Linux PCMCIA Programmer's Guide

David Hinds, dhinds@allegro.stanford.edu v1.24, 1995/07/31 05:48:28

This document describes how to write kernel device drivers for the Linux PCMCIA Card Services interface. It also describes how to write user-mode utilities for communicating with Card Services. The latest version of this document can always be found at cb-iris.stanford.edu in /pub/pcmcia/doc.

Contents

1 Introduction

The Linux kernel PCMCIA system has three main components. At the lowest level are the socket drivers. Next is the Card Services module. Drivers for specific cards are layered on top of Card Services. One special Card Services client, called Driver Services, provides a link betweek user level PCMCIA utility programs and the kernel PCMCIA facilities.

The socket driver layer is loosely based on the Socket Services API. There are two socket driver modules. The `tcic` module supports the Databook TCIC-2 family of PCMCIA controllers. The `i82365` module supports the Intel i82365sl family and various Intel-compatible controllers, including Cirrus, VLSI, Ricoh, and Vadem chips.

Card Services is the largest single component of the PCMCIA package. It provides an API somewhat similar to DOS Card Services, adapted to a Unix environment. The Linux implementation was based in part on the Solaris PCMCIA interface specification. It is implemented in the `pcmcia_core` module.

The Driver Services layer implements a user mode pseudo-device for accessing some Card Services functions from PCMCIA utility programs. It is responsible for keeping track of all PCMCIA client drivers, and for matching up drivers with physical sockets. It is implemented in the `ds` module.

This document describes the kernel interface to the Card Services and Driver Services modules, and the user interface to Driver Services. It is intended for use by PCMCIA device driver developers. The Linux PCMCIA-HOWTO describes how to install and use Linux PCMCIA support. It is available from `cb-iris.stanford.edu` in `/pub/pcmcia`.

1.1 Copyright notice and disclaimer

1.2 Acknowledgements

I'd like to thank all the Linux users who have helped test and debug this PCMCIA software, and who have helped with driver development. I should also thank Linus Torvalds, Donald Becker, Alan Cox, and Bjorn Ekwall for Linux kernel development help. I'm especially grateful to Michael Bender for many helpful discussions about the Solaris PCMCIA implementation.

2 Basic PCMCIA Concepts

2.1 The PCMCIA socket interface

The PCMCIA card bus has two basic operating modes: “memory-only” and “memory and IO”. The first mode was defined by the original Version 1.0 specification and only supports simple memory cards. The second mode, defined in Version 2.0, redefines a few of the memory card control signals to support IO port addressing and IO interrupt signalling.

PCMCIA cards have two memory spaces: “attribute memory” and “common memory”. The PCMCIA interface can address up to 16MB of each type of memory. Attribute memory is typically used for holding descriptive information and configuration registers. Common memory may include the bulk storage of a memory card, or device buffers in the case of IO cards. All cards that are compliant with the version 2.0 PCMCIA specification should have a Card Information Structure (or “CIS”) in attribute memory, which describes the card and how it should be configured.

Separate control signals allow cards to signal their operating status to the host. These signals include card detect, ready/busy, write protect, battery low, and battery dead.

The "memory and IO" interface mode allows cards to address up to 64K of IO ports. It also allows cards to signal IO interrupts, and routes one card output to the host system's speaker. In this mode, several of the memory card control signals are unavailable because those pins are used to carry the extra IO card signals. On some cards, these signals can instead be read from a special configuration register in attribute memory, the "Pin Replacement Register".

2.2 The PCMCIA socket controller

The PCMCIA controller serves as a bridge between PCMCIA cards and the system bus. There are several varieties of controllers, but all share the same basic functionality. The Socket Services software layer takes care of all the details of how to program the PCMCIA controller.

The PCMCIA controller has the job of mapping windows of addresses in the host memory and IO spaces to windows of addresses in card space. All controllers support at least four independent memory windows and two IO windows per socket.

Each memory window is defined by a base address in the host address space, a base address in the card address space, and a window size. Some controllers differ in their alignment rules for memory windows, but all controllers will support windows whose size is at least 4K and also a power of two, and where the base address is a multiple of the window size. Each window can be programmed to point to either attribute or common memory.

IO windows differ from memory windows in that host addresses that fall within an IO window are not modified before they are passed on to an IO card. Effectively, the base addresses of the window in the host and card address spaces are always equal. IO windows also have no alignment or size restrictions; an IO window can start and end on any byte boundary in the 64K IO address space.

The PCMCIA card bus defines a single interrupt signal from the card to the controller. The controller then has the responsibility of steering this interrupt to an appropriate interrupt request ("irq") line. All controllers support steering card IO interrupts to essentially any free interrupt line. Because steering happens in the controller, the card itself is unaware of which interrupt it uses.

All PCMCIA controllers can generate interrupts in response to card status changes. These interrupts are distinct from the IO interrupts generated by an IO card, and use a separate interrupt line. Signals that can generate interrupts include card detect, ready/busy, write protect, battery low, and battery dead.

3 Card Services subfunction descriptions

Card Services calls have the general form:

```
#include "cs_types.h"
#include "cs.h"

int CardServices(int subfunc, void *arg1, void *arg2, ...);
```

Some Card Services functions require additional `#include` statements. The particular subfunction determines the number of expected arguments. A return code of `CS_SUCCESS` indicates that a call succeeded. Other return codes indicate errors.

3.1 Client management functions

Device drivers that use Card Services functions are called "clients". A device driver should use the `RegisterClient` call to get a client handle before using other services. Most Card Services functions will take this client handle as an argument. Before unloading, drivers should also unregister with `DeregisterClient`.

3.1.1 RegisterClient

```
int CardServices(RegisterClient, client_handle_t *client, client_reg_t *reg);
```

The `client_reg_t` data structure is given by:

```
typedef struct client_reg_t {
        dev_info_t              *dev_info;
        u_long                  Attributes;
        u_long                  EventMask;
        int                     (*event_handler)(event_t event, int priority,
                                                 event_callback_args_t *args);
        event_callback_args_t   event_callback_args;
        u_long                  Version;
} client_reg_t;
```

`RegisterClient` establishes a link between a client driver and Card Services, and connects the client with an appropriate socket. The `dev_info` parameter is used by Card Services to match the client with a socket; this correspondence is normally established by Driver Services via a call to `BindDevice`. If successful, a client handle will be returned in `client`.

The following flags can be specified in `Attributes`:

`INFO_DRIVER_SERVICES`

For use only by the Driver Services client, among other things, specifies that this client should not be automatically unbound when a card is ejected from this socket.

`INFO_IO_CLIENT`

Specifies that this client is an IO card driver.

`INFO_MTD_CLIENT`

Specifies that this client is a Memory Technology Driver.

`INFO_MEM_CLIENT`

Specifies that this client is a memory card driver.

`INFO_CARD_SHARE`

Included for compatibility, has no effect.

INFO_CARD_EXCL

Included for compatibility, has no effect.

EventMask specifies what events this client should be notified of. The event_handler entry point will be called by Card Services when an event in EventMask is processed. The event_handler_args structure is a template for the structure that will be passed to the event handler. The Version parameter identifies the Card Services version level that this driver expects; it is currently ignored.

A driver should be prepared to handle Card Services events before calling RegisterClient. This call will always generate a CS_REGISTRATION_COMPLETE event, and may also generate an artificial CS_CARD_INSERTION event if the socket is currently occupied.

Return codes:

CS_OUT_OF_RESOURCE

An appropriate socket could not be found for this driver.

3.1.2 DeregisterClient

```
int CardServices(DeregisterClient, client_handle_t client);
```

DeregisterClient severs the connection between a client and Card Services. It should be called after the client has freed any resources it has allocated. Once a connection is broken, it cannot be reestablished until after another call to BindDevice.

Return codes:

CS_BAD_HANDLE

The client handle is invalid.

CS_IN_USE

The client still has allocated resources, such as IO port windows or an interrupt, or the socket configuration is locked.

3.1.3 SetEventMask

```
int CardServices(SetEventMask, client_handle_t client, eventmask_t *mask);
```

The eventmask_t structure is given by:

```
typedef struct eventmask_t {
        u_long          Attributes;
        u_long          EventMask;
} eventmask_t;
```

`SetEventMask` updates the mask that determines which events this client will be notified of.

Return codes:

`CS_BAD_HANDLE`

The client handle is invalid.

3.1.4 BindDevice

```
int CardServices(BindDevice, bind_req_t *req);
```

The `bind_req` structure is given by:

```
typedef struct bind_req_t {
        socket_t        Socket;
        dev_info_t      *dev_info;
} bind_req_t;
```

`BindDevice` associates a device driver with a particular socket. It is normally called by Driver Services after a newly inserted card has been identified. Once a driver has been bound to a socket, it will be eligible to register as a client of that socket. Note that this call does not take a client handle as an argument. This is the only Card Services call that takes a socket number as an argument.

Return codes:

`CS_BAD_SOCKET`

The specified socket number is invalid.

3.2 Socket state control

These functions are more or less concerned with getting and setting the current operating state of a socket. `GetStatus` returns the current socket state. `ResetCard` is used to send a hard reset signal to a socket. `SuspendCard` and `ResumeCard` can be used to power down and power up a socket without releasing the drivers currently bound to that socket. `EjectCard` and `InsertCard` essentially mimic real card ejection and insertion events.

3.2.1 GetStatus

```
int CardServices(GetStatus, client_handle_t client, status_t *status);
```

The `status_t` data structure is given by:

```
typedef struct status_t {
        u_long          CardState;
        u_long          SocketState;
} status_t;
```

`GetStatus` returns the current status of a client's socket. The following flags are defined in `CardState`:

`CS_EVENT_CARD_DETECT`

Specifies that the socket is occupied.

`CS_EVENT_WRITE_PROTECT`

Specifies that the card is currently write protected.

`CS_EVENT_BATTERY_LOW`

Specifies that the card battery is low.

`CS_EVENT_BATTERY_DEAD`

Specifies that the card battery is dead.

`CS_EVENT_READY_CHANGE`

Specifies that the card is ready.

`CS_EVENT_PM_SUSPEND`

Specifies that the socket is suspended.

`SocketState` is currently unused, but in theory, it should latch changes in the state of the fields in `CardState`.

Return codes:

`CS_BAD_HANDLE`

The client handle is invalid.

3.2.2 ResetCard

```
int CardServices(ResetCard, client_handle_t client);
```

`ResetCard` requests that a client's socket be reset. When this call is made, Card Services sends all clients a `CS_EVENT_RESET_REQUEST` event. If any client rejects the request, Card Services sends the initiating client a `CS_EVENT_RESET_COMPLETE` event with `event_callback_args.info` set to the return code of the client that rejected the request.

If all clients agree to the request, Card Services sends a `CS_EVENT_RESET_PHYSICAL` event, then resets the socket. When the socket signals that it is ready, a `CS_EVENT_CARD_RESET` event is generated. Finally, a `CS_EVENT_RESET_COMPLETE` event is sent to the initiating client, with `event_callback_args.info` set to zero.

Return codes:

`CS_BAD_HANDLE`

The client handle is invalid.

`CS_NO_CARD`

The socket assigned to this client is currently vacant.

`CS_IN_USE`

This socket is currently being reset.

3.2.3 SuspendCard

```
int CardServices(SuspendCard, client_handle_t client);
```

Card Services sends all clients `CS_EVENT_PM_SUSPEND` events, then shuts down and turns off power to the socket.

Return codes:

`CS_BAD_HANDLE`

The client handle is invalid.

`CS_NO_CARD`

The socket assigned to this client is currently vacant.

`CS_IN_USE`

This socket is already suspended.

3.2.4 ResumeCard

```
int CardServices(ResumeCard, client_handle_t client);
```

After restoring power to the socket, Card Services will notify all clients with `CS_EVENT_PM_RESUME` events.

Return codes:

`CS_BAD_HANDLE`

The client handle is invalid.

`CS_NO_CARD`

The socket assigned to this client is currently vacant.

`CS_IN_USE`

This socket is not currently suspended.

3.2.5 EjectCard

```
int CardServices(EjectCard, client_handle_t client);
```

Card Services sends eject events to all clients, then shuts down and turns off power to the socket. All clients except for Driver Services will be unlinked from the socket.

Return codes:

`CS_BAD_HANDLE`

The client handle is invalid.

`CS_NO_CARD`

The socket assigned to this client is currently vacant.

3.2.6 InsertCard

```
int CardServices(InsertCard, client_handle_t client);
```

Card Services sends insertion events to all clients of this socket (normally, only Driver Services).

Return codes:

`CS_BAD_HANDLE`

The client handle is invalid.

`CS_NO_CARD`

The socket assigned to this client is currently vacant.

`CS_IN_USE`

The socket has already been configured.

3.3 IO card configuration calls

The normal order of events is for a driver to reserve IO ports and an interrupt line with calls to `RequestIO` and `RequestIRQ`, then to call `RequestConfiguration` to actually configure the socket. If any of these calls fails, a driver should be sure to release any resources it successfully reserved.

3.3.1 RequestIO

```
int CardServices(RequestIO, client_handle_t client, io_req_t *req);
```

The `io_req_t` data structure is given by:

```
typedef struct io_req_t {
        ioaddr_t        BasePort1;
        ioaddr_t        NumPorts1;
        u_long          Attributes1;
        ioaddr_t        BasePort2;
        ioaddr_t        NumPorts2;
        u_long          Attributes2;
        u_long          IOAddrLines;
} io_req_t;
```

`RequestIO` reserves IO port windows for a card. If `BasePort1` is non-zero, it specifies the IO port address of the window to be reserved; if it is zero, Card Services will find an available window and set `BasePort1` to this address. If `NumPorts2` is non-zero, a second IO port window will also be reserved. `IOAddrLines` specifies the number of address lines that are actually decoded by the PCMCIA card; this is not currently used.

This call does not actually configure a socket's IO windows: this is done by a subsequent call to `RequestConfiguration`.

The following flags can be specified in `Attributes1` and `Attributes2`:

`IO_DATA_PATH_WIDTH`

This field may either be `IO_DATA_PATH_WIDTH_16` for 16-bit access, or `IO_DATA_PATH_WIDTH_8` for 8-bit access, or `IO_DATA_PATH_WIDTH_AUTO` to dynamically size the bus based on the access size.

Return codes:

`CS_BAD_HANDLE`

The client handle is invalid.

`CS_NO_CARD`

The socket assigned to this client is currently vacant.

`CS_IN_USE`

This socket's IO windows have already been reserved.

`CS_CONFIGURATION_LOCKED`

This socket's configuration has been locked by a call to `RequestConfiguration`.

`CS_BAD_ATTRIBUTE`

An unsupported attribute flag was specified.

3.3.2 ReleaseIO

```
int CardServices(ReleaseIO, client_handle_t client, io_req_t *req);
```

`ReleaseIO` un-reserves IO port windows allocated by a previous call to `RequestIO`. The `req` parameter should be the same one passed to `RequestIO`.

Return codes:

`CS_BAD_HANDLE`

The client handle is invalid.

`CS_CONFIGURATION_LOCKED`

This socket's configuration has been locked by a call to `RequestConfiguration`. The configuration should be released before calling ReleaseIO.

`CS_BAD_ARGS`

The parameters in `req` do not match the parameters passed to `RequestIO`.

3.3.3 RequestIRQ

```
int CardServices(RequestIRQ, client_handle_t client, irq_req_t *req);
```

The `irq_req_t` structure is given by:

```
typedef struct irq_req_t {
        u_long          Attributes;
        u_long          AssignedIRQ;
        u_long          IRQInfo1, IRQInfo2;
} irq_req_t;
```

`RequestIRQ` reserves an interrupt line for use by a PCMCIA card. The `IRQInfo1` and `IRQInfo2` fields correspond to the interrupt description bytes in a `CFTABLE_ENTRY` tuple. If `IRQ_INFO2_VALID` is set in `IRQInfo1`, then `IRQInfo2` is a bit-mapped mask of allowed interrupt values. Each bit corresponds to one interrupt line: bit 0 = irq 0, bit 1 = irq 1, etc. So, a mask of 0x1100 would mean that interrupts 12 and 8 could be used. If `IRQ_INFO2_VALID` is not set, `IRQInfo1` is just the desired interrupt number. If the call is successful, the reserved interrupt is returned in `AssignedIRQ`.

Note that this call does not install an interrupt handler. The calling driver should use the `request_irq()` kernel function to actually catch the assigned interrupt. Also, the interrupt will not be enabled until a call to `RequestConfiguration`.

The following flags can be specified in `Attributes`:

`IRQ_FORCED_PULSE`

Specifies that the interrupt should be configured for pulsed mode, rather than the default level mode.

`IRQ_TYPE_TIME`

Specifies that this interrupt can be time-shared with other Card Services drivers. Only one driver should enable the interrupt at any time.

`IRQ_FIRST_SHARED`

In conjunction with `IRQ_TYPE_TIME`, this should be set by the first driver requesting a shared interrupt.

Return codes:

`CS_BAD_HANDLE`

The client handle is invalid.

`CS_NO_CARD`

The socket assigned to this client is currently vacant.

`CS_IN_USE`

An interrupt has already been reserved for this socket, or the requested interrupt is unavailable.

`CS_CONFIGURATION_LOCKED`

This socket's configuration has been locked by a call to `RequestConfiguration`.

`CS_BAD_ATTRIBUTE`

An unsupported attribute flag was specified.

3.3.4 ReleaseIRQ

```
int CardServices(ReleaseIRQ, client_handle_t client, irq_req_t *req);
```

`ReleaseIRQ` un-reserves an interrupt assigned by an earlier call to `RequestIRQ`. The `req` structure should be the same structure that was passed to `RequestIRQ`.

Return codes:

`CS_BAD_HANDLE`

The client handle is invalid.

`CS_CONFIGURATION_LOCKED`

This socket's configuration has been locked by a call to `RequestConfiguration`. The configuration should be released before calling ReleaseIRQ.

`CS_BAD_IRQ`

The parameters in `req` do not match the parameters passed to `RequestIRQ`.

3.3.5 RequestConfiguration

```
int CardServices(RequestConfiguration, client_handle_t client, config_req_t *req);
```

The `config_req_t` structure is given by:

```
typedef struct config_req_t {
        u_long          Attributes;
        u_long          Vcc, Vpp1, Vpp2;
        u_long          IntType;
        caddr_t         ConfigBase;
        u_char          Status, Pin, Copy;
        u_char          ConfigIndex;
        u_long          Present;
} config_req_t;
```

`RequestConfiguration` is responsible for actually configuring a socket. This includes setting voltages, setting CIS configuration registers, setting up IO port windows, and setting up interrupts.

`IntType` specifies the type of interface to use for this card. It may either be `INT_MEMORY` or `INT_MEMORY_AND_IO`. Voltages are specified in units of 1/10 volt. Currently, `Vpp1` must equal `Vpp2`.

The following flags can be specified in `Attributes`. DMA and speaker control are not supported on all systems.

`CONF_ENABLE_IRQ`

Enable the IO interrupt reserved by a previous call to `RequestIRQ`.

`CONF_ENABLE_DMA`

Enable DMA accesses for this socket.

`CONF_ENABLE_SPKR`

Enable speaker output from this socket.

The `Present` parameter is a bit map specifying which CIS configuration registers are implemented by this card. `ConfigBase` gives the offset of the configuration registers in attribute memory. The following registers can be specified:

`PRESENT_OPTION`

Specifies that the Configuration Option Register is present. The COR register will be set using the `ConfigIndex` parameter.

`PRESENT_STATUS`

Specifies that the Card Configuration and Status Register is present. The CCSR will be initialized with the `Status` parameter.

`PRESENT_PIN_REPLACE`

Specifies that the Pin Replacement Register is present. The PRR will be initialized with the `Pin` parameter.

`PRESENT_COPY`

Specifies that the Socket and Copy Register is present. The SCR will be initialized with the `Copy` parameter.

Return codes:

`CS_BAD_HANDLE`

The client handle is invalid.

`CS_NO_CARD`

The socket assigned to this client is currently vacant.

`CS_OUT_OF_RESOURCE`

Card Services was unable to allocate a memory window to access the card's configuration registers.

`CS_CONFIGURATION_LOCKED`

This socket's configuration has already been locked by another call to `RequestConfiguration`.

`CS_BAD_VCC`

The requested Vcc voltage is not supported.

`CS_BAD_VPP`

The requested Vpp1/Vpp2 voltage is not supported.

3.3.6 ModifyConfiguration

```
int CardServices(ModifyConfiguration, client_handle_t client, modconf_t *mod);
```

The `modconf_t` structure is given by:

```
typedef struct modconf_t {
        u_long          Attributes;
        u_long          Vcc, Vpp1, Vpp2;
} modconf_t;
```

`ModifyConfiguration` modifies some attributes of a socket that has been configured by a call to `RequestConfiguration`.

The following flags can be specified in `Attributes`:

`CONF_IRQ_CHANGE_VALID`

Indicates that the CONF_ENABLE_IRQ setting should be updated.

`CONF_ENABLE_IRQ`

Specifies that IO interrupts should be enabled for this socket.

`CONF_VCC_CHANGE_VALID`

Indicates that Vcc should be updated.

`CONF_VPP1_CHANGE_VALID`

Indicates that Vpp1 should be updated.

`CONF_VPP2_CHANGE_VALID`

Indicates that Vpp2 should be updated.

Currently, Vpp1 and Vpp2 must always have the same value. So, the two values must always be changed at the same time.

Return codes:

`CS_BAD_HANDLE`

The client handle is invalid.

`CS_NO_CARD`

The socket assigned to this client is currently vacant.

`CS_CONFIGURATION_LOCKED`

This actually means that this socket has **not** been locked.

`CS_BAD_VCC`

The requested Vcc voltage is not supported.

`CS_BAD_VPP`

The requested Vpp1/Vpp2 voltage is not supported.

3.3.7 ReleaseConfiguration

```
int CardServices(ReleaseConfiguration, client_handle_t client, config_req_t *req);
```

`ReleaseConfiguration` un-configures a socket previously set up by a call to `RequestConfiguration`. The `req` parameter should be the same one used to configure the socket.

Return codes:

`CS_BAD_HANDLE`

The window handle is invalid, or the socket is not configured.

3.3.8 GetConfigurationInfo

```
int CardServices(GetConfigurationInfo, client_handle_t client, config_t *config);
```

The `config_t` structure is given by:

```
typedef struct config_t {
        u_long          Attributes;
        u_long          Vcc, Vpp1, Vpp2;
        u_long          IntType;
        caddr_t         ConfigBase;
        u_char          Status, Pin, Copy, Option;
        u_long          Present;
        u_long          AssignedIRQ;
        u_long          IRQAttributes;
        ioaddr_t        BasePort1;
        ioaddr_t        NumPorts1;
        u_long          Attributes1;
        ioaddr_t        BasePort2;
        ioaddr_t        NumPorts2;
        u_long          Attributes2;
        u_long          IOAddrLines;
} config_t;
```

`GetConfigurationInfo` returns the current socket configuration as it was set up by `RequestIO`, `RequestIRQ`, and `RequestConfiguration`. It can only be applied to a fully configured socket.

Return codes:

`CS_BAD_HANDLE`

The window handle is invalid, or the socket is not configured.

`CS_NO_CARD`

The socket assigned to this client is currently vacant.

`CS_CONFIGURATION_LOCKED`

This actually means that the configuration has **not** been locked.

3.4 Card Information Structure (CIS) calls

The definition of the Card Information Structure (CIS) is the darkest chapter of the PCMCIA standard. All version 2 PCMCIA cards should have a CIS, which describes the card and how it should be configured. The CIS is a linked list of "tuples" in the card's attribute memory space. Each tuple consists of an identification code, a length byte, and a series of data bytes. The layout of the data bytes for some tuple types is absurdly complicated, in an apparent effort to use every last bit.

The `ValidateCIS` call checks to see if a card has a reasonable CIS. The `GetFirstTuple` and `GetNextTuple` calls are used to step through CIS tuple lists. `GetTupleData` extracts data bytes from a tuple. And `ParseTuple` interprets a limited number of particularly important tuple types.

3.4.1 GetFirstTuple, GetNextTuple

```
#include "cistpl.h"

int CardServices(GetFirstTuple, client_handle_t client, tuple_t *tuple);
int CardServices(GetNextTuple, client_handle_t client, tuple_t *tuple);
```

The `tuple_t` data structure is given by:

```
typedef struct tuple_t {
        u_long          Attributes;
        cis_data_t      DesiredTuple;
        u_long          Flags;
        cisdata_t       TupleCode;
        u_long          TupleLink;
        cisdata_t       TupleOffset;
        cisdata_t       TupleDataMax;
        cisdata_t       TupleDataLen;
        cisdata_t       *TupleData;
} tuple_t;
```

`GetFirstTuple` searches a card's CIS for the first tuple code matching `DesiredTuple`. The special code `RETURN_FIRST_TUPLE` will match the first tuple of any kind. If successful, `TupleCode` is set to the code of the first matching tuple found, and `TupleLink` is the address of this tuple in attribute memory.

`GetNextTuple` is like `GetFirstTuple`, except that given a `tuple_t` structure returned by a previous call to `GetFirstTuple` or `GetNextTuple`, it will return the next tuple matching `DesiredTuple`.

The following flags can be specified in `Attributes`:

`TUPLE_RETURN_LINK`

Indicates that link tuples (`CISTPL_LONGLINK_A`, `CISTPL_LONGLINK_C`, `CISTPL_NOLINK`, `CISTPL_LINKTARGET`) should be returned. Normally these tuples are processed silently.

Return codes:

`CS_BAD_HANDLE`

The client handle is invalid.

`CS_OUT_OF_RESOURCE`

Card Services was unable to set up a memory window to map the card's CIS.

`CS_NO_MORE_ITEMS`

There were no tuples matching `DesiredTuple`.

3.4.2 GetTupleData

```
#include "cistpl.h"

int CardServices(GetTupleData, client_handle_t client, tuple_t *tuple);
```

`GetTupleData` extracts a series of data bytes from the specified tuple, which must have been returned by a previous call to `GetFirstTuple` or `GetNextTuple`. A maximum of `TupleDataMax` bytes will be copied into the `TupleData` buffer, starting at an offset of `TupleOffset` bytes. The number of bytes copied is returned in `TupleDataLen`.

Return codes:

`CS_BAD_HANDLE`

The client handle is invalid.

`CS_OUT_OF_RESOURCE`

Card Services was unable to set up a memory window to map the card's CIS.

`CS_NO_MORE_ITEMS`

The tuple does not contain any more data. `TuppleOffset` is greater than or equal to the length of the tuple.

3.4.3 ParseTuple

```
#include "cistpl.h"

int CardServices(ParseTuple, client_handle_t client, tuple_t *tuple, cisparse_t *parse);
```

The `cisparse_t` data structure is given by:

```
typedef union cisparse_t {
        cistpl_device_t        device;
        cistpl_checksum_t      checksum;
        cistpl_longlink_t      longlink;
        cistpl_vers_1_t        version_1;
        cistpl_altstr_t        altstr;
        cistpl_jedec_t         jedec;
        cistpl_manfid_t        manfid;
        cistpl_funcid_t        funcid;
        cistpl_config_t        config;
        cistpl_cftable_entry_t cftable_entry;
        cistpl_device_geo_t    device_geo;
        cistpl_vers_2_t        version_2;
        cistpl_org_t           org;
} cisparse_t;
```

`ParseTuple` interprets tuple data returned by a previous call to `GetTupleData`. The structure returned depends on the type of the parsed tuple. See the `cistpl.h` file for these structure definitions; some of them are quite complex.

Return codes:

`CS_BAD_TUPLE`

An error was encounted during parsing of this tuple. The tuple may be incomplete, or may be formatted incorrectly.

`CS_UNSUPPORTED_FUNCTION`

`ParseTuple` cannot parse the specified tuple type.

3.4.4 ValidateCIS

```
int CardServices(ValidateCIS, client_handle_t client, cisinfo_t *cisinfo);
```

The `cisinfo_t` structure is given by:

```
typedef struct cisinfo_t {
        u_long         Chains;
} cisinfo_t;
```

`ValidateCIS` attempts to verify that a card has a reasonable Card Information Structure. It returns the number of tuples found in `Chains`. If the CIS appears to be uninterpretable, `Chains` will be set to 0.

Return codes:

`CS_BAD_HANDLE`

The client handle is invalid.

`CS_OUT_OF_RESOURCE`

Card Services was unable to set up a memory window to map the card's CIS.

3.5 Memory window control

Each socket can have up to four active memory windows, mapping portions of PCMCIA memory into the host system address space. A PCMCIA device can address at most 16MB of both common and attribute memory. Windows should typically be sized to a power of two, and be aligned on a boundary that is a multiple of the window size in both the host and card address spaces.

A memory window is initialized by a call to `RequestWindow`. Some window attributes can be modified using `ModifyWindow`. The segment of card memory mapped to the window can be modified using `MapMemPage`. And windows are released with `ReleaseWindow`. Unlike almost all other Card Services subfunctions, the memory window functions normally act on `window_handle_t` handles, rather than `client_handle_t` handles.

3.5.1 RequestWindow

```
int CardServices(RequestWindow, client_handle_t *handle, win_req_t *req);
```

The `win_req_t` structure is given by:

```
typedef struct win_req_t {
        u_long          Attributes;
        caddr_t         Base;
        u_long          Size;
        u_long          AccessSpeed;
} win_req_t;
```

`RequestWindow` maps a window of card memory into system memory. On entry, the `handle` parameter should point to a valid client handle. On return, this will be replaced by a `window_handle_t` handle that should be used in subsequent calls to `ModifyWindow`, `MapMemPage`, and `ReleaseWindow`.

The following flags can be specified in `Attributes`:

`WIN_MEMORY_TYPE`

This field can be either `WIN_MEMORY_TYPE_CM` for common memory, or `WIN_MEMORY_TYPE_AM` for attribute memory.

WIN_DATA_WIDTH

Either WIN_DATA_WIDTH_16 for 16-bit accesses, or WIN_DATA_WIDTH_8 for 8-bit access.

WIN_ENABLE

If this is set, the window is turned on.

WIN_USE_WAIT

Specifies that the controller should observe the card's MWAIT signal.

Base specifies the base address of the window in system memory. If NULL, Card Services will set Base to the first available window address. Size specifies the window size in bytes. AccessSpeed specifies the memory access speed, in nanoseconds.

Return codes:

CS_BAD_HANDLE

The client handle is invalid.

CS_NO_CARD

The socket assigned to this client is currently vacant.

CS_BAD_ATTRIBUTE

An unsupported window attribute was requested.

CS_OUT_OF_RESOURCE

The maximum number of memory windows for this socket are already being used.

CS_IN_USE

RequestWindow was unable to find a free window of system memory.

3.5.2 ModifyWindow

```
int CardServices(ModifyWindow, window_handle_t, modwin_t *);
```

The modwin_t structure is given by:

```
typedef struct modwin_t {
        u_long          Attributes;
        u_long          AccessSpeed;
} modwin_t;
```

ModifyWindow modifies the attributes of a window handle returned by a previous call to RequestWindow. The following attributes can be changed:

WIN_MEMORY_TYPE

This field can be either WIN_MEMORY_TYPE_CM for common memory, or WIN_MEMORY_TYPE_AM for attribute memory.

`WIN_DATA_WIDTH`

Either `WIN_DATA_WIDTH_16` for 16-bit accesses, or `WIN_DATA_WIDTH_8` for 8-bit access.

`WIN_ENABLE`

If this is set, the window is turned on.

`AccessSpeed` gives the new memory access speed, in nanoseconds.

Return codes:

`CS_BAD_HANDLE`

The window handle is invalid.

3.5.3 MapMemPage

```
int CardServices(MapMemPage, window_handle_t, memreq_t *);
```

The `memreq_t` structure is given by:

```
typedef struct memreq_t {
        u_long          CardOffset;
        page_t          Page;
} memreq_t;
```

`MapMemPage` sets the address of card memory that is mapped to the base of a memory window to `CardOffset`. The window should have been created by a call to `RequestWindow`. The `Page` parameter is not implemented in this version and should be set to 0.

Return codes:

`CS_BAD_HANDLE`

The window handle is invalid.

`CS_BAD_PAGE`

The `Page` value was non-zero.

3.5.4 ReleaseWindow

```
int CardServices(ReleaseWindow, window_handle_t handle);
```

`ReleaseWindow` releases a memory window previously allocated with `RequestWindow`.

Return codes:

`CS_BAD_HANDLE`

The window handle is invalid.

3.6 Bulk Memory Services

Bulk memory services provide a higher level interface for accessing memory regions than that provided by the memory window services. A client using bulk memory calls does not need to know anything about the underlying memory organization or access methods. The device-specific code is packaged into a special Card Services client called a Memory Technology Driver.

3.6.1 RegisterMTD

```
int CardServices(RegisterMTD, client_handle_t handle, mtd_reg_t *reg);
```

The `mtd_reg_t` data structure is given by:

```
typedef union mtd_reg_t {
        u_long          Attributes;
        u_long          Offset;
        u_long          MediaID;
} mtd_reg_t;
```

`RegisterMTD` informs Card Services that this client MTD will handle requests for a specified memory region. The `Offset` field specifies the starting address of the memory region. The following fields are defined in `Attributes`:

`REGION_TYPE`

Either `REGION_TYPE_CM` for common memory, or `REGION_TYPE_AM` for attribute memory.

The `MediaID` field is recorded by Card Services, and will be passed to the MTD as part of any request that references this memory region.

Once an MTD is bound to a memory region by a call to `RegisterMTD`, it will remain bound until the MTD calls `DeregisterClient`.

Return codes:

`CS_BAD_HANDLE`

The client handle is invalid.

`CS_BAD_OFFSET`

Either the offset does not match a valid memory region for this card, or another MTD has already registered for this region.

3.6.2 GetFirstRegion, GetNextRegion

```
int CardServices(GetFirstRegion, client_handle_t handle, region_info_t *region);
int CardServices(GetNextRegion, client_handle_t handle, region_info_t *region);
```

The region_info_t data structure is given by:

```
typedef union region_info_t {
        u_long          Attributes;
        u_long          CardOffset;
        u_long          RegionSize;
        u_long          AccessSpeed;
        u_long          BlockSize;
        u_long          PartMultiple;
        u_char          JedecMfr, JedecInfo;
        memory_handle_t next;
} region_info_t;
```

GetFirstRegion and GetNextRegion summarize the information in a card's CISTPL_DEVICE, CISTPL_JEDEC, and CISTPL_DEVICE_GEO tuples. CardOffset gives the starting address of a region. RegionSize gives the length of the region in bytes. AccessSpeed gives the device's cycle time in nanoseconds. BlockSize gives the erase block size in bytes, and PartMultiple gives the minimum granularity of partitions on this device, in units of BlockSize. JedecMfr and JedecInfo give the JEDEC identification bytes for this region.

The following fields are defined in Attributes:

REGION_TYPE

Either REGION_TYPE_CM for common memory, or REGION_TYPE_AM for attribute memory.

When these calls are made by an MTD client, only regions that have been bound to this client through calls to BindMTD will be returned.

Return codes:

CS_BAD_HANDLE

The client handle is invalid.

CS_NO_MORE_ITEMS

No more memory regions are defined.

3.6.3 OpenMemory

```
int CardServices(OpenMemory, client_handle_t *handle, open_mem_t *req);
```

The open_mem_t structure is given by:

```
typedef struct open_mem_t {
        u_long          Attributes;
        u_long          Offset;
} open_mem_t;
```

`OpenMemory` is used to obtain a handle for accessing a memory region via the other bulk memory services. The `Offset` field specifies the base address of the region to be accessed. If successful, the client handle argument is replaced by the new memory handle.

The following fields are defined in `Attributes`:

`MEMORY_TYPE`

Either `MEMORY_TYPE_CM` for common memory, or `MEMORY_TYPE_AM` for attribute memory.

`MEMORY_EXCLUSIVE`

Specifies that this client should have exclusive access to this memory region.

Return codes:

`CS_BAD_HANDLE`

The window handle is invalid.

`CS_BAD_OFFSET`

Either the offset does not specify a valid region, or the region does not have an associated MTD to service bulk memory requests.

3.6.4 CloseMemory

```
int CardServices(CloseMemory, memory_handle_t handle);
```

`CloseMemory` releases a memory handle returned by a previous call to `OpenMemory`. A client should release all memory handles before calling `DeregisterClient`.

Return codes:

`CS_BAD_HANDLE`

The memory handle is invalid.

3.6.5 ReadMemory, WriteMemory

```
int CardServices(ReadMemory, memory_handle_t handle mem_op_t *req, caddr_t buf);
int CardServices(WriteMemory, memory_handle_t handle, mem_op_t *req, caddr_t buf);
```

The `mem_io_t` structure is given by:

```
typedef struct mem_op_t {
        u_long          Attributes;
        u_long          Offset;
        u_long          Count;
} mem_op_t;
```

ReadMemory and WriteMemory read from and write to a card memory area defined by the specified memory handle, returned by a previous call to OpenMemory. The Offset field gives the offset of the operation from the start of the card memory region. The Count field gives the number of bytes to be transferred. The buf field points to a host memory buffer to be the destination for a ReadMemory operation, or the source for a WriteMemory operation.

The following fields are defined in Attributes:

MEM_OP_BUFFER

Either MEM_OP_BUFFER_USER if the host buffer is in a user memory segment, or MEM_OP_BUFFER_KERNEL if the host buffer is in kernel memory.

MEM_OP_DISABLE_ERASE

Specifies that a card area should not be erased before it is written.

MEM_OP_VERIFY

Specifies verification of write operations.

Return codes:

CS_BAD_HANDLE

The window handle is invalid.

CS_BAD_OFFSET

The specified card offset is beyond the end of the memory region.

CS_BAD_SIZE

The specified transfer size extends past the end of the memory region.

3.6.6 RegisterEraseQueue

```
int CardServices(RegisterEraseQueue, client_handle_t *handle, eraseq_hdr_t *header);
```

The eraseq_hdr_t structure is given by:

```
typedef struct erase_queue_header_t {
        int             QueueEntryCount;
        eraseq_entry_t  *QueueEntryArray;
} eraseq_hdr_t;
```

This call registers a queue of erase requests with Card Services. An eraseq_handle_t handle will be returned in *handle. When this client calls CheckEraseQueue, Card Services will scan the queue and begin asynchronous processing of any new requests.

The eraseq_entry_t structure is given by:

```
typedef struct eraseq_entry_t {
        memory_handle_t Handle;
        u_char          State;
        u_long          Size;
        u_long          Offset;
        u_long          *Optional;
} eraseq_entry_t;
```

In an erase queue entry, the `Header` field should be a memory handle returned by a previous call to `OpenMemory`. The `State` field indicates the state of the erase request. The following values are defined:

`ERASE_QUEUED`

Set by the client to indicate that this is a new request.

`ERASE_IDLE`

Set by the client to indicate that this entry is not active.

`ERASE_PASSED`

Set by the MTD to indicate successful completion.

`ERASE_FAILED`

Set by the MTD to indicate that the erase failed.

`ERASE_MEDIA_WRPROT`

Indicates that the region is write protected.

`ERASE_NOT_ERASABLE`

Indicates that this region does not support erase operations.

`ERASE_BAD_OFFSET`

Indicates that the erase does not start on an erase block boundary.

`ERASE_BAD_SIZE`

Indicates that the requested erase size is not a multiple of the erase block size.

`ERASE_BAD_SOCKET`

Set by the MTD to indicate that there is no card present.

Additionally, the macro `ERASE_IN_PROGRESS()` will return a true condition for values of `State` that indicate an erase is being processed.

The `Size` field gives the size of the erase request in bytes. The `Offset` field gives the offset from the start of the region. The size and offset should be aligned to erase block boundaries. The `Optional` field is not used by Card Services and may be used by the client driver.

Return codes:

`CS_BAD_HANDLE`

The client handle is invalid.

3.6.7 DeregisterEraseQueue

```
int CardServices(DeregisterEraseQueue, eraseq_handle_t handle);
```

`DeregisterEraseQueue` frees a queue previously registered by a call to `RegisterEraseQueue`. If there are any pending requests in the specified queue, the call will fail.

Return codes:

`CS_BAD_HANDLE`

The erase queue handle is invalid.

`CS_BUSY`

The erase queue has erase requests pending.

3.6.8 CheckEraseQueue

```
int CardServices(CheckEraseQueue, eraseq_handle_t handle);
```

This call notifies Card Services that there are new erase requests in a queue previously registered with `RegisterEraseQueue`.

Typically, a client will initially assign each erase queue entry the state value `ERASE_IDLE`. When new requests are added to the queue, the client will set their states to `ERASE_QUEUED`, and call `CheckEraseQueue`. When the client is notified of an erase completion event, it will check the state field to determine whether the request was successful.

Return codes:

`CS_BAD_HANDLE`

The erase queue handle is invalid.

3.7 Miscellaneous calls

3.7.1 GetCardServicesInfo

```
int CardServices(GetCardServicesInfo, servinfo_t *info);
```

The `servinfo_t` structure is given by:

```
typedef struct servinfo_t {
        char            Signature[2];
        u_long          Count;
        u_long          Revision;
        u_long          CSLevel;
        char            *VendorString;
} servinfo_t;
```

`GetCardServicesInfo` returns revision information about this version of Card Services. `Signature` is set to "CS". `Count` is set to the number of sockets currently configured. `Revision` is set to the revision level of the Card Services package, and `CSLevel` is set to the level of compliance with the PCMCIA standard. These are encoded as BCD numbers. `VendorString` is set to point to an RCS identification string.

This call always succeeds.

3.7.2 AccessConfigurationRegister

```
#include "cisreg.h"

int CardServices(AccessConfigurationRegister, client_handle_t handle, conf_reg_t *reg);
```

The `conf_reg_t` structure is given by:

```
typedef struct conf_reg_t {
        u_long          Action;
        off_t           Offset;
        u_long          Value;
} conf_reg_t;
```

The `Action` parameter can be one of the following:

`CS_READ`

Read the specified configuration register and return `Value`.

`CS_WRITE`

Write `Value` to the specified configuration register.

`AccessConfigurationRegister` either reads or writes the one-byte CIS configuration register at offset `Offset` from the start of the config register area. It can only be used for a socket that has been configured with `RequestConfiguration`.

The following values for `Offset` are defined in `cistpl.h`:

`CISREG_COR`

The Configuration Option Register.

`CISREG_CCSR`

The Card Configuration and Status Register.

`CISREG_PRR`

The Pin Replacement Register.

`CISREG_SCR`

The Socket and Copy Register.

Return codes:

CS_BAD_HANDLE

The client handle is invalid.

CS_BAD_ARGS

The specified `Action` is not supported.

CS_CONFIGURATION_LOCKED

This actually means that the configuration has **not** been locked.

CS_OUT_OF_RESOURCE

Card Services was unable to allocate a memory window to access the card's configuration registers.

3.7.3 AdjustResourceInfo

```
int CardServices(AdjustResourceInfo, client_handle_t handle, adjust_t *adj);
```

The `adjust_t` structure is given by:

```
typedef struct adjust_t {
        u_long          Action;
        u_long          Resource;
        u_long          Attributes;
        union {
                struct memory {
                        caddr_t         Base;
                        u_long          Size;
                } memory;
                struct io {
                        ioaddr_t        BasePort;
                        ioaddr_t        NumPorts;
                        u_long          IOAddrLines;
                } io;
                struct irq {
                        u_long          IRQ;
                } irq;
        } resource;
} adjust_t;
```

`AdjustResourceInfo` is used to tell Card Services what resources may or may not be allocated by PCMCIA devices. The normal Linux resource management systems (the *_region calls for IO ports, interrupt allocation) are respected by Card Services, but this call gives the user another level of control.

The `Action` parameter can have the following values:

`ADD_MANAGED_RESOURCE`

Place the specified resource under Card Services control, so that it may be allocated by PCMCIA devices.

`REMOVE_MANAGED_RESOURCE`

Remove the specified resource from Card Services control.

At initialization time, Card Services assumes that it can use all available interrupts, but IO ports and memory regions must be explicitly enabled with `ADD_MANAGED_RESOURCE`.

The `Resource` parameter can have the following values:

`RES_MEMORY_RANGE`

Specifies a memory range resource, described by `adj->resource.memory`.

`RES_IO_RANGE`

Specifies an IO port resource, described by `adj->resource.io`.

`RES_IRQ`

Specifies an interrupt resource, described by `adj->resource.irq`.

The following flags may be specified in `Attributes`:

`RES_RESERVED`

Indicates that the resource should be reserved for PCMCIA devices that specifically request it. The resource will not be allocated for a device that asks Card Services for any available location. This is not implemented yet.

Return codes:

`CS_UNSUPPORTED_FUNCTION`

The specified `Action` or `Resource` is not supported.

`CS_BAD_BASE`

The specified IO address is out of range.

`CS_BAD_SIZE`

The specified memory or IO window size is out of range.

`CS_IN_USE`

The specified interrupt is currently allocated by a Card Services client.

3.7.4 ReportError

```
int CardServices(ReportError, char *prefix, int func, int ret);
```

`ReportError` generates a kernel error message given a Card Services function code and its return code, with an optional prefix string. For example:

```
CardServices(ReportError, "serial_cs", RequestIO, CS_BAD_HANDLE);
```

would generate the following message:

```
serial_cs: RequestIO: Bad handle
```

This call always succeeds.

4 Card Information Structure Definitions

4.1 CIS Tuple Definitions

The Card Services `ParseTuple` function interprets raw CIS tuple data from a call to `GetTupleData` and returns the tuple contents in a form dependant on the tuple type.

```
#include "cistpl.h"
```

4.1.1 CISTPL_CHECKSUM

The `cistpl_checksum_t` structure is given by:

```
typedef struct cistpl_checksum_t {
        u_short         addr;
        u_short         len;
        u_char          sum;
} cistpl_checksum_t;
```

4.1.2 CISTPL_LONGLINK_A, CISTPL_LONGLINK_C, CISTPL_LINKTARGET, CISTPL_NOLINK

The `cistpl_longlink_t` structure is given by:

```
typedef struct cistpl_longlink_t {
        u_long          addr;
} cistpl_longlink_t;
```

These tuples are pointers to additional chains of CIS tuples, either in attribute or common memory. Each CIS tuple chain can have at most one long link. `CISTPL_LONGLINK_A` tuples point to attribute memory, and `CISTPL_LONGLINK_C` tuples point to common memory. The standard CIS chain starting at address 0 in attribute memory has an implied long link to address 0 in common memory. A `CISTPL_NOLINK` tuple can be used to cancel this default link.

The first tuple of a chain pointed to by a long link must be a `CISTPL_LINKTARGET`. The CS tuple handling code will automatically follow long links and verify link targets; these tuples are normally invisible unless the `TUPLE_RETURN_LINK` attribute is specified in `GetNextTuple`.

4.1.3 CISTPL_DEVICE, CISTPL_DEVICE_A

The `cistpl_device_t` structure is given by:

```
typedef struct cistpl_device_t {
        int             ndev;
        struct device_info {
                u_char          type;
                u_char          wp;
                u_long          speed;
                u_long          size;
        } dev[CISTPL_MAX_DEVICES];
} cistpl_device_t;
```

The `CISTPL_DEVICE` tuple describes address regions in a card's common memory. The `CISTPL_DEVICE_A` tuple describes regions in attribute memory. The `type` flag indicates the type of memory device for this region. The `wp` flag indicates if this region is write protected. The `speed` field is in nanoseconds, and `size` is in bytes. Address regions are assumed to be ordered consecutively starting with address 0. The following device types are defined:

`CISTPL_DTYPE_NULL`

Specifies that there is no device, or a "hole" in the card address space.

`CISTPL_DTYPE_ROM`

Masked ROM

`CISTPL_DTYPE_OTPROM`

One-type programmable ROM.

`CISTPL_DTYPE_EPROM`

UV erasable PROM.

`CISTPL_DTYPE_EEPROM`

Electrically erasable PROM.

`CISTPL_DTYPE_FLASH`

Flash EPROM.

`CISTPL_DTYPE_SRAM`

Static or non-volatile RAM.

`CISTPL_DTYPE_DRAM`

Dynamic or volatile RAM.

`CISTPL_DTYPE_FUNCSPEC`

Specifies a function-specific device, such as a memory-mapped IO device or buffer, as opposed to general purpose storage.

`CISTPL_DTYPE_EXTEND`

Specifies an extended device type. This type is reserved for future use.

4.1.4 CISTPL_VERS_1

The `cistpl_vers_1_t` structure is given by:

```
typedef struct cistpl_vers_1_t {
        u_char          major;
        u_char          minor;
        int             ns;
        int             ofs[CISTPL_VERS_1_MAX_PROD_STRINGS];
        char            str[254];
} cistpl_vers_1_t;
```

The `ns` field specifies the number of product information strings in the tuple. The string data is contained in the `str` array. Each string is null terminated, and `ofs` gives the offset to the start of each string.

4.1.5 CISTPL_ALTSTR

The `cistpl_altstr_t` structure is given by:

```
typedef struct cistpl_altstr_t {
        int             ns;
        int             ofs[CISTPL_ALTSTR_MAX_STRINGS];
        char            str[254];
} cistpl_altstr_t;
```

The `ns` field specifies the number of alternate language strings in the tuple. The string data is contained in the `str` array. Each string is null terminated, and `ofs` gives the offset to the start of each string.

4.1.6 CISTPL_JEDEC_C, CISTPL_JEDEC_A

The `cistpl_jedec_t` structure is given by:

```
typedef struct cistpl_jedec_t {
        int             nid;
        struct jedec_id {
                u_char  mfr;
                u_char  info;
        } id[CISTPL_MAX_DEVICES];
} cistpl_jedec_t;
```

JEDEC identifiers describe the specific device type used to implement a region of PCMCIA memory. The `nid` field specifies the number of JEDEC identifiers in the tuple. There should be a one-to-one correspondence between JEDEC identifiers and device descriptions in the corresponding `CISTPL_DEVICE` tuple.

4.1.7 CISTPL_CONFIG

The `cistpl_config_t` structure is given by:

```
typedef struct cistpl_config_t {
        u_char          last_idx;
        u_long          base;
        u_long          rmask[4];
        u_char          subtuples;
} cistpl_config_t;
```

The `last_idx` field gives the index of the highest numbered configuration table entry. The `base` field gives the offset of a card's configuration registers in attribute memory. The `rmask` array is a series of bit masks indicating which configuration registers are present. Bit 0 of rmask0 is for the COR, bit 1 is for the CCSR, and so on. The `subtuples` field gives the number of bytes of subtuples following the normal tuple contents.

4.1.8 CISTPL_CFTABLE_ENTRY

The `cistpl_cftable_entry_t` structure is given by:

```
typedef struct cistpl_cftable_entry_t {
        u_char          index;
        u_char          flags;
        u_char          interface;
        cistpl_power_t  vcc, vpp1, vpp2;
        cistpl_timing_t timing;
        cistpl_io_t     io;
        cistpl_irq_t    irq;
        cistpl_mem_t    mem;
        u_char          subtuples;
} cistpl_cftable_entry_t;
```

A `CISTPL_CFTABLE_ENTRY` structure describes a complete operating mode for a card. Many sections are optional. The `index` field gives the configuration index for this operating mode; writing this value to the card's Configuration Option Register selects this mode. The following fields are defined in `flags`:

CISTPL_CFTABLE_DEFAULT

Specifies that this is the default configuration table entry.

CISTPL_CFTABLE_BVDS

Specifies that this configuration implements the BVD1 and BVD2 signals in the Pin Replacement Register.

CISTPL_CFTABLE_WP

Specifies that this configuration implements the write protect signal in the Pin Replacement Register.

CISTPL_CFTABLE_RDYBSY

Specifies that this configuration implements the Ready/Busy signal in the Pin Replacement Register.

CISTPL_CFTABLE_MWAIT

Specifies that the WAIT signal should be observed during memory access cycles.

CISTPL_CFTABLE_AUDIO

Specifies that this configuration generates an audio signal that can be routed to the host system speaker.

CISTPL_CFTABLE_READONLY

Specifies that the card has a memory region that is read-only in this configuration.

CISTPL_CFTABLE_PWRDOWN

Specifies that this configuration supports a power down mode, via the Card Configuration and Status Register.

The cistpl_power_t structure is given by:

```
typedef struct cistpl_power_t {
        u_char          present;
        u_char          flags;
        u_long          param[7];
} cistpl_power_t;
```

The present field is bit mapped and indicates which parameters are present for this power signal. The following indices are defined:

CISTPL_POWER_VNOM

The nominal supply voltage.

CISTPL_POWER_VMIN

The minimum supply voltage.

CISTPL_POWER_VMAX

The maximum supply voltage.

CISTPL_POWER_ISTATIC

The continuous supply current required.

CISTPL_POWER_IAVG

The maximum current averaged over one second.

CISTPL_POWER_IPEAK

The maximum current averaged over 10 ms.

CISTPL_POWER_IDOWN

The current required in power down mode.

Voltages are given in units of 10 microvolts. Currents are given in units of 100 nanoamperes.

The `cistpl_timing_t` structure is given by:

```
typedef cistpl_timing_t {
        u_long          wait, waitscale;
        u_long          ready, rdyscale;
        u_long          reserved, rsvscale;
} cistpl_timing_t;
```

Each time consists of a base time in nanoseconds, and a scale multiplier. Unspecified times have values of 0.

The `cistpl_io_t` structure is given by:

```
typedef struct cistpl_io_t {
        u_char          flags;
        int             nwin;
        struct {
                u_long          base;
                u_long          len;
        } win[CISTPL_IO_MAX_WIN;
} cistpl_io_t;
```

The number of IO windows is given by `nwin`. Each window is described by a base address, `base`, and a length in bytes, `len`. The following bit fields are defined in `flags`:

CISTPL_IO_LINES_MASK

The number of IO lines decoded by this card.

CISTPL_IO_8BIT

Indicates that the card supports split 8-bit accesses to 16-bit IO registers.

CISTPL_IO_16BIT

Indicates that the card supports full 16-bit accesses to IO registers.

The `cistpl_irq_t` structure is given by:

```
typedef struct cistpl_irq_t {
        u_long          IRQInfo1;
        u_long          IRQInfo2;
} cistpl_irq_t;
```

The following bit fields are defined in `IRQInfo1`:

`IRQ_MASK`

A specific interrupt number that this card should use.

`IRQ_NMI_ID`, `IRQ_IOCK_ID`, `IRQ_BERR_ID`, `IRQ_VEND_ID`

When `IRQ_INFO2_VALID` is set, these indicate if a corresponding special interrupt signal may be assigned to this card. The four flags are for the non-maskable, IO check, bus error, and vendor specific interrupts.

`IRQ_INFO2_VALID`

Indicates that `IRQInfo2` contains a valid bit mask of allowed interrupt request numbers.

`IRQ_LEVEL_ID`

Indicates that the card supports level mode interrupts.

`IRQ_PULSE_ID`

Indicates that the card supports pulse mode interrupts.

`IRQ_SHARE_ID`

Indicates that the card supports sharing interrupts.

If `IRQInfo1` is 0, then no interrupt information is available.

The `cistpl_mem_t` structure is given by:

```
typedef struct cistpl_mem_t {
        u_char          nwin;
        struct {
                u_long          len;
                u_long          card_addr;
                caddr_t         host_addr;
        } win[CISTPL_MEM_MAX_WIN;
} cistpl_mem_t;
```

The number of memory windows is given by `nwin`. Each window is described by an address in the card memory space, `card_addr`, an address in the host memory space, `host_addr`, and a length in bytes, `len`. If the host address is 0, the position of the window is arbitrary.

4.1.9 CISTPL_MANFID

The `cistpl_manfid_t` structure is given by:

```
typedef struct cistpl_manfid_t {
        u_short         manf;
        u_short         card;
} cistpl_manfid_t;
```

The `manf` field identifies the card manufacturer. The `card` field is chosen by the vendor and should identify the card type and model.

4.1.10 CISTPL_FUNCID

The `cistpl_funcid_t` structure is given by:

```
typedef struct cistpl_funcid_t {
        u_char          func;
        u_char          sysinit;
} cistpl_funcid_t;
```

The `func` field identifies the card function. The `sysinit` field contains several bit-mapped flags describing how the card should be configured at boot time.

The following functions are defined:

`CISTPL_FUNCID_MULTI`

A multi-function card.

`CISTPL_FUNCID_MEMORY`

A simple memory device.

`CISTPL_FUNCID_SERIAL`

A serial port or modem device.

`CISTPL_FUNCID_PARALLEL`

A parallel port device.

`CISTPL_FUNCID_FIXED`

A fixed disk device.

`CISTPL_FUNCID_VIDEO`

A video interface.

`CISTPL_FUNCID_NETWORK`

A network adapter.

`CISTPL_FUNCID_AIMS`

An auto-incrementing mass storage device.

The following flags are defined in **`sysinit`**:

`CISTPL_SYSINIT_POST`

Indicates that the system should attempt to configure this card during its power-on initialization.

`CISTPL_SYSINIT_ROM`

Indicates that the card contains a system expansion ROM that should be configured at boot time.

4.1.11 CISTPL_DEVICE_GEO

The **`cistpl_device_geo_t`** structure is given by:

```
typedef struct cistpl_device_geo_t {
        int             ngeo;
        struct geo {
                u_char          buswidth;
                u_long          erase_block;
                u_long          read_block;
                u_long          write_block;
                u_long          partition;
                u_long          interleave;
        } geo[CISTPL_MAX_DEVICES];
} cistpl_device_geo_t;
```

The **`erase_block`**, **`read_block`**, and **`write_block`** sizes are in units of **`buswidth`** bytes times **`interleave`**. The **`partition`** size is in units of **`erase_block`**.

4.1.12 CISTPL_VERS_2

The **`cistpl_vers_2_t`** structure is given by:

```
typedef struct cistpl_vers_2_t {
        u_char          vers;
        u_char          comply;
        u_short         dindex;
        u_char          vspec8, vspec9;
        u_char          nhdr;
        int             vendor, info;
        char            str[244];
} cistpl_vers_2_t;
```

The `vers` field should always be 0. The `comply` field indicates the degree of standard compliance and should also be 0. The `dindex` field reserves the specified number of bytes at the start of common memory. The `vspec8` and `vspec9` fields may contain vendor-specific information. The `nhdr` field gives the number of copies of the CIS that are present on this card. The `str` array contains two strings: a vendor name, and an informational message describing the card. The offset of the vendor string is given by `vendor`, and the offset of the product info string is in `info`.

4.1.13 CISTPL_ORG

The `cistpl_org_t` structure is given by:

```
typedef struct cistpl_org_t {
        u_char          data_org;
        char            desc[30];
```

This tuple describes the data organization of a memory partition. The following values are defined for `data_org`:

`CISTPL_ORG_FS`

The partition contains a filesystem.

`CISTPL_ORG_APPSPEC`

The partition is in an application specific format.

`CISTPL_ORG_XIP`

The partition follows the Execute-In-Place specification.

The `desc` field gives a text description of the data organization.

4.2 CIS configuration registers

The PCMCIA standard defines a few standard configuration registers located in a card's attribute memory space. A card's `CONFIG` tuple specifies which of these registers are implemented. Programs using these definitions should include:

```
#include "cisreg.h"
```

4.2.1 Configuration Option Register

This register should be present for virtually all IO cards. Writing to this register selects a configuration table entry and enables a card's IO functions.

The following bit fields are defined:

`COR_CONFIG_MASK`

Specifies the configuration table index describing the card's current operating mode.

`COR_LEVEL_REQ`

Specifies that the card should generate level mode (edge-triggered) interrupts, the default.

`COR_SOFT_RESET`

Setting this bit performs a "soft" reset operation. Drivers should use the `ResetCard` call to reset a card, rather than writing directly to this register.

4.2.2 Card Configuration and Status Register

The following bit fields are defined:

`CCSR_INTR_PENDING`

Signals that the card is currently asserting an interrupt request. This signal can be used to support interrupt sharing.

`CCSR_POWER_DOWN`

Setting this bit signals that the card should enter a power down state.

`CCSR_AUDIO_ENA`

Specifies that the card's audio output should be enabled.

`CCSR_IOIS8`

This is used by the host to indicate that it can only perform 8-bit IO operations and that 16-bit accesses will be carried out as two 8-bit accesses.

`CCSR_SIGCHG_ENA`

This indicates to the card that it should use the `SIGCHG` signal to indicate changes in the `WP`, `READY`, `BVD1`, and `BVD2` signals.

`CCSR_CHANGED`

This bit signals to the host that one of the signals in the Pin Replacement Register has changed state.

4.2.3 Pin Replacement Register

Signals in this register replace signals that are not available when a socket is operating in memory and IO mode. An IO card will normally assert the `SIGCHG` signal to indicate that one of these signals has changed state, then a driver can poll this register to find out specifically what happened.

The following bit fields are defined:

`PRR_WP_STATUS`

The current state of the write protect signal.

`PRR_READY_STATUS`

The current state of the ready signal.

`PRR_BVD2_STATUS`

The current state of the battery warn signal.

`PRR_BVD1_STATUS`

The current state of the battery dead signal.

`PRR_WP_EVENT`

Indicates that the write protect signal has changed state since the PRR register was last read.

`PRR_READY_EVENT`

Indicates that the ready signal has changed state since the PRR register was last read.

`PRR_BVD2_EVENT`

Indicates that the battery warn signal has changed state since the PRR register was last read.

`PRR_BVD1_EVENT`

Indicates that the battery dead signal has changed state since the PRR register was last read.

This register can also be written. In this case, the `STATUS` bits act as a mask; if a `STATUS` bit is set, the corresponding `EVENT` bit is updated by the write.

4.2.4 Socket and Copy Register

This register is used when several identical cards may be set up to share the same range of IO ports, to emulate an ISA bus card that would control several devices. For example, an ISA hard drive controller might control several drives, selectable by writing a drive number to an IO port. For several PCMCIA drives to emulate this controller interface, each needs to "know" which drive it is, so that it can identify which IO operations are intended for it.

The following bit fields are defined:

`SCR_SOCKET_NUM`

This should indicate the socket number in which the card is located.

`SCR_COPY_NUM`

If several identical cards are installed in a system, this field should be set to a unique number identifying which of the identical cards this is.

5 Card Services Event Handling

Card Services events have several sources:

- Card status changes reported by the low-level socket drivers.
- Artificial events generated by Card Services itself.
- Advanced Power Management (APM) events.
- Events generated by other Card Services clients.

Socket driver events may be either interrupt-driven or polled.

5.1 Event handler operations

When Card Services recognizes that an event has occurred, it checks the event mask of each client to determine which clients should receive an event notification. When a client registers with Card Services, it specifies an event handler callback function. This handler should have the form:

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

The `priority` parameter is set to either `CS_EVENT_PRI_LOW` for ordinary events, or `CS_EVENT_PRI_HIGH` for events that require an immediate response. The only high priority event is `CS_EVENT_CARD_REMOVAL`. A client event handler should process this event as efficiently as possible so that Card Services can quickly notify other clients.

The `event_callback_args_t` structure is given by:

```
typedef struct event_callback_args_t {
        client_handle_t         client_handle;
        void                    *info;
        void                    *mtdrequest;
        void                    *buffer;
        void                    *misc;
        void                    *client_data;
} event_callback_args_t;
```

The `client_handle` member is set to the handle of the client whose socket was responsible for the event. This is useful if a driver is bound to several sockets. The `info` field is currently only used to return an exit status from a call to `ResetCard`. The `client_data` field may be used by a driver to point to a local data structure associated with this device. The remaining fields are currently unused.

5.2 Event descriptions

CS_EVENT_CARD_INSERTION

This event signals that a card has been inserted. If a driver is bound to an already occupied socket, Card Services will send the driver an artificial insertion event.

CS_EVENT_CARD_REMOVAL

This event signals that a card has been removed. This event should be handled with minimum delay so that Card Services can notify all clients as quickly as possible.

CS_EVENT_BATTERY_LOW

This event signals a change of state of the "battery low" signal.

CS_EVENT_BATTERY_DEAD

This event signals a change of state of the "battery dead" signal.

CS_EVENT_READY_CHANGE

This event signals a change of state of the "ready" signal.

CS_EVENT_WRITE_PROTECT

This event signals a change of state of the "write protect" signal.

CS_EVENT_REGISTRATION_COMPLETE

This event is sent to a driver after a successful call to `RegisterClient`.

CS_EVENT_RESET_REQUEST

This event is sent when a client calls `ResetCard`. An event handler can veto the reset operation by returning failure.

CS_EVENT_RESET_PHYSICAL

This is sent to all clients just before a reset signal is sent to a card.

CS_EVENT_CARD_RESET

This event signals that a reset operation is finished. The success or failure of the reset should be determined using `GetStatus`.

CS_EVENT_RESET_COMPLETE

This event is sent to a client that has called `ResetCard` to signal the end of reset processing.

CS_EVENT_PM_SUSPEND

This event signals that Card Services has received either a user initiated or APM suspend request. An event handler can veto the suspend by returning failure.

CS_EVENT_PM_RESUME

This signals that the system is back on line after a suspend/resume cycle.

CS_EVENT_MTD_REQUEST

This is used to initiate an MTD memory operation. A description of the request is passed in the mtdrequest field of the callback arguments. A host buffer address may be passed in buffer.

CS_EVENT_ERASE_COMPLETE

This is used to signal a client that a queued erase operation has completed. A pointer to the erase queue entry is returned in the info field of the callback arguments.

5.3 Client driver event handling responsibilities

A client driver should respond to CS_EVENT_CARD_INSERTION and CS_EVENT_CARD_REMOVAL events by configuring and un-configuring the socket. Because card removal is a high priority event, the driver should immediately block IO to the socket, perhaps by setting a flag in a device structure, and schedule all other shutdown processing to happen later using a timer interrupt.

When a CS_EVENT_PM_RESET_REQUEST event is received, a driver should block IO and release a locked socket configuration. When a CS_EVENT_CARD_RESET is received, a driver should restore the socket configuration and unblock IO.

A CS_EVENT_PM_SUSPEND event should be handled somewhat like a CS_EVENT_PM_RESET_REQUEST event, in that IO should be blocked and the socket configuration should be released. When a CS_EVENT_PM_RESUME event is received, a driver can expect a card to be ready to be reconfigured, similar to when a CS_EVENT_CARD_RESET event is received.

6 Memory Technology Drivers

A Memory Technology Driver ("MTD") is used by Card Services to implement bulk memory services for a particular type of memory device. An MTD should register as a normal Card Services client with a call to RegisterClient. When it receives a card insertion event, it should use GetFirstRegion and GetNextRegion to identify memory regions that it will administer. Then, it should use RegisterMTD to take control of these regions. MTD read, write, copy, and erase requests are packaged into CS_EVENT_MTD_REQUEST events by Card Services, and passed to the MTD's event handler for processing.

6.1 MTD request handling

An MTD receives requests from Card Services in the form of CS_EVENT_MTD_REQUEST events. Card Services passes a description of the request in the mtdrequest field of the event callback arguments. For requests that transfer data to or from the host, the host buffer address is passed in the buffer field.

The mtd_request_t structure is given by:

```
typedef struct mtd_request_t {
        u_long          SrcCardOffset;
        u_long          DestCardOffset;
```

```
        u_long          TransferLength;
        u_long          Function;
        u_long          MediaID;
        u_long          Status;
        u_long          Timeout;
} mtd_request_t;
```

The `Function` field is bit mapped and describes the action to be performed by this request:

`MTD_REQ_ACTION`

Either `MTD_REQ_ERASE`, `MTD_REQ_READ`, `MTD_REQ_WRITE`, or `MTD_REQ_COPY`.

`MTD_REQ_NOERASE`

For a write command that is sized and aligned on erase block boundaries, this specifies that no erase should be performed.

`MTD_REQ_VERIFY`

Specifies that writes should be verified.

`MTD_REQ_READY`

Indicates that this request is a retry of a previously request that was delayed until the card asserted `READY`.

`MTD_REQ_TIMEOUT`

Indicates that this request is a retry of a previously request that was delayed by a timeout.

`MTD_REQ_FIRST`

Indicates that this request is the first in a series of requests.

`MTD_REQ_LAST`

Indicates that this request is the last of a series of requests.

`MTD_REQ_KERNEL`

Indicates that the host buffer for a read or write command is located in kernel memory, as opposed to user memory.

The `MediaID` field is the value specified in the `RegisterMTD` request for this region. The `Status` field is used by the MTD when it is unable to satisfy a request because a device is busy. MTD requests normally run without blocking. If an MTD request would block, it should return an error code of `CS_BUSY`, and set `Status` to one of the have the following values:

`MTD_WAITREQ`

Specifies that the request should be retried after another MTD request currently in progress completes.

`MTD_WAITTIMER`

Specifies that the request should be continued after the time specified in the `timeout` field.

MTD_WAITRDY

Specifies that the request should be continued when the card signals READY, or when the time specified in Timeout elapses, whichever happens first.

MTD_WAITPOWER

Specifies that the request should be retried after something happens that affects power availability to the socket.

For MTD_WAITTIMER and MTD_WAITRDY, the Timeout field will specify the timeout interval in milliseconds.

6.2 MTD helper functions

Since an MTD processes requests generated by Card Services, there may be some restrictions on the sorts of Card Services calls that can be safely made from the MTD event handler. The MTD helper functions provide a limited set of special services that may be needed by an MTD but would be tricky to implement using the normal Card Services calls. In the Linux PCMCIA implementation, most CS calls can be safely made from an MTD event handler, but the MTD helper interface is included for compatibility.

```
#include "cs_types.h"
#include "cs.h"
#include "bulkmem.h"

int MTDHelperEntry(int subfunc, void *arg1, void *arg2);
```

6.2.1 MTDModifyWindow

```
int MTDHelperEntry(MTDModifyWindow, memory_handle_t handle, mtd_mod_req_t *mod);
```

The mtd_mod_req_t structure is give by:

```
typedef struct mtd_mod_req_t {
        u_long          Attributes;
        u_long          AccessSpeed;
        u_long          CardOffset;
} mtd_mod_req_t;
```

MTDModifyWindow is essentially equivalent to using the normal ModifyWindow and MapMemPage calls.

The following flags can be specified in Attributes:

WIN_MEMORY_TYPE

Either WIN_MEMORY_TYPE_CM for common memory, or WIN_MEMORY_TYPE_AM for attribute memory.

WIN_USE_WAIT

Specifies that the controller should observe the card's MWAIT signal.

A window configured with `MTDModifyWindow` will always be enabled, and have a 16 bit data width.

Return codes:

`CS_BAD_HANDLE`

The memory handle is invalid.

6.2.2 MTDSetVpp

```
int MTDHelperEntry(MTDSetVpp, client_handle_t client, mtd_vpp_req_t *req);

typedef struct mtd_vpp_req_t {
        u_char          Vpp1, Vpp2;
} mtd_vpp_req_t;
```

`MTDSetVpp` changes the programming voltage for a socket. `Vpp1` and `Vpp2` should be given in units of 1/10 volt. Currently, `Vpp1` should always equal `Vpp2`.

Return codes:

`CS_BAD_HANDLE`

The client handle is invalid.

`CS_BAD_VPP`

The specified Vpp is not available, or Vpp1 does not equal Vpp2.

6.2.3 MTDRDYMask

```
int MTDHelperEntry(MTDRDYMask, client_handle_t client, mtd_rdy_req_t *req);

typedef struct mtd_rdy_req_t {
        u_long          Mask;
} mtd_rdy_req_t;
```

`MTDRDYMask` selects whether or not `CS_EVENT_READY_CHANGE` events will be enabled. The client should already have indicated to Card Services that it should receive ready change events, via a call to either `RegisterClient` or `SetEventMask`. Ready change events will be enabled if the `CS_EVENT_READY_CHANGE` bit is set in the `Mask` argument.

Return codes:

`CS_BAD_HANDLE`

The client handle is invalid.

7 Driver Services interface

Driver Services provides a link between Card Services client drivers and user mode utilities like the **cardmgr** daemon. It is a sort of Card Services "super-client". Driver Services uses the **BindDevice** function to link other client drivers with their corresponding PCMCIA cards. Unlike other clients, Driver Services remains permanently bound to all sockets as cards are inserted and removed.

7.1 Interface to other client drivers

Driver Services keeps track of all client drivers that are installed and ready to attach to a socket. Client drivers need to have entry points for creating and deleting device "instances", where one device instance is everything needed to manage one PCMCIA card.

Each client driver is identified by a unique 16-character tag that has the special type **dev_info_t**, defined in **cs_types.h**. Each device instance is described by a **dev_link_t** structure.

7.1.1 The dev_link_t structure

The **dev_link_t** data structure is given by:

```
#include "ds.h"

typedef struct dev_link_t {
        char                    dev_name[8];
        u_char                  major, minor;
        u_long                  state;
        u_long                  open;
        struct wait_queue       *pending
        struct timer_list       release
        client_handle_t         handle;
        io_req_t                io;
        irq_req_t               irq;
        config_req_t            conf;
        window_handle_t         win;
        void                    *priv;
        struct dev_link_t       *next;
} dev_link_t;
```

The **dev_name** field should be filled in by the driver with a device file name for accessing this device, if appropriate. For example, the **serial_cs** driver uses names like "**cua1**". The **major** and **minor** fields give major and minor device numbers for accessing this device. Driver Services relays these fields to user mode programs via the DS_GET_DEVICE_INFO ioctl.

The **state** field should be used to keep track of the current device state. The following flags are defined:

DEV_PRESENT

Indicates that the card is present. This bit should be set and cleared by the driver's event handler in response to card insertion and removal events.

`DEV_CONFIG`

Indicates that the card is configured for use.

`DEV_CONFIG_PENDING`

Indicates that configuration is in progress.

`DEV_SUSPEND`

Indicates that the card is suspended.

`DEV_BUSY`

Indicates that an IO operation is in progress. This bit may be used as an interlock to prevent access conflicts.

`DEV_STALE_CONFIG`

For some drivers, when a running card is ejected, the socket should not be unconfigured until any devices corresponding to this card are closed. This flag indicates that the socket should be unconfigured when the device is closed.

`DEV_STALE_LINK`

A driver instance should not be deleted until all its PCMCIA resources are released. This flag indicates that this driver instance should be freed as soon as the socket is unconfigured.

The `open` field is a usage count for this device. The device should only be freed when the open count is zero. The `pending` field can be used to manage a queue of processes waiting to use the device.

The `release` field is used to schedule device shutdown processing when a card is ejected. A card removal event needs to be handled at high priority, so a driver's event handler will typically deal with an eject by resetting the `DEV_PRESENT` bit in the device state, then scheduling the shutdown processing to run at a later time.

The `handle`, `io`, `irq`, `conf`, and `win` fields comprise all the normal PCMCIA data structures needed to configure an ordinary PCMCIA IO card.

The `priv` field can be used for any sort of private data structure needed to manage the device. The `next` field can be used to build linked lists of `dev_link_t` structures, for drivers that can handle multiple instances.

7.1.2 register_pcmcia_driver

```
int register_pcmcia_driver(dev_info_t *dev_info,
                           dev_link_t *(*attach)(void),
                           void (*detach)(dev_link_t *));
```

`register_pcmcia_driver` informs Driver Services that a client driver is present and ready to be bound to sockets. When Driver Services receives a `DS_BIND_REQUEST` ioctl that matches this driver's `dev_info` string, it will call the driver's `attach()` entry point. When it gets a `DS_UNBIND_REQUEST` ioctl, it will call `detach()`.

7.1.3 unregister_pcmcia_driver

```
int unregister_pcmcia_driver(dev_info_t *dev_info);
```

This informs Driver Services that it should no longer bind sockets to the specified client driver.

7.2 Interface to user mode PCMCIA utilities

Driver Services creates a pseudo-device for communicating with user mode PCMCIA utilities. The major number of the device is chosen dynamically, and PCMCIA utilities should read `/proc/devices` to determine it. Minor device numbers correspond to socket numbers, starting with 0.

Only one process is allowed to open a socket for read/write access. Other processes can open the socket in read-only mode. A read-only connection to Driver Services can perform a subset of `ioctl` calls. A read/write connection can issue all `ioctl` calls, and can also receive Card Services event notifications.

7.2.1 Card Services event notifications

Driver Services implements `read()` and `select()` functions for event notification. Reading from a pcmcia device returns an unsigned long value containing all the events received by Driver Services since the previous `read()`. If no events have been received, the call will block until the next event. A `select()` call can be used to monitor several sockets for new events.

The following events are monitored by Driver Services: `CS_EVENT_CARD_INSERTION`, `CS_EVENT_CARD_REMOVAL`, `CS_EVENT_RESET_PHYSICAL`, `CS_EVENT_CARD_RESET`, and `CS_EVENT_RESET_COMPLETE`.

7.2.2 `ioctl` descriptions

Most Driver Services `ioctl` operations directly map to Card Services functions. An ioctl call has the form:

```
int ioctl(int fd, int cmd, ds_ioctl_arg_t *arg);
```

The ds_ioctl_arg_t structure is given by:

```
typedef union ds_ioctl_arg_t {
        servinfo_t      servinfo;
        adjust_t        adjust;
        config_t        config;
        tuple_t         tuple;
        tuple_parse_t   tuple_parse;
        client_req_t    client_req;
        status_t        status;
        conf_reg_t      conf_reg;
        cisinfo_t       cisinfo;
        region_info_t   region;
        bind_info_t     bind_info;
```

```
        mtd_info_t      mtd_info;
} ds_ioctl_arg_t;
```

The following `ioctl` commands execute the corresponding Card Services function:

`DS_GET_CARD_SERVICES_INFO`

Calls `CardServices(GetCardServicesInfo, ..., &arg->servinfo)`.

`DS_ADJUST_RESOURCE_INFO`

Calls `CardServices(AdjustResourceInfo, ..., &arg->adjust)`.

`DS_GET_CONFIGURATION_INFO`

Calls `CardServices(GetConfigurationInfo, ..., &arg->config)`.

`DS_GET_FIRST_TUPLE`

Calls `CardServices(GetFirstTuple, ..., &arg->tuple)`.

`DS_GET_NEXT_TUPLE`

Calls `CardServices(GetNextTuple, ..., &arg->tuple)`.

`DS_GET_TUPLE_DATA`

Calls `CardServices(GetTupleData, ..., &arg->tuple_parse.tuple)`. The tuple data is returned in `arg->tuple_parse.data`.

`DS_PARSE_TUPLE`

Calls `CardServices(ParseTuple, ..., &arg->tuple_parse.tuple, &arg->tuple_parse.parse)`.

`DS_RESET_CARD`

Calls `CardServices(ResetCard, ...)`.

`DS_GET_STATUS`

Calls `CardServices(GetStatus, ..., &arg->status)`.

`DS_ACCESS_CONFIGURATION_REGISTER`

Calls `CardServices(AccessConfigurationRegister, ..., &arg->conf_reg)`.

`DS_VALIDATE_CIS`

Calls `CardServices(ValidateCIS, ..., &arg->cisinfo)`.

`DS_SUSPEND_CARD`

Calls `CardServices(SuspendCard, ...)`.

`DS_RESUME_CARD`

Calls `CardServices(ResumeCard, ...)`.

`DS_EJECT_CARD`

Calls `CardServices(EjectCard, ...)`.

DS_INSERT_CARD

Calls `CardServices(InsertCard, ...)`.

DS_GET_FIRST_REGION

Calls `CardServices(GetFirstRegion, ..., &arg->region)`.

DS_GET_NEXT_REGION

Calls `CardServices(GetNextRegion, ..., &arg->region)`.

The following `ioctl` commands invoke special Driver Services functions. They act on `bind_info_t` structures:

```
typedef struct bind_info_t {
        dev_info_t             dev_info;
        struct dev_info_t      *instance;
        char                   name[8];
        u_char                 major, minor;
        void                   *next;
} bind_info_t;
```

DS_BIND_REQUEST

This call connects a socket to a client driver. The specified device ID `dev_info` is looked up in the list of registered drivers. If found, the driver is bound to this socket using the `BindDevice` call. Then, Driver Services calls the client driver's `attach()` entry point to create a device instance. The new `dev_link_t` pointer is returned in `instance`.

DS_GET_DEVICE_INFO

This call retrieves the `dev_name`, `major`, and `minor` entries from the `dev_link_t` structure pointed to by `instance`.

DS_UNBIND_REQUEST

This call calls the `detach()` function for the specified driver and instance, shutting down this device.

Finally, the `DS_BIND_MTD` request takes an argument of type `mtd_info_t`:

```
typedef struct mtd_info_t {
        dev_info_t      dev_info;
        u_long          Attributes;
        u_long          CardOffset;
} mtd_info_t;
```

This call associates an MTD identified by `dev_info` with a memory region described by `Attributes` and `CardOffset`, which have the same meanings as in the Card Services `BindMTD` call.

8 Anatomy of a Card Services Client Driver

Each release of the Linux PCMCIA package comes with a well-commented "skeleton" client driver that should be used as a starting point for writing a new driver. Look for it in `modules/skeleton.c`.

8.1 Module initialization and cleanup

All loadable modules must supply `init_module()` and `cleanup_module()` functions, which are invoked by the module support code when the module is installed and removed. A PCMCIA client driver's init function should register the driver with Driver Services, via the `register_pcmcia_driver()` call. The cleanup function should use `unregister_pcmcia_driver()` to unregister with Driver Services. Depending on the driver, the cleanup function may also need to free any device structures that still exist at shutdown time.

8.2 The *_attach() and *_detach() functions

The `*_attach()` entry point is responsible for creating an "instance" of the driver, setting up any data structures needed to manage one PCMCIA card. The `*_attach()` function should allocate and initialize a `dev_link_t` structure, and call `RegisterClient` to establish a link with Card Services. It returns a pointer to the new `dev_link_t` structure, or `NULL` if the new instance could not be created.

The `*_detach()` entry point deletes a driver instance created by a previous call to `*_attach`. It also breaks the link with Card Services, using `DeregisterClient`.

The `*_attach()` entry point is called by Driver Services when a card has been successfully identified and mapped to a matching driver by a `DS_BIND_REQUEST` ioctl(). The `*_detach()` entry point is called in response to a `DS_UNBIND_REQUEST` ioctl() call.

8.3 The *_config() and *_release() functions

The `*_config()` function is called to prepare a card for IO. Most drivers read some configuration details from the card itsef, but most have at least some built-in knowledge of how the device should be set up. For example, the serial card driver reads a card's `CFTABLE_ENTRY` tuples to determine appropriate IO port base addresses and corresponding configuration indices, but the driver ignores the interrupt information in the CIS. The `*_config` function will parse relevant parts of a card's CIS, then make calls to `RequestIO`, `RequestIRQ`, and/or `RequestWindow`, then call `RequestConfiguration`.

When a card is successfully configured, the `*_config()` routine should fill in the `dev_name`, `major`, and `minor` fields in the `dev_link_t` structure. These fields will be returned to user programs by Driver Services in response to a `DS_GET_DEVICE_INFO` ioctl().

The `*_release()` function should release any resource allocated by a previous call to `*_config()`, and blank out the device's `dev_name` field.

The `*_config()` and `*_release` functions are normally called in response to card status change events or from timer interrupts. Thus, they cannot sleep, and should not call other kernel functions that might block.

8.4 The PCMCIA event handler

The `*_event()` entry point is called from Card Services to notify a driver of card status change events.

8.5 Locking and synchronization issues

A configured socket should only be released when all associated devices are closed. Releasing a socket allows its system resources to be allocated for use by another device. If the released resources are reallocated while IO to the original device is still in progress, the original driver may interfere with use of the new device.

A driver instance should only be freed after its corresponding socket configuration has been released. Card Services requires that a client explicitly release any allocated resources before a call to `DeregisterClient` will succeed.

All loadable modules have a "use count" that is used by the system to determine when it is safe to unload a module. The convention in the PCMCIA client drivers is to increment the use count when a device is opened, and to decrement the count when a device is closed. So, a driver can be unloaded whenever all associated devices are closed. in particular, a driver can be unloaded even if it is still bound to a socket, and the module cleanup code needs to be able to appropriately free any such resources that are still allocated. This should always be safe, because if the driver has a use count of zero, all devices are closed, which means all active sockets can be released, and all device instances can be detached.

If a driver's `*_release()` function is called while a device is still open, it should set the `DEV_STALE_CONFIG` flag in the device state, to signal that the device should be released when the driver's `close()` function is called. If `*_detach()` is called for a configured device, the `DEV_STALE_LINK` flag should be set to signal that the instance should be detached when the `*_release()` function is called.

8.6 Using existing Linux drivers to access PCMCIA devices

Many of the current PCMCIA client drivers use existing Linux driver code to perform device IO operations. The Card Services client module handles card configuration and responds to card status change events, but delegates device IO to a compatible driver for a conventional ISA bus card. In some cases, a conventional driver can be used without modification. However, to fully support PCMCIA features like hot swapping and power management, there needs to be some communication between the PCMCIA client code and the device IO code.

Most Linux drivers expect to probe for devices at boot time, and are not designed to handle adding and removing devices. One side-effect of the move towards driver modularization is that it is usually easier to adapt a modularized driver to handle removable devices.

It is important that a device driver be able to recover from having a device disappear at an inappropriate time. At best, the driver should check for device presence before attempting any IO operation or before handling an IO interrupt. Loops that check device status should have timeouts so they will eventually exit if a device never responds.

9 Where to Go for More Information

The *Linux Kernel Hackers' Guide*, written by Michael Johnson, is a good source of general information about writing Linux device drivers. It is available from the usual Linux FTP sites.

The genuine PCMCIA standard is only available for a steep price from the PCMCIA association itself. The new PC Card Standard is supposed to cost $399, but old copies of the 2.1 standard may still be available at a deep discount.

```
Personal Computer Memory Card International Association
1030 East Duane Avenue, Suite G
Sunnyvale, CA 94086 USA
(408) 720-0107, (408) 720-9416 FAX, (408) 720-9388 BBS
```

One alternative is the *PCMCIA Developer's Guide*, written by Michael Mori, available from Sycard Technology, ISBN 0-9640342-1-2, for approximately $89.95:

```
Sycard Technology
1180-F Miraloma Way
Sunnyvale, CA 94086 USA
(408) 749-0130, (408) 749-1323 FAX
```

Larry Levine has written a more general introduction to PCMCIA called the *PCMCIA Primer*, which should be on bookstore shelves in the summer of 1995. It is published by M & T Books, ISBN 1-55828-437-0.

Another book tentatively called "The Insider's Guide to PCMCIA Computing" is in the works, ISBN 0-57398-010-2.

Programming information for various PCMCIA controllers is available from the corresponding chip vendors:

```
Intel Corporation
(800) 628-8686

Cirrus Logic
(510) 623-8300

Vadem
(408) 943-9301

Databook Inc.
(716) 889-4204
```