# TODO

<https://www.cnblogs.com/biglucky/p/4059582.html>

TYPEC

<https://blog.csdn.net/zoosenpin/article/details/49963031>

<https://blog.csdn.net/Fybon/article/details/78115198>

# 概述

分析用的**内核版本为5.1.3**

mfd是Multifunction device的简称，即多功能设备，是许多有共性的设备的集合，mfd由核心层(core)以及其下的“子设备”组成。从下文将会看到，mfd只是将设备注册到platform总线——因此，其子设备属于platform设备。它并没有对涉及到的设备或驱动做实质性改变。但是，因为某些设备的共性，所以可以在mfd中提供共同的函数给其下子设备进行调用

————————————————

版权声明：本文为CSDN博主「李迟」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/subfate/article/details/53464641

## 为何会出现MFD子系统

--由于出现了一类具有多种功能的外围设备或cpu内部集成的硬件模块

## 3. 有哪些多功能设备呢?

　　3.1 PMIC,电源管理芯片

　　　　da9063: 调节器,led控制器,看门狗,实时时钟控制器,温度传感器,震动马达驱动,长按关机功能(ON key)

　　　　max77843: 调节器,充电器,燃油量表,触觉反馈,led控制器,micro USB接口控制器

　　　　wm831x: 调节器,时钟,实时时钟控制器,看门狗,触摸控制器,温度传感器,背光控制器,状态led控制器,GPIO,长按关机功能(ON key),ADC

　　　　其它: 甚至具有codec功能

　　3.2 atmel-hlcdc: 显示控制器和背光pwm

　　3.3 Diolan DLN2: USB转I2C,SPI和GPIO控制器

3.4 Realtek PCI-E读卡器: SD/MMC和记忆棒读取器

## MFD子系统解决的主要问题

在不同的内核子系统中注册这些驱动。特别是外部外围设备仅仅由一个结构体struct device(或是指定的i2c\_client或spi\_device)呈现

## 5. MFD子系统的优点有哪些?

　　5.1 允许在多个子系统中注册相同的设备

　　5.2 MFD驱动必须能否复用总线(主要是关于锁的处理)和处理中断请求

　　5.3 处理时钟

　　5.4 需要配置IP

　　5.5 允许驱动重用,多个多功能设备能重用其它子系统中的驱动

# 源码分析

## MFD提供的API

include/linux/mfd/core.h中,在drivers/mfd/mtd-core.c中被实现

**static inline int** mfd\_add\_hotplug\_devices(**struct** device \*parent,  
 **const struct** mfd\_cell \*cells, **int** n\_devs)  
{  
 **return** mfd\_add\_devices(parent, PLATFORM\_DEVID\_AUTO, cells, n\_devs,  
 NULL, 0, NULL);  
}  
  
**extern void** mfd\_remove\_devices(**struct** device \*parent);

### 结构体mfd\_cell

*/\*  
 \* This struct describes the MFD part ("cell").  
 \* After registration the copy of this structure will become the platform data  
 \* of the resulting platform\_device  
 \*/***struct** mfd\_cell {  
 **const char** \*name;  
 **int** id;  
  
 */\* refcounting for multiple drivers to use a single cell \*/* atomic\_t \*usage\_count;  
 **int** (\*enable)(**struct** platform\_device \*dev);  
 **int** (\*disable)(**struct** platform\_device \*dev);  
  
 **int** (\*suspend)(**struct** platform\_device \*dev);  
 **int** (\*resume)(**struct** platform\_device \*dev);  
  
 */\* platform data passed to the sub devices drivers \*/* **void** \*platform\_data;  
 size\_t pdata\_size;  
 */\*  
 \* Device Tree compatible string  
 \* See: Documentation/devicetree/usage-model.txt Chapter 2.2 for details  
 \*/* **const char** \*of\_compatible;  
  
 */\* Matches ACPI \*/* **const struct** mfd\_cell\_acpi\_match \*acpi\_match;  
  
 */\*  
 \* These resources can be specified relative to the parent device.  
 \* For accessing hardware you should use resources from the platform dev  
 \*/* **int** num\_resources;  
 **const struct** resource \*resources;  
  
 */\* don't check for resource conflicts \*/* bool ignore\_resource\_conflicts;  
  
 */\*  
 \* Disable runtime PM callbacks for this subdevice - see  
 \* pm\_runtime\_no\_callbacks().  
 \*/* bool pm\_runtime\_no\_callbacks;  
  
 */\* A list of regulator supplies that should be mapped to the MFD  
 \* device rather than the child device when requested  
 \*/* **const char** \* **const** \*parent\_supplies;  
 **int** num\_parent\_supplies;  
};

部分常见的成员介绍如下：

name：设备平台。

platform\_data：平台私有数据指针，数据大小使用pdata\_size表示。

resources：资源结构体，资源数量使用num\_resources表示。

ignore\_resource\_conflicts：为true表示不检查资源冲突

of\_compatible：设备树匹配compatible的字符串（具体参考Documentation/devicetree/usage-model.txt Chapter 2.2）这个根据我的理解，是用于platform device的，只是写在了mfd设备上

## mfd设备添加

mfd核心代码位于drivers/mfd/mfd-core.c文件中。对外提供添加设备和删除设备的接口：mfd\_add\_devices、mfd\_remove\_devices。设备添加函数原型如下：

int mfd\_add\_devices(struct device \*parent, int id,

const struct mfd\_cell \*cells, int n\_devs,

struct resource \*mem\_base,

int irq\_base, struct irq\_domain \*domain)

下面主要分析其中一部分参数。

id：即设备ID号。它指示着设备的个数。一般可以设置为-1。即表示系统有且仅有一个这样的设备。如果有多个foo设备，则需要使用id来区别。在/sys/bus/platform/devices目录下会产生foo.0，foo.1等设备。详情可以看platform设备添加函数过程。

celss：即mfd\_cell结构体数组，n\_devs为其数组大小，即设备数量。

mem\_base：资源resource结构体

mfd\_add\_devices函数内部根据设备数量n\_devs循环调用mfd\_add\_device添加设备。该函数完成下面的工作：

1、申请platform\_device空间。申请resource空间。

2、调用platform\_device\_add\_data添加platform设备私有数据，亦即platform\_data。

3、调用mfd\_platform\_add\_cell将mfd\_cell拷贝到platform\_device的mfd\_cell成员。(使用kmemdup实现)

4、根据参数，设置申请到的resource空间。并调用platform\_device\_add\_resources添加到platform\_device的resource成员。这样就能在platform驱动模块中获取到resource资源了。

５、调用platform\_device\_add添加platform设备。此过程中会调用到对应驱动的probe函数——当然，提前是已经存在对应的驱动。

至此，mfd设备的添加就完成了，最终调用驱动的probe函数。从这个过程中知道，mfd实质上就是封装一个接口，将一些可以归纳到一起的platform设备注册到platform总线上。它就是一个收纳盒子。里面的设备该是怎样处理就怎样处理

# 示例分析

## 分析tps6507x的多功能驱动

### 涉及的文件

　　　　drivers/mfd/tps6507x.c

　　　　include/linux/mfd/tps6507x.h

　　　　drivers/regulator/tps6507x-regulator.c

　　　　drivers/input/touchscreen/tps6507x-ts.c

　drivers/regulator/tps6507x-regulator.c,这里面实现电源管理功能（电压调节器驱动）

　drivers/input/touchscreen/tps6507x-ts.c,这里面实现触摸屏功能

### 涉及的结构体

**static const struct** mfd\_cell tps6507x\_devs[] = {  
 {  
 .name = **"tps6507x-pmic"**,  
 },  
 {  
 .name = **"tps6507x-ts"**,  
 },  
};

从以上结构体可以得出,tps6507x系列芯片提供两种功能: 电源管理功能(regulator)+触摸屏功能(touchscreen)

**static struct** i2c\_driver tps6507x\_i2c\_driver = {  
 .driver = {  
 .name = **"tps6507x"**,  
 .of\_match\_table = of\_match\_ptr(tps6507x\_of\_match),  
 },  
 .probe = tps6507x\_i2c\_probe,  
 .remove = tps6507x\_i2c\_remove,  
 .id\_table = tps6507x\_i2c\_id,  
};

这个结构体为tps6507x提供探测函数tps6507x\_i2c\_probe

include/linux/mfd/tps6507x.h

*/\*\*  
 \* struct tps6507x\_dev - tps6507x sub-driver chip access routines  
 \* @read\_dev() - I2C register read function  
 \* @write\_dev() - I2C register write function  
 \*  
 \* Device data may be used to access the TPS6507x chip  
 \*/***struct** tps6507x\_dev {  
 **struct** device \*dev;  
 **struct** i2c\_client \*i2c\_client;  
 **int** (\*read\_dev)(**struct** tps6507x\_dev \*tps6507x, **char** reg, **int** size,  
 **void** \*dest);  
 **int** (\*write\_dev)(**struct** tps6507x\_dev \*tps6507x, **char** reg, **int** size,  
 **void** \*src);  
  
 */\* Client devices \*/* **struct** tps6507x\_pmic \*pmic;  
};

tps6507x 的读写接口就是放在这个结构体中,这也就是所谓的共性

### 加载驱动

drivers/mfd/tps6507x.c

**static int** \_\_init tps6507x\_i2c\_init(**void**)  
{  
 **return** i2c\_add\_driver(&tps6507x\_i2c\_driver);  
}  
*/\* init early so consumer devices can complete system boot \*/*subsys\_initcall(tps6507x\_i2c\_init);

#### tps6507x\_i2c\_probe

作用：注册tps6507x的读写函数: tps6507x\_i2c\_read\_device和tps6507x\_i2c\_write\_device到结构体struct tps6507x\_dev中

**static int** tps6507x\_i2c\_probe(**struct** i2c\_client \*i2c,  
 **const struct** i2c\_device\_id \*id)  
{  
 **struct** tps6507x\_dev \*tps6507x;  
  
 tps6507x = devm\_kzalloc(&i2c->dev, **sizeof**(**struct** tps6507x\_dev),  
 GFP\_KERNEL);  
 **if** (tps6507x == NULL)  
 **return** -ENOMEM;  
  
 i2c\_set\_clientdata(i2c, tps6507x);  
 tps6507x->dev = &i2c->dev;  
 tps6507x->i2c\_client = i2c;  
 tps6507x->read\_dev = tps6507x\_i2c\_read\_device;  
 tps6507x->write\_dev = tps6507x\_i2c\_write\_device;  
  
 **return** mfd\_add\_devices(tps6507x->dev, -1, tps6507x\_devs,  
 ARRAY\_SIZE(tps6507x\_devs), NULL, 0, NULL);  
}

### tps6507x电压调节器驱动

drivers/regulator/tps6507x-regulator.c,这里面实现电源管理功能（电压调节器驱动）

**static struct** platform\_driver tps6507x\_pmic\_driver = {  
 .driver = {  
 .name = **"tps6507x-pmic"**,  
 },  
 .probe = tps6507x\_pmic\_probe,  
};  
  
**static int** \_\_init tps6507x\_pmic\_init(**void**)  
{  
 **return** platform\_driver\_register(&tps6507x\_pmic\_driver);  
}  
subsys\_initcall(tps6507x\_pmic\_init);

#### tps6507x\_pmic\_probe

探测函数tps6507x\_pmic\_probe作用：

获取共用的结构体struct tps6507x\_dev

　　　　　　再注册相关的结构体以便提供pmic的相关操作接口,如下:

**static int** tps6507x\_pmic\_probe(**struct** platform\_device \*pdev)  
{  
 **struct** tps6507x\_dev \*tps6507x\_dev = dev\_get\_drvdata(pdev->dev.parent);  
 **struct** tps\_info \*info = &tps6507x\_pmic\_regs[0];  
 **struct** regulator\_config config = { };  
 **struct** regulator\_init\_data \*init\_data;  
 **struct** regulator\_dev \*rdev;  
 **struct** tps6507x\_pmic \*tps;  
 **struct** tps6507x\_board \*tps\_board;  
 **struct** of\_regulator\_match \*tps6507x\_reg\_matches = NULL;  
 **int** i;  
 **int** error;  
 **unsigned int** prop;  
  
 */\*\*  
 \* tps\_board points to pmic related constants  
 \* coming from the board-evm file.  
 \*/* tps\_board = dev\_get\_platdata(tps6507x\_dev->dev);  
 **if** (IS\_ENABLED(CONFIG\_OF) && !tps\_board &&  
 tps6507x\_dev->dev->of\_node)  
 tps\_board = tps6507x\_parse\_dt\_reg\_data(pdev,  
 &tps6507x\_reg\_matches);  
 **if** (!tps\_board)  
 **return** -EINVAL;  
  
 */\*\*  
 \* init\_data points to array of regulator\_init structures  
 \* coming from the board-evm file.  
 \*/* init\_data = tps\_board->tps6507x\_pmic\_init\_data;  
 **if** (!init\_data)  
 **return** -EINVAL;  
  
 tps = devm\_kzalloc(&pdev->dev, **sizeof**(\*tps), GFP\_KERNEL);  
 **if** (!tps)  
 **return** -ENOMEM;  
  
 mutex\_init(&tps->io\_lock);  
  
 */\* common for all regulators \*/* tps->mfd = tps6507x\_dev;  
  
 **for** (i = 0; i < TPS6507X\_NUM\_REGULATOR; i++, info++, init\_data++) {  
 */\* Register the regulators \*/* tps->info[i] = info;  
 **if** (init\_data->driver\_data) {  
 **struct** tps6507x\_reg\_platform\_data \*data =  
 init\_data->driver\_data;  
 tps->info[i]->defdcdc\_default = data->defdcdc\_default;  
 }  
  
 tps->desc[i].name = info->name;  
 tps->desc[i].id = i;  
 tps->desc[i].n\_voltages = info->table\_len;  
 tps->desc[i].volt\_table = info->table;  
 tps->desc[i].ops = &tps6507x\_pmic\_ops;  
 tps->desc[i].type = REGULATOR\_VOLTAGE;  
 tps->desc[i].owner = THIS\_MODULE;  
  
 config.dev = tps6507x\_dev->dev;  
 config.init\_data = init\_data;  
 config.driver\_data = tps;  
  
 **if** (tps6507x\_reg\_matches) {  
 error = of\_property\_read\_u32(  
 tps6507x\_reg\_matches[i].of\_node,  
 **"ti,defdcdc\_default"**, &prop);  
  
 **if** (!error)  
 tps->info[i]->defdcdc\_default = prop;  
  
 config.of\_node = tps6507x\_reg\_matches[i].of\_node;  
 }  
  
 rdev = devm\_regulator\_register(&pdev->dev, &tps->desc[i],  
 &config);  
 **if** (IS\_ERR(rdev)) {  
 dev\_err(tps6507x\_dev->dev,  
 **"failed to register %s regulator\n"**,  
 pdev->name);  
 **return** PTR\_ERR(rdev);  
 }  
  
 */\* Save regulator for cleanup \*/* tps->rdev[i] = rdev;  
 }  
  
 tps6507x\_dev->pmic = tps;  
 platform\_set\_drvdata(pdev, tps6507x\_dev);  
  
 **return** 0;  
}

#### tps6507x\_pmic\_ops

**static struct** regulator\_ops tps6507x\_pmic\_ops = {  
.is\_enabled = tps6507x\_pmic\_is\_enabled, 检查tps6507x的pmic功能是否已经使能了

.enable = tps6507x\_pmic\_enable, 使能tps6507x的pmic功能

.disable = tps6507x\_pmic\_disable, 禁用tsp6507x的pmic功能

.get\_voltage\_sel = tps6507x\_pmic\_get\_voltage\_sel, 获取电压值

.set\_voltage\_sel = tps6507x\_pmic\_set\_voltage\_sel, 设置电压值

.list\_voltage = regulator\_list\_voltage\_table, 列出电压表

.map\_voltage = regulator\_map\_voltage\_ascend,  
};

### tps6507x触摸屏驱动

drivers/input/touchscreen/tps6507x-ts.c

#### 分析probe函数都做了些什么?

　　　　　　获取公用的结构体struct tps6507x\_dev

　　　　　　填充结构体struct tps6507x\_ts，关键是注册了函数tps6507x\_ts\_poll

## 分析da9063驱动

8.2.1 mfd驱动

　　　　8.2.1.1 相关源码

　　　　　　drivers/mfd/da9063-i2c.c

　　　　8.2.1.2 分析探测函数da9063\_i2c\_probe的调用路径

　　　　　　da9063\_i2c\_probe->da9063\_device\_init

　　　　8.2.1.3 da9063\_device\_init做了些什么?

　　　　　　读取da9063的芯片ID，检查是否匹配

　　　　　　读取da9063的variant ID,不同的variant ID表示不同的封装

　　　　　　通过接口devm\_mfd\_add\_devices添加具体的结构体struct mfd\_cell数组,这个数组里包含了多个驱动相关的信息,如名字，资源等

　　　　8.2.1.4 结构体数组da906\_common\_devs

[复制代码](javascript:void(0);)

static const struct mfd\_cell da9063\_common\_devs[] = {

{

.name = DA9063\_DRVNAME\_REGULATORS,

.num\_resources = ARRAY\_SIZE(da9063\_regulators\_resources),

.resources = da9063\_regulators\_resources,

},

{

.name = DA9063\_DRVNAME\_LEDS,

},

{

.name = DA9063\_DRVNAME\_WATCHDOG,

.of\_compatible = "dlg,da9063-watchdog",

},

{

.name = DA9063\_DRVNAME\_HWMON,

.num\_resources = ARRAY\_SIZE(da9063\_hwmon\_resources),

.resources = da9063\_hwmon\_resources,

},

{

.name = DA9063\_DRVNAME\_ONKEY,

.num\_resources = ARRAY\_SIZE(da9063\_onkey\_resources),

.resources = da9063\_onkey\_resources,

.of\_compatible = "dlg,da9063-onkey",

},

{

.name = DA9063\_DRVNAME\_VIBRATION,

},

};

[复制代码](javascript:void(0);)

　　　　　　这个结构体数组中就包含了调节器，led控制器，看门狗，硬件监测（电压监测，温度监测），长按关键功能（onkey），震动等驱动名称，也就是da9063会关联（具有）这些功能,da9063有两种硬件版本，一种为DA9063,另一种为DA9063L,这两种硬件的差异在于DA9063具有实时时钟功能，而后者没有此功能

　　　　8.2.1.5 结构体数组da9063\_devs

[复制代码](javascript:void(0);)

/\* Only present on DA9063 , not on DA9063L \*/

static const struct mfd\_cell da9063\_devs[] = {

{

.name = DA9063\_DRVNAME\_RTC,

.num\_resources = ARRAY\_SIZE(da9063\_rtc\_resources),

.resources = da9063\_rtc\_resources,

.of\_compatible = "dlg,da9063-rtc",

},

};

[复制代码](javascript:void(0);)

　　　　8.2.1.6 da9063\_common\_devs和da9063\_devs中的这些具体的驱动实现在哪里?

　　　　　　　　drivers/regulator/da9063-regulator.c (调节器驱动)

　　　　　　　　(没有找到da9063的led控制器驱动)

　　　　　　　　drivers/watchdog/da9063\_wdt.c (看门狗驱动)

　　　　　　　　(没有找到da9063的硬件监测驱动)

　　　　　　　　drivers/input/misc/da9063\_onkey.c （onkey驱动）

　　　　　　　　(没有找到da9063的震动功能驱动)

　　　　　　　　drivers/rtc/rtc-da9063.c (实时时钟驱动)

　　　　8.2.1.7 重要的结构体struct da9063

[复制代码](javascript:void(0);)

struct da9063 {

/\* Device \*/

struct device \*dev;

enum da9063\_type type;

unsigned char variant\_code;

unsigned int flags;

/\* Control interface \*/

struct regmap \*regmap;

/\* Interrupts \*/

int chip\_irq;

unsigned int irq\_base;

struct regmap\_irq\_chip\_data \*regmap\_irq;

};

## lpc驱动

本节介绍一下LPC驱动中WDT设备添加的过程。——因为前面讲了GPIO设备的添加。

e3800系列的WDT隐藏于ACPI中。后续文章将会进行介绍。这里有个概念即可。

在LPC探测函数lpc\_ich\_probe对ACPI基地址进行赋值，代码如下：

    priv->abase = ACPIBASE; // ACPI基地址

    priv->actrl\_pbase = ACPICTRL\_PMCBASE;

其定义是：

#define ACPIBASE 0x40

#define ACPICTRL\_PMCBASE 0x44

所有地址都可以在手册对应章节中找到。

初始化WDT在函数lpc\_ich\_init\_wdt中。这个函数获取ACPI基地址。并初始化wdt\_ich\_res资源结构体数组，数组包含了ICH\_RES\_IO\_TCO和ICH\_RES\_IO\_SMI。TCO就是WDT使用到的部分。主要代码功能描述如下。

1、读取ACPI基地址值，即通过LPC这个PCI设备的配置空间偏移值ACPIBASE。

pci\_read\_config\_dword(dev, priv->abase, &base\_addr\_cfg);

base\_addr = base\_addr\_cfg & 0x0000ff80;

2、设置resource，即把前面获取到的base\_addr地址加上TCO偏移值赋给resource的start成员变量。

res = wdt\_io\_res(ICH\_RES\_IO\_TCO);

res->start = base\_addr + ACPIBASE\_TCO\_OFF;

res->end = base\_addr + ACPIBASE\_TCO\_END;

3、添加mfd设备。

lpc\_ich\_finalize\_cell(dev, &lpc\_ich\_cells[LPC\_WDT]);

ret = mfd\_add\_devices(&dev->dev, -1, &lpc\_ich\_cells[LPC\_WDT],

1, NULL, 0, NULL);

到这样就完成了mfd的添加。

————————————————

版权声明：本文为CSDN博主「李迟」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/subfate/article/details/53464641>

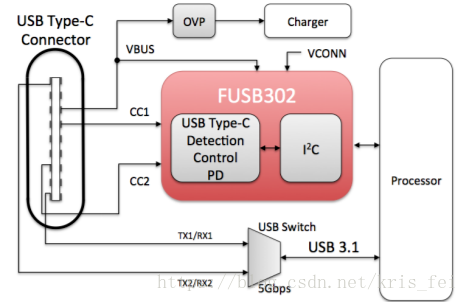
# fusb302

FUSB302是仙童FAIRCHILD推出的一款旨在帮助系统设计人员实现具有少量可编程性的DRP/SRC/SNKUSB Type-C接口

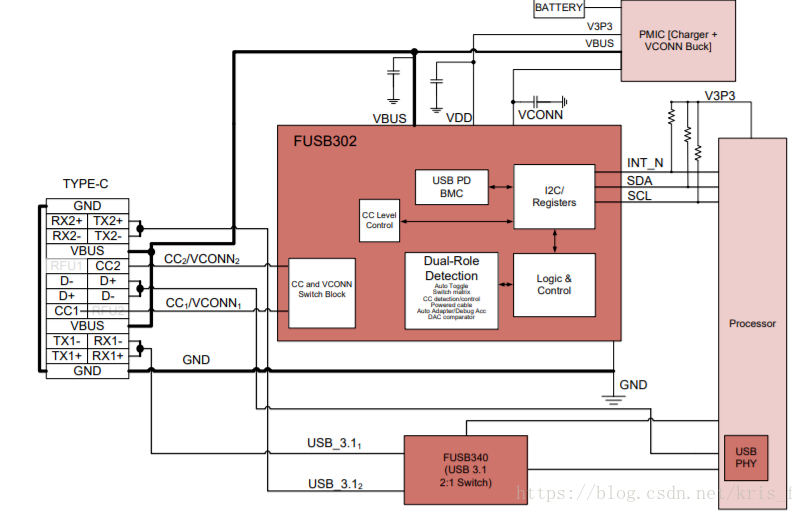
由于usb 3.0的type-c接口需要支持不同电压的外设(5V, 12V等)，如果不做控制，那么设置12V接5V的外设将会出问题。

## 电路介绍

fusb302可以实现此控制，根据不同的外设电压来调整电流，框图



**应用电路**



其中要注意的是INT\_N引脚，此pin会接到processor端的gpio，当有usb插拔时，INT\_N pin会被拉低，以通知cpu通过I2C去读取USB状态信息，如果dts中gpio配置得不对，usb也将无法被识别。

调试的时候可以看下/proc/interrupts中有没有fusb302的中断信息，或者直接在驱动(drivers/mfd/fusb302.c)中加Log

rk3399-mid-818-android.dts：

fusb0: fusb30x@22 {

compatible = "fairchild,fusb302";

reg = <0x22>;

pinctrl-names = "default";

pinctrl-0 = <&fusb0\_int>;

int-n-gpios = <&gpio1 1 GPIO\_ACTIVE\_HIGH>; //即gpio1 A1。

status = "okay";

};

//这里也要一起修改

fusb30x {

fusb0\_int: fusb0-int {

rockchip,pins =

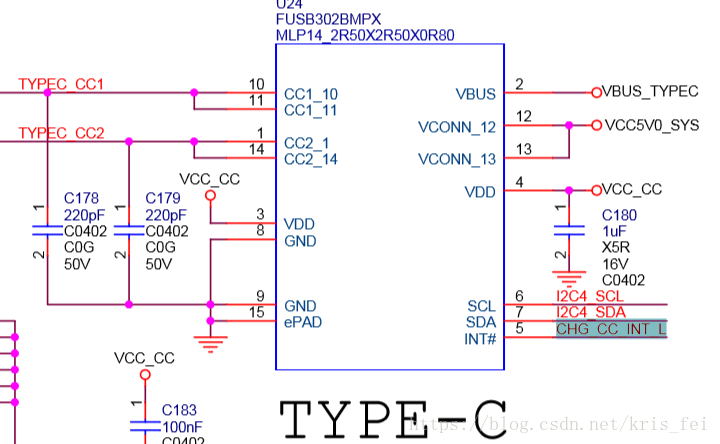
<1 1 RK\_FUNC\_GPIO &pcfg\_pull\_up>;

};

};

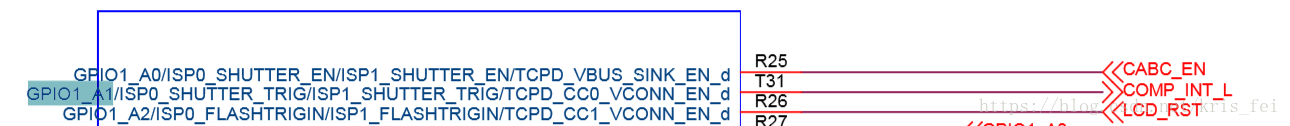
1

对应原理图：



————————————————

rk参考设计用的是gpio1 A2，我们做了修改



## 源码分析

### 驱动模块加载框架

module\_i2c\_driver。I2C设备驱动。Linux内核给出的接口只有两个，一个是注册，另一个就是卸载。module\_i2c\_driver这个宏定义，因为有它的存在，I2C设备驱动的开发可以不用在意你的I2C驱动需要如何注册以及如何卸载的，全部的精力都放在i2c\_driver的完善上就可以了。

static const struct of\_device\_id fusb30x\_dt\_match[] = {  
 { .compatible = FUSB30X\_I2C\_DEVICETREE\_NAME },  
 {},  
};  
MODULE\_DEVICE\_TABLE(of, fusb30x\_dt\_match);  
  
static const struct i2c\_device\_id fusb30x\_i2c\_device\_id[] = {  
 { FUSB30X\_I2C\_DRIVER\_NAME, 0 },  
 {}  
};  
MODULE\_DEVICE\_TABLE(i2c, fusb30x\_i2c\_device\_id);  
  
static struct i2c\_driver fusb30x\_driver = {  
 .driver = {  
 .name = FUSB30X\_I2C\_DRIVER\_NAME,  
 .of\_match\_table = of\_match\_ptr(fusb30x\_dt\_match),  
 },  
 .probe = fusb30x\_probe,  
 .remove = fusb30x\_remove,  
 .id\_table = fusb30x\_i2c\_device\_id,  
};  
  
module\_i2c\_driver(fusb30x\_driver);  
  
MODULE\_LICENSE("GPL");  
MODULE\_AUTHOR("zain wang <zain.wang@rock-chips.com>");  
MODULE\_DESCRIPTION("fusb302 typec pd driver");

为何这里fusb30x\_i2c\_device\_id要写两次，fusb30x\_driver定义和MODULE\_DEVICE\_TABLE

Type-C中的PD的意思指是:USB Power Delivery功率传输协议。

USB PD 协议基于USB3.1，是USB3.1 中即type-c端口后提出的功率传输概念。可以为这种技术带来更大的灵活性，将充电能力扩大为目前的10倍：最高可达100瓦

<https://zhidao.baidu.com/question/1772307446156134100.html>

#define FUSB30X\_I2C\_DRIVER\_NAME "fusb302"

#### id\_table

驱动名字用来干嘛的呢？

static const struct i2c\_device\_id fusb30x\_i2c\_device\_id[] = {  
 { FUSB30X\_I2C\_DRIVER\_NAME, 0 },  
 {}  
};

#### of\_match\_table

static const struct of\_device\_id fusb30x\_dt\_match[] = {  
 { .compatible = FUSB30X\_I2C\_DEVICETREE\_NAME },  
 {},  
};

#define FUSB30X\_I2C\_DEVICETREE\_NAME "fairchild,fusb302"

设备描述表匹配，arch/arm64/boot/dts/rockchip/rk3399-box-rev1.dts

&i2c4 {  
 status = "okay";  
 fusb0: fusb30x@22 {  
 compatible = "fairchild,fusb302";  
 reg = <0x22>;  
 pinctrl-names = "default";  
 pinctrl-0 = <&fusb0\_int>;  
 vbus-5v-gpios = <&gpio1 3 GPIO\_ACTIVE\_LOW>;  
 int-n-gpios = <&gpio1 2 GPIO\_ACTIVE\_HIGH>;  
 status = "okay";  
 };  
};

#### fusb30x\_probe

### 结构体

#### fusb30x\_chip

struct fusb30x\_chip {

struct i2c\_client \*client;  
struct device \*dev;  
struct regmap \*regmap;  
struct work\_struct work;  
struct workqueue\_struct \*fusb30x\_wq;  
struct hrtimer timer\_state\_machine;//状态机定时器  
struct hrtimer timer\_mux\_machine;  
struct PD\_CAP\_INFO pd\_cap\_info;  
struct notify\_info notify;  
struct notify\_info notify\_cmp;  
struct extcon\_dev \*extcon;  
enum connection\_state conn\_state;//连接状态  
struct gpio\_desc \*gpio\_vbus\_5v;//和其他芯片通信的gpio  
struct gpio\_desc \*gpio\_vbus\_other;  
struct gpio\_desc \*gpio\_int;//用于中断？？  
int timer\_state;//状态值  
int timer\_mux;  
int port\_num;  
int work\_continue;  
spinlock\_t irq\_lock;  
int gpio\_int\_irq; //用于中断？？  
int enable\_irq;//用于中断使能？  
  
/\*  
 \* ---------------------------------  
 \* | role 0x03 << 2, | cc\_use 0x03 |  
 \* | src 1 << 2, | cc1 1 |  
 \* | snk 2 << 2, | cc2 2 |  
 \* ---------------------------------  
 \*/  
u8 cc\_state;//这个很重要，但是我理解不了  
int cc1;  
int cc2;  
/\* 0 cc1 : 1 cc2 \*/  
bool cc\_polarity;  
u8 val\_tmp;  
u8 debounce\_cnt;  
int sub\_state;  
int caps\_counter;  
u32 send\_load[7];  
u32 rec\_load[7];  
u16 send\_head;  
u16 rec\_head;  
int msg\_id;  
enum tx\_state tx\_state;  
int hardrst\_count;  
u32 source\_power\_supply[7];  
/\* 50mv unit \*/  
u32 source\_max\_current[7];  
/\* 10ma uint\*/  
int pos\_power;  
/\*  
 \* if PartnerCap[0] == 0xffffffff  
 \* show Partner Device do not support supply  
 \*/  
u32 partner\_cap[7];  
int n\_caps\_used;  
int vdm\_state;  
int vdm\_substate;  
int vdm\_send\_state;  
u32 dp\_status;  
u16 vdm\_svid[12];  
int vdm\_svid\_num;  
u32 vdm\_id;  
u8 chip\_id;  
bool vconn\_enabled;  
int togdone\_pullup;

#### connection\_state

### fusb30x\_probe

static int fusb30x\_probe(struct i2c\_client \*client,  
 const struct i2c\_device\_id \*id)  
{  
 struct fusb30x\_chip \*chip;  
 struct PD\_CAP\_INFO \*pd\_cap\_info;  
 int ret;

#### 为芯片结构体分配内存

fusb30x\_chip结构体分配内存。并指定一个port\_num，

static u8 fusb30x\_port\_used;  
static struct fusb30x\_chip \*fusb30x\_port\_info[256];

这里最大允许挂载255个同类型的设备

chip = devm\_kzalloc(&client->dev, sizeof(\*chip), GFP\_KERNEL);  
if (!chip)  
 return -ENOMEM;  
  
if (fusb30x\_port\_used == 0xff)  
 return -1;  
  
chip->port\_num = fusb30x\_port\_used++;  
fusb30x\_port\_info[chip->port\_num] = chip;

#### 为寄存器分配

为寄存器分配存储空间

chip->dev = &client->dev;  
chip->regmap = devm\_regmap\_init\_i2c(client, &fusb302\_regmap\_config);

struct regmap\_config fusb302\_regmap\_config = {  
 .reg\_bits = 8,  
 .val\_bits = 8,  
 .writeable\_reg = is\_write\_reg,  
 .volatile\_reg = is\_volatile\_reg,  
 .max\_register = FUSB302\_MAX\_REG,  
 .cache\_type = REGCACHE\_RBTREE,  
};

#### 初始化GPIO

其实就是解析dts文件

ret = fusb\_initialize\_gpio(chip);

dts描述为：

vbus-5v-gpios = <&gpio1 3 GPIO\_ACTIVE\_LOW>;  
int-n-gpios = <&gpio1 2 GPIO\_ACTIVE\_HIGH>;

static int fusb\_initialize\_gpio(struct fusb30x\_chip \*chip)  
{  
 chip->gpio\_int = devm\_gpiod\_get\_optional(chip->dev, "int-n", GPIOD\_IN);  
 /\* some board support vbus with other ways \*/  
 chip->gpio\_vbus\_5v = devm\_gpiod\_get\_optional(chip->dev, "vbus-5v", PIOD\_OUT\_LOW)  
 gpiod\_set\_raw\_value(chip->gpio\_vbus\_5v, 0);  
 chip->gpio\_vbus\_other = devm\_gpiod\_get\_optional(chip->dev,  
 "vbus-other",  
 GPIOD\_OUT\_LOW);  
 gpiod\_set\_raw\_value(chip->gpio\_vbus\_other, 0);  
  
 return 0;  
}

devm\_gpiod\_get\_optional只是对gpiod\_get\_index的包装而已，并且index为0。gpiod\_get\_index()本质上和gpio\_request()是一个功能，是申请gpio的，只是它是从device tree去查找， 因此看到第二个参数”con\_id”是字符串类型，也就是gpio的名字。

xxx-gpios可以为数组的，比如

led-gpios = <&gpio 15 GPIO\_ACTIVE\_HIGH>, /\* red \*/

<&gpio 16 GPIO\_ACTIVE\_HIGH>, /\* green \*/

<&gpio 17 GPIO\_ACTIVE\_HIGH>; /\* blue \*/

如果index是0,那么对应的就是gpio 15; 　如果index是1,那么对应就是gpio 16，以此类推

#### 初始化定时器

我们的定时器也该这么写的，这里的timer\_handler是共用的？？

fusb\_initialize\_timer(chip);

static void fusb\_initialize\_timer(struct fusb30x\_chip \*chip)  
{  
 hrtimer\_init(&chip->timer\_state\_machine, CLOCK\_MONOTONIC,  
 HRTIMER\_MODE\_REL);  
 chip->timer\_state\_machine.function = fusb\_timer\_handler;  
  
 hrtimer\_init(&chip->timer\_mux\_machine, CLOCK\_MONOTONIC,  
 HRTIMER\_MODE\_REL);  
 chip->timer\_mux\_machine.function = fusb\_timer\_handler;  
  
 chip->timer\_state = T\_DISABLED;  
 chip->timer\_mux = T\_DISABLED;  
}

#### 初始化工作队列

chip->fusb30x\_wq = create\_workqueue("fusb302\_wq");  
INIT\_WORK(&chip->work, fusb302\_work\_func);

static void fusb302\_work\_func(struct work\_struct \*work)

{

struct fusb30x\_chip \*chip;

chip = container\_of(work, struct fusb30x\_chip, work);

state\_machine\_typec(chip);

}

把工作轮询工作和状态机连接起来，提供给内核调用

#### Tcpm初始化

Tcpm是个什么玩意？？

tcpm\_init(chip);  
tcpm\_set\_rx\_enable(chip, 0);  
chip->conn\_state = unattached;  
tcpm\_set\_cc(chip, FUSB\_MODE\_DRP);

chip->n\_caps\_used = 1;  
chip->source\_power\_supply[0] = 0x64;  
chip->source\_max\_current[0] = 0x96;

#### PD\_CAP\_INFO初始化

/\*  
 \* these two variable should be 1 if support DRP,  
 \* but now we do not support swap,  
 \* it will be blanked in future  
 \*/  
pd\_cap\_info = &chip->pd\_cap\_info;  
pd\_cap\_info->dual\_role\_power = 0;  
pd\_cap\_info->data\_role\_swap = 0;  
  
pd\_cap\_info->externally\_powered = 1;  
pd\_cap\_info->usb\_suspend\_support = 0;  
pd\_cap\_info->usb\_communications\_cap = 0;  
pd\_cap\_info->supply\_type = 0;  
pd\_cap\_info->peak\_current = 0;

#### 初始化extcon

chip->extcon = devm\_extcon\_dev\_allocate(&client->dev, fusb302\_cable);  
if (IS\_ERR(chip->extcon)) {  
 dev\_err(&client->dev, "allocat extcon failed\n");  
 return PTR\_ERR(chip->extcon);  
}  
  
ret = devm\_extcon\_dev\_register(&client->dev, chip->extcon);  
if (ret) {  
 dev\_err(&client->dev, "failed to register extcon: %d\n",  
 ret);  
 return ret;  
}  
  
ret = extcon\_set\_property\_capability(chip->extcon, EXTCON\_USB,  
 EXTCON\_PROP\_USB\_TYPEC\_POLARITY);  
if (ret) {  
 dev\_err(&client->dev,  
 "failed to set USB property capability: %d\n",  
 ret);  
 return ret;  
}  
  
ret = extcon\_set\_property\_capability(chip->extcon, EXTCON\_USB\_HOST,  
 EXTCON\_PROP\_USB\_TYPEC\_POLARITY);  
if (ret) {  
 dev\_err(&client->dev,  
 "failed to set USB\_HOST property capability: %d\n",  
 ret);  
 return ret;  
}  
  
ret = extcon\_set\_property\_capability(chip->extcon, EXTCON\_DISP\_DP,  
 EXTCON\_PROP\_USB\_TYPEC\_POLARITY);  
if (ret) {  
 dev\_err(&client->dev,  
 "failed to set DISP\_DP property capability: %d\n",  
 ret);  
 return ret;  
}  
  
ret = extcon\_set\_property\_capability(chip->extcon, EXTCON\_USB,  
 EXTCON\_PROP\_USB\_SS);  
if (ret) {  
 dev\_err(&client->dev,  
 "failed to set USB USB\_SS property capability: %d\n",  
 ret);  
 return ret;  
}  
  
ret = extcon\_set\_property\_capability(chip->extcon, EXTCON\_USB\_HOST,  
 EXTCON\_PROP\_USB\_SS);  
if (ret) {  
 dev\_err(&client->dev,  
 "failed to set USB\_HOST USB\_SS property capability: %d\n",  
 ret);  
 return ret;  
}  
  
ret = extcon\_set\_property\_capability(chip->extcon, EXTCON\_DISP\_DP,  
 EXTCON\_PROP\_USB\_SS);  
if (ret) {  
 dev\_err(&client->dev,  
 "failed to set DISP\_DP USB\_SS property capability: %d\n",  
 ret);  
 return ret;  
}

#### gpiod\_to\_irq中断分配

enable\_irq有用的么？

i2c\_set\_clientdata(client, chip);  
  
spin\_lock\_init(&chip->irq\_lock);  
chip->enable\_irq = 1;  
  
chip->gpio\_int\_irq = gpiod\_to\_irq(chip->gpio\_int);  
if (chip->gpio\_int\_irq < 0) {  
 dev\_err(&client->dev,  
 "Unable to request IRQ for INT\_N GPIO! %d\n",  
 ret);  
 ret = chip->gpio\_int\_irq;  
 goto IRQ\_ERR;  
}  
  
ret = devm\_request\_threaded\_irq(&client->dev,  
 chip->gpio\_int\_irq,  
 NULL,  
 cc\_interrupt\_handler,  
 IRQF\_ONESHOT | IRQF\_TRIGGER\_LOW,  
 client->name,  
 chip);

### fusb30x\_remove

的主要把工作队列给释放掉

static int fusb30x\_remove(struct i2c\_client \*client)  
{  
 struct fusb30x\_chip \*chip = i2c\_get\_clientdata(client);  
  
 destroy\_workqueue(chip->fusb30x\_wq);  
 return 0;  
}

### 中断流程

看之前的电路图，注意的是INT\_N引脚，此pin会接到processor端的gpio，当有usb插拔时，INT\_N pin会被拉低，以通知cpu通过I2C去读取USB状态信息，如果dts中gpio配置得不对，usb也将无法被识别

#### 注册

在探测fusb30x\_probe的时候，注册一个中断。fusb\_initialize\_gpio申请到了gpio

chip->gpio\_int = devm\_gpiod\_get\_optional(chip->dev, "int-n", GPIOD\_IN);

然后把gpio转化为irq？

chip->enable\_irq = 1;  
//将gpio转为对应的irq，然后注册该irq的中断**handler**

//获取对应的中断号  
chip->gpio\_int\_irq = gpiod\_to\_irq(chip->gpio\_int);

ret = devm\_request\_threaded\_irq(&client->dev,  
 chip->gpio\_int\_irq,// 监听的中断号，eg:107  
 NULL,  
 cc\_interrupt\_handler, //实际的中断处理程序  
 IRQF\_ONESHOT | IRQF\_TRIGGER\_LOW,//标志位，IRQF\_TRIGGER\_LOW低电平触发中断  
 client->name,

chip);

\* name - 中断设备名称的ASCII 表示，这些会被/proc/irq和/proc/interrupts文件使用,

#define FUSB30X\_I2C\_DRIVER\_NAME "fusb302"

\* dev - 用于共享中断线，多个中断程序共享一个中断线时(共用一个中断号)，依靠dev来区别各个中断程序

\* 返回值：

\* 执行成功：0

\* 执行失败：非0

\*/

#### cc\_interrupt\_handler

**static** irqreturn\_t cc\_interrupt\_handler(**int** irq, **void** \*dev\_id)  
{  
 **struct** f**usb30x\_chip \*chip = dev\_id;//最好加类型保护**  
  
 queue\_work(chip->fusb30x\_wq, &chip->work);  
 fusb\_irq\_disable(chip);  
 **return** IRQ\_HANDLED;  
}

irq中断插拔肯定是有的，但是cc1

在关机装填

#### fusb\_irq\_disable

**void** fusb\_irq\_disable(**struct** fusb30x\_chip \*chip)  
{  
 **unsigned long** irqflags = 0;  
  
 spin\_lock\_irqsave(&chip->irq\_lock, irqflags);  
 **if** (chip->enable\_irq) {  
 disable\_irq\_nosync(chip->gpio\_int\_irq);  
 chip->enable\_irq = 0;  
 } **else** {  
 dev\_warn(chip->dev, **"irq have already disabled\n"**);  
 }  
 spin\_unlock\_irqrestore(&chip->irq\_lock, irqflags);  
}

为何需要触发中断，然后又禁止呢。fusb\_irq\_disable呢

因为要让中断处理程序去处理任务？？？

#### fusb\_irq\_enable

处理完成，在使能

**if** (!platform\_get\_device\_irq\_state(chip))  
 fusb\_irq\_enable(chip);

如果中断有效的话，**gpiod\_get\_value**

来获取/设置gpio端口的值，gpio寄存器

platform\_get\_device\_irq\_state(**struct** fusb30x\_chip \*chip)  
{  
 **return** !**gpiod\_get\_value(chip->gpio\_int)**;  
}

**if** (!chip->enable\_irq) {  
 enable\_irq(**chip->gpio\_int\_irq**);  
 chip->enable\_irq = 1;  
}

### dump\_notify\_info

打印思想好

dev\_dbg(chip->dev, "port %d\n", chip->port\_num);  
dev\_dbg(chip->dev, "orientation %d\n", chip->notify.orientation);  
dev\_dbg(chip->dev, "power\_role %d\n", chip->notify.power\_role);  
dev\_dbg(chip->dev, "data\_role %d\n", chip->notify.data\_role);  
dev\_dbg(chip->dev, "cc %d\n", chip->notify.is\_cc\_connected);  
dev\_dbg(chip->dev, "pd %d\n", chip->notify.is\_pd\_connected);  
dev\_dbg(chip->dev, "enter\_mode %d\n", chip->notify.is\_enter\_mode);  
dev\_dbg(chip->dev, "pin support %d\n",  
 chip->notify.pin\_assignment\_support);  
dev\_dbg(chip->dev, "pin def %d\n", chip->notify.pin\_assignment\_def);  
dev\_dbg(chip->dev, "attention %d\n", chip->notify.attention);

## 状态机原理state\_machine\_typec

### Chip检查

tcpc\_alert(chip, &evt);  
mux\_alert(chip, &evt);  
if (!evt)  
 goto BACK;

### 状态更新

Dfg

if (chip->notify.is\_cc\_connected) {  
 if (evt & EVENT\_CC) {  
 if ((chip->cc\_state & 0x04) &&  
 (chip->conn\_state !=  
 policy\_snk\_transition\_default)) {  
 if (!tcpm\_check\_vbus(chip))  
 set\_state\_unattached(chip);  
 } else if (chip->conn\_state !=  
 policy\_src\_transition\_default) {  
 tcpm\_get\_cc(chip, &cc1, &cc2);  
 if (!(chip->cc\_state & 0x01))  
 cc1 = cc2;  
 if (cc1 == TYPEC\_CC\_VOLT\_OPEN)  
 set\_state\_unattached(chip);  
 }  
 }  
}  
  
if (evt & EVENT\_RX) {  
 tcpm\_get\_message(chip);  
 if ((!PD\_HEADER\_CNT(chip->rec\_head)) &&  
 (PD\_HEADER\_TYPE(chip->rec\_head) == CMT\_SOFTRESET)) {  
 if (chip->notify.power\_role)  
 set\_state(chip, policy\_src\_send\_softrst);  
 else  
 set\_state(chip, policy\_snk\_send\_softrst);  
 }  
}  
  
if (evt & EVENT\_TX) {  
 if (chip->tx\_state == tx\_success)  
 chip->msg\_id++;  
}

### 状态处理

应该重构为状态处理函数就好了

Df

## 5v-GPIO通信

platform\_set\_vbus\_lvl\_enable

static void platform\_set\_vbus\_lvl\_enable(struct fusb30x\_chip \*chip, int vbus\_5v,  
 int vbus\_other)  
{  
 if (chip->gpio\_vbus\_5v) {  
 gpiod\_set\_raw\_value(chip->gpio\_vbus\_5v, vbus\_5v);  
 /\* Only set state here, don't sync notifier to PMIC \*/  
 extcon\_set\_state(chip->extcon, EXTCON\_USB\_VBUS\_EN, vbus\_5v);  
 } else {  
 extcon\_set\_state(chip->extcon, EXTCON\_USB\_VBUS\_EN, vbus\_5v);  
 extcon\_sync(chip->extcon, EXTCON\_USB\_VBUS\_EN);  
 dev\_info(chip->dev, "fusb302 send extcon to enable vbus 5v\n");  
 }  
  
 if (chip->gpio\_vbus\_other)  
 gpiod\_set\_raw\_value(chip->gpio\_vbus\_5v, vbus\_other);  
}

## 实战

### TODO

查看中断是否有产生

/proc/irq和/proc/interrupts文件使用,

#define FUSB30X\_I2C\_DRIVER\_NAME "fusb302"

验证一下这个irq中断号

cc\_interrupt\_handler(**int** irq,

platform\_get\_device\_irq\_state都是永真的？？

### 解决办法，

查看掉电的中断信息就好了和上电都会有中断？

还是因为platform\_get\_device\_irq\_state逻辑禁用了？？

把这个结构体的所有信息都打印出来吧

## RKK

RK3399 Type-c 模块需要外挂一个 fusb302 逻辑检测芯片来识别接入设备类型及 USB 的 正反插。Fusb302 的软件驱动在 dts 里面的配置如下：

rk3399-mid-818-android.dts：

fusb0: fusb30x@22 {

compatible = "fairchild,fusb302";

reg = <0x22>;

pinctrl-names = "default";

pinctrl-0 = <&fusb0\_int>;

int-n-gpios = <&gpio1 1 GPIO\_ACTIVE\_HIGH>; //我用的是gpio1 A1。

status = "okay";

};

中断脚配置：

&pinctrl {

fusb30x {

fusb0\_int: fusb0-int { rockchip,pins = ; };

}

; } 在 fusb302 及 usb phy 检测区分接入 **type-c 口的是哪一类设备**（充电器、USB、OTG、 DP 等）之后，通知系统，所以相关联的模块代码需要注册 fusb302 的 **extcon notifier** 来接收， 需要在模块 dts 配置加入 extcon = <&fusb0>。

如 rk818 dts 节点中加入 extcon = <&fusb0>，通过 fusb302 及 usb phy 检测区分充电 器、USB、OTG 的拔插后，rk818 模块决定相关的充电电流配置及 OTG 的开关。 目前 sdk 参考 dts 中默认 enable 了 fusb302 的配置，如果产品未使用 type-c 接口、未使 用 fusb302，产品 dts 中请 disabled 节点 tcphy0 及 fusb0，并将 USB 相关联的模块 dts 中 extcon = <&fusb0>改为 extcon = <&u2phy0>

## 参考

https://blog.csdn.net/kris\_fei/article/details/80803912

# 参考

linux内核中的MFD子系统

<https://www.cnblogs.com/dakewei/p/10991941.html>

https://bootlin.com/pub/conferences/2015/elce/belloni-mfd-regmap-syscon/belloni-mfd-regmap-syscon.pdf

我的内核学习笔记8：多功能设备mfd驱动

https://blog.csdn.net/subfate/article/details/53464641