凌阳科技

跟我学 eCos 嵌入式操作系统

在 SPCE3200 上的应用——驱动开发篇

V1.0 – 2007.9.12

凌阳单片机技术资料

http://www.unsp.com

版权声明

前　言

SPCE3200 是台湾凌阳科技推出以 S+core7（凌阳科技自主研发）为内核的 32 位嵌入式开发系统，内嵌 12 位 ADC、16 位 DAC；具有 UART、SPI、I2C、SIO、USB 等标准硬件控制器接口；具有 6 个 Timer、实时时钟和时基；有 Nor 型 Flash、Nand 型 Flash、SD 卡控制器；具有 TFT、STN 型 LCD 控制器及 TV 控制器接口；有 MPEG4 编解码器、CMOS 接口单元等资源。

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，而且它的的代码量很小，通常为几十到几百 KB。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。

本书介绍 eCos 嵌入式操作系统在 SPCE3200 平台上的设备驱动开发，全书共分为 7 章：

第 1 章 eCos 设备驱动程序概述介绍了设备驱动程序的基础知识，包括设备驱动程序的分类、驱动程序脚本 cdl 文件、eCos 数据库文件 ecos.db 等。

第 2 章字符设备驱动程序初步介绍了字符型设备的驱动程序结构、编写步骤及程序设计等。

第 3 章字符型设备驱动以 IOB 及 SPI 设备为例介绍了两种典型字符型设备驱动程序的编写方法。

第 4 章块设备驱动程序初步介绍了块设备驱动程序区别于字符设备驱动程序模块的设计方法。

第 5 章基于底层设备的设备驱动程序设计介绍了基于其他设备的设备驱动程序的设计方法。

第 6 章 SD 卡驱动程序设计介绍了 SD 卡即作为块设备又作为基于底层设备的设备驱动程序的设计方法。

本书从使用的角度进行介绍，没有涉及太多的原理或者概念介绍，在不需要非常清楚地理解 eCos 概念的前提下，按照本书就可以编写一些基本设备的驱动程序。在本书中涉及有关 eCos 概念，请读者查找 eCos 相关资料及书籍；有关 SPCE3200 的原理及其他信息，请读者查找凌阳科技大学计划提供的 SPCE3200 相关书籍及资料。书中的例子都是典型设备，代表了一类设备，读者通过这些例子的学习，可以掌握 eCos 字符及块设备驱动的设计思想和方法。

全书由北京北阳电子技术有限公司大学计划处罗亚非处长统一审稿，由凌阳科技大学计划处技术支持部封鸿雁、李健编写。

有关 SPCE3200 的最新资料和相关信息，请关注凌阳科技大学计划网站 www.unsp.com 和产品附带光盘；如果读者在阅读本书使用 SPCE3200 的过程中有疑惑之处，请到凌阳科技大学计划网站技术论坛（www. bbs.unsp.com）提出，或者邮件至 unsp@sunplus.com.cn，凌阳科技大学计划工程师将及时与广大用户交流。

由于编者水平有限，书中难免错误纰漏，请广大读者批评指正。

编者

2007-9-12

目　录

1　eCos设备驱动程序概述 1

1.1　设备驱动程序基础知识 1

1.1.1　概述 1

1.1.2　驱动程序分类 1

1.2　加载和移除设备驱动程序包 2

1.2.1　加载设备驱动程序包 2

1.2.2　移除设备驱动程序包 4

1.3　cdl脚本 5

1.3.1　cdl命令简介 5

1.3.2　cdl脚本基本结构 6

1.3.3　cdl属性 7

1.3.4　驱动程序包的脚本cdl文件 9

1.4　eCos库的数据库文件ecos.db 10

1.4.1　加载驱动程序包到数据库 10

1.4.2　在ecos.db文件中加载和删除驱动程序包 11

1.5　编写设备驱动的一般步骤 12

2　字符设备驱动程序初步 14

2.1　字符设备驱动程序概述 14

2.2　字符设备驱动程序的目录结构 14

2.3　字符设备驱动程序的基本结构 14

2.4　字符设备驱动程序设计 15

2.4.1　设计CDL文件 15

2.4.2　编写字符设备表入口 17

2.4.3　实现字符设备接口函数 18

2.4.4　简单的字符设备驱动程序解析 22

2.4.5　向组件库添加驱动程序包 25

2.5　字符设备驱动程序的使用 25

3　字符型设备驱动程序设计 31

3.1　GPIO驱动程序设计 31

3.1.1　建立驱动程序目录 31

3.1.2　建立驱动程序文件 31

3.1.3　设计CDL脚本文件 32

3.1.4　设计头文件 34

3.1.5 设计源程序文件 34
3.1.6 编写IOB设备表入口 34
3.1.7 编写IOB设备驱动接口函数 35
3.1.8 IOB驱动程序范例 41
3.1.9 向eCos数据库中添加IOB驱动程序组件包 44
3.1.10 用户测试程序 45
3.2 SPI驱动程序设计 48
3.2.1 建立驱动程序目录 48
3.2.2 建立驱动程序文本文件 48
3.2.3 设计CDL脚本文件 48
3.2.4 设计头文件 51
3.2.5 设计源程序文件 51
3.2.6 编写SPI设备表入口 52
3.2.7 编写SPI设备驱动中断服务程序 53
3.2.8 编写SPI设备驱动接口函数 55
3.2.9 SPI驱动程序范例 64
3.2.10 向eCos数据库中添加SPI驱动程序组件包 68
3.2.11 用户测试程序 69
4 块设备驱动程序初步 73
4.1 块设备驱动程序概述 73
4.2 块设备驱动程序设计 73
4.2.1 设计CDL脚本文件 73
4.2.2 编写设备表入口 74
4.2.3 实现设备接口函数 75
4.2.4 块设备驱动程序解析 77
4.2.5 向eCos数据库中添加驱动程序包 79
4.3 块设备驱动程序的使用 80
5 基于底层设备的设备驱动程序设计 82
5.1 设计CDL脚本文件 82
5.2 设计头文件 84
5.3 设计源程序文件 85
5.4 向eCos数据库中添加JoyStick驱动程序组件包 94
5.5 用户测试程序 96
6 SD卡驱动程序设计 99
6.1 建立SD卡驱动程序目录结构 99
6.2 建立SD卡驱动程序文件 99

6.3 设计CDL脚本文件 100
6.4 编写SD卡设备表入口 102
6.5 实现SD卡设备接口函数 102
6.6 SD卡驱动程序范例 111
6.7 向eCos数据库中添加SD卡驱动程序组件包 111
6.8 用户测试程序 112
6.9 使用FAT文件系统访问SD卡 114
7 附录 121
7.1 SPCE3200 在eCos系统上的设备名列表 121
7.2 eCos中各设备的key值列表（SPCE3200 平台） 121

1 eCos设备驱动程序概述

1.1 设备驱动程序基础知识

1.1.1 概述

设备驱动程序是介于硬件和 eCos 系统内核之间的软件接口，是一种低级的、专用于某一硬件的软件组件。

在 eCos 系统中，设备驱动程序是以包（Package）的形式存在的，当需要使用该设备时，通过 eCos 环境配置工具把该设备驱动程序包加载到内核中，编译后生成一个库文件，用户使用时，只需要打开该设备，直接调用接口 API 函数即可。

这种以包的形式存在的驱动程序的优点是当需要使用一个硬件设备的时候，只需要把该设备的驱动程序包加载并编译到内核中，如果不需要使用硬件设备，则不需要加载该硬件驱动程序包，这样使得用户使用比较灵活，并可以减少内核占用空间。

eCos 的设备驱动程序支持分层结构，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功能和特性。例如 Touch Panel 可以作为 ADC 的上层设备，它提供的功能是 ADC 不具备的。

本书主要介绍 SPCE3200 硬件设备驱动程序的编写方法和驱动程序的简单使用示例。

1.1.2 驱动程序分类

在 eCos 系统中一般把设备分为 4 类：字符设备、块设备、网络接口和其他设备。在 4 类设备中，eCos 看待设备的方式有所区别，每种方式是为了实现不同的任务。相应地，设备驱动程序也分为 4 类。下面分别介绍这 4 类设备。

1、字符设备

字符设备是指存取时没有缓存的设备，可像文件一样访问字符设备，字符设备驱动程序负责实现这些行为。这样的驱动程序通常会实现 lookup、read、write 系统调用，也会实现 open、read、write 的文件系统调用。SPCE3200 的通用 I/O 口和串口就是字符设备的例子。通过文件系统节点可以访问字符设备，例如/dev/gpio 和/dev/spi。字符设备和普通文件系统间的唯一区别是，普通文件允许在其上来回读/写，而大多数字符设备仅仅是数据通道，只能顺序读/写。当然，也存在这样的字符设备，看起来像一个数据区，可来回读取其中的数据。

2、块设备

块设备是文件系统的宿主，如 SD 卡，Nand 型 Flash、磁盘等。在大多数 Unix 系统中，只能将块设备看作多个块进行访问。一个块设备通常是 1KB 数据。eCos 允许像字符设备那样读取块设备——允许一次传输任意数目的字节。块设备与字符设备只在内核内部的管理上有所区别，因此也就是在内核/驱动程序间的软件接口上有所区别。正如字符设备一样，每个块设备也通过文件系统节点来读/写数据。块设备驱动程序和内核的接口与字符设备驱动程序的接口类似，通过一个传统的面向块的接口与内核通信。

3、网络接口设备

eCos 在网络方面提供了非常强有力的支持，它所包含的公共网络协议包（Common NetWorking Package）支持完整的 TCP/IP 协议，提供了基于 OpenBSD 和 FreeBSD 的两种实现。eCos 支持的服务包括 FTP、TFTP、SNMP、DNS、HTTP 等。

eCos 源码中虽然提供了许多以太网驱动程序，但是由于在具体开发中可能会用到各式各样的网络芯片，针对这些网络芯片编写的驱动程序叫做网络设备驱动，这些芯片叫做网络接口设备。

网络接口设备不同于字符设备或者块设备，它们专用于对以太网各种协议的支持。一般来说，网络接口设备本身是一种上层设备。

4、其他设备

SPCE3200 上还有一些硬件模组不属于 eCos 的以上 3 类设备，例如 LCD、MPEG4 等，这些设备需要专门的接口函数来完成它们的功能，而不能像字符设备和块设备那样简单地进行读写。

本书主要介绍字符设备和块设备驱动程序的编写，同时介绍基于设备的上层设备的驱动程序的编写。网络接口和其他设备（如 LCD）驱动程序将在下一本书中详细介绍。

1.2 加载和移除设备驱动程序包

在编译内核前，需要先加载设备的驱动程序包，前面已经介绍，当需要使用设备时才需要加载设备驱动程序包，不使用时可以不加载；如果已经加载了一个不使用的设备驱动程序包，可以移除掉该设备驱动程序包，以减少内核占用空间。

注意：如果没有特别说明，本书中的加载和移除设备驱动程序包中指把数据库中已经存在的驱动程序包加载参与编译，或者移除不参与编译。请注意不要和加载设备驱动程序包到数据库中混淆。

1.2.1 加载设备驱动程序包

加载设备驱动程序包的方法有两种：一种办法是通过ecos.db文件加载，另一种办法是通过eCos的配置工具来加载；通过ecos.db加载的方法将在1.4.1节详细介绍，这里只介绍通过eCos配置工具加载的办法，本书的介绍都是SPCE3200 硬件开发平台，如图 1.1。

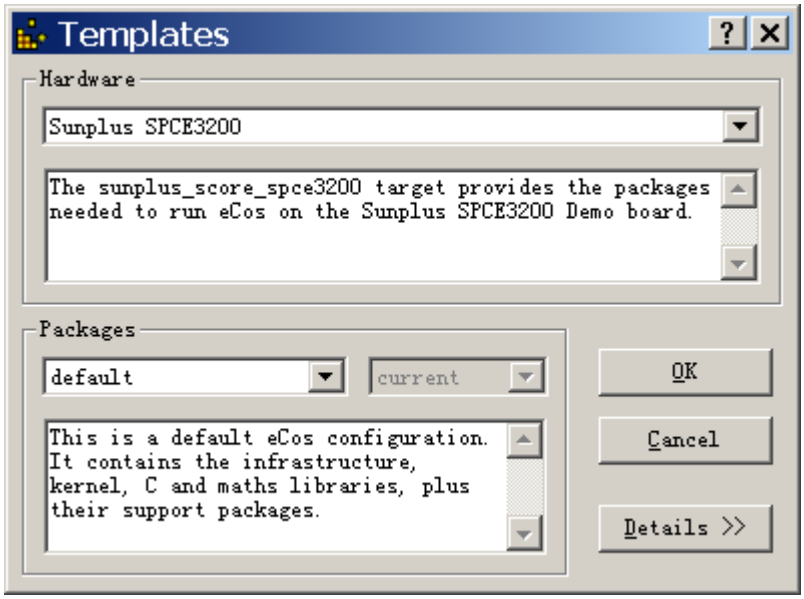

图 1.1 选择 SPCE3200 开发平台

加载设备驱动程序包的前提是，必须已经编写好设备驱动程序，并且已经存在该设备驱动程序包，如图 1.2，找到需要加载的设备驱动程序包（这里选择设备dev1 的驱动程序包），点击“Add”，再点击“OK”就可以加载到内核中，加载后eCos配置工具界面如图 1.3，编译后即可以用该设备的驱动程序。

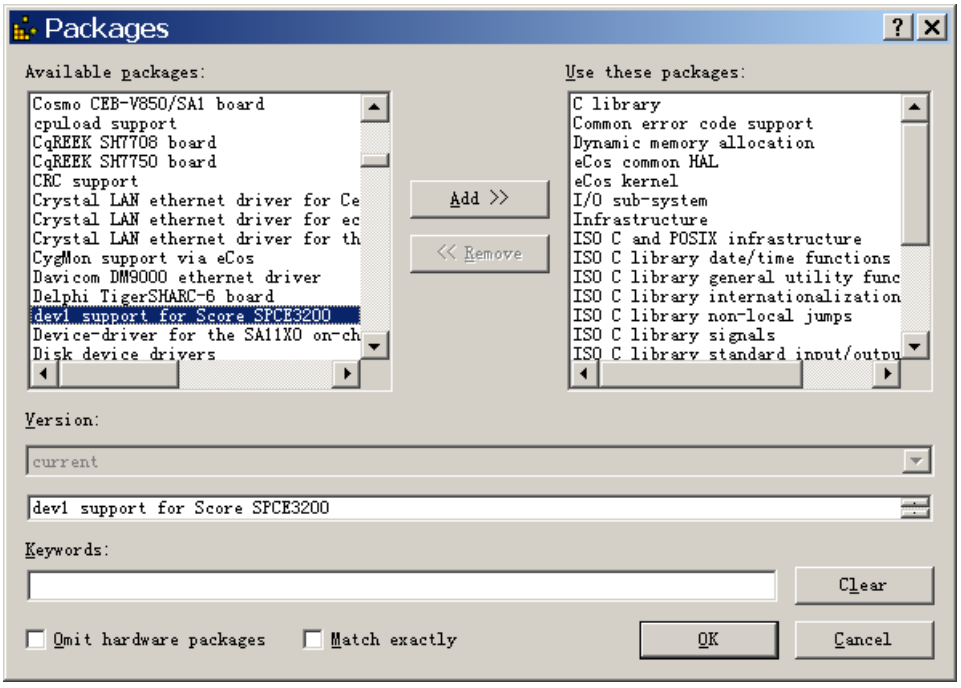

图 1.2 加载设备 dev1 驱动程序包

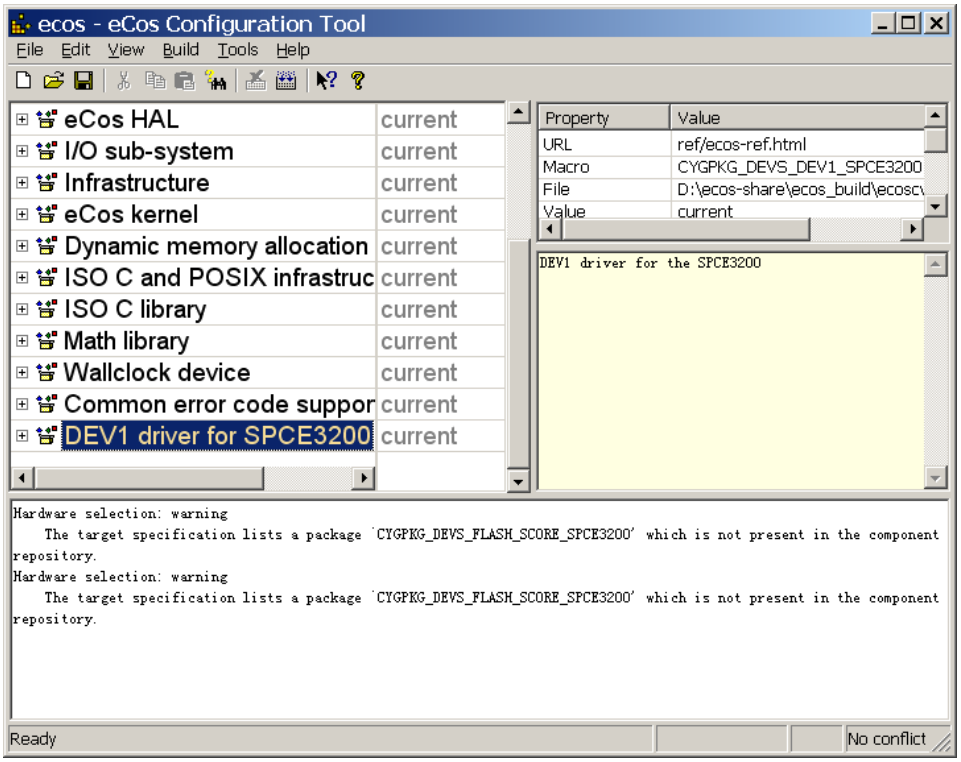

图 1.3 加载设备 dev1 驱动程序包后 eCos 界面

1.2.2 移除设备驱动程序包

移除设备驱动程序包的方法同样有两个：一种办法是通过ecos.db文件移除，另一种办法是通过eCos的配置工具来移除；通过ecos.db移除的方法将在1.4.1节详细介绍，这里只介绍通过eCos配置工具移除的办法。

移除设备驱动程序包的前提是，已经加载了该设备驱动程序包。如图 1.2，找到需要移除的设备驱动程序包（这里选择设备dev1 的驱动程序包），点击“Remove”，再点击“OK”就可以把该设备驱动程序包从参与编译的包中移除。

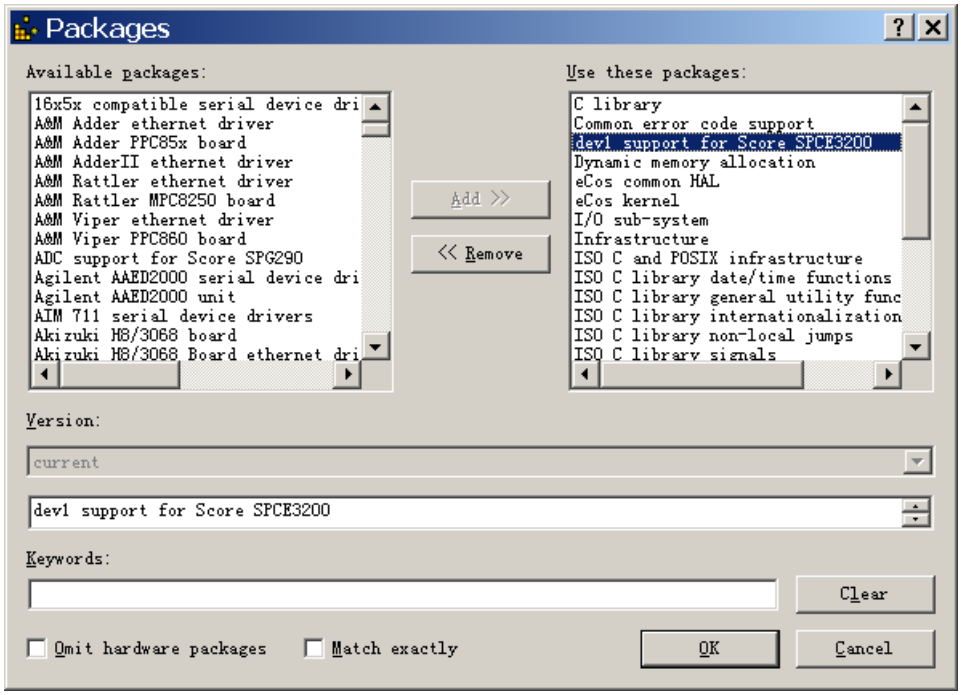

图 1.4 移除设备 dev1 驱动程序包

1.3 cdl脚本

前面介绍，eCos 的设备驱动程序一般是以包的形式存在。而 eCos 中所有的包都必须具有至少一个 cdl 脚本对该包进行描述。这种 cdl 脚本文件中包含了该包中所有配置选项的详细信息，并且提供了如何对该包进行编译的信息。同样，当编写一个设备的驱动程序的时候，必须编写相应的 cdl 脚本文件。

1.3.1 cdl命令简介

在 cdl 脚本文件中可能存在 4 种 cdl 命令：cdl_package，cdl_component，cdl_option 和 cdl_interface。下面逐一介绍：

1、cdl_option 命令。

```
cdl_option name{
……
}
```

选项名

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。每个选项一般都对应于用户的一个选择。

2、cdl_component 命令。

```
cdl_component name{
……
```

组件名

```
}
```

cdl_component 命令定义一个组件（component），是一组配置选项的集合。组件是一个包含其它选项和子组件的一个配置选项。

3、cdl_package 命令。

```
cdl_package name{
……
}
```

包名

cdl_package 命令定义一个包，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，并且可以选择加载那个版本的包。包同时也是一个组件，它以层次结构的形式包含其它的组件和选项。

4、cdl_interface 命令。

```
cdl_interface name{
……
}
```

接口名

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。

1.3.2 cdl脚本基本结构

通常，cdl 脚本的的所有组件、选项和接口都嵌套定义在 cdl_package 命令体中，如下：

```
cdl_package CYGPKG_PACKAGE_NAME {
    …
    cdl_component CYGPKG_COMPONENT1_NAME {
        …
        cdl_option CYGPKG_OPTION11_NAME {
            …
        }
        cdl_option CYGPKG_OPTION12_NAME {
            …
        }
        …
```

```
        }
        cdl_component CYGPKG_COMPONENT2_NAME {
            …
            cdl_option CYGPKG_OPTION21_NAME {
                …
            }
            cdl_option CYGPKG_OPTION22_NAME {
                …
            }
            …
        }
        cdl_option CYGPKG_OPTION_NAME {
            …
        }
        …
    }
```

带边框的文字为包、组件或者选项名，用户可以自行定义。

也可以把组件、选项和接口定义在 cdl_package 命令体外，定义在外面和嵌套在 cdl_package 命令体中的效果相同，但是，当 cdl_package 命令体内有组件、选项或接口的定义，体外也有定义，那么体内的组件、选项或接口会最先得到处理。

1.3.3 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。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。

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 属性只与组件有关；define_header、hardware、include_dir、include_files 和 library 等属性只适用于包；calculated、default_value 和 flavor 等属性与包和接口没有关系；legal_value 属性与包无关等。

根据所服务的不同目的把 cdl 属性分为 6 类：信息类属性、配置层次属性、值相关属性、配置头文件相关属性、编译控制类属性和其它属性。信息类属性包括 display、description 和 doc 属性；配置层次属性包括 parent 和 script 属性；值相关属性包括 flavor、calculated、default_value、legal_values、

active_if、implements、requires 属性；配置头文件相关属性包括 define_header、no_define、define_format、define、if_define 和 define_proc 属性；编译控制类属性包括 compile、make、make_object、library、include_dir 和 include_files 属性；其它属性只指 hardware 属性。限于篇幅，这里不对这些属性逐一介绍，请读者参考 eCos 相关书籍。

属性典型应用参考如下解析：

```
cdl_package CYGPKG_PACKAGE_NAME {
    display       "…"
    include_dir   …

    requires      …
    compile       …

    description "…"

    cdl_component CYGPKG_COMPONENT_NAME {
        display   "…"
        flavor    …
        no_define

        cdl_option CYGPKG_OPTION_NAME {
            display          "…"
            flavor           data
            no_define
            default_value { “…” }
            description "…"
        }
    }
}
```

带有边框的属性功能如下：

display 属性：定义了在 eCos 配置工具的配置界面中显示的名称；

include_dir 属性：指定了源代码中的头文件（include 文件夹中的文件）在编译时的保存路径，应用程序可以使用这个路径引用驱动程序的头文件；

requires 属性：规定了包的依赖关系；

compile 属性：指定了参与编译的源文件集合，一般驱动源程序文件都需要参与此编译；

description 属性：定义的字符串在 eCos 配置工具的配置界面中做为包的详细注释存在。

1.3.4 驱动程序包的脚本cdl文件

在驱动程序包的脚本 cdl 文件中，需要描述如下几方面内容：

1、用 cdl_package 命令定义驱动程序包，驱动程序包中含 display、include_dir、compile、description 等属性。

2、用 cdl_component 命令描述驱动程序包中的组件。

3、用 cdl_option 命令描述驱动程序包或者组件中的选项。

4、描述其他内容。

注意：包中有多少组件、多少选项以及组件和选项的内容是什么可以由驱动程序编写人员自行定义。

下面为设备 dev1 的典型脚本 cdl 文件：

```
cdl_package CYGPKG_DEVS_DEV1_SPCE3200 {
    display         " dev1 driver for SPCE3200"
    include_dir     cyg/io
    requires    CYGPKG_IO_DISK
    requires        CYGPKG_IO
    requires        CYGFUN_KERNEL_API_C

    compile             -library=libextras.a dev1_spce3200.c

    description " dev1 driver for SCORE SPCE3200"

    cdl_component CYGPKG_DEVS_ DEV1_SPCE3200_OPTIONS {
        display "Compile Options"
        flavor    none
        no_define

        cdl_option CYGPKG_DEVS_ DEV1_SPCE3200_XXX {
            …
        }
```

包名

包的属性

组件名

```
            …
        }
        …
    }
```

1.4 eCos库的数据库文件ecos.db

配置工具使用 eCos 库的数据库来理解库中的组件。库的数据库中包的描述包含在文件 ecos.db 中，该文件在源码 ecos\packages 子目录中。ecos.db 文件使用 cdl 描述所有包和数据库中的目标硬件。

1.4.1 加载驱动程序包到数据库

一个设备驱动程序包在用 cdl 脚本对其进行描述后，必须加载到 eCos 库的数据库中才能正常使用，这时候需要通过 ecos.db 文件对该设备驱动程序包进行描述。

在写好一个驱动程序包的 cdl 脚本后，添加该包到数据库中的常见方法如下：

1、定义包：用 package 命令定义设备驱动程序包，其包名必须与 cdl 文件中 cdl_package 定义的包名相同。

2、定义包体：在包体内用 alias 属性描述该包的功能，此属性描述的内容会显示在 eCos 库的数据库里；另外用 directory 属性指定该包的 cdl 脚本文件所在路径；用 script 属性说明该包的 cdl 脚本文件名；最后用 description 属性描述该包的详细功能。

假设有一个设备 dev1 的驱动程序包，其 cdl 脚本文件名为 dev1_spce3200.cdl，其典型的 ecos.db 中包的描述如下：

```
package CYGPKG_DEVS_DEV1_SPCE3200 {
    alias          { "dev1support for Score SPCE3200"}
    directory      devs/dev1/score/spce3200
    script         dev1_spce3200.cdl
        description "
            This package contains hardware support for the dev1
        on the Score SPCE3200 DEV Board."
}
```

CYGPKG_DEVS_DEV1_SPCE3200 为设备dev1 在驱动cdl文件中已经定义好的cdl包名；alias是对于这个包的功能简要概述，作为dev1 设备的驱动程序名，显示在eCos配置工具的eCos数据库中，如图 1.5；directory指定dev1 设备驱动程序的源路径；script指定驱动程序cdl文件的文件名，在对内核编译的时候，编译工具通过directory指定的路径和script指定的cdl文件名找到设备驱动程序的cdl文件，通过cdl文件描述的内容把设备驱动程序编译到内核中并生成库；description是对dev1 设备驱动程序的简单描述。

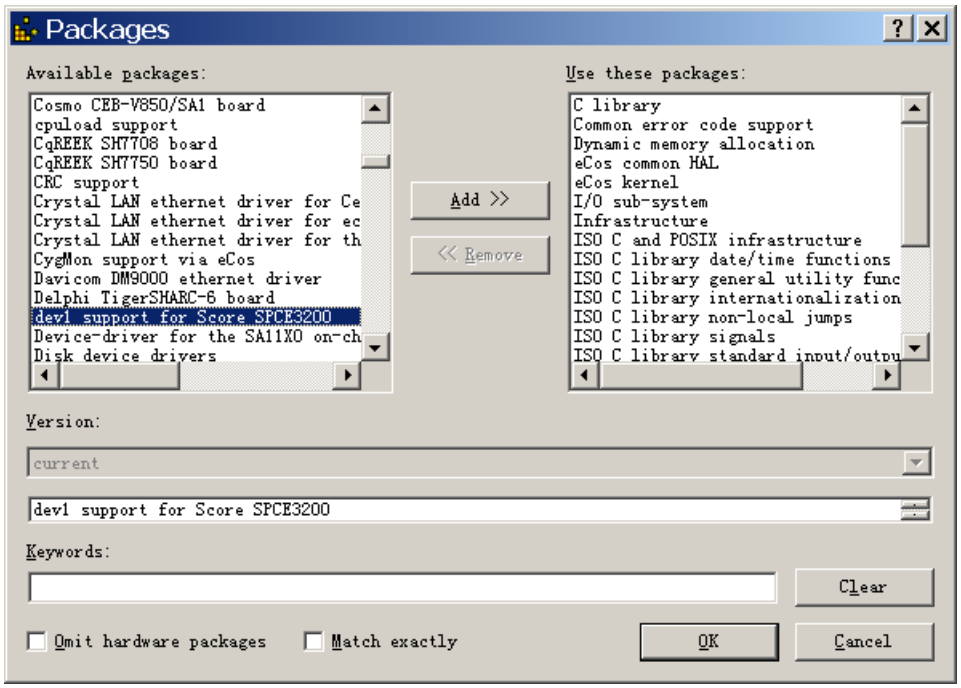

图 1.5 ecos.db 中加入 CYGPKG_DEVS_DEV1_SPCE3200 包

如图 1.5，经过上述描述后，在eCos的数据库中就会存在dev1 的驱动程序包CYGPKG_DEVS_DEV1_SPCE3200。

1.4.2 在ecos.db文件中加载和删除驱动程序包

在介绍加载和移除设备驱动程序包时提到，除了在配置工具中加载和移除外，还可以在 ecos.db 文件中进行加载或者移除。

当 eCos 系统移植到一个硬件平台上后，在 ecos.db 文件中一定会有硬件平台的描述，例如把 eCos 系统移植到 SPCE3200 平台上后，会有如下描述：

```
target sunplus_score_spce3200 {
        alias       { "Sunplus SPCE3200" }
    packages            { CYGPKG_HAL_SCORE
                              CYGPKG_HAL_SCORE_SPCE3200
                              CYGPKG_DEVS_FLASH_SCORE_SPCE3200
                              CYGPKG_DEVS_FLASH_SST_39VFXXX
        }
        description "
            The sunplus_score_spce3200 target provides the packages needed to run
            eCos on the Sunplus SPCE3200 Demo board. "
```

```
}
```

在上面的描述中，packages 属性中描述的是 SPCE3200 平台上已经加载参与编译的包，如果在该平台上加载一个数据库中已经存在的驱动程序包，只需要把数据库中 package 定义的包加到 packages 属性内即可。例如把 dev1 的驱动程序包 CYGPKG_DEVS_DEV1_SPCE3200 加载到 SPCE3200 平台中的平台描述如下：

```
target sunplus_score_spce3200 {
        alias       { "Sunplus SPCE3200" }
    packages            { CYGPKG_HAL_SCORE
                              CYGPKG_HAL_SCORE_SPCE3200
                              CYGPKG_DEVS_FLASH_SCORE_SPCE3200
                              CYGPKG_DEVS_FLASH_SST_39VFXXX
                              CYGPKG_DEVS_DEV1_SPCE3200
        }
        description "
            The sunplus_score_spce3200 target provides the packages needed to run
            eCos on the Sunplus SPCE3200 Demo board. "
}
```

CYGPKG_DEVS_DEV1_SPCE3200 为加载的dev1 设备驱动程序包。此时加载的效果和图 1.3相同。

如果要删除，和上面的操作相反，只需要把 CYGPKG_DEVS_DEV1_SPCE3200 从 packages 属性中删除即可。

1.5 编写设备驱动的一般步骤

编写设备驱动的一般步骤如图 1.6：在eCos源码设备ecos\packages\devs目录下建立设备驱动程序目录结构；建立cdl文件和驱动源程序文件（一般为C文件），必要时建立头文件；编写cdl脚本文件，建立设备驱动程序组件包，定义配置选项；编写驱动源程序文件，编写其中的设备表入口、驱动程序接口函数等，必要时编写其头文件；把设备驱动程序组件包加载到eCos数据库中；在数据库中找到该设备驱动程序组件包，加载到开发平台；编译生成*.ecc库文件；编写测试程序测试驱动程序。

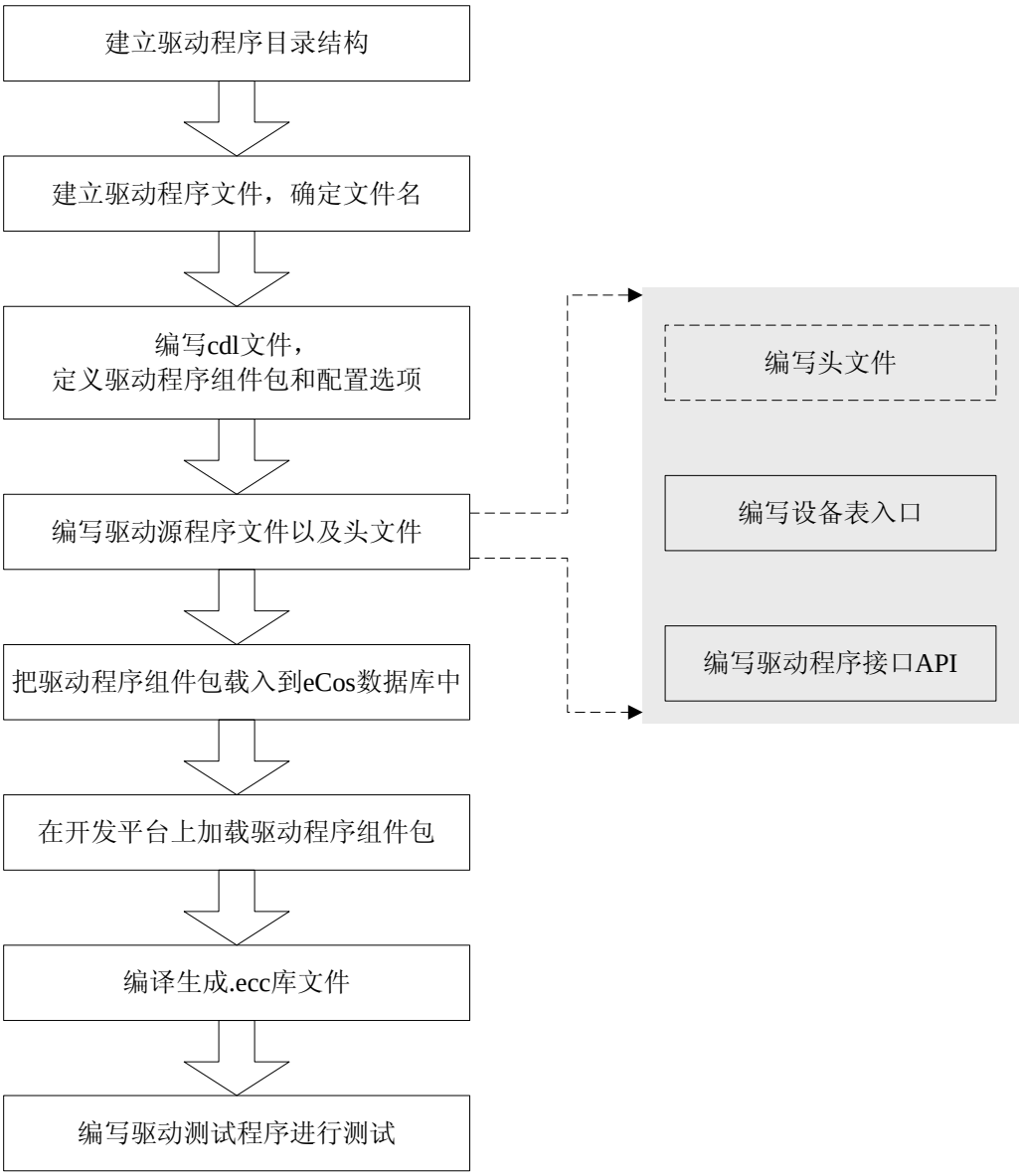

图 1.6 编写设备驱动的一般步骤

2 字符设备驱动程序初步

2.1 字符设备驱动程序概述

字符设备是指存取时没有缓存的设备。典型的字符设备包括鼠标、键盘、串行口等。在一个实际的系统中，往往是字符设备占多数。

eCos 使用设备标示来辨识字符设备。每个在内核中活动的字符设备驱动程序，都有唯一的设备标示。内核利用设备标示将设备与相应的驱动程序对应起来。因此，要向系统增加一个设备驱动程序意味着必须要赋予它一个设备标示。一旦设备注册到内核表中，则无论何时操作与设备驱动程序的设备标示匹配的设备文件，内核都会调用驱动程序中的正确函数。

注意：块设备也有设备标示，它与字符设备的设备标示相互独立。

2.2 字符设备驱动程序的目录结构

根据图 1.6，在编写设备驱动程序时，先建立目录结构。

字符设备驱动程序的目录结构如图 2.1所示。这里假设设备名为dev1。dev1 的驱动程序放置在/devs目录下的dev1 目录里，在S+core7 内核SPCE3200 芯片对应的目录下，存在cdl、include和src三个目录，分别保存着dev1 驱动的cdl文件、驱动程序头文件和驱动源程序文件。其中，include文件夹内的文件将会被复制到编译目录（编译目录路径由cdl文件中的include_dir属性定义）内，可以被应用程序包含并引用。

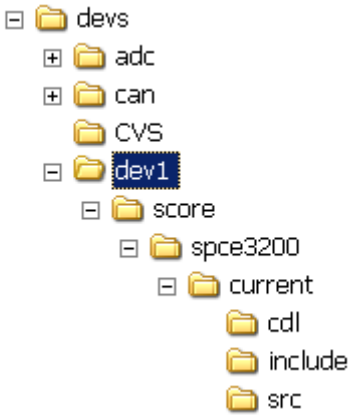

图 2.1 dev1 设备驱动程序的目录结构

2.3 字符设备驱动程序的基本结构

图 2.2为字符设备驱动程序结构及其在eCos系统中的位置示意图。应用程序在使用设备时，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。

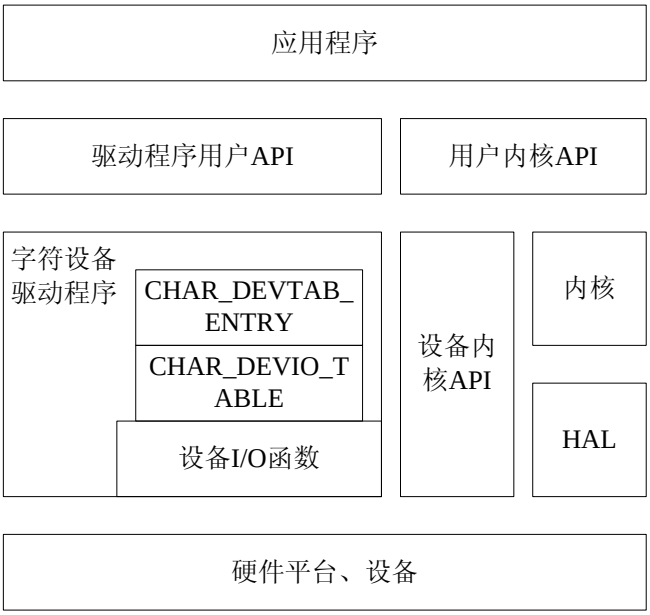

图 2.2 字符设备驱动程序的基本结构

字符设备驱动程序一般分为三个部分：设备表入口 CHAR_DEVTAB_ENTRY、设备 I/O 函数表 CHAR_DEVIO_TABLE 和设备 I/O 函数（下文称为设备接口函数），设备驱动程序的主要组成模块全部定义在编译目录的 cyg/io 路径下的 devtab.h 文件中。

2.4 字符设备驱动程序设计

字符设备驱动程序包括以下几个基本步骤：设计 CDL 文件、编写设备表入口、实现设备接口函数、向组件库添加驱动程序包等。

下面分别介绍字符设备驱动程序的设计步骤。

2.4.1 设计CDL文件

CDL 文件为设备定义了一系列的编译属性和行为，并可以根据用户的需要对编译进行源码级控制。

字符设备一个典型的 CDL 文件的写法如下所示：首先要建立一个设备驱动程序组件包，在包体内定义各种配置选项和组件。

```
cdl_package CYGPKG_DEVS_DEV1_SPCE3200 {
    display       " dev1 driver for SPCE3200"
    include_dir   cyg/io
    requires      CYGPKG_IO

    compile           -library=libextras.a dev1_spce3200.c
```

```
description " dev1 driver for SCORE SPCE3200"

cdl_component CYGPKG_DEVS_ DEV1_SPCE3200_OPTIONS {
    display "Compile Options"
    flavor  none
    no_define

    cdl_option CYGPKG_DEVS_ DEV1_SPCE3200_CFLAGS {
        display         "Additional compiler flags"
        flavor          data
        no_define
        default_value { "" }
        description "
            This option modifies the set of compiler flags for
            building the keypad driver package. These flags
            are used in addition to the set of global flags."
    }

    cdl_option CYGDAT_DEVS_ DEV1_SPCE3200_NAME {
        display "Device name for the dev1 driver"
        flavor data
        default_value {"\"/dev/dev1\""}
        description " This option specifies the name of the dev1 device"
    }

    cdl_option CYGNUM_DEVS_ DEV1_SPCE3200_DEBUG_MODE {
        display "Debug Message"
        default_value   0
        description "
            This option will enable the debug message outputing if set to 1,
            will disable the outputing if set to 0."
```

```
            }
        }
    }
```

上面的代码中，cdl_package 命令定义了一个叫做 CYGPKG_DEVS_DEV1_SPCE3200 的包，在 ecos.db 中需要使用此名称来将包添加至数据库，以便在 eCos 配置工具中作为一个组件进行配置。

2.4.2 编写字符设备表入口

使用 CHAR_DEVTAB_ENTRY 宏可以描述一个字符设备，该宏的完整格式为：

```
CHAR_DEVTAB_ENTRY(_l, _name, _dep_name, _handlers, _init, _lookup, _priv);
```

其中：

- _l：设备表入口的标识符，即本设备表的句柄
- _name：设备的名称，一般是以“/dev/”开头的字符串，如“/dev/dev1”
- _dep_name：对于层次设备，此参数为该设备所依赖的底层设备的名称
- _handlers：设备的 I/O 函数表的句柄指针
- _init：设备的初始化函数，当 eCos 处于初始化阶段时将调用此函数以便对该设备进行初始化。用户需要编写该函数以便对设备进行初始化，或完成一些针对该设备的预处理工作。_init 为初始化函数名，用户可以自定义该函数名，习惯上该函数名定义为设备名+‘_’+init 的格式，比如某一设备的设备名为 dev1，该函数名习惯上定义为 dev1_init
- _lookup：设备的查找函数，当使用 cyg_io_lookup()函数对该设备进行查找时将调用此函数。用户需要编写该函数以便在应用程序尝试查找该设备时作出相应处理。_lookup 同样为查找函数的函数名，也可自定义，设备名为 dev1 的设备查找函数名习惯上定义为 dev1_lookup。
- _priv：该设备驱动程序所需的专用数据的存放位置

设备表入口中的句柄_handlers 提供了一组字符设备驱动程序接口函数。_handlers 是字符设备 I/O 操作函数表 CHAR_DEVIO_TABLE 的指针，CHAR_DEVIO_TABLE 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，也是应用程序访问字符设备的接口函数。

字符设备 I/O 函数表通过 CHAR_DEVIO_TABLE 宏来定义，其格式如下：

```
CHAR_DEVIO_TABLE(_l,_write,_read,_select,_get_config,_set_config)
```

其中：

- _l：该设备 I/O 函数表的标识符，即本函数表的句柄
- _write：cyg_io_write()函数所调用的函数，实现向字符设备传送数据。用户一般需要编写该函数，完成数据的写操作。与_init 类似，_write 也为函数名，用户可以自定义

- _read：cyg_io_read()函数所调用的函数，实现从字符设备读取数据。用户一般需要编写该函数，完成数据的读操作。与_init 类似，_read 为函数名，用户可以自定义
- _select：cyg_io_select()函数所调用的函数。与_init 类似，_select 为函数名，用户可以自定义
- _get_config：cyg_io_get_config()函数所调用的函数，完成对设备设置信息的读取操作。用户可以编写该函数，用以向应用程序返回特定信息。与_init 类似，_get_config 为函数名，用户可以自定义
- _set_config：cyg_io_set_config()函数所调用的函数，完成对设备的设置操作。用户可以编写该函数，用以接受应用程序的控制。与_init 类似，_set_config 为函数名，用户可以自定义

设备表入口以及设备I/O函数表通常与设备接口函数的实现代码放在同一个文件（也称源程序文件，通常放在图 2.1的src文件夹里），在介绍接口函数的实现时将举例说明。

2.4.3 实现字符设备接口函数

在设备表入口以及设备 I/O 函数表中都包含了一组函数的指针，这些函数就是设备驱动接口函数。在完成设备表入口以及设备 I/O 函数表之后，用户需要逐个实现这些函数，包括：

1. 设备初始化函数

字符设备初始化函数在内核启动期间被调用，用以对设备进行初始化操作。设备初始化函数的完整形式为：

bool 函数名(cyg_devtab_entry *tab);

该函数由用户编写，主要实现设备硬件的设置、使能操作。

其中：

- 函数名应与 CHAR_DEVTAB _ENTRY 宏中定义的初始化函数名一致，从而内核可以依靠设备表入口找到这个函数，在内核启动的过程中对该设备进行初始化操作，比如 CHAR_DEVTAB_ENTRY 中的初始化名定义为：dev1_init，此时的初始化函数可以定义为如下伪代码形式：

```
static bool dev1_init(struct cyg_devtab_entry *tab)
{
    函数体;
}
```

- tab：指向当前设备的指针，用户可以通过该指针对当前设备进行查找操作

2. 设备查找函数

当应用程序使用 cyg_io_lookup()函数打开设备时，内核将调用该函数。在该函数中用户一般需要建立线程以便对设备进行查询读写操作，或开启中断服务程序来操作设备。对于层次设备来说，在该函数取得本设备所依赖的底层设备的操作句柄指针，并做一些打开设备时的操作。设备查找函数的完整形式为：

Cyg_ErrNo 函数名(struct cyg_devtab_entry **tab, struct cyg_devtab_entry *sub_tab, const char *name);

其中：

- 函数名应与 CHAR_DEVTAB_ENTRY 宏中定义的查找函数名一致，从而内核可以依靠设备表入口找到这个函数，比如 CHAR_DEVTAB_ENTRY 中的查找函数名定义为：dev1_lookup，此时的查找函数可以定义为如下伪代码形式：

```
static Cyg_ErrNo dev1_lookup (struct cyg_devtab_entry **tab,
                              struct cyg_devtab_entry *st,
                              const char *name)
{
    函数体；
}
```

- tab：指向当前设备的指针，用户可以通过该指针对当前设备的信息进行读写操作
- sub_tab：指向当前设备所依赖的下层设备的指针，用户可以通过该指针对下层设备进行操作。在 lookup 设备的时候，如果有下层设备，cyg_io_lookup()函数中会同时打开下层设备并通过参数 sub_tab 返回该设备的指针
- name：应用程序希望查找的设备名比当前设备名多出的部分。譬如，应用层希望打开“/dev/disk0”，而系统中只有设备“/dev/disk”，内核会将此设备做为最佳匹配设备打开，并将名称中多出的“0”字符串传递给该函数

3. 设备写操作函数

当应用程序使用 cyg_io_write()函数对设备进行写操作时，内核将调用该函数。在该函数中用户需要实现对设备的写操作。设备写操作函数的完整形式为：

Cyg_ErrNo 函数名(cyg_io_handle_t handle,void *buffer,cyg_uint32 *len);

其中：

- 函数名应与 CHAR_DEVIO_TABLE 宏中定义的写操作函数名一致，从而内核可以依靠 I/O 函数表找到这个函数，比如 CHAR_DEVIO_TABLE 中的写函数名定义为：dev1_write，此时的写函数可以定义为如下伪代码形式：

```
static Cyg_ErrNo dev1_write(cyg_io_handle_t handle,void *buffer,cyg_uint32 *len)
{
    函数体;
}
```

- handle：应用程序操作设备的设备句柄
- buf：缓冲区指针，用户需要从该指针指向的缓冲区内读取数据并写入到设备
- len：写入到设备的数据长度。函数返回时该参数包含实际写入的数据大小

4. 设备读操作函数

当应用程序使用 cyg_io_read()函数对设备进行读操作时，内核将调用该函数。在该函数中用户需要实现对设备的读操作。设备读操作的完整形式为：

Cyg_ErrNo 函数名(cyg_io_handle_t handle, void *buf, cyg_uint32 *len);

其中：

- 函数名应与 CHAR_DEVIO_TABLE 宏中定义的读操作函数名一致，从而内核可以依靠 I/O 函数表找到这个函数，比如 CHAR_DEVIO_TABLE 中的写函数名定义为：dev1_read，此时的写函数可以定义为如下伪代码形式：

```
static Cyg_ErrNo dev1_read(cyg_io_handle_t handle,void *buffer,cyg_uint32 *len)
{
    函数体;
}
```

- handle：应用程序操作设备的设备句柄
- buf：缓冲区指针，用户需要从设备中读取数据并将数据写入该指针指向的缓冲区内
- len：从设备读取的数据长度。函数返回时该参数包含实际读取的数据大小

5. 设备检测函数

当应用程序使用 cyg_io_select()函数，内核将调用该函数进行设备的可用性检测。设备选择函数的完整形式为：

cyg_bool 函数名(cyg_io_handle_t handle, cyg_uint32 which, CYG_ADDRWORD info);

- 函数名应与 CHAR_DEVIO_TABLE 宏中定义的选择函数名一致，从而内核可以依靠 I/O 函数表找到这个函数，比如 CHAR_DEVIO_TABLE 中的写函数名定义为：dev1_select，此时

的写函数可以定义为如下伪代码形式：

```
static cyg_bool dev1_select(cyg_io_handle_t handle, cyg_uint32 which, CYG_ADDRWORD info)
{
    函数体；
}
```

- handle：应用程序操作设备的设备句柄
- which：选择类型，一般在利用文件系统操作设备时，which 为文件类型只读、只写等。
- info：选择信息，例如等待标志等选择信息。

6. 读设备配置状态函数

当应用程序使用 cyg_io_get_config()函数时，内核将调用该函数。在该函数中用户一般需要实现为应用程序提供与设备相关配置信息的功能。读设备配置函数的完整形式为：

Cyg_ErrNo 函数名(cyg_io_handle_t handle, cyg_uint32 key, void *buf, cyg_uint32 *len);

其中：

- 函数名应与 CHAR_DEVIO_TABLE 宏中定义的读设备配置函数名一致，从而内核可以依靠 I/O 函数表找到这个函数，比如 CHAR_DEVIO_TABLE 中的读设备配置函数名定义为：dev1_get_config，此时的读设备函数可以定义为如下伪代码形式：

```
static Cyg_ErrNo dev1_ get_config (    cyg_io_handle_t handle,
                                       cyg_uint32 key,
                                       void *buf,
                                       cyg_uint32 *len    )
{
    函数体；
}
```

- handle：应用程序操作设备的设备句柄
- key：得到信息的类型。每个驱动程序的 key 值可能不同，在源码 io/common/include 路径下的 config_keys.h 文件中定义了一些常用的 key 值。也可以由用户在头文件中定义 key 值
- buf：放置配置数据的缓冲区指针
- len：得到的配置信息的长度。函数返回时，该参数应该包含实际得到的数据大小

7. 设备配置函数

当应用程序使用 cyg_io_set_config()函数时，内核将调用该函数。在该函数中用户一般需要实现根据应用程序提供的参数对设备进行配置的功能。写设备配置函数的完整形式为：

Cyg_ErrNo 函数名(cyg_io_handle_t handle, cyg_uint32 key, const void *buf, cyg_uint32 *len);

其中：

- 函数名应与 CHAR_DEVIO_TABLE 宏中定义的写设备配置函数名一致，从而内核可以依靠 I/O 函数表找到这个函数，比如 CHAR_DEVIO_TABLE 中的写设备配置函数名定义为：dev1_ set_config，此时的写设备函数可以定义为如下伪代码形式：

```
static Cyg_ErrNo dev1_ set_config (    cyg_io_handle_t handle,
                                       cyg_uint32 key,
                                       void *buf,
                                       cyg_uint32 *len    )
{
    函数体;
}
```

- handle：应用程序操作设备的设备句柄
- key：设置信息的类型。每个驱动程序的 key 值可能不同，在源码 io/common/include 路径下的 config_keys.h 文件中定义了一些常用的 key 值。也可以由用户在头文件中定义 key 值
- buf：放置配置数据的缓冲区指针
- len：配置信息的长度。函数返回时，该参数应该包含实际设置的数据大小

2.4.4 简单的字符设备驱动程序解析

下面是一个简单的字符设备驱动程序的完整代码，这个代码完整的实现了字符设备驱动源程序的各个部分，包括设备表入口以及设备 I/O 函数表的实现。接下来的块驱动程序设计过程中，可以把它当作模版。这里假设设备为 dev1。

```
#include <pkgconf/hal.h>
#include <pkgconf/devs_dev1_spce3200.h>
#include <cyg/infra/cyg_type.h>
#include <cyg/io/devtab.h>


// ----------------------------------声明接口函数--------------------------------//
```

其中，dev1_spce3200 必须与 dev1 的脚本 cdl 文件相同。该头文件编译时会自动生成，生成了 cdl 文件定义的一些选项常量

```
static  cyg_ErrNo dev1_read(cyg_io_handle_t handle, void *buffer, cyg_uint32 *len);
static  cyg_ErrNo dev1_write(cyg_io_handle_t handle, void *buffer, cyg_uint32 *len);
static  cyg_boolkbd_select(cyg_io_handle_thandle,  cyg_uint32which,  cyg_addrword_t info);
static  cyg_ErrNo dev1_set_config(cyg_io_handle_t handle, cyg_uint32 key, const void *buffer, cyg_uint32 *len);
static  cyg_ErrNo dev1_get_config(cyg_io_handle_t handle, cyg_uint32 key, void *buffer, cyg_uint32 *len);
static  bool dev1_init(struct cyg_devtab_entry *tab);
static  cyg_ErrNo dev1_lookup(struct cyg_devtab_entry **tab, struct cyg_devtab_entry *st, const char *name);

//----------------------------------------------定义设备入口表-----------------------------------------//
CHAR_DEVIO_TABLE(SPCE3200_dev1_handlers,
    dev1_write,
    dev1_read,
    dev1_select,
    dev1_get_config,
    dev1_set_config
);

CHAR_DEVTAB_ENTRY(SPCE3200_ dev1_device,
    CYGDAT_DEVS_ DEV1_SPCE3200_NAME,
    NULL,                    // Base device name
    &SPCE3200_ dev1_handlers,
    dev1_init,
    dev1_lookup,
    NULL                     // Private data pointer
);

//---------------------------------定义接口函数-------------------------------------//
static cyg_ErrNo dev1_read(cyg_io_handle_t handle, void *buf, cyg_uint32 *len)
{
    // Add Read Code here
```

读函数

```
    return ENOERR;
}
static cyg_ErrNo dev1_write(cyg_io_handle_t handle, const void *buf, cyg_uint32 *len)
{
    // Add Write code here

    return ENOERR;
}
static cyg_bool dev1_select(cyg_io_handle_t handle, cyg_uint32 which, cyg_addrword_t info)
{
    // Add Select code here

    return true;
}
static cyg_ErrNo dev1_getconfig(disk_channel* chan, cyg_uint32 key, const void* buf, cyg_uint32*
len)
{
    // Add Get Config code here

    return EINVAL;
}
static Cyg_ErrNo dev1_setconfig(disk_channel* chan, cyg_uint32 key, const void* buf, cyg_uint32*
len)
{
    // Add Set Config code here

    return EINVAL;
}
static bool dev1_init(struct cyg_devtab_entry *tab)
{
    // Add Initial code here
```

写函数

选择函数

读设备配置函数

设备配置函数

设备初始化函数，在内核启动过程中将被调用

```
        return true;
    }
    static Cyg_ErrNo dev1_lookup(struct cyg_devtab_entry **tab, struct cyg_devtab_entry *st, const char
*name)
    {
        // Add Lookup code here


        return ENOERR;
    }
```

查找函数

2.4.5 向组件库添加驱动程序包

写好 CDL 文件后，设备便以包的形式存在，并具有一些编译选项。将这个包添加到 eCos 数据库中，以便 eCos 配置工具可以识别到这个包，并可以将其加入到编译队列中。这里假设在 dev\dev1\score\spce3200 路径下的 cdl 文件里定义了一个叫做 CYGPKG_DEVS_DEV1_SPCE3200 的包，其典型添加包到数据库的代码如下所示：

```
package CYGPKG_DEVS_DEV1_SPCE3200 {
    alias           { "dev1 support for Score SPCE3200"}
    directory       devs/dev1/score/spce3200
    script          dev1_spce3200.cdl
        description "
          This package contains hardware support for the dev1 on the Score SPCE3200 EV Board. "
}
```

设备名

驱动程序源路径

驱动程序 cdl 文件名

上面的代码中，package 命令向组件仓库内添加了一个组件包，组件包的名称为 CYGPKG_DEVS_DEV1_SPCE3200，这个名称应该与驱动程序 CDL 文件中使用 cdl_package 命令定义的包名称一致。

2.5 字符设备驱动程序的使用

设备驱动程序是介于硬件和 eCos 系统内核之间的软件接口，在用户函数中，一般不直接调用驱动程序接口函数，而在前面介绍驱动程序接口函数时介绍过，字符设备驱动函数都会在 eCos 的 I/O 子系统 API 函数中调用，用户使用时只需要调用这些 I/O 子系统 API 函数即可。这就使得不管多么复杂的驱动程序，使用时只需要掌握几个简单的 I/O 子系统 API 函数即可。

调用I/O子系统API函数对设备进行操作的一般流程如图 2.2。

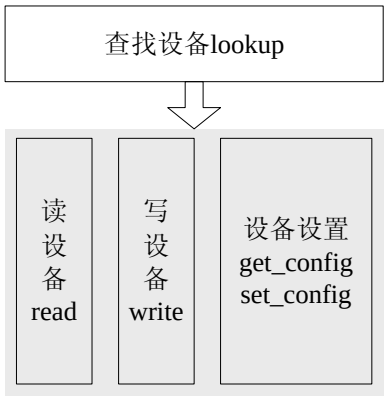

图 2.3 字符设备驱动程序使用流程

eCos 的 I/O 子系统 API 函数主要有以下 5 个：

1. I/O 子系统设备查找函数

当应用程序使用 I/O 子系统设备查找函数 cyg_io_lookup()查找设备时，内核将调用设备驱动程序的设备查找函数。I/O 子系统设备查找函数的完整形式为：

Cyg_ErrNo cyg_io_lookup(const char *name,cyg_io_handle_t *handle);

其中：

- name：应用程序希望查找的设备名，如“/dev/dev1”，在 cyg_io_lookup 调用字符设备驱动程序中的查找设备函数（如 dev1_lookup）时，内核会将此设备名传递给 xxx_lookup 函数作为第一个参数。
- handle：应用程序操作设备的设备句柄。在调用 cyg_io_lookup 时就会建立设备的句柄 handle，在调用其它设备操作函数时，直接可以通过该句柄找到设备。

2. I/O 子系统写函数

当应用程序使用 I/O 子系统写函数 cyg_io_write()对设备进行写操作时，内核将调用设备驱动程序的写函数。I/O 子系统写函数的完整形式为：

Cyg_ErrNo cyg_io_write(cyg_io_handle_t handle,void *buffer,cyg_uint32 *len);

其中：

- handle：cyg_io_lookup 传递过来的参数，应用程序操作设备的设备句柄
- buf：缓冲区指针，用户需要从该指针指向的缓冲区内读取数据并写入到设备
- len：写入到设备的数据长度。函数返回时该参数包含实际写入的数据大小

调用 cyg_io_write 时，cyg_io_write 会根据 cyg_io_lookup 传递过来的设备句柄调用设备驱动写函数，并把三个参数原封不动地传递给设备驱动写函数。

3. I/O 子系统读函数

当应用程序使用 I/O 子系统读函数 cyg_io_read()对设备进行读操作时，内核将调用设备驱动程序的读函数。I/O 子系统读函数的完整形式为：

Cyg_ErrNo cyg_io_read(cyg_io_handle_t handle, void *buf, cyg_uint32 *len);

其中：

- handle：cyg_io_lookup 传递过来的参数，应用程序操作设备的设备句柄
- buf：缓冲区指针，用户需要从设备中读取数据并将数据写入该指针指向的缓冲区内
- len：从设备读取的数据长度。函数返回时该参数包含实际读取的数据大小

调用 cyg_io_read 时，cyg_io_read 会根据 cyg_io_lookup 传递过来的设备句柄调用设备驱动读函数，并把三个参数原封不动地传递给设备驱动读函数。

4. I/O 子系统配置函数

当应用程序使用 I/O 子系统配置函数 cyg_io_set_config()时，内核将调用设备驱动程序的设备配置函数。I/O 子系统配置函数的完整形式为：

```
Cyg_ErrNo cyg_io_set_config (
                cyg_io_handle_t handle,
                cyg_uint32 key,
                const void *buf,
                cyg_uint32 *len
            );
```

其中：

- handle：cyg_io_lookup 传递过来的参数，应用程序操作设备的设备句柄
- key：设置信息的类型。每个驱动程序的 key 值可能不同，在源码 io/common/include 路径下的 config_keys.h 文件中定义了一些常用的 key 值；用户也可以在头文件中自定义 key 值
- buf：放置配置数据的缓冲区指针
- len：配置信息的长度。函数返回时，该参数应该包含实际设置的数据大小

调用 cyg_io_set_config 时，cyg_io_set_config 会根据 cyg_io_lookup 传递过来的设备句柄调用设备驱动配置函数，并把四个参数原封不动地传递给设备驱动配置函数。

5. I/O 子系统读配置状态函数

当应用程序使用 I/O 子系统读配置状态函数 cyg_io_get_config()时，内核将调用设备驱动程序中的读设备配置状态函数。I/O 子系统读配置状态函数的完整形式为：

Cyg_ErrNo cyg_io_get_config(

```
    cyg_io_handle_t handle,
    cyg_uint32 key,
    void *buf,
    cyg_uint32 *len
);
```

其中：

- handle：应用程序操作设备的设备句柄
- key：得到信息的类型。每个驱动程序的 key 值可能不同，在源码 io/common/include 路径下的 config_keys.h 文件中定义了一些常用的 key 值；用户也可以在头文件中自定义 key 值
- buf：放置配置数据的缓冲区指针
- len：得到的配置信息的长度。函数返回时，该参数应该包含实际得到的数据大小

调用 cyg_io_set_config 时，cyg_io_set_config 会根据 cyg_io_lookup 传递过来的设备句柄调用设备驱动读配置状态函数，并把四个参数原封不动地传递给设备驱动读配置状态函数。

在对设备进行读/写或者配置操作的时候，必须先正确查找到设备。如下是一个典型的字符设备驱动用户程序段：（这里假设字符设备为 dev1。）

```
cyg_io_handle_t     hDrvDEV1;
void dev1_read_thread(cyg_addrword_t data)
{
    cyg_uint32          len ;
    cyg_uint8           ReadData[4];

    if (ENOERR != cyg_io_lookup("/dev/dev1", &hDrvDEV1))
    {
        CYG_TEST_FAIL_FINISH("Error opening /dev/dev1");
    }
    printf("Open /dev/dev1 OK\n");
    while(true)
    {
        len = 1;
        if(ENOERR != cyg_io_read(hDrvDEV1, ReadData, &len))
        {
            CYG_TEST_FINISH("Error Read /dev/dev1");
```

在线程中对设备进行操作时，必须先建立一个线程 dev1_read_thread

查找设备 dev1

读设备 dev1

```
            }
            printf("dev1 ReadData = %d\n", ReadData[0]);
            cyg_thread_delay(10);
        }
    }
```

事实上，除了上面介绍的通过调用 I/O 子系统 API 函数来实现对设备的 lookup、读、写和配置等操作外，还可以通过调用文件系统（devfs）API 函数实现对设备的打开、读、写等操作，这时候设备相当于一个文件；而这些文件系统 API 函数也会通过指针调用设备驱动程序函数。

如下为一个典型利用文件系统对设备进行打开和读/写操作的程序段：

```
int main( void )
{
    unsigned int writebuff[10],readbuff[10];
    int err, fd;

    writebuff[0] = 66;
    CYG_TEST_INIT();
    fd = open( "/dev/dev1",O_RDWR );

    err = write( fd, writebuff, 1);
    if(err < 0 )
        printf("write err");

    err = read( fd, readbuff, 20);
    if(err < 0)
        printf("read err");

    close(fd);
    CYG_TEST_PASS_FINISH("dev1 devfs");
```

```
}
```

3 字符型设备驱动程序设计

SPCE3200 的大多数硬件模块都可以划分为字符设备，比如 GPIO、ADC、PWM（Timer）、UART、SPI、I2C 等。限于篇幅，这里只介绍两个比较典型的设备 GPIO 和 SPI 的驱动程序编写方法。

3.1 GPIO驱动程序设计

SPCE3200 最多拥有 79 个通用 I/O 口（GPIO——General Purpose I/O ports），其中有 8 个 I/O 口只做为 GPIO 使用，其它 I/O 口与其它功能模块接口复用。关于 79 个 GPIO 口详见《嵌入式微处理器 SPCE3200 原理与应用》第四章 GPIO 一节。这里以 IOB 为例介绍 GPIO 驱动的编写，约定 IOB 的设备名为 iob。

按照图 1.6介绍编写驱动程序的一般步骤，先从建立驱动程序目录入手。

3.1.1 建立驱动程序目录

在devs目录下建立如图 3.1所示的目录结构。

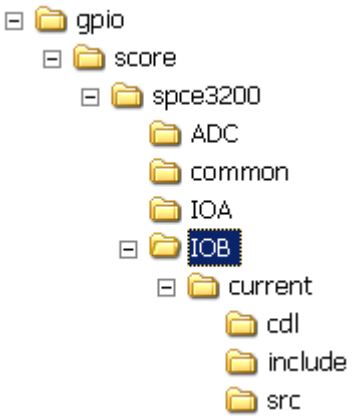

图 3.1 IOB 驱动程序组件包的目录结构

3.1.2 建立驱动程序文件

驱动程序文件包括 cdl 脚本文件、源程序文件和头文件。

在图 3.1目录结构中显示的cdl文件夹里建立一个cdl脚本文件，假设文件名为iob_spce3200.cdl；

在图 3.1目录结构中显示的src文件夹里建立一个C文件作为驱动程序的源程序文件，假设文件名为iob_spce3200.c。

在图 3.1目录结构中显示的include文件夹里建立一个头文件作为驱动程序的头文件，假设文件名为iob_spce3200.h。

3.1.3 设计CDL脚本文件

在编写 CDL 文件前，首先要考虑在该驱动函数中有多少个可以通过配置工具中配置的项，分别是什么。这里建立两个可配置的项 debug 调试配置信息和 IOB 口属性设置项配置信息。

按照2.4.1中介绍的CDL脚本文件设计方法，在脚本文件（假设为iob_spce3200.cdl）中编写cdl程序：先要建立一个IOB驱动程序的组建包，在组建包中要定义前面提到的几个配置选项宏。CDL脚本文件的参考程序如下：

```
cdl_package CYGPKG_DEVS_IOB_SPCE3200 {
    display       " IOB driver for SPCE3200"
    include_dir   cyg/io
    requires      CYGPKG_IO

    compile           -library=libextras.a iob_spce3200.c

    description " IOB driver for SCORE SPCE3200"

    cdl_component CYGPKG_DEVS_IOB_SPCE3200_OPTIONS {
        display "Compile Options"
        flavor  none
        no_define

        cdl_option CYGPKG_DEVS_IOB_SPCE3200_CFLAGS {
            display         "Additional compiler flags"
            flavor          data
            no_define
            default_value { "" }
            description "
                This option modifies the set of compiler flags for
                building the keypad driver package. These flags
                are used in addition to the set of global flags."
        }

        cdl_option CYGDAT_DEVS_IOB_SPCE3200_NAME {
```

建立 IOB 驱动程序包

头文件将被拷贝到编译路径 cyg/io

编译资源文件 iob_spce3200.c

建立 IOB 驱动程序包的编译调试组件

定义 IOB 驱动设备名

```
            display "Device name for the IOB driver"
            flavor data
            default_value {"\"/dev/iob\""}
            description " This option specifies the name of the IOB device"
        }

        cdl_option CYGNUM_DEVS_IOB_SPCE3200_DEBUG_MODE {
            display "Debug Message"
            default_value    0
            description "
                This option will enable the debug message outputing if set to 1,
                will disable the outputing if set to 0."
        }
    }
    cdl_component CYGHWR_DEVS_IOB_SPCE3200_OPTIONS {
        display "Connection Options"
        flavor    none
        no_define

        cdl_option CYGNUM_DEVS_IOB_SPCE3200_EVENT_BUFFER_SIZE {
            display "Number of events the driver iob buffer"
            flavor data
            legal_values 1 to 512
            default_value 32
            description "
                This option defines the size of the keypad device internal
            buffer. The cyg_io_read() function will return as many of these
            as there is space for in the buffer passed."
        }

        cdl_option CYGNUM_DEVS_IOB_SPCE3200_CONFIG {
            display "IOB config enable"
```

IOB 驱动设备名为：/dev/iob

定义 IOB 驱动的调试模式

0：不使用调试模式
1：使用调试模式

定义 IOB 驱动的硬件设置

定义驱动数据 Buffer 的大小

有效数据：1~512

定义 IOB 口的硬件配置项

```
                default_value    1
                description "
                    This option will enable the config if set to 1,
                    will disable the config if set to 0."
            }
        }
    }
```

1：该项使能，IOB 口的属性可设置
0：该项不使能，IOB 口的属性不可设置

以上带边框的文字中，CYGDAT_DEVS_IOB_SPCE3200_NAME、CYGNUM_DEVS_IOB_SPCE3200_DEBUG_MODE、CYGNUM_DEVS_IOB_SPCE3200_CONFIG分别为 IOB 设备名、debug 选项和 IOB 口设置项。

CYGPKG_DEVS_IOB_SPCE3200 是 IOB 设备驱动程序组件包，写好驱动源程序文件后可以把该包加到 eCos 数据库中。

3.1.4 设计头文件

一般在头文件中定义一些常用的常量、结构体、cyg_io_set_config 及 cyg_io_get_config 函数函数要传递的参数 key 的常量等。

在 IOB 驱动中，只需定义 cyg_io_set_config 及 cyg_io_get_config 函数函数要传递的参数 key 的常量。如下：

```
#define CYGNUM_DEVS_IOB_SPCE3200_ATTRIB      0x1121          // IOB 口的属性
                                                             // 作为 key
```

3.1.5 设计源程序文件

按照图 1.6设备驱动编写步骤，编写完cdl脚本文件及头文件后，在源程序文件（假设为iob_spce3200.c）里编写驱动设备表入口和驱动的接口函数。

IOB 为通用的输入输出口，可以输入或者输出高电平和低电平，而这些输入或者输出没有严格的时序控制，只要用户需要，就可以对其端口通过寄存器进行读/写。IOB 设备驱动源程序文件实现的功能是对 IOB 口的输入输出状态进行设置，同时可以读端口的输入数据或者输出数据到端口，而这些输入或者输出可以直接通过操作寄存器的方式来实现。

3.1.6至3.1.8节为IOB设备源程序文件程序的详细设计。

3.1.6 编写IOB设备表入口

根据2.4.2介绍，定义字符设备表入口就是定义两个宏：CHAR_DEVTAB_ENTRY 和

CHAR_DEVIO_TABLE。分别定义如下：

```
CHAR_DEVIO_TABLE(spce3200_iob_handlers,// IOB 函数表的句柄
    iob_write,                  // IOB 设备写函数
    iob_read,                   // IOB 设备读函数
    NULL,                       // IOB 设备选择函数，这里没有指定指针入口
    iob_get_config,             // IOB 读设备设置函数，
    iob_set_config              // IOB 写设备设置函数
);

CHAR_DEVTAB_ENTRY(spce3200_ iob_device,    // 设备入口表的句柄
    CYGDAT_DEVS_IOB_SPCE3200_NAME,     // 设备名，在 cdl 文件中进行宏定义
    NULL,                              //
    &spce3200_iob_handlers,            // IOB 函数表的句柄指针
    iob_init,                          // IOB 设备初始化函数
    iob_lookup,                        // IOB 设备查找函数
    NULL                               //
);
```

其中iob_write、iob_read、iob_get_config、iob_set_config、iob_init、iob_lookup六个接口函数都需要定义；CYGDAT_DEVS_IOB_SPCE3200_NAME为设备名，在iob的设备脚本cdl文件定义该宏为“dev\iob”，详见3.1.3节。

3.1.7 编写IOB设备驱动接口函数

GPIO 设备有 6 个设备驱动接口函数：iob_write、iob_read、iob_get_config、iob_set_config、iob_init、iob_lookup；下面一一介绍：

1、 IOB 写函数 iob_write

由2.4.3节知道，IOB写函数格式为：

```
Cyg_ErrNo iob_write (cyg_io_handle_t handle,void *buffer,cyg_uint32 *len)
{
}
```

在 IOB 的写函数中，主要完成的功能是向 IOB 的输出数据寄存器写入 buffer 内的数据。调用一次 iob_write 函数写数据的个数为 len。参考程序段如下：

```
static Cyg_ErrNo iob_write(cyg_io_handle_t handle, void *buffer, cyg_uint32 *len)
{
    cyg_uint32 *bp = (cyg_uint32 *)buffer;
    cyg_uint32 tempdata,tempdata1;
    int tot = *len;

#if CYGNUM_DEVS_IOB_SPCE3200_DEBUG_MODE == 1
    printf("Debug – IOB write\n");                    // 打印调试信息
#endif
    while(tot>0)
    {
        tempdata = *bp;
        tempdata &= 0x0000003f;
        tempdata1 = *P_IOB_GPIO_SETUP;
        tempdata1 &= 0xffffffc0;
        tempdata |= tempdata1;
        *P_IOB_GPIO_SETUP = tempdata;              // 通过 IOB 输出数据 tempdata
        tot -= 1;
        bp++;
    }
    return ENOERR;
}
```

2、 IOB 读函数 iob_read

由2.4.3节知道，IOB读函数格式为：

```
Cyg_ErrNo iob_ read (cyg_io_handle_t handle,void *buffer,cyg_uint32 *len)
{
}
```

在 IOB 的读函数中，主要完成的功能是从 IOB 的输入数据寄存器读回数据，并把该数据写入 buffer。调用一次 iob_read 函数读数据的个数为 len。参考程序段如下：

```
static Cyg_ErrNo iob_read(cyg_io_handle_t handle, void *buffer, cyg_uint32 *len)
```

```
{
    unsigned char *bp = (unsigned char *)buffer;
    cyg_uint32 tempdata;
    int tot = *len;

#if CYGNUM_DEVS_IOB_SPCE3200_DEBUG_MODE == 1
    printf("Debug – IOB read\n");                    // 打印调试信息
#endif
    while(tot>0)
    {
        tempdata = *P_IOB_GPIO_INPUT;              // 读 IOB 口数据
        tempdata &= 0x00003f00;
        *bp = tempdata;
        tot -= 1;
        bp++;
    }
    return ENOERR;
}
```

3、 IOB 设备配置函数 iob_set_config

由2.4.3节知道，IOB设置函数格式为：

```
Cyg_ErrNo iob_set_config(   cyg_io_handle_t handle,
                            cyg_uint32 key,
                            const void *buffer,
                            cyg_uint32 *len
                        )
{
}
```

在 IOB 的设置函数中，主要完成的功能设置 IOB 的输入输出，如果设置为输入时，设置为上拉或者下拉输入。参考程序段如下：

```
static Cyg_ErrNo iob_set_config(  cyg_io_handle_t handle,
```

```
                                    cyg_uint32 key,
                                    const void *buffer,
                                    cyg_uint32 *len
                                   )
{
    cyg_uint32 *bp = (cyg_uint32 *)buffer;
    #if CYGNUM_DEVS_IOB_SPCE3200_DEBUG_MODE == 1
    printf("Debug - iob Set Config\n");
    #endif
    cyg_uint32 iob_attrib;
    #ifdef CYGNUM_DEVS_IOB_SPCE3200_CONFIG          // 在 iob_spce3200.cdl 中定义
// 在配置工具中可以使能该项，如果使能，IOB 端口可配置；否则不可配置。该项默认为使能
    switch(key)
    {
        case CYGNUM_DEVS_IOB_SPCE3200_ATTRIB:       // 在 iob_spce3200.h 中定义
        //------------------ 设置 IOB 输入输出方向 ------------------//
        {
            iob_attrib = *bp;
            *P_IOB_GPIO_SETUP = iob_attrib;
        }
            break;

        default:
            break;
    }
    #endif
    return ENOERR;
}
```

4、 读 IOB 设备配置状态函数 iob_get_config

由2.4.3节知道，IOB读设置状态函数格式为：

```
Cyg_ErrNo iob_ get_config ( cyg_io_handle_t handle,
```

```
                                    cyg_uint32 key,
                                    void *buf,
                                    cyg_uint32 *len
                                    )
{
}
```

在 IOB 的读设置状态函数中，读出目前 IOB 的输入输出状态，以及上/下拉状态。参考程序段如下：

```
static Cyg_ErrNo iob_ get_config(cyg_io_handle_t handle,
                                 cyg_uint32 key,
                                 const void *buffer,
                                 cyg_uint32 *len
                                 )
{
    cyg_uint32 *bp = (cyg_uint32 *)buffer;
    #if CYGNUM_DEVS_IOB_SPCE3200_DEBUG_MODE == 1
    printf("Debug - iob Get Config\n");
    #endif
    cyg_uint32 iob_attrib;

    #ifdef CYGNUM_DEVS_IOB_SPCE3200_CONFIG
    switch(key)
    {
        case CYGNUM_DEVS_IOB_SPCE3200_ATTRIB:
        {
            iob_attrib = *P_IOB_GPIO_SETUP;
            *bp = iob_attrib;                    // 获取 IOB 口的属性
        }
        break;
        default:
            break;
    }
```

```
        #endif
        return ENOERR;
    }
```

5、 IOB 初始化函数

由2.4.3节知道，IOB初始化函数格式为：

```
static bool iob_init(struct cyg_devtab_entry *tab)
{
}
```

IOB 初始化函数在 eCos 内核启动之前的硬件初始化函数里调用。IOB 初始化函数用来设置 IOB 口的初始状态。参考程序段如下：

```
static bool iob_init(struct cyg_devtab_entry *tab)
{
    #if CYGNUM_DEVS_IOB_SPCE3200_DEBUG_MODE == 1
        diag_printf("Debug - IOB Init\n");
    #endif
    return true;
}
```

这里没有对 IOB 口的任何寄存器进行操作，默认的 IOB 口为带下拉电阻的输入口。

6、 IOB 设备查找函数

由2.4.3节知道，IOB设备查找函数格式为：

```
static Cyg_ErrNo iob_lookup (    struct cyg_devtab_entry **tab,
                                 struct cyg_devtab_entry *st,
                                 const char *name
                            )
{
}
```

IOB 设备查找函数的主要功能是传递参数 tab，该函数在 cyg_ iob_lookup 中调用，用来查找是否存在 IOB 设备。参考程序段如下：

```
static Cyg_ErrNo iob_lookup (    struct cyg_devtab_entry **tab,
```

```
                                    struct cyg_devtab_entry *st,
                                    const char *name
                                )
{
    #if CYGNUM_DEVS_IOB_SPCE3200_DEBUG_MODE == 1
        printf("Debug - IOB Lookup\n");
    #endif
    return ENOERR;
}
```

IOB 设备查找函数中还可以有其他操作，比如建立线程，建立中断等，以上是一个最简单的设备查找函数，用户可以在写驱动函数时可以根据自己的需求来具体决定。

3.1.8 IOB驱动程序范例

在完成了上面各个模块的编写后，要进行整个驱动源程序文件的完整设计。IOB的驱动源程序文件包含如图 3.2所示的模块。

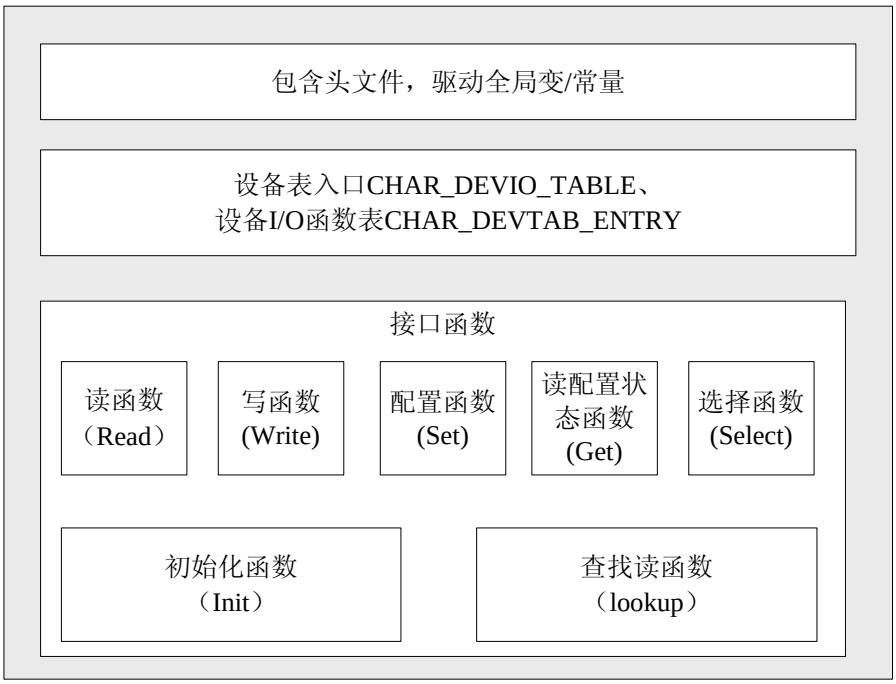

图 3.2 IOB 驱动源程序文件结构示意图

其中设备表入口、设备I/O函数表入口、接口函数都已经介绍，下面主要考虑要包含的头文件和全局变量/常量。根据上两节设备表入口接口函数确定要包含的头文件和需要定义的全局变量，如图3.3。在前面编写设备表入口和接口函数时没有用到全局变量，所以不需要定义。

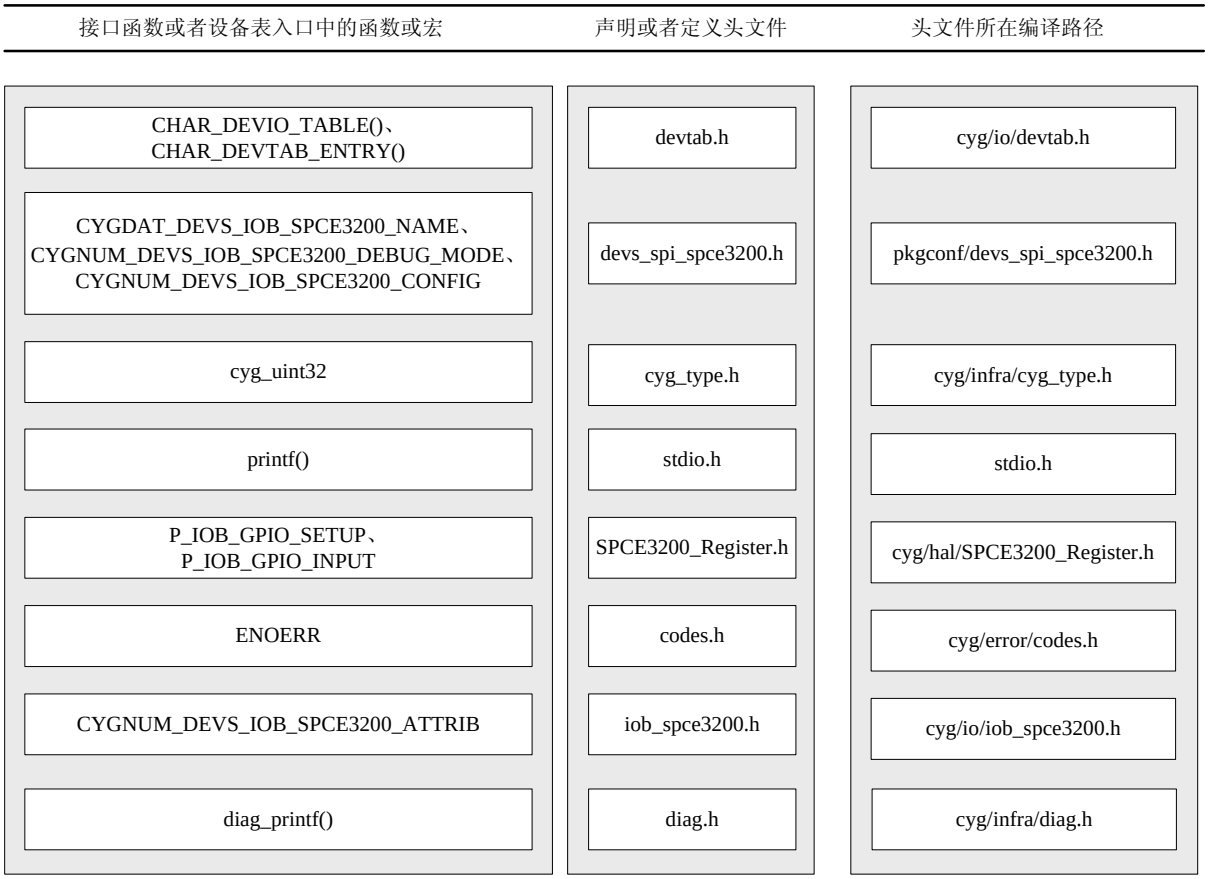

图 3.3 IOB 驱动源程序文件需要包含的头文件示意图

如下为一个完整的 IOB 设备驱动源程序文件的伪代码：

```
//---------------------------------------包含相关头文件--------------------------//
#include <pkgconf/hal.h>                    // hal 的宏头文件，由系统自动生成，一般建议包含
#include <pkgconf/devs_iob_spce3200.h>      // spi 驱动 cdl 文件宏头文件，由系统自动生成

#include <cyg/infra/cyg_type.h>             // 定义类型
#include <cyg/hal/SPCE3200_Register.h>      // 定义 SPCE3200 硬件寄存器
#include <cyg/error/codes.h>                // 定义错误常量，如 ENOERR
#if CYGNUM_DEVS_IOB_SPCE3200_DEBUG_MODE == 1// 如果调试模式使能
    #include <cyg/infra/diag.h>             // diag 函数声明
    #include <stdio.h>                      // 调试函数如 printf 等声明
#endif
#include <cyg/io/devtab.h>                  // 设备 I/O 入口表定义及声明
#include<cyg/io/iob_spce3200.h>         // 定义 SPCE3200 的 SPI 模块相关常量，可含 key 的定义
//-------------------------------------------------------------------------------//
```

```
//------------------------------接口函数声明---------------------------------//
static Cyg_ErrNo iob_write(cyg_io_handle_t handle, void *buffer, cyg_uint32 *len);
static Cyg_ErrNo iob_read(cyg_io_handle_t handle, void *buffer, cyg_uint32 *len);
static Cyg_ErrNo iob_set_config(cyg_io_handle_t handle,
                                cyg_uint32 key,const void *buffer,cyg_uint32 *len);
static Cyg_ErrNo iob_get_config(cyg_io_handle_t handle,
                                cyg_uint32 key,const void *buffer,cyg_uint32 *len);
static bool iob_init(struct cyg_devtab_entry *tab);
static Cyg_ErrNo iob_lookup (struct cyg_devtab_entry **tab,
                             struct cyg_devtab_entry *st,const char *name);

CHAR_DEVIO_TABLE(…);
CHAR_DEVTAB_ENTRY(…);
```

设备表入口详见 3.1.6

```
static Cyg_ErrNo iob_write(cyg_io_handle_t handle, void *buffer, cyg_uint32 *len)
{
    …
}
```

函数体详见 3.1.7 IOB 写函数

```
static Cyg_ErrNo iob_read(cyg_io_handle_t handle, void *buffer, cyg_uint32 *len)
{
    …
}
```

函数体详见 3.1.7 IOB 读函数

```
static Cyg_ErrNo iob_set_config(cyg_io_handle_t handle, cyg_uint32 key, const void *buffer,
cyg_uint32 *len)
{
    …
}
```

函数体详见 3.1.7 IOB 设置函数

```
static Cyg_ErrNo iob_get_config(cyg_io_handle_t handle, cyg_uint32 key, void *buffer, cyg_uint32
*len)
{
```

```
        …
    }

    static bool iob_init(struct cyg_devtab_entry *tab)
    {
        …
    }

    static Cyg_ErrNo iob_lookup(struct cyg_devtab_entry **tab, struct cyg_devtab_entry *st, const char
*name)
    {
        …
    }
```

函数体详见 3.1.7 IOB 读设置状态函数

函数体详见 3.1.7 IOB 初始化函数

函数体详见 3.1.7 IOB 设备查找函数

3.1.9 向eCos数据库中添加IOB驱动程序组件包

根据1.4.1加载驱动程序组件包到数据库中办法，在ecos.db中的任意位置加如下程序段：

```
package CYGPKG_DEVS_IOB_SPCE3200 {
    alias           { "IOB support for Score SPCE3200"}
    directory  devs/gpio/score/spce3200/IOB
    script          iob_spce3200.cdl
        description "
          This package contains hardware support for the IOB read/write on the Score SPCE3200
    EV Board. "
}
```

这里假设IOB驱动程序组件包的结构如图 3.1。

此时如果用eCos配置工具添加包时，就可以看见IOB驱动程序组件包已经出现在eCos数据库中，如图 3.4：点击“Add”加载到平台上进行编译生成*.ecc库文件后，即可以使用IOB驱动程序。

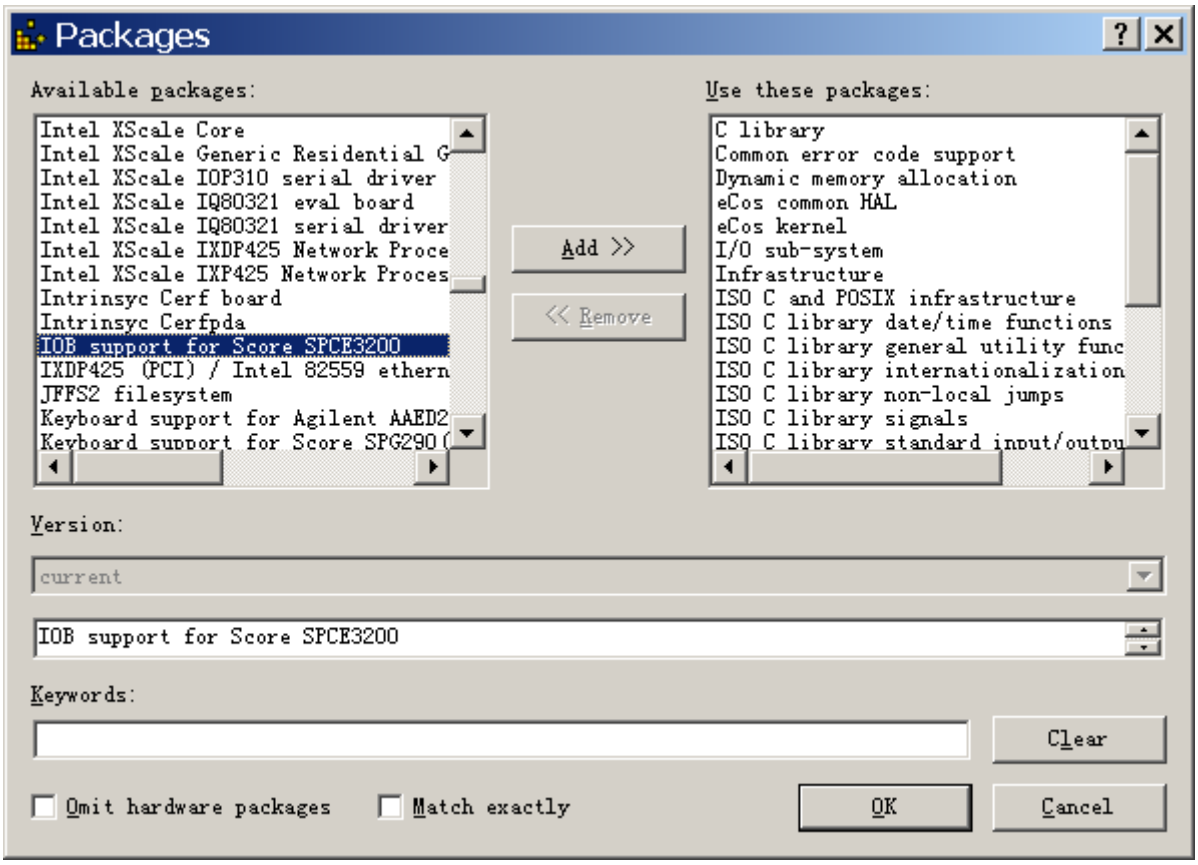

图 3.4 IOB 驱动程序组件包在 eCos 数据库中界面

3.1.10 用户测试程序

在把 IOB 设备驱动程序组件包加载到 SPCE3200 开发平台编译没有错误后，就可以编写测试程序验证驱动程序是否工作正常。

为了方便测试，SPCE3200 的 IOB0~2 连接三个按键，IOB3~5 连接三个 LED，编写一个测试程序，当有键按下时点亮与之对应的 LED。

分析：该测试程序需要完成两个功能：取键值即读 IOB 设备，点亮发光二极管即输出高电平到 IOB 设备；另外，在读/写 IOB 设备之前，必须把 IOB0~2 设置为输入口，IOB3~5 设置为输出口。

参考程序如下：

```
//-----------------------------------包含相关头文件-----------------------------------//
#include <cyg/infra/testcase.h>          // 定义测试宏
#include <cyg/io/io.h>                   // 声明 I/O 子系统 API 函数
#include <cyg/io/iob_spce3200.h>         // 定义了 CYGNUM_DEVS_IOB_SPCE3200_ATTRIB
#include <cyg/kernel/kapi.h>             // 定义针对内核的 API 函数，同步机制的 API 函数等
#include <stdio.h>                       // 声明 printf 等调试函数
```

```
cyg_io_handle_t         hDrvIOB;
cyg_mutex_t             mut_shared_iob
// 为了防止其他线程使用 IOB 口，造成的端口资源冲突，定义一个互斥量
//----------------------点亮按键按下按键对应的 LED 线程 iob_thread ----------------------------//
void iob_thread(cyg_addrword_t data)
{
    cyg_uint32              len = 1;
    cyg_uint32              ReadData[1];
    cyg_uint32              WriteData[1];
    cyg_uint32              Setup[1];

    if (ENOERR != cyg_io_lookup("/dev/iob", &hDrvIOB)) // 查找 IOB 设备
    {
        CYG_TEST_FAIL_FINISH("Error opening /dev/iob");
    }
    printf("Open /dev/iob OK\n");
    Setup[0] = 0x07073800;              // 设置 IOB0~2 为带下拉电阻输入口
                                        // 设置 IOB3~5 为输出口
    while(true)
    {
        cyg_mutex_lock(&mut_shared_iob);    // 锁定互斥量
        if(ENOERR != iob_set_config(hDrvIOB,
                                    CYGNUM_DEVS_IOB_SPCE3200_ATTRIB,
                                   Setup, &len))  // 设置 IOB 口
        {
            CYG_TEST_FAIL_FINISH("Error set config /dev /iob");
        }

        len = 1;
        if(ENOERR != cyg_io_read(hDrvIOB, ReadData, &len))// 读键值
        {
            CYG_TEST_FINISH("Error Read /dev/iob");
```

```
            }
            printf("IOB0-2 input Data = %d\n", ReadData[0]);
            if(ReadData[0]!=0)
            {
                WriteData[0] = (ReadData[0] >> 5);
                if(ENOERR != cyg_io_write(hDrvIOB, WriteData, &len))//  点亮 LED
                {
                    CYG_TEST_FINISH("Error Write /dev/iob");
                }
                printf("IOB3-5 output Data = %d\n", WriteData[0]);
            }
            cyg_mutex_unlock(&mut_shared_iob);// 释放互斥量，其他线程这时候可以用 IOB 口
            cyg_thread_delay(10);
        }
    }

    cyg_uint8 stack[4096];
    cyg_handle_t simple_threadA;
    cyg_thread thread_s;

    void cyg_user_start(void)
    {
        cyg_uint32        len;
        CYG_TEST_INIT();
        cyg_thread_create(4, iob_thread,
                          (cyg_addrword_t) 0,
                          "iob_thread",
                          (void *) stack, 4096,
                          &simple_threadA, &thread_s);     // 建立一个线程 iob_thread
        cyg_thread_resume(simple_threadA);
        cyg_mutex_init(&mut_shared_iob);                   // 初始化互斥量 mut_shared_iob
        cyg_scheduler_start();
```

```
}
```

3.2 SPI驱动程序设计

SPCE3200 内嵌一个 SPI 总线接口，支持主/从模式的单字节和连续多字节数据传输。关于 SPI 总线接口的详细说明请参考《嵌入式微处理器 SPCE3200 原理与应用》第四章 SPI 一节。

3.2.1 建立驱动程序目录

在devs目录下建立如图 3.5所示的目录结构。

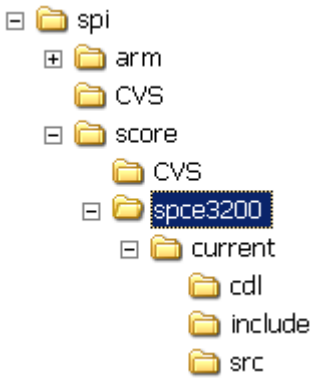

图 3.5 SPI 驱动程序组件包的目录结构

3.2.2 建立驱动程序文本文件

驱动程序文件包括 cdl 脚本文件和源程序文件，本例中不需要建立头文件。

在图 3.5目录结构中显示的cdl文件夹里建立一个cdl脚本文件，假设文件名为spi_spce3200.cdl；

在图 3.5目录结构中显示的include文件夹里建立一个.h文件作为驱动程序的头文件，假设文件名为spi_spce3200.h。

在图 3.5目录结构中显示的src文件夹里建立一个C文件作为驱动程序的源程序文件，假设文件名为spi_spce3200.c。

3.2.3 设计CDL脚本文件

这里建立两个可配置的项 debug 调试配置信息和 SPI 主/从模式选择配置信息。

按照2.4.1中介绍的CDL脚本文件设计方法，在脚本文件（假设为spi_spce3200.cdl）中编写cdl程序：先要建立一个SPI驱动程序的组件包，在组件包中要定义前面提到几个配置选项的宏。CDL脚本文件的参考程序如下：

建立 SPI 驱动程序组件包

```
cdl_package CYGPKG_DEVS_SPI_SPCE3200 {
```

```
display       "SPI driver for SPCE3200"
include_dir cyg/io
requires      CYGPKG_IO
compile           -library=libextras.a spi_spce3200.c
description "SPI driver for the SPCE3200"

cdl_component CYGPKG_DEVS_SPI_SPCE3200_OPTIONS {
    display "Compile Options"
    flavor    none
    no_define

    cdl_option CYGPKG_DEVS_SPI_SPCE3200_CFLAGS {
        display           "Additional compiler flags"
        flavor            data
        no_define
        default_value { "" }
        description "
            This option modifies the set of compiler flags for
            building the spi driver package. These flags
            are used in addition to the set of global flags."
    }

    cdl_option CYGDAT_DEVS_SPI_SPCE3200_NAME {
        display "Device name for the spi driver"
        flavor data
        default_value {"\"/dev/spi\""}
        description " This option specifies the name of the spi device"
    }

    cdl_option CYGNUM_DEVS_SPI_SPCE3200_DEBUG_MODE {
        display "Debug Message"
        default_value    0
```

头文件将被拷贝到编译路径 cyg/io

编译资源文件 spi_spce3200.c

建立 SPI 驱动程序包的编译调试组件

定义 SPI 驱动设备名

IOB 驱动设备名为：/dev/spi

定义 IOB 驱动的调试模式配置项

```
            description "
                This option will enable the debug message outputing if set to 1,
                will disable the outputing if set to 0."
        }
    }

    cdl_component CYGHWR_DEVS_SPI_SPCE3200_OPTIONS {
        display "Connection Options"
        flavor  none
        no_define

        cdl_option CYGNUM_DEVS_SPI_SPCE3200_EVENT_BUFFER_SIZE {
            display "Number of events the driver can buffer"
            flavor data
            legal_values 1 to 512
            default_value 32
            description "
                This option defines the size of the keypad device internal
            buffer. The cyg_io_read() function will return as many of these
            as there is space for in the buffer passed."
        }

        cdl_option CYGNUM_DEVS_SPI_SPCE3200_MODE_SELECT {
            display "SPI Master/Slave Mode Select"
            flavor data
            legal_values    { "MASTER" "SLAVE" }
            default_value {"MASTER"}
            description "
                This option select Master or Slave mode."
        }
    }
}
```

0：不使用调试模式
1：使用调试模式

定义 SPI 口的硬件配置组件

定义驱动数据 Buffer 的大小

有效数据：1~512

定义 SPI 模式选择配置项

MASTER： SPI 主模式
SLAVE：SPI 从模式

以上带边框的文字中，CYGDAT_DEVS_SPI_SPCE3200_NAME、CYGNUM_DEVS_SPI_SPCE3200_DEBUG_MODE、CYGNUM_DEVS_SPI_SPCE3200_MODE_SELECT为后面驱动接口函数中将会出现的宏，功能分别是：定义 SPI 设备名、定义 debug 选项和 SPI 主/从模式设置项。

CYGPKG_DEVS_SPI_SPCE3200是 SPI 设备驱动程序组件包，写好驱动源程序文件和头文件后可以把该包加到 eCos 数据库中。

3.2.4 设计头文件

一般在头文件中定义一些常用的常量、结构体、cyg_io_set_config 及 cyg_io_get_config 函数函数要传递的参数 key 的常量等。

在 SPI 驱动中，只需定义 cyg_io_set_config 及 cyg_io_get_config 函数函数要传递的参数 key 的常量。如下：

```
#define CYGNUM_DEVS_SPI_SPCE3200_MASTER_CLOCK    0x1601 //  主频时钟设置 key
#define CYGNUM_DEVS_SPI_SPCE3200_SPH             0x1602 // SPH 设置 key
#define CYGNUM_DEVS_SPI_SPCE3200_SPO             0x1603 // SPO 设置 key
```

3.2.5 设计源程序文件

按照图 1.6设备驱动编写步骤，编写完cdl脚本文件及头文件后，在源程序文件（假设为spi_spce3200.c）里编写驱动设备表入口和驱动的接口函数。

SPCE3200 的 SPI 控制器有严格的通讯时序，同时在通讯的过程中不能被打断，且一次通讯中不能被其他任务占用 SPI 总线。这样，保证通讯的不中断性，却使得 CPU 在和 SPI 总线交互一批指定数据的过程中，不能处理其他的任务，虽然有时接收数据还没有被送到接收口，CPU 必须等待数据到达接收口且接收完这批数据才能进行其他操作，同样，虽然还没有准备好发送数据，CPU 必须等到准备好发送数据且完成数据发送后才能进行其他操作。

为了提高效率，利用中断的方式实现数据的收发，设计如图 3.6所示的SPI设备驱动程序结构和调用流程。把驱动程序分为物理层和逻辑层两个层，逻辑层只进行操作Buffer，物理层进行与硬件的交互，即把发送Buffer中的数据填入发送数据寄存器，或者把接收数据寄存器接收到的数据填入接收Buffer。在物理层里通过中断发送或者接收数据，这样在没有数据要接收时，CPU可以转向进行其他操作，或者在把发送数据全部填到Buffer后，不需要专门等待其发送，而是在CPU有空闲的时候或者执行完高优先级的任务后发送，只要保证发送完填入Buffer中的所有数据即可。

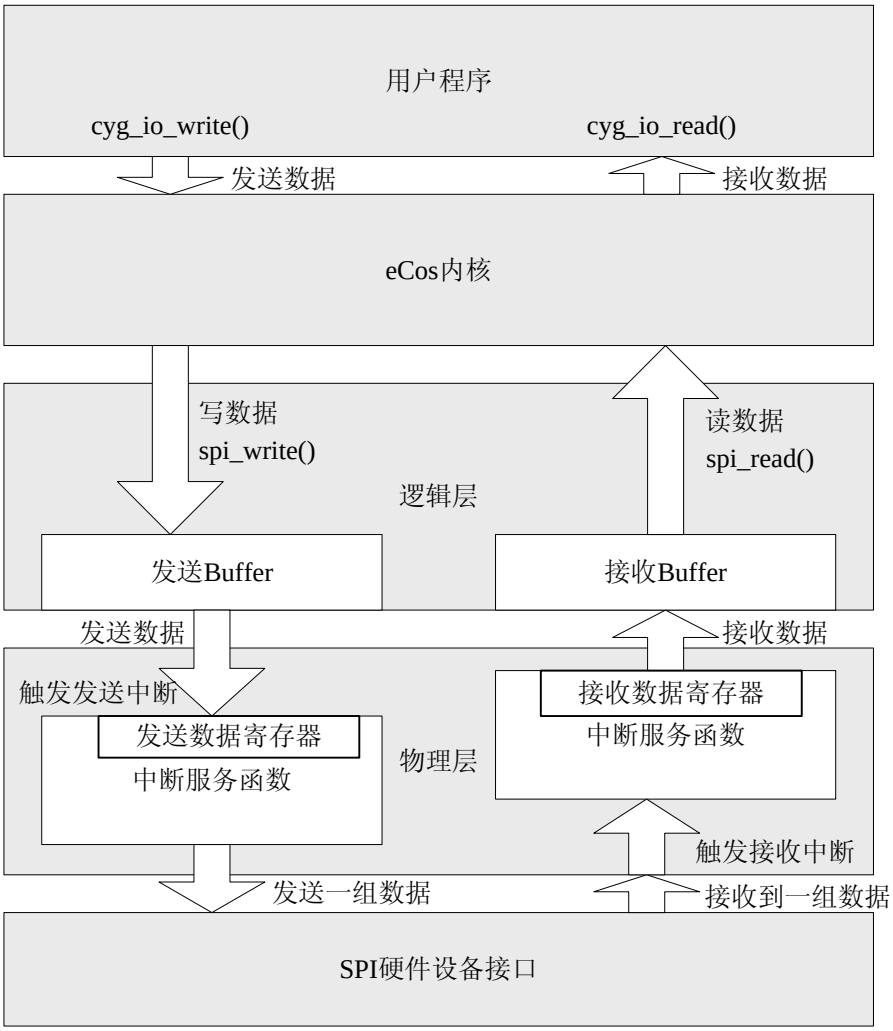

图 3.6 SPI 设备驱动程序结构和调用流程

3.2.6至3.2.9为SPI驱动源程序文件的详细设计过程。

3.2.6 编写SPI设备表入口

根据2.4.2介绍，定义字符设备表入口就是定义两个宏：CHAR_DEVTAB_ENTRY 和 CHAR_DEVIO_TABLE。分别定义如下：

```
CHAR_DEVIO_TABLE(spce3200_spi_handlers, // SPI 设备 I/O 函数表句柄
    spi_write,                          // SPI 设备写函数
    spi_read,                           // SPI 设备读函数
    NULL,                               //
    spi_get_config,                     // SPI 读设备设置状态函数
    spi_set_config                      // SPI 设备设置函数
);
```

```
CHAR_DEVTAB_ENTRY(spce3200_spi_device,      // SPI 设备表入口句柄
    CYGDAT_DEVS_SPI_SPCE3200_NAME,          // 设备名，在 cdl 文件中进行宏定义
    NULL,                                   //
    &spce3200_spi_handlers,                 // SPI 设备 I/O 函数表句柄指针
    spi_init,                               // SPI 初始化函数
    spi_lookup,                             // SPI 设备查找函数
    NULL
);
```

其中spi_write、spi_read、spi_get_config、spi_set_config、spi_init、spi_lookup六个接口函数都需要定义；CYGDAT_DEVS_SPI_SPCE3200_NAME为设备名，在spi的设备脚本文件定义该宏为“dev\spi”，详见3.1.3节。

3.2.7 编写SPI设备驱动中断服务程序

按照图 3.6的设计思路，中断服务程序完成的功能是如果正在接收数据，发生了接收中断时，把接收数据写到接收Buffer中；如果正在发送数据，发生了发送中断，从发送Buffer中取出要发送的数据填到发送数据寄存器中等待发送。

SPI 设备驱动中断服务参考程序如下：

```
static cyg_uint32 spi_INTService( cyg_uint32 vector, CYG_ADDRWORD data )
{
    cyg_uint8 tempdata;

#if CYGNUM_DEVS_SPI_SPCE3200_DEBUG_MODE == 1   // 如果是调试模式
    CYG_TEST_CHECK( ISR_DATA == data , "Bad data passed to ISR");
                                                // 检测安装数据是否成功
    CYG_TEST_CHECK( 43 == vector , "Bad vector passed to ISR");
                                                // 检测是否发生了 IRQ43 中断
    printf("Debug - spi Int Services\n");
#endif
    HAL_INTERRUPT_ACKNOWLEDGE( vector );        // 响应 IRQ43 中断
    if(TxFlag==1)                               // 如果正在发送数据
```

```
    {
        if((*P_SPI_TX_STATUS&C_SPI_TX_FLAG)!=0)    // 如果是发送中断
        {
            *P_SPI_TX_STATUS &= ~C_SPI_TX_INTEN; // 禁能中断
            *P_SPI_RX_STATUS &= ~C_SPI_RX_INTEN; // 禁能中断
            if(_TxData_Num>0)                    // 发送 Buffer 中是否有数据
            {
                if(*P_SPI_TXRX_STATUS&C_SPI_TXFIFO_NOTFULL)
                {                                // SPI 发送 FIFO 非满
                    tempdata = _Tx_buffer[_TxData_get]; // 取发送 Buffer 中数据
                    *P_SPI_TX_DATA = tempdata;       // 写发送数据到 SPI 数据寄存器
                    _TxData_get += 1;
                    _TxData_Num -= 1;
                }
                if(*P_SPI_TXRX_STATUS&C_SPI_RXFIFO_NOTEMPTY)
                    tempdata = *P_SPI_RX_DATA;       // 无效数据处理，清空读 FIFO
            }
            if(_TxData_get==MAX_EVENTS)              //
                _TxData_get = 0;
        }
    }
    if(RxFlag==1)                                    // 如果正在接收数据
    {
        if((*P_SPI_RX_STATUS&C_SPI_RX_FLAG)!=0)    // 如果是接收中断
        {
            *P_SPI_RX_STATUS &= ~C_SPI_RX_INTEN;// 禁能中断
            if(_RxData_Num<MAX_EVENTS)               // 接收 Buffer 中非满
            {
                *P_SPI_TX_DATA = 0xff;               // 启动接收
                if(*P_SPI_TXRX_STATUS&C_SPI_RXFIFO_NOTEMPTY)
                {
                    tempdata = *P_SPI_RX_DATA;       // 读接收数据
```

```
                    _Rx_buffer[_RxData_put] = tempdata; // 把接收数据存入接收 Buffer
                    _RxData_put += 1;
                    _RxData_Num += 1;
                }
                if(_RxData_put==MAX_EVENTS)
                    _RxData_put = 0;
            }
        }
    }
    return CYG_ISR_HANDLED;                    // ISR 中断服务程序返回
}
```

3.2.8 编写SPI设备驱动接口函数

按照3.2.6节的设备表入口，SPI设备有 6 个设备驱动接口函数：spi_write、spi_read、spi_get_config、spi_set_config、spi_init、spi_lookup；按照图 3.6，除了以上的接口函数外，SPI设备驱动还必须有一个中断服务函数，这里函数名命名为spi_INTService。下面一一介绍：

1、 SPI 写函数 spi_write

由2.4.3节知道，SPI写函数格式为：

```
Cyg_ErrNo spi_write (cyg_io_handle_t handle,void *buffer,cyg_uint32 *len)
{
}
```

根据图 3.6，在SPI的写函数中，主要完成的功能是把发送数据填入发送数据Buffer，并保证数据发送完成；在整个数据发送的过程中保证不能被其他任务占用SPI总线。SPI写函数流程如图 3.7。为了保证在整个数据发送的过程中SPI总线不被其他任务占用，进入写函数首先wait一次信号量（在初始化SPI时会post一次信号量，详见SPI初始化函数），这样在本次写的过程中，其他任务必须wait信号量，直到信号量被post。

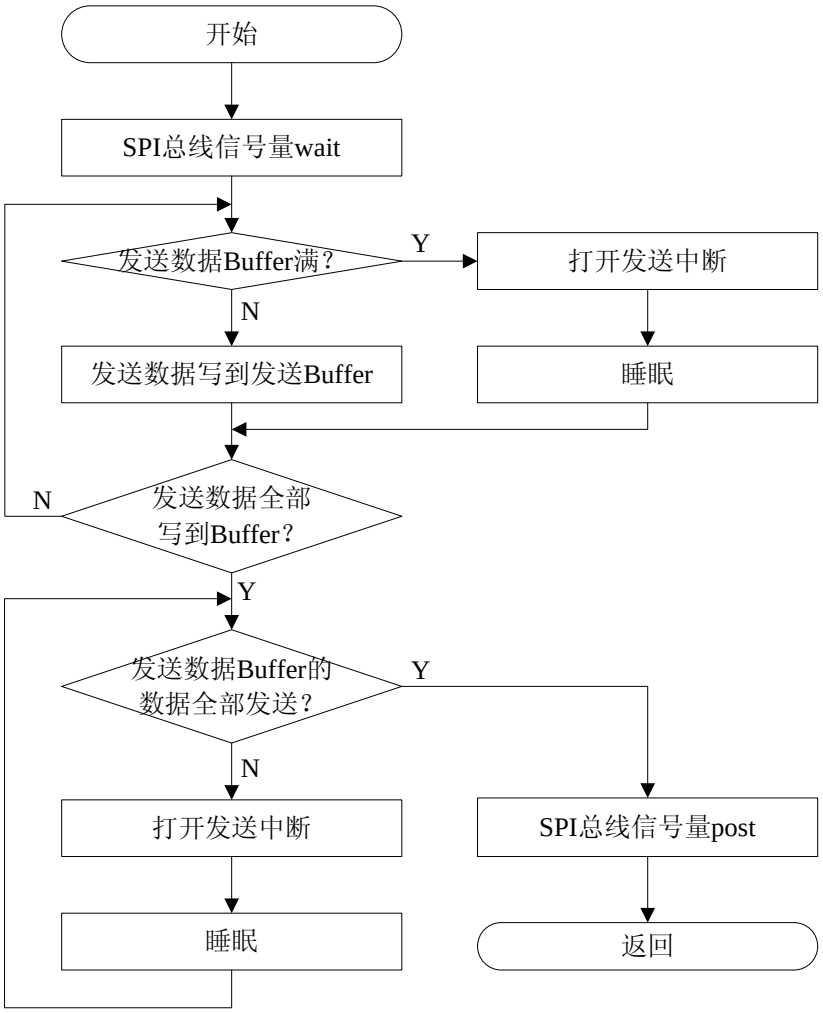

图 3.7 SPI 写函数流程图

参考程序段如下：

```
static Cyg_ErrNo spi_write(cyg_io_handle_t handle, void *buffer, cyg_uint32 *len)
{
    cyg_uint8 *bp = (cyg_uint8 *)buffer;
    cyg_uint8 tempdata;
    int tot = *len;

#if CYGNUM_DEVS_SPI_SPCE3200_DEBUG_MODE == 1
    printf("Debug - spi write\n");
#endif
    cyg_semaphore_wait(&sem_spi_data);                    // wait 信号量
```

```
        TxFlag = 1;
        while(tot>0)
        {
            if(_TxData_Num<MAX_EVENTS)              // 发送 Buffer 满？
            {
                tempdata = *bp;
                _Tx_buffer[_TxData_put] = tempdata;     // 否，发送数据写入 Buffer
                _TxData_put += 1;
                _TxData_Num += 1;
                bp += 1;
                tot -= 1;
                if(_TxData_put==MAX_EVENTS)
                    _TxData_put = 0;
            }
            else
            {
                *P_SPI_TX_STATUS |= C_SPI_TX_INTEN; // 使能发送中断
                cyg_thread_delay(2);                    // 睡眠
            }
        }
        while(_TxData_Num>0)                        // Buffer 的数据全部发送？
        {
            *P_SPI_TX_STATUS |= C_SPI_TX_INTEN;     // 否，使能中断
            cyg_thread_delay(2);                    // 睡眠
        }
        TxFlag = 0;
        *P_SPI_TX_STATUS &= ~C_SPI_TX_INTEN;        // 禁能中断
        cyg_semaphore_post(&sem_spi_data);          // 释放信号量
        return ENOERR;
    }
```

2、 SPI 读函数 spi_read

由2.4.3节知道，SPI读函数格式为：

```
Cyg_ErrNo spi_read(cyg_io_handle_t handle,void *buffer,cyg_uint32 *len)
{
}
```

根据图 3.6，在SPI的读函数中，主要完成的功能从接收数据Buffer中读数据，并保证读够需要的数据；在整个数据接收的过程中保证不能被其他任务占用SPI总线。和写函数类似，SPI读函数流程如图 3.8。

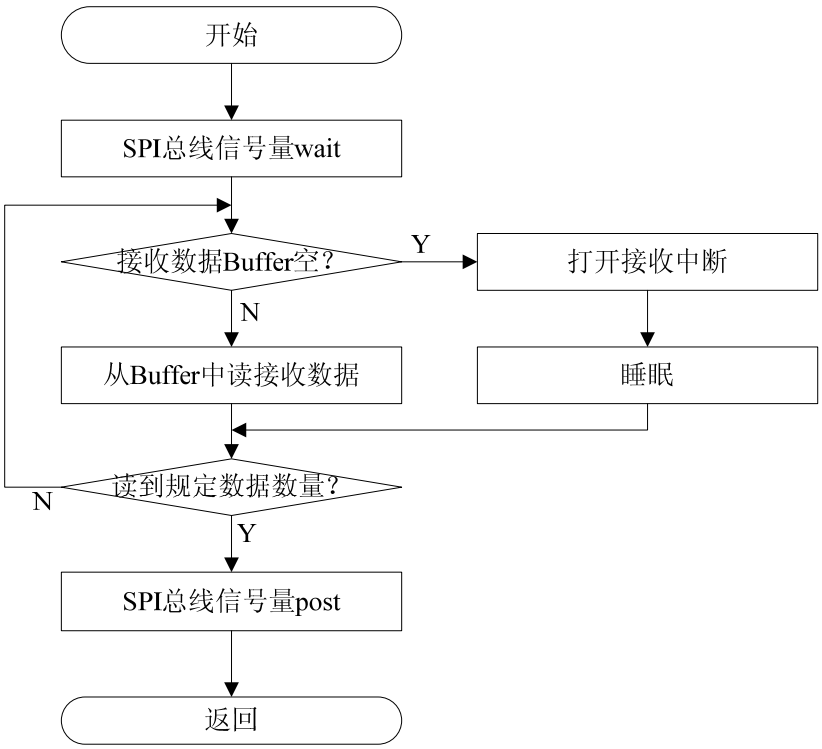

图 3.8 SPI 读函数流程图

参考程序段如下：

```
static Cyg_ErrNo spi_read(cyg_io_handle_t handle, void *buffer, cyg_uint32 *len)
{
    cyg_uint8 *bp = (cyg_uint8 *)buffer;
    cyg_uint8 tempdata;
    int tot = *len;

#if CYGNUM_DEVS_SPI_SPCE3200_DEBUG_MODE == 1
    printf("Debug - spi read\n");
#endif
```

```
        RxFlag = 1;
        cyg_semaphore_wait(&sem_spi_data);                  // wait 信号量

        while(tot>0)
        {
            if(_RxData_Num>0)                               // 接收 Buffer 非空
            {
                tempdata = _Rx_buffer[_RxData_get];         // 是，读接收数据
                *bp = tempdata;
                _RxData_get += 1;
                _RxData_Num -= 1;
                bp += 1;
                tot -= 1;
                if(_RxData_get==MAX_EVENTS)
                    _RxData_get = 0;
            }
            else                                            // 否
            {
                *P_SPI_RX_STATUS |= C_SPI_RX_INTEN;         // 使能中断
                cyg_thread_delay(2);                        // 睡眠
            }
        }
        cyg_semaphore_post(&sem_spi_data);                  // 释放信号量
        RxFlag = 0;
        return ENOERR;
    }
```

3、 SPI 设备配置函数 spi_set_config

由2.4.3节知道，SPI设置函数格式为：

```
Cyg_ErrNo spi_set_config(   cyg_io_handle_t handle,
                            cyg_uint32 key,
                            const void *buffer,
```

```
                                    cyg_uint32 *len
                                )
{
}
```

在 SPI 的设置函数中，当 SPCE3200 作为 SPI 总线的主机时，可以设置 SPI 总线时钟频率、时钟相位及时钟极性。参考程序段如下：

```
static Cyg_ErrNo iob_set_config(  cyg_io_handle_t handle,
                                  cyg_uint32 key,
                                  const void *buffer,
                                  cyg_uint32 *len
                                 )
{
    cyg_uint32 *bp = (cyg_uint32 *)buffer;
#if CYGNUM_DEVS_SPI_SPCE3200_DEBUG_MODE == 1
    printf("Debug - spi Set Config\n");
#endif

    cyg_uint32 spi_clock;
    cyg_uint32 spi_sph;
    cyg_uint32 spi_spo;
    cyg_uint32 spi_set;

    switch(key)
    {
        case CYGNUM_DEVS_SPI_SPCE3200_MASTER_CLOCK:// 设置时钟频率
        {
            spi_clock = *bp;
            spi_clock &= 0x07;
            spi_set = *P_SPI_MODE_CTRL;
            spi_set &= 0xfffffff8;
            spi_set |= spi_clock;
            *P_SPI_MODE_CTRL = spi_set;
```

```
                break;
            }
            case    CYGNUM_DEVS_SPI_SPCE3200_SPH:   // 设置时钟相位
            {
                spi_sph = *bp;
                spi_sph &= 0x80;
                spi_set = *P_SPI_MODE_CTRL;
                spi_set &= 0xffffff7f;
                spi_set |= spi_sph;
                *P_SPI_MODE_CTRL = spi_set;
                break;
            }
            case    CYGNUM_DEVS_SPI_SPCE3200_SPO:   // 设置时钟极性
            {
                spi_spo = *bp;
                spi_spo &= 0x40;
                spi_set = *P_SPI_MODE_CTRL;
                spi_set &= 0xffffffbf;
                spi_set |= spi_spo;
                *P_SPI_MODE_CTRL = spi_set;
                break;
            }
        }
        return EINVAL;
    }
```

4、 读 SPI 设备配置状态函数 spi_get_config

由2.4.3节知道，SPI读设置状态函数格式为：

```
Cyg_ErrNo spi_get_config(   cyg_io_handle_t handle,
                            cyg_uint32 key,
                            void *buf,
                            cyg_uint32 *len
```

```
                                )
{
}
```

在 SPI 的读设置状态函数中，读出目前 SPI 的模式状态。参考程序段如下：

```
static Cyg_ErrNo spi_get_config( cyg_io_handle_t handle,
                                 cyg_uint32 key,
                                 const void *buffer,
                                 cyg_uint32 *len
                                 )
{
    cyg_uint32 *bp = (cyg_uint32 *)buffer;
#if CYGNUM_DEVS_SPI_SPCE3200_DEBUG_MODE == 1
    printf("Debug - spi Get Config\n");
#endif
    cyg_uint32 spi_set_status;
    spi_set_status = *P_SPI_MODE_CTRL;
    *bp = spi_set_status;
    return EINVAL;
}
```

5、 SPI 初始化函数

由2.4.3节知道，SPI初始化函数格式为：

```
static bool spi_init(struct cyg_devtab_entry *tab)
{
}
```

SPI 初始化函数在 eCos 内核启动之前的硬件初始化函数里调用。SPI 初始化函数用来使能 SPI 模块时钟、SPI 接口及 SPI 模块，选择 SPI 的初始工作模式。参考程序段如下：

```
static bool iob_init(struct cyg_devtab_entry *tab)
{
#if CYGNUM_DEVS_SPI_SPCE3200_DEBUG_MODE == 1
    diag_printf("Debug - spi Init\n");
```

```
#endif
    *P_SPI_CLK_CONF = C_SPI_CLK_EN                  // SPI 时钟使能
                    | C_SPI_RST_DIS;                // SPI 模块不复位
    *P_SPI_INTERFACE_SEL |= C_SPI_PORT_SEL;         // 选择 SPI 复用接口为 SPI 功能
    *P_SPI_TXRX_STATUS |= C_SPI_AUTO_CLR;           // 自动清除中断标志模式
#ifdef CYGNUM_DEVS_SPI_SPCE3200_MODE_SELECT_SLAVE// 如果选择从模式
    *P_SPI_MODE_CTRL = C_SPI_SLAVE_MODE             // 设置为从模式
                    | C_SPI_CTRL_EN;                // SPI 接口控制器使能
#else                                               // 如果选择主模式
    *P_SPI_MODE_CTRL = C_SPI_CLK_27MDIV128          // 时钟频率选择为 27M 的 128 分频
                    | C_SPI_MASTER_MODE             // 设置主模式
                    | C_SPI_CTRL_EN                 // SPI 接口控制器使能
                    ;
#endif

    _TxData_get=_TxData_put=_RxData_get=_RxData_put=_TxData_Num=_RxData_Num=0;
    TxFlag = 0;
    RxFlag = 0;

    cyg_semaphore_init(&sem_spi_data,0);            // 初始化一个信号量
    cyg_semaphore_post(&sem_spi_data);              // 信号量值+1，唤醒等待信号量的线程
    return true;
}
```

6、 SPI 设备查找函数

由2.4.3节知道，SPI设备查找函数格式为：

```
static Cyg_ErrNo spi_lookup (   struct cyg_devtab_entry **tab,
                                struct cyg_devtab_entry *st,
                                const char *name
                            )
{
}
```

SPI 设备查找函数的主要功能是传递参数 tab，该函数在 cyg_ iob_lookup 中调用，用来查找是否存在 SPI 设备。参考程序段如下：

```
static Cyg_ErrNo iob_lookup (     struct cyg_devtab_entry **tab,
                                  struct cyg_devtab_entry *st,
                                  const char *name
                             )
{
#if CYGNUM_DEVS_SPI_SPCE3200_DEBUG_MODE == 1
      printf("Debug - spi Lookup\n");
#endif
if (!_SPIDrv_is_open)
      {
            _SPIDrv_is_open = true;
            cyg_drv_interrupt_create(43,                    // 创建 43 号中断
                                     1,
                                     ISR_DATA,
                                     spi_INTService,        // 中断服务程序为 spi_INTService
                                     NULL,
                                     &intr_handle,
                                     &intr);
            cyg_drv_interrupt_attach(intr_handle);          // 邦定中断向量
            cyg_drv_interrupt_unmask(43);                   // 允许 43 号中断向量的中断
      }
      return ENOERR;
}
```

在查找 SPI 设备的时候，创建 43 号 SPI 中断服务程序，并打开允许 SPI 的 43 号中断向量中断。这样，在查找到 SPI 设备，使能 SPI 中断后，当发送数据寄存器填入数据，立即触发 SPI 中断，发送数据；同样，使能 SPI 中断后，当有数据到达接收口，触发 SPI 中断，接收数据存储在数据寄存器中，在中断服务程序里即可以存储。

3.2.9 SPI驱动程序范例

和IOB的驱动源程序文件类似，SPI的驱动源程序文件包含如图 3.9所示的模块。

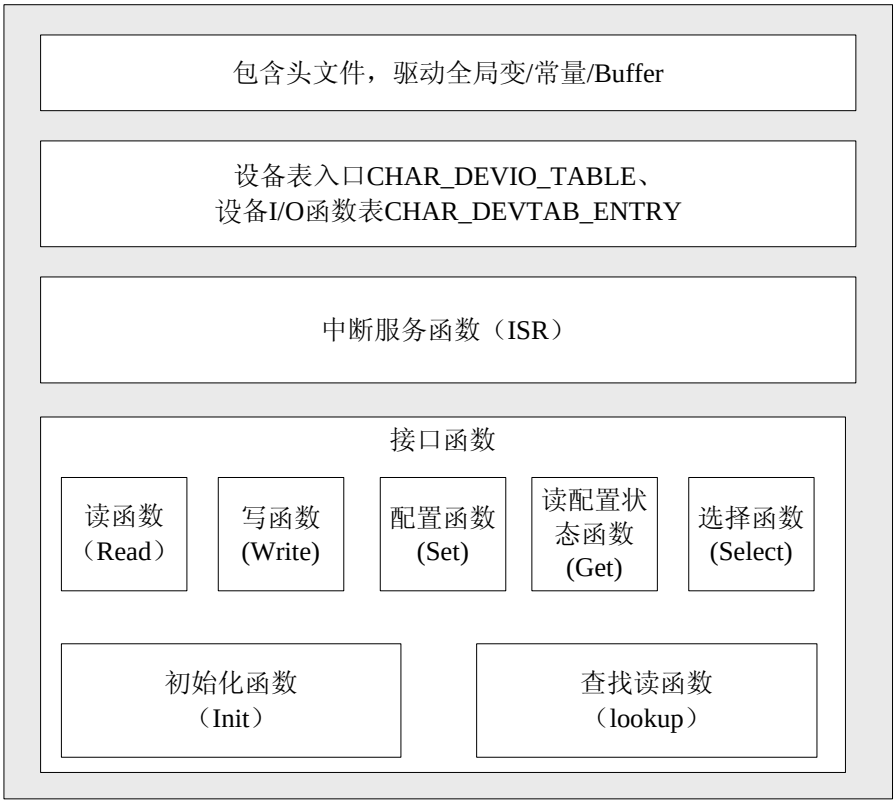

图 3.9 SPI 驱动源程序文件结构示意图

根据上两节设备表入口接口函数确定要包含的头文件和需要定义的全局变量，如图 3.9为需要定义的全部变量。

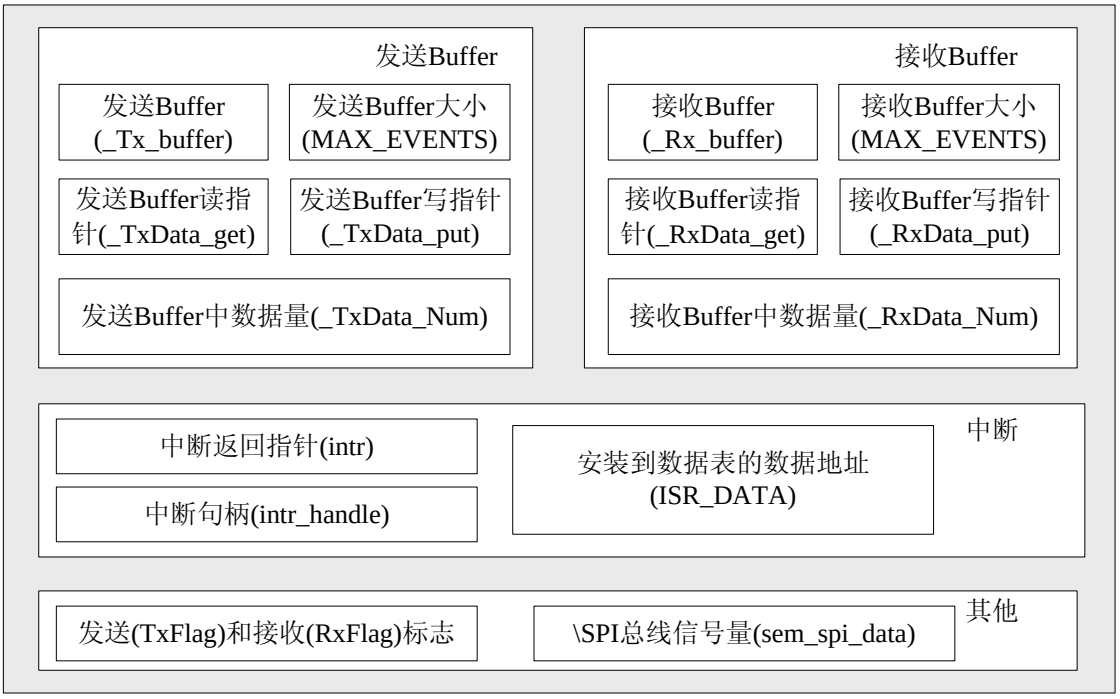

图 3.10 SPI 驱动程序资源全局变量/常量示意图

按照图 3.6所示的驱动程序结构，在逻辑层要建立一个发送数据Buffer和接收Buffer，同时在物理层对读发送Buffer中的数据进行发送，并且读接收数据存储到Buffer，所有这些读写操作都需要读写指针来指示；在eCos系统中利用中断时需要中断的句柄、中断的返回地址和中断安装到数据表的数据地址；为了保证SPI总线不被其他任务占用需要同步机制，如信号量。所有这些变量需要进行全局定义。另外，由于SPI通信为双工通信，无论发送或者接收数据都会触发发送和接收中断，为了判断正在进行的操作是发送还是接收，需要定义发送和接收的标志来区分。

图 3.11SPI驱动源程序文件中需要包含的头文件。

接口函数或者设备表入口中的函数或宏	声明或者定义头文件	头文件所在编译路径
CHAR_DEVIO_TABLE()、CHAR_DEVTAB_ENTRY()	devtab.h	cyg/io/devtab.h
CYGDAT_DEVS_SPI_SPCE3200_NAME、CYGNUM_DEVS_SPI_SPCE3200_DEBUG_MODE等	devs_spi_spce3200.h	pkgconf/devs_spi_spce3200.h
cyg_uint32	cyg_type.h	cyg/infra/cyg_type.h
printf()	stdio.h	stdio.h
P_SPI_TX_STATUS、P_SPI_TX_DATA等硬件寄存器	SPCE3200_Register.h	cyg/hal/SPCE3200_Register.h
C_SPI_TX_FLAG、C_SPI_TX_INTEN等常量	SPCE3200_Constant.h	cyg/hal/SPCE3200_Constant.h
ENOERR	codes.h	cyg/error/codes.h
CYGNUM_DEVS_SPI_SPCE3200_MASTER_CLOCK CYGNUM_DEVS_SPI_SPCE3200_SPH CYGNUM_DEVS_SPI_SPCE3200_SPO	iob_spce3200.h	cyg/io/iob_spce3200.h
diag_printf()	diag.h	cyg/infra/diag.h
CYG_TEST_CHECK()	testcase.h	cyg/infra/testcase.h
HAL_INTERRUPT_ACKNOWLEDGE()	hal_intr.h	cyg/hal/hal_intr.h
cyg_drv_interrupt_create()、cyg_drv_interrupt_attach、cyg_drv_interrupt_unmask	drv_api.h	cyg/hal/drv_api.h

图 3.11 SPI 驱动源程序文件需要包含的头文件示意图

如下为一个完整的 SPI 设备驱动源程序文件的伪代码：

```
//---------------------------------------包含相关头文件---------------------------//
#include <pkgconf/hal.h>                    // hal 的宏头文件，由系统自动生成
#include <pkgconf/devs_spi_spce3200.h>      // spi 驱动 cdl 文件宏头文件，由系统自动生成


#include <cyg/infra/cyg_type.h>             // 定义类型
#include <cyg/hal/hal_intr.h>               // 定义中断相关宏
```

```
#include <cyg/hal/drv_api.h>                    // 驱动程序头文件
#include <cyg/hal/SPCE3200_Register.h>   // 定义 SPCE3200 硬件寄存器
#include <cyg/hal/SPCE3200_Constant.h>  // 定义 SPCE3200 常量
#include <cyg/error/codes.h>                    // 定义错误常量，如 ENOERR
#if CYGNUM_DEVS_SPI_SPCE3200_DEBUG_MODE == 1
    #include <cyg/infra/testcase.h>          // 测试宏定义，如 CYG_TEST_CHECK
    #include <cyg/infra/diag.h>              // diag 函数声明
    #include <stdio.h>                       //调试函数如 printf 等声明
#endif
#include <cyg/io/devtab.h>                      // 设备 I/O 入口表定义及声明
#include <cyg/io/spce3200_spi.h>            // 定义 SPCE3200 的 SPI 模块相关常量，这里含 key
//--------------------------------------------------------------------------//
// Functions in this module
static Cyg_ErrNo spi_write(cyg_io_handle_t handle, void *buffer, cyg_uint32 *len);
static Cyg_ErrNo spi_read(cyg_io_handle_t handle, void *buffer, cyg_uint32 *len);
static Cyg_ErrNo spi_set_config(cyg_io_handle_t handle,
                                cyg_uint32 key,const void *buffer,cyg_uint32 *len);
static Cyg_ErrNo spi_get_config(cyg_io_handle_t handle,
                                cyg_uint32 key,const void *buffer,cyg_uint32 *len);
static bool spi_init(struct cyg_devtab_entry *tab);
static Cyg_ErrNo spi_lookup (struct cyg_devtab_entry **tab,
                             struct cyg_devtab_entry *st,const char *name);

CHAR_DEVIO_TABLE(…);
CHAR_DEVTAB_ENTRY(…);

static cyg_uint32 spi_INTService( cyg_uint32 vector, CYG_ADDRWORD data )
{
    …
}
static Cyg_ErrNo spi_write(cyg_io_handle_t handle, void *buffer, cyg_uint32 *len)
{
```

设备表入口详见 3.2.6

函数体详见 3.2.7IOB 中断服务(ISR)函数

```
        …
    }
    static Cyg_ErrNo spi_read(cyg_io_handle_t handle, void *buffer, cyg_uint32 *len)
    {
        …
    }

    static Cyg_ErrNo spi_set_config(cyg_io_handle_t handle, cyg_uint32 key, const void *buffer,
cyg_uint32 *len)
    {
        …
    }

    static Cyg_ErrNo spi_get_config(cyg_io_handle_t handle, cyg_uint32 key, void *buffer, cyg_uint32
*len)
    {
        …
    }

    static bool spi_init(struct cyg_devtab_entry *tab)
    {
        …
    }

    static Cyg_ErrNo spi_lookup(struct cyg_devtab_entry **tab, struct cyg_devtab_entry *st, const char
*name)
    {
        …
    }
```

函数体详见 3.2.8 IOB 写函数

函数体详见 3.2.8 IOB 读函数

函数体详见 3.2.8 IOB 设置函数

函数体详见 3.2.8 IOB 读设置状态函数

函数体详见 3.2.8 IOB 初始化函数

函数体详见 3.2.8 IOB 设备查找函数

3.2.10 向eCos数据库中添加SPI驱动程序组件包

根据1.4.1加载驱动程序组件包到数据库中办法，在ecos.db中的任意位置加如下程序段：

```
package CYGPKG_DEVS_SPI_SPCE3200 {
    alias          { "SPI support for Score SPCE3200"}
    directory  devs/spi/score/spce3200
    script         spi_spce3200.cdl
        description "
            This package contains hardware support for the SPI
        on the Score SPCE3200 DEV Board."
}
```

这里假设SPI驱动程序组件包的目录结构如图 3.5。

此时如果用eCos配置工具添加包时，就会发现SPI驱动程序组件包已经出现在eCos数据库中，如图 3.4：点击“Add”加载到平台上进行编译生成*.ecc库文件后，即可以使用SPI驱动程序。

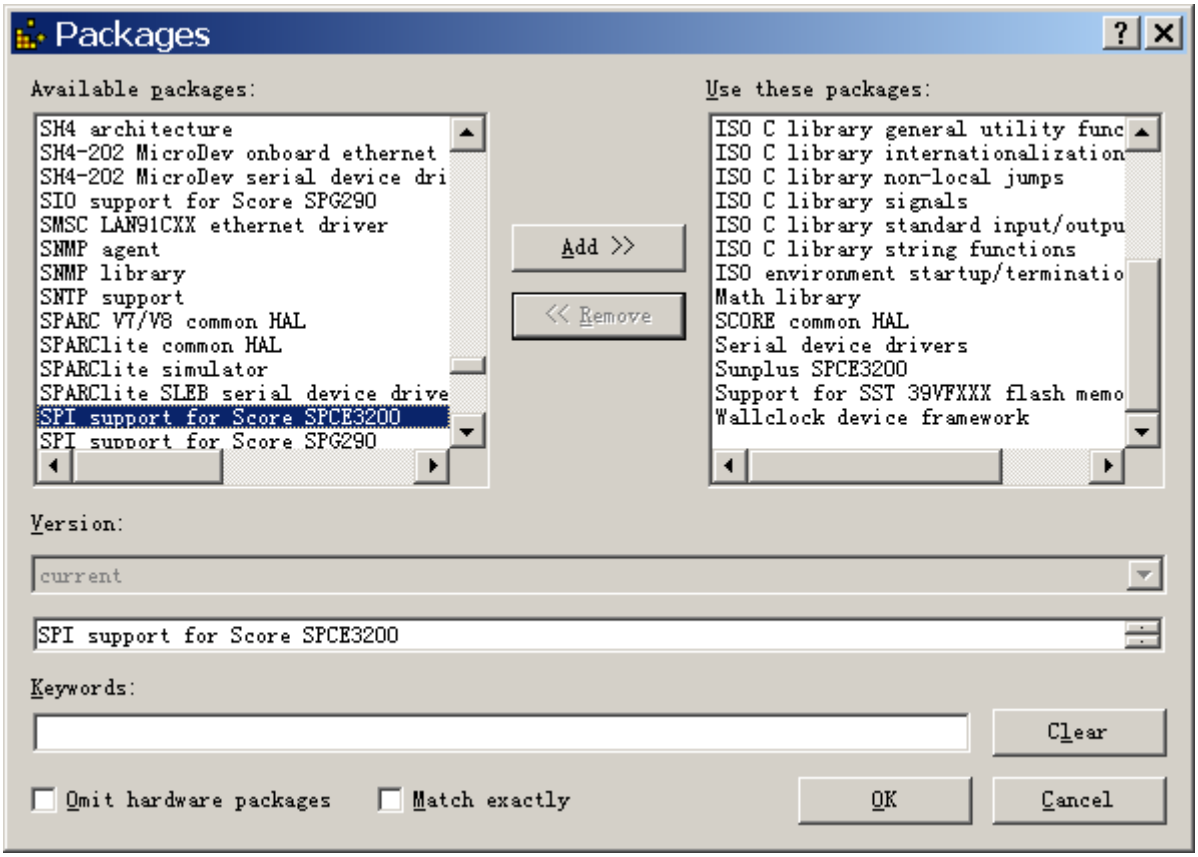

图 3.12 SPI 驱动程序组件包在 eCos 数据库中界面

3.2.11 用户测试程序

在把 SPI 设备驱动程序组件包加载到 SPCE3200 开发平台编译没有错误后，就可以编写测试程序验证驱动程序是否工作正常。

为了方便测试，直接把 SPCE3200 的发送脚与接收脚连接起来。

分析：该测试程序可以借助串口调试助手或者超级终端来观察发送和接收数据，发送一个数据观察接收的数据是否正确。

参考程序如下：

```
//-----------------------------------包含相关头文件---------------------------------//
#include <cyg/infra/testcase.h>          // 测试宏定义，如 CYG_TEST_FAIL_FINISH
#include <cyg/io/io.h>                   // 声明 I/O 子系统 API 函数
#include <cyg/kernel/kapi.h>             // 定义针对内核的 API 函数，同步机制的 API 函数等
#include <stdio.h>                       // 声明 printf 等调试函数
// #include <cyg/io/spce3200_spi.h>      // 定义 SPCE3200 的 SPI 模块相关常量，这里含 key

cyg_io_handle_t    hDrvSPI;              // 定义 SPI 线程的句柄
void spi_thread(cyg_addrword_t data)     // 通过 SPI 线程进行发送和接收数据
{
    cyg_uint32          len = 1;
    cyg_uint32          RxData[1];
    cyg_uint32          TxData[1];

    TxData[0] = 66;                      //  发送数据 66

    if (ENOERR != cyg_io_lookup("/dev/spi", &hDrvSPI))
    {                                    //  查找 spi 设备
        CYG_TEST_FAIL_FINISH("Error opening /dev/spi");
    }
    printf("Open /dev/spi OK\n");        // 查找成功，打印成功信息
    while(true)
    {
        len = 1;
        if(ENOERR != cyg_io_write(hDrvSPI, TxData, &len))
        {                                // 调用 cyg_io_write 发送数据
            CYG_TEST_FINISH("Error write /dev/spi");
        }
```

```
            printf("SPI TxNum = %d\n",len);  // 返回实际发送数据量
            if(len != 0)
            {                                // 发送成功，打印发送数据信息
                printf("SPI TxData = %d\n", TxData[0]);
            }
            if(ENOERR != cyg_io_read(hDrvSPI, RxData, &len))
            {                                // 调用 cyg_io_ read 接收数据
                CYG_TEST_FINISH("Error Read /dev/spi");
            }
            if(len != 0)
            {                                // 接收成功，打印接收数据信息
                printf("SPI RxData = %d\n", RxData[0]);
            }
            cyg_thread_delay(10);            // 线程睡眠
        }
    }

    cyg_uint8 stack[4096];                   // 线程堆栈
    cyg_handle_t simple_threadA;             // 线程句柄
    cyg_thread thread_s;                     // 线程信息

    void cyg_user_start(void)
    {
        CYG_TEST_INIT();
        cyg_thread_create( 4,                // 创建线程 spi_thread
                           spi_thread,
                           (cyg_addrword_t) 0,
                           "can0_thread",
                           (void *) stack, 4096,
                           &simple_threadA,
                           &thread_s);
        cyg_thread_resume(simple_threadA);
```

```
        cyg_scheduler_start();
}
```

4 块设备驱动程序初步

4.1 块设备驱动程序概述

块设备是指数据传输以块为单位的（例如软盘、SD 卡以及硬盘等）设备，这里硬件的块一般被称作“扇区（Sector）”。

在 eCos 系统中，对块设备的管理与对字符设备的管理基本是一致的，只是在读写操作上有所差异，另外，块设备一般来说比字符设备有更多的配置参数。

块设备常常被用来作为文件系统的宿主，文件系统通过 eCos 内核提供的接口操作这些块设备，从而提供给用户更为方便的操作方法。

与字符设备类似，在 eCos 系统中，块设备同样依靠设备表入口以及设备 I/O 函数表将自己注册到 eCos 系统中。只不过，块设备的设备入口表名和设备 I/O 函数表的形式与字符设备有些差异。

4.2 块设备驱动程序设计

与字符设备驱动程序的设计方法和步骤类似，块设备驱动程序同样包括以下几个基本步骤：建立驱动程序目录及文件、设计 CDL 文件、编写设备表入口、实现设备接口函数、向组件库添加驱动程序包等。另外，由于块设备的特殊性，经常将他们做为文件系统的宿主，所以，在基本的驱动程序设计完成之后，用户还可以选择将块设备添加到文件系统的管理范围之内。

下面主要介绍块设备驱动程序区别于字符设备的设计步骤。

4.2.1 设计CDL脚本文件

CDL 文件为设备定义了一系列的编译属性和行为，并可以根据用户的需要对编译进行源码级控制。块设备的 CDL 文件的设计与字符设备的 CDL 文件的设计方法一致，典型的块设备的 CDL 文件的写法如下所示，这里不再做详细解释。

```
cdl_package CYGPKG_DEVS_DEV1_SPCE3200 {
    display       "Dev1 driver for SPCE3200"
    include_dir   cyg/dev1
    compile           -library=libextras.a dev1_spce3200.c
    description "Dev1 driver for SCORE SPCE3200"

    cdl_component CYGPKG_DEVS_DEV1_SPCE3200_OPTIONS {
        display "Compile Options"
        flavor   none
        no_define

        cdl_option CYGPKG_DEVS_DEV1_SPCE3200_CFLAGS {
            display          "Additional compiler flags"
            flavor           data
```

```
                no_define
                default_value { "" }
                description "
                    This option modifies the set of compiler flags for
                    building the keypad driver package. These flags
                    are used in addition to the set of global flags."
            }

            cdl_option CYGDAT_DEVS_DEV1_SPCE3200_NAME {
                display "Device name for the dev1 driver"
                flavor data
                default_value {"\"/dev/dev1\""}
                description " This option specifies the name of the dev1 device"
            }
        }
    }
```

4.2.2 编写设备表入口

使用 BLOCK_DEVTAB_ENTRY 宏可以描述一个块设备，该宏的完整格式为：

```
BLOCK_DEVTAB_ENTRY(_l, _name, _dep_name, _handlers, _init, _lookup, _priv);
```

其中：

- _l：该设备表入口的标识符，即本设备表的句柄
- _name：设备的名称，一般是以“/dev/”开头的字符串，如“/dev/dev1”
- _dep_name：对于层次设备，此参数为该设备所依赖的底层设备的名称
- _handlers：设备的 I/O 函数表的句柄指针
- _init：设备的初始化函数，当 eCos 处于初始化阶段时将调用此函数以便对该设备进行初始化。用户需要编写该函数以便对设备进行初始化，或完成一些针对该设备的预处理工作。_init 为初始化函数名，用户可以自定义该函数名，习惯上该函数名定义为设备名+‘_’+init 的格式，比如某一设备的设备名为 dev1，该函数名习惯上定义为 dev1_init
- _lookup：设备的查找函数，当使用 cyg_io_lookup()函数对该设备进行查找时将调用此函数。用户需要编写该函数以便在应用程序尝试查找该设备时作出相应处理。_lookup 同样为查找函数的函数名，也可自定义，设备名为 dev1 的设备查找函数名习惯上定义为 dev1_lookup。
- _priv：设备驱动程序所需的专用数据的存放位置

设备表入口中的句柄_handlers 提供了一组块设备驱动程序接口函数。_handlers 是块设备 I/O 操作函数表 BLOCK_DEVIO_TABLE 的指针，BLOCK_DEVIO_TABLE 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，也是应用程序访问块设备的接口。

块设备 I/O 函数表通过 BLOCK_DEVIO_TABLE 宏来定义，其格式如下：

```
BLOCK_DEVIO_TABLE(_l,_bwrite,_bread,_select,_get_config,_set_config)
```

其中：

- _l：设备 I/O 函数表的标识符，即本函数表的句柄
- _bwrite：cyg_io_bwrite()函数所调用的函数，实现向块设备传送数据。用户一般需要编写该函数，完成数据的写操作。与_init 类似，_bwrite 也为函数名，用户可以自定义
- _bread：cyg_io_bread()函数所调用的函数，实现从块设备读取数据。用户一般需要编写该函数，完成数据的读操作。与_init 类似，_bread 为函数名，用户可以自定义
- _select：cyg_io_select()函数所调用的函数。与_init 类似，_select 为函数名，用户可以自定义
- _get_config：cyg_io_get_config()函数所调用的函数，完成对设备配置信息的读取操作。用户可以编写该函数，用以向应用程序返回特定配置信息。与_init 类似，_get_config 为函数名，用户可以自定义
- _set_config：cyg_io_set_config()函数所调用的函数，完成对设备的配置操作。用户可以编写该函数，用以接受应用程序对设备的配置。与_init 类似，_set_config 为函数名，用户可以自定义

设备表入口以及设备 I/O 函数表通常与设备接口函数的实现代码放在同一个文件，在介绍接口函数的实现时将举例说明。

4.2.3 实现设备接口函数

上面提到，在设备表入口以及设备 I/O 函数表中都包含了一组函数的指针，这些函数就是设备驱动接口函数。在完成设备表入口以及设备 I/O 函数表之后，用户需要逐个实现这些函数，包括：

1. 设备初始化函数

块设备初始化函数同样在内核启动期间被调用，用以对设备进行初始化操作。设备初始化函数的完整形式为：

bool 函数名(struct cyg_devtab_entry *tab);

该函数由用户编写，内核在启动期间对其进行调用，用以对设备做初始化动作。

其中：

- 函数名应与 BLOCK_DEVTAB_ENTRY 宏中定义的初始化函数名一致，从而内核可以依靠设备表入口找到这个函数
- tab：指向当前设备的指针，用户可以通过该指针对当前设备进行查找操作

2. 设备查找函数

当应用程序使用 cyg_io_lookup()函数打开设备时，内核将调用该函数。在该函数中用户一般需要建立线程以便对设备进行查询读写操作，或开启中断服务程序来操作设备。设备查找函数的完整形式为：

Cyg_ErrNo 函数名(struct cyg_devtab_entry **tab, struct cyg_devtab_entry *sub_tab, const char *name);

其中：

- 函数名应与 BLOCK_DEVTAB_ENTRY 宏中定义的查找函数名一致，从而内核可以依靠设备表入口找到这个函数
- tab：指向当前设备的指针，用户可以通过该指针对当前设备的信息进行读写操作
- sub_tab：指向当前设备所依赖的下层设备的指针，用户可以通过该指针对下层设备进行操作。在 lookup 设备的时候，如果有下层设备，cyg_io_lookup()函数中会同时打开下层设备并通过参数 sub_tab 返回该设备的指针。
- name：应用程序希望查找的设备名比当前设备名多出的部分。如，应用层希望打开“/dev/dev1”，而系统中只有设备“/dev/dev”，内核会将此设备做为最佳匹配设备打开，并将名称中多出的“1”字符串传递给该函数

3. 设备批量写操作函数

当应用程序使用 cyg_io_bwrite()函数对设备进行写操作时，内核将调用该函数。在该函数中用户需要实现对设备的批量写操作。设备批量写操作函数的完整形式为：

Cyg_ErrNo 函数名(void *disk, const void *buf_arg, cyg_uint32 *blocks, cyg_uint32 first_block);

其中：

- 函数名应与 BLOCK_DEVIO_TABLE 宏中定义的批量写操作函数名一致，从而内核可以依靠 I/O 函数表找到这个函数
- disk：应用程序操作设备的设备句柄
- buf_arg：缓冲区地址指针，用户需要从该指针指向的缓冲区内读取数据并写入到设备
- blocks：写入到设备数据的块数量。函数返回时该参数包含实际写入块的数量
- first_block：写入数据的起始块号

4. 设备批量读操作函数

当应用程序使用 cyg_io_bread()函数对设备进行读操作时，内核将调用该函数。在该函数中用户需要实现对设备的读操作。设备批量读操作的完整形式为：

Cyg_ErrNo 函数名(void *disk, void *buf_arg, cyg_uint32 *blocks, cyg_uint32 first_block);

其中：

- 函数名应与 BLOCK_DEVIO_TABLE 宏中定义的批量读操作函数名一致，从而内核可以依靠 I/O 函数表找到这个函数
- disk：应用程序操作设备的设备句柄
- buf_arg：缓冲区地址指针，用户需要从设备中读取数据并将数据写入该指针指向的缓冲区内
- blocks：读取设备数据的块数量。函数返回时该参数包含实际读出块的数量
- first_block：读取数据的起始块号

5. 设备检测函数

当应用程序使用 cyg_io_select()函数，内核将调用该函数进行设备的可用性检测。

cyg_bool 函数名(void *disk, cyg_uint32 which, CYG_ADDRWORD info);

- 函数名应与 CHAR_DEVIO_TABLE 宏中定义的选择函数名一致，从而内核可以依靠 I/O 函数表找到这个函数
- disk：应用程序操作设备的设备句柄
- which：选择类型，一般在利用文件系统操作设备时，which 为文件类型只读、只写等
- info：选择信息，例如等待标志等选择信息。

6. 读设备配置状态函数

当应用程序使用 cyg_io_get_config()函数时，内核将调用该函数。在该函数中用户一般需要实现为应用程序提供与设备相关的配置信息的功能。读设备配置函数的完整形式为：

Cyg_ErrNo 函数名(disk_channel* chan, cyg_uint32 key, const void* buf, cyg_uint32* len);

其中：

- 函数名应与 BLOCK_DEVIO_TABLE 宏中定义的读设备配置状态函数名一致，从而内核可以依靠 I/O 函数表找到这个函数
- chan：应用程序操作设备的设备句柄
- key：得到信息的类型。每个驱动程序的 key 值可能不同，在<io/common/include>文件夹下的 config_keys.h 文件中定义了一些常用的 key 值，也可以在自定义的头文件中定义，但是注意不要和 config_keys.h 中的 key 值重复
- buf：放置配置数据的缓冲区指针
- len：得到的配置信息的长度。函数返回时，该参数应该包含实际得到的数据大小

7. 设备配置函数

当应用程序使用 cyg_io_set_config()函数时，内核将调用该函数。在该函数中用户一般需要实现根据应用程序提供的参数对设备进行配置的功能。写设备配置函数的完整形式为：

Cyg_ErrNo 函数名(disk_channel* chan, cyg_uint32 key, const void* buf, cyg_uint32* len);

其中：

- 函数名应与 BLOCK_DEVIO_TABLE 宏中定义的设备配置函数名一致，从而内核可以依靠 I/O 函数表找到这个函数
- chan：应用程序操作设备的设备句柄
- key：设置信息的类型。每个驱动程序的 key 值可能不同，通常包含在<io/common/include>文件夹下的 config_keys.h 文件中或者设备驱动程序的头文件中
- buf：放置配置数据的缓冲区指针
- len：配置信息的长度。函数返回时，该参数应该包含实际设置的数据大小

4.2.4 块设备驱动程序解析

下面给出了最简单的块设备驱动程序的完整代码，这个代码完整的实现了块驱动设备源程序的各个部分，包括设备表入口以及设备 I/O 函数表的实现。接下来的块驱动程序设计过程中，可以把它当作模版。这里假定设备名为 dev1。

```
#include <pkgconf/devs_xxx_xxxx.h>
```

该文件与 CDL 文件同名，包含了在 CDL 里定义的编译选项的宏定义

```
#include <cyg/infra/cyg_type.h>                    // 定义类型，如 cyg_uint32
//-----------------------------------------设备接口函数-----------------------------------------//
static cyg_bool dev1_init(struct cyg_devtab_entry* tab)
{
    // Add initial code here

    return true;
}
static cyg_bool dev1_lookup(struct cyg_devtab_entry* tab)
{
    // Add lookup code here

    return ENOERR;
}
static Cyg_ErrNo dev1_ReadSector(void* disk, void* buf_arg, cyg_uint32 *blocks, cyg_uint32
first_block)
{
    // Add Read Sector Code here

    return ENOERR;
}
static cyg_ErrNo dev1_WriteSector(void *disk, const void *buf_arg, cyg_uint32 *blocks, cyg_uint32
first_block)
{
    // Add Write Sector code here

    return ENOERR;
}
static cyg_bool dev1_select(cyg_io_handle_t handle, cyg_uint32 which, cyg_addrword_t info)
{
    // Add Select code here

    return true;
}
static cyg_ErrNo dev1_GetConfig(disk_channel* chan, cyg_uint32 key, const void* buf, cyg_uint32*
len)
{
```

设备初始化函数，在内核启动过程中将被调用

查找函数

批量读函数

批量读函数

选择函数

读设备配置状态函数

```
        // Add Get Confit code here

        return ENOERR;
    }
    static Cyg_ErrNo dev1_SetConfig(disk_channel* chan, cyg_uint32 key, const void* buf, cyg_uint32*
len)
    {
        // Add Set Confit code here

        return ENOERR;
    }
    //-----------------------------------------定义设备表入口-----------------------------------------//
    BLOCK_DEVIO_TABLE(SPCE3200_dev1_handlers,
                        dev1_WriteSector,
                        dev1_ReadSector,
                        Null,
                        dev1_GetConfig,
                        dev1_GetConfig
    );
    BLOCK_DEVTAB_ENTRY(SPCE3200_ dev1_device,
                        CYGDAT_DEVS_ DEV1_SPCE3200_NAME, //  该宏来自于 cdl 文件中的配置
                        NULL,                                   //  底层设备名
                        &SPCE3200_ dev1_handlers,
                        dev1_init,
                        dev1_lookup,
                        NULL                                    //
    );
    //-------------------------------------------------------------------------------------------------//
```

设备配置函数

4.2.5 向eCos数据库中添加驱动程序包

写好了 CDL 文件后，设备便以包的形式存在，并具有了一些编译选项。接下来就要将这个包添加到 ecos 的组件数据库中，以便 configtools 工具可以识别到这个包，并可以将其加入到编译的队列中。

仍然以上面的 dev1 设备为例，典型的注册包的代码如下所示：

```
    package CYGPKG_DEVS_DEV1_SPCE3200 {
        alias           { "Dev1 support for Score SPCE3200"}
```

```
        directory       devs/dev1/score/spce3200
        script          dev1_spce3200.cdl
            description "
                This package contains hardware support for the Dev1
            on the Score SPCE3200 EV Board."
    }
```

4.3 块设备驱动程序的使用

块设备可以像字符设备那样，首先使用 cyg_lookup()函数来打开设备，然后使用读写函数来对设备进行基本的读写操作。与块设备相关的 I/O 接口函数有：

1、Cyg_ErrNo cyg_io_lookup(const char *name, cyg_io_handle_t *handle);

设备查找函数，与字符设备一致。

2、Cyg_ErrNo cyg_io_bwrite(cyg_io_handle_t handle, const void *buf, cyg_uint32 *len, cyg_uint32 pos);

块设备批量写函数。

- handle 指定设备的句柄
- buf 指定缓冲区的首地址
- len 指定写设备的块数量
- pos 指定写设备的起始块

3、Cyg_ErrNo cyg_io_bread(cyg_io_handle_t handle, void *buf, cyg_uint32 *len, cyg_uint32 pos);

块设备批量读函数。

- handle 指定设备的句柄
- buf 指定缓冲区的首地址
- len 指定读设备的块数量
- pos 指定了读设备的起始块

4、Cyg_ErrNo cyg_io_get_config(cyg_io_handle_t handle, cyg_uint32 key, void *buf, cyg_uint32 *len);

读设备配置状态函数，与字符设备一致。

5、Cyg_ErrNo cyg_io_set_config(cyg_io_handle_t handle, cyg_uint32 key, const void *buf, cyg_uint32 *len);

设备配置函数，与字符设备一致。

6、cyg_bool cyg_io_select(cyg_io_handle_t handle, cyg_uint32 which, CYG_ADDRWORD info);

设备检测函数，与字符设备一致。

下面给出一个对块设备进行读写的范例代码：

```
cyg_io_handle_t     hDrvDEV1;
void dev1_read_thread(cyg_addrword_t data)
```

必须先建立一个线程 dev1_read_thread

```
{
    cyg_uint32              len;
    cyg_uint8               ReadData [512 * 4];

    if (ENOERR != cyg_io_lookup("/dev/dev1", &hDrvDEV1))
    {
        CYG_TEST_FAIL_FINISH("Error opening /dev/dev1");
    }
    printf("Open /dev/dev1 OK\n");
    len = 4;                                    // Read 4 Blocks
    if(ENOERR != cyg_io_bread(hDrvDEV1, ReadData, &len, 0))
    {
        CYG_TEST_FIAL_FINISH(“Err reading /dev/dev1”);
    }
    printf(“Read 4 blocks from /dev/dev1 OK\n”);
    while(true);
}
```

查找设备 dev1

读取从 0 扇区开始的 4 个扇区

除了基本的读写操作，块设备经常做为文件系统的宿主，用户通过文件系统间接访问块设备。针对不同的块设备，用户可以选择不同的文件系统，比如，SD 卡设备上可以使用 FAT 文件系统来管理，Nand 型 Flash 一般选择 jffs 文件系统来管理等等。关于块设备与文件系统配合使用的方法在下面的章节介绍具体的设备驱动程序编写时将做详细介绍。

5 基于底层设备的设备驱动程序设计

上面介绍了一般字符设备和块设备的驱动程序编写，而对于某些设备来说，是基于其他设备的。比如把 6 个 LED 分别接在 IOB 口的六个口上，如果这时候把 LED 作为一个设备来说，LED 设备就是基于 IOB 设备的 eCos 设备，这时候 IOB 设备称作底层设备，LED 设备称作基于底层设备的设备。通常这些设备在 eCos 设备中占一大部分。

本章以 JoyStick 的驱动程序为例讲解基于底层设备的设备驱动程序设计。基于底层设备的驱动程序结构和设计步骤与一般的字符设备及块设备类似，只是设计过程基本上是在已有的底层设备驱动程序上进行的。本章只介绍区别于其他设备的部分。

JoyStick 也是字符设备，所不同的是 JoyStick 是基于底层设备的字符设备。

目前在 SPCE3200 两个开发系统上都有专门的 JoyStick 接口，实验仪上有一个 JoyStick 接口，其时钟信号 CLK、LOAD 信号、数据信号 DATA 分别于作为 GPIO 使用时的 ADC_CH0~2 连接；实验箱上有两个 JoyStick 接口，两个时钟信号和 LOAD 信号共用，分别与 IOB0 和 IOB1 连接，两个 JoyStick 的数据信号 DATA0 和 DATA1 分别与 IOB2、IOB3 连接。这里以实验箱的 JoyStick 设备为例。

实验箱的JoyStick设备是以IOB设备为底层设备的，IOB设备的驱动程序在第3章已经学习。

5.1 设计CDL脚本文件

JoyStick 设备的脚本文件设计方法和字符设备及块设备类似，先要建立一个 JoyStick 驱动程序的组建包，在组建包中要定义配置选项宏，同时，设备驱动基于 IOB 设备，所以要通过 active_if 属性定义与 IOB 设备驱动的依赖关系，参考程序如下：

```
cdl_package CYGPKG_DEVS_JOYSTICK_SPCE3200 {
    active_if       CYGPKG_DEVS_IOB_SPCE3200
    display         "JoyStick driver for SPCE3200"
    hardware
    include_dir     cyg/io
    requires        CYGPKG_IO
    compile         -library=libextras.a joystick_spce3200.c
    description "JoyStick driver for the SPCE3200"

    cdl_component CYGPKG_DEVS_JOYSTICK_SPCE3200_OPTIONS {
        display "Compile Options"
        flavor  none
        no_define

        cdl_option CYGPKG_DEVS_JOYSTICK_SPCE3200_CFLAGS {
            display         "Additional compiler flags"
            flavor          data
            no_define
```

当 IOB 驱动组件包有效时该包有效

```
            default_value { "" }
            description "
                This option modifies the set of compiler flags for
                building the JoyStick driver package. These flags
                are used in addition to the set of global flags."
        }

        cdl_option CYGDAT_DEVS_JOYSTICK_SPCE3200_NAME {
            display "Device name for the JoyStick driver"
            flavor data
            default_value {"\"/dev/JoyStick\""}
            description " This option specifies the name of the JoyStick device"
        }

        cdl_option CYGNUM_DEVS_JOYSTICK_SPCE3200_DEBUG_MODE {
            display "Debug Message"
            default_value    0
            description "
                This option will enable the debug message outputing if set to 1,
                will disable the outputing if set to 0."
        }
    }

    cdl_component CYGHWR_DEVS_JOYSTICK_SPCE3200_OPTIONS {
        display "Connection Options"
        flavor    none
        no_define

        cdl_option CYGNUM_DEVS_JOYSTICK_SPCE3200_EVENT_BUFFER_SIZE {
            display "Number of events the JoyStick driver buffer"
            flavor data
            legal_values 1 to 512
            default_value 32
            description "
                This option defines the size of the JoyStick device internal
            buffer. The cyg_io_read() function will return as many of these
```

```
                as there is space for in the buffer passed."
            }
        }
    }
```

active_if CYGPKG_DEVS_IOB_SPCE3200 表明 JoyStick 设备与 IOB 设备的依赖关系，即 JoyStick 设备依赖于 IOB 设备，必须在加载 IOB 设备组件包的前提下，加载 JoyStick 设备组件包才有效，如果在编译平台中只加载 JoyStick 设备组件包而不加载 IOB 设备组件包是没有意义的。

5.2 设计头文件

在头文件中定义 key 值，并定义 JoyStick 键值供用户使用，并定义接口相关的一些常量，以便驱动程序使用。

头文件参考程序如下：

```
//==========================================================================
//    定义键值
//==========================================================================
#define JOYS_UP                 0x0008      // 上（Up 键）
#define JOYS_DOWN               0x0004      // 下（Down 键）
#define JOYS_LEFT               0x0002      // 左（Left 键）
#define JOYS_RIGHT              0x0001      // 右（Right 键）
#define JOYS_SELECT             0x0020      // Select 键
#define JOYS_START              0x0010      // Start 键
#define JOYS_A                  0x0040      // A 键
#define JOYS_B                  0x0080      // B 键
//==========================================================================
//    定义 key 值
//==========================================================================
#define SEL_READJOY             0x1801
//==========================================================================
//    定义 JoyStick 选择常量
//==========================================================================
#define READ_JOY1               1           // 读 JoyStick1 常量
#define READ_JOY2               2           // 读 JoyStick2 常量
//==========================================================================
//    定义 IOB 口寄存器中各偏移量
//==========================================================================
#define JOYS_OE_OFFSET          8           // GPIO_OUTPUTEN 最低有效位偏移量
```

```
#define JOYS_OUTPUT_OFFSET      0       // GPIO_OUTPUT 最低有效位偏移量
#define JOYS_PULLUP_OFFSET      16      // GPIO_PULLUP 最低有效位偏移量
#define JOYS_PULLDOWN_OFFSET    24      // GPIO_PULLDOWN 最低有效位偏移量
#define JOYS_INPUT_OFFSET       8       // GPIO_INPUT 最低有效位偏移量
//=============================================================================
//    定义 JoyStick 占用的 IOB 口，0 表示 IOB0，1 表示 IOB1
//=============================================================================
#define JOYS_CLK_BIT            0       // CLK 位
#define JOYS_LOAD_BIT           1       // LOAD 位
#define JOYS_DAT0_BIT           2       // Data0 位
#define JOYS_DAT1_BIT           3       // Data1 位
```

5.3 设计源程序文件

JoyStick 的源程序文件编写方法与字符设备及块设备类似，同样包含设备表入口和接口函数的编写。所不同的是，在编写设备表入口的时候指定底层设备，在适当的时候读取底层设备的设备句柄，并通过句柄读写底层设备以完成对 JoyStick 设备的各种操作，下面一一介绍：

1、 设备表入口

JoyStick 是字符设备，所以同样使用 CHAR_DEVTAB_ENTRY 宏将 JoyStick 设备注册到 eCos 系统中：

```
CHAR_DEVTAB_ENTRY(spce3200_joystick_device,     // JoyStick 设备表入口句柄
    CYGDAT_DEVS_JOYSTICK_SPCE3200_NAME,         // 设备名，在 cdl 文件中进行宏定义
    "/dev/iob",                                 // 底层设备名
    &spce3200_joystick_handlers,                // JoyStick 设备 I/O 函数表句柄指针
    joystick_init,                              // JoyStick 初始化函数
    joystick_lookup,                            // JoyStick 设备查找函数
    NULL
);
```

在该宏中指定了 JoyStick 的底层设备为/dev/iob，这样在用户调用 cyg_io_lookup()函数查找/dev/JoyStick 设备时候，先会查找/dev/iob 设备，并通过 JoyStick 驱动的接口 lookup 函数返回/dev/iob 设备的句柄，详见 JoyStick 驱动的接口 lookup 函数。

I/O 函数表同样通过 CHAR_DEVIO_TABLE 宏来定义：

```
CHAR_DEVIO_TABLE(spce3200_joystick_handlers, // JoyStick 设备 I/O 函数表句柄
    NULL,                                    // JoyStick 设备写函数
    joystick_read,                           // JoyStick 设备读函数
    NULL,                                    //
    joystick_get_config,                     // JoyStick 读设备设置状态函数
```

```
    joystick_set_config                         // JoyStick 设备设置函数
);
```

2、 JoyStick 设备驱动接口函数

（1） 设备查找函数 joystick_lookup

在介绍设备表入口的时候介绍，当系统调用 cyg_io_lookup()函数查找 JoyStick 设备的时候，先会查找/dev/iob 设备，并且通过 JoyStick 驱动的接口 lookup 函数返回/dev/iob 设备的句柄，所以在本函数中可以读到底层设备/dev/iob 设备的句柄。

JoyStick 的设备查找函数 joystick_lookup 参考程序如下：

```
static Cyg_ErrNo joystick_lookup(struct cyg_devtab_entry **tab, struct cyg_devtab_entry *st, const char *name)
{
#if CYGNUM_DEVS_JOYSTICK_SPCE3200_DEBUG_MODE == 1
    printf("Debug - JoyStick Lookup\n");            // 打印调试信息
#endif
    cyg_uint32 len = 1;
    cyg_uint32 Setup[1];
    cyg_uint32 WriteData[1];

    hDrvIOB = (cyg_io_handle_t)st;
    // DAT0,DAT1 设置为上拉
    Setup[0] &= ~ ((0x01<<(JOYS_DAT0_BIT+JOYS_PULLUP_OFFSET))
                  +(0x01<<(JOYS_DAT1_BIT+JOYS_PULLUP_OFFSET)));
    // DAT0,DAT1 设置为不下拉
    Setup[0] &= ~((0x01<<(JOYS_DAT0_BIT+JOYS_PULLDOWN_OFFSET))
                  +(0x01<<(JOYS_DAT1_BIT+JOYS_PULLDOWN_OFFSET)));
    // DAT0,DAT1 设置为输入
    Setup[0] &= ~((0x01<<(JOYS_DAT0_BIT+JOYS_OE_OFFSET))
                  +(0x01<<(JOYS_DAT1_BIT+JOYS_OE_OFFSET)));
    // CLK,LOAD 设置为输出
    Setup[0] |= (0x01<<(JOYS_CLK_BIT+JOYS_OE_OFFSET))
                +(0x01<<(JOYS_LOAD_BIT+JOYS_OE_OFFSET));
    if(ENOERR != cyg_io_set_config(hDrvIOB,CYGNUM_DEVS_IOB_SPCE3200_ATTRIB, Setup,&len))
    {
        CYG_TEST_FAIL_FINISH("Error set config DAT0,DAT1,CLK,LOAD");
    }
```

```
        // JoyStick LOAD 清零
        WriteData[0] &= ~(0x00000001<<(JOYS_LOAD_BIT+JOYS_OUTPUT_OFFSET));
        // JoyStick CLK 置位
        WriteData[0] |= (0x00000001<<(JOYS_CLK_BIT+JOYS_OUTPUT_OFFSET));
        if(ENOERR != cyg_io_write(hDrvIOB, WriteData, &len))
        {
            CYG_TEST_FINISH("Error Clear LOAD or Set CLK");
        }
        if (!_JoyStickDrv_is_open)
        {
            _JoyStickDrv_is_open = true;
            cyg_thread_create(1,                        // 优先级
                              JoyStick_Scan,            // 线程入口函数
                              0,                        // 入口参数
                              "JoyStick scan",          // 线程名
                              &JoyStick_stack[0],       // 堆栈
                              STACK_SIZE,               // 堆栈大小
                              &JoyStick_thread_handle,  // 线程句柄
                              &JoyStick_thread_data     // 线程数据
                             );
            cyg_thread_resume(JoyStick_thread_handle);  // 线程启动
        }
        return ENOERR;
    }
```

调用 cyg_io_lookup()函数时，通过 JoyStick 驱动的接口 lookup 函数的第二个参数 st 返回 IOB 设备的句柄。另外，对 JoyStick 设备的各个接口的操作，都是通过 I/O 子系统接口函数调用 IOB 驱动接口函数实现，也就是说，对 JoyStick 设备的各个接口的操作，事实上就是在操作 IOB 设备，从 IOB 设备的角度来说，JoyStick 设备就是一个用户，JoyStick 设备的驱动程序就是 IOB 设备驱动程序的用户程序。例如以上程序段中的 cyg_io_write(hDrvIOB, WriteData, &len)。

JoyStick 驱动接口 lookup 函数实现的功能是初始化 JoyStick 的接口、设置 JoyStick 设备硬件的初始状态，并且建立一个线程扫描 JoyStick 设备。不同于一般的字符设备，初始化接口和设置 JoyStick 设备硬件的初始状态并没有在初始化函数中进行，这是因为初始化函数在 eCos 系统启动的时候就已经被调用，在并没有查找 JoyStick 设备之前，已经改变了 JoyStick 的接口，也就是 IOB 口的状态，这样会影响到其他设备对 IOB 口的操作，或者由于其他设备占先了 IOB 口，导致 JoyStick 设备并没有达到初始化和设置的效果。

（2） JoyStick_Scan 函数

在设备查找函数 joystick_lookup 中建立了一个线程，该线程的入口函数命名为 JoyStick_Scan，

这个函数实现的功能是扫描 JoyStick 设备，参考程序如下：

```
void JoyStick_Scan(cyg_addrword_t data)
{
    short i;
    unsigned short TempCode0, TempCode1;
    cyg_uint32          len = 1;
    cyg_uint32          ReadData[1];
    cyg_uint32          WriteData[1];

    while(true)
    {
        TempCode0 = 0x0000;
        TempCode1 = 0x0000;
        WriteData[0] |= (0x00000001<<(JOYS_LOAD_BIT+JOYS_OUTPUT_OFFSET));
        if(ENOERR != cyg_io_write(hDrvIOB, WriteData, &len))
        {                                       // Load 置高电平
            CYG_TEST_FINISH("Error Set LOAD");
        }

        cyg_thread_delay(2);                    // 线程睡眠延时
        WriteData[0] &= ~(0x00000001<<(JOYS_LOAD_BIT+JOYS_OUTPUT_OFFSET));
        if(ENOERR != cyg_io_write(hDrvIOB, WriteData, &len))
        {                                       // Load 置低电平
            CYG_TEST_FINISH("Error Clear LOAD");
        }

        for(i=0; i<8; i++)                      // 读 8 位 JoyStick 键值数据
        {
            WriteData[0] &= ~(0x00000001<<(JOYS_CLK_BIT+JOYS_OUTPUT_OFFSET));
            if(ENOERR != cyg_io_write(hDrvIOB, WriteData, &len))
            {                                   // 时钟置低电平，即提供下跳沿
                CYG_TEST_FINISH("Error Clear CLK");
            }
            TempCode0 <<= 1;                    // 左移一位以便读下一位
            TempCode1 <<= 1;
            if(ENOERR != cyg_io_read(hDrvIOB, ReadData, &len))
```

```
        {                                   // 读一位数据
            CYG_TEST_FINISH("Error Read DAT0,DAT1");
        }

        TempCode0 |=                        // 读到 JoyStick1 的数据，保存到 TempCode0
            ((ReadData[0]>>(JOYS_DAT0_BIT+JOYS_INPUT_OFFSET))&0x0001);
        TempCode1 |=                        // 读到 JoyStick2 的数据，保存到 TempCode1
            ((ReadData[0]>>(JOYS_DAT1_BIT+JOYS_INPUT_OFFSET))&0x0001);
        cyg_thread_delay(2);                // 线程睡眠延时
        WriteData[0] |= (0x00000001<<(JOYS_CLK_BIT+JOYS_OUTPUT_OFFSET));
        if(ENOERR != cyg_io_write(hDrvIOB, WriteData, &len))
        {                                   // 时钟置高电平
            CYG_TEST_FINISH("Error Clear LOAD or Set CLK");
        }
    }
    if(TempCode0!=0xff)                     // 如果 JoyStick1 有键按下
    {
        if(_JData_Num1<MAX_EVENTS)      // 如果 JoyStick1 的键值 Buffer 非满
        {
            TempCode0 = (TempCode0^0xff);
            JoyStick1_Buffer[_Joy_put1] = TempCode0;// 键值存入 Buffer
            _Joy_put1++;
            _JData_Num1++;
            if(_Joy_put1==MAX_EVENTS)
                _Joy_put1 = 0;
            cyg_semaphore_post(&sem_JoyStick_data);// Buffer 中有数据， post 信号量
        }
    }
    if(TempCode1!=0xff)                     // 如果 JoyStick1 有键按下
    {
        if(_JData_Num2<MAX_EVENTS)      // 如果 JoyStick2 的键值 Buffer 非满
        {
            TempCode1 = (TempCode1^0xff); // 键值存入 Buffer
            JoyStick2_Buffer[_Joy_put1] = TempCode1;
            _Joy_put2++;
            _JData_Num2++;
```

```
                    if(_Joy_put2==MAX_EVENTS)
                        _Joy_put2 = 0;
                    cyg_semaphore_post(&sem_JoyStick_data); // Buffer 中有数据，post 信号量
                }
            }
        }
    }
```

如上参考程序，在 JoyStick_Scan 函数中，按照 JoyStick 的工作时序读 JoyStick 发出的每一位数据，并处理成键值写到 JoyStick 的数据 Buffer 中。

（3） 设备初始化函数 joystick_init

设备初始化函数主要进行 JoyStick Buffer 指针及数据数量的初始化、信号量的初始化操作。参考程序如下：

```
static bool joystick_init(struct cyg_devtab_entry *tab)
{
#if CYGNUM_DEVS_JOYSTICK_SPCE3200_DEBUG_MODE == 1
    diag_printf("Debug - JoyStick Init\n");
#endif
    _JData_Num1=_JData_Num2=_Joy_put1= _Joy_get1=_Joy_put2=_Joy_get2=0;
    Read_Flag = 0;
    cyg_semaphore_init(&sem_JoyStick_data,0);
        // 初始化信号量，该信号量控制系统共享 Buffer 中的数据
    return true;
}
```

（4） 设备读函数 joystick_read

设备读函数 joystick_read 实现的功能是从 JoyStick 的数据 Buffer 读出键值数据。参考程序如下：

```
static Cyg_ErrNo joystick_read(cyg_io_handle_t handle, void *buffer, cyg_uint32 *len)
{
    unsigned short *bp = (unsigned short *)buffer;
    int tot = *len;

#if CYGNUM_DEVS_JOYSTICK_SPCE3200_DEBUG_MODE == 1
    printf("Debug - JoyStick read\n");
#endif
```

```
while(tot)                                  // 是否读到参数 len 个数据
{
    if(Read_Flag==READ_JOY1)                // 如果正在读 JoyStick1
    {
        if(_JData_Num1>0)                   // 键值数据 Buffer 中是否有数据
        {
            *bp = JoyStick1_Buffer[_Joy_get1];   // 是，读一个键值数据
            _Joy_get1++;
            _JData_Num1--;
            if(_Joy_get1==MAX_EVENTS)
                _Joy_get1 = 0;
            bp++;
            tot--;
        }
        else                                // 否，信号量 wait
        {
            cyg_semaphore_wait(&sem_JoyStick_data);
        }
    }
    if(Read_Flag==READ_JOY2)                // 如果正在读 JoyStick1
    {
        if(_JData_Num2>0)                   // 键值数据 Buffer 中是否有数据
        {

            *bp = JoyStick2_Buffer[_Joy_get1];   // 是，读一个键值数据
            _Joy_get2++;
            _JData_Num2--;
            if(_Joy_get2==MAX_EVENTS)
                _Joy_get2 = 0;
            bp++;
            tot--;
        }
        else                                // 否，信号量 wait
        {
            cyg_semaphore_wait(&sem_JoyStick_data);
        }
```

```
            }
        }
        return ENOERR;
    }
```

在设备读函数 joystick_read 中，先要判断两个 JoyStick 各自的键值数据 Buffer 中是否有数据，如果没有，信号量 wait，直到 Buffer 中有数据，信号量 post。

（5） 设备设置函数 joystick_set_config

由于 SPCE3200 实验箱上由两个 JoyStick 接口，所以在读设备时，必须要指定是哪个 JoyStick 设备接口，这些功能在设备设置函数 joystick_set_config 中完成，这样用户可以通过 cyg_io_set_config() 函数来指定使用的 JoyStick 设备接口。

```
    static Cyg_ErrNo joystick_set_config(cyg_io_handle_t handle, cyg_uint32 key, const void *buffer, cyg_uint32 *len)
    {
        unsigned short *bp = (unsigned short *)buffer;
    #if CYGNUM_DEVS_JOYSTICK_SPCE3200_DEBUG_MODE == 1
        printf("Debug - JoyStick Set Config\n");
    #endif
        switch(key)
        {
            case SEL_READJOY:
            {
                if(*bp==READ_JOY1)
                    Read_Flag = READ_JOY1;
                if(*bp==READ_JOY2)
                    Read_Flag = READ_JOY1;
                break;
            }
            default:
                break;
        }
        return EINVAL;
    }
```

（6） 读设备设置状态函数 joystick_get_config

读设备设置状态函数 joystick_get_config 中可以读到正在读取的 JoyStick 接口，或者 JoyStick 的其他状态。参考程序如下：

```
static Cyg_ErrNo joystick_get_config(cyg_io_handle_t handle, cyg_uint32 key, void *buffer, cyg_uint32 *len)
{
#if CYGNUM_DEVS_JOYSTICK_SPCE3200_DEBUG_MODE == 1
    printf("Debug - JoyStick Get Config\n");
#endif
    return EINVAL;
}
```

在参考程序中，没有进行任何操作，用户可以根据自己的需求定义相关操作。

以下为 JoyStick 设备源程序文件中驱动程序参考伪代码：

```
//----------------------------------------包含相关头文件----------------------------------------//
#include <pkgconf/hal.h>                    // hal 的宏头文件，由系统自动生成
#include <pkgconf/devs_joystick_spce3200.h>// JoyStick 驱动 cdl 文件宏头文件，由系统自动生成
#include <cyg/infra/cyg_type.h>             // 定义类型
#include <cyg/error/codes.h>                // 定义错误常量，如 ENOERR
#if CYGNUM_DEVS_JOYSTICK_SPCE3200_DEBUG_MODE == 1
    #include <cyg/infra/diag.h>             // diag 函数声明
    #include <stdio.h>                      // 调试函数如 printf 等声明
#endif
#include <cyg/io/devtab.h>                  // 设备 I/O 入口表定义及声明
#include <cyg/io/io.h>                      // 声明 I/O 子系统 API 函数
#include <cyg/io/spce3200_joystick.h>       // 定义 JoyStick 模块相关常量，这里含 key
#include <cyg/io/spce3200_iob.h>            // 定义 SPCE3200 的 IOB 模块相关常量，这里含 key
//-----------------------------------定义 Buffer 及 Buffer 指针----------------------------------//
#define MAX_EVENTS CYGNUM_DEVS_JOYSTICK_SPCE3200_EVENT_BUFFER_SIZE
static int     _JData_Num1,_JData_Num2;
static int     _Joy_put1, _Joy_get1,_Joy_put2,_Joy_get2;
static short JoyStick1_Buffer[MAX_EVENTS];
static short JoyStick2_Buffer[MAX_EVENTS];
static int    Read_Flag;
//-------------------------------定义线程堆栈、句柄、信号量等-------------------------------//
static cyg_io_handle_t hDrvIOB;
static bool _JoyStickDrv_is_open = false;
#define STACK_SIZE CYGNUM_HAL_STACK_SIZE_TYPICAL
static char JoyStick_stack[STACK_SIZE];
static cyg_thread JoyStick_thread_data;
```

```
static cyg_handle_t JoyStick_thread_handle;
static cyg_sem_t sem_JoyStick_data;
//---------------------------------------线程函数及接口函数-----------------------------------//
void JoyStick_Scan(cyg_addrword_t data)
{…}
static Cyg_ErrNo joystick_read(cyg_io_handle_t handle, void *buffer, cyg_uint32 *len)
{…}
static Cyg_ErrNo joystick_set_config(cyg_io_handle_t handle, cyg_uint32 key, const void *buffer, cyg_uint32 *len)
{…}
static Cyg_ErrNo joystick_get_config(cyg_io_handle_t handle, cyg_uint32 key, void *buffer, cyg_uint32 *len)
{…}
static bool joystick_init(struct cyg_devtab_entry *tab)
{…}
static Cyg_ErrNo joystick_lookup(struct cyg_devtab_entry **tab, struct cyg_devtab_entry *st, const char *name)
{…}
//-----------------------------------------设备表入口-----------------------------------------//
CHAR_DEVIO_TABLE(…);
CHAR_DEVTAB_ENTRY(…);
```

5.4 向eCos数据库中添加JoyStick驱动程序组件包

在 ecos.db 中的任意位置加如下程序段：

```
package CYGPKG_DEVS_JOYSTICK_SPCE3200 {
    alias          { "JoyStick support for Score SPCE3200"}
    directory      devs/joystick/score/spce3200
    script         joystick_spce3200.cdl
        description "
            This package contains hardware support for the JoyStick
        on the Score SPCE3200 DEV Board/Box."
}
```

以上程序段把 JoyStick 驱动程序组件包添加到 eCos 数据库中。

5.1节介绍cdl文件时提到，JoyStick设备基于IOB设备，在加载JoyStick驱动程序组件包编译时必须加载IOB驱动程序组件包，否则JoyStick驱动程序组件包是不能参与编译的。如图 5.1为加载了JoyStick驱动程序组件包但是没有加载IOB驱动程序组件包，此时“JoyStick driver for SPCE3200”显示为灰色，表明此组件包不可用。

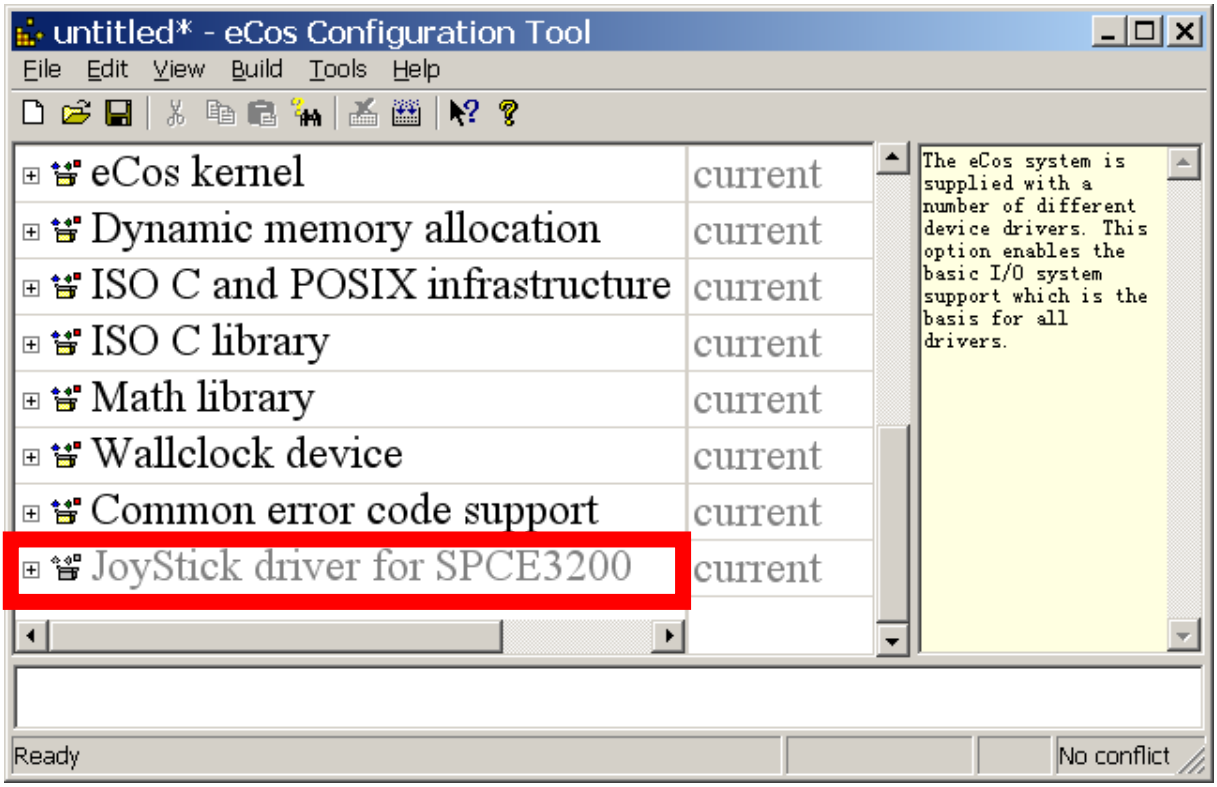

图 5.1 没有加载 IOB 驱动程序组件包

图 5.2加载了JoyStick驱动程序组件包和IOB驱动程序组件包，此时“JoyStick driver for SPCE3200”由图 5.1的灰色变为黑色，JoyStick驱动程序组件包可用。

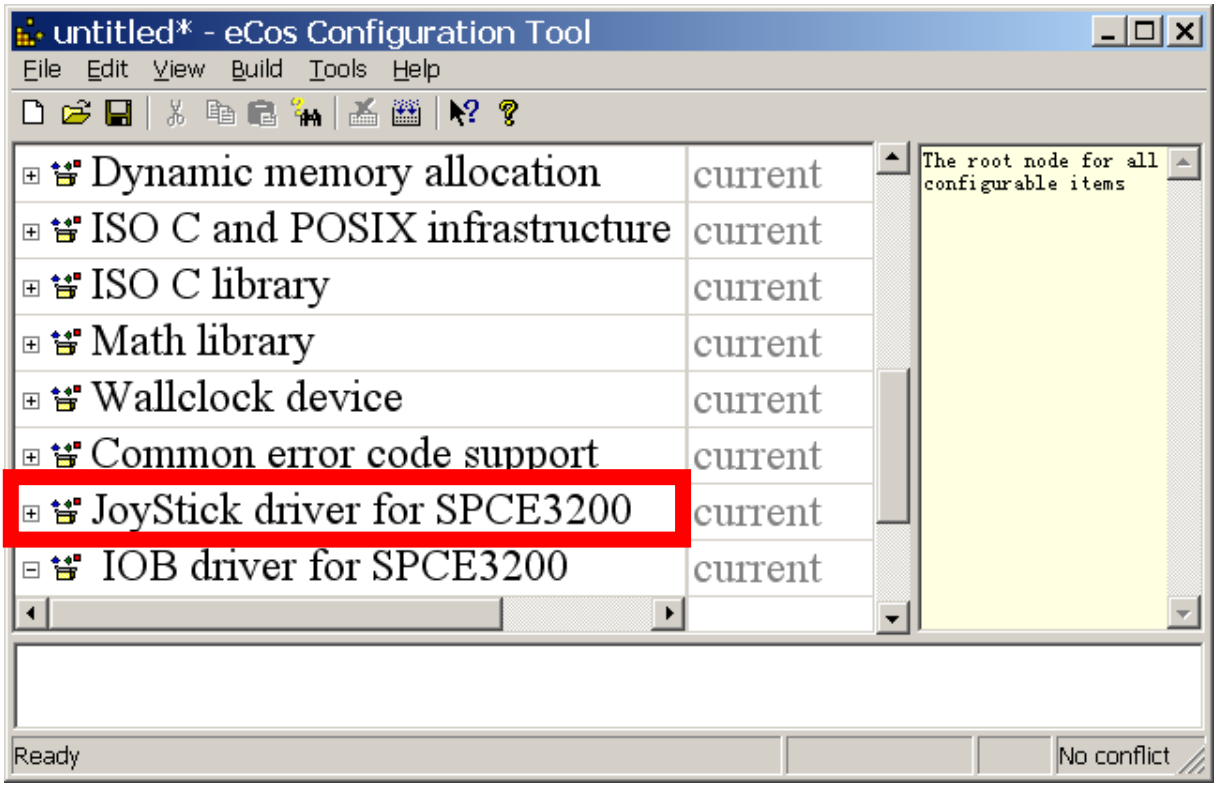

图 5.2 加载 IOB 驱动程序组件包

按照图 5.2添加包并选择好合适的选项后，编译生成*.ecc库文件即可使用JoyStick驱动程序。

5.5 用户测试程序

在按照图 5.2编译没有错误后，就可以编写测试程序验证驱动程序是否工作正常。

在实验箱的 VB2 JoyStick 接口连接一个游戏手柄。测试参考程序如下：

```
//-----------------------------------------包含相关头文件-----------------------------------------//
#include <cyg/infra/testcase.h>          // 测试宏定义，如 CYG_TEST_FAIL_FINISH
#include <cyg/io/io.h>                   // 声明 I/O 子系统 API 函数
#include <cyg/kernel/kapi.h>             // 定义针对内核的 API 函数，同步机制的 API 函数等
#include <stdio.h>                       // 声明 printf 等调试函数
#include <cyg/io/spce3200_joystick.h>    // 定义 JoyStick 模块相关常量，这里含 key

cyg_io_handle_t    hDrvJoyStick;
//----------------------获取 JoyStick 按键键值的线程 JoyStick_thread ----------------------------//
void JoyStick_thread(cyg_addrword_t data)
{
    cyg_uint32              len = 1;
    unsigned short          Joy_Value[1];
    int                     Joy_Sel[1];
    Joy_Value[0] = 0;

    if (ENOERR != cyg_io_lookup("/dev/JoyStick", &hDrvJoyStick))    // 查找 JoyStick 设备
    {
        CYG_TEST_FAIL_FINISH("Error opening /dev/JoyStick");
    }
    printf("Open /dev/JoyStick OK\n");

    Joy_Sel[0] = READ_JOY1;                                         // 读 JoyStick1
    if(EINVAL != cyg_io_set_config(hDrvJoyStick,SEL_READJOY,Joy_Sel,&len)) //
    {
        CYG_TEST_FAIL_FINISH("Error set config /dev/JoyStick");
    }

    while(true)
    {
        len = 1;
```

```
        if(ENOERR != cyg_io_read(hDrvJoyStick, Joy_Value, &len))   // 读键值
        {
            CYG_TEST_FINISH("Error Read /dev/JoyStick");
        }
        switch(Joy_Value[0])
        {
            case JOYS_UP:
                printf("JOYS_UP\n");              // 如果按下 Up 键，打印 JOYS_UP
                break;
            case JOYS_DOWN:
                printf("JOYS_DOWN\n");            // 如果按下 Down 键，打印 JOYS_DOWN
                break;
            case JOYS_LEFT:
                printf("JOYS_LEFT\n");            // 如果按下 Left 键，打印 JOYS_LEFT
                break;
            case JOYS_RIGHT:
                printf("JOYS_RIGHT\n");           // 如果按下 Right 键，打印 JOYS_RIGHT
                break;
            case JOYS_SELECT:
                printf("JOYS_SELECT\n");          // 如果按下 Select 键，打印 JOYS_SELECT
                break;
            case JOYS_START:
                printf("JOYS_START\n");           // 如果按下 Start 键，打印 JOYS_START
                break;
            case JOYS_A:
                printf("JOYS_A\n");               // 如果按下 A 键，打印 JOYS_A
                break;
            case JOYS_B:
                printf("JOYS_B\n");               // 如果按下 B 键，打印 JOYS_B
                break;
            default:
                printf("JOYS_NoKey\n");           // 其他，打印 JOYS_NoKey
                break;
        }
        cyg_thread_delay(10);                     // 线程睡眠
    }
```

```
}

cyg_uint8 stack[4096];
cyg_handle_t simple_threadA;
cyg_thread thread_s;

void cyg_user_start(void)
{
    CYG_TEST_INIT();
    cyg_thread_create(4, JoyStick_thread,
                      (cyg_addrword_t) 0,
                      "iob_thread",
                      (void *) stack, 4096,
                      &simple_threadA, &thread_s);     // 建立一个线程 JoyStick_thread
    cyg_thread_resume(simple_threadA);
    cyg_scheduler_start();
}
```

6 SD卡驱动程序设计

SPCE3200 内部具有 SD 卡控制器，本节将介绍如何为 SPCE3200 内部的 SD 卡控制器编写驱动程序，以便完成 SD 卡的读写操作。这里假设 SD 卡使用“/dev/sd”做为其设备名称。

SD 卡设备属于块设备，为了提高效率，通过 SPCE3200 的 DMA 对 SD 卡进行读写，所以 SD 卡也是基于底层设备的设备，这里的底层设备指的是 DMA 设备。

6.1 建立SD卡驱动程序目录结构

在编写驱动程序前，首先需要为SD卡驱动建立其目录结构。根据前面章节的介绍，所有驱动程序均保存在/packages/devs目录下的对应的子目录内，首先按照图 6.1所示在/devs/packages/sd目录下建立各级目录。其中， “cdl”、“include”、“src”三个目录分别用于保存SPCE3200 下的SD卡驱动的cdl文件、驱动程序头文件、源程序文件。

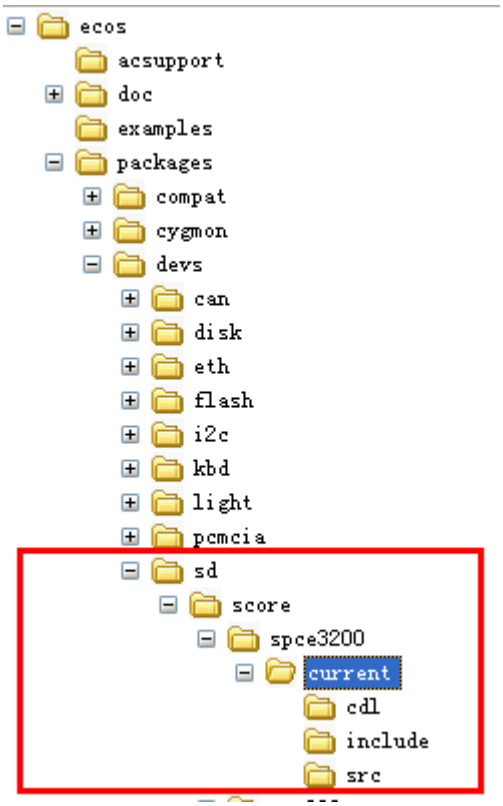

图 6.1 SD 卡驱动的目录结构

6.2 建立SD卡驱动程序文件

在 cdl 目录内，需要用户创建设备的 CDL 脚本文件，文件扩展名为.cdl。该文件的文件名任意，但一般推荐以“设备名_平台名.cdl”的形式进行命名，以便可以较好的说明该文件的作用，比如这里可以叫做“sd_spce3200.cdl”。

然后，在 src 目录内需要建立驱动程序的源程序文件。如使用 C 语言编写驱动程序，则可以建立*.c 的文件，如使用 C++编写驱动程序，则可以建立*.cxx 的文件。源程序文件的文件名也是任意

的，但一般推荐以“设备名_平台名.c”的形式进行命名，比如这里可以叫做“sd_card_spce3200.c”。另外，源程序文件可以有多个，用户可以合理安排源程序文件的数量和作用，以便让代码看起来更清晰。

建立好文件之后的目录结构如图 6.2所示。

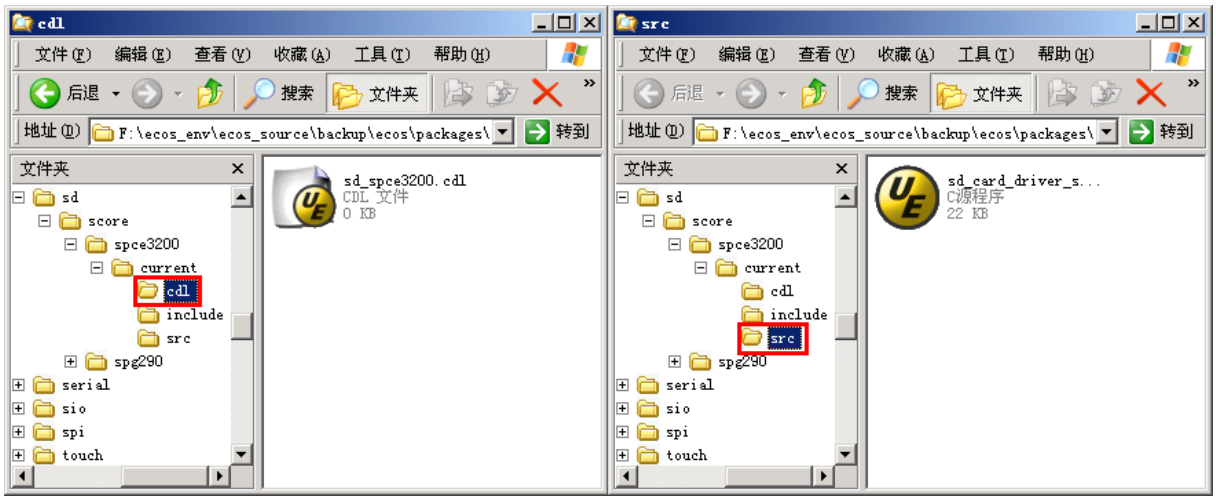

图 6.2 SD 卡驱动的文件结构

6.3 设计CDL脚本文件

与其他设备驱动程序类似，在编写 SD 卡驱动程序代码之前，首先需要设计一些配置选项，这些配置选项将以宏的形式供源程序使用，对设备进行编译前的配置。

SD 卡设备参考 CDL 文件程序如下：

```
cdl_package CYGPKG_DEVS_SD_SPCE3200 {
    display       "SD driver for SPCE3200"
    include_dir   cyg/io
    active_if     CYGPKG_IO_DISK
    active_if     CYGPKG_DEVS_DMA_SPCE3200
    compile         -library=libextras.a sd_card_spce3200.c
    description "SD driver for SCORE SPCE3200"

    cdl_component CYGPKG_DEVS_SD_SPCE3200_OPTIONS {
        display "Compile Options"
        flavor  none
        no_define

        cdl_option CYGPKG_DEVS_SD_SPCE3200_CFLAGS {
            display         "Additional compiler flags"
            flavor          data
```

```
            no_define
            default_value { "" }
            description "
                This option modifies the set of compiler flags for
                building the keypad driver package. These flags
                are used in addition to the set of global flags."
        }

        cdl_option CYGDAT_DEVS_SD_SPCE3200_NAME {
            display "Device name for the keyboard driver"
            flavor data
            default_value {"\"/dev/sd\""}
            description " This option specifies the name of the sd-card device"
        }

        cdl_option CYGNUM_DEVS_SD_SPCE3200_DEBUG_MODE {
            display "Debug Message"
            default_value    0
            description "
                This option will enable the debug message outputing if set to 1,
                will disable the outputing if set to 0."
        }

        cdl_option CYGNUM_DEVS_SD_SPCE3200_WAIT_INSERT {
            display "Timeout of Waiting SD Card Insert"
            flavor data
            default_value    0xffff
            description "
                This option defines the time waiting for SD Card Insert."
        }
    }
}
```

如上参考程序，首先建立 SD 卡组件包 CYGPKG_DEVS_SD_SPCE3200，利用这个组件包名可以把 SD 卡设备注册到 eCos 系统中。在 CYGPKG_DEVS_SD_SPCE3200 组件包中定义各组件和配置选项，其中，CYGDAT_DEVS_SD_SPCE3200_NAME 宏代表了用户在配置工具中设定的设备名称，在编写驱动程序时使用该宏做为设备名称的标识；CYGNUM_DEVS_SD_SPCE3200_DEBUG_MODE 宏允许用户打开调试信息输出；CYGNUM_DEVS_SD_SPCE3200_WAIT_INSERT 宏允许用户设定卡

插入检测超时时间；“active_if　CYGPKG_IO_DISK”指定 SD 卡设备与 disk 的依赖关系，在使用文件系统的管理 SD 卡时候，可以把 SD 卡作为 disk 设备进行管理，这项可不要；“active_if CYGPKG_DEVS_DMA_SPCE3200”指定 SD 卡设备与 DMA 设备的依赖关系，DMA 设备作为 SD 卡设备的底层设备。

6.4　编写SD卡设备表入口

CDL 文件描述了当前设备的基本特性，接下来，需要在 src/sd_spce3200.c 文件中编写设备表入口。为了方便理解，首先确定以下几个函数的函数名：

- 设备初始化函数：SD_Initial()
- 设备查找函数：SD_Lookup()
- 设备批量写操作函数：SD_WriteSector()
- 设备批量读操作函数：SD_ReadSector()
- 设备检测函数：不需要
- 读设备配置状态函数：SD_GetConfig()
- 设备配置函数：SD_SetConfig()

注：这些函数名用户可以自定义。

以上这些函数作为 SD 卡设备驱动的接口函数。确定了以上这些函数名之后，便可以编写设备 I/O 函数表宏以及设备表入口宏：

```
BLOCK_DEVIO_TABLE(sd_spce3200_handlers,     //SD 卡设备 I/O 函数表句柄，名称可自定义
        SD_WriteSector,                     // SD 卡写函数
        SD_ReadSector,                      // SD 卡读函数
        NULL,                               //  检测函数，这里不需要
        SD_GetConfig,                       //  读 SD 卡配置状态函数
        SD_SetConfig                        // SD 卡配置函数
);
BLOCK_DEVTAB_ENTRY(sd_spce3200_device,      // SD 卡设备表入口句柄
        CYGDAT_DEVS_SD_SPCE3200_NAME,       // SD 卡设备的名称，在 CDL 文件里定义
        "/dev/dma",                         // SD 卡驱动基于 DMA 设备
        &sd_spce3200_handlers,              // SD 卡设备 I/O 函数表句柄指针
        &SD_Initial,                        // SD 卡初始化函数
        &SD_Lookup,                         // SD 卡查找函数
        0                                   // SD 卡驱动的私有指针，这里不需要
);
```

6.5　实现SD卡设备接口函数

在设备 I/O 函数表宏以及设备表入口宏中调用了 SD 卡的接口函数，这些接口函数是上层访问硬

件设备的唯一通道。

SD 卡设备有 6 个设备驱动接口函数：SD_Initial、SD_Lookup、SD_WriteSector、SD_ReadSector、SD_GetConfig、SD_SetConfig，下面一一介绍：

1、 SD 卡设备初始化函数 SD_Initial()

在 SD 卡设备初始化函数中主要完成对控制器的使能以及信号量的初始化。参考代码如下：

```
static cyg_bool SD_Initial(struct cyg_devtab_entry* tab)
{
#ifdef CYGNUM_DEVS_SD_SPCE3200_DEBUG_MODE
        diag_printf("Debug - SD_Initial()\n");
#endif
    *P_NAND_GPIO_PULL = 0xfc19f819;                  // DAT3 设置为悬浮，其他为上拉
    *P_SD_INTERFACE_SEL |= C_SD_PORT_SEL;            // SD 卡接口使能
    *P_SD_CLK_CONF = C_SD_CLK_EN | C_SD_RST_DIS;// SD 卡时钟使能、不复位
    cyg_semaphore_init(&sem_sd_ready,0);             // 初始化信号量
    cyg_semaphore_post(&sem_sd_ready);// post 信号量以便对 SD 卡进行第一次操作
    return true;
}
```

2、 SD 卡设备查找函数 SD_Lookup()

SD 卡设备查找函数主要完成对 SD 卡的初始化，包括卡插入检测、发送初始化命令序列、等待 SD 卡进入 StandBy 模式等。参考代码如下：

```
static Cyg_ErrNo SD_Lookup(struct cyg_devtab_entry **tab, struct cyg_devtab_entry *st, const char *name)
{
    int i, loopcnt=0;
    unsigned int response[4];
    unsigned int tranunit, timevalue;
    unsigned int maxspeed;
    unsigned int c_size, mult, blocklen;
    unsigned int SpeedTable[]={0, 10, 12, 13, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 70, 80,};
    unsigned int TranTable[]={1, 10, 100, 1000,};
#ifdef CYGNUM_DEVS_SD_SPCE3200_DEBUG_MODE
        diag_printf("Debug - SD_Lookup()\n");
#endif
    if(__IsInit__)
        return ENOERR;
```

```
        __IsInit__ = true;
        hDrvDMA = (cyg_io_handle_t)st;                    // 读 DMA 设备驱动的句柄
    #if CYGNUM_DEVS_SD_SPCE3200_WAIT_INSERT > 0// 如果等待卡插入超时时间大于 0
        i = 0;
        #ifdef CYGNUM_DEVS_SD_SPCE3200_DEBUG_MODE
            diag_printf("Debug - SD Wait Card Insert\n");
        #endif
        while((!__SDDrv_SDDrv_CheckCard__()) && (i < CYGNUM_DEVS_SD_SPCE3200_WAIT_
INSERT))                                                  // 等待卡插入
            i++;
        if(i >= CYGNUM_DEVS_SD_SPCE3200_WAIT_INSERT)
    #else
        if(!__SDDrv_SDDrv_CheckCard__())                  // 如果卡没有插入
    #endif
        {
        #ifdef CYGNUM_DEVS_SD_SPCE3200_DEBUG_MODE
            diag_printf("Debug - SD Card Not Present\n");  // 打印卡未插入信息
        #endif
            return -EAGAIN;
        }
        #ifdef CYGNUM_DEVS_SD_SPCE3200_DEBUG_MODE
            diag_printf("Debug - SD Card Present, Start Initial...\n");// 卡插入，开始初始化
        #endif
        *P_SD_INT_CTRL = 0x0000;                          // 禁能所有 SD 卡中断
        // 第一步，复位 SD 卡控制器
        *P_SD_MODE_CTRL = C_BLOCKLEN_512BYTE  // 扇区长度为 512Byte
                        | C_INIT_CLOCK_SPEED  // 时钟频率设置
                        | C_SD_PORT_EN        // SD 总线使能
                        | C_SD_DMA_EN;        // 使用 DMA 传输模式
        // 清除所有命令
        *P_SD_COMMAND_SETUP = C_SD_CTRL_IDLE; // SD 卡控制器回到 Idle 状态
        while (*P_SD_INT_STATUS & C_SD_CTRL_BUSY);// 等待 SD 卡总线非忙
        // 清除所有状态
        loopcnt = 0;
        // 第二步,通过 SD 总线的时钟线发送 74 个脉冲
        *P_SD_COMMAND_SETUP = C_SD_74CLK_START;
```

```
if (__SDDrv_WaitSDStatus__(C_SD_CMD_COMPLETE))// 命令传输完成标志是否置位
    return -EIO;
// 第三步，复位命令，应该没有反应
__SDDrv_SDCommand__(C_SD_CMD0, 0x00000000, response);//传输 CMD0 命令

loopcnt = 0;
__SDDrv_gi_CardType__ = SDCARD;
// 第四步，运行 ACMD41 直到卡完成上电时序
do                                                  // 循环直到 OCR 寄存器的最高位置 1
{
    if (__SDDrv_gi_CardType__ == SDCARD)
    {
        if (__SDDrv_SDCommand__(C_SD_CMD55, 0x00000000, response))
            __SDDrv_gi_CardType__ = MMCCARD;
                                                    //传输 CMD55 命令，卡类型为 MMC
    }
    if (__SDDrv_gi_CardType__ == SDCARD) // ACMD 41
        __SDDrv_SDCommand__(C_SD_ACMD41, 0x00200000, response);// SD 卡
    else
        __SDDrv_SDCommand__(C_SD_CMD1, 0x00200000, response);    // MMC 卡
    loopcnt = loopcnt + 1;
} while (((response[0] & 0x80000000) == 0) && loopcnt < 20000);
if (loopcnt == 20000)
    return 1;
// 第五步, CMD2 命令读 CID 寄存器
if(__SDDrv_SDCommandr2__(C_SD_CMD2, 0x00000000, response))
    return -EIO;
if (__SDDrv_gi_CardType__ == SDCARD)
{
    // 第六步，CMD3 读新的 RCA，SD 将会自动产生 RCA
    if(__SDDrv_SDCommand__(C_SD_CMD3, 0x00000000, response))
        return -EIO;
    __SDDrv_gui_RCA__ = response[0];
}
else
{
```

```
        // 第六步，CMD3 设置新的 RCA，MMC 需要分派一个新的 RCA
        if(__SDDrv_SDCommand__(C_SD_CMD3, 0xFFFF0000, response))
            return -EIO;
        __SDDrv_gui_RCA__ = 0xFFFF0000;
    }
    // 读状态
    __SDDrv_SDCommand__(C_SD_CMD13, __SDDrv_gui_RCA__, response);
    if (response[0] != 0x0700)
        return -EIO;
    // 第七步，CMD9 命令读 CSD 寄存器
    for (i=0;i<4;i++)
        response[i] = 0;
    if(__SDDrv_SDCommandr2__(C_SD_CMD9, __SDDrv_gui_RCA__, response))
        return -EIO;
    // 计算总的存储空间大小
    blocklen = ((response[1] & 0x000F0000)>>16)-8;
    c_size = ((response[1] & 0x000003FF)<<2) + (response[2]>>30);
    mult = ((response[2] & 0x00030000)>>15) + ((response[2] & 0x0000FFFF)>>15);
    __SDDrv_gi_SDCardTotalSector__ = (blocklen*(c_size+1))<<(mult + 2);
    timevalue = ((response[0] & 0x00000078)>>3);
    tranunit = response[0] & 0x00000007;
    if((timevalue == 0) || (tranunit > 3))
        return -EIO;
    maxspeed = SpeedTable[timevalue] * TranTable[tranunit] / 100;
    if (maxspeed > 13)
        // 增大时钟频率
        *P_SD_MODE_CTRL = C_BLOCKLEN_512BYTE
                        | C_MIN_CLOCK_SPEED
                        | C_SD_PORT_EN
                        | C_SD_DMA_EN;
    else
        // 增大时钟频率
        *P_SD_MODE_CTRL = C_BLOCKLEN_512BYTE
                        | C_MMC_CLOCK_SPEED
                        | C_SD_PORT_EN
                        | C_SD_DMA_EN;
```

```
    __SDDrv_SDCommand__(C_SD_CMD13, __SDDrv_gui_RCA__, response);
    __SDDrv_SDCommand__(C_SD_CMD7, __SDDrv_gui_RCA__, response);
    // SD 总线设置为 4 bits 模式
    if (__SDDrv_gi_CardType__ == SDCARD)
    {
        //降低块大小到 8 bytes
        *P_SD_MODE_CTRL = C_BLOCKLEN_8BYTE|(*P_SD_MODE_CTRL&0xffff);
        //ACMD51
        __SDDrv_SDCommand__(C_SD_CMD55, __SDDrv_gui_RCA__, response);
        if(__SDDrv_SDCommandr3__(C_SD_ACMD51, __SDDrv_gui_RCA__, response))
            return -EIO;
        if ((response[0] & 0x00000400) != 0x00)
        {
            //设置总线宽度为 4 位
            __SDDrv_SDCommand__(C_SD_CMD55, __SDDrv_gui_RCA__, response);
            __SDDrv_SDCommand__(C_SD_ACMD6, 0x00000002, response);
            *P_SD_MODE_CTRL = C_BLOCKLEN_512BYTE
                              | C_MIN_CLOCK_SPEED
                              | C_SD_PORT_EN
                              | C_SD_DMA_EN
                              | C_SD_BUS_4BIT;
        }
        else
            *P_SD_MODE_CTRL = C_BLOCKLEN_512BYTE
                              | C_MIN_CLOCK_SPEED
                              | C_SD_PORT_EN
                              | C_SD_DMA_EN;
    }
    __SDDrv_SDCommand__(C_SD_CMD13, __SDDrv_gui_RCA__, response);
    *P_SD_MODE_CTRL = C_BLOCKLEN_512BYTE|(*P_SD_MODE_CTRL&0xffff);
#ifdef CYGNUM_DEVS_SD_SPCE3200_DEBUG_MODE
    printf("Debug - SD Initial OK\n");
#endif
    return ENOERR;
}
```

关于 SPCE3200 的 SD 卡控制器及对 SD 卡的操作请参考《嵌入式微处理器 SPCE3200 原理及应

用》。

3、 SD 卡设备批量写操作函数 SD_WriteSector()

SD 卡设备批量写操作函数主要完成对 SD 卡数据的批量写操作。参考程序如下：

```
static int SD_WriteSector(void* disk, const void* buf_arg, cyg_uint32 *blocks, cyg_uint32 first_block)
{
    int i, ret = ENOERR;
#ifdef CYGNUM_DEVS_SD_SPCE3200_DEBUG_MODE
    printf("Debug - SD Write...Wait sd_ready sem...\n");
#endif
    cyg_semaphore_wait(&sem_sd_ready);              // 信号量 wait，等待 SD 设备空闲
#ifdef CYGNUM_DEVS_SD_SPCE3200_DEBUG_MODE
    printf("Debug - SD Writing...\n");
#endif
    if(__SDDrv_DrvSDCWriteMulCommand__(first_block))// 发送批量写命令
    {
        cyg_semaphore_post(&sem_sd_ready);          // 发送命令失败，post 信号量
        return 1;                                   // 返回错误标志
    }
    for(i=0; i<*blocks; i++)                        // 批量写数据
    {
        ret = __SDDrv_DrvSDCWriteMulData__(buf_arg+i*512);
        if(ret == 1)                                // 写数据失败
        {
            __SDDrv_DrvSDCWriteMulStop__();         // 写停止命令
            cyg_semaphore_post(&sem_sd_ready);      // post 信号量
            return ret;                             // 返回错误标志
        }
    }
    if(__SDDrv_DrvSDCWriteMulStop__())              // 写停止命令
    {
        cyg_semaphore_post(&sem_sd_ready);          // 写命令失败，post 信号量
        return 1;                                   // 返回错误标志
    }
    cyg_semaphore_post(&sem_sd_ready);              // 写成功，post 信号量
    return 0;
```

```
}
```

4、 SD 卡设备批量读操作函数 SD_ReadSector()

SD 卡设备批量读操作函数主要完成对 SD 卡的批量读操作。参考程序如下：

```
Cyg_ErrNo SD_ReadSector(void* disk, void* buf_arg, cyg_uint32 *blocks, cyg_uint32 first_block)
{
    int i, ret;
#ifdef CYGNUM_DEVS_SD_SPCE3200_DEBUG_MODE
    printf("Debug - SD Read...Wait sd_ready sem...\n");
#endif
    cyg_semaphore_wait(&sem_sd_ready);                  // 信号量 wait，等待 SD 设备空闲
#ifdef CYGNUM_DEVS_SD_SPCE3200_DEBUG_MODE
    printf("Debug - SD Reading...\n");
#endif
    if(__SDDrv_DrvSDCReadMulCommand__(first_block))  // 发送批量读命令
    {
        cyg_semaphore_post(&sem_sd_ready);// 命令发送失败，post 信号量，释放 SD 设备
        return 1;                                       // 返回错误标志
    }
    for(i=0; i<*blocks; i++)                            // 批量读操作
    {
        ret = __SDDrv_DrvSDCReadMulData__(buf_arg+i*512);
        if(ret == 1)
        {                                               // 批量读失败
            __SDDrv_DrvSDCReadMulStop__();              // 发送读停止命令
            cyg_semaphore_post(&sem_sd_ready);          // post 信号量
            return ret;                                 // 返回错误标志
        }
    }
    if(__SDDrv_DrvSDCReadMulStop__())                   // 读成功，发送读停止命令
    {
        cyg_semaphore_post(&sem_sd_ready);              // 发送命令失败，post 信号量
        return 1;                                       // 返回错误标志
    }
    cyg_semaphore_post(&sem_sd_ready);                // 成功，post 信号量，释放 SD 设备
    return 0;
```

```
}
```

5、 SD 卡读设备配置状态函数 SD_SetConfig()

SD 卡设备批量写操作函数参考程序如下：

```
static Cyg_ErrNo SD_GetConfig(disk_channel* chan, cyg_uint32 key, const void* buf, cyg_uint32*
len)
{
#ifdef CYGNUM_DEVS_SD_SPCE3200_DEBUG_MODE
    printf("Debug - SD Get Config\n");
#endif
    return -EINVAL;
}
```

这里没有读设备的任何配置状态，根据用户的实际情况可自行添加程序。

6、 SD 卡设备设置函数 sd_set_config()

SD 卡做为文件系统的宿主，为了配合 eCos 下的 Disk 组件的管理，需要对 CYG_IO_SET_CONFIG_DISK_MOUNT 和 CYG_IO_SET_CONFIG_DISK_UMOUNT 两个配置命令进行响应。SD 卡设备设置函数参考程序如下：

```
static Cyg_ErrNo SD_SetConfig(disk_channel* chan, cyg_uint32 key, const void* buf, cyg_uint32*
len)
{
    Cyg_ErrNo result    = ENOERR;
#ifdef CYGNUM_DEVS_SD_SPCE3200_DEBUG_MODE
    printf("Debug - SD Set Config\n");
#endif
    switch(key) {
      case CYG_IO_SET_CONFIG_DISK_MOUNT:          // SD 卡挂载状态
          break;
      case CYG_IO_SET_CONFIG_DISK_UMOUNT:         // SD 卡卸载状态
        if (0 == chan->info->mounts) {                    //  如果这时最后一次卡的卸载
            result = (chan->callbacks->disk_disconnected)(chan);//  标注卡处于未连接状态
        }
        break;
    }
    return result;
}
```

6.6 SD卡驱动程序范例

由于 SD 卡驱动程序的完整代码比较庞大，限于篇幅，这里不做介绍，请参考 eCos 源码对应目录下的文件（路径：ecos\packages\devs\sd\score\spce3200\current\src）。

6.7 向eCos数据库中添加SD卡驱动程序组件包

根据1.4.1加载驱动程序组件包到数据库中的办法，在ecos.db中的任意位置加如下程序段：

```
package CYGPKG_DEVS_SD_SPCE3200 {
    alias          { "SD support for Score SPCE3200"}
    directory      devs/sd/score/spce3200
    script         sd_spce3200.cdl
        description "
            This package contains hardware support for the SD Card
        on the Score SPCE3200 EV Board."
}
```

其中，CYGPKG_DEVS_SD_SPCE3200 是 SD 卡设备的宏标识，该宏的名称应与 sd_spce3200.cdl 文件中定义 cdl_package 的宏名称一致。

“devs/sd/score/spce3200”指示了 SD 卡设备驱动的路径，该路径是相对于/packages 目录的。

此时如果用eCos配置工具添加包时，就可以看见SD卡驱动程序组件包已经出现在eCos数据库中，如图 6.3所示。点击“Add”加载到平台上进行编译生成*.ecc库文件后，即可以使用SD卡驱动程序。

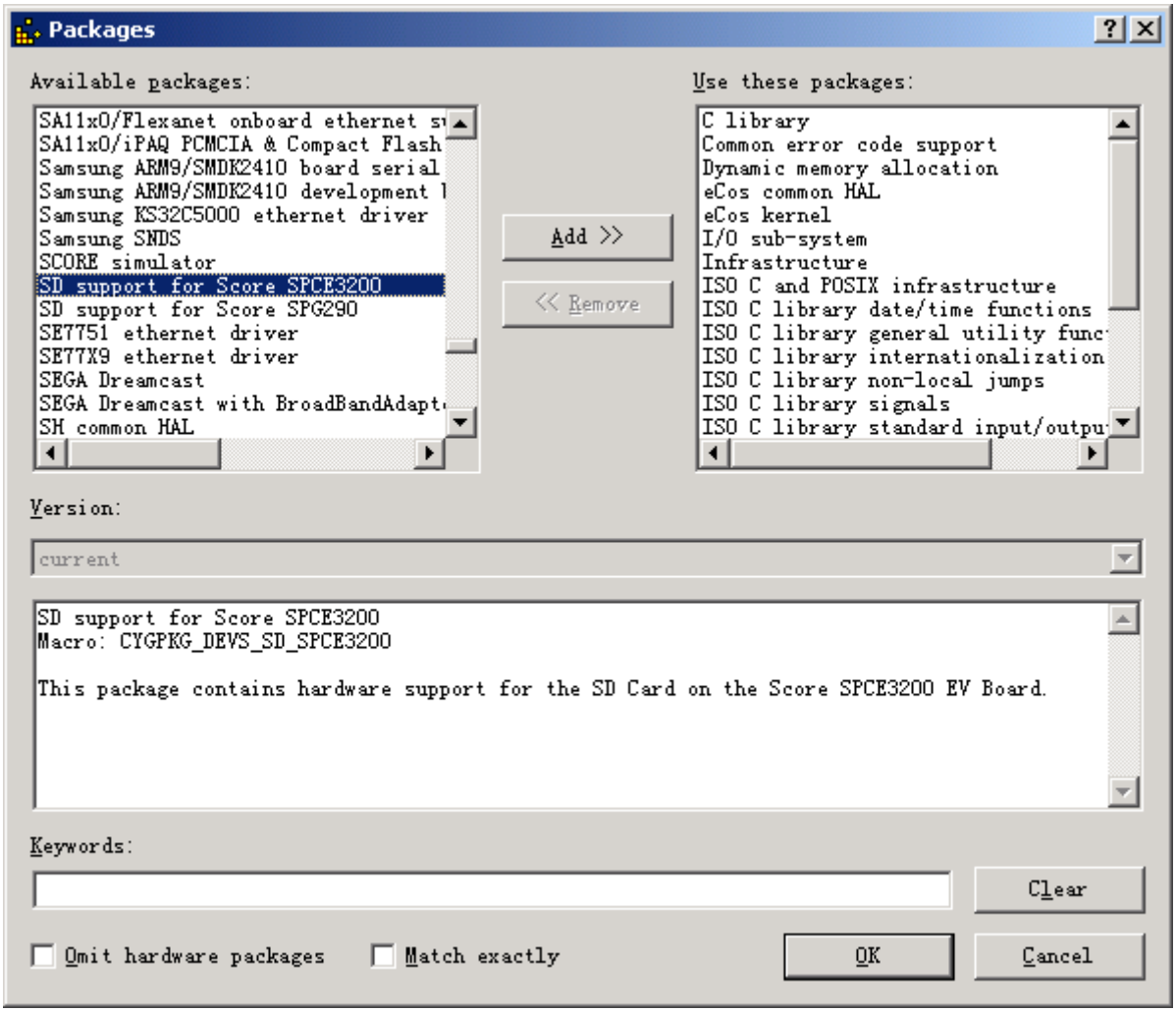

图 6.3 SD 卡驱动程序组件包在 eCos 数据库中界面

6.8 用户测试程序

在把 SD 卡设备驱动程序组件包加载到 SPCE3200 开发平台编译没有错误后，就可以编写测试程序验证驱动程序是否工作正常。

这里，我们使用 eCos 操作系统的 I/O 层提供的接口函数访问 SD 卡设备。在测试程序中首先向 SD 卡的两个扇区写入数据，然后读回，并校验数据是否正确。

参考程序如下：

```
//-----------------------------------包含相关头文件-----------------------------------//
#include <cyg/infra/testcase.h>          // 测试宏定义，如 CYG_TEST_FAIL_FINISH
#include <cyg/io/io.h>                   // 声明 I/O 子系统 API 函数
#include <cyg/kernel/kapi.h>             // 定义针对内核的 API 函数，同步机制的 API 函数等
#include <stdio.h>                       // 声明 printf 等调试函数


cyg_io_handle_t      hDrvSD;
```

```
//----------------------SD 卡驱动测试程序---------------------------//
void sd_thread(cyg_addrword_t data)
{
    cyg_uint32          len;
    cyg_uint8           ReadData[512*2];
    cyg_uint8           WriteData[512*2];
    cyg_uint32          i;

    if (ENOERR != cyg_io_lookup("/dev/sd", &hDrvSD)) //  查找 SD 设备
    {
        CYG_TEST_FAIL_FINISH("Error opening /dev/sd");
    }
    printf("Open /dev/sd OK\n");
    for(i = 0; i < 512*2; i++)                          //  初始化写缓冲区
    {
        WriteData[i] = (cyg_uint8)i;
    }
    memset(ReadData, 0, 512*2);                         //  初始化读缓冲区
    len = 2;
    if(ENOERR != cyg_io_bwrite(hDrvSD, WriteData, &len, 0))
                                                        //  从 0 扇区开始写入两个扇区的数据
    {
        CYG_TEST_FAIL_FINISH(“Error write /dev/sd”);
    }
    len = 2;
    if(ENOERR != cyg_io_bread(hDrvSD, ReadData, &len, 0))
                                                        //  从 0 扇区开始读取两个扇区的数据
    {
        CYG_TEST_FAIL_FINISH(“Error read /dev/sd”);
    }
    for(i = 0; i < 512 * 2; i++)                        //  校验读出的数据是否正确
    {
        if(ReadData[i] != (cyg_uint8)i)
        {
            printf(“Error data, Address = %x; Value = %x\n”, i, ReadData[i]);
        }
```

```
        }
        CYG_TEST_FINISH("Test Over");
    }

    cyg_uint8 stack[4096];
    cyg_handle_t simple_threadA;
    cyg_thread thread_s;

    void cyg_user_start(void)
    {
        cyg_uint32        len;
        CYG_TEST_INIT();
        cyg_thread_create(4, sd_thread,
                          (cyg_addrword_t) 0,
                          "sd_thread",
                          (void *) stack, 4096,
                          &simple_threadA, &thread_s);     // 建立一个线程 sd_thread
        cyg_thread_resume(simple_threadA);
    }
```

6.9 使用FAT文件系统访问SD卡

除了通过 I/O 组件访问块设备之外，eCos 还提供了 FAT 文件系统用以管理块设备。FAT 文件系统的使用，可以大大方便嵌入式系统与 PC 系统之间的文件交换，并为应用程序提供更为方便的操作接口。下面介绍使用 eCos 下的 FAT 文件系统访问 SD 卡的步骤。

首先，确保 eCos 数据库内已经有 SD 卡的驱动，并且驱动程序运行没有问题。

运行eCos的配置工具，选择“Sunplus SPCE3200”平台，选择“default”模版，如图 6.4所示。

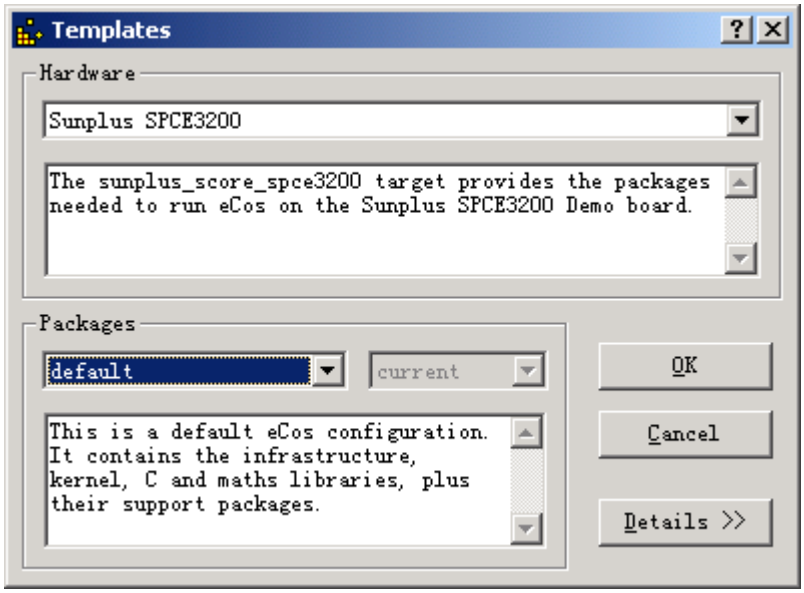

图 6.4 FAT 文件系统——建立工程模版

打开packages管理界面，在左侧找到“SD support for Score SPCE3200”，并单击“Add”按钮，如图 6.5所示。

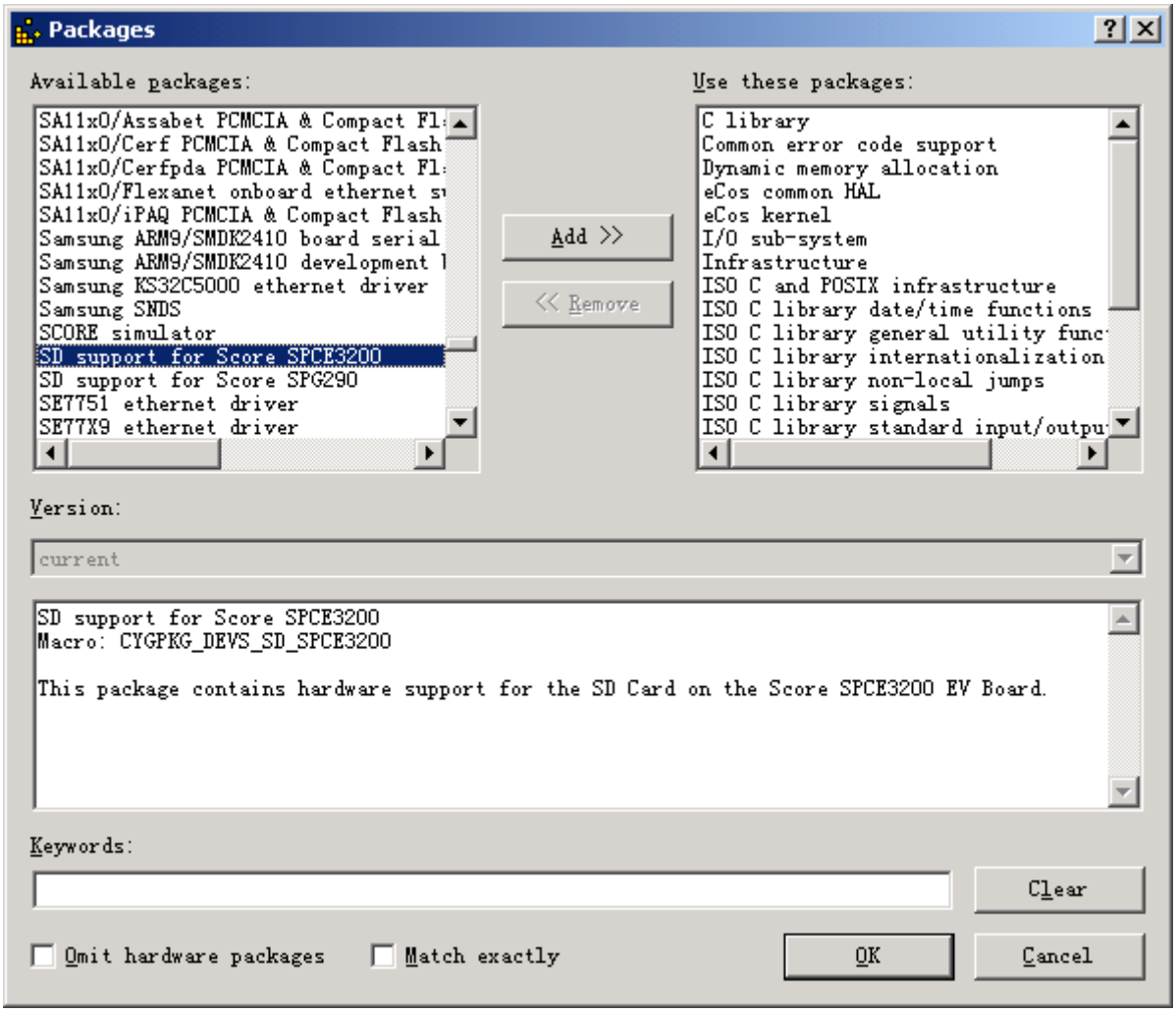

图 6.5 FAT 文件系统——添加 SD 卡驱动程序包

然后使用同样的方法，依次将下面几个组件添加至工程：

- Disk Device Driver
- Linux compatibility
- Block cache and access library
- File IO

最后，找到“FAT filesystem”组件，如图 6.6所示，单击“Add”添加至工程。

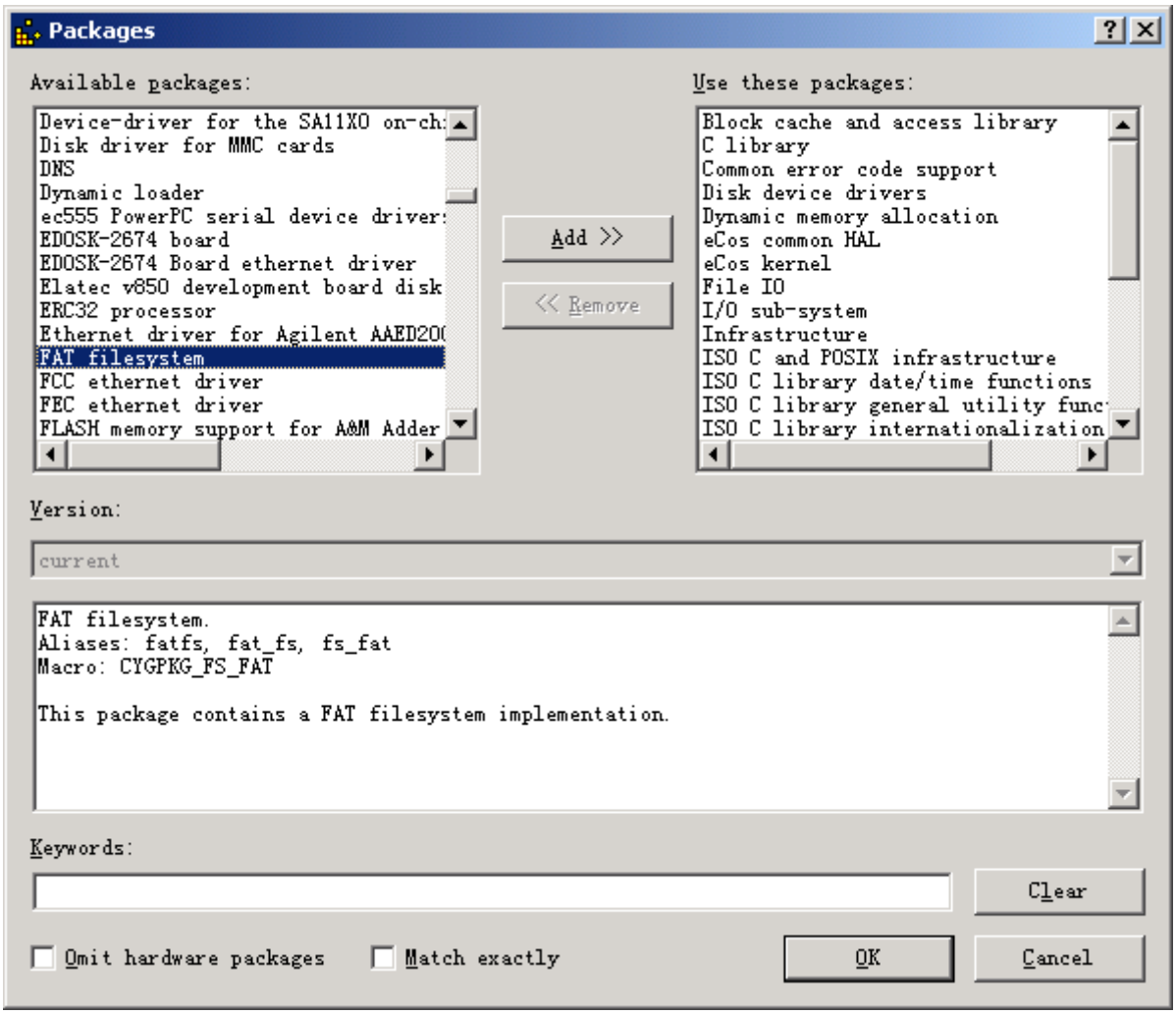

图 6.6 FAT 文件系统——添加 FAT 文件系统程序包

最后点击“OK”，完成组件的添加工作。在点击“OK”之后，会弹出图 6.7和图 6.8所示的警告，分别点击“Continue”和“是(Y)”，eCos配置工具将自动调整组件的配置。

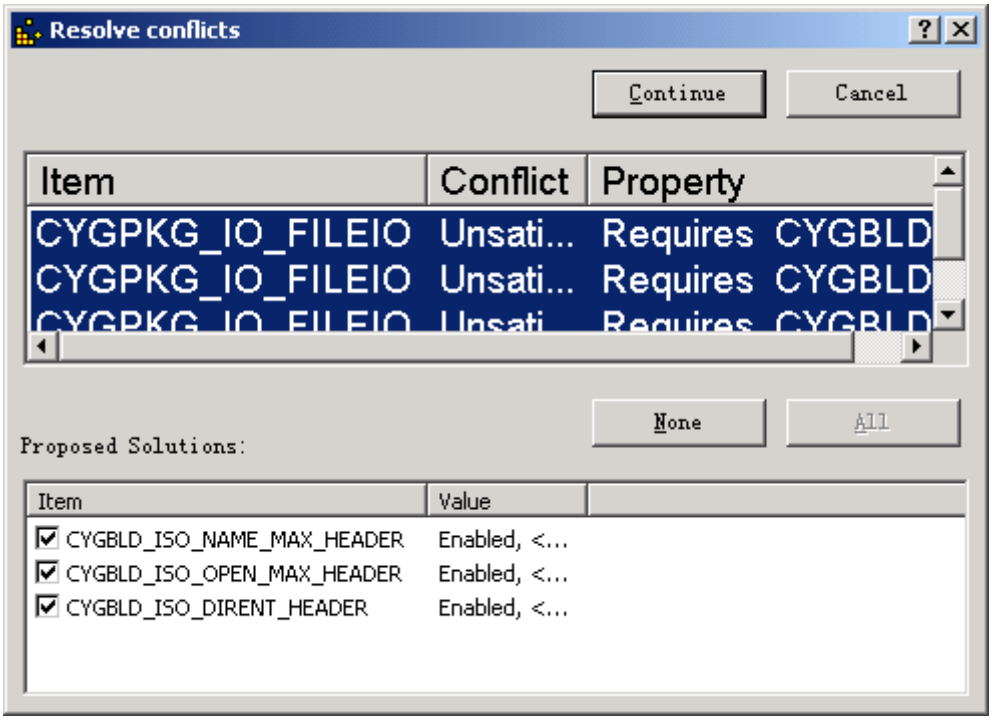

图 6.7 FAT 文件系统——组件冲突(1)

图 6.8 FAT 文件系统——组件冲突(2)

完成组件添加工作之后的eCos配置工具界面如图 6.9所示。

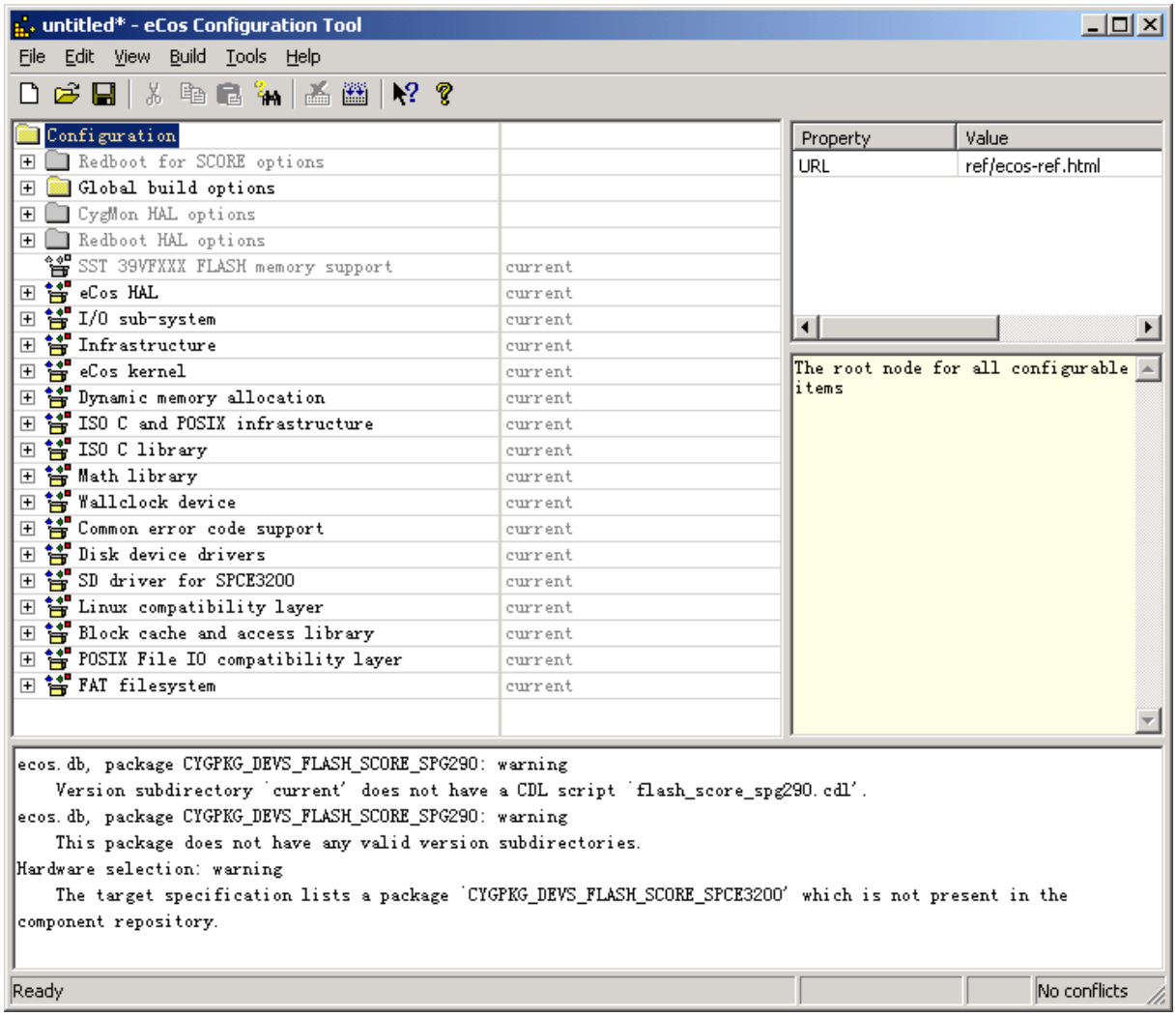

图 6.9 FAT 文件系统——工程建立完成

点击工具栏里的 Save 按钮，将工程保存为*.ecc 库。之后，点击工具栏里的 Build 按钮开始编译，最终生成带有 FAT 文件系统的库文件。

为了测试文件系统是否可以正常工作，在 eCos 源码目录下的/packages/fs/fat/current/tests/子目录内找到 fatfs1.c 文件，并将程序中所有的 mount 函数的第一个参数修改为“/dev/sd”。代码片段如下所示：

```
……（前面的代码省略）
int main( int argc, char **argv )
{
    int err;
    int existingdirents=-1;
#if defined(CYGSEM_FILEIO_BLOCK_USAGE)
    struct cyg_fs_block_usage usage;
#endif
```

```
    CYG_TEST_INIT();

    // ------------------------------------------------------------
    err = mount( "/dev/disk0/1", "/", "fatfs" );    // 修改为 err = mount("/dev/sd", "/", "fatfs");
                                                    // 程序中所有对 mount 函数的调用均按此修改
    if( err < 0 ) SHOW_RESULT( mount, err );

    err = chdir( "/" );
    if( err < 0 ) SHOW_RESULT( chdir, err );

    checkcwd( "/" );

    listdir( "/", true, -1, &existingdirents );
    // ------------------------------------------------------------
#if defined(CYGSEM_FILEIO_BLOCK_USAGE)
    err = cyg_fs_getinfo("/", FS_INFO_BLOCK_USAGE, &usage, sizeof(usage));
    if( err < 0 ) SHOW_RESULT( cyg_fs_getinfo, err );
    diag_printf("<INFO>: total size: %6lld blocks, %10lld bytes\n",
        usage.total_blocks, usage.total_blocks * usage.block_size);
    diag_printf("<INFO>: free size:    %6lld blocks, %10lld bytes\n",
        usage.free_blocks, usage.free_blocks * usage.block_size);
    diag_printf("<INFO>: block size: %6u bytes\n", usage.block_size);
#endif
    // ------------------------------------------------------------
    createfile( "/foo", 20257 );
    checkfile( "foo" );
    copyfile( "foo", "fee");
    checkfile( "fee" );
    comparefiles( "foo", "/fee" );
    diag_printf("<INFO>: mkdir bar\n");
    err = mkdir( "/bar", 0 );
    ……（后面的代码省略）
```

可以看到，eCos 下的 FAT 文件系统使用 mount()函数将 SD 卡设备映射到某一个指定的路径，此后，用户使用各种文件管理函数对该路径操作时，将被映射为对指定块设备的操作。

用户可以按照 SD 卡驱动程序的形式，编写其他与 SD 卡类似的块设备的驱动程序。

7 附录

7.1 SPCE3200 在eCos系统上的设备名列表

表 7.1 SPCE3200 在 eCos 系统上的设备名列表

设备	设备名	驱动程序路径
IOA	dev/ioa	ecos\packages\devs\gpio\score\spce3200\IOA
IOB	dev/iob	ecos\packages\devs\gpio\score\spce3200\IOB
GPIO（ADC 接口）	dev/gpioadc	ecos\packages\devs\gpio\score\spce3200\ADC
键盘（1*8）	dev/1mul8	ecos\packages\devs\kbd\score\spce3200\1MUL8
键盘（4*4）	dev/4mul4	ecos\packages\devs\kbd\score\spce3200\4MUL4
LED 灯（1*8）	dev/led	ecos\packages\devs\light\score\spce3200
ADC	dev/adc	ecos\packages\devs\adc\score\spce3200
UART	dev/uart	ecos\packages\devs\serial\score\spce3200
SPI	dev/spi	ecos\packages\devs\spi\score\spce3200
I2C	dev/i2c	ecos\packages\devs\i2c\score\spce3200
JoyStick	dev/joystick	ecos\packages\devs\joystick\score\spce3200
PS/2	dev/ps2	ecos\packages\devs\ps2\score\spce3200
PC 机键盘	dev/kbd	ecos\packages\devs\kbd\score\spce3200\kbd
鼠标	dev/mouse	ecos\packages\devs\mouse\score\spce3200
Touch Panel	dev/Touch	ecos\packages\devs\touch\score\spce3200
IPOD	dev/ipod	ecos\packages\devs\ipod\score\spce3200
SD 卡	dev/sd	ecos\packages\devs\sd\score\spce3200
LCD	dev/lcd	ecos\packages\devs\lcd\score\spce3200
TV	dev/tv	ecos\packages\devs\tv\score\spce3200
以太网	dev/eth	ecos\packages\devs\eth\score\spce3200

注：为了规范，这里约定了各设备名及设备驱动路径，并不是所有的驱动程序都在源代码里提供。

7.2 eCos中各设备的key值列表（SPCE3200 平台）

表 7.2 eCos 中各设备的 key 值列表

设备	key 值

serial	0x01XX
tty	0x02XX
dsp	0x03XX
termios	0x04XX
FLASH	0x06XX
DISK	0x07XX
CAN	0x08XX
I/O 子系统	0x10XX~0x11XX
IOA	0x111X
IOB	0x112X
GPIO(ADC)	0x113X
键盘（1*8）	0x121X
键盘（4*4）	0x122X
LED 灯（1*8）	0x13XX
ADC	0x14XX
UART	0x15XX
SPI	0x16XX
I2C	0x17XX
JoyStick	0x18XX
PS/2	0x19XX
PC 机键盘	0x1aXX
鼠标	0x1bXX
Touch Panel	0x1cXX
IPOD	0x1eXX
SD 卡	0x1fXX
LCD	0x20XX
TV	0x21XX
以太网	0x22XX

注：

1、 以上“X”表示任意十六进制数；

2、 key 值定义不能重复；

3、 前 8 行为 eCos 系统已经占去的 key 值。