Upgrading I2C Drivers to the new 2.6 Driver Model

Ben Dooks <ben-linux@fluff.org>

Introduction

This guide outlines how to alter existing Linux 2.6 client drivers from the old to the new new binding methods.

Example old-style driver

```
struct example_state {
	struct i2c_client	client;
	....
};

static struct i2c_driver example_driver;

static unsigned short ignore[] = { I2C_CLIENT_END };
static unsigned short normal_addr[] = { OUR_ADDR, I2C_CLIENT_END };

I2C_CLIENT_INSMOD;

static int example_attach(struct i2c_adapter *adap, int addr, int kind)
{
	struct example_state *state;
	struct device *dev = &adap->dev;  /* to use for dev_ reports */
	int ret;

	state = kzalloc(sizeof(struct example_state), GFP_KERNEL);
	if (state == NULL) {
		dev_err(dev, "failed to create our state\n");
		return -ENOMEM;
	}

	example->client.addr    = addr;
	example->client.flags   = 0;
	example->client.adapter = adap;

	i2c_set_clientdata(&state->i2c_client, state);
	strlcpy(client->i2c_client.name, "example", I2C_NAME_SIZE);

	ret = i2c_attach_client(&state->i2c_client);
	if (ret < 0) {
		dev_err(dev, "failed to attach client\n");
		kfree(state);
		return ret;
	}

	dev = &state->i2c_client.dev;
```

```
        /* rest of the initialisation goes here. */

        dev_info(dev, "example client created\n");

        return 0;
}

static int __devexit example_detach(struct i2c_client *client)
{
        struct example_state *state = i2c_get_clientdata(client);

        i2c_detach_client(client);
        kfree(state);
        return 0;
}

static int example_attach_adapter(struct i2c_adapter *adap)
{
        return i2c_probe(adap, &addr_data, example_attach);
}

static struct i2c_driver example_driver = {
        .driver         = {
                .owner          = THIS_MODULE,
                .name           = "example",
        },
        .attach_adapter = example_attach_adapter,
        .detach_client  = __devexit_p(example_detach),
        .suspend        = example_suspend,
        .resume         = example_resume,
};
```

Updating the client

The new style binding model will check against a list of supported devices and their associated address supplied by the code registering the busses. This means that the driver .attach_adapter and .detach_adapter methods can be removed, along with the addr_data, as follows:

```
- static struct i2c_driver example_driver;

- static unsigned short ignore[] = { I2C_CLIENT_END };
- static unsigned short normal_addr[] = { OUR_ADDR, I2C_CLIENT_END };

- I2C_CLIENT_INSMOD;

- static int example_attach_adapter(struct i2c_adapter *adap)
- {
-       return i2c_probe(adap, &addr_data, example_attach);
- }

 static struct i2c_driver example_driver = {
-       .attach_adapter = example_attach_adapter,
-       .detach_client  = __devexit_p(example_detach),
 }
```

Add the probe and remove methods to the i2c_driver, as so:

```
 static struct i2c_driver example_driver = {
+       .probe          = example_probe,
+       .remove         = __devexit_p(example_remove),
 }
```

Change the example_attach method to accept the new parameters
which include the i2c_client that it will be working with:

```
- static int example_attach(struct i2c_adapter *adap, int addr, int kind)
+ static int example_probe(struct i2c_client *client,
+                          const struct i2c_device_id *id)
```

Change the name of example_attach to example_probe to align it with the
i2c_driver entry names. The rest of the probe routine will now need to be
changed as the i2c_client has already been setup for use.

The necessary client fields have already been setup before
the probe function is called, so the following client setup
can be removed:

```
-       example->client.addr    = addr;
-       example->client.flags   = 0;
-       example->client.adapter = adap;
-
-       strlcpy(client->i2c_client.name, "example", I2C_NAME_SIZE);
```

The i2c_set_clientdata is now:

```
-       i2c_set_clientdata(&state->client, state);
+       i2c_set_clientdata(client, state);
```

The call to i2c_attach_client is no longer needed, if the probe
routine exits successfully, then the driver will be automatically
attached by the core. Change the probe routine as so:

```
-       ret = i2c_attach_client(&state->i2c_client);
-       if (ret < 0) {
-               dev_err(dev, "failed to attach client\n");
-               kfree(state);
-               return ret;
-       }
```

Remove the storage of 'struct i2c_client' from the 'struct example_state'
as we are provided with the i2c_client in our example_probe. Instead we
store a pointer to it for when it is needed.

```
struct example_state {
-       struct i2c_client       client;
+       struct i2c_client       *client;
```

the new i2c client as so:

```
-       struct device *dev = &adap->dev;  /* to use for dev_ reports */
+       struct device *dev = &i2c_client->dev;  /* to use for dev_ reports */
```

And remove the change after our client is attached, as the driver no
longer needs to register a new client structure with the core:

```
-       dev = &state->i2c_client.dev;
```

In the probe routine, ensure that the new state has the client stored
in it:

```
static int example_probe(struct i2c_client *i2c_client,
                         const struct i2c_device_id *id)
{
        struct example_state *state;
        struct device *dev = &i2c_client->dev;
        int ret;

        state = kzalloc(sizeof(struct example_state), GFP_KERNEL);
        if (state == NULL) {
                dev_err(dev, "failed to create our state\n");
                return -ENOMEM;
        }

+       state->client = i2c_client;
```

Update the detach method, by changing the name to _remove and
to delete the i2c_detach_client call. It is possible that you
can also remove the ret variable as it is not not needed for
any of the core functions.

```
- static int __devexit example_detach(struct i2c_client *client)
+ static int __devexit example_remove(struct i2c_client *client)
{
        struct example_state *state = i2c_get_clientdata(client);

-       i2c_detach_client(client);
```

And finally ensure that we have the correct ID table for the i2c-core
and other utilities:

```
+ struct i2c_device_id example_idtable[] = {
+       { "example", 0 },
+       { }
+};
+
+MODULE_DEVICE_TABLE(i2c, example_idtable);

static struct i2c_driver example_driver = {
        .driver         = {
                .owner          = THIS_MODULE,
                .name           = "example",
        },
+       .id_table       = example_ids,
```

Our driver should now look like this:

```
struct example_state {
        struct i2c_client       *client;
        ....
};

static int example_probe(struct i2c_client *client,
                         const struct i2c_device_id *id)
{
        struct example_state *state;
        struct device *dev = &client->dev;

        state = kzalloc(sizeof(struct example_state), GFP_KERNEL);
        if (state == NULL) {
                dev_err(dev, "failed to create our state\n");
                return -ENOMEM;
        }

        state->client = client;
        i2c_set_clientdata(client, state);

        /* rest of the initialisation goes here. */

        dev_info(dev, "example client created\n");

        return 0;
}

static int __devexit example_remove(struct i2c_client *client)
{
        struct example_state *state = i2c_get_clientdata(client);

        kfree(state);
        return 0;
}

static struct i2c_device_id example_idtable[] = {
        { "example", 0 },
        { }
};

MODULE_DEVICE_TABLE(i2c, example_idtable);

static struct i2c_driver example_driver = {
        .driver         = {
                .owner          = THIS_MODULE,
                .name           = "example",
        },
        .id_table       = example_idtable,
        .probe          = example_probe,
        .remove         = __devexit_p(example_remove),
        .suspend        = example_suspend,
        .resume         = example_resume,
};
```