Coreboot Tutorial

...as used in Chrome OS

(YMMV)

Presented at OSCON 2013

http://goo.gl/jsE8EE

Agenda

(Don't panic - not every section is of equal length)

Intro / Background

Chrome OS Firmware

Development System

Preparing a Test System

Hands On

What Next?

Who are we?

- Ron Minnich
  - Started LinuxBIOS in 1999, been working on it since. Knows everything. Couldn't be here today.
- Stefan Reinauer
  - Renamed the project to Coreboot in 2008, been working on it since 2001. Knows *almost* everything.
- Bill Richardson
  - Was just going to watch, until Ron backed out. **Not** the former Governor of New Mexico. Hi.

We work for Google, but don't speak for them. All opinions are our own.

What is coreboot?

- http://www.coreboot.org
- A Free Software (GPLv2) project to replace the proprietary BIOS in modern computers.
- Originally intended for clusters, now used all over the place.
- It performs just a little bit of hardware initialization and then executes a *payload*.
- Lots of possible payloads: Linux kernel, GRUB2, Open Firmware, Etherboot/GPXE, SeaBIOS, ...

What is Chrome OS?

- "A fast, simple, and more secure computing experience for people who spend most of their time on the web." http://www.google.com/chromeos
- Chrome OS is only available on hardware.
- But **Chromium OS** is the open source project, with code available to anyone. http://www.chromium.org
- We'll take advantage of that today...

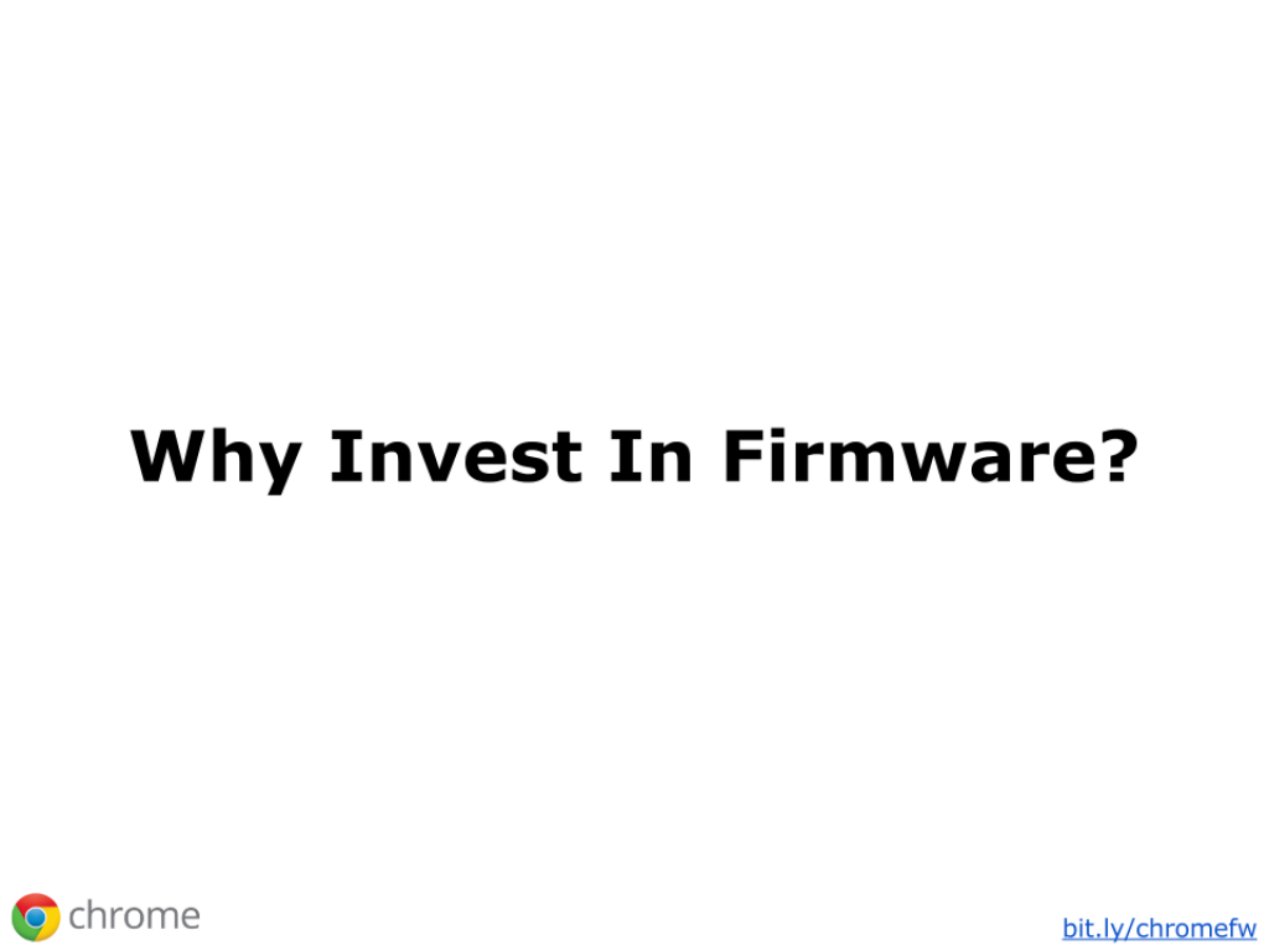
Coreboot in Chrome OS

- The first three Chromebooks used a proprietary BIOS, based on UEFI.
- The newer x86-based Chromebooks use Coreboot, with U-Boot as a payload.
- ARM-based Chromebooks use only U-Boot.

Why coreboot?

- Duncan Laurie is a Chrome OS engineer who presented at Linux Conf AU earlier this year
- His slides are better than mine, so I'm going to quote them...
- http://bit.ly/chromefw to see the rest

Duncan Laurie, at linux.conf.au 2013:

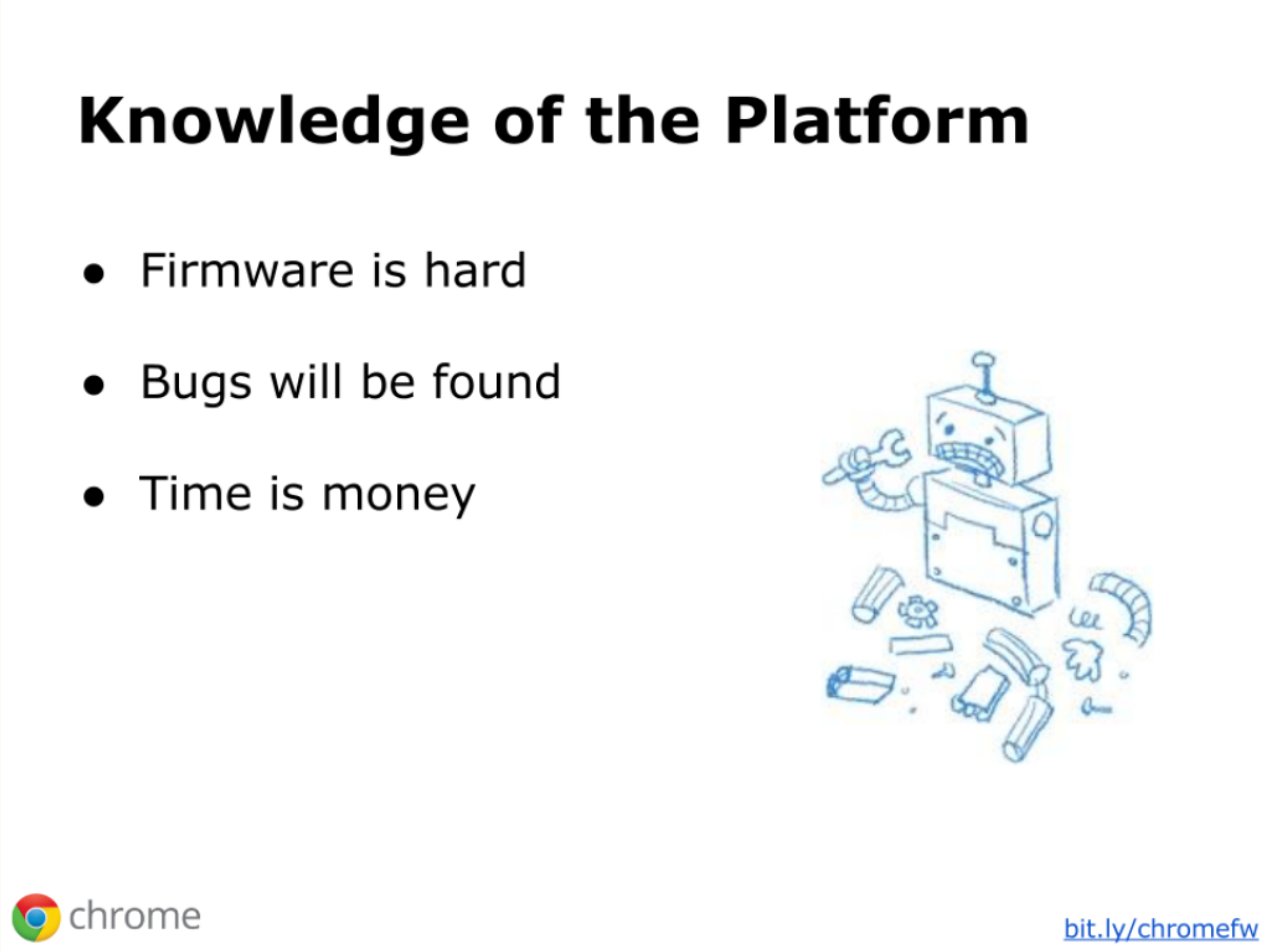


Why Invest In Firmware?

chrome

bit.ly/chromefw

Duncan Laurie, at linux.conf.au 2013:



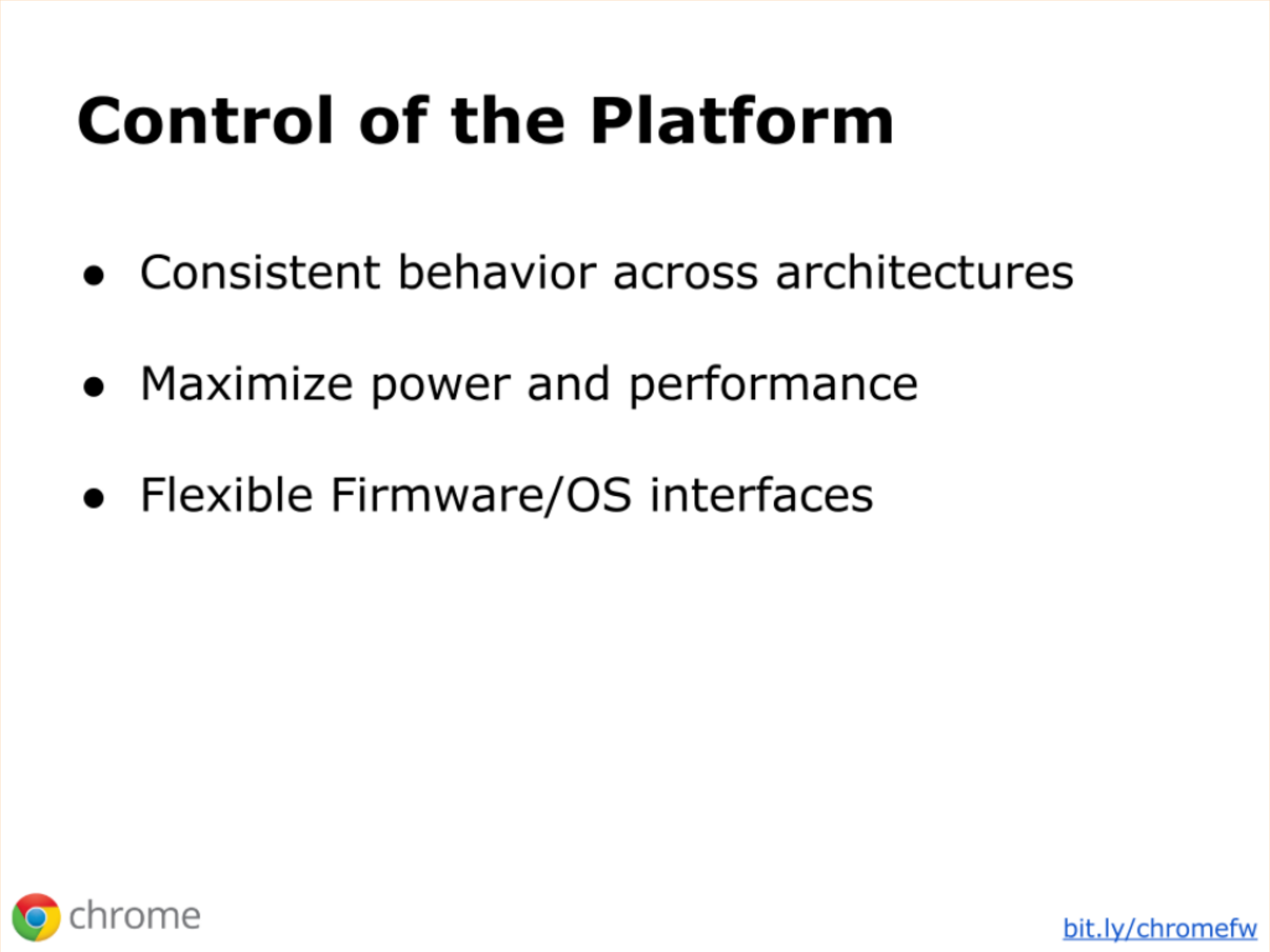
Knowledge of the Platform

- Firmware is hard
- Bugs will be found
- Time is money

chrome

bit.ly/chromefw

Duncan Laurie, at linux.conf.au 2013:



Control of the Platform

- Consistent behavior across architectures
- Maximize power and performance
- Flexible Firmware/OS interfaces

chrome

bit.ly/chromefw

Duncan Laurie, at linux.conf.au 2013:

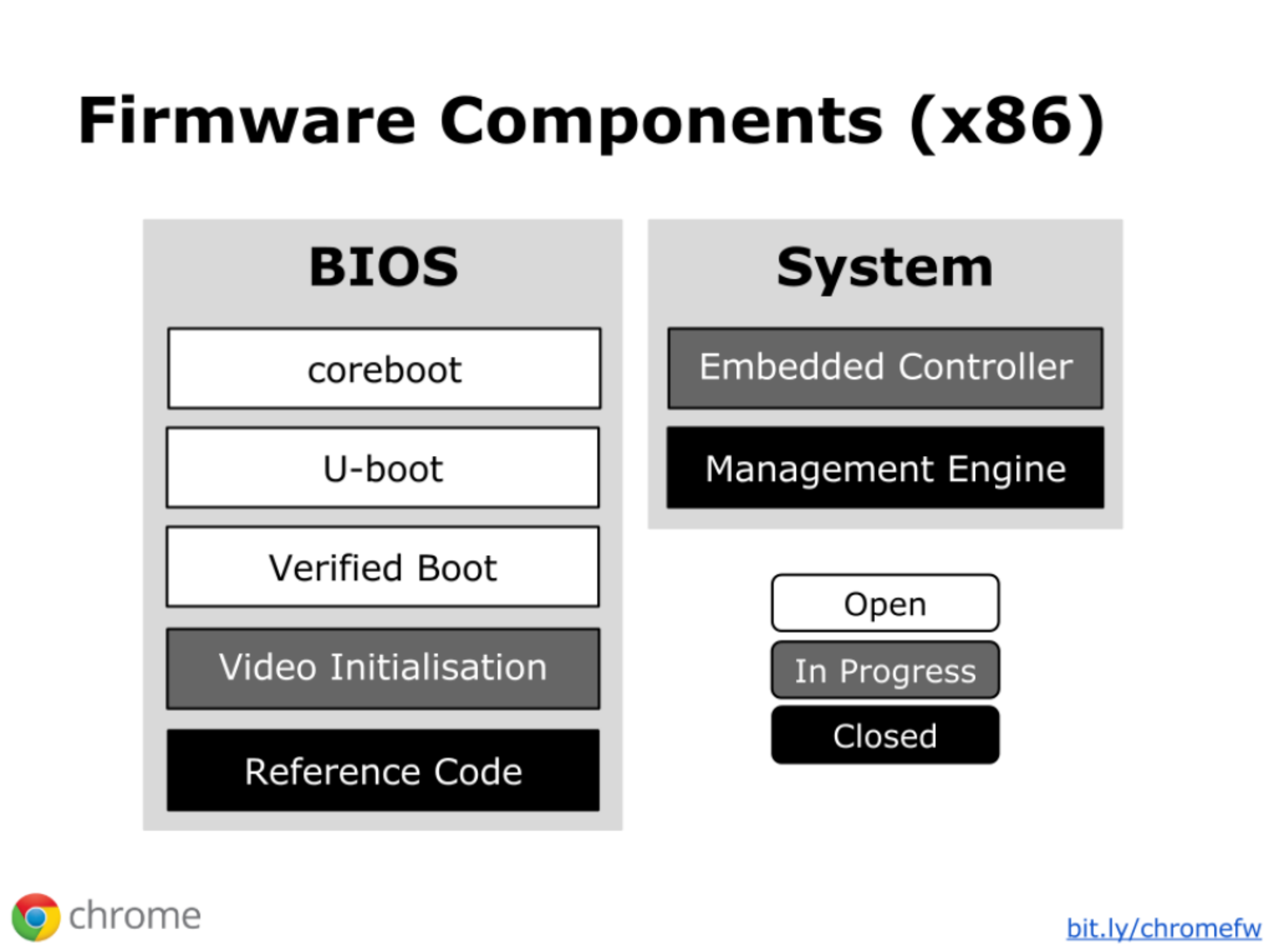
Firmware Components

Chrome OS Verified Boot

chrome

bit.ly/chromefw

Duncan Laurie, at linux.conf.au 2013:



Firmware Components (x86)

BIOS

- coreboot
- U-boot
- Verified Boot
- Video Initialisation
- Reference Code

System

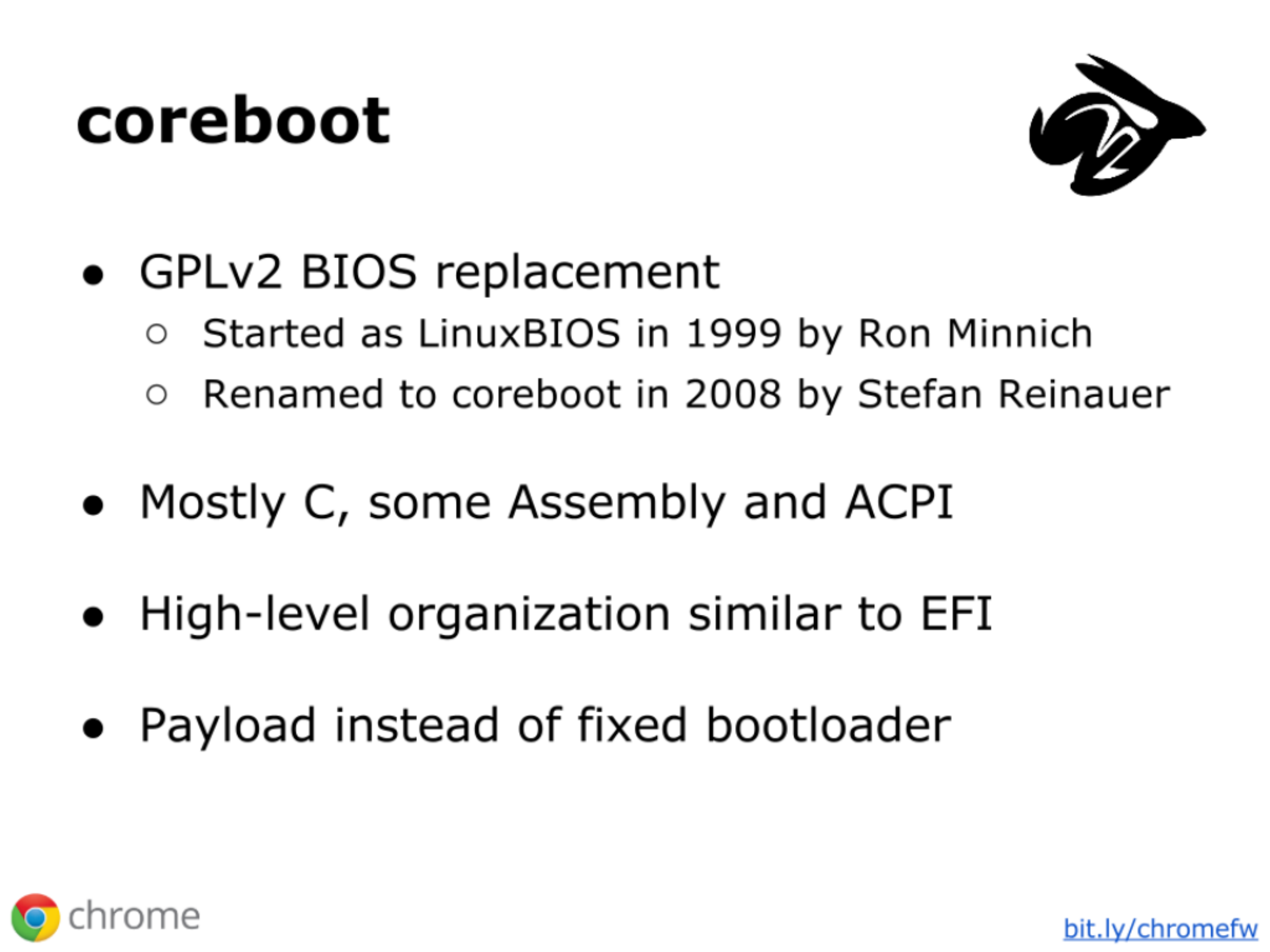
- Embedded Controller
- Management Engine

- □ Open
- ▨ In Progress
- ■ Closed

chrome

bit.ly/chromefw

Duncan Laurie, at linux.conf.au 2013:



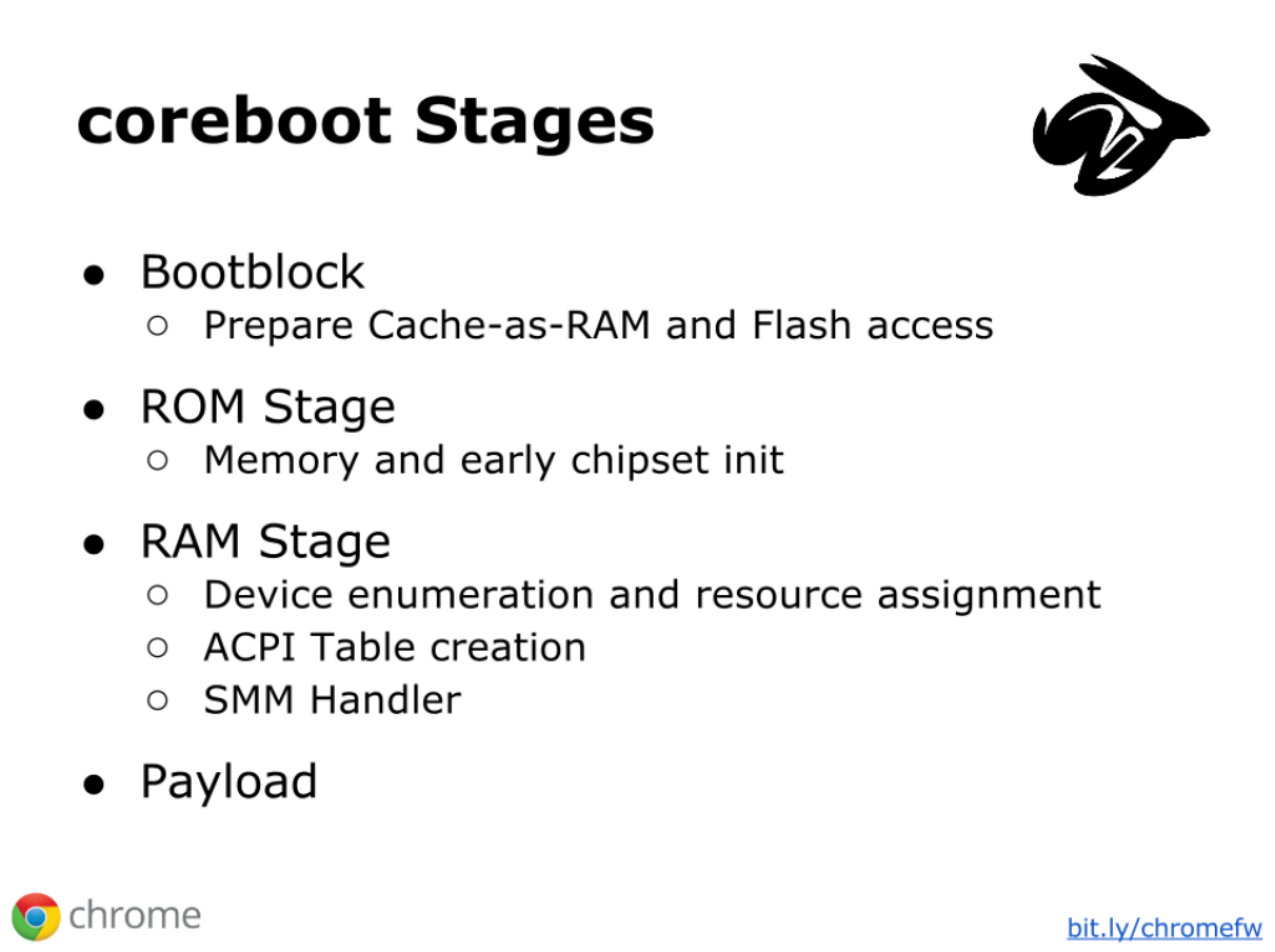
coreboot

- GPLv2 BIOS replacement
  - Started as LinuxBIOS in 1999 by Ron Minnich
  - Renamed to coreboot in 2008 by Stefan Reinauer
- Mostly C, some Assembly and ACPI
- High-level organization similar to EFI
- Payload instead of fixed bootloader

chrome

bit.ly/chromefw

Duncan Laurie, at linux.conf.au 2013:



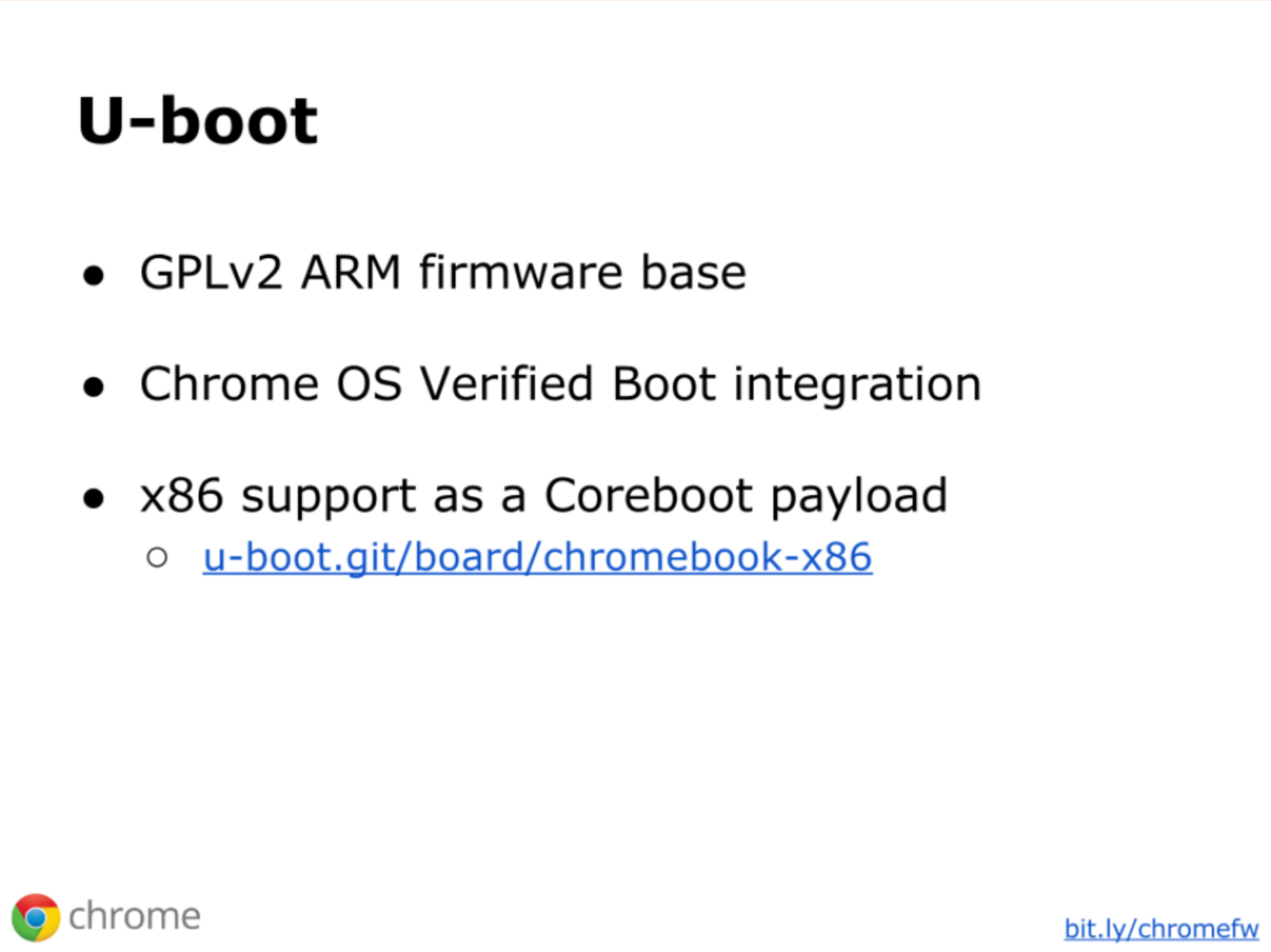
coreboot Stages

- Bootblock
  - Prepare Cache-as-RAM and Flash access
- ROM Stage
  - Memory and early chipset init
- RAM Stage
  - Device enumeration and resource assignment
  - ACPI Table creation
  - SMM Handler
- Payload

chrome

bit.ly/chromefw

Duncan Laurie, at linux.conf.au 2013:



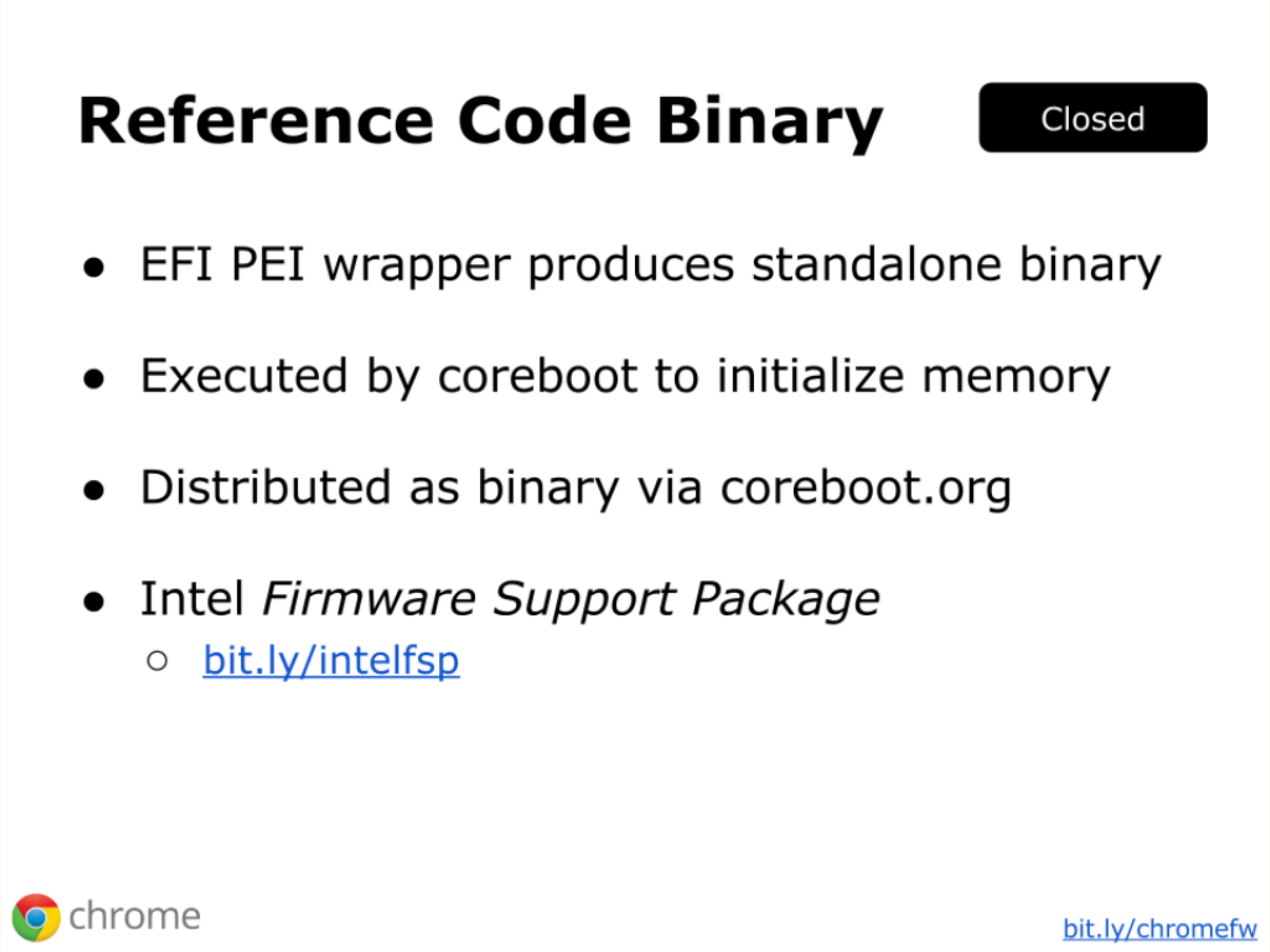
U-boot

- GPLv2 ARM firmware base
- Chrome OS Verified Boot integration
- x86 support as a Coreboot payload
  - u-boot.git/board/chromebook-x86

chrome

bit.ly/chromefw

Duncan Laurie, at linux.conf.au 2013:



Reference Code Binary

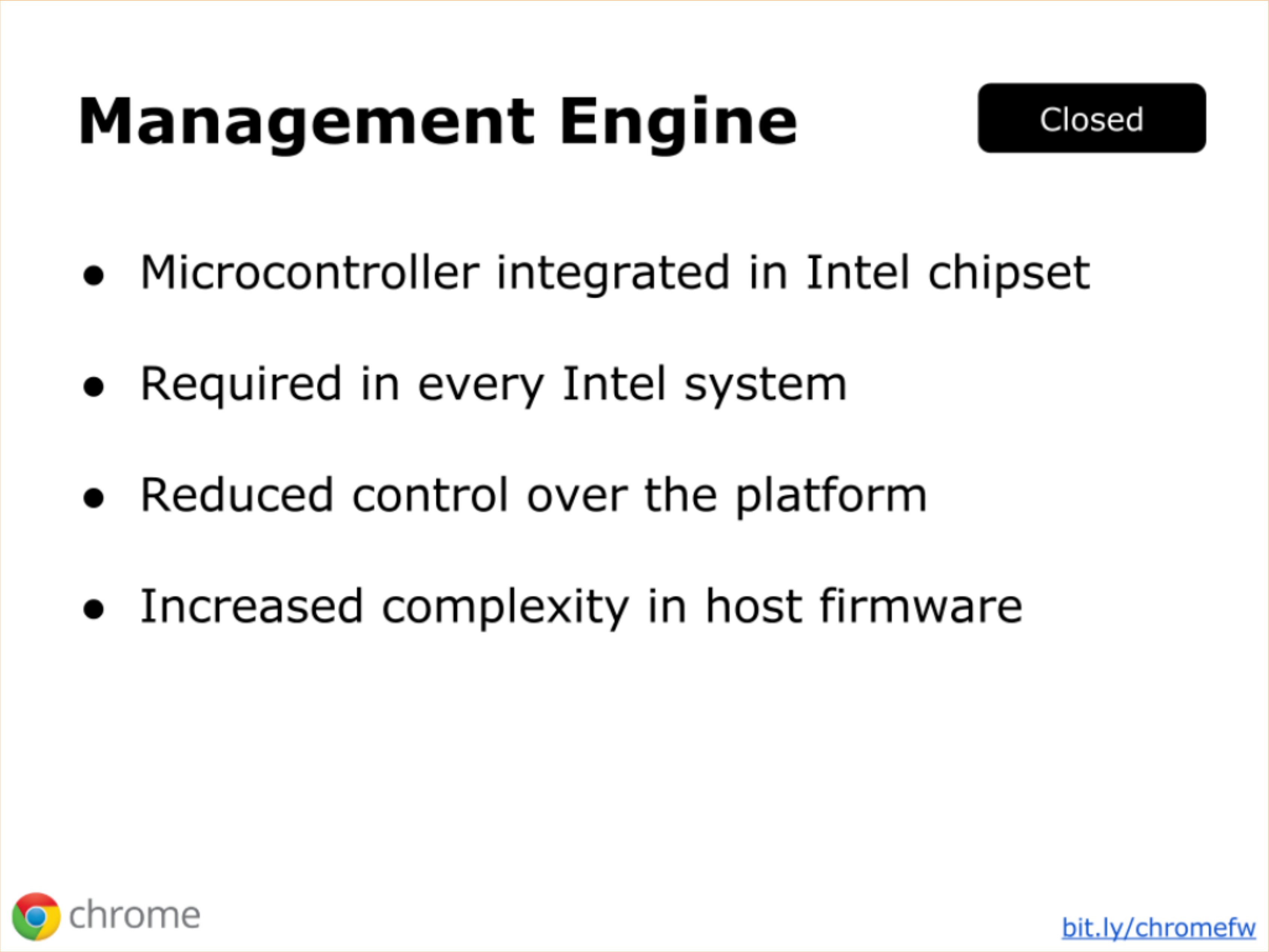
Closed

- EFI PEI wrapper produces standalone binary
- Executed by coreboot to initialize memory
- Distributed as binary via coreboot.org
- Intel *Firmware Support Package*
  - bit.ly/intelfsp

chrome

bit.ly/chromefw

Duncan Laurie, at linux.conf.au 2013:



Management Engine

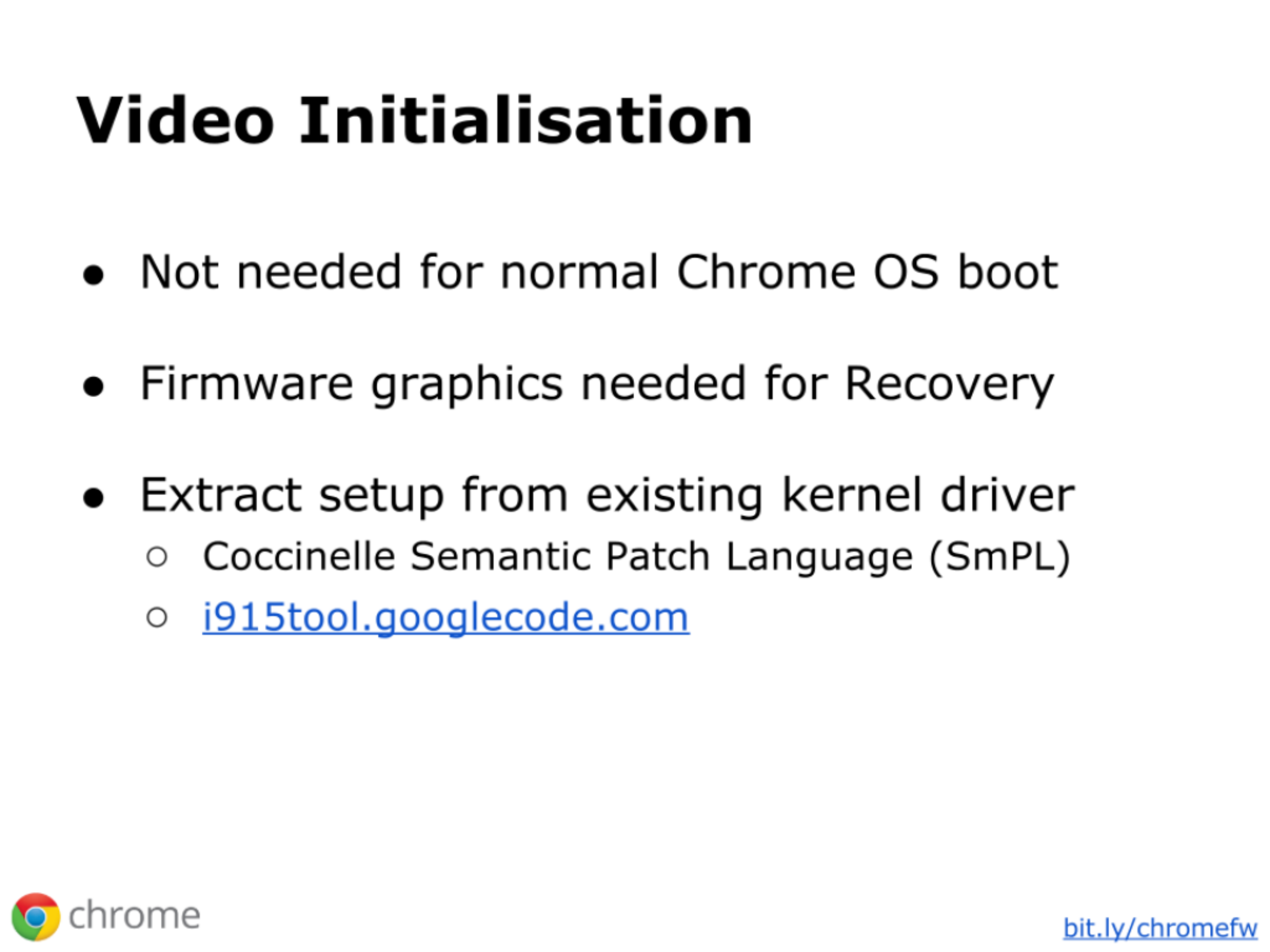
Closed

- Microcontroller integrated in Intel chipset
- Required in every Intel system
- Reduced control over the platform
- Increased complexity in host firmware

chrome

bit.ly/chromefw

Duncan Laurie, at linux.conf.au 2013:



Video Initialisation

- Not needed for normal Chrome OS boot
- Firmware graphics needed for Recovery
- Extract setup from existing kernel driver
  - Coccinelle Semantic Patch Language (SmPL)
  - i915tool.googlecode.com

chrome

bit.ly/chromefw

Agenda

Intro / Background

Chrome OS Firmware

Development System

Preparing a Test System

Hands On

What Next?

Verified Boot

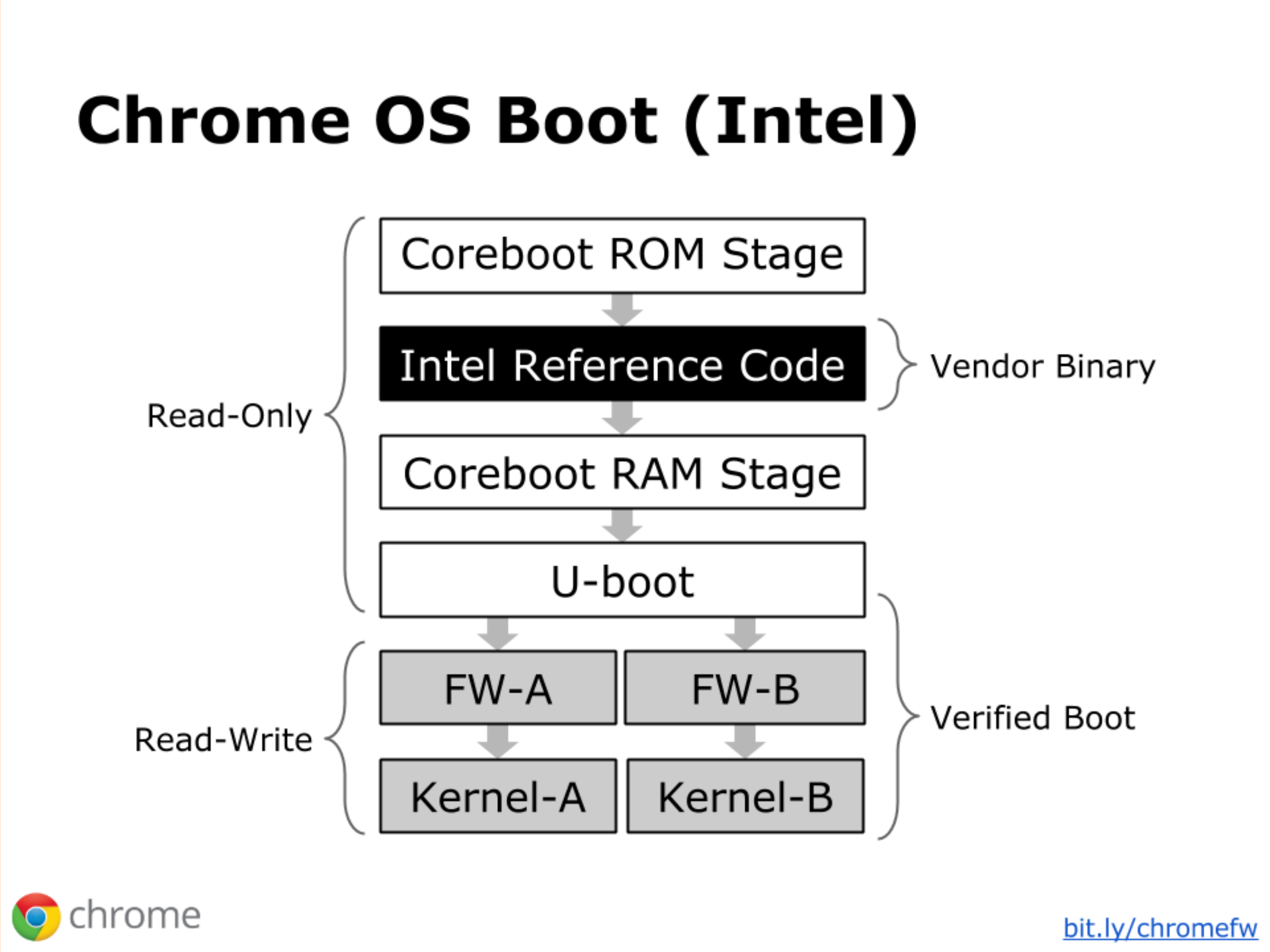
- Part of the BIOS flash is read-only
- The read-only BIOS runs first*
- The read-only BIOS verifies the read-write BIOS, then executes it
- The read-write BIOS verifies the kernel, then executes it
- The kernel verifies the rootfs as each block comes off the drive.
- If anything fails, it reboots into Recovery mode (read-only BIOS again).

*Okay, the ME runs before the BIOS gets a chance. But that's a separate thing.

Coreboot

- Coreboot is the first part of the read-only BIOS
- Its payload is U-Boot, which does the verification of the read-write BIOS
- The read-write BIOS is *just* U-Boot (but that's changing)
- Because it's read-only:
  - It can't be updated
  - It had better work
  - Playing with it is tricky and dangerous
  - Hence this class...

Duncan Laurie, at linux.conf.au 2013:



Chrome OS Boot (Intel)

Read-Only

Coreboot ROM Stage

Intel Reference Code

Vendor Binary

Coreboot RAM Stage

U-boot

Read-Write

FW-A

FW-B

Kernel-A

Kernel-B

Verified Boot

bit.ly/chromefw

FMAP

- https://code.google.com/p/flashmap
- This is just a way of identifying various sections in a ROM image.
- We have a number of different sections in the Chrome OS BIOS
- You don't *have* to know anything about them, but it helps, especially if you want to hack on coreboot for Chrome OS, not just coreboot by itself
- They're not always 100% correct, though.

```
$ dump_fmap -h link_bios.rom
# name                     start       end         size
SI_BIOS                    00200000    00800000    00600000
  WP_RO                      00600000    00800000    00200000
    RO_SECTION                 00610000    00800000    001f0000
      BOOT_STUB                  00700000    00800000    00100000
      GBB                        00611000    00700000    000ef000
      RO_FRID_PAD                00610840    00611000    000007c0
      RO_FRID                    00610800    00610840    00000040
      FMAP                       00610000    00610800    00000800
    RO_UNUSED                  00604000    00610000    0000c000
    RO_VPD                     00600000    00604000    00004000
  RW_LEGACY                  00400000    00600000    00200000
  RW_UNUSED                  003fe000    00400000    00002000
  RW_VPD                     003fc000    003fe000    00002000
  RW_ENVIRONMENT             003f8000    003fc000    00004000
  RW_SHARED                  003f4000    003f8000    00004000
    VBLOCK_DEV                 003f6000    003f8000    00002000
    SHARED_DATA                003f4000    003f6000    00002000
  RW_ELOG                    003f0000    003f4000    00004000
  RW_MRC_CACHE               003e0000    003f0000    00010000
  RW_SECTION_B               002f0000    003e0000    000f0000
    RW_FWID_B                  003dffc0    003e0000    00000040
    EC_RW_B                    003c0000    003dffc0    0001ffc0
    FW_MAIN_B                  00300000    003c0000    000c0000
    VBLOCK_B                   002f0000    00300000    00010000
  RW_SECTION_A               00200000    002f0000    000f0000
    RW_FWID_A                  002effc0    002f0000    00000040
    EC_RW_A                    002d0000    002effc0    0001ffc0
    FW_MAIN_A                  00210000    002d0000    000c0000
    VBLOCK_A                   00200000    00210000    00010000
SI_ALL                     00000000    00200000    00200000
  SI_ME                      00001000    00200000    001ff000
  SI_DESC                    00000000    00001000    00001000
$
```

```
$ dump_fmap -h parrot_bios.rom
# name                     start       end         size
SI_BIOS                    00200000    00800000    00600000
  WP_RO                      00400000    00800000    00400000
    RO_SECTION                 00610000    00800000    001f0000
      BOOT_STUB                  00700000    00800000    00100000
      GBB                        00611000    00700000    000ef000
      RO_FRID_PAD                00610840    00611000    000007c0
      RO_FRID                    00610800    00610840    00000040
      FMAP                       00610000    00610800    00000800
    RO_UNUSED                  00604000    00610000    0000c000
    RO_VPD                     00600000    00604000    00004000
    RO_SI_ALL                  00400000    00600000    00200000
      RO_SI_ME                   00401000    00600000    001ff000
      RO_SI_DESC                 00400000    00401000    00001000
  RW_UNUSED                  003fe000    00400000    00002000
  RW_VPD                     003fc000    003fe000    00002000
  RW_ENVIRONMENT             003f8000    003fc000    00004000
  RW_SHARED                  003f4000    003f8000    00004000
    VBLOCK_DEV                 003f6000    003f8000    00002000
    SHARED_DATA                003f4000    003f6000    00002000
  RW_ELOG                    003f0000    003f4000    00004000
  RW_MRC_CACHE               003e0000    003f0000    00010000
  RW_SECTION_B               002f0000    003e0000    000f0000
    RW_FWID_B                  003dffc0    003e0000    00000040
    FW_MAIN_B                  00300000    003dffc0    000dffc0
    VBLOCK_B                   002f0000    00300000    00010000
  RW_SECTION_A               00200000    002f0000    000f0000
    RW_FWID_A                  002effc0    002f0000    00000040
    FW_MAIN_A                  00210000    002effc0    000dffc0
    VBLOCK_A                   00200000    00210000    00010000
SI_ALL                     00000000    00200000    00200000
  SI_ME                      00001000    00200000    001ff000
  SI_DESC                    00000000    00001000    00001000
$
```

For example...

- Link has 2M of read-only BIOS
- Parrot has 4M of read-only BIOS

Link uses that extra 2M of read-write flash to hold a copy of SeaBIOS.

Parrot uses it for a backup read-only copy of the ME firmware. Although I don't think it's actually present...

Agenda

Intro / Background

Chrome OS Firmware

Development System

Preparing a Test System

Hands On

What Next?

Basic system

- You'll need a 64-bit Linux distro
- I'm using Ubuntu 12.04.2 LTS. The package names may vary in other distros.
- Add some generally useful packages:

```
sudo apt-get install \
    git-core gitk git-gui subversion curl
```

Flashrom

- We'll need this ~~when~~ if things go wrong
- Download the latest tarball from http://www.flashrom.org
- Install the prerequisite packages

```
sudo apt-get install \
   build-essential zlib1g-dev libftdi-dev pciutils-dev
```

- Build it

```
make CONFIG_DEDIPROG=yes
sudo make install
```

- Note: Chromebooks have their own copy of flashrom too. That is built slightly differently from the upstream.

Coreboot

```
$ sudo apt-get install libncurses5-dev m4 bison flex iasl
$ git clone http://review.coreboot.org/p/coreboot.git
$ cd coreboot
$ make menuconfig
$ make
```

- If it works, it will create a file named

  ```
  build/coreboot.rom
  ```

- I got errors the first time. This fixed it:

```
$ make clean
$ make crossgcc
$ make
```

"make menuconfig" selections

General Setup

- **Allow use of binary-only repository**

Mainboard

- **Vendor Google**
- **Model Parrot**

Chipset

- **Add a System Agent Binary**
- **Filename: 3rdparty/northbridge/intel/sandybridge/systemagent-r6.bin**

VGA BIOS

- **Add a VGA BIOS**
- **Filename: 3rdparty/mainboard/google/parrot/snm_2130_coreboot.bin**

make menuconfig (continued)

Console

- **Disable Serial port console output**
- **Enable USB 2.0 EHCI debug dongle support**
- **Enable Send console output to a CBMEM buffer**

Save and Exit

Coreboot alternate source

- For Chrome OS, changes are **rapid**
- If you're building for Chrome OS, you may want to pull from the Chromium OS repo

```
$ git remote add cros-coreboot \
https://git.chromium.
org/git/chromiumos/third_party/coreboot

$ git branch --track cros \ remotes/cros-
coreboot/chromeos-2013.04

$ git checkout cros
$ make menuconfig
$ make
```

Agenda

Intro / Background

Chrome OS Firmware

Development System

Preparing a Test System

Hands On

What Next?

Which Chromebook?

- To date, 10 Chrome OS devices have shipped
- Some don't use coreboot
- Some aren't x86-based
- Some use flush-mounted flash chips (so you can't clip on to them)
- Costs and features vary

Acer C7 Chromebook (aka "Parrot")

- Minuses
  - Royal pain to open up
  - A little slow to boot

- Pluses
  - Relatively recent design
  - 8M flash, plenty of room to experiment
  - Flash chip can be clipped onto
  - Huge (for Chromebooks) 320GB hard drive
  - **Under $200**

Back up the original BIOS!

- Get a root shell (Developer Mode)
- Read your BIOS using flashrom to get the VPD section, GBB bitmaps, etc.
- Extract the BIOS (and other stuff) from the shellball to get the ME firmware.
- Copy the two BIOSes (orig_bios.bin and bios.bin) **SOMEWHERE ELSE**.
- If you want to restore everything exactly, you'll have to assemble the original image from those two parts.

From the root shell:

```
# mkdir /tmp/ho
# cd /tmp/ho

# flashrom -p internal:bus=spi -r orig_bios.bin

# chromeos-firmwareupdate --sb_extract .

# scp orig_bios.bin bios.bin USER@HOST:
```

- Or you can just read the entire BIOS flash using a debugger. We'll go over that later.

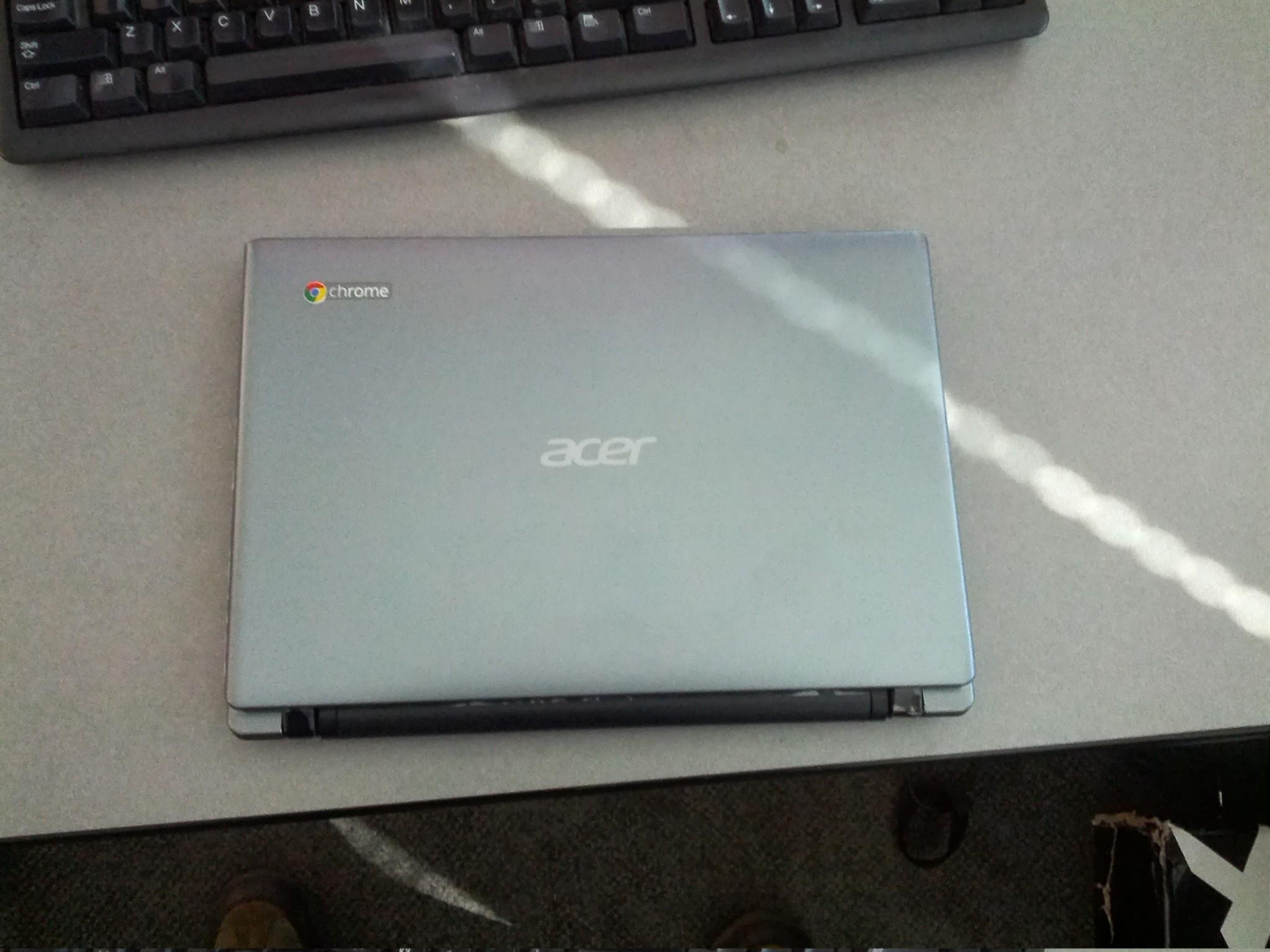
Void your warranty

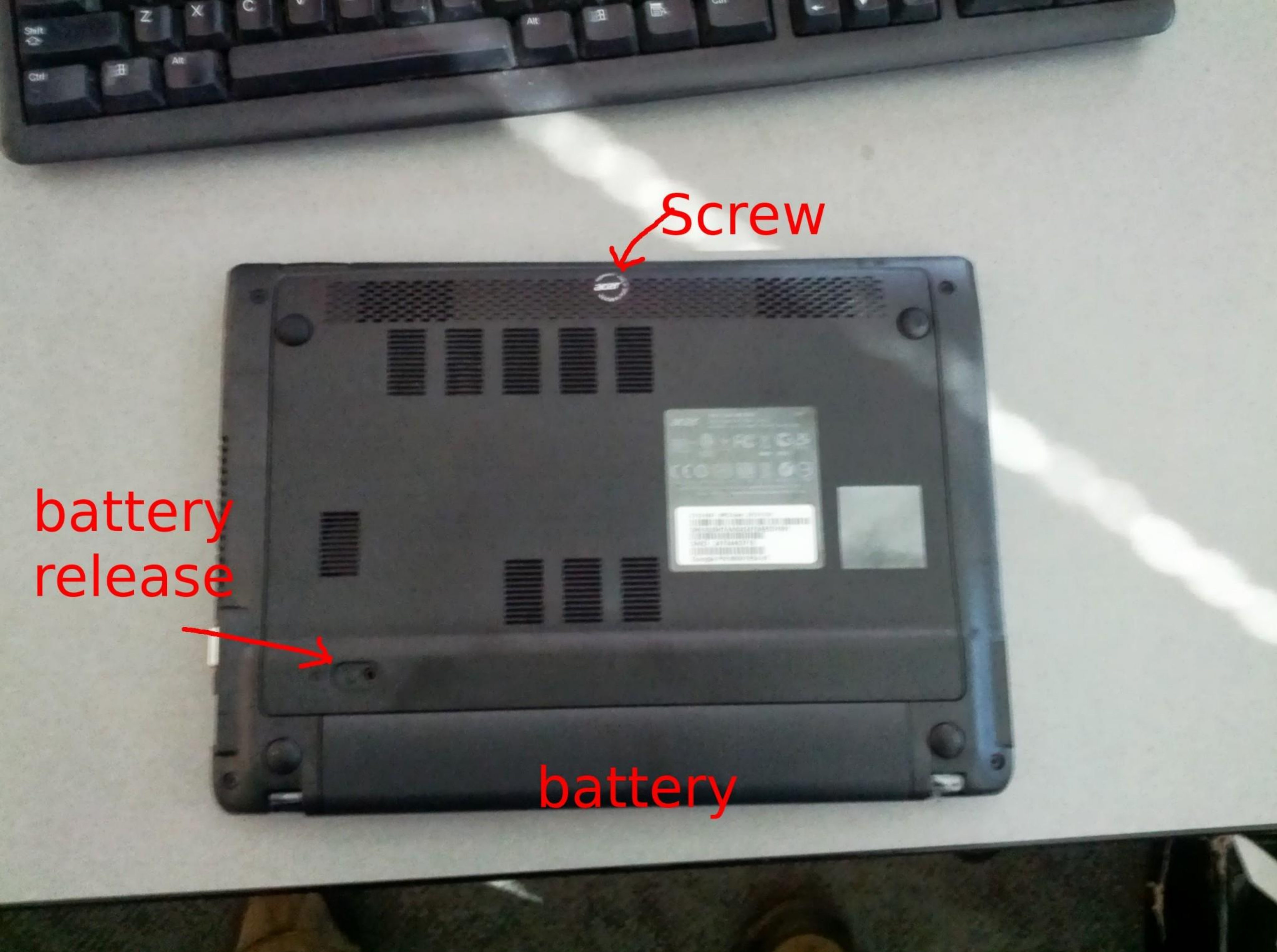
- The only way to disable the BIOS write protection is to open up the machine.
- Unplug the charger and remove the battery first, just to be safe
- To take the back off, remove the single screw under the Warranty-Voiding sticker.
- Slide the back cover away from the battery side about 1/8", and it should lift off.

FIXME: need better photos!

Yes, all the photos are **horrible**. I only had my phone, in bad lighting. I'll replace them with better ones as soon as I can.

- Bill


chrome
acer



Screw
battery release
battery

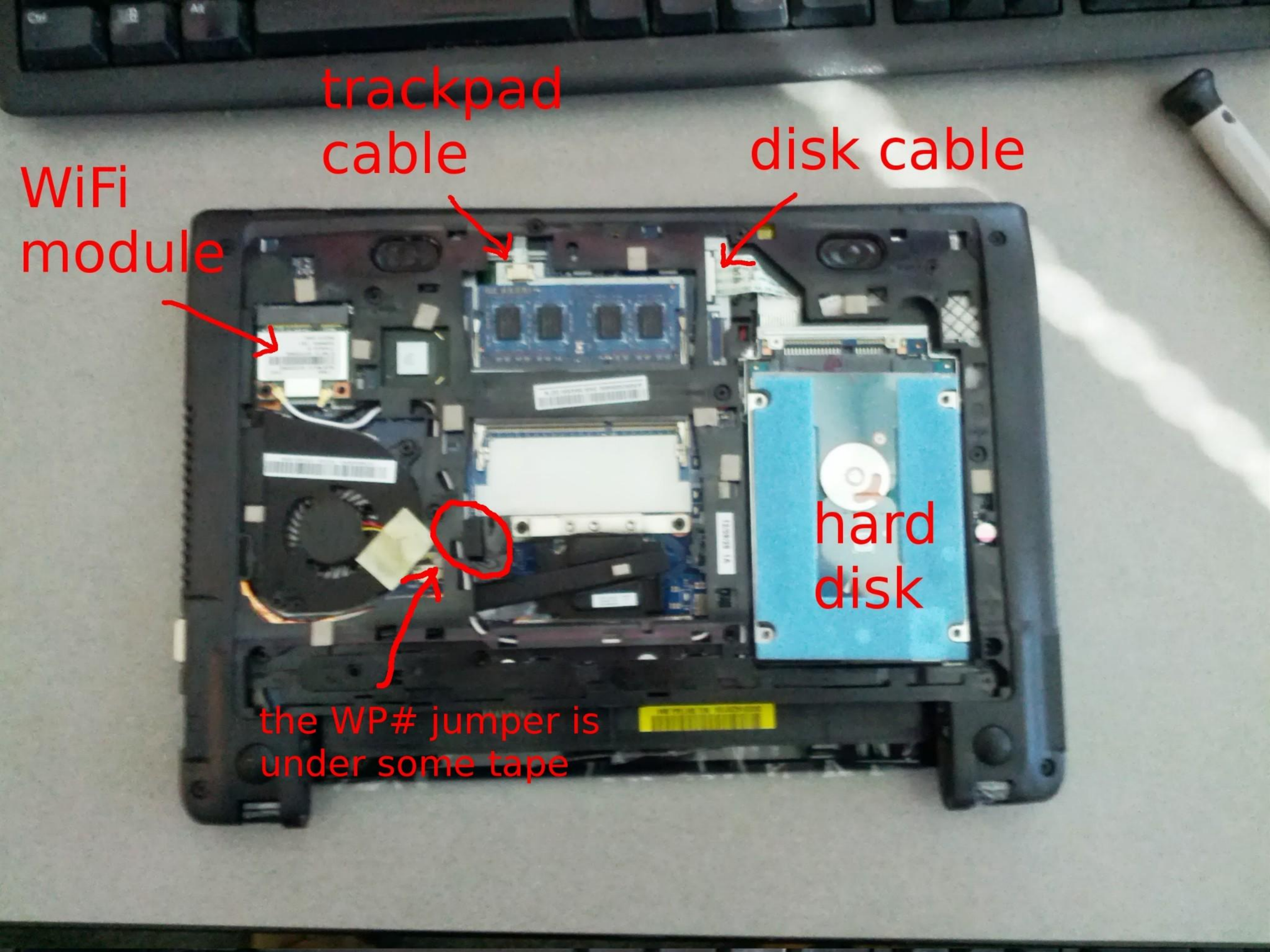
BIOS flash write protection

- First, the SPI EEPROM status register sets a write protect range. EEPROM in this range cannot be erased or written. The --wp-range option to flashrom changes this setting.

- Second, the SPI EEPROM status register can also protect the status register itself from being changed. The --wp-enable and --wp-disable options to flashrom change this setting (which makes it kind of pointless, IMHO).

BIOS flash write protection

- Third, if the **WP#** pin on the SPI EEPROM chip is asserted, the chip pays attention to the status register protection bit.
- When **WP#** is asserted, the protection bit can be set, but cannot be cleared.
- If **WP#** is deasserted, flashrom can write to the status register even if --wp-enable is set.
- The state of **WP#** is controlled by a physical connection. Each model of Chromebook is slightly different - on Parrot, it's a jumper.


trackpad cable
disk cable
WiFi module
hard disk
the WP# jumper is under some tape

Disable Write Protection

- First, make a note of the current settings. The range varies among Chromebooks.
- Connect the charger, turn it on, and get a root shell.
- flashrom will display the settings:

```
localhost ~ # flashrom -p internal:bus=spi --wp-status
WP: status: 0x98
WP: status.srp0: 1
WP: write protect is enabled.
WP: write protect range: start=0x00400000, len=0x00400000
localhost ~ #
```

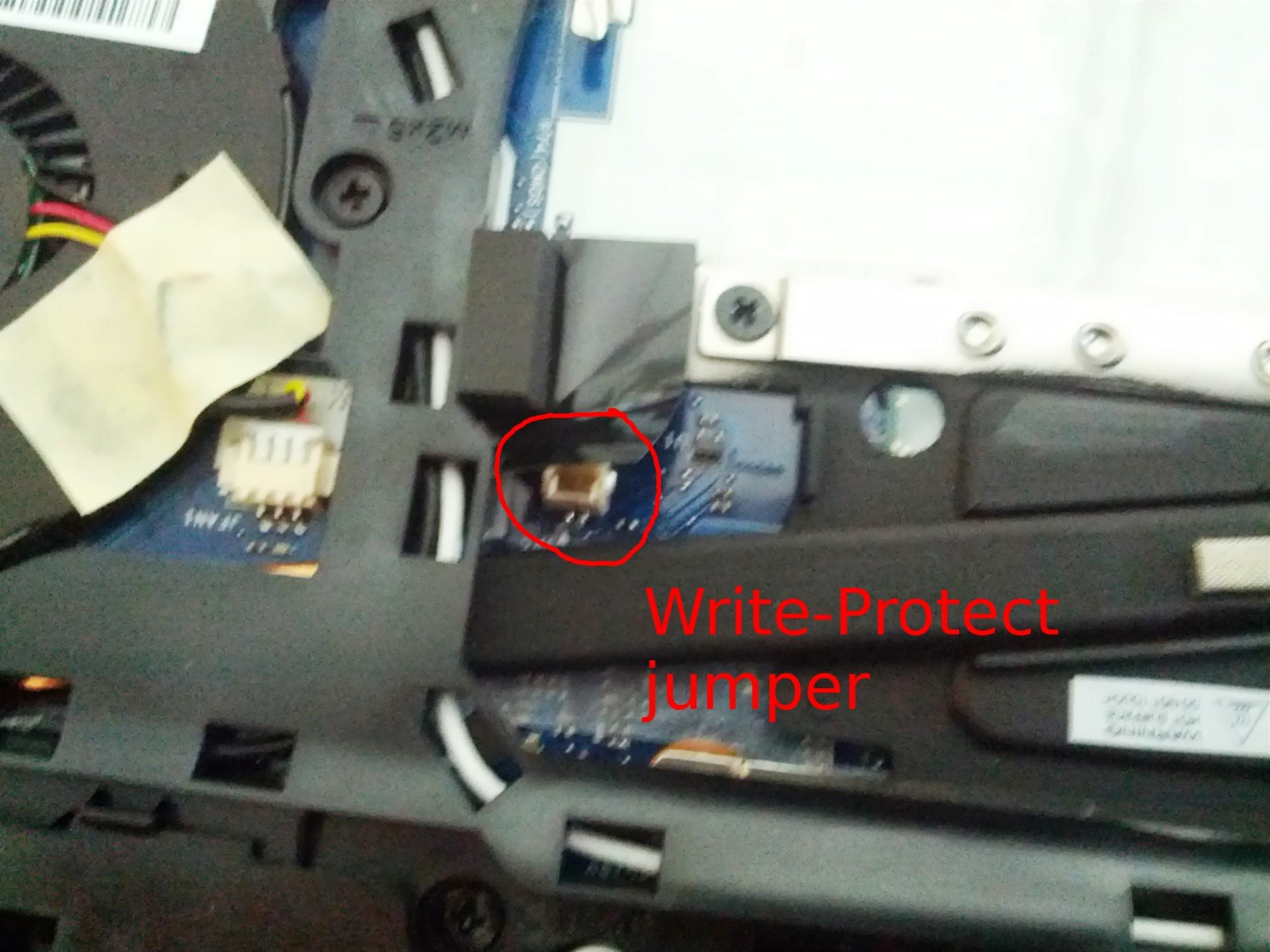
Disable Write Protection

- Check WP# using the crossystem command
- The last two lines show the state at boot and the current value

```
wpsw_boot      = 1
wpsw_cur       = 1
```

- Put a screw or paperclip into the jumper and wiggle it around while running crossystem until you see

```
wpsw_boot      = 1
wpsw_cur       = 0
```



Write-Protect jumper

Disable Write Protect

- Once **WP#** is deasserted, run

```
flashrom -p internal:bus=spi --wp-disable
flashrom -p internal:bus=spi --wp-range 0 0
```

- Verify that it's disabled with

```
flashrom -p internal:bus=spi --wp-status
```

Reenable Write Protect (but not now)

- You can stop fiddling with the jumper
- Just don't change the value with flashrom, and it will stay unprotected
- If you do want to reenable it, just run

```
flashrom -p internal:bus=spi \
  --wp-range 0x00400000 0x00400000
flashrom -p internal:bus=spi --wp-enable
```

- You don't need to disable **WP#** to enable write protection. It's a one-way operation.

Now you're ready to brick your Chromebook

- Copy your newly-built `coreboot.rom` file to the Chromebook

- Replace the BIOS firmware

```
flashrom -p internal:bus=spi -w coreboot.rom
```

- And reboot

Huh

- It didn't work, did it?
- Nuts.
- Now what?

What went wrong?

- You chose an invalid setting in the coreboot configuration
- You're missing some vital binary blobs
- Coreboot has a bug in it
- A nearly infinite number of other things

- Firmware is tricky like that

How can we make it work again?

- We need to use an external programmer to replace the borked BIOS with a good one.
  - I've only used the Dediprog SF100 (http://www.dediprog.com)
  - Other solutions might also work (Bus Pirate, etc.)

- Now we'll **really** have to take things apart

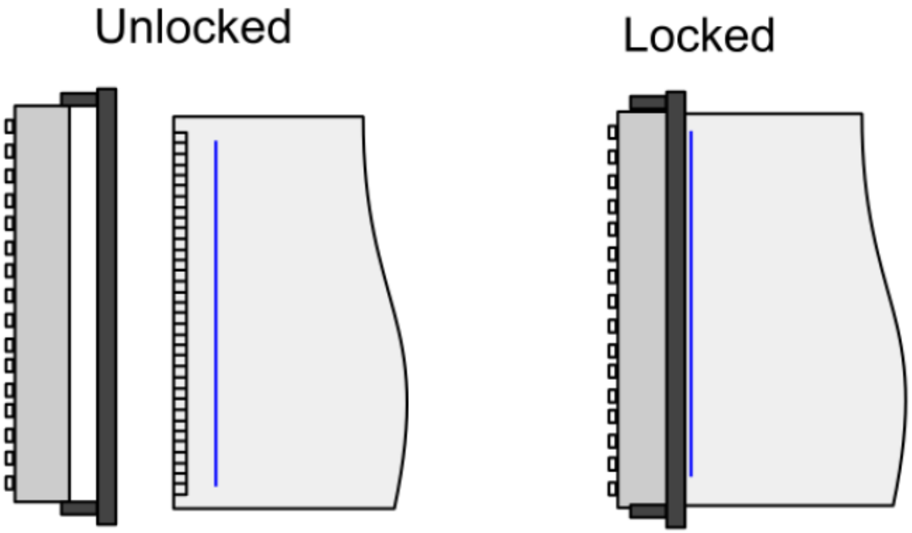
A brief digression...

- Some Chromebooks (Parrot, for example) use flash chips that are easy to clip on to.
- Others use low-profile or surface-mount chips that are much trickier, or that may require soldering.
- Some models use custom ribbon cables and circuit boards to expose JTAG and other signals. We don't even try to use a Dediprog on those.
- You should probably check some teardown sites before you buy one to play with.

Disconnect the hard drive

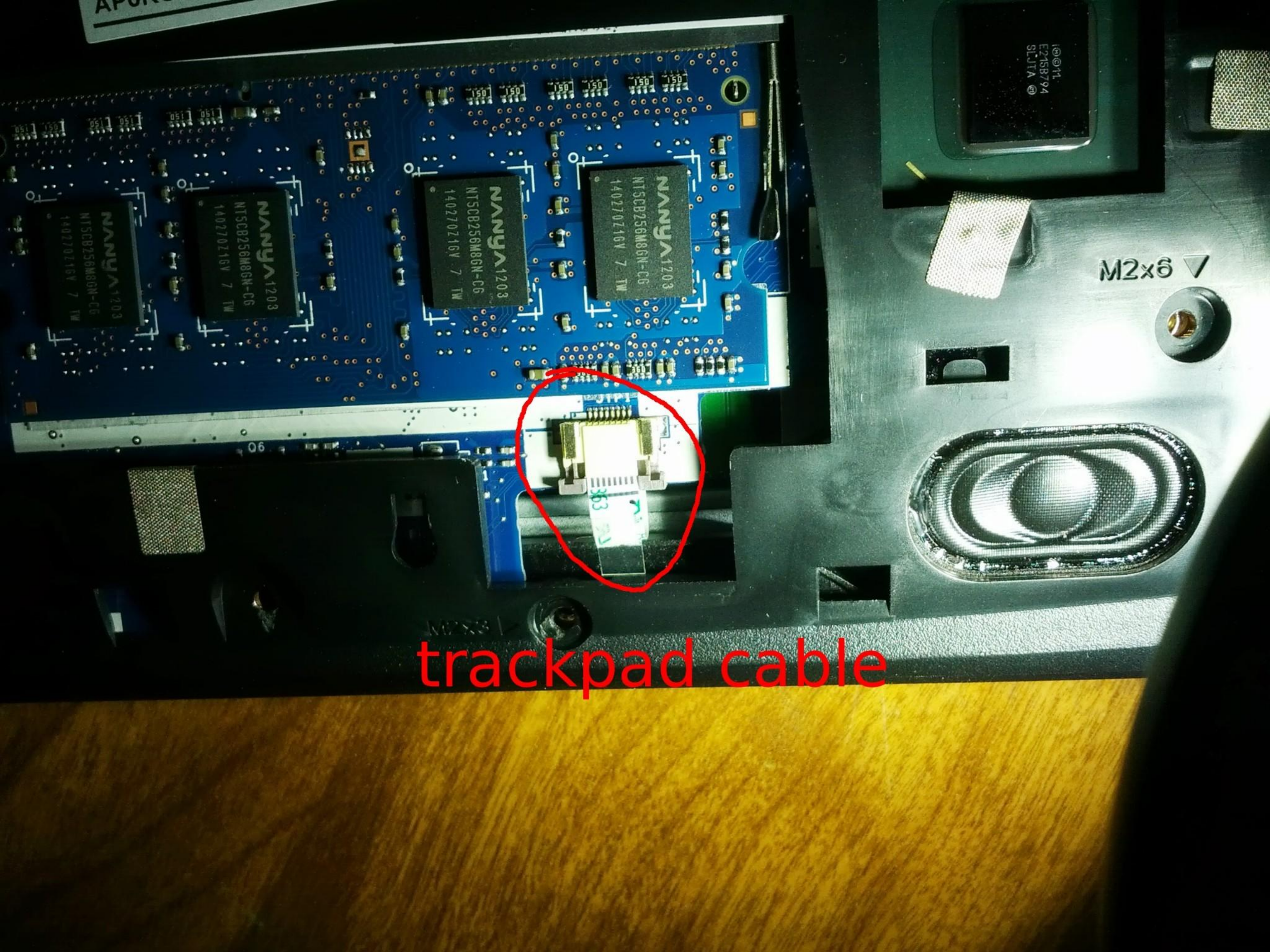
- You've probably noticed that the hard drive tends to flop around a lot.
- Unlock the ribbon cable and remove the drive.
- The ribbon connector has a bar that moves towards the cable to unlock, or towards the connector to lock.
- The cable has a line painted on it to help you tell when it's fully inserted.

How the connector works



Trackpad cable

- The trackpad cable uses the same type of connector as the hard drive.
- It's easier to just leave it connected, but it's pretty short.
- Be very careful not to yank it out accidentally.



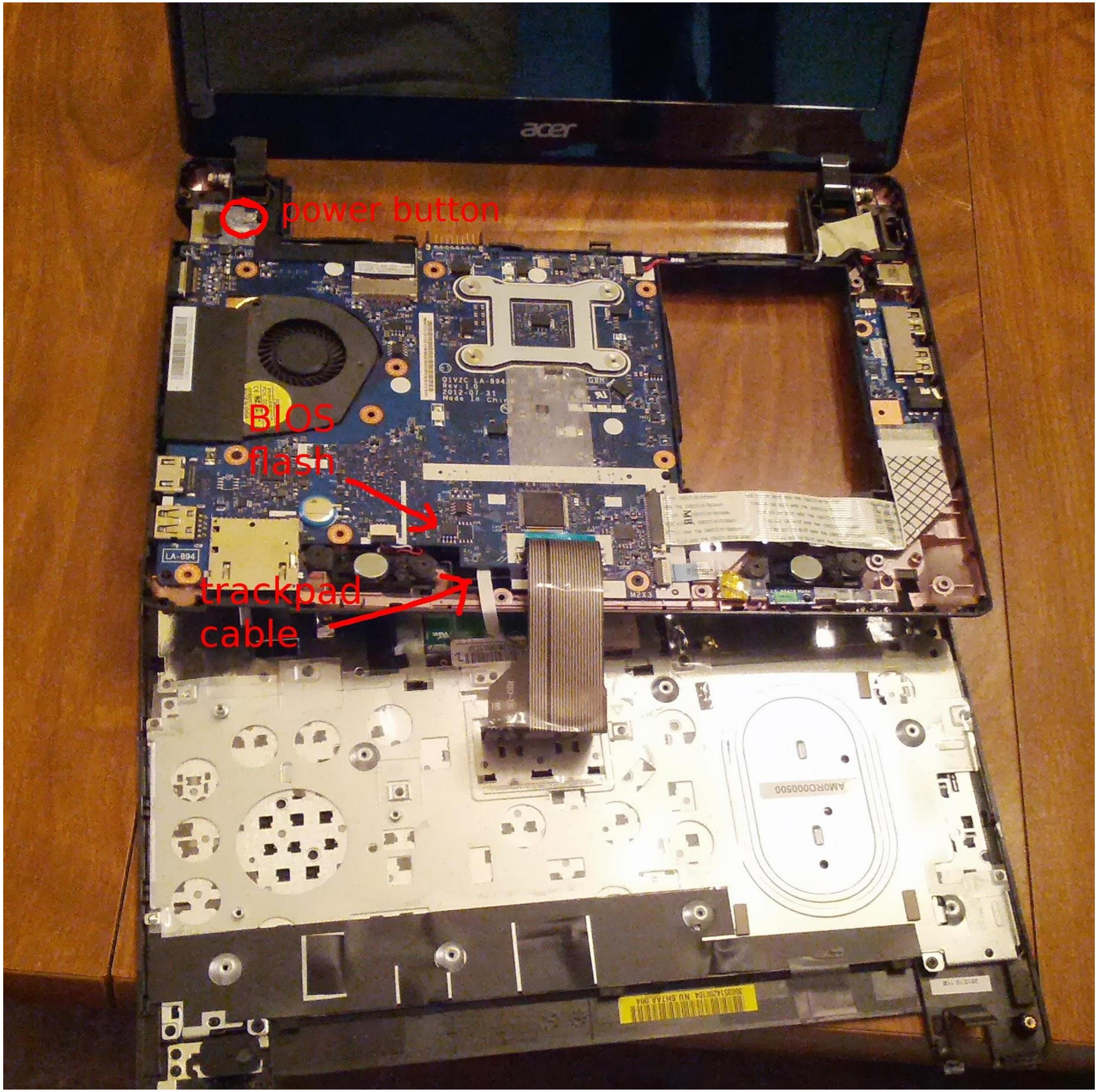
trackpad cable

Getting at the flash chip

- The BIOS flash chip is conveniently located on the **other** side of the motherboard.
- There are about 18 screws to remove.
- There are at least four tricky ones:
  - There's one tiny screw near the edge that doesn't look big enough to matter. It does.
  - There are two on the fan mount.
  - The screw that holds the WiFi module in has to be removed also. That was my favorite.

Getting at the flash chip

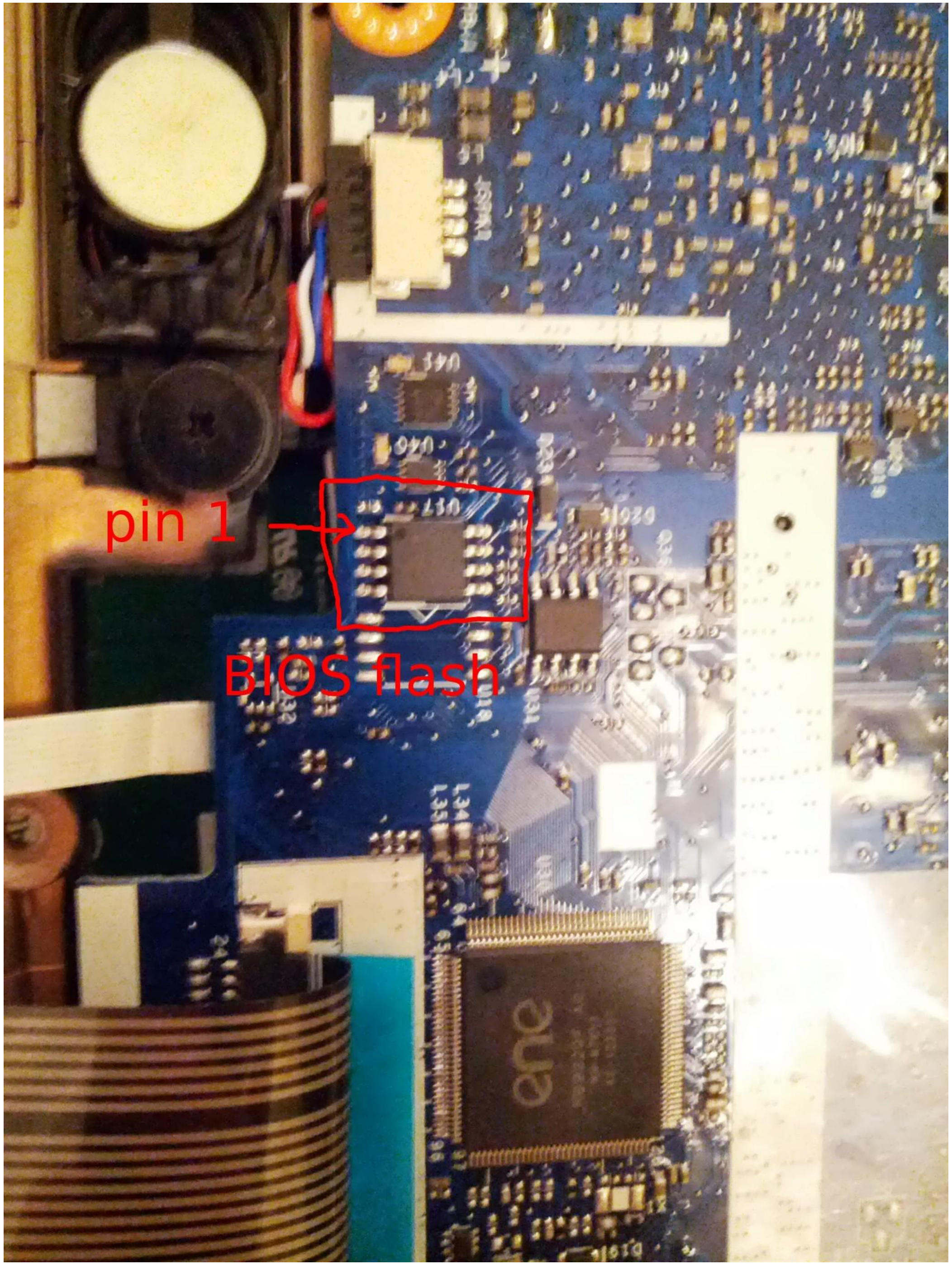
- Once all the screws are removed you can **carefully** and **gently** pry the edges apart.
- There are lots of tiny plastic catches, all along the edges. Patience and a sharp screwdriver are required.
- If the two halves are not separating easily, you've probably missed a screw.
- There are still some ribbon cables attaching the keyboard to the motherboard. Rotate the keyboard underneath once it's free.

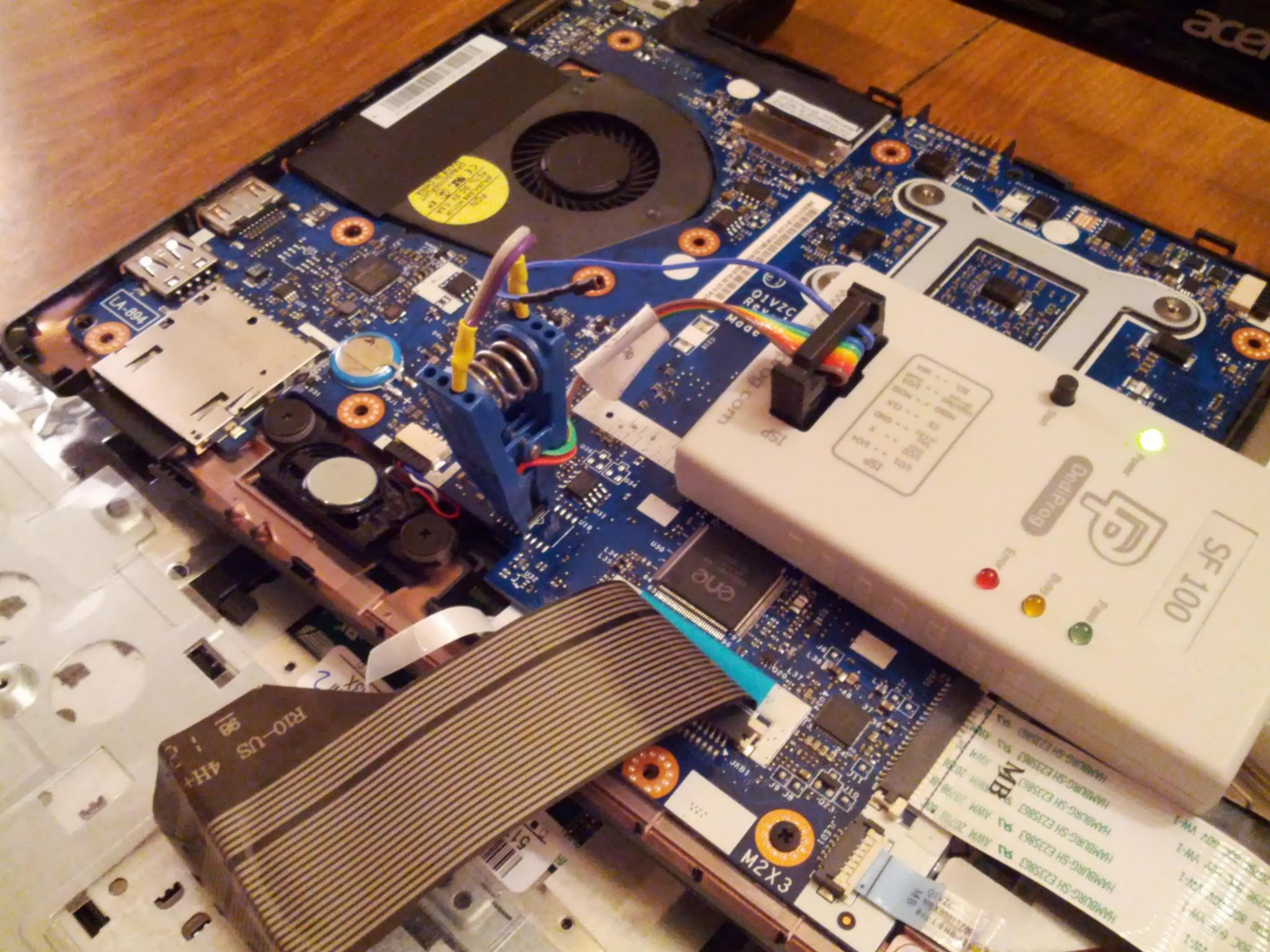

power button
BIOS flash
trackpad cable
acer

Reflash the BIOS

- Use the Dediprog to put a valid BIOS back on the system.
- You can use flashrom on your development machine to do that.

```
$ sudo flashrom -p dediprog -w bios.bin
```


pin 1
BIOS flash
ene


acer
LA-894
DediProg
SF 100
ISP
Start
Power
Error
Busy
Pass
ene
M2X3
R10-US
4H+
HAMBURG-SH E235863
MB

Wait, which BIOS do I restore?

- The Chrome OS BIOS arguably has **two** read-only sections.
- This is a side-effect of Intel's Management Engine ("ME") stuff.
- The x86 CPU fetches its first instruction from high memory, so that part of the BIOS flash needs to be read-only.
- We ensure this with **WP#**.
- The FMAP region named `BOOT_STUB` contains that code (coreboot, yay).

Wait, which BIOS do I restore?

- But the ME executes its firmware before the CPU starts.
- If the ME firmware is missing or corrupted, the CPU will never come out of reset.
- We'd like the ME firmware to be read-only.
- But the ME firmware has to be located in the writable part of flash, so it can write to it at random times.
- To protect itself, the SPI controller hides the ME region from the CPU.

Wait, which BIOS do I restore?

- By the time the kernel boots, the ME's portion of the writeable BIOS flash is inaccessible to the CPU via the SPI bus.
- So when we created our backup copies from the root shell, the ME section is blank (flashrom ignores the errors and returns 0xFF).
- The "shellball", which is used to restore or update the BIOS, contains the original ME firmware.

Wait, which BIOS do I restore?

- The shellball doesn't contain your original `RO_VPD` or `GBB` sections, since those are updated during manufacturing.
- `RO_VPD` has things like part numbers that are mostly used for warranty service.
- `GBB` contains the BIOS bitmaps displayed in Developer or Recovery mode.
- Since we're writing the entire BIOS flash, we want to use the one from the shellball that has the ME firmware.

One more thing...

- The ME can still interfere with the Dediprog.
- You'll need to unplug the charger in order for the Dediprog to erase the entire BIOS flash.
- Once you've reflashed the BIOS, it should work again.

```
$ sudo flashrom -p dediprog -w bios.bin
```

How do we debug?

- What we need is a serial port.
- What we've got is ... uh...
- Because Chromebooks aren't PCs, they don't have the "standard" LPC connectors that can access the traditional UARTs.
- You'd think a mini-PCIe serial adapter in the WiFi socket would work, but it doesn't.
- We've had the most luck with USB host-to-host debugging adapters.

How do we debug

- Those are the USB equivalents of a null-modem cable. Each end sees a USB serial adapter.
- But it only works when both USB ports are powered.
- When the Chromebook is off, so is its USB port, so the development system can't see it.
- Usually, if you start minicom while `/dev/ttyUSB0` is active, it will complain when it's gone, but will still work when it comes back.

Example output from a bad BIOS

```
USB


coreboot-4.0-4428-g4 PDT 2013 starting...
Setting up static southbridge registers... done.
Disabling Watchdog reboot... done.
Setting up static northbridge registers... done.
Initializing Graphics...
Back from sandybridge_early_initialization()
SMBus controller enabled.
CPU id(206a7): Intel(R) Celeron(R) CPU 847 @ 1.10GHz
AES NOT supported, TXT NOT supported, VT supported
PCH type: NM70, device id: 1e5f, rev id 4
Intel ME early init
Intel ME firmware is ready
ME: Requested 16MB UMA
Starting UEFI PEI System Agent
REC MODE GPIO 68: 0
Read scrambler seed    0x00007d92 from CMOS 0x98
Read S3 scrambler seed 0x00004a81 from CMOS 0x9c
No FMAP found at ffe10000.
FMAP: area RW_MRC_CACHE not found
```

Agenda

Intro / Background

Chrome OS Firmware

Development System

Preparing a Test System

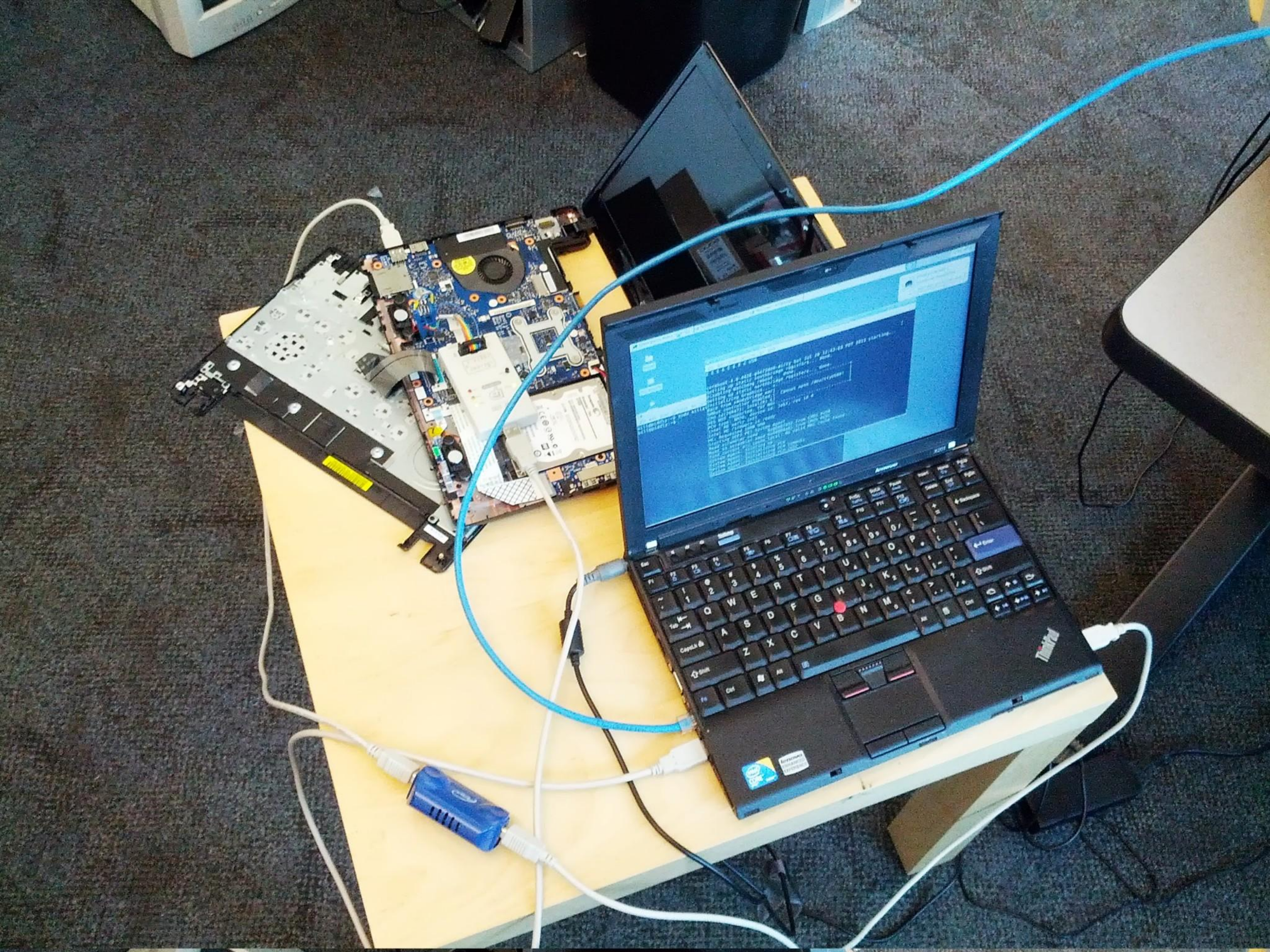
Hands On

What Next?

Let's do it!

- We have:
  - A Parrot, already disassembled
  - A laptop with a fresh Ubuntu install
  - A Dediprog
  - A USB debugger
  - Helpful instructors

- Y'all build your own BIOSes, and we'll try them out on our Parrot first.



Agenda

Intro / Background

Chrome OS Firmware

Development System

Preparing a Test System

Hands On

What Next?

Duh. What did you think I'd say?

- Keep hacking
- Submit patches
- Get involved
    - http://www.coreboot.org
    - http://www.chromium.org

Backup material

vboot_reference tools

- There are several utilities for poking at the BIOS that are part of the verified boot sources. Build them like so:

```
sudo apt-get install libssl-dev uuid-dev liblzma-dev libyaml-dev libtspi-dev

git clone https://git.chromium.org/git/chromiumos/platform/vboot_reference
cd vboot_reference
make
sudo make install
```