Project Number: CS-MLC-SL99

A Symbol Spectrum24 Wireless LAN Device Driver for Linux

A Major Qualifying Project Report

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Lee J. Keyser-Allen

May 7, 2000

Approved:

_____

Mark L. Claypool, Major Advisor

Abstract

With Linux becoming a viable alternative to commercial operating systems and the recent explosion of wireless technologies, Symbol Technologies is seeking a driver that will allow them to tap into the Linux market. This project created a Linux device driver for the PCMCIA version of Symbol's Spectrum24 series wireless network card. This driver was implemented and tested with Linux kernel 2.2.x and released under GPL. Its performance is comparable to the Windows version of the driver for the same card.

Acknowledgments

I wish to thank my sponsor, Symbol technologies, for their generous donation of hardware and documents that made this project possible. I'd also like to thank my liaisons at Symbol, Brad Lefore and Bob Neilsen, for their hard work and assistance in setting up the sponsorship and answering many questions.

Finally, I wish to thank my advisor, Dr. Claypool, for his support and assistance from across the country throughout the creation of the driver, as well as his invaluable assistance in the writing of this document.

Table of Contents

1. Introduction

Linux and the open source movement have recently gained popularity as people are looking for a low priced high performance solution for many computing applications. Linux is a low cost alternative to Microsoft Windows or a commercial *NIX[1]. Linux is an operating system (OS) that was developed by Linus Torvalds while he was a graduate student. It was subsequently released to the public under a form of the GNU Public License (GPL). In keeping with the ideals behind the open source movement and the GPL, people who saw its value voluntarily worked with Linus to remove its bugs and turn it into a viable OS. It has since benefited greatly from the open source movement and has become the free operating system of choice for millions.

One of the major advantages of Linux is the rapid development of new drivers. Since the source code is available to anyone who wants it, it is relatively easy to extend it to perform some new task. As all these changes get incorporated into the kernel, it becomes incredibly diverse.

As the demand for Linux has grown, so has the demand for applications and drivers to take advantage of the latest and greatest technology. One such budding technology, wireless networking, is just beginning to catch on in the Linux community. This is evidenced by the wireless extensions added to the 2.2.x series of Linux kernels. The

[1] *NIX refers to UNIX and its many clones of which Linux is one.

growing number of requests for Linux drivers from their clientele prompted Symbol to sponsor this project.

Wireless networking is the future. As far as speed is concerned, there are many better alternatives to wireless links, but for general usage, wireless is more than fast enough. Low-end wireless cards fall in the 1-2 Mbps range and the faster cards range from 11-45 Mbps, faster than most wired networks[2].

With the recent explosion of wireless technology, coupled with the popularity of the Internet, the next logical step is a more comprehensive wireless digital network, which would allow mobile Internet access. In the next few years, there will be networks in place similar to cell phone networks that will allow widespread wireless connectivity.

The current market for wireless networking is large companies with many laptop users (enough that it becomes feasible to install a wireless network.) One of the advantages of wireless is that after initial setup, it is trivial to add another MU (Mobile Unit.) The ability of one MU to operate at many different sites without a change in software or configuration is another boon for employees who frequently travel between offices. Employees can walk out of their home offices, where their laptops are configured, travel

[2] The standard for wired networks at the time of writing is 10 Mbps. Newer wired networks are rapidly embracing a 100 Mbps standard and some are even running gigabit Ethernet, although the latter is usually run over fiber-optic cable as opposed to standard copper.

to remote partner offices with similar networks, walk in, and instantly have network access[3].

Symbol Technologies started out marketing barcode scanners that were interconnected using a wireless network. They then branched out to incorporate the wireless networking market, and have just recently added wireless voice to their product lines. The wireless phones that they market attach to an existing local wireless data network and use voice over IP technology to connect to the PBX. Their line of solutions now ranges from mobile computing devices to barcode readers to network cards, all of which can be connected using wireless networking technology.

Symbol sponsored this project after pressure from their clients to come out with Linux drivers. This is their first driver for Linux (although not the first Unix driver, as there is an SCO Unix driver out for the same card.)

Thus, the main goal of this project was to create a functional driver for Symbol's Spectrum24 wireless LAN card. The first portion of the project took a week and was oriented towards research of the tools that were going to be utilized. It also included some background research of wireless networking under Linux and the PCMCIA package. The second portion was the implementation part. It involved writing, testing, and debugging the driver over the following six weeks. Once the driver was functional and stable, there

[3] There are, of course, security issues with this model, but with relatively simple pre-configuration, and knowledge of the RSA keys for each office, 40-bit encryption could be added with no perceived overhead.

was complete rewrite to clean up the code and insert useful debugging statements incase of future need. Symbol is contracting to add the rest of the functionality mentioned in the Future Work section.

The rest of this report is laid out as follows: Section 2 contains an overview of the structure and ideology of the Linux kernel. Section 3 describes the PCMCIA package and the functionality that it offers. Section 4 covers the wireless standard and gives an overview of wireless networking. Section 5 contains the implementation specifics, and describes the code organization. Section 6 gives some performance data and compares the final driver to the Windows version written by Symbol. Section 7 contains the conclusion, and Section 8 describes the future work. The source code is attached as an appendix. It is also included on the CD that accompanies this report along with much of the support it needs for compilation.

2. The Linux Kernel

The Linux kernel started as a monolithic kernel. A monolithic kernel has all of the functionality from memory management to floppy controllers compiled into a single executable. The kernel is loaded into memory at boot-up and stays there until the machine is powered down. Through its evolution, Linux has progressed so that it has become a combination of kernel designs. The other kernel structure that Linux has begun to incorporate is the micro-kernel design. In a micro-kernel, much of the non-core functionality is pushed out of the main kernel and into separate fragments. These

fragments traditionally run in user space as daemons, however Linux implements this by creating kernel modules.

Linux kernel modules run in kernel space once they are loaded and have access to all of the kernel internals and kernel functionality. In essence, they become part of the kernel for as long as they are loaded. This tight coupling allows very fine-grained control over timing, which is important in many hardware applications, and guarantees that routines do not get pre-empted when they have time sensitive tasks. There is a tradeoff, however. The sharing of kernel space eliminates fault isolation, one of the fundamental aspects of a true micro-kernel design. This puts more responsibility on the module author, because mistakes can cripple or hang a system.

Modules can be loaded at any time after boot and can be unloaded at anytime, as long as they are not in use. By convention, modules are loaded upon demand and unloaded after they have been idle for a set amount of time. Modules are loaded either by the kernel itself, when it needs the functionality they offer, or by a privileged user or daemon process. Modules are loaded by a form of the **insmod** command and parameters may be passed to them at load time. They are unloaded by the **rmmod** command.

3. The PCMCIA package

One of the other tools used in this project is the Linux PCMCIA package. This package has drivers for many different PCMCIA controllers as well as card resource management services. The package is divided into two main parts. The kernel modules are the drivers

for the PCMCIA controllers and a special card services module that handles resource allocation. The user level daemon is in charge of loading modules and providing a user level interface to the controller cards. The package also includes some utilities to send the cards PCMCIA events to change the cards' state.

As mentioned previously, kernel modules can be loaded by a number of methods. The PCMCIA package has provisions to load the appropriate module depending on the card that is inserted. Since each PCMCIA card has an identification string (called a ' maintains a lookup table of identification tuples matched to modules. It is the job of its user level daemon to search the table and load the appropriate module.

The PCMCIA package also handles the allocation of I/O ports, interrupts, and memory windows for the client modules through the card services module.

4. A Wireless Overview

The current wireless standard is IEEE 802.11. This standard is very broad in that it encompasses not only different frequencies and bit rates, but also different signal coding techniques.

Section 4.1 focuses on signal coding techniques, while section 4.2 covers wireless networks.

4.1 Spread Spectrum

All wireless devices covered under the IEEE standard use unlicensed bands. While there are no licenses, the FCC has strict requirements on how much power a device can radiate, and how the band is to be shared. One of these requirements is that a device on these bands must use a **spread spectrum** technique to minimize its impact on the band. What this means is that the devices must not hog the band or overpower other devices sharing the band. These techniques are also beneficial to the devices using them, because they are designed to allow communication of noisy bands. The two main ways that this is accomplished are **direct sequencing** and **frequency hopping**.

Direct sequencing involves blasting the information out over a very wide section of the band, but at a relatively low power. This minimizes localized disturbances, but with a receiver tuned to listen to the proper coding, the signal can be understood at the other end.

The second technique is frequency hopping. Frequency hopping eliminates local disturbances by hopping around the band. It sends out a higher powered signal over a very narrow section of the band and then hops to another frequency to send out its next packet. A system joins this network by listening to a set frequency for the transmitter to hop by, and when it does, it synchronizes itself to the master and hops with it. This technique is the one that the Symbol card uses.

4.2 Wireless Networks

A wireless network, also known as wireless LAN, or WLAN, is a collection of wireless units which all share the same bandwidth. Every wireless network requires some sort of identifier. This allows more than one wireless network to share the same airspace. The 802.11 standard calls that identifier the ESSID. The ESSID is a thirty-two character alphanumeric string that uniquely identifies a network.

The standard wireless network is set up with some number of access points (APs) and many slave mobile units that associate with whichever AP they are closest to at any point (determined by a voting system based on signal strength.) Usually the APs serve as bridges to connect the wireless network to a standard wired network.

The 802.11 standard also describes something that they call ad-hoc networking, where multiple equal units create their own wireless network. In Symbol's scheme, however, there is always a master unit. Any of their cards can perform as the AP, but one has to be explicitly put into Micro AP(MAP) mode. The rest of the units are left in MU mode and perform as though the MAP was a standard (non-bridging) access point. As such, it doesn't strictly conform to the standard.

5. Implementation

It was decided that the driver should be created as a kernel module for simplicity and so that it could take advantage of the PCMCIA card services offered by the PCMCIA

package. The PCMCIA package was chosen to avoid having to re-invent the PCMCIA controller interface and to abstract some of the resource management.

Much of the initial research for this project involved searching through the PCMCIA programmers manual as well as sample code that was available from the Linux PCMCIA package's Web site.

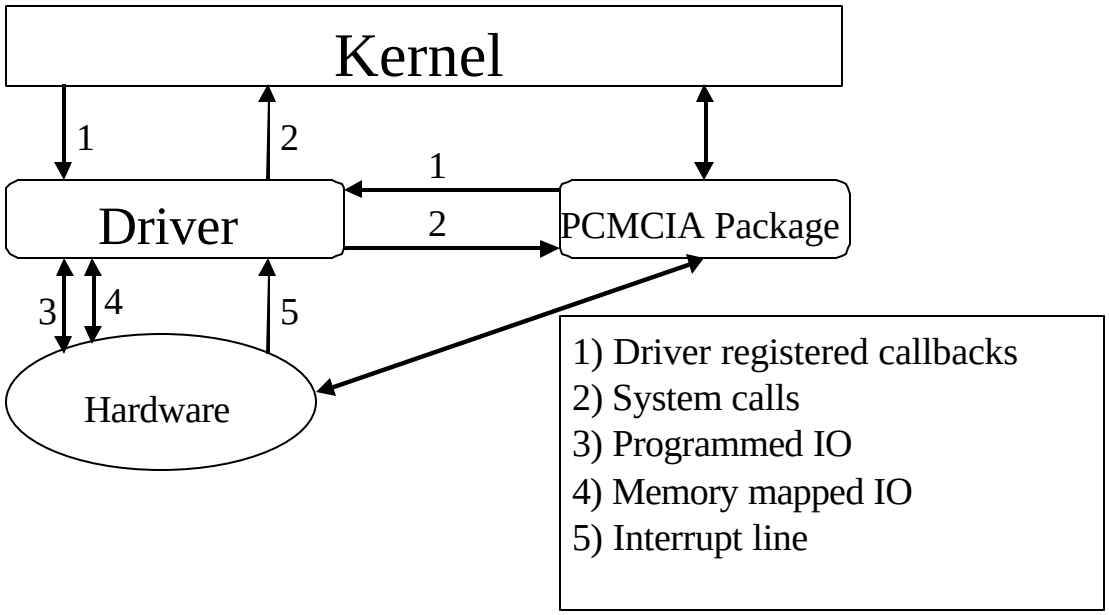

Fig. 5.1 An overview of driver/kernel communication

Figure 5.1 shows a simplified view of the communication channels between the driver, the kernel, the PCMCIA package, and the hardware. The driver registers callbacks (1) with the kernel so that the kernel can tell it to bring the interface up or down, and so that the kernel can inform it when it has a packet to send. The driver registers callbacks (1) with the PCMCIA package so that the package can inform the driver when a card is inserted, or removed, and can keep it informed of different PCMCIA events. The driver sends messages to the kernel and PCMCIA package via system calls (2). These system

calls are functions that have been exported so that any code in kernel space has access to them. The driver talks to the hardware through either a combination of memory mapped I/O (4) and programmed I/O (3), or strictly programmed I/O. The hardware talks to the driver via an interrupt line (5) and the chosen I/O mode(s).

One of the first programming challenges presented was how to best access the card, since the card has two access modes (memory mapped I/O or programmed I/O.) Initially memory mapped mode was chosen for its slightly better performance characteristics, but it soon became clear that some older (ISA based) PCMCIA controllers had problems with creating a large contiguous memory area. The final solution was to add in a programmed I/O mode, and then use this mode as a fallback if we are unable to create the memory window required.

An advantage of working with a card that is as advanced as the Spectrum24, is that much of the underlying network complexity is already handled on the card itself. The driver doesn't have to worry about the MAC layer at all, since the firmware already handles MAC level retransmission. There are even hooks for higher level IP constructs, such as multicast.

The majority of the programming work entailed getting the interface to the card, via the PCMCIA controller, up and running, and registering with the appropriate kernel subsystems. Once that was done, the actual communication from the driver to the card was comparatively easy.

One of the main goals of this driver was to make it as modular as possible, while still maintaining functionality and conforming to the current PCMCIA programming paradigm. One of the ways this is done is by minimizing global variables and putting all the card specific data into data structures that are linked to the PCMCIA slot itself, and passed around by the callback functions. These data types, such as the **dev** structure and **local_info_t**, maintain all of the data that needs to be accessible to the functions that talk directly to the card (namely the I/O ports and the memory locations if it's memory mapped.)

The reason for modularity and data hiding is that this driver can be used for more than one card. As such, it needs to be reentrant. This was accomplished through semaphores, disabling interrupts, and spin-locks, as well as data hiding.

Section 5.1 gives a basic overview of the code organization, while section 5.2 covers the same material in greater depth. Section 5.3 shows how the driver interacts with the kernel and the PCMCIA structure on load, unload, and packet transmission.

5.1 Code Overview

This section contains a brief functional overview of each of the procedures in the driver module. It is broken up into data types and functions, and then further divided based upon their functional sub-group.

5.1.1 Global variables

Here is an overview of the global variables and each of their purposes.

5.1.1.1 PCMCIA Globals

The first few global variables are PCMCIA specific. They are defined inside the driver, and are global within the driver, regardless of the number of cards it happens to be supporting:

dev_info is just a static string that is used to identify the driver to the PCMCIA package.

dev_list is a linked list of the device instances that are currently being supported by this driver.

5.1.1.2 Configuration Globals

These are the configuration parameter globals. The purpose of these is to allow some of the parameters to be entered as command line parameters.

ESSID[*] is used to store the initial ESSID. The ESSID can be later changed using the wireless extensions ioctl call.

def_ESSID is used so that a default ESSID can be statically set at compile time. This is used in the case that one is not provided, or the one provided is invalid for some reason.

map_mode[*] is used to determine if the card is in Micro AP mode.

psp_mode[*] is to set the power saving modes for the card. It is currently not implemented.

map_dtim[*] , **map_hop_set**[*] , and **map_hop_pattern**[*] are for setting the Micro AP parameters. **Dtim** determines how often the AP sends out its beacon packet. A beacon packet has all the information that a new MU needs to join the network and start hopping with the AP. **Hop set** is the set of frequencies that the AP hops through, and **hop pattern** determines the order in which it hops.

---

[*] These parameters are settable on the **insmod** command line.

[*] These parameters are settable on the **insmod** command line.

5.1.2 Functions

Here is an overview of what each function does, divided into functional subgroups.

5.1.2.1 The basics

init_module and **cleanup_module** are standard module functions. They are called when the module is loaded or unloaded, and contain module initialization. The initialization consists of some basic configuration as well as registering and unregistering the PCMCIA callbacks.

5.1.2.2 PCMCIA

Here are the functions that are more or less for PCMCIA specific initialization. Some of these functions incidentally set up the Ethernet structures, but their primary purpose is resource management using the PCMCIA package.

s24_attach initializes a lot of the data structures necessary for the PCMCIA package. It also sets up the Ethernet callback structure to tell the kernel that it just gained a network card. It doesn't actually register itself with the kernel, but it puts all of the addresses of its callbacks into the proper data structure.

s24_detach undoes everything that **s24_attach** does. That is, it frees memory associated with those structures, and removes the device from the linked list of active devices.

s24_config sets up the card resources using the PCMCIA package. This is where it creates the memory windows it uses, and determines if it will use I/O mode or memory mapped mode. It is here that it registers the Ethernet callbacks that we set up earlier in **s24_attach**.

s24_release undoes everything that **s24_config** does (this includes unregistering it from the kernel's list of network cards and freeing memory windows and I/O ports and interrupts.)

s24_event handles PCMCIA events such as insertion, removal, and power management.

5.1.2.3 Wireless Extensions

These are the functions that are for the wireless extensions, or heavily based on them. The wireless extensions kernel option creates a set of ioctl calls and a proc file system entry to monitor and configure running wireless cards.

iw_statistics is used to fill in the proc file system entry that wireless extensions creates.

s24_ioctl is the ioctl handler. It currently supports many of the wireless extensions.

5.1.2.4 Ethernet Functions

These functions are, for the most part, the Ethernet callbacks that get registered with the kernel.

s24_eth_config is called by the kernel to change I/O ports or interrupts. It is currently unimplemented.

s24_init is for the Ethernet initialization. It isn't used because all of the initialization is done earlier (in the PCMCIA section.)

s24_tx is a callback in charge of the actual packet transmission. It is registered in **s24_attach**, and gets called by the kernel when there is a packet to deliver. It unwraps the packet and writes it to the card and then tells the card to send it.

s24_open is used to bring the card up and start radio activity (it actually calls **s24_reset** to do the hardware part) but it sets the flags so that the kernel knows that it's up and going.

s24_interrupt is the interrupt service routine. It is called when the card throws an interrupt. It handles newly received packets, packets that have been successfully sent, and status change events. It also has a provision to deal with the (now obsolete) RX_ERROR condition.

s24_multicast is still not implemented, but will deal with the multicast aspects of the card.

s24_stats is in charge of the statistics reporting. (errors, bytes sent, etc.)

s24_stop is the counterpart to **s24_open**. It sets the flags that say the card is not running anymore, and it calls the **cold_reset** routine to turn the card off.

5.1.2.5 The Hardware

These functions are the ones responsible for actually talking to the card and getting it ready to talk. Note: **s24_tx** and **s24_interrupt** also do a lot of communicating with the card, but they are more Ethernet based, and, as such, are covered in the previous section.

cold_reset is used to get the card up and going, or issue a full reset. This will take the card from any state and set it back to a known state. That known state is: alive and in the right mode (MU or MAP,) but not talking.

s24_reset sets the receiver characteristics (including multicast, etc.) and the transmitter characteristics, the MAP parameters (if necessary), PSP, and the ESSID (which starts radio operation.)

5.2 Code Details

This section gives a far more detailed rendition of the information above.

5.2.1 The basic functions in depth

These are the functions that are associated with any loadable module.

5.2.1.1 init_module

init_module is the first function called when the module is loaded. It checks that the ESSID is valid, and it registers the driver, along with the **s24_attach** and **s24_detach** functions, with the PCMCIA package.

5.2.1.2 cleanup_module

cleanup_module is the last function called when the module is unloaded. It unregisters the driver with the PCMCIA package, and then traverses the linked list to make sure that all of the cards have been removed, and all of their resources are released before it exits.

5.2.2 The PCMCIA functions in depth

5.2.2.1 s24_attach

s24_attach is the main PCMCIA function. It sets up the **link** structure, which will be used by the card for as long as it's in the computer. The **link** structure stores all the

PCMCIA specific information about the card, and what resources it needs before they are allocated. The **link** structure also contains an auxiliary field, which points to the **dev** structure that houses the Ethernet specific data.

s24_attach starts by allocating the memory for the structure, and then populating it with information as to how the driver wants to handle the card (whether it want to assign an interrupt, how many I/O ports it needs, and whether it wants to try to use memory and I/O mode, or if it's just a memory card.)

Once it has populated that structure, it goes on to allocate and populate the **dev** structure, as well as the structure called **local_info_t** which is used to store private card specific configuration data (such as its ESSID.)

It then registers the driver with PCMCIA card services.

5.2.2.2 s24_detach

s24_detach is used to free all of the resources allocated above in **s24_attach**. It finds the **link** structure, determines what has been freed, and what still needs to be freed, and it uses the **link->state** flag to indicate what state it's in. It then makes sure that **s24_release** has been called before freeing all the other data structures and unregistering from card services.

5.2.2.3 s24_config

s24_config is there to do most of the actual interfacing with card services. It deals with the tuple data, sets up the **link->state** flag, and creates the memory window request. It then tries to allocate the address space for the window. If it fails, it falls back to I/O mode. Next, it requests I/O ports, and an interrupt.

Finally, it sets the flags for the **dev** structure, sends the card a **cold_reset**, registers the Ethernet callbacks, and tells the kernel the card's Ethernet address.

5.2.2.4 s24_release

This undoes everything that **s24_config** does. It verifies that the card is ready to be released, and that the link has been closed. It then releases the I/O ports and the IRQ and the Memory window.

It also unregisters the network device and sets the **link->state** flag according to its current state.

5.2.2.5 s24_event

This is the PCMCIA event handler. The PCMCIA events include the insert, eject, reset, power down, and power up events. It uses the event to trigger the other functions to carry out the request and then returns the exit status (successful or unsuccessful) of the functions it invoked to the PCMCIA package.

5.2.3 Wireless Extensions

These are the wireless extension based functions.

5.2.3.1 iw_statistics

This gets called when there is a request to view the wireless statistics in the proc file system. It fills in the **iw_statistics** structure to report things like the current frequency, the signal strength, etc. It is currently unimplemented

5.2.3.2 s24_ioctl

This is the ioctl handler, it gets called when a program runs that has an ioctl call to this device in it. The purpose of ioctl is to manage various parts of the driver on the fly. Using this, settings such as the ESSID can be modified, and, in the future, MAP mode, map_hop_set, and map_hop_pattern will be able to be dynamically modified as well.

5.2.4 Ethernet Specific Functions

These functions are callbacks that get registered with the Linux kernel networking layer.

5.2.4.1 s24_init

s24_init is placed there for Ethernet drivers in case there is initialization that needs to be done. This is probably the place to do a lot of ISA configuration, but it isn't needed in the PCMCIA version.

5.2.4.2 s24_eth_config

s24_eth_config could be used to modify the I/O ports the card is using, or change the interrupt. It is currently unused.

5.2.4.3 s24_tx

This is, as the name suggests, the function actually in charge of the packet transmission. The kernel calls it when there is a packet to be sent out. The driver has to copy the packet from kernel space to the card, and then notify the card that there is a packet to be sent.

5.2.4.4 s24_open

s24_open is registered with the kernel, and the kernel calls it to bring the interface up. It marks the link as open in the **link** structure, and increments the module use counter (the module use counter is used by the kernel to tell if it is safe to remove a module, or if it is in-use.) It sets up the **dev** structure to indicate to the kernel that it is up and running, and it calls **s24_reset** to bring up the hardware.

5.2.4.5 s24_stop

s24_stop is the counterpart to **s24_open**. It modifies both the **link** structure and the **dev** structure flags to say the card is no longer up, and it calls the **cold_reset** routine to turn the card off. It also decrements the module use counter.

5.2.4.6 s24_interrupt

This is the interrupt service routine. It handles the various interrupts that the card can generate. The card signals when it's done sending a packet, when it has a packet to deliver, and when it moves to a different wireless network. Currently, the only conditions that are handled in a useful way are the **TX_DONE** and **RX_DONE** conditions. **TX_DONE** allows the driver to reset the transmit locks and notify the kernel that the last packet was successfully sent. **RX_DONE** tells the driver that it needs to go and get the new packet from the card and hand it off to the kernel for processing.

5.2.4.7 s24_multicast

s24_multicast will move the card in and out of multicast mode and maintain the multicast lists.

5.2.4.8 s24_stats

This is the statistics reporting function. It keeps track of how many packets were sent, and received, and how many errors there were. These statistics get displayed by utilities such as ifconfig.

5.2.5 Card Specific Functions

These are most of the low-level card interface functions. Some of the ones that are notably absent are **s24_tx** and **s24_interrupt** which are documented above.

5.2.5.1 cold_reset

cold_reset is, as the name implies, a way to give the card the hardest reset you can, short of power cycling. It takes advantage of the host command interface to set the **cold_restart** flag, and then it holds down the reset line for a couple milliseconds before letting the card come back up. It is responsible for setting the adapter mode (MAP or MU) since that needs to happen early, and can only be done after a cold restart. It then issues the power-up command and waits for it to wake up. If it doesn't hear from the card in 600ms (about twice as long as the card should take to come up) it returns failure.

5.2.5.2 s24_reset

This sends the card a reset command which does a soft reset (this is different from using its reset line as is done in **cold_reset**.) It configures the receiver and the transmitter and sets up the MAP characteristics if it is going to be run in MAP mode (these include **hop set**, **hop pattern** and **dtim**.) Next it configures PSP (the power saving features), and then it sets the ESSID which tells the card to start talking.

5.3 Runtime Overview

This is a description of the steps involved in loading, unloading, and running the driver.

5.3.1 Loading

i) The PCMCIA card services detect the card that is inserted, and compare its identification tuple with its database to find the correct driver.

ii) The PCMCIA package then loads the driver, at which point the driver registers itself with PCMCIA card services.

iii) The PCMCIA package calls the **s24_attach** function for the card it's dealing with, and the **s24_attach** function sets up the data structures and then registers itself (again) with the card services module.

iv) The card services module determines that this is the correct module for the job, and uses the information that was given to call the **s24_config** function to set up the card and get it on its feet.

v) **s24_config** gets all the necessary resources and tells the kernel that it just gained a network card (in the process, it registers all the necessary callback functions.)

vi) Everything just sits there and waits for the kernel to decide that it wants to use the network card. It is waiting to be assigned an IP address and to be brought up by the kernel. In the case of PCMCIA cards, this is automated by an external script that PCMCIA services runs.

vii) Once the card is brought up, the kernel calls **s24_open** which in turn calls **s24_reset** to get the card talking.

5.3.2 Packet Handling

This is how the driver handles incoming and outgoing Ethernet packets.

5.3.2.1 Receiving Packets

i) The card receives the packet and triggers an interrupt to alert the driver.

ii) The driver's interrupt service routine (ISR) reads the packet off the card, and allocates memory for the packet structure that the kernel requires.

iii) The ISR puts the packet in the packet structure and notifies the kernel of the packet's arrival.

iv) The ISR tells the cards that it received the packet, and checks for any more.

v) If there are more packets, it goes back to ii and starts again, if not, then it exits.

5.3.2.2 Sending Packets

i) The kernel passes the packet to the driver through the **s24_tx** callback.

ii) The driver waits until the card is clear to send.

iii) The driver extracts the packet from the packet structure and writes it to the card.

iv) The driver signals to the card that it has a packet to send.

v) The card sends an interrupt to acknowledge that it either sent the packet or timed out trying.

vi) The ISR passes that information back up to the kernel, and clears its transmit lock.

5.3.3 Unloading

i) When the card is ejected, it is brought down by a PCMCIA script (which calls **s24_stop**.)

ii) Card services unloads the driver, at which point, the driver calls **s24_release** and **s24_detach** to free the resources.

6. Performance Analysis

The driver was tested against the Windows driver to give a picture of its relative performance. Both transfer modes of the Linux driver were tested (I/O mode and memory mapped mode) to get an idea of how the perceived performance advantage of memory mapped access held up under testing. It was tested on a network with a single AP and a single MU (the one on the machine being tested) to minimize interference. The tests involved communication with a computer directly connected to the same network segment as the AP. At the time of the tests the network traffic on this segment was very low to non-existent. The card was set into continuous access mode (the power saving mode that has the best performance) on both machines.

The windows system was a PII 266MHz laptop running windows 98. It had 64 MB of memory, a 3 GB hard drive, and it was running the Spectrum24 2Mb/s network adapter. The Linux system was a Pentium 200MHz laptop running the SuSE distribution of Linux with Linux kernel 2.2.10. It had 32 MB of memory, a 2 GB hard drive, and it was using the Spectrum24 2Mb/s network adapter.

6.1 Ping Test Results

The first test performed was a simple ping test. 15 ping packets of 64 bytes each were sent from the test machine to the host. The statistics were gathered from the local ping program. The test was run multiple times, to insure the consistency of the results, however only one set of data is included here.

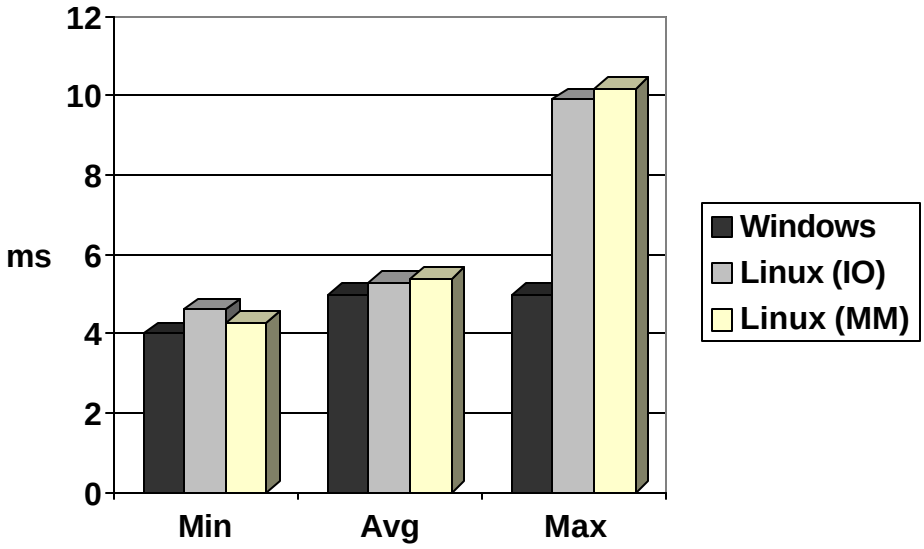

Fig. 6.1 Ping Test Comparison Graph

The ping results were relatively consistent. The most important statistic in this case is the average time. Most of the discrepancy lies in the ping program itself. The Linux program had a much more precise time counter and gave all its statistics in tenths of milliseconds as opposed to the whole millisecond granularity of the windows ping client. The outstanding maximum ping time happened once in each incarnation of the Linux tests, however this is probably related more computer location and spurious RF interference than to the driver efficiency.

The conclusion that can be gathered from this data is that the Linux driver is very close, if not equal to the Windows version for response time. There is an insignificant amount of variance in response times between the two Linux I/O implementations.

6.2 File Transfer Results

The second test was a file transfer test. The local ftp program was used, and again, its output was used when recording statistics. The file transfer test involved a file of 100 kilobytes (102400 bytes) in length. The file was entirely filled with 0s. The statistics are based on ten transfers, five uploads from the test machine to the host, and five downloads from the host to the test machine.

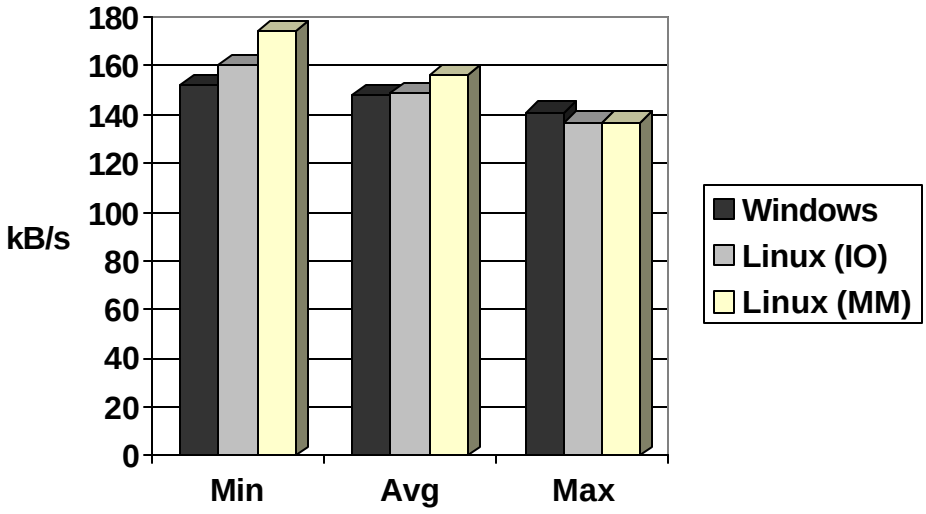

Fig. 6.2 File Transfer Comparison Graph

In this test the Linux driver outperforms the Windows driver in the average case. This difference is small enough that it could be attributed to differences in the TCP/IP stack or the FTP program used. The interesting difference here is the slight improvement that memory mapped mode has over programmed I/O. Since these statistics were gathered using exactly the same TCP stack and the same FTP program, the only difference would be the quality of the wireless link at the time and the difference due to the access method. Due to the size of the file being transmitted and the fact that the two tests were run under nearly identical conditions, wireless interference can be ruled out almost entirely. The difference is large enough and consistent enough to be statistically significant without being overwhelming. It was postulated that access mode would make a larger difference, and it appears that the performance hit that is taken by relying strictly on programmed I/O, although not trivial, is not substantial.

7. Conclusion

An atmosphere of experimentation surrounds the Linux project, and has since its inception. This is due in part to the fact that many of the early Linux users were programmers looking for something to play around with. This is one of the reasons that its hardware support has moved forward so rapidly. This project was the perfect opportunity to contribute to the advancement of a quality open source product.
This project contributed a wireless driver for Symbol's Spectrum24 card to Linux.

The wireless aspects of the project provided a unique challenge. The versatility that wireless affords and the lack of hard connections make it that much harder to trace and debug. Ultimately the most effective debugging system involved a separate computer whose sole purpose was to sniff wireless packets. This computer would hop with whichever access point ESSID was provided and would log the entire contents of the packets that it saw. Searching through those logs was a good way to become very familiar with the structure of wireless packets, not to mention ARP packets, and a number of other Ethernet constructs.

This project developed into a functional driver throughout the seven-week term. Much of the first weeks were spent researching and developing a plan of attack. Then a complete PCMCIA interface shell had to be created so that resources could be allocated to communicate with the card in the first place. Next, the card interface had to be organized and the card finally got a chance to show off its blinking lights. By then the code had

become a major disaster from an organizational standpoint, so it warranted a complete rewrite to clean it up. I took this opportunity to add modularity as well as a programmed I/O interface.

The resulting code was released to the general public under GPL and is included on the CD accompanying this report, along with the PCMCIA package it was developed with. One of the great elements of working with an open source product is the amount of constructive feedback that users are able to give. In the short time that this driver has been available there have already been a number of e-mails of appreciation and suggestions (most of the suggestions are outlined in Future Work.)

So, in conclusion, Symbol was pleased with the final result, given the timeframe, and has since contracted with me to complete the remainder of the driver as outlined in Future Work. My advisor was pleasantly surprised to see the amount of progress and functionality that the project had attained by the conclusion of the term. Lastly, I am excited to have created an open source driver that is being used successfully in the world, and I'm pleased that it performs as well as it does.

8. Future Work

The current driver is written for the 2.2.x kernel. It would be good to port it back to the 2.0.x kernels as well. This would involve some manipulation of the way that memory is allocated as well as a few changes to the format of some system calls.

Much of the configuration data that is currently set as load-time parameters should be moved to dynamically modifiable parameters. This requires more ioctl handling and a user mode utility to send the appropriate ioctl call.

PSP support needs to be implemented. This entails changing **s24_reset** to incorporate the load-time parameter. It could also be set up as a dynamically modifiable parameter.

Statistics reporting needs to be incorporated as well. This would involve investigating the card structures to find the memory where the card stores these statistics and reading them out and resetting them each time there is a request made.

Multicast needs to be implemented. It involves the addition of a number of data structures, and a complex interface to the cards own multicast features to make the most efficient implementation.

The final item that should be added is an interface to the card's security features. This is another complicated process, as it involves more card communication and configuration. It would most likely be placed in **s24_reset** and should be dynamically modifiable as well.

References

D. Hinds. “Linux PCMCIA Programmer’s Guide.”

http://pcmcia.sourceforge.org/ftp/doc/PCMCIA-PROG.html

A. Rubini. Linux Device Drivers. O’Reilly and Associates: Sebastopol, CA, Feb. 1998.

J. Tourrilhes. “The Linux Wireless LAN

http://www.hpl.hp.com/personal/Jean_Tourrilhes/Linux

A. Werback. “Spectrum 24 802.11 Host Adapter Interface Specification.” Specification Version E, Symbol Technologies Inc.: San Jose, CA, Feb 1999.

Appendix

spectrum24.h

```
/* (c) 1999 Symbol Technologies, all rights reserved. */
/* May be distributed under the terms of GNU's General Public License,
the
   text of which can be found online at
http://www.gnu.org/copyleft/gpl.html */

#ifndef __SPECTRUM24_H__
#define __SPECTRUM24_H__


/* The values to set the COR to in order to toggle IO mode. */
#define ENABLE_IO_MODE  0x41
#define DISABLE_IO_MODE 0x00

/* make bitmapped bytes a little easier to read (although I think if
you've
   got far enough to read this you know which bits are on where) */

#define BIT_0     0x01
#define BIT_1     0x02
#define BIT_2     0x04
#define BIT_3     0x08
#define BIT_4     0x10
#define BIT_5     0x20
#define BIT_6     0x40
#define BIT_7     0x80

/* io port layout */
#define HSR 0x0
#define CMD 0x1
#define BSSR      0x2
#define HREG      0x3
#define PREG      0x3
#define LMA0      0x4
#define LMA1      0x5
#define MEM0      0x6
#define MEM1      0x7

/* HSR bits */
#define HSR_P188_ACTIVE BIT_5
#define HSR_HREG  BIT_2
#define HSR_P188_INT    BIT_0

/* Host command register commands */
#define HCR_RESET BIT_7
#define HCR_PWRUP BIT_5
#define GEN_P188_INT    BIT_4
#define ENABLE_P188_INT BIT_0
```

```
/* HREG commands */
#define HREG_INIT_MU    1
#define HREG_PSP_WAKEUP 3
#define HREG_RESUME     4
#define HREG_SLEEP      5
#define HREG_INIT_MAP   6

/* Common memory layout */

/* the host interface command buffer */
#define HCB 0x100

/* the adapter status registers */
#define ASR 0x110
#define RSR 0x120
#define TSR 0x130

/* the Write Operation Complete byte */
#define WOC 0x112

/* the transmit data registers */
#define TX_NEXT_BUFFER  0x140
#define TX_BUFF_SIZE    0x142
#define TX_BUFF_OFF     0x144
#define TX_BUFF_TOT     0x146

/* the receive data registers */
#define RX_PKT_LEN      0x150
#define RX_PKT_START    0x152

/* the location of the MAC address */
#define ETH_ADDR_LOC    0x160

/* the ordinal pointers */
#define ORD_TBL_1 0x170
#define ORD_TBL_2 0x172

/* the ap table pointers */
#define AP_TBL_LEN      0x174
#define AP_TBL_PTR      0x176
#define ROM_LOG_PTR     0x178

/* association table pointers (for MAP only) */
#define ASS_TBL_PTR     0x17A
#define ACL_PTR         0x17C

/* statistics buffer register */
#define STAT_TBL_LEN    0x180
#define STAT_TBL_PTR    0x182

/* 4 byte adapter time and 1 byte time valid flag */
#define ADAPTER_TIME    0x184
#define TIME_VALID      0x188

/* restart flag */
#define RESTART_FLAG    0x189
```

```
/* transmit disable flag */
#define TX_DISABLE      0x18B

/* flash memory info */
#define FLSH_CFG_PTR    0x18C
#define FLSH_STAT 0x18E

/* MKK stuff ... for japanese firmware only */
#define MKK_ENABLED     0x190
#define MKK_CALLSIGN    0x191

/* country info */
#define COUNTRY_ID      0x1A0
#define COUNTRY_NAME    0x1A2

/* restart control stuff */
#define WARM_START_REG  0x380
#define RESTART_COUNT   0x384
#define OP_MODE         0x386 /* one byte */
#define SELF_TEST_CURR  0x388
#define SELF_TEST_CUM   0x38C

/* the association status register */
#define ASSC_SR   0x50C

/* HCB commands */
#define CMD_RESET       0x00
#define CMD_SRC               0x01
#define CMD_STC               0x02
#define CMD_AMA               0x03
#define CMD_DMA               0x04
#define CMD_SAM               0x05
#define CMD_TXL               0x08
#define CMD_SKR               0x09
#define CMD_ASR_CLR           0x0C
#define CMD_RSR_CLR           0x0D
#define CMD_TSR_CLR           0x0E
#define CMD_SCN_CLR           0x0F
#define CMD_IMR_SET           0x10
#define CMD_TXI               0x11
#define CMD_TIM               0x12
#define CMD_IDL               0x13
#define CMD_STST        0x14
#define CMD_CLS               0x15
#define CMD_RTC               0x16
#define CMD_RTS_THRES         0x17
#define CMD_PMG               0x18
#define CMD_FLAGS       0x1A
#define CMD_ASR_SET           0x1B
#define CMD_FLASH       0x1C
#define CMD_ACL_ADD           0x1D
#define CMD_ACL_DEL           0x1E
#define CMD_ACL_CLR           0x1F
#define CMD_MAP_CONFIG        0x21
#define CMD_ESSID       0x27
#define CMD_PREFER            0x28
```

```
#define CMD_MANDAT            0x29
#define CMD_TX_RATE           0x2A
#define CMD_MAC_ADRS          0x2B
#define CMD_SET_AUTH_TYPE     0x2C
#define CMD_ENCRYPT_KEY       0x2D
#define CMD_ENCRYPT_KEY_ID    0x2E
#define CMD_INTERNATIONAL     0x2F

/* the asr bits */
#define RX_COMPLETE     BIT_7
#define RX_ERROR  BIT_6
#define TX_COMPLETE     BIT_5
#define STATUS_CHANGE   BIT_4
#define TX_BUSY         BIT_1
#define TX_FREE         BIT_0

/* the interrupt bits in the ASR */
#define ASR_INT_MASK    0xF0

/* the tx/rx buffer layout */
                        /* these are all words except */
#define LINK_PTR  0x00
#define BUFF_LEN  0x02
#define DATA_OFF  0x04
#define TIMESTMP_MSW    0x08
#define DEST_ADDR 0x0A  /* 6 bytes (duh) */
#define SRC_ADDR  0x10  /* "   "     "   */
#define TIMESTMP_LSW    0x16
#define RSSI            0x1C  /* 1 byte */
#define FREQ            0x1D  /* "   "  */

/* Ordinal table statistics pointer */
#define RX_PACKETS      33
#define TX_PACKETS      2
#define RX_BYTES  43
#define TX_BYTES  12
#define RX_ERRORS --
#define TX_ERRORS 16
//#define MULTICAST
#define COLLISIONS

/* extended recieve errors */
#define RX_LENGTH_ERRORS
#define RX_OVER_ERRORS
#define RX_CRC_ERRORS
#define RX_FRAME_ERRORS
#define RX_FIFO_ERRORS
#define RX_MISSED_ERRORS

#define TX_ABORTED_ERRORS
#define TX_CARRIER_ERRORS
#define TX_FIFO_ERRORS
#define TX_HEARTBEAT_ERRORS
#define TX_WINDOW_ERRORS

#endif
```

spectrum24_cs.h

```
/* (c) 1999 Symbol Technologies, all rights reserved. */
/* May be distributed under the terms of GNU's General Public License,
the
   text of which can be found online at
http://www.gun.org/copyleft/gpl.html */
#ifndef __SPECTRUM24_CS_H__
#define __SPECTRUM24_CS_H__

/* these includes give us a clue as to our build environment */
#include <pcmcia/config.h>
#include <pcmcia/k_compat.h>

#define DEBUG

#if defined(PCMCIA_DEBUG) && !defined(DEBUG)
#define DEBUG
#endif

/* I'm not sure whether or not the make call from the pcmcia package
will take
   care of this, so better safe than sorry */

#ifdef CONFIG_SMP
#ifndef __SMP__
#define __SMP__
#endif
#include <linux/smp.h>
#endif

#ifdef CONFIG_MODVERSIONS
#ifndef MODVERSIONS
#define MODVERSIONS
#endif
#include <linux/modversions.h>
#endif

#ifndef MODULE
#error "This code is intended to be used as a module."
#endif

/* Linux Headers that are useful */
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/ptrace.h>
#include <linux/malloc.h>
#include <linux/string.h>
#include <linux/timer.h>
#include <linux/interrupt.h>
#include <linux/in.h>
#include <linux/delay.h>
#include <asm/io.h>
#include <asm/system.h>
```

```
#include <asm/bitops.h>

#include <linux/netdevice.h>
#include <linux/etherdevice.h>
#include <linux/skbuff.h>
#include <linux/if_arp.h>
#include <linux/ioport.h>
#include <linux/fcntl.h>

#include <linux/wireless.h>

/* pcmcia headers */
#include <pcmcia/cs_types.h>
#include <pcmcia/cs.h>
#include <pcmcia/cistpl.h>
#include <pcmcia/cisreg.h>
#include <pcmcia/ds.h>
#include <pcmcia/version.h>

/* spectrum24 interface stuff */
#include "spectrum24.h"

/* Set up all the normal useful macros */

#define MIN(x,y) (x) > (y) ? y : x

#ifdef DEBUG
#define debug(str,args...) printk(KERN_DEBUG str,## args) ;
printk("\n")
#else
#define debug(str,args...)
#endif

#ifndef FALSE
#  define FALSE   0
#endif

#ifndef TRUE
#  define TRUE    !FALSE
#endif

#define MAC_ADDR_LEN    ETH_ALEN

#ifndef IW_ESSID_MAX_SIZE
#define IW_ESSID_MAX_SIZE 32
#endif

/* local structures */

typedef struct local_info
{
  dev_node_t node;
  struct net_device_stats stats;
#ifdef WIRELESS_EXT
  struct iw_statistics iw_stats;
#endif
  volatile u_char *mem_area;
```

```
  char essid[IW_ESSID_MAX_SIZE];
  int essid_len;
  int adapter_mode;
  int psp;
  int io_mode;
  struct map_info
  {
    u_char hop_set,hop_pattern,dtim;
  }map;

} local_info_t;

/* defines for adapter_mode */
#define MU  0
#define MAP 1

/* declaring an empty string here should cause it to broadcast
for an essid and then set its essid to the first one it finds */
#define DEFAULT_ESSID   ""

/* this is used by wireless extensions to determine the name of the
card. */
#define IF_NAME         "Spectrum24"

#define MAX_ARGS  15
#define MEM_WIN_SIZE    (32 * 1024)

typedef struct eth_header
{
  u_char dest_addr[MAC_ADDR_LEN],src_addr[MAC_ADDR_LEN];
} eth_hdr_t;

typedef struct rt_buff
{
  u_short rt_next;            // 0x00
  u_short rt_len;       // 0x02
  u_short rt_data_off;        // 0x04
  u_char unused1[2];          // 0x06
  u_short rt_timestamp_MSW;   // 0x08
  u_char rt_dest_addr[6];     // 0x0A
  u_char rt_src_addr[6];      // 0x10
  u_short rt_timestamp_LSW;   // 0x16
  u_char unused2[4];          // 0x18
  u_char rt_RSSI;       // 0x1C
  u_char rt_freq;       // 0x1D
  u_char unused3[2];          // 0x1E
} rt_buff_t;

/* function prototypes */

static dev_link_t *s24_attach(void);
static void s24_detach(dev_link_t *);
static int s24_eth_config (struct device *, struct ifmap *);
static void s24_config(dev_link_t *);
static void s24_release(u_long);
static int s24_tx(struct sk_buff *, struct device *);
static int s24_open(struct device *dev);
```

```
static void s24_interrupt IRQ(int , void *, struct pt_regs *);
static int s24_event(event_t event, int priority, event_callback_args_t
*args);
static int s24_stop(struct device *dev);
static struct net_device_stats *s24_stats(struct device *dev);

static void s24_reset(struct device *dev);
static void s24_multicast(struct device *);
static int s24_init(struct device *dev);
static int s24_ioctl(struct device *dev, struct ifreq *ifr, int cmd);

static void fill_stats(struct device *dev);

#ifdef HAS_WIRELESS_EXTENSIONS
static struct iw_statistics * s24_wireless_stats(struct device *dev);
#endif


/*====================================================================
**
* cs_error                                                 *
*                                                          *
* displays CardServices errors                                    *
====================================================================
*/

static inline void cs_error(client_handle_t handle, int func, int ret)
{
    error_info_t err = { func, ret };
    CardServices(ReportError, handle, &err);
}

/*====================================================================
**
* save_LMA, restore_LMA                                        *
*                                                          *
* saves the state of the LMA buffers so interrupts don't destroy other
*
* card functions                                           *
====================================================================
*/
static inline u_short save_LMA(struct device *dev)
{
  local_info_t *info = dev->priv;

  if (info->io_mode)
  {
    unsigned ioaddr = dev->base_addr;
    u_char save[2];

    save[0] = inb(ioaddr + LMA0);
    save[1] = inb(ioaddr + LMA1);

    return *(u_short *)save;
  }

  return 0;
```

```
}

static inline void restore_LMA(struct device *dev, u_short val)
{
  local_info_t *info = dev->priv;

  if (info->io_mode)
  {
    unsigned ioaddr = dev->base_addr;
    u_char *save = (u_char *)&val;

    outb(save[0], ioaddr + LMA0);
    outb(save[1], ioaddr + LMA1);
  }
}


/*********************************************************************
**
* get_bytes                                          *
*                                                          *
* grabs n bytes from the card and puts them in the buffer pointed to by
     *
* buffer.                                                  *
*********************************************************************
*/

static inline void get_bytes(struct device *dev, u_char *buffer,
u_short start, size_t count)
{
  local_info_t *info = dev->priv;
  debug("get_bytes");

  if (info->io_mode)
  {
    int i;
    unsigned ioaddr = dev->base_addr;
    debug("io mode");

    outb(start & 0xFF, ioaddr + LMA0);
    outb((start & 0xFF00) >> 8, ioaddr + LMA1);

    for (i = 0 ; i < count ; i++)
    {
      /* since the address autoincrements, we can just read and be
happy */
      buffer[i] = inb(ioaddr + MEM0);
    }
  }
  else
  {
    memcpy(buffer,(void *)(info->mem_area + start),count);
  }
}

/*********************************************************************
**
```

```
* put_bytes                                                    *
*                                                             *
* grabs count bytes from buffer and writes them to the card.
     *
****************************************************************
*/

static inline void put_bytes(struct device *dev, const u_char *buffer,
u_short start, size_t count)
{
  local_info_t *info = dev->priv;
  debug("put_bytes");

  if (info->io_mode)
  {
    int i;
    unsigned ioaddr = dev->base_addr;

    debug("io mode");

    outb(start & 0xFF, ioaddr + LMA0);
    outb((start & 0xFF00) >> 8, ioaddr + LMA1);

    for (i = 0 ; i < count ; i++)
    {
      /* since the address autoincrements, we can just write and be
happy */
      outb(buffer[i], ioaddr + MEM0);
    }
  }
  else
  {
#if defined(DEBUG) && 1
    if (count == 2)
    {
      debug("Short value is %d.",le16_to_cpu(*(u_short *)buffer));
    }
#endif
    debug("About to do memcpy to location %x for %x
bytes",start,count);
    memcpy((void *)(info->mem_area + start),buffer,count);
  }
}

/****************************************************************
**
* get_byte                                                     *
*                                                             *
* returns one byte from the card at location loc                  *
****************************************************************
*/

static inline u_char get_byte(struct device *dev, u_short loc)
{
  local_info_t *info = dev->priv;

  if (info->io_mode)
```

```
  {
    unsigned ioaddr = dev->base_addr;

    outb(loc & 0xFF, ioaddr + LMA0);
    outb((loc & 0xFF00) >> 8, ioaddr + LMA1);

    return inb(ioaddr + MEM0);
  }

  return info->mem_area[loc];
}

/*************************************************************************
**
* put_byte                                                    *
*                                                             *
* writes byte to the card at location loc                  *
*************************************************************************
*/

static inline void put_byte(struct device *dev, u_char byte, u_short
loc)
{
  local_info_t *info = dev->priv;

  if (info->io_mode)
  {
    unsigned ioaddr = dev->base_addr;

    outb(loc & 0xFF, ioaddr + LMA0);
    outb((loc & 0xFF00) >> 8, ioaddr + LMA1);

    outb(byte, ioaddr + MEM0);
  }
  else
  {
    info->mem_area[loc] = byte;
  }
}

/* the last of these functions (maybe) */

static inline u_short get_word(struct device *dev, u_short loc)
{
  u_short word;
  get_bytes(dev,(u_char *)&word,loc,2);
  return le16_to_cpu(word);
}

static inline void put_word(struct device *dev, u_short word, u_short
loc)
{
  word = le16_to_cpu(word);
  put_bytes(dev,(u_char *)&word,loc,2);
}
```

```
/*****************************************************************
**
* wait_on_woc                                                    *
*                                                          *
* waits for the woc bit to go true. due to the conditions under which
     *
* it's called, it shouldn't have to be reentrant.                 *
*****************************************************************
*/

static inline int wait_on_woc(struct device *dev, u_long timeout)
{
  int i;
  int ret_val = 0;

  debug("Waiting on WOC for %lu milliseconds", timeout);

  for (i = jiffies + (((HZ * timeout)/1000) ? ((HZ * timeout)/1000) :
1); i >= jiffies ; )
  {
    if (get_byte(dev,WOC))
    {
      ret_val = 1;
      break;
    }
  }

  return ret_val;
}

/*****************************************************************
**
* moon_cmd                                                 *
*                                                          *
* send a command to the card                                     *
* this assumes that you have already made sure that you have the woc
     *
* bit.                                                           *
*****************************************************************
*/

static inline void moon_cmd(struct device *dev, u_char cmd, const
u_char * args, int no_args)
{
  unsigned ioaddr = dev->base_addr;
  put_byte(dev,0,WOC);
  put_byte(dev,cmd,HCB);

  debug("Got command %.2x, with %d args",cmd,no_args);

  if (no_args > 0)
    put_bytes(dev,args,HCB+1,no_args);

  outb(GEN_P188_INT | inb(ioaddr+CMD),ioaddr+CMD);
}
```

```
/*****************************************************************
**
* moon_cmd_byte                                                  *
*                                                          *
* send a command to the card with a one byte argument             *
* this assumes that you have already made sure that you have the woc
     *
* bit.                                                           *
*****************************************************************
*/

static inline void moon_cmd_byte(struct device *dev, u_char cmd, const
u_char arg)
{
  unsigned ioaddr = dev->base_addr;
  put_byte(dev,0,WOC);
  put_byte(dev,cmd,HCB);
  put_byte(dev,arg,HCB+1);

  outb(GEN_P188_INT | inb(ioaddr+CMD),ioaddr+CMD);
}

#ifdef DEBUG
/*****************************************************************
**
* print_buffer                                                   *
*                                                        *
* a quick and dirty function to dump the contents of a memory location
     *
* used only for debugging. I could optimize it for io mode, but it's
     *
* for debugging, what do I care how efficient it is :p
     *
*****************************************************************
*/

static inline void print_buffer(u_long start, u_long end, struct device
*dev)
{
  int q;
  for ( ; start <= end ; start += 0x10)
  {
    printk(KERN_DEBUG);
    for (q = 0 ; q < 0x10 ; q++)
      printk("%.2x ", get_byte(dev,start + q));
    printk("\n");
  }
}
#else
#  define print_buffer(X, Y, Z)
#endif /* DEBUG */

#endif /* __SPECTRUM24_CS_H__ */
```

spectrum24_cs.c

```
/* (c) 1999 Symbol Technologies, all rights reserved. */
/* May be distributed under the terms of GNU's General Public License,
the
   text of which can be found online at
http://www.gnu.org/copyleft/gpl.html */
#include "spectrum24_cs.h"

/* globals */

static dev_info_t dev_info = "spectrum24_cs";
static dev_link_t *dev_list = NULL;  /* a linked list of devices
associated with this driver */

static char *essid = NULL;
static char *def_essid = DEFAULT_ESSID;
static int map_mode = FALSE;
static int psp_mode = 0;
static u_char map_dtim = 10;
static u_char map_hop_set = 0x01;
static u_char map_hop_pattern = 0xFF;
static int force_io_mode = FALSE;

MODULE_PARM(essid,"s");
MODULE_PARM(psp_mode,"i");
MODULE_PARM(map_mode,"i");
MODULE_PARM(map_dtim,"b");
MODULE_PARM(map_hop_set, "b");
MODULE_PARM(map_hop_pattern, "b");
MODULE_PARM(force_io_mode,"i");

static u_int irq_mask = 0xdeb8; /* standard irq mask */

int init_module(void)
{
  debug("init_module");

  EXPORT_NO_SYMBOLS;

  if (essid == NULL)
    essid = def_essid;
  else if (strlen(essid) > 32)
  {
    printk(KERN_NOTICE "%s: ESSID too long, using default.", dev_info);
    essid = def_essid;
  }
  register_pccard_driver(&dev_info,&s24_attach,&s24_detach);
  return 0;
}

void cleanup_module(void)
{
  debug("cleanup_module");
  unregister_pccard_driver(&dev_info);
```

```
  while (dev_list)
  {
    s24_detach(dev_list);
  }
}

static int s24_init(struct device *dev)
{
  return 0;
}

static int s24_ioctl(struct device *dev, struct ifreq *ifr, int cmd)
{
  u_long flags;
  local_info_t *info = dev->priv;
  struct iwreq *iwr = (struct iwreq *) ifr;
  int ret_val = 0;

  /* I'm not sure if this is necessary, but the sample code I looked at
did it so ... */
  save_flags(flags);
  cli();

  switch(cmd)
  {
  case SIOCGIWNAME:
    strcpy(iwr->u.name,IF_NAME);
    break;
  case SIOCGIWESSID:
    if (copy_to_user(iwr->u.data.pointer,info->essid,info->essid_len))
    {
      ret_val = -EFAULT;
      break;
    }
    iwr->u.data.length = info->essid_len;
    iwr->u.data.flags = TRUE;
    break;
  case SIOCSIWESSID:
    if (copy_from_user(info->essid,iwr->u.data.pointer,iwr-
>u.data.length))
    {
      ret_val = -EFAULT;
      break;
    }
    /* for some reason, it insists on including the nul character I'm
not sure if this breaks down around IW_ESSID_MAX_SIZE */
    info->essid_len = iwr->u.data.length - 1;
    if (dev->start)
    {
      s24_reset(dev);
    }
    break;
  default:
    ret_val = -EOPNOTSUPP;
  }
  restore_flags(flags);
```

```
  return ret_val;
}

static void fill_stats(struct device *dev)
{

}

#ifdef HAS_WIRELESS_EXTENSIONS
static struct iw_statistics * s24_wireless_stats(struct device *dev)
{
  debug("s24_wireless_stats");
  fill_stats(dev);
  return &((local_info_t *)dev->priv)->iw_stats;
}
#endif

/*====================================================================
**
* s24_attach                                                    *
*                                                          *
* sets up all the data types when a card is inserted              *
*====================================================================
*/

static dev_link_t *s24_attach(void)
{
  client_reg_t client_reg;
  dev_link_t *link;
  struct device *dev;
  local_info_t *info;
  int ret;

  debug("s24_attach");

  /* set up our link structure */
  link = kmalloc(sizeof(struct dev_link_t), GFP_KERNEL);

  if (link == NULL)
  {
    printk(KERN_NOTICE "%s: Unable to get free memory for link
struct.\n", dev_info);
    return NULL;
  }

  /* initialize it to zero. */
  memset(link,0,sizeof(struct dev_link_t));

  /* populate it */
  link->release.function = &s24_release;
  link->release.data = (u_long) link;
  link->io.NumPorts1 = 16;
  link->io.Attributes1 = IO_DATA_PATH_WIDTH_AUTO;
  link->io.IOAddrLines = 4;   /* I don't really know, and this isn't
curently used, should be modified to be correct at some later date. */
  link->irq.Attributes = IRQ_HANDLE_PRESENT;
  link->irq.IRQInfo1 = IRQ_INFO2_VALID;
```

```
  link->irq.IRQInfo2 = irq_mask;
  link->irq.Handler = &s24_interrupt;
  link->conf.Attributes = CONF_ENABLE_IRQ;
  link->conf.Vcc = 50;        /* specified in tenths of a volt */
  link->conf.IntType = INT_MEMORY_AND_IO;
  link->conf.ConfigIndex = ENABLE_IO_MODE;
  link->conf.Present = PRESENT_OPTION;

  dev = kmalloc(sizeof(struct device),GFP_KERNEL);

  if (dev == NULL)
  {
    s24_detach(link);
    printk(KERN_NOTICE "%s: Unable to get free memory for dev
struct.\n", dev_info);
    return NULL;
  }
  memset(dev,0,sizeof(struct device));
  dev->priv = kmalloc(sizeof(local_info_t),GFP_KERNEL);
  if (dev->priv == NULL)
  {
    s24_detach(link);
    printk(KERN_NOTICE "%s: Unable to get free memory for private
struct.\n", dev_info);
    return NULL;
  }
  memset(dev->priv,0,sizeof(local_info_t));

  info = dev->priv;

  info->essid_len = strlen(essid);
  memcpy(info->essid,essid,info->essid_len);

  if (map_mode)
    info->adapter_mode = MAP;
  else
    info->adapter_mode = MU;

  info->map.hop_set = map_hop_set;
  info->map.hop_pattern = map_hop_pattern;
  info->map.dtim = map_dtim;

  info->psp = psp_mode;

  if (force_io_mode)
  {
    info->io_mode = TRUE;
  }
  ether_setup(dev);

  dev->hard_start_xmit = &s24_tx;
  dev->set_config = &s24_eth_config;
  dev->get_stats = &s24_stats;
  dev->set_multicast_list = &s24_multicast;
  dev->init = &s24_init;
  dev->open = &s24_open;
  dev->stop = &s24_stop;
```

```
  dev->do_ioctl = &s24_ioctl;

#ifdef HAS_WIRELESS_EXTENSIONS
  dev->get_wireless_stats = &s24_wireless_stats;
#endif

  dev->name = info->node.dev_name;
  dev->tbusy = 1;
  link->priv = link->irq.Instance = dev;

  link->next = dev_list;

  dev_list = link;

  client_reg.dev_info = &dev_info;
  client_reg.Attributes = INFO_IO_CLIENT;
  client_reg.EventMask =
      CS_EVENT_CARD_INSERTION | CS_EVENT_CARD_REMOVAL |
      CS_EVENT_RESET_PHYSICAL | CS_EVENT_CARD_RESET |
      CS_EVENT_PM_SUSPEND | CS_EVENT_PM_RESUME;
  client_reg.event_handler = &s24_event;
  client_reg.Version = 0x0309; /* the version of card services?
currently ignored, see comment about address lines */
  client_reg.event_callback_args.client_data = link;

  ret = CardServices(RegisterClient,&(link->handle), &client_reg);
  if (ret != CS_SUCCESS)
  {
    cs_error(link->handle,RegisterClient,ret);
    s24_detach(link);
    return NULL;
  }

  debug("Successfully attached");
  return link;
}

/*====================================================================
**
* s24_detach                                                    *
*                                                          *
* releases all the data structures allocated with s24_attach()
      *
====================================================================
*/
static void s24_detach(dev_link_t *link)
{
  dev_link_t **linkp;
  long flags;

  debug("s24_detach");

  /* look through the linked list until either we find our link, or we
reach the end */
  for (linkp = &dev_list; *linkp != NULL && *linkp != link; linkp =
&(*linkp)->next)
      ;
```

```
  /* if we reached the end, return and don't do anything */
  if (*linkp == NULL)
  {
    debug("couldn't find in linked list.");
    return;
  }

  save_flags(flags);
  cli();

  if (link->state & DEV_RELEASE_PENDING)
  {
    del_timer(&link->release);
    link->state &= ~DEV_RELEASE_PENDING;
  }

  restore_flags(flags);

  /* if the device is currently active, mark it as stale and clean it
up when we get the release call */
  if (link->state & DEV_CONFIG)
  {
    s24_release((u_long)link);
    if (link->state & DEV_STALE_CONFIG)
    {
      debug("Got detach, but still busy.");
      link->state |= DEV_STALE_LINK;
      return;
    }
  }

  /* if the link was registered, deregister it */
  if (link->handle)
    CardServices(DeregisterClient, link->handle);

  /* Remove this from the linked list */
  *linkp = link->next;

  /* Free memory */
  if (link->priv != NULL)
  {
    struct device *dev = link->priv;
    if (dev->priv != NULL)
      kfree(dev->priv);
    kfree(link->priv);
  }

  kfree(link);
} /* s24_detach */

/*====================================================================
**
* s24_eth_config                                          *
*                                                         *
* is called by ifconfig to change things like irq and ioport currently
     *
```

```
* won't change any thing, in the future could be modifed to be more
     *
* flexible.                                                     *
*****************************************************************
*/

static int s24_eth_config (struct device *dev, struct ifmap *map)
{
  debug("s24_eth_config");
  return 0;
}

/*****************************************************************
**
* cold_reset                                                    *
*                                                               *
* will completely reset the adapter, and then bring it up in MU or MAP
     *
* mode depending on the adapter_mode flag                       *
*****************************************************************
*/

static int cold_reset(struct device *dev)
{
  u_long i;
  local_info_t *info = dev->priv;
  unsigned ioaddr = dev->base_addr;

  put_byte(dev,0,WARM_START_REG);

  debug("register looks like %.2x", (u_char)inb(ioaddr + HSR));

  /* set the reset bit */
  outb(BIT_7,ioaddr + CMD);

  mdelay(2);

  /* clear the reset bit */
  outb(0,ioaddr + CMD);

  mdelay(1);

  debug("register looks like %.2x", (u_char)inb(ioaddr + HSR));

  switch(info->adapter_mode)
  {
  case MAP:
    outb(HREG_INIT_MAP, ioaddr + HREG);
    break;
  case MU:
    outb(HREG_INIT_MU, ioaddr + HREG);
    break;
  default:
    return 1;
  }

  outb(HCR_PWRUP,ioaddr + CMD);
```

```
  /* this should allow it to cycle through waiting for the adapter to
wakeup,
   unfortunately it's a busy wait, but I don't know of a better
solution since
   this can be called at interrupt time */
  for (i = jiffies + HZ*3/5 ; jiffies < i && !(inb(ioaddr) & 0x20) ; )
  {
    mdelay(1);
  }

  debug("register looks like %.2x", (u_char)inb(ioaddr + HSR));

  if (inb(ioaddr + HSR) & HSR_P188_ACTIVE)
    return 0;

  /* if it gets here, it didn't wake up. */
  return 1;
}

#define CS_CHECK(fn,args...)\
if ((last_ret = CardServices(fn,args)) != CS_SUCCESS) \
{ cs_error(handle,fn,last_ret); s24_release((u_long)link); return; }

/*======================================================================
**
* s24_config                                                    *
*                                                          *
* sets up all the card resources (memory windows, ioports and irqs)
     *
======================================================================
*/

static void s24_config(dev_link_t *link)
{
  client_handle_t handle;
  tuple_t tuple;
  cisparse_t parse;
  struct device *dev;
  local_info_t *info;
  int last_ret;
  u_char buf[64];
  win_req_t winrq;

  debug("s24_config");

  handle = link->handle;
  dev = link->priv;
  info = dev->priv;

  tuple.DesiredTuple = CISTPL_CONFIG;
  tuple.Attributes = 0;
  tuple.TupleData = buf;
  tuple.TupleDataMax = sizeof(buf);
  tuple.TupleOffset = 0;

  CS_CHECK(GetFirstTuple, handle, &tuple);
```

```
  CS_CHECK(GetTupleData, handle, &tuple);
  CS_CHECK(ParseTuple, handle, &tuple, &parse);
  link->conf.ConfigBase = parse.config.base;
  link->conf.Present = parse.config.rmask[0];

  link->state |= DEV_CONFIG;

  winrq.Attributes = WIN_MEMORY_TYPE_CM | WIN_DATA_WIDTH_8 |
WIN_ENABLE;
  winrq.Base = 0;
  winrq.Size = MEM_WIN_SIZE;
  winrq.AccessSpeed = 100;

  link->win = (window_handle_t)handle;

  if (!info->io_mode && CardServices(RequestWindow,&link->win, &winrq)
!= CS_SUCCESS)
  {
    printk(KERN_NOTICE "%s: Unable to get a memory window, falling back
to IO mode\n", dev_info);
    info->io_mode = TRUE;
  }

  if (!info->io_mode)
  {
    debug("Remapping memory from %lx", winrq.Base);

    info->mem_area = ioremap(winrq.Base,MEM_WIN_SIZE);

    debug("Remapped to %p", info->mem_area);

    if (info->mem_area == NULL)
    {
      printk(KERN_NOTICE "%s: ioremap() failed, falling back to IO
mode\n", dev_info);
      info->io_mode = TRUE;
    }
  }

  debug("info->io_mode = %d",info->io_mode);

  CS_CHECK(RequestIRQ,handle,&(link->irq));
  debug("Got IRQ %d",link->irq.AssignedIRQ);

  CS_CHECK(RequestIO,handle,&(link->io));
  debug("Got IOPort %x",link->io.BasePort1);

  CS_CHECK(RequestConfiguration,handle,&(link->conf));

  dev->irq = link->irq.AssignedIRQ;
  dev->base_addr = link->io.BasePort1;
  dev->tbusy = 0;

  if (cold_reset(dev))
  {
    printk(KERN_NOTICE "%s: cold_reset() failed!\n", dev_info);
    s24_release((u_long)link);
```

```
    return;
  }

  if (register_netdev(dev))
  {
    printk(KERN_NOTICE "%s: register_netdev() failed!\n", dev_info);
    s24_release((u_long)link);
    return;
  }

  get_bytes(dev, dev->dev_addr, ETH_ADDR_LOC, MAC_ADDR_LEN);

#if 0
  debug("memory dump:");
  print_buffer(0x100,0x160,dev);
#endif

  link->dev = &((local_info_t *)dev->priv)->node;
  link->state &= ~DEV_CONFIG_PENDING;
}

/*======================================================================
**
* s24_release                                                  *
*                                                    *
* Releases everything created with s24_config                      *
======================================================================*/

static void s24_release(u_long arg)
{
  dev_link_t *link = (dev_link_t *)arg;
  local_info_t *info = ((struct device *)link->priv)->priv;

  debug("s24_release");

  if (link->open)
  {
    debug("release postponed, link still open.");
    link->state |= DEV_STALE_CONFIG;
    return;
  }

  CardServices(ReleaseConfiguration,link->handle);
  CardServices(ReleaseIO,link->handle, &link->io);
  CardServices(ReleaseIRQ,link->handle, &link->irq);

  CardServices(ReleaseWindow,link->win);

  if (link->dev != NULL)
  {
    unregister_netdev(link->priv);
    link->dev = NULL;
  }

  if (info->mem_area != NULL)
  {
```

```
    debug("iounmap %p",info->mem_area);
    iounmap((void *)info->mem_area);
    info->mem_area = NULL;
  }

  link->state &= ~(DEV_CONFIG | DEV_STALE_CONFIG);

  if (link->state & DEV_STALE_LINK)
    s24_detach(link);

}


/*======================================================================
**
* s24_tx                                                    *
*                                                           *
* the transmit function                                          *
======================================================================
*/

static int s24_tx(struct sk_buff *skb, struct device *dev)
{
  u_short len, n, save_len;
  u_long flags;
  char *data;
  u_short rt_buff, tx_buffer;

  debug("s24_tx");

  /* I'm not sure which interrupt level this runs at, but in case it
could be
     reentrant ... */
  save_flags(flags);
  cli();
  if (get_byte(dev,ASR) & TX_BUSY || dev->tbusy) /* theoretically
shouldn't happen */
  {
    debug("TX_BUSY or tbusy, %.2x %.2lx", get_byte(dev, ASR) & TX_BUSY,
dev->tbusy);
    if (get_byte(dev, ASR) & TX_BUSY)
    {
      restore_flags(flags);
      return -EBUSY;
    }
    printk(KERN_INFO "%s: failed to release tbusy.\n", dev_info);
  }

  if (!(get_byte(dev, ASR) & TX_FREE))
  {
    restore_flags(flags);
    debug("not enough space for a packet");
    return -EBUSY;
  }

  dev->tbusy = 1;
  restore_flags(flags);
```

```
  len = ETH_ZLEN < skb->len ? skb->len : ETH_ZLEN;
  data = skb->data;
  dev->trans_start = jiffies;

  /* actually write the data to the card */

  if (!wait_on_woc(dev,12))
  {
    debug("Timed out waiting for WOC");
    dev->tbusy = 0;
    return -EBUSY;
  }

  debug("got WOC, setting TX_BUSY");
  moon_cmd_byte(dev,CMD_ASR_SET,TX_BUSY);

  rt_buff = get_word(dev,TX_NEXT_BUFFER);

  put_bytes(dev, data, rt_buff + DEST_ADDR, 2 * MAC_ADDR_LEN);

  data += (2*MAC_ADDR_LEN);
  len -= (2*MAC_ADDR_LEN);

  /* save the length of the data portion of our buffer (minus the MAC
addresses) */
  save_len = le16_to_cpu(len);

#if 0
  /* if the type/length field is a length, it is an 802.3 frame
     in which case, it already has the header information, so we
     need to modify the transmit buffer registers to reflect that */
  if (htons(*(u_short *)data) <= 1500)
  {
    u_short data_size, data_off;

    data_off = get_word(dev,TX_BUFF_OFF);
    data_size = get_word(dev,TX_BUFF_SIZE);

    data_size += 8;
    data_off -= 8;

    put_word(dev,data_off,TX_BUFF_OFF);
    put_word(dev,data_size,TX_BUFF_SIZE);
  }
#endif

#if 0
  /* for some reason, TX_BUFF_OFF != rt_buff + DATA_OFF, and since
using the first values give problems, I'm trying the second */
  tx_buffer = get_word(dev,TX_BUFF_OFF);
  debug("got %x for the buffer offset from tx_buff_off and %x from the
buffer itself",tx_buffer,get_word(dev,rt_buff+DATA_OFF));
  n = MIN(get_word(dev,TX_BUFF_SIZE),len);
#endif

  while (len > 0 && rt_buff != 0)
```

```
  {
    tx_buffer = get_word(dev,rt_buff+DATA_OFF);
    n = MIN(get_word(dev,rt_buff+BUFF_LEN),len);
    put_bytes(dev,data,tx_buffer + rt_buff,n);
    data += n;
    len -= n;
    rt_buff = get_word(dev,rt_buff+LINK_PTR);
  }

  if (!wait_on_woc(dev,12))
  {
    debug("Timed out waiting for WOC");
    dev->tbusy = 0;
    return 1;
  }
  debug("Got WOC, sending tx_list_command.");

  moon_cmd(dev,CMD_TXL,(u_char *)&save_len,2);

  dev_kfree_skb(skb);

  return 0;
}

/*********************************************************************
**
* s24_open                                                  *
*                                                           *
* The callback that should bring the card up and start radio activity.
     *
*********************************************************************
*/

static int s24_open(struct device *dev)
{
  dev_link_t *link;
  debug("s24_open");

  for(link = dev_list; link != NULL && link->priv != dev; link = link-
>next)
      ;

  if (link == NULL)
    return -ENODEV;

  link->open++;

  MOD_INC_USE_COUNT;

  dev->interrupt = 0;
  dev->tbusy = 0;
  dev->start = 1;

  s24_reset(dev);

  return 0;
}
```

```
/*****************************************************************
**
* s24_interrupt                                                 *
*                                                       *
* The ISR. It deals with RX_COMPLETE, TX_COMPLETE, RX_ERROR and
      *
* STATUS_CHANGE events                                          *
*****************************************************************
*/

static void s24_interrupt IRQ(int irq, void *dev_id, struct pt_regs
*regs)
{
  struct device *dev = dev_id;
  local_info_t *info;
  int ioaddr;
  int i;
  u_char status, tstat;
  u_short save;

  debug("int");

  if (dev == NULL || !dev->start)
    return;

  info = dev->priv;
  ioaddr = dev->base_addr;

  save = save_LMA(dev);

  for(i = 0;(status = get_byte(dev,ASR)) & ASR_INT_MASK && i < 100 ;
i++)
  {
    debug("status = 0x%.2x, ASR = 0x%.4x",status, ASR);

    if (status & RX_COMPLETE)
    {
      struct sk_buff *skb;
      u_short pkt_offset, pkt_len, len;
      u_short data;

      pkt_offset = get_word(dev,RX_PKT_START);
      pkt_len = get_word(dev,RX_PKT_LEN);

      debug("Got packet! %x long at %x",pkt_len,pkt_offset);

      skb = dev_alloc_skb(pkt_len + (2 * MAC_ADDR_LEN) + 2);

      if (skb == NULL)
      {
        printk(KERN_NOTICE "Unable to get memory for skb!\n");
        goto end_receive;
      }

      /* aligns the ip header on a 16 byte boundry (2 + the 14 for the
ethernet header) */
```

```
      skb_reserve(skb,2);

      get_bytes(dev,skb_put(skb,(2 *
MAC_ADDR_LEN)),pkt_offset+DEST_ADDR,(2 * MAC_ADDR_LEN));

      while (pkt_len > 0)
      {
        if ((len = get_word(dev,pkt_offset+BUFF_LEN)) > pkt_len)
        {
          debug("buff size error, pkt_len = %x, buff_len =
%x",pkt_len,len);
          len = pkt_len;
        }

        data = get_word(dev, pkt_offset + DATA_OFF) + pkt_offset;

        if (skb_tailroom(skb) < len)
        {
          printk(KERN_NOTICE "%s: serious error with receive socket
buffer. Please report this to lkeyser@wpi.edu\n", dev_info);
          dev_kfree_skb(skb);
          goto end_receive;
        }

        get_bytes(dev,skb_put(skb,len),data,len);
        pkt_len -= len;
        pkt_offset = get_word(dev,pkt_offset+LINK_PTR);

        if (pkt_len && ! pkt_offset)
        {
          printk(KERN_NOTICE "packet buffer error\n");
          goto end_receive;
        }
      }

      skb->dev = dev;
      skb->protocol = eth_type_trans(skb,dev);

      netif_rx(skb);

      end_receive:

      if (!wait_on_woc(dev,12))
      {
        debug("Timed out waiting for WOC");
        break;
      }
      debug("Got WOC, clearing ASR");
      moon_cmd_byte(dev,CMD_ASR_CLR,RX_COMPLETE);

      if (!wait_on_woc(dev,12))
      {
        debug("Timed out waiting for WOC");
        break;
      }
      debug("Got WOC, issuing skip recieve");
      moon_cmd(dev,CMD_SKR,NULL,0);
```

```
  wait_on_woc(dev,12);
  debug("clear to cycle");
}

if (status & TX_COMPLETE)
{
  debug("tx_done");
  if (!((tstat = get_byte(dev,TSR)) & TX_COMPLETE))
    info->stats.tx_errors++;

  if (!wait_on_woc(dev,12))
  {
    debug("Timed out waiting for WOC");
    break;
  }
  debug("Got WOC, clearing transmit status");

  moon_cmd_byte(dev,CMD_TSR_CLR,tstat);
  wait_on_woc(dev,12);
  debug("clear to cycle");

  if (!(get_byte(dev,ASR) & TX_BUSY))
  {
    debug("buffer free");
    dev->tbusy = 0;
    mark_bh(NET_BH);
  }
}

if (status & STATUS_CHANGE)
{
  debug("Status change");

  if (!wait_on_woc(dev,12))
  {
    debug("Timed out waiting for WOC");
    break;
  }
  debug ("Got WOC, clearing status change bit in ASR");

  moon_cmd_byte(dev,CMD_ASR_CLR,STATUS_CHANGE);

  wait_on_woc(dev,12);
  debug("clear to cycle");
}

/* this is obsolete but it is included for completeness */

if (status & RX_ERROR)
{
  debug("Got rx error");

  info->stats.rx_errors++;

  if (!wait_on_woc(dev,12))
  {
```

```
        debug("Timed out waiting for WOC");
        break;
      }

      moon_cmd_byte(dev,CMD_ASR_CLR,RX_ERROR);

      if (!wait_on_woc(dev,12))
      {
        debug("Timed out waiting for WOC");
        break;
      }

      moon_cmd_byte(dev,CMD_ASR_CLR,RX_ERROR);
      wait_on_woc(dev,12);
      debug("clear to cycle");
    }
  }

  /* restore the LMA registers if need be */
  restore_LMA(dev,save);

  /* acknowledge the interrupt */
  outb(HSR_P188_INT,ioaddr + HSR);
}

/*****************************************************************
**
* s24_multicast                                                  *
*                                                          *
* This deals with the instances in which we are thrown changes to our
     *
* interface flags.                                               *
*****************************************************************
*/

static void s24_multicast(struct device *dev)
{
  debug("set_multicast");
}

/*****************************************************************
**
* s24_stats                                                 *
*                                                           *
* Does statistics reporting                                      *
*****************************************************************
*/

static struct net_device_stats *s24_stats(struct device *dev)
{
  debug("s24_stats");
  fill_stats(dev);
  return &((local_info_t *)dev->priv)->stats;
}

/*****************************************************************
**
```

```
* s24_stop                                                     *
*                                                             *
* The callback in charge of bringing the card down.                *
*****************************************************************
*/

static int s24_stop(struct device *dev)
{
  dev_link_t *link;

  debug("s24_stop");

  for (link = dev_list; link != NULL && link->priv != dev; link = link-
>next)
      ;

  if (link == NULL)
    return -ENODEV;

  link->open--;

  dev->tbusy = 1;
  dev->start = 0;

  cold_reset(dev);

  MOD_DEC_USE_COUNT;

  return 0;
}

/*****************************************************************
**
* s24_event                                                   *
*                                                             *
* Handles pcmcia events                                            *
*****************************************************************
*/

static int s24_event(event_t event, int priority, event_callback_args_t
*args)
{
  dev_link_t *link = args->client_data;
  struct device *dev = link->priv;

  debug("s24_event");

  switch(event)
  {
  case CS_EVENT_CARD_INSERTION:
    debug("Got event card insertion");
    link->state |= DEV_PRESENT | DEV_CONFIG_PENDING;
    s24_config(link);
    break;
  case CS_EVENT_CARD_REMOVAL:
    debug("Got event card removal");
    link->state &= ~DEV_PRESENT;
```

```
    if (link->state & DEV_CONFIG)
    {
      dev->tbusy = 1;
      dev->start = 0;
      link->release.expires = RUN_AT(HZ/20);
      add_timer(&(link->release));
    }
    break;
  case CS_EVENT_PM_SUSPEND:
    debug("Got event pm suspend");
    link->state |= DEV_SUSPEND;
    debug("falling through to event reset physical...");
  case CS_EVENT_RESET_PHYSICAL:
    debug("Got event reset physical");
    if (link->state & DEV_CONFIG)
    {
      if (link->open)
      {
            dev->tbusy = 1;
            dev->start = 0;
      }
//      CardServices(ReleaseConfiguration, link->handle);
    }
    break;
  case CS_EVENT_PM_RESUME:
    debug("Got event pm resume");
    link->state &= ~DEV_SUSPEND;
    debug("falling through to event card reset...");
  case CS_EVENT_CARD_RESET:
    debug("Got event card reset");
    if (link->state & DEV_CONFIG)
    {
//      CardServices(RequestConfiguration, link->handle, &(link-
>conf));
      if (link->open)
      {
        cold_reset(dev);
        s24_reset(dev);
        dev->tbusy = 0;
        dev->start = 1;
      }
    }
    break;
  default:
    debug("Got unknown event");
  }

  return 0;
}

/*****************************************************************
**
* s24_reset                                                     *
*                                                               *
* Resets the on-card configuration parameters                    *
*****************************************************************
*/
```

```
static void s24_reset(struct device *dev)
{
  local_info_t *info = dev->priv;
  int ioaddr = dev->base_addr;

  debug("s24_reset");

  if (!wait_on_woc(dev,12))
  {
    debug("s24_reset couldn't get WOC, bailing out");
    return;
  }
  debug("Got WOC, sending reset");

  moon_cmd(dev,CMD_RESET,NULL,0);

  if (!wait_on_woc(dev,12))
  {
    debug("s24_reset couldn't get WOC, bailing out");
    return;
  }
  debug("got WOC, initializing receive");

  /* turn on the reciever (Bit 7) enable broadcast (bit 2)  *
   * disable antenna diversity (bit 0)                      *
   * the other options are enable multicast (bit 3)         *
   * and promiscuous mode (bit 4)                      */

  moon_cmd_byte(dev, CMD_SRC, BIT_7 | BIT_3 | BIT_2 | BIT_0);

  debug("waiting for WOC");

  if (!wait_on_woc(dev,12))
  {
    debug("s24_reset couldn't get WOC, bailing out");
    return;
  }
  debug("got WOC, initializing transmit");

  /* bit 7 is enable transmit and bit 0 is loop back packets */
  moon_cmd_byte(dev, CMD_STC, BIT_7);

  /* set up the map if that's our goal */
  if (info->adapter_mode == MAP)
  {
    if (!wait_on_woc(dev,12))
    {
      debug("s24_reset couldn't get WOC, bailing out");
      return;
    }
    debug("got WOC, setting MAP mode");

    moon_cmd(dev, CMD_MAP_CONFIG, (u_char *)&info->map, 3);
  }

  /* PSP stuff -- needs to be completed */
```

```
  if (!wait_on_woc(dev,12))
  {
    debug("s24_reset couldn't get WOC, bailing out");
    return;
  }
  debug("got WOC, setting PSP mode");

  moon_cmd_byte(dev, CMD_PMG, info->psp ? BIT_0 : 0);

  /* make the essid go */
  if (!wait_on_woc(dev,12))
  {
    debug("s24_reset couldn't get WOC, bailing out");
    return;
  }
  debug("got WOC, initializing essID ... essid_len = %d", info-
>essid_len);

  /* put the essid in the extra command args buffer assuming it exists
*/
  if (info->essid_len)
    put_bytes(dev,info->essid, 0xE0, info->essid_len);

  /* length of essid 0 means broadcast */
  put_word(dev, info->essid_len, HCB+1);

  moon_cmd(dev, CMD_ESSID, NULL,0);

  if (!wait_on_woc(dev,12))
  {
    debug("Unable to get WOC, bailing out.");
    return;
  }
  debug("got WOC, clear to set interrupts");

  outb(ENABLE_P188_INT | inb(ioaddr+CMD),CMD +ioaddr);
}
```