Embedded Linux From Scratch

in 40 minutes!

Michael Opdenacker

Free Electrons

http://free-electrons.com/

nada + 40 min = 

Created with OpenOffice.org 2.0

Rights to copy

creative commons
COMMONS DEED

Attribution – ShareAlike 2.0

You are free

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions

Attribution. You must give the original author credit.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

License text: http://creativecommons.org/licenses/by-sa/2.0/legalcode

Document sources, updates and translations:
http://free-electrons.com/articles/elfs

Corrections, suggestions, contributions and translations are welcome!

Tutorial goals

Build a tiny embedded system entirely from scratch, in 40 minutes

- Linux kernel configuring and compiling
- Root filesystem creation
- Busybox compiling and installation
- Device file creation
- System initialization scripts: virtual filesystems, networking
- Setup of a simple HTTP interface to the system

Show you how simple this can be!

Top-down approach

Top-down approach to building an embedded system

- Starting from a complete desktop GNU/Linux distribution (Debian, Fedora...) and removing unneeded stuff.
- Very tedious job: need to go through a huge number of files and packages. Need to understand what each file and package is about before removing it.
- Keeping unnecessarily complex scripts and configuration files.
- The end result is still quite big, as standard desktop toolsets and libraries are used. Lots of shared libraries still needed too.

Bottom-up approach

Bottom-up approach to building embedded systems

- Starting with an empty or minimalistic root filesystem, adding only things that you need.
- Much easier to do! You just spend time on things you need.
- Much easier to control and maintain: you build an understanding about the tools you use.
- You only need very simple configuration scripts.
- The end result can be extremely small, all the more as you use lightweight toolsets instead.

Tools used in this tutorial

Explanatory slides to show while compiling

qemu

http://qemu.org

Fast processor emulator

using a portable dynamic translator.

2 operating modes

- Full system emulation: processor and various peripherals
  Supported: `x86`, `x86_64`, `ppc`
- User mode emulation (Linux host only): can run applications compiled for another CPU.
  Supported: `x86`, `ppc`, `arm`, `sparc`

qemu examples

User emulation

- Easy to run Busybox for `arm` on `i386` GNU / Linux:
```
qemu-arm -L /usr/local/arm/3.3.2 \
/home/bart/arm/busybox-1.00-pre8/busybox ls
```
- `-L`: target C library binaries path (here cross-compiler toolchain path)

System emulation

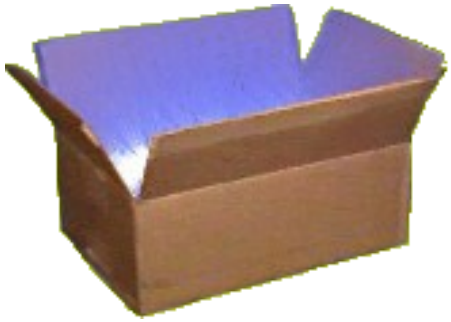
- Even easier to run:
```
qemu linux.img
```
- `linux.img`: full partition image including the kernel

  Plenty of images for free operating systems on http://freeoszoo.org!

General purpose toolbox: busybox

http://www.busybox.net/ from Codepoet Consulting

- Most Unix command line utilities within a single executable! Even includes a web server!
- Sizes less than 1 MB (statically compiled with glibc)
  less than 500 MB (statically compiled with uClibc)
- Easy to configure which features to include
- The best choice for
  - Initrds with complex scripts
  - Any embedded system!



Busybox commands!

addgroup, adduser, adjtimex, ar, arping, ash, awk, basename, bunzip2, bzcat, cal, cat, chgrp, chmod, chown, chroot, chvt, clear, cmp, cp, cpio, crond, crontab, cut, date, dc, dd, deallocvt, delgroup, deluser, devfsd, df, dirname, dmesg, dos2unix, dpkg, dpkg-deb, du, dumpkmap, dumpleases, echo, egrep, env, expr, false, fbset, fdflush, fdformat, fdisk, fgrep, find, fold, free, freeramdisk, fsck.minix, ftpget, ftpput, getopt, getty, grep, gunzip, gzip, halt, hdparm, head, hexdump, hostid, hostname, **httpd**, hush, hwclock, id, ifconfig, ifdown, ifup, inetd, init, insmod, install, ip, ipaddr, ipcalc, iplink, iproute, iptunnel, kill, killall, klogd, lash, last, length, linuxrc, ln, loadfont, loadkmap, logger, login, logname, logread, losetup, ls, lsmod, makedevs, md5sum, mesg, mkdir, mkfifo, mkfs.minix, mknod, mkswap, mktemp, modprobe, more, mount, msh, mt, mv, nameif, nc, netstat, nslookup, od, openvt, passwd, patch, pidof, ping, ping6, pipe_progress, pivot_root, poweroff, printf, ps, pwd, rdate, readlink, realpath, reboot, renice, reset, rm, rmdir, rmmod, route, rpm, rpm2cpio, run-parts, rx, sed, seq, setkeycodes, sha1sum, sleep, sort, start-stop-daemon, strings, stty, su, sulogin, swapoff, swapon, sync, sysctl, syslogd, tail, tar, tee, telnet, telnetd, test, tftp, time, top, touch, tr, traceroute, true, tty, udhcpc, udhcpd, umount, uname, uncompress, uniq, unix2dos, unzip, uptime, usleep, uudecode, uuencode, vconfig, vi, vlock, watch, watchdog, wc, wget, which, who, whoami, xargs, yes, zcat

glibc

http://www.gnu.org/software/libc/

- C library from the GNU project
- Designed for performance, standards compliance and portability
- Found on all GNU / Linux host systems
- Quite big for small embedded systems: about ~1.7MB on Familiar Linux iPAQs (`libc`: 1.2 MB, `libm`: 500 KB)
- Example “hello world” program size: 12 KB (dynamically linked), 350 KB (statically linked)

uClibc

http://www.uclibc.org/ for CodePoet Consulting

- Lightweight C library for small embedded systems, with most features though.
- The whole Debian Woody was recently ported to it... You can assume it satisfied most needs!
- Example size (`arm`): approx. 400KB (`libuClibc`: 300 KB, `libm`: 55KB)
- Example “hello world” program size: 2 KB (dynamically linked), 18 KB (statically linked).

Kernel userspace interface

A few examples:

- `/proc/cpuinfo`: processor information
- `/proc/meminfo`: memory status
- `/proc/version`: version and build information
- `/proc/cmdline`: kernel command line
- `/proc/<pid>/environ`: calling environment
- `/proc/<pid>/cmdline`: process command line

... and many more! Complete details in the kernel sources:
`Documentation/filesystems/proc.txt`

What we did

Compiling the Linux kernel

- Getting the Linux sources from http://kernel.org
- Setting `EXTRA_VERSION` in `Makefile`
- Setting a minimalistic kernel configuration:
  `make allnoconfig`
- Adding settings specific to the embedded system:
  `make xconfig` or `make menuconfig`
- Compiling:
  `make`
- Result: compressed kernel image
  `arch/i386/boot/bzImage`

Creating a root filesystem

- Creating an empty file with a `320K` size:
  `dd if=/dev/zero of=rootfs.img bs=320k count=1`
- Formating this file for the `ext2` filesystem:
  `mkfs.ext2 -i 1024 -F rootfs.ext2`

1 inode every 1024 bytes -> 320 files

instead of 1 inode every 4096 bytes -> only 80 files!

Compiling busybox

- Getting the sources from http://busybox.net
- Configuring busybox:
  `make menuconfig`
  Choosing to build a statically, natively compiled executable.
- Compiling busybox:
  `make`
- Pre-installing busybox (in the `_install/` subdirectory):
  `make install`
- Result: a `500K` executable implementing
  all the commands that we need!

Re-compiling busybox

`500K` was already way too big for a perfect embedded system!

- Configuring busybox again
  `make menuconfig`
  Choosing to build a statically, “cross-compiled” executable, using a `uClibc` toolchain instead of the standard `glibc` one.
- Compiling busybox:
  `make`
- Pre-installing busybox (in the `_install/` subdirectory):
  `make install`
- Result: a `250K` executable implementing all the commands that we need!

Populating the root filesystem

Logged as `root`:

- Creating a mount point:
  ```
  mkdir /mnt/rootfs
  ```
- Mounting the root filesystem image:
  ```
  mount -o loop rootfs.img /mnt/rootfs
  ```
- Copying the busybox file structure into the mounted image:
  ```
  rsync -a busybox/_install/ /mnt/rootfs/
  chown -R root:root /mnt/rootfs/
  ```
- Flushing the changes into the mounted filesystem image:
  ```
  sync
  ```

Booting the virtual system

Using the `qemu` emulator as a bootloader

(no need to copy the kernel to the target storage)

```
qemu \
  -m 32 \
  -hda rootfs.img \
  -kernel linux-2.6.12/arch/i386/boot/bzImage \
  -append "root=/dev/hda clock=pit"
```

Amount of memory (MB) in the emulated target

Contents of the emulated hard disk

Kernel image

Kernel command line

Creating device files

- Creating device files when programs complain:

```
mkdir /mnt/rootfs/dev
mknod /mnt/rootfs/dev/console c 5 1
mknod /mnt/rootfs/dev/null 1 3
```

- Taking the GNU/Linux host as an example to find correct major and minor numbers:

```
ls -l /dev/console
ls -l /dev/null
```


Mounting virtual filesystems

Making `/proc` and `/sys` available
(required by several command line tools such as `ps`)

- Mounting `/proc`:
```
mount -t proc none /proc
```
- Mounting `/sys`:
```
mount -t sysfs none /sys
```

Filesystem type

Raw device
or filesystem image
In the case of virtual
filesystems, any string is fine

Mount point

/etc/inittab file for busybox init

Creating the `/etc/inittab` file required by busybox `init`

Getting an example from busybox documentation

(not from the GNU/Linux host... missing features!)

```
# This is run first script
::sysinit:/etc/init.d/rcS
# Start an "askfirst" shell on the console
::askfirst:-/bin/sh
# Stuff to do when restarting the init process
::restart:/sbin/init
# Stuff to do before rebooting
::ctrlaltdel:/sbin/reboot
::shutdown:/bin/umount -a -r
```


Setting up networking

- Adding TCP/IP and network card driver to the kernel
- Bringing up the network interface:
  `ifconfig eth0 172.20.0.2`
- Using the GNU/Linux host as a gateway:
  `route add default gw 172.20.0.1`
- Testing networking:
  `ping -c 3 172.20.0.1`
  `-c 3:` useful when `[Ctrl][C]` doesn't work (missing tty settings)
- Testing routing:
  `ping -c 3 <external address>`

Starting up a http server

- Copying HTML pages on `/www` (for example)
- Creating CGI scripts in `/www/cgi-bin/`
- Starting the busybox http server:
  `/usr/sbin/httpd -h /www/ &`

/etc/init.d/rcS startup script

```
#!/bin/sh
mount -t proc none /proc
mount -t sysfs none /sys
ifconfig eth0 172.20.0.2
route add default gw 172.20.0.1
/usr/sbin/httpd -h /www/ &
/bin/sh
```

See how simple this can be!

A simplistic CGI script

/www/cgi-bin/uptime:

```
#!/bin/sh
echo "Content-type: text/html"
echo ""
echo "<html><header></header><body>"
echo "<h1>Uptime information</h1>"
echo "Your embedded device has been
running for:<pre><font color=Blue>"
echo `uptime`
echo "</font></pre></u>"
echo "</body></html>"
```

Limitations

A few minor limitations

- CGI scripts: can't implement non-trivial scripts
  Need to code in C to support posting and URL parsing.
- System specific software: can't be part of busybox.
  Need more C executables. As a consequence, need to include the uClibc library and compile the executables with shared library support.

They are easy and cheap to overcome!

Real embedded systems

This tutorial has already been done on real development boards!

- Need to install and configure a bootloader (if missing)
- Need to transfer kernel and root filesystem images to the target. An efficient way is to make the target boot on a NFS exported directory on the GNU/Linux host.

Related documents

This document belongs to the materials of an Embedded Linux training from Free Electrons, available under a free documentation license (more than 900 pages!)

http://free-electrons.com/training

- Introduction to Unix and GNU / Linux
- Embedded Linux kernel and driver development
- Tools for embedded Linux systems
- Audio in embedded Linux systems
- Multimedia in embedded Linux systems

http://free-electrons.com/articles

- Java in embedded Linux systems
- What's new in Linux 2.6?
- Introduction to uClinux
- Linux real-time extensions
- Linux on TI OMAP processors
- Introduction to software development tools