Linux PCMCIA Programmers Guide

David Hinds, dhinds@allegro.stanford.edu v1.10, 1995/02/08 17:51:54

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, 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

2 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.

2.1 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`.

2.1.1 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, following a call to `BindDevice`. If successful, a client handle will be returned in `client`.

`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 when it calls `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.

2.1.2 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.

2.1.3 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.

2.1.4 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.

2.2 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.

2.2.1 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.

2.2.2 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.

2.2.3 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.

2.2.4 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.

2.2.5 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.

2.2.6 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.

2.3 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.

2.3.1 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.

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.

2.3.2 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`.

2.3.3 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`.

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.

2.3.4 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`.

2.3.5 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.

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.

2.3.6 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.

2.3.7 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.

2.3.8 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.

2.4 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.

2.4.1 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`.

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`.

2.4.2 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.

2.4.3 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_vers_1_t         version_1;
        cistpl_config_t         config;
        cistpl_cftable_entry_t  cftable_entry;
} 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; they are quite complex. Currently, `ParseTuple` can parse `DEVICE`, `VERS_1`, `CONFIG`, and `CFTABLE_ENTRY` tuples.

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.

2.4.4 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.

2.5 Memory window control

PCMCIA cards have two memory spaces: attribute memory, and common memory. Attribute memory is commonly used for holding descriptive information (i.e., the CIS) and configuration registers. Common memory may contain device buffers in the case of IO cards, or the actual bulk storage of a memory card. 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 aligned to at least 4K boundaries 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.

2.5.1 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.

`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.

2.5.2 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.

2.5.3 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.

2.5.4 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.

2.6 Miscellaneous calls

2.6.1 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.

2.6.2 AccessConfigurationRegister

```
#include "cisreg.h"

int CardServices(AccessConfigurationRegister, 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.

2.6.3 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.

2.6.4 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.

3 Card Information Structure Definitions

3.1 CISTPL_DEVICE

The `cistpl_device_t` structure is given by:

```
typedef struct cistpl_device_t {
        u_char          type;
        u_char          wp;
        u_char          speed;
        u_long          size;
} cistpl_device_t;
```

3.2 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;
        char            pi[CISTPL_VERS_1_MAX_PROD_STRINGS][CISTPL_VERS_1_STRLEN];
} cistpl_vers_1_t;
```

3.3 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];
} cistpl_config_t;
```

3.4 CISTPL_CFTABLE_ENTRY

The `cistpl_cftable_entry_t` structure is given by:

```
typedef struct cistpl_cftable_entry_t {
        u_char          index;
        u_char          Default;
        u_char          interface;
        u_char          features;
        cistpl_power_t  power[4];
        cistpl_timing_t timing;
```

```
        cistpl_io_t     io;
        cistpl_irq_t    irq;
        cistpl_mem_t    mem;
} cistpl_cftable_entry_t;
```

4 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.

4.1 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.

4.2 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 can 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.

5 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.

5.1 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.

5.1.1 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 one 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.

5.1.2 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()`.

5.1.3 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.

5.2 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.

5.2.1 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`.

5.2.2 `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;
        bind_info_t     bind_info;
        cisinfo_t       cisinfo;
} 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(GetNextTuple, ..., &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, ...)`.

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.

6 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 you may still be able to get old copies of the 2.1 standard 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 for $89.95:

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

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
```