【1】OMAP335X-内核 BSP 之 C 代码启动那些事.

分类： linux-驱动篇 linux-内核篇 开发-心的体会 2013-11-25 19:08 888 人阅读 评论(0) 收藏 举报

PC 系统 ：ubuntu10.04

MPU 平台：OAMP3352

内核版本：3.2.0

声明：我讲解的范畴是从内核解压以后经过汇编代码执行最后跳到第一个 C 代码这个点开始讲解，一直讲到文件系统被正确的挂载起来，用户可以登入！至于之前的汇编代码我会以后另开文章讲解。

目标：本文想阐述清楚 OAMP335X 这个平台的 BSP 部分代码的启动过程。

第一步：

init/main.c 文件是汇编过来的第一个被调用的 C 代码文件；

asmlinkage void __init start_kernel(void)是汇编过来的第一个被执行的 C 函数；

```
asmlinkage void __init start_kernel(void)
{
 ..........
 setup_arch(&command_line);
 ..........
 rest_init();
}
```

这个函数中有很多的调用函数，每一个都具有举足轻重的地位，够我们研究一段时间的，它们中的许多都肩负着初始化内核中的某个子系统的重要使命，而 Linux 内核中每一个子系统

都错综复杂，牵涉到各种软件、硬件的复杂算法，所以我们要慢慢的理解他。这里我们讲解其中和本文章相关的两个函数，其他的以后用到再讲。

```
void __init setup_arch(char **cmdline_p)

{

struct machine_desc *mdesc;

setup_processor();
mdesc = setup_machine_fdt(__atags_pointer);
if (!mdesc)
mdesc = setup_machine_tags(machine_arch_type);
machine_desc = mdesc;
machine_name = mdesc->name;
if (mdesc->restart_mode)

reboot_setup(&mdesc->restart_mode);

.....................

}
```

这里的 setup_machine()函数的作用就是找到我们想要的 struct machine_desc 类型的变量，也就是在 BSP 里定义那个变量。

```
MACHINE_START(AM335XEVM, "am335xevm")
 /* Maintainer: Texas Instruments */
    .atag_offset = 0x100,
    .map_io  = am335x_evm_map_io,
    .init_early = am33xx_init_early,
    .init_irq = ti81xx_init_irq,
    .handle_irq      = omap3_intc_handle_irq,
    .timer  = &omap3_am33xx_timer,
    .init_machine = am335x_evm_init,
MACHINE_END
```

这样我们板级代码的初始化就完成了，但这只是仅仅数据结构得到初始化，真正的资源都在这个结构体里面，那在哪里通过这个变量调用这个数据结构里面的成员或成员函数的呢？

在 arch/arm/kernel/setup.c 里我们看到这样一个函数

```
static int __init customize_machine(void)
{
        /* customizes platform devices, or adds new ones */
        if (machine_desc->init_machine)
                machine_desc->init_machine();
        return 0;
}
arch_initcall(customize_machine);
```

终于看到了，成员函数 init_machine 就是在这里被调用的。但是它没有被显示调用，而是放在了 arch_initcall 这个宏里，很显然这个宏肯定在什么地方被调用了！

第二步：

我们先看一些宏的定义（定义在文件 include/linux/init.h 中）

```
#define pure_initcall(fn)               __define_initcall("0",fn,0)
#define core_initcall(fn)               __define_initcall("1",fn,1)
#define core_initcall_sync(fn)          __define_initcall("1s",fn,1s)
#define postcore_initcall(fn)           __define_initcall("2",fn,2)
#define postcore_initcall_sync(fn)      __define_initcall("2s",fn,2s)
#define arch_initcall(fn)               __define_initcall("3",fn,3)
#define arch_initcall_sync(fn)          __define_initcall("3s",fn,3s)
```

#define subsys_initcall(fn) __define_initcall("4",fn,4)

#define subsys_initcall_sync(fn) __define_initcall("4s",fn,4s)

#define fs_initcall(fn) __define_initcall("5",fn,5)

#define fs_initcall_sync(fn) __define_initcall("5s",fn,5s)

#define rootfs_initcall(fn) __define_initcall("rootfs",fn,rootfs)

#define device_initcall(fn) __define_initcall("6",fn,6)

#define device_initcall_sync(fn) __define_initcall("6s",fn,6s)

#define late_initcall(fn) __define_initcall("7",fn,7)

#define late_initcall_sync(fn) __define_initcall("7s",fn,7s)

#define __define_initcall(level,fn,id) static initcall_t __initcall_##fn##id __used __attribute__((__section__(".initcall" level ".init"))) = fn

这其中 initcall_t 是函数指针，原型：typedef int (*initcall_t)(void);

而编译器宏 __attribute__((__section__())) 则表示把对象放在一个这个由括号中的名称所指代的 section 中，__define_initcall("6",fn,6) 则是把 fn 的地址放到.initcall6.init 这个 selection 中

所以__define_initcall 的含义是：

1) 声明一个名称为__initcall_##fn##id 的函数指针；

2) 将这个函数指针初始化赋值为 fn；

3) 编译的时候需要把这个函数指针变量放置到名称为 ".initcall" level ".init"的 section 中

明确了__define_initcall 的含义，就知道了它其实是分别将这些初始化函数地址以指针方式放到各自的 section 中的。

SECTION“.initcall”level”.init”被放入 INITCALLS（include/asm-generic/vmlinux.lds.h)

```
#define INITCALLS                                       \
        *(.initcallearly.init)                          \
        VMLINUX_SYMBOL(__early_initcall_end) = .;       \
        *(.initcall0.init)                              \
        *(.initcall0s.init)                             \
        *(.initcall1.init)                              \
        *(.initcall1s.init)                             \
        *(.initcall2.init)                              \
        *(.initcall2s.init)                             \
        *(.initcall3.init)                              \
        *(.initcall3s.init)                             \
        *(.initcall4.init)                              \
        *(.initcall4s.init)                             \
        *(.initcall5.init)                              \
        *(.initcall5s.init)                             \
        *(.initcallrootfs.init)                         \
        *(.initcall6.init)                              \
        *(.initcall6s.init)                             \
        *(.initcall7.init)                              \
        *(.initcall7s.init)

VMLINUX_SYMBOL(__early_initcall_end) = .;
```

这里的 . 是一个特殊的符号，它是定位器，一个位置指针，指向程序地址空间内的某位置(或某 section 内的偏移，如果它在 SECTIONS 命令内的某 section 描述内)，该符号只能在 SECTIONS 命令内使用。

这句命令的意思，是对__early_initcall_end 变量赋值为当前 section 内的偏移

*(.initcall7s.init) 表示的是所有输入文件中的名为.initcall7s.init 的 SECTIONS

__initcall_start 和__initcall_end 以及 INITCALLS 中定义的 SECTION 都是在 arch/xxx/kernel/vmlinux.lds.S 中放在.init 段的。

```
SECTIONS

{

    .init : {

        __initcall_start = .;

            INITCALLS

        __initcall_end = .;

    }

}
```

.init 只是用于对 SECTIONS 进行分类

__initcall_start = .; 和__initcall_end = .; 表示分别对变量 __initcall_start 和 __initcall_end 赋值为当前 section 内的偏移。

而这些 SECTION 里的函数在初始化时被顺序执行（init 内核线程 ->do_basic_setup()[main.c#778]->do_initcalls()）。

程序（init/main.c 文件 do_initcalls()函数）如下，do_initcalls()把 XX.initcallXX.init 中的函数按顺序都执行一遍。

```
for (call = __early_initcall_end; call < __initcall_end; call++)
```

do_one_initcall(*call);

看到这里我们知道内核对于函数的执行顺序是通过一个链接脚本来进行排号。

也就是说这个链接脚本在内核编译链接完成后初始化函数的执行先后顺序就确定了。这是通过链接器在链接时调用链接脚本/arch/arm/kernel/vmlinux.lds 和 **include/asm-generic/vmlinux.lds.h** 来完成的。脚本规定了不同代码段，例如_init、text、data 等不同属性的代码段存放的位置。

假如同属_init 函数 A()和函数 B()就全部放在脚本中定义的_init 地址上，但是具体是 A()函数在前还是 B()函数在前呢?这是由目录下的 Makefile 文件来决定了。Makefile 文件中函数存放的先后顺序来决定了，假如 obj+y = A()在 obj+y = B()之前，则最后连接的时候 A 函数就在 B()函数之前执行了。所以最后结论是决定函数执行的先后顺序是由：

1.vmlinux.lds 链接脚本

2.驱动目录下 Makefile 文件

共同确定的；

但是我还是只看到 customize_machine()被放到了.initcall3.init 里。这只是说顺序是安排好了，那究竟它在哪里被调用呢？

在/init/main.c 里一个叫 **do_initcalls()**的函数里被调用，我们老看看这个函数的实现：

```
extern initcall_t __initcall_start[], __initcall_end[], __early_initcall_end[];

static void __init do_initcalls(void)

{

    initcall_t *fn;

    for (fn = __early_initcall_end; fn < __initcall_end; fn++)

        do_one_initcall(*fn);

}
```

看到第 1 行，很熟悉吧。在 for 循环里依次调用了从__early_initcall_start 开始到__initcall_end 结束的所有函数。customize_machine()也是在其间被调用了，也就是说

我们的 BSP 的初始化也被执行了。

好了，到这里差不多该结束了，最后总结一下这些函数调用顺序

【第一步】start_kernel()　　【/init/main.c】

【第二步】setup_arch(&command_line);　【arch/arm/kernel/setup.c】

该函数找到匹配的 machine_desc 但没有调用里面的函数

【第三步】rest_init()->kernel_thread(kernel_init,NULL,CLONE_FSCLONE_SIGHAND)->

regs.ARM_r5 = (unsigned long)kernel_init;

do_fork(flags|CLONE_VM|CLONE_UNTRACED, 0, &regs, 0, NULL, NULL)->

kernel_init()->do_basic_setup()->do_initcalls() 【/init/main.c】

【第四步】customize_machine()　　【/init/main.c】

【第五步】
am335x_evm_init　　【arch/arm/mach-oamp2/board-am335xevm.c】

struct machine_desc 结构体的其他各个成员函数在不同时期被调用！这个以后再分析

总结：明确了 汇编过来的内核第一个 C 函数.start_kernel()

明确了 start_kernel()函数中对于函数的启动顺序是依据什么来安排的？

明确了 嵌入式的 BSP 代码是什么时候初始化的？成员变量又是什么时候被调用执行的？

明确了这些对我们的开发工作有什么帮助呢？ 这个在下一章节中进行讲解。

【2】OMAP335X-内核 BSP 之资源注册那些事.

分类： linux-内核篇 linux-驱动篇 2013-11-26 09:12 802 人阅读 评论(0) 收藏 举报

PC 系统 ：ubuntu10.04

MPU 平台：OAMP3352

内核版本：3.2.0

声明：我讲解的范畴是从内核解压以后经过汇编代码执行最后跳到第一个 C 代码这个点开始讲解，一直讲到文件系统被正确的挂载起来，用户可以正常登入！至于之前的解压缩内核、汇编启动代码我会以后另开文章讲解。

目标：本文想阐述清楚 OAMP335X 这个平台的 BSP 部分的设备注册过程。

第一步：

在上一讲中介绍了 BSP 的启动过程，明确了这个启动过程对我们有什么样的帮助呢？

第一点：有利于我们的调试，对于内核起不来的开发者就有思路了，至少可以作为一个参考思路。

第二点：让我们写驱动代码或系统工程师知道但前哪些设备的资源是被注册进内核的，让我们知道每种片内资源注册的先后顺序情况。

第三点：让我们更容易的找到自己想要的代码。

在上一节中我们知道在板级代码的初始化中，是有先后的初始化顺序的，那么我们看看在 AM3352 上面 BSP 代码中先后初始化了哪些设备。

第一级的初始化：pure_initcall（）

没有

第二级的初始化：core_initcall（）

omap_hwmod.c: core_initcall(omap_hwmod_setup_all);

omap_device.c: core_initcall(omap_device_init);

serial.c: core_initcall(omap_serial_early_init);

第三级的初始化：core_initcall_sync（）

没有

第四级的初始化：postcore_initcall（）

devices.c: postcore_initcall(omap3_l3_init);

devices.c: postcore_initcall(omap4_l3_init);

pm.c: postcore_initcall(omap2_common_pm_init);

gpio.c: postcore_initcall(omap2_gpio_init);

第五级的初始化：postcore_initcall_sync（）

omap_l3_smx.c:postcore_initcall_sync(omap3_l3_init);

第六级的初始化：arch_initcall（）

clock3xxx.c:arch_initcall(omap3xxx_clk_arch_init);

devices.c:arch_initcall(omap2_init_devices);

dma.c:arch_initcall(omap2_system_dma_init);

pm-debug.c:arch_initcall(pm_dbg_init);

timer.c:arch_initcall(omap2_dm_timer_init);

devices.c:arch_initcall(omap_init_devices);

fb.c:arch_initcall(omap_init_fb);

dma.c:arch_initcall(omap_system_dma_init);

第七级的初始化：arch_initcall_sync（）

没有

第八级的初始化：subsys_initcall（）

devices.c:subsys_initcall(omap_init_wdt);

prm2xxx_3xxx.c: subsys_initcall(omap3xxx_prcm_init);

i2c.c:subsys_initcall(omap_register_i2c_bus_cmdline);

mailbox.c:subsys_initcall(omap_mbox_init);

第九级的初始化：subsys_initcall_sync（）

没有

第十级的初始化：fs_initcall（）

kernel/setup.c:fs_initcall(proc_cpu_init);

mm/dma-mapping.c:fs_initcall(dma_debug_do_init);

mm/alignment.c:fs_initcall(alignment_init);

第十一级的初始化：fs_initcall_sync（）

没有

第十二级的初始化：rootfs_initcall（）

drivers/iommu/intr_remapping.c:rootfs_initcall(ir_dev_scope_init);

init/initramfs.c:rootfs_initcall(populate_rootfs);

init/noinitramfs.c:rootfs_initcall(default_rootfs);

第十三级的初始化：device_initcall（）

arch/arm/mach-omap2/cpuidle33xx.c:device_initcall(am33xx_cpuidle_init);

arch/arm/mach-omap2/opp3xxx_data.c:device_initcall(omap3_opp_init);

arch/arm/mach-omap2/mailbox.c:device_initcall(omap2_mbox_init);

第十四级的初始化：device_initcall_sync（）

没有

第十五级的初始化：late_initcall（）

mach-omap2/pm33xx.c:late_initcall(am33xx_pm_init);

mach-omap2/board-m3352.c:late_initcall(backlight_init);

mach-omap2/mux.c:late_initcall(omap_mux_late_init);

mach-omap2/pm.c:late_initcall(omap2_common_pm_late_init);

plat-omap/clock.c:late_initcall(clk_disable_unused);

plat-omap/clock.c:late_initcall(omap_clk_enable_autoidle_all);

plat-omap/clock.c:late_initcall(clk_debugfs_init);

第十六级的初始化：late_initcall_sync（）

没有

第十七级的初始化：__initcall（）

没有

第十八级的初始化：__exitcall（）

没有

第十九级的初始化：console_initcall（）

没有

第二十级的初始化：security_initcall（）

没有

好了终于结束了一共有 20 级别，我这个只是 arch 下面对应的 20 个级别的代码，要是放整个内核几乎让人崩溃！还好我没崩溃！

好了我们看一下几个和本文相关的几个重要的函数：

omap_hwmod.c: core_initcall(omap_hwmod_setup_all);

omap_device.c: core_initcall(omap_device_init);

serial.c: core_initcall(omap_serial_early_init);

---

devices.c: postcore_initcall(omap3_l3_init);

devices.c: postcore_initcall(omap4_l3_init);

pm.c: postcore_initcall(omap2_common_pm_init);

gpio.c: postcore_initcall(omap2_gpio_init);

---

omap_l3_smx.c:postcore_initcall_sync(omap3_l3_init);

---

clock3xxx.c:arch_initcall(omap3xxx_clk_arch_init);

devices.c:arch_initcall(omap2_init_devices);

dma.c:arch_initcall(omap2_system_dma_init);

pm-debug.c:arch_initcall(pm_dbg_init);

timer.c:arch_initcall(omap2_dm_timer_init);

devices.c:arch_initcall(omap_init_devices);

fb.c:arch_initcall(omap_init_fb);

dma.c:arch_initcall(omap_system_dma_init);

---

devices.c:subsys_initcall(omap_init_wdt);

prm2xxx_3xxx.c: subsys_initcall(omap3xxx_prcm_init);

i2c.c:subsys_initcall(omap_register_i2c_bus_cmdline);

mailbox.c:subsys_initcall(omap_mbox_init);

---

arch/arm/mach-omap2/cpuidle33xx.c:device_initcall(am33xx_cpuidle_init);

arch/arm/mach-omap2/opp3xxx_data.c:device_initcall(omap3_opp_init);

arch/arm/mach-omap2/mailbox.c:device_initcall(omap2_mbox_init);

---

mach-omap2/pm33xx.c:late_initcall(am33xx_pm_init);

mach-omap2/board-m3352.c:late_initcall(backlight_init);

mach-omap2/mux.c:late_initcall(omap_mux_late_init);

mach-omap2/pm.c:late_initcall(omap2_common_pm_late_init);

plat-omap/clock.c:late_initcall(clk_disable_unused);

plat-omap/clock.c:late_initcall(omap_clk_enable_autoidle_all);

plat-omap/clock.c:late_initcall(clk_debugfs_init);

接下来只能老老实实的一部分一部分的分析吧

总结：通过本文的分析 我们至少明确了 BSP 在启动的过程中是先后启动了一系列的初始化函数有核心级的、有体系级的、有系统级的、有设备级等　接下来的文章介绍每一部分。

【3】OMAP335X-内核 BSP 之 hwmod（一）.

分类： linux-驱动篇 linux-内核篇 2013-11-26 14:09 1076 人阅读 评论(0) 收藏 举报

PC 系统　：ubuntu10.04

MPU 平台：OAMP3352

内核版本：3.2.0

声明：我讲解的范畴是从内核解压以后经过汇编代码执行最后跳到第一个 C 代码这个点开始讲解，一直讲到文件系统被正确的挂载起来，用户可以正常登入！至于之前的解压缩内核、汇编启动代码我会以后另开文章讲解。

目标：本文想阐述清楚 OAMP335X 这个平台的 BSP 部分的 hwmod。

start_kernel -->rest_init() -->kernel_init() --> do_basic_setup() -->do_initcalls()

TI 的芯片很多，面对的应用领域也很广，而且更新换代的速度也相对要快点。这样的话就带来一个问题，芯片之间的代码可重用性就显得很重要，若果你一直有开发三星 MPU 的话，你可能就深有感触（S3C6410 的代码里面的一些数据结构竟然会在 S3C2410 的头文件里面，这样的情形在 TI 里面也是屡见不鲜）。所以今天说的这个 hwmod 的这个东西就是基于这种环境下的一个产物。

TI 在 omap_hwmod.c 文件里面有对这个 hwmod 的介绍，我翻译了一下如下所示：

快速的来查看 OMAP SoC 的一个全局概貌的方法是，可以把 SOC 看成是通过互相连接在一起的一些 IP 模块的大集合。而这个 IP 模块包括如 ARM 处理器的设备、音频串行接口、UART 等等。这些设备中的一些，比如像 DSP，是由 TI 自己创建，其他的一些设备是由其他大厂家的制造商来提供的（如 SGX）。在 TI 的架构中，TI 把芯片上的片内设备称为“OMAP 模块”。这些 IP 模块是跨越几个相同 OMAP 版本，无需频繁修订。

这些 OMAP 模块是由不同的互连器连接在一起。大多数模块之间的地址和数据流是通过基于 OCP（一种片内模块资源通信协议）的互连（如 L3 和 L4 总线），

OMAP hwmod 提供一种统一的方式来描述芯片硬件模块并将其融入到芯片的其余部分。此描述可以自动从 TI 生成硬件数据库（mach-omap2/omap_hwmod_33xx_data.c 这个文件有详细的解释）。 OMAP hwmod 提供了一个标准，一致的 API 如复位、使能、睡眠以及禁止这些硬件模块功能。Hwmod 也提供了一种方法用于其他核心代码，如 Linux 设备代码或 OMAP 电源管理或地址空间映射代码或查询硬件数据库。

说说怎么使用这个 hwmod：

驱动不应该直接调用硬件模块功能函数（我也这么认为的，但是太多的封装就受不了，你看 android 做了 TMD 多少封装啊，难怪用过 IOS 以后用 google 的就 你懂的），这个应该由 omap_device 的代码来干这个事情，或是在一些极其特殊的情况下由板级代码来执行。omap_device 的代码功能会使用 omap_hwmod 来建立一个 platform_device 结构体，而这种调用方式特体现了 hwmode 数据和 linux 设备驱动模型之间的通信方式。很多的驱动只可以调用 pm_runtime*()来调用 omap_hwmod 功能函数。

下面看看 OMAP hwmod 的代码在整个内核分层结构中所处的一个位置是是怎么样的：

```
*          +-----------------------------+
*          |     Device driver code      |
*          |     (e.g., drivers/)        |
*          +-----------------------------+
*          |     Linux driver model      |
*          |    (platform_device /       |
*          | platform_driver data/code)  |
*          +-----------------------------+
*          | OMAP core-driver integration |
*          |(arch/arm/mach-omap2/devices.c)|
```

```
*          +------------------------------+
*          |      omap_device code        |
*          | (../plat-omap/omap_device.c)    |
*          +------------------------------+
*  ---->    |    omap_hwmod code/data      |    <-----
*          | (../mach-omap2/omap_hwmod*)   |
*          +------------------------------+
*          | OMAP clock/PRCM/register fns   |
*          | (__raw_{read,write}l, clk*)       |
*          +------------------------------+
*
```

这个层次我认为应该是从下往上的一个逻辑顺序！应该是这样的 .....

设备的驱动不应该直接包含 OMAP 特殊平台的的代码或数据，而只应该包含去如何操作这个 IP 模块的代码，这也是设备驱动的职责所在。这样的话 IP 模块也可以在其他平台上使用（比如达芬奇），做到一个平台无关性。

OMAP 的 hwmod 代码在设备启动过程中也可以进行复位或睡眠这个设备，这样让设备可以处于一个正确的运行状态。

OMAP 硬件模块的状态：

刚一开始时候硬件模块是未知的状态，一旦被注册的话，就处于被注册状态，若果该模块的时钟被得到分配，就进入时钟已初始化状态，最后模块设置完成了就出一个已经初始化状态。

Hwmod 代码也包括：处理 I/O 映射代码功能

总线的吞吐率和模块延时的测试代码

上面零零碎碎的啰嗦了一下这个 hwmod 的是个什么东西，当时 “最好的文档还是代码本身” 这是我一直奉行的观念！

第一步：L3 和 L4

AM3352 把片内的资源之间的相互连接用一种 2 层分级的方式来互联，这两个分级成为 L3 和 L4；

L3：是一个基于 Network-On-Chip(NoC)协议的高性能的互联器，Network-On-Chip(NoC)使用一种内部基于包一样的协议方式来发送和接受数据命令在每个模块之间；当时呢这个基于 NOC 的互联器的外部接口（发送或接受）都是遵循 OCPIP2.2 协议的，至于这个 OCPIP2.2 什么个什么协议的话，请君自己去查查吧。下面是一个 L3 的拓补图结构（根据时钟快慢分成 L3F 和 L3S 两个区域）

其实我们可以看出时钟快的是对应处理数据量大的部分。注意有些模块既可以是主动发送者，也可以是接受者。每一个模块的发生器和接收器都唯一的 ID 号；至于更详细的的 L3 互联器介绍去看芯片文档。

L4：是一种非阻塞的外设互联器，该互联器用于访问一些低带宽和一些分散的物理模块，其总线的架构和内存映射外设如下图所示：

仔细看会发现这个 L4 是嵌入到 L3S 里面的。其具体实现可以参考芯片手册。

好了我们说了一下这个L3和L4的互联器，但是这个和我文本的目标hwmod有什么关系呢？我也觉得没什么关系，非要说有点的话我认为是搞出这个 hwmod 的理由之一吧。也就是说 hwmod 为了符合这个邮件模块而搞出来的吧！

【3】OMAP335X-内核 BSP 之 hwmod（二）

分类： linux-驱动篇 linux-内核篇 2013-11-26 14:13 991 人阅读 评论(0) 收藏 举报

第二步:struct omap_hwmod

这个结构体 TI 的解释是：

integration data for OMAP hardware "modules" (IP blocks)

struct omap_hwmod {

const char *name; //硬件模块的名称

struct omap_hwmod_class *class; //硬件模块属于哪个类（硬件都有类了）

struct omap_device *od; //硬件模块对应的具体设备

struct omap_hwmod_mux_info *mux; //硬件模块的管教复用信息

struct omap_hwmod_irq_info *mpu_irqs; //硬件模块的中断信息

struct omap_hwmod_dma_info *sdma_reqs;//硬件模块的 DMA 信息

struct omap_hwmod_rst_info *rst_lines; //硬件模块的设置信息

union {

struct omap_hwmod_omap2_prcm omap2;//硬件模块特殊的 PRCM 数据

struct omap_hwmod_omap4_prcm omap4;//硬件模块特殊的 PRCM 数据

} prcm;

const char *main_clk;   //OMAP 时钟的名字

struct clk *_clk;   //主时钟

struct omap_hwmod_opt_clk *opt_clks; //其他设备的时钟

char *clkdm_name; //

struct clockdomain *clkdm;

char *vdd_name;//电压的名称

struct omap_hwmod_ocp_if **masters; /* connect to *_IA */ //IA 是指发生器的代理（就是说模块和互联器之前不是直接相连的而是通过这个代理器来收发数据和命令，嗨 TI 搞的好复杂可见 BSP 和硬件的耦合性是多么的紧密！）

struct omap_hwmod_ocp_if **slaves; /* connect to *_TA *///TA 是目标的代理

void *dev_attr;

u32 _sysc_cache;

void __iomem *_mpu_rt_va; //模块 cache 寄存器的起始地址

spinlock_t _lock;

struct list_head node;

u16 flags;

u8 _mpu_port_index;

u8 response_lat;

u8 rst_lines_cnt;

u8 opt_clks_cnt;

u8 masters_cnt;

u8 slaves_cnt;

u8 hwmods_cnt;

u8 _int_flags;

u8 _state; //模块的状态（未知、已注册、已初始化）

u8 _postsetup_state;

}

第三步： omap_hwmod_33xx_data.c

该文件里面定义了所有的片内资源（以 hwmod 形式）；

int __init am33xx_hwmod_init(void)

{

return omap_hwmod_register(am33xx_hwmods); **//注册了所有的片内模块资源并以链表形式链接在一起！omap_hwmod_list**

}

第四步：omap_hwmod.c

这个文件使用于 OMAP2/3/4 平台。

```
static int __init omap_hwmod_setup_all(void)

{

int r;

      if (!mpu_oh) {

pr_err("omap_hwmod: %s: MPU initiator hwmod %s not yet registered\n",

       __func__, MPU_INITIATOR_NAME);

return -EINVAL;

}

//为每一个模块找到自己的寄存器虚拟首地址

r = omap_hwmod_for_each(_populate_mpu_rt_base, NULL);

//为每一个模块初始化时钟

r = omap_hwmod_for_each(_init_clocks, NULL);

WARN(IS_ERR_VALUE(r),

     "omap_hwmod: %s: _init_clocks failed\n", __func__);

//为每一个模块进行配置操作

omap_hwmod_for_each(_setup, NULL);

return 0;

}

core_initcall(omap_hwmod_setup_all);
```

同时这个文件还提供了:

```
/*对一个模块进行分配相应的资源（中断、寄存器地址、内存空间、DMA 资源）*/
```

int omap_hwmod_fill_resources(struct omap_hwmod *oh, struct resource *res)

/*得到一个目标模块寄存器的虚拟首地址*/

void __iomem *omap_hwmod_get_mpu_rt_va(struct omap_hwmod *oh)

/*读或写一个值到某个模块的一个寄存器里面*/

u32 omap_hwmod_read(struct omap_hwmod *oh, u16 reg_offs)

void omap_hwmod_write(u32 v, struct omap_hwmod *oh, u16 reg_offs)

/*通过名字找到一个对应的模块*/

struct omap_hwmod *omap_hwmod_lookup(const char *name)

/*对一个模块进行复位*/

/*对一个模块进行使能*/

/*对一个模块进行睡眠*/

/*对一个模块进行停止*/

/*对一个模块进行使能和禁止时钟*/

/*对一个模块:应许这个模块唤醒张整个系统*/

还有很多的其他函数用于操作这个硬件模块，君可以自己去查看；

总结：前前后后的讲了这么多的 hwmod；

至少明确了：为什么 TI 要高一套 hwmod 的软件机制（硬件使用了这套 L3 和 L4 的硬件模块互联机制）；

至少明确了：hwmod 在 BSP 启动中充当一个什么的角色（一个 hwmod 封装了所有的当前属于这个硬件模块的所有资源，以及对硬件模块的操作功能函数，注意这个是和平台有关性的，所以 omap_device.c（基于 linux 驱动模型的方式）对这些功能函数又进行一次平台无关性的封装，这个下一节讲。）

明确了：真正的 BSP 代码是和硬件的耦合性很大的，这一部分的代码才是水平的体现，你不仅要对整个硬件体系架构很清楚而且还要对每个硬件资源的性能有使用经验，而且还要把这些东西抽象出一套符合平台型的一个软件架构。最后这个架构还要很好的整合到 linux 那套底层 BSP 架构中。而且还要考虑到内核的一系列问题（性能、资源利用、调度、通信协议等等）。这些基本上都是芯片厂商的事情，等我们拿到一块 TI 的评估板 这一切的一切 都已经做的很好了，连顶层的应用程序都在一个 SDK 的包里面提供给用户，所以我们工作中很多的是软件方面事情(至少中国国内的整个大环境是这样的)。