Adding Runtime Power Management Capabilities to Device Drivers
About me

Shreeya Patel

➢ Associate Software Engineer at Collabora
➢ Hardware Enablement related projects
  • RK3588 upstreaming work
  • Steam Deck Kernel Development
➢ KernelCI Regression tracking
➢ Outreachy 2020 – Sound Open Firmware project
➢ Started my Linux Kernel Development journey with IIO subsystem
Agenda

- Power Management in Linux Kernel and Runtime PM
- Subsystem-level RPM
- Helper functions
- LTRF216A light sensor driver
- Adding RPM support to the LTRF216A driver
- Some issues and it’s solution
Power Management in Linux Kernel

Introduction to Kernel Power Management - by Kevin Hilman
Why was Runtime PM introduced?

➢ Devices were bored of being idle.
➢ System sleep suspends all the devices together.
➢ A mechanism to put individual devices to sleep was needed.
➢ Overall less system power consumption.
Relationship between PM Core and Runtime PM

➢ PM core manages the system-wide PM in Linux Kernel.
  • Synchronizes Runtime PM and system-wide PM.
  • kernel/power/main.c

➢ PM workqueue → pm_wq
  • Devices put their work items like suspend, resume, etc in the pm_wq.

```c
struct dev_pm_ops {
  ...
  int (*runtime_suspend)(struct device *dev);
  int (*runtime_resume)(struct device *dev);
  int (*runtime_idle)(struct device *dev);
  ...
};
```
Relationship between PM Core and Runtime PM

➢ A set of Runtime PM fields are provided by the `power` member of the device structure.

```c
struct device {
    struct dev_pm_info power;
}
```

➢ Helper functions defined in `drivers/base/power/runtime.c` carries out Runtime PM operations with the help of PM core.
Subsystem-level RPM

Callback functions

Subsystem level?

Yes

dev->pm_domain

dev->class and dev->class->pm

dev->type and dev->type->pm

dev->bus and dev->bus->pm

No

dev->driver->pm
Subsystem-level Runtime PM

```c
static struct bus_type ac97_bus_type = {
    ....
    .pm             = &ac97_pm,
    .probe          = ac97_bus_probe,
    .remove         = ac97_bus_remove,
};

static const struct dev_pm_ops ac97_pm = {
    ....
    SET_RUNTIME_PM_OPS(
        ac97_pm_runtime_suspend,
        ac97_pm_runtime_resume,
        NULL)
};

static int ac97_pm_runtime_suspend(struct device *dev)
{
    struct ac97_codec_device *codec = to_ac97_device(dev);
    int ret = pm_generic_runtime_suspend(dev); // calls runtime_suspend for the device

    if (ret == 0 && dev->driver) {
        // disables the codec clock
        if (pm_runtime_is_irq_safe(dev))
            clk_disable(codec->clk);
        else
            clk_disable_unprepare(codec->clk);
    }
}
```

sound/ac97/bus.c
Parent-Child Relationship

```c
pm_suspend_ignore_children()
power.ignore_children = true
```
Initial runtime PM status of all the devices is ‘suspended’

- That might not be it’s true physical state.

`devm_pm_runtime_enable()` automatically handles the disabling of the Runtime PM.

Callbacks can be executed in atomic context with interrupts disabled using `pm_runtime_irq_safe()`

- It will set the `power.irq_safe` flag
- It will also not let the parent device being runtime-suspended.

### Helper functions - Initialization and Enabling

- `pm_runtime_set_active()`
- `pm_runtime_enable()`
- `devm_pm_runtime_enable()`
- `pm_runtime_active()`
- `pm_runtime_init()`
- `pm_runtime_irq_safe()`
- `pm_runtime_is_irq_safe()`
Helper functions - Before and After I/O

➢ There can be multiple users of the device.
➢ Reference counting helps to keep a track of the device usage requests.
  • `_get_()` → power.usage_count += 1
  • `_put_()` → power.usage_count -= 1
➢ `pm_runtime_get()` increments the usage_count even on returning an error.
➢ Want to wait for the power state transition to complete before executing the next code block?
  • Use `_sync_()` Runtime PM helper functions.
Imagine you have a device which does I/O operation every 1 sec.

A lot of time and energy is wasted.

Add an inactivity period with the help of autosuspend delay.

Deffering the suspend until inactivity period has elapsed.
Helper functions - Autosuspend

➢ Inactivity period is set using the
  \texttt{pm\_runtime\_set\_autosuspend\_delay(dev, delay)}

➢ Drivers need to keep a record of their last I/O operation time.
  
  • \texttt{pm\_runtime\_mark\_last\_busy()} updates the power.last_busy field.

➢ Inactivity period can also be handled through sysfs.
  
  • \texttt{/sys/devices/.../power/autosuspend_delay_ms}

➢ Race Conditions?

\begin{itemize}
  \item \texttt{pm\_runtime\_use\_autosuspend()}
  \item \texttt{pm\_runtime\_autosuspend()}
  \item \texttt{pm\_runtime\_put\_autosuspend()}
  \item \texttt{pm\_runtime\_mark\_last\_busy()}
  \item \texttt{pm\_runtime\_set\_autosuspend\_delay()}
  \item \texttt{pm\_runtime\_autosuspend\_expiration()}
\end{itemize}
Helper functions - Removal and some more...

➢ All the pending PM operations are completed/cancelled while disabling the device.
➢ pm_runtime_remove() will disable the runtime PM and unregister device from Runtime PM framework.
➢ pm_runtime_forbid()
  
  ```
  echo "on" > /sys/devices/.../power/control
  ```
➢ pm_runtime_allow()
  
  ```
  echo "auto" > /sys/devices/.../power/control
  ```
➢ pm_runtime_no_callbacks() will delete the runtime PM attributes.

https://docs.kernel.org/power/runtime_pm.html
LTRF216A light sensor driver

➢ I2C Ambient Light Sensor (ALS)

➢ ALS DATA is the digital representation of ambient light level stored in the registers regardless of light sources.
   • Raw value

➢ Lux_calc is a meaningful representation of the actual light intensity in a specific environment.
   • Processed value

➢ Supports multiple integration time.
   • 25ms, 50ms, 100ms, 200ms and 400ms.
LTRF216A light sensor driver

static int ltrf216a_probe(struct i2c_client *client)
{
    ... 
    indio_dev = devm_iio_device_alloc(&client->dev, sizeof(*data));

    data = iio_priv(indio_dev);
    data->regmap = devm_regmap_init_i2c(client, &ltrf216a_regmap_config);

    i2c_set_clientdata(client, indio_dev);

    indio_dev->info = &ltrf216a_info;
    indio_dev->name = "ltrf216a";
    indio_dev->channels = ltrf216a_channels;
    indio_dev->num_channels = ARRAY_SIZE(ltrf216a_channels);

    /* reset sensor, chip fails to respond to this, so ignore any errors */
    ltrf216a_reset(indio_dev);

    ret = ltrf216a_enable(indio_dev);

    data->int_time = 100000;
    data->int_time_fac = 100;
    data->als_gain_fac = 3;

    return devm_iio_device_register(&client->dev, indio_dev);
}
static int ltrf216a_read_raw(struct iio_dev *indio_dev,...)
{
    struct ltrf216a_data *data = iio_priv(indio_dev);
    int ret;

    switch (mask) {
    case IIO_CHAN_INFO_RAW:
        mutex_lock(&data->lock);
        ret = ltrf216a_read_data(data, LTRF216A_ALS_DATA_0);
        mutex_unlock(&data->lock);
        *val = ret;
        return IIO_VAL_INT;
    case IIO_CHAN_INFO_PROCESSED:
        mutex_lock(&data->lock);
        ret = ltrf216a_get_lux(data);
        mutex_unlock(&data->lock);
        *val = ret;
        return IIO_VAL_INT;
    case IIO_CHAN_INFO_INT_TIME:
        mutex_lock(&data->lock);
        ret = ltrf216a_get_int_time(data, val, val2);
        mutex_unlock(&data->lock);
        return ret;
    default:
        return -EINVAL;
    }
}

static int ltrf216a_write_raw(struct iio_dev *indio_dev,...)
{
    ...
    switch (mask) {
    case IIO_CHAN_INFO_INT_TIME:
        mutex_lock(&data->lock);
        ret = ltrf216a_set_int_time(data, val2);
        mutex_unlock(&data->lock);
        return ret;
    ...
}

static int ltrf216a_read_available(struct iio_dev *indio_dev,...)
{
    switch (mask) {
    case IIO_CHAN_INFO_INT_TIME:
        *length = ARRAY_SIZE(ltrf216a_int_time_available)*2;
        *vals = (const int *)ltrf216a_int_time_available;
        *type = IIO_VAL_INT_PLUS_MICRO;
        return IIO_AVAIL_LIST;
    ...
}
static int ltrf216a_probe(struct i2c_client *client)
{
    ...
    indio_dev = devm_iio_device_alloc(&client->dev, sizeof(*data));

    data = iio_priv(indio_dev);
    data->regmap = devm_regmap_init_i2c(client, &ltrf216a_regmap_config);
    i2c_set_clientdata(client, indio_dev);
    indio_dev->info = &ltrf216a_info;
    indio_dev->name = "ltrf216a";
    indio_dev->channels = ltrf216a_channels;
    indio_dev->num_channels = ARRAY_SIZE(ltrf216a_channels);

    /* reset sensor, chip fails to respond to this, so ignore any errors */
    ltrf216a_reset(indio_dev);

    ret = ltrf216a_enable(indio_dev);

    data->int_time = 100000;
    data->int_time_fac = 100;
    data->als_gain_fac = 3;

    return devm_iio_device_register(&client->dev, indio_dev);
}
LTRF216A light sensor driver

```
(B+)@steamdeck ~ # ls -l /sys/bus/iio/devices/iio\:device0/
total 0
-rw-r--r-- 1 root root 4096 Jun 21 03:24 in_illuminance_input
-rw-r--r-- 1 root root 4096 Jun 21 03:24 in_illuminance_integration_time
-r-r--r-- 1 root root 4096 Jun 21 03:24 in_illuminance_integration_time_available
-rw-r--r-- 1 root root 4096 Jun 21 03:24 in_illuminance_raw
-r--r--r-- 1 root root 4096 Jun 21 03:24 name
-rwrxr-xr-x 2 root root 0 Jun 21 03:24 power
-rwrxrwxrwx 1 root root 0 Jun 21 03:24 subsystem -> ../../../bus/iio
-rw-r--r-- 1 root root 4096 Jun 21 03:24 uevent

(B+)@steamdeck ~ # cat /sys/bus/iio/devices/iio\:device0/in_illuminance_integration_time
0.100000

(B+)@steamdeck ~ # cat /sys/bus/iio/devices/iio\:device0/in_illuminance_integration_time_available
0.400000 0.200000 0.100000 0.050000 0.025000

(B+)@steamdeck ~ # cat /sys/bus/iio/devices/iio\:device0/in_illuminance_raw
90

(B+)@steamdeck ~ # cat /sys/bus/iio/devices/iio\:device0/in_illuminance_input
13.350000000
```

Collabora Open First
```
static int ltrf216a_probe(struct i2c_client *client) {
    ...
    indio_dev->info = &ltrf216a_info;
    indio_dev->name = "ltrf216a";
    indio_dev->channels = ltrf216a_channels;
    indio_dev->num_channels = ARRAY_SIZE(ltrf216a_channels);
    indio_dev->modes = INDIO_DIRECT_MODE;

    ret = pm_runtime_set_active(&client->dev);
    if (ret)
        return ret;
    ltrf216a_reset(indio_dev);

    ret = ltrf216a_enable(indio_dev);
    if (ret)
        return ret;

    ret = devm_add_action_or_reset(&client->dev, ltrf216a_cleanup,
                                   indio_dev);
    ...
```
static int ltrf216a_read_raw(struct iio_dev *indio_dev,...) {
    switch (mask) {
    case IIO_CHAN_INFO_RAW:
        ret = ltrf216a_set_power_state(data, true);
        if (ret)
            return ret;
        mutex_lock(&data->lock);
        ret = ltrf216a_read_data(data, LTRF216A_ALS_DATA_0);
        mutex_unlock(&data->lock);
        ltrf216a_set_power_state(data, false);
        if (ret < 0)
            return ret;
        *val = ret;
        return IIO_VAL_INT;
    case IIO_CHAN_INFO_PROCESSED:
        mutex_lock(&data->lock);
        ret = ltrf216a_get_lux(data);
        mutex_unlock(&data->lock);
        if (ret < 0)
            return ret;
        *val = ret;
        return IIO_VAL_INT;
    case IIO_CHAN_INFO_INT_TIME:
        mutex_lock(&data->lock);
        ret = ltrf216a_get_int_time(data, val, val2);
        mutex_unlock(&data->lock);
        ....

static int ltrf216a_get_lux(struct ltrf216a_data *data) {
    ...
    ret = ltrf216a_set_power_state(data, true);
    if (ret)
        return ret;
    greendata = ltrf216a_read_data(data, LTRF216A_ALS_DATA_0);
    if (greendata < 0)
        return greendata;
    ltrf216a_set_power_state(data, false);
    lux = greendata * 45 * LTRF216A_WIN_FAC * 100;
    div = data->als_gain_fac * data->int_time_fac * 100;
    return div_u64(lux, div);
}
int ltrf216a_set_power_state(struct ltrf216a_data *data, bool on)
{
    struct device *dev = &data->client->dev;
    int ret = 0;

    if (on) {
        ret = pm_runtime_resume_and_get(dev);
        if (ret) {
            dev_err(dev, "failed to resume runtime PM: %d\n", ret);
            return ret;
        }
    } else {
        pm_runtime_mark_last_busy(dev);
        pm_runtime_put_autosuspend(dev);
    }

    return ret;
}
Adding RPM support to LTRF216A light sensor driver

```c
static int ltrf216a_runtime_suspend(struct device *dev)
{
    ...
    ret = ltrf216a_disable(indio_dev);
    if (ret)
        return ret;
    ...
}

static int ltrf216a_runtime_resume(struct device *dev)
{
    ...
    ret = ltrf216a_enable(indio_dev);
    if (ret)
        goto cache_only;
    return 0;
    ...
}
```

```c
static DEFINE_RUNTIME_DEV_PM_OPS(ltrf216a_pm_ops, ltrf216a_runtime_suspend, ltrf216a_runtime_resume, NULL);

static struct i2c_driver ltrf216a_driver = {
    .driver = {
        .name = "ltrf216a",
        .pm = pm_ptr(&ltrf216a_pm_ops),
        .of_match_table = ltrf216a_of_match,
    },
    .probe_new = ltrf216a_probe,
    .id_table = ltrf216a_id,
};

module_i2c_driver(ltrf216a_driver);
```

```
(B+)@root@steamdeck ~# ls -l /sys/bus/iio/devices/iio\:device0/power/
total 0
-rw-r--r-- 1 root root 4096 Jun 21 03:26 autosuspend_delay_ms
-rw-r--r-- 1 root root 4096 Jun 21 03:26 control
-rw-r--r-- 1 root root 4096 Jun 21 03:25 runtime_active_time
-rw-r--r-- 1 root root 4096 Jun 21 03:25 runtime_status
-rw-r--r-- 1 root root 4096 Jun 21 03:25 runtime_suspended_time
(B+)@root@steamdeck ~#
```
Powertop

The battery reports a discharge rate of 4.76 W
The energy consumed was 93.8 J

<table>
<thead>
<tr>
<th>Usage</th>
<th>Device name</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.3%</td>
<td>CPU misc</td>
</tr>
<tr>
<td>19.3%</td>
<td>CPU core</td>
</tr>
<tr>
<td>100.0%</td>
<td>USB device: xHCI Host Controller</td>
</tr>
<tr>
<td>100.0%</td>
<td>USB device: Steam Deck Controller (Valve Software)</td>
</tr>
<tr>
<td>5.5%</td>
<td>Display backlight</td>
</tr>
<tr>
<td>0.0%</td>
<td>USB device: xHCI Host Controller</td>
</tr>
<tr>
<td>100.0%</td>
<td>PCI Device: Advanced Micro Devices, Inc. [AMD] VanGogh Data Fabric; Function 1</td>
</tr>
<tr>
<td>100.0%</td>
<td>PCI Device: Advanced Micro Devices, Inc. [AMD] Renoir PCIe Dummy Host Bridge</td>
</tr>
<tr>
<td>100.0%</td>
<td>PCI Device: Advanced Micro Devices, Inc. [AMD] Renoir PCIe Dummy Host Bridge</td>
</tr>
<tr>
<td>100.0%</td>
<td>PCI Device: Advanced Micro Devices, Inc. [AMD] VanGogh PCIe GPP Bridge</td>
</tr>
<tr>
<td>100.0%</td>
<td>PCI Device: Phison Electronics Corporation P55013 E13 NVMe Controller</td>
</tr>
<tr>
<td>100.0%</td>
<td>PCI Device: Advanced Micro Devices, Inc. [AMD] VanGogh Data Fabric; Function 5</td>
</tr>
<tr>
<td>100.0%</td>
<td>PCI Device: 02 Micro, Inc. SD/MMC Card Reader Controller</td>
</tr>
<tr>
<td>100.0%</td>
<td>Radio device: rtw_8822ce</td>
</tr>
<tr>
<td>100.0%</td>
<td>PCI Device: Advanced Micro Devices, Inc. [AMD] VanGogh PCIe GPP Bridge</td>
</tr>
<tr>
<td>100.0%</td>
<td>PCI Device: Advanced Micro Devices, Inc. [AMD] VanGogh USB2</td>
</tr>
</tbody>
</table>

(B+)(root@steamdeck ~)# ls -l /sys/bus/iio/devices/iio\:device0/power/*time
-r--r--r-- 1 root root 4096 Jun 21 03:25 /sys/bus/iio/devices/iio:device0/power/runtime_active_time
-r--r--r-- 1 root root 4096 Jun 21 03:25 /sys/bus/iio/devices/iio:device0/power/runtime_suspended_time
What issues you might face?

➢ Loss of data
  • Registers will be reset to their initial values after resuming the device.
  • Drivers need to implement a way to restore the data before suspending the device.

➢ Not all devices work independently
  • The Generic Power Domain (genpd) framework.
  • Genpd facilitates coordinated power management at both the device level and subsystem level.
Special Thanks

Dmitrii Osipenko

Sebastian Reichel
Thank you
We are hiring
col.la/careers