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.

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.

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.

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.

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.

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.

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.