of 224 clock ticks
Thread 0 : and now a delay of 249 clock ticks
Thread 1 : and now a delay of 202 clock ticks
Thread 0 : and now a delay of 235 clock ticks
......
```

程序中使用系统调用 cyg _ thread _ create () 函数产生了两个线程。此外，它还产生了一个互斥体 mutex 用于两个线程对 printf 函数调用的同步。这是因为在默认配置下，C 库标准 I/O 函数不具备线程安全性 (thread-safe)，这意味着当多个线程使用标准 I/O 函数时可能会出现冲突。使用互斥体可以解决这一问题，线程在 cyg _ mutex _ lock () 函数返回 (另一个线程调用 cyg _ mutex _ unlock () 函数释放该互斥体后) 之前不会调用 printf () 函数。

如果想在这种情况下避免使用互斥体，在配置 eCos 时必须将 C 库配置为具有线程安全性 (thread-safe)，其配置选项为 CYGSEM _ LIBC _ STDIO _ THREAD _ SAFE _ STREAMS。值得注意的是，如果 C 库具有线程安全性，那么在 cyg _ user _ start () 内不能使用 printf () 函数。

3.4.4　时钟和告警处理程序

如果应用程序要在给定的时间点或者周期性地执行某个任务，它可以使用循环的方式不停地检查实时时钟看是否已到达指定的时间。这种方式效率极低，它占用系统大量的资源。操作系统通常会提供一些系统调用，允许在指定的时间发生中断。eCos 提供了一组丰富的计时方式，包括计数器、时钟、告警以及定时器等。

这一节将以告警 (Alarm) 作为例子来说明如何使用这些计时机制。告警是在给定的时间点发生的事件，既可以是一次性的，也可以是周期性的。线程可以给告警指定一个告警处理函数，每当告警发生时都将调用该函数。

eCos 源码 examples 目录下的 simple-alarm.c 是一个简单的告警例子程序，该程序创建了一个产生告警的线程。test_alarm_func() 函数对告警进行处理，该告警处理函数对一个变量进行设置。主线程观察该变量值的变化，并输出消息。

```
/* this is a very simple program meant to demonstrate a basic use of time, alarms and alarm-handling
functions in eCos
*/
    #include <cyg/kernel/kapi.h>
    #include <stdio.h>
    #define NTHREADS 1
    #define STACKSIZE 4096

    static cyg_handle_t thread[NTHREADS];
    static cyg_thread thread_obj[NTHREADS];
    static char stack[NTHREADS][STACKSIZE];
    static void alarm_prog(cyg_addrword_t data);

/* we install our own startup routine which sets up threads and starts the scheduler */
    void cyg_user_start(void)
    {
      cyg_thread_create(4, alarm_prog, (cyg_addrword_t)0,
              "alarm_thread", (void *)stack[0],
              STACKSIZE, &thread[0], &thread_obj[0]);
      cyg_thread_resume(thread[0]);
    }

/* we need to declare the alarm handling function (which is defined below), so that we can pass it to
   cyg_alarm_initialize()
*/
    cyg_alarm_t test_alarm_func;

/* alarm_prog() is a thread which sets up an alarm which is then handled by test_alarm_func()
*/
    static void alarm_prog(cyg_addrword_t data)
    {
    cyg_handle_t test_counterH, system_clockH, test_alarmH;
    cyg_tick_count_t ticks;
    cyg_alarm test_alarm;
    unsigned how_many_alarms = 0, prev_alarms = 0, tmp_how_many;

    system_clockH = cyg_real_time_clock();
    cyg_clock_to_counter(system_clockH, &test_counterH);
    cyg_alarm_create(test_counterH, test_alarm_func,
```

```
                (cyg_addrword_t) &how_many_alarms,
                &test_alarmH, &test_alarm);
    cyg_alarm_initialize(test_alarmH, cyg_current_time() + 200, 200);

/* get in a loop in which we read the current time and print it out, just to have something scrolling by
*/
    for (;;) {
        ticks = cyg_current_time();
        printf("Time is %llu\n", ticks);
        /* note that we must lock access to how_many_alarms, since the alarm handler might change
        it. this involves using the annoying temporary variable tmp_how_many so that I can keep the
        critical region short
        */
        cyg_scheduler_lock();
        tmp_how_many = how_many_alarms;
        cyg_scheduler_unlock();
        if (prev_alarms != tmp_how_many) {
            printf(" --- alarm calls so far: %u\n", tmp_how_many);
            prev_alarms = tmp_how_many;
        }
        cyg_thread_delay(30);
    }
}

/* test_alarm_func() is invoked as an alarm handler, so it should be quick and simple. in this case it
increments the data that is passed to it.
*/

void test_alarm_func(cyg_handle_t alarmH, cyg_addrword_t data)
{
    ++*((unsigned *) data);
}
```

使用前面介绍的方法对其进行编译,通过 GDB 将可执行文件加载到目标平台并启动。执行结果如下:

```
Time is 0
Time is 30
Time is 60
Time is 90
Time is 120
Time is 150
Time is 180
Time is 210
```

```
--- alarm calls so far:1
    Time is 240
    Time is 270
    Time is 300
    Time is 330
    Time is 360
    Time is 390
    Time is 420
--- alarm calls so far:2
    Time is 450
    Time is 480
```

该例子程序中有几个值得注意的地方：

① **程序使用了** cyg _ real _ time _ clock **()函数，它返回默认的系统实时时钟句柄。**

② **告警以计数器为基础，因此** cyg _ alarm _ create **()函数使用了一个计数器的句柄。程序中使用了** cyg _ clock _ to _ counter **()函数将时钟转换为计数器。**

③ **一旦告警产生，**cyg _ alarm _ initialize **()函数将其进行初始化，它对告警发生的时间以及周期进行设置。**

④ **告警处理** test _ alarm _ func **()函数遵守告警处理程序和其他滞后服务程序的规则：不调用任何可能锁定调度器的函数。**

⑤ **该程序有一个临界区：主线程对变量** how _ many _ alarms **进行访问，而告警处理程序要对该变量进行修改，因此有可能发生对该变量的竞争。为解决这种竞争问题，主线程通过调用** cyg _ scheduler _ lock **()函数和** cyg _ scheduler _ unlock **()函数进行保护，当调度器被锁定的时候，将不会运行告警线程。**

第 4 章　RedBoot

RedBoot 是 Red Hat Embedded Debug and Bootstrap 的英文缩写，它是 Red Hat 的一个标准嵌入式系统引导和 Debug 环境。RedBoot 替代了它的前期 Debug Firmware 产品 CygMon 和 GDB，它为一定范围内的嵌入式操作系统（如嵌入式 Linux、eCos 等）提供了一个完整的引导环境。它还包含了一些其他功能，如网络加载和网络 Debug 等。另外，RedBoot 还为引导映像提供了一个简单的 flash 文件系统。

4.1　功能与应用

RedBoot 为嵌入式目标系统程序的加载和执行控制提供了许多工具，还提供了对目标系统环境进行管理的一些工具。它可用于产品开发（具有 Debug 支持），也可用于最终产品配置（flash 引导和网络引导）。

RedBoot 的一些主要功能有：

① 支持引导脚本。

② 提供对 RedBoot 进行配置和管理的简单命令行界面，可通过串口和以太网进行访问。

③ 内建 GDB，用于与 host debuger 通过串口或网络（只限局域网内）进行连接与通信。

④ 属性配置。用户可以控制系统的一些属性，如系统日期、时间、默认的引导 Flash 映像、默认的 failsafe 映像、静态 IP 地址等等。

⑤ 具有可配置性和可扩展性，对具体的目标系统环境具有很好的适应能力。

⑥ 支持网络引导，可以通过 BOOTP、DHCP 和 TFTP 等协议对系统进行设置并加载程序。

⑦ X/Y Modem 支持，可以通过串口加载映像文件。

⑧ 加电自测试。

RedBoot 尽管起源于 Red Hat eCos，但它可以作为一种通用的系统 debug 和引导控制软件，适用于任何嵌入式系统和操作系统。举例来说，适当增加 RedBoot 的一些功能就可以替代 PC 机上普遍使用的 BIOS。当前 Red Hat 正在将 RedBoot 作为一种标准引导程序安装在所有嵌入式平台上，Red Hat 嵌入式 Linux 和 eCos 都将 RedBoot 作为它的一个组成部分。

4.1.1　RedBoot 的安装

当目标系统初始加电时，RedBoot 通常从目标平台的 flash 引导区或引导 ROM 中开始运行。RedBoot 也支持其他的运行方式，如 X86 PC 可以从软盘运行 RedBoot。不同的目标系统可能有不同的方法将 RedBoot 映像文件写入 Flash ROM 或 ROM。一些目标系统平台在出厂时可能已经安装了 RedBoot。如果没有安装 RedBoot，就必须自己将 RedBoot 写入到目标平台的 Flash ROM 或 ROM 内。RedBoot 的安装方法可以参考具体目标系统平台的相应说明。一旦 RedBoot 安装完毕，可以使用 fconfig 命令进行具体的一些配置。

4.1.2 RedBoot 用户界面

RedBoot 提供了一个命令行方式的用户界面 (CLI)。在最小配置下,用户接口通常由串口提供。如果有多个串口,Redboot 一般选择任意一个串口作为用户接口,并独占该端口。如果平台具有网络通信能力,Redboot 也可以通过网络端口使用 telnet 协议提供用户接口 CLI。默认情况下,Redboot 的 telnet 使用端口 TCP/9000,用户可以对其进行配置和设置。

Redboot 包含一组支持 GDB 远程协议的 GDB 程序。用户界面命令行的第一个字符" $ "出现的时候将自动调用 GDB 服务。不管采用串口连接还是网络连接的连接方式,在 GDB 明确指定进行脱连操作之前 GDB 都是可用的。在以网络方式连接时,用户程序必须注意对网络连接的保护。

4.1.3 RedBoot 环境配置

一般情况下,RedBoot 都可以正常运行。但在某些情况下必须进行相应的配置,这种配置主要根据系统是否支持 Flash 以及是否具有网络支持能力来进行。

1. 目标系统的网络配置

网络系统中的每一个节点都需要一个惟一的地址。由于 RedBoot 支持 TCP/IP 协议,因此需要使用 IP 地址。RedBoot 使用两种 IP 地址设置方式。一种是静态 IP 地址,它由用户指定并存放在目标系统上。另一种方式是动态 IP 地址,RedBoot 使用 BOOTP 协议 (DHCP 的一个子集) 从网络服务器上获取动态 IP 地址。

IP 地址方式的选择由 RedBoot 命令 fconfig 来进行。对 RedBoot 的配置完成以后,配置信息将保存在 Flash 内存内。RedBoot 只在系统复位时获取这些配置信息,如果改变了配置信息,这些被改变的配置只有在系统重启后才有效。

利用 fconfig 进行配置的例子如下:

```
RedBoot> fconfig -l
Run script at boot: false
Use BOOTP for network configuration: false
Local IP address: 192.168.1.29
Default server IP address: 192.168.1.101
GDB connection port: 9000
Network debug at boot time: false
```

在上面的例子中,IP 地址采用静态分配的方式,被指定为 192.168.1.29。"Default server IP address"指定提供 TFTP 服务的服务器地址,即 192.168.1.101。TFTP 服务器地址在 TFTP 命令中可以单独指定。

"Use BOOTP for network configuration"选项用于指定是否使用 BOOTP 协议。如果该项设置为 true,则采用动态 IP 地址方式,IP 地址在系统启动时通过 BOOTP 协议获取。不管采用哪种 IP 地址方式,都需要对"GDB connection port"进行设置,这是 RedBoot 用于接收命令的 TCP 连接端口。这种连接可用于 GDB,也可用于一般的 RedBoot 命令。下面是在 Linux 下使用 telnet 协议与 RedBoot 进行通信的例子:

```
% telnet redboot _ board 9000
Connected to redboot _ board
Escape character is '^]'.
RedBoot>
```

2. Host 主机系统的网络设置

RedBoot 要求 Host 主机系统提供两种不同的网络服务，即：

① 使用 BOOTP 协议的动态 IP 地址分配。

② 用于文件加载的 TFTP 访问。

在某些系统中，上述服务需要使用手工配置。下面以 Red Hat Linux 为例介绍其设置方法。

在 RedHat Linux 6.2 设置 TFTP：

1）确认已经安装 TFTP 服务 RPM 包。

2）编辑控制文件 /etc/inted.conf，找到下面这样的行并将其前面的注释符去掉：

```
tftp dgram udp wait root /usr/sbin/tcpd /usr/sbin/in.tftpd
```

3）重新启动 inetd server：

```
service inet reload
```

在 Red Hat Linux 7.x 设置 TFTP：

1）确认 xinetd RPM 已经安装。

2）确认 TFTP RPM 已经安装。

3）使能 TFTP：

```
/sbin/chkconfig tftp on
```

4）重新加载 inetd 配置：

```
/sbin/service xinetd reload
```

5）创建目录/tftpboot：

```
mkdir /tftpboot
```

在 Red Hat Linux 系统中启动 BOOTP/DHCP 服务：

1）确认已经安装 DHCP 包（不是 DHCPD）。

2）配置 DHCP 服务器。根据目标系统以太网 MAC 地址对 DHCP 配置文件/etc/dhcpd.conf 进行设置。下面是配置文件的一个例子，目标系统以太网 MAC 地址为 08:00:3E:28:79:B8。详细设置过程参考 DHCP 说明。

```
---------- /etc/dhcpd.conf ----------
    default-lease-time 600;
    max-lease-time 7200;
    option subnet-mask 255.255.255.0;
    option broadcast-address 192.168.1.255;
    option domain-name-servers 198.41.0.4, 128.9.0.107;
```

```
option domain-name "bogus.com";
allow bootp;
shared-network BOGUS {
subnet 192.168.1.0 netmask 255.255.255.0 {
option routers 192.168.1.101;
range 192.168.1.1 192.168.1.254;
}
}
host mbx {
hardware ethernet 08:00:3E:28:79:B8;
fixed-address 192.168.1.20;
filename "/tftpboot/192.168.1.21/zImage";
default-lease-time -1;
server-name "srvr.bugus.com";
server-identifier 192.168.1.101;
option host-name "mbx";
}
```

3）**启动** DHCP **服务：**

```
service dhcpd start
```

一旦网络设置完成，在重启系统后，就可以在 Host **系统上用** ping **命令来测试与目标系统的网络通信状态。如果网络通信正常，可以尝试在目标系统上使用** RedBoot **的文件加载命令从** Host **下载文件。**

4.2 RedBoot 命令

RedBoot **提供三类基本命令：**

① **加载程序和执行命令。**

② flash **映像和配置管理命令。**

③ **其他命令。**

4.2.1 基本命令格式

RedBoot **命令的基本格式是：**

```
RedBoot> COMMAND [-S] [-s val] operand
```

其中：

[-S]：**可选开关。该选项用于指定某一特定行为的发生。如：**

```
RedBoot> fis init -f
```

这里的-f **开关指出要进行全部的文件系统初始化操作。**

[-s val]：**带参数的可选开关。如：**

```
RedBoot> load -b 0x00100000 data_file
```

指明加载一个文件(通过 TFTP)到内存区,起始地址为 0x00100000.

operand 操作数。RedBoot 的某些命令中必须有一个操作数。如:

```
RedBoot> go 0x10044
```

该例子用于执行起始地址为 0x10044 的程序。

在 RedBoot 操作界面下,通过 help 命令可以获得该系统所支持的详细命令列表,包括其语法。

例如:

```
RedBoot> help
Manage aliases kept in FLASH memory
alias name [value]
Set/Query the system console baud rate
baudrate [-b <rate>]
Manage machine caches
cache [ON | OFF]
Display/switch console channel
channel [-l|<channel number>]
Display disk partitions
disks
Display (hex dump) a range of memory
dump -b <location> [-l <length>]
Manage flash images
fis {cmds}
Manage configuration kept in FLASH memory
fconfig [-i] [-l] [-n] [-f] | nickname [value]
Execute code at a location
go [-w <timeout>] [entry]
Help about help?
help [<topic>]
Load a file
load [-r] [-v] [-d] [-h <host>] [-m {TFTP | xyzMODEM | disk}] [-b <base_address>] <file_
name>
Network connectivity test
ping [-v] [-n <count>] [-t <timeout>] [-i <IP_addr] -h <IP_addr>
Reset the system
reset
Display RedBoot version information
version
```

命令可以使用缩写形式。上面例子中,d、du、dum 和 dump 都是有效的 dump 命令。fconfig 命令可以用 fc 的缩写,但不能用 f,因为这样会与命令 fis 混淆。

另外有一个特殊的命令。当 RedBoot 检测到命令中的第一个字符为 $ 时,它将转到 GDB 方式。此时将进入 eCos GDB 程序,允许与 GDB 主机进行通信。如果要从 GDB 方式返回到 RedBoot,必须重启目标系统。

对于具体的目标系统,可以根据需要对其命令进行扩展。

4.2.2 RedBoot 普通命令

命令格式:

```
command <options, paraments>
```

命令元素之间必须用空格键进行分离。数字 (如内存地址) 可以用十进制和十六进制 (需用 0x 的前缀),命令可以用单一的缩写形式。

ping 检查网络连接命令

```
ping [-v] [-n <count>] [-l <length>] [-t <timeouts>] [-r <rate>] [-i <IP_addr>] -h <IP_addr>
```

该命令对网络连接进行检查。通过本地网络发送特定的包到主机,主机自动返回这些包。参数如下:

-v	详细显示包的信息。
-n <count>	包的数目,默认为 10。
-t <timeout>	超时时间 (ms),默认为 1000ms。
-r <rate>	包的发送速度,即包连续发送的时间间隔,默认为 1000ms。"-r 0"指明尽可能快地进行包的发送。
-l <length>	包的数据长度,默认为 64B,最大为 1400B。
-h <host IP>	网络连接另一端的 IP 地址。

Alias name [value] 别名命令

别名命令用于将长的表达式用短的名字替代。别名被保存在 RedBoot 的非易失性配置区内 (Flash Rom)。在命令行或命令脚本中用 % {name} 的形式引用该别名。

例如,为"-b 0x1000000"设置一个别名"SBUF":

```
RedBoot> alias SBUF "-b 0x100000"
Update RedBoot non-volatile configuration - are you sure (y/n) ?y
... Unlock from 0x50f80000-0x50fc0000:.
... Erase from 0x50f80000-0x50fc0000:.
... Program from 0x0000b9e8-0x0000c9e8 at 0x50f80000:.
... Lock from 0x50f80000-0x50fc0000:.
```

可以使用下面的方法查看和引用该别名:

```
RedBoot> alias SBUF
'SBUF' = '-b 0x100000'
RedBoot> d % {SBUF}
```

```
0x00100000：FE03 00EA 0000 0000 0000 0000 0000 0000 |................|
0x00100010：0000 0000 0000 0000 0000 0000 0000 0000 |................|
```

baudrate [-b value] 设置串口波特率命令

设置串口通信波特率。如果目标平台支持非易失性配置数据，则保存此次设定的值，下次系统复位后还将使用此次设定的波特率。

cache [ON | OFF] Cache 操作命令

该命令对处理器的 Cache 进行操作。未使用选项 ON 或 OFF 时，该命令对系统 Cache 状态进行描述。当使用选项时，将执行关闭 Cache (OFF) 或使能 Cache (ON) 的操作。

channel [-l | <channel number>] 控制台通道命令

没有参数时，该命令显示当前控制台通道号。

当带一个大于 0 (包括 0) 的参数时，该命令将控制台通道转换到指定的通道号。

当参数为-l 时，该命令使 RedBoot 响应最先收到输入信息的通道。

cksum -b <location> -l <length> 计算校验和命令

计算指定内存范围 (RAM 或 FLASH) 的 POSIX 校验和。

disk 显示磁盘分区命令

使用该命令可以列举出 RedBoot 可以识别的磁盘分区。

dump -b <location> [-l <length>] 内存 dump 命令

显示指定区域的内存数据 (16 进制)。如

```
RedBoot> du -b 0x100 -l 0x80
0x00000100：3C60 0004 6063 2000 7C68 03A6 4E80 0020 |<'.. 'c .|h..N.. |
0x00000110：0000 0000 0000 0000 0000 0000 0000 0000 |................|
0x00000120：0000 0000 0000 0000 0000 0000 0000 0000 |................|
0x00000130：0000 0000 0000 0000 0000 0000 0000 0000 |................|
0x00000140：0000 0000 0000 0000 0000 0000 0000 0000 |................|
0x00000150：0000 0000 0000 0000 0000 0000 0000 0000 |................|
0x00000160：0000 0000 0000 0000 0000 0000 0000 0000 |................|
0x00000170：0000 0000 0000 0000 0000 0000 0000 0000 |................|

RedBoot> d -b 0xfe00b000 -l 0x80
0xFE00B000：2025 700A 0000 0000 4174 7465 6D70 7420 | %p.....Attempt |
0xFE00B010：746F 206C 6F61 6420 532D 7265 636F 7264 |to load S-record|
0xFE00B020：2064 6174 6120 746F 2061 6464 7265 7373 | data to address|
0xFE00B030：3A20 2570 205B 6E6F 7420 696E 2052 414D |：%p [not in RAM|
0xFE00B040：5D0A 0000 2A2A 2A20 5761 726E 696E 6721 |]...*** Warning!|
0xFE00B050：2043 6865 636B 7375 6D20 6661 696C 7572 | Checksum failur|
0xFE00B060：6520 2D20 4164 6472 3A20 256C 782C 2025 |e - Addr：%lx，%|
```

```
0xFE00B070：3032 6C58 203C 3E20 2530 326C 580A 0000 |02lX <> %02lX...|
0xFE00B080：456E 7472 7920 706F 696E 743A 2025 702C |Entry point：%p,|
```

reset 复位命令

系统复位。对于大多数的目标系统，该命令相当于加电复位。但在某些系统中，该命令只是跳转到复位程序，对系统重新进行初始化操作。

version 显示版本信息

显示 RedBoot 的版本信息。如：

```
RedBoot> version
RedBoot (tm) debug environment - built 09：12：03，Feb 12 2001
Platform：XYZ (PowerPC 860)
Copyright (C) 2000，2001，Red Hat，Inc.
RAM：0x00000000-0x00400000
RedBoot>
```

load 加载命令

该命令将数据加载到目标系统。可以通过网络连接使用 TFTP 协议或通过串口连接使用 X/Y modem 协议进行数据加载操作。文件可以直接从本地磁盘加载，文件格式可以是可执行的映像文件（SREC 格式），也可以是纯数据格式。

命令格式：

```
load [file] [-v] [-d] [-b location] [-r] [-m {[xmodem] | [ymodem] |
    [tftp] | [disk]}] [-h host _ IP _ address]
```

参数：

file 位于 TFTP 服务器或本地磁盘上的文件名。TFTP 的文件名格式请参考具体的 TFTP 服务器说明。本地磁盘文件名格式为 disk：filename 。磁盘分区必须与 disk 命令所列出的磁盘分区相符。

-v 显示加载过程。串行加载时该参数无意义。

-d 在加载过程中对被压缩的映像进行解压缩操作。

-b 指定文件加载的起始位置。

-r 加载纯数据。如果出现该选项，必须使用-b 选项。

-m 选择加载方式。其选项有：

- xmodem。
- ymodem。

  上面两个参数通过控制台串口使用标准协议进行串行加载。使用该选项时，不需要 file 参数。
- tftp 使用 TFTP 协议进行网络加载。
- disk 从本地磁盘加载。

-h 指明用于加载的主机名。只用于 TFTP 方式。

例如：

```
RedBoot> lo redboot.ROM -b 0x8c400000
Address offset = 0x0c400000
Entry point：0x80000000，address range：0x80000000-0x8000fe80
```

4.2.3 Flash 映像系统（FIS）

如果平台具有 Flash 内存，RedBoot 可以用其保存映像文件。可以使用简单的文件保存命令将执行映像以及数据保存在 Flash 内存里。fis 命令用于 flash 映像的操作和管理。

下面介绍 fis 的相关命令。

fis init [-f]

该命令用于初始化 flash 映像系统 FIS。该命令仅在硬件首次安装 RedBoot 时执行一次。如果再次执行该命令，将会造成 Flash 数据的丢失。如果使用-f 选项，所有的 Flash 数据都将被擦除。

例如：

```
RedBoot> fis init -f
About to initialize [format] flash image system - are you sure (y/n) ?n
```

fis [-c] [-d] list

该命令列出 FIS 中当前所有可用的映像。某些 RedBoot 使用的映像具有固定的名字，其他映像由用户指定。其使用方法见下例。如果使用-c 选项，在 Mem Addr 域上将显示该映像的校验和。如果使用-d 选项，将在 length（Flash 使用长度）显示数据长度 datalength。datalength 是指定的 Flash 映像实际使用的数据长度。

例如：

```
RedBoot> fis list
Name              flash addr   Mem addr     Length       Entry point
RedBoot           0xA0000000   0xA0000000   0x020000     0x80000000
RedBoot [backup]  0xA0020000   0x8C010000   0x010000     0x8C010000
RedBootconfig 0xA0FC0000   0xA0FC0000   0x020000   0x00000000
FIS directory  0xA0FE0000   0xA0FE0000   0x020000   0x00000000
RedBoot> fis list -c
Name              flash addr   Checksum     Length       Entry point
RedBoot           0xA0000000   0x34C94A57   0x020000     0x80000000
RedBoot [backup]  0xA0020000   0x00000000   0x010000     0x8C010000
RedBoot config    0xA0FC0000   0x00000000   0x020000     0x00000000
RedBoot config    0xA0FE0000   0x00000000   0x020000     0x00000000
```

fis free

该命令显示当前未使用的 Flash 内存区域。

例如：

```
RedBoot> fis free
0xA0040000 .. 0xA07C0000
0xA0840000 .. 0xA0FC0000
```

fis create

命令格式：

```
fis create -b <mem_base> -l <length> [-f <flash_addr>] [-e <entry_point>]
        [-r <ram_addr>] [-s <data_length>] [-n] <name>
```

该命令在 FIS **目录中产生一个映像。映像数据必须在** RAM **中已经存在。通常可以首先用** load **命令将映像加载到** RAM**，然后再使用** fis create **命令将其写入** Flash**。**

参数：

name	**将在** FIS **目录中出现的文件名。**
-b	**映像在** RAM **中的位置，该选项是必需的。**
-l	**映像长度。如果映像已经存在，则其长度由以前产生的映像得出。如果指定了长度并且映像已经存在，则其长度必须与原来的长度相符。**
-f	**映像在** Flash **中的位置。如果没有指定，将根据已有映像进行确定。**
-e	**执行点入口地址。用于映像的起始地址未知或者需要被覆盖的情况下。**
-r	**当通过** fis load **加载映像时的** RAM **地址。该选项只用于使用** fis load **命令加载的映像。固定映像如** RedBoot **等不需要该参数。**
-s	**写入** Flash **的实际数据长度。如果没有指定，则采用映像长度 (**-l **参数)。如果**-s **指定的值小于**-l **指定的值，**Flash **内映像的其余部分将保持在擦除状态。该选项的使用可以产生一个全空的** Flash **映像。例如，可以用其为应用程序 (非** RedBoot**) 保留一定的空间。**
-n	**如果使用**-n **参数，则只更新** FIS **目录，不会有任何数据从** RAM **复制到** Flash**。该参数可以用于** FIS **被破坏时的恢复操作。**

命令举例：

```
RedBoot> fis create RedBoot -f 0xa0000000 -b 0x8c400000 -l 0x20000
An image named 'RedBoot' exists - are you sure (y/n) ?n
RedBoot> fis create junk -b 0x8c400000 -l 0x20000
... Erase from 0xa0040000-0xa0060000:.
... Program from 0x8c400000-0x8c420000 at 0xa0040000:.
... Erase from 0xa0fe0000-0xa1000000:.
... Program from 0x8c7d0000-0x8c7f0000 at 0xa0fe0000:.
```

fis load

命令格式：

```
fis load [-b <memory load address>] [-c] [-d] name
```

该命令将映像从 Flash **传送到** RAM**。当命令执行完成后，可以使用** go **命令执行该映像。**

如果使用-b 参数,则将指定的映像从 Flash 加载到指定地址的 RAM。如果没有使用-b 参数,则映像被加载到该映像产生时所给定的加载地址。

参数:

name　FIS 中的文件名。
-b　指定文件加载到内存的起始地址。
-c　在映像被加载到内存后,计算并输出映像数据的校验和。
-d　在映像从 Flash 加载到 RAM 时执行解压操作

命令举例:

```
RedBoot> fis load RedBoot [backup]
RedBoot> go
```

fis delete name

该命令从 FIS 中删除一个映像。该命令在执行时将擦除 Flash,并从 FIS 目录中删除该映像名字。

命令举例:

```
RedBoot> fis list
Name              flash addr     Mem addr       Length     Entry point
RedBoot           0xA0000000     0xA0000000     0x020000   0x80000000
RedBoot [backup]  0xA0020000     0x8C010000     0x020000   0x8C010000
RedBoot config    0xA0FC0000     0xA0FC0000     0x020000   0x00000000
FIS directory     0xA0FE0000     0xA0FE0000     0x020000   0x00000000
junk              0xA0040000     0x8C400000     0x020000   0x80000000
RedBoot> fis delete junk
Delete image 'junk' - are you sure (y/n) ?y
... Erase from 0xa0040000-0xa0060000:.
... Erase from 0xa0fe0000-0xa1000000:.
... Program from 0x8c7d0000-0x8c7f0000 at 0xa0fe0000:.
```

fis lock

命令格式:

```
fis lock -f <flash_addr> -l <length>
```

该命令对部分 Flash 进行写保护(lock)。如果要修改该部分 Flash,必须首先使用 fis unlock 命令解除写保护。该命令是可选的,只实现在支持 Flash 写保护的硬件平台中。

命令举例:

```
RedBoot> fis lock -f 0xa0040000 -l 0x20000
... Lock from 0xa0040000-0xa0060000:.
```

fis unlock

命令格式:

fis unlock -f <flash _ addr> -l <length>

该命令对被锁定的 Flash 进行解锁,使该部分 Flash 可以被更新。

命令举例:

RedBoot> fis unlock -f 0xa0040000 -l 0x20000
... Unlock from 0xa0040000-0xa0060000:.

fis erase

命令格式:

fis erase -f <flash _ addr> -l <length>

该命令强制擦除指定的部分 Flash。

命令举例:

RedBoot> fis erase -f 0xa0040000 -l 0x20000
... Erase from 0xa0040000-0xa0060000:.

fis write

fis write -b <location> -l <length> -f <flash addr>

该命令将指定范围的 RAM 数据写入指定的 Flash。

4.2.4 Flash 内配置信息的管理

RedBoot 提供对 Flash 管理的支持,可以对存储在 Flash 的多个可执行映像以及一些非易失性信息 (如 IP 地址等一些网络信息) 进行管理。使用 fconfig 命令,可以显示或修改保存在 Flash 内的这些信息。需要注意的是,只有在支持 Flash 映像系统的平台中才支持 fconfig 命令。

命令格式:

fconfig [-i] [-l] [-n] [-f] | nickname [value]

如果使用-i 选项,则配置数据将被复位,并使用默认值。

如果使用-l 选项,则只简单地列举出配置数据。否则,每个配置参数将依次出现,并可对其进行编辑修改。单独的回车键不会改变参数值,布尔变量值用 t (Ture) 或 f (False) 输入,使用"^"键可以回到上一个项目的编辑。任何时候都可以使用句点 (.) 来停止 fconfig 的编辑过程。如果改变了配置信息,在提示用户之后将更新后的数据写回 Flash。

如果使用-n 选项 (有或无-l),则要用到"nicknames"。使用 nicknames 使名字变得更简练。如果要使用全名,则应该加上-f 标志。

nicknames 的设置可用如下命令格式:

RedBoot> fconfig nickname value

如果没有指定 value,该命令将只列举出该项。如果指定了 value,则该项将被设置为 value 值。例如:

```
RedBoot> fconfig -l -n
boot _ script:false
bootp:false
bootp _ my _ ip:10.16.19.176
bootp _ server _ ip:10.16.19.66
gdb _ port:9000
net _ debug:false

RedBoot> fconfig bootp _ my _ ip 10.16.19.177
bootp _ my _ ip:10.16.19.176 Setting to 10.16.19.177
Update RedBoot non-volatile configuration - are you sure (y/n)?y
... Unlock from 0x507c0000-0x507e0000:.
... Erase from 0x507c0000-0x507e0000:.
... Program from 0x0000a8d0-0x0000acd0 at 0x507c0000:.
... Lock from 0x507c0000-0x507e0000:.
RedBoot>
```

使用 fconfig **命令可以设置一段在启动时运行的脚本 (**Script**) 命令。这一段** Script **脚本由一组系统启动时运行的** RedBoot **命令组成,在使用** fconfig **进行设置时还可以对** Script **命令的运行进行超时设置。例如:**

```
RedBoot> fconfig
Run script at boot:false t
Boot script:
Enter script, terminate with empty line
>> fi li
Boot script timeout:0 10
Use BOOTP for network configuration:false .
Update RedBoot non-volatile configuration - are you sure (y/n)?y
... Erase from 0xa0fc0000-0xa0fe0000:.
... Program from 0x8c021f60-0x8c022360 at 0xa0fc0000:.
RedBoot>
RedBoot (tm) debug environment - built 08:22:24, Aug 23 2000
Copyright (C) 2000, Red Hat, Inc.
RAM:0x8c000000-0x8c800000
flash:0xa0000000 - 0xa1000000, 128 blocks of 0x00020000 bytes ea.
Socket Communications, Inc:Low Power Ethernet CF Revision C \
5V/3.3V 08/27/98 IP:192.168.1.29, Default server:192.168.1.101 \
== Executing boot script in 10 seconds - enter ^C to abort
RedBoot> fi li
Name                  flash addr   Mem addr     Length     Entry point
RedBoot               0xA0000000   0xA0000000   0x020000   0x80000000
RedBoot [backup]      0xA0020000   0x8C010000   0x020000   0x8C010000
RedBoot config        0xA0FC0000   0xA0FC0000   0x020000   0x00000000
```

```
FIS directory          0xA0FE0000  0xA0FE0000   0x020000    0x00000000
RedBoot>
```

上面例子中,粗体字是从控制台输入的字符。最后执行的 fi li 命令不是由控制台输入的,而是来自 Script 命令。

4.2.5 RedBoot 程序执行控制

一旦通过 load 命令或者 fis load 命令将映像加载到内存后,就可以启动该映像的执行。执行命令如下。

go 程序执行命令
命令格式:

```
RedBoot> go [-w time] [location]
```

参数:

-w	指定程序执行前的等待时间。通常用在启动 Script 脚本中。
location	程序执行的起始地址。如果没有指定,则从上一次映像加载的入口地址处开始执行。

exec 执行 Linux 内核映像
该命令不支持所有平台,通常用于执行非 eCos 应用程序,如 Linux 内核。
命令格式:

```
RedBoot> exec [-w time] [-b <load addr> [length]] [-r <ramdisk addr> [-s
        <ramdisk length>]] [-c "kernel command line"] [<entry_point>]
```

参数:

-w	程序执行前的等待时间。
-b	内核程序在内存的位置。
-r -s	给内核传递 ramdisk 的位置参数。
-c	给内核传递文本方式的命令行信息。

4.3 RedBoot 的配置与编译

eCos 的源码中包含了 RedBoot 源码。如果对 RedBoot 的命令进行了扩展,或者对 RedBoot 的源码作了增加或修改,则必须重新对 RedBoot 进行编译。大多数平台的硬件抽象层 HAL 都提供了配置导出文件,使用该文件可以简化系统的开发过程。RedBoot 作为 eCos 的一部分,其配置和编译过程都使用了基于组件定义语言 CDL 的配置技术。

RedBoot 的配置和编译过程与 eCos 应用程序类似,既可以使用 eCos 图形配置工具,也可以使用命令行配置工具 ecosconfig。

4.3.1 RedBoot 软件结构

图 4-1 是 RedBoot 的软件结构图,图中列举了它的主要组成模块。RedBoot 的软件结构具

有可伸缩性，根据具体需要可以增加新的软件模块或新的功能，也可以将一些不需要和不适用的软件模块或功能进行裁减。RedBoot 使用的底层程序是 eCos 的硬件抽象层 HAL 和设备驱动程序。

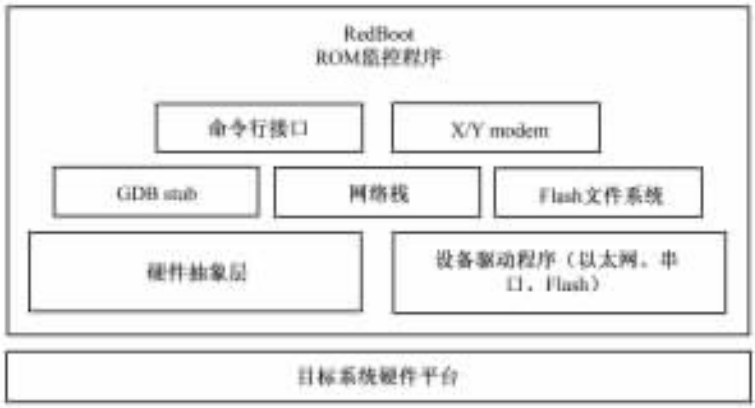


图 4-1　RedBoot 软件结构图

所有 RedBoot 源码都在 packages \ redboot 目录下。表 4-1 为 RedBoot 源码目录。

表 4-1　RedBoot 源码目录

目　录	说　明
cdl	包含 CDL 脚本，详细描述了 RedBoot 配置和编译信息。配置工具使用 CDL 文件来了解 RedBoot 源码
include	RedBoot 编译所需头文件目录
include \ fs	RedBoot 文件系统头文件
include \ net	RedBoot 网络支持头文件
misc	包含 eCos 最小配置文件（.ecm），被导入到配置工具形成最基本配置
src	RedBoot 主要源程序，包括启动、Flash 内存、I/O 通信、命令行解析等程序
src \ fs	RedBoot 文件系统源程序
src \ net	RedBoot 网络支持源程序，包括 BOOTP 和 TFTP 源程序

4.3.2　使用 eCos 图形配置工具

在 Windows 和 Linux 环境下都可以使用 eCos 图形配置工具对 RedBoot 进行配置和编译。下面以“i386 pc target”作为硬件平台，描述 RedBoot 的配置与编译过程。

1）运行 eCos 图形配置工具。

2）在配置工具的“Build”菜单选择模板“Templates”，在出现的模板对话框中选择硬件平台（Hardware）“i386 PC target”和模板软件包（Packages）“RedBoot”，如图 4-2 所示。

3）根据需要增加或删减软件模块。在配置工具的“Build”菜单选择“Packages”，在 Packages 对话框中增加或删除相应的软件模块（本例中不作任何操作），如图 4-3 所示。

图 4-2　使用 eCos 配置工具选择 RedBoot 硬件平台和软件模板

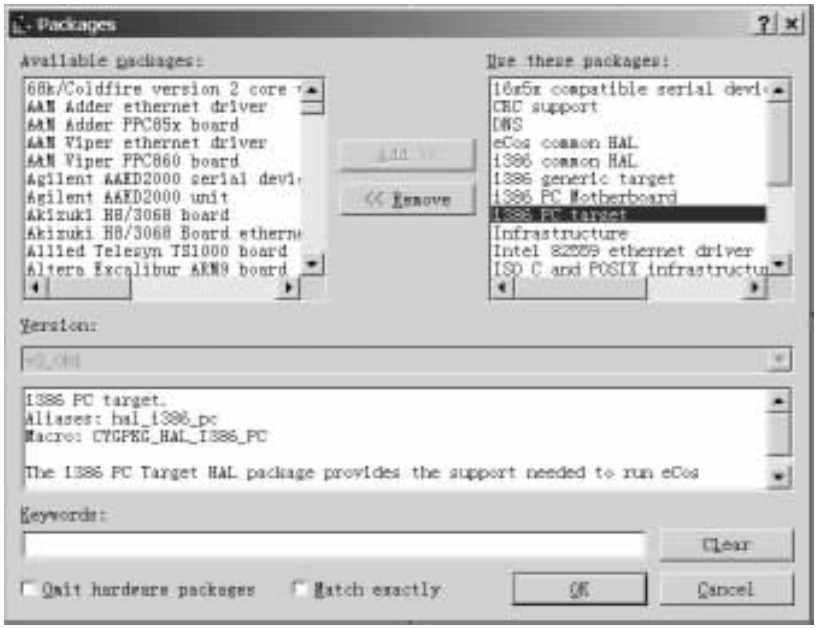


图 4-3　增加或删除软件模块

4）**对 RedBoot 选项进行配置，如图 4-4 所示。由于 RedBoot 还需要其他一些软件模块，如硬件抽象层 HAL 和设备驱动程序等，因此除了对 RedBoot 相关选项进行配置外，还要根据具体平台的实际要求对 HAL 和设备驱动程序进行配置。例如，一个具有网络支持能力的平台需要在“Common Ethernet Support”包中使能以太网驱动程序，同时还需要对 RedBoot 的网络配置选项进行设置，包括对目标平台默认的 IP 地址进行配置。**

在本例中，需要产生一个软盘引导的 RedBoot，所以在配置工具的“Configuration→eCos HAL→i386 architecture→i386 PC target→Startup type”选项选择“FLOPPY”，见图 4-4。

5）**保存配置。在进行编译前，需要对已经完成的配置进行保存。在配置工具的“File”菜单选择“Save as”，将其保存为 RedBoot _ Floppy. ecc。此时将产生 RedBoot 的工作目录，这些目录与 RedBoot _ Floppy. ecc 处于同一目录下。表 4-2 为工作目录结构。**

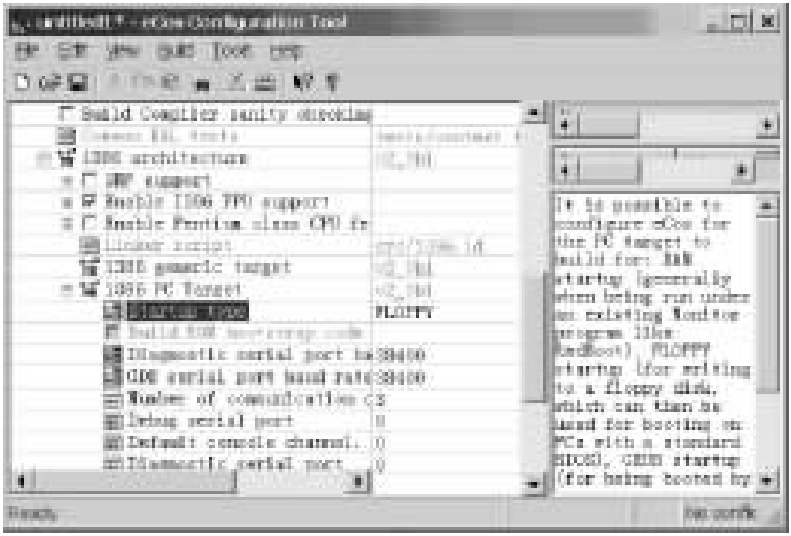


图 4-4　对 RedBoot 进行配置

表 4-2　RedBoot 工作目录结构

目　录	说　明
Redboot _ floppy _ build	包含 RedBoot 编译过程中产生的所有文件
Redboot _ floppy _ install	RedBoot 编译后的输出文件
Redboot _ floppy _ install \ bin	各种 RedBoot 映像文件。所产生的文件类型根据处理器的不同而有所不同。文件格式包括二进制格式、ELF 格式和 SREC 格式
Redboot _ floppy _ install \ include	包含编译时所需的头文件
Redboot _ floppy _ install \ lib	产生 RedBoot 映像文件时所需的库文件和链接器脚本文件
Redboot _ floppy _ mlt	包含配置工具所用的内存布局文件

6) 编译 RedBoot。在“Build”菜单中选择“Library”,开始对 RedBoot 进行编译。编译完成后,所产生的 RedBoot 映像文件存放在 Redboot _ floppy _ install \ bin 目录下,即 Redboot _ floppy. bin。使用该文件可以制作一个 RedBoot 引导软盘。

还有另一种方法用于产生一个新的 RedBoot 配置(替代上述 2)和 3)操作)。这种方法就是导入一个 eCos 最小配置文件(. ecm)。每一个 RedBoot 支持的硬件平台都有一个最小配置文件,最小配置文件位于该平台硬件抽象层 HAL 的 misc 子目录下(hal \ misc)。

最小配置文件包含了指定硬件平台的一些基本配置信息。开发人员可以将这些配置文件作为起点,根据目标平台对 RedBoot 映像的具体要求对其配置信息进行修改,以满足实际需要。最小配置文件使用的是 CDL 描述语言。下面是基于 PowerPC 的 Motorola MBX 开发板的一个最小配置文件(redboot _ rom. ecm)。在该. ecm 文件中,可以发现在 cdl _ configuration 命令下包含了一组 RedBoot 配置所需要的软件包,而 cdl _ option 命令对一些特殊选项的可选值进行了设置。

```
---------- redboot _ rom. ecm ----------
cdl _ savefile _ version 1;
```

```
cdl_savefile_command cdl_savefile_version {};
cdl_savefile_command cdl_savefile_command {};

cdl_configuration eCos {
description "" ;
hardware mbx ;
template redboot ;
package -hardware CYGPKG_HAL_POWERPC current ;
package -hardware CYGPKG_HAL_POWERPC_MPC8xx current ;
package -hardware CYGPKG_HAL_POWERPC_MBX current ;
} ;

cdl_option CYGDBG_HAL_DEBUG_GDB_INCLUDE_STUBS {
inferred_value 1
} ;

cdl_option CYGSEM_HAL_ROM_MONITOR {
inferred_value 1
} ;
```

大多数平台都包含了分别基于 ROM 和 RAM 的两个 RedBoot 最小配置文件 (Redboot_ROM.ecm 和 Redboot_RAM.ecm),它们都可以用作配置的起点。ROM 和 RAM 分别表示 RedBoot 映像文件将驻留在 ROM 内还是 RAM 内。RAM 配置一般用于对软硬件进行调试的阶段,由其产生的 RedBoot 将在 RAM 中运行。ROM 配置一般用在软硬件调试完成之后将要写入到 ROM 或 FlashROM 的场合,其 RedBoot 将在 ROM 中运行。基于 RAM 的 RedBoot 映像还可以用于对驻留在 FlashROM 内的 RedBoot 映像进行更新操作。对于 i386 PC 的目标平台,除了这两个配置文件外还有一个 FLOPPY 配置文件,FLOPPY 配置文件用于生成从软盘引导的 RedBoot 映像文件。

使用最小配置文件对 RedBoot 进行编译的方法是:在配置工具的"File"菜单选择"Import",将指定的.ecm 文件导入到 eCos 配置工具。其余步骤与前面介绍的 4) 和 5) 相同。

4.3.3 使用命令行配置工具 ecosconfig

eCos 提供了一个命令行配置工具 ecosconfig,包括其源码。如果要重新编译该工具,可以根据源码中的 host\README 文件指定的步骤进行。例如,在 Linux 环境下对 ecosconfig 的编译过程如下:

```
mkdir $TEMP/redboot-build
cd $TEMP/redboot-build
$ECOSDIR/host/configure --prefix=$TEMP/redboot-build --with-tcl=/usr
make
```

在得到可用的 ecosconfig 后,可以使用它对 RedBoot 进行配置。首先要设定 ECOS_REPOSITORY 环境变量,使其指向 eCos/RedBoot 源目录,然后用下列命令对指定目标系统

(TARGET)的 RedBoot 进行编译：

```
ecosconfig new TARGET redboot
ecosconfig tree
make
```

这里的 TARGET 为目标平台名，如“assabet”。上述命令得到的是最简单的 RedBoot，不包含目标平台所支持的网络、Flash 或 Compact Flash Ethernet 等特性。如果要支持这些特性，应该使用下列命令进行编译：

```
ecosconfig new TARGET redboot
ecosconfig add flash
ecosconfig add pcmcia net _ drivers cf _ eth _ drivers
ecosconfig tree
make
```

其中的第二、三行命令将平台所需的支持模块加入到编译过程。实际上，大多数平台的硬件抽象层 HAL 都包含了一个配置导出文件（.ecm），使用该配置文件可以对 RedBoot 的编译进行正确的配置，避免增加一些多余的软件包。上述命令最后得到的 RedBoot 可用于测试目的，但只能在 RAM 中运行。如果要将 RedBoot 安装到 ROM 或 FlashROM，则必须产生一个 RedBoot 的 ROM 映像。通过下列命令可以得到 RedBoot 的 ROM 版：

```
cat >RedBoot _ ROM.ecm <<EOF
cdl _ component CYG _ HAL _ STARTUP {
user _ value ROM
};
EOF
ecosconfig import RedBoot _ ROM.ecm
ecosconfig tree
make
```

上述命令中用到了配置导出文件 RedBoot _ ROM.ecm。一些平台的硬件抽象层 HAL 提供了用于 RAM 启动和 ROM 启动的不同配置文件（RedBoot _ RAM.ecm 和 RedBoot _ ROM.ecm），使用这些配置文件有助于 RedBoot 的编译。这些配置文件一般位于平台 HAL 的 misc 子目录下。

经过上述编译过程将得到三种不同格式的 RedBoot 映像文件。这些映像文件位于目录 install/bin 下。开发人员可以根据具体硬件平台的实际需要选择相应格式的 RedBoot 映像。三种格式的映像文件分别是：

- install/bin/redboot.elf　ELF 格式的 RedBoot 映像。
- install/bin/redboot.srec　Motorola SREC 格式。
- install/bin/redboot.bin　二进制映像，可直接写入 ROM/FlashROM。

4.4　RedBoot 的更新与运行

RedBoot 通常运行在 FlashROM 和 ROM 内。各种平台都具有各自将 RedBoot 映像安装

到这些非易失性内存的方法。一般来讲,可以用软件将其写入到 FlashROM,而对于 ROM 则要使用 ROM 编程工具。在使用 Flash 的情况下,可以直接使用 RedBoot 对其进行更新。

要完成 RedBoot 的更新,需要两个版本的 RedBoot :一个从 Flash 运行,而另一个从 RAM 运行。更新的主要过程是 :首先从 Flash 运行 RedBoot,在此环境下加载并运行基于 RAM 的 RedBoot,然后加载新版基于 Flash 的 RedBoot,最后将此 RedBoot 写入到 Flash。

下面介绍更新 RedBoot 的具体步骤:

1) 启动目标平台,运行 Flash 内即将被更新的 RedBoot。由于更新过程要用到 Flash 映像系统命令 (FIS),因此首先必须运行下面的命令对 FIS 进行初始化:

```
RedBoot> fis init
```

该命令对 Flash 映像系统 FIS 进行初始化,并使能新映像文件,使其可以使用 fis 命令对 Flash 进行编程操作。在 FIS 初始化完成后,可以使用如下命令查看 Flash 内存中所有的映像:

```
RedBoot> fis list
```

Name	flash addr	Mem addr	Length	Entry point
RedBoot	0xA0000000	0xA0000000	0x020000	0x80000000
RedBoot [backup]	0xA0020000	0x8C010000	0x010000	0x8C010000
RedBoot config	0xA0FC0000	0xA0FC0000	0x020000	v0x00000000
FIS directory	0xA0FE0000	0xA0FE0000	0x020000	0x00000000

上述命令的执行结果仅是一个例子,地址、长度、入口点在不同的平台中各有不同。从该命令运行的输出结果可以看出,名字为"RedBoot"的映像就是当前正在运行的 RedBoot 映像。名字为"RedBoot [backup]"的映像是备份映像,可以在 RAM 执行。

2) 加载并启动另一个从 RAM 运行的 RedBoot。如果 FIS 中已经存在一个 RedBoot 的 RAM 映像,例如上面 fis list 命令所列出的 RedBoot [backup],则该映像可以直接被加载到 RAM 并运行。命令如下:

```
RedBoot> fis load RedBoot [backup]
RedBoot> go
```

如果 FIS 中不存在这样的 RedBoot 映像,或者该映像不能运行,则必须从外部加载 RedBoot 的 RAM 映像。RedBoot 的 RAM 映像可以通过前面介绍的 RedBoot 编译方法得到。需要注意的是,在配置时必须选择启动类型 (CYG _ HAL _ STARTUP) 为 RAM。RAM 映像和 ROM 映像要使用不同的工作目录。如果有了 RedBoot 的 RAM 映像文件,则可以通过下面的命令将该映像文件通过串口 X/Y Modem 协议或者 TFTP 网络协议加载到 RAM 内,并执行该映像。

```
RedBoot> load redboot _ ram. bin
RedBoot> go
```

此时目标平台上运行的是最新被加载的 RedBoot RAM 映像。

3) 加载并更新新版 RedBoot Flash 映像。使用下面的命令加载 RedBoot 映像:

```
RedBoot> load redboot_rom.bin -b <ram_addr>
```

这里的 ram_addr 是映像文件加载到内存的位置。redboot_rom.bin 映像在被写入到 Flash 之前将临时存放在 RAM 内。当该映像加载到内存后，就可以准备进行 Flash 编程操作了。一些平台支持对 Flash 内存的锁定（lock）和解锁（unlock）操作，在进行 Flash 编程前必须对 Flash 进行解锁：

```
RedBoot> fis unlock -f <flash_addr> -l <flash_len>
```

上面命令的 flash_addr 和 flash_len 可以通过 fis list 命令得到。解锁后可以使用下面的命令进行 Flash 编程，将临时存放在 RAM 内的 RedBoot Flash 映像写入 Flash 中：

```
RedBoot> fis create RedBoot -f <flash_addr> -b <ram_addr> -l
         <flash_len> -s <data_len>
```

这里的 flash_addr 是该映像写入到 Flash 的地址，可以从 fis list 命令得到。ram_addr 是该映像临时存放在 RAM 中的地址。flash_len 是 RedBoot Flash 映像的长度，也可以从 fis list 命令得到。data_len 是将被写入到 Flash 的 redboot_rom.bin 文件的长度。

当新的 RedBoot Flash 映像被写入到 Flash 后，如果平台支持 Flash 锁定（lock）操作，则使用下面的命令对其进行锁定：

```
RedBoot> fis lock -f <flash_addr> -l <flash_len>
```

4）重启目标平台，此时将运行刚才被更新的 RedBoot 映像。

第5章　系统内核

eCos的核心部分是它功能强大、灵活且具有可配置性的嵌入式内核。内核主要由多线程机制、调度机制、同步机制、内存分配机制、例外处理机制、中断处理机制以及定时机制组成。它的模块化结构和可配置特性可以保证在对其某些模块进行改变和更换时不会对内核的其余部分造成影响。eCos内核的主要功能包括：

1）调度算法的选择。

2）内存分配算法的选择。

3）提供了一组丰富的同步原语。

4）支持对称多处理（SMP）。

5）中断处理。

6）例外处理。

7）Cache控制。

8）提供定时器、计数器和告警机制。

9）支持多线程。

10）支持多线程GDB调试。

本章主要介绍内核的调度机制、中断和例外处理机制、定时机制、内存分配机制以及它们相应的API，另外线程及同步机制将在下一章讨论。对应用程序的入口点也作介绍。

5.1　系统内核

内核是eCos系统的一个关键性的包。它为多线程应用程序的开发提供核心支持。它具有在系统开始启动和系统运行期间创建新线程的能力，可以对系统中的各种线程进行控制和操作。它可以选择一种调度算法来决定当前执行哪一个线程，并提供了大量的线程同步原语。系统内核还提供了例外和中断处理机制。大多数操作系统通常在其内核内包含了内存分配机制和设备驱动程序，而eCos却不同，为了支持系统的可配置性，它采用一个单独的包来支持对内存的分配，对每一个设备驱动程序也提供一个单独的包来支持。这种方式的采用使得可以根据具体应用程序的实际需求，通过eCos的配置技术来对各种包进行组合，为系统提供最大限度上的可配置能力。

eCos内核本身也是一个可选的包。在开发一个单线程的应用程序时，有的情况下可以不需要内核的支持，RedBoot就是一个例子。这种无内核的单线程应采用一种循环查询的方式，连续不停地对所有设备进行检查，当有I/O事件发生时再进行相应的处理。这种查询方式的每一次循环都可能需要进行少量的计算，从I/O事件的发生到查询循环体检测到该事件的发生之间的延时将会因此而增加。如果这样可以满足应用需求，那么采用循环查询的方式是可行的，可以简化应用程序的开发过程，避免使用复杂的多线程机制和线程间的同步机制。当应用较为复杂需要使用多线程机制时，就必须使用内核。事实上，eCos的许多高级软件包（如

TCP/IP)都使用了多线程机制。如果应用程序使用了这样的软件包,那么内核已经变成一种必需的包,而不是一个可选的包。

可以采用两种方式来使用内核所提供的功能:一种方式是应用程序和其他的软件包直接调用内核提供的 API 接口函数,内核提供了许多像 cyg _ thread _ create 和 cyg _ mutex _ lock 这样的 C 函数作为其 API;另一种方式是使用 eCos 兼容层提供的标准函数。eCos 有许多软件包提供对现有 API 的兼容,如 POSIX 和 μITRON。应用程序可以调用一些标准函数(如 pthread _ create),这些标准函数使用了 eCos 内核所提供的基本功能。在 eCos 应用程序中使用标准函数可以更容易实现在其他环境下开发的软件的可重用性,从而实现程序代码的共享。

虽然不同的兼容包对内核有相似的需求,但它们的具体实现会有所不同。例如,严格意义上的 μITRON 兼容要求禁止内核的时间片。这就需要对内核进行配置,这种配置的具体实现主要通过 eCos 的配置技术来完成。内核提供了许多配置选项来控制它们所需要的功能,各种各样的兼容包需要对这些选项进行特殊的设置。由于不同的兼容包在同一个配置中可能引起内核内部选项的冲突,因此,通常在同一个配置内不能同时使用两个不同的兼容包。另外,由于内核内部配置选项的多样性,内核自己的 API 函数只是一种不精确的实现。例如,API 函数 cyg _ mutex _ lock 尝试对一个互斥体进行锁定,它具有一些不同的配置选项,可以对这些选项进行配置来确定当互斥体已被锁定以及可能存在优先级倒置时应该采取的行为。

内核的这种可选特性给某些程序特别是设备驱动程序增加了复杂性。不管内核是否存在,设备驱动程序都应该能够工作。在包含内核的多线程环境中,系统的某些部分尤其是与中断处理相关的部分的实现方法不同于在无内核的单线程环境中的实现方法。为了在这两种环境下都能正确处理,硬件抽象层的公共 HAL 包提供了一些驱动程序 API 函数,如 cyg _ drv _ interrupt _ attach(),当系统中包含有内核时,这些 API 函数将映射到等同的内核函数,如 cyg _ interrupt _ attach。当系统没有内核时,公共 HAL 包将直接实现这些驱动程序 API 函数。公共 HAL 包的这种直接实现方式的假设条件是在单线程环境,比在有内核环境下的实现要简单。

5.2 内核调度机制

eCos 内核的核心是调度器。当系统包含多个线程时,必须使用调度器来决定当前执行哪一个线程。调度器对线程的运行进行控制,并为线程提供一种同步机制。此外,它还对中断影响线程执行的方式进行控制。单个调度器不可能适用于所有可能的系统配置,eCos 内核为此提供了两种可选的调度机制,当前版本的 eCos 具有两个调度器:

① 位图(bitmap)调度器。

② 多级队列(MLQ)调度器。

相对来说,位图调度器效率要高一些,但存在许多限制。大多数系统通常采用多级队列调度器。这两个调度器都使用一个简单的数字优先级来决定当前应该执行哪一个线程。优先级的数目可以通过配置选项 CYGNUM _ KERNEL _ SCHED _ PRIORITIES 在配置工具中指定。典型的系统通常设置为 32 个优先级,线程的优先级范围从 0 到 31,0 的优先级最高,31 的优先级最低。一般只有系统的空闲线程运行在最低优先级。线程的优先级具有绝对性,因此在所有高优先级的线程被阻塞时,内核只能执行一个优先级较低的线程。

图 5-1 为 eCos 配置工具对内核调度器进行配置的示意图。

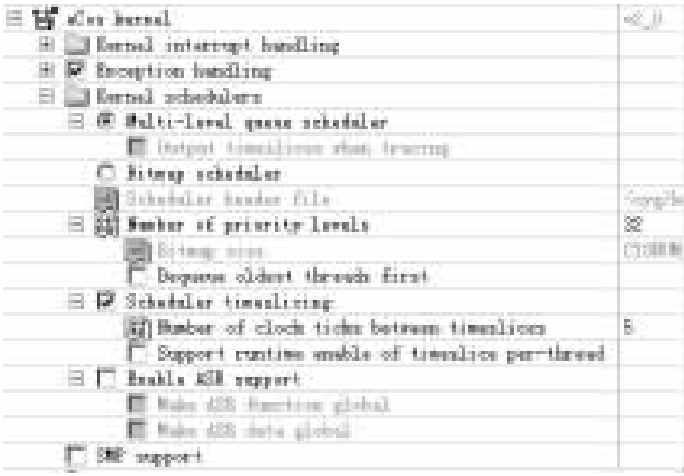


图 5-1　eCos 内核调度器的配置

5.2.1　位图调度器

位图调度器是一个相当简单的调度器。其主要思想是设置若干个不同的线程优先级（具体多少个线程优先级可以通过 eCos 配置工具进行设定），任何时刻每一个优先级不可能同时有两个或两个以上的线程，也就是说任何时刻不可能有相同优先级的线程。位图调度器所允许的线程个数有严格的限制。如果系统被配置成具有 32 个优先级，那么系统中最多只能有 32 个线程。在许多应用中 32 个线程已足够满足其需要。位图调度器使用一个位图（bitmap）来表示所有的线程，位图中的每一位都对应于一个可运行的线程，每一个线程的优先级是惟一的。从位图上可以很容易查出当前哪一个线程处于活跃状态、哪一个线程具有最高优先级。使用位图还可以查找出哪些线程正在等待一个互斥体或其他同步原语。对位图进行简单的操作就可以识别出最高优先级线程和处于等待状态的线程，使用一个数组索引操作就可以获取线程自身的数据结构。因此，位图调度器具有简单、快速、准确的特点。

5.2.2　多级队列调度器

多级队列调度器是一种具有优先级的 FIFO 调度策略。它具有多个线程优先级，允许多个线程具有相同的优先级。因此，只要有足够可用的内存，系统所允许的线程数目是没有限制的。像查找最高优先级线程这样的操作在多级队列调度机制中的开销要比位图调度器中的开销要大。多级队列调度器支持优先级继承。

多级队列调度器中相同优先级的线程形成一个 FIFO 队列。这些线程采用时间片轮转策略进行调度，时间片的大小可以通过配置工具进行设置。只有在同一优先级具有两个以上的线程而且没有更高优先级线程的情况下才启动时间片。如果时间片被禁止，那么线程不会被另一个同一优先级的线程抢先，它将继续运行直到它产生某个结果或者直到被阻塞（例如等待一个同步原语）。用于控制时间片的配置选项是 CYGSEM _ KERNEL _ SCHED _ TIMESLICE 和 CYGNUM _ KERNEL _ SCHED _ TIMESLICE _ TICKS。位图调度器不提供

时间片支持,在每一个优先级只允许运行一个线程,因此不可能存在同一优先级的线程抢先。

影响多级队列调度器的另一个重要配置选项是 CYGIMP _ KERNEL _ SCHED _ SORTED _ QUEUES,它决定当线程被阻塞时的行为。系统所采取的默认行为是后进先出队列。例如,如果系统有几个线程正在等待一个信号量 (semaphore),那么被唤醒的线程将是最后一个调用 cyg _ semaphore _ wait 等待该信号量的线程。这种方式使队列的入列操作和出列操作既简单又迅速。然而,如果进入队列的线程具有不同的优先级,被唤醒的可能不是最高优先级的线程。在实际运行过程中这种现象很少出现,队列中通常最多只有一个线程处于等待状态,或者在有多个等待线程的情况下它们可能具有相同优先级。如果需要有严格的优先级队列,可以使能 CYGIMP _ KERNEL _ SCHED _ SORTED _ QUEUES 配置选项。但它有一个缺点,当对线程进行队列操作时要做更多的工作,并且该操作需要锁定调度器,因此将会造成很大的系统延迟。

内核的某些功能目前只受到多级队列调度器的支持,位图调度器并不支持它们。这些功能包括对 SMP 系统的支持、使用互斥体优先级置顶或优先级继承的优先级倒置保护机制等。

5.2.3 调度器操作及 API 函数

eCos 目前只允许在其目标系统中使用一个单独的调度器,用户在进行配置时可根据实际需要在这两个调度器中进行选择。未来版本的 eCos 可以允许多个调度器协同工作。

为保证调度安全,必须提供一种机制,在对调度器数据结构进行同时访问时使用这种保护机制对这些数据结构进行保护。传统的保护方法是采用禁止中断的方法。但是,这种方法将会增加中断处理的延时,这在实时系统中是应该尽量避免的。eCos 通过对计数器 Scheduler::sched _ lock 的管理来实现这一机制。当该计数器的值不为 0 时,它阻止再次调度的发生。当前线程通过调用 cyg _ scheduler _ lock () 函数对其进行锁定,计数器加 1,阻止其他调度操作。cyg _ scheduler _ unlock () 函数使计数器减 1,如果计数器为 0,就允许继续进行调度操作。线程可以多次调用 cyg _ scheduler _ lock,每一次都将使计数器加 1。但为了重新使能线程之间的切换,必须调用 cyg _ scheduler _ unlock 相同的次数。这种行为与互斥体不同,如果一个线程试图对某个互斥体进行多次锁定操作将会导致死锁或错误的发生。

为保证在发生中断的情况下这种保护机制能正常工作,这就需要中断服务程序 ISR 将任何与调度有关的操作推迟到解锁 (计数器为 0) 后进行处理。为了实现这一目的,eCos 将中断服务程序 ISR 分割成两部分。它的第一部分在响应中断时进行即时处理,负责处理与实时相关的部分,它的第二部分称为滞后服务程序 DSR,这一部分的实时相关性小,DSR 排队等待直到调度器认为安全时才继续执行。

eCos 提供了一些对调度器状态进行控制的 API 函数,应用程序在使用它们的时候应该包含头文件<cyg/kernel/kapi.h>。它们分别是:

```
void cyg _ scheduler _ start (void) ;
```

该函数启动调度器开始工作。它使能系统中断,使系统可以开始进行 I/O 操作。在启动调度器之后,系统控制权将被交给优先级最高的线程。该函数被调用时将不会再返回。它只能被调用一次,表明系统的初始化操作已经结束。在典型的配置中,系统启动程序 (startup) 通

常会自动调用该函数，应用程序不应该再调用该函数。在某些不使用系统启动程序的应用中，必须调用该函数。

```
void cyg_scheduler_lock(void);
```

该函数用于锁定调度器，它阻止线程被另一个线程抢先。调度器的锁定机制使用了一个锁计数器，当调用该函数时，锁计数器将加 1。为重新使能多个任务之间的切换，必须调用 cyg_scheduler_unlock() 进行解锁，而且调用 cyg_scheduler_unlock 的次数应该等于 cyg_scheduler_lock 的次数。

```
void cyg_scheduler_safe_lock(void);
```

如果调度器还没有被锁定，则锁定调度器。如果调度器已经被锁定一次或多次，则该函数对锁计数器不起任何作用。如果锁计数器为 0（即调度器还没有被锁定），则锁定调度器并将锁计数器置为 1。

```
void cyg_scheduler_unlock(void);
```

对调度器解锁。每一次调用该函数都将使锁计数器减 1。为保证对调度器的真正解锁操作并使能线程之间的切换，对每一次 cyg_scheduler_lock 的调用都应该调用一次 cyg_scheduler_unlock。

```
cyg_ucount32 cyg_scheduler_read_lock(void);
```

该函数读取调度器的锁计数器，获取调度器的当前锁定次数。如果线程没有锁定调度器，则返回 0。如果锁计数器为 N，则应该调用 cyg_scheduler_unlock 函数 N 次，从而重新使能线程上下文切换操作。

一般来说，应用程序不应使用调度器锁，应该使用诸如互斥体和信号等这样的同步原语。虽然锁定调度器可以使当前线程不会被抢先，但也阻碍了具有更高优先级的线程的运行。它还将使系统中的 DSR 无法运行，从而导致设备驱动程序不能对 I/O 请求进行服务。

尽管如此，还是有一种情况合适于调度器锁定机制的使用。如果某些数据结构需要被应用线程和某个中断的 DSR 所共享，那么该线程可以使用调度器锁来防止 DSR 对这些数据结构的并发请求，从而安全地对数据进行操作。这种锁定只需要很短的一段时间。在某些异常情况下，使用调度器锁比使用互斥体将更为有效。

cyg_scheduler_start 只能在系统初始化过程中被调用，这是因为它标志着这一初始化阶段的结束。调度器的其他 API 函数可以在线程和 DSR 的环境下被调用。在 DSR 内对调度器进行锁定将不会有任何实际效果，中断机制在运行 DSR 之前自动进行了这种锁定操作。

5.3 内存分配

大多数的 eCos 系统调用在被使用时，要求将预先指定的由系统调用所产生的目标对象的内存地址传递给该系统调用（即对其进行静态内存分配）。在嵌入式系统中，这是一种常用的较好方法。对内存进行静态分配可以对资源进行很好的控制。

eCos 是一个单进程多线程系统，作为一个嵌入式可配置操作系统，它的内存管理相对简

单,不分段也不分页。惟一较复杂的地方是其可配置部分。eCos 采用一种基于内存池的动态内存分配机制。这是由 μITRON 兼容层实现的一种灵活有效的内存管理方式。eCos 的内存分配机制不在内核包内,而是使用一个单独的内存分配包,其目录位于 cyg \ packages \ services \ memalloc。

5.3.1 内存分配机制

eCos 提供两种可供选择的内存模板 Cyg _ Mempoolt 和 Cyg _ Mempolt2,每种内存模板又提供两种可选的内存池:一种是变长内存池(variable size memory pool),根据申请的大小进行分配;另一种是定长内存池(fixed size memory pool),以固定大小的块为单位进行分配。变长内存池使用链表来进行管理,定长内存池使用位图来进行管理。默认情况下,eCos 使用变长内存池方式管理内存,C 库函数 malloc () 使用了变长内存池来实现内存分配。如果在配置时选择了标准 C 库,编程时可以调用库函数 malloc () 进行内存分配。

图 5-2 为 eCos 图形配置工具中的内存分配机制的配置方法。内存分配器的选择通过选项 CYGBLD _ MEMALLOC _ MALLOC _ IMPLEMENTATION _ HEADER 来指定。在选择新的分配器的时候,必须将新分配器的"requires"条件设置为该内存分配器所使用的头文件所在位置,否则将会引起冲突。使能新的内存分配器的时候还应该对 CYGINT _ MEMALLOC _ MALLOC _ ALLOCATORS 进行声称。

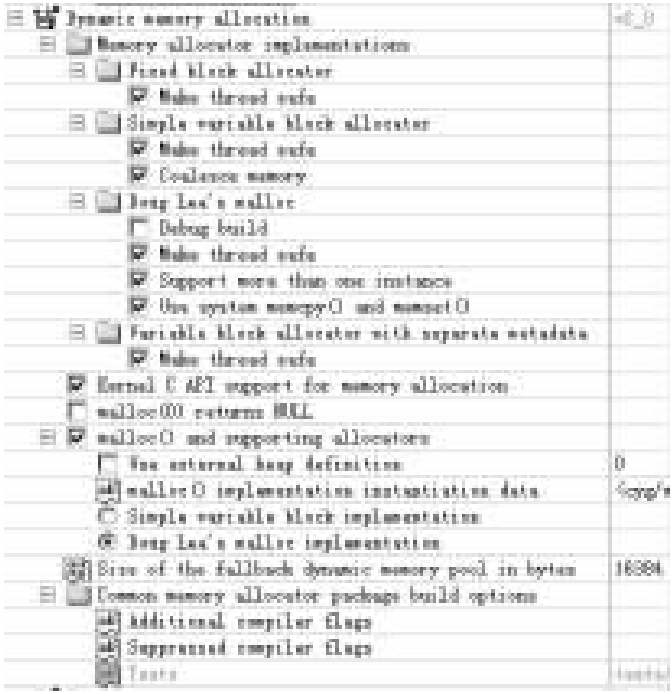


图 5-2 eCos 配置工具中的内存分配机制的配置

在多线程环境下,必须以一种安全的方式使用内存分配器。目前所有内存分配器所采用的一种有效方式是使用模板 Cyg _ Mempolt2。该模板提供了额外的一些函数,例如具有阻塞能力的 alloc () 函数,这些函数在返回前等待内存被释放,并且还具有一个用于超时处理的定时

变量。

eCos 允许对拥有所有可用内存的内存池进行自动定义，可以自动分配堆的大小。要实现这一功能，必须使用 eCos 配置工具中的内存布局工具 MLT 对用户定义段 (user-defined section) 进行定义。这些段的名字必须冠以“heap”的前缀，如“heap1”、“heap2”、“heapdram”等等。用户定义段可以是固定大小，也可以是一个未知的大小。如果是未知大小，它的大小可以通过下一个具有绝对地址的段的位置或内存区的末端来确定，通常使用后者。如果没有发现以“heap”开头的用户定义段，将使用一个 fallback 静态数组 (fallback static array)，其大小在配置时指定。

在具有多个不连续的内存区并且没有存储管理单元 MMU 将它们连接成一个连续的内存空间时，可以定义多个堆段。此时将自动使用一个特殊封装的分配器，该分配器定义于 memjoin. hxx 文件内，是 Cyg _ Mempool _ Joined 模板类的一个实例。它将每一个堆段的记录形成一个列表，根据该表可以确定使用的是哪一个内存池。使用 Cyg _ Mempool _ Joined 类将增加开销，如果开销过大，最好不要使用这种多个不连续的堆。

在一些特殊情况下，某些系统具有现场增加内存的支持能力。这种情形要求对这些内存进行自动分配。为此，在硬件抽象层 HAL 中 (hal _ intr. h) 有一个宏定义：

```
HAL _ MEM _ REAL _ REGION _ TOP ( cyg _ uint8 * regionend )
```

该宏采用正常情况下的内存末端地址作为参数。它与 MLT 中的内存区大小相对应，是内存没有扩充之前的内存区的最高端地址。它返回一个由 HAL 实时检测到的实际内存末端地址。通过该宏的使用，可以灵活地对多个内存区进行操作。

5.3.2 固定长度内存分配 API

eCos 的内存分配包提供了一些用于进行固定长度内存分配操作的 API 函数。这些函数的定义位于头文件＜cyg/memalloc/kapi. h＞内。下面是这些 API 函数的简单介绍。

```
void cyg _ mempool _ fix _ create
(
    void * base,              /* pointer to memory to use as heap */
    cyg _ int32 size,         /* size of memory to use as heap */
    cyg _ int32 blocksize,    /* size of blocks in fixed mempool */
    cyg _ handle _ t * handle, /* returned handle to pool */
    cyg _ mempool _ fix * fix  /* fix mempool structure */
)
```

该函数用于产生一个可以进行固定大小内存分配的定长内存池。参数 size 不一定是整个可用内存的大小。定长内存池在速度上要优于变长内存池。新产生的内存池可以通过 handle 对其进行访问。

```
void cyg _ mempool _ fix _ delete
(
    cyg _ handle _ t fixpool /* fixed sized memory pool to delete */
)
```

该函数删除一个定长内存池。使用时应该注意不要删除正在使用的内存池,否则将会引起系统错误。

```
void * cyg_mempool_fix_alloc
(
    cyg_handle_t fixpool /* fixed sized memory pool to allocate from */
)
```

该函数从内存池中分配一个固定大小的内存块。内存分配的最小边界为 4B。如果没有可用的内存,调用该函数的任务将被阻塞,直至有足够的满足需要的内存。其返回值为新分配的内存指针,如果没有分配内存则为 NULL。

```
void * cyg_mempool_fix_timed_alloc
(
    cyg_handle_t fixpool,        /* fixed memory pool to allocate from */
    cyg_tick_count_t abstime     /* absolute timeout value */
)
```

该函数从内存池中分配一个固定长度的内存块。内存分配的最小边界为 4B。如果没有可用的内存,调用该函数的任务将被阻塞,直到有足够的满足需要的内存或者到达 abstime 指定的时间(超时)。其返回值为新分配的内存指针,如果超时则为 NULL。

```
void * cyg_mempool_fix_try_alloc
(
    cyg_handle_t fixpool /* fixed memory pool to allocate from */
)
```

该函数从内存池中分配一个固定长度的内存块。内存分配的最小边界为 4B。如果没有可用的内存,则立即返回 NULL。其返回值为新分配的内存指针,如果不能分配内存则为 NULL。

```
void cyg_mempool_fix_free
(
    cyg_handle_t fixpool, /* pool memory was allocated from */
    void * p              /* memory to return to pool */
)
```

该函数释放从定长内存池中分配的内存。必须确认被释放的内存是在同一个内存池 fixpool 中分配的。如果把从某个内存池分配的内存释放到另一个内存池,则将出现无法预料的结果。

```
cyg_bool_t cyg_mempool_fix_waiting
(
    cyg_handle_t fixpool /* fixpool to check */
)
```

该函数检查是否有线程正在等待从定长内存池中分配内存而处于被阻塞状态。如果有线

程被阻塞，则返回 true，否则返回 false。

```
void cyg_mempool_fix_get_info
(
    cyg_handle_t fixpool,  /* pool to get info on */
    cyg_mempool_info *info /* receives info */
)
```

该函数返回定长内存池的信息。返回的内存池信息是一个结构体：

```
typedef struct {
        cyg_int32 totalmem;
        cyg_int32 freemem;
        void *base;
        cyg_int32 size;
        cyg_int32 blocksize;
        cyg_int32 maxfree;
} cyg_mempool_info;
```

5.3.3 可变长度内存分配 API

除了固定长度内存分配 API 外，eCos 还提供了可变长度内存分配的一些 API 函数，函数的定义位于头文件<cyg/memalloc/kapi.h>内。下面是这些 API 函数的简单介绍。

```
void cyg_mempool_var_create
(
    void *base,              /* pointer to memory to use as heap */
    cyg_int32 size,          /* size of memory to use as heap */
    cyg_handle_t *handle,    /* returned handle to pool */
    cyg_mempool_var *var     /* mempool structure */
)
```

该函数用于产生一个可以进行可变长度内存分配的变长内存池。参数 size 不一定是整个可用内存的大小。它的功能等同于标准 C 函数 free 和 malloc。新产生的内存池可以通过 handle 进行访问。

```
void cyg_mempool_var_delete
(
    cyg_handle_t varpool /* variable sized memory pool to delete */
)
```

该函数删除一个变长内存池。使用时应该注意不要删除正在使用的内存池，否则将引起系统错误。

```
void *cyg_mempool_var_alloc
(
    cyg_handle_t varpool, /* variable memory pool to allocate from */
```

```
    cyg_int32 size          /* size of memory block to allocate */
)
```

该函数从变长内存池中分配一个任意长度的内存块。内存分配的最小边界为 4B。如果没有可用的内存,调用该函数的任务将被阻塞,直至有足够的满足需要的内存。其返回值为新分配的内存指针,如果没有分配内存则为 NULL。

```
void *cyg_mempool_var_timed_alloc
(
    cyg_handle_t varpool,     /* variable memory pool to allocate from */
    cyg_int32 size,           /* size of memory block to allocate */
    cyg_tick_count_t abstime  /* absolute timeout value */
)
```

该函数从变长内存池中分配一个任意长度的内存块。内存分配的最小边界为 4B。如果没有可用的内存,调用该函数的任务将被阻塞,直至有足够的满足需要的内存或者到达 abstime 指定的时间(超时)。其返回值为新分配的内存指针,如果超时则为 NULL。

```
void *cyg_mempool_var_try_alloc
(
    cyg_handle_t varpool, /* variable memory pool to allocate from */
    cyg_int32 size        /* size of memory block to allocate */
)
```

该函数从变长内存池中分配一个任意长度的内存块。内存分配的最小边界为 4B。如果没有可用的内存,则立即返回 NULL。其返回值为新分配的内存指针,如果不能分配内存则为 NULL。

```
void cyg_mempool_var_free
(
    cyg_handle_t varpool, /* pool memory was allocated from */
    void *p               /* memory to return to pool */
)
```

该函数释放从变长内存池中分配的内存。必须确认被释放的内存是在同一个内存池 varpool 中分配的。如果把从某个内存池分配的内存释放到另一个内存池,则其结果无法预料。

```
cyg_bool_t cyg_mempool_var_waiting
(
    cyg_handle_t varpool /* varpool to check */
)
```

该函数检查是否有线程正在等待从变长内存池 varpool 中分配内存而处于被阻塞状态。如果有线程被阻塞,则返回 true,否则返回 false。

```
void cyg_mempool_var_get_info
```

```
(
    cyg_handle_t varpool,  /* pool to get info on */
    cyg_mempool_info *info /* receives info */
)
```

该函数返回一个变长内存池的信息。返回的内存池信息是一个结构体：

```
typedef struct {
        cyg_int32 totalmem;
        cyg_int32 freemem;
        void *base;
        cyg_int32 size;
        cyg_int32 blocksize;
        cyg_int32 maxfree;
} cyg_mempool_info;
```

5.4 中断处理

中断是由外部设备引起的异步事件。它可能随时发生，可能与当前正在运行的线程没有任何关联关系。在实时操作系统设计中，对中断的处理颇为复杂。中断向量的定义、中断如何交付软件处理、中断的屏蔽方法等等都与具体硬件结构有着密切的联系。eCos 采取的方法是采用一个通用的中断处理机制，它具有足够多的函数钩用于放置与硬件结构相关的程序代码。

5.4.1 线程与中断处理程序

在正常情况下，处理器上运行的是系统中的某一个线程，它可能是应用线程，也可能是系统线程或空闲线程。当发生中断时，系统将转到中断处理程序进行处理。在中断完成后，再由系统调度器来决定是返回被中断的线程还是转入其他线程。

线程和中断处理程序必须能够进行交互。如果一个线程正在等待 I/O 操作的完成，相应的中断处理程序必须具有通知该线程此次 I/O 操作已经完成的能力。有许多方法可以实现这种需求。一种简单的实现方法是使用变量，中断处理程序对该变量进行设置，而线程则对该变量进行连续或定时查询。连续查询会占用大量的 CPU 时间，而定时查询将会造成一定的延时。另外一种更好的方法是使用同步原语，中断处理程序可以使用条件变量、信号或其他的同步原语来通知正在等待中断操作完成的线程。这种方式在 I/O 事件发生之前不会占用 CPU 时间，而且在 I/O 事件发生的时候该线程可以立即重新运行 (当然是在没有更高优先级线程的条件下)。

同步原语有一些数据是共享的，在并发访问时应该注意数据的安全性。如果被中断的线程只进行一些计算操作，则中断处理程序可以十分安全地访问同步原语。但如果被中断的线程正在执行内核调用，则有可能破坏内核数据。避免这一问题的方法之一是在处于临界区的内核函数中禁止中断。许多系统可以简单而快速地实现这种方法，但同时也带来了一个问题：中断被频繁地禁止，而且被禁止的时间可能相当长。在某些应用中这种问题可以不考虑，但许多嵌入式应用要求中断能够尽快得到处理，这种情况下不适合使用禁止中断的方式。

为解决这一问题,内核将中断处理分成两个层次。与中断向量直接相连的是中断服务程序 ISR,它对中断进行尽可能的快速处理。ISR 只能使用少量的内核调用,不能使用唤醒线程的调用。当 ISR 检测到 I/O 操作已经完成因而要唤醒线程的时候,它引起中断的另一层处理程序的运行,这一层中断处理程序叫做滞后中断服务程序 DSR。DSR 可以进行更多的内核调用,它可以给条件变量一个信号,也可以发出一个信号量。

禁止中断可以阻止 ISR 的运行,但对中断的禁止要尽可能少,而且时间不能太长。线程禁止中断的主要原因是它要对一些 ISR 共享数据进行操作。例如,线程在一个空闲 buffer 链表上增加一个 buffer 时需要禁止中断,如果在对链表操作过程中有硬件中断发生,该中断将被挂起直至中断被重新使能。

内核还使用了调度锁对中断处理进行控制。使用各种内核函数 (如 cyg _ mutex _ lock 和 cyg _ semaphore _ post 等) 对调度锁进行声称,在对内核数据的操作完成之后再释放调度锁。如果中断要求运行 DSR 但调度器已被锁定时,DSR 将被挂起。当调度锁被释放之后,再运行被挂起的 DSR。可以给同步原语发送事件,唤醒具有更高优先级的线程。

综上所述,中断处理程序包括两部分。其中一部分为中断服务程序 ISR,当中断发生时立即执行这一部分程序,执行过程中中断是被禁止的。由于整个 ISR 都要禁止中断,因此 ISR 应尽可能简短,而且不要使用任何系统服务。当 ISR 执行完后,将执行中断程序的另一部分即滞后服务程序 DSR。在 DSR 的执行过程中,中断处于使能状态。

5.4.2 中断的处理

eCos 内核提供了一个对中断进行处理的接口,该接口具有许多 API 函数,它们用于安装中断处理程序并对中断进行控制。这些函数主要被设备驱动程序和那些与硬件直接交互的应用程序所使用。在大多数情况下,应该避免对这些内核函数的直接使用,而应该使用公共 HAL 所提供的设备驱动程序 API 函数。内核包是一个可选包,某些应用 (如 RedBoot) 不需要使用多线程机制和同步机制。在这种无内核的 eCos 配置中,如果程序直接调用内核 API 而不是调用设备驱动程序 API,那么这样的程序将不能正常工作。

不同体系结构对中断的处理过程不完全一致,内核所提供的中断处理对这些不同的处理细节进行抽象和提炼,从而简化了应用程序的开发过程。中断处理的具体过程都在硬件抽象层 HAL 中实现。eCos 为每一个中断都分配了一个中断向量,当系统调用涉及到中断处理程序时,将会用到这些向量。中断向量的具体分配依赖于具体硬件平台所使用的微处理器和中断控制器。

eCos 采用一个通用的中断处理机制,它具有足够多的函数钩子用于放置与硬件结构相关的程序代码。在系统初始化阶段,要完成中断句柄的产生、中断与中断向量的挂接以及中断的配置等操作。在安装中断处理程序时,首先要根据中断向量和中断优先级以及相应的中断服务程序 ISR 和滞后中断服务程序 DSR 使用内核 API 函数 cyg _ interrupt _ create 为中断产生一个中断句柄,然后再调用 cyg _ interrupt _ attach 将实际中断挂接到中断句柄上。一个中断向量可能有多个中断句柄,但只能使用其中的一个。在更换中断向量的中断句柄时,必须先调用 cyg _ interrupt _ detach 解除该中断句柄与中断的挂接关系,然后再调用 cyg _ interrupt _ attach 将中断挂接到新的中断句柄上。如果不再需要某个中断句柄,则可用 cyg _ interrupt _ delete 将其删除。在某些系统中需要对中断的触发方式进行配置,使用 cyg _ interrupt _

configure 函数可以选择中断的触发方式:电平触发或者边沿触发、高电平触发或者低电平触发、上升沿触发或者下降沿触发。

有两种方法可用于控制是否允许中断的发生。一种方法是使用 cyg _ interrupt _ disable 函数和 cyg _ interrupt _ enable 函数,全局性地禁止或使能所有中断。这些工作通常由 CPU 在内部对自身进行操作来实现。另一种方法是利用中断控制器的屏蔽寄存器对指定的具体中断进行屏蔽操作。有两个函数用于屏蔽指定的中断:cyg _ interrupt _ mask 和 cyg _ interrupt _ intunsafe。对中断的屏蔽不是一个原子操作,如果两个线程同时进行中断屏蔽操作可能会出现问题。cyg _ interrupt _ mask 函数对中断屏蔽进行操作时,要禁止所有的中断。在已经知道中断被禁止的情况下,可以使用 cyg _ interrupt _ intunsafe 函数进行屏蔽操作。

为支持 SMP 系统,内核提供了两个针对 SMP 系统的中断处理函数。cyg _ interrupt _ set _ cpu 函数用于把某个中断指定给某个 CPU 进行处理,只有该 CPU 才能检测到该中断的发生,而且相应的向量服务程序 VSR 以及中断服务程序 ISR 和 DSR 也只能在该 CPU 上运行。另一个函数 cyg _ interrupt _ get _ cpu 函数用于查找中断由哪一个处理器进行处理。

当中断发生时,首先将进入向量服务程序 VSR。VSR 是一小段程序,一般是一段汇编程序,eCos 提供了默认的 VSR。默认 VSR 的运行独立于处理该中断的 ISR,并为 ISR 建立 C 语言环境。在某些应用中,有可能需要替换 eCos 的默认 VSR 而直接对中断进行处理。cyg _ interrupt _ get _ vsr 函数用于保存指定中断向量的当前 VSR,在需要的时候可以恢复 VSR。cyg _ interrupt _ set _ vsr 函数用于安装一个 VSR。

5.4.3 内核中断处理 API 函数

前面已经提及了内核对中断进行处理的一些 API 函数,这些函数的定义位于头文件<cyg/kernel/kapi.h>内,使用时应该包含该文件。下面简单介绍这些 API 函数。

```
void cyg_interrupt_create
(
    cyg_vector_t vector,        /* interrupt vector */
    cyg_priority_t priority,    /* priority of interrupt */
    cyg_addrword_t data,        /* data pointer */
    cyg_ISR_t *isr,             /* interrupt service routine */
    cyg_DSR_t *dsr,             /* deferred service routine */
    cyg_handle_t *handle,       /* returned handle to interrupt */
    cyg_interrupt *intr         /* put interrupt here */
)
```

该函数用于创建一个新的中断句柄。其中参数 vector 和 priority 分别为中断向量和中断的优先级。data 是对中断进行处理时传递给 ISR 和 DSR 的数据的指针。当响应中断时,首先执行的是 VSR,然后再调用参数 isr 所指定的中断服务程序 ISR。ISR 是一个 C 函数,其函数格式如下:

```
cyg_uint32
isr_function(cyg_vector_t vector, cyg_addrword_t data)
{    cyg_bool_t dsr_required = 0;
```

```
    ...
    return dsr_required ?CYG_ISR_CALL_DSR :CYG_ISR_HANDLED;
}
```

ISR 函数的第一个参数是中断向量;第二个参数就是传送给 cyg_interrupt_create 的参数。如果 ISR 返回 CYG_ISR_HANDLED,则不再调用 DSR,如果返回 CYG_ISR_CALL_DSR,则还要调用 DSR。与该中断相对应的 DSR 由参数 dsr 指定,DSR 的函数原型如下:

```
void dsr_function(cyg_vector_t vector,
                  cyg_ucount32 count,
                  cyg_addrword_t data);
```

DSR 函数的第一个参数为中断向量;第二个参数指明该中断发生并且由 ISR 调用该 DSR 的次数,其值通常为 1,当系统负载很大时可能大于 1;第三个参数就是传递给 cyg_interrupt_create 的 data 参数。当 ISR 返回 CYG_ISR_CALL_DSR 时,将调用 DSR,此时如果调度器没有被锁定,则立即运行 DSR。如果被中断的线程处于内核调用期间且锁定了调度器,则 DSR 将被推迟到调度器被释放时才运行。

cyg_interrupt_create 的另一个参数是 handle,它是该函数所创建的新的中断句柄。另外还有一个参数 intr,它为内核提供一个保存该中断句柄和相应数据的内存区域。

```
void cyg_interrupt_delete
(
    cyg_handle_t interrupt /* interrupt to delete */
)
```

该函数从系统中删除一个中断句柄。

```
void cyg_interrupt_attach
(
    cyg_handle_t interrupt /* interrupt to attach */
)
```

该函数将中断挂接到中断句柄上。在调用此函数之前,不能对中断进行处理。

```
void cyg_interrupt_detach
(
    cyg_handle_t interrupt /* interrupt to detach */
)
```

该函数解除中断与中断句柄的挂接。

```
void cyg_interrupt_get_vsr
(
    cyg_vector_t vector, /* vector to get */
    cyg_VSR_t ** vsr /* pointer to store vsr pointer */
)
```

该函数通过第二个参数获取中断的 VSR。VSR 由 eCos 的硬件抽象层提供,通常不需要

修改。

```
void cyg_interrupt_set_vsr
(
    cyg_vector_t vector, /* vector to set */
    cyg_VSR_t *vsr     /* pointer to new vsr */
)
```

该函数设置中断的 VSR。它将中断与相应的 VSR 进行挂接。

```
void cyg_interrupt_disable
(
    void
)
```

该函数禁止系统中的所有中断。除非在非用不可的情况才使用,正常情况下应尽量避免使用该函数,该函数将会引起一定的中断延迟。该函数的每一次调用都要有相应的 cyg_interrupt_enable 调用对中断进行重新使能。

```
void cyg_interrupt_enable
(
    void
)
```

该函数用于使能中断。

```
void cyg_interrupt_mask
(
    cyg_vector_t vector /* vector to mask */
)
```

该函数屏蔽指定的一个中断。

```
void cyg_interrupt_mask_intunsafe
(
    cyg_vector_t vector /* vector to mask */
)
```

该函数屏蔽指定的某个中断,它不具有中断安全性。

```
void cyg_interrupt_unmask
(
    cyg_vector_t vector /* vector to unmask */
)
```

该函数解除对一个指定中断的屏蔽。

```
void cyg_interrupt_unmask_intunsafe
(
```

```
    cyg_vector_t vector /* vector to unmask */
)
```

该函数解除对一个指定中断的屏蔽。该调用不具有中断安全性。

```
void cyg_interrupt_acknowledge
(
    cyg_vector_t vector /* vector to acknowledge */
)
```

该函数对中断进行应答(清中断)。

```
void cyg_interrupt_configure
(
    cyg_vector_t vector, /* vector to configure */
    cyg_bool_t level,    /* level or edge triggered */
    cyg_bool_t up        /* rising/falling edge, high/low level */
)
```

该函数对指定中断进行配置。参数 level 选择电平触发还是边沿触发,参数 up 选择高电平触发还是低电平触发,或者选择上升沿触发还是下降沿触发。

```
void cyg_interrupt_set_cpu
(
    cyg_vector_t vector, /* vector to control */
    cyg_cpu_t cpu        /* CPU to set */
)
```

该函数将 SMP 系统中的某个中断分配给指定的 CPU 进行处理。

```
cyg_cpu_t cyg_interrupt_get_cpu
(
    cyg_vector_t vector /* vector to control */
)
```

该函数获取 SMP 系统中某个中断被指定给哪个 CPU 进行处理的信息。

5.5 例外处理

例外是线程运行时引起的同步事件。它包括硬件例外(如内存故障、非法指令等)和软件例外(如超时)。在实时系统中采用标准 C 的例外处理代价太高,而且有些例外有可能得不到正确处理。最简单和最灵活的例外处理方法就是采用调用例外处理函数的方法。这种例外处理函数需要一个运行环境,要求对运行数据进行访问。它还需要一些有关例外的一些数据,包括例外信息数据指针、例外向量、错误码及其他一些与例外有关的信息。从例外处理函数返回后,线程将继续运行。根据对配置选项的设置,例外处理可以是全局的,也可以是以线程为单位的,或者两者都是。如果是以线程为单位的,则每个线程都必须有一组例外处理程序。

5.5.1 例外处理程序

与中断处理一样,不同平台的例外处理有着很大的区别,因此例外处理程序的具体实现位于硬件抽象层 HAL 内。对于每一个例外都分配一个与其相对应的向量,即例外向量。当系统调用涉及到例外处理程序时,将使用该例外向量。例外的发生在很大程度上取决于系统硬件,尤其是处理器。处理器的相关说明文档有详细的例外说明,硬件抽象层中的头文件 hal _ intr. h 中有例外的相关定义。

例外处理是一个可选项,可以通过配置选项 CYGPKG _ KERNEL _ EXCEPTIONS 对其进行使能或禁止。在应用程序经过严格的测试,并且能确信不会出现任何例外的情况下,可以禁止该选项,这样可以减少程序和数据的代码量。如果例外处理被使能,系统将为各种例外提供默认的例外处理程序,但这些例外程序不做任何操作。如果应用程序要安装自己的例外处理程序和了解例外的详细情况,则必须使能 CYGSEM _ KERNEL _ EXCEPTIONS _ DECODE 选项。该选项使能内核对例外进行译码,并将例外交给与其相对应的例外处理程序进行处理。图 5-3 为 eCos 配置工具对例外和中断处理进行配置的示意图。

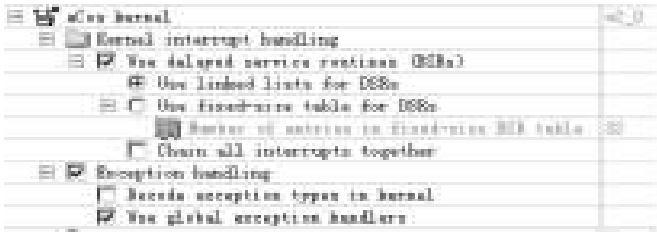


图 5-3 eCos 配置工具对例外和中断处理的配置

如果要使用一个新的例外处理程序,可以使用 API 函数 cyg _ exception _ set _ handler 函数进行安装,它为例外程序产生一个例外句柄。使用该函数时需要一个例外号、新的例外处理程序的函数指针和传递给该处理程序的一个参数。该函数在返回时将带回原来的例外句柄的信息,允许对它进行重新安装。如果不需要原来的例外处理程序的信息,则它返回时带回的可以是一个空指针。

例外处理程序所采用的格式如下:

```
void
my _ exception _ handler (cyg _ addrword _ t data,
                          cyg _ code _ t exception,
                          cyg _ addrword _ t info)
{
    ...
}
```

例外处理程序的第一个参数 data 是已传递给 cyg _ exception _ set _ handler 函数的第三个参数;第二个参数是例外号;第三个参数与硬件及例外有关。

如果需要的话,可以使用 cyg _ exception _ clear _ handler 函数来恢复默认的例外处理程序。

在默认情况下,系统提供一组全局例外处理程序。由于例外是同步发生的,因此有些时候以线程为单位对例外进行处理较为有效,这种情况下每一个线程都有一组不同的例外处理程序。在配置工具中通过禁止 CYGSEM _ KERNEL _ EXCEPTIONS _ GLOBAL 选项可以实现这种基于以线程为单位的例外处理。如果使用这种以线程为单位的例外处理,那么 cyg _ exception _ set _ handler 函数和 cyg _ exception _ clear _ handler 函数将只作用于当前线程。

5.5.2 例外处理内核 API 函数

内核为例外处理提供了几个 API 函数,头文件<cyg/kernel/kapi.h>内有这些函数的定义。下面是这些函数的简单说明。

```
void cyg _ exception _ set _ handler
(
    cyg _ code _ t exception _ number,              /* exception number */
    cyg _ exception _ handler _ t *new _ handler,  /* pointer to new handler */
    cyg _ addrword _ t new _ data,                 /* new handler data gument */
    cyg _ exception _ handler _ t **old _ handler, /* receives old handler */
    cyg _ addrword _ t *old _ data                 /* receives old data */
)
```

该函数创建一个新的例外处理句柄,并取回原来的例外处理句柄。其中 new _ handler 为新的例外处理程序的函数指针,new _ data 是例外发生时传递给例外处理程序的一个参数。原来的例外处理句柄的信息将由参数 old _ handler 和 old _ data 带回。

```
void cyg _ exception _ clear _ handler
(
    cyg _ code _ t exception _ number /* exception to clear */
)
```

该函数从系统中清除一个例外句柄。

```
void cyg _ exception _ call _ handler
(
    cyg _ handle _ t thread,            /* thread ID */
    cyg _ code _ t exception _ number,  /* exception number */
    cyg _ addrword _ t error _ code     /* error code */
)
```

该函数调用一个例外处理程序,参数 error _ code 是传递给例外处理程序的第三个参数 info。

5.6 SMP 支持

eCos 支持对称多处理器(SMP)系统,但只对某些体系结构和平台才提供支持。这种支持具有一定的限制性,具体细节可参阅第 11 章硬件抽象层与 eCos 移植。这一节主要介绍 eCos

内核为支持 SMP 系统而采取的一些措施。

5.6.1 SMP 系统的启动

在 SMP 系统中,系统的启动与单处理器系统有所不同,但对应用程序是透明的。启动的主要过程只由一个 CPU 完成,这个 CPU 称为主 CPU。而其他所有的 CPU 都是从 CPU,在系统复位后可能是处于挂起状态,也可能在硬件抽象层将它们启动后进入一种循环状态。主 CPU 负责对数据结构进行初始化,并调用硬件抽象层中的初始化程序,最后主 CPU 将调用 cyg _ start 进入应用程序。

只有在应用程序调用 cyg _ scheduler _ start 的时候,从 CPU 才进行初始化操作。该函数扫描所有可用的从 CPU,并调用 HAL _ SMP _ CPU _ START 启动每一个 CPU。最后它调用一个内部函数 Cyg _ Scheduler ::start _ cpu,从而使从 CPU 进入调度器。

每个从 CPU 都在 HAL 中被启动,在内核函数 cyg _ kernel _ cpu _ startup 被调用之前它们都已经完成了各自的初始化。从 CPU 对调度锁进行声称,并调用 Cyg _ Scheduler ::start _ up。

Cyg _ Scheduler ::start _ cpu 是主 CPU 和从 CPU 所共有的,它的第一步工作是为该 CPU 的机间中断安装一个中断对象,该函数此后的工作与单 CPU 的情形完全一致,即选择并运行初始线程。

在完成上述工作后,所有 CPU 都处于平等地位。但在进行中断处理时可能有所区别。

5.6.2 SMP 系统的调度

为保证系统功能的正确性,操作系统内核必须对关键数据的并发访问进行保护,例如对执行队列的保护。在单处理器系统中,所要考虑的并发行为只有异步中断,内核可以简单地禁止中断就可以对数据进行保护。但是在 SMP 系统中这种方法并不完全适用,它不能阻塞其他 CPU 的访问。

eCos 内核使用调度锁来保护关键数据。在单处理器系统中,调度锁是一个简单的计数器,在获取调度锁时它自动加 1,释放调度锁时自动减 1。当锁计数器减为 0 时,调度器可以选择运行另一个线程。由于在调度锁被声称的时候还可以继续为中断提供服务,因此不允许 ISR 为内核数据提供服务和访问内核函数,这些操作被放置到滞后中断服务程序 DSR 中进行。

内核锁机制对数据的保护并不依赖于中断操作,因此,它的使用可以比其他方法更易于使 eCos 支持 SMP。使 eCos 具有 SMP 安全性的一个主要变化是将调度锁转化为一个可嵌套的 spinlock,这可以通过在原来的锁计数器中增加一个 spinlock 和 CPU 的标识号 id 就可以实现。

获取调度锁的算法非常简单。如果调度锁的 CPU id 与当前 CPU 相符,则只需对计数器加 1 就可以进行工作。如果不相匹配,则当前 CPU 必须等待 spinlock,在获得 spinlock 之后它可以对锁计数器加 1,并写入它自己的 CPU id。在对锁进行释放时,计数器减 1。如果其值为 0,那么 CPU id 值必须置为 NULL,并清 spinlock。

为保护这些操作序列不受中断的影响,在进行这种操作时必须禁止中断。由于这些程序代码量很少,因此不会对中断造成很大的延时。

目前只有多级队列调度器 MLQ 具有支持 SMP 配置的能力。为使调度器支持 SMP 系统,需要对它进行一定的改进。一个主要变化是使调度器能够处理同时执行的多个线程。正

在执行的线程具有一个标识符表明它运行于哪一个 CPU 上。在对线程进行调度时,调度器跳过正在执行的线程,查找出一个处于悬挂状态的线程。

另一个需要改进的地方是用于决定什么时候调用调度器来选择运行新线程的程序段。调度器尽量保持每个 CPU 上运行一个最高优先级的线程(有多少个 CPU 就运行多少个当前优先级最高的线程)。由于一个 CPU 上的事件或中断可能需要对另一个 CPU 进行重新调度,因此必须有一个决策机制。为此,eCos 目前提供了一个非常简单的算法。假设有一个刚被唤醒的新线程(或者刚改变其优先级),调度器将从当前 CPU 开始对所有 CPU 进行审查,查找一个当前正在运行的优先级比新线程要低的线程。当找到这样一个线程时,给执行这一线程的 CPU 发送一个重新调度的中断。此后调度器将继续进行查找,但此时作为候选线程的是刚才被重新调度的 CPU 的当前线程。这种调度方法可以使新线程尽可能快地得到执行。新线程最有可能运行在当前 CPU 上,其余的 CPU 通过对重新调度中断的处理可以从剩余的线程中依次挑选出优先级最高的线程运行。

调度器还有一个需要进行改进的地方是对时间片的处理。虽然所有 CPU 都必须处理时间片,但只能有一个 CPU 接收定时中断。接收定时中断的 CPU 必须为所有的 CPU 的时间片计数器进行操作,而不仅仅只为自己。当某个 CPU 的时间片计数器到达 0 时,它将给该 CPU 发送一个时间片中断。该目标 CPU 收到中断后进入调度器查找并执行另一个具有相同优先级的线程。

在 SMP 系统中,可以使用单处理器系统中的已有的所有同步机制。此外,SMP 系统还增加了一个 SMP 的同步机制 spinlock,这将在第 6 章线程与同步中加以介绍。

5.6.3 SMP 系统的中断处理

SMP 系统中一个值得重视的地方是设备驱动程序和中断处理。在 SMP 系统中,设备驱动程序的 ISR、DSR 和线程组件极有可能运行在不同的 CPU 上。因此,具有 SMP 能力的设备驱动程序对与中断相关的函数的正确使用显得尤为重要。设备驱动程序通常使用驱动程序 API 函数,而不是直接调用内核函数,但在 SMP 系统中不使用内核包的可能性极小。

内核提供了两条 API 函数用于支持 SMP 系统中的中断路由,可以将指定的中断路由到指定的 CPU 进行处理。这两个 API 函数分别是 cyg _ interrupt _ set _ cpu 和 cyg _ interrupt _ get _ cpu,本章第 4 节对其进行了介绍。eCos 目前只支持对中断的静态路由。某个中断一旦被路由到指定的 CPU 进行处理,该中断的屏蔽和配置操作也将由该 CPU 进行处理。

SMP 系统中对中断的处理在第 8 章设备驱动程序与 PCI 库中有详细介绍。

5.7 计数器与时钟

硬件必须提供一个周期性的时钟或定时器,用于支持系统中与时间相关的功能部件。目前许多 CPU 都有一个内置定时器,它提供周期性的中断,可以对时间相关的功能部件提供支持。如果没有内置定时器,则必须使用外部定时器或时钟。

eCos 提供的定时机制包括计数器(Counter)、时钟(Clock)、告警器(Alarm)和定时器(Timer)。计数器对某些特殊事件进行单调递增计数。时钟是一个对具有一定周期性的时间滴答进行计数的计数器(对时间进行计数)。告警器是在计数器的基础上增加一个产生提示功

能的机制,或者基于计数器的值产生具有周期性的事件。定时器是一个简单的附加在时钟上面的告警器。

5.7.1 计数器

内核计数器用于对特殊事件发生的次数进行跟踪,这些特殊事件通常是某种类型的外部信号。计数器的最普通的一个实现就是时钟。应用程序可以在计数器上附加一个告警器,当某类事件发生次数达到一定的数目时调用某个相应的函数(告警函数)。

通过调用 cyg_counter_create 函数可以产生一个新的计数器。该函数的第一个参数用于返回新计数器的句柄,随后对该计数器的操作将使用这个句柄进行;它的第二个参数允许应用程序为该计数器提供所需的内存,从而减少了在内核中进行动态内存分配的需求。如果不再需要某个计数器并且没有任何告警器与其相连,则可以使用 cyg_counter_delete 函数删除该计数器并释放其占用的资源,使 cyg_counter 数据结构可以被重新使用。

产生一个新计数器时不会自动将事件源连接到该计数器上。无论相对应的事件何时发生,都需要调用 cyg_counter_tick 函数使计数器计数,并有可能引起告警器的触发。计数器的当前计数值可以使用 cyg_counter_current_value 函数读取,也可以使用 cyg_counter_set_value 函数进行修改。后一函数通常只在初始化期间进行,例如,将时钟设置为墙上时间时需要调用该函数。在需要的时候也可以用它对计数器进行复位。这一计数器设置函数不会引起告警器的触发。新产生的计数器的初始值为 0。

内核提供了计数器的两种不同实现方法,默认的计数器实现方法是 CYGIMP_KERNEL_COUNTERS_SINGLE_LIST,它将所有连接到该计数器上的告警器都存放到一个单独的链表中。这种方法简单而有效,但当事件发生的时候内核程序必须在该链表中来回移动。因此,如果与该计数器相连的告警器太多,势必影响系统的分配延时。另一种实现方法是 CYGIMP_KERNEL_COUNTERS_MULTI_LIST,它具有多个存放告警器的链表,每个告警器被存放到其中的一个链表,这样在每一个事件发生时最多只需查找一个链表。这种方法需要更多的程序代码和数据,但能提高实时响应速度。链表的数目可以在配置时指定,它的默认值为 8。另一个与计数器相关的配置选项是 CYGIMP_KERNEL_COUNTERS_SORT_LIST,默认情况下它是被禁止的。这一配置选项对计数器链表进行分类,减少了对计数器的触发事件进行处理时的工作量。使用该选项可以选择两种不同的工作量:给计数器增加一个新的告警器所需完成的工作;事件发生时进行处理所需完成的工作。在使用时可根据具体情况进行选择。eCos 图形配置工具中对计数器和时钟的配置如图 5-4 所示。

下面是内核提供的计数器 API 函数,头文件<cyg/kernel/kapi.h>内有这些函数的定义,使用时必须包含该头文件。

```
void cyg_counter_create
(
    cyg_handle_t * handle, /* returned counter handle */
    cyg_counter * counter  /* counter object */
)
```

该函数创建一个新的计数器,新计数器的句柄通过 handle 返回。

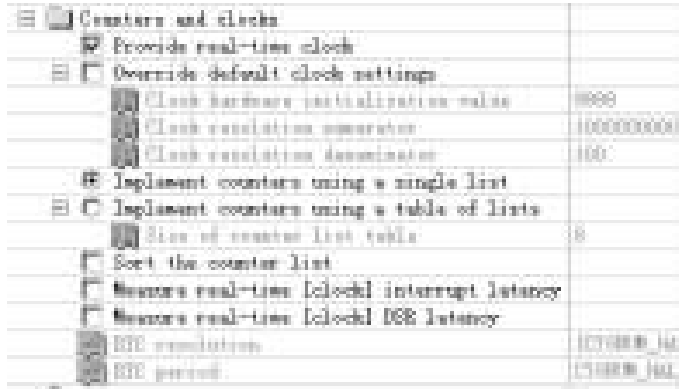


图 5-4　eCos 配置工具对计数器和时钟的配置

```
void cyg _ counter _ delete
(
    cyg _ handle _ t counter /* counter to delete */
)
```

该函数删除一个计数器。

```
cyg _ tick _ count _ t cyg _ counter _ current _ value
(
    cyg _ handle _ t counter /* counter to get the value of */
)
```

该函数读取指定计数器的当前计数值。

```
void cyg _ counter _ set _ value
(
    cyg _ handle _ t counter,          /* counter to set */
    cyg _ tick _ count _ t new _ value  /* new value of counter */
)
```

该函数设置计数器的值。

```
void cyg _ counter _ tick
(
    cyg _ handle _ t counter /* counter to advance */
)
```

该函数使计数器加 1。

```
void cyg _ counter _ multi _ tick
(
    cyg _ handle _ t counter,  /* counter to advance */
    cyg _ tick _ count _ t ticks /* number of ticks to advance */
)
```

该函数使计数器的值增加参数 ticks 所指定的数目。

5.7.2 时钟

eCos 内核所提供的时钟是计数器的一种特殊形式，它们与一种特殊类型的硬件相连，这些硬件可以产生非常精确的时间间隔。在默认配置下，内核提供一个单独的时钟实例，即实时时钟 RTC，它用于时间片以及一些与超时相关的操作（如 cyg _ semaphore _ timed _ wait）。如果不需要这种功能，可以使用配置选项 CYGVAR _ KERNEL _ COUNTERS _ CLOCK 将其从系统中删除（见图 5-4）。实时时钟 RTC 可以被 cyg _ real _ time _ clock 函数访问，允许应用程序对其附加一个告警器，它的当前计数值可以使用 cyg _ current _ time 函数读取。

应用程序根据需要可以使用 cyg _ clock _ create 函数和 cyg _ clock _ delete 函数创建和废除所附加的时钟。cyg _ clock _ create 的第一个参数指定了时钟的分辨率，第二个参数用于返回该时钟对象的句柄，第三个参数为内核提供该时钟对象所需要的内存。应用程序负责对硬件定时器进行初始化，使其以适当的频率产生中断，并为此中断安装一个中断处理程序，在 DSR 内部调用 cyg _ counter _ tick 函数，形成所创建时钟的滴答源。与每一个时钟相关联的是一个内核计数器，该计数器的句柄可以使用 cyg _ clock _ to _ counter 函数获取。

内核级的所有与时钟相关的操作包括延时、超时和告警等都是以时钟滴答为单位进行工作的，而不是以秒或微秒为单位。如果应用程序或其他软件需要使用秒或微秒这样的时间单位，则需要将这些时间单位转换为时钟滴答。这是因为只有时钟滴答才能精确地反映出硬件的支持条件。如果使用传统意义上的时间单位如纳秒（ns），则硬件有可能不能提供支持。另一个原因是在时钟滴答和传统意义上的时间单位之间的转换需要浪费很多的代码和数据，采用时间滴答为单位可以节省代码量和 CPU 时间。进行这种时钟单位的转换需要一个分辨率，分辨率是一个具有两个域的数据结构，一个域为被除数，另一个域为除数，用于指定在两个时钟滴答之间的纳秒数。例如，一个以 100Hz 运行的时钟在两个时钟滴答之间的时间是 10ms（10000000ns），那么其分辨率（被除数和除数之间的比率）是 10000000∶1，被除数和除数的典型值为 1000000000 和 100。如果时钟运行在不同的频率，如 60Hz，那么它们的值相应为 1000000000 和 60。如果给定一个以纳秒为单位的延时，则可以让其与分辨率中的除数相乘，再除以被除数，就可以将其转换为时钟滴答。

例如，一个 50ms 的延时（50000000ns），时钟频率为 100Hz，那么可以通过下面的计算转换为时钟滴答：

50000000 × 100／1000000000 次 = 5 次

平台实时时钟 RTC 的默认频率是 100Hz，通常可以根据硬件的实际情况来使用配置选项修改频率值（见图 5-4）。时钟的分辨率可以通过 cyg _ clock _ get _ resolution 函数获取。对于应用程序创建的时钟，还有一个 cyg _ clock _ set _ resolution 函数可用于设置分辨率，但它不会对定时器硬件产生影响。

内核提供了一些时钟 API 函数，这些函数定义于头文件＜cyg／kernel／kapi. h＞内，使用时应包含该头文件。下面是这些 API 函数的简单介绍。

```
void cyg _ clock _ create
(
```

```
    cyg_resolution_t resolution,  /* resolution */
    cyg_handle_t *handle,         /* created handle */
    cyg_clock *clock              /* clock object */
)
```

该函数使用给定的分辨率创建一个新的时钟，新时钟的句柄由 handle 带回。

```
void cyg_clock_delete
(
    cyg_handle_t clock /* clock to delete */
)
```

该函数删除一个时钟。调用此函数时应确认系统中已经没有使用该时钟的组件。

```
void cyg_clock_to_counter
(
    cyg_handle_t clock,       /* clock to convert */
    cyg_handle_t *counter     /* address of counter object */
)
```

该函数将时钟转换为计数器。返回的计数器对象由 counter 带回。

```
void cyg_clock_set_resolution
(
    cyg_handle_t clock,           /* clock */
    cyg_resolution_t resolution   /* new resolution to set clock */
)
```

该函数对时钟的分辨率进行设置。分辨率是一个数据结构，其定义如下：

```
typedef struct {
    cyg_uint32 dividend;
    cyg_uint32 divisor;
} cyg_resolution_t

cyg_resolution_t cyg_clock_get_resolution
(
    cyg_handle_t clock /* clock to get resolution of */
)
```

该函数获取时钟的分辨率。

```
cyg_handle_t cyg_real_time_clock
(
    void
)
```

该函数获取系统的实时时钟（RTC）。实时时钟用于系统的延时、阻塞等待等操作。

```
cyg_tick_count_t cyg_current_time
```

```
(
    void
)
```

该函数获取当前的系统时间,单位为时钟滴答。系统时间用 64 位的数字表示。

5.7.3 告警器

内核告警器与计数器一起使用,使得当某种事件发生了一定的次数时采取相应的行动。如果与计数器一起使用的是时钟,那么当时间滴答的个数达到适当的值也就是在一定的时间周期后时将发生告警行为。

设置一个告警器需要进行两步操作。首先必须调用 cyg _ alarm _ create 函数创建一个告警器。该函数有五个参数,第一个参数表明该告警器与哪一个计数器相连。如果告警器与系统的实时时钟相连,那么 cyg _ real _ time _ clock 函数和 cyg _ clock _ to _ counter 函数可以用来获取相应的句柄。随后的两个参数是一个函数指针和数据,用于指定当告警被触发时所采取的行为。这个告警函数的格式如下:

```
void
alarm _ handler (cyg _ handle _ t alarm, cyg _ addrword _ t data)
{
    ...
}
```

它的 data 参数就是传递给 cyg _ alarm _ create 的第三个参数。cyg _ alarm _ create 的第四个参数用于返回新产生的告警器对象,最后一个参数提供该告警器对象所需要的内存,这样可以避免在内核中进行动态内存分配。

在创建一个新的告警器之后,必须调用 cyg _ alarm _ initialize 函数对它进行激活。它的第一个参数指定告警器,第二个参数指定该告警器被触发之前事件 (如时钟滴答) 发生的次数。如果第三个参数为 0,那么该告警器只被触发一次,如果是一个非 0 值,那么它将指定告警器将被重复触发,触发的间隔为前面所指定的事件发生次数。

使用 cyg _ alarm _ disable 函数和 cyg _ alarm _ enable 函数可以临时禁止和使能告警器。另外还有一个 cyg _ alarm _ initialize 函数可用于修改告警器的行为。如果不再需要某个告警器,则可以使用 cyg _ alarm _ delete 函数释放其资源。

这些函数都是内核提供的告警器 API 函数,在头文件 < cyg/kernel/kapi. h > 内有定义。下面是它们的详细说明。

```
void cyg _ alarm _ create
(
    cyg _ handle _ t counter,   /* counter to attach to alarm */
    cyg _ alarm _ t *alarmfn,   /* alarm call back function */
    cyg _ addrword _ t data,    /* data to be passed to callback */
    cyg _ handle _ t *handle,   /* returned handle to alarm object */
    cyg _ alarm *alarm          /* alarm object */
)
```

该函数创建一个新的告警器。其参数在上面已有介绍。告警器的触发周期由 cyg _ alarm _ initialize 函数设置。当告警器被触发时,将调用“ * alarmfn”函数,“data”为其参数。告警器可以被设置为可重复触发方式和单次触发方式。新产生的告警器句柄由 handle 带回。

```
void cyg _ alarm _ delete
(
    cyg _ handle _ t alarm /* alarm to delete */
)
```

该函数从系统中删除一个告警器,并释放该告警器句柄。一旦删除,就不能再使用该告警器。

```
void cyg _ alarm _ initialize
(
    cyg _ handle _ t alarm,          /* handle of alarm to initialize */
    cyg _ tick _ count _ t trigger, /* absolute trigger time */
    cyg _ tick _ count _ t interval  /* retrigger interval */
)
```

该函数初始化并启动一个告警器,告警器的触发时间由 trigger 指定,它是一个绝对时间。可以对计数器调用 cyg _ counter _ current _ value 函数来获取一个时钟的当前触发时间。如果告警器以一个有规律的间隔周期重复触发,则 interval 应该为一个非 0 值。当告警器被触发时,将调用相应的告警器函数。

```
void cyg _ alarm _ get _ times
(
    cyg _ handle _ t alarm,           /* alarm to get the times of */
    cyg _ tick _ count _ t * trigger, /* next trigger time */
    cyg _ tick _ count _ t * interval  /* current retrigger interval */
)
```

该函数返回告警器的下一次触发的绝对时间和它的触发间隔。如果不需要返回其中的某个返回数据,可以使用 NULL 来代替这个参数 (函数的第二个参数或第三个参数)。

```
void cyg _ alarm _ enable
(
    cyg _ handle _ t alarm /* alarm to re-enable */
)
```

该函数重新使能先前被禁止的告警器,通常用于具有周期性的告警器。被重新使能的周期性告警器的触发间隔与它被禁止之前是相同的。例如,假设一个周期性的告警器每隔 10s 被触发一次,触发时刻为 T0、T10、T20、T30,如果在 T31 处被禁止,那么被禁止 15s 后重新使能时 (T46) 被触发的时刻将是 T50、T60、T70、等等。如果想复位它的间隔周期,可以使用 cyg _ alarm _ initialize 函数。

```
void cyg _ alarm _ disable
```

```
(
    cyg_handle_t alarm /* alarm to disable */
)
```

该函数禁止一个警告器。通常用于周期性的警告器。

5.8 应用程序入口

eCos 启动以后将经历许多不同的阶段,包括对硬件的设置和调用 C++ 静态构造函数。在这些过程中,中断被禁止,调度器被锁定。在一个包含有内核的配置中,最后的操作将是调用 cyg_scheduler_start 函数。从此处开始中断被使能,调度器被解锁,控制权将交给具有最高优先级的线程。如果包含有 C 库软件包,那么 C 库的启动软件将创建一个调用应用程序 main() 入口点的线程。

某些应用程序还可以在调度器启动之前运行,这些程序运行于初始化的环境之中。如果应用程序使用了 C++,那么将为静态对象调用构造函数。另外,应用程序还可以定义一个 cyg_user_start 函数,该函数在 C++ 静态构造函数之后被调用。这种方式允许应用程序全部使用 C 语言编程。作为应用程序入口点的 cyg_user_start 函数,其格式如下:

```
void
cyg_user_start(void)
{
    /* Perform application-specific initialization here */
}
```

应用程序不需要提供 cyg_user_start 函数,系统已经提供了该函数的一个默认实现,但它没有做任何操作。

在静态构造函数或 cyg_user_start 函数内所完成的工作主要包括创建线程和同步原语、设置告警器,并注册应用程序专用的中断处理程序。事实上,大部分的应用程序都在此完成这些创建操作,并且使用了静态分配数据,避免了动态内存分配的需要。

5.8.1 调用环境

eCos 定义了许多上下文环境,在每一种环境下只允许进行某些调用。例如,在中断服务程序 ISR 的上下文环境中,对线程和同步原语的大部分操作是不允许的。这些上下文环境包括初始化环境、线程环境、ISR 环境和 DSR 环境。

在初始化环境中,中断被禁止,调度器被锁定。在这种环境中的程序不允许重新使能中断和对调度器解锁,这是因为在这种环境下系统的完全一致性得不到保证。因此,初始化程序不能使用诸如 cyg_semaphore_wait 这样的同步原语来等待外部事件的发生。对互斥体的锁定和解锁是允许的,此时没有其他的线程运行,从而可以保证互斥体还没有被锁定,因此对它的锁定操作不会被阻塞。这有助于可能使用了内部互斥体的库调用。

系统在启动序列的最后阶段将调用 cyg_scheduler_start 函数,各种线程将开始运行。在线程环境下,几乎所有的内核函数都是可用的。根据具体目标平台硬件的特性,在进行中断相

关的操作时可能会有一些限制。例如,硬件可能要求在返回线程环境之前在 ISR 或 DSR 内进行中断应答。在这种情况下,cyg _ interrupt _ acknowledge 函数不应该被线程调用。

处理器在任何时刻都有可能收到外部中断,引起控制权从当前线程被转移。此时系统通常运行的是 eCos 提供的 VSR,由它来确定发生的是哪一个中断,然后再从 VSR 转移到适当的 ISR。ISR 可能由硬件抽象层或者设备驱动程序提供,也可能是由应用程序提供。此时系统运行在 ISR 环境,大多数的内核函数调用是不允许的,包括各种同步原语。在 ISR 内部,通常所做的只能是与中断机制自己相关的一些操作,如屏蔽一个中断或者应答一个已经得到处理的中断。在 SMP 系统中可以使用 spinlock。

当 ISR 返回时,它可能请求运行相应的 DSR。DSR 一旦安全就可以运行,此时将运行在 DSR 环境。这种环境可用于运行告警器函数,线程可以通过锁定调度器与 DSR 进行临时的上下文切换。DSR 环境下也只能调用部分内核函数,但能够调用的函数要比 ISR 环境要多。尤其是它可以使用一些同步原语而不会引起阻塞,这些同步原语包括 cyg _ semaphore _ post、cyg _ cond _ signal、cyg _ cond _ broadcast、cyg _ flag _ setbits 和 cyg _ mbox _ tryput。不能使用那些可能引起阻塞的同步原语如 cyg _ semaphore _ wait、cyg _ mutex _ lock 或 cyg _ mbox _ put,在 DSR 内调用这些函数可能引起系统的挂起。

5.8.2 应用程序编程要求

eCos 是一个实时操作系统,基于 eCos 的程序与基于 UNIX 和 Windows 这些分时系统和虚拟内存系统的程序不完全相同。在进行 eCos 应用程序编程时必须了解相应的编程要求。

eCos 用户应用程序的入口点是 cyg _ user _ start ( ),如果在配置时选择了 ISO C 库包,也可以使用 main ( ) 作为程序的入口点。eCos 的内核 API 调用函数定义于头文件 < cyg/kernel/kapi.h > 内,任何一个使用 eCos 系统内核调用的程序必须在其源程序文件最前面使用下面语句:

```
#include <cyg/kernel/kapi.h>
```

在编程时必须注意头文件路径的设置。其路径可以通过一个系统环境变量 C _ INCLUDE _ PATH 进行设置,也可以在编译器命令行中使用"-I"标志对其进行指定。

在完成对 eCos 的配置后,可以对其进行编译,编译结果将形成一个单独的库,即 libtarget.a,它包含了配置时所有被选择的 eCos 组件。libtarget.a 不包含其他用户库。如果需要使用其他的用户库,在链接命令中必须对所使用的用户库加以指明。

利用 eCos 配置工具对 eCos 进行编译后得到的结果不能直接加载到目标系统运行。RedBoot 是一个例外,RedBoot 编译后得到的结果可以直接加载到目标系统内存或者 ROM 内运行。用户应用程序必须与前面提及的对 eCos 进行编译后得到的 libtarget.a 和 GNC C 编译器提供的库 libgcc.a 进行链接,链接时还必须使用 eCos 编译后得到的链接脚本 target.ld。进行链接的时候不能使用标准 C++ 库,可以用 -nostdlib 选项来加以说明。对应用程序进行链接的命令如下 (其中 GCC 是目标平台的交叉编译器,如 i386-elf-gcc):

```
GCC [options] [object files] -Ttarget.ld -nostdlib
```

编译得到的最终结果包含了 eCos 系统和应用程序。

5.8.3 应用程序的启动

经过编译和链接的 eCos 在目标平台上运行时，首先运行的是硬件抽象层中的启动代码。硬件抽象层中包含了所有的启动代码，其主要启动过程可以概括为：

1）HAL 对硬件进行初始化操作，与 ROM 监控程序协同工作，并进行诊断操作。

2）激活所有静态和全局 C++ 静态构造函数。

3）HAL 跳转到启动函数 void cyg _ start (void)。

下面分别介绍 cyg _ start () 函数和该函数内部的一些与启动应用程序相关的几个函数，包括用户应用程序入口 cyg _ user _ start () 函数。

1. 启动函数 cyg _ start ()

启动 cyg _ start () 函数是 eCos 启动机制的核心。该函数位于源文件 infra/current/src/startup.cxx，它依次调用下列函数：

```
cyg _ prestart ()
cyg _ package _ start ()
cyg _ user _ start ()
```

如果在配置时选择了调度器，它还将启动被选择的 eCos 调度器。

这只是 eCos 所提供的一个实现方法。用户在进行自己的开发时，也可以使用下面的函数原型对该函数进行修改：

```
void cyg _ start (void)
```

一般来说不需要对该函数进行修改，因为 cyg _ prestart () 函数和 cyg _ user _ start () 函数具有足够的灵活性，允许用户加入其他程序代码，几乎可以满足所有的应用需求。

2. 函数 cyg _ prestart ()

该函数位于源文件 infra/current/src/prestart.cxx 内。eCos 提供了默认的 cyg _ prestart () 函数，不做任何操作。如果在进行其他系统级初始化操作之前还需要进行某些初始化操作，则可以使用该函数来实现。该函数原型为：

```
void cyg _ prestart ( void )
```

3. 函数 cyg _ package _ start ()

该函数位于源程序 infra/current/src/pkgstart.cxx 内。它允许在进入用户主程序之前对个别包进行初始化操作。cyg _ package _ start () 函数包含了两个包：μITRON 包和标准 C 库包。基础结构包中包含了两个配置选项 CYGSEM _ START _ UITRON _ COMPATIBILITY 和 CYGSEM _ START _ ISO _ C _ COMPATIBILITY，它们用于控制这些特殊包的初始化。

函数原型为：

```
void cyg _ package _ start ()
```

用户可以根据该原型编写自己的 cyg _ package _ start () 函数，但在初始化默认包时必须加以小心。下面是用户编写该函数的一个例子：

```
void cyg _ package _ start (void)
```

```
{
    #ifdef CYGSEM_START_UITRON_COMPATABILITY
    cyg_uitron_start(); /* keep the uITRON initialization */
    #endif
    my_package_start(); /* make sure I initialize my package */
}
```

4. **函数** cyg_user_start()

该函数位于源文件 infra/current/src/userstart.cxx **内。** cyg_user_start() **函数是用户程序的正常入口点。**eCos **源码提供的该函数不做任何操作。这是用户创建自己线程的一个理想的地方。如果在配置时没有选择** ISO **标准** C **库包,则必须实现该函数,此时它是用户程序的一个强制性入口。其函数原型为:**

```
void cyg_user_start(void)
```

用户可以编写自己的 cyg_user_start() **函数。**eCos **为该函数提供了一个默认实现,但不做任何工作。应用程序使用该函数作为入口时,将覆盖其默认实现,实现应用程序与** eCos **系统的连接。**

当从 cyg_user_start() **函数返回时,**cyg_start() **函数将启动调度器,用户在** cyg_user_start() **函数中所产生和唤醒的所有线程都将开始执行。注意最好不要在** cyg_user_start() **函数内直接启动调度器,而应该让其自动启动。另一个值得注意的地方是由于** cyg_user_start() **函数是在调度器启动之前执行的,因此在该函数中不要使用任何需要调度器的内核服务。**

第 6 章　线程与同步

eCos 是一个抢占式多任务实时操作系统。它的基本运行单位是线程，是一个多线程系统。在某一时间段内，系统中可以存在多个活动线程，但在某一时刻只有一个线程被执行。多个线程的运行通过可配置的两种调度策略进行调度。系统中的线程可以具有不同的优先级，同一优先级的线程以时间片轮转的方式调度运行。

为允许多个线程之间的协同工作和它们对资源的竞争，需要提供一种同步和通信机制。典型的同步机制是采用互斥、条件变量和信号量，eCos 也同样采用了这些方法。此外，eCos 还采用其他一些在实时系统中普遍使用的同步/通信机制，如事件标志、消息队列等。

线程和同步机制是 eCos 内核的一个重要组成部分。本章主要介绍 eCos 线程机制和同步机制，并介绍与其相关的一些内核 API 函数。

6.1　线程的创建

线程是 eCos 的基本运行单位。简单的系统中只需要一个线程，较复杂的系统需要使用多线程。eCos 内核提供一个 API 函数 cyg _ thread _ create 用于创建一个新的线程。在大多数系统中，只在系统初始化的时候使用该函数创建线程，并对线程所需要的数据进行动态分配。如果需要的话，也可以在其他任何时候调用该函数创建一个新的线程，但在滞后中断服务程序 DSR 中不能调用该函数。新创建的线程处于挂起状态，在调用另一个 API 函数 cyg _ thread _ resume 之前它不会启动运行，在系统初始化期间所产生的线程在 eCos 调度器被启动之前也不会运行。

6.1.1　创建新线程

创建线程的 API 函数定义于头文件＜cyg/kernel/kapi.h＞内，其函数原型为：

```
void cyg _ thread _ create
(
    cyg _ addrword _ t sched _ info,  /* scheduling info (priority) */
    cyg _ thread _ entry _ t * entry, /* thread entry point */
    cyg _ addrword _ t entry _ data, /* entry point argument */
    char * name,                     /* name of thread */
    void * stack _ base,             /* pointer to stack base */
    cyg _ ucount32 stack _ size,     /* size of stack in bytes */
    cyg _ handle _ t * handle,       /* returned thread handle */
    cyg _ thread * thread            /* space to store thread data */
)
```

新产生的线程都分配有一个惟一的句柄（即线程 ID），它通过参数 handle 带回。此后所有

对该线程的操作都使用该句柄来表示这个线程。参数 name 是该线程的名字，主要用于调试目的，使用它可以容易跟踪 cyg _ thread 结构与哪一个应用级的线程相对应。内核的一个配置选项 CYGVAR _ KERNEL _ THREADS _ NAME 用于控制是否使用线程名字。eCos 配置工具对线程的配置如图 6-1 所示。

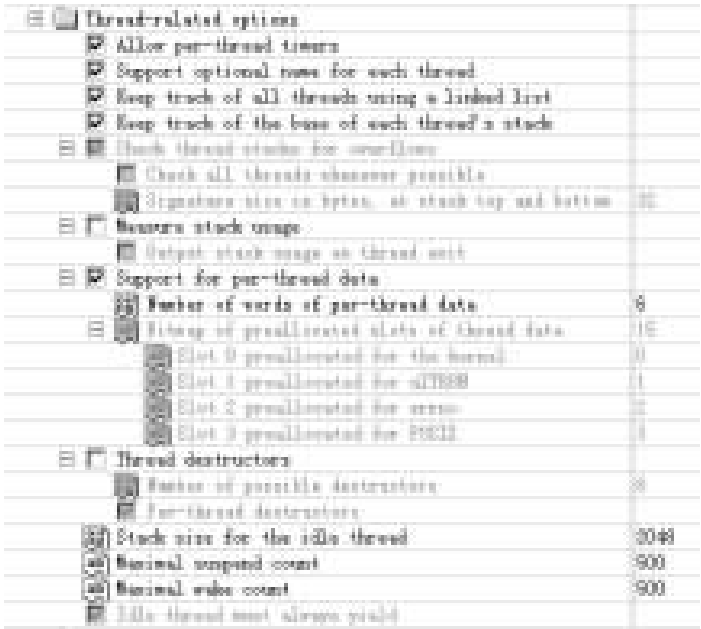


图 6-1　eCos 配置工具对线程相关部分的配置

对于每一个线程，内核都需要为其提供一个小的内存空间，该空间具有 cyg _ thread 数据结构的形式，用于存放线程的相关信息（如线程的当前状态）。为避免在内核内进行动态内存分配，该内存空间由高层程序进行分配，通常使用静态变量的形式。参数 thread 提供了该空间。

参数 entry 和 entry _ data 分别是线程的入口函数和入口函数的参数，stack _ base 和 stack _ size 分别是分配给该线程的栈和栈大小。参数 sched _ info 提供了线程的优先级信息。

6.1.2　线程入口函数

线程创建函数中的第二个参数指定了新线程的入口函数。线程入口函数的格式如下：

```
void
thread _ entry _ function (cyg _ addrword _ t data)
 {
      ...
}
```

其参数 data 是线程创建函数的第三个参数。线程创建函数的这个参数通常是一个指向一些静态数据的指针，或者是一个小的整数。如果该线程不需要任何数据，则线程创建函数的该参数为 0。新创建的线程在被 cyg _ thread _ resume 函数启动运行后，将进入该线程的入口

函数。

如果线程入口函数始终能够返回,则它等同于一个调用了 cyg _ thread _ exit 函数返回的线程。当线程返回后,即使线程不会再次运行,仍然保留它在调度器中的注册。如果应用程序需要回收该线程的 cyg _ thread 数据结果,则必须调用 cyg _ thread _ delete 函数将该线程删除。

6.1.3 线程优先级

线程创建函数的第一个参数 sched _ info 给调度器提供了一些有关该线程的优先级信息。信息的详细内容与使用的调度器相关。对于位图调度器和多级队列调度器,它是一个小的整数,通常是 0～31 之间的一个值,表示该线程的优先级,其中 0 的优先级最高。最低优先级 CYG _ THREAD _ MIN _ PRIORITY 通常只有系统的空闲线程使用。优先级的具体数目可以通过内核配置选项 CYGNUM _ KERNEL _ SCHED _ PRIORITIES 进行控制。在使用位图调度器时,同一优先级不能有两个线程。

在开发 eCos 应用程序时,必须了解系统中的各种线程(包括 eCos 包所产生的线程),并且要保证所有的线程都以适当的优先级运行。

内核提供了三个 API 函数:cyg _ thread _ set _ priority、cyg _ thread _ get _ priority 和 cyg _ thread _ get _ current _ priority,用于对线程的优先级进行操作。

6.1.4 堆栈和堆栈大小

每一个线程都需要有自己的堆栈空间,用于本地变量和跟踪函数的调用和返回。堆栈的分配应该由调用程序以静态数据的形式提供,以避免内核对它进行动态内存分配操作。线程创建函数 cyg _ thread _ create 有两个参数与堆栈有关,一个参数指向堆栈的基地址,另一个参数指明了堆栈的大小。在许多处理器结构中,堆栈通常是自顶向下分配的,因此内核确定堆栈的起始地址的方法是基地址加上堆栈大小。

线程对堆栈大小的具体要求与许多因素有关。最主要的一个因素是在该线程环境下执行的程序,如果这些程序使用了大量的嵌套调用、递归或大的数组,则堆栈大小应该设置为适当高的值。CPU 体系结构方面对堆栈的大小也有影响,例如,CPU 寄存器的数目以及调用约定都对堆栈的使用有影响。另外,根据具体的配置情况,有时可能有其他的一些程序(如中断处理程序)会在当前线程堆栈中运行。这种情况可以使用一些配置选项(如配置选项 CYGIMP _ HAL _ OMMON _ INTERRUPTS _ USE _ INTERRUPT _ STACK 以及 CYGSEM _ HAL _ COMMON _ INTERRUPTS _ ALLOW _ NESTING 等)对它们进行控制。

开发应用程序时,应该由开发人员自己确定应用程序实际所需的堆栈大小,内核无法事先知道线程中将运行什么程序。为此,系统提供了两个常数,它们用于指导选择合理的堆栈大小。这两个常数分别是 CYGNUM _ HAL _ STACK _ SIZE _ MINIMUM 和 CYGNUM _ HAL _ STACK _ SIZE _ TYPICAL,它们被定义在硬件抽象层中。MINIMUM 值(最小堆栈大小)适合于那些只运行一个函数而且只有简单的系统调用的线程。TYPICAL 值(典型堆栈大小)适合于那些嵌套调用层次不多并且没有太大数组的应用程序。

如果对堆栈的大小估计不足,则有可能发生堆栈溢出,其结果是可能会破坏内存数据。出现这种情况时,用常规调试手段很难跟踪到错误。内核提供了一些程序代码用于帮助对堆栈溢出的检测,配置选项 CYGFUN _ KERNEL _ THREADS _ STACK _ CHECKING 可以对此

进行控制。使能该选项时，在堆栈的界限处保留少量的空间，该空间具有特殊的标记，每一次发生线程上下文切换时对该标记进行检查，如果标记被改变则表示发生了堆栈溢出。当系统的 Debug 功能被使能时，这种堆栈检查方法处于默认使能状态。

6.1.5 线程创建例子程序

下面举例说明线程是如何创建的。该例产生了五个线程，一个是 producer，另外四个是 worker。线程创建于系统的 cyg _ user _ start 函数，也可以在 main 入口函数内产生。

线程创建例子程序：

```
----------------example program for create thread----------------------
#include <cyg/hal/hal_arch.h>
#include <cyg/kernel/kapi.h>

//These numbers depend entirely on your application
#define NUMBER_OF_WORKERS      4
#define PRODUCER_PRIORITY    10
#define WORKER_PRIORITY       11
#define PRODUCER_STACKSIZE CYGNUM_HAL_STACK_SIZE_TYPICAL
#define WORKER_STACKSIZE  (CYGNUM_HAL_STACK_SIZE_MINIMUM + 1024)
static unsigned char producer_stack[PRODUCER_STACKSIZE];

static unsigned char worker_stacks[NUMBER_OF_WORKERS][WORKER_STACKSIZE];
static cyg_handle_t producer_handle, worker_handles[NUMBER_OF_WORKERS];
static cyg_thread_t producer_thread, worker_threads[NUMBER_OF_WORKERS];

static void
producer(cyg_addrword_t data)
{
    ...
}

static void
worker(cyg_addrword_t data)
{
    ...
}

void
cyg_user_start(void)
{
    int i;
    cyg_thread_create(PRODUCER_PRIORITY, &producer, 0, "producer",
                      producer_stack, PRODUCER_STACKSIZE,
```

```
                &producer _ handle, &producer _ thread) ;
    cyg _ thread _ resume (producer _ handle) ;
    for (i = 0 ;i < NUMBER _ OF _ WORKERS ;i++) {
        cyg _ thread _ create (WORKER _ PRIORITY, &worker, i, "worker",
                worker _ stacks [i], WORKER _ STACKSIZE,
                & (worker _ handles [i]), & (worker _ threads [i])) ;
        cyg _ thread _ resume (worker _ handles [i]) ;
    }
}
```

6.2 线程信息的获取

对线程进行操作时,有时需要了解线程相关的一些信息。内核提供了一些函数用于获取线程的信息,包括线程句柄、线程堆栈基地址和线程堆栈大小等。下面分别对这些 API 函数进行简单的介绍。

```
cyg _ handle _ t cyg _ thread _ self (void)
```

该函数返回当前线程的句柄。其值与当前线程在创建时被 cyg _ thread _ create 函数返回的 handle 值相同。调用该函数得到的句柄可以被其他线程函数使用。cyg _ thread _ self 函数只能被当前线程调用。

```
cyg _ handle _ t cyg _ thread _ idle _ thread (void)
```

该函数返回空闲线程的句柄。空闲线程是内核自动产生的一个线程,应用程序除了使用该函数外,没有其他方法可以获取这一信息。该函数可以在线程和 DSR 环境下调用,但只能在系统初始化完成之后进行。

```
cyg _ addrword _ t cyg _ thread _ get _ stack _ base (cyg _ handle _ t thread)
```

该函数返回指定线程的堆栈基地址。返回的基地址可能与在创建该线程时传递给 cyg _ thread _ create 函数的值相同,也可能不一致。这是因为为了满足 debug 调试和字节排列的需要,有可能对它的堆栈基地址进行了修改。在线程被创建后,可以在任何时间调用该函数来获取线程的实际堆栈基地址。

```
cyg _ uint32 cyg _ thread _ get _ stack _ size (cyg _ handle _ t thread)
```

该函数返回指定线程的堆栈大小。返回的堆栈大小可能与在创建该线程时传递给 cyg _ thread _ create 函数的值相同,也可能不一致,这是因为为了满足 debug 调试和字节排列的需要,有可能对线程堆栈大小进行了修改。在线程被创建后,可以在任何时间调用该函数来获取线程的实际堆栈大小。

```
cyg _ uint32 cyg _ thread _ measure _ stack _ usage (cyg _ handle _ t thread) ;
```

该函数只有在配置选项 CYGFUN _ KERNEL _ THREADS _ STACK _ MEASUREMENT 被使能时才可用。它的返回值是指定线程到目前为止所使用的堆栈空间的最大字节数。返回

的值不一定是堆栈空间的实际上限。如果返回 0,则有可能会发生堆栈溢出。这是一个必需的 debug 函数。

6.3 线程的控制

新线程的启动、线程在什么时候被唤醒以及挂起一个线程需要有一个机制对它们进行控制。eCos 内核提供了一些 API 函数对这些行为进行控制。应用程序在使用这些函数时,有时还需要适当地使用条件变量或者信箱等一些同步原语。下面是这些 API 函数的介绍。

```
void cyg_thread_yield(void)
```

该函数使线程放弃对处理器的控制,允许另一个相同优先级的线程运行。这对具有比当前线程优先级高的线程没有任何作用,因为如果有比当前线程优先级要高的线程,则当前线程自然会被优先级高的线程抢先。同样,对于优先级比当前线程要低的线程也不会有任何作用,因为当前线程总是会抢先在优先级低的线程之前运行。因此,该函数只在使用多级队列调度器时允许多个线程具有相同优先级的情况下才有用。如果没有相同优先级的其他线程,该函数将不起作用。

即使采用了多级队列调度器,cyg_thread_yield 函数也不一定有用。由于使用了时间片轮转,同一优先级的所有线程都可以公平地得到 CPU 时间。如果使用配置选项 CYGSEM_KERNEL_SCHED_TIMESLICE 将时间片禁止,那么可以使用这一函数来实现多任务之间的协同工作。

```
void cyg_thread_delay(cyg_tick_count_t delay)
```

该函数将线程挂起一定的时间,参数 delay 为被挂起的时钟滴答数。假设系统时钟频率为 100Hz,如果 delay 为 1,那么线程将被挂起 10ms。线程控制的这种挂起功能依赖于系统的实时时钟,受到配置选项 CYGVAR_KERNEL_COUNTERS_CLOCK 的控制。

如果应用程序要求使用毫秒或类似的时间单位而不使用时钟滴答,则必须进行转换计算,通常需要由开发者自己进行这种计算。程序在每一次调用该函数之前进行这种计算势必会增加系统不必要的开销。

```
void cyg_thread_suspend(cyg_handle_t thread)
```

该函数挂起指定的线程。线程可以被挂起多次,每调用一次该函数,都必须相应地调用一次 cyg_thread_resume 函数,使线程走出挂起状态。

每一个线程都有一个悬挂计数器,当线程被创建时,该计数器的值为 1。调用该函数时将使悬挂计数器加 1,而 cyg_thread_resume 函数将使悬挂计数器减 1。调度器不会调度运行一个悬挂计数器不为 0 的线程,因此新创建的线程在被启动之前不会运行。

```
void cyg_thread_resume(cyg_handle_t thread)
```

该函数启动指定线程重新运行,它使线程的悬挂计数器减 1。如果计数器的值变为 0,处于挂起状态的线程将被恢复并继续运行。如果线程被退出,则重新被初始化。

在对线程进行挂起和恢复操作时,有时可能会出现一个问题,那就是线程被挂起的次数要

多于被恢复的参数，因此该线程将永远不会再次运行，这将导致非常严重的后果。为了帮助对该问题进行调试，内核提供了一个配置选项 CYGNUM _ KERNEL _ MAX _ SUSPEND _ COUNT _ ASSERT，它给出了一个悬挂次数与恢复次数不相匹配的上限。

void cyg _ thread _ release (cyg _ handle _ t thread)

该函数将处于阻塞等待状态的线程释放出来。当一个线程在等待某个同步原语时，例如它在等待一个信号量或互斥体、或者正在等待一个告警器被触发时，使用该函数可以强迫唤醒该线程。这种操作通常会通知受到影响的同步原语返回 false，指明此次操作没有成功。使用该函数时要非常小心。应该特别加以注意的是，它只能使用在那些设计适当并对所有返回代码进行检查的线程中。该函数主要用在 POSIX 兼容层内。

6.4 线程的终止和消除

在大多数嵌入式应用中，许多线程在系统的初始化阶段被创建，这些线程采用了静态内存分配方式，它们从来不需要终止。这样避免了动态分配内存和其他资源管理的需要。然而，如果某个应用要求动态创建线程、终止线程以及回收这些线程的堆栈等资源，那么内核必须提供相应的支持。

6.4.1 线程终止函数

eCos 内核提供了三个用于终止线程的函数，它们分别是 cyg _ thread _ exit、cyg _ thread _ kill 和 cyg _ thread _ delete。

void cyg _ thread _ exit (void)

该函数允许线程自己退出。线程在执行这一函数的时候将自行退出，调度器不会再对其进行调度，但在使用 cyg _ thread _ create 函数创建该线程时的 cyg _ thraed 数据结构仍然被使用，该线程的句柄仍然有效。其他线程可以对被终止的线程进行某些操作，如使用 cyg _ thread _ measure _ stack _ usage 函数来确定它的堆栈使用情况。如果不再需要其句柄和 cyg _ thread 结构，可以使用 cyg _ thread _ delete 函数释放这些资源。如果堆栈是动态分配的，在调用 cyg _ thread _ delete 函数之前堆栈不会被释放。一般在调用该函数之前应该确认该线程的资源已经释放，否则可能引起系统的崩溃。

void cyg _ thread _ kill (cyg _ handle _ t thread)

该函数用于杀死另外的一个线程。其效果与被杀死的线程自己调用 cyg _ thread _ exit 函数相同。该函数的使用是相当危险的，如果线程在被杀死的时候还有一些分配给它的资源（包括内存、信号量、互斥体等等）没有被释放，则有可能造成系统死锁。因此应该尽量避免使用该函数，最好使用 cyg _ thread _ exit 函数来终止线程。该函数不能被用来杀死线程自己。

void cyg _ thread _ delete (cyg _ handle _ t thread)

该函数删除一个线程。在一个线程已经退出并且不再需要的时候，可以使用该函数将其删除。调用该函数后，被删除线程的句柄不再有效，cyg _ thread 数据结构和线程堆栈可以被

重新使用或被释放。对一个仍在运行的线程调用该函数将其删除时隐含调用了 cyg _ thread _ kill 函数。使用该函数也是危险的,如果可以的话,可以给被删除的线程发送一个消息让它自己使用 cyg _ thread _ exit 函数退出。

6.4.2 线程消除函数

有些应用要求线程在退出的时候能自动调用一个函数,释放被该线程所占用的资源。这个函数称为线程的消除函数。这种支持必须使用配置选项 CYGPKG _ KERNEL _ THREADS _ DESTRUCTORS 对其进行使能。eCos 内核提供了两个对线程消除函数进行管理的 API 函数。

```
cyg _ bool _ t cyg _ thread _ add _ destructor
(
    cyg _ thread _ destructor _ fn fn, /* destructor function */
    cyg _ addrword _ t data           /* argument to destructor */
)
```

该函数为调用此函数的线程增加一个消除函数 fn,当该线程退出时将执行这一消除函数。消除函数的函数类型为 cyg _ thread _ destructor _ fn,可以将参数 data 作为消除函数的参数。如果成功增加消除函数,该函数返回 true,否则返回 false。

```
cyg _ bool _ t cyg _ thread _ rem _ destructor
(
    cyg _ thread _ destructor _ fn fn, /* destructor function */
    cyg _ addrword _ t data           /* argument to destructor */
)
```

该函数将从调用该函数的线程中删除(禁止)消除函数。在线程退出或被杀死之前应该调用消除函数。为了使消除函数删除成功,该函数中的 fn 和 data 参数必须与安装消除函数时的 cyg _ thread _ add _ destructor 函数的参数完全一致。如果成功删除消除函数,该函数返回 true,否则返回 false。

在默认情况下,线程的消除函数只对一个线程有效,这意味着注册一个消除函数仅仅是只将其注册到当前线程。每一个线程都有自己的消除函数。如果在配置的时候禁止了配置选项 CYGSEM _ KERNEL _ THREADS _ DESTRUCTORS _ PER _ THREAD,那么任何一个线程在退出时都将调用所有被注册的消除函数。也就是说如果禁止了该配置选项,那么线程消除函数是全局性的,所有线程都有相同的消除函数。

对于被注册的消除函数的个数有一定的限制,配置选项 CYGNUM _ KERNEL _ THREADS _ DESTRUCTORS 可以对它进行控制。在增加该配置选项的值时,所使用的内存增加量很小,但当配置选项 CYGSEM _ KERNEL _ THREADS _ DESTRUCTORS _ PER _ THREAD 被使能的时候,每个线程所使用的内存都将有所增加。当消除函数的注册个数达到限定值时,cyg _ thread _ add _ destructor 函数将返回 false。

6.5 线程优先级操作

调度器使用线程优先级来决定下一次启动哪一个线程运行。优先级的组成与具体选择的

调度器有关。一般来说,线程的优先级使用 0～31 的整数来表示,只有空闲线程运行于最低优先级。优先级数目可以在 eCos 配置工具中指定,相应的配置选项是 CYGNUM _ KERNEL _ SCHED _ PRIORITIES。

eCos 内核提供了三个对线程优先级进行操作的 API 函数。通过它们可以取得线程的优先级或者对线程优先级进行设置。

```
cyg _ priority _ t cyg _ thread _ get _ priority
(
    cyg _ handle _ t thread /* thread ID */
)
```

该函数返回指定线程的优先级。这个优先级是对该线程进行最近的一次优先级设置(使用 cyg _ thread _ set _ priority 函数)所设定的值,或者是线程被首次创建时的优先级。在某些情况下,线程可能实际运行在一个更高的优先级上。为解决优先级倒置问题,它有可能拥有一个互斥体,并使用优先级置顶或者优先级继承等措施,它的优先级可能被临时提高。在这种情况下,该函数返回的不是线程的当前优先级,而是正常设置的优先级。优先级的倒置问题将在后面介绍。

```
cyg _ priority _ t cyg _ thread _ get _ current _ priority
(
    cyg _ handle _ t thread /* thread ID */
)
```

该函数返回指定线程的当前优先级。如果线程的优先级被采用了优先级置顶或优先级继承等机制的互斥体所改变,该函数将返回被改变之后的当前优先级,而不是正常设置的优先级。

```
void cyg _ thread _ set _ priority
(
    cyg _ handle _ t thread,      /* thread ID */
    cyg _ priority _ t priority   /* new priority */
)
```

该函数将指定线程的优先级修改为指定的值。与 UNIX 一样,优先级的值越小其优先级越高。优先级 0 的线程在系统中的优先级最高。配置选项 CYG _ THREAD _ MIN _ PRIORITY 指定的优先级是最低优先级,它的值与所使用的调度器有关。在许多应用中,可以给线程静态指定一个合适的优先级。但有时需要对线程的优先级进行动态修改,此时可以使用该函数实现这一目的。

6.6 per-thread 数据

在一些应用程序和某些程序库中,每一个线程都可能需要用到一些专用数据。例如,在 POSIX 兼容包中的许多函数用-1 的返回值来表示一个错误,并使用一个全局变量 error 来保存一些附加的信息。如果有多个线程并发调用 POXIS 库的函数,而且 error 是一个全局变量,

那么线程将无法知道当前的 error 是否是它自己所进行调用的返回值，也无法知道是否是同时运行的其他线程在调用不同的 POSIX 函数时对这个全局变量进行了修改。为避免这种混淆，error 应该作为一个线程专用数据，即 pre-thread 数据，每一个线程都应该有它的一个实例。

这种对 pre _ thread 数据的支持可以通过配置选项 CYGVAR _ KERNEL _ THREADS _ DATA 进行使能控制。如果被使能，每一个 cyg _ thread 数据结构都将保持有一个小的字(word)数组，该数组的大小由配置选项 CYGNUM _ KERNEL _ THREADS _ DATA _ MAX 来指定。当线程被创建时，该数组的所有元素被置 0。

如果应用程序需要使用 pre-thread 数据，它需要该数组的一个还没有分配给其他程序的索引。这个索引可以通过调用内核提供的 API 函数 cyg _ thread _ new _ data _ index 得到，并被随后调用的另一个 API 函数 cyg _ thread _ get _ data 所使用。索引的分配通常在系统初始化期间进行，并保存在具体变量内。由于某种原因不再需要数组中的某个槽位并且可以重新使用它的时候，可以调用 cyg _ thread _ free _ data _ index 函数将它释放。

可以使用 API 函数 cyg _ thread _ get _ data 获取当前线程在给定槽位上的 pre-thread 数据。使用 cyg _ thread _ set _ data 函数可以对 pre-thread 数据进行更新。如果 pre-thread 的某个特殊项被重复使用，可以使用 cyg _ thread _ get _ data _ ptr 函数来获取数据的指针，利用这个指针可以有效地对数据进行间接检查和更新。

一些包(如错误包和 POSIX 包)在 per-thread 数据中具有预先分配的槽。正常情况下，应用程序不得使用这些槽，而应该在初始化期间调用 cyg _ thread _ new _ data _ index 函数进行分配。假如在配置时不包含 POSIX 包，那么应用程序可以重新使用这些预先分配给这些未使用包的槽位。

这些对 pre-thread 数据进行操作的内核 API 函数定义于头文件＜cyg/kernel/kapi.h＞内，下面分别对它们加以介绍。

```
cyg _ ucount32 cyg _ thread _ new _ data _ index (void)
```

该函数获取一个新的没有被使用的数据索引。被分配的索引可用于保存每个线程的专用数据。例如，全局变量 error 可以被分配为 pre _ thread 数据变量。如果没有新的索引，该函数返回 - 1。

```
void cyg _ thread _ free _ data _ index
(
    cyg _ ucount32 index /* index to free */
)
```

该函数释放一个数据索引，系统可以重新使用该索引。

```
CYG _ ADDRWORD cyg _ thread _ get _ data
(
    cyg _ ucount32 index /* index of per thread data */
)
```

该函数获取 pre-thread 数据。

```
CYG _ ADDRWORD * cyg _ thread _ get _ data _ ptr
```

```
(
    cyg_ucount32 index /* index of per thread data */
)
```

该函数返回 per-thread 数据的指针。在使用 cyg_thread_get_data 和 cyg_thread_set_data 函数时可以使用该函数。该指针只在当前线程的上下文环境中有效。

```
void cyg_thread_set_data
(
    cyg_ucount32 index, /* index of per thread data */
    CYG_ADDRWORD data /* data to write */
)
```

该函数对 pre-thread 数据进行设置。

6.7 同步原语

eCos 是一个多线程系统,为保证多个线程能够协同工作和它们对资源的合理竞争,它提供一种同步和通信机制。eCos 内核的同步机制提供了许多同步原语,包括互斥、条件变量、信号量,信箱和事件标志等。

互斥 (Mutexes) 允许多个线程能安全地共享资源。线程首先必须锁定互斥体,然后再对共享资源进行操作,最后要对该互斥体进行解锁。其他的同步原语用于线程之间的通信,或者用于滞后中断服务程序 DSR 与线程的通信。

线程在锁定一个互斥体的时候可能需要等待某个条件得到满足,这就需要使用条件变量 (Condition Variables)。条件变量只在线程需要等待的时候才使用,其他线程或 DSR 可以唤醒处于等待状态的该线程。当线程等待一个条件变量时,它在进入等待状态之前将释放互斥体,在被唤醒后又重新拥有互斥体。这种操作是原子操作,不需要使用同步竞争条件。

信号量 (Semaphores) 用于指明某个特殊事件的发生。线程可以等待某个事件的发生,而该事件将由另一个线程产生。信号量具有一个计数器,如果事件连续快速发生多次,也不会丢失信息。

信箱 (Mail boxes) 也可以用来表示已经发生的某个特殊事件,允许在每个事件之间交换数据。由于信箱需要保存数据,因此它的能力有限。如果线程产生信箱事件的速度快于对这些事件的消耗,为避免溢出,该线程将被阻塞直到具有可用的信箱空间。因此,信箱通常不能被 DSR 用来唤醒线程,它只能用于两个线程之间。

事件标志 (Event Flags) 可以用于等待一定数量的不同事件的发生,并对发生的事件发出信号。与信号量要对事件发生的次数进行跟踪不同,事件标志关心的只是事件是否发生,而不是其发生的次数。它不可能像信箱那样随事件一起发送数据,它不可能出现溢出,因此它可以在 DSR 和线程之间已及两个线程之间使用。

Spinlock 是为运行在 SMP 环境下的线程提供的一个附加的同步原语。

eCos 的硬件抽象层提供了一些包含上述这些同步原语的设备驱动程序 API 函数,允许中断处理程序的 DSR 给高层软件发出事件信号。如果是一个包含内核的配置,驱动程序 API 函

数将被直接映射到同等的内核函数。如果是一个没有内核的配置，并且应用程序只是一个以查询方式检测 I/O 事件发生的单线程的程序，那么驱动程序 API 将完全实现于硬件抽象层内，不需要对多线程进行考虑，其实现显然非常简单。

利用 eCos 的图形配置工具可以对同步机制进行配置，图 6-2 是对同步原语进行配置的一个示意图。

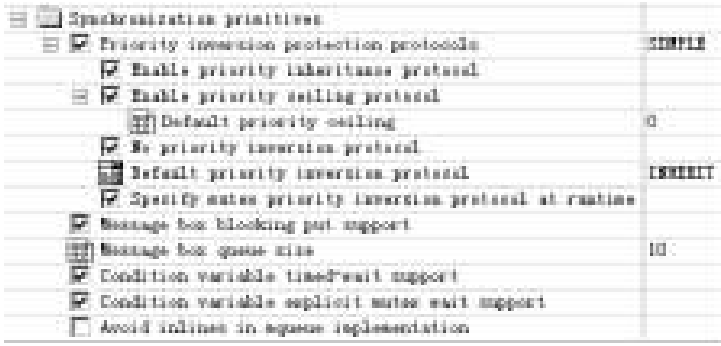


图 6-2　eCos 配置工具对同步原语的配置

6.8　互斥体

使用互斥体 (Mutex) 的目的是实现线程对资源的安全共享。如果两个或两个以上的线程要对同一个数据结构进行操作而没有对它进行锁定，系统可能在当时不会出现什么问题，但不久以后数据结构将会变得不一致，应用程序可能会出现异常并最终导致系统崩溃。这种现象也可能出现在使用单个变量或其他资源的时候。

考虑下面的程序例子：

```
static volatile int counter = 0;
void
process_event (void)
{
    ...
    counter++;
}
```

假设在某个时候 counter 的值是 42，此时有两个在同一优先级上运行的线程 A 和 B，它们都调用上面的 process_event 函数。线程 A 将读取 counter 的值，并将其值加 1，此时 counter 为 43。线程 B 也做同样的操作，counter 的值为 44。但是如果线程 A 在读取 counter 的值为 42 之后，在将其值加 1 之前调度器调度运行线程 B，此时线程 B 读取的仍然是 counter 原来的值 42，操作完成后 counter 变为 43。这样 counter 的值只增加了 1，而不是 2，因此最后 counter 的值只是 43，而不是 44。这足以说明该应用程序的运行将是不可靠的。

像上面例子中对共享数据进行操作的程序代码段通常被称作临界区。为避免出现上述现象，程序在进入临界区之前应该声称一个锁，在离开时再释放这个锁。互斥体就是为此而实现的一个同步原语。

上面的例子可以按下面的方法使用互斥体：

```
static volatile int counter = 0;
static cyg_mutex_t lock;
void
process_event(void)
{
    ...
    cyg_mutex_lock(&lock);
    counter++;
    cyg_mutex_unlock(&lock);
}
```

6.8.1 互斥体的实现与操作

在使用互斥体之前，必须调用内核 API 函数 cyg_mutex_init 对互斥体进行初始化。互斥体是一个 cyg_mutex_t 类型的数据结构，通常被静态分配。它有可能是某个大的数据结构的一部分。如果不再需要某个互斥体，并且没有任何线程正在等待该互斥体，则可以调用 cyg_mutex_destroy 函数将其作废。

使用互斥体的主要函数是 cyg_mutex_lock 和 cyg_mutex_unlock。在正常情况下，cyg_mutex_lock 在成功对互斥锁进行声称后将返回 true，如果互斥体被另外的线程所拥有，该线程将被阻塞等待该互斥体。如果其他程序调用 cyg_mutex_release 函数或者 cyg_thread_release 函数，则这种锁定操作有可能失败。因此在使用这些函数时应该检查它们的返回值。当前拥有互斥体的线程在不再需要该互斥体的时候，可以调用 cyg_mutex_unlock 函数对互斥体进行解锁，解锁操作必须由互斥体的拥有者进行，而不能由另外的线程进行。

cyg_mutex_trylock 函数是 cyg_mutex_lock 的一个变体，在被调用后将立即返回。这个函数很少被使用。由于程序只是在进入临界区之前才对互斥体进行锁定，因此如果不能锁定互斥体，那么当前线程将不会做其他工作。如果线程运行的优先级较低，它在使用这个函数时可能会发生优先级倒置现象，因为优先级继承程序可能还没有被启动。

cyg_mutex_release 函数可以用于唤醒那些由于调用了 cyg_mutex_lock 函数而当前正处于被阻塞状态的所有线程。这些线程对 cyg_mutex_lock 的调用可能返回 false，但不会影响当前拥有该互斥体的线程。

eCos 内核互斥体的实现不支持递归锁。如果线程锁定了一个互斥体，那么试图对该互斥体进行再一次的锁定操作（通常在复杂的递归调用中出现）将导致错误的发生，或者导致线程进入死锁状态。

6.8.2 互斥体 API 函数

上面提到的 eCos 内核对互斥体进行操作的 API 函数均被定义在头文件＜cyg/kernel/kapi.h＞内，使用这些函数时应该包含该头文件。下面是这些 API 函数的简要介绍。

```
void cyg_mutex_init
(
```

```
    cyg_mutex_t *mutex /* mutex to initialize */
)
```

该函数初始化一个互斥体。在 eCos 系统中,同一个线程不能对互斥体进行多次锁定。如果对同一个线程进行了多次锁定,将会发生无法预料的结果。

```
void cyg_mutex_destroy
(
    cyg_mutex_t *mutex /* mutex to destroy (invalidate) */
)
```

该函数使一个互斥体作废(失效)。使用该函数时应该注意没有其他线程正在等待或使用该互斥体。如果作废一个正在被使用的互斥体,则有可能会造成系统死锁。

```
cyg_bool_t cyg_mutex_lock
(
    cyg_mutex_t *mutex /* mutex to lock */
)
```

该函数锁定一个互斥体。如果互斥体不可用,线程将被阻塞直到该互斥体可用,或者直到该线程被一个信号唤醒。如果互斥体被成功锁定则返回 true,否则返回 false。

```
cyg_bool_t cyg_mutex_trylock
(
    cyg_mutex_t *mutex /* mutex to attempt lock */
)
```

该函数尝试对互斥体进行锁定。如果互斥体不可用,则返回 false,否则返回 true。

```
void cyg_mutex_unlock
(
    cyg_mutex_t *mutex /* mutex to unlock */
)
```

该函数对互斥体进行解锁。对一个处于非锁定状态或被另一个线程锁定的互斥体进行解锁的结果是不确定的。

```
void cyg_mutex_release
(
    cyg_mutex_t *mutex /* mutex to release */
)
```

该函数释放所有等待指定互斥体的线程。所有正在等待该互斥体的线程将收到一个错误条件,指明没有获取该互斥体。

```
void cyg_mutex_set_ceiling
(
    cyg_mutex_t *mutex,          /* mutex to set ceiling of */
    cyg_priority_t priority      /* ceiling priority */
```

```
)
```

该函数设置互斥体的置顶优先级。只有在互斥体被设置为使用 CYG _ MUTEX _ CEILING 协议(参考下面互斥体协议设置函数)时,该函数才有意义。优先级置顶的互斥体将引起获得该互斥体的线程临时继承置顶优先级,可以避免死锁的发生。

```
void cyg _ mutex _ set _ protocol
(
    cyg _ mutex _ t  * mutex,                    /* mutex to set protocol of */
    enum cyg _ mutex _ protocol protocol          /* protocol to use */
)
```

该函数设置互斥体协议。互斥体具有下述协议:

① CYG _ MUTEX _ NONE ——没有优先级继承。

② CYG _ MUTEX _ INHERIT ——继承当前拥有互斥体的线程优先级。

③ CYG _ MUTEX _ CEILING ——继承互斥体的置顶优先级。

只有使拥有互斥体的线程优先级提升的优先级才被继承。

6.8.3 优先级倒置

实时系统中需要解决的一个问题是优先级倒置(Priority Inversion)问题。当一个高优先级任务通过同步机制(如互斥体)访问共享资源时,如果该互斥体已被一低优先级任务占有,而这个低优先级任务在访问共享资源时可能又被其他一些中等优先级的任务抢先,因此造成高优先级任务被许多具有较低优先级的任务阻塞,实时性难以得到保证。这就是优先级倒置问题。

互斥体的使用可能会引起优先级倒置问题的出现。假设有三个不同优先级的线程 A、B、C,A 运行于高优先级,B 运行于中优先级,C 运行于低优先级。线程 A 和线程 B 由于等待事件的发生而处于阻塞等待状态,因此线程 C 有机会运行。线程 C 在进入临界区时锁定了一个互斥体。此时线程 A 和线程 B 被唤醒(唤醒顺序无关紧要),线程 A 需要对同一个互斥体进行声称,但它在线程 C 离开临界区并释放该互斥体之前不得不处于等待状态。与此同时,线程 B 可以毫无问题地正常运行。由于线程 C 的优先级要比线程 B 的优先级低,它在线程 B 由于某些原因被阻塞之前将没有机会运行。这样,线程 A 将不能运行,其结果是具有高优先级的线程 A 由于优先级比它低的线程 B 的原因而不能继续运行,这就发生了优先级倒置。

有几种方法用于解决优先级倒置问题。

在简单的应用程序中,将程序代码进行适当的组织安排就有可能避免优先级倒置的发生。例如,确保互斥体不被处于不同优先级的线程所共享。但这种方法即使是在应用程序级也可能难以实现,它的底层程序可能还使用了其他的一些互斥体。因此,必须对整个系统进行仔细分析,确保不会发生优先级倒置现象。

解决优先级倒置问题普遍使用的技术是采用优先级置顶协议(Priority Ceiling Protocol)和优先级继承协议(Priority Inheritance Protocol)。较简单的解决方法是采用优先级置顶协议。优先级置顶意味着占有互斥体的线程在运行时的优先级比任何其他可以获取该互斥体的线程的优先级都要高。使用优先级置顶协议时,每个互斥体都被分配一个优先级,该优先级通常与

所有可以拥有该互斥体的线程中的最高优先级相对应。当优先级较低的线程调用函数 cyg _ mutex _ lock 或 cyg _ mutex _ trylock 占有互斥体后，该线程的优先级被提升到该互斥体的优先级。在上面的例子中，分配给互斥体的优先级将是线程 A 的优先级，线程 C 只要拥有该互斥体，它的优先级就会被提高，并且将优先于线程 B 得到运行。当线程 C 释放该互斥体后时，它的优先级又将回到原来的正常值，允许线程 A 开始运行并声称该互斥体。对互斥体进行优先级设置的函数是 cyg _ mutex _ set _ ceiling，它通常在初始化时被调用。置顶优先级可以被动态改变，但这种变化只对后面的锁定操作起作用，不会影响互斥体的当前拥有者。优先级置顶非常适合于简单应用，在这种情况下，系统中的每个线程完全有可能得知哪些互斥体将被访问。对于更复杂的应用要得知哪些互斥体可能被访问较为困难，尤其是在线程优先级被实时改变的情况下。优先级置顶有许多不足之处。它需要事先知道使用该互斥体的所有线程的最大优先级，而 eCos 的组件包通常无法知道系统中各种线程的详细信息，因此无法对组件包内部使用的互斥体设置合适的置顶优先级。如果置顶优先级太高，它就如同一个全局锁从而禁止所有的调度操作。如果内部使用的互斥体不能导出到应用程序，那么优先级置顶是不可行的。内核具有一个配置选项，该选项提供一个默认的置顶优先级，该配置选项是 CYGSEM _ KERNEL _ SYNCH _ MUTEX _ PRIORITY _ INVERSION _ PROTOCOL _ DEFAULT _ PRIORITY。

另一种更好的解决优先级倒置问题的方法就是采用优先级继承协议，它将占有互斥体的线程优先级提升到所有正在等待该互斥体的线程优先级的最高值。当一个线程等待正被另一优先级较低的线程占有的互斥体时，拥有该互斥体的线程优先级被提升到正在等待该互斥体的线程优先级。这种方法不需要事先知道即将使用该互斥体的所有线程的优先级，拥有互斥体的线程只在有更高优先级的线程等待该互斥体时才提升其优先级。它减少了对其他线程进行调度的影响。它只在需要的时候才提升线程的优先级，并且这个优先级是实时而不是静态决定的，因此它比优先级置顶协议的效率要高。优先级继承协议也存在一些缺点，由于每一次同步调用都要使用该协议，从而增加了同步调用的开销。另外，当运行在不同优先级的多个线程试图锁定单个互斥体、或者在互斥体的当前拥有者试图锁定其他的互斥体时，优先级继承协议的实现要比优先级置顶要复杂得多。

eCos 具有许多与优先级倒置相关的配置选项。如果经过仔细分析而确定系统不会出现优先级倒置时，可以禁止 CYGSEM _ KERNEL _ SYNCH _ MUTEX _ PRIORITY _ INVERSION _ PROTOCOL 组件。在大多数系统中，该组件一般都是使能的。此时需要使能 CYGSEM _ KERNEL _ SYNCH _ MUTEX _ PRIORITY _ INVERSION _ PROTOCOL _ INHERIT 和 CYGSEM _ KERNEL _ SYNCH _ MUTEX _ PRIORITY _ INVERSION _ PROTOCOL _ CEILING 两个选项中的一个，以便使这两个协议中有一个协议可用于所有的互斥体。也可以选择使用多个协议，一些互斥体采用优先级置顶，而另一些则使用优先级继承，或者根本不使用优先级倒置保护措施。所有互斥体所使用的默认协议由配置选项 CYGSEM _ KERNEL _ SYNCH _ MUTEX _ PRIORITY _ INVERSION _ PROTOCOL _ DEFAULT 进行控制，并且可以使用 cyg _ mutex _ set _ protocol 函数进行动态改变。

eCos 目前采用一种较为简单的互斥体优先级继承机制，这一机制只在采用多级队列调度器时才被使用。虽然它还不能很好地处理不常见的互斥体嵌套情形，但它具有快速性和确定性的特点。如果应用程序不需要使用互斥体优先级继承协议，在配置时可以将其禁止，这样可

以减少程序代码和数据占用的空间。eCos 将来会提供另外一些互斥体优先级继承机制，开发人员可以对它们进行选择。

6.9 条件变量

条件变量（condition variable）是允许线程同时给多个线程发信号的一个同步机制。条件变量与互斥体联合使用时可实现对某些条件的长期等待。线程在等待条件变量时，释放互斥体，在被唤醒后又重新获取该互斥体。

6.9.1 条件变量的使用

首先来看下面的一个例子。假设有一组对资源池访问进行控制的函数：

```
cyg_mutex_t res_lock;
res_t res_pool[RES_MAX];
int res_count = RES_MAX;

void res_init(void)
{
    cyg_mutex_init(&res_lock);
    <fill pool with resources>
}

res_t res_allocate(void)
{
    res_t res;
    cyg_mutex_lock(&res_lock);          //lock the mutex
    if(res_count == 0)                  //check for free resource
        res = RES_NONE;                 //return RES_NONE if none
    else
    {
        res_count--;                    //allocate a resources
        res = res_pool[res_count];
    }
    cyg_mutex_unlock(&res_lock);        //unlock the mutex
    return res;
}

void res_free(res_t res)
{
    cyg_mutex_lock(&res_lock);          //lock the mutex
    res_pool[res_count] = res;          //free the resource
    res_count++;
```

```
    cyg_mutex_unlock(&res_lock);      //unlock the mutex
}
```

上面的这个例子程序使用了一个变量 res_count 对可用资源进行跟踪。如果没有可用资源，res_allocate 函数将返回 RES_NONE，调用该函数的程序必须对该返回结果进行检查并采取适当的处理措施。假如在没有资源的情况下不想让其返回 RES_NONE，而是让它等待其中的一个资源变为可用，这就需要使用到一个条件变量。再看修改后的程序：

```
cyg_mutex_t res_lock;
cyg_cond_t res_wait;
res_t res_pool[RES_MAX];
int res_count = RES_MAX;

void res_init(void)
{
    cyg_mutex_init(&res_lock);
    cyg_cond_init(&res_wait, &res_lock);
    <fill pool with resources>
}

res_t res_allocate(void)
{
    res_t res;
    cyg_mutex_lock(&res_lock);        //lock the mutex
    while(res_count == 0)             //wait for a resources
    cyg_cond_wait(&res_wait);
    res_count--;                      //allocate a resource
    res = res_pool[res_count];
    cyg_mutex_unlock(&res_lock);      //unlock the mutex
    return res;
}

void res_free(res_t res)
{
    cyg_mutex_lock(&res_lock);        //lock the mutex
    res_pool[res_count] = res;        //free the resource
    res_count++;
    cyg_cond_signal(&res_wait);       //wake up any waiting allocators
    cyg_mutex_unlock(&res_lock);      //unlock the mutex
}
```

在这一段修改后的程序中，当 res_allocate 函数检测到没有可用资源时，它将调用内核 API 函数 cyg_cond_wait。该 API 函数完成了两个操作：解锁互斥体、将调用线程转入睡眠状态等待条件变量。当 res_free 函数最后被调用时，它将资源送回资源池中，并调用另一个

API 函数 cyg _ cond _ signal 唤醒正在等待该条件变量的线程。当处于等待状态的线程又开始运行时，在从 cyg _ cond _ wait 返回之前它将重新锁定互斥体。

上面例子中的这种操作过程有两个值得注意的地方。首先值得注意的是在 cyg _ cond _ wait 函数内对互斥体的解锁和等待操作是一种原子操作，在解锁和等待之间不会有其他线程运行。如果不是如此的话，线程在调用 res _ free 函数时所释放的资源可能会在调用 cyg _ cond _ signal 函数的时候又被丢失，从而造成被唤醒的线程在结束等待状态时将没有可用的资源。另一个值得注意的是对 cyg _ cond _ wait 函数的调用是在一个 while 循环内而不是在简单的一个 if 语句内。这是因为在重新唤醒等待线程时，cyg _ cond _ wait 函数需要重新锁定互斥体，如果有其他线程已经进入锁定该互斥体的队列，则必须等待。根据调度器和队列顺序的实际情况，在被唤醒的线程运行之前可能有许多其他的线程已经进入临界区，因此它所等待的条件变量可能再次进入 false 状态。使用循环的方式等待条件变量是惟一可以保证在等待结束时该条件变量仍为 true 的方法。

在使用一个条件变量之前，必须使用 API 函数 cyg _ cond _ init 对其进行初始化。该函数使用了两个参数，一个是该条件变量的数据结构，另一个是一个已经存在的互斥体。互斥体的初始化不在 cyg _ cond _ init 函数内进行，必须单独调用 cyg _ mutex _ init 函数对其进行初始化。如果不再需要某个条件变量，则可使用 cyg _ cond _ destroy 将其作废。

当线程需要等待某个条件被满足时，可以调用 cyg _ cond _ wait 函数。线程在调用该函数时必须已经锁定在 cyg _ cond _ init 中指定的互斥体。互斥体在 cyg _ cond _ wait 函数内被解锁，线程将被挂起，该操作是一种原子操作。当其他线程给该线程一个信号或广播操作时，该线程将被唤醒，并自动再一次声称互斥体，允许它对全局状态进行检查并确定该条件是否被满足。eCos 内核还提供了该函数的一个变体函数 cyg _ cond _ timed _ wait，它用于等待一个条件变量并具有超时功能。当 cyg _ cond _ times _ wait 函数返回时，不管它是否是超时返回还是被一个信号引起的返回，都将重新声称互斥体。

当一个线程改变共享状态时，可能会影响到其他被条件变量阻塞的线程。此时它应该调用 cyg _ cond _ signal 或 cyg _ cond _ broadcast 函数。这种调用不需要拥有互斥体，但在改变共享状态之前通常要对互斥体进行声称。一个信号只能唤醒正在等待条件变量的第一个线程，而一个广播操作则可以唤醒所有等待该条件变量的线程。如果此时没有等待条件变量的线程，信号或者广播将不会产生任何效果，信号也不会被保存或被计数。一般来说，在所有线程都检查同一个条件变量、并且最多只有一个线程能继续运行的时候可以使用信号 在线程只检查条件的细微变化或者改变一个全局变量可以引起多个线程继续运行的时候，可以使用广播。

6.9.2 条件变量 API 函数

上一节已经介绍了如何使用 eCos 内核提供的条件变量 API 函数，这些函数的定义在头文件＜cyg/kernel/kapi.h＞内。下面是这些函数的简单介绍。

```
void cyg_cond_init
(
    cyg_cond_t *cond,   /* condition variable to initialize */
    cyg_mutex_t *mutex/* associated mutex */
)
```

该函数初始化一个条件变量。使用条件变量可以让一个线程给多个线程发信号,它通常与互斥体一起使用,该互斥体与多个线程所共享的一些数据相对应。

```
void cyg_cond_destroy
(
cyg_cond_t *cond /* condition variable to destroy (invalidate) */
)
```

该函数作废(失效)一个条件变量。使用时应该注意没有其他线程正在等待或使用该条件变量。如果作废一个正在使用的条件变量,将可能造成系统死锁。

```
cyg_bool_t cyg_cond_wait
(
    cyg_cond_t *cond /* condition variable to wait for */
)
```

该函数等待一个条件变量。如果没有错误则返回 true,否则返回 false。

```
void cyg_cond_signal
(
    cyg_cond_t *cond /* condition variable to signal */
)
```

该函数唤醒一个正在等待条件变量 cond 的线程。它只唤醒一个等待条件变量的线程。如果有多个线程正在等待该条件变量,将由调度器来决定首先唤醒哪一个线程。通常被唤醒的线程是优先级最高的一个线程。

```
void cyg_cond_broadcast
(
    cyg_cond_t *cond /* condition variable to signal */
)
```

该函数唤醒所有正在等待条件变量 cond 的线程。

```
cyg_bool_t cyg_cond_timed_wait
(
    cyg_cond_t *cond,          /* condition variable to wait for */
    cyg_tick_count_t abstime   /* absolute timeout */
)
```

该函数等待一个条件变量 cond,并具有超时返回功能。当等待时间超过 abstime 个时间滴答时,将返回 false。如果没有超时,则返回 true。

6.10 信号量

信号量(semaphore)是一个允许线程等待直到事件发生的同步原语。事件可以由另一个线程产生,也可以由处理硬件中断的 DSR 产生。每一个信号量都有一个整数计数器,用于对

事件发生且还没有得到处理的次数进行计数。如果计数器的值为 0,那么等待该信号量的线程将被阻塞,直至其他线程或 DSR 给信号量一个新的事件。如果计数器大于 0,那么等待该信号量的线程将消耗一个事件,也就是对计数器减 1,并立即返回。给信号量提交一个事件将唤醒当前正在等待该信号量的第一个线程,该线程将在等待信号量的操作内被恢复运行,并对计数器减 1。

信号量的另一个用途是对资源的管理。计数器的值与当前可用资源的数目相对应,需要对资源进行声称的线程将等待该信号量,释放该资源时再给信号量一个事件。实际上,条件变量更适合于这种操作。

6.10.1 信号量的使用

在使用信号量时,首先要调用内核 API 函数 cyg _ semaphore _ init 对信号量进行初始化。它具有两个参数,一个是信号量的数据结构的指针,另一个是计数器的初始值。必须注意在对信号量进行操作时,它不像其他内核 API 函数那样使用句柄,而是使用数据结构的指针。这种方式可以更易于将信号量的数据结构嵌入到其他的大的数据结构内。计数器的初始值可以是任何数字,0、正数或负数。计数器为 0 时通常表示还没有事件发生。

cyg _ semaphore _ wait 函数由等待某个事件的线程调用。如果当前计数器的值大于 0,也就是说已经有事件发生,则计数器将被减 1,并立即返回调用线程。如果不大于 0,那么该线程将被阻塞,直至 cyg _ semaphore _ post 函数被调用。cyg _ semaphore _ post 函数在事件发生的时候被调用,它使计数器加 1,并唤醒等待该信号量的第一个线程。被唤醒的线程通常在 cyg _ semaphore _ wait 函数内被启动继续运行并对计数器减 1,但也有其他的可能性。例如,假设调用 cyg _ semaphore _ post 的线程运行在高优先级,其他运行在中优先级的线程打算调用 cyg _ semaphore _ wait 获取下一个运行机会,一个低优先级的线程正在等待信号量。当高优先级的线程由于某种原因被停止调度后,中优先级的线程将开始运行,它对 cyg _ semaphore _ wait 的调用立即成功返回。低优先级的线程在稍后一段时间将开始运行,但发现计数器的值为 0 时,它将被阻塞直到另一个事件的发生。如果有多个线程阻塞于某个信号量,则可以使用配置选项 CYGIMP _ KERNEL _ SCHED _ SORTED _ QUEUES 来决定哪一个线程将被事件的发生唤醒。

cyg _ semaphore _ wait 函数返回一个布尔值。正常情况下,它将被阻塞直到它成功对计数器减 1 (如果需要的话将进行重试),最后成功返回。但这种等待操作可以被 cyg _ thread _ release 函数的调用所终止,此时 cyg _ semaphore _ wait 将返回 false。该函数的一个变体是 cyg _ semaphore _ timed _ wait,它具有超时功能,在事件确实发生或者超时时该函数将返回 (事件发生时返回 true,超时返回 false)。它被 cyg _ thread _ release 函数终止时也将返回 false。如果系统采取没有实时时钟的配置,那么该函数不可用。cyg _ semaphore _ trywait 是它的另一个变体函数,该函数将立即返回,不会被阻塞。

cyg _ semaphore _ peek 函数可以用于获取信号量计数器的当前值。除调试阶段外,很少使用该函数。计数器的值随时都有可能被线程或 DSR 改变。

6.10.2 信号量 API 函数

eCos 内核提供了一些对信号量进行操作的 API 函数,上一节已经介绍了使用这些函数对

信号量的操作方法。这些函数的定义在头文件<cyg/kernel/kapi.h>内,下面是它们的简单介绍。

```
void cyg_semaphore_init
(
    cyg_sem_t *sem,  /* semaphore to initialize */
    cyg_count32 val  /* initial semaphore count */
)
```

该函数初始化一个信号量 sem,其计数器的初始值为 val。

```
void cyg_semaphore_destroy
(
    cyg_sem_t *sem /* semaphore to invalidate */
)
```

该函数作废(失效)一个信号量。在作废一个信号量时应该注意此时没有任何线程正在等待或使用该信号量,否则可能引起系统死锁。

```
cyg_bool_t cyg_semaphore_wait
(
    cyg_sem_t *sem /* semaphore to wait on */
)
```

该函数等待一个信号量。如果信号量的计数器为 0,它将阻塞调用该函数的线程直到计数器被加 1。如果有多个线程都在等待同一个信号量,调度器将决定哪一个线程最先获取该信号量,通常是最高优先级的线程获取。如果线程被其他同步原语唤醒,该函数将返回 false,不会获取该信号量。如果成功获取信号量,该函数返回 true,否则返回 false。

```
cyg_bool_t cyg_semaphore_timed_wait
(
    cyg_sem_t *sem,          /* semaphore to wait on */
    cyg_tick_count_t abstime /* absolute timeout value */
)
```

该函数等待一个信号量 sem,如果等待时间超过 abstime(时间滴答),将超时返回。它是 cyg_semaphore_wait 函数带有超时功能的变体。超时退出时不会获取该信号量。成功获取信号量时返回 true,否则返回 false。

```
int cyg_semaphore_trywait
(
    cyg_sem_t *sem /* semaphore to get */
)
```

该函数尝试获取一个信号量,如果信号量的计数器不为 0,则获取该信号量并返回 true,计数器减 1。如果没有获取该信号量(计数器为 0 时),返回 false。它不会阻塞调用该函数的线程。

```
void cyg_semaphore_post
(
    cyg_sem_t *sem /* semaphore to increment count of */
)
```

该函数使指定信号量 sem 的计数器加 1。当事件发生时,调用该函数使信号量计数器加1。

```
void cyg_semaphore_peek
(
    cyg_sem_t *sem, /* semaphore to get count of */
    cyg_count32 *val /* pointer to receive count */
)
```

该函数获取信号量 sem 的当前计数器值,计数器的值由 val 带回。

6.11 信箱

信箱 (Mail Box) 是一个类似于信号量的同步原语,线程可以使用信箱来等待某个事件的发生。与信号量不同的是,信箱还可以在事件发生时被线程用来传递一些数据。这些被称为消息的数据通常是数据结构的指针,保存在信箱内。产生事件并提供这些事件的线程在其他线程准备好接收事件之前不会被阻塞。信箱只具有有限的容量,通常只有十个槽位。即使系统为事件的生产和消耗保持一种平衡,使事件被消耗的速度至少与事件被产生的速度一致,系列突发事件也可能将信箱填满,此时产生事件的线程将被阻塞,直到信箱空间再次可用。这种行为与信号量非常不同,信号量只需要对计数器进行维护,因此不会产生溢出。

6.11.1 信箱的使用

在使用信箱之前,必须使用 API 函数 cyg_mbox_create 产生一个信箱。每个信箱都有一个惟一的句柄,由第一个参数带回,使用其他信箱操作函数对信箱进行操作时,都将使用这个句柄。信箱的产生需要内核为其提供一个内存区,第二个参数将带回该内存区的指针。如果不再需要某个信箱,可以使用 cyg_mbox_delete 函数将其删除,它只简单地作废没有发送的消息。

用于等待信箱的函数是 cyg_mbox_get。如果信箱内有一个悬挂的被 cyg_mbox_put 函数发出的消息,则 cyg_mbox_get 将立即返回,并带回信箱内的消息。如果信箱内没有悬挂消息,它将被阻塞等待直到有消息可用。cyg_thread_release 函数可以打破线程的这种阻塞状态,这时 cyg_mbox_get 将返回一个空指针。信箱内消息的读取顺序按照它们进入信箱的顺序进行,也就是采用先进先出的策略,不支持带优先级的消息。

cyg_mbox_get 函数有两个变体。它的一个变体函数是 cyg_mbox_timed_get,该函数将等待直到有一个可用的消息或者超时发生。如果在超时时间内没有消息,它将返回空指针。另一个变体函数是 cyg_mbox_tryget,这是一个非阻塞函数,它将返回一个消息或者一个空指针。

如果要向信箱发送新的消息，可以调用 cyg _ mbox _ put 函数或它的变体函数。该函数有两个参数，一个是信箱的句柄，另一个是消息的指针。如果信箱中有空的槽位，则新消息将被立即放置到该槽位上，如果此时有一个正在等待的线程，该线程将被唤醒以便它能够收到该消息。如果信箱满，cyg _ mbox _ put 函数将被阻塞直到有可用的槽位。其变体函数 cyg _ mbox _ timed _ put 具有时间限制，如果在指定的时间范围内不能完成操作，它将超时退出。另一个变体函数 cyg _ mbox _ tryput 是一个非阻塞函数，如果没有可用的空槽位，它将立即返回 false，该函数不能发送消息时不会受到阻塞。

有四个 API 函数可以用于检查信箱的当前状态。在使用它们的时候，应该注意它们返回的结果可能已经被其他线程改变。cyg _ mbox _ peek 函数返回信箱内当前的消息数，cyg _ mbox _ peek _ item 函数提取信箱内的第一个消息，但消息仍然保留在信箱内。cyg _ mbox _ waiting _ to _ get 和 cyg _ mbox _ waiting _ to _ put 函数表明当前是否有线程被阻塞在对信箱的读或取操作上。

每个信箱的槽位数可以通过配置选项 CYGNUM _ KERNEL _ SYNCH _ MBOX _ QUEUE _ SIZE 进行控制，其默认值是 10。所有信箱大小是相同的。

6.11.2 信箱 API 函数

上一节介绍了如何使用 eCos 内核提供 API 函数对信箱进行操作。这些 API 函数的定义在头文件＜cyg/kernel/kapi.h＞内。下面是这些 API 函数的简单介绍。

```
void cyg _ mbox _ create
(
    cyg _ handle _ t * handle, /* returned handle to mbox object */
    cyg _ mbox * mbox          /* mbox object */
)
```

该函数创建一个信箱。信箱类似于其他操作系统中的消息队列，但 eCos 的所有信箱都具有相同的大小，由配置工具的配置选项指定。对新产生的信箱的操作可以使用该函数返回的句柄 handle 进行。

```
void cyg _ mbox _ delete
(
    cyg _ handle _ t mbox /* mbox to delete */
)
```

该函数删除一个信箱。调用该函数时要确认系统中没有任何线程正在等待或使用该信箱。

```
void * cyg _ mbox _ get
(
    cyg _ handle _ t mbox /* mbox to read data from */
)
```

该函数从信箱内读数据。如果信箱内没有数据，该函数将阻塞等待直到有可用的数据。该函数返回一个指向信箱内数据的指针。

```
void * cyg_mbox_timed_get
(
    cyg_handle_t mbox,          /* mbox to read */
    cyg_tick_count_t abstime    /* absolute timeout */
)
```

该函数在指定时间范围内读取信箱数据。如果信箱内没有可用数据,调用该函数的线程将被阻塞,直到有可用的数据或者直到超时。超时时间单位为时钟滴答。如果超时退出,将返回 NULL,如果函数读取数据成功则返回该数据的指针。

```
void * cyg_mbox_tryget
(
    cyg_handle_t mbox /* mailbox to read */
)
```

该函数从信箱内读取数据,如果信箱内没有数据,它不会被阻塞并立即返回 NULL。它成功读取信箱数据时返回该数据指针,如果信箱数据为空则返回 NULL。

```
void * cyg_mbox_peek_item
(
    cyg_handle_t mbox /* mailbox to read */
)
```

该函数从信箱内读取数据但数据仍然保留在信箱内。该函数将立即返回。如果信箱内有数据,它将返回该数据的指针,但不会从信箱内的消息队列中删除其指针。如果信箱没有数据,将返回 NULL。

```
cyg_bool_t cyg_mbox_put
(
    cyg_handle_t mbox, /* mbox to add item to */
    void * item        /* item to add to mbox */
)
```

该函数将消息送入一个信箱内。如果信箱已经满,它将阻塞直到消息可以放置到信箱内。如果线程在等待过程中被内核唤醒,该函数将返回错误。如果消息被成功送入信箱则返回 true,否则返回 false。

```
cyg_bool_t cyg_mbox_timed_put
(
    cyg_handle_t mbox,          /* mbox to add item to */
    void * item,                /* item to add to mbox */
    cyg_tick_count_t abstime    /* absolute timeout value */
)
```

该函数在指定时间发送消息到信箱内。如果信箱满,则将阻塞等待直到消息被送入信箱或超时返回。如果成功将消息送入信箱,则返回 true,否则返回 false。

```
cyg_bool_t cyg_mbox_tryput
(
    cyg_handle_t mbox, /* mbox to add item to */
    void *item         /* item to add to mbox */
)
```

该函数尝试将消息送入信箱，执行后将立即返回，它不会被阻塞。如果信箱已经满，它将立即返回 false。如果成功将消息送入信箱，则返回 true。

```
cyg_count32 cyg_mbox_peek
(
    cyg_handle_t mbox /* mbox to peek into */
)
```

该函数返回信箱内等待处理的消息数。

```
cyg_bool_t cyg_mbox_waiting_to_get
(
    cyg_handle_t mbox /* mbox to check */
)
```

该函数指明是否有线程正在等待信箱中的消息。如果有线程处于阻塞等待状态则返回 true，否则返回 false。

```
cyg_bool_t cyg_mbox_waiting_to_put
(
    cyg_handle_t mbox /* mbox to check */
)
```

该函数指明是否有线程正在等待从信箱中删除消息以便可以送入新消息。如果有线程处于阻塞等待状态，将返回 true，否则返回 false。

6.12 事件标志

事件标志 (Event flags) 也是一个同步原语，它允许线程等待一个或几个不同类型的事件发生。它还可以用于等待某些事件组合的发生。其实现原理相对来说比较直观，每一个事件标志是一个 32 位的整数，应用程序可以将它的每一位对应于一个特殊事件。例如，事件标志的 bit0 用于表示某个 I/O 事件的发生并且其数据可用，而 bit1 表示用户按下某个启动按钮，等等。产生事件的线程或 DSR 可能引起事件标志中的一位或几位被置 1，而等待这些标志位的线程可以被它们唤醒。

事件标志不像信号量那样对事件发生的次数进行计数，它不管发生了一次还是多次，与事件对应的标志位置都只被改变一次。信号量不能像事件标志一样被用来处理多个事件源。事件标志还通常被用来替代条件变量，但不能完全替代条件变量，它只支持相当于条件变量的广播，不支持发送信号。

6.12.1 事件标志的使用

在使用事件标志之前，必须调用内核 API 函数 cyg _ flag _ init 对其进行初始化。它使用了一个 cyg _ flag _ t 的数据结构，该数据结构就是被初始化的事件标志，它可以是一个大的数据结构中的一部分。在初始化的时候，事件标志的全部 32 位被清 0，表示还没有发生任何事件。如果不再需要某个事件标志，则可用使用 cyg _ flag _ destroy 函数将其废除，使得其 cyg _ flag _ t 结构所占用的内存可以被重新使用。

线程可以调用函数 cyg _ flag _ wait 来等待一个或多个事件。该函数有三个参数，第一个参数对指定事件标志进行标识；第二个参数是位的组合，表示对哪些事件感兴趣；第三个参数是下列值中的一个：

⑴ CYG _ FLAG _ WAITMODE _ AND。函数的调用者将被阻塞，直到所有指定事件的发生为止（此时事件标志中所有对应的位为 1）。等待成功时，事件标志中的这些位不会被清，将保持被设置的状态。

⑵ CYG _ FLAG _ WAITMODE _ OR。函数的调用者被阻塞，直到至少有一个指定的事件发生（事件标志中所有与指定事件相对应的位中只要其中一位被置为 1）。返回时不清事件标志。

⑶ CYG _ FLAG _ WAITMODE _ AND | CYG _ FLAG _ WAITMODE _ CLR。函数的调用者将被阻塞，直到所有指定事件的发生（事件标志中所有对应的位为 1），当成功返回时清整个事件标志。使用这种方式的操作时，不能用单个事件标志来保存相互独立的多个事件。每个独立的事件应该有自己的事件变量。

⑷ CYG _ FLAG _ WAITMODE _ OR | CYG _ FLAG _ WAITMODE _ CLR。函数的调用者被阻塞，直到至少有一个指定事件的发生（事件标志中所有与指定事件相对应的位中只要一位被置为 1），当成功返回时清整个事件标志。

线程在调用 cyg _ flag _ wait 时通常会被阻塞直到所需的条件得到满足，操作成功时它将返回此时事件标志的值，这个返回值所包含的事件可能不只是被请求的事件。如果使用 cyg _ thread _ release 函数打破线程的阻塞等待状态，cyg _ flag _ wait 将返回 0。

cyg _ flag _ timed _ wait 函数是 cyg _ flag _ wait 的一个变体，它增加了超时功能。等待操作必须在指定的时间内完成，否则将超时返回 0。cyg _ flag _ poll 函数是它的非阻塞变体函数，如果操作成功则立即返回，它的行为与 cyg _ flag _ wait 函数完全一致，如果操作不成功则立即返回 0。

线程或 DSR 可以调用 cyg _ flag _ setbits 函数对事件标志进行设置，该函数被用来对事件标志进行设置的参数与事件标志原来的值进行位或操作。如果此时消耗线程正在等待的条件得到满足，它可以唤醒这个等待线程。cyg _ flag _ maskbits 函数用于清事件标志中的一位或多位。当某个特殊条件不再被满足时（如用户不再按下某个按钮），产生事件的线程可以调用该函数。如果等待事件发生的消耗线程在其等待操作中不使用 CYG _ FLAG _ WAITMODE _ CLR，它可以使用该函数来说明它只消耗了所有活跃事件中的一部分事件。如果有多个没有使用 CYG _ FLAG _ WAITMODE _ CLR 的消耗线程正在进行等待操作，通常需要使用其他的同步机制（如互斥体）来防止多个线程消耗同一事件。

另外还有两个函数用于查询事件标志的当前状态。cyg _ flag _ peek 函数返回事件标志

的当前值，cyg _ flag _ waiting 函数可以用于发现是否有线程当前正处于等待该事件标志的阻塞状态。由于其他线程可能正在对事件标志进行操作，因此在使用这两个函数时应加以注意。

6.12.2 事件标志 API 函数

eCos 内核提供对事件标志进行操作的 API 函数定义于头文件＜cyg/kernel/kapi.h＞内，下面是这些 API 函数的简单介绍。

```
void cyg _ flag _ init
(
    cyg _ flag _ t * flag    /* flag to initialize */
)
```

该函数初始化一个事件标志。事件标志允许线程等待一个或一组条件，每个条件对应于事件标志中的一位，对每一位的具体定义由用户实现。

```
void cyg _ flag _ destroy
(
    cyg _ flag _ t * flag    /* flag to destroy (invalidate) */
)
```

该函数作废或失效一个事件标志。使用该函数时应该注意没有其他线程正在等待或使用该标志，否则将会造成线程死锁。

```
void cyg _ flag _ setbits
(
    cyg _ flag _ t * flag,          /* flag to modify */
    cyg _ flag _ value _ t value    /* bits to set */
)
```

该函数对事件标志位进行设置。参数 value 的值将与事件标志的原来值进行位或操作。该函数可能会唤醒正在等待该标志的线程。

```
void cyg _ flag _ maskbits
(
    cyg _ flag _ t * flag,          /* flag to modify */
    cyg _ flag _ value _ t value    /* bits to clear */
)
```

该函数清事件标志中的某些位。事件标志与 value 进行位与操作，得到事件标志的新值。如果 value 所有位都是 0，则事件标志的所有位都清 0。如果 value 的所有位为 1，则不会清事件标志中的任何一位。由于该操作只清事件标志，因此它不会唤醒任何线程。

```
cyg _ flag _ value _ t cyg _ flag _ wait
(
    cyg _ flag _ t * flag,          /* flag to wait on */
```

```
    cyg_flag_value_t pattern,   /* pattern to wait for */
    cyg_flag_mode_t mode        /* mode of waiting */
)
```

该函数使线程阻塞等待一组事件的发生。flag **表示等待哪个事件标志。**mode **参数用于如何解释** pattern**。上一节已经介绍了** mode **参数的使用,此处不再重复。如果线程被其他原因所唤醒,则返回** 0**。**

```
cyg_flag_value_t cyg_flag_timed_wait
(
    cyg_flag_t *flag,           /* flag to wait on */
    cyg_flag_value_t pattern,   /* pattern to wait for */
    cyg_flag_mode_t mode        /* mode of waiting */
    cyg_tick_count_t abstime    /* absolute timeout value */
)
```

该函数使线程在指定时间内等待一个或多个事件的发生,如果超时则返回 0**。除了超时功能外,它与** cyg_flag_wait **函数功能完全一样。参数** mode **对** pattern **的解释进行说明,其意义与** cyg_flag_wait **函数中的** mode **参数完全一样。**

```
cyg_flag_value_t cyg_flag_poll
(
    cyg_flag_t *flag,           /* flag to wait on */
    cyg_flag_value_t pattern,   /* pattern to wait for */
    cyg_flag_mode_t mode        /* mode of waiting */
)
```

该函数检查事件标志中的一组事件是否已经发生。参数 mode **对** pattern **的解释进行说明,其意义与** cyg_flag_wait **函数中的** mode **参数完全一样。如果条件满足(相应事件已经发生),则返回事件标志的值。如果线程被其他原因唤醒,则返回** 0**。**

```
cyg_flag_value_t cyg_flag_peek
(
    cyg_flag_t *flag   /* flag to peek at */
)
```

该函数返回事件标志的当前值。

```
cyg_bool_t cyg_flag_waiting
(
    cyg_flag_t *flag /* flag to check */
)
```

该函数报告当前是否有线程正处于等待事件标志的阻塞状态。如果有线程处于阻塞状态则返回 true**,否则返回** false**。**

6.13 Spinlock

Spinlock 是为 SMP 系统中的应用程序提供的一个同步原语。Spinlock 的运行级别要低于其他同步原语 (如互斥体),在大多数情况下将优先使用级别较高的同步原语。在某些环境下,需要使用 spinlock,特别是在对中断进行处理以及在线程需要共享硬件资源的情况下。在 SMP 系统中,内核自身的实现也需要使用 spinlock。

6.13.1 Spinlock 的使用

Spinlock 实质上仅仅是一个简单的标志,当程序试图声称一个 spinlock 时,它检查该标志是否已经被设置。如果没有被设置,则声称成功并设置该标志。Spinlock 的具体实现与硬件相关。例如,它可以使用 test-and-set 指令,即使在几个处理器同时对 spinlock 进行声称时都可以保证它所需要的行为。如果不能对一个 spinlock 进行声称,则当前线程将进入一个小的循环而处于自旋状态,它对标志进行重复检查直到该标志被清。这种行为与其他同步原语 (如互斥体等) 有着显著的区别,使用其他同步原语时所产生的竞争将引起线程被挂起。使用 spinlock 同步机制时,有一个假设条件,这就是对 spinlock 的拥有时间非常短。如果由于对 spinlock 的声称而引起当前线程被挂起,那么在中断处理程序中就不能使用 spinlock。

这种假设条件给使用 spinlock 的程序施加了一个强制性的约束条件。必须强调的是,对 spinlock 的拥有时间必须很短,一般为几十条指令。否则,其他处理器将被该 spinlock 阻塞一段较长时间,在这段时间内不能做任何事情。另一个值得注意的地方是拥有 spinlock 的线程不能被抢先,因为这样会使另一个处理器在整个时间片或更长的时间内处于自旋状态。为实现这一点,可采用的一种方法是在当前处理器上禁止中断,cyg _ spinlock _ spin _ intsave 函数提供了这一功能。

在单处理器系统中,不应该使用 spinlock。假设有一个高优先级的线程试图声称一个 spinlock,而该 spinlock 已经被一个低优先级的线程所拥有,这种情况下高优先级的线程将永远处于循环状态而低优先级的线程将不会有任何机会运行并释放 spinlock。即使两个线程具有相同优先级,试图声称 spinlock 的线程将进入自旋状态直到属于它的时间片被耗尽,造成 CPU 时间的浪费。如果一个中断处理程序试图声称一个被线程拥有的 spinlock,中断程序将永远处于循环之中。由此可见,spinlock 只适用于 SMP 系统,在 SMP 系统中 spinlock 的当前拥有者可以在不同的处理器上运行。

在使用一个 spinlock 之前,必须使用内核 API 函数 cyg _ spinlock _ init 对其进行初始化。该函数具有两个参数,一个是 cyg _ spinlock _ t 数据结构的指针,另一个是指定 spinlock 启动时的初始状态是锁定状态还是未锁定状态的标志。如果不再需要某个 spinlock,则可以使用 cyg _ spinlock _ destroy 函数将其作废。

有两个函数用于对 spinlock 进行声称,它们分别是 cyg _ spinlock _ spin 函数和 cyg _ spinlock _ spin _ intsave 函数。前者可以用在已经知道当前程序不会被抢先的情形下,例如中断处理程序或者中断被禁止的程序。后者除了声称 spinlock 外,还要进行禁止中断的操作,因此它在任何情形下的使用都是安全的。它的第二个参数用于返回原来的中断状态,在调用 cyg _ spinlock _ clear _ intsave 函数的时候将使用这种中断状态。

同样，有两个用于释放 spinlock 的函数，它们分别是 cyg _ spinlock _ clear _ int 函数和 cyg _ spinlock _ clear _ intsave 函数。前者通常用于释放被 cyg _ spinlock _ spin 函数声称的 spinlock，而后者则用于释放被 cyg _ spinlock _ spin _ intsave 函数声称的 spinlock。

还有另外两个函数可用于对 spinlock 的操作。cyg _ spinlock _ try 函数是一个非阻塞函数，它是 cyg _ spinlock _ spin 函数的一个变体。如果可能，该函数将对 spinlock 进行成功声称后立即返回 true，否则将立即返回 false。cyg _ spinlock _ test 函数可以被用来检查 spinlock 目前是否处于被锁定状态。该函数在使用时应该特别小心，因为 spinlock 的状态可能会随时改变。

Spinlock 的拥有时间必须非常短，对 spinlock 的声称不会引起线程被挂起。因此，spinlock 不会存在优先级倒置问题，也就不需要使用优先级置顶协议和优先级继承协议。

6.13.2 Spinlock 内核 API 函数

eCos 内核提供了上述对 spinlock 进行操作的 API 函数，在头文件＜cyg/kernel/kapi.h＞内对这些函数进行了定义。下面简单介绍这些 API 函数。

```
void cyg _ spinlock _ init (
    cyg _ spinlock _ t      lock,    /* spinlock to initialize */
    cyg _ bool _ t          locked   /* init locked or unlocked */
)
```

该函数对 spinlock 进行初始化。参数 locked 为 true 时，spinlock 的初始状态是被锁定的，为 false 时其初始状态为非锁定状态。

```
void cyg _ spinlock _ destroy (
    cyg _ spinlock _ t lock   /* spinlock to destroy (invalidate) */
)
```

该函数作废（失效）一个 spinlock。该函数在使用时应该特别小心，不得作废一个其他线程正在等待或使用的 spinlock。

```
void cyg _ spinlock _ spin (
    cyg _ spinlock _ t lock   /* spinlock to be claimed */
)
```

该函数用于对 spinlock 进行声称。只有在已经知道拥有该 spinlock 的程序不会被抢先的情况下才使用。

```
void cyg _ spinlock _ clear (
    cyg _ spinlock _ t  * lock /* spinlock to be released */
)
```

该函数释放一个 spinlock，通常用于释放被 cyg _ spinlock _ spin 函数声称的 spinlock。

```
cyg _ bool _ t cyg _ spinlock _ try (
    cyg _ spinlock _ t lock   /* spinlock to be claimed */
)
```

该函数声称一个 spinlock**,如果声称成功,立即返回** true**,如果不能声称,则立即返回** false**。该函数是一个非阻塞函数。**

```
cyg _ bool _ t cyg _ spinlock _ test (
    cyg _ spinlock _ t lock /* spinlock to be check */
)
```

该函数检查当前是否有线程正在阻塞等待 spinlock**。如果有线程阻塞,则返回** true**,否则返回** false**。**

```
void cyg _ spinlock _ spin _ intsave (
    cyg _ spinlock _ t lock,  /* spinlock to be claimed */
    cyg _ addrword _ t istate /* interrupt state       */
)
```

该函数声称 spinlock**,并禁止中断。中断状态被存放在** istate **内。该函数是安全的。**

```
void cyg _ spinlock _ clear _ intsave (
    cyg _ spinlock _ t * lock, /* spinlock to be released */
    cyg _ addrword _ t istate  /* interrupt state         */
)
```

该函数释放一个 spinlock**。通常用于释放被** cyg _ spinlock _ spin _ intsave **函数声称的** spinlock**。**

第 7 章　标准 C 与数学库

eCos 提供了一个标准 C 库，它与 ISO 9899：1990 标准 C 库说明兼容，这一说明实质上与众所周知的 ANSI C3.159-1989 说明（C-89）是相同的。eCos 的这种兼容性来自三个方面。首先，eCos 提供了一个 C 库（C Library Package），实现了除数学函数之外的所有 ISO 标准定义的函数。其次，eCos 提供了一个数学库（Math Library），它实现了 ISO C 库内的数学函数。第三个方面是 eCos 提供了一个标准 C 库环境，在该环境下可以运行使用标准 C 库编写的应用程序。这种环境是由 C 库启动程序建立起来的，它提供一个主入口函数 main（），一个退出函数 exit（）以及一个能读取环境参数的函数 getenv（）。

本章的重点主要是介绍 C 库及数学库中与 eCos 相关的部分，其中大部分与 eCos 的可配置性相关。有关标准 C 的介绍读者可以参阅其他 C 语言参考书。

7.1　标准 C 与数学库的配置

eCos 提供的标准 C 库和数学库都是可选的组件，可以根据实际需要选择安装相应的包。如果选择了标准 C 库，应用程序可以使用入口函数 main（），否则只能使用 cyg _ user _ start（）作为应用程序的入口函数。

使用图形配置工具可以对标准 C 库和数学库的具体配置选项进行配置，如图 7-1 所示。

图 7-1　标准 C 与数学库的配置

对标准 C 库可以进行下述几个方面的配置：

① ISO C 库国际通用函数的配置。

② ISO C 库长跳转函数的配置。

③ ISO C 库信号函数的配置。

④ ISO C 环境、启动函数 (main) 和退出函数 (exit) 的配置。
⑤ ISO C 库标准输入/输出函数的配置。
⑥ ISO C 库普通应用函数的配置。
⑦ ISO C 库字符串函数的配置。
⑧ ISO C 库日期与时间函数的配置。

对数学库的配置主要包括：

① 数学库兼容方式的选择，包括 POSIX、IEEE、X/OPEN、SVID 等。
② 线程安全性选择。
③ 对调试支持的配置。
④ 数学库编译选项。

7.2 非 ISO 标准函数

eCos 的 C 库包含了一些非 ISO 标准的一些函数。eCos 支持 POSIX 标准，下面的这些非 ISO 标准的 C 函数来自 POSIX 标准：

① extern char * * environ 变量，用于建立一种环境，使用 getenv () 函数可读取这种环境
② _ exit () 立即退出当前程序。
③ strtok _ r () 可重入的 strtok ()，在字符串中查找。
④ rand _ r () 可重入的 rand ()，产生一个随机数。
⑤ asctime _ r () 可重入的 asctime ()，将保存在数据结构内的时间值转换为字符串。
⑥ ctime _ r () 可重入的 ctime ()，将时间转换为字符串。
⑦ localtime _ r () 可重入的 localtime ()，时间函数。
⑧ gmtime _ r () 可重入的 gmtime ()，将时间值转换为表示时间的数据结构。

除了上述这些出自 POSIX 标准的非 ISO 函数外，eCos 的标准 C 库还专门提供了一些函数用于调整日期和时间的设置。这些函数如下：

```
void cyg _ libc _ time _ setdst (
        cyg _ libc _ time _ dst state
) ;
```

该函数对夏令时 (DST) 的状态进行设置。夏令时的状态具有下列值：

```
CYG _ LIBC _ TIME _ DSTNA    unknown
CYG _ LIBC _ TIME _ DSTOFF   off
CYG _ LIBC _ TIME _ DSTON    on

void cyg _ libc _ time _ setzoneoffsets (
        time _ t stdoffset,
        time _ t dstoffset
) ;
```

当夏令时使能或关闭时，该函数对与 UTC (协调世界时) 时间的偏差进行设置。这种偏差使用 time _ t 类型表示，单位是秒。

```
cyg_libc_time_dst cyg_libc_time_getzoneoffsets(
        time_t *stdoffset,
        time_t *dstoffset
);
```

该函数读取夏令时 DST 的当前设置以及标准时 STD 和夏令时 DST 的偏差。这些偏差均使用 time_t 类型表示,单位为秒。

```
cyg_bool cyg_libc_time_settime(
        time_t utctime
);
```

该函数对当前系统时间进行设置。时间使用 UTC 格式,用 time_t 类型表示。如果设置发生错误,将返回非 0 值。

7.3 数学库兼容方式

eCos 数学库具有在几种不同兼容方式下运行的能力。它支持四种兼容方式:

① ANSI/POSIX 1003.1。

② IEEE-754。

③ XPG3 (X/Open Portability Guide Issue 3)。

④ System V Interface Definition Edition 3。

这些兼容方式有着各自对错误进行处理的方法。

在 IEEE 方式下,不调用 matherr() 函数,在 stderr 输出流中不会输出警告信息,也不会设置错误代码 errno。

在 ANSI/POSIX 方式下,将有错误代码 error 产生,但不会调用 matherr() 函数,在 stderr 输出流中不会有警告信息输出。

在 X/Open 方式下,将有错误代码 error 产生,也将调用 matherr() 函数,但在 stderr 输出流中不会有警告信息输出。

在 SVID 方式下,函数溢出时将返回 HUGE 值 (在头文件 math.h 中有定义),这是单精度浮点数的最大值 (HUGE_VAL 表示无穷大)。这种方式下将有错误代码产生,matherr() 函数也会被调用。如果 matherr() 返回 0,在 stderr 输出流中将有针对某些错误的警告信息输出。

在 X/Open 方式和 SVID 方式下,可以使用 matherr() 函数,该函数对数学错误进行处理,其格式如下:

```
int matherr(struct exception *e)
```

其参数使用了结构 exception,定义如下:

```
struct exception {
        int type;
        char *name;
        double arg1, arg2, retval;
};
```

其中：

type 是下列例外之一：

DOMAIN

SING

OVERFLOW

UNDERFLOW

TLOSS

PLOSS

name 是包含出错函数名的字符串；

arg1、arg2 是传递给出错函数的参数；

retval 是出错函数返回的默认值，可以被 matherr () 函数修改。

如果 matherr () 函数返回 0，或者用户没有自己补充 matherr () 函数，那么在 SVID 方式下通常会出现表 7-1 所述行为。

表 7-1　数学函数例外行为

例外类型	行　为
DOMAIN	返回 0.0，errno = EDOM，stderr 有信息输出
SING	返回带有符号的 HUGE，errno = EDOM，stderr 有信息输出
OVERFLOW	返回带有符号的 HUGE，errno = ERANGE
UNDERFLOW	返回 0.0，errno = ERANGE
TLOSS	返回 0.0，errno = ERANGE，stderr 有信息输出
PLOSS	当前版本不返回这种类型的例外

X/Open 方式与此相似，不同之处是 stderr 不会输出信息，它使用 HUGE _ VAL 替代 HUGE。

如果对配置选项进行适当的设置，数学库在下述条件下具有线程安全性：

① 对 C 库的 error 变量进行设置具有线程安全性。

② 使用 C 库函数 fput () 将错误消息输出到 stderr 输出流具有线程安全性。

③ 用户补充的 matherr () 函数具有线程安全性。

另外，如果数学库总处于 IEEE 兼容方式，除 gamma * () 函数和 lgamma * () 函数之外的函数都具有可重入性。

7.4　一些实现细节

在 eCos 的 C 库中，有一些值得注意和了解的地方。

① 在对 eCos 进行配置时，有可能包含了标准 C 库而没有包含内核，这样可以使用更少的内存。如果没有包含内核，就不能使用内存分配、线程安全性以及某些 stdio 函数 (如输入等函数)。其他 C 库函数不会受到影响。

② clock()函数返回的数据类型为 clock_t,它是一个 64 位的整数。只有在内核被配置为具有实时时钟支持的情况下,clock()返回的值才是正确的。内核的这种配置是由 kernel.h 中的配置选项 CYGVAR_KERNEL_COUNTERS_CLOCK 来实现的。

③ FILE 类型不是一个结构体,而是 CYG_ADDRESS。

④ GNU C 编译器将使用编译器本身的一些内置实现来替代某些 C 库函数。可以使用-fno-builtin 编译选项关闭这种特性。受影响的函数有:abs()、cos()、fabs()、labs()、memcmp()、memcpy()、sin()、sqrt()、strcmp()、strcpy()以及 strlen()。

⑤ GCC 编译时使用-fno-builtin 编译选项将会禁止编译器使用其内置实现,如果要使程序具有更快的执行速度,应该避免使用这种选项,让编译器使用其内置实现。

⑥ memory()函数和 memset()函数位于基础结构包内,而不是在 C 库包内。这是由于编译器需要使用这两个函数,如果配置时没有包含 C 库,内核就必须实现这两个函数。

⑦ 错误码(如 EDOM 和 ERANGE)以及 strerror()函数是在错误包内实现的。错误包与 C 库和数学库是分开的,这样即使在 eCos 配置中没有包含 C 库,eCos 的其余部分仍然可以使用错误处理函数。

⑧ 当调用 free()函数时,heap 内存通常会被合并。如果没有对配置选项 CYGSEM_KERNEL_MEMORY_COALESCE 进行设置,内存就不会被合并,这种情况可能会引起程序运行错误。

⑨ 在正常工作的程序环境中使用 raise()发出信号时,要保证信号能正常工作。在中断服务程序 ISR 和滞后中断服务程序 DSR 环境中发出的信号不一定能正常工作。另外,在对配置选项 CYGSEM_LIBC_SIGNALS_HWEXCEPTIONS 进行了设置的情况下,不能保证信号的正常工作。这种情况下,如果要处理 signal()函数发出的信号,就必须以例外的形式来捕获。虽然如此,信号的实现仍然是与 ISO C 兼容的,ISO C 也不能提供这种保证。

⑩ 除了配置选项 CYGPKG_LIBC_ENVIRONMENT 被关闭的情况外,都实现了 getenv()函数。但是没有提供任何 shell 函数或 putenv()对环境进行动态设置。可以使用一个全局变量 environ 对环境进行设置,其声称如下:

```
extern char * * environ;//Standard environment definition
```

使用 CYGDAT_LIBC_DEFAULT_ENVIRONMENT 选项可以在系统启动时对环境进行静态初始化。如果这样的话,环境变量数组的最后一项在初始化后应该为 NULL。

下面是一个使用环境的 eCos 小程序:

```
----------example for environment----------
#include <stdio.h>
#include <stdlib.h> //Main header for stdlib functions

extern char * * environ;//Standard environment definition

int
main(int argc, char * argv[])
{
    char * str;
```

```
    char * env[] = {"PATH=/usr/local/bin:/usr/bin",
                    "HOME=/home/fred",
                    "TEST=1234=5678",
                    "home=hatstand",
                     NULL
                   };

    printf("Display the current PATH environment variable\n");
    environ = (char **)&env;
    str = getenv("PATH");
    if (str==NULL) {
            printf("The current PATH is unset\n");
    } else {
            printf("The current PATH is \"%s\"\n", str);
    }
    return 0;
    }
```

7.5 线程安全性

当多线程在同一时间调用同一函数时，就存在线程安全性问题。ISO C 库提供了一个控制其线程安全性（thread safety）的配置选项。eCos 对这个选项进行设置可以保证这种安全性。

在多线程环境下使用下面的函数时必须加以小心，这些函数的使用应该具有正确的线程安全性配置。

① mblen()。
② mbtowc()。
③ wctomb()。
④ printf()（以及除 sprintf()和 sscanf()之外的所有标准 I/O 函数）。
⑤ strtok()。
⑥ rand()和 srand()。
⑦ signal()和 raise()。
⑧ asctime()、ctime()、gmtime()和 localtime()。
⑨ 变量 errno。
⑩ 变量 environ。
⑪ 日期和时间的设置。

在某些情况下，eCos 为使开发更容易，它对某些函数提供了另外一种可替换的可重入函数，如 rand_r()、strtok_r()、asctime_r()、ctime_r()、gmtime_r()以及 localtime_r()等，从而保证其线程安全性。

在另外一些情况下，eCos 提供了一些配置选项来实现线程安全性，这些选项对函数或者

函数的共享数据进行锁定控制。

还有其他的一些情况，如日期和时间的设置，由于这些函数相对简单，不值得对其进行安全性处理（例如很少对日期和时间进行设置），因此对这些函数没有提供可重入的具有线程安全性的替换函数，也没有提供相应的配置选项。

7.6 C 库启动函数

eCos 的 C 库包含了下面这样一个函数：

```
void cyg_iso_c_start(void)
```

该函数着手建立一种环境，ISO C 程序可以在该环境下以最兼容的方式运行。该函数所做的工作是产生一个将要调用 main() 主函数的线程，main() 函数通常被认为是程序的入口点。如果使用配置选项 CYGDAT_LIBC_ARGUMENTS，它还可以为 main() 函数提供参数，当程序从 main() 返回或者调用 exit() 函数返回时，处于挂起状态的 stdio 输出文件将输出，并且将调用通过 atexit() 函数进行登记过的所有函数。

线程在 eCos 调度器启动的时候开始工作。如果不使用 eCos 内核包，也就是说没有调度器时，cyg_iso_c_start() 函数将直接调用 main() 函数，在 main() 函数返回之前，该函数不会返回。

main() 函数的定义如下。如果在 C++ 文件中使用了这种定义就必须使用“C”链接。

```
extern int main(int argc, char *argv[])
```

如果需要的话，可以对被 cyg_iso_c_start() 函数启动的线程进行直接操作。例如，直接对线程进行挂起操作。内核中的 C API 函数在进行这种操作的时候需要使用一个句柄 handle，这种句柄可以在程序代码中通过包含下面的语句来获取：

```
extern cyg_handle_t cyg_libc_main_thread;
```

这样，可以通过下面的语句挂起一个线程：

```
cyg_thread_suspend(cyg_libc_main_thread);
```

如果用户调用了 cyg_iso_c_start() 函数，但没有提供自己的 main() 函数，那么系统将提供一个 main() 函数，这一 main() 函数将简单地直接返回。

在 eCos 的默认配置中，cyg_iso_c_start() 函数自动被 cyg_package_start() 函数调用。这样，在最简单的情况下，最简单的用户程序如下：

```
int main(int argc, char *argv[])
{
    printf("Hello eCos\n");
}
```

如果不考虑使用 cyg_package_start() 函数或 cyg_start() 函数，或者关闭了基础结构包的配置选项 CYGSEM_START_ISO_C_COMPATIBILITY，那么必须保证用户程序自己调用函数 cyg_iso_c_main()，使用户程序可以自动地在程序入口点 main() 处开始执行。

第 8 章　设备驱动程序与 PCI 库

eCos 源码中的 I/O 包包含了多种设备类型的驱动程序，这些设备驱动程序以组件的形式出现在 I/O 包内。设备驱动程序组件也可以像其他所有组件一样，系统可以根据实际需要对它们进行配置。开发人员也可以设计自己的设备驱动程序，并将其加入到系统中来。

驱动程序模块支持分层结构，一个设备可以是另一个设备的上层设备。处于上层的设备可以灵活地增加一些底层设备没有提供的一些功能和特性。例如，TTY 设备建立在简单串行设备的上层，它提供的行缓冲和行编辑功能是简单串行设备所不具备的。

eCos 还提供了一个可选的 PCI 库。当目标平台需要对 PCI 总线及其设备进行操作时，可以使用 PCI 库提供的支持。利用 PCI 库可以进行 PCI 总线的初始化、查找 PCI 设备、PCI 配置空间的读写、中断处理等操作。

本章主要讲述 eCos 设备驱动程序与系统其他部件的接口（API 函数），详细介绍设备驱动程序的结构的编程方法，并举例说明设备驱动程序的设计方法。最后对 PCI 库的内容和使用方法进行介绍。

8.1　设备驱动程序用户 API

eCos 提供了一些驱动程序用户 API 函数，应用程序使用这些函数可以对设备进行操作。这些操作包括对设备进行配置、获取配置信息、对设备进行数据的读写等。

设备驱动程序必须采用设备句柄（handle）对其进行访问。系统中的每一个设备都有一个惟一的名字，cyg _ io _ lookup () 函数用于将该名字映射到设备句柄。该函数还可以在启动初期用于对设备进行初始化操作。设备标准名字采用诸如“/dev/console”、“/dev/serial0”等这样的格式，其前缀“/dev/”表示这是一个设备的名字。设备驱动程序所提供的基本功能包括对设备进行数据读写操作，此外它还提供了其他一些函数对具体设备以及驱动程序的状态进行控制和处理。

驱动程序用户 API 函数的参数均采用指针的形式，可以有效地实现与驱动程序之间的信息传递。最明显的例子是它的读写函数中对数据长度参数 length 的传递，在调用读函数和写函数时，length 包含了将要传送的数据长度，函数返回时 length 包含了实际传送的数据长度。

除了 cyg _ io _ lookup () 函数外，所有的驱动程序用户 API 函数都要使用设备句柄 handle。所有函数都将返回一个具有 Cyg _ ErrNo 类型的值。如果有错误发生，返回的值将是负值，并具体指明错误代码。根据错误代码可以在头文件 cyg/error/code. h 中查看相应的错误。如果函数执行正常，返回值为 ENOERR，表示没有错误。

下面是设备驱动程序的一些最基本的用户 API 函数，应用程序可以直接调用这些函数对设备进行控制和访问。

8.1.1 设备的查找

```
Cyg_ErrNo cyg_io_lookup(
    const char *name,
    cyg_io_handle_t *handle
)
```

该函数用于查找某个设备，将该设备名字映射到适当的设备句柄并返回。如果参数 *name 所指定名字的设备在系统中不存在，则返回错误码 -ENOENT。如果该设备存在，则通过句柄指针参数 *handle 返回相应的设备句柄 handle。

8.1.2 向设备传送数据

```
Cyg_ErrNo cyg_io_write(
    cyg_io_handle_t handle,
    const void *buf,
    cyg_uint32 *len
)
```

该函数向指定的设备传送数据。数据传送的大小由参数 *len 指定，函数返回时 *len 为实际被传送的数据大小。

8.1.3 读取设备数据

```
Cyg_ErrNo cyg_io_read(
    cyg_io_handle_t handle,
    void *buf,
    cyg_uint32 *len
)
```

该函数从指定的设备读取数据。读取数据的大小由参数 *len 指定，函数返回时 *len 为实际读取的数据大小。

8.1.4 读取设备配置信息

```
Cyg_ErrNo cyg_io_get_config(
    cyg_io_handle_t handle,
    cyg_uint32 key,
    void *buf,
    cyg_uint32 *len
)
```

该函数读取指定设备的实时配置信息。参数 key 指定所需读取的信息类型，返回的信息存放在参数 *buf 指定的位置。参数 *len 指定所需读取配置信息的大小，其值必须至少与 key 所选择的信息类型的大小相同。函数返回时 *len 的值为实际读取的信息大小。每一个驱动程序都有不同的 key 值，在头文件 cyg/io/config_keys.h 中列举了所有的 key 值，不同的

设备类型具有不同的 key 值定义。

8.1.5 对设备的配置

```
Cyg_ErrNo cyg_io_set_config(
    cyg_io_handle_t handle,
    cyg_uint32 key,
    const void *buf,
    cyg_uint32 *len
)
```

该函数用于处理和改变设备的运行时配置。配置信息的类型由参数 key 指定,所需的配置数据从参数 *buf 指定的位置获取。参数 *len 包含了所提供的配置数据的大小,其值应该与所选择的 key 所对应的配置信息类型相匹配。

8.2 驱动程序与内核及 HAL 的接口

这一节介绍设备驱动程序与内核和硬件抽象层 HAL 之间的 API 接口。这些 API 接口函数主要对中断以及中断处理程序的 ISR、DSR 和线程的同步进行控制和管理。在没有内核 kernel 的配置中这些 API 也必须存在,此时由 HAL 直接提供这些 API 函数。

8.2.1 eCos 中断模块

eCos 设备驱动程序的中断模块分为三个层次,分别是中断服务程序 ISR、中断滞后服务程序 DSR 和中断线程。中断服务程序 ISR 在响应中断时立即调用,中断滞后服务程序 DSR 由 ISR 发出调用请求后调用,而中断线程为驱动程序的客户程序。图 8-1 说明了中断模块的组成以及它们的执行过程。

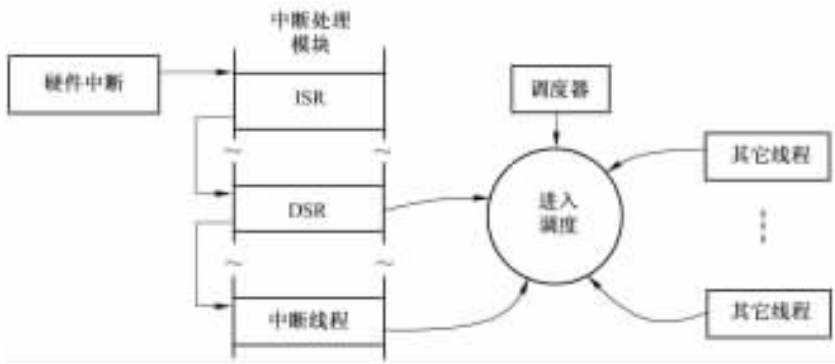


图 8-1 中断模块的组成与处理过程

硬件中断在最短的时间内交付给 ISR 处理。硬件抽象层 HAL 对硬件中断源进行译码并调用对应的中断服务程序 ISR。ISR 可以对硬件进行操作,但它能够使用的驱动程序 API 函数具有一定的限制。当 ISR 返回时,它可以请求与其相对应的中断滞后服务程序 DSR 进入调

度运行。

中断滞后服务程序 DSR 可以在不会防碍调度器正常工作的时候安全运行。在大多数情况下,DSR 将在 ISR 执行完成后立即运行。但如果当前线程正处于调度运行中,那么 DSR 将被推迟到当前线程完成后才开始运行。与 ISR 相比,中断滞后服务程序 DSR 能够使用更多的驱动程序 API 函数,特别是它可以调用 cyg _ drv _ cond _ signal () 函数来唤醒正在等待的线程。

中断线程可以使用所有的驱动程序 API 函数,它可以对互斥体和条件变量进行等待。

对于一个具有中断的设备驱动程序,它首先必须提供 ISR 和 DSR,然后再调用 cyg _ drv _ interrept _ create () 函数产生一个中断对象 (interrupt object)。驱动程序将使用该函数返回的句柄 handle 调用 cyg _ drv _ interrupt () 函数,完成中断与硬件向量的连接。

8.2.2 同步

eCos 对中断处理提供了三个层次的同步支持:

1. 中断服务程序 ISR 的同步

在临界时期通常使用禁止中断的形式来防止 ISR 的运行。在 SMP 环境中,还需要使用 spinlock 来实现运行在其他处理器的 ISR、DSR 和中断线程的同步。cyg _ drv _ isr _ lock () 函数和 cyg _ drv _ isr _ unlock 函数用于实现这种同步操作。这种同步机制的使用应尽量少,并且只能在短时间内使用。如果要实现更好的同步,还必须使用专用 spinlock。

2. 中断滞后服务程序 DSR 的同步

这种同步的实现机制是在内核中锁定 (lock) 调度器,以防止 DSR 在临界时期的运行。在无内核的配置中,这种同步必须使用非内核的形式加以实现。函数 cyg _ drv _ dsr _ lock () 和 cyg _ drv _ dsr _ unlock () 用于实现 DSR 的同步。与 ISR 的同步一样,也应该尽量少地使用这种同步机制。只有 DSR 和中断线程可以使用这种同步机制,而 ISR 不允许使用。

3. 中断线程的同步

中断线程的同步是通过使用互斥体和条件变量来实现的。只有中断线程可以锁定互斥体和等待条件变量,DSR 可以对条件变量发信号。

中断处理的三个不同层次 (ISR、DSR 和中断线程) 对数据的访问应该受到保护,以避免对这些数据的并发访问。中断服务程序 ISR 所访问的数据必须使用 ISR lock 或 spinlock 来加以保护,中断滞后服务程序 DSR 和中断线程共享的数据应该使用 DSR lock 来加以保护。只有中断线程才可以访问的数据必须使用互斥体来加以保护。

中断服务程序 ISR 和中断滞后服务程序 DSR 中的一些 API 调用有可能会引起当前线程进入重新调度的危险状态。为避免这种状态的发生,对这些 API 函数都指定了相应的同步级别。这些同步级别分别为:

① 中断线程 (thread) 级。具有该同步级别的函数只能在中断线程内被调用。通常是中断客户端程序对设备驱动程序的调用。在无内核的配置中,进行这种调用的程序应该在默认的非中断级别运行。

② DSR 级。具有 DSR 同步级别的函数可以被中断滞后服务程序 DSR 和中断线程调用。

③ ISR 级。具有 ISR 同步级别的函数可以被 ISR、DSR 和中断线程调用。

表 8-1 列举了每一个 API 函数的可调用级别。

表 8-1　驱动程序与内核接口 API 函数调用级别

函　　数	可调用级别		
	ISR	DSR	Thread
cyg _ drv _ isr _ lock	X	X	X
cyg _ drv _ isr _ unlock	X	X	X
cyg _ drv _ spinlock _ init			X
cyg _ drv _ spinlock _ destroy			X
cyg _ drv _ spinlock _ spin	X	X	X
cyg _ drv _ spinlock _ clear	X	X	X
cyg _ drv _ spinlock _ try	X	X	X
cyg _ drv _ spinlock _ test	X	X	X
cyg _ drv _ spinlock _ spin _ intsave	X	X	X
cyg _ drv _ spinlock _ clear _ intsave	X	X	X
cyg _ drv _ dsr _ lock		X	X
cyg _ drv _ dsr _ unlock		X	X
cyg _ drv _ mutex _ init			X
cyg _ drv _ mutex _ destroy			X
cyg _ drv _ mutex _ lock			X
cyg _ drv _ mutex _ trylock			X
cyg _ drv _ mutex _ unlock			X
cyg _ drv _ mutex _ release			X
cyg _ drv _ cond _ init			X
cyg _ drv _ cond _ destroy			X
cyg _ drv _ cond _ wait			X
cyg _ drv _ cond _ signal		X	X
cyg _ drv _ cond _ broadcast		X	X
cyg _ drv _ interrupt _ create			X
cyg _ drv _ interrupt _ delete			X
cyg _ drv _ interrupt _ attach	X	X	X
cyg _ drv _ interrupt _ detach	X	X	X
cyg _ drv _ interrupt _ mask	X	X	X
cyg _ drv _ interrupt _ unmask	X	X	X
cyg _ drv _ interrupt _ acknowledge	X	X	X
cyg _ drv _ interrupt _ configure	X	X	X
cyg _ drv _ interrupt _ level	X	X	X
cyg _ drv _ interrupt _ set _ cpu	X	X	X
cyg _ drv _ interrupt _ get _ cpu	X	X	X

8.2.3 SMP 支持

某些目标系统可能包含多个 CPU，eCos 必须为其提供 SMP 支持。设备驱动程序有许多方法可以用来支持 SMP 目标系统。由于中断处理的三个模块 ISR、DSR 和中断线程有可能运行在不同的 CPU 上，因此，对于支持 SMP 系统的驱动程序必须正确使用驱动程序 API 函数。

为满足中断线程和中断滞后服务程序 DSR 之间的同步要求，中断线程应该使用 cyg _ drv _ dsr _ lock () 函数和 cyg _ drv _ dsr _ unlock () 函数对共享数据的访问进行保护。而对于中断服务程序 ISR 和中断滞后服务程序 DSR 或中断线程之间的同步要求，应该使用 cyg _ drv _ isr _ lock () 函数和 cyg _ drv _ isr _ unlock () 函数对所有关键数据的访问进行保护。由于中断线程和中断滞后服务程序 DSR 有可能运行在不同的 CPU 上，而中断的使能和禁止只对当前 CPU 有效，因此仅仅只对中断进行禁止或屏蔽是不够的。

对于 SMP 系统来说，ISR lock 不仅能禁止当前 CPU 的中断，而且还得到一个 spinlock 对数据进行保护，使这些数据不会受到其他 CPU 的并发访问。由于 ISR 的运行不会请求调度器锁，因此这种保护是必要的，这样 ISR 可以与设备驱动程序的其他组件并行运行。

驱动程序 API 提供的 ISR lock 是一种共享的 spinlock，所有的驱动程序都可以使用这个共享 spinlock。驱动程序如果需要更精细的锁定，它可以使用私有 spinlock，利用 cyg _ drv _ spinlock _ * () 之类的函数对这些 spinlock 进行访问。

8.2.4 驱动程序模式

有许多构建 eCos 设备驱动程序的方法，具体使用哪种驱动程序模式要根据设备的具体属性以及所要求实现的功能来决定。具体来说，有三种可以采用的驱动程序模式。

第一种模式是在中断服务程序 ISR 内完成所有的设备处理工作。当调用 ISR 时，ISR 直接对设备硬件进行操作，在内存里直接对被传送的数据进行访问。ISR 应该调用 cyg _ drv _ interrupt _ acknowledge () 函数进行相应的处理。ISR 执行完成后，它可能会请求调用中断滞后服务程序 DSR。DSR 除了调用 cyg _ drv _ cond _ signal () 函数唤醒一个中断线程外，不会进行其他操作。中断线程在对共享内存进行操作时必须调用 cyg _ drv _ isr _ lock () 函数或 cyg _ drv _ interrupt _ mask () 函数对 ISR 的运行进行保护。

第二种模式是将设备处理工作推迟到中断滞后服务程序 DSR 内进行。中断服务程序 ISR 只是简单地通过对设备进行编程或调用 cyg _ drv _ interrupt _ mask () 函数来防止新中断的产生。ISR 然后通过调用 cyg _ drv _ interrupt _ acknowledge () 函数允许响应其他中断，它最后将调用 DSR 进行进一步的处理。在 DSR 运行的时候，它完成了大部分的设备处理工作，并有可能对某个条件变量产生一个信号来唤醒中断线程。DSR 在结束的时候将调用 cyg _ drv _ interrupt _ unmask () 函数，使能新的中断。中断线程在对共享内存进行操作时，使用 cyg _ drv _ dsr _ lock () 函数保护 DSR 的运行。eCos 的串口驱动程序采用的就是这种方法。

第三种模式是将设备处理工作推迟到中断线程进行。中断服务程序 ISR 在这种模式中的行为与第二种模式相同，在它请求 DSR 运行前简单地阻塞中断和应答中断。中断滞后服务程序 DSR 只调用 cyg _ drv _ cond _ signal () 函数唤醒中断线程。被唤醒的中断线程完成全部的设备处理工作，它具有对所有内核功能进行完全访问的能力。中断线程在结束时将调用 cyg _ drv _ interrupt _ unmask () 函数，重新使能设备中断。eCos 的网络驱动程序使用的就是这

种模式。

对于需要立即进行处理并且与中断线程交互相对较少的设备，最好采用第一种模式。第二种模式对设备的处理有一定的延迟，它受同步机制的干预较少。最后一种模式允许设备的处理工作可以与其他线程一起进行调度，设备处理具有更多的灵活性。

8.2.5 驱动程序与内核及 HAL 的接口 API 函数

这一节主要介绍设备驱动程序与内核的接口 API 函数。在无内核的配置中，硬件抽象层应该直接支持这些函数。这些 API 函数的定义在头文件 cyg/hal/drv_api.h 中可以找到。

cyg_drv_isr_lock

函数原型：

```
void cyg_drv_isr_lock(void)
```

参数：

无。

返回结果：

无。

描述：

该函数禁止响应中断，防止所有的中断服务程序 ISR 的运行。该函数对一个计数器进行维护，记录其被调用的次数。

cyg_drv_isr_unlock

函数原型：

```
void cyg_drv_isr_unlock(void)
```

参数：

无。

返回结果：

无。

描述：

该函数重新使能响应中断，允许中断服务程序 ISR 运行。该函数对 cyg_drv_isr_lock() 所维护的计数器进行减一操作，只有在计数器为 0 时才允许中断的发生。

cyg_drv_spinlock_init

函数原型：

```
void cyg_drv_spinlock_init(cyg_spinlock_t *lock,
                           cyg_bool_t locked
                          )
```

参数：

lock — 被初始化的 spinlock 的指针。

locked — lock 的初始状态。

返回结果：

无。

描述：

初始化一个 spinlock。**参数** locked **说明如何对** spinlock **进行初始化：**locked **为** TRUE **时初始状态为** locked，locked **为** FALSE **时初始状态为** unlocked。

cyg _ drv _ spinlock _ destroy

函数原型：

```
void cyg_drv_spinlock_destroy(cyg_spinlock_t *lock)
```

参数：

lock — **指向被删除的** spinlock **指针。**

返回结果：

无。

描述：

删除一个不再使用的 spinlock。**调用该函数时，不应该有任何** CPU **对该锁进行声称，否则将会出现无法预料的结果。**

cyg _ drv _ spinlock _ spin

函数原型：

```
void cyg_drv_spinlock_spin(cyg_spinlock_t *lock)
```

参数：

lock — **指向所要声称的** spinlock **的指针。**

返回结果：

无。

描述：

对一个 spinlock **进行声称，它将循环等待直至获取该** spinlock。**不管从何处调用该函数，这种操作都将使** CPU **等待直至操作成功。这种操作的使用应该尽量少，并且只能用在不会发生死锁/活锁的情形下。参阅** cyg _ drv _ spinlock _ spin _ intsave() **函数。**

cyg _ drv _ spinlock _ clear

函数原型：

```
void cyg_drv_spinlock_clear(cyg_spinlock_t *lock)
```

参数：

lock — **指向被清的** spinlock **的指针。**

返回结果：

无。

描述：

清 spinlock。**该函数对指定的** spinlock **进行清操作，允许其他** CPU **对其进行声称。如果有多个** CPU **在** cyg _ drv _ spinlock _ spin() **中等待，则只能允许其中的一个** CPU **获取该** spinlock。

cyg _ drv _ spinlock _ try

函数原型：

```
cyg _ bool _ t cyg _ drv _ spinlock _ try (cyg _ spinlock _ t * lock)
```

参数：

lock — **指向被声称的** spinlock **的指针。**

返回结果：

如果可以声称 spinlock**，则返回** TRUE**，否则返回** FALSE**。**

描述：

该函数尝试对一个 spinlock **进行声称，它没有等待过程。如果可以立即声称** spinlock**，则返回** TRUE**。如果** spinlock **已经被声称，则返回** FALSE**。**

cyg _ drv _ spinlock _ test

函数原型：

```
cyg _ bool _ t cyg _ drv _ spinlock _ test (cyg _ spinlock _ t * lock)
```

参数：

lock — **指向被测试的** spinlock **的指针。**

返回结果：

如果 spinlock **可用，则返回** TRUE**，否则返回** FALSE**。**

描述：

检查指定 spinlock **的状态。如果** spinlock **没有被锁定，则返回结果** TRUE**，如果该** spinlock **已被锁定则返回** FALSE**。**

cyg _ drv _ spinlock _ spin _ intsave

函数原型：

```
void cyg _ drv _ spinlock _ spin _ intsave (cyg _ spinlock _ t * lock,
                                           cyg _ addrword _ t * istate
                                          )
```

参数：

lock — **指向所要声称的** spinlock **的指针。**

istate — **指向中断状态保存位置的指针。**

返回结果：

无。

描述：

该函数非常类似于 cyg _ drv _ spinlock _ spin () **函数，不同之处是该函数在试图声称该锁之前还将禁止中断。当前中断的使能状态被保存在参数** istate **所指定的位置。一旦** spinlock **被声称，中断将保持禁止状态，必须通过调用** cyg _ drv _ spinlock _ clear _ intsave () **函数来恢复。一般来说，设备驱动程序应该使用该函数对** spinlock **进行声称和释放操作，以确保运行在本** CPU **和其他** CPU **上的程序的隔离。**

cyg _ drv _ spinlock _ clear _ intsave

函数原型：

```
void cyg _ drv _ spinlock _ clear _ intsave (
        cyg _ spinlock _ t  * lock,
        cyg _ addrword _ t istate
)
```

参数：

lock — **指向被清** spinlock **的指针。**

istate — **所要恢复的中断状态。**

返回结果：

无。

描述：

该函数与 cyg _ drv _ spinlock _ clear () **函数十分相似，不同之处是该函数还将恢复被函数** cyg _ drv _ spinlock _ spin _ intsave () **保存的中断状态。参数** istate **必须已经被先前调用的** cyg _ drv _ spinlock _ spin _ intsave () **函数所初始化。**

cyg _ drv _ dsr _ lock

函数原型：

```
void cyg _ drv _ dsr _ lock (void)
```

参数：

无。

返回结果：

无。

描述：

禁止对中断滞后服务程序 DSR **的调度。该函数对一个计数器进行维护，该计数器记录该函数被调用的次数。**

cyg _ drv _ dsr _ unlock

函数原型：

```
void cyg _ drv _ dsr _ unlock (void)
```

参数：

无。

返回结果：

无。

描述：

重新使能对中断滞后服务程序 DSR **的调度。该函数对** cyg _ drv _ dsr _ lock () **所维护的计数器进行减一操作，只有当计数器为** 0 **时才允许** DSR **运行。**

cyg _ drv _ mutex _ init

函数原型：

```
void cyg_drv_mutex_init(cyg_drv_mutex_* mutex)
```

参数：

mutex — **指向被初始化的互斥体。**

返回结果：

无。

描述：

该函数用于对参数 mutex **指定的互斥体进行初始化。**

cyg_drv_mutex_destroy

函数原型：

```
void cyg_drv_mutex_destroy(cyg_drv_mutex_* mutex)
```

参数：

mutex — **指向要取消的互斥体。**

结果：

无。

描述：

取消参数 mutex **指定的互斥体。调用该函数时，该互斥体应该处于非锁定状态，并且没有线程等待对该互斥体的锁定。**

cyg_drv_mutex_lock

函数原型：

```
cyg_bool cyg_drv_mutex_lock(cyg_drv_mutex_* mutex)
```

参数：

mutex — **指向被锁定的互斥体。**

返回结果：

如果成功锁定该互斥体，则返回 TRUE，**否则返回** FALSE。

描述：

对参数 mutex **指定的互斥体进行锁定操作。如果该互斥体已经被另一个线程锁定，则调用该函数的线程将处于等待状态，直至锁定该互斥体的线程释放。如果该函数返回结果为** FALSE，**则该线程的等待状态被其他线程所打破，互斥体在这种情况下将不会被锁定。**

cyg_drv_mutex_trylock

函数原型：

```
cyg_bool cyg_drv_mutex_trylock(cyg_drv_mutex_* mutex)
```

参数：

mutex — **指向被锁定的互斥体。**

返回结果：

如果互斥体被成功锁定，则返回 TRUE，**否则返回** FALSE。

描述：

对参数 mutex **指定的互斥体进行锁定操作，该函数没有等待过程。如果互斥体已经被其**

他线程锁定则该函数返回 FALSE。如果可以锁定互斥体而且不需要等待,则返回 TRUE。

cyg _ drv _ mutex _ unlock

函数原型:

```
void cyg_drv_mutex_unlock(cyg_drv_mutex *mutex)
```

参数:

mutex — 指向被解锁的互斥体的指针。

返回结果:

无。

描述:

对 mutex 指定的互斥体进行解锁操作。如果有线程正在等待对该互斥体的声称,其中的一个线程将被唤醒并获取对该互斥体的声称。

cyg _ drv _ mutex _ release

函数原型:

```
void cyg_drv_mutex_release(cyg_drv_mutex *mutex)
```

参数:

mutex — 指向被释放的互斥体的指针。

返回结果:

无。

描述:

释放正在等待参数 mutex 所指定互斥体的所有线程。这些线程将从 cyg _ drv _ mutex _ lock() 函数返回,结果为 FALSE,并且不会获取对该互斥体的声称。该函数不会对已经声称该互斥体的线程有任何影响。

cyg _ drv _ cond _ init

函数原型:

```
void cyg_drv_cond_init(cyg_drv_cond *cond,
                       cyg_drv_mutex *mutex
                      )
```

参数:

cond — 被初始化的条件变量。

mutex — 与该条件变量相对应的互斥体。

返回结果:

无。

描述:

对参数 cond 指定的条件变量进行初始化。参数 mutex 必须指向该条件变量所对应的互斥体。线程在已经锁定对应的互斥体时,可以等待该条件变量。这种等待将引起互斥体的解锁,当该线程被重新唤醒时,它将自动声称该互斥体。

cyg _ drv _ cond _ destroy

函数原型：

```
void cyg_drv_cond_destroy(cyg_drv_cond *cond)
```

参数：

cond — **被取消的条件变量。**

返回结果：

无。

描述：

取消参数 cond **指定的条件变量。**

cyg _ drv _ cond _ wait

函数原型：

```
void cyg_drv_cond_wait(cyg_drv_cond *cond)
```

参数：

cond — **将要等待的条件变量。**

返回结果：

无。

描述：

等待参数 cond **所指定条件变量的一个信号。进行这种操作的线程在等待该条件变量之前必须已经锁定对应的互斥体(参阅** cyg _ drv _ cond _ init **()函数),互斥体在线程等待条件变量时将被解锁,该函数返回前将重新锁定互斥体。在线程等待条件变量的过程中有可能偶尔被虚假唤醒,因此在使用该函数时最好将它放在一个对测试条件进行查询的循环体中。值得注意的是,该函数隐含了一个调度器的解锁/重锁(**unlock/relock**)过程,所以它可以用在** cyg _ drv _ dsr _ lock **()…**cyg _ drv _ dsr _ unlock **()结构中。**

cyg _ drv _ cond _ signal

函数原型：

```
void cyg_drv_cond_signal(cyg_drv_cond *cond)
```

参数：

cond — **接收信号的条件变量。**

返回结果：

无。

描述：

发信号给参数 cond **指定的条件变量。如果有其他正在等待该条件变量的线程,则至少有一个线程将被唤醒。值得注意的是,在某些配置中该函数与** cyg _ drv _ cond _ broadcast **()函数没有任何区别。**

cyg _ drv _ cond _ broadcast

函数原型：

```
void cyg_drv_cond_broadcast(cyg_drv_cond *cond)
```

参数：

cond — **被广播的条件变量。**

返回结果：

无。

描述：

给参数 cond **指定的条件变量发信号。如果有线程正在等待该条件变量，则这些线程将被唤醒。**

cyg _ drv _ interrupt _ create

函数原型：

```
void cyg_drv_interrupt_create(cyg_vector_t vector,
                              cyg_priority_t priority,
                              cyg_addrword_t data,
                              cyg_ISR_t *isr,
                              cyg_DSR_t *dsr,
                              cyg_handle_t *handle,
                              cyg_interrupt *intr
                             )
```

参数：

vector — **中断向量。**

priority — **队列优先级。**

data — **数据指针。**

isr — **中断服务程序** ISR。

dsr — **中断滞后服务程序** DSR。

handle — **返回句柄。**

intr — **中断对象存放位置。**

返回结果：

无。

描述：

产生一个中断对象 (interrupt object**) 并返回其句柄** handle。**中断对象包含的信息有：分配给中断对象的中断向量、将要调用的中断服务程序** ISR **和中断滞后服务程序** DSR。**中断对象位于参数** intr **所指定的内存位置。中断对象的分配必须调用** cyg _ interrupt _ attach () **函数来完成。**

cyg _ drv _ interrupt _ delete

函数原型：

```
void cyg_drv_interrupt_delete(cyg_handle_t interrupt)
```

参数：

interrupt — **将要删除的中断。**

返回结果：

无。

描述：

将中断与中断向量分离，释放由 cyg _ drv _ interrupt _ create () **函数的参数** intr **所分配的内存。**

cyg _ drv _ interrupt _ attach

函数原型：

```
void cyg_drv_interrupt_attach(cyg_handle_t interrupt)
```

参数：

interrupt — **将要连接的中断。**

返回结果：

无。

描述：

将中断连接到中断向量，使得当中断发生时可以将中断交付给相应的中断服务程序 ISR **进行处理。**

cyg _ drv _ interrupt _ detach

函数原型：

```
void cyg_drv_interrupt_detach(cyg_handle_t interrupt)
```

参数：

interrupt — **将要分离的中断。**

返回结果：

无。

描述：

将中断与中断向量分离，使得中断不会再交付给中断服务程序 ISR **处理。**

cyg _ drv _ interrupt _ mask

函数原型：

```
void cyg_drv_interrupt_mask(cyg_vector_t vector)
```

参数：

vector — **将要屏蔽的中断向量。**

返回结果：

无。

描述：

对中断控制器进行编程，中断向量 vector **所对应的中断停止交付处理。在实现了中断优先级的系统中，该函数还将禁止比该中断优先级低的所有中断。**

cyg _ drv _ interrupt _ mask _ intunsafe

函数原型：

```
void cyg_drv_interrupt_mask_intunsafe(cyg_vector_t vector)
```

参数：

vector — 将要屏蔽的中断向量。

返回结果：

无。

描述：

对中断控制器进行编程，中断向量 vector 所对应的中断停止交付处理。在实现了中断优先级的系统中，该函数还将禁止比该中断优先级低的所有中断。该函数与 cyg _ drv _ interrupt _ mask () 的区别在于它不具有中断安全性。在已经知道中断被禁止的情况下，可以调用该函数来避免额外的系统开销。

cyg _ drv _ interrupt _ unmask

函数原型：

```
void cyg_drv_interrupt_unmask(cyg_vector_t vector)
```

参数：

vector — 将被解除屏蔽的中断向量。

返回参数：

无。

描述：

对中断控制器进行编程，重新允许中断向量 vector 对应的中断交付处理。

cyg _ drv _ interrupt _ unmask _ intunsafe

函数原型：

```
void cyg_drv_interrupt_unmask_intunsafe(cyg_vector_t vector)
```

参数：

vector — 将要解除屏蔽的中断向量。

返回结果：

无。

描述：

对中断控制器进行编程，重新允许中断向量 vector 对应的中断交付处理。与 cyg _ drv _ interrupt _ unmask () 函数的不同之处是它不具有中断安全性。

cyg _ drv _ interrupt _ acknowledge

函数原型：

```
void cyg_drv_interrupt_acknowledge(cyg_vector_t vector)
```

参数：

vector — 中断向量。

返回结果：

无。

描述：

中断应答。对中断控制器和 CPU 进行必需的操作，用以清除基于 vector 的当前中断请

求。中断服务程序 ISR 可能还需要对设备硬件进行操作以防止中断立即重新触发。

cyg _ drv _ interrupt _ configure

函数原型：

```
void cyg_drv_interrupt_configure(cyg_vector_t vector,
                                 cyg_bool_t level,
                                 cyg_bool_t up
                                )
```

参数：

vector — 被配置的中断向量。

level — 中断的触发方式：电平触发或边沿触发。

up — 上升沿/下降沿，高电平/低电平。

返回结果：

无。

描述：

根据中断源的特性对中断控制器进行配置。参数 level 指定中断的触发方式——电平触发还是边沿触发。参数 up 在电平触发方式中指定是高电平触发还是低电平触发，在边沿触发方式中指定是上升沿触发还是下降沿触发。该函数只对控制这些参数的中断控制器进行操作。

cyg _ drv _ interrupt _ level

函数原型：

```
void cyg_drv_interrupt_level(cyg_vector_t vector,
                             cyg_priority_t level
                            )
```

参数：

vector — 被配置的中断向量。

level — 中断优先级。

返回结果：

无。

描述：

对中断控制器进行编程，给中断指定一个优先级。该函数只对能够控制这种优先级参数的中断控制器进行操作。

cyg _ drv _ interrupt _ set _ cpu

函数原型：

```
void cyg_drv_interrupt_set_cpu(cyg_vector_t vector,
                               cyg_cpu_t cpu
                              )
```

参数：

vector — 中断向量。

cpu — 目标 CPU。

返回结果：

无。

描述：

对中断进行路由设置。将 vector 指定的所有中断路由到指定的 CPU，由该 CPU 对这些中断进行处理。只有在提供这种路由支持能力的硬件平台上才进行这种操作，在单 CPU 系统中该函数不做任何工作。

cyg _ drv _ interrupt _ get _ cpu

函数原型：

```
cyg _ cpu _ t cyg _ drv _ interrupt _ get _ cpu(cyg _ vector _ t vector)
```

参数：

vector — 中断向量。

返回结果：

中断向量 vector 被路由到哪一个 CPU。

描述：

在多处理器系统中，该函数返回 CPU 的标识号，该 CPU 处理中断向量 vector 对应的所有中断。在单 CPU 系统中，该函数返回 0。

cyg _ ISR _ t

原型：

```
typedef cyg _ uint32 cyg _ ISR _ t (cyg _ vector _ t vector,
                                    cyg _ addrword _ t data
                                   )
```

参数：

vector — 中断向量。

data — 由客户程序提供的数据值。

返回结果：

表明中断是否被处理以及是否应该调用中断滞后服务程序 DSR 的位表征码。

描述：

中断服务程序 ISR。在产生一个中断对象时，具有该函数原型的中断服务程序 ISR 的指针被传递给 cyg _ interrupt _ create() 函数。当一个中断交付处理时将调用该函数，并使用传递到 cyg _ interrupt _ create() 函数的中断向量 vector 和数据值 data。

中断服务程序 ISR 的返回值是一个位表征码，包含一个或两个下面的值：

① CYG _ ISR _ HANDLED。表示中断被该 ISR 处理。这是一个决定是否阻止更多 ISR 运行的配置选项。

② CYG _ ISR _ CALL _ DSR。引起传递给 cyg _ interrupt _ create() 函数的中断滞后服务程序 DSR 调度运行。

cyg _ DSR _ t

原型：

```
typedef void cyg _ DSR _ t (cyg _ vector _ t vector,
                            cyg _ ucount32 count,
                            cyg _ addrword _ t data
                           )
```

参数：

vector — **中断向量。**

count — DSR **被调度的次数。**

data — **由客户程序提供的数据值。**

返回结果：

无。

描述：

中断滞后服务程序 DSR**。在产生一个中断对象时，具有该函数原型的中断滞后服务程序** DSR **的指针被传递给** cyg _ interrupt _ create () **函数。当** ISR **请求调度该中断的** DSR **时，该** DSR **函数将在稍后的某个时刻开始运行。其参数** vector **和** data **必须与传递给** ISR **的参数相同。该函数还有一个参数** count **用于记录** ISR **请求** DSR **被调度运行的次数，在** DSR **的每一次实际运行时该计数器将置** 0**，因此计数器的值表明了自从** DSR **上次运行以来共发生了多少次中断。**

8.3 eCos 驱动程序设计

设备驱动程序主要用于对设备进行数据的读写操作以及对设备进行配置和读取配置信息的操作，它还可以使用和管理来自设备的中断。所有设备的驱动程序接口都具有普遍性，这种接口形式独立于具体的设备驱动程序。使用驱动程序对设备进行访问时，应该使用这种通用的接口形式。在设计一个具体的设备驱动程序时，必须考虑如何使设备的访问尽可能简单、有效。

设备驱动程序所涉及的是信息的传送，如串行接口中的字节数据、网络设备中的数据包等等。在大多数设备中，为了提高系统的效率，通常使用了中断。设备的数据传送过程通常需要花费很长的一段时间，如果让 CPU **在整个设备数据传送过程中处于等待状态，将会造成极大的资源浪费。中断的使用可以实现在设备数据传送过程的同时让** CPU **处理其他应用程序。系统在处理正常应用程序时，中断的发生表示出现了一些需要处理的事件。以串口的数据传送过程为例，串口在一个字符被发送之后将产生一个中断，此时串行接口已为下一次传送作好了准备。通过中断的使用，一旦当前数据传送完成，驱动程序就可以立即启动下一次数据传送，而不需要任何应用程序的参与。**

8.3.1 设备驱动程序的基本结构

图 8-2 **为** eCos **设备驱动程序在系统中的位置及其组成结构示意图。应用程序在使用设**

备的时候，它通过驱动程序的用户 API 访问设备驱动程序，而设备驱动程序通过设备内核 API 与内核和硬件抽象层 HAL 进行交互，设备驱动程序和内核再通过 HAL 对硬件平台进行操作，从而实现对设备的访问。

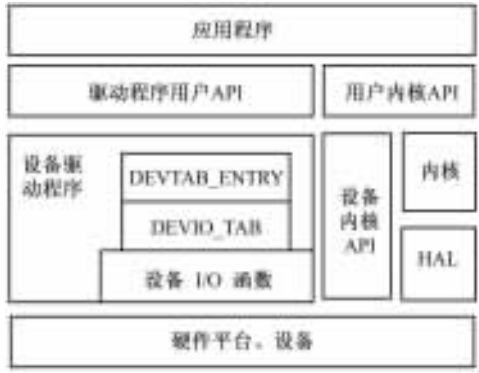


图 8-2　设备驱动程序结构

设备驱动程序一般可分为三个部分，分别为设备表入口 DEVTAB _ ENTRY、设备 I/O 函数表 DEVIO _ TAB 和设备 I/O 函数。设备驱动程序的主要组成模块全部定义在头文件 cyg/io/devtab.h 中。

eCos 中的所有设备驱动程序都使用设备表入口来进行描述。设备表入口是一个具有 cyg _ devtab _ ebtry _ t 类型的数据结构，使用宏 DEVTAB _ ENTRY () 可以生成一个设备表入口，其格式为：

DEVTAB _ ENTRY (l, name, dep _ name, handlers, init, lookup, priv)

其中：

l — 该设备表入口的“C”标识符。

name — 该设备的“C”字符串名字。

dep _ name — 对于一个层次设备，此参数是该设备的下层设备的“C”字符串名字。

handles — I/O 函数句柄指针 (见下面的设备 I/O 函数表 DEVIO _ TAB 的详细说明)。

init — 当 eCos 处于初始化阶段时被调用的函数，该函数可以进行查找设备、对硬件进行设置等操作。

lookup — 当调用 cyg _ io _ lookup () 函数对该设备进行操作时所调用的一个函数。

priv — 该设备驱动程序所需的专用数据存放位置。

上面所定义的设备表入口只适用于字符设备。对于块设备，要使用宏 BLOCK _ DEVTAB _ ENTRY 进行定义。其格式为：

BLOCK _ DEVTAB _ ENTRY (l, name, dep _ name, handlers, init, lookup, priv)

设备表入口中的句柄 handles 提供了一组设备驱动程序接口函数。handlers 是设备 I/O 函数表 DEVIO _ TAB 的指针，DEVIO _ TAB 包含了一组函数的指针，这些函数是各种接口函数 cyg _ io _ XXX () 的具体实现。

设备 I/O 函数表通过 DEVIO _ TAB 宏来定义，其格式如下：

DEVIO _ TABLE (, write, read, get _ config, set _ config)

其中：

l — 该表的“C”标识符。

write — cyg _ io _ write () 函数所调用的函数，实现向设备传送数据。

read — cyg _ io _ read () 函数所调用的函数，实现从设备读取数据。

get _ config — cyg _ io _ get _ config () 函数所调用的函数，实现对设备配置信息的读取操作。

set _ config — cyg _ io _ set _ config () 函数所调用的函数，完成对设备的配置操作

在 eCos 的初始化引导过程中，对系统中的所有设备都要调用其相应的 init () 函数。init () 函数有可能返回错误，出现这种情况时，设备将处于“离线”状态，所有针对离线设备的 I/O 请求都将返回错误。

当使用某个设备的名字调用 cyg _ io _ lookup () 函数时，该设备的设备表入口中指定的 lookup () 将被调用。lookup () 函数将使设备处于“在线”状态，处于在线状态下的设备对所有针对该设备的 I/O 请求进行处理。

后面的章节将通过举例来说明如何进行设备驱动程序的设计。

8.3.2 串口驱动程序设计

eCos 提供的标准串口驱动程序由与硬件无关的部分以及与硬件相关的接口模块两部分组成。如果要增加并支持一个新的串口，可以使用已有的与硬件无关的部分，只需增加对实际设备进行具体操作处理的接口模块，与硬件无关的部分不需要修改。这种串口驱动程序与串口设备模块之间的接口定义包含在头文件 cyg/io/serial.h 内。这一节主要描述如何设计一个串口硬件接口驱动程序。

1. DevTab 入口

与串口硬件相关的接口模块包含了一个设备表入口 (DEVTAB _ ENTRY)，该表具有如下格式：

```
DEVTAB _ ENTRY (<<module _ name>>,
                <<device _ name>>,
                0,
                &serial _ devio,
                <<module _ init>>,
                <<module _ lookup>>,
                &<<serial _ channel>>
               );
```

其参数说明如下：

module _ name—设备表入口 DEVTAB _ ENTRY 的 C 语言标识符。

device _ name—设备名字的 C 语言字符串描述，如：/dev/serial0。

serial _ devio—I/O 函数表，这是一组在串口驱动程序与硬件无关部分中定义的函数。

module _ init—模块初始化函数。

module _ lookup—设备查找函数，该函数通常对设备进行设置、使能中断、配置端口等操作。

serial _ channel—串行通道表，包含了接口模块和串口驱动程序之间接口的表（见下一节的说明）。

2. Serial _ Channel 结构

每一个串口设备必须有一个 Serial _ Channel，这是一组用于描述对设备所进行的全部操作的数据结构。如果设备具有数据缓冲能力，那么这种数据结构还包含了 buffer 等信息。通过使用下面的宏来产生 Serial _ Channel 结构：

```
SERIAL _ CHANNEL _ USING _INTERRUPTS (l, funs, dev _ priv, baud,stop, parity,
                                      word _ length,flags, out _ buf, out _ buflen, in _ buf,
                                      in _ buflen)
```

其参数说明如下：

l—该数据结构的 C 语言标识符。

funs—接口函数组。

baud—波特率初始值（cyg _ serial _ baud _ t）。

stop—停止位初始值（cyg _ serial _ stop _ bits _ t）。

parity—奇偶方式初始值（cyg _ serial _ parity _ t）。

word _ length—字长初始值（cyg _ serial _ word _ length _ t）。

flags—驱动程序初始标志值。

out _ buf—输出 buffer 指针，如果不需要 buffer，则为 NULL。

out _ buflen—输出 buffer 长度。

in _ buf—输入 buffer 指针，如果不需要则为 NULL。

in _ buflen—输入 buffer 长度。

如果输入 buffer 或输出 buffer 的长度为 0，则不会形成输入或输出的 buffer 操作，这种情况下的函数只使用查询方式。

3. 串口函数结构

Serial _ Channel 结构中的 funs 表包含了与硬件无关的驱动程序模块和硬件接口模块之间的接口函数。funs 表由下面的宏进行定义：

```
SERIAL _ FUNS (l, putc, getc, set _ config, start _ xmit, stop _ xmit)
```

其参数说明如下：

l—该 funs 表的 C 语言标识符。

putc—函数：

```
bool (* putc) (serial _ channel * priv, unsigned char c)
```

该函数发送一个字符到接口。如果字符已被传送，则返回 true 如果接口没有空间，则返回 false。

getc—函数：

```
unsigned char (* getc) (serial _ channel * priv)
```

该函数从接口读取一个字符，它只用于非中断驱动方式，通过查询设备是否处于准备(ready)状态来等待一个字符。

set _ config一函数：

```
bool (* set _ config) (serial _ channel * priv, cyg _ serial _ info _ t * config)
```

该函数用于对端口进行配置。如果对硬件配置成功，则返回 true 如果端口不支持给定的配置参数，则返回 false。例如，大多数串口设备不能同时支持停止位为 1.5 和数据位为 8 的设置，因此如果使用了这种配置，就会返回 false。

start _ xmit一函数：

```
void (* start _ xmit) (serial _ channel * priv)
```

在中断方式下，该函数使能发送端，允许发送中断的产生。

4. 回调函数

串口设备接口模块可以使用 chan->callbacks 的形式调用一些与硬件无关的驱动程序模块内的函数。这些回调函数有：

```
void (* serial _ init) (serial _ channel * chan)
```

该函数用于串行通道的初始化操作。只有在以中断方式使用通道的情况下才需要这一函数。

```
void (* xmt _ char) (serial _ channel * chan)
```

对发送中断(表示有字符可以被发送)进行处理的中断处理程序将调用该函数。高层驱动程序将调用 putc 函数向设备发送更多的数据。

```
void (* rcv _ char) (serial _ channel * chan, unsigned char c)
```

该函数用于告诉驱动程序已经有一个字符到达接口。通常由中断处理程序来调用该函数。

此外，如果设备具有 FIFO，那么与硬件无关的驱动程序模块应该提供块传输功能(驱动程序 CDL 描述脚本内包含了"implements CYGINT _ IO _ SERIAL _ BLOCK _ TRANSFER")。在这种情况下，还必须提供下面的一些函数：

```
bool (* data _ xmt _ req) (serial _ channel * chan,
                           int space,
                           int * chars _ avail,
                           unsigned char * * chars)
void (* data _ xmt _ done) (serial _ channel * chan)
```

驱动程序在调用 xmt _ char()函数时，一次传送操作只发送一个字符。如果驱动程序以循环方式调用 data _ xmt _ req()函数，则可以请求进行数据块的传送。调用该函数时使用参数 space 来指明 FIFO 中的可用空间大小。

如果 data _ xmt _ req()函数返回 true，驱动程序可以从 cbars 指定的位置读取 cbars _ avail 个字符，并将这些字符拷贝到 FIFO。如果返回 false，则表示 buffer 内没有字符，驱动程序将继

续工作，但不会对 FIFO 进行写操作。

当所有的数据传送完成后，驱动程序必须调用 data _ xmt _ done () 函数。

```
bool (* data _ rcv _ req) (serial _ channel * chan,
                           int avail,
                           int * space _ avail,
                           unsigned char * * space)
void (* data _ rcv _ done) (serial _ channel * chan)
```

驱动程序在调用 rcv _ char () 函数时，一次传送操作只读取一个字符。如果驱动程序以循环方式调用 data _ rcv _ req () 函数，则可以请求进行数据块的传送，它请求一定的空间用于从 FIFO 内读取数据。参数 avail 指明驱动程序希望从 FIFO 中读取的字符个数。

如果 data _ rcv _ req () 函数返回 true，则驱动程序将复制 space _ avail 个字符到 space 指定的空间。如果返回 false，则说明输入 buffer 满，此时需要驱动程序来决定如何进行处理。

当所有数据从 FIFO 读取完成后，驱动程序必须调用 data _ rcv _ done () 函数，表示 FIFO 数据读取操作完成。

8.4 串口驱动程序

eCos 提供两种标准的串口驱动程序，它们是串口 (raw serial) 驱动程序和类 TTY (tty-like) 驱动程序，分别用 Serial 和 TTY 来表示。本节介绍 eCos 的这两种串口驱动程序的详细实现过程。读者可以通过了解串口驱动程序的结构和具体内容来掌握 eCos 驱动程序的设计方法。

8.4.1 串口 (raw serial) 驱动程序

Serial 串口驱动程序实现串行设备数据块的接收或发送，它还可以对串口硬件进行配置。该串口驱动程序使用头文件 cyg/io/serialio.h。

一个系统中可能有多个这种驱动程序的实例来对应每一个串行通道。每一个串行通道与一个具体的物理设备相对应，并产生一个相应的设备模块。设备模块本身具有可配置性，这种可配置性可以对具体的硬件特性进行配置。

1. 运行时 (runtime) 配置

serial 串口驱动程序具有运行时的配置能力，可以通过调用 cyg _ io _ set _ config () 函数和 cyg _ io _ get _ config () 函数将配置数据传递给驱动程序来实现这种运行时的配置。这种配置数据的结构定义如下：

```
typedef struct {
        cyg _ serial _ baud _ rate _ t baud;
        cyg _ serial _ stop _ bits _ t stop;
        cyg _ serial _ parity _ t parity;
        cyg _ serial _ word _ length _ t word _ length;
        cyg _ uint32 flags;
} cyg _ serial _ info _ t;
```

其中：

word _ length **包含了字(字符)的长度,其值必须为下列四种值之一：**

CYGNUM _ SERIAL _ WORD _ LENGTH _ 5
CYGNUM _ SERIAL _ WORD _ LENGTH _ 6
CYGNUM _ SERIAL _ WORD _ LENGTH _ 7
CYGNUM _ SERIAL _ WORD _ LENGTH _ 8

baud **为所选择的波特率,必须是下述值之一：**

CYGNUM _ SERIAL _ BAUD _ 50
CYGNUM _ SERIAL _ BAUD _ 75
CYGNUM _ SERIAL _ BAUD _ 110
CYGNUM _ SERIAL _ BAUD _ 134 _ 5
CYGNUM _ SERIAL _ BAUD _ 150
CYGNUM _ SERIAL _ BAUD _ 200
CYGNUM _ SERIAL _ BAUD _ 300
CYGNUM _ SERIAL _ BAUD _ 600
CYGNUM _ SERIAL _ BAUD _ 1200
CYGNUM _ SERIAL _ BAUD _ 1800
CYGNUM _ SERIAL _ BAUD _ 2400
CYGNUM _ SERIAL _ BAUD _ 3600
CYGNUM _ SERIAL _ BAUD _ 4800
CYGNUM _ SERIAL _ BAUD _ 7200
CYGNUM _ SERIAL _ BAUD _ 9600
CYGNUM _ SERIAL _ BAUD _ 14400
CYGNUM _ SERIAL _ BAUD _ 19200
CYGNUM _ SERIAL _ BAUD _ 38400
CYGNUM _ SERIAL _ BAUD _ 57600
CYGNUM _ SERIAL _ BAUD _ 115200
CYGNUM _ SERIAL _ BAUD _ 234000

stop **为停止位,其值必须是下列值之一：**

CYGNUM _ SERIAL _ STOP _ 1
CYGNUM _ SERIAL _ STOP _ 1 _ 5
CYGNUM _ SERIAL _ STOP _ 2

在大多数系统中,停止位 1.5 **只在字长为** 5 **时才有效。**

parity **指定奇偶方式,其值为下列值之一：**

CYGNUM _ SERIAL _ PARITY _ NONE
CYGNUM _ SERIAL _ PARITY _ EVEN
CYGNUM _ SERIAL _ PARITY _ ODD
CYGNUM _ SERIAL _ PARITY _ MARK
CYGNUM _ SERIAL _ PARITY _ SPACE

flags 为位屏蔽,它对串口驱动程序的行为进行控制。它的值将根据下面定义的 CYG _ SERIAL _ FLAGS _ xxx 的值来产生:

```
#define CYG_SERIAL_FLAGS_RTSCTS 0x0001
```

如果该值设为 1,则串口处于"硬握手"方式。在这种方式下,CTS 和 RTS 信号对串口数据的发送和接收进行控制。如果硬件不支持这种握手,则忽略这一位。

另一个用于运行时配置的数据结构是:

```
typedef struct {
  cyg_int32 rx_bufsize;
  cyg_int32 rx_count;
  cyg_int32 tx_bufsize;
  cyg_int32 tx_count;
} cyg_serial_buf_info_t;
```

其中:

rx _ bufsize—指定数据输入 buffer 的大小,对于不支持缓冲的设备其值为 0。

rx _ count—指定当前数据输入 buffer 中的字节数目。对于不支持缓冲的设备其值为 0。

tx _ bufsize—指定数据发送 buffer 的大小,对于不支持缓冲的设备其值为 0。

rx _ count—指定当前数据发送 buffer 中的字节数目。对于不支持缓冲的设备其值为 0。

2. 串口驱动程序 API

串口驱动程序提供了一些 API 函数,应用程序可以使用这些函数对串口进行操作。

```
cyg_io_write(handle, buf, len)
```

该函数将数据从缓冲区 buf 发送到设备。驱动程序提供一个缓冲区来保存数据,这种中间缓冲区的大小可以使用接口模块进行配置。存放在缓冲区内的数据不会被修改。函数返回时,len 包含了被实际传输的字符数目。

可以使用配置工具对该函数的调用进行配置,使其以块方式(默认方式)或者非块方式向设备写数据。非块方式要求使能配置选项 CYGOPT _ IO _ SERIAL _ SUPPORT _ NON BLOCKING,并且必须将指定的设备设置为以非块方式进行写操作(参阅 cyg _ io _ set _ config ()函数)。

在以块方式的调用中,在缓冲区有空闲空间并且参数 buf 指定的所有内容传送完成之前,该函数不会返回。

在以非块方式进行的调用中,将尽可能多地从参数 buf 指定的位置传送数据。如果所有的数据传送完成,则该函数返回 ENOERR。如果只有部分数据被传送,则返回-EAGAIN,并且必须再次调用该函数来完成余下的数据传送操作。参数 len 包含了此次调用实际传送的字符数目。

如果调用 cyg _ io _ get _ config ()函数时参数 key 使用了 ABORT,则 cyg _ io _ write ()函数将返回-EINTR。

```
cyg_io_read(handle, buf, len)
```

该函数从设备接收数据并将数据存放到参数 buf 指定的缓冲区内。在数据传送之前不会

对数据进行操作。当没有挂起的读操作时，由中断驱动的接口模块对从设备来的数据进行处理，将其存放到驱动程序的缓冲区内。数据的缓冲操作与写函数一样，完全具有可配置性。函数返回时，len 包含了实际接收到的字符数目。

该函数的调用可以被配置工具配置为块方式（默认方式）或者非块方式从设备接收数据。非块方式要求使能配置选项 CYGOPT _ IO _ SERIAL _ SUPPORT _ NONBLOCKING，并且指定的设备必须已被设置为以非块方式进行读操作（参阅 cyg _ io _ set _ config () 函数）。

在块方式调用中，该函数在所有被请求的数据传送完成之前不会返回。

在非块方式调用中，在设备缓冲区中等待的数据将被存放到参数 buf 所指定的位置，随后该函数立即返回。如果读请求的所有数据全部传送完成，则返回 ENOERR，如果只完成了读请求的部分数据，则返回-EAGAIN，此时必须重新调用该函数来完成余下的数据传送操作。len 指明了实际读取的字符数。

如果调用 cyg _ io _ get _ config () 函数时参数 key 使用了 ABORT，则 cyg _ io _ read () 函数将返回-EINTR。

```
cyg _ io _ get _ config (handle, key, buf, len)
```

该函数返回设备和（或）驱动程序的当前（运行时）信息。其参数 key 指定读取哪一类信息。各种 key 值及相应的 buf 类型和函数功能说明如下：

key 值：CYG _ IO _ GET _ CONFIG _ SERIAL _ INFO

buf 类型：

```
cyg _ serial _ info _ t
```

函数功能：

该函数读取驱动程序和硬件的当前状态。buf 返回的信息包含了硬件波特率、停止位和奇偶方式。还包含了一组串口控制标志，如硬件流控制等。

key 值：CYG _ IO _ GET _ CONFIG _ SERIAL _ BUFFER _ INFO

buf 类型：

```
cyg _ serial _ buf _ info _ t
```

函数功能：

该函数读取串口驱动程序内部软 buffer 的当前状态。它返回接收 buffer 和发送 buffer 的大小和 buffer 内的字节数。它不包括串行设备内部自身的 buffer 如 FIFO 或寄存器。

key 值：CYG _ IO _ GET _ CONFIG _ SERIAL _ OUTPUT _ DRAIN

buf 类型：

```
void *
```

函数功能：

该函数等待任何被缓冲的输出完成。只有在没有任何发送到设备的遗留数据才算操作完成。

key 值：CYG _ IO _ GET _ CONFIG _ SERIAL _ OUTPUT _ FLUSH

buf 类型：

void *

函数功能：

该函数作废掉所有被缓冲的输出数据。

key **值：**CYG_IO_GET_CONFIG_SERIAL_INPUT_DRAIN

buf **类型：**

void *

函数功能：

该函数作废掉所有被缓冲的输入数据。

key **值：**CYG_IO_GET_CONFIG_SERIAL_ABORT

buf **类型：**

void *

函数功能：

该函数将使所有处于挂起状态的对该设备进行读写操作的调用返回-EABORT**。**

key **值：**CYG_IO_GET_CONFIG_SERIAL_READ_BLOCKING

buf **类型：**

cyg_unit32 **(值为** 0 **或** 1**)**

函数功能：

该函数将读取对该设备进行读操作调用时的块方式设置信息。只有在配置选项 CYGOPT_IO_SERIAL_SUPPORT_NONBLOCKING **被使能时，该调用才可用。**

key **值：**CYG_IO_GET_CONFIG_SERIAL_WRITE_BLOCKING

buf **类型：**

cyg_unit32 **(值为** 0 **或** 1**)**

函数功能：

该函数将读取对该设备进行写操作调用时的块方式设置信息。只有在配置选项 CYGOPT_IO_SERIAL_SUPPORT_NONBLOCKING **被使能时，该调用才可用。**

cyg_io_set_config **(**handle，key，buf，len**)**

该函数用于更新或改变串口的运行时配置。其参数 key **指定对哪一类信息进行修改。各种** key **值及相应的** buf **类型和函数功能说明如下：**

key **值：**CYG_IO_SET_CONFIG_SERIAL_INFO

buf **类型：**

cyg_serial_info_t

函数功能：

该函数对驱动程序和硬件信息进行更新，buf **包含的信息有硬件波特率、停止位和奇偶方式。还包含了一组串口控制标志，如硬件流控制等。**

key **值：**CYG_IO_SET_CONFIG_SERIAL_READ_BLOCKING

buf 类型：

cyg _ unit32 (值为 0 或 1)

函数功能：

该函数对该设备进行读操作调用时的块方式进行设置。只有在配置选项 CYGOPT _ IO _ SERIAL _ SUPPORT _ NONBLOCKING 被使能时，该调用才可用。

key 值：CYG _ IO _ SET _ CONFIG _ SERIAL _ WRITE _ BLOCKING

buf 类型：

cyg _ unit32 (值为 0 或 1)

函数功能：

该函数将对该设备进行写操作调用时的块方式进行设置。只有在配置选项 CYGOPT _ IO _ SERIAL _ SUPPORT _ NONBLOCKING 被使能时，该调用才可用。

8.4.2 TTY 驱动程序

TTY 驱动程序建立在 serial 串口驱动程序之上，通常用于人机接口设备，如终端设备。它提供更为细化的数据输出格式，并且允许对其输入进行行编辑。

TTY 驱动程序使用头文件 cyg/io/ttyio.h。

1. 运行时 (runtime) 配置

TTY 驱动程序具有运行时配置能力，可以通过调用函数 cyg _ io _ set _ config () 和 cyg _ io _ get _ config () 将配置数据传递给驱动程序来实现这种运行时的配置。这种配置数据的结构定义如下：

```
typedef struct {
        cyg _ uint32 tty _ out _ flags;
        cyg _ uint32 tty _ in _ flags;
} cyg _ tty _ info _ t;
```

其中：

tty _ out _ flags

用于当数据发送到串口时对数据的处理进行控制。它包含了一个位映像，该位映像由下面定义的 CYG _ TTY _ OUT _ FLAGS _ xxx 值组成：

```
#define CYG _ TTY _ OUT _ FLAGS _ CRLF 0x0001 //Map '\n' => '\n\r' on output
```

如果在 tty _ out _ flags 中该位为 1，则当出现字符“\n”时将在发送到设备之前被替换为“\n\r”。

tty _ in _ flags

用于控制如何处理从串口接收到的数据。它包含了一个位映像，该位映像由下面定义的 CYG _ TTY _ IN _ FLAGS _ xxx 值组成：

```
#define CYG _ TTY _ IN _ FLAGS _ CR 0x0001 //Map '\r' => '\n' on input
```

如果 tty _ in _ flags 中的该位为 1,则字符“ \ r”(大多数键盘中的回车键) 将被映像到“ \ n”。

```
#define CYG_TTY_IN_FLAGS_CRLF 0x0002 //Map '\n\r' => '\n' on input
```

如果 tty _ in _ flags 中的该位为 1,则字符“ \ n \ r”(通常由 DOS/Windows 主机发出) 将被映像到“ \ n”。

```
#define CYG_TTY_IN_FLAGS_BINARY 0x0004 //No input processing
```

如果 tty _ in _ flags 中该位为 1,则在输入数据放置到用户 buffer 之前不会对这些数据做任何处理。

```
#define CYG_TTY_IN_FLAGS_ECHO 0x0008 //Echo characters as processed
```

如果 tty _ in _ flags 中该位为 1,则字符在被处理时被回送到串口。

2. TTY 驱动程序 API 函数

TTY 驱动程序提供了一些 API 函数,应用程序可以通过使用这些函数对 TTY 设备进行操作。

```
cyg_io_read (handle, buf, len)
```

该函数用于从设备读取数据。在默认情况下,读取数据直到出现行结束符“ \ n”或“ \ r”。此外,输入字符被回送到终端设备。该函数还支持有限的输入编辑。

值得注意的是当使用 GDB 连接目标平台时,在 GDB 处于连接状态时不可能提供控制台输入功能。GDB 远程协议不支持输入,如果需要这种输入功能,则必须使 GDB 脱连。

```
cyg_io_write (handle, buf, len)
```

该函数用于向设备发送数据。默认情况下,行结束符“ \ n”被“ \ n \ r”替代。

```
cyg_io_get_config (handle, key, buf, len)
```

该函数用于读取由 key 指定的运行时通道配置信息。参数 key 指定信息类型,key 值和相应的参数 buf 类型以及函数功能说明如下:

key 值:CYG _ IO _ GET _ CONFIG _ TTY _ INFO

buf 类型:

```
cyg_tty_info_t
```

函数功能:

该函数读取当前驱动程序的状态。

Serial 串口驱动程序的 key 值也可以在此指定。在这种情况下,这种调用被直接传递给 serial 串口驱动程序。

```
cyg_io_set_config (handle, key, buf, len)
```

该函数用于实时修改通道运行时的配置信息。其 key 值指定将要被修改的配置信息。key 值和相应的参数 buf 类型以及函数功能说明如下:

key 值:CYG _ IO _ SET _ CONFIG _ TTY _ INFO

buf 类型：

cyg _ tty _ info _ t

函数功能：

该函数改变 TTY 驱动程序的当前状态。

Serial 串口驱动程序的 key 值也可以在此指定。在这种情况下，这种调用被直接传递给 serial 串口驱动程序。

8.5 PCI 库

PCI 库是 eCos 的一个可选组件，只有在具有 PCI 总线的平台中才可以使用 PCI 库。PCI 库提供了许多与 PCI 总线相关的函数，主要实现如下功能：

① 扫描 PCI 总线，查找指定的设备或某种类型的设备。

② 读取和修改普通 PCI 配置信息。

③ 读取和修改设备专用 PCI 配置信息。

④ 给 PCI 设备分配 Memory 空间和 I/O 空间。

⑤ 将设备 PCI 中断转换成等同的 HAL 中断向量。

PCI 总线支持几种地址空间，即 memory 空间、I/O 空间和配置空间。所有的 PCI 设备必须具有强制性的配置空间寄存器。一些设备可能具有 I/O 映像资源，也可能具有 memory 映像资源，也可能同时具有这两种映像资源。总线上的设备在被使用之前，必须对它们进行配置。这种配置过程通常是首先对 PCI 设备进行 PCI I/O 空间和 memory 空间的分配，然后再对该设备进行使能。所有的 PCI 设备在整个配置空间中有一个惟一的地址，这种地址由总线号、设备号以及功能号组成。PCI 桥是一种特殊的 PCI 设备，它连接两条不同的 PCI 总线。在 PCI 标准中，最多可支持 255 条 PCI 总线，每条总线最多可支持 32 个设备，每一设备最多可支持 8 个功能。

在 eCos 开发环境中，应该根据具体的目标平台特性来确定 eCos 是否以及如何对 PCI 总线上的设备进行配置。如果平台是单条 PCI 总线上的 Host，则其 PCI 设备的配置工作可以独立于相应的设备驱动程序而单独进行。如果平台不是主 Host，比如是插在 PC 机上的一块 PCI 卡，那么 PCI 设备的配置工作可以让 PC BIOS 来进行。如果涉及到 PCI－PCI 桥，那么所有 PCI 设备的配置最好放在引导过程中完成，这是因为在对 PCI 桥进行配置之前必须知道其另一条 PCI 总线上的所有设备所需的 I/O 空间和 memory 空间。

8.5.1 PCI 总线操作

eCos 的 PCI 库提供了一些用于对 PCI 总线进行操作的函数，这些操作包括对 PCI 总线的初始化操作、PCI 设备的查找、PCI 配置空间的读写、PCI 设备 I/O 空间和 memory 空间的分配、设备中断的处理、PCI 设备的启动等。头文件 cyg/io/pci.h 包含了对这些函数的声称和定义，使用 PCI 库函数的程序应该包含该头文件。eCos 提供的源码中还包含了一个 PCI 测试程序 io/pci/<release>/tests/pci1.c，这一测试程序对上述所有的 PCI 总线操作进行测试。

1. PCI 总线的初始化

在使用 PCI 总线之前必须对其进行初始化。系统中这种初始化操作一般只需进行一次。

在某些平台中,这种初始化包含在硬件抽象层 HAL 中,在对平台进行初始化的时候完成对 PCI 的初始化操作。而另外一些平台的 HAL 不对 PCI 总线进行初始化,PCI 总线的初始化由应用程序或设备驱动程序完成。

对 PCI 总线的初始化可以使用下面的函数完成:

```
void cyg_pci_init(void)
```

该函数只对 PCI 总线进行惟一的一次初始化,因此对于多个调用该函数的驱动程序来讲,对该函数的多次调用都是安全的。

2. 查找 PCI 设备

对 PCI 总线的初始化操作完成之后,可能需要查找 PCI 设备。这种查找操作可以通过下面的函数来实现:

```
cyg_bool cyg_pci_find_next(cyg_pci_device_id cur_devid,
                           cyg_pci_device_id *next_devid)
```

该函数从 cur_devid 处开始从 PCI 总线上查找设备。如果发现一个设备,则将该设备 ID 号 devid 保存到 next_devid,并返回 true。

在 PCI 测试程序 pci1.c 中有一个 pci_scan() 函数用于列举所有的 PCI 设备,它使用了一个循环体对所有 PCI 设备进行查找,该段程序简单描述如下:

```
cyg_pci_init();
if (cyg_pci_find_next(CYG_PCI_NULL_DEVID, &devid)) {
  do {
      <use devid>
  } while (cyg_pci_find_next(devid, &devid));
}
```

程序首先调用 cyg_pci_init() 函数对 PCI 总线进行初始化,然后使用 cyg_pci_find_next(CYG_PCI_NULL_DEVID, &devid) 语句开始进行 PCI 设备的查找工作。CYG_PCI_NULL_DEVID 启动 cyg_pci_find_next() 函数对 PCI 设备的查找过程。如果总线上没有 PCI 设备,则返回 false。如果返回 true,则使用所找到的 devid 进入循环体。循环体内对 devid 的使用完成后,开始查找下一个 PCI 设备。这样,当所有 PCI 设备全部查找完后,程序将退出循环体。

这是一种对 PCI 设备进行查找的常用方法,它对 PCI 总线上的设备进行枚举。PCI 库还提供了其他的一些函数用于对指定设备的查找,这些函数可以根据指定的设备类型 (如 SCSI 控制器等) 或指定的设备 vendor 对 PCI 设备进行查找。这些函数是:

查找指定类型的设备:

```
cyg_bool cyg_pci_find_class(cyg_uint32 dev_class,
                            cyg_pci_device_id *devid)
```

查找指定 vendor 的设备:

```
cyg_bool cyg_pci_find_device(cyg_uint16 vendor, cyg_uint16 device,
                             cyg_pci_device_id *devid)
```

对这些函数的使用方法与 cyg _ pci _ find _ next () 函数的使用方法基本相同。不同之处在于这些函数只有在找到满足指定条件 (dev _ class、vedio/device qualifiers) 时才返回 true。参数 devid 既用于函数的输入参数,又用于函数的返回结果。设备的查找过程从给定的 devid 处开始,当找到设备时,返回的 devid 为新找到的设备 devid。

在 eCos 源码中,头文件 pci.h 包含了另一个头文件 cyg/io/pci _ cfg.h,它包含了可用于这些函数的 PCI 设备类型 (PCI class)、厂商 (vendor) 和设备码 (device code) 等信息。当然这不是一个完整的表,在进行 eCos 开发时可以根据实际需要增加新的内容。

3. PCI 配置信息

PCI 配置信息使用了数据结构 cyg _ pci _ device,头文件 pci.h 对其进行了定义。在对 PCI 设备配置信息进行读写操作时,需要使用该数据结构。这一数据结构保存了 PCI 的配置信息。下面是该数据结构的具体定义:

```
typedef struct                          //PCI device data
{
cyg _ pci _ device _ id devid;           //ID of this device
                                        //The following fields are read out of the config space
for this device.
Cyg _ uint16 vendor;                    //vendor ID
cyg _ uint16 device;                    //device ID
cyg _ uint16 command;                   //command register
cyg _ uint16 status;                    //status register
cyg _ uint32 class _ rev;               //class + revision
cyg _ uint8 cache _ line _ size;        //cache line size
cyg _ uint8 latency _ timer;            //latency timer
cyg _ pci _ header _ type header _ type; //header type
cyg _ uint8 bist;                       //Built-in Self-Test
cyg _ uint32 base _ address[6];         //Memory base address registers
//The following fields are used by the resource allocation
//routines to keep track of allocated resources.
Cyg _ uint32 num _ bars;
cyg _ uint32 base _ size[6];            //Memory size for each base address
cyg _ uint32 base _ map[6];             //Physical address mapped
CYG _ ADDRWORD hal _ vector;            //HAL interrupt vector used by
                                        //device if int _ line != 0
//One of the following unions will be filled in according to
//the value of the header _ type field.
Union
{
    struct
    {
    cyg _ uint32 cardbus _ cis;         //CardBus CIS Pointer
    cyg _ uint16 sub _ vendor;          //subsystem vendor id
```

```
        cyg_uint16 sub_id;                  //subsystem id
        cyg_uint32 rom_address;             //ROM address register
        cyg_uint8 cap_list;                 //capability list
        cyg_uint8 reserved1[7];
        cyg_uint8 int_line;                 //interrupt line
        cyg_uint8 int_pin;                  //interrupt pin
        cyg_uint8 min_gnt;                  //timeslice request
        cyg_uint8 max_lat;                  //priority-level request
      } normal;
      struct
       {
    cyg_uint8 pri_bus;                      //primary bus number
    cyg_uint8 sec_bus;                      //secondary bus number
    cyg_uint8 sub_bus;                      //subordinate bus number
    cyg_uint8 sec_latency_timer;            //secondary bus latency
    cyg_uint8 io_base;
    cyg_uint8 io_limit;
    cyg_uint16 sec_status;                  //secondary bus status
    cyg_uint16 mem_base;
    cyg_uint16 mem_limit;
    cyg_uint16 prefetch_base;
    cyg_uint16 prefetch_limit;
    cyg_uint32 prefetch_base_upper32;
    cyg_uint32 prefetch_limit_upper32;
    cyg_uint16 io_base_upper16;
    cyg_uint16 io_limit_upper16;
        cyg_uint8 reserved1[4];
        cyg_uint32 rom_address;             //ROM address register
        cyg_uint8 int_line;                 //interrupt line
        cyg_uint8 int_pin;                  //interrupt pin
        cyg_uint16 control;                 //bridge control
      } bridge;
    struct
        {
        //Not yet supported
      } cardbus_bridge;
  } header;
} cyg_pci_device;
```

当查找到 PCI 设备后，可以使用下面的函数对 PCI 的配置空间进行读写操作：

```
void cyg_pci_get_device_info(cyg_pci_device_id devid,
                             cyg_pci_device * dev_info)
void cyg_pci_set_device_info(cyg_pci_device_id devid,
```

```
cyg_pci_device * dev_info)
```

在测试程序 pci1.c 内，使用了下面的程序代码对 PCI 设备配置空间进行操作：

```
//Get device info
cyg_pci_get_device_info(devid, &dev_info);
//Print stuff
diag_printf("Found device on bus %d, devfn 0x%02x:\n",
            CYG_PCI_DEV_GET_BUS(devid),
            CYG_PCI_DEV_GET_DEVFN(devid));
diag_printf("\n Command 0x%04x, Status 0x%04x\n",
            dev_info.command, dev_info.status);
```

可以对 PCI 配置空间的命令寄存器进行写操作，用以控制该设备是否响应来自总线的 IO 访问或 memory 访问。

上面的函数只能对普通的 PCI 配置寄存器进行操作。某些 PCI 设备可能还有其他一些 PCI 标准中没有说明的专用配置寄存器。对这些专用配置寄存器的访问可以使用下述函数来实现：

```
void cyg_pci_read_config_uint8(cyg_pci_device_id devid,
                               cyg_uint8 offset, cyg_uint8 *val)
void cyg_pci_read_config_uint16(cyg_pci_device_id devid,
                               cyg_uint8 offset, cyg_uint16 *val)
void cyg_pci_read_config_uint32(cyg_pci_device_id devid,
                               cyg_uint8 offset, cyg_uint32 *val)
void cyg_pci_write_config_uint8(cyg_pci_device_id devid,
                               cyg_uint8 offset, cyg_uint8 val)
void cyg_pci_write_config_uint16(cyg_pci_device_id devid,
                               cyg_uint8 offset, cyg_uint16 val)
void cyg_pci_write_config_uint32(cyg_pci_device_id devid,
                               cyg_uint8 offset, cyg_uint32 val)
```

必须注意，在使用这些函数对配置寄存器进行写操作时，这种写操作只能针对专用配置寄存器，不能对普通配置寄存器进行写操作。否则将会引起先前已经读取的存放在 cyg_pci_device 数据结构内的配置信息失效。

4. 内存分配

PCI 设备在启动之前不会响应任何 I/O 访问和 Memory 访问，PCI 设备的启动必须在其配置操作完成之后进行。对 PCI 设备的配置操作实际上是告诉设备其 IO 资源和 Memory 资源被映射到什么地方，这种配置操作可以使用下面的函数进行：

```
cyg_bool cyg_pci_configure_device(cyg_pci_device *dev_info)
cyg_bool cyg_pci_configure_bus(cyg_uint8 bus, cyg_uint8 *next_bus)
```

对于非桥 PCI 设备的配置，全部使用 cyg_pci_configure_device()函数来处理所有的 IO 和 Memory 空间配置。当系统平台使用 PCI 桥连接多条 PCI 总线时，应该使用 cyg_pci_con-

figure _ bus () 函数进行配置，这一函数将递归配置指定总线 (参数 bus) 上的所有设备和所有从属总线。cyg _ pci _ configure _ bus () 将使用 cyg _ pci _ configure _ device () 对单个非桥设备进行配置。

分配给 PCI 设备的每个区分别使用其配置空间的基地址寄存器 BAR 来表示，这种空间的分配将根据空间的类型分别使用下面的函数进行：

① Memory 空间的分配：

```
cyg _ bool cyg _ pci _ allocate _ memory (cyg _ pci _ device * dev _ info,
                                         cyg _ uint32 bar,
                                         CYG _ PCI _ ADDRESS64 * base)
```

② IO 空间的分配：

```
cyg _ bool cyg _ pci _ allocate _ io (cyg _ pci _ device * dev _ info,
                                     cyg _ uint32 bar,
                                     CYG _ PCI _ ADDRESS32 * base)
```

在对 PCI 设备进行空间分配时，基地址 (IO 基地址和 Memory 基地址) 将依次增加。如果基地址超出了地址空间的限制，将会引起空间分配的失败。IO 空间地址的上限是 1MB，Memory 空间地址的上限是 2^{32}B 或 2^{64}B。

应用程序和驱动程序在需要的时候可以直接调用上述这些函数。基地址在初始化时被设置为 HAL 所提供的默认值。如果应用程序希望改变这些基地址，可以使用下述函数：

```
void cyg _ pci _ set _ memory _ base (CYG _ PCI _ ADDRESS64 base)
void cyg _ pci _ set _ io _ base (CYG _ PCI _ ADDRESS32 base)
```

在完成对 PCI 设备的配置操作后，cyg _ pci _ device 数据结构中的基地址将是属于 CPU 地址空间的物理地址，CPU 可以对其进行访问。对于 PCI 设备的每一个基地址寄存器 BAR，该数据结构中均采用一个 32 位的 base _ map [] 来表示。对于 32 位 PCI 的内存空间，每一个 base _ map [] 都代表一个实际的指针，驱动程序可以直接使用这些指针。对于 64 位 PCI 的内存空间，则有可能会超出 CPU 的地址空间。这种情况下，要求驱动程序知道如何分段访问这些空间。

5. 中断的处理

有些 PCI 设备可能会产生中断，对中断进行处理时必须知道系统为该设备所分配的中断向量。对于给定的设备，硬件抽象层 HAL 中与其相对应的中断向量在不同的系统平台中会有所不同。驱动程序可以使用下面的函数获取给定设备的实际中断向量：

```
cyg _ bool cyg _ pci _ translate _ interrupt (cyg _ pci _ device * dev _ info,
                                             CYG _ ADDRWORD * vec)
```

调用该函数时如果返回 false，说明该设备不会产生中断。如果返回 true，该设备所使用的 HAL 中断向量将由指针 vec 带回。下面是 pci1.c 测试程序中使用该函数的例子：

```
if (cyg _ pci _ translate _ interrupt (&dev _ info, &irq))
    diag _ printf ("Wired to HAL vector %d \n", irq);
else
```

```
diag _ printf (" Does not generate interrupts. \ n") ;
```

应用程序和驱动程序在启动一个 PCI 设备之前必须将中断处理程序和设备中断进行连接。

6. PCI 设备的启动

PCI 设备在被启动之前,不会响应任何 IO 和 Memory 访问请求。在配置操作完成后,必须对其进行启动操作。驱动程序在完成对设备的初始化操作后,可以通过使能 PCI 配置寄存器中的命令寄存器的使能标志对其进行启动。下面是 pci1.c 测试程序对 PCI 设备进行启动的例子:

```
#ifdef ENABLE _ PCI _ DEVICES
{
cyg _ uint16 cmd;
//Don't use cyg _ pci _ set _ device _ info since it clears
//some of the fields we want to print out below.
cyg _ pci _ read _ config _uint16 (dev _ info.devid,
                                  CYG _ PCI _ CFG _ COMMAND, &cmd) ;
cmd | = CYG _ PCI _ CFG _ COMMAND _ IO|CYG _ PCI _ CFG _ COMMAND _ MEMORY;
cyg _ pci _ write _ config _ uint16 (dev _ info.devid,
                                  CYG _ PCI _ CFG _ COMMAND, cmd) ;
}
diag _ printf (" * * * * Device IO and MEM access enabled \ n") ;
#endif
```

该例使用了 cyg _ pci _ write _ config _ uint16 () 函数对命令寄存器进行写操作,使能对 IO 空间和 Memory 空间的访问,从而完成了该 PCI 设备的启动操作。值得注意的是,启动 PCI 设备的最好办法是通过 cyg _ pci _ set _ device _ info () 函数来进行。上例是一种特殊情况,这是由于该测试程序需要输出 PCI 设备被启动前的 cyg _ pci _ device 数据结构内的信息,如果使用 cyg _ pci _ set _ device _ info () 函数,在调用该函数时该数据结构的内容已经被修改,输出的配置信息将是被修改后的内容。

8.5.2 PCI 库 API

PCI 库为 PCI 总线配置空间的访问提供了一组使用非常方便的函数。它提供两种类型的 API 函数:高级 API 函数和低级 API 函数。高级 API 主要供设备驱动程序和其他程序访问 PCI 配置空间。低级 API 主要供 PCI 库自己对硬件进行访问,也可以为设备驱动程序提供一种直接访问 PCI 配置空间的方法。在低级 API 之下是硬件抽象层 HAL,它对配置空间的基本操作提供支持。这种支持只针对 PCI 库,其他程序代码通常不使用这种支持。

1. 数据结构与类型定义

PCI 库所提供的 API 函数和数据类型都在头文件 cyg/io/pci.h 中有定义,该头文件中包含了对 PCI 配置空间数据结构的定义和设备名、厂家名及类型代码。下面介绍一些典型的数据类型。

设备 ID 数据类型:

```
typedef CYG_WORD32 cyg_pci_device_id;
```

cyg_pci_device_id 是一个 32 位的数据类型，包含了总线号（bus number）、设备号（device number）和功能号（function number）。两个宏定义 CYG_PCI_DEV_MAKE_ID() 和 CYG_PCI_DEV_MAKE_DEVFN() 可以用来将总线号、设备号和功能号组合成一个设备 ID。另外，eCos 还定义了四个宏定义，可以从设备 ID 中提取出总线号、设备号和功能号。这四个宏定义分别是：

```
CYG_PCI_DEV_GET_BUS()
CYG_PCI_DEV_GET_DEVFN()
CYG_PCI_DEV_GET_DEV()
CYG_PCI_DEV_GET_FN()
```

正常情况下，一般不会用到这些宏定义。下面是使用这些宏定义的一个例子：

```
//Create a packed representation of device 1, function 0
  cyg_uint8 devfn = CYG_PCI_DEV_MAKE_DEVFN(1,0);
//Create a packed devid for that device on bus 2
  cyg_pci_device_id devid = CYG_PCI_DEV_MAKE_ID(2, devfn);
  diag_printf("bus %d, dev %d, func %d\n",
              CYG_PCI_DEV_GET_BUS(devid),
              CYG_PCI_DEV_GET_DEV(CYG_PCI_DEV_GET_DEVFN(devid)),
              CYG_PCI_DEV_GET_FN(CYG_PCI_DEV_GET_DEVFN(devid)
);
```

PCI 配置信息数据结构：

```
typedef struct cyg_pci_device;
```

上一节已经对该数据结构进行了介绍。它包含了从 PCI 设备配置空间读取的数据，也用于记录对该设备的资源分配信息。PCI 配置信息可以使用 cyg_pci_get_device_info() 函数读取。

PCI 地址空间类型：

```
typedef CYG_WORD64 CYG_PCI_ADDRESS64;
typedef CYG_WORD32 CYG_PCI_ADDRESS32;
```

这两种类型为 PCI 的地址提供了定义。PCI 的 IO 地址空间指针为 32 位，Memory 空间地址指针为 32 位或 64 位。

2. 高级 API

PCI 库为应用程序和驱动程序提供了一组高级 API 函数，使用这些函数可以对 PCI 总线及 PCI 设备进行操作。下面简单介绍这些高级 API 函数。

```
void cyg_pci_init(void)
```

该函数对 PCI 库进行初始化，并与硬件建立联系。系统内所有驱动程序都可以调用该函数，也可以仅由应用程序的初始化函数进行调用。

```
cyg_bool cyg_pci_find_device(cyg_uint16 vendor,
                             cyg_uint16 device,
                             cyg_pci_device_id *devid)
```

该函数查找符合指定的厂家号 vendor 和设备号 (device) 的设备。这种查找从 devid 指定的设备处开始 (若 devid 为 CYG_PCI_NULL_DEVID 则从第一个 PCI 槽开始),如果发现了相应的设备,则其设备 ID 将通过 devid 带回。发现相应的设备时,函数返回 true,否则返回 false。

```
cyg_bool cyg_pci_find_class(cyg_uint32 dev_class,
                            cyg_pci_device_id *devid)
```

该函数查找给定设备类型代码 dev_class 的设备。查找过程开始于 devid 指定的设备 (如果 devid 为 CYG_PCI_NULL_DEVID,则从 PCI 的第一个槽开始)。如果查找到相应的设备,则返回 true 且 devid 包含被查找到的设备 ID,否则返回 false。

```
cyg_bool cyg_pci_find_next(cyg_pci_device_id cur_devid,
                           cyg_pci_device_id *next_devid)
```

该函数查找 cur_devid 指定设备之后的有效设备。如果 cur_devid 为 CYG_PCI_NULL_DEVID,则从第一个 PCI 槽开始查找。该函数允许 next_devid 指向 cur_devid。如果查找到另一个设备,函数返回 true,否则返回 false。

```
cyg_bool cyg_pci_find_matching(cyg_pci_match_func *matchp,
                               void *match_callback_data,
                               cyg_pci_device_id *devid)
```

该函数查找设备属性与匹配函数 matchp 相匹配的设备。查找过程开始于 devid 指定的设备,如果 devid 为 CYG_PCI_NULL_DEVID,则从第一个 PCI 槽开始查找。如果查找到相匹配的设备,则返回 true 且 devid 包含被查找到的设备 ID,否则返回 false。匹配函数的类型声称如下:

```
typedef cyg_bool (cyg_pci_match_func)(cyg_uint16 vendor,
                                      cyg_uint16 device,
                                      cyg_uint32 class,
                                      void *user_data)
```

其中的 vendor、device 和 class 三个参数来自 PCI 设备配置空间,user_data 为传递给 cyg_pci_find_matching 函数的回调数据 (即 match_callback_data)。

```
void cyg_pci_get_device_info(cyg_pci_device_id devid,
                             cyg_pci_device *dev_info)
```

该函数获取 devid 所指定设备的 PCI 配置信息。配置信息存放于 dev_info 指定的 cyg_pci_device 数据结构内。如果该设备没有被使能,则该函数将从基地址寄存器取回其大小和类型信息,并放置于 cyg_pci_device 中的 base_size[] 数组内。

```
void cyg_pci_set_device_info(cyg_pci_device_id devid,
```

```
                                    cyg _ pci _ device * dev _ info)
```

该函数对 devid 指定的设备进行 PCI 配置信息的设置。这种设置只对可写配置寄存器进行设置。一旦所有信息设置完成,设备的配置信息将被读回到 dev _ info,dev _ info 反映了硬件的真实状态。

```
void cyg _ pci _ read _ config _ uint8 (cyg _ pci _ device _ id devid,
                                    cyg _ uint8 offset, cyg _ uint8 * val)
void cyg _ pci _ read _ config _ uint16 (cyg _ pci _ device _ id devid,
                                    cyg _ uint8 offset, cyg _ uint16 * val)
void cyg _ pci _ read _ config _ uint32 (cyg _ pci _ device _ id devid,
                                    cyg _ uint8 offset, cyg _ uint32 * val)
```

这三个函数用于读取指定设备 devid 的配置寄存器。它们主要用于读取设备专用配置寄存器。对普通 PCI 配置寄存器的访问最好通过 cyg _ pci _ get _ device _ info () 函数进行。

```
void cyg _ pci _ write _ config _ uint8 (cyg _ pci _ device _ id devid,
                                    cyg _ uint8 offset, cyg _ uint8 val);
void cyg _ pci _ write _ config _ uint16 (cyg _ pci _ device _ id devid,
                                    cyg _ uint8 offset, cyg _ uint16 val);
void cyg _ pci _ write _ config _ uint32 (cyg _ pci _ device _ id devid,
                                    cyg _ uint8 offset, cyg _ uint32 val);
```

这三个函数用于写指定设备 devid 的 PCI 配置寄存器。主要用于访问设备专用配置寄存器。对普通 PCI 配置寄存器的访问最好通过 cyg _ pci _ set _ device _ info () 函数进行。如果使用这三个函数对普通 PCI 配置寄存器进行写操作,则有可能导致 cyg _ pci _ device 内的数据失效。

利用上述 API 函数可以进行 PCI 总线的初始化操作,并可以对 PCI 配置空间进行访问。另外还有下面的一些 API 函数,这些函数可以被用来对 PCI 设备进行资源分配:

```
cyg _ bool cyg _ pci _ configure _ device (cyg _ pci _ device * dev _ info)
```

该函数对 PCI 设备的所有基地址寄存器进行设置,为 PCI 设备分配 Memory 和 IO 空间。所设置的基地址存放在 * dev _ info 内的 base _ map [] 内,可以直接使用。如果 * dev _ info 没有有效的 base _ size [],函数结果将返回 false。该函数还调用了 cyg _ pci _ translate _ interrupt () 函数将 PCI 设备中断转换成 HAL 中断向量。

```
cyg _ bool cyg _ pci _ configure _ bus (cyg _ uint8 bus,
                                    cyg _ uint8 * next _ bus)
```

该函数对指定总线及其从属总线上所有设备的所有基地址寄存器进行设置,为 PCI 设备分配 Memory 和 IO 空间。如果总线上有 PCI-PCI 桥,该函数将对自己进行递归调用,完成对该桥另一侧总线的配置。在对桥设备的 Memory 和 IO 窗口进行配置之前,必须完成桥设备另一侧总线上所有设备的 Memory 和 IO 空间分配。参数 next _ bus 指向分配给下一条子总线的总线号,总线号将随着新总线的发现而增加。函数执行成功返回 true,否则返回 false。

```
cyg _ bool cyg _ pci _ translate _ interrupt (cyg _ pci _ device * dev _ info,
```

```
                                        CYG_ADDRWORD *vec)
```

该函数将设备的 PCI 中断(INTA#～INTAD#)转换成相应的 HAL 中断向量。这种转换与 PCI 设备在总线上的位置相关。如果设备可以产生中断,则被转换的中断向量号将被存放在 vec 内,并返回 true,否则返回 false。

```
cyg_bool cyg_pci_allocate_memory(cyg_pci_device *dev_info,
                                 cyg_uint32 bar,
                                 CYG_PCI_ADDRESS64 *base)
cyg_bool cyg_pci_allocate_io(cyg_pci_device *dev_info,
                             cyg_uint32 bar,
                             CYG_PCI_ADDRESS32 *base)
```

上面两个函数对 bar 指定的基地址寄存器进行设置,对 PCI 设备分配 Memory 和 IO 空间。参数 *base 必须具有正确的字节边界,如果空间分配成功,函数返回时该参数将带回下一个未被分配的空间地址。如果基地址寄存器被分配一个错误类型的地址,或者 dev_info 所包含的不是有效的 base_size[],则返回结果为 false。这些函数允许设备驱动程序建立自己所需的地址映像。大多数设备应该使用 cyg_pci_configure_device()函数来实现这种地址空间的分配。

```
void cyg_pci_set_memory_base(CYG_PCI_ADDRESS64 base)
void cyg_pci_set_io_base(CYG_PCI_ADDRESS32 base)
```

这两个函数分别对内存分配程序所使用的 Memory 和 IO 映像的基地址进行设置。正常情况下这些基地址使用系统平台的默认基地址。应用程序可根据需要使用这两个函数来改变这些基地址的设置。

3. 低级 API

PCI 库提供了一些硬件 API 函数,主要为 PCI 库自己提供访问 PCI 总线配置空间的方法。驱动程序和应用程序也可以使用这些函数来实现 PCI 库不支持的一些 PCI 总线操作。下面是这些低级 API 函数的简单介绍。

```
void cyg_pcihw_init(void)
```

该函数对 PCI 硬件进行初始化,使得 PCI 配置空间可以被访问。

```
void cyg_pcihw_read_config_uint8(cyg_uint8 bus,
                    cyg_uint8 devfn, cyg_uint8 offset, cyg_uint8 *val)
void cyg_pcihw_read_config_uint16(cyg_uint8 bus,
                    cyg_uint8 devfn, cyg_uint8 offset, cyg_uint16 *val)
void cyg_pcihw_read_config_uint32(cyg_uint8 bus,
                    cyg_uint8 devfn, cyg_uint8 offset, cyg_uint32 *val)
```

这些函数分别读取 PCI 配置空间内的不同长度的寄存器,寄存器地址由参数 bus、devfn、offset 指定。

```
void cyg_pcihw_write_config_uint8(cyg_uint8 bus,
                    cyg_uint8 devfn, cyg_uint8 offset, cyg_uint8 val)
```

```
void cyg_pcihw_write_config_uint16(cyg_uint8 bus,
                    cyg_uint8 devfn, cyg_uint8 offset, cyg_uint16 val)
void cyg_pcihw_write_config_uint32(cyg_uint8 bus,
                    cyg_uint8 devfn, cyg_uint8 offset, cyg_uint32 val)
```

这些函数分别对 PCI 配置空间内的不同长度的寄存器进行设置，寄存器地址由参数 bus、devfn、offset 指定。

```
cyg_bool cyg_pcihw_translate_interrupt(cyg_uint8 bus,
                                       cyg_uint8 devfn,
                                       CYG_ADDRWORD *vec)
```

该函数询问 PCI 设备哪一个 HAL 中断向量与其相对应。

4. 硬件抽象层 HAL 对 PCI 的支持

硬件抽象层 HAL 对 PCI 的支持由一组 C 语言宏定义组成，这些宏为 PCI 低级 API 提供了具体实现。下面简单介绍这些宏定义。

```
HAL_PCI_INIT()
```

初始化 PCI 总线。

```
HAL_PCI_READ_UINT8(bus, devfn, offset, val)
HAL_PCI_READ_UINT16(bus, devfn, offset, val)
HAL_PCI_READ_UINT32(bus, devfn, offset, val)
```

这三个宏定义分别读取 PCI 配置空间内的不同长度的寄存器，寄存器地址由参数 bus、devfn、offset 指定。

```
HAL_PCI_WRITE_UINT8(bus, devfn, offset, val)
HAL_PCI_WRITE_UINT16(bus, devfn, offset, val)
HAL_PCI_WRITE_UINT32(bus, devfn, offset, val)
```

这三个宏定义分别对 PCI 配置空间内的不同长度的寄存器进行设置，寄存器地址由参数 bus、devfn、offset 指定。

```
HAL_PCI_TRANSLATE_INTERRUPT(bus, devfn, *vec, valid)
```

将 PCI 设备中断转换成 HAL 中断向量。

```
HAL_PCI_ALLOC_BASE_MEMORY
HAL_PCI_ALLOC_BASE_IO
```

这两个宏定义了用于对 Memory 和 IO 分配指针进行初始化的默认基地址。

```
HAL_PCI_PHYSICAL_MEMORY_BASE
HAL_PCI_PHYSICAL_IO_BASE
```

PCI 的 Memory 空间和 IO 空间并不总是直接使用物理 Memory 地址和物理 IO 地址。这些地址空间常常是 CPU 地址空间内某一 offset 处的一处窗口。这两个宏对这种 offset 进行了定义，该 offset 加上 PCI 基地址将 PCI 总线地址转换成物理内存地址，使用这种转换后的物理

地址可以访问分配给 PCI 设备的 Memory 和 IO 空间。使用这种 CPU 地址窗口机制给 PCI 设备分配的可直接寻址的内存空间要比实际可提供给 PCI 的内存空间要小。在这种情况下，设备驱动程序必须使用段（segment）对 PCI 内存空间进行访问。

HAL _ PCI _ IGNORE _ DEVICE（bus，dev，fn）

如果对该宏进行了定义，那么在使用 PCI 总线扫描函数查找设备时，设备的指定将会受到限制。这是因为某些时候需要对设备进行特殊处理。如果该宏的值为 true，则在使用 cyg _ pci _ find _ next（）函数或其他总线扫描函数查找设备时，将不会发现参数 bus、dev、fn 所指定的设备。

第9章 文件系统

本章主要介绍 eCos 文件系统的基本结构和基本的文件 IO 操作。eCos 文件系统主要实现于 FILEIO 包，该包提供了与 POSIX 兼容的文件 IO 操作，另外还提供了 BSD Socket 网络 API。本章的侧重点主要在于文件系统和基于文件系统的网络协议栈的客户端接口。如果读者想详细了解这些 API 的具体说明，可以参阅其他相关标准和说明。

eCos 文件系统并不局限于真正意义上的文件。通过文件系统接口所访问的对象可以是网络协议 Sockets、设备驱动程序、FIFO、消息队列，或者其他任何具有类文件接口的对象。以设备为例，设备的访问可以通过一个安装 (mount) 于“/dev”的伪文件系统“devfs”来实现。设备的打开操作被转换为对 cyg _ io _ lookup () 函数的调用，如果调用成功，则其文件对象中 f _ ops 内的函数将文件系统的 API 函数转换为对设备 API 的调用。

eCos 提供的 FILEIO 源码中包含了一组包含许多指针的表格，这些指针指向文件系统的主要接口函数。这种方法可以避免函数名字的混淆 (例如在多个文件系统中可能都有一个相同的 read () 函数)。同时，采用这种方法还可以减少动态内存分配的需要。

用户在进行开发时，可以创建新的文件系统。通过将 FILEIO 的函数调用转换为现有文件系统的调用，可以很容易地将其他系统现有的文件系统移植到 eCos 中来，而且这种移植的代码量很小。

本章还介绍了 RAM 文件系统和 ROM 文件系统。

9.1 文件系统表格

eCos 文件系统包含了一些文件系统表格和文件操作函数。这些表格有文件系统表 (File System Table)、安装表 (Mount Table) 和文件表 (File Table)。文件系统表用于描述具体 eCos 系统中所有的文件系统以及对每个文件系统进行操作的一些函数信息，安装表描述了系统中处于活跃状态的文件系统，而文件表所描述的是被打开的文件对象的一些详细信息。

9.1.1 文件系统表 (File System Table)

文件系统表是一个数组形式的表格，数组的每一项对应于系统中的每一个文件系统，它们分别对相应的文件系统进行描述。系统中的每一个文件系统都应当使用宏 FSTAB _ ENTRY () 在文件系统表中注册一个相应的表项。目前版本的 eCos 不支持对文件系统表项的动态增加或删除，但提供了一个可以向文件系统表增加新表项的一个类似于 mount () 的 API 函数。

文件系统表用于存放已安装在系统上的所有文件系统的入口，它采用如下的数组形式表示：

```
_ externC cyg _ fstab _ entry fstab [] ;
```

文件系统表中的每一个表项对应一个文件系统。表项是一个数据结构，其结构定义如下：

```
struct cyg_fstab_entry
{
    const char          *name;          //filesystem name
    CYG_ADDRWORD        data;           //private data value
    cyg_uint32          syncmode;       //synchronization mode

    int (*mount) (cyg_fstab_entry *fste, cyg_mtab_entry *mte);
    int (*umount)(cyg_mtab_entry *mte);
    int (*open)  (cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                  int mode, cyg_file *fte);
    int (*unlink)(cyg_mtab_entry *mte, cyg_dir dir, const char *name);
    int (*mkdir) (cyg_mtab_entry *mte, cyg_dir dir, const char *name);
    int (*rmdir) (cyg_mtab_entry *mte, cyg_dir dir, const char *name);
    int (*rename)(cyg_mtab_entry *mte, cyg_dir dir1, const char *name1,
                  cyg_dir dir2, const char *name2);
    int (*link)  (cyg_mtab_entry *mte, cyg_dir dir1, const char *name1,
                  cyg_dir dir2, const char *name2, int type);
    int (*opendir)(cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                  cyg_file *fte);
    int (*chdir) (cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                  cyg_dir *dir_out);
    int (*stat)  (cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                  struct stat *buf);
    int (*getinfo)(cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                  int key, char *buf, int len);
    int (*setinfo)(cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                  int key, char *buf, int len);
};
```

其中：

name — **文件系统名字，如“**nomfs**”、“**msdos**”、“**ext2**”等。**

data — **文件系统私有数据。**

syncmode — **同步方式，描述该文件系统被访问时所使用的锁定协议。**

结构体内的其余部分是对该文件系统进行文件和目录操作的一些函数：

mount — **安装文件系统。**

umount — **卸载文件系统。**

open — **打开一个文件。**

unlink — **取消文件连接。**

mkdir — **创建文件目录。**

rmdir — **删除文件目录。**

link — **给文件创建一个新的连接。**

opendir — **打开一个文件目录。**

chdir — 改变当前目录。

stat — 获取文件信息。

getinfo — 获取文件系统信息。

setinfo — 设置文件系统信息。

除 mount() 函数和 umount() 函数外，其他函数都包含了三个标准参数：安装表指针、目录指针和文件名，这些参数用于对操作对象进行定位。

宏 FSTAB_ENTRY() 用于在文件系统表中注册一个文件系统。每一个驻留在系统中的文件系统都要使用宏 FSTAB_ENTRY 在文件系统表中注册一个入口，为将来安装文件系统做好准备。FSTAB_ENTRY() 的使用格式如下：

```
FSTAB_ENTRY(_l,          //文件系统表表项的 C 语言标识符
            _name,       //文件系统名字
            _data,       //文件系统私有数据
            _syncmode,   //同步方式
            _mount,      //文件系统内部 mount 函数指针
            _umount,     //文件系统内部 umount 函数指针
            _open,       //文件系统内部 open 函数指针
            _unlink,     //文件系统内部 unlink 函数指针
            _mkdir,      //文件系统内部 mkdir 函数指针
            _rmdir,      //文件系统内部 rmdir 函数指针
            _rename,     //文件系统内部 rename 函数指针
            _link,       //文件系统内部 link 函数指针
            _opendir,    //文件系统内部 opendir 函数指针
            _chdir,      //文件系统内部 chdir 函数指针
            _stat,       //文件系统内部 stat 函数指针
            _getinfo,    //文件系统内部 getinfo 函数指针
            _setinfo     //文件系统内部 setinfo 函数指针
           )
```

9.1.2 安装表 (Mount Table)

文件系统安装表记录了系统中当前处于活跃状态的文件系统。eCos 的这种安装表类似于 UNIX 系统中的文件系统安装点。

安装表表项有两个来源。文件系统和其他组件可以使用宏 MTAB_ENTRY() 向安装表输出静态表项，另一种方法是使用 mount() 函数在运行时安装新的表项。通过这两种方法安装的表项都可以调用 umount() 函数进行卸载。

系统中正在运行的文件系统安装表采用下面的数组形式表示：

```
_externC cyg_mtab_entry mtab[];
```

安装表表项也是一个数据结构，其定义如下：

```
struct cyg_mtab_entry
{
```

```
    const char          * name;     //name of mount point
    const char          * fsname;   //name of implementing filesystem
    const char          * devname;  //name of hardware device
    CYG_ADDRWORD        data;       //private data value
    cyg_bool            valid;      //valid entry?
    cyg_fstab_entry     * fs;       //pointer to fstab entry
    cyg_dir             root;       //root directory pointer
};
```

安装表中的 name 为安装点 (mount point) 名字。它用于将具有根的文件名 (以“/”开始的文件名) 指向正确的文件系统。当使用以“/”开始的文件名时,系统将其与安装表中所有有效的表项名字进行比较,表项名字在字符“/”出现之前或字符串结束之前与文件名具有最长匹配的表项即为该文件所属的文件系统安装表表项。文件名的剩余部分、该安装表表项的指针以及作为目录指针的 root 值一起被当作参数传送到文件系统表表项中的相应函数。

例如,假设一个安装表具有如下的表项内容:

```
{"/", "msdos", "/dev/hd0", ...}
{"/fd", "msdos", "/dev/fd0", ...}
{"/rom", "romfs", "", ...}
{"/tmp", "ramfs", "", ...}
{"/dev", "devfs", "", ...}
```

当试图打开文件“/tmp/foo”时,该文件被定向到 RAM 文件系统 (ramfs),而“/bar/bundy”则被定向到硬盘 MSDOS 文件系统 (msdos)。打开“/dev/tty0”的操作将被定向到设备管理文件系统 (devfs) 的设备表中的 lookup () 函数。

不带根的文件名 (不以“/”开始的文件名) 将直接定向到包含当前目录的文件系统。当前目录是由一个安装表表项和一个目录指针组合起来表示的。

安装表表项中的 fsname 为文件系统名字。它与文件系统表中的 name 相对应。在初始化过程中,系统将对安装表进行扫描,并在文件系统表中查找相应的 fsname 表项。这种匹配过程每成功一次,都将调用该文件系统的 mount 函数。如果 mount 成功,该安装表表项将被标记为有效,而且还要对表项中的 fs 进行设置,使其指向相应的文件系统。

devname 包含了文件系统所使用的设备名字。它可能与设备表中的某个表项相匹配,也可能是文件系统的一个特定的字符串 (在文件系统具有自己的内部设备驱动程序的情况下)。

data 是安装表表项的私有数据。当安装表表项属于静态定义时,data 的设置是静态的。data 的值也可以在 mount () 操作中对其进行动态设置。

valid 表示该安装点是否已被成功安装。在对名字进行匹配的搜索过程中,将忽略 valid 值为 false 的表项。

vs 指向具体的文件系统,在 mount () 操作成功后对其进行设置。

root 为目录指针。文件系统将其认为是其目录树的根 (root)。当使用带根的文件名进行文件操作时,root 将作为文件系统函数的参数 dir 传递给相应的函数。文件系统的 mount () 函数必须对 root 进行初始化。

宏 MTAB_ENTRY () 用于向安装表输出静态表项,它的使用格式如下:

```
MTAB_ENTRY(_l,          //安装表表项C语言标识符
           _name,       //安装点
           _fsname,     //文件系统名字
           _devname,    //文件系统设备名字
           _data        //安装表表项私有数据
          )
```

9.1.3 文件表

文件被打开后,将用一个打开的文件对象(file object)来表示该文件。文件对象是从一个文件对象数组(static cyg_file file[CYGNUM_FILEIO_NFILE])中分配的,这种数组就是eCos文件系统中的文件表。用户程序使用另外一个索引数组(static cyg_file *desc[CYGNUM_FILEIO_NFD])来访问这些被打开的文件对象。eCos的这种文件访问机制具有UNIX系统的文件描述符功能,拥有多种文件复制机制。

文件表表项结构定义如下:

```
struct CYG_FILE_TAG
{
    cyg_uint32              f_flag;         /* file state                 */
    cyg_uint16              f_ucount;       /* use count                  */
    cyg_uint16              f_type;         /* descriptor type            */
    cyg_uint32              f_syncmode;     /* synchronization protocol   */
    struct CYG_FILEOPS_TAG  *f_ops;         /* file operations            */
    off_t                   f_offset;       /* current offset             */
    CYG_ADDRWORD            f_data;         /* file or socket             */
    CYG_ADDRWORD            f_xops;         /* extra type specific ops    */
    cyg_mtab_entry          *f_mte;         /* mount table entry          */
};
```

文件表表项中的f_flag包含了一些FILEIO控制位(bit16~bit23),该标志的其他一些位来自open()函数的flags参数(bit0~bit15,由CYG_FILE_MODE_MASK所定义),bit24~bit31保留。

f_ucount为使用次数计数器,用于控制文件关闭的时机。文件每复制一次,该计数器加1。每进行一次文件I/O操作,该计数器也加1,这样保证了在进行当前I/O操作时文件不会被关闭。

f_type为文件对象的类型,可能的值有CYG_FILE_TYPE_FILE、CYG_FILE_TYPE_SOCKET和CYG_FILE_TYPE_DEVICE。

f_syncmode是相应的文件系统表中的syncmode的拷贝,表示该文件的同步方式。

f_offset记录当前文件的位置。对文件进行操作的函数必须对其进行更新。

f_data为文件系统的私有数据,通常是该文件所属文件系统对象的一个指针或句柄,用来指向数据结点。

f_xops是一个指针,指向文件操作函数的一些额外的数据类型。例如,Socket的I/O系

统具有一个指针，该指针指向实现标准 Socket 操作的函数表。

f_mte 是一个指向该文件所属的文件安装表的指针，主要用于同步协议的实现。

f_ops 是一个指向文件 I/O 操作函数表的指针，该表具有下面的结构：

```
struct CYG_FILEOPS_TAG
{
    int  (*fo_read)     (struct CYG_FILE_TAG *fp, struct CYG_UIO_TAG *uio);
    int  (*fo_write)    (struct CYG_FILE_TAG *fp, struct CYG_UIO_TAG *uio);
    int  (*fo_lseek)    (struct CYG_FILE_TAG *fp, off_t *pos, int whence);
    int  (*fo_ioctl)    (struct CYG_FILE_TAG *fp, CYG_ADDRWORD com,
                         CYG_ADDRWORD data);
    int  (*fo_select)   (struct CYG_FILE_TAG *fp, int which,
                         CYG_ADDRWORD info);
    int  (*fo_fsync)    (struct CYG_FILE_TAG *fp, int mode);
    int  (*fo_close)    (struct CYG_FILE_TAG *fp);
    int  (*fo_fstat)    (struct CYG_FILE_TAG *fp, struct stat *buf);
    int  (*fo_getinfo)  (struct CYG_FILE_TAG *fp, int key, char *buf,
                         int len);
    int  (*fo_setinfo)  (struct CYG_FILE_TAG *fp, int key, char *buf,
                         int len);
};
```

上面结构体内的每一个域分别是相应文件操作函数的指针。其中 fo_read()函数和 fo_write()函数的第二个参数是一个指向 UIO 结构的指针，UIO 结构定义如下：

```
struct CYG_UIO_TAG
{
    struct CYG_IOVEC_TAG *uio_iov;    /* pointer to array of iovecs */
    int                   uio_iovcnt; /* number of iovecs in array */
    off_t                 uio_offset; /* offset into file this uio corresponds to */
    ssize_t               uio_resid;  /* residual i/o count */
    enum cyg_uio_seg      uio_segflg; /* see above */
    enum cyg_uio_rw       uio_rw;     /* see above */
};
```

其中 uio_iov 是一个 CYG_IODEV_TAG 结构：

```
struct CYG_IOVEC_TAG
{
    void *iov_base;/* Base address. */
     ssize_t iov_len;/* Length. */
};
```

这些数据结构包含了数据传送操作的参数。它支持 scatter/gather 操作，并记录数据传送的进展状态。

当打开一个文件时，或者在使用其他方法（例如使用 socket()或 accept()函数）创建一个文件对象时，文件系统的 open 操作负责对文件表中除 f_ucount、f_syncmode 和 f_mte 以外的每个域进行初始化。由于文件表的 f_flag 中含有属于 FILEIO 基本结构的一些信息，因此对它所作的任何变化都必须进行相应的操作。

9.2 文件目录

文件系统的所有操作都要使用目录指针作为它的一个参数。目录指针是一个由文件系统进行管理的不透明的句柄，它应该被封装成在一个文件系统内对一个特定目录的引用。它可能是一个指向某个目录的数据结构的指针（如索引节点 inode），也可能是一个指向某个目录路径的指针。

指定当前目录的方法是使用一个安装表入口指针和一个目录指针：

```
__externC cyg_mtab_entry * cdir_mtab__entry;
__externC cyg_dir cdir_dir;
```

文件系统表中的内部 chdir()函数用于改变当前目录，它具有两种使用方式。当使用 dir_out 参数来传递一个指针时，它将定位到指定的目录，并将目录指针指向该处。如果参数 dir_out 为 NULL，则其参数 dir 是一个以前形成的目前可以取消的目录指针。系统调用 chdir()调用了两次文件系统表中的内部 chdir()函数，第一次调用是取当前新目录的目录指针，如果调用成功再进行第二次调用，清除旧的目录指针。

文件系统表中的内部 opendir()函数为文件系统的读操作打开一个目录。它所允许的对文件的操作只是 read、lseek 和 close。当到达目录的底端时，opendir()返回 0。惟一允许的 seek 操作只能是定位到目录的起点，使用参数 offset=0 和 whence=SEEK_SET。

这些操作细节对于文件系统的客户端来说是不可见的，目录的访问可以通过 POSIX 函数 opendir()、readdir()和 closedir()进行。

获取当前工作目录的绝对路径可以使用 getcwd()函数。

9.3 同步

eCos 的 FILEIO 基础结构提供了一个同步机制用于对文件系统的并发访问进行控制。这种同步机制使得其他系统中现有的即使没有同步机制的文件系统也可以轻松移植到 eCos 系统。同时，这种同步机制的实现还使得新文件系统的创建更加容易，可以不用再考虑其同步问题。

eCos 基础结构为文件系统表、安装表和文件表的每一个表项都提供了一个互斥体。在对文件系统进行操作时，每一次操作之前都会锁定相应的互斥体。它们的定义如下：

① 文件系统表互斥体：

```
Cyg_Mutex fstab_lock[CYGNUM_FILEIO_FSTAB_MAX];
```

② 安装表互斥体：

```
Cyg_Mutex mtab_lock[CYGNUM_FILEIO_MTAB_MAX];
```

③ 文件表互斥体：

```
static Cyg_Mutex file_lock[CYGNUM_FILEIO_NFILE] \
        CYGBLD_ATTRIB_INIT_PRI(CYG_INIT_IO);
```

文件系统表的 syncmode 域描述了文件系统所需要的同步协议。它是下列标志的组合：

① CYG_SYNCMODE_FILE_FILESYSTEM。在文件系统级的所有操作过程中锁定文件系统表互斥体。

② CYG_SYNCMODE_FILE_MOUNTPOINT。在文件系统级的所有操作过程中锁定安装表互斥体。

③ CYG_SYNCMODE_IO_FILE。在所有文件 I/O 操作中锁定文件表互斥体。

④ CYG_SYNCMODE_IO_FILESYSTEM。在所有文件 I/O 操作中锁定文件系统表互斥体。

⑤ CYG_SYNCMODE_IO_MOUNTPOINT。在所有文件 I/O 操作中锁定安装表互斥体。

⑥ CYG_SYNCMODE_SOCK_FILE。在所有 Socket 操作中锁定文件表互斥体。

⑦ CYG_SYNCMODE_SOCK_NETSTACK。在所有 Socket 操作中锁定网络栈表互斥体。

⑧ CYG_SYNCMODE_NONE。在所有操作中不进行锁定操作。

在 open() 函数成功完成后，系统将文件系统表中的 syncmode 值复制到打开的文件对象中。

9.4 初始化和安装

前面已经提到，安装表表项来源于两个地方，一个来源是静态定义，另一个来源是使用 mount() 函数。静态安装表表项可以使用宏 MTAB_ENTRY() 来定义，这种静态表项在系统启动时被自动安装。对于安装表中的每一个 name 域非空的表项，将对文件系统表进行搜索，查找安装表的 name 域与文件系统表的 fsname 域相匹配的文件系统表表项。如果发现相匹配的文件系统表表项，则调用该表项的内部 mount 函数。mount 函数执行成功时，相应的安装表表项被标记为有效，并对其 fs 域进行初始化。mount 函数负责其 root 域的初始化。

安装表的大小由配置选项 CYGNUM_FILEIO_MTAB_MAX 的值来定义。任何没有被静态定义的安装表表项都可以使用动态安装。

文件系统可以调用 mount() 函数进行动态安装。该函数原型如下：

```
int mount(const char *devname,
          const char *dir,
          const har *fsname);
```

其参数说明如下：

devname—该文件系统所使用的设备名，其值将赋给对应的安装表表项的 devneme 域。

dir—安装点名称，将赋值给对应的安装表表项的 name 域。

fsname—文件系统名字，将赋值给对应的安装表表项的 fsname 域。

文件系统的动态安装过程如下：首先，对安装表进行搜索，查找一个 name 为 NULL 的表项作为新的安装点。然后对文件系统表进行搜索，查找名字与 fsname 相匹配的一个文件系统表表项。如果查找成功，则对安装表表项进行初始化，并调用文件系统的内部 mount 函数。如果 mount 成功返回，则将该安装表表项标记为有效，并对其 fs 域进行初始化。

卸载文件系统可以使用 umount () 函数来实现。静态安装的文件系统和动态安装的文件系统都可以使用该函数进行卸载。umount () 函数原型如下：

```
int umount(const char *name);
```

该函数对安装表进行搜索，查找 name 域与其参数 name 相匹配的安装表表项。当这种匹配成功时，将调用文件系统的内部 umount 函数，成功返回后将安装表表项的 valid 域置为 false，name 域置为 NULL，使该安装表表项失效。

9.5 文件操作

eCos 是一个支持 POSIX 标准的嵌入式实时操作系统，这种支持来自于两个包：POSIX 包和 FILEIO 包。POSIX 包提供对线程、信号、同步、定时器和消息队列的支持，而 FILEIO 包提供了对文件和设备 I/O 的支持。eCos 对文件系统的操作主要包括文件系统的安装和卸载、目录操作、文件操作等几个方面。它实现了 POSIX 标准中的下列文件和目录函数：

```
DIR *opendir(const char *dirname);
struct dirent *readdir(DIR *dirp);
int readdir_r(DIR *dirp, struct dirent *entry, struct dirent **result);
void rewinddir(DIR *dirp);
int closedir(DIR *dirp);
int chdir(const char *path);
char *getcwd(char *buf, size_t size);
int open(const char * path , int oflag , ...);
int creat(const char * path, mode_t mode);
int close (int filedes);
int link(const char *existing, const char *new);
int mkdir(const char *path, mode_t mode);
int unlink(const char *path);
int rmdir(const char *path);
int rename(const char *old, const char *new);
int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int access(const char *path, int amode);
long pathconf(const char *path, int name);
long fpathconf(int fd, int name);
```

前面已经提到，eCos 对文件进行操作时也采取 UNIX 文件系统中的文件描述符来引述被打开的文件。文件描述符是一个非负整数。当打开一个现存文件或创建一个新文件时，将返

回一个文件描述符。当读、写一个文件时，用 open 或 creat 返回的文件描述符来描述该文件，并将其作为参数传送给 read 或 write 操作函数。

与其他操作系统一样，eCos 对文件操作也具有一些限制：

① 文件名的最大长度为 64 个字符 (包括路径名)。

② 允许打开的最大文件数由配置选项 CYGNUM _ FILEIO _ NFILE 指定。

③ 文件描述符的最大值由配置选项 CYGNUM _ FILEIO _ NFD 指定。

通过 eCos 配置工具可以对文件系统的操作进行配置。图 9-1 是 eCos 图形配置工具中 POSIX 文件 IO 兼容层的配置选项信息。从其配置选项可以看出，最多允许打开文件数量、最多打开文件描述符、最多允许文件系统安装的数量等都可以在配置期间进行指定。

9.5.1 文件系统的安装 mount 与卸载 umount

前面已经介绍了文件系统的安装和卸载的两个函数 mount () 和 umount ()。在对一个文件系统进行操作时，首先必须使用 mount () 函数将其安装到系统的文件系统表中。如果不再使用该文件系统，可以使用 umount () 函数将其从文件系统表中卸载。

下面是 RAM 文件系统 ramfs 的安装与卸载的例子：

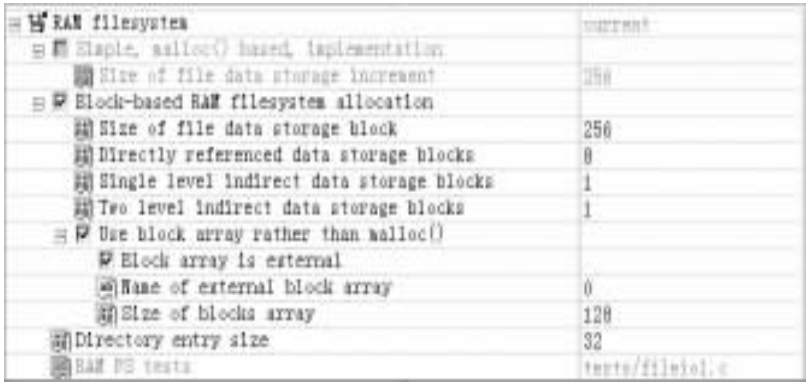


图 9-1　POSIX 文件 IO 兼容层的配置

```
int main(int argc, char * * argv)
{
    int err;
    int fd;
    err = mount("", "/ram", "ramfs");                    //安装 ramfs 文件系统
    if(err < 0) SHOW_RESULT(mount, err);                 //显示安装错误
    ...
    fd = open("/ram/test", O_WRONLY|O_CREAT);
    ... //文件操作 (略)
    close(fd);
    ...
    err = umount("/ram");                                //卸载 ramfs 文件系统
    if(err < 0) SHOW_RESULT(umount, err);                //显示卸载错误
}
```

该例中使用了 mount()函数将名字为“ramfs”的 RAM 文件系统进行安装，安装点为“/ram”。文件操作完成后，使用 umount()函数将位于安装点“/ram”处的文件系统 ramfs 进行卸载操作。例子中打开一个名为“test”的文件，其路径为“/ram”，表示它属于 ramfs 文件系统。

9.5.2 open、creat 和 close 函数

open()函数用于打开或创建一个文件。函数原型为：

```
int open(const char * path, int oflag, .../*, mode_t mode */);
```

若成功则返回文件描述符，若出错则返回-1。

参数 path 是将要打开或创建的文件的名字。

参数 oflag 用来说明此函数的多个选择项。如果使用第三个参数 mode，eCos 的当前版本将忽略此参数。使用下列一个或多个常数的或操作构成 oflag 参数（这些常数定义在<fcntl.h>头文件中）：

① O_RDONLY 只读打开。

② O_WRONLY 只写打开。

③ O_RDWR 读、写打开。

上面这三个常数只能选择一个。下列常数则是可选择的：

① O_CREAT 若此文件不存在则创建它。

② O_EXCL 如果同时指定了 O_CREAT 而文件已经存在，则出错。可以用来测试一个文件是否存在，如果不存在则创建此文件。

③ O_NOCTTY 如果 path 指的是终端设备，则不将此设备分配为控制终端。

④ O_TRUNC 如果此文件存在，而且为只读或只写成功打开，则将其长度截短为 0。

⑤ O_APPEND 每次写操作都加到文件的尾端。

⑥ O_NONBLOCK 将本次打开操作和后续的 I/O 操作设置为非阻塞方式。

⑦ O_DSYNC 写 I/O 数据同步（每次 write 数据都要等到物理 I/O 操作完成）。

⑧ O_RSYNC I/O 读同步（每次 read 都等要到物理 I/O 操作完成）。

⑨ O_SYNC 写 I/O 文件同步（每次 write 文件都要等到物理 I/O 操作完成）。

creat()函数也用于创建一个新文件，函数原型为：

```
int creat(const char * path, mode_t mode);
```

若成功则返回只写打开的文件描述符，若出错则返回-1。

此函数等效于：

```
open(path, O_WRONLY|O_CREAT|O_TRUNC, mode);
```

close()函数用于关闭一个已打开的文件。函数原型为：

```
int close (int fd);
```

若成功则返回 0，若出错则返回-1。参数 fd 为文件描述符。

9.5.3 read、write 和 lseek 函数

read()函数用于从文件中读数据。函数原型为：

```
ssize_t read(int fd, void *buf, size_t nbyte);
```

若 read 成功,则返回读取的字节数。如果到达文件末尾,则返回 0。

参数 fd 为文件描述符,buf 为读回的数据存放地址指针,nbyte 为将要读取的字节数目。

write()函数用于向文件写数据。函数原型为:

```
ssize_t write(int fd, const void *buf, size_t nbyte);
```

若成功则返回已写的字节数,若出错则返回-1。其返回值通常与参数 nbyte 的值不同。该函数出错的一个常见原因是 磁盘已写满,或者超过了文件长度限制。

参数 fd 为文件描述符,buf 为写数据地址指针,nbyte 为写字节长度。

对于普通文件,写操作从文件的当前位置(当前文件指针)开始。如果在打开该文件时选择了 O_APPEND,则在每次写操作之前都将文件指针设置在文件的尾端。在写操作成功完成后,该文件指针将增加实际写的字节数。

lseek()函数用于显式地定位一个打开文件。函数原型为:

```
off_t lseek(int fd, off_t offset, int whence);
```

若成功则返回新的文件指针,若出错则返回-1。

参数 fd 为文件描述符,对参数 offset 的解释与参数 whence 的值有关。

① 若 whence 是 SEEK_SET,则将该文件指针设置为距文件开始处 offset 个字节。

② 若 whence 是 SEEK_CUR,则将该文件指针设置为其当前值加 offset,offset 可为正或负。

③ 若 whence 是 SEEK_END,则将该文件指针设置为文件长度加 offset,offset 可为正或负。

每个被打开的文件都有一个与其相关联的"当前文件指针"。它是一个非负整数,用以度量从文件开始处计算的字节数。一般来说,读、写操作都从当前文件指针处开始,并使指针增加读或写的字节数。在默认方式下,当打开一个文件时,除非指定 O_APPEND 选择项,否则该指针被设置为 0。采用 lseek()函数可以对一个已打开的文件进行定位。它仅返回当前的文件指针,并不引起任何 I/O 操作,返回的文件指针用于后续的读或写操作。

9.5.4 fcntl 函数

fcntl()函数可以改变已经被打开的文件的性质。函数原型为:

```
int fcntl(int fd, int cmd,.../* int arg */);
```

若成功,它的返回值与参数 cmd 有关(见下面的说明),若出错则返回-1。

fcntl 函数有五种功能:

① 复制一个现有的文件描述符(cmd=F_DUPFD)。

② 获取/设置文件描述符标记(cmd = F_GETFD 或 F_SETFD)。

③ 获取/设置文件状态标志(cmd = F_GETFL 或 F_SETFL)。

④ 获取/设置异步 I/O 拥有权限(cmd = F_GETOWN 或 F_SETOWN)。

⑤ 获取/设置记录锁(cmd = F_GETLK,F_SETLK 或 F_SETLKW)。

参数 cmd 具有下述值:

① F _ DUPFD 复制文件描述符 fd,新文件描述符作为函数的返回值。新描述符与 fd 共享同一文件表表项。但新描述符有它自己的一套文件描述符标志,其 FD _ CLOEXEC 文件描述符标志则被清除。

② F _ GETFD 与 fd 对应的文件描述符标志作为函数的返回值。当前版本的 eCos 只定义了一个文件描述符标志 FD _ CLOEXEC。

③ F _ SETFD 对 fd 设置文件描述符标志。新标志值按第三个参数 (取整型值) 设置。

④ F _ GETFL 与 filedes 对应的文件状态标志作为函数的返回值。在 open 函数的介绍中已说明了文件状态标志。

⑤ F _ SETFL 将文件状态标志设置为第三个参数的值 (取整型值)。可以更改的几个标志是:O _ APPEND,O _ NONBLOCK,O _ SYNC 和 O _ ASYNC。

F _ GETLK、F _ SETLK 或 F _ SETLKW 用于记录锁。第三个参数是一个指向 flock 结构的指针 (为方便起见,用 flockprt 表示):

① F _ GETLK 获取记录锁信息。

② F _ SETLK 设置由 flockptr 所描述的记录锁。

③ F _ SETLKW 设置由 flockptr 所描述的锁,如果被阻塞就等待。

flock 的结构定义如下:

```
struct flock {
        short l_type;     /* F_RDLCK, F_WRLCK, F_UNLCK */
        short l_whence;   /* Flag for starting offset */
        off_t l_start;    /* Relative offset in bytes */
        off_t l_len;      /* Size; if 0, then until EOF */
        pid_t l_pid;      /* Process ID of the process holding the lock,
                             returned with F_GETLK. */
};
```

其中:

l _ type一锁的期望类型:F _ RDLCK (共享读锁)、F _ WRLCK (独占性写锁) 或 F _ UNLCK (解锁一个区域)。

l _ whence 和 l _ start一被加锁或解锁的区域的起始地址。l _ start 是相对位移量 (字节),l _ whence 则决定了相对位移量的起点。这与 lseek 函数中最后两个参数类似。

l _ len一区域的长度。

l _ pid一该锁的拥有者 ID 号,当 cmd 为 F _ GETLK 时返回。

eCos 当前只实现了 F _ DUPFD 命令。

9.5.5 dup 和 dup2 函数

这两个函数都用于复制一个现有的文件描述符。函数原型为:

```
int dup (int fd);
int dup2 (int fd,int fd2);
```

若成功则返回新的文件描述符,若出错则返回-1。

由 dup 返回的新文件描述符一定是当前可用文件描述符中的最小数值。而 dup2 则可以用参数 fd2 指定新描述符的数值。如果 fd2 已经打开，则先将其关闭。如果 fd 等于 fd2，则 dup2 返回 fd2，而不关闭它。

这些函数返回的新文件描述符与参数 fd 共享同一个文件表项。

9.5.6 stat 和 fstat 函数

这两个函数都用于返回文件信息。函数原型如下：

```
int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
```

若成功则返回 0，若出错则返回-1。

对于一个给定的文件名 path，stat 函数返回一个与该文件有关的信息结构，fstat 函数获取已在文件描述符 fd 上打开的文件的相关信息。这些函数的第二个参数是个指针，它指向一个表示文件信息的结构，函数在执行过程中填写该结构（buf 指向该结构）。该结构定义如下：

```
struct stat {
    mode_t    st_mode;    /* File mode */
    ino_t     st_ino;     /* File serial number */
    dev_t     st_dev;     /* ID of device containing file */
    nlink_t   st_nlink;   /* Number of hard links */
    uid_t     st_uid;     /* User ID of the file owner */
    gid_t     st_gid;     /* Group ID of the file's group */
    off_t     st_size;    /* File size (regular files only) */
    time_t    st_atime;   /* Last access time */
    time_t    st_mtime;   /* Last data modification time */
    time_t    st_ctime;   /* Last file status change time */
};
```

该结构提供了文件的一些具体信息，包括文件类型、用户 ID、文件组、包含该文件的设备 ID、文件大小、上次访问时间、最后数据修改时间、最后状态改变时间、文件许可权限等等。

表 9-1 列举了五种文件类型的宏定义，它们可用于对 st_mode 作出判断。

表 9-1 文件类型宏

宏	文件类型
S_ISREG()	普通文件
S_ISDIR()	目录文件
S_ISCHR()	字符特殊文件
S_ISBLK()	块特殊文件
S_ISFIFO()	FIFO

st_mode 还包含了文件的访问权限，见表 9-2。

9.5.7 access 函数

当用 open 函数打开一个文件时，需要对用户 ID 和组 ID 进行文件访问权限的测试。access()函数按实际的用户 ID 和实际的组 ID 对该文件进行访问权限测试。其函数原型为：

```
int access(const char * path, int mode);
```

若成功则返回 0，若出错则返回-1。

其中，mode 是表 9-3 中所列常数的逐位或运算。

表 9-2 文件访问权限许可屏蔽位

t_mode 屏蔽	意　义
S_IRUSR	用户－读
S_IWUSR	用户－写
S_IXUSR	用户－执行
S_IRWXU	用户－读、写、执行
S_IRGRP	组－读
S_IWGRP	组－写
S_IXGRP	组－执行
S_IRWXG	组－读、写、执行
S_IROTH	其他－读
S_IWOTH	其他－写
S_IXOTH	其他－执行
S_IRWXO	其他－读、写执行

表 9-3 access 函数的 mode 常数

mode	说　明
R_OK	测试读权限
W_OK	测试写权限
X_OK	测试执行权限
F_OK	测试文件是否存在

在 eCos 的当前版本中，access()只对 F_OK 进行测试，对其他三个选项进行测试时均返回 1。

9.5.8 link，unlink，remove 和 rename 函数

像 UNIX 文件系统一样，eCos 文件系统也允许一个文件同时拥有多个连接，它所允许的最大连接数为 8。link()函数用于创建文件连接，其函数原型为：

```
int link(const char * existingpath, const char * newpath);
```

若成功则返回 0，若出错则返回-1。此函数为现有文件 existingpath 创建一个新的连接

newpath，如若 newpath 已经存在，则返回错误。

如果要删除一个文件连接，则可以使用 unlink () 函数，其函数原型为：

```
int unlink (const char * pathname) ;
```

若成功则返回 0，若出错则返回-1。此函数删除文件连接，pathname 所引用的文件的连接计数减 1。如果该文件还有其他连接，则仍可通过其他的连接访问该文件数据。如果出错，则不对该文件作任何更改。只有当连接计数达到 0 时，该文件的内容才可被删除。

一些 eCos 系统提供了文件删除 remove () 函数。remove () 函数是 ANSI C 提供的函数，若系统没有该函数，则应该使用 unlink () 函数进行文件的删除操作。

remove () 函数原型如下：

```
int remove (const char * pathname) ;
```

若成功则返回 0，若出错则返回-1。

eCos 另外还提供了一个对文件进行更名的函数 rename ()，其函数原型为：

```
int rename (const char * old _ name, const char * new _ name) ;
```

若成功则返回 0，若出错则返回-1。该函数将 old _ name 所指的文件名字用 new _ name 指定的新名字替代。

9.5.9 mkdir 和 rmdir 函数

mkdir () 函数用于创建目录，rmdir () 函数用于删除目录。

mkdir () 的函数原型如下：

```
int mkdir (const char * pathname, mode _ t mode) ;
```

若成功则返回 0，若出错则返回-1。此函数创建一个 pathname 指定的新的空目录。mode 指定该目录的文件访问权限。

rmdir () 函数原型如下：

```
int rmdir (const char * pathname) ;
```

若成功则返回 0，若出错则返回-1。此函数删除 pathname 指定目录。

9.5.10 opendir、readdir、rewinddir 和 closedir 函数

这四个函数用于实现对目录的读操作。函数原型为：

```
DIR * opendir (const char * dirname) ;
struct dirent * readdir (DIR * dirp) ;
void rewinddir (DIR * dirp) ;
int closedir (DIR * dirp) ;
```

opendir () 函数若成功则返回一个 DIR 指针，若出错则返回 NULL。

readdir () 函数若成功则返回一个 dirent 结构的指针，若在目录尾或出错则返回 NULL。

rewinddir () 函数和 closedir () 函数若成功则返回 0，若出错则返回-1。

dirent 结构定义如下：

```
struct dirent {
    char d_name [NAME_MAX + 1]; /* null-terminated filename */
}
```

DIR 结构是一个内部结构，上述四个函数用它来保存被读目录的相关信息。opendir 返回的指向 DIR 结构的指针被另外三个函数所使用。opendir 执行初始化操作，readdir 的第一次调用将读取目录中的第一个目录项。

9.5.11 chdir 和 getcwd 函数

chdir()函数用于改变当前工作目录，getcwd()函数用于获取当前工作目录的绝对路径。它们的函数原型如下：

```
int chdir(const char *path);
char *getcwd(char *buf, size_t size);
```

chdir()函数若成功则返回 0，若出错则返回-1。它将当前目录改变到参数 path 指定的目录。

getcwd()函数若成功则返回 buf，若出错则返回 NULL。它有两个参数，一个是缓存地址 buf，另一个是缓存的长度 size。该缓存必须有足够的长度以容纳绝对路径名再加上一个 null 终止字符，否则返回错误。

9.5.12 Socket 操作

如果 eCos 系统被配置成支持网络协议栈，它的 FILEIO 还将提供对标准 BSD socket 调用的支持。Socket 是网络编程的入口，它提供了大量的系统调用。一个 socket 就是一个网络连接，是网络传输的入口，它位于网络协议之上，屏蔽了不同网络协议之间的差异。eCos 向用户提供了针对 socket 的文件 I/O 操作，网络通信的操作就像对文件的操作一样方便。这些操作与网络协议紧密联系在一起，是应用程序和网络协议之间的接口。

eCos 使用一个网络栈表(Netstack Table)来表示对网络的支持，该表的每一个表项分别对系统所支持的每一个网络协议栈进行描述。对于系统所支持的每一个网络协议栈，都要使用宏 NSTAB_ENTRY()进行注册，向该表输出一个表项。

网络协议栈表的表项结构定义如下：

```
struct cyg_nstab_entry
{
    cyg_bool            valid;          //true if stack initialized
    cyg_uint32          syncmode;       //synchronization protocol
    char                *name;          //stack name
    char                *devname;       //hardware device name
    CYG_ADDRWORD        data;           //private data value
    int (*init)         (cyg_nstab_entry *nste);
    int (*socket)       (cyg_nstab_entry *nste, int domain, int type,
```

```
                int protocol, cyg_file *file);
};
```

网络协议栈表是一个类似于文件系统表和安装表的组合表。如果所定义的协议栈初始化 init()函数成功返回,则 valid 域为 true, syncmode 域包含了相应的同步方式 CYG_SYNCMODE_SOCK_*(9.3 节已有说明)。name 为该协议栈的名字。devname 是该协议栈所使用的设备名字,它可能是在/dev 下的一个设备名字,也有可能是该协议栈专用的一个设备名字。

系统在初始化时通过调用 init()函数启动该协议栈运行。如果 init()函数返回一个非 0 值,则 valid 域被置为 false,系统将忽略该协议栈。

通过调用 socket()函数可以产生一个基于该协议栈的 socket。当调用 API 函数 socket()时,它对网络协议栈表进行扫描,并调用每一个有效表项内的内部 socket 函数,如果返回一个非 0 值,则继续扫描下一个有效的协议栈表项。如果到达表的尾部,则终止这种扫描并返回错误。

一个成功的 socket 调用将初始化一个文件对象,文件对象的 f_xops 域指向一个包含各种函数调用的结构体:

```
struct cyg_sock_ops
{
    int (*bind)       (cyg_file *fp, const sockaddr *sa, socklen_t len);
    int (*connect)    (cyg_file *fp, const sockaddr *sa, socklen_t len);
    int (*accept)     (cyg_file *fp, cyg_file *new_fp,
                       struct sockaddr *name, socklen_t *anamelen);
    int (*listen)     (cyg_file *fp, int len);
    int (*getname)    (cyg_file *fp, sockaddr *sa, socklen_t *len,
                       int peer);
    int (*shutdown)   (cyg_file *fp, int flags);
    int (*getsockopt) (cyg_file *fp, int level, int optname,
                       void *optval, socklen_t *optlen);
    int (*setsockopt) (cyg_file *fp, int level, int optname,
                       const void *optval, socklen_t optlen);
    int (*sendmsg)    (cyg_file *fp, const struct msghdr *m,
                       int flags, ssize_t *retsize);
    int (*recvmsg)    (cyg_file *fp, struct msghdr *m,
     socklen_t *namelen, ssize_t *retsize);
};
```

从结构体内的每一个函数名字可以看出它们支持哪一个 API 函数。getname 函数支持两个 API 函数:getsockname()和 getpeername(),sendmsg 和 recvmsg 函数对 send()、sendto()、sendmsg()、recv()、recvfrom()和 recvmsg()等 API 函数提供支持。

9.6 创建文件系统

除了 eCos 源码所提供的已有文件系统外,用户在开发过程中还可以根据具体平台的实际

需求创建新的文件系统。创建新的文件系统时，首先必须定义一个文件系统表表项，并对文件IO操作进行定义。

创建新文件系统最简单也最为方便的一个方法是复制一个现有文件系统，再进行相应的修改。既可以使用eCos源码中FILEIO包提供的test文件系统作为蓝本进行复制，也可以使用RAM文件系统或ROM文件系统进行复制。

下面以RAM文件系统为例，说明如何创建一个新文件系统。

创建新文件系统的第一步是提供组成文件系统接口的函数。RAM文件系统的接口函数定义如下：

```
//文件系统操作函数：
static int ramfs_mount      (cyg_fstab_entry *fste, cyg_mtab_entry *mte);
static int ramfs_umount     (cyg_mtab_entry *mte);
static int ramfs_open       (cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                             int mode, cyg_file *fte);
static int ramfs_unlink     (cyg_mtab_entry *mte, cyg_dir dir, const char *name);
static int ramfs_mkdir      (cyg_mtab_entry *mte, cyg_dir dir, const char *name);
static int ramfs_rmdir      (cyg_mtab_entry *mte, cyg_dir dir, const char *name);
static int ramfs_rename     (cyg_mtab_entry *mte, cyg_dir dir1, const char *name1,
                             cyg_dir dir2, const char *name2);
static int ramfs_link       (cyg_mtab_entry *mte, cyg_dir dir1, const char *name1,
                             cyg_dir dir2, const char *name2, int type);
static int ramfs_opendir    (cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                             cyg_file *fte);
static int ramfs_chdir      (cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                             cyg_dir *dir_out);
static int ramfs_stat       (cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                             struct stat *buf);
static int ramfs_getinfo    (cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                             int key, void *buf, int len);
static int ramfs_setinfo    (cyg_mtab_entry *mte, cyg_dir dir, const char *name,
                             int key, void *buf, int len);

//文件操作函数：
static int ramfs_fo_read    (struct CYG_FILE_TAG *fp, struct CYG_UIO_TAG *uio);
static int ramfs_fo_write   (struct CYG_FILE_TAG *fp, struct CYG_UIO_TAG *uio);
static int ramfs_fo_lseek   (struct CYG_FILE_TAG *fp, off_t *pos, int whence);
static int ramfs_fo_ioctl   (struct CYG_FILE_TAG *fp, CYG_ADDRWORD com,
                             CYG_ADDRWORD data);
static int ramfs_fo_fsync   (struct CYG_FILE_TAG *fp, int mode);
static int ramfs_fo_close   (struct CYG_FILE_TAG *fp);
static int ramfs_fo_fstat   (struct CYG_FILE_TAG *fp, struct stat *buf);
static int ramfs_fo_getinfo (struct CYG_FILE_TAG *fp, int key, void *buf, int len);
static int ramfs_fo_setinfo (struct CYG_FILE_TAG *fp, int key, void *buf, int len);
```

```
//目录操作函数：
static int ramfs_fo_dirread (struct CYG_FILE_TAG * fp, struct CYG_UIO_TAG * uio);
static int ramfs_fo_dirlseek (struct CYG_FILE_TAG * fp, off_t * pos, int whence);
```

上面对文件系统的所有文件 IO 操作进行了定义，其中有两个可选的文件 IO 操作函数 fo_read 和 fo_lseek。

完成上述定义后，第二步的工作是对文件系统表表项进行定义。使用宏 FSTAB_ENTRY 来完成这一步工作：

```
//文件系统表表项：
//Fstab entry.
//This defines the entry in the filesystem table.
//For simplicity we use _FILESYSTEM synchronization for all accesses since
//we should never block in any filesystem operations.
FSTAB_ENTRY (ramfs_fste, "ramfs", 0,
                    CYG_SYNCMODE_FILE_FILESYSTEM |
                    CYG_SYNCMODE_IO_FILESYSTEM,
                    ramfs_mount,
                    ramfs_umount,
                    ramfs_open,
                    ramfs_unlink,
                    ramfs_mkdir,
                    ramfs_rmdir,
                    ramfs_rename,
                    ramfs_link,
                    ramfs_opendir,
                    ramfs_chdir,
                    ramfs_stat,
                    ramfs_getinfo,
                    ramfs_setinfo);
```

宏 FSTAB_ENTRY () 的第一个参数给出了该文件系统表表项的名字“ramfs_fste”，其余部分被初始化程序所使用，用于对文件系统表表项结构的各个成员域进行初始化。

创建文件系统的第三步是定义一个文件操作表。该表被安装在所有被打开的文件表表项内，在进行文件操作时将调用文件操作表中的这些函数。RAM 文件系统的文件操作表定义如下：

```
//文件操作：
//This set of file operations are used for normal open files.
static cyg_fileops ramfs_fileops =
{
    ramfs_fo_read,
    ramfs_fo_write,
```

```
        ramfs _ fo _ lseek,
        ramfs _ fo _ ioctl,
        cyg _ fileio _ seltrue,
        ramfs _ fo _ fsync,
        ramfs _ fo _ close,
        ramfs _ fo _ fstat,
        ramfs _ fo _ getinfo,
        ramfs _ fo _ setinfo
};
```

除了 fo _ select 域使用函数 cyg _ fileio _ seltrue () 的指针外，该结构体内其他所有域都指向文件系统提供的相应函数。cyg _ fileio _ seltrue () 函数由 FILEIO 包提供，它是一个 select 函数，对所有操作均返回 true。

创建文件系统的最后一步是定义一组读目录使用的文件操作函数。下面是 RAM 文件系统的读目录操作函数表，该表只定义了 fo _ dirread 和 fo _ dirlseek 操作，其余操作除 cyg _ fileio _ seltrue 和 fo _ close 之外，都使用 FILEIO 包中的占位函数，只返回一个错误代码。

```
//目录操作
//This set of operations are used for open directories. Most entries
//point to error-returning stub functions. Only the read, lseek and
//close entries are functional.
static cyg _ fileops ramfs _ dirops =
{
        ramfs _ fo _ dirread,
        (cyg _ fileop _ write *) cyg _ fileio _ enosys,
        ramfs _ fo _ dirlseek,
        (cyg _ fileop _ ioctl *) cyg _ fileio _ enosys,
        cyg _ fileio _ seltrue,
        (cyg _ fileop _ fsync *) cyg _ fileio _ enosys,
        ramfs _ fo _ close,
        (cyg _ fileop _ fstat *) cyg _ fileio _ enosys,
        (cyg _ fileop _ getinfo *) cyg _ fileio _ enosys,
        (cyg _ fileop _ setinfo *) cyg _ fileio _ enosys
};
```

上述步骤已完成对文件系统的创建，如果希望在系统启动时能自动安装新创建的文件系统，则必须定义一个相应的安装表表项。使用宏 MTAB _ ENTRY () 可以完成这一工作。下面是 RAM 文件系统使用该宏的例子：

```
MTAB _ ENTRY (ramfs _mte1,
                        "/",
                        "ramfs",
                        "",
                        0);
```

第一个参数为该表项提供一个名字，其余参数分别对应安装表表项中的 name、fsname、devname 和 data 域。

至此，新的文件系统的基本骨架已经形成，剩下的工作是具体实现上面所定义的函数。显然，这些函数的具体实现形式完全依赖于文件系统本身。如何实现这些函数可以参阅 eCos 源码所提供的 RAM 和 ROM 文件系统的具体实现方法。

9.7 RAM 文件系统

RAM 文件系统是建立在内存 RAM 区的一个文件系统，可以用于临时存放文件。文件系统的大小可以根据文件的实际需求而动态增减。RAM 文件系统可以对其文件进行读写操作和目录操作。很显然，系统断电时将不会保留 RAM 文件系统中的文件。

图 9-2 描述了 eCos 图形配置工具中 RAM 文件系统的一些配置选项。在对 RAM 文件系统进行配置时，有两种数据存储机制可供选择。此外，还可以对文件数据块大小等信息进行配置。

图 9-2 RAM 文件系统的配置选项

9.7.1 文件和目录节点

RAM 文件系统中的所有文件和目录都使用节点 (node) 来表示。每一个节点所包含的信息如下：

① 节点类型 (mode)：表示该节点是文件还是目录。

② 节点引用计数 (refcnt)：对该节点的引用进行计数。对于文件，每一次打开操作被当作一次引用；对于目录，当它是当前目录或者它被打开读时被认为是一次引用。当关闭文件或目录以及离开当前目录时，相应的节点引用计数减 1。

③ 节点连接数 (blink)：该节点的所有连接数。引用该节点的每一个目录项都是该节点的一个连接。

④ 节点大小 (size)：该节点的大小，以字节为单位。

⑤ 最近访问时间 (atime)：该节点最近被访问的时间。

⑥ 最近修改时间(mtime):该节点最近被修改的时间。

⑦ 最近状态改变时间(ctime):该节点的状态信息最近改变时间。

根据配置时所选择的两种不同数据存储方式(见后面 9.7.3 节),节点还应该分别包含如下信息:

① 如果选择使用简单的单一内存块分配机制,节点信息内容还包括了所分配的内存块的大小(datasize)和指针(* data)。

② 如果选择使用基于块的内存分配机制,还包含一个指向分配给该节点的所有内存块指针数组(* direct[],或 * * indirect1[],或 * * * indirect2[]),数组的大小在配置时指定。

文件和目录的节点结构定义如下:

```
struct ramfs_node
{
    mode_t          mode;           //node type
    cyg_ucount32    refcnt;         //open file/current dir references
    nlink_t         nlink;          //number of links to this node
    size_t          size;           //size of file in bytes
    time_t          atime;          //last access time
    time_t          mtime;          //last modified time
    time_t          ctime;          //last changed status time
#ifdef CYGPKG_FS_RAM_SIMPLE
//The data storage in this case consists of a single malloced
//memory block, together with its size.
    size_t          datasize;       //size of data block
    cyg_uint8       *data;          //malloced data buffer
#else
//The data storage in this case consists of arrays of pointers to
//data blocks.
#if CYGNUM_RAMFS_BLOCKS_DIRECT > 0
    //Directly accessible blocks from the inode.
    ramfs_block *direct[CYGNUM_RAMFS_BLOCKS_DIRECT];
#endif
#if CYGNUM_RAMFS_BLOCKS_INDIRECT1 > 0
    //Single level indirection
    ramfs_block **indirect1[CYGNUM_RAMFS_BLOCKS_INDIRECT1];
#endif
#if CYGNUM_RAMFS_BLOCKS_INDIRECT2 > 0
    //Two level indirection
    ramfs_block ***indirect2[CYGNUM_RAMFS_BLOCKS_INDIRECT2];
#endif
#endif
};
```

9.7.2 目录

目录是一个节点数据为一组目录项的节点。为简化目录的管理,长的目录项将按具有固定大小的数据结构 ramfs _ dirent 进行分割,这些被分割的片段组成的链即为目录项。ramfs _ dirent 的结构定义如下:

```
struct ramfs _ dirent
{
    ramfs _ node    * node;          //pointer to node
    unsigned int    inuse :1,        //entry in use?
                    first :1,        //first directory entry fragment?
                    last :1,         //last directory entry fragment?
                    namelen :8,      //bytes in whole name
                    fraglen :8;      //bytes in name fragment
    off _ t         next;            //offset of next dirent
    //Name fragment, fills rest of entry.
    char            name [CYGNUM _ RAMFS _ DIRENT _ SIZE-
                        sizeof (ramfs _ node *) -
                        sizeof (cyg _ uint32) -
                        sizeof (off _ t)];
};
```

该结构每一个域的说明如下:

node—指向该目录项所引用的节点,出现在目录项的每一个片段中。

inuse—如果该目录项处于使用状态,则为 1,否则为 0。

first—如果这是目录项中的第一个片段,则为 1。

last—如果这是目录项中的最后一个片段,则为 1。

namelen—整个文件名的长度。

fraglen—文件名存放在该片段内的字节数。

next—该目录项的下一个片段的偏移(offset)。

name—文件名存放在该片段内的字符。

一般来说,短的文件名存放在单个片段中,而长的文件名通常以多个片段链的形式存放。

9.7.3 数据存储机制

前面已经提到,RAM 文件系统具有两种配置时可以选择的文件数据存储机制。这两种机制分别是简单数据存储机制和块数据存储机制。简单数据存储机制的配置选项是 CYGPKG _ FS _ RAM _ SIMPLE,而块数据存储机制的配置选项是 CYGPKG _ FS _ RAM _ BLOCKS。

简单数据存储机制简单地用 malloc()函数和 free()函数对节点和文件数据分配内存。文件数据被存放在 malloc()函数所分配的内存区,根据文件大小的实际需要还可以为其再分配所需的内存。

简单数据存储机制的优点是 RAM 文件系统只占用它实际所需的内存空间,剩下的内存

空间可以为其他组件所用。它的另一个好处是用于对文件系统进行管理的数据结构和程序比较简单。但这种机制有其缺点。在有些文件的堆空间所占比例相当可观时,即使还有足够多的可用内存,也存在存储碎片妨碍文件进行进一步扩展的危险。另外,这种机制要求系统提供 malloc()函数。如果该函数也是系统中其他组件所需要的,这还不会成为问题。但如果仅仅是为了该文件系统的使用而在系统中增加实现该函数的软件包,显然系统将额外增加大量的数据和程序代码,这在嵌入式系统中这是难以接受的。

另一种机制是块数据存储机制。这种机制将用于文件存储的内存分成固定大小的块。这些内存块既可以用 malloc()函数和 free()函数进行分配和释放,也可以从专门为此保留的内存块阵列中获取。在 eCos 的配置过程中,可以对内存块的大小、内存分配机制进行设置和选择。

采用块数据存储机制时,节点内的数据存储采用了三个指向这些内存块的指针数组。第一个数组直接指向内存数据块,这些数据存储内存块称为直接数据存储块,该数组的配置选项是 CYGNUM _ RAMFS _ BLOCKS _ DIRECT,默认大小为 8。第二个数组指向另一个内存块,该内存块包含了指向实际存储数据的内存块指针,这些数据存储内存块称为单级间接数据存储块,该数组的配置选项是 CYGNUM _ RAMFS _ BLOCKS _ INDIRECT1。第三个数组所指向的内存块的内容是指向另一个包含实际存储数据的内存块指针的内存块,这些数据存储内存块称为二级间接数据存储块,该数组的配置选项是 CYGNUM _ RAMFS _ BLOCKS _ INDIRECT2。在默认方式下,后两个数组分别只有一个数组元素。图 9-3 为默认情况下块数据存储机制的数据块分配示意图(块大小为 256B)。

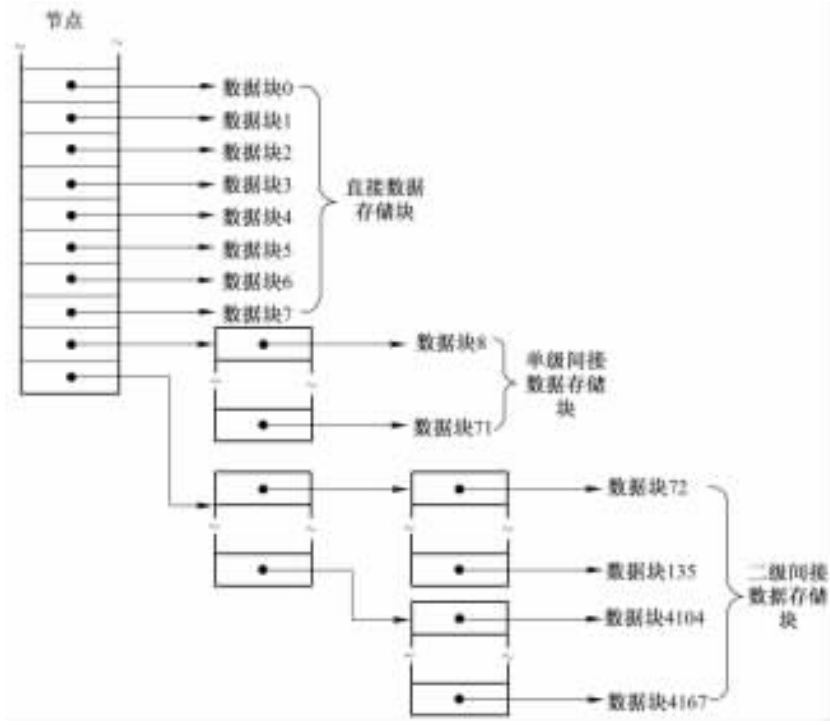


图 9-3　默认配置下 RAM 文件系统的块数据存储机制

采用块数据存储机制也有其优缺点。优点之一是可以按经过仔细考虑过的大小将所使用的内存分成固定大小的内存块,对它们的管理更为容易。另一个优点是在使用 malloc()函数

对这些内存块进行分配时，可以使用任何一处具有适当大小的空闲内存。如果使用内存块阵列，根本就不需要使用 malloc() 函数，使整个系统减少数据和程序的代码量成为可能。

块数据存储机制的缺点之一是当使用 malloc() 函数分配内存时，每一个内存块都要使用该函数进行内存分配，这种内存分配的开销是基于每一个内存块的，而不是基于一个文件。可能会造成整个可用于数据存储的内存空间变小。当使用内存块阵列时，这些内存块将为 RAM 文件系统永久保留，系统中的其他组件不能使用这些内存。

9.8 ROM 文件系统

ROM 文件系统是建立在系统 ROM 区的一个文件系统，可用于存放永久性的文件。文件系统的大小是固定的，不能加以改变。与 RAM 文件系统不同，ROM 文件系统只能进行只读操作和读目录操作，不能进行写操作。系统断电时文件数据不会丢失。

ROM 文件系统具有一个对其进行描述的头结构 Header。它的文件和目录也使用节点来表示。所有的节点和目录结构都使用头的地址作为它们的偏移（offset）基地址。头的定义如下：

```
struct romfs_disk
{
    cyg_uint32      magic;      //0-3 Marks a valid ROMFS entry
    cyg_ucount32    nodecount;  //4-7 Count of nodes in this filesystem
    cyg_ucount32    disksize;   //8-11 Count of bytes in this filesystem
    cyg_uint32      dev_id;     //12-15 ID of disk (put into stat.st_dev)
    char            name[16];   //16-31 Name - pads to 32 bytes
    romfs_node      node[0];
};
```

其中各域说明如下：

magic—表示这是一个有效的 ROM 文件系统，其值为 ROMFS_MAGIC（0x526f6d2e）。如果是 ROMFS_CIGAN（0x2e6d6f52），则说明字节排列方式有错。

nodecount—ROM 文件系统中的节点数。

disksize—ROM 文件系统的字节大小。

dev_id—ROM 盘的 ID 号（该值将赋给 stat.st_dev）。

name[]—32B 边界的填充名字。

node[]—ROM 文件系统的第一个节点。

紧跟在头之后的是节点表，节点表由各个节点组成。所有文件和目录都用节点表示。节点具有下面的的结构定义：

```
struct romfs_node
{
    cyg_uint32      mode;   //0-3     node type
    cyg_ucount32    nlink;  //4-7     number of links to this node
    cyg_uint16      uid;    //8-9     Owner id
```

```
    cyg_uint16      gid;    //10-11 Group id
    cyg_uint32      size;   //12-15 size of file in bytes
    cyg_uint32      ctime;  //16-19 creation status time
    cyg_uint32      offset; //20-23 offset of data from start of ROMFS
    cyg_uint32      pad[2]; //24-31 padding to align to 32byte boundary
};
```

其中：

mode—**节点类型，文件和目录。**

nlink—**节点连接数，每一个引用该节点的目录项是一个连接。**

uid—**该节点的拥有者** ID。

gid—**该节点所属组** ID。

size—**节点内数据的字节长度。**

ctime—**该文件的创建时间（不是** ROMFS）。

offset—**从头的开始处到该节点的第一个数据字节的偏移。**

pad[2]—32B **边界的填充数据。**

与 RAM **文件系统同样，**ROM **文件系统的目录也是一个其数据为一组目录项表的节点。目录项结构定义如下：**

```
struct romfs_dirent
{
    cyg_ucount32    node;   //Index of node in romfs_disk structure
    cyg_uint32      next;   //Offset from start of directory of
                            //a) the next entry, or
                            //b) the end of the directory data
    char            name[0]; //The name, NUL terminated
};
```

其中：

node—**该目录项所引用的节点在** romfs_disk **表中的索引，目录项的每一段都有该值。**

next—**下一个名字项的偏移。**

name—**与该节点连接相对应的文件名。**

ROM **文件系统中每个文件都将数据存储在其节点所指的单个连续的内存块内。**

9.9 文件操作实例

本节介绍 RAM **文件系统的一个文件操作实例。这是** eCos **源码中提供的一个测试程序。这些测试程序除了可以用来测试系统的正确性之外，读者还可以从测试程序中了解对文件系统进行操作的方法。**

该程序首先安装 RAM **文件系统，其安装点为“/”。在此根目录下创建文件** foo，**然后对该文件进行检查，将其复制为文件** fee **并比较这两个文件。随后创建新目录/**bar，**将文件** fee **复制到新目录下的** fum **文件（/**bar/fum），**并对它们进行比较。最后，删除所有生成的文件和目录。**

该程序还示范了对长文件名的操作。原测试程序还将 RAM 文件系统安装到另一个安装点“/ram”进行类似的操作，这里为节省篇幅，不再列出。

```
//==========================================
#include <pkgconf/hal.h>
#include <pkgconf/kernel.h>
#include <pkgconf/io_fileio.h>
#include <cyg/kernel/ktypes.h>          //base kernel types
#include <cyg/infra/cyg_trac.h>         //tracing macros
#include <cyg/infra/cyg_ass.h>          //assertion macros
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <errno.h>
#include <string.h>
#include <dirent.h>
#include <cyg/fileio/fileio.h>
#include <cyg/infra/testcase.h>
#include <cyg/infra/diag.h>             //HAL polled output
//==========================================
#define SHOW_RESULT(_fn, _res) \
diag_printf("<FAIL>:" #_fn "() returned %d %s\n", _res, _res<0 ? strerror(errno) :"");
//==========================================
#define IOSIZE 100
#define LONGNAME1       "long_file_name_that_should_take_up_more_\
                            than_one_directory_entry_1"
#define LONGNAME2       "long_file_name_that_should_take_up_more_\
                            than_one_directory_entry_2"
//==========================================
//列举目录和文件
static void listdir(char *name, int statp, int numexpected, int *numgot)
{
    int err;
    DIR *dirp;
    int num=0;

    diag_printf("<INFO>:reading directory %s\n",name);
    dirp = opendir(name);
    if(dirp == NULL) SHOW_RESULT(opendir, -1);
    for(;;)
    {
        struct dirent *entry = readdir(dirp);
        if(entry == NULL)
```

```
            break;
        num++;
        diag_printf("<INFO>:entry %14s",entry->d_name);
        if(statp)
        {
            char fullname[PATH_MAX];
            struct stat sbuf;
            if(name[0])
            {
                strcpy(fullname, name);
                if(!(name[0] == '/' && name[1] == 0))
                strcat(fullname, "/");
            }
            else fullname[0] = 0;
            strcat(fullname, entry->d_name);
            err = stat(fullname, &sbuf);
            if(err < 0)
            {
                if(errno == ENOSYS)
                    diag_printf("<no status available>");
                else SHOW_RESULT(stat, err);
            }
            else
            {
                diag_printf(" [mode %08x ino %08x nlink %d size %d]",
                    sbuf.st_mode,sbuf.st_ino,sbuf.st_nlink,sbuf.Nst_size);
            }
        }
        diag_printf("\n");
    }
    err = closedir(dirp);
    if(err < 0) SHOW_RESULT(stat, err);
    if (numexpected >= 0 && num != numexpected)
        CYG_TEST_FAIL("Wrong number of dir entries\n");
    if (numgot != NULL)
        *numgot = num;
}
//=====================================
//创建文件
static void createfile(char *name, size_t size)
{
    char buf[IOSIZE];
    int fd;
```

```
    ssize_t wrote;
    int i;
    int err;
    diag_printf("<INFO>:create file %s size %d\n",name,size);
    err = access(name, F_OK);
    if(err < 0 && errno != EACCES) SHOW_RESULT(access, err);
    for(i = 0;i < IOSIZE;i++) buf[i] = i%256;
    fd = open(name, O_WRONLY|O_CREAT);
    if(fd < 0) SHOW_RESULT(open, fd);
    while(size > 0)
    {
        ssize_t len = size;
        if (len > IOSIZE) len = IOSIZE;
        wrote = write(fd, buf, len);
        if(wrote != len) SHOW_RESULT(write, wrote);
        size -= wrote;
    }
    err = close(fd);
    if(err < 0) SHOW_RESULT(close, err);
}
//=======================================
//检查指定文件
static void checkfile(char *name)
{
    char buf[IOSIZE];
    int fd;
    ssize_t done;
    int i;
    int err;
    off_t pos = 0;
    diag_printf("<INFO>:check file %s\n",name);
    err = access(name, F_OK);
    if(err != 0) SHOW_RESULT(access, err);
    fd = open(name, O_RDONLY);
    if(fd < 0) SHOW_RESULT(open, fd);
    for(;;)
    {
        done = read(fd, buf, IOSIZE);
        if(done < 0) SHOW_RESULT(read, done);
        if(done == 0) break;
        for(i = 0;i < done;i++)
            if(buf[i] != i%256)
            {
```

```
                diag_printf("buf[%d+%d](%02x) != %02x\n",pos,i,buf[i],i%256);
                CYG_TEST_FAIL("Data read not equal to data written\n");
            }
        pos += done;
    }
    err = close(fd);
    if(err < 0) SHOW_RESULT(close, err);
}
//=====================================
//复制文件
static void copyfile(char *name2, char *name1)
{
    int err;
    char buf[IOSIZE];
    int fd1, fd2;
    ssize_t done, wrote;
    diag_printf("<INFO>:copy file %s -> %s\n",name2,name1);
    err = access(name1, F_OK);
    if(err < 0 && errno != EACCES) SHOW_RESULT(access, err);
    err = access(name2, F_OK);
    if(err != 0) SHOW_RESULT(access, err);
    fd1 = open(name1, O_WRONLY|O_CREAT);
    if(fd1 < 0) SHOW_RESULT(open, fd1);
    fd2 = open(name2, O_RDONLY);
    if(fd2 < 0) SHOW_RESULT(open, fd2);
    for(;;)
    {
        done = read(fd2, buf, IOSIZE);
        if(done < 0) SHOW_RESULT(read, done);
        if(done == 0) break;
        wrote = write(fd1, buf, done);
        if(wrote != done) SHOW_RESULT(write, wrote);
        if(wrote != done) break;
    }
    err = close(fd1);
    if(err < 0) SHOW_RESULT(close, err);
    err = close(fd2);
    if(err < 0) SHOW_RESULT(close, err);
}
//=====================================
//比较两个文件
static void comparefiles(char *name2, char *name1)
{
```

```
    int err;
    char buf1[IOSIZE];
    char buf2[IOSIZE];
    int fd1, fd2;
    ssize_t done1, done2;
    int i;
    diag_printf("<INFO>:compare files %s == %s\n",name2,name1);
    err = access(name1, F_OK);
    if(err != 0) SHOW_RESULT(access, err);
    err = access(name1, F_OK);
    if(err != 0) SHOW_RESULT(access, err);
    fd1 = open(name1, O_RDONLY);
    if(fd1 < 0) SHOW_RESULT(open, fd1);
    fd2 = open(name2, O_RDONLY);
    if(fd2 < 0) SHOW_RESULT(open, fd2);
    for(;;)
    {
        done1 = read(fd1, buf1, IOSIZE);
        if(done1 < 0) SHOW_RESULT(read, done1);
        done2 = read(fd2, buf2, IOSIZE);
        if(done2 < 0) SHOW_RESULT(read, done2);
        if(done1 != done2)
            diag_printf("Files different sizes\n");
        if(done1 == 0) break;
        for(i = 0;i < done1;i++)
            if(buf1[i] != buf2[i])
            {
                diag_printf("buf1[%d](%02x)
buf1[%d](%02x)\n",i,buf1[i],i,buf2[i]);
                CYG_TEST_FAIL("Data in files not equal\n");
            }
    }
    err = close(fd1);
    if(err < 0) SHOW_RESULT(close, err);
    err = close(fd2);
    if(err < 0) SHOW_RESULT(close, err);
}
//=====================================
//检查是否是指定目录
void checkcwd(const char *cwd)
{
    static char cwdbuf[PATH_MAX];
    char *ret;
```

```
    ret = getcwd(cwdbuf, sizeof(cwdbuf));
    if(ret == NULL) SHOW_RESULT(getcwd, ret);
    if(strcmp(cwdbuf, cwd) != 0)
    {
        diag_printf("cwdbuf %s cwd %s\n",cwdbuf, cwd);
        CYG_TEST_FAIL("Current directory mismatch");
    }
}

//==========================================
//main
int main(int argc, char **argv)
{
    int err;
    int existingdirents=-1;
    //-------------------------------------------------------------------------
    err = mount("","/","ramfs");  //安装 RAM 文件系统
    if(err < 0) SHOW_RESULT(mount, err);
    err = chdir("/");//进入根目录
    if(err < 0) SHOW_RESULT(chdir, err);
    checkcwd("/");//检查当前目录是否是根目录
    listdir("/", true, -1, &existingdirents);  //列出根目录下所有文件和目录
    if (existingdirents < 2)
      CYG_TEST_FAIL("Not enough dir entries\n");
    //-------------------------------------------------------------------------
    createfile("/foo", 202);  //创建文件 foo
    checkfile("foo");  //检查文件 foo
    copyfile("foo","fee");  //将文件 foo 复制到文件 fee
    checkfile("fee");  //检查文件 fee
    comparefiles("foo","/fee");  //比较 foo 和 fee 两个文件
    diag_printf("<INFO>:mkdir bar\n");
    err = mkdir("/bar", 0);  //创建目录/bar
    if(err < 0) SHOW_RESULT(mkdir, err);
    listdir("/", true, existingdirents+3, NULL);  //列举根目录文件
    copyfile("fee","/bar/fum");  //复制文件 fee 到文件/bar/fum
    checkfile("bar/fum");  //检查新复制的文件 fum
    comparefiles("/fee","bar/fum");  //比较 fee 和 fum 两个文件
    diag_printf("<INFO>:cd bar\n");
    err = chdir("bar");  //进入 bar 子目录
    if(err < 0) SHOW_RESULT(chdir, err);
    checkcwd("/bar");  //检查 bar 目录
    diag_printf("<INFO>:rename /foo bundy\n");
    err = rename("/foo","bundy");  //将文件/foo 更名为 bundy
```

```
if(err < 0) SHOW_RESULT(rename, err);
listdir("/", true, existingdirents+2, NULL);
listdir("", true, 4, NULL);
checkfile("/bar/bundy");
comparefiles("/fee", "bundy");
//--------------------------------------------------------------------------------
//长文件名操作：
createfile(LONGNAME1, 123);
checkfile(LONGNAME1);
copyfile(LONGNAME1, LONGNAME2);
listdir("", false, 6, NULL);
diag_printf("<INFO>:unlink " LONGNAME1 "\n");
err = unlink(LONGNAME1);
if(err < 0) SHOW_RESULT(unlink, err);
diag_printf("<INFO>:unlink " LONGNAME2 "\n");
err = unlink(LONGNAME2);
if(err < 0) SHOW_RESULT(unlink, err);
//--------------------------------------------------------------------------------
//删除文件和目录：
diag_printf("<INFO>:unlink fee\n");
err = unlink("/fee");
if(err < 0) SHOW_RESULT(unlink, err);
diag_printf("<INFO>:unlink fum\n");
err = unlink("fum");
if(err < 0) SHOW_RESULT(unlink, err);
diag_printf("<INFO>:unlink /bar/bundy\n");
err = unlink("/bar/bundy");
if(err < 0) SHOW_RESULT(unlink, err);
diag_printf("<INFO>:cd /\n");
err = chdir("/");
if(err < 0) SHOW_RESULT(chdir, err);
checkcwd("/");
diag_printf("<INFO>:rmdir /bar\n");
err = rmdir("/bar");  //删除目录/bar
if(err < 0) SHOW_RESULT(rmdir, err);
listdir("/", false, existingdirents, NULL);
(省略)
CYG_TEST_PASS_FINISH("fileio1");
}
```

第 10 章　网络支持与编程

随着网络应用的普及，嵌入式系统对网络支持的要求也越来越迫切。eCos 为满足这种应用需求，在网络方面也提供了强有力的支持。它所包含的公共网络协议包（Common Networking Package）支持完整的 TCP/IP 网络协议栈，提供了基于 OpenBSD 和 FreeBSD 的两种实现。eCos 目前支持的网络服务包括 FTP、TFTP、SNMP、DNS、HTTP 等等。

本章首先描述 eCos 网络驱动程序的设计方法，然后介绍基于 OpenBSD 和 FreeBSD 的网络协议栈支持，并简单介绍目前 eCos 提供的一些网络服务。最后用简单的例子来说明如何进行 eCos 的网络编程。

10.1　eCos 网络配置

使用 eCos 配置工具可以对网络支持进行配置。eCos 的网络支持由多个包组成，有两种方法将这些网络支持包加入到系统中来。一种方式是采用 eCos 提供的模板，如“net”模板，在图形配置工具中选择菜单选项“Build→Templates”，在出现的模板选择对话框中选择“net”包。另一种方式是直接使用菜单选项“Build→Packages”，在出现的包选择对话框中选择加入相应的网络支持包。

一旦选择了网络支持包，就可以使用配置工具对具体的网络支持特性进行配置。对网络的配置主要有两个方面，一个是基本的网络支持配置，另一个是网络设备的配置。

图 10-1 是使用图形配置工具对 eCos 的基本网络支持进行配置的示意图。基本网络支持的配置包括的主要内容有：

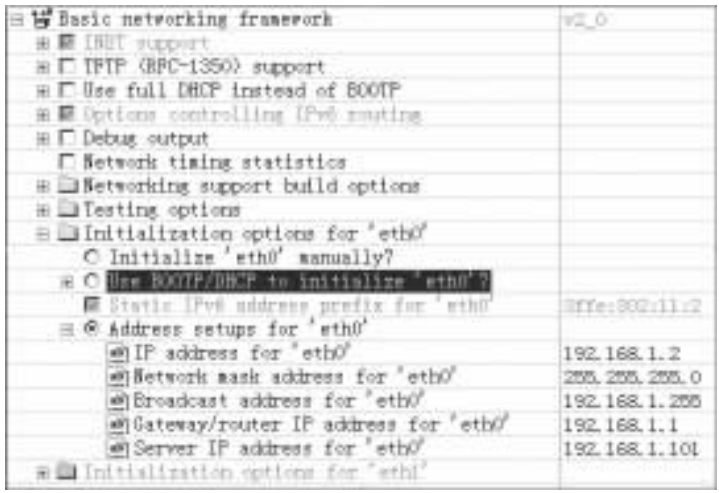


图 10-1　基本网络支持的配置

① 基本网络协议栈（TCP/IP 等）的配置。

② 网络协议栈的使用方式。

③ 网络属性设置,包括 IP 地址、网关、子网掩码等。

④ 编译和调试选项配置。

对网络设备的配置包括对以太网驱动程序的配置和网卡设备选项的配置,如图 10-2 所示。网络设备的配置位于 I/O 子系统包内。

图 10-2　网络设备的配置

IP 地址可以单独进行设置。可以使用手工设置(在必须通过编程来实现的情况下),也可以使用 BOOTP/DHCP 来配置,还可以明确指定。如果增加其他的网络接口,则必须对这些新的接口进行手工配置。配置所使用的数据包括:

① IP 地址。

② 子网屏蔽。

③ 广播地址。

④ 网关/路由。

⑤ 服务器地址。

10.2　以太网驱动程序设计

eCos 源码中虽然提供了许多以太网驱动程序,但在进行具体开发时可能会使用各种各样的网络芯片,这就需要进行驱动程序设计。驱动程序的设计在第 8 章中有详细介绍,这里将重点介绍如何进行网络驱动程序的设计。

eCos 的网络驱动程序分为两个层次。其中的一个层次(高层驱动程序)独立于具体硬件,包含了所有网络协议栈专用程序。另一个层次(底层驱动程序)与平台硬件密切相关,它使用简单的 API 接口与独立于硬件的那一层驱动程序进行通信。如果其他网络协议栈使用同一

个 API 接口，那么它们也可以使用与硬件相关的网络设备驱动程序。eCos 的这种设计方法有利于开发其他的网络协议栈，使得与硬件相关的程序代码具有可重用性。

与具体硬件没有直接联系的高层驱动程序是网络协议栈的一个组成部分，在其下面有一个或多个与实际网络硬件相关的底层驱动程序。每一个驱动程序包含了一个或多个驱动程序实例。底层驱动程序并不了解使用它们的上层网络协议栈的详细信息，同一个驱动程序可以不加修改地用于 TCP/IP 协议栈、RedBoot 等。eCos 网络支持的这种结构如图 10-3 所示。

图 10-3　eCos 网络驱动程序层次结构

10.2.1　底层驱动程序基本框架

底层驱动程序必须提供对硬件进行操作的函数，这些操作包括对硬件进行初始化和启动、硬件的停止、设备控制、状态查询、数据的发送与接收、中断处理等等。所有这些对硬件操作的函数都被封装到一个数据结构 eth _ hwr _ funs 中。这个数据结构的格式如下：

```
struct eth _ hwr _ funs {
    //Initialize hardware (including startup)
    void (* start)(struct eth _ drv _ sc * sc,
                   unsigned char * enaddr,
                   int flags);
    //Shut down hardware
    void (* stop)(struct eth _ drv _ sc * sc);
    //Device control (ioctl pass-thru)
    int (* control)(struct eth _ drv _ sc * sc,
                    unsigned long key,
                    void * data,
                    int data _ length);
    //Query - can a packet be sent?
    int (* can _ send)(struct eth _ drv _ sc * sc);
    //Send a packet of data
    void (* send)(struct eth _ drv _ sc * sc,
                  struct eth _ drv _ sg * sg _ list,
```

```
                    int sg_len,
                    int total_len,
                    unsigned long key);
    //Receive [unload] a packet of data
    void (*recv)(struct eth_drv_sc *sc,
                 struct eth_drv_sg *sg_list,
                 int sg_len);
    //Deliver data to/from device from/to stack memory space
    //(moves lots of memcpy()s out of DSRs into thread)
    void (*deliver)(struct eth_drv_sc *sc);
    //Poll for interrupts/device service
    void (*poll)(struct eth_drv_sc *sc);
    //Get interrupt information from hardware driver
    int (*int_vector)(struct eth_drv_sc *sc);
    //Logical driver interface
    struct eth_drv_funs *eth_drv, *eth_drv_old;
};
```

一个驱动程序实例(instance)包含在下面的数据结构 eth_drv_sc **内：**

```
struct eth_drv_sc {
    struct eth_hwr_funs *funs;
    void                *driver_private;
    const char          *dev_name;
    int                 state;
    struct arpcom       sc_arpcom;/* ethernet common */
};
```

如果同一硬件设备有两个实例,则只需要提供一个 eth_hwr_funs **结构,同一硬件设备的不同驱动程序实例可以共享该结构。**

上面的 eth_hwr_funs **结构只提供了对设备进行操作的函数,为了能使驱动程序与网络协议栈的其余部分通信(高层驱动程序),还应该提供包括这些通信函数在内的另一个数据结构** eth_drv_funs **：**

```
struct eth_drv_funs {
    //Logical driver - initialization
    void (*init)(struct eth_drv_sc *sc,
                 unsigned char *enaddr);
    //Logical driver - incoming packet notifier
    void (*recv)(struct eth_drv_sc *sc,
                 int total_len);

    //Logical driver - outgoing packet notifier
    void (*tx_done)(struct eth_drv_sc *sc,
                    CYG_ADDRESS key,
```

```
                    int status);
};
```

底层驱动程序中不需要产生该结构的实例。头文件 eth _ drv. h 具有该结构的定义，驱动程序可以直接使用它。

与网络协议栈进行通信所需要的另外一个 API 函数是：

```
extern void eth _ drv _ dsr(cyg _ vector _ t vector,
                            cyg _ ucount32 count,
                            cyg _ addrword _ t data);
```

该函数被当作中断处理程序的 DSR (滞后服务程序) 被注册，它将唤醒“网络传输线程”来调用驱动程序的传输程序。

使用宏 ETH _ DRV _ SC() 可以产生数据结构 eth _ drv _ sc 的一个实例，该结构包括了一些函数原型。如果要对以太网驱动程序进行修改，比如需要增加新的函数时，在对驱动程序进行更新之前不需要对原来的驱动程序进行重新编译，其他所有函数不会因为接口的变化而受到影响。

以太网驱动程序的启动类似于 IO 子系统。使用宏 NETSEVTAB _ ENTRY() 可以启动其驱动程序，该宏定义了一个初始化函数和底层驱动程序的基本数据结构。宏的定义如下：

```
typedef struct cyg _ netdevtab _ entry {
    const char          * name;
    bool                (* init)(struct cyg _ netdevtab _ entry * tab);
    void                * device _ instance;
    unsigned long       status;
} cyg _ netdevtab _ entry _ t;
```

该结构中的 device _ instance 指向前面已经定义的 eth _ drv _ sc 结构。这种实现方式使得网络驱动程序可以被设置为任何类型的驱动程序，不仅仅是以太网驱动程序。

综上所述，以太网驱动程序可以使用下述模板来产生其一个实例并挂接到系统上：

```
#include <cyg/infra/cyg _ type. h>
#include <cyg/hal/hal _ arch. h>
#include <cyg/infra/diag. h>
#include <cyg/hal/drv _ api. h>
#include <cyg/io/eth/netdev. h>
#include <cyg/io/eth/eth _ drv. h>
ETH _ DRV _ SC(DRV _ sc,
               0,                   //No driver specific data needed
               "eth0",              //Name for this interface
               HRDWR _ start,
               HRDWR _ stop,
               HRDWR _ control,
               HRDWR _ can _ send
               HRDWR _ send,
               HRDWR _ recv,
```

```
                    HRDWR _ deliver,
                    HRDWR _ poll,
                    HRDWR _ int _ vector
);
NETDEVTAB _ ENTRY (DRV _ netdev,
                    "DRV",
                    DRV _ HRDWR _ init,
                    &DRV _ sc) ;
```

这样,剩下的工作是实现驱动程序中所涉及的对设备进行具体操作的函数,下一节将介绍这些函数的实现方法。

如果要使用同一个底层驱动程序来处理多个类似的硬件接口,则必须多次使用这两个宏:ETH _ DRV _ SC ()、NETDEVTAB _ ENTRY ()。如果具有与该实例相关的专用数据(如基地址、中断号等),可以在上面模板中 ETH _ DRV _ SC () 的第二个参数处(上例为 0)加一个指向专用数据的指针。

10.2.2 驱动程序内部函数的实现

使用上面的模板只是形成一个以太网驱动程序的基本框架,还必须实现对设备进行具体操作的函数。下面将介绍各种函数的实现功能,在具体实现时要使用具体设备相应的函数名,避免系统中出现相同的函数定义。

init 函数

函数原型:

```
static bool DRV _ HRDWR _ init (struct cyg _ netdevtab _ entry * tab)
```

该函数作为系统初始化的一部分,在系统初始化期间被调用。其主要功能是判断设备是否正在工作,确定是否需要将设备接口变为可用的状态。如果需要,那么该函数必须以下面的形式调用以太网驱动程序的内部 init 函数:

```
struct eth _ drv _ sc * sc = (struct eth _ drv _ sc *) tab->device _ instance;
    .... initialization code....
    //Initialize upper level driver
        (sc->funs->eth _ drv->init) (sc, unsigned char * enaddr) ;
```

其中参数 enaddr 为网卡的 MAC 地址指针,用于通知上层网络协议栈该设备已经准备就绪。MAC 地址是标识网卡的惟一地址,一般保存在网卡的 ROM 内。某些平台的 RedBoot 提供了对其进行设置的 fconfig 命令,另外在 CDL 脚本或 EPROM 中也可以对它进行指定。驱动程序一般先选择 RedBoot 运行时指定的 MAC 地址。如果 RedBoot 没有指定,则选用 CDL 指定的地址,最后才选择 EPROM 设定的地址。

start 函数

函数原型:

```
static void HRDWR _ start (struct eth _ drv _ sc * sc,
                    unsigned char * enaddr,
```

```
int flags)
```

在系统初始化完成后,通常在系统或应用程序准备激活网络接口的时候调用该函数。其目的是对硬件接口进行设置,以便启动网络数据包的接收和发送。在调用该函数之前,不应该使能网络硬件接收器。

在网络接口逻辑发生变化时,该函数也将被调用。当 IP 地址被改变的时候,或者在应用程序通过 ioctl()对方式进行选择的时候都需要调用此函数。这种调用在系统中可能会多次发生。

stop 函数

函数原型:

```
static void HRDWR_stop(struct eth_drv_sc *sc)
```

该函数所进行的操作与 start 函数相反,它关闭网络接口硬件,阻止系统与网络设备之间的交互通信。

control 函数

函数原型:

```
static int HRDWR_control(
        struct eth_drv_sc *sc,
        unsigned long key,
        void *data,
        int len)
```

该函数用于对网络设备接口进行底层的控制操作。这种操作通常由 ioctl()函数(BSD 协议栈内)启动,可以改变对硬件的设置。其参数 key 指定所选择的操作,data 和 len 是该操作所需的数据和数据长度。

key 值可选择如下操作:

```
ETH_DRV_SET_MAC_ADDRESS
```

设置网络设备的 MAC 地址。该地址是惟一的,通常保存在非易失性内存中。进行该操作时,该函数必须使用新的 MAC 地址对网络接口进行设置,可以根据硬件的需要对非易失性内存进行更新。

```
ETH_DRV_GET_IF_STATS_UD,
ETH_DRV_GET_IF_STATS
```

这两个操作用于获取网络接口的统计信息,并将这些信息写入 data 指定的内存内。"UD"指明获取最新的统计消息。这种统计操作的完成可能需要一定的时间。

```
ETH_DRV_SET_MC_LIST
```

这种操作对设备进行设置,使其对多点传送(multicast)的网络数据进行筛选,只接收那些对参数 data 中所列的多点发送 MAC 地址进行寻址的数据包。data 包含了所列 MAC 地址的数目和具体的 MAC 地址,其格式定义如下:

```
#define ETH_DRV_MAX_MC 8
```

```
struct eth_drv_mc_list {
    int len;
    unsigned char addrs[ETH_DRV_MAX_MC][ETHER_ADDR_LEN];
};
ETH_DRV_SET_MC_ALL
```

这种操作指示网络设备接收所有的多点发送数据包,不使用筛选机制。

如果指定的操作成功完成,该函数返回 0,否则将返回一个非 0 值。

can_send 函数

函数原型:

```
static int HRDWR_can_send(struct eth_drv_sc *sc)
```

该函数用于判断网络接口是否可以启动一个数据包的发送。有些网络接口允许多个数据包排队等待,这一函数可以用于这种方式。

该函数的返回值表示可接受的包的个数,返回 0 时表示网络接口处于忙(饱和)状态。

send 函数

函数原型:

```
static void HRDWR_send(
                struct eth_drv_sc *sc,
                struct eth_drv_sg *sg_list,
                int sg_len,
                int total_len,
                unsigned long key)
```

其中 eth_drv_sg 结构定义为:

```
struct eth_drv_sg {
    CYG_ADDRESS buf;
    CYG_ADDRWORD len;
};
```

该函数用于发送一个网络数据包。具体如何向网络发送数据包正是该函数所需要做的工作。大部分硬件在进行发送操作时只需要被传送数据块的地址和长度值。

网络驱动程序支持数据的 scatter-gather 操作。被发送和接收的所有数据都通过一个 scatter-gather 表来指定,scatter-gather 表是一个描述多段被传送数据的表,是一个包含数据块地址和长度的数组。上面定义的 eth_drv_sg 就是该数组元素的定义。

一旦数据发送成功或者发生错误,驱动程序都将调用(sc->funs->eth_drv->tx_done)(),调用时将使用指定的 key 值。只有在高层驱动程序释放该数据包的资源时才能启动下一次发送操作。

deliver 函数

函数原型:

```
static void HRDWR_deliver(struct eth_drv_sc *sc)
```

网络传输线程调用该函数使设备驱动程序接收数据包。数据包的接收是一种耗时的操作,通常要从硬件或指定的某个内存区将整个包复制到网络栈内存区。在对输入包或者悬挂的发送状态进行处理之后,它可以解除对设备中断的屏蔽并释放相关的资源,从而可以对更多的包进行处理。

网络设备中断处理程序调用 eth _ drv _ dsr (vector, count, (cyg _ addrword _ t) sc) 函数提醒系统对某些事情进行关注时,将调用 deliver 函数。eth _ drv _ dsr 函数的调用必须是在中断处理程序的 DSR 内,不能在 ISR 内。eth _ drv _ dsr 函数的第三个参数 sc 必须是一个有效的 eth _ drv _ sc 结构的指针。

为了将一个新收到的包交付给网络栈,deliver 函数必须调用 (sc->funs->eth _ drv->recv) (sc, len) 函数,这个函数又将调用 receive 函数。

receive 函数

函数原型:

```
static void HRDWR _ recv (
                struct eth _ drv _ sc * sc,
                struct eth _ drv _ sg * sg _ list,
                int sg _ len)
```

这是一个回调函数,只有在 deliver 函数知道网络接口中有可用的数据包时,在调用高层函数 (sc->funs->eth _ drv->recv) (struct eth _ drv _ sc * sc, int total _ len) 之后才调用这个函数。sc->funs->eth _ drv->recv) () 函数在为数据准备好网络传输 buffer 和相关数据结构之后,调用 HRDWR _ recv () 从网络接口读取数据。

在使用该函数时,也可以实现 scatter-gather 操作,scatter _ gather 表的结构 eth _ drv _ sg 将被使用多次。

poll 函数

函数原型:

```
static void HRDWR _ poll (struct eth _ drv _ sc * sc)
```

该函数用于中断被完全禁止的系统,它允许驱动程序以查询方式来确定是否需要进行数据的发送和接收操作,或者是否需要进行其他的处理操作。该函数的一种比较好的实现方法如下:

```
static void
HRDWR _ poll (struct eth _ drv _ sc * sc)
 {
     my _ interrupt _ ISR (sc) ;
     HRDWR _ deliver (struct eth _ drv _ sc * sc) ;
}
```

中断向量函数

函数原型:

```
static int HRDWR _ int _ vector (struct eth _ drv _ sc * sc)
```

该函数将返回接收网络中断所使用的中断向量。主要用于 GDB 对网络的调试。GDB 在

对中断进行屏蔽操作时，需要知道以太网设备所使用的是哪个中断。

10.2.3 高层驱动程序函数

驱动程序在进行包的接收和发送时需要调用高层驱动程序函数。高层驱动程序与硬件没有直接联系，它所提供的函数可以不加修改地用于不同的网络协议栈。这些函数都需要有一个对网络接口进行描述的 eth _ drv _ sc 结构的指针。底层硬件驱动程序拥有该指针，在适当的时候把该指针传送到高层驱动程序。

init 回调函数

函数原型：

```
void (sc->funs->eth _ drv->init)(
        struct eth _ drv _ sc * sc,
        unsigned char * enaddr)
```

该函数在初始化期间对设备进行设置。对于每一个设备实例，它只能被初始化函数调用一次。该函数在被调用的时候，硬件应该已经完成了全部的初始化工作。

tx _ done 回调函数

函数原型：

```
void (sc->funs->eth _ drv->tx _ done)(
        struct eth _ drv _ sc * sc,
        unsigned long key,
        int status)
```

在一个包的发送操作完成后，该函数将被调用。其中参数 key 必须是上一节介绍的 HRDWR _ send() 函数所提供的 key 值。如果发送操作有错误，则 status 为非 0 值，如果发送成功，其值则为 0。

底层驱动程序函数 deliver 和 poll 将调用该函数。

receive 回调函数

函数原型：

```
void (sc->funs->eth _ drv->recv)(
        struct eth _ drv _ sc * sc,
        int len)
```

该函数被调用时，说明长度为 len 的包已经到达了网络接口。前面介绍的 HDRWR _ recv() 函数将这些数据从该接口读入到高层驱动程序所使用的 buffer。

10.2.4 数据的发送和接收过程

进行网络驱动程序设计时，必须详细了解数据发送和接收的整个控制过程。在对数据发送和接收进行控制的过程中，驱动程序将使用中断。在发生异步事件时，中断滞后服务程序 DSR 将这些异步事件通知前台，请求对这些事件进行相应的处理。

1. **发送过程**

在前台运行的任务可以使用网络协议栈来发送数据包，这些前台任务包括应用程序、SN-

MP 的 daemon 程序、DHCP 管理线程等等，它们也可以使用网络协议栈来发送诸如“ping”和“ARP”这样的请求应答包。驱动程序在进行包的发送时，其主要流程如下：

1）驱动程序调用底层 HRDWR _ can _ send () 函数，确定是否可以进行发送操作。

2）HRDWR _ can _ send () 返回可用的“slots”数目，一个“slots”可以存放一个未发送的包。如果此时不能发送，该包将在硬件驱动程序外排队等待。在这种情况下，硬件正处于发送忙状态，因此希望在当前包流出时有中断产生。

3）如果此时可以发送包，则调用 HRDWR _ send () 函数将数据复制到指定的硬件 buffer 内，或者指示硬件将其发送。此时要记住该发送请求的 key 值。

4）在发送完成后，硬件发出一个异步中断。中断服务程序 ISR 响应中断，识别硬件中断源，并请求相应的中断滞后服务程序 DSR 运行。

5）DSR 调用 eth _ drv _ dsr () 函数（它可能就是 DSR 本身）。

6）eth _ drv _ dsr () 函数唤醒网络传输线程，该线程调用 HRDWR _ deliver () 发送函数。

7）发送函数得知一个发送请求完成后，调用 tx-done 回调函数（sc->funs->eth _ drv-> tx _ done）()，使用第 3 步所使用的 key 值。

8）tx-done 回调函数使用 key 值来查找此次发送请求相关的信息。这样，网络栈可以知道此次发送已经完成，并释放其资源。

9）tx-done 回调函数还将调用 HRDWR _ can _ send () 函数询问此时是否可以进行发送操作，如果可以，则从队列中取下一个发送请求。然后再调用 HRDWR _ send () 函数将数据复制到硬件 buffer 或指示硬件进行发送，这一过程正是前面步骤的重复。这些调用最后都将返回到网络传输线程，在包的发送操作完成后，该线程将进入睡眠状态，等待下一个异步事件的发生。

2. 接收过程

当网络上有包从外部进入硬件 buffer 的时候，将产生一个异步中断，宣告接收 buffer 中有接收数据。中断服务程序 ISR 在对中断源进行处理后，将请求运行中断滞后服务程序 DSR。包的接收过程如下：

1）ISR 请求 DSR 运行，DSR 将调用 eth _ drv _ dsr () 函数（它可能就是 DSR 本身）。

2）eth _ drv _ dsr () 函数唤醒网络传输线程，该线程调用 HRDWR _ deliver () 传送函数。

3）传送函数识别出已准备就绪的数据，然后调用 receive 回调函数（sc->funs-> eth _ drv->recv）()，告诉它有多少字节已经准备好。

4）receive 回调函数在栈内分配内存，并为包的接收准备一组 scatter-gather 缓冲区。

5）然后调用 HRDWR _ recv () 函数将数据从硬件 buffer 复制到第 4 步操作所提供的 scatter-gather 缓冲区，并返回。

6）至此，网络栈已经得到了接收数据，此后它可以进行其他的工作。这些工作包括用于发送一个响应包的一组递归调用（包的发送过程）。当所有这些工作完成后，网络传输线程将再次进入睡眠状态，等待下一次异步事件的发生。

10.3 TCP/IP 协议栈支持

eCos 完全支持 TCP/IP 网络协议栈。这种支持来源于 FreeBSD 和 OpenBSD 的 TCP/IP

协议栈，具有相当强的完整性和稳健性。eCos 使用两个软件包分别实现 FreeBSD 和 OpenBSD 的 TCP/IP 协议栈，它们分别是“FreeBSD Stack”包（“net”模板）和“OpenBSD Stack”包（“old_net”模板）。在使用 eCos 图形配置工具进行配置时，可以根据需要选择这两个包。

10.3.1 特性支持与配置

eCos 提供的两个 TCP/IP 网络协议栈包对 TCP/IP 的支持有一定的区别，对它们的配置操作也略有不同。

1. FreeBSD Stack

FreeBSD Stack 对下述协议提供支持：

① IPv4。

② UDP。

③ TCP。

④ ICMP。

⑤ 原始信报接口（Raw Packet Interface）。

⑥ 多点广播（Multi-cast）寻址。

⑦ IPv6（包括 UDP、TCP、ICMP）。

出现在 FreeBSD Stack 包内但不支持的特性有：

① Berkeley Packet 筛选器。

② 点点传送（uni-cast）。

③ 多点广播路由（routing）。

在使用配置工具对 FreeBSD Stack 进行配置时，在配置工具中选择“Build→Packages”，将出现一个“Packages”操作对话框，如图 10-4 所示。

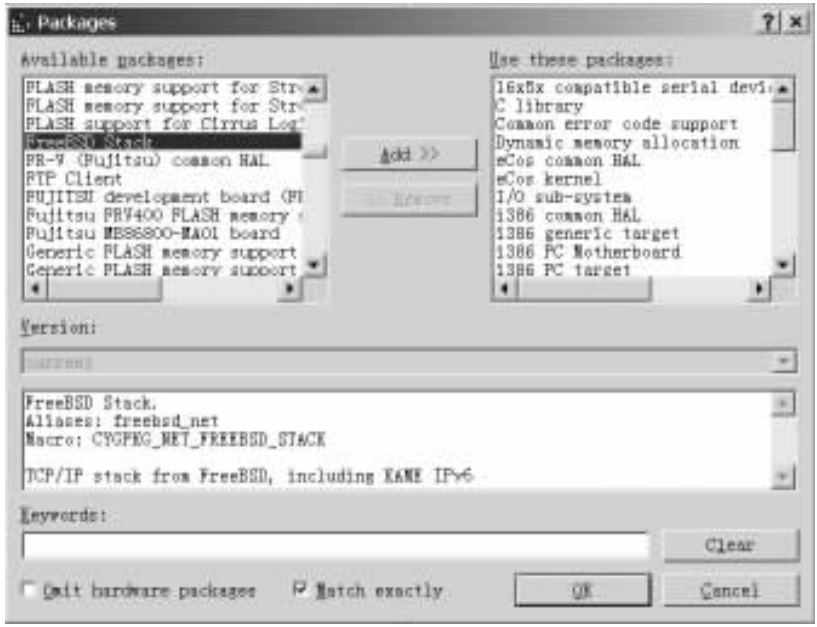


图 10-4 对 TCP/IP 协议栈的配置

在配置中选择并增加“Networking”、“FreeBSD Stack”和“Common Ethernet Support”三个包，它们的名字分别是 CYGPKG _ NET，CYGPKG _ NET _ FREEBSD _ STACK 和 CYGPKG _ NET _ ETH _ DRIVERS。所有这些操作可以使用一种简捷的方式实现，这种简捷方式就是选择“net”模板（使用配置工具的菜单选项“Build→templates”），如图 10-5 所示。这种模板已经包含了网络支持所需要的软件包，另外可能还包含了典型的网络设备驱动程序和相应的 PCI I/O 子系统。如果是新的网络设备，则应将相应的驱动程序加入到配置中。

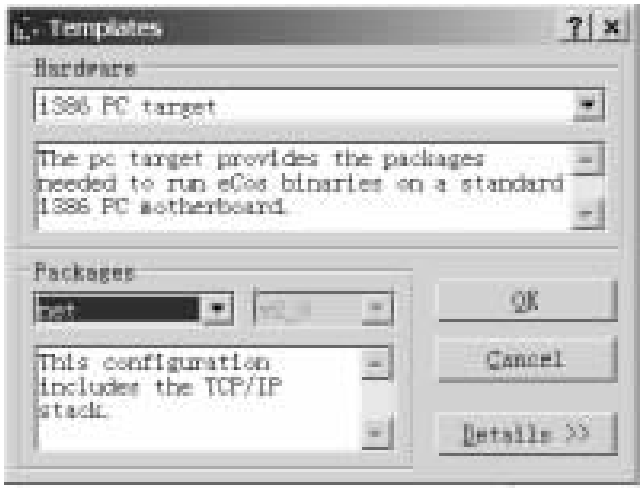


图 10-5　选择包含 TCP/IP 支持的模板

2. OpenBSD Stack

OpenBSD Stack 包所支持的协议有：

① IPv4。

② UDP。

③ TCP。

④ ICMP。

⑤ 原始信报接口。

它还包含了其他一些目前还不支持的特性：

① Berkeley Packet 筛选器。

② 多点广播和点点传送，包括多点广播路由。

③ IPv6。

OpenBSD Stack 包的配置与 FreeBSD Stack 类似。只是在“Packages”对话框中选择的是“OpenBSD Stack”包，它的名字是 CYGPKG _ NET _ OPENBSD _ STACK。在选择模板时要选择“old _ net”模板，而不是“net”。

10.3.2　API 函数

eCos 对 TCP/IP 协议栈的支持来源于 FreeBSD 和 OpenBSD，因此它所提供的也是标准的 API 函数。在使用这些 API 函数进行网络编程时，应该注意对它们进行定义的头文件位置。头文件位置也符合传统的编程习惯。下面是程序中包含这些头文件的举例：

install/include/arpa/tftp.h
install/include/netinet/tcpip.h
install/include/sys/socket.h
install/include/sys/socketvar.h
install/include/sys/sockio.h

下面是部分标准 TCP/IP 网络协议 API 函数的简单介绍，读者可以参考其他相关书籍或资料对它们进行详细了解。

accept　　接受 socket 上的连接
bind　　给 socket 绑定一个名字
close　　关闭 socket
connect　　初始化 Socket 连接
gethostbyname，gethostbyaddr　　获取主机信息
getpeername　　获取 socket 另一端的名字
getprotobyname，getprotobynumber　　获取网络协议信息
getservbyname，getservbyport　　获取 server 信息
getsockname　　获取 socket 的名字
getsockopt　　获取 socket 相关选项信息
inet _ aton，inet _ addr，inet _ addr　　Internet 地址操作函数
ioctl　　对设备进行控制
listen　　监听 socket 上的连接
read　　从 socket 接收信息
recvfrom　　从 socket 接收信息
select　　多路 I/O 同步
sendto　　从 socket 发送信息
setsockopt　　对 socket 相关的选项进行设置
shutdown　　关闭全双工连接
socket　　建立一个通信端点
write　　向 socket 发送信息

eCos 对标准的 select() 函数做了扩展。标准 select() 函数不支持重启（restart）功能，eCos 提供的扩展函数允许所有正在等待的线程即使在 select 条件未被满足时也可以被重启。这个被扩展的函数是：

```
int cyg_select_with_abort(
        int nfd,
        fd_set *in,
        fd_set *out,
        fd_set *ex,
        struct timeval *tv)
```

该函数完全具备标准 select() 函数的功能。此外，它还提供另一个功能，即调用下面的函数：

```
void cyg_select_abort(void)
```

该函数引起所有在 cyg _ select _ with _ abort () 函数调用中正在等待的线程终止等待过程，开始继续执行。

10.4 FTP 客户端

eCos 的网络包实现了一个 FTP 客户端。它提供了一些 FTP 客户端 API 函数，这些 API 函数定义于头文件 install/include/ftpclient.h 内。使用这些 API 时，应该包含下述头文件：

```
#include <network.h>
#include <ftpclent.h>
```

ftpclient.h 定义了三个 FTP 客户端 API 函数，它们分别是 ftp _ get、ftp _ put 和 ftpclient _ printf。

```
int ftp _ get (char * hostname,
               char * username,
               char * passwd,
               char * filename,
               char * buf,
               unsigned buf _ size,
               ftp _ printf _ t ftp _ printf) ;
```

ftp _ get 函数使用 FTP 协议从某个服务器获取文件，它只支持二进制方式。hostname 是 FTP 服务器名字或 IP 地址，username 是连接 FTP 服务器的用户名，passwd 为登录口令。参数 filename 指定了文件名，它可以包含文件的目录路径，但只能使用 UNIX 文件隔离符“/”，不能使用“\”。ftp _ printf 为输出信息时调用的函数。文件被存放到参数 buf 指定的缓冲区内，参数 buf _ size 指定了最大 buf 值。如果文件比 buf 大，文件传输将失败，该函数返回 FTP _ TOOBIG。如果文件传输成功，则返回接收到的字节数。

```
int ftp _ put (char * hostname,
               char * username,
               char * passwd,
               har * filename,
               char * buf,
               unsigned buf _ size,
               ftp _ printf _ t ftp _ printf) ;
```

ftp _ put 函数用于向某个 FTP 服务器发送文件，只支持二进制方式。与 ftp _ get 函数一样，hostname 是 FTP 服务器名字或 IP 地址，username 是连接 FTP 服务器的用户名，passwd 为登录口令。参数 filename 指定了文件名，它可以包含文件的目录路径，但只能使用 UNIX 文件隔离符“/”，不能使用“\”。buf 内容为传送给服务器的文件内容。ftp _ printf 为输出信息时调用的函数。如果文件传送失败，将返回一个错误码，如果成功则返回 0。

```
void ftpclient _ printf (unsigned error,
                         const char * fmt, ...) ;
```

ftp _ get 和 ftp _ put 两个函数都具有一个函数指针参数 ftp _ printf，它用于输出诊断信息和错误信息，ftpclient _ printf 就是这种函数的具体实现。eCos 提供的该函数只是一种参考实现，如果需要，开发人员可以根据实际情况修改这一函数。当输出错误消息时，它的第一个参数 error 为 true。

FTP 客户端程序编程例子可以参阅 eCos 源码提供的 FTP 测试程序 packages \ net \ ftpclient \ v2 _ 0 \ tests \ ftpclient1.c。

10.5 DNS 客户端

eCos 和 RedBoot 都可以使用 DNS 包提供的 DNS API 函数来寻找网络名字。DNS 客户端使用标准 BSD API 函数 gethostbyname () 和 gethostbyaddr ()。这两个函数的使用具有下面的限制：

① 只支持 IPv4，不能查找 IPv6 地址。

② 如果 DNS 服务器返回一个主机名的多条记录，那么 hostent 结构只包含第一条记录。

③ 这些函数具有线程安全性，可以多个线程调用 gethostbyname () 函数。同一个线程都调用这两个函数则是不安全的，一个函数调用返回的结果将破坏在该函数之前使用另一个函数返回的结果。

在使用 DNS 服务时，首先要使用下述方式对 DNS 客户端进行初始化：

```
#include <network.h>
int cyg _ dns _ res _ init (struct in _ addr * dns _ server)
```

其中 dns _ server 为 DNS 服务器的地址。当有错误发生时将返回-1，否则返回 0。在该初始化函数之前调用的查找函数将失败并返回 NULL。

在默认情况下，通过 CDL 选项 CYGDAT _ NS _ DNS _ DEFAULT _ SERVER 可以强制指定 DNS 服务器的地址。配置选项 CYGPKG _ NS _ DNS _ DEFAULT 用于控制该地址的使用，当该选项被使能时，init _ all _ network _ interfaces 将使用指定的 DNS 服务器进行初始化。DHCP 客户端和用户程序可以使用 cyg _ dns _ res _ init 函数来重新指定 DNS 服务器的地址。

DNS 客户端认为目标系统是在同一个域 (domain) 内，默认情况下不使用域。Host 名字的查找使用的是全名。域名的设置和提取可以使用下面的函数：

```
int getdomainname (char * name, size _ t len) ;
int setdomainname (const char * name, size _ t len) ;
```

此外，还可以使用 CDL 来强制指定域名。配置选项 CYGPKG _ NS _ DNS _ DOMAINNAME 将使能这种 CDL 指定的域名，而域名可以从 CYGPKG _ NS _ DNS _ DOMAINNAME _ NAME 选项中获取。这种方式可以使用图形配置工具进行 DNS 客户端的设置，如图 10-6 所示。

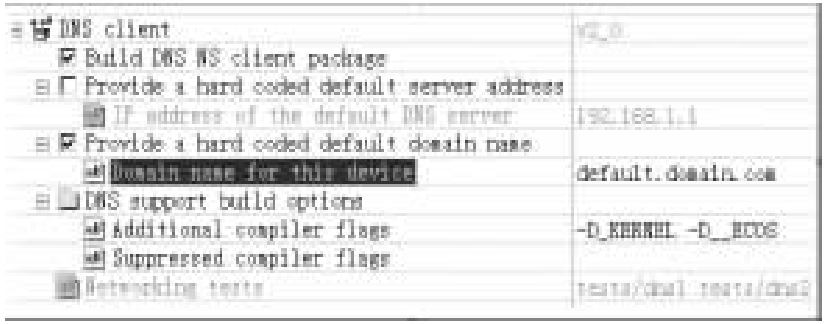


图 10-6　DNS 客户端设置

10.6　eCos 网络编程实例

eCos 源码中提供了许多对网络通信进行测试的程序。这些测试程序不仅仅是为了测试目标平台网络通信的正确性，它的另一个主要目的是提供应用程序网络编程的框架和实例。在编写网络应用程序时，可以参考相应的测试例子程序。另外它还提供了一组性能测试程序，可以用于对嵌入式 eCos 系统的网络吞吐率和延迟进行测试。

10.6.1　网络通信测试程序

在对 eCos 进行网络配置并编译后，需要运行测试程序来测试网络通信的正确性。下面将按照对一个新的 eCos 系统进行网络通信验证的顺序简要介绍一些主要的测试程序。

值得注意的是，在默认配置中不会对这些测试程序进行编译。如果要编译这些测试程序，必须在配置工具中使能 CYGPKG _ NET _ BUILD _ TESTS 配置选项，并在配置工具中选择“Build→Tests”菜单对它们进行编译。另外，这些测试程序还要求硬件接口被配置为使用 BOOTP 或静态配置的初始化方式。

mbuf _ test.c

这是在新系统中运行的第一个测试程序。它的测试内容是网络系统能否正确初始化、协议栈使用的内部内存管理功能是否正确。

socket _ test.c

该测试程序对基本的网络 API 函数接口进行测试。

server _ test.c

该测试程序在目标平台上产生一个服务器程序，监听 TCP 端口 7734。为验证其是否能正常工作，可以在其他主机上尝试对它进行连接。例如，在 Linux 主机上执行命令“telnet eCos 7734”，其中“eCos”是目标系统的名字。一旦连接成功，eCos 应用程序将发出一个“Hello”的响应消息，并等待输入一行命令，命令行将显示在目标系统的调试通道（通常为串口）上。

ping _ test.c

该测试程序将给另外一台“服务器”主机发出一个 ICMP 请求包“echo”。其输出信息类似于 Linux 的“ping”命令。另外，该测试程序还将对另一个伪造的 IP 地址进行 ping 测试，该地址是服务器 IP + 32（假设该 IP 地址不存在），这种测试的主要目的是验证 ICMP 的超时机制。

ftp _ test.c

该测试程序尝试与另一台 FTP 服务器进行连接。它用于验证基本的 TCP 功能。

nc _ test _ master.c 和 nc _ test _ slave.c

这两个测试程序可以用于对目标系统的网络吞吐率和延迟进行测试。它们分别运行于 eCos 目标系统和 Linux 主机上(分别在各自环境下进行编译),一般在 Linux 主机上运行 master 程序,eCos 目标系统上运行 slave 程序。在这种测试环境下,master 程序将尝试与 slave 程序进行连接,并使用 UDP 和 TCP 协议进行各种测试。另外,在 eCos 目标系统上运行的 slave 程序还测试 CPU 在网络通信上的利用率。在 Linux 主机上进行编译时,可以在 eCos 的源码目录(不能在安装目录 install)下执行下面的命令进行编译:

```
make -f make.linux
```

tcp _ echo.c、tcp _ sink.c 和 tcp _ source.c

这一组测试程序类似于上面介绍的 nc _ test _ XXX 程序。它们对 eCos 系统进行全面的吞吐率测试。这一组测试需要两台 Linux 主机,一台 Linux 主机运行"tcp _ source eCos",另一台运行"tcp _ sink eCos",而 eCos 目标平台上运行"tcp _ echo"程序。这种测试环境将测试从一台 Linux 主机上发送 TCP 数据到 eCos、再由 eCos 将数据转发到另一台主机的吞吐率和延迟。也可以使用同一台 Linux 主机,这时将变成单向 echo 测试。这种测试环境是实际网络通信环境的一种模拟。

dhcp _ test.c

这是一个简单的 ping 测试,但可以进行 DHCP 出租操作。它非常类似于 ping 测试程序,用于对 DHCP 操作的外部 API 函数进行测试。

tftp _ client _ test.c 和 tftp _ server _ test.c

tftp 是一个简单的文件传输协议。这一组测试程序用于进行 tftp 测试。一般 tfpt _ server _ test.c 运行于 Linux 主机上,tftp _ client _ test.c 运行在 eCos 目标系统上。在进行测试前,必须对 tftp 服务器进行设置,使其能正常工作。tftp 服务器通常在 Linux 的/tftpboot/目录下操作,在该目录下准备一个用来进行传输测试的文件 tftp _ get(包含任意数据),并产生一个空文件 tftp _ put。运行在 eCos 上的 tftp _ client _ test.c 将从服务器上获取文件 tftp _ get,并将其更名为 tftp _ put 传送到 Linux 主机。Linux 主机上原来的空文件 tftp _ put 将被其覆盖。

10.6.2 编程实例——ping 程序

本节通过一个简单的 ping 程序例子来介绍 eCos 的网络编程方法。这是 eCos 源码中的一个例子程序,读者可以参考其他更多的程序实例。

```
//==========================================
//PING test code
#include <network.h>
```

(注:使用单个 network.h 头文件,它包含了所有需要的网络支持和配置信息。)

```
#define STACK _ SIZE CYGNUM _ HAL _ STACK _ SIZE _ TYPICAL
static char stack[STACK _ SIZE];
```

```
static cyg_thread thread_data;
static cyg_handle_t thread_handle;
#define NUM_PINGS 16
    #define MAX_PACKET 4096
    static unsigned char pkt1[MAX_PACKET], pkt2[MAX_PACKET];
    #define UNIQUEID 0x1234
    void cyg_test_exit(void) {
        diag_printf("... Done\n");
        while(1);
    }
    void pexit(char *s) {
        perror(s);
        cyg_test_exit();
    }
    //计算 INET 校验和
    int inet_cksum(u_short *addr, int len) {
        register int nleft = len;
        register u_short *w = addr;
        register u_short answer;
        register u_int sum = 0;
        u_short odd_byte = 0;
        /* Our algorithm is simple, using a 32 bit accumulator (sum),
        we add sequential 16 bit words to it, and at the end, fold
        back all the carry bits from the top 16 bits into the lower
        16 bits.
        */
        while(nleft > 1) {
            sum += *w++;
            nleft -= 2;
        }
            /* mop up an odd byte, if necessary */
            if(nleft == 1) {
                *(u_char *)(&odd_byte) = *(u_char *)w;
                sum += odd_byte;
            }
            /* add back carry outs from top 16 bits to low 16 bits */
            sum = (sum >> 16) + (sum & 0x0000ffff);/* add hi 16 to low 16 */
            sum += (sum >> 16);                    /* add carry */
            answer = ~sum;                         /* truncate to 16 bits */
            return(answer);
}
static int
show_icmp(unsigned char *pkt, int len,
```

```
                struct sockaddr_in *from, struct sockaddr_in *to)
{
    cyg_tick_count_t *tp, tv;
    struct ip *ip;
    struct icmp *icmp;
    tv = cyg_current_time();
    ip = (struct ip *)pkt;
    if ((len < sizeof(*ip)) || ip->ip_v != IPVERSION) {
        diag_printf("%s: Short packet or not IP! - Len: %d, Version: %d\n",
                    inet_ntoa(from->sin_addr), len, ip->ip_v);
        return 0;
    }
    icmp = (struct icmp *)(pkt + sizeof(*ip));
    len -= (sizeof(*ip) + 8);
    tp = (cyg_tick_count_t *)&icmp->icmp_data;
    if (icmp->icmp_type != ICMP_ECHOREPLY) {
        diag_printf("%s: Invalid ICMP - type: %d\n",
                    inet_ntoa(from->sin_addr), icmp->icmp_type);
        return 0;
    }
    if (icmp->icmp_id != UNIQUEID) {
        diag_printf("%s: ICMP received for wrong id - sent: %x, recvd: %x\n",
                    inet_ntoa(from->sin_addr), UNIQUEID, icmp->icmp_id);
    }
    diag_printf("%d bytes from %s:", len, inet_ntoa(from->sin_addr));
    diag_printf("icmp_seq=%d", icmp->icmp_seq);
    diag_printf(", time=%dms\n", (int)(tv - *tp)*10);
    return (from->sin_addr.s_addr == to->sin_addr.s_addr);
}
    static void
    ping_host(int s, struct sockaddr_in *host) {
        struct icmp *icmp = (struct icmp *)pkt1;
        int icmp_len = 64;
        int seq, ok_recv, bogus_recv;
        cyg_tick_count_t *tp;
        long *dp;
        struct sockaddr_in from;
        int i, len, fromlen;
        ok_recv = 0;
        bogus_recv = 0;
        diag_printf("PING server %s\n", inet_ntoa(host->sin_addr));
```

(注：inet_ntoa 函数将 IP 地址转换为字符串，可用于输出 IP 地址。)

```
for (seq = 0;seq < NUM_PINGS;seq++) {//Build ICMP packet
    icmp->icmp_type = ICMP_ECHO;
    icmp->icmp_code = 0;
    icmp->icmp_cksum = 0;
    icmp->icmp_seq = seq;
    icmp->icmp_id = 0x1234;
    //Set up ping data
    tp = (cyg_tick_count_t *)&icmp->icmp_data;
*tp++ = cyg_current_time();
    dp = (long *)tp;
    for (i = sizeof(*tp);i < icmp_len;i += sizeof(*dp)) {
        *dp++ = i;
      }
      //Add checksum
      icmp->icmp_cksum = inet_cksum((u_short *)icmp, icmp_len+8);
      //Send it off
if (sendto(s,icmp,icmp_len+8,0,(struct sockaddr *)host, sizeof(*host)) < 0)
{
```

(注:sendto 函数在此处使用 ICMP 协议发送一个包,目的地址由参数 host 指定。)

```
    perror("sendto");
    continue;
  }
    //Wait for a response
fromlen = sizeof(from);
len = recvfrom(s,pkt2,sizeof(pkt2),0,(struct sockaddr *)&from, &fromlen);
```

(注:recvfrom 函数等待包被发送到此接口。如果在给定时间范围内没有收到包,操作将失败,超时值由 ping_test()函数指定。)

```
if (len <0) {
        perror("recvfrom");
  } else {
        if (show_icmp(pkt2, len, &from, host)) {
            ok_recv++;
        } else {
            bogus_recv++;
        }
    }
  }
    diag_printf("Sent %d packets, received %d OK, %d bad\n", NUM_PINGS, ok_recv, bogus
_recv);
}
static void ping_test(struct bootp *bp) {
```

```
    struct protoent *p;
    struct timeval tv;
    struct sockaddr_in host;
    int s;
    if ((p = getprotobyname("icmp")) == (struct protoent *)0) {
```

(注:getprotobyname 函数获取 ICMP 协议的信息,在设置 socket 时将使用这些信息。)

```
        perror("getprotobyname");
        return;
    }
    s = socket(AF_INET, SOCK_RAW, p->p_proto);
```

(注:产生 socket "s",socket 是一个抽象的对象,也称为句柄或端点。socket 有多种类型,SOCK_RAW 用于非结构性协议通信,SOCK_DATAGRAM 用于包的导向协议如 UDP,SOCK_STREAM 用于可靠而有序的字节流,如 TCP。)

```
    if (s < 0) {
        perror("socket");
        return;
    }
    tv.tv_sec = 1;
    tv.tv_usec = 0;
    setsockopt(s, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));
```

(注:setsockopt 函数在此要求进行接收操作的系统(上面的 ping_host())必须在 1 s 内完成操作或发出一个超时错误。)

```
        //Set up host address
        host.sin_family = AF_INET;
        host.sin_addr = bp->bp_siaddr;
        host.sin_port = 0;
        ping_host(s, &host);
        //Now try a bogus host
        host.sin_addr.s_addr = htonl(ntohl(host.sin_addr.s_addr) + 32);
        ping_host(s, &host);
    }
```

(注:下面的函数开始进行 ping 测试。)

```
    void net_test(cyg_addrword_t p) {
        diag_printf("Start PING test\n");
        init_all_network_interfaces();
```

(注:这是一个对网络进行初始化的调用。它必须在线程环境下执行,因此不能将其放在下面的 cyg_start()函数中。init_all_network_interfaces()函数将引起网络通信系统的初始化,并对硬件接口进行设置。根据系统的配置信息,硬件接口可以使用 BOOTP 启动,或者使

用预先配置好的静态 IP 信息启动。它也可能指明将使用该函数以外的用户程序进行配置。)

```
#ifdef CYGHWR_NET_DRIVER_ETH0
if (eth0_up) {
        ping_test(&eth0_bootp_data);
    }
#endif
```

(注:所有网络硬件接口共有一个关于 BOOTP 信息的数据结构。除非选择手工配置,否则在调用 init_all_network_interfaces() 函数时应该产生这样一个数据结构。应用程序使用这些信息可以了解诸如本地 IP 地址、服务器名字之类的信息。)

```
#ifdef CYGHWR_NET_DRIVER_ETH1
if (eth1_up) {
        ping_test(&eth1_bootp_data);
    }
#endif
```

(注:根据系统硬件的不同配置,系统可以支持多个网络硬件接口。如果系统只使用单个以太网设备,那么应该定义 CYGHWR_NET_DRIVER_ETH0 。如果这是第二个接口,则应该定义 CYGHWR_NET_DRIVER_ETH1。)

```
    cyg_test_exit();
}
```

下面的函数创建一个运行该程序的初始线程。简单地将 net_test() 函数更名为 main() 也可以运行此程序。

```
void cyg_start(void) {
//Create a main thread, so we can run the scheduler and have time 'pass'
    cyg_thread_create(10,                   //Priority -just a number
                      net_test,             //entry
                      0,                    //entry parameter
                      "Network test",       //Name
                      &stack[0],            //Stack
                      STACK_SIZE,           //Size
                      &thread_handle,       //Handle
                      &thread_data          //Thread data structure
        );
    cyg_thread_resume(thread_handle);//Start it
    cyg_scheduler_start();
}
```

第 11 章　硬件抽象层与 eCos 移植

eCos 是一种可移植的嵌入式操作系统，它可以移植到 16 位、32 位以及 64 位的各种处理器和平台上。eCos 由各种组件构成，根据具体硬件平台的需要可以分别将这些组件加入到系统中来，从而实现各种所需的功能。eCos 的这种层次结构的最底层是硬件抽象层（Hardware Abstraction Layer），通常称为 HAL。硬件抽象层 HAL 对处理器结构和系统硬件平台进行抽象，当需要在一个新的目标平台上运行 eCos 时，只需对底层的硬件抽象层进行修改，便可迅速地将整个 eCos 系统移植到新的平台上。

11.1　硬件抽象层 HAL

硬件抽象层处于 eCos 层次结构中的最底层。根据所描述的硬件对象的不同，可以将硬件抽象层分成三个不同的子模块，它们分别是体系结构抽象层（Architecture HAL）、变体抽象层（Variant HAL）和平台抽象层（Platform HAL）。

第一个子模块是体系结构抽象层。eCos 所支持的不同处理器系列都具有不同的体系结构，如 ARM 系列、PowerPC 系列、MIPS 系列等等。体系结构抽象层对 CPU 的基本结构进行抽象和定义，此外它还包括中断的交付处理、上下文切换、CPU 启动以及该类处理器结构的指令系统等等。

第二个子模块是变体抽象层。变体指的是该处理器在该处理器系列中所具有的特殊性，这些特殊性包括在 Cache、MMU（内存管理部件）和 FPU（浮点部件）等方面与其处理器系列的基本结构具有的这样或那样的差异。eCos 的变体抽象层就是对这些特殊性进行抽象和封装。如果处理器具有片内（on-chip）内存和片内中断控制器，变体抽象层，也必须对它们进行处理。对于结构性的变体，实际上通常由体系结构抽象层来实现这种变体，变体抽象层只简单地提供适当的配置定义。

第三个子模块是平台抽象层。平台抽象层对当前系统的硬件平台进行抽象，包括平台的启动、芯片选择与配置、定时设备、I/O 寄存器访问以及中断寄存器等等。

硬件抽象层的这三个子模块之间没有很明显的界限。对于不同的目标平台，这种区分具有一定的模糊性。例如，Cache 和 MMU 可能在这个平台上是属于体系结构抽象层的范围，而在另一个平台上则可能属于变体抽象层的范围。同样，内存和中断控制器有可能是一种片内设备而属于变体抽象层，也有可能是片外设备而属于平台抽象层。

一般来说，目标系统应该将体系结构抽象层、变体抽象层和平台抽象层分别使用不同的包来加以实现。在一些早期的目标系统中或者在不需要的情况下，变体抽象层是不存在的。

eCos 在实现硬件抽象层时，采用了下述主要原则：

① 尽管大部分 eCos 内核都使用 C++，但其硬件抽象层 HAL 均用 C 语言和汇编语言加以实现。这使得 HAL 的适用范围更为广泛。

② 所有与 HAL 的接口均采用 C++ 宏（Macros）加以实现。采用这种方式的好处是可以

用内嵌 C 程序、内嵌汇编程序、外部 C 函数和外部汇编程序的形式对它们进行调用。同时，这种方式可以选择最有效的实现方法而不会影响到接口。在平台抽象层或变体抽象层需要对体系结构抽象层的定义进行更换或改进时，还可以采用这种宏定义的方式对它们进行重定义。

③ 硬件抽象层提供简单而具有可移植的机制来处理广泛范围内的处理器结构和硬件平台。虽然可以绕过硬件抽象层 (不使用硬件抽象层) 而直接对硬件进行操作，但这种对硬件直接操作的方式移植性较差。

eCos 的硬件抽象层提供了许多的配置选项，可以根据具体硬件平台的实际需要进行相应的配置。图 11-1 是使用图形配置工具对基于 i386 PC 的目标系统硬件抽象层 HAL 进行配置的示意图。

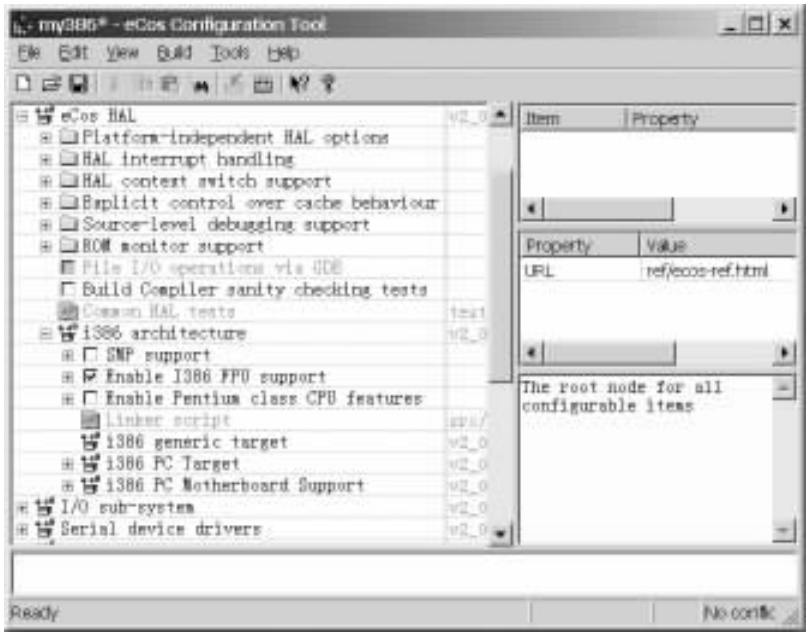


图 11-1　硬件抽象层的配置

利用配置工具可以对硬件抽象层进行下面几个方面的配置：

① Platform-independent HAL options。提供一些与平台硬件无关的配置选项，为所有平台的 HAL 包所共有。例如，对上下文切换过程中保存的状态量进行控制的选项。对于不同的体系结构，这些选项的实现不尽相同。

② HAL interrupt handling。提供与中断处理相关的一些配置选项，大多数的 HAL 包都共有这些选项。对于不同的硬件平台，它们的具体实现可能有差异。

③ HAL context switch support。提供一些与线程上下文相关的配置选项，大多数的 HAL 包都共有这些选项。对于不同的硬件平台，它们的具体实现可能有差异。

④ Explicit control over cache behaviour。提供一些对 Cache 进行控制的选项，使得对 Cache 行为的控制变得简单。

⑤ Source-level debugging support。对源码级调试工具 GDB 的支持进行配置。

⑥ ROM monitor support。对 ROM 监控程序进行配置。

⑦ i386 architecture。提供对 i386 体系结构的 HAL 包的配置。除了提供该体系结构一般的配置选项外,它还提供对具体目标平台专用的 HAL 包的配置。

11.2 硬件抽象层的结构

在开发一个新的 eCos 系统时,首先必须对硬件抽象层 HAL 的结构有所了解。开发人员必须针对目标系统硬件平台的特殊性而对硬件抽象层中相关部分进行必要的修改。同时,还应该了解 HAL 与系统其他部分之间的相互影响和关系。

11.2.1 HAL 的类型

上一节已经介绍了硬件抽象层的三个子模块。实际上,在具体的实现中通常可以将 HAL 分为五种类型,即公共抽象层(Common HAL)、体系结构抽象层、变体抽象层、平台抽象层以及辅助抽象层(Auxiliary HAL)。公共抽象层包含了所有结构和平台的硬件抽象层所共享的配置选项和函数,包括常用的调试功能、驱动程序 API、测试程序等等。辅助抽象层包含了处理器的一些变体所共享的公共模块,如 Motorola PowerPC 中的 QUICC 就是这样一个模块。表 11-1 对这五种类型的 HAL 进行了描述。

表 11-1 HAL 的五种类型

HAL 的类型	描 述	功 能
公共抽象层 (所在目录:hal/common)	所有类型的 HAL 共享的配置选项和函数	常用 debug 功能、驱动程序 API、eCos/ROM 监视调用接口、测试程序
体系结构抽象层 (所在目录: hal/<architecture>/arch)	该体系结构的特殊功能、在变体抽象层和平台抽象层中可以对其加以变化的一些功能的默认实现	体系结构特殊的 debug 功能(单步处理、例外信号转换等)、例外/中断向量的定义和处理程序、Cache 定义和控制、上下文切换程序、系统初期初始化汇编程序、配置选项、测试程序等
变体抽象层 (所在目录: hal/<architecture>/<variant>)	体系结构相同的 CPU 系列由一系列具有变体结构的 CPU 组成,如 MIPS 系列具有 32 位和 64 位不同版本的 CPU;另外还有一些嵌入到 CPU 内核的变体	变体对体系结构进行扩展的程序(Cache、例外/中断)、配置选项、片内设备的驱动程序、测试程序
平台抽象层 (所在目录: hal/<architecture>/<platform>)	包含平台专有的功能和配置选项	平台初期初始化程序、平台内存布局说明、配置选项(处理器速度、编译选项)、诊断 IO 函数、debug IO 函数、平台用于扩展体系结构和变体的程序、测试程序
辅助抽象层 (所在目录: hal/<architecture>/<module>)	一些变体所共享的公共模块(如 PowerPC QUICC)	特殊模块功能(中断控制器、简单设备驱动程序、测试程序)

11.2.2 硬件抽象层文件描述

在进行 eCos 的移植操作时,需要对构成硬件抽象层 HAL 的文件组织结构有所了解。本节主要对 eCos 硬件抽象层源码文件进行简单的介绍。

表 11-2 所列举的是公共抽象层的文件描述。

表 11-2 公共抽象层文件描述

文 件	描 述
include/dbg-thread-syscall.h	调试线程的系统调用函数定义。ROM 监视程序用它访问 RAM 应用程序中的调试线程 API 函数
include/dbg-threads-api.h	调试线程 API 定义
include/drv_api.h	驱动程序 API 定义
include/generic-stub.h	通用 stub 程序特性定义
include/hal_if.h	ROM/RAM 调用接口 API 定义
include/hal_misc.h	整个 HAL 所共享的各种各样的辅助函数
include/hal_stub.h	GDB stub 程序特性定义
src/dbg-threads-syscall.c	调试线程的实现
src/drv_api.c	驱动程序 API 的实现
src/dummy.c	空文件,保证编译时能生成 libtarget.a
src/generic-stub.c	通用 GDB stub 程序的实现。提供一个通过串口或网络与 GDB 进行通信的通信协议
src/hal_if.c	ROM/RAM 调用接口的实现
src/hal_misc.c	所有平台和体系结构所共享的各种各样的辅助函数
src/hal_stub.c	eCos HAL 所提供的通用 GDB stub 程序所需要的功能
src/stubrom/stubrom.c	用于生成 eCos GDB stub 程序映像。它是 ROM GDB stub 主程序,它简单地设置调试陷阱和断点,从而可以与 GDB 通信
src/thread-packets.c	为多线程调试提供支持
src/thread-pkts.h	线程调试相关函数定义

某些体系结构抽象层可能增加了一些源码文件用于支持其体系结构的特殊串口驱动程序,或者用于其中断和例外的处理。表 11-3 列举了体系结构抽象层的源码文件说明。这些源码文件中的许多定义是一种有条件的定义,如果在变体抽象层或平台抽象层中对它们进行了定义,则将替换体系结构抽象层中的相同定义。

有些变体抽象层可能会为该变体的特殊串口驱动程序或中断和例外的处理增加一些源码文件。表 11-4 为变体抽象层源码文件的描述。如果 CPU 的变体可以通过普通的体系结构宏定义进行控制,那么这些文件有可能大部分都是空文件。这些文件中的定义也是有条件的定义,如果平台抽象层的头文件也进行了相同的定义,则将替换这些在变体抽象层中的定义。

表 11-3　体系结构抽象层文件描述

文　件	描　述
include/arch.inc	系统初始化过程所使用的各种汇编宏定义
include/basetype.h	基本类型定义（Endian、lable、alignment、type size）。覆盖 CYGPKG_INFRA 的默认定义
nclude/hal_arch.h	寄存器格式定义，与线程、寄存器、堆栈相关的宏定义
include/hal_cache.h	Cache 定义和 Cache 控制宏定义
include/hal_intr.h	例外和中断定义，中断配置和控制宏定义，eCos 实时时钟控制宏定义
include/hal_io.h	IO 设备访问宏定义
include/<arch>_regs.h	体系结构寄存器定义
include/<arch>_stub.h	体系结构 stub 程序定义。特别是 GDB 所使用的寄存器结构定义，可能与 eCos 所使用的不同
include/<arch>.inc	体系结构宏定义
src/<arch>.ld	链接器宏定义
src/context.S	上下文处理和 setjump/longjump 的函数
src/hal_misc.c	例外和中断处理程序以及具体各种函数
src/hal_mk_defs.c	用于将 C 头文件的定义输出到汇编头文件
src/hal_intr.c	中断处理函数
src/<arch>stub.c	体系结构 stub 程序。某些将 eCos 例外转换为 UNIX 信号的函数，单步函数
src/vectors.S	例外、中断和早期初始化程序

表 11-4　变体抽象层文件描述

文　件	描　述
include/var_arch.h	寄存器格式，各种与线程、寄存器和堆栈相关的宏定义
include/var_cache.h	与 Cache 相关的宏定义
include/var_intr.h	与中断相关的宏定义
include/var_regs.h	CPU 变体的额外寄存器定义
include/variant.inc	系统初始化时所使用的各种汇编宏定义
src/var_intr.c	中断函数
src/var_misc.c	hal_variant_init 函数和其他函数
src/variant.S	中断处理程序表定义
src/<arch>_<variant>.ld	链接器宏定义

平台抽象层也可以增加一些源码文件，用于支持平台特有的串口驱动程序。如果需要，还可以增加中断和例外处理的文件。表 11-5 为平台抽象层的源码文件描述。

平台抽象层还包含了一些文件专门用于对目标系统平台的内存布局进行说明。这些文件位于 include/pkgconf。

表 11-5　平台抽象层文件描述

文　件	描　述
include/hal _ diag. h	HAL 诊断输出函数定义
include/platform. inc	平台初始化程序，包括内存控制器、向量、监视器初始化程序。根据具体的体系结构，还可以包含其他的定义 :中断译码、状态寄存器初始值等
include/plf _ cache. h	平台专用 Cache 处理
include/plf _ intr. h	平台专用中断处理
include/plf _ io. h	PCI I/O 定义和宏定义，如果平台的字节排列方式与 CPU 不同，这里还可以定义 IO 宏来替换普通 HAL 中 IO 宏
include/plf _ stub. h	Stub 初始化程序定义和复位定义
src/hal _ diag. c	硬件抽象层诊断输出函数。可以包含低级设备驱动程序，也可以放置到 plf _ stub. c 文件
src/platform. S	内存控制器设置宏定义。如果需要，可能包含有中断跳转程序
src/plf _ misc. c	平台初始化程序
src/plf _ mk _ defs. c	用于将 C 头文件的定义导出到汇编头文件
src/plf _ stub. c	平台专用的 stub 初始化程序和低级设备驱动程序

11.3　硬件抽象层接口

硬件抽象层 HAL 对底层的硬件进行抽象，它提供了对硬件操作的接口，上层系统可以使用 HAL 提供的接口函数来实现对硬件的操作和访问。HAL 的接口包括 CPU 体系结构、中断处理、I/O 操作、Cache 控制、诊断支持、SMP 支持等各个方面。这一节首先介绍硬件抽象层的一些基本定义，然后对硬件抽象层接口的各个方面进行介绍。

11.3.1　基本定义

在编译 eCos 内核的可移植部分时，需要用到与基本体系结构相关的一些特性定义，如数据类型、字节排列方式、标号方式等。这些定义位于头文件 cyg/hal/basetype. h，属于体系结构抽象层。该头文件自动包括在头文件 cyg/infra/cyg _ type. h 内。下面简单介绍这些定义。

(1) 字节排列方式定义。

```
CYG _ BYTEORDER
```

它对目标系统的字节排列方式进行定义，它的值必须是 CYG _ LSBFIST (little-endian 方式) 或 CYG _ MSBFIRST (big-endian 方式)。

(2) 标号转换。

```
CYG _ LABEL _ NAME (name)
```

在 C 和 C++ 源文件中可以用这种定义方式，它可以使用在汇编程序或链接器脚本中所定义的标号。只有在 cyg/infra/cyg _ type. h 中的默认实现 (不加变化地传送参数 name) 不适当的情况下才需要这种定义。它与 CYG _ LABEL _ DEFN () 相对应。

CYG _ LABEL _ DEFN (name)

这是一个与 CYG _ LABEL _ NAME () 相对应的定义,在汇编源程序和链接器脚本中可以用它来定义标号。只有在 cyg/infra/cyg _ type. h 中的默认实现 (不加变化地传送参数 name) 不适当的情况下才需要这种定义。

(3) 基本类型。

```
cyg _ halint8
cyg _ halint16
cyg _ halint32
cyg _ halint64
cyg _ halcount8
cyg _ halcount16
cyg _ halcount32
cyg _ halcount64
cyg _ halbool
```

这些宏定义了 C 的基本类型,用于对给定大小的变量进行定义。只有在 cyg/infra/cyg _ type. h 中的默认定义不能使用的情况下才需要这种定义。这些定义只给出了基本类型,它们可以与 signed 和 unsigned 组合,形成更完整的类型说明。

(4) 原子类型。

```
cyg _ halatomic CYG _ ATOMIC
```

这种类型可以保证其读写操作不会被中断。它的大小与体系结构有关,但必须至少是 1B。

11.3.2 体系结构描述

硬件抽象层对 CPU 的基本体系结构进行了定义,这些定义包括:CPU 的上下文保存格式、上下文切换、位操作、断点、堆栈大小和地址转换等等。这些定义大多数位于头文件 cyg/hal/hal _ arch. h 内,体系结构抽象层将使用该头文件。如果有变体抽象层或平台抽象层的特殊定义,则其定义位于 cyg/hal/var _ arch. h 或 cyg/hal/plf _ arch. h 文件内。

(1) 寄存器保存格式。

```
typedef struct HAL _ SavedRegisters
{
    /* 与结构相关的寄存器列表 */
} HAL _ SavedRegisters
```

该结构对保存在堆栈内的 CPU 状态 (CPU 寄存器) 进行描述。在线程上下文切换、中断和例外处理等过程中将保存 CPU 状态。对这些过程所需保存状态的数量并不相同,一般来说,中断所需保存的状态是例外所保存状态的一个子集,而线程上下文状态又是中断状态的一个子集。出于调试目的,这三种情况的 CPU 状态都采用同一个结构。但在所需保存的 CPU 状态信息有显著区别时,这三种状态可以采用一种联合体的形式包含在该结构中。

(2) 线程上下文初始化。

HAL _ THREAD _ INIT _ CONTEXT(sp, arg, entry, id)

该宏定义对线程上下文环境进行初始化,以便 HAL _ THREAD _ SWITCH _ CONTEXT()可以实现上下文切换。其参数为:

sp—线程堆栈的当前指针所在位置。它是一个变量或结构成员。

arg—传送给入口函数的第一个参数的值。

entry—入口函数的地址,根据 C 调用约定对其进行调用,arg 的值将作为它的第一个参数。该函数原型为:

```
void entry (CYG _ ADDRWORD arg)
```

id—线程 id 值,仅用于调试目的。它被或入未使用的寄存器(与寄存器的初始值进行或操作),有助于从寄存器 dump 值中识别该线程。该值的最低 16 位应该为 0,为寄存器的标识符保留空间。

(3) 线程上下文切换。

```
HAL _ THREAD _ LOAD _ CONTEXT(to)
HAL _ THREAD _ SWITCH _ CONTEXT(from, to)
```

这两个宏用于实现线程的上下文切换。参数为:

from—当前线程堆栈指针存放位置的指针。

to—将要读取的下一个线程堆栈指针所在位置的指针。

使用 HAL _ THREAD _ LOAD _ CONTEXT()时,当前线程的 CPU 状态被丢弃,目的线程的 CPU 状态被读取。该宏只被使用一次,启动调度器加载系统的第一个线程的时候使用。

使用 HAL _ THREAD _ SWITCH _ CONTEXT()时,当前线程的 CPU 状态被保存到它的堆栈(使用当前堆栈指针值和参数 from 提供的状态存放地址),并从 to 指定的位置加载新线程的状态。

这两个宏的实现均采用了内嵌汇编程序的形式。HAL _ THREAD _ SWITCH _ CONTEXT()保存 CPU 的当前状态,包括当前中断状态。虽然对所有寄存器的值都进行保存有利于系统调试,但考虑到系统的性能因素,它只保存相关的一些寄存器。配置选项 CYGDBG _ HAL _ COMMON _ CONTEXT _ SAVE _ MNIUM 用于控制寄存器被保存的数量。

宏 HAL _ THREAD _ LOAD _ CONTEXT()用于加载一个新线程的上下文环境,丢弃当前的上下文环境。在实现时,它可以与 HAL _ THREAD _ SWITCH _ CONTEXT()共享一些程序代码。加载一个线程的上下文环境只需将被保存在堆栈内的寄存器值进行恢复,并使用跳转或返回指令回到被保存的程序计数器 PC。

值得注意的是,在这种上下文切换过程中并没有禁止中断。任何中断的发生都将被递送到当前 CPU 堆栈指针所指向的堆栈。因此,堆栈指针永远不能失效,也不能加载一个可能引起被保存的状态被中断所破坏的值。作为线程上下文环境的一部分,中断状态也同时被保存或恢复。如果某个线程在禁止了中断之后再进行上下文切换,在转换到另一个线程之后,这个新的线程有可能重新使能中断。一旦原来的线程重新获取了系统控制权后,中断将再一次被禁止。

(4) 位索引操作。

HAL _ LSBIT _ INDEX (index, mask)
HAL _ MSBIT _ INDEX (index, mask)

这两个宏分别计算 mask 中的最低有效位和最高有效位的位置,并将结果放置到 index。有些 CPU 提供了这样的指令进行位索引操作。如果没有这样的指令,这些宏必须调用 C 函数来完成这种工作。

⑸ 空闲 (idle) 线程行为。

HAL _ IDLE _ THREAD _ ACTION (count)

在某些环境下,内核空闲线程在其循环过程中可能需要 HAL 执行一些程序代码,例如可能要其执行处理器的 halt 指令。该宏定义为此提供了一种方法。它的参数 count 是空闲线程循环计数器的一个副本,可以用来触发一种间隔时间比每次循环时间更长的行为。

⑹ 排序栅栏 (reorder barrier)。

HAL _ REORDER _ BARRIER ()

在进行编译优化时,编译器可能会对某些程序代码重新排序。这种排序对多线程系统中的某些部分起着至关重要的作用,有时这种排序可能会导致一些问题的出现。在不需要进行这种重新排序的地方可以插入宏 HAL _ REORDER _ BARRIER (),用来防止编译器进行优化时的可能的程序代码迁移。该宏应该放置在必须按源程序代码顺序执行的两条语句之间。

⑺ 断点支持。

HAL _ BREAKPOINT (label)
HAL _ BREAKINST
HAL _ BREAKINST _ SIZE

这三个宏定义提供了对断点的支持。HAL _ BREAKPOINT () 执行一条断点指令,其参数 label 为断点指令处的标识符,例外处理程序据此可以检测到执行的是哪一个断点。HAL _ BREAKINST 包含的是断点指令代码的整型值。HAL _ BREAKINST _ SIZE 是断点指令的字节大小。这三个宏可以一起用来在程序的任何一处设置断点。

⑻ GDB 支持。

HAL _ THREAD _ GET _ SAVED _ REGISTERS (sp, regs)
HAL _ GET _ GDB _ REGISTERS (regval, regs)
HAL _ SET _ GDB _ REGISTERS (regs, regval)

这些宏为 GDB 到 HAL 的接口提供支持。

HAL _ THREAD _ GET _ SAVED _ REGISTERS () 从一个堆栈指针值 sp 提取 HAL _ Saved Registers 结构的指针。参数 sp 传来的堆栈指针应该是被线程上下文宏所保存的值。该宏将 HAL _ SavedRegisters 结构的指针赋给第二个参数变量 regs。

HAL _ GET _ GDB _ REGISTERS () 将 HAL 所保存的寄存器状态以 GDB 所希望的格式进行 dump 操作。参数 regs 为 HAL _ SavedRegisters 结构的指针,参数 regval 为 GDB 寄存器 dump 内容被保存的目的内存指针。

HAL _ SET _ GDB _ REGISTERS () 将 GDB 格式的寄存器 dump 内容转换为 HAL 所希望

的格式。参数 regval 为保存 GDB 寄存器 dump 内容的内存指针，参数 regs 为 HAL _ SavedRegisters 结构的指针。

(9) Setjmp 和 longjmp 支持。

```
CYGARC _ JMP _ BUF _ SIZE
hal _ jmp _ buf [CYGARC _ JMP _ BUF _ SIZE]
hal _ setjmp ( hal _ jmp _ buf env )
hal _ longjmp ( hal _ jmp _ buf env, int val )
```

这些函数为 C 函数 setjmp () 和 longjmp () 提供支持。

(10) 堆栈大小。

```
CYGNUM _ HAL _ STACK _ SIZE _ MINIMUM
CYGNUM _ HAL _ STACK _ SIZE _ TYPICAL
```

这两个宏分别定义了线程堆栈大小的最小值和标准值。

CYGNUM _ HAL _ STACK _ SIZE _ MINIMUM 对线程堆栈大小的最小值进行了定义。堆栈空间的最小值应该足够保证其线程功能的正确性，并且能够允许中断的发生，同时还应该允许进行上下文切换操作。它还应该具有足够的空间来执行简单的线程入口函数以及调用基本内核操作 (如互斥和信号)。除此之外，最小堆栈空间不提供更多的空间。应用程序在为它们自己的线程创建堆栈时，应该确定为实现其目的所需要的堆栈空间大小，并加到 CYGNUM _ HAL _ STACK _ SIZE _ MINIMUM 宏定义上。

CYGNUM _ HAL _ STACK _ SIZE _ TYPICAL 定义了堆栈的标准大小，它是在 CYGNUM _ HAL _ STACK _ SIZE _ MINIMUM 基础上的一个合理值，通常为 1kB。它可以满足大多数普通线程的需要。只有数据量非常大的线程以及程序调用层次非常深的线程才需要更大的堆栈空间。

(11) 地址转换。

```
CYGARC _ CACHED _ ADDRESS (addr)
CYGARC _ UNCACHED _ ADDRESS (addr)
CYGARC _ PHYSICAL _ ADDRESS (addr)
```

这些宏提供对内存地址的转换操作。在许多体系结构中，同一位置的内存空间根据视角的不同可能会有不同的地址值。通过 CPU 的内存管理单元或其他地址转换单元提供给程序的内存地址可能是与物理地址不同的虚地址。使用 HAL 所提供的这些宏可以进行内存地址的转换操作。

CYGARC _ CACHED _ ADDRESS () 将指定的地址转换到 cached memory 空间。这样的地址通常是应用程序访问内存的地址。

CYGARC _ UNCACHED _ ADDRESS () 将指定的地址转换到 uncached memory 空间。这样的地址通常是驱动程序为了避免出现 cache 问题而对内存空间进行访问的地址。另外，在该内存空间全有效之前对 cache 进行刷新时也需要使用这种地址转换。

CYGARC _ PHYSICAL _ ADDRESS () 将指定的地址转换到其物理地址空间 (转换为物理地址)。这些地址通常是需要传递给设备硬件 (如 DMA、网络设备、PCI 总线桥等) 的地址。程

序所使用的有可能不是物理地址，此时必须通过地址转换对其进行映射。

(12) 全局指针。

```
CYGARC_HAL_SAVE_GP()
CYGARC_HAL_RESTORE_GP()
```

这两个宏在被使用时，将在程序中分别插入一段保存或恢复全局数据指针的代码。在两个 eCos 事例之间进行上下文切换时需要使用它们。例如，在 eCos 应用程序和 RedBoot 之间进行切换时，需要使用这两个宏。

11.3.3 中断处理

HAL 的接口提供了一些有关中断处理的宏定义，包括例外和中断向量、中断的使能和屏蔽以及运行时的设置操作等。这些宏定义位于头文件 cyg/hal/hal_intr.h 内，属于体系结构抽象层。变体抽象层和平台抽象层的专用定义位于头文件 cyg/hal/var_intr.h、cyg/hal/plf_intr.h 或 cyg/hal/hal_platform_ints.h 中。

(1) 中断向量。

```
CYGNUM_HAL_VECTOR_XXXX
CYGNUM_HAL_VSR_MIN
CYGNUM_HAL_VSR_MAX
CYGNUM_HAL_VSR_COUNT

CYGNUM_HAL_INTERRUPT_XXXX
CYGNUM_HAL_ISR_MIN
CYGNUM_HAL_ISR_MAX
CYGNUM_HAL_ISR_COUNT

CYGNUM_HAL_EXCEPTION_XXXX
CYGNUM_HAL_EXCEPTION_MIN
CYGNUM_HAL_EXCEPTION_MAX
CYGNUM_HAL_EXCEPTION_COUNT
```

这里定义了系统中所有可能的 VSR (向量服务程序)、例外和中断向量，同时还提供了向量范围的最大值和最小值。有两个向量范围 VSR 和 ISR，它们之间没有固定的关系，不应将相同的 VSR 和 ISR 等同起来。VSR 与例外向量相对应，它们中的大多数是 CPU 的内部例外陷阱。与 ISR 相对应的是外部中断，它通过中断 VSR 对中断控制器进行译码而得出。

在 CPU 支持同步例外的情况下，这些同步例外向量的范围由 CYGNUM_HAL_EXCEPTION_MIN (最小值) 和 CYGNUM_HAL_EXCEPTION_MAX (最大值) 给出。CYGNUM_HAL_EXCEPTION_XXXX 是例外的标准名字 (XXXX 表示对应的例外)，与具体平台结构无关的一段通用程序 (适合于所有目标平台) 用其来检测具体目标系统是否存在某一特殊的例外。

CYGNUM_HAL_ISR_COUNT、CYGNUM_HAL_VSR_COUNT 和 CYGNUM_HAL_EXCEPTION_COUNT 分别定义了 ISR、VSR 和例外的数目。

⑵ 中断状态控制。

```
CYG_INTERRUPT_STATE
HAL_DISABLE_INTERRUPTS(old)
HAL_RESTORE_INTERRUPTS(old)
HAL_ENABLE_INTERRUPTS()
HAL_QUERY_INTERRUPTS(state)
```

这些宏对 CPU 中断屏蔽机制的状态进行控制,它们通常对 CPU 的某个状态寄存器进行操作,从而实现中断的使能和禁止。它们不直接对中断控制器进行操作。

CYG_INTERRUPT_STATE 是一个数据类型,用于保存 HAL_DISABLE_INTERRUPTS() 和 HAL_QUERY_INTERRUPTS() 两个宏返回的中断状态,或者将中断状态传递给 HAL_RESTORE_INTERRUPTS()。

HAL_DISABLE_INTERRUPTS() 禁止中断,并将原来的中断屏蔽状态保存到参数 old。

HAL_RESTORE_INTERRUPTS() 将中断屏蔽状态恢复为原来保存在 old 内的值。

HAL_ENABLE_INTERRUPTS() 使能中断,它不考虑当前的中断屏蔽状态。

HAL_QUERY_INTERRUPTS() 将中断屏蔽状态保存到参数 state。保存于该参数的中断屏蔽状态可以在稍后的某个时刻作为 HAL_RESTORE_INTERRUPTS() 的参数来恢复中断屏蔽状态。

⑶ ISR 和 VSR 的管理。

```
HAL_INTERRUPT_IN_USE(vector, state)
HAL_INTERRUPT_ATTACH(vector, isr, data, object)
HAL_INTERRUPT_DETACH(vector, isr)
HAL_VSR_SET(vector, vsr, poldvsr)
HAL_VSR_GET(vector, pvsr)
HAL_VSR_SET_TO_ECOS_HANDLER(vector, poldvsr)
```

这些宏将中断和例外分别与它们相应的服务程序 ISR 和 VSR 进行连接分配。

HAL_INTERRUPT_IN_USE() 对参数 vector 指定的中断向量的状态进行检测,检查是否已经有中断服务程序 ISR 与其挂接。如果已经分配了 ISR,则将参数 state 设为 1,否则设为 0。对于每一个中断向量,HAL 只允许为其分配一个 ISR。因此,在使用 HAL_INTERRUPT_ATTACH() 之前,最好先使用 HAL_INTERRUPT_IN_USE() 函数来判断是否已经分配了 ISR。

HAL_INTERRUPT_ATTACH() 给参数 vector 指定的中断向量分配 ISR (isr)、数据指针 (data) 和对象指针 (object)。当发生与该中断向量对应的中断时,将调用由此函数分配的中断服务程序 ISR,并将向量号和数据指针分别作为 ISR 的第一个参数和第二个参数。

HAL_INTERRUPT_DETACH() 将中断服务程序 ISR 与中断向量 vector 进行分离。

HAL_VSR_SET() 将参数 vsr 指定的 VSR 替换原来分配给例外向量 vector 的 VSR。原来的 VSR 由参数 pvsr 带回。

HAL_VSR_GET() 读取例外向量 vector 相应的 VSR,由参数 pvsr 带回。

HAL_VSR_SET_TO_ECOS_HANDLER() 用于确保指定例外 (vector) 的 VSR 是

eCos 例外VSR，而不是 RedBoot 或其他 ROM monitor 程序的例外 VSR。当处于 RedBoot 阶段时，例外由 RedBoot 处理并传送给 GDB。该宏定义将例外转向到eCos，从而使应用程序可以对它进行处理。参数 vector 指定需要进行 VSR 转向的例外向量，poldvsr 用于保存原来的 VSR 指针，以后还可以用它来恢复原来的 VSR。

(4) 中断控制器的管理。

```
HAL_INTERRUPT_MASK(vector)
HAL_INTERRUPT_UNMASK(vector)
HAL_INTERRUPT_ACKNOWLEDGE(vector)
HAL_INTERRUPT_CONFIGURE(vector, level, up)
HAL_INTERRUPT_SET_LEVEL(vector, level)
```

这些宏用于对具有优先级控制能力的中断控制器进行控制。如果没有优先级控制器，它们应该为空。这些宏不具备可重入能力，在中断处于使能状态下对它们的使用应该特别小心。在中断服务程序 ISR 和中断使能之前的初始化程序中使用它们是安全的。但是在 DSR 和线程中使用这些宏时必须先禁止中断。下面是 DSR 中的一个使用例子：

```
...
HAL_DISABLE_INTERRUPTS(old);
HAL_INTERRUPT_UNMASK(CYGNUM_HAL_INTERRUPT_ETH);
HAL_RESTORE_INTERRUPTS(old);
...
```

HAL_INTERRUPT_MASK()屏蔽参数 vector 指定的中断。

HAL_INTERRUPT_UNMASK()使能参数 vector 指定的中断。

HAL_INTERRUPT_ACKNOWLEDGE()对参数 vector 对应的中断进行中断应答。通常用在中断服务程序 ISR 内，在其允许响应中断时可以使用该宏。许多中断控制器在允许响应下一个中断之前要求进行中断应答。

HAL_INTERRUPT_CONFIGURE()对中断的触发方式进行配置。参数定义如下：

vector—指定被配置的中断。

level—指定触发方式：true 为电平触发，false 为边沿触发。

up—如果中断被设置为电平触发方式，则 up 为 true 时将是高电平触发，up 为 false 时为低电平触发。

HAL_INTERRUPT_SET_LEVEL()设置中断优先级。参数定义如下：

vector—指定要设置的中断。

level—中断优先级。一些系统可以通过改变中断优先级来实现中断的屏蔽和使能操作。

(5) 时钟控制。

```
HAL_CLOCK_INITIALIZE(period)
HAL_CLOCK_RESET(vector, period)
HAL_CLOCK_READ(pvalue)
```

系统内核在提供超时、延时和调度服务时需要用到时钟和定时设备，上面这些宏可以用于对这些设备进行控制。时钟的实现可以认为是一种具有某种形式的计数器通过一个外部信号

源对其进行加或减操作，当该计数器达到一个预定值时就产生一个中断。

HAL_CLOCK_INITIALIZE()对定时设备进行初始化，使其周期性地产生中断。

HAL_CLOCK_RESET()对定时器重新进行初始化。只有在每次中断后需要对定时设备进行复位时才使用该宏。

HAL_CLOCK_READ()读取定时计数器的当前值。读取的值存放在 pvalue 指定的位置。

(6) 微秒级延时。

```
HAL_DELAY_US(us)
```

该宏的实现是可选的，它可实现微秒级的延时。该宏通常在需要很短的延时情况下使用，如硬件控制、Flash 设备编程等等。由于它有可能禁止中断，而且它是采用一种忙循环的方式实现的，因此在对中断延时或上下文切换延时敏感的地方不要使用它。

11.3.4 I/O 操作

对设备进行访问和控制时需要访问 I/O 寄存器。硬件抽象层提供了一些宏用于 I/O 寄存器的读写操作。这些宏一般位于头文件 cyg/hal/hal_io.h 内。如果变体抽象层和平台抽象层有专用宏对 I/O 进行访问，则这些宏位于 cyg/hal/var_io.h 和 cyg/hal/plf_io.h 内。

(1) 寄存器地址。

```
HAL_IO_REGISTER
```

这是一个存放 I/O 寄存器地址的类型定义，通常是一个内存地址、一个端口地址或者一个 I/O 空间的偏移地址。在较复杂的体系结构中，它可能是一个地址空间的基地址加一个地址偏移所组成的一个字，或者是用一个结构体的形式表示寄存器地址。

这种类型的变量和常数的值通常在配置机制或专用的头文件中指定。

(2) 寄存器读操作。

```
HAL_READ_XXX(register, value)
HAL_READ_XXX_VECTOR(register, buffer, count, stride)
```

这些宏用于读取各种长度的 I/O 寄存器。其中的 XXX 可以是 UINT8、UINT16、UINT32。

HAL_READ_XXX()从寄存器 register 读取适当长度的值并保存到参数 value。

HAL_READ_XXX_VECTOR()读取 count 个适当长度的寄存器值并存放到 buffer 指定的空间。参数 stride 控制寄存器空间的指针如何移动，为 0 时表示重复读同一个寄存器，为 1 时读连续相邻的寄存器(指针每次加 1)，为 2 时隔一个读一个(指针每次加 2)，依次类推。

(3) 寄存器写操作。

```
HAL_WRITE_XXX(register, value)
HAL_WRITE_XXX_VECTOR(register, buffer,count, stride)
```

这些宏用于对寄存器进行各种长度的写操作。其中的 XXX 可以是 UINT8、UINT16、UINT32。

HAL _ WRITE _ XXX () 以适当的长度将参数 value 的值写到寄存器 register。

HAL _ WRITE _ XXX _ VECTOR () 以适当的长度将 buffer 指定空间内的值写入 count 个寄存器。参数 stride 控制寄存器空间的指针如何移动，为 0 时表示重复写同一个寄存器，为 1 时写连续相邻的寄存器（指针每次加 1），为 2 时隔一个写一个（指针每次加 2），依次类推。

11.3.5 Cache 控制

硬件抽象层提供对 Cache 控制的支持。Cache 控制的宏定义通常位于头文件 cyg/hal/hal _ cache.h 内。不同系统可能在不同的层次实现对 Cache 的控制，因此该头文件可能相应地位于体系结构抽象层、变体抽象层或者平台抽象层内。一般来说，体系结构抽象层内有一些普通的 Cache 控制宏定义，但变体抽象层和普通抽象层可以覆盖它们，或者解除这些定义。变体抽象层和普通抽象层对 Cache 控制的宏定义分别位于头文件 cyg/hal/var _ cache.h 和 cyg/hal/plf _ cache.h 内。

有些 Cache 宏定义要分别实现对数据 Cache 和指令 Cache 的控制，在宏的名字中使用 DCACHE 和 ICACHE 来对它们加以区别。在某些体系结构中，只使用一个统一的 Cache，数据 Cache 和指令 Cache 使用同一个 Cache。这种情况下，宏的名字使用 UCACHE，具有 DCACHE 和 ICACHE 名字的宏都将调用 UCACHE 宏。在后面的介绍中，统一用 XCACHE 来表示它们中的任何一个。

一些目标平台对某些 Cache 宏的使用具有一定的限制，使用时应该加以注意。在使用具有破坏性的 Cache 宏时要特别小心。在 Cache 失效之前进行 Cache 同步操作是不安全的，这是因为在同步操作之后到 Cache 失效前的这一时间段内可能有中断产生，有可能引起中断期间所产生的脏数据行的状态丢失。解决这一问题的一种方法是在进行同步和失效操作时临时关闭 Cache，这样保证了在中断期间 Cache 中不会有新的数据。如果目标系统不具备这种能力，则只能在对 Cache 进行操作时禁止中断，但这样做显然会耗费较长的一段时间。

对于某些目标平台，HAL 提供了一对用于查询其 Cache 状态的宏，它们分别是 HAL _ ICACHE _ IS _ ENABLED (x) 和 HAL _ DCACHE _ IS _ ENABLED (x)。如果指令 Cache 或数据 Cache 处于使能状态，则它们分别将其参数 x 置为 1。像其他许多 Cache 控制宏定义一样，这两个宏也是可选的。对于具有不同支持能力的目标系统平台来说，这种可选方式为其提供了很大的选择余地。

(1) Cache 粒度。

```
HAL _ XCACHE _ SIZE
HAL _ XCACHE _ LINE
_ SIZE
HAL _ XCACHE _ WAYS
HAL _ XCACHE _ SETS
```

这些宏对指令 Cache 和数据 Cache 的大小和粒度进行了定义。

HAL _ XCACHE _ SIZE 定义整个 Cache 的字节大小。

HAL _ XCACHE _ LINE _ SIZE 定义 Cache 行的字节大小。

HAL _ XCACHE _ WAYS 对每组 Cache 的组相联数目进行定义。如果是直接映射 Cache，则为 1 如果是两路组相联 Cache，则为 2 如果是 4 路组相联 Cache，则为 4。依此类推。

HAL_XCACHE_SETS 定义 Cache 组数目，它是根据上面定义的 Cache 大小和 Cache 组相联数目进行计算的结果。其计算方法为：

$$\text{HAL\_XCACHE\_SETS} = \frac{\text{HAL\_XCACHE\_SIZE}}{\text{HAL\_XCACHE\_LINE\_SIZE} * \text{HAL\_XCACHE\_WAYS}}$$

⑵ 全局 Cache 控制。

```
HAL_XCACHE_ENABLE()
HAL_XCACHE_DISABLE()
HAL_XCACHE_INVALIDATE_ALL()
HAL_XCACHE_SYNC()
HAL_XCACHE_BURST_SIZE(size)
HAL_DCACHE_WRITE_MODE(mode)
HAL_XCACHE_LOCK(base, size)
HAL_XCACHE_UNLOCK(base, size)
HAL_XCACHE_UNLOCK_ALL()
```

这些宏用于改变 Cache 状态。

HAL_XCACHE_ENABLE() 使能 Cache，HAL_XCACHE_DISABLE() 关闭 Cache。

HAL_XCACHE_INVALIDATE_ALL() 对整个 Cache 进行失效操作。有些硬件可能要求在进行失效操作时关闭 Cache，在这种情况下，应该使用 HAL_XCACHE_IS_ENABLED() 来保存和恢复前面的 Cache 状态。在 HAL_XCACHE_SYNC() 之后如果使用该宏清 Cache（把脏数据写回内存后失效 Cache），则必须在这两个宏之间禁止中断。下面是其使用例子：

```
...
HAL_DISABLE_INTERRUPTS(old);
HAL_XCACHE_SYNC();
HAL_XCACHE_INVALIDATE_ALL();
HAL_RESTORE_INTERRUPTS(old);
...
```

由于这种操作占用时间较长，对系统的实时性具有一定的影响，因此只有在必需的情况下才进行这种操作，而且这种延时不得对驱动程序和应用程序的操作有影响。

HAL_XCACHE_SYNC() 使 Cache 内容与内存同步。在某些系统中，它等同于 HAL_XCACHE_INVALIDATE_ALL()。

在某些系统中，Cache 和内存之间可能采用 burst 方式进行数据传输，HAL_XCACHE_BURST_SIZE() 用于对这种 burst 方式的数据大小进行控制。该宏只有在 Cache 具有 burst 功能时才实现。

HAL_DCACHE_WRITE_MODE() 对数据 Cache 行写回内存的方式进行控制。典型的方式有 HAL_DCACHE_WRITEBACK_MODE（写回）和 HAL_DCACHE_WRITETHRU_MODE（写穿透）等。只有在具有这种功能的条件下才实现该宏。

HAL_XCACHE_LOCK() 将数据锁定在 Cache 内。参数 base 和 size 指定数据被锁的内存区。一次能否锁定多个内存区以及这种锁定是否会终止其他内存区域的 Cache 操作都与具

体的体系结构相关。只有在具有这种功能的条件下才实现该宏。

HAL _ XCACHE _ UNLOCK ()取消对指定内存区域的锁定。只有在具有这种功能的条件下才实现该宏。

HAL _ XCACHE _ UNLOCK _ ALL ()取消所有对内存区域的锁定。在某些体系结构中,有可能把它当成 Cache 初始化时需要完成的一种操作。只有在具有这种功能的条件下才实现该宏。

(3) Cache 行控制。

```
HAL _ DCACHE _ ALLOCATE(base , size)
HAL _ DCACHE _ FLUSH(base , size)
HAL _ XCACHE _ INVALIDATE(base , size)
HAL _ DCACHE _ STORE(base , size)
HAL _ DCACHE _ READ _ HINT(base , size)
HAL _ DCACHE _ WRITE _ HINT(base , size)
HAL _ DCACHE _ ZERO(base , size)
```

这些宏对所有与参数 base 和 size 指定的内存区相匹配的 Cache 行进行某种 Cache 操作。只有在具有这种功能的条件下才实现该宏。这些宏并不一定只对这些指定的区域进行操作,在某些系统中可能是对整个 Cache 进行操作。

HAL _ DCACHE _ ALLOCATE ()将 Cache 行分配给指定的内存区,但不从内存读数据,因此 Cache 行的内容是不确定的。可用于内存块的复制。

HAL _ DCACHE _ FLUSH ()在脏数据行写回内存后失效指定范围的所有 Cache 行。

HAL _ XCACHE _ INVALIDATE ()失效所有指定范围内的 Cache 行。脏数据被失效但不写回内存。

HAL _ DCACHE _ STORE ()将指定区域内的所有脏数据行写入内存,但不失效 Cache 行。

HAL _ DCACHE _ READ _ HINT ()提示指定区域的 Cache 即将被读。它有可能引起指定区域的内存进入 Cache。

HAL _ DCACHE _ WRITE _ HINT ()提示指定区域的 Cache 即将被写。

HAL _ DCACHE _ ZERO ()对指定区域分配 Cache 行并清 0,但不读内存。可用于大块内存的清操作。

11.3.6 SMP 支持

eCos 对 SMP 提供有限的支持,这些支持只适用于某些体系结构和平台。它对 SMP 目标系统硬件有下面的限制条件:

① 适度多处理:系统支持的 CPU 个数通常为两个到四个,最多不超过八个。

② SMP 同步支持:硬件必须提供一种机制允许运行在两个 CPU 上的软件进行同步。例如,可以提供这样一些指令:test-and-set、compare-and-swap、load _ link/store-conditional 等。另一种方法是提供硬件信号寄存器来实现这些操作的串行化。这些硬件功能用于实现 eCos 的 spinlock。

③ Cache 一致性:由硬件实现 Cache 一致性,软件在访问共享内存时不需做更多的 Cache

一致性工作。

④ 统一编址：所有 CPU 对共享内存的访问使用相同的地址。

⑤ 统一设备编址：所有 CPU 都可以平等访问所有的设备。

⑥ 中断路由：硬件平台必须提供一个能将中断路由到指定 CPU 的中断控制器。可以将所有中断路由到一个 CPU，或者将某些中断路由到指定的一些 CPU。eCos 目前不支持动态路由，也不支持同时将所有中断路由到所有 CPU。

⑦ 机间中断：必须提供一个机间中断机制，允许一个 CPU 中断另一个 CPU。CPU 的 ID 号通常由 CPU 状态寄存器提供，或者由机间中断机制提供。CPU 的 ID 应该是简单的一个小的正整数。

硬件抽象层提供了一些操作用于支持 SMP 系统。这些对 SMP 的支持位于头文件 cyg/hal/hal_smp.h 内。变体抽象层和平台抽象层的一些专用定义分别位于头文件 cyg/hal/var_smp.h 和 cyg/hal/plf_smp.h 内。硬件抽象层对 SMP 的支持包括下述几个方面：

(1) CPU 控制。

硬件抽象层提供了一些对 SMP 系统中的 CPU 进行描述和管理的宏定义。它们分别是：

HAL_SMP_CPU_TYPE

CPU ID 的类型定义。CPU ID 是一个小的正整数，常用于对某些基于每个 CPU 的数组变量进行管理。

HAL_SMP_CPU_MAX

系统支持的最大 CPU 数目。

HAL_SMP_CPU_COUNT()

返回当前正在运行的 CPU 个数。

HAL_SMP_CPU_THIS()

返回当前 CPU 的 ID 号。

HAL_SMP_CPU_NONE

一个不是实际 CPU ID 的值。用在 CPU 类型变量必须是空值的地方。

HAL_SMP_CPU_START(cpu)

启动指定 CPU 在 HAL 的入口点处开始执行。在完成硬件抽象层内的初始化操作后，CPU 进入内核启动点 cyg_kernel_cpu_startup()。

HAL_SMP_CPU_RESCHEDULE_INTERRUPT(cpu, wait)

给 CPU 发送一个重新调度(reschedule)的中断。如果 wait 为非 0 值，则等待应答。响应该中断的 CPU 将在其滞后服务程序 DSR 中调用函数 cyg_scheduler_set_need_reschedule()来启动重新调度。

HAL_SMP_CPU_TIMESLICE_INTERRUPT(cpu, wait)

给 CPU 发送时间片(timeslice)中断。如果 wait 为非 0 值，则等待应答。响应该中断的

CPU 将调用 cyg _ scheduler _ timeslice _ cpu () 函数处理时间片事件。

(2) test-and-set 支持。

test-and-set 是 SMP 同步机制的基础,它具有下述宏定义:

HAL _ TAS _ TYPE

所有 test-and-set 变量的类型定义。test-and-set 宏只支持一位操作,通常是 HAL _ TAS _ TYPE 的最低有效位。

HAL _ TAS _ SET (tas, oldb)

在 tas 位置形成一个测试和设置操作。如果该位置已经被设置,则由 oldb 返回 true,否则返回 false。

HAL _ TAS _ CLEAR (tas, oldb)

在 tas 位置形成测试和清操作。如果该位置已经被设置,则由 oldb 返回 true,如果它被清则由 oldb 返回 false。

(3) Spinlock。

Spinlock 为 CPU 之间提供一种锁机制。它通常实现在 test-and-set 机制基础之上,但有时也可以通过其他方法实现,比如由硬件直接提供 Spinlock 支持。HAL 提供了下述用于 Spinlock 的数据类型、值和函数的宏定义:

HAL _ SPINLOCK _ TYPE

Splinlock 变量类型。

HAL _ SPINLOCK _ INIT _ CLEAR

初始化 Spinlock 清操作的值。

HAL _ SPINLOCK _ INIT _ SET

初始化 Spinlock 所设置的值。

HAL _ SPINLOCK _ SPIN (lock)

调用者处于忙循环状态等待清 lock。当 lock 被清后,它对 lock 进行设置并跳出循环继续执行。这种处理是自动进行的,CPU 之间不会出现空转情况。

HAL _ SPINLOCK _ CLEAR (lock)

调用者清 lock。等待该 lock 的所有 CPU 中的一个 CPU 将跳出等待状态。

HAL _ SPINLOCK _ TRY (lock, val)

尝试对 lock 进行设置。如果对 lock 的声称成功,则将 val 设置为 true,否则为 false。

HAL _ SPINLOCK _ TEST (lock, val)

对 lock 的当前值进行测试。如果 lock 可以被声称,则将 val 设置为 true,如果 lock 处于清状态则 val 设置为 false。

(4) 调度锁 (Scheduler Lock)。

调度锁是所有内核数据结构的一个主要保护措施。在默认情况下,内核自己使用一个 Spinlock 来实现调度锁。如果硬件不支持 Spinlock 或者有其他更为有效的实现方法,则可以由 HAL 提供一些宏来实现调度锁。这些宏包括:

```
HAL_SMP_SCHEDLOCK_DATA_TYPE
```

数据类型 (可能是一个结构体) 宏定义,它包含实现调度锁所需要的任何数据。Cyg_Scheduler_SchedLock 类的静态成员是这种类型的一个变量例子。下面介绍的每个宏都要用到它。

```
HAL_SMP_SCHEDLOCK_INIT(lock, data)
```

初始化调度锁。参数 lock 是调度锁计数器,参数 data 是一个 HAL_SMP_SCHEDLOCK_DATA_TYPE 类型的变量。

```
HAL_SMP_SCHEDLOCK_INC(lock, data)
```

调度锁 lock 加 1。引起调度锁 lock 从 0 加到 1 的 CPU 将处于等待状态直到另一个 CPU 将其清 0。调度锁 lock 从 1 以后的加 1 操作的时间开销较小,因为 CPU 已经拥有了该锁。

```
HAL_SMP_SCHEDLOCK_ZERO(lock, data)
```

调度锁 lock 清 0。该操作将清调度锁 lock,使其他 CPU 可以对 lock 进行声称。

```
HAL_SMP_SCHEDLOCK_SET(lock, data, new)
```

将调度锁 lock 设置一个不同的值 new。只有在已经知道当前 CPU 拥有该 lock 的时候才可以调用此宏。不能使用此宏对 lock 清 0 或将 lock 从 0 加到 1。

(5) 中断路由。

SMP 系统需要将中断分配给指定的 CPU 进行处理,硬件抽象层为此提供了一些中断路由的宏定义。hal_intr.h 提供两个接口支持用于将中断路由到不同的 CPU。一旦中断被路由到一个新的 CPU,中断屏蔽和配置操作应考虑到 CPU 的中断路由。例如,如果这些屏蔽和配置操作没有被目的 CPU 所调用,就有可能要求 HAL 将这些操作转换到目的 CPU。这两个宏定义分别是:

```
HAL_INTERRUPT_SET_CPU(vector, cpu)
```

将 vector 指定的中断路由到参数 cpu 指定的 CPU。

```
HAL_INTERRUPT_GET_CPU(vector, cpu)
```

获取中断 vector 路由目的 CPU 的 ID 号,ID 号由参数 cpu 带回。

11.3.7 诊断支持

硬件抽象层提供对低级诊断 IO 操作的支持。这些诊断操作在开发新目标系统的早期过程中非常有效。在早期开发过程中,通常使用一个 UART 和其他串行 IO 设备作为调试端口,也可以使用带输出通道的模拟器、ROM 模拟器、LCD 等等输出设备。硬件抽象层对这些调试

用 IO 设备提供了相应的函数支持。这些函数包括：

HAL _ DIAG _ INIT ()

对输出诊断信息的设备进行初始化。对于 UART，这种初始化包括对波特率、停止位、奇偶位等参数的设置。对其他的设备可能要对相应的控制器进行初始化，或者建立与远程设备的连接。

HAL _ DIAG _ WRITE _ CHAR (c)

对诊断输出设备写一个字符。

HAL _ DIAG _ READ _ CHAR (c)

从诊断设备读取一个字符。该函数不一定支持所有的诊断设备。

头文件 cyg/hal/hal _ diag. h 对这些宏进行了定义，一般位于变体抽象层和平台抽象层内。

11.3.8 链接脚本

eCos 应用程序在进行编译链接时要受到链接脚本的控制。链接脚本定义了程序代码和数据在内存区域中的内存地址和大小，并对编译器所产生的各种区段进行分配。

对应用程序进行链接时，实际使用的链接脚本是位于安装目录 install 的 lib/target. ld 文件。它的产生来源于两个文件：一个基本链接脚本文件和一个由内存布局工具产生的. ldi 文件。基本链接脚本文件通常由体系结构抽象层或变体抽象层提供，它由一组以 C 预处理宏形式表示的链接脚本段组成，这些脚本段对链接操作所产生的主要输出段 (section) 进行定义。基本链接脚本文件中包含了一个. ldi 文件，该文件使用基本链接脚本文件中的宏定义给这些输出段分配所需的内存区和链接地址。

. ldi 文件由平台抽象层提供，它包含了目标平台的内存布局信息。这些文件的名称一般都符合标准命名约定，具有如下格式：

pkgconf/mlt _ <architecture> _ <variant> _ <platform> _ <startup> . ldi

其中<architecture>、<variant>和<platform>是相应的 HAL 名字。<startup>是启动类型，通常是 ROM、RAM 或 ROMRAM。

除了. ldi 文件外，还有一些. h 文件可以被应用程序用来访问. ldi 文件中的信息。它包含了内存布局信息和用户自定义的一些信息，如堆空间、PCI 总线内存访问窗口等。

. ldi 文件由内存布局工具 MLT 生成。MLT 将内存配置信息保存到平台抽象层中的一个具有下述名字格式的文件内：

include/pkgconf/mlt _ <architecture> _ <variant> _ <platform> _ <startup> . mlt

MTL 工具使用该文件生成. ldi 文件和. h 文件。如果直接修改了这两个文件，再次运行 MTL 时重新生成的文件将覆盖所作的修改。pkgconf/system. h 中的宏定义对. ldi 和. h 文件的名字进行了定义，分别为 CYGHWR _ MEMORY _ LAYOUT _ LDI 和 CYGHWR _ MEMORY _ LAYOUT _ H。

11.4 例外处理

前面介绍的硬件抽象层的大部分操作都是对硬件进行访问然后返回的操作。这一节将介绍硬件抽象层对例外的处理,包括同步硬件陷阱和异步设备中断。对例外的处理首先是由HAL进行相应的处理,然后再将控制权交给eCos或其应用。在eCos完成对它的处理后,控制权又交还给HAL,并从例外发生处恢复正常的程序处理。

硬件抽象层的例外处理程序通常位于体系结构抽象层中的文件vector.s文件内。vector.s通常包含了reset入口点,它对系统的启动进行处理。例外处理程序一般要完成的功能如下:

① 系统启动和初始化。

② 硬件例外的交付处理。

③ 同步例外的默认处理。

④ 异步中断的默认处理。

11.4.1 HAL的启动处理

系统启动时通常从复位向量处开始执行程序。硬件抽象层从复位向量开始就必须保证系统能够正常运行,它负责建立程序执行环境,并最后调用应用程序的入口点函数。下面是系统启动时复位向量处理程序所需要完成的工作。在某些系统配置中,有些工作是不需要的。

(1) 对硬件进行初始化。这种初始化涉及到体系结构抽象层、变体抽象层以及平台抽象层,可能要对多个子系统进行初始化。它们包括:

① CPU寄存器初始化,最为重要的是要对CPU的中断屏蔽进行设置,禁止中断。

② 如果使用了存储管理部件MMU,则对其进行初始化。在许多系统中,只能通过MMU对某些地址范围的Cache能力进行控制。另外,在访问RAM和设备寄存器时需要通过MMU进行地址映射。为简单起见,这种映射应该尽可能地接近一对一的虚实转换方式。

③ 对存储控制器进行初始化设置,使RAM、ROM和I/O设备能够被访问。在此之前,RAM是不能访问的。如果是ROMRAM启动,此时可以将程序代码从ROM搬移到RAM,并转到RAM继续执行程序的其他部分。

④ 对总线桥及芯片进行初始化设置。在需要通过它们输出早期诊断信息的平台中这一步工作特别重要。

⑤ 对诊断机制进行初始化。

⑥ 初始化浮点部件和其他的扩展部件,如SIMD(单指令多数据流)和多媒体部件。

⑦ 初始化中断控制器。初始化工作至少应该对其进行配置,屏蔽所有的中断。针对具体硬件的实际需要,有可能要建立从中断控制器的向量表空间到CPU的例外表空间的映射关系。在主、从中断控制器之间也需要建立类似的映射关系。

⑧ Cache的禁止与初始化。在系统启动的最初阶段通常不要使能Cache,但可能需要清Cache并对它们进行初始化,以便它们在后面的操作中可以被使能。

⑨ 初始化定时器、时钟。

上述这些初始化操作的执行顺序与具体平台结构有关,某些系统可能不需要其中的某些部分。初始化工作的主要目的是使系统能够正常运行,并能够进行C函数的调用。难以用汇

编语言编写的更为复杂的初始化可以延缓到 hal _ variant _ init () 函数或 hal _ platform _ init () 函数执行完成后再进行。

⑵ 建立堆栈指针，使后续的初始化程序可以进行适当的过程调用。中断堆栈通常用于此目的。

⑶ 对访问全局变量所需要的全局指针 GP 进行初始化，使后续初始化程序能够访问全局变量。

⑷ 如果系统从 ROM 启动，则将 ROM 中的程序代码复制到 RAM。

⑸ 对 .bss (未初始化数据段) 清 0。

⑹ 产生一个适合于 C 函数调用的栈结构。

⑺ 调用 hal _ variant _ init () 函数和 hal _ platform _ init () 函数，对平台的其他部分进行初始化。通常包括对中断控制器、PCI 总线桥、基本 IO 设备进行更详细的初始化，并对 Cache 进行使能。

⑻ 调用 cyg _ hal _ invoke _ constructors () 函数运行静态构造程序。

⑼ 调用 cyg _ start () 函数，启动应用程序。如果 cyg _ start () 函数返回，则进入死循环。

11.4.2 同步例外与异步中断的处理

无论是同步例外还是异步中断，CPU 将所有的例外都交付给一组向量服务程序 VSR 处理。所有的例外都分别对应于一个例外向量，每一个例外向量都有一个服务程序对其进行处理。由于体系结构的多样性，不可能采用一个通用的机制来对例外向量进行直接处理。eCos 使用一种可移植的程序代码来对例外向量进行处理。对于不同体系结构的平台系统，只需对与具体硬件相关的部分进行移植就可以使用 eCos 提供的例外向量处理机制。

eCos 所采取的例外向量处理机制是在每一个硬件向量上都加有一小段跳转程序，该跳转程序使用了一个表，通过该表间接跳转到相应的例外处理程序。这种例外处理程序称为向量服务程序 VSR，该表称为 VSR 表。跳转程序只进行最简单的例外识别处理，当知道所发生的是哪一个例外时，就跳转到相应的 VSR。VSR 负责保存 CPU 的状态，并采取相应的措施对例外和中断进行处理。图 11-2 是 eCos 的例外处理机制示意图。

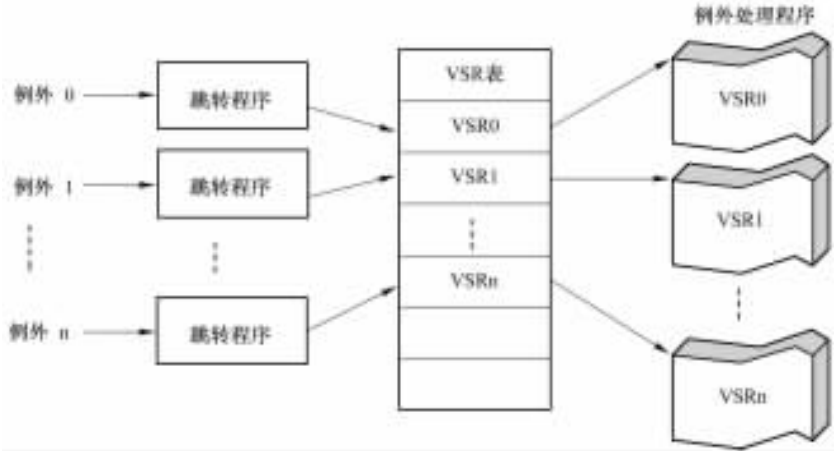


图 11-2 eCos 的例外处理机制

在默认情况下,大多数同步例外的 VSR 表项都指向同一个默认的例外 VSR,该 VSR 以一种通用的方式对所有的例外进行处理。它简单地保存 CPU 的状态,对 CPU 状态进行一定的改变后再调用 cyg _ hal _ exception _ handle () 函数进行进一步的处理。cyg _ hal _ exception _ handle () 函数然后再将例外交给其他一些处理程序,它的最终目的有两个:进入 GDB 或者将例外交给 eCos 应用程序。具体进入到哪一个目的程序与具体配置有关,当 eCos 包含有 GDB stub 程序时,例外将交给 GDB,否则的话,将交给 eCos 应用程序进行处理。

如果 eCos 应用程序是通过 RedBoot 加载的,VSR 表项将指向 RedBoot 的例外 VSR,此时发生的例外将进入 GDB。如果 eCos 应用程序要求自己对某个例外进行处理,则它必须将相应的 VSR 表项进行替换,使其指向它自己提供的 VSR 服务程序。可以使用宏 HAL _ VSR _ SET _ TO _ ECOS _ HANDLER () 来实现这种替换操作。

在默认情况下,大多数的异步中断向量都将指向同一个默认的中断 VSR 进行处理。中断 VSR 对从中断控制器来的中断进行译码,并调用相应的中断服务程序 ISR 进行处理。如果要求与内核进行交互以及允许中断引起线程的抢先,则这种默认的中断 VSR 必须实现相当多的功能。为支持这种 VSR,需要使用 ISR 向量表。每一个有效的中断向量都具有三个指针与其相对应,这三个指针分别是 ISR 指针、数据指针和 HAL 不可见的中断对象指针(内核所需)。这三个指针可以合用一个表,也可以用三个独立的表来存放,具体采用哪种实现方式可根据具体情况来决定。

中断 VSR 按照顺序所需要完成的操作如下:

① 保存 CPU 状态。在 no-debug 配置中,可以不保存 CPU 的全部状态。在某些目标平台中,可以使用配置选项 CYGDBG _ HAL _ COMMON _ INTERRUPTS _ SAVE _ MINIMUM _ CONTEXT 来实现这种最小状态的保存。

② 使内核调度锁加 1。调度锁是 Cyg _ Scheduler 类型的一个成员,为了使汇编程序可以对其进行访问,它使用别名 cyg _ scheduler _ sched _ lock。

③ 中断堆栈的处理。这是一个可选的操作,由配置选项 CYGIMP _ HAL _ COMMON _ INTERRUPTS _ USE _ INTERRUPT _ STACK 来决定是否进行这种处理。

④ 对中断控制器递交的外部中断进行译码,产生 ISR 向量号。

⑤ 重新使能中断,允许中断嵌套,允许更高优先级的中断产生。

⑥ 根据 ISR 向量号从 ISR 向量表中找回 ISR 指针和数据指针。

⑦ 建立 C 调用栈结构。包括栈空间、参数、返回指针等。

⑧ 调用 ISR,将向量号和数据指针传递给 ISR。在调用过程中将保存向量号和状态保存区的指针。

⑨ 如果是非嵌套的中断并且使用一个单独的中断堆栈,则切换到被中断的线程的堆栈。

⑩ 用保存的 ISR 向量号从 ISR 向量表中获得中断对象指针。

⑪ 调用 interrupt _ end (),将 ISR 的返回值、中断对象指针和 CPU 状态保存区指针传递给它。该函数由内核实现,负责中断处理的结束工作。根据 ISR 的返回值,它可能要求运行 DSR,并将调度锁减 1。如果调度锁被其归零,则将调用它所要求运行的 DSR,这时又有可能引起线程上下文切换的产生。

⑫ interrupt _ end () 函数 可能在其被调用之后的一段时间后返回。在此期间可能执行其他的线程。根据具体体系结构的要求,可能需要再一次禁止中断。

上述操作的详细步骤根据体系结构的不同而可能有轻微的区别，特别是中断使能和中断禁止的时机。

11.5 虚拟向量

一些平台本身就具有通过 CygMon 进行调试的能力，但有些体系结构的平台（如 PowerPC、ARM、SH 等）只有在具有 GDB stub 时才支持这种调试功能。eCos 通过提供一个 ROM／RAM 调用接口可以使所有的平台都具有这种调试功能，这种调用接口就是虚拟向量（Virtual Vector）。

虚拟向量是一个位于目标系统静态内存空间的表格，它包含 64 个向量，每一个向量都指向相应的服务程序。基于 ROM 和 RAM 的启动配置都可以访问虚拟向量。虚拟向量的使用使得 ROM 启动配置所提供的服务也可以被 RAM 启动配置中的应用所使用。如果没有虚拟向量，基于 RAM 和 ROM 启动配置的应用是不可能共同使用这些服务的。由于这两种配置都被单独编译和链接，它们具有不同的名字空间，因此不能进行相互间的直接调用。虚拟向量所指向的服务可以实现于 ROM 监控程序，也可以实现于 RAM 应用程序，而且在运行时可以在它们之间进行切换。通过将向量指向一个虚构函数的方法可以禁止这种服务。

公共硬件抽象层内的 hal _ if.h 文件对所有可用的服务进行了定义，它和 hal _ if.c 文件提供了虚拟向量服务的调用接口 API。这些 API 提供了一组服务，不同平台或同一平台中不同版本的 ROM 监控程序所实现的服务可能有多有少，虚拟向量表中不支持的服务应该映射到一个 NOP（空）服务，NOP 服务在被调用时将返回 0（即 FALSE）。

虚拟向量提供的服务有：

VERSION

虚拟向量表的版本，用于检查虚拟向量表提供支持的程度。

KILL _ VECTOR

该向量定义了一个当系统收到一个 kill 信号时调用的函数。

CONSOLE _ PROCS

控制台 IO 使用的通信程序表。

DEBUG _ PROCS

调试器使用的通信程序表。

FLUSH _ DCACHE

Flush 指定区域的数据 Cache。

FLUSH _ ICACHE

Flush（失效）指定区域的指令 Cache。

SET _ DEBUG _ COMM

改变 debug 通信通道。

SET _ CONSOLE _ COMM

改变控制台通信通道。

DBG _ SYSCALL

用于 ROM 内的调试器函数和 RAM 内的调试器函数之间通信的向量。RAM 配置的 eCos 可以在此放置一个函数指针，ROM 监控程序使用该函数指针从运行在 RAM 中的内核获取相关线程信息。

RESET

对系统进行复位。如果不能从软件复位系统，它将跳转至产生软复位的 ROM 入口点。

CONSOLE _ INTERRUPT _ FLAG

在处理控制台 IO 时，如果检测到一个 debuger 中断，则对其进行设置。它允许在返回到 RAM 程序时还可以对断点进行处理。

DELAY _ US

微秒级延时。

FLASH _ CFG _ OP

用于访问保存在 FLASH 内存内的配置信息。

INSTALL _ BPT _ FN

在指定地址处设置断点。可用于异步断点的支持。

虚拟向量的使用既有其有利的一面，也有其不利的一面。在某些场合中，它利大于弊，而在另一些场合中可能是弊大于利。在使用虚拟向量的时候，要根据具体情况进行取舍。

虚拟向量有利的方面包括：

① 可以在没有 stub 时进行调试。

② 可以使用任意通道开始调试工作。

③ 由于是 ROM 监控程序提供相应的服务，因此应用程序映像更小。

虚拟向量的不利因素包括：

① 虚拟向量的使用增加了一个间接层，从而增加了应用程序代码量，降低了系统性能。

② 间接层增加了系统故障的可能性。如果虚拟向量表被破坏，将导致系统崩溃。

③ 间接层增加了硬件抽象层 HAL 的复杂度。

11.6 eCos 的移植

eCos 的硬件抽象层在不同层次上对目标系统的平台硬件和 CPU 体系结构的具体操作进行了描述，这种层次结构使得 eCos 可以很容易地移植到新的平台上。通过硬件抽象层的移植，eCos 的核心功能 (基本构架、内核、μITRON 等等) 能够运行在新的平台上。另外，新的平

台可能还需要一些平台专用的程序,如串口驱动程序、显示驱动程序、网络驱动程序等。

前面已经介绍了硬件抽象层的结构,它主要由三个方面组成,即平台抽象层、变体抽象层和体系结构抽象层。eCos 的移植也将从这三个方面进行,分别为平台抽象层的移植、变体抽象层的移植和体系结构抽象层的移植。下面分别对这三个方面的移植工作进行介绍。

11.6.1 平台抽象层的移植

在三种硬件抽象层的移植中,平台抽象层移植的工作量是最小的。平台抽象层主要包括内存的布局、平台早期初始化程序、中断控制器以及简单串口驱动程序等等。在进行平台抽象层移植的时候,要求已经完成了体系结构抽象层的移植和变体抽象层的移植。一般来说,在进行 eCos 开发时,移植的主要工作在于平台抽象层,eCos 的源码中通常已经包含了相应的体系结构抽象层和变体抽象层。如果要采用当前 eCos 源码不支持的一种新的 CPU 体系结构,就必须进行体系结构抽象层和变体抽象层的移植工作。

1. 平台抽象层的移植过程

对于一个新的平台系统,构造其硬件抽象层的最简单的方法是利用 eCos 源码所提供的具有相同体系结构和 CPU 型号的参考平台硬件抽象层,将其作为模板,复制并修改所有与新平台相关的文件。如果 eCos 没有提供相应的体系结构或 CPU 型号的硬件抽象层,则可以用另外一种体系结构或 CPU 型号的类似的硬件抽象层作为其模板。

⑴ 移植工作的开始。

进行平台硬件抽象层移植的最好途径是从 RedBoot 开始,实现的第一个目标是使 RedBoot 能够在新平台上运行。RedBoot 比 eCos 要简单,它没有使用中断和线程机制,但包含了大部分最基本的启动所需要的功能。RedBoot 通常运行在 Flash 或 ROM 之外,提供程序加载和调试功能。在进行 HAL 的开发过程中,它允许使用 RAM 启动的配置。开发过程可能需要多次进行程序的加载操作,RAM 启动配置可以将程序直接加载到 RAM,而不需要对 Flash 或 ROM 进行反复的更新和擦除。

实现这一目标有两种方法。一种方法是在平台开发板上配备有 ROM 监控程序,可以用其加载并启动运行 ELF、二进制、S-record 或其他格式的映像文件。开发 RedBoot 时,可以直接加载并运行程序代码。如果可能的话,应该在平台上运行一个小的 stub 程序用于检查各种硬件寄存器,这有助于平台初始化程序的编程和调试。当基于 RAM 启动的 RedBoot 可以正常工作时,可以将其配置为 ROM 启动方式并加载到 Flash 或 ROM。如果条件允许,应该保留最初的 ROM 监控程序,在需要的时候可以重新使用它。

另外一种方法是在开发板不具备 ROM 监控程序的情况下,首先必须对平台进行初始化并使其能够运行一小段 stub 程序。这一过程可能需要反复多次对程序进行修改、测试和更新 Flash 或 ROM 的操作。如果可能的话,最好使用一些相应的调试工具。将 RedBoot 从 RAM 启动转到 ROM 启动的最初阶段也可能需要使用这种方法。

⑵ 建造 RedBoot。

在进行移植之前,应该对目标系统硬件平台和 eCos 源码对应的参考平台之间的差异进行分析和了解。除了对目标系统硬件有足够的了解外,还应该通读一遍 eCos 参考源码中的相应部分。建立目标平台的 RedBoot 通常按下列步骤进行:

1) 从 eCos 的源码中复制选定的平台硬件抽象层。根据实际需要对文件进行更名,CDL

(组件定义)和 MLT (内存布局)相关的文件名应该使用<arch>_<variant>_<platform>的组合形式。

2) 调整 CDL 选项。包括选项的名字、实时时钟/计数器、CYGHWR _ MEMORY _ LAYOUT 变量以及其他的一些选项。检查与体系结构和 CPU 型号相关的 CDL 文件,如果在复制的 HAL 中没有它们所需要的 CDL 选项,则增加相应的选项。

3) 在顶层 ecos.db 文件中加入所需要的包,并增加对目标平台的描述。在最初的 ecos.db 文件中,该目标平台的入口可以只包含 HAL 包,其他硬件支持包将在以后陆续加入。

4) 对 include/pkgconf 中的 MLT 文件进行修改,使其符合目标平台的内存布局。在最初的测试阶段,对相应的.h 和.ldi 文件进行手工编辑就可以满足要求,在开发工作进行到一定阶段后,可以使用配置工具中的内存布局编辑器生成所有的文件。

5) 针对所选择的启动类型,对 misc/redboot _<STARTUP>.ecm 进行编辑。对平台专用的选项进行改名,删除不需要的选项。

6) 如果默认的 IO 宏定义不正确,则在 plt _ io.h 中对它们进行定义。

7) 尽可能删除或注释掉使能 Cache 和 MMU 的代码。(在初期开发阶段,执行速度并不重要)

8) 实现简单的串口驱动程序(只使用查询方式)。eCos 提供的参考源码实现了其大部分代码,只需要增加或修改与目标系统硬件访问相关的部分。

9) 修改或增加平台初始化程序。如果是建造一个基于 RAM 启动的 RedBoot,那么可以使用开发板上的 ROM 监控程序对开发板进行初始化,这一步工作可以延缓实现。

10) 定义 HAL _ STUB _ PLATFORM _ RESET 和 HAL _ STUB _ PLATFORM _ RESET _ ENTRY 宏,使 RedBoot 可以进行软复位,不需要每一次加载 RedBoot 都对开发板进行硬复位操作。

上述工作完成后,接下来的工作就是对 RedBoot 进行编译。以一个 ROM 启动的 RedBoot 为例,可以通过下面的方法得到 RedBoot 的映像文件:

```
$ ecosconfig new <target _ name> redboot
$ ecosconfig import $(ECOS _ REPOSITORY)/hal/<architecture>/
                    <platform>/<version>/misc/redboot _ ROM.ecm
$ ecosconfig tree
$ make
```

一旦编译成功,可以增加更多的一些功能和变化。编译完成后生成的 RedBoot 映像文件位于 install/bin 目录内。通过 objcopy 命令可以将其转换为可以更新到 Flash 或 ROM 的适当格式的映像文件。在将其更新到开发板后,复位并运行新的 RedBoot,在串口输出终端将出现下述 RedBoot 界面:

```
RedBoot(tm) bootstrap and debug environment [ROMRAM]
Non-certified release, version UNKNOWN - built 23:02:41, Feb 18 2003
Platform: <PLATFORM> (<ARCHITECTURE> <VARIANT>)
Copyright (C) 2000, 2001, 2002, Red Hat, Inc.
RAM: 0x00000000-0x01000000, 0x000293e8-0x00ed1000 available
FLASH: 0x24000000 - 0x26000000, 256 blocks of 0x00020000 bytes each.
```

(3) 建造平台硬件抽象层。

完成上述步骤的工作之后，目标平台上已经可以运行一个基本的 RedBoot。这说明目标平台已经能够正确地完成初始化操作，并且其串口驱动程序也能正常运行。下一步的工作是对硬件抽象层进行补充，使其形成一个完整的平台硬件抽象层。可以按下面的步骤来完成这一工作。

1) 软复位的实现。前面已经提及到软复位的实现。当 GDB 脱连时，GDB 将给 RedBoot 发送一个 kill 包，RedBoot 首先调用 HAL _ STUB _ PLATFORM _ RESET()，尝试进行软复位操作。如果它不能引起复位，RedBoot 将跳转到 HAL _ STUB _ PLATFORM _ RESET _ ENTRY，这是 CPU 在复位后执行程序的起始地址。在完成 kill 包所引起的复位后，目标系统将再一次准备与 GDB 进行连接。应该注意在使用 GDB 的 detach 命令进行脱连时，有可能不会引起目标系统的复位。

2) 单步、断点支持。在进行调试时，需要使用单步和断点操作。

3) 使用实时时钟 (RTC) 中断驱动 eCos 的调度器时钟。许多嵌入式 CPU 都具有一个内部定时器或减计数器用于这种目的，这时将由体系结构抽象层或变体抽象层提供这种支持，平台抽象层所需要做的工作是计算一个适当的值，让其对平台 CDL 文件中的 CYGNUM _ HAL _ RTC _ CONSTANTS 进行定义。有些目标平台可能需要一个平台专用的时间源来驱动实时时钟，这种情况下除了也要对 CDL 定义给出一个适当的值外，还必须给 HAL _ CLOCK _ XXXX 宏定义一个合适的版本。

4) 中断的译码。在不同的平台中，由于设备数目和设备类型的不一致，中断的译码方式也会不一致。应该在 plt _ intr. h 文件中对体系结构抽象层或变体抽象层所定义的默认中断向量进行扩充或替换。另外，还必须对 HAL _ INTERRUPT _ XXXX 的控制宏进行定义。

5) Cache 的定义和操作。不同体系结构和 CPU 型号的 Cache 可能是不一样的，Cache 的大小与粒度可能也有区别。在一些支持多级 Cache 的平台中，Cache 的差异更大。在使用 Cache 的系统中，在系统启动到一定阶段时要使能 Cache。在开发过程中，首先应该验证系统在 RAM 启动时的稳定性，然后再生成新的基于 ROM 启动的 RedBoot，并测试 Cache 的使能、同步和 Flush 操作的正确性。

6) 异步断点支持。使用异步断点可以停止应用程序的执行并进入 debug 进行调试工作。

通过上述步骤可以得到一个完整的平台硬件抽象层。此时可以运行 eCos 源码提供的所有测试程序来验证其稳定性和完整性。由于使用了 RedBoot，因此可以利用它对所有出现的错误进行调试。

2. 平台硬件抽象层 CDL 描述

平台 CDL 包含了建立 eCos 系统的一些详细信息和平台专用的配置选项。不同平台之间的选项不完全相同。读者可以参阅第 12 章并参考 eCos 源码中的 CDL 文件来了解如何使用 CDL 语言对各种配置选项的表述方法。这里只简单描述一些最普通的选项。

(1) eCos 数据库。

eCos 配置系统通过使用一个数据库文件 ecos. db 来了解它的包的信息。对于一个新的硬件平台，必须在 ecos. db 中增加对该平台相应的包的描述，包括包的名字、说明、包的目录位置和相应的 CDL 文件。在配置工具选择该包时，配置工具中将会出现该包的名字和相应的描述

信息。以 TX39/JMR3904 平台为例,它的 CDL 描述如下:

```
package CYGPKG_HAL_MIPS_TX39_JMR3904 {
alias       {"Toshiba JMR-TX3904 board" hal_tx39_jmr3904 tx39_jmr3904_hal}
directory hal/mips/jmr3904
script      hal_mips_tx39_jmr3904.cdl
hardware
description "
            The JMR3904 HAL package should be used when targetting the
            actual hardware. The same package can also be used when
            running on the full simulator, since this provides an
            accurate simulation of the hardware including I/O devices.
            To use the simulator in this mode the command
            'target sim --board=jmr3904' should be used from inside gdb."
}
```

为了对新平台所进行的配置进行编译和测试,在 ecos.db 文件中还应该包含一个 target 项:

```
target jmr3904 {
    alias       {"Toshiba JMR-TX3904 board" jmr tx39}
  packages      {CYGPKG_HAL_MIPS
                 CYGPKG_HAL_MIPS_TX39
                 CYGPKG_HAL_MIPS_TX39_JMR3904
                 CYGPKG_IO_SERIAL_TX39_JMR3904
                }
  description "The jmr3904 target provides the packages needed to run
          eCos on a Toshiba JMR-TX3904 board. This target can also
          be used when running in the full simulator, since the
          simulator provides an accurate simulation of the hardware
          including I/O devices. To use the simulator in this mode the
          command 'target sim --board=jmr3904' should be used from
          inside gdb."
}
```

这里最重要的部分是 packages 段,它定义了用于支持该平台的各种硬件专用包。上面例子中包含了 MIPS 体系结构包、TX39 CPU 型号包和 JMR-TX3940 平台包。其他包(如串口驱动程序、网络驱动程序和 Flash 内存驱动程序)也可以包含在此。

⑵ CDL 文件。

平台的所有选项都包含在一个 CDL 包内,该包的名字是 CYGPKG_HAL_<architecture>_<variant>_<platform>。下面是其一个例子:

```
cdl_package CYGPKG_HAL_MIPS_TX39_JMR3904 {
    display             "JMR3904 evaluation board"
    parent              CYGPKG_HAL_MIPS
```

```
        requires            CYGPKG_HAL_MIPS_TX39
        define_header hal_mips_tx39_jmr3904.h
        include_dir         cyg/hal
        description         "
                The JMR3904 HAL package should be used when targeting the
                actual hardware. The same package can also be used when
                running on the full simulator, since this provides an
                accurate simulation of the hardware including I/O devices.
                To use the simulator in this mode the command
                'target sim --board=jmr3904' should be used from inside gdb."
        compile             platform.S plf_misc.c plf_stub.c
        define_proc {
            puts $::cdl_system_header "#define CYGBLD_HAL_TARGET_H \
<pkgconf/hal_mips_tx39.h>"
            puts $::cdl_system_header "#define CYGBLD_HAL_PLATFORM_H \
<pkgconf/hal_mips_tx39_jmr3904.h>"
        }
        ...
    }
```

包中内容说明了该平台包属于 MIPS **包,需要** TX39 **的变体抽象层,所有的配置设置被保存在** cyg/hal/hal_mips_tx39_jmt3904.h **内。其中的** compile **说明了当使能该包时需要被编译的文件,**define_proc **定义了一些用于访问** CPU **型号或体系结构和平台配置选项的宏。**

(3) 启动类型。

eCos **提供了一个配置选项来选择各种启动类型。启动类型通常有** RAM、ROM、ROM-RAM、FLOPPY **和** Grub**。启动类型的** CDL **包例子如下:**

```
cdl_component CYG_HAL_STARTUP {
    display             "Startup type"
    flavor              data
    legal_values        {"RAM" "ROM"}
    default_value       {"RAM"}
no_define
define -file system.h CYG_HAL_STARTUP
    description         "
        When targeting the JMR3904 board it is possible to build
        the system for either RAM bootstrap, ROM bootstrap, or STUB
        bootstrap. RAM bootstrap generally requires that the board
        is equipped with ROMs containing a suitable ROM monitor or
        equivalent software that allows GDB to download the eCos
        application on to the board. The ROM bootstrap typically
        requires that the eCos application be blown into EPROMs or
        equivalent technology."
```

```
}
```

上面例子中的 no _ define **和** define **用于将文件** system. h **中出现的该选项的值替换头文件中的默认值。**

(4) 编译选项。

在 CDL **组件** CYGBLD _ GLOBAL _ OPTIONS **和** CYGHWR _ MEMORY _ LAYOUT **下有一组选项,这些选项指定了如何对** eCos **进行编译,内容包括编译工具、编译选项、使用哪一个链接器脚本段等。例如:**

```
cdl _ component CYGBLD _ GLOBAL _ OPTIONS {
    display "Global build options"
    flavor none
    parent CYGPKG _ NONE
    description "
    Global build options including control over
    compiler flags, linker flags and choice of toolchain."
    cdl _ option CYGBLD _ GLOBAL _ COMMAND _ PREFIX {
        display "Global command prefix"
        flavor data
        no _ define
        default _ value {"mips-tx39-elf"}
        description "
            This option specifies the command prefix used when
            invoking the build tools."
    }
    cdl _ option CYGBLD _ GLOBAL _ CFLAGS {
        display "Global compiler flags"
        flavor data
        no _ define
        default _ value {"-Wall -Wpointer-arith -Wstrict-prototypes -Winline -Wundef -Woverloaded-
virtual -g -O2 -ffunction-sections -fdata-sections -fno-rtti -fno-exceptions -fvtable-gc -finit-priority"}
        description "
            This option controls the global compiler flags which
            are used to compile all packages by
            default. Individual packages may define
            options which override these global flags."
    }
    cdl _ option CYGBLD _ GLOBAL _ LDFLAGS {
    display "Global linker flags"
    flavor data
    no _ define
    default _ value {"-g -nostdlib -Wl,--gc-sections -Wl,-static"}
    description "
```

```
            This option controls the global linker flags.  Individual
            packages may define options which override these global flags."
        }
    }
    cdl_component CYGHWR_MEMORY_LAYOUT {
        display "Memory layout"
        flavor data
        no_define
        calculated {CYG_HAL_STARTUP == "RAM" ?"mips_tx39_jmr3904_ram" :\
                                                "mips_tx39_jmr3904_rom"}
        cdl_option CYGHWR_MEMORY_LAYOUT_LDI {
            display "Memory layout linker script fragment"
            flavor data
            no_define
            define -file system.h CYGHWR_MEMORY_LAYOUT_LDI
            calculated      {CYG_HAL_STARTUP == "RAM" ? \
"<pkgconf/mlt_mips_tx39_jmr3904_ram.ldi>" :\
                    "<pkgconf/mlt_mips_tx39_jmr3904_rom.ldi>"}
        }
        cdl_option CYGHWR_MEMORY_LAYOUT_H {
            display "Memory layout header file"
            flavor data
            no_define
            define -file system.h CYGHWR_MEMORY_LAYOUT_H
            calculated {CYG_HAL_STARTUP == "RAM" ?\
                    "<pkgconf/mlt_mips_tx39_jmr3904_ram.h>" :\
                        "<pkgconf/mlt_mips_tx39_jmr3904_rom.h>"}
        }
    }
```

CDL 文件除了上述内容外，还包括一些公共的目标系统选项。这些选项与实时时钟 RTC、输出通道、ROM 监控程序等相关。在支持 RedBoot 的平台中，还有一些选项用于对 RedBoot 进行配置。

3. 内存布局

eCos 使用配置工具中的内存配置窗口对平台的内存布局进行定义。如果不使用窗口配置工具，也可以手工对 .h 和 .ldi 文件进行编辑。内存配置的详细信息被保存在下面的三个文件中：

① mlt 文件。这是配置工具保存文件，只能被配置工具使用。

② ldi 文件。链接脚本文件。使用体系结构和 CPU 链接器脚本中的宏定义对内存和段地址进行定义。

③ h 文件。该文件使用 C 宏定义描述一些内存区的细节，允许 eCos 或其应用程序适应指定配置中的内存布局。

为满足一些特殊需要，应该为系统保留一些内存区，这些内存区可以为例外向量和各种表提供空间。基于 RAM 启动的配置还应该在内存空间的底部为 ROM 监控程序保留一部分空间。这些保留空间的名字都有一个前缀“reserved _”，配置工具将对它们进行特殊的处理。

4. 串口设备支持

在移植过程中，为支持串口设备的第一步工作就是建立 CDL 定义。有下面的这些配置选项需要进行设置：

CYGNUM _ HAL _ VIRTUAL _ VECTOR _ COMM _ CHANNELS

串口通道数，通常为 0、1 或 2；

CYGNUM _ HAL _ VIRTUAL _ VECTOR _ DEBUG _ CHANNEL

GDB 所使用的串口；

CYGNUM _ HAL _ VIRTUAL _ VECTOR _ DEBUG _ CHANNEL _ BAUD

Debug 串口的初始波特率；

CYGNUM _ HAL _ VIRTUAL _ VECTOR _ CONSOLE _ CHANNEL

用于控制台的串口通道；

CYGNUM _ HAL _ VIRTUAL _ VECTOR _ CONSOLE _ CHANNEL _ BAUD

串口控制台的初始波特率；

CYGNUM _ HAL _ VIRTUAL _ VECTOR _ CONSOLE _ CHANNEL _ DEFAULT

默认的控制台串口通道。

文件 hal _ diag. c 中的程序代码需要进行修改，使其支持新的串口设备。如果是同样的设备，则只需对其进行复制。为支持新的串口设备，需要修改的函数和数据类型有：

struct channel _ data _ t；

包含所支持的每一个串口设备的基地址、超时时间、ISR 向量号。如果需要，还可以增加新的域。

xxxx _ ser _ channels []；

channel _ data _ t 数组，被初始化为每一个串口通道的参数。数组的索引为 CDL 选项中的通道号。

void cyg _ hal _ plf _ serial _ init _ channel (void * __ ch _ data)

对串口设备进行初始化。参数是一个指向 channel _ data _ t 的指针。

void cyg _ hal _ plf _ serial _ putc (void * __ ch _ data, char * c)

给串口设备发送一个字符。该函数查询设备发送字符的准备状态，一旦就绪，就给设备发送一个字符。在实现时，最好还应该使用查询的方式来确认发送操作的完成。

bool cyg _ hal _ plf _ serial _ getc _ nonblock (void * __ ch _ data, cyg _ uint8 * ch)

从串口设备读取一个字符。该函数对设备进行检测，如果有一个字符从设备发送过来，则将其放置到 * ch 并返回 TRUE。如果没有字符出现，则立即返回 FALSE。

```
int cyg_hal_plf_serial_control(void *__ch_data, __comm_control_cmd_t __func, ...)
```

这是一个类似 IOCTL 的函数，对串口设备的各个方面进行控制。这里所需要做的工作是在__COMMCTL_IRQ_ENABLE 和__COMMCTL_IRQ_DISABLE 情形下使能和禁止中断。

```
int cyg_hal_plf_serial_isr(void *__ch_data, int *__ctrlc, CYG_ADDRWORD __vector, CYG_ADDRWORD __data)
```

这是一个伪中断向量调用的中断处理程序，专门处理从 GDB 来的 Ctrl-C 中断。调用该函数所完成的工作如下：

① 检查输入字符，类似于 cyg_hal_plf_serial_getc_nonblock()。

② 读取字符并调用 cyg_hal_is_break()。

③ 如果结果为 true，设置 *__ctrlc 为 1。

④ 返回 CYG_ISR_HANDLED。

```
void cyg_hal_plf_serial_init()
```

对每一个串口通道进行初始化。首先对每一个通道调用 cyg_hal_plf_serial_init_channel()，然后再对每一个通道调用宏 CYGACC_COMM_IF_*。后面这一组宏调用对所有通道都是相同的，因此可以使用已有的例子进行复制和编辑。

11.6.2 变体抽象层的移植

变体抽象层的移植相对来说还是比较简单，但也有一定的工作量。变体抽象层所描述的是指定 CPU 与同类体系结构的普通 CPU 之间的差异。变体抽象层可以对 Cache、MMU、中断和其他一些特性进行重定义，这种重定义将覆盖体系结构抽象层中的默认实现。在进行变体抽象层的移植之前，应该已经完成了体系结构抽象层的移植。

1. 变体抽象层的移植过程

与平台抽象层的移植一样，建造一个新的变体抽象层最容易的方法也是复制 eCos 源码中的一个类似的变体抽象层，然后再对所有与新的 CPU 变体相关的文件进行修改。如果是某体系结构中的第一个变体，要确定变体抽象层中应该包含哪些部分是比较困难的，这需要对其他的结构变体有所了解。

通常，不同变体之间的 Cache、中断以及例外处理可能会有所不同。初始化程序、处理各种核心部件（FPU、DSP、MMU 等）的程序也可能不一致，或者某些变体就根本不存在这些部件。它们的链接脚本可能也不尽相同。另外，某些 CPU 的变体可能还需要专用的编译器。

2. 变体抽象层的 CDL 描述

变体抽象层中的 CDL 往往依赖于变体所支持的具体功能。如果它实现了某些在平台抽象层中描述过的设备，那么这些设备的 CDL 应该位于变体抽象层，而不是在平台抽象层。

在 eCos 数据库文件 ecos.db 中，每一个变体都应该有一个入口。以 MIPS 的 VR4300 为例，它在 ecos.db 中有它的一个入口：

```
package CYGPKG_HAL_MIPS_VR4300 {
    alias       {"VR4300 chip HAL" hal_vr4300 vr4300_hal vr4300_arch_hal}
    directory hal/mips/vr4300/
    script      hal_mips_vr4300.cdl
    hardware
        description "
            The VR4300 variant HAL package provides generic support
            for this processor architecture.  It is also necessary to
            select a specific target platform HAL package."
}
```

可以看出,变体入口非常类似于平台入口。

下面介绍的是变体抽象层的 CDL 文件。

变体 CDL 文件包含一个以体系结构和变体命名的一个包的入口,该包的名字将出现在 ecos.db 文件内。下面是 MIPS VR4300 CDL 文件的开始部分:

```
cdl_package CYGPKG_HAL_MIPS_VR4300 {
    display             "VR4300 variant"
    parent              CYGPKG_HAL_MIPS
    implements          CYGINT_HAL_MIPS_VARIANT
    hardware
    include_dir         cyg/hal
    define_header hal_mips_vr4300.h
    description "
        The VR4300 variant HAL package provides generic support
        for this processor architecture.  It is also necessary to
        select a specific target platform HAL package."
```

上面这一部分对包进行了定义,并将其置于 MIPS 体系结构包之下。其中的 implements 指出这是一个 MIPS 的变体。体系结构包使用它来检查具体被配置的是哪一个变体。

变体定义了一些选项,这些选项使体系结构抽象层对自己进行配置,使其支持该变体。还是以前面所举的 CDL 文件为例:

```
cdl_option CYGHWR_HAL_MIPS_64BIT {
    display       "Variant 64 bit architecture support"
    calculated 1
}
cdl_option CYGHWR_HAL_MIPS_FPU {
    display       "Variant FPU support"
    calculated 1
}
cdl_option CYGHWR_HAL_MIPS_FPU_64BIT {
    display       "Variant 64 bit FPU support"
    calculated 1
```

```
}
```

上面这些选项告诉体系结构这是一个 64 位的 MIPS 体系结构，它具有一个浮点部件，应该使用 64 位方式而不是 32 位方式。

CDL 文件最后还包含了一些编译选项，例如：

```
define _ proc {
puts $ ::cdl _ header "#include <pkgconf/hal _ mips.h>"
}
compile       var _ misc.c
make {
<PREFIX>/lib/target.ld:<PACKAGE>/src/mips _ vr4300.ld
$ (CC) -E -P -Wp,-MD,target.tmp -DEXTRAS=1 -xc $ (INCLUDE _ PATH) $ (CFLAGS) -o $
@ $ <
@echo $ @ ": \ \" > $ (notdir $ @).deps
@tail +2 target.tmp >> $ (notdir $ @).deps
@echo >> $ (notdir $ @).deps
@rm target.tmp
}
cdl _ option CYGBLD _ LINKER _ SCRIPT {
display "Linker script"
flavor data
no _ define
calculated {"src/mips _ vr4300.ld"}
}
```

其中的 define _ proc 为该变体将体系结构配置文件包含到配置文件中。compile 指出需要对文件 var _ misc.c 进行编译。make 指出了一些编译规则，与链接脚本组合成 .ldi 文件，用以产生 target.ld。在 MIPS 体系结构中，主链接脚本是在变体中定义而不是在体系结构中定义，因此在这里定义了 CYGBLD _ LINKER _ SCRIPT。

3. Cache 支持

变体抽象层中的主要部分可能是对 Cache 的支持。CPU 的变体常常是根据其 Cache 的大小粒度进行区分的。在一些体系结构中，大多数的 Cache 操作都放在体系结构抽象层进行，变体抽象层只对 Cache 的粒度进行定义。下面是 MIPS VR4300 在 var _ cache.h 中对 Cache 的粒度进行定义的例子：

```
//Data cache
#define HAL _ DCACHE _ SIZE          (8 * 1024)      //Size of data cache in bytes
#define HAL _ DCACHE _ LINE _ SIZE   16              //Size of a data cache line
#define HAL _ DCACHE _ WAYS          1               //Associativity of the cache
//Instruction cache
#define HAL _ ICACHE _ SIZE          (16 * 1024)     //Size of cache in bytes
#define HAL _ ICACHE _ LINE _ SIZE   32              //Size of a cache lin
```

```
#define HAL_ICACHE_WAYS          1                    //Associativity of the cache
#define HAL_DCACHE_SETS (HAL_DCACHE_SIZE/(HAL_DCACHE_LINE_SIZE * HAL
_DCACHE_WAYS))
#define HAL_ICACHE_SETS (HAL_ICACHE_SIZE/(HAL_ICACHE_LINE_SIZE * HAL_
ICACHE_WAYS))
```

在此还可以对其他的一些 Cache 宏进行定义,或者对默认定义进行覆盖。有些体系结构具有管理 Cache 行的一些指令,Cache 的使能和禁止操作可能全部通过变体的专用寄存器进行,在这种情况下 var_cache.h 还应该定义 HAL_XCACHE_ENABLE() 宏和 HAL_XCACHE_DISABLE() 宏。

如果变体不支持某些普通特性,则 var_cache.h 要禁止对某些操作的定义。具体实现方法与体系结构有关。

11.6.3 体系结构抽象层的移植

对于一个新的体系结构来说,其体系结构抽象层的建立相对比较困难。eCos 源码支持大部分当前广泛使用的嵌入式 CPU,它已经具有了支持各种体系结构的硬件抽象层。因此,在进行 eCos 开发时很少需要编写新的体系结构抽象层。这一节主要介绍体系结构硬件抽象层的移植过程。

1. 体系结构抽象层的移植过程

进行体系结构抽象层移植的最简单的方法是选择一个体系结构最接近的硬件抽象层,对其进行复制,然后根据新的体系结构的特点对所有需要改变的源码文件进行修改。MIPS 体系结构抽象层的层次结构和编程约定都比较规范,如果适当的话可以使用它的体系结构抽象层作为模板。

eCos 采用 GCC 作为它的编译工具,在进行 eCos 开发时需要相应的 C/C++ 编译器及其适当版本的支持。值得注意的是 C++ 不支持 8 位和 16 位的 CPU。在进行 eCos 的移植操作之前,应该确认有相应的编译器支持。

下面是对一个新的体系结构进行体系结构抽象层移植的一些步骤。对于不同的体系结构,它们的顺序可能不完全一致。在对体系结构抽象层进行移植测试时,变体抽象层和平台抽象层的移植工作也应该同时进行。

1) 在 eCos 源码库的 hal 目录下为新的体系结构建立一个目录,在此目录下创建 arch 子目录,并按标准建立所有的包目录(可仿照其他体系结构抽象层)。

2) 从模板 HAL 复制 CDL 文件,改变其名字使其与新的体系结构相匹配。对 CDL 文件进行编辑,修改选项的名字。删除那些原体系结构专用的选项,增加新体系结构所需要的一些选项。

3) 从模板 HAL 复制文件 hal_arch.h。对该文件进行下述修改:

① 定义 HAL_SavedRegisters 结构。该结构必须能够体现寄存器的保存顺序、中断和例外的保存格式以及程序调用约定。可能的话,还必须满足可选的浮点部件 FPU 和其他功能部件的需要。

② 定义位索引处理程序 HAL_LSBIT_INDEX() 和 HAL_MSBIT_INDEX()。如果体系结构具有这样的指令和相关操作,则应该将其定义为内嵌汇编指令段。

③ 定义 HAL _ THREAD _ INIT _ CONTEXT ()。它对一个可恢复的 CPU 上下文环境进行初始化，使后面的 HAL _ THREAD _ LOAD _ CONTEXT () 调用和 HAL _ THREAD _ SWITCH _ CONTEXT () 调用能够正常执行。

④ 定义上下文切换函数 HAL _ THREAD _ LOAD _ CONTEXT () 和 HAL _ THREAD _ SWITCH _ CONTEXT ()。

⑤ 定义排序栅栏函数 HAL _ REORDER _ BARRIER ()。程序在必须顺序执行的地方可以用它来防止编译器优化时对程序代码的移动。该宏在所有体系结构中的实现都是相同的，不需要改动。

⑥ 对断点支持进行定义。宏 HAL _ BREAKPONT (label) 将产生一个断点，它是一个内嵌汇编指令段。在所有体系结构中，该宏都是相同的，不需要修改。

⑦ 对 GDB 的支持进行定义。GDB 将目标系统的寄存器当成一个线性数组，每一个寄存器都有一个已定义好的偏移。该数组可以与 HAL _ SavedRegisters 中的顺序不一致。HAL _ GET _ GDB _ REGISTERS () 和 HAL _ SET _ GDB _ REGISTERS () 两个宏用于在 GDB 的寄存器数组和 HAL _ SavedRegister 结构之间进行转换。HAL _ THREAD _ GET _ SAVED _ REGISTERS () 用于将上下文切换宏所保存的堆栈指针转换为一个 HAL _ SavedRegisters 结构的指针。

⑧ 对长跳转支持进行定义。类型 hal _ jmp _ buf 以及 hal _ stjmp () 函数和 hal _ longjmp () 函数为 C 库函数 setjmp () 和 longjmp () 提供底层实现。

⑨ 对线程空闲行为 (宏 HAL _ IDLE _ THREAD _ ACTION ()) 进行定义。

⑩ 定义栈空间大小。宏 CYGNUM _ HAL _ STACK _ SIZE _ MINIMUM 和 CYGNUM _ HAL _ STACK _ SIZE _ TYPICAL 定义线程堆栈的最小值和默认值。在 HAL 的一些汇编程序中可能要用到它们，因此应该使用数字值进行定义。

⑪ 定义内存访问宏。这些宏在 Cache 空间、非 Cache 空间和物理内存之间进行转换。

⑫ 定义全局指针保存/恢复宏。只有在调用约定需要一个全局指针的体系结构 (如 MIPS) 中才需要对这种宏进行定义，不需要时应该为空。

4) 从模板 HAL 中复制文件 hal _ intr. h，对其进行如下修改：

① 定义例外向量。CPU 体系结构的说明文档中有例外向量的详细描述，在进行 eCos 移植之前必须仔细阅读相应的资料。CPU 的每一个例外入口点在 VSR 表中都应该有对应的一个表项，VSR 表项的偏移使用 CYGNUM _ HAL _ VECTOR _ * 的形式进行定义。VSR 表的大小也应该在此定义。

② 将硬件例外定义一个标准名字，例外向量的名字通常采用 CYGNUM _ HAL _ EXCEPTION _ * 的标准形式。与结构无关的一段通用程序通过检测是否有 CYGNUM _ HAL _ EXCEPTION _ * 的定义来确定该体系结构是否有这种例外，如果有定义，则其值就是该例外的向量。这种对应关系可以不是一对一的，几个不同的 CYGNUM _ HAL _ EXCEPTION _ * 定义可以具有相同的值。中断向量通常在平台和变体抽象层中进行定义。中断号范围和 VSR 号范围可能是连续的，这时它们共享一个向量表 (如 i386)；它们也可能处于不同的范围，这种情况要对它们进行译码，MIPS 和 PowerPC 就是这种情况。

③ 对硬件抽象层用于处理中断和例外的所有静态数据进行声称。中断通常有三个向量：hal _ interrupt _ handlers []，hal _ interrupt _ data [] 和 hal _ interrupt _ objects []，要根据中断向量

的定义对它们的大小进行定义。除了 VSR 表外，还应该定义 hal _ vsr _ table []。这些向量通常在 vectors. S 或 hal _ misc. c 中进行定义。

④ 定义中断使能/禁止宏。这些宏通常是内嵌汇编指令段，这些程序段对包含有 CPU 中断使能位的寄存器进行操作或执行相应的指令。

⑤ 许多硬件抽象层还具有在中断堆栈中执行 DSR 的能力。对这种能力的支持并不是必需的，在移植的最初阶段最好不要实现。如果需要这种特性，应该对宏 HAL _ INTERRUPT _ STACK _ CALL _ PENDING _ DSRS () 进行定义。

⑥ 定义中断和 VSR 的连接。

5) 对其他的头文件进行修改：

① basetype. h。该文件包含了 eCos 使用的基本类型的定义，以及字节排列方式和其他的一些特性定义。只有在该体系结构与 cyg _ type. h 中的默认定义不一致时，才需要对该文件进行修改。

② hal _ io. h。该文件包含了对设备 IO 寄存器进行访问的宏定义。如果体系结构采用内存映射 IO，则可以从另外的体系结构 (如 MIPS) 中复制而不需要修改。如果体系结构使用专用的 IO 指令，则需要将这些宏以内嵌汇编程序段的形式进行定义 (可参考 i386 的 HAL)。访问 PCI 总线的宏通常在变体抽象层或平台抽象层中定义。

③ hal _ cache. h。该文件包含了 Cache 访问宏定义。如果该体系结构具有 Cache 访问的指令或控制寄存器，则应该在此文件中定义这些宏。否则，应该在变体抽象层或平台抽象层中进行定义。Cache 的粒度 (整个 Cache 的大小、Cache 行大小、Cache 组相联数等等) 通常在变体抽象层中定义。

④ arch. inc 和 <architecture>. inc。这两个头文件是 vector. S 和 context. S 所使用的汇编程序头文件。<architecture>. inc 是一个通用头文件，包含了寄存器别名、ABI (应用软件二进制接口) 定义和普通汇编程序所使用的宏等内容。如果体系结构中没有这些定义，则不需要提供这些文件。arch. inc 包含了其他一些宏定义，它们提供了与 eCos 相关的各种操作，如 CPU、Cache、FPU 的初始化等。这些文件内所进行的定义有可能在变体抽象层或平台抽象层中被重新配置或覆盖。

6) 编写 vector. S。这是硬件抽象层中最重要的一个文件，它包含了 CPU 初始化程序、例外和中断处理程序。对于一个新的体系结构来说，该文件的大部分都需要重新编写。该文件需要提供的主要程序段包括：

① 复位向量程序。复位向量程序通常位于 ROM 或 FLASH 的起始位置，因此在它自己的链接脚本段中应该体现复位 (reset) 向量。通过链接脚本，可以将复位向量放置到正确的位置。通常复位向量程序很小，可能仅是一条跳转到标号为_ start 的语句。

② 例外向量程序。这些例外向量程序是一些跳转程序，根据 VSR 表跳转到硬件例外入口点。在许多体系结构中，它们处于复位向量邻近的位置，可以使用同一个链接脚本段。跳转程序提供的支持至少应该是能够将控制权从硬件向量转移给 VSR，具体实现方法与体系结构有关。

③ 内核启动程序。这是从复位向量跳转过来的程序。它首先对 CPU 和其他硬件组件进行初始化，此时最好使用 arch. inc 或者变体抽象层和平台抽象层中所提供的宏调用函数。它所要完成的其他工作有 堆栈指针的初始化、根据需要将数据从 ROM 复制到 RAM、将 BSS 数

据段清零、调用变体初始化程序和平台初始化程序、调用 cyg _ hal _ invoke _ constructors () 函数、根据需要调用 initialize _ stub () 函数。最后它将调用 cyg _ start () 函数。

④ 默认例外 VSR。该 VSR 被装入到 VSR 表中的所有同步例外向量表项。(见第 11.4.2 节)

⑤ 默认中断 VSR。该 VSR 被装入到 VSR 表中的所有与外部中断相关的表项。(见第 11.4.2 节)

⑥ hal _ interrupt _ stack _ call _ pending _ dsrs () 函数。如果在 hal.arch.h 中有该函数的定义,则应该在此实现。该函数的目的是在中断堆栈中调用 DSR,而不是在当前线程堆栈中调用。该函数完成如下工作:

- 复制当前堆栈指针 SP,然后转至中断堆栈。
- 在中断堆栈中保存老的堆栈指针 SP、CPU 状态寄存器(或者任何一个包含中断使能状态的寄存器)和其他在函数调用中可能会被破坏的寄存器。
- 使能中断。
- 调用 cyg _ interrupt _ call _ pending _ dsrs ()。这是一个调用正处于悬挂状态的 DSR 的内核函数。
- 从中断堆栈中恢复被保存的寄存器,并转回到当前线程堆栈。
- 将被保存的 CPU 状态寄存器中的中断使能状态与当前状态寄存器的值进行合并,恢复以前的使能状态。如果状态寄存器不包含其他任何稳定不变的状态,则只简单地进行寄存器的恢复。但如果寄存器包含了其他可能已被 DSR 改变的状态位,则应该加以小心,不要破坏它们。
- 定义所有所需的数据项。vectors.S 通常包含了 VSR 表、中断表和中断堆栈的定义。

7) 编写 context.S。它包含了上下文切换程序。(详细信息见 11.3.2 节)

8) 编写 hal _ misc.c。该文件包含了 HAL 所需要的 C 函数和数据,可能包括如下内容:

① hal _ interrupt _ * []。在某些硬件抽象层中,如果在 vector.S 内没有定义这些数组,则应该在此进行定义。

② cyg _ hal _ exception _ handler ()。该函数将被例外 VSR 调用。它通常对例外进行译码,针对 FPU 陷阱、总线错误或内存例外等事件调用某些专用的处理程序。如果对于某个例外没有专用的例外程序可调用的话,它将通过__ handle _ exception () 调用 GDB stub 程序或者通过 cyg _ hal _ deliver _ exception () 对内核进行调用。

③ hal _ arch _ default _ isr ()。hal _ interrupt _ handlers [] 数组被初始化后通常指向默认的中断服务程序 hal _ default _ isr (),这是公共 HAL 所定义的一个函数,它处理象 Ctrl-C 这样的一些事件。如果这些事件与该函数不相关,则它将调用 hal _ arch _ default _ isr () 函数。正常情况下,该函数应该只返回 0。

④ cyg _ hal _ invoke _ constructors ()。该函数在程序启动之前为所有的静态对象调用构造器(constructor)。eCos 依靠这些调用(以正确的顺序)来实现其功能的正确性。不同体系结构中对构造器的处理可能不一样,大多数系统使用在标号__ CTOR _ LIST __ 和__ CTOR _ END __之间的一个简单的函数指针表进行处理,必须按自顶向下的顺序进行调用。一般来说,可以从一个已有的体系结构抽象层中之间复制该函数。

⑤ 位索引函数。如果宏 HAL _ LSBIT _ INDEX () 和 HAL _ MSBIT _ INDEX () 被定义为

函数调用，则应该在此提供这些函数。

⑥ hal _ delay _ us()。如果在 hal _ intr.h 中定义了宏 HAL _ DELAY _ US()，则应该将其定义为调用该函数。

⑦ hal _ idle _ thread _ action()。如果调用了宏 HAL _ IDLE _ THREAD _ ACTION()，那么空闲线程将使用此宏调用该函数。正常情况下该函数不做任何事情，但在开发过程中，可以在该函数中实现将系统重要信息输出到输出设备。它可以用来监视系统状态并报告系统的异常。如果体系结构支持 halt 指令，可以在此函数中加入内嵌汇编程序段来执行这一指令。此外，它还是对系统节能行为进行处理的一个理想之处。

9）生成＜architecture＞.ld 文件。虽然该文件最后可能要被移至变体抽象层内，但应该先在此对其进行定义，只有在需要的时候才将其移走。该文件定义了一组宏，目标平台的.ldi 文件将用这些宏生成链接脚本。由于大多数 GCC 工具非常相似，因此可以从已有的体系结构抽象层中复制该文件，需要编辑的地方主要是 OUTPUT _ FORMAT() 的定向，另外可能还需要增加一些宏定义。

10）如果要在 RedBoot 或 eCos 中支持 GDB 的 stub 程序，则应该实现这种支持。提供这种支持的主要有两个文件：include/＜architecture＞-stub.h 和 src/＜architecture＞-stub.c。前一个文件包含了一些 GDB stub 程序用来描述大小、类型、数字和 CPU 寄存器名字的定义，GDB 对该体系结构的支持文档说明提供了相关信息。它还包含了 src/＜architecture＞-stub.c 中的一些函数协议，这些函数协议可以从其他 HAL 中复制。后一个文件对其头文件中定义的函数进行实现。可以参考其他 HAL 来了解在此需要做哪些工作。较为复杂的是对单步的支持，GDB 经常要使用单步操作对程序进行调试。

2. 对 CDL 的要求

体系结构抽象层对 CDL 的要求在很大程度上要依赖于该体系结构的需求。这包括对不同变体 CPU 的支持以及 FPU、MMU 和 Cache 的使用等等。另外，体系结构抽象层、变体抽象层和平台抽象层之间的划分也具有一定的模糊性。下面以 i386 为例粗略介绍如何建立一个体系结构的 CDL 文件。

CDL 文件首先要对该 CDL 包进行描述，并将其置于主 HAL 包（CYGPKG _ HAL）之下。它必须指出 CDL 包所涉及的包含文件位于 include/cyg/hal 目录，CDL 文件的定义将出现在头文件 include/pkgconf/hal _ i386.h 内。在对该包进行编译时，src 目录中被编译的文件由 compile 指定。

下面是 i386 体系结构抽象层的 CDL 描述文件：

```
cdl _ package CYGPKG _ HAL _ I386 {
    display          "i386 architecture"
    parent           CYGPKG _ HAL
    hardware
    include _ dir    cyg/hal
    define _ header  hal _ i386.h
    description      "
        The i386 architecture HAL package provides generic
        support for this processor architecture.  It is also
```

```
    package."
    compile       hal_misc.c context.S i386_stub.c hal_syscall.c
```

完成上面的工作后,需要指明编译规则,用于产生所需的文件。首先是对 vector.S **的编译,它的编译结果不进入库,而是与应用程序直接进行链接。其次是由** i386.ld **文件和选择启动类型的**.ldi **文件生成的** target.ld**。这些内容可以从模板** CDL **文件中直接复制,再进行适当的修改。**i386 **的这种** CDL **描述为:**

```
make {
    <PREFIX>/lib/vectors.o :<PACKAGE>/src/vectors.S
    $(CC) -Wp,-MD,vectors.tmp $(INCLUDE_PATH) $(CFLAGS) -c -o $@ $<
    @echo $@ ": \ \" > $(notdir $@).deps
    @tail +2 vectors.tmp >> $(notdir $@).deps
    @echo >> $(notdir $@).deps
    @rm vectors.tmp
}
make {
    <PREFIX>/lib/target.ld :<PACKAGE>/src/i386.ld
    $(CC) -E -P -Wp,-MD,target.tmp -DEXTRAS=1 -xc $(INCLUDE_PATH) $(CFLAGS)
-o $@ $<
    @echo $@ ": \ \" > $(notdir $@).deps
    @tail +2 target.tmp >> $(notdir $@).deps
    @echo >> $(notdir $@).deps
    @rm target.tmp
}
```

如果支持 SMP**,则** CDL **应该使能** HAL **对** SMP **的支持。一般可以在内核中使用** requires **语句进行使能。如:**

```
cdl_component CYGPKG_HAL_SMP_SUPPORT {
display          "SMP support"
default_value 0
requires {CYGHWR_HAL_I386_FPU_SWITCH_LAZY == 0}
cdl_option CYGPKG_HAL_SMP_CPU_MAX {
    display          "Max number of CPUs supported"
    flavor           data
    default_value 2
    }
}
```

i386 **的硬件抽象层对** FPU **的支持是可选的,默认情况下支持** FPU**。也可以禁止** FPU**,这样可以提高系统的性能。对** FPU **的支持有两个选项:每一次上下文切换都保存和恢复** FPU **状态、只在需要的时候才转换** FPU **的状态。相应的** CDL **描述如下:**

```
cdl _ component CYGHWR _ HAL _ I386 _ FPU {
display          "Enable I386 FPU support"
default _ value 1
description      "This component enables support for the
                 I386 floating point unit."
cdl _option CYGHWR _ HAL _ I386 _ FPU _ SWITCH _ LAZY {
    display          "Use lazy FPU state switching"
    flavor           bool
    default _ value 1
    description "
                 This option enables lazy FPU state switching.
                   The default behaviour for eCos is to save and
                   restore FPU state on every thread switch, interrupt
                 and exception. While simple and deterministic, this
                 approach can be expensive if the FPU is not used by
                 all threads. The alternative, enabled by this option,
                 is to use hardware features that allow the FPU state
                 of a thread to be left in the FPU after it has been
                 descheduled, and to allow the state to be switched to
                 a new thread only if it actually uses the FPU. Where
                 only one or two threads use the FPU this can avoid a
                 lot of unnecessary state switching."
    }
}
```

此外,i386 **的硬件抽象层还支持不同型号的** CPU**,特别是** Pentium **类型的** CPU **还具有更多的功能模块,**GDB **有时可能希望得到更多的寄存器内容。因此必须有一些选项来支持这些特性。一般可以在变体或平台的** CDL **包或者在**. ecm **文件中使用** requires **语句来使能这些特性。例如:**

```
cdl _ component CYGHWR _ HAL _ I386 _ PENTIUM {
display          "Enable Pentium class CPU features"
default _ value 0
description      "This component enables support for various
                 features of Pentium class CPUs."
cdl _ option CYGHWR _ HAL _ I386 _ PENTIUM _ SSE {
    display          "Save/Restore SSE registers on context switch"
    flavor           bool
    default _ value 0
    description "
                     This option enables SSE state switching. The default
                       behaviour for eCos is to ignore the SSE registers.
                       Enabling this option adds SSE state information to
```

```
                every thread context."
    }
    cdl_option CYGHWR_HAL_I386_PENTIUM_GDB_REGS {
        display "Support extra Pentium registers in GDB stub"
        flavor  bool
        default_value 0
        description "
                This option enables support for extra Pentium registers
                  in the GDB stub.  These are registers such as CR0-CR4, and
                  all MSRs.  Not all GDBs support these registers, so the
                  default behaviour for eCos is to not include them in the
                  GDB stub support code."
        }
    }
```

在 i386 的硬件抽象层中,链接脚本由体系结构抽象层提供。在其他体系结构如 MIPS 的硬件抽象层中是由变体抽象层提供链接脚本。下面的选项为配置系统中其他组件提供了链接脚本文件的名字(i386.ld):

```
cdl_option CYGBLD_LINKER_SCRIPT {
        display "Linker script"
        flavor data
    no_define
        calculated {"src/i386.ld"}
}
```

最后,还需要提供一个接口表明该平台是否实现了 hal_i386_mem_real_region_top()函数。如果实现了该函数,则应该包含这样一行:implements CYGINT_HAL_I386_MEM_REAL_REGION_TOP,像 RedBoot 这样的包可以据此来检测是否存在该函数,从而对它进行调用。如:

```
cdl_interface CYGINT_HAL_I386_MEM_REAL_REGION_TOP {
        display "Implementations of hal_i386_mem_real_region_top()"
    }
}
```

至此,形成了一个完整的基于 i386 结构的体系结构抽象层的 CDL 文件。其他体系结构的 CDL 文件可以按照这种方法来产生。

第 12 章　组件结构与 CDL

eCos 最为显著的特征是其可配置性。针对具体开发项目的不同,可以对系统的组件进行适当的选择,实现系统在源码级的配置与裁剪,充分满足嵌入式系统的在系统资源方面和实时方面的需求。eCos 的这种特性主要得力于它的组织结构和配置方式。它使用组件框架对系统中的所有组件进行管理,为了实现这种管理,它采用了组件定义语言 CDL 对所有的软件包和组件进行描述。这一章首先对 eCos 的配置机制和组件结构进行介绍,然后详细介绍组件定义语言 CDL。在开发基于 eCos 的新的软件包和组件时,需要使用 CDL 语言。

12.1　eCos 的配置机制

eCos 组件框架着重强调的是组件的可配置性,其主要目的是可以利用可重用的软件组件构造大部分的嵌入式应用程序,而这些可重用的软件组件又具有很高的可配置性。在开发嵌入式应用软件的过程中,通常会受到几个方面的限制。首先是来自资源方面的限制。为节省成本,许多嵌入式应用可供其使用的内存都很小,最后形成的可存放于 ROM、EPROM 或 Flash ROM 的映像文件除了包含应用所需程序代码外,不应该包含其他多余的代码。在像 Windows 这样的桌面系统的应用程序中,通常对这些多余代码不会加以特别注意,但在嵌入式应用中这种问题必须加以考虑。组件框架必须具有对组件进行配置的能力,剔除多余的不需要的功能。另一种限制来自于嵌入式应用的实时性,这种实时需要往往会增加程序的代码量和程序的执行时间。因此组件框架应该对组件的时间特性进行控制。还有一种限制来自嵌入式应用程序的调试。一般来说,嵌入式应用程序的调试较为困难,特别是与时间相关的一些信息难以重现和跟踪。在组件框架中可以使用一些可重用的用于调试目的的组件,结合源码级调试工具 (如 GDB),可以有效地对嵌入式应用进行调试。一旦调试工作完成,再使用组件框架将这些用于调试目的的组件删除。

可配置性的目的是控制组件的行为。以调度器组件为例,它可以选择是否支持时间片,也可以选择是否支持多优先级,它还可以选择是否对传递给调度器函数的参数的合法性进行检查。以桌面应用程序的图形界面为例,对这种行为的选择包括它的某个按钮是采用文本方式还是图形方式、采用文本方式时的字符大小、其前景和背景采用什么颜色等等。对组件的所有这些行为都需要进行一系列的选择,组件框架为此提供了一种选择和配置机制。对这些行为进行控制的方法之一是在应用程序运行时进行控制,这种方法必须实现各种可能的选择,即使实际应用不需要的行为也可能包括在应用程序中,因此其缺点是代码量大。另一种控制方法是在链接时进行控制,其典型的例子是面向对象语言中的继承机制。以上面提到的文本按钮和图形按钮为例,按钮库提供一个抽象基础类 Button 和派生类 TextButton 与 PictureButton,如果应用程序只使用文本按钮则只需生成 TextButton 类的对象,而不会用到 PictureButton 类的代码。这种方法在大多数情况下都很有效,能减少最后的映像文件的大小。但这种方法也有其缺点,如果派生类过多就难于管理。

相对于这两种方法而言，eCos 组件框架在更早的阶段就对组件的行为进行控制。它在对源码进行编译和建库的时候就着手对组件行为进行控制。以上面提到的按钮为例，按钮组件提供了一些选项，比如一个只需要文本按钮支持的选项。在对组件进行编译并生成应用所需的库文件时，库文件中只包含了应用程序所需要的代码。具有不同需求的不同应用程序必须具有不同版本的库文件，这些库文件通过各自不同的配置所产生。

从理论上来说，这种编译时的可配置性可以在减少程序代码方面取得最好的效果。它可以在单个语句级别上进行控制，而不是在函数级和对象级上进行控制。可以用一个与嵌入式系统密切相关的支持多线程的包作为例子来加以说明。这种包中有一个标准函数用于异步终止线程（在 POSIX 中该函数是 pthread _ cancel，在 μITRON 中是 ter _ tsk）。这些函数本身具有相当可观的代码量，但问题并不在此，而是在于系统的其他部分为了保证线程终止函数的正确运行需要另外的程序代码和数据。例如，当一个线程在等待一个互斥体而处于阻塞状态时，如果另一个线程要终止该线程则可能必须完成两种操作：从等待该互斥体的线程队列中删除该线程、解除其优先级继承所造成的影响。为实现这些操作，在线程数据结构中必须增加一部分数据用来表示线程当前状态，同时在互斥体程序中必须增加相应的程序代码来对数据结构中的这一部分数据进行操作。然而，许多嵌入式应用程序并不需要这种异步线程终止的能力，这种线程终止程序不应该包含在最终的应用映像文件内。但是，如果不在编译时对其进行配置和控制，在最终的应用映像中仍然会包含互斥体程序中的这些无用的程序代码和数据。eCos 具有编译时的可配置能力，可以很好地解决这一问题。例如，在其互斥体程序中包含了一些用于线程终止的语句，这些语句只有在需要该功能时才被编译。这种配置的结果是最后的应用映像文件只包含了实际应用所需要的代码和数据，而不会包含其他多余的程序代码和数据。

eCos 的这种编译时的配置能力具有一定的复杂性。它不能完全取代其他配置方法，它与运行时的选择控制能力和链接时的选择控制能力相互补充。编译时的配置能力是这三种配置方法中最有效的一种配置方法，最适合于嵌入式系统的开发。

eCos 的组件支持从简单到复杂的各种不同程度的配置能力，不同的组件可能具有不同的可配置程度。最简单的配置仅仅是选择是否加载某一特定的软件包，而该软件包可能没有任何可配置的选项。具有高可配置性的组件可能会包含几个甚至几十个配置选项，可以提供非常精细的配置。

12.2 eCos 组织结构及编译过程

为满足系统的可配置性和可裁剪性，eCos 对一些相对独立的软件以包的形式进行封装。eCos 的软件包必须符合组件框架的一些规则才可以使用。软件包要以统一的格式发布，组件仓库管理工具使用这种统一的格式来对软件包进行管理。同时，软件包要为组件框架提供一个对该包进行描述的顶层 CDL 脚本。为了保证软件包在不同环境下仍然能够使用，对该包进行编译的方法也必须满足一些条件。

12.2.1 软件包与组件仓库

eCos 具有一个组件仓库，这是一个包含所有已经安装的包的目录结构。组件框架提供了

一个管理工具，可以用其安装新的软件包或者安装已有软件包的新版本，也可以删除软件包。组件框架包含了一个由管理工具进行维护的简单的数据库，该数据库包含了各种软件包的详细信息。图 12-1 为 eCos 组件仓库结构示意图。

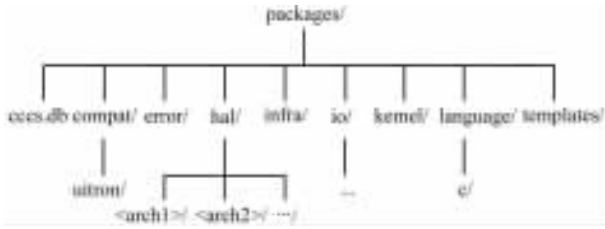


图 12-1　软件包的目录结构

每个包在组件仓库中都有自己的目录结构，不允许多个包同时出现在同一个目录下。error 包、infra 包和 kernel 包都位于组件仓库的最顶层。其他一些软件包也有自己固定的目录：compat 目录用于实现兼容目的的软件包如 μITRON、POSIX 等，hal 目录包含的软件包用于将 eCos 移植到不同的系统结构和平台，io 目录包含了设备驱动程序软件包，language 目录用于语言支持库（如 C 语言库）。在安装一个新的软件包时，对于将该包放置到哪个目录并没有严格的限制。如果需要的话，可以创建一个新的目录来安装新的软件包。

ecos.db 文件保存的是组件仓库数据库，由管理工具进行管理。eCos 配置工具在启动时通过读取该文件来获取已安装的各种软件包的信息。在开发一个新的软件包时，需要在该文件中加入一些相关的信息，也就是必须更新数据库文件 ecos.db。templates 目录保存了各种配置模板。

在每一个软件包的目录下可以有一个或多个版本的以版本号命名的子目录。这样允许用户安装同一个软件包的多个版本，用户可以选择其中的一个版本来开发应用程序。管理工具可以删除不需要的版本。

12.2.2　软件包的内容与格式

软件包通常包含如下内容：

① 用于生成库文件的源程序文件，应用程序与这些库文件链接生成可执行文件。有些源文件可能用于其他目的，如提供一个链接脚本文件。

② 对软件包提供的接口进行定义的导出头文件。

③ 在线说明。

④ 测试源程序，用于检查软件包是否能在指定硬件和配置下正常工作。

⑤ 用于描述该软件包的一个或多个 CDL 脚本文件

软件包通常还包含一个记录该包变化历史的文件 ChangLog。一个软件包不一定包含上述全部内容，但必须至少具有一个描述该包的 CDL 脚本文件，否则组件框架将无法对其进行处理。有些软件包可能没有任何源代码，例如一个只对其他包实现的公共接口进行定义的软件包。一些包可能只有源代码而没有任何导出头文件，如以太网设备驱动程序，它只实现一个标准接口而不提供任何附加功能。另外，软件包不一定都有在线说明。

组件框架建议软件包采用图 12-2 所示的目录结构，对软件包的内容可以按功能的不同进

行划分。

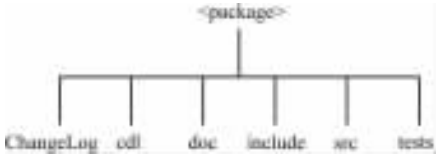


图 12-2　eCos 组件仓库示意图

12.2.3　编译过程

在对 eCos 进行编译之前需要对其进行配置。配置工作可由 eCos 图形配置工具或命令行配置工具进行。eCos 的编译涉及到三个目录树:组件仓库 (component repository)、编译树 (build tree)、安装树 (install tree)。

① 组件仓库。组件仓库包含了所有软件包的源代码、CDL 脚本文件以及说明文档等等。从编译的角度来看,这是一个源目录树,可以看成是一个只读资源。开发人员只能在安装或删除软件包的时候通过管理工具对组件仓库进行修改。

② 编译树。每一种配置都具有自己的编译树,在任何时间都可以通过使用 eCos 配置保存文件如 ecos.ecc 来生成该目录树。在完成对 eCos 的配置后,可以将这种配置保存在 ecos.ecc 配置文件中 (文件名不一定是 ecos.ecc,但扩展名必须是 .ecc)。编译树只包含中间文件,主要是编译时产生的目标文件。一旦完成全部的编译工作,编译树所包含的信息就不再有用,可以删除。但如果对配置进行了修改,则需要再一次使用编译树。

③ 安装树。该目录树的内容在编译过程中产生,它包含了与应用开发相关的所有文件。该目录树有一个 lib 子目录,包含了库文件 libtarget.a、链接脚本、启动代码等等。它还有一个 include 子目录,包含了各种软件包的导出头文件。另外还有一个 include/pkgconf 子目录包含了各种配置头文件,这些配置头文件主要由选项的 # define 语句组成。在成功完成编译后,开发应用程序时只使用安装树,不需要使用组件仓库和编译树中的任何文件。安装树中不会有任何软件包的说明文档,这些说明文档只保存在组件仓库内。

编译过程的一般步骤如下:

1) 对于一个给定的配置,组件框架负责生成编译树和安装树的所有目录。如果这些目录树已经存在,组件框架将对其进行清扫工作。例如,如果一个软件包被删除,则组件框架将删除其在编译树和安装树中的所有相关文件。配置头文件将在此时生成。组件框架根据具体的主机开发环境将生成 makefile 文件或其他用于编译各种软件包的类似文件。为保证所有的选项能发挥作用,对配置进行的每一次修改都必须重复这一步工作。

2) 在实际编译过程中,首先要确认安装树包含了所有的导出头文件。安装树的 include 子目录将是编译过程中查找头文件的目录之一。

3) 对与当前配置相关的所有源代码文件进行编译。编译时使用一组与目标平台相关的编译标志,每个软件包可以对这些编译标志进行修改,用户也可以修改这些编译标志。编译生成的所有目标文件将保存在编译树内。

4) 一旦全部源文件编译完成,所有的目标文件将生成一个库文件,该库文件通常是 libtarget.a,应用程序可以与该库文件进行链接产生执行文件。所生成的库文件位于安装树内。

5）组件框架使用 make _ object 和 make 属性可以对编译步骤进行定制。

12.3 组件定义语言 CDL

组件定义语言 CDL (Component Definition Language) 是 eCos 组件框架的一个关键部分。eCos 中所有的包都必需具有至少一个 CDL 脚本对该包进行描述。这种 CDL 脚本包含了该包中所有配置选项的详细信息，并且提供了一些如何对该包进行编译的信息。当实现一个新的组件或者将其他软件包转换为一个 eCos 组件时，必须编写相应的 CDL 脚本。

下面是一个简单的 CDL 脚本：

```
cdl _ package CYGPKG _ ERROR {
    display       "Common error code support"
                  compile strerror.cxx
    include _ dir cyg/error
    description "
                  This package contains the common list of error and
                  status codes. It is held centrally to allow
                  packages to interchange error codes and status
                  codes in a common way, rather than each package
                  having its own conventions for error/status
                  reporting. The error codes are modelled on the
                  POSIX style naming e.g. EINVAL etc. This package
                  also provides the standard strerror() function to
                  convert error codes to textual representation."
}
```

这是一个简单的错误代码包，它没有任何子组件和配置选项。包内有一个内部名字 CYGPKG _ ERROR，其他包的 CDL 脚本可以使用“require CYGPKG _ ERROR”来引用它。在一个配置头文件中还可以对该符号使用 # define。除了包的名字外，CDL 脚本还包含了该组件包的许多属性。属性 display 对该包进行了简短的描述，而属性 description 则对该包进行详细描述，用户可以从该属性了解到更多有关该组件包的信息。compile 和 include _ dir 属性列举了该组件包在编译时的一些信息。该包没有提供它的在线说明信息。

有些包的 CDL 脚本甚至比这个例子更为简单。如果包仅仅只提供一个接口而没有任何需要编译的文件，则它不需要 compile 属性。如果没有任何头文件或者头文件位于 install/include 的顶级目录，则不需要 include _ dir 属性。

然而，大多数的包都要比错误包复杂得多，它们包含了各种各样的子组件和配置选项。下面例子出自于 infrastructure 包：

```
cdl _ component CYGDBG _ INFRA _ DEBUG _ TRACE _ ASSERT _ BUFFER {
    display         "Buffered tracing"
    default _ value 1
    active _ if     CYGDBG _ USE _ TRACING
```

```
        description    "
            An output module which buffers output from tracing and
            assertion events. The stored messages are output when an
            assert fires, or CYG_TRACE_PRINT() (defined in
            <cyg/infra/cyg_trac.h>) is called. Of course, there will
            only be stored messages if tracing per se (CYGDBG_USE_TRACING)
            is enabled above."

        cdl_option CYGDBG_INFRA_DEBUG_TRACE_BUFFER_SIZE {
            display          "Trace buffer size"
            flavor           data
            default_value 32
            legal_values   5 to 65535
            description    "
                The size of the trace buffer. This counts the number of
                trace records stored. When the buffer fills it either
                wraps, stops recording, or generates output."
        }
        ...
    }
```

与 cdl_package **一样,**cdl_component **也有名字和体,体包含了该组件各种各样的属性,它还可以包含子组件或选项。同样,**cdl_option **也有名字和属性体。上面这个例子还列出了一些新的属性:**default_value、active_if、flavor **以及** legal_values**。**

12.3.1 CDL 命令

在顶层 CDL **脚本中可能包含有四个** CDL **命令,它们是** cdl_package,cdl_component,cdl_option **和** cdl_interface**。这四个命令是** CDL **语言的基本构造单位,它们具有相同的基本格式:**

(1) cdl_option **命令。**

```
cdl_option <name> {
    ...
}
```

cdl_option **命令定义单个配置选项(**option**)。选项是可配置的一个基本单位。每一个选项一般都对应于用户的一个选择。**

(2) cdl_component **命令。**

```
cdl_component <name> {
    ...
}
```

cdl_component **命令定义一个组件(**component**),是一组配置选项的集合。组件是一个包**

含其他选项和子组件的一个配置选项。

(3) cdl _ package 命令。

```
cdl _ package <name> {
    ...
}
```

cdl _ package 命令定义一个包 (package),一个可以发布的组件。包是一个发布单位,它也是一个配置选项,用户可以选择是否将特定的包加入到配置中,并且可以选择加载哪个版本的包。包同时也是一个组件,它以层次结构的形式包含其他的组件和选项。

(4) cdl _ interface 命令。

```
cdl _ interface <name> {
    ...
}
```

cdl _ interface 命令定义一个接口 (interface)。接口是一种特殊类型的配置选项。

紧跟在这些 CDL 命令后面的 name 是被定义的选项、组件、包或接口的名字,随后是 { } 内的命令属性体。包和组件还可以包含其他项,因此 cdl _ package 和 cdl _ component 在其体内还可以具有嵌套命令。在一个给定的配置中,名字必须是惟一的。例如,如果 C 库包和 TCP/IP 协议栈包都具有一个相同名字的选项,那么这两个包不能同时出现在同一个配置中。使用命名约定可以避免名字冲突。如果两个包不可能同时出现在同一个配置中时,则可以使用相同的名字。在硬件抽象层 HAL 包内会出现这种情况。

每一个包都有一个顶层 CDL 脚本,在包的数据库入口 ecos.db 中对该脚本有详细说明。通常顶层 CDL 脚本名字与相应的包有关,例如内核包使用的 CDL 脚本的名字为 kernel.cdl。顶层脚本中第一条命令是 cdl _ package,它所使用的名字 name 与 ecos.db 中所使用的名字相同。每一个包只能有一个 cdl _ package 命令。

各种各样的 CDL 实体均具有层次结构。以内核包为例,它包含一个调度组件、一个同步原语组件以及其他一些组件,同步组件又包含各种各样的选项 (例如,是否使能互斥体优先级继承的选项)。对于组件嵌套的层数没有上限和下限的规定,但一般不会超过四层。

采用这种层次结构有两个目的。使用层次结构可以对选项进行统一控制,禁止某个组件将自动禁止该组件下的所有选项。另一个目的是在图形配置工具中可以更加简单明了地表示配置结构,便于浏览和修改配置选项。

默认情况下,包都位于层次结构的顶层。但有时可能使用 parent 属性来指定它在层次结构中的位置。例如,体系结构抽象层 HAL 包 (如 CYGPKG _ HAL _ SH) 通常使用 parent 属性来将其置于 CYGPKG _ HAL 之下,而平台抽象层 HAL 包又通过 parent 属性将其置于这种体系结构抽象层 HAL 之下。这种方法可以使得这种层次结构更具有可读性。组件、选项和接口都可以使用 parent 属性,但不常用。

在顶层脚本中定义的所有组件、选项和接口都直接放置在包 (cdl _ component) 之下。它们也可以嵌套在 cdl _ package 命令体中。下面两个脚本具有相同的效果:

脚本 1:

```
cdl _ package CYGPKG _ LIBC {
```

```
    ...
}
cdl_component CYGPKG_LIBC_STRING {
    ...
}
cdl_option CYGPKG_LIBC_CTYPE_INLINES {
    ...
}
```

脚本 2：

```
cdl_package CYGPKG_LIBC {
    ...
    cdl_component CYGPKG_LIBC_STRING {
        ...
    }
    cdl_option CYGPKG_LIBC_CTYPE_INLINES {
        ...
    }
}
```

如果一个脚本既有位于 cdl_package 命令体外的选项定义，也有位于其体内的选项定义，那么命令体内的选项将最先得到处理。

组件还可以包含选项和其他一些 CDL 实体，这实际上也是它与选项之间的区别。这种定义可以在 cdl_component 命令体内进行，例如：

```
cdl_component CYGPKG_LIBC_STDIO {
    cdl_component CYGPKG_LIBC_STDIO_FLOATING_POINT {
        ...
    }
    cdl_option CYGSEM_LIBC_STDIO_THREAD_SAFE_STREAMS {
        ...
    }
}
```

这种组件命令体内的嵌套选项只在配置选项数量有限的简单包中较为有效，当选项数量增加时就难以取得令人满意的效果。解决的方法是将 CDL 内容以组件为基本单位分成多个 CDL 脚本。此时将用到 script 属性。例如，在 C 库内所有与 stdio 相关的配置选项都位于 stdio.cdl，顶层 CDL 脚本 libc.cdl 将包含下面语句：

```
cdl_package CYGPKG_LIBC {
    ...
    cdl_component CYGPKG_LIBC_STDIO {
        ...
        script stdio.cdl
```

```
    }
}
```

组件 CYGPKG _ LIBC _ STDIO _ FLOATING _ POINT 和选项 CYGSEM _ LIBC _ STDIO _ THREAD _ SAFE _ STREAMS 被放置到 stdio.cdl 的顶层。cdl _ component 命令体内可能有一些嵌套选项，另外还可以通过 script 属性的形式来使用位于其他文件内的选项。在这种情况下，首先将处理嵌套选项，然后再读取其他脚本中的选项。由 script 属性指定的脚本只能用于对新的选项、组件或接口进行定义，它不应该包含任何当前组件的附加属性。

12.3.2 CDL 属性

包、组件、选项以及接口都有一个属性体，属性体用于说明如何对它们的每一个选项进行处理。例如，属性 description 用于描述文本消息，它告诉用户该选项在目标应用中的使用效果。而 default 属性用于说明选项的默认值是什么，如指定某个特定选项的默认值是使能还是禁止。所有的属性都是可选的，配置选项的属性体可以是一个空体。

不同的属性具有不同的目的，因此它们的语法并不完全一致。某些属性可能没有任何值，一些属性可能有单一的值（如描述字符串），而另一些属性则可能有一个参数列表，例如用于指定哪些文件应该编译的属性 compile。

大多数的属性可以被用在任何一个 cdl _ package、cdl _ component、cdl _ option 和 cdl _ interface 命令中。但有些属性具有专用性，例如 script 属性，它只与组件有关。define _ header、hardware、include _ dir、include _ files 和 library 等属性只适用于包，因此这些属性只能出现在 cdl _ package 命令中。calculated、default _ value 和 flavor 等属性与包以及接口无关，legal _ value 属性与包无关。

下面根据属性所服务的不同目的对它们进行分组介绍。

1. 信息类属性

如果选项具有足够的说明信息，用户就可以很方便地进行配置操作。有三个属性用于对选项提供纯文本方式的解释：

① display 属性。为选项提供一个文本方式的别名。它提供一个比 CYGPKG _ LIBC _ TIME _ ZONES 这样的选项名字更易于理解的名字。

② description 属性。为选项提供一段很详细的文字说明。

③ doc 属性。指定与该选项相关的在线说明文档位置。

例如：

```
cdl _ package CYGPKG _ UITRON {
    display         "uITRON compatibility layer"
    doc               ref/ecos-ref.a.html
    description   "
        eCos supports a uITRON Compatibility Layer, providing
        full Level S (Standard) compliance with Version 3.02 of
        the uITRON Standard, plus many Level E (Extended) features.
        uITRON is the premier Japanese embedded RTOS standard."
    ...
```

```
}
```

在图形配置工具中,用户主要是通过 display 属性所提供的字符串来辨别配置选项的,例如上例中的"uITRON compatibility layer"。当选中配置选项时,图形配置工具将会显示 description 属性所提供的详细说明文字。如果想要了解更多的信息,则可以通过图形配置工具访问 doc 属性提供的在线说明文档。

这三个属性都只有一个参数。display 和 description 属性的参数只是一个字符串,doc 属性的参数则是一个 HTML 文件。配置工具将从该包的 doc 子目录中查找该文件,如果没有则从该包的顶级目录查找。

2. 配置层次属性

有两个与组件和选项的层次结构相关的属性:parent 属性和 script 属性。

parent 属性用于控制选项在配置层次结构中的位置,使用该属性可以将 CDL 实体置于层次结构中的任何位置。该属性在包中被经常使用,用以避免所有的包都出现在配置层次结构的顶层。例如,体系结构 HAL 包 CYGPKG _ HAL _ SH 使用 parent 属性将其置于公共 HAL 包 CYGPKG _ HAL 之下:

```
cdl _ package CYGPKG _ HAL _ SH {
    display         "SH architecture"
    parent          CYGPKG _ HAL
    ...
}
```

parent 属性还可以用在 cdl _ component、cdl _ option 和 cdl _ interface 命令体内,但这种情况出现较少。

script 属性用于从另一个 CDL 脚本导入附加的配置信息。该属性仅在 cdl _ component 命令中使用。它使用另一个 CDL 脚本的文件名作为参数,该 CDL 脚本文件包含一些置于当前组件之下的附加的选项、子组件和接口。配置工具将从该包的 cdl 子目录中查找该文件,如果没有则从该包的顶级目录查找。script 属性使用例子如下:

```
cdl _ component CYGPKG _ LIBC _ STDIO {
    display         "Standard input/output functions"
    flavor          bool
    requires        CYGPKG _ IO
    requires        CYGPKG _ IO _ SERIAL _ HALDIAG
    default _ value 1
    description     "
        This enables support for standard I/O functions from <stdio.h>."

    script          stdio.cdl
}
```

3. 值相关属性

有七个属性与选项的值或状态相关。它们分别是:flavor、calculated、default _ value、legal _ values、active _ if、implements、requires。

在可配置的环境中，选项值的作用非同寻常。一个具体的配置可能使用也可能不使用某个选项。有可能只选择数学库而不选择内核，而数学库的 CDL 脚本仍然要涉及到内核选项，例如，CYGSEM _ LIBM _ THREAD _ SAFE _ COMPAT _ MODE 对 CYGVAR _ KERNEL _ THREADS _ DATA 有一个强制需要。即使选项被使用，它也不一定是活跃的，这依赖于层次结构中它的更高层的行为。例如，如果 C 库的 CYGPKG _ LIBC _ STDIO 组件被禁止，那么其他的一些选项如 CYGNUM _ LIBC _ STDIO _ BUFSIZE 就变成不相干的选项。

每一个选项都有一个用于使能/禁止的布尔值和一个数据值。许多选项只关心它的布尔值，而有些选项则只关心它的数据值。使用 flavor 属性可以对选项的这种性质进行控制：

① flavor none。指明选项的布尔值和数据值都是不可修改的。选项总是处于使能状态，而且其数据值总是为 1。它常常用于仅仅把组件当成层次结构中的一个占位符的情形。

② flavor bool。只有选项的布尔值可以被修改，数据值固定为 1。

③ flavor data。只有选项的数据值可以被修改，布尔值固定为使能。

④ flavor booldata。选项的布尔值和数据值都可以被修改。

flavor 属性不能用于包和接口。包总是具有 booldata 值的 flavor 属性，接口总是具有 data 值的 flavor 属性。因为大多数的配置选择是一种简单的“是”与“否”的选择，因此选项和组件的默认 flavor 属性值是 bool。

calculated 属性用于那些用户不能修改、由目标硬件平台或其他选项的当前值所决定的选项。calculated 属性不能用于包和接口。一般来说，应该避免出现具有这种属性的选项，它容易使人混淆。该属性使用普通的 CDL 表达式作为其参数。例如：

```
# A constant on some target hardware, perhaps user-modifiable on other
# targets.
cdl _ option CYGNUM _ HAL _ RTC _ PERIOD {
    display       "Real-time clock period"
    flavor        data
    calculated    12500
}
```

default _ value 属性用 CDL 表达式为选项提供一个默认值。它与 calculated 属性相似，但只指定一个可以修改的默认值。该属性与包和接口无关。下面是它的一个例子：

```
cdl _ option CYGDBG _ HAL _ DEBUG _ GDB _ THREAD _ SUPPORT {
    display       "Include GDB multi-threading debug support"
    requires      CYGDBG _ KERNEL _ DEBUG _ GDB _ THREAD _ SUPPORT
    default _ value CYGDBG _ KERNEL _ DEBUG _ GDB _ THREAD _ SUPPORT
    ...
}
```

legal _ values 属性对选项的数据值作出一个限制。它只适用于 flavor 属性为 data 或 booldata 的选项。legal _ values 属性不能用于包。例子如下：

```
cdl _ option CYGNUM _ LIBC _ TIME _ STD _ DEFAULT _ OFFSET {
    display       "Default Standard Time offset"
```

```
    flavor          data
    legal_values    -- -90000 to 90000
    default_value -- 0
    ...
}
```

active_if 属性对选项或其他 CDL 实体的活跃状态进行控制。配置选项或其他 CDL 实体的状态可能是活跃的也可能是不活跃的,这种活跃状态通常受到该选项在层次结构中所处位置的控制。它与选项的状态有关,而与选项的值无关。例如,如果 CYGPKG_LIBC_STDIO 组件被禁止,那么位于它下面的所有选项都将处于不活跃状态,而且对结果不会有任何影响。在某些情况下,这种层次结构对选项的活跃状态难以提供足够的控制。以数学库为例,数学库可以提供浮点例外支持,但它还需要有硬件的支持。相应的数学库配置选项在整个层次结构中的位置处于 CYGPKG_LIBM 包之下,除非有适当的硬件支持,否则这些选项将是不活跃的。在这种情况下,需要使用 active_if 属性。active_if 属性还可以用来避免配置层次结构中出现过多的嵌套。例子如下:

```
# Do not provide extra semaphore debugging if there are no semaphores
cdl_option CYGDBG_KERNEL_INSTRUMENT_BINSEM {
    active_if CYGPKG_KERNEL_SYNCH
    ...
}
```

implements 属性与 CDL 接口相关。该属性使得一个选项提供一种可通用的接口。如果一个选项处于活跃、使能状态并且实现了一个特定的接口,那么它的接口值为 1。implements 属性只有一个参数,这个参数就是接口的名字。下面是其使用例子:

```
cdl_package CYGPKG_NET_EDB7XXX_ETH_DRIVERS {
    display         "Cirrus Logic ethernet driver"
    implements      CYGHWR_NET_DRIVERS
    implements      CYGHWR_NET_DRIVER_ETH0
    ...
}
```

requires 属性对用户的选择进行约束。如果要使能或激活一个选项,则必须满足它的 requires 属性所列出的所有约束条件。例如,如果希望 C 库的某个函数的实现具有线程安全性(thread-safe),就必须使用内核并且使能内核。例子如下:

```
cdl_option CYGSEM_LIBC_PER_THREAD_ERRNO {
    display         "Per-thread errno"
    doc             ref/ecos-ref.15.html
    requires        CYGVAR_KERNEL_THREADS_DATA
    default_value 1
    ...
}
```

4. 配置头文件相关属性

在生成或者更新一个编译树的时候，组件框架对于每一个包都将产生一个配置头文件。默认情况下，它将对每一个处于活跃和使能状态的选项、组件或接口都产生一条 # define 语句。对于 flavor 属性为 data 或 booldata 值的选项，它的 # define 语句将使用选项的数据值，其他 flavor 属性值将使用常数 1。这种 # define 语句举例如下：

```
# define CYGFUN _ LIBC _ TIME _ POSIX 1
# define CYGNUM _ LIBC _ TIME _ DST _ DEFAULT _ STATE -1
```

有六个属性用于控制这种头文件的产生过程，它们分别是：define _ header、no _ define、define _ format、define、if _ define 和 define _ proc。

define _ header 属性用于指定配置文件名。组件框架为每个包产生的配置头文件的默认文件名是以该包的名字为基础，去掉前缀（第一个下划线前面的字符，包括下划线），剩余部分变为小写，再加上“.h”作为后缀。例如 CYGPKG _ KERNEL 包的配置头文件是 pkgconf/kernel.h。如果不想使用这种默认方式的文件名，可以使用 define _ header 属性来指定配置头文件名。下面的例子指定头文件名为 hal _ sparclite.h：

```
cdl _ package CYGPKG _ HAL _ SPARCLITE {
    display "SPARClite architecture"
    parent          CYGPKG _ HAL
    hardware
    define _ header hal _ sparclite.h
    ...
}
```

no _ define 属性用于禁止默认 # define 语句的产生。如果某个选项所产生的 # define 没有受到任何程序源码的检查，就可以使用该属性来取消其 # define 语句的产生。它还可以与 define、if _ define 或 define _ proc 等属性一起使用。该属性没有参数。例如：

```
cdl _ component CYG _ HAL _ STARTUP {
    display         "Startup type"
    flavor          data
    legal _ values  {"RAM" "ROM"}
    default _ value {"RAM"}
    no _ define
    define -file system.h CYG _ HAL _ STARTUP
    ...
}
```

除了产生默认 # define 语句之外，还可以使用 define 属性来产生另外一条 # define 语句。上面例子中就使用了 define 属性。它只有一个参数，该参数就是所要定义的符号名。它通过选项-file 和-format 来控制哪一个配置头文件将产生该 # define 语句以及该语句的格式。上面例子中将在头文件 system.h 内产生一条 # define CYG _ HAL _ STARTUP RAM 的语句（假设配置时选择 RAM 启动方式）。

define _ format 属性用来控制默认 # define 语句中值的格式。下面例子中的格式串“0x%04x”用来产生一个 4 位 16 进制数。

```
cdl _ option CYGNUM _ UITRON _ VER _ ID        {
    display          "OS identification"
    flavor           data
    legal _ values   0 to 0xFFFF
    default _ value 0
    define _ format "0x%04x"
    description     "
        This value is returned in the 'id'
        field of the T _ VER structure in
        response to a get _ ver() system call."
}
```

if _ define 属性用于在配置头文件中产生 # ifdef 以及 # endif 等预处理语句。其格式如下：

```
if _ define [-file=<filename>] <symbol1> <symbol2>
```

该属性有两个参数。如果当前选项是活跃的并且处于使能状态，则配置头文件中将产生如下格式的 C 预处理结构：

```
#ifdef <symbol1>
#define <symbol2>
#endif
```

如果当前选项不活跃或者被禁止，则不会产生这样这些语句。-file 用于指定这些语句输出到哪一个配置头文件，默认值是当前包的配置头文件。if _ define 属性的使用例子如下：

```
cdl _ option CYGDBG _ KERNEL _ USE _ ASSERTS {
    display "Assertions in the kernel package"
    ...
    if _ define CYGSRC _ KERNEL CYGDBG _ USE _ ASSERTS
    requires   CYGDBG _ INFRA _ ASSERTION _ SUPPORT
}
```

define _ proc 属性使用一段 TCL 代码向配置头文件输出一段代码数据。例如：

```
cdl _ package CYGPKG _ HAL _ ARM _ PID {
    display          "ARM PID evaluation board"
    parent           CYGPKG _ HAL _ ARM
    define _ header hal _ arm _ pid.h
    include _ dir    cyg/hal
    hardware
    define _ proc {
        puts $ ::cdl _ system _ header "#define CYGBLD _ HAL _ TARGET _ H    <pkgconf/hal _
```

```
arm.h>"
        puts $ ::cdl_system_header "#define CYGBLD_HAL_PLATFORM_H <pkgconf/hal
_arm_pid.h>"
        puts $ ::cdl_header ""
        puts $ ::cdl_header "#define HAL_PLATFORM_CPU      \"ARM 7TDMI\""
        puts $ ::cdl_header "#define HAL_PLATFORM_BOARD    \"PID\""
        puts $ ::cdl_header "#define HAL_PLATFORM_EXTRA    \"\""
        puts $ ::cdl_header ""
    }
    ...
}
```

5. **编译控制类属性**

有六个属性用于控制编译过程，它们分别是：compile、make、make_object、library、include_dir **和** include_files**。后面三个属性只适用于包，并且只出现在** cdl_package **命令体内。**

大多数的源码文件通过选用目标平台相应的编译器以及一些编译标志和设置就可以进行编译，编译的最后结果将得到库文件 libtarget.a**，应用程序将与该库进行链接。** compile **属性用于列举这些编译源文件。例如：**

```
cdl_package CYGPKG_ERROR {
    display         "Common error code support"
    compile         strerror.cxx
    include_dir     cyg/error
    ...
}
```

compile **属性的参数可以是一个或多个文件。包通常具有多个源文件，因此在** cdl_package **命令体中可能有多个** compile **属性。有些源程序可能是专用于某些特殊的配置选项的，只有当这些选项被使能时才会对它们进行编译。这种情况下，应该在相应的** cdl_option、cdl_component **或者** cdl_interface **命令体内用** compile **属性来列举这些源文件。**

有些包可能有比较复杂的编译要求。eCos **编译的最终结果是产生一个单一的库** libtarget.a**，但有时还需要产生其他一些目标。例如，体系结构** HAL **包通常要产生一个链接器脚本和一些** start-up **代码。这些目标的产生可以通过** make **属性来实现。有时，需要采取特殊的编译步骤来产生目标文件，这时就需要使用** make_object **属性。例如：**

```
cdl_package CYGPKG_HAL_MN10300_AM33 {
    display         "MN10300 AM33 variant"
    ...
    make {
        <PREFIX>/lib/target.ld:<PACKAGE>/src/mn10300_am33.ld
        $(CC) -E -P -Wp,-MD,target.tmp -DEXTRAS=1 -xc $(INCLUDE_PATH) \
            $(CFLAGS) -o $@ $<
        @echo $@": \" > $(notdir $@).deps
        @tail +2 target.tmp >> $(notdir $@).deps
```

```
        @echo >> $(notdir $@).deps
        @rm target.tmp
    }
}
```

默认的最终编译结果是库文件 libtarget.a，使用 library 属性可以对其进行改变。一般应该避免使用这种属性，因为它将增加应用开发的复杂性和难度。下面例子将生成库文件 libSomePackage.a：

```
cdl_package <SOME_PACKAGE> {
    ...
    library    libSomePackage.a
}
```

include_dir 属性和 include_files 属性与包的导出头文件有关。大多数的包都将导出一个或多个对它们的公共接口进行定义的头文件。例如，C 库将导出头文件 stdio.h 和 ctype.h。默认情况下，包的头文件将导出到 install/include 目录。如果将所有的导出头文件都放置到同一个目录，则有可能造成文件名的冲突。解决这一问题的方法是使用 include_dir 属性为其指定一个子目录。下面例子中的头文件将导出到子目录 install/cyg/infra 中。

```
cdl_package CYGPKG_INFRA {
    display          "Infrastructure"
    include_dir      cyg/infra
    ...
}
```

include_files 属性用于指定包的导出头文件。

6. 其他属性

hardware 属性只与包相关。该属性用于说明该包专用于指定的硬件。某些包（如设备驱动程序和硬件抽象层 HAL 包）是硬件专用的，如果在目标系统中没有相应的硬件，则这些包不会有任何意义。通常在配置工具中选择目标平台时会自动进行硬件包的选择。hardware 属性不带参数，它只是标识这是一个硬件专用的包。例子如下：

```
cdl_package CYGPKG_HAL_MIPS {
    display "MIPS architecture"
    parent              CYGPKG_HAL
    hardware
    include_dir     cyg/hal
    define_header hal_mips.h
    ...
}
```

12.4 选项命名约定

在一个具体的配置中，所有的选项都处于同一个名字空间，任何两个不同的选项不能有相

同的名字。为了避免同名现象的出现,选项的名字必须符合命名约定。

对于每一个处于活跃和使能状态的选项,组件框架通常要为其输出一条 # define 语句,# define 语句要用到选项的名字,这些名字是已经定义好的符号。这就要求所有的名字都是有效的 C 语言预处理符号。即使选项具有 no _ define 属性(不输出 # define 语句),其名字也必须符合这种约定。预处理符号可以是大写英文字母 AZ、小写英文字母 az、下划线"_"和数字 0-9 的任意组合,但第一个字符不能是数字。同时,第一个字符最好不要使用下划线,以避免与保留标识符相冲突。另外的一个约定是预处理符号只使用大写英文字母。

一个具有代表性的选项名字是 CYGSEM _ KERNEL _ SCHED _ BITMAP,它由几个部分组成:

1)前缀 CYG。前面的几个字符(此例中的前三个字符 CYG)用于标识该包出自哪一个社团组织。由于历史原因,源自 Red Hat 公司的包使用的前缀为 CYG 而不是 RHAT。用户在写自己的包时,可以使用自己定义的前缀。

2)前缀后面的三个字符 SEM 表示该选项的性质,例如它是否对接口有影响,或者仅仅是一种实现(implementation)。后面将介绍一些这样的通用标记。

3)KERNEL _ SCHED 部分表示选项在整个层次结构中的位置。此例中说明该选项是内核包中调度组件的一部分。在选项名字中体现层次结构的位置有助于了解可配置代码,并减少名字冲突的可能。

4)最后部分 BITMAP 是选项自己的一个标识。

前缀后面三个字符的标记可以提供有关选项的一些信息。目前已经定义了许多的这种标记。有些标记的选用并不是绝对的。例如,与硬件相关的选项一般用 HWR,而数字选项则用 NUM,与平台相关的数字选项(如中断堆栈大小)则可以使用这两个标记的任何一个。下面是目前 eCos 所使用的一些通用标记(前缀 xxx 是提供该包的社团组织的标识符):

① xxxARC _。ARC 标记表示选项与处理器体系结构相关,这种选项通常只出现在体系结构抽象层 HAL 包或变体抽象层 HAL 包中。

② xxxHWR _。HWR 标记表示选项与特定的目标平台相关。通常只出现在平台抽象层 HAL 包中。

③ xxxPKG _。PKG 标记表示这是一个包或者是一个组件,是一个用于扩展配置层次结构的选项。

④ xxxGLO _。GLO 标记表示这是一个全局配置选项。

⑤ xxxDBG _。DBG 标记表示选项与 debug 相关。

⑥ xxxTST _。TST 标记表示这是一个与测试相关的选项。它们通常不会影响实际应用代码。

⑦ xxxFUN _。FUN 标记表示这是一个影响包的接口的选项。有许多用于表示与接口相关的标记,xxxFUN _主要用于那些控制是否提供一个或多个函数的包的选项。如果没有其他可用的与接口相关的标记,那么可以使用 FUN 标记。

⑧ xxxVAR _。VAR 标记类似于 FUN 标记,但主要用于那些控制是否出现一个或多个变量或对象的选项。

⑨ xxxCLS _。CLS 标记只用于那些提供面向对象接口的包,控制一个类的出现与否。

⑩ xxxMFN _。MFN 只用于面向对象的接口,表示一个成员函数(而不是一个类)的出现

与否。

⑪ xxxSEM _。SEM 选项不会影响接口。它用于那些对包的语义行为有根本影响的选项。例如,内核调度器的选择实质上是一种语义行为,它对接口不会有影响,特别是函数 cyg _ thread _ create 的存在不受调度器选择的影响。它主要对系统的行为有影响。

⑫ xxxIMP _。IMP 标记主要用于执行 (implementation) 选项,不会对接口与语义行为造成影响。一个典型的执行选项是控制一个或一组函数是否应该内联 (inline)。

⑬ xxxNUM _。NUM 标记用于数字选项。例如调度优先级数目。

⑭ xxxDAT _。DAT 标记用于非数字的数据选项,如设备名字。

⑮ xxxBLD _。BLD 标记表示对编译过程有影响的选项,如编译器标志的设置。

⑯ xxxINT _。INT 标记通常用于 CDL 接口。

⑰ xxxPRI _。PRI 标记一般不用于配置选项。它通常被 define _ proc 属性内的 CDL 脚本所使用,该 CDL 脚本将一些附加信息通过配置头文件传递给源码程序。

⑱ xxxSRC _。SRC 标记一般不用于配置选项。它通常被包用来使其源码与这些选项相互作用,尤其是在使用 if _ define 属性的环境下。

12.5 Tcl 简介

所有的 CDL 脚本都是一种 Tcl 脚本,它由一个标准的 Tcl 解释器进行处理。Tcl (Tool Command Language) 是一种非常简单的编程语言。CDL 对 Tcl 只进行了很小的扩展,如 cdl _ component 和 cdl _ option 等。在写 CDL 脚本时,并不需要详细了解 Tcl,只需参考已有的脚本,对其进行复制并适当加以变化就可以编写新的 CDL 脚本。虽然如此,由于 CDL 语言在某些地方 (如 define _ proc 属性中) 需要用到 Tcl 脚本,同时在分析和理解 eCos 各组件包时也需要有一定的 Tcl 知识,因此有必要对 Tcl 有一个初步的了解。本节将结合 CDL 对 Tcl 进行简单的介绍。

12.5.1 基本语法

Tcl 脚本由一组命令组成,这些命令通过换行符或分号进行分离。Tcl 命令由命令字和参数组成,命令字和参数之间用空格分开。所有的命令都采用下例所示的基本格式:

```
expr 20 + 10
```

该命令计算 20 与 10 的和,并返回结果 30。其命令字为 expr,参数为 20 和 10。Tcl 中不同的命令对参数的要求和处理都有所不同。所有的 Tcl 命令都返回一个结果,如果没有实际结果,则返回一个空串。

可以从下面的例子来进一步了解 Tcl 的基本语法:

```
puts Hello
set x 32
```

这是一个由两条命令组成的 Tcl 脚本,它也可以写成一行 (用分号隔开):

```
puts Hello ;set x 32
```

其中第一条命令的命令字是 puts,它只有一个参数:Hello。第二条命令的命令字是 set,它

有两个参数：x 和 32。当使用分号将命令进行分离时，分号前后的空格将被忽略。

12.5.2 变量

Tcl 的变量可以被赋值，在后续命令中可以使用该变量值。命令 set 用于设置和读取变量值。例如：

```
set  x  32
set  x
```

第一条命令给变量 x 赋值 32，第二条命令读取变量 x 的值。

Tcl 不需要对变量进行声明，在对变量进行第一次设置时将自动产生该变量。Tcl 变量没有类型，可以赋以任何类型的值。

在命令中可以通过变量的引用来使用变量值，如：

```
expr  $ x * 2
```

当命令中出现字符 $ 时，Tcl 将该字符后面的字符串 (字符和数字) 当作变量名处理，并将该变量名用变量的值来替代。在此例中，expr 命令的实际参数是 32 * 2 (前面例子中变量 x 已被赋值 32，此处将 $ x 替换为 32)。

可以在任何命令的任何位置使用这种变量，如：

```
set cmd expr
set x 11
 $ cmd  $ x *  $ x
```

除了在使用 Tcl 脚本的地方外，CDL 很少使用这种变量替换。如果在选项描述中使用了实际的字符 $ (不是 Tcl 变量)，则应该在其前面加上反斜杠符号，即“ \ $ ”。

12.5.3 命令替换

所谓命令替换，就是将一条命令的结果作为另一条命令的一个参数。例如：

```
set a 44
set b [expr  $ a * 4]
```

当一条命令中出现方括号 [] 时，Tcl 将方括号内的所有内容当作一条嵌套命令进行处理。Tcl 首先执行这一嵌套命令，并将该命令执行结果替换方括号及其内容。上例中第二条命令的第二个参数将是 176 (expr $ a * 4 的结果是 176)。

在 CDL 语言中，只有在像 define _ proc 这样的属性使用 Tcl 脚本的地方才有可能使用这种命令替换，其他地方很少使用。值得注意的是要避免出现意外的命令替换。例如，在一个选项的描述中如果包含有方括号“ [] ”，则应该使用反斜杠符号，即“ \ ['和“ \ ] ”。

12.5.4 引号和花括弧的使用

首先来看一个例子：

```
set x Hello world
```

这是一个非法的 Tcl 命令。set 命令只有变量名和值两个参数，而此例中有三个参数：x、Hello 和 world。可以使用引号来避免这种现象的出现：

```
set x "Hello world"
```

引号的出现使 Hello world 成为一个单独的参数。当 Tcl 解释器遇到引号时，它将引号内的所有内容当成当前参数的一部分，但不包括引号本身。由于解释器将剔除引号，因此上例中 set 命令的第二个参数是没有引号的 Hello world 字符串。引号的使用在 CDL 脚本中很值得注意，例如：

```
cdl_option CYG_HAL_STARTUP {
    ...
    default_value "RAM"
}
```

Tcl 解释器对上例进行处理时将会把引号剔除掉，CDL 表达式解析器看到的将是 RAM 而不是"RAM"。它被当作一个未知选项 RAM 的引用，而不是一个字符串常量。CDL 表达式解析器将其当作未加载的选项进行处理，RAM 的值为 0。因此，选项 CYG_HAL_STARTUP 最后的默认值（default_value）将是 0。使用花括弧或反斜杠可以避免这种现象的发生，在上例中可以使用花括弧：default_value {"RAM"}。

引号内的换行符和分号不会终止当前命令，引号内可以有多行内容。CDL 命令的 description 属性通常使用引号的这种特性，例如：

```
cdl_package CYGPKG_ERROR {
    description "
        This package contains the common list of error and
        status codes. It is held centrally to allow
        packages to interchange error codes and status
        codes in a common way, rather than each package
        having its own conventions for error/status
        reporting. The error codes are modelled on the
        POSIX style naming e.g. EINVAL etc. This package
        also provides the standard strerror() function to
        convert error codes to textual representation."
    ...
}
```

花括弧的作用与引号类似，它将花括弧内的全部内容当成单个参数处理。与引号不同之处是在花括弧内不存在变量替换、命令替换以及反斜杠替换（惟一例外是位于行末的反斜杠替换）。对于前面例子中的 default_value {"RAM"}，Tcl 解释器将剔除花括弧，CDL 表达式解析器看到的是字符串常量"RAM"。

12.5.5 反斜杠和注释

前面已经提到了反斜杠的一些用法。Tcl 支持其他许多编程语言广泛使用的反斜杠替换

方法。例如 \ n 代表换行、\ \ 表示一个“ \ ”字符等等。在命令行最末端的反斜杠将使该反斜杠字符、换行字符以及下一行起始端的任何空格被一个空格所替代。下面两条 Tcl 命令是等同的：

```
puts  "Hello \ nworld \ n"

puts \
"Hello
world
"
```

下面三条语句具有相同的效果：

```
default _ value {"RAM"}
default _ value \ "RAM \ "
default _ value " \ "RAM \ ""
```

Tcl 脚本中使用字符“ # ”来进行注释。注释用的 # 字符只能出现在命令字的位置，如果没有注释的话，字符 # 的位置出现的应该是命令字。看下面的例子：

```
# This is a comment
puts "Hello" # world
```

上例中的第一行是有效的注释，注释字符 # 出现在命令字的位置。但第二行中的字符 # 被认为是一个参数，在 puts 命令中这是一个非法参数，因此该命令将会导致错误的发生。如果第二行改成下面的形式，则成为合法的 Tcl 语句：

```
puts "Hello"; # world
```

其中的分号表明当前命令的结束，字符 # 后面的“world”是其注释内容

12.6 表达式和值

某些配置选项 (如 CYGVAR _ KERNEL _ THREADS _ DATA) 可以用具体的值对它们进行配置。这些值可以影响到使用这些选项的表达式 (例如 requires CYGVAR _ KERNEL _ THREADS _ DATA)。同时，它们还将影响这些选项的最终结果，选项的值不仅影响编译过程，而且还影响到其他任何受该选项 (如 CYGVAR _ KERNEL _ THREADS _ DATA) 的约束条件所限制的选项的行为。

12.6.1 选项的值

选项的值包含有四个要素，它们表示的意义分别如下：

① 选项可以被加载，也可以不被加载。

② 如果选项被加载，它可以处于活跃状态，也可以不处于活跃状态。

③ 即使选项处于活跃状态，它可能被使能，也可以被禁止。

④ 如果选项被加载，处于活跃状态而且被使能，则它具有一些选项值数据。

在任何时候,一个具体的配置不可能包含所有的包,有些包不能同时出现在同一个配置中。以体系结构抽象层 HAL 包为例,它包含有字节排列方式选项、基本数据类型大小等一些与体系结构相关的选项。在同一个配置中不能同时加载两个这样的体系结构抽象层 HAL 包,否则将引起名字冲突,从而导致配置的失败。对于任何一个没有出现在当前配置中的选项的处理如下:

① 未被加载的选项在被引用时,其值为 0。

② 未加载的选项不对编译过程产生任何直接的影响。它不会在配置头文件中产生相应的 # define 语句,也不会有相应的文件被编译。如果选项没有被加载,组件框架就无法了解其 compile 等类似的属性。通过表达式的引用,可以使未加载的选项间接影响编译过程。

③ 未加载的选项对配置不会产生任何约束条件。组件框架无法了解未加载选项的 requires 和 legal _ values 等属性。

配置选项都是以组件和子组件的层次结构来进行组织的。如果禁止 (disable) 一个组件,那么位于该组件下的所有选项都将处于不活跃状态 (inactive)。在图形配置工具中,不活跃的选项呈灰色状态。选项的活跃状态 (active 和 inactive) 对其他的包具有一定的影响。例如,假如某个包使用了 sprintf 函数,它要求有浮点转换的支持,如果相关的选项处于不活跃状态,那么这种约束条件就得不到满足。选项的不活跃状态准确定义如下:

① 如果一个选项的父选项 (parent) 处于不活跃或禁止状态,则该选项也将处于不活跃状态。

② 选项的 active _ if 属性可以使选项处于不活跃状态。

③ 如果选项处于不活跃状态,那么在引用该选项的 CDL 表达式中该选项将被赋值为 0。

④ 处于不活跃状态的选项对编译过程不起作用。不会产生相应的 # define 语句,其 compile 等类似属性将被忽略。

⑤ 处于不活跃状态的选项对配置不会产生任何约束条件。

在 CDL 中,选项的值由两部分组成。一部分是布尔 (Boolean) 部分,用于控制选项的使能状态;另一部分是数据部分,为选项提供附加信息。选项值的两个部分由其 flavor 属性来控制,如表 12-1。

表 12-1 Flavor 对选项值的控制

Flavor	Enabled	Data
none	总是使能	1,不可修改
bool	可修改	1,不可修改
data	总是使能	可修改
booldata	可修改	可修改

选项值的布尔部分和数据部分的作用如下:

① 如果选项被禁止 (disable),也就是说选项的布尔值为 false,则它在 CDL 表达式中的值为 0。其作用与不活跃的选项相同,选项值的数据部分没有意义。在具有 none 和 data 的 flavors 属性的情况下,选项总是处于使能状态,不适用于该规则。

② 如果选项处于使能 (enable) 状态,则在引用它的 CDL 表达式中将使用该选项的数据值作为该选项的值。对于具有 none 和 bool 的 flavors 属性的选项,其数据值固定为 1。

③ 如果一个组件或包被禁止,则在其下的所有子组件和选项都是不活跃的。

④ 如果选项被禁止,则它不会对整个配置施加任何约束条件,其 requires 和 legal _ value 属性将被忽略。如果选项被使能,则它的约束条件应该被满足,否则将出现各种各样的冲突。

⑤ 如果选项被禁止,则它不会对编译过程有直接的影响。不会产生相应的 # define 语句,也不会有相应的文件被编译。如果选项被使能,则将直接影响编译过程。其选项名和数据值用于在相应的配置头文件中产生 # define 语句。

12.6.2 普通表达式

CDL 表达式具有一些语法约定。表达式的使用例子如下:

```
default _ value CYGGLO _ CODESIZE > CYGGLO _ SPEED
    default _ value { (CYG _ HAL _ STARTUP == "RAM" &&
                       ! CYGDBG _ HAL _ DEBUG _ GDB _ INCLUDE _ STUBS &&
                       ! CYGINT _ HAL _ USE _ ROM _ MONITOR _ UNSUPPORTED &&
                       ! CYGSEM _ HAL _ POWERPC _ COPY _ VECTORS) ?1 :0 }
    default _ value {"\"/dev/ser0\""}
```

CDL 表达式由四个元素组成:配置选项的引用、字符串常量、整数、浮点数。这四个元素通过各种操作符组合在一起。这些操作符有:

- 一元操作符:-,!
- 算术操作符:+,-,*,/,%
- 移位操作符:<<,>>
- 比较操作符:==,!=,<,<=,>,>=
- 位操作符:&,^,|
- 逻辑操作符:&&,||
- 条件运算符:A?B:C

CDL 表达式允许出现包含在括弧内的子表达式。表 12-2 是各种操作符列表。

表 12-2 表达式操作符

运算优先级	操 作 符	类 别
13	引用和常量	基本元素
12		位非
12	!	逻辑非
12	-	算术负
11	* / %	算术乘、除、取余
10	+ -	算术加、减
9	<< >>	移位
8	<= < > >=	不等式
7	== !=	比较
6	&	位与
5	^	位异或
4	|	位或
3	&&	逻辑与
2	||	逻辑或
1	?:	条件运算

表达式中的 CDL 标识符 (如上例中的 CYGGLO _ SPEED) 是对配置选项的引用,使用了该选项的名字。当前配置中不一定加载了该选项。当组件框架对表达式赋值时,它将根据该选项是否被加载以及该选项的活跃状态和使能状态等情况来确定该选项的值。

字符串常量是用引号描述的一串字符。在使用字符串常量的时候必须注意字符串常量的使用方法,防止 Tcl 解释器在 CDL 表达式解析器看到该字符串之前丢掉字符串的引号。看下面的例子:

```
define _ value "RAM"
```

上面例子在 CDL 表达式解析器看到字符串之前,其引号已被丢掉,因此该表达式将被解释为一个配置选项 RAM 的引用,其值为 0。前面已经介绍了解决这种问题的方法,使用花括弧或其他 Tcl 引号机制可以避免这种现象的出现。

字符串常量由引号内的数据组成,如果数据本身需要包含引号字符,则需要再一次使用引号,例如:

```
default _ value {"\"/dev/ser0\""}
```

整数常量由一组数字组成,可以在其前面加上正负符号 (+、-)。十六进制整数用 0x 或 0X 前缀表示,最左边数字为 0 时表示八进制。整型数据长度为 64 位,如果常量太大,可用双精度表示。双精度数字可以使用传统的语法格式,如 3.141592 或-3E6。

在对表达式赋值时,可以对操作数进行适当的转换。例如,比较运算符“>”可以用于整数和双精度数,它首先尝试将它的两个操作数进行从字符串到整数的转换,如果失败再进行从字符串到双精度的转换。如果两种转换都不成功,则报告赋值例外冲突。

12.6.3 目标表达式

目标表达式用于描述无冲突配置所需要满足的一个目标。如果一个 requires 约束条件没有得到满足,则组件框架推理机将检查目标表达式,以确定是否可以采取一定的方法对配置加以适当的改变而不会引起新的冲突,从而使目标表达式的值为 true,使冲突得以解决。

某些属性特别是 requires 和 active _ if 属性的参数构成一种目标表达式。与普通表达式一样,所有参数都组合在一起然后交给表达式解析器进行处理。在处理字符串常量时,必须注意对引号的使用。因此,全局表达式通常使用花括弧将整个表达式括起来,表达式解析器将其当作一个单独的参数处理。

目标表达式主要由一些普通表达式组成,下面是目标表达式的一个例子:

```
requires {CYGDBG _ HAL _ DEBUG _ GDB _ INCLUDE _ STUBS
          ! CYGDBG _ HAL _ DEBUG _ GDB _ BREAK _ SUPPORT
          ! CYGDBG _ HAL _ DEBUG _ GDB _ CTRLC _ SUPPORT}
```

上面例子由三个单独的表达式组成,它们将被赋值为非 0 值。该表达式还可以写成如下的形式:

```
requires {CYGDBG _ HAL _ DEBUG _ GDB _ INCLUDE _ STUBS  &&
          ! CYGDBG _ HAL _ DEBUG _ GDB _ BREAK _ SUPPORT &&
          ! CYGDBG _ HAL _ DEBUG _ GDB _ CTRLC _ SUPPORT}
```

使用下列语句也具有相同的效果：

```
requires CYGDBG_HAL_DEBUG_GDB_INCLUDE_STUBS
requires !CYGDBG_HAL_DEBUG_GDB_BREAK_SUPPORT
requires !CYGDBG_HAL_DEBUG_GDB_CTRLC_SUPPORT
```

目标表达式的值是一个布尔值。

12.6.4 列表表达式

列表表达式是一组 legal_values 属性的值，legal_values 属性的参数构成列表表达式。与普通表达式和目标表达式一样，所有的参数组合在一起然后再交给表达式解析器进行处理。在采用字符串常量时必须注意引号的使用。列表表达式通常将整个表达式用花括弧括起来，表达式解析器将其视为一个单独的参数。

大多数列表表达式都采取下面两种格式之一：

```
legal_values <expr1> <expr2> <expr3> ...
legal_values <expr1> to <expr2>
```

其中 expr1、expr2 等等都是普通表达式，它们通常是常量，或者是对体系结构抽象层 HAL 的推算选项的引用。如果需要，它们也可以是任意表达式。上述第一种格式指明一组可能的值，不一定是数字值。第二种格式指明一个数字范围，其上限和下限必须是数字值。下面是列表表达式的一些例子：

```
legal_values {"red" "green" "blue"}
legal_values 1 2 4 8 16
legal_values 1 to CYGARC_MAXINT
legal_values 1.0 to 2.0
```

这两种格式也可能组合在一起，如：

```
legal_values 1 2 4 to CYGARC_MAXINT -1024 -20.0 to -10
```

它表示三个合法值 1、2、－1024，一个从 4 到 CYGARC_MAXINT 的整数范围，一个从 20.0 到－10.0 的浮点数范围。这种列表表达式很少出现。

在图形配置工具中使用这样的 legal_values 列表表达式使得用户对选项值的操作非常方便。列表表达式在图形配置工具中将以小窗口的形式出现，例如 {"red" "green" "blue"} 表达式将在图形配置工具中出现一个下拉菜单供选择，而对于 {1 to 16} 表达式则将出现一个选项值的调节控制项。

12.7 接口

虽然选项为系统的可配置性提供了很大程度上的支持，但有时这种支持还不够，需要为可配置性提供更高层次上的支持。例如，有些包需要标准内核调度器接口实现的支持，这是一种约束条件，这种约束条件可以通过多级队列调度器、位图调度器或其他一些调度器来满足。这种依赖性可以使用下面语句来表示：

```
requires CYGSEM _ KERNEL _ SCHED _ MLQUEUE || CYGSEM _ KERNEL _ SCHED _ BITMAP
```

CDL 接口提供了这样一种抽象机制:约束条件可以用一个抽象概念来表达(如"scheduler"),而不是用特殊实现(如 CYGSEM _ KERNEL _ SCHED _ MLQUEUE 和 CYGSEM _ KERNEL _ SCHED _ BITMAP)来表示。从本质上来说,接口是一种可推算的配置选项。下面就是一个接口例子:

```
cdl _ interface CYGINT _ KERNEL _ SCHEDULER {
    display   "Number of schedulers in this configuration"
    ...
}
```

可以使用单独的调度器来实现这一接口:

```
cdl _ option CYGSEM _ KERNEL _ SCHED _ MLQUEUE {
    display          "Multi-level queue scheduler"
    default _ value 1
    implements       CYGINT _ KERNEL _ SCHEDULER
    ...
}

cdl _ option CYGSEM _ KERNEL _ SCHED _ BITMAP {
    display          "Bitmap scheduler"
    default _ value 0
    implements       CYGINT _ KERNEL _ SCHEDULER
    ...
}
```

接口的值是实现该接口的活跃而且使能的选项的数目。这样,需要调度器实现支持的包可以按下面的方式使用约束条件:

```
requires CYGINT _ KERNEL _ SCHEDULER
```

如果所有的调度器都没有使能,或者没有加载内核包,那么 CYGINT _ KERNEL _ SCHEDULER 将被赋值为 0。如果至少有一个调度器处于活跃和使能状态,则该约束条件将会得到满足。

由于接口具有一个可推算出来的值,因此在 cdl _ interface 命令体中不会出现也不应该出现属性 defaul _ vale 和 calculated。接口总是具有 flavor data 属性,因此也不会出现 flavor 属性。一般情况下,接口可以使用其他的一些属性,如 requires 和 compile 等。

如果有一个选项实现了一个 CDL 接口,那么该选项通常会提供一些特殊的 C 和 C++函数接口。例如,以太网设备驱动程序实现了 CDL 接口 HWR _ NET _ DRIVERS,它提供了一组可用于 TCP/IP 协议栈的 C 函数。同样,CYGSEM _ KERNEL _ SCHED _ MLQUEUE 实现了 CDL 接口 CYGINT _ KERNEL _ SCHEDULER,提供了一些可用的调度函数。

12.8 更新 ecos.db 数据库

eCos 要求所有的包都处于一个组件仓库内，组件数据库 ecos.db 列举了所有的组件。在进行 eCos 开发时，除使用管理工具增加或删除组件外，其他时候可以认为组件仓库是一种只读资源。但如果是开发一个新的组件，则需要对组件仓库进行操作。这种操作要对数据库进行更新，将新开发的包更新到数据库。

与大多数与组件框架相关的文件一样，ecos.db 数据库实际上是一个 Tcl 脚本。典型的包的入口如下：

```
package CYGPKG_LIBC {
    alias          {"C library" libc clib clibrary}
    directory      language/c/libc
    script                 libc.cdl
      description  "
This package enables compatibility with the ISO C standard - ISO/IEC
9899:1990. This allows the user application to use well known standard
C library functions, and in eCos starts a thread to invoke the user
function main()"
}

package CYGPKG_IO_PCI    {
    alias          {"PCI configuration library" io_pci}
    directory      io/pci
    script                 io_pci.cdl
    hardware
      description "
          This package contains the PCI configuration library."
}
```

这种 package 命令有两个参数，即包的名字 name 和体 body。name 必须与包的顶层 CDL 脚本中 cdl_package 命令的 name 相同。body 可以包含五种命令：alias、directory、script、hardware 和 description。

① alias。每一个包必须有一个或多个别名。当列举包的清单时，一般使用第一个别名。很显然，使用别名“C library”比使用“CYGPKG_LIBC”更具有可读性，更容易理解。其他别名一般不用于输出操作，但是可以作为输入。例如，使用命令行工具 ecosconfig 时可以用“add libc”作为一个命令选项，其操作结果与“add CYGPKG_LIBC”相同。这里 libc 是 CYGPK_GLIBC 的另一个别名。

② directory。该命令用于指定包的相对位置（相对于组件仓库的根目录）。需要说明的是，该包的位置不能因其后续版本的原因而被改变。如果必须在其他位置安装该包的新版本，则首先必须将其旧版本全部卸载。数据库不能为同一个包提供两个不同的位置。

③ script。该命令指定包的顶层 CDL 脚本（包含有该包的 cdl_package 定义）的位置。它

的位置一般是在 cdl 子目录。该文件不应该被其后续版本改变。

④ hardware。该命令用于说明该包与特殊硬件紧密相关，如设备驱动程序和 HAL 包。CDL 脚本中的 cdl _ package 命令和数据库入口中都应该对此加以说明。

⑤ description。该命令对包进行简短的描述。可以使用 cdl _ package 命令 description 属性所提供的文本信息。

ecos.db 数据库中还具有各种目标平台的一些信息。在将 eCos 移植到新的目标平台时，需要将该目标平台的信息加入到数据库中。另外，还必须将该平台的硬件抽象层 HAL 包和其他平台相关的包的详细信息加入到数据库中。

第 13 章　eCos 嵌入式 Web 服务器

作为一种移植性强、可配置性好的免费嵌入式实时操作系统，eCos 已在许多领域得到了成功应用。目前使用 eCos 作为其嵌入式软件平台的产品包括：移动电话、个人数字助理(PDA)、MP3 播放器、手持多媒体设备、激光打印机、磁盘阵列、卫星地面设备、指纹识别设备、汽车数字设备等等。eCos 是一种开放源代码，随着它的不断发展和完善，其应用范围也将越来越广泛。作为 eCos 应用的一个例子，本章将介绍一个运行于普通 PC 微机上的基于 eCos 的嵌入式 Web 服务器。通过对该应用实例的介绍，读者将会对 eCos 应用的开发过程有一个更清楚的了解。

13.1　嵌入式 Web 服务器 LibHTTPD

随着信息技术的发展，基于互联网的 Web 服务已日益普及，对某些设备进行 Web 访问也成为日常生活中的一个普通需求。虽然有许多可用的 Web 服务器实现方法，但大多数标准的 Web 服务器因过于庞大而不便于使用。将 Web 服务器嵌入到一个独立设备或应用中时，该 Web 服务器的实现应该尽量简单而且便宜。Hughes Technologies 公司提供的 LibHTTPD 正是这样的一个 Web 服务器软件。

LibHTTPD 是一个自由软件，可以从 http://www.hughes.com.au 处下载最新版本。使用 LibHTTPD 可以提供安全而有效的 Web 访问，其访问内容既可以是静态的，也可以是动态产生的。LibHTTPD 提供了许多 API 函数，使用这些函数可以产生一个 Web 服务器并对其进行操作。其操作基本过程如下：

1) 创建一个 Web 服务器。

2) 定义 Web 服务器访问内容。

3) 接收并解释从浏览器发送过来的 HTTP 请求。

4) 对请求进行处理并通过 HTTP 将结果返回给浏览器。

LibHTTPD 提供了用于完成上述工作的 API 函数。在创建 Web 服务器时，可以调用 httpCreate() 函数，其返回值为服务器的句柄。其他对该 Web 服务器进行操作的 API 函数都将使用此句柄。如果需要的话，可以使用 httpSetErrorLog() 函数和 httpSetAccessLog() 函数对 Web 服务器的错误和访问信息进行记录。

在创建一个 Web 服务器后，必须对服务器的访问内容进行设置和定义。LibHTTPD 与其他 Web 服务器软件不同，其 Web 内容必须使用其 API 函数产生。使用 API 函数可以产生如下形式的 Web 内容：

① 静态内容：保存在静态文本 buffer 内的 HTML 内容。

② 文件内容：一个外部文件。

③ 通配符(Wildcard)内容：在指定目录内的任何相关文件。

④ 动态内容：使用 C 回调函数产生所需的输出。

在对 Web 服务器内容进行定义后，可以使程序进入一个循环，在循环体内对 Web 请求进行处理。每一次循环都调用下述 API 函数：

httpdGetConnection () ——接受 HTTP 连接
httpdReadRequest () ——读 HTTP 请求
httpdProcessRequest () ——生成输出信息
httpdEndRequest () ——结束此次 HTTP 连接

当调用 httpdProcessRequest () 函数时，LibHTTPD 将对客户端的请求内容进行识别，确定 Web 服务器内是否有所需的内容，并采取适当的措施将被请求的内容发送到客户端。

表 13-1 列举了 LibHTTPD 提供的一些 API 函数，这些 API 函数的详细说明和使用方法可以参考 LibHTTPD 源码中的说明文档。

表 13-1 LibHTTPD API 函数

类 别	函 数	说 明
服务器设置函数	httpdCreate ()	创建 Web 服务器
	httpdSetAccessLog ()	设置访问日志文件
	httpdSetErrorLog ()	设置错误日志文件
	httpdSetFileBase ()	设置文件基本路径名
Web 内容定义函数	httpdAddCContent ()	调用 C 函数产生输出内容
	httpdAddFileContent ()	将一个外部文件加入到输出内容
	httpdStaticContent ()	将一个内部文本 buffer 加入到 HTML 输出内容
	httpdAddWildcardContent ()	增加与通配符匹配的文件内容
	httpdAddCWildcardContent ()	请求指定目录中的任何文件时调用 C 回调函数
连接和请求处理函数	httpdGetConnection ()	接受一个 HTTP 连接请求
	httpdReadRequest ()	读取并保存从客户端发送过来的请求和数据
	httpdProcessRequest ()	对请求进行处理，并将请求内容发送到客户端
	httpdEndRequest ()	请求结束处理
响应处理函数	httpdOutput ()	将文本 buffer 内容发送到客户端浏览器
	httpdPrintf ()	按指定格式将内容输出到客户端浏览器
	httpdSetContentType ()	设置除 HTML 文本以外的内容类型，如“image／jpeg”
	httpdSetResponse ()	设置返回给客户端浏览器的响应代码
	httpdAddHeader ()	增加 HTML 头内容
	httpdSendHeaders ()	发送 HTML 头
表格数据、Cookies、符号表函数	httpdGetVariableByName ()	在符号表中查找变量
	httpdGetVariableByPrefix ()	获取第一个与指定前缀相匹配的变量
	httpdGetNextVariableByPrefix ()	获取下一个与指定前缀相匹配的变量
	httpdGetVariableByPrefixedName ()	在符号表中查找变量
	httpdAddVariable ()	在符号表中增加变量
	httpdDumpVariables ()	Dump 符号表内容
	httpdSetCookie ()	设置 Cookies
身份验证	httpdAuthenticate ()	使用用户名和口令进行身份认证
	httpdForceAuthenticate ()	强迫身份认证
访问控制表 (ACL) 函数	httpdAddAcl ()	在 ACL 表中增加访问控制项
	httpdSetDefaultAcl ()	设置默认 ACL
	httpdCheckAcl ()	进行 ACL 检查

(续)

类　别	函　数	说　明
其他函数	httpdUrlEncode()	进行 URL 编码
	httpdRequestMethod()	获取访问方式(HTTP_GET/HTTP_POST)
	httpdRequestMethodName()	获取访问方式名字(GET/POST)
	httpdRequestPath()	获取 URL 请求路径
	httpdRequestContentType()	获取当前请求内容类型
	httpdRequestContentLength()	获取当前请求发送的内容长度

作为 eCos 的一个应用,可以使用 LibHTTPD 建立一个简单的基于 eCos 的小型嵌入式 Web 服务器。该服务器的建立分两步进行。首先要对 eCos 进行配置和编译,产生应用程序(即 Web 服务器)可以与之链接的 eCos 库。在得到 eCos 链接库后,再对应用程序(包括 LibHTTPD 库和 Web 服务器程序)进行编译和链接,最后得到可在目标平台上运行的 Web 服务器软件。

为方便学习和实践,运行嵌入式 Web 服务器的硬件平台选用普通的 PC 微机(可以不需要键盘、鼠标和显示器),要求使用 Intel i82559 网卡(如 Intel Entherpress Pro 10/100 网卡),eCos 开发平台 HOST 与目标系统的连接使用串口连接。目标平台使用软盘引导 RedBoot。这里假设已经产生了一个用于软盘引导的 RedBoot,且能正常引导。

13.2 配置和建造 eCos

使用图形配置工具对 eCos 进行配置。由于 Web 服务器需要网络支持,因此在选用模板时应该选择"net"模板,目标平台为"i386 PC"。目前 eCos 源码只为 i386 PC 目标平台提供 Intel i82559 系列网卡的支持,因此目标平台上只能使用这种网卡。如果要支持其他网卡,必须为这些网卡编写新的驱动程序。

在对 eCos 进行配置时,要注意选择启动方式(Startup)为"RAM",这是因为最后生成的 Web 服务器可执行代码将通过 GDB 加载到目标系统内存中运行。

Web 服务器需要对网络进行配置。配置过程如图 13-1 所示。主要配置内容包括:

① IP 地址。

IP address :172.26.4.100

② 子网掩码。

Network mask address :255.255.255.0

③ 广播地址。

Broadcast address :172.26.4.255

④ 网关。

Gateway/router IP address :172.26.4.1

最后产生的 eCos Web 服务器使用的 IP 地址为 172.26.4.100,网关为 172.26.4.1,子网掩码为 255.255.255.0,广播地址为 172.26.4.255。

除此之外,其他配置选项都可以使用默认配置。也可以根据需要适当增加某些软件包或删减一些软件包,或修改某些配置选项的值。

图 13-1　对 eCos 网络进行配置

在完成对 eCos 的配置后，将其进行保存。例如，将其保存为配置文件 net386.ecc**，路径为** h：\ ecos20**。（eCos 组件仓库路径为** h：\ ecos20 \ packages**）。此时将产生三个子目录：**h：\ ecos20 \ net386 _ build、h：\ ecos20 \ net386 _ mlt **和** h：\ ecos20 \ net386 _ install**。**

使用图形配置工具的“Build**”菜单对已经配置好的 eCos 进行编译。编译完成后，将在其安装目录（**h：\ ecos20 \ net386 _ install**）下得到编译应用程序所需要的头文件和库文件。**

13.3　Web 服务器编程

作为 eCos 应用程序的 LibHTTPD **嵌入式 Web 服务器软件由两部分组成。其中一部分是提供** httpd **服务器库函数的** LibHTTPD**，另一部分是作为 Web 服务器演示程序。这些软件可以从下述网站上获取：**

LibHTTPD：http：//www.mlbassoc.com/examples/libhttpd-1.3.tar.gz
LibHTTPD **补丁**：http：//www.mlbassoc.com/examples/libhttpd.patch
Web **服务器演示程序**：http：//www.mlbassoc.com/examples/httpd _ application.tar.gz

本书配套光盘上也包含了这些程序，位于目录/ecos _ application **下。为配合 eCos 图形配置工具的使用，某些文件已经过修改，可以直接使用。**

得到上述文件后，进入 Cygwin **环境。进行下面的操作，创建工作目录** httpd**：**

```
$ cd /h/ecos20
$ mkdir httpd
$ cd httpd
```

将上述文件复制到该目录下（h：\ ecos20 \ httpd**），对它们进行解压缩并运行** LibHTTPD **补丁程序：**

```
$ cd /h/ecos20/httpd _ test
```

```
$ tar -zxvf  libhttpd-1.3.tar.gz
$ tar -zxvf  httpd_application.tar.gz
$ patch -p0 <libhttpd.patch
```

完成上述操作后，将光盘上的 build_Make.params **文件复制到当前目录。这是一个用于自动生成** makefile **文件的脚本文件，也可以从** eCos **源码中的** examples **目录下找到该文件。必须根据具体环境对该文件进行简单的修改，才能继续进行后面的操作。下面是本书光盘提供的一个** build_Make.params **文件，该文件已经进行了修改，修改内容主要是使其能使用正确的** eCos **安装目录路径，在此例中安装目录的路径为**/h/ecos20/net386_install**。如果要使用不同的路径或配置名（本例为** net386**），只需对** HOME **参数进行修改。**

```
---------- build_Make.params 文件 ----------
# !/bin/sh
# This script will set up a Makefile fragment with
# platform specifics.  This fragement can be used by
# the automatically generated Makefile (via the script
# 'build_Makefile')
# Copied from 'makefile' the "install" tree
#HOME=${1-'pwd'}
HOME=/h/ecos20/net386_
if [ !-d ${HOME}install ];then
  echo "Not an eCos install tree"
  echo "usage:<eCos_repository>/build_Make.params [<eCos_install_dir>]"
  exit
fi
cat <<EOF >Make.params
# Copied from 'makefile' in the "install" tree
EOF
grep export ${HOME}build/makefile >>Make.params
cat <<EOF >>Make.params
#
# Target specific flags, etc.
#
EOF
cat ${HOME}install/include/pkgconf/ecos.mak >>Make.params
```

对 build_Make.params **文件进行修改后，进入** httpd_application **目录，执行该脚本文件生成编译应用程序所需的** makefile **文件，然后对应用程序（包括** LibHTTPD**）按下面的操作命令进行编译：**

```
$ cd  httpd_application
$ sh  ../build_Make.params  /h/ecos20/net386
$ make
```

上面第二行命令的/h/ecos20/net386 **指的是** eCos **安装树所在目录（即** net386_install **目**

录)。

对应用程序进行编译后,在 h:\ ecos20 \ httpd \ httpd _ application 目录下将得到一个 test 文件,这是一个可在目标系统上运行的可执行文件,也就是简单的一个 Web 服务器软件。

在 httpd _ application 目录下有一个 httpd _ test.c 文件,这就是 Web 服务器演示程序。此程序产生一个简单的 Web 服务器,它提供一个静态页面 test1.html、两个动态页面 index.html 和 test2.html、一个通配符页面 test3.html、两个身份认证页面 login.html 和 login2.html。

下面是本书光盘上的例子程序 httpd _ test.c。LibHTTPD 可以运行在多个嵌入式操作系统之下,它提供的演示程序包含了对其他嵌入式系统的支持。光盘中的该程序已经过修改,只保留了与 eCos 相关的部分。读者可以仔细分析该程序,了解如何创建 eCos Web 服务器以及如何对 Web 服务器内容进行定义和设置的方法。

```
---------- httpd _ test.c 文件 ----------
#include <cyg/hal/hal _ arch.h>
#include <cyg/kernel/kapi.h>
#include <cyg/hal/hal _ if.h>
#define main httpd _ test
#include "config.h"
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>
#include "httpd.h"

/*
** This is a static page of HTML.  It is loaded into the content
** tree using httpdAddStaticContent().
*/
#define test1 _ html "<HTML><BODY>This is just a test</BODY>"

/*
** Below are 2 dynamic pages, each generated by a C function.  The first
** is a simple page that offers a little dynamic info (the process ID)
** and the setups up a test link and a simple form.
**
** The second page processes the form.  As you can see, you can access
** the form data from within your C code by accessing the symbol table
** using httpdGetVariableByName() (and other similar functions).  You
** can also include variables in the string passed to httpdOutput() and
** they will be expanded automatically.
*/
void index _ html(server)
    httpd *server;
{
```

```
        httpdPrintf(server,
                "Hello! This is a eCos Embeded Web Server! <P>\n");
        httpdPrintf(server,
                "Welcome to the httpd server running in process number %d<P>\n",getpid());
        httpdPrintf(server,
            "Click <A HREF=/test1.html>here</A> to view a test page<P>\n");
        httpdPrintf(server,
            "Click <A HREF=/login.html>here</A> to authenticate<P>\n");
        httpdPrintf(server,
            "Or <A HREF=/wildcard/foo>here</A> for a test wildcard page<P>\n");
        httpdPrintf(server, "<P><FORM ACTION=test2.html METHOD=POST>\n");
        httpdPrintf(server, "Enter your name <INPUT NAME=name SIZE=10>\n");
        httpdPrintf(server, "<INPUT TYPE=SUBMIT VALUE=Click!><P></FORM>\
n");
        return;
}

static httpd *_server;
static char _buf[8192];

static void
_do_printf(char *fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);
    vsprintf(_buf, fmt, ap);
    httpdPrintf(_server, _buf);
}

void test2_html(server)
        httpd *server;
{
        httpVar         *variable;
        /*
         * * Grab the symbol table entry to see if the variable exists
         */
        variable = httpdGetVariableByName(server, "name");
        if (variable == NULL)
        {
        httpdPrintf(server,"Missing form data!");
        return;
        }
        /*
```

```
        * * Use httpdOutput() rather than httpdPrintf() so that the variable
        * * embedded in the text is expanded automatically
        */
        httpdOutput(server,"Hello $name");

_server = server;
httpdPrintf(server, "<pre>\n");
show_network_tables(_do_printf);
httpdPrintf(server, "</pre>\n");
}

void test3_html(server)
        httpd *server;
{
        char   *path;
        path = httpdRequestPath(server);
        httpdOutput(server,"Wilcard dynamic request received<P>");
        httpdPrintf(server,"The requested path was %s<P>", path);
}

void login_html(server)
        httpd *server;
{
        if (httpdAuthenticate(server, "LibHTTPD Test") == 0)
                return;
        httpdPrintf(server, "Your username is '%s'<P>\n",
                server->request.authUser);
        httpdPrintf(server, "Your password is '%s'<P>\n",
                server->request.authPassword);
        httpdOutput(server,
                "Click <A HREF=login2.html>here</A> to force reauthentication");
        httpdOutput(server, ".  Use a username = test password = 123");
}

void login2_html(server)
        httpd *server;
{
        if (httpdAuthenticate(server, "LibHTTPD Test") == 0)
        {
                httpdOutput(server, "Authentication failure(1).");
                return;
        }
        if (strcmp(server->request.authUser, "test") != 0 ||
```

```
            strcmp(server->request.authPassword, "123") != 0)
        {
            httpdForceAuthenticate(server, "LibHTTPD Test");
            httpdOutput(server, "Authentication failure (2).");
            return;
        }
        httpdOutput(server, "Your login was accepted.");
}

int main(argc, argv)
        int     argc;
        char    *argv[];
{
        httpd *server;
        char    *host;
        int     port,
                errFlag,
                result;
        extern char *optarg;
        extern int optind, opterr, optopt;
        int c;
        struct          timeval timeout;
char *version = CYGACC_CALL_IF_MONITOR_VERSION();
if (version) {
      printf(version);
} else {
printf("***  Unknown RedBoot version!\n");
}
printf("Starting HTTP server\n");
init_all_network_interfaces();
        host = NULL;
        port = 80;
        /*
        ** Create a server and setup our logging
        */
        server = httpdCreate(host,port);
        if (server == NULL)
        {
            perror("Can't create server");
            exit(1);
        }
        printf("Server OK!\n");
        httpdSetAccessLog(server, stdout);
```

```
        httpdSetErrorLog (server, stdout) ;
        /*
        * * Setup some content for the server
        */
        httpdAddCContent (server,"/", "index.html", HTTP_TRUE,
                NULL, index_html) ;
        httpdAddCContent (server,"/", "test2.html", HTTP_FALSE,
                NULL, test2_html) ;
        httpdAddCContent (server,"/", "login.html", HTTP_FALSE,
                NULL, login_html) ;
        httpdAddCContent (server,"/", "login2.html", HTTP_FALSE,
                NULL, login2_html) ;
        httpdAddCWildcardContent (server,"/wildcard", NULL, test3_html) ;
        httpdAddStaticContent (server, "/", "test1.html", HTTP_FALSE,
                NULL, test1_html) ;

        /*      * * Go into our service loop     */
        timeout.tv_sec = 60 ;
        timeout.tv_usec = 0 ;
        while (1 == 1)
        {
                result = httpdGetConnection (server, &timeout) ;
                if (result == 0)
                {
                        printf ("Timeout ... \n") ;
                        continue ;
                }
                if (result < 0)
                {
                        printf ("Error ... \n") ;
                        continue ;
                }
                if (httpdReadRequest (server) < 0)
                {
                        httpdEndRequest (server) ;
                        continue ;
                }
                httpdProcessRequest (server) ;
                httpdEndRequest (server) ;
        }
}
#define STACK_SIZE (CYGNUM_HAL_STACK_SIZE_TYPICAL + 0x1000)
static char stack [STACK_SIZE] ;
```

```
static cyg_thread thread_data;
static cyg_handle_t thread_handle;

void
cyg_start(void)
{    //Create a main thread, so we can run the scheduler and have time 'pass'
    cyg_thread_create(10,                //Priority - just a number
                      httpd_test,        //entry
                      0,                 //entry parameter
                      "HTTP test",       //Name
                      &stack[0],         //Stack
                      STACK_SIZE,        //Size
                      &thread_handle,    //Handle
                      &thread_data       //Thread data structure
            );
    cyg_thread_resume(thread_handle);    //Start it
    cyg_scheduler_start();
}

char *
strdup(char *s)
{
    char *res = malloc(strlen(s)+1);
    if(res) {
        strcpy(res, s);
    }
    return res;
}
```

13.4 运行 Web 服务器

对 LibHTTPD 和 httpd_test.c 进行编译后，得到执行文件 test。此时可以使用 GDB 将 test 文件加载到目标系统（PC 微机）运行。其步骤如下：

首先将预先准备好的 RedBoot 引导软盘插入目标平台的软驱中，启动目标平台进入 RedBoot。

在 HOST 主机上的 Cygwin 环境下，运行下述命令进入 GDB 图形界面：

```
$ cd /h/ecos20/httpd/httpd_application
$ i386-elf-gdb test
```

在 GDB 图形界面上选择菜单选项“View→Console”，打开控制台窗口。同时选择菜单“Run→Connect to Target”，设置窗口连接参数（波特率为 38400 bit/s），与目标平台进行连接。

选择菜单“Run→Download”，加载 Web 服务器程序 test，此时 GDB 将加载 test 文件到目

标平台上，如图 13-2 所示。

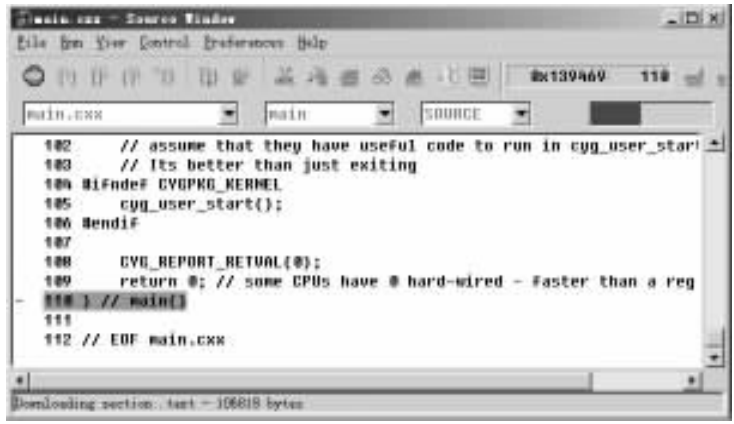


图 13-2　GDB 加载 Web 服务器程序 test

test 程序加载成功后，点击菜单项“Run→Run”，启动 Web 服务器程序 test 在目标系统上运行。当 Web 服务器正常启动后，GDB 控制台窗口（Console Window）将显示服务器正常启动的信息，包括服务器 IP 地址、网关地址、子网掩码、网卡 MAC 地址、广播地址等等，如图 13-3 所示。

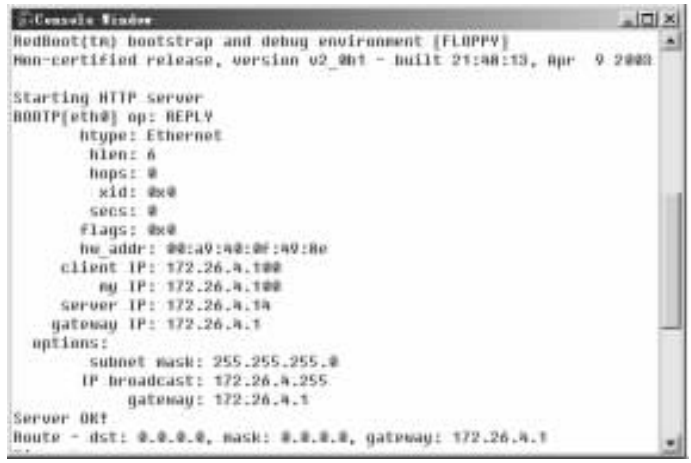


图 13-3　Web 服务器启动后 GDB 控制台信息

Web 服务器程序 test 在目标系统上成功运行后，就成功组建了一个嵌入式 Web 服务器。其他主机通过 IE 浏览器可以浏览该 Web 服务器所提供的演示内容。

例如在 IP 地址为 172.26.4.12 的主机上对该 Web 服务器进行访问，在 IE 浏览器地址栏输入 http://172.26.4.100 并回车后，将出现 Web 服务器提供的 index.html 主页，如图 13-4 所示。

在 index.html 主页上的“Entry your name”输入框内输入“YourName”，并点击“Click !”，将出现下一个网页 test2.html，该网页将显示刚才输入的名字“YourName”，并显示网络相关统计信息，如图 13-5 所示。

通过点击 index.html 主页上不同的链接，可以查看其他的主页内容。

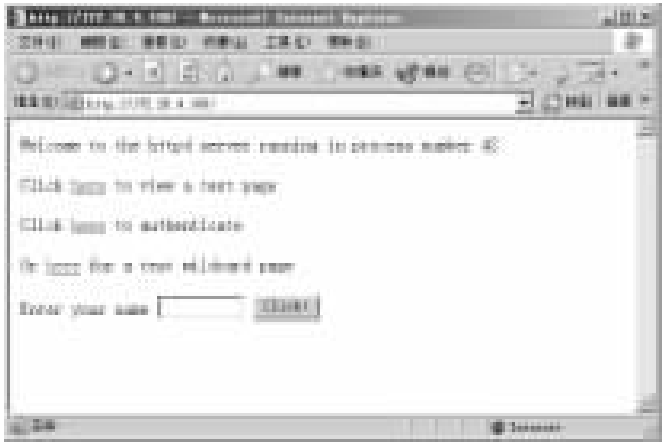


图 13-4　使用浏览器访问嵌入式 Web 服务器

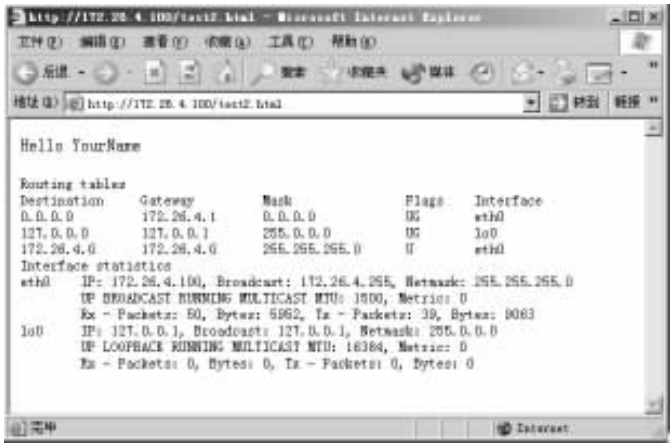


图 13-5　Web 服务器动态网页返回内容

Web 服务器的运行状态以及访问记录将在 GDB 控制台窗口显示。访问记录包括访问该 Web 服务器的浏览器客户端 IP 地址、访问方式（GET 或 POST）、访问目录路径、访问响应码等等。图 13-6 为 Web 服务器的访问记录。

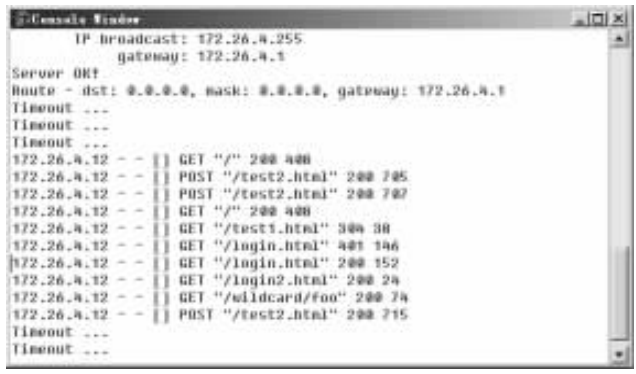


图 13-6　Web 服务器访问记录

Web 服务器软件中提供了超时记录。当超过一定时间没有被访问时，将输出一条超时记录信息“Timeout ...”。

上述例子仅仅是一个演示程序。如果读者有兴趣，可以自己修改 httpd _ test.c 程序，增加 Web 服务器的网页内容。除了以普通 PC 机作为目标平台外，还可以使用其他嵌入式开发平台，只需在 eCos 配置阶段进行相应的配置即可，应用程序（LibHTTPD 库和 httpd _ test.c）不需修改。

附　录

附录 A　eCos 硬件支持情况

这里列举 eCos 对硬件的支持情况，包括对处理器、开发板、设备等的支持，截止时间为 2003 年 6 月。从 eCos 网站上可以查看最新的硬件支持情况。

eCos 对处理器和开发板的支持

处　理　器	开　发　板	厂　商	支持特性			
			以太网	USB	Flash	RedBoot
ARM 处理器系列						
ARM710T	CMA222 CMA232	Cogent				
ARM7TDMI	ARM Integrator	ARM	支持		支持	支持
ARM7TDMI	EPI Dev7	EPI			支持	支持
ARM7TDMI	PID7T	ARM			支持	支持
ARM966E-S	ARM Integrator	ARM	支持		支持	支持
ARM940T	PID9T	ARM			支持	支持
ARM940T	EPT Dev9	EPI			支持	支持
ARM920T	Aglient AAED2000	EPI	支持		支持	支持
Altera Excalibur（ARM922T）	Excalibur EPXA10	Altera			支持	支持
Atmel AT91R40807 （ARM7TDMI 核）	AT91EB40 Evaluation Kit	Atmel			支持	支持
Atmel AT91R40008 （ARM7TDMI 核）	AT91EB40A Evaluation Kit	Atmel			支持	支持
Atmel AT91M42800A （ARM7TDMI 核）	AT91EB42 Evaluation Board	Atmel			支持	支持
Atmel AT91M55800A （ARM7TDMI 核）	AT91EB55 Evaluation Board	Atmel			支持	支持
Cirrus Logic CL-PS7111 （ARM710a）	CL-PS7111 Development Kit	Cirrus Logic	支持		支持	
Cirrus Logic Maverick EP7209 （ARM720T CPU，ARM7TDMI 核）	EP7209 Development Kit	Cirrus Logic	支持		支持	
Cirrus Logic Maverick EP7211 （ARM720T CPU，ARM7TDMI 核）	EP7211 Development Kit	Cirrus Logic	支持		支持	支持

(续)

处 理 器	开 发 板	厂 商	支持特性			
			以太网	USB	Flash	RedBoot
Cirrus Logic Maverick EP7212 (ARM720T CPU，ARM7TDMI 核)	EP7212 Development Kit	Cirrus Logic	支持		支持	支持
Cirrus Logic Maverick EP7312 (ARM720T CPU，ARM7TDMI 核)	EP7312 Development Kit	Cirrus Logic	支持		支持	支持
Samsung KS32C50100 (ARM7TDMI core)	Evaluator-7T	ARM				支持
Samsung KS32C50100 (ARM7TDMI core)	SNDS100	Sam-	sung			支持
Sharp LH77790	ARM AEB-1	ARM				
Intel StrongARM SA-110	EBSA-285	Intel	支持		支持	支持
Intel StrongARM SA-110	Intel SA-1100 Evaluation Platform (Brutus)	Intel				支持
Intel StrongARM SA-110	Intel SA-1100 Multimedia Board	Intel			支持	支持
Intel StrongARM SA-110	Intel SA-1110 Microprocessor 开发板 (Assabet)	Intel	支持	支持	支持	支持
ARM 系列						
Intel StrongARM SA-110	hp iPAQ PocketPC	HP	支持		支持	支持
Intel StrongARM SA-110	Bright Star Engineering commEngine	Bright Star	支持		支持	支持
Intel StrongARM SA-110	Bright Star Engineering nanoEngine	Bright Star	支持		支持	支持
Intel StrongARM SA-110	Bright Star Engineering Flexanet	Bright Star	支持	支持	支持	支持
Intel StrongARM SA-110	CerfCube	Intrinsyc	支持	支持	支持	支持
Intel StrongARM SA-110	CerfPDA	Intrinsyc	支持	支持	支持	支持
Intel XScale IOP310	Intel IQ80310 开发板	Intel	支持		支持	支持
Intel XScale IOP321	Intel IQ80321 开发板	Intel	支持		支持	支持
Intel XScale IXP425	Intel IXDP425	Intel	支持		支持	支持
Intel XScale IXP425	Generic Residential Gateway	ADI	支持		支持	支持
Intel XScale IXC1100	Motorola PrPMC1100	Motorola			支持	支持
Intel XScale PXA250	MPC 5.0	Microplex			支持	支持
Intel XScale PXA250	uE250	Intrinsyc	支持		支持	支持
Intel XScale IOP321	NPWR	Team ASA	支持		支持	支持
IA-32 处理器系列						
x86 及兼容处理器	PC 主板	各厂商	支持			支持

(续)

处 理 器	开 发 板	厂 商	支持特性			
			以太网	USB	Flash	RedBoot
x86 及兼容处理器	Linux synthetic target	各厂商	支持		支持	
Matsushita AM3x 系列						
Panasonic AM31	stdeval1	Syoichi Yamamoto Kyoto				
Panasonic AM31	stdeval1 simulator	GNU 开发工具				
Panasonic AM33	STB reference platform	Syoichi Yamamoto Kyoto			支持	
Panasonic MN103E010 (AM33-2)	ASB2303/ASB2305	Matsushita			支持	支持
MIPS 系列						
MIPS 4Kc	Atlas	MIPS	支持		支持	支持
MIPS 4Kp	Atlas	MIPS	支持		支持	支持
MIPS 4Km	Atlas	MIPS	支持		支持	支持
MIPS 5K	Atlas	MIPS	支持		支持	支持
NEC VR4300	DDB-VRC4373/DDB-VRC4375	NEC	支持		支持	支持
PMC-Sierra RM7000A	Ocelot	Moment-um	支持		支持	支持
Toshiba TMPR3904	JMR-TX3904	Toshiba				
Toshiba TMPR3904	JMR-TX3904 simulator	GNU 开发工具				
Toshiba TMPR4955F	TMPR4955 参考板	Toshiba				
IDT 79RC32334 Integrated Communications Processor	IDT79S334A eval board	IDT	支持		支持	支持
NEC V8xx						
NEC V850/SA1	Cosmo CEB-V850/SA1	Cosmo				
NEC V850/SB1	Cosmo CEB-V850/SB1	Cosmo				
PowerPC 系列						
Motorola MPC555	CME-0555	Axiom			支持	支持
Motorola MPC555	EC555	wuerz-elektronik			支持	支持
Motorola MPC823	CMA287-23	Cogent				
Motorola MPC850	CMA287-50	Cogent				
Motorola MPC860	CMA286-60	Cogent				
Motorola MPC860	MPC8xxFADS	Motorola				
Motorola MPC8245	CSB281	Cogent	支持		支持	支持

(续)

处 理 器	开 发 板	厂 商	支持特性			
			以太网	USB	Flash	RedBoot
Motorola MPC8260	Delphi TigerSHARC-6	Delphi	支持		支持	支持
Motorola MPC8260	MPC8260VADS	Motorola	支持		支持	支持
Motorola MPC860	MBX860	Motorola	支持		支持	支持
Motorola MPC860T	Viper	Analogue & Micro	支持		支持	支持
Motorola MPC850	Adder	Analogue & Micro	支持		支持	支持
Motorola MPC852T	Adder	Analogue & Micro	支持		支持	支持
Motorola MPC855T	TS1000	Allied Telesyn	支持		支持	支持
Motorola MPC603	PSIM architectural simulator	GNU 开发工具				
SPARC 系列						
Fujitsu MB86831，MB86832 或 MB86833	Fujitsu MB86800-MA01	Fujitsu				
LEON 或 ERC32	TSIM ERC32/LEON simulator	Gaisler				
Fujitsu MB86831，MB86832 或 MB86833	SPARClite architectural simulator	GNU 开发工具				
SuperH 系列						
Renesas SH3 SH7708	EDK/SH7708	Renesas			支持	支持
Renesas SH3 SH7708S	CqREEK SH7708	CQ Publishing				
Renesas SH3 SH7750S	CqREEK SH7750	CQ Publishing			支持	支持
Renesas SH3 SH7729R	HS7729PCI	Renesas			支持	
Renesas SH3 SH7709R	SE7709RP01	Renesas				支持
Renesas SH3 SH7709R	SE7709SE01	Renesas				支持
Renesas SH3 SH7729R	SE7729SE01	Renesas				支持
Renesas SH4 SH7751	SE7751	Renesas				支持
Renesas SH4 SH7750	Sega Dreamcast	Sega				支持
M68K	Motorola Coldfire MCF5272	MCF5272C3 Evaluation Board	Motorola	支持		支持

(续)

处 理 器	开 发 板	厂 商	支持特性			
			以太网	USB	Flash	RedBoot
H8	Renesas H8/300H	Akizuki H8/3068 Network Board	Akizuki Denshi	支持		支持
Renesas H8/300H	H8/300H simulator	GNU 开发工具				
FR-V						
Fujitsu FR400	FR-V 400	Fujitsu	支持		支持	支持
CalmRISC	Samsung CalmRISC16	CalmRISC16 Core Evaluation Board	Samsung			支持
Samsung CalmRISC16	CalmRISC32 Core Evaluation Board	Samsung				支持

eCos 对硬件设备的支持

厂 商	型 号
Flsah 设备	
AMD	AM29xxxxx 系列，包括 Am29F040B、AM29LV160、AM29LV320、AM29LV200、STM29W200B、AM29LV640、AM29DL322D、AM29DL323D、AM29DL324D、AM29DL640D、AM29F800、AM29LV800、Toshiba TC58FVB800
Atmel	AT29CxxxA 系列，包括 AT29F040A
	AT29LVxxx 系列，包括 AT29LV1024
	AT49xx1604 系列，包括 AT49BV1604A 和 AT49LV604A
Intel	具有 Boot Block 的 Flash 系列，包括 28F004/400B3、28F008/800B3、28F016/160B3、28F320B3、28F640B3
	StrataFlash 系列，包括 28F128J3A、28F640J3A、28F320J3A
Sharp	LH28F016SCT 系列，包括 LH28F016SCT-Z4、LH28F016SCT-95
串口设备	
Motorola	QUICC SMC/QUICC2 SCC
各厂商	1655x 兼容产品
各厂商	8250 兼容产品
各厂商	eCos 所支持的片内 (on-chip) 串口控制器
以太网控制器	
AMD	Am79C97x (PCNet 系列)
Cirrus Logic	CS8900

(续)

厂　商	型　号
Flsah 设备	
Intel	82559 10/100 以太网控制器 (如 Intel Pro/100)
Intel	82554 Gigabit 以太网控制器 (如 Intel Pro/1000)
Intel	21143PCI/CardBus * 10/100 LAN 控制器
Motorola	QUICC SCC
Motorola	QUICC FEC
Motorola	QUICC2 FEC
National	DP8390 (如 Socket CF LPE + 或 PRETEC CompactLAN 网卡)
SMCC	LAN91C96/LAN91C110
VIA	VIA RHINE
各厂商	eCos 所支持的片内 (on-chip) 以太网控制器
USB 设备	
Intel	SA11X0 on-chip
NEC	uPD985xx on-chip
计时设备	
Dallas	DS1724

附录 B　eCos 实时特性

Intel x86 PC 实时特性

Board：PC
CPU：433MHz Celeron

Startup，main stack　　：stack used　124 size　2912
Startup　　：Interrupt stack used　280 size　4108
Startup　　：Idlethread stack used　62 size　2048

eCos Kernel Timings
Notes：all times are in microseconds（.000001）unless otherwise stated

Reading the hardware clock takes 8 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took　6.75 microseconds（8 raw clock ticks）

Testing parameters：

Clock samples：	32
Threads：	64
Thread switches：	128
Mutexes：	32
Mailboxes：	32
Semaphores：	32
Scheduler operations：	128
Counters：	32
Alarms：	32

				Confidence		
Ave	Min	Max	Var	Ave	Min	Function
3.93	1.68	8.38	0.93	68%	3%	Create thread
0.71	0.00	3.35	0.84	59%	59%	Yield thread [all suspended]
0.65	0.00	5.03	0.84	64%	64%	Suspend [suspended] thread
0.63	0.00	1.68	0.79	62%	62%	Resume thread
0.76	0.00	1.68	0.83	54%	54%	Set priority
0.39	0.00	1.68	0.60	76%	76%	Get priority

1.34	0.00	6.70	0.67	73%	25%	Kill [suspended] thread
0.68	0.00	1.68	0.81	59%	59%	Yield [no other] thread
0.92	0.00	1.68	0.83	54%	45%	Resume [suspended low prio] thread
0.63	0.00	1.68	0.79	62%	62%	Resume [runnable low prio] thread
0.84	0.00	1.68	0.84	100%	50%	Suspend [runnable] thread
0.73	0.00	1.68	0.82	56%	56%	Yield [only low prio] thread
0.58	0.00	1.68	0.76	65%	65%	Suspend [runnable->not runnable]
1.26	0.00	3.35	0.67	71%	26%	Kill [runnable] thread
0.86	0.00	3.35	0.86	98%	50%	Destroy [dead] thread
1.44	0.00	1.68	0.40	85%	14%	Destroy [runnable] thread
4.45	3.35	6.70	0.89	53%	40%	Resume [high priority] thread
1.62	0.00	1.68	0.10	96%	3%	Thread switch
0.41	0.00	1.68	0.61	75%	75%	Scheduler lock
0.48	0.00	1.68	0.69	71%	71%	Scheduler unlock [0 threads]
0.59	0.00	1.68	0.76	64%	64%	Scheduler unlock [1 suspended]
0.45	0.00	1.68	0.65	73%	73%	Scheduler unlock [many suspended]
0.45	0.00	1.68	0.65	73%	73%	Scheduler unlock [many low prio]
0.52	0.00	1.68	0.72	68%	68%	Init mutex
0.79	0.00	5.03	0.93	96%	59%	Lock [unlocked] mutex
0.84	0.00	5.03	0.94	96%	56%	Unlock [locked] mutex
0.63	0.00	1.68	0.79	62%	62%	Trylock [unlocked] mutex
0.52	0.00	1.68	0.72	68%	68%	Trylock [locked] mutex
0.58	0.00	1.68	0.76	65%	65%	Destroy mutex
3.40	3.35	5.03	0.10	96%	96%	Unlock/Lock mutex
0.99	0.00	1.68	0.81	59%	40%	Create mbox
0.47	0.00	1.68	0.68	71%	71%	Peek [empty] mbox
0.79	0.00	5.03	0.93	96%	59%	Put [first] mbox
0.42	0.00	1.68	0.63	75%	75%	Peek [1 msg] mbox
0.79	0.00	1.68	0.83	53%	53%	Put [second] mbox
0.37	0.00	1.68	0.57	78%	78%	Peek [2 msgs] mbox
0.73	0.00	3.35	0.87	59%	59%	Get [first] mbox
0.73	0.00	1.68	0.82	56%	56%	Get [second] mbox
0.79	0.00	3.35	0.88	56%	56%	Tryput [first] mbox
0.73	0.00	3.35	0.87	59%	59%	Tryget [non-empty] mbox
0.68	0.00	3.35	0.85	62%	62%	Peek item [non-empty] mbox
0.63	0.00	1.68	0.79	62%	62%	Peek item [empty] mbox
0.68	0.00	1.68	0.81	59%	59%	Tryget [empty] mbox
0.26	0.00	1.68	0.44	84%	84%	Waiting to get mbox
0.63	0.00	1.68	0.79	62%	62%	Waiting to put mbox
0.73	0.00	3.35	0.87	59%	59%	Delete mbox

```
  3.25    1.68    3.35    0.20   93%    6% Put/Get mbox

  0.63    0.00    1.68    0.79   62%   62% Init semaphore
  0.63    0.00    1.68    0.79   62%   62% Post [0] semaphore
  0.63    0.00    1.68    0.79   62%   62% Wait [1] semaphore
  0.52    0.00    1.68    0.72   68%   68% Trywait [0] semaphore
  0.52    0.00    1.68    0.72   68%   68% Trywait [1] semaphore
  0.52    0.00    1.68    0.72   68%   68% Peek semaphore
  0.21    0.00    1.68    0.37   87%   87% Destroy semaphore
  3.30    1.68    3.35    0.10   96%    3% Post/Wait semaphore

  0.79    0.00    3.35    0.88   56%   56% Create counter
  0.42    0.00    1.68    0.63   75%   75% Get counter value
  0.37    0.00    1.68    0.57   78%   78% Set counter value
  0.73    0.00    1.68    0.82   56%   56% Tick counter
  0.63    0.00    1.68    0.79   62%   62% Delete counter

  0.89    0.00    3.35    0.89   96%   50% Create alarm
  0.84    0.00    1.68    0.84  100%   50% Initialize alarm
  0.52    0.00    1.68    0.72   68%   68% Disable alarm
  0.89    0.00    3.35    0.89   96%   50% Enable alarm
  0.58    0.00    1.68    0.76   65%   65% Delete alarm
  0.63    0.00    1.68    0.79   62%   62% Tick counter [1 alarm]
  5.03    3.35    6.70    0.10   93%    3% Tick counter [many alarms]
  0.94    0.00    1.68    0.82   56%   43% Tick & fire counter [1 alarm]
 11.16   10.06   11.73    0.76   65%   34% Tick & fire counters [>1 together]
  5.19    5.03    6.70    0.28   90%   90% Tick & fire counters [>1 separately]
  0.01    0.00    1.68    0.03   99%   99% Alarm latency [0 threads]
  0.13    0.00    1.68    0.24   92%   92% Alarm latency [2 threads]
  0.94    0.00    3.35    0.85   53%   45% Alarm latency [many threads]
  1.75    1.68    6.70    0.15   96%   96% Alarm -> thread resume latency

   41       0     368  (main stack:  1036)  Thread stack used (1712 total)
All done, main stack               : stack used  1036 size  2912
All done                           :  Interrupt stack used   368 size  4108
All done                           : Idlethread stack used   288 size  2048

Timing complete - 28520 ms total

PASS :<Basic timing OK>
EXIT :<done>
```

Intel IQ80310 XScale 开发板实时特性

Board：Intel IQ80310 XScale Development Kit
CPU：Intel XScale 600MHz

Startup，main stack ：stack used 388 size 2400
Startup ： Interrupt stack used 148 size 4096
Startup ：Idlethread stack used 76 size 1120

eCos Kernel Timings
Notes：all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 73 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took 12.11 microseconds (399 raw clock ticks)

Testing parameters：

Clock samples：	32
Threads：	64
Thread switches：	128
Mutexes：	32
Mailboxes：	32
Semaphores：	32
Scheduler operations：	128
Counters：	32
Alarms：	32

				Confidence		
Ave	Min	Max	Var	Ave	Min	Function
6.53	5.48	8.55	0.50	53%	23%	Create thread
0.37	0.03	3.24	0.18	87%	1%	Yield thread [all suspended]
0.24	0.00	2.06	0.12	87%	1%	Suspend [suspended] thread
0.25	0.00	0.73	0.06	71%	1%	Resume thread
0.36	0.09	0.82	0.10	89%	1%	Set priority
0.03	0.00	0.42	0.05	90%	90%	Get priority
1.07	0.52	6.39	0.18	92%	1%	Kill [suspended] thread
0.33	0.06	0.91	0.08	78%	3%	Yield [no other] thread
0.55	0.03	1.06	0.09	85%	1%	Resume [suspended low prio] thread
0.28	0.00	1.79	0.11	84%	4%	Resume [runnable low prio] thread
0.43	0.00	1.00	0.12	76%	1%	Suspend [runnable] thread

0.31	0.00	1.24	0.09	82%	4%	Yield [only low prio] thread
0.21	0.00	0.42	0.04	73%	1%	Suspend [runnable->not runnable]
1.00	0.88	1.45	0.04	78%	4%	Kill [runnable] thread
0.59	0.42	3.97	0.13	81%	87%	Destroy [dead] thread
1.43	1.27	1.94	0.07	78%	7%	Destroy [runnable] thread
3.12	2.58	5.09	0.33	56%	34%	Resume [high priority] thread
0.87	0.36	1.39	0.07	86%	0%	Thread switch
0.15	0.00	1.39	0.21	81%	81%	Scheduler lock
0.16	0.00	0.64	0.08	85%	7%	Scheduler unlock [0 threads]
0.16	0.00	0.64	0.08	75%	8%	Scheduler unlock [1 suspended]
0.16	0.00	0.70	0.08	78%	6%	Scheduler unlock [many suspended]
0.16	0.00	0.64	0.07	81%	4%	Scheduler unlock [many low prio]
0.45	0.00	1.39	0.34	56%	46%	Init mutex
0.43	0.18	3.27	0.23	87%	87%	Lock [unlocked] mutex
0.48	0.09	3.88	0.26	84%	71%	Unlock [locked] mutex
0.35	0.21	2.24	0.21	87%	84%	Trylock [unlocked] mutex
0.26	0.00	0.67	0.13	78%	9%	Trylock [locked] mutex
0.21	0.00	1.27	0.24	78%	75%	Destroy mutex
2.58	2.09	3.09	0.13	75%	9%	Unlock/Lock mutex
0.99	0.21	2.48	0.41	65%	28%	Create mbox
0.04	0.00	0.39	0.07	90%	87%	Peek [empty] mbox
0.47	0.27	3.48	0.29	90%	78%	Put [first] mbox
0.02	0.00	0.39	0.03	90%	90%	Peek [1 msg] mbox
0.29	0.15	0.58	0.04	68%	3%	Put [second] mbox
0.02	0.00	0.45	0.04	93%	93%	Peek [2 msgs] mbox
0.48	0.21	3.67	0.26	84%	87%	Get [first] mbox
0.35	0.09	0.82	0.11	75%	3%	Get [second] mbox
0.50	0.21	3.18	0.33	90%	68%	Tryput [first] mbox
0.39	0.15	1.39	0.19	78%	68%	Peek item [non-empty] mbox
0.43	0.18	3.33	0.23	87%	90%	Tryget [non-empty] mbox
0.28	0.03	0.79	0.06	68%	3%	Peek item [empty] mbox
0.28	0.21	0.58	0.05	71%	65%	Tryget [empty] mbox
0.01	0.00	0.36	0.02	96%	90%	Waiting to get mbox
0.05	0.00	0.45	0.09	87%	84%	Waiting to put mbox
0.42	0.09	2.88	0.20	84%	12%	Delete mbox
1.39	1.27	2.39	0.14	87%	87%	Put/Get mbox
0.35	0.00	1.36	0.45	75%	68%	Init semaphore
0.19	0.00	0.45	0.04	81%	3%	Post [0] semaphore
0.25	0.21	0.88	0.06	84%	81%	Wait [1] semaphore

```
  0.32    0.06    1.79    0.21   78%   68% Trywait [0] semaphore
  0.20    0.00    0.52    0.06   62%    3% Trywait [1] semaphore
  0.07    0.00    0.45    0.10   84%   81% Peek semaphore
  0.06    0.00    0.52    0.06   71%   78% Destroy semaphore
  1.45    1.42    1.79    0.04   87%   87% Post/Wait semaphore

  0.70    0.00    2.88    0.47   43%   34% Create counter
  0.05    0.00    0.42    0.09   87%   84% Get counter value
  0.02    0.00    0.45    0.04   93%   93% Set counter value
  0.38    0.12    0.58    0.06   59%    3% Tick counter
  0.03    0.00    0.48    0.05   93%   78% Delete counter

  1.10    0.39    4.30    0.47   62%   53% Create alarm
  0.58    0.03    3.12    0.18   87%    3% Initialize alarm
  0.04    0.00    0.42    0.07   90%   90% Disable alarm
  0.54    0.36    1.36    0.12   84%   43% Enable alarm
  0.03    0.00    0.70    0.06   84%   84% Delete alarm
  0.50    0.24    0.97    0.08   84%    6% Tick counter [1 alarm]
  5.30    5.12    5.97    0.14   84%   75% Tick counter [many alarms]
  0.82    0.64    1.36    0.11   78%   43% Tick & fire counter [1 alarm]
 14.13   13.85   14.55    0.09   78%    3% Tick & fire counters [>1 together]
  5.56    5.45    6.00    0.09   78%   71% Tick & fire counters [>1 separately]
  9.69    9.45   12.52    0.22   64%   71% Alarm latency [0 threads]
  9.98    9.48   12.76    0.23   69%   14% Alarm latency [2 threads]
 10.38    9.48   24.67    0.59   74%   45% Alarm latency [many threads]
 11.72   11.30   21.33    0.32   81%   58% Alarm -> thread resume latency

  1.87    1.82   10.42    0.00          Clock/interrupt latency

  3.02    2.58    7.67    0.00          Clock DSR latency

     9       0     260  (main stack:  776)  Thread stack used (1120 total)
All done, main stack               : stack used  776 size  2400
All done                           : Interrupt stack used  268 size  4096
All done                           : Idlethread stack used  244 size  1120

Timing complete - 30300 ms total

PASS:<Basic timing OK>
EXIT:<done>
```

ARM PID 开发板实时特性

Board：ARM PID Evaluation Board

CPU : ARM 7TDMI 20 MHz

Startup, main stack : stack used 404 size 2400
Startup : Interrupt stack used 136 size 4096
Startup : Idlethread stack used 84 size 2048

eCos Kernel Timings
Notes : all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 6 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took 120.74 microseconds (150 raw clock ticks)

Testing parameters :

Clock samples :	32
Threads :	50
Thread switches :	128
Mutexes :	32
Mailboxes :	32
Semaphores :	32
Scheduler operations :	128
Counters :	32
Alarms :	32

				Confidence		
Ave	Min	Max	Var	Ave	Min	Function
99.01	68.00	129.60	15.62	50%	26%	Create thread
21.60	21.60	21.60	0.00	100%	100%	Yield thread [all suspended]
15.65	15.20	16.00	0.39	56%	44%	Suspend [suspended] thread
15.79	15.20	16.00	0.31	74%	26%	Resume thread
23.65	23.20	24.00	0.39	56%	44%	Set priority
2.26	1.60	2.40	0.24	82%	18%	Get priority
51.39	51.20	52.00	0.29	76%	76%	Kill [suspended] thread
21.60	21.60	21.60	0.00	100%	100%	Yield [no other] thread
29.47	28.00	29.60	0.22	86%	2%	Resume [suspended low prio] thread
15.60	15.20	16.00	0.40	100%	50%	Resume [runnable low prio] thread
27.73	24.00	28.00	0.40	74%	2%	Suspend [runnable] thread
21.60	21.60	21.60	0.00	100%	100%	Yield [only low prio] thread
15.65	15.20	16.00	0.39	56%	44%	Suspend [runnable->not runnable]
51.39	51.20	52.00	0.29	76%	76%	Kill [runnable] thread
27.66	27.20	28.80	0.41	54%	44%	Destroy [dead] thread
68.93	64.80	69.60	0.35	72%	2%	Destroy [runnable] thread

91.26	90.40	107.20	0.64	66%	32%	Resume [high priority] thread
49.14	48.80	49.60	0.39	57%	57%	Thread switch
2.20	1.60	2.40	0.30	75%	25%	Scheduler lock
10.20	9.60	10.40	0.30	75%	25%	Scheduler unlock [0 threads]
10.20	9.60	10.40	0.30	75%	25%	Scheduler unlock [1 suspended]
10.20	9.60	10.40	0.30	75%	25%	Scheduler unlock [many suspended]
10.20	9.60	10.40	0.30	75%	25%	Scheduler unlock [many low prio]
6.85	6.40	7.20	0.39	56%	43%	Init mutex
18.40	18.40	18.40	0.00	100%	100%	Lock [unlocked] mutex
19.57	19.20	20.00	0.40	53%	53%	Unlock [locked] mutex
16.55	16.00	16.80	0.34	68%	31%	Trylock [unlocked] mutex
14.55	14.40	15.20	0.24	81%	81%	Trylock [locked] mutex
3.55	3.20	4.00	0.39	56%	56%	Destroy mutex
119.85	119.20	120.00	0.24	81%	18%	Unlock/Lock mutex
12.85	12.80	13.60	0.09	93%	93%	Create mbox
1.65	1.60	2.40	0.09	93%	93%	Peek [empty] mbox
20.70	20.00	20.80	0.17	87%	12%	Put [first] mbox
1.65	1.60	2.40	0.09	93%	93%	Peek [1 msg] mbox
20.70	20.00	20.80	0.17	87%	12%	Put [second] mbox
1.65	1.60	2.40	0.09	93%	93%	Peek [2 msgs] mbox
20.85	20.80	21.60	0.09	93%	93%	Get [first] mbox
20.85	20.80	21.60	0.09	93%	93%	Get [second] mbox
19.90	19.20	20.00	0.17	87%	12%	Tryput [first] mbox
17.60	17.60	17.60	0.00	100%	100%	Peek item [non-empty] mbox
20.90	20.80	21.60	0.17	87%	87%	Tryget [non-empty] mbox
16.80	16.80	16.80	0.00	100%	100%	Peek item [empty] mbox
17.65	17.60	18.40	0.09	93%	93%	Tryget [empty] mbox
1.85	1.60	2.40	0.34	68%	68%	Waiting to get mbox
1.85	1.60	2.40	0.34	68%	68%	Waiting to put mbox
19.40	19.20	20.00	0.30	75%	75%	Delete mbox
65.05	64.80	65.60	0.34	68%	68%	Put/Get mbox
7.05	6.40	7.20	0.24	81%	18%	Init semaphore
15.55	15.20	16.00	0.39	56%	56%	Post [0] semaphore
17.35	16.80	17.60	0.34	68%	31%	Wait [1] semaphore
14.60	14.40	15.20	0.30	75%	75%	Trywait [0] semaphore
14.20	13.60	14.40	0.30	75%	25%	Trywait [1] semaphore
4.55	4.00	4.80	0.34	68%	31%	Peek semaphore
3.75	3.20	4.00	0.34	68%	31%	Destroy semaphore
70.85	70.40	71.20	0.39	56%	43%	Post/Wait semaphore

```
  6.05    5.60    6.40    0.39   56%   43% Create counter
  2.25    1.60    2.40    0.24   81%   18% Get counter value
  2.25    1.60    2.40    0.24   81%   18% Set counter value
 19.70   19.20   20.00    0.37   62%   37% Tick counter
  3.45    3.20    4.00    0.34   68%   68% Delete counter

  9.05    8.80    9.60    0.34   68%   68% Create alarm
 29.60   29.60   29.60    0.00  100%  100% Initialize alarm
  2.15    1.60    2.40    0.34   68%   31% Disable alarm
 29.35   28.80   29.60    0.34   68%   31% Enable alarm
  5.10    4.80    5.60    0.37   62%   62% Delete alarm
 23.20   23.20   23.20    0.00  100%  100% Tick counter [1 alarm]
138.00  137.60  138.40    0.40  100%   50% Tick counter [many alarms]
 40.40   40.00   40.80    0.40  100%   50% Tick & fire counter [1 alarm]
704.25  697.60  804.00   12.47   93%   93% Tick & fire counters [>1 together]
155.20  155.20  155.20    0.00  100%  100% Tick & fire counters [>1 separately]
105.20  104.80  151.20    0.76   99%   94% Alarm latency [0 threads]
117.57  104.80  149.60    7.13   57%   25% Alarm latency [2 threads]
117.49  104.80  148.80    7.10   58%   26% Alarm latency [many threads]
192.59  177.60  316.00    1.93   98%    0% Alarm -> thread resume latency

 22.10   21.60   24.00    0.00          Clock/interrupt latency

 38.69   32.80   61.60    0.00          Clock DSR latency

  297     276     316   (main stack:  752)  Thread stack used  (1120 total)
All done, main stack               :stack used   752 size  2400
All done                           : Interrupt stack used   288 size  4096
All done                           :Idlethread stack used   272 size  2048

Timing complete - 30350 ms total

PASS :<Basic timing OK>
EXIT :<done>
```

Motorola MBX 实时特性

```
Board :Motorola MBX
CPU   :Motorola MPC860 66MHZ

Startup, main stack                :stack used   643 size  5664
```

```
Startup          : Interrupt stack used   427 size  4096
Startup          : Idlethread stack used  236 size  2048

eCos Kernel Timings
Notes : all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took   25.36 microseconds (79 raw clock ticks)

Testing parameters :
    Clock samples :          32
    Threads :                16
    Thread switches :        128
    Mutexes :                32
    Mailboxes :              32
    Semaphores :             32
    Scheduler operations :   128
    Counters :               32
    Alarms :                 32

                                        Confidence
     Ave      Min      Max      Var     Ave   Min   Function
  ====  ====  ====  ====  =====  ======
    27.58    25.60    44.16     2.07   93%   93%  Create thread
     5.94     5.76     7.04     0.22   93%   62%  Yield thread [all suspended]
     6.06     5.44    10.56     0.57   75%   75%  Suspend [suspended] thread
     5.42     4.80     9.60     0.53   87%   81%  Resume thread
     7.10     6.40    14.08     0.90   93%   87%  Set priority
     0.86     0.64     1.92     0.22   93%   50%  Get priority
    16.74    15.04    36.48     2.47   93%   93%  Kill [suspended] thread
     6.14     5.76    10.56     0.55   93%   93%  Yield [no other] thread
     9.74     8.96    18.56     1.10   93%   93%  Resume [suspended low prio] thread
     5.28     4.80     9.28     0.54   93%   81%  Resume [runnable low prio] thread
     9.40     8.32    18.56     1.14   93%   93%  Suspend [runnable] thread
     6.04     5.76     8.96     0.38   93%   93%  Yield [only low prio] thread
     5.68     5.12     9.60     0.52   68%   75%  Suspend [runnable->not runnable]
    16.10    14.40    35.20     2.39   93%   93%  Kill [runnable] thread
     8.54     7.68    16.00     0.94   93%   87%  Destroy [dead] thread
    20.20    18.56    40.64     2.55   93%   93%  Destroy [runnable] thread
    39.02    36.48    57.28     3.28   87%   87%  Resume [high priority] thread
    13.13    12.80    22.08     0.15   78%   20%  Thread switch
```

0.59	0.32	1.60	0.09	82%	16%	Scheduler lock
3.67	3.52	5.12	0.17	99%	54%	Scheduler unlock [0 threads]
3.67	3.52	4.80	0.17	99%	53%	Scheduler unlock [1 suspended]
3.67	3.52	4.80	0.17	54%	54%	Scheduler unlock [many suspended]
3.69	3.52	5.12	0.17	99%	50%	Scheduler unlock [many low prio]
2.41	2.24	5.44	0.25	96%	75%	Init mutex
6.83	6.40	11.84	0.34	75%	90%	Lock [unlocked] mutex
6.74	6.40	13.12	0.40	96%	96%	Unlock [locked] mutex
5.53	5.12	9.60	0.25	84%	12%	Trylock [unlocked] mutex
4.84	4.48	7.36	0.17	78%	15%	Trylock [locked] mutex
0.34	0.00	0.96	0.06	90%	3%	Destroy mutex
56.10	55.68	59.52	0.21	93%	3%	Unlock/Lock mutex
4.72	4.48	10.24	0.37	96%	96%	Create mbox
0.75	0.64	1.92	0.16	75%	75%	Peek [empty] mbox
6.79	6.40	12.80	0.41	96%	90%	Put [first] mbox
0.46	0.32	1.60	0.19	93%	68%	Peek [1 msg] mbox
6.68	6.40	12.16	0.37	96%	96%	Put [second] mbox
0.50	0.32	1.60	0.20	93%	56%	Peek [2 msgs] mbox
7.13	6.40	14.08	0.49	90%	46%	Get [first] mbox
6.97	6.40	13.44	0.47	84%	78%	Get [second] mbox
6.24	5.76	11.52	0.38	78%	81%	Tryput [first] mbox
5.98	5.44	11.20	0.39	78%	62%	Peek item [non-empty] mbox
6.52	6.08	13.12	0.49	93%	81%	Tryget [non-empty] mbox
5.50	5.12	10.24	0.30	68%	28%	Peek item [empty] mbox
5.76	5.44	10.88	0.32	96%	96%	Tryget [empty] mbox
0.50	0.32	1.60	0.19	96%	53%	Waiting to get mbox
0.50	0.32	1.60	0.19	96%	53%	Waiting to put mbox
7.45	7.04	15.04	0.49	96%	93%	Delete mbox
37.47	36.80	48.64	0.70	96%	96%	Put/Get mbox
2.49	2.24	6.08	0.28	96%	56%	Init semaphore
5.09	4.80	8.64	0.27	46%	46%	Post [0] semaphore
6.25	5.76	10.88	0.32	93%	3%	Wait [1] semaphore
4.84	4.48	8.32	0.23	68%	25%	Trywait [0] semaphore
4.98	4.80	8.00	0.26	96%	71%	Trywait [1] semaphore
1.66	1.28	3.84	0.20	68%	15%	Peek semaphore
1.24	0.96	3.20	0.17	65%	31%	Destroy semaphore
40.74	40.32	49.28	0.53	96%	96%	Post/Wait semaphore
2.65	2.24	6.08	0.23	84%	9%	Create counter
0.85	0.64	2.24	0.22	90%	53%	Get counter value

```
  0.68    0.64    1.92    0.08   96%   96% Set counter value
  7.13    6.72    8.64    0.24   78%   18% Tick counter
  1.30    0.96    3.20    0.12   84%   12% Delete counter

  3.69    3.52    7.68    0.29   96%   84% Create alarm
  8.98    8.32   17.60    0.61   68%   62% Initialize alarm
  0.96    0.64    2.88    0.14   71%   21% Disable alarm
  8.76    8.32   17.60    0.59   96%   87% Enable alarm
  1.99    1.60    5.12    0.21   81%   12% Delete alarm
  7.44    7.36    9.92    0.15   96%   96% Tick counter [1 alarm]
 21.68   21.44   24.64    0.25   96%   53% Tick counter [many alarms]
 10.95   10.56   15.04    0.26   78%   18% Tick & fire counter [1 alarm]
132.79  132.48  136.32    0.23   59%   37% Tick & fire counters [>1 together]
 25.18   24.96   28.80    0.29   96%   65% Tick & fire counters [>1 separately]
 23.06   22.72   47.36    0.40   98%   98% Alarm latency [0 threads]
 31.53   27.20   56.00    0.63   96%    0% Alarm latency [2 threads]
 36.86   30.40   58.88    4.15   50%   28% Alarm latency [many threads]

 11.41    8.96   16.32    0.00         Clock/interrupt latency

  609     603     651  (main stack:  1059)  Thread stack used (1704 total)
All done, main stack               : stack used  1059 size  5664
All done                  :  Interrupt stack used   251 size  4096
All done                  : Idlethread stack used   587 size  2048

Timing complete - 23690 ms total

PASS :< Basic timing OK >
EXIT :< done >
```

Intel SA1100 (Brutus) 实时特性

```
Board : Intel SA1100 (Brutus)
CPU  :  StrongARM 221.2 MHz

Microseconds for one run through Dhrystone :     3.3
Dhrystones per Second :                        306748.5

VAX MIPS rating =       174.586

Startup, main stack                : stack used   404 size  2400
Startup                   :  Interrupt stack used   136 size  4096
```

```
Startup                    : Idlethread stack used    87 size  2048

eCos Kernel Timings
Notes : all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took    3.09 microseconds (11 raw clock ticks)

Testing parameters :
    Clock samples :          32
    Threads :                64
    Thread switches :        128
    Mutexes :                32
    Mailboxes :              32
    Semaphores :             32
    Scheduler operations :   128
    Counters :               32
    Alarms :                 32

                                          Confidence
     Ave     Min     Max     Var     Ave   Min   Function
  ====    ====    ====    ====    =====  ======
    6.63    5.43   18.99    0.77    70%   37%  Create thread
    0.83    0.81    2.17    0.04    98%   98%  Yield thread [all suspended]
    1.27    0.81    5.15    0.30    68%   73%  Suspend [suspended] thread
    1.25    0.81    5.15    0.25    82%    1%  Resume thread
    1.52    1.09    7.87    0.30    78%   75%  Set priority
    0.97    0.54    2.71    0.28    64%   51%  Get priority
    3.45    2.71   19.53    0.66    84%   76%  Kill [suspended] thread
    0.90    0.81    6.24    0.17    98%   98%  Yield [no other] thread
    1.86    1.36    6.24    0.33    68%   50%  Resume [suspended low prio] thread
    1.25    0.81    5.15    0.25    82%    1%  Resume [runnable low prio] thread
    2.01    1.63   10.04    0.32    70%   84%  Suspend [runnable] thread
    0.90    0.81    6.24    0.17    98%   98%  Yield [only low prio] thread
    1.25    0.81    5.15    0.24    84%    1%  Suspend [runnable->not runnable]
    2.92    1.90   18.72    0.57    85%   43%  Kill [runnable] thread
    2.45    1.90   10.31    0.33    95%   54%  Destroy [dead] thread
    3.95    2.71   23.60    0.89    68%   54%  Destroy [runnable] thread
    8.55    6.24   19.53    1.15    60%   23%  Resume [high priority] thread
    1.85    1.63   11.94    0.21    49%   49%  Thread switch

    0.25    0.00    1.63    0.05    89%   10%  Scheduler lock
```

0.52	0.27	1.90	0.07	85%	13%	Scheduler unlock [0 threads]
0.51	0.27	1.36	0.06	85%	13%	Scheduler unlock [1 suspended]
0.51	0.27	1.36	0.06	85%	13%	Scheduler unlock [many suspended]
0.51	0.27	1.63	0.06	85%	13%	Scheduler unlock [many low prio]
0.58	0.27	3.53	0.20	71%	21%	Init mutex
1.07	0.54	5.70	0.35	87%	59%	Lock [unlocked] mutex
1.14	0.81	6.51	0.40	96%	81%	Unlock [locked] mutex
0.96	0.54	5.15	0.34	68%	65%	Trylock [unlocked] mutex
0.94	0.54	4.88	0.34	65%	65%	Trylock [locked] mutex
0.33	0.27	2.17	0.11	96%	96%	Destroy mutex
4.21	3.80	10.85	0.41	71%	96%	Unlock/Lock mutex
0.76	0.54	4.07	0.25	96%	56%	Create mbox
0.75	0.54	1.90	0.20	84%	50%	Peek [empty] mbox
1.56	1.09	6.78	0.39	68%	59%	Put [first] mbox
0.75	0.54	1.90	0.20	84%	50%	Peek [1 msg] mbox
1.55	1.09	6.78	0.40	68%	62%	Put [second] mbox
0.77	0.54	1.63	0.17	46%	37%	Peek [2 msgs] mbox
1.67	1.09	6.24	0.31	87%	34%	Get [first] mbox
1.63	1.09	6.24	0.31	75%	34%	Get [second] mbox
1.50	1.09	6.51	0.40	56%	62%	Tryput [first] mbox
1.58	1.09	5.43	0.37	68%	53%	Peek item [non-empty] mbox
1.79	1.09	7.05	0.43	71%	25%	Tryget [non-empty] mbox
1.29	1.09	5.15	0.32	87%	87%	Peek item [empty] mbox
1.33	1.09	5.97	0.37	96%	84%	Tryget [empty] mbox
0.73	0.54	1.90	0.21	84%	56%	Waiting to get mbox
0.76	0.54	1.90	0.19	40%	43%	Waiting to put mbox
1.47	1.09	6.78	0.39	59%	84%	Delete mbox
2.70	2.17	12.75	0.63	96%	96%	Put/Get mbox
0.47	0.27	2.71	0.20	96%	50%	Init semaphore
0.89	0.54	4.88	0.33	56%	75%	Post [0] semaphore
0.96	0.54	5.15	0.33	71%	75%	Wait [1] semaphore
0.86	0.54	4.88	0.32	96%	81%	Trywait [0] semaphore
0.69	0.54	3.26	0.22	96%	75%	Trywait [1] semaphore
0.49	0.27	3.26	0.28	84%	84%	Peek semaphore
0.39	0.27	2.44	0.19	96%	78%	Destroy semaphore
2.83	2.44	11.66	0.55	96%	96%	Post/Wait semaphore
0.52	0.27	3.26	0.20	56%	40%	Create counter
0.59	0.00	2.71	0.34	81%	46%	Get counter value
0.36	0.00	2.44	0.21	81%	9%	Set counter value
1.13	0.81	2.98	0.26	59%	37%	Tick counter

```
  0.39    0.27    1.90    0.19   90%   78% Delete counter

  0.86    0.54    4.07    0.24   65%   31% Create alarm
  1.86    1.36    9.77    0.54   96%   90% Initialize alarm
  0.77    0.54    2.71    0.23   84%   50% Disable alarm
  1.86    1.36    9.22    0.51   96%   75% Enable alarm
  0.89    0.54    3.26    0.25   65%   21% Delete alarm
  0.99    0.81    3.26    0.21   96%   59% Tick counter [1 alarm]
  4.22    4.07    6.78    0.22   96%   71% Tick counter [many alarms]
  1.51    1.36    4.61    0.24   96%   78% Tick & fire counter [1 alarm]
 20.29   20.07   23.33    0.23   96%   53% Tick & fire counters [>1 together]
  4.71    4.61    7.87    0.20   96%   96% Tick & fire counters [>1 separately]
  2.88    2.71   23.87    0.33   99%   99% Alarm latency [0 threads]
  3.24    2.71   17.36    0.40   79%   58% Alarm latency [2 threads]
 15.71   12.48   27.40    1.47   53%   17% Alarm latency [many threads]
  5.95    5.43   64.56    1.02   97%   97% Alarm -> thread resume latency

  3.25    0.81   14.11    0.00         Clock/interrupt latency

  2.68    1.09   12.75    0.00         Clock DSR latency

   29       0     316   (main stack:  764)  Thread stack used  (1120 total)
All done, main stack               : stack used   764 size  2400
All done                           : Interrupt stack used   288 size  4096
All done                           : Idlethread stack used   260 size  2048

Timing complete - 30280 ms total
```

Cogent CMA MPC860 (PowerPC) 评估板实时特性

```
Board : Cogent CMA MPC860 (PowerPC) Evaluation
CPU   : MPC860, revision A3 33MHz

eCOS Kernel Timings
Note : all times are in microseconds (.000001) unless otherwise stated

Reading the hardware clock takes 0 'ticks' overhead
... this value will be factored out of all other measurements
Clock interrupt took   14.46 microseconds (30 raw clock ticks)

Testing parameters :
    Clock samples :        32
```

Threads : 24
Thread switches : 128
Mutexes : 32
Mailboxes : 32
Semaphores : 32
Scheduler operations : 128
Counters : 32
Alarms : 32

				Confidence		
Ave	Min	Max	Var	Ave	Min	Function
26.78	23.52	41.76	1.97	66%	37%	Create thread
4.00	3.84	4.80	0.23	70%	70%	Yield thread [all suspended]
3.78	3.36	7.68	0.38	50%	45%	Suspend [suspended] thread
3.56	3.36	7.68	0.37	95%	91%	Resume thread
5.28	4.32	12.96	0.76	83%	66%	Set priority
0.84	0.48	3.84	0.39	91%	54%	Get priority
11.76	10.08	32.16	1.70	95%	95%	Kill [suspended] thread
4.14	3.84	8.64	0.45	95%	75%	Yield [no other] thread
7.14	5.76	17.76	1.07	79%	70%	Resume [suspended low prio] thread
3.60	3.36	8.16	0.42	95%	87%	Resume [runnable low prio] thread
6.10	5.28	14.88	0.80	62%	70%	Suspend [runnable] thread
4.00	3.84	5.76	0.25	79%	79%	Yield [only low prio] thread
3.66	3.36	8.64	0.47	95%	79%	Suspend [runnable->not runnable]
11.66	10.08	30.24	1.58	79%	91%	Kill [runnable] thread
31.12	27.84	53.28	2.35	87%	50%	Resume [high priority] thread
7.52	7.20	15.84	0.30	50%	48%	Thread switch
1.00	0.48	2.88	0.21	63%	14%	Scheduler lock
2.57	2.40	3.84	0.23	65%	65%	Scheduler unlock [0 threads]
2.58	2.40	4.32	0.23	64%	64%	Scheduler unlock [1 suspended]
2.59	2.40	4.32	0.24	62%	62%	Scheduler unlock [many suspended]
2.59	2.40	4.32	0.24	61%	61%	Scheduler unlock [many low prio]
1.69	1.44	5.76	0.37	96%	71%	Init mutex
4.15	3.84	10.56	0.47	96%	75%	Lock [unlocked] mutex
5.82	5.28	10.56	0.38	62%	28%	Unlock [locked] mutex
3.70	3.36	8.64	0.41	96%	59%	Trylock [unlocked] mutex
3.42	2.88	6.72	0.26	75%	15%	Trylock [locked] mutex
0.36	0.00	1.92	0.25	62%	34%	Destroy mutex
43.41	42.72	45.12	0.34	81%	3%	Unlock/Lock mutex

3.27	2.88	8.16	0.39	96%	50%	Create mbox
0.57	0.00	2.40	0.34	50%	21%	Peek [empty] mbox
6.16	5.76	11.04	0.48	87%	87%	Put [first] mbox
0.48	0.00	1.92	0.27	50%	28%	Peek [1 msg] mbox
5.92	5.28	10.56	0.35	90%	6%	Put [second] mbox
0.60	0.00	2.40	0.30	62%	12%	Peek [2 msgs] mbox
4.69	4.32	12.00	0.54	93%	93%	Get [first] mbox
4.68	4.32	11.52	0.52	93%	93%	Get [second] mbox
5.86	5.28	11.04	0.47	62%	31%	Tryput [first] mbox
4.00	3.36	9.12	0.38	87%	9%	Peek item [non-empty] mbox
4.59	3.84	12.48	0.61	71%	75%	Tryget [non-empty] mbox
3.75	3.36	7.68	0.34	53%	43%	Peek item [empty] mbox
3.93	3.36	9.60	0.45	65%	31%	Tryget [empty] mbox
0.63	0.00	2.40	0.28	68%	6%	Waiting to get mbox
0.54	0.00	1.92	0.19	75%	9%	Waiting to put mbox
4.84	4.32	12.00	0.47	56%	40%	Delete mbox
24.18	23.52	29.76	0.66	81%	75%	Put/Get mbox
1.72	0.96	3.84	0.33	90%	6%	Init semaphore
3.15	2.88	6.24	0.34	96%	62%	Post [0] semaphore
3.85	3.36	8.64	0.30	68%	28%	Wait [1] semaphore
3.24	2.88	6.24	0.34	46%	46%	Trywait [0] semaphore
3.22	2.88	6.24	0.32	50%	46%	Trywait [1] semaphore
0.96	0.48	2.88	0.12	84%	12%	Peek semaphore
0.99	0.96	1.92	0.06	96%	96%	Destroy semaphore
24.71	24.00	28.80	0.40	87%	6%	Post/Wait semaphore
2.31	1.44	6.24	0.77	46%	56%	Create counter
0.45	0.00	0.96	0.08	87%	9%	Get counter value
0.42	0.00	0.96	0.16	75%	18%	Set counter value
4.14	3.84	4.80	0.26	50%	43%	Tick counter
0.91	0.48	2.40	0.19	71%	21%	Delete counter
5.23	4.32	7.68	0.61	65%	53%	Create alarm
5.58	4.80	12.96	0.72	68%	84%	Initialize alarm
0.75	0.48	1.92	0.30	90%	56%	Disable alarm
8.02	7.20	14.40	0.53	84%	68%	Enable alarm
1.32	0.96	3.84	0.29	56%	40%	Delete alarm
4.63	4.32	6.24	0.28	53%	43%	Tick counter [1 alarm]
23.67	23.52	25.44	0.23	78%	78%	Tick counter [many alarms]
7.24	6.72	10.56	0.21	84%	12%	Tick & fire counter [1 alarm]
106.83	106.56	110.40	0.35	96%	65%	Tick & fire counters [>1 together]
26.18	25.44	29.76	0.46	81%	9%	Tick & fire counters [>1 separately]

```
 10.79    10.08    29.28     0.66   53%   55% Alarm latency [0 threads]
 17.20    13.92    35.52     1.48   67%   21% Alarm latency [2 threads]
 29.69    22.56    47.04     3.58   57%   17% Alarm latency [many threads]

  7.66     3.84    19.20     0.00          Clock/interrupt latency

Timing complete - 23530 ms total

PASS :< Basic timing OK >
EXIT :< done >
```

参 考 文 献

1 Nick Garnett, Jonathan Larmour, Andrew Lunn, Gary Thomas, Bart Veer. eCos Reference Manual. Red Hat Inc., 2003

2 eCos User's Guide. Red Hat Inc. & eCosCentric Ltd., 2003

3 eCos Tutorial. Red Hat Inc., 2002

4 Bart Veer, John Dallaway. The eCos Component Writer's Guide. Red Hat Inc., 2001

5 Anthony J. Massa, Embedded Software Development with eCos. Prentice Hall PTR, 2002

6 LibHTTPD API Guide and Reference. Hughes Technologies Pty Ltd., 2002