IP-Over-USB Gateway

Final Project Report

Ben Greenberg Bartosz Mach

Adviser: Dr. Vincenzo Liberatore

Name	Signature	Date
B. Greenberg		
B. Mach		
V. Liberatore		

Executive Summary

The goal of team members of the IP-over-USB gateway project, Ben Greenberg, Bartosz Mach, and Prof. Vincenzo Liberatore was to implement a robust gateway on the Linux platform that is compatible with any device that implements a standard Ethernet over the USB protocol. The gateway allows such devices to automatically receive a temporary IP address and Internet access with minimal user invention required in configuration, using only a standard USB 1.1 or 2.0 connection. In practice, computers running the Linux operating system can access the gateway easily. Other operating systems have not been tested extensively.

The gateway was implemented on a desktop PC equipped with USB gadget interface cards and running the Debian 3.1 distribution of Linux. The gateway is designed to allow both USB host devices (normally other desktop PCs) and USB gadgets (PDAs, cameras, and similar devices) to connect, but only hosts were able to be tested at this time. The main services provided include Dynamic Host Configuration Protocol (DHCP), which dynamically configures the client machine's IP settings, and Network Address translation (NAT), which allows the clients to access a wider network such as the Internet. The gateway is setup to automatically configure itself at boot time and can adapt to any number of gadget interface cards being installed. Testing was done with PC clients running Debian 3.1 as well as Ubuntu 5.04 (a Linux distribution very similar to Debian) running on both x86 (Intel/AMD) and PowerPC (Macintosh) platforms.

Tests have been performed with up to three concurrent clients. Performance, when using a USB 2.0, is faster than 100baseT Ethernet but latency is worse.

One feature that remains to be implemented is manual configuration of network hardware addresses. The current prototype generates hardware addresses for the USB interfaces randomly, making for a very slight possibility of conflict if two interfaces have the same address. Besides this feature and the testing of USB gadgets as clients, the project follows the original plan and functions successfully.

Abstract

The team members of the IP-over-USB project, team leader Ben Greenberg, Bartosz Mach, and adviser Prof. Vincenzo Liberatore, have built and configured a network gateway device that provides NAT and DHCP services over USB connections. The gateway is implemented on a Linux desktop PC and provides network services over USB using the `usbnet` and `g_ether` drivers that ship with the Linux kernel. These drivers support the CDC Ethernet standard.

The gateway auto-configures itself using a variety of scripting hooks, including hotplugging. The gateway has been tested with up to three Linux clients connected concurrently and, when using a USB 2.0 interface, has transfer speeds competitive with Ethernet.

Introduction

As small electronic devices become commonplace, the desire for these devices to have network connectivity is increasing. The most common network protocol today is Internet Protocol (IP), which is used in the worldwide Internet as well as most private local area and wide area networks. Traditionally, devices networked via IP require the use of cat5 Ethernet cable. However, the ability to fit a cat5 connector onto increasingly smaller form factors is becoming difficult. Dr. Liberatore has suggested the use of USB to network these devices, because USB offers a “mini-B” form factor connector that is less than half the size of an Ethernet connection.

To prove that USB can be used to build complex and useful networking configurations, a full featured USB-over-IP gateway running on the Linux platform has been implemented. The gateway allows Linux-based PCs that implement the CDC Ethernet Protocol to be automatically connected to an internal network and the access the Internet with very little configuration. The gateway provides DHCP and NAT services to provide these features.

A team consisting of Ben Greenberg (team leader) and Bartosz Mach has implemented the gateway under the advisement of Dr. Vincenzo Liberatore. Though many of the tasks were completed cooperatively by Mr. Greenberg and Mr. Mach, the task of modifying the USB card device drivers was completed by Mr. Greenberg, and the configuration of NAT service and most of the work in configuring the DHCP service was completed by Mr. Mach.

Takahiro Hirofuchi is currently working on a similar project entitled USB/IP Project (http://usbip.naist.jp). This project is similar in some ways, but Hirofuchi is working on doing the opposite of this project; he is carrying USB traffic over traditional IP networks so that a USB device can be shared over an existing network.

Methodology

The gateway is implemented on a desktop PC running Debian GNU/Linux 3.1 (a.k.a "sarge;" http://www.debian.org/releases/sarge/). The gateway is equipped with a standard Ethernet adapter connected to CWRUnet (to provide access to the Internet), four USB Gadget adapters to provide connectivity to other desktop PCs, and two USB Host ports to provide connectivity to USB peripherals. The gateway, running NAT and DHCP services, is implemented on the machine called `sasha`. In the lab environment, three other available machines are connected to `sasha` via USB and are acting as clients. Two of these, `bruno` and `borat`, are EECS department provided machines also running Debian GNU/Linux 3.1. The other machine, "Bart's Laptop" is group member Bartosz Mach's personal laptop running Ubuntu Linux 5.04 (http://www.ubuntulinux.com). All are running 2.6 series Linux kernels. The Hardware Overview (Appendix I) shows how the router is connected to the client machines. The Software Overview (Appendix I) shows these interconnections in terms of software components. Note that the USB Host ports are currently unused. Hardware to connect to these ports was not obtainable.

Three major steps were required to build the gateway: encapsulation of Ethernet frames over USB, configuration of NAT and DHCP, and automation.

Ethernet Over USB

In order to send IP packets over USB, encapsulation of an existing physical layer protocol was chosen. The alternative would have been creating a physical layer protocol for USB, which would be a daunting task. Additionally, the Linux 2.6 series kernel includes drivers that provide encapsulation of Ethernet framing over USB. The `usbnet` and `g_ether` drivers provide this capability for USB host and USB gadget, respectively. These drivers utilize the CDC Ethernet Networking Control Model, a standard created by the USB Implementer's Forum. When these drivers are loaded, they create a standard network interface visible to the operating system that can be used the same as any conventional network interface.

Using these drivers eliminates most of the work of using USB to handle network traffic. However, the `g_ether` driver that ships with the Linux kernel version 2.6.8 (the development platform) only supports a single gadget adapter card. Each adapter card only has one gadget interface, but the gateway specification calls for more than one interface to be available. To support more than one card, the g_ether driver was modified to bind a driver to all gadget interface cards found on the system. The new version of the driver was installed over the existing version on the `sasha` machine and supports all four of the computer's adapter cards. The output of the `diff` command, showing the modifications to the driver, is attached in Appendix II.

NAT and DHCP Configuration

The core functionality of the gateway is NAT, network address translation.

Configuration of NAT in Linux is handled through the `iptables` tool. Rules were formulated to allow client machine's packets to be forward to CWRUnet and incoming responses to be forwarded back to the correct recipient.

The network settings of client machines are dependent on the configuration of the gateway, so DHCP functionality was added to provide “plug and play” configuration of clients. A client machine simply needs to connect the USB cables and run a DHCP client program to retrieve the settings from the gateway. On the Linux clients used in the lab, the `dhclient` program is used. When the gateway is configured properly, this step is all that is necessary for the client to establish a network connection to CWRUnet via the gateway.

Automation

Configuring the gateway properly requires a series of commands to be executed and configuration files to be modified. These settings vary depending on the number of interface cards enabled. While altering these settings by hand is quite possible, our gateway spec calls for automatic configuration. A series of shell scripts, called by the system's startup procedure and the Linux kernel's “hotplug” functionality, fully configures the gateway's NAT and DHCP settings. After boot up, the machine is ready to be connected to a client and forward packets. The system also adapts to the number of interface cards that are installed at boot time.

Scripting occurs in three sections: a hook to dhclient that is called after an IP address is assigned to the Ethernet interface, an init.d script that is called at boot time, and a modification to the hotplugging subsystem net.agent script. The flow of how these scripts are called is shown in Appendix III. Output of the relevant scripts is attached in Appendix IV.

Comparison With Original Plan

The project originally had the following tasks:

1. ✓ Order and install USB gadget cards.
2. ✓ Modify `net2280` driver to support multiple cards.
3. ✓ Configure NAT.
4. Develop hardware addressing scheme.
5. ✓ Configure dynamic addressing server.
6. Obtain and configure a USB gadget to network.
7. ✓ Alter scripts to provide automation for configuration.

Tasks that have been completed are checked off.

A hardware addressing scheme never was completed. At one point, this task was removed from the project plan but added again later. By the time this task was added a second time, it was determined that implementing a general and correct solution would require further modification of the `net2280` driver. Development

on the driver had already been completed and it was decided that the cost of further modifications (in terms of time and possible conflicts) was not expendable at this point in the project. The purpose of implementing a static hardware addressing scheme is to ensure that none of the hardware addresses randomly generated by the `g_ether` driver collide with each other. Each hardware address is 48 bits long, meaning that the probability of a collision is $2/2^{48}$, or $7.1x10^{-15}$.

At the time the project was started, it was believed that a compatible USB gadget would be obtainable from a lab on campus. As time went on, it became apparent that a gadget was not easy to come by. This task had to be dropped.

Results

Functionality

A set of requirements to verify the completeness of the project was established. These requirements are divided for the gateway machine (`sasha`) and the client machines.

Gateway

1. After system boot, all network interfaces (Ethernet and USB) will be created and enabled.
2. After system boot, NAT will be fully configured to allow CWRUnet access to client machines and forward appropriate packets back to these machines.
3. After system boot, a DHCP server should be configured and enabled such that any client machine may request network settings from the gateway.
4. All of the above should work no matter how many interface cards are installed in the gateway.
5. The gateway should continue to operate properly if any clients suddenly disconnect.

Client

1. The appropriate driver should load when a USB cable is connected from the client to the gateway.
2. All necessary network settings should be configurable via DHCP—no other configuration should be required.
3. After DHCP configuration, the client should be able communicate with the gateway directly.
4. After DHCP configuration, the client should be able to communicate with any publicly accessible machine on CWRUnet (this includes the Internet).

All of these requirements have been met by the project at this point.

Performance

Throughput and latency were tested in order to compare the USB network with a traditional Ethernet network. The lab computers all had USB 1.1 ports, but Bart's laptop (running Ubuntu Linux 5.04) had USB 2.0 ports. USB 2.0 is rated at 480Mbps, USB 1.1 at 12Mbps, and the Ethernet used is 100BaseT.

Within the local network

These results show bandwidth and latency on the local network of the gateway, within the EECS Networks Lab. Throughput results were generated by transmitting a 200MB file from the sender to the receiver via FTP using the GNU `ftp` utility. For each computer pair the file was transmitted 3 times and the average throughput is listed. RTT times were generated by issuing 50 pings (using the GNU `ping` utility) from the sender to the receiver. For each pair, the minimum, average, and maximum RTT is listed.

Sender	*Receiver*	*Avg Throughput (Kbs)*	*RTT (min/avg/max) (ms)*	*Protocol*
`sasha`	Laptop	12738	0.2/0.3/1.3	USB 2.0
`sasha`	`bruno`	972	0.6/1.4/3.1	USB 1.1
`sasha`	`alig`	7480	0.2/0.2/0.3	Ethernet

Table 1: Results of tests on the local network.

Within CWRUnet

These results show performance when connected to another machine on the Case network, specifically a machine in the Jennings Computing Center. The tests are the same ones used on the local network.

Sender	*Receiver*	*Avg Throughput (Kbs)*	*RTT (min/avg/max) (ms)*	*Protocol*
Laptop	`megadeth`	2100	0.2/0.3/1.3	USB 2.0
`bruno`	`megadeth`	691	0.6/1.4/3.1	USB 1.1
`alig`	`megadeth`	2000	0.2/0.2/0.3	Ethernet

Table 2. Results of tests on CWRUnet.

Over the Internet

To test performance on the wider Internet, a 42.6MB file was fetched via FTP from each machine. The file was fetched three times and the average transfer speed is listed. To determine RTT, 50 pings were sent from the sender to the receiver and the average, minimum, and maximum RTTs are listed. The file fetched is `ftp.kernel.org/pub/linux/kernel/v2.6/linux-2.6.9.tar.gz` and `ftp.kernel.org` is the server pinged.

Sender	Receiver	Avg Throughput (Kbs)	RTT (min/avg/max) (ms)	Protocol
sasha	Laptop	867	52.2/52.9/58.9	USB 2.0
sasha	bruno	498	52.8/5.7/56.1	USB 1.1
sasha	alig	533	51.0/51.6/57.4	Ethernet

Table 3: Results of tests on the Internet.

These results show that USB 2.0 is highly competitive with Ethernet when it comes to bandwidth. In fact, the USB 2.0 transfers are much faster than Ethernet when transferring directly to the gateway. This was expected due to USB 2.0's 480Mbs rated speed in considerably higher the 100BaseT Ethernet used in the tests. The USB 1.1 transfers, rated at 12Mbs, were significantly slower. Although USB 2.0's transfer speeds were fast, there were a good deal lower than the rated speed. It is likely that this is due to the overhead introduced the by the encapsulation drivers, and further research is needed on this issue.

Surprisingly, USB 2.0 did a good deal better than Ethernet when downloading files from the Internet. The explanation is not clear, since this traffic is received by the gateway over Ethernet before being sent over USB, but it should be noted that these tests were casual and not performed in a scientific manner.

USB 2.0 is also competitive when it comes to latency, although it did have higher maximum values than Ethernet. USB 1.1 performed very poorly. However, even USB 1.1's miserable performance doesn't matter when accessing the Internet, where latency bottlenecks overshadow these differences.

Implications

The major implication of the IP-Over-USB gateway project is that it paves the way for a standalone device that provides gateway functionality over USB. In such a device became commonplace, any device with CDC Ethernet and a USB port could connect to the Internet in a “plug and play” fashion. It would be feasible for a manufacturer to load a scaled-down Linux operating system into a small device such as a PDA or digital camera and have confidence that a user with access to such a router could easily have it access the Internet.

The bandwidth and latency results show that USB (at least USB 2.0) has the performance to be a viable technology for networking.

Conclusions

The team's main goal of configuring a Linux-based PC to act as an IP-Over-USB gateway has been fully implemented. Linux-based client PCs are able to connect to the host PC and automatically receive a temporary IP address, routing and DNS information, and receive full access to the local network (including other client PCs connected to the gateway) and the Internet. The gateway

automatically configures itself to provide these services at boot time.

One of the original goals that was not completed was the task of obtaining a CDC Ethernet compliant USB gadget, such as a PDA, and using it to demonstrate IP-Over-USB on small devices. This task was dropped due to the lack of availability of such a gadget. Additionally, one of the three client PCs broke down and was no longer usable for testing. The team members contacted those in charge of equipment in the networking lab and were informed that repair was not possible at the time. However, a workaround was implemented by replacing the broken-down PC with Bart's notebook running Ubuntu Linux. This workaround allowed the team members to successfully test IP-Over-USB on a portable computer running a distribution of Linux different from that of the host PC, as well as test performance over USB 2.0 connections.

Another original goal that was dropped was configuring the host PC to eliminate the chance of two USB devices receiving the same MAC address. This task was dropped due to the fact that it would require extensive modification of the USB card device driver that time would not allow. However, the probability of a conflict is very unlikely.

Recommendations

Over the course of the project, the configuration changes in the host PC were applied manually by the team members. Although a person with knowledge of Linux could replicate the team members' work, a first step to furthering the project would be to automate the configuration in an easy-to-use install script that would prepare a desktop PC with appropriate hardware to act as a gateway. Additionally, testing should be done using USB gadgets as clients and the issue of a possible MAC address clash should also be solved to make the modified USB card driver “airtight.”

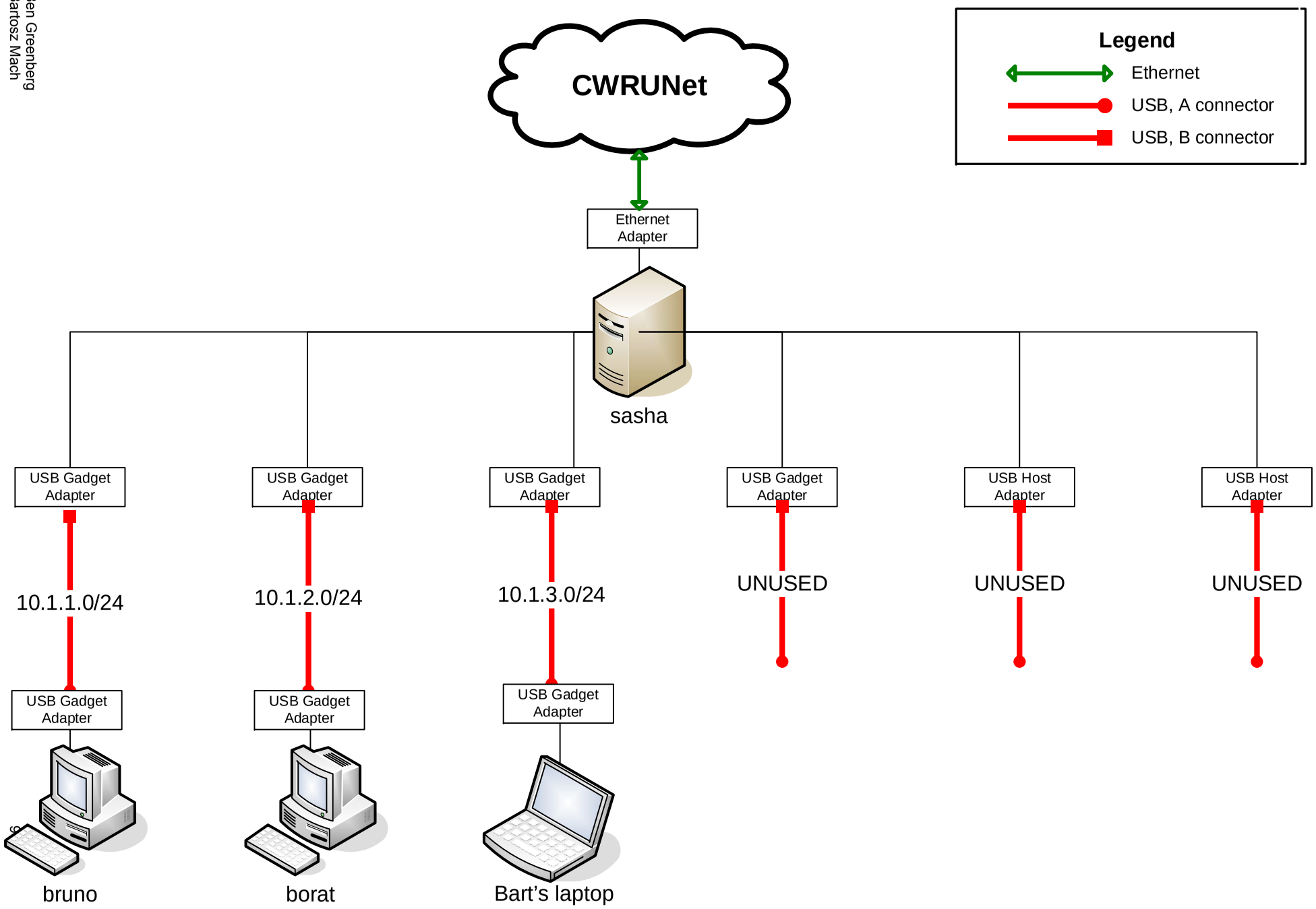
In the long run, a scaled down Linux distribution consisting of only what is required for the gateway could be created and compiled for an embedded processor. This would allow a manufacturer to produce a small standalone device acting as the gateway, similar to how Ethernet gateways are currently sold.

More tests should be done on USB's performance. It would be interesting to know how much of a role the encapsulation driver overhead plays on performance. It would also be useful to have a detailed, scientific analysis of results. Additionally, it is important to determine how cable length and quality plays a role on performance.

HARDWARE OVERVIEW

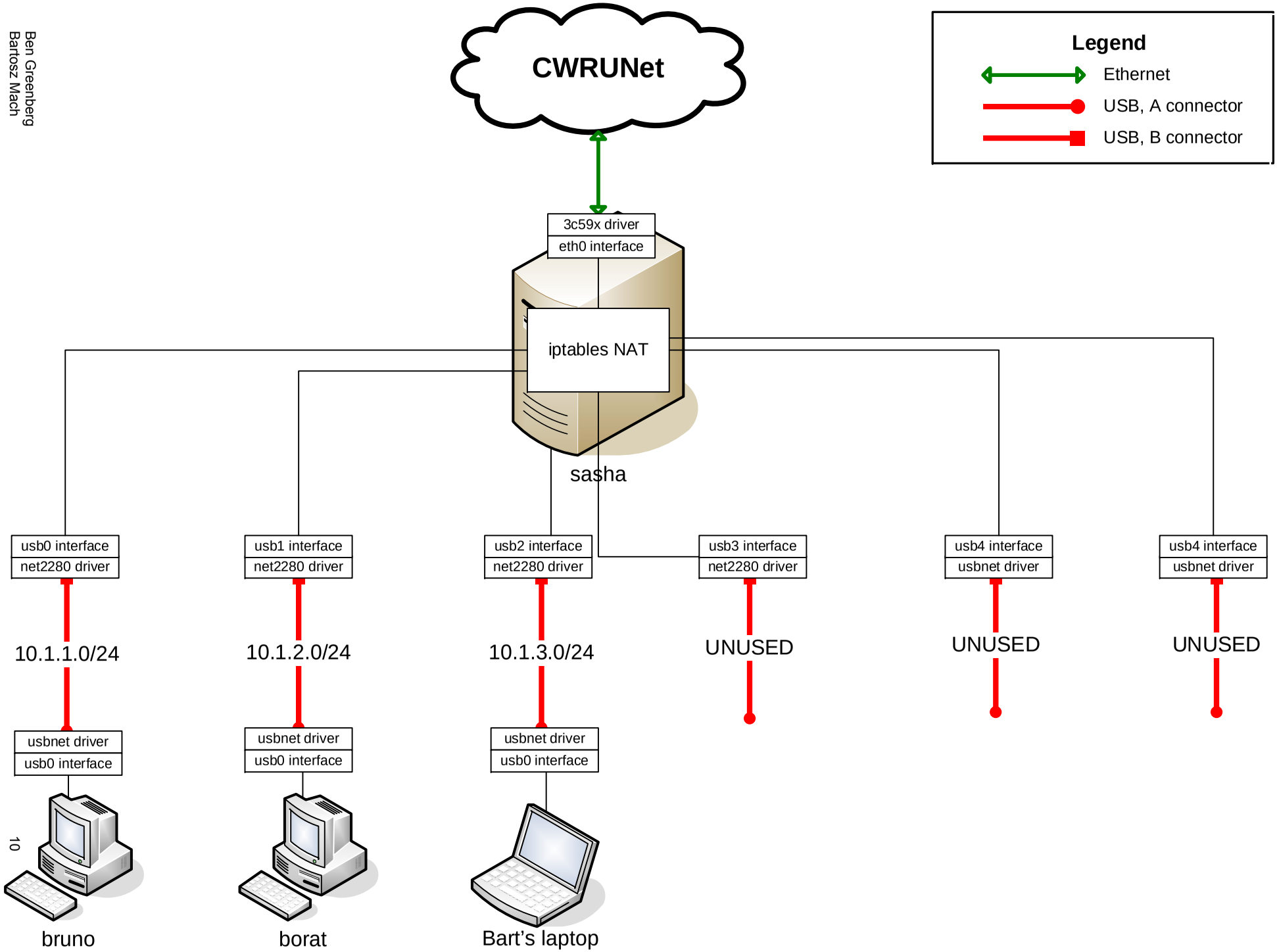
Ben Greenberg
Bartosz Mach

APPENDIX I, Part A



SOFTWARE OVERVIEW

Ben Greenberg
Bartosz Mach



APPENDIX II: Changes made to the net2280 driver

```
--- net2280/net2280.c   2005-02-19 18:12:15.000000000 -0500
+++ kernel-source-2.6.8/drivers/usb/gadget/net2280.c    2004-08-14 01:36:14.000000000 -0400
@@ -44,8 +44,8 @@
  * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
  */

-#define        DEBUG           /* messages on error and most fault paths */
-#define        VERBOSE         /* extra debug messages (success too) */
+#undef DEBUG           /* messages on error and most fault paths */
+#undef VERBOSE         /* extra debug messages (success too) */

 #include <linux/config.h>
 #include <linux/module.h>
@@ -82,7 +82,6 @@
 #define USE_RDK_LEDS           /* GPIO pins control three LEDs */
 #define USE_SYSFS_DEBUG_FILES

-#define MAX_CONTROLLERS                10

 static const char driver_name [] = "net2280";
 static const char driver_desc [] = DRIVER_DESC;
@@ -1802,12 +1801,7 @@ EXPORT_SYMBOL (net2280_set_fifo_mode);
  * perhaps to bind specific drivers to specific devices.
  */

-//static struct net2280        *the_controller;
-//static struct net2280        *controllers[MAX_CONTROLLERS];
-//static int           controllers_size = 0;
-
-//static struct controller_item        controllers;
-static struct list_head                controller_list;
+static struct net2280  *the_controller;

 static void usb_reset (struct net2280 *dev)
 {
@@ -1940,9 +1934,7 @@ static void ep0_start (struct net2280 *d
  */
 int usb_gadget_register_driver (struct usb_gadget_driver *driver)
 {
-       struct net2280          *dev = NULL;
-       struct controller_item  *controller_item;
-       struct list_head        *list_ptr;
+       struct net2280          *dev = the_controller;
        int                     retval;
        unsigned                i;

@@ -1956,49 +1948,41 @@ int usb_gadget_register_driver (struct u
                        || !driver->unbind
                        || !driver->setup)
                return -EINVAL;
+       if (!dev)
+               return -ENODEV;
+       if (dev->driver)
+               return -EBUSY;

-       list_ptr = &controller_list;
-       for (list_ptr = controller_list.next; list_ptr != &controller_list; list_ptr = list_ptr->next)
-       {
-               controller_item = list_entry(list_ptr, struct controller_item, list);
-               dev = controller_item->the_controller;
-
-               if (!dev)
-                       return -ENODEV;
-               if (dev->driver)
-                       return -EBUSY;
```

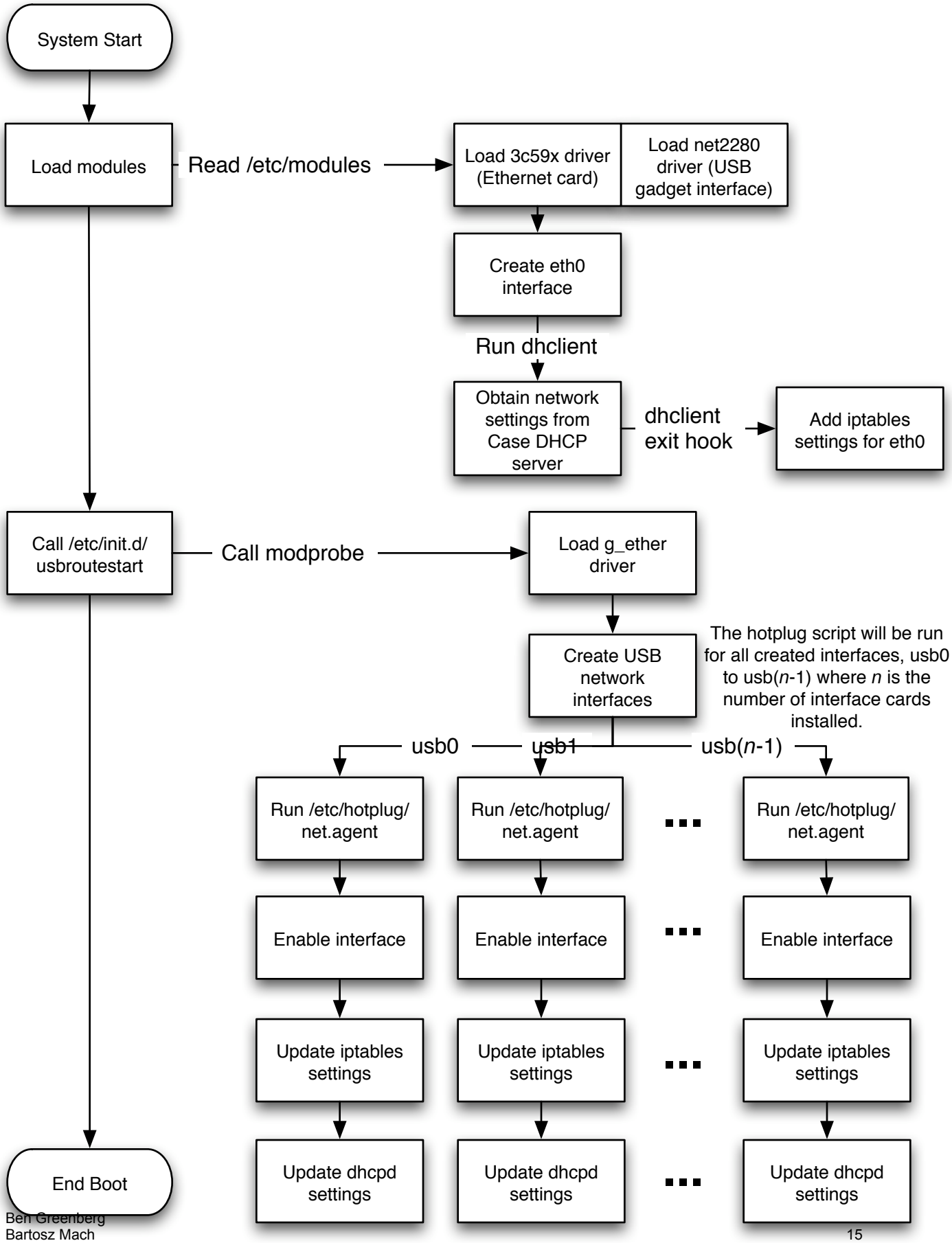
```
-
-               for (i = 0; i < 7; i++)
-                       dev->ep [i].irqs = 0;
+       for (i = 0; i < 7; i++)
+               dev->ep [i].irqs = 0;

-               /* hook up the driver ... */
-               driver->driver.bus = NULL;
-               dev->driver = driver;
-               dev->gadget.dev.driver = &driver->driver;
-               retval = driver->bind (&dev->gadget);
-               if (retval) {
-                       DEBUG (dev, "bind to driver %s --> %d\n",
-                                       driver->driver.name, retval);
-                       dev->driver = NULL;
-                       dev->gadget.dev.driver = NULL;
-                       return retval;
-               }
-
-               device_create_file (&dev->pdev->dev, &dev_attr_function);
-               device_create_file (&dev->pdev->dev, &dev_attr_queues);
-
-               /* ... then enable host detection and ep0; and we're ready
-                * for set_configuration as well as eventual disconnect.
-                */
-               net2280_led_active (dev, 1);
-               ep0_start (dev);
-
-               DEBUG (dev, "%s ready, usbctl %08x stdrsp %08x\n",
-                               driver->driver.name,
-                               readl (&dev->usb->usbctl),
-                               readl (&dev->usb->stdrsp));
-
+       /* hook up the driver ... */
+       driver->driver.bus = NULL;
+       dev->driver = driver;
+       dev->gadget.dev.driver = &driver->driver;
+       retval = driver->bind (&dev->gadget);
+       if (retval) {
+               DEBUG (dev, "bind to driver %s --> %d\n",
+                               driver->driver.name, retval);
+               dev->driver = NULL;
+               dev->gadget.dev.driver = NULL;
+               return retval;
        }
+
+       device_create_file (&dev->pdev->dev, &dev_attr_function);
+       device_create_file (&dev->pdev->dev, &dev_attr_queues);
+
+       /* ... then enable host detection and ep0; and we're ready
+        * for set_configuration as well as eventual disconnect.
+        */
+       net2280_led_active (dev, 1);
+       ep0_start (dev);
+
+       DEBUG (dev, "%s ready, usbctl %08x stdrsp %08x\n",
+                       driver->driver.name,
+                       readl (&dev->usb->usbctl),
+                       readl (&dev->usb->stdrsp));
+
        /* pci writes may still be posted */
        return 0;
 }
@@ -2032,38 +2016,27 @@ stop_activity (struct net2280 *dev, stru
```

```
 int usb_gadget_unregister_driver (struct usb_gadget_driver *driver)
 {
-       struct net2280  *dev = NULL;
-       struct list_head *list_ptr;
-       struct controller_item *controller_item;
+       struct net2280  *dev = the_controller;
        unsigned long   flags;

-       while (!list_empty(&controller_list))
-       {
-               list_ptr = controller_list.next;
-               controller_item = list_entry(list_ptr, struct controller_item, list);
-               dev = controller_item->the_controller;
-
-               if (!dev)
-                       return -ENODEV;
-               if (!driver || driver != dev->driver)
-                       return -EINVAL;
+       if (!dev)
+               return -ENODEV;
+       if (!driver || driver != dev->driver)
+               return -EINVAL;

-               spin_lock_irqsave (&dev->lock, flags);
-               stop_activity (dev, driver);
-               spin_unlock_irqrestore (&dev->lock, flags);
+       spin_lock_irqsave (&dev->lock, flags);
+       stop_activity (dev, driver);
+       spin_unlock_irqrestore (&dev->lock, flags);

-               driver->unbind (&dev->gadget);
-               dev->gadget.dev.driver = NULL;
-               dev->driver = NULL;
+       driver->unbind (&dev->gadget);
+       dev->gadget.dev.driver = NULL;
+       dev->driver = NULL;
+
+       net2280_led_active (dev, 0);
+       device_remove_file (&dev->pdev->dev, &dev_attr_function);
+       device_remove_file (&dev->pdev->dev, &dev_attr_queues);

-               net2280_led_active (dev, 0);
-               device_remove_file (&dev->pdev->dev, &dev_attr_function);
-               device_remove_file (&dev->pdev->dev, &dev_attr_queues);
-
-               DEBUG (dev, "unregistered driver '%s'\n", driver->driver.name);
-               list_del(list_ptr);
-               kfree(controller_item);
-       }
+       DEBUG (dev, "unregistered driver '%s'\n", driver->driver.name);
        return 0;
 }
 EXPORT_SYMBOL (usb_gadget_unregister_driver);
@@ -2738,7 +2711,7 @@ static void net2280_remove (struct pci_d

        INFO (dev, "unbind\n");

-//     the_controller = NULL;
+       the_controller = NULL;
 }

 /* wrap this driver around the specified device, but
@@ -2748,22 +2721,18 @@ static void net2280_remove (struct pci_d
 static int net2280_probe (struct pci_dev *pdev, const struct pci_device_id *id)
 {
```

```
        struct net2280          *dev;
-       struct controller_item  *controller_item_ptr;
        unsigned long           resource, len;
        void                    *base = NULL;
        int                     retval, i;
        char                    buf [8], *bufp;
-

        /* if you want to support more than one controller in a system,
         * usb_gadget_driver_{register,unregister}() must change.
         */
-       /*
        if (the_controller) {
                dev_warn (&pdev->dev, "ignoring\n");
                return -EBUSY;
        }
-       */

        /* alloc, and start init */
        dev = kmalloc (sizeof *dev, SLAB_KERNEL);
@@ -2891,18 +2860,7 @@ static int net2280_probe (struct pci_dev
                        use_dma
                                ? (use_dma_chaining ? "chaining" : "enabled")
                                : "disabled");
-       //the_controller = dev;
-
-       /* Add this device to the controller array */
-       controller_item_ptr = kmalloc(sizeof(struct controller_item), SLAB_KERNEL);
-       if (controller_item_ptr == NULL)
-       {
-               ERROR(dev, "Unable to allocate memory\n");
-               retval = -ENOMEM;
-               goto done;
-       }
-       controller_item_ptr->the_controller = dev;
-       list_add_tail(&controller_item_ptr->list, &controller_list);
+       the_controller = dev;

        device_register (&dev->gadget.dev);
        device_create_file (&pdev->dev, &dev_attr_registers);
@@ -2949,9 +2907,6 @@ static int __init init (void)
 {
        if (!use_dma)
                use_dma_chaining = 0;
-
-       INIT_LIST_HEAD(&controller_list);
-
        return pci_module_init (&net2280_pci_driver);
 }
 module_init (init);
```

APPENDIX III

Automation Boot Process Flowchart



APPENDIX IV, Part A: Contents of /etc/init.d/usbroutestart on sasha

```
rm /etc/usbroute/last_addr
rm /etc/usbroute/interfaces
modprobe g_ether
cat /etc/usbroute/dhcp > /etc/dhcpd.conf
echo -e "\n\n" >> /etc/dhcpd.conf
```

APPENDIX IV, Part B: Contents of /etc/dhcp-exit-hooks

```
#init nat

iptables -t nat -D POSTROUTING 1        #delete the rule
ipaddr=`/sbin/ifconfig eth0 | grep "inet addr:" | sed 's/.*addr:\([0-9][0
-9]*\.[0-9][0-9]*\.[0-9][0-9]*\.[0-9][0-9]*\).*/\1/1'`
iptables -t nat -I POSTROUTING 1 -o eth0 -j SNAT --to $ipaddr
```

Appendix IV,Part C: Relevant portion of /etc/hotplug/net.agent

```
case $ACTION in
add|register)
    case $INTERFACE in
        usb*)
        DHCPDCONF=/etc/dhcpd.conf

        while [ -f /var/lock/usbroute.lock ]
        do
                sleep .1s
#               echo $INTERFACE: sleep >> /usblog
        done
        touch /var/lock/usbroute.lock
#       echo $INTERFACE: done sleeping >> /usblog
        start_addr=`head -n1 /etc/usbroute/start_addr`
        if [ ! -f /etc/usbroute/last_addr ]
        then
                ipaddr=$start_addr
                ip1=`cat /etc/usbroute/start_addr | sed 's/\([0-9][0
-9]*\)\.[0-9][0-9]*\.[0-9][0-9]*\.[0-9][0-9]*.*/\1/1'`
                ip2=`cat /etc/usbroute/start_addr | sed 's/[0-9][0
-9]*\.\([0-9][0-9]*\)\.[0-9][0-9]*\.[0-9][0-9]*.*/\1/1'`
                ip3=`cat /etc/usbroute/start_addr | sed 's/[0-9][0-9]*\.[0
-9][0-9]*\.\([0-9][0-9]*\)\.[0-9][0-9]*.*/\1/1'`
                ip4=`cat /etc/usbroute/start_addr | sed 's/[0-9][0-9]*\.[0
-9][0-9]*\.[0-9][0-9]*\.\([0-9][0-9]*\).*/\1/1'`
        #       ip3=`expr $ip3 + 1`
        #       nextaddr="$ip1.$ip2.$ip3.$ip4"
#               echo ip1=$ip1, ip2=$ip2, ip3 = $ip3, ip4=$ip4 > /firsttime
                echo First Addr >> /etc/usbroute/log
                echo $ipaddr > /etc/usbroute/last_addr
        else
                ip1=`cat /etc/usbroute/last_addr | sed 's/\([0-9][0
-9]*\)\.[0-9][0-9]*\.[0-9][0-9]*\.[0-9][0-9]*.*/\1/1'`
                ip2=`cat /etc/usbroute/last_addr | sed 's/[0-9][0
-9]*\.\([0-9][0-9]*\)\.[0-9][0-9]*\.[0-9][0-9]*.*/\1/1'`
                ip3=`cat /etc/usbroute/last_addr | sed 's/[0-9][0-9]*\.[0
-9][0-9]*\.\([0-9][0-9]*\)\.[0-9][0-9]*.*/\1/1'`
                ip4=`cat /etc/usbroute/last_addr | sed 's/[0-9][0-9]*\.[0
-9][0-9]*\.[0-9][0-9]*\.\([0-9][0-9]*\).*/\1/1'`
                ip3=`expr $ip3 + 1`
                ipaddr="$ip1.$ip2.$ip3.$ip4"
                echo $ipaddr > /etc/usbroute/last_addr
        fi
        sleep 1s
        echo First Addr 2 >> /etc/usbroute/log
        echo "subnet $ip1.$ip2.$ip3.0 netmask 255.255.255.0" { >>
```

```
$DHCPDCONF
        echo "range $ip1.$ip2.$ip3.2 $ip1.$ip2.$ip3.20;" >> $DHCPDCONF
        echo "option routers $ip1.$ip2.$ip3.1;" >> $DHCPDCONF
        echo -e "}\n" >> $DHCPDCONF

        /sbin/ifconfig $INTERFACE $ipaddr netmask 255.255.255.0

        iptables -t nat -D PREROUTING 1
        iptables -t nat -I PREROUTING -i eth0 -j DNAT --to $start_addr-
$ip1.$ip2.$ip3.20
        echo -n "$INTERFACE " >> /etc/usbroute/interfaces
        DHCPINTERFACES=`head -n1 /etc/usbroute/interfaces`
        echo INTERFACES=\"$DHCPINTERFACES\" > /etc/default/dhcp
        /etc/init.d/dhcp restart
        rm /var/lock/usbroute.lock
        echo $INTERFACE END >> /etc/usbroute/log
        ;;
```