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:

------------------------------

base -> | struct boot\_param\_header |

------------------------------

| (alignment gap) (\*) |

------------------------------

| memory reserve map |

------------------------------

| (alignment gap) |

------------------------------

| |

| device-tree structure |

| |

------------------------------

| (alignment gap) |

------------------------------

| |

| device-tree strings |

| |

-----> ------------------------------

|

|

--- (base + totalsize)

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:

------------------------------

0x33f80000 ->| u-boot |

------------------------------

| u-boot所使用的内存(栈等)|

------------------------------

| |

| |

| 空闲区域 |

| |

| |

| |

| |

------------------------------

0x30008000 ->| zImage |

------------------------------ uImage = 64字节的头部+zImage

0x30007FC0 ->| uImage头部 |

------------------------------

0x30004000 ->| 内核创建的页表 | head.S

------------------------------

| |

| |

-----> ------------------------------

|

|

--- (内存基址 0x30000000)

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)

00-Linux设备树系列-简介 - 飞翔de刺猬 - CSDN博客.html

https://blog.csdn.net/lhl\_blog/article/details/82387486

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