I2C driver

Thursday, August 25, 2016

7:44 AM

Start with Free Electrons slides

<http://venkateshabbarapu.blogspot.in/2012/11/i2c-driver-in-linux.html>

<http://opensourceforu.com/2015/01/writing-i2c-clients-in-linux/>

<http://invo-tronics.com/i2c-driver-for-linux-based-embedded-system/> (Good Explaination)

<https://i2c.wiki.kernel.org/index.php/Driver_Architecture>

<http://www.embeddedlinux.org.cn/essentiallinuxdevicedrivers/final/ch08lev1sec2.html>

<http://rts.lab.asu.edu/web_438/CSE438_598_slides_yhlee/438_6_Linux_I2C_SMBus.pdf> (Nice Explaination)

<http://www.ti.com/lit/an/sbaa170/sbaa170.pdf>

<https://i2c.wiki.kernel.org/index.php/File:I2c-layers.png>

<https://i2c.wiki.kernel.org/index.php/Driver_Architecture>

From Free electrons (kernel-slides)

▶ Like all bus subsystems, the I2C subsystem is responsible for:

▶ Providing an API to implement I2C controller drivers

▶ Providing an API to implement I2C device drivers, in kernel

space

▶ Providing an API to implement I2C device drivers, in user

space

▶ The core of the I2C subsystem is located in drivers/i2c.

▶ The I2C controller drivers are located in drivers/i2c/busses.

▶ The I2C device drivers are located throughout drivers/,

depending on the type of device (ex: drivers/input for input

devices).

▶ Like all bus subsystems, the I2C subsystem defines a

struct i2c\_driver that inherits from

struct device\_driver, and which must be instantiated and

registered by each I2C device driver.

▶ As usual, this structure points to the ->probe() and

->remove() functions.

▶ It also contains an id\_table field that must point to a list of

*device IDs* (which is a list of tuples containing a string and

some private driver data). It is used for non-DT based probing

of I2C devices.

▶ The i2c\_add\_driver() and i2c\_del\_driver() functions are

used to register/unregister the driver.

▶ If the driver doesn't do anything else in its init()/exit()

functions, it is advised to use the module\_i2c\_driver()

macro instead.

. .

static const struct i2c\_device\_id <driver>\_id[] = {

{ "<device-name>", 0 },

{ }

};

MODULE\_DEVICE\_TABLE(i2c, <driver>\_id);

#ifdef CONFIG\_OF

static const struct of\_device\_id <driver>\_dt\_ids[] = {

{ .compatible = "<vendor>,<device-name>", },

{ }

};

MODULE\_DEVICE\_TABLE(of, <driver>\_dt\_ids);

#endif

static struct i2c\_driver <driver>\_driver = {

.probe = <driver>\_probe,

.remove = <driver>\_remove,

.id\_table = <driver>\_id,

.driver = {

.name = "<driver-name>",

.owner = THIS\_MODULE,

.of\_match\_table = of\_match\_ptr(<driver>\_dt\_ids),

},

};

module\_i2c\_driver(<driver>\_driver);

▶ On non-DT platforms, the struct i2c\_board\_info structure

allows to describe how an I2C device is connected to a board.

▶ Such structures are normally defined with the

I2C\_BOARD\_INFO() helper macro.

▶ Takes as argument the device name and the slave address of

the device on the bus.

▶ An array of such structures is registed on a per-bus basis using

i2c\_register\_board\_info(), when the platform is

initialized.

. .

static struct i2c\_board\_info <board>\_i2c\_devices[] \_\_initdata = {

{

I2C\_BOARD\_INFO("cs42l51", 0x4a),

},

};

void board\_init(void)

{

/\*

\* Here should be the registration of all devices, including

\* the I2C controller device.

\*/

i2c\_register\_board\_info(0, <board>\_i2c\_devices,

ARRAY\_SIZE(<board>\_i2c\_devices));

/\* More devices registered here \*/

}

In order to get the better understanding of the i2c subsystem, I went through the am335x0-evm board in the kernel linux-am33x. There is a board dependent file called board-am335xevm.c. Following are the I2C devices registered for the board:

Static struct i2c\_boardinfo \_\_initdata am335x\_i2c0\_boardinfo[] = {

{I2C\_BOARD\_INFO("24C256", 0),

.platform\_data = &am335x\_daughter\_board\_eeprom\_info,},

}

Having understood the current framework in linux-2.6.31 as per the above documents, the next intention was to understand the older i2c framework in Linux kernel 2.6.20. As a part of that, tried exploring the at24.c driver and want to understand the connections with board set up file. The nice explanation is available at kernel documentation itself.

As a part of driver registration, we have attach\_adpater and detach\_adapter as below:

Static struct i2c\_driver at24\_driver = {

.driver = {

.name = "at24",

.owner = THIS\_MODULE,

}

.attach\_adapter = at24\_attach\_adapter,

.detach\_client = at24\_detach\_client,

};

Single driver structure will be implemented and all the clients will be instantiated from it. Driver structure contains general access routines, a client structure specific information like the actual i2C address.

Static int \_\_init at24\_init(void)

{

Return i2c\_add\_driver(&at24\_driver);

}

This registers a driver with i2c core.

Static int attach\_adapter(struct i2c\_adapter \*adapter)

{

Return i2c\_probe(adapter, &addr\_data, at24\_detect);

}

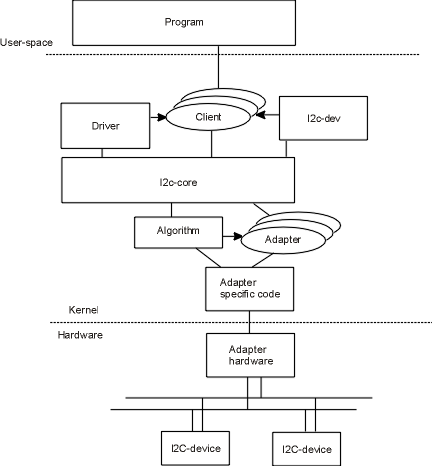
Whenever a new adapter is inserted or for all adapters if the driver is being registered, the callback attach\_adapter is called.

Now is the  
 [205](http://lxr.linux.no/linux+v2.6.20/Documentation/i2c/writing-clients#L205)time to determine what devices are present on the adapter, and to register  
 [206](http://lxr.linux.no/linux+v2.6.20/Documentation/i2c/writing-clients#L206)a client for each of them.  
 [207](http://lxr.linux.no/linux+v2.6.20/Documentation/i2c/writing-clients#L207)  
 [208](http://lxr.linux.no/linux+v2.6.20/Documentation/i2c/writing-clients#L208)The attach\_adapter callback is really easy: we just call the generic  
 [209](http://lxr.linux.no/linux+v2.6.20/Documentation/i2c/writing-clients#L209)detection function. This function will scan the bus for us, using the  
 [210](http://lxr.linux.no/linux+v2.6.20/Documentation/i2c/writing-clients#L210)information as defined in the lists explained above. If a device is  
 [211](http://lxr.linux.no/linux+v2.6.20/Documentation/i2c/writing-clients#L211)detected at a specific address, another callback is called.  
 [212](http://lxr.linux.no/linux+v2.6.20/Documentation/i2c/writing-clients#L212)  
 [213](http://lxr.linux.no/linux+v2.6.20/Documentation/i2c/writing-clients#L213) int foo\_attach\_adapter(struct i2c\_adapter \*adapter)  
 [214](http://lxr.linux.no/linux+v2.6.20/Documentation/i2c/writing-clients#L214) {  
 [215](http://lxr.linux.no/linux+v2.6.20/Documentation/i2c/writing-clients#L215) return i2c\_probe(adapter,&addr\_data,&foo\_detect\_client);  
 [216](http://lxr.linux.no/linux+v2.6.20/Documentation/i2c/writing-clients#L216) }  
 [217](http://lxr.linux.no/linux+v2.6.20/Documentation/i2c/writing-clients#L217)  
 [218](http://lxr.linux.no/linux+v2.6.20/Documentation/i2c/writing-clients#L218)Remember, structure `addr\_data' is defined by the macros explained above

so you do not have to define it yourself.

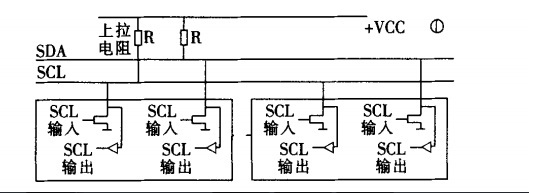
The i2c\_probe function will call the foo\_detect\_client  
 [222](http://lxr.linux.no/linux+v2.6.20/Documentation/i2c/writing-clients#L222)function only for those i2c addresses that actually have a device on  
 [223](http://lxr.linux.no/linux+v2.6.20/Documentation/i2c/writing-clients#L223)them (unless a `force' parameter was used). In addition, addresses that  
 [224](http://lxr.linux.no/linux+v2.6.20/Documentation/i2c/writing-clients#L224)are already in use (by some other registered client) are skipped.

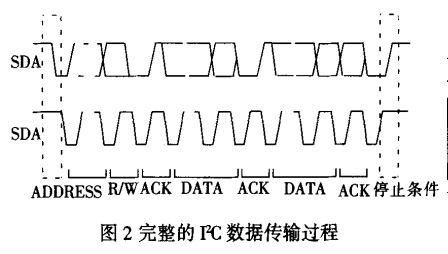
From <<http://lxr.linux.no/linux+v2.6.20/Documentation/i2c/writing-clients>>

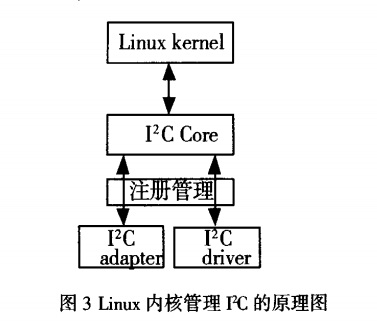


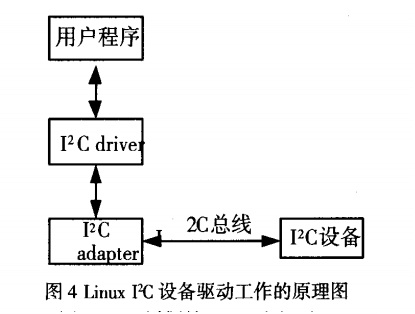
Abstract I2C bus is a very common bus, with a simple and efficient characteristics, widely used in a variety of consumer electronic products and audio and video equipment in the development of embedded systems are often used. This paper analyzes the structure of an embedded linux system I2C driver, combined with a specific I2C clock chip DS1307, I2C device driver development described in the general process of embedded linux system.

[Keywords] I2C bus embedded linux driver development **1, I2C Bus Introduction** I2C (Inter-Integrated Circuit) bus is a PHILIPS developed by the two-wire serial bus for connecting the microcontroller and its peripherals. I2C bus main advantage is simplicity and effectiveness. **1.1 I2C bus works**I2C bus is a serial bus data line SDA and clock SCL constituted by various control devices are connected in parallel on this bus, each device has a unique address recognition, as the bus a device or the receiving device (as determined by the function of the device) [1] sent. I2C bus interface circuit configuration shown in Figure 1. Figure 1 I2C bus interface circuit [1] **1.2 signal state of the I2C several bus [1]** 1. Idle status: SDA and SCL are high. 2. Start condition (S): SCL is high, SDA from high to low transition, start transferring data. 3. The end condition (P): SCL is low, SDA from low to high transition, the end of the data transfer. 4. Data valid: during the high period of SCL, SDA remains stable, the data is valid. SDA changes can only occur at the end of the period the level of SCL. 5. ACK signal: during a data transfer, the receiving device receives each byte of data to produce an ACK signal, issue a specific low level pulse to the transmitting device, it said it had received data. **Basic Operations 1.3 I2C bus** I2C bus must (typically a microcontroller) controlled by the master, the master generates the serial clock (SCL), while controlling the transmission direction of the bus, and generates start and stop conditions. Data transmission, the first master generates a start condition followed by the control byte of the device (the front seven is the slave device address, the last one to write bits). Next is the data read and write operations, and ACK response signal. At the end of the data transfer, the master generates a stop condition [1]. A specific process shown in Figure 2. Figure 2 complete I2C data transmission process [1] **2. I2C driver under Linux Analysis 2.1 Linux system I2C driver hierarchy** Linux system I2C device has good support, Linux system under I2C driver can logically divided into three parts: 1. I2C bus driver I2C core: to achieve the I2C bus, I2C adapter and I2C driver management. 2. I2C controller driver I2C adapter: for different types of I2C controller, to achieve specific method for I2C bus access. Driver I2C driver 3. I2C devices: I2C for a particular device, to achieve specific functions, including the interface to read, write and ioctl and other operations on the user level. These three parts of the hierarchy shown in Figure 3 and Figure 4.









**Core 2.2 I2C bus driver I2C**

I2C Core Linux kernel is the core part for maintenance and management of I2C, which maintains two static List, were recorded in the system I2C driver structure and I2C adapter structure. I2C core provides interface functions that allow a I2C adatper, I2C driver and I2C client initialization registered in the I2C core, and log off when you exit. It also provides a generic interface I2C bus read and write access to (specific implementation implemented in I2C adapter associated with the I2C controller's), mainly used in I2C device driver. **2.3 I2C Controller Adapter driver I2C** I2C Adapter for different types of I2C controller hardware, to achieve specific method for I2C bus access compare the underlying. I2C adapter construct one pair I2C core layer of the interface data structures, and the I2C core registered a controller through an interface function. I2C adapter main achievement of the algorithm I2C bus access, iic\_xfer () function is the I2C adapter to the bottom of the I2C bus to read and write methods of implementation. Meanwhile I2C adpter also realize the I2C controller interrupt handler. **Driver I2C driver 2.4 I2C device** I2C driver provides a driver a generic I2C device access interface to achieve the character type devices, particularly access to the device through the I2C adapter to achieve. I2C driver construct one pair I2C core layer of the interface data structure registered an I2C device driver interface functions through the I2C Core.Meanwhile I2C driver construct a user interface layer, data structures, and through the interface to the kernel function is registered as a major number of 89 character-type devices. I2C driver to achieve user-level access I2C devices, including open, read, write, ioctl, release and other conventional file operations, we can open the I2C device files open function, by setting ioctl functions to be accessed from the address of the device, and then I2C devices can be completed on the read and write operations through read and write functions. General procedure provided by the I2C driver can access any I2C device, but which achieve the read, write and ioctl functions to achieve is entirely based on the general equipment, all operations are based on the data byte stream, there is no clear format and significance. To make it easier and efficient use of I2C devices, we can develop a specific I2C devices specific I2C device drivers, complete interpretation of specific data formats, and achieve some special features in the drive. **3. A specific development I2C device driver** DS1307 I2C interface is a compact real-time clock chip with low power consumption, full BCD code output clock and calendar, 12/24-hour mode, minutes and seconds, weeks, years month day timing data, automatic leap year compensation valid until 2100, plus 56 Bytes of NV RAM (nonvolatile RAM), etc. [3]. The following DS1307 example to illustrate a specific design elements I2C device driver. **The general structure of 3.1 I2C device driver** a specific I2C device drivers need to implement two interfaces, one for I2C core layer interface to mount I2C adapter layer to achieve the I2C bus and I2C device specific access method including to achieve attach\_adapter, detach\_client, command and other interface functions. Another is the user interface to the application layer, providing user access to program I2C interface of the device, including the realization of interface functions open, release, read, write and, most importantly ioctl other standard file operations. Specific functional interface function I2C core layer is explained as follows: attach\_adapter: I2C Driver when calling I2C\_add\_driver () register, the discovery of each I2C adapter (which corresponds to an I2C bus) must call this function to check compliance with the I2C adapter I2C driver specific conditions, if the conditions are in line with this I2C adapter is connected, via I2C adapter to achieve access to the I2C bus and I2C devices. detach\_client: I2C driver called in to remove a I2C device when the function clear description of the I2C device data structures, so that later you can not access the device up. command: the characteristics of the equipment, implement a series of sub-functions, underlying user interface ioctl functions to achieve. **3.2 DS1307 drivers realize I2C core layer interface** in the driver must implement a struct **I2C** \_driver data structure, and register to drive a I2C I2C core when the drive module initialization, I2C adapter and complete the related operations. struct **I2C** \_driver ds1307\_driver = { name: "the DS1307", the above mentioned id: I2C\_DRIVERID\_DS1307, the flags: I2C\_DF\_NOTIFY, attach\_adapter: ds1307\_probe, detach\_client: ds1307\_detach, Command: ds1307\_command }; data structure ds1307\_driver the name: "DS1307", Id: I2C\_DRIVERID\_DS1307 used to identify DS1307 drivers. flags: I2C\_DF\_NOTIFY said in an I2C bus drivers notice the changes. ds1307\_probe correspond **I2C** \_driver data structure attach\_adapter, the main function: Call I2C core layer provides **I2C** \_probe function to find an I2C bus to see if there is the presence of the device the DS1307, DS1307, if present, then the corresponding I2C adapter and DS1307 devices are attached together, and through the I2C adapter to achieve access to the DS1307. Enabling the DS1307, and calls **I2C** \_attach\_client () to register with the I2C core layer DS1307. ds1307\_detach correspond **I2C** \_driver data structure detach\_client, the main function: Call **I2C** \_detach\_client () to the I2C core layer cancellation DS1307, does not enable the DS1307, so I2C driver can not access the DS1307. ds1307\_command correspond **I2C** \_driver data structure of command, the main function: for DS1307 clock chip features, such as the realization of a series of DS1307\_GETTIME, DS1307\_SETTIME, DS1307\_GETDATETIME, DS1307\_MEM\_READ, DS1307\_MEM\_WRITE other sub-functions, underlying user interface ioctl functions to achieve. More than three interface functions make the DS1307 driver implements the I2C bus and I2C adpater mount, so you can share a common interface I2C core provides I2C bus read and write access to the driver development implement DS1037 user application layer interface function. **3.3 DS1307 driver implements the application layer user interfaces** in the driver must implement a struct file\_operations data structure to the kernel is registered as a character type device (a separate major number is identified), or registered as a device miscdevice (All miscdevice equipment together a major number of different minor number, all miscdevice apparatus for forming a linked list, for finding the corresponding miscdevice device when the device is accessed according to the minor number, then call the application layer interface for which struct file\_operations registered conduct operating). file\_operations rtc\_fops = struct { owner: THIS\_MODULE, the ioctl: ds1307\_rtc\_ioctl, the Read: ds1307\_rtc\_read, the Write: ds1307\_rtc\_read, open: ds1307\_rtc\_open, Release: ds1307\_rtc\_release }; data structure rtc\_fops in ds1307\_rtc\_open and ds1307\_rtc\_release correspond file\_operations the open and release, respectively, to open Close and DS1307. ds1307\_rtc\_ioctl correspond file\_operations the ioctl, a series of control clock chip specific commands provided by the user: RTC\_GET\_TIME: a fixed data format to read the real-time clock. RTC\_SET\_TIME: a fixed set of data format real time clock. RTC\_SYNC\_TIME: real-time clock and system clock between the time synchronization. ds1307\_rtc\_read correspond correspondence file\_operations the read, and ds1307\_rtc\_ioctl achieve sub-functions RTC\_GET\_TIME same functionality, as well as read data from the NV RAM. ds1307\_rtc\_write correspond file\_operations the write, implement and ds1307\_rtc\_ioctl sub-functions RTC\_SET\_TIME the same function, and write data NV RAM. **3.4 DS1307 loading and testing the driver** initialization function ds1307\_init DS1307 drive module (), first through **i2c** register the device driver a I2C of \_add\_driver (& ds1307\_driver) to the I2C core layer, and then by misc\_register (& ds1307\_rtc\_miscdev) The DS1307 is registered as a miscdevice device, so that the user program can access the DS1307 through the major number 10 minor numbers device node 135 of / dev / rtc. The DS1307 drivers compiled into a modular approach, through insmod command to load into the kernel, and then use the test code to test all the features implemented DS1307 drivers have reached the desired results.Since the DS1307 drivers underlying implementation for DS1307 clock chip data interpretation and conversion, so the user program obtained is a fixed format and meaning of data, thereby facilitating access to the user program, improve the efficiency of application development. **4. Summary** I2C bus is a compact, simple bus protocol, application is very extensive, easy access to them. linux system I2C driver has a clear hierarchy, you can easily to a specific I2C device driver development. Based on the linux system I2C driver, as well as a detailed analysis of clock chip DS1307 drive structure, basically you can clearly see a I2C device driver development process. The key is divided into two partial implementation, 1. To

I2C core interface must implement struct **I2C** \_drvier data structure several specific performance function. These functions are the basis for communication between I2C driver with I2C bus physical layer (I2C controller) and I2C devices devices. 2. The application layer user interfaces, data structures must implement struct file\_operation some specific features functions such as open, release, read, write, lseek and other functions. The above two types of interfaces for I2C core interface is I2C access to the infrastructure, to achieve the I2C bus specific access methods; the user interface of the application layer is to facilitate application development, to achieve device-specific functions essential section. **References:** [1] Philips Corporation, the I2C Bus Specification Version 2.1,2000 [2] the Linux Kernel, Version 2.4.30 .. [3] the Integrated Products's Maxim, the DS1307 Datasheet INC.'s USA, 2004 [4] Aless and with Robin, Wei Sun translated, "LINUX device drivers (second Edition)", Beijing, China Electric power Press, 2004

From <<http://blog.csdn.net/gl1987807/article/details/8744877>>

Linux I2C driver is a driver for embedded Linux driver developers often need to write, because those who use the system to I2C devices, almost all need to prepare the corresponding I2C driver to configure and control it, such as RTC real time clock chip, audio and video capture chip, audio and video output chip, EEROM chip, AD / DA converter chips and so on.

[**Linux**](http://lib.csdn.net/base/linux) I2C driver or with a lot of knowledge relates mainly divided into Linux I2C bus driver (I2C BUS Driver) and device drivers (I2C Clients Driver), this article focuses on how to quickly accomplish a specific I2C device drivers (I2C Clients Driver). Overall on Linux I2C-driven [**architecture**](http://lib.csdn.net/base/architecture) , the core principles, etc. You can search other related articles online learning.

In this paper, the main reference document ./Documentation/i2c/writing-clients Linux kernel source directory. In a video capture chip TVP5158 hand-driven goals, write Linux I2C device driver.

**1. i2c\_driver structure objects**

Each I2C device driver, you must first create a i2c\_driver structure of the object, the structure comprises a I2C device and method for detecting and some basic information about cancellation, for example:

* 1. static struct i2c\_driver tvp5158\_i2c\_driver = {
  2. .driver = {
  3. = .name "tvp5158\_i2c\_driver" ,
  4. },
  5. .attach\_adapter = & tvp5158\_attach\_adapter,
  6. .detach\_client = & tvp5158\_detach\_client,
  7. .command = NULL,
  8. };

Wherein, name this field identifies the name of the driver (not more than 31 characters), attach\_adapter and detach\_client field as a function pointer, these two functions will be automatically called I2C device register, these two functions need to implement their own, will be detailed later about.

2. **i2c\_client structure objects**

Defined above i2c\_driver objects, abstract-driven model i2c provides for i2C device detection and logout methods, while i2c\_client structure is represent a specific i2c device, the structure has a data pointer can point to any private the device data, the driving point of the complex may be used. Examples are as follows:

* 1. struct tvp5158\_obj {
  2. struct i2c\_client Client;
  3. **int** the Users; // How MANY a using the Users at The Driver
  4. };
  5. struct tvp5158\_obj \* g\_tvp5158\_obj;

Wherein, users as an example, users can add their own field of interest in this tvp5158\_obj structure inside, but i2c\_client field essential. Speak in detail later specific usage.

**3. Device Registration and detection function**

This step is critical, according to the requirements of the standard to write, the Linux system will automatically call the relevant code to detect your I2C devices, and add to the list of I2C devices in the system for later access.

We know that each I2C device chip, through hardware device connected to set up the I2C address of the device. Thus, detection of I2C device is generally accomplished by the device address. So, you must first declare your I2C device address list to be detected in the driver code, as well as a macro. Examples are as follows:

* 1. static unsigned **Short** normal\_i2c [] = {
  2. 0xbc >> 1,
  3. 0xbe >> 1,
  4. I2C\_CLIENT\_END
  5. };
  6. I2C\_CLIENT\_INSMOD;

normal\_i2c array contains a list of I2C device address you need to probe, and must I2C\_CLIENT\_END as the end, note that the above code 0xbc and 0xbe is my hardware for my tvp5158 assigned address, I support the hardware by jumpers the address is set to 0xbc or 0xbe, so these two addresses are written into the detection list, let the system be detected. If your I2C device address is fixed, then, where you can just write your own I2C device address, attention must be shifted to the right one.

Macro I2C\_CLIENT\_INSMOD role many online articles explaining in detail, I will not described in detail, remember to add the line, we focus on implementation.

The next step should be prepared in step 1 of the two callback functions, one for registering the device, a device for the cancellation. Example detection function as follows:

* 1. static **int** tvp5158\_attach\_adapter ( struct i2c\_adapter \* Adapter)
  2. {
  3. return i2c\_probe (Adapter, & addr\_data, & tvp5158\_detect\_client);
  4. }

The system will automatically call the callback function, we only need to follow the above code in the form of trying to be difficult to write, here called I2C device detection function of the system, i2c\_probe (), the third parameter to the callback function to detect the specific device, the system will detect the device when calling this function, it needs its own implementation. Examples are as follows:

* 1. static **int** tvp5158\_detect\_client ( struct i2c\_adapter Adapter \*, **int** address, **int** kind)
  2. {
  3. struct tvp5158\_obj \* pObj;
  4. **int** ERR = 0;
  5. printk (KERN\_INFO "the I2C: tvp5158\_detect\_client address AT X% ... \ N" , address);
  6. IF (g\_tvp5158\_obj! = NULL) {
  7. // Already allocated, inc user count, and return the allocated handle
  8. g\_tvp5158\_obj-> users ++;
  9. R eturn 0;
  10. }
  11. / \* Alloc obj \* /
  12. = kmalloc pObj ( the sizeof ( struct tvp5158\_obj), GFP\_KERNEL);
  13. IF (pObj == 0) {
  14. return -ENOMEM;
  15. }
  16. memset (pObj, 0, the sizeof ( struct tvp5158\_obj));
  17. pObj-> client.addr = address;
  18. pObj-> client.adapter = adapter;
  19. pObj-> client.driver = & tvp5158\_i2c\_driver;
  20. pObj-> client.flags = I2C\_CLIENT\_ALLOW\_USE;
  21. pObj-> users ++;
  22. / \* Attach i2c client to sys i2c clients list \* /
  23. IF ((ERR = i2c\_attach\_client (& pObj-> Client))) {
  24. printk (KERN\_ERR "the I2C: ERROR: Fail i2c\_attach\_client address =% X \ N!" , address);
  25. return ERR;
  26. }
  27. // Store the pObj
  28. g\_tvp5158\_obj = pObj;
  29. printk (KERN\_ERR "! the I2C: i2c\_attach\_client address the ok =% X \ N" , address);
  30. return 0;
  31. }

So far, the detection device and the code registration has been completed, after the device can be accessed via I2C g\_tvp5158\_obj this global pointer proceed.

**4. Log off I2C devices**

Similarly, the device will automatically log off callback function called by the system, you only need to write a template according to equipment write-off code examples are as follows:

* 1. static **int** tvp5158\_detach\_client ( struct i2c\_client \* Client)
  2. {
  3. **int** ERR;
  4. IF (! Client-> Adapter) {
  5. return -ENODEV;
  6. }
  7. IF ((ERR = i2c\_detach\_client (Client))) {
  8. printk (KERN\_ERR "deregistration failed The Client (address = X%), Client Not detached \ N." , Client-> addr);
  9. return ERR;
  10. }
  11. client-> adapter = NULL;
  12. IF (g\_tvp5158\_obj) {
  13. kfree (g\_tvp5158\_obj);
  14. }
  15. return 0;
  16. }

So far, registration and deregistration codes equipment has been completed, the next thing is to provide a method to read and write I2C device.

**5. I2C devices to read and write**

Read and write I2C devices, Linux system provides a variety of interfaces, can be found in the kernel i2c.h in here briefly one of the two interfaces.

Interface [a]:

* 1. extern **int** i2c\_master\_send ( struct i2c\_client \*, const **char** \*, **int** );
  2. extern **int** i2c\_master\_recv ( struct i2c\_client \*, **char** \*, **int** );

The first parameter is i2c\_client object pointer, and the second parameter is a pointer to buffer data to be transmitted, and the third parameter is the size of the buffer.

Interface [two]:

* 1. extern **int** i2c\_transfer ( struct i2c\_adapter ADAP \*, struct i2c\_msg \* msg, **int** NUM);

This interface supports send multiple messages to the I2C device, each message can be read also can write, read or write, and destination address (register address) to read and write are contained in the msg parameter message inside.

These interfaces are only the bottom of the read and write methods on how specific I2C devices interact with, such as how to read a specific value of a particular register of the chip, which need to see specific manual chip, each chip will have a specific I2C I2C register read and write timing diagram.Therefore, in order to provide better access interface in the driver, but also according to the specific timing requirements for these functions to read and write further encapsulated, these will be covered in a later article.

**6. The module initialization and other**

The next step is the initialization code initialization code and reverse the entire module, and a module declared.

* 1. static **int** the \_\_init tvp5158\_i2c\_init ( void )
  2. {
  3. g\_tvp5158\_obj = NULL;
  4. return i2c\_add\_driver (& tvp5158\_i2c\_driver);
  5. }
  6. static void \_\_exit tvp5158\_i2c\_exit ( void )
  7. {
  8. i2c\_del\_driver (& tvp5158\_i2c\_driver);
  9. }
  10. module\_init (tvp5158\_i2c\_init);
  11. module\_exit (tvp5158\_i2c\_exit);
  12. MODULE\_DESCRIPTION ( "TVP5158 I2C Driver" );
  13. MODULE\_AUTHOR ( "Lujun @hust" );
  14. MODULE\_LICENSE ( "the GPL" );

In the initialization code inside, add this module i2c driver objects in reverse initialization code inside, remove i2c driver objects of this module.

**7. Summary**

So far, considered from the application point to write a I2C device driver code is finished, a lot of the principles of things I have no specific analysis (in fact, I know not deep), the future will slowly learn and understand more deeply , what about the text is not the right place, please leave a message or letter lujun-hust@gmail.com exchange.

Read Finally, we may have a doubt, the driver finished the user how to use the space (Application Layer) do it? Since this confused too many do not want to put the code is too complex to increase the difficulty of understanding the fear, so I did not say, in fact, in order to use the I2C device driver in user space, you also need to use a character device driver to complete, is the I2C device drive package one character device driver, so that the user can access the space through the I2C device character device driver to access, this way I will be covered in a later article.

From <<http://blog.csdn.net/gl1987807/article/details/8745120>>