



# Unisoc Confidential For waterworld

## GPIO 和 EIC 客制化指导手册

文档版本  
发布日期


V1.9  
2023-06-09

## 版权所有 © 紫光展锐（上海）科技有限公司。保留一切权利。

本文件所含数据和信息都属于紫光展锐（上海）科技有限公司（以下简称紫光展锐）所有的机密信息，紫光展锐保留所有相关权利。本文件仅为信息参考之目的提供，不包含任何明示或默示的知识产权许可，也不表示有任何明示或默示的保证，包括但不限于满足任何特殊目的、不侵权或性能。当您接受这份文件时，即表示您同意本文件中内容和信息属于紫光展锐机密信息，且同意在未获得紫光展锐书面同意前，不使用或复制本文件的整体或部分，也不向任何其他方披露本文件内容。紫光展锐有权在未经事先通知的情况下，在任何时候对本文件做任何修改。紫光展锐对本文件所含数据和信息不做任何保证，在任何情况下，紫光展锐均不负任何与本文件相关的直接或间接的、任何伤害或损失。

请参照交付物中说明文档对紫光展锐交付物进行使用，任何人对紫光展锐交付物的修改、定制化或违反说明文档的指引对紫光展锐交付物进行使用造成的任何损失由其自行承担。紫光展锐交付物中的性能指标、测试结果和参数等，均为在紫光展锐内部研发和测试系统中获得的，仅供参考，若任何人需要对交付物进行商用或量产，需要结合自身的软硬件测试环境进行全面的测试和调试。

## 商标声明

**紫光展锐**、**UNISOC**、、展讯、Spreadtrum、SPRD、锐迪科、RDA 及其他紫光展锐的商标均为紫光展锐（上海）科技有限公司及/或其子公司、关联公司所有。

**Bluetooth®**文字商标和徽标为蓝牙技术联盟的注册商标，紫光展锐（上海）科技有限公司对此类商标的任何使用均已获得许可。

本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

## 免责声明

本文档可能包含第三方内容，包括但不限于第三方信息、软件、组件、数据等。紫光展锐不控制且不对第三方内容承担任何责任，包括但不限于准确性、兼容性、可靠性、可用性、合法性、适当性、性能、不侵权、更新状态等，除非本文档另有明确说明。在本文档中提及或引用任何第三方内容不代表紫光展锐对第三方内容的认可、承诺或保证。

用户有义务结合自身情况，检查上述第三方内容的可用性。若需要第三方许可，应通过合法途径获取第三方许可，除非本文档另有明确说明。

# 紫光展锐（上海）科技有限公司



# 前言

## 概述

本文档介绍了基于 Uboot、Lk、Kernel4.4、Kernel4.14、Kernel5.4、Kernel5.15 的 GPIO、EIC 模块客制化开发过程，内容涵盖 Uboot、Kernel 的 GPIO、EIC 简介及相关应用、调试方法、注意事项等。

## 读者对象




本文档主要适用于需要在展锐平台上进行 GPIO、EIC 配置的驱动工程师。

## 缩略语

缩略语	英文全名	中文解释
DTS	Device Tree Source	设备树源码
EIC	External Interrupt Controller	外部中断控制器
GPIO	General Purpose Input Output	通用目的输入/输出端口
PMIC	Power Management Integrated Circuit	电源管理芯片
SP	System Processor	系统处理器

## 符号约定

在本文中可能出现下列符号，每种符号的说明如下。

符号	说明
 <b>说明</b>	用于突出重要或关键信息、补充信息和小窍门等。 “说明”不是安全警示信息，不涉及人身、设备及环境伤害。
 <b>注意</b>	用于突出容易出错的操作。 “注意”不是安全警示信息，不涉及人身、设备及环境伤害。
 <b>警告</b>	用于可能无法恢复的失误操作。 “警告”不是危险警示信息，不涉及人身及环境伤害。

## 变更信息

文档版本	发布日期	修改说明
V1.9	2023-06-09	增加 UWS9157 信息
V1.8	2023-03-27	增加 UWS6137 信息
V1.7	2022-11-14	增加 UIS7870SC、UIS7885 信息
V1.6	2022-11-04	<ul style="list-style-type: none"> <li>增加 Kernel4.4 内容。</li> <li>增加 LK 内容</li> <li>增加 UMS9621 信息</li> </ul>
V1.5	2022-08-02	文档增加支持 Kernel 5.15 的描述。
V1.4	2022-03-10	<ul style="list-style-type: none"> <li>增加 <a href="#">3.6 Kernel5.4 和 Kernel5.15 用户空间文件节点</a>。</li> <li>增加 <a href="#">4.3 GPIO&amp;EIC 中断唤醒源调试</a>。</li> </ul>
V1.3	2021-03-25	<ul style="list-style-type: none"> <li>修改文档名从《Kernel4.14 GPIO 客制化指导手册》成《GPIO 和 EIC 客制化指导手册》。</li> <li>全文增加 EIC 内容、增加 Uboot 内容。</li> <li>优化文档技术描述。</li> </ul>
V1.2	2020-07-29	优化文档描述，更新部分内容。
V1.1	2020-03-18	文档名修改，优化文档描述。
V1.0	2019-08-01	第一次正式发布。

## 关键字

GPIO、EIC、Kernel4.4、Kernel4.14、Kernel5.4、Kernel5.15、Uboot、Lk、DTS

# 目 录

1 概览.....	1
1.1 简介.....	1
1.2 各类 IP 休眠唤醒支持情况.....	1
2 Uboot 应用.....	3
2.1 Pinmap.....	3
2.2 LK GPIO.....	4
2.2.1 操作接口.....	4
2.2.2 操作示例.....	6
2.3 UBOOT GPIO.....	6
2.3.1 操作接口.....	6
2.3.2 操作示例.....	8
2.4 LK/UBOOT EIC.....	9
2.4.1 操作接口.....	9
2.4.2 EIC 编号.....	10
2.4.3 操作示例.....	11
3 Kernel 应用.....	12
3.1 驱动框架.....	12
3.2 DTS 配置.....	13
3.2.1 Kernel4.4.....	13
3.2.2 Kernel4.14 and later version.....	14
3.3 驱动模块应用.....	16
3.3.1 GPIO.....	16
3.3.2 EIC.....	17
3.3.3 EXTINT 的 EIC ID.....	19
3.3.4 操作示例.....	20
3.4 常用函数接口.....	21
3.4.1 基于描述符操作.....	21
3.4.2 基于编号操作.....	23
3.4.3 DTS 节点解析.....	25
3.4.4 动态开关中断.....	25
3.5 Kernel4.4 和 Kernel4.14 用户空间文件节点.....	26
3.5.1 查询 gpio base.....	26
3.5.2 创建的节点.....	26
3.5.3 节点操作.....	26
3.5.4 文件节点权限.....	27
3.6 Kernel5.4 和 Kernel5.15 用户空间文件节点.....	27

3.6.1 查询 gpiochip .....	28
3.6.2 节点操作 .....	28
3.6.3 文件节点权限 .....	30
4 调试方法 .....	31
4.1 文件节点 .....	31
4.2 读写寄存器 .....	31
4.2.1 lookat 操作命令 .....	31
4.2.2 GPIO 调试 .....	32
4.2.3 EIC 调试 .....	35
4.3 GPIO&EIC 中断唤醒源调试 .....	37
5 注意事项 .....	40
5.1 GPIO 边沿触发与电平触发的区别 .....	40
5.2 DeepSleep 时 GPIO 唤醒 AP 系统 .....	40
5.3 GPIO 使用风险提示 .....	41
6 参考文档 .....	43

Unisoc Confidential For waterworld

## 图目录

图 2-1 AON EIC Source List .....	11
图 3-1 驱动框架图 .....	12
图 4-1 GPIO 地址分布图 .....	33
图 5-1 电平触发示意图 .....	40

Unisoc Confidential For waterworld

## 表目录

表 1-1 各类 IP 休眠唤醒情况表.....	1
表 5-1 平台 GPIO 和 EIC 情况表.....	41

Unisoc Confidential For waterworld



# 1 概览

## 1.1 简介

GPIO（General Purpose Input Output，通用目的输入/输出端口）是一种灵活的数字信号，可以配置为输入或输出。当配置为输入模式时，它可以被编程来触发 Arm®中断。

EIC（External Interrupt Controller，外部中断控制器）与 GPIO 类似，但是只能配置为输入，触发中断使用，不能配置为输出。EIC 支持防抖配置，支持 4 种模式，分别是 debounce、latch、async、sync，其中前 3 种中断支持休眠状态下唤醒系统。每颗芯片供用户使用的一般是 Function 为 EXTINT0~EXTINT15 的 PIN，共 16 个。

PMIC EIC 是电源管理芯片上集成的 EIC 功能，但是只支持 debounce 模式。由于与上述 EIC debounce 功能完全一致，因此本文除特殊场景外，不单独介绍。

### 说明

- SC9863A 芯片对应的 GPIO IP 为 GPIO Plus，支持上述 GPIO 和 EIC 的功能，但是在 Uboot、Kernel 里操作接口及调用方法与 GPIO 完全一致，因此本文除在特定场景里会补充说明外，不单独介绍。
- 关于 GPIO\EIC IP 的详细介绍，请参考紫光展锐发布的《XXXX Device Specification》（XXXX 表示对应的芯片名称，后同）

## 1.2 各类 IP 休眠唤醒支持情况

表1-1 各类 IP 休眠唤醒情况表

IP	中断模式	触发类型	备注
GPIO	GPIO	Level	支持休眠唤醒，但是要求唤醒过程中电平稳定保持 120ms 以上或者直到中断回调函数被处理完后才能改变电平，否则 AP CPU 可能会出现异常。它不适用于按键、指纹、触屏等抖动严重的场景。
		Edge	不支持休眠唤醒。
EIC	DBNC	Level	RTC_1K，支持休眠唤醒，支持硬件防抖，电平检测时间可以配置 1ms~4s，要求电平保持时间为所配置时间 +2ms 才能检测到中断。该类型 IP 只支持 level 模式，但是 Kernel 里 EIC 驱动通过自动翻转支持 Edge 的中断方式，其他模块调用时可以直接配置为 Edge 模式。
	LATCH	Level	不需要时钟，支持休眠唤醒。

IP	中断模式	触发类型	备注
	ASYNC	Level/Edge	RTC_32K, 支持休眠唤醒, 要求电平保持时间 61 $\mu$ s 以上 (2 个 CLK)。
	SYNC	Level/Edge	不支持休眠唤醒。
GPIO Plus (仅 SC9863A)	DBNC	Level	RTC_1K, 支持休眠唤醒, 支持硬件防抖, 电平检测时间可以配置 1ms~4s, 要求电平保持时间为所配置时间 +2ms 才能检测到中断, 在 Kernel 中如果使用 Level 的中断触发类型, 并且配置了 debounce 时间, 会自动用 DBNC 模式, 如果没有配置 debounce 时间, 则自动为 LEVEL 模式 (同 GPIO 的 Level 触发类型)。
	EDGE	Edge	与 EIC ASYNC 一样。
	LATCH	Level	与 EIC LATCH 一样。
	LEVEL	Level	同 GPIO 的 Level 触发类型。

## 说明

支持休眠唤醒 AP, 除了 IP 本身要支持外, 还需要电平满足足够的持续时间, 否则功能将会失效。每颗芯片一般都支持 GPIO 和 EIC, 需要根据实际的要求选择。

Unisoc Confidential For waterworld

# 2 Uboot 应用

## 2.1 Pinmap

使用 GPIO 或 EIC 的前提是确保对应 Pin 的 Function 配置正确，否则会导致相应功能不生效。Pin List 可以参考《XXXX Device Specification》的 Pin 章节里。

Pinmap 在 Uboot 阶段就会初始化好，后续大部分不会再修改，配置路径：

“u-boot15\board\spreadtrum\[实际使用的 Board]\pinmap.c”

通过 BITS\_PIN\_AF 配置对应的 Function，如下以某颗芯片为例进行说明，实际应用时需要查询对应芯片平台的《XXXX Device Specification》

Cell Name	Pin Name	Function0	Type	Function1	Type	Function2	Type	Function3	Type
SPSCBC2_8X_W_HL	EXTINT10	EXTINT10	I			BAT_DET	I	GPIO79	I/O/T

{REG\_PIN\_EXTINT10, BITS\_PIN\_AF(3)}, //如果配置为 3，通过表格可以看到对应的是 GPIO79。

//如果配置为 0，即 Function0 为 EXTINT10，在 EIC Source List 中

//可以查询到对应的 EIC 控制器，说明可以作 EIC 中断使用。

### 说明

- 更详细的 Pinmap 配置方法可以参考紫光展锐发布的《XXXX Pinmap 配置指南》(XXXX 表示对应的芯片名称)，如该芯片平台没有该文档，请咨询 FAE。
- 一般情况下：
  - 将 Pin 的 Function 配置为 EXTINT0~EXTINT15，说明作为 EIC 功能使用。
  - 将 Pin 的 Function 配置为 GPIOx，说明作为 GPIO 使用，即：BITS\_PIN\_AF (3)，如果是安全 GPIO，则不是 3，具体以 PINList 为准。
- Kernel 中可以通过 pinctrl 动态切换 Pin function，但该方法使用较少。

## 2.2 LK GPIO

### 2.2.1 操作接口

#### 2.2.1.1 sprd\_gpio\_request

##### 【函数功能】

请求/使能 GPIO

##### 【函数原型】

```
int sprd_gpio_request(unsigned offset)
```

##### 【参数说明】

offset: PINList 的 GPIO ID, 如 GPIO79, 传入 79 即可。

##### 【返回值】

- 0 为成功
- 非 0 为失败

#### 2.2.1.2 sprd\_gpio\_direction\_output

##### 【函数功能】

GPIO 配置方向为输出模式, 并配置输出高或低电平。

##### 【函数原型】

```
int sprd_gpio_direction_output(unsigned offset, int value)
```

##### 【参数说明】

- offset: PINList 的 GPIO ID, 如 GPIO79, 传入 79 即可。
- Value: 配置输出电平, 0 为低电平, 1 为高电平。

##### 【返回值】

- 0 为成功
- 非 0 为失败

#### 2.2.1.3 sprd\_gpio\_set

##### 【函数功能】

GPIO 配置为输出电平高或低, 仅在输出模式有效。

**【函数原型】**

```
void sprd_gpio_set(unsigned offset, int value)
```

**【参数说明】**

- offset: PINList 的 GPIO ID, 如 GPIO79, 传入 79 即可。
- Value: 配置输出电平, 0 为低电平, 1 为高电平。

**【返回值】**

无

#### 2.2.1.4 sprd\_gpio\_direction\_input

**【函数功能】**

GPIO 配置为输入模式。

**【函数原型】**

```
int sprd_gpio_direction_input(unsigned offset)
```

**【参数说明】**

offset: PINList 的 GPIO ID, 如 GPIO79, 传入 79 即可。

**【返回值】**

- 0 为成功
- 非 0 为失败

#### 2.2.1.5 sprd\_gpio\_get

**【函数功能】**

获取外部电平是高或低。

**【函数原型】**

```
int sprd_gpio_get(unsigned offset)
```

**【参数说明】**

offset: PINList 的 GPIO ID, 如 GPIO79, 传入 79 即可。

**【返回值】**

- 负数: 读取失败
- 0: 低电平
- 正数: 高电平

## 2.2.2 操作示例

以下仅提供部分操作，并且省略掉返回值的判断。

- 将 GPIO79 拉高再拉低

```
//先使能GPIO，只需调用一次即可
sprd_gpio_request(79);
//配置为输出，并默认为高电平，只需调用一次即可
sprd_gpio_direction_output(79, 1);
//再拉低，在需要转变电平高低的地方调用
sprd_gpio_set(79, 0);
```

- 读取 GPIO79 所在的 PIN 是高电平还是低电平

```
//先使能GPIO，只需调用一次即可
sprd_gpio_request(79);
//配置为输入，用于读取外部电平，只需调用一次即可
sprd_gpio_direction_input(79);
//获取外部电平，在需要获取电平高低的地方调用
value = sprd_gpio_get(79);
if (value > 0)
    //说明是高电平，注意：不能用==1判断是高电平，要用 > 0
elseif (value == 0)
    //说明是低电平
else
    //说明操作失败
```

## 2.3 UBOOT GPIO

### 2.3.1 操作接口

#### 2.3.1.1 sprd\_gpio\_request

##### 【函数功能】

请求/使能 GPIO

##### 【函数原型】

```
int sprd_gpio_request(struct gpio_chip *chip, unsigned offset)
```

**【参数说明】**

- chip: GPIO CHIP 指针，一般传 NULL 即可。
- offset: PINList 的 GPIO ID，如 GPIO79，传入 79 即可。

**【返回值】**

- 0 为成功
- 非 0 为失败

## 2.3.1.2 sprd\_gpio\_direction\_output

**【函数功能】**

GPIO 配置为输出模式，并配置输出高或低电平。

**【函数原型】**

```
int sprd_gpio_direction_output(struct gpio_chip *chip, unsigned offset, int value)
```

**【参数说明】**

- chip: GPIO CHIP 指针，一般传 NULL 即可。
- offset: PINList 的 GPIO ID，如 GPIO79，传入 79 即可。
- Value: 配置输出电平，0 为低电平，1 为高电平。

**【返回值】**

- 0 为成功
- 非 0 为失败

## 2.3.1.3 sprd\_gpio\_set

**【函数功能】**

GPIO 配置为输出电平高或低，仅在输出模式有效。

**【函数原型】**

```
void sprd_gpio_set(struct gpio_chip *chip, unsigned offset, int value)
```

**【参数说明】**

- chip: GPIO CHIP 指针，一般传 NULL 即可。
- offset: PINList 的 GPIO ID，如 GPIO79，传入 79 即可。
- Value: 配置输出电平，0 为低电平，1 为高电平。

**【返回值】**

无

#### 2.3.1.4 sprd\_gpio\_direction\_input

##### 【函数功能】

GPIO 配置为输入模式。

##### 【函数原型】

```
int sprd_gpio_direction_input(struct gpio_chip *chip, unsigned offset)
```

##### 【参数说明】

- chip: GPIO CHIP 指针，一般传 NULL 即可。
- offset: PINList 的 GPIO ID，如 GPIO79，传入 79 即可。

##### 【返回值】

- 0 为成功
- 非 0 为失败

#### 2.3.1.5 sprd\_gpio\_get

##### 【函数功能】

获取外部电平是高或低。

##### 【函数原型】

```
int sprd_gpio_get(struct gpio_chip *chip, unsigned offset)
```

##### 【参数说明】

- chip: GPIO CHIP 指针，一般传 NULL 即可。
- offset: PINList 的 GPIO ID，如 GPIO79，传入 79 即可。

##### 【返回值】

- 负数: 读取失败
- 0: 低电平
- 正数: 高电平

### 2.3.2 操作示例

以下仅提供部分操作，并且省略掉返回值的判断。

- 将 GPIO79 拉高再拉低

```
//先使能GPIO，只需调用一次即可
sprd_gpio_request(NULL, 79);
//配置为输出，并默认为高电平，只需调用一次即可
```



```
sprd_gpio_direction_output(NULL, 79, 1);
```

```
//再拉低，在需要转变电平高低的地方调用
```

```
sprd_gpio_set(NULL, 79, 0);
```

- 读取 GPIO79 所在的 PIN 是高电平还是低电平

```
//先使能GPIO，只需调用一次即可
```

```
sprd_gpio_request(NULL, 79);
```

```
//配置为输入，用于读取外部电平，只需调用一次即可
```

```
sprd_gpio_direction_input(NULL, 79);
```

```
//获取外部电平，在需要获取电平高低的地方调用
```

```
value = sprd_gpio_get(NULL, 79);
```

```
if (value > 0 )
```

```
    //说明是高电平，注意：不能用==1判断是高电平，要用 > 0
```

```
elseif (value == 0)
```

```
    //说明是低电平
```

```
else
```

```
    //说明操作失败
```

## 2.4 LK/UBOOT EIC

AP 上的 EIC 和 PMIC（Power Management Integrated Circuit，电源管理芯片）上的 EIC 使用相同的函数接口，通过编号进行区分。

### 说明

使用 EIC 功能，需要 `#include <asm/arch/sprd_eic.h>`

### 2.4.1 操作接口

#### 2.4.1.1 sprd\_eic\_request

##### 【函数功能】

请求/使能 EIC

##### 【函数原型】

```
int sprd_eic_request(unsigned offset)
```

##### 【参数说明】

offset: 编号后的 EIC ID。

**【返回值】**

- 0 为成功
- 非 0 为失败

## 2.4.1.2 sprd\_eic\_get

**【函数功能】**

获取外部电平是高或低。

**【函数原型】**

```
int sprd_eic_get( unsigned offset)
```

**【参数说明】**

offset: 编号后的 EIC ID。

**【返回值】**

- 负数: 读取失败
- 0: 低电平
- 正数: 高电平

## 2.4.2 EIC 编号

EIC 编号定义的头文件如下:

“\u-boot\15\arch\arm\include\asm\arch-XXXX\sprd\_eic.h”

对应的内容类似如下: (以某颗芯片为例, 不同芯片可能会有所不同)

```
/* pmic eic */
#define SPRD_ADIE_EIC_START 160
#define SPRD_ADIE_EIC_END   167
/* d-die eic */
#define SPRD_DDIE_EIC_START    0
#define SPRD_DDIE_EIC_END      7
#define SPRD_DDIE_EIC1_START   8
#define SPRD_DDIE_EIC1_END     15
```

d-die eic 即 AON EIC, 也就是 AP 侧的 EIC, 上述 SPRD\_DDIE\_EIC 对应 EXTINT0~EXTINT7 所在的 EIC Controller, SPRD\_DDIE\_EIC1 对应 EXTINT8~EXTINT15 所在的 EIC Controller。根据《XXXX Device Specification》中的 EIC->Application Notes 的 AON EIC Source List, 可以获取 EXTINT0~EXTINT15 在 Uboot 使用 EIC 的编号 (2.4.1 操作接口内的输入参数 offset), 如下以 UMS512 为例, 其他芯片可能会有所不同。

图2-1 AON EIC Source List

EIC_EXT	Lat/Dbnc/Async/Sync Input	Num	Offset
EIC_EXT2	PAD: EXTINT0	0	0
	PAD: EXTINT1	1	1
	PAD: EXTINT2	2	2
	PAD: EXTINT3	3	3
	PAD: EXTINT4	4	4
	PAD: EXTINT5	5	5
	PAD: EXTINT6	6	6
	PAD: EXTINT7	7	7
EIC_EXT3	PAD: EXTINT8	0	8
	PAD: EXTINT9	1	9
	PAD: EXTINT10	2	10
	PAD: EXTINT11	3	11
	PAD: EXTINT12	4	12
	PAD: EXTINT13	5	13
	PAD: EXTINT14	6	14
	PAD: EXTINT15	7	15

### 2.4.3 操作示例

EIC 只支持输入，因此无需配置 direction。

//先使能EIC，只需调用一次即可

```
sprd_eic_request(EIC_KEY2_7S_RST_EXT_RSTN_ACTIVE);
```

//获取外部电平，在需要获取电平高低的地方调用

```
value = sprd_eic_get(EIC_KEY2_7S_RST_EXT_RSTN_ACTIVE);
```

```
if (value > 0)
```

```
    //说明是高电平，注意：不能用==1判断是高电平，要用 > 0
```

```
elseif (value == 0)
```

```
    //说明是低电平
```

```
else
```

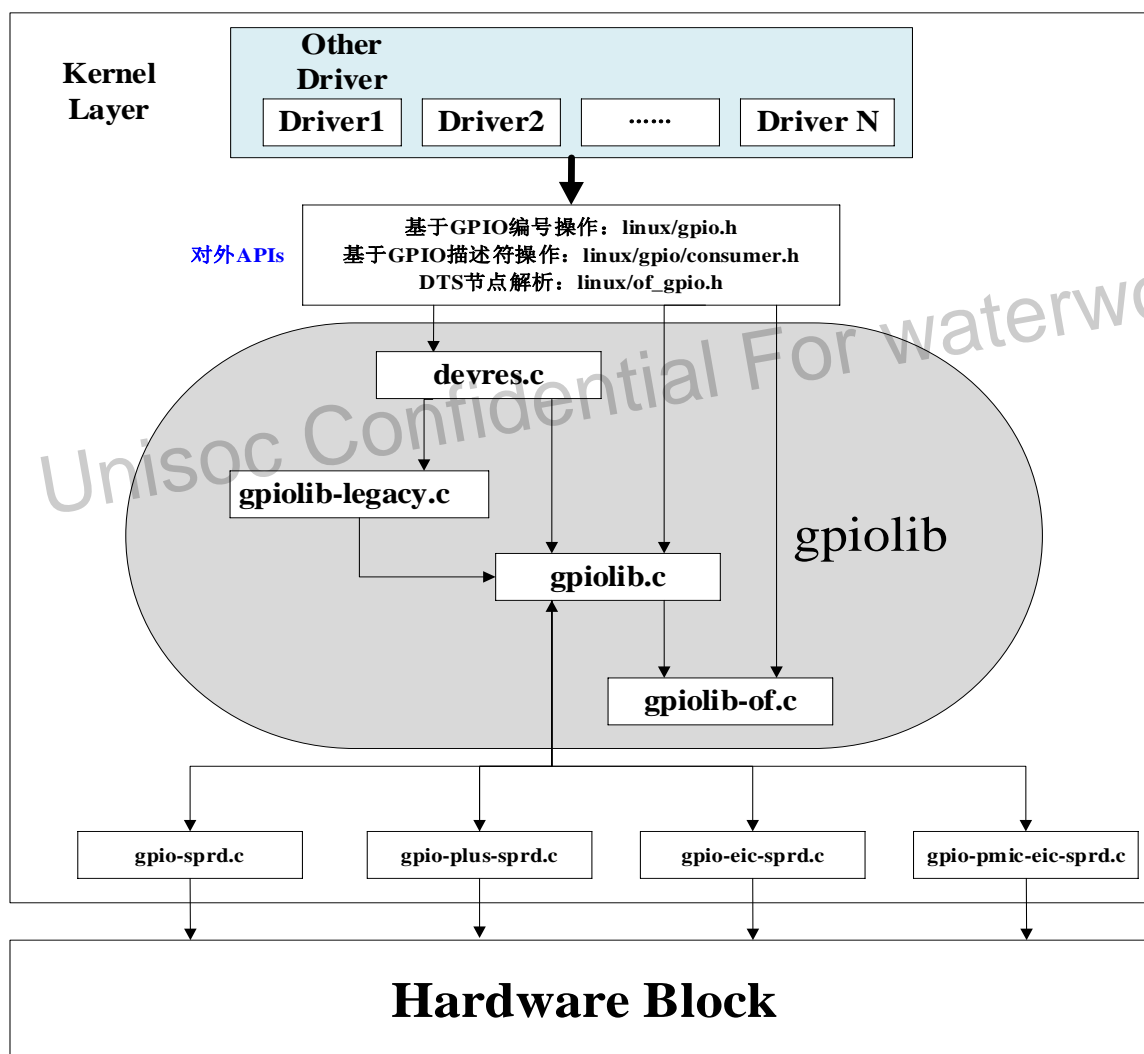
```
    //说明操作失败
```

# 3 Kernel 应用

## 3.1 驱动框架

不同于 Uboot，Kernel 中将 GPIO、EIC 相关的驱动挂在 GPIO Device 中，因此不管是 GPIO 还是 EIC，在 Kernel 里所使用的操作函数是完全一样的，只是通过 DTS（Device Tree Source，设备树源码）配置区分不同的 IP。

图3-1 驱动框架图



## 3.2 DTS 配置

### 3.2.1 Kernel4.4

以 SC9863A 的配置为例，定义的文件在 sharkl3.dtsi，可以通过如下文件查找节点所在文件。

“\arch\arm64\boot\dts\sprd\sharkl3.dtsi”

```
#include "sharkl3.dtsi"

ap_gpio: gpio-controller@402c0000 {
    compatible = "sprd,gpio-plus";
    reg = <0x0 0x402c0000 0x0 0x1000>;
    gpio-controller;
    #gpio-cells = <2>;
    sprd,gpiobase = <0>; //编号base id
    sprd,ngpios = <256>;
    interrupt-controller;
    #interrupt-cells = <2>;
    interrupts = <GIC_SPI 130 IRQ_TYPE_LEVEL_HIGH>;
};
```

```
ap_eic: gpio-controller@40210000 {
    compatible = "sprd,ap-eic";
    reg = <0x0 0x40210000 0x0 0x80>,
        <0x0 0x40370000 0x0 0x80>;
    gpio-controller;
    #gpio-cells = <2>;
    sprd,gpiobase = <288>; //编号base id
    sprd,ngpios = <32>;
    interrupt-controller;
    #interrupt-cells = <2>;
    interrupts = <GIC_SPI 37 IRQ_TYPE_LEVEL_HIGH>;
};
```

```
ap_eic_async: gpio-controller@402100a0 {
    compatible = "sprd,ap-eic-async";
    reg = <0x0 0x402100a0 0x0 0x40>,
        <0x0 0x403700a0 0x0 0x40>;
    gpio-controller;
```

```
#gpio-cells = <2>;
sprd,gpiobase = <336>; //编号base id
sprd,ngpios = <32>;
interrupt-controller;
#interrupt-cells = <2>;
interrupts = <GIC_SPI 37 IRQ_TYPE_LEVEL_HIGH>;
};
```

## 说明

- 驱动模块调用 GPIO 还是 EIC 是通过 ap\_gpio、ap\_eic、ap\_eic\_async 来标识的。

### 3.2.2 Kernel4.14 and later version

GPIO、EIC 驱动所使用的 DTS 节点一般定义在以芯片别名命名的 dtsi 里。为方便介绍，以 UMS512 的配置为例，定义的文件在 sharkl5Pro.dtsi，可以通过如下文件查找节点所在文件。

“\arch\arm64\boot\dts\sprd\ums512.dtsi”

```
#include "sharkl5Pro.dtsi" //GPIO、EIC节点所在文件
```

```
ap_gpio: gpio@32070000 {
    compatible = "sprd,sharkl5Pro-gpio", "sprd,sharkl5-gpio";
    reg = <0 0x32070000 0 0x10000>; //0x32070000为GPIO控制寄存器的基地址
    gpio-controller;
    #gpio-cells = <2>;
    interrupt-controller;
    #interrupt-cells = <2>;
    interrupts = <GIC_SPI 61 IRQ_TYPE_LEVEL_HIGH>;
};
```

```
eic_debounce: gpio@32000000 {
    compatible = "sprd,sharkl5pro-eic-debounce",
                "sprd,sharkl5-eic-debounce";
    reg = <0 0x32020000 0 0x80>, //AON EIC2
          <0 0x32030000 0 0x80>, //AON EIC3
          <0 0x32230000 0 0x80>, //AON EIC4
          <0 0x32270000 0 0x80>; //AON EIC5
    gpio-controller;
    #gpio-cells = <2>;
    interrupt-controller;
    #interrupt-cells = <2>;
    interrupts = <GIC_SPI 72 IRQ_TYPE_LEVEL_HIGH>;
```

```
};
eic_latch: gpio@32000080 {
    compatible = "sprd,sharkl5pro-eic-latch",
        "sprd,sharkl5-eic-latch";
    reg = <0 0x32020080 0 0x20>, //AON EIC2 latch 模式偏移0x80
        <0 0x32030080 0 0x20>, //AON EIC3
        <0 0x32230080 0 0x20>, //AON EIC4
        <0 0x32270080 0 0x20>; //AON EIC5
    gpio-controller;
    #gpio-cells = <2>;
    interrupt-controller;
    #interrupt-cells = <2>;
    interrupts = <GIC_SPI 72 IRQ_TYPE_LEVEL_HIGH>;
};
eic_async: gpio@320000a0 {
    compatible = "sprd,sharkl5pro-eic-async",
        "sprd,sharkl5-eic-async";
    reg = <0 0x320200a0 0 0x20>, //AON EIC2 async 模式偏移0xa0
        <0 0x320300a0 0 0x20>,
        <0 0x322300a0 0 0x20>,
        <0 0x322700a0 0 0x20>;
    gpio-controller;
    #gpio-cells = <2>;
    interrupt-controller;
    #interrupt-cells = <2>;
    interrupts = <GIC_SPI 72 IRQ_TYPE_LEVEL_HIGH>;
};
eic_sync: gpio@320000c0 {
    compatible = "sprd,sharkl5pro-eic-sync",
        "sprd,sharkl5-eic-sync";
    reg = <0 0x320200c0 0 0x20>, //AON EIC2 sync 模式偏移0xc0
        <0 0x320300c0 0 0x20>,
        <0 0x322300c0 0 0x20>,
        <0 0x322700c0 0 0x20>;
    gpio-controller;
    #gpio-cells = <2>;
    interrupt-controller;
```

```
#interrupt-cells = <2>;
interrupts = <GIC_SPI 72 IRQ_TYPE_LEVEL_HIGH>;
};
```

### 说明

- DTS 节点里定义的 EIC 地址对应的 EIC 控制器，可以参考《XXXX Device Specification》中的 EIC->Control Registers。
- 驱动模块调用 GPIO 还是 EIC 是通过 ap\_gpio、eic\_debounce、eic\_latch、eic\_async、eic\_sync 来标识的。

## 3.3 驱动模块应用

驱动模块如果要使用 GPIO 或 EIC，定义的名称以“-gpio”或“-gpios”结尾，使用的格式为：

```
foo-gpio = <&IP_TYPE ID ACTIVE_TYPE>
foo-gpios = <&IP_TYPE ID ACTIVE_TYPE>
```

也可以直接“gpio”或“gpios”，格式为：

```
gpio = <&IP_TYPE ID ACTIVE_TYPE>
gpios = <&IP_TYPE ID ACTIVE_TYPE>
```

### 说明

- IP\_TYPE 的值包括：ap\_gpio、eic\_debounce、eic\_latch、eic\_async、eic\_sync、pmic\_eic
- ID:
  - 对于 GPIO，为实际的 GPIO ID，如：GPIO79，就直接配置为 79 即可。
  - 对于 EIC，需要根据已定义的 EIC 控制器的对应关系进行映射，具体方法参考 [3.3.2 EIC](#)。
- ACTIVE\_TYPE：包括 GPIO\_ACTIVE\_HIGH（高有效）和 GPIO\_ACTIVE\_LOW（低有效）。具体是高有效还是低有效应以硬件设计为准。

### 3.3.1 GPIO

例：需要从 DTS 里获取 GPIO79 的配置，并且配置为输入，参考代码如下：

- DTS 里增加代码

```
foo_device {
    compatible = "acme,foo";
    .....

    foo-gpio = <&ap_gpio 79 GPIO_ACTIVE_HIGH>; //79为实际的GPIO ID
    .....
};
```

- 驱动代码里增加代码

```
data->gpiod = devm_gpiod_get(&pdev->dev, "foo", GPIOD_IN); //从dts获取gpio配置
//并且默认为input
```



```
if (IS_ERR(data->gpiod)) {
    dev_err(&pdev->dev, "Failed to get foo GPIO\n");
    return PTR_ERR(data->gpiod);
}
```

### 3.3.2 EIC

EIC 的 ID 需要根据 DTS 的定义与 EIC Source List (《XXXX Device Specification》中的 EIC->Application Notes 的 AON EIC Source List, 后同) 进行对应。下面以 UMS512 的配置为例, 对应关系如下:

EIC_EXT	Lat/Dbnc/Async/Sync Input	Num	Kernel中的编号
EIC_EXT0	PAD: UART0_RXD	0	未定义
	PAD: UART0_CTSN	1	未定义
	PAD: UART1_RXD	2	未定义
	PAD: UART2_RXD	3	未定义
	PAD: UART3_RXD	4	未定义
	PAD: UART4_RXD	5	未定义
	PAD: UART4_CTSN	6	未定义
	PAD: UART5_RXD	7	未定义
EIC_EXT1	PAD: UART6_RXD	0	未定义
	PAD: USB_VP   USB_VM	1	未定义
	1'b0	2	未定义
	1'b0	3	未定义
	1'b0	4	未定义
	1'b0	5	未定义
	1'b0	6	未定义
	1'b0	7	未定义
EIC_EXT2	PAD: EXTINT0	0	0
	PAD: EXTINT1	1	1
	PAD: EXTINT2	2	2
	PAD: EXTINT3	3	3
	PAD: EXTINT4	4	4
	PAD: EXTINT5	5	5
	PAD: EXTINT6	6	6
	PAD: EXTINT7	7	7
EIC_EXT3	PAD: EXTINT8	0	8
	PAD: EXTINT9	1	9
	PAD: EXTINT10	2	10
	PAD: EXTINT11	3	11
	PAD: EXTINT12	4	12
	PAD: EXTINT13	5	13
	PAD: EXTINT14	6	14
	PAD: EXTINT15	7	15

```
eic_debounce: gpio@32000000 {
    compatible = "sprd,sharkl5pro-eic-debounce",
        "sprd,sharkl5-eic-debounce";
    reg = <0 0x32020000 0 0x80>, //AON EIC2
        <0 0x32030000 0 0x80>, //AON EIC3
        <0 0x32230000 0 0x80>, //AON EIC4
        <0 0x32270000 0 0x80>, //AON EIC5

    gpio-controller;
    #gpio-cells = <2>;
    interrupt-controller;
    #interrupt-cells = <2>;
    interrupts = <GIC_SPI 72 IRQ_TYPE_LEVEL_HIGH>;
};
```

通过 DTS 里的配置可以看出, Kernel 只支持 AON EIC2~ AON EIC5, 所以 Kernel 中用到的 EIC ID 就从 EIC\_EXT2 开始, 从 0 开始依次往下开始编号。假设要使用 EXTINT10, 并且用 debounce 模式, 则 DTS 引用方法如下:

```
foo_device {
    compatible = "acme,foo";
    .....
};
```

```
foo-gpio = <&eic_debounce 10 GPIO_ACTIVE_HIGH>; //同一PIN的其他模式: latch/async/sync
//使用的ID均相同
.....
};
data->gpiod = devm_gpiod_get(&pdev->dev, "foo", GPIOD_IN); //从dts获取gpio配置
//EIC只能配置输入
if (IS_ERR(data->gpiod)) {
dev_err(&pdev->dev, "Failed to get foo EIC\n");
return PTR_ERR(data->gpiod);
}
```

假设 sharkl5Pro.dtsi 把 AON\_EIC0、AON\_EIC1 也加上，则对应关系变更为如下：

```
eic_debounce: gpio@32000000 {
    compatible = "sprd,sharkl5pro-eic-debounce",
        "sprd,sharkl5-eic-debounce";
    reg = <0 0x32000000 0 0x80>, //AON EIC0
        <0 0x32010000 0 0x80>, //AON EIC1
        <0 0x32020000 0 0x80>, //AON EIC2
        <0 0x32030000 0 0x80>, //AON EIC3
        <0 0x32230000 0 0x80>, //AON EIC4
        <0 0x32270000 0 0x80>; //AON EIC5
    gpio-controller;
    #gpio-cells = <2>;
    interrupt-controller;
    #interrupt-cells = <2>;
    interrupts = <GIC_SPI 72 IRQ_TYPE_LEVEL_HIGH>;
};
```

EIC_EXT	Lat/Dbnc/Async/Sync Input	Num	Kernel中的编号
EIC_EXT0	PAD: UART0_RXD	0	0
	PAD: UART0_CTSN	1	1
	PAD: UART1_RXD	2	2
	PAD: UART2_RXD	3	3
	PAD: UART3_RXD	4	4
	PAD: UART4_RXD	5	5
	PAD: UART4_CTSN	6	6
EIC_EXT1	PAD: UART5_RXD	7	7
	PAD: UART6_RXD	0	8
	PAD: USB_VP USB_VM	1	9
	1'b0	2	10
	1'b0	3	11
	1'b0	4	12
	1'b0	5	13
EIC_EXT2	1'b0	6	14
	1'b0	7	15
	PAD: EXTINT0	0	16
	PAD: EXTINT1	1	17
	PAD: EXTINT2	2	18
	PAD: EXTINT3	3	19
	PAD: EXTINT4	4	20
EIC_EXT3	PAD: EXTINT5	5	21
	PAD: EXTINT6	6	22
	PAD: EXTINT7	7	23
	PAD: EXTINT8	0	24
	PAD: EXTINT9	1	25
	PAD: EXTINT10	2	26
	PAD: EXTINT11	3	27
	PAD: EXTINT12	4	28
	PAD: EXTINT13	5	29
	PAD: EXTINT14	6	30
	PAD: EXTINT15	7	31

如果此时要使用 EXINT10 的 EIC 功能，ID 的值会变成 26，如下：

```
foo-gpio = <&eic_debounce 26 GPIO_ACTIVE_HIGH>
```

## 说明

- 如果只是用 EXTINT 的 PIN 做 EIC 中断，可以通过 [3.3.3 EXTINT 的 EIC ID](#) 的内容快速获取 EIC ID

- 如果不是 EXTINT，具体地址需要计算得到，则需要参考 EIC Source List。若《XXXX Device Specification》不包含 AON EIC Source List，请联系 CPM 获取。

### 3.3.3 EXTINT 的 EIC ID

开放给客户端使用的 EIC 中断一般是 Function 包含 EXTINT0~EXTINT15 的 PIN，通过 EXTINTxx 后面的数字也可以快速获取要在 Kernel DTS 定义的 ID，根据芯片不同定义如下：

芯片	Kernel DTS ID	注意事项
SC7731E	xx	EXTINT0~EXTINT 7 只支持 debounce 模式 EXTINT8~EXTINT15 均支持 4 种模式
SC9832E	xx	EXTINT0~EXTINT 7 只支持 debounce 模式 EXTINT8~EXTINT15 均支持 4 种模式
SC9863A	xx	EXTINT0~EXTINT 7 只支持 debounce 模式 EXTINT8~EXTINT15 均支持 4 种模式
UMS312	xx	EXTINT0~EXTINT15 均支持 4 种模式
UMS512/UMS512T	xx	EXTINT0~EXTINT15 均支持 4 种模式
UDX710	xx	只支持 EXTINT0~EXTINT4，均支持 4 种模式
UDS710	16+xx	EXTINT0~EXTINT15 均支持 4 种模式
UMS9230	16+xx	EXTINT0~EXTINT15 均支持 4 种模式
UMS9620	16+xx	EXTINT0~EXTINT15 均支持 4 种模式
UMS9621	16+xx	EXTINT0~EXTINT15 均支持 4 种模式
UIS7870SC	16+xx	EXTINT0~EXTINT15 均支持 4 种模式
UIS7885	16+xx	EXTINT0~EXTINT15 均支持 4 种模式
UWS6137	xx	EXTINT0~EXTINT 7 只支持 debounce 模式 EXTINT8~EXTINT15 均支持 4 种模式
UMS9157	16+xx	EXTINT0~EXTINT15 均支持 4 种模式

#### 说明

上表中的 xx 表示 EXTINT 后面的数字。

上述表格是针对配置 Function 为 EXTINTxx 时的 EIC ID 获取方法，同一根 PIN 的不同 Function 可能会对对应到不同的 EIC 控制器。

以 UMS512 的 EXTINT9 为例，如果配置为 Fucntion0（EXTINT9）则 EIC ID 为 9。如果配置为 Fucntion2（BUA\_TF\_DET，用于 T 卡热插拨），则 EIC ID 为 19。EXTINT0~EXTINT15 之外的配置一般是有特殊用途，建议不要自行修改，如有疑问，请联系紫光展锐 FAE。

EIC_EXT	Lat/Dbnc/Async/Sync Input	Num	Kernel中的编号
EIC_EXT2	PAD: EXTINT0	0	0
	PAD: EXTINT1	1	1
	PAD: EXTINT2	2	2
	PAD: EXTINT3	3	3
	PAD: EXTINT4	4	4
	PAD: EXTINT5	5	5
	PAD: EXTINT6	6	6
	PAD: EXTINT7	7	7
EIC_EXT3	PAD: EXTINT8	0	8
	PAD: EXTINT9	1	9
	PAD: EXTINT10	2	10
	PAD: EXTINT11	3	11
	PAD: EXTINT12	4	12
	PAD: EXTINT13	5	13
	PAD: EXTINT14	6	14
	PAD: EXTINT15	7	15
EIC_EXT4	PAD: SIM0_BUA_DET_PUBCP	0	16
	PAD: SIM1_BUA_DET_PUBCP	1	17
	PAD: SIM0_BUA_DET_AP	2	18
	PAD: SDIO0_BUA_DET_AP	3	19
	PAD: SDIO1_BUA_DET_AP	4	20

Name	Function0	Function1	Function2	Function3
EXTINT9	EXTINT9	KEYOUT3	BUA_TF_DET	GPIO78

### 3.3.4 操作示例

以 UMS512 的 EXTINT10 为例，将该 PIN 作为 EIC 中断使用，并且是高有效，中断方式为上升沿触发中断。

- 检查对应的 Pinmap 是否 Function 为 EXINT10，每个项目都要通过 Pin List 确认，不一定是 0  
{REG\_PIN\_EXTINT10, BITS\_PIN\_AF(0)}

- 在驱动 DTS 节点里增加 GPIO 节点，如：foo-gpio

```
foo_device {
    compatible = "acme,foo";
    .....

    foo-gpio = <&eic_debounce 10 GPIO_ACTIVE_HIGH>;    //根据上一节快速查表
//EXTINT10的编号为10
    .....
};
```

- 驱动代码里解析 DTS 并申请中断

```
data->gpiod = devm_gpiod_get(&pdev->dev, "foo", GPIOD_IN); //解析DTS，并申请GPIO
if (IS_ERR(data->gpiod)) {
    dev_err(&pdev->dev, "Failed to get foo GPIO\n");
    return PTR_ERR(data->gpiod);
}
```

```
ret = gpiod_get_raw_value(data->gpiod); //获取外部电平：0为低，1为高，负值说明读取失败
if (ret < 0) {
    dev_err(&pdev->dev, "Failed to get gpio state\n");
    return ret;
}

irq = gpiod_to_irq(data->gpiod); //根据gpio描述符获取虚拟中断号
if (irq < 0) {
    dev_err(&pdev->dev, "Failed to translate GPIO to IRQ\n");
    return irq;
}

ret = devm_request_threaded_irq(&pdev->dev, irq, NULL, //注册中断信息
                                foo_callback, //中断下半部回调函数，如果使用上半部，则为前一个参数
                                IRQF_ONESHOT | IRQF_TRIGGER_RISING,
                                pdev->name, data);

enable_irq_wake(irq); //启用唤醒，只有在相应IP对应的模式支持才能生效
```

## 3.4 常用函数接口

Kernel 提供给外部的函数接口可以分为 3 类：

- 基于 GPIO 描述符操作：linux/gpio/consumer.h
- 基于 GPIO 编号操作：linux/gpio.h
- DTS 节点解析：linux/of\_gpio.h

推荐尽量使用基于 GPIO 描述符的操作函数，该类操作函数里已经包含相应的 DTS 解析函数，可以实现 GPIO 的所有操作。GPIO、EIC 驱动均是挂在 GPIO Device 下，所以相关操作函数完全一样，因此本节所述的 GPIO 函数，实际对 EIC 也是一样的。

以下会针对这 3 类操作中比较常用的函数接口进行说明。本文主要介绍常见的应用场景，关于具体的参数说明可以参考相应的函数原型。

### 3.4.1 基于描述符操作

需要包含如下头文件：

“#include <linux/gpio/consumer.h>”

1. 解析 DTS 并使能 GPIO，同时配置为输入或输出（支持输出高或低配置）。

```
// 只定义一个GPIO时
struct gpio_desc * devm_gpiod_get(struct device *dev,
                                const char *con_id,
                                enum gpiod_flags flags);
```

//定义多个GPIO时，按索引获取GPIO信息

```
struct gpio_desc * devm_gpiod_get_index(struct device *dev,
                                         const char *con_id,
                                         unsigned int idx,
                                         enum gpiod_flags flags);
```

举例：

```
foo_device {
    compatible = "acme,foo";
    ...
    led-gpios = <&ap_gpio 15 GPIO_ACTIVE_HIGH>, /* red */
               <&ap_gpio 16 GPIO_ACTIVE_HIGH>, /* green */
               <&ap_gpio 17 GPIO_ACTIVE_HIGH>; /* blue */
    power-gpio = <&ap_gpio 1 GPIO_ACTIVE_LOW>;
};
```

获取方法：

```
struct gpio_desc *red, *green, *blue, *power;
red = devm_gpiod_get_index (dev, "led", 0, GPIOD_OUT_HIGH);
green = devm_gpiod_get_index (dev, "led", 1, GPIOD_OUT_HIGH);
blue = devm_gpiod_get_index (dev, "led", 2, GPIOD_OUT_HIGH);
power = devm_gpiod_get(dev, "power", GPIOD_IN); // power-gpio只定义1个
```

### 注意

gpiod\_get()和 gpiod\_get\_index()与上述使用方法完全一致，不再具体介绍。devm\_\*的接口会进行统一的资源管理，在操作失败时会自动释放

## 2. 配置为输入模式。

//配置为输入模式，如果使用前面解析函数已经通过flag配置为输入的，则无需再调用

```
int gpiod_direction_input(struct gpio_desc *desc);
```

## 3. 配置为输出模式和默认电平。

//如果DTS配置的是高有效，当value为1时配置为输出高电平，当value为0时配置为输出低电平

//如果DTS配置的是低有效，当value为1时配置为输出低电平，当value为0时配置为输出高电平

```
int gpiod_direction_output(struct gpio_desc *desc, int value);
```

//不管DTS配置的是高有效还是低有效，value为1时配置为高电平，value为0时配置为低电平

```
int gpiod_direction_output_raw(struct gpio_desc *desc, int value);
```

## 4. 输入模式时，获取外部电平是高还是低。

//如果DTS配置的是高有效，则高电平返回1，低电平返回0

//如果DTS配置的是低有效，则高电平返回0，低电平返回1

```
int gpiod_get_value(const struct gpio_desc *desc);
```

```
//不管DTS配置的是高有效还是低有效，高电平返回1，低电平返回0
```

```
int gpiod_get_raw_value(const struct gpio_desc *desc);
```

5. 输出模式时，设置外部电平是高还是低。

```
//如果DTS配置的是高有效，当value为1时配置为输出高电平，当value为0时配置为输出低电平
```

```
//如果DTS配置的是低有效，当value为1时配置为输出低电平，当value为0时配置为输出高电平
```

```
void gpiod_set_value(struct gpio_desc *desc, int value);
```

```
//不管DTS配置的是高有效还是低有效，value为1时配置为高电平，value为0时配置为低电平
```

```
void gpiod_set_raw_value(struct gpio_desc *desc, int value);
```

6. 硬件去抖配置。

```
// debounce的去抖时间，单位为us，如：配置为10ms，则debounce的值为 10 * 1000
```

```
static inline int gpiod_set_debounce(struct gpio_desc *desc, unsigned debounce)
```

### 注意

硬件去抖配置仅 EIC Debounce 模式和 SC9863 的 GPIO Plus 支持，其他情况配置不生效。

7. GPIO 编号转化为 GPIO 描述符。

```
//由于多个IP挂在GPIO Device下，所以会对GPIO、EIC进行重新编号，如下gpio值是编号后的ID
```

```
struct gpio_desc *gpio_to_desc(unsigned gpio);
```

8. GPIO 描述符转化为 GPIO 编号。

```
//返回值为编号后的ID，并不是DTS里配置的ID
```

```
int desc_to_gpio(const struct gpio_desc *desc);
```

9. 在/sys/class/gpio 路径下创建文件该 GPIO 的节点，供用户空间操作。

```
int gpiod_export(struct gpio_desc *desc, bool direction_may_change)
```

## 3.4.2 基于编号操作

需要包含如下头文件：

```
“#include <linux/gpio.h>”
```

推荐使用基于描述符的操作接口，不建议使用基于编号的操作函数，编号的获取方法请参考本节后续内容。这类函数均有输入参数 unsigned gpio，其值对应的并不是直接的 ID，而是编号后的值。

1. 申请 GPIO。

```
int devm_gpio_request(struct device *dev, unsigned gpio, const char *label);
```

```
//申请GPIO，同时可以配置flags，flags对应linux/gpio.h中的GPIOF_*相关的宏
```

```
int devm_gpio_request_one(struct device *dev, unsigned gpio,
```

```
unsigned long flags, const char *label);
```

```
int gpio_request(unsigned gpio, const char *label)
```

```
//申请GPIO，同时可以配置flags，flags对应linux/gpio.h中的GPIOF_*相关的宏
```

```
int gpio_request_one(unsigned gpio,
```

```
unsigned long flags, const char *label)
```



- 配置为输入或输出模式：如果已经用上述带有 flags 的函数配置了输入或输出模式，则不需要再调用。

```
//配置为输入模式
int gpio_direction_input(unsigned gpio)
//配置为输出模式
int gpio_direction_output(unsigned gpio, int value)
```

- 获取外部电平是高还是低。

```
//如果配置了GPIOF_ACTIVE_LOW，则高电平返回0，低电平返回1
//如果未配置则高电平返回1，低电平返回0
int gpio_get_value(unsigned gpio)
```

- 设置输出电平是高还是低。

```
//如果配置了GPIOF_ACTIVE_LOW，value为1则输出低电平，value为0则输出高电平
//如果未配置，value为1则输出高电平，value为0则输出低电平
void gpio_set_value(unsigned gpio, int value)
```

- Kernel4.4 版本编号的获取方法，即上述函数 unsigned gpio 的传入值。

dts 里直接定义 base id，可通过 3.2.1 节 Kernel4.4 查询，在 Userdebug 版本，adb shell 后可以通过 **cat /sys/kernel/debug/gpio** 命令，查看当前 Kernel 已经申请的 GPIO 情况，以及对应的编号。下面以 SC9863A 为例进行说明：

```
s9863a1h10:/ # cat /sys/kernel/debug/gpio
GPIOs 0-255, platform/402c0000.gpio-controller, sprd-gpio-plus: //GPIO, base id 为 0
gpio-76 ( |flash-en-gpios ) in lo
gpio-88 ( |flash-torch-en-gpios) in lo
gpio-89 ( |flash-chip-en-gpios ) out hi
gpio-124 ( |Volume Down Key ) in hi //说明 DTS 的配置为<&ap_gpio 124 .....>
gpio-130 ( |microarray_eint ) in lo
GPIOs 288-319, platform/40210000.gpio-controller, sprd-ap-eic: //EIC_DBNC, base id 为 288
GPIOs 320-335, platform/41800000.spi:pmic@0:gpio-controller@280, sprd-pmic-eic: //PMIC, base id 为 320
gpio-320 ( |musb vbus detect ) in hi
GPIOs 336-367, platform/402100a0.gpio-controller, sprd-ap-eic-async: //EIC_ASYNC, base id 为 336
gpio-345 ( |cd ) in hi //说明 DTS 的配置为<&ap_eic_async 9 .....>
```

## 说明

通过上述内容，EIC 只支持 DBNC 和 ASYNC 模式，GPIOs 0-255（共 256 个编号），说明是普通的 GPIO，只有普通的 GPIO 才会占用 256 个编号，其他都会比较少。

- Kernel4.14 及以上版本编号的获取方法，即上述函数 unsigned gpio 的传入值。

因为 GPIO、EIC、PMIC EIC 驱动均是挂在 GPIO Device 下，所以需要对各种 IP 进行重新编号才能正确地操作。每个 IP 均会自动分配一个 base id，实际操作的编号=base id + ID（ID 为 DTS 上配置的值）。对于同一芯片相关 base id 是相同的，不同的芯片 base id 会有所不同。



在 Userdebug 版本，adb shell 后可以通过 `cat /sys/kernel/debug/gpio` 命令，查看当前 Kernel 已经申请的 GPIO 情况，以及对应的编号。下面以 UMS512\_1h10 为例进行说明：

```
ums512_1h10:/ # cat /sys/kernel/debug/gpio
gpiochip5: GPIOs 112-127, parent: platform/sc27xx-eic, sc27xx-eic: //PMIC EIC, base id 为 112
gpio-113 (      |Power Key      ) in  hi IRQ //说明 DTS 的配置为<&pmic_eic 1 .....>
gpio-116 (      |Volume Up Key ) in  lo IRQ //说明 DTS 的配置为<&pmic_eic 4 .....>
gpiochip4: GPIOs 128-383, parent: platform/32070000.gpio, 32070000.gpio: //base id 为 128
gpio-252 (      |Volume Down Key ) in  hi IRQ //说明 DTS 的配置为<&ap_gpio 124 .....>
gpiochip3: GPIOs 384-415, parent: platform/320200c0.gpio, eic-sync: //base id 为 384
gpiochip2: GPIOs 416-447, parent: platform/320200a0.gpio, eic-async: //base id 为 416
gpio-429 (      |microarray_eint ) in  lo      //说明 DTS 的配置为<&eic_async 13 .....>
gpiochip1: GPIOs 448-479, parent: platform/32020080.gpio, eic-latch: //base id 为 448
gpiochip0: GPIOs 480-511, parent: platform/32020000.gpio, eic-debounce: //base id 为 480
```

### 说明

- 通过上述内容，EIC 的 4 种模式可以直接看出来，而 gpiochip4: GPIOs 128-383（共 256 个编号），说明是普通的 GPIO，只有普通的 GPIO 才会占用 256 个编号，其他都会比较少。
- 从上面显示的内容，也可以得到一些 GPIO 的申请信息，说明如：gpio-252 ( |Volume Down Key [申请 GPIO 时传入的名字] ) in[作为输入] hi[当前为高电平] IRQ[作为中断功能]

### 3.4.3 DTS 节点解析

需要包含如下头文件：

```
#include <linux/of_gpio.h>
```

解析 DTS 并获取编号（已经是 base id+实际的 ID）

```
// propname为完整的名字，如：led-gpios=<&ap_gpio.....，则传入的是”led-gpios”，而不是”led”
```

```
int of_get_named_gpio(struct device_node *np,
                     const char *propname, int index)
```

```
//propname默认为”gpios”，如：gpios=<&ap_gpio.....，则可以直接用如下函数
```

```
int of_get_gpio(struct device_node *np, int index)
```

### 说明

上述返回值是已经是加上 base id 了，解析完后并不会申请 GPIO 和设置方向，需要自己 request 和设置输入输出，这也是与基于描述符操作接口的区别之一。

### 3.4.4 动态开关中断

Request IRQ 后，中断就 enable 了，如果有使用过程中要动态开关中断，可参考如下方法：

- 关闭中断功能

```
irq_set_status_flags(bdata->irq, IRQ_DISABLE_UNLAZY);
disable_irq(bdata->irq);
```

- 开启中断功能

```
enable_irq(bdata->irq);
```

### 说明

在执行中断回调函数的上半部之前中断会关闭，执行完后会自动打开，因此如果在中断上半部的回调函数调用上述接口无效。

## 3.5 Kernel4.4 和 Kernel4.14 用户空间文件节点

`/sys/class/gpio/gpiochip` 节点为原生支持的文件，可用于用户空间操作 GPIO，该文件节点功能依赖 `CONFIG_GPIO_SYSFS`，Kernel4.4 和 Kernel4.14 支持，但是 Kernel5.4 及之后的版本原生已关闭。

如下以 UMS512，操作 GPIO79 为例进行说明：

### 3.5.1 查询 gpio base

轮询 `/sys/class/gpio/gpiochip*` 的 `ngpio`，查看值为 256 的路径。

```
ums512_1h10:/sys/class/gpio/gpiochip128 # cat ngpio
256
ums512_1h10:/sys/class/gpio/gpiochip128 # cat base //说明 base id 为 128
128
```

### 3.5.2 创建的节点

创建节点，编号 = base id + 实际的 ID，所以创建的编号为：128+79=207（仅以 UMS512 为例）

```
ums512_1h10:/# echo 207 > /sys/class/gpio/export
ums512_1h10:/sys/class/gpio/gpio207 # ls
active_low device direction edge power subsystem uevent value
```

### 说明

如果该 GPIO 在 Kernel 中已经被其他驱动申请过，则 `export` 会失败，错误值为 -16 (-BUSY)。如果要开放给用户空间操作，要在对应的驱动调用 `gpiod_export()` 函数，这样就会自动生成相应的文件节点。

### 3.5.3 节点操作

- `/sys/class/gpio/export` 文件用于通知系统需要导出控制的 GPIO 引脚编号。
- `/sys/class/gpio/unexport` 用于通知系统取消导出。
- `/sys/class/gpio/gpiochipX` 目录保存系统中 GPIO 设备的信息，包括控制引脚的起始编号 `base`，寄存器名称，引脚总数导出一个引脚的操作步骤：
  - `direction` 控制是输出还是输入模式
    - 如果想设置为输入：`echo in > direction`
    - 如果想设置为输出：`echo out > direction`（默认输出电平为低）
    - 如果想设置为输出且默认输出低电平：`echo low > direction`

- 如果想设置为输出且默认输出高电平：echo high > direction
- value 在输出模式时，控制高低电平，输入时则是读取外部电平
  - 高电平：echo 1 > value
  - 低电平：echo 0 > value
- edge 控制中断触发模式
  - 无：echo none > edge
  - 上升沿触发：echo rising > edge
  - 下降沿触发：echo falling > edge
  - 双边沿：echo both > edge
- active\_low 控制翻转电平，低电平有效还是高电平有效。
  - 高电平有效：echo 0 > active\_low
  - 低电平有效：echo 1 > active\_low

### 3.5.4 文件节点权限

如果是在上层代码里操作文件节点，需要设置相应节点的 SELinux 权限，关于 SELinux 权限可以参考紫光展锐发布的《Androidxx SELinux 客制化指导文档》（xx 表示 Android 版本）。

#### 📖 说明

类似/sys/class/gpio/gpio207/只是一个链接，并不是绝对路径，如果要声明权限，需要使用绝对路径，可以通过“ls -l”命令查看，如：

```
ums512_1h10:/ # ls -l sys/class/gpio/gpio207
lrwxrwxrwx 1 root root 0 2021-03-20 06:34 sys/class/gpio/gpio207
-> ../../devices/platform/soc/soc:aon/32070000.gpio/gpiochip4/gpio/gpio207
```

从上面的路径可以看出/sys/class/gpio/gpio207 实际的路径是：  
/sys/devices/platform/soc/soc:aon/32070000.gpio/gpiochip4/gpio/gpio207

假设要操作的是该节点目录下的 value，则声明权限的路径为：  
/sys/devices/platform/soc/soc:aon/32070000.gpio/gpiochip4/gpio/gpio207/value

未增加 SELinux 权限，代码执行时会访问失败并上报类似如下的 log（如果通过 adb shell setenforce 0 临时关闭，可以访问成功，但是仍然会输出如下 log），此时也可以获取到 sys/class/gpio/gpio207/value 的绝对路径。

```
03-20 17:54:30.035 2124 2124 I ng.wtsarcontrol: type=1400 audit(0.0:136): avc: denied { open } for
path="/sys/devices/platform/soc/soc:aon/32070000.gpio/gpiochip4/gpio/gpio207/value" dev="sysfs"
ino=33035 scontext=u:r:radio:s0 tcontext=u:object_r:sysfs:s0 tclass=file permissive=1
```

## 3.6 Kernel5.4 和 Kernel5.15 用户空间文件节点

Linux 内核引入了一个基于字符设备的新用户空间 API，用于管理和控制 GPIO。使用的 line 代表每个 gpiochip 上的线路，具有以下的特点：

- 每一个 gpiochip 对应一个“/dev”下的 gpiochip<number>文件。
- 使用 open、ioctl、poll 和 read 等方法操作 GPIO 口。
- 支持同时申请多个 IO 口并支持设置、获取多个 IO 口的值。

- 可以通过别名定位 gpiochip 和 line。
- 可以为 line 增加 open drain 和 open source 标识。
- 每一个 line 都有一个 consumer 字符串，用于标识 line 的使用者。

### 3.6.1 查询 gpiochip

在 Userdebug 版本，adb shell 后可以通过 **cat d/gpio** 命令，查看当前 Kernel 已经申请的 GPIO 情况，以及对应的 gpiochip。下面以 UMS512\_1h10 为例：

```
ums512_1h10:/ # cat d/gpio
gpiochip5: GPIOs 112-127, parent: platform/32100000.spi:pmic@0:gpio@280, 32100000.spi:pmic@0:gpio@280:
gpio-112 (          |vbus          ) in  hi IRQ
gpio-113 (          |Power Key      ) in  hi IRQ ACTIVE LOW
gpio-115 (          |bat-detect      ) in  hi IRQ
gpio-116 (          |Volume Up Key   ) in  lo IRQ
gpio-118 (          |aud_int_all     ) in  lo IRQ

gpiochip4: GPIOs 128-383, parent: platform/32070000.gpio, 32070000.gpio:
gpio-136 (          |mipi-switch-mode-gpi) out lo
gpio-143 (          |lr              ) out lo
gpio-144 (          |mic             ) out lo
gpio-160 (          |sdiohal_gpio    ) in  lo IRQ
```

#### 说明

gpiochip4: GPIOs 128-383（共 256 个编号），说明是普通的 GPIO，只有普通的 GPIO 才会占用 256 个编号。

### 3.6.2 节点操作

- 获取 chip\_info

chip\_info 定义：

```
struct gpiochip_info {
    char name[32];
    char label[32];
    __u32 lines;
};
```

使用 ioctl 的 GPIO\_GET\_CHIPINFO\_IOCTL 命令获取 gpiochip\_info 结构体，gpiochip\_info 包含芯片名称、标签、线路。

示例：

```
struct gpiochip_info info;
int fd, ret;

fd = open("/dev/gpiochip4", O_RDWR); //打开字符设备
```

```
ret = ioctl(fd, GPIO_GET_LINEINFO_IOCTL, &line_info);
```

- 获取 line\_info

```
struct gpioline_info {
    __u32 line_offset;
    __u32 flags;
    char name[32];
    char consumer[32];
};
```

gpioline\_info 中的 flag 包含:

```
#define GPIOLINE_FLAG_KERNEL (1UL << 0)
#define GPIOLINE_FLAG_IS_OUT (1UL << 1)
#define GPIOLINE_FLAG_ACTIVE_LOW (1UL << 2)
#define GPIOLINE_FLAG_OPEN_DRAIN (1UL << 3)
#define GPIOLINE_FLAG_OPEN_SOURCE (1UL << 4)
```

使用 ioctl 的 GPIO\_GET\_LINEINFO\_IOCTL 命令获取 line\_info, 进一步查询每条 GPIO 线的状态。

示例:

```
struct gpioline_info info;
line_info.line_offset = gpio_num;           //gpio_num为gpio编号
ret = ioctl(fd, GPIO_GET_LINEINFO_IOCTL, &line_info);
```

- 申请 lines 对其值进行操作, 需要用到的结构体。

```
struct gpiohandle_request {
    __u32 lineoffsets[GPIOHANDLES_MAX];
    __u32 flags;
    __u8 default_values[GPIOHANDLES_MAX];
    char consumer_label[32];
    __u32 lines;
    int fd;
};
```

Flags 包含:

```
#define GPIOHANDLES_MAX 64
#define GPIOHANDLE_REQUEST_INPUT (1UL << 0)
#define GPIOHANDLE_REQUEST_OUTPUT (1UL << 1)
#define GPIOHANDLE_REQUEST_ACTIVE_LOW (1UL << 2)
#define GPIOHANDLE_REQUEST_OPEN_DRAIN (1UL << 3)
#define GPIOHANDLE_REQUEST_OPEN_SOURCE (1UL << 4)
```

使用 ioctl 的 GPIO\_GET\_LINEHANDLE\_IOCTL 命令申请 lines, 对其值进行操作。

示例：

- 申请 lines 为输出

```
struct gpiohandle_request req;
req.lines = 1;           //申请1路lines
req.lineoffsets[0]= gpio_num;
req.flags = GPIOHANDLE_REQUEST_OUTPUT;
ret = ioctl(fd, GPIO_GET_LINEHANDLE_IOCTL, &req);
```

- 申请 lines 为输入

```
struct gpiohandle_request req;
req.lines = 1;           //申请1路lines
req.lineoffsets[0]= gpio_num;
req.flags = GPIOHANDLE_REQUEST_INPUT;
ret = ioctl(fd, GPIO_GET_LINEHANDLE_IOCTL, &req);
```

- 获取/设置 GPIO 的值

传递值使用 gpiohandle\_data 结构体。

```
struct gpiohandle_data {
    __u8 values[GPIOHANDLES_MAX];
};
```

示例：

- 设置 GPIO 的值

```
struct gpiohandle_data data;
data.values[0] = value;
ret = ioctl(req.fd, GPIOHANDLE_SET_LINE_VALUES_IOCTL, &data);
```

- 获取 GPIO 的值

```
ret = ioctl(req.fd, GPIOHANDLE_GET_LINE_VALUES_IOCTL, &data);
```

### 说明

如果该 GPIO 在 Kernel 中已经被其他驱动申请过，则无法被用户空间申请。

## 3.6.3 文件节点权限

参考 [3.5.4 文件节点权限](#)。

# 4 调试方法

本章介绍的调试方法均是基于 Userdebug 版本下进行的，user 版本无法使用。当 GPIO 或者 EIC 功能异常时，首先要确认的是对应 Pinmap 配置是否正确，如果正确，再根据本章介绍的方法进行查找和分析。

## 4.1 文件节点

- 通过在 `/sys/class/gpio/` 路径下创建相应的文件节点进行操作，可以直接调试相关功能，具体方法请参考 [3.5 Kernel4.4 和 Kernel4.14 用户空间文件节点](#) 的相应介绍。
- 通过 `cat /sys/kernel/debug/gpio`，查看对应的 GPIO 是否成功申请，输入（当前外部是高还是低电平）还是输出（输出高还是低）模式，是否作为中断功能使用。
- 通过 `cat /proc/interrupts`，查看中断产生次数，中断类型（level 还是 edge）。

## 4.2 读写寄存器

### 4.2.1 lookat 操作命令

执行 `adb shell`，然后执行指令 `su`，就可使用 `lookat` 命令来实现以下功能：

读取多个连续寄存器的值

- 命令格式：  

```
lookat [-l nword] addr_in_hex
```
- 示例：读取 0x32070480 开始的 8 个寄存器内容

```
lookat -l 8 0x32070480
[I] nword = 0x8
  ADDRESS |  VALUE
-----+-----
0x32070480 | 0x00001003
0x32070484 | 0x00001003
0x32070488 | 0x00001002
0x3207048c | 0x0000fffe
0x32070490 | 0x00000000
0x32070494 | 0x0000fffe
0x32070498 | 0x00000001
0x3207049c | 0x00000000
```

## 设置寄存器的值

- 命令格式: `lookat [-s value] addr_in_hex`
- 示例: 将 0x32070480 的值置为 0x0

```
lookat -s 0x0000000 0x32070480
```

如果只是设置某个 BIT 位, 则需要先获取该寄存器的值, 然后将对应的 BIT 位修改完再写入, 否则会把该寄存器的其他 BIT 位也改写了。如果要按位置 1 或清 0, 可参考如下的做法。

- 按 BIT 位置 1
  - 命令格式:

```
lookat -s bit_x reg_address+0x1000
```
  - 示例: 假设 0x32070480 原来的值为 0x00001003, 要将 BIT7 置为 1, 其他位不变, 则命令是

```
lookat -s 0x80 0x32071480
```

执行完后 0x32070480 的值变为 0x00001083
- 按 BIT 位清 0
  - 命令格式:

```
lookat -s bit_x reg_address+0x2000
```
  - 示例: 假设 0x32070480 原来的值为 0x00001003, 要将 BIT12 置为 0, 其他位不变, 则命令是

```
lookat -s 0x1000 0x32072480
```

执行完后 0x32070480 的值变为 0x00000003

## 4.2.2 GPIO 调试

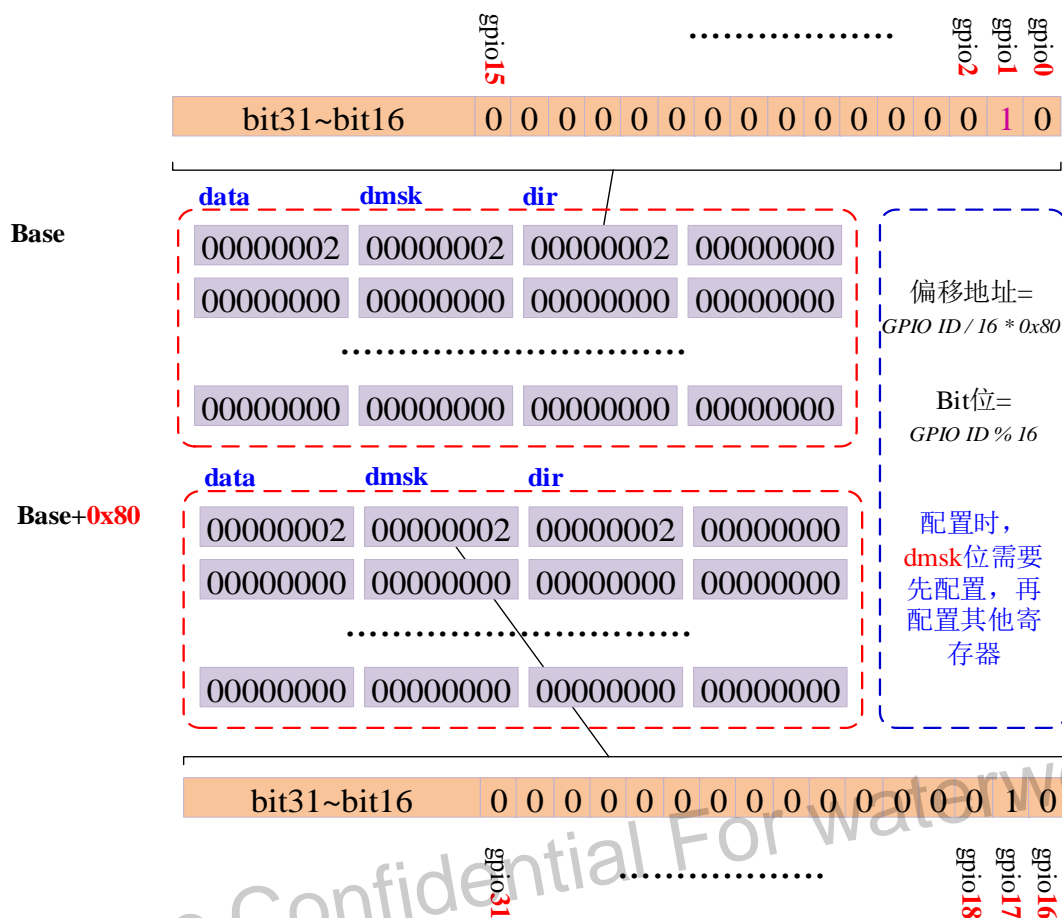
GPIO 的 IP 介绍、Base 地址等内容可以参考《XXXX Device Specification》中 GPIO 的 Control Registers 章节。Base 地址对应的是 Memory Map 的地址, 不同的芯片对应的地址可能会有所不同。

### 地址分布

GPIO 是分组控制的, 每 0x80 的地址控制 16 个 GPIO, 对应到每个寄存器的低 16 位, 具体对应关系如下图所示:



图4-1 GPIO 地址分布图



## 说明

SC9863A 使用的是 GPIO Plus，与上述的方法不一样，具体请参考《XXXX Device Specification》中 GPIO 的相关介绍。

## 地址与 GPIO ID 对应关系

地址	GPIO ID 范围
Base	GPIO0 ~ GPIO15
Base+0x80	GPIO16 ~ GPIO31
Base+0x100	GPIO32 ~ GPIO47
Base+0x180	GPIO48 ~ GPIO63
Base+0x200	GPIO64 ~ GPIO79
Base+0x280	GPIO80 ~ GPIO95
Base+0x300	GPIO96 ~ GPIO111
Base+0x380	GPIO112 ~ GPIO127

地址	GPIO ID 范围
Base+0x400	GPIO128 ~ GPIO143
Base+0x480	GPIO144 ~ GPIO159
Base+0x500	GPIO160 ~ GPIO175
Base+0x580	GPIO176 ~ GPIO191
Base+0x600	GPIO192 ~ GPIO207
Base+0x680	GPIO208 ~ GPIO223
Base+0x700	GPIO224 ~ GPIO239
Base+0x780	GPIO240 ~ GPIO255

## 说明

每颗芯片支持 GPIO 个数有所不同，并且 ID 不一定是连续的，实际支持的 GPIO 列表，请以芯片 SPEC 文档为准。

## 控制寄存器

Offset	Reg Name	Description	Notes
0x0000	gpio_data	GPIO bits data register	GPIO bits data input
0x0004	gpio_dmsk	GPIO bits mask register	GPIO DATA register can be read/write if GPIO DMSK set "1"
0x0008	gpio_dir	GPIO bits direction register	"1" configure gpio bits to be output; "0" configure gpio bits to be input.
0x000C	gpio_is	GPIO bits interrupt sense register	"1" detect signals level; "0" detect signals edge.
0x0010	gpio_ibe	GPIO bits both edges interrupt register	"1" both edges trigger an interrupt; "0" interrupt generation event is controlled by GPIO IEV.
0x0014	gpio_iev	GPIO bits interrupt event register	GPIO bits interrupt event register. "1" high level (posedge) trigger interrupts; "0" low level (posedge) trigger interrupts.
0x0018	gpio_ie	GPIO bits interrupt enable register	GPIO bits interrupt enable register. "1" corresponding bit interrupt is enabled. "0" corresponding bit interrupt isn't enabled
0x001C	gpio_ris	GPIO bits raw interrupt status register	GPIO bits raw interrupt status register. "1" interrupt condition met

Offset	Reg Name	Description	Notes
			“0” condition not met
0x0020	gpio_mis	GPIO bits mask interrupt status register	GPIO bits mask interrupt status register.(R0) “1” Interrupt active “0” interrupt not active
0x0024	gpio_ic	GPIO bits interrupt clear register	GPIO bits interrupt clear register. “1” clears detected interrupt. “0” has no effect.
0x0028	gpio_inen	GPIO input enable register	GPIO input enable register. “1” input enable. “0” input disable.

## 示例：

以 UMS512 芯片上操作 GPIO79（Function 切换为 GPIO79）为例：

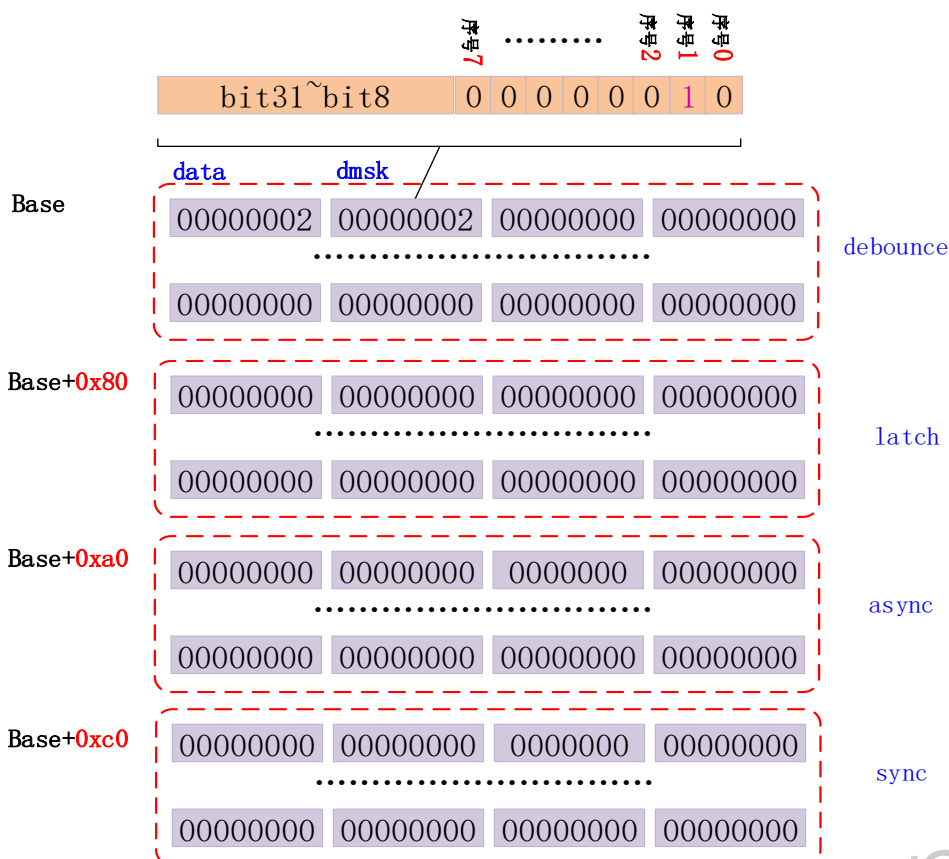
1. 通过 Memory Map 可以查到 GPIO Base 地址为：0x32070000
2. 通过本节[地址与 GPIO ID 对应关系](#)，可以查到 GPIO79 对应的地址为 Base+0x200，所操作的基地址是 0x32070200。
3. 通过本节[地址分布](#)图示的公式，操作的 BIT 位为 GPIOID % 16，所以 BIT 位=79%16，即：BIT15。
4. 再根据本节[控制寄存器](#)图示的控制寄存器，先配置 gpio\_dmsk(0x32070200+0x0004 的 BIT15)，再配置 gpio\_dir(0x32070200+0x0008 的 BIT15)，然后根据需要配置其他寄存器的值。这里需要注意的是要按 BIT 位操作，先读上来，再修改对应 BIT 位的值，再写入。

## 4.2.3 EIC 调试

每个 EIC 控制器包含 4 种模式，8 个通道（来自 PAD 的输入源），相关 IP、Base、控制寄存器请参考《XXXX Device Specification》中关于 EIC 的描述。

### EIC 寄存器的地址分布

EIC 寄存器地址分布图例如下。



## 控制寄存器

如下仅以 debounce 模式为例，如下图。

Offset	RegName	R/W	Field Description
0x0000	EIC_DBNCR_DATA	RO	EIC_DBNCR bits data input
0x0004	EIC_DBNCR_DMSK	RW	EIC_DBNCR_DATA register can be read if EIC_DBNCR_DMSK set "1"
0x0014	EIC_DBNCR_IEV	RW	EIC_DBNCR bits interrupt status register: "1" high levels trigger interrupts, "0" low levels trigger interrupts.
0x0018	EIC_DBNCR_IE	RW	"1" corresponding bit interrupt is enabled. "0" corresponding bit interrupt isn't enabled
0x001C	EIC_DBNCR_RIS	RO	"1" interrupt condition met "0" condition not met
0x0020	EIC_DBNCR_MIS	RO	"1" Interrupt active "0" interrupt not active
0x0024	EIC_DBNCR_IC	WC	"1" clears detected interrupt. "0" has no effect.
0x0028	EIC_DBNCR_TRIG	WO	"1": generate the trig_start pulse "0": no effect It must set EIC_DBNCR_TRIG for using de-bounce function and getting active interrupt
0x0040	EIC0_DBNCR_CTRL	[15]	1: clock of dbnc forced open; 0: no effect
0x005C	~	[14]	de-bounce mechanism enable or disable: 1 enable(not bypass), 0 disable(bypass)
step 0x4	EIC7_DBNCR_CTRL	[11:0]	de-bounce counter period value setting, the one unit is 0.977 (1000/1024) millisecond

## 说明

在 not bypass 模式（开启硬件防抖功能，芯片设计时的默认值）中，产生中断的条件是 EIC\_DBNC\_IC 和 EIC\_DBNC\_TRIG 对应的 BIT 位置 1，所以每次处理完中断都要将这两个寄存器的对 BIT 位置 1，才能在满足条件的时候产生下一次中断。

## 示例

UMS512 芯片上操作 GPIO79（Function 切换为 EXTINT10）为例：

1. 通过 AON Source List 可以找到 EXTINT10 在 EIC\_EXT3。

EIC_EXT3	PAD: EXTINT8	0
	PAD: EXTINT9	1
	PAD: EXTINT10	2
	PAD: EXTINT11	3
	PAD: EXTINT12	4
	PAD: EXTINT13	5
	PAD: EXTINT14	6
	PAD: EXTINT15	7

2. 在 MemoryMap 上其对应的控制基地址为 0x32030000，注意两者命名会有点差异，EIC\_EXT3 即是 AON EIC3。

Base Addr Range	Addr Map Description
0x3200_0000~0x3200_FFFF	AON EIC0
0x3201_0000~0x3201_FFFF	AON EIC1
0x3202_0000~0x3202_FFFF	AON EIC2
0x3203_0000~0x3203_FFFF	AON EIC3
0x3223_0000~0x3223_FFFF	AON EIC4
0x3227_0000~0x3227_FFFF	AON EIC5
0x3228_0000~0x3228_FFFF	WTLCP EIC
0x3247_0000~0x3247_FFFF	CPALL EIC
0x3281_0000~0x3281_FFFF	SEC EIC
0x5106_0000~0x5106_FFFF	SP GPIO EIC
0x5107_0000~0x5107_FFFF	SP EIC

3. EIC 只支持输入模式，因此无需配置模式，如果要读取外部电平高低，先配置 EIC\_DBNC\_MASK(0x32030000+0x0004 的 BIT2)，再读取 EIC\_DBNC\_DATA(0x32030000 的 BIT2)，如果 BIT2 为 1，说明此时外部电平为高，否则为低电平。

## 4.3 GPIO&EIC 中断唤醒源调试

模块使用 GPIO/EIC 作为唤醒源唤醒系统。系统进入 suspend，但频繁被 GPIO 或者 EIC 的中断唤醒，需确认唤醒源。在中断函数中添加如下打印信息确认具体的 GPIO 号/EIC 编号，然后在 DTS 中搜索是哪个模块使用的该 GPIO/EIC。

- **【GPIO】**

“drivers\gpio\gpio-sprd.c”

```
static void sprd_gpio_irq_handler(struct irq_desc *desc)
{
    for_each_set_bit(n, &reg, SPRD_GPIO_BANK_NR) {
        girq = irq_find_mapping(chip->irq.domain,
                                bank * SPRD_GPIO_BANK_NR + n);
        +pr_emerg ("sprd_gpio_irq_handler offset:%ld\n", (bank * SPRD_GPIO_BANK_NR + n)); //添加
打印
        generic_handle_irq(girq);
    }
}
```

- **【EIC】**

“drivers\gpio\gpio-eic-sprd.c”

```
static void sprd_eic_handle_one_type(struct gpio_chip *chip)
{
    struct sprd_eic *sprd_eic = gpiochip_get_data(chip);
    u32 bank, n, girq;
    .....
    for_each_set_bit(n, &reg, SPRD_EIC_PER_BANK_NR) {
        u32 offset = bank * SPRD_EIC_PER_BANK_NR + n;
        +pr_emerg ("sprd_eic_handle_one_type offset:%ld type:%d\n", offset, sprd_eic->type); //添加打
印
        girq = irq_find_mapping(chip->irq.domain, offset);
        .....
    }
}
```

- **【PMIC EIC】**

“drivers\gpio\gpio-pmic-eic-sprd.c”

```
static irqreturn_t sprd_pmic_eic_irq_handler(int irq, void *data)
{
    for_each_set_bit(n, &status, chip->ngpio) {
        /* Clear the interrupt */
        sprd_pmic_eic_update(chip, n, SPRD_PMIC_EIC_IC, 1);
        +pr_emerg ("sprd_pmic_eic_irq_handler offset:%ld\n", n); //添加打印
        .....
    }
}
```

## 示例：

1. 以 GPIO 为例，在 kernel log 中搜索 wake up by 关键字确认唤醒源是 GPIO。

```
04096 <6> [ 186.239537][09-02 19:10:29.239] #--Wake up by 61(AP_INTC:GPIO)!
```

```
0428B <6> [ 187.146824][09-02 19:10:30.146] #--Wake up by 61(AP_INTC:GPIO)!
```

- 在 GPIO 的中断处理函数中添加如下打印信息，编译版本重新测试复现。

“drivers/gpio/gpio-sprd.c”

```
static void sprd_gpio_irq_handler(struct irq_desc *desc)
{
    .....
    for_each_set_bit(n, &reg, SPRD_GPIO_BANK_NR) {
        girq = irq_find_mapping(chip->irq.domain,
            bank * SPRD_GPIO_BANK_NR + n);
        +pr_emerg ("sprd_gpio_irq_handler offset:%ld\n", (bank * SPRD_GPIO_BANK_NR + n)); //添加
打印
        generic_handle_irq(girq);
    }
}
```

- 通过查看复现 log，确认唤醒源是 gpio32。

```
27F7C <6> [ 2322.445946][09-16 11:09:16.445] #--Wake up by 61(AP_INTC:GPIO)!
```

```
27F7D <0> [ 2322.785952][09-16 11:09:16.785] sprd_gpio_irq_handler offset:32
```

- 在项目 DTS 文件中找到使用 gpio32 的模块，搜索关键字 `&ap_gpio32`。发现 gpio32 是被 WCN 模块使用，进一步由相应模块确定频繁唤醒的原因。

“arch/arm64/boot/dts/sprd/ums512-1h10-overlay.dts”

```
m2-wakeup-ap-gpios = <&ap_gpio 32 GPIO_ACTIVE_LOW>;
```

## 说明

添加的代码仅用于调试唤醒源，正式版本必须删除，否则会影响性能或正常功能

# 5 注意事项

## 5.1 GPIO 边沿触发与电平触发的区别

- 边沿触发中断是可以锁住的。  
即：产生中断后，如果外部电平发生切换，也不会导致 GPIO 中断消失，要配置对应的 Interrupt clear register 为 1 才会清除。
- 电平触发中断是随着电平的消失就消失，不会锁住。  
即：产生中断后，如果外部电平高低发生切换，会导致 GPIO 中断自动消失。  
使用电平触发要求电平稳定持续一定的时间（不同的场景时间可能不同），确保软件中断处理程序可以处理完，避免引起 AP CPU hang 等异常，因此按键、触摸、指纹等人为操作，抖动严重，不能使用 GPIO 电平触发中断。

### 注意

SC9863A 例外，支持防抖。

图5-1 电平触发示意图



## 5.2 DeepSleep 时 GPIO 唤醒 AP 系统

- 边沿触发
  - 没有 RCO 时钟方案的项目（SC9820E\SC9832E\SC7731E 等），在 chip deep sleep 时，AON\_APB 是没有时钟的，不能使用边沿触发唤醒 AP 系统。
  - 有 RCO 时钟方案的项目（UMS312\UMS512 等），在 chip deep sleep 且 SP（System Processor，系统处理器）未进入 deep sleep 时（如：sensorhub 工作时 SP 则不会进入 deep sleep），AON\_APB 有时钟的，才可以使用边沿触发唤醒 AP 系统。



考虑此场景下用边沿触发唤醒系统有上述条件的限制，因此不要使用边沿触发唤醒系统功能。

- 如果需要使用唤醒 AP 系统，建议使用支持 EIC 的 PIN 脚，具体请参考芯片 SPEC。
- GPIO\_plus 支持 debounce、latch 等模式，支持唤醒 AP 系统，如：SC9863A。

### 说明

项目是否支持 RCO 方案请联系对应的 FAE 确认。

- 电平触发：一般都支持中断唤醒系统，正如前一节所述，要求稳定持续一定的时间。**像按键、触摸、指纹等人为操作由于电平抖动严重，变化不可控，因此不能使用电平触发唤醒系统。**例外：SC9863A 使用的是 GPIO\_plus，支持防抖，可以使用。

## 5.3 GPIO 使用风险提示

中断方式支持情况：仅针对按键、触摸、指纹等抖动严重的人为操作。

表5-1 平台 GPIO 和 EIC 情况表

芯片	是否支持 RCO	GPIO 中断方式			EIC
		电平触发	边沿触发（非 sleep）	边沿触发（sleep）	
SC7731E	×	×	√	×	√
SC9820E	×	×	√	×	√
SC9832E	×	×	√	×	√
SC9863A	√	√	√	√	√
UDX710	×	×	√	×	√
UDS710	√	×	√	有依赖才支持	√
UMS312	√	×	√	有依赖才支持	√
UMS512	√	×	√	有依赖才支持	√
UMS9230	√	×	√	有依赖才支持	√
UMS9620	√	×	√	有依赖才支持	√
UMS9621	√	×	√	有依赖才支持	√
UIS7870SC	√	×	√	有依赖才支持	√
UIS7885	√	×	√	有依赖才支持	√
UWS6137	×	×	√	×	√
UMS9157	√	×	√	有依赖才支持	√

- 仅部分 PIN 支持 EIC 方式，具体需要查阅《XXXX Device Specification》中 EIC Source List。

- 上表中不能使用电平触发的方式的芯片，是由于此方式会违反了 Arm® GIC 对中断源的要求，会概率性出现手机 hang，而且该 hang 的行为通过 *minidump/log* 是无法定位的。
- SC9863A 使用的是 GPIO plus 的 IP，支持 debounce 和 latch，因此比较特殊。
- 支持 RCO 方案的项目，通过 GPIO 边沿触发唤醒系统有依赖条件（见 [5.2 DeepSleep 时 GPIO 唤醒 AP 系统](#) 里边沿触发中第 2 项描述），因此不建议使用。

Unisoc Confidential For waterworld

# 6

## 参考文档

---

1. 《XXXX Device Specification》(XXXX 表示对应的芯片名称)
2. 《Androidxx SELinux 客制化指导文档》(xx 表示 Android 版本)
3. 《XXXX Pinmap 配置指南》(XXXX 表示对应的芯片名称)

Unisoc Confidential For waterworld