I2C is a protocol for communication between devices. In this column, the author takes the reader through the process of writing I2C clients in Linux.

I2C is a multi-master synchronous serial communication protocol for devices. All devices have addresses through which they communicate with each other. The I2C protocol has three versions with different communication speeds 100kHz, 400kHz and 3.4MHz. The I2C protocol has a bus arbitration procedure through which the master is decided on the bus, and then the master supplies the clock for the system and reads and writes data on the bus. The device that is communicating with the master is the slave device.

The Linux I2C subsystem

The Linux I2C subsystem is the interface through which the system running Linux can interact with devices connected on the system's I2C bus. It is designed in such a manner that the system running Linux is always the I2C master. It consists of the following subsections.

I2C adapter: There can be multiple I2C buses on the board, so each bus on the system is represented in Linux using the *struct i2c_adapter (defined in include/linux/i2c.h)*. The following are the important fields present in this structure.

bus number: Each bus in the system is assigned a number that is present in the I2C adapter structure which represents it.

I2C algorithm: Each I2C bus operates with a certain protocol for communicating between devices. The algorithm that the bus uses is defined by this field. There are currently three algorithms for the I2C bus, which are pca, pcf and bitbanging. These algorithms are used to communicate with devices when the driver requests to write or read data from the device.

I2C client: Each device that is connected to the I2C bus on the system is represented using the *struct i2c_client (defined in include/linux/i2c.h)*. The following are the important fields present in this structure.

Address: This field consists of the address of the device on the bus. This address is used by the driver to communicate with the device.

Name: This field is the name of the device which is used to match the driver with the device.

Interrupt number: This is the number of the interrupt line of the device.

I2C adapter: This is the *struct i2c_adapter* which represents the bus on which this device is connected. Whenever the driver makes requests to write or read from the bus, this field is used to identify the bus on which this transaction is to be done and also which algorithm should be used to communicate with the device.

I2C driver: For each device on the system, there should be a driver that controls it. For the I2C device, the corresponding driver is represented by *struct i2c_driver (defined in include/linux/i2c.h).* The following are the important fields defined in this structure.

Driver.name: This is the name of the driver that is used to match the I2C device on the system with the driver.

Probe: This is the function pointer to the driver's probe routine, which is called when the device and driver are both found on the system by the Linux device driver subsystem.

To understand how to write I2C device information and the I2C driver, let's consider an example of

a system in which there are two devices connected on the I2C bus. A description of these devices is given below.

Device 1

Device type: EEPROM
Device name: eeprom_xyz
Device I2C address: 0x30
Device interrupt number: 4
Device bus number: 2

Device 2

Device type: Analogue to digital converter
Device name: adc_xyz
Device I2C address: 0x31
Device interrupt number: Not available
Device bus number: 2

Writing the I2C device file

I2C devices connected on the system are represented by *struct i2c_client.* This structure is not directly defined but, instead, struct *i2c_board_info* is defined in the board file. *struct i2c_client* is defined using *struct i2c_board_info* structure by the Linux I2C subsystem, the fields of the *i2c_board_info* object is copied to *i2c_client* object created.

Note: Board files reside in *arch/ folder* in Linux. For example, the board file for the ATSTK1000 board of the AVR32 architecture is *arch/avr32/boards/atstk1000.c* and the board file for Beagle Board of ARM OMAP3 architecture is *arch/arm/mach-omap2/board-omap3beagle.c.*

struct i2c_board_info (defined in include/linux/i2c.h) has the following important fields.
type: This is the name of the I2C device for which this structure is defined. This will be copied to the name field of *i2c_client object* created by the I2C subsystem.
addr: This is the address of the I2C device. This field will be copied to address the field of *i2c_client object* created by the I2C subsystem.
irq: This is the interrupt number of the I2C device. This field will be copied to the irq field of the *i2c_client object* created by the I2C subsystem.
An array of *struct i2c_board_info* object is created, where each object represents the I2C device connected on the bus. For our example system, the *i2c_board_info* object is written as follows:

```
static struct i2c_board_info xyz_devices[] = {
{
.type = "eeprom_xyz",
.addr = 0x30,
.irq = 4,
},
{
.type = "adc_xyz",
.addr = 0x31,
},
};
```

I2C device registration

I2C device registration is a process with which the kernel is informed about the device present on the I2C bus. The I2C device is registered using the *struct i2c_board_info* object defined. The kernel gets information about the device's address, bus number and name of the device being registered. Once the kernel gets this information, it stores this information in its global linked list *__i2c_board_list*, and when the *i2c_adapter* which represents this bus is registered, the kernel creates the *i2c_client object* from this *i2c_board_info* object.

I2C device registration is done in the board init code present in the board file. I2C devices are registered in the Linux kernel using the following two methods.

Case 1: In most cases, the bus number on which the device is connected is known; in this case the device is registered using the bus number. When the bus number is known, I2C devices are registered using the following API:

```
int i2c_register_board_info(int busnum, struct i2c_board_info
*info, unsigned len);
```

where,

busnum = the number of the bus on which the device is connected. This will be used to identify the *i2c_adapter* object for the device.

info = array of struct i2c_board_info object, which consists of information of all the devices present in the bus.

len = number of elements in the info array.

For our example system, I2C devices are registered as follows:

```
i2c_register_board_info(2, xyz_devices, ARRAY_SIZE(xyz_devices));
```

What the *i2c_register_board_info* does is link the *struct i2c_board_info* object in *__i2c_board_list*, which is the global linked list.

Now, when the I2C adapter is registered using *i2c_register_adapter* API (defined in drivers/i2c/i2c-core.c), it will search for devices that have the same bus number as the adapter, through the *i2c_scan_static_board_info* API. When the *i2c_board_info* object is found with the bus number which is the same as that of the adapter being registered, a new *i2c_client* object is created using the i2c_new_device API.

The *i2c_new_device* API creates a new struct i2c_client object and the fields of the *i2c_client* object are initialised with the fields of the *i2c_board_info* object. The *new i2c_client* object is then registered with the I2C subsystem. During registration, the kernel matches the name of all the I2C drivers with the name of the I2C client created. If any I2C driver's name matches with the I2C client, then the probe routine of the I2C driver will be called.

Case 2: In some cases, instead of the bus number, the *i2c_adapter* on which the device is connected is known; in this case, the device is registered using the *struct i2c_adapter* object.

When the *i2c_adapter* is known instead of the bus number, the I2C device is registered using the following API:

```
struct i2c_client *
i2c_new_device(struct i2c_adapter *adap, struct i2c_board_info
const *info);
```

where,

adap = i2c_adapter representing the bus on which the device is connected.

info = i2c_board_info object for each device.

In our example, the device is registered as follows.
For device 1:

```
i2c_new_device(adap, &xyz_devices[0]);
```

For device 2:

```
i2c_new_device(adap, &xyz_devices[1]);
```

Writing the I2C driver

As mentioned earlier, generally, the device files are present in the *arch/xyz_arch/boards* folder and similarly, the driver files reside in their respective driver folders. For example, typically, all the RTC drivers reside in the *drivers/rtc* folder and all the keyboard drivers reside in the *drivers/input/keyboard* folder.

Writing the I2C driver involves specifying the details of the *struct i2c_driver*. The following are the required fields for *struct i2c_driver* that need to be filled:

driver.name = name of the driver that will be used to match the driver with the device.

driver.owner = owner of the module. This is generally the *THIS_MODULE* macro.

probe = the probe routine for the driver, which will be called when any I2C device's name in the system matches with this driver's name.

Note: It's not just the names of the device and driver that are used to match the two. There are other methods to match them such as *id_table* but, for now, let's consider their names as the main parameter for matching. To understand the way in which the ID table is used, refer to the Linux source code.

For our example of the EEPROM driver, the driver file will reside in the *drivers/misc/eeprom* folder and we will give it a *name'eeprom_xyz.c*. The *struct i2c_driver* will be written as follows:

```
static struct i2c_driver eeprom_driver = {
.driver = {
.name = "eeprom_xyz",
.owner = THIS_MODULE,
},
.probe = eeprom_probe,
};
```

For our example of an *adc* driver, the driver file will reside in the *drivers/iio/adc* folder, which we will name as *adc_xyz.c*, and the *struct i2c_driver* will be written as follows:

```
static struct i2c_driver adc_driver = {
.driver = {
.name = "adc_xyz",
.owner = THIS_MODULE,
},
.probe = adc_probe,
};
```

The *struct i2c_driver* now has to be registered with the I2C subsystem. This is done in the *module_init* routine using the following API:

```
i2c_add_driver(struct i2c_driver *drv);
```

'where *drv i*s the *i2c_driver* structure written for the device.

For our example of an EEPROM system, the driver will be registered as:

```
i2c_add_driver(&eeprom_driver);
```
and the adc driver will be registered as:

```
i2c_add_driver(&adc_driver);
```
What this *i2c_add_driver* does is register the passed driver with the I2C subsystem and match the name of the driver with all the *i2c_client names*. If any of the names match, then the probe routine of the driver will be called and the *struct i2c_client* will be passed as the parameter to the probe routine. During the probe routine, it is verified that the device represented by the *i2c_client* passed to the driver is the actual device that the driver supports. This is done by trying to communicate with the device represented by *i2c_client* using the address present in the *i2c_client* structure. If this fails, it returns an error from the probe routine informing the Linux device driver subsystem that the device and driver are not compatible; or else it continues with creating device files, registering interrupts and registering with the application subsystem.
The probe skeleton for our example EEPROM system will be as follows:

```
static int eeprom_probe(struct i2c_client *client, const struct
i2c_device_id *id)
{
check if device exists;
if device error
{
return error;
}
else
{
do basic configuration of eeprom using
client->addr;

register with eeprom subsystem;

register the interrupt using client->irq;
}
return 0;
}

static int adc_probe(struct i2c_client *client, const struct
i2c_device_id *id)
{
check if device exists;
if device error
{
return error;
}
else
{
do basic configuration of adc using
client->addr;

register with adc subsystem;
}
return 0;
```

```
}
```
After the probe routine is called and all the required configuration is done, the device is active and the user space can read and write the device using system calls. For a very clear understanding of how to write *i2c_driver*, refer to the drivers present in the Linux source code for example, the RTC driver on *i2c* bus, the *drivers/rtc/rtc-ds1307.c* file and other driver files.
For reading and writing data on the I2C bus, use the following API.

Reading bytes from the I2C bus:

```
i2c_smbus_read_byte_data(struct i2c_client *client, u8 command);
```
client: i2c_client object received from driver probe routine.
command: the command that is to be transferred on the bus.

Reading words from the I2C bus:

```
i2c_smbus_read_word_data(struct i2c_client *client, u8 command);
```
client: i2c_client object received from driver probe routine.
command: the command that is to be transferred on the bus.

Writing bytes on an I2C bus:

```
i2c_smbus_write_byte_data(struct i2c_client *client, u8 command,
u8 data);
```
client: i2c_client object received from driver probe routine.
command: the command that is to be transferred on the bus.
data: the data that is to be written to the device.

Writing words on an I2C bus:

```
i2c_smbus_write_word_data(struct i2c_client *client, u8 command,
u16 data);
```
client: i2c_client object received from driver probe routine.
command: the command that is to be transferred on the bus.
data: the data that is to be written to the device
When the read or write command is issued, the request is completed using the adapters algorithm, which has the routines to read and write on the bus.