Writing PCMCIA Device Drivers

SunSoft, Inc.
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

THE NETWORK IS THE COMPUTER™

Copyright 1997 Sun Microsystems, Inc. 2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A. All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X / Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, SunSoft, SunDocs, SunExpress, and Solaris are trademarks, registered trademarks, or service marks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK and Sun™ Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6 / 87) and FAR 52.227-19(6 / 87), or DFAR 252.227-7015(b)(6 / 95) and DFAR 227.7202-3(a).

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright 1997 Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, Californie 94043-1100 Etats-Unis. Tous droits réservés.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Des parties de ce produit pourront être dérivées des systèmes Berkeley BSD licenciés par l'Université de Californie. UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X / Open Company, Ltd.

Sun, Sun Microsystems, le logo Sun, SunSoft, SunDocs, SunExpress, et Solaris sont des marques de fabrique ou des marques déposées, ou marques de service, de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

L'interface d'utilisation graphique OPEN LOOK et Sun™ a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui en outre se conforment aux licences écrites de Sun.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" ET AUCUNE GARANTIE, EXPRESSE OU IMPLICITE, N'EST ACCORDEE, Y COMPRIS DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DE LA PUBLICATION A REPONDRE A UNE UTILISATION PARTICULIERE, OU LE FAIT QU'ELLE NE SOIT PAS CONTREFAISANTE DE PRODUIT DE TIERS. CE DENI DE GARANTIE NE S'APPLIQUERAIT PAS, DANS LA MESURE OU IL SERAIT TENU JURIDIQUEMENT NUL ET NON AVENU.

Please
Recycle

Adobe PostScript

Contents

Figures

Preface

Writing PCMCIA Device Drivers presents a brief overview of PC Card technology and provides information on the Solaris™ UNIX® implementation of the PC Card Standard. It also describes how to write a PC Card driver for the Solaris environment and presents a complete structured walkthrough of a sample Solaris PC Card driver.

Who Should Use This Book

This document is intended for UNIX device driver writers who are writing PC Card drivers and for PC Card driver writers who need information on the Solaris implementation of the PC Card Standard.

How This Book Is Organized

This document is organized into the following chapters.

Chapter 1, "PC Cards and Drivers," provides a brief overview of PC Card technology and presents introductory information on the Solaris UNIX implementation of the PC Card Standard.

Chapter 2, "PCMCIA System Architecture," provides overview information on the PCMCIA system architecture.

Chapter 3, "Solaris Card Services," discusses the Solaris implementation of the Card Services Standard.

Chapter 4, "Card Information Structure (CIS)," describes the Card Information Structure (CIS).

Chapter 5, "PC Card Driver Autoconfiguration," describes the support a PC Card driver must provide for driver autoconfiguration.

Chapter 6, "PC Card Event Management," discusses event handling in the PCMCIA framework.

Chapter 7, "PC Card Configuration," discusses allocation of system resources to a PC Card.

Chapter 8, "Portable PC Card Drivers," discusses portability issues that a driver writer should be aware of when writing a PC Card driver that is portable across Solaris platforms.

Chapter 9, "PCMCIA Parallel Port Driver," presents the complete code for a PCMCIA parallel port driver and shows how a similar driver could be written.

Related Books

Driver writers can benefit from reading the following book before writing a PC Card driver for the Solaris environment:

- *Writing Device Drivers*, SunSoft, 1997.

For detailed information on PCMCIA, see the PCMCIA specification:

- PCMCIA / JEIDA, *PC Card Standard*, Februrary 1995.

Ordering Sun Documents

The SunDocs℠ program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals using this program.

For a list of documents and how to order them, see the catalog section of the SunExpress™ Internet site at `http://www.sun.com/sunexpress`.

What Typographic Changes Mean

The following table describes the typographic changes used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	**Meaning**	**Example**
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% You have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`** `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.

PC Cards and Drivers 1

This chapter provides a brief overview of PC Card technology and the PCMCIA organization. It also presents introductory information on the Solaris UNIX implementation of the PC Card 95 Standard.

What Is PCMCIA?

PCMCIA is an acronym for Personal Computer Memory Card International Association. The PCMCIA develops standards for low-cost, credit card-sized, interchangeable computer peripheral devices called PC Cards. The published specification for these devices is called the PC Card Standard.

The PC Card Standard specifies the electrical and mechanical interface between the PC Card device and the socket it is plugged into. The standard also defines the software support needed to configure and control the device. The PCMCIA software architecture enables PC Cards to be configured dynamically at the time of insertion, thus providing hot-plugging capability. Hot plugging allows users to add (or remove) peripheral devices to their computer system without rebooting it.

Terminology: PCMCIA or PC Card

The terms PCMCIA and PC Cards are often used interchangeably, and PCMCIA can refer to any of the following:

- A set of electrical, mechanical, and software specifications (defined in the PC Card Standard)

- The organization that created the PC Card Standard
- The class of low-cost, credit card-sized peripheral devices

In this manual, physical devices are referred to as PC Cards, and drivers are referred to as PC Card drivers. PCMCIA refers to the system framework that supports PC Cards and drivers as well as to the PCMCIA.

PCMCIA Background

As the size of chips decreased during the 1980s, the physical size of computer systems also decreased, and laptop computing became more prevalent. Hardware vendors became concerned about the proliferation of proprietary device designs for laptop computing, and recognized the need to standardize on a physical card size for peripheral devices. Vendors also acknowledged the need for common electrical specifications and standard software access mechanisms.

In 1985, the Japan Electronic Industry Development Association (JEIDA) was formed to promote memory cards, personal computers, and other portable "information products". In 1989, the PCMCIA was formed specifically to standardize memory cards. The membership of the PCMCIA includes peripheral vendors, software developers, and computer and semiconductor manufacturers.

In 1990, the PCMCIA completed the PCMCIA Standard Release 1.0, which supported only memory card devices. The specification included 68-pin and socket connectors originally defined by JEIDA as well as the Card Information Structure (CIS) format and the electrical and physical requirements for memory cards. PCMCIA Standard Release 2.0 added support for I/O cards.

The latest version of the PCMCIA Standard is called the PC Card Standard or PC Card 95 Standard, and is the result of the convergence of the PCMCIA and JEIDA standards organizations. The PC Card 95 Standard was released in February 1995.

For detailed information on the PC Card 95 Standard or the PCMCIA organization, contact the PCMCIA organization at `http://www.pc-card.com/`.

1

PCMCIA in the Solaris Environment

The Solaris system provides a UNIX implementation of the PC Card Standard, bundled with a number of drivers for common PC Cards. The Solaris PCMCIA implementation eliminates the x86 architectural dependencies in the original PCMCIA specification in favor of an open systems approach. In particular, the calling and argument passing conventions of the PCMCIA interface standard have been respecified in ANSI C language bindings in the Solaris environment to allow for both platform and operating system independence. Solaris PCMCIA enables drivers to be written that are independent of the particular platform and host architecture.

Card Services Interface

A major component of the PCMCIA architecture is Card Services. The Card Services interface forms a standardized software layer between PC Card devices and the operating system.

Card Services routines allow drivers to manage functions associated with card insertion and card removal for hot-pluggable devices. These routines ensure that users do not have to reboot their operating systems after plugging PC Cards into (or removing PC Cards from) PC Card sockets.

As implemented in the Solaris system, Card Services provides driver source code compatibility across Solaris platforms that support PCMCIA adapter hardware. Binary code compatibility is provided for PC Card drivers across different platform implementations that share the same instruction set architecture.

Card Services Binding

Because of its history, the PC Card Standard is bound both to PC operating systems (such as MS-DOS and Windows) and x86 instruction set architecture. The PC Card Standard contains functional notation that does not correspond exactly to ANSI C bindings but instead is mapped to Intel-specific assembly language. The contents of various x86 registers, for example, are passed as pointers and arguments to functions. The concepts behind IRQ routing and the number of bits required to express I/O addresses are unique to x86 platforms.

The PCMCIA standards body attempted to remedy the porting problems that arose from the bias toward x86 architecture by including a *bindings* section in the specification, where the specifics of Card Services functions under various operating system combinations can be addressed. The bindings section attempted to make up for the reliance on MS-DOS-based operating systems and x86 hardware by providing the capability to bind PCMCIA to operating systems not based on MS-DOS.

The Solaris system modifies and augments the basic PC Card Standard in the following ways:

- Provides its own C language bindings
- Adds common access functions (equivalent to the `ddi_put8`(9F) and `ddi_get8`(9F) family of functions) to Solaris Card Services to provide driver compatibility across different platforms
- Adds Solaris-specific utility functions

There are other differences between the PC Card Standard and Solaris Card Services. Since in Solaris Card Services, there is only one instance of a PC Card driver attached to a function on a PC card, the PC Card driver does not need to keep track of the socket number in which the card is inserted. In addition, Solaris Card Services can dispatch I/O interrupts to the appropriate PC Card driver's interrupt handler.

The Solaris Card Services implementation is discussed in subsequent chapters. For more information on the PCMCIA architecture, see Chapter 2, "PCMCIA System Architecture."

PC Card Device Drivers

While PC Card device drivers are very similar to other Solaris device drivers, they differ from other drivers in their use of Card Services as an interface to the PCMCIA framework. PC Card drivers use Card Services to:

- Register with the PCMCIA framework
- Determine card characteristics
- Configure cards
- Activate cards
- Modify card resources
- Handle events

In other aspects, PC Card drivers are the same as other Solaris character or block drivers. PC Card drivers, for example, use the standard Device Driver Interface/Driver-Kernel Interface (DDI/DKI) entry points and interface with the same DDI/DKI functions and structures as other Solaris device drivers.

The remainder of this book discusses the specifics of implementing PC Card drivers. For background information on device drivers in the Solaris system and for specific information on non-PC Card character and block drivers, see the *Writing Device Drivers* manual.

Loading a PC Card Driver

A PC Card driver is loaded like any other Solaris driver. See *Writing Device Drivers* for information on loading device drivers.

Sample PC Card Drivers in Solaris

Complete source code for the following sample drivers is included in the Solaris DDK.

PC Memory Card Driver (`pcsram`)

The `pcsram` driver is an SRAM/DRAM memory card client driver providing disk-like I/O access to SRAM/DRAM memory cards and nonvolatile SRAM/DRAM memory cards. `pcsram` can be used for system memory expansion or as a pseudo-floppy disk type of device containing an MS-DOS or UFS file system. This driver currently supports only a single partition.

PC Card I/O Driver (`pcepp`*)*

The `pcepp` driver is a character device driver for parallel port printers. It supports all cards providing a standard PC-compatible parallel port. It is a unidirectional STREAMS driver that interfaces with Solaris printer administration. For more information on the `pcepp` driver, see Chapter 9, “PCMCIA Parallel Port Driver.”

1

PCMCIA System Architecture 2

This chapter discusses the Solaris implementation of the PCMCIA system architecture. It describes the PCMCIA system framework and the relationship of the major components of the architecture.

Solaris PCMCIA System Architecture

The Solaris PCMCIA system architecture includes PC Card drivers, Solaris Card Services, PCMCIA bus adapter drivers, and a user events daemon. Solaris PCMCIA supports PC Card hardware, such as PC Card devices, and PCMCIA bus adapter hardware, which includes sockets and controllers.

Figure 2-1 illustrates the relationship among the components in the Solaris PCMCIA architecture. This architecture is described in more detail in the sections that follow.

PC Card Driver

A PC Card driver provides all the normal services associated with device drivers, such as standard open, close, read, and write entry points. In addition to its normal driver responsibilities, a PC Card driver calls the Card Services interface, which provides PCMCIA-specific functionality.

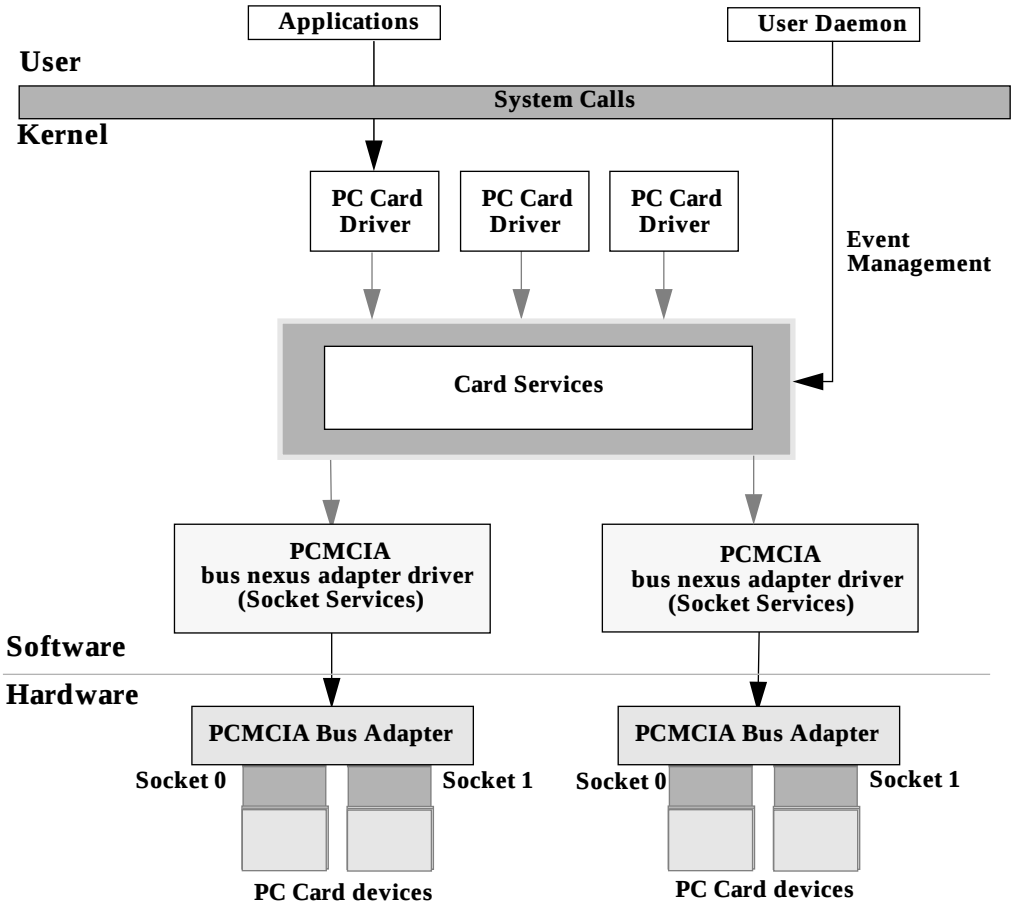


Figure 2-1 PCMCIA System Architecture

Because the PCMCIA framework provides interfaces that allow the PC Card driver to interact directly with Card Services, the PC Card driver is responsible for two important functions:

- Configuring the PC Card – Through Card Services and the Card Information Structure, the PC Card driver determines an appropriate configuration for the PC Card.

- Handling PC Card events – The PC Card driver responds to events such as card insertion and removal.

For a complete example of a PC Card driver, see Chapter 9, “PCMCIA Parallel Port Driver.”

Card Services

Card Services provides operating system services to PC Card drivers, such as:

- Resource management
- Event notification
- Card identification and configuration
- Access to the Card Information Structure (CIS)

Card Services also manages access to PC Card hardware and system resources by multiple clients. Architecturally, Card Services is the layer between PC Card drivers and PCMCIA bus nexus adapter drivers, which interface directly with the operating system. For more information on Card Services, see Chapter 3, “Solaris Card Services”.

PCMCIA Bus Nexus Adapter Drivers and Socket Services

PCMCIA adapters provide the physical slots or *sockets* for the PC Card(s) and a hardware interface between the PCMCIA bus and the system bus. The PC Card Standard specifies a standard interface to the adapter hardware called Socket Services. The Socket Services API is a hardware-independent interface between Card Services and the software layer that controls a particular adapter.

The PCMCIA Socket Services layer corresponds to a nexus driver in the Solaris architecture. In the Solaris system, bus nexus adapter drivers control all access to the hardware, with a separate nexus driver required for each different type of adapter hardware. Each PCMCIA bus nexus adapter driver is implemented as a private interface that encapsulates much of the standard PCMCIA Socket Services functionality. Functions similar to Socket Services are implemented as hardware-specific calls for each adapter supported by the platform.

Depending on the PCMCIA adapters on the platform, one or more of the bus nexus adapter drivers manage the interface between adapter hardware and (through Card Services) the PC Card drivers. This insulates the PC Card drivers from the specifics of the PCMCIA bus adapter hardware.

PCMCIA bus nexus adapter drivers are provided with the Solaris system, are platform dependent, and are configured in a manner that reflects the hardware configuration within the platform. Different bus nexus adapter drivers are available for different platforms supporting PC Cards, such as SPARC and x86.

Note – Although Card Services communicates to bus nexus adapter drivers through the PC Card Standard Socket Services interfaces, these interfaces are not exported to PC Card drivers or to any other system component.

Bus Nexus Adapter Driver Functionality

PCMCIA bus nexus adapter drivers support the following functions:

- Status interrupt generation
- Socket status reporting
- PC Card interrupt handling
- Memory and I/O mapping

In addition, the Solaris PCMCIA bus nexus adapter driver must:

- Identify cards in sockets
- Create device nodes if necessary
- Provide an interface to the PCMCIA event management driver

The encapsulation of these functions by the bus nexus adapter driver makes it possible to implement all of Card Services and much of the remainder of the PCMCIA framework in an architecture-neutral way. Only the bus nexus adapter drivers need be different on different adapter hardware.

This encapsulation has the added benefit of allowing driver writers to concentrate on a relatively small number of standard function calls and data structures. The major benefits include a significant decrease in the overhead costs of driver development and an increase in portability of drivers.

The bus nexus adapter drivers provide the core framework functionality and perform PCMCIA functions at the request of the PC Card driver. For example, when the PC Card driver requests hardware services on behalf of its device hardware (such as requesting an IRQ), it calls Card Services, which in turn calls the bus nexus driver adapter driver interface to perform the operation.

Event Status Changes and Interrupts

During configuration, the bus nexus adapter driver is programmed by the PC Card driver (through Card Services) to recognize addresses that reside on the PC Card and map them into system memory. Subsequently, when a card insertion or removal event occurs, the bus nexus adapter driver detects the event and relays information by generating socket status change interrupts, which are sent by the adapter driver to Card Services.

Card Services then calls back the adapter driver to determine the type and source of the interrupt. After this has been determined, Card Services relays the information to the PC Card driver. The PC Card driver uses Card Services to instruct the bus nexus adapter driver on the appropriate action to take to service the interrupt. This process is illustrated in Figure 2-2.

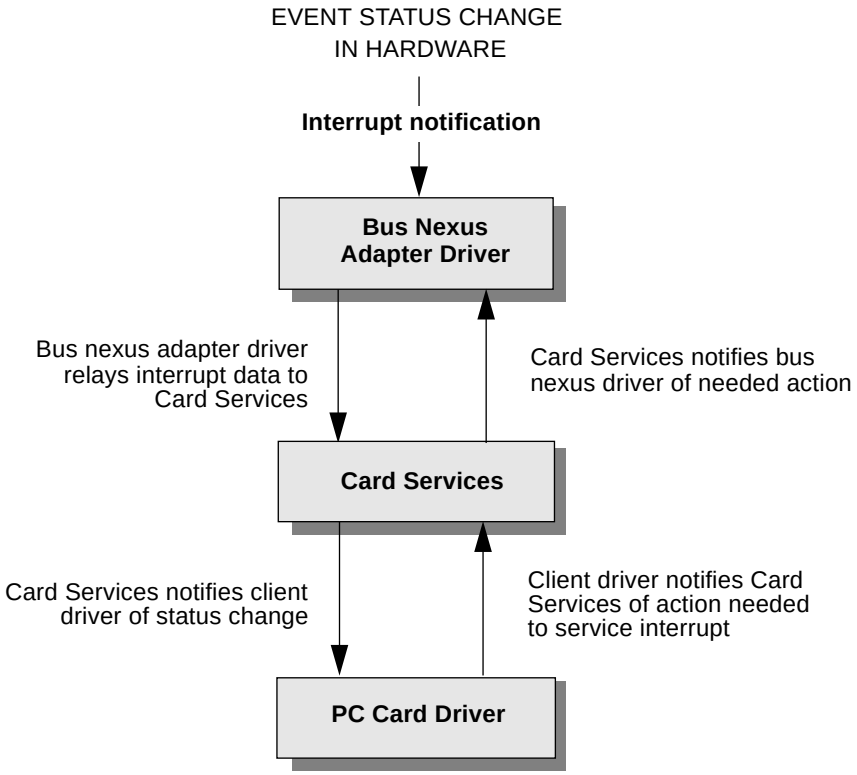


Figure 2-2 Event Status Change Process

PCMCIA Event Management (User) Daemon

The PCMCIA Event Management user daemon (`pcmciad`) is a system component that provides communication between the kernel and other user-level system components. It provides user-level hot plugging and volume management support for PC Cards. In addition, it provides services to the Solaris operating system to manage the device tree for all PC Cards in the system.

More specifically, the `pcmciad` Event Management daemon performs the following functions:

- Handles PCMCIA framework and card events. For example, it loads and attaches a PC Card driver on card insertion.
- Causes `/devices` entries to be created.
- When a PC Card driver creates a minor node, the daemon creates `/dev` links by invoking the standard link generator commands such as the `disks`(1M) and `devlinks`(1M) commands.

Solaris Card Services 3

This chapter discusses the Solaris implementation of the Card Services Standard.

Solaris Card Services Interface Overview

Solaris Card Services is an architecture-neutral implementation of the PC Card Services Standard. As illustrated in Figure 3-1, Solaris Card Services defines the software interface between PC Card device drivers and the bus nexus adapter drivers that manage the PC Card hardware. The Card Services interface is independent of the hardware that manipulates PC Cards. As a result, Card Services enables a PC Card driver to control the adapter without knowledge of the specific adapter hardware.

The Solaris PCMCIA framework maps the Card Services interface into hardware-specific functions through bus nexus adapter drivers. In the Solaris system, a PCMCIA nexus is the child of the system bus nexus, and a PCMCIA nexus driver is the parent of all PC Card drivers for cards in that adapter's sockets. Unlike other device drivers in the Solaris architecture, a PC Card driver does not interact directly with its parent nexus. Instead, as specified by the PC Card Standard, the PC Card driver calls into the Card Services layer, and the Card Services layer calls into the nexus adapter driver.

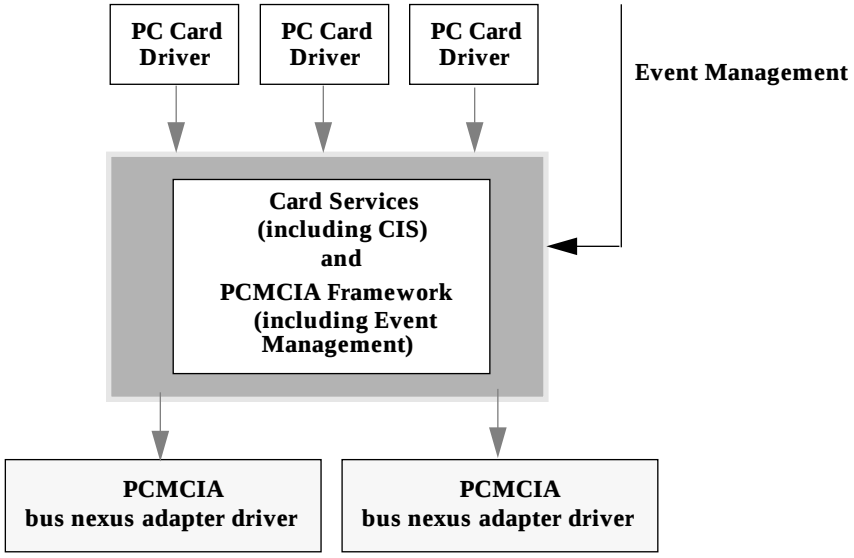


Figure 3-1 PCMCIA Card Services

PC Card drivers use the Card Services interfaces to determine hardware configuration and manage allocation of system resources. In addition, drivers use Card Services interfaces to handle PC Card insertion and removal. PC Card drivers are clients of Card Services; other Card Services clients include application programs and utility programs.

The Solaris implementation of Card Services is a collection of function entry points in the kernel. These functions are located in a kernel module that is not part of the device driver hierarchy. The module handles all Card Services requests from all PC Card device drivers and makes the calls into the appropriate parent nexus.

There can be only one Card Services implementation per host system. This is in contrast to PCMCIA bus nexus adapter drivers, where there can be multiple bus nexus drivers to accommodate multiple adapters.

Card Services Functionality

Card Services provides the following operating system services to PC Card drivers:

- Driver registration
- PC Card identification and configuration
- Event notification
- Resource allocation

The following sections provide overview information on these services.

Driver Registration

To inform Card Services of its presence, a driver must register with Card Services by calling `csx_RegisterClient`(9F). Driver registration is typically done in the driver's `attach(`9E) entry point. In the registration call, the driver specifies whether it is a memory or I/O client, and it also notifies Card Services of its interest in certain events.

The Card Services functions require the driver to pass in a client handle that uniquely identifies the requesting driver instance. The handle is returned by `csx_RegisterClient`(9F) as an opaque value to Card Services.

For more information on driver autoconfiguration routines, see Chapter 5, "PC Card Driver Autoconfiguration."

PC Card Identification and Configuration

The PC Card Standard specifies a structure called the *Card Information Structure* (CIS), which stores PC Card identification and configuration information. This structure enables the system software to configure the card without reference to any external card information, thereby making a PC Card self-identifying. Most PC Cards provide a CIS structure.

The CIS structure provides a variety of information about PC Card attributes, such as device size. Card Services configuration software uses the CIS structure to determine the configuration requirements of the PC Card.

See Chapter 4, "Card Information Structure (CIS)," for more information on the CIS structure. See Chapter 7, "PC Card Configuration," for information on PC Card configuration.

Event Notification

PCMCIA events inform the PC Card driver of changes to the state of the PC Card or changes in the system. Card Services provides a way to abstract the physical event, such as card insertion or removal, into a software callback mechanism in order to inform client software (through PC Card drivers) that an event has taken place. A PC Card driver may receive callbacks for any set of predefined events and may receive notification that an event has occurred at any time.

Card Services notifies clients of hardware and software events through an event handler. Information about events is passed back to the client event handler based on the type of event that occurred.

The event handler is registered with Card Services. At registration time, the PC Card driver supplies an address for the event handler and a description of the events it wants to be notified of. When events occur, the driver event handler processes known events, ignores unknown events, and returns a status result code to Card Services. For more information on how the driver handles PCMCIA events, see Chapter 6, "PC Card Event Management."

Resource Allocation

A PC Card driver interacts with Card Services to determine a PC Card's characteristics and to configure and activate the card based on the card's resource requirements. Card Services coordinates the mapping of PC Card resources to system resources.

System Resources

PC Cards may require the use of three types of system resources: I/O address space, interrupt requests (IRQs), and memory address space. A client acquires resources through requests to Card Services. If the resource is available and the socket hardware is capable of using it, Card Services allocates the resources.

Card Services maintains a table of system resources available to PC Cards and sockets. Resources are allocated to PC Cards through the `csx_RequestIO`(9F), `csx_RequestIRQ`(9F), and `csx_RequestWindow`(9F) functions. These functions are discussed in Chapter 7, "PC Card Configuration."

For most efficient system resource utilization, PC Card drivers must deallocate resources obtained during card insertion and reacquire the resources when the card is reinserted.

PC Card Resources

PC Cards have three distinct address spaces on the card: attribute memory space, common memory space, and I/O space. Each address space is 64 Mbytes in length, although usually only a fraction of the available address space is used by a PC Card.

Attribute memory is typically used to store CIS configuration information and configuration registers. This configuration information is used by the PCMCIA framework during card configuration. Common memory typically contains shared memory registers and may contain CIS information. A PC Card driver maps its memory space into allocated system memory and then specifies the area of the PC Card that appears in the allocated memory. PC Card I/O space contains registers and may contain shared memory.

To determine what resources a particular card provides in each of its address spaces, a device driver writer should consult the specifications for the card.

Solaris Modifications to the Card Services Interface

The PC Card Standard is oriented to the MS-DOS operating system and the x86 processor architecture. The standard specifies that Card Services has a single-function entry point through which all Card Services requests are vectored. This single entry point is specified to require the same number of arguments no matter which Card Services function is being requested. The arguments specified in the standard are of fixed type, and different arguments for different Card Services calls must be packed as untyped bytes into a variable-length byte array.

In the current PC Card Standard, each processor type, processor mode (protected, real, and so on), subroutine call method (near, far), and operating system on which Card Services is implemented specifies an implementation-specific binding that dictates the details required to make the call. The model is similar to an MS-DOS INT 21 function call.

This design hinders driver portability for operating systems such as the Solaris system that run on multiple system architectures and multiple processors. As part of Sun's efforts on behalf of the PC Card Standard, a C language calling convention has been developed. This binding implements a Card Services function entry point for each Card Services function with a variable argument list of pointers to function-specific structures and/or opaque data types.

Details such as the sizes of various address and data types, byte ordering for shared structure members, and generic system resources managed by the kernel are kept hidden from the driver developer. Selective data hiding allows a device driver to be source compatible between the Solaris system on the SPARC architecture and the Solaris system on the x86 architecture.

Certain differences between the SPARC and x86 architectures, such as card I/O port accesses, are handled through Card Services common access functions. Some adapters for SPARC machines map virtual registers into the device driver virtual memory address space, while other adapters on an x86 machine, for example, may use privileged I/O instructions.

Solaris Card Services differs from the PC Card Standard in that the Solaris Card Services functions are strongly typed with explicit parameters. For example, the following call is similar to the PC Card Standard Card Services specification, but eliminates null arguments and variable-length argument lists.

```
int32_t csx_RegisterClient(client_handle_t *, client_reg_t *)
```

Solaris Card Services Interfaces

Solaris Card Services provides interface functions that allow PC Card drivers to communicate with devices. These interfaces fall into the following functional groups:

- PC Card registration and information interfaces
- PC Card configuration interfaces
- System resource interfaces

PC Card Registration and Information Interfaces

Table 3-1 lists the interfaces available for registering with Card Services and retrieving information from Card Services.

Table 3-1 PC Card Registration and Information Interfaces

Functional Category	Function Name	Description
Card Services registration	`csx_RegisterClient`(9F)	Registers a client
	`csx_DeregisterClient`(9F)	Removes a client from Card Services list
Card Services information	`csx_GetStatus`(9F)	Returns the current status of the PC Card
	`csx_MapLogSocket`(9F)	Returns the physical socket number associated with the client handle

PC Card Configuration Interfaces

Table 3-2 lists the interfaces that a driver can use for PC Card configuration.

Table 3-2 PC Card Configuration Interfaces

Functional Category	Function Name	Description
PC Card configuration information	`csx_RequestConfiguration`(9F)	Configures the PC Card and socket
	`csx_ModifyConfiguration`(9F)	Modifies the socket and PC Card Configuration Register
	`csx_ReleaseConfiguration`(9F)	Releases the PC Card and socket
	`csx_AccessConfigurationRegister` (9F)	Reads or writes a PC Card Configuration Register
CIS tuple information	See Chapter 4, “Card Information Structure (CIS)” for the CIS Tuple Function and Utility tables	

System Resource Interfaces

Table 3-3 lists the interfaces needed for resource allocation. These interfaces can be categorized as follows:

- I/O resources
- IRQ resources
- Memory resources
- Event resources
- Common access functions

For information on additional interfaces, see "Additional Card Services Functions Provided in the Solaris System" on page 21.

Table 3-3 System Resource Interfaces

Functional Category	Function Name	Description
I/O resources	`csx_RequestIO`(9F)	Requests I/O resources for the client
	`csx_ReleaseIO`(9F)	Releases I/O resources for the client
IRQ resources	`csx_RequestIRQ`(9F)	Requests IRQ resource
	`csx_ReleaseIRQ`(9F)	Releases IRQ resource
Memory resources	`csx_RequestWindow`(9F)	Requests window resources
	`csx_ReleaseWindow`(9F)	Releases window resources
	`csx_ModifyWindow`(9F)	Modifies window attributes
	`csx_MapMemPage`(9F)	Maps the memory area on a PC Card
Event resources	`csx_RequestSocketMask`(9F)	Starts event handling
	`csx_GetEventMask`(9F)	Returns the client event mask
	`csx_SetEventMask`(9F)	Sets the client event mask
	`csx_ReleaseSocketMask`(9F)	Clears the client event mask
Common access	`csx_Put8`(9F)	Writes 1 byte to device register
	`csx_Put16`(9F)	Writes 2 bytes to device register
	`csx_Put32`(9F)	Writes 4 bytes to device register
	`csx_Put64`(9F)	Writes 8 bytes to device register

Table 3-3 System Resource Interfaces *(Continued)*

Functional Category	Function Name	Description
	`csx_Get8`(9F) `csx_Get16`(9F) `csx_Get32`(9F) `csx_Get64`(9F)	Reads data from device register
	`csx_RepPut8`(9F) `csx_RepPut16`(9F) `csx_RepPut32`(9F) `csx_RepPut64`(9F)	Writes repetitively to device register
	`csx_RepGet8`(9F) `csx_RepGet16`(9F) `csx_RepGet32`(9F) `csx_RepGet64`(9F)	Reads repetitively from device register
	`csx_GetMappedAddr`(9F)	Returns mapped virtual address
	`csx_DupHandle`(9F)	Duplicates access handle
	`csx_FreeHandle`(9F)	Frees access handle

Additional Card Services Functions Provided in the Solaris System

Card Services provides the following utility functions in addition to the functions defined by the PC Card Standard.

`csx_ConvertSize`*(9F)*

This utility function converts device size units from *devsize* format to *bytes* and vice versa. See *"Tuple Utility Functions"* in Chapter 4, "Card Information Structure (CIS)" for more information.

`csx_ConvertSpeed`*(9F)*

This utility function converts device speed units from *devspeed* format to *nanoseconds* and vice versa. See *"Tuple Utility Functions"* in Chapter 4, "Card Information Structure (CIS)" for more information.

csx_CS_DDI_Info*(9F)*

This function returns the dev_info_t and instance number associated with a particular PC Card driver and socket number. Some PC Card drivers encode the physical socket number as part of the dev_t of device nodes that they create.

The xx_getinfo(9E) driver entry point is required for drivers that export cb_ops(9S) entry points, and the driver's xx_getinfo(9E) function is required to return either the dev_info_t pointer or the instance number associated with the passed dev_t argument. Card Services maintains a table of dev_info_t pointers and instance numbers that correspond to particular client/socket number combinations; csx_CS_DDI_Info(9F) is the mechanism by which a PC Card driver can retrieve this information.

csx_Event2Text*(9F)*

This function returns text strings corresponding to an event. It is normally used for printing error messages.

csx_Error2Text*(9F)*

This function returns text strings corresponding to a Card Services error return code. It is normally used for printing error messages.

csx_MakeDeviceNode*(9F)*

This function creates minor nodes on behalf of PC Card drivers.

csx_RemoveDeviceNode*(9F)*

This function removes minor nodes on behalf of PC Card drivers.

Card Information Structure (CIS) 4

This chapter describes the Card Information Structure (CIS) and lists the Card Services tuple parsing functions.

Card Information Structure Overview

The Card Information Structure (CIS) is a data structure accessed through Card Services that contains identification and configuration information about PC Cards. The CIS includes information such as device speed, data size, and system resources required for PC Card operation. A PC Card driver accesses the CIS during initialization to determine the configuration options supported by a PC Card.

The CIS is a singly linked list of variable-length data blocks (called *tuples)* that describe the function and characteristics of a PC Card. Each tuple has a one-byte code describing the tuple type, and a one-byte link that is the offset to the start of the next tuple in the list. Each tuple can contain subtuples that elaborate on the information provided by the parent tuple.

PC Card drivers use the `csx_GetFirstTuple`(9F), `csx_GetNextTuple`(9F), and `csx_GetTupleData`(9F) functions to determine the exact configuration requirements of a PC Card. Drivers can then verify through Card Services the availability of the resources needed by the PC Card. If Card Services is able to allocate these resources, PC Card drivers secure them one at a time until all resources are obtained. Once all card resources are obtained, the PC Card driver can configure the PC Card.

Self-Identifying PC Cards

The CIS provides the mechanism for making a PC Card self-identifying. A self-identifying card provides information to the framework identifying the driver that needs to be used with a particular device.

Because PC Cards are self-identifying, PC Card drivers do not require a `.conf` file. However, a driver may provide a `.conf` file to define driver-specific properties. See *Writing Device Drivers* for more information on self-identifying devices.

Note – Currently, the Open Boot PROM does not recognize PC Cards (or the CIS) and does not use FCode to interpret data for use with self-identifying devices. The Solaris kernel and Card Services, however, are able to use CIS information to establish PC Card resource requirements.

PC Card Attribute and Common Memory

PC Cards (whether memory card or I/O card) provide a type of memory on the card called *attribute memory*, in which CIS card information is stored and where configuration registers (if any) are mapped. This information is later accessed by the PCMCIA framework during configuration and evaluated against platform hardware to determine the most appropriate configuration options for the card.

To simplify card design and accommodate 8-bit host systems, the CIS is only placed into attribute memory space at even-byte addresses. The contents of odd-byte addresses of attribute memory space are not defined. Attribute memory is always 8-bit, and data is only meaningful on even-byte addresses.

PC Cards also provide a type of memory on the card called *common memory*. Common memory may be used to store configuration information or to map memory arrays for stored data. Common memory is either 8-bit or (by default) 16-bit, and the data is only addressible on even-byte addresses. Note that access to 16-bit common memory should be done on even addresses only.

Note – Both PC Memory Cards and PC I/O Cards use attribute and common memory. PC I/O Cards, however, have a third address range reserved exclusively for I/O space. For more information, see the PC Card Standard.

Metaformat

The PCMCIA specification refers to the combination of all the tuples used to describe a PC Card as a *metaformat*. Because of the Metaformat structure, the CIS can be read by any operating system.

There are four different types of Metaformat. Solaris CIS parsing functions account for the differences in information storage when reading the CIS information, and present the CIS information in a defined structure. However, for vendor-specific tuples, a PC Card driver may have to parse the CIS information and handle the organization of the data itself. See the PC Card Standard for information on parsing vendor-specific tuples.

16-Bit PC Card Metaformat in Attribute Memory Space

The CIS always begins at location zero of attribute memory space and is stored only in even bytes. Odd-byte data stored in the attribute memory space is not defined.

16-bit PC Card Metaformat in Common Memory Space

The CIS information always begin at location zero of attribute memory and can be stored only in even bytes. The common memory CIS can be stored immediately following the attribute memory CIS. The CIS attribute information can be stored in both even and odd bytes in common memory space; however, data can only be accessed on even-byte addresses.

16-bit PC Card Metaformat for Multiple Function Cards

The multiple function link tuple, CISTPL_LONGLINK_MFC, must be included in the first or global CIS on a 16-bit PC Card to identify the card as containing multiple functions. This tuple does not apply to and must never be used by a CardBus PC Card. The 16-bit PC Card is also required to contain multiple CIS structures for each set of configuration registers on the card.

CardBus PC Card Metaformat

Tuple chains may be located in any of the card space with the exception of I/O space. Tuple chains can be located in the configuration space, memory space, or expansion ROM. Multiple function CardBus PC Cards have an independent configuration space and CIS for each function.

4

Tuple Parsing

A *tuple* provides information to the PCMCIA framework and PC Card drivers about configuration options available for a PC Card. This information is presented in a raw format; Solaris Card Services provides tuple parsing functions that allow drivers to interpret the raw tuple information into card configuration options.

Each tuple contains a unique identifier that describes the type of information contained in the tuple, the number of bytes contained in the tuple, and a data area that contains the tuple information. Tuples can be defined in relation to three PC Card layers:

- The *basic compatibility layer* defines the tuples necessary for PC Cards to be compatible on all host systems.
- The *data recording format layer* defines the tuple format of PC Memory Cards. Memory cards may be either block addressable or byte addressable.
- The *data organization layer* defines the organization of a particular partition on a PC Memory Card. For example, a tuple could specify whether the partition contains a file system, application-specific information, or executable code images.

Solaris Card Services CIS Parser

The Solaris Card Services framework provides a set of CIS structure tuple parsers that parse tuples into structures defined in the Solaris PCMCIA header files. The CIS parser performs the following functions:

- Creates a tuple list in memory to handle PC Cards on which the CIS is nonaccessible once the card is configured
- Prevents duplication of tuple parsing code
- Provides tuple parsing functions for most tuples described in the PC Card 95 Standard Metaformat specification

A client device driver need not include any tuple parsing code unless information from a nonstandard tuple is required. For nonstandard tuples, the device driver is required to handle tuple parsing itself using Card Services functions that return the raw contents of a tuple.

The Solaris PCMCIA framework depends on accurate, complete CIS information to bind drivers to PC Cards. At a minimum, the CIS must follow the PCMCIA 2.1 CIS/Metaformat specification for single-function cards and must follow the PC Card 95 Standard for multifunction cards.

The PC Card Standard requires that I/O cards have at least one configuration tuple to specify where the configuration registers are. PC Cards with no CIS (or with a CIS but no configuration tuple) are treated by default as memory cards.

Note – The Solaris 2.6 PCMCIA framework only supports multifunction cards that conform to the PC Card 95 Standard specification.

Tuple Parsing Functions

A PC Card driver can use the tuple parsing functions listed in Table 4-1 to parse CIS tuple entries into a usable form.

Table 4-1 Tuple Parsing Functions

Tuple Parsing Function	Description
`csx_Parse_CISTPL_DEVICE`(9F)	Device information
`csx_Parse_CISTPL_CHECKSUM`(9F)	Checksum control
`csx_Parse_CISTPL_LONGLINK_A`(9F)	Long-link to attribute memory
`csx_Parse_CISTPL_LONGLINK_C`(9F)	Long-link to common memory
`csx_Parse_CISTPL_LONGLINK_MFC`(9F)	Multifunction
`csx_Parse_CISTPL_LINKTARGET`(9F)	Link target control
`csx_Parse_CISTPL_NO_LINK`(9F)	No link control
`csx_Parse_CISTPL_VERS_1`(9F)	Level 1 version information
`csx_Parse_CISTPL_ALTSTR`(9F)	Alternate language string
`csx_Parse_CISTPL_DEVICE_A`(9F)	Attribute memory device information
`csx_Parse_CISTPL_JEDEC_C`(9F)	JEDEC programming information for common memory
`csx_Parse_CISTPL_JEDEC_A`(9F)	JEDEC programming information for attribute memory
`csx_Parse_CISTPL_CONFIG`(9F)	Configuration

Table 4-1 Tuple Parsing Functions *(Continued)*

Tuple Parsing Function	Description
csx_Parse_CISTPL_CFTABLE_ENTRY(9F	Configuration table entry
csx_Parse_CISTPL_DEVICE_OC(9F)	Device information under a set of operating conditions for common memory
csx_Parse_CISTPL_DEVICE_OA(9F)	Device information under a set of operating conditions for attribute memory
csx_Parse_CISTPL_MANFID(9F)	Manufacturer identification
csx_Parse_CISTPL_FUNCID(9F)	Function identification
csx_Parse_CISTPL_FUNCE(9F)	Function extension
csx_Parse_CISTPL_SWIL(9F)	Software interleave
csx_Parse_CISTPL_VERS_2(9F)	Level 2 version information
csx_Parse_CISTPL_FORMAT(9F)	Format type
csx_Parse_CISTPL_GEOMETRY(9F)	Geometry
csx_Parse_CISTPL_BYTEORDER(9F)	Byte order
csx_Parse_CISTPL_DATE(9F)	Card initialization date
csx_Parse_CISTPL_BATTERY(9F)	Battery replacement date
csx_Parse_CISTPL_ORG(9F)	Organization
csx_Parse_CISTPL_SPCL(9F)	Special purpose
csx_ParseTuple(9F)	Generic tuple parser entry point

No tuple parsing routines are provided for the following tuples:

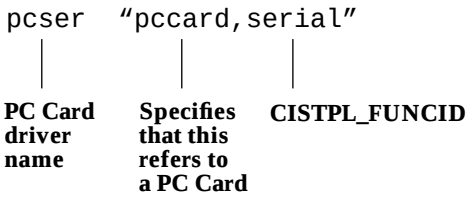
- CISTPL_NULL
- Any vendor-specific tuples (tuple codes in the range of 0x80 to 0x8f)
- CISTPL_END

Example Driver Alias Construction

The following sections provide examples of how raw tuple data is parsed into meaningful configuration information so that the system can bind a particular PC Card device driver to a particular PC Card, PC Card function, or to a class of PC Cards. The driver-device bindings are typically added to the `/etc/driver_aliases` file by the `add_drv`(1M) command when the driver is first installed in the system, and they are referred to as *driver aliases.*

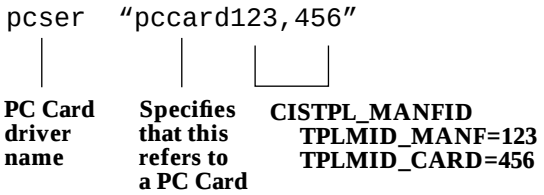
Using the CISTPL_FUNCID Tuple (Generic Names)

In this example, any PC Card that has a CISTPL_FUNCID tuple of type `serial` will cause the `pcser` driver to be bound to that card.



Using the CISTPL_MANFID Tuple

In this example, any PC Card that has a CISTPL_MANFID tuple that contains TPLMID_MANF = 123 and TPLMID_CARD = 456 will cause the `pcser` driver to be bound to that card. The TPLMID_MANF and TPLMID_CARD values are specified in hexadecimal. Note that there is *no* comma between `pccard` and the value for TPLMID_MANF.



Using the CISTPL_VERS_1 Tuple

In this example, any PC Card that has a CISTPL_VERS_1 tuple that contains the string(s) `Intel_MODEM_2400+_iNC110US` will cause the `pcser` driver to be bound to that card. Spaces in the PC Card's CISTPL_VERS_1 string must be replaced by underscores "_".

```
pcser  "pccard,Intel_MODEM_2400+_iNC110US"
```

PC Card driver name — **Specifies that this refers to a PC Card** — **CISTPL_VERS_1**

Matching is done using the string specified in the alias. The shorter of the alias string and the CISTPL_VERS_1 strings determines how many characters in the strings are matched against.

For example, the following alias:

```
pcser "pccard,Intel_MODEM_2400+"
```

would match PC Cards that contain the string `"Intel_MODEM_2400+"` in their CISTPL_VERS_1 tuple.

The following alias:

```
pcser "pccard,Intel_MODEM"
```

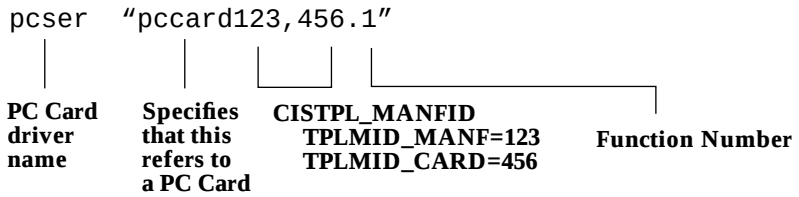
would match PC Cards that contain the string `"Intel_MODEM"` in their CISTPL_VERS_1 tuple, which would include PC Cards that contain one of the following strings in their CISTPL_VERS_1 tuple:

```
"Intel_MODEM"
"Intel_MODEM_2400+"
"Intel_MODEM_2400+_iNC110US"
```

All strings in the CISTPL_VERS_1 are concatentated into one string for matching purposes. All trailing spaces in the string are removed before matching.

Using the CISTPL_MANFID Tuple for Multifunction Cards

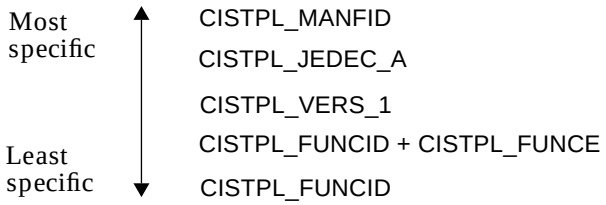
In this example, any PC Card *function* that has a CISTPL_MANFID tuple that contains TPLMID_MANF = 123 and TPLMID_CARD = 456 and is function number 1 will cause the `pcser` driver to be bound to that function on that card. The TPLMID_MANF, TPLMID_CARD, and function number values are specified in decimal. Note that there is *no* comma between `pccard` and the value for TPLMID_MANF.



This form of alias is only usable for multifunction PC Cards that comply with the PC Card 95 Multifunction CIS specification. Older PC Cards that use vendor-specific multifunction CIS layouts will appear to the Solaris system as a single-function card and will very likely require a custom device driver.

Driver Alias Matching

The general rule for determining which one of several driver aliases will match a particular PC Card and PC Card function is that the more specific alias will always take precedence over the less-specific alias. The order of matching, from most specific to least specific, is:



For example, given the following driver aliases:

```
pcxxx "pccard,serial"
pcyyy "pccard,Intel_MODEM_2400+_iNC110US"
pczzz "pccard123,456"
```

if a PC Card with the following tuples is inserted:

```
CISTPL_FUNCID = "serial"
CISTPL_VERS_1 = "Acme Serial Card"
CISTPL_MANFID = "654,321"
```

the `pcxxx` driver would be bound to that card, since neither the CISTPL_VERS_1 nor the CISTPL_MANFID tuples match any of the aliases.

If a PC Card with the following tuples is inserted:

```
CISTPL_FUNCID = "serial"
CISTPL_MANFID = "654,321"
CISTPL_VERS_1 = "Intel_MODEM_2400+"
```

the `pcyyy` driver would be bound to that card, since the CISTPL_VERS_1 tuple (partially) matches that driver alias. The CISTPL_FUNCID tuple also matches the `pccard,serial` alias; however, the CISTPL_VERS_1 match is more specific than the CISTPL_FUNCID match.

If a PC Card with the following tuples is inserted:

```
CISTPL_FUNCID = "serial"
CISTPL_MANFID = "123,456"
CISTPL_VERS_1 = "Intel_MODEM_2400+"
```

the `pczzz` driver would be bound to that card, since the CISTPL_MANFID matches the `123,456` alias. Although the CISTPL_VERS_1 tuple matches the string in the `pccard,Intel_MODEM_2400+_iNC110US` alias, the CISTPL_MANFID match is performed before the CISTPL_VERS_1 match.

If a PC Card has no CIS or a malformed CIS, the driver that matches the `pccard,memory` alias will get bound to that card. This is to handle the case of older PCMCIA memory cards that do not have a CIS. Use of PC Cards without a proper CIS is discouraged.

General Rules for PC Card-Driver Bindings

PC Card devices use a *generic* name whenever it can be determined. Generic names describe the function of a device without identifying the specified driver for the device. Devices with a Function ID tuple will always be in generic form with the following recognized names:

- memory
- network
  - ethernet (if the sys extension tuple `funce` is present)
  - token-ring (if the sys extension tuple `funce` is present)
  - localtalk
  - fddi
  - atm
  - wireless
  - reserved
- disk (ata)
- parallel
- video
- aims
- scsi
- serial
- security

Devices with generic device names must support a *compatible* property. The *compatible* property contains a list of one or more possible driver names for the device; it is initialized with a set of aliases derived from the Card Information Structure. PC Card driver-to-device binding is done by the Solaris framework from aliases listed in the *compatible* property. The naming rules are:

- If no CIS is present, then *compatible* is `pccard,memory` or `memory`.
- If Version 1 strings are present, an alias is constructed of the form: `pccard,<version-1-tuple>`. `<version-1-tuple>` is the Version 1 tuple string with all spaces replaced with '_' and using ',' as the separator between strings. The resulting alias is appended to the *compatible* property.
- If a Manufacturer tuple is present, `pccardVVVV,DDDD`, where VVVV is the Vendor ID in hexadecimal (lowercase, no leading 0s) and DDDD is the device information in hexadecimal (lowercase, no leading 0s), is appended to the *compatible* property.
- If the PC Card is a multifunction card, another form of the Manufacturer tuple alias is constructed for each function. The format for each alias is `pccardVVVV,DDDD.F` where F is the function number. These aliases (if present) appear in the *compatible* property list before the Manufacturer tuple alias.

- If a Function ID tuple is present, the function ID tuple specifies the generic name. The generic name is used to construct two aliases: the first is `pccard,<generic-name>` and the second is `<generic-name>`.
- If no generic name is found, then the device node is named according to the `pccardVVVV,DDDD` format if possible; otherwise, the name `pccard` is used.

For cards such as modem/serial cards, ATA cards, and memory cards, the naming string follows the definitions listed in Table 4-2.

Table 4-2 PC Card Generic Device Names

Card Type	Generic Device Name
Memory nexus	`memory`
SRAM memory	`pcram`
Modem/Serial	`serial`
3COM EtherLinkIII	`network`
ATA card	`disk`

Note – For more information on generic device names, refer to the *Writing Device Drivers* manual.

Tuple Utility Functions

The tuple utility functions listed in Table 4-3 enable CIS tuple entries to be converted from a Card Services internal representation into common units usable by PC Card drivers.

Table 4-3 Tuple Utility Functions

Tuple Utility Function	Description
`csx_ConvertSize`(9F)	Converts device size units from Card Services internal representation to bytes and vice versa.
`csx_ConvertSpeed`(9F)	Converts device speed units from Card Services internal representation to nanoseconds and vice versa.

Code Example 4-1 demonstrates the use of `csx_ConvertSpeed`(9F).

Code Example 4-1 Use of `csx_ConvertSpeed`(9F)

```
pcsram_state_t *rs;
int             ret;
win_req_t       win_req;
convert_speed_t convert_speed;

/ * Get a memory window for Common Memory space */
win_req.Attributes = (WIN_MEMORY_TYPE_CM |
                      WIN_DATA_WIDTH_16 |
                      WIN_ENABLE |
                      WIN_ACC_NEVER_SWAP |
                      WIN_ACC_STRICT_ORDER);

win_req.Base.base = 0;/*let CardService find us a base addr */
win_req.Size = 0;     /* let CardService return the smallest */
                      /* size window it finds */

/* Use csx_ConvertSpeed(9F) function to generate the */
/* appropriate AccessSpeed code unit. */

convert_speed.Attributes = CONVERT_NS_TO_DEVSPEED;
convert_speed.nS = 250;

csx_ConvertSpeed(&convert_speed);
win_req.win_params.AccessSpeed = convert_speed.devspeed;
if ((ret = csx_RequestWindow(rs->client_handle,
               &rs->window_handle, &win_req)) != CS_SUCCESS) {
       error2text_tcft;
       cft.item = ret;
       csx_Error2Text(&cft);
       cmn_err(CE_CONT, "pcsram: RequestWindow failed %s
                  (0x%x)\n", cft.text, ret);
       return (ret);
};
```

4

PC Card Driver Autoconfiguration 5

This chapter describes the support a PC Card driver must provide for driver autoconfiguration.

PC Card Driver Autoconfiguration Overview

In the autoconfiguration process, an individual instance of a PC Card driver is attached to a function on a particular PC Card in a given socket. On PC cards providing multiple functions, a unique instance of a PC Card driver is attached to each function. In the Solaris system, an instance of a PC Card driver can only access one function.

The basic steps in the autoconfiguration process are the same for PC Card drivers as for other drivers in the Solaris system; however, some details differ for PC Card drivers. For example, `attach`(9E) initializes the instance of the driver for a PC Card, but PC Card configuration itself does not occur until Card Services notifies the driver that the card is ready. The separation of driver autoconfiguration and card configuration allows PC Cards to be dynamically configured and unloaded as they are inserted and removed from the system.

PC Card drivers must provide the following standard set of device loading and autoconfiguration entry points and data structures.

- The `_init`(9E), `_info`(9E), and `_fini`(9E) routines are required for module loading. The loadable module routines used by non-PCMCIA drivers can be used by PC Card drivers without modification.

- The autoconfiguration entry points `attach`(9E), `getinfo`(9E), and `detach`(9E) are required.
- The loadable module and driver initialization data structures `modlinkage`(9S), `modldrv`(9S), `dev_ops`(9S), and `cb_ops`(9S) are required.

The sections that follow describe autoconfiguration information specific to writing a PC Card driver. For general information on autoconfiguration in the Solaris system, see the *Writing Device Drivers* manual. See Chapter 9, "PCMCIA Parallel Port Driver," in this manual for a comprehensive look at the autoconfiguration routines of the `pcepp.c` sample driver.

Autoconfiguration Entry Points

Each PC Card driver must provide the following autoconfiguration entry points:

- `attach`(9E)
- `getinfo`(9E)
- `detach`(9E)

`attach`*(9E)*

A driver's `attach`(9E) entry point initializes an instance of the driver. A PC Card `attach`(9E) routine should perform the common initialization tasks that all drivers require and should handle additional tasks specific to Card Services, such as driver registration and event handling setup. Typically, a PC Card driver `attach`(9E) routine does the following:

- Allocates the soft state structures for the driver instance
- Registers the driver instance with Card Services
- Allocates any memory needed for this instance of the driver
- Initializes per-instance mutexes and condition variables
- Installs a Card Services event handler
- Gets the logical socket number associated with the client handle
- Enables the event handler

Note that system resource allocation, such as mapping the device's registers or registering device interrupts, which is typically done at attach time, is not done by a PC Card driver until the driver receives an insertion event. For information on resource allocation, see Chapter 7, "PC Card Configuration."

Driver Registration With Card Services

A driver registers with Card Services by calling csx_RegisterClient(9F).

```
int32_t csx_RegisterClient(client_handle_t *, client_reg_t *);
```

When calling csx_RegisterClient(9F), the PC Card driver must provide the following information in a client_reg_t(9S) structure:

- Type of client
- Event types
- Event handler
- Event callback data
- Card Services version number
- Device instance number (driver dip)
- Driver name

The client_reg_t(9S) structure is defined in the cs.h header file as:

```
typedef struct client_reg_t {
    uint32_t                Attributes;
    uint32_t                EventMask;
    event_callback_args_t   event_callback_args;
    uint32_t                Version;        /* CS version */
    csfunction_t            *event_handler;
    ddi_iblock_cookie_t     *iblk_cookie;  /* event iblk cookie */
    ddi_idevice_cookie_t    *idev_cookie;  /* event idev cookie */
    dev_info_t              *dip;          /* client dip */
    char                    driver_name[MODMAXNAMELEN];
} client_reg_t;
```

Client Types

When calling csx_RegisterClient(9F), the PC Card driver must specify the type of client to be registered. Two categories of PC Card drivers are defined:

- PC Card I/O drivers
- PC Card memory drivers

A PC Card driver specifies whether it is a memory or I/O client driver by setting the Attributes field of the csx_RegisterClient(9F) request to INFO_IO_CLIENT or INFO_MEM_CLIENT. A client type must be specified.

The Card Services client registration process uses the driver category to determine the order in which events are sent to the drivers. The PC Card Standard specifies that PC Card I/O drivers must receive events before PC Card Memory drivers. However, no order is defined for events sent to multiple instances of the same driver. Aside from event sequencing, there are no other functional differences between the driver categories.

Event Types

The PC Card driver specifies the types of events for which it is to receive notification using the `EventMask` field in the `client_reg_t` structure. For a list of valid event types, see "Event Types" on page 50.

Event Handler and Callback Data

Each instance of a PC Card driver must register an event handler to manage events associated with the PC Card. The driver event handler is registered using the `event_handler` field of the `client_req_t structure`.

The driver may also provide client data that is passed to its event handler function; this is done using the `event_callback_arg.client_data` field. Typically, this argument is the driver instance's soft-state pointer.

Card Services Version Number

The Version field contains the Card Services version number that the client driver expects to use. Typically, the driver will use the `CS_VERSION` macro to specify to Card Services which version of Card Services the client expects. The `CS_VERSION` macro is defined in the `cs.h` header file.

Return Values

`csx_RegisterClient`(9F) returns a unique client handle that the driver must use to make future Card Services requests. This identifies the instance of the driver.

`csx_RegisterClient`(9F) also returns a high-priority interrupt cookie that must be used to create a mutex to protect any data shared between the PC Card event handler (when handling a card removal event) and the rest of the driver. The driver uses the high-priority interrupt block cookie to initialize the high-level mutex lock that is used in the event handler entry point. For more information, see `csx_event_handler`(9E).

Note – PC Card drivers must register with Card Services. A PC Card driver can determine whether Card Services has been installed on the host machine by calling `csx_GetCardServicesInfo`(9F) or indirectly by calling `csx_RegisterClient`(9F) from `attach`(9E). If Card Services is not available when `csx_GetCardServicesInfo`(9F) is called, an error is returned.

Enabling Event Notification

In order for the event handler to start receiving events, the PC Card driver must call `csx_RequestSocketMask`(9F). Although `csx_RegisterClient`(9F) registers the driver's event handler, no events are delivered to the driver until after a call to `csx_RequestSocketMask`(9F) has been successful.

`csx_RequestSocketMask`(9F) requests that the client be notified of status changes for this socket. When calling `csx_RequestSocketMask`(9F), the driver provides an event mask in the `sockevent_t` structure to specify events that the PC Card driver is registering to receive for the socket. The socket event mask can be used to modify the global event mask registered with `csx_RegisterClient`(9F).

Note that for Solaris PCMCIA, there is only one PC Card driver instance per socket, and all events occur on a per-socket basis. Each Solaris PC Card driver instance gets called only for its particular socket and instance, and the driver only has access to that instance's data. Similarly, a Solaris PC Card driver only receives notification of events for its specific socket and instance; it is unaware of other PC Card drivers in the system.

Once `csx_RequestSocketMask`(9F) has been called, Card Services delivers a CS_EVENT_REGISTRATION_COMPLETE event if the PC Card driver has specified in its event mask that it is to receive this event.

Note – Note that PC Cards that use interrupts normally would not have to install an interrupt handler at attach time. Typically, a PC Card driver would install a high-level interrupt handler and a software interrupt handler when the card is inserted and becomes ready. The soft interrupt handler and high-level interrupt handler would then be removed when the card is removed to free these resources for use by other cards. For more information on installing an interrupt handler, see "Installing an Interrupt Handler" on page 68.

Example attach*(9E) Routine*

Code Example 5-1 shows a partial implementation of an attach(9E) routine. For a complete example of a PC Card attach routine, see "pcepp_attach()" in Chapter 9.

Code Example 5-1 PC Card attach(9E) Routine

```
static int
xx_attach(dev_info_t*dip, ddi_attach_cmd_tcmd)
{
   int                instance;
   int                ret;
   xx_state_t         *xx;
   get_status_t       get_status;
   client_reg_t       client_reg;
   sockmask_t         sockmask;
   map_log_socket_t   map_log_socket;

   instance = ddi_get_instance(dip);

   switch (cmd) {
   ...
       case DDI_ATTACH:
           break;
   ...
   }

   /* Allocate per-instance soft state */
   if (ddi_soft_state_zalloc(xx_soft_state_p,
                  instance) != DDI_SUCCESS) {
       return (DDI_FAILURE);
   }

   xx = ddi_get_soft_state(xx_soft_state_p, instance);

   /* Remember dev_info structure for xx_getinfo  */
   xx->dip = dip;
   xx->instance = instance;
   ddi_set_driver_private(dip, (caddr_t)xx);
   ...

   /* Register with Card Services */
   client_reg.Attributes = INFO_IO_CLIENT |
                      INFO_CARD_SHARE | INFO_CARD_EXCL;
```

```
client_reg.EventMask = (CS_EVENT_REGISTRATION_COMPLETE |
               CS_EVENT_CARD_READY |
               CS_EVENT_CARD_INSERTION |
               CS_EVENT_CARD_REMOVAL |
               CS_EVENT_CARD_REMOVAL_LOWP |
               CS_EVENT_CLIENT_INFO);
client_reg.event_handler = (csfunction_t *)xx_event;
client_reg.event_callback_args.client_data = xx;
client_reg.Version = CS_VERSION;
client_reg.dip = dip;
(void) strcpy(client_reg.driver_name, XX_NAME);

ret = csx_RegisterClient(&xx->client_handle, &client_reg);

/* Get logical socket number and store in xx_state_t */
ret = csx_MapLogSocket(xx->client_handle, &map_log_socket);

xx->sn = map_log_socket.PhySocket;
...

/* Initialize the event handler mutexes and cv */
mutex_init(&xx->event_hi_mutex, "xx->event_hi_mutex",
       MUTEX_DRIVER, *(client_reg.iblk_cookie));

mutex_init(&xx->event_mutex, "xx->event_mutex",
       MUTEX_DRIVER, NULL);

cv_init(&xx->readywait_cv, "xx->readywait_cv",
       CV_DRIVER, (void *)NULL);

/* After RequestSocketMask, start receiving events. */
mutex_enter(&xx->event_mutex);
sockmask.EventMask = (CS_EVENT_CARD_INSERTION |
           CS_EVENT_CARD_REMOVAL);

ret = csx_RequestSocketMask(xx->client_handle,&sockmask);
...

/*
 * If the card is inserted and this attach is triggered by
 * an open, the open will wait until card insertion is complete
 */
 xx->card_event |= XX_CARD_WAIT_READY;
 while ((pps->card_event &
    (XX_CARD_READY | XX_CARD_ERROR)) == 0) {
    cv_wait(&xx->readywait_cv, &xx->event_mutex);
```

```
    }
    ...

    mutex_exit(&xx->event_mutex);
    ddi_report_dev(dip);
    return (DDI_SUCCESS);
    ...
}
```

getinfo*(9E)*

The getinfo(9E) entry point in a PC Card driver functions as in any other Solaris device driver with the exception of the use (and interpretation) of a driver instance number.

Since most host systems have at least two PCMCIA sockets, the driver should encode the socket number—rather than the instance number—as the minor number of the device. This allows easier identification of a PC Card with its appropriate socket. The CS_DDI_Info(9F) function can be used to retrieve the instance number, which identifies the appropriate socket number for the PC Card.

Code Example 5-2 shows the getinfo(9E) entry point. In this code example, XX_SOCKET is a driver-specific macro that retrieves the socket number from arg.

Code Example 5-2 getinfo(9E) Routine

```
static int
xx_getinfo(dev_info_t*dip, ddi_info_cmd_tcmd, void *arg,
               void **result)
{
    int             rval = DDI_SUCCESS;
    xx_state_t      *xx;
    cs_ddi_info_t   cs_ddi_info;

    switch (cmd) {

    case DDI_INFO_DEVT2DEVINFO:
        cs_ddi_info.Socket = XX_SOCKET((dev_t)arg);
        cs_ddi_info.driver_name = xx_name;
        if (csx_CS_DDI_Info(&cs_ddi_info) != CS_SUCCESS) {
            return (DDI_FAILURE);
        }
```

```
        if (!(xx = ddi_get_soft_state(xx_soft_state_p,
                   cs_ddi_info.instance))) {
            *result = NULL;
        } else {
            *result = xx->dip;
        }
        break;

    case DDI_INFO_DEVT2INSTANCE:
        cs_ddi_info.Socket = XX_SOCKET((dev_t)arg);
        cs_ddi_info.driver_name = xx_name;
        if (csx_CS_DDI_Info(&cs_ddi_info) != CS_SUCCESS) {
            return (DDI_FAILURE);
        }
        *result = (void *)cs_ddi_info.instance;
        break;

    default:
        rval = DDI_FAILURE;
        break;
    }
    return (rval);

}
```

detach*(9E)*

The detach(9E) entry point removes a PC Card from the system. detach(9E) releases resources allocated with attach(9E) or with card insertion, releases the driver socket mask, deregisters with Card Services, and frees the various mutex and condition variables. Code Example 5-3 shows an example of a detach(9E) entry point.

Code Example 5-3 detach(9E) Routine

```
static int
xx_detach(dev_info_t*dip, ddi_detach_cmd_tcmd)
{
    int                     instance;
    int                     ret;
    xx_state_t              *xx;
    release_socket_mask_trsm;
    error2text_t            cft;

    instance = ddi_get_instance(dip);
```

```
    switch (cmd) {
    case DDI_SUSPEND:
        /*
         * DDI_SUSPEND/DDI_RESUME should be implemented
         *  to always succeed.
         */
        return (DDI_SUCCESS);

    case DDI_DETACH:
        break;

    default:
        return (DDI_FAILURE);
    }

    xx = ddi_get_soft_state(xx_soft_state_p, instance);
    if (xx == NULL) {
        cmn_err(CE_NOTE, "xx%d: no soft state\n", instance);
        return (DDI_FAILURE);
    }

    /* Call xx_card_removal to do any final card cleanup */
    if (xx->card_event & XX_CARD_READY) {
        mutex_enter(&xx->event_mutex);
        (void) xx_card_removal(xx);
        mutex_exit(&xx->event_mutex);
    }

    /* Release driver socket mask */
    ret = csx_ReleaseSocketMask(xx->client_handle,&rsm);

    /* Deregister with Card Services to stop getting events.*/
    ret = csx_DeregisterClient(xx->client_handle);

    /* Free the various mutex and condition variables */
    mutex_destroy(&xx->event_hi_mutex);
    mutex_destroy(&xx->event_mutex);
    cv_destroy(&xx->readywait_cv);

    ddi_soft_state_free(xx_soft_state_p, instance);
    return (DDI_SUCCESS);
}
```

PC Card Power Management Suspend and Resume

Card Services manages PC Card driver suspend and resume requests in a manner consistent with the PC Card standard. When power is suspended, the driver receives a suspend event and card removal event. The driver should handle the suspend as a card removal event because the driver has no way of knowing when the system is powered off whether the PC Card has been removed. Similarly, when power is restored, if the PC Card is present when this occurs, the driver will receive a resume event followed by a card insertion event. The driver should handle the resume event as a card insertion, as the driver will not know whether the card was removed while power was suspended.

A PC Card driver should be written to always return `DDI_FAILURE` in response to a `DDI_SUSPEND` or `DDI_RESUME` command, as shown in Code Example 5-4. Because PC Card suspend and resume events are handled as card removal and card insertion events, they do not need to be implemented in `attach`(9E) or `detach`(9E).

For more information on power management in the Solaris system, see the *Writing Device Drivers* manual.

Code Example 5-4 Power Management Suspend and Resume

```
static int
xx_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
   switch (cmd) {
   case DDI_RESUME:
       return (DDI_FAILURE);
   ...
   }
}

static int
xx_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
   switch (cmd) {
   case DDI_SUSPEND:
       return (DDI_FAILURE);
   ...
   }
}
```

5

PC Card Event Management 6

This chapter discusses event handling in the PCMCIA framework and provides code examples for several events.

PC Card Event Management Overview

PCMCIA event management provides the mechanism that informs PC Card drivers of hardware and software status changes. PCMCIA *events* are generated by physical state changes or by changes in the software framework. Software events occur directly as a result of changes in the software state of the PCMCIA framework or indirectly due to a change in hardware state of the PCMCIA framework or underlying host system.

For example, inserting a PC Card into a socket causes Card Services to deliver a card insertion event to the PC Card driver. A battery low event informs the driver when the battery is beginning to fail. PCMCIA events also notify drivers of the completion of an asynchronous task; for example, a card ready event informs the driver that the card has completed its internal initialization and is ready for operation.

To respond to events, the PC Card client must supply an event handler routine and specify the events that it is interested in. Once the event handler is registered with Card Services, the event handler drives the completion of the card configuration process as well as driver responses to other events.

Event Types

Card Services defines a set of event types that the PC Card driver uses to register interest in certain events. A driver registers interest in events by specifying an event bit mask in the driver registration call `csx_RegisterClient`(9F). The event mask can be modified using the `csx_RequestSocketMask`(9F) function or the `csx_GetEventMask`(9F) and `csx_SetEventMask`(9F) functions.

Table 6-1 lists the possible event types.

Table 6-1 Event Types

Event Types	Description
CS_EVENT_BATTERY_LOW	The PC Card battery charge is weak and needs to be replaced.
CS_EVENT_BATTERY_DEAD	The PC Card battery is no longer providing operational voltage.
CS_EVENT_CARD_INSERTION	A PC Card has been inserted in a socket.
CS_EVENT_CARD_READY	A PC Card's READY line has changed from the busy to ready state.
CS_EVENT_CARD_REMOVAL	A PC Card has been removed from a socket.
CS_EVENT_CARD_RESET	A hardware reset has occurred.
CS_EVENT_CARD_LOCK	A mechanical latch has been manipulated, preventing the removal of the PC Card from the socket.
CS_EVENT_CARD_UNLOCK	A mechanical latch has been manipulated, enabling the removal of the PC Card from the socket.
CS_EVENT_CLIENT_INFO	A request that the client return its client information data.
CS_EVENT_EJECTION_REQUEST	A request that the PC Card be ejected from a socket using a motor-driven mechanism.
CS_EVENT_EJECTION_COMPLETE	A motor has ejected a PC Card from a socket.
CS_EVENT_ERASE_COMPLETE	A queued erase request that is processed in the background is complete.

Table 6-1 Event Types *(Continued)*

Event Types	Description
CS_EVENT_INSERTION_REQUEST	A request that a PC Card be inserted into a socket using a motor-driven mechanism.
CS_EVENT_INSERTION_COMPLETE	A motor has inserted a PC Card in a socket.
CS_EVENT_RESET_REQUEST	A request for a physical reset by a client.
CS_EVENT_REGISTRATION_COMPLETE	Client registration has completed.
CS_EVENT_RESET_COMPLETE	A reset request that is processed in the background has been completed.
CS_EVENT_RESET_PHYSICAL	A reset is about to occur.
CS_EVENT_WRITE_PROTECT	The write-protect status of the PC Card in the indicated socket has changed.

Note – In all cases, Card Services delivers an event to each driver instance associated with a function on a multiple-function PC Card.

Event Priorities

Events are dispatched to the PC Card driver as either CS_EVENT_PRI_HIGH for high-priority events or CS_EVENT_PRI_LOW for low-priority events. Some events can be delivered at both high priority and low priority. Card Services specifies the priority of each event to the driver's event handler in an argument to the driver event handler. The driver uses the priority flag to determine which behavior to follow.

High-priority events are delivered above lock level (above the level of the system scheduler). The driver must use its high-level event mutex initialized with the `iblk_cookie` returned by `csx_RegisterClient`(9F) to protect such events.

Low-priority events are delivered below lock level (below the level of the system scheduler), and—unless the cookie has been returned by a call to `ddi_add_softintr`(9F)—the driver must use its low-level event mutex initialized with a NULL interrupt cookie to protect these events.

Event Handler Entry Point

Each instance of a PC Card driver must register an event handler to manage events associated with its PC Card. The driver also specifies the events that it is interested in. When Card Services notices that an event has occurred, it notifies interested clients. Event notification takes place through the driver's `csx_event_handler`(9E) entry point.

The driver event handler is registered using the `csx_RegisterClient`(9F) function. This typically occurs at driver initialization in the `attach`(9E) function. The `csx_RegisterClient`(9F) `client_req_t` structure includes fields for the event handler and event mask. For more information on driver registration, see "Driver Registration With Card Services" on page 39.

Once `csx_RegisterClient`(9F) returns to the driver, the PC Card driver event handler must be capable of receiving events. Sometimes a card insertion event can occur without a PC Card actually being inserted. This is called an *artificial insertion event*. An artificial insertion event is generated by Card Services and not by the action of a card being inserted. Card Services delievers a CS_EVENT_REGISTRATION_COMPLETE event when registration is complete and all artificial events have been generated.

`csx_event_handler`*(9E)*

Card Services calls the driver event handler with the event type, the event priority, and optional client data. The event handler entry point is defined as:

```
int32_t event_handler(event_t event,int32_t priority,
                      event_callback_args_t *args);
```

The client data is specified in the driver registration call using the `event_callback_args.client_data` field in the `csx_RegisterClient`(9F) `client_req_t` structure. Typically, this argument is the driver instance's soft-state pointer.

Code Example 6-1 shows a PC Card driver event handler that handles a basic set of events, including card ready, card insertion, and card removal events. The driver ignores events that it has not registered interest in and returns a status code to Card Services.

Code Example 6-1 PC Card Event Handler

```
static int32_t
xx_event_handler(event_t event, int32_t priority,
                  event_callback_args_t *eca)
{
   xx_unit_t      *xx = eca->client_data;
   client_info_t  *ci = &eca->client_info;
   int32_t        ret = CS_UNSUPPORTED_EVENT;

   switch (event) {
       case CS_EVENT_REGISTRATION_COMPLETE:
           ret = CS_SUCCESS;
           break;
       case CS_EVENT_CARD_READY:
           if (priority & CS_EVENT_PRI_LOW) {
               mutex_enter(&xx->event_mutex);
               ret = xx_card_ready(xx);
               mutex_exit(&xx->event_mutex);
           }
           break;
       case CS_EVENT_CARD_INSERTION:
           if (priority & CS_EVENT_PRI_LOW) {
               mutex_enter(&xx->event_mutex);
               ret = xx_card_insertion(xx);
               mutex_exit(&xx->event_mutex);
           }
           break;
       case CS_EVENT_CARD_REMOVAL:
       case CS_EVENT_CARD_REMOVAL_LOWP:
           if (priority & CS_EVENT_PRI_HIGH) {
               mutex_enter(&xx->event_hi_mutex);
               cbt->card_state &= ~XX_CARD_INSERTED;
               mutex_exit(&xx->event_hi_mutex);
               rval = CS_SUCCESS;
           } else {
               mutex_enter(&xx->event_mutex);
               ret = xx_card_removal(xx);
               mutex_exit(&xx->event_mutex);
           }
           break;
       case CS_EVENT_CLIENT_INFO:
           mutex_enter(&xx->event_mutex);
           if (GET_CLIENT_INFO_SUBSVC(ci->Attributes) ==
                   CS_CLIENT_INFO_SUBSVC_CS) {
               /*
```

```
                * Note that CardServices will prepare the
                * DriverName field on the driver's behalf.
                */
                ci->Revision = XX_CI_REVISION;
                ci->CSLevel = CS_VERSION;
                ci->RevDate = XX_REV_DATE;
                strcpy(ci->ClientName, XX_CLIENT_DESCRIPTION);
                strcpy(ci->VendorName, XX_VENDOR_DESCRIPTION);
                ci->Attributes |= CS_CLIENT_INFO_VALID;
                ret = CS_SUCCESS;
            }
            mutex_exit(&xx->event_mutex);
            break;
    }
    return (ret);
}
```

Event Handling Examples

Typically, PC Card drivers are primarily concerned with card insertion, card ready, and card removal events. A driver may also want to set up a timeout routine to handle the delay that may occur between card insertion and card readiness. The following sections provide information and sample code for these events.

Card Insertion

When a PC Card is inserted, Card Services delivers a CS_EVENT_CARD_INSERTION event to the driver's event handler. Note that PC Card drivers only receive card insertion events for their specific cards and are not informed about other PC Cards in the system.

In Code Example 6-2, the driver determines whether the card is ready for further processing. If the card is ready, the driver calls the card ready routine, and card configuration continues. If the card is not ready, the driver sets a timeout and waits for the card to become ready.

Code Example 6-2 Card Insertion Routine

```
static int
xx_card_insertion(xx_state_t *xx)
{
    int             ret = CS_OUT_OF_RESOURCE;
```

```
    get_status_t   get_status;

    ASSERT(mutex_owned(&xx->event_mutex));

    xx->card_event &= ~(XX_CARD_READY | XX_CARD_ERROR);
    xx->card_event |= XX_CARD_INSERTED;

    ret = csx_GetStatus(xx->handle, &get_status);

    if (get_status.CardState & CS_EVENT_CARD_READY)
        (void) xx_card_ready(xx);

    if ((xx->card_event &
        (XX_CARD_READY | XX_CARD_ERROR)) == 0) {
        /* If the card isn't ready yet, we set up a timeout
         * handler to trap cards that never do give us a
         * Card Ready event,and then return to wait for either
         * the Card Ready event or the timeout to expire.
         */
        xx->ready_timeout_id =
            timeout(xx_card_ready_timeout, (caddr_t)xx,
                (long)(drv_usectohz(1000000) *
                XX_READY_TIMEOUT));
    }
    return (CS_SUCCESS);
}
```

Note – PC Card drivers do not need to parse CIS to determine if their card is inserted. However, some cards may need to parse CIS in order to determine configuration changes. For example, in a system where a fully-relocatable modem is inserted and later replaced by a different modem card, the framework will consider both cards the same, but the new card will require a different configuration.

Card Ready

When a PC Card becomes ready after insertion, Card Services delivers a CS_EVENT_CARD_READY event to the driver's event handler. This callback indicates that the driver may now access and initialize the card, as shown in Code Example 6-3. For information on card configuration, see Chapter 7, "PC Card Configuration."

Code Example 6-3 Card Ready Routine

```
static int
xx_card_ready(xx_state_t *xx)
{
   int                ret;
   modify_config_t    modify_config;

   ASSERT(mutex_owned(&xx->event_mutex));

   /* Remove any pending card ready timer */
   if (xx->ready_timeout_id) {
       int id = xx->ready_timeout_id;
       xx->ready_timeout_id = 0;
       mutex_exit(&xx->event_mutex);
       untimeout(id);
       mutex_enter(&xx->event_mutex);
   }

   /*
    * If  the card is just now becoming ready,
    * perform basic card configuration and initialization
    */
   if ((xx->card_event & XX_CARD_READY) == 0) {
       if (xx_card_configuration(xx) == 0) {
           xx->card_event |= XX_CARD_ERROR;
           if (xx->card_event & XX_CARD_WAIT_READY) {
               xx->card_event &= ~XX_CARD_WAIT_READY;
               ASSERT(mutex_owned(&xx->event_mutex));
               cv_broadcast(&xx->readywait_cv);
           }
           return (CS_SUCCESS);
       }
   }
   return (CS_SUCCESS);
}
```

Card-Ready Timeout

The PC Card Standard acknowledges that a PC Card may take a few seconds to become ready after insertion. Card Services delivers a CS_EVENT_CARD_INSERTION event when the card is inserted, but the card may not be ready to use at that time. When the card becomes ready to use, Card Services delivers a CS_EVENT_CARD_READY event.

For best results, the client driver should provide a timeout routine to handle the case when the card is not ready when inserted. This routine will enable the driver to detect a card that has not become ready within a reasonable period of time. When the timeout occurs, the driver can issue a signal on the `readywait_cv` to wake up the `attach`(9E) function. This will prevent `attach`(9E) from hanging forever. Code Example 6-4 shows a card-ready timeout routine.

Code Example 6-4 Card-Ready Timeout Routine

```
static void
xx_card_ready_timeout(xx_state_t *xx)
{
   mutex_enter(&xx->event_mutex);
   if (xx->ready_timeout_id) {
           xx->ready_timeout_id = 0;
           xx->card_event |= XX_CARD_READY | XX_CARD_ERROR;
   }
   if (xx->card_event & XX_CARD_WAIT_READY) {
           xx->card_event &= ~XX_CARD_WAIT_READY;
           cv_broadcast(&xx->readywait_cv);
   }
   mutex_exit(&xx->event_mutex);
}
```

Card Removal

When a PC Card is removed from a PCMCIA socket, Card Services delivers a CS_EVENT_CARD_REMOVAL event to the PC Card driver's event handler for the instance of that driver associated with that card. Card removal events are delivered to the driver twice, first as a high-priority card removal event, then as a low-priority card removal event. These event priorities always occur in a predetermined sequence, with high-priority events occuring first and low-priority events later.

The purpose of the high-priority card removal event is to immediately notify the driver that the card has been removed. The high-priority card removal event allows the driver to set a bit in the driver's soft state so that when the card is removed the driver can break out of any loops it may be processing, and can stop attempting to service interrupts for the card.

The purpose of the low-priority card removal event is to release all I/O and IRQ resources acquired from the time the card was inserted. These resources can be released using the functions `csx_ReleaseConfiguration`(9F), `csx_ReleaseIO`(9F), `csx_ReleaseIRQ`(9F), and `csx_ReleaseWindow`(9F).

High-Priority Card Removal

High-priority card removal events are delivered with priority CS_EVENT_PRI_HIGH and are always delivered before the corresponding low-priority card removal event. High-priority card removal events are delivered to the driver above lock level, and the event handler must acquire the high-priority event mutex to manage this event.

Code Example 6-5 shows an event handler dealing with a high-priority card removal event.

Code Example 6-5 Event Handler for High-Priority Event

```
xx_event_handler(event_t event, int priority,
    event_callback_args_t *eca)
{
   int             rval;
   xx_unit_t      *xx = eca->client_data;

   switch (event) {
   ...

   case CS_EVENT_CARD_REMOVAL:
       if (priority & CS_EVENT_PRI_HIGH) {
               mutex_enter(&xx->event_hi_mutex);
               cbt->card_state &= ~XX_CARD_INSERTED;
               mutex_exit(&xx->event_hi_mutex);
               rval = CS_SUCCESS;
       } else {
               rval = xx_card_removal(xx);
       }
       break;
   ...

   return (rval);
}
```

Low-Priority Card Removal

After the high-priority card removal event has been delivered, Card Services schedules and delivers a low-priority card removal event of priority CS_EVENT_PRI_LOW. The driver should use its low-priority event mutex to manage this event.

When a card removal event is delivered, a driver that uses interrupts must first acquire the I/O mutex in the event handler to cause the driver to spin until any pending I/O interrupts have completed. Clearing the card inserted or ready state during the earlier high-priority Card Removal event permits the driver to ensure that any interrupt triggered after that event can return without initiating any activity on the device or even acquiring the I/O mutexes. This is important since the device may be sharing the interrupt line with other devices, and the driver should not stall or interfere with those devices in any way.

Code Example 6-6 shows a typical low-priority card removal event.

Code Example 6-6 Event Handler for Low-Priority Event

```
static int
xx_card_removal(xx_state_t *xx)
{
   int                    ret;
   sockevent_t            sockevent;
   remove_device_node_t   remove_device_node;
   io_req_t               io_req;
   modify_config_t        modify_config;
   release_config_t       release_config;
   irq_req_t              irq_req;

   ASSERT(mutex_owned(&xx->event_mutex));

   /* Remove any pending card ready timer */
   if (xx->ready_timeout_id) {
       int id = xx->ready_timeout_id;
       xx->ready_timeout_id = 0;
       mutex_exit(&xx->event_mutex);
       untimeout(id);
       mutex_enter(&xx->event_mutex);
   }

   /* Did the card fail initialization? */
```

```
if (xx->card_event & XX_CARD_ERROR) {
    xx->card_event &= ~(XX_CARD_READY |
        XX_CARD_WAIT_READY | XX_CARD_INSERTED);
    xx->card_event &= ~XX_CARD_BUSY;
    return (CS_SUCCESS);
}

/*
 * First, grab the I/O mutex. This will cause the driver
 * to spin until any pending I/O interrupts have completed.
 */
xx->card_event &= ~XX_CARD_INSERTED;

mutex_enter(&xx->irq_mutex);

xx->card_event &= ~(XX_CARD_READY | XX_CARD_ERROR |
    XX_CARD_WAIT_READY);

/*
 * Now that we are sure the interrupt handlers won't
 * attempt to initiate any activity, we can continue
 * freeing this card's I/O resources.
 */
mutex_exit(&xx->irq_mutex);

modify_config.Attributes = 0;
modify_config.Vpp1 = 0;
modify_config.Vpp2 = 0;
modify_config.Attributes = (CONF_VPP1_CHANGE_VALID |
    CONF_VPP1_CHANGE_VALID | CONF_VPP2_CHANGE_VALID);

ret = csx_ModifyConfiguration(xx->handle, &modify_config);

ret = csx_ReleaseConfiguration(xx->handle, &release_config);

ret = csx_ReleaseIRQ(xx->handle, &irq_req);

mutex_destroy(&xx->irq_mutex);

ret = csx_ReleaseIO(xx->handle, &io_req);

remove_device_node.Action = REMOVAL_ALL_DEVICE_NODES;
remove_device_node.NumDevNodes = 0;

ret = csx_RemoveDeviceNode(xx->handle, &remove_device_node);
```

```
    xx->card_event &= ~XX_CARD_BUSY;

    return (CS_SUCCESS);
}
```

6

PC Card Configuration 7

This chapter discusses allocation of system resources to a PC Card and provides example code illustrating PC Card configuration.

PC Card Configuration Overview

When a PC Card has been inserted and is ready for further configuration (Card Services delivers a CS_EVENT_CARD_READY event), the PC Card driver determines the card's capabilities and negotiates with Card Services for the resources it needs.

For cards requiring I/O or IRQ resources, the driver examines the card's CIS structure to determine the needed I/O address space and interrupts, and then requests the resources through Card Services. When all resources have been obtained, the driver configures the card to use the resources. For cards requiring memory resources, the driver may need to examine the CIS structure to determine the characteristics of the PC Card's memory address space before mapping the card's memory into a block of system address space.

In general, the basic tasks in PC Card configuration are:

- Selecting a configuration option appropriate for the card and the available system resources
- Requesting resources
- Configuring the card, if necessary

The following sections discuss these tasks in more detail.

Selecting a Configuration Option

To select a configuration option, PC Card drivers examine the card's CIS tuple list to determine the card's characteristics and possible configurations of registers and interrupts. The driver and Card Services use this information to select the best configuration, given the card's capabilities and the available system resources.

The PCMCIA framework usually presents drivers with a list of possible configuration choices. A PC Card driver should extract all possible configurations and then sort those configurations into a preferred order. The PC Card driver then requests the set of configuration parameters it wants, and Card Services informs the driver if those resources are available. If they are not, the driver tries another combination. It isn't always possible to acquire the exact resources desired; however, different configurations might be found to work.

In general, configurations that specify resources that are typically used by non-PCMCIA devices in the system should have the lowest preference. Configurations that specify the minimum number of address lines should be given the highest preference. For example, if a serial card has five configurations where the first four specify the I/O addresses for COM1 through COM4, but the last configuration allows the I/O address to be relocated anywhere, the last one is the most preferred. If the card is essentially the same as a standard PC device (serial cards usually fit this criteria), then a configuration should be tried with the preference going to the value least likely to exist in a system. For serial cards, the order would be COM4 through COM1, for example.

PC Card configuration requirements vary depending on the type of card being configured. In general, PC Card memory drivers have simple configuration requirements (which may all be contained, for example, in the Device Information Tuple), while I/O PC Card drivers may experience considerably more difficulty matching resources to PC Card requirements.

Processing the CIS Tuples

To examine a PC Card's CIS, the driver uses the Card Services functions csx_GetFirstTuple(9F) and csx_GetNextTuple(9F) to step through the tuple data structures and locate the desired tuple. The csx_GetFirstTuple(9F) and csx_GetNextTuple(9F) functions enable a client to traverse the CIS without being aware of how tuple links are evaluated.

The linked list of tuples may be inspected one by one, or the driver may narrow the search by requesting only tuples of a particular type. The driver requests a specific tuple by filling in the tuple(9S) structure with the DesiredTuple and other necessary information. The tuple(9S) data structure is defined as:

```
typedef struct tuple_t {
   uint32_t    Socket;           /* socket number */
   uint32_t    Attributes;       /* tuple attributes */
   cisdata_t   DesiredTuple;     /* tuple to search for */
   cisdata_t   TupleOffset;      /* tuple data offset */
   cisdata_t   TupleDataMax;     /* max tuple data size */
   cisdata_t   TupleDataLen;     /* actual tuple data length */
   cisdata_t   TupleData[CIS_MAX_TUPLE_DATA_LEN]; /* body tuple */
   cisdata_t   TupleCode;        /* tuple type code */
   cisdata_t   TupleLink;        /* tuple link */
} tuple_t;
```

Once a tuple has been located, the PC Card driver can inspect the tuple data using the tuple parsing functions, such as csx_ParseTuple(9F) and csx_Parse_CISTPL_DEVICE(9F). For a list of the tuple parsing functions, see "Tuple Parsing Functions" on page 27. Note that for tuples for which no tuple parsing function is provided, the PC Card driver can retrieve the raw tuple data using csx_GetTupleData(9F). This might be the case for nonstandard, vendor-specific tuples, which the driver would need to parse itself.

Code Example 7-1 shows how the tuple data for the CISTPL_VERS_1 tuple can be obtained. For a complete example of tuple parsing, see "pcepp_parse_cis()" on page 125.

Code Example 7-1 Parsing CISTPL_VERS_1 Tuple Data

```
static int
xx_parse_cis(xx_state_t *xx, xx_cftable_t **cftable)
{
```

```
int                 i;
int                 ret;
...
tuple_t             tuple;
...
cistpl_vers_1_t     cistpl_vers_1;

...
/* Clear the CIS saving information structure */
bzero(cis_vars, sizeof (xx_cis_vars_t));

/* CISTPL_VERS_1 processing */
bzero(&tuple, sizeof (tuple));
tuple.DesiredTuple = CISTPL_VERS_1;
ret = csx_GetFirstTuple(xx->client_handle, &tuple);

bzero(&cistpl_vers_1, sizeof (struct cistpl_vers_1_t));

ret = csx_Parse_CISTPL_VERS_1(xx->client_handle,
        &tuple, &cistpl_vers_1);
    cis_vars->major_revision = cistpl_vers_1.major;
    cis_vars->minor_revision = cistpl_vers_1.minor;
    cis_vars->nstring = cistpl_vers_1.ns;
    for (i = 0; i < cistpl_vers_1.ns; i++) {
        strcpy(cis_vars->prod_strings[i],
                cistpl_vers_1.pi[i]);
    }
}
....
```

Card Configuration for I/O Cards

I/O PC Cards typically require the use of system I/O space and interrupts. The process of card configuration for cards requiring I/O resources normally includes these tasks:

- Requesting I/O registers with `csx_RequestIO`(9F)
- Installing an interrupt handler with `csx_RequestIRQ`(9F)
- Configuring the card with `csx_RequestConfiguration`(9F)

These tasks are described in the sections that follow. Note that some I/O cards may require memory windows; for information on allocating memory resources, see “Card Configuration for Memory Cards” on page 77.

Requesting I/O Resources

If a PC Card requires I/O resources, the driver requests the resources from Card Services by calling `csx_RequestIO`(9F).

```
int32_t csx_RequestIO(client_handle_t ch, io_req_t *ir);
```

The `csx_RequestIO`(9F) `io_req_t` structure describes the requested resources. It specifies one or two ranges of addresses, the logical socket number, and the number of address lines decoded by the PC Card in the specified socket. Each address range is specified by the:

- Base port address
- Number of contiguous ports
- Address range attributes, which specify the data path width, and byte-ordering and data-ordering characteristics

The `io_req_t` structure contains the following members:

```
typedef struct io_req_t {
   uint32_t       Socket;            /* logical socket number */
   uint32_t       Baseport1.base;    /* base port address */
   acc_handle_t   Baseport1.handle;  /* access handle */
   uint32_t       NumPorts1;         /* 1st set contiguous ports */
   uint32_t       Attributes1;       /* 1st  attributes */
   uint32_t       Baseport2.base;    /* base port address */
   acc_handle_t   Baseport2.handle;  /* access handle */
   uint32_t       NumPorts2;         /* 2nd set contiguous ports */
   uint32_t       Attributes2;       /* 2nd attributes */
   uint32_t       IOAddrLines;/* number of address lines decoded */
} io_req_t;
```

If the requested base port is set to zero, Card Services returns an I/O resource based on the available I/O resources and the number of contiguous ports requested. In this case, Card Services aligns the returned resource in the host system's I/O address space on a boundary that is a multiple of the number of contiguous ports requested, rounded up to the nearest power of two. For example, if a client requests two I/O ports, the resource returned will be a multiple of two. If a client requests five contiguous I/O ports, the resource returned will be a multiple of eight.

`csx_RequestIO`(9F) returns an access handle corresponding to the first byte of the allocated I/O window. The PC Card driver must use this handle to access locations within the requested I/O port through the Card Services Common Access functions.

On some systems, the driver will have to make multiple calls to csx_RequestIO(9F) with different resource requests to find an acceptable combination of parameters that can be used by Card Services to allocate I/O resources. Note that the card is not configured until the driver calls csx_RequestConfiguration(9F); csx_RequestIO(9F) does not configure the card.

Notes on I/O Resource Allocation

- It is important for drivers to use the minimum amount of I/O resources necessary. One way to do this is for the driver to parse the CIS of the PC Card and call csx_RequestIO(9F) with the minimum number of address lines necessary to decode the I/O space on the PC Card.
- The driver must take care not to choose a configuration that would cause system resource conflicts. On x86 machines, for example, 0x3F8 is generally known as the I/O port address of the COM1 serial port. Drivers should avoid using this address (or address range).
- For cards that support relocatable I/O addresses, the preferred way of requesting an I/O address is to request an address of 0. This results in an I/O address being allocated on the appropriate boundary for the size.

Installing an Interrupt Handler

If the PC Card requires an interrupt line, the driver installs the interrupt and registers an interrupt handler using csx_RequestIRQ(9F).

```
int32_t csx_RequestIRQ(client_handle_t, irq_req_t *);
```

In the irq_req_t structure, the driver must set the irq_handler field to the address of the interrupt handler, and set the IRQ_TYPE_EXCLUSIVE attribute, indicating that the system IRQ is dedicated to the PC Card. The irq_handler_arg field is normally set to the address of the driver's per-instance soft-state structure. csx_RequestIRQ(9F) returns an iblk_cookie that must be used to set up the mutex used in the driver interrupt handler. The irq_req_t structure is defined as:

```
typedef struct irq_req_t {
   uint32_t              Socket;
   uint32_t              Attributes;  /* IRQ attribute flags */
   csfunction_t          *irq_handler;
   void                  *irq_handler_arg;
```

```
   ddi_iblock_cookie_t   *iblk_cookie;/* IRQ iblk cookie */
   ddi_idevice_cookie_t  *idev_cookie;/* IRQ idev cookie */
} irq_req_t;
```

High-Level Interrupts

Because PCMCIA interrupts are always shared, PC Card driver interrupt handlers always run above lock level and may not allocate memory or use most other system services. In addition to installing a high-level interrupt handler, the driver must install a soft interrupt handler with `ddi_add_softintr`(9F) and request a soft interrupt cookie that must be used to set up the mutex used for the soft interrupt handler. See `ddi_intr_hilevel`(9F) for more information about the restrictions imposed on high-level interrupt handlers.

Two mutexes are needed to manage a high-level interrupt. One mutex protects the high-level interrupt handler and must be initialized with the interrupt cookie returned by `csx_RequestIRQ`(9F). The other mutex protects the soft interrupt handler and must be initialized with the interrupt cookie returned by `ddi_get_soft_iblock_cookie`(9F). This second mutex is also used throughout the driver to protect data from parallel access by an interrupt.

Code Example 7-2 installs a high-level interrupt handler and a soft interrupt handler.

Code Example 7-2 Installing High-Level and Soft Interrupt Handlers

```
static int
xx_card_configuration(xx_state_t *xx)
...

   xx_t   *xx;       /* soft state */
   u_int  xx_intr(caddr_t);/* high-level interrupt handler */
   u_int  xx_softintr(caddr_t);/* software interrupt handler */

   ...
   /* Allocate an IRQ */
   irq_req.Attributes = IRQ_TYPE_EXCLUSIVE;
   irq_req.irq_handler = (csfunction_t *)xx_intr;
   irq_req.irq_handler_arg = (caddr_t)xx;

   ret = csx_RequestIRQ(xx->client_handle, &irq_req);
```

```
/*
 * Initialize the interrupt mutex for protecting
 *  the card registers.
 */
mutex_init(&xx->high_mutex, "xx->high_mutex", MUTEX_DRIVER,
               *(irq_req.iblk_cookie));

/*
 * Add soft interrupt handler, which is triggered from
 * high-level interrupt handler to service the interrupt.
 */
ret = ddi_get_soft_iblock_cookie(xx->xx_dip, DDI_SOFTINT_LOW,
         &xx->softint_cookie);

mutex_init(&xx->softint_mutex, "softint mutex", MUTEX_DRIVER,
       xx->softint_cookie);

ret = ddi_add_softintr(xx->dip, DDI_SOFTINT_LOW,
       &xx->softint_id, &xx->softint_cookie,
       (ddi_idevice_cookie_t *)NULL, xx_softintr,
       (caddr_t)xx);
...
```

The protocol between the high-level interrupt handler and the software interrupt must be carefully arranged. The high-level interrupt handler must only acquire the mutex initialized with the cookie returned by `csx_RequestIRQ`(9F). This interrupt handler must then determine if the device is in fact interrupting, arrange to begin servicing the device, trigger the software interrupt to continue servicing the interrupt by calling `ddi_trigger_softintr`(9F), and finally, return `DDI_INTR_CLAIMED`. If the device is not generating the interrupt, the interrupt handler should return `DDI_INTR_UNCLAIMED`.

Example High-Level Interrupt Handler

The high-level interrupt handler determines whether this instance of the device is the interrupting device, services the device, and triggers the software interrupt.

Note that the high-level interrupt handler must not acquire any locks other than the lock initialized with the high-level interrupt cookie, and it must drop the high-level mutex before triggering the software interrupt.

Code Example 7-3 High-Level Interrupt Handler

```
static u_int
xx_intr(caddr_t arg)
{
   xx_state_t *xx = (xx_state_t *)arg;

   /*
    * If the card isn't inserted or fully initialized yet,
    * this isn't an interrupt for us.  We must do this before
    * grabbing the mutex, since with shared interrupts, we
    * may get interrupts from other sources before we are fully
    * prepared for them.  We also need to stop accessing the
    * card promptly when the card gets yanked out
    * from under us.  The high-level card removal processing
    * clears the Card Inserted bit.
    */
   if (XX_CARD_IS_READY(xx)) {
       return (DDI_INTR_UNCLAIMED);
   }

   mutex_enter(&xx->high_mutex);

   /* If we are already interrupting, not for us */
   if (xx->is_interrupting == 1) {
       mutex_exit(&xx->high_mutex);
       return (DDI_INTR_UNCLAIMED);
   }

   if ( /* the device is interrupting */ ) {
       /* service device and disable interrupts */
       xx->is_interrupting = 1;
   }
   mutex_exit(&xx->high_mutex);

   if (xx->is_interrupting == 1) {
       ddi_trigger_softintr(xx->softint_id);
       return (DDI_INTR_CLAIMED);
   }

   return (DDI_INTR_UNCLAIMED);
}
```

Example Soft Interrupt Handler

The software interrupt routine is started by the high-level interrupt handler. It completes the task of processing the data.

Note that the software interrupt must first acquire the mutex initialized with the cookie returned by `ddi_get_soft_iblock_cookie`(9F), then acquire the high-level mutex before continuing on to service the device interrupt.

Code Example 7-4 Soft Interrupt Handler

```
static u_int
xx_softintr(caddr_t arg)
{
   xx_t    *xx = (xx_t *)arg;

   if (!XX_CARD_IS_READY(xx)) {
       return (DDI_INTR_UNCLAIMED);
   }

   mutex_enter(&xx->softint_mutex);
   mutex_enter(&xx->high_mutex);

   if (xx->is_interrupting == 0) {
       mutex_exit(&xx->high_mutex);
       mutex_exit(&xx->softint_mutex);
       return (DDI_INTR_UNCLAIMED);
   }

   mutex_exit(&xx->high_mutex);
   /* service the interrupt */

   mutex_enter(&xx->high_mutex);
   /* re-enable device interrupts here */

   xx->is_interrupting = 0;
   mutex_exit(&xx->high_mutex);

   mutex_exit(&xx->softint_mutex);

   return (DDI_INTR_CLAIMED);
}
```

Removing an Interupt Handler and Soft Interrupt Handler

Code Example 7-5 releases allocated IRQ resources. This code might be included in a card removal routine.

Code Example 7-5 Releasing IRQ Resources

```
...
/* Unregister the softinterrupt handler */
ddi_remove_softintr(xx->softint_id);

/* Release allocated IRQ resources. */
ret = csx_ReleaseIRQ(xx->client_handle, &irq_req);

/* Destroy mutexes */
mutex_destroy(&xx->high_mutex);
mutex_destroy(&xx->softint_mutex);
...
```

Configuring the PC Card and Socket

Once suitable IO and IRQ resources are found, the driver must call `csx_RequestConfiguration`(9F) to apply power to the socket and make the I/O and IRQ resources active. PC Card drivers can read or write to the allocated I/O port after calling `csx_RequestConfiguration`(9F).

```
int32_t csx_RequestConfiguration(client_handle_t, config_req_t *);
```

If the configuration attribute CONF_ENABLE_IRQ_STEERING is set in the `csx_RequestConfiguration`(9F) structure `config_req_t Attributes` field, `csx_RequestConfiguration`(9F) connects the PC Card interrupt line to a system interrupt previously selected by a call to `csx_RequestIRQ`(9F). When `csx_RequestConfiguration`(9F) returns successfully, the driver will receive IRQ callbacks at the interrupt handler established in the call to `csx_RequestIRQ`(9F).

Note – By default, PC Cards are in memory-only mode. The I/O address range in the PC Card is not enabled until `csx_RequestConfiguration`(9F) is called to configure the card.

---

csx_AccessConfigurationRegister(9F) provides access to configuration registers in attribute memory space. This function accesses the requested configuration register directly and does not return an access handle; consequently, it does not require the PC Card driver to use the Card Services Common Access functions to access these registers.

Releasing I/O and IRQ Resources

To release the current PC Card and socket configuration, the driver must call csxReleaseConfiguration(9F). After a call to csxReleaseConfiguration(9F), any I/O or IRQ resources that are no longer needed should be returned to Card Services. To return I/O resources to Card Services, use csx_ReleaseIO(9F). To return IRQ resources to Card Services, use csx_ReleaseIRQ(9F).

I/O Card Configuration Example

Code Example 7-6 shows an example of a card configuration routine for an I/O device, with emphasis on allocating I/O resources and configuring the card.

Code Example 7-6 Card Configuration for an I/O Device

```
static int
xx_card_configuration(xx_state_t *xx)
{
   int                    ret;
   char                   devname[16];
   char                   *dname;
   io_req_t               io_req;
   irq_req_t              irq_req;
   get_status_t           get_status;
   sockevent_t            sockevent;
   config_req_t           config_req;
   make_device_node_t     make_device_node;
   devnode_desc_t         *dnd;
   xx_cftable_t           *cftable = NULL;
   xx_cftable_t           *cft;
   xx_cis_vars_t          *cis_vars = &xx->cis_vars;

   ASSERT(mutex_owned(&xx->event_mutex));

   /* Get card state status */
```

```
/* Get PC Card CIS information */
ret = xx_parse_cis(xx, &cftable);

/* Try to allocate IO resources; if fail then exit */
cft = cftable;
while (cft) {
    io_req.BasePort1.base = cft->p.card_base1;
    io_req.NumPorts1 = cft->p.length1 + 1;
    io_req.Attributes1 = (IO_DATA_PATH_WIDTH_8 |
                WIN_ACC_NEVER_SWAP |
                WIN_ACC_STRICT_ORDER);
    io_req.BasePort2.base = 0;
    io_req.NumPorts2 = 0;
    io_req.Attributes2 = 0;
    io_req.IOAddrLines = cft->p.addr_lines;

    ret = csx_RequestIO(xx->client_handle, &io_req);
        cis_vars->card_base1 = cft->p.card_base1;
        cis_vars->length1 = cft->p.length1;
        cis_vars->card_base2 = cft->p.card_base2;
        cis_vars->length2 = cft->p.length2;

        cis_vars->addr_lines = cft->p.addr_lines;
        cis_vars->card_vcc = cft->p.card_vcc;
        cis_vars->card_vpp1 = cft->p.card_vpp1;
        cis_vars->card_vpp2 = cft->p.card_vpp2;
        cis_vars->pin = cft->p.pin;
        cis_vars->config_index = cft->p.config_index;
        break;

    cft = cft->next;
}
...

/* Normal eight contiguous registers */
xx->card_handle = io_req.BasePort1.handle;

xx->flags |= XX_REQUESTIO;

/* Allocate an IRQ */
irq_req.Attributes = IRQ_TYPE_EXCLUSIVE;
irq_req.irq_handler = (csfunction_t *)xx_intr;
irq_req.irq_handler_arg = (caddr_t)xx;

ret = csx_RequestIRQ(xx->client_handle, &irq_req);
```

```
/* Initialize the interrupt mutex */
mutex_init(&xx->irq_mutex, "xx->irq_mutex", MUTEX_DRIVER,
               *(irq_req.iblk_cookie));
xx->flags |= XX_REQUESTIRQ;

/* Set up the client event mask */
sockevent.Attributes = CONF_EVENT_MASK_CLIENT;
ret = csx_GetEventMask(xx->client_handle,&sockevent);

sockevent.EventMask |= CS_EVENT_CARD_READY;
ret = csx_SetEventMask(xx->client_handle, &sockevent);

/* Configure the PC Card */
config_req.Attributes = 0;
config_req.Vcc = cis_vars->card_vcc;
config_req.Vpp1 = cis_vars->card_vpp1;
config_req.Vpp2 = cis_vars->card_vpp2;
config_req.IntType = SOCKET_INTERFACE_MEMORY_AND_IO;
config_req.ConfigBase = cis_vars->config_base;
config_req.Status = 0;
config_req.Pin = cis_vars->pin;
config_req.Copy = 0;
config_req.ConfigIndex = cis_vars->config_index;
config_req.Present = cis_vars->present;

ret = csx_RequestConfiguration(xx->client_handle,&config_req);

xx->flags |= XX_REQUESTCONFIG;

/*
 *  Create the minor devices for this instance
 *  The minor number is the socket number
 */
dname = devname;
make_device_node.Action = CREATE_DEVICE_NODE;
make_device_node.NumDevNodes = 1;

make_device_node.devnode_desc =
    kmem_zalloc(sizeof (struct devnode_desc) *
        make_device_node.NumDevNodes, KM_SLEEP);

dnd = &make_device_node.devnode_desc[0];
dnd->name = dname;
dnd->spec_type = S_IFCHR;
dnd->minor_num = xx->sn;
dnd->node_type = XX_NT_PARALLEL;
```

```
    ret = csx_MakeDeviceNode(xx->client_handle,&make_device_node);
    ...
}
```

Card Configuration for Memory Cards

The memory allocation process for memory PC Cards involves mapping the memory address range within the PC Card driver's address space in system address space into PC Card memory. Although memory PC Cards typically don't require I/O space or IRQs, they may request I/O resources if necessary. In addition, nonmemory cards that have common or attribute memory that the driver needs to access (for example, some I/O cards) may allocate memory windows.

Drivers that require memory resources can map the PC Card's memory to system address space by:

- Requesting system address space with `csx_RequestWindow`(9F)
- Mapping PC Card memory to the allocated system address space with `csx_MapMemPage`(9F)

The following sections provide information on these tasks.

Requesting System Address Space

To request that an area of system address space be assigned to a region of PC Card common or attribute memory, a PC Card driver calls the `csx_RequestWindow`(9F) function.

```
int32_t csx_RequestWindow(client_handle_t, window_handle_t *,
                          win_req_t *);
```

The `csx_RequestWindow`(9F) `win_req_t` structure describes the requested memory resources. It specifies the following:

- Base address of the memory window
- Size of the requested memory area
- Memory window attributes that define the type of memory window, the data path width, and the byte-ordering and data-ordering characteristics. An attribute also sets a bit to enable the memory window.

- Parameter for the window access speed if the driver is requesting a memory window
- Parameter for I/O address lines decoded for an I/O window if the driver is using csx_RequestWindow(9F) to request an I/O address range
- Logical socket number
- Window offset for use with csx_MapMemPage(9F)

The win_req_t structure is defined as:

```
typedef struct win_req_t {
   uint32_t       Socket;
   uint32_t       Attributes;    /* window flags */
   uint32_t       Base.base;     /* requested window base address */
   acc_handle_t   Base.handle;   /* handle for base of window */
   uint32_t       Size;          /* window size */
   uint32_t       win_params.AccessSpeed;/* window access speed */
   uint32_t       win_params.IOAddrLines;/* for I/O windows only */
   uint32_t       ReqOffset;     /* required window offset */
} win_req_t;
```

Note the following when requesting memory address space:

- The driver should set the base address of the memory window to zero. The driver should *not* request a specific physical address, as a request of this type will most likely fail.
- Due to hardware dependencies, do not assume that drivers can request a memory window of any specific size; instead, be prepared to deal with different window sizes. For example, if there are memory window sizes that the driver can optimize, it can request those sizes first. In most cases, the driver will register a memory window size of zero to instruct Card Services to return the smallest available window.

csx_RequestWindow(9F) returns the size of the allocated memory area and returns a window handle that corresponds to the first byte of the allocated memory window. The driver will use this handle when accessing the PC Card's memory space through the Common Access functions.

Note – Driver developers should be aware that if any nonzero memory window size is specified, the driver may not be portable. This is due to the fact that some adapters have fixed size memory windows of 1 Mbyte, while others have variable-sized windows. Drivers that request specific sizes must be

prepared to negotiate sizes in order to be portable. Also note that SPARC machines require data access on natural boundaries (byte on byte, short on short, and long on long), while the x86 platform has no such alignment constraints and can access a long on any arbitrary boundary.

Mapping PC Card Memory to System Address Space

Once a memory window has been allocated and the window handle has been obtained, the driver can map the memory area on the PC Card to the allocated memory using the `csx_MapMemPage`(9F) function. `csx_MapMemPage`(9F) sets the offset of the PC Card to the allocated window. This is illustrated in Figure 7-1, where a 4 Kbyte window is shown mapped to a PC Card with 64 Kbytes of common or attribute memory space.

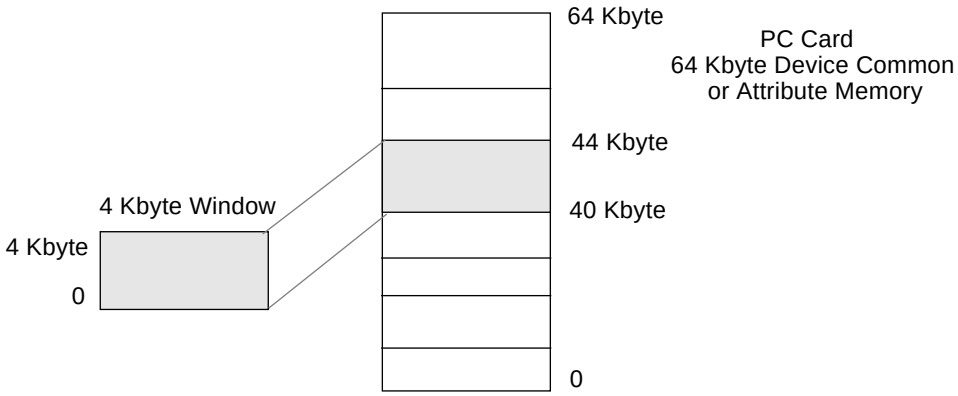


Figure 7-1 Mapping PC Card Memory to System Address Space

The 4 Kbyte memory window in the figure can be moved or remapped anywhere within the 64 Kbyte common or attribute memory space according to the needs of the driver or the PC Card device. In this sense, windows can be made to *slide* across PC Card memory space.

The offset in bytes from the beginning of the PC Card to the memory area is defined by the value of `map_mem_page_t` `CardOffset`. The driver cannot depend on the card offset being set to any particular default value for any particular window.

```
int32_t csx_MapMemPage(window_handle_t, map_mem_page_t *);
```

```
typedef struct map_mem_page_t {
   uint32_t       CardOffset;/* card offset */
   uint32_t       Page;      /* page number */
} map_mem_page_t;
```

Note that a driver should never position a window to an arbitrary offset. A window offset must also take into account window alignment and size requirements.

The csx_RequestWindow(9F) win_req_t structure returns a WIN_OFFSET_SIZE flag in the Attributes field that indicates that all offsets used when calling csx_MapMemPage(9F) must be a multiple of the window size returned from csx_RequestWindow(9F). For example, if csx_RequestWindow(9F) returns 4 Kbytes in the Size field of the win_req_t structure, the PC Card driver is only allowed to give csx_MapMemPage offsets in multiples of 4 Kbytes (0 Kbytes, 4 Kbytes, 8 Kbytes, 16 Kbytes, and so on).

Modifying a Memory Address Window

A PC Card driver can modify the memory access speed or the type of memory address window with csx_ModifyWindow(9F). The type of memory window can be changed from common to attribute, or vice versa.

Releasing Memory Resources

To release memory resources, the driver must call csx_ReleaseWindow(9F). This function releases window resources that were allocated by csx_RequestWindow(9F). No adapter or socket hardware is modified by csx_ReleaseWindow(9F).

Memory Card Configuration Example

Code Example 7-7 shows a card configuration routine for a memory device, with emphasis on requesting memory resources and configuring the card.

Code Example 7-7 Card Configuration for a Memory Device

```
static int
xx_card_insertion(xx_state_t  *xx)
```

```
{
   int                ret;
   int                rval = CS_SUCCESS;
   sockevent_t        se;
   win_req_t          win_req;
   convert_speed_t    convert_speed;
   map_mem_page_t     map_mem_page;
   get_status_t       get_status;

   /* Get card state status */
   ret = csx_GetStatus(xx->client_handle, &get_status);

   /* Make sure that there is a card in the socket */
   if (!(get_status.CardState & CS_EVENT_CARD_INSERTION)) {
       /* error handling */
   }

   /* Set up the client event mask */
   se.Attributes = CONF_EVENT_MASK_CLIENT;
   ret = csx_GetEventMask(xx->client_handle, &se);

   se.EventMask |= (CS_EVENT_BATTERY_LOW |
              CS_EVENT_BATTERY_DEAD |
              CS_EVENT_WRITE_PROTECT);

   ret = csx_SetEventMask(xx->client_handle, &se);

   /*
   /* if necessary, release any allocated windows before
   /* requesting new windows
   /*
   ...

   /* Try to get a memory window to Common Memory space */
   win_req.Attributes = (WIN_MEMORY_TYPE_CM |
              WIN_DATA_WIDTH_16 |
              WIN_ENABLE |
              WIN_ACC_NEVER_SWAP |
              WIN_ACC_STRICT_ORDER);
   win_req.Base.base = 0;/* let CS find a base address */
   win_req.Size = 0;/* let CS return the smallest size window */

   convert_speed.Attributes = CONVERT_NS_TO_DEVSPEED;
   convert_speed.nS = 250;
   csx_ConvertSpeed(&convert_speed);
```

```
    win_req.win_params.AccessSpeed = convert_speed.devspeed;

    ret = csx_RequestWindow(xx->client_handle,
            &xx->window_handle, &win_req);

    xx->flags |= XX_HAS_WINDOW;

    /* Now map the offset to the start of the card */
    map_mem_page.CardOffset = 0;
    map_mem_page.Page = 0;

    ret = csx_MapMemPage(xx->window_handle,&map_mem_page);

    /* Store xx->access_handle */
    xx->access_handle = win_req.Base.handle;
    xx->win_size = win_req.Size;

    /* Read CIS information for the card size */
    ...

    /* Create the device nodes */
    ...

    xx->card_event |= XX_CARD_IS_READY;
    ...

    return (rval);
}
```

Portable PC Card Drivers 8

The Solaris Card Services interfaces have been designed to permit PC Card drivers to be written in an architecturally independent manner. This chapter discusses portability issues that a driver writer should be aware of when writing a PC Card driver that is portable across Solaris platforms.

Portability Issues

A PC Card driver should be written so that it is portable across all Solaris platforms. This leads to easier code maintainability, easier testing, and reduced overall product costs. There are no performance advantages to be gained from writing a nonportable driver. The Solaris PCMCIA software framework makes it possible to write portable drivers without sacrificing performance.

To achieve portability, a driver needs to correctly handle the following:

- Data byte order
- Memory alignment
- Accessing memory windows
- Accessing I/O space
- Interrupts

Byte Ordering

Solaris Card Services provides a set of Common Access I/O and memory functions to insulate a driver from host byte-ordering dependencies. However, the driver writer needs to be aware of the byte ordering on the device itself. This can be specified at mapping time.

PC Card devices normally provide registers and memory interfaces in little-endian byte order. However, it is possible that a vendor may implement a PC Card with a big-endian access mode. A further concern is that a driver for a PC Card may be running on either a little-endian or a big-endian host architecture.

When establishing an I/O or memory mapping, the driver may specify either little-endian, big-endian, or "no swap" access. If, for example, a register set on a PC Card is defined to be little-endian and the driver establishes a little-endian mapping to this register set, then the driver can perform reads from or writes to the registers through the Common Access functions, maintaining proper byte order without regard to the host byte ordering.

A driver writer would normally ensure that memory or registers that represent data are not swapped. For example, data blocks on a disk may be mapped so that no swapping takes place on reads or writes.

Memory Alignment

Some platforms have alignment constraints on data. To be portable, a driver must ensure that the memory objects are properly aligned. For example, objects that are 4 bytes in size need to be aligned on a 4-byte boundary.

SPARC systems require strict alignment. x86 systems do not necessarily require strict alignment, but strict alignment is a more efficient mechanism and is a requirement for portable PC Card drivers.

Accessing Memory Windows

Driver writers should take into account that different architectures may place different restrictions on the amount of PC Card memory that can be mapped into system memory at any one time. In SPARC systems, the current SBus adapter allows mapping two 1 Mbyte windows per socket. In x86 systems, adapters vary considerably in their capabilities, depending on the adapter and

host bus type. Mappings on x86 systems that use PCIC-based adapters vary window sizes from 4 Kbytes to 16 Mbytes. However, that much physical memory may not be present on the machine.

PC Card drivers should inquire about limits and be designed to work in a smaller memory window, remapping as necessary. A driver should also check error returns because it is possible that a window of the requested size is not available. In this case, the driver would need to adjust downward.

The window mapping functions fail if the desired mapping is not possible. To avoid failure, a driver should follow these guidelines:

- Never request a specified physical address.
- Be prepared to deal with different memory window sizes, since requesting a specific memory window size might not succeed. If there are window sizes that the driver can optimize, it could request those sizes first. Typically, the driver should register a window size of 0 and let the framework choose the available window size.
- Never position a window to an arbitrary offset. A window offset must take into account window alignment and size requirements.

To be portable, a driver should access memory through the Common Access functions. Direct memory access may be possible but requires obtaining a mapped address, which some adapters may not support.

Accessing I/O Space

A PC Card driver should only access the PC Card I/O space through the Common Access functions provided by Solaris Card Services.

Despite the fact that obtaining memory-mapped or physical I/O addresses may work in a number of cases, there are many reasons for not using memory mapped or physical I/O addresses. Some architectures do not support a separate I/O address space, and the Common Access functions provide the abstraction necessary for accessing those addresses. Also, the Common Access functions hide some of the more unusual behavior of I/O on PC Cards and adapters.

csx_RequestIO(9F) and csx_RequestWindow(9F) supply a handle that (along with an I/O address offset) is used by the Common Access functions to perform the I/O operation. The Common Access functions are listed in Chapter 3, "Solaris Card Services."

Constraints on Use of I/O and Memory Windows

It is important to use only the necessary number of windows so that the driver can work in a variety of environments. For example, the current SPARC SBus adapter provides only two windows per socket; therefore, a PC Card driver that needs more than two windows will not work on a SPARC platform that uses the SBus PCMCIA adapter. The limits placed on windows also vary considerably across platforms.

Interrupts

A PC Card that needs to use an interrupt must be able to use an above lock-level interrupt to ensure that it can operate in a variety of architectures and environments. A PC Card driver must assume that interrupts are shared with other devices. There is no way to determine if interrupt sharing is possible, nor is there any way to request an exclusive interrupt. The type of interrupt available may change between card insertion and card removal.

Interrupt resources and the necessary mutexes should always be acquired and initialized at card insertion time, and these resources should then be freed at card removal time, so that other drivers may then use them.

For more information on interrupts, see Chapter 7, "PC Card Configuration."

PCMCIA Parallel Port Driver 9

This chapter presents the complete code for a PCMCIA parallel port driver (`pcepp.c`) and shows how a similar driver could be written. The `pcepp` driver controls a simple interrupt-per-character line printer. This driver is a unidirectional STREAMS driver and has no read-side processing.

The sample driver has the following main sections:

- Driver header files
- Local driver data and system routines and variables
- STREAMS structures
- Character/block, device operations, and modlinkage structures
- Initialization routines
- Autoconfiguration routines
- Card Services routines
- CIS configuration routines
- Device interrupt handler routines
- STREAMS routines

The `pcepp` driver source code is included in the Device Driver Developer's Kit.

9

Include Files and Header Files for Solaris Parallel Port Driver

The following `#include` header files and `#ifdef` preprocessor directives are required for the `pcepp` driver:

```
#include <sys/types.h>
#include <sys/cmn_err.h>
#include <sys/conf.h>
#include <sys/cred.h>
#include <sys/errno.h>
#include <sys/file.h>
#include <sys/termio.h>
#include <sys/termios.h>
#include <sys/kmem.h>
#include <sys/ksynch.h>
#include <sys/modctl.h>
#include <sys/open.h>
#include <sys/stat.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/strtty.h>
#include <sys/uio.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

PCMCIA Card Services Header Files

```
#include <sys/pccard.h>
```

`pcepp` *Header Files*

```
#include "pcepp.h"
#if defined(DEBUG)
static intpcepp_debug= 0;
static intpcepp_debug_cis= 0;
static intpcepp_debug_events= 0;
static intpcepp_debug_timeout= 0;
static intpcepp_debug_card_ready= 0;

#definePCEPP_DEBUG(args)if (pcepp_debug) cmn_err args
static void pcepp_display_cftable_list(pcepp_cftable_t *);
static void pcepp_display_card_config(pcepp_state_t *);
#else
#definePCEPP_DEBUG(args)
#endif
```

Local Driver Data and System Routines and Variables

The developer defines the local driver data. System routines and variables are defined in the Solaris DDK/DDI. System routines include:

- Driver configuration
- Character and block operations
- Card Services interfaces
- Card Services CIS parser interfaces
- STREAMS interfaces
- Interrupt management

Local Driver Data

```
static void *pcepp_soft_state_p = NULL;
char *pcepp_name = PCEPP_NAME;
```

Autoconfiguration Function Prototypes

```
static int pcepp_getinfo(dev_info_t *dip, ddi_info_cmd_t cmd,
                                void *arg, void **result);
static int pcepp_attach(dev_info_t *dip, ddi_attach_cmd_t cmd);
static int pcepp_detach(dev_info_t *dip, ddi_detach_cmd_t cmd);
```

STREAMS Driver open/close Procedures and Function Prototypes

```
static int pcepp_open(queue_t *q, dev_t *devp, int flag,
                                        int sflag, cred_t *credp);
static int pcepp_close(queue_t *q, int flag, cred_t *credp);
static int pcepp_ioctl(queue_t *, mblk_t *);
static void pcepp_srvioc(queue_t *, mblk_t *);
```

Card Services Function Prototypes

```
static int pcepp_event(event_t, int, event_callback_args_t *);
static int pcepp_card_ready(pcepp_state_t *);
static int pcepp_card_insertion(pcepp_state_t *);
static int pcepp_card_configuration(pcepp_state_t *pps);
static int pcepp_card_removal(pcepp_state_t *);
static void pcepp_card_ready_timeout(pcepp_state_t *);
```

Card Services CIS Parser Function Prototypes

```
static int pcepp_parse_cis(pcepp_state_t *, pcepp_cftable_t **);
static void pcepp_destroy_cftable_list(pcepp_cftable_t **);
```

```
static void pcepp_set_cftable_desireability(pcepp_cftable_t *);
static void pcepp_sort_cftable_list(pcepp_cftable_t **);
static void pcepp_swap_cft(pcepp_cftable_t **, pcepp_cftable_t *);
```

STREAMS Driver put Procedures and Function Prototypes

```
static int pcepp_rput(queue_t *, mblk_t *);
static int pcepp_wput(queue_t *, mblk_t *);
static void pcepp_start(pcepp_state_t *);
static int pcepp_xmit(pcepp_state_t *);
static void pcepp_strobe_pulse(pcepp_state_t *);
```

Interrupt Handler Function Prototypes

```
static u_int pcepp_intr(pcepp_state_t *);
static u_int pcepp_softintr(caddr_t arg);
```

Print Function Prototypes

```
static int pcepp_prtstatus(pcepp_state_t *);
```

STREAMS Structures

The pcepp driver is a unidirectional STREAMS driver. Refer to the *STREAMS Programmer's Guide* for more detailed information.

Module Information

```
static struct module_info pcepp_minfo = {
        PCEPP_IDNUM,            /* module id number */
        PCEPP_NAME,             /* module name */
        0,                      /* min packet size */
        INFPSZ,                 /* max packet size */
        PCEPP_HIWAT,            /* hi-water mark */
        PCEPP_LOWAT             /* low-water mark */
};
```

Read QUEUE Structure (qinit)

```
static struct qinit pcepp_rinit = {
        pcepp_rput,             /* put proc */
        NULL,                   /* service proc */
        pcepp_open,             /* called on startup */
        pcepp_close,            /* service procedure */
```

```
        NULL,                   /* qadmin */
        &pcepp_minfo,           /* module information structure */
        NULL                    /* module statistics structure */
};
```

Write QUEUE Structure (qinit)

```
static struct qinit pcepp_winit = {
        pcepp_wput,             /* put proc */
        NULL,                   /* service proc */
        NULL,                   /* qopen */
        NULL,                   /* qclose */
        NULL,                   /* qadmin */
        &pcepp_minfo,           /* module information structure */
        NULL                    /* module statistics structure */
};
```

STREAMS Entity Declaration Structure (streamtab)

```
struct streamtab pcepp_info = {
        &pcepp_rinit,     /* Read qinit structure */
        &pcepp_winit,     /* Write qinit structure */
        NULL,             /* Multiplexor driver - Lower Read */
        NULL              /* Multiplexor driver - Lower Write */
};
```

Autoconfiguration and Modlinkage Structures

The `cb_ops`(9S) character and block structure defines the entry points for character and block operations of device drivers. Since this is a STREAMS driver, the `cb_ops` field `cd_str` must refer to `streamtab`(9S).

Along with various autoconfiguration routines, the `dev_ops`(9S) structure contains a `cb_ops`(9S) pointer to character or block driver entry points and a `bus_ops`(9S) pointer to bus operation entry points. The `dev_ops`(9S) structure allows the kernel to find the autoconfiguration entry points of the device driver.

The `modlinkage`(9S) and `modldrv`(9S) structures are exported to the kernel, enabling drivers to be loadable modules.

Character/Block Operations Structure (cb_ops)

```
static struct cb_ops pcepp_cb_ops = {
        nodev,                  /* open                         */
        nodev,                  /* close                        */
        nodev,                  /* strategy - block devs only   */
        nodev,                  /* print - block devs only      */
        nodev,                  /* dump - block devs only       */
        nodev,                  /* read                         */
        nodev,                  /* write                        */
        nodev,                  /* ioctl                        */
        nodev,                  /* devmap                       */
        nodev,                  /* mmap                         */
        nodev,                  /* segmap                       */
        nochpoll,               /* chpoll                       */
        ddi_prop_op,            /* prop_op                      */
        &pcepp_info,            /* cb_str - STREAMS only        */
        D_NEW | D_MP,           /* driver compatibility flag    */
        CB_REV,                 /* cb_rev                       */
        nodev,                  /* async I/O read entry point   */
        nodev                   /* async I/O write entry point  */
};
```

Device Operations Structure (dev_ops)

```
static struct dev_ops pcepp_ops = {
        DEVO_REV,               /* driver build version         */
        0,                      /* device reference count       */
        pcepp_getinfo,          /* info routine                 */
        nulldev,                /* identify routine             */
        nulldev,                /* probe routine                */
        pcepp_attach,           /* attach routine               */
        pcepp_detach,           /* detach routine               */
        nodev,                  /* reset routine                */
        &pcepp_cb_ops,          /* cb_ops pntr for leaf dvr     */
        (struct bus_ops *)NULL  /* bus_ops pntr for nexus dvr   */
};
```

Module Linkage Information for the Kernel

```
extern struct mod_ops mod_driverops;

static struct modldrv modldrv = {
```

```
        &mod_driverops,         /* Type of module - driver */
        PCEPP_DRIVERID,         /* Driver identifier string     */
        &pcepp_ops              /* Device operation structure   */
};

static struct modlinkage modlinkage = {
        MODREV_1,               /* rev of loadable modules system */
        &modldrv,
        NULL
};
```

Module Initialization Functions

These routines are focused on resource allocation and resource release. The module initialization functions include:

- `_init`(9F)
- `_info`(9F)
- `_fini`(9F)

Any one-time resource allocation or data initialization should be performed during driver loading in `_init`(9E). `_fini`(9E) releases the resources allocated by `_init`(9E). `_info`(9E) identifies the loadable module (the device driver).

_init()

```
/* _init() is called by modload() */
int
_init(void)
{
        register int    error;

        PCEPP_DEBUG((CE_CONT, "pcepp: _init\n"));

        if ((error = ddi_soft_state_init(&pcepp_soft_state_p,
                        sizeof (pcepp_state_t), 1)) != 0) {
                PCEPP_DEBUG((CE_CONT,
                     "pcepp: _init ddi_soft_state_init failed\n"));
                return (error);
        }

        if ((error = mod_install(&modlinkage)) != 0) {
```

```
                        PCEPP_DEBUG((CE_CONT,
                                "pcepp: _init mod_install failed\n"));
                        ddi_soft_state_fini(&pcepp_soft_state_p);
                }

                return (error);
}
```

_info()

```
/* _info() is called by modinfo() */
int
_info(struct modinfo *modinfop)
{
        return (mod_info(&modlinkage, modinfop));
}
```

_fini()

```
/* _fini() is called by modunload() */
int
_fini(void)
{
        int     error;

        PCEPP_DEBUG((CE_CONT, "pcepp: _fini\n"));

        if (error = mod_remove(&modlinkage)) {
                PCEPP_DEBUG((CE_CONT,
                        "pcepp: _fini mod_remove failed\n"));
                return (error);
        }

        ddi_soft_state_fini(&pcepp_soft_state_p);

        return (error);
}
```

Autoconfiguration Routines

Autoconfiguration routines are used to access the soft-state structure of the device, identify the minor number of the device, attach the device to the system, transfer data and commands, and detach the device when it is no longer needed. The autoconfiguration routines are:

- pcepp_getinfo()
- pcepp_attach()
- pcepp_detach()

pcepp_getinfo()

pcepp_getinfo(9E) is called during module loading and at other times during the life of the driver. It uses csx_CS_DDI_Info(9F) as a mechanism to retrieve the instance information, since PC Card drivers encode the physical socket number as part of the dev_t of device nodes instead of the instance number.

```
static int
pcepp_getinfo(dev_info_t *dip, ddi_info_cmd_t cmd, void *arg,
                                                void **result)
{
        int             error = DDI_SUCCESS;
        pcepp_state_t   *pps;
        cs_ddi_info_t   cs_ddi_info;

        PCEPP_DEBUG((CE_CONT, "pcepp: getinfo\n"));

        switch (cmd) {

        case DDI_INFO_DEVT2DEVINFO:
                cs_ddi_info.Socket = PCEPP_SOCKET((dev_t)arg);
                cs_ddi_info.driver_name = pcepp_name;
                if (csx_CS_DDI_Info(&cs_ddi_info) != CS_SUCCESS) {
                        return (DDI_FAILURE);
                }
                if (!(pps = ddi_get_soft_state(pcepp_soft_state_p,
                                        cs_ddi_info.instance))) {
                        *result = NULL;
                } else {
                        *result = pps->dip;
```

```
                }
                break;

        case DDI_INFO_DEVT2INSTANCE:
                cs_ddi_info.Socket = PCEPP_SOCKET((dev_t)arg);
                cs_ddi_info.driver_name = pcepp_name;
                if (csx_CS_DDI_Info(&cs_ddi_info) != CS_SUCCESS) {
                        return (DDI_FAILURE);
                }
                *result = (void *)cs_ddi_info.instance;
                break;

        default:
                error = DDI_FAILURE;
                break;

        }  /* switch (cmd) */

        return (error);
}
```

pcepp_attach()

`attach`(9E) is called to attach an instance of the driver. Driver attachment can be specified as a series of steps. These steps are:

- Allocate and initialize driver soft-state structure
- Register with Card Services
- Obtain a logical socket number
- Initialize mutex and condition variables
- Install a Card Services event handler and enable event handling
- Check PCMCIA card insertion

Note that system resource allocation, such as registering device interrupts, is done in the card insertion routine, which is called when the driver receives an insertion event.

```
static int
pcepp_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
        int                     instance;
        int                     ret;
```

```
    pcepp_state_t           *pps;
                            /* CardServices variables */
    get_status_t            get_status;
    client_reg_t            client_reg;
    sockmask_t              sockmask;
    map_log_socket_t        map_log_socket;

    instance = ddi_get_instance(dip);

    PCEPP_DEBUG((CE_CONT, "pcepp%d: attach\n”, instance);

   switch (cmd) {
    case DDI_ATTACH:
        break;

    default:
        PCEPP_DEBUG((CE_NOTE, "pcepp%d: attach cmd 0x%x\n",
            instance, cmd));
        return (DDI_FAILURE);
}

PCEPP_DEBUG((CE_CONT, "pcepp%d: attach\n", instance));
```

Allocate and initialize driver soft state associated with this instance

```
    /* Allocate soft state associated with this instance. */
    if (ddi_soft_state_zalloc(pcepp_soft_state_p,
                            instance) != DDI_SUCCESS) {
            PCEPP_DEBUG((CE_CONT, "pcepp%d: attach, "
                    "could not allocate state structure\n",
                    instance));
            return (DDI_FAILURE);
    }

    pps = ddi_get_soft_state(pcepp_soft_state_p, instance);
    if (pps == NULL) {
            PCEPP_DEBUG((CE_CONT, "pcepp%d: attach, "
                 "could not get state structure\n", instance));
            goto out;
    }

    /* Remember dev_info structure for pcepp_getinfo  */
```

```
pps->dip = dip;
pps->instance = instance;
ddi_set_driver_private(dip, (caddr_t)pps);

/* clear driver state flags */
pps->flags = 0;

/*
 * Initialize the card event
 * when we get a card insertion event
 * this card_event will change.
 */
pps->card_event = 0;
```

Register with Card Services

```
client_reg.Attributes = INFO_IO_CLIENT |
                                INFO_CARD_SHARE |
                                INFO_CARD_EXCL;
client_reg.EventMask = (CS_EVENT_REGISTRATION_COMPLETE |
                        CS_EVENT_CARD_READY |
                        CS_EVENT_CARD_INSERTION |
                        CS_EVENT_CARD_REMOVAL |
                        CS_EVENT_CARD_REMOVAL_LOWP |
                        CS_EVENT_CLIENT_INFO);
client_reg.event_handler = (csfunction_t *)pcepp_event;
client_reg.event_callback_args.client_data = pps;
client_reg.Version = CS_VERSION;
client_reg.dip = dip;
(void) strcpy(client_reg.driver_name, PCEPP_NAME);

if ((ret = csx_RegisterClient(&pps->client_handle,
                              &client_reg)) != CS_SUCCESS) {
        error2text_t cft;

        cft.item = ret;
        csx_Error2Text(&cft);
        PCEPP_DEBUG((CE_CONT, "pcepp%d: attach, "
                "RegisterClient failed %s (0x%x)\n",
                instance, cft.text, ret));
        goto out;
}
```

Get logical socket number and store in soft state

```
if ((ret = csx_MapLogSocket(pps->client_handle,
                        &map_log_socket)) != CS_SUCCESS) {
        error2text_t cft;

        cft.item = ret;
        csx_Error2Text(&cft);
        PCEPP_DEBUG((CE_CONT, "pcepp%d: attach, "
                "MapLogSocket failed %s (0x%x)\n",
                instance, cft.text, ret));
}

pps->sn = map_log_socket.PhySocket;

pps->flags |= PCEPP_REGCLIENT;
```

Prepare to receive Card Services events

```
/* Set up the event handler hi-level mutex */
mutex_init(&pps->event_hi_mutex, "pps->event_hi_mutex",
    MUTEX_DRIVER, *(client_reg.iblk_cookie));

/* set up the mutex to protect the pcepp_state_t */
mutex_init(&pps->event_mutex, "pps->event_mutex",
    MUTEX_DRIVER, &pps->softint_cookie);

/* Init readywait_cv */
cv_init(&pps->readywait_cv, "pps->readywait_cv",
    CV_DRIVER, (void *)NULL);

pps->flags |= PCEPP_DID_INITMUTEX;

mutex_enter(&pps->event_mutex);

/*
 * After the RequestSocketMask call,
 *      we can start receiving events
 */
sockmask.EventMask = (CS_EVENT_CARD_INSERTION |
                        CS_EVENT_CARD_REMOVAL);
```

```
if ((ret = csx_RequestSocketMask(pps->client_handle,
                                &sockmask)) != CS_SUCCESS) {
        error2text_t cft;

        cft.item = ret;
        csx_Error2Text(&cft);
        PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: attach, "
                "RequestSocketMask failed %s (0x%x)\n",
                instance, pps->sn, cft.text, ret));
       mutex_exit(&pps->event_mutex);
        goto out;
}

pps->flags |= PCEPP_REQSOCKMASK;
```

Check card insertion

```
/*
 * If the card is inserted and this attach is triggered
 *      by an open, that open will wait until
 *      pcepp_card_insertion() is completed
 */
pps->card_event |= PCEPP_CARD_WAIT_READY;
while ((pps->card_event &
    (PCEPP_CARD_READY | PCEPP_CARD_ERROR)) == 0) {
    cv_wait(&pps->readywait_cv, &pps->event_mutex);
}

/*
 * PC Card now must be present and fully functional
 */
if (!PCEPP_CARD_IS_READY(pps)) {
    PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: attach, "
        "card not ready\n", instance, pps->sn));
    mutex_exit(&pps->event_mutex);
    goto out;
}

mutex_exit(&pps->event_mutex);

ddi_report_dev(dip);

return (DDI_SUCCESS);
```

```
     out:
        (void) pcepp_detach(dip, DDI_DETACH);
        return (DDI_FAILURE);
}
```

pcepp_detach

detach(9E) is the inverse operation to attach(9E). It is called for each device instance, receiving a command of DDI_DETACH, when the system attempts to unload a driver module. The system only calls the DDI_DETACH case of detach(9E) for a device instance if the device instance is not open. No calls to other driver entry points for that device instance occur during detach(9E), although interrupts and time-outs may occur.

The main purpose of detach(9E) is to free resources allocated by attach(9E) for the specified device and to prepare the driver for unloading.

Driver detachment can be defined as a series of steps. These steps are:

- Obtain the soft-state structure
- Call pcepp_card_removal to handle final cleanup
- Release driver socket masks
- Deregister with Card Services
- Free soft state

```
static int
pcepp_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
        int             instance;
        int             ret;
        pcepp_state_t   *pps;
```

Obtain soft-state structure

```
        instance = ddi_get_instance(dip);

       switch (cmd) {
           case DDI_DETACH:
           break;

       default:
           PCEPP_DEBUG((CE_NOTE, "pcepp%d: detach cmd 0x%x\n",
```

```
        instance, cmd));
    return (DDI_FAILURE);
}

PCEPP_DEBUG((CE_CONT, "pcepp%d: detach\n", instance));

pps = ddi_get_soft_state(pcepp_soft_state_p, instance);
if (pps == NULL) {
    PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: detach, "
        "could not get state structure\n",
        instance, pps->sn));
    return (DDI_FAILURE);
}
```

Call card removal to do final cleanup

```
/*
 * Call pcepp_card_removal to do final card cleanup
 */
 if (pps->card_event & PCEPP_CARD_READY) {
    mutex_enter(&pps->event_mutex);
    (void) pcepp_card_removal(pps);
    mutex_exit(&pps->event_mutex);
}
```

Release driver socket mask

```
/* Release driver socket mask */
if (pps->flags & PCEPP_REQSOCKMASK) {
        release_socket_mask_t   rsm;
       if ((ret = csx_ReleaseSocketMask(pps->client_handle,
                                &rsm)) != CS_SUCCESS) {
             error2text_t cft;

             cft.item = ret;
             csx_Error2Text(&cft);
            PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: attach, "
                  "ReleaseSocketMask failed %s (0x%x)\n",
                   instance, pps->sn, cft.text, ret));
        }
}
```

Deregister with Card Services

```
        /*
         * Deregister with Card Services
         *      we will stop getting events at this point.
         */
        if (pps->flags & PCEPP_REGCLIENT) {
               if ((ret = csx_DeregisterClient(pps->client_handle))
                                                  != CS_SUCCESS) {
                     error2text_t cft;

                     cft.item = ret;
                     csx_Error2Text(&cft);
                     PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: attach, "
                            "DeregisterClient failed %s (0x%x)\n",
                             instance, pps->sn, cft.text, ret));
                }
        }
```

Free mutex and condition variables

```
        /* Finally free the various mutex and condition variables */
        if (pps->flags & PCEPP_DID_INITMUTEX) {
               mutex_destroy(&pps->event_hi_mutex);
               mutex_destroy(&pps->event_mutex);
               cv_destroy(&pps->readywait_cv);
        }
```

Free soft state

```
        ddi_soft_state_free(pcepp_soft_state_p, instance);

        return (DDI_SUCCESS);
}
```

9

Card Services Routines

Card Services routines are used to allocate resources and manage events for PC Card drivers. The `pcepp` sample driver shows the use of the following routines:

- Event handler – `pcepp_event()`
- Card readiness – `pcepp_card_ready()`
- Card insertion – `pcepp_card_insertion()`
- Card configuration – `pcepp_card_configuration()`
- Card ready time-out – `pcepp_card_ready_timeout()`
- Card removal – `pcepp_card_removal()`

pcepp_event()

Card Services calls the `csx_event_handler`(9E) driver entry point to notify the driver of an event, such as card insertion, card ready, or card removal.

```
int
pcepp_event(event_t event, int priority,
                  event_callback_args_t *eca)
{
        int             retcode = CS_UNSUPPORTED_EVENT;
        pcepp_state_t   *pps = eca->client_data;
        client_info_t   *ci = &eca->client_info;

    #if defined(DEBUG)
        if (pcepp_debug_events) {
            event2text_tevent2text;

            event2text.event = event;
            csx_Event2Text(&event2text);
            cmn_err(CE_CONT, "pcepp%d.%d: %s (%s priority)\n",
                pps->instance, (int)pps->sn,
                event2text.text, (priority & CS_EVENT_PRI_HIGH) ?
                "high" : "low");
        }
    #endif

        /*
         * If the priority argument indicates CS_EVENT_PRI_HIGH,
         *      the callback handler is being called at the nexus
```

```
 *      event interrupt priority which is above lock-level.
 *      You cannot use a mutex defined at a lower priority
 *      for the high priority event.  At the same time,
 *      you do not want to use the high priority mutex
 *      when it is not necessary since it blocks out other
 *      interrupts while held.
 */
if (priority & CS_EVENT_PRI_HIGH) {
        /*
        * Use pps->event_hi_mutex mutex for a high priority
         *      event to modify the card event state.
         */
        mutex_enter(&pps->event_hi_mutex);
} else {
        /*
         * Use pps->event_mutex for a low priority event
         *      to modify the driver state structure.
         */
        mutex_enter(&pps->event_mutex);
}
```

Determine event received and act accordingly

```
/*
 * Find out which event we got and do the right thing
 */
switch (event) {

        /*
         * When the driver registers with CardServices
         *   through csx_RegisterClient() function,
         *   CardServices saves the client information
         *   and immediately returns to the client.
         *   CardServices then performs the registration
         *   in the background.  When the registration
         *   processing is completed, CS notifies
         *   the request client through pcepp_event
         *   with CS_EVENT_REGISTRATION_COMPLETE event.
         */
        case CS_EVENT_REGISTRATION_COMPLETE:
                PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: event, "
                        "Registration is completed\n",
                        pps->instance, pps->sn));
```

```
            retcode = CS_SUCCESS;
            break;

    /*
     * This event indicates that a ready/busy signal,
     *   +RDY/-BSY, has transitioned from the busy
     *   state to ready state.
     * It is available as a low priority event.
     */
    case CS_EVENT_CARD_READY:
        ASSERT(priority & CS_EVENT_PRI_LOW);
        retcode = pcepp_card_ready(pps);
        break;

    /*
     * This event indicates a PC Card has been inserted
     *   in a socket or a client has just registered
     *   for insertion events.
     * It is available as a low priority event.
     */
    case CS_EVENT_CARD_INSERTION:
        ASSERT(priority & CS_EVENT_PRI_LOW);
        retcode = pcepp_card_insertion(pps);
        break;

    /*
     * This event indicates a PC Card has been removed
     *      from a socket.
     * It is available as both a low priority
     *   and high priority event.  As a high priority
     *   event the client is expected to set the
     *   internal state the card has been removed,
     *   and that the event at low priority will be
     *   used to release any resources allocated
     *   by the client.
     */
    case CS_EVENT_CARD_REMOVAL:
    case CS_EVENT_CARD_REMOVAL_LOWP:
        if (priority & CS_EVENT_PRI_HIGH) {
            pps->card_event &= ~PCEPP_CARD_INSERTED;
            retcode = CS_SUCCESS;
        } else {
```

```
                retcode = pcepp_card_removal(pps);
            }
            break;

        /*
         * This event requests that the client return its
         *      client information data such as the driver
         *      revision level and date.
         */
        case CS_EVENT_CLIENT_INFO:
              if (GET_CLIENT_INFO_SUBSVC(ci->Attributes) ==
                                CS_CLIENT_INFO_SUBSVC_CS) {
                    ci->Attributes |= CS_CLIENT_INFO_VALID;
                        ci->Revision = PCEPP_REV_LEVEL;
                        ci->CSLevel = CS_VERSION;
                        ci->RevDate = PCEPP_REV_DATE;
                        strcpy(ci->ClientName,
                                PCEPP_CLIENT_DESCRIPTION);
                        strcpy(ci->VendorName,
                                PCEPP_VENDOR_DESCRIPTION);
                      /*
                       * Note that PC Card driver does not
                       *    need to initialize DriverName
                       *    field since Card Services will
                       *    copy the name of PC card driver
                       *    into this space after the PC
                       *    Card driver has sucessfully
                       *    processed this event.
                       */
                        retcode = CS_SUCCESS;
                } /* CS_CLIENT_INFO_SUBSVC_CS */
                break;
        default:
                retcode = CS_UNSUPPORTED_EVENT;
                break;

    } /* switch(event) */

    if (priority & CS_EVENT_PRI_HIGH) {
            mutex_exit(&pps->event_hi_mutex);
    } else {
            mutex_exit(&pps->event_mutex);
```

```
        }

        return (retcode);
}
```

pcepp_card_ready

`pcepp_card_ready()` is called from the event handler in response to a card ready event.

```
static int
pcepp_card_ready(pcepp_state_t *pps)
{
        int                ret;
        modify_config_t    modify_config;

        ASSERT(mutex_owned(&pps->event_mutex));

        #if defined(DEBUG)
        /* Debugging to verify card ready timeout handling */
        if (pcepp_debug_timeout) {
            cmn_err(CE_CONT, "pcepp%d.%d: ignoring card_ready\n",
                pps->instance, (int)pps->sn);
            return (CS_SUCCESS);
        }
        #endif

         /* Remove any pending card ready timer */
        if (pps->ready_timeout_id) {
            int id = pps->ready_timeout_id;
            pps->ready_timeout_id = 0;
            mutex_exit(&pps->event_mutex);
            untimeout(id);
            mutex_enter(&pps->event_mutex);
        }

        /*
         * If our card is just now becoming ready,
         * perform basic card configuration and initialization
         */
        if ((pps->card_event & PCEPP_CARD_READY) == 0) {
```

```
                if (pcepp_card_configuration(pps) == 0) {
                    pps->card_event |= PCEPP_CARD_ERROR;
                    if (pps->card_event & PCEPP_CARD_WAIT_READY) {
                        pps->card_event &= ~PCEPP_CARD_WAIT_READY;
                        ASSERT(mutex_owned(&pps->event_mutex));
                        cv_broadcast(&pps->readywait_cv);
                    }
                    return (CS_SUCCESS);
                }

                     /*
                      * CONF_ENABLE_IRQ_STEERING is used to enable the
                      *      PC Card interrupt, and CONF_IRQ_CHANGE_VALID
                      *      is used to request the IRQ steering enable to
                      *      be changed.
                      */
                     modify_config.Attributes =
                             (CONF_IRQ_CHANGE_VALID |
                                     CONF_ENABLE_IRQ_STEERING);

                  if ((ret = csx_ModifyConfiguration(pps->client_handle,
                                     &modify_config)) != CS_SUCCESS) {
                             error2text_t cft;

                             cft.item = ret;
                             csx_Error2Text(&cft);
                         PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: card_ready, "
                                 "ModifyConfiguration (IRQ) failed %s\n",
                                 pps->instance, (int)pps->sn, cft.text));
                             pps->card_event |= PCEPP_CARD_ERROR;
                     } /* ModifyConfiguration */

                    pps->card_event |= PCEPP_CARD_READY;
                    if (pps->card_event & PCEPP_CARD_WAIT_READY) {
                        pps->card_event &= ~PCEPP_CARD_WAIT_READY;
                        ASSERT(mutex_owned(&pps->event_mutex));
                        cv_broadcast(&pps->readywait_cv);
                    }
            }
            return (CS_SUCCESS);
}
```

pcepp_card_insertion

This function is called from the event handler in response to a card insertion event.

```
static int
pcepp_card_insertion(pcepp_state_t *pps)
{
    int           ret = CS_OUT_OF_RESOURCE;
    get_status_t  get_status;

    ASSERT(mutex_owned(&pps->event_mutex));

    PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: card_insertion, "
        pps->instance, pps->sn));

    pps->card_event &= ~(PCEPP_CARD_READY | PCEPP_CARD_ERROR);
    pps->card_event |= PCEPP_CARD_INSERTED;

    /* Get card state status */
    if ((ret = csx_GetStatus(pps->client_handle, &get_status))
                      != CS_SUCCESS) {
        error2text_t cft;

        cft.item = ret;
        csx_Error2Text(&cft);
        PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: card_insertion, "
            "csx_GetStatus failed %s (0x%x)\n",
            pps->instance, (int)pps->sn, cft.text, ret));
    }

    if ((get_status.CardState & CS_EVENT_CARD_READY) == 0) {
#if defined(DEBUG)
        /*
         * Debugging to verify card ready timeout handling
         */
        if (!(pcepp_debug_timeout || pcepp_debug_card_ready)) {
            (void) pcepp_card_ready(pps);
        }
#else
        (void) pcepp_card_ready(pps);
#endif
```

```
    }

    if ((pps->card_event &
        (PCEPP_CARD_READY | PCEPP_CARD_ERROR)) == 0) {
        /*
         * If the card isn't ready yet, we set up a
         * timeout handler to trap cards that never
         * do give us a Card Ready event, and then
         * return to wait for either the Card Ready
         * event or the timeout to expire.
         */
        PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: card_insertion, "
            "PC Card is NOT ready\n",
            pps->instance, pps->sn));
        pps->ready_timeout_id =
            timeout(pcepp_card_ready_timeout, (caddr_t)pps,
                (long)(drv_usectohz(1000000) *
                PCEPP_READY_TIMEOUT));
    }

    return (CS_SUCCESS);
}
```

pcepp_card_configuration

This function is called from the event handler to allocate system resources. Card configuration can be thought of as a series of steps. These steps include:

- Parse CIS to configure the card
- Allocate IO and IRQ resources
- Add the driver's soft interrupt handler
- Set up the client event mask
- Call `csx_RequestConfiguration`(9F) to make the resources active
- Create the minor device

Note that these resources are deallocated when the card is removed and reallocated when the card is inserted again.

```
static int
pcepp_card_configuration(pcepp_state_t *pps)
{
    int                 ret;
    char                devname[16];
    char                *dname;
    io_req_t            io_req;
    irq_req_t           irq_req;
    get_status_t        get_status;
    sockevent_t         sockevent;
    config_req_t        config_req;
    make_device_node_t make_device_node;
    devnode_desc_t      *dnd;
    pcepp_cftable_t     *cftable = NULL;
    pcepp_cftable_t     *cft;
    pcepp_cis_vars_t    *cis_vars = &pps->cis_vars;

    ASSERT(mutex_owned(&pps->event_mutex));

    PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: card_configuration\n",
        pps->instance, pps->sn));
```

Determine whether card is ready

```
    /* Get card state status */
    if ((ret = csx_GetStatus(pps->client_handle, &get_status))
                        != CS_SUCCESS) {
        error2text_t cft;

        cft.item = ret;
        csx_Error2Text(&cft);
        PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: card_insertion, "
            "csx_GetStatus failed %s (0x%x)\n",
            pps->instance, (int)pps->sn, cft.text, ret));
    }

    if ((get_status.CardState & CS_EVENT_CARD_READY) == 0) {
        PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: card_insertion, "
            "PC Card is NOT ready (0x%x)\n",
            pps->instance, pps->sn));
        return (0);
    }
```

Parse PC Card CIS information

```
/* Get PC Card CIS information */
if ((ret = pcepp_parse_cis(pps, &cftable)) != CS_SUCCESS) {
    cmn_err(CE_CONT, "pcepp%d.%d: card_insertion, "
        "unable to get CIS information (0x%x)\n",
        pps->sn, ret);
    (void) pcepp_destroy_cftable_list(&cftable);
    return (0);
 }
```

Allocate I/O resources

```
/*
 * Try to allocate IO resources; if we fail to get
 *      an IO range from the system, then we exit
 *      since there's not much we can do.
 */
cft = cftable;
while (cft) {
   io_req.BasePort1.base = cft->p.card_base1;
    io_req.NumPorts1 = cft->p.length1 + 1;
    io_req.Attributes1 = (IO_DATA_PATH_WIDTH_8 |
           WIN_ACC_NEVER_SWAP |
           WIN_ACC_STRICT_ORDER);
    io_req.BasePort2.base = 0;
    io_req.NumPorts2 = 0;
    io_req.Attributes2 = 0;
    io_req.IOAddrLines = cft->p.addr_lines;

   if ((ret = csx_RequestIO(pps->client_handle, &io_req))
                  == CS_SUCCESS) {
       /*
        * We found a good IO range, so save the
        *information and break out of this loop
        */

       cis_vars->card_base1 = cft->p.card_base1;
       cis_vars->length1 = cft->p.length1;
       cis_vars->card_base2 = cft->p.card_base2;
       cis_vars->length2 = cft->p.length2;

       cis_vars->addr_lines = cft->p.addr_lines;
```

```
            cis_vars->card_vcc = cft->p.card_vcc;
            cis_vars->card_vpp1 = cft->p.card_vpp1;
            cis_vars->card_vpp2 = cft->p.card_vpp2;
            cis_vars->pin = cft->p.pin;
            cis_vars->config_index = cft->p.config_index;
            break;
        } /* RequestIO */

        cft = cft->next;
    } /* while (cft) */

    /*
     * Now destroy the config table entries list since
     *      we don't need it anymore.
     */
    pcepp_destroy_cftable_list(&cftable);

    /*
     * If we weren't able to get an IO range, then
     *      report that to the user and return.
     */

   if (!cft) {
            cmn_err(CE_CONT, "pcepp%d.%d: card_insertion, "
                    "socket %d unable to get IO range\n",
                    (int)pps->sn);
            return (0);
    }

    /* Normal eight contiguous registers */
    pps->card_handle = io_req.BasePort1.handle;

    pps->flags |= PCEPP_REQUESTIO;
```

Allocate an IRQ

```
    /*
     * Allocate an IRQ
     */
    irq_req.Attributes = IRQ_TYPE_EXCLUSIVE;
    irq_req.irq_handler = (csfunction_t *)pcepp_intr;
    irq_req.irq_handler_arg = (caddr_t)pps;
```

```
if ((ret = csx_RequestIRQ(pps->client_handle, &irq_req))
                                        != CS_SUCCESS) {
        error2text_t cft;

        cft.item = ret;
        csx_Error2Text(&cft);
       PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: card_insertion, "
                "RequestIRQ failed %s (0x%x)\n",
              pps->instance, (int)pps->sn, cft.text, ret));
        return (0);
} /* RequestIRQ */
```

Initialize the Card Register mutex

```
/*
 * Initialize the interrupt mutex for protecting
 *      the card registers.
 */

mutex_init(&pps->irq_mutex, "pps->irq_mutex", MUTEX_DRIVER,
           *(irq_req.iblk_cookie));
pps->flags |= PCEPP_REQUESTIRQ;
```

Add soft interrupt handler

```
/*
 * Add low level soft interrupt handler
 *  It is used in pcepp_intr() when ddi_trigger_softintr()
 *  is called to start pcepp_softintr() for transferring
 *  next data.
 */
ret = ddi_add_softintr(pps->dip, PCEPP_SOFT_PREF,
       &pps->softint_id, &pps->softint_cookie,
       (ddi_idevice_cookie_t *)0, pcepp_softintr,
       (caddr_t)pps);
if (ret != DDI_SUCCESS) {
    PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: card_insertion, "
        "ddi_add_softintr failed\n",
        pps->instance, pps->sn));
    return (0);
}

pps->flags |= PCEPP_SOFTINTROK;
```

Prepare for card ready and other registered events

```
/*
 * Set up the client event mask to give us card ready
 *      events as well as what other events we have
 *      already registered for.
 * Note that since we set the global event mask in the call
 *      to RegisterClient in pcepp_attach, we don't have to
 *      duplicate those events in this event mask.
 */

sockevent.Attributes = CONF_EVENT_MASK_CLIENT;
if ((ret = csx_GetEventMask(pps->client_handle,
                        &sockevent)) != CS_SUCCESS) {
        error2text_t cft;

        cft.item = ret;
        csx_Error2Text(&cft);
       PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: card_insertion, "
                "GetEventMask failed %s (0x%x)\n",
              pps->instance, (int)pps->sn, cft.text, ret));
        return (0);
} /* GetEventMask */

sockevent.EventMask |= CS_EVENT_CARD_READY;
if ((ret = csx_SetEventMask(pps->client_handle, &sockevent))
                                        != CS_SUCCESS) {
        error2text_t cft;

        cft.item = ret;
        csx_Error2Text(&cft);
       PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: card_insertion, "
                "SetEventMask failed %s (0x%x)\n",
              pps->instance, (int)pps->sn, cft.text, ret));
        return (0);
} /* SetEventMask */
```

Configure the PC Card

```
/*
 * Configure the PC Card
 */
config_req.Attributes = 0;
```

```
        config_req.Vcc = cis_vars->card_vcc;
        config_req.Vpp1 = cis_vars->card_vpp1;
        config_req.Vpp2 = cis_vars->card_vpp2;
        config_req.IntType = SOCKET_INTERFACE_MEMORY_AND_IO;
        config_req.ConfigBase = cis_vars->config_base;
        config_req.Status = 0;
        config_req.Pin = cis_vars->pin;
        config_req.Copy = 0;
        config_req.ConfigIndex = cis_vars->config_index;
        config_req.Present = cis_vars->present;

        if ((ret = csx_RequestConfiguration(pps->client_handle,
                                &config_req)) != CS_SUCCESS) {
                error2text_t cft;

                cft.item = ret;
                csx_Error2Text(&cft);
               PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: card_insertion, "
                         "RequestConfiguration failed %s (0x%x)\n",
                       pps->instance, (int)pps->sn, cft.text, ret));
                return (0);
        } /* RequestConfiguration */

        pps->flags |= PCEPP_REQUESTCONFIG;

    #if defined(DEBUG)
     if (pcepp_debug_cis) {
         pcepp_display_card_config(pps);
     }
     #endif
```

Create minor devices for this instance

```
        /*
         * Create the minor devices for this instance
         *      The minor number is the socket number
         */
        dname = devname;
        sprintf(dname, "pcepp");

        make_device_node.Action = CREATE_DEVICE_NODE;
        make_device_node.NumDevNodes = 1;
```

```
        make_device_node.devnode_desc =
                kmem_zalloc(sizeof (struct devnode_desc) *
                        make_device_node.NumDevNodes, KM_SLEEP);

        dnd = &make_device_node.devnode_desc[0];
        dnd->name = dname;
        dnd->spec_type = S_IFCHR;
        dnd->minor_num = pps->sn;
        dnd->node_type = PCEPP_NT_PARALLEL;

        if ((ret = csx_MakeDeviceNode(pps->client_handle,
                              &make_device_node)) != CS_SUCCESS) {
                error2text_t cft;

                cft.item = ret;
                csx_Error2Text(&cft);
               PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: card_insertion, "
                        "MakeDeviceNode failed %s (0x%x)\n",
                      pps->instance, (int)pps->sn, cft.text, ret));
                return (0);
        } /* SetEventMask */

        /*
         * We don't need this structure anymore since we've
         *   created the devices.  If we need to keep track
         *   of the devices that we've created for some reason,
         *   then you'll want to keep this structure and the
         *   make_device_node_t structure around in a per-instance
         *   data area.
         */
        kmem_free(make_device_node.devnode_desc,
                        sizeof (struct devnode_desc) *
                                make_device_node.NumDevNodes);
        make_device_node.devnode_desc = NULL;

        pps->flags |= PCEPP_MAKEDEVICENODE;
        return (1);
}
```

pcepp_card_ready_timeout()

This function detects a card that fails to become ready within a reasonable amount of time.

```
static void
pcepp_card_ready_timeout(pcepp_state_t *pps)
{
    PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: card ready timeout\n",
                   pps->instance, pps->sn));

    mutex_enter(&pps->event_mutex);
    if (pps->ready_timeout_id) {
        pps->ready_timeout_id = 0;
        pps->card_event |= PCEPP_CARD_READY | PCEPP_CARD_ERROR;
    }
    if (pps->card_event & PCEPP_CARD_WAIT_READY) {
        pps->card_event &= ~PCEPP_CARD_WAIT_READY;
        cv_broadcast(&pps->readywait_cv);
    }
    mutex_exit(&pps->event_mutex);
}
```

pcepp_card_removal()

This function is called from the event handler in response to a card removal event. Card removal can be broken down into a series of steps. These steps are:

- Set up the client event mask so that the driver will not receive the card ready events
- Call `csx_ModifyConfiguration()` to turn off the power
- Release the card configuration
- Release allocated IRQ resources
- Release allocated IO resources
- Remove the minor devices

```
static int
pcepp_card_removal(pcepp_state_t *pps)
{
    int             ret;
    sockevent_t     sockevent;
```

```
ASSERT(mutex_owned(&pps->event_mutex));

PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: card_removal\n",
    pps->instance, pps->sn));

/*
 * Remove any pending card ready timer
 */
if (pps->ready_timeout_id) {
    int id = pps->ready_timeout_id;
    pps->ready_timeout_id = 0;
    mutex_exit(&pps->event_mutex);
    untimeout(id);
    mutex_enter(&pps->event_mutex);
}

/*
* First, grab the I/O mutex; this will cause us to
 * spin until any pending I/O interrupts have completed.
 * Dropping the Card Inserted state (already occurred)
 * ensures that any interrupt triggered after
 * we get the locks will return without initiating
 * any activity on the device or even acquiring the I/O
 * mutexes.  This is important since our device may be
 * sharing the interrupt line with other devices and
 * we don't want to stall or interfere with those
 * devices in any way.  Our device cannot be generating
 * any interrupts since it isn't there anymore.
 */
pps->card_event &= ~PCEPP_CARD_INSERTED;

if (pps->flags & PCEPP_REQUESTIRQ) {
    mutex_enter(&pps->irq_mutex);
}

pps->card_event &= ~(PCEPP_CARD_READY | PCEPP_CARD_ERROR |
   PCEPP_CARD_WAIT_READY);

/*
 * Now that we are sure our interrupt handlers won't
 * attempt to initiate any activity, we can continue
 * with freeing this card's I/O resources.
```

```
 */
if (pps->flags & PCEPP_REQUESTIRQ) {
    mutex_exit(&pps->irq_mutex);
}
```

Set up client event mask

```
/*
 * Set up the client event mask to NOT give us card ready
 *      events; we will still receive other events we have
 *      registered for.
 * Note that since we set the global event mask in call
 *      to RegisterClient in pcepp_attach, we don't have
 *      to duplicate those events in this event mask.
 */
sockevent.Attributes = CONF_EVENT_MASK_CLIENT;
if ((ret = csx_GetEventMask(pps->client_handle,
                        &sockevent)) != CS_SUCCESS) {
        error2text_t cft;

        cft.item = ret;
        csx_Error2Text(&cft);
        PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: card_removal, "
                "GetEventMask failed %s (0x%x)\n",
              pps->instance, (int)pps->sn, cft.text, ret));
       pps->card_event &= ~PCEPP_CARD_BUSY;
       pps->flags &= ~PCEPP_ISOPEN;
        return (ret);
} /* GetEventMask */

sockevent.EventMask &= ~CS_EVENT_CARD_READY;
if ((ret = csx_SetEventMask(pps->client_handle, &sockevent))
                    != CS_SUCCESS) {
        error2text_t cft;

        cft.item = ret;
        csx_Error2Text(&cft);
        PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: card_removal, "
                "SetEventMask failed %s (0x%x)\n",
              pps->instance, (int)pps->sn, cft.text, ret));
       pps->card_event &= ~PCEPP_CARD_BUSY;
       pps->flags &= ~PCEPP_ISOPEN;
        return (ret);
```

```
} /* SetEventMask */

if (pps->flags & PCEPP_REQUESTCONFIG) {
        modify_config_t modify_config;
        release_config_t release_config;
```

Turn off VPP1 and VPP2

```
        /*
         * First, turn off VPP1 and VPP2 since we
         *      don't need them anymore. This will
         *      take care of the case of the driver
         *      being unloaded but the card still
         *      being present.
         */
        modify_config.Attributes =
                        (CONF_VPP1_CHANGE_VALID |
                                CONF_VPP2_CHANGE_VALID);
        modify_config.Vpp1 = 0;
        modify_config.Vpp2 = 0;

        ret = csx_ModifyConfiguration(pps->client_handle,
                                        &modify_config);
        if ((ret != CS_NO_CARD) && (ret != CS_SUCCESS)) {
                error2text_t cft;

                cft.item = ret;
                csx_Error2Text(&cft);
                PCEPP_DEBUG((CE_CONT,
                        "pcepp%d.%d: card_removal, "
                        "ModifyConfiguration "
                        "(Vpp1/Vpp2) failed %s (0x%x)\n",
                        pps->instance, (int)pps->sn,
                        cft.text, ret));
        } /* ModifyConfig != (CS_NO_CARD || CS_SUCCESS) */
```

Release card configuration

```
        if ((ret = csx_ReleaseConfiguration(pps->client_handle,
                        &release_config)) != CS_SUCCESS) {
                error2text_t cft;
```

```
                        cft.item = ret;
                        csx_Error2Text(&cft);
                        PCEPP_DEBUG((CE_CONT,
                                "pcepp%d.%d: card_removal, "
                                "ReleaseConfiguration "
                                "failed %s (0x%x)\n",
                                pps->instance, (int)pps->sn,
                                cft.text, ret));
                } /* ReleaseConfiguration */
                pps->flags &= ~PCEPP_REQUESTCONFIG;
        }  /* PCEPP_REQUESTCONFIG */
```

Release allocated IRQ resources

```
        /*
         * Unregister the softinterrupt handler
         */
        if (pps->flags & PCEPP_SOFTINTROK) {
            ddi_remove_softintr(pps->softint_id);
            pps->flags &= ~PCEPP_SOFTINTROK;
        }

        if (pps->flags & PCEPP_REQUESTIRQ) {
                irq_req_t       irq_req;

                if ((ret = csx_ReleaseIRQ(pps->client_handle,
                                        &irq_req)) != CS_SUCCESS) {
                        error2text_t cft;

                        cft.item = ret;
                        csx_Error2Text(&cft);
                        PCEPP_DEBUG((CE_CONT,
                                "pcepp%d.%d: card_removal, "
                                "ReleaseIRQ failed %s (0x%x)\n",
                                pps->instance, (int)pps->sn,
                                cft.text, ret));
                } /* ReleaseIRQ */

                /*
                 * destroy mutex
                 */
                mutex_destroy(&pps->irq_mutex);
                pps->flags &= ~PCEPP_REQUESTIRQ;
```

```
} /* PCEPP_REQUESTIRQ */
```

Release allocated I/O resources

```
if (pps->flags & PCEPP_REQUESTIO) {
        io_req_t        io_req;

        if ((ret = csx_ReleaseIO(pps->client_handle,
                                &io_req)) != CS_SUCCESS) {
                error2text_t cft;

                cft.item = ret;
                csx_Error2Text(&cft);
                PCEPP_DEBUG((CE_CONT,
                        "pcepp%d.%d: card_removal, "
                        "ReleaseIO failed %s (0x%x)\n",
                        pps->instance, (int)pps->sn,
                        cft.text, ret));
        } /* ReleaseIO */
        pps->flags &= ~PCEPP_REQUESTIO;
} /* PCEPP_REQUESTIO */
```

Remove minor devices

```
if (pps->flags & PCEPP_MAKEDEVICENODE) {
     remove_device_node_t remove_device_node;

     remove_device_node.Action = REMOVAL_ALL_DEVICE_NODES;
     remove_device_node.NumDevNodes = 0;

        if ((ret = csx_RemoveDeviceNode(pps->client_handle,
                     &remove_device_node)) != CS_SUCCESS) {
                error2text_t cft;

                cft.item = ret;
                csx_Error2Text(&cft);
                PCEPP_DEBUG((CE_CONT,
                        "pcepp%d.%d: card_removal, "
                     "RemoveDeviceNode failed %s (0x%x)\n",
                        pps->instance, (int)pps->sn,
                        cft.text, ret));
        } /* MakeDeviceNode */
        pps->flags &= ~PCEPP_MAKEDEVICENODE;
```

```
        } /* PCEPP_MAKEDEVICENODE */

        pps->card_event &= ~PCEPP_CARD_BUSY;
        pps->flags &= ~PCEPP_ISOPEN;

        return (CS_SUCCESS);
}
```

CIS Configuration Information Routines

The CIS configuration routines parse the CIS tuples and include support routines for tuple table configuration management. The following `pcepp` sample driver routines show the use of Card Services to parse CIS information.

- `pcepp_parse_cis()`
- `pcepp_display_cftable_list()`
- `pcepp_display_card_config()`
- `pcepp_destroy_cftable_list()`
- `pcepp_set_cftable_desireability()`
- `pcepp_sort_cftable_list()`
- `pcepp_swap_cft()`

pcepp_parse_cis()

This function searches the CIS for device data contained in CISTPL structures to configure the card. The function uses `csx_GetFIrstTuple()` and `csx_GetNextTuple()` to find the desired tuple and uses the following CIS parser functions to process the tuple:

- `csx_Parse_CISTPL_CONFIG()` – Parses the configuration tuple
- `csx_Parse_CISTPL_VERS_1()` – Parses the version number tuple
- `csx_Parse_CISTPL_MANFID()` – Parses the manufacturer information tuple
- `csx_Parse_CISTPL_CFTABLE_ENTRY()` – Parses the configuration table entry tuple
- `csx_Parse_CISTPL_FUNCID()` – Parses the function identification tuple
- `csx_Parse_CISTPL_FUNCE()` – Parses the function extension tuple

```
static int
pcepp_parse_cis(pcepp_state_t *pps, pcepp_cftable_t **cftable)
{
        int                             i;
        int                             ret;
        int                             last_config_index;
        int                             default_cftable;
        tuple_t                         tuple;
        cistpl_cftable_entry_t          cistpl_cftable_entry;
       cistpl_cftable_entry_io_t     *io =&cistpl_cftable_entry.io;
        pcepp_cis_vars_t                *cis_vars = &pps->cis_vars;
        pcepp_cftable_t                 *cft, *dcft, *ocft, defcft;
        cistpl_config_t                 cistpl_config;
        cistpl_vers_1_t                 cistpl_vers_1;
        cistpl_manfid_t                 cistpl_manfid;

        PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: parse_cis\n",
                                        pps->instance, pps->sn));

        /*
         * Clear the CIS saving information structure
         */
        bzero(cis_vars, sizeof (pcepp_cis_vars_t));
```

Parse CISTPL_CONFIG Tuple

```
        /*
         * CISTPL_CONFIG processing.
         *      Search for the first config tuple so that we
         *      can get a pointer to the card's configuration
         *      registers. If this tuple is not found, there's
         *      no point in searching for anything else.
         */
        bzero(&tuple, sizeof (tuple));
        tuple.DesiredTuple = CISTPL_CONFIG;
        if ((ret = csx_GetFirstTuple(pps->client_handle, &tuple))
                                                != CS_SUCCESS) {
                cmn_err(CE_CONT, "pcepp%d.%d: parse_cis, "
                        "CISTPL_CONFIG tuple not found (0x%x)\n",
                        pps->instance, pps->sn, ret);
                return (ret);
        }
```

```
bzero(&cistpl_config, sizeof (struct cistpl_config_t));

if ((ret = csx_Parse_CISTPL_CONFIG(pps->client_handle,
                                   &tuple,
                        &cistpl_config)) != CS_SUCCESS) {
        cmn_err(CE_CONT, "pcepp%d.%d: parse_cis, "
                "csx_Parse_CISTPL_CONFIG failed (0x%x)\n",
                pps->instance, pps->sn, ret);
        return (ret);
} else {
        /*
         * This is the last CISTPL_CFTABLE_ENTRY
         *      tuple index that we need to look at.
         */
        last_config_index = cistpl_config.last;

        if (cistpl_config.nr) {
                cis_vars->config_base = cistpl_config.base;
                cis_vars->present = cistpl_config.present;
        } else {
                cmn_err(CE_CONT, "pcepp%d.%d: parse_cis, "
                        "CISTPL_CONFIG no configuration "
                        "registers found\n", pps->sn);
                return (CS_BAD_CIS);
        }
}
```

Parse CISTPL_VERS_1 Tuple

```
/*
 * CISTPL_VERS_1 processing.
 *      Get major/minor version and
 *      product information strings
 */
bzero(&tuple, sizeof (tuple));
tuple.DesiredTuple = CISTPL_VERS_1;
if ((ret = csx_GetFirstTuple(pps->client_handle, &tuple))
                                        != CS_SUCCESS) {
        cmn_err(CE_CONT, "pcepp%d.%d: parse_cis, "
                "CISTPL_VERS_1 tuple not found (0x%x)\n",
                pps->instance, pps->sn, ret);
        return (ret);
}
```

```
bzero(&cistpl_vers_1, sizeof (struct cistpl_vers_1_t));

if ((ret = csx_Parse_CISTPL_VERS_1(pps->client_handle,
                &tuple, &cistpl_vers_1)) != CS_SUCCESS) {
        cmn_err(CE_CONT, "pcepp%d.%d: parse_cis, "
                "csx_Parse_CISTPL_VERS_1 failed (0x%x)\n",
                pps->instance, pps->sn, ret);
        return (ret);
} else {
        cis_vars->major_revision = cistpl_vers_1.major;
        cis_vars->minor_revision = cistpl_vers_1.minor;
        cis_vars->nstring = cistpl_vers_1.ns;
        for (i = 0; i < cistpl_vers_1.ns; i++) {
                strcpy(cis_vars->prod_strings[i],
                                cistpl_vers_1.pi[i]);
        }
}
```

Parse CISTPL_MANFID Tuple

```
/*
 * CISTPL_MANFID processing.
 *      Get PCMCIA PC Card manufacturer code and
 *      manufacturer information
 */
bzero(&tuple, sizeof (tuple));
tuple.DesiredTuple = CISTPL_MANFID;
if ((ret = csx_GetFirstTuple(pps->client_handle, &tuple))
                                        != CS_SUCCESS) {
        cmn_err(CE_CONT, "pcepp%d.%d: parse_cis, "
                "CISTPL_MANFID tuple not found (0x%x)\n",
                pps->instance, pps->sn, ret);
        return (ret);
}

bzero(&cistpl_manfid, sizeof (struct cistpl_manfid_t));

if ((ret = csx_Parse_CISTPL_MANFID(pps->client_handle,
                &tuple, &cistpl_manfid)) != CS_SUCCESS) {
        cmn_err(CE_CONT, "pcepp%d.%d: parse_cis, "
                "csx_Parse_CISTPL_MANFID failed (0x%x)\n",
                pps->instance, pps->sn, ret);
```

```
                return (ret);
        } else {
                cis_vars->manufacturer_id = cistpl_manfid.manf;
                cis_vars->card_id = cistpl_manfid.card;
        }
```

Parse CISTPL_CFTABLE_ENTRY Tuple

```
        /*
         * CISTPL_CFTABLE_ENTRY processing.
         */
        bzero(&tuple, sizeof (tuple));
        tuple.DesiredTuple = CISTPL_CFTABLE_ENTRY;
        if ((ret = csx_GetFirstTuple(pps->client_handle, &tuple))
                                                != CS_SUCCESS) {
                cmn_err(CE_CONT, "pcepp%d.%d: parse_cis, "
                    "CISTPL_CFTABLE_ENTRY tuple not found (0x%x)\n",
                         pps->instance, pps->sn, ret);
                return (ret);
        }

        /* Initialize variable pcepp_cftable_t */
        dcft = &defcft;

        /* Clear the default values */
        bzero(dcft, sizeof (pcepp_cftable_t));

        /*
         * Look through the CIS for all CISTPL_CFTABLE_ENTRY
         *      tuples and store them in a list. Stop searching
         *      if we find a tuple with a config index equal to
         *      the "last_config_index" value that we got from
         *      the CISTPL_CONFIG tuple.
         */
        do {

                bzero(&cistpl_cftable_entry,
                        sizeof (struct cistpl_cftable_entry_t));

                if ((ret = csx_Parse_CISTPL_CFTABLE_ENTRY(
                                pps->client_handle, &tuple,
                           &cistpl_cftable_entry)) != CS_SUCCESS) {
                        cmn_err(CE_CONT, "pcepp%d.%d: parse_cis, "
```

```
                "csx_Parse_CISTPL_CFTABLE_ENTRY "
                "failed (0x%x)\n",
                pps->instance, pps->sn, ret);
        return (ret);
}

/*
 * This particular application example does not
 *      support any CISTPL_CFTABLE_ENTRY tuple
 *      that does not have TPCE_IO information
 */
if ((cistpl_cftable_entry.flags &
                CISTPL_CFTABLE_TPCE_FS_IO) == 0) {
        continue;
}

ocft = kmem_zalloc(sizeof (pcepp_cftable_t),
                             KM_SLEEP);

if (!*cftable) {
        *cftable = ocft;
        cft = ocft;
        cft->prev = NULL;
} else {
        cft->next = ocft;
        cft->next->prev = cft;
        cft = cft->next;
}

cft->next = NULL;

/*
 * See if this tuple has default values that we
 *   should save. If so, copy the default values
 *   that we've seen so far into the current
 *   cftable structure.
 */
if (cistpl_cftable_entry.flags &
                     CISTPL_CFTABLE_TPCE_DEFAULT) {
        default_cftable = 1;
} else {
        default_cftable = 0;
```

```
}

bcopy(&dcft->p, (caddr_t)&cft->p,
        sizeof (pcepp_cftable_params_t));

cft->p.config_index = cistpl_cftable_entry.index;


/*
 * Starting to process:
 *      CISTPL_CFTABLE_TPCE_IF
 *      CISTPL_CFTABLE_TPCE_FS_PWR
 *              CISTPL_CFTABLE_TPCE_FS_PWR_VCC
 *              CISTPL_CFTABLE_TPCE_FS_PWR_VPP1
 *              CISTPL_CFTABLE_TPCE_FS_PWR_VPP2
 *      CISTPL_CFTABLE_TPCE_FS_IO
 *              CISTPL_CFTABLE_TPCE_FS_IO_RANGE
 */

if (cistpl_cftable_entry.flags &
                        CISTPL_CFTABLE_TPCE_IF) {
        cft->p.pin = cistpl_cftable_entry.pin;
        if (default_cftable) {
            dcft->p.pin = cistpl_cftable_entry.pin;
        }
}

if (cistpl_cftable_entry.flags &
                      CISTPL_CFTABLE_TPCE_FS_PWR) {
        struct cistpl_cftable_entry_pd_t *pd;

        pd = &cistpl_cftable_entry.pd;

        if (pd->flags &
                CISTPL_CFTABLE_TPCE_FS_PWR_VCC) {
                if (pd->pd_vcc.nomV_flags &
                        CISTPL_CFTABLE_PD_EXISTS) {
                        cft->p.card_vcc =
                                pd->pd_vcc.nomV;
                        if (default_cftable) {
                                dcft->p.card_vcc =
                                   pd->pd_vcc.nomV;
```

```
                        }
                } /* CISTPL_CFTABLE_PD_EXISTS */
        } /* CISTPL_CFTABLE_TPCE_FS_PWR_VCC */

        if (pd->flags &
                CISTPL_CFTABLE_TPCE_FS_PWR_VPP1) {
                if (pd->pd_vpp1.nomV_flags &
                        CISTPL_CFTABLE_PD_EXISTS) {
                        cft->p.card_vpp1 =
                                pd->pd_vpp1.nomV;
                        if (default_cftable) {
                                dcft->p.card_vpp1 =
                                  pd->pd_vpp1.nomV;
                        }
                } /* CISTPL_CFTABLE_PD_EXISTS */
        } /* CISTPL_CFTABLE_TPCE_FS_PWR_VPP1 */

        if (pd->flags &
                CISTPL_CFTABLE_TPCE_FS_PWR_VPP2) {
                if (pd->pd_vpp2.nomV_flags &
                        CISTPL_CFTABLE_PD_EXISTS) {
                        cft->p.card_vpp2 =
                                pd->pd_vpp2.nomV;
                        if (default_cftable) {
                                dcft->p.card_vpp2 =
                                  pd->pd_vpp2.nomV;
                        }
                } /* CISTPL_CFTABLE_PD_EXISTS */
        } /* CISTPL_CFTABLE_TPCE_FS_PWR_VPP2 */

} /* CISTPL_CFTABLE_TPCE_FS_PWR */


if (cistpl_cftable_entry.flags &
                        CISTPL_CFTABLE_TPCE_FS_IO) {

        cft->p.addr_lines = io->addr_lines;

        if (default_cftable) {
               dcft->p.addr_lines = io->addr_lines;
        }
```

```
if ((cistpl_cftable_entry.io.flags &
   CISTPL_CFTABLE_TPCE_FS_IO_RANGE) == 0) {
        /*
         * If there's no IO ranges for
         *  this configuration, then we
         *  need to calculate the length
         *  of the IO space by using the
         *  number of IO address
         *  lines value. i.e. 4 lines
         *  means 16 addresses
         */
        cft->p.length1 =
                (1 << cft->p.addr_lines);
        /*
         * cft->p.length1 must be
         *   decremented by 1 since it is
         *   incremented by 1 when it
         *   is used for NumPorts1 in
         *   CardService(RequestIO)
         *   See pcepp_card_insertion()
         */
        cft->p.length1--;
        if (default_cftable) {
          dcft->p.length1 = cft->p.length1;
        }
} else {
        /*
         * Get multiple IO address bases
         *  and IO register lengths
         */
     cft->p.card_base1 = io->range[0].addr;
      cft->p.length1 = io->range[0].length;

        if (io->ranges == 2) {
            cft->p.card_base2 =
                         io->range[1].addr;
            cft->p.length2 =
                       io->range[1].length;
        } else {
            cft->p.card_base2 = 0;
            cft->p.length2 = 0;
        }
```

```
                                        if (default_cftable) {
                                            dcft->p.card_base1 =
                                                        io->range[0].addr;
                                            dcft->p.length1 =
                                                      io->range[0].length;

                                            if (io->ranges == 2) {
                                                dcft->p.card_base2 =
                                                        io->range[1].addr;
                                                dcft->p.length2 =
                                                       io->range[1].length;
                                            } else {
                                                dcft->p.card_base2 = 0;
                                                dcft->p.length2 = 0;
                                            }
                                        }

                              }  /* if (CISTPL_CFTABLE_TPCE_FS_IO_RANGE) */

                        } /* CISTPL_CFTABLE_TPCE_FS_IO */

                        cis_vars->n_cistpl_cftable_entry++;

                    (void) pcepp_set_cftable_desireability(cft);

            } while ((cistpl_cftable_entry.index != last_config_index) &&
                    ((ret = csx_GetNextTuple(pps->client_handle, &tuple))
                                                 == CS_SUCCESS));

          /*
           * Now we've got all of the possible configurations,
           *      so sort the list of configurations.
           */
          (void) pcepp_sort_cftable_list(cftable);


#if defined(DEBUG)
      cmn_err(CE_CONT, "DEBUG:  socket %d --- sorted cftable ---\n",
                                                          pps->sn);
        (void) pcepp_display_cftable_list(*cftable);
#endif
```

```
/*
 * If GetNextTuple gave us any error code other than
 *      CS_SUCCESS or CS_NO_MORE_ITEMS, this is a
 *      CIS parser error
 */
if ((ret != CS_SUCCESS) && (ret != CS_NO_MORE_ITEMS)) {
        cmn_err(CE_CONT, "pcepp%d.%d: parse_cis, "
                "CIS parser error (0x%x)\n",
                pps->instance, pps->sn, ret);
        return (ret);
}
```

Parse CISTPL_FUNCID and CISTPL_FUNCE Tuples

```
/*
 * CISTPL_FUNCID and CISTPL_FUNCE processing.
 *   We should really check to be sure that the
 *   CISTPL_FUNCID tuple we see is for a parallel port.
 *   The assumption is made that by the time we get here,
 *   Card Services would have already validated this.
 *   Also, we should only search for one CISTPL_FUNCID
 *   tuple, since a CIS is only allowed to have one
 *   CISTPL_FUNCID tuple per function.
 */
bzero(&tuple, sizeof (tuple));
tuple.DesiredTuple = CISTPL_FUNCID;
if ((ret = csx_GetFirstTuple(pps->client_handle, &tuple))
                                        != CS_SUCCESS) {
        cmn_err(CE_CONT, "pcepp%d.%d: parse_cis, "
                "CISTPL_FUNCID tuple not found (0x%x)\n",
                pps->instance, pps->sn, ret);
        return (ret);
}

do {
        cistpl_funcid_t         cistpl_funcid;
        cistpl_funce_t          cistpl_funce;


        bzero((caddr_t)&cistpl_funcid,
                        sizeof (struct cistpl_funcid_t));
```

```
if ((ret = csx_Parse_CISTPL_FUNCID(pps->client_handle,
           &tuple, &cistpl_funcid)) != CS_SUCCESS) {
           cmn_err(CE_CONT, "pcepp%d.%d: parse_cis, "
                   "csx_Parse_CISTPL_FUNCID "
                   "failed (0x%x)\n",
                   pps->instance, pps->sn, ret);
           return (ret);
   }

   /*
    * Search for any CISTPL_FUNCE tuples.
    */
   tuple.DesiredTuple = CISTPL_FUNCE;
   while ((ret = csx_GetNextTuple(pps->client_handle,
                          &tuple)) == CS_SUCCESS) {

           bzero((caddr_t)&cistpl_funce,
                          sizeof (cistpl_funce_t));

           if ((ret = csx_Parse_CISTPL_FUNCE(
                         pps->client_handle, &tuple,
                          &cistpl_funce,
                          cistpl_funcid.function))
                                  != CS_SUCCESS) {
                   cmn_err(CE_CONT,
                          "pcepp%d.%d: parse_cis, "
                          "csx_Parse_CISTPL_FUNCE "
                          "failed (0x%x)\n",
                        pps->instance, pps->sn, ret);
                   return (ret);
           }

           /*
            * Note that there is no CISTPL_FUNCE
            *  tuple for this sample parallel card.
            * But if a card has the CISTPL_FUNCE
           *  tuple, this is where it can be processed.
            */

   } /* while (GetNextTuple) */

   tuple.DesiredTuple = CISTPL_FUNCID;
```

```
                cis_vars->n_cistpl_funcid++;

        } while ((ret = csx_GetNextTuple(pps->client_handle, &tuple))
                                                        == CS_SUCCESS);

        return (CS_SUCCESS);
}
```

pcepp_display_cftable_list()

This utility routine prints the configuration table on the console.

```
void
pcepp_display_cftable_list(pcepp_cftable_t *cft)
{
        while (cft) {
                cmn_err(CE_CONT,
                        "\nDEBUG: pcepp_display_cftable_list\n");
                cmn_err(CE_CONT, "    desireability: 0x%x\n",
                        (int)cft->desireability);
                cmn_err(CE_CONT, "    config_index: 0x%x\n",
                        (int)cft->p.config_index);
                cmn_err(CE_CONT, "    addr_lines: 0x%x\n",
                        (int)cft->p.addr_lines);
                cmn_err(CE_CONT, "    pin: 0x%x\n",
                        (int)cft->p.pin);
                cmn_err(CE_CONT, "    card_vcc: %d\n",
                        (int)cft->p.card_vcc);
                cmn_err(CE_CONT, "    card_vpp1: %d\n",
                        (int)cft->p.card_vpp1);
                cmn_err(CE_CONT, "    card_vpp2: %d\n",
                        (int)cft->p.card_vpp2);

                cmn_err(CE_CONT, "    card_base1: 0x%x\n",
                        (int)cft->p.card_base1);
                cmn_err(CE_CONT, "    length1: 0x%x\n",
                        (int)cft->p.length1);
                cmn_err(CE_CONT, "    card_base2: 0x%x\n",
                        (int)cft->p.card_base2);
                cmn_err(CE_CONT, "    length2: 0x%x\n",
                        (int)cft->p.length2);
```

```
                cft = cft->next;
        }  /* while (cft) */
}
```

pcepp_display_card_config()

This utility routine prints the card configuration on the console.

```
void
pcepp_display_card_config(pcepp_state_t *pps)
{
     int                     i;
     pcepp_cis_vars_t        *cis_vars = &pps->cis_vars;

     cmn_err(CE_CONT, "\nDEBUG: "
                "pcepp_display_card_config Socket %d\n", pps->sn);

     cmn_err(CE_CONT, "    flags: 0x%x\n", (int)cis_vars->flags);
     cmn_err(CE_CONT, "    config_base: 0x%x\n",
                        (int)cis_vars->config_base);
    cmn_err(CE_CONT, "   present: 0x%x\n", (int)cis_vars->present);
     cmn_err(CE_CONT, "    major_revision: 0x%x\n",
                        (int)cis_vars->major_revision);
     cmn_err(CE_CONT, "    minor_revision: 0x%x\n",
                        (int)cis_vars->minor_revision);
     cmn_err(CE_CONT, "    prod_strings:\n");
                for (i = 0; i < cis_vars->nstring; i++) {
                        cmn_err(CE_CONT, "\t%d: [%s]\n",
                                i, cis_vars->prod_strings[i]);
                }
     cmn_err(CE_CONT, "    manufacturer_id: 0x%x\n",
                        (int)cis_vars->manufacturer_id);
    cmn_err(CE_CONT, "   card_id: 0x%x\n", (int)cis_vars->card_id);
     cmn_err(CE_CONT, "    config_index: 0x%x\n",
                        (u_char)cis_vars->config_index);
     cmn_err(CE_CONT, "    addr_lines: 0x%x\n",
                         (int)cis_vars->addr_lines);
     cmn_err(CE_CONT, "    pin: 0x%x\n", (int)cis_vars->pin);
     cmn_err(CE_CONT, "    n_cistpl_cftable_entry: 0x%x\n",
                         (int)cis_vars->n_cistpl_cftable_entry);
     cmn_err(CE_CONT, "    n_cistpl_funcid: 0x%x\n",
                        (int)cis_vars->n_cistpl_funcid);
```

```
        cmn_err(CE_CONT, "\n");
}
```

pcepp_destroy_cftable_list()

This utility frees the configuration table.

```
static void
pcepp_destroy_cftable_list(pcepp_cftable_t **cftable)
{
        pcepp_cftable_t         *cft, *ocft = NULL;

        cft = *cftable;

        while (cft) {
                ocft = cft;
                cft = cft->next;
        }

        while (ocft) {
                cft = ocft->prev;
                kmem_free(ocft, sizeof (pcepp_cftable_t));
                ocft = cft;
        }

        *cftable = NULL;
}
```

pcepp_set_cftable_desireability()

This utility sets the desirability factor based on the desirability order in the `pcepp_cis_addr_tran_t` table.

```
static void
pcepp_set_cftable_desireability(pcepp_cftable_t *cft)
{
        int     i;

        for (i = 0; i < (sizeof (pcepp_cis_addr_tran) /
                        sizeof (pcepp_cis_addr_tran_t)); i++) {
                if (cft->p.card_base1 ==
                                pcepp_cis_addr_tran[i].card_base1) {
                        cft->desireability =
```

```
                                        ((cft->desireability & 0x0ffff0000) |
                                    (pcepp_cis_addr_tran[i].desireability));
                                return;
                        }
                }
        }
```

pcepp_sort_cftable_list()

This utility sorts the configuration based on the desirability factor.

```
static void
pcepp_sort_cftable_list(pcepp_cftable_t **cftable)
{
        int                     did_swap = 1;
        pcepp_cftable_t         *cft;

        do {
                cft = *cftable;
                did_swap = 0;
                while (cft) {
                    if (cft->prev) {
                        if (cft->desireability <
                                        cft->prev->desireability) {
                                (void) pcepp_swap_cft(cftable, cft);
                                did_swap = 1;
                        } /* if (cft->desireability) */
                    } /* if (cft->prev) */
                    cft = cft->next;
                } /* while (cft) */
        } while (did_swap);
}
```

pcepp_swap_cft()

This utility sort routine swaps configuration table entries.

```
static void
pcepp_swap_cft(pcepp_cftable_t **cftable, pcepp_cftable_t *cft)
{
        pcepp_cftable_t         *cfttmp;

        if (cft->next) {
```

```
                cft->next->prev = cft->prev;
        }

        cft->prev->next = cft->next;
        cft->next = cft->prev;

        if (cft->prev->prev) {
                cft->prev->prev->next = cft;
        }
        cfttmp = cft->prev;

        if ((cft->prev = cft->prev->prev) == NULL) {
                *cftable = cft;
        }
       cfttmp->prev = cft;
}
```

Device Interrupt Handler Routines

The interrupt-handler routines include:

- `pcepp_intr()`
- `pcepp_softintr()`

pcepp_intr()

This is the driver interrupt-handler routine.

```
static u_int
pcepp_intr(pcepp_state_t *pps)
{
    int        serviced = DDI_INTR_UNCLAIMED;

    PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: intr\n",
                   pps->instance, pps->sn));

    /*
     * If the card isn't inserted or fully initialized yet,
     * this isn't an interrupt for us.  We must do this before
     * grabbing the mutex, since with shared interrupts, we
     * may get interrupts from other sources before we are fully
     * prepared for them.  We also need to stop accessing the
```

```
     * card with some promptness when the card gets yanked out
     * from under us.  The high-level card removal processing
     * clears the Card Inserted bit.
     */
    if (PCEPP_CARD_IS_READY(pps) && (pps->flags & PCEPP_ISOPEN)) {

        /*
         * Take care xmit interrupt
         */
        mutex_enter(&pps->irq_mutex);
        serviced = pcepp_xmit(pps);
        mutex_exit(&pps->irq_mutex);

        /*
         * if we've got any work to do,
         *  schedule a softinterrupt
         */
        if (serviced == PCEPP_XMIT_SUCCESS) {
            ddi_trigger_softintr(pps->softint_id);
        }
        return (DDI_INTR_CLAIMED);

    }  /* if (PCEPP_CARD_IS_READY) */

    return (DDI_INTR_UNCLAIMED);
}
```

pcepp_softintr()

When there is more data to be transferred to the printer, `pcepp_softintr()` calls this function to transfer the expected data.

```
static u_int
pcepp_softintr(caddr_t arg)
{
        register pcepp_state_t  *pps = (pcepp_state_t *)arg;
        int     ret = DDI_INTR_UNCLAIMED;

        /*
         * Make sure that we made it through attach.It's OK to run
         *  some of this code if there's work to do even if there
         *  is no card in the socket,since the work we're being
```

```
 *  asked to do might be cleanup from a card removal.
 */
if (pps->flags & PCEPP_ISOPEN) {
        /*
         * If there is more data, pcepp_start will
         *      continue to transfer the next data
         */
       PCEPP_MUTEX_ENTER(pps);
       (void) pcepp_start(pps);
       PCEPP_MUTEX_EXIT(pps);
        ret = DDI_INTR_CLAIMED;
}  /* if (pps->flags) */

return (ret);
}
```

STREAMS Routines

The following routines are standard STREAMS modules. Refer to the *STREAMS Programmer's Guide* for details. The STREAMS-related routines include:

- `open`(9E)
- `close`(9E)
- `ioctl`(9E)
- `pcepp_rput`()
- `pcepp_wput`()
- `pcepp_srvioc`()

pcepp_rput()

This routine is the read-side put procedure. Since this driver is unidirectional, the message is simply queued.

```
static int
pcepp_rput(
        queue_t *q,     /* pointer to the queue */
        mblk_t  *mp)    /* message pointer */
{
        register pcepp_state_t  *pps = (pcepp_state_t *)q->q_ptr;
```

```
        PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: rput\n",
                                        pps->instance, pps->sn));

        putq(q, mp);

        return (0);
}
```

pcepp_wput()

This routine is the write-side put procedure.

```
static int
pcepp_wput(
        queue_t *q,     /* pointer to the queue */
        mblk_t  *mp)    /* message pointer */
{
        register pcepp_state_t  *pps = (pcepp_state_t *)q->q_ptr;

        PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: wput\n",
                                        pps->instance, pps->sn));

        switch (mp->b_datap->db_type) {

        default:
                freemsg(mp);
                break;

        case M_FLUSH:   /* Canonical flush handling */
                if (*mp->b_rptr & FLUSHW) {
                        /*
                         * If the FLUSHW is set, the write message
                         *      queue is flushed.
                         */
                       mutex_enter(&pps->event_mutex);
                        PCEPP_MUTEX_ENTER(pps);
                        pps->card_event &= ~PCEPP_CARD_BUSY;
                        PCEPP_MUTEX_EXIT(pps);
                        mutex_exit(&pps->event_mutex);

                          flushq(q, FLUSHDATA);
                          *mp->b_rptr &= ~FLUSHW;
```

```
                }

                if (*mp->b_rptr & FLUSHR) {
                        /*
                         * If the FLUSHR is set, the read queue
                         *      is flushed, the message is sent
                         *      upstream using qreply()
                         */
                        flushq(RD(q), FLUSHDATA);
                        qreply(q, mp);
                } else {
                        /*
                         * If the FLUSHR is not set,
                         *      the message is discarded
                         */
                        freemsg(mp);
                }

                break;

        case M_IOCTL:
                (void) pcepp_ioctl(q, mp);
                break;

        case M_DATA:
                putq(q, mp);
               mutex_enter(&pps->event_mutex);
               (void) pcepp_start(pps);
               mutex_exit(&pps->event_mutex);
                break;

        }  /* switch (mp->b_datap->db_type) */

        return (0);
}
```

pcepp_open()

This routine is the STREAMS driver open procedure. It sets up the PCMCIA card, STREAMS queues, and internal `termio`(7I) structures, and then turns on the queue processing.

```
static int
pcepp_open(
        queue_t *q,          /* pointer to the read queue */
        dev_t   *devp,       /* pointer to major/minor device # */
        int     flag,        /* file flags */
        int     sflag,       /* stream open flags */
        cred_t  *credp)      /* pointer to a credentials struct */
{
        int                     ret;
        register struct strtty  *tp;
        pcepp_state_t           *pps;
        cs_ddi_info_t           cs_ddi_info;


        /*
         * This driver sample does not support the clone
         *    feature and will not do module open.
         *
         *      sflag values:
         *              MODOPEN         normal module open
         *              0               normal driver open
         *              CLONEOPEN       clone driver open
         */
        if (sflag) {
                PCEPP_DEBUG((CE_CONT, "pcepp: open failed, "
                        "Only support normal driver open"
                        "sflag (0x%x)\n", sflag));
                return (ENXIO);
        }

        /*
         * Since PC Card driver encodes physical socket number
         *  as part of the dev_t of device nodes instead of
         *  the instance number we have to use csx_CS_DDI_Info(9F)
         *  to get instance number that is associated with
         *  a particular PC Card driver and socket number
```

```
 *  for using in ddi_get_soft_state().
 */
cs_ddi_info.Socket = PCEPP_SOCKET((dev_t)*devp);
cs_ddi_info.driver_name = pcepp_name;
if (csx_CS_DDI_Info(&cs_ddi_info) != DDI_SUCCESS) {
        PCEPP_DEBUG((CE_NOTE,
               "pcepp: open, csx_CS_DDI_Info() failed\n"));
        return (ENXIO);
}

pps = ddi_get_soft_state(pcepp_soft_state_p,
                                cs_ddi_info.instance);
if (pps == NULL) {
        PCEPP_DEBUG((CE_CONT, "pcepp: open, "
                "could not get state structure\n"));
        return (ENODEV);
}

PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: open\n",
                        cs_ddi_info.instance, pps->sn));

if (!PCEPP_CARD_IS_READY(pps)) {
        PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: open, "
                "Found no PC Card\n", cs_ddi_info.instance,
                pps->sn));
        return (ENODEV);
}

/*
 * Check if open already. Cannot have multiple opens.
 */
if (q->q_ptr) {
        return (EBUSY);
}

/* Protect card structure */
mutex_enter(&pps->event_mutex);
PCEPP_HUTEX_ENTER(pps);

/*
 * Set up the hardware.
 *      Disable interrupt bit and configure the
```

```
 *      parallel port as the standard output port.
 */
csx_Put8(pps->card_handle, PCEPP_REGS_DCR,
        csx_Get8(pps->card_handle, PCEPP_REGS_DCR) &
                                ~PCEPP_ECR_INTR_ENB);
csx_Put8(pps->card_handle, PCEPP_REGS_DCR,
        PCEPP_ECR_SEL_PRT | PCEPP_ECR_INIT_PRT);

/*
 * Check printer condition
 */
if (pcepp_prtstatus(pps)) {
        PCEPP_MUTEX_EXIT(pps);
        mutex_exit(&pps->event_mutex);
        return (EIO);
}

/*
 * now set up the streams queues
 */
pps->rdq_ptr = q;
pps->wrq_ptr = WR(q);
q->q_ptr = (caddr_t)pps;
WR(q)->q_ptr = (caddr_t)pps;

/*
 * set up the tty structure for CENTRONICS-TYPE printer.
 */
tp = &pps->pcepp_tty;
tp->t_rdqp = q;
tp->t_dev = pps->sn;

if ((tp->t_state & (ISOPEN | WOPEN)) == 0) {
        tp->t_iflag = IGNPAR;
        tp->t_cflag = B300 | CS8 | CLOCAL;
}
tp->t_state &= ~WOPEN;
tp->t_state |=  CARR_ON | ISOPEN;

/*
 * start queue processing
 */
```

```
        qprocson(q);
        pps->flags |= PCEPP_ISOPEN;

        PCEPP_MUTEX_EXIT(pps);
        mutex_exit(&pps->event_mutex);

        return (0);
}
```

pcepp_close()

This routine is the STREAMS close procedure. It shuts down the PCMCIA card, turns off STREAMS queue processing, and cleans up the `termio`(7I) structure and STREAMS queues.

```
static int
pcepp_close(
        queue_t *q,         /* pointer to the read queue */
        int     flag,       /* file flags */
        cred_t  *credp)     /* pointer to a credentials struct */
{
        register struct strtty  *tp;
        pcepp_state_t           *pps;

        pps = (pcepp_state_t *)q->q_ptr;

        PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: close\n",
                                pps->instance, pps->sn));
        mutex_enter(&pps->event_mutex);

        /*
         * Make sure the card is presented before we
         *      reset card registers.
         */
        if (PCEPP_CARD_IS_READY(pps)) {
                /*
                 * Disable interrupt bit
                 */
                PCEPP_MUTEX_ENTER(pps);
                csx_Put8(pps->card_handle, PCEPP_REGS_DCR,
                        csx_Get8(pps->card_handle, PCEPP_REGS_DCR) &
                                                ~PCEPP_ECR_INTR_ENB);
```

```
                PCEPP_MUTEX_EXIT(pps);
        }  /* PCEPP_CARD_IS_READY */

        /*
         * clean up the queue pointers
         */
        qprocsoff(q);

        tp = &pps->pcepp_tty;
        tp->t_state &= ~(ISOPEN | CARR_ON);
        tp->t_rdqp = NULL;

        q->q_ptr = (queue_t *)NULL;
        WR(q)->q_ptr = (queue_t *)NULL;

        PCEPP_MUTEX_ENTER(pps);
        pps->rdq_ptr = (queue_t *)NULL;
        pps->wrq_ptr = (queue_t *)NULL;
        pps->flags &= ~PCEPP_ISOPEN;
        PCEPP_MUTEX_EXIT(pps);

        mutex_exit(&pps->event_mutex);

        return (0);
}
```

pcepp_ioctl()

This routine processes `M_IOCTL` messages sent to the driver. Since this is a STREAMS driver, the I/O control messages are not processed through an entry point in `cb_ops`(9S). `pcepp_ioctl()` is called by `pcepp_start()` while processing messages sent to this STREAMS module.

```
static int
pcepp_ioctl(
        queue_t *q,     /* pointer to the queue */
        mblk_t  *mp)    /* message pointer */
{
        struct iocblk   *ioc;
        pcepp_state_t   *pps;
```

```
/* LINTED */
ioc = (struct iocblk *)mp->b_rptr;
pps = (pcepp_state_t *)q->q_ptr;

PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: ioctl\n",
                        pps->instance, pps->sn));

switch (ioc->ioc_cmd) {

case TCSBRK:
case TCSETAW:
case TCSETSW:
case TCSETSF:
case TCSETAF:
        if (q->q_first) {
                (void) putq(q, mp);
                break;
        }
        (void) pcepp_srvioc(q, mp);
        break;

case TCSETS:
case TCSETA:    /* immediate parm set   */
        if (pps->card_event & PCEPP_CARD_BUSY) {
                (void) putq(q, mp);
                break;
        }
        (void) pcepp_srvioc(q, mp);
        break;

case TCGETS:
case TCGETA:    /* immediate parm retrieve */
        (void) pcepp_srvioc(q, mp);
        break;

default:
        /*
         * Passing the unknown IOCTL to upper module
         */
        ioc->ioc_error = EINVAL;
        ioc->ioc_rval = (-1);
        mp->b_datap->db_type = M_IOCNAK;
```

```
                qreply(q, mp);
                break;

        }  /* switch (ioc->ioc_cmd) */

        return (0);
}
```

pcepp_srvioc()

This routine processes the `termio`(7I) I/O control commands for `pcepp_ioctl()`.

```
static void
pcepp_srvioc(
        queue_t *q,     /* pointer to the queue */
        mblk_t  *mp)    /* message pointer */
{
        struct strtty   *tp;
        struct iocblk   *ioc;
        struct termio   *cb;
        struct termios  *scb;
        pcepp_state_t   *pps;
        mblk_t          *mpr;
        mblk_t          *mp1;

        /* LINTED */
        ioc = (struct iocblk *)mp->b_rptr;
        pps = (pcepp_state_t *)q->q_ptr;
        tp = &pps->pcepp_tty;

        PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: srvioc\n",
                                pps->instance, pps->sn));

        switch (ioc->ioc_cmd) {

        case TCSBRK:
                /* Skip the break since it's a parallel port. */
                mp->b_datap->db_type = M_IOCACK;
                mp1 = unlinkb(mp);
                if (mp1) {
```

```
                        freeb(mp1);
                }
                ioc->ioc_count = 0;
                qreply(q, mp);
                break;

        case TCSETA:
        case TCSETAW:
                if (!mp->b_cont) {
                        ioc->ioc_error = EINVAL;
                        mp->b_datap->db_type = M_IOCNAK;
                        ioc->ioc_count = 0;
                        qreply(q, mp);
                        break;
                }
                /* LINTED */
                cb = (struct termio *)mp->b_cont->b_rptr;
                tp->t_cflag = cb->c_cflag;
                tp->t_iflag = cb->c_iflag;
                mp->b_datap->db_type = M_IOCACK;
                mp1 = unlinkb(mp);
                if (mp1) {
                        freeb(mp1);
                }
                ioc->ioc_count = 0;
                qreply(q, mp);
                break;

        case TCSETS:
        case TCSETSW:
                if (!mp->b_cont) {
                        ioc->ioc_error = EINVAL;
                        mp->b_datap->db_type = M_IOCNAK;
                        ioc->ioc_count = 0;
                        qreply(q, mp);
                        break;
                }

                /* LINTED */
                scb = (struct termios *)mp->b_cont->b_rptr;
                tp->t_cflag = scb->c_cflag;
                tp->t_iflag = scb->c_iflag;
```

```
                mp->b_datap->db_type = M_IOCACK;
                mp1 = unlinkb(mp);
                if (mp1) {
                        freeb(mp1);
                }
                ioc->ioc_count = 0;
                qreply(q, mp);
                break;

        case TCGETA:
                /* immediate parm retrieve */
                if (mp->b_cont) {
                        /* bad user supplied parameter */
                        freemsg(mp);
                }

                if ((mpr = allocb(sizeof (struct termio),
                                        BPRI_MED)) == NULL) {
                        (void) putbq(q, mp);
                        return;
                }

                mp->b_cont = mpr;
                /* LINTED */
                cb = (struct termio *)mp->b_cont->b_rptr;
                cb->c_iflag = tp->t_iflag;
                cb->c_cflag = tp->t_cflag;

                mp->b_cont->b_wptr += sizeof (struct termio);
                mp->b_datap->db_type = M_IOCACK;

                ioc->ioc_count = sizeof (struct termio);
                qreply(q, mp);
                break;

        case TCGETS:
                /* immediate parm retrieve */
                if (mp->b_cont) {
                        freemsg(mp->b_cont);
                }
                if ((mpr = allocb(sizeof (struct termio),
                                        BPRI_MED)) == NULL) {
```

```
                        (void) putbq(q, mp);
                        return;
                }

                mp->b_cont = mpr;
                /* LINTED */
                scb = (struct termios *)mp->b_cont->b_rptr;
                scb->c_iflag = tp->t_iflag;
                scb->c_cflag = tp->t_cflag;

                mp->b_cont->b_wptr += sizeof (struct termio);
                mp->b_datap->db_type = M_IOCACK;

                ioc->ioc_count = sizeof (struct termio);
                qreply(q, mp);
                break;

        default:
                /* unexpected ioctl type */
                if (canput(RD(q)->q_next) == 1) {
                        mp->b_datap->db_type = M_IOCNAK;
                        ioc->ioc_count = 0;
                        qreply(q, mp);
                } else {
                        (void) putbq(q, mp);
                }
                break;

        }  /* switch (ioc->ioc_cmd) */
}
```

pcepp_xmit()

This routine sends a character to the printer if the printer is not busy.

```
static int
pcepp_xmit(pcepp_state_t *pps)
{
        PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: xmit\n",
                                pps->instance, pps->sn));

        if (pcepp_prtstatus(pps)) {
```

```
                        return (PCEPP_XMIT_BUSY);
                }

                /* Disable interrupt bit */
                csx_Put8(pps->card_handle, PCEPP_REGS_DCR,
                        csx_Get8(pps->card_handle, PCEPP_REGS_DCR) &
                                                ~PCEPP_ECR_INTR_ENB);

                (void) pcepp_strobe_pulse(pps);

                /* Clear busy flag */
                pps->card_event &= ~PCEPP_CARD_BUSY;

                return (PCEPP_XMIT_SUCCESS);
}
```

pcepp_strobe_pulse()

After the data is loaded into the data register, the strobe pulse signal must be generated.

```
static void
pcepp_strobe_pulse(pcepp_state_t *pps)
{
        /* Enable strobe signal */
        csx_Put8(pps->card_handle, PCEPP_REGS_DCR,
                csx_Get8(pps->card_handle, PCEPP_REGS_DCR) |
                PCEPP_ECR_STROBE_ENB);

        PCEPP_HIMUTEX_EXIT(pps);
        /* Generate 10msec STROBE pulse */
        drv_usecwait(10);
        PCEPP_HIMUTEX_ENTER(pps);

        /* Enable strobe signal */
        csx_Put8(pps->card_handle, PCEPP_REGS_DCR,
                csx_Get8(pps->card_handle, PCEPP_REGS_DCR) &
                ~PCEPP_ECR_STROBE_ENB);
}
```

pcepp_start()

This routine processes M_IOCTL and M_DATA messages. M_IOCTL messages are forwarded to pcepp_ioctl(). M_DATA messages provide the data that is sent to the printer one at a time. An interrupt that is triggered when the character is received by the printer causes the interrupt handler to invoke this routine again to cause the next character to be sent.

```
static void
pcepp_start(register pcepp_state_t *pps)
{
        register queue_t        *q;
        register mblk_t         *bp;

        PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: start\n",
                                pps->instance, pps->sn));

        if (pps->card_event & PCEPP_CARD_BUSY) {
                PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: start, "
                        "Card is BUSY\n", pps->instance, pps->sn));
                return;
        }

        if ((q = pps->wrq_ptr) == NULL) {
                cmn_err(CE_CONT, "pcepp%d.%d: start, "
                       "q is NULL\n", pps->instance, (int)pps->sn);
                return;
        }

        /*
         * handle next message block (if any)
         */
        if ((bp = getq(q)) == NULL) {
                return;
        }

        /*
         * Grab a hardwire mutex here since some of these flags
         *      are modified in our interrupt handlers
         */
        PCEPP_MUTEX_ENTER(pps);
        pps->msg = bp;
```

```
PCEPP_MUTEX_EXIT(pps);

switch (bp->b_datap->db_type) {

    case M_IOCTL:
        (void) pcepp_ioctl(q, bp);
        break;

    case M_DATA:

        /* Transfer to Tx soft data buffer */
        PCEPP_MUTEX_ENTER(pps);
        pps->pcepp_txdata = *bp->b_rptr;
        bp->b_rptr++;
        PCEPP_MUTEX_EXIT(pps);

        /*
         * If there is more data to be transferred,
         *      then just put this remaining data
         *      block back and continue send data to
         *      the printer
         */
        if ((bp->b_wptr - bp->b_rptr) > 0) {
                putbq(q, bp);
        } else {
                PCEPP_MUTEX_ENTER(pps);
                bp = pps->msg->b_cont;
                pps->msg->b_cont = NULL;
                freeb(pps->msg);
                PCEPP_MUTEX_EXIT(pps);
        }

        if (PCEPP_CARD__IS_READY(pps)) {
                PCEPP_MUTEX_ENTER(pps);
                pps->card_event |= PCEPP_CARD_BUSY;
                /*
                 * Printer is ready, send a character
                 *      and generating STROBE pulse
                 */
                csx_Put8(pps->card_handle, PCEPP_REGS_DATA,
                                        pps->pcepp_txdata);
                /*
```

```
                                 * Enable the transmitter empty interrupts
                                 */
                                csx_Put8(pps->card_handle, PCEPP_REGS_DCR,
                                        csx_Get8(pps->card_handle,
                                                        PCEPP_REGS_DCR) |
                                        PCEPP_ECR_INTR_ENB);
                                PCEPP_MUTEX_EXIT(pps);
                        }  /* PCEPP_CARD_IS_READY */
                        break;

                    default:
                        freemsg(bp);
                        break;
                }  /* switch (bp->b_datap->db_type) */
        }
```

pcepp_prtstatus()

This routine determines whether the printer is ready to print.

```
static int
pcepp_prtstatus(register pcepp_state_t *pps)
{
      u_char                  prtstatus;
      int                     retval = 0;

      prtstatus = csx_Get8(pps->card_handle, PCEPP_REGS_DSR);

      if (prtstatus & PCEPP_ESR_PERROR) {
              cmn_err(CE_CONT, "pcepp%d.%d: Printer PAPER OUT\n",
                                      pps->instance, pps->sn);
              retval = 1;
      }

      if ((prtstatus & PCEPP_ESR_SELECT) == 0) {
              cmn_err(CE_CONT, "pcepp%d.%d: Printer is OFFLINE\n",
                                      pps->instance, pps->sn);
              retval = 1;
      }

      if ((prtstatus & PCEPP_ESR_FAULT) == 0) {
              cmn_err(CE_CONT, "pcepp%d.%d: "
```

```
                        "ERROR has been detected\n",
                        pps->instance, pps->sn);
                retval = 1;
        }

        if ((prtstatus & PCEPP_ESR_PRN_READY) == 0) {
          PCEPP_DEBUG((CE_CONT, "pcepp%d.%d: Printer is NOT READY\n",
                                        pps->instance, pps->sn));
                retval = 1;
        }

        return (retval);
}
```

Index

D

E

F

G

H

I

L

M

N

O

P

R

S

T

V

W

X