This is a small guide for those who want to write kernel drivers for I2C
or SMBus devices, using Linux as the protocol host/master (not slave).

To set up a driver, you need to do several things. Some are optional, and
some things can be done slightly or completely different. Use this as a
guide, not as a rule book!

General remarks
===============

Try to keep the kernel namespace as clean as possible. The best way to
do this is to use a unique prefix for all global symbols. This is
especially important for exported symbols, but it is a good idea to do
it for non-exported symbols too. We will use the prefix `foo_' in this
tutorial.

The driver structure
====================

Usually, you will implement a single driver structure, and instantiate
all clients from it. Remember, a driver structure contains general access
routines, and should be zero-initialized except for fields with data you
provide. A client structure holds device-specific information like the
driver model device node, and its I2C address.

```
static struct i2c_device_id foo_idtable[] = {
        { "foo", my_id_for_foo },
        { "bar", my_id_for_bar },
        { }
};

MODULE_DEVICE_TABLE(i2c, foo_idtable);

static struct i2c_driver foo_driver = {
        .driver = {
                .name   = "foo",
        },

        .id_table       = foo_ids,
        .probe          = foo_probe,
        .remove         = foo_remove,
        /* if device autodetection is needed: */
        .class          = I2C_CLASS_SOMETHING,
        .detect         = foo_detect,
        .address_list   = normal_i2c,

        .shutdown       = foo_shutdown,  /* optional */
        .suspend        = foo_suspend,   /* optional */
        .resume         = foo_resume,    /* optional */
        .command        = foo_command,   /* optional, deprecated */
}
```

The name field is the driver name, and must not contain spaces. It
should match the module name (if the driver can be compiled as a module),

although you can use MODULE_ALIAS (passing "foo" in this example) to add
another name for the module. If the driver name doesn't match the module
name, the module won't be automatically loaded (hotplug/coldplug).

All other fields are for call-back functions which will be explained
below.

Extra client data
==================

Each client structure has a special `data' field that can point to any
structure at all. You should use this to keep device-specific data.

        /* store the value */
        void i2c_set_clientdata(struct i2c_client *client, void *data);

        /* retrieve the value */
        void *i2c_get_clientdata(const struct i2c_client *client);

Note that starting with kernel 2.6.34, you don't have to set the `data' field
to NULL in remove() or if probe() failed anymore. The i2c-core does this
automatically on these occasions. Those are also the only times the core will
touch this field.

Accessing the client
====================

Let's say we have a valid client structure. At some time, we will need
to gather information from the client, or write new information to the
client.

I have found it useful to define foo_read and foo_write functions for this.
For some cases, it will be easier to call the i2c functions directly,
but many chips have some kind of register-value idea that can easily
be encapsulated.

The below functions are simple examples, and should not be copied
literally.

```
int foo_read_value(struct i2c_client *client, u8 reg)
{
        if (reg < 0x10) /* byte-sized register */
                return i2c_smbus_read_byte_data(client, reg);
        else            /* word-sized register */
                return i2c_smbus_read_word_data(client, reg);
}

int foo_write_value(struct i2c_client *client, u8 reg, u16 value)
{
        if (reg == 0x10)        /* Impossible to write - driver error! */
                return -EINVAL;
        else if (reg < 0x10)    /* byte-sized register */
                return i2c_smbus_write_byte_data(client, reg, value);
        else                    /* word-sized register */
```

```
                    return i2c_smbus_write_word_data(client, reg, value);
}
```

Probing and attaching
=====================

The Linux I2C stack was originally written to support access to hardware
monitoring chips on PC motherboards, and thus used to embed some assumptions
that were more appropriate to SMBus (and PCs) than to I2C. One of these
assumptions was that most adapters and devices drivers support the SMBUS_QUICK
protocol to probe device presence. Another was that devices and their drivers
can be sufficiently configured using only such probe primitives.

As Linux and its I2C stack became more widely used in embedded systems
and complex components such as DVB adapters, those assumptions became more
problematic. Drivers for I2C devices that issue interrupts need more (and
different) configuration information, as do drivers handling chip variants
that can't be distinguished by protocol probing, or which need some board
specific information to operate correctly.

Device/Driver Binding

System infrastructure, typically board-specific initialization code or
boot firmware, reports what I2C devices exist. For example, there may be
a table, in the kernel or from the boot loader, identifying I2C devices
and linking them to board-specific configuration information about IRQs
and other wiring artifacts, chip type, and so on. That could be used to
create i2c_client objects for each I2C device.

I2C device drivers using this binding model work just like any other
kind of driver in Linux: they provide a probe() method to bind to
those devices, and a remove() method to unbind.

```
        static int foo_probe(struct i2c_client *client,
                             const struct i2c_device_id *id);
        static int foo_remove(struct i2c_client *client);
```

Remember that the i2c_driver does not create those client handles. The
handle may be used during foo_probe(). If foo_probe() reports success
(zero not a negative status code) it may save the handle and use it until
foo_remove() returns. That binding model is used by most Linux drivers.

The probe function is called when an entry in the id_table name field
matches the device's name. It is passed the entry that was matched so
the driver knows which one in the table matched.

Device Creation

If you know for a fact that an I2C device is connected to a given I2C bus,
you can instantiate that device by simply filling an i2c_board_info
structure with the device address and driver name, and calling

i2c_new_device(). This will create the device, then the driver core will
take care of finding the right driver and will call its probe() method.
If a driver supports different device types, you can specify the type you
want using the type field. You can also specify an IRQ and platform data
if needed.

Sometimes you know that a device is connected to a given I2C bus, but you
don't know the exact address it uses. This happens on TV adapters for
example, where the same driver supports dozens of slightly different
models, and I2C device addresses change from one model to the next. In
that case, you can use the i2c_new_probed_device() variant, which is
similar to i2c_new_device(), except that it takes an additional list of
possible I2C addresses to probe. A device is created for the first
responsive address in the list. If you expect more than one device to be
present in the address range, simply call i2c_new_probed_device() that
many times.

The call to i2c_new_device() or i2c_new_probed_device() typically happens
in the I2C bus driver. You may want to save the returned i2c_client
reference for later use.

Device Detection

Sometimes you do not know in advance which I2C devices are connected to
a given I2C bus. This is for example the case of hardware monitoring
devices on a PC's SMBus. In that case, you may want to let your driver
detect supported devices automatically. This is how the legacy model
was working, and is now available as an extension to the standard
driver model.

You simply have to define a detect callback which will attempt to
identify supported devices (returning 0 for supported ones and -ENODEV
for unsupported ones), a list of addresses to probe, and a device type
(or class) so that only I2C buses which may have that type of device
connected (and not otherwise enumerated) will be probed. For example,
a driver for a hardware monitoring chip for which auto-detection is
needed would set its class to I2C_CLASS_HWMON, and only I2C adapters
with a class including I2C_CLASS_HWMON would be probed by this driver.
Note that the absence of matching classes does not prevent the use of
a device of that type on the given I2C adapter. All it prevents is
auto-detection; explicit instantiation of devices is still possible.

Note that this mechanism is purely optional and not suitable for all
devices. You need some reliable way to identify the supported devices
(typically using device-specific, dedicated identification registers),
otherwise misdetections are likely to occur and things can get wrong
quickly. Keep in mind that the I2C protocol doesn't include any
standard way to detect the presence of a chip at a given address, let
alone a standard way to identify devices. Even worse is the lack of
semantics associated to bus transfers, which means that the same
transfer can be seen as a read operation by a chip and as a write
operation by another chip. For these reasons, explicit device
instantiation should always be preferred to auto-detection where
possible.

Device Deletion

Each I2C device which has been created using i2c_new_device() or
i2c_new_probed_device() can be unregistered by calling
i2c_unregister_device(). If you don't call it explicitly, it will be
called automatically before the underlying I2C bus itself is removed, as a
device can't survive its parent in the device driver model.

Initializing the driver
========================

When the kernel is booted, or when your foo driver module is inserted,
you have to do some initializing. Fortunately, just registering the
driver module is usually enough.

```
static int __init foo_init(void)
{
        return i2c_add_driver(&foo_driver);
}

static void __exit foo_cleanup(void)
{
        i2c_del_driver(&foo_driver);
}

/* Substitute your own name and email address */
MODULE_AUTHOR("Frodo Looijaard <frodol@dds.nl>"
MODULE_DESCRIPTION("Driver for Barf Inc. Foo I2C devices");

/* a few non-GPL license types are also allowed */
MODULE_LICENSE("GPL");

module_init(foo_init);
module_exit(foo_cleanup);
```

Note that some functions are marked by `__init'. These functions can
be removed after kernel booting (or module loading) is completed.
Likewise, functions marked by `__exit' are dropped by the compiler when
the code is built into the kernel, as they would never be called.

Power Management
================

If your I2C device needs special handling when entering a system low
power state -- like putting a transceiver into a low power mode, or
activating a system wakeup mechanism -- do that in the suspend() method.
The resume() method should reverse what the suspend() method does.

These are standard driver model calls, and they work just like they
would for any other driver stack. The calls can sleep, and can use
I2C messaging to the device being suspended or resumed (since their

parent I2C adapter is active when these calls are issued, and IRQs
are still enabled).

System Shutdown
===============

If your I2C device needs special handling when the system shuts down
or reboots (including kexec) -- like turning something off -- use a
shutdown() method.

Again, this is a standard driver model call, working just like it
would for any other driver stack: the calls can sleep, and can use
I2C messaging.

Command function
================

A generic ioctl-like function call back is supported. You will seldom
need this, and its use is deprecated anyway, so newer design should not
use it.

Sending and receiving
======================

If you want to communicate with your device, there are several functions
to do this. You can find all of them in <linux/i2c.h>.

If you can choose between plain I2C communication and SMBus level
communication, please use the latter. All adapters understand SMBus level
commands, but only some of them understand plain I2C!

Plain I2C communication
----

        int i2c_master_send(struct i2c_client *client, const char *buf,
                            int count);
        int i2c_master_recv(struct i2c_client *client, char *buf, int count);

These routines read and write some bytes from/to a client. The client
contains the i2c address, so you do not have to include it. The second
parameter contains the bytes to read/write, the third the number of bytes
to read/write (must be less than the length of the buffer, also should be
less than 64k since msg.len is u16.) Returned is the actual number of bytes
read/written.

        int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msg,
                         int num);

This sends a series of messages. Each message can be a read or write,
and they can be mixed in any way. The transactions are combined: no
stop bit is sent between transaction. The i2c_msg structure contains
for each message the client address, the number of bytes of the message

and the message data itself.

You can read the file `i2c-protocol' for more information about the
actual I2C protocol.

SMBus communication

```
        s32 i2c_smbus_xfer(struct i2c_adapter *adapter, u16 addr,
                           unsigned short flags, char read_write, u8 command,
                           int size, union i2c_smbus_data *data);
```

This is the generic SMBus function. All functions below are implemented
in terms of it. Never use this function directly!

```
        s32 i2c_smbus_read_byte(struct i2c_client *client);
        s32 i2c_smbus_write_byte(struct i2c_client *client, u8 value);
        s32 i2c_smbus_read_byte_data(struct i2c_client *client, u8 command);
        s32 i2c_smbus_write_byte_data(struct i2c_client *client,
                                      u8 command, u8 value);
        s32 i2c_smbus_read_word_data(struct i2c_client *client, u8 command);
        s32 i2c_smbus_write_word_data(struct i2c_client *client,
                                      u8 command, u16 value);
        s32 i2c_smbus_process_call(struct i2c_client *client,
                                   u8 command, u16 value);
        s32 i2c_smbus_read_block_data(struct i2c_client *client,
                                      u8 command, u8 *values);
        s32 i2c_smbus_write_block_data(struct i2c_client *client,
                                       u8 command, u8 length, const u8 *values);
        s32 i2c_smbus_read_i2c_block_data(struct i2c_client *client,
                                          u8 command, u8 length, u8 *values);
        s32 i2c_smbus_write_i2c_block_data(struct i2c_client *client,
                                           u8 command, u8 length,
                                           const u8 *values);
```

These ones were removed from i2c-core because they had no users, but could
be added back later if needed:

```
        s32 i2c_smbus_write_quick(struct i2c_client *client, u8 value);
        s32 i2c_smbus_block_process_call(struct i2c_client *client,
                                         u8 command, u8 length, u8 *values);
```

All these transactions return a negative errno value on failure. The 'write'
transactions return 0 on success; the 'read' transactions return the read
value, except for block transactions, which return the number of values
read. The block buffers need not be longer than 32 bytes.

You can read the file `smbus-protocol' for more information about the
actual SMBus protocol.

General purpose routines
========================

Below all general purpose routines are listed, that were not mentioned

before.

```
/* Return the adapter number for a specific adapter */
int i2c_adapter_id(struct i2c_adapter *adap);
```