RadiSys.

Using CardSoft

Version 1.0

www.radisys.com
World Headquarters
5445 NE Dawson Creek Drive • Hillsboro, OR
97124 USA
Phone: 503-615-1100 • Fax: 503-615-1121
Toll-Free: 800-950-0044

International Headquarters
Gebouw Flevopoort • Televisieweg 1A
NL-1322 AC • Almere, The Netherlands
Phone: 31 36 5365595 • Fax: 31 36 5365620

RadiSys Microware Communications Software Division, Inc.
1500 N.W. 118th Street
Des Moines, Iowa 50325
515-223-8000

Revision B
July 2000

Table of Contents

Chapter 1: PCMCIA Concepts

This chapter presents a design for performing I/O to PCMCIA connected cards (PC cards) within the OS-9 operating system environment. This design is consistent with the OS-9 I/O model and supports hot-swapping, auto-recognition, and auto-configuration of PC cards. It also reflects a modified version of the interface detailed in the ***SystemSoft API Specification*** Ver. 1.00 dated 12 March 1996. This modified version is fully contained in this manual in Chapter 3: CSF API Reference.

PCMCIA Hardware and Software

The primary hardware component that needs to be supported in any PCMCIA design is the Host Bus Adapter (HBA), also sometimes referred to as a PC Card Socket Controller (PCSC). The HBA:

- provides a physical interface to one or more PCMCIA sockets
- contains logic to generate interrupts for a variety of conditions related to PC card insertion and removal
- performs programmable mapping of the three address spaces (common, attribute, and I/O) of any inserted card to the address space of the host processor
- enables control of supply voltages to an inserted card

Because a single socket can have PC cards of various types interchangeably inserted, two levels of abstraction are needed to control the interface to an inserted PC card:

- the socket level, which is independent of any PC card that may be inserted
- the card level, which has as many varieties as the types of PC cards being supported

Any I/O design for PCMCIA must account for these two distinct levels of interaction.

Support methodologies for PCMCIA software have developed in the Intel DOS, and Windows platform environments and reflect the single-threaded, single-process, system-level computing model of those environments. OS-9, on the other hand, is a multi-process environment. Run-time binding of components is needed in OS-9 to provide a connection between a process and a particular I/O device. This design provides a model consistent with that of OS-9, while providing a framework for the incorporation of existing PCMCIA software technology.

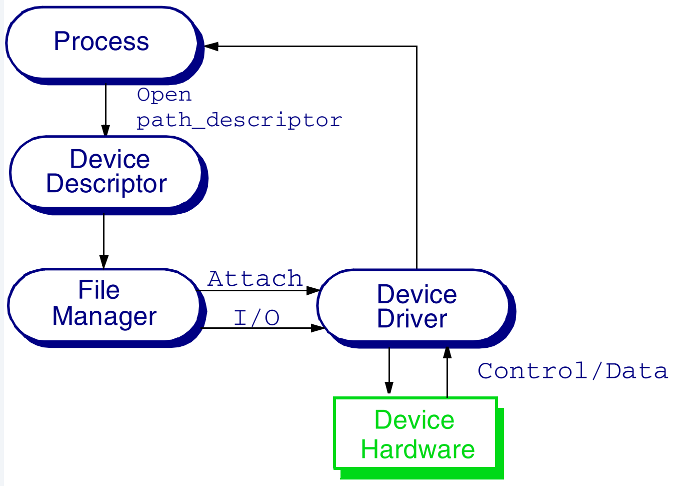
Many aspects of the PCMCIA standard and of HBA design provide backward compatibility between current PC cards and older DOS or Windows drivers and applications. For example, mapping a modem PC card registers to an I/O window where a program would normally expect to find the COM1 UART registers allows that program to operate the modem PC card as if it where a modem attached via a serial connection to port COM1. These considerations do not necessarily apply in the OS-9 environment, because there is not a legacy of implicit or default code and hardware standards. However, because OS-9 running on a PC platform must support standard I/O hardware configurations, mapping of PC card addresses to the processor I/O and memory address spaces is the same under OS-9 as it may be under DOS or Windows.

I/O in OS-9 Systems

Execution of I/O in OS-9 is performed via a file I/O system. The file I/O system provides a link between the process performing the I/O and the actual hardware device (or virtual device) that sources and/or sinks data. A file I/O system is composed of three types of entities (referred to as modules in OS-9):

- a file manager
- one or more device drivers
- one or more device descriptors

Figure 1-1 I/O Execution Sequence



A process makes a connection to a device by specifying the target device descriptor for the real or virtual device being connected. The device descriptor, in turn, specifies the device driver and file manager to be used for performing I/O to this device. The device descriptor also contains parameters relevant to the operation of the target device and is accessed by the parent driver. The system-level connection to the device is made via an `_os_attach()` call. After a device has been attached, it is available to processes for file I/O operations. A process connects to a device or file by issuing an `Open` call that creates a connection called a path. The system data that defines the path is called a path descriptor.

In the OS-9 computing model paths of code execution exist at two levels, user (process) and system (kernel). At the user level, processes execute concurrently in user state and respond to events from the user interface via shell commands. At the system level, system state control flows respond to hardware generated events and creation of a path flows from the bottom-most level. At the user level, the control flow originates from a top-most level. The creation of a path to an I/O device proceeds from top to bottom, for example, a process makes the necessary calls to establish the path. For this and for subsequent I/O calls over that path, the file manager and device driver can be viewed as re-entrant system state extensions to the calling process and can contain data private to that process, such as the path descriptor. Once the device driver is initialized and hardware interrupt sources are connected to interrupt service routines (ISRs), the bottom layers of the I/O path provide a control and data flows upward from interrupt sources at the bottom. These upward data/control flows may stop temporarily at the driver level but must eventually target one or more processes, such as cause an OS-9 signal or event to be sent to a waiting process. For this to properly occur, a path must exist between the bottom-most data/control source and the target process.

PCMCIA Sockets in an OS-9 I/O Model

For a body of code to monitor the supported PC card sockets and to report changes in their state, it must have open connections to those sockets via the file I/O system.

You should consider the sockets separately from the PC cards they may have inserted because an empty socket still has an identity as a source of events, such as changes of state as PC cards are installed and removed. After a PC card is installed, there are operations that must be performed that are generic to the socket and its associated HBA and not specific to the installed card. These operations are collectively called Socket Services. A set of socket services is defined in the OS-9 CSF API Specification. Socket Services must have a defined file system that provides I/O to the physical sockets. This file system targets individual sockets for connection via a path, rather than the entire collection of sockets (as is done in the Windows/DOS environment). This allows different processes to privately connect to specific sockets. The sockets are labeled 1 through n. Some HBAs can support multiple sockets and others just a single socket. The OS-9 I/O model accommodates both situations, and HBAs of both types can be supported in a single system.

The CSF File Manager

For access to the PCMCIA sockets consistent with the CSF API, we define a PCMCIA file manager designated csf. It provides access to all sockets as if they were a set of generic sockets and provides the Socket Services defined in the CSF API. These include

- getting socket status
- establishing memory and I/O windows
- configuring the socket
- registering the calling process to receive notification of supervisory-class events
- reading CIS (Card Information Structure) Tuples when a card is inserted

Every variety of HBA requires a device driver that is customized to the operation of that HBA variety and that supports calls from CSF. The HBA types that are supported are:

- Intel 82365 and compatible PC Card Socket Controllers
- VADEM VG-465 (Single) PC Card Socket Controller (register-compatible with the Intel 82365SL and Vadem VG-365)
- Two-Card PCMCIA 2.1 Master Interface of the Motorola MPC821 (PowerPC) RISC System Chip 1 (RSC1).
- Cirrus Logic 6700
- Texas Instruments PCI-1130

Every OS-9 File Manager must support a standard set of calls that are mapped to the actual functions supported. For CSF, the `Attach` and `Detach` perform the standard system functions. The `Open` and `Close` calls create and remove the connection to a specified socket device by creating a path descriptor. The `Setstat` call supports the CSF API functions. The remaining functions, `Getstat`, `Chgdir`, `Create`, `Delete`, `Dup`, `Makdir`, `Read`, `Readln`, `Seek`, `Write`, and `Writeln`, all operate as in any standard OS-9 system.

CSF supports up to three connections to a given socket. Any process connected to a socket can register to be notified via an OS-9 signal upon the occurrence of any number of socket events. After a notification has been received by a registered process, it can query the socket to determine which event(s) caused the notification.

The CSF Daemon Process

A single, user-state process, called the CSF Daemon (`csfd`) opens connections to all sockets in the system. It monitors sockets for card insertions and removals and performs an appropriate action based on the socket status change. This process can also spawn (fork) other processes to manage PC cards that have been previously inserted. In this way, the `csfd` monitors the state of all sockets in the system.

The CSF Daemon maintains a socket status table for all open sockets. This table contains entries that enable `csfd` to signal or fork another process when a card is inserted in a socket and to signal or kill that same process when a card is removed. The socket status table also contains an entry that denotes the type of PC card currently in the socket (including NONE).

The actions performed by `csfd` when a card is inserted in a socket are:

1. Parse the card's **Card Information Structure** (CIS) to determine the type of card inserted.
2. Configure the socket for the inserted type of card, including creating the memory or I/O windows necessary to make the card look like a standard hardware resource in the system.
3. Attach the appropriate device descriptor to bind the PC card into the system.
4. Fork a handler process specific to the card type.
   or
   Signal a global user-interface process that a card of the determined type has been inserted into the given socket.

When a card removal notification is received, `csfd` kills any process that was previously forked for that card/device and performs a special hard `Detach` of the device from the system.

The services provided by the CSF Daemon are often referred to collectively as Card Services and Client Driver Services.

Card Information Structure

The Card Information Structure (CIS) is the primary standard configuration data layer on a PC Card device. All cards conforming to the PC Card 2.0 specification must have a Card Information Structure stored in non-volatile memory, which must be mapped at card memory location zero. It is organized as a linked list of tuples which contain configuration data relevant to the inserted card.

Each tuple has a type code, a link to the tuple, and some tuple-specific data. The PCMCIA standard specifies the tuple type codes used to describe card attributes and configuration.

• `TPL_CODE`	Always the first byte, contains the tuple type code of the `TPL_DATA`.
• `TPL_LINK`	Always the second byte, contains a count of the number of bytes remaining in the current tuple.
• `TPL_DATA`	Always starts at byte 3 and is the length specified in the `TPL_LINK` byte. This is the `TPL_CODE`-specific configuration data area.

For More Information

For additional information on CIS structure and tuple codes, please see a reference such as:

PCMCIA System Architecture, 2nd edition
by Don Anderson, 1995, Addison Wesley Publishing Co.
ISBN: 0201409917

The PCMCIA Standard Software Model

The PCMCIA standard provides a model for software that interfaces and operates PC cards. Three software levels exist in this model: Socket Services, Card Services, and (generic) Client Driver Services.

Socket Services

Socket Services include:

1. Basic initialization of the HBA (including programming of supply voltages) upon detection of a card insertion.
2. Programming an attribute memory window and parsing the CIS to determine its type and the resources (address space, I/O space, interrupt lines, levels, and vectors) it will require.
3. Programming of the HBA registers to permit card access by the host (based on the required resources).
4. Responding to interrupts (or the polling of HBA registers) to monitor socket status change events.
5. Releasing system resources and clearing the HBA when a card is removed.

In this design, Socket Services are provided primarily by the CSF file system and CSF Daemon.

Card Services/Client Driver Services

Card Services and Client Driver Services include:

1. Client services that provide a registration facility for clients to be notified of socket events.
2. Resource management that enables clients to access and program memory windows and card-level interrupt sources.

3. Client utilities that provide a set of functions commonly used by all clients, such as reading of tuples from a card’s CIS.
4. Bulk memory services that provide, in conjunction with an appropriate Memory Technology Driver (MTD), functions to read, write, copy, and erase blocks of data within memory cards.

Card Services and Client Driver Services are provided primarily by the `csfd` process.

Chapter 2: CardSoft Operation

This chapter contains a product overview followed by operation instructions for CardSoft for OS-9.

RadiSys.

MICROWARE SOFTWARE

Overview

The CardSoft Daemon (`csfd`) for OS-9 is a high-level, user-state task that monitors one or more PCMCIA PC card sockets for card insertions and removals. When a card is inserted, `csfd` attempts to identify the card. If the card is supported, `csfd` configures the card to match one of a set of specified OS-9 device descriptors.

Upon start-up, `csfd` attempts to open one or more PC Card sockets identified by device descriptors cs1, cs2, cs3, cs4. If at least one socket is successfully opened, `csfd` links to a data module named `csfdb`. The `csfdb` module defines a table that binds PC Card types with device descriptors. For example, a serial PC Card might be bound to device descriptors `t2`, `t3`, and/or `t4`, or an ATA Disk (hard disk) PC Card might be bound to descriptors `mhc1` and/or `mhe1`. If `csfd` cannot find or otherwise link to the `csfdb` module, it reverts to a default built-in table of bindings.

The CSF Daemon may selectively report a number of internal, operational, and error conditions via text messages written to `stdout`. Additional features that can be selected are:

- Audio enunciation of successful/failed configuration of inserted PC Cards and of card removal (this feature requires support hardware accessed via a program module named “bell”.)
- Automatic breaks to RomBug at specific parts of `csfd` operation
- Automatic call to `_os_attach()` for the device selected for a successfully configured PC card
- Write to a specified (or default) path announcing the device selected for a successfully configured PC card
- Poll sockets periodically for activity to eliminate the need for interrupts from the socket controller
- Complete processing despite a bad (or non-existent) CIS parse by way of a default or user specified card type (and memory size)
- Initialize and resize (such as overriding the device descriptor size with the size read in the CIS) of an `rbftl` flash volume

- ignore CIS or override size information for a flash card and assume the value specified in the device descriptor is correct
- enable 12Vpp for flash cards (assuming the socket can supply 12V). This can be dangerous because 5 V-only cards will be damaged; the daemon does not distinguish 5 V-only cards from 12V cards. This capability is provided primarily for testing with older 12V flash cards.

The code implementing these features may be selectively included or excluded from the `csfd` code image by performing a customized make of `csfd`. For those features that have been built into a particular version of `csfd`, they can be enabled via command-line parameter flags when the daemon is executed. The CSF Daemon is port independent; no code changes or conditional flags are required to specify the target port or processor under which `csfd` will operate. Any necessary compiler/linker flags that need to be specified for the target processor are supplied automatically during the make.

Interaction Between CSF Product Modules

If you wish to make configuration changes to the CardSoft product, it is useful to know how the different modules in CSF interact with each other.

The hardware necessary to support your CardSoft system includes a PC Card socket controller. This device maps PC card memory space and interrupts onto equivalent host memory space and interrupts. The socket controller driver is the software that configures this device. Under OS-9, a small module called a descriptor is used to by the driver to store configuration specifics, such as the device address and interrupt vector. The socket device descriptors in the CSF package are named `cs1` through `cs4`, depending on how many unique hardware sockets the port has.

Under the OS-9 I/O model, a file manager enables user-level applications to communicate with a device driver. Some file managers implement a complex file structure on a given device, while others simply pass the application calls on directly to the device driver. The latter is the main function of the `csf` file manager module.

This is typically where system ends and a user application begins in a traditional OS-9 I/O system. For the CardSoft package, this is only half of the system. There is a daemon process, `csfd`, which registers with the `csf` file manager for card insertion and removal interrupts. When a card insertion occurs, the daemon reads, through the file manager, the Card Information Structure (CIS) of the inserted device. The CIS indicates the card type and acceptable configurations for the card hardware. The CSF daemon uses this information to locate an appropriate device descriptor for the hardware present on the card. The master mapping table of card type to device descriptor name is stored in a data module, `csfdb`. The daemon reads the `csfdb` data module to get the appropriate device descriptor.

Once a card device descriptor has been chosen by the daemon, it programs (by calling the socket driver through the `csf` file manager) the appropriate device interrupt vector and I/O window specified in the card device descriptor. At this point, a card device is ready for use in the system and appears to a device driver as standard hardware on the bus.

When a card is removed from a socket, the daemon is notified by the `csf` file manager. It then marks the device descriptor as available and waits for future insertion notification. Depending on command line flags, the daemon may also take other actions, such as forcibly de-initializing a device.

Configuring the CSF Data Module

The CSF data module `csfdb` is used by the daemon to control mappings of card types to system descriptors. The information in a descriptor is then used by the daemon to configure the PC Card socket controller so the device appears as a normal device on the system bus.

Changing these mappings involves modifying the `csfdb.des` file located in `MWOS/OS9000/SRC/IO/CSF/DATA/<PORT>`. The following steps explain how to add a device entry to the CSF data module.

Step 1. Add a `dev_tab_initializer` entry to the `csfdb.des` file. For example, if you have a configuration with a second ATA disk, insert:

```
init dev_tab_initializer[7]
{
   type = ATA_DISK;
   devname = devname_string7;
};
string devname_string7 = "mhe1";
```

Step 2. Now you must change the count in `mod_body` to reflect any changes you made to the `dev_tab_initializer`. Following the example in Step 1, `mod_body` is changed as follows:

```
init mod_body
{
   sanity = 0xface;
   count = 8;
};
```

Step 3. Save the updated `csfdb.des` file.

Step 4. Go to `MWOS/OS9000/<CPU>/PORTS/<PORT>/CSF/DATA` and perform a make. This creates an updated `csfdb` module in your `MWOS/OS9000/<CPU>/PORTS/<PORT>/CMDS/BOOTOBJS/DATA` directory.

Note

When running under DOS/Windows or UNIX, the make command is `os9make`. When running resident under OS-9, the make command is `make`.

Building the CSF Daemon

To build the CSF Daemon, you first customize the supplied makefile `COPTS` definition, then execute `os9make` on the customized makefile. This makefile builds all processor-specific versions. The makefile is located in the daemon source directory:

```
MWOS/OS9000/SRC/IO/CSF/DAEMON
```

Step 1. Modify the makefile `COPTS` definition to include (via a `-d` macro definition) the features needed in your customized CSF daemon. Symbol definitions for `COPTS` are given in the following section.

Step 2. Execute `os9make` on the customized makefile.

The DEFS declaration in the daemon makefile may look similar to the following if you enable all options:

```
DEFS = $(MWOS_DEFS) -v=$(MWOS_DDIR) -v=$(MWOS_DDIR)/HW      \
      -v=$(MWOS)/OS9000/SRC/IO/RBF/DRVR/RBFTL               \
      -v=$(MWOS_DDIR)/IO                                    \
      -dPCIS -dPCONFIG -dPEVENT -dPDEBUG -dPERRORdPBREAK    \
      -dPBELL -dANNOUNCE -dPBREAK -dPATTACH -dPRBFTL        \
      -dPVPP12 -dPOVERIDE -dPPOLL -dPMEMSIZE                \
      -d_SER_8250 -d_ATA_DISK -d_FLASH -d_NETWORK
```

Symbol Definitions for COPTS

Symbol definitions included in the makefile `COPTS` definition enable you to customize the `csfd` as follows:

`SER_8250`	adds code for processing serial PC cards such as modems.
`ATA_DISK`	adds code for processing ATA/IDE Disk PC cards.
`NETWORK`	adds code for support of 3COM 3C509 PC Cards.

`FLASH`	adds code for processing linear (non-ATA) flash cards. This capability requires a SystemSoft license for the `rbftl` TrueFFS flash file system to both make and operate.
`ANNOUNCE`	adds code to process the `-a` command line options flag to provide an audible announcement of card insertions and removals.
`PBELL`	adds code to process the `-b` command line options flag to provide an audible bell signalling card insertions and removals.
`PDEBUG`	adds code to process the `-d` command line options flag for extended card processing information.
`PERROR`	adds code to process the `-e` command line options flag to provide extended card processing error information.
`PRBFTL`	adds code to process the `-f` command line options flag to initialize and enable post-configuration of an `rbftl` driver.
`PCONFIG`	adds code to process the `-g` command line options flag to display all possible card configurations.
`PCIS`	adds code to process the `-i` command line options flag to display the card information structure at insertion.
`PBREAK`	adds code to process the `-k` command line options flag that will jump into the ROM debugger on insertion.
`PPOLL`	adds code to process the `-l` command line options flag that turns off the use of interrupts to detect card insertions and removals.
`POVERIDE`	adds code to process the `-o` and `-s` command line flags that are used to override card information structure initialization.
`PMEMSIZE`	adds code to process the `-m` command line options flag that does not override the descriptor specified volume size.
`PVPP12`	adds code to process the `-p` command line options flag that sets socket voltage to 12 V.

`PEVENT`	adds code to process the `-v` command line options flag that provides a verbal announcement of card insertion and removal to `stdout`.
`PATTACH`	adds code to process the `-z` command line options flag that automatically attaches a device upon insertion.

Setting Up the Target System

When configuring a system boot to have PC Card support for ATA disk and serial, it is necessary to have the following modules in memory (either configured into a boot or loaded into a system at runtime):

file manager:	`csf`
socket descriptors:	`cs1`, `cs2`, `cs3`, and `cs4`

controller:	82365	MPC821/860	PCI-1100	CL6700
socket driver:	`cs82365`	`cs821`	`cs1100`	`cs6700`

daemon:	`csfd`
data module:	`csfdb`

	ATA DISK	SERIAL
device descriptors:	`mhc1 or mhe1`	`t2`
file managers:	`pcf`	`scf`
device drivers:	`rb1003`	`sc16550`

additional useful modules to have:

`dir` `kermit` `irqs` `devs` `list` `procs`

As an example, if you are configuring a bootlist for a PC system, the following list of module paths may be appended to the image bootlist:

```
OS9000/80386/CMDS/bell
OS9000/80386/CMDS/csfd
OS9000/80386/CMDS/BOOTOBJS/csf
OS9000/80386/PORTS/PCAT/CMDS/BOOTOBJS/cs82365
OS9000/80386/PORTS/PCAT/CMDS/BOOTOBJS/DATA/csfdb
OS9000/80386/PORTS/PCAT/CMDS/BOOTOBJS/DESC/CS82365/cs1
OS9000/80386/PORTS/PCAT/CMDS/BOOTOBJS/DESC/CS82365/cs2
OS9000/80386/PORTS/PCAT/CMDS/BOOTOBJS/DESC/RB1003/mhe1
   OS9000/80386/PORTS/PCAT/CMDS/BOOTOBJS/DESC/SC16550/t2
```

Finally, you should uncomment the lines for desired modules `rb1003`, `sc16550`, `pcf`, `scf`, `dir`, `kermit`, `irqs`, `devs`, `list`, and `procs`.

Testing Your CSF Configuration

The following sections contain procedures for testing your CSF configuration when inserting, removing, and operating a PC Card device.

Testing Card Recognition

Once you have the CSF modules in memory and are ready for a first test, you must start the CSF daemon before you are able to use a PC Card device.

Step 1. Insert a PC Card (either ATA or Serial).

Step 2. Start the daemon in the foreground with all the debugging options active. This can be done by typing:

```
csfd -igvde
```

The daemon displays a good deal of debugging information in this mode, including Card Information Structure (CIS), tuple processing, and device descriptor selection. You may need to press return several times to continue message display. To see a continuous display, type

```
tmode nopause
```

before typing the `csfd` command.

The messages displayed are for an ATA interface disk similar to:

```
Card type is : ATA DISK

Testing card for use with device descriptor mhc1.
Configuration registers base address = 00000200.
Configuration registers mask value   = 0000000f.
Configuring an ATA Hard Disk card.
```

which indicates that the CIS information has been parsed correctly and that the card was found to be an ATA interface disk.

Step 3. Near the end of the processing, you should see the message:

```
Found a usable configuration
```

which indicates card identification was successful and a matching device descriptor is available in the system.

If you do not see these messages, please recheck your configuration and verify that all necessary modules are in memory, a PC Card is inserted into the slot, and the daemon was started correctly.

For More Information

Please see the section Configuring the CSF Data Module.

Testing Card Insertion or Removal

Step 1. Remove the PC Card in order to verify that card insertion/removal interrupts occur correctly. You may see several messages similar to:

```
Socket signal received from /cs1.
Timer signal received for socket /cs1.
```

and then finally:

```
Debounce delay for /cs1 complete - ready to
   recognize insertions.
```

This is normal removal behavior.

Step 2. Insert the card again.Messages displayed are similar to:

```
Socket signal received from /cs1.
Timer signal received for socket /cs1.
```

and then:

```
Debounce delay for /cs1 complete - process card.
Card is in socket /cs1 - beginning processing.
```

Following the card insertion recognition, you should see the identical sequence of CIS tuple interpretation that you saw in Step 2. of Testing Card Recognition.

If the daemon does not receive card removal or insertion interrupts, check (see configuration section) to ensure that the interrupt is not being shared with another device in your system. The `irqs` utility may help in this.

Step 3. Exit the daemon with `^E`.

Testing PC Card Operation

Once you have verified correct operation of the CSF daemon, you should try using a PC Card in the system.

Step 1. Start the daemon in the background with no debugging options by typing:

```
csfd &
```

Step 2. If you have an MS-DOS formatted ATA interface disk, a good test is:

```
dir -e /mhc1
```

which provides a directory of the device. If successful, you can try more complicated things like copying files to and from the disk device.

If you have a PC system, use the name `/mhe1` instead of `/mhc1`.

Step 3. If you have a serial device, such as a modem card, you can try using the kermit utility to connect.

```
kermit cl /t2
```

Typing "AT" to most modems will cause them to respond with "OK". On some modems, you can type "AT$" or "AT&V" to get additional output without needing to dial and connect to another computer.

csfd Operation

Operating the CSF Daemon

The CSF Daemon is a command line program executed from the shell. Operation is controlled by setting one or more flags when executing. The following pages contains the complete reference for `csfd` operation and options.

Performing I/O on an Inserted PC card

No special steps have to be taken to perform I/O on an inserted PC card.

Note

A point of concern is that a process may have a path open to a device and is performing I/O when the associated PC card is physically removed. Because `csfd` does a "hard shutdown" of the device, any I/O operation in progress will fail. Monitoring failure returns for I/O operations is a method that enables the process to terminate operations with that device in the correct manner and sequence.

Purpose

`csfd` acts on PC Card insertions and removals by changing the state of a running system.

Syntax

```
csfd [<opts>] [<redirect>] [&]
```

Options

The CSF Daemon may be executed with one or more options set. Options specify the features that are enabled during operation. If an option is specified that was not included in the build of a customized daemon, an error is returned. The options are interpreted as follows:

`-a[<path>]`
: enables announcement via `<path>` (default is `/nil`) of the device selected for a successfully configured PC Card. A string of the form:

```
Device <type> is now available.
```

is written or appended to the specified device or file.

`-b` enables audio enunciation when a card is inserted or removed. A bell module must be in the memory directory to use this flag. The sounds generated are industry standard tones as follows:

(a) A two-tone sequence of 660 Hz followed by 880 Hz signals successful configuration and attachment to a device descriptor of an inserted card.

(b) A single tone of 220 Hz signals unsuccessful card insertion.

(c) A two-tone sequence of 880 Hz followed by 660 Hz signals card ejection.

`-d` enables a printout of miscellaneous initialization and processing details.

`-e` enables a printout of all internal error conditions.

-f enables an auto-initialize and post-configuration of an rbftl supported flash card. This sets the card size from the CIS rather than from the device descriptor, where the latter value is an upper bound. Be sure to deinitialize the device (rcf1, pcf1) before ejecting the card. Use of this option requires a SystemSoft rbftl/TrueFFS flash file system license.

-g enables a printout of the fully parsed configurations within the CIS of an inserted card. The printout contains configurations up to and including the configuration that matches that of a candidate device descriptor.

-i enables a printout of the CIS of an inserted card at the time of parsing.

-k enables various breaks to RomBug after a card is inserted. A break occurs just before the CIS scan to allow manual examination of attribute memory and just after successful configuration to allow manual examination of the device hardware registers.

-l[<time>]
forces socket polling to disable use of interrupts by the driver. If the poll cycle time in msec is not specified, the default of 2000 msec is used.

-m instructs csfd to ignore the CIS and not to override the volume size information and assumes the value specified in the device descriptor.

-o[<opt>]
specifies CIS override. This causes csfd to ignore a bad or missing CIS and to use a user-specified value for the card type. A valid CIS parse always takes precedence over the override value. Override values may be one of the following:

1 is Serial 8250
5 is ATA Disk
7 is Flash (default)
9 is Network

-p causes socket V_{pp} voltage to be set at 12 V instead of the default 5 V for flash cards.

WARNING
Use the `-p` flag very carefully. 12 V V_{pp} will damage a 5 V card because `csfd` cannot automatically determine the voltage needed by an inserted flash card.

`-s[<size>]`
specifies a memory size for override (see `-o` flag) when the specified (or default) card type requires a memory size such as for flash or RAM cards. The default is 4 MB.

`-v` enables a printout of recognized card insertion and ejection events.

`-z` enables automatic attach of the device selected for a successfully configured card. This is useful when the `-a` option is not used, as it permits the user to determine what device has been selected by the execution of a `devs` command. This flag requires that you manually deinitialize a configured device before physically ejecting the PC Card. Using this flag is risky when testing new drivers for new cards because the `_os_attach()` call may crash the system.

`-?` prints out a help message. When the `?` flag is specified, all other flags are ignored, and `csfd` terminates after writing the help message to `stdout`. Also, the help message shows only those flags that have been built into the particular `csfd` image being executed.

Description

If redirection to a logfile path is not specified, all output will go to the current terminal output device. If concurrent execution isn't specified with the `&` operator, the current shell on the terminal will not be available while `csfd` is running. This can be useful (along with defaulting output to the console) when debugging/testing PC Cards and `csfd`. Output to the terminal with concurrent execution is not a viable option because of device competition with the standard shell.

Note

The current convention for redirection is to file `/dd/SYS/csfd.log`

The `csfd` expects the `csfdb` data module, the optional bell module, and any functional device descriptor modules that might be opened on behalf of inserted PC Cards to be resident in the current module directory. No attempt is made to load needed modules from the current data or execution directories. This facilitates operation in diskless systems.

If one or more PC cards are already inserted when `csfd` is started, they will be processed at the time of the start-up.

PC Cards and Drivers

The PC Cards supported by `csfd` are:

- Cards with an 8250/16450/16550 UART interface via the `sc16550` driver.
- ATA Disk cards (rotating or solid-state such as Hard disk or Flash disk) via the `rb1003` driver.
- Linear Flash cards operating under TrueFFS via the `rbftl` driver.
- 3COM 3C589 Ethernet LAN card via the `ife509` driver.

In the case of cards with an 8250/16450/16550 UART interface, such as modem cards, the descriptors `t2` through `t4` are used. This requires that the UART ports normally associated with these descriptors are disabled. On an architecture such as the PC, which requires an I/O region SCF driver, if no UART ports are available, then one or more ports must be disabled via CMOS setup or the PC Card support must be done via memory-mapping of the UART registers. The latter option would require a custom SCF driver.

To support a serial PC card via device descriptor `t2`, you must ensure that no COM2 port (0x2f8) is present in the system. This is an issue because some serial PC cards, such as the New Media Net Surfer, do not have selectable configurations COM3 (`t3`) or COM4 (`t4`) I/O port addresses; they can be accessed via COM1(`t1`) or COM2 (`t2`) I/O port addresses. You may be able to disable COM2 via the CMOS setup.

Table 2-1 shows which PC Cards have been successfully tested.

Table 2-1 Supported PC Cards

PC Card
Epson EHDD170 170 MB Hard Disk
SunDisk 2.5 MB Flashdisk Mass Storage System
I-O Data 5 MB PCMCIA Flash ATA Card

Table 2-1 Supported PC Cards (continued)

PC Card
Simple Technology STI-ATAFL/12, 12 MB Flash Memory
Viking Components 8, 16, and 20 MB ATA Flash Cards
SanDisk SDCFB-15-144 15 MB “CompactFlash” with PCMCIA-ATA Adaptor
Practical Peripherals PC144T2-EZ ProClass 14.4 Data/Fax Modem
New Media V.34 Net Surfer 28.8 Internal DAA Modem
Megahertz XJ1144 14.4 Fax Modem
Angia SJ288C SafeJack 28.8 Fax/Modem Card
IBM Wireless Modem for Cellular/CDPD
3COM 3C589 Ethernet LAN Card
AMD C and D Series 2, 4, and 8 Mbyte Flash Memory Cards
AMP FLASH-5C 4 Mbyte & FLASH-12HC 16 Mbyte Flash Memory Cards
Intel/ExCA Series 2 4 Mbyte Flash Memory Card
AMD 2, 4, 8 Mbyte 5V Flash Mini-Cards in SCM MCR-5 Adaptor
AMD 2, 4, 8 Mbyte 3V Flash Mini-Cards in SCM MCR-3 Adaptor

Socket Controllers and Platforms

Table 2-2 shows which CSF drivers to use on various processors and platforms:

Table 2-2 Processor/Platform/Driver Compatibility

Processor and Platform	CSF Driver
80X86 processor on the PC platform	`cs82365`
MPC821 processor on the 821ADS or 860ADS platform	`cs821`
MPC821 processor on the MBX821/860 platform	`cs821`
PowerPC processor with a TI PCI-1130 socket controller	`cs1130`
Cirrus Logic CL-7111ARM processor on the CLPS7111 platform	`cs6700`

Chapter 3: CSF API Reference

The CSF API defines twelve function calls to manage the I/O traffic and status of the PC Card. This section provides a complete reference to those functions.

Overview

The OS-9 CSF API specification is a modification of the SystemSoft API specification for use with the SystemSoft OS-9-hosted CSF File Manager. Hardware independence is maintained by virtue of the fact that OS-9 is portable over a large set of processors and platforms. Where there are processor and/or platform dependencies, they are noted.

This API is based on the ***PCMCIA Card Services Specification***, published by the Personal Computer Memory Card International Association (PCMCIA), but is modified and simplified to be appropriate for OS-9. This API hides the details of the slot hardware from the caller and presents a uniform interface to a variety of generic slots. This API also provides a set of operations that always operate on a single, unspecified slot. The slot is identified by the device path for which a CSF I/O call is being made.

CSF System Call Implementation

The functions in this API are implemented by an OS setstat I/O call to the opened path of the slot device. Although there is a standard C binding for OS-9 I/O calls, an I/O-independent C binding can be layered on top of the setstat call and might look like the C function specified in the following paragraph.

There are ten API functions defined. Each is implemented as an `I_SETSTAT` call, structured as a generic call:

```
((specific_pckt_type*)
  parm_pckt_pntr) -> code = function_code;
  error_code = _os_setstat(
     (path_id) path_handle,
     (u_int32) SS_CSF,
     (void*) parm_pckt_pntr );
```

The function code present in all parameter blocks specifies which of the ten API functions is being called. The `handle` present in all parameter blocks defines which (opened) slot is being accessed. The setstat return can either be specific to the executed function or it can be a standard OS-9 file I/O system error. Only the error codes specific to the function, if any, are shown for each function.

Definitions are located in:

```
  MWOS/OS9000/SRC/DEFS/csf.h
```

Function code

```
#include <csf.h>
I_SETSTAT, CSF_GET_STATUS
```

Parameter Block Structure

```
typedef struct get_status_parm_pkt
{
   csf_api_code code;
   u_int32      handle;
   u_int16      mask;
   u_int16      control;
   u_int8       flags;
   u_int8       state;
   u_int8       Vcc;
   u_int8       Vpp;
}
get_status_parm_pkt, *Get_status_parm_pkt;
```

Description

CSF_GET_STATUS returns information about the state of the socket. The parameter block for the CSF_GET_STATUS function is shown in Table 3-1 CSF_GET_STATUS Parameter Block.

Table 3-1 CSF_GET_STATUS Parameter Block

Structure	I/O	Description
u_int32	I	Client Handle from CSF_REG_CLIENT
u_int16	I/O	Event Notify Mask and Status (see Table 3-2 GetStat Event Notify Mask)

Table 3-1 CSF_GET_STATUS Parameter Block (continued)

Structure	I/O	Description
`u_int16`	`I`	Optional Socket Controls (see Table 3-3 CSF_GET_STATUS Socket Controls)
`u_int8`	`O`	Flags (see Table 3-4 CSF_GET_STATUS Flag Field)
`u_int8`	`O`	State (see Table 3-5 CSF_GET_STATUS State Field)
`u_int8`	`O`	Vcc value (in tenths of a volt)
`u_int8`	`O`	Vpp value (in tenths of a volt)

The "Event Notify Mask and Status" field represents which event has generated a notification (see `CSF_REG_CLIENT`). If the caller is a registered client for notifications and supplies this client handle, then the Event Notify Mask and Status fields contain valid information. If this is the case, only those bits set in the Mask at the time of this call are reported, and, for those with a return value of 1, they are reset internally at the completion of the call. Any bits set under the mask represent new notification(s) since the last execution of `CSF_GET_STATUS`. If a client handle value of zero is supplied, this part of the operation is bypassed. The meaning of the bits on return from the call are detailed in Table 3-2 GetStat Event Notify Mask.

.

Table 3-2 GetStat Event Notify Mask

Location	Description
Bit 0	Notification has been issued for Write Protect status change.
Bit 1	Notification has been issued for Card Lock change.
Bit 2	Notification has been issued for Ejection Request.
Bit 3	Notification has been issued for Insertion Request.
Bit 4	Notification has been issued for Battery-dead change.
Bit 5	Notification has been issued for Battery-low change.
Bit 6	Notification has been issued for Ready change.
Bit 7	Notification has been issued for Card Detect change.
Bit 8	Notification has been issued for Power Management change.
Bit 9	Notification has been issued for Card Reset.

Which of these notifications is available for a given socket is dependent on the controller (HBA) supporting that socket.

The optional socket controls allows low-level control of the socket to be performed as shown in Table 3-3 CSF_GET_STATUS Socket Controls.

Table 3-3 CSF_GET_STATUS Socket Controls

Location	Description
Bit 0	1 = Assert PC Card Reset
Bit 1	1 = Release PC Card Reset
Bit 2	1 = Enable Power to PC Card
Bit 3	1 = Disable Power to PC Card
Bit 4	1 = Enable Auto-Power-On
Bit 5	1 = Disable Auto-Power-On
Bit 6	1 = Enable PC Card Output Drivers
Bit 7	1 = Disable PC Card Output Drivers

Table 3-4 CSF_GET_STATUS Flag Field

Location	Description
Bit 0	0 = Memory-only mode 1 = Memory-and-I/O mode
Bit 1	0 = IREQ disabled 1 = IREQ enabled (valid only if Bit 0 is set)

Table 3-5 CSF_GET_STATUS State Field

Location	Description
Bit 0	0 = WP is not being asserted. 1 = WP is being asserted.
Bit 1	0 = Card is not locked. 1 = Card is locked.
Bit 2	Not used.
Bit 3	0 = Power to PC Card is Off. 1 = Power to PC Card is On.
Bit 4	0 = BVD1 is not being asserted. 1 = BVD1 is being asserted.
Bit 5	0 = BVD2 is not being asserted. 1 = BVD2 is being asserted.
Bit 6	0 = READY is not being asserted. 1 = READY is being asserted.
Bit 7	0 = Card is not present in socket. 1 = Card is present in socket[a].

a. If Bit 7 is 0, the other bits (except Bit 3) have no significance. If the socket is in I/O mode, Bits 0 and 4-6 are only valid if the card in the socket has a Pin Replacement Register and the presence of the PRR was noted in the most recent `CSF_REQ_CONFIG` call for that card.

Example

```
#include <csf.h>

get_status_parm_pkt pb;

pb.code    = CSF_GET_STATUS;
pb.handle = <SAVED HANDLE>;
pb.mask    = NO_BIT;
pb.control = NO_BIT;

error_code = _os_setstat((path_id) path, (u_int32)
SS_CSF,
                         (void *) &pb);
```

Errors

`EOS_CSF_BADHANDLE`	Client Handle is invalid or not assigned.

Function Code

```
#include <csf.h>
I_SETSTAT, CSF_REQ_WINDOW
```

Parameter Block

```
typedef struct window_parm_pkt
{
   csf_api_code code;
   u_int32      handle;
   u_int32      baseaddr;
   u_int32      size;
   int32        offset;
   u_int16      flags;
   u_int8       speed;
}
window_parm_pkt, *Window_parm_pkt;
```

Description

Before a device may be used in the system, you must map the device register set and/or memory into logical addresses on the system bus. This setstat is used to request that mapping.

`CSF_REQ_WINDOW` requests that a window on the socket be assigned to the caller and that the window be set up as specified by the other fields of the parameter block. The parameter block for the `CSF_REQ_WINDOW` function is as follows:

Table 3-6 CSF_REQ_WINDOW Parameter Block

Parameter	I/O	Description
`u_int32`	`I`	Window handle
`u_int32`	`I/O`	Window base address
`u_int32`	`I`	Window size (in bytes)
`u_int32`	`O`	Card offset of window base
`u_int16`	`O`	Flags (see Table 3-7 CSF_REQ_WINDOW Flags Field)
`u_int8`	`O`	Device access speed

The window base address, size, and card offset fields may be adjusted to correspond to the requirements of the PC Card adapter hardware, but the assigned window always encompasses at least the requested window extent and always allows access to the region of the card specified by the caller.

The “Flags” field is bit-mapped. The meaning of the bits is shown in **Table 3-7** on page 52:

Table 3-7 CSF_REQ_WINDOW Flags Field

Location	Description
Bit 0	0 = This is a memory window 1 = This is an I/O window
Bit 1	0 = This is a common memory window 1 = This is an attribute memory window (valid only if Bit 0 is 1)
Bit 2	0 = The window is write enabled 1 = The window is write protected
Bit 3	0 = The data path width is 8 bits 1 = The data path width is 16 bits
Bit 4	0 = The window is disabled 1 = The window is enabled
Bit 5	0 = The Zero Wait State option is deselected 1 = The Zero Wait State option is selected

The `Speed` field is in the format of an extended-device-speed code, as defined in the PCMCIA PC Card Standard, Section 5.2.7.1.3. Bit 7 of the `speed` byte must always be 0. Since the basic timing for an I/O cycle is fixed by the PCMCIA PC Card Standard, the device speed field may be ignored for an I/O window, depending on the implementation. If it is not ignored, a device speed value of 0x32 (nominally 250ns) is interpreted as requesting the standard I/O timing, while other device speed values generate I/O timings similar to the memory timings corresponding to those values. For OS-9 the currently defined speed values are 0 to 3, which correspond to wait-state values of 0 to 3.

If the call to `CSF_REQ_WINDOW` is successful, a “window handle” is returned in the parameter block. This handle is an opaque type but is guaranteed not to be 0. On error the “window handle” is not guaranteed to be 0.

Example

```
#include <csf.h>

window_parm_pkt pb;

pb.code     = CSF_REQ_WINDOW;
pb.handle   = <SAVED HANDLE>;
pb.baseaddr = <BASE ADDRESS IN SYSTEM MEMORY>;
pb.size     = <SIZE OF WINDOW>;
pb.offset   = 0;          /* Offset into card address
space */
pb.flags    = MEM_WINDOW | WIN_WE | DATA_WIDTH_8 |
WIN_ENABLED;
pb.speed    = 0;

eerror_code = _os_setstat((path_id) path, (u_int32)
SS_CSF,
                          (void *) &pb);
```

Errors

`EOS_CSF_ALLOCERR`	Window base address is invalid (0 is disallowed in OS-9) or no free windows are available to allocate.
`EOS_CSF_BADOFFSET`	Card offset is invalid.

Function Code

```
#include <csf.h>
I_SETSTAT, CSF_REL_WINDOW
```

Parameter Block

```
typedef struct window_parm_pkt
{
   csf_api_code code;
   u_int32      handle;
   u_int32      baseaddr;
   u_int32      size;
   int32        offset;
   u_int16      flags;
   u_int8       speed;
}
window_parm_pkt, *Window_parm_pkt;
```

Description

CSF_REL_WINDOW unmaps a window and makes it available again. This function code uses the same parameter block as I_SETSTAT, CSF_REQ_WINDOW, but all fields except the Window Handle are ignored. A handle value of -1 will release ALL open windows. After a successful call to CSF_REL_WINDOW, the handle becomes invalid.

Example

```
#include <csf.h>

window_parm_pkt pb;

pb.code      = CSF_REL_WINDOW;
pb.handle    = <SAVED HANDLE>;
```

```
error_code = _os_setstat((path_id) path, (u_int32)
SS_CSF,
                                (void *) &pb);
```

Errors

`EOS_CSF_BADHANDLE`	Window handle is invalid or not assigned.

I_SETSTAT, CSF_REQ_CONFIG

Set Socket to I/O Mode

Function Code

```
#include <csf.h>
I_SETSTAT, CSF_REQ_CONFIG
```

Parameter Block

```
typedef struct config_parm_pkt
{
   csf_api_code code;
   u_int16      flags;
   u_int8       Vcc;
   u_int8       Vpp;
   u_int32      base;
   u_int8       option;
   u_int8       status;
   u_int8       pin_rep;
   u_int8       socket_copy;
   u_int16      vector;
   u_int16      level;
   void*        entry;
}
config_parm_pkt, *Config_parm_pkt;
```

Description

`CSF_REQ_CONFIG`

- Sets the socket into I/O mode.
- Enables the card IREQ signal to generate an interrupt.
- Sets the voltages on the socket.
- Writes to the card configuration registers to set up the card for I/O operations.

Once `CSF_REQ_CONFIG` has been called on the socket, it cannot be called again until either the configuration is released or the configured card has been removed. Removing the card automatically releases the configuration.

Table 3-8 CSF_REQ_CONFIG Parameter Block

Parameter	I/O	Description
`u_int16`	I	Flags (see Table 3-9 CSF_REQ_CONFIG Flags Field)
`u_int8`	I	Vcc value (in tenths of a volt)
`u_int8`	I	Vpp value (in tenths of a volt)
`u_int32`	I	Configuration register base
`u_int8`	I	Configuration Option Register value
`u_int8`	I	Configuration & Status Register value
`u_int8`	I	Pin Replacement Register value
`u_int8`	I	Socket & Copy Register value
`u_int16`	I	IREQ IRQ vector number
`u_int16`	I	IREQ IRQ level (such as poll priority)
`void*`	I	IREQ IRQ ISR entry address

Table 3-9 CSF_REQ_CONFIG Flags Field

Location	Description
Bit 0	0 = Set up the memory-only interface 1 = Set up the I/O interface
Bit 1	0 = Disable the IREQ IRQ 1 = Enable the IREQ IRQ
Bit 2	0 = Set default IREQ mode on the card 1 = Set Bit3-specified IREQ mode on the card
Bit 3	0 = Set level-mode IREQ on the card 1 = Set pulse-mode IREQ on the card (valid only if bit 2 is set)
Bit 4-7	Unused
Bit 8	0 = Configuration Option Register not present 1 = Configuration Option Register present
Bit 9	0 = Card Configuration and Status Register not present 1 = Card Configuration and Status Register present
Bit 10	0 = Pin Replacement Register not present 1 = Pin Replacement Register present
Bit 11	0 = Socket and Copy Register not present 1 = Socket and Copy Register present

`CSF_REQ_CONFIG` sets the socket to the mode specified in the `Flags` field with the specified voltage levels and IRQ parameters.

The card is configured by writing the specified values to the card configuration registers. Registers not set in `CSF_REQ_CONFIG` are presumed to be absent. In particular, no Pin Replacement Register processing will be performed for `CSF_GET_STATUS` calls unless Bit 10 of the `Flags` field is set in the most recent call to `CSF_REQ_CONFIG`.

Example

```
#include <csf.h>

config_parm_pkt pb;

pb.code     = CSF_REQ_CONFIG;
pb.flags    = CFG_IO | ENABLE_IREQ | DEFAULT_IREQ_MODE |
             SET_IREQ_LEVEL | CFG_CONFIG_OPT_REG;
pb.Vcc      = VOLTS_5_0;
pb.Vpp      = VOLTS_0_0;
pb.base     = <CFG REG BASE ADDRESS>;
pb.option   = CFG_IREQ_LEVEL | <IREQ INDEX>;
pb.status   = 0;
pb.vector   = <DEVICE INTERRUPT VECTOR>;

error_code = _os_setstat((path_id) path, (u_int32)
SS_CSF,
                         (void *) &pb);
```

Errors

`EOS_CSF_NOCARD`	No card in the socket
`EOS_CSF_INUSE`	Card is already configured
`EOS_CSF_BADIRQ`	Bad IRQ vector or level
`EOS_CSF_BADVCC`	Bad V_{cc} value
`EOS_CSF_BADVPP`	Bad V_{pp} value, or invalid V_{cc}/V_{pp} combination

I_SETSTAT, CSF_REL_CONFIG

Release a Configuration

Function Code

```
#include <csf.h>
I_SETSTAT, CSF_REL_CONFIG
```

Parameter Block

```
typedef struct config_parm_pkt
{
   csf_api_code code;
   u_int16      flags;
   u_int8       Vcc;
   u_int8       Vpp;
   u_int32      base;
   u_int8       option;
   u_int8       status;
   u_int8       pin_rep;
   u_int8       socket_copy;
   u_int16      vector;
   u_int16      level;
   void*        entry;
}
config_parm_pkt, *Config_parm_pkt;
```

Description

`CSF_REL_CONFIG` releases an existing configuration. When a configuration is released, you can call `CSF_REQ_CONFIG` again.

Table 3-10 CSF_REL_CONFIG Parameter Block

Parameter	I/O	Description
`u_int16`	`I`	Ignored
`u_int8`	`I`	Ignored

Table 3-10 CSF_REL_CONFIG Parameter Block (continued)

Parameter	I/O	Description
u_int8	I	Ignored
u_int32	I	Ignored
u_int8	I	Ignored
u_int8	I	Ignored
u_int8	I	Ignored
u_int8	I	Ignored
u_int16	I	Ignored
u_int16	I	Ignored
void*	I	Ignored

Example

```
#include <csf.h>
config_parm_pkt pb;

pb.code     = CSF_REL_CONFIG;

error_code = _os_setstat((path_id) path, (u_int32)
SS_CSF,
                         (void *) &pb);
```

Errors

EOS_CSF_NOCARD	No card in the socket
EOS_CSF_NOCONFIG	Card is not configured

Function Code

```
#include <csf.h>
I_SETSTAT, CSF_REG_CLIENT
```

Parameter Block

```
typedef struct client_parm_pkt
{
   csf_api_code code;
   u_int32      handle;
   u_int32      signal;
   u_int32      att_win_base;
   u_int32      att_win_size;
   u_int32      proc_id;
   u_int16      flags;
   u_int16      mask;
}
client_parm_pkt, *Client_parm_pkt;
```

Description

`CSF_REG_CLIENT` registers a process with the CSF file manager to be informed when certain PC Card events occur. Card insertion and removal are the most important events, but several others may be received, depending on the hardware and software capabilities of a particular implementation.

In signal notification, a system signal is sent to a process when the event occurs.

.

Table 3-11 CSF_REG_CLIENT Parameter Block

Parameter	I/O	Description
`u_int32`	`O`	Client Handle
`u_int32`	`I`	Signal to send on events
`u_int32`	`O`	The memory address of the mapped PC Card CIS
`u_int32`	`O`	The size of the memory area dedicated to the PC Card CIS
`u_int32`	`I`	Process ID to receive signal
`u_int16`	`I`	Flags (see Table 3-13 CSF_REG_CLIENT Flag Field)
`u_int16`	`I`	Event Mask

Table 3-12 CSF_REG_CLIENT Parameters (Flag bit 3 not set or OS-9 host)

Parameter	I/O	Description
`u_int32`	`I`	Signal to send on events
`u_int32`	`I`	Process ID to receive signal

If a Process ID of zero is specified, the caller's process ID will be used. The signal value specified to be sent should be a value approved for this purpose and can be found in the supporting implementation include files.

Table 3-13 CSF_REG_CLIENT Flag Field

Location	Description
Bit 0	0 = This is a memory client 1 = This is an I/O client
Bit 1-2	Reserved
Bit 3	0 = Notify by signal (Bit 3 always set to 0) 1 = Notify by callback (this setting not used)

In the OS-9 implementation, the `Flags` field is ignored, and signal notification is assumed.

Table 3-14 CSF_REG_CLIENT Event Mask Field

Location	Description
Bit 0	Notify on Write Protect status change
Bit 1	Notify on Card Lock change
Bit 2	Notify on Ejection Request
Bit 3	Notify on Insertion Request
Bit 4	Notify on Battery-dead change
Bit 5	Notify on Battery-low change

Table 3-14 CSF_REG_CLIENT Event Mask Field

Location	Description
Bit 6	Notify on Ready change
Bit 7	Notify on Card Detect change
Bit 8	Notify on Power Management change
Bit 9	Notify on Card Reset

A particular implementation ignores any bits that ask for notification for unsupported events. To ask for notification on all events specify `0xffff` as the mask. This is not generally a good idea, though, because there may be overhead introduced by requiring the CSF file manager to look for unsupported events.

Note

The OS-9 implementation does not support callback functions and does not distinguish between memory and I/O clients.

Example

```
#include <csf.h>

client_parm_pkt pb;

pb.code     = CSF_REG_CLIENT;
pb.signal   = SCKT_EVENT_SIG + <SOCKET NUMBER>;
pb.proc_id = 0;
pb.flags    = 0;
pb.mask     = 0;

error_code = _os_setstat((path_id) path, (u_int32)
SS_CSF,
                                (void *) &pb);
<SAVED HANDLE>        = pb.handle;
<SAVED WINDOW BASE> = pb.att_win_base;
<SAVED WINDOW SIZE> = pb.att_win_size;
```

Errors

`EOS_CSF_ALLOCERR`	No more client registrations allowed

I_SETSTAT, CSF_DER_CLIENT

De-register a Process

Function Code

```
#include <csf.h>
I_SETSTAT, CSF_DER_CLIENT
```

Parameter Block

```
typedef struct client_parm_pkt
{
   csf_api_code code;
   u_int32      handle;
   u_int32      signal;
   u_int32      att_win_base;
   u_int32      att_win_size;
   u_int32      proc_id;
   u_int16      flags;
   u_int16      mask;
}
client_parm_pkt, *Client_parm_pkt;
```

Description

`CSF_DER_CLIENT` uses the same parameter block as `I_SETSTAT, CSF_REG_CLIENT` but the only field used is the client handle. On some systems the client handle may be checked to verify that the calling process is, in fact, its owner.

To change event notification you must register again for notification with the new event mask, then deregister the old handle.

Example

```
#include <csf.h>

client_parm_pkt pb;

pb.code     = CSF_DER_CLIENT;
pb.handle   = <SAVED HANDLE>;
```

```
error_code = _os_setstat((path_id) path, (u_int32)
SS_CSF,
                                (void *) &pb);
```

Errors

`EOS_CSF_BADHANDLE`	Client handle is invalid

I_SETSTAT, CSF_GET_TUPLE

Read CIS Entries

Function Code

```
#include <csf.h>
I_SETSTAT, CSF_GET_TUPLE
```

Parameter Block

```
typedef struct get_tuple_parm_pkt
{
   csf_api_code code;
   u_int8       findcode;
   u_int8       tcode;
   u_int8       tlink;
   u_int8       tdata[255];
}
get_tuple_parm_pkt, *Get_tuple_parm_pkt;
```

Description

`CSF_GET_TUPLE` reads selected CIS entries from an open attribute memory window. The parameters necessary for the read are set when the first window is opened.

Table 3-15 CSF_GET_TUPLE Parameter Block

Parameter	I/O	Description
`u_int8`	`I`	Desired tuple code (`0x00` for first; `0xff` for next)
`u_int8`	`O`	Tuple Code
`u_int8`	`O`	Tuple Link
`u_int8[255]`	`O`	Tuple Data

To use this function, a memory window to an inserted card's attribute-memory area must have been previously opened. This memory window should not be modified or released as long as the `CSF_GET_TUPLE` function is parsing CIS tuples in the attribute-memory area of the inserted card.

The tuple code may be changed between calls, in which case the new code will be scanned for, starting at the CIS tuple immediately following the last one returned (except for a code of `0x00`).

Example

```
#include <csf.h>

get_tuple_parm_pkt pb;

pb.code     = CSF_GET_TUPLE;
pb.findcode = 0x00;          /* reset tuple pointer */

error_code = _os_setstat((path_id) path, (u_int32)
SS_CSF,
                         (void *) &pb);

pb.findcode = CISTPL_CFTABLE_ENTRY; /*get a cfg table
tuple*/

error_code = _os_setstat((path_id) path, (u_int32)
SS_CSF,
                         (void *) &pb);
```

Errors

`EOS_CSF_NOATTRWIN`	No attribute memory window opened
`EOS_CSF_WINOVERRUN`	Attempted to read past window boundary (bad CIS)

I_SETSTAT, CSF_REMOVE_DEV

Detach and Remove Device

Function Code

```
#include <csf.h>
I_SETSTAT, CSF_REMOVE_DEV
```

Parameter Block

```
typedef struct remove_dev_parm_pkt
{
   csf_api_code code;
   u_int8       name[12];
}
remove_dev_parm_pkt, *Remove_dev_parm_pkt;
```

Description

`CSF_REMOVE_DEV` detaches a device and removes the device from the system device list. This function provides a recovery path when an OS-9 device has been attached for an inserted PC card and the card is then removed without the device having been detached.

The parameter block for `CSF_REMOVE_DEV` is shown in Table 3-16 CSF_REMOVE_DEV Parameter Block.

Table 3-16 CSF_REMOVE_DEV Parameter Block

Parameter	I/O	Description
`u_int8[12]`	I	Name of device to remove

Example

```
#include <csf.h>

remove_dev_parm_pkt pb;

pb.code = CSF_REMOVE_DEV;
```

```
strcpy(pb.name, <DEVICE NAME>);

error_code = _os_setstat((path_id) path, (u_int32)
SS_CSF,
                              (void *) &pb);
```

I_SETSTAT, CSF_REGISTER_OP

Operate Socket Controller Registers

Function Code

```
#include <csf.h>
I_SETSTAT, CSF_REGISTER_OP
```

Parameter Block

```
#define REG_READ  0
#define REG_WRITE 1
#define REG_OR    2
#define REG_AND   3
#define REG_EOR   4

typedef struct register_op_parm_pkt
{
   csf_api_code code;
   u_int8       op;
   u_int8       reg;
   u_int8       val;
}
register_op_parm_pkt, *Register_op_parm_pkt;
```

Description

`CSF_REGISTER_OP` enables direct interaction with the registers of the socket controller. `CSF_REGISTER_OP` is machine dependent and dangerous. It is provided for testing and for implementation of socket maintenance utilities.

Table 3-17 CSF_REGISTER_OP Parameter Block

Parameter	I/O	Description
`u_int32`	`I`	Desired operation code (see Table 3-18 CSF_REGISTER_OP Operations Codes)
`u_int8`	`I`	Register Number (typically 0 - 63)
`u_int8`	`I/O`	Register Value/Modifier

To use this function, specify one of the following operations in the operation code field:

Table 3-18 CSF_REGISTER_OP Operations Codes

Code	Operation Description
0	Read register (modifier = reg)
1	Write register (reg = modifier)
2	OR to register (reg = reg | modifier)
3	AND to register (reg = reg & modifier)
4	EOR to register (reg = reg ^ modifier)

Example

```
#include <csf.h>

register_op_parm_pkt pb;

pb.code = CSF_REGISTER_OP;
pb.op    = REG_OR;
pb.reg   = 0x00;
pb.val   = 0x00000001;

error_code = _os_setstat((path_id) path, (u_int32)
SS_CSF,
                       (void *) &pb);
```

Errors

`EOS_CSF_BADPARM`	Bad operation code

Error Return Codes

All CSF functions return an error code of 0 to indicate success. Codes returned due to non-system error conditions are defined here. The listed error is the low-order word of an error code whose high-order word identifies the CSF subsystem.

Table 3-19 Error Return Codes

Error Code	Description
`EOS_CSF_BADIRQ`	Bad IRQ vector or level specified
`EOS_CSF_BADVCC`	Bad Vcc value specified
`EOS_CSF_BADVPP`	Bad Vpp value or Vcc/Vpp combination
`EOS_CSF_ALLOCERR`	No window space available
`EOS_CSF_BADOFFSET`	Card offset is invalid
`EOS_CSF_NOCARD`	No card in the socket
`EOS_CSF_INUSE`	Socket already configured
`EOS_CSF_NOCONFIG`	Socket is not configured
`EOS_CSF_BADFLAG`	Bad bit or bits in a Flags field

Appendix A: Address Space Utilization by CSF

Note
This document does not address the static variable memory needs of the CSF file manager, the CSF daemon, and the various socket and PC card device drivers.

Note
All values are in hex unless otherwise indicated.

Note
MEM denotes a processor memory space address. I/O denotes a processor I/O space address.

Attribute Memory Windows for CIS Parsing

This memory is allocated by csfd from device descriptors for up to a maximum of 4 sockets.

Table A-1 Attribute Memory Windows for CIS Parsing

Port/Driver	Attribute Memory (MEM:)			
X86/PC (cs82365 driver)	`fff000 -ffffff`	`ffe000 -ffefff`	`ffd000 -ffdfff`	`ffc000 -ffcfff`
MPC821/821ADS (cs821 driver)	`40000000 -40000fff`	`40001000 -40001fff`		
MPC821/MBX821 (cs821 driver)	`e8000000 -e8000fff`	`e8001000 -e8001fff`		
MPC602/NEC STB (cs1130 driver)	`c23fe000 -c23fefff`	`c23ff000 -c23fffff`		
CL-7111ARM/ CL-PS7111 Board (cs6700 driver)	`40000000 -400003ff`	`50000000 -500003ff`		

Socket Controller Register Access

Socket controller register access for devices `cs0`, `cs1`, `cs2`, and `cs3` are shown in the following table.

Table A-2 Socket Controller Register Access

Port/Driver	Register Access			
X86/PC (cs82365 driver)	I/O: 3e0-3e1			
MPC821/821ADS (cs821 driver)	MEM:[a] 02200080 -0220011b			
MPC821/MBX821 (cs821 driver)	MEM:[a] fa200080 -fa20010b			
MPC602/NEC STB (cs1130 driver)	MEM: 800003e0 -800003e1	MEM: 80804000 -808040ff	MEM: 80804100 -808041ff	
CL-7111ARM/ CL-PS7111 Board (cs6700 driver)	MEM: 80000001	MEM: 80000041	MEM: 80000043	MEM: 80000400
	MEM: 80000003			

a. These values are contingent on the `immr` value.

Optional Hardware Control Accesses

The following table shows the optional hardware accesses.

Table A-3 Optional Hardware Contol Access

Port/Driver	Register Access
X86/PC (cs82365 driver)	`I/O: 042, 043, 061` (Bell)
MPC821/821ADS (cs821 driver)	`MEM: 02100004` (BCSR1)
MPC821/MBX821 (cs821 driver)	`MEM: fa00001` (BCSR2)
MPC602/NEC STB (cs1130 driver)	none
CL-7111ARM/ CL-PS7111 Board (cs6700 driver)	`MEM: 00002000` (CPCR)

Serial Modem Cards

The following table shows the serial modem card registers access for devices `t1`, `t2`, `t3`, and `t4`.

Table A-4 Serial Modem Card Register Access

Port/Driver	Register Access			
X86/PC (cs82365 driver)	`I/O (t3): 3e8-3ef`	`IO (t4): 2e8-2ef`	`I/O (t2): 2f8-2ff`	`I/O (t1): 3f8-3ff`
MPC821/821ADS (cs821 driver)	`MEM (t2): 800003f8 -800003ff`			
MPC821/MBX821 (cs821 driver)	`MEM: e00003f8 -e00003ff`			
MPC602/NEC STB (cs1130 driver)	`MEM (t2): 800002f8 -800002ff`			
CL-7111ARM/ CL-PS7111 Board (cs6700 driver)	`MEM: 480003f8 -480003ff`	`MEM: 580003f8 -580003ff`		

ATA/IDE Hard Disk Cards

The following table shows the registers access for ATA/IDE hard disk cards for devices `mhp1` and `mhc1`.

Table A-5 Socket Controller Register Access

Port/Driver	Register Access			
X86/PC (cs82365 driver)	`I/O:` `170-177`	`I/O:` `376-377`	`I/O:` `1f0-1f7`	`I/O:` `3f6-3f7`
MPC821/821ADS (cs821 driver)	`MEM:` `800001f0` `-800001f7`	`MEM:` `800003f6` `-800003f7`		
MPC821/MBX821 (cs821 driver)	`MEM:` `ec0001f0` `-ec0001f7`	`MEM:` `ec0003f6` `-ec0003f7`		
MPC602/NEC STB (cs1130 driver)	`MEM:` `800001f0` `-800001f7`	`MEM:` `800003f6` `-800003f7`		
CL-7111ARM/ CL-PS7111 Board (cs6700 driver)	`MEM:` `480001f0` `-480001ff`	`MEM:` `480003f6` `-480003f7`	`MEM:` `580001f0` `-580001ff`	`MEM:` `580003f6` `-580003f7`

3Com 3c589 Network Card

The following table shows the registers access for the 3COM 3c589 Network Card.

Table A-6 3COM 3c589 Network Card Register Access

Port/Driver	Register Access
X86/PC (cs82365 driver)	`I/O: 250-25f`
MPC821/821ADS (cs821 driver)	`MEM: 80000250-8000025f`
MPC821/MBX821 (cs821 driver)	`MEM: ec000250-ec00025f`
MPC602/NEC STB (cs1130 driver)	`MEM: 80000250-8000025f`
CL-7111ARM/ CL-PS7111 Board (cs6700 driver)	`MEM: 48000250-4800025f`

Appendix B: Port-Specific Technical Notes

This appendix contains various notes about the operation of certain PC Cards on specific systems.

ATA Disk PC Card and 386/PC

The PC Card standard addresses for ATA Disk are `0x1f0` for the primary and `0x170` for the secondary controller. Most current x86/PC systems have an on-board ATA Disk/IDE Disk interface and many have a secondary controller that is also on the motherboard. Because of this, in the PC architecture, the `mhe1` descriptor is chosen as the first PC Card device. The `mhe1` descriptor corresponds to the secondary controller `0x170` address.

If your base system has a secondary controller on the motherboard, you may need to disable this controller before using a PC Card ATA disk. Some computers allow this to be done through the BIOS while others may require a jumper change.

CS6700 Driver Notes

The cs6700 driver is not interrupt driven. The CSF Daemon must be executed with polling set. A typical command for use of the daemon in this environment is:

```
csfd -igvdel <>>>csfd.log&
```

where the RAM disk is the default data directory. Any interrupt-driven drivers used with PC Cards in this environment must have the descriptors modified to specify an interrupt vector of `0x45`.

To modify the descriptors:

Step 1. Open the `config.des` file.

Step 2. Locate the macro `PCSCVECT`.

Step 3. Change the interrupt vector value to `0x45`.

Step 4. Save and close the `config.des` file.

Step 5. Remake your descriptors.

CS82365 Driver and Descriptors Testing Notes

The `cs82365` driver and its associated descriptors, `cs1, cs2, cs3, cs4`, have been successfully tested on a Wang LP451 80486-33DX with the following AT-Bus PCMCIA Socket Controllers:

- SCM Microsystems MMCD-VBL Single-Socket Controller (Vadem VG465 PCSC)
- Cardwell Model Dual-Socket Controller (Vadem VG365 PCSC)

You may need to specify a different interrupt vector in the socket device descriptor (the above configuration is using vector `0x4c`). A possible alternate value is `0x4b` or operate without interrupts using the `csfd` polling mode.

To modify the descriptors:

Step 1. Open the `config.des` file.

Step 2. Locate the macro `PCSCVECT`.

Step 3. Change the interrupt vector value to `0x4b`.

Step 4. Save and close the `config.des` file.

Step 5. Remake your descriptors.

The Intel 82365 and Vadem VG365 PCSC devices each can interface two sockets. Unless a special I/O port address decode of the device (other than the standard `0x3e0`) is made, any board/system based on either of these devices can interface only two sockets. The Vadem VG465 PCSC has hardware provision for daisy-chaining up to 4 devices at the same I/O port address, which permits up to 4 sockets to be supported from a single port in a given system.

Architecture Limitations Affecting PC Card Support

The following sections discuss architecture limitations that may affect how you use CardSoft in a particular implementation.

x86

X86 ISA bus controllers must map all card space in addresses below 16 MB. Because of this limitation of the ISA bus, your system must have less than 16 MB in order to recognize and utilize PC Cards correctly. A PC BIOS may allow you to unmap the top 1 MB of memory in a 16 MB system. If your system BIOS supports this, then it may be possible to use the CSF product in a PC with 16 MB of RAM. You must have no more than 8 MB of RAM for a linear (non-ATA) flash card to be mapped into memory and used correctly with `rbftl`. This is also because the card must be mapped into memory in the ISA bus range. This is not a problem with an ATA flash disk card. If you wish to use an ATA disk or modem card in a PCAT host with 16 MB or more RAM, it may be possible to remap the card attribute memory to an unused portion of the 640-1024 KB region of memory. A good first choice may be `0xD0000`. Editing the CSF `config.des` file in the `PORT` directory and rebuilding the descriptors is all that would be required.

Newer x86 hosts often have a secondary IDE controller enabled on the motherboard (or secondary card). The PC standard for ATA/IDE devices only allows for two such masters in system, the first appearing at `0x1f0` and the second at `0x170`. The `mhe1` descriptor for x86 is configured as the second master in a system. To use a PC Card ATA disk device in a system with a primary and secondary controller, you may be required to disable the secondary controller through the BIOS (if supported) or by changing a jumper configuration. A similar problem for serial devices on x86 is solved by deactivating a second serial port on your host.

821ADS

821ADS PILOT, 821ADS Rev. A, and 821ADS Rev. B suffer from a problem where executable code running out of flash on the board cannot access a PC Card correctly. The 821ADS ENG board does not suffer from this problem and it is not a limitation of the MPC821 processor. If you need to test with such a configuration, load the OS into RAM at system startup (use `lr` instead of `bo`).

821ADS PILOT, 821ADS Rev. A, and 821ADS Rev. B suffer from another problem, in that a linear (non-ATA) flash device is not correctly accessible from code running out of Flash or RAM. The 821ADS ENG board does not suffer from this problem. There is no known work-around on newer ADS boards.

Index

Product Discrepancy Report

To: Microware Customer Support

FAX: 515-224-1352

From:____

Company:____________________________________

Phone:_______________________________________

Fax:____________________Email:_________________

Product Name:

Description of Problem:

Host Platform___________________________________

Target Platform_________________________________

RadiSys.
MICROWARE SOFTWARE