Linux I2C Device Drivers

Best-Practice Guidelines

Bill Gatliff
`bgat@billgatliff.com`

Freelance Embedded Systems Developer

“Best” Practices?!

Things that have worked for me:

- Learned over two decades of things NOT working!

Your mileage may vary:

- If yours is truly better, share it!
- My solutions might not address your problems

“Best” Practices?!

General theories:

- Not “if”, but “when” it breaks
- What will I need at that time?
- What do I wish I had last time?

But:

- We can’t unduly burden the nominal cases
- We must not increase the risk of defects!

“Best” Practices?!

Goals and objectives:

- Make our tools bring solutions
- Free our minds to understand the problems
- Redundancy is evil!

All of this is a work-in-progress

My Best Practices

In general:

- Fully embrace the Device Model
- Express our needs, make Linux address them
- Never tell Linux what to do, only tell it what we need
- (Whinging is ok here, but ONLY here!)

My Best Practices

More specifically:

- Control registers always have attributes
- Absolutely no platform-specific code in drivers
- No callbacks which tell Linux what to do
- No redundant code for redundant functionality
- NEVER trust external inputs

Control Register Accessors

Peripheral control registers:

- Be absolutely paranoid about their values!
- Read-only, write-only
- Reserved bits must never be touched!

Potentialy LOTS of redundant code:

- There’s an “app” for that! :-)

Control Register Accessors

The solution:

- Yes!

Control Register Accessors

```
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 Register Accessors

```
enum {
  BMA250_REG_CHIP_ID = 0,
  BMA250_REG_CHIP_ID__reserved = 0,
  ...
  BMA250_REG_X_AXIS_LSB = 4,
  BMA250_REG_X_AXIS_LSB__reserved = 0x3e,
  ...
};
```

Control Register Accessors

```
ssize_t bma_show_X_AXIS_LSB(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_REG_X_AXIS_LSB);
  mutex_unlock(&bma->mutex);
  ret &= BMA250_REG_X_AXIS_LSB__reserved;
  return ret < 0 ? ret : sprintf(buf, ‘‘%02x\n’’, ret);
}
```

Control Register Accessors

Ouch!

- Tedious, redundant, error-prone

Templates to the rescue!

- In C, they are called “macros”

Control Register Accessors

```
#define BMA_REG_READ(_name) \
int __bma_reg_read_##_name(struct bma *bma) \
{ \
  int ret = __bma_reg_read(bma, BMA250_REG_##_name); \
  if (ret < 0) \
    return ret; \
  ret &= ~BMA250_REG_##_name ## __reserved; \
  dev_dbg(&bma->client->dev, ‘‘%s: %02x\n’’, \
          __func__, ret); \
  return ret; \
}
```

Control Register Accessors

```
#define BMA_REG_WRITE(_name) \
int __bma_reg_write_##_name(struct bma *bma, int v) \
{ \
  dev_dbg(&bma->client->dev, ‘‘%s: %02x\n’’, __func__, v); \
  v &= ~BMA250_REG_##_name ## __reserved; \
  return __bma_reg_write(bma, BMA250_REG_##_name, v); \
}
```

Control Register Accessors

```
...
BMA_REG_READ(X_AXIS_LSB);
BMA_REG_READ(Y_AXIS_LSB);
...
```

Control Register Accessors

What about write-only registers?

- Let’s make them look read-write!
- (This greatly simplifies suspend/resume)

Control Register Accessors

```
#define BMA_REG_WRITE(_name) \
int __bma_reg_write_##_name(struct bma *bma, int v) \
{ \
  int ret; \
  v &= ~BMA250_REG_##_name ## __reserved; \
  ret = __bma_reg_write(bma, BMA250_REG_##_name, v); \
  if (ret < 0) \
    return ret; \
    switch (BMA250_REG_##_name) { \
      case BMA250_REG_RANGE: bma->RANGE = v; break; \
      case BMA250_REG_RESET: usleep(4000); break; \
    } \
  return 0; \
}
```

Control Register Accessors

```
#define BMA_REG_READ(_name) \
int __bma_reg_read_##_name(struct bma250 *bma) \
{ \
  int ret = __bma_reg_read(bma, BMA250_REG_##_name); \
  if (ret < 0) \
    return ret; \
    ret &= ~BMA250_REG_##_name ## __reserved; \
    switch (BMA250_REG_##_name) { \
      case BMA250_REG_RANGE: bma->RANGE = ret; break; \
    } \
    return ret; \
  }
```

Control Register Accessors

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

  mutex_lock_interruptible(&bma->mutex);
  ret = __bma250_reg_read_X_AXIS_LSB(bma);
  mutex_unlock(&bma->mutex);

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

Control Register Attribute Files

Peripheral controls:

- If it has one, I want to see it ... from the command line

Essential for root-cause analysis!

Control Register Attribute Files

```
#define BMA_REG_ATTR_STORE(_name) \
ssize_t bma_store_##_name(struct device *dev, \
                          struct device_attribute *attr, \
                          const char *buf, size_t len) \
{ \
  struct bma *bma = dev_get_drvdata(dev); \
  int ret; \
  unsigned long v; \
  ret = strict_strtoul(buf, 16, &v); \
  if (ret) \
    return ret; \
  ...
```

Control Register Attribute Files

```
...
pm_runtime_get_sync(dev); \
ret = mutex_lock_interruptible(&bma->mutex); \
if (ret < 0) \
  return ret; \
switch (BMA250_REG_##_name) { \
  case BMA250_REG_POWER: \
  ret = __bma_reg_write_POWER(bma, v); \
  break; \
...
```

Control Register Attribute Files

```
  ...
  default: \
    __bma_push_mode_active(bma); \
    ret = __bma_reg_write_##_name(bma, v);\
    __bma_pop_mode(bma); \
  break; \
} \
mutex_unlock(&bma->mutex); \
pm_runtime_mark_last_busy(dev); \
pm_runtime_put_autosuspend(dev); \
return (ret < 0) ? ret : len; \
```

Control Register Attribute Files

```
#define BMA_REG(_name) \
BMA_REG_READ(_name) \
BMA_REG_ATTR_SHOW(_name) \
DEVICE_ATTR(_name, S_IRUGO, bma_show_##_name, NULL);

...
BMA_REG(BMA_REG_X_AXIS_LSB);
BMA_REG(BMA_REG_Y_AXIS_LSB);
...
```

Control Register Attribute Files

```
#define BMA_REG_READWRITE(_name) \
BMA_REG_READ(_name) \
BMA_REG_WRITE(_name) \
BMA_REG_ATTR_SHOW(_name) \
BMA_REG_ATTR_STORE(_name) \
DEVICE_ATTR(_name, S_IRUGO | S_IWUSR, \
  bma_show_##_name, bma_store_##_name);

...
BMA_REG_READWRITE(BMA_REG_POWER);
...
```

Control Register Attribute Files

```
static struct attribute *bma_attributes[] = {
  &dev_attr_CHIP_ID.attr,
  &dev_attr_VERSION.attr,
  &dev_attr_TEMP.attr,
  &dev_attr_STATUS.attr,
  &dev_attr_DATA_INT.attr,
  &dev_attr_TAP_SLOPE_INT_STATUS.attr,
  ...
  };
```

Access Layer Portability

Choices, choices...

- I2C?
- SPI?

Is there a way to do both?

- I'm glad you asked! :-)

Access Layer Portability

```
static int __bma_reg_read(struct bma *bma, int reg)
{
  if (bma->client)
    return i2c_smbus_read_byte_data(bma->client, reg);
  if (bma->spi)
    return bma_spi_read_byte_data(bma->spi, reg);
  return -ENODEV;
}
```

Access Layer Portability

And then:

- Register a SPI and/or I2C driver
- Register a SPI and/or I2C device

Both drivers use `__bma_reg_read()`

No Platform-Specific Code!

```
struct foo_board_data {
  void (callback*)(void);
  };

  struct foo_board_data foo_board = {
    .callback = do_something_horrible();
  };

  struct platform_device foo_device = {
    .dev.platform_data = &foo_board,
  };
```

No Platform-Specific Code!

```
int foo_driver_method(struct foo *foo)
{
  ...
  foo->board_data->callback();
  ...
}
```

No Platform-Specific Code!

"Why is this so bad?"

- Because Linux doesn't know what is going on!

Linux can't help you if you don't ask

- And you may get in its way if you don't

No Platform-Specific Code!

Why are you doing it?

- Regulator reconfiguration?
- GPIO configuration changes?

There are APIs for that!

- Regulator notifier callbacks
- Pinctl API (ongoing)

No Platform-Specific Code!

```
struct foo_board_data {
  const char *vdd_name;
  };

  struct foo_board_data foo_board = {
    .vdd_name = ''VREG1234'',
  };

  struct platform_device foo_device = {
    .dev.platform_data = &foo_board,
  };
```

No Platform-Specific Code!

```
int foo_probe(...)
{
  ...
  /* TODO: no! */
  r = regulator_get(NULL, foo->board_data->vdd_name);
  ...
  regulator_enable(r);
  ...
}
```

No Platform-Specific Code!

Tell Linux only what you need:

- ... and make Linux figure out the solution

This is precisely what Device Model is for!

No Platform-Specific Code!

```
int foo_probe(struct device *dev, ...)
{
  ...
  /* get our VDD regulator, if any */
  r = regulator_get(dev, ‘‘VDD’’);
  ...
  regulator_enable(r);
  ...
}
```

Never Trust External Inputs

The real world sometimes Gets Real:

- Runaway incoming interrupts
- Sensor values out of range
- Persistent watchdog timeouts
- Bouncing switches
- ...

Never Trust External Inputs

```
irqreturn_t handler(int irq, void *foo)
{
  /* TODO: prepare to go !boom! */
  ...
}
```

Never Trust External Inputs

```
irq_handler(int irq, void *foo)
{
  irq_disable_nosync(irq);
  hrtimer_start(...);
}

timer_callback(...)
{
  irq_enable(irq);
}
```

Never Trust External Inputs

Watchdog timers:

- Do them right, or not at all
- Often a false sense of security

"Right":

- Double-sided, or layered access
- ALWAYS check reset status

Linux I2C Device Drivers

Best-Practice Guidelines

Bill Gatliff

`bgat@billgatliff.com`

Freelance Embedded Systems Developer