Linux 芯片级移植与底层驱动

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

1. SoC Linux 底层驱动的组成和现状

为了让 Linux 在一个全新的 ARM SoC 上运行，需要提供大量的底层支撑，如定时器节拍、中断控制器、SMP 启动、CPU hotplug 以及底层的 GPIO、clock、pinctrl 和 DMA 硬件的封装等。定时器节拍、中断控制器、SMP 启动和 CPU hotplug 这几部分相对来说没有像早期 GPIO、clock、pinctrl 和 DMA 的实现那么杂乱，基本上有个固定的套路。定时器节拍为 Linux 基于时间片的调度机制以及内核和用户空间的定时器提供支撑，中断控制器的驱动则使得 Linux 内核的工程师可以直接调用 local_irq_disable()、disable_irq()等通用的中断 API，而 SMP 启动支持则用于让 SoC 内部的多个 CPU 核都投入运行，CPU hotplug 则运行运行时挂载或拔除 CPU。这些工作，在 Linux 3.7 内核中，进行了良好的层次划分和架构设计。

在 GPIO、clock、pinctrl 和 DMA 驱动方面，Linux 2.6 时代，内核已或多或少有 GPIO、clock 等底层驱动的架构，但是核心层的代码太薄弱，各 SoC 对这些基础设施实现方面存在巨大差异,而且每个 SoC 仍然需要实现大量的代码。pinctrl 和 DMA 则最为混乱，几乎各家公司都定义了自己的独特的实现和 API。

社区必须改变这种局面，于是内核社区在 2011～2012 年进行了如下工作，这些工作在目前的 3.7 内核中基本准备就绪：

- ST-Ericsson 的工程师 Linus Walleij 提供了新的 pinctrl 驱动架构，内核新增加一个 drivers/pinctrl 目录，支撑 SoC 上的引脚复用，各个 SoC 的实现代码统一放入该目录；
- TI 的工程师 Mike Turquette 提供了 common clk 框架，让具体 SoC 实现 clk_ops 成员函数并通过 clk_register、clk_register_clkdev 注册时钟源以及源与设备对应关系，具体的 clock 驱动都统一迁移到 drivers/clk 目录；
- 建议各 SoC 统一采用 dmaengine 架构实现 DMA 驱动，该架构提供了通用的 DMA 通道 API 如 dmaengine_prep_slave_single()、dmaengine_submit()等，要求 SoC 实现 dma_device 的成员函数，实现代码统一放入 drivers/dma 目录；
- 在 GPIO 方面，drivers/gpio 下的 gpiolib 已能与新的 pinctrl 完美共存，实现引脚的 GPIO 和其他功能之间的复用，具体的 SoC 只需实现通用的 gpio_chip 结构体的成员函数。

经过以上工作，基本上就把芯片底层的基础架构方面的驱动的架构统一了，实现方法也统一了。另外，目前 GPIO、clock、pinmux 等功能都能良好的进行 Device Tree 的映射处理，譬如我们可以方面的在.dts 中定义一个设备要的时钟、pinmux 引脚以及 GPIO。

除了上述基础设施以外，在将 Linux 移植入新的 SoC 过程中，工程师常常强烈依赖于早期的 printk 功能，内核则提供了相关的 DEBUG_LL 和 EARLY_PRINTK 支持，只需要 SoC 提供商实现少量的 callback 或宏。

本文主要对上述各个组成部分进行架构上的剖析以及关键的实现部分的实例分析，以求完整归纳将 Linux 移植入新 SoC 的主要工作。本文基于 3.7.4 内核。

2. 用于操作系统节拍的 timer 驱动

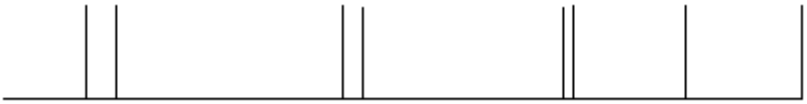
Linux 2.6 的早期（2.6.21 之前）基于 tick 设计，一般 SoC 公司在将 Linux 移植到自己的芯片上的时候，会从芯片内部找一个定时器，并将该定时器配置会 HZ 的频率，在每个时钟节拍到来时，调用 ARM Linux 内核核心层的 timer_tick()函数，从而引发系统里的一系列行为。如 2.6.17 中 arch/arm/mach-s3c2410/time.c 的做法是：

```
127/*
128 * IRQ handler for the timer
129 */
130static irqreturn_t
131s3c2410_timer_interrupt(int irq, void *dev_id, struct pt_regs *regs)
132{
133     write_seqlock(&xtime_lock);
134     timer_tick(regs);
135     write_sequnlock(&xtime_lock);
136     return IRQ_HANDLED;
137}
138
139static struct irqaction s3c2410_timer_irq = {
140     .name          = "S3C2410 Timer Tick",
141     .flags         = SA_INTERRUPT | SA_TIMER,
142     .handler       = s3c2410_timer_interrupt,
143};

252static void __init s3c2410_timer_init (void)
253{
254     s3c2410_timer_setup();
255     setup_irq(IRQ_TIMER4, &s3c2410_timer_irq);
256}
257
```

当前 Linux 多采用 tickless 方案，并支持高精度定时器，内核的配置一般会使能 NO_HZ（即 tickless，或者说动态 tick）和 HIGH_RES_TIMERS。要强调的是 tickless 并不是说系统中没有时钟节拍了，而是说这个节拍不再像以前那样，周期性地产生。Tickless 意味着，根据系统的运行情况，以事件驱动的方式动态决定下一个 tick 在何时发生。如果画一个时间轴，周期节拍的系统 tick 中断发生的时序看起来如下：

而 NO_HZ 的 Linux 看起来则是，2 次定时器中断发生的时间间隔可长可短：



在当前的 Linux 系统中，SoC 底层的 timer 被实现为一个 clock_event_device 和 clocksource 形式的驱动。在 clock_event_device 结构体中，实现其 set_mode()和 set_next_event()成员函数；在 clocksource 结构体中，主要实现 read()成员函数。而定时器中断服务程序中，不再调用 timer_tick()，而是调用 clock_event_device 的 event_handler()成员函数。一个典型的 SoC 的底层 tick 定时器驱动形如：

```
61static irqreturn_t xxx_timer_interrupt(int irq, void *dev_id)
62{
63     struct clock_event_device *ce = dev_id;
65     …
70     ce->event_handler(ce);
71
72     return IRQ_HANDLED;
73}
74
75/* read 64-bit timer counter */
```

```
76static cycle_t xxx_timer_read(struct clocksource *cs)
77{
78        u64 cycles;
79
80        /* read the 64-bit timer counter */
81        cycles = readl_relaxed(xxx_timer_base + XXX_TIMER_LATCHED_HI);
83        cycles = (cycles << 32) | readl_relaxed(xxx_timer_base + XXX_TIMER_LATCHED_LO);
84
85        return cycles;
86}
87
88static int xxx_timer_set_next_event(unsigned long delta,
89        struct clock_event_device *ce)
90{
91        unsigned long now, next;
92
93        writel_relaxed(XXX_TIMER_LATCH_BIT, xxx_timer_base + XXX_TIMER_LATCH);
94        now = readl_relaxed(xxx_timer_base + XXX_TIMER_LATCHED_LO);
95        next = now + delta;
96        writel_relaxed(next, xxx_timer_base + SIRFSOC_TIMER_MATCH_0);
97        writel_relaxed(XXX_TIMER_LATCH_BIT, xxx_timer_base + XXX_TIMER_LATCH);
98        now = readl_relaxed(xxx_timer_base + XXX_TIMER_LATCHED_LO);
99
100        return next - now > delta ? -ETIME : 0;
101}
102
103static void xxx_timer_set_mode(enum clock_event_mode mode,
104        struct clock_event_device *ce)
105{
107        switch (mode) {
108        case CLOCK_EVT_MODE_PERIODIC:
109                …
111        case CLOCK_EVT_MODE_ONESHOT:
112                …
114        case CLOCK_EVT_MODE_SHUTDOWN:
115                …
117        case CLOCK_EVT_MODE_UNUSED:
118        case CLOCK_EVT_MODE_RESUME:
119                break;
120        }
121}
144static struct clock_event_device xxx_clockevent = {
145        .name = "xxx_clockevent",
146        .rating = 200,
147        .features = CLOCK_EVT_FEAT_ONESHOT,
148        .set_mode = xxx_timer_set_mode,
149        .set_next_event = xxx_timer_set_next_event,
150};
151
152static struct clocksource xxx_clocksource = {
153        .name = "xxx_clocksource",
154        .rating = 200,
155        .mask = CLOCKSOURCE_MASK(64),
156        .flags = CLOCK_SOURCE_IS_CONTINUOUS,
157        .read = xxx_timer_read,
158        .suspend = xxx_clocksource_suspend,
159        .resume = xxx_clocksource_resume,
160};
161
162static struct irqaction xxx_timer_irq = {
163        .name = "xxx_tick",
164        .flags = IRQF_TIMER,
165        .irq = 0,
166        .handler = xxx_timer_interrupt,
167        .dev_id = &xxx_clockevent,
168};
169
```

```
176static void __init xxx_clockevent_init(void)
177{
178       clockevents_calc_mult_shift(&xxx_clockevent, CLOCK_TICK_RATE, 60);
179
180       xxx_clockevent.max_delta_ns =
181            clockevent_delta2ns(-2, &xxx_clockevent);
182       xxx_clockevent.min_delta_ns =
183            clockevent_delta2ns(2, &xxx_clockevent);
184
185       xxx_clockevent.cpumask = cpumask_of(0);
186       clockevents_register_device(&xxx_clockevent);
187}
188
189/* initialize the kernel jiffy timer source */
190static void __init xxx_timer_init(void)
191{
192       …
214
215       BUG_ON(clocksource_register_hz(&xxx_clocksource, CLOCK_TICK_RATE));
218
219       BUG_ON(setup_irq(xxx_timer_irq.irq, &xxx_timer_irq));
220
221       xxx_clockevent_init();
222}
249struct sys_timer xxx_timer = {
250       .init = xxx_timer_init,
251};
```

上述代码中，我们特别关注其中的如下函数：

clock_event_device 的 set_next_event 成员函数 xxx_timer_set_next_event()

该函数的 delta 参数是 Linux 内核传递给底层定时器的一个差值，它的含义是下一次 tick 中断产生的硬件定时器中计数器 counter 的值相对于当前 counter 的差值。我们在该函数中将硬件定时器设置为在“当前 counter 计数值” + delta 的时刻产生下一次 tick 中断。xxx_clockevent_init()函数中设置了可接受的最小和最大 delta 值对应的纳秒数，即 xxx_clockevent.min_delta_ns 和 xxx_clockevent.max_delta_ns。

clocksource 的 read 成员函数 xxx_timer_read()

该函数可读取出从开机以来到当前时刻定时器计数器已经走过的值，无论有没有设置计数器达到某值的时候产生中断，硬件的计数总是在进行的。因此，该函数给 Linux 系统提供了一个底层的准确的参考时间。

定时器的中断服务程序 xxx_timer_interrupt()

在该中断服务程序中，直接调用 clock_event_device 的 event_handler()成员函数，event_handler()成员函数的具体工作也是 Linux 内核根据 Linux 内核配置和运行情况自行设置的。

clock_event_device 的 set_mode 成员函数 xxx_timer_set_mode()

用于设置定时器的模式以及 resume 和 shutdown 等功能，目前一般采用 ONESHOT 模式，即一次一次产生中断。当然新版的 Linux 也可以使用老的周期性模式，如果内核编译的时候未选择 NO_HZ,该底层的 timer 驱动依然可以为内核的运行提供支持。

这些函数的结合，使得 ARM Linux 内核底层所需要的时钟得以运行。下面举一个典型的场景，假定定时器的晶振时钟频率为 1MHz（即计数器每加 1 等于 1us），应用程序透过 nanosleep() API 睡眠 100us，内核会据此换算出下一次定时器中断的 delta 值为 100，并间接调用到 xxx_timer_set_next_event()去设置硬件让其在 100us 后产生中断。100us 后，中断产生，xxx_timer_interrupt()被调用，event_handler()会间接唤醒睡眠的进程导致 nanosleep()函数返回，从而用户进程继续。

这里特别要强调的是，对于多核处理器来说，一般的做法是给每个核分配一个独立的定时器，各个核根据自身的运行情况动态设置自己时钟中断发生的时刻。看看我们说运行的电脑的 local timer 中断即知：

```
barry@barry-VirtualBox:~$ cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3
…
```

```
 20:      945        0        0        0   IO-APIC-fasteoi   vboxguest
 21:     4456        0        0    21592   IO-APIC-fasteoi   ahci, Intel 82801AA-ICH
 22:       26        0        0        0   IO-APIC-fasteoi   ohci_hcd:usb2
NMI:        0        0        0        0   Non-maskable interrupts
LOC:   177279   177517   177146   177139   Local timer interrupts
SPU:        0        0        0        0   Spurious interrupts
PMI:        0        0        0        0   Performance monitoring
…
```

而比较低效率的方法则是只给 CPU0 提供定时器，由 CPU0 将定时器中断透过 IPI（Inter Processor Interrupt，处理器间中断）广播到其他核。对于 ARM 来讲，1 号 IPIIPI_TIMER 就是来负责这个广播的，从 arch/arm/kernel/smp.c 可以看出：

```
62 enum ipi_msg_type {
 63        IPI_WAKEUP,
 64        IPI_TIMER,
 65        IPI_RESCHEDULE,
 66        IPI_CALL_FUNC,
 67        IPI_CALL_FUNC_SINGLE,
 68        IPI_CPU_STOP,
 69 };
```

3. 中断控制器驱动

在 Linux 内核中，各个设备驱动可以简单地调用 request_irq()、enable_irq()、disable_irq()、local_irq_disable()、local_irq_enable()等通用 API 完成中断申请、使能、禁止等功能。在将 Linux 移植到新的 SoC 时，芯片供应商需要提供该部分 API 的底层支持。

local_irq_disable()、local_irq_enable()的实现与具体中断控制器无关，对于 ARMv6 以上的体系架构而言，是直接调用 CPSID/CPSIE 指令进行，而对于 ARMv6 以前的体系结构，则是透过 MRS、MSR 指令来读取和设置 ARM 的 CPSR 寄存器。由此可见，local_irq_disable()、local_irq_enable()针对的并不是外部的中断控制器，而是直接让 CPU 本身不响应中断请求。相关的实现位于 arch/arm/include/asm/irqflags.h：

```
 11#if __LINUX_ARM_ARCH__ >= 6
 12
 13static inline unsigned long arch_local_irq_save(void)
 14{
 15        unsigned long flags;
 16
 17        asm volatile(
 18                "       mrs     %0, cpsr        @ arch_local_irq_save\n"
 19                "       cpsid   i"
 20                : "=r" (flags) : : "memory", "cc");
 21        return flags;
 22}
 23
 24static inline void arch_local_irq_enable(void)
 25{
 26        asm volatile(
 27                "       cpsie i                 @ arch_local_irq_enable"
 28                :
 29                :
 30                : "memory", "cc");
 31}
 32
 33static inline void arch_local_irq_disable(void)
 34{
 35        asm volatile(
 36                "       cpsid i                 @ arch_local_irq_disable"
 37                :
 38                :
 39                : "memory", "cc");
 40}
```

```
44#else
45
46/*
47 * Save the current interrupt enable state & disable IRQs
48 */
49static inline unsigned long arch_local_irq_save(void)
50{
51        unsigned long flags, temp;
52
53        asm volatile(
54                "       mrs     %0, cpsr        @ arch_local_irq_save\n"
55                "       orr     %1, %0, #128\n"
56                "       msr     cpsr_c, %1"
57                : "=r" (flags), "=r" (temp)
58                :
59                : "memory", "cc");
60        return flags;
61}
62
63/*
64 * Enable IRQs
65 */
66static inline void arch_local_irq_enable(void)
67{
68        unsigned long temp;
69        asm volatile(
70                "       mrs     %0, cpsr        @ arch_local_irq_enable\n"
71                "       bic     %0, %0, #128\n"
72                "       msr     cpsr_c, %0"
73                : "=r" (temp)
74                :
75                : "memory", "cc");
76}
77
78/*
79 * Disable IRQs
80 */
81static inline void arch_local_irq_disable(void)
82{
83        unsigned long temp;
84        asm volatile(
85                "       mrs     %0, cpsr        @ arch_local_irq_disable\n"
86                "       orr     %0, %0, #128\n"
87                "       msr     cpsr_c, %0"
88                : "=r" (temp)
89                :
90                : "memory", "cc");
91}
92 #endif
```

与 local_irq_disable()和 local_irq_enable()不同，disable_irq()、enable_irq()针对的则是外部的中断控制器。在内核中，透过 irq_chip 结构体来描述中断控制器。该结构体内部封装了中断 mask、unmask、ack 等成员函数，其定义于 include/linux/irq.h：

```
303struct irq_chip {
304        const char      *name;
305        unsigned int    (*irq_startup)(struct irq_data *data);
306        void            (*irq_shutdown)(struct irq_data *data);
307        void            (*irq_enable)(struct irq_data *data);
308        void            (*irq_disable)(struct irq_data *data);
309
310        void            (*irq_ack)(struct irq_data *data);
311        void            (*irq_mask)(struct irq_data *data);
312        void            (*irq_mask_ack)(struct irq_data *data);
313        void            (*irq_unmask)(struct irq_data *data);
314        void            (*irq_eoi)(struct irq_data *data);
315
316        int             (*irq_set_affinity)(struct irq_data *data, const struct cpumask *dest, bool force);
```

```
317     int         (*irq_retrigger)(struct irq_data *data);
318     int         (*irq_set_type)(struct irq_data *data, unsigned int flow_type);
319     int         (*irq_set_wake)(struct irq_data *data, unsigned int on);
334};
```

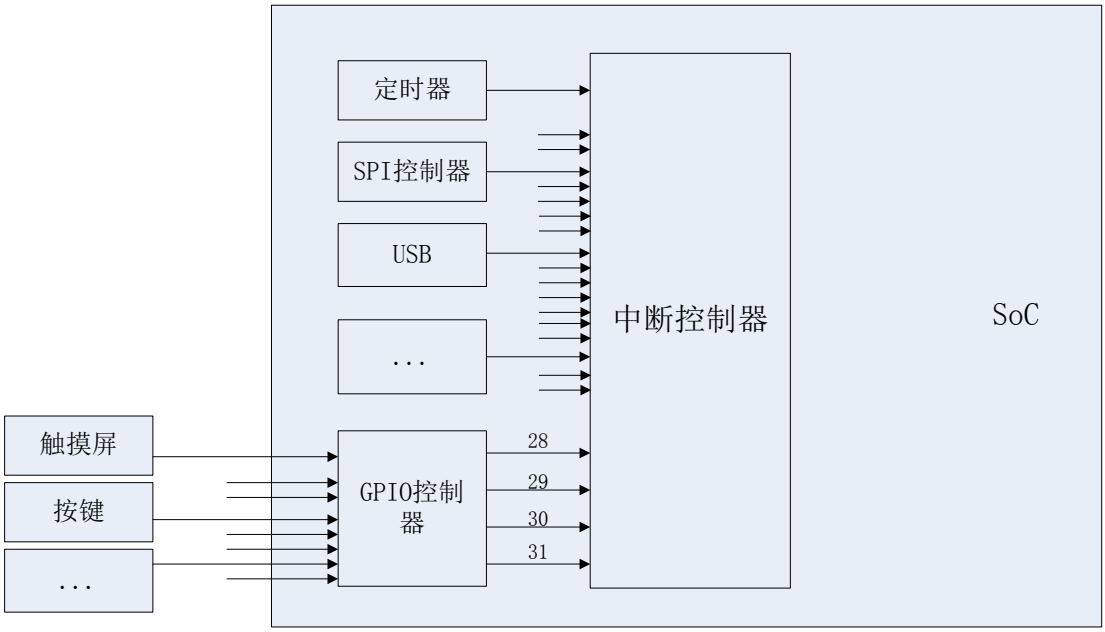
各个芯片公司会将芯片内部的中断控制器实现为 irq_chip 驱动的形式。受限于中断控制器硬件的能力，这些成员函数并不一定需要全部实现，有时候只需要实现其中的部分函数即可。譬如 drivers/pinctrl/pinctrl-sirf.c 驱动中的

```
1438static struct irq_chip sirfsoc_irq_chip = {
1439    .name = "sirf-gpio-irq",
1440    .irq_ack = sirfsoc_gpio_irq_ack,
1441    .irq_mask = sirfsoc_gpio_irq_mask,
1442    .irq_unmask = sirfsoc_gpio_irq_unmask,
1443    .irq_set_type = sirfsoc_gpio_irq_type,
1444};
```

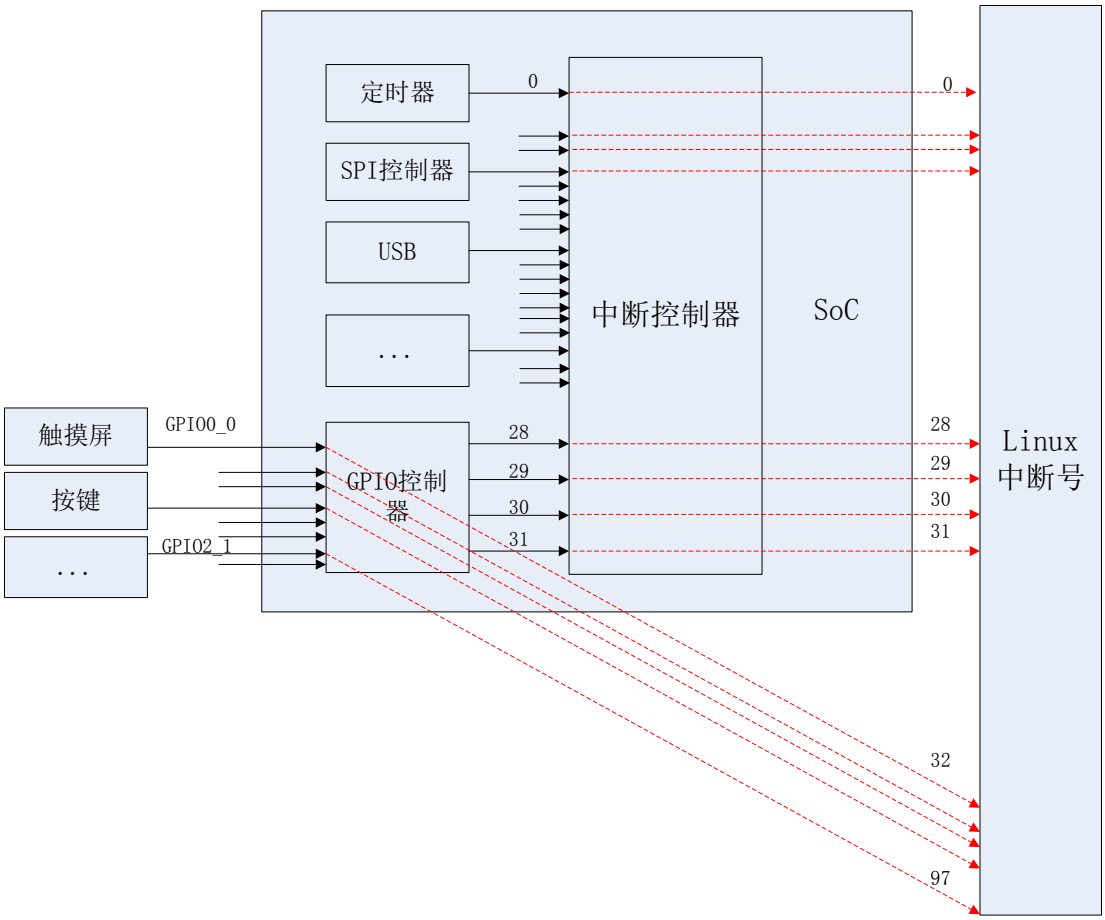
我们只实现了其中的 ack、mask、unmask 和 set_type 成员函数，ack 函数用于清中断，mask、unmask 用于中断屏蔽和取消中断屏蔽、set_type 则用于配置中断的触发方式，如高电平、低电平、上升沿、下降沿等。至于 enable_irq()的时候，虽然没有实现 irq_enable 成员函数，但是内核会间接调用到 irq_unmask 成员函数，这点从 kernel/irq/chip.c 可以看出：

```
192void irq_enable(struct irq_desc *desc)
193{
194     irq_state_clr_disabled(desc);
195     if (desc->irq_data.chip->irq_enable)
196         desc->irq_data.chip->irq_enable(&desc->irq_data);
197     else
198         desc->irq_data.chip->irq_unmask(&desc->irq_data);
199     irq_state_clr_masked(desc);
200}
```

在芯片内部，中断控制器可能不止 1 个，多个中断控制器之间还很可能是级联的。举个例子，假设芯片内部有一个中断控制器，支持 32 个中断源，其中有 4 个来源于 GPIO 控制器外围的 4 组 GPIO，每组 GPIO 上又有 32 个中断（许多芯片的 GPIO 控制器也同时是一个中断控制器），其关系如下图：



那么，一般来讲，在实际操作中，gpio0_0——gpio0_31 这些引脚本身在第 1 级会使用中断号 28，而这些引脚本身的中断号在实现 GPIO 控制器对应的 irq_chip 驱动时，我们又会把它映射到 Linux 系统的 32——63 号中断。同理，gpio1_0——gpio1_31 这些引脚本身在第 1 级会使用中断号 29，而这些引脚本身的中断号在实现 GPIO 控制器对应的 irq_chip 驱动时，我们又会把它映射到 Linux 系统的 64——95 号中断，以此类推。对于中断号的使用者而言，无需看到这种 2 级映射关系。如果某设备想申请 gpio1_0 这个引脚对应的中断，它只需要申请 64 号中断即可。这个关系图看起来如下：



还是以 drivers/pinctrl/pinctrl-sirf.c 的 irq_chip 部分为例，我们对于每组 GPIO 都透过 irq_domain_add_legacy()添加了相应的 irq_domain，每组 GPIO 的中断号开始于 SIRFSOC_GPIO_IRQ_START + i * SIRFSOC_GPIO_BANK_SIZE，而每组 GPIO 本身占用的第 1 级中断控制器的中断号则为 bank->parent_irq，我们透过 irq_set_chained_handler()设置了第 1 级中断发生的时候，会调用链式 IRQ 处理函数 sirfsoc_gpio_handle_irq()：

```
1689            bank->domain = irq_domain_add_legacy(np, SIRFSOC_GPIO_BANK_SIZE,
1690                    SIRFSOC_GPIO_IRQ_START + i * SIRFSOC_GPIO_BANK_SIZE, 0,
1691                    &sirfsoc_gpio_irq_simple_ops, bank);
1692
1693            if (!bank->domain) {
1694                    pr_err("%s: Failed to create irqdomain\n", np->full_name);
1695                    err = -ENOSYS;
1696                    goto out;
1697            }
1698
1699            irq_set_chained_handler(bank->parent_irq, sirfsoc_gpio_handle_irq);
1700            irq_set_handler_data(bank->parent_irq, bank);
```

而在 sirfsoc_gpio_handle_irq()函数的入口出调用 chained_irq_enter()暗示自身进入链式 IRQ 处理，在函数体内判决具体的 GPIO 中断，并透过 generic_handle_irq()调用到最终的外设驱动中的中断服务程序，最后调用 chained_irq_exit()暗示自身退出链式 IRQ 处理：

```
1446static void sirfsoc_gpio_handle_irq(unsigned int irq, struct irq_desc *desc)
1447{
1448     …
1454     chained_irq_enter(chip, desc);
1456     …
1477     generic_handle_irq(first_irq + idx);
1478     …
1484     chained_irq_exit(chip, desc);
1485}
```

很多中断控制器的寄存器定义呈现出简单的规律，如有一个 mask 寄存器，其中每 1 位可屏蔽 1 个中断等，这种情况下，我们无需实现 1 个完整的 irq_chip 驱动，可以使用内核提供的通用 irq_chip 驱动架构 irq_chip_generic，这样只需要实现极少量的代码，如 arch/arm/mach-prima2/irq.c 中，注册 CSR SiRFprimaII 内部中断控制器的代码仅为：

```
26static __init void
27sirfsoc_alloc_gc(void __iomem *base, unsigned int irq_start, unsigned int num)
28{
29      struct irq_chip_generic *gc;
30      struct irq_chip_type *ct;
31
32      gc = irq_alloc_generic_chip("SIRFINTC", 1, irq_start, base, handle_level_irq);
33      ct = gc->chip_types;
34
35      ct->chip.irq_mask = irq_gc_mask_clr_bit;
36      ct->chip.irq_unmask = irq_gc_mask_set_bit;
37      ct->regs.mask = SIRFSOC_INT_RISC_MASK0;
38
39      irq_setup_generic_chip(gc, IRQ_MSK(num), IRQ_GC_INIT_MASK_CACHE, IRQ_NOREQUEST, 0);
40}
```

特别值得一提的是，目前多数主流 ARM 芯片，内部的一级中断控制器都使用了 ARM 公司的 GIC，我们几乎不需要实现任何代码，只需要在 Device Tree 中添加相关的结点并将 gic_handle_irq()填入 MACHINE 的 handle_irq 成员。

如在 arch/arm/boot/dts/exynos5250.dtsi 即含有：

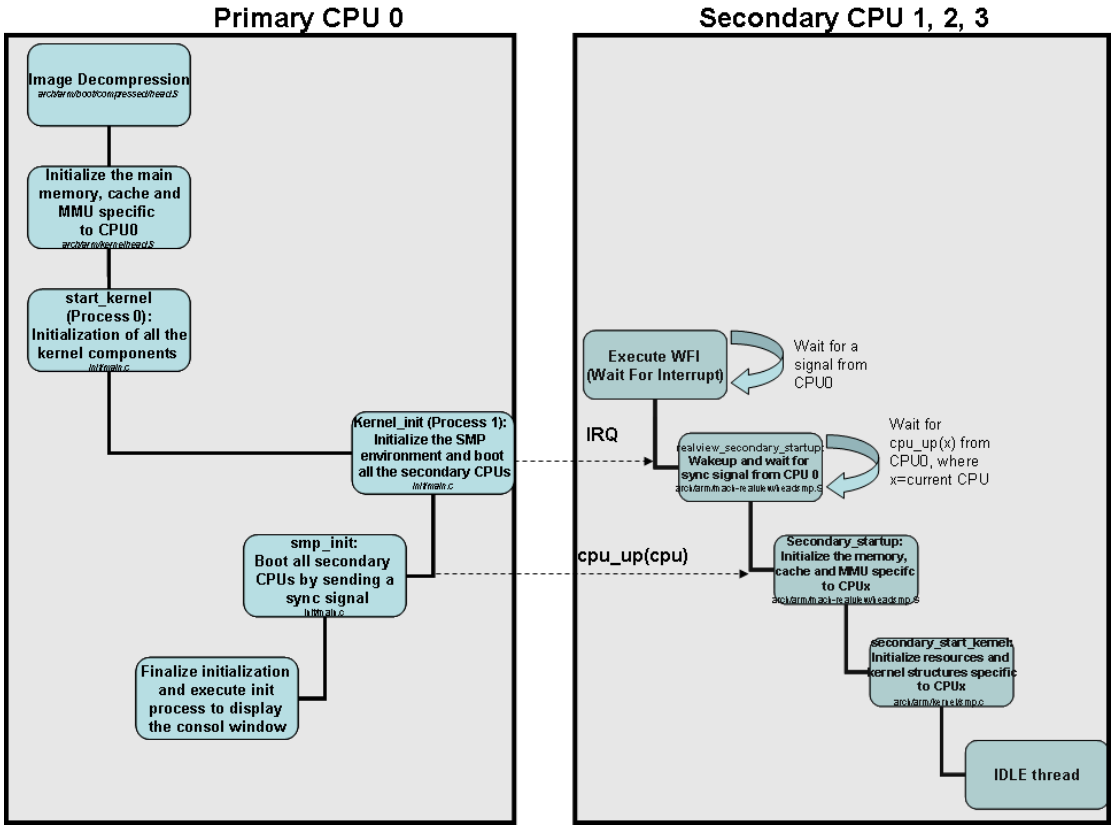
```
36      gic:interrupt-controller@10481000 {
37              compatible = "arm,cortex-a9-gic";
38              #interrupt-cells = <3>;
39              interrupt-controller;
40              reg = <0x10481000 0x1000>, <0x10482000 0x2000>;
41      };
```

而在 arch/arm/mach-exynos/mach-exynos5-dt.c 中即含有：

```
95DT_MACHINE_START(EXYNOS5_DT, "SAMSUNG EXYNOS5 (Flattened Device Tree)")
96      /* Maintainer: Kukjin Kim <kgene.kim@samsung.com> */
97      .init_irq       = exynos5_init_irq,
98      .smp            = smp_ops(exynos_smp_ops),
99      .map_io         = exynos5250_dt_map_io,
100     .handle_irq     = gic_handle_irq,
101     .init_machine   = exynos5250_dt_machine_init,
102     .init_late      = exynos_init_late,
103     .timer          = &exynos4_timer,
104     .dt_compat      = exynos5250_dt_compat,
105     .restart        = exynos5_restart,
106MACHINE_END
```

4. SMP 多核启动以及 CPU 热插拔驱动

在 Linux 系统中，对于多核的 ARM 芯片而言，Bootrom 代码中，CPU0 会率先起来，引导 Bootloader 和 Linux 内核执行，而其他的核则在上电时 Bootrom 一般将自身置于 WFI 或者 WFE 状态，并等待 CPU0 给其发 CPU 核间中断（IPI）或事件（一般透过 SEV 指令）唤醒之。一个典型的启动过程如下图：



被 CPU_0 唤醒的 CPU_n 可以在运行过程中进行热插拔。譬如运行如下命令即可卸载 CPU1 并且将 CPU1 上的任务全部迁移到其他 CPU：

echo 0 > /sys/devices/system/cpu/cpu1/online

同样地，运行如下命令可以再次启动 CPU1:

echo 1 > /sys/devices/system/cpu/cpu1/online

之后 CPU1 会主动参与系统中各个 CPU 之间要运行任务的负载均衡工作。

CPU0 唤醒其他 CPU 的动作在内核中被封装为一个 smp_operations 的结构体，该结构体的成员如下：

```
 83struct smp_operations {
 84#ifdef CONFIG_SMP
 85        /*
 86         * Setup the set of possible CPUs (via set_cpu_possible)
 87         */
 88        void (*smp_init_cpus)(void);
 89        /*
 90         * Initialize cpu_possible map, and enable coherency
 91         */
 92        void (*smp_prepare_cpus)(unsigned int max_cpus);
 93
 94        /*
 95         * Perform platform specific initialisation of the specified CPU.
 96         */
 97        void (*smp_secondary_init)(unsigned int cpu);
 98        /*
 99         * Boot a secondary CPU, and assign it the specified idle task.
100         * This also gives us the initial stack to use for this CPU.
101         */
102        int  (*smp_boot_secondary)(unsigned int cpu, struct task_struct *idle);
103#ifdef CONFIG_HOTPLUG_CPU
104        int  (*cpu_kill)(unsigned int cpu);
105        void (*cpu_die)(unsigned int cpu);
106        int  (*cpu_disable)(unsigned int cpu);
107#endif
108#endif
109};
```

我们从 arch/arm/mach-vexpress/v2m.c 看到 VEXPRESS 电路板用到的 smp_ops 为 vexpress_smp_ops：

```
666DT_MACHINE_START(VEXPRESS_DT, "ARM-Versatile Express")
667     .dt_compat    = v2m_dt_match,
668     .smp          = smp_ops(vexpress_smp_ops),
669     .map_io       = v2m_dt_map_io,
670     .init_early   = v2m_dt_init_early,
671     .init_irq     = v2m_dt_init_irq,
672     .timer        = &v2m_dt_timer,
673     .init_machine = v2m_dt_init,
674     .handle_irq   = gic_handle_irq,
675     .restart      = v2m_restart,
676MACHINE_END
```

透过 arch/arm/mach-vexpress/platsmp.c 的实现代码可以看出，smp_operations 的成员函数 smp_init_cpus() 即 vexpress_smp_init_cpus()会探测 SoC 内 CPU 核的个数，并设置了核间通信的方式为 gic_raise_softirq()。可见于 vexpress_smp_init_cpus()中调用的 vexpress_dt_smp_init_cpus()：

```
103static void __init vexpress_dt_smp_init_cpus(void)
104{
…
128     for (i = 0; i < ncores; ++i)
129             set_cpu_possible(i, true);
130
131     set_smp_cross_call(gic_raise_softirq);
132}
```

而 smp_operations 的成员函数 smp_prepare_cpus()即 vexpress_smp_prepare_cpus()则会透过 v2m_flags_set(virt_to_phys(versatile_secondary_startup))设置其他 CPU 的启动地址为 versatile_secondary_startup：

```
179static void __init vexpress_smp_prepare_cpus(unsigned int max_cpus)
180{
181     …
189
190     /*
191      * Write the address of secondary startup into the
192      * system-wide flags register. The boot monitor waits
193      * until it receives a soft interrupt, and then the
194      * secondary CPU branches to this address.
195      */
196     v2m_flags_set(virt_to_phys(versatile_secondary_startup));
197}
```

注意这部分的具体实现方法是 SoC 相关的，由芯片的设计以及芯片内部的 Bootrom 决定。对于 VEXPRESS 来讲，设置方法如下：

```
139void __init v2m_flags_set(u32 data)
140{
141     writel(~0, v2m_sysreg_base + V2M_SYS_FLAGSCLR);
142     writel(data, v2m_sysreg_base + V2M_SYS_FLAGSSET);
143}
```

即填充 v2m_sysreg_base + V2M_SYS_FLAGSCLR 地址为 0xFFFFFFFF,将其他 CPU 初始启动执行的指令地址填入 v2m_sysreg_base + V2M_SYS_FLAGSSET。这 2 个地址属于芯片实现时候设定的。填入的 CPUn 的起始地址都透过 virt_to_phys()转化为物理地址，因为此时 CPUn 的 MMU 尚未开启。

比较关键的是 smp_operations 的成员函数 smp_boot_secondary()，它完成最终的 CPU_n 的唤醒工作：

```
27static void __cpuinit write_pen_release(int val)
28{
29     pen_release = val;
30     smp_wmb();
31     __cpuc_flush_dcache_area((void *)&pen_release, sizeof(pen_release));
32     outer_clean_range(__pa(&pen_release), __pa(&pen_release + 1));
33}

59int __cpuinit versatile_boot_secondary(unsigned int cpu, struct task_struct *idle)
```

```
60{
61      unsigned long timeout;
62
63      /*
64       * Set synchronisation state between this boot processor
65       * and the secondary one
66       */
67      spin_lock(&boot_lock);
68
69      /*
70       * This is really belt and braces; we hold unintended secondary
71       * CPUs in the holding pen until we're ready for them.  However,
72       * since we haven't sent them a soft interrupt, they shouldn't
73       * be there.
74       */
75      write_pen_release(cpu_logical_map(cpu));
76
77      /*
78       * Send the secondary CPU a soft interrupt, thereby causing
79       * the boot monitor to read the system wide flags register,
80       * and branch to the address found there.
81       */
82      gic_raise_softirq(cpumask_of(cpu), 0);
83
84      timeout = jiffies + (1 * HZ);
85      while (time  before(jiffies, timeout)) {
86              smp_rmb();
87              if (pen_release == -1)
88                      break;
89
90              udelay(10);
91      }
92
93      /*
94       * now the secondary core is starting up let it run its
95       * calibrations, then wait for it to finish
96       */
97      spin_unlock(&boot_lock);
98
99      return pen_release != -1 ? -ENOSYS : 0;
100}
```

上述代码中高亮的部分首先会将 pen_release 变量设置为要唤醒的 CPU 核的 CPU 号 cpu_logical_map(cpu)，而后透过 gic_raise_softirq(cpumask_of(cpu), 0)给 CPU_{cpu} 发起 0 号 IPI，这个时候，CPU_{cpu} 核会从前面 smp_operations 中的 smp_prepare_cpus()成员函数即 vexpress_smp_prepare_cpus()透过 v2m_flags_set()设置的其他 CPU 核的起始地址 versatile_secondary_startup 开始执行，如果顺利的话，该 CPU 会将原先为正数的 pen_release 写为-1，以便 CPU0 从等待 pen_release 成为-1 的循环中跳出。

versatile_secondary_startup 实现于 arch/arm/plat-versatile/headsmp.S，是一段汇编：

```
21ENTRY(versatile_secondary_startup)
22      mrc     p15, 0, r0, c0, c0, 5
23      and     r0, r0, #15
24      adr     r4, 1f
25      ldmia   r4, {r5, r6}
26      sub     r4, r4, r5
27      add     r6, r6, r4
28pen:   ldr     r7, [r6]
29      cmp     r7, r0
30      bne     pen
31
32      /*
33       * we've been released from the holding pen: secondary_stack
34       * should now contain the SVC stack for this core
35       */
36      b       secondary_startup
37
```

```
38      .align
391:    .long   .
40      .long   pen_release
41ENDPROC(versatile_secondary_startup)
```

第 1 段高亮的部分实际上是等待 pen_release 成为 CPU0 设置的 cpu_logical_map(cpu)，一般直接就成立了。第 2 段高亮的部分则调用到内核通用的 secondary_startup()函数，经过一系列的初始化如 MMU 等，最终新的被唤醒的 CPU 将调用到 smp_operations 的 smp_secondary_init()成员函数，对于本例为 versatile_secondary_init()：

```
37void __cpuinit versatile_secondary_init(unsigned int cpu)
38{
39      /*
40       * if any interrupts are already enabled for the primary
41       * core (e.g. timer irq), then they will not have been enabled
42       * for us: do so
43       */
44      gic_secondary_init(0);
45
46      /*
47       * let the primary processor know we're out of the
48       * pen, then head off into the C entry point
49       */
50      write_pen_release(-1);
51
52      /*
53       * Synchronise with the boot thread.
54       */
55      spin_lock(&boot_lock);
56      spin_unlock(&boot_lock);
57}
```

上述代码中高亮的那 1 行会将 pen_release 写为-1，于是 CPU0 还在执行的 versatile_boot_secondary()函数中的如下循环就退出了：

```
85      while (time_before(jiffies, timeout)) {
86              smp_rmb();
87              if (pen_release == -1)
88                      break;
89
90              udelay(10);
91      }
```

此后 CPU0 和新唤醒的其他 CPU 各自狂奔。整个系统在运行过程中会进行实时进程和正常进程的动态负载均衡。

CPU hotplug 的实现也是芯片相关的，对于 VEXPRESS 而言，实现了 smp_operations 的 cpu_die()成员函数即 vexpress_cpu_die()。它会在进行 CPU_n 的拔除操作时将 CPU_n 投入低功耗的 WFI 状态，相关代码位于 arch/arm/mach-vexpress/hotplug.c：

```
90void __ref vexpress_cpu_die(unsigned int cpu)
91{
92      int spurious = 0;
93
94      /*
95       * we're ready for shutdown now, so do it
96       */
97      cpu_enter_lowpower();
98      platform_do_lowpower(cpu, &spurious);
99
100     /*
101      * bring this CPU back into the world of cache
102      * coherency, and then restore interrupts
103      */
104     cpu_leave_lowpower();
105
106     if (spurious)
107             pr_warn("CPU%u: %u spurious wakeup calls\n", cpu, spurious);
108}
57static inline void platform_do_lowpower(unsigned int cpu, int *spurious)
```

```
58{
59      /*
60       * there is no power-control hardware on this platform, so all
61       * we can do is put the core into WFI; this is safe as the calling
62       * code will have already disabled interrupts
63       */
64      for (;;) {
65              wfi();
66
67              if (pen_release == cpu_logical_map(cpu)) {
68                      /*
69                       * OK, proper wakeup, we're done
70                       */
71                      break;
72              }
73
74              /*
75               * Getting here, means that we have come out of WFI without
76               * having been woken up - this shouldn't happen
77               *
78               * Just note it happening - when we're woken, we can report
79               * its occurrence.
80               */
81              (*spurious)++;
82      }
83}
```

CPU_n 睡眠于 wfi()，之后再次 online 的时候，又会因为 CPU_0 给它发出的 IPI 而从 wfi() 函数返回继续执行，醒来时 CPU_n 也判决了是否 pen_release == cpu_logical_map(cpu)成立，以确定该次醒来确确实实是由 CPU_0 唤醒的一次正常醒来。

5. DEBUG_LL 和 EARLY_PRINTK

在 Linux 启动的早期，console 驱动还没有投入运行。当我们把 Linux 移植到一个新的 SoC 的时候，工程师一般非常需要早期就可以执行 printk()功能以跟踪调试启动过程。内核的 DEBUG_LL 和 EARLY_PRINTK 选项为我们提供了这样的支持。而在 Bootloader 引导内核执行的 bootargs 中，则需要使能 earlyprintk 选项。

为了让 DEBUG_LL 和 EARLY_PRINTK 可以运行，Linux 内核中需实现早期解压过程打印需要的 putc()和后续的 addruart、senduart 和 waituart 等宏。以 CSR SiRFprimaII 为例，putc()的实现位于 arch/arm/mach-prima2/include/mach/uncompress.h：

```
22static __inline__ void putc(char c)
23{
24      /*
25       * during kernel decompression, all mappings are flat:
26       *  virt_addr == phys_addr
27       */
28      while (__raw_readl((void __iomem *)SIRFSOC_UART1_PA_BASE +
SIRFSOC_UART_TXFIFO_STATUS)
29              & SIRFSOC_UART1_TXFIFO_FULL)
30              barrier();
31
32      __raw_writel(c, (void __iomem *)SIRFSOC_UART1_PA_BASE + SIRFSOC_UART_TXFIFO_DATA);
33}
```

由于解压过程中，MMU 还没有初始化，所以这个时候的打印是直接往 UART 端口 FIFO 对应的物理地址丢打印字符。

addruart、senduart 和 waituart 等宏的实现位于每个 SoC 对应的 MACHINE 代码目录的 include/mach/debug-macro.S ， SiRFprimaII 的实现 mach-prima2/include/mach/debug-macro.S 如下：

```
12      .macro  addruart, rp, rv, tmp
13      ldr     \rp, =SIRFSOC_UART1_PA_BASE           @ physical
```

```
14      ldr     \rv, =SIRFSOC_UART1_VA_BASE           @ virtual
15      .endm
16
17      .macro  senduart,rd,rx
18      str     \rd, [\rx, #SIRFSOC_UART_TXFIFO_DATA]
19      .endm
20
21      .macro  busyuart,rd,rx
22      .endm
23
24      .macro  waituart,rd,rx
251001:   ldr     \rd, [\rx, #SIRFSOC_UART_TXFIFO_STATUS]
26      tst     \rd, #SIRFSOC_UART1_TXFIFO_EMPTY
27      beq     1001b
28      .endm
```

其中的 senduart 完成了往 UART 的 FIFO 丢打印字符的过程。waituart 则相当于一个流量握手，等待 FIFO 为空。这些宏最终会被内核的 arch/arm/kernel/debug.S 引用。

6. GPIO 驱动

在 drivers/gpio 下实现了通用的基于 gpiolib 的 GPIO 驱动，其中定义了一个通用的用于描述底层 GPIO 控制器的 gpio_chip 结构体，并要求具体的 SoC 实现 gpio_chip 结构体的成员函数，最后透过 gpiochip_add()注册 gpio_chip。

gpio_chip 结构体封装了底层的硬件的 GPIO enable/disable 等操作，它定义为：

```
94struct gpio_chip {
95      const char              *label;
96      struct device           *dev;
97      struct module           *owner;
98
99      int                     (*request)(struct gpio_chip *chip,
100                                     unsigned offset);
101     void                    (*free)(struct gpio_chip *chip,
102                                     unsigned offset);
103
104     int                     (*direction_input)(struct gpio_chip *chip,
105                                     unsigned offset);
106     int                     (*get)(struct gpio_chip *chip,
107                                     unsigned offset);
108     int                     (*direction_output)(struct gpio_chip *chip,
109                                     unsigned offset, int value);
110     int                     (*set_debounce)(struct gpio_chip *chip,
111                                     unsigned offset, unsigned debounce);
112
113     void                    (*set)(struct gpio_chip *chip,
114                                     unsigned offset, int value);
115
116     int                     (*to_irq)(struct gpio_chip *chip,
117                                     unsigned offset);
118
119     void                    (*dbg_show)(struct seq_file *s,
120                                     struct gpio_chip *chip);
121     int                     base;
122     u16                     ngpio;
123     const char              *const *names;
124     unsigned                can_sleep:1;
125     unsigned                exported:1;
126
127#if defined(CONFIG_OF_GPIO)
128     /*
129      * If CONFIG_OF is enabled, then all GPIO controllers described in the
130      * device tree automatically may have an OF translation
131      */
```

```
132        struct device_node *of_node;
133        int of_gpio_n_cells;
134        int (*of_xlate)(struct gpio_chip *gc,
135                        const struct of_phandle_args *gpiospec, u32 *flags);
136#endif
137};
```

透过这层封装，每个具体的要用到 GPIO 的设备驱动都使用通用的 GPIO API 来操作 GPIO，这些 API 主要用于 GPIO 的申请、释放和设置：

```
int gpio_request(unsigned gpio, const char *label);
void gpio_free(unsigned gpio);
int gpio_direction_input(unsigned gpio);
int gpio_direction_output(unsigned gpio, int value);
int gpio_set_debounce(unsigned gpio, unsigned debounce);
int gpio_get_value_cansleep(unsigned gpio);
void gpio_set_value_cansleep(unsigned gpio, int value);
int gpio_request_one(unsigned gpio, unsigned long flags, const char *label);
int gpio_request_array(const struct gpio *array, size_t num);
void gpio_free_array(const struct gpio *array, size_t num);
int devm_gpio_request(struct device *dev, unsigned gpio, const char *label);
int devm_gpio_request_one(struct device *dev, unsigned gpio,
                  unsigned long flags, const char *label);
void devm_gpio_free(struct device *dev, unsigned int gpio);
```

注意，内核中针对内存、IRQ、时钟、GPIO、pinctrl 都有 devm_开头的 API，使用这部分 API 的时候，内核会有类似于 Java 资源自动回收机制，因此在代码中做出错处理时，无需释放相关的资源。

对于 GPIO 而言，特别值得一提的是，内核会创建/sys 结点 /sys/class/gpio/gpioN/，透过它我们可以 echo 值从而改变 GPIO 的方向、设置和获取 GPIO 的值。

在拥有 Device Tree 支持的情况之下，我们可以透过 Device Tree 来描述某 GPIO 控制器提供的 GPIO 引脚被具体设备使用的情况。在 GPIO 控制器对应的结点中，需定义#gpio-cells 和 gpio-controller 属性，具体的设备结点则透过 xxx-gpios 属性来引用 GPIO 控制器结点及 GPIO 引脚。

如 VEXPRESS 电路板 DT 文件 arch/arm/boot/dts/vexpress-v2m.dtsi 中拥有如下 GPIO 控制器结点：

```
73                v2m_sysreg: sysreg@00000 {
74                    compatible = "arm,vexpress-sysreg";
75                    reg = <0x00000 0x1000>;
76                    gpio-controller;
77                    #gpio-cells = <2>;
78                };
```

VEXPRESS 电路板上的 MMC 控制器会使用该结点 GPIO 控制器提供的 GPIO 引脚，则具体的 mmci@05000 设备结点的会通过-gpios 属性引用 GPIO:

```
111               mmci@05000 {
112                   compatible = "arm,pl180", "arm,primecell";
113                   reg = <0x05000 0x1000>;
114                   interrupts = <9 10>;
115                   cd-gpios = <&v2m_sysreg 0 0>;
116                   wp-gpios = <&v2m_sysreg 1 0>;
117                   …
121               };
```

其中的 cd-gpios 用于 SD/MMC 卡的 detection，而 wp-gpios 用于写保护，MMC 主机控制器驱动会透过如下方法获取这 2 个 GPIO，详见于 drivers/mmc/host/mmci.c：

```
1220 static void mmci_dt_populate_generic_pdata(struct device_node *np,
1221                                 struct mmci_platform_data *pdata)
1222 {
1223       int bus_width = 0;
1224
1225       pdata->gpio_wp = of_get_named_gpio(np, "wp-gpios", 0);
1226       pdata->gpio_cd = of_get_named_gpio(np, "cd-gpios", 0);
…
}
```

7. pinctrl 驱动

许多 SoC 内部都包含 pin 控制器，通过 pin 控制器的寄存器，我们可以配置一个或者一组引脚的功能和特性。在软件上，Linux 内核的 pinctrl 驱动可以操作 pin 控制器为我们完成如下工作：

- 枚举并且命名 pin 控制器可控制的所有引脚；
- 提供引脚复用的能力；
- 提供配置引脚的能力，如驱动能力、上拉下拉、开漏（open drain）等。

pinctrl 和引脚

在特定 SoC 的 pinctrl 驱动中，我们需要定义引脚。假设有一个 PGA 封装的芯片的引脚排布如下：

```
    A  B  C  D  E  F  G  H

8   o  o  o  o  o  o  o  o

7   o  o  o  o  o  o  o  o
6   o  o  o  o  o  o  o  o
5   o  o  o  o  o  o  o  o
4   o  o  o  o  o  o  o  o
3   o  o  o  o  o  o  o  o
2   o  o  o  o  o  o  o  o
1   o  o  o  o  o  o  o  o
```

在 pinctrl 驱动初始化的时候，需要向 pinctrl 子系统注册一个 pinctrl_desc 描述符，在该描述符中包含所有引脚的列表。可以通过如下代码来注册这个 pin 控制器并命名其所有引脚：

```
59 #include <linux/pinctrl/pinctrl.h>
60
61 const struct pinctrl_pin_desc foo_pins[] = {
62      PINCTRL_PIN(0, "A8"),
63      PINCTRL_PIN(1, "B8"),
64      PINCTRL_PIN(2, "C8"),
65      ...
66      PINCTRL  PIN(61, "F1"),
67      PINCTRL_PIN(62, "G1"),
68      PINCTRL_PIN(63, "H1"),
69 };
70
71 static struct pinctrl_desc foo_desc = {
72        .name = "foo",
73        .pins = foo_pins,
74        .npins = ARRAY_SIZE(foo_pins),
75        .maxpin = 63,
76        .owner = THIS_MODULE,
77 };
78
79 int __init foo_probe(void)
80 {
81        struct pinctrl_dev *pctl;
82
83        pctl = pinctrl_register(&foo_desc, <PARENT>, NULL);
84        if (IS_ERR(pctl))
85               pr_err("could not register foo pin driver\n");
86 }
```

引脚组（pin group）

在 pinctrl 子系统中，支持将一组引脚绑定为同一功能。假设{ 0, 8, 16, 24 }这一组引脚承担 SPI 的功能，而{ 24, 25 }这一组引脚承担 I^2C 接口功能。在驱动的代码中，需要体现这个分组关系，并且为这些分组实现 pinctrl_ops 的成员函数 get_groups_count、get_groups_count 和 get_groups_count，将 pinctrl_ops 填充到前文 pinctrl_desc 的实例 foo_desc 中。

```
130 #include <linux/pinctrl/pinctrl.h>
131
132 struct foo_group {
133         const char *name;
134         const unsigned int *pins;
135         const unsigned num_pins;
136 };
137
138 static const unsigned int spi0_pins[] = { 0, 8, 16, 24 };
139 static const unsigned int i2c0_pins[] = { 24, 25 };
140
141 static const struct foo_group foo_groups[] = {
142         {
143                 .name = "spi0_grp",
144                 .pins = spi0_pins,
145                 .num_pins = ARRAY_SIZE(spi0_pins),
146         },
147         {
148                 .name = "i2c0_grp",
149                 .pins = i2c0_pins,
150                 .num_pins = ARRAY_SIZE(i2c0_pins),
151         },
152 };
153
154
155 static int foo_get_groups_count(struct pinctrl_dev *pctldev)
156 {
157         return ARRAY_SIZE(foo_groups);
158 }
159
160 static const char *foo_get_group_name(struct pinctrl_dev *pctldev,
161                                       unsigned selector)
162 {
163         return foo_groups[selector].name;
164 }
165
166 static int foo_get_group_pins(struct pinctrl_dev *pctldev, unsigned selector,
167                               unsigned ** const pins,
168                               unsigned * const num_pins)
169 {
170         *pins = (unsigned *) foo  groups[selector].pins;
171         *num_pins = foo_groups[selector].num_pins;
172         return 0;
173 }
174
175 static struct pinctrl_ops foo_pctrl_ops = {
176         .get_groups_count = foo_get_groups_count,
177         .get_group_name = foo_get_group_name,
178         .get_group_pins = foo_get_group_pins,
179 };
180
181
182 static struct pinctrl_desc foo_desc = {
183         ...
184         .pctlops = &foo_pctrl_ops,
185 };
```

get_groups_count()成员函数用于告知 pinctrl 子系统该 SoC 中合法的被选引脚组有多少个，而 get_group_name()则提供引脚组的名字，get_group_pins()提供引脚组的引脚表。在

设备驱动调用 pinctrl 通用 API 使能某一组引脚的对应功能时，pinctrl 子系统的核心层会调用上述 callback 函数。

引脚配置

设备驱动有时候需要配置引脚，譬如可能把引脚设置为高阻或者三态（达到类似断连引脚的效果）,或通过某阻值将引脚上拉/下拉以确保默认状态下引脚的电平状态。驱动中可以自定义相应板级引脚配置 API 的细节，譬如某设备驱动可能通过如下代码将某引脚上拉：

```
#include <linux/pinctrl/consumer.h>
ret = pin_config_set("foo-dev", "FOO_GPIO_PIN", PLATFORM_X_PULL_UP);
```

其中的 PLATFORM_X_PULL_UP 由特定的 pinctrl 驱动定义。在特定的 pinctrl 驱动中，需要实现完成这些配置所需要的 callback 函数（pinctrl_desc 的 confops 成员函数）：

```
222 #include <linux/pinctrl/pinctrl.h>
223 #include <linux/pinctrl/pinconf.h>
224 #include "platform_x_pindefs.h"
225
226 static int foo_pin_config_get(struct pinctrl_dev *pctldev,
227                     unsigned offset,
228                     unsigned long *config)
229 {
230         struct my_conftype conf;
231
232         ... Find setting for pin @ offset ...
233
234         *config = (unsigned long) conf;
235 }
236
237 static int foo  pin  config  set(struct pinctrl_dev *pctldev,
238                     unsigned offset,
239                     unsigned long config)
240 {
241         struct my_conftype *conf = (struct my_conftype *) config;
242
243         switch (conf) {
244               case PLATFORM_X_PULL_UP:
245               ...
246               }
247         }
248 }
249
250 static int foo_pin_config_group_get (struct pinctrl_dev *pctldev,
251                     unsigned selector,
252                     unsigned long *config)
253 {
254         ...
255 }
256
257 static int foo_pin_config_group_set (struct pinctrl_dev *pctldev,
258                     unsigned selector,
259                     unsigned long config)
260 {
261         ...
262 }
263
264 static struct pinconf_ops foo_pconf_ops = {
265         .pin_config_get = foo_pin_config_get,
266         .pin_config_set = foo_pin_config_set,
267         .pin_config_group_get = foo_pin_config_group_get,
268         .pin_config_group_set = foo_pin_config_group_set,
269 };
270
271 /* Pin config operations are handled by some pin controller */
272 static struct pinctrl_desc foo_desc = {
273         ...
274         .confops = &foo_pconf_ops,
```

```
275 };
```

其中的 pin_config_group_get()、pin_config_group_set()针对的是可同时配置一个引脚组的状态情况，而 pin_config_get()、pin_config_set()针对的则是单个引脚的配置。

与 GPIO 子系统的交互

pinctrl 驱动中所覆盖的引脚可能同时可作为 GPIO 用，内核的 GPIO 子系统和 pinctrl 子系统本来是并行工作的，但是有时候需要交叉映射，这种情况下，需要在 pinctrl 驱动中告知 pinctrl 子系统核心层 GPIO 与底层 pinctrl 驱动所管理的引脚之间的映射关系。假设 pinctrl 驱动中定义的引脚 32~47 与 gpio_chip 实例 chip_a 的 GPIO 对应，引脚 64~71 与 gpio_chip 实例 chip_b 的 GPIO 对应，即映射关系为：

chip a:
- GPIO range : [32 .. 47]
- pin range : [32 .. 47]

chip b:
- GPIO range : [48 .. 55]
- pin range : [64 .. 71]

则在特定 pinctrl 驱动中可以透过如下代码注册 2 个 GPIO 范围：

```
305 struct gpio_chip chip_a;
306 struct gpio_chip chip_b;
307
308 static struct pinctrl_gpio_range gpio_range_a = {
309         .name = "chip a",
310         .id = 0,
311         .base = 32,
312         .pin_base = 32,
313         .npins = 16,
314         .gc = &chip_a;
315 };
316
317 static struct pinctrl_gpio_range gpio_range_b = {
318         .name = "chip b",
319         .id = 0,
320         .base = 48,
321         .pin_base = 64,
322         .npins = 8,
323         .gc = &chip_b;
324 };
325
326 {
327         struct pinctrl_dev *pctl;
328         ...
329         pinctrl_add_gpio_range(pctl, &gpio_range_a);
330         pinctrl_add_gpio_range(pctl, &gpio_range_b);
331 }
```

在基于内核 gpiolib 的 GPIO 驱动中，若设备驱动需进行 GPIO 申请 gpio_request()和释放 gpio_free()，GPIO 驱动则会调用 pinctrl 子系统中的 pinctrl_request_gpio()和 pinctrl_free_gpio()通用 API,pinctrl 子系统会查找申请的 GPIO 和 pin 的映射关系，并确认引脚是否被其他复用功能所占用。与 pinctrl 子系统通用层 pinctrl_request_gpio()和 pinctrl_free_gpio() API 对应，在底层的具体 pinctrl 驱动中，需要实现 **pinmux_ops** 结构体的 gpio_request_enable()和 gpio_disable_free()成员函数。

除了 gpio_request_enable()和 gpio_disable_free()成员函数外，**pinmux_ops** 结构体主要还用来封装 pinmux 功能 enable/disable 的 callback 函数，下面可以看到它更多的细节。

引脚复用（pinmux）

pinctrl 驱动中可处理引脚复用，它定义了 FUNCTIONS（功能），驱动可以设置某 FUNCTIONS 的 enable 或者 disable。各个 FUNCTIONS 联合起来组成一个一维数组，譬如{ spi0, i2c0, mmc0 }就描述了 3 个不同的 FUNCTIONS。

一个特定的功能总是要求一些引脚组（pin group）来完成，引脚组的数量为可以为 1 个或者多个。假设对前文所描述的 PGA 封装的 SoC 而言，如下图：

```
388       A   B   C   D   E   F   G   H
389      +---+
390    8 | o | o   o   o   o   o   o   o
391      |   |
392    7 | o | o   o   o   o   o   o   o
393      |   |
394    6 | o | o   o   o   o   o   o   o
395      +---+---+
396    5 | o | o | o   o   o   o   o   o
397      +---+---+           +---+
398    4   o   o   o   o   o   o | o | o
399                        |   |
400    3   o   o   o   o   o   o | o | o
401                        |   |
402    2   o   o   o   o   o   o | o | o
403      +-------+-------+-------+---+---+
404    1 | o   o | o   o | o   o | o | o |
405      +-------+-------+-------+---+---+
```

I^2C 功能由{ A5, B5 }引脚组成，而在定义引脚描述的 pinctrl_pin_desc 结构体实例 foo_pins 的时候，将它们的序号定义为了{ 24, 25 }；而 SPI 功能则由可以由{ A8, A7, A6, A5 }和 { G4, G3, G2, G1 }, 也即{ 0, 8, 16, 24 }和{ 38, 46, 54, 62 }两个引脚组完成（注意在整个系统中，引脚组的名字不会重叠）。

由此，功能和引脚组的组合就可以决定一组引脚在系统里的作用，因此在设置某组引脚的作用时，pinctrl 的核心层会将功能的序号以及引脚组的序号传递给底层 pinctrl 驱动中相关的 callback 函数。

在整个系统中，驱动或板级代码调用 pinmux 相关的 API 获取引脚后，会形成一个（pinctrl、使用引脚的设备、功能、引脚组）的映射关系，假设在某电路板上，将让 spi0 设备使用 pinctrl0 的 fspi0 功能以及 gspi0 引脚组，让 i2c0 设备使用 pinctrl0 的 fi2c0 功能和 gi2c0 引脚组，我们将得到如下的映射关系：

```
502  {
503   {"map-spi0", spi0, pinctrl0, fspi0, gspi0},
504   {"map-i2c0", i2c0, pinctrl0, fi2c0, gi2c0}
505  }
```

pinctrl 子系统的核心会保证每个引脚的排他性，因此一个引脚如果已经被某设备用掉了，而其他的设备又申请该引脚行驶其他的功能或 GPIO，则 pinctrl 核心层会让该次申请失败。

特定 pinctrl 驱动中 pinmux 相关的代码主要处理如何 enable/disable 某一{功能，引脚组}的组合，譬如，当 spi0 设备申请 pinctrl0 的 fspi0 功能和 gspi0 引脚组以便将 gspi0 引脚组配置为 SPI 接口时，相关的 callback 被组织进一个 pinmux_ops 结构体，而该结构体的实例最终成为前文 pinctrl_desc 的 pmxops 成员：

```
562 #include <linux/pinctrl/pinctrl.h>
563 #include <linux/pinctrl/pinmux.h>
564
565 struct foo_group {
566         const char *name;
567         const unsigned int *pins;
568         const unsigned num_pins;
569 };
570
571 static const unsigned spi0_0_pins[] = { 0, 8, 16, 24 };
572 static const unsigned spi0_1_pins[] = { 38, 46, 54, 62 };
573 static const unsigned i2c0_pins[] = { 24, 25 };
574 static const unsigned mmc0_1_pins[] = { 56, 57 };
575 static const unsigned mmc0_2_pins[] = { 58, 59 };
576 static const unsigned mmc0_3_pins[] = { 60, 61, 62, 63 };
577
578 static const struct foo_group foo_groups[] = {
579         {
```

```
580                 .name = "spi0_0_grp",
581                 .pins = spi0_0_pins,
582                 .num_pins = ARRAY_SIZE(spi0_0_pins),
583         },
584         {
585                 .name = "spi0_1_grp",
586                 .pins = spi0_1_pins,
587                 .num_pins = ARRAY_SIZE(spi0_1_pins),
588         },
589         {
590                 .name = "i2c0_grp",
591                 .pins = i2c0_pins,
592                 .num_pins = ARRAY_SIZE(i2c0_pins),
593         },
594         {
595                 .name = "mmc0_1_grp",
596                 .pins = mmc0_1_pins,
597                 .num_pins = ARRAY_SIZE(mmc0_1_pins),
598         },
599         {
600                 .name = "mmc0_2_grp",
601                 .pins = mmc0_2_pins,
602                 .num_pins = ARRAY_SIZE(mmc0_2_pins),
603         },
604         {
605                 .name = "mmc0  3  grp",
606                 .pins = mmc0_3_pins,
607                 .num_pins = ARRAY_SIZE(mmc0_3_pins),
608         },
609 };
610
611
612 static int foo_get_groups_count(struct pinctrl_dev *pctldev)
613 {
614         return ARRAY_SIZE(foo_groups);
615 }
616
617 static const char *foo_get_group_name(struct pinctrl_dev *pctldev,
618                                       unsigned selector)
619 {
620         return foo_groups[selector].name;
621 }
622
623 static int foo_get_group_pins(struct pinctrl_dev *pctldev, unsigned selector,
624                               unsigned ** const pins,
625                               unsigned * const num_pins)
626 {
627         *pins = (unsigned *) foo_groups[selector].pins;
628         *num_pins = foo_groups[selector].num_pins;
629         return 0;
630 }
631
632 static struct pinctrl_ops foo_pctrl_ops = {
633         .get_groups_count = foo_get_groups_count,
634         .get_group_name = foo_get_group_name,
635         .get_group_pins = foo_get_group_pins,
636 };
637
638 struct foo  pmx  func {
639         const char *name;
640         const char * const *groups;
641         const unsigned num_groups;
642 };
643
644 static const char * const spi0_groups[] = { "spi0_0_grp", "spi0_1_grp" };
645 static const char * const i2c0_groups[] = { "i2c0_grp" };
646 static const char * const mmc0_groups[] = { "mmc0_1_grp", "mmc0_2_grp",
```

```
647                                     "mmc0_3_grp" };
648
649 static const struct foo_pmx_func foo_functions[] = {
650         {
651                 .name = "spi0",
652                 .groups = spi0_groups,
653                 .num_groups = ARRAY_SIZE(spi0_groups),
654         },
655         {
656                 .name = "i2c0",
657                 .groups = i2c0_groups,
658                 .num_groups = ARRAY_SIZE(i2c0_groups),
659         },
660         {
661                 .name = "mmc0",
662                 .groups = mmc0_groups,
663                 .num_groups = ARRAY_SIZE(mmc0_groups),
664         },
665 };
666
667 int foo_get_functions_count(struct pinctrl_dev *pctldev)
668 {
669         return ARRAY_SIZE(foo_functions);
670 }
671
672 const char *foo_get_fname(struct pinctrl_dev *pctldev, unsigned selector)
673 {
674         return foo_functions[selector].name;
675 }
676
677 static int foo_get_groups(struct pinctrl_dev *pctldev, unsigned selector,
678                           const char * const **groups,
679                           unsigned * const num_groups)
680 {
681         *groups = foo_functions[selector].groups;
682         *num  groups = foo_functions[selector].num_groups;
683         return 0;
684 }
685
686 int foo_enable(struct pinctrl_dev *pctldev, unsigned selector,
687                unsigned group)
688 {
689         u8 regbit = (1 << selector + group);
690
691         writeb((readb(MUX)|regbit), MUX)
692         return 0;
693 }
694
695 void foo_disable(struct pinctrl_dev *pctldev, unsigned selector,
696                unsigned group)
697 {
698         u8 regbit = (1 << selector + group);
699
700         writeb((readb(MUX) & ~(regbit)), MUX)
701         return 0;
702 }
703
704 struct pinmux_ops foo_pmxops = {
705         .get  functions  count = foo  get  functions_count,
706         .get_function_name = foo_get_fname,
707         .get_function_groups = foo_get_groups,
708         .enable = foo_enable,
709         .disable = foo_disable,
710 };
711
712 /* Pinmux operations are handled by some pin controller */
713 static struct pinctrl_desc foo_desc = {
```

```
714         ...
715         .pctlops = &foo_pctrl_ops,
716         .pmxops = &foo_pmxops,
717 };
718
```

具体的 pinctrl、使用引脚的设备、功能、引脚组的映射关系，可以在板文件中透过定义 pinctrl_map 结构体的实例来展开，如：

```
828 static struct pinctrl_map __initdata mapping[] = {
829         PIN_MAP_MUX_GROUP("foo-i2c.o", PINCTRL_STATE_DEFAULT, "pinctrl-foo", NULL, "i2c0"),
830 };
```

PIN_MAP_MUX_GROUP 是一个快捷的宏，用于赋值 pinctrl_map 的各个成员：

```
88 #define PIN_MAP_MUX_GROUP(dev, state, pinctrl, grp, func)           \
89         {                                                           \
90                 .dev_name = dev,                                    \
91                 .name = state,                                      \
92                 .type = PIN_MAP_TYPE_MUX_GROUP,                     \
93                 .ctrl_dev_name = pinctrl,                           \
94                 .data.mux = {                                       \
95                         .group = grp,                               \
96                         .function = func,                           \
97                 },                                                  \
98         }
99
```

当然，这种映射关系最好是在 Device Tree 中透过结点的属性进行，具体的结点属性的定义方法依赖于具体的 pinctrl 驱动，最终在 pinctrl 驱动中透过 pinctrl_ops 结构体的.dt_node_to_map()成员函数读出属性并建立映射表。

又由于 1 个功能可能可由 2 个不同的引脚组实现，可能形成如下对于同 1 个功能有 2 个可选引脚组的 pinctrl_map：

```
static struct pinctrl_map __initdata mapping[] = {
    PIN_MAP_MUX_GROUP("foo-spi.0", "spi0-pos-A", "pinctrl-foo", "spi0_0_grp", "spi0"),
    PIN_MAP_MUX_GROUP("foo-spi.0", "spi0-pos-B", "pinctrl-foo", "spi0_1_grp", "spi0"),
};
```

在运行时，我们可以透过类似的 API 去查找并设置位置 A 的引脚组行驶 SPI 接口的功能：

```
954         p = devm_pinctrl_get(dev);
955         s = pinctrl_lookup_state(p, "spi0-pos-A ");
956         ret = pinctrl_select_state(p, s);
```

或者可以更加简单地使用：

```
p = devm_pinctrl_get_select(dev, "spi0-pos-A");
```

若想运行时切换位置 A 和 B 的引脚组行使 SPI 的接口功能，代码结构类似：

```
1163 foo_probe()
1164 {
1165         /* Setup */
1166         p = devm_pinctrl_get(&device);
1167         if (IS_ERR(p))
1168                 ...
1169
1170         s1 = pinctrl_lookup_state(foo->p, " spi0-pos-A");
1171         if (IS_ERR(s1))
1172                 ...
1173
1174         s2 = pinctrl_lookup_state(foo->p, " spi0-pos-B");
1175         if (IS_ERR(s2))
1176                 ...
1177 }
1178
1179 foo_switch()
1180 {
1181         /* Enable on position A */
1182         ret = pinctrl_select_state(s1);
1183         if (ret < 0)
1184             ...
1185
```

```
1186        ...
1187
1188        /* Enable on position B */
1189        ret = pinctrl_select_state(s2);
1190        if (ret < 0)
1191            ...
1192
1193        ...
1194 }
```

pinctrl 子系统中定义了 pinctrl_get_select_default()以及有 devm_前缀的 devm_pinctrl_get_select() API，许多驱动如 drivers/i2c/busses/i2c-imx.c、drivers/leds/leds-gpio.c、drivers/spi/spi-imx.c、drivers/tty/serial/omap-serial.c、sound/soc/mxs/mxs-saif.c 都是透过这一 API 来获取自己的引脚组的。xxx_get_select_default()最终会调用

```
pinctrl_get_select(dev, PINCTRL_STATE_DEFAULT);
```

其中 PINCTRL_STATE_DEFAULT 定义为"default"，它描述了缺省状态下某设备的 pinmux 功能和引脚组映射情况。

8. clock 驱动

在一个 SoC 中，晶振、PLL、divider 和 gate 等会形成一个 clock 树形结构，在 Linux 2.6 中，也存有 clk_get_rate()、clk_set_rate()、clk_get_parent()、clk_set_parent()等通用 API，但是这些 API 由每个 SoC 单独实现，而且各个 SoC 供应商在实现方面的差异很大，于是内核增加了一个新的 common clk 框架以解决这个碎片化问题。之所以称为 common clk，这个 common 主要体现在：

- 统一的 clk 结构体，统一的定义于 clk.h 中的 clk API，这些 API 会调用到统一的 clk_ops 中的 callback 函数；

这个统一的 clk 结构体的定义如下：

```
struct clk {
        const char              *name;
        const struct clk_ops    *ops;
        struct clk_hw           *hw;
        char                    **parent_names;
        struct clk              **parents;
        struct clk              *parent;
        struct hlist_head       children;
        struct hlist_node       child_node;
        ...
};
```

其中的 clk_ops 定义为：

```
struct clk_ops {
        int             (*prepare)(struct clk_hw *hw);
        void            (*unprepare)(struct clk_hw *hw);
        int             (*enable)(struct clk_hw *hw);
        void            (*disable)(struct clk_hw *hw);
        int             (*is_enabled)(struct clk_hw *hw);
        unsigned long   (*recalc_rate)(struct clk_hw *hw,
                                unsigned long parent_rate);
        long            (*round_rate)(struct clk_hw *hw, unsigned long,
                                unsigned long *);
        int             (*set_parent)(struct clk_hw *hw, u8 index);
        u8              (*get_parent)(struct clk_hw *hw);
        int             (*set_rate)(struct clk_hw *hw, unsigned long);
        void            (*init)(struct clk_hw *hw);
};
```

- 对于具体的 SoC 如何去实现针对自己 SoC 的 clk 驱动， 如何提供硬件特定的 callback 函数的方法也进行了统一。

在 common 的 clk 结构体中，clk_hw 是联系 clk_ops 中 callback 函数和具体硬件细节的纽带，clk_hw 中只包含 common clk 结构体的指针以及具体硬件的 init 数据：

```
struct clk_hw {
        struct clk *clk;
        const struct clk_init_data *init;
};
```

其中的 clk_init_data 包含了具体时钟的 name、可能的 parent 的 name 的列表 parent_names、可能的 parent 数量 num_parents 等，实际上这些 name 的匹配对建立时钟间的父子关系功不可没：

```
136 struct clk_init_data {
137         const char              *name;
138         const struct clk_ops    *ops;
139         const char              **parent_names;
140         u8                      num_parents;
141         unsigned long           flags;
142 };
```

从 clk 核心层到具体芯片 clk 驱动的调用顺序为：

clk_enable(clk);

➔ clk->ops->enable(clk->hw);

通用的 clk API（如 clk_enable）在调用底层的 clk 结构体的 clk_ops 的成员函数（如 clk->ops->enable）时，会将 clk->hw 传递过去。

一般在具体的驱动中会定义针对特定 clk（如 foo）的结构体，该结构体中包含 clk_hw 成员以及硬件私有数据：

```
struct clk_foo {
    struct clk_hw hw;
    ... hardware specific data goes here ...
};
```

并定义 to_clk_foo()宏以便通过 clk_hw 获取 clk_foo：

```
#define to_clk_foo(_hw) container_of(_hw, struct clk_foo, hw)
```

在针对 clk_foo 的 clk_ops 的 callback 函数中我们便可以透过 clk_hw 和 to_clk_foo 最终获得硬件私有数据并访问硬件读写寄存器以改变时钟的状态：

```
struct clk_ops clk_foo_ops {
    .enable         = &clk_foo_enable;
    .disable        = &clk_foo_disable;
};
int clk_foo_enable(struct clk_hw *hw)
{
    struct clk_foo *foo;

    foo = to_clk_foo(hw);

    ... perform magic on foo ...

    return 0;
};
```

在具体的 clk 驱动中，需要透过 clk_register()以及它的变体注册硬件上所有的 clk，通过 clk_register_clkdev()注册 clk 与使用 clk 的设备之间的映射关系，也即进行 clk 和使用 clk 的设备之间的绑定，这 2 个函数的原型为：

```
struct clk *clk_register(struct device *dev, struct clk_hw *hw);
int clk_register_clkdev(struct clk *clk, const char *con_id,
        const char *dev_fmt, ...);
```

另外，针对不同的 clk 类型（如固定频率的 clk、clk gate、clk divider 等），clk 子系统又提供了几个快捷函数以完成 clk_register()的过程：

```
struct clk *clk_register_fixed_rate(struct device *dev, const char *name,
            const char *parent_name, unsigned long flags,
            unsigned long fixed_rate);
struct clk *clk_register_gate(struct device *dev, const char *name,
            const char *parent_name, unsigned long flags,
            void __iomem *reg, u8 bit_idx,
            u8 clk_gate_flags, spinlock_t *lock);
struct clk *clk_register_divider(struct device *dev, const char *name,
            const char *parent_name, unsigned long flags,
```

```
        void __iomem *reg, u8 shift, u8 width,
        u8 clk_divider_flags, spinlock_t *lock);
```

以 drivers/clk/clk-prima2.c 为例，该驱动对应的芯片 SiRFprimaII 外围接了一个 26MHz 的晶振和一个 32.768KHz 供给的 RTC 的晶振，在 26MHz 晶振的后面又有 3 个 PLL，当然 PLL 后面又接了更多的 clk 结点，则我们看到它的相关驱动代码形如：

```
static unsigned long pll_clk_recalc_rate(struct clk_hw *hw,
        unsigned long parent_rate)
{
        unsigned long fin = parent_rate;
        struct clk_pll *clk = to_pllclk(hw);
        …
}

static long pll_clk_round_rate(struct clk_hw *hw, unsigned long rate,
        unsigned long *parent_rate)
{
        …
}

static int pll_clk_set_rate(struct clk_hw *hw, unsigned long rate,
        unsigned long parent_rate)
{
        …
}

static struct clk_ops std_pll_ops = {
        .recalc_rate = pll_clk_recalc_rate,
        .round_rate = pll_clk_round_rate,
        .set_rate = pll_clk_set_rate,
};

static const char *pll_clk_parents[] = {
        "osc",
};

static struct clk_init_data clk_pll1_init = {
        .name = "pll1",
        .ops = &std_pll_ops,
        .parent_names = pll_clk_parents,
        .num_parents = ARRAY_SIZE(pll_clk_parents),
};

static struct clk_init_data clk_pll2_init = {
        .name = "pll2",
        .ops = &std_pll_ops,
        .parent_names = pll_clk_parents,
        .num_parents = ARRAY_SIZE(pll_clk_parents),
};

static struct clk_init_data clk_pll3_init = {
        .name = "pll3",
        .ops = &std_pll_ops,
        .parent_names = pll_clk_parents,
        .num_parents = ARRAY_SIZE(pll_clk_parents),
};

static struct clk_pll clk_pll1 = {
        .regofs = SIRFSOC_CLKC_PLL1_CFG0,
        .hw = {
                .init = &clk_pll1_init,
        },
};

static struct clk_pll clk_pll2 = {
        .regofs = SIRFSOC_CLKC_PLL2_CFG0,
        .hw = {
```

```
                .init = &clk_pll2_init,
        },
};

static struct clk_pll clk_pll3 = {
        .regofs = SIRFSOC_CLKC_PLL3_CFG0,
        .hw = {
                .init = &clk_pll3_init,
        },
};
void __init sirfsoc_of_clk_init(void)
{
        …

        /* These are always available (RTC and 26MHz OSC)*/
        clk = clk_register_fixed_rate(NULL, "rtc", NULL,
                        CLK_IS_ROOT, 32768);
        BUG_ON(!clk);
        clk = clk_register_fixed_rate(NULL, "osc", NULL,
                        CLK_IS_ROOT, 26000000);
        BUG_ON(!clk);

        clk = clk_register(NULL, &clk_pll1.hw);
        BUG_ON(!clk);
        clk = clk_register(NULL, &clk_pll2.hw);
        BUG_ON(!clk);
        clk = clk_register(NULL, &clk_pll3.hw);
        BUG_ON(!clk);
        …
        clk = clk_register(NULL, &clk_gps.hw);
        BUG_ON(!clk);
        clk_register_clkdev(clk, NULL, "a8010000.gps");
        …
}
```

另外，目前内核更加倡导的方法是透过 Device Tree 来描述电路板上的 clk 树，以及 clk 和设备之间的绑定关系。通常我们需要在 clk 控制器的结点中定义#clock-cells 属性，并且在 clk 驱动中透过 of_clk_add_provider()注册 clk 控制器为一个 clk 树的提供者（provider），并建立系统中各个 clk 和 index 的映射表，如：

```
    Clock                ID
    ------------------------------
    rtc                 0
    osc                  1
    pll1                 2
    pll2                 3
    pll3                 4
    mem                   5
    sys                 6
    security              7
    dsp                  8
    gps                  9
    mf                  10
…
```

在每个具体的设备中，对应的.dts 结点上的 clocks = <&clks index>属性指向其引用的 clk 控制器结点以及使用的 clk 的 index，如：

```
gps@a8010000 {
      compatible = "sirf,prima2-gps";
      reg = <0xa8010000 0x10000>;
      interrupts = <7>;
      clocks = <&clks 9>;
};
```

要特别强调的是，在具体的设备驱动中，一定要透过通用 clk API 来操作所有的 clk，而不要直接透过读写 clk 控制器的寄存器来进行，这些 API 包括：

```
struct clk *clk_get(struct device *dev, const char *id);
struct clk *devm_clk_get(struct device *dev, const char *id);
```

```
int clk_enable(struct clk *clk);
int clk_prepare(struct clk *clk);
void clk_ unprepare(struct clk *clk);
void clk_disable(struct clk *clk);
static inline int clk_prepare_enable(struct clk *clk);
static inline void clk_disable_unprepare(struct clk *clk);
unsigned long clk_get_rate(struct clk *clk);
int clk_set_rate(struct clk *clk, unsigned long rate);
struct clk *clk_get_parent(struct clk *clk);
int clk_set_parent(struct clk *clk, struct clk *parent);
```

值得一提的是，名称中含有 prepare、unprepare 字符串的 API 是内核后来才加入的，过去只有 clk_enable 和 clk_disable。只有 clk_enable 和 clk_disable 带来的问题是，有时候，某些硬件的 enable/disable clk 可能引起睡眠使得 enable/disable 不能在原子上下文进行。加上 prepare 后，把过去的 clk_enable 分解成不可在原子上下文调用的 clk_prepare（该函数可能睡眠）和可以在原子上下文调用的 clk_enable。而 clk_prepare_enable 则同时完成 prepare 和 enable 的工作，当然也只能在可能睡眠的上下文调用该 API。

9. dmaengine 驱动

dmaengine 是一套通用的 DMA 驱动框架，该框架为具体使用 DMA 通道的设备驱动提供了一套统一的 API，而且也定义了具体的 DMA 控制器实现这一套 API 的方法。

对于使用 DMA 引擎的设备驱动而言，发起 DMA 传输的过程变得整洁，如在 sound 子系统的 sound/soc/soc-dmaengine-pcm.c 中，会使用 dmaengine 进行周期性的 DMA 传输，相关的代码如下：

```
static int dmaengine_pcm_prepare_and_submit(struct snd_pcm_substream *substream)
{
        struct dmaengine_pcm_runtime_data *prtd = substream_to_prtd(substream);
        struct dma_chan *chan = prtd->dma_chan;
        struct dma_async_tx_descriptor *desc;
        enum dma_ transfer_ direction direction;
        unsigned long flags = DMA_CTRL_ACK;

        …
        desc = dmaengine_prep_dma_cyclic(chan,
                substream->runtime->dma_addr,
                snd_pcm_lib_buffer_bytes(substream),
                snd_pcm_lib_period_bytes(substream), direction, flags);
                …
        desc->callback = dmaengine_pcm_dma_complete;
        desc->callback_ param = substream;
        prtd->cookie = dmaengine_submit(desc);
}

int snd_dmaengine_pcm_trigger(struct snd_pcm_substream *substream, int cmd)
{
        struct dmaengine_pcm_runtime_data *prtd = substream_to_prtd(substream);
        int ret;
        switch (cmd) {
        case SNDRV_PCM_TRIGGER_START:
                ret = dmaengine_pcm_prepare_and_submit(substream);
                …
                dma_async_issue_pending(prtd->dma_chan);
                break;
        case SNDRV_PCM_TRIGGER_RESUME:
        case SNDRV_PCM_TRIGGER_PAUSE_RELEASE:
                dmaengine_resume(prtd->dma_chan);
                break;
        …
}
```

这个过程可分为三步：

1. 透过 dmaengine_prep_dma_xxx 初始化一个具体的 DMA 传输描述符（本例中为结构体 dma_async_tx_descriptor 的实例 desc）
2. 透过 dmaengine_submit()将该描述符插入 dmaengine 驱动的传输队列
3. 在需要传输的时候透过类似 dma_async_issue_pending()的调用启动对应 DMA 通道上的传输。

也就是不管具体硬件的 DMA 控制器是如何实现的，在软件意义上都抽象为了设置 DMA 描述符、插入 DMA 描述符入传输队列以及启动 DMA 传输的过程。

除了前文用到的 dmaengine_prep_dma_cyclic()用于定义周期性 DMA 传输外，还有一组类似 API 可以定义各种类型的 DMA 描述符，特定硬件的 DMA 驱动的主要工作就是实现封装在内核 dma_device 结构体中的这些个成员函数（定义在 include/linux/dmaengine.h 头文件中）：

```
500 /**
501  * struct dma_device - info on the entity supplying DMA services
…
518  * @device_prep_dma_memcpy: prepares a memcpy operation
519  * @device_prep_dma_xor: prepares a xor operation
520  * @device_prep_dma_xor_val: prepares a xor validation operation
521  * @device_prep_dma_pq: prepares a pq operation
522  * @device_prep_dma_pq_val: prepares a pqzero_sum operation
523  * @device_prep_dma_memset: prepares a memset operation
524  * @device_prep_dma_interrupt: prepares an end of chain interrupt operation
525  * @device_prep_slave_sg: prepares a slave dma operation
526  * @device_prep_dma_cyclic: prepare a cyclic dma operation suitable for audio.
527  *      The function takes a buffer of size buf_len. The callback function will
528  *      be called after period_len bytes have been transferred.
529  * @device_prep_interleaved_dma: Transfer expression in a generic way.
…
*/
```

在底层的 dmaengine 驱动实例中，一般会组织好这个 dma_device 结构体，并透过 dma_async_device_register()注册之。在其各个成员函数中，一般会透过链表来管理 DMA 描述符的运行、free 等队列。

dma_device 的成员函数 device_issue_pending()用于实现 DMA 传输开启的功能，当每次 DMA 传输完成后，驱动中注册的中断服务程序的顶半部或者底半部会调用 DMA 描述符 dma_async_tx_descriptor 中设置的 callback 函数，该 callback 函数来源于使用 DMA 通道的设备驱动。

典型的 dmaengine 驱动可见于 drivers/dma/目录下的 sirf-dma.c、omap-dma.c、pl330.c、ste_dma40.c 等。

10. 总结

移植 Linux 到全新的 SMP SoC 上，需在底层提供定时器节拍、中断控制器、SMP 启动、GPIO、clock、pinctrl 等功能，这些底层的功能被封装好后，其他设备驱动只能调用内核提供的通用 API。这良好地体现了内核的分层设计。即驱动都调用与硬件无关的通用 API，而这些 API 的底层实现则更多的是填充内核规整好的 callback 函数。

Linux 内核社区针对 pinctrl、clock、GPIO、DMA 提供独立的子系统，既给具体的设备驱动提供了统一了 API，进一步提高了设备驱动的跨平台性，又为每个 SoC 和 machine 实现这些底层的 API 定义好了条条框框，从而可以最大程度上避免每个硬件实现过多的冗余代码。