Pulse Width Modulation (PWM) interface

This provides an overview about the Linux PWM interface

PWMs are commonly used for controlling LEDs, fans or vibrators in cell phones. PWMs with a fixed purpose have no need implementing the Linux PWM API (although they could). However, PWMs are often found as discrete devices on SoCs which have no fixed purpose. It's up to the board designer to connect them to LEDs or fans. To provide this kind of flexibility the generic PWM API exists.

Identifying PWMs

Users of the legacy PWM API use unique IDs to refer to PWM devices.

Instead of referring to a PWM device via its unique ID, board setup code should instead register a static mapping that can be used to match PWM consumers to providers, as given in the following example:

```
static struct pwm_lookup board_pwm_lookup[] = {
        PWM_LOOKUP("tegra-pwm", 0, "pwm-backlight", NULL,
                   50000, PWM_POLARITY_NORMAL),
};

static void __init board_init(void)
{
        ...
        pwm_add_table(board_pwm_lookup, ARRAY_SIZE(board_pwm_lookup));
        ...
}
```

Using PWMs

Legacy users can request a PWM device using pwm_request() and free it after usage with pwm_free().

New users should use the pwm_get() function and pass to it the consumer device or a consumer name. pwm_put() is used to free the PWM device. Managed variants of the getter, devm_pwm_get(), devm_of_pwm_get(), devm_fwnode_pwm_get(), also exist.

After being requested, a PWM has to be configured using:

```
int pwm_apply_state(struct pwm_device *pwm, struct pwm_state *state);
```

This API controls both the PWM period/duty_cycle config and the enable/disable state. There is also a usage_power setting: If set, the PWM driver is only required to maintain the power output but has more freedom regarding signal form. If supported by the driver, the signal can be optimized, for example to improve EMI by phase shifting the individual channels of a chip.

The pwm_config(), pwm_enable() and pwm_disable() functions are just wrappers around pwm_apply_state() and should not be used if the user wants to change several parameter at once. For example, if you see pwm_config() and pwm_{enable,disable}() calls in the same function, this probably means you should switch to pwm_apply_state().

The PWM user API also allows one to query the PWM state that was passed to the last invocation of pwm_apply_state() using pwm_get_state(). Note this is different to what the driver has actually implemented if the request cannot be satisfied exactly with the hardware in use. There is currently no way for consumers to get the actually implemented settings.

In addition to the PWM state, the PWM API also exposes PWM arguments, which are the reference PWM config one should use on this PWM. PWM arguments are usually platform-specific and allows the PWM user to only care about dutycycle relatively to the full period (like, duty = 50% of the period). struct pwm_args contains 2 fields (period and polarity) and should be used to set the initial PWM config (usually done in the probe function of the PWM user). PWM arguments are retrieved with pwm_get_args().

All consumers should really be reconfiguring the PWM upon resume as appropriate. This is the only way to ensure that everything is resumed in the proper order.

Using PWMs with the sysfs interface

If CONFIG_SYSFS is enabled in your kernel configuration a simple sysfs interface is provided to use the PWMs from userspace. It is exposed at /sys/class/pwm/. Each probed PWM controller/chip will be exported as pwmchipN, where N is the base of the PWM chip. Inside the directory you will find:

npwm
: The number of PWM channels this chip supports (read-only).

export
: Exports a PWM channel for use with sysfs (write-only).

unexport
: Unexports a PWM channel from sysfs (write-only).

The PWM channels are numbered using a per-chip index from 0 to npwm-1.

When a PWM channel is exported a pwmX directory will be created in the pwmchipN directory it is associated with, where X is the number of the channel that was exported. The following properties will then be available:

period

The total period of the PWM signal (read/write). Value is in nanoseconds and is the sum of the active and inactive time of the PWM.

duty_cycle

The active time of the PWM signal (read/write). Value is in nanoseconds and must be less than the period.

polarity

Changes the polarity of the PWM signal (read/write). Writes to this property only work if the PWM chip supports changing the polarity. The polarity can only be changed if the PWM is not enabled. Value is the string "normal" or "inversed".

enable

Enable/disable the PWM signal (read/write).

- 0 - disabled
- 1 - enabled

Implementing a PWM driver

Currently there are two ways to implement pwm drivers. Traditionally there only has been the barebone API meaning that each driver has to implement the pwm_*() functions itself. This means that it's impossible to have multiple PWM drivers in the system. For this reason it's mandatory for new drivers to use the generic PWM framework.

A new PWM controller/chip can be added using pwmchip_add() and removed again with pwmchip_remove(). pwmchip_add() takes a filled in struct pwm_chip as argument which provides a description of the PWM chip, the number of PWM devices provided by the chip and the chip-specific implementation of the supported PWM operations to the framework.

When implementing polarity support in a PWM driver, make sure to respect the signal conventions in the PWM framework. By definition, normal polarity characterizes a signal starts high for the duration of the duty cycle and goes low for the remainder of the period. Conversely, a signal with inversed polarity starts low for the duration of the duty cycle and goes high for the remainder of the period.

Drivers are encouraged to implement ->apply() instead of the legacy ->enable(), ->disable() and ->config() methods. Doing that should provide atomicity in the PWM config workflow, which is required when the PWM controls a critical device (like a regulator).

The implementation of ->get_state() (a method used to retrieve initial PWM state) is also encouraged for the same reason: letting the PWM user know about the current PWM state would allow him to avoid glitches.

Drivers should not implement any power management. In other words, consumers should implement it as described in the "Using PWMs" section.

Locking

The PWM core list manipulations are protected by a mutex, so pwm_request() and pwm_free() may not be called from an atomic context. Currently the PWM core does not enforce any locking to pwm_enable(), pwm_disable() and pwm_config(), so the calling context is currently driver specific. This is an issue derived from the former barebone API and should be fixed soon.

Helpers

Currently a PWM can only be configured with period_ns and duty_ns. For several use cases freq_hz and duty_percent might be better. Instead of calculating this in your driver please consider adding appropriate helpers to the framework.