Google

U-Boot with Chrome OS and firmware packaging

Simon Glass
sjg@chromium.org
Open Source Firmware Conference 2018

Agenda

- Intro
- U-Boot verified-boot implementations
- Implementing Chrome OS verified boot in U-Boot
- Little demo
- The firmware packaging problem
- Binman

About me

- Embedded software most of my career
- Mostly ARM, Linux, electronics (ARM, Bluewater Systems)
- Got into U-Boot in 2011
  - 'Snow' firmware lead, then upstreamed about 1200 patches
- Have been dabbling on and off since then
  - Sandbox, driver model, buildman, binman, dtoc, rockchip, tegra
  - About 4200 commits, mostly while working in Payments
  - Briefly x86 maintainer, one-time Rockchip, now just DM and DT
- Interests
  - ARM laptops
  - Run-time configuration in firmware
  - Colorado beers (including Coors Light)

U-Boot - Universal Boot Loader

- Widely used boot loader for embedded systems
- About 1200 boards, wide architecture support
- Small, fast, simple, portable, configurable
- Large, active user / developer community
- Vast array of features to enable
  - Strong driver model
  - Run-time configuration (device tree, of-platdata)
  - Filesystems, networking, scripting, EFI
  - Small code size
  - Last device loading, fast boot
  - Easy to hack
  - Test framework and wide array of native tests

U-Boot supports...

- U-Boot verified boot
  - Uses FIT
  - Sign 'configurations' consisting of FPGA/kernel/ramdisk etc.
  - Supports multiple signatures
- Android Verified Boot
  - A/B selection, rollback protection, chained partitions
  - Locked / unlocked state
  - Used with Android things (e.g. Raspberry Pi)
- Not Chrome OS verified boot
  - Code from 2013 culled and reused, but U-Boot's support has atrophied
  - Migrated into coreboot etc.
  - What would it take to get it running again in 2018?

Why support Chrome OS verified boot in U-Boot?

- Chrome OS verified boot maps onto embedded systems well
  - Small resource requirements
  - User-friendly firmware screens and recovery
  - Auto-update and rollback support
  - Good security record
- U-Boot is the most widely used boot loader in the embedded world
  - E.g. most ARM SoCs support U-Boot
  - ARM supports U-Boot and UEFI, 'EBBR' under active development
- Chrome OS build system is a big commitment for users
  - 'Depthcharge' bootloader 130k LOC
  - Community? Features? Run-time config? Coding style?

Goals with this work

- U-Boot was pretty basic 6 years ago
  - Huge user base, but needed a lot of work
- It has changed a lot since then
- How would we do Chrome OS verified boot in 2018?
- Could it be done so that it is easy to enable it on a random board?

- Warning: I'm not there yet, this is WIP

Chrome OS verified boot (2018 redux)

- Map the code onto current U-Boot features
- Using sandbox
- Using driver model
- Using device tree
- Using other features
- Using binman

U-Boot sandbox

- Allows U-Boot to run on Linux
- Supports most subsystems
  - UART, SPI, USB, PCI, video, block devices, host file access…
- Highly productive development environment for new features
  - Faster build/test time, conductive to unit testing and debugging
- Supports TPL, SPL
- Drivers can save state across runs
  - E.g. TPM rollback indices
  - Also supports persistent DRAM across runs
- Use sandbox to bring up verified boot!

Driver model

- 'Uclass' for each subsystem
  - E.g. EC uses driver model for the transport (SPI, LPC)
  - E.g. Power and reset uses the driver-model sysreset uclass
- Automatic private memory allocation
- Strong APIs and tests
- Uses device-tree (Linux bindings) for configuration
- Small execution and memory overhead
- Supports U-Boot's lazy init and command line

- See Fosdem "U-Boot bootloader port done right – 2018 edition" - Marek Vasut
  - Or search for ""U-Boot driver model""

Device tree

- Describe the hardware configuration
  - Use for software configuration too, since there is no user space in U-Boot
- Avoid writing configuration in code
- Well known for those working on ARM in Linux
  - Others have perhaps never heard of it?

- U-Boot uses it for all run-time configuration
- Driver model has built-in support for DT
- Flat- and live-tree implementations

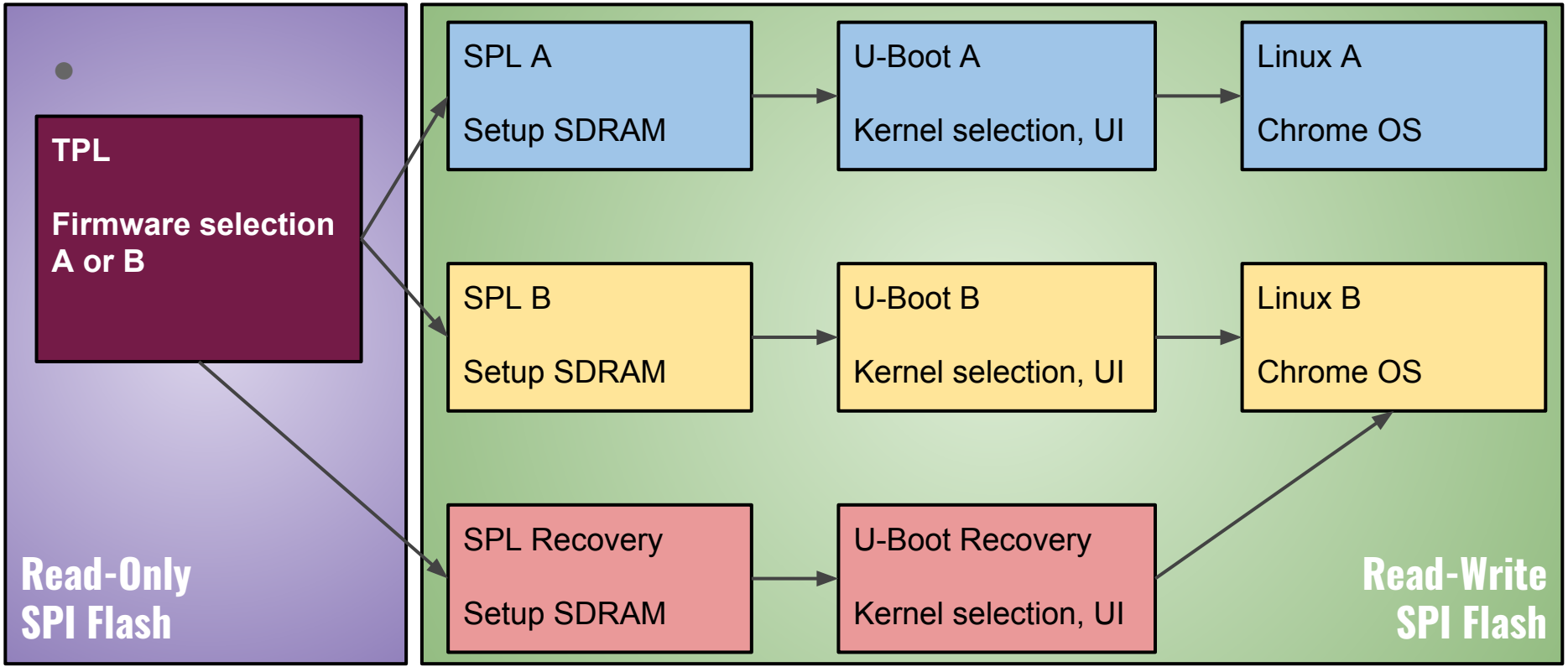
U-Boot device tree features

- Reduced size for SPL/TPL
  - fdtgrep automatically prunes out unwanted stuff
  - ~3KB overhead on Thumb 2 (+4KB for driver model)
- CONFIG_OF_PLATDATA
  - Converts device tree to C
  - Very small per-driver overhead vs. static C tables
- Live tree is optional
- Devices are bound at start-up using the DT but only probed when used
- Can also use static data to declare devices, particularly for PCI and USB

Mapping code to new U-Boot features

- Kconfig
  - Defining CONFIG_CHROMOS automatically enables most required features
- Use logging
  - Add a 'vboot' logging class and use that for most logging
  - vboot_log(LOGL_WARNING, "This is a %s warning", "big");
  - return log_msg_ret("Failed to engage hyperdrive", -ENOENT);
- TPL, SPL, U-Boot
  - Use all three stages for vboot
- 'Legacy' boot
  - Well, U-Boot already boots just about anything
  - It can even run EFI grub
- Bootstage for timestamps and durations

Boot stages


TPL
Firmware selection
A or B
Read-Only
SPI Flash
SPL A
Setup SDRAM
U-Boot A
Kernel selection, UI
Linux A
Chrome OS
SPL B
Setup SDRAM
U-Boot B
Kernel selection, UI
Linux B
Chrome OS
SPL Recovery
Setup SDRAM
U-Boot Recovery
Kernel selection, UI
Read-Write
SPI Flash

What we don't need to implement

- FIT, zboot
- ATF, OP-TEE
- EFI loader / payload
- Network boot
- Filesystems
- Scripting and environment

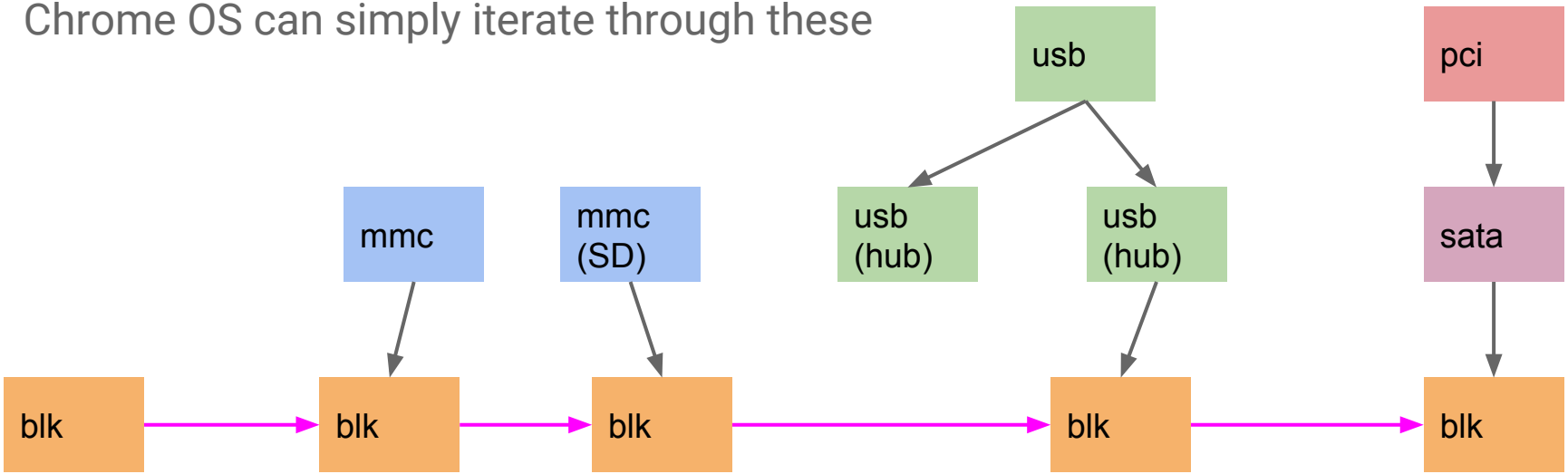
- U-Boot already has all of these
  - Many of these features have security implications
  - But are great for development and factory

A bit of a look at how driver model is used

- Some configuration and code examples
- All code comments are omitted
- Indentation is not retained
- Key
  - Device tree in grey
  - Code in purple

UCLASS_BLK

- Child device of things like USB, SCSI, MMC
- Chrome OS can simply iterate through these



UCLASS_SYSRESET

- Provides reset and power control
- Enough to implement Chrome OS requirements

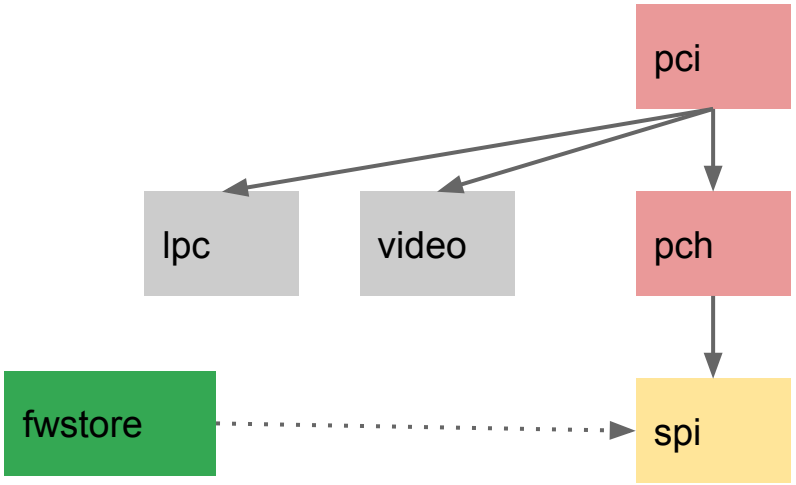
```
reset {
        compatible = "x86,reset";
        u-boot,dm-pre-reloc;
};
```

```
static int x86_sysreset_request(struct udevice *dev, enum sysreset_t type)
{
        int value;

        switch (type) {
        case SYSRESET_WARM:
                value = SYS_RST | RST_CPU;
                break;
        case SYSRESET_COLD:
        case SYSRESET_POWER:
                value = SYS_RST | RST_CPU | FULL_RST;
                break;
        default:
                return -ENOSYS;
        }

        outb(value, IO_PORT_RESET);

        return -EINPROGRESS;
}

static const struct udevice_id x86_sysreset_ids[] = {
        { .compatible = "x86,reset" },
        { }
};

static struct sysreset_ops x86_sysreset_ops = {
        .request = x86_sysreset_request,
        .get_last = x86_sysreset_get_last,
};

U_BOOT_DRIVER(x86_sysreset) = {
        .name = "x86-sysreset",
        .id = UCLASS_SYSRESET,
        .of_match = x86_sysreset_ids,
        .ops = &x86_sysreset_ops,
        .flags = DM_FLAG_PRE_RELOC,
};
```

UCLASS_FWSTORE (Chrome OS-specific)

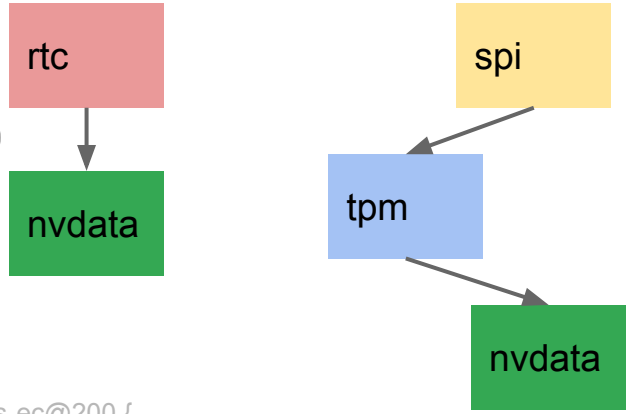
- Like BLK, fIrmware storage is backed by a parent device (e.g. SPI flash, MMC)

```
chromeos {
        compatible = "simple-bus";
                fwstore-spi {
                        compatible = "cros,fwstore-spi";
                        firmware-storage = <&fwstore_spi>;
                };
...
pch@1f,0 {
        reg = <0x0000f800 0 0 0 0>;
        compatible = "intel,broadwell-pch";
        spi: spi {
                #address-cells = <1>;
                #size-cells = <0>;
                compatible = "intel,ich9-spi";
                fwstore_spi: spi-flash@0 {
                        reg = <0>;
                        compatible = "winbond,w25q64", "spi-flash";
                        memory-map = <0xff800000 0x00800000>;
                };
        };
};
```




```
struct cros_fwstore_ops {
        int (*read)(struct udevice *dev, ulong offset, ulong count, void *buf);
        int (*write)(struct udevice *dev, ulong offset, ulong count, void *buf);
        int (*sw_wp_enabled)(struct udevice *dev);
};
```

UCLASS_NVDATA (Chrome OS-specific)



- Access to non-volatile data
- Secure (provided by TPM driver) and non-secure (provided by Cros EC / CMOS)

```
static int cros_ec_nvdata_read(struct udevice *dev, uint flags, uint8_t *data, int size)
{
        struct udevice *cros_ec = dev_get_parent(dev);

        if (flags != CROS_NV_DATA) {
                log(UCLASS_CROS_NVDATA, LOGL_ERR,
                    "Only CROS_NV_DATA supported (not %x)\n", flags);
                return -ENOSYS;
        }

        return cros_ec_read_nvdata(cros_ec, data, size);
}
```

```
int cros_nvdata_read_walk(uint index, uint8_t *data, int size);
int cros_nvdata_write_walk(uint index, const uint8_t *data, int size);
int cros_nvdata_setup_walk(uint index, uint attr, const uint8_t *data, int size);
```

```
lpc {
        cros-ec@200 {
                compatible = "google,cros-ec-lpc";
                reg = <0x204 1 0x200 1 0x880 0x80>;
                nvdata {
                        compatible = "google,cros-ec-nvdata";
                };
        };
};
rtc: rtc {
        compatible = "motorola,mc146818";
        u-boot,dm-pre-reloc;
        reg = <0x70 2>;
};
&rtc {
        #address-cells = <1>;
        #size-cells = <0>;
        nvdata {
                u-boot,dm-pre-reloc;
                compatible = "google,cmos-nvdata";
                reg = <0x26>;
        };
```

VBOOT_FLAG

- Provides values for flags
  - Can come from GPIO, static value, keypress, etc.
  - E.g. on sandbox hold R on start-up to get to recovery mode
  - Lid open, power button

```
int vboot_flag_read_walk_prev(enum vboot_flag_t flag, int *prevp)
{
...
        for (uclass_first_device(UCLASS_CROS_VBOOT_FLAG, &dev);
             dev;
             uclass_next_device(&dev)) {
                struct vboot_flag_uc_priv *uc_priv = dev_get_uclass_priv(dev);

                if (uc_priv->flag == flag)
                        break;
        }

        if (vboot_flag_read_walk(VBOOT_FLAG_DEVELOPER) == 1)
                ctx->flags |= VB2_CONTEXT_FORCE_DEVELOPER_MODE;
```

```
chromeos {
        compatible = "simple-bus";
        write-protect {
                compatible = "google,gpio-flag";
                gpio = <&gpio_a 1>;
        };
        developer {
                compatible = "google,const-flag";
                value = <1>;
        };
        lid-open {
                compatible = "google,gpio-flag";
                gpio = <&gpio_a 2>;
                sandbox-value = <1>;
        };
        power-off {
                compatible = "google,key-flag";
                /* Use KEY_PAUSE to similate power button */
                key = <116>;
        };
        oprom-loaded {
                compatible = "google,const-flag";
                value = <0>;
        };
        ec-in-rw {
                compatible = "google,gpio-flag";
                gpio = <&gpio_a 3>;
        };
};
```

Chrome OS 'global' configuration options (1)

- There are two main options for configuring how verified boot works on a particular machine
- Using Kconfig
  - Saves code size
  - But introduces new code paths and can complicate testing (need multiple builds)

1. Use CONFIG options in Kconfig

Kconfig

```
config CROS_EC
        bool "Enable Chrome OS EC"
        help
          Enable access to the Chrome OS EC. This is a separate
          microcontroller typically available on a SPI bus on Chromebooks. It
          provides access to the keyboard, some internal storage and may
          control access to the battery and main PMIC depending on the
          device. You can use the 'crosec' command to access it.
```

Code:

```
if (CONFIG_IS_ENABLED(CROS_EC)) {
        ret = uclass_get_device(UCLASS_CROS_EC, 0, &vboot->cros_ec);
        if (ret)
                return log_msg_ret("Cannot locate Chromium OS EC", ret);
        }
```

Chrome OS 'global' configuration options (2)

- Use properties in 'chromeos-config' node

```
chromeos-config {
        ec-slow-update;       /* EC firmware in sync with BIOS */
};
```

```
if (cros_ofnode_config_has_prop("ec-slow-update"))
        iparams->flags |= VB_INIT_FLAG_EC_SLOW_UPDATE;
```

Current status

- Sandbox
  - Boots to kernel, supports recovery, etc.
- Bare metal
  - Fiddling with Samus, chosen because it has existing U-Boot support
  - Currently boots to recovery
- Coreboot
  - Existing target, but now quite old
  - Attractive since it could allow running on most x86 devices
  - U-Boot now has a 'generic' coreboot target
  - Need to bring the power and drivers forward
- Raspberry Pi
  - Would like to securely boot my MAME console

Demo

- Disclaimer
  - This work was done in my spare time
  - Much of it is not in U-Boot mainline (targetting 2018.09, 2018.11 and 2019.01)
  - It is not ready for production use
  - Think of this as a demonstration and prototype
  - See 'halting problem'

Plan

- Upstream Binman additions for 2018.11
- Upstream additional features for 2019.01
  - Bloblist
  - Cros EC updates
  - Tidy up graphics
- All Chrome OS code in a cros/ subdirectory
  - 9000 LOC at present in 76 files
- Set up a tree to hold this, separate from U-Boot?
  - Select a particular vboot commit to build against?
- Ideas welcome

Binman Firmware Packaging

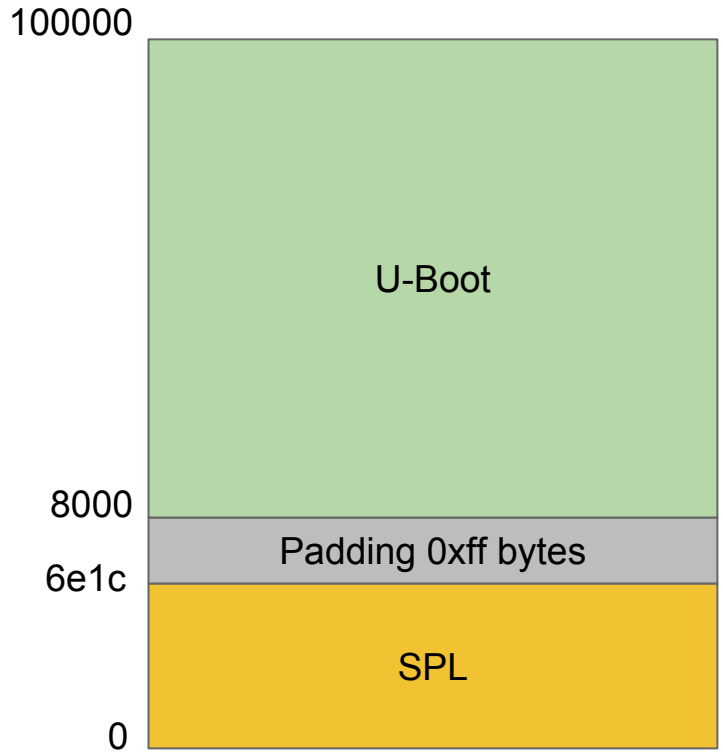
- Packaging used to be easy!
  - Stick SPL and U-Boot proper together
  - Maybe add a bit of padding for BSS
- But it is getting more complex
  - Additional binaries to include (e.g. FPGA, ATF)
  - SoC-specific signing
  - Data required by firmware (e.g. bitmap images, fonts)
- At present the solution is often ad-hoc
- Binman aims to solve this problem well

What does binman do?

- Pack entries into an image
- Align entries to boundaries
- Group entries into 'sections' (hierarchical images)
- Automatic packing /sorting / compression
- Allow content to come from a filesystem or external tool
- Allow entries to depend on other entries
- Provides run-time access to entry locations
    - Device tree, symbol injection
- Highly extensible via discoverable Python models

Example image

```
binman {
        size = <0x100000>;
        pad-byte = <0xff>;
        u-boot-spl {
        };

        u-boot {
                offset = <0x8000>;
        };
};
```



Ugly Image - used for all x86 boards (1)

```
binman {
        filename = "u-boot.rom";
        end-at-4gb;         offsets are x86 addresses
        sort-by-offset;     reorder entries by offset
        pad-byte = <0xff>;
        size = <CONFIG_ROM_SIZE>;
#ifdef CONFIG_HAVE_INTEL_ME
        intel-descriptor {
                filename = CONFIG_FLASH_DESCRIPTOR_FILE;
        };
        intel-me {      position provided by descriptor
                filename = CONFIG_INTEL_ME_FILE;
        };
#endif
        u-boot-with-ucode-ptr {
                offset = <CONFIG_SYS_TEXT_BASE>;
        };
        u-boot-dtb-with-ucode {
        };
        u-boot-ucode {
                align = <16>;
        };
#ifdef CONFIG_HAVE_MRC
        intel-mrc {
                offset = <CONFIG_X86_MRC_ADDR>;
        };
#endif
#ifdef CONFIG_HAVE_FSP
        intel-fsp {
                filename = CONFIG_FSP_FILE;
                offset = <CONFIG_FSP_ADDR>;
        };
#endif
```

Ugly Image - used for all x86 boards (2)

```
#ifdef CONFIG_HAVE_CMC
            intel-cmc {
                  filename = CONFIG_CMC_FILE;
                  offset = <CONFIG_CMC_ADDR>;
            };
#endif
#ifdef CONFIG_HAVE_VGA_BIOS
            intel-vga {
                  filename = CONFIG_VGA_BIOS_FILE;
                  offset = <CONFIG_VGA_BIOS_ADDR>;
            };
#endif
#ifdef CONFIG_HAVE_VBT
            intel-vbt {
                  filename = CONFIG_VBT_FILE;
                  offset = <CONFIG_VBT_ADDR>;
            };
#endif
```

```
#ifdef CONFIG_HAVE_REFCODE
            intel-refcode {
                  offset = <CONFIG_X86_REFCODE_ADDR>;
            };
#endif
            x86-start16 {
                  offset = <CONFIG_SYS_X86_START16>;
            };
      };
```

Litmus test - Can Binman do Chrome OS?

- Mostly
- See previous demo

- Does Chrome OS have the the most pieces in its firmware image?
  - In case you were wondering how the two topics in this talk are related
  - I am collecting votes

Binman testing

- Full test of tests for all features
- This helps to define the behaviour in corner cases (combining entry options)
- Test coverage is 100% for core and entry code
- Tests run in about 3 seconds

Why use Binman?

- Open-source firmware packer
- Maintained within an existing project (U-Boot)
- Wide range of features, but easy to extend
- Data-driven firmware packing
- Docs: https://github.com/u-boot/u-boot/tree/master/tools/binman
- Entry types: https://github.com/u-boot/u-boot/blob/master/tools/binman/README.entries
-

Thank you

- Simon Glass
  - To: u-boot@lists.denx.de
  - Cc: sjg@chromium.org
- Please give your dad / grandmother a Chromebook