1. Linux Device driver

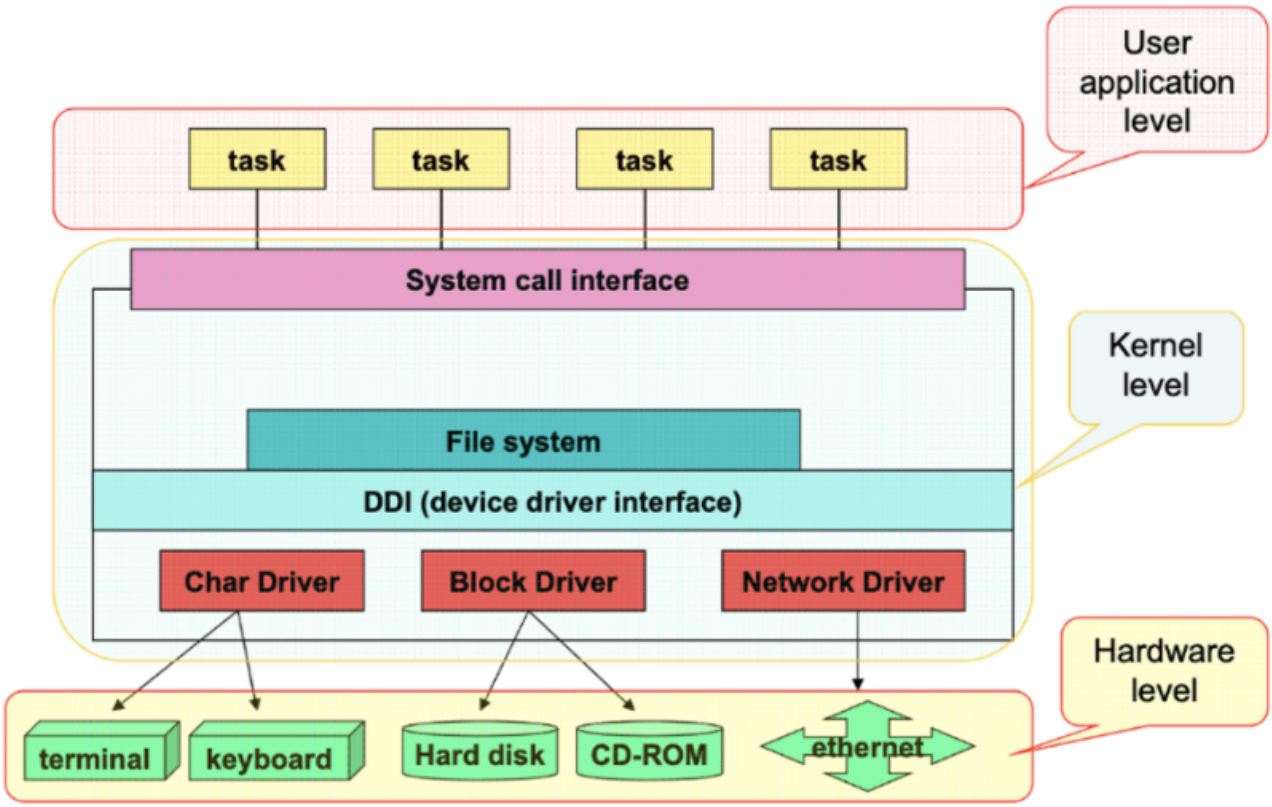
NXKR_YoungSikYang

Exported on 03/06/2024

Table of Contents

1. Types of devices in Linux

Linux kernel diagram including device drivers



1.1 Character Devices

- **Description**: Character devices, also known as "char devices," handle data one character at a time.
- **Use Cases**: They are typically used for devices that require sequential access, such as keyboards, mice, serial ports, and more.
- **Access**: Accessed through files in the `/dev` directory. Examples include `/dev/tty` for the terminal, `/dev/null`, and `/dev/random`.

1.2 Block Devices

- **Description**: Block devices handle data in blocks, which means they read and write data in fixed-size chunks. This allows for random access to data blocks, enabling users to jump to different locations on the device.
- **Use Cases**: Commonly used for storage devices, such as hard drives, SSDs, and USB flash drives.

- **Access**: Accessed through files in the `/dev` directory. Examples include `/dev/sda` for the first SATA drive, `/dev/nvme0n1` for the first NVMe drive, etc.

1.3 Network Devices

- **Description**: Network devices are used to send and receive data packets over a network.
- **Use Cases**: Examples include Ethernet adapters, Wi-Fi cards, and other interfaces that facilitate network communication.
- **Access**: Network interfaces are listed under /sys/class/net/ or can be seen using the ip addr command. They can be interacted with through the utilities like `ifconfig`, `ip`, `netstat`, and others.

2. Misc Driver

Misc drivers (miscellaneous drivers) in Linux are a category of device drivers that don't fit well into other standard categories of drivers like network, USB, or block drivers.

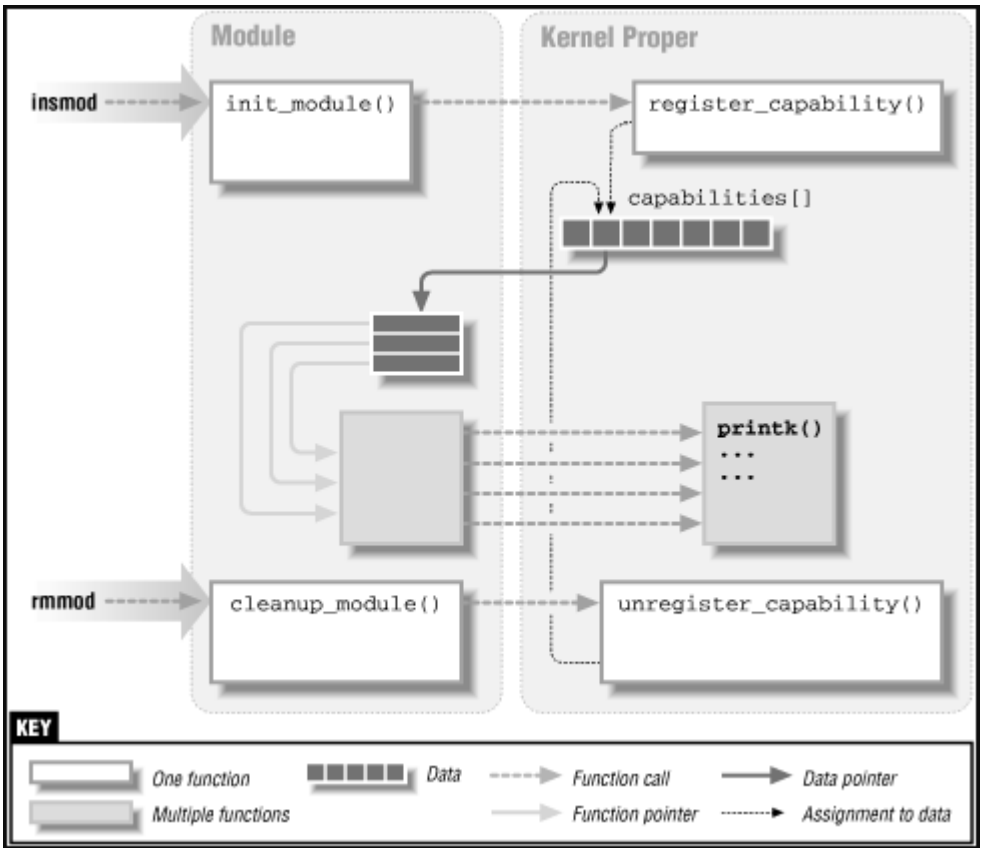
They are used for controlling various types of devices that do not belong to a common device class.

3. Module Driver

3.1 Explanation

- **Definition**: Module drivers refer to any drivers that are compiled as modules in the Linux kernel. This can include drivers for network interfaces, block devices, USB devices, etc.
- **Characteristics**:
    - **Loadable Kernel Modules (LKMs)**: Module drivers can be dynamically loaded into and unloaded from the running kernel, allowing hardware to be added or removed without rebooting the system. (LKM is an option)
    - **Modularity**: This approach supports modularity and extensibility, enabling the kernel to stay lean by loading only the necessary modules for the hardware present.
    - **Maintainability**: Modularization makes it easier to maintain the software. When code is organized into modules, developers can quickly update parts of the application without affecting the entire system.
    - **Tools for Management**: Utilities like `modprobe`, `insmod`, and `rmmod` are used to manage loading and unloading of module drivers.

3.2 Diagram showing how LKMs are loaded



4. Adding a dummy module driver to the kernel

This section describes how to load a loadable kernel module.

4.1 Source code

This is an example code of char driver

my_char.c

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/err.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/err.h>

dev_t dev = 0;
static struct class *dev_class;
static struct cdev my_cdev;

/*
** Function Prototypes
*/
static int      __init my_driver_init(void);
static void     __exit my_driver_exit(void);
static int      my_open(struct inode *inode, struct file *file);
static int      my_release(struct inode *inode, struct file *file);
static ssize_t  my_read(struct file *filp, char __user *buf, size_t len,loff_t *
off);
static ssize_t  my_write(struct file *filp, const char *buf, size_t len, loff_t *
off);

static struct file_operations fops =
{
    .owner      = THIS_MODULE,
    .read       = my_read,
    .write      = my_write,
    .open       = my_open,
    .release    = my_release,
};
```

```
/*
** This function will be called when we open the Device file
*/
static int my_open(struct inode *inode, struct file *file)
{
        pr_info("YANG Driver Open Function Called...!!!\n");
        return 0;
}

/*
** This function will be called when we close the Device file
*/
static int my_release(struct inode *inode, struct file *file)
{
        pr_info("YANG Driver Release Function Called...!!!\n");
        return 0;
}

/*
** This function will be called when we read the Device file
*/
static ssize_t my_read(struct file *filp, char __user *buf, size_t len, loff_t *off)
{
        pr_info("YANG Driver Read Function Called...!!!\n");
        return 0;
}

/*
** This function will be called when we write the Device file
*/
static ssize_t my_write(struct file *filp, const char __user *buf, size_t len, loff_t
*off)
{
        pr_info("YANG Driver Write Function Called...!!!\n");
        return len;
}

/*
** Module Init function
*/
static int __init my_driver_init(void)
{
        /*Allocating Major number*/
        if((alloc_chrdev_region(&dev, 0, 1, "my_Dev")) <0){
                pr_err("Cannot allocate major number\n");
                return -1;
        }
        pr_info("Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));

        /*Creating cdev structure*/
        cdev_init(&my_cdev,&fops);
```

```
        /*Adding character device to the system*/
        if((cdev_add(&my_cdev,dev,1)) < 0){
            pr_err("Cannot add the device to the system\n");
            goto r_class;
        }

        /*Creating struct class*/
        if(IS_ERR(dev_class = class_create(THIS_MODULE,"my_class"))){
            pr_err("Cannot create the struct class\n");
            goto r_class;
        }

        /*Creating device*/
        if(IS_ERR(device_create(dev_class,NULL,dev,NULL,"my_device"))){
            pr_err("Cannot create the Device 1\n");
            goto r_device;
        }
        pr_info("yang's char Device Driver Insert...Done!!!\n");
      return 0;

r_device:
        class_destroy(dev_class);
r_class:
        unregister_chrdev_region(dev,1);
        return -1;
}

/*
** Module exit function
*/
static void __exit my_driver_exit(void)
{
        device_destroy(dev_class,dev);
        class_destroy(dev_class);
        cdev_del(&my_cdev);
        unregister_chrdev_region(dev, 1);
        pr_info("yang's char Device Driver Remove...Done!!!\n");
}

module_init(my_driver_init);
module_exit(my_driver_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("your name");
MODULE_DESCRIPTION("Simple char device driver (File Operations)");
MODULE_VERSION("1.3");
```

4.2 Build the module driver

A module driver can be built using one of the 3 methods below.

4.2.1 Compiling the module along with the kernel

The source(my_char.c) is located in drivers/char.

4.2.1.1 Optional compile according to Kconfig

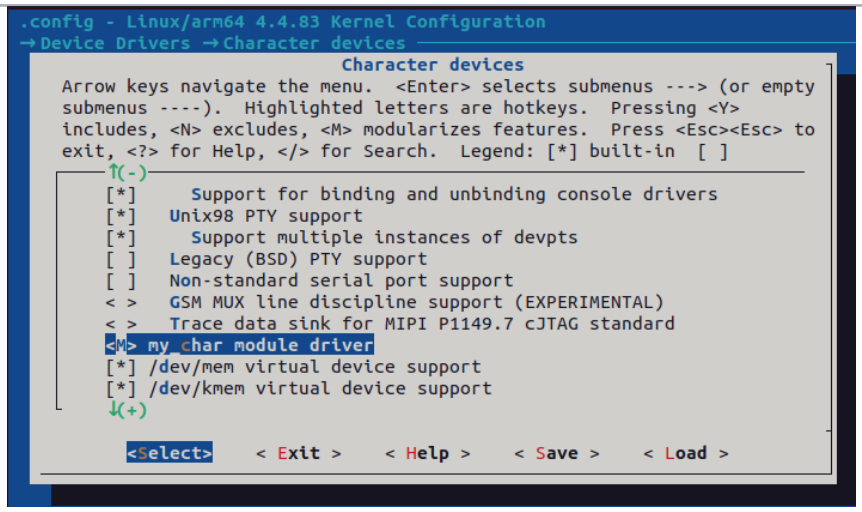
In this case, the module built can be included and excluded optionally via menuconfig

1. Edit `drivers/char/Kconfig`

```
config MY_CHAR
        tristate "my_char module driver"
        default m
        help
                "my_char module driver"
```

2. Modify and save menuconfig

```
make ARCH=arm64 menuconfig
```




```
make ARCH=arm64 savedefconfig
cp defconfig ./arch/arm64/configs/s5p6818_bitminer_defconfig
```

3. Modify drivers/char/Makefile

```
obj-$(CONFIG_MY_CHAR) += my_char.o
```

4.2.1.2 Always compile along with the kernel build

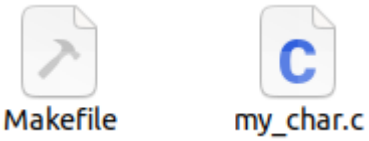
In this case, the module is always built along with the kernel.

Modify `drivers/char/Makefile`

```
obj-m += my_char.o
```

4.2.2 Manual build

1. Create this initial directory



2. Write Makefile

```
obj-m += my_char.o
KDIR = ~/work/dunfell-bitminer/sources/kernel/kernel-4.4.x
all:
    make -C $(KDIR)  M=$(shell pwd) modules ARCH=arm64
clean:
    make -C $(KDIR)  M=$(shell pwd) clean
```

3. Build

```
sudo make
```

4.3 Loading the module driver

Install the built module driver(.ko) to /home/root as described below.

4.3.1 If the module was built along with the kernel

1. Find where the module is

```
find / -type f -name "my_char.ko"
```

```
/lib/modules/4.4.83/kernel/drivers/char/my_char.ko
```

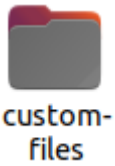
2. Load the module

```
insmod /lib/modules/4.4.83/kernel/drivers/char/my_char.ko
```

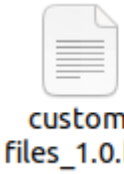
4.3.2 If the module was manually built

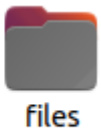
4.3.2.1 Using the yocto recipe

1. Create a recipe directory in recipes-core(or any recipes- directory)



2. Create a recipe and a `files` directory and add necessary files in the `files` directory





3. Write the recipe

```
DESCRIPTION = "Install custom file to /home/root"
LICENSE = "MIT"

LIC_FILES_CHKSUM = "file://${COREBASE}/meta/
COPYING.MIT;md5=3da9cfbcb788c80a0384361b4de20420"

SRC_URI = "file://my_char.ko"

do_install() {
    install -d ${D}/home/root  # Create the directory if it doesn't exist
    install -m 0644 ${WORKDIR}/my_char.c ${D}/home/root/my_char.ko
}

# package management system
FILES_${PN} += "/home/root/my_char.ko"
```

4. Add the recipe to the core recipe

```
IMAGE_INSTALL:append = "\
    custom-files \
```

4.3.2.2 Pushing the module to the target board

```
adb push my_char.ko /home/root
```

```
es/kernel/kernel-4.4.x/drivers/char$ adb push my_char.ko /home/root
my_char.ko: 1 file pushed. 9.9 MB/s (246416 bytes in 0.024s)
```

4.3.2.3 Load the module driver in the target board

```
insmod my_char.ko
```

```
root@s5p6818:~# insmod my_char.ko
[ 2008.780000] Major = 247 Minor = 0
[ 2008.780000] YANG Device Driver Insert...Done!!!
```

"rmmod my_char.ko" will remove the added module.

Reference

- https://embetronicx.com/tutorials/linux/device-drivers