PLATFORM DRIVERS

June 2, 2017

Part I
Enlaces de interes

Todos los ejemplos y situaciones de este documento estan basado en el kernel v4.4.0

Tutoriales sobre embedded linux : http://free-electrons.com/docs/

Linux kernel navegador : http://elixir.free-electrons.com/linux/latest/source

Kconfig doc : https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt

Tutorial sobre linux drivers : http://www.tldp.org/LDP/lkmpg/2.6/html/lkmpg.html

Part II
Diferencia entre módulo y driver

Ambos son en esencia lo mismo y su forma de programarse varia muy poco, solo cuando son inicializados. Un driver esta dentro del kernel y su carga es al iniciar el sistema, sin embargo un módulo puede ser cargado y descargado del kernel segun sea de interes para el usuario. Es importante decir que si programamos un modulo podriamos hacer en el fichero de configuracion hacer que se cargara como un driver, aunque lo haria con la prioridad de carga más baja.

A continuación veremos las prioridades de carga de drivers y modulos:

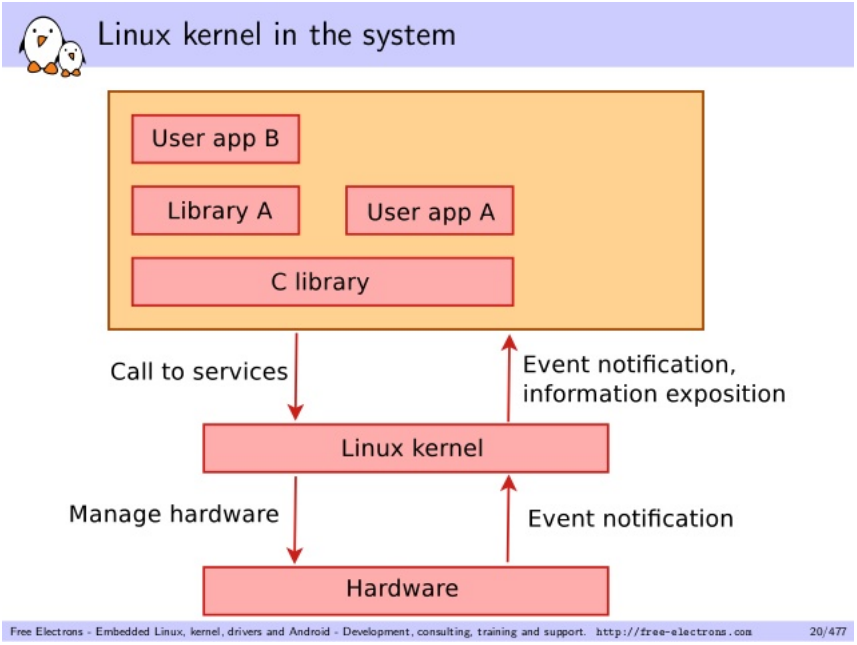
file in: /include/linux/init.h

```
#define pure_initcall(fn) __define_initcall(fn, 0)
#define core_initcall(fn) __define_initcall(fn, 1)
#define core_initcall_sync(fn) __define_initcall(fn, 1s)
#define postcore_initcall(fn) __define_initcall(fn, 2)
#define postcore_initcall_sync(fn) __define_initcall(fn, 2s)
#define arch_initcall(fn) __define_initcall(fn, 3)
#define arch_initcall_sync(fn) __define_initcall(fn, 3s)
#define subsys_initcall(fn) __define_initcall(fn, 4)
#define subsys_initcall_sync(fn) __define_initcall(fn, 4s)
#define fs_initcall(fn) __define_initcall(fn, 5)
#define fs_initcall_sync(fn) __define_initcall(fn, 5s)
#define rootfs_initcall(fn) __define_initcall(fn, rootfs)
#define device_initcall(fn) __define_initcall(fn, 6)
#define device_initcall_sync(fn) __define_initcall(fn, 6s)
#define late_initcall(fn) __define_initcall(fn, 7)
#define late_initcall_sync(fn) __define_initcall(fn, 7s)
#define __initcall(fn) device_initcall(fn)
```

```
file in: /include/linux/module.h
/** * module_init() - driver initialization entry point
* @x: function to be run at kernel boot time or module insertion *
* module_init() will either be called during do_initcalls()
(if * builtin) or at module insertion time (if a module).
There can only * be one per module.
*/
#define module_init(x) __initcall(x);
```

Part III

Estructura Kernel



El kernel de Linux se encarga de manejar los recursos hardware del sistema. Proporciona al user space una API independiente del Hw y la arquitectura.Maneja de forma concurrente el acceso al Hw.

System Calls

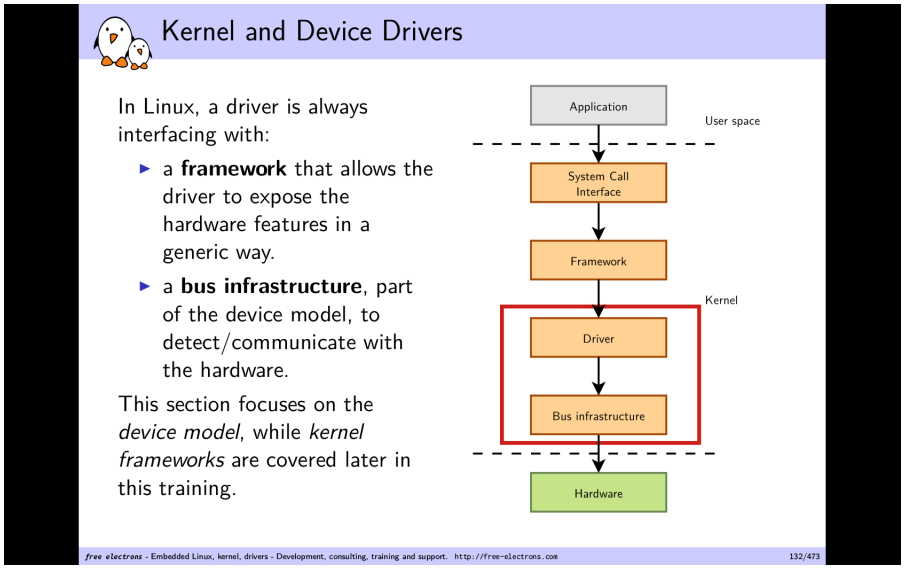
Es la principal interfaz de comunicación entre el kernel y el user space. Linux hace la información del sistema y del kernel a traves de pseudo filesystems tambien denominados virtual filesystems. Los pseudo filesystems permite a las aplicaciones ver directorios y ficheros que no estan almacenados en ningun lugar, son creados y actualizados por el kernel.

Los 2 pseudo filesystems más importantes son:

1. proc : usualmente montado en /proc. Información relativa al sistema operativo (processes, memory management parameters...).
2. sysfs : usualmente montado en /sys. Representación del sistema como un conjunto de buses y dispositivos.

Part IV
Conceptos Kernel Device Drivers

Componentes



1. struct bus_type : representa un tipo de bus (I2C, USB, PCI, ...). Proporcionan una API para la comunicacion con los devices y drivers segun el tipo.

2. struct device_driver : representa un driver que es capaz de manejar ciertos dispositivos conectados a un tipo de bus.

3. struct device : representa un dispositivo conectado a un bus.

Estructuras básicas driver

__init

Cuando el driver es cargado se debe registrar, para ello usamos la funciones vistas anteriormente. Cuando se inicializa usara la función __init, por lo que es donde prepararemos el setup de nuestro driver si requeriese de alguno.

```
static int __init example_driver_init(void) {
  return example_register(&example_struct_driver);
}
subsys_initcall(example_driver_init);
```

__exit

Función usada cuando el driver es descargado del kernel. Algunos drivers no contemplan esta situación ya que son drivers que no deben descargarse del kernel.

```
static void __exit example_driver_exit(void) {
  example_driver_unregister(&example_struct_driver);
}
module_exit(example_driver_exit);
```

probe()

Es un método que recibe como argumento una estructura del dispositivo. Es responsable de inicializar el dispositivo, cuando se conecta al bus. La inicialización es correcta si return 0.

remove()/disconnect(),...

Es la contraparte del método probe(), tambien recibe como argumento una estructura del dispositivo, y maneja cuando el dispositivo es desconectado del bus. La desconexión tiene éxito si return 0.

identificadores

Un driver puede manejar un conjunto de dispositivos que vienen definidos por su vendorId y su productId, para ello se usa la macro MODULE_DEVICE_TABLE() que permite en tiempo de compilación establecer una relación entre los identificadores y los drivers. A continuación un ejemplo de un driver USB.

```
static struct usb_device_id rtl8150_table[] =
{
{ USB_DEVICE(VENDOR_ID_REALTEK, PRODUCT_ID_RTL8150) },
{ USB_DEVICE(VENDOR_ID_MELCO, PRODUCT_ID_LUAKTX) },
{USB_DEVICE(VENDOR_ID_MICRONET, PRODUCT_ID_SP128AR)},
[...]
{}
};
MODULE_DEVICE_TABLE(usb, rtl8150_table);
```

Driver Struct

la instanciación del driver es a traves de la estructura device_driver, de la cual heredan los distintos drivers de cada bus. En el se establecen las llamadas a ciertos metodos como son probe y disconnect , con los metodos definidos por nosotros, junto con la tabla de identificadores y un nombre para el driver. Ademas de otros parametros que son especificos para cada bus y driver.

```
/**  /include/linux/device.h
* struct device_driver - The basic device driver
    structure
* @name:          Name of the device driver.
* @bus: The bus which the device of this driver
    belongs to.
* @owner:         The module owner.
* @mod_name:      Used for built-in modules.
* @suppress_bind_attrs: Disables bind/unbind via sysfs
    .
* @probe_type:  Type of the probe (synchronous or
    asynchronous) to use.
* @of_match_table: The open firmware table.
* @acpi_match_table: The ACPI match table.
* @probe:         Called to query the existence of a
    specific device,
*                 whether this driver can work with it,
    and bind the driver
*                 to a specific device.
* @remove:        Called when the device is removed from
     the system to
*                 unbind a device from this driver.
```

```
* @shutdown:     Called at shut-down time to quiesce
    the device.
* @suspend:      Called to put the device to sleep mode
    . Usually to a
*                low power state.
* @resume:       Called to bring a device from sleep
    mode.
* @groups:       Default attributes that get created by
     the driver core
*                automatically.
* @pm:           Power management operations of the
    device which matched
*                this driver.
* @p:            Driver core's private data, no one
    other than the driver
*                core can touch this.
*
* The device driver-model tracks all of the drivers
    known to the system.
* The main reason for this tracking is to enable the
    driver core to match
* up drivers with new devices. Once drivers are known
    objects within the
* system, however, a number of other things become
    possible. Device drivers
* can export information and configuration variables
    that are independent
* of any specific device.
*/
struct device_driver {
        const char              *name;
        struct bus_type         *bus;
        struct module           *owner;
        const char              *mod_name;      /*
            used for built-in modules */
        bool suppress_bind_attrs;       /* disables
            bind/unbind via sysfs */
        enum probe_type probe_type;
        const struct of_device_id       *
            of_match_table;
        const struct acpi_device_id     *
            acpi_match_table;
        int (*probe) (struct device *dev);
        int (*remove) (struct device *dev);
        void (*shutdown) (struct device *dev);
```

```
        int (*suspend) (struct device *dev,
            pm_message_t state);
        int (*resume) (struct device *dev);
        const struct attribute_group **groups;
        const struct dev_pm_ops *pm;
        struct driver_private *p;
};
```

```
static struct usb_driver rtl8150_driver = {
.name = "rtl8150",
.probe = rtl8150_probe,
.disconnect = rtl8150_disconnect,
.id_table = rtl8150_table,
.suspend = rtl8150_suspend,
.resume = rtl8150_resume
};
```

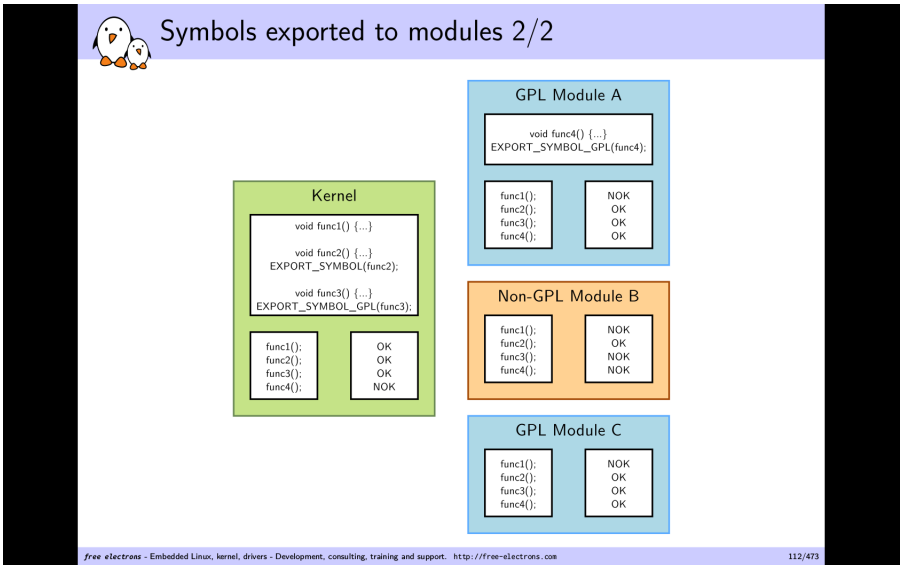
Conceptos

Los driver requieren de cierta información sobre ellos mismos, esta información se puede suministrar a traves de las siguientes macros:

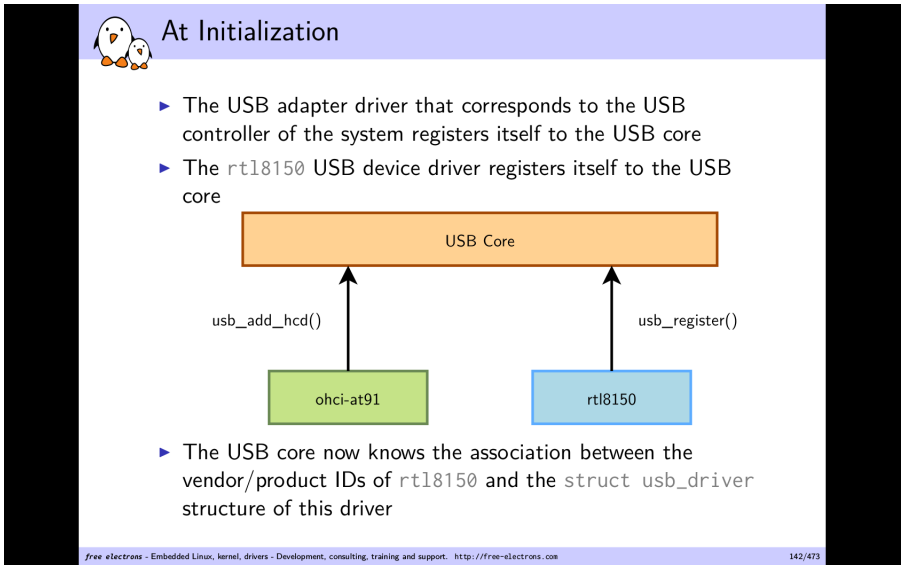
1. MODULE_AUTHOR() : especifica el programador del driver.
2. MODULE_DESCRIPTION() : descripción sobre el driver.
3. MODULE_LICENSE() : licencia del driver, por ejemplo GPL.

Desde un driver/modulo del kernel solo tiene un limitado numero de funciones del kernel que pueden ser llamadas. Las funciones que se pueden usar y ser visibles por otros se hacen a traves de las siguientes macros:

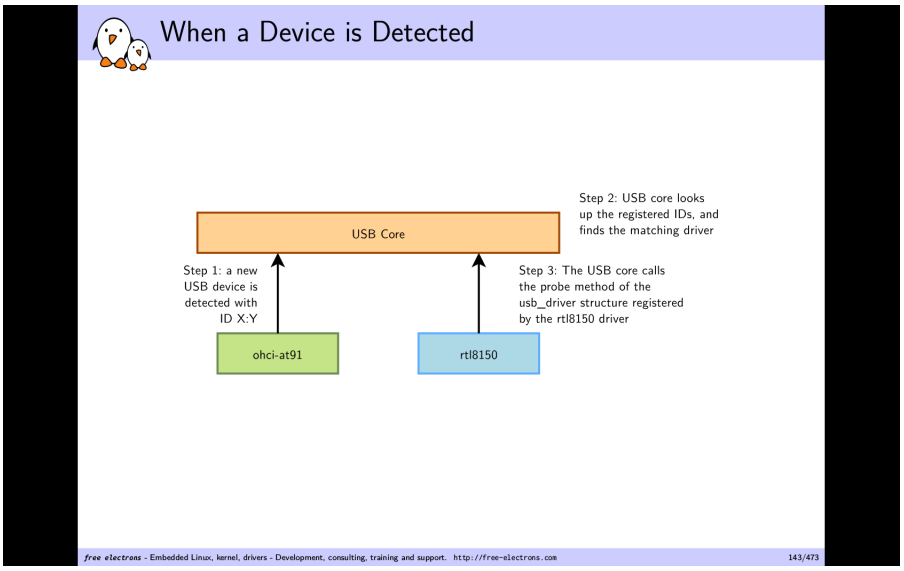
1. EXPORT_SYMBOL(fn/var) : exporta la funcion o variable al resto de modulos/drivers.
2. EXPORT_SYMBOL_GPL(fn/var) : exporta la funcion o variable solo a los modulos/drivers con licencia GPL.



Inicialización



Conexión dispositivo



Part V
Platform Drivers

Introducción

En los embedded systems, los dispositivos no estan normalmente conectados traves de un bus permitiendo asi la enumeración, hotpluggin y dando identificadores unicos para los dispositivos.

Muchos de los dispositivos son parte directa del SoC, sin embargo queremos que esos dispositivos sigan el modelo de driver visto en la sección anterior.

Por lo tanto estos dispositivos en vez de ser detectados dinamicamente sera descritos en el Device Tree.

En el kernel de Linux existe un bus especial denominado platform bus que ha sido creado para manejar este tipo de dispositivos. Tiene asociado platform drivers y platform devices.

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;
        const struct platform_device_id *id_table;
        bool prevent_deferred_probe;
};
```

```
static struct platform_driver serial_imx_driver = {
        .probe = serial_imx_probe,
        .remove = serial_imx_remove,
        .id_table = imx_uart_devtype,
        .driver = {
                .name = "imx-uart",
                .of_match_table = imx_uart_dt_ids,
                .pm = &imx_serial_port_pm_ops,
        },
};
```

Platform Device

Los platform devices no son detectados dinamicamente, por ello son definidos estaticamente en el Device Tree.

A continuación veremos con mas detalle un Device Tree.

Device Tree

La estructura del device tree se basa en un conjunto de nodos que contienen una serie de propiedades, estas propiedades son parejas clave-valor. un nodo su vez puede contener otros nodos. Los dispositivos son representados como nodos.

```
/dts-v1/;
/ {
    node1 {
        a-string-property = "A␣string";
            a-string-list-property = "first␣string",
                "second␣string";
          // hex is implied in byte arrays. no '0x'
              prefix is required
          a-byte-data-property = [01 23 34 56];
          child-node1 {
                  first-child-property;
                second-child-property = <1>;
                a-string-property = "Hello,␣world";
         };
        child-node2 {
        };
    };
   node2 {
        an-empty-property;
        a-cell-property = <1 2 3 4>; /* each number (
            cell) is a uint32 */
        child-node1 {
        };
    };
};
```

La estructura siempre comienza con el nodo raiz cuyo nobre es "/".
existen ciertas propiedades que se deben conocer:

compatible = "manufacter, model"

identifica el dispositivo con el driver que lo maneja. Todo dispositivo requiere de esta propiedad. Se enlaza con el driver a traves del campo of_match_table del device_driver.

```
dev0: exampleDevice {
        compatible = "dte,exampleDriver";
    };
```

```
static const struct of_device_id
    example_driver_of_match[] = {
         { .compatible = "dte,exampleDriver" },
         {},
};
MODULE_DEVICE_TABLE(of, example_driver_of_match);

static struct platform_driver example_driver = {
         .probe = example_driver_probe,
         .remove = example_driver_remove,
         .driver = {
                  .name = DRIVER_NAME,
                  .of_match_table = of_match_ptr(
                     example_driver_of_match),
         },
};
```

nombre nodos

la convencion de los nombres es la siguiente etiqueta: name@address, pero podria ser cualquiera que queramos. Al usar la etiqueta podremos usar el nodo en otras partes del device tree de la siguiente forma &etiqueta.

direccionamiento

para el direccionamiento se usan las siguientes propiedades: reg, #address-cells, #size-cells.

Cada dispositivo contiene la propiedad reg que se forma de la siguiente manera, reg = <add1 len1 add2 len2 ...> que representa el rango de direcciones del dispositivo.

Cada direccion es una lista con elementos de 32 bits denominado cells, la longitud puede ser una lista de cells o vacio.

la longitud de la lista se resuelve con los valores #address-cells y #size-cells del nodo padre más cercano.

Interrupciones

Son necesarias 4 propiedades para describir las interrupciones en un sistema.

1. interrupt-controller : propiedad vacia, se localiza en el nodo cuyo dispositivo recibe las señales de interrupcion, el interrupt Controller.

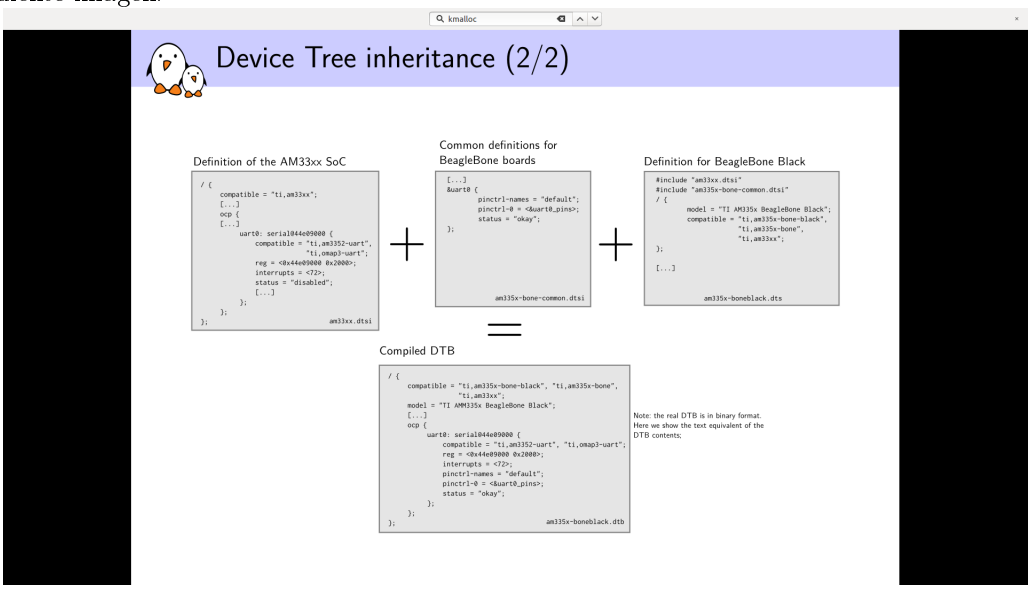
2. #interrupt-cells : especifia el numero de cells existen en la propiedad interrupts. Se localiza en el interrupt Controller.

3. interrupt-parent : propiedad que contiene un phandle al interrupt controller.

4. interrupts : propiedad que se encuentra en cada dispositivo que genera interrupciones y especifica una lista de interrupt specifiers que segun el interrupt controller manejara e interpretara de una forma.

include

la directiva #include permite añadir un dts a otro tal y como vemos en la siguiente imagen.



Ejemplo DTS

```
/dts-v1/;
/ {
    compatible = "acme,coyotes-revenge";
    #address-cells = <1>;
    #size-cells = <1>;
    interrupt-parent = <&intc>;
    cpus {
        #address-cells = <1>;
        #size-cells = <0>;
        cpu@0 {
            compatible = "arm,cortex-a9";
                        // cada CPU se asocia con un
                           UID,
                        // el size es vacio ya que la
                           cpu esta asociada a un
                           unica direccion
```

```
            reg = <0>;
        };
        cpu@1 {
            compatible = "arm,cortex-a9";
            reg = <1>;
        };
    };
    serial@101f0000 {
        compatible = "arm,pl011";
        reg = <0x101f0000 0x1000 >;
        interrupts = < 1 0 >;
    };
    serial@101f2000 {
        compatible = "arm,pl011";
        reg = <0x101f2000 0x1000 >;
        interrupts = < 2 0 >;
    };
    gpio@101f3000 {
        compatible = "arm,pl061";
                // el dispositivo esta en las regiones
                // 0x101f3000 - 0x101f4000
                // 0x101f4000 - 0x101f4010
        reg = <0x101f3000 0x1000
               0x101f4000 0x0010>;
        interrupts = < 3 0 >;
    };
    intc: interrupt-controller@10140000 {
        compatible = "arm,pl190";
        reg = <0x10140000 0x1000 >;
        interrupt-controller;
                // 2 cells, la primera identifica la
                    interrupcion
                // y el segundo valor codifica el flag
                // tal que activo alta vs activa baja,
                     edge vs level sensitive,...
                // ver especificaciones del
                    controlador
        #interrupt-cells = <2>;
    };
    spi@10115000 {
        compatible = "arm,pl022";
        reg = <0x10115000 0x1000 >;
        interrupts = < 4 0 >;
    };
};
```

DTS - Kernel

La manera más habitual de acceder a las propiedades del DTS desde el kernel es a traves del método of_get_property()

```
/* /drivers/of/base.c */

/*
 * Find a property with a given name for a given node
 * and return the value.
*/
const void *of_get_property(const struct device_node *
    np, const char *name,
                            int *lenp) {
        struct property *pp = of_find_property(np,
            name, lenp);
        return pp ? pp->value : NULL;
}
EXPORT_SYMBOL(of_get_property);

struct property *of_find_property(const struct
    device_node *np,
                                  const char *name,
                                  int *lenp) {
        struct property *pp;
        unsigned long flags;
        raw_spin_lock_irqsave(&devtree_lock,flags);
        //seccion critica
        pp = __of_find_property(np, name, lenp);
        //fin seccion critica
        raw_spin_unlock_irqrestore(&devtree_lock,
            flags);
        return pp;
}
EXPORT_SYMBOL(of_find_property);

static struct property *__of_find_property(const
    struct device_node *np,
                                           const char
                                              *name,
                                              int *
                                              lenp) {
        struct property *pp;
        if (!np)
                return NULL;
        for (pp = np->properties; pp; pp = pp->next) {
```

```
                if (of_prop_cmp(pp->name, name) == 0)
                    {
                        if (lenp)
                                *lenp = pp->length;
                        break;
                }
        }
        return pp;
}

/* /include/linux/of.h */

#define of_prop_cmp(s1, s2)             strcmp((s1), (
    s2))

struct property {
        char    *name;
        int     length;
        void    *value;
        struct property *next;
        unsigned long _flags;
        unsigned int unique_id;
        struct bin_attribute attr;
};

struct device_node {
        const char *name;
        const char *type;
        phandle phandle;
        const char *full_name;
        struct fwnode_handle fwnode;
        struct  property *properties;
        struct  property *deadprops;    /* removed
            properties */
        struct  device_node *parent;
        struct  device_node *child;
        struct  device_node *sibling;
        struct  kobject kobj;
        unsigned long _flags;
        void    *data;
#if defined(CONFIG_SPARC)
        const char *path_component_name;
        unsigned int unique_id;
        struct of_irq_controller *irq_trans;
#endif
};
```

Part VI
Programación Kernel

El kernel esta implementado en su mayoria en C y una parte en ensamblador. Todo el codigo es compilado con gcc. El kernel debe ser independiente del user space, por ello no se pueden usar las librerias estandar de C, sin embargo el kernel proporciona funciones similares.

A continuacion veremos algunas de las funciones más recurrentes a lo largo del código.

Linked Lists

se incluye en /include/linux/list.h y es usado continuamente en el kernel. Se añade una struct list_head a la estructura cuya instancias seran parte de la lista. La lista si es usada como variable global se define con la macro LIST_HEAD(), si no se define un elemento struct list_head y se inicializa con INIT_LIST_HEAD().

Los metodos mas relevantes son:

1. añadir elementos : list_add(), list_add_tail()
2. borrar elementos : list_del()
3. mover elementos : list_move(), list_move_tail()
4. reemplazar elementos : list_replace()
5. Iterar sobre la lista : list_for_each(), list_for_each_entry()

```
/*/include/linux/types.h*/

 struct list_head {
        struct list_head *next, *prev;
 };

/*/include/linux/list.h*/

 /*INICIALIZACION*/
 #define LIST_HEAD_INIT(name) { &(name), &(name) }

 #define LIST_HEAD(name) \
        struct list_head name = LIST_HEAD_INIT(name)

 static inline void INIT_LIST_HEAD(struct list_head *
    list) {
        list->next = list;
        list->prev = list;
```

```
 }

 /*AÑADIR ELEMENTOS*/
//[prev <-> new <-> next]
 static inline void __list_add(struct list_head *new,
                               struct list_head *prev,
                               struct list_head *next)
                                   {
         next->prev = new;
         new->next = next;
         new->prev = prev;
         prev->next = new;
 }
//[head new head->next]
/**
 * list_add - add a new entry
 * @new: new entry to be added
 * @head: list head to add it after
 *
 * Insert a new entry after the specified head.
 * This is good for implementing stacks.
 */
static inline void list_add(struct list_head *new,
    struct list_head *head) {
         __list_add(new, head, head->next);
}
//[head->prev new head]
/**
 * list_add_tail - add a new entry
 * @new: new entry to be added
 * @head: list head to add it before
 *
 * Insert a new entry before the specified head.
 * This is useful for implementing queues.
 */
static inline void list_add_tail(struct list_head *new
    , struct list_head *head) {
         __list_add(new, head->prev, head);
}

/*ITERACION*/

#define list_for_each(pos, head) \
         for (pos = (head)->next; pos != (head); pos =
            pos->next)
```

```
/**/
/* /include/linux/kernel.h *
 * container_of - cast a member of a structure out to
    the containing structure
 * @ptr:        the pointer to the member.
 * @type:       the type of the container struct this
    is embedded in.
 * @member:     the name of the member within the
    struct.
 *
 */
  /*
        struct container_struct{
                struct member_struct member_name
        }
        struct member_struct *mm = INIT_MEMBER_EXAMPLE
            ()
        struct container_struct *container =
            container_of(mm, container_struct,
            member_name)

        */
#define container_of(ptr, type, member) ({      \
                                const typeof( ((type
                                    *)0)->member ) *
                                    __mptr = (ptr);
                                    \
                                  (type *)( (char *)
                                     __mptr - offsetof
                                     (type,member) )
                                     ;})

#define list_entry(ptr, type, member) \
        container_of(ptr, type, member)

#define list_first_entry(ptr, type, member) \
        list_entry((ptr)->next, type, member)

#define list_next_entry(pos, member) \
        list_entry((pos)->member.next, typeof(*(pos)),
            member)

#define list_for_each_entry(pos, head, member)
                                \
        for (pos = list_first_entry(head, typeof(*pos)
            , member);        \
```

```
                &pos->member != (head);
                                                \
                pos = list_next_entry(pos, member))

/* Ejemplos */

struct atmel_tc {
        /* some members */
        struct list_head node;
};

/* Define the global list */
static LIST_HEAD(tc_list);

static int __init tc_probe(struct platform_device *
    pdev) {
        struct atmel_tc *tc;
        tc = kzalloc(sizeof(struct atmel_tc),
            GFP_KERNEL);
        /* Add an element to the list */
        list_add_tail(&tc->node, &tc_list);
        /*
                tc_list -> prev = node
                tc_list -> prev -> next = node

                elementos_tc_list ... node.

        */
}

struct atmel_tc *atmel_tc_alloc(unsigned block, const
    char *name) {
        struct atmel_tc *tc;
        /* Iterate over the list elements */
        list_for_each_entry(tc, &tc_list, node) {
                /* Do something with tc */
        }
        [...]
}
```

Kmalloc

Es la funcion para reservar espacio de memoria en el kernel.

Flags

Estas son las más usadas, el resto pueden encontrase en include/linux/gfp.h

1. GFP_KERNEL : reserva de memoria estandar, la reserva puede bloquearse para poder encontrar suficiente memoria disponible.
2. GFP_ATOMIC : reserva de RAM, no esta permitido el bloqueo, puede fallar si no hay memoria libre disponible.
3. GFP_DMA : reserva memoria en el area fisica usada por el DMA.

API

```
/*
*reserva size bytes y devuelveun puntero al area en
    memoria virtual
*/
void *kmalloc(size_t size, int flags);

void kfree(const void *objp);

/*inicializa a 0*/
void *kzalloc(size_t size, gfp_t flags);

/* reserva memoria para un array de n elementos de
    tamaño size y los inicializa a 0*/
void *kcalloc(size_t n, size_t size, gfp_t flags);

/*cambia el tamaño del buffer apuntado por p a
    new_size*/
void *krealloc(const void *p, size_t new_size, gfp_t
    flags);

/*
        Las siguientes funciones liberan el espacio de
            memoria
        automaticamente cuando el dispositivo se
            desconecta
        o falla en la inicializacion, por ello se usan
            en probe()
*/
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);
```

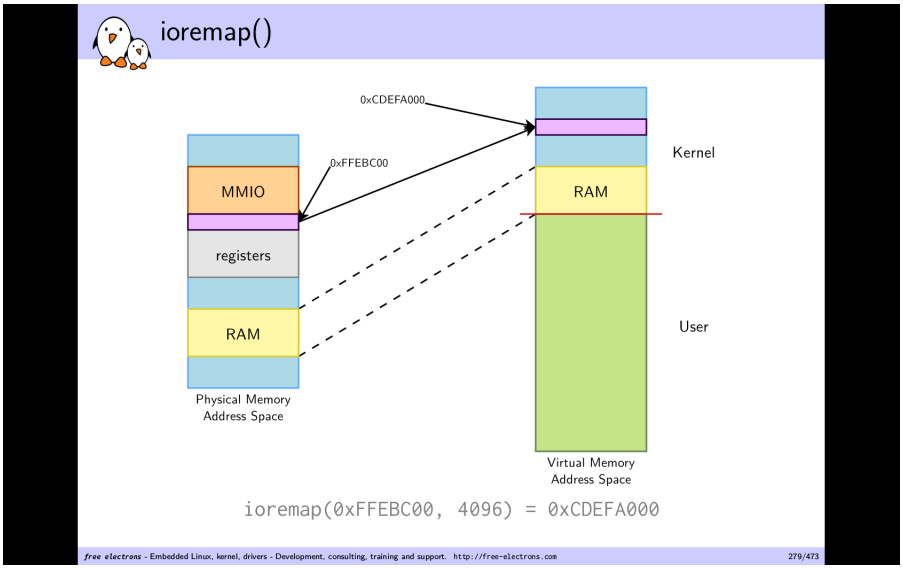
IO memory

mapeado en memoria virtual

```
/*
        https://lkml.org/lkml/2004/9/12/249
        iomem implica que nunca sera desreferenciado
            por ser un puntero noderef
        y esta en el espacio de memoria 2, diferente
            al normal que es 0
*/
# define __iomem        __attribute__((noderef,
    address_space(2)))

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

/*en device drivers estas funciones estan deprecadas y
     se usan las siguientes*/
void __iomem *devm_ioremap(struct device *dev,
    resource_size_t offset,resource_size_t size);
void devm_iounmap(struct device *dev, void __iomem *
    addr);
```



R/W

```
unsigned int ioread8(void __iomem *addr)
unsigned int ioread16(void __iomem *addr)
unsigned int ioread32(void __iomem *addr)
unsigned int ioread16be(void __iomem *addr)
unsigned int ioread32be(void __iomem *addr)

void iowrite8(u8 val, void __iomem *addr)
void iowrite16(u16 val, void __iomem *addr)
void iowrite32(u32 val, void __iomem *addr)
void iowrite16be(u16 val, void __iomem *addr)
void iowrite32be(u32 val, void __iomem *addr)
```

Spinlocks

locks usados para las secciones criticas o interrupt handles.

```
/*INICIALIZACION*/
DEFINE_SPINLOCK(my_lock);
void spin_lock_init(spinlock_t *lock);

/*VARIANTES*/

/*No desactiva interrupciones*/
void spin_lock(spinlock_t *lock);
void spin_unlock(spinlock_t *lock);

/*Desactiva las IRQ, guarda el estado en flags al
   bloquear y devuelve su valor al continuar*/
void spin_lock_irqsave(spinlock_t *lock, unsigned long
    flags);
void spin_unlock_irqrestore(spinlock_t *lock, unsigned
    long flags);

/*Desactiva las interrupciones software*/
void spin_lock_bh(spinlock_t *lock);
void spin_unlock_bh(spinlock_t *lock);
```

Part VII
AÑADIR DRIVER

En esta parte deberos ver 2 ficheros el Kconfig y Makefile en el directorio donde tenemos el codigo fuente de nuestro driver. /linux/drivers/....

Makefile

para añadir nuestro driver le diremos al fichero que hay un objeto en el directorio que debe compilarse

Ejemplo:
codigo fuente: driver_source_code.c
Makefile:
obj-x += driver_source-code.o
donde x puede tomar los siguientes valores:
y (built-in)
m (module)
n (no loaded)
CONFIG_X : donde X es un identificador del fichero kbuild

Kbuild

El fichero Kbuild en un fichero de configuración.

veremos algunas estructuras

```
/*se mostrara un menu de configuración con el nombre*/
menu "nombre"
...
...
...
endmenu

/*carga ficheros kbuild de otros directorios*/
source "directorio"

/*
 * entradas de configuracion
 * tipos : triestate (y,m,n)
 *         bool (y,n)
 *         int
 *         hex
 *         string
 * dependencias :
```

```
 *              A depends on B : A no es visible si B
    no esta activado.
 *              A select B : si A es activada
    automaticamente se activa B.
*/
config NOMBRE
        tipo "nombre que aparece en la interfaz"
        depends on B
        select C
        default VALOR
        help
                ..... //descripcion
```