The GNU/Linux "usbnet" Driver Framework

David Brownell
<dbrownell@users.sourceforge.net>

Last Modified: 27 September 2005

USB is a general purpose host-to-device (master-to-slave) I/O bus protocol. It can easily carry network traffic, multiplexing it along with all the other bus traffic. This can be done directly, or with one of the many widely available USB-to-network adapter products for networks like Ethernet, ATM, DSL, POTS, ISDN, and cable TV. There are several USB class standards for such adapters, and many proprietary approaches too.

This web page describes how to use the Linux *usbnet* driver, CONFIG_USB_USBNET in most Linux 2.4 (or later) kernels. This driver originally (2.4.early) focussed only on supporting less conventional types of USB networking devices. In current Linux it's now a generalized core, supporting several kinds of network devices running under Linux with "minidrivers", which are separate modules that can be as small as a pair of static data tables.

- One type is a host-to-host network cable. Those are good to understand, since some other devices described here need to be administered like those cables; Linux bridging is a useful tool to make those two-node networks more manageable, and Windows XP includes this functionality too.
- Linux PDAs, and other embedded systems like DOCSIS cable modems, are much the same. They act as Hosts in the networking sense while they are "devices" in the USB sense, so they behave like the other end of a host-to-host cable. All that's needed is the USB-IF *Communications Device Class* (CDC) "Ethernet" class, or a simplified variant if the hardware can't implement CDC to spec. (Unless you listen to Microsoft, who will tell you not to use such vendor-neutral protocols. They think a complex and poorly documented protocol they defined, RNDIS, is better for them.)
- Traditional Ethernet Adapters such as the high-speed (USB 2.0) ASIX 8817x based products.

It makes sense to have a common driver core because only a handful of control and setup operations really need product- or class-specific code. Most of the driver handles i/o queues and USB faults, which can easily be product-neutral. And for some reason, vendors seem to dislike using standard framing in their Windows drivers, so many minidrivers need to wrap a technically-unnecessary

layer of headers around Ethernet packets for better interoperability.

Another approach to using IP over USB is to make the device look like a serial line or telecommunications modem, and then run PPP over those protocols. This document doesn't address those approaches, used sometimes with USB drivers such as *cdc_acm*, *usb-serial*, and with adapters to IRDA or BlueTooth stacks.

Devices that Work with "usbnet"

Here's an incomplete list of devices that the *usbnet* driver works with. It's incomplete because Linux doesn't need to know anything specific about products (correctly) implementing the CDC Ethernet class specification. It's also incomplete because products that use specialized chips (or which reuse other product designs) may be repackaged without changing how they work. Two devices with different brand labeling (on the box and device) may look identical at the USB level. That often removes the need for driver updates, even for lower end devices that don't support the standard USB-IF CDC Ethernet class.

USB Host-to-Host Cables

Note that before Linux 2.6.14, the minidrivers were not split out into their own modules. With older kernels, just "modprobe usbnet" to get everything; newer kernels modprobe the minidriver, which depends on usbnet to do all the USB-specific work.

Device	Minidriver	Notes
Advance USBNET	*cdc_subset*	(eTEK design)
ALi M5632 (chip)	*cdc_subset*	USB 2.0 high speed; used in various products. The current Linux driver does NOT interoperate with the Win32 "USB Virtual Network Adapter" driver from ALi (now Uli). (That ALI

		code seems to need a seven byte header that nobody's taught Linux to use.)
AnchorChips 2720 (chip)	*cdc_subset*	used in various products
BAFO DirectLinq	*plusb*	(uses PL-2301)
Belkin USB DirectConnect	*cdc_subset*	(eTEK design)
eTEK (design)	*cdc_subset*	used in various products
GeneSys GL620USB-A	*gl620a*	used in various products, including some motherboards and at least one BAFO product. *the half-duplex GL620USB is NOT supported! products using it include the Inland Pro USB Quick Link*
Jaton USB ConNET	*plusb*	(uses PL-2302)
LapLink Gold	*net1080*	(uses NetChip 1080)
NetChip 1080 (chip)	*net1080*	used in various products
Prolific PL-2301/2302 (chips)	*plusb*	used in various products; these two chips seem to be all but identical
Xircom PGUNET	*cdc_subset*	(uses AnchorChips 2720)

Of those, support for the Prolific based devices is the least robust. (Likely better status handshaking would help a lot.) Seek out other options if you can. I've had the best luck with the designs used by Belkin and NetChip.

There are other USB 2.0 high speed (480 Mbit/sec) host-to-host link products available too, with data transfer speeds that comfortably break 100BaseT speed limits. A single USB 2.0 link should handle even full duplex 100BaseT switches (100 MBit/sec in each direction) with capacity to spare, and more capable EHCI controllers should handle several such links.

Smart USB Peripherals ("Gadgets")

There's another interesting case that the *usbnet* driver handles. You can connect your host (PC) to certain USB-enabled PDAs, cell phones, cable modems, or to any gadget that's very smart (maybe smart enough to embed Linux!) and uses one of the flavors of USB networking that this driver supports. Although you can program your PDA, it's not really a USB "host" (master), it's still "device" (slave). (Unless it supports USB OTG, a technology that's not yet widely available.) If that device talks like one of the host-to-host adapters listed above, a host won't know it's talking to a PDA that runs Linux directly.

Device	Minidriver	Notes
CDC Ethernet devices (including some cable modems)	*cdc_ether*	In Linux kernel 2.6, "usbnet" can support this standard USB-IF class specification, replacing the older "CDCEther" or "cdc-ether" driver. Devices that embed Linux will often support this, using the Linux-USB Gadget driver stack and the "g_ether" driver, on hardware such as the NetChip 2280 (USB 2.0 high speed).
Epson based	*cdc_subset*	Epson provides example firmware.

devices		
PXA-2xx based PDAs	*cdc_subset* or *rndis*	The PXA-250 and PXA-255 are used in successors to SA-1100 based products; there are other similar PXA-based products. "usbnet" talks to PDAs running standard ARM-Linux kernels with the "usb-eth" or "g_ether" drivers.
SA-1100 based PDAs	*cdc_subset*	found in iPAQ, YOPY, and other PDAs using standard ARM Linux kernels; also see the resources at handhelds.org
BLOB boot loader	*cdc_subset*	Some of the boot loaders used with embedded systems allow the OS to be downloaded over USB using TFTP. This is a big help when developing systems which don't have many I/O ports. One such boot loader is BLOB.
Sharp Zaurus SL-5000D, SL-5500, SL-5600, SL-6000, A-300, B-500, C-700,	*cdc_subset* or *rndis*	These SA-1100 (or PXA-25x) based products don't use standard ARM Linux kernels, but usbnet talks to the gadget-side stack they use. *Do NOT add the "usb**d**net"*

C-750, C-760, C-860...		*driver,* just get the latest "usbnet" patch if you have one of the newest Zaurus models.
RNDIS based devices	*rndis*	Recent Linux kernels (2.6.14 and later) include experimental support for the RNDIS protocol. Since that's the only USB networking protocol built into MS-Windows, it's interesting even though it's a proprietary protocol with only incomplete public documentation. The driver is young, but it seems to work with at least some Nokia cell phones.

The cable devices perform a master-to-slave conversion and a slave-to-master conversion ... but these kinds of gadgets don't need the slave-to-master conversion, they're natural slaves! The PDA side initialization is a bit different, but the host side initialization (and most of the other information provided here) stays the same. And of course, the USB-enabled gadget could be running some other OS, maybe an RTOS; it doesn't need to run Linux. It only needs to wrap network packets in one of a few ways, without many demands for control handshaking.

Ethernet Adapters

In addition to the "software emulated" adapter model used in smart peripherals, there are also single-purpose adapters using real hardware. In particular, the *ASIX 8817x* chips are used in a wide variety of high speed (480 Mbit/s) capable 10/100 Ethernet adapters. There are also 10/100/1000 versions.

Device	Minidriver	Notes
ASIX 88172, ATEN UC210T, D-Link DUB-E100, Hawking UF200, Linksys USB200M, Netgear FA120, Intellinet, ST Lab USB Ethernet, TrendNet TU2-ET100, ...	*asix*	All these are based on the same core hardware. This originally used separate driver, but then it merged with "usbnet". Later kernels split out this minidriver into its own module.

There are also Linux-USB device drivers for ethernet adapters that don't use this framework.

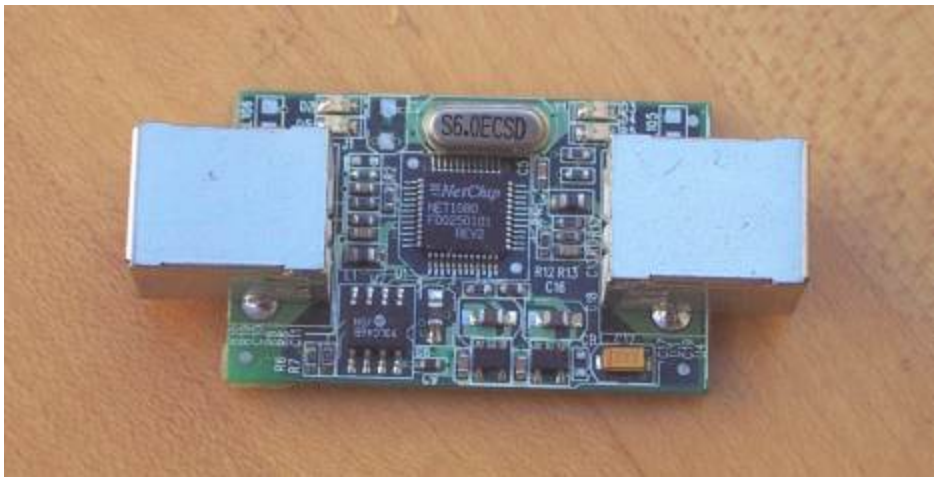
What are these "Host-to-Host" Cables?

These devices are unlike most other USB devices you'll see. That's because they connect to two different hosts, not just one. They use "A" connectors (rectangular) to connect to each host, and sometimes have two "B" connectors (squarish) going into the device. *The only way you can legally connect one host to another is through one of these special devices* which accepts commands from two hosts at the same time.

Sometimes they're sold as special "adapter cables" like this one (which happens to use a Prolific PL-2301 chip). Notice that this has two "A" connectors built in, as well as the special device (inside the molded plastic; use your X-Ray goggles).

Another way to package these devices takes a bit more money to provide two "B" connectors. They're hooked up using standard "A-to-B" cables, usually provided with the device, so these again connect to each host using an "A" connector.

What do these devices look like inside? You can often open them up to look. The Belkin device shown above has an AVR microcontroller and two general purpose USB interface chips, but most other such devices take a lower cost approach using specialized chips. Here's what one looks like. Like the Belkin device above, this one includes LEDs to show data traffic and errors; you'll have to imagine them flashing:



Yes, you may occasionally see "A-to-A" cables for sale; *don't waste your money buying them*. Those cables are forbidden in USB, since the electrical connections don't make any sense at all. (If you try to use one, you might even short out your USB electronics and so need to buy a new system.) Basically, they're missing the extra electronics shown above, which is necessary to let a USB "master" (host) talk to another one, by making both talk through a USB "slave" (device). (There is one time you may need such cables: when you're working with a hardware development system where the single USB port can be configured &emdash; for development only! &emdash; in either master or slave roles.)

Link Level Framing

One consequence of supporting multiple devices is that the "usbnet" driver supports several different link level framing solutions for IEEE 802 packets over USB. (Another is that most bugfixes automatically benefit even the devices they weren't seen with.)

Starting with Linux kernel 2.6.14, the framing is handled entirely by the minidriver (usually a module); the core usbnet framework doesn't know about framing rules any more. Standard Linux kernels supported them in roughly this order:

- Simple Framing ... this is the CDC Ethernet framing, but it uses a one byte packet where CDC uses a zero length packet. (Some systems aren't robust with zero length packets.) This is the default framing, used with most devices.
- NetChip 1080 ... framing was defined by NetChip, and uses a header and trailer to help recover from FIFO level problems.
- GeneSys ... framing was defined by GeneSys, packing multiple Ethernet frames in a single USB transfer. (On some USB hosts, queueing USB

transfers is more expensive than the packet merging.)
- Zaurus ... this adds an Ethernet CRC to the end of packets, accomodating bugs that can corrupt USB traffic. Note that this *can't interoperate with standard CDC* Ethernet devices, although early Zaurus models wrongly advertise themselves on USB as CDC Ethernet devices. (Newer models wrongly advertise themselvs as CDC MDLM devices, but that's much less of a problem.)
- RNDIS ... Microsoft's RNDIS packs several Ethernet frames into a single USB transfer, using its own scheme. (Again, the issue seems to be that queuing is pointlessly expensive compared to the extra copy needed to pack frames.)

As yet, there is no Linux support for the new CDC "Ethernet Emulation Model"; other than supporting that link management protocol, there's no end-user value in defining yet another framing scheme. Use the "Simple" framing for new devices; the only good reason to use anything else is to work around hardware problems, when for any reason that hardware can't be changed.

Connecting to Linux at the USB Level

When you connect a usbnet device to a Linux host, it normally issues a USB hotplug event, which will ensure that the *usbnet* driver is active. In most GNU/Linux distributions you shouldn't even notice whether the driver needed loading. It should just initialize, so that you can immediately use the device as a network interface. If it doesn't, then you probably didn't configure this driver (or its modular form) into your kernel build. To fix that, rebuild and reinstall as appropriate; at this time you might also want to upgrade to a recent kernel.

Once that driver starts using that USB device, you'll notice a message like this in your syslog files, announcing the presence of a new *usb0* (or *usb1*, *usb2*, etc) network interface that you can use with *ifconfig* and similar network tools.

```
usb0: register usbnet usb-00:02.0-1.3, Belkin, eTEK, or compatible, c2:91:27:cb:d7:29
```

Of course, that might also say "SA-1100 Linux Device" or "Zaurus" if you are using such a PDA, or identify some other hardware. Ethernet adapters, or devices that run like them (many cable modems), would normally use names like "eth0".

Instead of *usb-00:02.0-1.3* (which will likely be different on your machine)

older kernels may show a usbfs name like "005/003". The "usbfs" style device naming has problems since it's not "stable": it can easily change over time. The newer style, output of the *usb_make_path()* function in the kernel, is stable; it won't change until you re-cable your tree of USB devices (or perhaps move your USB adapter card). Stable names let you build systems with logic like "since this link goes to the test network, we will firewall it carefully when we bring it up".

Connecting to GNU/Linux at the Network Level

After a the driver binds to the device, the new interface causes a network hotplug event reporting that a new network interface has been registered. At this time, the interface might look like this through "ifconfig" or "ip":

```
sh# ifconfig usb0
usb0      Link encap:Ethernet  HWaddr 3C:13:AF:8C:03:9D
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
sh# ip addr show usb0
31: usb0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop qlen 100
    link/ether 3c:13:af:8c:03:9d brd ff:ff:ff:ff:ff:ff
sh#
```

That is, this appears like a normal Ethernet link, not like a point-to-point link. (The older *plusb* driver, found in 2.2 Linux, was set up as a point to point link.) That's done for several reasons, most of which boil down to making it easier to bridge these links together. Linux has a fully featured IEEE 802.1 bridging module (CONFIG_BRIDGE) with full spanning tree support as supported by normal Ethernet bridges. So it's easy to configure bridging; a laptop might connect to a desktop with a USB networking cable, and then to the local LAN through a bridge. With that working, DHCP or ZCIP are easy to use from the laptop. (The link level address will usually not be one from a manufacturer's ID prom, except on higher end devices. Instead, it will have been "locally assigned" during initialization of the "usbnet" driver.)

When you get these network hotplug events, you basically want to configure it. As a standard network link, you could just configure it for use with IPv4. Or, you can configure it to work with IPv6 . To bring the interface up by hand, you might type:

```
sh# ifconfig usb0 10.0.2.5
```

```
usb0      Link encap:Ethernet  HWaddr 3C:13:AF:8C:03:9D
          inet addr:10.0.2.5  Bcast:10.255.255.255  Mask:255.0.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
sh# ip addr show usb0
31: usb0:  mtu 1500 qdisc pfifo_fast qlen 100
    link/ether 3c:13:af:8c:03:9d brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.5/8 brd 10.255.255.255 scope global usb0
sh#
```

In general, you'd rather automate such things. At this writing, that's not easily done except for specific GNU/Linux distributions (or families of them). The standard hotplug distribution works for everything that supports the `ifup` command, but that command unfortunately requires some pre-configuration.

You should usually set the netmask to 255.255.255.0, for a "class C" (or /24) subnet. Get the right network settings from your local network admin.

Ethtool

With ethtool version 1.5 or later, and recent enough version of the *usbnet* driver, you can get additional information from the driver. Different devices may have different information available; for example, link availability is not always known. Linux defines some standard interpretations for the "message level" bits, which are not widely used ... but this framework uses them for all its devices, letting you mask which messages will be seen. (Many messages won't be available unless debugging is enabled.)

```
sh# ethtool usb0
Settings for usb0:
        Current message level: 0x00000001 (1)
sh# ethtool -i usb0
driver: usbnet
version: 17-Jul-2002
firmware-version: Prolific PL-2301/PL-2302
bus-info: usb-00:02.0-1.2
sh#
sh# ethtool usb1
Settings for usb1:
        Current message level: 0x00000001 (1)
        Link detected: no
sh# ethtool -i usb1
driver: usbnet
version: 17-Jul-2002
```

```
firmware-version: NetChip TurboCONNECT
bus-info: usb-00:02.0-1.4
sh#
```

You might want to use stable bus-info values to figure out what network address to assign to a given link, if your routing configuration needs that. You can use *ip link set* ***usbN*** *name* ***newname*** or similar tools. (NOTE: the *nameif* tool is unfortunately not going to help here, since it assumes that that Ethernet addresses solve this problem. For dynamically assigned Ethernet addresses, that can't work; using "bus-info" is the appropriate solution.)

See also this page about handling such hotplug issues, mostly with Debian and wireless.)

Configuring RedHat or Mandrake Linux

These distributions have an `ifup` command that requires each device to be pre-configured, with a unique config file. If your device is very "ethernet-like" (named *ethN*) then your sysadmin tools will probably recognize them and help you set up the interface; else you'll edit system config files. The rest of these configuration instructions are oriented towards devices that are not very "ethernet-like".

You can preconfigure those tools, modify the system setup to automate more of the setup, or more typically do both. The preconfiguration uses files like `/etc/sysconfig/network-scripts/ifup-`*`eth0`*: you shouldn't have much trouble making modified versions for *usb0* and other USB links you may use.

The network model used in these explanations is the core of many such models that you will likely need to handle. Two systems are being directly connected. One is a "leaf" system with no other network connectivity (perhaps a laptop, PDA, or printer). The other is a "host" that sits on some LAN, and probably has Internet access. Those two systems connect through USB network links, and the configuration problem is making sure there is complete IP connectivity. You'll have to arrange naming and routing yourself, and this section shows how to set up using static IP addressing.

"Leaf" System Pre-Configuration

Here's how I set up one laptop, which expects to connect to any of several desktop machines, with a 10.0.1/24 "USB subnet". Similar setups can use DHCP.

```
sh# cat /etc/sysconfig/network-scripts/ifcfg-usb0
DEVICE=usb0
BOOTPROTO=static
IPADDR=10.0.1.2

# IPADDR=10.0.1.2/24 should imply this ...
# but its support code seems to be buggy.
BROADCAST=10.0.1.255
NETMASK=255.255.255.0
NETWORK=10.0.0.0

# this is likely to break sometime
GATEWAY=10.0.1.1

ONBOOT=yes
sh#
```

That uses a USB host-to-host cable. USB "gadgets" that embed Linux can work much the same, but they seldom run RedHat's Linux distribution. See the sections on PDAs and other USB gadgets or, for dynamic configuration, zeroconf.

"Host" System Pre-Configuration

Each desktop machine that leaf connects to is set up up like this (different IP and routing on each interface).

```
sh# cat /etc/sysconfig/network-scripts/ifcfg-usb0
DEVICE=usb0
BOOTPROTO=static
IPADDR=10.0.1.1

# IPADDR=10.0.1.1/24 should imply this ...
BROADCAST=10.0.1.255
NETMASK=255.255.255.0
NETWORK=10.0.0.0

ONBOOT=yes
sh#
```

That desktop machine is also set up to do IP level routing, since it's on a LAN that has Internet access. You probably don't want to administer routing machinery except when you're deploying some kind of firewall. *Be very careful about routing, particularly if you use multiple such links on a given desktop/host. A bridged configuration will be less error prone.*

Bridged "Host" Systems

If a host bridges every USB link it sees onto the main LAN, then only the link to the main LAN needs to be pre-configured. Less configuration means fewer important things can go wrong. It also eliminates the need to route a two-node subnet for each new USB network device, making network administrators happier with your choice of peripheral hardware.

You may be familiar with how bridging works with Microsoft Windows XP, when you connect your second network link. Only the tools and commands are very different on Linux hosts; most distributions for Linux don't yet provide a way to automatically set up your bridge that's as easy.

Linux kernels include a standard IEEE 802 (Ethernet) Bridging module, using the Spanning Tree Protocol (STP) to interoperate with commercial Ethernet bridge and switch products. Use the Linux kernel "bridge" module along with the "bridge-utils" package, which includes the important *brctl* command. The latest code is available at http://bridge.sourceforge.net. Most current Linux distributions include that package, though usually it's not in the standard software profile. (You might have noticed that the MS-Windows driver provided with most USB host-to-host cables implements a limited form of bridging. A key limitation is usually "no loops": they don't support STP.)

The first part of bridge setup makes a logical LAN during network startup. You'll need to do that by hand, since most sysadmin tools don't understand bridge configuration. (If you're using Ubuntu or Debian, you're lucky to have some decent examples of how to set up bridges as part of your "ifupdown" documentation. They're not GUI tools, but they're a better start than what's sketched here.) I modified the network startup code to bring up eth0 as the core of bridge, instead of calling "ifup". So now it (re)boots into the right configuration, but this setup won't play nicely with RedHat's tools. If your LAN uses DHCP or ZCIP for dynamic address assignment, change the last step appropriately.

```
#
# From a modified /etc/init.d/network script
# On a host with a permanently assigned IP address
#
if [ "$i" = "eth0" ]; then
        # "lan" gets the address that "eth0" would normally get
        # "eth0" must never have an IP address
        brctl addbr lan &&
        brctl addif lan eth0 &&
        ifconfig eth0 0.0.0.0 &&
        action $"Setup LAN bridge $i" ifconfig lan 10.0.0.1
        # bridge all zeroconf traffic through here
```

```
        ip route replace 169.254.0.0/16 dev lan
else
        action $"Bringing up interface $i" ./ifup $i boot
fi
```

The second part of bridge setup makes hotplugging add all USB interfaces to that bridge. As shown here, nothing happens if there's no "lan" bridge; so this change could go into config files on any system that might ever use a bridge called "lan" in this particular way. On such systems, you can connect any number of these devices and they'd be automatically bridged as soon as they connect.

```
#
# modify /etc/hotplug/net.agent to do this
#
    case $INTERFACE in
        usb*)
            brctl addif lan $INTERFACE && exec ifconfig $INTERFACE up
            ;;
        # and then the existing code :
        ppp*|ippp*|isdn*|plip*|lo*|irda*)
        ...
    esac
#
```

The bridge may cause a short delay (one document said thirty seconds) before you can access the new devices, and should quickly start forwarding packets.

Be careful using Bridged configurations with PDAs. They may not have unique Ethernet addresses. Among other things, that means that if there's more than one such PDA in use at your site, everyone who may be bridging one of them should override that non-unique address. On a Zaurus &emdash; unless you're using Linux 2.6 based OpenZaurus ROMs, which don't have this issue &emdash; that'd mean modifying */etc/hotplug/net.agent* to make it call *ifconfig usbd0 hw ether xx:xx:xx:xx:xx:xx* before it brings the interface up. (Set the local assignment bit, 0x02 in the first octet, to ensure it still uses a name like "usb0" not "eth0".) Host-to-host cables automatically use pairs of unique "locally assigned" IEEE 802 addresses, and do not cause such problems.

Zeroconf

The current zcip software works with recent versions of usbnet. (There's also a version of this in current versions of Busybox.) It partially supports the goal of

a fully hands-off user install experience, so that USB peripherals don't require any sysadmin attention at all during setup, even on networks without DHCP service. (Read more about the IETF Zero Configuration Networking working group.)

A "leaf" type system might use ZCIP something like this:

```
#
# modify /etc/hotplug/net.agent to add a new case:
#
    case $INTERFACE in
        usb*)
            # picks address on 169.254/16, reserved for zeroconf
            # as of zcip 0.4 this setup is INCOMPLETE since you
            # must set up bridging, routing, and naming separately
            exec zcip -i $INTERFACE
            ;;
        # and then the existing code :
        ppp*|ippp*|isdn*|plip*|lo*|irda*)
        ...
    esac
#
```

Since that setup doesn't use "ifup" to bring interfaces up, you don't need to manually set up each potential usb link.

When USB Devices run Linux, not USB Hosts

Any embedded Linux system could use this same IP-over-USB approach if it acts as a USB "Device", not a USB "Host". Many people will be familiar with PDAs running Linux, discussed later in this section, but the embedded system doesn't need to be a PDA. It could be a home gateway, or any other kind of device or gadget where embedding Linux can give your product an edge.

In fact, that embedded system doesn't even have to run Linux, even when the processor it uses could support it (32bits, maybe an MMU, and so on). Some of those systems will run a real time OS, and microcontrollers often use very specialized operating environments. The latest version of the *usbnet* driver include support for some firmware that Epson provided to help system-on-chip applications (using Epson SOCs) interoperate better with Linux.

However, if that system does run Linux you can use the new USB Gadget framework to develop drivers there. That framework comes with a CDC Ethernet driver, which is used in conjunction with a driver for the specific hardware involved. The "Ethernet Gadget" code can achieve dozens of

megabytes of transfer speed in both directions, if the rest of the system supports such rates.

Most versions of that gadget stack also support Win32 interoperation using RNDIS; see linux/Documentation/usb/linux.inf and use the drivers bundled into XP by Microsoft. You should be able to use "usbnet" to talk to these gadgets from Linux hosts, and its device side acts much like the iPaq scenario described here. (Except that the interface name is likely *usb0* instead of *usbf* or *usbd0*.)

NOTE for RNDIS users: *MS-Windows has problems with its USB and RNDIS stacks, which can not be worked around by Linux peripherals.* Symptoms include at one extreme a blue screen (panic), to stopping communication after a while, to (the mildest failure) just a temporary lockup that goes away after a while. Among other things, it seems that MS-Windows does so much work when hooking up a new device that it's easy for one thing to go wrong, which can sometimes completely lock up the USB port to which you connect the device. You know these are bugs in MS-Windows because those things aren't allowed to happen no matter what the external USB device does. Unfortunately we can't expect such bugs to get fixed by Microsoft. (Another example of a clear bug in the MSFT code: when hooking the host link up to the built-in bridging on XP, it disables all use of multicast and broadcast packets, which are fundamental to operation of bridges.)

Most Linux PDAs: iPaq, Yopy, ...

StrongARM SA-1100 based PDAs running GNU/Linux (like the *Yopy*, or many *iPaq*s once you replace that other OS with Linux) will network through the *usbnet* host side driver. They get this by using a mainstream ARM kernel such as *2.6.18*, or perhaps the handhelds.org kernel distribution. (And maybe changing vendor and product IDs.) The CONFIG_SA1100_USB_NETLINK option enables the *usb-eth* driver inside that PDA, which talks to *usbnet* on the host.

Similar support can be provided for essentially any other "device side" Linux, including XScale PXA-25x based PDAs or other embedded systems. In particular, quite a lot of ARM chips have direct support in Linux 2.6 kernels.

The host side initialization in those cases is exactly as shown earlier, since the host uses the "usbnet" driver. The kernel in the PDA (or whatever embedded Linux device you're working with) uses a slightly different driver. For the *www.handhelds.org* kernels you'll use the interface name *usbf* (for USB Function?). See this wiki page for more information on how to set up your iPAQ (Yopy works the same) to talk to a USB Host running Linux (like current desktop or laptop machines).

Sharp Zaurus PDAs

At the same time the Linux community was doing the work above in public, Zaurus SL-5000D development was being done behind closed doors. The result was a second driver for everything mentioned above ... and some confusion. There's an incompatible derivative of "usbnet", called *usbdnet* (just an added 'd'), which expects to talk to an *eth-fd* driver (instead of "usb-eth") inside your Zaurus.

Those two Zaurus-specific drivers use nonstandard framing for Ethernet over USB, although the "eth-fd" driver enumerates as if it were conformant with the CDC Ethernet specification. (Newer versions break conformance in different ways, and claim to conform to the CDC MDLM specification.) That means standard CDC Ethernet drivers need to have a way to blacklist Zaurus products, since they are incompatible with the protocol standard they advertise. Zaurus framing adds an Ethernet CRC to the normal packet framing (explicitly disallowed by the CDC specification) as a partial work around for some DMA problems. The ARM Linux "usb-eth" driver and SA-1100 USB stack haven't needed such nonstandard changes. Another issue is that the two Ethernet addresses advertised by "eth-fd" don't seem to be unique, so that using them for all of the PDAs in a workgroup can be problematic.

In late October 2002 a patch was submitted to teach "usbnet" how to use the current Zaurus-specific protocol. The "usbnet" driver now works with all current Zaurus models, including the SL-5000D, SL-5500, SL-5600, C-700, and C-750/C-760. Current 2.6 kernels have it (2.5.45 on), as do 2.4.21 and later kernels. That's the preferred solution for Zaurus interoperability.

If you use a standard ARM Linux 2.6 kernel, running *g_ether* (or an old 2.4 kernel using *usb-eth*), the question shouldn't come up since "usbnet" has interoperated with "usb-eth" since about the Linux 2.4.10 release. The problem only comes up with code derived from that Zaurus work.

Note that until a patch to the *CDCEther* driver (on Linux 2.4 only) is available, preventing it from talking to these Zaurus products, you might need to add that driver to */etc/hotplug/blacklist* to prevent confusion like having your Zaurus appear as *eth0* and then not work right (because of the CRC). Some users have also found they need to shrink the mtu on the Zaurus, with *ifconfig usb0 mtu 1000*.

Until that updated "usbnet" starts to be more widely available, you'll want to read something like this SL-5000 HOWTO talking about how to do this with the original Zaurus software, and where to get the kernel patches you'll need if you want to recreate your own kernel. (Or, use the *www.handhelds.org* kernel tree, Opie, or openzaurus.org.)

Copyright and Licensing