本套视频面向这些学员：
1. 有 Linux 驱动开发基础的人，可以挑感兴趣的章节观看
2. 没有 Linux 驱动开发基础但是愿意学习的人,请按顺序全部观看,
   我会以比较简单的 LED 驱动为例讲解
3. 完全没有 Linux 驱动知识，又不想深入学习的人，比如应用开发人员，不得已要改改驱动,
   等全部录完后，我会更新本文档，那时再列出您需要观看的章节。

第一课．设备树的引入与体验
第 01 节_字符设备驱动程序的三种写法
a. 驱动程序编写有 3 种方法：传统方法、使用总线设备驱动模型、使用设备树
b. 这 3 种方法也核心都是一样的: 分配、设置、注册 file_operations 结构体
   这个结构体中有.open, .read, .write, .ioctl 等成员
   驱动程序要实现这些成员，在这些成员函数中操作硬件
c. 这 3 种方法的差别在于：如何指定硬件资源，比如如何指定 LED 引脚是哪个
c.1 传统方法: 在驱动程序代码中写死硬件资源, 代码简单/不易扩展
c.2 总线设备驱动模型: 把驱动程序分为两部分(platform_driver, platform_device)
   在 platform_device 中指定硬件资源,
   在 platform_driver 中分配/设置/注册 file_operations 结构体, 并从 platform_device 获得硬件资源
   特点：
   易于扩展，但是有很多冗余代码(每种配置都对应一个 platform_device 结构体),
   硬件有变动时需要重新编译内核或驱动程序。
c.3 使用设备树指定硬件资源: 驱动程序也分为两部分(platform_driver, 设备树*.dts)
   在设备树*.dts 中指定硬件资源, dts 被编译为 dtb 文件, 在启动单板时会将 dtb 文件传给内核,
   内核根据 dtb 文件分配/设置/注册多个 platform_device
   platform_driver 的编写方法跟"总线设备驱动模型"一样。
   特点：
   易于扩展，没有冗余代码
   硬件有变动时不需要重新编译内核或驱动程序，只需要提供不一样的 dtb 文件
   注: dts - device tree source // 设备树源文件
      dtb - device tree blob // 设备树二进制文件, 由 dts 编译得来
      blob - binary large object

第 02 节_字符设备驱动的传统写法
a. 分配 file_operations 结构体
b. 设置 file_operations 结构体
   该结构体中有.open,.read,.write 等成员,
   在这些成员函数中去操作硬件
c. 注册 file_operations 结构体:
   register_chrdev(major, name, &fops)
d. 入口函数: 调用 register_chrdev
e. 出口函数: 调用 unregister_chrdev

第 03 节_字符设备驱动的编译测试

第 04 节_总线设备驱动模型

a. 驱动程序分为 platform_device 和 platform_driver 两部分

platform_device : 指定硬件资源

platform_driver : 根据与之匹配的 platform_device 获得硬件资源，并分配/设置/注册 file_operations

b. 如何确定 platform_device 和 platform_driver 是否匹配?

b.1 platform_device 含有 name

b.2 platform_driver.id_table" 可能 " 指向一个数组，每个数组项都有 name，表示该 platform_driver 所能支持的 platform_device

b.3 platform_driver.driver 含有 name，表示该 platform_driver 所能支持的 platform_device

b.4 优先比较 b.1, b.2 两者的 name，若相同则表示互相匹配

b.5 如果 platform_driver.id_table 为 NULL，则比较 b.1，b.3 两者的 name，若相同则表示互相匹配

总线设备驱动模型只是一个编程技巧，它把驱动程序分为"硬件相关的部分"、"变化不大的驱动程序本身",

这个技巧并不是驱动程序的核心,

核心仍然是"分配/设置/注册 file_operations"

第 05 节_使用设备树时对应的驱动编程

a. 使用"总线设备驱动模型"编写的驱动程序分为 platform_device 和 platform_driver 两部分

platform_device : 指定硬件资源，来自.c 文件

platform_driver : 根据与之匹配的 platform_device 获得硬件资源，并分配/设置/注册 file_operations

b. 实际上 platform_device 也可以来自设备树文件.dts

dts 文件被编译为 dtb 文件,

dtb 文件会传给内核,

内核会解析 dtb 文件，构造出一系列的 device_node 结构体,

device_node 结构体会转换为 platform_device 结构体

所以: 我们可以在 dts 文件中指定资源，不再需要在.c 文件中设置 platform_device 结构体

c. "来自 dts 的 platform_device 结构体" 与 "我们写的 platform_driver" 的匹配过程:

"来自 dts 的 platform_device 结构体"里面有成员".dev.of_node"，它里面含有各种属性，比如 compatible, reg, pin

"我们写的 platform_driver"里面有成员".driver.of_match_table"，它表示能支持哪些来自于 dts 的 platform_device

如果"of_node 中的 compatible" 跟 "of_match_table 中的 compatible" 一致，就表示匹配成功，则调用 platform_driver 中的 probe 函数;

在 probe 函数中，可以继续从 of_node 中获得各种属性来确定硬件资源

第 06 节_只想使用不想深入研究怎么办

这是无水之源、无根之木,
只能寄希望于写驱动程序的人: 提供了文档/示例/程序写得好适配性强

一个写得好的驱动程序, 它会尽量确定所用资源,
只把不能确定的资源留给设备树, 让设备树来指定。

根据原理图确定"驱动程序无法确定的硬件资源", 再在设备树文件中填写对应内容
那么, 所填写内容的格式是什么?

a. 看文档: 内核 Documentation/devicetree/bindings/
b. 参考同类型单板的设备树文件
c. 网上搜索
d. 实在没办法时, 只能去研究驱动源码

第二课. 设备树的规范(dts 和 dtb)
第 01 节_DTS 格式
(1) 语法:
Devicetree node 格式:

```
[label:] node-name[@unit-address] {
    [properties definitions]
    [child nodes]
};
```

Property 格式 1:

```
[label:] property-name = value;
```

Property 格式 2(没有值):

```
[label:] property-name;
```

Property 取值只有 3 种:
arrays of cells(1 个或多个 32 位数据, 64 位数据使用 2 个 32 位数据表示),
string(字符串),
bytestring(1 个或多个字节)

示例:
a. Arrays of cells : cell 就是一个 32 位的数据

```
interrupts = <17 0xc>;
```

b. 64bit 数据使用 2 个 cell 来表示:

```
clock-frequency = <0x00000001 0x00000000>;
```

c. A null-terminated string (有结束符的字符串):
compatible = "simple-bus";

d. A bytestring(字节序列) :
local-mac-address = [00 00 12 34 56 78]; // 每个 byte 使用 2 个 16 进制数来表示
local-mac-address = [000012345678]; // 每个 byte 使用 2 个 16 进制数来表示

e. 可以是各种值的组合，用逗号隔开:
compatible = "ns16550", "ns8250";
example = <0xf00f0000 19>, "a strange property format";

(2)
DTS 文件布局(layout):
/dts-v1/;
[memory reservations] // 格式为: /memreserve/ <address> <length>;
/ {
    [property definitions]
    [child nodes]
};

(3) 特殊的、默认的属性:
a. 根节点:
#address-cells // 在它的子节点的 reg 属性中，使用多少个 u32 整数来描述地址(address)
#size-cells // 在它的子节点的 reg 属性中，使用多少个 u32 整数来描述大小(size)
compatible // 定义一系列的字符串，用来指定内核中哪个 machine_desc 可以支持本设备
                // 即这个板子兼容哪些平台
                // uImage : smdk2410 smdk2440 mini2440 ==> machine_desc

model // 咱这个板子是什么
                // 比如有 2 款板子配置基本一致，它们的 compatible 是一样的
                // 那么就通过 model 来分辨这 2 款板子

b. /memory
device_type = "memory";
reg // 用来指定内存的地址、大小

c. /chosen
bootargs // 内核 command line 参数，跟 u-boot 中设置的 bootargs 作用一样

d. /cpus
/cpus 节点下有 1 个或多个 cpu 子节点, cpu 子节点中用 reg 属性用来标明自己是哪一个 cpu
所以 /cpus 中有以下 2 个属性:

#address-cells // 在它的子节点的 reg 属性中，使用多少个 u32 整数来描述地址(address)

#size-cells // 在它的子节点的 reg 属性中，使用多少个 u32 整数来描述大小(size)
  // 必须设置为 0

e. /cpus/cpu*
device_type = "cpu";
reg // 表明自己是哪一个 cpu

(4) 引用其他节点:
a. phandle ：// 节点中的 phandle 属性，它的取值必须是唯一的(不要跟其他的 phandle 值一样)

```
pic@10000000 {
    phandle = <1>;
    interrupt-controller;
};

another-device-node {
    interrupt-parent = <1>;    // 使用 phandle 值为 1 来引用上述节点
};
```

b. label:

```
PIC: pic@10000000 {
    interrupt-controller;
};

another-device-node {
    interrupt-parent = <&PIC>;   // 使用 label 来引用上述节点,
                                 // 使用 lable 时实际上也是使用 phandle 来引用,
                                 // 在编译 dts 文件为 dtb 文件时，编译器 dtc 会在 dtb 中插入 phandle 属性
};
```

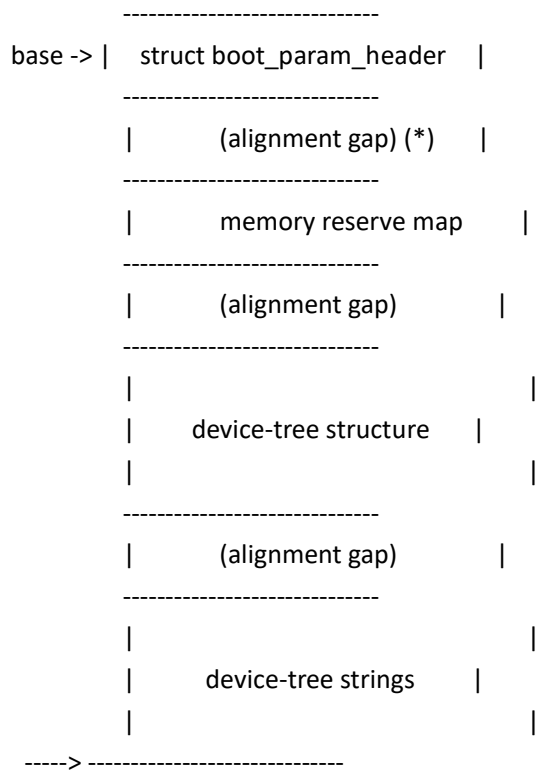
官方文档:
https://www.devicetree.org/specifications/

第 02 节_DTB 格式
官方文档:
https://www.devicetree.org/specifications/

内核文档:

Documentation/devicetree/booting-without-of.txt

DTB 文件布局:



第三课. 内核对设备树的处理
Linux uses DT data for three major purposes:
1) platform identification,
2) runtime configuration, and
3) device population.

第 01 节_从源头分析_内核 head.S 对 dtb 的简单处理

bootloader 启动内核时,会设置 r0,r1,r2 三个寄存器,
r0 一般设置为 0;
r1 一般设置为 machine id (在使用设备树时该参数没有被使用);
r2 一般设置 ATAGS 或 DTB 的开始地址

bootloader 给内核传递的参数时有 2 种方法:
ATAGS 或 DTB

对于 ATAGS 传参方法, 可以参考我们的"毕业班视频-自己写 bootloader"

从 www.100ask.net 下载页面打开百度网盘,
打开如下目录:
100ask 分享的所有文件
006_u-boot_内核_根文件系统(新 1 期_2 期间的衔接)
视频
第 002 课_从 0 写 bootloader_更深刻理解 bootloader

a. __lookup_processor_type：使用汇编指令读取 CPU ID，根据该 ID 找到对应的 proc_info_list 结构体(里面含有这类 CPU 的初始化函数、信息)
b. __vet_atags ：判断是否存在可用的 ATAGS 或 DTB
c. __create_page_tables ：创建页表，即创建虚拟地址和物理地址的映射关系
d. __enable_mmu ：使能 MMU，以后就要使用虚拟地址了
e. __mmap_switched ：上述函数里将会调用__mmap_switched
f. 把 bootloader 传入的 r2 参数，保存到变量__atags_pointer 中
g. 调用 C 函数 start_kernel

head.S/head-common.S ：
把 bootloader 传来的 r1 值，赋给了 C 变量: __machine_arch_type
把 bootloader 传来的 r2 值，赋给了 C 变量: __atags_pointer // dtb 首地址

第 02 节_对设备树中平台信息的处理(选择 machine_desc)
a. 设备树根节点的 compatible 属性列出了一系列的字符串,
表示它兼容的单板名,
从"最兼容"到次之

b. 内核中有多个 machine_desc,
其中有 dt_compat 成员，它指向一个字符串数组，里面表示该 machine_desc 支持哪些单板

c. 使用 compatile 属性的值,
跟
每一个 machine_desc.dt_compat
比较,
成绩为"吻合的 compatile 属性值的位置",

成绩越低越匹配，对应的 machine_desc 即被选中

函数调用过程:
start_kernel // init/main.c
setup_arch(&command_line); // arch/arm/kernel/setup.c
mdesc = setup_machine_fdt(__atags_pointer); // arch/arm/kernel/devtree.c

```
                    early_init_dt_verify(phys_to_virt(dt_phys)    // 判断是否有效的 dtb, drivers/of/ftd.c
                                        initial_boot_params = params;
                    mdesc = of_flat_dt_match_machine(mdesc_best, arch_get_next_mach); // 找到最匹配的 machine_desc, drivers/of/ftd.c
                                        while ((data = get_next_compat(&compat))) {
                                            score = of_flat_dt_match(dt_root, compat);
                                            if (score > 0 && score < best_score) {
                                                best_data = data;
                                                best_score = score;
                                            }
                                        }

        machine_desc = mdesc;
```

第 03 节_对设备树中运行时配置信息的处理

函数调用过程:

```
start_kernel // init/main.c
    setup_arch(&command_line);    // arch/arm/kernel/setup.c
        mdesc = setup_machine_fdt(__atags_pointer);    // arch/arm/kernel/devtree.c
                    early_init_dt_scan_nodes();          // drivers/of/ftd.c
                        /* Retrieve various information from the /chosen node */
                        of_scan_flat_dt(early_init_dt_scan_chosen, boot_command_line);

                        /* Initialize {size,address}-cells info */
                        of_scan_flat_dt(early_init_dt_scan_root, NULL);

                        /* Setup memory, calling early_init_dt_add_memory_arch */
                        of_scan_flat_dt(early_init_dt_scan_memory, NULL);
```

a. /chosen 节点中 bootargs 属性的值，存入全局变量： boot_command_line
b. 确定根节点的这 2 个属性的值: #address-cells, #size-cells
存入全局变量: dt_root_addr_cells, dt_root_size_cells
c. 解析/memory 中的 reg 属性，提取出"base, size"，最终调用 memblock_add(base, size);

第 04 节_dtb 转换为 device_node(unflatten)

函数调用过程:

```
start_kernel // init/main.c
    setup_arch(&command_line);    // arch/arm/kernel/setup.c
```

```
arm_memblock_init(mdesc);      // arch/arm/kernel/setup.c
    early_init_fdt_reserve_self();
        /* Reserve the dtb region */
        // 把 DTB 所占区域保留下来, 即调用: memblock_reserve
        early_init_dt_reserve_memory_arch(__pa(initial_boot_params),
                                          fdt_totalsize(initial_boot_params),
                                          0);
    early_init_fdt_scan_reserved_mem();   // 根据 dtb 中的 memreserve 信息, 调用 memblock_reserve

unflatten_device_tree();       // arch/arm/kernel/setup.c
    __unflatten_device_tree(initial_boot_params, NULL, &of_root,
                            early_init_dt_alloc_memory_arch,   false);                    // drivers/of/fdt.c

        /* First pass, scan for size */
        size = unflatten_dt_nodes(blob, NULL, dad, NULL);

        /* Allocate memory for the expanded device tree */
        mem = dt_alloc(size + 4, __alignof__(struct device_node));

        /* Second pass, do actual unflattening */
        unflatten_dt_nodes(blob, mem, dad, mynodes);
            populate_node
                np = unflatten_dt_alloc(mem, sizeof(struct device_node) + allocl,
                                        __alignof__(struct device_node));

                np->full_name = fn = ((char *)np) + sizeof(*np);

                populate_properties
                        pp = unflatten_dt_alloc(mem, sizeof(struct property),
                                                __alignof__(struct property));

                        pp->name      = (char *)pname;
                        pp->length = sz;
                        pp->value    = (__be32 *)val;
```

a. 在 DTB 文件中,
   每一个节点都以 TAG(FDT_BEGIN_NODE, 0x00000001)开始, 节点内部可以嵌套其他节点,
   每一个属性都以 TAG(FDT_PROP, 0x00000003)开始

b. 每一个节点都转换为一个 device_node 结构体:

```
struct device_node {
    const  char  *name;   // 来自节点中的 name 属性, 如果没有该属性, 则设为
```

"NULL"

```
        const char *type;   // 来自节点中的 device_type 属性，如果没有该属性，则设为"NULL"
        phandle phandle;
        const char *full_name;   // 节点的名字, node-name[@unit-address]
        struct fwnode_handle fwnode;

        struct   property *properties;   // 节点的属性
        struct   property *deadprops;      /* removed properties */
        struct   device_node *parent;     // 节点的父亲
        struct   device_node *child;      // 节点的孩子(子节点)
        struct   device_node *sibling;   // 节点的兄弟(同级节点)
    #if defined(CONFIG_OF_KOBJ)
        struct   kobject kobj;
    #endif
        unsigned long _flags;
        void       *data;
    #if defined(CONFIG_SPARC)
        const char *path_component_name;
        unsigned int unique_id;
        struct of_irq_controller *irq_trans;
    #endif
    };
```

c. device_node 结构体中有 properties，用来表示该节点的属性

每一个属性对应一个 property 结构体:

```
    struct property {
        char       *name;       // 属性名字，指向 dtb 文件中的字符串
        int length;            // 属性值的长度
        void       *value;     // 属性值，指向 dtb 文件中 value 所在位置，数据仍以 big endian 存储
        struct property *next;
    #if defined(CONFIG_OF_DYNAMIC) || defined(CONFIG_SPARC)
        unsigned long _flags;
    #endif
    #if defined(CONFIG_OF_PROMTREE)
        unsigned int unique_id;
    #endif
    #if defined(CONFIG_OF_KOBJ)
        struct bin_attribute attr;
    #endif
    };
```

d. 这些 device_node 构成一棵树，根节点为: of_root

第 05 节_device_node 转换为 platform_device

dts -> dtb -> device_node -> platform_device

两个问题:
a. 哪些 device_node 可以转换为 platform_device?
根节点下含有 compatile 属性的子节点
如果一个结点的 compatile 属性含有这些特殊的值("simple-bus","simple-mfd","isa","arm,amba-bus")之一，那么它的子结点(需含 compatile 属性)也可以转换为 platform_device
i2c, spi 等总线节点下的子节点，应该交给对应的总线驱动程序来处理，它们不应该被转换为 platform_device

b. 怎么转换?
platform_device 中含有 resource 数组，它来自 device_node 的 reg, interrupts 属性;
platform_device.dev.of_node 指向 device_node，可以通过它获得其他属性

本节总结:

a. 内核函数 of_platform_default_populate_init，遍历 device_node 树，生成 platform_device
b. 并非所有的 device_node 都会转换为 platform_device
只有以下的 device_node 会转换:
b.1 该节点必须含有 compatible 属性
b.2 根节点的子节点(节点必须含有 compatible 属性)
b.3 含有特殊 compatible 属性的节点的子节点(子节点必须含有 compatible 属性):
这些特殊的 compatilbe 属性为: "simple-bus","simple-mfd","isa","arm,amba-bus"

b.4 示例:
比如以下的节点,
/mytest 会被转换为 platform_device,
因为它兼容"simple-bus"，它的子节点/mytest/mytest@0 也会被转换为 platform_device

/i2c 节点一般表示 i2c 控制器，它会被转换为 platform_device，在内核中有对应的 platform_driver;
/i2c/at24c02 节点不会被转换为 platform_device，它被如何处理完全由父节点的 platform_driver 决定，一般是被创建为一个 i2c_client。

类似的也有/spi 节点，它一般也是用来表示 SPI 控制器，它会被转换为 platform_device，在内核中有对应的 platform_driver;
/spi/flash@0 节点不会被转换为 platform_device，它被如何处理完全由父节点的 platform_driver 决定，一般是被创建为一个 spi_device。

```
/ {
        mytest {
            compatile = "mytest", "simple-bus";
            mytest@0 {
                compatile = "mytest_0";
            };
        };

        i2c {
            compatile = "samsung,i2c";
            at24c02 {
                compatile = "at24c02";
            };
        };

        spi {
            compatile = "samsung,spi";
            flash@0 {
                compatible = "winbond,w25q32dw";
                spi-max-frequency = <25000000>;
                reg = <0>;
            };
        };
    };
```

函数调用过程:

a. of_platform_default_populate_init (drivers/of/platform.c) 被调用到过程:

```
start_kernel        // init/main.c
    rest_init();
        pid = kernel_thread(kernel_init, NULL, CLONE_FS);
                kernel_init
                    kernel_init_freeable();
                        do_basic_setup();
                            do_initcalls();
                                for (level = 0; level < ARRAY_SIZE(initcall_levels) - 1;
level++)
                                    do_initcall_level(level);          //    比    如
do_initcall_level(3)

for (fn = initcall_levels[3]; fn < initcall_levels[3+1]; fn++)

do_one_initcall(initcall_from_entry(fn));   // 就是调用"arch_initcall_sync(fn)"中定义的 fn 函数
```

b. of_platform_default_populate_init (drivers/of/platform.c) 生成 platform_device 的过程:

```
of_platform_default_populate_init
    of_platform_default_populate(NULL, NULL, NULL);
        of_platform_populate(NULL, of_default_bus_match_table, NULL, NULL)
            for_each_child_of_node(root, child) {
                rc = of_platform_bus_create(child, matches, lookup, parent, true);  // 调用过程看下面
                                dev = of_device_alloc(np, bus_id, parent);      // 根据 device_node 节点的属性设置 platform_device 的 resource
                if (rc) {
                    of_node_put(child);
                    break;
                }
            }
```

c. of_platform_bus_create(bus, matches, ...)的调用过程(处理 bus 节点生成 platform_devie, 并决定是否处理它的子节点):

```
        dev = of_platform_device_create_pdata(bus, bus_id, platform_data, parent);  // 生成 bus 节点的 platform_device 结构体
        if (!dev || !of_match_node(matches, bus))  // 如果 bus 节点的 compatile 属性不吻合 matches 成表, 就不处理它的子节点
            return 0;

        for_each_child_of_node(bus, child) {      // 取出每一个子节点
            pr_debug("    create child: %pOF\n", child);
            rc = of_platform_bus_create(child, matches, lookup, &dev->dev, strict);    // 处理它的子节点, of_platform_bus_create 是一个递归调用
            if (rc) {
                of_node_put(child);
                break;
            }
        }
```

d. I2C 总线节点的处理过程:

/i2c 节点一般表示 i2c 控制器, 它会被转换为 platform_device, 在内核中有对应的 platform_driver;

platform_driver 的 probe 函数中会调用 i2c_add_numbered_adapter:

```
i2c_add_numbered_adapter    // drivers/i2c/i2c-core-base.c
    __i2c_add_numbered_adapter
        i2c_register_adapter
            of_i2c_register_devices(adap);    // drivers/i2c/i2c-core-of.c
                for_each_available_child_of_node(bus, node) {
```

client = of_i2c_register_device(adap, node);

client = i2c_new_device(adap, &info); // 设备树中的 i2c 子节点被转换为 i2c_client

}

e. SPI 总线节点的处理过程:

/spi 节点一般表示 spi 控制器，它会被转换为 platform_device，在内核中有对应的 platform_driver;

platform_driver 的 probe 函数中会调用 spi_register_master，即 spi_register_controller:

spi_register_controller // drivers/spi/spi.c

of_register_spi_devices // drivers/spi/spi.c

for_each_available_child_of_node(ctlr->dev.of_node, nc) {

spi = of_register_spi_device(ctlr, nc); // 设备树中的 spi 子节点被转换为 spi_device

spi = spi_alloc_device(ctlr);

rc = of_spi_parse_dt(ctlr, spi, nc);

rc = spi_add_device(spi);

}

第 06 节_platform_device 跟 platform_driver 的匹配

drivers/base/platform.c

a. 注册 platform_driver 的过程:

platform_driver_register

__platform_driver_register

drv->driver.probe = platform_drv_probe;

driver_register

bus_add_driver

klist_add_tail(&priv->knode_bus, &bus->p->klist_drivers); // 把 platform_driver 放入 platform_bus_type 的 driver 链表中

driver_attach

bus_for_each_dev(drv->bus, NULL, drv, __driver_attach); // 对于 plarform_bus_type 下的每一个设备，调用__driver_attach

__driver_attach

ret = driver_match_device(drv, dev); // 判断 dev 和 drv 是否匹配成功

return drv->bus->match ? drv->bus->match(dev, drv) : 1; // 调用 platform_bus_type.match

driver_probe_device(drv, dev);

really_probe

drv->probe // platform_drv_probe
platform_drv_probe
struct platform_driver *drv = to_platform_driver(_dev->driver);
drv->probe

b. 注册 platform_device 的过程:
platform_device_register
platform_device_add
device_add
bus_add_device
klist_add_tail(&dev->p->knode_bus, &bus->p->klist_devices); // 把 platform_device 放入 platform_bus_type 的 device 链表中
bus_probe_device(dev);
device_initial_probe
__device_attach
ret = bus_for_each_drv(dev->bus, NULL, &data, __device_attach_driver); // // 对于 plarform_bus_type 下的每一个 driver, 调用 __device_attach_driver
__device_attach_driver
ret = driver_match_device(drv, dev);
return drv->bus->match ? drv->bus->match(dev, drv) : 1; // 调用 platform_bus_type.match
driver_probe_device

匹配函数是 platform_bus_type.match, 即 platform_match,
匹配过程按优先顺序罗列如下:
a. 比较 platform_dev.driver_override 和 platform_driver.drv->name
b. 比较 platform_dev.dev.of_node 的 compatible 属性 和 platform_driver.drv->of_match_table
c. 比较 platform_dev.name 和 platform_driver.id_table
d. 比较 platform_dev.name 和 platform_driver.drv->name
有一个成功, 即匹配成功

昨天有学员建议加录下面这 2 节, 非常感谢他们的建议,
如果你也有建议, 欢迎告诉我.
我不担心增加工作量, 录制精品才是我的目标.
"悦己之作, 方能悦人", 如果我的产品我都不满意, 怎能让你们满意?

第 07 节_内核中设备树的操作函数

include/linux/目录下有很多 of 开头的头文件:

dtb -> device_node -> platform_device

a. 处理 DTB
of_fdt.h // dtb 文件的相关操作函数，我们一般用不到，因为 dtb 文件在内核中已经被转换为 device_node 树(它更易于使用)

b. 处理 device_node
of.h // 提供设备树的一般处理函数，比如 of_property_read_u32(读取某个属性的 u32 值), of_get_child_count(获取某个 device_node 的子节点数)
of_address.h // 地址相关的函数，比如 of_get_address(获得 reg 属性中的 addr, size 值)
of_match_device(从 matches 数组中取出与当前设备最匹配的一项)
of_dma.h // 设备树中 DMA 相关属性的函数
of_gpio.h // GPIO 相关的函数
of_graph.h // GPU 相关驱动中用到的函数，从设备树中获得 GPU 信息
of_iommu.h // 很少用到
of_irq.h // 中断相关的函数
of_mdio.h // MDIO (Ethernet PHY) API
of_net.h // OF helpers for network devices.
of_pci.h // PCI 相关函数
of_pdt.h // 很少用到
of_reserved_mem.h // reserved_mem 的相关函数

c. 处理 platform_device
of_platform.h // 把 device_node 转换为 platform_device 时用到的函数,
// 比如 of_device_alloc(根据 device_node 分配设置 platform_device),
// of_find_device_by_node (根据 device_node 查找到 platform_device),
// of_platform_bus_probe (处理 device_node 及它的子节点)
of_device.h // 设备相关的函数，比如 of_match_device

第 08 节_在根文件系统中查看设备树(有助于调试)

a. /sys/firmware/fdt // 原始 dtb 文件

hexdump -C /sys/firmware/fdt

b. /sys/firmware/devicetree // 以目录结构程现的 dtb 文件，根节点对应 base 目录，每一个节点对应一个目录，每一个属性对应一个文件

c. /sys/devices/platform // 系统中所有的 platform_device，有来自设备树的，也有来有.c 文件中注册的
对于来自设备树的 platform_device,
可以进入 /sys/devices/platform/<设备名>/of_node 查看它的设备树属性

d. /proc/device-tree 是链接文件，指向 /sys/firmware/devicetree/base

第四课. u-boot 对设备树的支持

第 01 节_传递 dtb 给内核 : r2

a. u-boot 中内核启动命令:

```
bootm <uImage_addr>                                    // 无设备树,bootm 0x30007FC0
bootm <uImage_addr> <initrd_addr> <dtb_addr>     // 有设备树
```

比如 :

```
nand read.jffs2 0x30007FC0 kernel;        // 读内核 uImage 到内存 0x30007FC0
nand read.jffs2 32000000 device_tree;   // 读 dtb 到内存 32000000
bootm 0x30007FC0 - 0x32000000              // 启动，没有 initrd 时对应参数写为"-"
```

b. bootm 命令怎么把 dtb_addr 写入 r2 寄存器传给内核?

ARM 程序调用规则(ATPCS)

```
c_function(p0, p1, p2) // p0 => r0, p1 => r1, p2 => r2
```

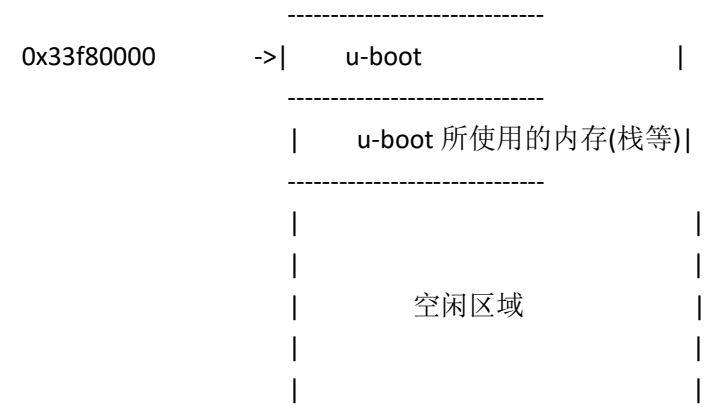
定义函数指针 the_kernel，指向内核的启动地址，
然后执行: the_kernel(0, machine_id, 0x32000000);

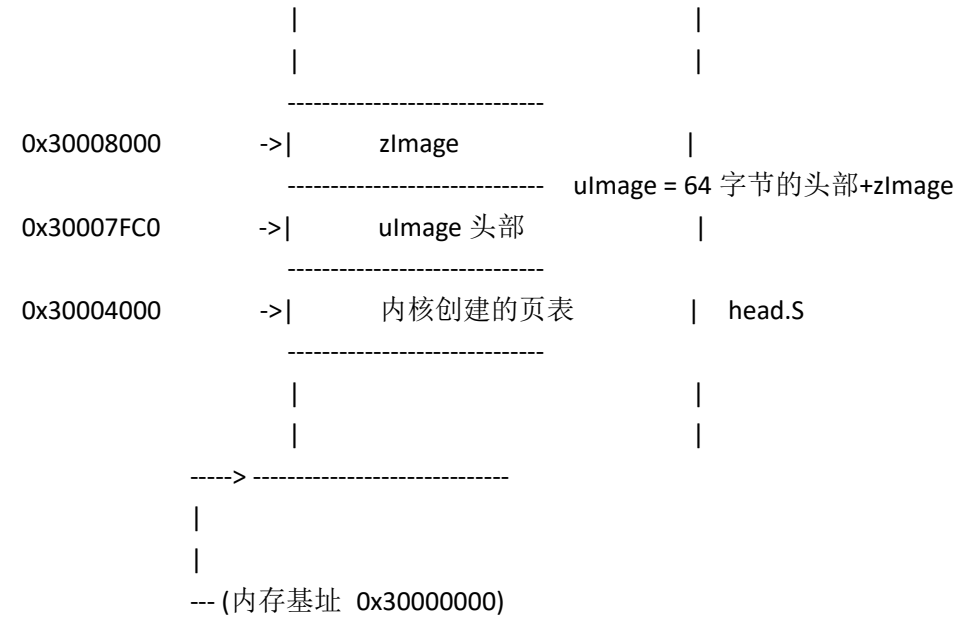
c. dtb_addr 可以随便选吗?

c.1 不要破坏 u-boot 本身

c.2 不要挡内核的路: 内核本身的空间不能占用，内核要用到的内存区域也不能占用
内核启动时一般会在它所处位置的下边放置页表，这块空间(一般是 0x4000 即 16K 字节)不能被占用

JZ2440 内存使用情况:





命令示例:
a. 可以启动:
nand read.jffs2 30000000 device_tree
nand read.jffs2 0x30007FC0 kernel
bootm 0x30007FC0 - 30000000

b. 不可以启动: 内核启动时会使用 0x30004000 的内存来存放页表，dtb 会被破坏
nand read.jffs2 30004000 device_tree
nand read.jffs2 0x30007FC0 kernel
bootm 0x30007FC0 - 30004000

第 02 节_dtb 的修改原理

例子 1. 修改属性的值,
假设 老值: len
新值: newlen (假设 newlen > len)

a. 把原属性 val 所占空间从 len 字节扩展为 newlen 字节:
把老值之后的所有内容向后移动(newlen - len)字节

b. 把新值写入 val 所占的 newlen 字节空间

c. 修改 dtb 头部信息中 structure block 的长度: size_dt_struct

d. 修改 dtb 头部信息中 string block 的偏移值: off_dt_strings

e. 修改 dtb 头部信息中的总长度: totalsize

例子 2. 添加一个全新的属性

a. 如果在 string block 中没有这个属性的名字,
   就在 string block 尾部添加一个新字符串: 属性的名
   并且修改 dtb 头部信息中 string block 的长度: size_dt_strings
   修改 dtb 头部信息中的总长度: totalsize

b. 找到属性所在节点, 在节点尾部扩展一块空间, 内容及长度为:
   TAG // 4 字节, 对应 0x00000003
   len // 4 字节, 表示属性的 val 的长度
   nameoff // 4 字节, 表示属性名的 offset
   val // len 字节, 用来存放 val

c. 修改 dtb 头部信息中 structure block 的长度: size_dt_struct

d. 修改 dtb 头部信息中 string block 的偏移值: off_dt_strings

e. 修改 dtb 头部信息中的总长度: totalsize

可以从 u-boot 官网源码下载一个比较新的 u-boot, 查看它的 cmd/fdt.c
ftp://ftp.denx.de/pub/u-boot/

fdt 命令调用过程:
   fdt set <path> <prop> [<val>]

a. 根据 path 找到节点
b. 根据 val 确定新值长度 newlen, 并把 val 转换为字节流
c. fdt_setprop
   c.1 fdt_setprop_placeholder // 为新值在 DTB 中腾出位置
      fdt_get_property_w // 得到老值的长度 oldlen
      fdt_splice_struct_ // 腾空间
         fdt_splice_ // 使用 memmove 移动 DTB 数据, 移动 (newlen-oldlen)
         fdt_set_size_dt_struct // 修改 DTB 头部, size_dt_struct
         fdt_set_off_dt_strings // 修改 DTB 头部, off_dt_strings

   c.2 memcpy(prop_data, val, len); // 在 DTB 中存入新值

第 03 节_dtb 的修改命令 fdt 移植

我们仍然使用 u-boot 1.1.6，在这个版本上我们实现了很多功能: usb 下载,菜单操作,网卡永远使能等，不忍丢弃.
需要在里面添加 fdc 命令命令，这个命令可以用来查看、修改 dtb
从 u-boot 官网下载最新的源码，把里面的 cmd/fdt.c 移植过来.

u-boot 官网源码:
ftp://ftp.denx.de/pub/u-boot/

最终的补丁存放在如下目录:
doc_and_sources_for_device_tree\source_and_images\u-boot\u-boot-1.1.6_device_tree_for_jz2440_add_fdt_20181022.patch

补丁使用方法:

```
export PATH=PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/work/system/gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabi/bin
tar xjf u-boot-1.1.6.tar.bz2                                          // 解压
cd u-boot-1.1.6
patch -p1 < ../u-boot-1.1.6_device_tree_for_jz2440_add_fdt_20181022.patch     // 打补丁
make  100ask24x0_config                                                // 配置
make
// 编译，可以得到 u-boot.bin
```

a. 移植 fdt 命令
a.1 先把代码移过去，修改 Makefile 来编译
u-boot-2018.11-rc2\lib\libfdt 主要用这个目录，
它里面的大部分文件是直接包含 scripts\dtc\libfdt 中的同名文件
只有 2 个文件是自己的版本
u-boot-2018.11-rc2\scripts\dtc\libfdt

把新 u-boot 中 cmd/fdt.c 重命名为 cmd_fdt.c ，和 lib/libfdt/* 一起复制到老 u-boot 的 common/fdt 目录
修改 老 u-boot/Makefile，添加一行: LIBS += common/fdt/libfdt.a
修改 老 u-boot/common/fdt/Makefile，仿照 drivers/nand/Makefile 来修改

a.2 根据编译的错误信息修改源码

移植时常见问题:
i. No such file or directory:

要注意,
#include "xxx.h" // 是在当前目录下查找 xxx.h
#include <xxx.h> // 是在指定目录下查找 xxx.h, 哪些指定目录呢?
// 编译文件时可以用"-I"选项指定头文件目录,
// 比如: arm-linux-gcc -I <dir> -c -o ....
// 对于 u-boot 来说, 一般就是源码的 include 目录

解决方法:
确定头文件在哪, 把它移到 include 目录或是源码的当前目录

ii. xxx undeclared :
宏, 变量, 函数未声明/未定义

对于宏, 去定义它;
对于变量, 去定义它或是声明为外部变量;
对于函数, 去实现它或是声明为外部函数;

iii. 上述 2 个错误是编译时出现的,
当一切都没问题时, 最后就是链接程序, 这时常出现: undefined reference to `xxx'
这表示代码里用到了 xxx 函数, 但是这个函数没有实现

解决方法: 去实现它, 或是找到它所在文件, 把这文件加入工程

b. fdt 命令使用示例
```
nand read.jffs2 32000000 device_tree   // 从 flash 读出 dtb 文件到内存(0x32000000)
fdt addr 32000000                      // 告诉 fdt, dtb 文件在哪
fdt print /led pin                     // 打印/led 节点的 pin 属性
fdt get value XXX /led pin             // 读取/led 节点的 pin 属性, 并且赋给环境变量 XXX
print XXX                              // 打印环境变量 XXX 的值
fdt set /led pin <0x00050005>          // 设置/led 节点的 pin 属性
fdt print /led pin                     // 打印/led 节点的 pin 属性
nand erase device_tree                 // 擦除 flash 分区
nand write.jffs2 32000000 device_tree // 把修改后的 dtb 文件写入 flash 分区
```

我给自己挖了一个大坑,
设备树课程中我想把中断讲清楚,
中断体系在 4.x 内核中变化很大, 要想彻底弄清楚设备树中对中断的描述, 必须讲中断体系;
中断体系又跟 pinctrl 系统密切相关,
pinctrl 中又涉及 GPIO 子系统.
这样讲下去的话, 设备树课程就变成驱动专题了.
所以我打算只讲中断体系统, 对于 pinctrl,gpio 等系统留待以后在驱动课程中扩展.

另一个原因是我的安卓视频推迟太久了，谢谢各位的体谅.

第五课. 中断系统中的设备树

基于设备树的 TQ2440 的中断（1）
https://www.cnblogs.com/pengdonglin137/p/6847685.html

基于设备树的 TQ2440 的中断（2）
https://www.cnblogs.com/pengdonglin137/p/6848851.html

基於 tiny4412 的 Linux 內核移植 --- 实例学习中断背后的知识(1)
http://www.cnblogs.com/pengdonglin137/p/6349209.html

Linux kernel 的中断子系统之（一）：综述
Linux kernel 的中断子系统之（二）：IRQ Domain 介绍
linux kernel 的中断子系统之（三）：IRQ number 和中断描述符
linux kernel 的中断子系统之（四）：High level irq event handler
Linux kernel 中断子系统之（五）：驱动申请中断 API
Linux kernel 的中断子系统之（六）：ARM 中断处理过程
linux kernel 的中断子系统之（七）：GIC 代码分析
http://www.wowotech.net/irq_subsystem/interrupt_subsystem_architecture.html

第 01 节_中断概念的引入与处理流程
这节视频来自"韦东山第 1 期裸板视频加强版"，如果已经理解了中断的概念，请忽略本节

第 02 节_Linux 对中断处理的框架及代码流程简述

a. 异常向量入口: arch\arm\kernel\entry-armv.S

```
    .section .vectors, "ax", %progbits
.L__vectors_start:
    W(b)    vector_rst
    W(b)    vector_und
    W(ldr)  pc, .L__vectors_start + 0x1000
    W(b)    vector_pabt
    W(b)    vector_dabt
    W(b)    vector_addrexcptn
    W(b)    vector_irq
    W(b)    vector_fiq
```

b. 中断向量: vector_irq

```
/*
```

```
 * Interrupt dispatcher
 */
    vector_stub irq, IRQ_MODE, 4     // 相当于 vector_irq: ...,
                                     // 它会根据 SPSR 寄存器的值,
                                     // 判断被中断时 CPU 是处于 USR 状态还是 SVC 状
态,
                                     // 然后调用下面的__irq_usr 或__irq_svc

    .long    __irq_usr                @   0   (USR_26 / USR_32)
    .long    __irq_invalid            @   1   (FIQ_26 / FIQ_32)
    .long    __irq_invalid            @   2   (IRQ_26 / IRQ_32)
    .long    __irq_svc                @   3   (SVC_26 / SVC_32)
    .long    __irq_invalid            @   4
    .long    __irq_invalid            @   5
    .long    __irq_invalid            @   6
    .long    __irq_invalid            @   7
    .long    __irq_invalid            @   8
    .long    __irq_invalid            @   9
    .long    __irq_invalid            @   a
    .long    __irq_invalid            @   b
    .long    __irq_invalid            @   c
    .long    __irq_invalid            @   d
    .long    __irq_invalid            @   e
    .long    __irq_invalid            @   f
```

c. __irq_usr/__irq_svc

这 2 个函数的处理过程类似:

保存现场

调用 irq_handler

恢复现场

d. irq_handler: 将会调用 C 函数 handle_arch_irq

```
    .macro    irq_handler
#ifdef CONFIG_GENERIC_IRQ_MULTI_HANDLER
    ldr r1, =handle_arch_irq
    mov r0, sp
    badr      lr, 9997f
    ldr pc, [r1]
#else
    arch_irq_handler_default
#endif
9997:
    .endm
```

e. handle_arch_irq 的处理过程: 请看视频和图片
读取寄存器获得中断信息: hwirq
把 hwirq 转换为 virq
调用 irq_desc[virq].handle_irq

对于 S3C2440, s3c24xx_handle_irq 是用于处理中断的 C 语言入口函数

中断处理流程:
假设中断结构如下:
sub int controller ---> int controller ---> cpu

发生中断时,
cpu 跳到"vector_irq", 保存现场, 调用 C 函数 handle_arch_irq

handle_arch_irq:
a. 读 int controller, 得到 hwirq
b. 根据 hwirq 得到 virq
c. 调用 irq_desc[virq].handle_irq

如果该中断没有子中断, irq_desc[virq].handle_irq 的操作:
a. 取出 irq_desc[virq].action 链表中的每一个 handler, 执行它
b. 使用 irq_desc[virq].irq_data.chip 的函数清中断

如果该中断是由子中断产生, irq_desc[virq].handle_irq 的操作:
a. 读 sub int controller, 得到 hwirq'
b. 根据 hwirq'得到 virq
c. 调用 irq_desc[virq].handle_irq

第 03 节_中断号的演变与 irq_domain

以前中断号(virq)跟硬件密切相关,
现在的趋势是中断号跟硬件无关, 仅仅是一个标号而已

以前, 对于每一个硬件中断(hwirq)都预先确定它的中断号(virq),
这些中断号一般都写在一个头文件里, 比如 arch\arm\mach-s3c24xx\include\mach\irqs.h
使用时,
a. 执行 request_irq(virq, my_handler) :
内核根据 virq 可以知道对应的硬件中断, 然后去设置、使能中断等
b. 发生硬件中断时,
内核读取硬件信息, 确定 hwirq, 反算出 virq,
然后调用 irq_desc[virq].handle_irq, 最终会用到 my_handler

怎么根据 hwirq 计算出 virq？
硬件上有多个 intc(中断控制器),
对于同一个 hwirq 数值，会对应不同的 virq
所以在讲 hwirq 时，应该强调"是哪一个 intc 的 hwirq",
在描述 hwirq 转换为 virq 时，引入一个概念：irq_domain，域，在这个域里 hwirq 转换为某一个 virq

当中断控制器越来越多、当中断越来越多，上述方法(virq 和 hwirq 固定绑定)有缺陷:
a. 增加工作量，你需要给每一个中断确定它的中断号，写出对应的宏，可能有成百上千个
b. 你要确保每一个硬件中断对应的中断号互不重复

有什么方法改进?
a. hwirq 跟 virq 之间不再绑定
b. 要使用某个 hwirq 时,
先在 irq_desc 数组中找到一个空闲项，它的位置就是 virq
再在 irq_desc[virq]中放置处理函数

新中断体系中，怎么使用中断:
a.以前是 request_irq 发起,
现在是先在设备树文件中声明想使用哪一个中断(哪一个中断控制器下的哪一个中断)

b. 内核解析设备树时,
会根据"中断控制器"确定 irq_domain,
根据"哪一个中断"确定 hwirq,
然后在 irq_desc 数组中找出一个空闲项，它的位置就是 virq
并且把 virq 和 hwirq 的关系保存在 irq_domain 中: irq_domain.linear_revmap[hwirq] = virq;

c. 驱动程序 request_irq(virq, my_handler)

d. 发生硬件中断时,
内核读取硬件信息，确定 hwirq，确定 virq = irq_domain.linear_revmap[hwirq];
然后调用 irq_desc[virq].handle_irq，最终会用到 my_handler

假设要使用子中断控制器(subintc)的 n 号中断，它发生时会导致父中断控制器(intc)的 m 号中断:
a. 设备树表明要使用<subintc n>
subintc 表示要使用<intc m>
b. 解析设备树时,

会为<subintc n>找到空闲项 irq_desc[virq'], sub irq_domain.linear_revmap[n] = virq';

会为<intc m> 找到空闲项 irq_desc[virq], irq_domain.linear_revmap[m] = virq;
并且设置它的 handle_irq 为某个分析函数 demux_func

c. 驱动程序 request_irq(virq', my_handler)

d. 发生硬件中断时,
内核读取 intc 硬件信息，确定 hwirq = m，确定 virq = irq_domain.linear_revmap[m];
然后调用 irq_desc[m].handle_irq，即 demux_func

e. demux_func:
读取 sub intc 硬件信息，确定 hwirq = n，确定 virq' = sub irq_domain.linear_revmap[n];
然后调用 irq_desc[n].handle_irq，即 my_handler

第 04 节_示例_在 S3C2440 上使用设备树描述中断体验

所用文件在: doc_and_sources_for_device_tree\source_and_images\第 5,6 课的源码及映像文件(使用了完全版的设备树)\内核补丁及设备树

先解压原始内核(source_and_images\kernel):
```
tar xzf linux-4.19-rc3.tar.gz
```
打上补丁:
```
cd linux-4.19-rc3
patch -p1 < ../linux-4.19-rc3_device_tree_for_irq_jz2440.patch
```
在内核目录下执行:
```
export
PATH=PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/work/system/gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabi/bin

cp config_ok .config
make uImage // 生成 arch/arm/boot/uImage
make dtbs // 生成 arch/arm/boot/dts/jz2440_irq.dtb
```

老内核:
```
/ # cat /proc/interrupts
           CPU0
 29:      17593          s3c  13 Edge      samsung_time_irq
 42:          0          s3c  26 Edge      ohci_hcd:usb1
 43:          0          s3c  27 Edge      s3c2440-i2c.0
 74:         86  s3c-level    0 Edge      s3c2440-uart
 75:        561  s3c-level    1 Edge      s3c2440-uart
```

```
 83:          0   s3c-level     9 Edge      ts_pen
 84:          0   s3c-level    10 Edge      adc
 87:          0   s3c-level    13 Edge      s3c2410-wdt
```

新内核：

nfs 30000000 192.168.1.124:/work/nfs_root/uImage; nfs 32000000 192.168.1.124:/work/nfs_root/jz2440_irq.dtb; bootm 30000000 - 32000000

```
/ # cat /proc/interrupts
           CPU0
  8:          0          s3c     8 Edge      s3c2410-rtc tick
 13:        936          s3c    13 Edge      samsung_time_irq
 30:          0          s3c    30 Edge      s3c2410-rtc alarm
 32:         15   s3c-level    32 Level     50000000.serial
 33:         60   s3c-level    33 Level     50000000.serial
 59:          0   s3c-level    59 Edge      53000000.watchdog
```

a. 某个设备要使用中断，需要在设备树中描述中断，如何?
   它要用哪一个中断？这个中断连接到哪一个中断控制器去?
   即：使用哪一个中断控制器的哪一个中断?

   至少有有 2 个属性:
   interrupts // 表示要使用哪一个中断，中断的触发类型等等
   interrupt-parent // 这个中断要接到哪一个设备去？即父中断控制器是谁

b. 上述的 interrupts 属性用多少个 u32 来表示?
   这应该由它的父中断控制器来描述,
   在父中断控制器中，至少有 2 个属性:
   interrupt-controller; // 表示自己是一个中断控制器
   #interrupt-cells // 表示自己的子设备里应该有几个 U32 的数据来描述中断

第 05 节_示例_使用设备树描述按键中断

第 2 期驱动: 在 linux 2.6.22.6 上编写
毕业班视频：在 linux 3.4.2 上编写
设备树视频：在 linux 4.19 上编写

基于这个驱动来修改: "第 5 课第 5 节_按键驱动_源码_设备树\000th_origin_code"，它来自毕业班视频
第 5 课第 4 节之前的内核，可以使用 "第 5 课第 5 节_按键驱动_源码_设备树\001th_buttons_drv"
第 5 课第 4 节之后的内核，可以使用 "第 5 课第 5 节_按键驱动_源码_设备树

\002th_buttons_drv"

在设备树的设备节点中描述"中断的硬件信息",
表明使用了"哪一个中断控制器里的哪一个中断，及中断触发方式",

设备节点会被转换为 platform_device,
"中断的硬件信息" 会转换为"中断号"，保存在 platform_device 的"中断资源"里,

驱动程序从 platform_device 的"中断资源"取出中断号，就可以 request_irq 了

实验:
a.
把"002th_buttons_drv/jz2440_irq.dts" 放入内核 arch/arm/boot/dts 目录,
在内核根目录下执行:
make dtbs // 得到 arch/arm/boot/dts/jz2440_irq.dtb

使用上节视频的 uImage 或这个 jz2440_irq.dtb 启动内核;

b. 编译、测试驱动:
b.1 把 002th_buttons_drv 上传到 ubuntu
b.2 编译驱动:
export
PATH=PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/work/system/gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabi/bin

cd 002th_buttons_drv
make // 得到 buttons.ko

b.3 编译测试程序:
export
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/usr/local/arm/4.3.2/bin

cd 002th_buttons_drv

arm-linux-gcc -o buttons_test buttons_test.c

b.4 测试:
insmod buttons.ko
./buttons_test &
然后按键

第 06 节_内核对设备树中断信息的处理过程

从硬件结构上看，处理过程分上下两个层面：中断控制器，使用中断的设备
从软件结构上看，处理过程分左右两个部分：在设备树中描述信息，在驱动中处理设备树
(1) 中断控制器
这又分为 root irq controller, gpf/gpg irq controller
a. root irq controller
a.1 在设备树中的描述
a.2 在内核中的驱动

b. 对于 S3C2440，还有：gpf/gpg irq controller
b.1 在设备树中的描述(在 pinctrl 节点里)
b.2 在内核中的驱动 (在 pinctrl 驱动中)

(2) 设备的中断
a.1 在设备节点中描述(表明使用"哪一个中断控制器里的哪一个中断，及中断触发方式")
a.2 在内核中的驱动 (在 platform_driver.probe 中获得 IRQ 资源，即中断号)

irq_domain 是核心：
a. 每一个中断控制器都有一个 irq_domain
b. 对设备中断信息的解析，
b.1 需要调用 irq_domain->ops->xlate (即从设备树中获得 hwirq, type)
b.2 获取未使用的 virq，保存：irq_domain->linear_revmap[hwirq] = virq;
b.3 在 hwirq 和 virq 之间建立联系：
要调用 irq_domain->ops->map，比如根据 hwirq 的属性设置 virq 的中断处理函数(是一个分发函数还是可以直接处理中断)
irq_desc[virq].handle_irq = 常规函数;
如果这个 hwirq 有上一级中断，假设它的中断号为 virq'，还要设置：
irq_desc[virq'].handle_irq = 中断分发函数;

s3c2440 设备树中断相关代码调用关系：

(1) 上述处理过程如何触发?
a. 内核启动时初始化中断的入口：

```
start_kernel // init/main.c
    init_IRQ();
        if (IS_ENABLED(CONFIG_OF) && !machine_desc->init_irq)
            irqchip_init();     // 一般使用它
        else
            machine_desc->init_irq();
```

b. 设备树中的中断控制器的处理入口:
irqchip_init // drivers/irqchip/irqchip.c
of_irq_init(__irqchip_of_table); // 对设备树文件中每一个中断控制器节点，调用对应的处理函数
为每一个符合的"interrupt-controller"节点,
分配一个 of_intc_desc 结构体，desc->irq_init_cb = match->data; // = IRQCHIP_DECLARE 中传入的函数
并调用处理函数

(先调用 root irq controller 对应的函数，再调用子控制器的函数，再调用更下一级控制器的函数...)

(2) root irq controller 的驱动调用过程:

a. 为 root irq controller 定义处理函数:
IRQCHIP_DECLARE(s3c2410_irq, "samsung,s3c2410-irq", s3c2410_init_intc_of);
//drivers/irqchip/irq-s3c24xx.c

其中:
```
#define IRQCHIP_DECLARE(name, compat, fn) OF_DECLARE_2(irqchip, name, compat, fn)
#define OF_DECLARE_2(table, name, compat, fn) \
        _OF_DECLARE(table, name, compat, fn, of_init_fn_2)
#define _OF_DECLARE(table, name, compat, fn, fn_type)                   \
    static const struct of_device_id __of_table_##name          \
        __used __section(__##table##_of_table)                  \
          = { .compatible = compat,                         \
              .data = (fn == (fn_type)NULL) ? fn : fn   }
```

展开为:
```
    static const struct of_device_id __of_table_s3c2410_irq         \
        __used __section("__irqchip_of_table")                  \
          = { .compatible = "samsung,s3c2410-irq",                      \
              .data = s3c2410_init_intc_of    }
```

它定义了一个 of_device_id 结构体，段属性为"__irqchip_of_table"，在编译内核时这些段被放在__irqchip_of_table 地址处。
即__irqchip_of_table 起始地址处,
放置了一个或多个 of_device_id，它含有 compatible 成员;
设备树中的设备节点含有 compatible 属性,
如果双方的 compatible 相同，并且设备节点含有"interrupt-controller"属性,
则调用 of_device_id 中的函数来处理该设备节点。

所以: IRQCHIP_DECLARE 是用来声明设备树中的中断控制器的处理函数。

b. root irq controller 处理函数的执行过程:

```
s3c2410_init_intc_of    // drivers/irqchip/irq-s3c24xx.c
    //  初始化中断控制器: intc, subintc
    s3c_init_intc_of(np, interrupt_parent, s3c2410_ctrl, ARRAY_SIZE(s3c2410_ctrl));

        //  为中断控制器创建 irq_domain
        domain = irq_domain_add_linear(np, num_ctrl * 32,
                                       &s3c24xx_irq_ops_of, NULL);

        intc->domain = domain;

        //  设置 handle_arch_irq, 即中断处理的 C 语言总入口函数
        set_handle_irq(s3c24xx_handle_irq);
```

(3) pinctrl 系统中 gpf/gpg irq controller 的驱动调用过程:

a. pinctrl 系统的驱动程序:

a.1 源代码: drivers/pinctrl/samsung/pinctrl-samsung.c

```
static struct platform_driver samsung_pinctrl_driver = {
    .probe          = samsung_pinctrl_probe,
    .driver = {
        .name     = "samsung-pinctrl",
        .of_match_table   =   samsung_pinctrl_dt_match,   //   含 有    {   .compatible   =
"samsung,s3c2440-pinctrl", .data = &s3c2440_of_data },
        .suppress_bind_attrs = true,
        .pm = &samsung_pinctrl_pm_ops,
    },
};
```

a.2 设备树中:

```
pinctrl@56000000 {
    reg = <0x56000000 0x1000>;
    compatible = "samsung,s3c2440-pinctrl";   //  据此找到驱动
```

a.3 驱动中的操作:

```
samsung_pinctrl_probe    // drivers/pinctrl/samsung/pinctrl-samsung.c
    最终会调用到  s3c24xx_eint_init // drivers/pinctrl/samsung/pinctrl-s3c24xx.c

        // eint0,1,2,3 的处理函数在处理 root irq controller 时已经设置;
        //  设置 eint4_7, eint8_23 的处理函数(它们是分发函数)
        for (i = 0; i < NUM_EINT_IRQ; ++i) {
            unsigned int irq;
```

```
            if (handlers[i]) /* add by weidongshan@qq.com, 不再设置 eint0,1,2,3 的处理函数 */
            {
                irq = irq_of_parse_and_map(eint_np, i);
                if (!irq) {
                    dev_err(dev, "failed to get wakeup EINT IRQ %d\n", i);
                    return -ENXIO;
                }

                eint_data->parents[i] = irq;
                irq_set_chained_handler_and_data(irq, handlers[i], eint_data);
            }
        }

        // 为 GPF、GPG 设置 irq_domain
        for (i = 0; i < d->nr_banks; ++i, ++bank) {

            ops = (bank->eint_offset == 0) ? &s3c24xx_gpf_irq_ops
                                           : &s3c24xx_gpg_irq_ops;

            bank->irq_domain = irq_domain_add_linear(bank->of_node, bank->nr_pins, ops, ddata);
        }
```

(4) 使用中断的驱动调用过程:

a. 在设备节点中描述(表明使用"哪一个中断控制器里的哪一个中断, 及中断触发方式")

比如:

```
    buttons {
        compatible = "jz2440_button";
        eint-pins    = <&gpf 0 0>, <&gpf 2 0>, <&gpg 3 0>, <&gpg 11 0>;
        interrupts-extended = <&intc 0 0 0 3>,
                              <&intc 0 0 2 3>,
                              <&gpg 3 3>,
                              <&gpg 11 3>;
    };
```

b. 设备节点会被转换为 platform_device,

"中断的硬件信息" 会转换为"中断号",

保存在 platform_device 的"中断资源"里

第 3 课第 05 节_device_node 转换为 platform_device, 讲解了设备树中设备节点转换为 platform_device 的过程;

我们只关心里面对中断信息的处理:

of_device_alloc (drivers/of/platform.c)
dev = platform_device_alloc("", PLATFORM_DEVID_NONE); // 分配 platform_device

num_irq = of_irq_count(np); // 计算中断数

of_irq_to_resource_table(np, res, num_irq) // drivers/of/irq.c, 根据设备节点中的中断信息, 构造中断资源
of_irq_to_resource
int irq = of_irq_get(dev, index); // 获得 virq, 中断号
rc = of_irq_parse_one(dev, index, &oirq); // drivers/of/irq.c, 解析设备树中的中断信息, 保存在 of_phandle_args 结构体中

domain = irq_find_host(oirq.np); // 查找 irq_domain, 每一个中断控制器都对应一个 irq_domain

irq_create_of_mapping(&oirq); // kernel/irq/irqdomain.c, 创建 virq 和中断信息的映射
irq_create_fwspec_mapping(&fwspec);
irq_create_fwspec_mapping(&fwspec);
irq_domain_translate(domain, fwspec, &hwirq, &type) // 调用 irq_domain->ops->xlate, 把设备节点里的中断信息解析为 hwirq, type

virq = irq_find_mapping(domain, hwirq); // 看看这个 hwirq 是否已经映射, 如果 virq 非 0 就直接返回

virq = irq_create_mapping(domain, hwirq); // 否则创建映射
virq = irq_domain_alloc_descs(-1, 1, hwirq, of_node_to_nid(of_node), NULL); // 返回未占用的 virq

irq_domain_associate(domain, virq, hwirq) // 调用 irq_domain->ops->map(domain, virq, hwirq), 做必要的硬件设置

c. 驱动程序从 platform_device 的"中断资源"取出中断号, 就可以 request_irq 了

第六课. 实践操作
第 01 节_使用设备树给 DM9000 网卡_触摸屏指定中断

修改方法:
根据设备节点的 compatible 属性,
在驱动程序中构造/注册 platform_driver,
在 platform_driver 的 probe 函数中获得中断资源

实验方法:
以下是修改好的代码:
第 6 课第 1 节_网卡_触摸屏驱动\001th_dm9000\dm9dev9000c.c
第 6 课第 1 节_网卡_触摸屏驱动\002th_touchscreen\s3c_ts.c

分别上传到内核如下目录:
drivers/net/ethernet/davicom
drivers/input/touchscreen

a. 编译内核
b. 使用新的 uImage 启动
c. 测试网卡:
ifconfig eth0 192.168.1.101
ping 192.168.1.1

d. 测试触摸屏:
hexdump /dev/evetn0 // 然后点击触摸屏

第 02 节_在设备树中时钟的简单使用

文档:
内核 Documentation/devicetree/bindings/clock/clock-bindings.txt
内核 Documentation/devicetree/bindings/clock/samsung,s3c2410-clock.txt

a. 设备树中定义了各种时钟, 在文档中称之为"Clock providers", 比如:

```
clocks: clock-controller@4c000000 {
        compatible = "samsung,s3c2440-clock";
        reg = <0x4c000000 0x20>;
        #clock-cells = <1>;          // 想使用这个 clocks 时要提供 1 个 u32 来指定它, 比如选择这个 clocks 中发出的 LCD 时钟、PWM 时钟
};
```

b. 设备需要时钟时, 它是"Clock consumers", 它描述了使用哪一个"Clock providers"中的哪一个时钟(id), 比如:

```
fb0: fb@4d000000{
        compatible = "jz2440,lcd";
        reg = <0x4D000000 0x60>;
```

```
        interrupts = <0 0 16 3>;
        clocks = <&clocks HCLK_LCD>;  // 使用 clocks 即 clock-controller@4c000000 中的
HCLK_LCD
    };
```

c. 驱动中获得/使能时钟:

```
    // 确定时钟个数
    int nr_pclks = of_count_phandle_with_args(dev->of_node, "clocks",
                                    "#clock-cells");
    // 获得时钟
    for (i = 0; i < nr_pclks; i++) {
        struct clk *clk = of_clk_get(dev->of_node, i);
    }

    // 使能时钟
    clk_prepare_enable(clk);

    // 禁止时钟
    clk_disable_unprepare(clk);
```

第 03 节_在设备树中 pinctrl 的简单使用

文档:
内核 Documentation/devicetree/bindings/pinctrl/samsung-pinctrl.txt

几个概念:

Bank: 以引脚名为依据, 这些引脚分为若干组, 每组称为一个 Bank
比如 s3c2440 里有 GPA、GPB、GPC 等 Bank,
每个 Bank 中有若干个引脚, 比如 GPA0,GPA1, ..., GPC0, GPC1,...等引脚

Group: 以功能为依据, 具有相同功能的引脚称为一个 Group
比如 s3c2440 中串口 0 的 TxD、RxD 引脚使用 GPH2,GPH3, 那这 2 个引脚可以列为一组
比如 s3c2440 中串口 0 的流量控制引脚使用 GPH0,GPH1, 那这 2 个引脚也可以列为一组

State: 设备的某种状态, 比如内核自己定义的"default","init","idel","sleep"状态;
也可以是其他自己定义的状态, 比如串口的"flow_ctrl"状态(使用流量控制)

设备处于某种状态时, 它可以使用若干个 Group 引脚

a. 设备树中 pinctrl 节点:

a.1 它定义了各种 pin bank, 比如 s3c2440 有 GPA,GPB,GPC,...,GPB 各种 BANK, 每个 BANK 中有若干引脚:

```
pinctrl_0: pinctrl@56000000 {
        reg = <0x56000000 0x1000>;

        gpa: gpa {
                gpio-controller;
                #gpio-cells = <2>;   /* 以后想使用 gpa bank 中的引脚时, 需要 2 个 u32 来指定引脚 */
        };

        gpb: gpb {
                gpio-controller;
                #gpio-cells = <2>;
        };

        gpc: gpc {
                gpio-controller;
                #gpio-cells = <2>;
        };

        gpd: gpd {
                gpio-controller;
                #gpio-cells = <2>;
        };
};
```

a.2 它还定义了各种 group(组合), 某种功能所涉及的引脚称为 group,
比如串口 0 要用到 2 个引脚: gph0, gph1:

```
uart0_data: uart0-data {
        samsung,pins = "gph-0", "gph-0";
        samsung,pin-function = <2>;     /* 在 GPHCON 寄存器中 gph0,gph1 可以设置以下值:
                                                0 --- 输入功能
                                                1 --- 输出功能
                                                2 --- 串口功能
                                           我们要使用串口功能,
                                           samsung,pin-function 设置为 2
                                         */
};

uart0_sleep: uart0_sleep {
        samsung,pins = "gph-0", "gph-1";
        samsung,pin-function = <0>;     /* 在 GPHCON 寄存器中 gph0,gph1 可以设置以下值:
```

```
                                                    0 --- 输入功能
                                                    1 --- 输出功能
                                                    2 --- 串口功能
                                                  我们要使用输入功能,
                                                    samsung,pin-function 设置为 0
                                                */
    };
```

b. 设备节点中要使用某一个 pin group:

```
    serial@50000000 {
        ......
        pinctrl-names = "default", "sleep";   /* 既是名字, 也称为 state(状态) */
        pinctrl-0 = <&uart0_data>;
        pinctrl-1 = <&uart0_sleep>;
    };
```

pinctrl-names 中定义了 2 种 state: default 和 sleep,
default 对应的引脚是: pinctrl-0, 它指定了使用哪些 pin group: uart0_data
sleep 对应的引脚是: pinctrl-1, 它指定了使用哪些 pin group: uart0_sleep

c. platform_device, platform_driver 匹配时:

"第 3 课第 06 节_platform_device 跟 platform_driver 的匹配" 中讲解了 platform_device 和 platform_driver 的匹配过程,
最终都会调用到 really_probe (drivers/base/dd.c)

```
really_probe:
    /* If using pinctrl, bind pins now before probing */
    ret = pinctrl_bind_pins(dev);
                dev->pins->default_state = pinctrl_lookup_state(dev->pins->p,
                                        PINCTRL_STATE_DEFAULT);    /* 获得"default"状态的
pinctrl */
                dev->pins->init_state = pinctrl_lookup_state(dev->pins->p,
                                        PINCTRL_STATE_INIT);      /* 获得"init"状态的 pinctrl */

                ret = pinctrl_select_state(dev->pins->p, dev->pins->init_state);      /* 优先
设置"init"状态的引脚 */
                ret = pinctrl_select_state(dev->pins->p, dev->pins->default_state); /* 如果没
有 init 状态, 则设置"default"状态的引脚 */

    ......
    ret = drv->probe(dev);
```

所以: 如果设备节点中指定了 pinctrl, 在对应的 probe 函数被调用之前, 先"bind pins", 即先绑定、设置引脚

d. 驱动中想选择、设置某个状态的引脚:
devm_pinctrl_get_select_default(struct device *dev); // 使用"default"状态的引脚
pinctrl_get_select(struct device *dev, const char *name); // 根据 name 选择某种状态的引脚

pinctrl_put(struct pinctrl *p); // 不再使用, 退出时调用

第 04 节_使用设备树给 LCD 指定各种参数

参考文章:
讓 TQ2440 也用上設備樹（1）
http://www.cnblogs.com/pengdonglin137/p/6241895.html

参考代码:
https://github.com/pengdonglin137/linux-4.9/blob/tq2440_dt/drivers/video/fbdev/s3c2410fb.c

实验方法:
所用文件在: doc_and_sources_for_device_tree\source_and_images\第 5,6 课的源码及映像文件(使用了完全版的设备树)\第 6 课第 4 节_LCD 驱动\02th_我修改的

a. 替换 dts 文件:
把"jz2440_irq.dts" 放入内核 arch/arm/boot/dts 目录,

b. 替换驱动文件:
把"s3c2410fb.c" 放入内核 drivers/video/fbdev/ 目录,
修改 内核 drivers/video/fbdev/Makefile :
obj-$(CONFIG_FB_S3C2410) += lcd_4.3.o
改为:
obj-$(CONFIG_FB_S3C2410) += s3c2410fb.o

c. 编译驱动、编译 dtbs:
export PATH=PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/work/system/gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabi/bin
cp config_ok .config
make uImage // 生成 arch/arm/boot/uImage
make dtbs // 生成 arch/arm/boot/dts/jz2440_irq.dtb

d. 使用上述 uImage, dtb 启动内核即可看到 LCD 有企鹅出现

(1). 设备树中的描述:

```
fb0: fb@4d000000{
        compatible = "jz2440,lcd";
        reg = <0x4D000000 0x60>;
        interrupts = <0 0 16 3>;
        clocks = <&clocks HCLK_LCD>;      /* a. 时钟 */
        clock-names = "lcd";
        pinctrl-names = "default";          /* b. pinctrl */
        pinctrl-0 = <&lcd_pinctrl &lcd_backlight &gpb0_backlight>;
        status = "okay";

        /* c. 根据 LCD 引脚特性设置 lcdcon5, 指定 lcd 时序参数 */
        lcdcon5 = <0xb09>;
        type = <0x60>;
        width = /bits/ 16 <480>;
        height = /bits/ 16 <272>;
        pixclock = <100000>;              /* 单位: ps, 10^-12 S,    */
        xres = /bits/ 16 <480>;
        yres = /bits/ 16 <272>;
        bpp = /bits/ 16 <16>;
        left_margin = /bits/ 16 <2>;
        right_margin =/bits/ 16    <2>;
        hsync_len = /bits/ 16 <41>;
        upper_margin = /bits/ 16 <2>;
        lower_margin = /bits/ 16 <2>;
        vsync_len = /bits/ 16 <10>;
    };

&pinctrl_0 {
    gpb0_backlight: gpb0_backlight {
        samsung,pins = "gpb-0";
        samsung,pin-function = <1>;
        samsung,pin-val = <1>;
    };
};
```

(2) 代码中的处理:

a. 时钟:

```
info->clk = of_clk_get(dev->of_node, 0);
clk_prepare_enable(info->clk);
```

b. pinctrl:
代码中无需处理，在 platform_device/platform_driver 匹配之后就会设置"default"状态对应的 pinctrl

c. 根据 LCD 引脚特性设置 lcdcon5，指定 lcd 时序参数:

直接读设备树节点中的各种属性值，用来设置驱动参数

```
of_property_read_u32(np, "lcdcon5", (u32 *)(&display->lcdcon5));
of_property_read_u32(np, "type", &display->type);
of_property_read_u16(np, "width", &display->width);
of_property_read_u16(np, "height", &display->height);
of_property_read_u32(np, "pixclock", &display->pixclock);
of_property_read_u16(np, "xres", &display->xres);
of_property_read_u16(np, "yres", &display->yres);
of_property_read_u16(np, "bpp", &display->bpp);
of_property_read_u16(np, "left_margin", &display->left_margin);
of_property_read_u16(np, "right_margin", &display->right_margin);
of_property_read_u16(np, "hsync_len", &display->hsync_len);
of_property_read_u16(np, "upper_margin", &display->upper_margin);
of_property_read_u16(np, "lower_margin", &display->lower_margin);
of_property_read_u16(np, "vsync_len", &display->vsync_len);
```

临时笔记:

(1) 下面是确定内核的虚拟地址、物理地址的关键信息，感兴趣的同学可以自己看：
vmlinux 虚拟地址的确定:
内核源码:
.config :

```
CONFIG_PAGE_OFFSET=0xC0000000
```

arch/arm/include/asm/memory.h

```
#define PAGE_OFFSET         UL(CONFIG_PAGE_OFFSET)
```

arch/arm/Makefile

textofs-y := 0x00008000
TEXT_OFFSET := $(textofs-y)

arch/arm/kernel/vmlinux.lds.S:
. = PAGE_OFFSET + TEXT_OFFSET; // // 即 0xC0000000+0x00008000 = 0xC0008000, vmlinux 的虚拟地址为 0xC0008000

arch/arm/kernel/head.S
#define KERNEL_RAM_VADDR (PAGE_OFFSET + TEXT_OFFSET) // 即 0xC0000000+0x00008000 = 0xC0008000

vmlinux 物理地址的确定:
内核源码:
arch/arm/mach-s3c24xx/Makefile.boot :
zreladdr-y += 0x30008000 // zImage 自解压后得到 vmlinux, vmlinux 的存放位置
params_phys-y := 0x30000100 // tag 参数的存放位置，使用 dtb 时不再需要 tag

arch/arm/boot/Makefile:
ZRELADDR := $(zreladdr-y)

arch/arm/boot/Makefile:
UIMAGE_LOADADDR=$(ZRELADDR)

scripts/Makefile.lib:
UIMAGE_ENTRYADDR ?= $(UIMAGE_LOADADDR)

// 制作 uImage 的命令, uImage = 64 字节的头部 + zImage, 头部信息中含有内核的入口地址(就是 vmlinux 的物理地址)
cmd_uimage = $(CONFIG_SHELL) $(MKIMAGE) -A $(UIMAGE_ARCH) -O linux \
-C $(UIMAGE_COMPRESSION) $(UIMAGE_OPTS-y) \
-T $(UIMAGE_TYPE) \
-a $(UIMAGE_LOADADDR) -e $(UIMAGE_ENTRYADDR) \
-n $(UIMAGE_NAME) -d $(UIMAGE_IN) $(UIMAGE_OUT)

Linux kernel 的中断子系统之（二）：IRQ Domain 介绍_搜狐科技_搜狐网.html
http://www.sohu.com/a/201793206_467784

基于设备树的 TQ2440 的中断（1）
https://www.cnblogs.com/pengdonglin137/p/6847685.html

基于设备树的 TQ2440 的中断（2）
https://www.cnblogs.com/pengdonglin137/p/6848851.html

基於 tiny4412 的 Linux 內核移植 --- 实例学习中断背后的知识(1)
http://www.cnblogs.com/pengdonglin137/p/6349209.html

Linux kernel 的中断子系统之（一）：综述
http://www.wowotech.net/irq_subsystem/interrupt_subsystem_architecture.html

Linux kernel 的中断子系统之（二）：IRQ Domain 介绍

linux kernel 的中断子系统之（三）：IRQ number 和中断描述符

linux kernel 的中断子系统之（四）：High level irq event handler

Linux kernel 中断子系统之（五）：驱动申请中断 API

Linux kernel 的中断子系统之（六）：ARM 中断处理过程

linux kernel 的中断子系统之（七）：GIC 代码分析