arm

Embedded Linux

Introduction to Linux Kernel modules

Goal

To discuss the CPU-I/O interface and the Virtual File System abstraction, and to present an introduction to Linux Kernel modules

Summary

Introduction

CPU – I/O interface

I/O taxonomy

Linux devices

Virtual File System abstraction

Linux kernel modules

Summary

Introduction

CPU – I/O interface

I/O taxonomy

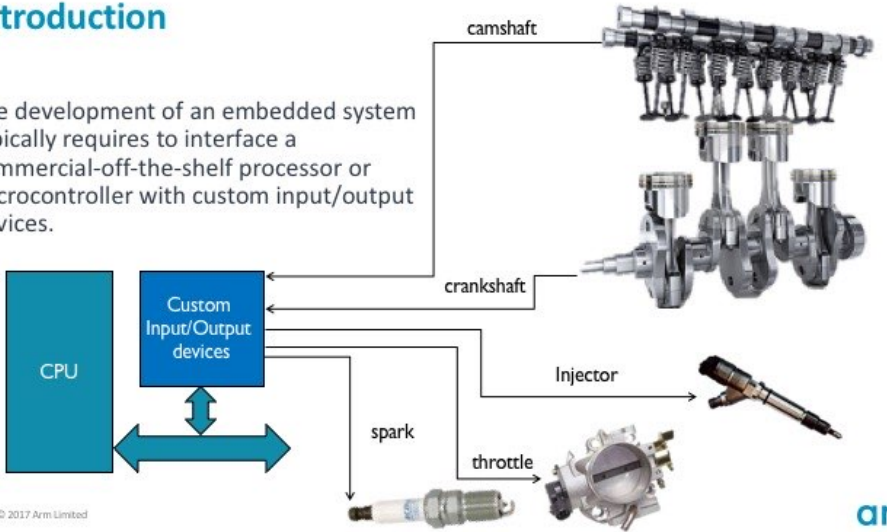
Linux devices

Virtual File System abstraction

Linux kernel modules

Introduction

The development of an embedded system typically requires to interface a commercial-off-the-shelf processor or microcontroller with custom input/output devices.

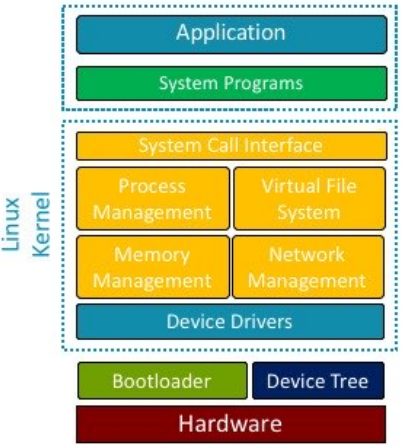


arm

Introduction

From the software point of view interfacing with custom input/output (I/O) devices requires:

- A user-space application that reads/writes data from/to a suitable abstraction interface of the hardware
- A device driver that translates the operations of the abstraction interface into hardware-specific operations



arm

Summary

Introduction

CPU – I/O interface

I/O taxonomy

Linux devices

Virtual File System abstraction

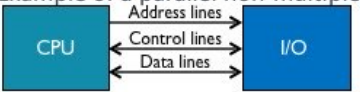
Linux Kernel modules

CPU – I/O Interface

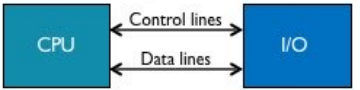
The interconnection between CPU and I/O device can be broadly classified as:

- Parallel, where N independent lines connect the CPU with the I/O, and one word (e.g., 8-/16-/32-bits) or blocks of words are transferred at each operation
- Serial, where M (M<<N) lines connect the CPU with the I/O, and one bit is transferred at each operation according to a serial communication protocol (such as SPI, I2C, USB)

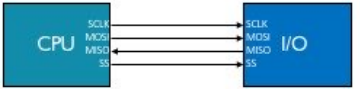
Example of a parallel non-multiplexed bus



Example of a parallel multiplexed bus



Example of a serial bus (SPI)



arm

CPU – I/O Interface

CPU and I/O operate asynchronously:

- CPU runs software.
- I/O performs its own tasks.

When the CPU has to read/write from/to the I/O, a read/write operation is performed.

- Read/write operations are always initiated by the CPU.

How is it possible for the CPU to recognize that an I/O has data ready to be read?

Two techniques are possible:

- Polling: The CPU checks the peripheral periodically.
- Interrupt: The peripheral asks the CPU's attention when needed.

CPU – I/O Interface with Polling

The software periodically reads the status of each peripheral.

In case the I/O needs to be serviced by the CPU, the corresponding program (service routine) is executed.

Advantage

- Simple to implement

Disadvantages

- High latency as I/O are polled serially (time when the I/O is served – time when the peripheral needs service)
- CPU time is wasted when polled devices are not to be serviced.

Example

- Device A provides 1 byte.
- Device B provides 2 bytes.

```
while(1)
{
  if(A_is_ready())
  {
    resA = read_byte_from_A();
  }
  if(B_is_ready())
  {
    resB_1 = read_byte_from_B();
    resB_2 = read_byte_from_B();
  }
```

} Device A provides 1 byte.
Device B provides 2 bytes.

arm

CPU – I/O Interface with Interrupt

A dedicated line (interrupt request, IRQ) connects the I/O with the CPU.

In case the I/O needs to be serviced, it asserts the IRQ line.

At the end of each instruction, the CPU checks for the IRQ line; if asserted, it runs the corresponding service routing.

Advantage

- Low latency (time when the peripheral is served – time when the peripheral needs service)

Disadvantage

- Higher hardware complexity

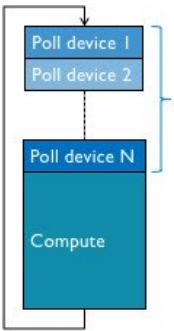
Example

- Device A provides 1 byte (IRQ1)
- Device B provides 2 bytes (IRQ2)

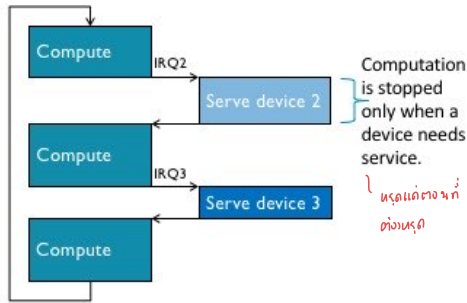
```
IRQ1()
{
  resA = read_byte_from_A();
}

IRQ2()
{
  resB_1 = read_byte_from_B();
  resB_2 = read_byte_from_B();
}
```

CPU – I/O Interface

In case of polling, the application execution timeline will be the following.



In case of interrupt, the application execution timeline will be the following.



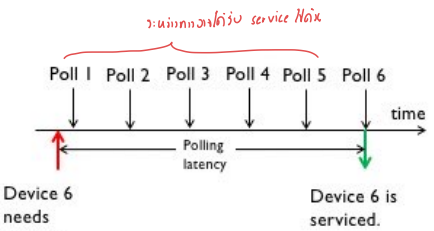
arm

CPU – I/O Interface Latency

In case of polling, the latency depends on the order in which the peripherals are polled.

Example

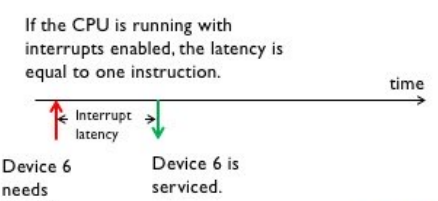
- 10 devices; only device 6 needs service.



In case of interrupt, the latency depends on the number of requests simultaneously asserted, the CPU operating mode, and its architecture.

Example

- 10 devices; only device 6 needs service.



arm

CPU – I/O Interface

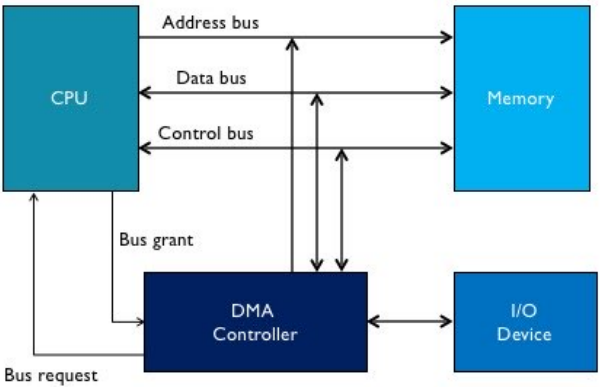
Data transfer can happen in two fashions:

- Character-based transfer, where data are transferred one word (e.g., 8/16/32/... bits) at a time
- Block-based transfer, where data are transferred as clusters of bytes (e.g, 64/128/... bytes)

The data transfer can be performed by:

- The CPU, which moves data from memory to I/O (or vice-versa) through a program
- The DMA, which releases the CPU from data transfer

Direct Memory Access (DMA) Architecture



arm

Direct Memory Access (DMA) Transfer Modes → data transfer in 3 mode

Burst

- An entire block of data is transferred in one contiguous sequence.
- The CPU remains inactive for relatively long periods of time (until the whole transfer is completed).

Cycle stealing

- DMA transfers one byte of data and then releases the bus returning control to the CPU.
- It continually issues requests, transferring one byte of data per request, until it has transferred the entire block of data.
- It takes longer to transfer data/the CPU is blocked for less time.

Transparent

- DMA transfers data when the CPU is performing operations that do not use the system buses.

Summary

Introduction

CPU – I/O interface

I/O taxonomy

Linux devices

Virtual File System abstraction

Linux Kernel modules

arm

I/O Taxonomy

1. Output devices which actuate commands via:

- Analog signals: output pins carrying voltages or currents
- Digital level-triggered discrete signals: output pins forming parallel bus carrying digital voltage levels
- Digital pulse-width modulated (PWM) discrete signals: output pins forming parallel bus carrying digital square waveforms with given frequencies and duty cycles
- Bus-based signals: output pins implementing serial communication protocols

2. Inputs devices which acquire sensors status via:

- Analog inputs: input pins carrying voltages, which required A/D conversion
- Digital level-triggered discrete signals: input pins carrying digital voltage levels
- Digital pulse-width modulated discrete signals: input pins forming parallel bus carrying digital information in the form of pulses duration and/or number of pulses
- Bus-based signals: input pins implementing serial communication protocols

Typical Operations

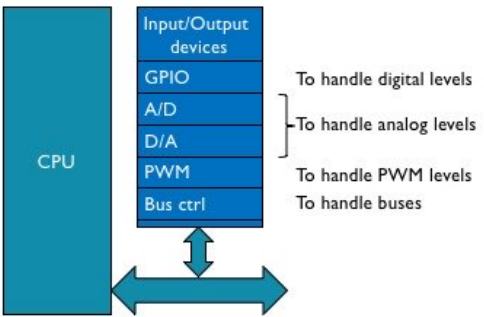
Output devices

- Generate analog levels through D/A
- General digital levels
- Generate PWM signals
- General bus transfer

Input devices

- Acquire an analog value though A/D conversion
- Read digital level
- Measure timing/repetition of digital pulses
- Read bus transfer

CPUs for embedded systems contain dedicated hw for these operations.



arm

Summary

Introduction

CPU – I/O interface

I/O taxonomy

Linux devices

Virtual File System abstraction

Linux Kernel modules

arm

Linux Devices

Linux provides an abstraction to make communication with I/O easy.

- Software developer does not need to know every detail of the physical device.
- Portability can be increased by using the same abstraction for different I/O devices.

Linux recognizes three classes of devices:

- Character devices, which are devices that can be accessed as stream of words (e.g., 8-/16-/32-/... bits) as in a file; reading word n requires reading all the preceding words from 0 to n-1.
- Block devices, which are devices that can be accessed only as multiples of one block, where a block is 512 bytes of data or more. Typically, block devices host file systems.
- Network interfaces, which are in charge of sending and receiving data packets through the network subsystem of the kernel

Summary

Introduction

CPU – I/O interface

I/O taxonomy

Linux devices

Virtual File System abstraction

Linux Kernel modules

arm

The Virtual File System (VFS) Abstraction

Character/block devices are accessed as files stored in the file system, as each device is associated with a device file.

Typical usage:

- Open the device file
- Read/Write data from/to device file
- Close the device file

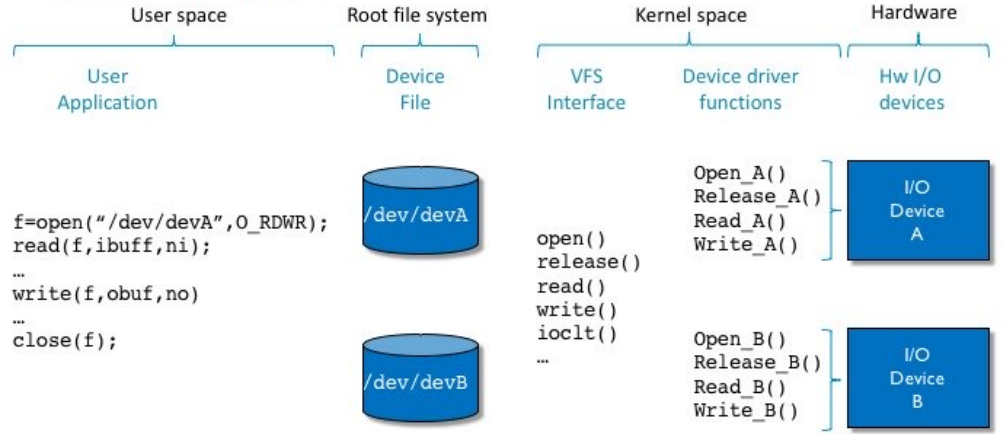
Linux forwards the open/read/write/close operations to the I/O device associated to the device file.

The operations for each I/O device are implemented by a custom piece of software in the Linux kernel: the device driver.

In the following, we will consider only character (char) devices.

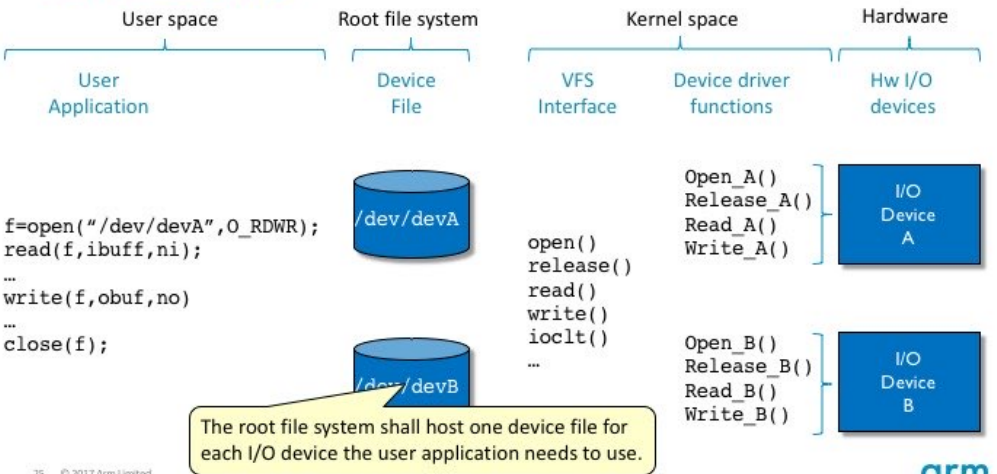
VFS: An Example

How device operated on, what from its take in each level



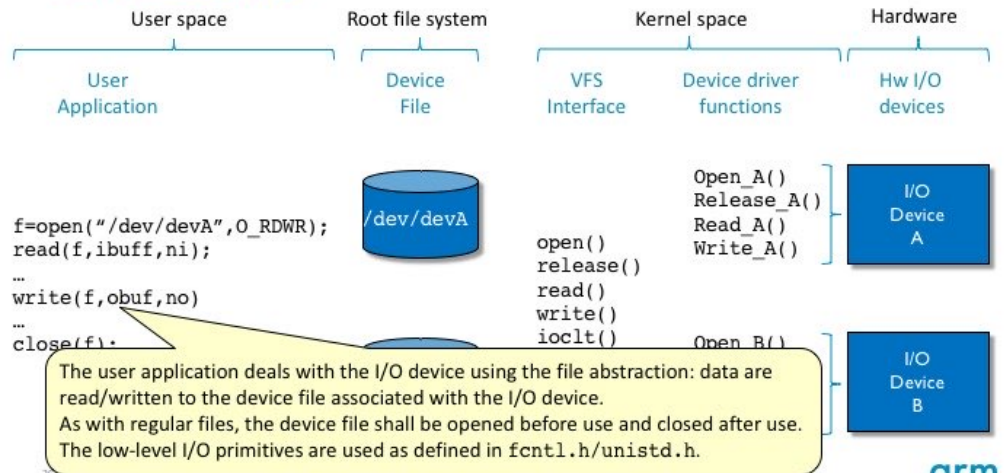
arm

VFS: An Example

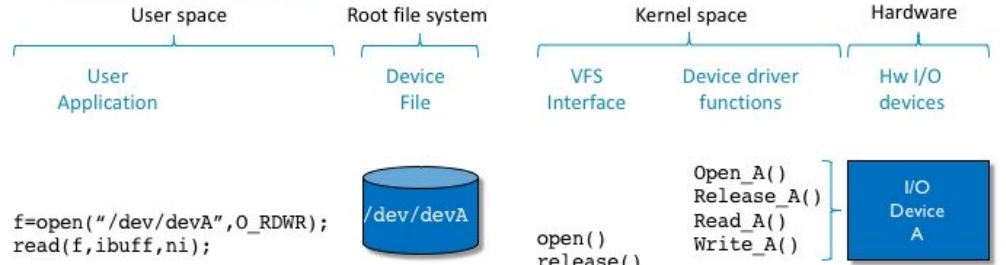


arm

VFS: An Example



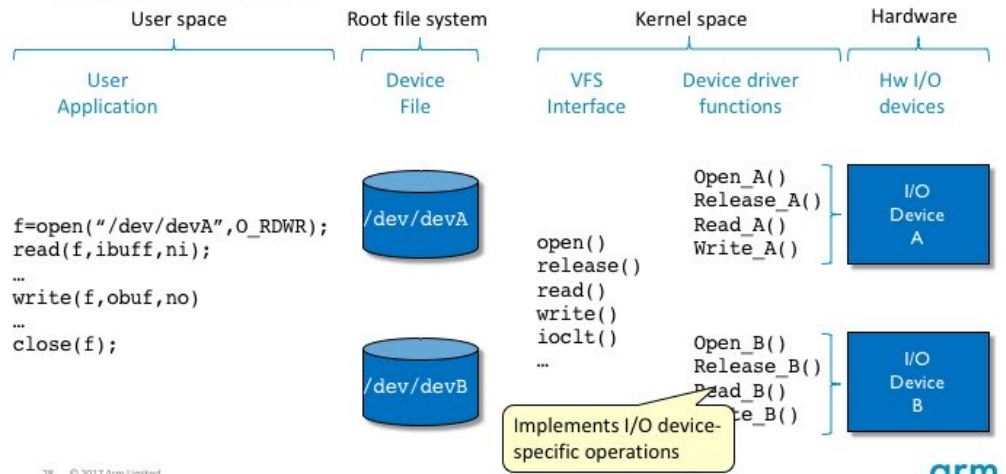
VFS: An Example



The VFS establishes the association between the low-level I/O primitives used in the user application and the corresponding device driver functions. For example:

User application	VFS	Device Driver
f=open("/dev/devA",O_RDWR) →	open() →	Open_A()

VFS: An Example



arm

VFS Functions: include/linux/fs.h

```
struct file_operations {
  struct module *owner;
  loff_t (*llseek) (struct file *, loff_t, int);
  ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
  ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
  ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
  ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
  int (*iterate) (struct file *, struct dir_context *);
  int (*iterate_shared) (struct file *, struct dir_context *);
  unsigned int (*poll) (struct file *, struct poll_table_struct *);
  long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
  long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
  int (*mmap) (struct file *, struct vm_area_struct *);
  int (*open) (struct inode *, struct file *);
  int (*flush) (struct file *, fl_owner_t id);
  int (*release) (struct inode *, struct file *);
  int (*fsync) (struct file *, loff_t, loff_t, int datasync);
  int (*aio_fsync) (struct kiocb *, int datasync);
  int (*fasync) (int, struct file *, int);
  int (*lock) (struct file *, int, struct file_lock *);
  ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
  unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
  int (*check_flags)(int);
  int (*flock) (struct file *, int, struct file_lock *);
  ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
  ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
  int (*setlease)(struct file *, long, struct file_lock **, void **);
  long (*fallocate)(struct file *file, int mode, loff_t offset,
  loff_t len);
  void (*show_fdinfo)(struct seq_file *m, struct file *f);
  #ifndef CONFIG_MMU
    unsigned (*mmap_capabilities)(struct file *);
  #endif
  ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
  loff_t, size_t, unsigned int);
  int (*clone_file_range)(struct file *, loff_t, struct file *, loff_t,
  u64);
  ssize_t (*dedupe_file_range)(struct file *, u64, u64, struct file *,
  u64);
 };
```

VFS functions: prototypes of the functions Linux sets available for accessing a file. In case of device files, the actions each function performs are defined by the corresponding device driver.

VFS Functions: `include/linux/fs.h`

For character devices the most commonly used VFS functions are the following:

- `ssize_t (*read) (struct file *, char *__user, size_t, loff_t *)`: it reads data from a file.
- `ssize_t (*write) (struct file *, const char *__user, size_t, loff_t *)`: it writes data to a file.
- `int (*ioctl) (struct *inode, struct file *, unsigned int, unsigned long)`: it performs custom operations to the file.
- `int (*open) ( struct *inode, struct file * )`: it prepares a file for use.
- `int (*release) ( struct inode *, struct file * )`: it indicates the file is no longer in use.

The Device File Concept

The device file is the intermediator through which a user application can exchange data with a device driver.

The device file does not contain any data, while its descriptor contains the relevant information to identify the corresponding driver:

- The device file type, which could be either c = character device, b = block device, or p = named pipe (inter-process communication mechanism)
- The major number, which is an integer number that identifies univocally a device driver in the Linux kernel
- The minor number, which is used to discriminate among multiple instances of I/O devices handled by the same device driver

Summary

Introduction

CPU – I/O interface

I/O taxonomy

Linux devices

Virtual File System abstraction

Linux Kernel modules

arm

Linux Kernel Modules

A device driver provides an I/O-device specific implementation of the Virtual File System (VFS) abstraction, and it is located in the kernel space.

A device driver can be:

- Linked with the Linux Kernel and executed at system bootstrap
- Kernel module, which is loaded at runtime through suitable system programs, after the Linux Kernel is booted

In the following slides, we will focus on kernel modules, but the same concepts apply to device drivers linked with the Linux Kernel.

Linux Kernel Modules

System programs

- `mknod`, to create a device file
- `insmod`, to insert the module in the kernel
- `rmmod`, to remove the module from the kernel
- `lsmod`, to list the modules loaded in the kernel

Functions provided by a kernel module:

- Initialization function, called upon the execution of the `insmod` system program, takes care of making Linux aware that a new device driver is available
- Clean-up function, called upon the execution of `rmmod`, to remove the device driver from the Linux Kernel
- Custom-specific implementations of the VFS abstraction

Linux Kernel Modules: The Initialization Function

```
static dev_t dummy_dev;

struct cdev dummy_cdev;

static char buffer[64];

struct file_operations dummy_fops = {
    .owner = THIS_MODULE,
    .read = dummy_read,
};

static int __init dummy_module_init(void)
{
    printk(KERN_INFO "Loading dummy_module\n");
    alloc_chrdev_region(&dummy_dev, 0, 1, "dummy_dev");
    printk(KERN_INFO "%s\n", format_dev_t(buffer, dummy_dev));
    cdev_init(&dummy_cdev, &dummy_fops);
    dummy_cdev.owner = THIS_MODULE;
    cdev_add(&dummy_cdev, dummy_dev, 1);
    return 0;
}
```

Linux Kernel Modules: The Initialization Function

```
static dev_t dummy_dev;

struct cdev dummy_cdev;

static char buffer[64];

struct file_operations dummy_fops = {
    .owner = THIS_MODULE,
    .read = dummy_read,
};

static int __init dummy_module_init(void)
{
    printk(KERN_INFO "Loading dummy_module\n");
    alloc_chrdev_region(&dummy_dev, 0, 1, "dummy_dev");
    printk(KERN_INFO "%s\n", format_dev_t(buffer, dummy_dev));
    cdev_init(&dummy_cdev, &dummy_fops);
    dummy_cdev.owner = THIS_MODULE;
    cdev_add(&dummy_cdev, dummy_dev, 1);
    return 0;
}
```

Data structure containing the major number and the first minor number for the module. It identifies univocally the module in the kernel. It shall be used when creating the device file associated with the module.

Linux Kernel Modules: The Initialization Function

```
static dev_t dummy_dev;

struct cdev dummy_cdev;

static char buffer[64];

struct file_operations dummy_fops = {
    .owner = THIS_MODULE,
    .read = dummy_read,
};

static int __init dummy_module_init(void)
{
    printk(KERN_INFO "Loading dummy_module\n");
    alloc_chrdev_region(&dummy_dev, 0, 1, "dummy_dev");
    printk(KERN_INFO "%s\n", format_dev_t(buffer, dummy_dev));
    cdev_init(&dummy_cdev, &dummy_fops);
    dummy_cdev.owner = THIS_MODULE;
    cdev_add(&dummy_cdev, dummy_dev, 1);
    return 0;
}
```

Data structure used to describe the properties of a character device (see next slide)

Linux Kernel Modules: The cdev Data Structure

include/linux/cdev.h

```
struct cdev {
  struct kobject kobj;
  struct module *owner;
  const struct file_operations *ops;
  struct list_head list;
  dev_t dev;
  unsigned int count;
};
```

The relevant fields for the module programmer are:

- `ops`: pointer to the structure defining the association between Virtual Filesystem (VFS) functions and their specific implementations for the module being developed
- `dev`: major number associated with the module
- `count`: minor number associated with the module

Linux Kernel Modules: The Initialization Function

```
static dev_t dummy_dev;

struct cdev dummy_cdev;

static char buffer[64];

struct file_operations dummy_fops = {
    .owner = THIS_MODULE,
    .read = dummy_read,
};

static int __init dummy_module_init(void)
{
    printk(KERN_INFO "Loading dummy_module\n");
    alloc_chrdev_region(&dummy_dev, 0, 1, "dummy_dev");
    printk(KERN_INFO "%s\n", format_dev_t(buffer, dummy_dev));
    cdev_init(&dummy_cdev, &dummy_fops);
    dummy_cdev.owner = THIS_MODULE;
    cdev_add(&dummy_cdev, dummy_dev, 1);
    return 0;
}
```

Buffer used for displaying output messages on the Linux console

Linux Kernel Modules: The Initialization Function

```
static dev_t dummy_dev;

struct cdev dummy_cdev;

static char buffer[64];

struct file_operations dummy_fops = {
    .owner = THIS_MODULE,
    .read = dummy_read,
};

static int __init dummy_module_init(void)
{
    printk(KERN_INFO "Loading dummy_module\n");
    alloc_chrdev_region(&dummy_dev, 0, 1, "dummy_dev");
    printk(KERN_INFO "%s\n", format_dev_t(buffer, dummy_dev));
    cdev_init(&dummy_cdev, &dummy_fops);
    dummy_cdev.owner = THIS_MODULE;
    cdev_add(&dummy_cdev, dummy_dev, 1);
    return 0;
}
```

Data structure used to associate the VFS functions to their module-specific implementations.
In this example, the `read()` VFS function is implemented by the `dummy_read()` function.

Linux Kernel Modules: The Initialization Function

```
static dev_t dummy_dev;

struct cdev dummy_cdev;

static char buffer[64];

struct file_operations dummy_fops = {
    .owner = THIS_MODULE,
    .read = dummy_read,
};

static int __init dummy_module_init(void)
{
    printk(KERN_INFO "Loading dummy_module\n");
    alloc_chrdev_region(&dummy_dev, 0, 1, "dummy_dev");
    printk(KERN_INFO "%s\n", format_dev_t(buffer, dummy_dev));
    cdev_init(&dummy_cdev, &dummy_fops);
    dummy_cdev.owner = THIS_MODULE;
    cdev_add(&dummy_cdev, dummy_dev, 1);
    return 0;
}
```

The module initialization function. It is executed as soon as the module enters the Linux kernel and takes care of making the Kernel aware that the new module is available.

Linux Kernel Modules: The Initialization Function

```
static dev_t dummy_dev;

struct cdev dummy_cdev;

static char buffer[64];

struct file_operations dummy_fops = {
    .owner = THIS_MODULE,
    .read = dummy_read,
};

static int __init dummy_module_init(void)
{
    printk(KERN_INFO "Loading dummy_module\n");
    alloc_chrdev_region(&dummy_dev, 0, 1, "dummy_dev");
    printk(KERN_INFO "%s\n", format_dev_t(buffer, dummy_dev));
    cdev_init(&dummy_cdev, &dummy_fops);
    dummy_cdev.owner = THIS_MODULE;
    cdev_add(&dummy_cdev, dummy_dev, 1);
    return 0;
}
```

Kernel equivalent of the `printf()`.

Linux Kernel Modules: The Initialization Function

```
static dev_t dummy_dev;

struct cdev dummy_cdev;

static char buffer[64];

struct file_operations dummy_fops = {
    .owner = THIS_MODULE,
    .read = dummy_read,
};

static int __init dummy_module_init(void)
{
    printk(KERN_INFO "Loading dummy_module\n");
    alloc_chrdev_region(&dummy_dev, 0, 1, "dummy_dev");
    printk(KERN_INFO "%s\n", format_dev_t(buffer, dummy_dev));
    cdev_init(&dummy_cdev, &dummy_fops);
    dummy_cdev.owner = THIS_MODULE;
    cdev_add(&dummy_cdev, dummy_dev, 1);
    return 0;
}
```

It registers a range of character device numbers with the function.

`int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count, const char *name)`

where:
`dev` is the dynamically-selected major number of the module;
`baseminor` is the first minor number for the module;
`count` is the number of minor number to reserve for the module;
and `name` is the module name.

Linux Kernel Modules: The Initialization Function

```
static dev_t dummy_dev;

struct cdev dummy_cdev;

static char buffer[64];

struct file_operations dummy_fops = {
    .owner = THIS_MODULE,
    .read = dummy_read,
};

static int __init dummy_module_init(void)
{
    printk(KERN_INFO "Loading dummy_module\n");
    alloc_chrdev_region(&dummy_dev, 0, 1, "dummy_dev");
    printk(KERN_INFO "%s\n", format_dev_t(buffer, dummy_dev));
    cdev_init(&dummy_cdev, &dummy_fops);
    dummy_cdev.owner = THIS_MODULE;
    cdev_add(&dummy_cdev, dummy_dev, 1);
    return 0;
}
```

It prints on the Linux console the major number and the minor number associated with the just-registered character device.

Linux Kernel Modules: The Initialization Function

```
static dev_t dummy_dev;

struct cdev dummy_cdev;

static char buffer[64];

struct file_operations dummy_fops = {
    .owner = THIS_MODULE,
    .read = dummy_read,
};

static int __init dummy_module_init(void)
{
    printk(KERN_INFO "Loading dummy_module
    alloc_chrdev_region(&dummy_dev, 0, 1, "
    printk(KERN_INFO "%s\n", format_dev_t(buffer
    cdev_init(&dummy_cdev, &dummy_fops);
    dummy_cdev.owner = THIS_MODULE;
    cdev_add(&dummy_cdev, dummy_dev, 1);
    return 0;
}
```

It initializes the character device data structure.

```
void cdev_init( struct cdev *cdev, const struct
file_operations * fops);
```

where:
`cdev` is the structure to initialize and
`fops` is the `file_operations` for the device.

Linux Kernel Modules: The Initialization Function

```
static dev_t dummy_dev;

struct cdev dummy_cdev;

static char buffer[64];

struct file_operations dummy_fops = {
    .owner = THIS_MODULE,
    .read = dummy_read,
};

static int __init dummy_module_init(void)
{
    printk(KERN_INFO "Loading dum
    alloc_chrdev_region(&dummy_de
    printk(KERN_INFO "%s\n", form
    cdev_init(&dummy_cdev, &dummy_fops);
    dummy_cdev.owner = THIS_MODULE;
    cdev_add(&dummy_cdev, dummy_dev, 1);
    return 0;
}
```

It sets the owner of the module using the `THIS_MODULE` macro.

Linux Kernel Modules: The Initialization Function

```
static dev_t dummy_dev;

struct cdev dummy_cdev;

static char buffer[64];

struct file_operations dummy_fops
    .owner = THIS_MODULE,
    .read = dummy_read,
};

static int __init dummy_module_in
{
    printk(KERN_INFO "Loading dumm
    alloc_chrdev_region(&dummy_dev
    printk(KERN_INFO "%s\n", format_
    cdev_init(&dummy_cdev, &dummy_fops);
    dummy_cdev.owner = THIS_MODULE;
    cdev_add(&dummy_cdev, dummy_dev, 1);
    return 0;
}
```

It adds the character device to the Linux Kernel

```
int cdev_add( struct cdev *p, dev_t dev, unsigned  count);
```

where:
`p` is the character device structure already initialized;
`dev` is the major number for the device;
and `count` is number of minor numbers for which the device is responsible.

Linux Kernel Modules: The Initialization Function

```
static dev_t dummy_dev;

struct cdev dummy_cdev;

static char buffer[64];

struct file_operations dummy_fops = {
    .owner = THIS_MODULE,
    .read = dummy_read,
};

static int __init dummy_module_init(void)
{
    printk(KERN_INFO "Loading dummy_module\n");
    alloc_chrdev_region(&dummy_dev, 0, 1, "dummy_dev");
    printk(KERN_INFO "%s\n", format_dev_t(buffer, dummy_dev));
    cdev_init(&dummy_cdev, &dummy_fops);
    dummy_cdev.owner = THIS_MODULE;
    cdev_add(&dummy_cdev, dummy_dev, 1);
    return 0;
}
```

It indicates the function terminated correctly.
A non zero value indicates an error occurred.

Linux Kernel Modules: The Clean-up Function

```
static void __exit dummy_module_cleanup(void)
{
    printk(KERN_INFO "Cleaning-up dummy_dev.\n");

    cdev_del(&dummy_cdev);
    unregister_chrdev_region(dummy_dev, 1);
}
```

Linux Kernel Modules: The Clean-up Function

```
static void __exit dummy_module_cleanup(void)
{
    printk(KERN_INFO "Cleaning-up dummy_dev.\n");

    cdev_del(&dummy_cdev);
    unregister_chrdev_region(dummy_dev, 1);
}
```

It removes the character device from the Linux kernel.

It frees the range of major/minor numbers previously registered.

Linux Kernel Modules: Custom VFS Functions

```
ssize_t dummy_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
    printk(KERN_INFO "Dummy read (count=%d, offser=%d)\n", (int)count, (int)*f_pos );

    return 1;
}
```

Linux Kernel Modules: Custom VFS Functions

```
ssize_t dummy_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
    printk(KERN_INFO "Dummy re                *d, offser=%d)\n", (int)count, (int)*f_pos );

    return 1;
}
```

Implementation of the VFS function to read from a file, where
`filp` is the pointer to the data structure describing the opened file;
`buf` is the bufer to fill with the data read from the file;
`count` is the size of the buffer, and
`f_pos` is the current reading position in the file.

Linux Kernel Modules: Custom VFS Functions

```
ssize_t dummy_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
    printk(KERN_INFO "Dummy read (count=%d, offser=%d)\n", (int)count, (int)*f_pos );

    return 1;
}
```

It prints a message to show that the read functions has been executed.

It returns the number of byte read from the file.
Returning 0 will block the caller application, which will wait until at least one byte is returned.