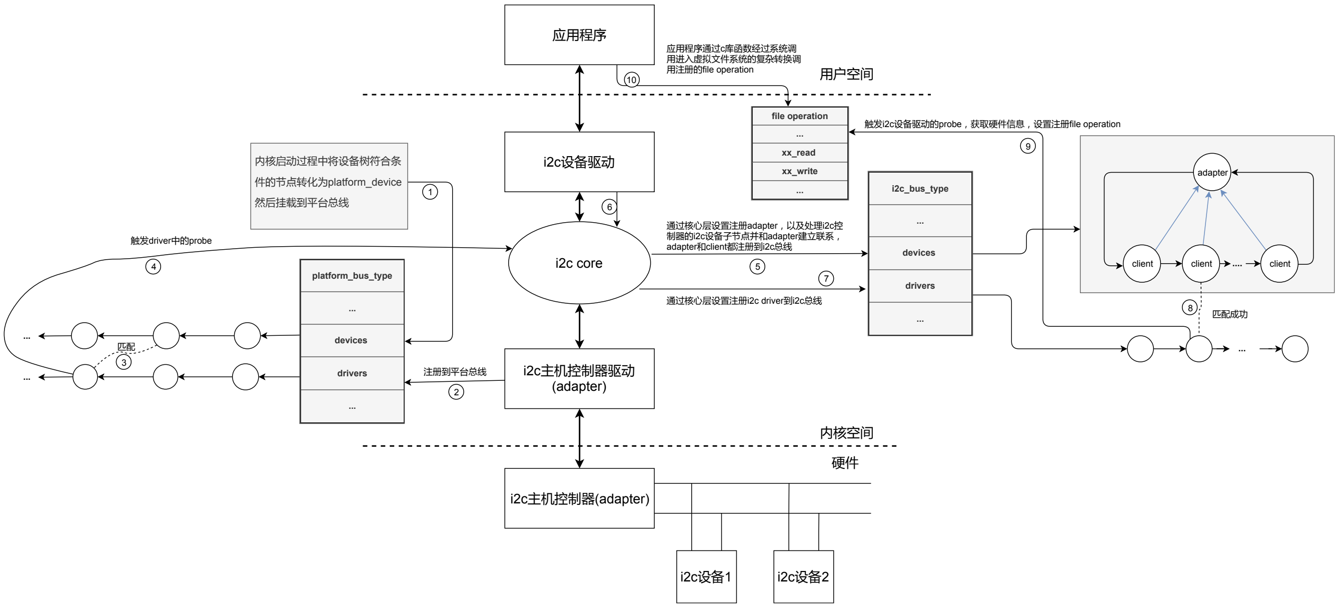
linux i2c驱动子系统分析

一、i2c驱动框架



linux的i2c驱动架构如上图所示，应用程序通过访问设备文件经过内核空间的i2c驱动访问i2c设备。重点关注内核空间的i2c驱动架构，分为3个组成部分。

1、 i2c设备驱动

调用i2c core提供的统一api控制主机控制器发出时序信号来访问i2c设备。这一层主要是实现 i2c_driver结构体，当注册i2c_driver后挂载到i2c_bus_type总线上，如果有匹配的i2c_client则调用 i2c_driver的probe函数注册包含有自定义的file operation结构体的设备结构体，当应用程序打开对应的设备文件操作时就会调用到file operation中的自定义函数，将用户数据和硬件进行交互。

2、 i2c核心(core)

i2c core提供统一的api，主要是给i2c设备驱层和控制器驱动层提供注册和注销方法，为i2c设备驱层提供统一的通信接口等。

3、 i2c主机控制器驱动(adapter)

i2c主机控制器的驱动，一般由原厂soc负责。主要设置注册adapter驱动结构体i2c_adapter，该结构体中的i2c_algorithm结构体中的master_xfer函数就是适配器用来和从设备通信的函数，负责将相应的数据根据i2c协议产生相应时序的信号和设备通信。并且主机控制器驱动会处理i2c设备信息生成相应的i2c_client结构体然后注册到i2c_bus_type总线上。

二、内核启动时对设备树的处理

1、解析处理dtb

以下两个结构体给下文引用。

```
DT_MACHINE_START(SOCFPGA, "Altera SOCFPGA")
    .smp                = smp_ops(socfpga_smp_ops),
    .map_io     = socfpga_map_io,
    .init_irq   = gic_init_irq,
    .handle_irq     = gic_handle_irq,
    .timer      = &dw_apb_timer,
    .nr_irqs        = SOCFPGA_NR_IRQS,
    .init_machine   = socfpga_cyclone5_init,
    .restart    = socfpga_cyclone5_restart,
    .reserve        = socfpga_ucosii_reserve,
    .dt_compat  = altera_dt_match,
MACHINE_END
```

```
const struct of_device_id of_default_bus_match_table[] = {
    { .compatible = "simple-bus", },
#ifdef CONFIG_ARM_AMBA
    { .compatible = "arm,amba-bus", },
#endif /* CONFIG_ARM_AMBA */
    {} /* Empty terminated list */
};
```

1.1 解析根结点的一些属性以及某些子节点

> start_kernel -> setup_arch -> setup_machine_fdt

- 根据根节点的compatile和machine_desc中的dt_compat匹配搜索最符合的machine_desc。
- 解析/chosen结点赋给boot_command_line。
- 解析根结点的#size-cells属性赋给dt_root_size_cells，#address-cells属性赋给dt_root_addr_cells。
- 解析/memory节点，将这个节点指定的区域保留使用。

```
struct machine_desc * __init setup_machine_fdt(unsigned int dt_phys)
{
    ...

    /* 映射dtb到内核虚拟地址，然后可以访问 */
    devtree = phys_to_virt(dt_phys);

    /* 检验设备树有效性 */
    if (be32_to_cpu(devtree->magic) != OF_DT_HEADER)
        return NULL;

    /* 根据根节点的compatile和machine_desc中的dt_compat匹配搜索最符合的machine_desc */
    initial_boot_params = devtree;
    dt_root = of_get_flat_dt_root();
    for_each_machine_desc(mdesc) {
        score = of_flat_dt_match(dt_root, mdesc->dt_compat);
        if (score > 0 && score < mdesc_score) {
```

```
            mdesc_best = mdesc;
            mdesc_score = score;
        }
    }
    ...


    /* 解析/chosen节点赋给boot_command_line */
    of_scan_flat_dt(early_init_dt_scan_chosen, boot_command_line);

    /*
     * 解析根结点的#size-cells属性赋给dt_root_size_cells
     * 解析根结点的#address-cells属性赋给dt_root_addr_cells
    */
    of_scan_flat_dt(early_init_dt_scan_root, NULL);

    /* 解析/memory节点，将这个节点指定的区域保留使用 */
    of_scan_flat_dt(early_init_dt_scan_memory, NULL);

    /* 返回最符合的machine_desc */
    return mdesc_best;
}
```

1.2 完全解析dtb为device_node

> start_kernel -> setup_arch -> unflatten_device_tree

- 分配内存,将dtb解析为device_node和property组成的树状结构。
- 获取/chosen节点的device_node指针赋给全局变量of_chosen，获取/aliasas节点的device_node指针赋给全局变量of_aliases。

```
void __init unflatten_device_tree(void)
{
    /* 分配内存,将dtb解析为device_node组成的树状结构 */
    __unflatten_device_tree(initial_boot_params, &allnodes,
                early_init_dt_alloc_memory_arch);

    /*
     * 获取/chosen节点的device_node指针赋给全局变量of_chosen
     * 获取/aliasas节点的device_node指针赋给全局变量of_aliases
    */
    of_alias_scan(early_init_dt_alloc_memory_arch);
}
```

2、将device_node转化为platform_device

> start_kernel -> rest_init -> kernel_init -> kernel_init_freeable -> do_basic_setup -> do_initcalls -> customize_machine -> socfpga_cyclone5_init -> of_platform_populate -> of_platform_bus_create -> of_platform_device_create_pdata

- 将根结点下的含有compatible属性的第一级子节点转化成platform_device。

- 对于compatible属性含有simple-bus、arm,amba-bus中的任意一种，对其子节点生成platform_device。

```
static void __init socfpga_cyclone5_init(void)
{
    ...
    /* 对所有满足条件的设备树子节点创建platform_device */
    of_platform_populate(NULL, of_default_bus_match_table,
        socfpga_auxdata_lookup, NULL);
    ...
}
```

```
int of_platform_populate(struct device_node *root,
            const struct of_device_id *matches,
            const struct of_dev_auxdata *lookup,
            struct device *parent)
{
    struct device_node *child;
    int rc = 0;

    /* 获取根结点device_node */
    root = root ? of_node_get(root) : of_find_node_by_path("/");
    if (!root)
        return -EINVAL;

    /* 对根节点下的含有compatible属性的子节点创建platform_device */
    for_each_child_of_node(root, child) {
        rc = of_platform_bus_create(child, matches, lookup, parent, true);
        if (rc)
            break;
    }

    of_node_put(root);
    return rc;
}
```

```
static int of_platform_bus_create(struct device_node *bus,
                  const struct of_device_id *matches,
                  const struct of_dev_auxdata *lookup,
                  struct device *parent, bool strict)
{
    const struct of_dev_auxdata *auxdata;
    struct device_node *child;
    struct platform_device *dev;
    const char *bus_id = NULL;
    void *platform_data = NULL;
    int rc = 0;

    /* 如果该节点没有compatible属性则直接返回 */
    if (strict && (!of_get_property(bus, "compatible", NULL))) {
        pr_debug("%s() - skipping %s, no compatible prop\n",
             __func__, bus->full_name);
        return 0;
```

```c
    }

    ...

    /* 对当前的节点创建platform_device */
    dev = of_platform_device_create_pdata(bus, bus_id, platform_data, parent);

    /*
     * 如果当前节点的compatile含有of_device_id of_default_bus_match_table
     * 中的simple-bus或者arm,amba-bus则对其含有compatible属性的子节点
     * 创建platform_device,否则不对子节点做platform_device创建处理直接返回。
    */
    if (!dev || !of_match_node(matches, bus))
        return 0;

    /* 到了这一步说明当前节点的compatile含有simple-bus或者arm,amba-bus，
     * 然后递归对子节点进行platform_device创建处理。
    */
    for_each_child_of_node(bus, child) {
        pr_debug("   create child: %s\n", child->full_name);
        rc = of_platform_bus_create(child, matches, lookup, &dev->dev, strict);
        if (rc) {
            of_node_put(child);
            break;
        }
    }
    return rc;
}
```

```c
struct platform_device *of_platform_device_create_pdata(
                    struct device_node *np,
                    const char *bus_id,
                    void *platform_data,
                    struct device *parent)
{
    struct platform_device *dev;

    /* 如果status属性不是okay则不创建platform_device */
    if (!of_device_is_available(np))
        return NULL;
    /*
     * 对device_node分配platform_device，针对device_node的property
     * 设置资源platform_device->resource
    */
    dev = of_device_alloc(np, bus_id, parent);
    if (!dev)
        return NULL;
    ...
    /* 设置platform_device挂载的bus为platform_bus_type */
    dev->dev.bus = &platform_bus_type;
    dev->dev.platform_data = platform_data;

    /* 注册platform_device */
```

```
    if (of_device_add(dev) != 0) {
        platform_device_put(dev);
        return NULL;
    }

    return dev;
}
```

```
int of_device_is_available(const struct device_node *device)
{
    const char *status;
    int statlen;

    /* 获取status属性 */
    status = of_get_property(device, "status", &statlen);
    if (status == NULL)
        return 1;

    /* 如果status属性是okay或者ok则返回1，表示成功 */
    if (statlen > 0) {
        if (!strcmp(status, "okay") || !strcmp(status, "ok"))
            return 1;
    }

    return 0;
}
```

```
struct platform_device *of_device_alloc(struct device_node *np,
                  const char *bus_id,
                  struct device *parent)
{
    struct platform_device *dev;
    int rc, i, num_reg = 0, num_irq;
    struct resource *res, temp_res;

    /* 分配platform_device空间 */
    dev = platform_device_alloc("", -1);
    if (!dev)
        return NULL;

    /* 计算io和irq资源数 */
    if (of_can_translate_address(np))
        while (of_address_to_resource(np, num_reg, &temp_res) == 0)
            num_reg++;
    num_irq = of_irq_count(np);

    /* 根据资源数分配platform_device的资源表空间，并解析生成其中的值 */
    if (num_irq || num_reg) {
        res = kzalloc(sizeof(*res) * (num_irq + num_reg), GFP_KERNEL);
        if (!res) {
            platform_device_put(dev);
            return NULL;
```

```
        }

        dev->num_resources = num_reg + num_irq;
        dev->resource = res;
        for (i = 0; i < num_reg; i++, res++) {
            rc = of_address_to_resource(np, i, res);
            WARN_ON(rc);
        }
        WARN_ON(of_irq_to_resource_table(np, res, num_irq) != num_irq);
    }

    /* 根据device可以找到对应的device_node */
    dev->dev.of_node = of_node_get(np);

    ...
}
```

三、I2C主机控制器驱动

1、注册驱动到平台总线

> start_kernel -> rest_init -> kernel_init -> kernel_init_freeable -> do_basic_setup -> do_initcalls -> dw_i2c_init_driver -> platform_driver_probe -> platform_driver_register -> driver_register -> bus_add_driver -> driver_attach -> driver_match_device -> driver_probe_device -> really_probe

- 注册platform_driver
- 发生匹配后调用dw_i2c_probe

```
static const struct of_device_id dw_i2c_of_match[] = {
    { .compatible = "snps,designware-i2c", },
    {},
};
MODULE_DEVICE_TABLE(of, dw_i2c_of_match);

static struct platform_driver dw_i2c_driver = {
    .remove     = __devexit_p(dw_i2c_remove),
    .driver     = {
        .name   = "i2c_designware",
        .owner  = THIS_MODULE,
        .of_match_table = of_match_ptr(dw_i2c_of_match),
        .pm = &dw_i2c_dev_pm_ops,
    },
};

static int __init dw_i2c_init_driver(void)
{
    return platform_driver_probe(&dw_i2c_driver, dw_i2c_probe);
}
subsys_initcall(dw_i2c_init_driver);
```

```
i2c0: i2c@ffc04000 {
    #address-cells = <1>;
    #size-cells = <0>;
    compatible = "snps,designware-i2c";
    reg = <0xffc04000 0x1000>;
    interrupts = <0 158 4>;
    ...
    status = "okay";
    ...
};
```

dw_i2c_init_driver通过宏subsys_initcall会链接到.initcall4.init的段中，会在do_initcalls中被调用。将dw_i2c_driver注册到平台总线platform_bus_type，由上文内核对设备树的处理分析知道dtb转化成device_node在do_initcalls之前完成，而device_node转化成platform_device也在do_initcalls中进行，但是customize_machine是用arch_initcall宏定义的，会链接在.initcall3.init中，级别比.initcall4.init高，因此也会在dw_i2c_init_driver被调用前完成，因此驱动加载前内核会生成一个i2c0主机控制器的platform_device。该控制器设备树的compatile属性和驱动的of_match_table中的内容都是"snps,designware-i2c"，所以通过dw_i2c_init_driver调用到后面的platform_driver_register将平台驱动注册到平台总线后(如下代码所示)会匹配，内核会在当匹配到对应的platform_device时，最终调用到dw_i2c_probe。

```
int platform_driver_register(struct platform_driver *drv)
{
    drv->driver.bus = &platform_bus_type;
    if (drv->probe)
        drv->driver.probe = platform_drv_probe;
    if (drv->remove)
        drv->driver.remove = platform_drv_remove;
    if (drv->shutdown)
        drv->driver.shutdown = platform_drv_shutdown;

    return driver_register(&drv->driver);
}
```

2、 匹配到对应的platform_device时触发probe

> really_probe -> dw_i2c_probe -> i2c_add_numbered_adapter -> i2c_register_adapter -> device_register

> really_probe -> dw_i2c_probe -> of_i2c_register_devices -> i2c_new_device -> device_register

- 从platform_device资源表或者device_node中获取io，irq，速度模式等，然后初始化控制器。
- 设置注册i2c_adapter到i2c_bus_type总线上，adapter中设置的重点成员是algorithm，负责产生i2c时序和附属的i2c设备进行数据传输。
- 处理控制器下的i2c设备子节点，生成i2c_client，并和i2c_adapter建立联系，和i2c_adapter一样注册到i2c_bus_type总线上。

```
static int __devinit dw_i2c_probe(struct platform_device *pdev)
{
    struct dw_i2c_dev *dev;
    struct device_node *np = pdev->dev.of_node;
    struct i2c_adapter *adap;
    struct resource *mem, *ioarea;
```

```
int irq, r;
int speed, speed_prop, ret;

/* 获取内存资源，包括io */
mem = platform_get_resource(pdev, IORESOURCE_MEM, 0);
if (!mem) {
    dev_err(&pdev->dev, "no mem resource?\n");
    return -EINVAL;
}

/* 获取irq */
irq = platform_get_irq(pdev, 0);
if (irq < 0) {
    dev_err(&pdev->dev, "no irq resource?\n");
    return irq; /* -ENXIO */
}

...

/* 获取速度模式 */
speed = DW_IC_CON_SPEED_FAST;
if (np) {
    ret = of_property_read_u32(np, "speed-mode", &speed_prop);
    if (!ret && (speed_prop == 0))
        speed = DW_IC_CON_SPEED_STD;
}

...
/* 根据得到的设置值初始化i2c控制器 */
r = i2c_dw_init(dev);
if (r)
    goto err_iounmap;

/* 注册中断处理函数 */
i2c_dw_disable_int(dev);
r = request_irq(dev->irq, i2c_dw_isr, IRQF_DISABLED, pdev->name, dev);
if (r) {
    dev_err(&pdev->dev, "failure requesting irq %i\n", dev->irq);
    goto err_iounmap;
}

/* 设置adapter */
adap = &dev->adapter;
i2c_set_adapdata(adap, dev);
adap->owner = THIS_MODULE;
adap->class = I2C_CLASS_HWMON;
strlcpy(adap->name, "Synopsys DesignWare I2C adapter",
        sizeof(adap->name));
adap->algo = &i2c_dw_algo;  //设置algorithm结构成员
adap->dev.parent = &pdev->dev;
adap->dev.of_node = pdev->dev.of_node;

adap->nr = pdev->id;
```

```
    /* 注册adapter */
    r = i2c_add_numbered_adapter(adap);
    if (r) {
        dev_err(&pdev->dev, "failure adding adapter\n");
        goto err_free_irq;
    }

    /*
     * 处理该i2c控制器下的i2c设备子节点生成对应的i2c_client，
     * 和控制器对应的adapter关联，和adapter一样注册到i2c_bus_type
     */
    of_i2c_register_devices(adap);

    return 0;

    ...
}
```

```
void of_i2c_register_devices(struct i2c_adapter *adap)
{
    void *result;
    struct device_node *node;

    /* 获取i2c控制器的device_node */
    if (!adap->dev.of_node)
        return;

    dev_dbg(&adap->dev, "of_i2c: walking child nodes\n");

    /* 处理每个i2c控制器下的i2c设备节点 */
    for_each_child_of_node(adap->dev.of_node, node) {
        struct i2c_board_info info = {};
        struct dev_archdata dev_ad = {};
        const __be32 *addr;
        int len;

        ...

        /* 从i2c设备子节点device_node中提取信息填入info */
        ...
        info.addr = be32_to_cpup(addr);
        ...
        info.irq = irq_of_parse_and_map(node, 0);
        info.of_node = of_node_get(node);
        info.archdata = &dev_ad;
        ...

        /* 根据adap和info设置i2c_client，注册到i2c_bus_type */
        result = i2c_new_device(adap, &info);
        if (result == NULL) {
            dev_err(&adap->dev, "of_i2c: Failure registering %s\n",
                    node->full_name);
```

```
            of_node_put(node);
            irq_dispose_mapping(info.irq);
            continue;
        }
    }
}
```

```
struct i2c_client *
i2c_new_device(struct i2c_adapter *adap, struct i2c_board_info const *info)
{
    struct i2c_client   *client;
    int         status;

    /* 分配设置i2c_client */
    client = kzalloc(sizeof *client, GFP_KERNEL);
    if (!client)
        return NULL;
    ...
    client->adapter = adap;
    client->flags = info->flags;
    client->addr = info->addr;
    client->irq = info->irq;
    ...

    /* 设置i2c_client要挂接的总线，和adapter一样i2c_bus_type */
    client->dev.bus = &i2c_bus_type;
    ...

    /* 注册i2c_client */
    status = device_register(&client->dev);
    if (status)
        goto out_err;

    ...
}
```

四、I2C核心层和I2C设备驱动

i2c核心层为I2C设备驱动和I2C主机控制器驱动提供统一的api，主要是给i2c设备驱层和控制器驱动层提供注册和注销方法，为i2c设备驱层提供统一的通信接口等。它的作用是两者的桥梁，由内核实现，原理上并不复杂，核心思想就是设备模型，具体实现暂不分析。

而对于I2C设备驱动的分析请看linux i2c设备驱动调试一文。