Linux 电源管理系统架构和驱动

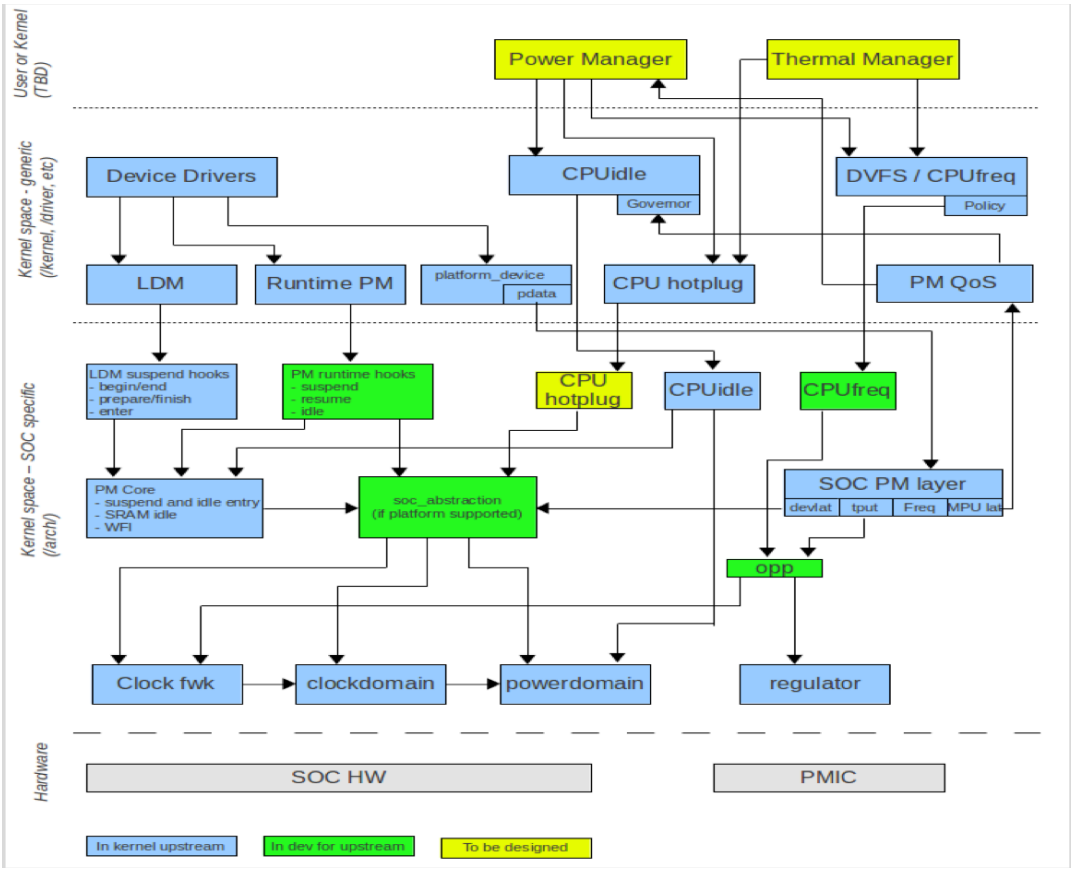
宋宝华 <21cnbao@gmail.com>
新浪微博: @宋宝华 Barry

1. Linux 电源管理全局架构

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于系统待机的支持和每个设备的运行时电源管理，可以说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，下图呈现了 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. 系统和设备对于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 Suspend to RAM/Disk 的一系列入口函数
6. SoC 进入 suspend 状态、SDRAM 自刷新的入口
7. 设备的 runtime（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



2. CPUFreq 驱动

CPUFreq 子系统位于 drivers/cpufreq 目录，负责进行运行过程中 CPU 频率和电压的动态调整，即 DVFS（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，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，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 的机制，可以在 CPUFreq 的 policy 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。

1.1 SoC 的 CPUFreq 驱动实现

具体的每个 SoC 的 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 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：

```
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 inttarget_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. structfreq_attr             **attr;
28. };
```

其中的 owner 成员一般被设置为 THIS_MODULE；name 成员是 CPUFreq 驱动名字，如 drivers/cpufreq/s5pv210-cpufreq.c 设置 name 为"s5pv210"，如 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

该 CPU 支持的最小频率和最大频率（单位是 kHz）。

policy->cpuinfo.transition_latency

CPU 进行频率切换所需要的延迟（单位是纳秒）

policy->cur

CPU 当前频率

policy->policy

policy->governor

policy->min

policy->max

定义该 CPU 的缺省 policy，以及缺省 policy 情况下该 policy 支持的最小、最大 CPU 频率。

verify()成员函数用于对用户的 CPUFreq policy 设置进行有效性验证和数据修正。每次用户设定一个新 policy 时，该函数会被传递一个 policy,governor,min,max 的集合，verify()可检验该集合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 intmin_freq, unsigned intmax_freq);
```

verify()必须确保至少有 1 个有效的频率或者频率范围出现在 policy->min 到 policy->max 之间，必要的情况下，为了满足这个要求，会修正 policy->min 或 policy->max。

绝大多数 ARM SoC 不实现 setpolicy()成员函数，因为芯片不具备设定一个频率 limit（即 policy->min 和 policy->max）的能力，一般的 ARM SoC 只是支持设定到某一特定频率。setpolicy()成员函数仅接受一个 policy 参数，并根据 policy 去设置 CPU 内部的动态频率调整单元的 limit 为 policy->max 和 policy->min。若 policy 为 CPUFREQ_POLICY_PERFORMANCE，则设置 CPU 内部的动态频率管理单元为高性能模式，相反地，若 policy 为 CPUFREQ_POLICY_POWERSAVE，则设置 CPU 内部的的动态频率管理单元为省电模式。

绝大多数 ARM SoC 会实现 target()成员函数，该函数用于实际的频率调整，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，暗示设置的频率应该大于或等于 target_freq；relation 若为 CPUFREQ_REL_H，暗示设置的频率应该小于或等于 target_freq。

实际上，由于芯片内部 PLL 和分频器的关系，ARM SoC 不太容易实现频率的随意变化，往往 SoC 的 CPUFreq 驱动会提供一个频率表，在该表的范围内进行变更，因此 CPUFreq 核心层提供了一组频率表相关的辅助 API。

```
int cpufreq_frequency_table_cpuinfo(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 cpuinfo.min_freq 和 cpuinfo.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 1 个有效的 CPU 频率位于 policy->min 到 policy->max 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 驱动的实例 drivers/cpufreq/s3c64xx-cpufreq.c 的核心结构如下：

```
1.  static unsigned long regulator_latency;
2.
3.  #ifdef CONFIG_CPU_S3C6410
4.  struct s3c64xx_dvfs {
5.      unsigned intvddarm_min;
6.      unsigned intvddarm_max;
7.  };
8.
9.  static struct s3c64xx_dvfs s3c64xx_dvfs_table[] = {
10.     [0] = { 1000000, 1150000 },
11.     …
12.     [4] = { 1300000, 1350000 },
13. };
14.
15. static struct cpufreq_frequency_table s3c64xx_freq_table[] = {
16.     { 0,    66000 },
17.     { 0, 100000 },
18.     { 0, 133000 },
19.     …
20.     { 0, CPUFREQ_TABLE_END },
21. };
22. #endif
23.
24. static int s3c64xx_cpufreq_verify_speed(struct cpufreq_policy *policy)
```

```
{
        if (policy->cpu != 0)
                return -EINVAL;

        return cpufreq_frequency_table_verify(policy, s3c64xx_freq_table);
}

static unsigned int s3c64xx_cpufreq_get_speed(unsigned intcpu)
{
        if (cpu != 0)
                return 0;

        return clk_get_rate(armclk) / 1000;
}

static int s3c64xx_cpufreq_set_target(struct cpufreq_policy *policy,
                                unsigned inttarget_freq,
                                unsigned int relation)
{
        …

        ret = cpufreq_frequency_table_target(policy, s3c64xx_freq_table,
                                        target_freq, relation, &i);
        …

        freqs.cpu = 0;
        freqs.old = clk_get_rate(armclk) / 1000;
        freqs.new = s3c64xx_freq_table[i].frequency;
        freqs.flags = 0;
        dvfs = &s3c64xx_dvfs_table[s3c64xx_freq_table[i].index];

        if (freqs.old == freqs.new)
                return 0;

        cpufreq_notify_transition(&freqs, CPUFREQ_PRECHANGE);

#ifdef CONFIG_REGULATOR
        if (vddarm&&freqs.new>freqs.old) {
                ret = regulator_set_voltage(vddarm,
                                        dvfs->vddarm_min,
                                        dvfs->vddarm_max);
                …
        }
#endif
```

```
69.
70.         ret = clk_set_rate(armclk, freqs.new * 1000);
71.         …
72.
73.         cpufreq_notify_transition(&freqs, CPUFREQ_POSTCHANGE);
74.
75.  #ifdef CONFIG_REGULATOR
76.         if (vddarm&&freqs.new<freqs.old) {
77.                 ret = regulator_set_voltage(vddarm,
78.                                         dvfs->vddarm_min,
79.                                         dvfs->vddarm_max);
80.                 …
81.         }
82.  #endif
83.
84.         return 0;
85.         …
86.  }
87.
88.  static int s3c64xx_cpufreq_driver_init(struct cpufreq_policy *policy)
89.  {
90.         …
91.         armclk = clk_get(NULL, "armclk");
92.         …
93.
94.  #ifdef CONFIG_REGULATOR
95.         vddarm = regulator_get(NULL, "vddarm");
96.         …
97.         s3c64xx_cpufreq_config_regulator();
98.  #endif
99.
100.        freq = s3c64xx_freq_table;
101.        while (freq->frequency != CPUFREQ_TABLE_END) {
102.                unsigned long r;
103.
104.                /* Check for frequencies we can generate */
105.                r = clk_round_rate(armclk, freq->frequency * 1000);
106.                r /= 1000;
107.                if (r != freq->frequency) {
108.                        pr_debug("%dkHz unsupported by clock\n",
109.                                freq->frequency);
110.                        freq->frequency = CPUFREQ_ENTRY_INVALID;
111.                }
112.
```

```
113.             /* If we have no regulator then assume startup
114.               * frequency is the maximum we can support. */
115.             if (!vddarm&& freq->frequency > s3c64xx_cpufreq_get_speed(0))
116.                     freq->frequency = CPUFREQ_ENTRY_INVALID;
117.
118.             freq++;
119.         }
120.
121.         policy->cur = clk_get_rate(armclk) / 1000;
122.
123.         /* Datasheet says PLL stabalisation time (if we were to use
124.           * the PLLs, which we don't currently) is ~300us worst case,
125.           * but add some fudge.
126.           */
127.         policy->cpuinfo.transition_latency = (500 * 1000) + regulator_latency;
128.
129.         ret = cpufreq_frequency_table_cpuinfo(policy, s3c64xx_freq_table);
130.         …
131.
132.         return ret;
133. }
134.
135. staticstruct cpufreq_driver s3c64xx_cpufreq_driver = {
136.         .owner          = THIS_MODULE,
137.         .flags              = 0,
138.         .verify         = s3c64xx_cpufreq_verify_speed,
139.         .target         = s3c64xx_cpufreq_set_target,
140.         .get        = s3c64xx_cpufreq_get_speed,
141.         .init       = s3c64xx_cpufreq_driver_init,
142.         .name           = "s3c",
143. };
144.
145. static int __init s3c64xx_cpufreq_init(void)
146. {
147.         return cpufreq_register_driver(&s3c64xx_cpufreq_driver);
148. }
149. module_init(s3c64xx_cpufreq_init);
```

1.2 CPUFreq 的 policy

SoC CPUFreq 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，但是它并不会管自身 CPU 究竟应该运行在什么频率上。究竟频率依据什么样的标准，进行何种变化，则完全由 CPUFreq 的 policy（策略）决定，这些 policy 如下表：

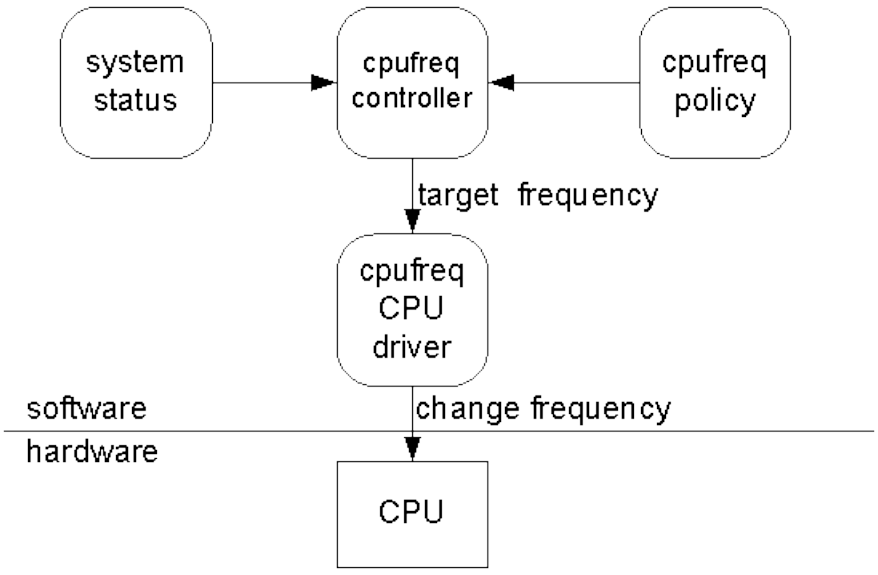
CPUFreq 的 policy	策略实现方法
cpufreq_ondemand	平时以低速方式运行，当系统负载提高时候按需自动提高频率
cpufreq_performance	CPU 以最高频率运行，即 scaling_max_freq
cpufreq_conservative	字面含义是传统的、保守的，跟 ondemand 相似，区别在于动态频率变更的时候采用渐进的方式
cpufreq_powersave	CPU 以最低频率运行，即 scaling_min_freq
cpufreq_userspace	让 root 用户透过 sys 结点 scaling_setspeed 设置频率

以 ondemand policy 为例，它根据 CPU 的利用率进行 CPU 频率的动态升降，其算法的伪代码如下：

```
For each CPU:
Every X milliseconds:
        Get utilization since last check
        if( utilization > UP_THRESHOLD)
                increase frequency to MAX
Every Y milliseconds:
        Get utilization since last
        check if( utilization < DOWN_THRESHOLD)
                decrease frequency 20%
```

在 Android 系统中，则增加了 1 个 interactive policy，该 policy 适合对延迟敏感的 UI 交互任务，当有 UI 交互任务的时候，该 policy 会更加激进和及时地调整 CPU 频率。

总地来说，系统的状态以及 CPUFreq 的 Policy 共同决定了 CPU 频率跳变的 target，CPUFreq 核心层并将 target 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：



1.3 CPUFreq 的性能测试和调优

Linux 3.1 已经将 cpupower-utils 中放入内核的 tools/power/cpupower 目录，该工具集当中的 cpufreq-bench 工具可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，然后计算在使用 ondemand、conservative、interactive 等 policy 的情况下，做同样的运算与在 performance 高频模式下做同样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，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 policy 相对于 performance policy 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%

显然不太理想，我们在同样的平台下采用 Android 的 interactive policy，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%

一般的目标是采用 CPUFreq 动态调整频率和电压后，性能应该为 performance 这个高性能 policy 情况下的 90%左右比较理想。

1.4 CPUFreq 通知

两种情况下，CPUFreq 子系统会发出通知：CPUFreq 的 policy 变化或者 CPU 运行频率变化。

在 policy 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考虑去修改 limit（即 policy->min 和 policy->max）；

CPUFREQ_INCOMPATIBLE：除非前面的 policy 设定可能会导致硬件的出错，被注册的 notifier 才可以改变 limit 等设定；

CPUFREQ_NOTIFY：所有注册的 notifier 都会被告知新的 policy 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 机制监控之。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，则它注册了 notifier 并在 CPUFREQ_PRECHANGE 和 CPUFREQ_POSTCHANGE 情况下分别进行不同的处理：

```
1.  fbi->freq_transition.notifier_call = sa1100fb_freq_transition;
2.  cpufreq_register_notifier(&fbi->freq_transition, CPUFREQ_TRANSITION_NOTIFIER);
3.
4.  static int
5.  sa1100fb_freq_transition(structnotifier_block *nb, unsigned long val,
6.  void *data)
7.  {
8.      struct sa1100fb_info *fbi = TO_INF(nb, freq_transition);
9.      struct cpufreq_freqs *f = data;
10.     u_intpcd;
11.
12.     switch (val) {
13.         case CPUFREQ_PRECHANGE:
14.             set_ctrlr_state(fbi, C_DISABLE_CLKCHANGE);
15.             break;
16.         case CPUFREQ_POSTCHANGE:
17.             pcd = get_pcd(fbi->fb.var.pixclock, f->new);
18.             fbi->reg_lccr3 = (fbi->reg_lccr3& ~0xff) | LCCR3_PixClkDiv(pcd);
19.             set_ctrlr_state(fbi, C_ENABLE_CLKCHANGE);
20.             break;
21.     }
22.     return 0;
23. }
```

此外，如果在系统 suspend/resume 的过程中 CPU 频率会发生变化，则 CPUFreq 只系统也会发出 CPUFREQ_SUSPENDCHANGE 和 CPUFREQ_RESUMECHANGE 这 2 个通知。

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 系列笔记本电脑而言，支持 ACPI（Advanced Configuration and Power Interface，高级配置和电源管理接口），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）：

State	Power (mW)	Latency (uS)
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：

```
ENTRY(cpu_v7_do_idle)
dsb                                             @ WFI may enter a low-power mode
wfi
mov         pc, lr
ENDPROC(cpu_v7_do_idle)
```

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

```
intcpuidle_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，其定义如下：

```
struct cpuidle_driver {
            const char                  *name;
            struct module               *owner;

            unsigned int                power_specified:1;
           /* set to 1 to use the core cpuidle time keeping (for all states). */
            unsigned int                en_core_tk_irqen:1;
            struct cpuidle_state        states[CPUIDLE_STATE_MAX];
            int state_count;
            int safe_state_index;
};
```

该结构体的关键成员是 1 个 cpuidle_state 的表，其实就是存储各种不同 IDLE 级别的信息，它的定义如下：

```
1.  struct cpuidle_state {
2.      char                    name[CPUIDLE_NAME_LEN];
3.      char desc[CPUIDLE_DESC_LEN];
4.
```

```
5.      unsigned int        flags;
6.      unsigned intexit_latency; /* in US */
7.      int power_usage; /* in mW */
8.      unsigned inttarget_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。

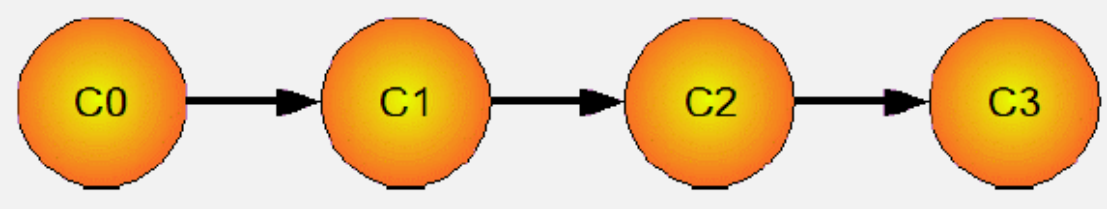
省略细节，一个具体的 SoC 的 CPUIdle 驱动实例可见于 arch/arm/mach-ux500/cpuidle.c，它有 2 个 IDLE 级别，即“WFI”和“ApIdle”：

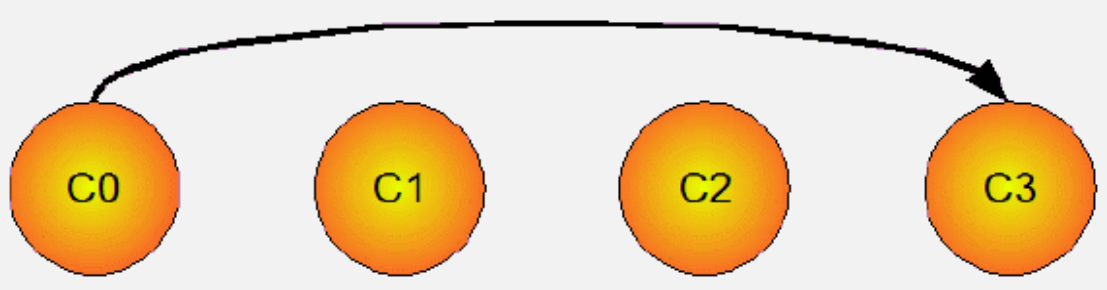
```
1.  staticatomic_t master = ATOMIC_INIT(0);
2.  static DEFINE_SPINLOCK(master_lock);
3.  static DEFINE_PER_CPU(struct cpuidle_device, ux500_cpuidle_device);
4.
5.  static inline int ux500_enter_idle(struct cpuidle_device *dev,
6.                          struct cpuidle_driver *drv, int index)
7.  {
8.      …
9.  }
10.
11. staticstruct 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 CPUIdle 子系统也有对应的 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的 IDLE 级别。前者适合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，后者依赖于内核的 NO_HZ 选项。

下图演示了 LADDER 步进从 C0 进入 C3，而 MENU 则可能直接从 C0 跳入 C3：



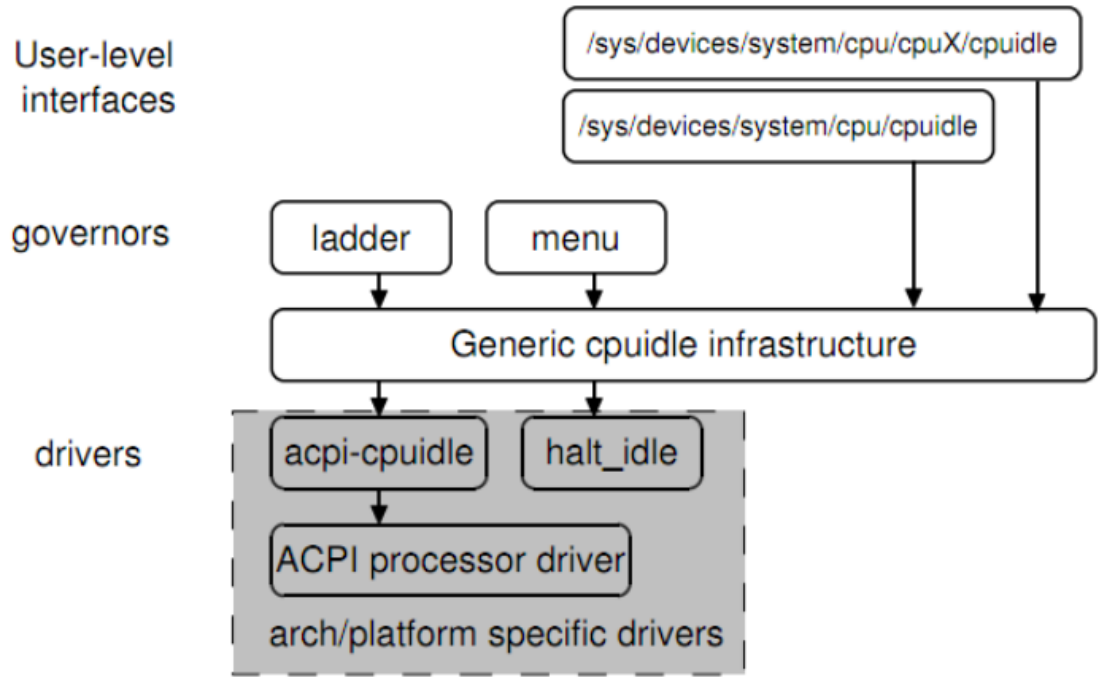


CPUIdle 子系统还透过 sys 向 userspace 导出了一些结点：

一类是针对整个系统的/sys/devices/system/cpu/cpuidle，透过其中的 current_driver、current_governor、available_governors 等结点可以获取或设置 CPUIdle 的驱动信息以及 governor。

一类是针对每个 CPU 的/sys/devices/system/cpu/cpux/cpuidle/，通过子结点暴露各个 online 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 子系统的总体架构图如下：



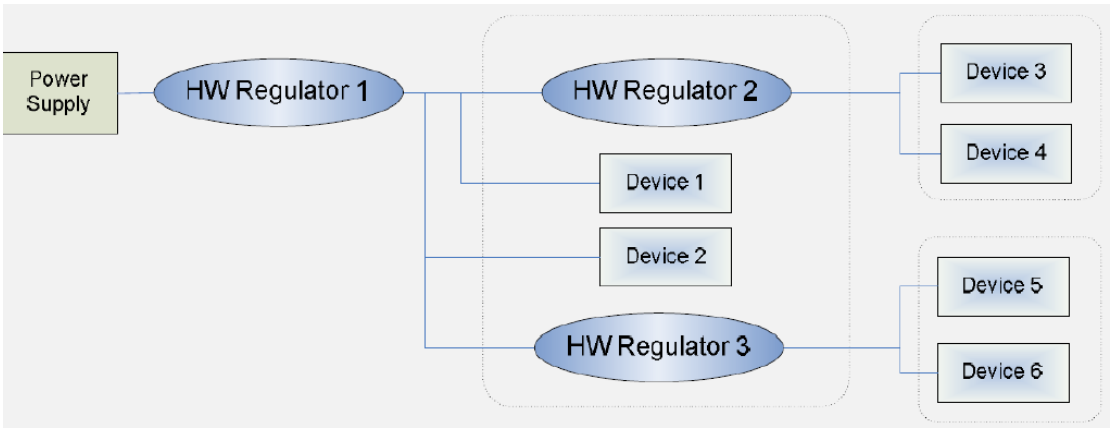
4. PowerTop

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，目前发布的版本是 2.2，采用了基于 TAB 的界面风格。

```
PowerTOP 2.0     Overview   Idle stats   Frequency stats   Device stats   Tunables

The battery reports a discharge rate of 14.3 W
The estimated remaining time is 93 minutes

Summary: 165.5 wakeups/second,  0.0 GPU ops/second, 0.0 VFS ops/sec and 4.1% CPU use

Power est.      Usage       Events/s    Category       Description
  2.74 W    100.0%                    Device         Display backlight
  831 mW    100.0%                    Device         USB device: USB Optical Mouse
  527 mW      1.0 ms/s      59.8        Interrupt      PS/2 Touchpad / Keyboard / Mouse
  351 mW    100.0%                    Device         Audio codec hwC0D3: Intel
  351 mW    100.0%                    Device         Audio codec hwC0D0: Realtek
  282 mW      6.2 ms/s      26.3        Process        /usr/bin/Xorg :0 -background none
  256 mW     24.9 ms/s       4.7        Process        xfce4-screensho
  170 mW    100.0%                    Device         USB device: AX88772
  160 mW    519.3 µs/s      18.0        Interrupt      [7] sched(softirq)
 80.1 mW    215.5 µs/s       9.0        Interrupt      [41] i915
 71.8 mW      2.0 ms/s       6.3        Process        /usr/bin/Terminal
 59.5 mW    379.2 µs/s       6.5        Interrupt      [23] ehci_hcd:usb2
 44.9 mW    146.4 µs/s       5.0        Process        iscsid
 40.8 mW    414.7 µs/s       4.3        Process        xfwm4 --display :0.0 --sm-client-
 30.8 mW     13.2 µs/s       3.5        Interrupt      [6] tasklet(softirq)
 26.8 mW      0.7 ms/s       2.4        Process        xfdesktop --display :0.0 --sm-cli
 20.8 mW      8.2 µs/s       2.4        kWork          console_callback
 15.6 mW    200.4 µs/s       1.6        Interrupt      [1] timer(softirq)
```

5. Regulator 驱动

Regulator 是 Linux 系统中电源管理的基础设施之一，前面介绍的 CPUFreq 驱动经常使用它来设定电压。Regulator 驱动主要用来管理系统中的供电单元即稳压器（如 LDO，即 low dropout regulator，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供电树形结构，如下图：



Linux 的 Regulator 子系统提供如下 API 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 2 个参数分别是 regulator_desc 结构体和 regulator_config 结构体的指针，前者是对这个稳压器属性和操作的封装:

```
struct regulator_desc {
        const char *name; /* regulator 的名字 */
        const char *supply_name; /* regulator supply 的名字 */
        int id;
```

```
        unsigned n_voltages;
        struct regulator_ops *ops;
        int irq;
        enum regulator_type type; /*  是电压还是电流 Regulator */
        struct module *owner;

        unsigned int min_uV; /*  线性映射情况下最低的 selector 的电压  */
        unsigned int uV_step; /*  线性映射情况下每步增加/减小的电压  */
        unsigned int ramp_delay; /*  电压改变后稳定下来所需时间  */

        const unsigned int *volt_table; /*  基于表映射情况下的电压映射表  */

        unsigned int vsel_reg;
        unsigned int vsel_mask;
        unsigned int enable_reg;
        unsigned int enable_mask;
        unsigned int bypass_reg;
        unsigned int bypass_mask;

        unsigned int enable_time;
};
```

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：

```
struct regulator_ops {

        /* enumerate supported voltages */
        int (*list_voltage) (struct regulator_dev *, unsigned selector);

        /* get/set regulator voltage */
        int (*set_voltage) (struct regulator_dev *, int min_uV, int max_uV,
                                    unsigned *selector);
        int (*map_voltage)(struct regulator_dev *, int min_uV, int max_uV);
        int (*set_voltage_sel) (struct regulator_dev *, unsigned selector);
        int (*get_voltage) (struct regulator_dev *);
        int (*get_voltage_sel) (struct regulator_dev *);

        /* get/set regulator current    */
        int (*set_current_limit) (struct regulator_dev *,
                                        int min_uA, int max_uA);
        int (*get_current_limit) (struct regulator_dev *);

        /* enable/disable regulator */
        int (*enable) (struct regulator_dev *);
        int (*disable) (struct regulator_dev *);
```

```
        int (*is_enabled) (struct regulator_dev *);

        …
};
```

在 drivers/regulator 目录下，包含大量的电源芯片对应的 Regulator 驱动，如 Dialog 的 DA9052、Intersil 的 ISL6271A、ST-Ericsson 的 TPS61050/61052、Wolfon 的 WM831x 系列等，它同时提供了一个 dummy 的 Regulator 驱动作为参考：

```
struct regulator_dev *dummy_regulator_rdev;
static struct regulator_init_data dummy_initdata;
static struct regulator_ops dummy_ops;
static struct regulator_desc dummy_desc = {
        .name = "regulator-dummy",
        .id = -1,
        .type = REGULATOR_VOLTAGE,
        .owner = THIS_MODULE,
        .ops = &dummy_ops,
};

static int __devinit dummy_regulator_probe(struct platform_device *pdev)
{
        struct regulator_config config = { };
        int ret;

        config.dev = &pdev->dev;
        config.init_data = &dummy_initdata;

        dummy_regulator_rdev = regulator_register(&dummy_desc, &config);
        if (IS_ERR(dummy_regulator_rdev)) {
                ret = PTR_ERR(dummy_regulator_rdev);
                pr_err("Failed to register regulator: %d\n", ret);
                return ret;
        }

        return 0;
}
```

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：

```
struct regulator * regulator_get(struct device *dev, const char *id);
struct regulator * devm_regulator_get(struct device *dev, const char *id);
struct regulator *regulator_get_exclusive(struct device *dev, const char *id);
void regulator_put(struct regulator *regulator);
void devm_regulator_put(struct regulator *regulator);
int regulator_enable(struct regulator *regulator);
int regulator_disable(struct regulator *regulator);
```

```
int regulator_set_voltage(struct regulator *regulator, int min_uV, int max_uV);
int regulator_get_voltage(struct regulator *regulator);
```

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

6. OPP

当今的 SoC 一般包含很多集成的组件，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 可以运行在更低的频率和电压，而其他 domain 可以运行在更高的频率和电压，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：

```
static struct omap_opp_def __initdata omap44xx_opp_def_list[] = {
        /* MPU OPP1 - OPP50 */
        OPP_INITIALIZER("mpu", true, 300000000,
OMAP4430_VDD_MPU_OPP50_UV),
        /* MPU OPP2 - OPP100 */
        OPP_INITIALIZER("mpu", true, 600000000,
OMAP4430_VDD_MPU_OPP100_UV),
        /* MPU OPP3 - OPP-Turbo */
        OPP_INITIALIZER("mpu", true, 800000000,
OMAP4430_VDD_MPU_OPPTURBO_UV),
        /* MPU OPP4 - OPP-SB */
        OPP_INITIALIZER("mpu", true, 1008000000,
OMAP4430_VDD_MPU_OPPNITRO_UV),
        …
};
/**
 * omap4_opp_init() - initialize omap4 opp table
 */
int __init omap4_opp_init(void)
{
        …
        r = omap_init_opp_table(omap44xx_opp_def_list,
                        ARRAY_SIZE(omap44xx_opp_def_list));

        return r;
}
device_initcall(omap4_opp_init);
int __init omap_init_opp_table(struct omap_opp_def *opp_def,
                u32 opp_def_size)
{
```

```
        …
        /* Lets now register with OPP library */
        for (i = 0; i < opp_def_size; i++, opp_def++) {
                …
                if (!strncmp(opp_def->hwmod_name, "mpu", 3)) {
                        /*
                         * All current OMAPs share voltage rail and
                         * clock source, so CPU0 is used to represent
                         * the MPU-SS.
                         */
                        dev = get_cpu_device(0);
                } …
                r = opp_add(dev, opp_def->freq, opp_def->u_volt);
                …
        }
        return 0;
}
```

针对 device 结构体指针 dev 对应的 domain 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 disable，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，可采用类似代码：

```
        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 2 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 2 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，其支撑该频率运行所需的电压也往往可以动态调低；反之，则可以调高，下面这 2 个 API 分别用于获取某 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 驱动像将 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()来依据前面注册的 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 target 成员函数 omap_target()中，则使用 OPP 相关的 API 来获取了频率和电压：

```
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>表方面底层使用了 OPP，在设置电压的时候又使用了规范的 Regulator API。

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，透过这个接口，用户可以设定自身对性能的期望。一类是系统级的需求，透过 cpu_dma_latency, network_latency, network_throughput 这些参数来设定；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：

```
static int ladder_select_state(struct cpuidle_driver *drv,
                                    struct cpuidle_device *dev)
{
```

```
        …
        int latency_req = pm_qos_request(PM_QOS_CPU_DMA_LATENCY);

        …

        /* consider promotion */
        if (last_idx < drv->state_count - 1 &&
            !drv->states[last_idx + 1].disabled &&
            !dev->states_usage[last_idx + 1].disable &&
            last_residency > last_state->threshold.promotion_time &&
            drv->states[last_idx + 1].exit_latency <= latency_req) {
                last_state->stats.promotion_count++;
                last_state->stats.demotion_count = 0;
                if (last_state->stats.promotion_count >=
last_state->threshold.promotion_count) {
                        ladder_do_selection(ldev, last_idx, last_idx + 1);
                        return last_idx + 1;
                }
        }
…
}
```

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 要小于透过 pm_qos_request(PM_QOS_CPU_DMA_LATENCY)得到的 PM QoS 请求的延迟。

同样的逻辑也出现于 drivers/cpuidle/governors/menu.c 中：

```
static int menu_select(struct cpuidle_driver *drv, struct cpuidle_device *dev)
{
        struct menu_device *data = &__get_cpu_var(menu_devices);
        int latency_req = pm_qos_request(PM_QOS_CPU_DMA_LATENCY);
        …
        /*
         * Find the idle state with the lowest power while satisfying
         * our constraints.
         */
        for (i = CPUIDLE_DRIVER_STATE_START; i < drv->state_count; i++) {
                struct cpuidle_state *s = &drv->states[i];
                struct cpuidle_state_usage *su = &dev->states_usage[i];

                if (s->disabled || su->disable)
                        continue;
                if (s->target_residency > data->predicted_us)
                        continue;
                if (s->exit_latency > latency_req)
                        continue;
                if (s->exit_latency * multiplier > data->predicted_us)
```

```
                                continue;

                        if (s->power_usage < power_usage) {
                                power_usage = s->power_usage;
                                data->last_state_idx = i;
                                data->exit_us = s->exit_latency;
                        }
                }

                return data->last_state_idx;
}
```

当摄像头关闭后，通过如下语句告知上述对 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。

8. CPU 热插拔

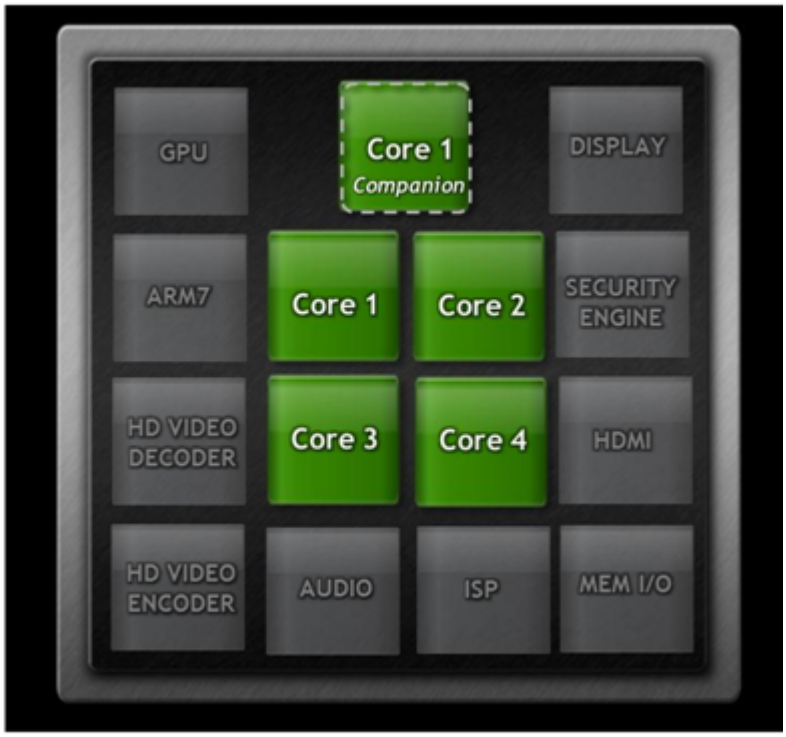
关于 CPU 热插拔的具体实现方法，可参考《Linux 芯片级移植与底层驱动》一文 SMP 支持相关的小节，作者的微博下载地址：http://vdisk.weibo.com/s/oGuO8。一般来讲，在用户空间可以透过/sys/devices/system/cpu/cpun/online 结点来操作一个 CPU 的 online 和 offline：

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

透过“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，保证这个拔除 CPU3 的过程中，系统仍然能正常运行。一旦透过“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 offline 的 CPU。目前 Linux 内核并没有统一的管理何时 CPU 进行热插拔的机制，仍然由各个芯片公司根据自身的情况自行实现。这里以 Tegra3 为例展开。

Tegra3 采用 vSMP（variable Symmetric Multiprocessing）架构，共 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：



在系统运行过程中，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 kernel 支持的 CPU hotplug 进行 CPU 的 UP/DOWN 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

运行过程中，我们会发现 4 个 G core 会动态热插拔，而 4 个 G core 和 1 个 LP core 之间，会根据运行负载进行 cluster 切换。这一部分都是在内核里面实现，和 tegra 的 cpufreq 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=e5d1ff22f5c5629f3c8078e1db308a4b45fa382e;hb=rel-14r7

如何判断自己是什么 core

每个 core 都可以通过调用 is_lp_cluster()来判断当前执行 CPU 是 LP 还是 G 处理器：

```
1.  static inline unsigned int is_lp_cluster(void)
2.  {
3.      unsigned int reg;
4.      reg =readl(FLOW_CTRL_CLUSTER_CONTROL);
5.      return (reg& 1); /* 0 == G, 1 == LP*/
6.  }
```

即读 FLOW_CTRL_CLUSTER_CONTROL 寄存器判断出来自己是 G core 还是 LP core。

G core 和 LP core cluster 的切换时机

[场景 1]何时从 LP 切换给 G：当前执行于 LPcluster，CPUFreq 驱动判断出 LP 需要升频率超过高值门限，即 TEGRA_HP_UP：

```
1.  caseTEGRA_HP_UP:
2.      if(is_lp_cluster() && !no_lp) {
3.          if(!clk_set_parent(cpu_clk, cpu_g_clk)) {
4.              hp_stats_update(CONFIG_NR_CPUS, false);
5.              hp_stats_update(0, true);
6.              /* catch-upwith governor target speed */
7.              tegra_cpu_set_speed_cap(NULL);
8.          }
```

[场景 2]何时从 G 切换给 LP：当前执行于 Gcluster，CPUFreq 驱动判断出某 G core 需要降频率到小于低值门限，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）：

```
1.  caseTEGRA_HP_DOWN:
2.      cpu= tegra_get_slowest_cpu_n();
3.      if(cpu < nr_cpu_ids) {
4.          ...
5.      }else if (!is_lp_cluster() && !no_lp) {
6.          if(!clk_set_parent(cpu_clk, cpu_lp_clk)) {
7.              hp_stats_update(CONFIG_NR_CPUS, true);
8.              hp_stats_update(0, false);
9.              /* catch-upwith governor target speed */
10.             tegra_cpu_set_speed_cap(NULL);
```

```
11.             } else
12.                     queue_delayed_work(
13.                             hotplug_wq, &hotplug_work, down_delay);
14.         }
15.         break;
```

切换实际上就发生在 clk_set_parent()更改 CPU 的父时钟里面，这部分代码写地比较丑，1 个函数完成 n 个功能，实际不仅切换了时钟，还切换了 G 和 LP cluster：

clk_set_parent(cpu_clk, cpu_lp_clk) ->

tegra3_cpu_cmplx_clk_set_parent(structclk *c, struct clk *p) ->

tegra_cluster_control(unsigned int us, unsigned int flags) ->

tegra_cluster_switch_prolog()->

tegra_cluster_switch_epilog()

G core 动态热插拔

何时进行 G core 的动态 plug 和 unplug：

[场景 3]当前执行于 G cluster，CPUFreq 驱动判断出某 Gcore 需要降频率到小于低值门限，即 TEGRA_HP_DOWN，且最慢的 CPUID 小于 nr_cpu_ids（实际上等价于还有 2 个或 2 个以上的 G core 活着）,关闭最慢的 CPU，留意 tegra_get_slowest_cpu_n()不会返回 0，意味着 CPU0 要么活着，要么切换给 LP（对应场景 2）：

```
1.  caseTEGRA_HP_DOWN:
2.      cpu= tegra_get_slowest_cpu_n();
3.      if(cpu < nr_cpu_ids) {
4.          up = false;
5.          queue_delayed_work(
6.                  hotplug_wq,&hotplug_work, down_delay);
7.          hp_stats_update(cpu, false);
8.      }
```

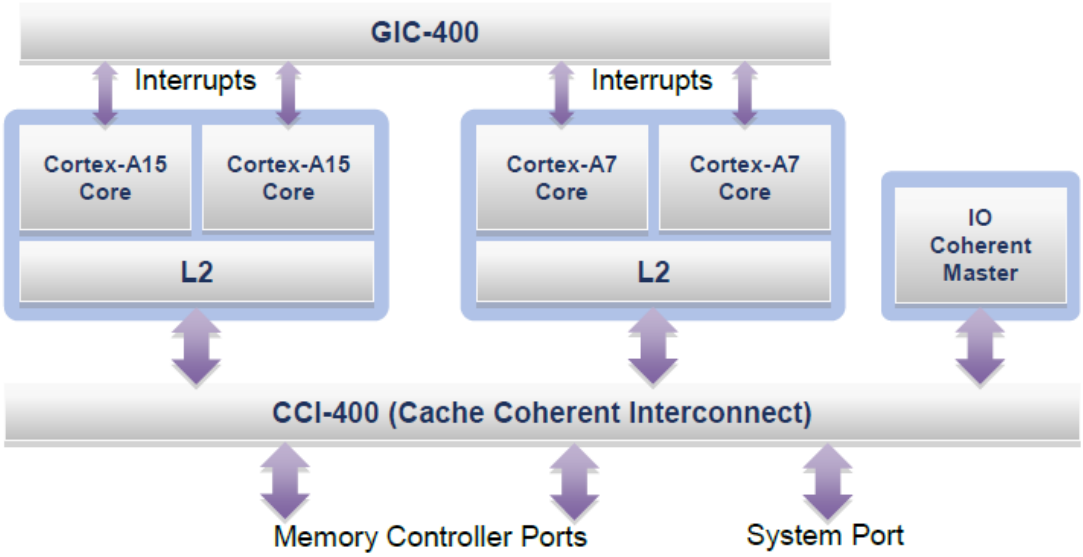
[场景 4]当前执行于 G cluster，CPUFreq 驱动判断出某 Gcore 需要设置频率大于高值门限，即 TEGRA_HP_UP，如果负载平衡状态为 TEGRA_CPU_SPEED_BALANCED，再开一个 core；如果状态为 TEGRA_CPU_SPEED_SKEWED，则关一个 core。TEGRA_CPU_SPEED_BALANCED 的含义是当前所有 Gcore 要求的频率都高于最高频率的 50%，TEGRA_CPU_SPEED_SKEWED 的含义是当前至少有 2 个 Gcore 要求的频率低于门限的 25%，即 CPU 频率的要求在各个 core 间有倾斜。

```
1.  caseTEGRA_HP_UP:
2.      if(is_lp_cluster() && !no_lp) {
3.          ...
```

```
4.      }else {
5.          switch (tegra_cpu_speed_balance()) {
6.          /* cpu speed is up and balanced - one more on-line */
7.          case TEGRA_CPU_SPEED_BALANCED:
8.                  cpu =cpumask_next_zero(0, cpu_online_mask);
9.                  if (cpu <nr_cpu_ids) {
10.                     up =true;
11.                     hp_stats_update(cpu, true);
12.                 }
13.                 break;
14.         /* cpu speed is up, but skewed - remove one core */
15.         case TEGRA_CPU_SPEED_SKEWED:
16.                 cpu =tegra_get_slowest_cpu_n();
17.                 if (cpu < nr_cpu_ids) {
18.                     up =false;
19.                     hp_stats_update(cpu, false);
20.                 }
21.                 break;
22.         /* cpu speed is up, butunder-utilized - do nothing */
23.         case TEGRA_CPU_SPEED_BIASED:
24.         default:
25.                 break;
26.         }
27.     }
```

上述代码中 TEGRA_CPU_SPEED_BIASED 路径的含义是有 1 个以上 Gcore 的频率低于最高频率的 50%但是未形成 SKEWED 条件，即只是“BIASED”，还没有达到“SKEWED”的程度，所以暂时什么都不做。

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。在 big.LITTLE 架构中，将高性能功耗也较高的 Cortex-A15 和稍低性能功耗低的 Cortex-A7 进行了结合：

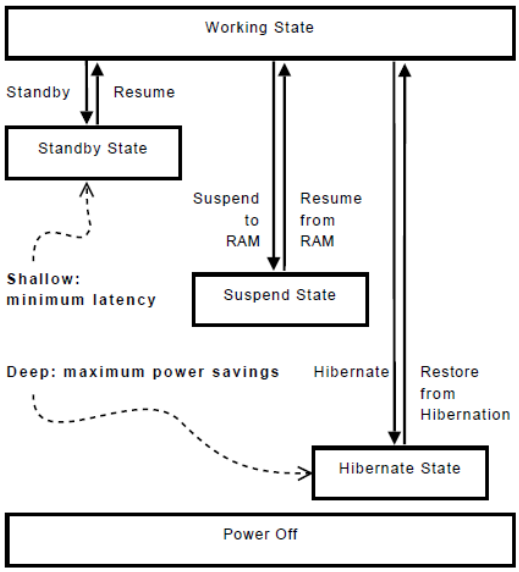


big.LITTLE 处理的设计旨在为适当的作业分配恰当的处理器。Cortex-A15 处理器是目前已开发的性能最高的低功耗 ARM 处理器，而 Cortex-A7 处理器是目前已开发的最节能的 ARM 应用程序处理器。可以利用 Cortex-A15 处理器的性能来承担繁重的工作负载，而 Cortex-A7 可以最有效地处理智能手机的大部分工作负载。这些操作包括操作系统活动、用户界面和其他持续运行、始终连接的任务。

三星在 CES（国际消费电子展） 2013 大会上发布了 Exynos 5 Octa 八核移动处理器，这款处理器也是采用 big.LITTLE 结构的第一款 CPU。

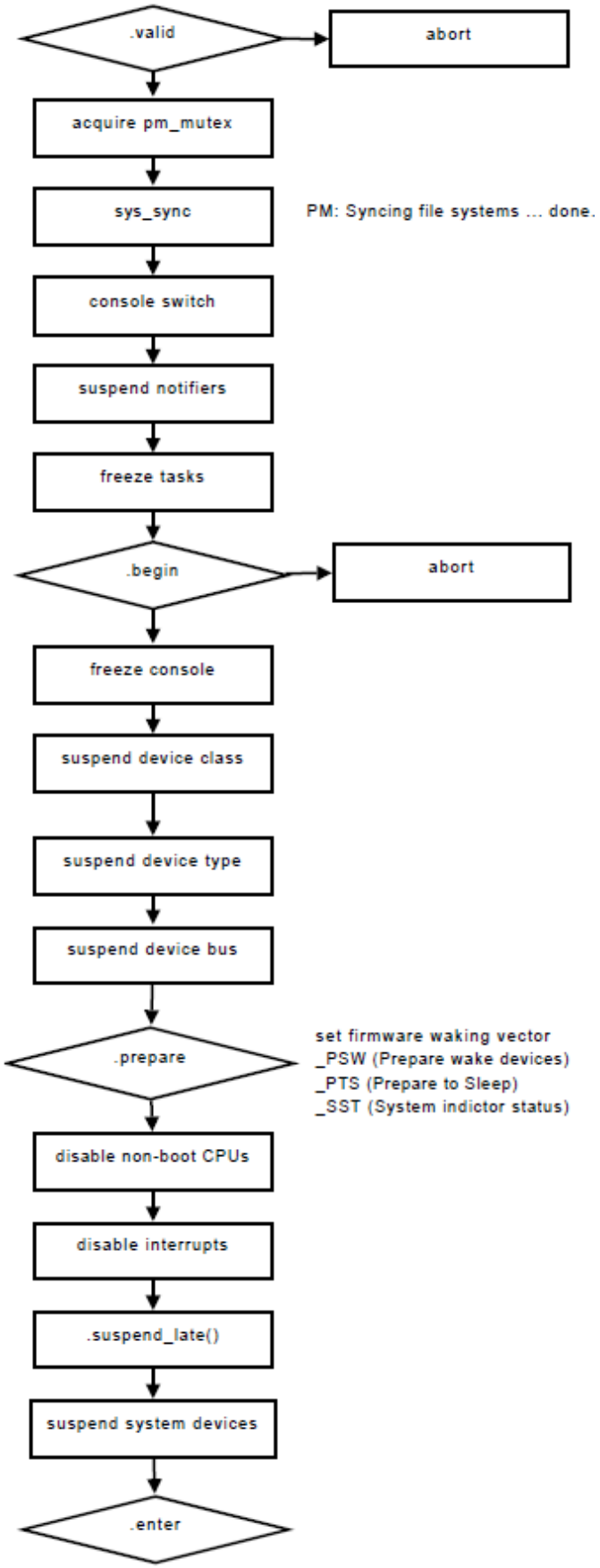
9. Suspend to RAM

Linux 支持 STANDBY、Suspend to RAM、Suspend to Disk 等形式的待机，一般的嵌入式产品仅仅实现 Suspend to RAM，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重新恢复系统。少数嵌入式 Linux 系统会实现 Suspend to Disk，它与 Suspend to RAM 的不同是前者并不关机，后者则把系统的状态保持于磁盘，然后 shutdown 整个系统。



通过向/sys/power/state 写入“mem”可开始 Suspend to RAM 的流程。在 Linux 内核

中，Suspend to RAM 的大致流程如下图（牵涉的操作包括同步文件系统、freeze 进程、设备驱动 suspend 以及系统的 suspend 入口等）：



在 Linux 内核 device_driver 结构中，含有一个 pm 成员，它是一个 dev_pm_ops 结构体指针，再该结构体中，封装了 Suspend to RAM 和 Suspend to Disk 所需要的 callback 函数：

```
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. };
```

目前比较推荐的做法是在 platform_driver、i2c_driver 和 spi_driver 等实例中，以上述结构体的形式封装 PM callback 函数。如 drivers/i2c/busses/i2c-omap.c 中的：

```
29. static struct dev_pm_ops omap_i2c_pm_ops = {
30.         SET_RUNTIME_PM_OPS(omap_i2c_runtime_suspend,
31.                            omap_i2c_runtime_resume, NULL)
32. };
33.
34. static struct platform_driver omap_i2c_driver = {
35.         .probe          = omap_i2c_probe,
36.         .remove         = __devexit_p(omap_i2c_remove),
37.         .driver         = {
38.                 .name   = "omap_i2c",
39.                 .owner  = THIS_MODULE,
40.                 .pm     = OMAP_I2C_PM_OPS,
41.                 .of_match_table = of_match_ptr(omap_i2c_of_match),
```

```
42.         },
43. };
```

但是 platform_driver、i2c_driver、spi_driver 等结构体中，仍然保留了过时的 suspend、resume 入口函数：

```
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，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()的逻辑就是：

```
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 suspend 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，resume 时则完成相反的操作。在 Suspend to RAM 的 suspend 和 resume 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 suspend 和 resume 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 suspend 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 中的：

```
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. }
```

上述代码中的 sirfsoc_pre_suspend_power_off()会将系统 resume 回来后重新开始执行的物理地址存入 SoC 相关的寄存器中（本例中为 SIRFSOC_PWRC_SCRATCH_PAD1），并进行设置唤醒源等操作：

```
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_PAD1);
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 SoC 相关的汇编实现的函数 sirfsoc_finish_suspend()真正完成最后的待机以及将系统置于深度睡眠，并置 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。

10. Runtime PM

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：

```
1. struct dev_pm_ops {
2.         ...
3.         int (*runtime_suspend)(struct device *dev);
4.         int (*runtime_resume)(struct device *dev);
5.         int (*runtime_idle)(struct device *dev);
6.         ...
7. };
```

Linux 提供一系列 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 suspend，执行相关的 runtime_suspend 函数。所谓相关的 runtime_suspend 函数，即依据如下顺序决定真正调用的 callback 函数:

```
1.         if (dev->pm_domain)
2.                 callback = dev->pm_domain->ops.runtime_suspend;
3.         else if (dev->type && dev->type->pm)
4.                 callback = dev->type->pm->runtime_suspend;
5.         else if (dev->class && dev->class->pm)
6.                 callback = dev->class->pm->runtime_suspend;
7.         else if (dev->bus && dev->bus->pm)
8.                 callback = dev->bus->pm->runtime_suspend;
9.         else
10.                callback = NULL;
11.
12.        if (!callback && dev->driver && dev->driver->pm)
13.                callback = dev->driver->pm->runtime_suspend;
```

只有当 dev->pm_domain、dev->type->pm、dev->class->pm、dev->bus->pm 相应 callback

函数都不存在的情况下，才直接进入设备驱动的的 runtime_suspend 函数执行。

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

“调度”设备的 suspend，延迟 delay 毫秒后将 suspend 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 suspend，autosuspend 的延迟到后，suspend 的工作项目被自动放入队列。

int pm_runtime_resume(struct device *dev);

引发设备的 resume，执行相关的 runtime_resume 函数。

int pm_request_resume(struct device *dev);

发起一个设备 resume 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 idle，执行相关的 runtime_idle 函数。

int pm_request_idle(struct device *dev);

发起一个设备 idle 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 runtime PM 支持。

int pm_runtime_disable(struct device *dev);

禁止设备的 runtime PM 支持。

int pm_runtime_get(struct device *dev);

增加设备的引用计数，类似于 clk_get()，会间接引发设备的 runtime_resume。

int pm_runtime_put(struct device *dev);

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

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 runtime PM 支持，在运行过程中动态地执行“pm_runtime_get() -> 做工作 -> pm_runtime_put()”的序列。如 drivers/i2c/busses/i2c-intel-mid.c I^2C 适配器驱动的 i2c_algorithm 的 master_xfer()成员函数 intel_mid_i2c_xfer()，在发起 I^2C 传输前执行 pm_runtime_get()，完成后执行 pm_runtime_put()：

```
1. static int intel_mid_i2c_xfer(struct i2c_adapter *adap,
2.                                struct i2c_msg *pmsg,
3.                                int num)
4. {
5.         pm_runtime_get(i2c->dev);
6.          ….
7.         pm_runtime_put(i2c->dev);
8. }
```

在具体的设备驱动中，直接使用上述引用计数的方法进行 suspend、idle 和 resume 不一定是合适的，因为 suspend 状态的进入和恢复需要一些时间，如果设备不会在 suspend 保留一定的时间，频繁进出 suspend 则反而带来新的开销。所以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 suspend（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 autosuspend 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 autospend 的 delay 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。一个典型用法如下：

```
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.     /* Send req result back to the user ... */
23. }
24.
25. int foo_runtime_suspend(struct device *dev)
26. {
27.     struct foo_priv foo = container_of(dev, ...);
28.     int ret = 0;
29.
30.     lock(&foo->private_lock);
31.     if (foo->num_pending_requests > 0) {
32.         ret = -EBUSY;
33.     } else {
34.         /* ... suspend the device ... */
35.         foo->is_suspended = 1;
36.     }
37.     unlock(&foo->private_lock);
38.     return ret;
39. }
40.
41. int foo_runtime_resume(struct device *dev)
42. {
```

```
43.                     struct foo_priv foo = container_of(dev, ...);
44.
45.                     lock(&foo->private_lock);
46.                     /* ... resume the device ... */
47.                     foo->is_suspended = 0;
48.                     pm_runtime_mark_last_busy(&foo->dev);
49.                     if (foo->num_pending_requests > 0)
50.                             foo_process_requests(foo);
51.                     unlock(&foo->private_lock);
52.                     return 0;
53.             }
```

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。