Linux Leds Documentation

The kernel development community

Jun 10, 2024

CONTENTS

CHAPTER

ONE

LED HANDLING UNDER LINUX

In its simplest form, the LED class just allows control of LEDs from userspace. LEDs appear in /sys/class/leds/. The maximum brightness of the LED is defined in max_brightness file. The brightness file will set the brightness of the LED (taking a value 0-max_brightness). Most LEDs don't have hardware brightness support so will just be turned on for non-zero brightness settings.

The class also introduces the optional concept of an LED trigger. A trigger is a kernel based source of led events. Triggers can either be simple or complex. A simple trigger isn't configurable and is designed to slot into existing subsystems with minimal additional code. Examples are the disk-activity, nand-disk and sharpsl-charge triggers. With led triggers disabled, the code optimises away.

Complex triggers while available to all LEDs have LED specific parameters and work on a per LED basis. The timer trigger is an example. The timer trigger will periodically change the LED brightness between LED_OFF and the current brightness setting. The "on" and "off" time can be specified via /sys/class/leds/<device>/delay_{on,off} in milliseconds. You can change the brightness value of a LED independently of the timer trigger. However, if you set the brightness value to LED_OFF it will also disable the timer trigger.

You can change triggers in a similar manner to the way an IO scheduler is chosen (via /sys/class/leds/<device>/trigger). Trigger specific parameters can appear in /sys/class/leds/<device> once a given trigger is selected.

1.1 Design Philosophy

The underlying design philosophy is simplicity. LEDs are simple devices and the aim is to keep a small amount of code giving as much functionality as possible. Please keep this in mind when suggesting enhancements.

1.2 LED Device Naming

Is currently of the form:

"devicename:color:function"

- **devicename:**
  it should refer to a unique identifier created by the kernel, like e.g. phyN for network devices or inputN for input devices, rather than to the hardware; the information related to the product and the bus to which given device is hooked is available in sysfs

and can be retrieved using get_led_device_info.sh script from tools/leds; generally this section is expected mostly for LEDs that are somehow associated with other devices.

- **color:**
  one of LED_COLOR_ID_* definitions from the header include/dt-bindings/leds/common.h.
- **function:**
  one of LED_FUNCTION_* definitions from the header include/dt-bindings/leds/common.h.

If required color or function is missing, please submit a patch to linux-leds@vger.kernel.org.

It is possible that more than one LED with the same color and function will be required for given platform, differing only with an ordinal number. In this case it is preferable to just concatenate the predefined LED_FUNCTION_* name with required "-N" suffix in the driver. fwnode based drivers can use function-enumerator property for that and then the concatenation will be handled automatically by the LED core upon LED class device registration.

LED subsystem has also a protection against name clash, that may occur when LED class device is created by a driver of hot-pluggable device and it doesn't provide unique devicename section. In this case numerical suffix (e.g. "_1", "_2", "_3" etc.) is added to the requested LED class device name.

There might be still LED class drivers around using vendor or product name for devicename, but this approach is now deprecated as it doesn't convey any added value. Product information can be found in other places in sysfs (see tools/leds/get_led_device_info.sh).

Examples of proper LED names:

- "red:disk"
- "white:flash"
- "red:indicator"
- "phy1:green:wlan"
- "phy3::wlan"
- ":kbd_backlight"
- "input5::kbd_backlight"
- "input3::numlock"
- "input3::scrolllock"
- "input3::capslock"
- "mmc1::status"
- "white:status"

get_led_device_info.sh script can be used for verifying if the LED name meets the requirements pointed out here. It performs validation of the LED class devicename sections and gives hints on expected value for a section in case the validation fails for it. So far the script supports validation of associations between LEDs and following types of devices:

- input devices
- ieee80211 compliant USB devices

The script is open to extensions.

There have been calls for LED properties such as color to be exported as individual led class attributes. As a solution which doesn't incur as much overhead, I suggest these become part of the device name. The naming scheme above leaves scope for further attributes should they be needed. If sections of the name don't apply, just leave that section blank.

1.3 Brightness setting API

LED subsystem core exposes following API for setting brightness:

- **led_set_brightness:**
  it is guaranteed not to sleep, passing LED_OFF stops blinking,
- **led_set_brightness_sync:**
  for use cases when immediate effect is desired - it can block the caller for the time required for accessing device registers and can sleep, passing LED_OFF stops hardware blinking, returns -EBUSY if software blink fallback is enabled.

1.4 LED registration API

A driver wanting to register a LED classdev for use by other drivers / userspace needs to allocate and fill a led_classdev struct and then call *[devm_]led_classdev_register*. If the non devm version is used the driver must call led_classdev_unregister from its remove function before free-ing the led_classdev struct.

If the driver can detect hardware initiated brightness changes and thus wants to have a brightness_hw_changed attribute then the LED_BRIGHT_HW_CHANGED flag must be set in flags before registering. Calling led_classdev_notify_brightness_hw_changed on a classdev not registered with the LED_BRIGHT_HW_CHANGED flag is a bug and will trigger a WARN_ON.

1.5 Hardware accelerated blink of LEDs

Some LEDs can be programmed to blink without any CPU interaction. To support this feature, a LED driver can optionally implement the blink_set() function (see <linux/leds.h>). To set an LED to blinking, however, it is better to use the API function led_blink_set(), as it will check and implement software fallback if necessary.

To turn off blinking, use the API function led_brightness_set() with brightness value LED_OFF, which should stop any software timers that may have been required for blinking.

The blink_set() function should choose a user friendly blinking value if it is called with **delay_on==0* && **delay_off==0* parameters. In this case the driver should give back the chosen value through delay_on and delay_off parameters to the leds subsystem.

Setting the brightness to zero with brightness_set() callback function should completely turn off the LED and cancel the previously programmed hardware blinking function, if any.

1.6 Hardware driven LEDs

Some LEDs can be programmed to be driven by hardware. This is not limited to blink but also to turn off or on autonomously. To support this feature, a LED needs to implement various additional ops and needs to declare specific support for the supported triggers.

With hw control we refer to the LED driven by hardware.

LED driver must define the following value to support hw control:

- **hw_control_trigger:**
  unique trigger name supported by the LED in hw control mode.

LED driver must implement the following API to support hw control:

- **hw_control_is_supported:**
  check if the flags passed by the supported trigger can be parsed and activate hw control on the LED.

  Return 0 if the passed flags mask is supported and can be set with hw_control_set().

  If the passed flags mask is not supported -EOPNOTSUPP must be returned, the LED trigger will use software fallback in this case.

  Return a negative error in case of any other error like device not ready or timeouts.

- **hw_control_set:**
  activate hw control. LED driver will use the provided flags passed from the supported trigger, parse them to a set of mode and setup the LED to be driven by hardware following the requested modes.

  Set LED_OFF via the brightness_set to deactivate hw control.

  Return 0 on success, a negative error number on failing to apply flags.

- **hw_control_get:**
  get active modes from a LED already in hw control, parse them and set in flags the current active flags for the supported trigger.

  Return 0 on success, a negative error number on failing parsing the initial mode. Error from this function is NOT FATAL as the device may be in a not supported initial state by the attached LED trigger.

- **hw_control_get_device:**
  return the device associated with the LED driver in hw control. A trigger might use this to match the returned device from this function with a configured device for the trigger as the source for blinking events and correctly enable hw control. (example a netdev trigger configured to blink for a particular dev match the returned dev from get_device to set hw control)

  Returns a pointer to a struct device or NULL if nothing is currently attached.

LED driver can activate additional modes by default to workaround the impossibility of supporting each different mode on the supported trigger. Examples are hardcoding the blink speed to a set interval, enable special feature like bypassing blink if some requirements are not met.

A trigger should first check if the hw control API are supported by the LED driver and check if the trigger is supported to verify if hw control is possible, use hw_control_is_supported to check if the flags are supported and only at the end use hw_control_set to activate hw control.

A trigger can use hw_control_get to check if a LED is already in hw control and init their flags.

When the LED is in hw control, no software blink is possible and doing so will effectively disable hw control.

1.7 Known Issues

The LED Trigger core cannot be a module as the simple trigger functions would cause nightmare dependency issues. I see this as a minor issue compared to the benefits the simple trigger functionality brings. The rest of the LED subsystem can be modular.

CHAPTER

TWO

FLASH LED HANDLING UNDER LINUX

Some LED devices provide two modes - torch and flash. In the LED subsystem those modes are supported by LED class (see *LED handling under Linux*) and LED Flash class respectively. The torch mode related features are enabled by default and the flash ones only if a driver declares it by setting LED_DEV_CAP_FLASH flag.

In order to enable the support for flash LEDs CONFIG_LEDS_CLASS_FLASH symbol must be defined in the kernel config. A LED Flash class driver must be registered in the LED subsystem with led_classdev_flash_register function.

Following sysfs attributes are exposed for controlling flash LED devices: (see Documentation/ABI/testing/sysfs-class-led-flash)

- flash_brightness
- max_flash_brightness
- flash_timeout
- max_flash_timeout
- flash_strobe
- flash_fault

2.1 V4L2 flash wrapper for flash LEDs

A LED subsystem driver can be controlled also from the level of VideoForLinux2 subsystem. In order to enable this CONFIG_V4L2_FLASH_LED_CLASS symbol has to be defined in the kernel config.

The driver must call the v4l2_flash_init function to get registered in the V4L2 subsystem. The function takes six arguments:

- **dev:**
  flash device, e.g. an I2C device
- **of_node:**
  of_node of the LED, may be NULL if the same as device's
- **fled_cdev:**
  LED flash class device to wrap

- **iled_cdev:**
  LED flash class device representing indicator LED associated with fled_cdev, may be NULL
- **ops:**
  V4L2 specific ops
  - **external_strobe_set**
    defines the source of the flash LED strobe - V4L2_CID_FLASH_STROBE control or external source, typically a sensor, which makes it possible to synchronise the flash strobe start with exposure start,
  - **intensity_to_led_brightness and led_brightness_to_intensity**
    perform enum led_brightness <-> V4L2 intensity conversion in a device specific manner - they can be used for devices with non-linear LED current scale.
- **config:**
  configuration for V4L2 Flash sub-device
  - **dev_name**
    the name of the media entity, unique in the system,
  - **flash_faults**
    bitmask of flash faults that the LED flash class device can report; corresponding LED_FAULT* bit definitions are available in <linux/led-class-flash.h>,
  - **torch_intensity**
    constraints for the LED in TORCH mode in microamperes,
  - **indicator_intensity**
    constraints for the indicator LED in microamperes,
  - **has_external_strobe**
    determines whether the flash strobe source can be switched to external,

On remove the v4l2_flash_release function has to be called, which takes one argument - struct v4l2_flash pointer returned previously by v4l2_flash_init. This function can be safely called with NULL or error pointer argument.

Please refer to drivers/leds/leds-max77693.c for an exemplary usage of the v4l2 flash wrapper.

Once the V4L2 sub-device is registered by the driver which created the Media controller device, the sub-device node acts just as a node of a native V4L2 flash API device would. The calls are simply routed to the LED flash API.

Opening the V4L2 flash sub-device makes the LED subsystem sysfs interface unavailable. The interface is re-enabled after the V4L2 flash sub-device is closed.

CHAPTER

THREE

MULTICOLOR LED HANDLING UNDER LINUX

3.1 Description

The multicolor class groups monochrome LEDs and allows controlling two aspects of the final combined color: hue and lightness. The former is controlled via the multi_intensity array file and the latter is controlled via brightness file.

3.2 Multicolor Class Control

The multicolor class presents files that groups the colors as indexes in an array. These files are children under the LED parent node created by the led_class framework. The led_class framework is documented in led-class.rst within this documentation directory.

Each colored LED will be indexed under the multi_* files. The order of the colors will be arbitrary. The multi_index file can be read to determine the color name to indexed value.

The multi_index file is an array that contains the string list of the colors as they are defined in each multi_* array file.

The multi_intensity is an array that can be read or written to for the individual color intensities. All elements within this array must be written in order for the color LED intensities to be updated.

3.3 Directory Layout Example

root:/sys/class/leds/multicolor:status# ls -lR -rw-r--r-- 1 root root 4096 Oct 19 16:16 brightness -r--r--r-- 1 root root 4096 Oct 19 16:16 max_brightness -r--r--r-- 1 root root 4096 Oct 19 16:16 multi_index -rw-r--r-- 1 root root 4096 Oct 19 16:16 multi_intensity

3.4 Multicolor Class Brightness Control

The brightness level for each LED is calculated based on the color LED intensity setting divided by the global max_brightness setting multiplied by the requested brightness.

led_brightness = brightness * multi_intensity/max_brightness

Example: A user first writes the multi_intensity file with the brightness levels for each LED that are necessary to achieve a certain color output from a multicolor LED group.

cat /sys/class/leds/multicolor:status/multi_index green blue red

echo 43 226 138 > /sys/class/leds/multicolor:status/multi_intensity

red -
intensity = 138 max_brightness = 255

green -
intensity = 43 max_brightness = 255

blue -
intensity = 226 max_brightness = 255

The user can control the brightness of that multicolor LED group by writing the global 'brightness' control. Assuming a max_brightness of 255 the user may want to dim the LED color group to half. The user would write a value of 128 to the global brightness file then the values written to each LED will be adjusted base on this value.

cat /sys/class/leds/multicolor:status/max_brightness 255 echo 128 > /sys/class/leds/multicolor:status/brightness

adjusted_red_value = 128 * 138/255 = 69 adjusted_green_value = 128 * 43/255 = 21 adjusted_blue_value = 128 * 226/255 = 113

Reading the global brightness file will return the current brightness value of the color LED group.

cat /sys/class/leds/multicolor:status/brightness 128

CHAPTER

FOUR

ONE-SHOT LED TRIGGER

This is a LED trigger useful for signaling the user of an event where there are no clear trap points to put standard led-on and led-off settings. Using this trigger, the application needs only to signal the trigger when an event has happened, then the trigger turns the LED on and then keeps it off for a specified amount of time.

This trigger is meant to be usable both for sporadic and dense events. In the first case, the trigger produces a clear single controlled blink for each event, while in the latter it keeps blinking at constant rate, as to signal that the events are arriving continuously.

A one-shot LED only stays in a constant state when there are no events. An additional “invert” property specifies if the LED has to stay off (normal) or on (inverted) when not rearmed.

The trigger can be activated from user space on led class devices as shown below:

```
echo oneshot > trigger
```

This adds sysfs attributes to the LED that are documented in: Documentation/ABI/testing/sysfs-class-led-trigger-oneshot

Example use-case: network devices, initialization:

```
echo oneshot > trigger # set trigger for this led
echo 33 > delay_on     # blink at 1 / (33 + 33) Hz on continuous traffic
echo 33 > delay_off
```

interface goes up:

```
echo 1 > invert # set led as normally-on, turn the led on
```

packet received/transmitted:

```
echo 1 > shot # led starts blinking, ignored if already blinking
```

interface goes down:

```
echo 0 > invert # set led as normally-off, turn the led off
```

CHAPTER

FIVE

LED TRANSIENT TRIGGER

The leds timer trigger does not currently have an interface to activate a one shot timer. The current support allows for setting two timers, one for specifying how long a state to be on, and the second for how long the state to be off. The delay_on value specifies the time period an LED should stay in on state, followed by a delay_off value that specifies how long the LED should stay in off state. The on and off cycle repeats until the trigger gets deactivated. There is no provision for one time activation to implement features that require an on or off state to be held just once and then stay in the original state forever.

Without one shot timer interface, user space can still use timer trigger to set a timer to hold a state, however when user space application crashes or goes away without deactivating the timer, the hardware will be left in that state permanently.

Transient trigger addresses the need for one shot timer activation. The transient trigger can be enabled and disabled just like the other leds triggers.

When an led class device driver registers itself, it can specify all leds triggers it supports and a default trigger. During registration, activation routine for the default trigger gets called. During registration of an led class device, the LED state does not change.

When the driver unregisters, deactivation routine for the currently active trigger will be called, and LED state is changed to LED_OFF.

Driver suspend changes the LED state to LED_OFF and resume doesn't change the state. Please note that there is no explicit interaction between the suspend and resume actions and the currently enabled trigger. LED state changes are suspended while the driver is in suspend state. Any timers that are active at the time driver gets suspended, continue to run, without being able to actually change the LED state. Once driver is resumed, triggers start functioning again.

LED state changes are controlled using brightness which is a common led class device property. When brightness is set to 0 from user space via echo 0 > brightness, it will result in deactivating the current trigger.

Transient trigger uses standard register and unregister interfaces. During trigger registration, for each led class device that specifies this trigger as its default trigger, trigger activation routine will get called. During registration, the LED state does not change, unless there is another trigger active, in which case LED state changes to LED_OFF.

During trigger unregistration, LED state gets changed to LED_OFF.

Transient trigger activation routine doesn't change the LED state. It creates its properties and does its initialization. Transient trigger deactivation routine, will cancel any timer that is active before it cleans up and removes the properties it created. It will restore the LED state to non-

transient state. When driver gets suspended, irrespective of the transient state, the LED state changes to LED_OFF.

Transient trigger can be enabled and disabled from user space on led class devices, that support this trigger as shown below:

```
echo transient > trigger
echo none > trigger
```

NOTE:
Add a new property trigger state to control the state.

This trigger exports three properties, activate, state, and duration. When transient trigger is activated these properties are set to default values.

- duration allows setting timer value in msecs. The initial value is 0.
- activate allows activating and deactivating the timer specified by duration as needed. The initial and default value is 0. This will allow duration to be set after trigger activation.
- state allows user to specify a transient state to be held for the specified duration.

activate

- one shot timer activate mechanism. 1 when activated, 0 when deactivated. default value is zero when transient trigger is enabled, to allow duration to be set.

  activate state indicates a timer with a value of specified duration running. deactivated state indicates that there is no active timer running.

duration

- one shot timer value. When activate is set, duration value is used to start a timer that runs once. This value doesn't get changed by the trigger unless user does a set via echo new_value > duration

state

- transient state to be held. It has two values 0 or 1. 0 maps to LED_OFF and 1 maps to LED_FULL. The specified state is held for the duration of the one shot timer and then the state gets changed to the non-transient state which is the inverse of transient state. If state = LED_FULL, when the timer runs out the state will go back to LED_OFF. If state = LED_OFF, when the timer runs out the state will go back to LED_FULL. Please note that current LED state is not checked prior to changing the state to the specified state. Driver could map these values to inverted depending on the default states it defines for the LED in its brightness_set() interface which is called from the led brightness_set() interfaces to control the LED state.

When timer expires activate goes back to deactivated state, duration is left at the set value to be used when activate is set at a future time. This will allow user app to set the time once and activate it to run it once for the specified value as needed. When timer expires, state is restored to the non-transient state which is the inverse of the transient state:

echo 1 > activate	starts timer = duration when duration is not 0.
echo 0 > activate	cancels currently running timer.
echo n > duration	stores timer value to be used upon next activate. Currently active timer if any, continues to run for the specified time.
echo 0 > duration	stores timer value to be used upon next activate. Currently active timer if any, continues to run for the specified time.
echo 1 > state	stores desired transient state LED_FULL to be held for the specified duration.
echo 0 > state	stores desired transient state LED_OFF to be held for the specified duration.

5.1 What is not supported

- Timer activation is one shot and extending and/or shortening the timer is not supported.

5.2 Examples

use-case 1:

```
echo transient > trigger
echo n > duration
echo 1 > state
```

repeat the following step as needed:

```
echo 1 > activate - start timer = duration to run once
echo 1 > activate - start timer = duration to run once
echo none > trigger
```

This trigger is intended to be used for the following example use cases:

- Use of LED by user space app as activity indicator.
- Use of LED by user space app as a kind of watchdog indicator -- as long as the app is alive, it can keep the LED illuminated, if it dies the LED will be extinguished automatically.
- Use by any user space app that needs a transient GPIO output.

CHAPTER

SIX

USB PORT LED TRIGGER

This LED trigger can be used for signalling to the user a presence of USB device in a given port. It simply turns on LED when device appears and turns it off when it disappears.

It requires selecting USB ports that should be observed. All available ones are listed as separated entries in a “ports” subdirectory. Selecting is handled by echoing “1” to a chosen port.

Please note that this trigger allows selecting multiple USB ports for a single LED.

This can be useful in two cases:

6.1 1) Device with single USB LED and few physical ports

In such a case LED will be turned on as long as there is at least one connected USB device.

6.2 2) Device with a physical port handled by few controllers

Some devices may have one controller per PHY standard. E.g. USB 3.0 physical port may be handled by ohci-platform, ehci-platform and xhci-hcd. If there is only one LED user will most likely want to assign ports from all 3 hubs.

This trigger can be activated from user space on led class devices as shown below:

```
echo usbport > trigger
```

This adds sysfs attributes to the LED that are documented in: Documentation/ABI/testing/sysfs-class-led-trigger-usbport

Example use-case:

```
echo usbport > trigger
echo 1 > ports/usb1-port1
echo 1 > ports/usb2-port1
cat ports/usb1-port1
echo 0 > ports/usb1-port1
```

CHAPTER

SEVEN

USERSPACE LEDS

The uleds driver supports userspace LEDs. This can be useful for testing triggers and can also be used to implement virtual LEDs.

7.1 Usage

When the driver is loaded, a character device is created at /dev/uleds. To create a new LED class device, open /dev/uleds and write a uleds_user_dev structure to it (found in kernel public header file linux/uleds.h):

```
#define LED_MAX_NAME_SIZE 64

struct uleds_user_dev {
    char name[LED_MAX_NAME_SIZE];
};
```

A new LED class device will be created with the name given. The name can be any valid sysfs device node name, but consider using the LED class naming convention of “device-name:color:function”.

The current brightness is found by reading a single byte from the character device. Values are unsigned: 0 to 255. Reading will block until the brightness changes. The device node can also be polled to notify when the brightness value changes.

The LED class device will be removed when the open file handle to /dev/uleds is closed.

Multiple LED class devices are created by opening additional file handles to /dev/uleds.

See tools/leds/uledmon.c for an example userspace program.

CHAPTER

EIGHT

LEDS BLINKM DRIVER

The leds-blinkm driver supports the devices of the BlinkM family.

They are RGB-LED modules driven by a (AT)tiny microcontroller and communicate through I2C. The default address of these modules is 0x09 but this can be changed through a command. By this you could dasy-chain up to 127 BlinkMs on an I2C bus.

The device accepts RGB and HSB color values through separate commands. Also you can store blinking sequences as “scripts” in the controller and run them. Also fading is an option.

The interface this driver provides is 2-fold:

8.1 a) LED class interface for use with triggers

The registration follows the scheme:

```
blinkm-<i2c-bus-nr>-<i2c-device-nr>-<color>

$ ls -h /sys/class/leds/blinkm-6-*
/sys/class/leds/blinkm-6-9-blue:
brightness  device  max_brightness  power  subsystem  trigger  uevent

/sys/class/leds/blinkm-6-9-green:
brightness  device  max_brightness  power  subsystem  trigger  uevent

/sys/class/leds/blinkm-6-9-red:
brightness  device  max_brightness  power  subsystem  trigger  uevent
```

(same is /sys/bus/i2c/devices/6-0009/leds)

We can control the colors separated into red, green and blue and assign triggers on each color.

E.g.:

```
$ cat blinkm-6-9-blue/brightness
05

$ echo 200 > blinkm-6-9-blue/brightness
$

$ modprobe ledtrig-heartbeat
```

```
$ echo heartbeat > blinkm-6-9-green/trigger
$
```

8.2 b) Sysfs group to control rgb, fade, hsb, scripts ...

This extended interface is available as folder blinkm in the sysfs folder of the I2C device. E.g. below /sys/bus/i2c/devices/6-0009/blinkm

> $ ls -h /sys/bus/i2c/devices/6-0009/blinkm/ blue green red test

Currently supported is just setting red, green, blue and a test sequence.

E.g.:

```
$ cat *
00
00
00
#Write into test to start test sequence!#

$ echo 1 > test
$

$ echo 255 > red
$
```

as of 6/2012

dl9pf <at> gmx <dot> de

CHAPTER

NINE

KERNEL DRIVER FOR INTEL CHERRY TRAIL WHISKEY COVE PMIC LEDS

9.1 /sys/class/leds/<led>/hw_pattern

Specify a hardware pattern for the Whiskey Cove PMIC LEDs.

The only supported pattern is hardware breathing mode:

```
"0 2000 1 2000"

    ^
    |
Max-|     ---
    |    /   \
    |   /     \
    |  /       \     /
    | /         \   /
Min-|-           ---
    |
    0------2------4--> time (sec)
```

The rise and fall times must be the same value. Supported values are 2000, 1000, 500 and 250 for breathing frequencies of 1/4, 1/2, 1 and 2 Hz.

The set pattern only controls the timing. For max brightness the last set brightness is used and the max brightness can be changed while breathing by writing the brightness attribute.

This is just like how blinking works in the LED subsystem, for both sw and hw blinking the brightness can also be changed while blinking. Breathing on this hw really is just a variant mode of blinking.

CHAPTER

TEN

KERNEL DRIVER FOR CRANE EL15203000

10.1 /sys/class/leds/<led>/hw_pattern

Specify a hardware pattern for the EL15203000 LED.

The LEDs board supports only predefined patterns by firmware for specific LEDs.

Breathing mode for Screen frame light tube:

```
"0 4000 1 4000"

    ^
    |
Max-|     ---
    |    /   \
    |   /     \
    |  /       \     /
    | /         \   /
Min-|-           ---
    |
    0------4------8--> time (sec)
```

Cascade mode for Pipe LED:

```
  "1 800 2 800 4 800 8 800 16 800"

      ^
      |
0 On -|----+                   +----+                   +---
      |    |                   |    |                   |
  Off-|    +-------------------+    +-------------------+
      |
1 On -|    +----+                   +----+
      |    |    |                   |    |
  Off |----+    +-------------------+    +------------------
      |
2 On -|         +----+                   +----+
      |         |    |                   |    |
  Off-|---------+    +-------------------+    +-------------
      |
3 On -|              +----+                   +----+
```

```
      |              |    |                   |    |
  Off-|--------------+    +-------------------+    +--------
      |
4 On -|                   +----+                   +----+
      |                   |    |                   |    |
  Off-|-------------------+    +-------------------+    +---
      |
      0---0.8--1.6--2.4--3.2---4---4.8--5.6--6.4--7.2---8--> time (sec)
```

Inverted cascade mode for Pipe LED:

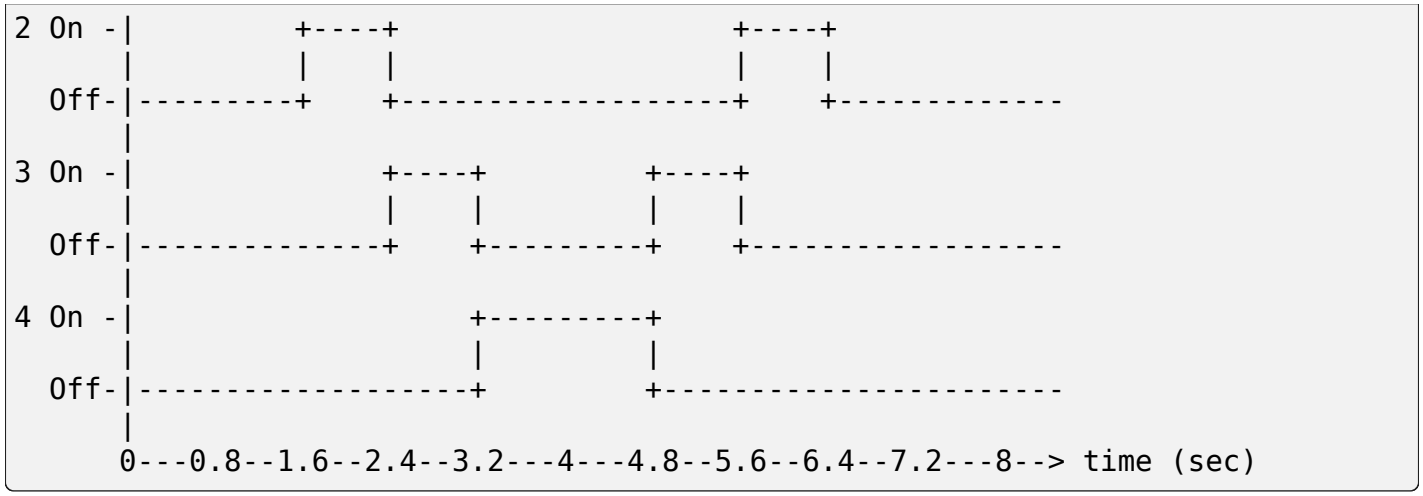
```
  "30 800 29 800 27 800 23 800 15 800"

      ^
      |
0 On -|    +-------------------+    +-------------------+
      |    |                   |    |                   |
  Off-|----+                   +----+                   +---
      |
1 On -|----+    +-------------------+    +------------------
      |    |    |                   |    |
  Off |    +----+                   +----+
      |
2 On -|---------+    +-------------------+    +-------------
      |         |    |                   |    |
  Off-|         +----+                   +----+
      |
3 On -|--------------+    +-------------------+    +--------
      |              |    |                   |    |
  Off-|              +----+                   +----+
      |
4 On -|-------------------+    +-------------------+    +---
      |                   |    |                   |    |
  Off-|                   +----+                   +----+
      |
      0---0.8--1.6--2.4--3.2---4---4.8--5.6--6.4--7.2---8--> time (sec)
```

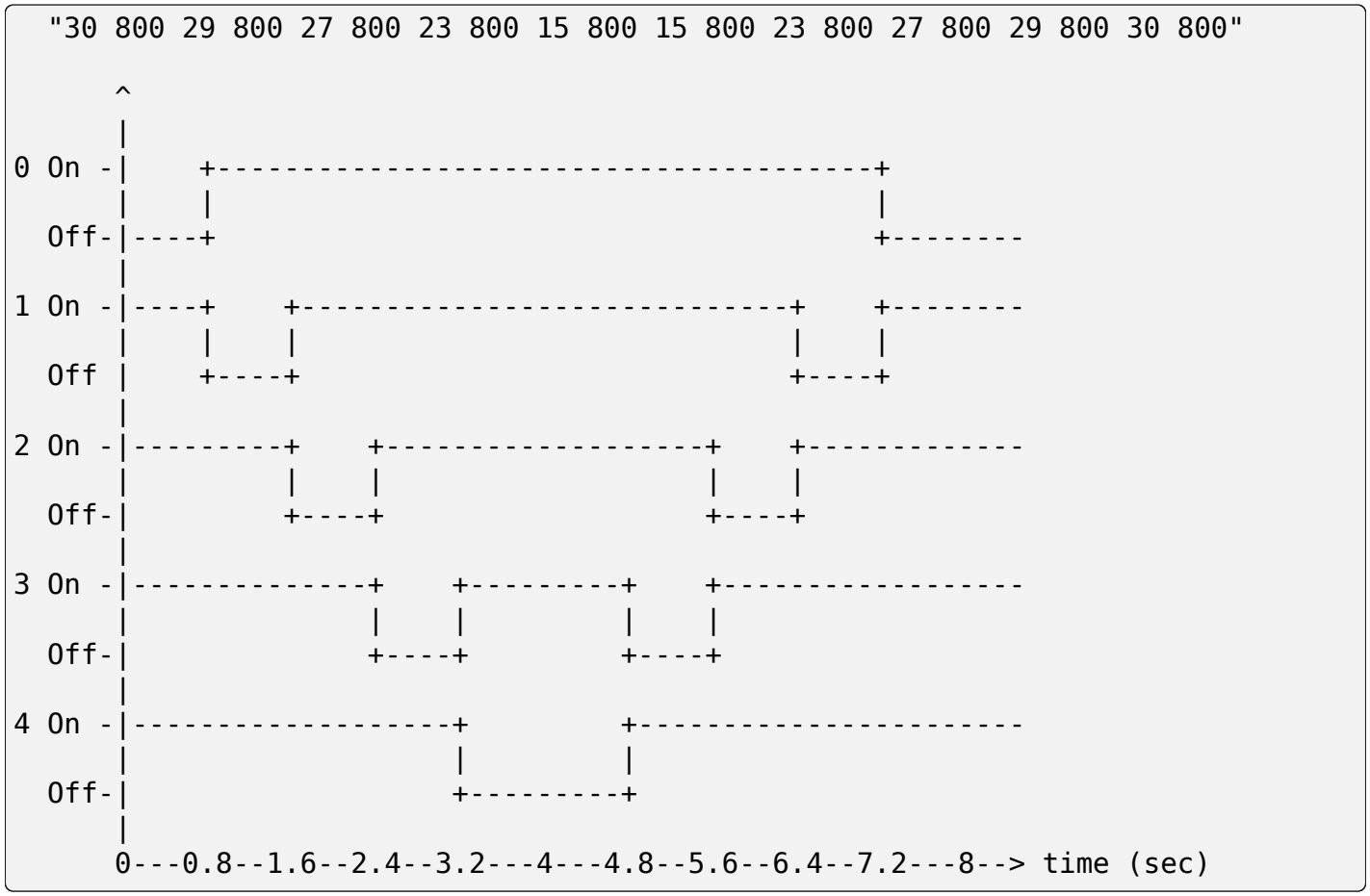
Bounce mode for Pipe LED:

```
  "1 800 2 800 4 800 8 800 16 800 16 800 8 800 4 800 2 800 1 800"

      ^
      |
0 On -|----+                                   +--------
      |    |                                   |
  Off-|    +-----------------------------------+
      |
1 On -|    +----+                         +----+
      |    |    |                         |    |
  Off |----+    +-------------------------+    +--------
      |
```



Inverted bounce mode for Pipe LED:



CHAPTER

ELEVEN

KERNEL DRIVER FOR LM3556

- Texas Instrument: 1.5 A Synchronous Boost LED Flash Driver w/ High-Side Current Source
- Datasheet: http://www.national.com/ds/LM/LM3556.pdf

Authors:

- Daniel Jeong

  Contact:Daniel Jeong(daniel.jeong-at-ti.com, gshark.jeong-at-gmail.com)

11.1 Description

There are 3 functions in LM3556, Flash, Torch and Indicator.

11.1.1 Flash Mode

In Flash Mode, the LED current source(LED) provides 16 target current levels from 93.75 mA to 1500 mA.The Flash currents are adjusted via the CURRENT CONTROL REGISTER(0x09).Flash mode is activated by the ENABLE REGISTER(0x0A), or by pulling the STROBE pin HIGH.

LM3556 Flash can be controlled through /sys/class/leds/flash/brightness file

- if STROBE pin is enabled, below example control brightness only, and ON / OFF will be controlled by STROBE pin.

Flash Example:

OFF:

```
#echo 0 > /sys/class/leds/flash/brightness
```

93.75 mA:

```
#echo 1 > /sys/class/leds/flash/brightness
```

...

1500 mA:

```
#echo 16 > /sys/class/leds/flash/brightness
```

11.1.2 Torch Mode

In Torch Mode, the current source(LED) is programmed via the CURRENT CONTROL REGISTER(0x09).Torch Mode is activated by the ENABLE REGISTER(0x0A) or by the hardware TORCH input.

LM3556 torch can be controlled through /sys/class/leds/torch/brightness file. * if TORCH pin is enabled, below example control brightness only, and ON / OFF will be controlled by TORCH pin.

Torch Example:

OFF:

```
#echo 0 > /sys/class/leds/torch/brightness
```

46.88 mA:

```
#echo 1 > /sys/class/leds/torch/brightness
```

...

375 mA:

```
#echo 8 > /sys/class/leds/torch/brightness
```

11.1.3 Indicator Mode

Indicator pattern can be set through /sys/class/leds/indicator/pattern file, and 4 patterns are pre-defined in indicator_pattern array.

According to N-lank, Pulse time and N Period values, different pattern wiill be generated.If you want new patterns for your own device, change indicator_pattern array with your own values and INDIC_PATTERN_SIZE.

Please refer datasheet for more detail about N-Blank, Pulse time and N Period.

Indicator pattern example:

pattern 0:

```
#echo 0 > /sys/class/leds/indicator/pattern
```

...

pattern 3:

```
#echo 3 > /sys/class/leds/indicator/pattern
```

Indicator brightness can be controlled through sys/class/leds/indicator/brightness file.

Example:

OFF:

```
#echo 0 > /sys/class/leds/indicator/brightness
```

5.86 mA:

```
#echo 1 > /sys/class/leds/indicator/brightness
```

...

46.875mA:

```
#echo 8 > /sys/class/leds/indicator/brightness
```

11.2 Notes

Driver expects it is registered using the i2c_board_info mechanism. To register the chip at address 0x63 on specific adapter, set the platform data according to include/linux/platform_data/leds-lm3556.h, set the i2c board info

Example:

```
static struct i2c_board_info board_i2c_ch4[] __initdata = {
        {
                I2C_BOARD_INFO(LM3556_NAME, 0x63),
                .platform_data = &lm3556_pdata,
         },
};
```

and register it in the platform init function

Example:

```
board_register_i2c_bus(4, 400,
                        board_i2c_ch4, ARRAY_SIZE(board_i2c_ch4));
```

CHAPTER

TWELVE

KERNEL DRIVER LP3944

- National Semiconductor LP3944 Fun-light Chip

  Prefix: 'lp3944'

  Addresses scanned: None (see the Notes section below)

  Datasheet:

  Publicly available at the National Semiconductor website http://www.national.com/pf/LP/LP3944.html

Authors:
Antonio Ospite <ospite@studenti.unina.it>

12.1 Description

The LP3944 is a helper chip that can drive up to 8 leds, with two programmable DIM modes; it could even be used as a gpio expander but this driver assumes it is used as a led controller.

The DIM modes are used to set _blink_ patterns for leds, the pattern is specified supplying two parameters:

- **period:**
  from 0s to 1.6s
- **duty cycle:**
  percentage of the period the led is on, from 0 to 100

Setting a led in DIM0 or DIM1 mode makes it blink according to the pattern. See the datasheet for details.

LP3944 can be found on Motorola A910 smartphone, where it drives the rgb leds, the camera flash light and the lcds power.

12.2 Notes

The chip is used mainly in embedded contexts, so this driver expects it is registered using the i2c_board_info mechanism.

To register the chip at address 0x60 on adapter 0, set the platform data according to include/linux/leds-lp3944.h, set the i2c board info:

```
static struct i2c_board_info a910_i2c_board_info[] __initdata = {
        {
                I2C_BOARD_INFO("lp3944", 0x60),
                .platform_data = &a910_lp3944_leds,
        },
};
```

and register it in the platform init function:

```
i2c_register_board_info(0, a910_i2c_board_info,
                ARRAY_SIZE(a910_i2c_board_info));
```

CHAPTER

THIRTEEN

KERNEL DRIVER FOR LP5521

- National Semiconductor LP5521 led driver chip
- Datasheet: http://www.national.com/pf/LP/LP5521.html

Authors: Mathias Nyman, Yuri Zaporozhets, Samu Onkalo

Contact: Samu Onkalo (samu.p.onkalo-at-nokia.com)

13.1 Description

LP5521 can drive up to 3 channels. Leds can be controlled directly via the led class control interface. Channels have generic names: lp5521:channelx, where x is 0 .. 2

All three channels can be also controlled using the engine micro programs. More details of the instructions can be found from the public data sheet.

LP5521 has the internal program memory for running various LED patterns. There are two ways to run LED patterns.

1) Legacy interface - enginex_mode and enginex_load Control interface for the engines:

   x is 1 .. 3

   enginex_mode:
   disabled, load, run

   enginex_load:
   store program (visible only in engine load mode)

   Example (start to blink the channel 2 led):

```
cd   /sys/class/leds/lp5521:channel2/device
echo "load" > engine3_mode
echo "037f4d0003ff6000" > engine3_load
echo "run" > engine3_mode
```

   To stop the engine:

```
echo "disabled" > engine3_mode
```

2) Firmware interface - LP55xx common interface

For the details, please refer to 'firmware' section in *LP5521/LP5523/LP55231/LP5562/LP8501 Common Driver*

sysfs contains a selftest entry.

The test communicates with the chip and checks that the clock mode is automatically set to the requested one.

Each channel has its own led current settings.

- /sys/class/leds/lp5521:channel0/led_current - RW
- /sys/class/leds/lp5521:channel0/max_current - RO

Format: 10x mA i.e 10 means 1.0 mA

example platform data:

```
static struct lp55xx_led_config lp5521_led_config[] = {
        {
              .name = "red",
                .chan_nr        = 0,
                .led_current    = 50,
              .max_current    = 130,
        }, {
              .name = "green",
                .chan_nr        = 1,
                .led_current    = 0,
              .max_current    = 130,
        }, {
              .name = "blue",
                .chan_nr        = 2,
                .led_current    = 0,
              .max_current    = 130,
        }
};

static int lp5521_setup(void)
{
      /* setup HW resources */
}

static void lp5521_release(void)
{
      /* Release HW resources */
}

static void lp5521_enable(bool state)
{
      /* Control of chip enable signal */
}

static struct lp55xx_platform_data lp5521_platform_data = {
        .led_config     = lp5521_led_config,
        .num_channels   = ARRAY_SIZE(lp5521_led_config),
        .clock_mode     = LP55XX_CLOCK_EXT,
        .setup_resources   = lp5521_setup,
```

```
        .release_resources = lp5521_release,
        .enable            = lp5521_enable,
};
```

Note:

chan_nr can have values between 0 and 2. The name of each channel can be configurable. If the name field is not defined, the default name will be set to 'xxxx:channelN' (XXXX : pdata->label or i2c client name, N : channel number)

If the current is set to 0 in the platform data, that channel is disabled and it is not visible in the sysfs.

CHAPTER

FOURTEEN

KERNEL DRIVER FOR LP5523

- National Semiconductor LP5523 led driver chip
- Datasheet: http://www.national.com/pf/LP/LP5523.html

Authors: Mathias Nyman, Yuri Zaporozhets, Samu Onkalo Contact: Samu Onkalo (samu.p.onkalo-at-nokia.com)

14.1 Description

LP5523 can drive up to 9 channels. Leds can be controlled directly via the led class control interface. The name of each channel is configurable in the platform data - name and label. There are three options to make the channel name.

a) Define the ‘name’ in the platform data

To make specific channel name, then use ‘name’ platform data.

- /sys/class/leds/R1 (name: ‘R1’)
- /sys/class/leds/B1 (name: ‘B1’)

b) Use the ‘label’ with no ‘name’ field

For one device name with channel number, then use ‘label’. - /sys/class/leds/RGB:channelN (label: ‘RGB’, N: 0 ~ 8)

c) Default

If both fields are NULL, ‘lp5523’ is used by default. - /sys/class/leds/lp5523:channelN (N: 0 ~ 8)

LP5523 has the internal program memory for running various LED patterns. There are two ways to run LED patterns.

1) Legacy interface - enginex_mode, enginex_load and enginex_leds

   Control interface for the engines:

   x is 1 .. 3

   enginex_mode:
   disabled, load, run

   enginex_load:
   microcode load

enginex_leds:
led mux control

```
cd /sys/class/leds/lp5523:channel2/device
echo "load" > engine3_mode
echo "9d80400004ff05ff437f0000" > engine3_load
echo "111111111" > engine3_leds
echo "run" > engine3_mode
```

To stop the engine:

```
echo "disabled" > engine3_mode
```

2) Firmware interface - LP55xx common interface

For the details, please refer to 'firmware' section in *LP5521/LP5523/LP55231/LP5562/LP8501 Common Driver*

LP5523 has three master faders. If a channel is mapped to one of the master faders, its output is dimmed based on the value of the master fader.

For example:

```
echo "123000123" > master_fader_leds
```

creates the following channel-fader mappings:

```
channel 0,6 to master_fader1
channel 1,7 to master_fader2
channel 2,8 to master_fader3
```

Then, to have 25% of the original output on channel 0,6:

```
echo 64 > master_fader1
```

To have 0% of the original output (i.e. no output) channel 1,7:

```
echo 0 > master_fader2
```

To have 100% of the original output (i.e. no dimming) on channel 2,8:

```
echo 255 > master_fader3
```

To clear all master fader controls:

```
echo "000000000" > master_fader_leds
```

Selftest uses always the current from the platform data.

Each channel contains led current settings. - /sys/class/leds/lp5523:channel2/led_current - RW - /sys/class/leds/lp5523:channel2/max_current - RO

Format: 10x mA i.e 10 means 1.0 mA

Example platform data:

```
static struct lp55xx_led_config lp5523_led_config[] = {
        {
                .name           = "D1",
                .chan_nr        = 0,
                .led_current    = 50,
                .max_current    = 130,
        },
...
        {
                .chan_nr        = 8,
                .led_current    = 50,
                .max_current    = 130,
        }
};

static int lp5523_setup(void)
{
        /* Setup HW resources */
}

static void lp5523_release(void)
{
        /* Release HW resources */
}

static void lp5523_enable(bool state)
{
        /* Control chip enable signal */
}

static struct lp55xx_platform_data lp5523_platform_data = {
        .led_config     = lp5523_led_config,
        .num_channels   = ARRAY_SIZE(lp5523_led_config),
        .clock_mode     = LP55XX_CLOCK_EXT,
        .setup_resources   = lp5523_setup,
        .release_resources = lp5523_release,
        .enable            = lp5523_enable,
};
```

Note

chan_nr can have values between 0 and 8.

CHAPTER

FIFTEEN

KERNEL DRIVER FOR LP5562

- TI LP5562 LED Driver

Author: Milo(Woogyom) Kim <milo.kim@ti.com>

15.1 Description

LP5562 can drive up to 4 channels. R/G/B and White. LEDs can be controlled directly via the led class control interface.

All four channels can be also controlled using the engine micro programs. LP5562 has the internal program memory for running various LED patterns. For the details, please refer to 'firmware' section in *LP5521/LP5523/LP55231/LP5562/LP8501 Common Driver*

15.2 Device attribute

engine_mux

3 Engines are allocated in LP5562, but the number of channel is 4. Therefore each channel should be mapped to the engine number.

Value: RGB or W

This attribute is used for programming LED data with the firmware interface. Unlike the LP5521/LP5523/55231, LP5562 has unique feature for the engine mux, so additional sysfs is required

LED Map

Red	...	Engine 1 (fixed)
Green	...	Engine 2 (fixed)
Blue	...	Engine 3 (fixed)
White	...	Engine 1 or 2 or 3 (selective)

15.3 How to load the program data using engine_mux

Before loading the LP5562 program data, engine_mux should be written between the engine selection and loading the firmware. Engine mux has two different mode, RGB and W. RGB is used for loading RGB program data, W is used for W program data.

For example, run blinking green channel pattern:

```
echo 2 > /sys/bus/i2c/devices/xxxx/select_engine     # 2 is for green␣
↪channel
echo "RGB" > /sys/bus/i2c/devices/xxxx/engine_mux    # engine mux for␣
↪RGB
echo 1 > /sys/class/firmware/lp5562/loading
echo "4000600040FF6000" > /sys/class/firmware/lp5562/data
echo 0 > /sys/class/firmware/lp5562/loading
echo 1 > /sys/bus/i2c/devices/xxxx/run_engine
```

To run a blinking white pattern:

```
echo 1 or 2 or 3 > /sys/bus/i2c/devices/xxxx/select_engine
echo "W" > /sys/bus/i2c/devices/xxxx/engine_mux
echo 1 > /sys/class/firmware/lp5562/loading
echo "4000600040FF6000" > /sys/class/firmware/lp5562/data
echo 0 > /sys/class/firmware/lp5562/loading
echo 1 > /sys/bus/i2c/devices/xxxx/run_engine
```

15.4 How to load the predefined patterns

Please refer to '*LP5521/LP5523/LP55231/LP5562/LP8501 Common Driver*"

15.5 Setting Current of Each Channel

Like LP5521 and LP5523/55231, LP5562 provides LED current settings. The 'led_current' and 'max_current' are used.

15.6 Example of Platform data

```
static struct lp55xx_led_config lp5562_led_config[] = {
        {
                .name           = "R",
                .chan_nr        = 0,
                .led_current    = 20,
                .max_current    = 40,
        },
        {
                .name           = "G",
                .chan_nr        = 1,
```

```
                .led_current    = 20,
                .max_current    = 40,
        },
        {
                .name           = "B",
                .chan_nr        = 2,
                .led_current    = 20,
                .max_current    = 40,
        },
        {
                .name           = "W",
                .chan_nr        = 3,
                .led_current    = 20,
                .max_current    = 40,
        },
};

static int lp5562_setup(void)
{
        /* setup HW resources */
}

static void lp5562_release(void)
{
        /* Release HW resources */
}

static void lp5562_enable(bool state)
{
        /* Control of chip enable signal */
}

static struct lp55xx_platform_data lp5562_platform_data = {
        .led_config     = lp5562_led_config,
        .num_channels   = ARRAY_SIZE(lp5562_led_config),
        .setup_resources   = lp5562_setup,
        .release_resources = lp5562_release,
        .enable            = lp5562_enable,
};
```

To configure the platform specific data, lp55xx_platform_data structure is used

If the current is set to 0 in the platform data, that channel is disabled and it is not visible in the sysfs.

CHAPTER

SIXTEEN

LP5521/LP5523/LP55231/LP5562/LP8501 COMMON DRIVER

Authors: Milo(Woogyom) Kim <milo.kim@ti.com>

16.1 Description

LP5521, LP5523/55231, LP5562 and LP8501 have common features as below.

> Register access via the I2C Device initialization/deinitialization Create LED class devices for multiple output channels Device attributes for user-space interface Program memory for running LED patterns

The LP55xx common driver provides these features using exported functions.

> lp55xx_init_device() / lp55xx_deinit_device() lp55xx_register_leds() / lp55xx_unregister_leds() lp55xx_regsister_sysfs() / lp55xx_unregister_sysfs()

(Driver Structure Data)

In lp55xx common driver, two different data structure is used.

- **lp55xx_led**
  control multi output LED channels such as led current, channel index.
- **lp55xx_chip**
  general chip control such like the I2C and platform data.

For example, LP5521 has maximum 3 LED channels. LP5523/55231 has 9 output channels:

```
lp55xx_chip for LP5521 ... lp55xx_led #1
                           lp55xx_led #2
                           lp55xx_led #3

lp55xx_chip for LP5523 ... lp55xx_led #1
                           lp55xx_led #2
                                 .
                                 .
                           lp55xx_led #9
```

(Chip Dependent Code)

To support device specific configurations, special structure 'lpxx_device_config' is used.

- Maximum number of channels
- Reset command, chip enable command

- Chip specific initialization
- Brightness control register access
- Setting LED output current
- Program memory address access for running patterns
- Additional device specific attributes

(Firmware Interface)

LP55xx family devices have the internal program memory for running various LED patterns.

This pattern data is saved as a file in the user-land or hex byte string is written into the memory through the I2C.

LP55xx common driver supports the firmware interface.

LP55xx chips have three program engines.

To load and run the pattern, the programming sequence is following.

(1) Select an engine number (1/2/3)

(2) Mode change to load

(3) Write pattern data into selected area

(4) Mode change to run

The LP55xx common driver provides simple interfaces as below.

select_engine:
Select which engine is used for running program

run_engine:
Start program which is loaded via the firmware interface

firmware:
Load program data

In case of LP5523, one more command is required, 'enginex_leds'. It is used for selecting LED output(s) at each engine number. In more details, please refer to '*Kernel driver for lp5523*'.

For example, run blinking pattern in engine #1 of LP5521:

```
echo 1 > /sys/bus/i2c/devices/xxxx/select_engine
echo 1 > /sys/class/firmware/lp5521/loading
echo "4000600040FF6000" > /sys/class/firmware/lp5521/data
echo 0 > /sys/class/firmware/lp5521/loading
echo 1 > /sys/bus/i2c/devices/xxxx/run_engine
```

For example, run blinking pattern in engine #3 of LP55231

Two LEDs are configured as pattern output channels:

```
echo 3 > /sys/bus/i2c/devices/xxxx/select_engine
echo 1 > /sys/class/firmware/lp55231/loading
echo "9d0740ff7e0040007e00a0010000" > /sys/class/firmware/lp55231/data
echo 0 > /sys/class/firmware/lp55231/loading
```

```
echo "000001100" > /sys/bus/i2c/devices/xxxx/engine3_leds
echo 1 > /sys/bus/i2c/devices/xxxx/run_engine
```

To start blinking patterns in engine #2 and #3 simultaneously:

```
for idx in 2 3
do
echo $idx > /sys/class/leds/red/device/select_engine
sleep 0.1
echo 1 > /sys/class/firmware/lp5521/loading
echo "4000600040FF6000" > /sys/class/firmware/lp5521/data
echo 0 > /sys/class/firmware/lp5521/loading
done
echo 1 > /sys/class/leds/red/device/run_engine
```

Here is another example for LP5523.

Full LED strings are selected by 'engine2_leds':

```
echo 2 > /sys/bus/i2c/devices/xxxx/select_engine
echo 1 > /sys/class/firmware/lp5523/loading
echo "9d80400004ff05ff437f0000" > /sys/class/firmware/lp5523/data
echo 0 > /sys/class/firmware/lp5523/loading
echo "111111111" > /sys/bus/i2c/devices/xxxx/engine2_leds
echo 1 > /sys/bus/i2c/devices/xxxx/run_engine
```

As soon as 'loading' is set to 0, registered callback is called. Inside the callback, the selected engine is loaded and memory is updated. To run programmed pattern, 'run_engine' attribute should be enabled.

The pattern sequence of LP8501 is similar to LP5523.

However pattern data is specific.

Ex 1) Engine 1 is used:

```
echo 1 > /sys/bus/i2c/devices/xxxx/select_engine
echo 1 > /sys/class/firmware/lp8501/loading
echo "9d0140ff7e0040007e00a001c000" > /sys/class/firmware/lp8501/data
echo 0 > /sys/class/firmware/lp8501/loading
echo 1 > /sys/bus/i2c/devices/xxxx/run_engine
```

Ex 2) Engine 2 and 3 are used at the same time:

```
echo 2 > /sys/bus/i2c/devices/xxxx/select_engine
sleep 1
echo 1 > /sys/class/firmware/lp8501/loading
echo "9d0140ff7e0040007e00a001c000" > /sys/class/firmware/lp8501/data
echo 0 > /sys/class/firmware/lp8501/loading
sleep 1
echo 3 > /sys/bus/i2c/devices/xxxx/select_engine
sleep 1
echo 1 > /sys/class/firmware/lp8501/loading
echo "9d0340ff7e0040007e00a001c000" > /sys/class/firmware/lp8501/data
```

```
echo 0 > /sys/class/firmware/lp8501/loading
sleep 1
echo 1 > /sys/class/leds/d1/device/run_engine
```

(‘run_engine’ and ‘firmware_cb’)

The sequence of running the program data is common.

But each device has own specific register addresses for commands.

To support this, ‘run_engine’ and ‘firmware_cb’ are configurable in each driver.

run_engine:
Control the selected engine

firmware_cb:
The callback function after loading the firmware is done.

Chip specific commands for loading and updating program memory.

(Predefined pattern data)

Without the firmware interface, LP55xx driver provides another method for loading a LED pattern. That is ‘predefined’ pattern.

A predefined pattern is defined in the platform data and load it(or them) via the sysfs if needed.

To use the predefined pattern concept, ‘patterns’ and ‘num_patterns’ should be configured.

Example of predefined pattern data:

```
/* mode_1: blinking data */
static const u8 mode_1[] = {
              0x40, 0x00, 0x60, 0x00, 0x40, 0xFF, 0x60, 0x00,
              };

/* mode_2: always on */
static const u8 mode_2[] = { 0x40, 0xFF, };

struct lp55xx_predef_pattern board_led_patterns[] = {
      {
              .r = mode_1,
              .size_r = ARRAY_SIZE(mode_1),
      },
      {
              .b = mode_2,
              .size_b = ARRAY_SIZE(mode_2),
      },
}

struct lp55xx_platform_data lp5562_pdata = {
...
      .patterns      = board_led_patterns,
      .num_patterns  = ARRAY_SIZE(board_led_patterns),
};
```

Then, mode_1 and mode_2 can be run via through the sysfs:

```
echo 1 > /sys/bus/i2c/devices/xxxx/led_pattern    # red blinking LED pattern
echo 2 > /sys/bus/i2c/devices/xxxx/led_pattern    # blue LED always on
```

To stop running pattern:

```
echo 0 > /sys/bus/i2c/devices/xxxx/led_pattern
```

CHAPTER

SEVENTEEN

KERNEL DRIVER FOR MELLANOX SYSTEMS LEDS

Provide system LED support for the nex Mellanox systems: “msx6710”, “msx6720”, “msb7700”, “msn2700”, “msx1410”, “msn2410”, “msb7800”, “msn2740”, “msn2100”.

17.1 Description

Driver provides the following LEDs for the systems “msx6710”, “msx6720”, “msb7700”, “msn2700”, “msx1410”, “msn2410”, “msb7800”, “msn2740”:

- mlxcpld:fan1:green
- mlxcpld:fan1:red
- mlxcpld:fan2:green
- mlxcpld:fan2:red
- mlxcpld:fan3:green
- mlxcpld:fan3:red
- mlxcpld:fan4:green
- mlxcpld:fan4:red
- mlxcpld:psu:green
- mlxcpld:psu:red
- mlxcpld:status:green
- mlxcpld:status:red

“status”

- CPLD reg offset: 0x20
- Bits [3:0]

“psu”

- CPLD reg offset: 0x20
- Bits [7:4]

“fan1”

- CPLD reg offset: 0x21

- Bits [3:0]

“fan2”

- CPLD reg offset: 0x21
- Bits [7:4]

“fan3”

- CPLD reg offset: 0x22
- Bits [3:0]

“fan4”

- CPLD reg offset: 0x22
- Bits [7:4]

Color mask for all the above LEDs:

[bit3,bit2,bit1,bit0] or [bit7,bit6,bit5,bit4]:

- [0,0,0,0] = LED OFF
- [0,1,0,1] = Red static ON
- [1,1,0,1] = Green static ON
- [0,1,1,0] = Red blink 3Hz
- [1,1,1,0] = Green blink 3Hz
- [0,1,1,1] = Red blink 6Hz
- [1,1,1,1] = Green blink 6Hz

Driver provides the following LEDs for the system “msn2100”:

- mlxcpld:fan:green
- mlxcpld:fan:red
- mlxcpld:psu1:green
- mlxcpld:psu1:red
- mlxcpld:psu2:green
- mlxcpld:psu2:red
- mlxcpld:status:green
- mlxcpld:status:red
- mlxcpld:uid:blue

“status”

- CPLD reg offset: 0x20
- Bits [3:0]

“fan”

- CPLD reg offset: 0x21

- Bits [3:0]

“psu1”

- CPLD reg offset: 0x23
- Bits [3:0]

“psu2”

- CPLD reg offset: 0x23
- Bits [7:4]

“uid”

- CPLD reg offset: 0x24
- Bits [3:0]

Color mask for all the above LEDs, excepted uid:

[bit3,bit2,bit1,bit0] or [bit7,bit6,bit5,bit4]:

- [0,0,0,0] = LED OFF
- [0,1,0,1] = Red static ON
- [1,1,0,1] = Green static ON
- [0,1,1,0] = Red blink 3Hz
- [1,1,1,0] = Green blink 3Hz
- [0,1,1,1] = Red blink 6Hz
- [1,1,1,1] = Green blink 6Hz

Color mask for uid LED:

[bit3,bit2,bit1,bit0]:

- [0,0,0,0] = LED OFF
- [1,1,0,1] = Blue static ON
- [1,1,1,0] = Blue blink 3Hz
- [1,1,1,1] = Blue blink 6Hz

Driver supports HW blinking at 3Hz and 6Hz frequency (50% duty cycle). For 3Hz duty cylce is about 167 msec, for 6Hz is about 83 msec.

CHAPTER

EIGHTEEN

THE DEVICE FOR MEDIATEK MT6370 RGB LED

18.1 Description

The MT6370 integrates a four-channel RGB LED driver, designed to provide a variety of lighting effect for mobile device applications. The RGB LED devices includes a smart LED string controller and it can drive 3 channels of LEDs with a sink current up to 24mA and a CHG_VIN power good indicator LED with sink current up to 6mA. It provides three operation modes for RGB LEDs: PWM Dimming mode, breath pattern mode, and constant current mode. The device can increase or decrease the brightness of the RGB LED via an I2C interface.

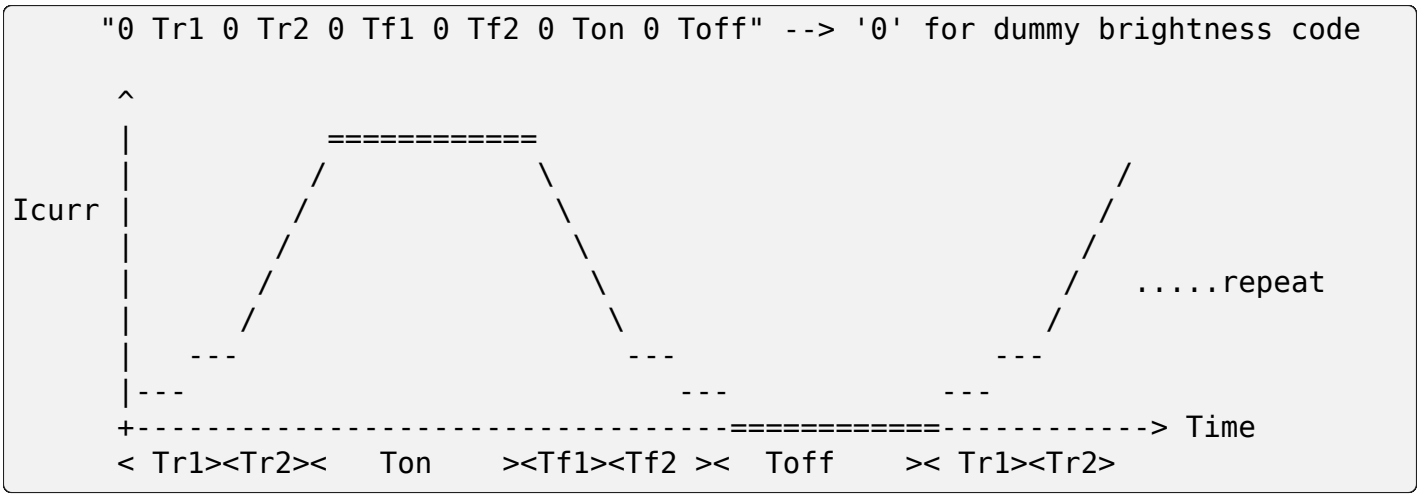
The breath pattern for a channel can be programmed using the “pattern” trigger, using the hw_pattern attribute.

18.2 /sys/class/leds/<led>/hw_pattern

Specify a hardware breath pattern for a MT6370 RGB LED.

The breath pattern is a series of timing pairs, with the hold-time expressed in milliseconds. And the brightness is controlled by ‘/sys/class/leds/<led>/brightness’. The pattern doesn’t include the brightness setting. Hardware pattern only controls the timing for each pattern stage depending on the current brightness setting.

Pattern diagram:



Timing description:

- Tr1: First rising time for 0% - 30% load.
- Tr2: Second rising time for 31% - 100% load.
- Ton: On time for 100% load.
- Tf1: First falling time for 100% - 31% load.
- Tf2: Second falling time for 30% to 0% load.
- Toff: Off time for 0% load.
- Tr1/Tr2/Tf1/Tf2/Ton: 125ms to 3125ms, 200ms per step.
- Toff: 250ms to 6250ms, 400ms per step.

Pattern example:

```
"0 125 0 125 0 125 0 125 0 625 0 1050"
```

This Will configure Tr1/Tr2/Tf1/Tf2 to 125m, Ton to 625ms, and Toff to 1050ms.

CHAPTER

NINETEEN

KERNEL DRIVER FOR SPREADTRUM SC27XX

19.1 /sys/class/leds/<led>/hw_pattern

Specify a hardware pattern for the SC27XX LED. For the SC27XX LED controller, it only supports 4 stages to make a single hardware pattern, which is used to configure the rise time, high time, fall time and low time for the breathing mode.

For the breathing mode, the SC27XX LED only expects one brightness for the high stage. To be compatible with the hardware pattern format, we should set brightness as 0 for rise stage, fall stage and low stage.

- Min stage duration: 125 ms
- Max stage duration: 31875 ms

Since the stage duration step is 125 ms, the duration should be a multiplier of 125, like 125ms, 250ms, 375ms, 500ms ... 31875ms.

Thus the format of the hardware pattern values should be: “0 rise_duration brightness high_duration 0 fall_duration 0 low_duration”.

CHAPTER

TWENTY

KERNEL DRIVER FOR QUALCOMM LPG

20.1 Description

The Qualcomm LPG can be found in a variety of Qualcomm PMICs and consists of a number of PWM channels, a programmable pattern lookup table and a RGB LED current sink.

To facilitate the various use cases, the LPG channels can be exposed as individual LEDs, grouped together as RGB LEDs or otherwise be accessed as PWM channels. The output of each PWM channel is routed to other hardware blocks, such as the RGB current sink, GPIO pins etc.

The each PWM channel can operate with a period between 27us and 384 seconds and has a 9 bit resolution of the duty cycle.

In order to provide support for status notifications with the CPU subsystem in deeper idle states the LPG provides pattern support. This consists of a shared lookup table of brightness values and per channel properties to select the range within the table to use, the rate and if the pattern should repeat.

The pattern for a channel can be programmed using the “pattern” trigger, using the hw_pattern attribute.

20.2 /sys/class/leds/<led>/hw_pattern

Specify a hardware pattern for a Qualcomm LPG LED.

The pattern is a series of brightness and hold-time pairs, with the hold-time expressed in milliseconds. The hold time is a property of the pattern and must therefore be identical for each element in the pattern (except for the pauses described below). As the LPG hardware is not able to perform the linear transitions expected by the leds-trigger-pattern format, each entry in the pattern must be followed a zero-length entry of the same brightness.

Simple pattern:

```
"255 500 255 0 0 500 0 0"

    ^
    |
255 +----+    +----+
    |    |    |    |      ...
  0 |    +----+    +----
```

```
    +---------------------->
    0    5   10   15     time (100ms)
```

The LPG supports specifying a longer hold-time for the first and last element in the pattern, the so called “low pause” and “high pause”.

Low-pause pattern:

```
"255 1000 255 0 0 500 0 0 255 500 255 0 0 500 0 0"

    ^
    |
255 +--------+    +----+    +----+    +--------+
    |        |    |    |    |    |    |        |      ...
  0 |        +----+    +----+    +----+        +----
    +------------------------------>
    0    5   10   15  20   25   time (100ms)
```

Similarly, the last entry can be stretched by using a higher hold-time on the last entry.

In order to save space in the shared lookup table the LPG supports “ping-pong” mode, in which case each run through the pattern is performed by first running the pattern forward, then backwards. This mode is automatically used by the driver when the given pattern is a palindrome. In this case the “high pause” denotes the wait time before the pattern is run in reverse and as such the specified hold-time of the middle item in the pattern is allowed to have a different hold-time.