# 嵌入式Linux系统中的电源管理研究

## 摘要

嵌入式Linux系统中的电源管理研究 1

摘要 1

第一章 绪论 2

1.1 研究背景 2

1.2 研究现状 2

1.3 本文研究工作 2

1.4 论文结构 2

第二章 嵌入式Linux系统电源管理技术分析 2

2.1 文献综述 2

2.2 存在问题 2

2.3 研究路线 2

2.4 相关理论与技术简介 2

第三章 嵌入式Linux系统电源管理的研究与设计 3

3.1嵌入式Linux系统电源管理架构 3

3.1.1 电源管理架构概述 3

3.1.2 Linux核心电源管理 5

3.1.3 CPUIdle模块设计 8

3.1.4 CPUFreq模块设计 8

3.1.5 CPU Hotplug模块设计 8

3.2设备驱动的Runtime PM设计 9

3.2.1通用设计思想 9

3.2.2 Atmel MXT1664触摸屏控制器驱动的Runtime PM设计 12

**3.2.2.1** **MXT1664的硬件特性** 12

**3.2.2.2** **软件设计目标** 13

3.2.2.3 **驱动流程** 13

第四章 嵌入式Linux系统电源管理的实现 15

第五章 总结与展望 15

参考文献 15

## 第一章 绪论

### 1.1 研究背景

### 1.2 研究现状

### 1.3 本文研究工作

### 1.4 论文结构

## 第二章 嵌入式Linux系统电源管理技术分析

### 2.1 文献综述

### 2.2 存在问题

### 2.3 研究路线

### 2.4 相关理论与技术简介

## 第三章 嵌入式Linux系统电源管理的研究与设计

### 3.1嵌入式Linux系统电源管理架构

#### 3.1.1 电源管理架构概述

嵌入式Linux系统的电源管理架构是在通用的Linux内核电源管理基础之上加入了用户空间电源管理机制而成的。整体架构可以用下图表示。其中，从下往上依次为硬件层，Linux内核层和用户空间层。而Linux内核层又可以根据与硬件的依赖关系分为硬件SoC平台相关和平台通用两层。

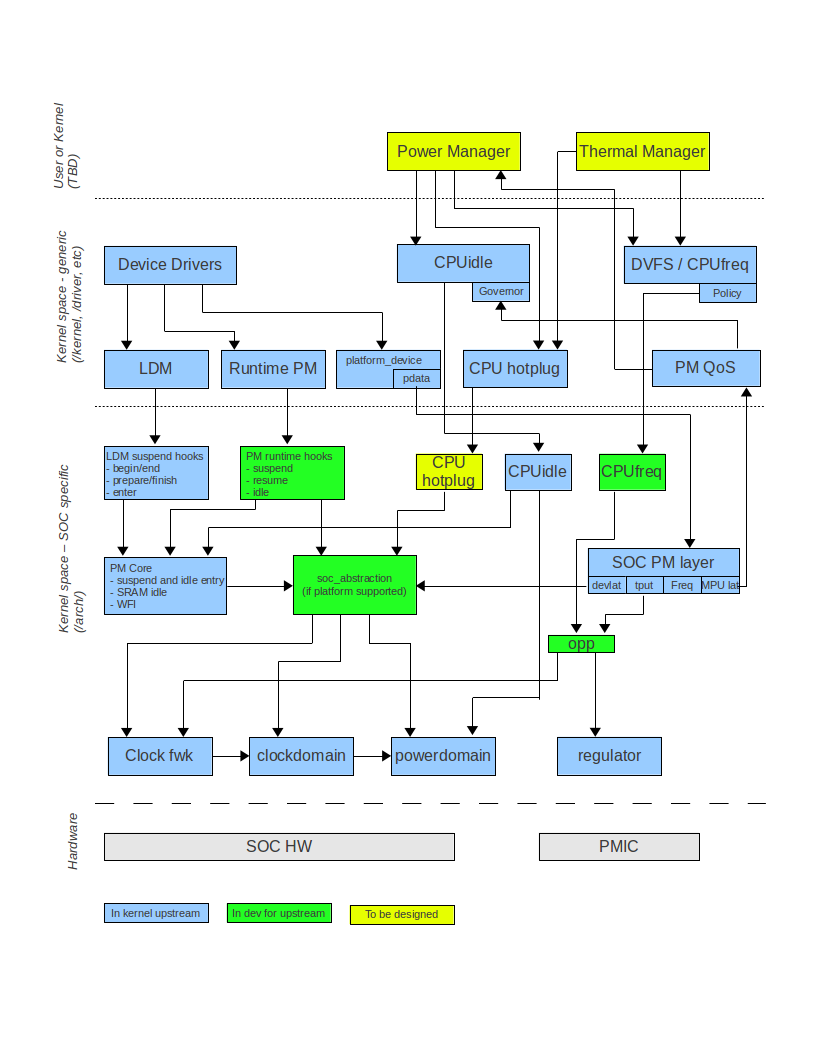
图 3 - 1 Linux系统电源管理架构[]

Figure 3 - 1 Linux system power management architecture[]

硬件层通常指的是作为嵌入式Linux系统硬件核心的SoC硬件以及与之相配套的电源管理芯片PMIC。SoC通常在硬件上提供了一系列电源管理特性比如处理器核的低功耗状态、处理器核心和外设总线时钟频率的设置寄存器、门控时钟（clock gating）等。PMIC通常由SoC硬件以及Linux内核中的PMIC驱动协调控制实现系统中各组件供电的打开与关闭以及电压的调整。由此可以实现处理器核心在低功耗状态下相应的电压调整。除此之外，在现代SoC集成度越来越高的趋势下，越来越多的处理器核如无线通信基带处理器，数字信号处理器，WiFi／蓝牙／GPS通信处理器等都集成到了同一个SoC系统内。因此SoC内也出现了专门负责资源管理到处理器核心来统一调度SoC内不同处理器核的电源管理状态。在HP Pro Slate 8项目中，所选取的SoC平台高通APQ8074AB中就包含了这样的一颗处理器核，称为Resource and Powere Manager (RPM)核。

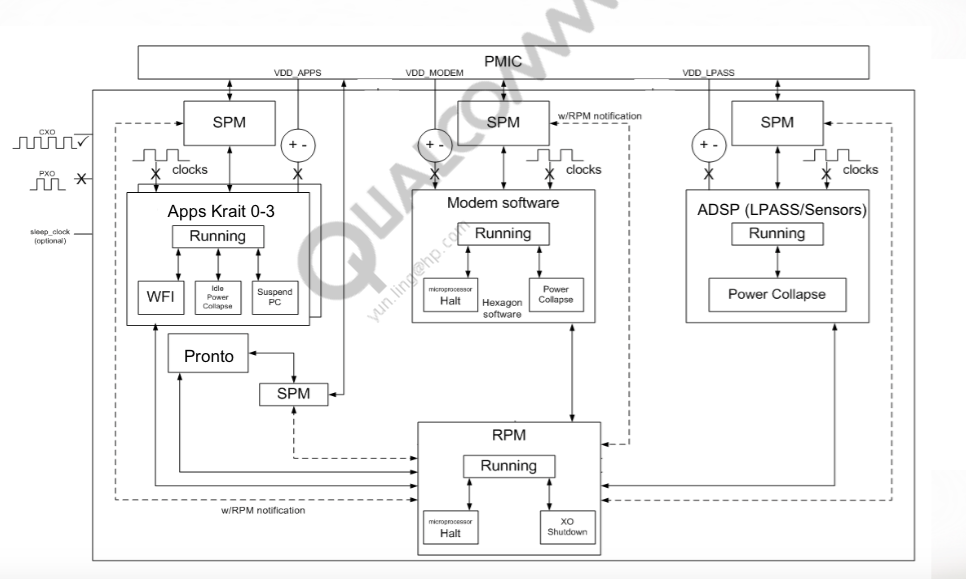


图 3 - 2 APQ8074AB 电源管理架构[]

Figure 3 - 2 APQ8074AB Power Management Architecture[]

从图中可以看到，PM8941 PMIC和APQ8074AB通过System Power Management Interface(SPMI)接口连接，通过PMIC的时钟和电源电路提供了SoC工作所需的时钟信号和处理器核心以及片上集成的DDR内存控制器、外部eMMC控制器、I2C总线控制器等主要组件的电源输入。当APQ8074AB里的Krate应用处理器核心的工作频率在正常工作范围内动态变化时，对应的PM8941里的Power Regulaor会根据驱动预先的设定值相应调整处理器核心的电压以达到功率的控制；当SoC除RPM以外的处理器核们进入深度休眠状态时，大部分的电源和系统主要的19.2MHz的晶振输入信号将被关闭，只保留RPM用以唤醒处理器核的电源和一个低频率的32.768K的时钟信号以达到整个SoC的最低功耗。除此之外，APQ8074AB上的时钟控制寄存器和PM8941的电源电路控制寄存器可用于在系统运行时由Linux内核的电源管理框架来实现设备的时钟和电源的控制，这些是实现运行时电源管理(Runtime PM)和系统休眠和唤醒的核心电源管理(Linux PM Core)的基础。APQ8074AB还提供了如WFI(Wait for Interrupt)这样的低功耗状态用以支持Linux内核的CPUIdle机制，实现处理器空闲情况下的低功耗管理。

在硬件层之上的就是嵌入式Linux系统的内核。在目前最新的3.x版本上它包含了实现电源管理所需要的时钟框架和电源驱动框架(Linux voltage and current regulator framework)。在此基础之上，它提供了系统休眠与唤醒机制，即Linux核心电源管理、基于Runtime PM框架的设备动态电源管理和一系列针对处理器核心的动态电源管理机制，如CPUIdle，CPUFreq和CPU Hotplug。Linux内核中的电源管理架构设计遵从Linux内核的一般设计准则，即分为上层硬件平台无关和底层硬件平台相关这两层。通常硬件平台无关层通过sysfs这样的接口向用户空间提供控制接口，使得用户空间的应用程序可以控制系统的电源管理行为。而硬件平台相关层则由系统SoC硬件平台代码实现，用以向各个电源管理模块的硬件平台无关层提供控制特定SoC硬件平台的接口，如调节处理器核的运行频率和电压。本节下面的内容将以APQ8074AB硬件平台为例介绍Linu核心电源管理、CPUIdle、CPUFreq和CPU Hotplug模块的设计。至于Runtime PM的设计，将在3.2节中详细介绍。

在Linux内核之上运行的是用户空间的电源管理模块，通常可以分为两类：一是负责控制系统电源状态的电源管理模块，比如控制系统的运行和休眠状态切换；二是负责调节系统的散热性能的热能管理模块，比如在系统温度过高时降低处理器核的运行频率和电压以达到散热的目的。

#### 3.1.2 Linux核心电源管理

##### 3.1.2.1 概述

Linux核心电源管理主要负责控制内核层面上的系统睡眠与唤醒[]。这里的睡眠指系统从正常工作状态切换到某一低功耗状态。在3.x版本到Linux内核中，定义了三种低功耗状态：待机（Standby）、睡眠（Suspend）和休眠（Hibernate）[2]。其中，待机状态下处理器核处于停止状态而大部分硬件设备也处于空闲状态。这种状态带来的功耗降低效果最少但回到正常工作状态但延迟最短，通常在1秒内。睡眠状态也被称为Suspend-to-RAM。在这种状态下，大部分硬件设备都处于电源关闭或者低功耗状态，处理器核的电源通常是关闭的。程序上下文保存在DRAM中，DRAM处于低功耗的自刷新状态。整个系统只保留用以唤醒系统的模块的供电。这些模块将用以在特定条件下，比如电源键被按下，唤醒处理器核并从DRAM中恢复程序上下文。与待机状态相比，睡眠状态带来的功耗降低效果大很多，通常对于嵌入式Linux系统来说，整个系统的耗电在10mA以下。从睡眠状态恢复到正常状态的延迟与待机状态相比略微增加，但仍可以保持在1秒内。休眠模式通常被称为Suspend-to-Disk。在该状态下，程序上下文保存在外部非易失性存储设备中，比如硬盘。因此，整个系统可以完全关闭电源从而达到最低功耗状态。但同时返回正常工作状态但延迟也最长，可以达到睡眠状态的数十倍。对于嵌入式设备，尤其是手持智能终端设备来说，时刻保持对用户对高度响应至关重要，因此，在这类系统里，Linux核心电源管理主要负责正常工作状态和睡眠状态之间的切换。

下面是Linux核心电源管理控制下的系统睡眠和唤醒的流程。在睡眠流程中，各个设备驱动可以实现prepare(), suspend(), suspend\_late()以及suspend\_no\_irq()这四个回调函数中的一个或多个。这几个回调函数的类型声明如下。

int (\*prepare)(struct device \*dev);

int (\*suspend)(struct device \*dev);

int (\*suspend\_late)(struct device \*dev);

int (\*suspend\_noirq)(struct device \*dev);

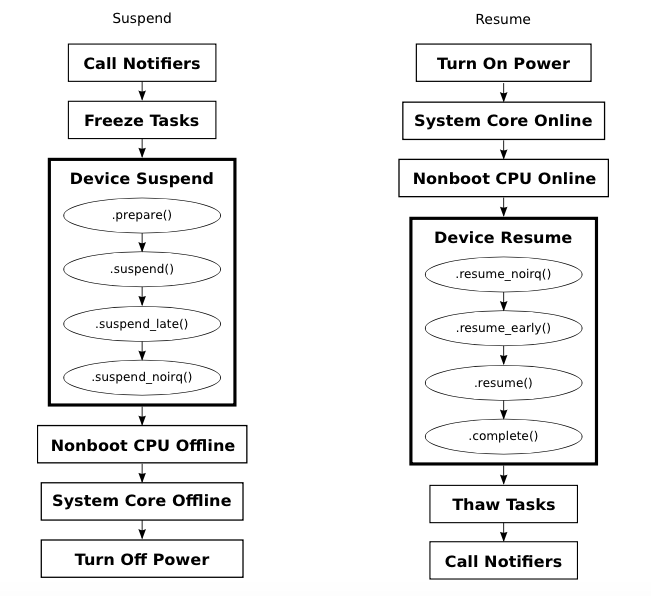
其中，prepare()用于通知设备准备进入睡眠状态。在这个回调函数调用时，用户空间的进程已经停止运行，设备的电源管理状态不应发生变化。在prepare()调用后，设备驱动将不再接受新的子设备的注册。suspend() 用于将设备切换到低功耗状态或彻底关闭供电以将功耗降至最低。通常在低功耗状态下，suspend()需要保存设备在正常工作状态下的寄存器设置以备将来在resume()中恢复，并且当设备所处的子系统的suspend()返回时，设备将不再进行任何IO动作或者是DMA。如果设备的睡眠在suspend()返回时还不能全部完成，那剩余部分通常会在它的suspend\_late()或者suspend\_noirq()回调函数中完成。suspend\_noirq()在执行时将保证该设备的中断处理函数不会被执行，这样任何设备休眠所需要的并且可能会和中段处理产生并发竞争的代码就可以在这里实现了。需要注意的是，如果设备需要在系统睡眠时能够唤醒系统，那这个配置通常需要在suspend()或者suspend\_noirq()中实现。suspend\_late()与suspend\_noirq()的不同只是在于对中断的处理上。类似地，在唤醒流程中，声明由如下回调函数：

Figure 3 - 3 Linux suspend and wake up flow

图 3 - 3 Linux睡眠与唤醒流程

int (\*resume\_noirq)(struct device \*dev);

int (\*resume\_early)(struct device \*dev);

int (\*resume)(struct device \*dev);

int (\*complete)(struct device \*dev);

其中，resume\_noirq()和resume\_early()分别与 suspend\_noirq()和suspend\_late()相对应，resume()和complete()分别与suspend()和prepare()相对应，它们将睡眠流程中各个回调函数所做的动作撤销，将设备从无供电或者低功耗状态切换到睡眠之前的状态。

除了设备驱动层面提供的回调函数之外，SoC平台还需要实现一个platform\_suspend\_ops结构用来实现SoC平台级的系统休眠与唤醒的回调函数。platform\_suspend\_ops结构的声明如下：

typedef int \_\_bitwise suspend\_state\_t;

#define PM\_SUSPEND\_ON ((\_\_force suspend\_state\_t) 0)

#define PM\_SUSPEND\_STANDBY ((\_\_force suspend\_state\_t) 1)

#define PM\_SUSPEND\_MEM ((\_\_force suspend\_state\_t) 3)

#define PM\_SUSPEND\_MAX ((\_\_force suspend\_state\_t) 4)

struct platform\_suspend\_ops {

int (\*valid)(suspend\_state\_t state);

int (\*begin)(suspend\_state\_t state);

int (\*prepare)(void);

int (\*prepare\_late)(void);

int (\*enter)(suspend\_state\_t state);

void (\*wake)(void);

void (\*finish)(void);

bool (\*suspend\_again)(void);

void (\*end)(void);

void (\*recover)(void);

};

其中，valid()用于决定该SoC平台是否支持state参数中所指定的睡眠状态。begin()用于启动特定睡眠状态在该SoC平台上的切换流程。它会在之前所描述的设备睡眠流程开始前执行。prepare()用于准备该SoC平台进入特定睡眠状态。它会在所有设备的suspend()后，suspend\_late()与suspend\_noirq()前执行。prepare\_late()用于完成prepare()中没有完成的流程。它会在所有设备的suspend\_late()和suspend\_noirq()之后，关闭非启动处理器核(Nonboot CPU offline)之前执行。enter()用于让该SoC真正进入begin()中或者是参数sate（如果begin()没有实现）所指定的睡眠状态。wake()在系统从某一睡眠状态唤醒，在非启动处理器核启动之后，设备驱动的resume\_early()和resume\_noirq()之前被调用。finish()用于完成SoC平台的唤醒流程。它在设备的resume()之前被调用。suspend\_again()用于决定系统是否应该再次睡眠。end()会在设备的resume()之后被调用，用来通知该SoC平台系统已经恢复到工作状态了。recover()会在设备睡眠过程中出现异常时调用，用来将SoC平台恢复到睡眠前的状态。在这些回调函数中，只有enter()是必须实现的。另外，如前所述，对于嵌入式设备来说，通常只需要支持Suspend-to-RAM(PM\_SUSPEND\_MEM)的睡眠状态，因此，Linux内核提供了suspend\_valid\_only\_mem()用来实现valid()。

##### 3.1.2.2 高通APQ8074AB平台设计

高通APQ8074AB平台支持以下四种低功耗模式：SWFI, Retention, Standalone power-collapse和Power-collapse。其中，SWFI模式只是关闭各个Krait处理器核的时钟；Retention模式在SWFI的基础上将Krait核切换到低电压模式；Standalone power-collapse模式下，Krait核的时钟和电源都处于关闭状态，但不通知RPM处理器，因此不改变APQ8074AB SoC中其他处理器的电源管理状态；最后的Power-collapse模式在Standalone power-collapse基础上通过RPM将SoC上所有处理器的电源和时钟关闭，并且将各处理器所共享的资源如外部晶振，L2 cache等置于低功耗模式[]。这四种模式在功耗是依次降低的，但在从该模式唤醒到延迟时间也是相应增加的。见下表：

|  |  |  |
| --- | --- | --- |
|  | 功耗预期（mW） | 唤醒延迟（us） |
| SWFI | 715 | 1 |
| Retention | 542 | 35 |
| Standalone power-collapse | 476 | 300 |
| Power-collapse | 400 | 500 |

表3 - 1 Krait核低功耗模式功耗与唤醒延迟

Table 3 - 1 Krait core power consumption and wake up latency in LPM

在APQ8074AB的电源管理设计中，我们将SWFI，Standalone power-collapse和Power-collapse这三种模式用于Suspend-to-RAM，其中默认使能的是SWFI，后两种可以在系统启动后由用户决定是否使能。这主要是为了调试和Suspend有关的系统稳定性问题时可以方便地排除影响因素。因此，在enter()回调函数中，需要首先决定进入何种低功耗模式。这由各种模式的功耗和用户的选择来共同决定。排除用户没有使能的模式，以功耗更低的模式为优先。在决定了需要进入的模式后就相应地执行进入该模式的流程直至系统被唤醒，完成整个enter()回调函数。其流程可表示如下：

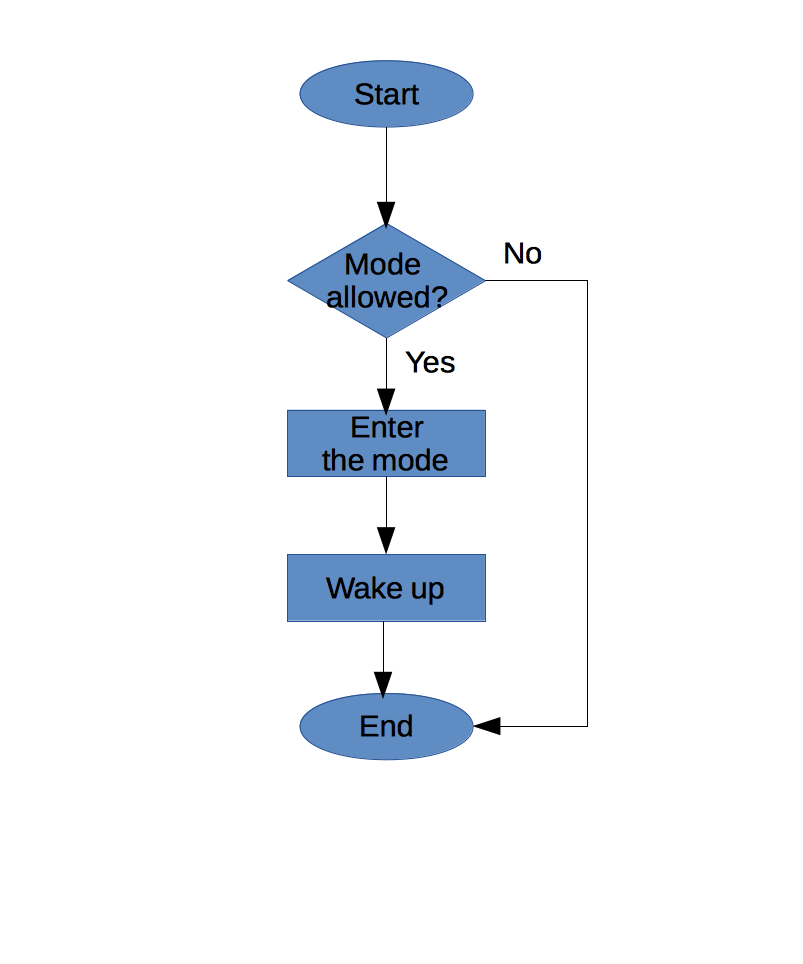


图 3 - 4 APQ8074AB enter()回调函数流程

Figure 3 - 4 APQ8074AB enter() callback flow

#### 3.1.3 CPUIdle模块设计

#### 3.1.4 CPUFreq模块设计

#### 3.1.5 CPU Hotplug模块设计

### 3.2设备驱动的Runtime PM设计

#### 3.2.1通用设计思想

Runtime PM框架是按照Linux内核中的设备模型设计的。在该设备模型中，设备之间是一个抽象的总线树形的关系：设备挂载在某一类型的总线上，它们是总线控制器的子设备，而它们本身又是下一级总线设备的父设备。例如在下图中，CPU下由PCI总线控制器，Display总线控制器和I2C总线控制器3个子设备。而在PCI总线控制器下又有3个PCI设备作为它的子设备。其中USB控制器又是下一级USB总线上各个USB设备的父设备。

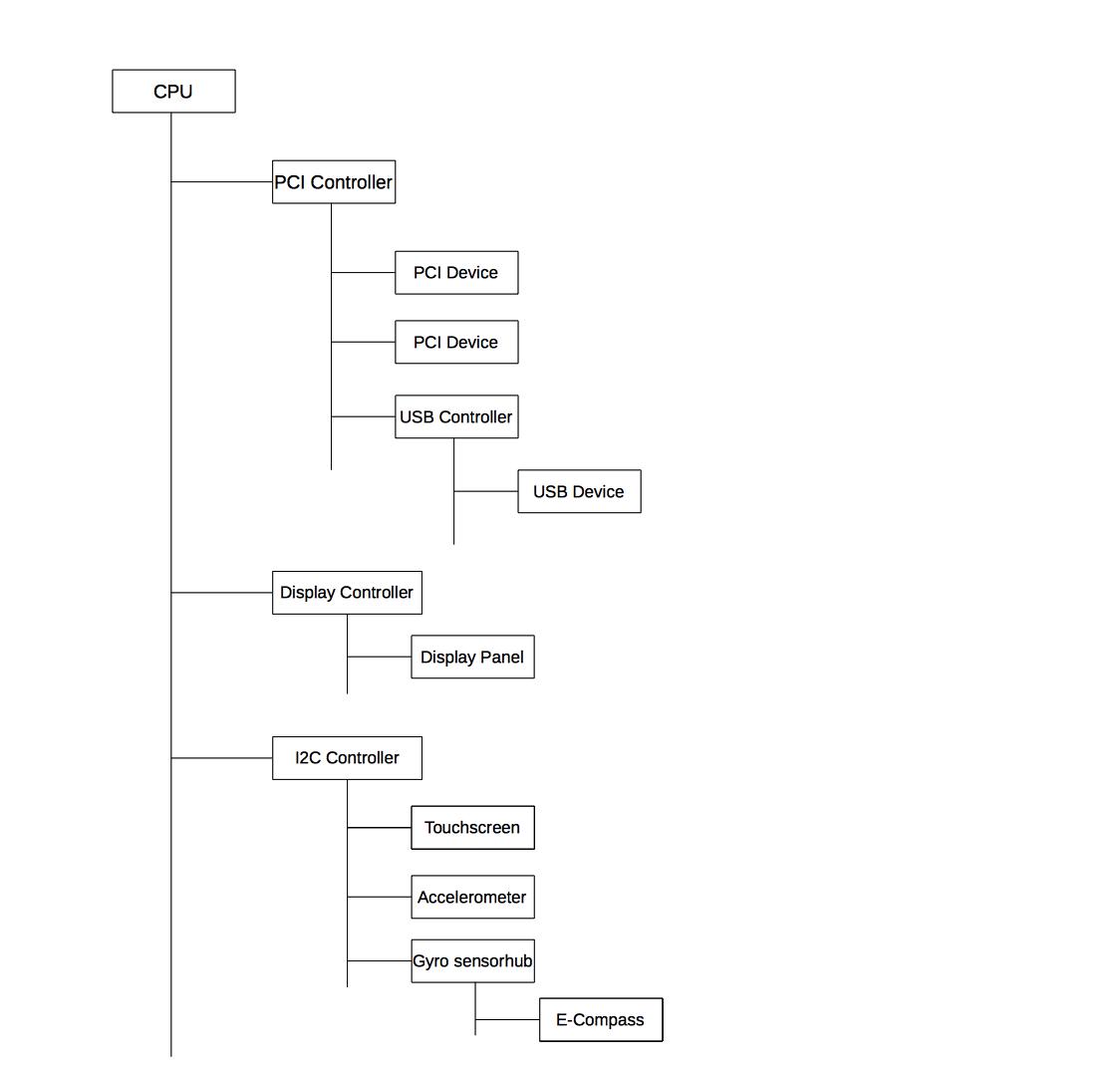


图 3‑2 Linux内核设备模型

Figure 3‑2 Linux device model

基于这一模型，Runtime PM为每一个设备定义了两个计数器：usage\_count和child\_count。其中，usage\_count用来纪录当前设备的使用量计数：每当当前设备的使用量发生改变，如开始使用或结束使用，usage\_count相应地增加或减少1；child\_count用来纪录当前设备的活跃(active)子设备的计数。

在Runtime PM框架中，设备有3个电源管理状态：活跃(active)、空闲(idle)和休眠(suspended)。Runtime PM在初始化时认为每一个设备都是处于suspended状态。当设备被用户开始使用后，设备的状态由suspended转换为active，同时相应地增加usage\_count和当前设备的父设备的child\_count。如果父设备此时处于suspended或idle状态，父设备也将转换为active状态，依此类推。当设备的所有用户使用结束，usage\_count减少到0并且同时child\_count也是0的话，设备将由active状态转换到idle状态。如果在idle状态下设备可以休眠，Runtime PM将会把设备切换到suspended状态。如果此时又出现用户使用当前设备或出现active状态的子设备，当前设备将切换到active状态。Usage\_count的增加和减少是通过调用pm\_runtime\_get()和pm\_runtime\_put()以及这两个函数的变种函数实现的。整个状态切换过程如下图所示：

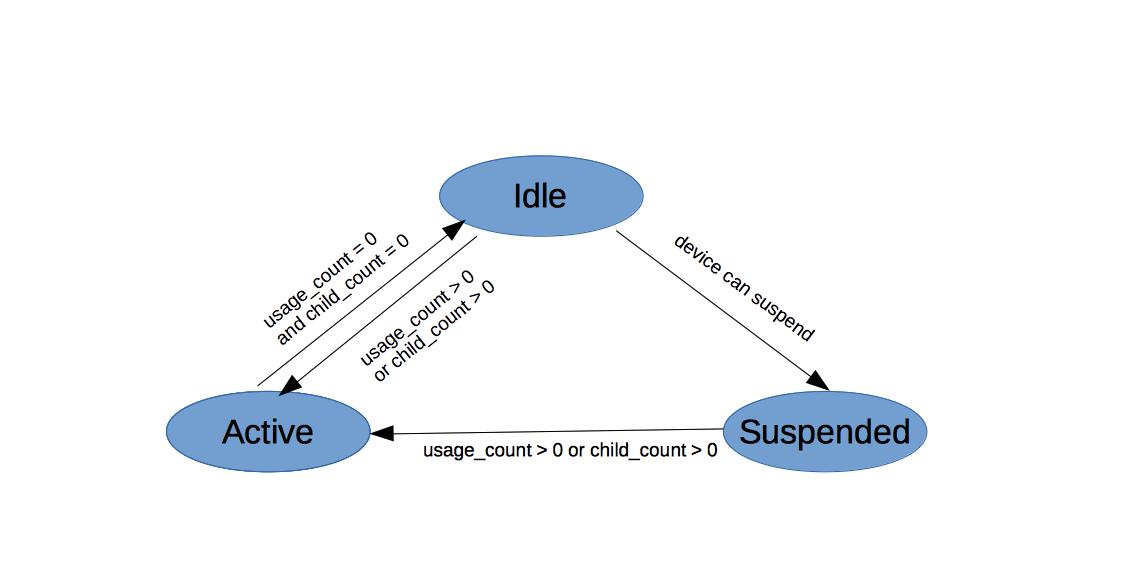


图 3‑2 Runtime PM状态

Figure 3‑2 Runtime PM states

在上述状态述切换过程中，设备驱动可以实现三个相应的回调函数(callback function)：runtime\_suepnd()，runtime\_resume()和 runtime\_idle()。函数的原型声明如下[]：

struct dev\_pm\_ops {

...

int (\*runtime\_suspend)(struct device \*dev);

int (\*runtime\_resume)(struct device \*dev);

int (\*runtime\_idle)(struct device \*dev);

...

};

其中，runtime\_suspend()用于设备在切换到suspended状态时将其切换到低功耗状态。这通常包括关闭不需要的电源和时钟输入和降低设备运行频率。runtime\_resume()用来恢复runtime\_suspend()中所做的操作，将设备切换到正常使用状态。runtime\_idle()的实现是可选的，当设备驱动认为设备可以切换到suspended状态时，该函数返回0，否则返回非0值。

在实际运用中，对于嵌入式系统来说，设备状态的改变代价并不小。这会花费宝贵的CPU时间和IO等待时间。因此，Runtime PM的设计遵守一个设备只有当确信会保持低功耗一定时间的情况下才需要将其切换到suspended状态的原则，以避免将设备频繁的切换在suspended和active状态之间。为此，Runtime PM引入了autosuspend的概念。Autosuspend并非是指让设备自动进入suspended状态，而是指让设备在保持了一定的不活跃时间后才进入suspended状态[]。

采用autosuspend机制的设备驱动需要调用pm\_runtime\_mark\_last\_busy()来更新Runtime PM框架中该设备的不活跃时间。这通常是在设备执行完IO任务之后并且在调用pm\_runtime\_put()来减少设备usage\_count之前完成的。设备的最小不活跃时间的确定可以根据的设备采用不同的策略。设备驱动在初始化时可以通过调用pm\_runtime\_set\_autosuspend\_delay()来告诉系统这一时间。在设备运行过程中，用户空间的应用程序也可以通过/sys/devices/.../power/autosuspend\_delay\_ms这个sysfs节点来更新这一设置值。另外，设备驱动还需要调用pm\_runtime\_use\_autosuspend()来声明该设备采用autosuspend机制并且在调用Runtime PM的状态切换函数时使用\_autosuspend后缀的函数：

pm\_runtime\_suspend()要改用pm\_runtime\_autosuspend();

pm\_schedule\_suspend()要改用pm\_request\_autosuspend();

pm\_runtime\_put()要改用pm\_runtime\_put\_autosuspend();

pm\_runtime\_put\_sync()要改用pm\_runtime\_put\_sync\_autosuspend().

由于autosuspend机制下，设备完成IO任务和进入suspended状态是异步的，设备驱动就必须实现runtime\_suspend()回调函数与IO任务之间的同步。一个典型的伪代码[]实现如下：

foo\_read\_or\_write(struct foo\_priv \*foo, void \*data)

{

lock(&foo->private\_lock);

add\_request\_to\_io\_queue(foo, data);

if (foo->num\_pending\_requests++ == 0)

pm\_runtime\_get(&foo->dev);

if (!foo->is\_suspended)

foo\_process\_next\_request(foo);

unlock(&foo->private\_lock);

}

foo\_io\_completion(struct foo\_priv \*foo, void \*req)

{

lock(&foo->private\_lock);

if (--foo->num\_pending\_requests == 0) {

pm\_runtime\_mark\_last\_busy(&foo->dev);

pm\_runtime\_put\_autosuspend(&foo->dev);

} else {

foo\_process\_next\_request(foo);

}

unlock(&foo->private\_lock);

/\* Send req result back to the user ... \*/

}

int foo\_runtime\_suspend(struct device \*dev)

{

struct foo\_priv foo = container\_of(dev, ...);

int ret = 0;

lock(&foo->private\_lock);

if (foo->num\_pending\_requests > 0) {

ret = -EBUSY;

} else {

/\* ... suspend the device ... \*/

foo->is\_suspended = 1;

}

unlock(&foo->private\_lock);

return ret;

}

int foo\_runtime\_resume(struct device \*dev)

{

struct foo\_priv foo = container\_of(dev, ...);

lock(&foo->private\_lock);

/\* ... resume the device ... \*/

foo->is\_suspended = 0;

pm\_runtime\_mark\_last\_busy(&foo->dev);

if (foo->num\_pending\_requests > 0)

foo\_process\_requests(foo);

unlock(&foo->private\_lock);

return 0;

}

这里的重点就在于当foo\_io\_completion()完成后要求设备autosuspend的时候，设备的runtime\_suspend()回调函数(foo\_runtime\_suspend())有可能会和设备的IO任务(foo\_read\_or\_write())发生竞争现象。因此，foo\_runtime\_suspend()在让设备进入suspended状态前就必须检查是否有等待处理的IO任务。

综上所述，在实现设备的Runtime PM中，需要在设备驱动中实现runtime\_suspend(), runtime\_resume()和runtime\_idle()回调函数。在设备使用开始和结束时调用pm\_runtime\_get()和pm\_runtime\_put()来维护设备的usage\_count。在采用autosuspend机制的设备驱动中还需要决定设备的最小不活跃时间，同步runtime\_suspend()回调函数和IO任务以避免竞争。

#### Atmel MXT1664触摸屏控制器驱动的Runtime PM设计

##### **MXT1664的硬件特性**

在该嵌入式系统项目中，我们使用的是Atmel公司的MXT1664电容触摸屏控制器。MXT1664的硬件特性包括：

* maXTouch™触摸屏解决方案
  + 支持双触摸屏。每个触摸屏支持12-bit多点触摸报告和最多16个触摸并发的实时坐标跟踪
  + 支持尺寸为8到14英寸的触摸屏
* 扫描速度
  + 单点触摸最高250Hz
  + 16点触摸最高100Hz
* 响应时间
  + idle状态下的首次触摸延迟小于15ms
* 接口
  + I2C总线接口，支持400kHz快速模式以及最高1.7MHz的高速模式
  + USB 2.0标准兼容接口, full speed 12Mbps
  + Microsoft Windows 8的HID-I2C接口
* 电源
  + 最低1.8V的数字供电
  + 2.7到3.3V的模拟供电

在该系统中，我们采用MXT1664的I2C总线接口，挂载在APQ8074AB的I2C总线上。数字电源采用1.8V和3.3V，模拟电源为3.3V。通过APQ8074AB的GPIO连接芯片的控制和状态管脚如复位和中断。其接口部分原理图如下：



图 3‑3 MXT1664接口原理图

Figure 3‑3 MXT1664 interface schematics

根据芯片数据手册，MXT1664有三种工作模式: Active, Idle和Deep Sleep。其中功耗最低的工作模式是DeepSleep。各种模式下的功耗都分为三部分: 模拟电源功耗(Analog Supply, AIdd), 数字电源(Digital Supply, DIdd)和X Drive Supply(XIdd)。在Deep Sleep下，这三部分的平均耗电流分别为6.8mA, 0.08mA和0.0004mA。总的平均耗电流约为6.9mA。

##### **软件设计目标**

目前MXT1664驱动中没有实现软件上的Runtime的电源管理，当空闲时，硬件工作在MXT664的Idle模式。软件设置的空闲情况下的扫描间隔为100ms。因此，Runtime PM的软件设计目标是让MXT664在空闲时进入更低功耗的Deep Sleep模式。由于用户的触摸动作是随机的，在一次触摸之后很可能会很快发生下一个，我们采用Runtime PM的autosuspend模式来延迟一定的时间来判断硬件是否真的可以进入Runtime PM的idle状态并进而进入suspended状态。然而，MXT1664在Deep Sleep模式下时无法响应用户的触摸动作的，因此我们还需要在MXT1664进入到Deep Sleep模式之后定时地将其唤醒到Idle模式下以检测下一个用户的触摸动作。

##### **驱动流程**

基于以上目标，MXT1664驱动的Runtime PM和mxt\_runtime\_suspend()回调函数的流程可以用下图表示：

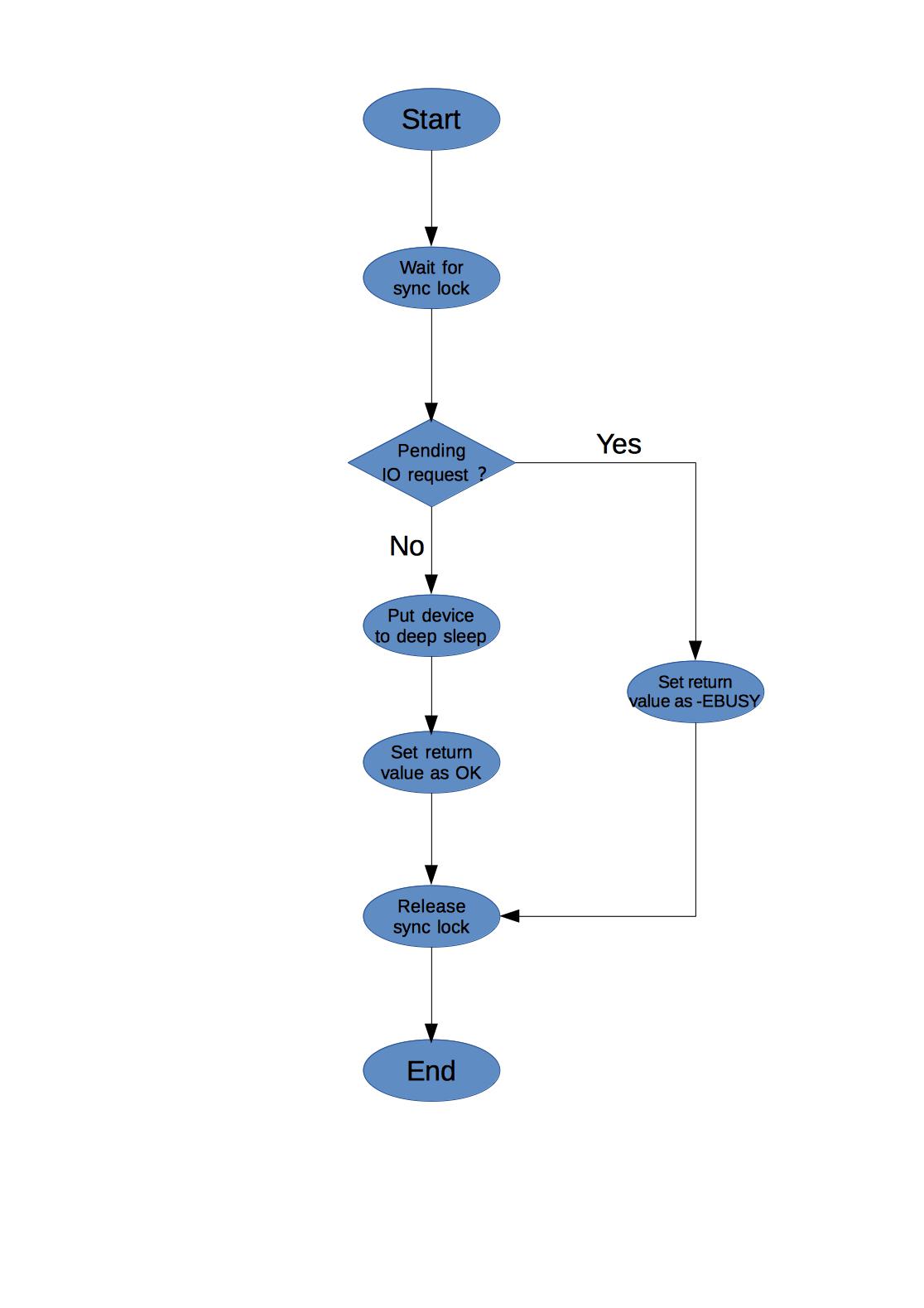
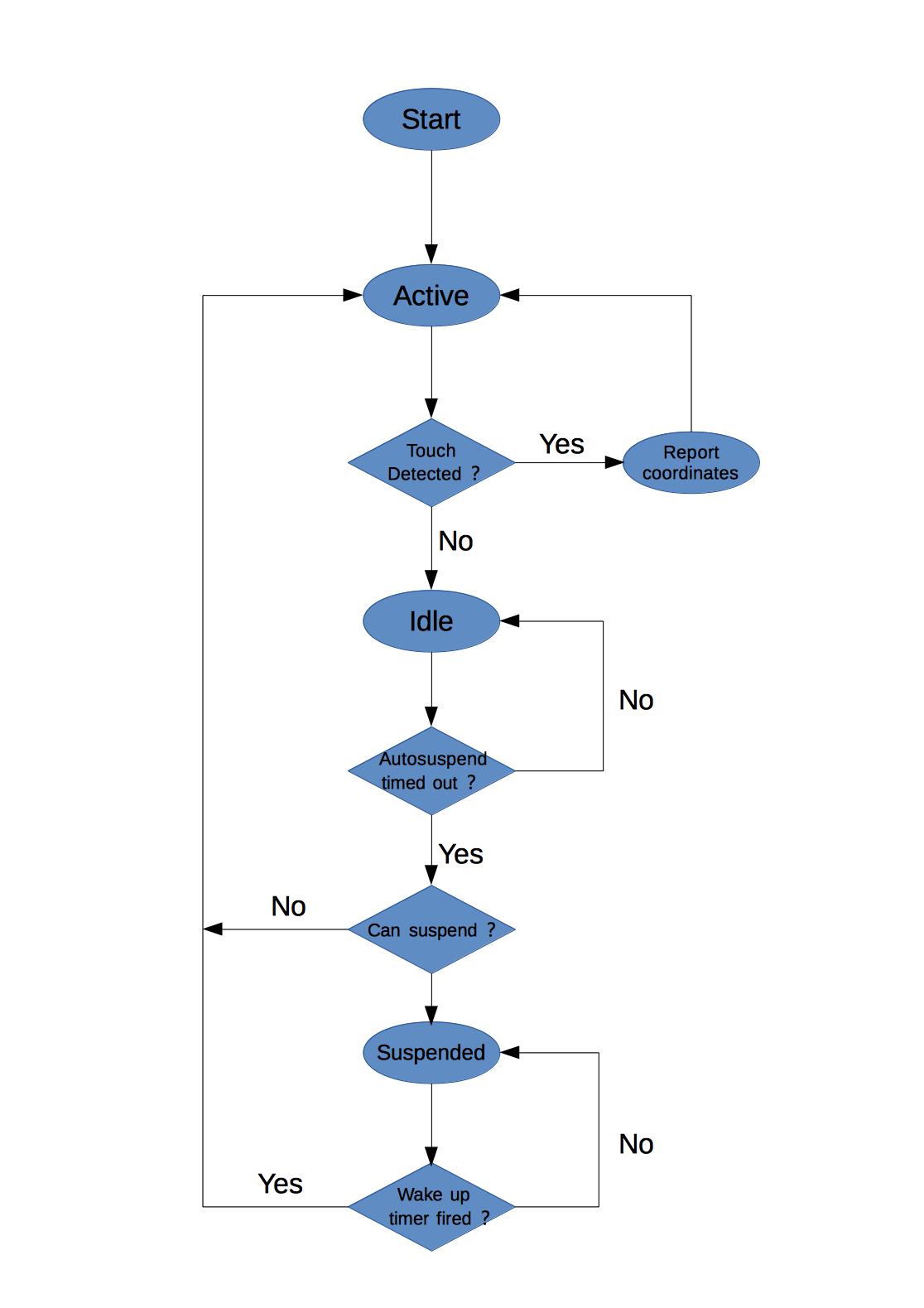


图 3‑4 MXT1664驱动流程 图 3‑5 mxt\_runtime\_suspend()流程

Figure 3‑4 MXT1664 driver flow Figure 3‑5 mxt\_runtime\_suspend() flow

mxt\_runtime\_resume()回调函数和为实现autosuspend()功能添加的mxt\_io\_complete()函数流程设计如下：

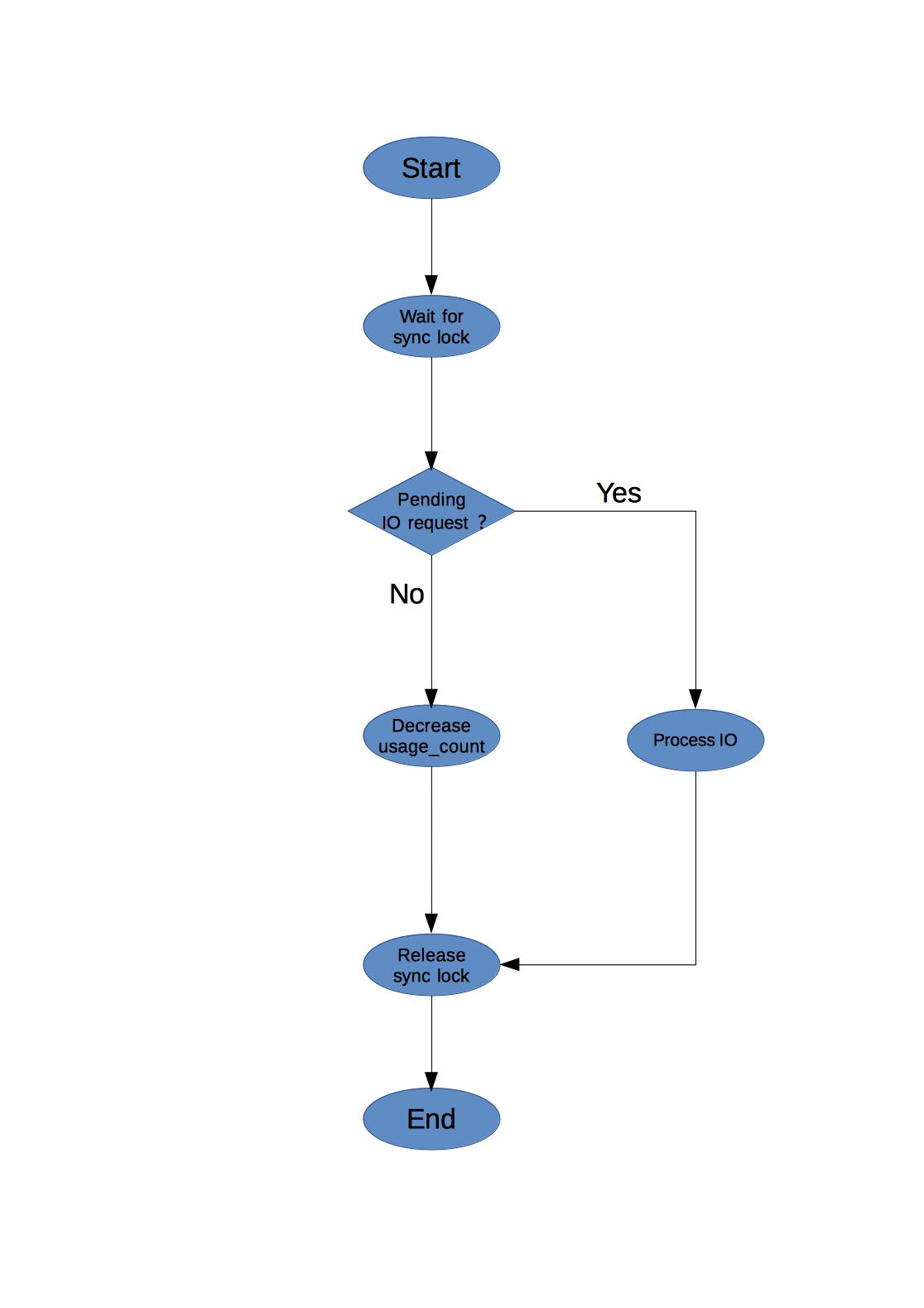
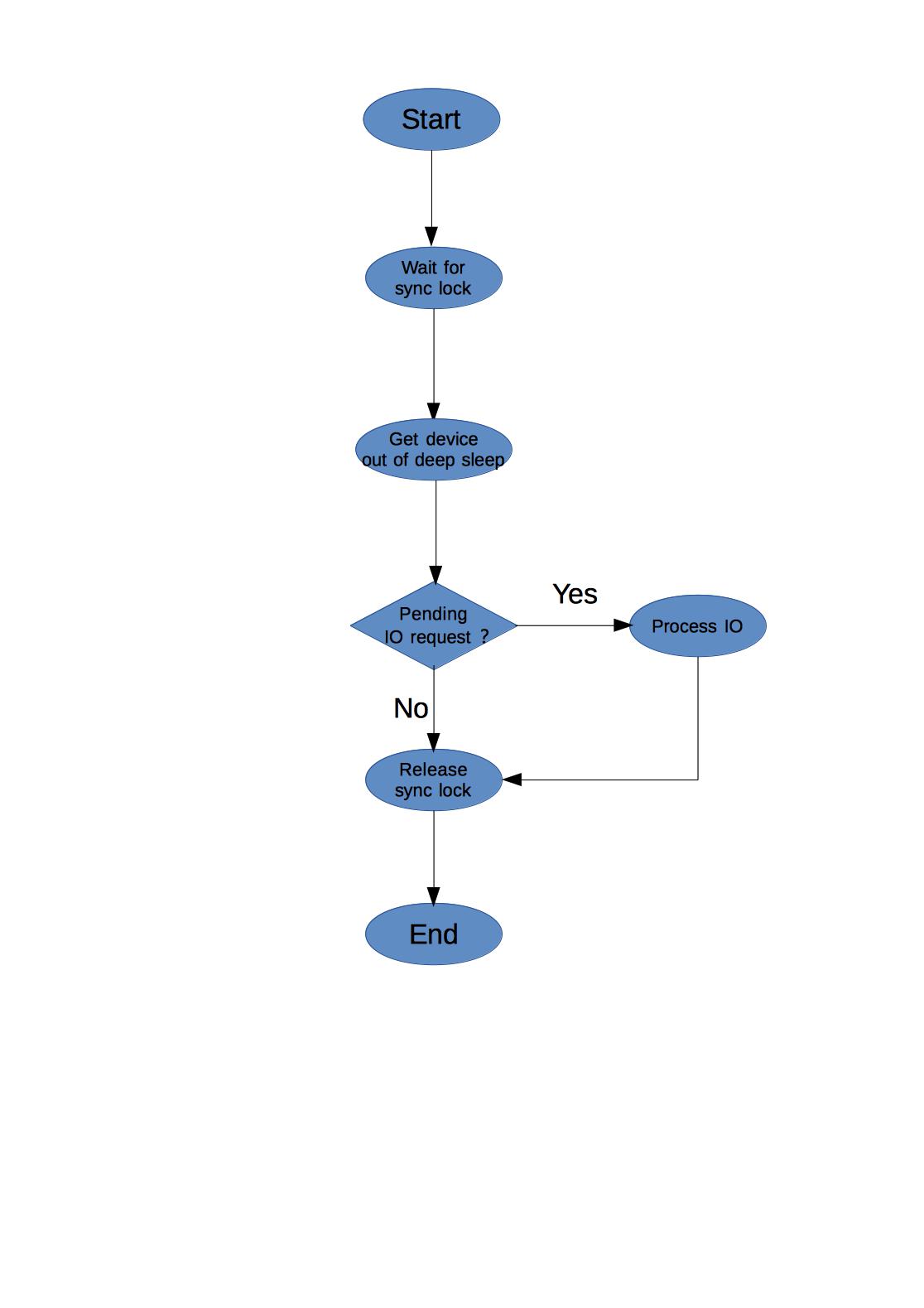


图 3‑6 mxt\_runtime\_resume()流程 图 3‑7 mxt\_io\_complete()流程

Figure 3‑6 mxt\_runtime\_resume() flow Figure 3‑7 mxt\_io\_complete() flow

## 第四章 嵌入式Linux系统电源管理的实现

## 第五章 总结与展望

## 参考文献

[] Linaro, PM Architecture Block Diagram, https://wiki.linaro.org/WorkingGroups/PowerManagement/Doc/Architecture

[] 80-NA157-25, MSM8974 AndroidTM Power and Thermal Management, Revision B, Qualcomm Technologies, Inc, 2012.

[] A. Leonard Brown, Rafael J. Wysocki, Suspend-to-RAM in Linux. Proceedings of the Linux Symposium, Volume One. July, 2008.

[] Patrick Mochel, Linux Kernel Power Management. Proceeding of the Linux Symposium. July, 2003.

[] Rafael J. Wysocki, Alan Stern, Runtime Power Management Framework for I/O Devices, https://www.kernel.org/doc/Documentation/power/runtime\_pm.txt