Runtime Power Management

Linux Device Drivers

Bill Gatliff
bgat@billgatliff.com

Freelance Embedded Systems Developer

Run-Time Power Management

Turn off unused devices:

- No interfaces opened
- No interrupts expected/ignored
- No active control loops, etc.

Goal is reduced power consumption:

- ... with minimal end-user experience impact

Run-Time Power Management

Can't we do this already?

- Initializers from `open()` methods
- Enabling/disabling interrupts
- Low-power modes of external components
- Peripheral clock management
- Voltage regulators

So, yes. Well, kind of...

Run-Time Power Management

Shortcomings of brute-force approaches:

- Device state is opaque to Device Model
- Races with platform suspend/resume
- What about devices with no user interfaces?
- How to coordinate states between devices?

A new framework was needed!

Linux Runtime PM

Device Model awareness:

- Reference count determines device state
- Framework callbacks implement state changes

Advantages:

- Kernel is aware of device state
- Framework won’t race during platform suspend
- Parent-child, bus relationships are observed

Linux Runtime PM

Disadvantages:

- More subtle than brute-force alternatives

“Subtle” behavior:

- Triggered by`device_get()`, `device_put()`
- Fundamentally different from system sleep!

struct dev_pm_ops

```
struct dev_pm_ops {
  ...
  int (*runtime_suspend)(struct device *dev);
  int (*runtime_resume )(struct device *dev);
  int (*runtime_idle   )(struct device *dev);
  ...
};
```

struct dev_pm_ops

```
struct device_driver {
  ...
  int (*suspend)(struct device *dev,
                 pm_message_t state);
  int (*resume )(struct device *dev);
  ...
  const struct dev_pm_ops *pm;
  ...
};
```

`runtime_suspend()`

Place the device into “suspended” state:

- Disable interrupts
- Save volatile data
- Place device into low-power mode
- Disable regulators, etc.

Invoked only when device isn’t in use:

- But device might not subsequently power down!
- (Why?)

runtime_resume()

Return from “suspended” state:

- NOT the same as making device “active”!
- Restore volatile data to device

`runtime_idle()`

Check to see if device is truly “idle”:

- Not generally useful, except for bus handlers
- Leave undefined for best results

bma250-mib.c

```
static int bma_runtime_suspend(struct device *dev)
{
        struct i2c_client *client = to_i2c_client(dev);
        struct bma250 *bma = i2c_get_clientdata(client);
        int ret;

        ret = mutex_lock_interruptible(&bma->mutex);
        if (ret)
                goto err;

        /* drive the chip into its SUSPEND state, so that it is truly
         * quiet even if its power sources don't go away (which might
         * be the case if that regulator is powering other devices) */
        ret = __bma_reg_write_POWER(bma, BMA250_REG_POWER__SUSPEND);
```

bma250-mib.c

```
        /* TODO: do we need a disable_irq() here? */

        /* tell Linux we don't need our regulators now; this doesn't
         * guarantee that our regulators will ACTUALLY turn off! */
        __bma_disable_regulators(bma);

        mutex_unlock(&bma->mutex);
err:
        return (ret < 0) ? ret : 0;
}
```

bma250-mib.c

```
static int bma_runtime_resume(struct device *dev)
{
        struct i2c_client *client = to_i2c_client(dev);
        struct bma250 *bma = i2c_get_clientdata(client);
        int ret;

        ret = mutex_lock_interruptible(&bma->mutex);
        if (ret < 0)
                return ret;

        __bma_enable_regulators(bma);

        ret = __bma_reset(bma);
        if (ret < 0)
                goto err_reset;
```

bma250-mib.c

```
	/* pre-populate caches with hardware defaults for RANGE and
	 * BANDWIDTH registers; probe() and userspace will update
	 * these later as they so choose, and we will subsequently
	 * preserve them across future suspend/resume operations */
	ret = __bma_reg_read_RANGE(bma);
	if (ret >= 0)
		ret = __bma_reg_read_BANDWIDTH(bma);
	if (ret < 0)
		goto err_init_caches;
```

bma250-mib.c

```
err_init_caches:
err_reset:

        /* place chip into SUSPEND mode; other entry points will push
         * the chip to higher functional modes as needed */
        ret = __bma_reg_write_POWER(bma, BMA250_REG_POWER__SUSPEND);
        mutex_unlock(&bma->mutex);

        pm_runtime_mark_last_busy(dev);

        return (ret < 0) ? ret : 0;
}
```

System Suspend and Resume

Runtime suspend:

- “You aren’t in use, disable yourself”

System suspend:

- “Disable yourself. Just do it.”
- Device is (usually) runtime-pm resumed first

Helper Functions

Common use cases:

- Device probing and removal
- Awakening device upon interface/attribute activity

pm_runtime_get_sync()

Increments use count:

- Will `runtime_resume()` device as necessary
- Blocks until `runtime_resume()` returns

```
int pm_runtime_get_sync(struct device *dev);
```

pm_runtime_put_sync()

Decrements use count:

- Invokes `runtime_suspend()` if count drops to zero
- Blocks until `runtime_suspend()` returns

```
int pm_runtime_put_sync(struct device *dev);
```

pm_runtime_put_autosuspend()

Decrements use count, but:

- Returns immediately
- The `runtime_suspend()` will run later upon timeout
- Helps avoid supend-resume-suspend cycle for sporadic device use
- Watch out for synchronization issues!

```
int pm_runtime_put_autosuspend(struct device *dev);
```

pm_runtime_mark_last_busy()

Restarts autosuspend timer:

- Indicates activity at the device
- Some kinds of “activity” might not merit full idle afterwards
- (I think the API got this backwards)

```
void pm_runtime_mark_last_busy(struct device *dev);
```

bma250-mib.c

```
static ssize_t bma_show_evdev_delay_ms(struct device *dev,
                                       struct device_attribute *attr,
                                       char *buf)
{
        struct bma250 *bma = dev_get_drvdata(dev);
        return sprintf(buf, "%d\n", bma->evdev_delay_ms);
}
```

bma250-mib.c

```
static ssize_t bma_store_evdev_delay_ms(struct device *dev,
                                        struct device_attribute *attr,
                                        const char *buf, size_t len)
{
        struct bma250 *bma = dev_get_drvdata(dev);
        int ret;
        unsigned long evdev_delay_ms;

        ret = strict_strtoul(buf, 10, &evdev_delay_ms);
        if (ret)
                return ret;
        if (!evdev_delay_ms)
                return -EINVAL;
```

bma250-mib.c

```
        pm_runtime_get_sync(dev);

        ret = mutex_lock_interruptible(&bma->mutex);
        if (ret < 0)
                goto done;

        bma->evdev_delay_ms = evdev_delay_ms;

        ret = __bma_configure_evdev(bma);
        mutex_unlock(&bma->mutex);

done:
        pm_runtime_mark_last_busy(dev);
        pm_runtime_put_autosuspend(dev);
        return (ret < 0) ? ret : len;
}
```

Initialization and Shutdown

`pm_runtime_init()`

- Initializes PM fields in device structure
- Default device state is “inactive”
- MUST invoke before enabling runtime PM

```
void pm_runtime_init(struct device *dev);
```

Initialization and Shutdown

`pm_runtime_enable()`

- Enables runtime power management logic

```
void pm_runtime_enable(struct device *dev);
```

Initialization and Shutdown

`pm_runtime_disable()`

- Stops runtime power management logic
- Can also be used to “pause” runtime-pm

```
void pm_runtime_disable(struct device *dev);
```

pm_runtime_set_active()

Sets initial state to "active":

- Default is "inactive"
- Used to get initialization states correct
- Invoke before `pm_runtime_enable()`

```
int pm_runtime_set_active(struct device *dev);
```

bma250-mib.c

```
static int bma_probe(struct i2c_client *client,
                        const struct i2c_device_id *id)
{
        struct bma250 *bma;
        struct bma25x_platform_data *pdata = client->dev.platform_data;
        int ret, chip_id, version;

        bma = kzalloc(sizeof(*bma), GFP_KERNEL);
        if (!bma)
                return -ENOMEM;

        i2c_set_clientdata(client, bma);
        bma->client = client;
        mutex_init(&bma->mutex);
        bma->evdev_delay_ms = (default_evdev_delay_ms > 0) ?
                default_evdev_delay_ms : 1000;
```

bma250-mib.c

```
        INIT_DELAYED_WORK(&bma->input_worker, bma_input_worker);

        ret = bma_get_regulators(bma);
        if (ret)
                goto err_get_regulators;

        /* awaken the chip to its SUSPEND mode */
        pm_runtime_enable(&bma->client->dev);
        ret = pm_runtime_resume(&bma->client->dev);
        if (ret)
                goto err_runtime_resume;
```

bma250-mib.c

```
        ret = mutex_lock_interruptible(&bma->mutex);
        if (ret)
                goto err_lock_mutex;

        /* NOTE: we don’t need to bring the chip out of its SUSPEND
         * mode in order to merely READ register values; writes
         * require us to push the chip into ACTIVE mode */

        /* read chip IDs, to make sure the chip is there */
        ret = __bma_reg_read_CHIP_ID(bma);
        if (ret < 0)
                goto err_read_chip_id;
        chip_id = ret;

        ret = __bma_reg_read_VERSION(bma);
        if (ret < 0)
                goto err_read_version;
        version = ret;

        mutex_unlock(&bma->mutex);
```

bma250-mib.c

```
        pm_runtime_set_autosuspend_delay(&bma->client->dev,
                                         default_autosuspend_ms);
        pm_runtime_mark_last_busy(&bma->client->dev);
        pm_runtime_use_autosuspend(&bma->client->dev);

        return 0;
```

pm_runtime_resume()

Briefly awakens the device:

- Most useful during `probe()` and similar situations
- Does not increment usage count
- Must call `pm_runtime_enable()` first!
- Device Model framework will suspend device later

```
int pm_runtime_resume(struct device *dev);
```

Userspace Control

Sysfs hooks allow some control:

- Locking device into “resumed” state
- Adjusting autosuspend timer

```
$ echo "on" > /sys/devices/.../power/control
$ echo "auto" > /sys/devices/.../power/control
```

Additional Recommendations

In general:

- Prefer device attributes a.k.a. “sysfs attributes” to `char` interfaces
- Prefer `struct cdev` to `struct miscdevice`
- Embrace the Device Model API fully!

Device attributes:

- Framework brings you the `struct device` pointer
- Standard APIs bring you device data

Additional Recommendations

```
static ssize_t
mpu_show_AUX_VDDIO(struct device *dev,
                   struct device_attribute *attr,
                   char *buf)
{
    struct mpu *mpu = dev_get_drvdata(dev);
    pm_runtime_get_sync(dev);
    ...
    pm_runtime_put_autosuspend(dev);
    return ret;
}
```

Additional Recommendations

`struct cdev` vs. `struct miscdevice`

- File operations provide `cdev` reference, but not `miscdevice`
- Careful structures and `container_of()` bring you device data

Additional Recommendations

```
static int foo_open(struct inode *inode,
                    struct file *file)
{
    struct cdev *c = inode->i_cdev;
    struct foo *foo
        = container_of(c, struct foo, cdev);

    file->private_data = foo;
    ...
}
```

Additional Recommendations

Embrace the Device Model:

- Nearly everything reacts to `device_get()`
- Brute-forcing things limits your benefits, creates work

Additional Recommendations

```c
int mpu_aux_xfer(struct i2c_adapter *adap,
                 struct i2c_msg *msg, int num)
{
    struct mpu *mpu = adap->algo_data;

    /* the i2c API is an interface and not
     * a device per se, so only we know to
     * tell the hardware to wake up */
    pm_runtime_get_sync(&mpu->aux.dev);
    ...
}
```

Runtime Power Management

Linux Device Drivers

Bill Gatliff

`bgat@billgatliff.com`

Freelance Embedded Systems Developer