



Basics of the Linux Device Model

*Resource: Embedded Linux Driver Development – slides by [Free Electrons](#).
(Suggested this is (at least partly) read through...).*

[Source](#) (below)

Device Model – Motivation

The Linux kernel runs on a wide range of architectures and hardware platforms, and therefore needs to maximize the reusability of code between platforms.

- For example, we want the same USB device driver to be usable on a x86 PC, or an ARM platform, even though the USB controllers used on these platforms are different.
- This requires a clean organization of the code, with the device drivers separated from the controller drivers, the hardware description separated from the drivers themselves, etc.
- This is what the Linux kernel Device Model allows, in addition to other advantages...

<< Source: “[Atmel32 Application Note: AVR32 AP7 Linux Kernel Module Application Example](#)” >>

“The Linux Device Model (LDM) is built around the concept of **busses, devices and drivers**.
All devices in the system are << must be >> connected to a bus of some kind.

The bus does not have to be a real one; busses primarily exist to gather similar devices together and coordinate initialization, shutdown and power management.

When a device in the system is found to match a driver, they are **bound** together. The specifics about how to match devices and drivers are **bus-specific**. The PCI bus, for example, compares the PCI Device ID of each device against a table of supported PCI IDs provided by the driver. << USB too works in a pretty much identical manner. >>

The **platform bus**, on the other hand, simply compares the **name** of each device against the name of each driver; if they are the same, the device matches the driver.

Binding a device to a driver involves **calling the driver's probe() function** passing a pointer to the device as a parameter. From this point on, it's the responsibility of the driver to get the device properly initialized and register it with any appropriate subsystems.

Devices that can be hot-plugged must be un-bound from the driver when they are removed from the system. This involves calling the driver's **remove()** function passing a pointer to the device as a parameter. This also happens if the driver is a dynamically loadable module and the module is unloaded. "

The standard driver model convention dictates that device discovery/enumeration is handled outside the driver; drivers in turn provide **probe()** and **remove()** methods. The *probe* method is invoked by the driver core on device discovery; the *remove* method upon device removal. Similarly, drivers are responsible for providing *suspend/resume* methods if they support it, along with other standardized PM (power management) hooks.

Additional Details:

<Documentation/driver-model/overview.txt>

Also, from 2.6 onwards, the **"user-interface" to the LDM - and thus all busses, devices and drivers** currently on the system – is via a pseudo-filesystem similar to proc, called **sysfs**. Sysfs is usually auto-mounted under /sys.

Sysfs is *always enabled* by default in the kernel configuration of 2.6 Linux onwards:

```
# grep CONFIG_SYSFS /boot/config-3.5.0-24-generic
# CONFIG_SYSFS_DEPRECATED is not set
CONFIG_SYSFS=y
#
```

<<

From *Documentation/driver-model/overview.txt* :

...

By virtue of having a complete hierarchical view of all the devices in the system, exporting a complete hierarchical view to userspace becomes relatively easy. This has been accomplished by implementing a special purpose virtual file system named sysfs.

...

Whenever a device is inserted into the tree, a directory is created for it. This directory may be populated at each layer of discovery - the global layer, the bus layer, or the device layer.

The global layer currently creates two files - 'name' and 'power'. The former only reports the name of the device. The latter reports the

current power state of the device. It will also be used to set the current power state.

The bus layer may also create files for the devices it finds while probing the bus. For example, the PCI layer currently creates 'irq' and 'resource' files for each PCI device.

A device-specific driver may also export files in its directory to expose device-specific data or tunable interfaces.


>>

Additional Resources:

- [The Driver-Model Core, Greg K Hartman](#)
 - *Embedded Linux Driver Development* – slides by [Free Electrons](#)
 - [Platform Device and Platform Driver @ Linux](#)
 - [How to write a Platform Device/Driver – ADC Driver using wm97xx codec](#)
-

Platform Devices and Drivers

[Source \(below\)](#)



Discoverable vs. non-discoverable hardware

- ▶ Certain busses have dynamic discoverability features
 - ▶ USB, PCI
 - ▶ Allow to enumerate devices on the bus, query their characteristics, at runtime.
 - ▶ No need to know in advance what's on the bus
- ▶ But many busses do not have such features
 - ▶ Memory-mapped devices inside SoC, I2C, SPI, SDIO, etc.
 - ▶ The system has to know in advance "where" the different devices are located, and their characteristics
 - ▶ Such devices, instead of being dynamically detected, must be statically described in either:
 - ▶ The kernel source code
 - ▶ The *Device Tree*, a hardware description file used on some architectures.

<< *Source: Documentation/driver-model/platform.txt* >>

Platform devices are devices that typically appear as autonomous entities in the system. This includes legacy port-based devices and host bridges to peripheral buses, and most controllers integrated into system-on-chip platforms. What they usually have in common is **direct addressing from a CPU bus**. Rarely, a platform_device will be connected through a segment of some other kind of bus; but its registers will still be directly addressable.

<<

What exactly is a *platform device*?

Take a look at this article on LWN:

[Platform devices and device trees \[LWN\]](#)

In a nutshell:

a platform is a pseudo bus usually used to connect lightweight devices integrated into SoCs, with the kernel's device model. Architecture-specific setup code (in `arch/<your-arch>/<your-platform>/`) adds the platform using `platform_device_add()` (or `platform_add_devices()`).

>>

Platform devices are “cold” devices (not hot-pluggable).

In a nutshell:

In order to have the kernel bind a platform device to a platform driver, setup the data structures – the *platform_device* and the *platform_driver* - to have the **same name** and register them appropriately.

For the **platform device**:

include/linux/platform_device.h

```
struct platform_device {
    const char *name;
    int id;
    bool id_auto;
    struct device dev;
    ...
}
```

Register it with:

```
int platform_device_register(struct platform_device *pdev);
```

The platform devices are usually encoded in the “board files” (C source files under *arch/<arch>/mach-<foo>*) or on more modern kernels, in a device tree source (*dts*) file(s) (seen later).

For the **platform driver**:

include/linux/platform_device.h

```
struct platform_driver {
    int (*probe)(struct platform_device *);
    int (*remove)(struct platform_device *);
    void (*shutdown)(struct platform_device *);
    int (*suspend)(struct platform_device *, pm_message_t state);
    int (*resume)(struct platform_device *);
    struct device_driver driver;
    ...
}
```

```
struct device_driver {
    const char *name;
    struct bus_type *bus;

    struct module *owner;
    ...
}
```

Register it with:

```
#define platform_driver_register(drv) \
    __platform_driver_register(drv, THIS_MODULE)
extern int __platform_driver_register(struct platform_driver *,
    struct module *);
```

<<

Lets take a look at the stuff on the “platform bus” on our system (it's a Lenovo X-220 Ultrabook running Ubuntu Linux 14.04 kernel ver 3.2.21):


```
# tree /sys/bus/platform/
/sys/bus/platform/
├── devices
│   ├── alarmtimer -> ../../../../devices/platform/alarmtimer
│   ├── dock.0 -> ../../../../devices/platform/dock.0
│   ├── dock.1 -> ../../../../devices/platform/dock.1
│   ├── dock.2 -> ../../../../devices/platform/dock.2
│   └── eisa.0 -> ../../../../devices/platform/eisa.0
├── ...
│   └── vboxdrv.0 -> ../../../../devices/platform/vboxdrv.0
└── drivers
    ├── 88pm860x-regulator
    │   ├── bind
    │   ├── uevent
    │   └── unbind
    └── ab8500-debug
        ├── bind
        ├── uevent
        └── unbind
```

--snip--

```
└── unbind
└── drivers_autoprobe
└── drivers_probe
└── uevent
```

>>

[Source \(below\)](#)



ARM code organization in the Linux kernel

- ▶ `arch/arm/{kernel,mm,lib,boot}/`
The core ARM kernel. Contains the code related to the ARM core itself (MMU, interrupts, caches, etc.). Relatively small compared to the SoC-specific code.
- ▶ `arch/arm/mach-<foo>/`
The SoC-specific code, and board-specific code, for a given SoC family (clocks, pinmux, power management, SMP, and more.)
 - ▶ `arch/arm/mach-<foo>/board-<bar>.c.`
The board-specific code.

<<

Recall: “a platform is a **pseudo bus** usually used to connect lightweight devices integrated into

SoCs, with the kernel's device model. Architecture-specific setup code (in arch/<your-arch>/<your-platform>/) adds the platform using `platform_device_add()` .”

A quick experiment: with a little scripting, we can find out (here on the 3.10.6 Linux kernel source tree):

- the *number* of platform devices (in source code) under *arch/arm* :

```
$ find ~/3.10.6/arch/arm/ -iname "*.c" |xargs grep -A1 "struct platform_device.*{" |grep
'\.name' |wc -l
1670
$
```

<< *Fyi, this number has reduced to 995 in 4.5.0 (likely due to more platform devices being migrated to use the DT mechanism)* >>

- the *names* of platform devices (in source code) under *arch/arm* :

```
$ find ~/3.10.6/arch/arm/ -iname "*.c" |xargs grep -A1 "struct platform_device.*{" |grep
'\.name' |sed -e 's/\/home\/kaiwan\/3.10.6\/\\g'
arch/arm/mach-realview/core.c-      .name      = "physmap-flash",
arch/arm/mach-realview/core.c-      .name      = "smc911x",
arch/arm/mach-realview/core.c-      .name      = "isp1760",
--snip--
arch/arm/mach-iop32x/iq31244.c-      .name      = "serial8250",
arch/arm/mach-vexpress/v2m.c-      .name      = "versatile-i2c",
--snip--
arch/arm/mach-iop13xx/tpmi.c-      .name = "iop-tpmi",
arch/arm/mach-iop13xx/tpmi.c-      .name = "iop-tpmi",
arch/arm/plat-samsung/devs.c-      .name      = "samsung-ac97",
arch/arm/plat-samsung/devs.c-      .name      = "s3c24xx-adc",
--snip--
arch/arm/mach-at91/at91sam9rl_devices.c-      .name = "atmel_usart",
arch/arm/mach-at91/board-sam9263ek.c-      .name = "gpio-keys",
arch/arm/mach-at91/board-yl-9200.c-      .name = "physmap-flash",
--snip--
arch/arm/mach-omap2/board-omap3touchbook.c-      .name = "gpio-keys",
arch/arm/mach-omap2/gpmc-nand.c-      .name = "omap2-nand",
arch/arm/mach-integrator/integrator_cp.c-      .name      = "physmap-flash",
arch/arm/mach-integrator/integrator_cp.c-      .name      = "smc91x",
--snip--
arch/arm/mach-ks8695/devices.c-      .name      = "ks8695_ether",
arch/arm/mach-ks8695/devices.c-      .name      = "ks8695_wdt",
$
```

>>

Platform devices are **given a name, used in driver binding**, and a list of resources such as addresses and IRQs.

```
struct platform_device {
    const char    *name;
    u32           id;
    struct device  dev;
    u32           num_resources;
    struct resource    *resource;
};
```

<< *Later/recent version (v4.7):*

```

struct platform_device {
    const char *name;
    int id;
    bool id_auto;
    struct device dev;
    u32 num_resources;
    struct resource *resource;

    const struct platform_device_id *id_entry;
    char *driver_override; /* Driver name to force a match */
    ...
}
>>

```

[Source \(below\)](#)

- ▶ Amongst the non-discoverable devices, a huge family are the devices that are directly part of a system-on-chip: UART controllers, Ethernet controllers, SPI or I2C controllers, graphic or audio devices, etc.
- ▶ In the Linux kernel, a special bus, called the **platform bus** has been created to handle such devices.
- ▶ It supports **platform drivers** that handle **platform devices**.
- ▶ It works like any other bus (USB, PCI), except that devices are enumerated statically instead of being discovered dynamically.

Platform drivers

~~~~~

Platform drivers follow the standard driver model convention, where **discovery/enumeration is handled outside the drivers, and drivers provide probe() and remove() methods**. They support power management and shutdown notifications using the standard conventions.

Kernel ver 4.16.8

```

struct platform_driver {
    int (*probe)(struct platform_device *);
    int (*remove)(struct platform_device *);
    void (*shutdown)(struct platform_device *);
    int (*suspend)(struct platform_device *, pm_message_t state);
    int (*resume)(struct platform_device *);
    struct device_driver driver; << has a 'name' member ; make this and the
platform_device 'name' member match in order to have the platform device and driver bound
together ! >>
    const struct platform_device_id *id_table;
    bool prevent_deferred_probe;
};

```

Note that *probe()* should (in) general verify that the specified device hardware actually exists;



sometimes platform setup code can't be sure. The probing can use device resources, including clocks, and device *platform\_data*.

Platform drivers register themselves the normal way:

```
/*
 * use a macro to avoid include chaining to get THIS_MODULE
 */
#define platform_driver_register(drv) \
    __platform_driver_register(drv, THIS_MODULE)
extern int __platform_driver_register(struct platform_driver *,
                                     struct module *);
```

Or, in common situations where the device is known not to be hot-pluggable, the *probe()* routine can live in an init section to reduce the driver's runtime memory footprint:

```
int platform_driver_probe(struct platform_driver *drv,
                          int (*probe)(struct platform_device *))
```

## Device Enumeration

~~~~~

As a rule, platform specific (and often board-specific) setup code will register platform devices:

```
int platform_device_register(struct platform_device *pdev);

int platform_add_devices(struct platform_device **pdevs, int ndev);
```

The general rule is to register only those devices that actually exist, but in some cases extra devices might be registered. For example, a kernel might be configured to work with an external network adapter that might not be populated on all boards, or likewise to work with an integrated controller that some boards might not hook up to any peripherals.

...

Excellent article: [*Platform devices and device trees, LWN, Jon Corbet*](#)

...

The driver need not know that the device was instantiated out of a device tree rather than from a hard-coded platform device definition.

Life is not always quite that simple, though. Device names appearing in the device tree (in the "**compatible**" property) tend to take a standardized form which does not necessarily match the name given to the driver in the Linux kernel; among other things, device trees really are meant to work with more than one operating system. So it may be desirable to attach specific names to a platform device for use with device trees. The kernel provides an *of_device_id* structure which can be used for this purpose:

```
static const struct of_device_id my_of_ids[] = {
    { .compatible = "long,funky-device-tree-name" },
    { }
};
```

When the platform driver is declared, it stores a pointer to this table in the driver substructure:

```
static struct platform_driver my_driver = {
    /* ... */
    .driver = {
        .name = "my-driver",
        .of_match_table = my_of_ids
    }
};
```

The driver can also declare the ID table as a device table to enable autoloading of the module as the device tree is instantiated:

```
MODULE_DEVICE_TABLE(of, my_of_ids);
```

The one other thing capable of complicating the situation is platform data. Needless to say, the device tree code is unaware of the specific structure used by a given driver for its platform data, so it will be unable to provide that information in that form. On the other hand, the device tree mechanism is equipped to allow the passing of just about any information that the driver may need to know. Making use of that information will require the driver to become a bit more aware of the device tree subsystem, though.

Drivers expecting platform data should check the `dev.platform_data` pointer in the usual way. If there is a non-null value there, the driver has been instantiated in the traditional way and device tree does not enter into the picture; the platform data should be used in the usual way. **If, however, the driver has been instantiated from the device tree code, the `platform_data` pointer will be null, indicating that the information must be acquired from the device tree directly.**

In this case, the driver will find a `device_node` pointer in the platform devices `dev.of_node` field. The various device tree access functions (`of_get_property()`, primarily) can then be used to extract the needed information from the device tree. After that, it's business as usual.

In summary: making platform drivers work with device trees is a relatively straightforward task. It is mostly a matter of getting the right names in place so that the binding between a device tree node and the driver can be made, with a bit of additional work required in cases where platform data is in use. The nice result is that the static `platform_device` declarations can go away, along with the board files that contain them. That should, eventually, allow the removal of a bunch of boilerplate code from the kernel while simultaneously making the kernel more flexible.

...

Example I : Platform Driver : SMSC911x Ethernet Controller on an ARM/Linux embedded system

[From Wikipedia entry on Microchip:](#)

“... As of 2011, Microchip Technology ships over a billion processors every year. In September 2011, Microchip Technology shipped the 10 billionth PIC microcontroller.[\[9\]](#)

In August 2012, Microchip acquired **Standard Microsystems Corporation (SMSC)**.[\[10\]](#) Among SMSC's assets were those it had previously acquired from Symwave, a start-up that specialized in [USB 3.0](#) chips, and two hi-fi [wireless audio](#) companies—Kleer Semiconductor and Wireless Audio IP BV.[\[11\]\[12\]\[13\]](#) ...”



```
total 0
--w----- 1 0      0      4.0K Aug 16 07:25 bind
lrwxrwxrwx 1 0      0      0 Aug 16 07:25 smsc911x -> ../../../../devices/platform/
smc911x/
--w----- 1 0      0      4.0K Aug 16 07:25 uevent
--w----- 1 0      0      4.0K Aug 16 07:25 unbind
ARM #
```

The **SMSC911x** is a **USB-to-Ethernet controller** (ethernet/USB) chip manufactured by [SMSC corporation](#) (now part of [Microchip Corp](#)).

From their website:

“ ... The USB to Ethernet connection takes advantage of the ubiquity and speed of USB, replacing older means of transferring information. USB-based networking technology offers a cost-effective and smart design alternative to traditional PCI/PCI-Express networking solutions due to the flexibility of routing and placement of Ethernet and USB connectivity ports.

SMSC integrates Hi-Speed USB 2.0 and high-performance 10/100 Ethernet in its line of standalone USB to Ethernet controllers and USB to Ethernet Hub controllers. ...”

The Linux design: it's considered a platform device and hence the driver is written as a platform driver; more correctly put, the driver binds itself to the kernel driver core via the platform bus (as no physical bus exists for it).

With a platform device/driver pair, we will have two parts:

1. The platform **device** definition for the SMSC911x chipset, and
2. The platform **driver** for it.

How exactly does one get the location in the source tree of both the platform **device** definition and the platform **driver**?

1. Platform Device

The **platform device definition** is usually found in either:

- the board-specific source files (under *arch/<arch>/...* ; in the case of ARM, usually under *arch/arm/mach-xxx/...*
- (modern) the DTB (device tree blob).

Ignoring the device tree for the time being (see it below!)) lets look for the SMSC911x under *arch/arm/mach-xxx/...* (note: below on **kernel ver 3.10.24**):

```
$ find arch/arm/mach-* -name "*.c" -type f |xargs grep -Hn -A1 "struct platform_device.*={*" |
grep -i "smc911x"
arch/arm/mach-exynos/mach-armlex4210.c:134:static struct platform_device armlex4210_smc911x = {
arch/arm/mach-exynos/mach-armlex4210.c:135-         .name           = "smc911x",
arch/arm/mach-exynos/mach-smdkv310.c:223:static struct platform_device smdkv310_smc911x = {
arch/arm/mach-exynos/mach-smdkv310.c:224-         .name           = "smc911x",
arch/arm/mach-imx/mach-armadillo5x0.c:461-         .name           = "smc911x",
arch/arm/mach-imx/mach-pcm037.c:242-         .name           = "smc911x",
arch/arm/mach-imx/mach-kzm_arm11_01.c:165-         .name           = "smc911x",
arch/arm/mach-imx/mach-mx31lite.c:89:static struct platform_device smc911x_device = {
arch/arm/mach-imx/mach-mx31lite.c:90-         .name           = "smc911x",
```

```

arch/arm/mach-imx/mach-mx31lilly.c-82-      .name      = "smc911x",
arch/arm/mach-imx/3ds_debugboard.c-79-      .name = "smc911x",
arch/arm/mach-pxa/raumfeld.c-311-      .name      = "smc911x",
arch/arm/mach-pxa/csb726.c-240-      .name      = "smc911x",
arch/arm/mach-realview/core.c-97-      .name      = "smc911x",
arch/arm/mach-s3c64xx/mach-smdk6410.c:201:static struct platform_device smdk6410_smc911x = {
arch/arm/mach-s3c64xx/mach-smdk6410.c-202-      .name      = "smc911x",
arch/arm/mach-shmobile/board-marzen.c-84-      .name      = "smc911x",
arch/arm/mach-shmobile/board-kota2.c-83-      .name      = "smc911x",
arch/arm/mach-shmobile/board-kzm9g.c-99-      .name      = "smc911x",
arch/arm/mach-shmobile/board-mackerel.c-310-      .name      = "smc911x",
arch/arm/mach-shmobile/board-ap4evb.c-266-      .name      = "smc911x",
arch/arm/mach-shmobile/board-kzm9d.c-58-      .name = "smc911x",
arch/arm/mach-shmobile/board-ag5evm.c-84-      .name      = "smc911x",
arch/arm/mach-shmobile/board-bonito.c-325-      .name      = "smc911x",
arch/arm/mach-ux500/board-mop500.c-225-      .name      = "smc911x",
arch/arm/mach-vexpress/v2m.c-117-      .name      = "smc911x",
$

```

[Note! On recent kernels (eg. 4.16) the matches are far fewer! In fact, the SMSC911x on the Versatile Express does not show up. Then where is it? In the DT...(see below)].

As we're looking at the ARM Versatile Express platform:

File : arch/arm/mach-vexpress/v2m.c [kernel ver 3.14.23]

```

...
112 static struct platform_device v2m_eth_device = {
113     .name      = "smc911x",
114     .id        = -1,
115     .resource   = v2m_eth_resources,
116     .num_resources = ARRAY_SIZE(v2m_eth_resources),
117     .dev.platform_data = &v2m_eth_config,
118 };
...
348     platform_device_register(&v2m_sysreg_device);
349     platform_device_register(&v2m_pcie_i2c_device);
350     platform_device_register(&v2m_ddc_i2c_device);
351     platform_device_register(&v2m_flash_device);
352     platform_device_register(&v2m_cf_device);
353     platform_device_register(&v2m_eth_device);
354     platform_device_register(&v2m_usb_device);
...
362 MACHINE_START(VEXPRESS, "ARM-Versatile Express")
363     .atag_offset   = 0x100,
364     .smp           = smp_ops(vexpress_smp_ops),
365     .map_io        = v2m_map_io,
366     .init_early    = v2m_init_early,
367     .init_irq      = v2m_init_irq,
368     .init_time     = v2m_timer_init,
369     .init_machine  = v2m_init,
370 MACHINE_END
...
444 DT_MACHINE_START(VEXPRESS_DT, "ARM-Versatile Express") << DT = Device Tree >>
445     .dt_compat     = v2m_dt_match,
446     .smp           = smp_ops(vexpress_smp_ops),
447     .smp_init      = smp_init_ops(vexpress_smp_init_ops),
448     .map_io        = v2m_dt_map_io,
449     .init_early    = v2m_dt_init_early,
450     .init_machine  = v2m_dt_init,
451 MACHINE_END

```

<< -----

Device Tree

What about the **platform device definition** via the **Device Tree**?

Indeed, it is present!

The device tree source file (.dtsi) for the board in question is (probably) present under `arch/arm/boot/dts/`. Looking here reveals the presence of a file “`vexpress-v2m.dtsi`”.

File : `arch/arm/boot/dts/vexpress-v2m.dtsi`

```
/*
 * ARM Ltd. Versatile Express
 *
 * Motherboard Express uATX
 * V2M-P1
 *
 * ...
motherboard {
    model = "V2M-P1";
    arm,hbi = <0x190>;
    arm,vexpress,site = <0>;
    compatible = "arm,vexpress,v2m-p1", "simple-bus";
    ...
    ethernet@3,02000000 {
        compatible = "smc,lan9118", "smc,lan9115";
        reg = <3 0x02000000 0x10000>;
        interrupts = <15>;
        phy-mode = "mii";
        reg-io-width = <4>;
        smc,irq-active-high;
        smc,irq-push-pull;
        vdd33a-supply = <&v2m_fixed_3v3>;
        vddvario-supply = <&v2m_fixed_3v3>;
    };
    ...
}
```

IMPORTANT

The precise syntax and properties – IOW, the *DT bindings* – for this SMSC chip are described in the kernel documentation here:

<https://www.kernel.org/doc/Documentation/devicetree/bindings/net/smc911x.txt>

Generically, all Linux kernel DT bindings:

<https://www.kernel.org/doc/Documentation/devicetree/bindings>

>>

2. Platform Driver

The platform device driver is usually found under the *drivers/* branch of the kernel source tree. In this particular case, as it's an Ethernet device, let's check under *drivers/net* for a platform driver data structure containing the **name 'smc911x'**. This, of course, is **because the platform device and platform driver are bound on the basis of their name !**

```
$ find drivers/net/ -name "*.c" -type f | xargs grep -Hn -A5 "struct platform_driver.*" | grep
"smc911x"
drivers/net/ethernet/smc/smc911x.c:2611:static struct platform_driver smc911x_driver = {
```

```

drivers/net/ethernet/smc/smc911x.c-2612- .probe = smc911x_drv_probe,
drivers/net/ethernet/smc/smc911x.c-2613- .remove = smc911x_drv_remove,
drivers/net/ethernet/smc/smc911x.c-2614- .driver = {
drivers/net/ethernet/smc/smc911x.c-2615-         .name = SMC_CHIPNAME,
drivers/net/ethernet/smc/smc911x.c-2616-         .owner = THIS_MODULE,
$

```

```

drivers/net/ethernet/smc/smc911x.c
#define SMC_CHIPNAME        "smc911x"

```

(Partial) [Driver code](#) walkthrough (based on the mainline 3.10.6 Linux kernel; in [drivers/net/ethernet/smc/smc911x.c](#) ; code snippets shown below not necessarily in line order...):

File : [drivers/net/ethernet/smc/smc911x.c](#)

```

...
#ifdef CONFIG_OF
static const struct of_device_id smc911x_dt_ids[] = {
    { .compatible = "smc,lan9115", },
    { /* sentinel */ }
};
MODULE_DEVICE_TABLE(of, smc911x_dt_ids);
#endif

static struct platform_driver smc911x_driver = {
    .probe = smc911x_drv_probe,
    .remove = smc911x_drv_remove,
    .driver = {
        .name = SMC_CHIPNAME,
<<
#define SMC_CHIPNAME        "smc911x"
>>
        .owner = THIS_MODULE,
        .pm = SMC911X_PM_OPS,
        .of_match_table = of_match_ptr(smc911x_dt_ids),
    },
};

/* Entry point for loading the module */
static int __init smc911x_init_module(void)
{
    SMC_INITIALIZE();
    return platform_driver_register(&smc911x_driver);
}

```

<<
Internals:

platform_driver_register() is a wrapper over __platform_driver_register() :

```

...
537 /**
538  * __platform_driver_register - register a driver for platform-level devices
539  * @drv: platform driver structure
540  * @owner: owning module/driver
541  */
542 int __platform_driver_register(struct platform_driver *drv,
543                               struct module *owner)
544 {
545     drv->driver.owner = owner;
546     drv->driver.bus = &platform_bus_type;
547     if (drv->probe)
548         drv->driver.probe = platform_drv_probe;

```

```
549     if (drv->remove)
550         drv->driver.remove = platform_drv_remove;
551     if (drv->shutdown)
552         drv->driver.shutdown = platform_drv_shutdown;
553
554     return driver_register(&drv->driver);
555 }
556 EXPORT_SYMBOL_GPL(__platform_driver_register);
...
>>
```


<<

Optional / FYI

*Note: The following code snippets are **not of direct relevance** to the platform device / driver model; it shows implementation details of an example platform driver - the SMSC911x ethernet chip, and the code is thus more specific to the implementation details of a Linux network (NIC) driver.*

You can safely skip this section if uninterested..

>>

...

<< The probe routine ... >>

```
static int smsc911x_drv_probe(struct platform_device *pdev)
{
    struct device_node *np = pdev->dev.of_node;
    struct net_device *dev;
    struct smsc911x_data *pdata;
    struct smsc911x_platform_config *config = pdev->dev.platform_data;
    struct resource *res, *irq_res;
    unsigned int intcfg = 0;
    int res_size, irq_flags;

...
    << #define request_mem_region(start,n,name) ... >>
    if (!request_mem_region(res->start, res_size, SMSC_CHIPNAME)) {
        retval = -EBUSY;
        goto out_0;
    }

    << Allocate and sets up an Ethernet device; get memory for your private data >>
    dev = alloc_etherdev(sizeof(struct smsc911x_data));
    if (!dev) {
        retval = -ENOMEM;
        goto out_release_io_1;
    }

    SET_NETDEV_DEV(dev, &pdev->dev);

    pdata = netdev_priv(dev);
    dev->irq = irq_res->start;
    irq_flags = irq_res->flags & IRQF_TRIGGER_MASK;
    << get the address for MMIO >>
    pdata->iobase = ioremap_nocache(res->start, res_size);
...

    /* assume standard, non-shifted, access to HW registers */
    pdata->ops = &standard_smc911x_ops;
...
    retval = smsc911x_init(dev);    << see notes below >>
    if (retval < 0)
        goto out_disable_resources;
...
    spin_unlock_irq(&pdata->mac_lock);
    netdev_info(dev, "MAC Address: %pM\n", dev->dev_addr);
    return 0;
}
}
```

In the smsc911x_init function:

```
/* Initializing private device structures, only called from probe */
static int smsc911x_init(struct net_device *dev)
{
```

```

    struct smsc911x_data *pdata = netdev_priv(dev);
...
<< setup Ethernet network device (appropriate defaults) >>
    ether_setup(dev);
    dev->flags |= IFF_MULTICAST;
<< setup the NAPI poll routine (in effect, the Rx path)
    void netif_napi_add(struct net_device *dev, struct napi_struct *napi,
                        int (*poll)(struct napi_struct *, int), int weight);
>>
    netif_napi_add(dev, &pdata->napi, smsc911x_poll, SMSC_NAPI_WEIGHT);
<< setup the network device operations >>
    dev->netdev_ops = &smc911x_netdev_ops; << see the netdev ops below >>
    dev->ethtool_ops = &smc911x_ethtool_ops;

    return 0;
}

...
static const struct net_device_ops smc911x_netdev_ops = {
    .ndo_open      = smc911x_open,
    .ndo_stop      = smc911x_stop,
    .ndo_start_xmit = smc911x_hard_start_xmit, << Tx entry point >>
    .ndo_get_stats  = smc911x_get_stats,
    .ndo_set_rx_mode = smc911x_set_multicast_list,
    .ndo_do_ioctl   = smc911x_do_ioctl,
    .ndo_change_mtu  = eth_change_mtu,
    .ndo_validate_addr = eth_validate_addr,
    .ndo_set_mac_address = smc911x_set_mac_address,
#ifdef CONFIG_NET_POLL_CONTROLLER
    .ndo_poll_controller = smc911x_poll_controller,
#endif
};

...

```

<< **Optional / FYI** : more on the SMSC911x ethernet network driver's **Rx and Tx paths** >>

```

/* NAPI poll function */
static int smc911x_poll(struct napi_struct *napi, int budget)
{
    struct smsc911x_data *pdata =
        container_of(napi, struct smsc911x_data, napi);
    struct net_device *dev = pdata->dev;
    int npackets = 0;

    while (npackets < budget) {
        unsigned int pktlength;
        unsigned int pktwords;
        struct sk_buff *skb;
        unsigned int rxstat = smc911x_rx_get_rxstatus(pdata);

        if (!rxstat) {
            unsigned int temp;
            /* We processed all packets available. Tell NAPI it can
             * stop polling then re-enable rx interrupts */
            smc911x_reg_write(pdata, INT_STS, INT_STS_RSFL_);
            napi_complete(napi);
            temp = smc911x_reg_read(pdata, INT_EN);
            temp |= INT_EN_RSFL_EN_;
            smc911x_reg_write(pdata, INT_EN, temp);
            break;
        }

        /* Count packet for NAPI scheduling, even if it has an error.
         * Error packets still require cycles to discard */
        npackets++;
    }
}

```

```

...
    << allocate an skbuff for rx on a specific device >>
    skb = netdev_alloc_skb(dev, pktwords << 2);
    ...
    pdata->ops->rx_readfifo(pdata,
        (unsigned int *)skb->data, pktwords);

    /* Align IP on 16B boundary */
    skb_reserve(skb, NET_IP_ALIGN);
    << Insert data at the end of skb buffer and update tail and len field. "This function
    extends the used data area of the buffer. If this would exceed the total buffer size the kernel will
    panic. A pointer to the first byte of the extra data is returned." >>

    skb_put(skb, pktlength - 4);
    skb->protocol = eth_type_trans(skb, dev);
    skb_checksum_none_assert(skb);
    << Pass the received packet up the network protocol stack.
    "netif_receive_skb() is the main receive data processing function. It always succeeds. The buffer
    may be dropped during processing for congestion control or by the protocol layers."
    Context: softirq, interrupts enabled. >>

    netif_receive_skb(skb);

    /* Update counters */
    dev->stats.rx_packets++;
    dev->stats.rx_bytes += (pktlength - 4);
}

/* Return total received packets */
return npackets;
}

...

/* Entry point for transmitting a packet */
    << the 'skb' parameter is the packet to transmit! from net device 'dev' >>
static int smsc911x_hard_start_xmit(struct sk_buff *skb, struct net_device *dev)
{
    << retrieve your private / context data pointer >>
    struct smsc911x_data *pdata = netdev_priv(dev);
    unsigned int freespace;
    unsigned int tx_cmd_a;
    unsigned int tx_cmd_b;
    ...
    pdata->ops->tx_writefifo(pdata, (unsigned int *)bufp, wrsz);
    freespace -= (skb->len + 32);
    skb_tx_timestamp(skb);
    dev_kfree_skb(skb); << free skb allocated by the protocol stack >>
    ...
    return NETDEV_TX_OK;
}
...

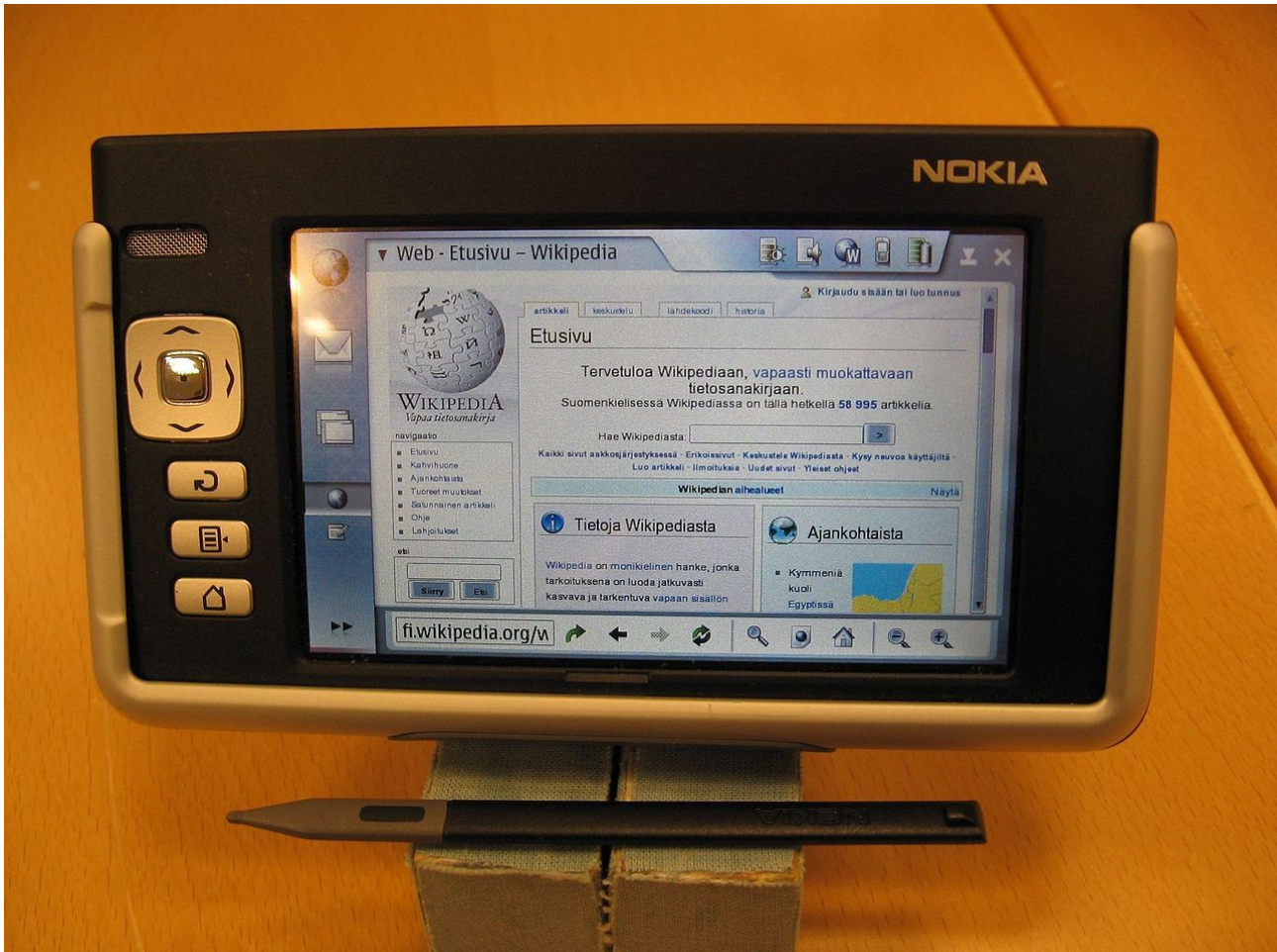
```

Example II – Nokia 770 keypad device

Another example of a simple platform device, and associated platform driver is the **Nokia 770** – this platform driver is meant particularly for the front **keypad device** (seen on the left side of below pic).

As is usual with a platform device/driver pair, we have two parts:

1. The **platform device** definition for the Nokia 770 keypad, and
2. The **platform driver** for it. As the SoC is a TI OMAP(4), the platform driver is a common driver – the OMAP4 keypad driver.



[Pic Source](#)

Kernel ver : 3.10.24

1. The platform device

See [arch/arm/mach-omap1/board-nokia770.c](#) .

File : arch/arm/mach-omap1/board-nokia770.c

<< The machine details are defined (right at the bottom of the file) >>

```
...
281 MACHINE_START(NOKIA770, "Nokia 770")
282     .atag_offset    = 0x100,
283     .map_io         = omap16xx_map_io,
284     .init_early     = omap1_init_early,
285     .init_irq       = omap1_init_irq,
286     .init_machine   = omap_nokia770_init,
287     .init_late      = omap1_init_late,
288     .init_time      = omap1_timer_init,
289     .restart        = omap1_restart,
```

290 MACHINE_END

Continuing with the source of the platform device:

File : arch/arm/mach-omap1/board-nokia770.c

```
...
257 static void __init omap_nokia770_init(void)
258 {
259     /* On Nokia 770, the SleepX signal is masked with an
260      * MPUIO line by default. It has to be unmasked for it
261      * to become functional */
262
263     /* SleepX mask direction */
264     omap_writew((omap_readw(0xffffb5008) & ~2), 0xffffb5008);
265     /* Unmask SleepX signal */
266     omap_writew((omap_readw(0xffffb5004) & ~2), 0xffffb5004);
267
268     platform_add_devices(nokia770_devices, ARRAY_SIZE(nokia770_devices));
269
270     ...
271
272     45 static const unsigned int nokia770_keymap[] = {
273         46     KEY(1, 0, GROUP_0 | KEY_UP),
274         47     KEY(2, 0, GROUP_1 | KEY_F5),
275         48     KEY(0, 1, GROUP_0 | KEY_LEFT),
276         49     KEY(1, 1, GROUP_0 | KEY_ENTER),
277         50     KEY(2, 1, GROUP_0 | KEY_RIGHT),
278         51     KEY(0, 2, GROUP_1 | KEY_ESC),
279         52     KEY(1, 2, GROUP_0 | KEY_DOWN),
280         53     KEY(2, 2, GROUP_1 | KEY_F4),
281         54     KEY(0, 3, GROUP_2 | KEY_F7),
282         55     KEY(1, 3, GROUP_2 | KEY_F8),
283         56     KEY(2, 3, GROUP_2 | KEY_F6),
284     };
285
286     59 static struct resource nokia770_kp_resources[] = {
287         60     [0] = {
288             61         .start = INT_KEYBOARD,
289             62         .end   = INT_KEYBOARD,
290             63         .flags = IORESOURCE_IRQ,
291         },
292     };
293
294     66 static const struct matrix_keymap_data nokia770_keymap_data = {
295         68     .keymap = nokia770_keymap,
296         69     .keymap_size = ARRAY_SIZE(nokia770_keymap),
297     };
298
299     71 static struct omap_kp_platform_data nokia770_kp_data = {
300         73     .rows = 8,
301         74     .cols = 8,
302         75     .keymap_data = &nokia770_keymap_data,
303         76     .delay = 4,
304     };
305
306     78 static struct platform_device nokia770_kp_device = {
307         80     .name = "omap-keypad",
308         81     .id = -1,
309         82     .dev = {
310             83         .platform_data = &nokia770_kp_data,
311         },
312         84     .num_resources = ARRAY_SIZE(nokia770_kp_resources),
313         85     .resource = nokia770_kp_resources,
314     };
315
316     87 };
```

```

88
89 static struct platform_device *nokia770_devices[] __initdata = {
90     &nokia770_kp_device,
91 };
...
...

```

2. The platform driver

The corresponding **platform driver** is here :

File `drivers/input/keyboard/omap4-keypad.c` :

```

...
427 #ifdef CONFIG_OF
428 static const struct of_device_id omap_keypad_dt_match[] = {
429     { .compatible = "ti,omap4-keypad" },
430     {} },
431 };
432 MODULE_DEVICE_TABLE(of, omap_keypad_dt_match);
433 #endif
434
435 static struct platform_driver omap4_keypad_driver = {
436     .probe      = omap4_keypad_probe,
437     .remove     = omap4_keypad_remove,
438     .driver      = {
439         .name     = "omap4-keypad",
440         .owner    = THIS_MODULE,
441         .of_match_table = of_match_ptr(omap_keypad_dt_match),
442     },
443 };
444 module_platform_driver(omap4_keypad_driver);
...

```

Several platform devices (besides the Nokia 770 shown above), use this *omap4-keypad* platform driver:

```

$ find arch/arm/ -iname '*.c' |xargs grep -A1 'struct platform_device.*{' | grep "omap-key"
arch/arm/mach-omap1/board-h3.c-         .name      = "omap-keypad",
arch/arm/mach-omap1/board-innovator.c-         .name      = "omap-keypad",
arch/arm/mach-omap1/board-fs-sample.c-         .name      = "omap-keypad",
arch/arm/mach-omap1/board-htcherald.c-         .name      = "omap-keypad",
arch/arm/mach-omap1/board-palmz71.c-         .name      = "omap-keypad",
arch/arm/mach-omap1/board-nokia770.c-         .name      = "omap-keypad",
arch/arm/mach-omap1/board-sx1.c-         .name      = "omap-keypad",
arch/arm/mach-omap1/board-palmtt.c-         .name      = "omap-keypad",
arch/arm/mach-omap1/board-perseus2.c-         .name      = "omap-keypad",
arch/arm/mach-omap1/board-h2.c-         .name      = "omap-keypad",
arch/arm/mach-omap1/board-ams-delta.c-         .name      = "omap-keypad",
arch/arm/mach-omap1/board-palmtt.c-         .name      = "omap-keypad",
arch/arm/mach-omap1/board-osk.c-         .name      = "omap-keypad",
$

```

*The same keypad device mapped via a **Device Tree**:*

File: `arch/arm/boot/dts/omap4.dtsi`


```
...
        keypad: keypad@4a31c000 {
            compatible = "ti,omap4-keypad";
            reg = <0x4a31c000 0x80>;
            interrupts = <0 120 0x4>;
            reg-names = "mpu";
            ti,hwmods = "kbd";
        };
...
```

OMAP Keypads - DT Linux kernel bindings documentation:

<https://www.kernel.org/doc/Documentation/devicetree/bindings/input/omap-keypad.txt>

<<

Also:

the “Simplest possible simple frame-buffer driver, *as a platform device*” : see [drivers/video/simplefb.c](#).

```
...
static const struct of_device_id simplefb_of_match[] = {
    { .compatible = "simple-framebuffer", },
    { },
};
MODULE_DEVICE_TABLE(of, simplefb_of_match);

static struct platform_driver simplefb_driver = {
    .driver = {
        .name = "simple-framebuffer",
        .owner = THIS_MODULE,
        .of_match_table = simplefb_of_match,
    },
    .probe = simplefb_probe,
    .remove = simplefb_remove,
};
module_platform_driver(simplefb_driver);
...
>>
```

<<

The `module_platform_driver()` function is interesting- a wrapper macro over the `module_init()` and `module_exit()` methods! From the source:

```
include/linux/platform\_device.h
...
/* module_platform_driver() - Helper macro for drivers that don't do
 * anything special in module init/exit. This eliminates a lot of
 * boilerplate. Each module may only use this macro once, and
 * calling it replaces module_init() and module_exit()
 */
#define module_platform_driver(__platform_driver) \
    module_driver(__platform_driver, platform_driver_register, \
                  platform_driver_unregister)
...

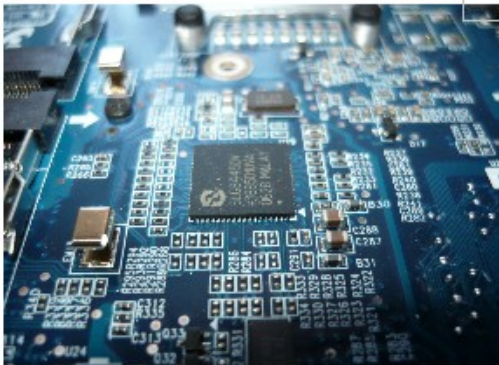
include/linux/device.h
...
```

```

/**
 * module_driver() - Helper macro for drivers that don't do anything
 * special in module init/exit. This eliminates a lot of boilerplate.
 * Each module may only use this macro once, and calling it replaces
 * module_init() and module_exit().
 *
 * @_driver: driver name
 * @_register: register function for this driver type
 * @_unregister: unregister function for this driver type
 * @...: Additional arguments to be passed to __register and __unregister.
 *
 * Use this macro to construct bus specific macros for registering
 * drivers, and do not use it on its own.
 */
#define module_driver(__driver, __register, __unregister, ...) \
static int __init __driver##_init(void) \
{ \
    return __register(&(__driver) , ##__VA_ARGS__); \
} \
module_init(__driver##_init); \
static void __exit __driver##_exit(void) \
{ \
    __unregister(&(__driver) , ##__VA_ARGS__); \
} \
module_exit(__driver##_exit);
...
>>

```

Linux Operating System Specialized

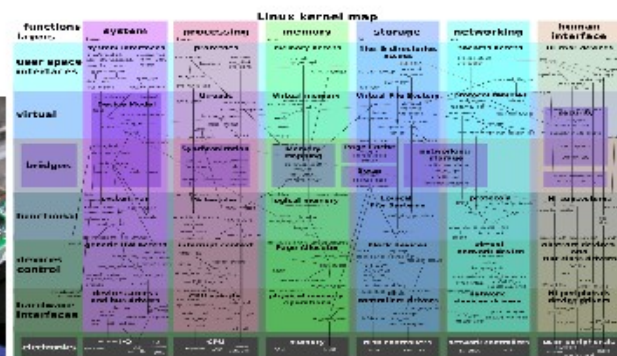
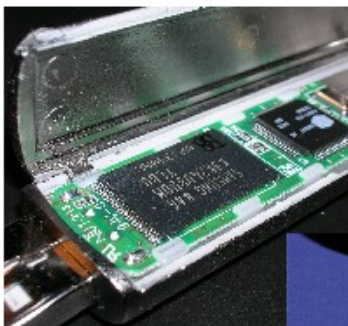


The highest quality Training on:

Linux Fundamentals, CLI and Scripting
Linux Systems Programming
Linux Kernel Internals
Linux Device Drivers
Embedded Linux
Linux Debugging Techniques
New! Linux OS for Technical Managers

Please do visit our website for details:

<http://kaiwantech.in>



<http://kaiwantech.in>
