# 总结

|  |
| --- |
| **I2C核心**  i2c\_transfer(i2c\_adapter, i2c\_msg) : I2C适配器和i2c设备之间的消息交互  |--i2c\_adapter->algo->master\_xfer()  i2c\_master\_send(i2c\_client, buf, len) : I2C发送  |--i2c\_transfer()  i2c\_master\_recv(i2c\_client, buf, len) : I2C接收  |--i2c\_transfer()  **master**\_**xfer  owl\_i2c\_xfer:**  **owl\_i2c\_xfer() :**  写数据 ：  将需要发送的设备地址和数据写入TX FIFO中，若FIFO满，等owl\_i2c\_interrupt(0中断产生后，在中断处理函数中继续写入剩余的数据  读数据 :  将需要发送的设备地址和寄存器地址写入FIFO中，等owl\_i2c\_interrupt(0中断产生后，在中断处理函数中读出RX FIFO中的数据。 |
| **I2C总线驱动: i2c\_adapter**  I2C总线驱动有四个: i2c0, i2c1, i2c2, i2c3.  struct owl\_i2c\_dev  |--struct i2c\_adapter adapter;  |--int irq; //中断号: DTS中设置  |--enum i2c\_freq\_mode freq\_mode; //工作频率  struct i2c\_adapter  |--int nr; //i2c adapter编号: i2c0, i2c1, i2c2, i2c3  |--const struct i2c\_algorithm \*algo;  |--int (\*master\_xfer)(struct i2c\_adapter \*adap, struct i2c\_msg \*msgs, int num);  i2c\_add\_adapter() : 增加i2c\_adapter  |-- i2c\_register\_adapter()  i2c\_del\_adapter() : 删除i2c\_adapter  i2c\_get\_adapter(int nr): 获取指定编号的adapter  i2c\_put\_adapter (struct i2c\_adapter \*adap): 释放adapter |
| **I2C设备驱动: i2c\_driver**  struct i2c\_driver //camera\_i2c\_driver  |--int (\*probe)(struct i2c\_client \*, const struct i2c\_device\_id \*);  //camera\_module\_probe()  |--const struct i2c\_device\_id \*id\_table;  |--struct device\_driver driver;  i2c\_register\_driver() : 增加i2c\_driver  i2c\_del\_driver() : 删除i2c\_driver  i2c\_add\_driver() : 等价于i2c\_register\_driver() |
| **I2C设备: i2c\_client**  struct i2c\_client  |--unsigned short addr; //board\_info中的配置  |--struct i2c\_adapter \*adapter; // i2c\_new\_device()中设置  |--struct i2c\_driver \*driver; //i2c\_device\_probe()中设置  |--struct device dev;  |--int irq;  i2c\_new\_device( i2c\_adapter, i2c\_board\_info) : 创建i2c\_client  i2c\_unregister\_device(struct i2c\_client \*client)：销毁i2c\_client |
| **I2c数据的发送与接收: i2c\_transfer()**  struct i2c\_msg {  \_\_u16 addr; /\* slave address \*/  \_\_u16 flags;  \_\_u16 len; /\* msg length \*/  \_\_u8 \*buf; /\* pointer to msg data \*/  };  Flag:  #define I2C\_M\_TEN 0x0010 /\* this is a ten bit chip address \*/  #define I2C\_M\_RD 0x0001 /\* read data, from slave to master \*/  #define I2C\_M\_STOP 0x8000 /\* if I2C\_FUNC\_PROTOCOL\_MANGLING \*/  #define I2C\_M\_NOSTART 0x4000 /\* if I2C\_FUNC\_NOSTART \*/  #define I2C\_M\_REV\_DIR\_ADDR 0x2000 /\* if I2C\_FUNC\_PROTOCOL\_MANGLING \*/  #define I2C\_M\_IGNORE\_NAK 0x1000 /\* if I2C\_FUNC\_PROTOCOL\_MANGLING \*/  #define I2C\_M\_NO\_RD\_ACK 0x0800 /\* if I2C\_FUNC\_PROTOCOL\_MANGLING \*/  #define I2C\_M\_RECV\_LEN 0x0400 /\* length will be first received byte \*/  i2c\_transfer(i2c\_msg) : i2c数据的通信 |

# I2c-core

## I2c\_init()

|  |  |  |  |
| --- | --- | --- | --- |
| postcore\_initcall(**i2c\_init**);  1. 注册i2c总线  |--**bus**\_**register**(&**i2c\_bus\_type**);   |  | | --- | | struct **bus\_type** **i2c**\_**bus**\_**type** = {  .name = "i2c",  .**match** = i2c\_device\_match,  .**probe** = i2c\_device\_probe,  .remove = i2c\_device\_remove,  .shutdown = i2c\_device\_shutdown,  .pm = &i2c\_device\_pm\_ops,  }; |   |--struct class\_compat \*i2c\_adapter\_compat\_class = class\_compat\_register("i2c-adapter");  2. 注册虚拟的i2c\_driver  |--**i2c**\_**add**\_**driver**(&**dummy**\_**driver**);  |-- **i2c**\_**register**\_**driver**(THIS\_MODULE, driver)   |  | | --- | | static struct **i2c**\_**driver** **dummy**\_**driver** = {  .driver.name = "dummy",  .probe = dummy\_probe,  .remove = dummy\_remove,  .id\_table = dummy\_id,  }; | | static const struct i2c\_device\_id dummy\_id[] = {  { "dummy", 0 },  { },  }; | |

## I2c bus

### i2c\_device\_match ()

|  |
| --- |
| **i2c\_device\_match**(struct device \*dev, struct device\_driver \*drv)  1. 比较device\_driver的of\_match\_table  |--if (of\_driver\_match\_device(dev, drv)) return 1;  |--return of\_match\_device(drv->**of**\_**match**\_**table**, dev) != NULL  2. 比较device\_driver的acpi\_match\_table  |--if (acpi\_driver\_match\_device(dev, drv)) return 1;  |--return !!acpi\_match\_device(drv->**acpi**\_**match**\_**table**, dev);  3. 比较i2c\_driver的id\_table  |--struct **i2c**\_**client** \*client = i2c\_verify\_client(dev);  |--struct **i2c**\_**driver** \*driver = to\_i2c\_driver(drv);  |--return i2c\_match\_id(driver->**id**\_**table**, client) != NULL; |

### i2c\_device\_probe()

|  |
| --- |
| **i2c\_device\_probe**(struct device \*dev)  |--struct i2c\_client \*client = i2c\_verify\_client(dev);  |--struct i2c\_driver \*driver = to\_i2c\_driver(dev->driver);  |--client->driver = driver;  |--**driver**->**probe**(client, i2c\_match\_id(driver->id\_table, client)); // camera\_module\_probe () |

## I2c\_adapter

### owl\_i2c\_init: actions,s700-i2c

|  |  |  |
| --- | --- | --- |
| subsys\_initcall(**owl\_i2c\_init**);  |--**platform**\_**driver**\_**register**(&owl\_i2c\_driver);   |  | | --- | | static struct **platform**\_**driver** owl\_i2c\_driver = {  .**probe** = owl\_i2c\_probe,  .remove = owl\_i2c\_remove,  .driver = {  .name = "i2c-owl",  .owner = THIS\_MODULE,  .**of**\_**match**\_**table** = of\_match\_ptr(owl\_i2c\_dt\_ids),  },  }; | | static const struct **of**\_**device**\_**id** owl\_i2c\_dt\_ids[] = {  {.compatible = "actions,s900-i2c"},  {.compatible = "actions,s700-i2c"},  {},  }; | |

### DTS: actions,s700-i2c

|  |
| --- |
| **i2c0: i2c@e0170000** {  compatible = "actions,s700-i2c";  reg = <0 0xe0170000 0 0x1000>;  **interrupts = <GIC\_SPI 25 IRQ\_TYPE\_LEVEL\_HIGH>;**  #address-cells = <1>;  #size-cells = <0>;  clocks = <&clock CLK\_I2C0>;  clock-names = "i2c0";  pinctrl-names = "default";  pinctrl-0 = <&pinctrl\_i2c0\_default>;  status = "disabled";  }; |
| **i2c1: i2c@e0174000** {  compatible = "actions,s700-i2c";  reg = <0 0xe0174000 0 0x1000>;  **interrupts = <GIC\_SPI 26 IRQ\_TYPE\_LEVEL\_HIGH>;**  #address-cells = <1>;  #size-cells = <0>;  clocks = <&clock CLK\_I2C1>;  clock-names = "i2c1";  //pinctrl-names = "default";  //pinctrl-0 = <&pinctrl\_i2c1\_default>;  status = "disabled";  }; |
| **i2c2: i2c@e0178000** {  compatible = "actions,s700-i2c";  reg = <0 0xe0178000 0 0x1000>;  **interrupts = <GIC\_SPI 27 IRQ\_TYPE\_LEVEL\_HIGH>;**  #address-cells = <1>;  #size-cells = <0>;  clocks = <&clock CLK\_I2C2>;  clock-names = "i2c2";  pinctrl-names = "default";  pinctrl-0 = <&pinctrl\_i2c2\_default>;  status = "disabled";  }; |
| **i2c3: i2c@e017c000** {  compatible = "actions,s700-i2c";  reg = <0 0xe017c000 0 0x1000>;  **interrupts = <GIC\_SPI 28 IRQ\_TYPE\_LEVEL\_HIGH>;**  #address-cells = <1>;  #size-cells = <0>;  clocks = <&clock CLK\_I2C3>;  clock-names = "i2c3";  status = "disabled";  }; |

### owl\_i2c\_probe

|  |  |
| --- | --- |
| **owl\_i2c\_probe**(struct platform\_device \*pdev)  1. 创建owl\_i2c\_dev设备  |--struct **owl**\_**i2c**\_**dev** \*dev = **devm**\_**kzalloc**(&pdev->dev, sizeof(\*dev), GFP\_KERNEL);    2. 设置i2c0/1/2/3的中断函数  |--dev->irq = platform\_get\_irq(pdev, 0);  |--**devm**\_**request**\_**irq**(&pdev->dev, dev->**irq**, **owl**\_**i2c**\_**interrupt**, 0, dev\_name(dev->dev), dev);    3. 设置i2c的工作始终频率  |--dev->clk = devm\_clk\_get(dev->dev, NULL);  |--clk\_prepare\_enable(dev->clk);    4. 配置及注册i2c\_adapter  |--struct **i2c**\_**adapter** \*adap = &dev->adapter;  |--adap->class = I2C\_CLASS\_HWMON;  |--adap->**algo** = &**owl**\_**i2c**\_**algorithm**;  |--**adap->nr = pdev->id**;   |  | | --- | | static struct i2c\_algorithm **owl**\_**i2c**\_**algorithm** = {  .**master**\_**xfer** = owl\_i2c\_xfer,  .functionality = owl\_i2c\_func,  }; |   |--i2c\_add\_numbered\_adapter(adap);  |--if (adap->nr == -1)  |--**i2c**\_**add**\_**adapter**(adap);  |--id = idr\_alloc(&i2c\_adapter\_idr, adapter,\_\_i2c\_first\_dynamic\_bus\_num, 0, GFP\_KERNEL);  |--**adapter->nr = id;**  |--**i2c**\_**register**\_**adapter**(adapter);  |--\_\_i2c\_add\_numbered\_adapter(adap);  |-- id = idr\_alloc(&i2c\_adapter\_idr, adap, adap->nr, adap->nr + 1,GFP\_KERNEL);  |-- **i2c**\_**register**\_**adapter**(adap); |

### i2c\_register\_adapter

|  |
| --- |
| **i2c\_register\_adapter**(struct i2c\_adapter \*adap)  1. 注册i2c\_adapter, device\_register()中会调用blocking\_notifier\_call\_chain(&dev->bus->p->bus\_notifier,BUS\_NOTIFY\_ADD\_DEVICE, dev);上报事件  |--adap->dev.bus = **&i2c\_bus\_type;**  |--adap->dev.type = **&i2c\_adapter\_type;**  |--**device**\_**register**(&adap->dev);  2. 通知i2c总线有新的i2c\_adapter注册  |--bus\_for\_each\_drv(&**i2c**\_**bus**\_**type**, NULL, adap, **\_\_process**\_**new**\_**adapter**);  |--\_\_process\_new\_adapter(struct device\_driver \*d, void \*data)  |--i2c\_do\_add\_adapter(to\_i2c\_driver(d), data); |
| **i2c\_do\_add\_adapter**(struct i2c\_driver \*driver, struct i2c\_adapter \*adap)  |--i2c\_detect(adap, driver);  |--struct **i2c**\_**client** \*temp\_client = **kzalloc**(sizeof(struct i2c\_client), GFP\_KERNEL);  |--temp\_client->adapter = adapter;    |--address\_list = driver->**address**\_**list**;  |--for (i = 0; address\_list[i] != I2C\_CLIENT\_END; i += 1)  |--temp\_client->addr = address\_list[i];  |--i2c\_detect\_address(temp\_client, driver);  |--**kfree**(temp\_client); |

## I2c\_driver

### camera\_module\_init()

|  |  |  |
| --- | --- | --- |
| **camera\_module\_init**()  1. 注册i2c\_driver： camera  |--**sensor**\_**mod**\_**init**(&**camera**\_**module**\_**link**, &**asoc\_camera\_device,** &**camera\_i2c\_driver**);  |--**i2c**\_**add**\_**driver**(camera\_i2c\_driver);  |-- **i2c**\_**register**\_**driver**(THIS\_MODULE, driver)   |  | | --- | | static struct **i2c**\_**driver** **camera**\_**i2c**\_**driver** = {  .driver = {  .name = CAMERA\_MODULE\_NAME,  },  .**probe** = camera\_module\_probe,  .suspend = camera\_module\_suspend,  .resume = camera\_module\_resume,  .remove = camera\_module\_remove,  .**id**\_**table** = camera\_module\_id,  }; | | static const struct **i2c**\_**device**\_**id** camera\_module\_id[] = {  {CAMERA\_MODULE\_NAME, 0}, // "EPC660"  {}  }; | |

### i2c\_register\_driver

|  |
| --- |
| **i2c\_register\_driver**(struct module \*owner, struct i2c\_driver \*driver)  1. 注册device\_driver  |--driver->driver.owner = owner;  |--driver->driver.bus = **&i2c\_bus\_type;**  |--**driver**\_**register**(&driver->driver);  2.  |--i2c\_for\_each\_dev(driver, **\_\_process\_new\_driver**);  |--\_\_process\_new\_driver(struct device \*dev, void \*data)  |--if (dev->type == &i2c\_adapter\_type)  |--i2c\_do\_add\_adapter(data, to\_i2c\_adapter(dev)); |
| **i2c\_do\_add\_adapter**(struct i2c\_driver \*driver, struct i2c\_adapter \*adap)  |--i2c\_detect(adap, driver);  |--struct i2c\_client \*temp\_client = kzalloc(sizeof(struct i2c\_client), GFP\_KERNEL);  |--temp\_client->adapter = adapter;    |--address\_list = driver->address\_list;  |--for (i = 0; address\_list[i] != I2C\_CLIENT\_END; i += 1)  |--temp\_client->addr = address\_list[i];  |--i2c\_detect\_address(temp\_client, driver);  |--kfree(temp\_client); |

### camera\_module\_probe()

|  |
| --- |
| **camera\_module\_probe**(struct i2c\_client \*client, const struct i2c\_device\_id \*did)  1. 创建camera\_module\_priv，包含v4l2\_subdev设备  |--struct **camera**\_**module**\_**priv** \*priv = **devm**\_**kzalloc**(&client->dev, sizeof(\*priv), GFP\_KERNEL);  2. 初始化v4l2\_subdev设备  |--**v4l2**\_**i2c**\_**subdev**\_**init**(&priv->subdev, client, &**module**\_**subdev**\_**ops**);  |--**camera**\_**module**\_**init**\_**ops**(&priv->hdl, &**camera\_module\_ctrl\_ops**);  |--priv->subdev.ctrl\_handler = &priv->hdl; |
| **v4l2\_i2c\_subdev\_init**(struct v4l2\_subdev \*sd, struct i2c\_client \*client, const struct v4l2\_subdev\_ops \*ops)  1. 构建v4l2子设备，其类型为I2c设备  |--sd->flags |= **V4L2\_SUBDEV\_FL\_IS\_I2C;**  |--sd->owner = **client**->driver->driver.owner;  |--v4l2\_set\_subdevdata(sd, **client**);  |--i2c\_set\_clientdata(**client**, sd); |

## i2c\_client

### board\_info

|  |
| --- |
| static struct **soc**\_**camera**\_**link** camera\_module\_link = {  .bus\_id = 0,  .power = camera\_module\_power,  .reset = camera\_module\_reset,  .**board**\_**info** = &asoc\_i2c\_camera,  .**i2c**\_**adapter**\_**id** = 1, /\*id num start from 0 \*/  .module\_name = CAMERA\_MODULE\_NAME,  .priv = &camera\_module\_info,  }; |
| static struct **i2c**\_**board**\_**info** asoc\_i2c\_camera = {  I2C\_BOARD\_INFO("**EPC660**", (**0x20**)/\*7bit\*/),  };  #define I2C\_BOARD\_INFO(dev\_type, dev\_addr) \  .**type** = dev\_type, .**addr** = (dev\_addr) |

### adapter\_id

|  |
| --- |
| sensors: isp-sensor {  compatible = "sensor\_common";  **i2c**\_**adapter** = <&**i2c1**>; /\* i2c3 -> i2c1 \*/  status = "okay";  }; |
| camera\_module\_init()  |--sensor\_mod\_init(&**camera**\_**module**\_**link**, &**asoc**\_**camera**\_**device**,&**camera**\_**i2c**\_**driver**);  |--parse\_config\_info(link, dsc, "sensor\_common");  |--struct **device**\_**node** \*fdt\_node = of\_find\_compatible\_node(NULL, NULL, name);  |--dsc->i2c\_adapter = get\_parent\_node\_id(fdt\_node, "**i2c**\_**adapter**", "i2c");  |--link->**i2c**\_**adapter**\_**id** = dsc->i2c\_adapter; |

### soc\_camera\_probe

|  |
| --- |
| **soc\_camera\_probe**(struct soc\_camera\_device \*icd)  |--struct **soc**\_**camera**\_**desc** \*sdesc = to\_soc\_camera\_desc(icd);  |--struct **soc**\_**camera**\_**host**\_desc \*shd = &sdesc->host\_desc;  |--if (shd->**board**\_**info**)  1. 创建i2c\_client设备  |--soc\_camera\_init\_i2c(icd, sdesc);  |--struct **i2c**\_**adapter** \*adap = i2c\_get\_adapter(shd->**i2c**\_**adapter**\_**id**);  |--struct **v4l2**\_**subdev** \*subdev = v4l2\_i2c\_new\_subdev\_board(&ici->v4l2\_dev, adap,shd->board\_info, NULL);  |--struct **i2c**\_**client** \*client = **i2c**\_**new**\_**device**(adapter, info);  |--struct **v4l2**\_**subdev** \*sd = i2c\_get\_clientdata(client);  |--**v4l2**\_**device**\_**register**\_**subdev**(v4l2\_dev, sd)  |--struct **i2c**\_**client** \*client = v4l2\_get\_subdevdata(subdev);  |--icd->control = &client->dev; |

### i2c\_new\_device

|  |
| --- |
| **i2c\_new\_device**(struct i2c\_adapter \*adap, struct i2c\_board\_info const \*info)  1. 创建i2c\_client设备  |--struct **i2c**\_**client** client = **kzalloc**(sizeof \*client, GFP\_KERNEL);  |--client->**adapter** = **adap**;  |--client->**addr** = info->**addr**; //EPC i2c addr  |--client->**irq** = info->**irq**;  |--strlcpy(client->**name**, info->type, sizeof(client->name)); //"EPC660"  2. 检查i2c设备的地址是否正常  |--i2c\_check\_client\_addr\_validity(client);  |--i2c\_check\_addr\_busy(adap, client->addr);  3. 注册i2c\_client设备  |--client->dev.bus = &**i2c**\_**bus**\_**type**;  |--client->dev.type = &**i2c**\_**client**\_**type**;  |--**device**\_**register**(&client->dev); //触发i2c\_driver驱动的probe() |

## i2c\_msg : i2c设备的读写控制

|  |
| --- |
| **I2C 访问时序的一般命令列表如下：**  **S : start**  **A : Addr**  **D : Data**  **P : Stop**  **RS : Restart**  **Ar=Slave address+Read Flag**  **RL=Release Bus**  **作主设备写数据：**  1、 S+A+Reg1+D+P  2、 S+A+ Reg1+ Reg2+D+P  3、 S+A+ Reg1+ D+P  **作主设备读数据：**  4、 S+A+ Reg1+RS+Ar+D+P  5、 S+A+ Reg1+ Reg2+RS+Ar+D+P  6、 S+A+ Reg1+P+S+Ar+ D+P （中间是 Stop 后再重新发 Start 来进行读）  **作从设备写数据：**  7、 D+RL  **作从设备读数据：**  8、 RL+D  **Sample :**  **如果是 S+A+ Reg1+ Reg2+RS+Ar+D+P 访问时序读 10Byte 数据，**  5、 I2Cx\_CNT 要配置为 10.  6、 往 I2Cx\_TxDAT 写进 1Byte 的从设备地址， 1Byte 的从设备内部 Memory 地址，  7、 再写 1Byte 的从设备地址，  8、 配置 CMD 寄存器的数值为： 0x8F37，  9、 然后才是从 I2Cx\_RxDAT 读回 10Byte 的数据。  如果是 S+A+Reg1+D+P 访问时序写 10Byte 数据  5、 I2Cx\_CNT 要配置为 10.  6、 往 I2Cx\_TxDAT 写进 1Byte 的从设备地址， 1Byte 的从设备内部 Memory 地址  7、 继续往 I2Cx\_TxDAT 写进 10Byte 要发送的数据  8、 配置 CMD 寄存器的数据值为： 0x8D05， |

### camera\_i2c\_write()

|  |
| --- |
| **camera**\_**i2c**\_**write**(struct i2c\_adapter \*i2c\_adap,unsigned int data\_width, unsigned int reg,unsigned int data)  |--tran\_array[0] = (**reg** >> 8) & 0xff;  |--tran\_array[1] = **reg** & 0xff;  |--tran\_array[2] = (**data** >> 8) & 0xff;  |--tran\_array[3] = **data** & 0xff  1. 发送reg和data的值给i2c设备 ;  |--struct **i2c**\_**msg** msg;  |--msg.**addr** = **0x20**; //7bit  |--msg.**flags** = **0;**  |--msg.**len** = reg\_width + data\_width;  |--msg.**buf** = tran\_array;  |--**i2c**\_**transfer**(i2c\_adap, &msg, 1); |

### camera\_i2c\_read

|  |
| --- |
| **camera**\_**i2c**\_**read**(struct i2c\_adapter \*i2c\_adap,unsigned int data\_width, unsigned int reg, unsigned int \*data)  |--tran\_array[0] = (reg >> 8) & 0xff;  |--tran\_array[1] = reg & 0xff;  1. 发送reg给i2c设备  |--struct **i2c**\_**msg** msg;  |--msg.**addr** = **0x20**; //7bit  |--msg.**flags** = 0;  |--msg.**len** = reg\_width;  |--msg.**buf** = tran\_array;  |--i2c\_transfer(i2c\_adap, &msg, 1);  2. 读i2c设备的reg返回值  |--msg.**flags** = **I2C**\_M\_**RD**;  |--msg.**len** = data\_width;  |--msg.**buf** = tran\_array;  |--**i2c**\_**transfer**(i2c\_adap, &msg, 1);  |--\*data = tran\_array[0] << 8 | tran\_array[1]; |

## i2c\_transfer

|  |
| --- |
| **i2c**\_**transfer**(struct i2c\_adapter \*adap, struct i2c\_msg \*msgs, int num)  |--\_\_**i2c**\_**transfer**(adap, msgs, num);  |--orig\_jiffies = jiffies;  |--for (ret = 0, try = 0; try <= adap->**retries**; try++)  |--ret = **adap->algo->master**\_**xfer**(adap, msgs, num); // owl\_i2c\_xfer  |--if (ret != -EAGAIN)  |--break;  |--time\_after(jiffies, orig\_jiffies + adap->timeout); |

## owl\_i2c\_xfer : 写i2c设备地址

|  |
| --- |
| **owl**\_**i2c**\_**xfer**(struct i2c\_adapter \*adap, struct i2c\_msg \*msgs,int num)  |--struct **owl**\_**i2c**\_**dev** \*dev = i2c\_get\_adapdata(adap);  1. 初始化i2c模块的寄存器  |--**owl**\_**i2c**\_**hwinit**(dev);  |--owl\_i2c\_writel(dev, I2C\_CTL\_**EN**, **I2C**\_**CTL**); //使能i2c模块: I2C\_CTL  |--**owl**\_**i2c**\_**clear**\_**status**(dev);  |--val = owl\_i2c\_readl(dev, I2C\_STAT);  |--owl\_i2c\_writel(dev, val, I2C\_STAT); //复位状态位: I2C\_STAT  |--**owl**\_**i2c**\_**reset**\_**fifo**(dev);  |--val = owl\_i2c\_readl(dev, I2C\_FIFOCTL);  |--val |= I2C\_FIFOCTL\_**RFR** | I2C\_FIFOCTL\_**TFR**; //enable Tx Rx  |--owl\_i2c\_writel(dev, val, **I2C**\_**FIFOCTL**); //清空FIFO: I2C\_FIFOCTL  |--**owl**\_**i2c**\_**set**\_**freq**(dev);  |--pclk = clk\_get\_rate(dev->clk);  |--div\_factor = (pclk + dev->clk\_freq \* 16 - 1) / (dev->clk\_freq \* 16);  |--owl\_i2c\_writel(dev, I2C\_CLKDIV\_DIV(div\_factor), **I2C**\_**CLKDIV**); //设置i2c的工作频率  |--owl\_i2c\_wait\_if\_busy(dev); //等待I2C\_STAT的bit6清0  2. 将buf中的数据写到i2c寄存器中  |--**owl**\_**i2c**\_**do**\_**transfer**(dev, msgs, num);  2.1 初始化complete控制量  |--**init**\_**completion**(&dev->cmd\_complete);  2.2使能irq  |--owl\_i2c\_writel(dev, I2C\_CTL\_**IRQE** | I2C\_CTL\_**EN**, **I2C**\_**CTL**);  2.3配置cmd属性  /\* 配置cmd属性：  \* I2C\_CMD\_EXEC : Start bit enable  \* I2C\_CMD\_MSS : Master Mode  \* I2C\_CMD\_SE : Stop Enable  \* I2C\_CMD\_DE : Data Enable  \* I2C\_CMD\_NS : NACK Enable  \* I2C\_CMD\_SBE : Start bit enable  \*/  |--fifo\_cmd = I2C\_CMD\_**EXEC** | I2C\_CMD\_**MSS** | I2C\_CMD\_**SE** | I2C\_CMD\_**DE** | I2C\_CMD\_**NS** | I2C\_CMD\_**SBE**;  2.3.1 两个i2c\_msg时，第一个一定是写操作，将写操作的地址和值写到TX中  |--if (num == 2)  /\* 配置cmd属性：  \* I2C\_CMD\_AS : Address select : (msgs[0].len + 1) byte address  \* I2C\_CMD\_SAS : Second Address select : 1 byte address  \* I2C\_CMD\_RBE : Restart bit enable  \*/  |--fifo\_cmd |= I2C\_CMD\_**AS**(msgs[0].len + 1) | I2C\_CMD\_**SAS**(1) | I2C\_CMD\_**RBE**;  |--**owl\_i2c\_writel(dev, addr, I2C\_TXDAT);**  |--for (i = 0; i < msgs[0].len; i++)  |--**owl\_i2c\_writel(dev, msgs[0].buf[i], I2C\_TXDAT);**  |--msg = &msgs[1];  |--else  /\* 配置cmd属性：  \* I2C\_CMD\_AS : Address select : 1 byte address  \*/  |--fifo\_cmd |= I2C\_CMD\_**AS**(1);  |--msg = &msgs[0];  2.3.2 处理最后一个i2c\_msg : 将地址和数据写到TX中  |--dev->curr\_msg = msg;  /\* 设置需要收发数据的数量 : I2C\_DATCNT : msg->len \*/  |--owl\_i2c\_writel(dev, msg->len, I2C\_**DATCNT**);  /\* 设置i2c设备的通信地址 : I2C\_TXDAT \*/  |--if (msg->flags & I2C\_M\_RD)  |--**owl\_i2c\_writel(dev, addr | 1, I2C\_TXDAT);**  |--dev->state = STATE\_READ\_DATA;  |--else  |--**owl\_i2c\_writel(dev, addr, I2C\_TXDAT);**  /\* 写操作 ： 还需将写数据填入 I2C\_TXDAT , 若FIFO满，则将msg\_prt执行下一个需要写入的数据，等irq中断产生后，在中断处理函数中继续写入 \*/  |--for (i = 0; i < msg->len; i++)  |--if (owl\_i2c\_readl(dev, I2C\_FIFOSTAT) & I2C\_FIFOSTAT\_**TFF**)  |--break;  |--**owl\_i2c\_writel(dev, msg->buf[i], I2C\_TXDAT);**  |--**dev->msg\_ptr = i;**  |--dev->state = STATE\_WRITE\_DATA;  /\* 配置FIFO的控制方式 \*/  |--if (msg->flags & I2C\_M\_IGNORE\_NAK)  |--owl\_i2c\_writel(dev, I2C\_FIFOCTL\_NIB, I2C\_FIFOCTL);  |--else  |--owl\_i2c\_writel(dev, 0, I2C\_**FIFOCTL**);  3. 启动i2c通信 : I2C\_CMD  |--owl\_i2c\_writel(dev, **fifo**\_**cmd**, I2C\_**CMD**);  4. 等待i2c通信结束  |--time\_left = **wait\_for\_completion\_timeout(&dev->cmd\_complete, time\_left);**  5. 设置i2c通信状态  |--if ((dev->state == STATE\_TRANSFER\_OVER))  |--ret = 0; //通信正常  |--else if (time\_left == 0)  |--ret = -EREMOTEIO; //通信超时  |--else  |--ret = -ENXIO; //通信错误  6. 停止i2c模块的工作  |--owl\_i2c\_writel(dev, **0**, **I2C**\_**CTL**);  7. 返回发送的i2c\_msg的个数  |--return (ret < 0) ? ret : num; |

## owl\_i2c\_interrupt : 读写i2c的数据

|  |
| --- |
| **owl**\_**i2c**\_**interrupt**(int irq, void \*dev\_id)  |--struct **owl**\_**i2c**\_**dev** \*dev = dev\_id;  1. 判断I2C的状态寄存器  |--stat = owl\_i2c\_readl(dev, **I2C**\_**STAT**);  |--fifostat = owl\_i2c\_readl(dev, **I2C**\_**FIFOSTAT**);  |--if (fifostat & I2C\_FIFOSTAT\_**RNB**) //no ACK from device  |--dev->state = STATE\_TRANSFER\_ERROR;  |--if (stat & I2C\_STAT\_**LAB**) //"lose arbitration"  |--dev->state = STATE\_TRANSFER\_ERROR;  |--if (stat & I2C\_STAT\_**BEB**) //"bus error"  |--dev->state = STATE\_TRANSFER\_ERROR;  2. 处理i2c数据的读写操作  |--struct i2c\_msg \*msg = dev->curr\_msg;  2.1 读I2C设备的数据: I2C\_RXDAT  |--if (msg->flags & I2C\_M\_**RD**)  |--while ((owl\_i2c\_readl(dev, I2C\_FIFOSTAT) & I2C\_FIFOSTAT\_**RFE**) && dev->msg\_ptr < msg->len)  |--**msg->buf[dev->msg\_ptr++] = owl\_i2c\_readl(dev, I2C\_RXDAT);**  2.2 将数据写到I2C设备: I2C\_TXDAT  |--else  |--while (!(owl\_i2c\_readl(dev, I2C\_FIFOSTAT) & I2C\_FIFOSTAT\_**TFF**)&& dev->msg\_ptr < msg->len)  |--**owl\_i2c\_writel(dev, msg->buf[dev->msg\_ptr++], I2C\_TXDAT);**  |--if (dev->msg\_ptr == msg->len)  |--dev->state **= STATE\_TRANSFER\_OVER**;  3. 判断i2c的状态寄存器  |--stat = owl\_i2c\_readl(dev, I2C\_STAT);  |--if (dev->state == **STATE\_TRANSFER\_ERROR**)  |--owl\_i2c\_reset(dev);  4. 结束i2c中断处理函数，唤醒i2c\_transfer()  |--if (dev->state == STATE\_TRANSFER\_ERROR || dev->state == STATE\_TRANSFER\_OVER)  |--**complete\_all(&dev->cmd\_complete);** |

# I2c-dev: /dev/i2c0

CONFIG\_I2C\_CHARDEV=y

module\_init(i2c\_dev\_init);

module\_exit(i2c\_dev\_exit);

## i2c\_dev\_init

|  |  |  |  |
| --- | --- | --- | --- |
| i2c\_dev\_init(void)  1. 创建字符设备 : i2c (需要手动mknod创建/dev/i2c节点)  |--register\_chrdev(I2C\_MAJOR, "i2c", &i2cdev\_fops);   |  | | --- | | #define I2C\_MAJOR 89 | | static const struct file\_operations i2cdev\_fops = {  .owner = THIS\_MODULE,  .llseek = no\_llseek,  .read = i2cdev\_read,  .write = i2cdev\_write,  .unlocked\_ioctl = i2cdev\_ioctl,  .open = i2cdev\_open,  .release = i2cdev\_release,  }; |   2. 创建/sys/class: i2c-dev  |--i2c\_dev\_class = class\_create(THIS\_MODULE, "i2c-dev");  3. 创建i2c bus的notify事件: 处理BUS\_NOTIFY\_ADD\_DEVICE和BUS\_NOTIFY\_DEL\_DEVICE事件。device\_register()中会调用blocking\_notifier\_call\_chain(&dev->bus->p->bus\_notifier,BUS\_NOTIFY\_ADD\_DEVICE, dev);上报事件  |--bus\_register\_notifier(&i2c\_bus\_type, &i2cdev\_notifier);   |  | | --- | | static struct notifier\_block i2cdev\_notifier = {  .notifier\_call = i2cdev\_notifier\_call,  }; |   4. 遍历i2c bus上的所有具有i2c\_adapter\_type属性的dev, 创建/dev/i2c[x]设备  |--i2c\_for\_each\_dev(NULL, i2cdev\_attach\_adapter);  |--bus\_for\_each\_dev(&i2c\_bus\_type, NULL, NULL, i2cdev\_attach\_adapter);  |--struct klist\_iter i;  |--klist\_iter\_init\_node(&bus->p->klist\_devices, &i,(start ? &start->p->knode\_bus : NULL));  |--while ((dev = next\_device(&i)) && !error)  |--i2cdev\_attach\_adapter(dev, NULL);  |--klist\_iter\_exit(&i); |
| i2cdev\_attach\_adapter(struct device \*dev, void \*dummy)  1. 只处理i2c\_adapter\_type类型的设备 : i2c0/1/2/3  |--if (dev->type != &i2c\_adapter\_type)  |--return 0  2. 创建/dev/i2c[x]设备节点  |--adap = to\_i2c\_adapter(dev);  |--i2c\_dev = get\_free\_i2c\_dev(adap);  |--i2c\_dev->dev = device\_create(i2c\_dev\_class, &adap->dev, MKDEV(I2C\_MAJOR, adap->nr), NULL, "i2c-%d", adap->nr); |

## i2cdev\_notifier\_call

|  |
| --- |
| i2cdev\_notifier\_call(struct notifier\_block \*nb, unsigned long action, void \*data)  {  |--switch (action) {  |--case BUS\_NOTIFY\_ADD\_DEVICE: //i2c\_adapter设备注册时，device\_register()会上报这个事件  |--i2cdev\_attach\_adapter(dev, NULL);  |--case BUS\_NOTIFY\_DEL\_DEVICE:  |--i2cdev\_detach\_adapter(dev, NULL); |
| i2cdev\_attach\_adapter(struct device \*dev, void \*dummy)  1. 只处理i2c\_adapter\_type类型的设备 : i2c0/1/2/3  |--if (dev->type != &i2c\_adapter\_type)  |--return 0  2. 创建/dev/i2c[x]设备节点  |--adap = to\_i2c\_adapter(dev);  |--i2c\_dev = get\_free\_i2c\_dev(adap);  |--i2c\_dev->dev = device\_create(i2c\_dev\_class, &adap->dev, MKDEV(I2C\_MAJOR, adap->nr), NULL, "i2c-%d", adap->nr); |

## i2cdev\_open

|  |
| --- |
| i2cdev\_open(struct inode \*inode, struct file \*file)  1. 根据inode获取adap  |--unsigned int minor = iminor(inode);  |--i2c\_dev = i2c\_dev\_get\_by\_minor(minor);  |--adap = i2c\_get\_adapter(i2c\_dev->adap->nr);  2. 创建i2c\_client  |--struct i2c\_client \*client = kzalloc(sizeof(\*client), GFP\_KERNEL);  |--snprintf(client->name, I2C\_NAME\_SIZE, "i2c-dev %d", adap->nr);  |--client->adapter = adap;  3. 保存i2c\_client到file中  |--file->private\_data = client; |

## i2cdev\_write/ i2cdev\_read

|  |
| --- |
| i2cdev\_write(struct file \*file, const char \_\_user \*buf, size\_t count, loff\_t \*offset)  |--struct i2c\_client \*client = file->private\_data;  |--char \*tmp = memdup\_user(buf, count);  |--i2c\_master\_send(client, tmp, count);  |--kfree(tmp); |
| i2cdev\_read(struct file \*file, char \_\_user \*buf, size\_t count, loff\_t \*offset)  |--struct i2c\_client \*client = file->private\_data;  |--char \*tmp = tmp = kmalloc(count, GFP\_KERNEL);  |--i2c\_master\_recv(client, tmp, count);  |--copy\_to\_user(buf, tmp, count);  |--kfree(tmp); |

## i2cdev\_ioctl

|  |
| --- |
| i2cdev\_ioctl(struct file \*file, unsigned int cmd, unsigned long arg)  |--struct i2c\_client \*client = file->private\_data;  |--switch (cmd)  |--case I2C\_SLAVE:  |--case I2C\_SLAVE\_FORCE:  |--client->addr = arg; return;  |--case I2C\_TENBIT:  |--client->flags |= I2C\_CLIENT\_PEC; return;  |--case I2C\_FUNCS:  |--funcs = i2c\_get\_functionality(client->adapter);  |--put\_user(funcs, (unsigned long \_\_user \*)arg); return;  |--case I2C\_RDWR:  |--i2cdev\_ioctl\_rdrw(client, arg);  |--I2C\_SMBUS:  |--i2cdev\_ioctl\_smbus(client, arg);  |--I2C\_RETRIES:  |--client->adapter->retries = arg;  |--I2C\_TIMEOUT:  |--client->adapter->timeout = msecs\_to\_jiffies(arg \* 10); |

# 芯片寄存器

## I2Cx\_CMD

|  |
| --- |
| **b[0] : SBE (Start bit enable : 1)**  #define **I2C\_CMD\_SBE** (0x1 << 0)  **b[3:1] : AS (Address select: 001)**  #define **I2C**\_**CMD**\_**AS**(x) (((x) & 0x7) << 1)  The address include slave address and slave internal memory address.  Start 命令后面跟着的地址域。  000: no address  001: 1 byte address  010: 2 byte address  011: 3 byte address  100: 4 byte address  101: 5 byte address  110: 6 byte address  111: 7 byte address  **b[4] : RBE (Restart bit enable : 0)**  #define **I2C\_CMD\_RBE** (0x1 << 4)  0: not send restart bit  1: send restart bit  **b[7:5] : SAS (Second address select)**  #define **I2C**\_**CMD**\_**SAS**(x) (((x) & 0x7) << 5)  Restart 命令后面跟着的地址域。  000: no address  001: 1 byte address  010: 2 byte address  011: 3 byte address  100: 4 byte address  101: 5 byte address  110: 6 byte address  111: 7 byte address  **b[8] : DE (Data enable : 1)**  #define **I2C**\_**CMD**\_**DE** (0x1 << 8)  The counts of data transmitted depend on the I2Cx\_CNT register  **b[9] : BS (NACK select : 0: not select, 1: select)**  #define **I2C**\_**CMD**\_**NS** (0x1 << 9)  generate the NACK signal at 9th clock of SCL of the last byte when read data  **b[10] : SE (Stop enable)**  #define **I2C**\_**CMD**\_**SE** (0x1 << 10)  **b[11] : MSS (Master or slave mode select : 0: slave mode, 1: Master mode)**  #define **I2C**\_**CMD**\_**MSS** (0x1 << 11)  **b[12] : WRS (Write or Read select: 0: write, 1: read)**  #define **I2C**\_**CMD**\_**WRS** (0x1 << 12)  作主设备时的读或写标志由填进去的跟在 Start Bit 后的从地址的 Bit0 来判别。  **b[15] : SECL (Start to execute the command list : 0: not execute, 1: execute command)**  #define **I2C**\_**CMD**\_**EXEC** (0x1 << 15)  如果没有使能该位，则 FIFO 不可用，但 I2C 模块的原来非 FIFO 的那一套可以使用。 |
|  |
|  |
|  |
|  |
|  |
|  |

# end