# I2C

## 框架



在内核中，根据总线-设备-模型，i2c驱动可以分成两层来实现——设备驱动和总线驱动。其中，设备驱动根据总线驱动给的接口，去实现具体设备device的write()、read()、ioctl()等方法，赋值给file\_operations，然后注册字符设备。总线驱动又分为i2c核心层（i2c-core）以及i2c适配器层（i2c-adapter）的硬件驱动层。I2c核心层可以理解成总线驱动的纯软件部分，实现了i2c的一些核心算法和数据结构；i2c适配器层是和单板相关的硬件驱动操作，提供i2c adapter的algorithm，用具体适配器的xxx\_xferf()函数来填充i2c\_algorithm的master\_xfer函数指针，并把赋值后的i2c\_algorithm再赋值给i2c\_adapter的algo指针。



在使用内核的数据结构添加了一个i2c设备后，根据总线-设备-驱动模型架构，会将该设备挂在i2c总线的i2c\_device链表中，并通过i2c\_client数据结构来描述；同理，若添加了一个i2c驱动后，会将该设备挂在i2c总线的i2c\_driver链表中，并通过i2c\_driver数据结构来描述。在图中只表示一些数据结构中比较核心的成员。在新添加设备或驱动时，i2c总线会遍历两边的链表，若数据结构中的name能够匹配，则调用i2c\_driver中的probe函数，去完成设备的初始化，使设备和驱动之间建立起联系。这实际上是总线-设备-驱动模型的一个实例化数据结构，详细成员如下图所示：

struct i2c\_client {

unsigned short flags; /\* div., see below \*/

unsigned short addr; /\* chip address - NOTE: 7bit \*/

/\* addresses are stored in the \*/

/\* \_LOWER\_ 7 bits \*/

char name[I2C\_NAME\_SIZE];

struct i2c\_adapter \*adapter; /\* the adapter we sit on \*/

struct i2c\_driver \*driver; /\* and our access routines \*/

struct device dev; /\* the device structure \*/

int irq; /\* irq issued by device \*/

struct list\_head detected;

};

struct i2c\_driver {

unsigned int class;

/\* Notifies the driver that a new bus has appeared or is about to be

\* removed. You should avoid using this, it will be removed in a

\* near future.

\*/

int (\*attach\_adapter)(struct i2c\_adapter \*) \_\_deprecated;

int (\*detach\_adapter)(struct i2c\_adapter \*) \_\_deprecated;

/\* Standard driver model interfaces \*/

int (\*probe)(struct i2c\_client \*, const struct i2c\_device\_id \*);

int (\*remove)(struct i2c\_client \*);

/\* driver model interfaces that don't relate to enumeration \*/

void (\*shutdown)(struct i2c\_client \*);

int (\*suspend)(struct i2c\_client \*, pm\_message\_t mesg);

int (\*resume)(struct i2c\_client \*);

/\* Alert callback, for example for the SMBus alert protocol.

\* The format and meaning of the data value depends on the protocol.

\* For the SMBus alert protocol, there is a single bit of data passed

\* as the alert response's low bit ("event flag").

\*/

void (\*alert)(struct i2c\_client \*, unsigned int data);

/\* a ioctl like command that can be used to perform specific functions

\* with the device.

\*/

int (\*command)(struct i2c\_client \*client, unsigned int cmd, void \*arg);

struct device\_driver driver;

const struct i2c\_device\_id \*id\_table;

/\* Device detection callback for automatic device creation \*/

int (\*detect)(struct i2c\_client \*, struct i2c\_board\_info \*);

const unsigned short \*address\_list;

struct list\_head clients;

};

## 添加i2c设备的四种方式

### i2c\_new\_device

struct i2c\_client \*

i2c\_new\_device(struct i2c\_adapter \*adap, struct i2c\_board\_info const \*info)；

如图所示，直接调用该函数接口进行添加一个i2c\_device到链表中。

定义一个i2c\_board\_info, 里面有:名字, 设备地址等关键信息，接着添加到adap参数对应的i2c适配器下。

struct i2c\_board\_info {

char type[I2C\_NAME\_SIZE]; //名字

unsigned short flags;

unsigned short addr; //地址

void \*platform\_data;

struct dev\_archdata \*archdata;

struct device\_node \*of\_node;

int irq;

};

该接口还有一个强化版本i2c\_new\_probed\_device，此接口用于在添加i2c设备前通过发送起始信号和从设备地址来确认从设备是否物理存在，确认收到应答信号之后再调用i2c\_new\_device(adap, info)去添加i2c设备。其核心逻辑如下：

i2c\_new\_probed\_device

probe(adap, addr\_list[i]) /\* 确定设备是否真实存在 \*/

info->addr = addr\_list[i];

i2c\_new\_device(adap, info);

### i2c\_register\_board\_info

int \_\_init

i2c\_register\_board\_info(int busnum,

struct i2c\_board\_info const \*info, unsigned len)

{

通过i2c\_register\_board\_info 函数，即将该结构体对应的i2c设备挂接到busnum对应的i2c适配器上，把它们放入\_\_i2c\_board\_list链表中list\_add\_tail(&devinfo->list, &\_\_i2c\_board\_list);该链表会在i2c\_adapter初始化的时候被解析，取出其中的结构体使用2.1中的接口进行i2c设备注册。其流程如下所示：

i2c\_register\_adapter > i2c\_scan\_static\_board\_info > i2c\_new\_device

使用限制：必须在 i2c\_register\_adapter 之前 i2c\_register\_board\_info，只适合静态添加i2c设备，不适合动态加载。

### 从用户空间创建设备

这里举一个例子：

创建设备

echo at24c08 0x50 > /sys/class/i2c-adapter/i2c-0/new\_device

i2c\_new\_device被调用，创建了一个at24c08设备，其从设备地址为0x50.

删除设备

echo 0x50 > /sys/class/i2c-adapter/i2c-0/delete\_device

i2c\_unregister\_device被调用，删除该设备相关信息。

### i2c\_add\_driver

前面的3种方法都要事先确定适配器(I2C总线,I2C控制器)，如果事先并不知道这个I2C设备在哪个适配器上，怎么办？这个接口就是来解决这个问题的。它在所有的适配器上查找，找到正确的适配器。其核心逻辑如下所示，其中，缩进代表下一层的调用。

i2c\_add\_driver

i2c\_register\_driver

a. at24cxx\_driver放入i2c\_bus\_type的drv链表

并且从dev链表里取出能匹配的i2c\_client并调用probe

driver\_register

b. 对于每一个适配器，调用\_\_process\_new\_driver

对于每一个适配器，调用它的函数确定address\_list里的设备是否存在

如果存在，再调用detect进一步确定、设置，然后i2c\_new\_device

/\* Walk the adapters that are already present \*/

i2c\_for\_each\_dev(driver, \_\_process\_new\_driver);

\_\_process\_new\_driver

/\* 在总线-设备-驱动模型中，i2c设备也包括adapter，这类device的type为i2c\_adapter\_type

\* 这个函数的目的就是去遍历device链表中的adapter

\*/

if (dev->type != &i2c\_adapter\_type) return 0；

i2c\_do\_add\_adapter

/\* Detect supported devices on that bus, and instantiate them \*/

i2c\_detect(adap, driver);

for (i = 0; address\_list[i] != I2C\_CLIENT\_END; i += 1) {

err = i2c\_detect\_address(temp\_client, driver);

/\* 判断这个设备是否存在：简单的发出S信号确定有ACK \*/

if (!i2c\_default\_probe(adapter, addr))

return 0;

memset(&info, 0, sizeof(struct i2c\_board\_info));

info.addr = addr;

// 设置info.type

err = driver->detect(temp\_client, &info);

i2c\_new\_device