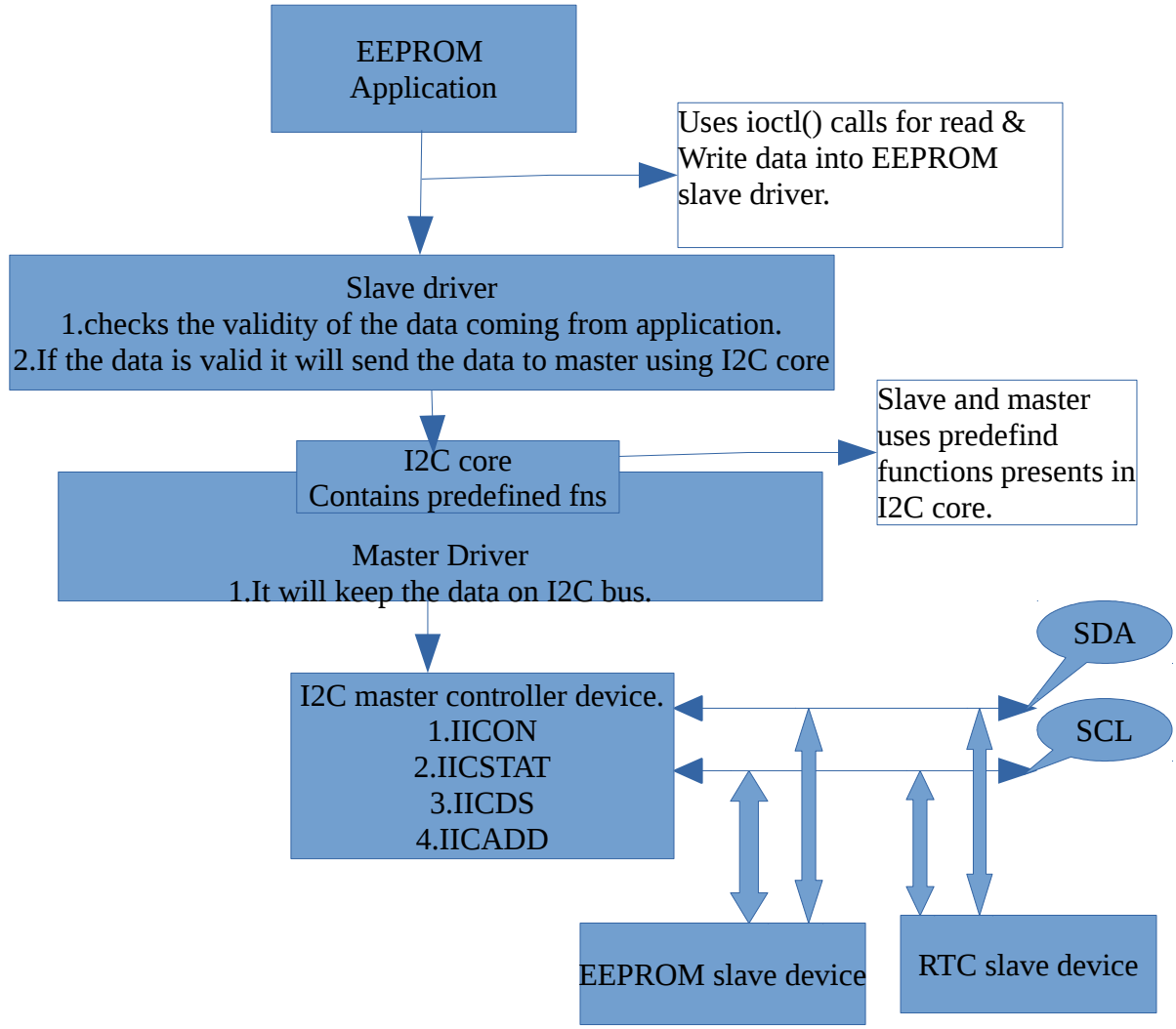
i2c Driver



Prerequisites to write a driver code:

1. Get the base address of the Master control register and IRQ line no from data sheet.

2. All these offsets of base address of registers are created as macros which are defined in **/linux-2.6.29/arch/arm/plat-s3c/include/plat/regs-iic.h.** The base address of the 1st register is defined in **linux-2.6.29/arch/arm/mach-s3c2410/include/mach/map.h** as S3C_PA_IIC = 0x54000000.
IRQ no macro is defined in **/linux-2.6.29/arch/arm/mach-s3c2410/include/mach/irqs.h.**

3. It is mandatory to create an object of type platform_device. During bootup time all platform device structures are maintained as a list of type struct platform_device using **platform_add_devices(smdk2410_devices, ARRAY_SIZE(smdk2410_devices)).** present in the file **/linux-2.6.29/arch/arm/mach-s3c2410/mach-smdk2410.c.**

```
static struct platform_device *smdk2410_devices[] __initdata = {
        &s3c_device_usb,
        &s3c_device_lcd,
        &s3c_device_wdt,
        &s3c_device_i2c0,
```

```
        &s3c_device_iis,
};
```

For I2C we have created this object in l**inux2.6.29/arch/arm/plat_s3c/dev_i2c0.c.**

In this object we need to initialize 4 members.

```
struct platform_device s3c_device_i2c0 = {
        .name           = "s3c2410-i2c",
#ifdef CONFIG_S3C_DEV_I2C1
        .id             = 0,
#else
        .id             = -1,
#endif
        .num_resources          = ARRAY_SIZE(s3c_i2c_resource),
        .resource       = s3c_i2c_resource,
};
```

Here the member resource has the base address of structural array is of type struct resource as given below which is defined in the same file **dev_i2c0.c**

```
static struct resource s3c_i2c_resource[] = {
        [0] = {
                .start = S3C_PA_IIC,
                .end   = S3C_PA_IIC + SZ_4K - 1,
                .flags = IORESOURCE_MEM,
        },
        [1] = {
                .start = IRQ_IIC,
                .end   = IRQ_IIC,
                .flags = IORESOURCE_IRQ,
        },
};
```

s3c_i2c_resourc[0] has base address info.
Start has base address of register.
Driver code uses the flag(IORESOURCE_MEM) to get the base address.

s3c_i2c_resourc[1] has IRQ line no info.
Start has the IRQ line no.
Driver code uses the flag(IORESOURCE_IRQ) to get the irq no.

Master Driver:

- master driver code is present in **linux2.6.29/drivers/i2c/buses/i2c-s3c2410.c.**
- The master driver initialization function base address is send as argument to subsys_initcall(i2c_adap_s3c_init) for master drivers.

```
static int __init i2c_adap_s3c_init(void)
{
        return platform_driver_register(&s3c24xx_i2c_driver);
}
```

Here i2c_adap_s3c_init() call access the particular platform device(I2C).

In this function we need to register your driver as a platform driver using **platform_driver_register()** call. Here s3c24xx_i2c_driver is a structure of type struct platform_driver defined in the same master driver file as given below.

```
static struct platform_driver s3c24xx_i2c_driver = {
        .probe          = s3c24xx_i2c_probe,
        .remove                 = s3c24xx_i2c_remove,
        .id_table       = s3c24xx_driver_ids,
        .driver         = {
                .owner = THIS_MODULE,
                .name  = "s3c-i2c",
                .pm    = S3C24XX_DEV_PM_OPS,
        },
};
```

Here probe, remove are funtion pointers.
id_table is a structural array of type struct platform_device_id defined in the same file as given below.

```
static struct platform_device_id s3c24xx_driver_ids[] = {
        {
                .name           = "s3c2410-i2c",
                .driver_data    = TYPE_S3C2410,
        }, {
                .name           = "s3c2440-i2c",
                .driver_data    = TYPE_S3C2440,
        }, { },
};
```

- if the string (“s3c2410-i2c”)present in struct platform_driver and struct platform_device are matched then
  1. access the corresponding node in the list(to get base address and IRQ line no)
  2. probe function is invoked and pass the platform_device structure object base address as an argument. static int s3c24xx_i2c_probe(struct platform_device *pdev).

- In the probe fuction create the device context info of type struct s3c24xx_i2c *i2c. Allocate the memory dynamically for device context info using **i2c = kzalloc(sizeof(struct s3c24xx_i2c), GFP_KERNEL);**

The device context info contains the following members which is defined in the same file.

```
struct s3c24xx_i2c {
        spinlock_t              lock;
        wait_queue_head_t       wait;
        unsigned int            suspended:1;

        struct i2c_msg          *msg;
        unsigned int            msg_num;
        unsigned int            msg_idx;
        unsigned int            msg_ptr;

        unsigned int            tx_setup;
```

```
        unsigned int            irq;

        enum s3c24xx_i2c_state      state;
        unsigned long           clkrate;

        void __iomem            *regs;
        struct clk              *clk;
        struct device           *dev;
        struct resource         *ioarea;
        struct i2c_adapter      adap;

#ifdef CONFIG_CPU_FREQ
        struct notifier_block   freq_transition;
#endif
};
```

- fill the members present in adap of type struct i2c_adapter.
- Get the base address present in platform_device (resouce member). platform_get_resource() is a kernel call which is used to access members at struct platform_device object.

  res = platform_get_resource(pdev, IORESOURCE_MEM, 0);

  it returns base address of first array element of resource because the matching flag i.e, IORESOURCE_MEM present in arrays first element.

- Check whether device registers are used by any other driver or not and get exclusive access of device registers use the call request_mem_region() . request_mem_region() call is used for memory maped architectures.

  i2c->ioarea = request_mem_region(res->start, resource_size(res), pdev->name);

- How to check whether this call is success or not?
  If this call is successful an entry is created in /proc/iomem.

- platform_get_resource is a physical address but we need virtual address. Because in memory mapped architectures uses LDA and STR instructions. These instructions operates on virutual address only. For this we need to convert physical address to virtual address.

  i2c->regs = ioremap(res->start, resource_size(res)); this call will convert physical address to virtual address and is stored in regs member in device context info.

- dev_dbg(&pdev->dev, "registers %p (%p, %p)\n", i2c->regs, i2c->ioarea, res);
  dev_dbg() call internally invokes printk.

- **ret = s3c24xx_i2c_init(i2c)** hardware initialization is done in this function.
  1. In hardware initialization 5th bit(irq line enable) and 7th bit(Acknowledgement enable) of iicon register are to be enabled. iicon = S3C2410_IICCON_IRQEN | S3C2410_IICCON_ACKEN. These macros are defined in **linux-2.6.29/arch/arm/plat-s3c/include/plat/regs-iic.h.**
  2. write the slave address in IICADD register using writeb(pdata->slave_addr, i2c->regs + S3C2410_IICADD);

3. set the 5th bit and 7th bit in IICON register using writel(iicon, i2c->regs + S3C2410_IICCON);

- platform_get_irq(pdev, 0) call internally invokes platform_get_resource() is used to get the irq number and is strored in irq member present in device context info.

- After getting irq line number you need to install interrupt handler in interrupt vector table using request_irq().
  ret = request_irq(i2c->irq, s3c24xx_i2c_irq, IRQF_DISABLED,
  dev_name(&pdev->dev), i2c);
  i2c->irq is the irq line no.
  s3c24xx_i2c_irq is the interrupt handler.
  IRQF_DISABLED is flags.
  dev_name(&pdev->dev) is the driver name.
  i2c is the base addess of device context info.

- ret = i2c_add_numbered_adapter(&i2c->adap). After filling all the members of adapter pass this i2c->adap as an argument to register the master driver with i2c core by using **i2c_add_numbered_adapter.**
  i2c_add_numbered_adapter internally invokes i2c_register_adapter().

Slave driver

- slave driver code is present in **linux2.6.29/drivers/i2c/i2c-dev.c**

The slave driver initialization function base address is send as argument to module_init() for slave drivers.

module_init(i2c_dev_init);

here i2c_dev_init() is the slave initialization function. In this funtion first you have to rigister your slave driver as charecter driver for that you have to use register _chardev();

res = register_chrdev(I2C_MAJOR, "i2c", &i2cdev_fops);
this is old kernel call to registering your driver as charecter.
the device files can created dynamically by using udev (unified device model) in sys-fs file system present in root directory.
Before creating device file we have to create udev class by using class_create() in **/sys/class/i2c-dev.** udev class is of type struct class. This udev class maintain info about device or driver

i2c_dev_class = class_create(THIS_MODULE, "i2c-dev");

it will return base address stored in (i2c_dev_class) pointer is of type struct class.
THIS_MODULE gives base address where your driver is loaded.
“i2c-dev” is a name by this name create a folder in /sys/class/i2c-dev.
The opposite of class_create(); is class_destroy(); it is used in exit function.
class_destroy(i2c_dev_class);

Udev is device manager for linux kenel ,by using udev you can create or remove device nodes or device files dynamically. Udev is a deamon process or background process (process is not associate

with input or ouput called as deamon process).

Udev process depends on sysfs file system. when device is removed or added then kernel events are produced, which will notify udev process running in user space, kernel events are also known as uevents.

res = i2c_add_driver(&i2cdev_driver);
it is inline function defined in **linux2.6.29/include/linux/i2c.h**. It is responsible for doing the following fuctionalities.
It internally invokes i2c_register_driver();
registering your driver with I2C core.
Registering your driver as slave driver.
This fuction accepts single argument of type struct i2c_driver. It has 3 members. Driver member contains the driver name and attach_adapter and detach_adapter are fuction pointers.

```
static struct i2c_driver i2cdev_driver = {
        .driver = {

                .name  = "dev_driver",
        },
        .attach_adapter         = i2cdev_attach_adapter,
        .detach_adapter         = i2cdev_detach_adapter,
};
```

i2cdev_attach_adapter accepts argument of type struct i2c_adapter.

i2c_dev = get_free_i2c_dev(adap);
this call will do the following things.
1. create an object of type struct i2c_dev.
2.it contains the members

```
struct i2c_dev {
        struct list_head list;
        struct i2c_adapter *adap;
        struct device *dev;
};
```

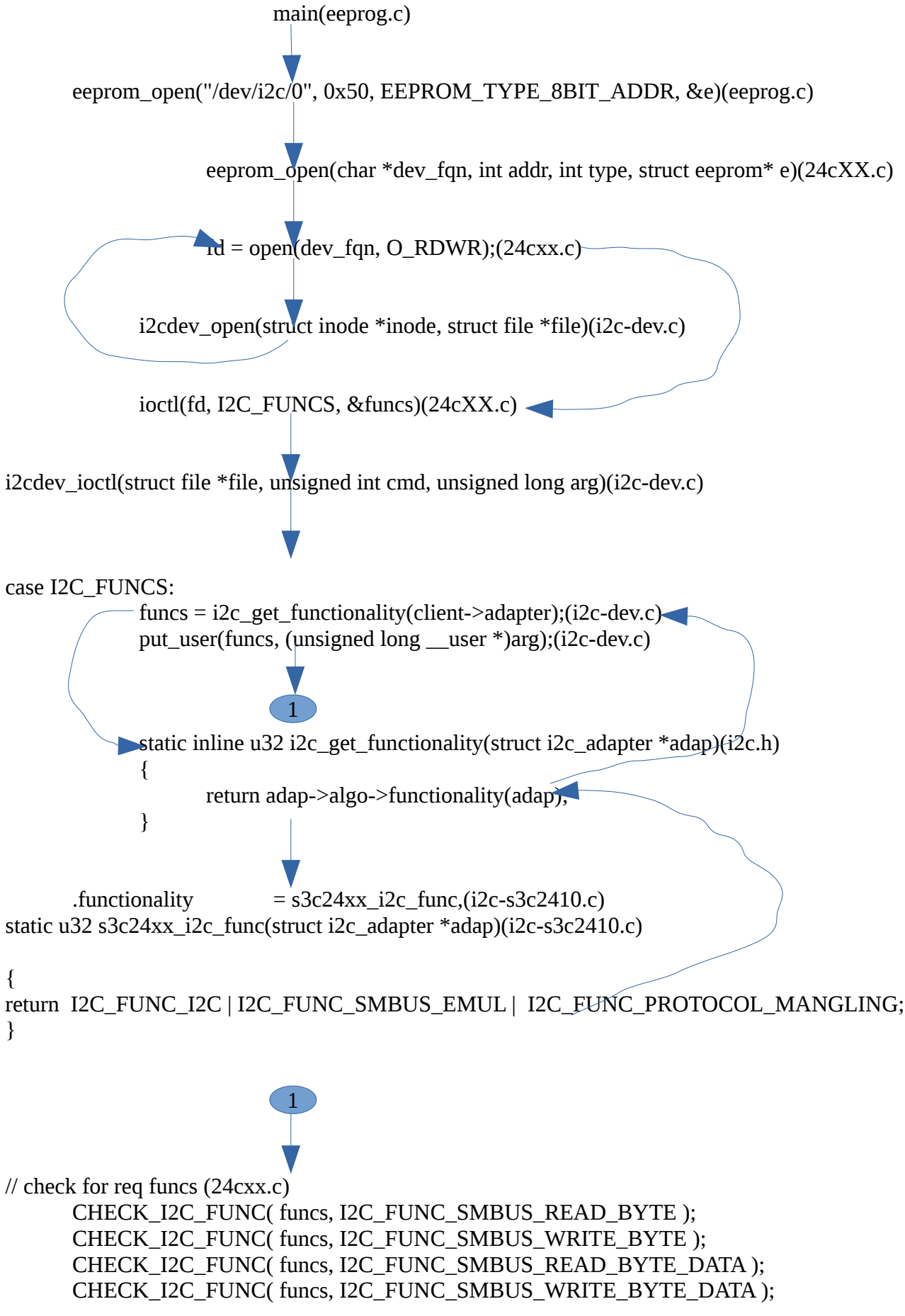
3. add this object to list.

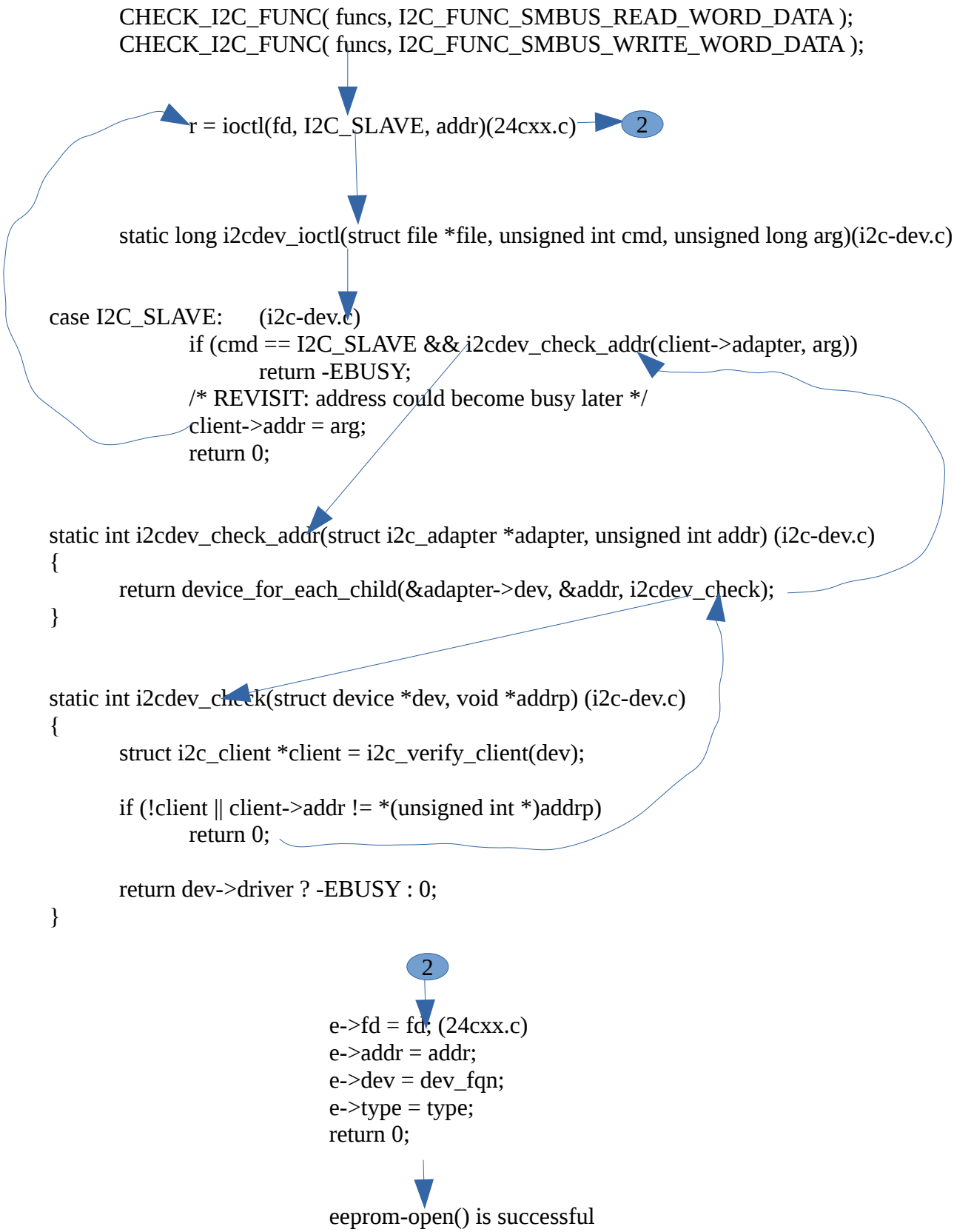
```
i2c_dev->dev = device_create(i2c_dev_class, &adap->dev,
                                MKDEV(I2C_MAJOR, adap->nr), NULL,
                                "i2c-%d", adap->nr);
```

1. it creates a device file.
2. it is responsible for creating uevents in /sys/class/i2c-dev/i2c-0/uevents. Uevents contains MAJOR NUMBER, MINOR NUMBER AND DEVNAME. Based on this infromation udev daemon creates device file. This information given to udev daemon.
3. udev daemon receives uevents from the kernel space driver after receiving uevents, will pass the uevent data(major, minor no, device file name), it will match the data with rules specified in the rules table(etc/udev/rules.d).

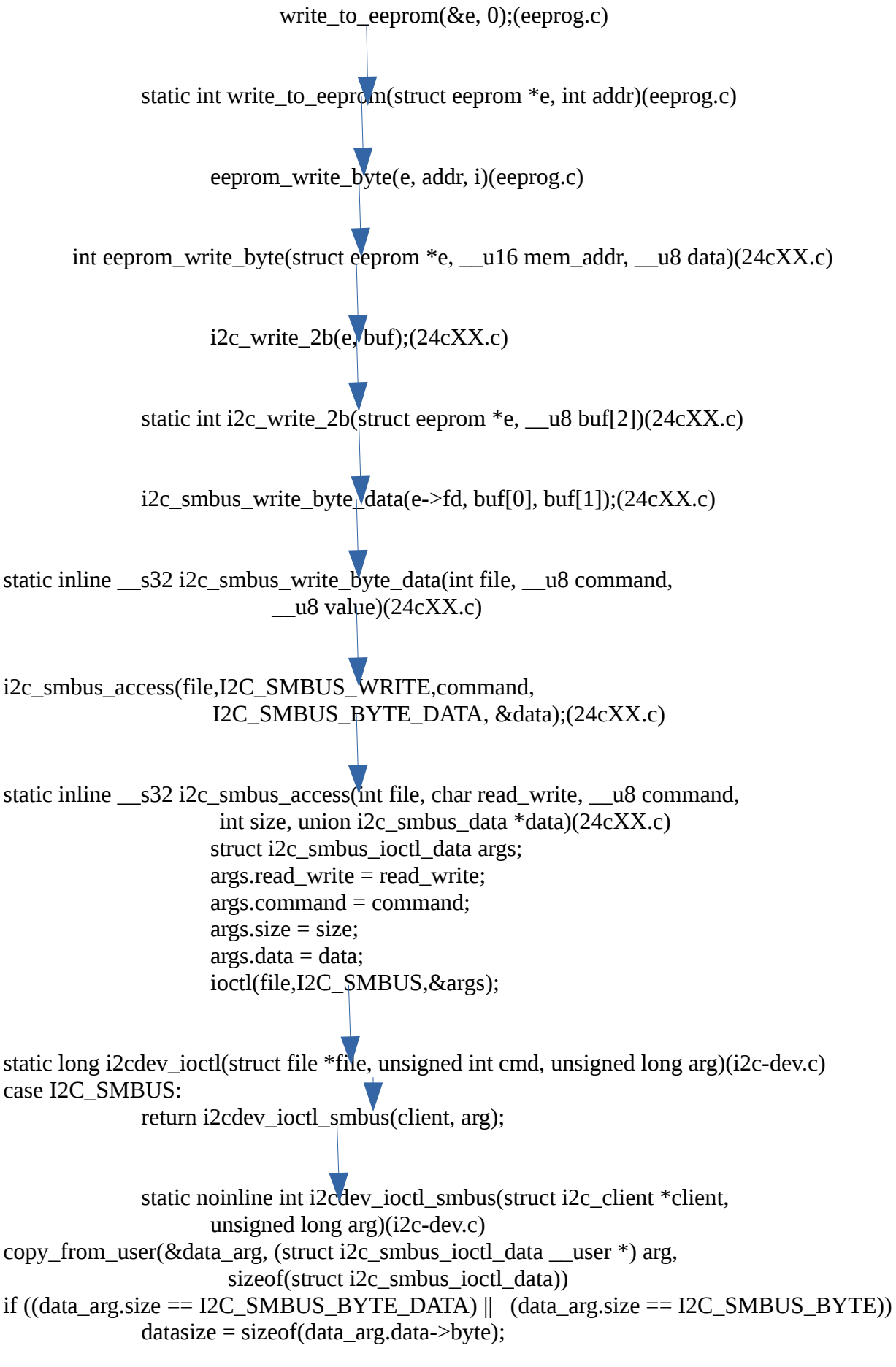
Application

EEPROM open


main(eeprog.c)
eeprom_open("/dev/i2c/0", 0x50, EEPROM_TYPE_8BIT_ADDR, &e)(eeprog.c)
eeprom_open(char *dev_fqn, int addr, int type, struct eeprom* e)(24cXX.c)
fd = open(dev_fqn, O_RDWR);(24cxx.c)
i2cdev_open(struct inode *inode, struct file *file)(i2c-dev.c)
ioctl(fd, I2C_FUNCS, &funcs)(24cXX.c)
i2cdev_ioctl(struct file *file, unsigned int cmd, unsigned long arg)(i2c-dev.c)
case I2C_FUNCS:
funcs = i2c_get_functionality(client->adapter);(i2c-dev.c)
put_user(funcs, (unsigned long __user *)arg);(i2c-dev.c)
1
static inline u32 i2c_get_functionality(struct i2c_adapter *adap)(i2c.h)
{
return adap->algo->functionality(adap);
}
.functionality = s3c24xx_i2c_func,(i2c-s3c2410.c)
static u32 s3c24xx_i2c_func(struct i2c_adapter *adap)(i2c-s3c2410.c)
{
return I2C_FUNC_I2C | I2C_FUNC_SMBUS_EMUL | I2C_FUNC_PROTOCOL_MANGLING;
}
1
// check for req funcs (24cxx.c)
CHECK_I2C_FUNC(funcs, I2C_FUNC_SMBUS_READ_BYTE);
CHECK_I2C_FUNC(funcs, I2C_FUNC_SMBUS_WRITE_BYTE);
CHECK_I2C_FUNC(funcs, I2C_FUNC_SMBUS_READ_BYTE_DATA);
CHECK_I2C_FUNC(funcs, I2C_FUNC_SMBUS_WRITE_BYTE_DATA);


CHECK_I2C_FUNC(funcs, I2C_FUNC_SMBUS_READ_WORD_DATA);
CHECK_I2C_FUNC(funcs, I2C_FUNC_SMBUS_WRITE_WORD_DATA);
r = ioctl(fd, I2C_SLAVE, addr)(24cxx.c)
2
static long i2cdev_ioctl(struct file *file, unsigned int cmd, unsigned long arg)(i2c-dev.c)
case I2C_SLAVE: (i2c-dev.c)
if (cmd == I2C_SLAVE && i2cdev_check_addr(client->adapter, arg))
return -EBUSY;
/* REVISIT: address could become busy later */
client->addr = arg;
return 0;
static int i2cdev_check_addr(struct i2c_adapter *adapter, unsigned int addr) (i2c-dev.c)
{
return device_for_each_child(&adapter->dev, &addr, i2cdev_check);
}
static int i2cdev_check(struct device *dev, void *addrp) (i2c-dev.c)
{
struct i2c_client *client = i2c_verify_client(dev);
if (!client || client->addr != *(unsigned int *)addrp)
return 0;
return dev->driver ? -EBUSY : 0;
}
2
e->fd = fd; (24cxx.c)
e->addr = addr;
e->dev = dev_fqn;
e->type = type;
return 0;
eeprom-open() is successful

EEPROM write



```
if ((data_arg.size == I2C_SMBUS_PROC_CALL) ||
        (data_arg.size == I2C_SMBUS_BLOCK_PROC_CALL) ||
        (data_arg.size == I2C_SMBUS_I2C_BLOCK_DATA) ||
        (data_arg.read_write == I2C_SMBUS_WRITE)) {
            if (copy_from_user(&temp, data_arg.data, datasize))
                return -EFAULT;
res = i2c_smbus_xfer(client->adapter, client->addr, client->flags, data_arg.read_write,
                        data_arg.command, data_arg.size, &temp);
            return res;

s32 i2c_smbus_xfer(struct i2c_adapter *adapter, u16 addr, unsigned short flags,
              char read_write, u8 command, int protocol,
              union i2c_smbus_data *data)(i2c-core.c)
res = i2c_smbus_xfer_emulated(adapter,addr,flags,read_write,
                                    command, protocol, data);

static s32 i2c_smbus_xfer_emulated(struct i2c_adapter * adapter, u16 addr, (i2c-core.c)
        unsigned short flags,  char read_write, u8 command, int size, union i2c_smbus_data * data)

            struct i2c_msg msg[2] = { { addr, flags, 1, msgbuf0 },
                    { addr, flags | I2C_M_RD, 0, msgbuf1 }   };

        case I2C_SMBUS_BYTE_DATA:
            if (read_write == I2C_SMBUS_READ)
                msg[1].len = 1;
            else {                                          Else part is executed for write
                msg[0].len = 2;
                msgbuf0[1] = data->byte;
            }
            break;

            status = i2c_transfer(adapter, msg, num);

        int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msgs, int num)
            ret = adap->algo->master_xfer(adap, msgs, num);(i2c-core.c)

        .master_xfer            = s3c24xx_i2c_xfer,(i2c-s3c2410.c)
        static int s3c24xx_i2c_xfer(struct i2c_adapter *adap, struct i2c_msg *msgs, int num)
        ret = s3c24xx_i2c_doxfer(i2c, msgs, num);

static int s3c24xx_i2c_doxfer(struct s3c24xx_i2c *i2c,  struct i2c_msg *msgs, int num)(i2c-core.c)
            ret = s3c24xx_i2c_set_master(i2c);(i2c-s3c2410.c)
                3
```

```
spin_lock_irq(&i2c->lock);

i2c->msg     = msgs;
i2c->msg_num = num;
i2c->msg_ptr = 0;
i2c->msg_idx = 0;
i2c->state   = STATE_START;

s3c24xx_i2c_enable_irq(i2c);
s3c24xx_i2c_message_start(i2c, msgs);
spin_unlock_irq(&i2c->lock);

timeout = wait_event_timeout(i2c->wait, i2c->msg_num == 0, HZ * 5);

ret = i2c->msg_idx;
```

3

```
static int s3c24xx_i2c_set_master(struct s3c24xx_i2c *i2c)(i2c-s3c2410.c)
        while (timeout-- > 0) {
        iicstat = readl(i2c->regs + S3C2410_IICSTAT);

        if (!(iicstat & S3C2410_IICSTAT_BUSBUSY))
                return 0;

        msleep(1);
}
```

In this function checking whether the bus is busy or not.

```
static inline void s3c24xx_i2c_enable_irq(struct s3c24xx_i2c *i2c) (i2c-s3c2410.c)
{
        unsigned long tmp;

        tmp = readl(i2c->regs + S3C2410_IICCON);
        writel(tmp | S3C2410_IICCON_IRQEN, i2c->regs + S3C2410_IICCON);
}
```

In this function enabling interrup enable in 5th bit of IICON register

```
static void s3c24xx_i2c_message_start(struct s3c24xx_i2c *i2c,
                                      struct i2c_msg *msg)(i2c-s3c2410.c)
        unsigned int addr = (msg->addr & 0x7f) << 1;
        s3c24xx_i2c_enable_ack(i2c);
```

4

```
stat |= S3C2410_IICSTAT_MASTER_TX;
iiccon = readl(i2c->regs + S3C2410_IICCON);
writel(stat, i2c->regs + S3C2410_IICSTAT);

dev_dbg(i2c->dev, "START: %08lx to IICSTAT, %02x to DS\n", stat, addr);
writeb(addr, i2c->regs + S3C2410_IICDS);

/* delay here to ensure the data byte has gotten onto the bus
 * before the transaction is started */
```

Enable the transmitt receive mode in 4th bit of IICSTAT register.

write the slave address in IICDS register.

```
ndelay(i2c->tx_setup);

dev_dbg(i2c->dev, "iiccon, %08lx\n", iiccon);
writel(iiccon, i2c->regs + S3C2410_IICCON);

stat |= S3C2410_IICSTAT_START;
writel(stat, i2c->regs + S3C2410_IICSTAT);
```

setting START bit writting into 5th bit of ICCSTAT.

4

```
inline void s3c24xx_i2c_enable_ack(struct s3c24xx_i2c *i2c) (i2c-s3c2410.c)
{
        unsigned long tmp;

        tmp = readl(i2c->regs + S3C2410_IICCON);
        writel(tmp | S3C2410_IICCON_ACKEN, i2c->regs + S3C2410_IICCON);
}
```

In this funtion enable the acknowledge in 7th bit of IICON register.

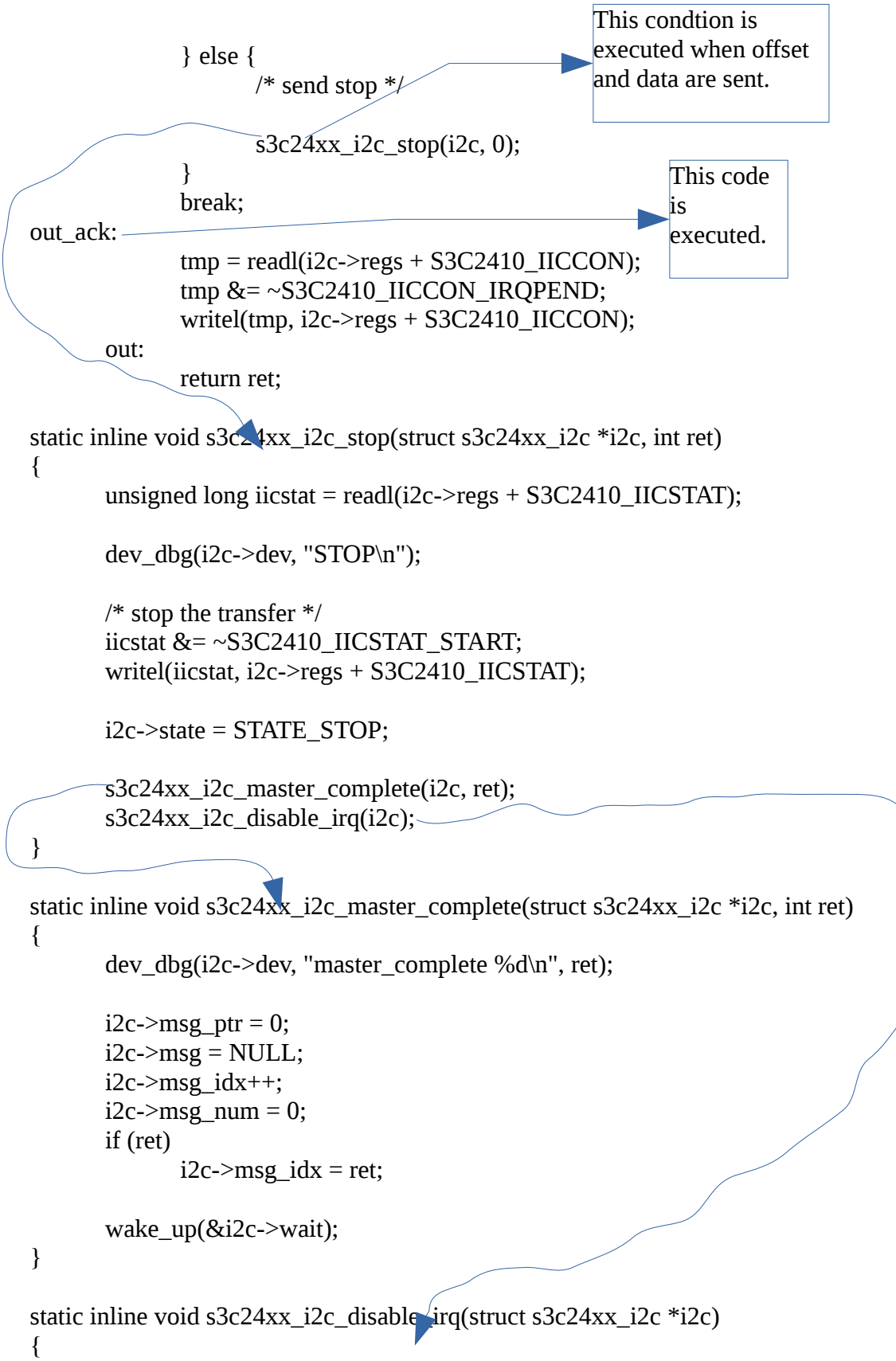
After setting START bit , automatically slave address present in IICDS register is send by master through SDAlines ,at the same time all slave device connected to IIC bus is goes to listening mode,if slave address matches with any slave device that particular slave device goes to active mode and remining all slave devices goes to inactive mode .

The slave device gives response back with ACK, if ACK on IIC bus is sensed by mater ,then interrupt handler gets invoked.

```
static irqreturn_t s3c24xx_i2c_irq(int irqno, void *dev_id)
        . i2s_s3c_irq_nextbyte(i2c, status)


static int i2s_s3c_irq_nextbyte(struct s3c24xx_i2c *i2c, unsigned long iicstat)
        switch (i2c->state)
                case STATE_START:
                        else
                        i2c->state = STATE_WRITE;
        out_ack:
                tmp = readl(i2c->regs + S3C2410_IICCON);
                tmp &= ~S3C2410_IICCON_IRQPEND;
                writel(tmp, i2c->regs + S3C2410_IICCON);
        out:
                return ret;

        case STATE_WRITE:
                /* we are writing data to the device... check for the
                 * end of the message, and if so, work out what to do
```

```
		*/

		if (!(i2c->msg->flags & I2C_M_IGNORE_NAK)) {
			if (iicstat & S3C2410_IICSTAT_LASTBIT) {
				dev_dbg(i2c->dev, "WRITE: No Ack\n");

				s3c24xx_i2c_stop(i2c, -ECONNREFUSED);
				goto out_ack;
			}
		}

retry_write:

		if (!is_msgend(i2c)) {
			byte = i2c->msg->buf[i2c->msg_ptr++];
			writeb(byte, i2c->regs + S3C2410_IICDS);

			/* delay after writing the byte to allow the
			 * data setup time on the bus, as writing the
			 * data to the register causes the first bit
			 * to appear on SDA, and SCL will change as
			 * soon as the interrupt is acknowledged */

			ndelay(i2c->tx_setup);

		} else if (!is_lastmsg(i2c)) {
			/* we need to go to the next i2c message */

			dev_dbg(i2c->dev, "WRITE: Next Message\n");

			i2c->msg_ptr = 0;
			i2c->msg_idx++;
			i2c->msg++;

			/* check to see if we need to do another message */
			if (i2c->msg->flags & I2C_M_NOSTART) {

				if (i2c->msg->flags & I2C_M_RD) {
					/* cannot do this, the controller
					 * forces us to send a new START
					 * when we change direction */

					s3c24xx_i2c_stop(i2c, -EINVAL);
				}

				goto retry_write;
			} else {
				/* send the new start */
				s3c24xx_i2c_message_start(i2c, i2c->msg);
				i2c->state = STATE_START;
			}
```

```
static inline int is_msgend(struct s3c24xx_i2c *i2c)
{
	return i2c->msg_ptr >= i2c->msg->len;
}
```




```
		} else {
			/* send stop */

			s3c24xx_i2c_stop(i2c, 0);
		}
		break;
out_ack:
		tmp = readl(i2c->regs + S3C2410_IICCON);
		tmp &= ~S3C2410_IICCON_IRQPEND;
		writel(tmp, i2c->regs + S3C2410_IICCON);
	out:
		return ret;

static inline void s3c24xx_i2c_stop(struct s3c24xx_i2c *i2c, int ret)
{
	unsigned long iicstat = readl(i2c->regs + S3C2410_IICSTAT);

	dev_dbg(i2c->dev, "STOP\n");

	/* stop the transfer */
	iicstat &= ~S3C2410_IICSTAT_START;
	writel(iicstat, i2c->regs + S3C2410_IICSTAT);

	i2c->state = STATE_STOP;

	s3c24xx_i2c_master_complete(i2c, ret);
	s3c24xx_i2c_disable_irq(i2c);
}

static inline void s3c24xx_i2c_master_complete(struct s3c24xx_i2c *i2c, int ret)
{
	dev_dbg(i2c->dev, "master_complete %d\n", ret);

	i2c->msg_ptr = 0;
	i2c->msg = NULL;
	i2c->msg_idx++;
	i2c->msg_num = 0;
	if (ret)
		i2c->msg_idx = ret;

	wake_up(&i2c->wait);
}

static inline void s3c24xx_i2c_disable_irq(struct s3c24xx_i2c *i2c)
{
	unsigned long tmp;

	tmp = readl(i2c->regs + S3C2410_IICCON);
	writel(tmp & ~S3C2410_IICCON_IRQEN, i2c->regs + S3C2410_IICCON);
}
```