技 术 文 件

技术文件名称：功耗调试与优化文档

技术文件编号：NJS20231107

版　　本：V1.0

共 <XX> 页

(包括封面)

拟　制　徐前

审　核

会　签

标准化

批　准

南京研发处-系统软件开发部

修改记录

文件编号	版本号	拟制人/修改人	拟制日期/修改日期	更改理由	主要更改内容（写要点即可）
NJS20231107	V1.0	徐前	20231107	创建	无
注：文件第一次归档时，“更改理由”、“主要更改内容”栏写“无”。					

目录

1 术语、缩略语

表 1.1

术语/定义	英文对应词	含　　义

2 概述

本文用于介绍功耗基本概念，功耗问题定位与分析调试手段及功耗优化。

3 基本概念

3.1 休眠与唤醒

3.1.1 内核的休眠机制

Android 系统刚诞生时，内核的休眠机制是在 linux kernel 原有休眠唤醒机制基础上打的一个补丁集，主要包括：
Wake Lock 唤醒锁机制；
EarlySuspend 休眠机制。
后来，随着 Android 系统的发展，市场份额越来越大，很多 DRC（driver src code）都脱离了 MainLine，因此，Linux 社区认为有必要将 Android 特性合并到 Linux 内核中，改名为 Autosleep 休眠机制，并且保留了 WakeLock 唤醒锁机制；
后来由于一些移动设备（例如 Android）需要满足频繁的唤醒和休眠设备，这样就会引入一个问题：系统 suspend 同步问题（在 suspend 过程中有唤醒事件产生的唤醒和休眠同步问题）。经过 linux 社区的长期辩论，提出了一种临时的解决方案，即 wakeup events framework（主要包括 Wake Lock，Autosleep，Wakeup count 机制）。

Linux 引入 Autosleep 和 Wakeup_count 机制主要是因为 Earlysuspend 被内核抛弃，需要自己打补丁，且使用 Earlysuspend，对内核改动较大，更改了 Suspend 流程，无法合入 Mainline，对设备的驱动影响较大，需要额外实现 Earlysuspend 接口，不利于 Upstream，Earlysuspend 已存在的问题，例如只支持开关屏时执行 Earlysuspend，不利于动态功耗的优化等。

三种休眠机制对比

我们现在使用的 RK 平台的情况是 Earlysuspend，Autosleep 已经不使用了（代码还在宏 CONFIG_PM_AUTOSLEEP 和 CONFIG_HAS_EARLYSUSPEND 都是关的），wake lock 和 wakeup count 机制仍在使用。底层休眠唤醒和锁机制的核心是：

System wakeup events framework
代码路径：
drivers/base/power/wakeup.c

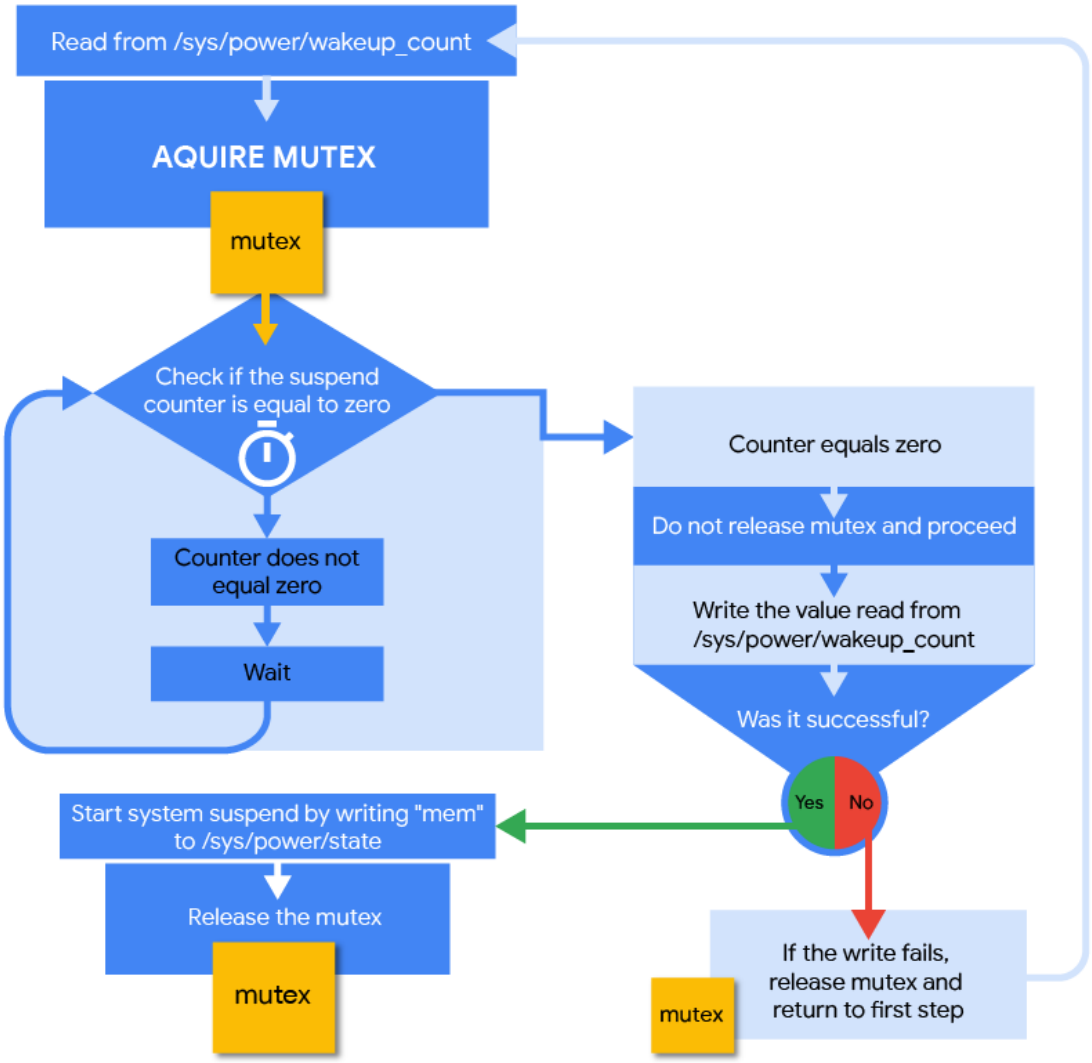
内核具体 suspend 流程：
Linux 电源管理(6)_Generic PM 之 Suspend 功能
这是底层的休眠唤醒机制，目前上层是使用 SystemSuspend 服务。

3.1.2 SystemSuspend 服务

在 Android 9 及更低版本中，libsuspend 中有一个负责发起系统挂起的线程。Android 10 在 SystemSuspend HIDL 服务中引入了等效功能。此服务位于系统映像中，由 Android 平台提供。libsuspend 的逻辑基本保持不变，以下情况除外：阻止系统挂起的每个用户空间进程都需要与 SystemSuspend 进行通信。

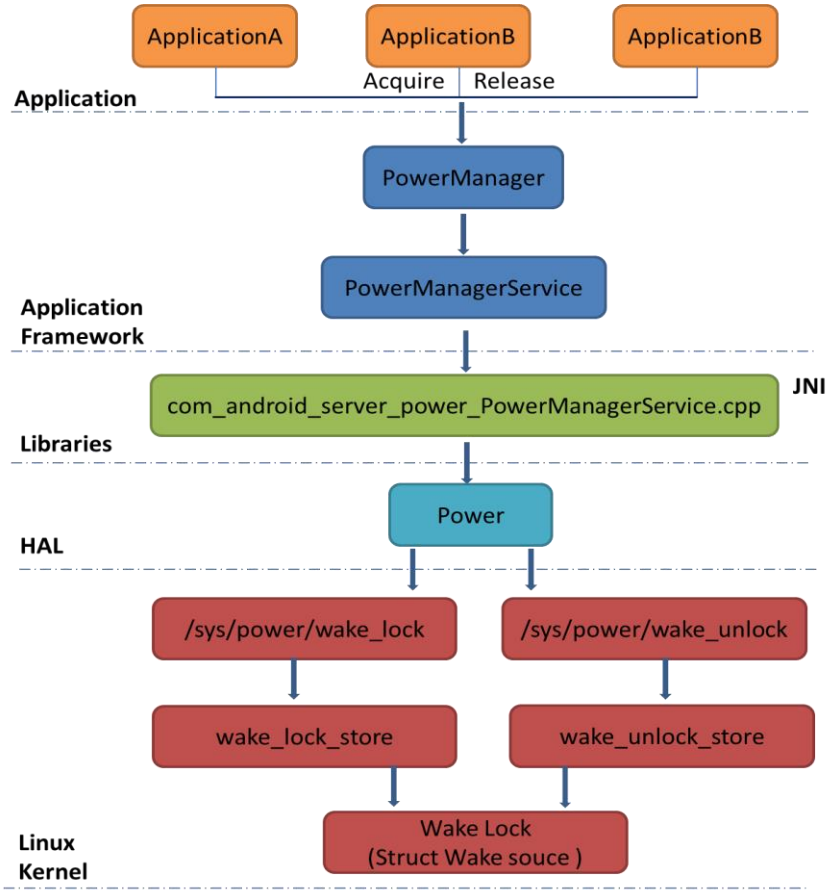
在 Android 10 中，SystemSuspend 服务取代了 libsuspend。libpower 经过重新实现，目前依赖于 SystemSuspend 服务（而不是 /sys/power/wake[un]lock），且不必更改 C API。

SystemSuspend 服务使用挂起计数器跟踪发出的唤醒锁数量。它有两个执行线程：
1、主线程响应 binder 调用。
2、挂起线程控制系统挂起。
主线程：
主线程响应来自客户端的请求以分配新的唤醒锁，从而递增/递减挂起计数器。
挂起线程：
挂起线程循环执行以下操作：
从 /sys/power/wakeup_count 读取。
获取互斥量。这可确保挂起线程在主线程尝试递增或递减挂起计数器时不会触发挂起计数器。如果挂起计数器达到零，并且挂起线程尝试运行，则系统会在发出或移除唤醒锁定时阻止主线程。等到计数器等于零。
将从 /sys/power /wakeup_count（第 1 步中）读取的值写入此文件。 如果写入失败，则返回到循环的开头。
通过将 mem 写入 /sys/power/state 来发起系统挂起。
释放互斥量。
成功返回唤醒锁定的请求后，挂起线程会被阻止。
流程图：



3.2 锁机制

Android 设备的休眠和唤醒主要基于 WakeLock 机制。WakeLock 是一种上锁机制，只要有进程获得了有效的 WakeLock 锁，系统就不会进入休眠，但可以进行浅度休眠操作，例如，在 lcd 和 tp 都关闭的情况下进行下载文件或播放音乐，这种现象就是浅待机，也称作浅休眠；当所有的 WakeLock 锁都无效时，系统就会进入深度待机（休眠）状态。WakeLock 可以设置超时，超时后会自动解锁。WakeLock 涉及到 Framwoke 层，Naticve 层以及 Kernel 层，整个框架如下图所示



本章节设计到源码文件名及位置：

PowerManager.java(/frameworks/base/core/java/android/os/PowerManager.java);

PowerManagerService.java(/framworks/base/service/core/java/com/android/server/power/PowerManagerService);

Power.c(/hardware/libhardware_legacy/power/Power.c);

Main.c(/kernel/kernel/power/Main.c);

WakeLock.h (/kernel/include/linux/WakeLock.h);

Pm_wakeup.h(/kernel/incude/linux/Pm_wakeup.h);

Wakeup.c(/kernel/drivers/base/power/Wakeup.c);

3.2.1 上层锁机制

Android wakelocks 提供的功能包括：

1）一个 sysfs 文件：/sys/power/wake_lock，用户程序向文件写入一个字符串，即可创建一个 wakelock，该字符串就是 wakelock 的名字。该 wakelock 可以阻止系统进入低功耗模

式。

2）一个 sysfs 文件：：/sys/power/wake_unlock，用户程序向文件写入相同的字符串，即可注销一个 wakelock。

3）当系统中所有的 wakelock 都注销后，系统可以自动进入低功耗状态。

4）向内核其它 driver 也提供了 wakelock 的创建和注销接口，允许 driver 创建 wakelock 以阻止睡眠、注销 wakelock 以允许睡眠。

App 层 wakelock：

应用使用 WakeLock 功能前，往往是需要先通过 getSystemService(Context.POWER_MANAGER)方法获取 PowerManager 对象，PowerManager 是 Framework 层给上层应用提供的调用接口。然后使用 PowerManager 对象创建一个 WakeLock 对象，通过 WakeLock 的 acquire()方法禁止系统休眠，应用完成工作后调用 release()方法来恢复休眠机制，否则系统将无法休眠，直到耗光所有电量。代码调用如下：

```
PowerManager pm= (PowerManager)Context.getSystemService(Context.POWER_MANAGER);
WakeLock wl = pm.newWakeLock(PowerManager.ACQUIRE_CAUSES_WAKEUP|PowerManager.SCREEN_BREIGHT_WAKE_LOCK, “bright”);// 创建一个 PowerManager.WakeLock 对象
wl.acquire();
……screen will stay on during this section
wl.release();
```

NATIVE 层 wakelock：

frameworks/native/services/sensorservice/SensorService.cpp

```
const char* SensorService::WAKE_LOCK_NAME = "SensorService_wakelock";
void SensorService::setWakeLockAcquiredLocked(bool acquire) {
    if (acquire) {
        if (!mWakeLockAcquired) {
            acquire_wake_lock(PARTIAL_WAKE_LOCK, WAKE_LOCK_NAME);
            mWakeLockAcquired = true;
        }
        mLooper->wake();
    } else {
        if (mWakeLockAcquired) {
            release_wake_lock(WAKE_LOCK_NAME);
            mWakeLockAcquired = false;
        }
    }
}
```

应用层提供的锁类型如下，这些锁都需要手动释放：

FLAG	CPU	屏幕	键盘
PARTIAL_WAKE_LOCK	开启	关闭	关闭
SCREEN_DIM_WAKE_LOCK	开启	变暗	关闭
SCREEN_BRIGHT_WAKE_LOCK	开启	变亮	关闭
FULL_WAKE_LOCK	开启	变亮	变亮

PowerManager.PARTIAL_WAKE_LOCK：

唤醒锁级别：确保 CPU 正在运行；屏幕和键盘 背光熄灭。

如果用户按下电源按钮，那么屏幕将会被关闭，但是 CPU 将一直保持着，直到所有唤醒锁被释放。

PowerManager.SCREEN_DIM_WAKE_LOCK：

已废弃，不管它。

PowerManager.SCREEN_BRIGHT_WAKE_LOCK：

已废弃，不管它。

PowerManager.FULL_WAKE_LOCK：

已废弃，不管它。

PowerManager.PROXIMITY_SCREEN_OFF_WAKE_LOCK：

唤醒锁定级别：当接近传感器激活时，关闭屏幕。

如果接近传感器检测到附近有物体，屏幕将立即关闭，但 CPU 会保持唤醒。在物体移动后不久，屏幕再次打开。

PowerManager.DOZE_WAKE_LOCK：

唤醒锁级别：使屏幕处于低功耗状态，如果没有其它锁，那么允许 CPU 暂停。

它已被{@hide}标识，不对外开放。

PowerManager.DRAW_WAKE_LOCK：

唤醒锁级别：保持设备足够清醒，以便进行绘制。

它已被{@hide}标识，不对外开放。

最新支持哪些锁可查安卓官网：

https://developer.android.google.cn/reference/android/os/PowerManager

3.2.2 内核锁机制

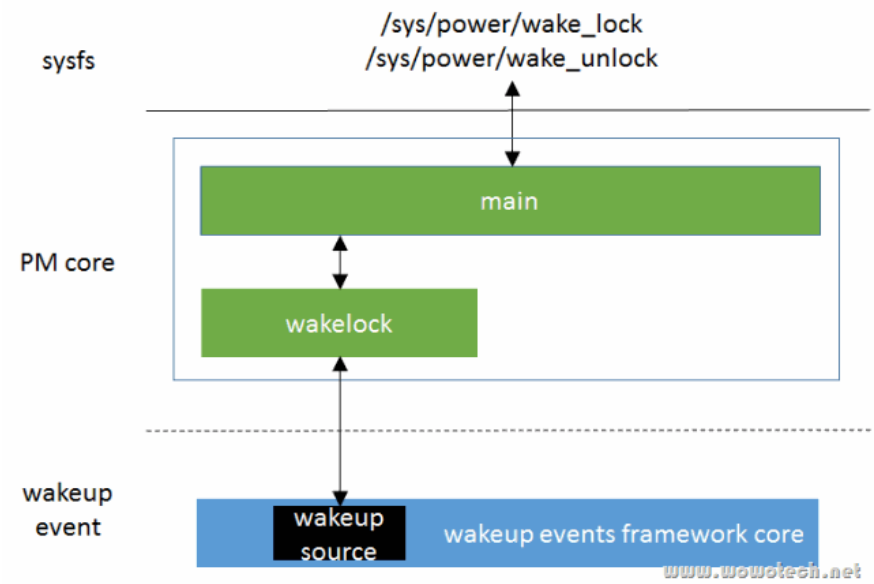
对比 Android wakelocks 要实现的功能，Linux kernel 的方案是：

允许 driver 创建 wakelock 以阻止睡眠、注销 wakelock 以允许睡眠：已经由 wakeup source 取代。

当系统中所有的 wakelock 都注销后，系统可以自动进入低功耗状态。

wake_lock 和 wake_unlock 功能：由本文所描述的 kernel wakelocks 实现，其本质就是将 wakeup source 开发到用户空间访问。

相比 Android wakelocks，Kernel wakelocks 的实现非常简单，就是在 PM core 中增加一个 wakelock 模块（kernel/power/wakelock.c），该模块依赖 wakeup events framework 提供的 wakeup source 机制，实现用户空间的 wakeup source（就是 wakelocks），并通过 PM core main 模块，向用户空间提供两个同名的 sysfs 文件，wake_lock 和 wake_unlock。



WakeLock 定义和接口在 WakeLock.h 和 Pm_wakeup.h 中进行定义，具体代码如下：

```
enum {
WAKE_LOCK_SUSPEND, //防止进入深度休眠
WAKE_LOCK_TYPE_COUNT //唤醒锁的数量
};

struct wakelock {
 char                    *name;
struct rb_node           node;
struct wakeup_source     ws;
#ifdef CONFIG_PM_WAKELOCKS_GC
struct list_head         lru;
#endif
};
```

WAKE_LOCK_SUSPEND 变量是防止进入深度休眠，WAKE_LOCK_TYPE_COUNT 表示唤醒锁的数量。Wakelock 中一个 name 指针，保存 wakelock 的名称；一个 rb node 节点，用

于组成红黑树；一个 wakeup source 变量；如果开启了 wakelocks 垃圾回收功能，一个用于 GC 的 list head。结构体 wakeup_source 在文件 Pm_wakeup.h 中定义，代码如下：

```
struct wakeup_source {
 const char              *name; //名称
 struct list_head entry; //链表节点
 spinlock_t        lock; //
 struct timer_list timer;
 unsigned long         timer_expires; //超时时间
 ktime_t total_time; //锁使用的时间
 ktime_t max_time;    //锁使用时间最长的一次
 ktime_t last_time;   //锁上次操作时间
 ktime_t start_prevent_time; //锁开始阻止时间
 ktime_t prevent_sleep_time; //阻止自动睡眠时间
 unsigned long         event_count; //事件触发数
 unsigned long         active_count; //处于激活状态计数
 unsigned long         relax_count; //处于非激活态计数
 unsigned long         expire_count; //超时数
 unsigned long         wakeup_count; //唤醒计数
 bool                active:1;   //处于激活态判断
 bool                autosleep_enabled:1; //是激化自动休眠模式
};
```

可以看到休眠锁 WakeLock 主要用于阻止系统进入深度休眠模式。WakeLock 主要部件有锁名称、链表节点以及统计锁的使用信息。接下来我们分析 WakeLock 对外提供的接口：

- **内核空间接口**

```
static inline void wake_lock_init(struct wake_lock *lock, int type, const char *name);
static inline void wake_lock_destroy(struct wake_lock *lock);
static inline void wake_lock(struct wake_lock *lock);
static inline void wake_lock_timeout(struct wake_lock *lock, long timeout);
static inline void wake_unlock(struct wake_lock *lock);
static inline int wake_lock_active(struct wake_lock *lock);
```

其中 wake_lock_init 用于初始化一个新锁，type 表示锁的类型；wake_lock_destroy 表示注销一个锁；wake_lock 和 wake_lock_timeout 用于初始化完成的锁激活，使之成为有效的永久锁和超时锁；wake_unlock 用于解锁使之成为无效锁；wake_lock_active 用于判断当前锁是否有效，如果有效则返回非 0 值。

内核超时锁例子：
http://10.110.1.100/#/c/RK3588/rk/kernel/+/754/

3.3 电源域和电压域

3.3.1 电压域和电源域

为了更好地对电进行控制，ARM 划分了两个电相关的概念：

电压域（voltage domain）

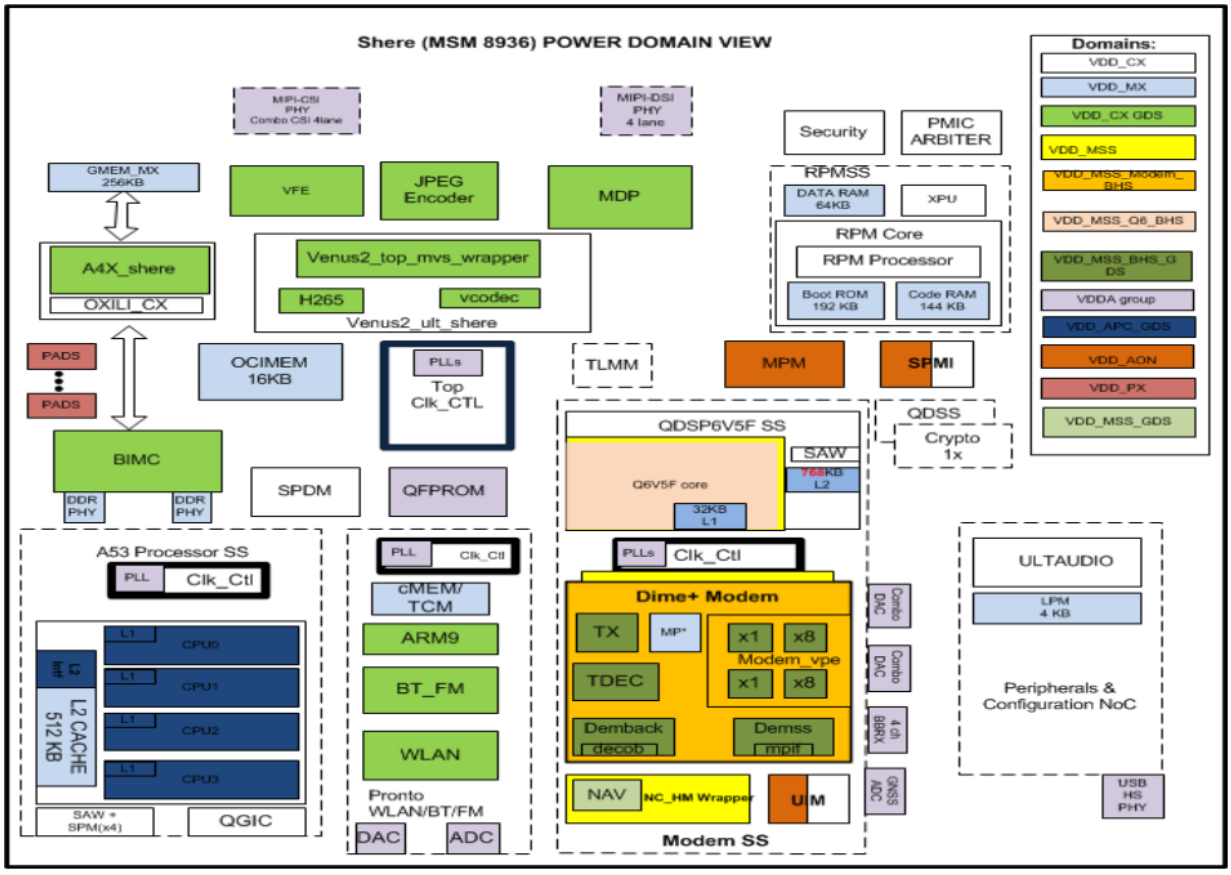
电源域（power domain）

电压域指使用同一个电压源的模块合集，如果几个模块使用相同的电压源，就认为这几个模块属于同一个电压域。

电源域指的是在同一个电压域内，共享相同电源开关逻辑的模块合集。即在同一个电源域的模块被相同的电源开关逻辑控制，同时上、下电。

一个电压域内的模块，可以根据设计需求，拆分到不同电源域。电压域对应的是功能是 dvfs，而电源域对应的是 power gating。

高通平台电源域示意图：



RK 平台电源域示意：

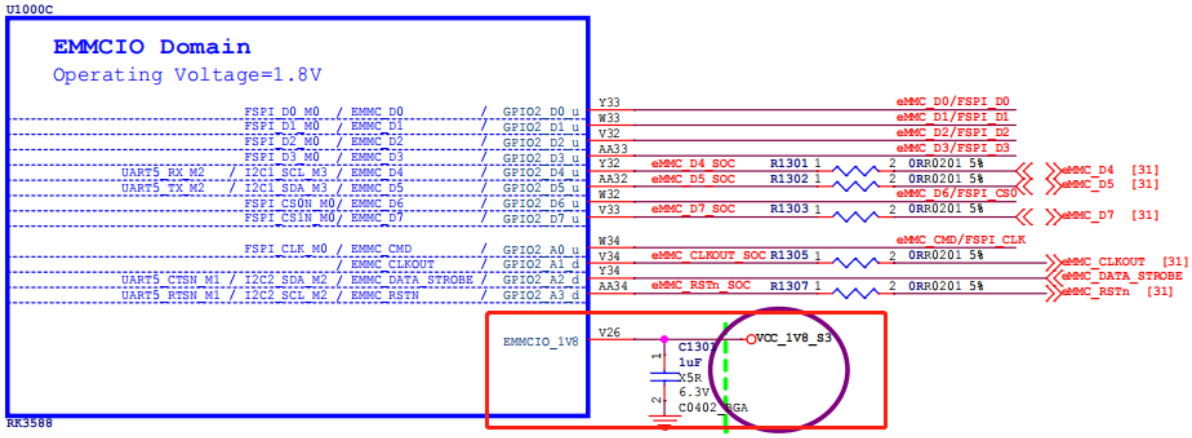
IO 电源域表格：

IO Power Domain Map

IO Domain	Pin Num	Support IO Voltage	Supply Power Pin Name	Power Source	Operating Voltage
PMUIO1	Pin N28	1.8V Only	PMUIO1_1V8	VCC_1V8_S3	1.8V
PMUIO2	Pin R27 Pin P28	1.8V or 3.3V	PMUIO2_1V8 PMUIO2	VCC_1V8_S3 VCC_3V3_S3	3.3V
EMMCIO	Pin V26	1.8V Only	EMMCIO_1V8	VCC_1V8_S0	1.8V
VCCIO1	Pin G20	1.8V Only	VDDIO1_1V8	VCC_1V8_S0	1.8V
VCCIO2	Pin AA7 Pin Y7	1.8V or 3.3V	VDDIO2_1V8 VCCIO2	VCC_1V8_S0 VCC_1V8/3V3_S0	1.8V/3.3V
VCCIO3	Pin Y26	1.8V Only	VDDIO3_1V8	VCC_1V8_S0	1.8V
VCCIO4	Pin H20 Pin H21	1.8V or 3.3V	VDDIO4_1V8 VCCIO4	VCC_1V8_S0 VCC_1V8_S0	1.8V
VCCIO5	Pin W25 Pin W26	1.8V or 3.3V	VDDIO5_1V8 VCCIO5	VCC_1V8_S0 VCC_1V8_S0	1.8V
VCCIO6	Pin AC25 Pin AC26	1.8V or 3.3V	VDDIO6_1V8 VCCIO6	VCC_1V8_S0 VCC_3V3_S0	3.3V

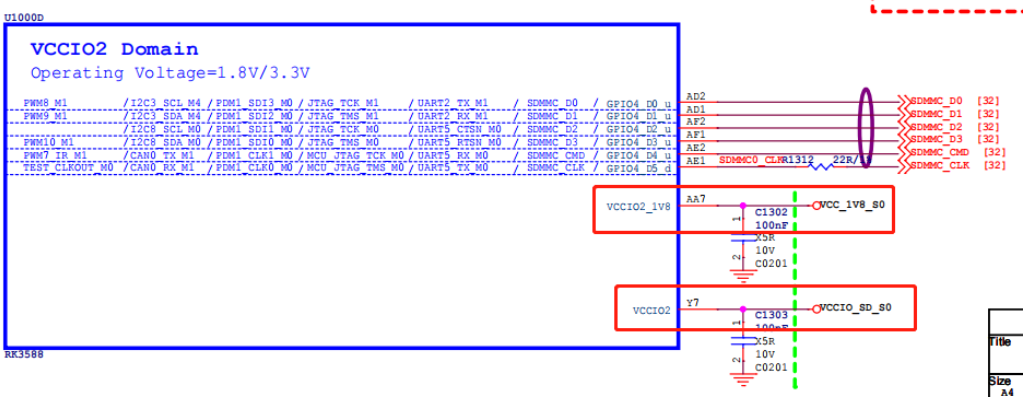
原理图上有该模块由哪个电源域供电：

RK3588_C(EMMCIO Domain)



RK3588_D(VCCIO2 Domain)

Note:
Caps of betwe
should be pla
Other caps sh



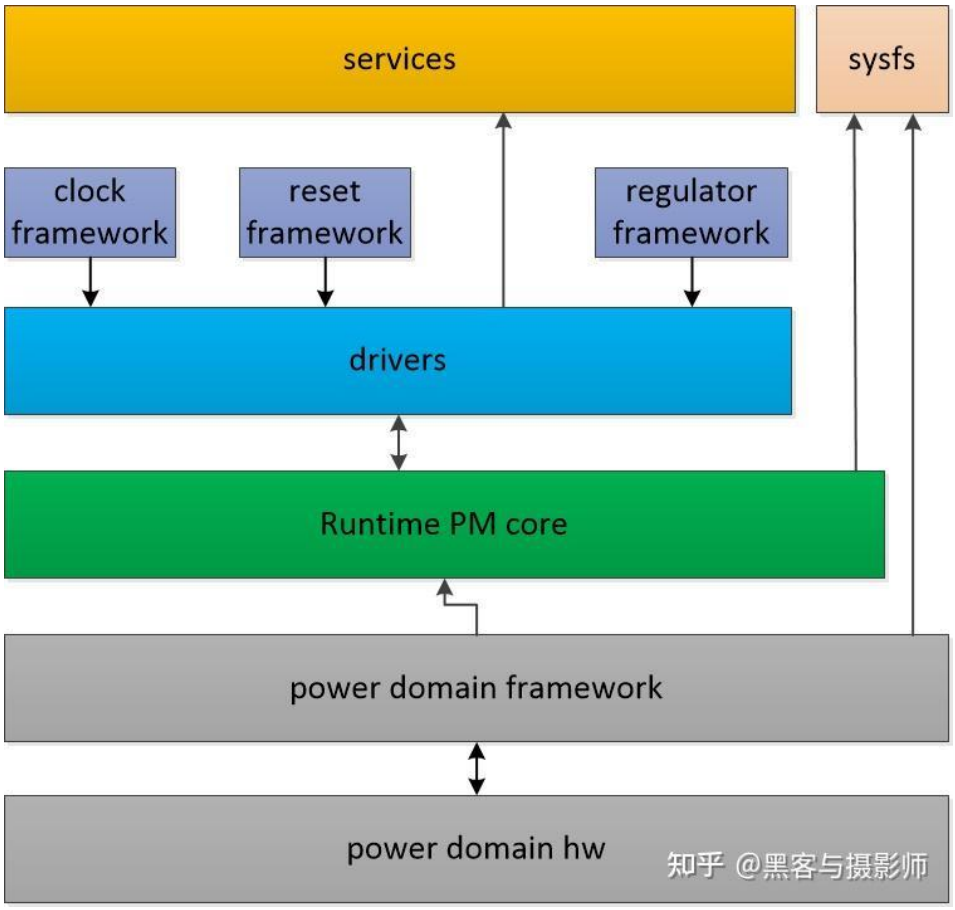
3.3.2 内核电源域框架

从字面上理解，power domain 指的是电源域。SOC 是由多功能模块组成的一个整体，对于工作在相同电压且功能内聚的功能模块，可以划为一个逻辑组，这样的一个逻辑组就是一个电源域。简单来说，电源域就是逻辑划分，在该逻辑划分中包含了物理实体和电源线的连接关系。电源域之间存在着包含关系，这样就是一个父子关系的电源域。电源域也存在着兄弟关系，这样就是同一级的电源域。因此，SOC 上众多电源域组成了一个电源域树，他们之间存在着相互的约束关系，子电源域打开前，需要父电源域打开，父电源域下所有子电源域关闭，父电源域才能关闭。

随着工艺制程越来越小、芯片的规模越来越大，芯片的 leakage 越来越大。为了减少 leakage，很重要的一个优化方式就是划分 power domain，根据需要，关闭不用的 power domain，从而有效地减少 SOC 的 leakage。

在理解了 power domain 后，我们再看一下 power domain framework。它主要是软件层面的一个概念，用来管理 SOC 上各 power domain 的依赖关系及根据需要做开关。它提供了管理和使用系统 power domain 的统一方法，结合 kernel 的 suspend、runtime pm、clock framework、regulator framework 等机制，以非常巧妙、灵活的方式，管理系统功耗，以达到高效、节能的目的。

内核电源域框架图：



power domain framework 主要管理 power domain 的状态，为使用它的上游驱动、框架或者用户空间所使用的文件操作节点，提供功能接口，对下层的 power domain hardware 的开关操作进行封装，然后内部的逻辑，实现具体的初始化、开关等操作。

对底层 power domain 硬件的操作：

对 power domain hw 的开启操作，包括开钟、上电、解复位、解除电源隔离等操作的功能封装；

对 power domain hw 的关闭操作，包括关钟、断电、复位、做电源隔离等操作的功能封装；

内部逻辑实现：

通过 dts 描述 power domain 框架的设备节点，并描述每个 power domain 节点。提供出一个 power domain framework 的设备节点，及每个 power domain 子设备的节点，并指定 power-domain-ccell = <1>，这样可以通过 power domain framework 的设备及 power domain 的编号查找具体的 power domain；

实现 dts 解析逻辑，获取 power domain 的配置信息，并通过初始化函数对每个 power domain 进行初始化，所有的 power domain 统一的放在一个全局链表中，将 power domain 下所有的设备，放到其下的一个设备链表中；

为 runtime pm、系统休眠唤醒等框架，注册相应的回调函数，并实现具体的回调函数对应的 power domain 的开关函数；

上层使用 power domain 的上游驱动、框架及 debug fs

使用 power domain 的上游驱动：power domain 内部 ip 的驱动。比如，dsp 子系统 power domain 下面，有多个 dsp 核，每个 dsp 核对应一个 power domain。这样，每个 dsp 核设备驱动都要关联到对应的 dsp 核 power domain 上，通过 dsp 核设备的 dts 里 power-domains 的属性引用（power domain framework 的节点引用及具体 power domain 的编号），将 dsp 核设备与相应的 power domain 关联起来；

使用 power domain 的框架：runtime pm/系统休眠唤醒；

linux 系统的 runtime power manager 框架通过提供 runtime_pm_get_xxx/runtime_pm_put_xxx 类接口给其他的 drivers，对设备的开、关做引用计数，当引用计数从 1->0 时，会进一步调到 power domain 注册的 runtime_suspend 回调，回调函数里会先调用设备的 runtime_suspend 回调，然后判断 power domain 下的设备链表中所有的设备是否已经 suspend，若已经 suspend 才真正关闭 power domain。当引用计数从 0->1 时，会先调用到 power domain 使用的 runtime_resume 回调，回调函数里会先调用 power domain 的开启操作，然后调用设备注册的 runtime_resume 回调函数；

系统休眠唤醒在 suspend_noirq/resume_noirq 时会进行 power domain 的关闭与开启的操作；

power domain 也提供了一些 debug fs 文件节点供用户 debug 使用，主要就是 /sys/kernel/debug/pm_genpd/目录及 power domain 名字目录下的一些文件节点：

pm_genpd_summary：打印所有的 power domain、状态及下面所挂的设备状态

power_domain 名字目录/current_state：power domain 当前的状态

power_domain 名字目录/sub_domains：power domain 当前的子 power domain 有

哪些

power_domain 名字目录/idle_states：power domain 对应的所有 idle 状态及其 off 状态的时间

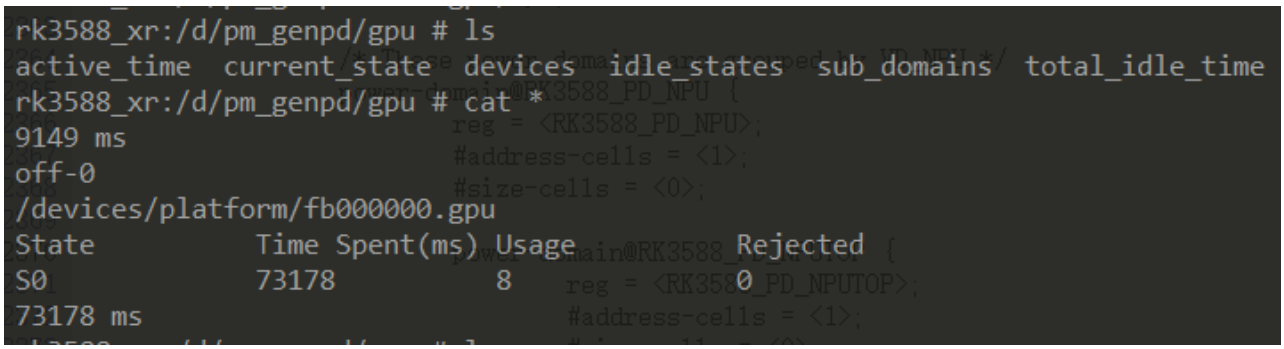
power_domain 名字目录/active_states：power domain 处于 on 状态的时间

power_domain 名字目录/total_idle_time：power domain 所有 idle 状态的 off 时间总和

power_domain 名字目录/devices：power domain 下所挂的所有 devices

power_domain 名字目录/perf_state：power domain 所处的性能状态

单独看某一个电源域的状态：

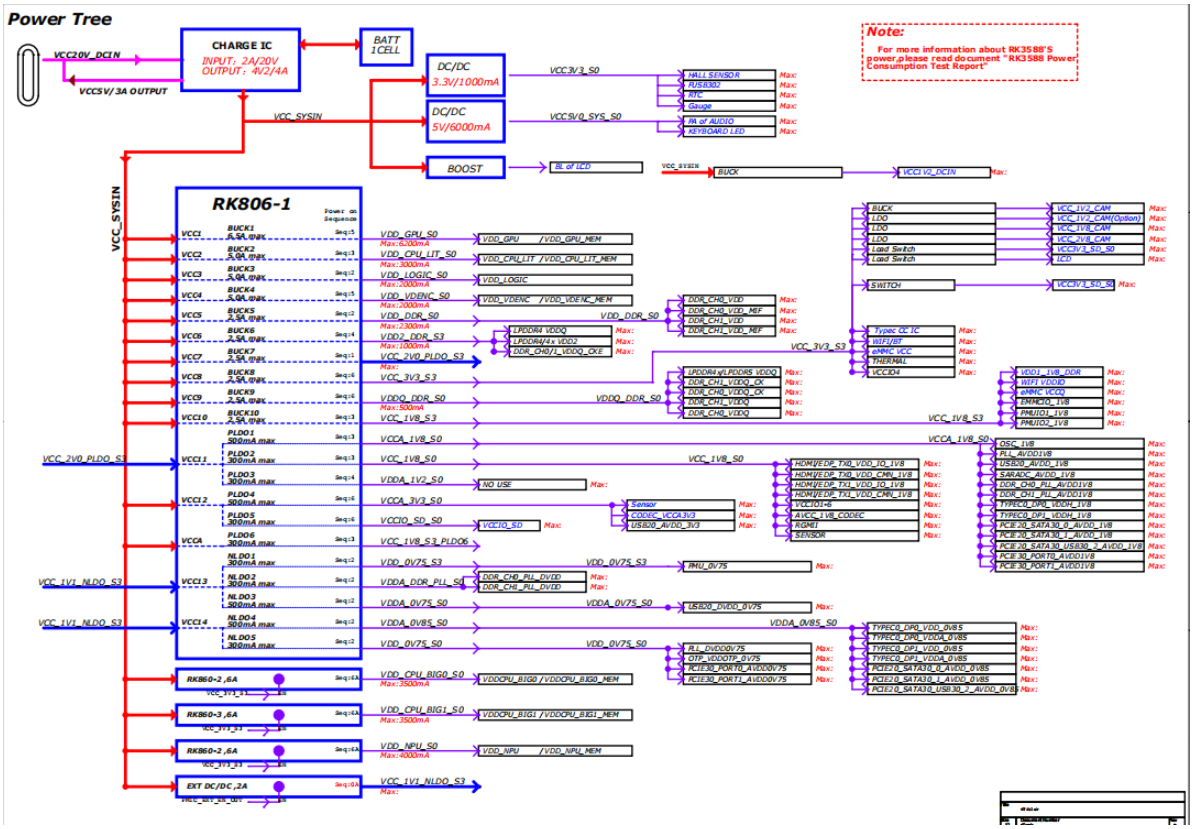


看整个系统电源域状态：

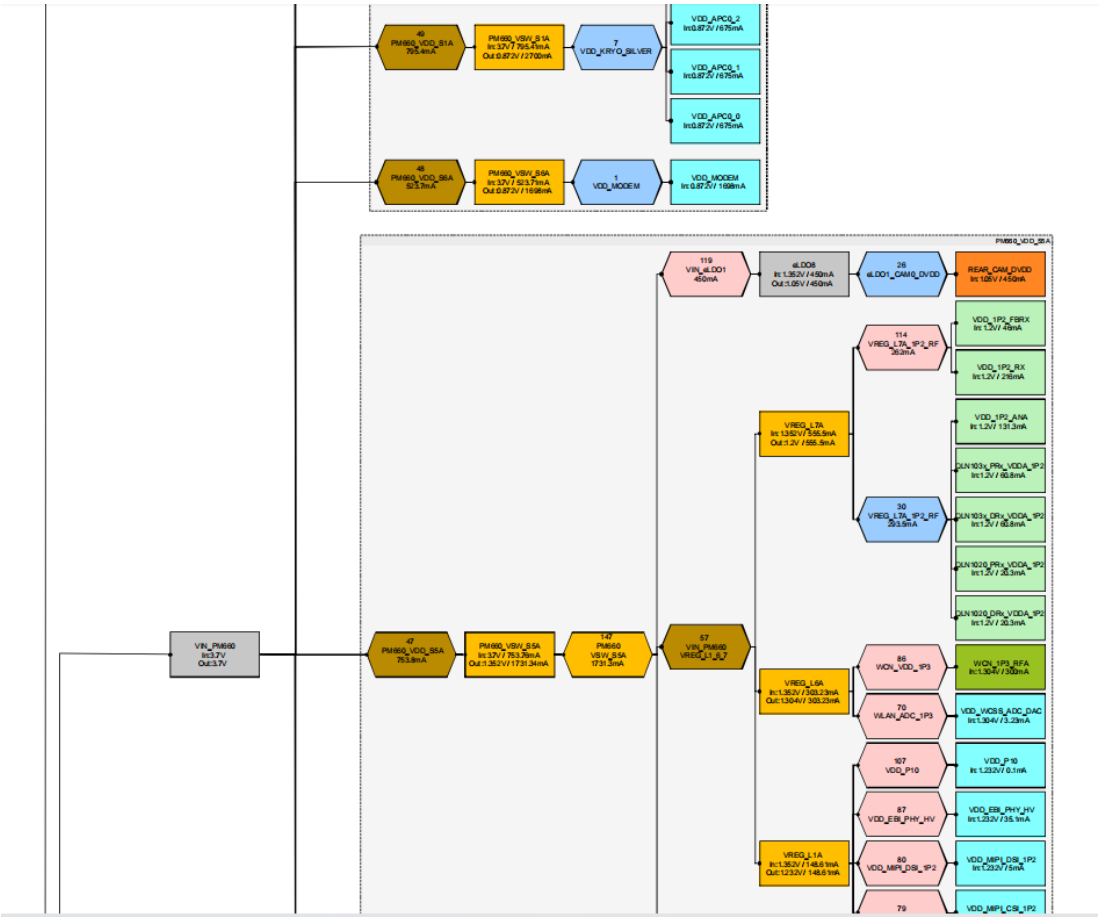
```
rk3588_xr:/d/pm_genpd # cat pm_genpd_summary
domain                          status    children
    /device                                             runtime status
----------------------------------------------------------------------
sdmmc                           on
    /devices/platform/fe2c0000.mmc                      unsupported
audio                           off-0
    /devices/platform/fe4a0000.i2s                      suspended
    /devices/platform/fe470000.i2s                      suspended
sdio                            off-0
pcie                            off-0
gmac                            off-0
usb                             on
    /devices/platform/usbdrd3_0/fc000000.usb            active
    /devices/platform/usbdrd3_1/fc400000.usb            active
rga31                           off-0
    /devices/platform/fdb70f00.iommu                    suspended
    /devices/platform/fdb70000.rga                      suspended
fec                             off-0
    /devices/platform/fdcd0f00.iommu                    suspended
    /devices/platform/fdcd8f00.iommu                    suspended
    /devices/platform/fdcd0000.rkispp                   suspended
    /devices/platform/fdcd8000.rkispp                   suspended
isp1                            off-0
    /devices/platform/fdcc7f00.iommu                    suspended
    /devices/platform/fdcc0000.rkisp                    suspended
vi                              off-0           isp1, fec
    /devices/platform/fdcb7f00.iommu                    suspended
    /devices/platform/fdce0800.iommu                    suspended
    /devices/platform/fdce0000.rkcif                    suspended
    /devices/platform/fdcb0000.rkisp                    suspended
vo1                             off-0
vo0                             on
    /devices/platform/fde50000.dp                       active
vop                             on              vo0
    /devices/platform/fdd90000.vop                      suspended
    /devices/platform/fde20000.dsi                      suspended
    /devices/platform/fde30000.dsi                      suspended
rga30                           off-0
    /devices/platform/fdb60f00.iommu                    suspended
    /devices/platform/fdb60000.rga                      suspended
av1                             off-0
    /devices/platform/fdca0000.iommu                    suspended
    /devices/platform/av1d-master                       suspended
vdpu                            off-0           av1, rkvdec0, rkvdec1, rga30
    /devices/platform/fdb50800.iommu                    suspended
    /devices/platform/fdb90480.iommu                    suspended
```

3.3.3 power tree

RK 平台 power tree 示例：



高通平台 power tree 示例：



3.4 Wakeup events framework

3.4.1 上层锁机制

暂无

3.4.2 上层锁机制

暂无

4 功耗优化

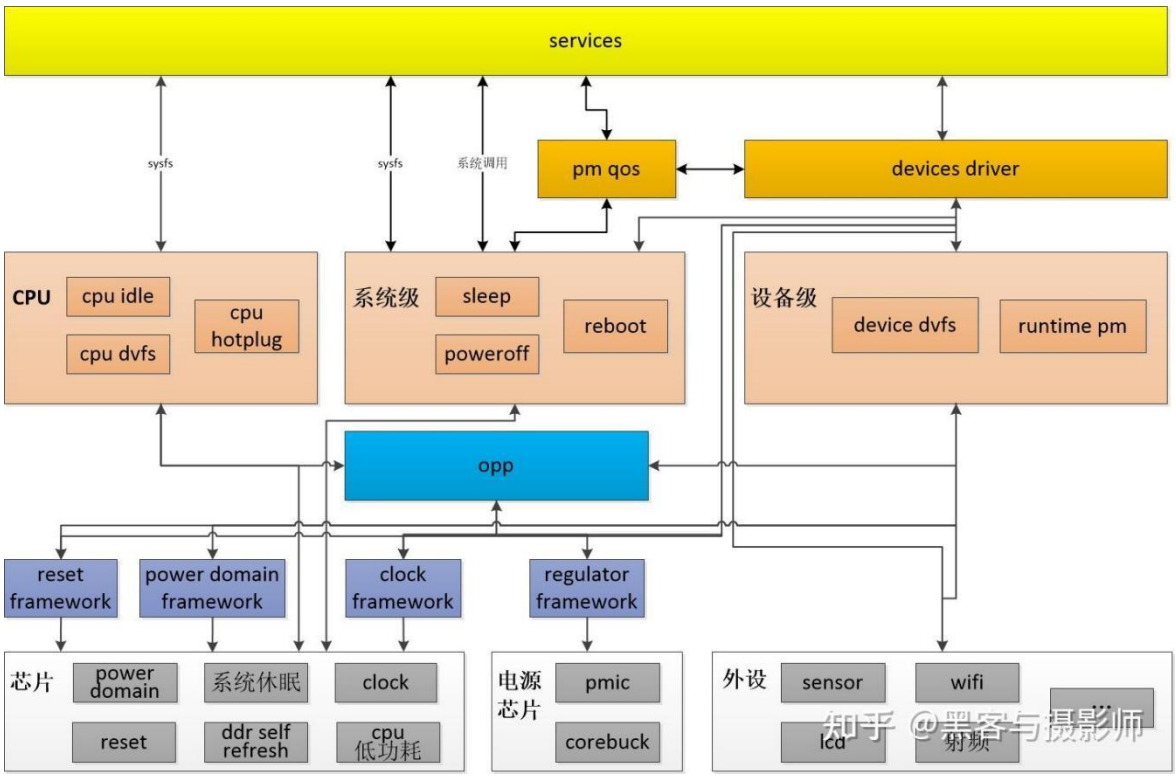
暂无

4.1 内核功耗优化机制

为了解决不必要功耗的消耗，linux 提供了多种电源管理方式。为了解决系统不工作时的功耗消耗，linux 提供了休眠（suspend to ram or disk）、关机（power off）、复位（reboot）。

为了解决运行时不必要的功耗消耗，linux 提供了 runtime pm、cpu/device dvfs、cpu hotplug、cpu idle、clock gate、power gate、reset 等电源管理的机制。为了解决运行时电源管理对性能的影响，linux 提供了 pm qos 的功能，用于平衡性能与功耗，这样既能降低功耗，又不影响性能。

电源管理框架图：



4.1.1 内核功耗优化机制概览

Clock framework：时钟管理框架统一管理系统的时钟资源，这些时钟资源包括 pll/mux/div/gate 等，对时钟资源操作做封装，维护成一个时钟树，抽象频率获取、频率设置、时钟开关等核心逻辑，为其他驱动子模块提供频率调整、频率查询、使能、失能等接口；

Regulator framework：电压管理框架统一管理系统的供电资源。这些供电资源包括 pmic、corebuck 等，对供电资源操作做封装，一般通过 iic、spi、gpio、pwm 等控制供电资源的电压、电流调整或者开/关，为其他驱动或者框架提供电压、电流调整、开关等接口；

Power domain framework：power domain 管理框架统一管理芯片的 power domain。对芯片内的 power domain 开关操作做封装，抽象核心的开关处理逻辑，为 runtime pm、系统休眠唤醒提供开关等接口；

Reset framework：reset 管理框架统一管理芯片的复位。对芯片内的 reset、dereset 寄存器

操作做封装。抽象核心的 reset/dereset 处理逻辑，为其他驱动提供统一的 reset/dereset 等接口；

Opp framework: opp（Operating Performance Point）管理框架统一管理使 CPU 或者 Devices 正常工作的电压和频率组合。内核提供这个 Layer，是为了给出一些相对固定的电压和频率组合，从而使调频调压变得更为简单；

Runtime pm：runtime pm 管理框架统一管理片内设备及相应 power domain 硬件操作。是在 power domain 提供的开关接口及设备运行时对应的时钟、复位、电的管理基础之上做的封装，设备会在 dts 指定具体的 power domain，设备驱动通过在 runtime_suspend、runtime_resume 实现对时钟、复位、电的管理，然后注册到 runtime pm 框架。runtime pm 为设备提供了 pm_runtime_get_xxx/pm_runtime_put_xxx 类接口，在使用设备前，先调用 get 接口，让设备退出低功耗，在使用完设备后调用 put 接口，让设备进入低功耗，当 power domain 下所有的设备都进低功耗后，就可以关闭 power domain，当 power domain 下有一个设备需要工作时，就要打开 power domain；

Device dvfs：device dvfs 管理框架统一管理设备的频率及电压调整，利用 opp framework 提供的接口，根据设备驱动给定的频率进行调频调压，跟 cpu dvfs 很类似；

Sleep：系统休眠唤醒能在系统不被使用的时候，进入一个功耗很低的状态，芯片功耗会很低，ddr 进入自刷新，并且在系统被使用的时候，能够快速地恢复。休眠唤醒的过程中会调用 device 提供的 system suspend、resume 函数，调用 power domain 提供的 suspend、resume 函数，会关闭、打开 cpuidle/cpuhotplug 功能；

Reboot/poweroff：系统重启/关机，重启和关机两个功能关系很紧密，可以简单地理解重启就是先关机再开机。系统重启能在系统出现异常后，或者在系统升级后，进行系统的复位。系统关机一般是用户不再使用机器，或者系统检测到用户不再使用或者可预见的时间不再使用，将系统断电的操作。系统重启和系统关机都是由 reboot 系统调用实现的，只是相应的参数不一。Service 发起重启和关机后，事件会给到 init 进程来处理。Init 会保存一些数据，停止所有 srvice，杀死普通进程，最后调用系统调用 reboot 进行重启或者关机；

Cpu idle: cpu idle 管理框架会根据当前一段时间 cpu 的空闲状态及接下来空闲时间的预估，选择进入相应 idle 等级，不同的等级功耗收益不同，进出时间也不一样，一般功耗收益低，但对应着比较少的退出时间；

Cpu dvfs：cpu dvfs 管理框架提供了不同 governor，用来控制 cpu 的频率、电压调整，这些 governor 有完全把 cpu dvfs 交给用户 service 的，用户 service 根据场景的不同进行相应的

频率、电压调整，也有根据负载进行动态频率、电压调整的；

Cpu hotplug：cpu hotplug 管理框架提供了一种机制给到 service：当前的空闲 cpu loading 比较多时，会由 service 直接拔掉一些 cpu，当 cpu loading 的需求增加时，再由 service 将 cpu 插上；

Pm qos：pm qos（quality of service）服务质量是解决低功耗可能会降低性能的问题，它向驱动或者上层 service 提供了一套对性能约束要求的接口，在 cpu dvfs、cpu idle 等低功耗管理的时候，会检查性能的约束需求，当发现低功耗动作会影响到性能的时候，便不会进行该次的低功耗操作；

4.2 安卓功耗优化机制

从 Android 6.0（API 级别 23）开始，Android 引入了两项省电功能，通过管理应用在设备未连接至电源时的行为方式，帮助用户延长电池寿命。当用户长时间未使用设备时，低电耗模式会延迟应用的后台 CPU 和网络活动，从而降低耗电量。应用待机模式会延迟用户近期未与之交互的应用的后台网络活动。

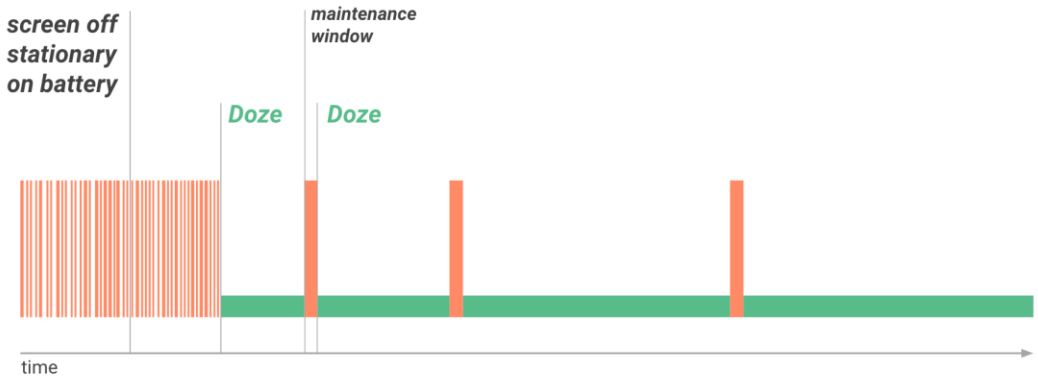
当设备处于低电耗模式时，应用对某些高耗电量资源的访问会延迟到维护期。电源管理限制中列出了具体的限制。

低电耗模式和应用待机模式管理在 Android 6.0 或更高版本上运行的所有应用的行为，无论它们是否专用于 API 级别 23。为确保用户获得最佳体验，请在低电耗模式和应用待机模式下测试您的应用，并对您的代码进行必要的调整。下面几部分提供了详细信息。

4.2.1 doze 模式

如果用户未插接设备的电源，在屏幕关闭的情况下，让设备在一段时间内保持不活动状态，那么设备就会进入低电耗模式。在低电耗模式下，系统会尝试通过限制应用访问占用大量网络和 CPU 资源的服务来节省电量。它还会阻止应用访问网络，并延迟其作业、同步和标准闹钟。

系统会定期退出低电耗模式一小段时间，让应用完成其延迟的活动。在此维护期内，系统会运行所有待处理的同步、作业和闹钟，并允许应用访问网络。



doze 耗模式限制

在低电耗模式下，您的应用会受到以下限制：

- 停访问网络。
- 系统忽略唤醒锁定。
- 标准 AlarmManager 闹钟（包括 setExact() 和 setWindow()）推迟到下一个维护期。

如果您需要设置在设备处于低电耗模式时触发的闹钟，请使用 setAndAllowWhileIdle() 或 setExactAndAllowWhileIdle()。

使用 setAlarmClock() 设置的闹钟将继续正常触发，系统会在这些闹钟触发之前不久退出低电耗模式。

- 系统不执行 WLAN 扫描。
- 系统不允许运行同步适配器。
- 系统不允许运行 JobScheduler。

4.2.2 应用待机模式

应用待机模式允许系统判定应用在用户未主动使用它时是否处于闲置状态。当用户有一段时间未触摸应用时，系统便会作出此判定，以下条件均不适用：

- 用户明确启动应用。
- 应用当前有一个进程在前台运行（作为活动或前台服务，或者正在由其他活动或前台服务使用）。

注意：您只能将前台服务用于用户希望系统立即执行或不中断的任务。 此类情况包括将照片上传到社交媒体，或者即使在音乐播放器应用不在前台运行时也能播放音乐。您不应该只是为了阻止系统判定您的应用处于闲置状态而启动前台服务。

- 应用生成用户可在锁定屏幕或通知栏中看到的通知。
- 应用是正在使用中的设备管理应用（例如设备政策控制器）。虽然设备管理应用通常在后台运行，但永远不会进入应用待机模式，因为它们必须保持可用性，以便随时从服务器接收策略。

当用户将设备插入电源时，系统会从待机状态释放应用，允许它们自由访问网络并执行任何待处理的作业和同步。如果设备长时间处于闲置状态，系统将允许闲置应用访问网络，频率大约每天一次。

4.2.3 省电模式

省电模式是系统里的一个开关，开启后会降低设备的性能，限制后台应用的活动和数据同步（如邮件接收），并限制 GPS 访问。

用户可以主动开启省电模式，也可以设置电量少于一定程度是自动开启省电模式。

一般 Android 设备厂商的定制系统，都会定制省电模式，只允许必要的 app 运行，并断开数据连接等。

设备连接到充电器的时候，一般情况下，系统会自动退出省电模式（定制系统可能行为会不一样）。

5 功耗调试

暂无

5.1 灭屏功耗分析

5.1.1 基于高通平台的灭屏功耗分析

暂无

(1) 飞行模式待机底电流是否正常

飞行模式待机底电流是待机功耗的基础，也是功耗调试的起点。当项目的硬件配置确定之后，可以根据高通发布的对应平台的功耗数据，加上项目配置的差异导致的差值（如 RAM 容量增加 1G，待机底电流会有 0.6mA 左右的增加），计算出来一个理论值，作为判定是否该项指标是否正常的参考。

(2) 确认网络因素

如果飞行模式待机底电流已正常，但注册网络条件下平均电流仍然偏大，则需要确认网络环境是否正常。当网络信号比较弱时，射频需要更大的发射功率，从电流波形上能看出较高的电流脉冲。或者出现网络信号变化、小区切换、掉网、搜网等情况时，电流波形上能看出明显的不规则变化。要确认是网络原因还是手机自身原因，需要在网络信号稳定的现网环境下测试，必要时可以在仪表下进行对比测试。

(3) CPU 是否可休眠

最简单和常用的方法是通过查看内核 log 来确认，如 log 中检测到如下的关键信息时，说明 CPU 可以进入休眠:

```
CPU0:msm_cpu_pm_enter_sleep mode:3 during suspend
```

(4) 上层是否有 wakelock 未释放

如果 CPU 不休眠，原因则可能是上层或者内核有 wakelock 未释放导致。可以先查看上层是否有 wakelock。可以使用 dumpsys power 命令来查看，如下截图中，就是有两个 wakelock 没有释放

```
Wake Locks: size=2
  mLock:101759918 PARTIAL_WAKE_LOCK                '*sync*/com.evernote.evernoteprovider/com.
 (uid=1000, pid=1529, ws=WorkSource{10114})
  mLock:81993039 PARTIAL_WAKE_LOCK               'SyncService' (uid=10114, pid=875, ws=null)
```

(5) 确认是哪个应用

如果是上层有 wakelock 不释放，需要进一步查明是那个应用持有的 wakelock，可以使 ps 命令来查看，如下截图中，是 com.evernote(印象笔记)持有 wakelock 导致系统不休眠。

```
shell@HWVNS-Q:/ $ ps | grep 875
u0_a114   875   590   1845896 140620 SyS_epoll_ 0000000000 S com.evernote
```

确认异常持有 wakelock 的模块后，需要找对应模块的负责人进一步分析该应用持锁不释放的原因。

(6) 内核是否有 wakelock

如果上层没有 wakelock，则需要确认内核中的 wakelock，可以使用命令

cat sys/kernel/debug/wakeup_sources 来查看

(7) 确认是哪个模块

active_since 这列对应的值不为 0，说明此刻内核中仍有 wakelock 没有释放。如下表中，

由于手机插着 USB 连电脑执行 adb 命令，所以跟 USB 相关的两个 wakelock 处在不释放状态，系统不能进入休眠。在分析问题的时候需要找除 USB 相关的这两个 wakelock 之外，还有哪些 wakelock 对应的 active_since 项不为 0，即为问题根源。

name	active_count	event_count	wakeup_count	expire_count	acti
qpnp-smbcharger-19	95	95	3	0	174
msm_otg	4	4	0	0	174

(8) 系统是否 vdd_min

如果 CPU 休眠，需要进一步查看整个系统是否能进入 vdd_min，这是系统的真正的低功耗状态。可以使用 adb 命令 cat sys/kernel/debug/rpm_stats 来查看，如下截图，vimin 对应的 count 值不为 0，说明系统是正常进入过 vdd_min 状态的。

(9) 是否有 clk 未正常关闭

CPU 已正常进休眠，而系统没有进 vdd_min 的话，需排查 APP 子系统中是否有 clk 在

待机时未正常关闭。

原生代码中 clk 的 debug 开关是没有打开的，要查看是否有 clk 未正常关闭，可以先使用 adb 命令 echo 1 > /sys/kernel/debug/clk/debug_suspend 打开开关，手动按键亮灭屏，每次等待一段时间，使系统休眠唤醒几次，抓出 dmesg，查看是否有在正常休眠时应该 disable 的 clk 仍处在 enable 状态

```
CPU0:msm_cpu_pm_enter_sleep mode:3 during suspend
Enabled clocks:
  xo_a_clk_src:2:2 [19200000]
  pcnoc_a_clk:1:1 [19200000]
  bimc_clk:1:1 [9568256]
  bimc_msmbus_clk:1:1 [9568256] -> bimc_clk:1:1 [9568256]
  bimc_a_clk:1:1 [99942400]
  bimc_msmbus_a_clk:1:1 [99942400] -> bimc_a_clk:1:1 [99942400]
  pcnoc_keepalive_a_clk:1:1 [19200000] -> pcnoc_a_clk:1:1 [19200000]
  qdss_clk:16:16 [1000]
  xo_a_clk:4:3 [19200000] -> xo_a_clk_src:2:2 [19200000]
  gpll0_ao:1:0 [800000000] -> xo_a_clk:4:3 [19200000] -> xo_a_clk_src:2:2 [19200000]
  a53ss_c1_pll:1:1 [1459200000, 2] -> xo_a_clk:4:3 [19200000] -> xo_a_clk_src:2:2 [192
  a53ss_cci_pll:1:1 [595200000, 1] -> xo_a_clk:4:3 [19200000] -> xo_a_clk_src:2:2 [192
  apss_ahb_clk_src:1:1 [19200000] -> xo_a_clk:4:3 [19200000] -> xo_a_clk_src:2:2 [1920
  gcc_boot_rom_ahb_clk:1:1 [0]
  gcc_mss_cfg_ahb_clk:1:1 [0]
  gcc_mss_q6_bimc_axi_clk:1:1 [0]
  gcc_usb2a_phy_sleep_clk:1:1 [0]
  a53ssmux_lc:1:0 [800000000, 3] -> gpll0_ao:1:0 [800000000] -> xo_a_clk:4:3 [19200000
  a53ssmux_bc:2:1 [1459200000, 6] -> a53ss_c1_pll:1:1 [1459200000, 2] -> xo_a_clk:4:3
  a53ssmux_cci:2:1 [595200000, 4] -> a53ss_cci_pll:1:1 [595200000, 1] -> xo_a_clk:4:3
Enabled clock count: 20
```

(10) 确认 clk 未正常关闭的模块

根据未关闭的 clk 的信息查找是哪个内核模块的 clk 在待机时没有正常关闭，走查相关的代码逻辑，找到问题原因。

(11) 是否有子系统不休眠

在 CPU 已休眠，clk 已正常关闭的情况下，如果系统仍没有进入 vdd_min 状态，则比较有可能是除了 APP 子系统之外，其他的 MSS、LPASS、WCSS 三个子系统中的一个或多个没有进入休眠，在投票时不允许 XO 关闭导致。要确认这种情况，需要抓到此时的 RAMDUMP，并对其进行解析，获取各个子系统的休眠状态。

①抓取 RAMDUMP，需要先触发系统死机，然后使用 QPST 来抓取 QPST。

②因为要保持 APPS 的休眠状态，所以不能连接 USB，有两种方式可以触发死机，一是拉低 PS_HOLD 200mS 以内的方式，另一种是通过 adb 命令修改寄存器值的方式修改音量下键功能，用音量下键来触发死机，抓取 RAMDUMP。

```
C:\Users\Administrator>adb shell
root@A3500:/ # cd /sys/kernel/debug/spmi/spmi-0
cd /sys/kernel/debug/spmi/spmi-0
root@A3500:/sys/kernel/debug/spmi/spmi-0 # echo 0x844 > address
echo 0x844 > address
root@A3500:/sys/kernel/debug/spmi/spmi-0 # echo 4 > count
echo 4 > count
root@A3500:/sys/kernel/debug/spmi/spmi-0 # cat data
cat data
00840 -- -- -- -- 0F 07 04 00
root@A3500:/sys/kernel/debug/spmi/spmi-0 # echo 0x00 0x00 0x01 0x00 > data
echo 0x00 0x00 0x01 0x00 > data
root@A3500:/sys/kernel/debug/spmi/spmi-0 # cat data
cat data
00840 -- -- -- -- 00 00 01 00
root@A3500:/sys/kernel/debug/spmi/spmi-0 # echo 0x00 0x00 0x01 0x80 > data
echo 0x00 0x00 0x01 0x80 > data
root@A3500:/sys/kernel/debug/spmi/spmi-0 # cat data
cat data
00840 -- -- -- -- 00 00 01 80
root@A3500:/sys/kernel/debug/spmi/spmi-0 #
```

③解析 RAMDUMP，需要使用 python 和高通 modem 代码中自带的 hansei 工具，解析后可以看到是哪个子系统不休眠导致 XO 无法关闭。

```
npa_resource (name: "/xo/cxo") (handle: 0x199060) (units: Enable) (resource max: 1) (active max: 1) (active state: 1)
        npa_client (name: MPSS) (handle: 0x19a850) (resource: 0x199060) (type: NPA_CLIENT_LIMIT_MAX) (request: 1)
        npa_client (name: MPSS) (handle: 0x19a810) (resource: 0x199060) (type: NPA_CLIENT_REQUIRED) (request: 1)
        npa_client (name: WCSS) (handle: 0x19a728) (resource: 0x199060) (type: NPA_CLIENT_LIMIT_MAX) (request: 1)
        npa_client (name: WCSS) (handle: 0x19a6e8) (resource: 0x199060) (type: NPA_CLIENT_REQUIRED) (request: 0)
        npa_client (name: LPASS) (handle: 0x19a2e0) (resource: 0x199060) (type: NPA_CLIENT_LIMIT_MAX) (request: 1)
        npa_client (name: LPASS) (handle: 0x19a2a0) (resource: 0x199060) (type: NPA_CLIENT_REQUIRED) (request: 0)
        npa_client (name: APSS) (handle: 0x199278) (resource: 0x199060) (type: NPA_CLIENT_REQUIRED) (request: 0)
```

(12) 确认各子系统不休眠的原因

①如果是 MSS，即 modem 子系统不休眠，通常高通代码 BUG 导致此情况的可能性较小，需重点排查 NV，看是否是某些 NV 项设置不正确导致。

②如果是 WCNSS，即 WIFI 子系统不休眠，则需要 WIFI 模块进一步排查，这种情况遇到的比较少。

③如果是 LPASS，即低功耗音频子系统不休眠，则需要排查 Audio 模块和 sensor 模块，看近期哪些模块代码有修改，需重点排查。

(13) 确认外设和主板漏电

如果系统已进入 vdd_min，但整机待机底电流仍然偏大，则比较可能是外设或者主板上漏电，需要使用硬件的方法进行排查，比如，拆器件法，把 TP、LCD、CAMERA、指纹等外设逐个拆除，直至剩下一个仅包括 AP、Memory 和 PMIC 的最小系统，看在拆除哪个器件的时候电流能够有下降，则进一步排查这个模块；或者，灌电流法，分解各个供电支路上消耗的电流，看哪个供电支路上电流异常，则针对它对其供电的模块重点排查。

(14) 主要唤醒源是 modem

如果系统能进入 vdd_min，待机底电流也正常，而用户仍然反馈耗电快，这种情况在用户的真实使用过程中比较常见。这种情况一般是由于系统唤醒频繁导致。

其中比较常见的一类唤醒源是 modem，modem 对应的 irq 号一般是 57 和 58，如下 log

```
 gic_show_resume_irq: 57 triggered qcom,smd-modem
 gic_show_resume_irq: 57 triggered qcom,smd-modem
] gic_show_resume_irq: 57 triggered qcom,smd-modem
```

(15) 确认 modem 唤醒系统原因

Modem 唤醒系统，可能是异常的，也可能是正常的。

①如果是由于网络信号变化导致的，并非手机自身原因的话，就不是异常。

```
RILJ : [3831]> SIGNAL_STRENGTH [SUB1]
NetworkController.MobileSignalController(slotId:0 subId:1): onSignalStrengthsChanged signalStrength=SignalStrength: 99 0 -120
NetworkController.MobileSignalController(slotId:0 subId:1): updateVolte: slotId: 0 subId: 1 iconId: 0
```

②如果是掉网、搜网等导致的，这种情况需进一步确认是手机自身原因还是网络环境原因。

```
RILJ    : [4446]< VOICE_REGISTRATION_STATE {1, a160, 00ab8029, 14, null, null, null, 0, null, null
```

③如果是手机中的应用与服务器侧有数据交互导致，需要使用 Wireshark 来分析 TCP DUMP，查看具体是哪些应用在与服务器交互，这些交互的频率是否正常。

(16) 主要唤醒源是 WLAN

用户如果连接的是移动现网，则可能 modem 唤醒系统较多。如果用户也连接了 WIFI，则可能有 WLAN 频繁唤醒系统，，WLAN 对应的 irq 号一般是 178，如下截图。

```
gic_show_resume_irq: 178 triggered wcnss_wlan
gic_show_resume_irq: 178 triggered wcnss_wlan
gic_show_resume_irq: 178 triggered wcnss_wlan
gic_show_resume_irq: 178 triggered wcnss_wlan
gic_show_resume_irq: 178 triggered wcnss_wlan
gic_show_resume_irq: 178 triggered wcnss_wlan
```

(17) 确认 WLAN 唤醒系统原因

与 modem 唤醒系统相似，WLAN 唤醒系统可能是异常的，也可能是正常的。

①可能是手机接入的 AP 本身有问题，这种情况就不是手机的问题。可以更换 AP 来确认。

②也可能是手机中应用与服务器侧有数据交互导致，与 modem 唤醒相同，这种情况也需要使用 Wireshark 来分析 TCP DUMP 来查看，具体是哪些应用在与服务器交互，这些交互的频率是否正常

(18) 主要唤醒源是 alarm，

```
__qpnpint_handle_irq: [AUDIO_W] 356 triggered [0x0, 0x61,0x1] qpnp_rtc_alarm
__qpnpint_handle_irq: [AUDIO_W] 356 triggered [0x0, 0x61,0x1] qpnp_rtc_alarm
__qpnpint_handle_irq: [AUDIO_W] 356 triggered [0x0, 0x61,0x1] qpnp_rtc_alarm
__qpnpint_handle_irq: [AUDIO_W] 356 triggered [0x0, 0x61,0x1] qpnp_rtc_alarm
__qpnpint_handle_irq: [AUDIO_W] 356 triggered [0x0, 0x61,0x1] qpnp_rtc_alarm
```

(19) 确认 alarm 设置与对齐唤醒是否正常

主要关注类型为 type0 和 type2 的定时器对系统的唤醒。

```
AlarmManager: Alarm triggering (type 0 or 2): [Alarm{f1503e type 0 when 1469368341821 android}]
AlarmManager: Alarm triggering (type 0 or 2): [Alarm{5ea3949 type 0 when 1469368571969 android}]
AlarmManager: Alarm triggering (type 0 or 2): [Alarm{608dfed type 2 when 205920455 com.android.bluetooth}]
AlarmManager: Alarm triggering (type 0 or 2): [Alarm{c26d35 type 0 when 1469368800000 android}]
AlarmManager: Alarm triggering (type 0 or 2): [Alarm{9cf1fe6 type 0 when 1469368802134 android}]
AlarmManager: Alarm triggering (type 0 or 2): [Alarm{a99b71f type 2 when 206031788 android}]
```

Alarm 的唤醒也同样有正常的唤醒和异常的唤醒

①如果是 com.android.phone 每隔 6 分钟一次对系统的唤醒，是正常的，这个是数据业务 DoRecover 的定时器，发现只有收包没有发包时，获取 Data_Call_list 用于查看数据业务当前状态，状态不对，则尝试重新激活数据业务。

②开机之后一般会有 calendar 相关的几次唤醒，这个一般也是正常的。

③很多应用都会有周期性对系统的唤醒，通过设置 alarm 来实现，如 QQ、微信等缺省的唤

醒周期在 3 到 6 分钟，这类唤醒一般也是正常的。
④系统中如果已经导入了 alarm 对齐唤醒功能，而测试到的电流波形仍是杂乱的，log 中看到的 alarm 唤醒是无规律的，则需要检查一下对齐唤醒功能是否已生效，是否工作正常。
(20) 主要唤醒源是内核中断
除了 modem、WLAN 和 alarm 这三类唤醒源外，有时在内核 log 中会遇到其他中断源频繁唤醒系统的情况
(21) 确认唤醒原因
对内核中断频繁唤醒系统的情况，首先是要找到该中断对应的模块。然后针对这些模块进行分析，常见的原因可能有某些模块代码逻辑存在缺陷、某些器件损坏、操作方法不合标准等。

5.1.2 基于 RK 平台的补充

带有 trust 的 SoC 平台，系统待机（system suspend）的工作都在 trust 中完成。因为各个平台的 trust 对于系统待机实现各不相同，所以 不同平台之间的待机配置选项/方法没有任何关联性和参考性，本文档仅适用于 RK3588 平台

系统待机流程一般会有如下操作：关闭 power domain、模块 IP、时钟、PLL、ddr 进入自刷新、系统总线切到低速时钟（24M 或 32K）、vdd_arm /vdd_log 断电、配置唤醒源等。

为了满足不同产品对待机模式的需求，目前都是通过 DTS 节点把相关配置在开机阶段传递给 trust。

DTS 节点

arch/arm64/boot/dts/rockchip/rk3588s.dtsi

```
rockchip_suspend: rockchip-suspend {
        compatible = "rockchip,pm-rk3588";
        status = "disabled";
        // 休眠 log 开关配置，0：关闭打印， 1：打开打印
        rockchip,sleep-debug-en = <0>;
         // 休眠配置
        rockchip,sleep-mode-config = <
                (0
                | RKPM_SLP_ARMOFF_LOGOFF
                | RKPM_SLP_PMU_PMUALIVE_32K
                | RKPM_SLP_PMU_DIS_OSC
                | RKPM_SLP_32K_EXT
                )
```

```
        >;
        // 唤醒源配置
        rockchip,wakeup-config = <
                (0
                | RKPM_GPIO_WKUP_EN
                )
        >;
};
```

需要根据具体产品对唤醒源的需求进行相关配置，比如 usb 唤醒，那休眠时就不能将 usb 的电源和时钟关闭，所以不能配置 RKPM_SLP_ARMOFF_LOGOFF、RKPM_SLP_PMU_DIS_OSC、RKPM_SLP_PMU_PMUALIVE_32K 等选项。

详见 RK3588 系统休眠配置

5.2 亮屏功耗分析

暂无

5.2.1 基础功耗

所有应用功耗问题，首先需要确认的是当前测试所用的硬件和软件状态下的基础功耗是否已正常。这个基础功耗，即飞行模式亮屏 idle 功耗（对灭屏应用场景，如语音通话、mp3 播放、流媒体下载等，基础功耗就是飞行模式灭屏 idle 功耗）。是指将手机设置为飞行模式（关闭 BT、WIFI、GPS），固定背光亮度和长时间亮屏（30 分钟或者屏幕不灭），测试手机在 home 界面 idle 状态下的功耗。

一、分解思路

如果基础功耗异常，则需对其进行分解，分解思路为：

亮屏 idle 功耗=屏幕背光功耗+屏幕模组功耗+系统 idle 功耗

测试方法为：

（1）测量亮屏 idle 功耗

（2）使用 adb 命令 echo 0 > /sys/class/leds/lcd-backlight/brightness，关闭屏幕背光，测量灭屏 idle 功耗

（3）拆掉屏幕，测量无屏 idle 功耗，即系统 idle 功耗

测量过程中结果记录如下：

测试项	亮屏 idle 功耗	灭屏 idle 功耗	去屏 idle 功耗
测得值	I1	I2	I3

功耗分解结果如下：

功耗项	屏幕背光耗电	屏幕模组耗电	系统 idle 耗电
分解值	I1-I2	I2-I3	I3

二、分析调试

与同平台对比机相比，针对分解出的各部分耗电中的异常项进行调试：

（1）如果屏幕背光耗电异常，则需检查背光控制相关策略，如 CABC 等功能是否已开启、参数是否已调至最优。

（2）如果屏幕模组耗电异常，则需检查 LCM 供电电路设计是否最优，屏幕模组是否有异常等。

（3）如果系统 idle 功耗异常，则需对 CPU 资源占用等系统负载进行分析，找到异常的部分，相应地进行处理。

（4）需要注意的是，如果机器使用的是 video 模式的屏幕，测到的电流 I3 会比 cmd 模式屏幕的机器大几十 mA。这部分耗电是由于屏幕自身原因导致，但耗电却会体现在系统上，是已知的正常情况。

通常情况下，基础功耗分析与调试这部分工作应该在项目早期功耗调试之初就完成，以达到平台理论值水平为准。后期开发过程中及用户使用过程中再遇到因基础功耗异常导致的功耗问题，一般比较可能是修改引入或者器件损坏等。

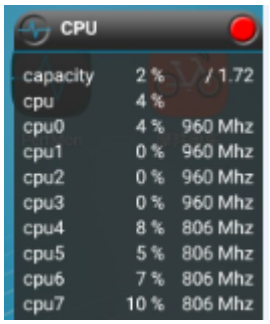
5.2.2 系统负载分析

（1）系统负载分析

在确认基础功耗正常后，如果应用场景下功耗仍然偏大，则需要对系统负载情况进行分析，主要包括对 CPU 运行状态、GPU 运行状态、系统 CLK 状态等进行分析。

（2） CPU 运行状态

先使用性能监视器 PerfMon 初步查看是否有 CPU 资源整体占比偏高、运行中的 CPU 核数较多、频率较高的现象



如果是灭屏状态下运行的业务，并且系统本身支持在此应用场景下 CPU 进休眠，如语音通话、mp3 播放、FM 播放等，则需要通过 log 确认 CPU 是否可以进入休眠状态。

如果确认 CPU 资源占用偏高，则需进一步确认是哪些应用导致的。简单的方法，使用 top 命令即可，也可以使用 PowerTop 工具抓取更多详细一些的信息

①Top 命令执行的方法和结果如下：

```
shell@HWVNS-Q:/ $ top -m 5



User 21%, System 8%, IOW 1%, IRQ 0%
User 531 + Nice 13 + Sys 221 + Idle 1677 + IOW 32 + IRQ 0 + SIRQ 16 = 2490

  PID PR CPU% S  #THR     VSS     RSS PCY UID      Name
30624  7   9% S    23 1809056K 114212K  tv u0_a18   com.huawei.camera
  613  7   3% S    31 287384K  17672K  fg camera   /system/bin/mm-qcamera-daemon
  794  7   3% S    60 386692K  26588K  tv media    /system/bin/mediaserver
 1533  5   2% S   188 2606620K 155192K  tv system   system_server
30529  4   2% S    26 1026796K  66436K  bg u0_a108  com.UCMobile:ppappstore



User 15%, System 6%, IOW 0%, IRQ 0%
User 369 + Nice 6 + Sys 162 + Idle 1955 + IOW 1 + IRQ 0 + SIRQ 2 = 2495

  PID PR CPU% S  #THR     VSS     RSS PCY UID      Name
  613  7   3% S    31 287384K  17672K  fg camera   /system/bin/mm-qcamera-daemon
  794  7   2% S    60 386692K  26592K  tv media    /system/bin/mediaserver
 1533  4   1% S   188 2606620K 155244K  tv system   system_server
30624  4   0% S    23 1809056K 113800K  tv u0_a18   com.huawei.camera
31218  1   0% R     1    4092K   1320K  fg shell    top
```

②PowerTop 工具抓到的信息更多一些，除了 Top 命令获取的信息外，还有一段时间内统计出来的 CPU 运行状态信息，如下所示

```
No msm_pm_stats available in procfs.            Enable CONFIG_MSM_IDLE_STATS in the kernel.
Cn              Avg residency
C0 (cpu running)         (27.3%)
C0        33.5ms ( 8.7%)
C1        98.9ms (58.7%)
C2       258.8ms ( 5.3%)
P-states (frequencies)
    400 Mhz    0.0%    0.0%    0.0%    0.0%    0.0%    0.0%
    691 Mhz   95.8%   95.8%   95.8%   95.8%    883 Mhz    99.8%   99.8%
    806 Mhz    0.0%    0.0%    0.0%    0.0%    941 Mhz     0.0%    0.0%
   1018 Mhz    0.0%    0.0%    0.0%    0.0%    998 Mhz     0.0%    0.0%
   1190 Mhz    0.0%    0.0%    0.0%    0.0%   1056 Mhz     0.0%    0.0%
   1306 Mhz    3.0%    3.0%    3.0%    3.0%   1114 Mhz     0.0%    0.0%
   1382 Mhz    0.0%    0.0%    0.0%    0.0%   1190 Mhz     0.0%    0.0%
   1402 Mhz    1.2%    1.2%    1.2%    1.2%   1248 Mhz     0.0%    0.0%
      0.0%    0.0%    0.0%    0.0%   1306 Mhz     0.0%    0.0%
      0.0%    0.0%    0.0%    0.0%   1382 Mhz     0.0%    0.0%
      0.0%    0.0%    0.0%    0.0%   1.62 Ghz     0.0%    0.0%
      0.0%    0.0%    0.0%    0.0%   1.75 Ghz     0.0%    0.0%
      0.0%    0.0%    0.0%    0.0%   1.81 Ghz     0.2%    0.2%
```

有时功耗的差别幅度不大，甚至看不出 CPU 占比的明显差别，从 PowerTop 中也看不出明显的差别，这时需要使用 Ftrace log 来进一步确认下

Comm	Pid	stime	stime%
cpu0/freq:1305600	0	624758	1.04
cpu0/freq:1401600	0	209880	0.35
cpu0/freq:691200	0	57301711	95.42
cpu1/freq:1305600	0	425870	0.71
cpu1/freq:691200	0	56529733	94.14
cpu1/freq:806400	0	1217259	2.03
irq39:arch_mem_timer hanc	0	465165	0.77
softirq1:TIMER	0	259674	0.43
softirq7:SCHED	0	137354	0.23
rcu_preempt	7	148574	0.25
kworker/u16:6	217	166664	0.28
msm-core:sampli	328	131545	0.22
surfaceflinger	452	235095	0.39
Binder_1	642	179488	0.30
Binder_2	644	177222	0.30
EventThread	749	918564	1.53
Binder_3	1441	181670	0.30
Binder_4	1442	179453	0.30
ndroid.systemui	4221	191525	0.32
RenderThread	4451	254356	0.42
droid.launcher3	4785	2308577	3.84
Binder_5	7617	176598	0.29
kworker/0:4	7764	120514	0.20

（3）系统 CLK 分析

抓到问题出现时系统中 CLK 的状态信息，查看各分支 CLK 的频率，与正常的机器或者版本进行对比，看是否有某些支路 CLK 频率偏高的情况，然后有针对性地进行分析。

（4）GPU 运行状态分析

一些与显示功能关系密切的应用场景，如视频播放等，除 CPU 外，还需对比一些 GPU 运行的状态。

简单操作的话，使用 adb 命令查看相关的频率与 GPU 占比即可，方法如：

```
adb shell cat /sys/kernel/debug/clk/gcc_oxili_gfx3d_clk/rate
adb shell cat /sys/kernel/debug/clk/bimc_clk/measure
adb shell cat sys/kernel/debug/clk/snoc_clk/measure
adb shell cat /sys/class/kgsl/kgsl-3d0/gpubusy
```

也可以写成脚本，将获取的 GPU 频率及负载情况保存出来进行分析，更加直观。

5.2.3 射频耗电排查

与网络相关的应用场景，语音业务如语音通话等，数据业务如流媒体下载、网页应用等，除了基础功耗外，还需要排除射频功耗的影响。可能有射频匹配未优化好、功率偏大等，需要射频模块进一步确认处理。

这部分工作通常也是在项目前期基础功耗调试阶段就已完成，以达到平台理论值水平为准。

如果项目后续开发中或者用户使用过程中有遇到网络相关应用的功耗问题，需将射频功耗是否正常作为一个排查项。

5.2.4 网络因素排查

网络因素对应用功耗的影响，主要在于移动现网信号较弱时，射频需要更大的发射功率，从而导致整机功耗偏大，这种情况下，需要换到网络信号较强且稳定的地方进行测试（通常情况下测试环境中的移动现网信号强度在-60dB 到-90dB 范围内时可认为网络环境较好，如果小于-100dB，则信号比较差了，一般会引起功耗偏大），或者在仪表下进行对比测试，来确认是网络环境原因还是手机自身原因。

5.3 Battery Historian

暂无

5.3.1 Battery Historian 安装与配置

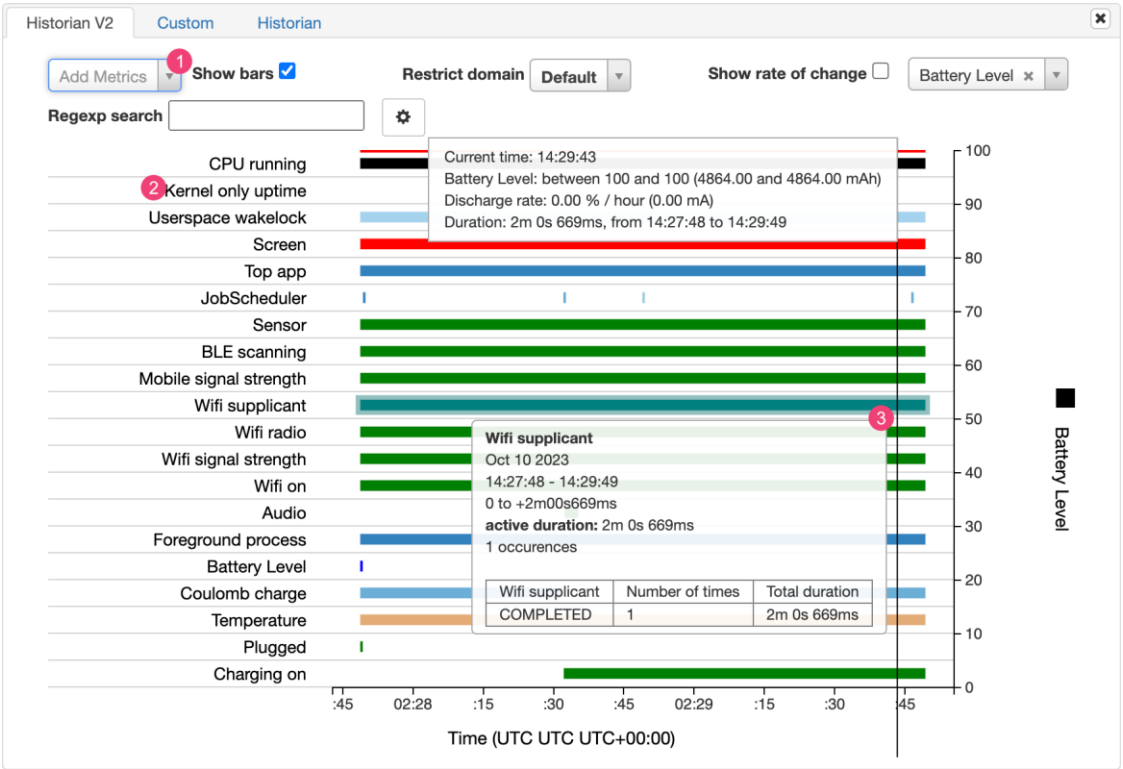
安装配置过程省略，具体见网络。

5.3.2 使用 Battery Historian 分析耗电情况

使用 Battery Historian 图表查看数据

Battery Historian 图表会显示一段时间内与电源相关的事件。

当系统组件由于处于活动状态而正在消耗电池电量时，每行都会显示一个彩色条形段。图表不会显示该组件使用的电量，仅表示相关应用处于活动状态。图表按类别进行整理，并以一个条形显示每个类别随时间的变化，如图表的 X 轴所示。



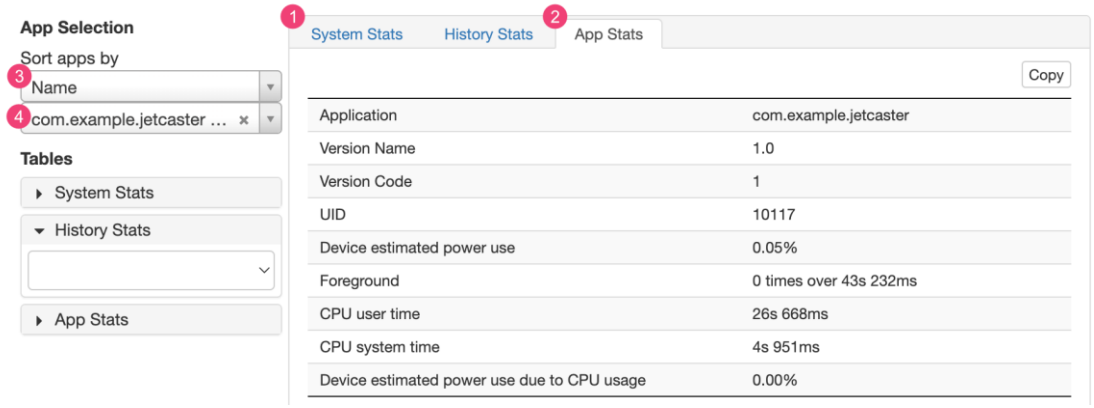
从下拉列表中添加其他指标。

将指针悬停在指标名称上可查看有关每个指标的更多信息，包括图表中使用的颜色对应的键。

将指针悬停在某个条形上可查看有关该指标的更多详细信息以及时间轴上特定点的电池统计信息。

其他 Batterystats 输出

在 Battery Historian 图表之后的统计信息部分中，您可以查看 batterystats.txt 文件中的其他信息。



System Stats 标签页（标示为 1）包含系统范围的统计信息，例如电池信号电平和屏幕亮度。此信息可全面反映设备的具体情况。这对于确保没有任何外部事件影响您的测试尤为有用。App Stats 标签页（标示为 2）包含有关特定应用的信息。请使用 App Selection 窗格中的 Sort apps by 下拉列表（标示为 3）对应用列表进行排序。您可以使用应用下拉列表（标示为 4）选择特定应用来查看统计信息。

6 参考文档

【Android】 android suspend/resume 总结（1）

https://blog.csdn.net/weixin_41028159/article/details/127859221

SystemSuspend 服务

https://source.android.google.cn/docs/core/power/systemsuspend?hl=zh-cn

Linux 电源管理(6)_Generic PM 之 Suspend 功能

http://www.wowotech.net/pm_subsystem/suspend_and_resume.html

几种休眠机制对比

https://blog.csdn.net/wlwl0071986/article/details/42672591

一文搞懂 Linux 电源管理（合集）

https://zhuanlan.zhihu.com/p/580754972

Doze 模式和应用待机模式

https://developer.android.google.cn/training/monitoring-device-state/doze-standby?hl=zh-cn

Rockchip RK3588 kernel dts 解析之系统休眠配置 rockchip_suspend

https://blog.csdn.net/weixin_43245753/article/details/126981052

Battery Historian

https://developer.android.google.cn/topic/performance/power/setup-battery-historian?hl=zh-cn