03 serial_ioctl

Moatasem Elsayed

Ioctl

System call

SYNOPSIS top

```
#include <sys/ioctl.h>

int ioctl(int fd, unsigned long request, ...);
```

DESCRIPTION top

The **ioctl**() system call manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g., terminals) may be controlled with **ioctl**() requests. The argument *fd* must be an open file descriptor.

The second argument is a device-dependent request code. The third argument is an untyped pointer to memory. It's traditionally **char** **argp* (from the days before **void *** was valid C), and will be so named for this discussion.

An **ioctl**() *request* has encoded in it whether the argument is an *in* parameter or *out* parameter, and the size of the argument *argp* in bytes. Macros and defines used in specifying an **ioctl**() *request* are located in the file *<sys/ioctl.h>*. See NOTES.

RETURN VALUE top

Usually, on success zero is returned. A few **ioctl**() requests use the return value as an output parameter and return a nonnegative value on success. On error, -1 is returned, and *errno* is set to indicate the error.

```c
    .fops={
        .owner = THIS_MODULE,
        .open = driver_open,
        .release = driver_close,
        .read = driver_read,
        .write = driver_write,
        .unlocked_ioctl=ioctl_handler
    }
};
```

```c
#ifndef IOCTL_GPIO
#define IOCTL_GPIO


#define WRITE_PIN _IOW('a', 'a', uint8_t *)

#endif
```

```c
#include "ioctl-gpio.h"

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <cstdint>
int main() {
    uint8_t pin_number=20;
    int dev = open("/dev/mygpio_number", O_WRONLY);
    if(dev == -1) {
        printf("Opening was not possible!\n");
        return -1;
    }

    ioctl(dev, WRITE_PIN, &pin_number);
    printf("The pin_number is %d\n", pin_number);

    close(dev);
    return 0;
}
```

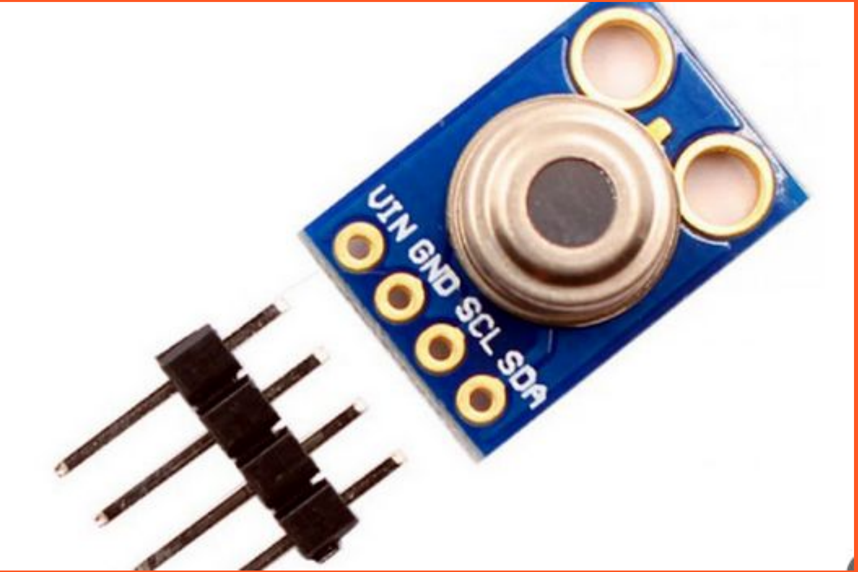
```c
uint8_t pin_number=21;
long ioctl_handler(struct file * f, unsigned int cmd, unsigned long arg)
{
    switch(cmd){
            case WRITE_PIN:
                gpio_set_value(pin_number, 0);
                gpio_free(pin_number);
                if(copy_from_user(&pin_number, (uint8_t *) arg, sizeof(pin_number))){
                    printk("ioctl_example - Error copying data from user!\n");
                    pin_number=21;
                }
                else{
                    printk("ioctl_example - Update the pin number to %d\n", pin_number);
                }
                break;

        }
    /* GPIO pin_number init */
    if(gpio_request(pin_number, "rpi-gpio")) {
        printk("Can not allocate GPIO\n");
        return -1;
    }
    /* Set GPIO 4 direction */
    if(gpio_direction_output(pin_number, GPIOF_INIT_LOW)) {
        printk("Can not set GPIO  output!\n");
        gpio_free(pin_number);
        return -1;
    }
    return 0;
}
```

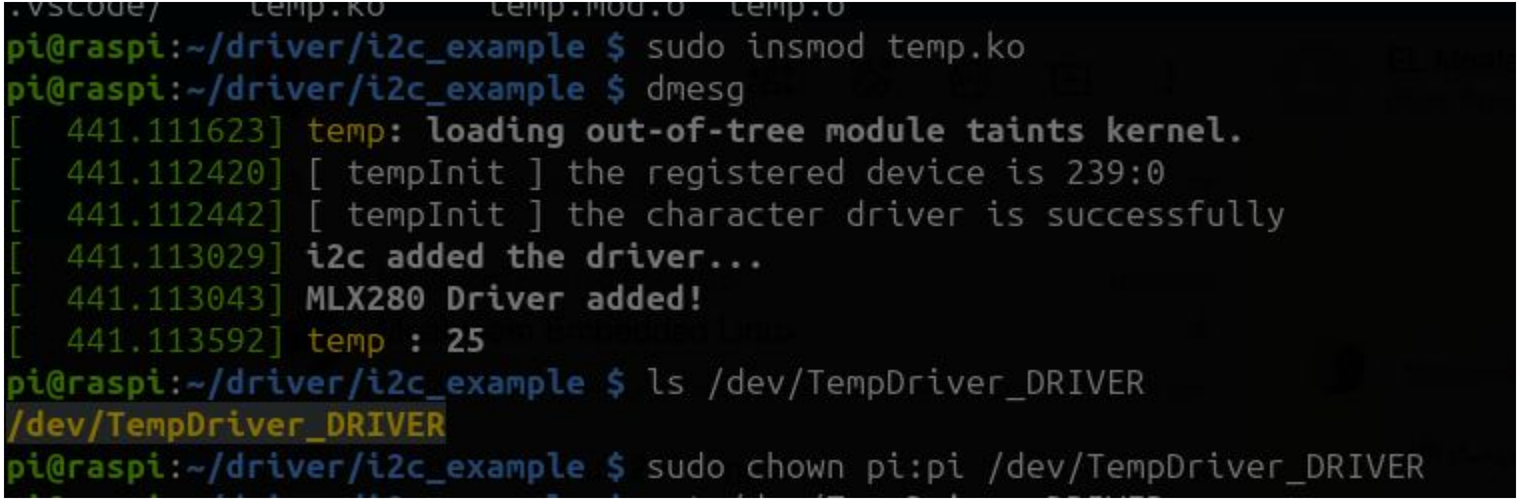
Test

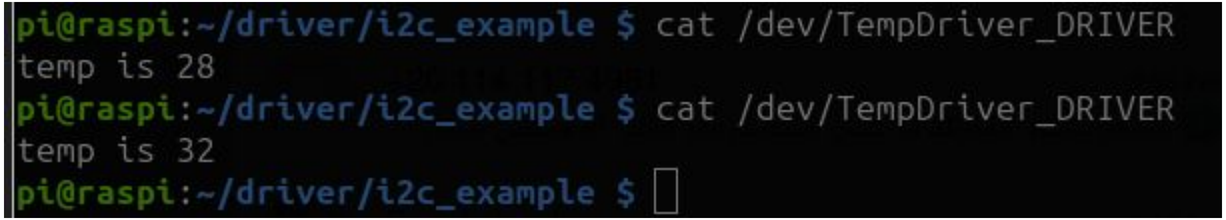
```
pi@raspi:~/driver/ioctl_example $ sudo insmod gpio_driver.ko
(reverse-i-search)`cho': e^Co 1 > /dev/mygpio_number
pi@raspi:~/driver/ioctl_example $ sudo chmod 777 /dev/mygpio_number
pi@raspi:~/driver/ioctl_example $
pi@raspi:~/driver/ioctl_example $
pi@raspi:~/driver/ioctl_example $
pi@raspi:~/driver/ioctl_example $ echo 1 > /dev/mygpio_number
pi@raspi:~/driver/ioctl_example $ echo 0 > /dev/mygpio_number
pi@raspi:~/driver/ioctl_example $ ./a.out
The pin_number is 20
pi@raspi:~/driver/ioctl_example $ echo 1 > /dev/mygpio_number
pi@raspi:~/driver/ioctl_example $ echo 0 > /dev/mygpio_number
pi@raspi:~/driver/ioctl_example $
```

I2C mlx90614



Test


.vscode/ temp.ko temp.mod.o temp.o
pi@raspi:~/driver/i2c_example $ sudo insmod temp.ko
pi@raspi:~/driver/i2c_example $ dmesg
[441.111623] temp: loading out-of-tree module taints kernel.
[441.112420] [tempInit] the registered device is 239:0
[441.112442] [tempInit] the character driver is successfully
[441.113029] i2c added the driver...
[441.113043] MLX280 Driver added!
[441.113592] temp : 25
pi@raspi:~/driver/i2c_example $ ls /dev/TempDriver_DRIVER
/dev/TempDriver_DRIVER
pi@raspi:~/driver/i2c_example $ sudo chown pi:pi /dev/TempDriver_DRIVER


pi@raspi:~/driver/i2c_example $ cat /dev/TempDriver_DRIVER
temp is 28
pi@raspi:~/driver/i2c_example $ cat /dev/TempDriver_DRIVER
temp is 32
pi@raspi:~/driver/i2c_example $

Same as any driver

```c
struct TempDriver
{
    dev_t driverTemp_number;
    struct cdev cdevTemp;
    struct file_operations fops;
    struct class *my_class;

} m_TempDriver = {
    .fops = {
        .owner = THIS_MODULE,
        .open = driver_open,
        .release = driver_close,
        .read = driver_read
    }};
```

```c
static int __init tempInit(void)
{
    s32 id;
    // device number , major , minor ,name
    int retval = alloc_chrdev_region(&m_TempDriver.driverTemp_number, 0, 1, DRIVER_NUMBER);
    if (retval == 0)
    {
        printk(KERN_INFO "[ %s ] the registered device is %d:%d \n", __func__, MAJOR(m_TempDriver.driverTemp_number), MINOR(m_TempDriver.driverTemp_number));
    }
    else
    {
        printk(KERN_ERR "[ %s ] cannot allocate Major or Minor number \n", __func__);
    }

    cdev_init(&m_TempDriver.cdevTemp, &m_TempDriver.fops);

    if (cdev_add(&m_TempDriver.cdevTemp, m_TempDriver.driverTemp_number, 1) < 0)
    {
        printk(KERN_ERR "[ %s ] cannot Register the character driver\n", __func__);
        goto CDEV_ERROR;
    }
    printk(KERN_INFO "[ %s ] the character driver is successfully \n", __func__);
    /* Create device class */
    if ((m_TempDriver.my_class = class_create(DRIVER_CLASS)) == NULL)
    {
        printk(KERN_ERR "[ %s ]  class can not be created!\n", __func__);
        goto ClassError;
    }

    /* create device file */
    if (device_create(m_TempDriver.my_class, NULL, m_TempDriver.driverTemp_number, NULL, DRIVER_NAME) == NULL)
    {
        printk(KERN_ERR "[ %s ] Can not create device file!\n", __func__);
        goto FileError;
    }
    /*********************************************/
```

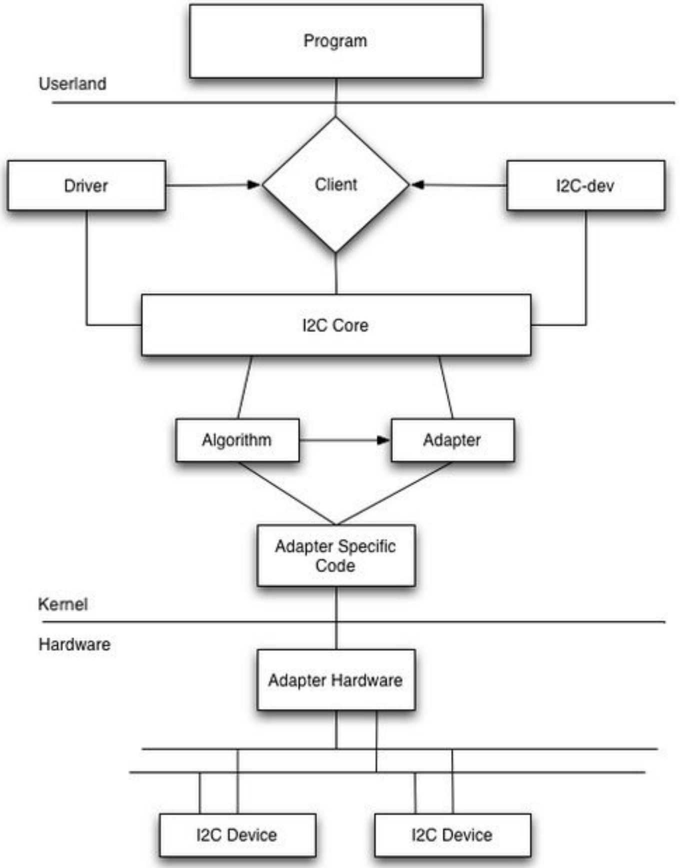
```c
}
module_init(tempInit);
module_exit(tempExit);
```

Driver Architecture

The picture shows the interrelationships of our kernel drivers. **The drivers at the top of the kernel section are "chip" drivers. Chip drivers exist for many chip types: RTC, EEPROM, I/O expander,**

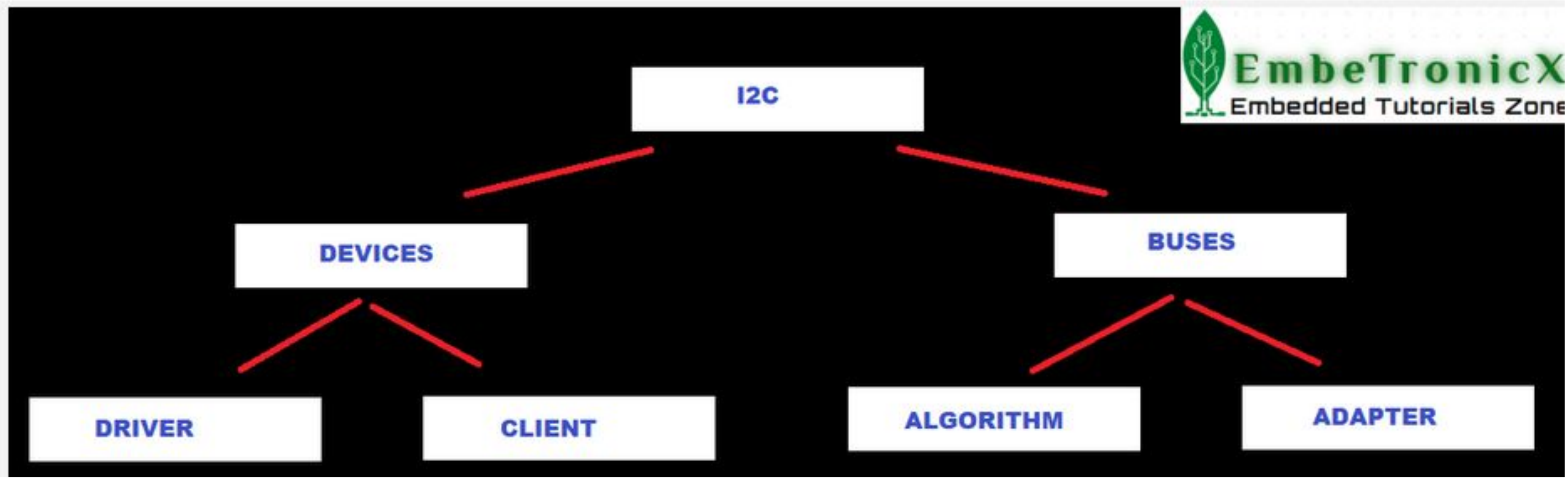
. In the middle is **i2c-core, which contains the I2C and SMBus protocol** implementations**. At the bottom of the kernel section are the algorithm and adapter drivers, which comprise the "bus" drivers for accessing the i2c bus** (algorithm and adapter drivers are generally combined, except for "bit banging" drivers which use a common algorithm).

The "program" section at the top represents all the user-space programs that end up **accessing the chips, either through the /dev interface, using the i2c-dev driver (for example i2cdetect, i2cdump or sensors-detect) o**r through sysfs, using chip-specific drivers (for example libsensors, fancontrol or custom shell scripts).

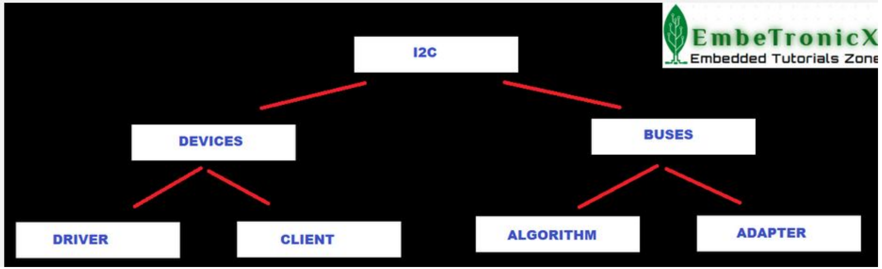


I2C Subsystem

The kernel divided the I2C subsystem by **Buses** and **Devices**. The **Buses** are again divided into **Algorithms** and **Adapters**. The **devices** are again divided into **Drivers** and **Clients.** The below image will give you some understandings.

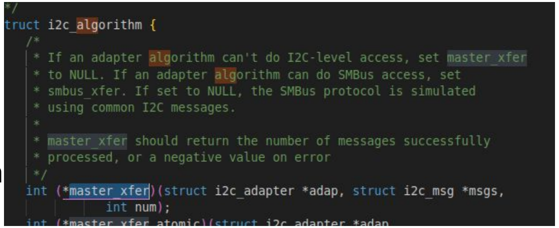


I2C Subsystem



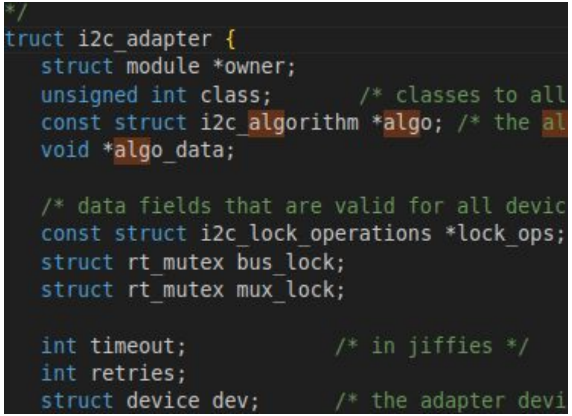
Algorithm

An Algorithm driver contains a **general code** that can be used for a whole class of I2C adapters.



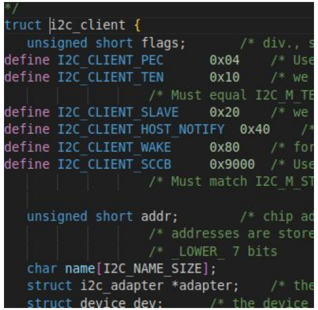
Adapters

An Adapter effectively **represents a bus** – it is used to tie up a particular I2C with an algorithm an specific adapter driver either depends on one algorithm driver or includes its own implementation.



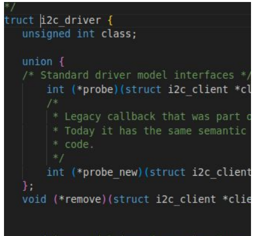
Clients

A Client represents a chip (slave) on the I2C.



Drivers

This is the driver that we are writing for the client.

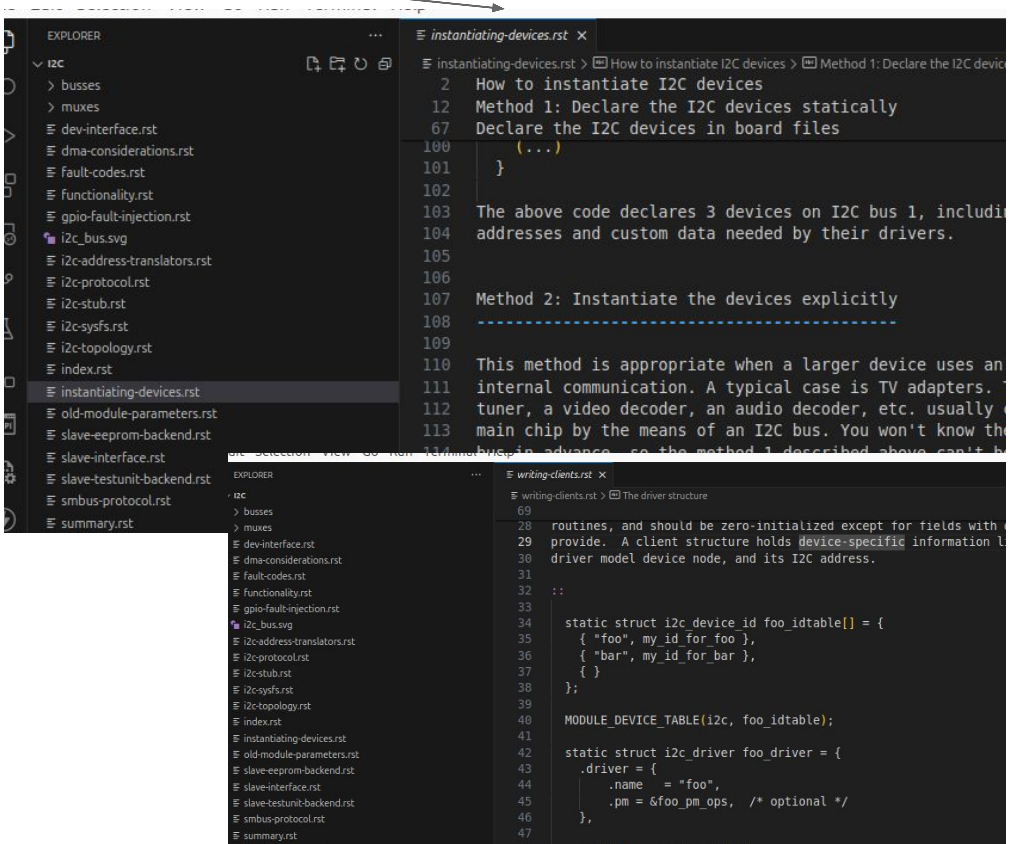


Steps to write the driver

I2C Driver in Linux Kernel

Steps that involve while writing the I2C device driver are given below.

1. Get the I2C adapter.
2. Create the `oled_i2c_board_info` structure and create a device using that.
3. Create the `i2c_device_id` for your slave device and register that.
4. Create the `i2c_driver` structure and add that to the I2C subsystem.
5. Now the driver is ready. So you can transfer the data between master and slave.
6. Once you are done, then remove the device.



```
struct i2c_adapter *i2c_get_adapter(int nr);
```

Where,

nr – I2C bus number. In our case (Raspberry Pi 4), it should be 1.

It returns the `struct i2c_adapter`.

```
/* create device file */
if (device_create(m_TempDriver.my_class, NULL, m_TempDriver.driverTemp_number, NULL, DRIVER_NAME) == NULL)
{
    printk(KERN_ERR "[ %s ] Can not create device file!\n", __func__);
    goto FileError;
}
/*********************************
// get adapter and add the device
/*********************************
MLX_i2c_adapter = i2c_get_adapter(I2C_BUS_AVAILABLE);

if (MLX_i2c_adapter != NULL)
{
```

```
#define I2C_BUS_AVAILABLE 1
```

The I2C Bus available on the raspberry
Expands to:

1

```
struct i2c_adapter *i2c_get_adapter(int nr)
{
	struct i2c_adapter *adapter;

	mutex_lock(&core_lock);
	adapter = idr_find(&i2c_adapter_idr, nr);
	if (!adapter)
		goto exit;

	if (try_module_get(adapter->owner))
		get_device(&adapter->dev);
	else
		adapter = NULL;

 exit:
	mutex_unlock(&core_lock);
	return adapter;
}
EXPORT_SYMBOL(i2c_get_adapter);
```

Create the board info

Once you get the adapter structure, then create the board info and using that board info, create the device.

Create Board info

Just create the `i2c_board_info` structure and assign required members of that.

```
struct i2c_board_info {
  char type[I2C_NAME_SIZE];
  unsigned short flags;
  unsigned short addr;
  void * platform_data;
  struct dev_archdata * archdata;
  struct device_node * of_node;
  struct fwnode_handle * fwnode;
  int irq;
};
```

```
/* Defines for device identification
#define I2C_BUS_AVAILABLE 1
#define SLAVE_DEVICE_NAME "MLX90614"
#define MLX280_SLAVE_ADDRESS 0x5a
```

```
static struct i2c_board_info MLX_i2c_board_info = {
    I2C_BOARD_INFO(SLAVE_DEVICE_NAME, MLX280_SLAVE_ADDRESS)};
```

```
  are provided using conventional syntax.
 */
#define I2C_BOARD_INFO(dev_type, dev_addr) \
    .type = dev_type, .addr = (dev_addr)
```

Create Device

Now board info structure is ready. Let's instantiate the device from that I2C bus.

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

where,

`*adap` – Adapter structure that we got from `i2c_get_adapter()`

`*info` – Board info structure that we have created

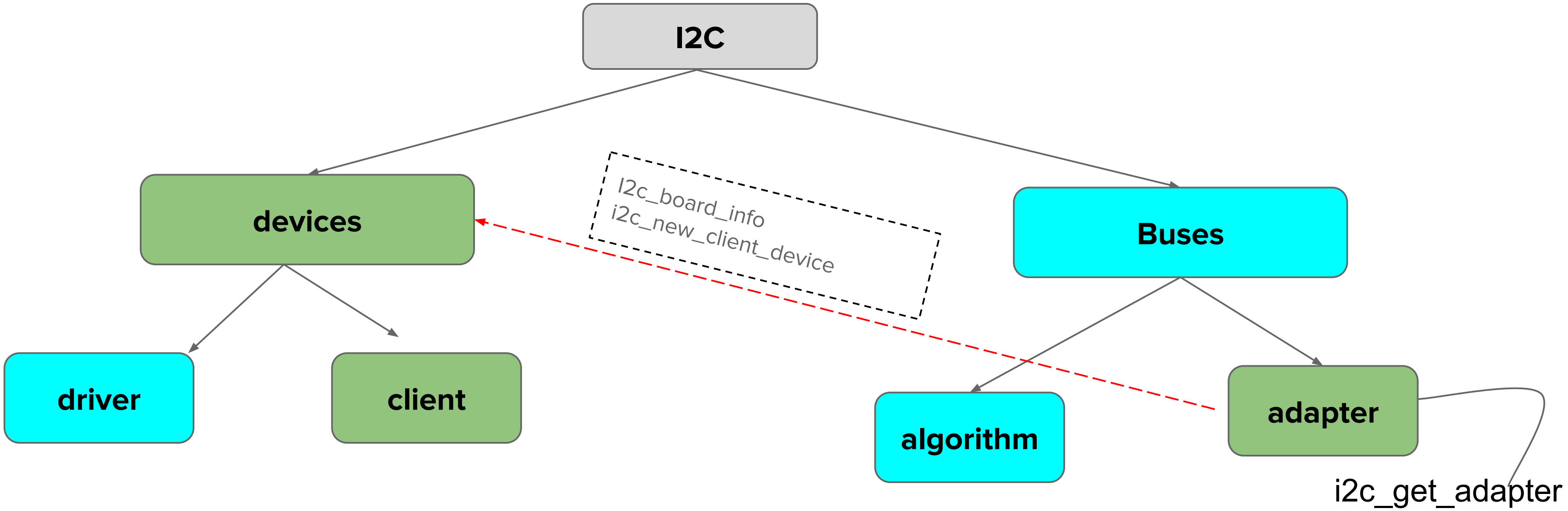
This will return the `i2c_client` structure. We can use this client for our future transfers.

Note: If you are using the newer kernel (**5.2 =<**), then you must use the `i2c_new_client_device` API instead of `i2c_new_device`.

Now we will see the example for this section. So that you will get some idea that how we are using this in our code.

```
/******************************************/
// get adapter and add the device
/******************************************/
MLX_i2c_adapter = i2c_get_adapter(I2C_BUS_AVAILABLE);

if (MLX_i2c_adapter != NULL)
{
    MLX280_i2c_client = i2c_new_client_device(MLX_i2c_adapter, &MLX_i2c_board_info);
    if (MLX280_i2c_client != NULL)
    {
```



1. Get the I2C adapter.
2. Create the `oled_i2c_board_info` structure and create a device using that.

```
/*******************************************/
// get adapter and add the device
/*******************************************/
MLX_i2c_adapter = i2c_get_adapter(I2C_BUS_AVAILABLE);

if (MLX_i2c_adapter != NULL)
{
    MLX280_i2c_client = i2c_new_client_device(MLX_i2c_adapter, &MLX_i2c_board_info);
    if (MLX280_i2c_client != NULL)
    {
```

3. Create the `i2c_device_id` for your slave device and register that.

Create the device id

Just create the structure `i2c_device_id` and initialize the necessary members.

```
struct i2c_device_id {
  char name[I2C_NAME_SIZE];
  kernel_ulong_t driver_data;
};
```

where,

`name` – Slave name

`driver_data` – Data private to the driver (This data will be passed to the respective driver)

After this, call `MODULE_DEVICE_TABLE(i2c, my_id_table)` in order to expose the driver along with its I2C device table IDs to userspace.

```
static const struct i2c_device_id MLX_id[] = {
    {SLAVE_DEVICE_NAME, 0},
    {}};
MODULE_DEVICE_TABLE(i2c, MLX_id);
```

```
#define MODULE_DEVICE_TABLE(type, name)                 \
extern typeof(name) __mod_##type##__##name##_device_table       \
  __attribute__ ((unused, alias(__stringify(name))))
#else  /* !MODULE */
```

4. Create the **i2c_driver** structure and add that to the I2C subsystem.

Create the i2c_driver

```
struct i2c_driver {
  unsigned int class;
  int (* attach_adapter) (struct i2c_adapter *);
  int (* probe) (struct i2c_client *, const struct i2c_device_id *);
  int (* remove) (struct i2c_client *);
  void (* shutdown) (struct i2c_client *);
  void (* alert) (struct i2c_client *, unsigned int data);
  int (* command) (struct i2c_client *client, unsigned int cmd, void *arg);
  struct device_driver driver;
  const struct i2c_device_id * id_table;
  int (* detect) (struct i2c_client *, struct i2c_board_info *);
  const unsigned short * address_list;
  struct list_head clients;
};
```

Where,

```
}
static struct i2c_driver MLX_driver = {
    .driver = {
        .name = SLAVE_DEVICE_NAME,
        .owner = THIS_MODULE,
        },
        .probe=mlx_probe,
        .remove=mlx_remove,
         .id_table =MLX_id,
        };
```

Add the I2C driver to the I2C subsystem

Now we have the i2c_driver structure. So we can add this structure to the I2C subsystem using the below API.

```
i2c_add_driver(struct i2c_driver *i2c_drive);
```

Where,

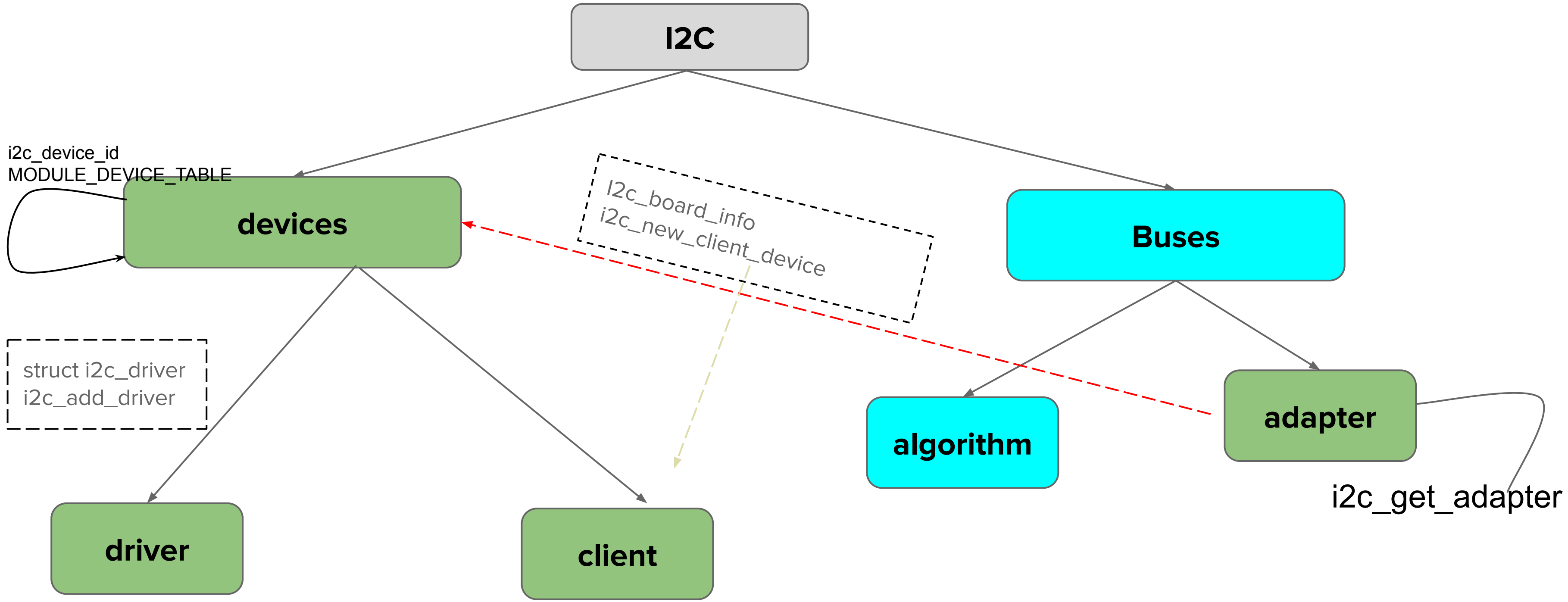
`i2c_drive` – The i2c_driver structure that we have created.

During the call to `i2c_add_driver` to register the I2C driver, all the I2C devices will be traversed. Once matched, the `probe` function of the driver will be executed.

You can remove the driver using `i2c_del_driver(struct i2c_driver *i2c_drive)`.

```
MLX_i2c_adapter = i2c_get_adapter(I2C_BUS_AVAILABLE);

if (MLX_i2c_adapter != NULL)
{
    MLX280_i2c_client = i2c_new_client_device(MLX_i2c_adapter, &MLX_i2c_board_info);
    if (MLX280_i2c_client != NULL)
    {
        if (i2c_add_driver(&MLX_driver) != -1)
        {
            printk("i2c added the driver...\n");
        }
        else
            printk("Can't add driver...\n");
    }
```




```
MLX_i2c_adapter = i2c_get_adapter(I2C_BUS_AVAILABLE);

if (MLX_i2c_adapter != NULL)
{
    MLX280_i2c_client = i2c_new_client_device(MLX_i2c_adapter, &MLX_i2c_board_info);
    if (MLX280_i2c_client != NULL)
    {
        if (i2c_add_driver(&MLX_driver) != -1)
        {
            printk("i2c added the driver...\n");
        }
        else
            printk("Can't add driver...\n");
    }
    i2c_put_adapter(MLX_i2c_adapter);
}
```

1. Get the I2C adapter.
2. Create the `oled_i2c_board_info` structure and create a device using that.
3. Create the `i2c_device_id` for your slave device and register that.
4. Create the `i2c_driver` structure and add that to the I2C subsystem.

5. Now the driver is ready. So you can transfer the data between master and slave.

```
    }
    printk("MLX280 Driver added!\n");
    id = i2c_smbus_read_word_data(MLX280_i2c_client, MLX90614_TOBJ1);
    printk("temp : %d\n", (id / 50) - 273);
```

Send data

i2c_master_send

This API issue a single I2C message in the master transmit mode.

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

Where,

`client` – Handle to the slave device

`buf` – Data that will be written to the slave

`count` – How many bytes to write, must be less than 64k since msg length is u16

It returns negative `errno`, or else the number of bytes written.

i2c_smbus_write_byte

This API is used to send one byte to the slave.

```
s32 i2c_smbus_write_byte ( const struct i2c_client * client, u8 value);
```

Where,

`client` – Handle to the slave device

`value` – Byte to be sent

It returning negative `errno` else zero on success.

i2c_smbus_write_byte_data

```
s32 i2c_smbus_write_byte_data ( const struct i2c_client * client, u8 command, u8 value);
```

Where,

`client` – Handle to the slave device

`command` – Byte interpreted by slave

Read data

i2c_master_recv

This API issue a single I2C message in master receive mode.

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

Where,

`client` – Handle to the slave device

`buf` – Data that will be read from the slave

`count` – How many bytes to read, must be less than 64k since msg length is u16

It returns negative `errno`, or else the number of bytes reads.

i2c_smbus_read_byte

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

Where,

`client` – Handle to the slave device

It is returning negative `errno` else the byte received from the device.

i2c_smbus_read_byte_data

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

Where,

`client` – Handle to the slave device

`command` – Byte interpreted by slave

It is returning negative `errno` else a data byte received from the device.

MLX90614

7.3.4 RAM

It is not possible to write into the RAM memory. It can only be read and only a limited numb
registers are of interest to the customer.

```c
ssize_t driver_read(struct file *file, char __user *user_buffer, size_t count, loff_t *offs)
{
    int not_copied;
    char out_string[20] = "";
    int tempvalue = 0;
    /* Get amount of data to copy */
    printk("%s: the count to read %ld \n", __func__, count);
    printk("%s: the offs %lld \n", __func__, *offs);
    if (count + *offs > 20)
    {
        count = 20 - *offs;
    }
    /*i2c */
    tempvalue = i2c_smbus_read_word_data(MLX280_i2c_client, MLX90614_TOBJ1);
    tempvalue = (tempvalue / 50) - 273;
    printk("temp : %d\n", tempvalue);
    snprintf(out_string, sizeof(out_string), "temp is %d\n", tempvalue);
    /***/
    not_copied = copy_to_user(user_buffer, &out_string[*offs], count);
    if (not_copied)
    {
        return -1;
    }
    *offs = count;
    printk("%s: not copied %d \n", __func__, not_copied);
    printk("%s: message: %s \n", __func__, user_buffer);
    return count;
}
```

```
#define MLX90614_TOBJ1 0x07
addresses
Expands to:
0x07
```

RAM (32x17)		
Name	**Address**	**Read access**
Melexis reserved	000h	Yes
...	...	...
Melexis reserved	005h	Yes
T_A	006h	Yes
T_{OBJ1}	007h	Yes
T_{OBJ2}	008h	Yes
Melexis reserved	009h	Yes
...	...	...
Melexis reserved	01Fh	Yes

```
pi@raspi:~/driver/sysfs_attr $ sudo echo 1 >  /sys/kernel/etx_sysfs/etx_value
pi@raspi:~/driver/sysfs_attr $ sudo cat /sys/kernel/etx_sysfs/etx_value
1pi@raspi:~/driver/sysfs_attr $
```

kobj_attribute

https://embetronicx.com/tutorials/linux/device-drivers/sysfs-in-linux-kernel/

```c
volatile int etx_value = 0;

struct kobject *kobj_ref;
```

```c
/*************** Sysfs functions **********************/
static ssize_t  sysfs_show(struct kobject *kobj,
                        struct kobj_attribute *attr, char *buf);
static ssize_t  sysfs_store(struct kobject *kobj,
                        struct kobj_attribute *attr,const char *buf, size_t count);

struct kobj_attribute etx_attr = __ATTR(etx_value, 0660, sysfs_show, sysfs_store);
```

```c
/*
** This function will be called when we read the sysfs file
*/
static ssize_t sysfs_show(struct kobject *kobj,
                struct kobj_attribute *attr, char *buf)
{
        pr_info("Sysfs - Read!!!\n");
        return sprintf(buf, "%d", etx_value);
}

/*
** This function will be called when we write the sysfsfs file
*/
static ssize_t sysfs_store(struct kobject *kobj,
                struct kobj_attribute *attr,const char *buf, size_t count)
{
        pr_info("Sysfs - Write!!!\n");
        sscanf(buf,"%d",&etx_value);
        return count;
}
```

```c
        /*Creating a directory in /sys/kernel/ */
        kobj_ref = kobject_create_and_add("etx_sysfs",kernel_kobj);

        /*Creating sysfs file for etx_value*/
        if(sysfs_create_file(kobj_ref,&etx_attr.attr)){
                pr_err("Cannot create sysfs file......\n");
                goto r_sysfs;
    }
        pr_info("Device Driver Insert...Done!!!\n");
        return 0;

r_sysfs:
        kobject_put(kobj_ref);
        sysfs_remove_file(kernel_kobj, &etx_attr.attr);
```

Exit

```c
*/
static void __exit etx_driver_exit(void)
{
        kobject_put(kobj_ref);
        sysfs_remove_file(kernel_kobj, &etx_attr.attr);
        device_destroy(dev_class,dev);
        class_destroy(dev_class);
        cdev_del(&etx_cdev);
        unregister_chrdev_region(dev, 1);
        pr_info("Device Driver Remove...Done!!!\n");
}

module_init(etx_driver_init);
```

Tasks

1- write i2c driver for any i2c device
2- write spi driver
3- add attributes to driver
4-write kernel thread