

第 20 章

Linux 芯片级移植及底层驱动

本章导读

本章主要讲解，在一个新的 ARM SoC 上，如何移植 Linux。当然，本章的内容也适合 MIPS、PowerPC 等其他的体系结构。

第 20.1 节先总体上介绍了 Linux 3.x 之后的内核在底层 BSP 上进行了哪些优化。

第 20.2 节讲解了如何提供操作系统的运行节拍。

第 20.3 节讲解了中断控制器驱动，以及它是如何为驱动提供标准接口的。

第 20.4 节讲解多核 SMP 芯片的启动。

第 20.6~20.9 节分别讲解了作为 Linux 运行底层基础设施的 GPIO、pinctrl、时钟和 dmaengine 驱动。

学习本章有助于工程师理解驱动调用的底层 API 的来源，以及直接进行 Linux 的平台移植。

20.1 ARM 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 热插拔以及底层的 GPIO、时钟、pinctrl 和 DMA 硬件的封装等。定时器节拍、中断控制器、SMP 启动和 CPU 热插拔这几部分相对来说没有像早期 GPIO、时钟、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 热插拔则运行运行时挂载或拔除 CPU。这些工作，在 Linux 3.0 之后的内核中，Linux 社区对比逐步进行了良好的层次划分和架构设计。

在 GPIO、时钟、pinctrl 和 DMA 驱动方面，在 Linux 2.6 时代，内核已或多或少有 GPIO、时钟等底层驱动的架构，但是核心层的代码太薄弱，各 SoC 在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独特的实现和 API。

社区必须改变这种局面，于是 Linux 社区在 2011 年后进行了如下工作，这些工作在目前的 Linux 内核中基本准备就绪：

- STEricsson 公司的工程师 Linus Walleij 提供了新的 pinctrl 驱动架构，内核中新增加一个 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 公司的工程师 Mike Turquette 提供了通过时钟框架，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的对应关系，具体的时钟驱动都统一迁移到 drivers/clk 目录中。
- 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 目录中。
- 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基础架构方面的驱动架构统一了，实现方法也统一了。另外，目前 GPIO、时钟、pinmux 等都能良好地进行设备树的映射处理，譬如我们可以方便地在 .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 提供商实现少量的回调函数或宏。

本章主要对上述各个组成部分进行架构上的剖析以及对关键的实现部分的实例分析，以求完整归纳出将 Linux 移植入新 SoC 的主要工作。

20.2 内核节拍驱动

Linux 2.6 的早期（Linux2.6.21 之前）内核是基于节拍设计的，一般 SoC 公司在将 Linux 移植到自己芯片上的时候，会从芯片内部找一个定时器，并将该定时器配置为赫兹的频率，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 Linux 2.6.17 中 arch/arm/mach-s3c2410/time.c 的做法类似于代码清单 20.1 所示。

代码清单 20.1 早期内核的节拍驱动

```

1  /*
2   * IRQ handler for the timer
3   */
4  static irqreturn_t
5  s3c2410_timer_interrupt(int irq, void *dev_id, struct pt_regs *regs)
6  {

```

```

7     write_seqlock(&xtime_lock);
8     timer_tick(regs);
9     write_sequnlock(&xtime_lock);
10    return IRQ_HANDLED;
11 }
12
13 static struct irqaction s3c2410_timer_irq = {
14     .name        = "S3C2410 Timer Tick",
15     .flags       = SA_INTERRUPT | SA_TIMER,
16     .handler     = s3c2410_timer_interrupt,
17 };
18 static void __init s3c2410_timer_init (void)
19 {
20     s3c2410_timer_setup();
21     setup_irq(IRQ_TIMER4, &s3c2410_timer_irq);
22 }

```

代码清单 20.1 将硬件的 TIMER4 定时器配置为周期触发中断，每个中断到来就会自动调用内核函数 timer_tick()。

当前 Linux 多采用无节拍方案，并支持高精度定时器，内核的配置一般会使能 NO_HZ (即无节拍，或者说动态节拍) 和 HIGH_RES_TIMERS。要强调的是无节拍并不是说系统中没有时钟节拍，而是说这个节拍不再像以前那样周期性地产生。无节拍意味着，根据系统的运行情况，以事件驱动的方式动态决定下一个节拍在何时发生。如果画一个时间轴，周期节拍的系统节拍中断发生的时序如图 20.1 所示：



图 20.1 周期节拍的系统节拍中断发生的时序

而 NO_HZ 的 Linux 的运行节拍如图 20.2 所示，看起来则是：两次定时器中断发生的时间间隔可长可短：



图 20.2 NO_HZ 的运行节拍

在当前的 Linux 系统中，SoC 底层的定时器被实现为一个 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。而在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 SoC 的底层节拍定时器驱动形如代码清单 20.2 所示。

代码清单 20.2 新内核基于 clocksource 和 clock_event 的节拍驱动

```

1 static irqreturn_t xxx_timer_interrupt(int irq, void *dev_id)
2 {
3     struct clock_event_device *ce = dev_id;
4     ...
5     ce->event_handler(ce);
6
7     return IRQ_HANDLED;
8 }
9
10 /* read 64-bit timer counter */
11 static cycle_t xxx_timer_read(struct clocksource *cs)
12 {
13     u64 cycles;
14
15     /* read the 64-bit timer counter */
16     cycles = readl_relaxed(xxx_timer_base + LATCHED_HI);
17     cycles = (cycles << 32) | readl_relaxed(xxx_timer_base + LATCHED_LO);
18
19     return cycles;
20 }
21
22 static int xxx_timer_set_next_event(unsigned long delta,
23         struct clock_event_device *ce)
24 {
25     unsigned long now, next;
26     now = readl_relaxed(xxx_timer_base + LATCHED_LO);
27     next = now + delta;
28     writel_relaxed(next, xxx_timer_base + SIRFSOC_TIMER_MATCH_0);
29     ...
30 }
31
32 static void xxx_timer_set_mode(enum clock_event_mode mode,
33         struct clock_event_device *ce)
34 {
35     switch (mode) {
36     case CLOCK_EVT_MODE_PERIODIC:
37         ...
38     case CLOCK_EVT_MODE_ONESHOT:
39         ...
40     case CLOCK_EVT_MODE_SHUTDOWN:
41         ...
42     case CLOCK_EVT_MODE_UNUSED:
43     case CLOCK_EVT_MODE_RESUME:
44         break;
45     }
46 }
47 static struct clock_event_device xxx_clockevent = {
48     .name = "xxx_clockevent",
49     .rating = 200,

```

```

50     .features = CLOCK_EVT_FEAT_ONESHOT,
51     .set_mode = xxx_timer_set_mode,
52     .set_next_event = xxx_timer_set_next_event,
53 };
54
55 static struct clocksource xxx_clocksource = {
56     .name = "xxx_clocksource",
57     .rating = 200,
58     .mask = CLOCKSOURCE_MASK(64),
59     .flags = CLOCK_SOURCE_IS_CONTINUOUS,
60     .read = xxx_timer_read,
61     .suspend = xxx_clocksource_suspend,
62     .resume = xxx_clocksource_resume,
63 };
64
65 static struct irqaction xxx_timer_irq = {
66     .name = "xxx_tick",
67     .flags = IRQF_TIMER,
68     .irq = 0,
69     .handler = xxx_timer_interrupt,
70     .dev_id = &xxx_clockevent,
71 };
72
73 static void __init xxx_clockevent_init(void)
74 {
75     clockevents_calc_mult_shift(&xxx_clockevent, CLOCK_TICK_RATE, 60);
76
77     xxx_clockevent.max_delta_ns =
78         clockevent_delta2ns(-2, &xxx_clockevent);
79     xxx_clockevent.min_delta_ns =
80         clockevent_delta2ns(2, &xxx_clockevent);
81
82     xxx_clockevent.cpumask = cpumask_of(0);
83     clockevents_register_device(&xxx_clockevent);
84 }
85
86 /* initialize the kernel jiffy timer source */
87 static void __init xxx_timer_init(void)
88 {
89     ...
90     BUG_ON(clocksource_register_hz(&xxx_clocksource, CLOCK_TICK_RATE));
91     BUG_ON(setup_irq(xxx_timer_irq.irq, &xxx_timer_irq));
92     xxx_clockevent_init();
93 }
94 struct sys_timer xxx_timer = {
95     .init = xxx_timer_init,
96 };

```

在上述代码中，我们特别关注如下的函数：

1. `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，它的含义是下一次节拍中断产生的硬件定时器中计数器的值相对于当前计数器的差值。我们在该函数中将硬件定时器设置为在“当前计数器计数值 +`delta`”的时刻产生下一次节拍中断。`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`。

2. `clocksource` 的 `read` 成员函数 `xxx_timer_read()`

该函数可读取出从开机到当前时刻定时器计数器已经走过的值，无论有没有设置当计数器达到某值时产生中断，硬件的计数总是在进行的（我们要理解，计数总是在进行，而计数到某值后要产生中断则需要软件设置）。因此，该函数给 Linux 系统提供了一个底层的准确的参考时间。

3. 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。

4. `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恢复、关闭等功能，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，如果内核在编译的时候未选择 NO_HZ，该底层的定时器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 1μs），应用程序通过 `nanosleep()` API 睡眠 100μs，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 `xxx_timer_set_next_event()` 去设置硬件让其在 100μs 后产生中断。100μs 后，中断产生，`xxx_timer_interrupt()` 被调用，`event_handler()` 会间接唤醒睡眠的进程并导致 `nanosleep()` 函数返回，从而让用户进程继续。

这里要特别强调的是，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地设置自己时钟中断发生的时刻。看一下我们所运行的 ARM vexpress 的中断（GIC 29 twd）即知：

```
# cat /proc/interrupts
          CPU0      CPU1      CPU2      CPU3
 29: 1548 1511 1501 1484   GIC 29 twd
 34:    7     0     0     0   GIC 34 timer
 36:    0     0     0     0   GIC 36 rtc-p1031
 37: 162   21     2    27   GIC 37 uart-p1011
 41:   88   105   149   121   GIC 41 mmc1-p118x (cmd)
 42: 5449 5443 5450 5863   GIC 42 mmc1-p118x (pio)
 44:    0     8     1     0   GIC 44 kmi-p1050
 45:    0   100     0     0   GIC 45 kmi-p1050
```

	0	0	0	0	GIC	47	eth0
IPI0:	0	1	1	1	CPU wakeup	interrupts	
IPI1:	0	0	0	0	Timer broadcast	interrupts	
IPI2:	454	266	436	642	Rescheduling	interrupts	
IPI3:	0	1	1	1	Function call	interrupts	
IPI4:	0	0	0	0	Single function call	interrupts	
IPI5:	0	0	0	0	CPU stop	interrupts	
IPI6:	0	0	0	0	IRQ work	interrupts	
IPI7:	0	0	0	0	completion	interrupts	
Err:	0						

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

```
enum ipi_msg_type {
    IPI_WAKEUP,
    IPI_TIMER,
    IPI_RESCHEDULE,
    IPI_CALL_FUNC,
    IPI_CALL_FUNC_SINGLE,
    IPI_CPU_STOP,
};
```

20.3 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 来完成中断申请、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 ARM v6 以上的体系结构而言，是直接调用 CPSID/CPSIE 指令进行，而对于 ARM v6 以前的体系结构，则是通过 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 中，如代码清单 20.3 所示。

代码清单 20.3 ARM Linux local_irq_disable()/enable() 底层实现

```
1 #if __LINUX_ARM_ARCH__ >= 6
2
3 static inline unsigned long arch_local_irq_save(void)
4 {
5     unsigned long flags;
6
7     asm volatile(
8         "mrs %0, cpsr" @ arch_local_irq_save\n"
9     );
10}
```

```

9          "      cpsid    i"
10         : "=r" (flags) : : "memory", "cc");
11     return flags;
12 }
13
14 static inline void arch_local_irq_enable(void)
15 {
16     asm volatile(
17         "      cpsie i           @ arch_local_irq_enable"
18         :
19         :
20         : "memory", "cc");
21 }
22
23 static inline void arch_local_irq_disable(void)
24 {
25     asm volatile(
26         "      cpsid i           @ arch_local_irq_disable"
27         :
28         :
29         : "memory", "cc");
30 }
31 #else
32
33 /*
34  * Save the current interrupt enable state & disable IRQs
35  */
36 static inline unsigned long arch_local_irq_save(void)
37 {
38     unsigned long flags, temp;
39
40     asm volatile(
41         "      mrs    %0, cpsr      @ arch_local_irq_save\n"
42         "      orr    %1, %0, #128\n"
43         "      msr    cpsr_c, %1"
44         : "=r" (flags), "=r" (temp)
45         :
46         : "memory", "cc");
47     return flags;
48 }
49
50 /*
51  * Enable IRQs
52  */
53 static inline void arch_local_irq_enable(void)
54 {
55     unsigned long temp;
56     asm volatile(
57         "      mrs    %0, cpsr      @ arch_local_irq_enable\n"
58         "      bic    %0, %0, #128\n"

```

```

59         "        msr      cpsr_c, %0"
60         : "=r" (temp)
61         :
62         : "memory", "cc");
63 }
64
65 /*
66 * Disable IRQs
67 */
68 static inline void arch_local_irq_disable(void)
69 {
70     unsigned long temp;
71     asm volatile(
72         "        mrs      %0, cpsr          @ arch_local_irq_disable\n"
73         "        orr      %0, %0, #128\n"
74         "        msr      cpsr_c, %0"
75         : "=r" (temp)
76         :
77         : "memory", "cc");
78 }
79 #endif

```

与 local_irq_disable() 和 local_irq_enable() 不同， disable_irq()、enable_irq() 针对的则是中断控制器，因此它们适用的对象是某个中断。 disable_irq() 的字面意思是暂时屏蔽掉某中断（其实在内核的实现层面上做了延后屏蔽），直到 enable_irq() 后再执行 ISR。实际上，屏蔽中断可以发生在外设、中断控制器、CPU 三个位置，如图 20.3 所示。对于外设端，是从源头上就不产生中断信号给中断控制器，由于它高度依赖于外设本身，所以 Linux 不提供标准的 API 而是由外设的驱动直接读写自身的寄存器。

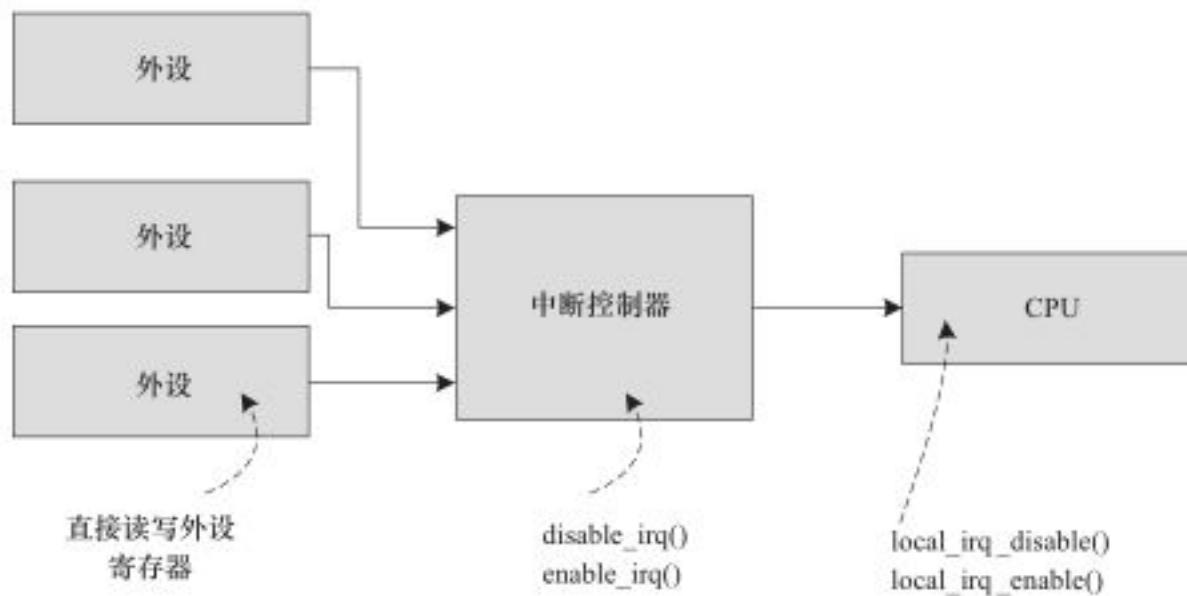


图 20.3 屏蔽中断的 3 个不同位置

在内核中，通过 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 中，如代码清单 20.4 所示。

代码清单 20.4 irq_chip 结构体

```

1 struct irq_chip {
2         const char      *name;
3         unsigned int    (*irq_startup)(struct irq_data *data);
4         void           (*irq_shutdown)(struct irq_data *data);
5         void           (*irq_enable)(struct irq_data *data);
6         void           (*irq_disable)(struct irq_data *data);
7
8         void           (*irq_ack)(struct irq_data *data);
9         void           (*irq_mask)(struct irq_data *data);
10        void          (*irq_mask_ack)(struct irq_data *data);
11        void          (*irq_unmask)(struct irq_data *data);
12        void          (*irq_eoi)(struct irq_data *data);
13
14        int           (*irq_set_affinity)(struct irq_data *data, const struct
15                                         cpumask *dest, bool force);
16        int           (*irq_retrigger)(struct irq_data *data);
17        int           (*irq_set_type)(struct irq_data *data, unsigned int
18                                     flow_type);
18    };

```

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/sirf/pinctrl-sirf.c 驱动中的下面代码部分：

```

static struct irq_chip sirfsoc_irq_chip = {
    .name = "sirf-gpio-irq",
    .irq_ack = sirfsoc_gpio_irq_ack,
    .irq_mask = sirfsoc_gpio_irq_mask,
    .irq_unmask = sirfsoc_gpio_irq_unmask,
    .irq_set_type = sirfsoc_gpio_irq_type,
};

```

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。至于到 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 irq_unmask() 成员函数，这点从 kernel/irq/chip.c 中可以看出：

```

void irq_enable(struct irq_desc *desc)
{
    irq_state_clr_disabled(desc);
    if (desc->irq_data.chip->irq_enable)
        desc->irq_data.chip->irq_enable(&desc->irq_data);
}

```

```

    else
        desc->irq_data(chip->irq_unmask(&desc->irq_data);
    irq_state_clr_masked(desc);
}

```

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），其关系如图 20.4 所示。

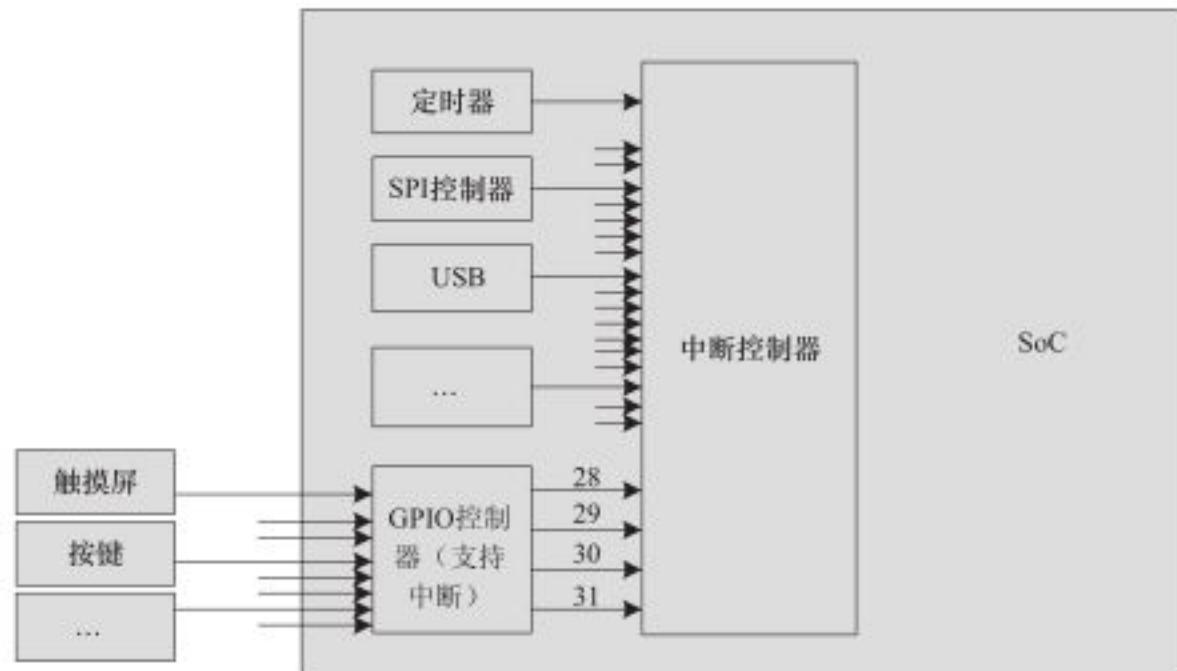


图 20.4 SoC 中断控制器的典型分布

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与 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与 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，无须看到这种 2 级映射关系。如果某设备想申请与 `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图 20.5 所示。

要特别注意的是，上述图 20.4 和 20.5 中所涉及的中断号的数值，无论是 base 还是具体某个 GPIO 对应的中断号是多少，都不一定是如图 20.4 和图 20.5 所描述的简单线性映射。Linux 使用 IRQ Domain 来描述一个中断控制器所管理的中断源。换句话说，每个中断控制器都有自己的 Domain。我们可以将 IRQ Domain 看作是 IRQ 控制器的软件抽象。在添加 IRQ Domain 的时候，内核中存在的映射方法有：`irq_domain_add_legacy()`、`irq_domain_add_linear()`、`irq_domain_add_tree()` 等。

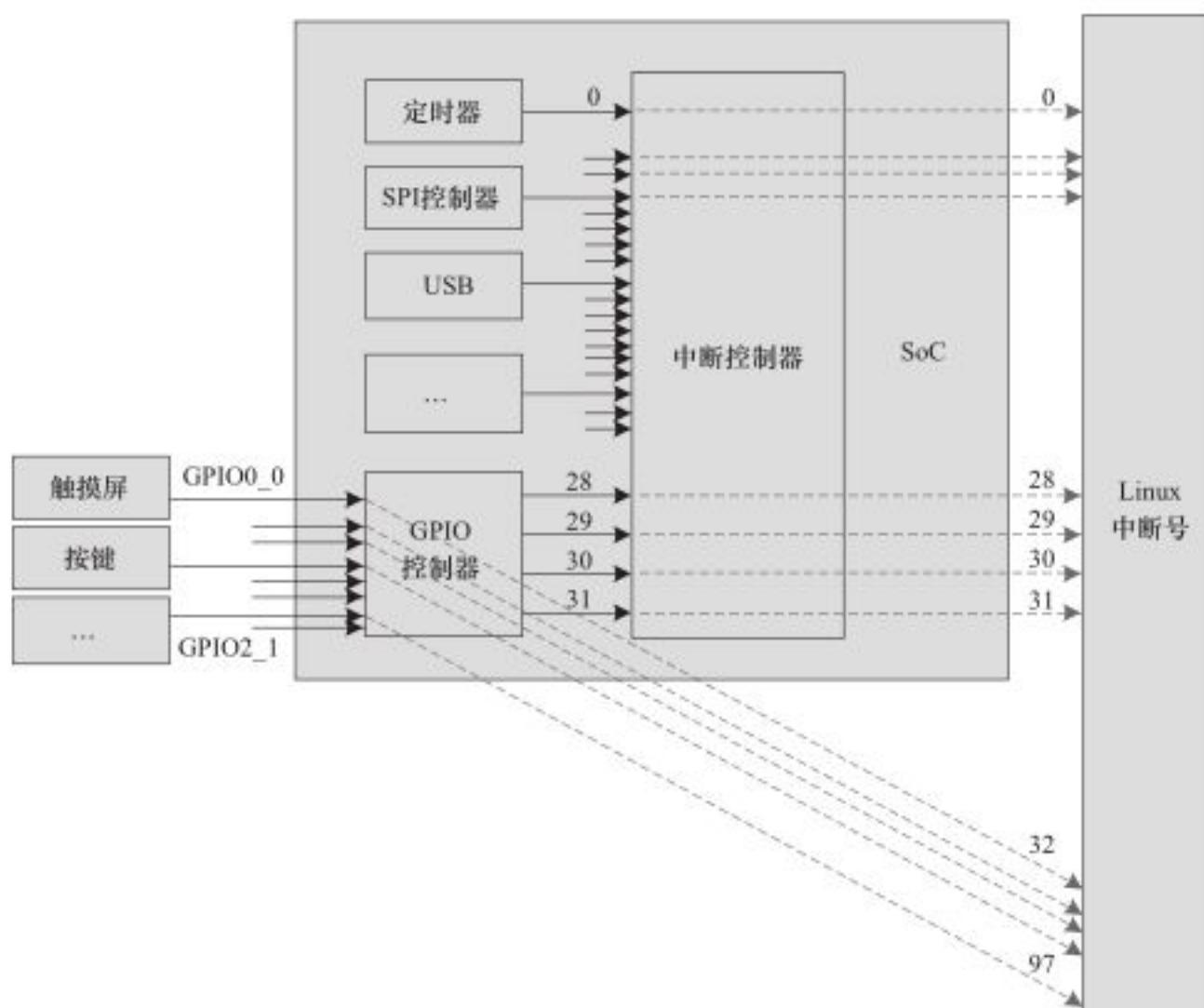


图 20.5 中断级联与映射

`irq_domain_add_legacy()` 实际上是一种过时的方法，它一般是由 IRQ 控制器驱动直接指定中断源硬件意义上的偏移（一般称为 hwirq）和 Linux 逻辑上的中断号的映射关系。类似图 20.5 的指定映射可以被这种方法弄出来。`irq_domain_add_linear()` 则在中断源和 `irq_desc` 之间建立线性映射，内核针对这个 IRQ Domain 维护了一个 hwirq 和 Linux 逻辑 IRQ 之间关系的一个表，这个时候我们其实也完全不关心逻辑中断号了；`irq_domain_add_tree()` 则更加灵活，逻辑中断号和 hwirq 之间的映射关系是用一棵 radix 树来描述的，我们需要通过查找的方法来寻找 hwirq 和 Linux 逻辑 IRQ 之间的关系，一般适合某中断控制器支持非常多中断源的情况。

实际上，在当前的内核中，中断号更多的是一个逻辑概念，具体数值是多少不是很关键。人们更多的是关心在设备树中设置正确的 `interrupt_parent` 和相对该 `interrupt_parent` 的偏移。

以 `drivers/pinctrl/sirf/pinctrl-sirf.c` 的 `irq_chip` 部分为例，在 `sirfsoc_gpio_probe()` 函数中，每组 GPIO 的中断都通过 `gpiochip_set_chained_irqchip()` 级联到上一级中断控制器的中断。

代码清单 20.5 二级 GPIO 中断级联到一级中断控制器

```

1 static int sirfsoc_gpio_probe(struct device_node *np)
2 {

```

```

3 ...
4 for (i = 0; i < SIRFSOC_GPIO_NO_OF_BANKS; i++) {
5     bank = &sgpio->sgpio_bank[i];
6     spin_lock_init(&bank->lock);
7     bank->parent_irq = platform_get_irq(pdev, i);
8     if (bank->parent_irq < 0) {
9         err = bank->parent_irq;
10        goto out_banks;
11    }
12
13    gpiochip_set_chained_irqchip(&sgpio->chip.gc,
14        &sirfsoc_irq_chip,
15        bank->parent_irq,
16        sirfsoc_gpio_handle_irq);
17 }
18
19 ...
20 }

```

对于 SIRFSOC_GPIO_NO_OF_BANKS 这多组 GPIO 进行循环，上述代码中第 15 行的 bank->parent_irq 是与这一组 GPIO 对应的“上级”中断号，sirfsoc_gpio_handle_irq() 则是与 bank->parent_irq 对应的“上级”中断服务程序。而 sirfsoc_gpio_handle_irq() 这个“上级”函数最终还是要调用 GPIO 这一级别的中断服务程序。

在 sirfsoc_gpio_handle_irq() 函数的入口处调用 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，并通过 generic_handle_irq() 调用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，如代码清单 20.6 所示。

代码清单 20.6 “上级”中断服务程序派生到下级

```

1 static void sirfsoc_gpio_handle_irq(unsigned int irq, struct irq_desc *desc)
2 {
3 ...
4     chained_irq_enter(chip, desc);
5
6     while (status) {
7         ctrl = readl(sgpio->chip.regs + SIRFSOC_GPIO_CTRL(bank->id, idx));
8
9         /*
10          * Here we must check whether the corresponding GPIO's interrupt
11          * has been enabled, otherwise just skip it
12          */
13         if ((status & 0x1) && (ctrl & SIRFSOC_GPIO_CTL_INTR_EN_MASK)) {
14             generic_handle_irq(irq_find_mapping(gc->irqdomain, idx +
15                                 bank->id * SIRFSOC_GPIO_BANK_SIZE));
16         }

```

```

17
18     idx++;
19     status = status >> 1;
20 }
21
22 chained_irq_exit(chip, desc);
23 }

```

下面用一个实例来呈现这个过程，假设 GPIO0_0~31 对应上级中断号 28，而外设 A 使用了 GPIO0_5（即第 0 组 GPIO 的第 5 个），并假定外设 A 的中断号为 37，即 32+5，中断服务程序为 deva_isr()。那么，当 GPIO0_5 中断发生的时候，内核的调用顺序是：sirfsoc_gpio_handle_irq()>generic_handle_irq()>deva_isr()。如果硬件的中断系统有更深的层次，这种软件上的中断服务程序级联实际上可以有更深的级别。

在上述实例中，GPIO0_0~31 的 interrupt_parrent 实际是上级中断控制器，而外设 A 的 interrupt_parrent 就是 GPIO0，这些都会在设备树中进行呈现。

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，在这种情况下，我们无须实现 1 个完整的 irq_chip 驱动，而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 drivers/irqchip/irq-sirfsoc.c 中，用于注册 CSR SiRFprimaII 内部中断控制器的代码（见代码清单 20.7）。

代码清单 20.7 使用 generic 的 irq_chip 框架

```

1 static __init void
2 sirfsoc_alloc_gc(void __iomem *base, unsigned int irq_start, unsigned int num)
3 {
4     struct irq_chip_generic *gc;
5     struct irq_chip_type *ct;
6     int ret;
7     unsigned int clr = IRQ_NOREQUEST | IRQ_NOPROBE | IRQ_NOAUTOEN;
8     unsigned int set = IRQ_LEVEL;
9
10    ret = irq_alloc_domain_generic_chips(sirfsoc_irqdomain, num, 1, "irq_sirfsoc",
11                                         handle_level_irq, clr, set, IRQ_GC_INIT_MASK_CACHE);
12
13    gc = irq_get_domain_generic_chip(sirfsoc_irqdomain, irq_start);
14    gc->reg_base = base;
15    ct = gc->chip_types;
16    ct->chip.irq_mask = irq_gc_mask_clr_bit;
17    ct->chip.irq_unmask = irq_gc_mask_set_bit;
18    ct->regs.mask = SIRFSOC_INT_RISC_MASK0;
19 }

```

irq_chip 驱动的入口声明方法形如：

```
IRQCHIP_DECLARE(sirfsoc_intc, "sirf,prima2-intc", sirfsoc_irq_init);
```

sirf,prima2-intc 是设备树中中断控制器的 compatible 字段，sirfsoc_irq_init 是匹配这个 compatible 字段后运行的初始化函数。

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，只需要在设备树中添加相关的节点。

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

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

打开 drivers/irqchip/irq-gic.c，发现 GIC 驱动的入口声明如下：

```
IRQCHIP_DECLARE(gic_400, "arm,gic-400", gic_of_init);
IRQCHIP_DECLARE(cortex_a15_gic, "arm,cortex-a15-gic", gic_of_init);
IRQCHIP_DECLARE(cortex_a9_gic, "arm,cortex-a9-gic", gic_of_init);
IRQCHIP_DECLARE(cortex_a7_gic, "arm,cortex-a7-gic", gic_of_init);
IRQCHIP_DECLARE(msm_8660_qgic, "qcom,msm-8660-qgic", gic_of_init);
IRQCHIP_DECLARE(msm_qgic2, "qcom,msm-qgic2", gic_of_init);
```

这说明 drivers/irqchip/irq-gic.c 这个驱动可以兼容 arm,gic-400、arm,cortex-a15-gic、arm,cortex-a7-gic 等，但是初始化函数都是统一的 gic_of_init。

20.4 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，在 Bootrom 代码中，每个 CPU 都会识别自身 ID，如果 ID 是 0，则引导 Bootloader 和 Linux 内核执行，如果 ID 不是 0，则 Bootrom 一般在上电时将自身置于 WFI 或者 WFE 状态，并等待 CPU0 给其发 CPU 核间中断或事件（一般通过 SEV 指令）以唤醒它。一个典型的多核 Linux 启动过程如图 20.6 所示。

被 CPU0 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 中：

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

同理，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。

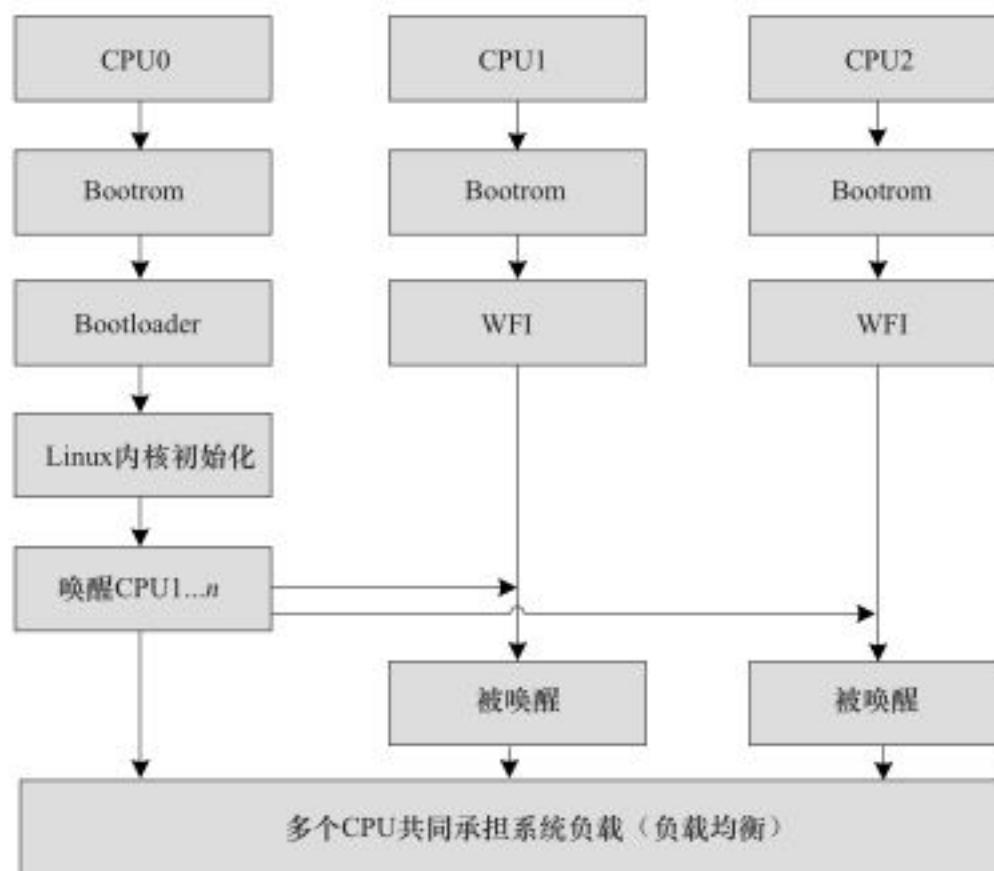


图 20.6 一个典型的多核 Linux 启动过程

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，对于 ARM 而言，它定义于 `arch/arm/include/asm/smp.h` 中。该结构体的成员函数如代码清单 20.8 所示。

代码清单 20.8 `smp_operations` 结构体

```

1 struct smp_operations {
2 #ifdef CONFIG_SMP
3     /*
4      * Setup the set of possible CPUs (via set_cpu_possible)
5      */
6     void (*smp_init_cpus)(void);
7     /*
8      * Initialize cpu_possible map, and enable coherency
9      */
10    void (*smp_prepare_cpus)(unsigned int max_cpus);
11
12    /*
13     * Perform platform specific initialisation of the specified CPU.
14     */
15    void (*smp_secondary_init)(unsigned int cpu);
16    /*
17     * Boot a secondary CPU, and assign it the specified idle task.
18     * This also gives us the initial stack to use for this CPU.
19     */
20    int (*smp_boot_secondary)(unsigned int cpu, struct task_struct *idle);
21 #ifdef CONFIG_HOTPLUG_CPU
22     int (*cpu_kill)(unsigned int cpu);
23 
```

```

23         void (*cpu_die)(unsigned int cpu);
24         int  (*cpu_disable)(unsigned int cpu);
25 #endif
26 #endif
27 };

```

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

```

DT_MACHINE_START(VEXPRESS_DT, "ARM-Versatile Express")
    .dt_compatible = v2m_dt_match,
    .smp           = smp_ops(vexpress_smp_ops),
    .map_io        = v2m_dt_map_io,
    ...
MACHINE_END

```

通过 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() 调用的 ct_ca9x4_init_cpu_map() 会探测 SoC 内 CPU 核的个数，并通过 set_cpu_possible() 设置这些 CPU 可见。

而 smp_operations 的成员函数 smp_prepare_cpus(), 即 vexpress_smp_prepare_cpus() 则会通过 v2m_flags_set(virt_to_phys(versatile_secondary_startup)) 设置其他 CPU 的启动地址为 versatile_secondary_startup，如代码清单 20.9 所示。

代码清单 20.9 在 smp_prepare_cpus() 中设置 CPU1...n 的启动地址

```

1 static void __init vexpress_smp_prepare_cpus(unsigned int max_cpus)
2 {
3     ...
4
5     /*
6      * Write the address of secondary startup into the
7      * system-wide flags register. The boot monitor waits
8      * until it receives a soft interrupt, and then the
9      * secondary CPU branches to this address.
10     */
11     v2m_flags_set(virt_to_phys(versatile_secondary_startup));
12 }

```

注意这部分的具体实现方法是与 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：

```

void __init v2m_flags_set(u32 data)
{
    writel(~0, v2m_sysreg_base + V2M_SYS_FLAGSCLR);
    writel(data, v2m_sysreg_base + V2M_SYS_FLAGSSET);
}

```

即填充 v2m_sysreg_base+V2M_SYS_FLAGSCLR 标记清除寄存器为 0xFFFFFFFF，将 CPU1...n 初始启动执行的指令地址填入 v2m_sysreg_base+V2M_SYS_FLAGSSET 寄存器。这

两个地址由芯片实现时内部的 Bootrom 程序设定的。填入 CPU $1\dots n$ 的起始地址都通过 virt_to_phys() 转化为物理地址，因为此时 CPU $1\dots n$ 的 MMU 尚未开启。

比较关键的是 smp_operations 的成员函数 smp_boot_secondary()，对于本例而言为 versatile_boot_secondary()，它完成 CPU 的最终唤醒工作，如代码清单 20.10 所示。

代码清单 20.10 CPU0 通过中断唤醒其他 CPU

```

1 static void write_pen_release(int val)
2 {
3     pen_release = val;
4     smp_wmb();
5     sync_cache_w(&pen_release);
6 }
7
8 int versatile_boot_secondary(unsigned int cpu, struct task_struct *idle)
9 {
10    unsigned long timeout;
11    ...
12    /*
13     * This is really belt and braces; we hold unintended secondary
14     * CPUs in the holding pen until we're ready for them. However,
15     * since we haven't sent them a soft interrupt, they shouldn't
16     * be there.
17     */
18    write_pen_release(cpu_logical_map(cpu));
19
20    /*
21     * Send the secondary CPU a soft interrupt, thereby causing
22     * the boot monitor to read the system wide flags register,
23     * and branch to the address found there.
24     */
25    arch_send_wakeup_ipi_mask(cpumask_of(cpu));
26
27    timeout = jiffies + (1 * HZ);
28    while (time_before(jiffies, timeout)) {
29        smp_rmb();
30        if (pen_release == -1)
31            break;
32
33        udelay(10);
34    }
35    ...
36    return pen_release != -1 ? -ENOSYS : 0;
37 }
```

上述代码第 18 行调用的 write_pen_release() 会将 pen_release 变量设置为要唤醒的 CPU 核的 CPU 号 cpu_logical_map(cpu)，而后通过 arch_send_wakeup_ipi_mask() 给要唤醒的 CPU 发 IPI 中断，这个时候，被唤醒的 CPU 会退出 WFI 状态并从前面 smp_operations 中的 smp_prepare_cpus() 成员函数，即 vexpress_smp_prepare_cpus() 里通过 v2m_flags_set() 设置的起始地址 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（见第 28~34 行）中跳出。

versatile_secondary_startup 实现于 arch/arm/plat-versatile/headsmp.S 中，是一段汇编，如代码清单 20.11 所示。

代码清单 20.11 被唤醒 CPU 的执行入口

```

1 ENTRY(versatile_secondary_startup)
2     mrc    p15, 0, r0, c0, c0, 5
3     and    r0, r0, #15
4     adr    r4, lf
5     ldmia  r4, [r5, r6]
6     sub    r4, r4, r5
7     add    r6, r6, r4
8 pen:   ldr    r7, [r6]
9      cmp    r7, r0
10     bne   pen
11
12     /*
13      * we've been released from the holding pen: secondary_stack
14      * should now contain the SVC stack for this core
15      */
16     b     secondary_startup
17
18     .align
19 l:    .long .
20     .long pen_release
21 ENDPROC(versatile_secondary_startup)

```

上述代码第 8~10 行的循环是等待 pen_release 变量成为 CPU0 设置的 cpu_logical_map(cpu)，一般直接就成立了。第 16 行则调用内核通用的 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 smp_operations 的 smp_secondary_init() 成员函数，对于本例为 versatile_secondary_init()，如代码清单 20.12 所示。

代码清单 20.12 被唤醒的 CPU 恢复 pen_release()

```

1 void versatile_secondary_init(unsigned int cpu)
2 {
3     /*
4      * let the primary processor know we're out of the
5      * pen, then head off into the C entry point
6      */
7     write_pen_release(-1);
8
9     /*
10      * Synchronise with the boot thread.
11      */
12     spin_lock(&boot_lock);
13     spin_unlock(&boot_lock);
14 }

```

上述代码第 7 行会将 pen_release 写为 -1，于是 CPU0 还在执行的代码清单 20.10 里

`versatile_boot_secondary()` 函数中的如下循环就退出了：

```
while (time_before(jiffies, timeout)) {
    smp_rmb();
    if (pen_release == -1)
        break;

    udelay(10);
}
```

这样 CPU0 就知道目标 CPU 已经被正确地唤醒，此后 CPU0 和新唤醒的其他 CPU 各自运行。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。

图 20.7 总结性地描述了前文提到的 `vexpress_smp_prepare_cpus()`、`versatile_boot_secondary()`、`write_pen_release()`、`versatile_secondary_startup()`、`versatile_secondary_init()` 这些函数的执行顺序。

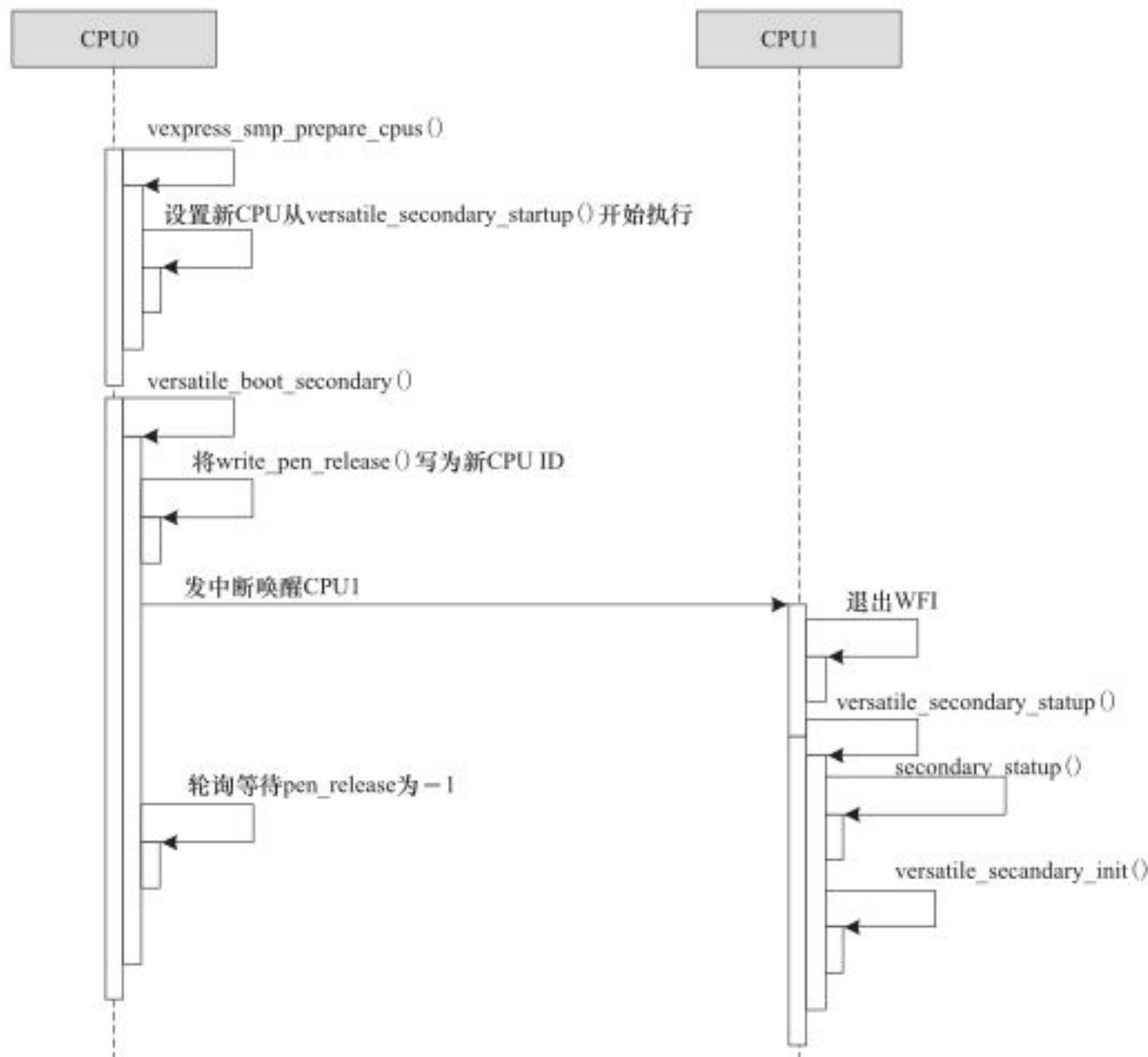


图 20.7 CPU0 唤醒其他 CPU 过程

CPU热插拔的实现也是与芯片相关的，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中，如代码清单20.13所示。

代码清单20.13 smp_operations的cpu_die()成员函数案例

```

1 void __ref vexpress_cpu_die(unsigned int cpu)
2 {
3     int spurious = 0;
4
5     /*
6      * we're ready for shutdown now, so do it
7      */
8     cpu_enter_lowpower();
9     platform_do_lowpower(cpu, &spurious);
10
11    /*
12     * bring this CPU back into the world of cache
13     * coherency, and then restore interrupts
14     */
15    cpu_leave_lowpower();
16
17    if (spurious)
18        pr_warn("CPU%u: %u spurious wakeup calls\n", cpu, spurious);
19 }
20 static inline void platform_do_lowpower(unsigned int cpu, int *spurious)
21 {
22     /*
23      * there is no power-control hardware on this platform, so all
24      * we can do is put the core into WFI; this is safe as the calling
25      * code will have already disabled interrupts
26      */
27     for (;;) {
28         wfi();
29
30         if (pen_release == cpu_logical_map(cpu)) {
31             /*
32              * OK, proper wakeup, we're done
33              */
34             break;
35         }
36
37         /*
38          * Getting here, means that we have come out of WFI without
39          * having been woken up - this shouldn't happen
40          *
41          * Just note it happening - when we're woken, we can report
42          * its occurrence.

```

```

43          */
44          (*spurious)++;
45      }
46 }

```

CPU n 睡眠于 wfi(), 之后再次在线的时候, 又会因为 CPU0 给它发出的 IPI 而从 wfi() 函数返回继续执行, 醒来时 CPU n 也判断 “pen_release==cpu_logical_map(cpu)” 是否成立, 以确定该次醒来确实是由 CPU0 唤醒的一次正常醒来。

20.5 DEBUG_LL 和 EARLY_PRINTK 的设置

在 Linux 启动的早期, 控制台驱动还没有投入运行。当我们把 Linux 移植到一个新的 SoC 时, 工程师一般非常想在刚开始就可以执行 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, 在 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, 相关的代码实现于 arch/arm/include/debug/sirf.S 中, 如代码清单 20.14 所示。

代码清单 20.14 DEBUG_LL 端口的驱动

```

1 .macro addruart, rp, rv, tmp
2 ldr    \rp, =SIRFSOC_UART1_PA_BASE           @ physical
3 ldr    \rv, =SIRFSOC_UART1_VA_BASE           @ virtual
4 .endm
5
6 .macro senduart,rd,rx
7 str    \rd, [\rx, #SIRFSOC_UART_TXFIFO_DATA]
8 .endm
9
10 .macro busyuart,rd,rx
11 .endm
12
13 .macro waituart,rd,rx
14 1001: ldr    \rd, [\rx, #SIRFSOC_UART_TXFIFO_STATUS]
15 tst    \rd, #SIRFSOC_UART1_TXFIFO_EMPTY
16 beq    1001b
17 .endm

```

这些代码没有复杂的框架和中断的支持, 只是单纯地往 UART 的 TX FIFO 寄存器写要发送的数据。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。

而对于本书与 vexpress QEMU 对应的实验平台而言, 相应的驱动则位于 arch/arm/

include/debug/pl01x.S 中，同样是实现了类似的宏。

在配置内核的时候，要进行正确的配置。譬如，对于 vexpress 的实验板子，我们选择的就是“Kernel low-level debugging port(Use PL011 UART0 at 0x10009000(V2P-CA9 core tile))”，对应的 UART 类型为 PL01X，如图 20.8 所示。

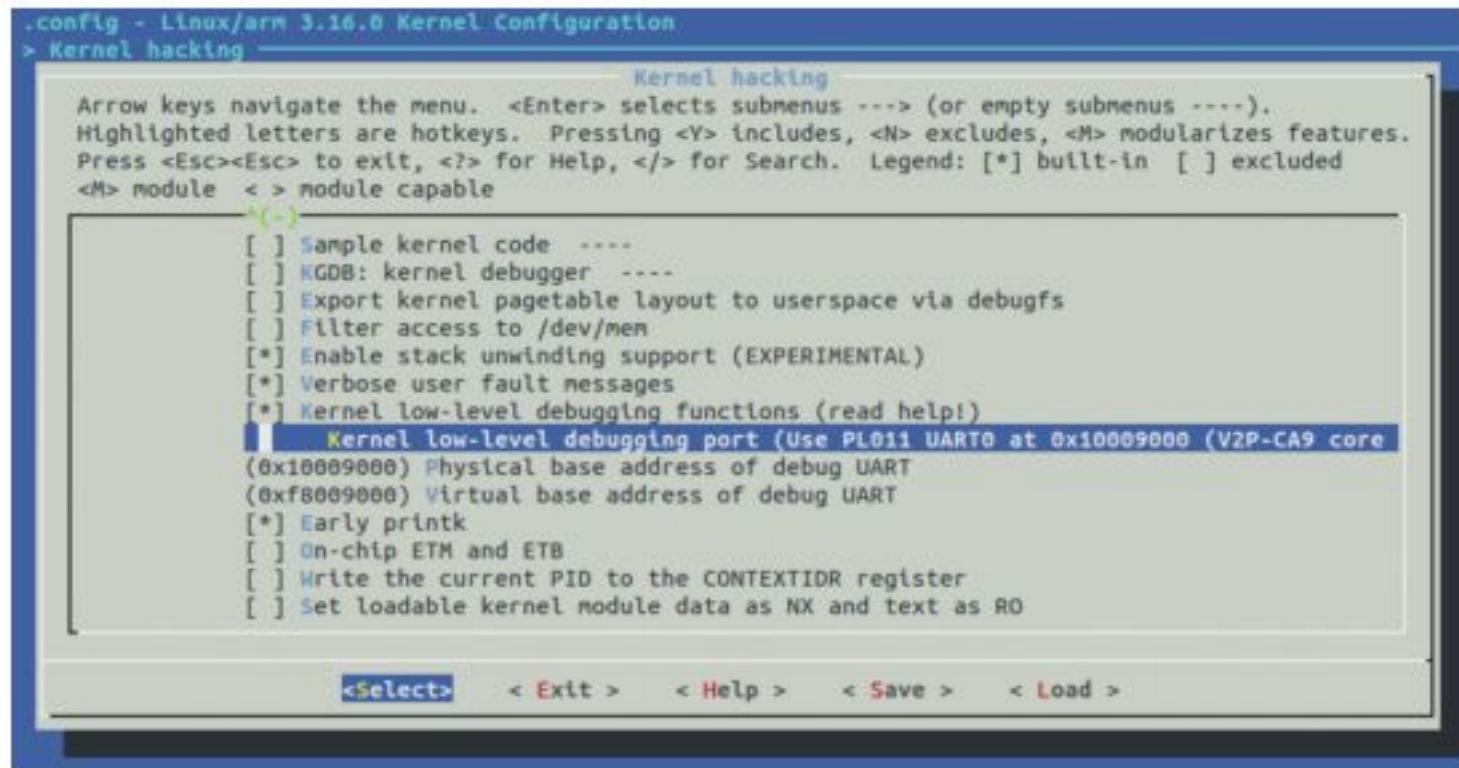


图 20.8 配置 DEBUG_LL 的端口

arch/arm/Kconfig.debug 会根据用户的配置选择对应的 arch/arm/include/debug/xxx.S，譬如：

```

config DEBUG_LL_INCLUDE
    string
    default "debug/B250.S" if DEBUG_LL_UART_8250 || DEBUG_UART_8250
    default "debug/pl01x.S" if DEBUG_LL_UART_PL01X || DEBUG_UART_PL01X
    default "debug/sirf.S" if DEBUG_SIRFPRIMA2_UART1 || DEBUG_SIRFMARCO_UART1

```

上述配置选项对应的 CONFIG_DEBUG_LL_INCLUDE 这个宏会被内核的 arch/arm/boot/compressed/debug.S、arch/arm/boot/compressed/head.S、arch/arm/kernel/debug.S 和 arch/arm/kernel/head.S 以 "#include CONFIG_DEBUG_LL_INCLUDE" 的形式引用。

20.6 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，最后通过 gpiochip_add() 注册 gpio_chip。GPIO 驱动可以存在于 drivers/gpio 目录中，但是在 GPIO 兼有多种功能且需要复杂配置的情况下，GPIO 的驱动部分往往直接移到 drivers/pinctrl 目录下并连同 pinmux 一起实现，而不存在于 drivers/gpio 目录中。

gpio_chip 结构体封装了底层硬件的 GPIO enable()/disable() 等操作，它的定义如代码清单 20.15 所示。

代码清单 20.15 gpio_chip 结构体

```

1 struct gpio_chip {
2     const char          *label;
3     struct device        *dev;
4     struct module       *owner;
5
6     int             (*request)(struct gpio_chip *chip,
7                               unsigned offset);
8     void            (*free)(struct gpio_chip *chip,
9                           unsigned offset);
10
11    int            (*direction_input)(struct gpio_chip *chip,
12                                    unsigned offset);
13    int            (*get)(struct gpio_chip *chip,
14                      unsigned offset);
15    int            (*direction_output)(struct gpio_chip *chip,
16                                     unsigned offset, int value);
17    int            (*set_debounce)(struct gpio_chip *chip,
18                                unsigned offset, unsigned debounce);
19
20    void           (*set)(struct gpio_chip *chip,
21                      unsigned offset, int value);
22
23    int            (*to_irq)(struct gpio_chip *chip,
24                           unsigned offset);
25
26    void           (*dbg_show)(struct seq_file *s,
27                             struct gpio_chip *chip);
28    int             base;
29    u16            ngpio;
30    const char      *const *names;
31    unsigned         can_sleep:l;
32    unsigned         exported:l;
33
34 #if defined(CONFIG_OF_GPIO)
35     /*
36      * If CONFIG_OF is enabled, then all GPIO controllers described in the
37      * device tree automatically may have an OF translation
38      */
39     struct device_node *of_node;
40     int of_gpio_n_cells;
41     int (*of_xlate)(struct gpio_chip *gc,
```

```

42                     const struct of_phandle_args *gpiospec, u32 *flags);
43 #endif
44 };

```

通过这层封装，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、Regulator 都有以 devm_ 开头的 API，使用这部分 API 的时候，内核会有类似于 Java 的资源自动回收机制，因此在代码中进行出错处理时，无须释放相关的资源。

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

在拥有设备树支持的情况下，我们可以通过设备树来描述某 GPIO 控制器提供的 GPIO 引脚被具体设备使用的情况。在 GPIO 控制器对应的节点中，需定义 #gpio-cells 和 gpio-controller 属性，具体的设备节点则通过 xxx-gpios 属性来引用 GPIO 控制器节点及 GPIO 引脚。

如 VEXPRESS 电路板 DT 文件 arch/arm/boot/dts/vexpress-v2m.dtsi 中有如下 GPIO 控制器节点：

```

v2m_sysreg: sysreg@00000 {
    compatible = "arm,vexpress-sysreg";
    reg = <0x00000 0x1000>;
    gpio-controller;
    #gpio-cells = <2>;
};

```

VEXPRESS 电路板上的 MMC 控制器会使用该节点 GPIO 控制器提供的 GPIO 引脚，则具体的 mmc@05000 设备节点会通过 -gpios 属性引用 GPIO：

```

mmc@05000 {
    compatible = "arm,p1180", "arm,primecell";
    reg = <0x05000 0x1000>;
}

```

```

    interrupts = <9 10>;
    cd-gpios = <&v2m_sysreg 0 0>;
    wp-gpios = <&v2m_sysreg 1 0>;
    ...
};

}

```

其中的 cd-gpios 用于 SD/MMC 卡的探测，而 wp-gpios 用于写保护，MMC 主机控制器驱动会通过如下方法获取这两个 GPIO，详见于 drivers/mmc/host/mmci.c：

```

static void mmci_dt_populate_generic_pdata(struct device_node *np,
                                            struct mmci_platform_data *pdata)
{
    ...
    pdata->gpio_wp = of_get_named_gpio(np, "wp-gpios", 0);
    pdata->gpio_cd = of_get_named_gpio(np, "cd-gpios", 0);
    ...
}

```

20.7 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。

1. 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图 20.9 所示。

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，该描述符的 pins 成员中包含所有引脚的列表。可以通过代码清单 20.16 的方法来注册这个 pin 控制器并命名它的所有引脚。

	A	B	C	D	E	F	G	H
8	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0

图 20.9 一个 PGA 封装的芯片的引脚排布

代码清单 20.16 pinctrl 引脚描述

```

1 #include <linux/pinctrl/pinctrl.h>
2
3 const struct pinctrl_pin_desc foo_pins[] = {
4     PINCTRL_PIN(0, "A8"),
5     PINCTRL_PIN(1, "B8"),
6     PINCTRL_PIN(2, "C8"),

```

```

7      ...
8      PINCTRL_PIN(61, "F1"),
9      PINCTRL_PIN(62, "G1"),
10     PINCTRL_PIN(63, "H1"),
11 };
12
13 static struct pinctrl_desc foo_desc = {
14     .name = "foo",
15     .pins = foo_pins,
16     .npins = ARRAY_SIZE(foo_pins),
17     .maxpin = 63,
18     .owner = THIS_MODULE,
19 };
20
21 int __init foo_probe(void)
22 {
23     struct pinctrl_dev *pctl;
24
25     pctl = pinctrl_register(&foo_desc, <PARENT>, NULL);
26     if (IS_ERR(pctl))
27         pr_err("could not register foo pin driver\n");
28 }

```

2. 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^C 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_group_name() 和 get_group_pins()，将 pinctrl_ops 填充到前文 pinctrl_desc 的实例 foo_desc 中，如代码清单 20.17 所示。

代码清单 20.17 pinctrl 驱动对引脚分组

```

1 #include <linux/pinctrl/pinctrl.h>
2
3 struct foo_group {
4     const char *name;
5     const unsigned int *pins;
6     const unsigned num_pins;
7 };
8
9 static const unsigned int spi0_pins[] = { 0, 8, 16, 24 };
10 static const unsigned int i2c0_pins[] = { 24, 25 };
11
12 static const struct foo_group foo_groups[] = {
13     {
14         .name = "spi0_grp",
15         .pins = spi0_pins,
16         .num_pins = ARRAY_SIZE(spi0_pins),

```

```

17         },
18         {
19             .name = "i2c0_grp",
20             .pins = i2c0_pins,
21             .num_pins = ARRAY_SIZE(i2c0_pins),
22         },
23     };
24
25
26 static int foo_get_groups_count(struct pinctrl_dev *pctldev)
27 {
28     return ARRAY_SIZE(foo_groups);
29 }
30
31 static const char *foo_get_group_name(struct pinctrl_dev *pctldev,
32                                     unsigned selector)
33 {
34     return foo_groups[selector].name;
35 }
36
37 static int foo_get_group_pins(struct pinctrl_dev *pctldev, unsigned selector,
38                             unsigned ** const pins,
39                             unsigned * const num_pins)
40 {
41     *pins = (unsigned *) foo_groups[selector].pins;
42     *num_pins = foo_groups[selector].num_pins;
43     return 0;
44 }
45
46 static struct pinctrl_ops foo_pctrl_ops = {
47     .get_groups_count = foo_get_groups_count,
48     .get_group_name = foo_get_group_name,
49     .get_group_pins = foo_get_group_pins,
50 };
51
52
53 static struct pinctrl_desc foo_desc = {
54     ...
55     .pctlops = &foo_pctrl_ops,
56 };

```

`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 子系统的底层会调用上述回调函数。

3. 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。在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回调函数（pinctrl_desc 的 confops 成员函数），如代码清单 20.18 所示。

代码清单 20-18 引脚的配置

```

39  {
40      ...
41  }
42
43 static struct pinconf_ops foo_pconf_ops = {
44     .pin_config_get = foo_pin_config_get,
45     .pin_config_set = foo_pin_config_set,
46     .pin_config_group_get = foo_pin_config_group_get,
47     .pin_config_group_set = foo_pin_config_group_set,
48 };
49
50 /* Pin config operations are handled by some pin controller */
51 static struct pinctrl_desc foo_desc = {
52     ...
53     .confops = &foo_pconf_ops,
54 };

```

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。

4. 与 GPIO 子系统的交互

pinctrl 驱动所覆盖的引脚可同时作为 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，在这种情况下，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 驱动中可以通过如下代码注册两个 GPIO 范围，如代码清单 20.19 所示。

代码清单 20.19 GPIO 与 pinctrl 引脚的映射

```

1 struct gpio_chip chip_a;
2 struct gpio_chip chip_b;
3
4 static struct pinctrl_gpio_range gpio_range_a = {
5     .name = "chip a",
6     .id = 0,
7     .base = 32,
8     .pin_base = 32,
9     .npins = 16,
10    .gc = &chip_a;

```

```

11  };
12
13 static struct pinctrl_gpio_range gpio_range_b = {
14     .name = "chip b",
15     .id = 0,
16     .base = 48,
17     .pin_base = 64,
18     .npins = 8,
19     .gc = &chip_b;
20 };
21
22 {
23     struct pinctrl_dev *pctl;
24     ...
25     pinctrl_add_gpio_range(pctl, &gpio_range_a);
26     pinctrl_add_gpio_range(pctl, &gpio_range_b);
27 }

```

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 和引脚的映射关系，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 功能使能 / 禁止的回调函数，下面可以看到它的更多细节。

5. 引脚复用 (pinmux)

在 pinctrl 驱动中可处理引脚复用，它定义了功能 (FUNCTIONS)，驱动可以设置某功能的使能或者禁止。各个功能联合起来组成一个一维数组，譬如 {spi0, i2c0, mmc0} 就描述了 3 个不同的功能。

一个特定的功能总是要求由一些引脚组来完成，引脚组的数量可以为 1 个或者多个。假设对前文所描述的 PGA 封装的 SoC 而言，引脚分组如图 20.10 所示

假设 I²C 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 功能则可以由 {A8, A7, A6, A5} 和 {G4, G3, G2, G1}，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）。

据此，由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 驱动中相关的回调函数。

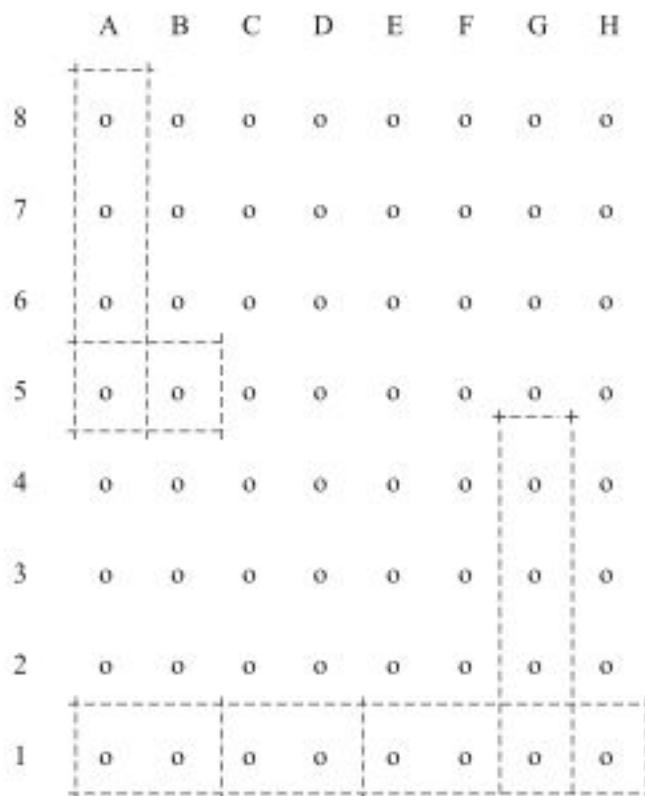


图 20.10 针对 PGA 封装的 SoC 的引脚分组

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：

```
{
    {"map-spi0", spi0, pinctrl0, fspi0, gspi0},
    {"map-i2c0", i2c0, pinctrl0, fi2c0, gi2c0}
}
```

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以行使其他的功能或 GPIO，则 pinctrl 核心层会让该次申请失败。

在特定 pinctrl 驱动中 pinmux 相关的代码主要处理如何使能 / 禁止某一 { 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，相关的回调函数被组织进一个 pinmux_ops 结构体中，而该结构体的实例最终成为前文 pinctrl_desc 的 pmxops 成员，如代码清单 20.20 所示。

代码清单 20.20 pinmux 的实现

```

1 #include <linux/pinctrl/pinctrl.h>
2 #include <linux/pinctrl/pinmux.h>
3
4 struct foo_group {
5     const char *name;
```

```
6         const unsigned int *pins;
7         const unsigned num_pins;
8     };
9
10 static const unsigned spi0_0_pins[] = { 0, 8, 16, 24 };
11 static const unsigned spi0_1_pins[] = { 38, 46, 54, 62 };
12 static const unsigned i2c0_pins[] = { 24, 25 };
13 static const unsigned mmc0_1_pins[] = { 56, 57 };
14 static const unsigned mmc0_2_pins[] = { 58, 59 };
15 static const unsigned mmc0_3_pins[] = { 60, 61, 62, 63 };
16
17 static const struct foo_group foo_groups[] = {
18     {
19         .name = "spi0_0_grp",
20         .pins = spi0_0_pins,
21         .num_pins = ARRAY_SIZE(spi0_0_pins),
22     },
23     {
24         .name = "spi0_1_grp",
25         .pins = spi0_1_pins,
26         .num_pins = ARRAY_SIZE(spi0_1_pins),
27     },
28     {
29         .name = "i2c0_grp",
30         .pins = i2c0_pins,
31         .num_pins = ARRAY_SIZE(i2c0_pins),
32     },
33     {
34         .name = "mmc0_1_grp",
35         .pins = mmc0_1_pins,
36         .num_pins = ARRAY_SIZE(mmc0_1_pins),
37     },
38     {
39         .name = "mmc0_2_grp",
40         .pins = mmc0_2_pins,
41         .num_pins = ARRAY_SIZE(mmc0_2_pins),
42     },
43     {
44         .name = "mmc0_3_grp",
45         .pins = mmc0_3_pins,
46         .num_pins = ARRAY_SIZE(mmc0_3_pins),
47     },
48 };
49
50
51 static int foo_get_groups_count(struct pinctrl_dev *pctldev)
52 {
53     return ARRAY_SIZE(foo_groups);
54 }
55
```

```
56 static const char *foo_get_group_name(struct pinctrl_dev *pctldev,
57                                     unsigned selector)
58 {
59     return foo_groups[selector].name;
60 }
61
62 static int foo_get_group_pins(struct pinctrl_dev *pctldev, unsigned selector,
63                               unsigned ** const pins,
64                               unsigned * const num_pins)
65 {
66     *pins = (unsigned *) foo_groups[selector].pins;
67     *num_pins = foo_groups[selector].num_pins;
68     return 0;
69 }
70
71 static struct pinctrl_ops foo_pctrl_ops = {
72     .get_groups_count = foo_get_groups_count,
73     .get_group_name = foo_get_group_name,
74     .get_group_pins = foo_get_group_pins,
75 };
76
77 struct foo_pmx_func {
78     const char *name;
79     const char * const *groups;
80     const unsigned num_groups;
81 };
82
83 static const char * const spi0_groups[] = { "spi0_0_grp", "spi0_1_grp" };
84 static const char * const i2c0_groups[] = { "i2c0_grp" };
85 static const char * const mmc0_groups[] = { "mmc0_1_grp", "mmc0_2_grp",
86                                         "mmc0_3_grp" };
87
88 static const struct foo_pmx_func foo_functions[] = {
89     {
90         .name = "spi0",
91         .groups = spi0_groups,
92         .num_groups = ARRAY_SIZE(spi0_groups),
93     },
94     {
95         .name = "i2c0",
96         .groups = i2c0_groups,
97         .num_groups = ARRAY_SIZE(i2c0_groups),
98     },
99     {
100        .name = "mmc0",
101        .groups = mmc0_groups,
102        .num_groups = ARRAY_SIZE(mmc0_groups),
103    },
104};
```

```
105
106 int foo_get_functions_count(struct pinctrl_dev *pctldev)
107 {
108     return ARRAY_SIZE(foo_functions);
109 }
110
111 const char *foo_get_fname(struct pinctrl_dev *pctldev, unsigned selector)
112 {
113     return foo_functions[selector].name;
114 }
115
116 static int foo_get_groups(struct pinctrl_dev *pctldev, unsigned selector,
117                           const char * const **groups,
118                           unsigned * const num_groups)
119 {
120     *groups = foo_functions[selector].groups;
121     *num_groups = foo_functions[selector].num_groups;
122     return 0;
123 }
124
125 int foo_enable(struct pinctrl_dev *pctldev, unsigned selector,
126                 unsigned group)
127 {
128     u8 regbit = (1 << selector + group);
129
130     writeb((readb(MUX) | regbit), MUX)
131     return 0;
132 }
133
134 void foo_disable(struct pinctrl_dev *pctldev, unsigned selector,
135                   unsigned group)
136 {
137     u8 regbit = (1 << selector + group);
138
139     writeb((readb(MUX) & ~regbit), MUX)
140     return 0;
141 }
142
143 struct pinmux_ops foo_pmxops = {
144     .get_functions_count = foo_get_functions_count,
145     .get_function_name = foo_get_fname,
146     .get_function_groups = foo_get_groups,
147     .enable = foo_enable,
148     .disable = foo_disable,
149 };
150
151 /* Pinmux operations are handled by some pin controller */
152 static struct pinctrl_desc foo_desc = {
153     ...
154 }
```

```

154     .pctllops = &foo_pctrl_ops,
155     .pmxops = &foo_pmxops,
156 };

```

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

```

static struct pinctrl_map __initdata mapping[] = {
    PIN_MAP_MUX_GROUP("foo-i2c.0", PINCTRL_STATE_DEFAULT, "pinctrl-foo", NULL, "i2c0"),
};

```

又由于 1 个功能可由两个不同的引脚组实现，所以对于同 1 个功能可能形成有两个可选引脚组的 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"),
};

```

其中调用的 PIN_MAP_MUX_GROUP 是一个快捷宏，用于赋值 pinctrl_map 的各个成员：

```

#define PIN_MAP_MUX_GROUP(dev, state, pinctrl, grp, func) \
{ \
    .dev_name = dev, \
    .name = state, \
    .type = PIN_MAP_TYPE_MUX_GROUP, \
    .ctrl_dev_name = pinctrl, \
    .data.mux = { \
        .group = grp, \
        .function = func, \
    }, \
}

```

当然，这种映射关系最好是在设备树中通过节点的属性进行，具体的节点属性的定义方法依赖于具体的 pinctrl 驱动，最终在 pinctrl 驱动中通过 pinctrl_ops 结构体的 .dt_node_to_map() 成员函数读出属性并建立映射表。

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

```

p = devm_pinctrl_get(dev);
s = pinctrl_lookup_state(p, "spi0-pos-A");
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");

```

若想在运行时切换位置 A 和 B 的引脚组以行使 SPI 的接口功能，代码结构类似清单 20.21 所示。

代码清单 20.21 pinctrl_lookup_state() 和 pinctrl_select_state()

```

1  foo_probe()
2  {
3      /* Setup */
4      p = devm_pinctrl_get(&device);
5      if (IS_ERR(p))
6          ...
7
8      s1 = pinctrl_lookup_state(foo->p, " spi0-pos-A");
9      if (IS_ERR(s1))
10         ...
11
12     s2 = pinctrl_lookup_state(foo->p, " spi0-pos-B");
13     if (IS_ERR(s2))
14         ...
15 }
16
17 foo_switch()
18 {
19     /* Enable on position A */
20     ret = pinctrl_select_state(s1);
21     if (ret < 0)
22         ...
23
24     ...
25
26     /* Enable on position B */
27     ret = pinctrl_select_state(s2);
28     if (ret < 0)
29         ...
30
31     ...
32 }
```

对于 "default" 状态下的引脚配置，驱动一般不需要完成 devm_pinctrl_get_select(dev, "default") 的调用。譬如对于 arch/arm/boot/dts/prima2-evb.dts 中的如下引脚组：

```

peri-iobg {
    uart@b0060000 {
        pinctrl-names = "default";
        pinctrl-0 = <&uart1_pins_a>;
    };
    spi@b00d0000 {
        pinctrl-names = "default";
        pinctrl-0 = <&spi0_pins_a>;
    };
    spi@b0170000 {
        pinctrl-names = "default";
        pinctrl-0 = <&spil_pins_a>;
    };
}
```

```
    );
}
```

由于 pinctrl-names 都是 "default" 的，所以 pinctrl 核实际会自动做类似 devm_pinctrl_get_select(dev, "default") 的操作。

20.8 时钟驱动

在一个 SoC 中，晶振、PLL、驱动和门等会形成一个时钟树形结构，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通用时钟框架以解决这个碎片化问题。之所以称为通用时钟，是因为这个通用主要体现在：

- 1) 统一的 clk 结构体，统一的定义于 clk.h 中的 clk API，这些 API 会调用统一的 clk_ops 中的回调函数；这个统一的 clk 结构体的定义如代码清单 20.22 所示。

代码清单 20.22 clk 结构体

```
1 struct clk {
2     const char          *name;
3     const struct clk_ops *ops;
4     struct clk_hw        *hw;
5     char                **parent_names;
6     struct clk           **parents;
7     struct clk           *parent;
8     struct hlist_head    children;
9     struct hlist_node   child_node;
10    ...
11};
```

其中第 3 行的 clk_ops 定义是关于时钟使能、禁止、计算频率等的操作集，定义如代码清单 20.23 所示。

代码清单 20.23 clk_ops 结构体

```
1 struct clk_ops {
2     int      (*prepare)(struct clk_hw *hw);
3     void    (*unprepare)(struct clk_hw *hw);
4     int      (*enable)(struct clk_hw *hw);
5     void    (*disable)(struct clk_hw *hw);
6     int      (*is_enabled)(struct clk_hw *hw);
7     unsigned long (*recalc_rate)(struct clk_hw *hw,
8                                  unsigned long parent_rate);
9     long    (*round_rate)(struct clk_hw *hw, unsigned long,
10                        unsigned long *);
11    int      (*set_parent)(struct clk_hw *hw, u8 index);
```

```

12          u8      (*get_parent)(struct clk_hw *hw);
13          int     (*set_rate)(struct clk_hw *hw, unsigned long);
14          void    (*init)(struct clk_hw *hw);
15  };

```

2) 对具体的 SoC 如何去实现针对自己 SoC 的 clk 驱动, 如何提供硬件特定的回调函数的方法也进行了统一。

在代码清单 20.22 这个通用的 clk 结构体中, 第 4 行的 clk_hw 是联系 clk_ops 中回调函数和具体硬件细节的纽带, clk_hw 中只包含通用时钟结构体的指针以及具体硬件的 init 数据, 如代码清单 20.24 所示。

代码清单 20.24 clk_hw 结构体

```

1 struct clk_hw {
2         struct clk *clk;
3         const struct clk_init_data *init;
4 };

```

其中的 clk_init_data 包含了具体时钟的名称、可能的父级时钟的名称列表 parent_names、可能的父级时钟数量 num_parents 等, 实际上这些名称的匹配对建立时钟间的父子关系功不可没, 如代码清单 20.25 所示。

代码清单 20.25 clk_init_data 结构体

```

1 struct clk_init_data {
2         const char           *name;
3         const struct clk_ops *ops;
4         const char           **parent_names;
5         u8                  num_parents;
6         unsigned long        flags;
7 };

```

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 clk 结构体的 clk_ops 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 的回调函数中，我们便可以通过 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);

    /* 访问硬件读写寄存器以改变时钟的状态 */
    ...

    return 0;
};
```

在具体的 clk 驱动中，需要通过 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 的一个 lookup（这样可以通过 con_id 或者 dev_id 字符串寻找到这个 clk），这两个函数的原型为：

```
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 门、clk 驱动等），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，与该驱动对应的芯片 SiRFprimaII 的外围接了一个 26MHz 的晶振和一个 32.768kHz 的 RTC 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 节点，则它的相关驱动代码如清单 20.26 所示。

代码清单 20.26 clk 驱动案例

```
1 static unsigned long pll_clk_recalc_rate(struct clk_hw *hw,
2 unsigned long parent_rate)
3 {
4     unsigned long fin = parent_rate;
5     struct clk_pll *clk = to_pllclk(hw);
6     ...
7 }
8
9 static long pll_clk_round_rate(struct clk_hw *hw, unsigned long rate,
10 unsigned long *parent_rate)
11 {
12     ...
13 }
14
15 static int pll_clk_set_rate(struct clk_hw *hw, unsigned long rate,
16 unsigned long parent_rate)
17 {
18     ...
19 }
20
21 static struct clk_ops std_pll_ops = {
22     .recalc_rate = pll_clk_recalc_rate,
23     .round_rate = pll_clk_round_rate,
24     .set_rate = pll_clk_set_rate,
25 };
26
27 static const char *pll_clk_parents[] = {
28     "osc",
29 };
30
31 static struct clk_init_data clk_pll1_init = {
32     .name = "pll1",
33     .ops = &std_pll_ops,
34     .parent_names = pll_clk_parents,
35     .num_parents = ARRAY_SIZE(pll_clk_parents),
36 };
37
38 static struct clk_init_data clk_pll2_init = {
39     .name = "pll2",
40     .ops = &std_pll_ops,
41     .parent_names = pll_clk_parents,
42     .num_parents = ARRAY_SIZE(pll_clk_parents),
43 };
44
45 static struct clk_init_data clk_pll3_init = {
46     .name = "pll3",
47     .ops = &std_pll_ops,
48     .parent_names = pll_clk_parents,
```

```

49 .num_parents = ARRAY_SIZE(pll_clk_parents),
50 };
51
52 static struct clk_pll clk_pll1 = {
53 .regofs = SIRFSOC_CLKC_PLL1_CFG0,
54 .hw = {
55     .init = &clk_pll1_init,
56 },
57 };
58
59 static struct clk_pll clk_pll2 = {
60 .regofs = SIRFSOC_CLKC_PLL2_CFG0,
61 .hw = {
62     .init = &clk_pll2_init,
63 },
64 };
65
66 static struct clk_pll clk_pll3 = {
67 .regofs = SIRFSOC_CLKC_PLL3_CFG0,
68 .hw = {
69     .init = &clk_pll3_init,
70 },
71 };
72 void __init sirfsoc_of_clk_init(void)
73 {
74 ...
75
76 /* These are always available (RTC and 26MHz OSC) */
77 clk = clk_register_fixed_rate(NULL, "rtc", NULL,
78     CLK_IS_ROOT, 32768);
79 BUG_ON(!clk);
80 clk = clk_register_fixed_rate(NULL, "osc", NULL,
81     CLK_IS_ROOT, 26000000);
82 BUG_ON(!clk);
83
84 clk = clk_register(NULL, &clk_pll1.hw);
85 BUG_ON(!clk);
86 clk = clk_register(NULL, &clk_pll2.hw);
87 BUG_ON(!clk);
88 clk = clk_register(NULL, &clk_pll3.hw);
89 BUG_ON(!clk);
90 ...
91 }

```

另外，目前内核更加倡导的方法是通过设备树来描述电路板上的时钟树，以及时钟和设备之间的绑定关系。通常我们需要在 clk 控制器的节点中定义 #clock-cells 属性，并且在 clk 驱动中通过 of_clk_add_provider() 注册时钟控制器为一个时钟树的提供者（Provider），并建立

系统中各个时钟和索引的映射表，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 节点上的 `clocks=<&clks index>` 属性指向其引用的 clk 控制器节点以及使用的时钟的索引，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，一定要通过通用 clk API 来操作所有的时钟，而不要直接通过读写 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，某些硬件使能 / 禁止时钟可能会引起睡眠以使得使能 / 禁止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()` 则同时完成准备

和使能的工作，当然也只能在可能睡眠的上下文调用该 API。

20.9 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，而且也定义了用具体的 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了，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，相关的代码如清单 20.27 所示。

代码清单 20.27 dmaengine API 的使用

```

1 static int dmaengine_pcm_prepare_and_submit(struct snd_pcm_substream *substream)
2 {
3     struct dmaengine_pcm_runtime_data *prtd = substream_to_prtd(substream);
4     struct dma_chan *chan = prtd->dma_chan;
5     struct dma_async_tx_descriptor *desc;
6     enum dma_transfer_direction direction;
7     unsigned long flags = DMA_CTRL_ACK;
8
9     ...
10    desc = dmaengine_prep_dma_cyclic(chan,
11        substream->runtime->dma_addr,
12        snd_pcm_lib_buffer_bytes(substream),
13        snd_pcm_lib_period_bytes(substream), direction, flags);
14 ...
15    desc->callback = dmaengine_pcm_dma_complete;
16    desc->callback_param = substream;
17    prtd->cookie = dmaengine_submit(desc);
18 }
19
20 int snd_dmaengine_pcm_trigger(struct snd_pcm_substream *substream, int cmd)
21 {
22     struct dmaengine_pcm_runtime_data *prtd = substream_to_prtd(substream);
23     int ret;
24     switch (cmd) {
25     case SNDDRV_PCM_TRIGGER_START:
26         ret = dmaengine_pcm_prepare_and_submit(substream);
27         ...
28         dma_async_issue_pending(prtd->dma_chan);
29         break;
30     case SNDDRV_PCM_TRIGGER_RESUME:
31     case SNDDRV_PCM_TRIGGER_PAUSE_RELEASE:
32         dmaengine_resume(prtd->dma_chan);
33         break;
34     ...
35 }
```

这个过程可分为 4 步：

- 1) 通过 `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`，本例是一个周期性 DMA，因此第 10 行调用的是 `dmaengine_prep_dma_cyclic()`）。
- 2) 通过 `dmaengine_submit()` 将该描述符插入 `dmaengine` 驱动的传输队列（见第 17 行）。
- 3) 在需要传输的时候通过类似 `dma_async_issue_pending()` 的调用启动对应 DMA 通道上的传输（见第 28 行）。
- 4) DMA 的完成，或者周期性 DMA 完成了一个周期，都会引发 DMA 传输描述符的完成回调函数被调用（本例中的赋值在第 15 行，对应的回调函数是 `dmaengine_pcm_dma_complete`）。

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、将 DMA 描述符插入传输队列以及启动 DMA 传输的过程。

除了前文提到的用 `dmaengine_prep_dma_cyclic()` 定义周期性 DMA 传输外，还有一组类似的 API 可以用来定义各种类型的 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` 结构体中的这些成员函数（定义在 `include/linux/dmaengine.h` 头文件中）：

```
/*
 * struct dma_device - info on the entity supplying DMA services
 *
 * @device_prep_dma_memcpy: prepares a memcpy operation
 * @device_prep_dma_xor: prepares a xor operation
 * @device_prep_dma_xor_val: prepares a xor validation operation
 * @device_prep_dma_pq: prepares a pq operation
 * @device_prep_dma_pq_val: prepares a pqzero_sum operation
 * @device_prep_dma_memset: prepares a memset operation
 * @device_prep_dma_interrupt: prepares an end of chain interrupt operation
 * @device_prep_slave_sg: prepares a slave dma operation
 * @device_prep_dma_cyclic: prepare a cyclic dma operation suitable for audio.
 *   The function takes a buffer of size buf_len. The callback function will
 *   be called after period_len bytes have been transferred.
 * @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，并通过 `dma_async_device_register()` 完成注册。在其各个成员函数中，一般会通过链表来管理 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，每当 DMA 传输完成后，在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` 中设置的回调函数，该回调函数来源于使用 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。

20.10 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、时钟、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回调函数。

Linux 内核社区针对 pinctrl、时钟、GPIO、DMA 提供独立的子系统，既给具体的设备驱动提供了统一的 API，进一步提高了设备驱动的跨平台性，又为每个 SoC 和设备实现这些底层 API 定义好了条条框框，从而可以在最大程度上避免每个硬件实现过多的冗余代码。