Build your own Wireless Access Point

Erik Andersen
Codepoet Consulting
andersen@codepoet.org

Abstract

This presentation will cover the software, tools, libraries, and configuration files needed to construct an embedded Linux wireless access point. Some of the software available for constructing embedded Linux systems will be discussed, and selection criteria for which tools to use for differing embedded applications will be presented. During the presentation, an embedded Linux wireless access point will be constructed using the Linux kernel, the uClibc C library, BusyBox, the syslinux bootloader, iptables, etc. Emphasis will be placed on the more generic aspects of building an embedded Linux system using BusyBox and uClibc. At the conclusion of the presentation, the presenter will (with luck) boot up the newly constructed wireless access point and demonstrate that it is working perfectly. Source code, build system, cross compilers, and detailed instructions will be made available.

1 Introduction

When I began working on embedded Linux, the question of whether or not Linux was small enough to fit inside a particular device was a difficult problem. Linux distributions[1] have historically been designed for server and desktop systems. As such, they deliver a full-featured, comprehensive set of tools for just about every purpose imaginable. Most Linux distributions, such as Red Hat, Debian, or SuSE, provide hundreds of separate software packages adding up to several gigabytes of software. The goal of server or desktop Linux distributions has been to provide as much value as possible to the user; therefore, the large size is quite understandable. However, this has caused the Linux operating system to be much larger then is desirable for building an embedded Linux system such as a wireless access point. Since embedded devices represent a fundamentally different target for Linux, it became apparent to me that embedded devices would need different software than what is commonly used on desktop systems. I knew that Linux has a number of strengths which make it extremely attractive for the next generation of embedded devices, yet I could see that developers would need new tools to take advantage of Linux within small, embedded spaces.

I began working on embedded Linux in the middle of 1999. At the time, building an 'embedded Linux' system basically involved copying binaries from an existing Linux distribution to a target device. If the needed software did not fit into the required amount of flash memory, there was really nothing to be done about it except to add more flash or give up on the project. Very little effort had been made to develop smaller application programs and li-

[1] The term "distribution" is used by the Linux community to refer to a collection of software, including the Linux kernel, application programs, and needed library code, which makes up a complete running system. Sometimes, the term "Linux" or "GNU/Linux" is also used to refer to this collection of software.

braries designed for use in embedded Linux.

As I began to analyze how I could save space, I decided that there were three main areas that could be attacked to shrink the footprint of an embedded Linux system: the kernel, the set of common application programs included in the system, and the shared libraries. Many people doing Linux kernel development were at least talking about shrinking the footprint of the kernel. For the past five years, I have focused on the latter two areas: shrinking the footprint of the application programs and libraries required to produce a working embedded Linux system. This paper will describe some of the software tools I've worked on and maintained, which are now available for building very small embedded Linux systems.

2 The C Library

Let's take a look at an embedded Linux system, the Linux Router Project, which was available in 1999. `http://www.linuxrouter.org/` The Linux Router Project, begun by Dave Cinege, was and continues to be a very commonly used embedded Linux system. Its self-described tagline reads "A networking-centric micro-distribution of Linux" which is "small enough to fit on a single 1.44MB floppy disk, and makes building and maintaining routers, access servers, thin servers, thin clients, network appliances, and typically embedded systems next to trivial." First, let's download a copy of one of the Linux Router Project's "idiot images." I grabbed my copy from the mirror site at `ftp://sunsite.unc.edu/pub/Linux/distributions/linux-router/dists/current/idiot-image_1440KB_FAT_2.9.8_Linux_2.2.gz`.

Opening up the idiot-image there are several very interesting things to be seen.

```
# gunzip \
  idiot-image_1440KB_FAT_2.9.8_Linux_2.2.gz
# mount \
  idiot-image_1440KB_FAT_2.9.8_Linux_2.2 \
  /mnt -o loop

# du -ch /mnt/*
34K     /mnt/etc.lrp
6.0K    /mnt/ldlinux.sys
512K    /mnt/linux
512     /mnt/local.lrp
1.0K    /mnt/log.lrp
17K     /mnt/modules.lrp
809K    /mnt/root.lrp
512     /mnt/syslinux.cfg
1.0K    /mnt/syslinux.dpy
1.4M    total

# mkdir test
# cd test
# tar -xzf /mnt/root.lrp

# du -hs
2.2M    .
2.2M    total

# du -ch bin root sbin usr var
460K    bin
8.0K    root
264K    sbin
12K     usr/bin
304K    usr/sbin
36K     usr/lib/ipmasqadm
40K     usr/lib
360K    usr
56K     var/lib/lrpkg
60K     var/lib
4.0K    var/spool/cron/crontabs
8.0K    var/spool/cron
12K     var/spool
76K     var
1.2M    total

# du -ch lib
24K     lib/POSIXness
1.1M    lib
1.1M    total

# du -h lib/libc-2.0.7.so
644K    lib/libc-2.0.7.so
```

Taking a look at the software contained in this embedded Linux system, we quickly notice that in a software image totaling 2.2 Megabytes, the libraries take up over half the space. If we look even closer at the set of libraries, we quickly find that the largest single component in the entire system is the GNU C library, in this case occupying nearly 650k. What is more, this is a very old version of the C library; newer versions of GNU glibc,

such as version 2.3.2, are over 1.2 Megabytes all by themselves! There are tools available from Linux vendors and in the Open Source community which can reduce the footprint of the GNU C library considerably by stripping unwanted symbols; however, using such tools precludes adding additional software at a later date. Even when these tools are appropriate, there are limits to the amount of size which can be reclaimed from the GNU C library in this way.

The prospect of shrinking a single library that takes up so much space certainly looked like low hanging fruit. In practice, however, replacing the GNU C library for embedded Linux systems was not easy task.

3 The origins of uClibc

As I despaired over the large size of the GNU C library, I decided that the best thing to do would be to find another C library for Linux that would be better suited for embedded systems. I spent quite a bit of time looking around, and after carefully evaluating the various Open Source C libraries that I knew of[2], I sadly found that none of them were suitable replacements for glibc. Of all the Open Source C libraries, the library closest to what I imagined an embedded C library should be was called `uC-libc` and was being used for uClinux systems. However, it also had many problems at the time—not the least of which was that uC-libc had no central maintainer. The only mechanism being used to support multiple architectures was a complete source tree fork, and there had already been a few such forks with plenty of divergant code. In short, uC-libc was a mess of twisty versions, all different. After spending some time with the code, I decided to fix it, and in the process changed the name to `uClibc` (no hyphen).

With the help of D. Jeff Dionne, one of the creators of uClinux[3], I ported uClibc to run on Intel compatible x86 CPUs. I then grafted in the header files from glibc 2.1.3 to simplify software ports, and I cleaned up the resulting breakage. The header files were later updated again to generally match glibc 2.3.2. This effort has made porting software from glibc to uClibc extremely easy. There were, however, many functions in uClibc that were either broken or missing and which had to be re-written or created from scratch. When appropriate, I sometimes grafted in bits of code from the current GNU C library and libc5. Once the core of the library was reasonably solid, I began adding a platform abstraction layer to allow uClibc to compile and run on different types of CPUs. Once I had both the ARM and x86 platforms basically running, I made a few small announcements to the Linux community. At that point, several people began to make regular contributions. Most notably was Manuel Novoa III, who began contributing at that time. He has continued working on uClibc and is responsible for significant portions of uClibc such as the stdio and internationalization code.

After a great deal of effort, we were able to build the first shared library version of uClibc in January 2001. And earlier this year we were able to compile a Debian Woody system using uClibc[4], demonstrating the library is now able

[2] The Open Source C libraries I evaluated at the time included Al's Free C RunTime library (no longer on the Internet); dietlibc available from `http://www.fefe.de/dietlibc/`; the minix C library available from `http://www.cs.vu.nl/cgi-bin/raw/pub/minix/`; the newlib library available from `http://sources.redhat.com/newlib/`; and the eCos C library available from `ftp://ecos.sourceware.org/pub/ecos/`.

[3] uClinux is a port of Linux designed to run on microcontrollers which lack Memory Management Units (MMUs) such as the Motorolla DragonBall or the ARM7TDMI. The uClinux web site is found at `http://www.uclinux.org/`.

[4] `http://www.uclibc.org/dists/`

to support a complete Linux distribution. People now use uClibc to build versions of Gentoo, Slackware, Linux from Scratch, rescue disks, and even live Linux CDs[5]. A number of commercial products have also been released using uClibc, such as wireless routers, network attached storage devices, DVD players, etc.

4 Compiling uClibc

Before we can compile uClibc, we must first grab a copy of the source code and unpack it so it is ready to use. For this paper, we will just grab a copy of the daily uClibc snapshot.

```
# SITE=http://www.uclibc.org/downloads
# wget -q $SITE/uClibc-snapshot.tar.bz2

# tar -xjf uClibc-snapshot.tar.bz2
# cd uClibc
```

uClibc requires a configuration file, `.config`, that can be edited to change the way the library is compiled, such as to enable or disable features (i.e. whether debugging support is enabled or not), to select a cross-compiler, etc. The preferred method when starting from scratch is to run `make defconfig` followed by `make menuconfig`. Since we are going to be targeting a standard Intel compatible x86 system, no changes to the default configuration file are necessary.

5 The Origins of BusyBox

As I mentioned earlier, the two components of an embedded Linux that I chose to work towards reducing in size were the shared libraries and the set common application programs. A typical Linux system contains a variety of command-line utilities from numerous different organizations and independent programmers. Among the most prominent of these utilities were GNU shellutils, fileutils, textutils (now combined to form GNU coreutils), and similar programs that can be run within a shell (commands such as `sed`, `grep`, `ls`, etc.). The GNU utilities are generally very high-quality programs, and are almost without exception very, very feature-rich. The large feature set comes at the cost of being quite large—prohibitively large for an embedded Linux system. After some investigation, I determined that it would be more efficient to replace them rather than try to strip them down, so I began looking at alternatives.

Just as with alternative C libraries, there were several choices for small shell utilities: BSD has a number of utilities which could be used. The Minix operating system, which had recently released under a free software license, also had many useful utilities. Sash, the stand alone shell, was also a possibility. After quite a lot of research, the one that seemed to be the best fit was BusyBox. It also appealed to me because I was already familiar with BusyBox from its use on the Debian boot floppies, and because I was acquainted with Bruce Perens, who was the maintainer. Starting approximately in October 1999, I began enhancing BusyBox and fixing the most obvious problems. Since Bruce was otherwise occupied and was no longer actively maintaining BusyBox, Bruce eventually consented to let me take over maintainership.

Since that time, BusyBox has gained a large following and attracted development talent from literally the whole world. It has been used in commercial products such as the IBM Linux wristwatch, the Sharp Zaurus PDA, and Linksys wireless routers such as the WRT54G, with many more products being released all the time. So many new features and applets have been added to BusyBox, that the biggest chal-

[5] Puppy Linux available from `http://www.goosee.com/puppy/` is a live linux CD system built with uClibc that includes such favorites as XFree86 and Mozilla.

lenge I now face is simply keeping up with all of the patches that get submitted!

6 So, How Does It Work?

BusyBox is a multi-call binary that combines many common Unix utilities into a single executable. When it is run, BusyBox checks if it was invoked via a symbolic link (a `symlink`), and if the name of the symlink matches the name of an applet that was compiled into BusyBox, it runs that applet. If BusyBox is invoked as `busybox`, then it will read the command line and try to execute the applet name passed as the first argument. For example:

```
# ./busybox date
Wed Jun 2 15:01:03 MDT 2004

# ./busybox echo "hello there"
hello there

# ln -s ./busybox uname
# ./uname
Linux
```

BusyBox is designed such that the developer compiling it for an embedded system can select exactly which applets to include in the final binary. Thus, it is possible to strip out support for unneeded and unwanted functionality, resulting in a smaller binary with a carefully selected set of commands. The customization granularity for BusyBox even goes one step further: each applet may contain multiple features that can be turned on or off. Thus, for example, if you do not wish to include large file support, or you do not need to mount NFS filesystems, you can simply turn these features off, further reducing the size of the final BusyBox binary.

7 Compiling Busybox

Let's walk through a normal compile of BusyBox. First, we must grab a copy of the BusyBox source code and unpack it so it is ready to use. For this paper, we will just grab a copy of the daily BusyBox snapshot.

```
# SITE=http://www.busybox.net/downloads
# wget -q $SITE/busybox-snapshot.tar.bz2
# tar -xjf busybox-snapshot.tar.bz2
# cd busybox
```

Now that we are in the BusyBox source directory we can configure BusyBox so that it meets the needs of our embedded Linux system. This is done by editing the file `.config` to change the set of applets that are compiled into BusyBox, to enable or disable features (i.e. whether debugging support is enabled or not), and to select a cross-compiler. The preferred method when starting from scratch is to run `make defconfig` followed by `make menuconfig`. Once BusyBox has been configured to taste, you just need to run `make` to compile it.

8 Installing Busybox to a Target

If you then want to install BusyBox onto a target device, this is most easily done by typing: `make install`. The installation script automatically creates all the required directories (such as `/bin`, `/sbin`, and the like) and creates appropriate symlinks in those directories for each applet that was compiled into the BusyBox binary.

If we wanted to install BusyBox to the directory /mnt, we would simply run:

```
# make PREFIX=/mnt install

[--installation text omitted--]
```

9 Let's build something that works!

Now that I have certainly bored you to death, we finally get to the fun part, building our own embedded Linux system. For hardware, I will be using a Soekris 4521 system[6] with an 133 Mhz AMD Elan CPU, 64 MB main memory, and a generic Intersil Prism based 802.11b card that can be driven using the `hostap`[7] driver. The root filesystem will be installed on a compact flash card.

To begin with, we need to create toolchain with which to compile the software for our wireless access point. This requires we first compile GNU binutils[8], then compile the GNU compiler collection—gcc[9], and then compile uClibc using the newly created gcc compiler. With all those steps completed, we must finally recompile gcc using using the newly built uClibc library so that `libgcc_s` and `libstdc++` can be linked with uClibc.

Fortunately, the process of creating a uClibc toolchain can be automated. First we will go to the uClibc website and obtain a copy of the uClibc `buildroot` by going here:

`http://www.uclibc.org/cgi-bin/cvsweb/buildroot/`

and clicking on the "Download tarball" link[10]. This is a simple GNU make based build system which first builds a uClibc toolchain, and then builds a root filesystem using the newly built uClibc toolchain.

For the root filesystem of our wireless access point, we will need a Linux kernel, uClibc, BusyBox, pcmcia-cs, iptables, hostap, wtools, bridgeutils, and the dropbear ssh server. To compile these programs, we will first edit the buildroot Makefile to enable each of these items. Figure 1 shows the changes I made to the buildroot Makefile:

Running `make` at this point will download the needed software packages, build a toolchain, and create a minimal root filesystem with the specified software installed.

On my system, with all the software packages previously downloaded and cached locally, a complete build took 17 minutes, 19 seconds. Depending on the speed of your network connection and the speed of your build system, now might be an excellent time to take a lunch break, take a walk, or watch a movie.

10 Checking out the new Root Filesystem

We now have our root filesystem finished and ready to go. But we still need to do a little more work before we can boot up our newly built embedded Linux system. First, we need to compress our root filesystem so it can be loaded as an initrd.

```
# gzip -9 root_fs_i386
# ls -sh root_fs_i386.gz
1.1M root_fs_i386.gz
```

Now that our root filesystem has been compressed, it is ready to install on the boot media. To make things simple, I will install the Compact Flash boot media into a USB card reader device, and copy files using the card reader.

```
# ms-sys -s /dev/sda
Public domain master boot record
successfully written to /dev/sda
```

[6] `http://www.soekris.com/net4521.htm`
[7] `http://hostap.epitest.fi/`
[8] `http://sources.redhat.com/binutils/`
[9] `http://gcc.gnu.org/`
[10] `http://www.uclibc.org/cgi-bin/cvsweb/buildroot.tar.gz?view=tar`

```
--- Makefile
+++ Makefile
@@ -140,6 +140,6 @@
 # Unless you want to build a kernel, I recommend just using
 # that...
-TARGETS+=kernel-headers
-#TARGETS+=linux
+#TARGETS+=kernel-headers
+TARGETS+=linux
 #TARGETS+=system-linux

@@ -150,5 +150,5 @@
 #TARGETS+=zlib openssl openssh
 # Dropbear sshd is much smaller than openssl + openssh
-#TARGETS+=dropbear_sshd
+TARGETS+=dropbear_sshd

 # Everything needed to build a full uClibc development system!
@@ -175,5 +175,5 @@

 # Some stuff for access points and firewalls
-#TARGETS+=iptables hostap wtools dhcp_relay bridge
+TARGETS+=iptables hostap wtools dhcp_relay bridge
 #TARGETS+=iproute2 netsnmp
```

Figure 1: Changes to the buildroot Makefile

```
# mkdosfs /dev/sda1
mkdosfs 2.10 (22 Sep 2003)

# syslinux /dev/sda1

# cp root_fs_i386.gz /mnt/root_fs.gz

# cp build_i386/buildroot-kernel /mnt/linux
```

So we now have a copy of our root filesystem and Linux kernel on the compact flash disk. Finally, we need to configure the bootloader. In case you missed it a few steps ago, we are using the syslinux bootloader for this example. I happen to have a ready to use syslinux configuration file, so I will now install that to the compact flash disk as well:

```
# cat  syslinux.cfg
TIMEOUT 0
PROMPT 0
DEFAULT linux
LABEL linux
 KERNEL linux
 APPEND initrd=root_fs.gz \
        console=ttyS0,57600 \
        root=/dev/ram0 boot=/dev/hda1,msdos rw

# cp syslinux.cfg /mnt
```

And now, finally, we are done. Our embedded Linux system is complete and ready to boot. And you know what? It is very, very small. Take a look at Table 1.

With a carefully optimized Linux kernel (which this kernel unfortunately isn't) we could expect to have even more free space. And remember, every bit of space we save is money that embedded Linux developers don't have to spend on expensive flash memory. So now comes the final test; it is now time to boot from our compact flash disk. Here is what you should see.

```
[----kernel boot messages snipped--]
```

```
# ll /mnt
total 1.9M
drwxr-r-     2  root  root   16K  Jun  2  16:39  ./
drwxr-xr-x  22  root  root  4.0K  Feb  6  07:40  ../
-r-xr-r-     1  root  root  7.7K  Jun  2  16:36  ldlinux.sys*
-rwxr-r-     1  root  root  795K  Jun  2  16:36  linux*
-rwxr-r-     1  root  root  1.1M  Jun  2  16:36  root_fs.gz*
-rwxr-r-     1  root  root   170  Jun  2  16:39  syslinux.cfg*
```

Table 1: Output of `ls -lh /mnt`.

```
Freeing unused kernel memory: 64k freed

Welcome to the Erik's wireless access point.

uclibc login: root

BusyBox v1.00-pre10 (2004.06.02-21:54+0000)
Built-in shell (ash)
Enter 'help' for a list of built-in commands.

# du -h / | tail -n 1
2.6M

#
```

And there you have it—your very own wireless access point. Some additional configuration will be necessary to start up the wireless interface, which will be demonstrated during my presentation.

11 Conclusion

The two largest components of a standard Linux system are the utilities and the libraries. By replacing these with smaller equivalents a much more compact system can be built. Using BusyBox and uClibc allows you to customize your embedded distribution by stripping out unneeded applets and features, thus further reducing the final image size. This space savings translates directly into decreased cost per unit as less flash memory will be required. Combine this with the cost savings of using Linux, rather than a more expensive proprietary OS, and the reasons for using Linux become very compelling. The example Wireless Access point we created is a simple but useful example. There are thousands of other potential applications that are only waiting for you to create them.

Proceedings of the Linux Symposium

Volume One

July 21st–24th, 2004
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Jes Sorensen, *Wild Open Source, Inc.*
Matt Domsch, *Dell*
Gerrit Huizenga, *IBM*
Matthew Wilcox, *Hewlett-Packard*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*