Linux Device Drivers – Quick Notes / Cheatsheet

- **Quick overview**
  - Linux OS architecture
  - the monolithic model

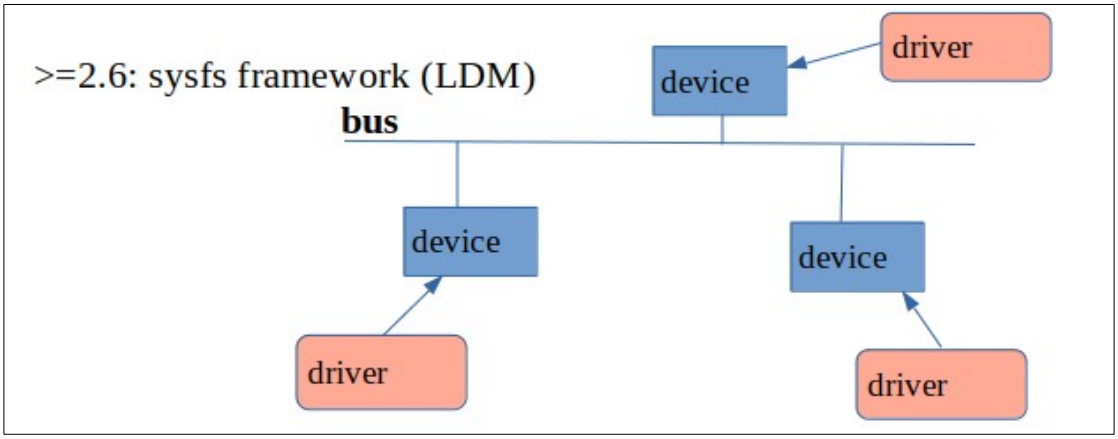
- **Kernel printk and memory (de)allocation APIs**
  - Driver authors are highly encouraged to use the equivalent **dev_*() macros**; in place of the usual pr_*() printk macros. They're the same as the pr_*() except that the extra first parameter is now a pointer to the device structure (that all drivers will possess). The advantage is getting extra information regarding the driver subsystem / device also printed out…
  - Slab allocator
    - devres : the `devm_k[m|z]alloc()` 'resource managed' APIs
    - but not anywhere or everywhere; it's usually appropriate in the probe method
  - page allocator

- **A 'better' Makefile** template for kernel modules (*link*)
  - Learn to use it!

- *Tip:* try and keep the kernel namespace clean; perhaps prefix all globals and functions with a standard prefix string (particular to your driver/module); this is especially true for exported symbols!

Linux Device Model (LDM >= 2.6 Linux)

- **bus | device | driver**



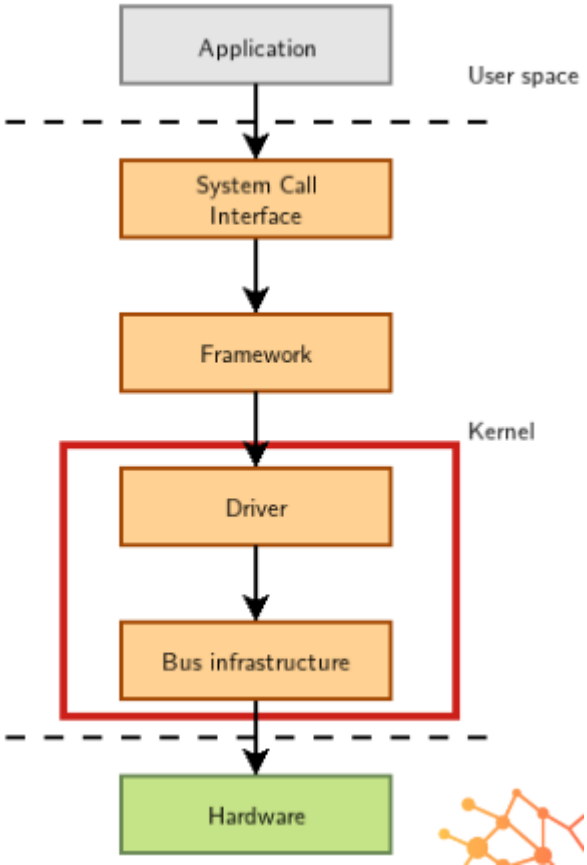
- All devices **must be on a bus**
  - bus driver(s) (depending on kernel config) are auto-loaded by the kernel at boot
  - many types of buses present, kernel has the corresponding bus drivers:

```
~ $ uname -r
5.15.0-50-generic
~ $ ls /sys/bus/
ac97/         dax/          isa/          mmc/          pci_express/  snd_seq/      vme/
acpi/         edac/         ishtp/        mmc_rpmb/     platform/     soundwire/    wmi/
auxiliary/    eisa/         machinecheck/ nd/           pnp/          spi/          workqueue/
cec/          event_source/ mdio_bus/     node/         rapidio/      thunderbolt/  xen/
clockevents/  gpio/         media/        nvmem/        scsi/         typec/        xen-backend/
clocksource/  hdaudio/      mei/          parport/      sdio/         usb/
container/    hid/          memory/       pci/          serial/       usb-serial/
cpu/          i2c/          mipi-dsi/     pci-epf/      serio/        virtio/
~ $
```

Provisioning – device and driver

- The typical modern driver will:
  - in the **init** code : client driver must **register** itself to an appropriate **bus driver** (infrastructure) as it relies upon it
  - in **probe()**, it *could* **register** itself to some existing **kernel framework**

  - Register to **bus** driver via `<foo>_register_driver()` ; where <***foo***> is one of `acpi_button, cnic, cpufreq, gameport, gio, hid, i2c, ide_pci, input, ipmi, mbus, memstick, mmc, parport, pci, platform, pnp, scsi, sdio, serio, spi, tty, usb, vme`



  - (not all are listed above); there are several `foo_register_driver()` APIs; see a list (for 5.4.0) here
  - also, simplest case, (un)register with kernel 'misc' framework with the `misc_[de]register()` APIs

  - a further shorcut: for module init and cleanup (when there's nothing special required there), use:
    `module_<foo>_driver(&driver_structure);`
    ; where <foo> is same as shown above
    - it's a wrapper over the above `<foo>_register_driver()` API!

- The bus driver, upon receiving a registration request from a client driver, **will run it's matching loop!** ... Attempting to match devices on the bus to the driver that just registered; if a match is found (by VID/PID or name), it *binds* the device to the driver by invoking the driver's probe() method

- [check out our *ldm_template* here; YMMV!]

THIS IS VERY IMPORTANT!

- The driver will typically:
  - in the init code: register itself to an appropriate bus driver (infrastructure) as it relies upon it
    - if the *module_<foo>_driver()* macro's used, it automatically registers the module/driver to the appropriate bus driver!
  - in probe(), your driver *could* register itself to some existing kernel framework
- The bus driver, upon receiving a registration request from a client driver, will run it's matching loop! Attempting to match devices on the bus to the driver that just registered; if a match is found (by *VID/PID* or *name*), it *binds* the device to the driver by invoking the driver's *probe()* method.

Examples:

- a serial driver (driving a serial chip which is on the SoC and thus not on a physical bus, and is thus a *platform device*), registers itself with the kernel's UART framework as well as with the platform bus (here, <foo> == uart):

```
static int __init imx_serial_init(void)
{
        ret = uart_register_driver(&imx_reg);
       [...]
        ret = platform_driver_register(&serial_imx_driver);
       [...]
        return 0;
}
```

- an RTC driver (*drivers/rtc/rtc-pcf8563.c*), the chip being on an I2C bus, registers itself with the I2C bus (here, <foo> == i2c):

```
module_i2c_driver(pcf8563_driver); /* internally calls
```

```
                                        i2c_add_driver() -->
    i2c_register_driver() */
```

and, in it's probe method, after initialization succeeds, with the kernel's RTC framework as well:

```
rtc_register_device(&pcf8563->rtc);
```

- a network driver *(drivers/net/usb/rtl8150.c)*, the chip being on a USB bus (via a USB adapter), registers itself with with the USB bus (here, <foo> == usb):

```
module_usb_driver(rtl8150_driver);  /* internally calls
                                       usb_register_driver() */
```

and, in it's probe method, after initialization succeeds, with the kernel's 'net' framework as well (this causes the kernel to construct the new network interface):

```
register_netdev(netdev);
```

- **Enumeration - binding a device to a driver**
  - how the match is performed depends on the bus type
  - for many buses it's based on unique identifiers
    - for PCI and USB, **VID/PID pair(s)** (Vendor ID, Product ID)
  - typically exported by the driver via a device table of devices that it can support (and the `MODULE_DEVICE_TABLE()` macro)
  - On the other hand, platform devices and drivers are bound **by name** (and/or the DT `compatible` property); true for I2C as well

  - Bus Driver
    - the *bus driver* is the critical thing; it has the list of all devices and drivers attached/registered to it
    - whenever a new device is discovered, or a new driver registered to it, it starts the 'matching loop', looking for a matching driver for the device and when it finds one, it 'binds' them
      - a uevent is sent via udev (or mdev) to the [systemd-]udevd userspace daemon (over a netlink socket); udevd will do a rule match and modprobe the driver
      - the driver's *probe* method will be immediately invoked

  [Ref: https://www.kernel.org/doc/html/latest/driver-api/driver-model/binding.html]

- *platform* devices and drivers
  - for devices that are not on a physical bus
  - ... and/or are not discoverable (eg. I2C and SPI 'slave' devices)
  - thus they are placed on a pseudo / virtual 'platform' bus and are driven by a 'platform' driver
    - But, not *all* platform devices are handled by platform drivers (again, I2C/SPI is a good example)
  - "When you register a driver with the *platform_driver_probe()* function, the kernel

> walks through the table of registered platform devices and looks for a match. If any, it calls the matched driver's probe function with the platform data."
> *[source: 'Linux Device Driver Development', John Madieu, Packt]*

<< Please see the superb 'Embedded Linux driver development', by Free Electrons (now named as bootlin), slides pg 10 – 18 ; (also see this PDF) >>

If nothing special needs to be done at driver init/exit, then use the *module_*_driver()* macros for convenience; a lot of boilerplate (init/exit) code can be eliminated!

Eg. `module_usb_driver(), module_pci_driver(), module_i2c_driver(), module_spi_driver(), module_platform_driver(), module_sdio_driver(), module_virtio_driver(), module_phy_driver(), module_i3c_driver(), module_hid_driver(), module_nd_driver(), module_serio_driver(), ... !`

An example:
drivers/usb/usb-skeleton.c

```
static struct usb_driver skel_driver = {
    .name =     "skeleton",
    .probe =    skel_probe,
    .disconnect =   skel_disconnect,
    .suspend =  skel_suspend,
    .resume =   skel_resume,
    .pre_reset =    skel_pre_reset,
    .post_reset =   skel_post_reset,
    .id_table = skel_table,
    .supports_autosuspend = 1,
};

module_usb_driver(skel_driver);
```

I2C drivers in mainline: see (most of?) them with number of them present:

```
find drivers/ -name "*.c" |xargs grep "module_i2c_driver" | wc -l
 (932 on 6.1.25)
```

Linux kernel documentation: *Implementing I2C device drivers*

`container_of()` - to access the parent structure; eg. `to_i2c_client(struct device *dev)` calls `container_of()` to access the device structure of which the i2c client is a member..

Device Tree

***Syntax, etc: look up the* Devicetree Specification *document on* devicetree.org**

`compatible` *property:*

“The compatible property value consists of one or more strings that define the specific programming model for the device. This list of strings should be used by a client program **for device driver selection**. The property value consists of a concatenated list of null terminated strings, **from most specific to most general**. They allow a device to express its compatibility with a family of similar devices, potentially allowing a single device driver to match against several devices.
The recommended format is **"manufacturer,model"**, where manufacturer is a string describing the name of the manufacturer (such as a stock ticker symbol), and model specifies the model number.

Example:
```
compatible = "fsl,mpc8641", "ns16550";
```

In this example, an operating system would first try to locate a device driver that supports *fsl,mpc8641*. If a driver was not found, it would then try to locate a driver that supported the more general *ns16550* device type.”

On x86[_64], the closest equivalent to the DT is the ACPI tables.

<< Sample DTS: arch/arm64/boot/dts/toshiba/tmpv7708.dtsi >>

DT properties

(To do wth the important DT property ‘**reg**’):

Property name: **#address-cells, #size-cells**
Value type: <u32>
Description:
-The #address-cells and #size-cells properties may be used in any device node **that has children** in the devicetree hierarchy and describes *how child device nodes should be addressed.*

`#address-cells` : defines the number of <u32> cells used to encode the address field in a child node’s ***reg*** property.
`#size-cells` : defines the number of <u32> cells used to encode the size field in a child node’s ***reg*** property.

...

If missing, a client program should assume a default value of 2 for #address-cells, and a value of 1 for #size-cells << in effect, assume a default appropriate for 64-bit address, 1 value >>.

"The presence of #size-cell and #address-cell in a given device does not affect the device itself, but its children. In other words, before interpreting *the **reg** property* of a given node, one must know the parent node's #address-cells and #size-cells values. The parent node is free to define whatever addressing scheme is suitable for device sub-nodes (children)."

Property name: **reg**
Property value: <prop-encoded-array> encoded as an arbitrary number of (address, length) pairs.

Usually an address; exact meaning depends on the bus the device is on.

Eg.

```
...
  #address-cells = <2>
  #size-cells = <2>     /* implying the children nodes (below) use
                         * 2 32-bit values to denote both an
                         * address and a size field */
...

memory@0 {
 device_type = "memory";   // deprecated !
 reg = <0x000000000 0x00000000 0x00000000 0x80000000>;
};
```

*Implies physical memory starting @ phy addr 0x0 for length 0x80000000 bytes (**64-bit** quantities)*

```
memory@100000000 {
 device_type = "memory";
 reg = <0x000000001 0x00000000 0x00000001 0x00000000>;
};
```

*Implies physical memory starting @ phy addr 0x100000000 for length 0x100000000 bytes (**64-bit** quantities)*

TIP:
Within the driver, call *of_get_property()* API to fetch the given property!

```
const void *of_get_property(const struct device_node *np, const char *name,
                int *lenp);
```

Eg.

```
...
if (pdev->dev.of_node) {
        prop = of_get_property(pdev->dev.of_node, "aproperty", &len);
        if (!prop) {
            pr_warn("%s: getting DT property failed\n", DRVNAME);
```

```
            return -1;
        }
        pr_info("DT property 'aproperty' = %s (len=%d)\n", prop, len);
}
```

DT Bindings

Within the kernel source tree, at
Documentation/devicetree/bindings

The docs here describe how exactly to specify a DT node for a given type of device / chip / ... Often vendor-specific... so search for your vendor's DT bindings and use them !

Also, the docs typically follow the syntax with examples.

Index of /doc/Documentation/devicetree/bindings/

F.e.:
Documentation/devicetree/bindings/mtd/partitions.txt : documentation on 'Flash partitions in device tree'.

```
...
Examples:


flash@0 {
      partitions {
            compatible = "fixed-partitions";
            #address-cells = <1>;
            #size-cells = <1>;

            partition@0 {
                  label = "u-boot";
                  reg = <0x0000000 0x100000>;
                  read-only;
            };

            uimage@100000 {
                  reg = <0x0100000 0x200000>;
            };
      };
};
...
```

DT examples

- Excellent 'lab' on setting up the TI Beaglebone Black board, and much more, from Bootlin: https://bootlin.com/doc/training/linux-kernel/linux-kernel-labs.pdf

Within this, also see the sections on:
  - Managing I2C buses and devices
  - Configuring pin muxing

- Excellent article on using DT fragments and overlays for the TI BBB!

Using Device Tree Overlays, example on BeagleBone Cape add-on boards
By mopdenacker

In U-Boot:

```
...
load mmc 0:1 0x81000000 zImage
load mmc 0:1 0x82000000 am335x-boneblack-uboot.dtb
fdt addr 0x82000000
load mmc 0:1 0x83000000 overlays/BBORG_RELAY-00A2.dtbo
fdt resize 8192
fdt apply 0x83000000
```

You are now ready to boot your kernel with its updated Device Tree:

```
bootz 0x81000000 - 0x82000000
```

...

A few notes below are from the excellent book 'Linux Device Driver Development', John Madieu, Packt, Oct 2017:

Exploiting the <foo>-names properties:

"Now let us create a fake device node entry to explain that:

```
fake_device {
      compatible = "packt,fake-device";
      reg = <0x4a064000 0x800>, <0x4a064800 0x200>, <0x4a064c00 0x200>;
      reg-names = "config", "ohci", "ehci";

      interrupts = <0 66 IRQ_TYPE_LEVEL_HIGH>, <0 67 IRQ_TYPE_LEVEL_HIGH>;
      interrupt-names = "ohci", "ehci";

      clocks = <&clks IMX6QDL_CLK_UART_IPG>, <&clks
      IMX6QDL_CLK_UART_SERIAL>;
      clock-names = "ipg", "per";
```

```
        dmas = <&sdma 25 4 0>, <&sdma 26 4 0>;
        dma-names = "rx", "tx";
};
```

The code in the driver to extract each named resource is as follows:

```
struct resource *res1, *res2;
res1 = platform_get_resource_byname(pdev, IORESOURCE_MEM, "ohci");
res2 = platform_get_resource_byname(pdev, IORESOURCE_MEM, "config");

struct dma_chan *dma_chan_rx, *dma_chan_tx;
dma_chan_rx = dma_request_slave_channel(&pdev->dev, "rx");
dma_chan_tx = dma_request_slave_channel(&pdev->dev, "tx");

int txirq, rxirq;
txirq = platform_get_irq_byname(pdev, "ohci");
rxirq = platform_get_irq_byname(pdev, "ehci");

structclk *clck_per, *clk_ipg;
clk_ipg = devm_clk_get(&pdev->dev, "ipg");
clk_ipg = devm_clk_get(&pdev->dev, "per");
```

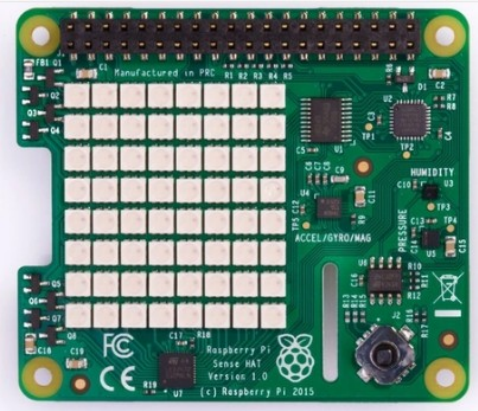
This way, you are sure to map the right name to the right resource, without needing to play with the index anymore."

A perhaps more generic routine to retrieve any DT property value given it's name:

```
const void *of_get_property(const struct device_node *np, const char *name,
                            int *lenp);
```

Usage example here.

Raspberry Pi3 Sense Hat Example:



This DTB Overlay (.dtbo file) is added to detect the devices at boot: */boot/overlays/rpi-sense.dtbo*
The DT now shows all the required sensors under the soc / i2c node!:

```
$ dtc -I fs /proc/device-tree/
...

soc {
      [ ... ]

      i2c@7e804000 {
            compatible = "brcm,bcm2835-i2c";
            clocks = < 0x07 0x14 >;
            status = "okay";
            #address-cells = < 0x01 >;
```

```
            interrupts = < 0x02 0x15 >;
            #size-cells = < 0x00 >;
            phandle = < 0x2a >;
            reg = < 0x7e804000 0x1000 >;  << I2C controller: addr and len >>
            clock-frequency = < 0x186a0 >;
            pinctrl-0 = < 0x17 >;
            pinctrl-names = "default";

            rpi-sense@46 {
                compatible = "rpi,rpi-sense";
                status = "okay";
                keys-int-gpios = < 0x10 0x17 0x01 >;
                reg = < 0x46 >;
            };

            lsm9ds1-accel6a {
                compatible = "st,lsm9ds1-accel";
                status = "okay";
                reg = < 0x6a >;
            };
```

<< eg. refer:
Documentation/devicetree/bindings/iio/imu/st_lsm6dsx.txt
>>

```
            hts221-humid@5f {
                compatible = "st,hts221-humid";
                status = "okay";
                reg = < 0x5f >;
            };

            lps25h-press@5c {
                compatible = "st,lps25h-press";
                status = "okay";
                reg = < 0x5c >;
            };

            lsm9ds1-magn@1c {
                compatible = "st,lsm9ds1-magn";
                status = "okay";
                reg = < 0x1c >;
            };

            ds1307@68 {             << unrelated: our DS1307 RTC chip >>
                compatible = "knb,ks1307";
                status = "okay";
                reg = < 0x68 >;
            };
        };

[ ... ]
```

Device-managed resources – devres (API/structure typically prefixed with 'devm' in codebase)
- 'managed' memory resources; key point: the allocated memory is auto-freed on driver detach!

eg. *devm_kzalloc()*; see https://www.kernel.org/doc/Documentation/driver-model/devres.txt for details.

- DON'T try and replace all allocations with the 'managed' variant APIs
- They are best suited to device driver probe() (or init) methods, ensuring correct initialization, and if it fails, auto-freeing of resources ensuring correct de-init
- first parameter to most devm_*() APIs is `struct device *`
- *Ref*:

"... Please note that managed resources (whether it's memory or some other resource) are meant to be used in code responsible for the probing the device. They are generally a wrong choice for the code used for opening the device, as the device can be closed without being disconnected from the system. Closing the device requires freeing the resources manually, which defeats the purpose of managed resources.

The memory allocated with kzalloc() should be freed with kfree(). The memory allocated with devm_kzalloc() is freed automatically. It can be freed with devm_kfree(), but it's usually a sign that the managed memory allocation is not a good fit for the task. ..."

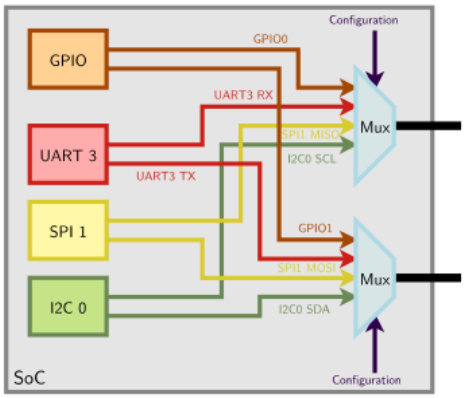
Tip
When using workqueues, the kernel warns if a WQ function took too long... F.e., I saw this warning on a distro-kernel Ubuntu 23.10 box (via dmesg / journalctl):
```
kern  :warn  : [59220.490087] workqueue: delayed_fput hogged CPU for
>10000us 4 times, consider switching to WQ_UNBOUND
```

Pin muxing

See excellent notes here:

- Bootlin Kernel / Drivers course materials
  - linux-kernel-slides PDF
    - Introduction to pin muxing

- Modern SoCs have too many hardware blocks compared to physical pins exposed on the chip package.
- Therefore, pins have to be multiplexed
- Pin configurations are defined in the Device Tree
- Correct pin multiplexing is mandatory to make a device work from an electronic point of view.



- Bootlin linux-kernel-labs PDF:
  - Using the I2C bus *<< practical discussion, useful >>*
    - Find pin muxing configuration information for i2c1
    - Add pinctrl properties to the Device Tree

Ref: *Pin multiplexing (IOMUX) [Digi]*
https://www.digi.com/resources/documentation/digidocs/embedded/dey/3.2/cc6/bsp_r_pin-multiplexing_cc6cc6qp.html

Resources

- *'Some Qemu images to play with'*
  - *includes ARMv6 (Raspberry Pi), AArch64 Debian Stretch, AArch64 Ubuntu, MIPS, PPC64, etc !*
  - *Download link.*

Books

- Linux Device Drivers Development, John Madieu, Packt, 2017
- Linux Driver Development for Embedded Processors, Learn to develop embedded Linux drivers with kernel 4.9 LTS; Alberto Liberal de los Rios, 2nd Ed, 2018
- Linux Device Driver Development Cookbook, Rodolfo Giometti, May 2019, Packt
- Essential Linux Device Drivers, Sreekrishan Venkateswaran, 2008
- Linux Device Drivers, Rubini, Corbet and GK-Hartman, 3rd Ed (LDD3), 2009
- Learning Linux Device Drivers Development [Video], Paul Olushile, Jan 2019

<< End document >>