According to linux driver's design, all platform devices are on platform bus->**p->klist_devices** linked-list, and all same type platform drivers

are on platform bus->**p->klist_drivers** linked list.

每当某个device被register到platform bus上时，它都会到platform bus的driver linked-list上去match，看是不是有driver认领该device;

而当某个driver被register到platform bus上时,它都会到platform bus的device link-list上去match，看是不是可以找到device来认领。

由于引入device tree后，现在一般都是device register要先于driver register。

1. create device tree

在setup_arch()中，kernel通过unflatten_device_tree()把在dtb中的所谓FDT(Flattened Device Tree)递归地展开成memory中的一个真正的device tree。该tree上每一个

device node代表了生成一个device的所需要的信息。这时在kernel初始化的前期，内核几乎刚刚开始。

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

{

    const struct machine_desc *mdesc;


    setup_processor();

    mdesc = setup_machine_fdt(__atags_pointer);

    if (!mdesc)

        mdesc = setup_machine_tags(__atags_pointer, __machine_arch_type);
```

```
machine_desc = mdesc;
machine_name = mdesc->name;

if (mdesc->reboot_mode != REBOOT_HARD)
        reboot_mode = mdesc->reboot_mode;

init_mm.start_code = (unsigned long) _text;
init_mm.end_code   = (unsigned long) _etext;
init_mm.end_data   = (unsigned long) _edata;
init_mm.brk        = (unsigned long) _end;

/* populate cmd_line too for later use, preserving boot_command_line */
strlcpy(cmd_line, boot_command_line, COMMAND_LINE_SIZE);
*cmdline_p = cmd_line;

parse_early_param();

early_paging_init(mdesc, lookup_processor_type(read_cpuid_id()));
setup_dma_zone(mdesc);
sanity_check_meminfo();
arm_memblock_init(mdesc);

paging_init(mdesc);
request_standard_resources(mdesc);
```

```
        if (mdesc->restart)
                arm_pm_restart = mdesc->restart;

        unflatten_device_tree();

        arm_dt_init_cpu_maps();
        psci_init();
#ifdef CONFIG_SMP
        if (is_smp()) {
                if (!mdesc->smp_init || !mdesc->smp_init()) {
                        if (psci_smp_available())
                                smp_set_ops(&psci_smp_ops);
                        else if (mdesc->smp)
                                smp_set_ops(mdesc->smp);
                }
                smp_init_cpus();
                smp_build_mpidr_hash();
        }
#endif

        if (!is_smp())
                hyp_mode_check();

        reserve_crashkernel();
```

```
#ifdef CONFIG_MULTI_IRQ_HANDLER
        handle_arch_irq = mdesc->handle_irq;
#endif

#ifdef CONFIG_VT
#if defined(CONFIG_VGA_CONSOLE)
        conswitchp = &vga_con;
#elif defined(CONFIG_DUMMY_CONSOLE)
        conswitchp = &dummy_con;
#endif
#endif

        if (mdesc->init_early)
                mdesc->init_early();
}
```

2. create devices according to the nodes from device tree

in arch/arm/mach-pegmatite/pegmatite.c

```
static void __init pegmatite_dt_init(void)
{
        /* Add devices not supported by device tree */
        platform_add_devices(platform_devices, ARRAY_SIZE(platform_devices));      (1)
```

```
    of_platform_populate(NULL, of_default_bus_match_table, NULL, NULL);          (2)
}
```

(1)

array platform_devices[]中的是还未支持device tree方式的driver，所以用legacy方式创建device.

(2)

of_platform_populate()就是根据memory中的device tree递归地创建device。这些device将被链接在platform bus的klist_devices上。

而pegmatite_dt_init()则是在

```
DT_MACHINE_START(PEGMATITE_DT, "Marvell Pegmatite (Device Tree)")
#ifdef CONFIG_SMP
    .smp            = smp_ops(pegmatite_smp_ops),
#endif
    .init_machine  = pegmatite_dt_init,
    .map_io         = pegmatite_map_io,
    .init_early = pegmatite_init_early,
    .init_irq    = pegmatite_init_irq,
    .init_time  = pegmatite_timer_and_clk_init,
    .restart    = pegmatite_restart,
    .dt_compat     = pegmatite_dt_compat,
#ifdef CONFIG_ZONE_DMA
    .dma_zone_size    = SZ_256M,
```

```
#endif
MACHINE_END

static int __init customize_machine(void)
{
	/*
	 * customizes platform devices, or adds new ones
	 * On DT based machines, we fall back to populating the
	 * machine from the device tree, if no callback is provided,
	 * otherwise we would always need an init_machine callback.
	 */
	if (machine_desc->init_machine)
		machine_desc->init_machine();
#ifdef CONFIG_OF
	else
		of_platform_populate(NULL, of_default_bus_match_table,
					NULL, NULL);
#endif
	return 0;
}
arch_initcall(customize_machine);
```

customize_machine()，也就是of_platform_populate()的运行是在arch_initcall阶段。

```
#define arch_initcall(fn)		__define_initcall(fn, 3)
```

initcall level 3阶段（这已经是kernel进入后期，开始各个driver的初始化）

即在该阶段kernel会create 各个device，并把它们挂在platform bus的device linked-list上。

由于built-in driver的初始化都是在module_init level。

#define module_init(x) __initcall(x);

#define __initcall(fn) device_initcall(fn)

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

即built-in driver的初始化一般在initcall level 6。所以此时platform driver linked-list上应该是空的（除非有些特殊device需要提前初始化，比如gpio的interrupt controller）

3. drivers initialize

在initcall level 6阶段，driver分别regsiter，这时每当一个driver regsiter时，它都会enumerate platform bus的device linked-list上的device,这是driver和device开始match，如果成功，则调用driver的**probe**()。

/**

 * __platform_driver_register - register a driver for platform-level devices

 * @drv: platform driver structure

 * @owner: owning module/driver

```
 */
int __platform_driver_register(struct platform_driver *drv,
                               struct module *owner)
{
        drv->driver.owner = owner;
        drv->driver.bus = &platform_bus_type;
        if (drv->probe)
                drv->driver.probe = platform_drv_probe;
        if (drv->remove)
                drv->driver.remove = platform_drv_remove;
        if (drv->shutdown)
                drv->driver.shutdown = platform_drv_shutdown;

        return driver_register(&drv->driver);
}


/**
 * driver_register - register driver with bus
 * @drv: driver to register
 *
 * We pass off most of the work to the bus_add_driver() call,
 * since most of the things we have to do deal with the bus
 * structures.
 */
```

```
int driver_register(struct device_driver *drv)
{
        int ret;
        struct device_driver *other;

        BUG_ON(!drv->bus->p);

        if ((drv->bus->probe && drv->probe) ||
            (drv->bus->remove && drv->remove) ||
            (drv->bus->shutdown && drv->shutdown))
                printk(KERN_WARNING "Driver '%s' needs updating - please use "
                        "bus_type methods\n", drv->name);

        other = driver_find(drv->name, drv->bus);            (1)
        if (other) {
                printk(KERN_ERR "Error: Driver '%s' is already registered, "
                        "aborting...\n", drv->name);
                return -EBUSY;
        }

        ret = bus_add_driver(drv);                                    (2)
        if (ret)
                return ret;
        ret = driver_add_groups(drv, drv->groups);
        if (ret) {
```

```
            bus_remove_driver(drv);
            return ret;
        }
        kobject_uevent(&drv->p->kobj, KOBJ_ADD);

        return ret;
}
```

(1)

在platform bus的driver linked-list查找，是否该driver已经register

(2)

core funcrion

```
bus_add_driver()
        |
        |
        \|/
driver_attach()
        |
        |
        \|/
bus_for_each_dev()
```

```
bus_for_each_dev(drv->bus, NULL, drv, __driver_attach);


/**
 * bus_for_each_dev - device iterator.
 * @bus: bus type.
 * @start: device to start iterating from.
 * @data: data for the callback.
 * @fn: function to be called for each device.
 *
 * Iterate over @bus's list of devices, and call @fn for each,
 * passing it @data. If @start is not NULL, we use that device to
 * begin iterating from.
 *
 * We check the return of @fn each time. If it returns anything
 * other than 0, we break out and return that value.
 *
 * NOTE: The device that returns a non-zero value is not retained
 * in any way, nor is its refcount incremented. If the caller needs
 * to retain this data, it should do so, and increment the reference
 * count in the supplied callback.
 */
int bus_for_each_dev(struct bus_type *bus, struct device *start,
                     void *data, int (*fn)(struct device *, void *))
{
        struct klist_iter i;
```

```
	struct device *dev;
	int error = 0;

	if (!bus || !bus->p)
		return -EINVAL;

	klist_iter_init_node(&bus->p->klist_devices, &i,
			     (start ? &start->p->knode_bus : NULL));
	while ((dev = next_device(&i)) && !error)          (3)
		error = fn(dev, data);                          (4)
	klist_iter_exit(&i);
	return error;
}
```

(3)

while loop就是enumerate platform bus上的device linked-list，取出某个device来

(4)

把当前regsiter的driver与某个device进行match.

具体的match过程如下

```
static int __driver_attach(struct device *dev, void *data)
{
```

```
struct device_driver *drv = data;

/*
 * Lock device and try to bind to it. We drop the error
 * here and always return 0, because we need to keep trying
 * to bind to devices and some drivers will return an error
 * simply if it didn't support the device.
 *
 * driver_probe_device() will spit a warning if there
 * is an error.
 */

if (!driver_match_device(drv, dev))              (5)
        return 0;

if (dev->parent)       /* Needed for USB */
        device_lock(dev->parent);
device_lock(dev);
if (!dev->driver)
        driver_probe_device(drv, dev);                    (6)
device_unlock(dev);
if (dev->parent)
        device_unlock(dev->parent);

return 0;
```

}

这里的参数data就是当前register的driver。

(5)

match的logic在driver_match_device()中

```
static inline int driver_match_device(struct device_driver *drv,
                                      struct device *dev)
{
        return drv->bus->match ? drv->bus->match(dev, drv) : 1;
}
```

它依赖的是bus的match function。当前bus是platform bus.

in drivers/base/platform.c

```
struct bus_type platform_bus_type = {
        .name           = "platform",
        .dev_groups     = platform_dev_groups,
        .match          = platform_match,                    (7)
        .uevent         = platform_uevent,
        .pm         = &platform_dev_pm_ops,
};
```

(6)

如果device与drive match，那么就要调用driver的probe()

```
/**
 * driver_probe_device - attempt to bind device & driver together
 * @drv: driver to bind a device to
 * @dev: device to try to bind to the driver
 *
 * This function returns -ENODEV if the device is not registered,
 * 1 if the device is bound successfully and 0 otherwise.
 *
 * This function must be called with @dev lock held.  When called for a
 * USB interface, @dev->parent lock must be held as well.
 */
int driver_probe_device(struct device_driver *drv, struct device *dev)
{
	int ret = 0;

	if (!device_is_registered(dev))
		return -ENODEV;

	pr_debug("bus: '%s': %s: matched device %s with driver %s\n",
		 drv->bus->name, __func__, dev_name(dev), drv->name);

	pm_runtime_barrier(dev);
```

```
        ret = really_probe(dev, drv);
        pm_request_idle(dev);

        return ret;
}
```

(7)

就是platform bus的match()

```
/**
 * platform_match - bind platform device to platform driver.
 * @dev: device.
 * @drv: driver.
 *
 * Platform device IDs are assumed to be encoded like this:
 * "<name><instance>", where <name> is a short description of the type of
 * device, like "pci" or "floppy", and <instance> is the enumerated
 * instance of the device, like '0' or '42'.  Driver IDs are simply
 * "<name>".  So, extract the <name> from the platform_device structure,
 * and compare it against the name of the driver. Return whether they match
 * or not.
 */
static int platform_match(struct device *dev, struct device_driver *drv)
{
        struct platform_device *pdev = to_platform_device(dev);
```

```
    struct platform_driver *pdrv = to_platform_driver(drv);

    /* When driver_override is set, only bind to the matching driver */
    if (pdev->driver_override)
        return !strcmp(pdev->driver_override, drv->name);

    /* Attempt an OF style match first */
    if (of_driver_match_device(dev, drv))                (8)
        return 1;

    /* Then try ACPI style match */
    if (acpi_driver_match_device(dev, drv))
        return 1;

    /* Then try to match against the id table */
    if (pdrv->id_table)                                  (9)
        return platform_match_id(pdrv->id_table, pdev) != NULL;

    /* fall-back to driver name match */
    return (strcmp(pdev->name, drv->name) == 0);    (10)
}
```

(8)

就是在这里把定义在driver中的of_match_table.compatible string与dts中的device node的"compatible" property进行比较.

(9)，(10)

这是legacy方式的driver与device match方式，不推荐了。