

第 19 章

Linux 电源管理的系统架构和驱动

本章导读

Linux 在消费电子领域的应用已经相当普遍，而对于消费电子产品而言，省电是一个重要的议题。

19.1 节阐述了 Linux 电源管理的总体架构。

19.2~19.8 节分别论述了 CPUFreq、CPUIdle、CPU 热插拔以及底层的基础设施 Regulator、OPP 以及电源管理的调试工具 PowerTop。

19.9 节讲解了系统挂起到 RAM 的过程以及设备驱动是如何对挂起到 RAM 支持的。

19.10 节讲解了设备驱动的运行时挂起。

本章是 Linux 设备驱动工程师必备的知识体系。

19.1 Linux 电源管理的全局架构

Linux 电源管理非常复杂，牵扯到系统级的待机、频率电压变换、系统空闲时的处理以及每个设备驱动对系统待机的支持和每个设备的运行时（Runtime）电源管理，可以说它和系统中的每个设备驱动都息息相关。

对于消费电子产品来说，电源管理相当重要。因此，这部分工作往往在开发周期中占据相当大的比重，图 19.1 呈现了 Linux 内核电源管理的整体架构。大体可以归纳为如下几类：

- 1) CPU 在运行时根据系统负载进行动态电压和频率变换的 CPUFreq。
- 2) CPU 在系统空闲时根据空闲的情况进行低功耗模式的 CPUIdle。
- 3) 多核系统下 CPU 的热插拔支持。
- 4) 系统和设备针对延迟的特别需求而提出申请的 PM QoS，它会作用于 CPUIdle 的具体策略。
- 5) 设备驱动针对系统挂起到 RAM/ 硬盘的一系列入口函数。
- 6) SoC 进入挂起状态、SDRAM 自刷新的入口。
- 7) 设备的运行时动态电源管理，根据使用情况动态开关设备。

8) 底层的时钟、稳压器、频率/电压表(OPP 模块完成)支撑,各驱动子系统都可能用到。

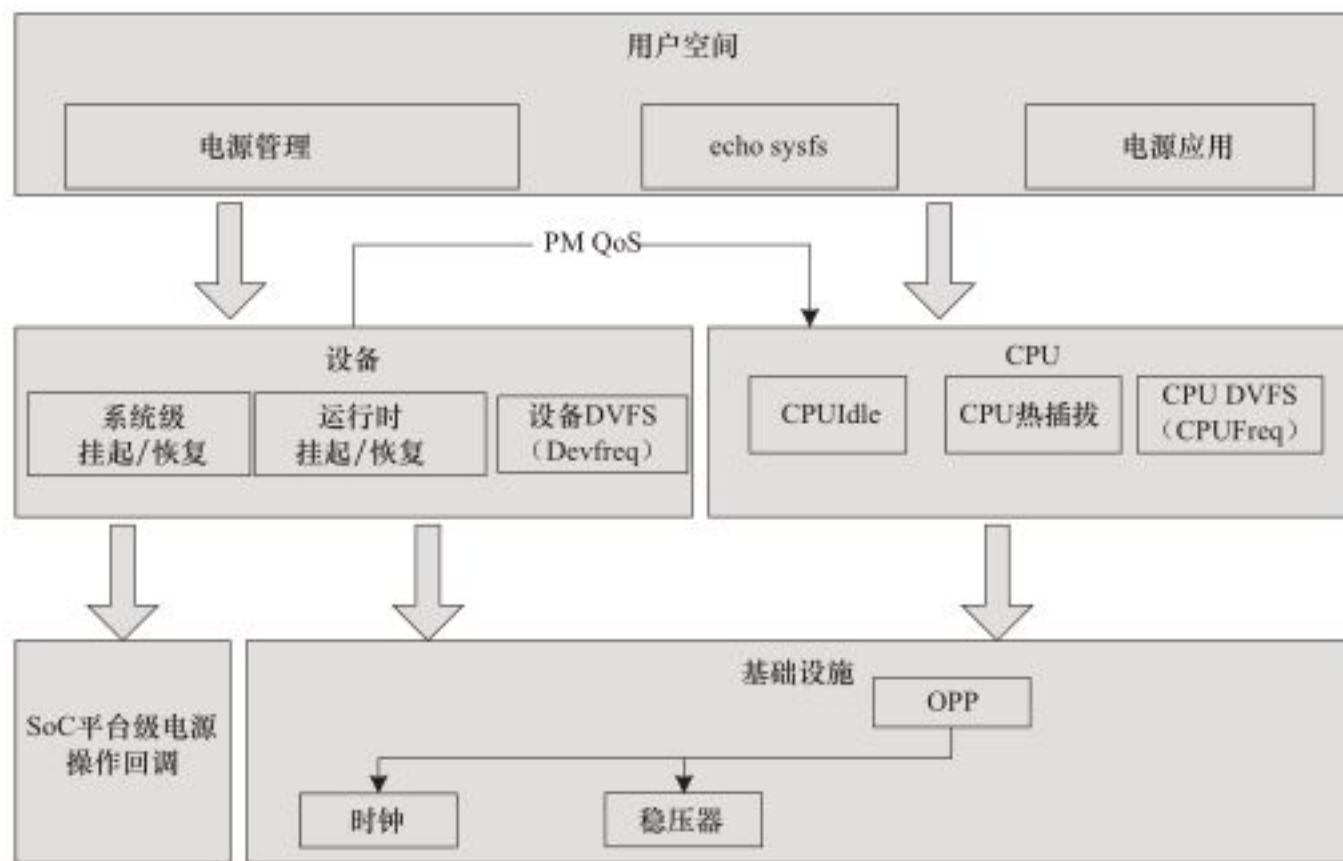


图 19.1 Linux 内核电源管理的整体架构

19.2 CPUFreq 驱动

CPUFreq 子系统位于 drivers/cpufreq 目录下, 负责进行运行过程中 CPU 频率和电压的动态调整, 即 DVFS (Dynamic Voltage Frequency Scaling, 动态电压频率调整)。运行时进行 CPU 电压和频率调整的原因是: CMOS 电路中的功耗与电压的平方成正比、与频率成正比 ($P \propto fV^2$), 因此降低电压和频率可降低功耗。

CPUFreq 的核心层位于 drivers/cpufreq/cpufreq.c 下, 它为各个 SoC 的 CPUFreq 驱动的实现提供了一套统一的接口, 并实现了一套 notifier 机制, 可以在 CPUFreq 的策略和频率改变的时候向其他模块发出通知。另外, 在 CPU 运行频率发生变化的时候, 内核的 loops_per_jiffy 常数也会发生相应变化。

19.2.1 SoC 的 CPUFreq 驱动实现

每个 SoC 的具体 CPUFreq 驱动实例只需要实现电压、频率表, 以及从硬件层面完成这些变化。

CPUFreq 核心层提供了如下 API 以供 SoC 注册自身的 CPUFreq 驱动:

```
int cpufreq_register_driver(struct cpufreq_driver *driver_data);
```

其参数为一个 cpufreq_driver 结构体指针，实际上，cpufreq_driver 封装了一个具体的 SoC 的 CPUFreq 驱动的主体，该结构体形如代码清单 19.1 所示。

代码清单 19.1 cpufreq_driver 结构体

```

1 struct cpufreq_driver {
2     struct module          *owner;
3     char                  name[CPUFREQ_NAME_LEN];
4     u8                    flags;
5
6     /* needed by all drivers */
7     int      (*init)        (struct cpufreq_policy *policy);
8     int      (*verify)      (struct cpufreq_policy *policy);
9
10    /* define one out of two */
11    int      (*setpolicy)    (struct cpufreq_policy *policy);
12    int      (*target)      (struct cpufreq_policy *policy),
13    unsigned int target_freq,
14    unsigned int relation);
15
16    /* should be defined, if possible */
17    unsigned int (*get)      (unsigned intcpu);
18
19    /* optional */
20    unsigned int (*getavg)   (struct cpufreq_policy *policy,
21    unsigned intcpu);
22    int      (*bios_limit)  (intcpu, unsigned int *limit);
23
24    int      (*exit)        (struct cpufreq_policy *policy);
25    int      (*suspend)     (struct cpufreq_policy *policy);
26    int      (*resume)      (struct cpufreq_policy *policy);
27    struct freq_attr       **attr;
28 };

```

其中的 owner 成员一般被设置为 THIS_MODULE；name 成员是 CPUFreq 驱动的名字，如 drivers/cpufreq/s5pv210-cpufreq.c 设置 name 为 s5pv210，drivers/cpufreq/omap-cpufreq.c 设置 name 为 omap；flags 是一些暗示性的标志，譬如，若设置了 CPUFREQ_CONST_LOOPS，则是告诉内核 loops_per_jiffy 不会因为 CPU 频率的变化而变化。

init() 成员是一个 per-CPU 初始化函数指针，每当一个新的 CPU 被注册进系统的时候，该函数就被调用，该函数接受一个 cpufreq_policy 的指针参数，在 init() 成员函数中，可进行如下设置：

```

policy->cpuinfo.min_freq
policy->cpuinfo.max_freq

```

上述代码描述的是该 CPU 支持的最小频率和最大频率（单位是 kHz）。

```

policy->cpuinfo.transition_latency

```

上述代码描述的是 CPU 进行频率切换所需要的延迟（单位是 ns）

```
policy->cur
```

上述代码描述的是 CPU 的当前频率。

```
policy->policy
policy->governor
policy->min
policy->max
```

上述代码定义该 CPU 的缺省策略，以及在缺省策略情况下，该策略支持的最小、最大 CPU 频率。

`verify()` 成员函数用于对用户的 CPUFreq 策略设置进行有效性验证和数据修正。每当用户设定一个新策略时，该函数根据老的策略和新的策略，检验新策略设置的有效性并对无效设置进行必要的修正。在该成员函数的具体实现中，常用到如下辅助函数：

```
cpufreq_verify_within_limits(struct cpufreq_policy *policy, unsigned int min_freq,
    unsigned int max_freq);
```

`setpolicy()` 成员函数接受一个 `policy` 参数（包含 `policy->policy`、`policy->min` 和 `policy->max` 等成员），实现了这个成员函数的 CPU 一般具备在一个范围（limit，从 `policy->min` 到 `policy->max`）里自动调整频率的能力。目前只有少数驱动（如 `intel_pstate.c` 和 `longrun.c`）包含这样的成员函数，而绝大多数 CPU 都不会实现此函数，一般只实现 `target()` 成员函数，`target()` 的参数直接就是一个指定的频率。

`target()` 成员函数用于将频率调整到一个指定的值，接受 3 个参数：`policy`、`target_freq` 和 `relation`。`target_freq` 是目标频率，实际驱动总是要设定真实的 CPU 频率到最接近于 `target_freq`，并且设定的频率必须位于 `policy->min` 到 `policy->max` 之间。在设定频率接近 `target_freq` 的情况下，`relation` 若为 `CPUFREQ_REL_L`，则暗示设置的频率应该大于或等于 `target_freq`；`relation` 若为 `CPUFREQ_REL_H`，则暗示设置的频率应该小于或等于 `target_freq`。

表 19.1 描述了 `setpolicy()` 和 `target()` 所针对的 CPU 以及调用方式上的区别。

表 19.1 `setpolicy()` 和 `target()` 所针对的 CPU 及其调用方式上的区别

Setpolicy()	Target()
CPU 具备在一定范围内独立调整频率的能力	CPU 只能被指定频率
CPUfreq policy 调用到 <code>setpolicy()</code> ，由 CPU 独立在一个范围内调整频率	由 CPUFreq 核心层根据系统负载和策略综合决定目标频率

根据芯片内部 PLL 和分频器的关系，ARM SoC 一般不具备独立调整频率的能力，往往 SoC 的 CPUFreq 驱动会提供一个频率表，频率在该表的范围内进行变更，因此一般实现 `target()` 成员函数。

CPUFreq 核心层提供了一组与频率表相关的辅助 API。

```
int cpufreq_frequency_table_cpubinfo(struct cpufreq_policy *policy,
                                     struct cpufreq_frequency_table *table);
```

它是 cpufreq_driver 的 init() 成员函数的助手，用于将 policy->min 和 policy->max 设置为与 cpubinfo.min_freq 和 cpubinfo.max_freq 相同的值。

```
int cpufreq_frequency_table_verify(struct cpufreq_policy *policy,
                                   struct cpufreq_frequency_table *table);
```

它是 cpufreq_driver 的 verify() 成员函数的助手，确保至少有 1 个有效的 CPU 频率位于 policy->min 到 policy->max 的范围内。

```
int cpufreq_frequency_table_target(struct cpufreq_policy *policy,
                                   struct cpufreq_frequency_table *table,
                                   unsigned int target_freq,
                                   unsigned int relation,
                                   unsigned int *index);
```

它是 cpufreq_driver 的 target() 成员函数的助手，返回需要设定的频率在频率表中的索引。

省略掉具体的细节，1 个 SoC 的 CPUFreq 驱动实例 drivers/cpufreq/s3c64xx-cpufreq.c 的核心结构如代码清单 19.2 所示。

代码清单 19.2 S3C64xx 的 CPUFreq 驱动

```

1 static unsigned long regulator_latency;
2
3 struct s3c64xx_dvfs {
4     unsigned int vddarm_min;
5     unsigned int vddarm_max;
6 };
7
8 static struct s3c64xx_dvfs s3c64xx_dvfs_table[] = {
9     [0] = { 1000000, 1150000 },
10    ...
11    [4] = { 1300000, 1350000 },
12 };
13
14 static struct cpufreq_frequency_table s3c64xx_freq_table[] = {
15     { 0, 66000 },
16     { 0, 100000 },
17     ...
18     { 0, CPUFREQ_TABLE_END },
19 };
20
21 static int s3c64xx_cpufreq_verify_speed(struct cpufreq_policy *policy)
22 {
23     if (policy->cpu != 0)
24         return -EINVAL;
```

```
25
26 return cpufreq_frequency_table_verify(policy, s3c64xx_freq_table);
27 }
28
29 static unsigned int s3c64xx_cpufreq_get_speed(unsigned intcpu)
30 {
31 if (cpu != 0)
32     return 0;
33
34 return clk_get_rate(armclk) / 1000;
35 }
36
37 static int s3c64xx_cpufreq_set_target(struct cpufreq_policy *policy,
38             unsigned inttarget_freq,
39             unsigned int relation)
40 {
41 ...
42 ret = cpufreq_frequency_table_target(policy, s3c64xx_freq_table,
43             target_freq, relation, &i);
44 ...
45 freqs.cpu = 0;
46 freqs.old = clk_get_rate(armclk) / 1000;
47 freqs.new = s3c64xx_freq_table[i].frequency;
48 freqs.flags = 0;
49 dvfs = &s3c64xx_dvfs_table[s3c64xx_freq_table[i].index];
50
51 if (freqs.old == freqs.new)
52     return 0;
53
54 cpufreq_notify_transition(&freqs, CPUFREQ_PRECHANGE);
55
56 if (vddarm&&freqs.new>freqs.old) {
57     ret = regulator_set_voltage(vddarm,
58             dvfs->vddarm_min,
59             dvfs->vddarm_max);
60     ...
61 }
62
63 ret = clk_set_rate(armclk, freqs.new * 1000);
64 ...
65 cpufreq_notify_transition(&freqs, CPUFREQ_POSTCHANGE);
66
67 if (vddarm&&freqs.new<freqs.old) {
68     ret = regulator_set_voltage(vddarm,
69             dvfs->vddarm_min,
70             dvfs->vddarm_max);
71     ...
72 }
73
74 return 0;
```

```

75 }
76
77 static int s3c64xx_cpufreq_driver_init(struct cpufreq_policy *policy)
78 {
79 ...
80 armclk = clk_get(NULL, "armclk");
81 ...
82 vddarm = regulator_get(NULL, "vddarm");
83 ...
84 s3c64xx_cpufreq_config_regulator();
85
86 freq = s3c64xx_freq_table;
87 while (freq->frequency != CPUFREQ_TABLE_END) {
88     unsigned long r;
89     ...
90 }
91
92 policy->cur = clk_get_rate(armclk) / 1000;
93 policy->cpuinfo.transition_latency = (500 * 1000) + regulator_latency;
94 ret = cpufreq_frequency_table_cpuinfo(policy, s3c64xx_freq_table);
95 ...
96 return ret;
97 }
98
99 static struct cpufreq_driver s3c64xx_cpufreq_driver = {
100 .owner      = THIS_MODULE,
101 .flags      = 0,
102 .verify     = s3c64xx_cpufreq_verify_speed,
103 .target     = s3c64xx_cpufreq_set_target,
104 .get        = s3c64xx_cpufreq_get_speed,
105 .init       = s3c64xx_cpufreq_driver_init,
106 .name       = "s3c",
107 };
108
109 static int __init s3c64xx_cpufreq_init(void)
110 {
111     return cpufreq_register_driver(&s3c64xx_cpufreq_driver);
112 }
113 module_init(s3c64xx_cpufreq_init);

```

第37行 s3c64xx_cpufreq_set_target() 就是完成目标频率设置的函数，它调用了 cpufreq_frequency_table_target() 从 s3c64xx 支持的频率表 s3c64xx_freq_table 里找到合适的频率。在具体的频率和电压设置环节，用的都是 Linux 的标准 API regulator_set_voltage() 和 clk_set_rate() 之类的函数。

第111行在模块初始化的时候通过 cpufreq_register_driver() 注册了 cpufreq_driver 的实例，第94行，在 CPUFreq 的初始化阶段调用 cpufreq_frequency_table_cpuinfo() 注册了频率表。关于频率表，比较新的内核喜欢使用后面章节将介绍的 OPP。

19.2.2 CPUFreq 的策略

SoCCPUFreq 驱动只是设定了 CPU 的频率参数，以及提供了设置频率的途径，但是它并不会管 CPU 自身究竟应该运行在那种频率上。究竟频率依据的是哪种标准，进行何种变化，而这些完全由 CPUFreq 的策略 (policy) 决定，这些策略如表 19.2 所示。

表 19.2 CPUFreq 的策略及其实现方法

CPUFreq 的策略	策略的实现方法
cpufreq_ondemand	平时以低速方式运行，当系统负载提高时按需自动提高频率
cpufreq_performance	CPU 以最高频率运行，即 scaling_max_freq
cpufreq_conservative	字面含义是传统的、保守的，跟 ondemand 相似，区别在于动态频率在变更的时候采用渐进的方式
cpufreq_powersave	CPU 以最低频率运行，即 scaling_min_freq
cpufreq_userspace	让根用户通过 sys 节点 scaling_setspeed 设置频率

在 Android 系统中，则增加了 1 个交互策略，该策略适合于对延迟敏感的 UI 交互任务，当有 UI 交互任务的时候，该策略会更加激进并及时地调整 CPU 频率。

总而言之，系统的状态以及 CPUFreq 的策略共同决定了 CPU 频率跳变的目标，CPUFreq 核心层并将目标频率传递给底层具体 SoC 的 CPUFreq 驱动，该驱动修改硬件，完成频率的变换，如图 19.2 所示。

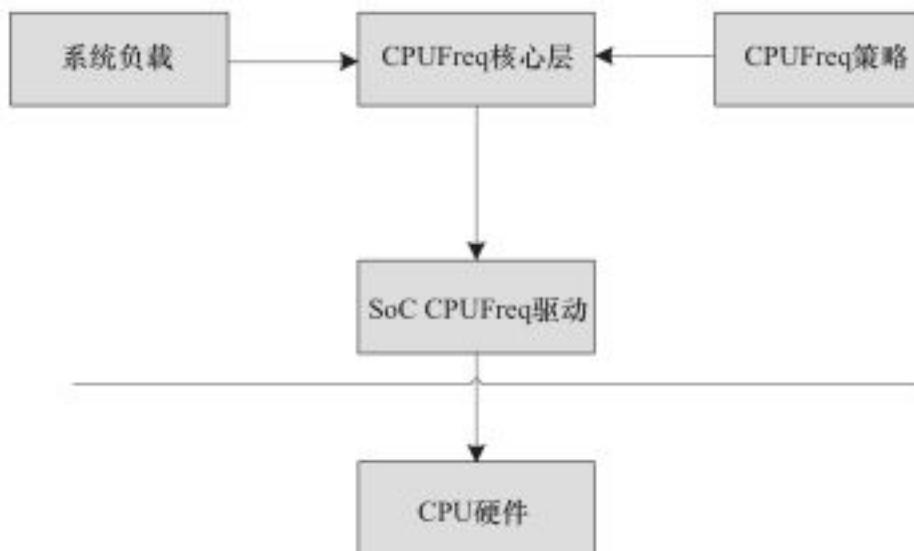


图 19.2 CPUFreq、系统负载、策略与调频

用户空间一般可通过 /sys/devices/system/cpu/cpu0/cpufreq 节点来设置 CPUFreq。譬如，我们要设置 CPUFreq 到 700MHz，采用 userspace 策略，则运行如下命令：

```
# echo userspace > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
# echo 700000 > /sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed
```

19.2.3 CPUFreq 的性能测试和调优

Linux 3.1 以后的内核已经将 cpupower-utils 工具集放入内核的 tools/power/cpupower 目录

中，该工具集当中的 cpufreq-bench 工具可以帮助工程师分析采用 CPUFreq 后对系统性能的影响。

cpufreq-bench 工具的工作原理是模拟系统运行时候的“空闲→忙→空闲→忙”场景，从而触发系统的动态频率变化，然后在使用 ondemand、conservative、interactive 等策略的情况下，计算在做与 performance 高频模式下同样的运算完成任务的时间比例。

交叉编译该工具后，可放入目标电路板文件系统的 /usr/sbin/ 等目录下，运行该工具：

```
# cpufreq-bench -l 50000 -s 100000 -x 50000 -y 100000 -g ondemand -r 5 -n 5 -v
```

会输出一系列的结果，我们提取其中的 Round *n* 这样的行，它表明了 -g ondemand 选项中设定的 ondemand 策略相对于 performance 策略的性能比例，假设值为：

```
Round 1 - 39.74%
Round 2 - 36.35%
Round 3 - 47.91%
Round 4 - 54.22%
Round 5 - 58.64%
```

这显然不太理想，我们在同样的平台上采用 Android 的交互策略，得到新的测试结果：

```
Round 1 - 72.95%
Round 2 - 87.20%
Round 3 - 91.21%
Round 4 - 94.10%
Round 5 - 94.93%
```

一般的目标是在采用 CPUFreq 动态调整频率和电压后，性能应该为 performance 这个高性能策略下的 90% 左右，这样才比较理想。

19.2.4 CPUFreq 通知

CPUFreq 子系统会发出通知的情况有两种：CPUFreq 的策略变化或者 CPU 运行频率变化。

在策略变化的过程中，会发送 3 次通知：

- CPUFREQ_ADJUST：所有注册的 notifier 可以根据硬件或者温度的情况去修改范围（即 policy->min 和 policy->max）；
- CPUFREQ_INCOMPATIBLE：除非前面的策略设定可能会导致硬件出错，否则被注册的 notifier 不能改变范围等设定；
- CPUFREQ_NOTIFY：所有注册的 notifier 都会被告知新的策略已经被设置。

在频率变化的过程中，会发送 2 次通知：

- CPUFREQ_PRECHANGE：准备进行频率变更；
- CPUFREQ_POSTCHANGE：已经完成频率变更。

notifier 中的第 3 个参数是一个 cpufreq_freqs 的结构体，包含 cpu（CPU 号）、old（过去的频率）和 new（现在的频率）这 3 个成员。发送 CPUFREQ_PRECHANGE 和 CPUFREQ_POSTCHANGE 的代码如下：

```
srcu_notifier_call_chain(&cpufreq_transition_notifier_list,
CPUFREQ_PRECHANGE, freqs);
srcu_notifier_call_chain(&cpufreq_transition_notifier_list,
CPUFREQ_POSTCHANGE, freqs);
```

如果某模块关心 CPUFREQ_PRECHANGE 或 CPUFREQ_POSTCHANGE 事件，可简单地使用 Linux notifier 机制监控。譬如，drivers/video/sa1100fb.c 在 CPU 频率变化过程中需对自身硬件进行相关设置，因此它注册了 notifier 并在 CPUFREQ_PRECHANGE 和 CPUFREQ_POSTCHANGE 情况下分别进行不同的处理，如代码清单 19.3 所示。

代码清单 19.3 CPUFreq notifier 案例

```
1 fbi->freq_transition.notifier_call = sa1100fb_freq_transition;
2 cpufreq_register_notifier(&fbi->freq_transition, CPUFREQ_TRANSITION_NOTIFIER);
3 ...
4 sa1100fb_freq_transition(struct notifier_block *nb, unsigned long val,
5 void *data)
6 {
7     struct sa1100fb_info *fbi = TO_INF(nb, freq_transition);
8     struct cpufreq_freqs *f = data;
9     u_intpcd;
10
11     switch (val) {
12         case CPUFREQ_PRECHANGE:
13             set_ctrlr_state(fbi, C_DISABLE_CLKCHANGE);
14             break;
15         case CPUFREQ_POSTCHANGE:
16             pcd = get_pcd(fbi->fb.var.pixclock, f->new);
17             fbi->reg_lccr3 = (fbi->reg_lccr3 & ~0xff) | LCCR3_PixClkDiv(pcd);
18             set_ctrlr_state(fbi, C_ENABLE_CLKCHANGE);
19             break;
20     }
21     return 0;
22 }
```

此外，如果在系统挂起 / 恢复的过程中 CPU 频率会发生变化，则 CPUFreq 子系统也会发出 CPUFREQ_SUSPENDCHANGE 和 CPUFREQ_RESUMECHANGE 这两个通知。

值得一提的是，除了 CPU 以外，一些非 CPU 设备也支持多个操作频率和电压，存在多个 OPP。Linux 3.2 之后的内核也支持针对这种非 CPU 设备的 DVFS，该套子系统为 Devfreq。与 CPUFreq 存在一个 drivers/cpufreq 目录相似，在内核中也存在一个 drivers/devfreq 的目录。

19.3 CPUIdle 驱动

目前的 ARM SoC 大多支持几个不同的 Idle 级别，CPUIdle 驱动子系统存在的目的就是对这些 Idle 状态进行管理，并根据系统的运行情况进入不同的 Idle 级别。具体 SoC 的底层 CPUIdle 驱动实现则提供一个类似于 CPUFreq 驱动频率表的 Idle 级别表，并实现各种不同 Idle 状态的进入和退出流程。

对于 Intel 系列笔记本计算机而言，支持 ACPI (Advanced Configuration and Power Interface，高级配置和电源接口)，一般有 4 个不同的 C 状态（其中 C0 为操作状态，C1 是 Halt 状态，C2 是 Stop-Clock 状态，C3 是 Sleep 状态），如表 19.3 所示。

表 19.3 4 个不同的 C 状态

状态	功耗 (mW)	延迟 (μs)
C0	-1	0
C1	1000	1
C2	500	1
C3	100	57

而对于 ARM 而言，各个 SoC 对于 Idle 的实现方法差异比较大，最简单的 Idle 级别莫过于将 CPU 核置于 WFI（等待中断发生）状态，因此在默认情况下，若 SoC 未实现自身的芯片级 CPUIdle 驱动，则会进入 `cpu_do_idle()`，对于 ARM V7 而言，其实现位于 `arch/arm/mm/proc-v7.S` 中：

```
ENTRY(cpu_v7_do_idle)
    dsb
    wfi
    mov    pc, lr
ENDPROC(cpu_v7_do_idle)
```

与 CPUFreq 类似，CPUIdle 的核心层提供了如下 API 以用于注册一个 `cpuidle_driver` 的实例：

```
int cpuidle_register_driver(struct cpuidle_driver *drv);
```

并提供了如下 API 来注册一个 `cpuidle_device`：

```
int cpuidle_register_device(struct cpuidle_device *dev);
```

CPUIdle 驱动必须针对每个 CPU 注册相应的 `cpuidle_device`，这意味着对于多核 CPU 而言，需要针对每个 CPU 注册一次。

`cpuidle_register_driver()` 接受 1 个 `cpuidle_driver` 结构体的指针参数，该结构体是 CPUIdle 驱动的主体，其定义如代码清单 19.4 所示。

代码清单 19.4 `cpuidle_driver` 结构体

```
1 struct cpuidle_driver {
2     const char           *name;
```

```

3 struct module          *owner;
4
5 unsigned int           power_specified:1;
6     /* set to 1 to use the core cpuidle time keeping (for all states). */
7 unsigned int           en_core_tk_irqgen:1;
8 struct cpuidle_state   states[CPUIDLE_STATE_MAX];
9 int state_count;
10 int safe_state_index;
11 };

```

该结构体的关键成员是 1 个 cpuidle_state 的表，其实该表就是用于存储各种不同 Idle 级别的信息，它的定义如代码清单 19.5 所示。

代码清单 19.5 cpuidle_state 结构体

```

1 struct cpuidle_state {
2     char name[CPUIDLE_NAME_LEN];
3     char desc[CPUIDLE_DESC_LEN];
4
5     unsigned int flags;
6     unsigned int exit_latency; /* in US */
7     int power_usage; /* in mW */
8     unsigned int target_residency; /* in US */
9     bool disabled; /* disabled on all CPUs */
10
11    int (*enter) (struct cpuidle_device *dev,
12                  struct cpuidle_driver *drv,
13                  int index);
14
15    int (*enter_dead) (struct cpuidle_device *dev, int index);
16 };

```

name 和 desc 是该 Idle 状态的名称和描述，exit_latency 是退出该 Idle 状态需要的延迟，enter() 是进入该 Idle 状态的实现方法。

忽略细节，一个具体的 SoC 的 CPUIdle 驱动实例可见于 arch/arm/mach-ux500/cpuidle.c（最新的内核已经将代码转移到了 drivers/cpuidle/cpuidle-ux500.c 中），它有两个 Idle 级别，即 WFI 和 ApIdle，其具体实现框架如代码清单 19.6 所示。

代码清单 19.6 ux500 CPUIdle 驱动案例

```

1 static atomic_t master = ATOMIC_INIT(0);
2 static DEFINE_SPINLOCK(master_lock);
3 static DEFINE_PER_CPU(struct cpuidle_device, ux500_cpuidle_device);
4
5 static inline int ux500_enter_idle(struct cpuidle_device *dev,
6                                   struct cpuidle_driver *drv, int index)
7 {
8     ...

```

```

9  }
10
11 static struct cpuidle_driver ux500_idle_driver = {
12   .name = "ux500_idle",
13   .owner = THIS_MODULE,
14   .en_core_tk_irqen = 1,
15   .states = {
16     ARM_CPUIDLE_WFI_STATE,
17     {
18       .enter      = ux500_enter_idle,
19       .exit_latency = 70,
20       .target_residency = 260,
21       .flags      = CPUIDLE_FLAG_TIME_VALID,
22       .name       = "ApIdle",
23       .desc       = "ARM Retention",
24     },
25   },
26   .safe_state_index = 0,
27   .state_count = 2,
28 };
29
30 /*
31  * For each cpu, setup the broadcast timer because we will
32  * need to migrate the timers for the states >= ApIdle.
33  */
34 static void ux500_setup_broadcast_timer(void *arg)
35 {
36   intcpu = smp_processor_id();
37   clockevents_notify(CLOCK_EVT_NOTIFY_BROADCAST_ON, &cpu);
38 }
39
40 int __init ux500_idle_init(void)
41 {
42 ...
43   ret = cpuidle_register_driver(&ux500_idle_driver);
44 ...
45   for_each_online_cpu(cpu) {
46     device = &per_cpu(ux500_cpuidle_device, cpu);
47     device->cpu = cpu;
48     ret = cpuidle_register_device(device);
49 ...
50   }
51 ...
52 }
53 device_initcall(ux500_idle_init);

```

与CPUFreq类似，在CPUIidle子系统中也有对应的governor来抉择何时进入何种Idle级别的策略，这些governor包括CPU_IDLE_GOV_LADDER、CPU_IDLE_GOV_MENU。LADDER在进入和退出Idle级别的时候是步进的，它以过去的Idle时间作为参考，而

MENU 总是根据预期的空闲时间直接进入目标 Idle 级别。前者适用于没有采用动态时间节拍的系统（即没有选择 NO_HZ 的系统），不依赖于 NO_HZ 配置选项，而后者依赖于内核的 NO_HZ 选项。

图 19.3 演示了 LADDER 步进从 C0 进入 C3，而 MENU 则可能直接从 C0 跳入 C3。

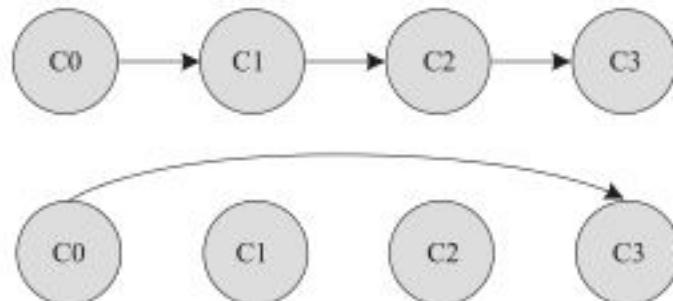


图 19.3 LADDER 与 MENU 的区别

CPUIdle 子系统还通过 sys 向 userspace 导出了一些节点：

- 一类是针对整个系统的 /sys/devices/system/cpu/cpuidle，通过其中的 current_driver、current_governor、available_governors 等节点可以获取或设置 CPUIdle 的驱动信息以及 governor。
- 一类是针对每个 CPU 的 /sys/devices/system/cpu/cpux/cpuidle，通过子节点暴露各个在线的 CPU 中每个不同 Idle 级别的 name、desc、power、latency 等信息。

综合以上的各个要素，可以给出 Linux CPUIdle 子系统的总体架构，如图 19.4 所示。

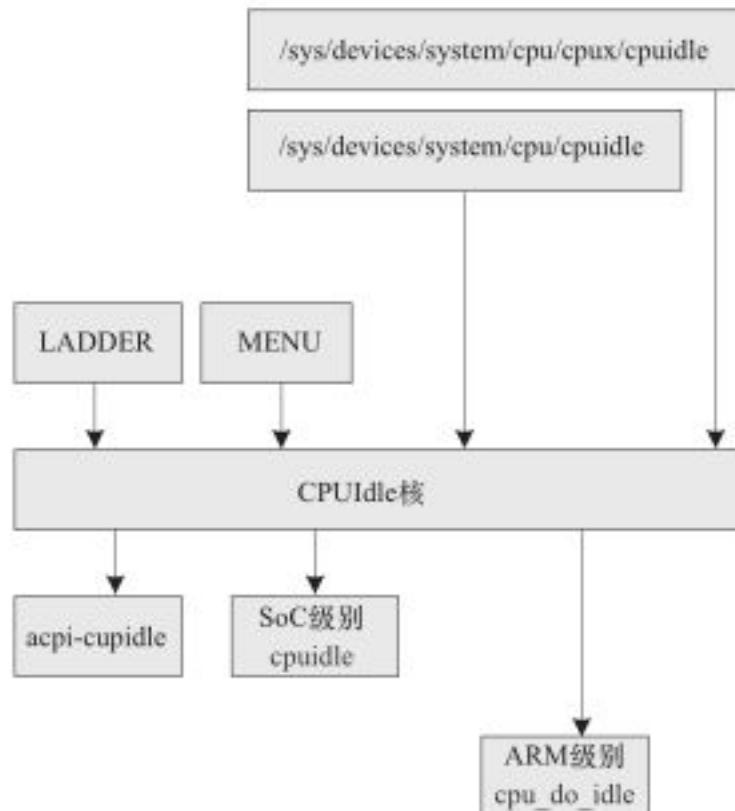


图 19.4 Linux CPUIdle 子系统的整体架构

19.4 PowerTop

PowerTop 是一款开源的用于进行电量消耗分析和电源管理诊断的工具，其主页位于 Intel 开源技术中心的 <https://01.org/powertop/>，维护者是 Arjan van de Ven 和 Kristen Accardi。PowerTop 可分析系统中软件的功耗，以便找到功耗大户，也可显示系统中不同的 C 状态（与 CPUIdle 驱动对应）和 P 状态（与 CPUFreq 驱动对应）的时间比例，并采用了基于 TAB 的界面风格，如图 19.5 所示。



图 19.5 PowerTOP

19.5 Regulator 驱动

Regulator 是 Linux 系统中电源管理的基础设施之一，用于稳压电源的管理，是各种驱动子系统中设置电压的标准接口。前面介绍的 CPUFreq 驱动就经常使用它来设定电压，比如代码清单 19.2 的第 57~59 行。

而 Regulator 则可以管理系统中的供电单元，即稳压器（Low Dropout Regulator, LDO，即低压差线性稳压器），并提供获取和设置这些供电单元电压的接口。一般在 ARM 电路板上，各个稳压器和设备会形成一个 Regulator 树形结构，如图 19.6 所示。

Linux 的 Regulator 子系统提供如下 API 以用于注册 / 注销一个稳压器：

```
struct regulator_dev * regulator_register(const struct regulator_desc *regulator_desc, const struct regulator_config *config);
void regulator_unregister(struct regulator_dev *rdev);
```

regulator_register() 函数的两个参数分别是 regulator_desc 结构体和 regulator_config 结构

体的指针。

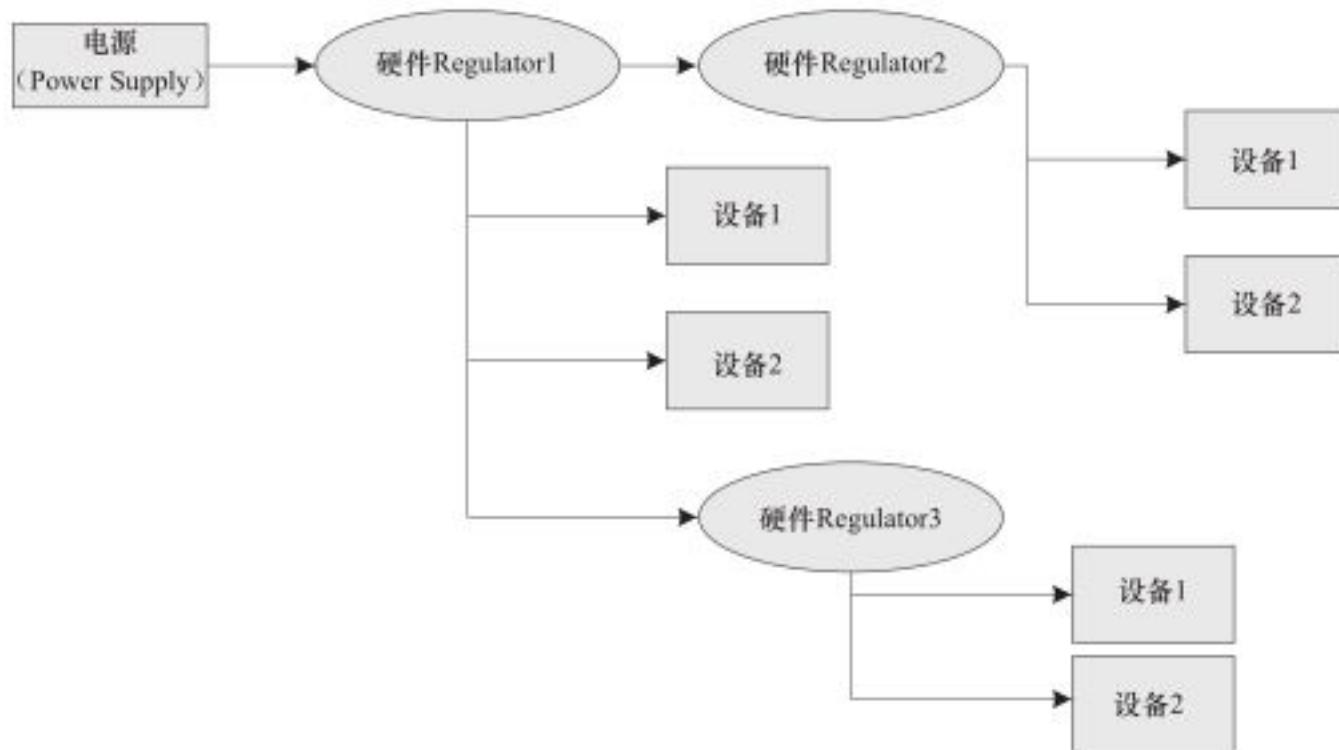


图 19.6 Regulator 树形结构

`regulator_desc` 结构体是对这个稳压器属性和操作的封装，如代码清单 19.7 所示。

代码清单 19.7 `regulator_desc` 结构体

```

1 struct regulator_desc {
2     const char *name;           /* Regulator 的名字 */
3     const char *supply_name;    /* Regulator Supply 的名字 */
4     int id;
5     unsigned n_voltages;
6     struct regulator_ops *ops;
7     int irq;
8     enum regulator_type type; /* 是电压还是电流 Regulator */
9     struct module *owner;
10
11    unsigned int min_uV;        /* 在线性映射情况下最低的 Selector 的电压 */
12    unsigned int uV_step;       /* 在线性映射情况下每步增加 / 减小的电压 */
13    unsigned int ramp_delay;   /* 电压改变后稳定下来所需时间 */
14
15    const unsigned int *volt_table; /* 基于表映射情况下的电压映射表 */
16
17    unsigned int vsel_reg;
18    unsigned int vsel_mask;
19    unsigned int enable_reg;
20    unsigned int enable_mask;
21    unsigned int bypass_reg;
22    unsigned int bypass_mask;
23
  
```

```

24         unsigned int enable_time;
25     };

```

上述结构体中的 regulator_ops 指针 ops 是对这个稳压器硬件操作的封装，其中包含获取、设置电压等的成员函数，如代码清单 19.8 所示。

代码清单 19.8 regulator_ops 结构体

```

1 struct regulator_ops {
2     /* enumerate supported voltages */
3     int (*list_voltage) (struct regulator_dev *, unsigned selector);
4
5     /* get/set regulator voltage */
6     int (*set_voltage) (struct regulator_dev *, int min_uV, int max_uV,
7                         unsigned *selector);
8     int (*map_voltage)(struct regulator_dev *, int min_uV, int max_uV);
9     int (*set_voltage_sel) (struct regulator_dev *, unsigned selector);
10    int (*get_voltage) (struct regulator_dev *);
11    int (*get_voltage_sel) (struct regulator_dev *);
12
13    /* get/set regulator current */
14    int (*set_current_limit) (struct regulator_dev *,
15                             int min_uA, int max_uA);
16    int (*get_current_limit) (struct regulator_dev *);
17
18    /* enable/disable regulator */
19    int (*enable) (struct regulator_dev *);
20    int (*disable) (struct regulator_dev *);
21    int (*is_enabled) (struct regulator_dev *);
22
23    ...
24 };

```

在 drivers/regulator 目录下，包含大量的与电源芯片对应的 Regulator 驱动，如 Dialog 的 DA9052、Intersil 的 ISL6271A、ST-Ericsson 的 TPS61050/61052、Wolfson 的 WM831x 系列等，它同时提供了一个虚拟的 Regulator 驱动作为参考，如代码清单 19.9 所示。

代码清单 19.9 虚拟的 Regulator 驱动

```

1 struct regulator_dev *dummy_regulator_rdev;
2 static struct regulator_init_data dummy_initdata;
3 static struct regulator_ops dummy_ops;
4 static struct regulator_desc dummy_desc = {
5     .name = "regulator-dummy",
6     .id = -1,
7     .type = REGULATOR_VOLTAGE,
8     .owner = THIS_MODULE,
9     .ops = &dummy_ops,
10 };

```

```

11
12 static int __devinit dummy_regulator_probe(struct platform_device *pdev)
13 {
14     struct regulator_config config = { };
15     int ret;
16
17     config.dev = &pdev->dev;
18     config.init_data = &dummy_initdata;
19
20     dummy_regulator_rdev = regulator_register(&dummy_desc, &config);
21     if (IS_ERR(dummy_regulator_rdev)) {
22         ret = PTR_ERR(dummy_regulator_rdev);
23         pr_err("Failed to register regulator: %d\n", ret);
24         return ret;
25     }
26
27     return 0;
28 }

```

Linux 的 Regulator 子系统提供消费者 (Consumer) API 以便让其他的驱动获取、设置、关闭和使能稳压器：

```

structregulator * regulator_get(structdevice *dev, const char *id);
structregulator * devm_regulator_get(structdevice *dev, const char *id);
structregulator * regulator_get_exclusive(structdevice *dev, const char *id);
voidregulator_put(structregulator *regulator);
voiddevm_regulator_put(structregulator *regulator);
intregulator_enable(structregulator *regulator);
intregulator_disable(structregulator *regulator);
intregulator_set_voltage(structregulator *regulator, intmin_uv, intmax_uv);
intregulator_get_voltage(structregulator *regulator);

```

这些消费者 API 的地位大致与 GPIO 子系统的 gpio_request()、时钟子系统的 clk_get()、dmaengine 子系统的 dmaengine_submit() 等相当，属于基础设施。

19.6 OPP

现今的 SoC 一般包含很多集成组件，在系统运行过程中，并不需要所有的模块都运行于最高频率和最高性能。在 SoC 内，某些 domain 可以运行在较低的频率和电压下，而其他 domain 可以运行在较高的频率和电压下，某个 domain 所支持的<频率，电压>对的集合被称为 Operating Performance Point，缩写为 OPP。

```
int opp_add(struct device *dev, unsigned long freq, unsigned long u_volt);
```

目前，TI OMAP CPUFreq 驱动的底层就使用了 OPP 这种机制来获取 CPU 所支持的频率和电压列表。在开机的过程中，TI OMAP4 芯片会注册针对 CPU 设备的 OPP 表（代码位于

arch/arm/mach-omap2/中), 如代码清单 19.10 所示。

代码清单 19.10 TI OMAP4 CPU 的 OPP 表

```

1 static struct omap_opp_def __initdata omap4xx_opp_def_list[] = {
2     /* MPU OPP1 - OPP50 */
3     OPP_INITIALIZER("mpu", true, 300000000, OMAP4430_VDD_MPU OPP50_UV),
4     /* MPU OPP2 - OPP100 */
5     OPP_INITIALIZER("mpu", true, 600000000, OMAP4430_VDD_MPU OPP100_UV),
6     /* MPU OPP3 - OPP-Turbo */
7     OPP_INITIALIZER("mpu", true, 800000000, OMAP4430_VDD_MPU_OPPTURBO_UV),
8     /* MPU OPP4 - OPP-SB */
9     OPP_INITIALIZER("mpu", true, 1008000000, OMAP4430_VDD_MPU OPPNITRO_UV),
10    ...
11 };
12 /**
13  * omap4_opp_init() - initialize omap4 opp table
14  */
15 int __init omap4_opp_init(void)
16 {
17     ...
18     r = omap_init_opp_table(omap4xx_opp_def_list,
19                         ARRAY_SIZE(omap4xx_opp_def_list));
20
21     return r;
22 }
23 device_initcall(omap4_opp_init);
24 int __init omap_init_opp_table(struct omap_opp_def *opp_def,
25                                u32 opp_def_size)
26 {
27     ...
28     /* Lets now register with OPP library */
29     for (i = 0; i < opp_def_size; i++, opp_def++) {
30         ...
31         if (!strcmp(opp_def->hwmod_name, "mpu", 3)) {
32             /*
33              * All current OMAPs share voltage rail and
34              * clock source, so CPU0 is used to represent
35              * the MPU-SS.
36              */
37             dev = get_cpu_device(0);
38         } ...
39         r = opp_add(dev, opp_def->freq, opp_def->u_volt);
40         ...
41     }
42     return 0;
43 }
```

针对与 device 结构体指针 dev 对应的 domain 中增加一个新的 OPP, 参数 freq 和 u_volt 即为该 OPP 对应的频率和电压。

```
int opp_enable(struct device *dev, unsigned long freq);
int opp_disable(struct device *dev, unsigned long freq);
```

上述 API 用于使能和禁止某个 OPP，一旦被禁止，其 available 将成为 false，之后有设备驱动想设置为这个 OPP 就不再可能了。譬如，当温度超过某个范围后，系统不允许 1GHz 的工作频率，可采用类似下面的代码实现：

```
if (cur_temp > temp_high_thresh) {
    /* Disable 1GHz if it was enabled */
    rCU_read_lock();
    opp = opp_find_freq_exact(dev, 1000000000, true);
    rCU_read_unlock();
    /* just error check */
    if (!IS_ERR(opp))
        ret = opp_disable(dev, 1000000000);
    else
        goto try_something_else;
}
```

上述代码中调用的 opp_find_freq_exact() 用于寻找与一个确定频率和 available 匹配的 OPP，其原型为：

```
struct opp *opp_find_freq_exact(struct device *dev, unsigned long freq,
                                bool available);
```

另外，Linux 还提供两个变体，opp_find_freq_floor() 用于寻找 1 个 OPP，它的频率向上接近或等于指定的频率；opp_find_freq_ceil() 用于寻找 1 个 OPP，它的频率向下接近或等于指定的频率，这两个函数的原型为：

```
struct opp *opp_find_freq_floor(struct device *dev, unsigned long *freq);
struct opp *opp_find_freq_ceil(struct device *dev, unsigned long *freq);
```

我们可用下面的代码分别寻找 1 个设备的最大和最小工作频率：

```
freq = ULONG_MAX;
rCU_read_lock();
opp_find_freq_floor(dev, &freq);
rCU_read_unlock();

freq = 0;
rCU_read_lock();
opp_find_freq_ceil(dev, &freq);
rCU_read_unlock();
```

在频率降低的同时，支撑该频率运行所需的电压也往往可以动态调低；反之，则可能需要调高，下面这两个 API 分别用于获取与某 OPP 对应的电压和频率：

```
unsigned long opp_get_voltage(struct opp *opp);
```

```
unsigned long opp_get_freq(struct opp *opp);
```

举个例子，当某 CPUFreq 驱动想将 CPU 设置为某一频率的时候，它可能会同时设置电压，其代码流程为：

```
soc_switch_to_freq_voltage(freq)
{
    /* do things */
    rCU_read_lock();
    opp = opp_find_freq_ceil(dev, &freq);
    v = opp_get_voltage(opp);
    rCU_read_unlock();
    if (v)
        regulator_set_voltage(..., v);
    /* do other things */
}
```

如下简单的 API 可用于获取某设备所支持的 OPP 的个数：

```
int opp_get_opp_count(struct device *dev);
```

前面提到，TI OMAP CPUFreq 驱动的底层就使用了 OPP 这种机制来获取 CPU 所支持的频率和电压列表。它在 omap_init_opp_table() 函数中添加了相应的 OPP，在 TI OMAP 芯片的 CPUFreq 驱动 drivers/cpufreq/omap-cpufreq.c 中，则借助了快捷函数 opp_init_cpufreq_table() 来根据前面注册的 OPP 建立 CPUFreq 的频率表：

```
static int __cpuinit omap_cpu_init(struct cpufreq_policy *policy)
{
    ...
    if (!freq_table)
        result = opp_init_cpufreq_table(mpu_dev, &freq_table);
    ...
}
```

而在 CPUFreq 驱动的目标成员函数 omap_target() 中，则使用与 OPP 相关的 API 来获取频率和电压：

```
static int omap_target(struct cpufreq_policy *policy,
                      unsigned int target_freq,
                      unsigned int relation)
{
    ...
    if (mpu_reg) {
        opp = opp_find_freq_ceil(mpu_dev, &freq);
        ...
        volt = opp_get_voltage(opp);
        ...
    }
    ...
}
```

drivers/cpufreq/omap-cpufreq.c 相对来说较为规范，它在<频率，电压>表方面，在底层使用了 OPP，在设置电压的时候又使用了规范的 Regulator API。

比较新的驱动一般不太喜欢直接在代码里面固化 OPP 表，而是喜欢在相应的节点处添加 operating-points 属性，如 imx27.dtso 中的：

```
cpus {
    #size-cells = <0>;
    #address-cells = <1>;

    cpu: cpu@0 {
        device_type = "cpu";
        compatible = "arm,arm926ej-s";
        operating-points = <
            /* kHz uV */
            266000 1300000
            399000 1450000
        >;
        clock-latency = <62500>;
        clocks = <&clks IMX27_CLK_CPU_DIV>;
        voltage-tolerance = <5>;
    };
};
```

如果 CPUFreq 的变化可以使用非常标准的 regulator、clk API，我们甚至可以直接使用 drivers/cpufreq/cpufreq-dt.c 这个驱动。这样只需要在 CPU 节点上填充好频率电压表，然后在平台代码里面注册 cpufreq-dt 设备就可以了，在 arch/arm/mach-imx/imx27-dt.c、arch/arm/mach-imx/mach-imx51.c 中可以找到类似的例子：

```
static void __init imx27_dt_init(void)
{
    struct platform_device_info devinfo = { .name = "cpufreq-dt", };

    of_platform_populate(NULL, of_default_bus_match_table, NULL, NULL);

    platform_device_register_full(&devinfo);
}
```

19.7 PM QoS

Linux 内核的 PM QoS 系统针对内核和应用程序提供了一套接口，通过这个接口，用户可以设定自身对性能的期望。一类是系统级的需求，通过 `cpu_dma_latency`、`network_latency` 和 `network_throughput` 这些参数来设定；另一类是单个设备可以根据自身的性能需求发起 per-device 的 PM QoS 请求。

在内核空间，通过 pm_qos_add_request() 函数可以注册 PM QoS 请求：

```
void pm_qos_add_request(struct pm_qos_request *req,
int pm_qos_class, s32 value);
```

通过 pm_qos_update_request() 函数可以更新已注册的 PM QoS 请求：

```
void pm_qos_update_request(struct pm_qos_request *req,
                           s32 new_value);
void pm_qos_update_request_timeout(struct pm_qos_request *req, s32 new_value,
                                   unsigned long timeout_us);
```

通过 pm_qos_remove_request() 函数可以删除已注册的 PM QoS 请求：

```
void pm_qos_remove_request(struct pm_qos_request *req);
```

譬如在 drivers/media/platform/via-camera.c 这个摄像头驱动中，当摄像头开启后，通过如下语句可以阻止 CPU 进入 C3 级别的深度 Idle：

```
static int viacam_streamon(struct file *filp, void *priv, enum v4l2_buf_type t)
{
    ...
    pm_qos_add_request(&cam->qos_request, PM_QOS_CPU_DMA_LATENCY, 50);
    ...
}
```

这是因为，在CPUidle 子系统中，会根据 PM_QOS_CPU_DMA_LATENCY 请求的情况选择合适的 C 状态，如 drivers/cpuidle/governors/ladder.c 中的 ladder_select_state() 就会判断目标 C 状态的 exit_latency 与 QoS 要求的关系，如代码清单 19.11 所示。

代码清单 19.11 CPUidle LADDER governor 对 QoS 的判断

```
1 static int ladder_select_state(struct cpuidle_driver *drv,
2                                struct cpuidle_device *dev)
3 {
4     ...
5     int latency_req = pm_qos_request(PM_QOS_CPU_DMA_LATENCY);
6
7     ...
8
9     /* consider promotion */
10    if (last_idx < drv->state_count - 1 &&
11        !drv->states[last_idx + 1].disabled &&
12        !dev->states_usage[last_idx + 1].disable &&
13        last_residency > last_state->threshold.promotion_time &&
14        drv->states[last_idx + 1].exit_latency <= latency_req) {
15        last_state->stats.promotion_count++;
16        last_state->stats.demotion_count = 0;
17        if(last_state->stats.promotion_count>=
18            last_state->threshold.promotion_count) {
```

```

19             ladder_do_selection(ldev, last_idx, last_idx + 1);
20             return last_idx + 1;
21         }
22     ]
23     ...
24 }

```

LADDER 在选择是否进入更深层次的 C 状态时，会比较 C 状态的 exit_latency 要小于通过 pm_qos_request(PM_QOS_CPU_DMA_LATENCY) 得到的 PM QoS 请求的延迟，见代码清单 19.11 的第 14 行。

同样的逻辑也出现于 drivers/cpuidle/governors/menu.c 中，如代码清单 19.12 的第 18~19 行。

代码清单 19.12 CPUIidle MENU governor 对 QoS 的判断

```

1 static int menu_select(struct cpuidle_driver *drv, struct cpuidle_device *dev)
2 {
3     struct menu_device *data = &__get_cpu_var(menu_devices);
4     int latency_req = pm_qos_request(PM_QOS_CPU_DMA_LATENCY);
5     ...
6     /*
7      * Find the idle state with the lowest power while satisfying
8      * our constraints.
9      */
10    for (i = CPUIDLE_DRIVER_STATE_START; i < drv->state_count; i++) {
11        struct cpuidle_state *s = &drv->states[i];
12        struct cpuidle_state_usage *su = &dev->states_usage[i];
13
14        if (s->disabled || su->disable)
15            continue;
16        if (s->target_residency > data->predicted_us)
17            continue;
18        if (s->exit_latency > latency_req)
19            continue;
20        if (s->exit_latency * multiplier > data->predicted_us)
21            continue;
22
23        if (s->power_usage < power_usage) {
24            power_usage = s->power_usage;
25            data->last_state_idx = i;
26            data->exit_us = s->exit_latency;
27        }
28    }
29
30    return data->last_state_idx;
31 }

```

还是回到 drivers/media/platform/via-camera.c 中，当摄像头关闭后，它会通过如下语句告知上述代码对 PM_QOS_CPU_DMA_LATENCY 的性能要求取消：

```

static int viacam_streamon(struct file *filp, void *priv, enum v4l2_buf_type t)
{
    ...
    pm_qos_remove_request(&cam->qos_request);
    ...
}

```

类似的在设备驱动中申请 QoS 特性的例子还包括 drivers/net/wireless/ipw2x00/ipw2100.c、drivers/tty/serial/omap-serial.c、drivers/net/ethernet/intel/e1000e/netdev.c 等。

应用程序则可以通过向 /dev/cpu_dma_latency 和 /dev/network_latency 这样的设备节点写入值来发起 QoS 的性能请求。

19.8 CPU 热插拔

Linux CPU 热插拔的功能已经存在相当长的时间了，Linux 3.8 之后的内核里一个小小的改进就是 CPU0 也可以热插拔。

一般来讲，在用户空间可以通过 /sys/devices/system/cpu/cpun/online 节点来操作一个 CPU 的在线和离线：

```

# echo 0>/sys/devices/system/cpu/cpu3/online
CPU 3 is now offline
# echo 1>/sys/devices/system/cpu/cpu3/online

```

通过 echo 0>/sys/devices/system/cpu/cpu3/online 关闭 CPU3 的时候，CPU3 上的进程都会被迁移到其他的 CPU 上，以保证这个拔除 CPU3 的过程中，系统仍然能正常运行。一旦通过 echo 1>/sys/devices/system/cpu/cpu3/online 再次开启 CPU3，CPU3 又可以参与系统的负载均衡，分担系统中的任务。

在嵌入式系统中，CPU 热插拔可以作为一种省电的方式，在系统负载小的时候，动态关闭 CPU，在系统负载增大的时候，再开启之前离线的 CPU。目前各个芯片公司可能会根据自身 SoC 的特点，对内核进行调整，来实现运行时“热插拔”。这里以 Nvidia 的 Tegra3 为例进行说明。

Tegra3 采用 vSMP (variableSymmetric Multiprocessing) 架构，共有 5 个 Cortex-A9 处理器，其中 4 个为高性能设计的 G 核，1 个为低功耗设计的 LP 核，如图 19.7 所示。

在系统运行过程中，Tegra3 的 Linux 内核



图 19.7 Tegra3 的架构

会根据 CPU 负载切换低功耗处理器和高功耗处理器。除此之外，4 个高性能 ARM 核心也会根据运行情况，动态借用 Linux 内核支持的 CPU 热插拔进行 CPU 的插入 / 拔出操作。

用华硕 EeePad 运行高负载、低负载应用，通过 dmesg 查看内核消息也确实验证了多核的热插拔以及 G 核和 LP 核之间的动态切换：

```
<4>[104626.426957] CPU1: Booted secondary processor
<7>[104627.427412] tegra CPU: force EDP limit 720000 kHz
<4>[104627.427670] CPU2: Booted secondary processor
<4>[104628.537005] stop_machine_cpu_stop cpu=0
<4>[104628.537017] stop_machine_cpu_stop cpu=2
<4>[104628.537059] stop_machine_cpu_stop cpu=1
<4>[104628.537702] __stop_cpus: wait_for_completion_timeout+
<4>[104628.537810] __stop_cpus: smp=0 done.executed=1 done.ret =0-
<5>[104628.537960] CPU1: clean shutdown
<4>[104630.537092] stop_machine_cpu_stop cpu=0
<4>[104630.537172] stop_machine_cpu_stop cpu=2
<4>[104630.537739] __stop_cpus: wait_for_completion_timeout+
<4>[104630.538060] __stop_cpus: smp=0 done.executed=1 done.ret =0-
<5>[104630.538203] CPU2: clean shutdown
<4>[104631.306984] tegra_watchdog_touch
```

高性能处理器和低功耗处理器切换：

```
<3>[104666.799152] LP=>G: prolog 22 us, switch 2129 us, epilog 24 us, total 2175 us
<3>[104667.807273] G=>LP: prolog 18 us, switch 157 us, epilog 25 us, total 200 us
<4>[104671.407008] tegra_watchdog_touch
<4>[104671.408816] nct1008_get_temp: ret temp=35C
<3>[104671.939060] LP=>G: prolog 17 us, switch 2127 us, epilog 22 us, total 2166 us
<3>[104672.938091] G=>LP: prolog 18 us, switch 156 us, epilog 24 us, total 198 us
```

在运行过程中，我们发现 4 个 G 核会动态热插拔，而 4 个 G 核和 1 个 LP 核之间，会根据运行负载进行集群切换。这一部分都是在内核里面实现的，和 tegra 的 CPUF req 驱动（DVFS 驱动）紧密关联。相关代码可见于 <http://nv-tegra.nvidia.com/gitweb/?p=linux-2.6.git;a=tree;f=arch/arm/mach-tegra;h=e5d1ff2;hb=rel-14r7>

1. 如何判断自己是什么核

每个核都可以通过调用 is_lp_cluster() 来判断当前正在执行的 CPU 是 LP 还是 G 处理器：

```
static inline unsigned int is_lp_cluster(void)
{
    unsigned int reg;
    reg = readl(FLOW_CTRL_CLUSTER_CONTROL);
    return (reg & 1); /* 0 == G, 1 == LP */
}
```

即读 FLOW_CTRL_CLUSTER_CONTROL 寄存器判断自己是 G 核还是 LP 核。

2. G 核和 LP 核集群的切换时机

[场景 1] 何时从 LP 核切换给 G 核：当前执行于 LP 集群，CPUFreq 驱动判断出 LP 核需要增频率到超过高值门限，即 TEGRA_HP_UP；

```

caseTEGRA_HP_UP:
    if(is_lp_cluster() && !no_lp) {
        if(!clk_set_parent(cpu_clk, cpu_g_clk)) {
            hp_stats_update(CONFIG_NR_CPUS, false);
            hp_stats_update(0, true);
            /* catch-upwith governor target speed */
            tegra_cpu_set_speed_cap(NULL);
        }
    }

```

[场景2] 何时从G核切换给LP核：当前执行于G集群，CPUFreq驱动判断出某G核需要降频率到低于低值门限，即TEGRA_HP_DOWN，且最慢的CPUID不小于nr_cpu_ids（实际上代码逻辑跟踪等价于只有CPU0还活着的情况）：

```

caseTEGRA_HP_DOWN:
    cpu= tegra_get_slowest_cpu_n();
    if(cpu < nr_cpu_ids) {
        ...
    }else if (!is_lp_cluster() && !no_lp) {
        if(!clk_set_parent(cpu_clk, cpu_lp_clk)) {
            hp_stats_update(CONFIG_NR_CPUS, true);
            hp_stats_update(0, false);
            /* catch-upwith governor target speed */
            tegra_cpu_set_speed_cap(NULL);
        } else
            queue_delayed_work(
                hotplug_wq, &hotplug_work, down_delay);
    }
    break;

```

切换实际上就发生在clk_set_parent()更改CPU的父时钟里面，这部分代码写得比较不好，1个函数完成n个功能，实际上不仅切换了时钟，还切换了G和LP集群：

```

clk_set_parent(cpu_clk, cpu_lp_clk) ->
    tegra3_cpu_cmplx_clk_set_parent(struct clk *c, struct clk *p) ->
        tegra_cluster_control(unsigned int us, unsigned int flags) ->
            tegra_cluster_switch_prolog()->
tegra_cluster_switch_epilog()

```

3. G核动态热插拔

何时进行G核的动态插拔，具体如下。

[场景3] 当前执行于G集群，CPUFreq驱动判断出某G核需要降频率到低于低值门限，即TEGRA_HP_DOWN，且最慢的CPUID小于nr_cpu_ids（实际上等价于还有两个或两个以上的G核活着的情况），关闭最慢的CPU，留意tegra_get_slowest_cpu_n()不会返回0，这意味着CPU0要么活着，要么切换给了LP核，对应于[场景2]：

```

caseTEGRA_HP_DOWN:
    cpu= tegra_get_slowest_cpu_n();
    if(cpu < nr_cpu_ids) {
        up = false;
        queue_delayed_work(

```

```

        hotplug_wq, &hotplug_work, down_delay);
hp_stats_update(cpu, false);
}

```

[场景4] 当前执行于G集群，CPUFreq驱动判断出某G核需要设置频率大于高值门限，即TEGRA_HP_UP，如果负载平衡状态为TEGRA_CPU_SPEED_BALANCED，再开一个核；如果状态为TEGRA_CPU_SPEED_SKewed，则关一个核。TEGRA_CPU_SPEED_BALANCED的含义是当前所有G核要求的频率都高于最高频率的50%，TEGRA_CPU_SPEED_SKewed的含义是当前至少有两个G核要求的频率低于门限的25%，即CPU频率的要求在各个核之间有倾斜。

```

caseTEGRA_HP_UP:
    if(is_lp_cluster() && !no_lp) {
        ...
    }else {
        switch (tegra_cpu_speed_balance()) {
            /* cpu speed is up and balanced - one more on-line */
            case TEGRA_CPU_SPEED_BALANCED:
                cpu =cpumask_next_zero(0, cpu_online_mask);
                if (cpu < nr_cpu_ids) {
                    up =true;
                    hp_stats_update(cpu, true);
                }
                break;
            /* cpu speed is up, but skewed - remove one core */
            case TEGRA_CPU_SPEED_SKewed:
                cpu =tegra_get_slowest_cpu_n();
                if (cpu < nr_cpu_ids) {
                    up =false;
                    hp_stats_update(cpu, false);
                }
                break;
            /* cpu speed is up, but under-utilized - do nothing */
            case TEGRA_CPU_SPEED_BIASED:
            default:
                break;
        }
    }
}

```

上述代码中TEGRA_CPU_SPEED_BIASED路径的含义是有1个以上G核的频率低于最高频率的50%但是还未形成“SKewed”条件，即只是“BIASED”，还没有达到“SKewed”的程度，因此暂时什么都不做。

目前，ARM和Linux社区都在从事关于big.LITTLE架构下，CPU热插拔以及调度器方面有针对性的改进工作。在big.LITTLE架构中，将高性能且功耗也较高的Cortex-A15和稍低性能且功耗低的Cortex-A7进行了结合，或者在64位下，进行Cortex-A57和Cortex-A53的组合，如图19.8所示。

big.LITTLE架构的设计旨在为适当的作业分配恰当的处理器。Cortex-A15处理器是目前已开发的性能最高的低功耗ARM处理器，而Cortex-A7处理器是目前已开发的最节能的

ARM 应用程序处理器。可以利用 Cortex-A15 处理器的性能来承担繁重的工作负载，而用 Cortex-A7 可以最有效地处理智能手机的大部分工作负载。这些操作包括操作系统活动、用户界面和其他持续运行、始终连接的任务。

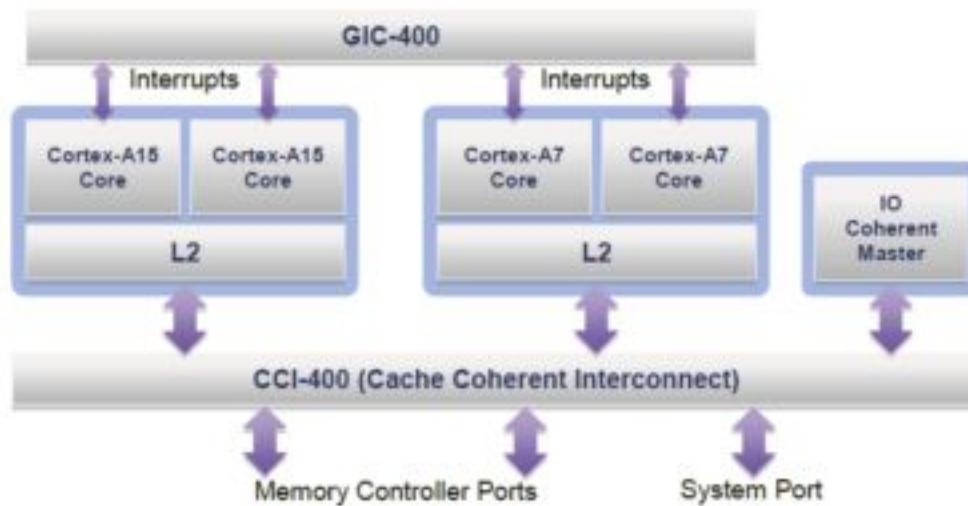


图 19.8 ARM 的 big.LITTLE 架构

三星在 2013 年 CES（国际消费电子展）大会上发布了 Exynos 5 Octa 8 核移动处理器，这款处理器也是采用 big.LITTLE 架构的第一款 CPU。

19.9 挂起到 RAM

Linux 支持 STANDBY、挂起到 RAM、挂起到硬盘等形式的待机，如图 19.9 所示。一般的嵌入式产品仅仅只实现了挂起到 RAM（也简称为 s2ram，或常简称为 STR），即将系统的状态保存于内存中，并将 SDRAM 置于自刷新状态，待用户按键等操作后再重新恢复系统。少数嵌入式 Linux 系统会实现挂起到硬盘（简称 STD），它与挂起到 RAM 的不同是 s2ram 并不关机，STD 则把系统的状态保持于磁盘，然后关闭整个系统。

在 Linux 下，这些行为通常是由用户空间触发的，通过向 /sys/power/state 写入 mem 可开始挂起到 RAM 的流程。当然，许多 Linux 产品会有一个按键，一按就进入挂起到 RAM。这通常是由于与这个按键对应的输入设备驱动汇报了一个和电源相关的 input_event，

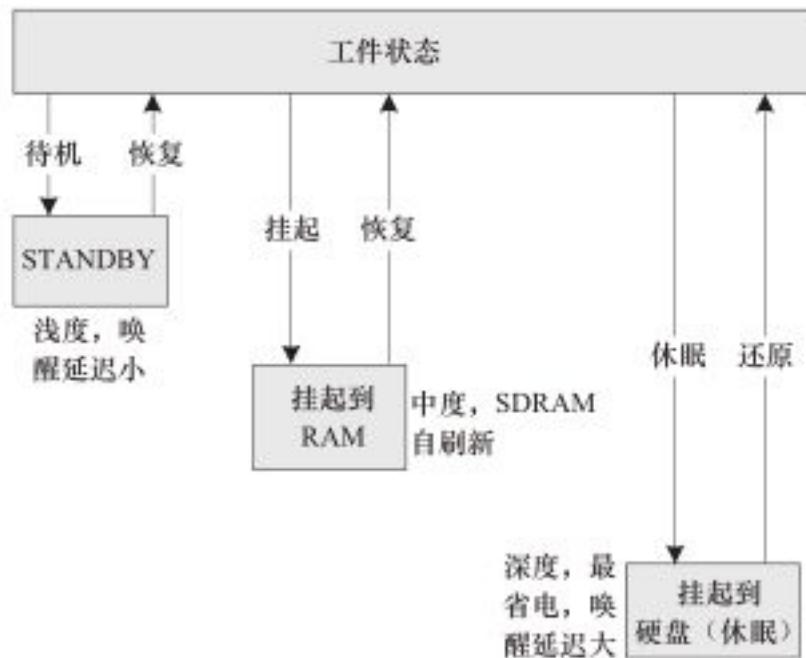


图 19.9 Linux 的待机模式

用户空间的电源管理 daemon 进程收到这个事件后，再触发 s2ram 的。当然，内核也有一个 INPUT_APMPOWER 驱动，位于 drivers/input/apm-power.c 下，它可以在内核级别侦听 EV_PWR 类事件，并通过 `apm_queue_event(APM_USER_SUSPEND)` 自动引发 s2ram：

```
static void system_power_event(unsigned int keycode)
{
    switch (keycode) {
    case KEY_SUSPEND:
        apm_queue_event(APM_USER_SUSPEND);
        pr_info("Requesting system suspend...\n");
        break;
    default:
        break;
    }
}

static void apmpower_event(struct input_handle *handle, unsigned int type,
                           unsigned int code, int value)
{
    ...
    switch (type) {
    case EV_PWR:
        system_power_event(code);
        break;
    ...
    }
}
```

在 Linux 内核中，大致的挂起到 RAM 的挂起和恢复流程如图 19.10 所示（牵涉的操作包括同步文件系统、freeze 进程、设备驱动挂起以及系统的挂起入口等）。

在 Linux 内核的 `device_driver` 结构中，含有一个 `pm` 成员，它是一个 `dev_pm_ops` 结构体指针，在该结构体中，封装了挂起到 RAM 和挂起到硬盘所需要的回调函数，如代码清单 19.13 所示。

代码清单 19.13 `dev_pm_ops` 结构体

```
1 struct dev_pm_ops {
2     int (*prepare)(struct device *dev);
3     void (*complete)(struct device *dev);
4     int (*suspend)(struct device *dev);
5     int (*resume)(struct device *dev);
6     int (*freeze)(struct device *dev);
7     int (*thaw)(struct device *dev);
8     int (*poweroff)(struct device *dev);
9     int (*restore)(struct device *dev);
10    int (*suspend_late)(struct device *dev);
11    int (*resume_early)(struct device *dev);
12    int (*freeze_late)(struct device *dev);
```

```

13     int (*thaw_early)(struct device *dev);
14     int (*poweroff_late)(struct device *dev);
15     int (*restore_early)(struct device *dev);
16     int (*suspend_noirq)(struct device *dev);
17     int (*resume_noirq)(struct device *dev);
18     int (*freeze_noirq)(struct device *dev);
19     int (*thaw_noirq)(struct device *dev);
20     int (*poweroff_noirq)(struct device *dev);
21     int (*restore_noirq)(struct device *dev);
22     int (*runtime_suspend)(struct device *dev);
23     int (*runtime_resume)(struct device *dev);
24     int (*runtime_idle)(struct device *dev);
25 );

```

图 19.10 实际上也给出了在挂起到 RAM 的时候这些 PM 回调函数的调用时机。

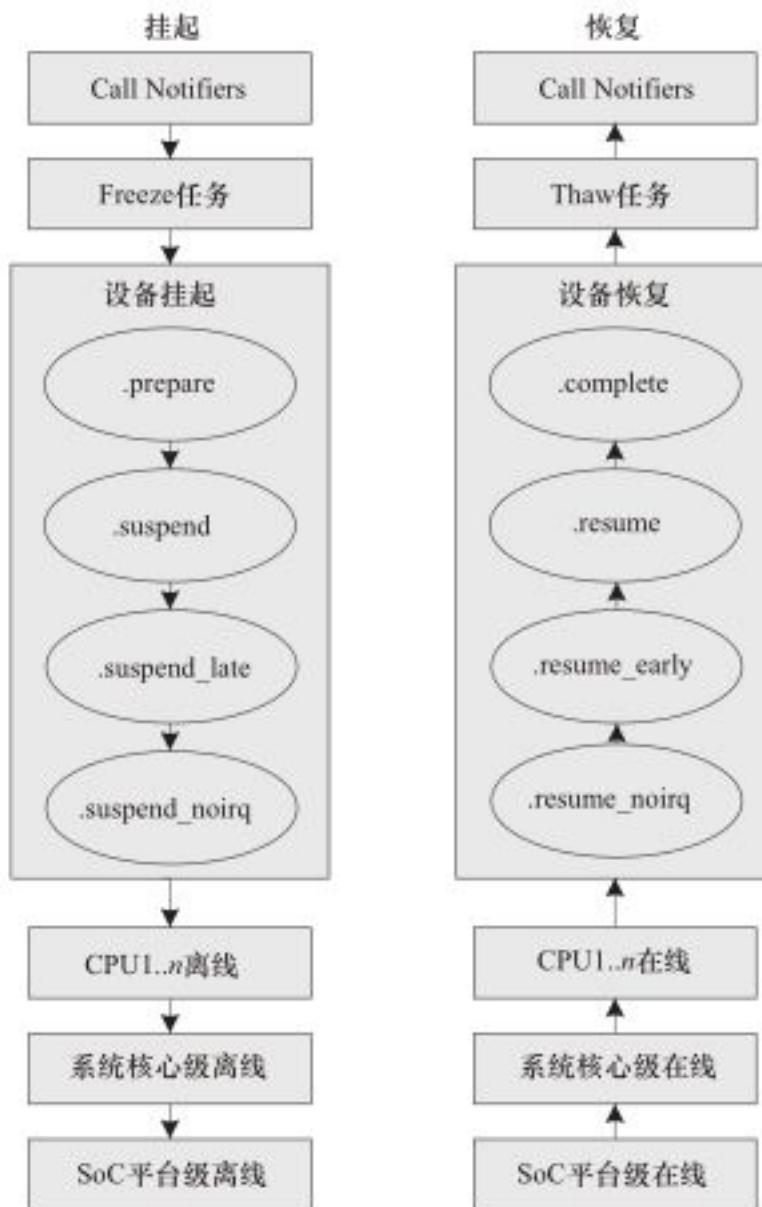


图 19.10 Linux 挂起到 RAM 流程

目前比较推荐的做法是在 platform_driver、i2c_driver 和 spi_driver 等 xxx_driver 结构体

实例的 driver 成员中，以上述结构体的形式封装 PM 回调函数，并赋值到 driver 的 pm 字段。如代码清单 19.14 中的第 50 行，在 drivers/spi/ spi-s3c64xx.c 中，platform_driver 中的 pm 成员被赋值。

代码清单 19.14 设备驱动中的 pm 成员

```

1 #ifdef CONFIG_PM_SLEEP
2 static int s3c64xx_spi_suspend(struct device *dev)
3 {
4     struct spi_master *master = dev_get_drvdata(dev);
5     struct s3c64xx_spi_driver_data *sdd = spi_master_get_devdata(master);
6
7     int ret = spi_master_suspend(master);
8     if (ret)
9         return ret;
10
11    if (!pm_runtime_suspended(dev)) {
12        clk_disable_unprepare(sdd->clk);
13        clk_disable_unprepare(sdd->src_clk);
14    }
15
16    sdd->cur_speed = 0; /* 输出时钟停止 */
17
18    return 0;
19 }
20
21 static int s3c64xx_spi_resume(struct device *dev)
22 {
23     struct spi_master *master = dev_get_drvdata(dev);
24     struct s3c64xx_spi_driver_data *sdd = spi_master_get_devdata(master);
25     struct s3c64xx_spi_info *sci = sdd->cntrlr_info;
26
27     if (sci->cfg_gpio)
28         sci->cfg_gpio();
29
30     if (!pm_runtime_suspended(dev)) {
31         clk_prepare_enable(sdd->src_clk);
32         clk_prepare_enable(sdd->clk);
33     }
34
35     s3c64xx_spi_hwinit(sdd, sdd->port_id);
36
37     return spi_master_resume(master);
38 }
39 #endif /* CONFIG_PM_SLEEP */
40
41 static const struct dev_pm_ops s3c64xx_spi_pm = {
42     SET_SYSTEM_SLEEP_PM_OPS(s3c64xx_spi_suspend, s3c64xx_spi_resume)
43     SET_RUNTIME_PM_OPS(s3c64xx_spi_runtime_suspend,
44                         s3c64xx_spi_runtime_resume, NULL)
45 };
46

```

```

47 static struct platform_driver s3c64xx_spi_driver = {
48     .driver = {
49         .name      = "s3c64xx-spi",
50         .pm        = &s3c64xx_spi_pm,
51         .of_match_table = of_match_ptr(s3c64xx_spi_dt_match),
52     },
53     .probe    = s3c64xx_spi_probe,
54     .remove   = s3c64xx_spi_remove,
55     .id_table = s3c64xx_spi_driver_ids,
56 };

```

s3c64xx_spi_suspend() 完成了时钟的禁止，s3c64xx_spi_resume() 则完成了硬件的重新初始化、时钟的使能等工作。第 42 行的 SET_SYSTEM_SLEEP_PM_OPS 是一个快捷宏，它完成 suspend、resume 等成员函数的赋值：

```

#define SET_SYSTEM_SLEEP_PM_OPS(suspend_fn, resume_fn) \
    .suspend = suspend_fn, \
    .resume = resume_fn, \
    .freeze = suspend_fn, \
    .thaw = resume_fn, \
    .poweroff = suspend_fn, \
    .restore = resume_fn,

```

除了上述推行的做法以外，在 platform_driver、i2c_driver、spi_driver 等 xxx_driver 结构体中仍然保留了过时（Legacy）的 suspend、resume 入口函数（目前不再推荐使用过时的接口，而是推荐赋值 xxx_driver 中的 driver 的 pm 成员），譬如代码清单 19.15 给出的 platform_driver 就包含了过时的 suspend、resume 等。

代码清单 19.15 设备驱动中过时的 PM 成员函数

```

1 struct platform_driver {
2     int (*probe)(struct platform_device *);
3     int (*remove)(struct platform_device *);
4     void (*shutdown)(struct platform_device *);
5     int (*suspend)(struct platform_device *, pm_message_t state);
6     int (*resume)(struct platform_device *);
7     struct device_driver driver;
8     const struct platform_device_id *id_table;
9 };

```

在 Linux 的核心层中，实际上是优先选择执行 xxx_driver.driver.pm.suspend() 成员函数，在前者不存在的情况下，执行过时的 xxx_driver.suspend()，如 platform_pm_suspend() 的逻辑如代码清单 19.16 所示。

代码清单 19.16 驱动核心层寻找 PM 回调的顺序

```

1 int platform_pm_suspend(struct device *dev)
2 {

```

```

3     struct device_driver *drv = dev->driver;
4     int ret = 0;
5
6     if (!drv)
7         return 0;
8
9     if (drv->pm) {
10         if (drv->pm->suspend)
11             ret = drv->pm->suspend(dev);
12     } else {
13         ret = platform_legacy_suspend(dev, PMSG_SUSPEND);
14     }
15
16     return ret;
17 }

```

一般来讲，在设备驱动的挂起入口函数中，会关闭设备、关闭该设备的时钟输入，甚至是关闭设备的电源，在恢复时则完成相反的操作。在挂起到 RAM 的挂起和恢复过程中，系统恢复后要求所有设备的驱动都工作正常。为了调试这个过程，可以使能内核的 PM_DEBUG 选项，如果想在挂起和恢复的过程中，看到内核的打印信息以关注具体的详细流程，可以在 Bootloader 传递给内核的 bootargs 中设置标志 no_console_suspend。

在将 Linux 移植到一个新的 ARM SoC 的过程中，最终系统挂起的入口需由芯片供应商在相应的 arch/arm/mach-xxx 中实现 platform_suspend_ops 的成员函数，一般主要实现其中的 enter 和 valid 成员，并将整个 platform_suspend_ops 结构体通过内核通用 API suspend_set_ops() 注册进系统，如 arch/arm/mach-prima2/pm.c 中 prima2 SoC 级挂起流程的逻辑如代码清单 19.17 所示。

代码清单 19.17 系统挂起到 RAM 的 SoC 级代码

```

1 static int sirfsoc_pm_enter(suspend_state_t state)
2 {
3     switch (state) {
4     case PM_SUSPEND_MEM:
5         sirfsoc_pre_suspend_power_off();
6
7         outer_flush_all();
8         outer_disable();
9         /* go zzz */
10        cpu_suspend(0, sirfsoc_finish_suspend);
11        outer_resume();
12        break;
13    default:
14        return -EINVAL;
15    }
16    return 0;
17 }

```

```

18
19 static const struct platform_suspend_ops sirfsoc_pm_ops = {
20     .enter = sirfsoc_pm_enter,
21     .valid = suspend_valid_only_mem,
22 };
23
24 int __init sirfsoc_pm_init(void)
25 {
26     suspend_set_ops(&sirfsoc_pm_ops);
27     return 0;
28 }

```

上述代码中第5行的 `sirfsoc_pre_suspend_power_off()` 的实现如代码清单 19.18 所示，它会将系统恢复回来后重新开始执行的物理地址存入与 SoC 相关的寄存器中。与本例对应的寄存器为 `SIRFSOC_PWRC_SCRATCH_PADI`，并向该寄存器写入了 `virt_to_phys(cpu_resume)`。在系统重新恢复中，会执行 `cpu_resume` 这段汇编，并进行设置唤醒源等操作。

代码清单 19.18 SoC 设置恢复回来时的执行地址

```

1 static int sirfsoc_pre_suspend_power_off(void)
2 {
3     u32 wakeup_entry = virt_to_phys(cpu_resume);
4
5     sirfsoc_rtc_iobrg_writel(wakeup_entry, sirfsoc_pwrc_base +
6                               SIRFSOC_PWRC_SCRATCH_PADI);
7
8     sirfsoc_set_wakeup_source();
9
10    sirfsoc_set_sleep_mode(SIRFSOC_DEEP_SLEEP_MODE);
11
12    return 0;
13 }

```

而 `cpu_suspend(0, sirfsoc_finish_suspend)` 以及其中调用的与 SoC 相关的用汇编实现的函数 `sirfsoc_finish_suspend()` 真正完成最后的待机并将系统置于深度睡眠，同时置 SDRAM 于自刷新状态的过程，具体的代码高度依赖于特定的芯片，其实现一般是一段汇编。

19.10 运行时的 PM

在前文给出的 `dev_pm_ops` 结构体中，有 3 个以 `runtime` 开头的成员函数：`runtime_suspend()`、`runtime_resume()` 和 `runtime_idle()`，它们辅助设备完成运行时的电源管理：

```

struct dev_pm_ops {
    ...
    int (*runtime_suspend)(struct device *dev);
    int (*runtime_resume)(struct device *dev);
}

```

```

    int (*runtime_idle)(struct device *dev);
    ...
};
```

运行时的 PM 与前文描述的系统级挂起到 RAM 时候的 PM 不太一样，它是针对单个设备，指系统在非睡眠状态的情况下，某个设备在空闲时可以进入运行时挂起状态，而在不是空闲时执行运行时恢复使得设备进入正常工作状态，如此，这个设备在运行时会省电。Linux 运行时 PM 最早是在 Linux2.6.32 内核中被合并的。

Linux 提供了一系列 API，以便于设备可以声明自己的运行时 PM 状态：

```
int pm_runtime_suspend(struct device *dev);
```

引发设备的挂起，执行相关的 runtime_suspend() 函数。

```
int pm_schedule_suspend(struct device *dev, unsigned int delay);
```

“调度”设备的挂起，延迟 delay 毫秒后将挂起工作挂入 pm_wq 等待队列，结果等价于 delay 毫秒后执行相关的 runtime_suspend() 函数。

```
int pm_request_autosuspend(struct device *dev);
```

“调度”设备的挂起，自动挂起的延迟到后，挂起的工作项目被自动放入队列。

```
int pm_runtime_resume(struct device *dev);
```

引发设备的恢复，执行相关的 runtime_resume() 函数。

```
int pm_request_resume(struct device *dev);
```

发起一个设备恢复的请求，该请求也是挂入 pm_wq 等待队列。

```
int pm_runtime_idle(struct device *dev);
```

引发设备的空闲，执行相关的 runtime_idle() 函数。

```
int pm_request_idle(struct device *dev);
```

发起一个设备空闲的请求，该请求也是挂入 pm_wq 等待队列。

```
void pm_runtime_enable(struct device *dev);
```

使能设备的运行时 PM 支持。

```
int pm_runtime_disable(struct device *dev);
```

禁止设备的运行时 PM 支持。

```
int pm_runtime_get(struct device *dev);
```

```
int pm_runtime_get_sync(struct device *dev);
```

增加设备的引用计数 (usage_count)，这类似于 clk_get()，会间接引发设备的 runtime_

resume()。

```
int pm_runtime_put(struct device *dev);
int pm_runtime_put_sync(struct device *dev)
```

减小设备的引用计数，这类似于 clk_put()，会间接引发设备的 runtime_idle()。

我们可以这样简单地理解 Linux 运行时 PM 的机制，每个设备（总线的控制器自身也属于一个设备）都有引用计数 usage_count 和活跃子设备（Active Children，子设备的意思就是该级总线上挂的设备）计数 child_count，当两个计数都为 0 的时候，就进入空闲状态，调用 pm_request_idle (dev)。当设备进入空闲状态，与 pm_request_idle (dev) 对应的 PM 核并不一定直接调用设备驱动的 runtime_suspend()，它实际上在多数情况下是调用与该设备对应的 bus_type 的 runtime_idle()。下面是内核的代码逻辑：

```
static pm_callback_t __rpm_get_callback(struct device *dev, size_t cb_offset)
{
    pm_callback_t cb;
    const struct dev_pm_ops *ops;

    if (dev->pm_domain)
        ops = &dev->pm_domain->ops;
    else if (dev->type && dev->type->pm)
        ops = dev->type->pm;
    else if (dev->class && dev->class->pm)
        ops = dev->class->pm;
    else if (dev->bus && dev->bus->pm)
        ops = dev->bus->pm;
    else
        ops = NULL;

    if (ops)
        cb = *(pm_callback_t *)((void *)ops + cb_offset);
    else
        cb = NULL;

    if (!cb && dev->driver && dev->driver->pm)
        cb = *(pm_callback_t *)((void *)dev->driver->pm + cb_offset);

    return cb;
}
```

据此可知，bus_type 级的回调函数实际上可以被 pm_domain、type、class 覆盖掉，这些都统称为子系统。bus_type 等子系统级别的 runtime_idle() 行为完全由相应的总线类型、设备分类和 pm_domain 因素决定，但是一般的行为是子系统级别的 runtime_idle() 会调度设备驱动的 runtime_suspend()。

在具体的设备驱动中，一般的用法则是在设备驱动 probe() 时运行 pm_runtime_enable() 使能运行时 PM 支持，在运行过程中动态地执行 “pm_runtime_get_xxx()->做工作->pm_

“runtime_put_xxx()”的序列。如代码清单 19.19 中的 drivers/watchdog/omap_wdt.c OMAP 的看门狗驱动。在 omap_wdt_start() 中启动了 pm_runtime_get_sync(), 而在 omap_wdt_stop() 中调用了 pm_runtime_put_sync()。

代码清单 19.19 运行时 PM 的 pm_runtime_get() 和 pm_runtime_put()

```

1 static int omap_wdt_start(struct watchdog_device *wdog)
2 {
3     struct omap_wdt_dev *wdev = watchdog_get_drvdata(wdog);
4     void __iomem *base = wdev->base;
5
6     mutex_lock(&wdev->lock);
7
8     wdev->omap_wdt_users = true;
9
10    pm_runtime_get_sync(wdev->dev);
11
12    /* initialize prescaler */
13    while (readl_relaxed(base + OMAP_WATCHDOG_WPS) & 0x01)
14        cpu_relax();
15    ...
16    mutex_unlock(&wdev->lock);
17
18    return 0;
19 }
20 static int omap_wdt_stop(struct watchdog_device *wdog)
21 {
22     struct omap_wdt_dev *wdev = watchdog_get_drvdata(wdog);
23
24     mutex_lock(&wdev->lock);
25     omap_wdt_disable(wdev);
26     pm_runtime_put_sync(wdev->dev);
27     wdev->omap_wdt_users = false;
28     mutex_unlock(&wdev->lock);
29     return 0;
30 }
31
32 static const struct watchdog_ops omap_wdt_ops = {
33     .owner          = THIS_MODULE,
34     .start          = omap_wdt_start,
35     .stop           = omap_wdt_stop,
36     .ping           = omap_wdt_ping,
37     .set_timeout    = omap_wdt_set_timeout,
38 };

```

上述代码第 10 行的 pm_runtime_get_sync(wdev->dev) 告诉内核要开始用看门狗这个设备了，如果看门狗设备已经进入省电模式（之前引用计数为 0 且执行了运行时挂起），会导致该设备的运行时恢复；第 26 行告诉内核不用这个设备了，如果引用计数变为 0 且活跃子设备为 0，则导致该看门狗设备的运行时挂起。

在某些设备驱动中，直接使用上述引用计数的方法进行挂起、空闲和恢复不一定合适，因为挂起状态的进入和恢复需要一些时间，如果设备不在挂起之间保留一定的时间，频繁进出挂起反而会带来新的开销。因此，我们可根据情况决定只有设备在空闲了一段时间后才进入挂起（一般来说，一个一段时间没有被使用的设备，还会有一段时间不会被使用），基于此，一些设备驱动也常常使用自动挂动模式进行编程。

在执行操作的时候声明 pm_runtime_get()，操作完成后执行 pm_runtime_mark_last_busy() 和 pm_runtime_put_autosuspend()，一旦自动挂动的延时到期且设备的使用计数为 0，则引发相关 runtime_suspend() 入口函数的调用。一个典型用法如代码清单 19.20 所示。

代码清单 19.20 运行时 PM 的自动挂动

```

1 foo_read_or_write(struct foo_priv *foo, void *data)
2 {
3     lock(&foo->private_lock);
4     add_request_to_io_queue(foo, data);
5     if (foo->num_pending_requests++ == 0)
6         pm_runtime_get(&foo->dev);
7     if (!foo->is_suspended)
8         foo_process_next_request(foo);
9     unlock(&foo->private_lock);
10 }
11
12 foo_io_completion(struct foo_priv *foo, void *req)
13 {
14     lock(&foo->private_lock);
15     if (--foo->num_pending_requests == 0) {
16         pm_runtime_mark_last_busy(&foo->dev);
17         pm_runtime_put_autosuspend(&foo->dev);
18     } else {
19         foo_process_next_request(foo);
20     }
21     unlock(&foo->private_lock);
22     /* 将请求结果返回给用户 ... */
23 }
```

在上述代码的第 6 行开始进行 I/O 传输了，因此运行了 pm_runtime_get() 之后，当 I/O 传输结束的时候，第 16~17 行向内核告知该设备最后的忙时刻，并执行了 pm_runtime_put_autosuspend()。

设备驱动 PM 成员的 runtime_suspend() 一般完成保存上下文、切到省电模式的工作，而 runtime_resume() 一般完成对硬件上电、恢复上下文的工作。代码清单 19.21 给出了一个 drivers/spi/spi-pl022.c 的案例。

代码清单 19.21 运行时 PM 的 runtime_suspend/resume() 案例

```

1 #ifdef CONFIG_PM
2 static int pl022_runtime_suspend(struct device *dev)
```

```

3  {
4      struct p1022 *p1022 = dev_get_drvdata(dev);
5
6      clk_disable_unprepare(p1022->clk);
7      pinctrl_pm_select_idle_state(dev);
8
9      return 0;
10 }
11
12 static int p1022_runtime_resume(struct device *dev)
13 {
14     struct p1022 *p1022 = dev_get_drvdata(dev);
15
16     pinctrl_pm_select_default_state(dev);
17     clk_prepare_enable(p1022->clk);
18
19     return 0;
20 }
21 #endif
22
23 static const struct dev_pm_ops p1022_dev_pm_ops = {
24     SET_SYSTEM_SLEEP_PM_OPS(p1022_suspend, p1022_resume)
25     SET_RUNTIME_PM_OPS(p1022_runtime_suspend, p1022_runtime_resume, NULL)
26 };

```

第25行的SET_RUNTIME_PM_OPS()是一个快捷宏，它完成了runtime_suspend、runtime_resume的赋值动作，其定义如下：

```
#define SET_RUNTIME_PM_OPS(suspend_fn, resume_fn, idle_fn) \
    .runtime_suspend = suspend_fn, \
    .runtime_resume = resume_fn, \
    .runtime_idle = idle_fn,
```

其实，除了SET_RUNTIME_PM_OPS()和前文介绍的SET_SYSTEM_SLEEP_PM_OPS()，在include/linux/pm.h中还定义了SIMPLE_DEV_PM_OPS()、UNIVERSAL_DEV_PM_OPS()等更快捷的宏：

```
#define SIMPLE_DEV_PM_OPS(name, suspend_fn, resume_fn) \
const struct dev_pm_ops name = { \
    SET_SYSTEM_SLEEP_PM_OPS(suspend_fn, resume_fn) \
}
#define UNIVERSAL_DEV_PM_OPS(name, suspend_fn, resume_fn, idle_fn) \
const struct dev_pm_ops name = { \
    SET_SYSTEM_SLEEP_PM_OPS(suspend_fn, resume_fn) \
    SET_RUNTIME_PM_OPS(suspend_fn, resume_fn, idle_fn) \
}
```

在内核里充斥着这些宏的使用例子。我们从UNIVERSAL_DEV_PM_OPS()这个宏的定义可以看出，它针对的是挂起到RAM和运行时PM行为一致的场景。

19.11 总结

Linux 内核的 PM 框架涉及众多组件，弄清楚这些组件之间的依赖关系，在合适的着眼点上进行优化，采用正确的方法进行 PM 的编程，对改善代码的质量、辅助功耗和性能测试都有极大的好处。

另外，在实际工程中，尤其是在消费电子的领域，可能有超过半数的 bug 都属于电源管理。这个时候，电源管理的很多工作就是在搞定鲁棒性和健壮性，可以说，在很多时候，这就是个体力活，需要工程师有足够的耐性。