Plug'N'Play Unix

Andrew McRae

MITS Real Time Ltd.
2/37 Waterloo Rd
North Ryde
andrew@mega.com.au

This paper presents the design and implementation of a Plug and Play system for FreeBSD, a freely available version of BSD Unix†, based around PCMCIA cards (PC-Card). A brief introduction of the PC-Card architecture is given, followed by a description of the FreeBSD implementation. It describes the internal workings of card recognition, and the automatic installation and configuration of the cards into the Unix kernel, and how this affects system configuration and setup.

1. Introduction.

The evolution of Laptop PCs has been rapid and impressive. New hardware features such as power management, low power LCD screen displays and low voltage high performance CPUs and memory have increased the power and functionality of mobile computers significantly. One such development has been the widespread acceptance of the Personal Computer Memory Card International Association (PCMCIA) standard, a standard for connecting small circuit cards to computing equipment, now renamed the PC-Card standard (which is a whole lot easier to say than P-C-M-C-I-A).

Unfortunately, Laptops almost exclusively use DOS/Windows, which has notoriously poor support for networking and communications. Virtually all PC-Card products are centered around the DOS/Windows world, and emerging standards such as Exchangeable Card Architecture (ExCA) and Execute In Place (XIP) are almost totally geared towards a DOS/Windows environment.

Unix, on the other hand, has generally been a non-migratory beast, usually residing on machines that are too heavy to lift let alone carry on a plane. With the advent of the BSD ports of Unix to the PC environment, it is now possible to take advantage of the Laptop PC technology to have a mobile Unix environment. *FreeBSD* is one such freely available port. It seemed logical to integrate the PC-Card technology into FreeBSD as a way of leveraging Unix into the mobile computing world, and the result is presented in this paper.

2. PC-Card Overview.

The PC-Card architecture was originally designed as a memory card system, allowing easy transfer of semiconductor based mass storage. The standard was revised (Release 2.01) in 1991 to incorporate peripheral devices such as communication cards, network cards etc. The next major revision (Release 2.1) standardised support for such devices such as modems. Support has grown for the system such that it is now seen as a standard I/O bus for desktop as well as mobile computers.

In its basest form, the PC-Card architecture can be thought of as simply another peripheral bus connection standard, but it has been designed with a number of features that make it interesting:

- A very small form factor so that cards may have the physical dimensions of a thick credit card. Whilst all PC-Cards have the same width and height, the depth of the card may vary from 3.3 mm (Type I) to 10.5 mm (Type III). Some cards may have an extended length to allow extra room for

† Unix is a trademark of *someone* at the moment, but I forget just who it is this week.

peripheral devices etc.

- Environmently and physically the card is robust, being completely cased in a mechanically stringently specified surrounding.
- The card and connector is designed for manual insertion and removal, even while the socket is powered up ('hot-swapping'). The connector is keyed to prevent reverse installation, and is specified for up to 10,000 insertion and removal cycles.
- Every card contains a Card Information Structure (CIS), describing in a Card Metaformat the exact type, capabilities and interfacing requirements of this particular card. Thus an inserted card may be queried to determine what kind of card it is and how it should be interfaced. The CIS consists of a series of self-describing configuration *tuples*.

The PC-Card architecture is a natural contender for implementing a true Plug and Play system, since every card can be easily inserted or removed, and every card can be uniquely identified and interfaced as required.

2.1. PC-Card Interfacing.

Figure 1 shows a typical Laptop implementation of a PC-Card interface, where a slot/socket controller is used to interface to two separate card slots.

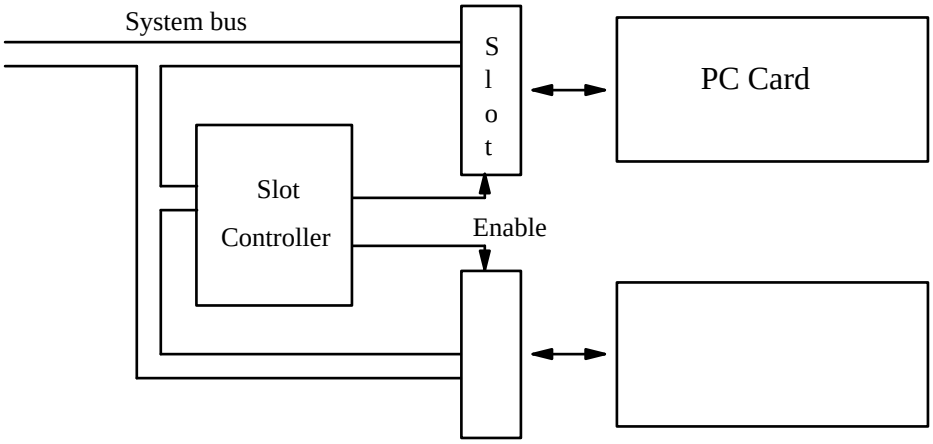

Figure 1: Slot Interface

The slot controller is typically a device such as an Intel 82365 or Databook TCIC. The controller (along with the support circuitry) provides monitoring of the slot connectors, generation of card events such as card insertion or removal, power control, bus buffering enabling, address decoding and interrupt steering. The system bus connects to the cards through the intermediatary of the slot controller.

2.2. Memory Architecture.

PC-Cards, once connected to the system, are accessed similar to other devices via the memory and I/O bus. Each PC-Card has the capability of two distinct memory address spaces, *Common* and *Attribute*. The slot controller has a number of *memory windows*, each of which translates system addresses to corresponding address areas in one of the two address spaces on the PC-Card (as shown in figure 2).

Each PC-Card slot has 26 address lines, allowing a total addressing capability of 64 Mbytes for both Common and Attribute memory. Attribute memory locations only exist on the even address bytes, and is intended to contain the card's CIS and other configuration registers. Every card *must* respond to Attribute memory accesses (though a neat trick in the CIS tuples effectively allows a card to contain no separate Attribute memory). Attribute memory can therefore only be accessed via byte wide memory references (the odd byte is ignored on 16 bit accesses).

Common memory is accessed using standard 16 bit or 8 bit memory references, and is intended to contain the shared memory areas of the card that the system can access.

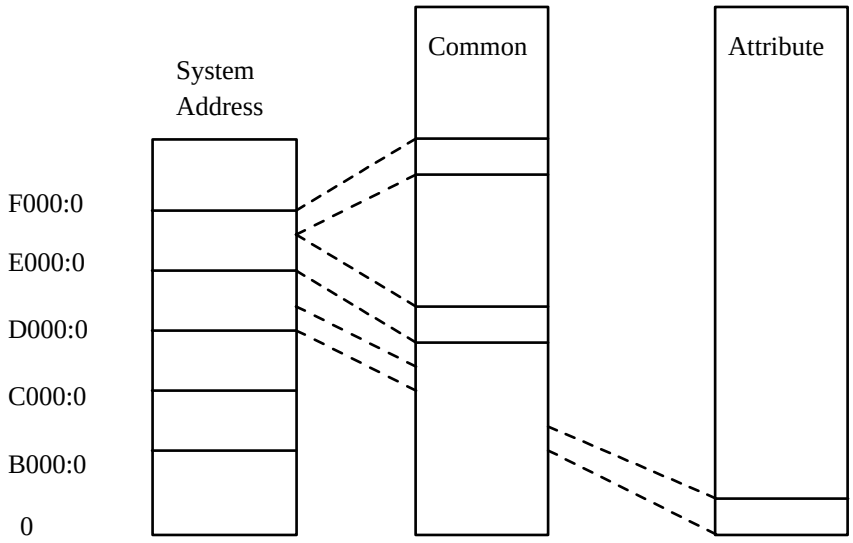

Figure 2: Memory Mapping

If the card is an I/O card (as opposed to a memory-only card), then the system I/O bus may be used to access up to 64K I/O ports on the card. Cards (within their CIS) indicate the number of I/O lines that are decoded within the I/O address space; cards may decode I/O port addressing themselves, or allow the slot controller to decode the I/O addresses and enable the slot accordingly.

Interrupts may be generated by cards, and slot controllers have *interrupt steering* capability, allowing the system to configure each card's interrupt to appear on different interrupt lines.

The combination of memory and I/O address windows and interrupt steering allow (given a compliant card configuration) to relocate and manage the resources required for each card dynamically, without address pre-allocation or presetting. This is very different from the ISA model, where the configuration of the memory, I/O and interrupt resources is at best a complicated and restrictive exercise.

2.3. PC-Card Metaformat.

The organisation of the card's CIS is the key to the plug and play nature of the PC-Card architecture. The structure is a series of self-describing tuples (see figure 3), each of which contains a code and a link to the next tuple. The link is effectively the length of the data following, allowing easy traversal of the whole CIS structure.

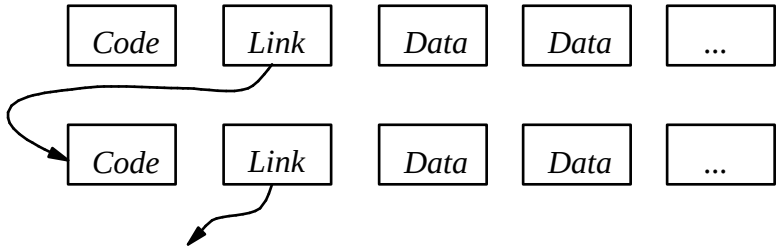

Figure 3: Metaformat Tuples

The PC-Card standard has defined a number of tuple codes which are used to identify the card environment such as power requirements, memory blocks available, I/O addressing etc. Each tuple definition has an optional variable sized data block associated with it that contains parameters for the particular configuration item being described. Not all cards need to contain all the tuple codes; a minimum subset is defined for all cards, however.

The following table provides a sample set of tuple codes:

Code	Use
00	Null tuple (ignored)
01	Describes device parameters for common memory devices
10	Checksum
11	Link to tuple list in Attribute memory
15	Version data describing manufacturer etc.
1A	Configuration data header
1B	Configuration entry
21	Function identification
22	Functional extension (extra data for e.g modems)
FF	End of tuple list

Cards may contain multiple configuration entries, each of which defines different card setup parameters such as I/O address decoding, memory arrangement etc. The system software will select one of the configuration entries and then enable that entry by writing a specified value to a configuration control register in attribute space. The location of this register is given in one of the CIS tuples.

The decoding and understanding of the CIS is one of the more complex areas of PC-Card processing, especially when card vendors produce incomplete or inaccurate CIS entries (not an uncommon problem).

3. FreeBSD Integration.

The challenge of integrating PC-Card technology into FreeBSD is not so much of 'Can it be done?', but more 'How to get the most out of the technology?'. The one existing driver that incorporated some PC-Card support (a Network driver) was written for one specific slot controller, and worked on only one kind of Network card, and everything was done within the network driver. No insertion or removal was handled, and there was no method of allocating dynamic resources such as memory and I/O port allocation etc.

The general plan was to develop an implementation with the following goals:

- Provide a user-level configuration file containing enough information to allow card recognition and setup without writing any card-specific code.
- Support the 'hot' insertion and removal of cards, enabling or disabling the card drivers as needed.
- Dynamically allocate system resources such as memory areas, I/O ports and interrupts from a pool of available resources.
- Support multiple types of slot controllers. Different Laptops have different slot controllers, and the idea is to provide a consistent user or kernel level interface to all controllers.
- As much as possible, use existing drivers. This is a realistic goal, since most cards emulated ISA devices quite closely.
- Have some fun with my new Laptop.

Extended goals for the PC-Card implementation was to allow some research into plug and play concepts, and support mobile communications via modem, packet radio, wireless LAN etc.

3.1. Overall Design.

Figure 4 shows an overall view of the PC-Card interface to FreeBSD. The general idea was to provide a number of interacting entities that would be extendable and allow new ideas to be incorporated without having to re-work the whole system. The two key elements are a user level daemon ('*pcmciad*') which reads a configuration file ('*/etc/pcmcia.conf*') describing the cards and associated configuration data, and a socket/slot device driver which interfaces the socket controller to the system. The driver and daemon communicate via a typical read/write/ioctl interface.

Part of the configuration action of the daemon is to *exec* external programs to configure installed cards, such as running *ifconfig* etc. The daemon is normally inactive, and waits on a *select* to the slot driver. When a card is installed or removed, the select comes true and the daemon queries the driver for the particular event. Each slot is assigned a separate minor device number, which allows each access to each slot using standard tools such as *cat*, *dd* and *hexdump*.

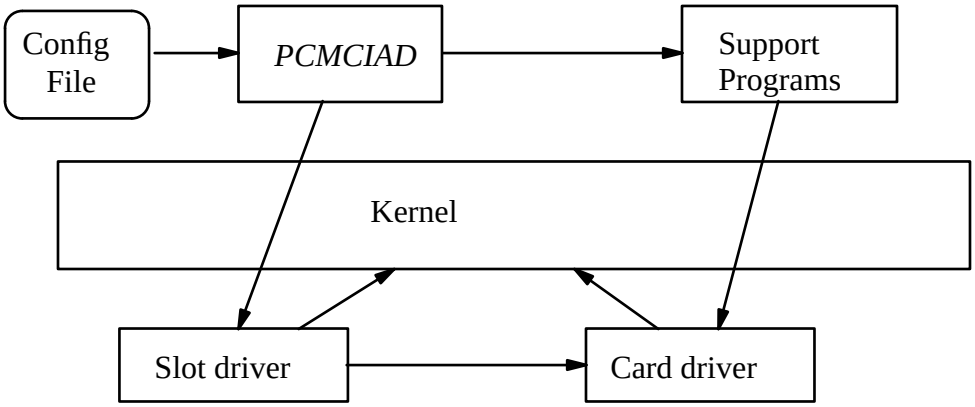

Figure 4: Overall Design

3.2. Slot Driver.

As discussed, the slot driver is accessed via a set of ioctl calls and read/write calls. The ioctl calls are used to manage the slot resources such as memory and I/O windows, and provide hooks for allocation of interrupts and drivers to the slot. By default, a portion of ISA memory address space is allocated and shared between all slots for use in a read/write call. When the read or write entry for the driver is called, a spare memory window for that slot is configured with the ISA memory address as the host address, with the file offset providing the card memory space address. The read/write call is then performed as a simple memory copy. If the size of the request is too large for the memory window, the request is broken in smaller blocks and the memory window is reconfigured for each block.

The use of read/write for the card memory interface means that the slot driver does not need to know anything about the card's CIS structure, nor about the reading or writing of any of the card's configuration registers existing in the Attribute memory space. User programs can access these registers simply by doing an *lseek* followed by a *write*.

The slot driver also maintains a state for each slot, which includes the contexts for the I/O and memory windows, the interrupt number assigned to the slot etc.

3.3. Card Driver.

Once a card is installed, eventually a driver is assigned to the card. The driver is a normal device driver, such as for a serial UART port or network controller. A BSD Unix driver normally is only set up at system configuration time, at which time each driver's *probe* routine is called to test whether a device exists at the desired address. If a probe is successful, then the driver's *attach* routine is called to complete the device setup. PC-Card drivers are similar, except a probe at system startup will fail since the card configuration as not been set up by the daemon, or a card may be plugged in after the system is started.

Once a driver is probed, it is assumed (naturally) to always exist at the probed location and interrupt number. No normal BSD Unix driver *expects* to have its device untimely ripped from its socket's connector! This of course can happen using PC-Card technology, but at this stage little work has been done on the existing drivers to incorporate this change. Current drivers rely on a device reset to ensure the hardware is set up. Conversely, when a card is removed, the best that can happen at the moment is that the daemon will execute a support program disabling the device, such as executing an *ifconfig xx0 down* in the event of a network card removal.

The latest release of FreeBSD allows kernel loadable modules; this would mean easy device driver loading and unloading. Unfortunately, this is not allowed with some of the more interesting drivers such as network drivers.

3.4. Kernel Changes.

Little change has been made to the core FreeBSD kernel; the ISA routines have been extended to allow lookup of a kernel device table using a name and unit number, and some early changes allowed device probes to be called after system startup.

Some miscellaneous changes were made to make it easier to install generic devices, such as allowing the physical Ethernet address to be assigned via an ioctl to the network driver.

3.5. Daemon and Configuration File.

Most of the hard work is done by the *pcmciad* program, which parses the configuration file and attempts to install the right driver for that card, which has been hopefully configured at the expected memory and I/O address with the correct interrupt number. This involves reading and decoding the CIS tuples, and matching the card name/version to one in the configuration file, selecting a valid configuration from the CIS tuples, and trying to match it with the free resources available, and then enabling the card resources, and then assigning a driver to the card.

A typical configuration file would look thus:

```
#
# Sample configuration file.
#
# Pool parameters.
#
io 0x2F8 - 0x360
irq 5 6 8 9 10 15
memory 0xd4000 96k
#
# Card database.
#
card "RPTI LTD." "EP400" net  # NE2000 clone
        ether 0x110
        config 0x30 "ed0" 5
        config 0x31 "ed1" 6
        insert ifconfig $device physical $ether

card "RIPICAA" "RC144ACL" tty
        config 0x21 "sio1" 10

device net
        insert ifconfig $device bean
        remove ifconfig $device down

device net
        insert ifconfig $device bean-2
        remove ifconfig $device down

device tty
        insert echo start getty here
        remove echo stop getty
```

A pool of free card resources is defined, followed by a card database and different device descriptions for each instance of the different device classes. Once a card is recognised and installed, the daemon remembers the card setup even if the card is removed, since once the driver is probed and attached, the driver's resources are allocated permanently and so those resources are not available to other devices. Re-inserting the card will then re-allocate the same resources as before.

A typical series of events following a card insertion is:

1. The slot driver receives a card event interrupt, signalling a card insertion.
2. The card is reset and powered up, and the card event flag for the slot device select is set true.
3. The daemon (if it is waiting on the select) will register a change (via select) and read the new card status.
4. The card's CIS is read and decoded.
5. The CIS version and manufacturer name is matched against the card database from the configuration file. If none is found, no more processing is done.
6. The configuration entries from the CIS are matched against the entries in the file, and one selected that matches the free resources.
7. The memory, I/O and interrupt resources are allocated to the card, and the memory and I/O contexts programmed for that slot.
8. The slot driver is called to probe the selected driver and assign the driver. If the probe succeeds, then the support commands for that particular card and device class are executed.

Card removal is a little simpler, as the configuration entries do not need to be searched or matched.

4. Results.

The design as described has been implemented on FreeBSD 1.1.5.1 and FreeBSD 2.0 on a range of Laptops. Whilst the overall objective has been successful, in some cases it has highlighted what needs to be developed to get the most out of the PC-Card technology (see later). Network cards and serial cards are the predominant use of the system so far, but conceivably any PC-Card can be utilised to a greater or lesser extent. The work to date has been submitted to the FreeBSD core team, and should appear in FreeBSD 2.1.

One impressive demonstration is to start pinging another host, then remove the network card (causing ping to return 'no route to host' errors) and insert the card again, allowing ping to continue as if nothing had happened.

Some problems have arisen, some of which are solvable by more development on FreeBSD, and some which are related either to limitations within the PC-Card technology or standards, or vendor's implementations.

In particular, the following conclusions were reached about the PC-Card technology:

- Incorrect or incomplete CIS implementations are a major pain in the neck. Under DOS/Windows these are often fixed by the vendor supplying a proprietary *ENABLER* program designed to set up the card so that it can be successfully used. It is unlikely that the vendor would supply an equivalent FreeBSD program.
- Whilst the memory and I/O address mapping is independant of the ISA bus, the interrupt line uses the PC interrupt controller, and so the device configuration must take this into account when allocating/probing devices.
- The PC-Card standard is not. Specifically, the CIS Metaformat does not provide tuple configuration standards for various I/O cards such as network controllers.

5. Future Work.

Most of the future work centers around the FreeBSD kernel (what could be more fun?) and is aimed at adding the driver and kernel interfaces to allow detaching and reattaching of drivers/devices, even network drivers. It is surprising just how many pieces of code this will affect throughout the system.

Other changes that are planned are:

- Re-organisation of the slot driver to split it into a slot controller specific portion and device independant portion; this is similar to the structure under DOS/Windows which splits Card Services from Socket Services (though the interface will be much narrower). This would also allow different types of slot controllers to be managed within the one system.

- Generation of a separate bus structure to split the PC-Card drivers from the current ISA bus drivers; this will allow better integration of PC-Card specific functions such as power management and hot swapping, and also provide an interface to the kernel separate to the ISA routines.
- Incorporation of memory cards, either by a user level NFS process using the read/write interface to the card, or by a *vnode* interface module.
- Integration of mobile communication support for such things as roaming and IP encapsulation.

6. Conclusion.

The PC-Card technology is a fast growing and vital part of the Laptop and mobile computing field. Integrating this technology with FreeBSD has been quite easy (well, not as hard as it could have been), but further development that impacts considerably more on the FreeBSD kernel will be necessary to obtain the most benefit from PC-Cards. This work is progressing, and will be released as part of the FreeBSD project.

The latest release of this code is available for ftp from **dmssyd.syd.dms.csiro.au** under **/pri/mcrae/pcmcia-2.0.tgz**.

7. Further Reading.

The PCMCIA Developers Guide, Michael T. Mori, Sycard Technology, ISBN 0-9640342-0-4

PC Card Standard - Release 2.01, Personal Computer Memory Card International Association, Sunnyvale CA.