RK 平台移植 3.10 内核

常用模块相关改动案例

1.1 LCD/BL

具体请参考文档《RockChip_DSS Development Guide v1.2.pdf》

1.2SENSORS

sensors 包括了 RK 平台上的各类驱动，驱动代码路径和原来路径一致，在 drivers/inpput/sensors/目录下。驱动已经移植完成，正常不需要进行修改。sensor 的配置只需要在 dts 文件中填入项目对应的信息。

1.2.1 驱动需要获取的信息：

Sensor-dev.h 中 我们定义了 sensor 的驱动所需要的信息结构，在.dts 文件中填入相关信息，驱动中就可以自动获取到信息进行展开。

```
struct sensor_platform_data {
    int type;               //sensor 类型见 sensor_type 定义，必填，不能错
    int irq;                //中断号
    int irq_pin;             //中断触发 GPIO
    int power_pin;          //电源控制 GPIO，部分 sensor 用到
    int reset_pin;          //reset GPIO，部分 sensor 用到
    int irq_enable;            //使用中断方式填 1，使用轮训填 0 ，必填
    int poll_delay_ms;          //读取数据的频率，单位 ms，必填
    int x_min;      //最小有效值，x 低于该值忽略，目前陀螺仪使用
    int y_min;      //最小有效值，y 低于该值忽略，目前陀螺仪使用
    int z_min;       //最小有效值，z 低于该值忽略，目前陀螺仪使用
    unsigned char address;          // I2C 设备地址
    int layout;                     //sensor 方向，1-8
    signed char orientation[9];          //sensor 方向矩阵，dts 中不用定义，由 layout 自动转换
    unsigned long   irq_flags;          //中断触发方式，必填
    short m_layout[4][3][3];       //指南针方向矩阵，可选
    char project_name[64];              //指南针名称，可选
};
```

蓝色的为在 dts 文件中必须定义的信息，其他为可选项，没用到可以不填。

1.2.2 dts 中如何进行定义

目前 dts 中的信息填写和之前的 board 有很大区别，这里用用 mma8452 的 gsensor 作为案例：

```
 &i2c0{                                          //挂载 i2c0 总线上
        sensor@1d {                              //sensor，i2c 地址
                compatible = "gs_mma8452";              //设备 ID 号
                reg = <0x1d>;                            //设备地址
                type = <SENSOR_TYPE_ACCEL>;          //sensor 类型，accel 为 gsensor
                irq-gpio = <&gpio0 GPIO_B7 IRQ_TYPE_EDGE_FALLING>;      //中断 GPIO
                irq_enable = <1>;                        //使用中断
                poll_delay_ms = <30>;                    //轮询时间
                layout = <4>;                          //方向矩阵，1-8
    };
 };
```

另若 sensor 的 pwr 脚和 rst 脚需要定义的话，dts 描述如下：

```
reset-gpio = <&gpio0 GPIO_B6 GPIO_ACTIVE_LOW>;
power-gpio = <&gpio0 GPIO_C5 GPIO_ACTIVE_LOW>;
```

senesor type 定义,在 include/dt-bindings/sensor-dev.h 中，目前在 dts 中想使用到的宏定义都放在 dt-bindings/目录下，可被 dts/dtsi 引用到：

```
5 #define SENSOR_TYPE_NULL    0
6 #define SENSOR_TYPE_ANGLE 1
7 #define SENSOR_TYPE_ACCEL 2                    //gsensor
8 #define SENSOR_TYPE_COMPASS 3                  //指南针
9 #define SENSOR_TYPE_GYROSCOPE 4                //陀螺仪
10 #define SENSOR_TYPE_LIGHT 5                   //光感
11 #define SENSOR_TYPE_PROXIMITY 6
12 #define SENSOR_TYPE_TEMPERATURE 7             //温度计
13 #define SENSOR_TYPE_PRESSURE 8                //压力传感器
14 #define SENSOR_TYPE_HALL 9                    //距离传感器
15 #define SENSOR_NUM_TYPES 10
```

irq-gpio = <&gpio0 GPIO_B7 IRQ_TYPE_EDGE_FALLING>; 定义了中断触发脚的 GPIO 为 GPIO0_B7，中断触发方式为下降沿触发，触发方式可定义为以下五种：

IRQ_TYPE_EDGE_RISING	上升沿触发
IRQ_TYPE_EDGE_FALLING	下降沿触发
IRQ_TYPE_EDGE_BOTH	沿触发
IRQ_TYPE_LEVEL_HIGH	高电平触发
IRQ_TYPE_LEVEL_LOW	低电平触发

1.2.3 驱动中如何获取 dts 信息

首先要跟 dts 文件建立通讯，具体代码如下：

```
static struct of_device_id sensor_dt_ids[] = {
{ .compatible = "gs_mma8452" },          //这边定义必须和 dts 中定义的一样。
};

static struct i2c_driver sensor_driver = {
        .probe = sensor_probe,
        .remove = sensor_remove,
        .shutdown = sensor_shut_down,
        .id_table = sensor_id,
        .driver = {
                .owner = THIS_MODULE,
```

```
        .name = "sensors",
        .of_match_table = of_match_ptr(sensor_dt_ids),          //3.10 中增加的，必须要有
    },
```

dts 中定义的种种属性，我们都是在 probe 中获取的，如下：

```
1680        of_property_read_u32(np,"type",&(pdata->type));
1681
1682        pdata->irq_pin = of_get_named_gpio_flags(np, "irq-gpio", 0,(enum of_gpio_flags
*)&irq_flags);
1683        pdata->reset_pin = of_get_named_gpio_flags(np, "reset-gpio",0,&rst_flags);
1684        pdata->power_pin = of_get_named_gpio_flags(np, "power-gpio",0,&pwr_flags);
1685
1686        of_property_read_u32(np,"irq_enable",&(pdata->irq_enable));
1687        of_property_read_u32(np,"poll_delay_ms",&(pdata->poll_delay_ms));
1688
1689        of_property_read_u32(np,"x_min",&(pdata->x_min));
1690        of_property_read_u32(np,"y_min",&(pdata->y_min));
1691        of_property_read_u32(np,"z_min",&(pdata->z_min));
1692        of_property_read_u32(np,"factory",&(pdata->factory));
1693        of_property_read_u32(np,"layout",&(pdata->layout));
1694
1695        of_property_read_u8(np,"address",&(pdata->address));
1696        pdata->project_name = of_get_property(np, "project_name", NULL);
```

gpio 获取的同时，还会把 dts 中定义的属性传到对应的 flag 参数中，包括中断触发方式，GPIO 的输出/输入有效电平。

1.3 CAMERA

目前 camera 版本，已经支持的 sensor，需要配置修改的都在 device/rockchip/common/camera/cam_board.xml 中，内核 dts 相关不需要改动；

具体请参考文档《RK3288_Camera_User_Manual_v1.1.pdf》；

1.4 pinctrl 相关改动

详见《pinctl 驱动介绍资料.rar》

Dts 介绍

2.1 dts/dtsi 为何物

Device Tree 是一种描述硬件的数据结构，它起源于 OpenFirmware (OF)。在 Linux 2.6 中,ARM 架构的板极硬件细节过多地被硬编码在 arch/arm/plat-xxx 和 arch/arm/mach-xxx，采用 Device Tree 后，许多硬件的细节可以直接透过它传递给 Linux，而不再需要在 kernel 中进行大量的冗余编码。
Device Tree 由一系列被命名的结点（node）和属性（property）组成，而结点本身可包含子结点。所谓属性，其实就是成对出现的 name 和 value。在 Device Tree 中，可描述的信息包括（原先这些信息大多被 hard code 到 kernel 中）：

- CPU 的数量和类别
- 内存基地址和大小
- 总线和桥
- 外设连接
- 中断控制器和中断使用情况
- GPIO 控制器和 GPIO 使用情况
- Clock 控制器和 Clock 使用情况

它基本上就是画一棵电路板上 CPU、总线、设备组成的树，Bootloader 会将这棵树传递给内核，然后内核可以识别这棵树，并根据它展开出 Linux 内核中的 platform_device、i2c_client、spi_device 等设备，而这些设备用到的内存、IRQ 等资源，也被传递给了内核，内核会将这些资源绑定给展开的相应的设备。

.dts 文件是一种 ASCII 文本格式的 Device Tree 描述，此文本格式非常人性化，适合人类的阅读习惯。基本上，在 ARM Linux 在，一个.dts 文件对应一个 ARM 的 machine，一般放置在内核的 arch/arm/boot/dts/目录。由于一个 SoC 可能对应多个 machine（一个 SoC 可以对应多个产品和电路板），势必这些.dts 文件需包含许多共同的部分，Linux 内核为了简化，把 SoC 公用的部分或者多个 machine 共同的部分一般提炼为.dtsi，类似于 C 语言的头文件。其他的 machine 对应的.dts 就 include 这个.dtsi。

2.2 rk 平台集成的 dtsi

clocks.dtsi
mmc.dtsi

pinctl.dtsi
rk3xxx.dtsi

2.3 rk 平台 dts 编写规范和注意事项

详见 kernel/Documentation/devicetree/bindings 目录下各模块的说明文档。

2.4 如何编译生成新的 kernel.img

各个项目都需要编写特定的 dts 文件，命名为 rk.3288-xxx.dts
编译之前请使用命令： make rockchip_defconfig 生成 .config 文件
编译的时候使用命令 ： make rk3288-xxx.img
可以生成 kernel.img 和相对应的 resource.img

Dts 引发的驱动变更

3.1 驱动如何与 dts 通信

有了 Device Tree 后，大量的板级信息都不再需要。在 SoC 对应的 machine 的 .init_machine 成员函数中，调用 of_platform_bus_probe(NULL, xxx_of_bus_ids, NULL);即可自动展开所有的 platform_device。实际来源于.dts 中设备结点的 reg、interrupts 属性也同时被引用。

3.2 注册 i2c_board_info,指定 IRQ 等板级信息

Device Tree 中的 I2C client 会透过 I2C host 驱动的 probe()函数中调用 of_i2c_register_devices(&i2c_dev->adapter);被自动展开。

```
&i2c0 {
	status = "okay";
	rt5631@1a {
		compatible = "rt5631";
		reg = <0x1a>;
	};
	sensor@1d {
		 compatible = "gs_mma8452";
		 reg = <0x1d>;
		 type = <SENSOR_TYPE_ACCEL>;
```

```
            rq-gpio = <&gpio0 GPIO_B7 IRQ_TYPE_EDGE_FALLING>;
            irq_enable = <1>;
            poll_delay_ms = <30>;
            layout = <4>;
    };
```

3.3 注册 platfrom_device,绑定 resource(内存，irq 等)

使用 Device Tree 后，驱动需要与.dts 中描述的设备结点进行匹配，从而引发驱动的 probe()函数执行。对于 platform_driver 而言，需要添加一个 OF 匹配表，如前文的.dts 文件的"gs_mma8452"兼容 I2C 控制器结点的 OF 匹配表可以是：

```
static struct of_device_id sensor_dt_ids[] = {
        { .compatible = "gs_mma8452" },
        {},
}
static struct i2c_driver sensor_driver = {
         .probe = sensor_probe,
         .remove = sensor_remove,
         .shutdown = sensor_shut_down,
         .id_table = sensor_id,
         .driver = {
                .owner = THIS_MODULE,
                .name = "sensors",
                .of_match_table = of_match_ptr(sensor_dt_ids),
         },
};
```

常用 OF_API

struct device_node *of_find_compatible_node(struct device_node *from,
const char *type, const char *compatible);
根据 compatible 属性，获得设备结点。遍历 Device Tree 中所有的设备结点，看看哪个结点的类型、compatible 属性与本函数的输入参数匹配，大多数情况下，from、type 为 NULL。

int of_property_read_u8_array(const struct device_node *np,
const char *propname, u8 *out_values, size_t sz);
int of_property_read_u16_array(const struct device_node *np,
const char *propname, u16 *out_values, size_t sz);
int of_property_read_u32_array(const struct device_node *np,
const char *propname, u32 *out_values, size_t sz);

int of_property_read_u64(const struct device_node *np, const char *propname, u64 *out_value);

读取设备结点 np 的属性名为 propname，类型为8、16、32、64位整型数组的属性。对于32位处理器来讲，最常用的是 of_property_read_u32_array()。

有些情况下，整形属性的长度可能为1，于是内核为了方便调用者，又在上述 API 的基础上封装出了更加简单的读单一整形属性的 API，它们为

```
int of_property_read_u8();
Int of_property_read_u16();
Int of_property_read_u30();
```

int of_property_read_string(struct device_node *np, const char *propname, const char **out_string);

int of_property_read_string_index(struct device_node *np, const char *propname, int index, const char **output);

前者读取字符串属性，后者读取字符串数组属性中的第 index 个字符串。

of_get_named_gpio_flags