# 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。

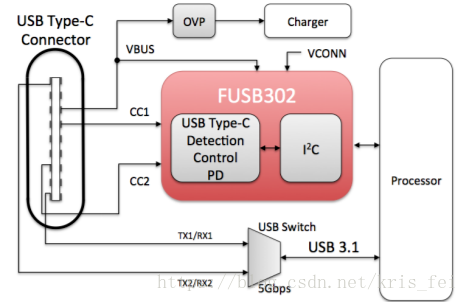
原文链接：<https://blog.csdn.net/subfate/article/details/53464641>

# fusb302

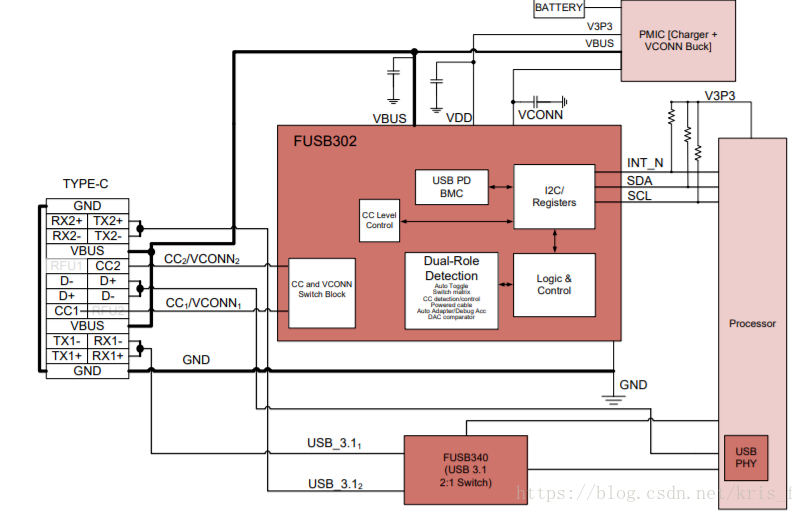
FUSB302是仙童FAIRCHILD推出的一款旨在帮助系统设计人员实现具有少量可编程性的DRP/SRC/SNKUSB Type-C接口

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。   
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>; //我用的是gpio1 A1。

status = "okay";

};

//这里也要一起修改

fusb30x {

fusb0\_int: fusb0-int {

rockchip,pins =

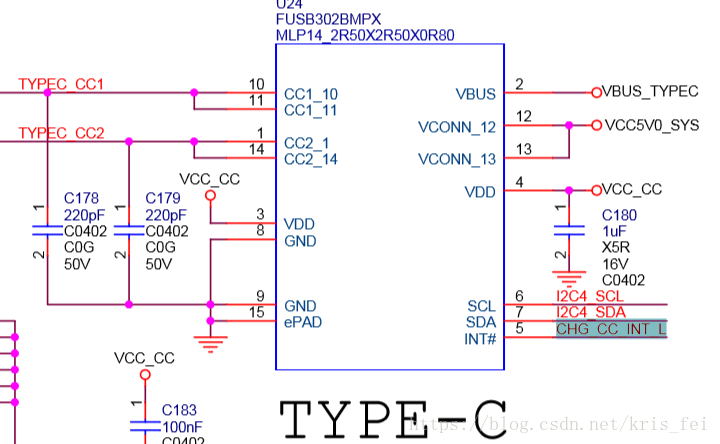
<1 1 RK\_FUNC\_GPIO &pcfg\_pull\_up>;

};

};

1

对应原理图：

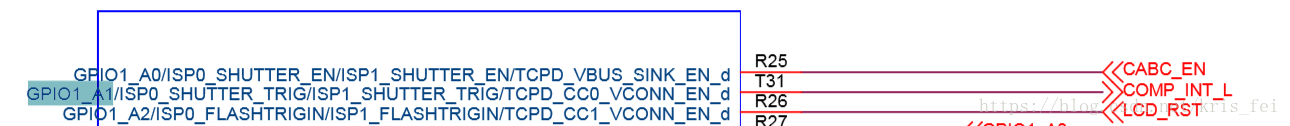


————————————————

版权声明：本文为CSDN博主「KrisFei」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/kris_fei/article/details/80803912>

rk参考设计用的是gpio1 A2，我们做了修改



## 实战

在探测的时候，注册一个中断

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

ret = devm\_request\_threaded\_irq(&client->dev,  
 chip->gpio\_int\_irq,  
 NULL,  
 cc\_interrupt\_handler,  
 IRQF\_ONESHOT | IRQF\_TRIGGER\_LOW,  
 client->name,  
 chip);  
**if** (ret) {  
 dev\_err(&client->dev, **"irq request failed\n"**);  
 **goto** IRQ\_ERR;  
}

### 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);  
}

### cc\_interrupt\_handler

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

#### 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);  
}

### 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);  
}

### 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