Kernel and Device Drivers:

In Linux, a driver is always interfacing with:

- *a framework that allows the driver to expose the hardware features in a generic way.*
- *a bus infrastructure, part of the device model, to detect/communicate with the hardware*

Device Model data structures:

-The device model is organized around three main data structures:

- *The struct bus_type structure, which represents one type of bus (USB, PCI, I2C, etc.)*
- *The struct device_driver structure, which represents one driver capable of handling certain devices on a certain bus.*
- *The struct device structure, which represents one device connected to a bus*

-The kernel uses inheritance to create more specialized versions of struct device_driver and struct device for each bus subsystem.

Bus Drivers:

-The first component of the device model is the bus driver:One bus driver for each type of bus: USB, PCI, SPI, MMC, I2C, etc.

- It is responsible for:

- *Registering the bus type (struct bus_type)*
- *Allowing the registration of adapter drivers (USB controllers, I2C adapters, etc.), able to detect the connected devices (if possible), and providing a communication mechanism with the devices*
- *Allowing the registration of device drivers (USB devices, I2C devices, PCI devices, etc.), managing the devices*
- *Matching the device drivers against the devices detected by the adapter drivers.*
- *Provides an API to implement both adapter drivers and device drivers*
- *Defining driver and device specific structures, mainly struct usb_driver and struct usb_interface*

***Example**: USB Bus*

Core infrastructure (bus driver)

- *drivers/usb/core/*
- *struct bus_type is defined in drivers/usb/core/driver.c and registered in drivers/usb/core/usb.c*

Adapter drivers

- *drivers/usb/host/*

Device drivers

- *Everywhere in the kernel tree, classified by their type (Example: drivers/net/usb/).*

Device Identifiers:

- Defines the set of devices that this driver can manage, so that the USB core knows for which devices this driver should be used

*- The MODULE_DEVICE_TABLE() macro allows depmod (run by **make modules_install**) to extract the relationship between device identifiers and drivers, so that drivers can be loaded automatically by udev.*

Instantiation of usb_driver

- struct usb_driver is a structure defined by the USB core. Each USB device driver must instantiate it, and register itself to the USB core using this structure

- This structure inherits from struct device_driver, which is defined by the device model.

** Driver (Un)Registration*

- When the driver is loaded or unloaded, it must register or unregister itself from the USB core

- Done using usb_register() and usb_deregister(), provided by the USB core.

At Initialization:

- The USB adapter driver that corresponds to the USB controller of the system registers itself to the USB core

- The rtl8150 USB device driver registers itself to the USB core

- The USB core now knows the association between the vendor/product IDs of rtl8150 and the struct usb_driver structure of this driver.

When a device is detected

Step_1: a new USB device is detected with ID X:Y

Step_2: USB core looks up the registered ID, and finds the matching driver.

Step_3: The USB core calls the probe() method of the usb_driver structure registered by the rtl8150 driver

Probe Method:

- Invoked for each device bound to a driver

- The probe() method receives as argument a structure describing the device, usually specialized by the bus infrastructure (struct pci_dev, struct usb_interface, etc.)

- This function is responsible for:

- *Initializing the device, mapping I/O memory, registering the interrupt handlers. The bus infrastructure provides methods to get the addresses, interrupt numbers and other device-specific information.*
- *Registering the device to the proper kernel framework, for example the network infrastructure.*

Platform drivers

Plateform devices:

- 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.

Implementation of a Platform Driver

- The driver implements a struct platform_driver structure and registers its driver to the platform driver infrastructure.
- Most drivers actually use the module_platform_driver() macro when they do nothing special in init() and exit() functions.

Platform Device Instantiation: old style

- As platform devices cannot be detected dynamically, they are defined statically

- *By direct instantiation of struct platform_device structures, as done on a few old ARM platforms. Definition done in the board- specific or SoC specific code.*
- *By using a device tree, as done on Power PC (and on most ARM platforms) from which struct platform_device structures are created*

- Example on ARM, where the instantiation was done in: arch/arm/mach-imx/mx1ads.c

The Resource Mechanism

- Each device managed by a particular driver typically uses different hardware resources: addresses for the I/O registers, DMA channels, IRQ lines, etc.
- Such information can be represented using struct resource, and an array of struct resource is associated to a struct platform_device
- Allows a driver to be instantiated for multiple devices functioning similarly, but with different addresses, IRQs, etc.

Using Resources:

- When a struct platform_device was added to the system using platform_add_devices(), the probe() method of the platform driver was called
- This method is responsible for initializing the hardware, registering the device to the proper framework (in our case, the serial driver framework)

Platform_data Mechanism:

- In addition to the well-defined resources, many drivers require driver-specific information for each platform device
- Such information could be passed using the platform_data field of struct device (from which struct platform_device inherits)
*- As it is a void * pointer, it could be used to pass any type of information.*
- Typically, each driver defines a structure to pass information through struct platform_data

!!!!! Device Tree:

- On many embedded architectures, manual instantiation of platform devices was considered to be too verbose and not easily maintainable.
- Such architectures are moving, or have moved, to use the Device Tree.
- It is a tree of nodes that models the hierarchy of devices in the system, from the devices inside the processor to the devices on the board.
- Each node can have a number of properties describing various properties of the devices: addresses, interrupts, clocks, etc.
- At boot time, the kernel is given a compiled version, the Device Tree Blob, which is parsed to instantiate all the devices described in the DT.
-On ARM, they are located in arch/arm/boot/dts/.

Device Tree example:

```
uart0: serial@44e09000 {
        compatible = "ti,omap3-uart";
        ti,hwmods = "uart1";
        clock-frequency = <48000000>;
        reg = <0x44e09000 0x2000>;
        interrupts = <72>;
        status = "disabled";
};
```

- *serial@44e09000 is the node name*
- *uart0 is a label, that can be referred to in other parts of the DT as &uart0*
- *Other lines are properties. Their values are usually strings, list of integers, or references to other nodes.*

Device Tree inheritance

- Each particular hardware platform has its own device tree.

- However, several hardware platforms use the same processor, and often various processors in the same family share a number of similarities.

- To allow this, a device tree file can include another one. The trees described by the including file overlays the tree described by the included file. This can be done:

- *Either by using the /include/ statement provided by the Device Tree language.*
- *Either by using the #include statement, which requires calling the C preprocessor before parsing the Device Tree.*

Device Tree: compatible string

- With the device tree, a device is bound to the corresponding driver using the compatible string.

- The of_match_table field of struct device_driver lists the compatible strings supported by the driver.

Device Tree Resources

- The available resources list will be built up by the kernel at boot time from the device tree, so that you don't need to make any unnecessary lookups to the DT when loading your driver.

- Any additional information will be specific to a driver or the class it belongs to, defining the bindings

Device Tree bindings

- The compatible string and the associated properties define what is called a device tree binding.

- Since the Device Tree is normally part of the kernel ABI, the bindings must remain compatible over time:

- *A new kernel must be capable of using an old Device Tree.*
- *This requires a very careful design of the bindings. They are all reviewed on the devicetree@vger.kernel.org mailing list.*

- A Device Tree binding should contain only a description of the hardware and not configuration.

Sysfs:

- The sysfs virtual filesystem offers a mechanism to export such information to user space

- Used for example by udev to provide automatic module loading, firmware loading, mounting of external media, etc.

- sysfs is usually mounted in /sys

- */sys/bus/ contains the list of buses*
- */sys/devices/ contains the list of devices*
- */sys/class enumerates devices by class (net, input, block...), whatever bus they are connected to.*

I2C subsystems:

What is I2C:

- It is a master/slave bus: only the master can initiate transactions, and slaves can only reply to transactions initiated by masters.

- In a Linux system, the I2C controller embedded in the processor is typically the master, controlling the bus

- Each slave device is identified by an I2C address (you can't have 2 devices with the same address on the same bus). Each transaction initiated by the master contains this address, which allows the relevant slave to recognize that it should reply to this particular transaction.

The I2C bus driver

- Like all bus subsystems, the I2C bus driver is responsible for:

- *Providing an API to implement I2C controller drivers*
- *Providing an API to implement I2C device drivers, in kernel space*
- *Providing an API to implement I2C device drivers, in user space*

- The core of the I2C bus driver is located in drivers/i2c/.

- The I2C controller drivers are located in drivers/i2c/busses/.

- The I2C device drivers are located throughout drivers/, depending on the type of device.

Registering an I2C device driver:

- Like all bus subsystems, the I2C subsystem defines a struct i2c_driver that inherits from struct device_driver, and which must be instantiated and registered by each I2C device driver.
  - As usual, this structure points to the ->probe() and ->remove() functions.
  - It also contains an id_table, used for non-DT based probing of I2C devices.
  - A ->probe_new() function can replace ->probe() when no id_table is provided.
- The i2c_add_driver() and i2c_del_driver() functions are used to register/unregister the driver.
- If the driver doesn't do anything else in its init()/exit() functions, it is advisedto use the module_i2c_driver() macro instead.

Registering an I2C device: non-DT (Non device tree)

- On non-DT platforms, the struct i2c_board_info structure allows to describe how an I2C device is connected to a board.
- Such structures are normally defined with the I2C_BOARD_INFO() helper macro.
- Takes as argument the device name and the slave address of the device on the bus.
- An array of such structures is registered on a per-bus basis using i2c_register_board_info(), when the platform is initialized.

Registering an I2C device, in the DT

- In the Device Tree, the I2C controller device is typically defined in the .dtsi file that describes the processor.
  - Normally defined with status = "disabled".
- At the board/platform level:
  - the I2C controller device is enabled (status = "okay")
  - the I2C bus frequency is defined, using the clock-frequency property.
  - the I2C devices on the bus are described as children of the I2C controller node, where the reg property gives the I2C slave address on the bus.

probe_new() and remove()

-The ->probe_new() function is responsible for initializing the device and registering it in the appropriate kernel framework. It receives as argument:

- A struct i2c_client pointer, which represents the I2C device itself. This structure inherits from struct device.

-Alternatively, the ->probe() function receives as arguments:

- A similar struct i2c_client pointer.
- A struct i2c_device_id pointer, which points to the I2C device ID entry that matched the device that is being probed.

-The ->remove() function is responsible for unregistering the device from the kernel framework and shut it down. It receives as argument:

- The same struct i2c_client pointer that was passed as argument to ->probe_new() or ->probe()

Communicating with the I2C device: raw API

The most basic API to communicate with the I2C device provides functions to either send or receive data:

- int i2c_master_send(const struct i2c_client *client, const char *buf, int count);
  ⇒ Sends the contents of buf to the client.
- int i2c_master_recv(const struct i2c_client *client, char *buf, int count);
  ⇒ Receives count bytes from the client, and store them into buf.

Communicating with the I2C device: message transfer

The message transfer API allows to describe transfers that consists of several messages, with each message being a transaction in one direction:

- int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msgs, int num);
- The struct i2c_adapter pointer can be found by using client->adapter
- The struct i2c_msg structure defines the length, location, and direction of the message.

SMBus calls

- SMBus is a subset of the I2C protocol.
- It defines a standard set of transactions, for example to read or write a register into a device.
- Linux provides SMBus functions that should be used instead of the raw API, if the I2C device supports this standard type of transactions. The driver can then be used on both SMBus and I2C adapters

Kernel frameworks for device drivers

Types of devices:

- Under Linux, there are essentially three types of devices:

- *Network devices: They are represented as network interfaces, visible in user space using ip a*
- *Block devices: They are used to provide user space applications access to raw storage devices (hard disks, USB keys). They are visible to the applications as device files in /dev.*
- *Character devices: They are used to provide user space applications access to all other types of devices (input, sound, graphics, serial, etc.). They are also visible to the applications as device files in /dev.*

Major and minor numbers

- Within the kernel, all block and character devices are identified using a major and a minor number.
- The major number typically indicates the family of the device.
- The minor number typically indicates the number of the device (when there are for example several serial ports)
- Most major and minor numbers are statically allocated, and identical across all Linux systems.

Devices: everything is a file

- A very important UNIX design decision was to represent most system objects as files
- It allows applications to manipulate all system objects with the normal file API (open, read, write, close, etc.). So, devices had to be represented as files to the applications
- This is done through a special artifact called a device file
- It is a special type of file, that associates a file name visible to user space applications to the triplet (type, major, minor) that the kernel understands
- All device files are by convention stored in the /dev directory

Creating device files:

- the device files had to be created manually using the mknod command: mknod /dev/<device> [c|b] major minor
- The devtmpfs virtual filesystem can be mounted on /dev and contains all the devices known to the kernel. The CONFIG_DEVTMPFS_MOUNT kernel configuration option makes the kernel mount it automatically at boot time, except when booting on an initramfs.

Caracter driver:

- From the point of view of an application, a character device is essentially a file.
- The driver of a character device must therefore implement operations that let applications think the device is a file: open, close, read, write, etc.
- In order to achieve this, a character driver must implement the operations described in the struct file_operations structure and register them.
- The Linux filesystem layer will ensure that the driver's operations are called when a user space application makes the corresponding system call.

*int foo_open(struct inode *i, struct file *f)*

- *Called when user space opens the device file.*
- *Only implement this function when you do something special with the device at open() time.*
- *struct inode is a structure that uniquely represents a file in the system (be it a regular file, a directory, a symbolic link, a character or block device)*
- *struct file is a structure created every time a file is opened. Several file structures can point to the same inode structure.*

*int foo_release(struct inode *i, struct file *f)*

- *Called when user space closes the file.*
- *Only implement this function when you do something special with the device at close() time.*

*ssize_t foo_read(struct file *f, char __user *buf, size_t sz, loff_t *off)*

- *Called when user space uses the read() system call on the device.*
- *Must read data from the device, write at most sz bytes to the user space buffer buf, and update the current position in the file off. f is a pointer to the same file structure that was passed in the open() operation*
- *Must return the number of bytes read. 0 is usually interpreted by userspace as the end of the file.*
- *On UNIX, read() operations typically block when there isn't enough data to read from the device.*

*ssize_t foo_write(struct file *f, const char __user *buf, size_t sz, loff_t *off)*

- *Called when user space uses the write() system call on the device*
- *The opposite of read, must read at most sz bytes from buf, write it to the device, update off and return the number of bytes written.*

Exchanging data with user space

- Kernel code isn't allowed to directly access user space memory, using memcpy() or direct pointer dereferencing
- To keep the kernel code portable, secure, and have proper error handling, your driver must use special kernel functions to exchange data with user space.
-A single value

- *get_user(v, p); The kernel variable v gets the value pointed by the user space pointer p*
- *put_user(v, p); The value pointed by the user space pointer p is set to the contents of the kernel variable v.*

-A buffer

- *unsigned long copy_to_user(void __user *to, const void *from, unsigned long n);*
- *unsigned long copy_from_user(void *to, const void __user *from, unsigned long n);*

-The return value must be checked. Zero on success, non-zero on failure.
Zero copy access to user memory:
- Having to copy data to or from an intermediate kernel buffer can become expensive when the amount of data to transfer is large (video).
Zero copy options are possible:

- *mmap(): system call to allow user space to directly access memory mapped I/O space.*
- *get_user_pages(): and related functions to get a mapping to user pages without having to copy them.*

unlocked_ioctl()
*long unlocked_ioctl(struct file *f, unsigned int cmd, unsigned long arg)*
- Allows to extend the driver capabilities beyond the limited read/write API. For example: changing the speed of a serial port, setting video output format, querying a device serial number...

- *cmd is a number identifying the operation to perform*
- *arg is the optional argument passed as third argument of the ioctl() system call.*

The concept of kernel frameworks

Device managed allocations
-The probe() function is typically responsible for allocating a significant number of resources: memory, mapping I/O registers, registering interrupt handlers, etc.
-These resource allocations have to be properly freed:

- *In the probe() function, in case of failure.*
- *In the remove() function.*

This required a lot of failure handling code that was rarely tested.
-To solve this problem, device managed allocations have been introduced.
-The idea is to associate resource allocation with the struct device, and automatically release those resources:

- *When the device disappears*
- *When the device is unbound from the driver*

Memory allocation example:
*- Normally done with kmalloc(size_t, gfp_t), released with kfree(void *)*
- Device managed with ***devm_kmalloc(struct device *, size_t, gfp_t)***
Driver-specific Data Structure
- Each framework defines a structure that a device driver must register to be recognized as a device in this framework

- *struct uart_port for serial ports, struct net_device for network devices, struct fb_info for framebuffers, etc.*

- In addition to this structure, the driver usually needs to store additional information about each device.
- This is typically done

- *By subclassing the appropriate framework structure*
- *By storing a reference to the appropriate framework structure*
- *Or by including your information in the framework structure*

Links between structures
*- The framework structure typically contains a struct device * pointer that the driver must point to the corresponding struct device*

- *It's the relation between the logical device (for example a network interface) and the physical device (for example the USB network adapter)*

*- The device structure also contains a void * pointer that the driver can freely use.*

- *It's often used to link back the device to the higher-level structure from the framework.*

- *It allows, for example, from the struct platform_device structure, to find the structure describing the logical device*

The input subsystem

-The input subsystem takes care of all the input events coming from the human user.
-Initially written to support the USB HID (Human Interface Device) devices, it quickly grew up to handle all kind of inputs (using USB or not): keyboards, mice, joysticks, touchscreens, etc.
-The input subsystem is split in two parts:

- *Device drivers: they talk to the hardware (for example via USB), and provide events (keystrokes, mouse movements, touchscreen coordinates) to the input core*
- *Event handlers: they get events from drivers and pass them where needed via various interfaces (most of the time through evdev)*

Input subsystem API

- An input device is described by a very long struct input_dev structure.
- Before being used it, this structure must be allocated and initialized, typically with:

*struct input_dev *devm_input_allocate_device(struct device *dev);*

- Depending on the type of events that will be generated, the input bit fields evbit and keybit must be configured: For example, for a button we only generate EV_KEY type events, and from these only BTN_0 events code:

- *set_bit(EV_KEY, myinput_dev.evbit);*
- *set_bit(BTN_0, myinput_dev.keybit);*

- set_bit() is an atomic operation allowing to set a particular bit to 1.
- Once the input device is allocated and filled, the function to register it is:

- *int input_register_device(struct input_dev *);*

- The events are sent by the driver to the event handler using:

- *input_event(struct input_dev *dev, unsigned int type, unsigned int code, int value);*

- An event is composed by one or several input data changes (packet of input data changes) such as the button state, the relative or absolute position along an axis, etc..
- After submitting potentially multiple events, the input core must be notified by calling:

- *void input_sync(struct input_dev *dev)*

- The input subsystem provides other wrappers such as input_report_key(), input_report_abs().

Polling input devices:

-The input subsystem provides an API to support simple input devices that do not raise interrupts but have to be periodically scanned or polled to detect changes in their state.
- Setting up polling is done using input_setup_polling():

- *int input_setup_polling(struct input_dev *dev, void (*poll_fn) (struct input_dev *dev));*

- poll_fn is the function that will be called periodically.
- The polling interval can be set using input_set_poll_interval() or input_set_min_poll_interval() and input_set_max_poll_interval()

evdev user space interface

-The main user space interface to input devices is the event interface .Each input device is represented as a /dev/input/event<X> character device
-A user space application can use blocking and non-blocking reads, but also select() (to get notified of events) after opening this device.
-Each read will return struct input_event structures
-A very useful application for input device testing is evtest.

Memory Management

Virtual Memory Organization

- 1GB reserved for kernel-space:

- *Contains kernel code and core data structures, identical in all address spaces*
- *Most memory can be a direct mapping of physical memory at a fixed offset*

- Complete 3GB exclusive mapping available for each user space process

Accessing more physical memory

- Only less than 1GB memory addressable directly through kernel virtual addresses. If more physical memory is present on the platform, part of the memory will not be accessible by kernel space, but can be used by user space
- To allow the kernel to access more physical memory:

- *Change the 3GB/1GB memory split to 2GB/2GB or 1GB/3GB (CONFIG_VMSPLIT_2G or CONFIG_VMSPLIT_1G) ⇒ reduce total user memory available for each process*

User space memory

-New user space memory is allocated either from the already allocated process memory, or using the mmap system call

-Note that memory allocated may not be physically allocated:

- *Kernel uses demand fault paging to allocate the physical page (the physical page is allocated when access to the virtual address generates a page fault)*

Allocators in the Kernel

page allocator:

-Allow to allocate contiguous area of physical page (4K, 8K, 16K, 32K). A page is usually 4K, but can be made greater in some architectures.

- The allocated area is contiguous in the kernel virtual address space, but also maps to also physically contiguous pages. It is allocated in the identity-mapped part of the kernel memory space.

Page Allocator Flags:

-The most common ones are:

GFP_KERNEL

- *Standard kernel memory allocation. The allocation may block in order to find enough available memory. Fine for most needs, except in interrupt handler context.*

GFP_ATOMIC

- *RAM allocated from code which is not allowed to block (interrupt handlers or critical sections). Never blocks, allows to access emergency pools, but can fail if no free memory is readily available.*

GFP_DMA

- *Allocates memory in an area of the physical memory usable for DMA transfers. See our DMA chapter*

SLAB Allocator

- The SLAB allocator allows to create caches, which contain a set of objects of the same size. The object size can be smaller or greater than the page size

-The SLAB allocator takes care of growing or reducing the size of the cache as needed, depending on the number of allocated objects. It uses the page allocator to allocate and free pages.

- SLAB caches are used for data structures that are present in many many instances in the kernel: directory entries, file objects, network packet descriptors, process descriptors, etc.

⇒ There are three different, but API compatible, implementations of a SLAB allocator in the Linux kernel. A particular implementation is chosen at configuration time: SLAB, SLOB, SLUB.

kmalloc Allocator

- The kmalloc allocator is the general purpose memory allocator in the Linux kernel

- For small sizes, it relies on generic SLAB caches, named kmalloc-XXX in /proc/slabinfo

- For larger sizes, it relies on the page allocator

- The allocated area is guaranteed to be physically contiguous

- The allocated area size is rounded up to the size of the smallest SLAB cache in which it can fit (while using the SLAB allocator directly allows to have more flexibility)

kmalloc API:

```
#include <linux/slab.h>
void *kmalloc(size_t size, int flags);
```

- *Allocate size bytes, and return a pointer to the area (virtual address)*
- *size: number of bytes to allocate*
- *flags: same flags as the page allocator*

```
void kfree(const void *objp);
```

- *Free an allocated area*

```
void *kzalloc(size_t size, gfp_t flags);
```

- *Allocates a zero-initialized buffer*

```
void *kcalloc(size_t n, size_t size, gfp_t flags);
```

- *Allocates memory for an array of n elements of size size, and zeroes its contents.*

```
void *krealloc(const void *p, size_t new_size, gfp_t flags);
```

- *Changes the size of the buffer pointed by p to new_size, by reallocating a new buffer and copying the data, unless new_size fits within the alignment of the existing buffer.*

devm_ kmalloc functions

Allocations with automatic freeing when the corresponding device or module is unprobed.

- *void *devm_kmalloc(struct device *dev, size_t size, int flags);*
- *void *devm_kzalloc(struct device *dev, size_t size, int flags);*
- *void *devm_kcalloc(struct device *dev, size_t n, size_t size, gfp_t flags);*
- *void *devm_kfree(struct device *dev, void *p);* ==> *Useful to immediately free an allocated buffer*

vmalloc Allocator

-The vmalloc() allocator can be used to obtain memory zones that are contiguous in the virtual addressing space, but not made out of physically contiguous pages. The requested memory size is rounded up to the next page.

-API in include/linux/vmalloc.h

- *void *vmalloc(unsigned long size);* ⇒ *Returns a virtual address*
- *void vfree(void *addr);*

Kernel memory debugging

-KASAN (Kernel Address Sanitizer)

- *Dynamic memory error detector, to find use-after-free and out-of-bounds bugs.*

-Kmemleak

- *Dynamic checker for memory leaks*

I/O Memory and Port:

Port I/O vs. Memory-Mapped I/O

Memory-Mapped I/O (MMIO)

- *Same address bus to address memory and I/O devices*
- *Access to the I/O devices using regular instructions*
- *Most widely used I/O method across the different architectures supported by Linux*

Port I/O (PIO)

- *Different address spaces for memory and I/O devices*
- *Uses a special class of CPU instructions to access I/O devices*
- *Example on x86: IN and OUT instructions*

Requesting I/O ports

-Tells the kernel which driver is using which I/O ports.

-Allows to prevent other drivers from using the same I/O ports, but is purely voluntary.

*struct resource *request_region(unsigned long start, unsigned long len, char *name);*

⇒ *Tries to reserve the given region and returns NULL if unsuccessful*

void release_region(unsigned long start, unsigned long len);

Accessing I/O ports

-Functions to read/write bytes (b), word (w) and longs (l) to I/O ports:

- *unsigned in[bwl](unsigned long port)*
- *void out[bwl](value, unsigned long port)*

-And the strings variants: often more efficient than the corresponding C loop, if the processor supports such operations!

- *void ins[bwl](unsigned port, void *addr, unsigned long count)*
- *void outs[bwl](unsigned port, void *addr, unsigned long count)*

Requesting I/O memory

- Functions equivalent to request_region() and release_region(), but for I/O memory.

*struct resource *request_mem_region(unsigned long start, unsigned long len, char *name);*

void release_mem_region(unsigned long start, unsigned long len);

Mapping I/O memory in virtual memory

- Load/store instructions work with virtual addresses.To access I/O memory, drivers need to have a virtual address that the processor can handle, because I/O memory is not mapped by default in virtual memory.

-The ioremap function satisfies this need:

#include <asm/io.h>

*void __iomem *ioremap(phys_addr_t phys_addr, unsigned long size);*

*void iounmap(void __iomem *addr);*

Managed API

-Using request_mem_region() and ioremap() in device drivers is now deprecated. You should use the below ”managed” functions instead, which simplify driver coding and error handling:

- *devm_ioremap()*
- *devm_iounmap()*

- *devm_ioremap_resource()*

Accessing MMIO devices

- Directly reading from or writing to addresses returned by ioremap() (pointer dereferencing) may not work on some architectures.

- To do PCI-style, little-endian accesses, conversion being done automatically:

- *unsigned read[bwl](void *addr);*
- *void write[bwl](unsigned val, void *addr);*

- To do raw access, without endianness conversion:

- *unsigned __raw_read[bwl](void *addr);*
- *void __raw_write[bwl](unsigned val, void *addr);*

/dev/mem: *Used to provide user space applications with direct access to physical addresses.*

Misc subsystem:

-There are some devices that do not fit in any of the existing frameworks that the kerel offer it to the user, like Highly customized devices implemented in a FPGA. The drivers for such devices could be implemented directly as raw character drivers (with cdev_init() and cdev_add()).

⇒ So the misc subsystem make this work more easier.

Misc subsystem API

-The misc subsystem API mainly provides two functions, to register and unregister a single misc device:

- *int misc_register(struct miscdevice * misc);*
- *void misc_deregister(struct miscdevice *misc);*

- A misc device is described by a struct miscdevice structure.

- The main fields to be filled in struct miscdevice are:

- *minor, the minor number for the device, or MISC_DYNAMIC_MINOR to get a minor number automatically assigned.*
- *name, name of the device, which will be used to create the device node if devtmpfs is used.*
- *fops, pointer to the same struct file_operations structure that is used for raw character drivers, describing which functions implement the read, write, ioctl, etc. operations.*
- *parent, pointer to the struct device of the underlying "physical" device (platform device, I2C device)*

User space API for misc devices:

- The operations they support in user space depends on the operations the kernel driver implements:

- *The open() and close() system calls to open/close the device.*
- *The read() and write() system calls to read/write to/from the device.*
- *The ioctl() system call to call some driver-specific operations.*