Building a wireless access point on Linux

Custom solution offers flexibility and customizability -- and a chance to learn

Peter Seebach

July 22, 2003

When the ability to write and modify your own management software is the main objective, a custom-built wireless access point is the way to go. Take a look at what's involved in building a wireless bridge using Linux, including software and hardware considerations.

In this article, I'll take you through the process of building a wireless access point running Linux. I won't cover every last line of code, every intermediate step, or every detail of hardware; that would take a book and would be obsolete by the time you read it. The goal is to show you what kinds of concerns and pitfalls you'll face should you want to do this. For this piece, we build the access point to operate as a bridge; simply forwarding packets between the wireless network and a local ethernet. This allows wireless devices to simply be turned on and attached using your existing network -- no new configuration, no special routing.

Why bother?

Maybe I should start with "Why not to bother." It's pretty clear that buying an off-the-shelf device is a lot cheaper than building a custom box. The hardware that went into this ran me around $400, and that's not including the possible value of a few hours of my time. Having said that, there are two good reasons to build your own wireless access point.

The first is flexibility and customizability. Want a firewall? No problem. Custom routing? NAT? All easily managed. Custom Web-based configuration? Half an hour's work with CGI scripts. You can add traffic graphing if you like. That off-the-shelf box may have a special Windows-only configuration tool, so it won't be possible to ssh in and change your settings. You won't be able to run your weblog off of the Web server. If there's a problem, you have to wait for a "firmware upgrade" -- which may or may not address your problem.

The other -- and perhaps more compelling -- reason is that it's fun, and it's a good way to learn about the issues you may face running an embedded Linux system. Think of this as a hobbyist's test project, and if you come up with a killer app for it, you might just be able to make a profit anyway.
Requirements
The computer I used for this project is a Soekris net4521 embedded system. The CPU is a 133Mhz 486-class processor; plenty of power. The OS gets loaded on a CompactFlash (CF) card. I needed a wireless adapter; I grabbed a cheap LinkSys WPC11, based on the Intersil Prism 2 chipset. Similar systems would work as well.

To follow the whole thing, you'll also need another Linux box, to format and configure the flash card. A laptop is ideal, but anything with a USB port should work, if you have a CF reader that works under Linux.

As for software, there are existing packages that come pretty close to working out of the box, or you could start from scratch with an old set of install CDs and do it the hard way. For this article, I started with Pebble Linux, a small distribution that comes close to meeting my needs right off the bat. However, there's plenty of fun to be had tweaking settings and maybe adding a few custom features. From here on in, I'll be assuming you've got basic experience using Linux: editing files, running commands, and maybe even configuring a simple network. If you don't, your friendly neighborhood Linux hacker will probably walk you through this project for the price of a pizza, and explain what's going on if you include extra cheese.

For this project I used 802.11b wireless, the standard used by Apple's original Airport network and the most widely-available wireless standard. Theoretical bandwidth is 11Mbps, or just a tad faster than slow ethernet. The practical bandwidth is good enough to keep up with cheap broadband. The reception range is variable, but most people can reach a back yard or a neighboring apartment.

Project plan
This plan is broken into phases; at the end of each phase, there are concrete, testable, results. Each phase should be manageable within an hour or so of playing around.

The phases are:

- **Initial setup:** We get the embedded system to boot and to offer a console login prompt.
- **Network setup:** We get the network up and running, so the machine can actually be used as a functioning access point.
- **Tweaking:** We start on the long road to changing and adapting our machine's functionality.

Initial setup
Obtain an embedded system. I used the Soekris Net4521, but other systems, such as the Technologic TS-5500, are probably about the same experience. If you're desperate, you can probably use an old laptop! The important part, for now, is a PCMCIA slot and a disk you can format and configure from another machine. Embedded systems mostly use CF disks as their primary drives.

You'll also need a host system. A laptop running Linux will work, as will a desktop running Linux with a supported CF drive. Being perverse at heart, I used an already-installed embedded system with a USB CF drive, but it really doesn't make any difference.
I used a 128MB CF card for this project. That may seem huge -- and it is -- but it's always nice to have some spare room. The distribution in question (Pebble Linux) fits reasonably well even in a 64MB card. This spare room will be used later to install extra toys.

Once your hardware is together, boot your host system. Plug in a card reader with your 128MB CF card in it. The card should show up as sda; test this by running:

```
fdisk /dev/sda
```

You should see a prompt.

Now, partition the disk. Enter `p` to print the partition table. If your card hasn't been used for this before, it will probably show a single DOS partition. Delete it by entering `d`, and enter the partition number when prompted. Create a new partition, with the `n` command; the default range should be fine. Mark the partition as active by entering `a`, and enter the partition number when prompted. Now, enter `w`. You have now partitioned the drive to have a single Linux partition on it.

If you're using a PCMCIA-to-CF adapter in a laptop, your card will probably show up using the `hd` driver rather than `sd`. That's fine; just use the right name for it.

Next, format the disk. The command to do this is `mke2fs /dev/sda1`. Once it's formatted, mount it.

Now, extract the Pebble distribution. Pebble's installer assumes you'll have unpacked on another disk and will then copy over to the new disk. If you don't have another disk handy on your target system, you'll have to edit the installer; you can just extract the archive on your target disk, then update the scripts locally on it. If you're using a disk other than one of the suggested ones, you need to make a new `lilo.conf` file for your target drive; just copy the `lilo.conf` for `hdc`, and replace the device name. If you're determined to do things by hand, you need to run `lilo`, make ssh host keys, and set a root password. You also probably need to configure in some extra kernel modules; likely sets for a couple of systems are provided in `/etc/modules.*`. For instance, `/etc/modules.net4521` is the set of modules to use with a Soekris Net4521 system.

One tricky thing here is that the disk you're targeting will likely be a different disk in the target system. The provided `lilo` configuration files reflect this; they specify a disk, but then specify "bios 0x80" to warn `lilo` that the disk will be the primary disk on the new system. Similarly, in your new `fstab` you'll want to make sure you refer to the right disk; in the typical embedded system, that will be `/dev/hda1`, not `/dev/sda1`. Finally, make sure you set up the right console port in `inittab`; for the Soekris box, the default console speed is 19200kbps, rather than the 9600 Pebble Linux shipped with.

Once this is all in place, you should have a CF card that will simply boot and come up on your target system. Once you have a login prompt, login and look around. There are some useful scripts provided with Pebble Linux to let you remount the root filesystem as either read-only or read/write. The default is to come up read-only -- very nice for a system that you may want to unplug a lot.
If you're on a network with DHCP, you'll probably find that your system has already come up with an IP address on its primary ethernet port. (On the Soekris Net4521, I plugged a cable into eth0.) This will save us some work in the next step.

**Network setup**

What we really want, for an access point, is a simple bridge; packets from one network show up on the other network, nothing special. The box isn't a router or a DHCP server or anything, it's just a bridge. It could, of course, be set up as a DHCP server or any of a dozen other things, but this is the easy way. A bridge will simply send packets from the wire to the wireless and back. That way, the wireless device looks for all intents as though it's simply attached to the physical network.

Pebble Linux is based on Debian, so that means playing around with `/etc/network/interfaces`. A quick Google search yields a sample set of commands to play with:

**Listing 1. Building bridges**

```bash
iface br0 inet dhcp
  bridge_ports eth0 wlan0
  pre-up ifconfig eth0 0.0.0.0 up
  pre-up ifconfig wlan0 0.0.0.0 up
  pre-up iwconfig wlan0 mode master
  pre-up brctl addbr br0
  pre-up brctl addif eth0
  pre-up brctl addif wlan0
  post-down ifconfig eth0 0.0.0.0 down
  post-down ifconfig wlan0 0.0.0.0 down
  post-down brctl delif br0 eth0
  post-down brctl delif br0 wlan0
  post-down brctl delbr br0
```

This looks intimidating, but it's really not bad at all. The interface itself is a bridge (br0), using dhcp for configuration, bridging between eth0 (the first ethernet) and wlan0 (our wireless card). We have a set of commands to run before the interface comes up (pre-up), and another set to run after it comes down (post-down).

In this case, we want to bring up the two interfaces we'll be bridging, put our wireless card in "master" mode (making it an access point), then build a bridge interface and attach those two interfaces to it. Once that's done, the dhcp qualifier makes our box obtain itself an IP address through dhcp. That's actually optional; the bridge doesn't need to have its own IP address, since it will transparently forward packets from one interface to the other. However, if it has its own IP address, we can use the access point itself over the network; this is less secure, but more fun to use for a demo.

When we shut the interface down, we turn off the ethernet ports, remove them from the bridge, and remove the bridge interface. We had to create the bridge interface before adding the ethernet ports to it, so now we remove them before deleting the bridge interface.

This procedure assumes the interfaces aren't being configured by any other mechanism, so we want to comment out other lines referring to wlan0 or eth0, although we do use `/etc/pcmcia/hostap_cs.conf` to set the ESSID (base station name) for our wireless card. We could do this in
the pre-up script for br0, except that, on my particular card, if you set the ESSID and WEP key too closely together, the card hangs for a while. (What WEP key? The one we'll add in a little bit.)

You might think you could just put "auto br0" in /etc/network/interfaces and have the interface configured automatically. Unfortunately, if your wireless card is a PCMCIA card, its interface won't be available when network interfaces are normally started. So, we'll do it later during boot. Create a file in /etc/rc2.d called "S99local." In it, put the following lines:

```
#!/bin/sh
ifup --force -v br0
```

Both options are, in fact, optional. The interface shouldn't be up right now, so we shouldn't need the `--force` option. The `-v` is just there so we can see the commands that are being run during configuration.

Now, save your changes and reboot. If everything went according to plan, you should now be able to attach to the wireless network we just made, network ID "test," run a DHCP client, and get DHCP service from your regular DHCP server. (If you don't have one, a static IP address in the netblock that your wired network uses will work just fine.)

Of course, there's a glaring flaw here: We're not using a WEP (Wired Equivalent Privacy) key. In theory, WEP is supposed to make your network secure; in practice, it will only slow down intruders a little bit. Lacking one isn't that big a deal if you don't mind random passers-by using your network. But, just in case you do care, you'll want to set a WEP key. We do that by changing the `iwconfig` line in the setup for br0 in /etc/network/interfaces to this:

```
pre-up iwconfig wlan0 mode master key s:blahblahblah1
```

This specifies a key as a 13-character (104-bit) string. Some network cards may not be able to support a 104-bit key; in that case, use a five-character (40-bit) key. If your wireless card hangs when you do this, try adding "pre-up sleep 5" before the line in /etc/network/interfaces that sets the wireless key.

At this point, we have a wireless access point using WEP and a read-only filesystem, so you can unplug it, plug it back in, and it will come up -- not as quickly as some commercial ones, maybe, but it will come back up, with no damage and no lost files.

**Tweaking**

Here's where this gets really fun. We can make this box do all sorts of things. The traditional example is a Web server; it's the kind of thing the media likes to portray as a giant liquid-cooled machine with lights and an ominous humming sound; but which you and I know is actually just a few megabytes of disk space and a nearly imperceptible load on the system. In this section, we'll add a couple of packages, including the Web server, using apt-get, and start running CGI scripts on our new local Web server.

Let's start by getting a list of current packages. Running `/usr/local/sbin/remountrw` will give us write access to local media. Next, we run `apt-get update`. This downloads a fairly large chunk of
data (which is why we needed a 128MB CF card) -- we're getting a list of the available packages for this version of Debian.

We're going to install the Apache Web server, which requires a little bit of forethought; we want to have a space for log files, and we want it in the read-write space on the temp disk. Make a new directory called /var/log/apache. Because /var/log is actually a symbolic link to /rw/var/log, it will be created in the right place. Next, we run apt-get install apache. Hit return at a few prompts and -- poof! -- you have a Web server. Go to it; your Web server will be at whatever IP address your box was assigned, and it will have the default Apache configuration Debian uses.

Unfortunately, the /rw filesystem is discarded on every reboot. If you want the log files to get created correctly next time, too, you'll have to modify the template for the /rw filesystem, which is the /ro directory. Go into /ro/var/log and make a directory called apache. Now, after a reboot, your Web server should be up and running. On my system, this brought me to 63MB in use -- probably too much to fit on a 64MB CF card.

What next? How about a CGI script? Go to /usr/lib/cgi-bin. You'll need to mount the filesystem read-write to create a script. Let's start with hello.cgi:

### Listing 2. Hello wireless world!

```bash
#!/usr/bin/perl -w
print <<EOH;
Content-Type: text/plain
Hello, world!
EOH
```

Save this, make it executable (mode 755), and try visiting /cgi-bin/hello.cgi on your Web server.

What else could we do? Anything we want. Once you've got a Web server and Perl, the world's your oyster. Let's add another script, called stats.cgi:

### Listing 3. Getting statistical

```bash
#!/bin/sh
cat <<EOF
Content-Type: text/html

<HTML><HEAD><TITLE>WAP Stats</TITLE></HEAD>
<BODY>
<H2>Stats:</H2>
<pre>
ifconfig -a
cat <<EOF
</pre>
</BODY>
</HTML>
EOF
```

Look! Traffic reporting... It's not well formatted, but who cares? It took 30 seconds to write.
Hmm... What if we want to review our configuration? Here's file.cgi:

**Listing 4. Just checking**

```sh
#!/bin/sh
cat <<EOF
Content-Type: text/plain
EOF
cat $1
```

To call this a security hole is to understate the point woefully. But it's a great demo of how insanely flexible our wireless access point is now. In fact, I used this script to get the other scripts off of the system; easier than using scp.

These examples aren't very in-depth; a detailed discussion of CGI programming is beyond the scope of this article. The point is just to give you a few starting ideas. You could host a weblog on a box like this, although you'd want to have some kind of secondary storage for the actual data files. Of course, you could mount a remote disk over NFS!

The embedded system I used (the Soekris Net4521) had more than 36MB of free memory and about 50MB of free disk space when I was done setting it up with the above. That's enough space for a lot of stuff, and it leaves enough CPU time free to run all sorts of small services.

**Conclusions**

This project is a lot easier than it probably sounded at first. The key here is building on other peoples' work. I'm building on Pebble. They're building on Debian. A lot of the hardest parts of this were already done. For instance, it could take a fair amount of testing and rebooting to figure out how to build a system to boot from a read-only filesystem. How many of us actually remember which files need to be modified during boot? For instance, `/etc/network/ifstate` is actually modified by the system. Log files are easy to remember, but other files can be harder to remember. Similarly, the installation process for getting minimal dependencies satisfied for an installation of Apache is handled automatically by the Debian package system.

Open source turned a project that once would have taken days or weeks into an idle afternoon's playing around. This makes hobbyist projects practical, but it also makes a good prototype; if you're planning to do an embedded system of some sort and want to make it an access point, this would be a good way to get some practice playing around with the software configuration long before your first hardware samples are ready.
Related topics

- For this project, the author used a custom embedded system, the Soekris Net4521.
- This O'Reilly Network article takes a look at 802.11b Tips, Tricks, and Facts.
- The IEEE 802 LAN/MAN Standards Committee Web pages provide more detailed information on the protocols and standards involved than you will ever need in one lifetime.
- If you're feeling inventive, get inspiration and more background on the history of wireful and wireless communication and networking from TelecomWriting.com.