Linux I2C Device Drivers

Bill Gatliff
bgat@billgatliff.com

Freelance Embedded Systems Developer

Overview

Roadmap:

- I2C kernel API
- `struct i2c_adapter`
- `struct i2c_client`
- `struct i2c_driver`
- `struct i2c_board_info`
- Examples

I2C Kernel API

Transaction-oriented:

- More complicated than simple read, write
- Requres a bus adapter
- Sleeps while waiting for responses

No sleeping!

- Can only use in a process context
- NOT in interrupt handlers, tasklets, etc.

I2C Kernel API

"Adapter":

- I2c bus adapter
- One per bus
- Chips always connect to adapters

I2C Kernel API

"Driver":

- Associated with zero or more clients
- Matched with chips based on text names
- Related to Device Model

"Client":

- A.k.a. the chip
- Always associated with an adapter
- Each chip has a bus address

I2C Kernel API

"Board Info":

- Associates a chip, adapter and driver

i2c_smbus_read_byte()

Reads a single byte:

- Host sends address, sets R (read) bit
- Target must respond with exactly one byte
- Return value is negative on error

```
s32 i2c_smbus_read_byte(
    struct i2c_client *client);
```

i2c_smbus_read_byte_data()

Also reads a single byte:

- ... from a specified register
- (Useful only if the chip works that way)

```
s32 i2c_smbus_read_byte_data(
    struct i2c_client *client, u8 reg);
```

`struct i2c_client`

The “client” pointer:

- Identifies the chip, adapter
- Used by the i2c subsystem
- Generally opaque to API users

“Where does it come from?”

- Returned from `i2c_new_device()`
- Can be created manually in special circumstances

struct i2c_client

```
struct i2c_client {
  unsigned short      addr;
  char                name[I2C_NAME_SIZE];
  struct i2c_adapter *adapter;
  struct i2c_driver  *driver;
  struct device       dev;
  int                 irq;
};
```

struct i2c_client

```
foo()
{
  struct i2c_client c;
  _s32 b;

  c.addr = ADDR;
  c.adapter = i2c_get_adapter(0);
  b = i2c_smbus_read_byte(&c);
  ...
  i2c_put_adapter(c.adapter);
}
```

struct i2c_new_device()

Adds a device to a bus:

- Associates the device and adapter
- Produces the `struct i2c_client`

```
struct i2c_client *
    i2c_new_device(struct i2c_adapter *a,
                   struct i2c_board_info *b);
```

struct i2c_board_info

Information about an I2C chip:

- Associates the device and adapter
- Captures platform-specific information

```
struct i2c_board_info {
  char                type[I2C_NAME_SIZE],
  unsigned short      addr;
  void               *platform_data;
  ...
  int                 irq;
  ...
};
```

i2c_driver

```
struct i2c_driver {
  ...
  int  (*probe   )(struct i2c_client *,
                   const struct i2c_device_id *);
  int  (*remove  )(struct i2c_client *);

  void (*shutdown)(struct i2c_client *);
  int  (*suspend )(struct i2c_client *,
                   pm_message_t mesg);
  int  (*resume  )(struct i2c_client *);
  ...
  struct device_driver driver;
  const struct i2c_device_id *id_table;
};
```

`i2c_driver`

(Haven’t I seen that before?)

`bma250.c`

The Bosch, GmbH `bma250` accelerometer:

- Three-axis acceleration
- I2C interface
- Extensive on-board power management
- Optional motion-triggered interrupt

`bma250.c`

How to model this?

- An `i2c_client` for the low-level interface
- Attributes for each control register
- An `evdev` a.k.a. “input device” for X, Y, Z data
- Blocking show-attribute for interrupt event

`bma250.c`

Challenges:

- Take advantage of chip's onboard power management
- Use runtime-pm for device (model) management
- Keep code as simple as possible, but no simpler

`bma250.c`

Motivations:

- Get “first light” as quickly as possible
- Fully describe the chip to Linux
- Offer extensive, non-disruptive hooks for troubleshooting
- Absolute generalization, NO platform dependence
- Discoverable, information-oriented interfaces
- Utilize, expose unique chip features if possible
- Robust, straightforward code

Control Registers

```
enum {
  BMA250_REG_CHIP_ID = 0,

  /* TODO: marked as ‘‘reserved’’ in my datasheet! */
  BMA250_REG_VERSION = 1,

  BMA250_REG_X_AXIS_LSB = 2,
  BMA250_REG_X_AXIS_MSB = 3,
  BMA250_REG_Y_AXIS_LSB = 4,
  BMA250_REG_Y_AXIS_MSB = 5,
  ...
};
```

Control Registers

```
ssize_t bma_show_CHIP_ID(struct device *dev,
                         struct device_attribute *attr,
                         char *buf)
{
  struct bma *bma = dev_get_drvdata(dev);
  _s32 ret;

  mutex_lock_interruptible(&bma->mutex);
  ret = i2c_smbus_read_byte_data(bma->client,
                                 BMA250_CHIP_ID);
  mutex_unlock(&bma->mutex);

  return ret < 0 ? ret : sprintf(buf, ‘‘%02x\n’’, ret);
}
```

Control Registers

Note:

- The previous code won't "scale" well
- (We'll come back to this later)

Device Data Structure

Captures driver data:

- Client pointer, for attributes and elsewhere
- Voltage regulator references
- Mutexes, completions, etc.
- Cached values of some registers
- ...

Device Data Structure

```
struct bma {
  struct i2c_client *client;
  struct regulator  *vdd;
  struct regulator  *vddio;
  struct mutex       mutex;
  struct input_dev  *input;
  ...
  int POWER, RANGE, BANDWIDTH;
  ...
}
```

`probe()`

Invoked at device-driver binding:

- Allocate device data structure
- Get chip under control
- Publish interfaces
- ...
- Profit!

`probe()`

Watch out!

- Device vs. driver vs. interface

NOW schema is key!

`probe()`

`i2c_client`:

- How we are related to other devices

Attributes:

- Modeling the chip data to users

`input_dev`:

- Uniform data protocol to users

probe()

```
static int bma_probe(struct i2c_client *client,
                     const struct i2c_device_id *id)
{
  struct bma *bma;

  bma = kzalloc(sizeof *bma, GFP_KERNEL);
  i2c_set_client_data(client, bma);
  bma->client = client;
  ...
```

probe()

```
...
bma->vdd = regulator_get(&bma->client->dev, ‘‘VDD’’);
regulator_set_voltage(bma->vdd, 1620000,36000000);
...
ret = bma_read_CHIP_ID(bma);
...
pm_runtime_enable(&bma->client->dev);
pm_runtime_resume(&bma->client->dev);
...
```

probe()

```
...
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);
chip_id = ret;
...
```

probe()

```
...
sysfs_create_group(&bma->client->dev.kobj,
                   &bma_attribute_group);

bma->input = input_allocate_device();
input_set_drvdata(bma->input, bma);
bma->input->open = bma_input_open;
...
ret = input_register_device(bma->input);
...
```

probe()

```
  ...
  if (pdata->irq > 0) {
    bma->irq = pdata->irq;

    /* TODO: IRQF flags should come from platform data */
    ret = request_threaded_irq(bma->irq,
              NULL, bma_irq_handler,
              pdata->irq_flags ?
              pdata->irq_flags : 0 /* TODO: */,
              bma->client->name, bma);
  }
  ...
  return 0;
}
```

Runtime Power Management

General ideas:

- In `runtime_suspend()`, we are NOT in use
- After `runtime_resume()`, we might be

Do a state-transition diagram!

Runtime Power Management

```
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;
  ...
```

Runtime Power Management

```
  ...
  /* drive the chip into its SUSPEND state */
  ret = __bma_reg_write_POWER(bma,
                              BMA250_REG_POWER__SUSPEND);

  /* tell Linux we don’t need our regulator now */
  regulator_disable(bma->vdd);

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

Runtime Power Management

```
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;

  regulator_enable(bma->vdd);
  ...
```

Runtime Power Management

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

Runtime Power Management

```
  ...
  /* place chip into SUSPEND mode; other entry
   * points will bump 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;
}
```

Runtime Power Management

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

  ret = strict_strtoul(buf, 16, &v);
  if (ret)
    return ret;
  ...
```

Runtime Power Management

```
...
pm_runtime_get_sync(dev);
ret = mutex_lock_interruptible(&bma->mutex);
if (ret < 0)
  return ret;
...
```

Runtime Power Management

```
    ...
    __bma_push_mode_active(bma);
    ret = __bma_reg_write_RANGE(bma, v);
    __bma_pop_mode(bma);

    mutex_unlock(&bma->mutex);
    pm_runtime_mark_last_busy(dev);
    pm_runtime_put_autosuspend(dev);
    return (ret < 0) ? ret : len;
}
```

Runtime Power Management

```
int __bma_push_mode_active(struct bma250 *bma)
{
  BUG_ON(bma->pushpop_count != 0);

  bma->pushpop_count++;
  bma->prev_POWER = bma->POWER;

  if (bma->POWER & (BMA250_REG_POWER__SUSPEND
      | BMA250_REG_POWER__LOWPOWER_EN))
    return __bma_reg_write_POWER(bma, 0);
  return 0;
}
```

Platform suspend and resume

General idea:

- On suspend, save device state and quiet the device
- On resume, put THAT state back

Linux I2C Device Drivers

Bill Gatliff
bgat@billgatliff.com

Freelance Embedded Systems Developer