2014 Chrome OS Firmware Summit

Chrome EC

- Chrome EC is open source
  - [chromiumos/platform/ec.git]
- Chrome EC is designed for security
  - RO and RW regions
  - RW update is signed and handled by host firmware
  - EC Software Sync is part of Verified Boot
- Support for different ARM SOCs
  - Texas Instruments Stellaris Cortex-M4
  - ST Micro STM32 Cortex-M3
  - More in progress...

Source Tree Layout

board/$BOARD/	Board specific code and configuration details. This includes the GPIO map, battery parameters, and set of tasks to run for the device.
chip/$CHIP/	IC specific code for interfacing with registers and hardware blocks (adc, jtag, pwm, uart etc…)
common/	A mix of upper-level code that is shared across boards. This includes the charge state machine, fan control, and the keyboard driver (amongst other things).
core/	Lower level code for task and memory management.
driver/	Low-level drivers for ALS, charge controllers, I2C/onewire LED controllers, and I2C temperature sensors.
include/	Header files for core and common code.
power/	Power sequencing logic for specific host CPU families (Ivy Bridge, Haswell, Tegra, etc.)

Source Tree Layout (continued)

test/	Unit tests for the EC (“make runtests”). Please contribute new tests if writing new functionality.
util/	Host utilities and scripts for flashing the EC. Also includes “ectool” used to query and send commands to the EC from userspace.
build/	Build artifacts are generated here. Be sure to delete this and rebuild when switching branches.

Terminology

AP / host	Application Processor. The system CPU, which actually runs Chrome OS.
Hibernate	EC power-down state. Entered when on battery power and the AP has been off for 60 minutes.
Battery cutoff / ship mode	Battery is logically disconnected or disabled for storage/shipping.
Task	One of several priority-sorted independent threads of execution (power states, charger, keyboard, etc.)
Watchdog	Reboots the EC if normal tasks are stuck or starved.
Silego chip	Hardware am-I-trustworthy module. It has a one-way latch that is cleared at a hard EC reset and set when the RO firmware jumps to the RW firmware.

Firmware architecture overview

- Tasks
- GPIOs & Interrupts
- Modules
- Hooks
- Console
- Host commands
    - memory-mapped
- Config

Tasks

- EC typically runs 6-10 independent tasks
  - Configured in board/*/ec.tasklist
- Each task has its own stack (256-640 bytes)
- Task switching is interrupt-driven
- Strictly ordered list of task priorities
  - When a higher-priority task wakes up, it rips control away from a lower-priority task.
- Events, mutexes, timers

- There is **no** heap (no malloc/free). Instead there is a small number of fixed-size memory buffers which may be temporarily allocated by tasks when needed.

board/*/ec.tasklist

```
/**
 * List of enabled tasks in the priority order
 *
 * The first one has the lowest priority.
 *
 * For each task, use the macro TASK_ALWAYS(n, r, d, s) for base tasks and
 * TASK_NOTEST(n, r, d, s) for tasks that can be excluded in test binaries,
 * where :
 * 'n' is the name of the task
 * 'r' is the main routine of the task
 * 'd' is an opaque parameter passed to the routine at startup
 * 's' is the stack size in bytes; must be a multiple of 8
 */
#define CONFIG_TASK_LIST \
        TASK_ALWAYS(HOOKS, hook_task, NULL, LARGER_TASK_STACK_SIZE) \
        TASK_ALWAYS(CHARGER, charger_task, NULL, TASK_STACK_SIZE) \
        TASK_NOTEST(CHIPSET, chipset_task, NULL, TASK_STACK_SIZE) \
        TASK_NOTEST(KEYPROTO, keyboard_protocol_task, NULL, TASK_STACK_SIZE) \
        TASK_ALWAYS(HOSTCMD, host_command_task, NULL, TASK_STACK_SIZE) \
        TASK_ALWAYS(CONSOLE, console_task, NULL, LARGER_TASK_STACK_SIZE) \
        TASK_ALWAYS(POWERBTN, power_button_task, NULL, TASK_STACK_SIZE) \
        TASK_NOTEST(KEYSCAN, keyboard_scan_task, NULL, TASK_STACK_SIZE)
```

GPIOs (board/*/board.c)

GPIOs are the normal inputs, outputs, and interrupts.

```
const struct gpio_info gpio_list[] = {
        /* Inputs with interrupt handlers are first for efficiency */
        {"POWER_BUTTON_L",      LM4_GPIO_A, (1<<2), GPIO_INT_BOTH_DSLEEP,
         power_button_interrupt},
        {"LID_OPEN",            LM4_GPIO_A, (1<<3), GPIO_INT_BOTH_DSLEEP,
         lid_interrupt},
        {"AC_PRESENT",          LM4_GPIO_H, (1<<3), GPIO_INT_BOTH_DSLEEP,
         extpower_interrupt},
…
       /* Other inputs */
        {"FAN_ALERT_L",         LM4_GPIO_B, (1<<0), GPIO_INPUT, NULL},
        {"USB1_OC_L",           LM4_GPIO_E, (1<<7), GPIO_INPUT, NULL},
        {"USB2_OC_L",           LM4_GPIO_E, (1<<0), GPIO_INPUT, NULL},
…
       /* Outputs; all unasserted by default except for reset signals */
        {"CPU_PROCHOT",         LM4_GPIO_B, (1<<1), GPIO_OUT_LOW, NULL},
        {"PP1350_EN",           LM4_GPIO_H, (1<<5), GPIO_OUT_LOW, NULL},
        {"PP3300_DSW_GATED_EN", LM4_GPIO_J, (1<<3), GPIO_OUT_LOW, NULL},
        {"PP3300_DX_EN",        LM4_GPIO_J, (1<<2), GPIO_OUT_LOW, NULL},
…
```

Modules

Common functionality and state machines are grouped into logical modules. These include

ADC, CHARGER, CHIPSET, DMA, GPIO, HOOK, I2C, KEYBOARD, LPC, PECI, POWER_LED, PWM_FAN, PWM_LED, THERMAL, UART,

and many others. The modules are largely self-contained once all the GPIOs are connected and configured.

Each module has its own initialization routines to prepare its state machine and enable its interrupts.

board/*/board.c

Configuring special-purpose GPIOs for some common modules.

```
/* Pins with alternate functions */
const struct gpio_alt_func gpio_alt_funcs[] = {
        {GPIO_A, 0x03, 1, MODULE_UART, GPIO_PULL_UP},    /* UART0 */
        {GPIO_A, 0x40, 3, MODULE_I2C},                   /* I2C1 SCL */
        {GPIO_A, 0x80, 3, MODULE_I2C, GPIO_OPEN_DRAIN}, /* I2C1 SDA */
        {GPIO_B, 0x04, 3, MODULE_I2C},                   /* I2C0 SCL */
        {GPIO_B, 0x08, 3, MODULE_I2C, GPIO_OPEN_DRAIN}, /* I2C0 SDA */
        {GPIO_B, 0x40, 3, MODULE_I2C},                   /* I2C5 SCL */
        {GPIO_B, 0x80, 3, MODULE_I2C, GPIO_OPEN_DRAIN}, /* I2C5 SDA */
        {GPIO_G, 0x30, 1, MODULE_UART},                  /* UART2 */
        {GPIO_J, 0x40, 1, MODULE_PECI},                  /* PECI Tx */
        {GPIO_J, 0x80, 0, MODULE_PECI, GPIO_ANALOG},     /* PECI Rx */
        {GPIO_L, 0x3f, 15, MODULE_LPC},                  /* LPC */
        {GPIO_M, 0x33, 15, MODULE_LPC},                  /* LPC */
        {GPIO_N, 0x0c, 1, MODULE_PWM_FAN},               /* FAN0PWM2 */
};
const int gpio_alt_funcs_count = ARRAY_SIZE(gpio_alt_funcs);
```

Hooks

- Hooks are callback functions
    - Associated with specific event categories
    - Invoked when that event occurs.
- May also be called after some specified delay ("deferred" hooks).
- Typically registered by one module, but invoked by a different module.
- The callback functions execute in the stack of the *calling* task (so deadlock is possible).

Hook Type	Called When
INIT	At EC initialization
PRE_FREQ_CHANGE	About to change the clock frequency
FREQ_CHANGE	Clock frequency changed
SYSJUMP	About to jump to another firmware image (RO to RW, for example)
CHIPSET_PRE_INIT	Before the AP starts up
CHIPSET_STARTUP	AP is starting up
CHIPSET_RESUME	AP is resuming from suspend (or booting)
CHIPSET_SHUTDOWN	AP is shutting down or suspending
AC_CHANGE	AC power state changed
LID_CHANGE	Lid was opened or closed (debounced)
POWER_BUTTON_CHANGE	Power button pressed or released (debounced)
CHARGE_STATE_CHANGE	Charge state machine status has changed
TICK	Every 250ms (or 500ms on some systems)
SECOND	Once per second

Console

The console task provides a command-line interface on the EC’s serial port.

The serial port is accessible through the servo board, via the debug connector.

This is the primary means of testing and debugging.

Console commands are trivial to add:

```
int command_foo(int argc, char *argv[]) { … }
DECLARE_CONSOLE_COMMAND(foo, command_foo, …);
```

Console commands

```
> help
Known commands:
  adc            fanset         hostcmd        pecirdpkg      syslock
  apreset        flasherase     hostevent      pecitemp       t6cal
  apshutdown     flashinfo      i2cscan        peciwrpkg      taskinfo
  apthrottle     flashwp        i2cxfer        port80         taskready
  battery        flashwrite     kbd            powerbtn       temps
  battfake       gettime        kblight        powerindebug   thermalget
  chan           gpioget        kblog          powerinfo      thermalset
  charger        gpioset        kbpress        powerled       timerinfo
  codeset        hangdet        ksstate        pwmduty        tmp006
  crash          hash           lidclose       reboot         typematic
  ctrlram        hcdebug        lidopen        rtc            usbchargemode
  dptftemp       help           lightbar       rw             version
  fanauto        hibdelay       mmapinfo       shmem          waitms
  fanduty        hibernate      panicinfo      sysinfo        ww
  faninfo        history        peciprobe      sysjump
HELP LIST = more info; HELP CMD = help on CMD.
>
```

Host commands

The AP communicates with the EC through host commands.

1. AP sends a command + data to the EC
2. EC processes the command
3. EC responds with result + data to the AP

The data can be of varying size (including zero), in each direction. The commands, data structures, result codes, and numerous other parameters are defined in

```
include/ec_commands.h
```

Mapped memory

Some systems have memory regions shared between the EC and AP address space.

Those systems provide a small number of read-only (to the AP) memory locations where the EC maintains various values that the AP may find interesting (battery voltage, fan speeds, sensor readings, etc.)

These are defined in `include/ec_commands.h`

For systems without shared memory, there are host commands to read those values.

include/config.h

This file defines all the build-time configuration options to select various modules, features, customization, debugging levels, etc.

When adding new features, the appropriate CONFIG_ option should be added to this file first.

Each board then enables the appropriate options in `board/*/board.h` and initializes any data structures in `board/*/board.c`

board/*/board.h

```
/* Optional features */
#define CONFIG_BACKLIGHT_REQ_GPIO GPIO_PCH_BKLTEN
#define CONFIG_BATTERY_SMART
#define CONFIG_BOARD_VERSION
#define CONFIG_CHARGER
#define CONFIG_CHARGER_BQ24738
#define CONFIG_CHARGER_DISCHARGE_ON_AC
#define CONFIG_CHIPSET_CAN_THROTTLE
#define CONFIG_CHIPSET_HASWELL
#define CONFIG_POWER_COMMON
#define CONFIG_CMD_GSV
#define CONFIG_EXTPOWER_FALCO
#define CONFIG_EXTPOWER_GPIO
#define CONFIG_FANS 1
#define CONFIG_KEYBOARD_BOARD_CONFIG
#define CONFIG_KEYBOARD_PROTOCOL_8042
#define CONFIG_LOW_POWER_IDLE
#define CONFIG_PECI_TJMAX 100
#define CONFIG_POWER_BUTTON
#define CONFIG_POWER_BUTTON_X86
#define CONFIG_SWITCH_DEDICATED_RECOVERY
#define CONFIG_TEMP_SENSOR
#define CONFIG_TEMP_SENSOR_G781
```

Primary responsibilities of the EC

- AP Power sequencing
- Battery Charging
- Thermal Management
- Keyboard
- Buttons and Switches
- Backlights, Indicator LEDs
- Various other board-specific peripherals

Power Sequencing

- Each AP family has its own
  - Power states
  - Voltage regulators
  - Control GPIOs (both input and output)
  - Transition rules
  - Timing requirements
  - Trigger events
- The EC must manage and respond to all those requirements as the AP boots, sleeps, idles, or transitions between various subtle states.
- It must also ensure that certain peripherals are brought up and down accordingly (USB, WiFi, etc.)

power/haswell.c

```
case POWER_S5S3:
         /* Enable PP5000 (5V) rail. */
          gpio_set_level(GPIO_PP5000_EN, 1);
          if (power_wait_signals(IN_PGOOD_PP5000)) {
                  chipset_force_shutdown();
                  return POWER_S5G3;
          }
         /* Wait for the always-on rails to be good */
          if (power_wait_signals(IN_PGOOD_ALWAYS_ON)) {
                  chipset_force_shutdown();
                  return POWER_S5G3;
          }
          /* Turn on power to RAM */
          gpio_set_level(GPIO_PP1350_EN, 1);
          if (power_wait_signals(IN_PGOOD_S3)) {
                 chipset_force_shutdown();
                  return POWER_S5G3;
          }

          gpio_set_level(GPIO_ENABLE_TOUCHPAD, 1);

          /* Call hooks now that rails are up */
          hook_notify(HOOK_CHIPSET_STARTUP);
          return POWER_S3;
```

Battery Charging

Most Chromebooks use Smart Battery technology

http://sbs-forum.org/specs/sbdat110.pdf

- The battery asks for specific voltage and current
- The charger circuitry provides it

The EC handles a few special cases:

- Trickle-charging a fully-discharged battery until it starts working again
- Forcing different charging curves under high- or low-power conditions
- Working around errata in the Smart Battery components
- Ensuring that temperatures are within safe operating ranges
- Manual charging for non-Smart batteries

Each board may have some parameters that are specific to an individual battery pack or charger.

board/*/battery.c

```
static const struct battery_info info = {
        .voltage_max    = 8400,
        .voltage_normal = 7400,
        .voltage_min    = 6000,

        /* Pre-charge values. */
        .precharge_current  = 256,      /* mA */

        .start_charging_min_c = 0,
        .start_charging_max_c = 45,
        .charging_min_c       = 0,
        .charging_max_c       = 45,
        .discharging_min_c    = -10,
        .discharging_max_c    = 60,
};
```

Thermal Management

Temp sensors

```
> temps
  I2C-USB C-Die       : 312 K = 39 C
  I2C-USB C-Object    : Not calibrated
  I2C-PCH D-Die       : 314 K = 41 C
  I2C-PCH D-Object    : Not calibrated
  I2C-Hinge C-Die     : 316 K = 43 C
  I2C-Hinge C-Object  : Not calibrated
  I2C-Charger D-Die   : 313 K = 40 C
  I2C-Charger D-Object: Not calibrated
  ECInternal          : 321 K = 48 C
  PECI                : 324 K = 51 C
>
```

Thermal Management

Independent thresholds can deliver host events (ACPI, PROCHOT), control fan speeds, or force power off based on any sensor readings.

```
> thermalget
sensor  warn  high  halt   fan_off fan_max   name
  0        0     0      0      0       0     I2C-USB C-Die
  1        0     0      0      0       0     I2C-USB C-Object
  2        0     0      0      0       0     I2C-PCH D-Die
  3        0     0      0      0       0     I2C-PCH D-Object
  4        0     0      0      0       0     I2C-Hinge C-Die
  5        0     0      0      0       0     I2C-Hinge C-Object
  6        0     0      0      0       0     I2C-Charger D-Die
  7        0     0      0      0       0     I2C-Charger D-Object
  8        0     0      0      0       0     ECInternal
  9      373   375    377    333     363     PECI
>
```

Thermal Management

Fans are continuously variable, with hysteresis and min/max settings. They can also be manually controlled by the AP (but we think the EC will do a better job).

```
> faninfo
Actual:    0 rpm
Target:    0 rpm
Duty:   0%
Status: 0 (not spinning)
Mode:   rpm
Auto:   yes
Enable: yes
Power:  yes
>
```

Keyboard

For x86-based systems, the EC provides a “standard” 8042 AT-style interface (common/keyboard_8042.c)

ARM-based systems use a binary format (common/keyboard_mkbp.c), which merely pushes changes in the scan matrix up to the kernel (enabled in the kernel by CONFIG_KEYBOARD_CROS_EC)

In either case, the keyboard scan matrix is defined in the board-specific configuration.

board/*/board.c

```
struct keyboard_scan_config keyscan_config = {
        .output_settle_us = 40,
        .debounce_down_us = 6 * MSEC,
        .debounce_up_us = 30 * MSEC,
        .scan_period_us = 1500,
        .min_post_scan_delay_us = 1000,
        .poll_timeout_us = SECOND,
        .actual_key_mask = {
                0x14, 0xff, 0xff, 0xff, 0xff, 0xf5, 0xff,
                0xa4, 0xff, 0xf6, 0x55, 0xfa, 0xc8
        },
};
```

Various other peripherals

- LEDs
- Backlight
- WiFi/USB power
- Lightbar, accelerometer, dedicated buttons, …

These are generally board-specific. In most cases they can be handled by HOOK callbacks.

EC Software Sync

It is important that the AP firmware (BIOS) and the EC firmware remain compatible through upgrades. At every cold boot/reset of the EC, this happens[*]

1. The EC boots its RO firmware, and powers on the AP.
2. The AP boots its RO firmware.
3. The AP verifies its RW firmware and jumps to it.
4. The EC computes a hash of its RW firmware.
5. The AP RW firmware contains a copy of the EC's RW firmware. The AP compares its hash with the EC's hash.
6. If they differ, the AP gives the EC the correct RW firmware, which the EC writes to its flash.
7. The EC jumps to its RW firmware.

[*] Normal boot only. In recovery mode, they both stay in RO. There also are a few other tricks to ensure the EC isn't lying about its hash.

Software Sync

If you’re developing new EC code, it’s **really** annoying when the AP silently replaces your changes at every boot.

To disable Software Sync, we can set a flag in the GBB that instructs the AP to skip that step:

```
    /usr/share/vboot/bin/set_gbb_flags.sh 0x239
```

```
0x00000001 GBB_FLAG_DEV_SCREEN_SHORT_DELAY
0x00000008 GBB_FLAG_FORCE_DEV_SWITCH_ON
0x00000010 GBB_FLAG_FORCE_DEV_BOOT_USB
0x00000020 GBB_FLAG_DISABLE_FW_ROLLBACK_CHECK
0x00000200 GBB_FLAG_DISABLE_EC_SOFTWARE_SYNC
```

Supporting a new AP board

- Start with the most similar system (reference design)
- Build the reference firmware yourself to test the process. Bonus points for running it on hardware.
- Copy the reference board sources into a new board/ directory
- Disable or stub out as much as possible
  - Edit `board/*/board.h`, `board/*/board.c`
  - Verify tasks in `board/*/ec.tasklist`
  - File bugs and add TODO() comments for missing features
- Start by updating the GPIO assignments
- Use the EC console to test GPIOs (gpioget, gpioset)
- Configure additional modules one at a time
- Compile, test, debug, repeat

Development and debugging tools

servo	Hardware debug board from Google. Connects to debug port on Chromebook, USB on Linux desktop.
servod	Daemon to arbitrate access to servo. Exposes EC and AP serial ports, JTAG, numerous controls and switches.
openocd	Programs the EC flash through servo's JTAG ports.
flashrom	Programs the AP flash through servo, EC and AP flash from the Chromebook itself.

Questions?