1.SoCLinux　底层驱动的组成和现状

为了让 Linux 在一个全的 ARMSoC 上运行，需要供给大量的底层支撑，如定时器节拍、中断掌握器、SMP 启动、CPUhotplug 以及底层的 GPIO、clock、pinctrl 和 DMA 硬件的封装等。定时器节拍、中断掌握器、SMP 启动和 CPU hotplug 这几局部相对来说没有像早期 GPIO、clock、pinctrl 和 DMA 的实现那么杂乱，根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供给支撑，中断掌握器的驱动则使得 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，CPUhotplug 则运行运行时挂载或拔除 CPU。这些工作，在 Linux3.7 内核中，进展了良好的层次划分和架构设计。

在 GPIO、clock、pinctrl 和 DMA 驱动方面，Linux2.6 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 对这些根底设施实现方面存在巨大差异,而且每个 SoC 仍旧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。

社区必需转变这种局面，于是内核社区在 2023～2023 年进展了如下工作，这些工作在目前的 3.7 内核中根本预备就绪：

§ ST-Ericsson 的工程师 LinusWalleij 供给了的 pinctrl 驱动架构，内核增加一个 drivers/pinctrl 名目，支撑 SoC 上的引脚复用，各个 SoC 的实现代码统一放入该名目；

§ TI 的工程师 MikeTurquette 供给了 commonclk 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 名目；

§ 建议各 SoC 统一承受 dmaengine 架构实现 DMA 驱动，该架构供给了通用的 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 名目；

§ 在 GPIO 方面，drivers/gpio 下的 gpiolib 已能与的 pinctrl 完善共存，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 构造体的成员函数。

经过以上工作，根本上就把芯片底层的根底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展DeviceTree 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。

除了上述根底设施以外，在将 Linux 移植入的 SoC 过程中，工程师常常猛烈依靠于早期的printk 功能，内核则供给了相关的 DEBUG_LL 和 EARLY_PRINTK 支持，只需要 SoC 供给商实现少量的 callback 或宏。

本文主要对上述各个组成局部进展架构上的剖析以及关键的实现局部的实例分析，以求完整归纳将 Linux 移植入 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

Linux2.6 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ARMLinux 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*IRQhandlerforthetimer
129*/
130staticirqreturn_t
```

```
131s3c2410_timer_interrupt(intirq,void*dev_id,structpt_regs*regs)
132{
133write_seqlock(&xtime_lock);
134timer_tick(regs);
135write_sequnlock(&xtime_lock);
136returnIRQ_HANDLED;
137}
138
139staticstructirqactions3c2410_timer_irq={
140.name                   =  “S3C2410TimerTick  “,
141.flags                 =SA_INTERRUPT|SA_TIMER,
142.handler                    =s3c2410_timer_interrupt,
143};
252staticvoid_inits3c2410_timer_init(void)
253{
254s3c2410_timer_setup;
255setup_irq(IRQ_TIMER4,&s3c2410_timer_irq);
256}
257
```

当前 Linux 多承受 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，依据系统的运行状况，以大事驱动的方式动态打算下一个tick 在何时发生。假设画一个时间轴，周期节拍的系统 tick 中断发生的时序看起来如下：



@宋宝华Barry

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：



@宋宝华Barry

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 构造体中，实现其 set_mode和 set_next_event成员函数；在 clocksource 构造体中，主要实现 read成员函数。而定时器中断效劳程序中，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：

```
61staticirqreturn_txxx_timer_interrupt(intirq,void*dev_id)
```

以上内容仅为本文档的试下载部分，为可阅读页数的一半内容。如要下载或阅读全文，请访问：
https://d.book118.com/836143032120010155