Hacking Linux-Powered Devices

Stefan Arentz
<stefan@soze.com>

Part I

Introduction

What is Embedded Linux?

Embedded usually means that it is a device with limited and specialized capabilities. It is not a ‘personal computer’ as your laptop or PC on your desk.

Embedded Linux means that there is a Linux kernel running on such a device.

Usually together with a combination of proprietary software and other OSS components running on top of that kernel. (The “user space” parts.)

Example: an imaginary portable DivX player

(From a Linux POV)

Hardware: CPU, RAM, Flash card, screen, bunch of buttons.

Process listing of an imaginary portable DivX player

```
PID   Uid       VmSize Stat Command
  1     0          396 S    init
  2     0         4829 S    mplayer
```

This could be a real world example, sometimes it really is this simple.

Some Real Examples of Linux-Powered Devices

TomTom GO
GPS Navigation

DreamBox
Digital TV/Radio Tuner

Linksys WRT54G
Wireless AP

Linux is a paradigm shift for hardware vendors

- They have to trust a “community work”
- They have to publish (parts of) their own work (‘The GNU GPL Revisited’ lecture)
  - There is still the ‘object code only kernel modules’ thing
- They are moving away from proprietary embedded operating systems
  - Great because those were closed

End Result for “Us”

Access to a product’s source code: at least the kernel source and other OSS components used.

Easier to reverse engineer the closed parts and easier to hack and modify the device as a whole.

Part II

Breaking the EULA
Real World Example

First things First

Share Your Work and Research

Start a Wiki!

Example - Linksys WRT54G

wrt54g_2.02.7_code.bin

Our Goal

Get access to the contents of the (read-only) filesystem that is embedded in the firmware.

If we can do this then we have basically opened up the device; we can modify it’s default behavior and add our own modifications.

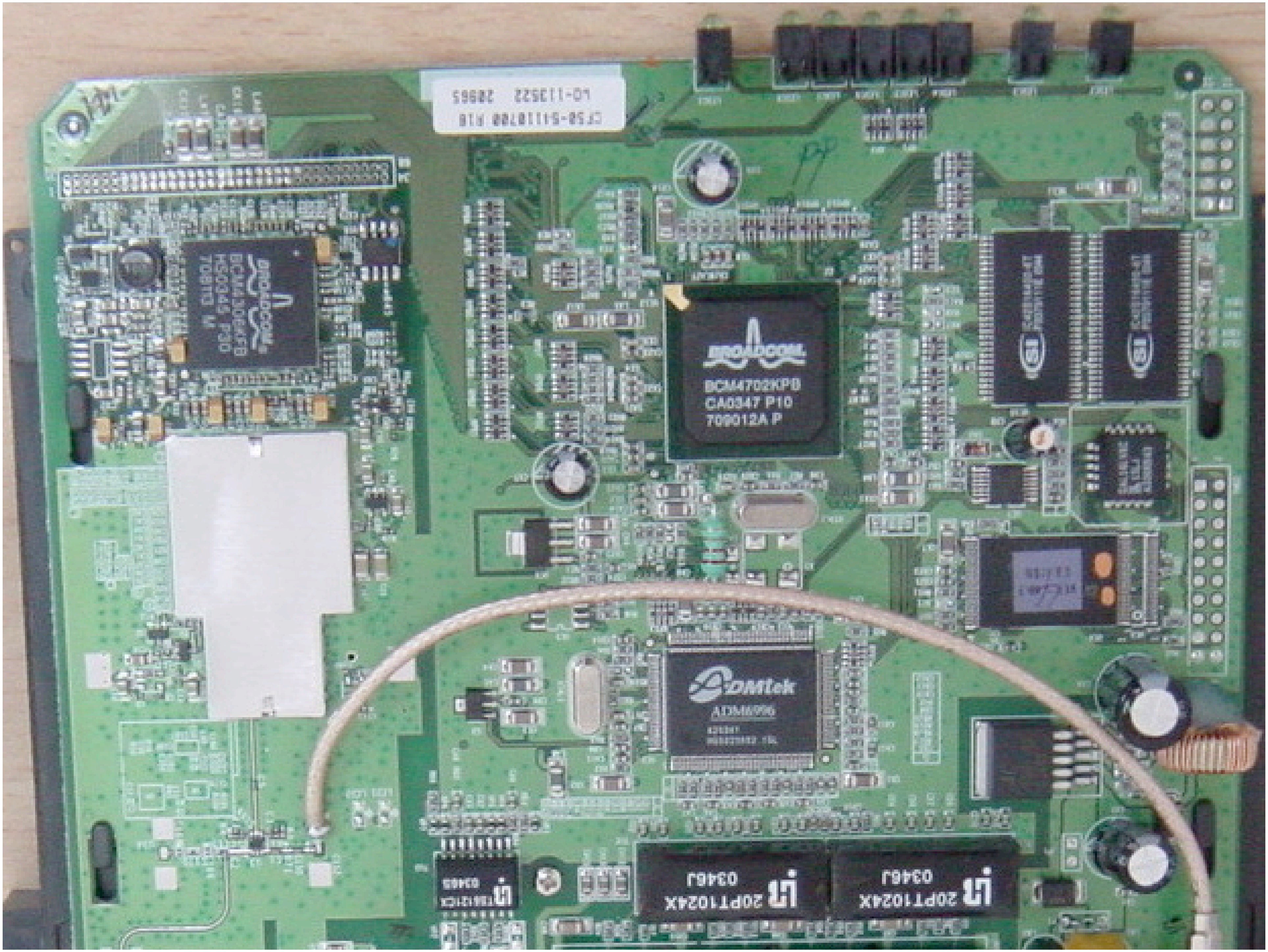
Understand the hardware

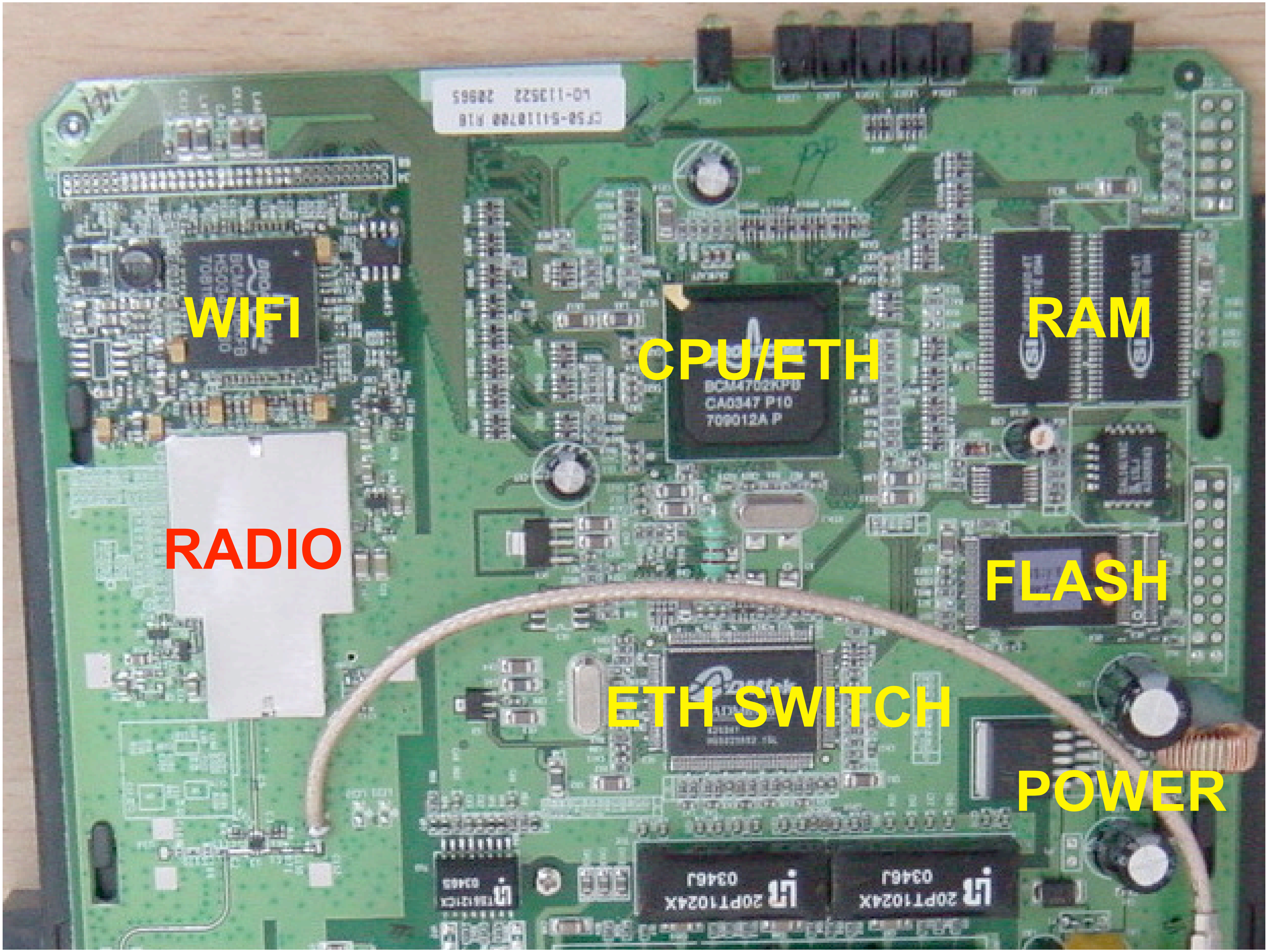
- Opening the box will void the warranty!
- Be careful, electricity can kill you!
- Static electricity can kill the device!

- Look at relations and connections between parts, connectors and things like switches.
- Look at part numbers (gooooogle them)

Goooooogle for the Datasheets

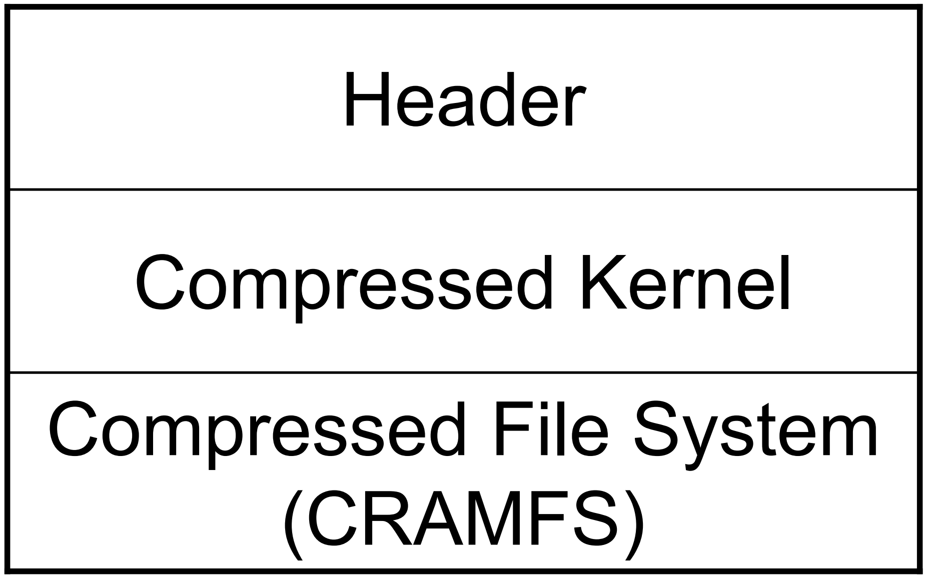
- Most vendors have them online (PDF)
- You don’t have to understand it all, electronics is a different discipline
- But it helps you to understand the device better
- And … you might find surprises!


BROADCOM
BCM4702KPB
CA0347 P10
709012A P
ADMtek
ADM6996
20PT1024X
0346J
20PT1024X
0346J


WIFI
CPU/ETH
BCM4702KPB
CA0347 P10
709012A P
RAM
RADIO
FLASH
ETH SWITCH
POWER

Back to our Goal: Hacking the Firmware Image

wrt54g_2.02.7_code.bin



Header

Compressed Kernel

Compressed File System (CRAMFS)

Firmware Image Header

```
% hexdump -C ~/WRT54G_1.30.1_US_code.bin
00000000  57 35 34 47 00 00 00 00  03 06 17 01 1e 01 55 32  |W54G..........U2|
00000010  4e 44 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |ND..............|
00000020  48 44 52 30 00 d0 29 00  78 53 6c d5 00 00 01 00  |HDR0.?).xSl?....|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

```
struct trx_header {
        uint32_t magic;         /* "HDR0" */
        uint32_t len;           /* Length of file including header */
        uint32_t crc32;         /* 32-bit CRC */
        uint32_t flag_version;  /* 0:15 flags, 16:31 version */
        uint32_t offsets[3];    /* Offsets of sections */
};
```

Extract the Kernel and CRAMFS

```
# Extract the file system (from the end)
% dd if=code.bin of=cramfs bs=1c skip=786464

# Extract the kernel (from the beginning, skip the header)
% dd if=code.bin of=kernel bs=1c skip=32 \
    count=786432
```

Mount the CRAMFS section

```
% sudo mount -o loop cramfs.section /mnt

% ls -l /mnt
drwxr-xr-x  1 root root  444 1970-01-01 01:00 bin/
drwxr-xr-x  1 root root    0 1970-01-01 01:00 dev/
drwxr-xr-x  1 root root   88 1970-01-01 01:00 etc/
drwxr-xr-x  1 root root  164 1970-01-01 01:00 lib/
drwxr-xr-x  1 root root    0 1970-01-01 01:00 mnt/
drwxr-xr-x  1 root root    0 1970-01-01 01:00 proc/
drwxr-xr-x  1 root root  292 1970-01-01 01:00 sbin/
drwxr-xr-x  1 root root    0 1970-01-01 01:00 tmp/
drwxr-xr-x  1 root root   64 1970-01-01 01:00 usr/
lrwxrwxrwx  1 root root    7 1970-01-01 01:00 var -> tmp/var
drwxr-xr-x  1 root root 1328 1970-01-01 01:00 www/
```

```
# ls -l /mnt/bin
-rwxr-xr-x  1 root root 268408 1970-01-01 01:00 busybox*
lrwxrwxrwx  1 root root      7 1970-01-01 01:00 cat -> busybox*
lrwxrwxrwx  1 root root      7 1970-01-01 01:00 chmod -> busybox*
lrwxrwxrwx  1 root root      7 1970-01-01 01:00 cp -> busybox*
lrwxrwxrwx  1 root root      7 1970-01-01 01:00 date -> busybox*
lrwxrwxrwx  1 root root      7 1970-01-01 01:00 dd -> busybox*
lrwxrwxrwx  1 root root      7 1970-01-01 01:00 df -> busybox*
lrwxrwxrwx  1 root root      7 1970-01-01 01:00 echo -> busybox*
lrwxrwxrwx  1 root root      7 1970-01-01 01:00 false -> busybox*
lrwxrwxrwx  1 root root      7 1970-01-01 01:00 grep -> busybox*

# file /mnt/bin/busybox
bin/busybox: ELF 32-bit LSB MIPS-I executable, MIPS,
  version 1 (SYSV), for GNU/Linux 2.3.99,
  dynamically linked (uses shared libs), stripped
```

```
% ls -l /mnt/lib
-rwxr-xr-x  1 root root 140264 1970-01-01 01:00 ld.so.1*
-rwxr-xr-x  1 root root  35180 1970-01-01 01:00 libcrypt.so.1*
-rwxr-xr-x  1 root root 871936 1970-01-01 01:00 libc.so.6*
-rwxr-xr-x  1 root root  15460 1970-01-01 01:00 libdl.so.2*
-rwxr-xr-x  1 root root  13564 1970-01-01 01:00 libm.so.6*
-rwxr-xr-x  1 root root  13564 1970-01-01 01:00 libnsl.so.1*
drwxr-xr-x  1 root root     20 1970-01-01 01:00 modules/

% strings /mnt/lib/libc.so.6 | grep GLIBC
GLIBC_2.2.3
```

Building a Toolchain (Optional)

Now that we know …

- The processor architecture (MIPS-I/LSB)
- The C Library used (glibc2 2.2.4)

… we can build a compatible toolchain. Building cross compilers is complex, but “crosstool” will handle all details for you. It even comes with an example script for the WRT54G!

```
% cd crosstool-0.28
% ./demo-mipsel.sh
```

Crosstool supports many other configurations too.

Modify and Regenerate the CRAMFS image

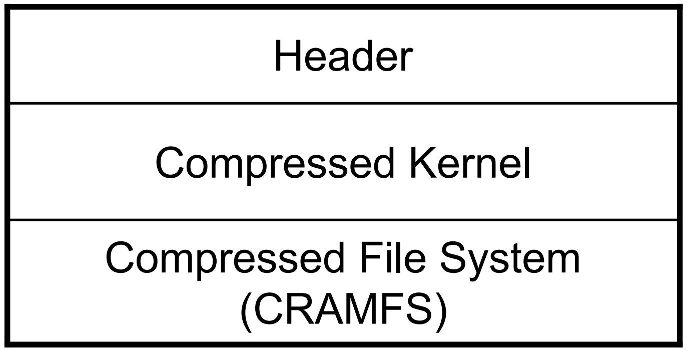
```
# Make a copy of the file system
% cp --archive /mnt ~/newrootfs

# Add a new server, make changes …
% cp myserver ~/newrootfs/usr/sbin/
% chmod 755 usr/sbin/myserver

# Change our copy back into a cramfs image
% cd ~/newrootfs
% mkcramfs . ~/newcramfs
```

Regenerate the Firmware Image

A scripting language is your friend for quick hacks like this.

```
% ./make-firmware-image.rb kernel newcramfs > code.bin
```

The script simply takes the kernel and the CRAMFS sections and creates a new firmware image with a header with the right CRC32 checksum.



Header

Compressed Kernel

Compressed File System (CRAMFS)

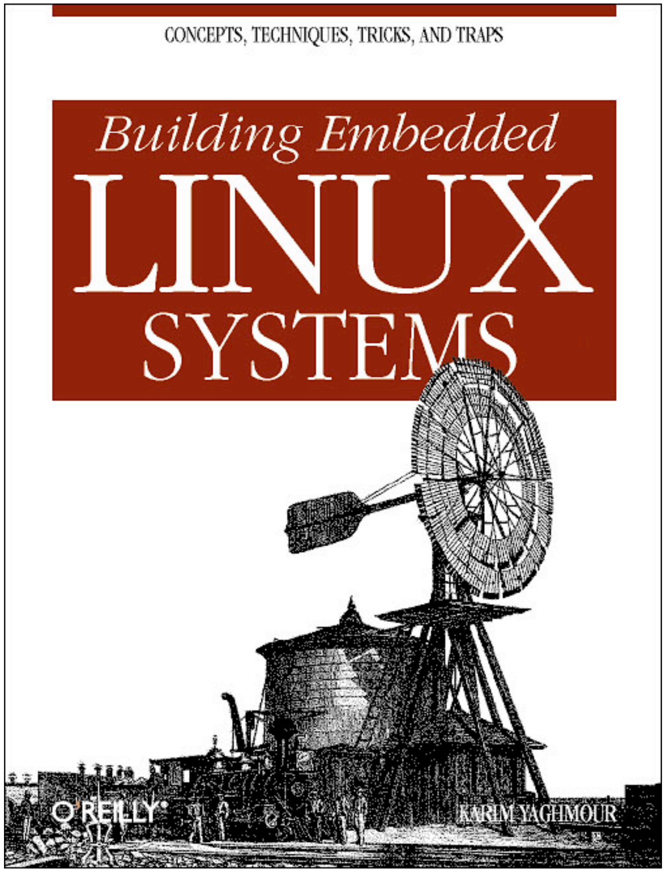
You can then upload this new firmware image to the WRT54G and use it. Hack done!

Conclusion

- Hacking Linux-Powered devices is definitely possible. Be creative and persistent!
- Don’t underestimate the power of a collective effort. Sharing is key.

References

- http://www.openwrt.org
- http://www.opentom.org
- Google for ‘embedded linux’



CONCEPTS, TECHNIQUES, TRICKS, AND TRAPS

Building Embedded LINUX SYSTEMS

O'REILLY

KARIM YAGHMOUR

Q&A