3. U-Boot Drivers

4/11/2013

This section lists the functional requirements for Chrome OS drivers and describes how to implement the drivers using U-Boot APIs and configuration files. It provides links to useful code samples from the U-Boot source tree.

Contents

Board Configuration

Each board has a file that contains config options for each board component. For example, the Samsung SMDK5250 (Exynos5250) board is specified in the file `include/configs/smdk5250.h`. This file controls which components are enabled and specifies certain parameters for each board component.

Driver Configuration

To add a driver for a particular class of peripheral, you need to do the following:

- Implement the APIs specified in the driver class header file.
- Add config option(s) for the peripheral to the board configuration file in `include/configs`.
- Add a node for the driver to the device tree file (*.dts*), specifying its properties and values as well as any additional devices that are connected to it.

The following sections provide details for each class of driver, including performance requirements and implementation tips. For Chrome OS implementations, also see the Main Processor Firmware Specification.

Audio Codec and Inter-Integrated Circuit Sound (I2S)

The audio codec is commonly connected to I2S to provide the audio data and to I2C to set up the codec (for example, to control volume, output speed, headphone and speaker output).

Implementation Notes

The `sound.c` file defines the `sound_init()` function, which sets up the audio codec and I2S support. Currently, this file supports the Samsung WM8994 audio codec and the Samsung I2S driver. This system would need to be expanded for other architectures to add support for new codec and I2S drivers. The `sound_play()` file, also defined in `sound.c`, plays a sound at a particular frequency for a specified period.

The `samsung-i2s.c` file defines the `i2s_tx_init()` function, which sets up the I2S driver for sending audio data to the codec. It also defines `i2s_transfer_tx_data()`, which transfers the data to the codec.

The `wm8994.c` file defines the `wm8994_init()` function, which initializes the codec hardware.

Command Line Interface

The console commands `sound_init` and `sound_play` can be used to control the audio codec.

Header File	`include/sound.h`
Implementation File	`drivers/sound/sound.c`, `drivers/sound/wm8994.c`, `drivers/sound/samsung-i2s.c`
Makefile	`drivers/sound/Makefile`
Example	`drivers/sound/wm8994.c`

back to top

Clock

The AP has a number clocks that drive devices such as the eMMC, SPI flash, and display. Although the clock structure can be very complex, with a tree of 50 or more interdependent clocks, U-Boot has a simple clock implementation. With U-Boot, you need to turn on a clock, set its frequency, and then just let it run.

Implementation Notes

The preferred technique is to create `clock.c`, which implements the clock functionality. Important clock functions to implement include the following:

- `clock_set_rate()` - sets the rate for a particular clock
- `clock_start_periph_pll()` - starts a peripheral clock at a particular rate
- `clock_set_enable()` - enable and disable a clock
- `reset_periph()` - reset a peripheral
- `clock_get_rate()` - queries the clock rate

Header File	`arch/arm/include/asm/arch-`*xxxx*`/clock.h`

Implementation File	*typically in AP directory, e.g.,* arch/arm/cpu/armv7/arch-*xxxx*/clock.c
Makefile	*typically, also in AP directory*
Example	arch/arm/cpu/tegra20-common/clock.c

back to top

Ethernet

An Ethernet connection is often used during development to download a kernel from the network. This connection is also used in the factory to download the kernel and ramdisk.

U-Boot supports the following network protocols:

- **TFTP** - for downloading a kernel and also for uploading trace data
- **NFS** - also used for downloading a kernel
- **BOOTP/DHCP** - for obtaining an IP address
- **ping** - for checking network connectivity

Many x86 devices have a built-in Ethernet port. Another way to provide Ethernet to a system is to connect a USB-to-Ethernet adapter to the USB port. If the device has a built-in port, Ethernet is detected when the board starts up and is available for use. To enable the USB-to-Ethernet connection, use the U-Boot command `usb start`.

Another useful feature for development is that when you want to use an NFS root from the network, U-Boot can provide suitable boot arguments to the kernel on the Linux command line.

Implementation Notes

The structure `eth_device` in the file `net.h` describes the Ethernet driver. The board file calls the `probe()` function, which probes for Ethernet hardware, sets up the `eth_device` structure, and then calls `eth_register()`.

You need to implement the following functions for the Ethernet driver:

- `init()` - brings up the Ethernet device
- `halt()` - shuts down the Ethernet device
- `send()` - sends packets over the network
- `recv()` - receives packets over the network
- `write_hwaddr()` - writes the MAC address to the hardware from the `ethaddr` environment variable.

For USB Ethernet, the structure `ueth_data` in the file `usb_ether.h` describes the USB Ethernet driver. The `usb_ether.h` file also defines a set of three functions that must be implemented for each supported adapter. For example, here are the functions for the Asix adapter:

```
#ifdef CONFIG_USB_ETHER_ASIX
_ void asix_eth_before_probe(void);
_ int asix_eth_probe(struct usb_device *dev, unsigned int ifnum,
_                     struct ueth_data *ss);
_ int asix_eth_get_info(struct usb_device *dev, struct ueth_data *ss,
_                      struct eth_device *eth);
  #endif
```

The `xxx_eth_probe()` function probes for the device and must return nonzero if it finds a device. The `xxx_eth_get_info()` function obtains information about the device and fills in the `ueth_data` structure.

Header File	`include/net.h` `include/usb_ether.h`
Implementation File	`drivers/net/lan91c96.c` *(private to driver)*
Makefile	`drivers/net/Makefile`
Example	`drivers/usb/eth/asix.c` *(specific adapter)* `drivers/usb/eth/usb_ether.c` *(generic interface)*

back to top

GPIO

Modern APs can contain hundreds of GPIOs (General Purpose Input/Output pins). GPIOs can be used to *control* a given line or to *sense* its current state. A given GPIO can serve multiple purposes. Also, peripheral pins can often also be used as GPIOs. For example, an AP MMC interface requires 11 pins that can be used as GPIOs if the MMC function is not needed.

A GPIO can be either an *input* or an *output*. If an input, its value can be read as 0 or 1. If an output, then its value can be set to 0 or 1.

Generic GPIO Interface

U-Boot provides a generic GPIO interface in `include/asm-generic/gpio.h`. This interface provides the following functions:

Function	
`int gpio_request(unsigned gpio, const char *label);`	Request
`int gpio_free(unsigned gpio);`	Frees the
`int gpio_direction_input(unsigned gpio);`	Makes th
`int gpio_direction_output(unsigned gpio, int value);`	Makes th value
`int gpio_get_value(unsigned gpio);`	Gets the
`int gpio_set_value(unsigned gpio, int value);`	Sets the

In U-Boot, GPIOs are numbered from 0, with enums specified in the AP header file `gpio.h`. For example:

```
enum gpio_pin {

_          GPIO_PA0 = 0,    /* pin 0 */
_          GPIO_PA1,
_          GPIO_PA2,
_          GPIO_PA3,
_          GPIO_PA4,
_          GPIO_PA5,
_          GPIO_PA6,
_          GPIO_PA7,
_          GPIO_PB0,        /* pin 8 */
_
```

```
_            GPIO_PB1,
_            GPIO_PB2,
.
.
.
};
```

The generic GPIO functions specify the GPIO pin by its number, as described in `gpio.h`.

Additional Functions

The generic GPIO interface does not cover all features of a typical AP. For example, custom AP functions are required to specify the following:

- **Drive strength** is defined in a chip-specific function.
- **Pinmux** selects which function a pin has (for example, MMC, LCD, GPIO) and what is controlling the pin. The pinmux module typically handles this function.
- **Pullup and pulldown** functionality is defined in a chip-specific function so that there are no floating lines.

Command Line Interface

The GPIO driver has a corresponding `gpio` command line interface that can be used to set and get GPIO values. See `common/cmd_gpio.c` for a list of commands (`input`, `set`, `clear`, `toggle`). The gpio `status` command, which you must implement, displays the status of all GPIOs in the system. This useful command should be able to accept both numbers and names for GPIO pins, as defined in `gpio.h`.

Header File	`arch/arm/include/asm/arch-tegra20/gpio.h`
Implementation File	*typically in AP directory, e.g.,* `drivers/gpio/gpio-`*`xxx`*`.c`
Makefile	`drivers/gpio/Makefile`
Example	`drivers/gpio/tegra_gpio.c (?)`

back to top

Inter-Integrated Circuit Communication (I2C)

The inter-integrated circuit communication (I2C) driver is the most-used driver in U-Boot for Chrome OS. For example, the I2C driver is used to send and receive data from the following devices:

- PMIC (Power Module)
- Trusted Platform Module (TPM)
- Embedded Controller
- Battery gas gauge
- Battery charger
- LCD/EDID
- Audio codec

Because I2C drivers form a critical part of U-Boot, they should be tested to ensure that a given I2C bus works correctly with multiple slaves and at all supported speeds. Be sure the driver correctly handles NAK messages from slaves and provides robust error handling.

Setup

Ordering of multiple I2C buses (there are usually half a dozen or more) is specified in the device tree file aliases section (for example, see `board/samsung/dts/exynos5250-smdk5250.dts`). Bus numbering is zero-based.

The following function is called by the board file to set up I2C ports:

```
void board_i2c_init(const void *blob);
```

In this function, `blob` is the device tree.

Given a node in the device tree, the following function returns the bus number of that node:

```
int i2c_get_bus_num_fdt(int node);
```

An I2C bus typically runs at either 100 kHz or 400 kHz. Ideally the driver should support exactly these speeds. In no case should the driver exceed the specified speed. The bus speed is specified in the device tree for a given bus. Although the U-Boot driver header files include functions for setting I2C bus speeds, these functions should not be used directly. Instead, set one speed for each I2C bus in the device tree, choosing the speed that matches the slowest device on a given bus.

Communication

Several of the I2C functions use the concept of a "current bus":

- `i2c_set_bus_num()` sets the current bus number
- `i2c_get_bus_num()` returns the current bus number

Typically, you follow this pattern:

1. Call `i2c_get_bus_num()` to obtain the current bus.
2. Store this bus number so that you can restore this state when you are finished with your transaction.
3. Call `i2c_set_bus_num()` to set the bus for your current transaction.
4. Perform processing on this bus (sending and receiving data).
5. Finally, call `i2c_set_bus_num()` to reset the bus to its original number.

The functions `i2c_read()` and `i2c_write()` are used to receive and send data from the I2C bus. Because multiple devices share the same bus, the functions require information about both the chip address and the memory address within the chip. For example, the syntax for i2c_read is as follows:

```
int i2c_read(uchar chip, uint addr, int alen, uchar *buffer, int l
```

where

`chip`
: is the I2C chip address, in the range 0 to 127

`addr`
: is the memory address within the chip (the register)

`alen`
: is the length of the address (1 for 7-bit addressing, 2 for 10-bit addressing)

`buffer`
: is where to read the data

`len`
: is how many bytes to read

Command Line Interface

The I2C bus has a corresponding `i2c` command line interface that can be used to read and write data.

Header File	[include/i2c.h]
Implementation File	[drivers/i2c/*driverName*.c]
Makefile	[drivers/i2c/Makefile]
Example	[drivers/tegra_i2c.c]

back to top

Keyboard

In Chrome OS, the keyboard is managed by the embedded controller (EC), which reports key presses to the AP using messages. Implementing support in U-Boot for the keyboard driver is different for x86 and ARM systems. On x86 systems, 8042 keyboard emulation is used over ACPI to report key presses. On ARM systems, the Chrome OS EC protocol is used to report key scans.

Implementation Notes

On x86 Systems

On x86 systems, the 8042 protocol is handled by a keyboard driver that communicates with the AP using an x86 I/O port. On these systems, you are responsible for implementing the keyboard driver that reports key presses to the AP.

Header File	`include/`
Implementation File	[drivers/input/keyboard.c] [drivers/input/i8042.c]
Makefile	`drivers/`
Example	`drivers/`

On ARM Systems

On ARM systems, the `cros_ec` driver communicates with the EC and requests key scans. Each message contains the state of every key on the keyboard. This design is chosen to keep the EC as simple as possible. On ARM systems, the U-Boot and Linux code provides built-in support for converting key scans into key presses through the input layer (`input.c` and `key_matrix.c`).

The device tree contains a `keyboard` node that has a `linux, keymap` property that defines the keycode at each position in the keyboard matrix. You may need to edit this file to reflect your keyboard.

Setting up the keyboard driver. Three functions are used to initialize and register a new keyboard driver (see the function `tegra_kbc_check ()` in`tegra-kbc.c` for an example of waiting for input and then checking for key presses):

- `input_init()` - initializes a new keyboard driver.
- `read_keys()` - called when the input layer is ready for more keyboard input.

- `input_stdio_register()` - registers a new input device (see "Functions Used by Input Devices," below).

Functions provided by the input layer. The following functions are defined by the input layer (`input.c`) and must be implemented for your driver: *(TRUE? or do they just use these functions?)*

- `key_matrix_decode()` - converts a list of key scans into a list of key codes.
- `input_send_keycodes()` - sends key codes to the input system for processing by U-Boot.

Keyboard auto-repeat is handled by the input layer automatically.

Functions used by input devices. U-Boot supports multiple console devices for input and output. Input devices are controlled by the environment variable `stdin` which contains a list of devices that can supply input. It is common for this variable to contain both serial input and keyboard input, so you can use either type of input during development.

An input device has three main functions to implement for use by `input_stdio_register()`. Each of these functions communications with the input layer.

- `getc()` - obtains a character and then passes it to `input_getc()` in the input layer.
- `tstc()` - checks whether a character is present (but does not read it) and then passes the result to `input_tst()`.
- `start()` - starts the input device; is called by the input system when the device is selected for input.

Configuration options. The following configuration options describe how the keyboard is connected to the EC. Include the appropriate option in the board configuration file.

`CONFIG_CROS_EC`	Enable EC protocol
`CONFIG_CROS_EC_I2C`	Select the I2C bus for communication with the EC
`CONFIG_CROS_EC_SPI`	Select the SPI bus for communication with the EC
`CONFIG_CROS_EC_LPC`	Select the LPC bus for communication with the EC
`CONFIG_CROS_EC_KEYB`	Enable the keyboard driver

Header File	`include/configs/`*`boardname`*
Implementation File	`drivers/input/cros_ec_keyb.c` *(uses standard input layer of U-Boot:* `drivers/input/input.c` *and* `drivers/input/key_matrix.c`)
Makefile	`drivers/`
Example	`drivers/input/tegra-kbc.c`

back to top

LCD/Video

The display is used to present several screens to the user. If the firmware is unable to boot, the screen displays recovery boot instructions for the user. Similarly, when the system enters developer mode, a special screen warns the user before entering this unprotected mode.

(sample screen here)

Requirements

Total wait time to enable the LCD should be as close to zero as possible. Initialization of the LCD can be interspersed with other startup operations, but blocking time during the initialization process should be less than 10 ms.

Implementation Notes

Board File. Add a function to the board file, `board_early_init_f()`, to set up three LCD parameters in the `panel_info` struct.

- `vl_col`
- `vl_row`
- `vl_bpix`

U-Boot allocates memory for your display based on these parameters. (Alternatively, set up the LCD driver in the driver `.c` file and then add a function to the board file that calls the setup function in your driver `.c` file. For an example of this technique, see `drivers/video/tegra.c`.)

`lcd_ctrl_init()` function. Sets up the display hardware. You may want to set up cache flushing in this function to speed up your display. See `lcd_set_flush_dcache()`, which is provided for you in `common/lcd.c`.

Efficient Initialization. Because LCD initialization takes a long time (sometimes as much as .5 to 1 second), you may want to use a function that manages the initialization efficiently and keeps things moving. For example, see `tegra_lcd_check_next_stage()` in `tegra.c`.

`lcd_enable()` function. As its name suggests, this function is used to enable the LCD. However, it is normally a null operation in Chrome OS because the `lcd_check_next_stage()` function will enable the LCD.

U-Boot controls drawing characters and images and scrolling. The driver specifies a basic data structure that describes the screen parameters. The generic driver is defined in the `lcd.h` file:

```
    typedef struct vidinfo {
           ushort  vl_col;         /* Number of columns (i.e. 640) */
           ushort  vl_row;         /* Number of rows (i.e. 480) */
           ushort  vl_width;       /* Width of display area in millimeters *
          ushort  vl_height;      /* Height of display area in millimeters *

          /* LCD configuration register */
           u_char  vl_clkp;        /* Clock polarity */
           u_char  vl_oep;         /* Output Enable polarity */
           u_char  vl_hsp;         /* Horizontal Sync polarity */
           u_char  vl_vsp;         /* Vertical Sync polarity */
           u_char  vl_dp;          /* Data polarity */
           u_char  vl_bpix;        /* Bits per pixel, 0 = 1, 1 = 2, 2 = 4, 3
           u_char  vl_lbw;         /* LCD Bus width, 0 = 4, 1 = 8 */
           u_char  vl_splt;        /* Split display, 0 = single-scan, 1 = du
           u_char  vl_clor;        /* Color, 0 = mono, 1 = color */
           u_char  vl_tft;         /* 0 = passive, 1 = TFT */

           /* Horizontal control register. Timing from data sheet */
           ushort  vl_wbl;         /* Wait between lines */

           /* Vertical control register */
           u_char  vl_vpw;         /* Vertical sync pulse width */
           u_char  vl_lcdac;       /* LCD AC timing */
           u_char  vl_wbf;         /* Wait between frames */
   } vidinfo_t;
```

The LCD driver specifies where screen starts in memory; pixel depth, width, and height of screen. It also declares functions to enable the screen and turn it on, including the backlight.

The ARM and x86 platforms use slightly different APIs. ARM uses `lcd.h` and x86 uses `video.h`. The implementation files for both ARM and X86 are located in the `drivers/video` directory.

ARM Files

Header File	`include/lcd.h`
Implementation File	`drivers/video`
Makefile	`drivers/video/Makefile`
Example	`CONFIG_LCD` `drivers/video/tegra.c`

x86 Files

On the x86 platform, video has its own U-Boot interface, but the existing coreboot driver will initialize the video without any further modification to U-Boot.

Header File	`include/video.h`
Implementation File	`drivers/video`
Makefile	`drivers/video/Makefile`
Example	`CONFIG_VIDEO` `drivers/video/coreboot_fb.c`

back to top

NAND

U-Boot provides support for the following types of raw NAND drivers:

- MTD - drivers for communicating with a NAND device as a block device. A wide variety of NAND chips are supported.
- ECC - support for error correction and detection
- YAFFS2 - a type of file system used by NAND flash
- UBIFS - another type of file system used by NAND flash

Chrome OS currently does not use raw NAND flash. Instead, it uses EMMC or SATA drivers, which provide a high-level interface to the underlying NAND flash.

Header File	`include/nand.h`
Implementation File	`drivers/mtd/nand`
Makefile	`drivers/mtd/Makefile`

back to top

Pin Multiplexing

Each AP vendor is responsible for writing the code in `pinmux.c`, which uses the device tree to set up the pinmux for a particular driver. U-Boot uses the same pinmux bindings as the kernel. These settings are normally static (that is, the settings are selected once and remain unchanged).

Implementation Notes

For maximum speed, U-Boot should initialize only the pins it uses during its boot. All other pins are left in their default configuration and can be initialized later by the kernel. For example, the WiFi pinmux is not required at boot time and can be initialized later by the kernel.

Header File	`arch/arm/include/asm/arch-tegra20/pinmux.h`
Implementation File	*typically in AP directory, e.g.,* `arch/arm/cpu/armv7/arch-xxxxx/pinmux.c`
Makefile	*typically, also in AP directory*
Example	`arch/arm/cpu/tegra20-common/pinmux.c`

back to top

Power

U-Boot code initializes both the AP power as well as the power to the system peripherals. Power to the AP is initialized using the PMIC (Power Management IC) driver. In addition to PMIC, the drivers/power directory includes subdirectories for power-related drivers for controlling the battery and for the fuel gauge that measures the current amount of power in the battery. (Chrome OS may or may not use these additional drivers.)

Implementation Notes

The board-specific code is located in `board/`*`manufacturer/boardname/boardname.c`* (for example, `board/samsung/trats/trats.c`). In this file, you implement the following initialization function, which is called by the file `arch/arm/lib/board.c`:

```
int power_init_board(void);
```

This function initializes the AP power through the PMIC and turns on peripherals such as the display and eMMC. It also checks the battery if needed.

Header File	`include/power/pmic.h` `include/power/battery.h`

Implementation File	`drivers/power/pmic/`
	`drivers/power/battery/`
	`drivers/power/fuel_gauge`

Makefile	`drivers/power/Makefile`

Example	`CONFIG_POWER_MAX8998 (for PMIC)`
	`CONFIG_POWER_BATTERY_`
	`board/samsung/trats/trats.c`

back to top

Pulse Width Modulation (PWM)

The pulse width modulation (PWM) driver is often used to control display contrast and LCD backlight brightness.

Implementation Notes

This driver requires you to implement generic functions defined in `include/pwm.h` as well as chip-specific functions defined in the AP directory. Basic functions to implement include the following:

- `pwm_init()` - sets up the clock speed and whether or not it is inverted
- `pwm_config()` - sets up the duty and period, in nanoseconds
- `pwm_enable()` - enables the PWM driver
- `pwm_disable()` - disables the PWM driver

Header File	`include/pwm.h` *(basic interface)* `arch/arm/include/asm/arch-tegra20/pwm.h` *(AP functions)*
Implementation File	`arch/arm/cpu/armv7/tegra20/pwm.c`
Makefile	*typically, also in AP directory*
Example	`arch/arm/cpu/armv7/tegra20/pwm.c`

back to top

SDMMC and eMMC

The same driver is typically used by both the SDMMC and the eMMC. Secure Digital Multimedia Memory Card (SDMMC) refers to an external SD card. eMMC is an internal mass storage device.

We do not currently use eMMC boot blocks. Chrome OS can boot from an external SD card or from internal eMMC. It is convenient to be able to boot from eMMC for development purposes.

Chrome OS divides the disk using the EFI partitions table. The kernel is read directly from the partition without using the file system. Be sure to enable the EFI partition table in the board file using the following option:

- `#define CONFIG_EFI_PARTITION`

Requirements

SDMMC

- 4-bit.
- Speed not critical as this device is used only in developer/recovery modes.

eMMC

- 8-bit data width, ideally DDR.
- *Speed:* For reading, 40Mbytes/sec or better is required. Writing speed is less critical because the eMMC rarely performs write operations in U-Boot.

Implementation Notes

Set up the struct `mmc` and call `mmc_register(mmc)` for each MMC device (for example, once for the SDMMC and once for the eMMC). Important functions to implement include the following:

- `send_cmd()` - send a command to the MMC device
- `set_ios()` - to set the I/O speed
- `init()` - to initialize the driver

Some of the functions perform differently depending on which type of device is being initialized. For example, `mmc_getcd()` indicates whether an SC card is currently in the slot. This function always returns TRUE for an eMMC card.

Key values to set in struct `mmc` include the following:

- `voltages` - supported voltages
- `host_caps` - capabilities of your chip
- `f_min` - minimum frequency
- `f_max` - maximum frequency

Header File	include/mmc.h
Implementation File	drivers/mmc
Makefile	drivers/mmc/Makefile
Example	CONFIG_TEGRA_MMC drivers/mmc/tegra_mmc.c

back to top

SPI

The SPI driver specifies the list of known SPI buses in a structure stored in the `drivers/spi/` directory (for an example, see the `exynos_spi_slave` structure in `drivers/spi/exynos_spi.c`). The first field in this structure is another `spi_slave structure`, `slave`, which describes the bus (`slave.bus`) and chip select (`slave.cs`) for each SPI slave device.

Ordering of multiple SPI buses is specified in the device tree file (for example, `board/samsung/dts/exynos5250-smdk5250.dts`).

Each device connected to the SPI bus (for example, the EC, touchpad, and SPI flash could all be connected to this bus) contains a reference to the `spi_slave structure` in its implementation file (for example, see `drivers/mtd/spi/winbond.c` for the SPI flash chip and `drivers/misc/cros-ec.c` for the EC.

Implementation Notes

Use this function to set up a device on the SPI bus:

```
struct spi_slave *spi_setup_slave(unsigned int busnum,          // bus
                                  unsigned int cs,              // chip
                                  unsigned int max_hz,          // maxir
                                  unsigned int mode)            // mode
```

To drop the device from the bus, use this function:

```
void spi_free_slave(struct spi_slave *slave)
```

Other key functions are used to claim control over the bus (so communication can start) and to release it:

```
int spi_claim_bus(struct spi_slave *slave);

void spi_release_bus(struct spi_slave *slave);
```

When a slave claims the bus, it maintains control over the bus until you call the `spi_release_bus()` function.

Header File	include/spi.h
Implementation File	drivers/spi/exynos_spi.c
Makefile	drivers/spi/Makefile
Example	drivers/spi/exynos_spi.c

back to top

SPI Flash

The SPI flash stores the firmware and consists of a Read Only section that cannot be changed after manufacturing and a Read/Write section that can be updated in the field.

Requirements

Size: The SPI flash is typically 4 or 8 Mbytes, with half allocated to the Read Only portion and half to the Read/Write portion.

Speed: Because this component directly affects boot time, it must be fast: 5 Mbytes/second performance is required.

Implementation Notes

There are several steps to implementing the SPI flash. First, define a prototype for your SPI flash, with a name in the form `spi_flash_probe_`*`yourDeviceName`*`()`. Add this function to `spi_flash_internal.h`.

Also, modify `spi_flash.c`, adding the config option that links your prototype to the #define. For example, for the Winbond SPI flash, these lines link the `spi_flash_probe_winbond()` prototype to its #define:

```
#ifdef CONFIG_SPI_FLASH_WINBOND
        { 0, 0xef, spi_flash_probe_winbond, },
#endif
```

This code passes in the first byte of the chip's ID (here, `0xef`). If the ID code matches that of the attached SPI flash, the struct `spi_flash` is created.

You also need to define the `spi_flash` structure that describes the SPI flash:

```
struct spi_flash {
  struct spi_slave *spi;
  const char       *name;
  /* Total flash size */
  u32              size;
  /* Write (page)  size */
  u32              page_size;
  /* Erase (sector) size */
```

```
    u32                 sector_size;
    int                 (*read)(struct spi_flash *flash, u32 o
    int                 (*write)(struct spi_flash *flash, u32 
    int                 (*erase)(struct spi_flash *flash, u32 
};
```

In this structure, you set the appropriate fields and either implement the functions for reading, writing, and erasing the SPI flash or use existing functions defined in `spi_flash_internal.h`.

Header File	`include/spi_flash.h` `drivers/mtd/spi/spi_flash_internal.h`
Implementation File	`drivers/mtd/spi/`*`yourDriver.c`*
Makefile	`drivers/mtd/spi/Makefile`/
Example	`CONFIG_SPI_FLASH` and `CONFIG_SPI_FLASH_winbond` `drivers/mtd/spi/winbond.c`

back to top

Thermal Management Unit (TMU)

The thermal management unit (TMU) monitors the temperature of the AP. If the temperature reaches the upper limit, the TMU can do one of the following:

- Slow the system down until it cools to a certain point
- Power off the system before it melts

The `dtt` command can be used on the console to perform these functions.

Implementation Notes

To implement this driver, you have two tasks:

- Create a TMU driver and connect it to the `dtt` command by modifying the file `common/cmd_dtt.c`.
- Set up a thermal trip in your board file. The temperature limits should be defined in the device tree.

Header File	
Implementation File	`common/cmd_dtt.c`
Makefile	
Example	`board/samsung/smdk5250/smdk5250.c`

back to top

Timer

The U-Boot timer is measured in milliseconds and increases monotonically from the time it is started. There is no concept of time of day in U-Boot.

Implementation

The timer requires two basic functions:

- `timer_init()` - called early in U-Boot, before relocation
- `get_timer()` - can be called any time after initialization

Typical Use

The timer is commonly used in U-Boot while the system is waiting for a specific event or response. The following example shows a typical use of the timer that can be used to ensure that there are no infinite loops or hangs during boot:

```
start = get_timer(0)
while ( ...) {
    if (get_timer (start) > 100){

        debug "%s:Timeout while waiting for response\^");
        return -1;
    }
    .
    .
    .
}
```

A `base_value` is passed into the `get_timer()` function, and the function returns the elapsed time since that `base_value` (that is, it subtracts the `base_value` from the current value and returns the difference).

Other Uses

The `timer_get_us()` function returns the current monotonic time in microseconds. This function is used in Chrome OS verified boot to obtain the current time. It is also used by bootstage to track boot time (see `common/bootstage.c`). It should be as fast and as accurate as possible.

Delays

The `__udelay()` function, also implemented in `timer.c`, is called by U-Boot internally from its `udelay` (delay for a period in microseconds) and `mdelay` (delay for a period in milliseconds) functions (declared in `common.h`):

```
void __udelay (unsigned long);
```

This function introduces a delay for a given number of microseconds. The delay

Command Line Interface

The `time` command can be used to time other commands. This feature is useful for benchmarking.

Header File	include/common.h
Implementation File	`timer.c`, *in the AP directory*
Makefile	
Example	arch/arm/cpu/armv7/s5p_common/timer.c

back to top

Trusted Platform Module (TPM)

The Trusted Platform Module (TPM) is the security chip that maintains rollback counters for firmware and kernel versions and stores keys for the system. The TPM is normally connected on an LPC, SPI, or I2C bus.

Requirements

The TPM is a critical contributor to Chromium OS boot time, so it must be as fast as possible.

Implementation

All message formatting is handled in `vboot_reference`, so the functions you must implement for the TPM in U-Boot are for low-level initialization, open/close, and send/receive functionality:

```
int tis_init();
int tis_open();
int tis_close();
int tis_sendrecv
(const uint8_t *sendbuf, size_t send_size, uint8_t *recvbuf,
                size_t *recv_len);
```

Header File	include/tpm.h
Implementation File	drivers/tpm/*yourDriver.c*
Makefile	drivers/tpm/Makefile
Example	CONFIG_GENERIC_LPC_TPM drivers/tpm/generic_lpc_tpm.c

back to top

UART

The UART is used for serial communication to drive the serial console, which provides output during the boot process. This console allows the user to enter commands for testing and debugging.

Requirements

- 115K2 baud
- 3-wire port (no handshaking)

Implementation Notes

You register this driver by setting up a structure (`serial_device{}`) and then call `serial_register()`. The AP typically has built-in serial functions that this driver can use.

```
struct serial_device {
        /* enough bytes to match alignment of
following func pointer */
        char    name[16];

        int     (*start)(void);
        int     (*stop)(void);
        void    (*setbrg)(void);
        int     (*getc)(void);
        int     (*tstc)(void);
        void    (*putc)(const char c);
        void    (*puts)(const char *s);
#if CONFIG_POST & CONFIG_SYS_POST_UART
        void    (*loop)(int);
#endif
        struct serial_device    *next;
};
```

In the final shipping product, the console needs to run in silent mode. For example, this code in the driver tells the console to run in silent mode:

```
 if (fdtdec_get_config_int(gd->fdt_blob,
"silent_console", 0))
         gd->flags |= GD_FLG_SILENT;
```

Header File	include/serial.h
Implementation File	drivers/serial/
Makefile	drivers/serial/Makefile
Example	CONFIG_SYS_NS16550 drivers/serial/serial_ns16550.c

back to top

USB Host

USB host is used by Chrome OS in two ways:

- *In recovery mode:* U-Boot software must be able to detect the presence of a USB storage device and load a recovery image from the device.
- *For Ethernet connectivity:* On Chrome OS platforms without a built-in Ethernet connector, a USB-to-Ethernet adapter can be used to provide an Ethernet connection.

On x86 systems, USB is often used to connect other peripherals, such as cameras and SD card readers, but Chrome OS does not require U-Boot drivers for these USB peripherals.

U-Boot supports both the EHCI and OHCI standards for USB. Be sure to test a variety of common USB storage devices to ensure that they work with your U-Boot driver.

USB Hub

In some cases, the USB or AP is connected to a USB hub to expand the number of USB ports. The board file may need to power up this hub and configure it. Be careful to power up the hub only if USB is used by the system.

EFI Partition Table

Chrome OS uses an EFI partition table. To enable this table, add the following entry to the board file:

- `#define CONFIG_EFI_PARTITION`

Implementation Notes

USB should not be initialized in the normal boot path. Loading a kernel from USB is a slow process (it could take a second or more) as well as a potential security risk.

The two main functions to implement are the following:

```
int ehci_hcd_init(int index, struct ehci_hccr **hccr, struct ehc

int ehci_hcd_stop(int index)
```

These functions create (and destroy) the appropriate control structures to manage a new EHCI host controller. Much of this interface is standardized and implemented by U-Boot. You just point U-Boot to the address of the peripheral.

Configuration Options

The configuration options for USB are specified in the board configuration file (for example, `include/configs/seaboard.h`). There are a number of #defines for different aspects of USB, including the following:

`CONFIG_USB_HOST_ETHER`	Enables U-Boot support for USB Ethernet
`CONFIG_USB_ETHER_ASIX`	Enables the driver for the ASIX USB-to-Ethernet adapter
`CONFIG_USB_ETHER_SMSC95XX`	Enables the driver for the SMSC95XX USB-to-Ethernet adapter
`CONFIG_USB_EHCI`	Enables EHCI support in U-Boot
`CONFIG_USB_EHCI_TEGRA`	Enables EHCI for a specific chip
`CONFIG_USB_STORAGE`	Enables a USB storage device
`CONFIG_CMD_USB`	Enables the USB command

Header File	`drivers/usb/host/ehci.h` `drivers/usb/host/ohci.h`
Implementation File	`drivers/usb/host/ehci-`*controllerName.c* `drivers/usb/host/ohci-`*controllerName.c*
Makefile	
Example	`drivers/usb/host/ehci-tegra.c`

back to top

Other sections in *U-Boot Porting Guide*

1. Overview of the Porting Process
2. Concepts
3. U-Boot Drivers (this page)