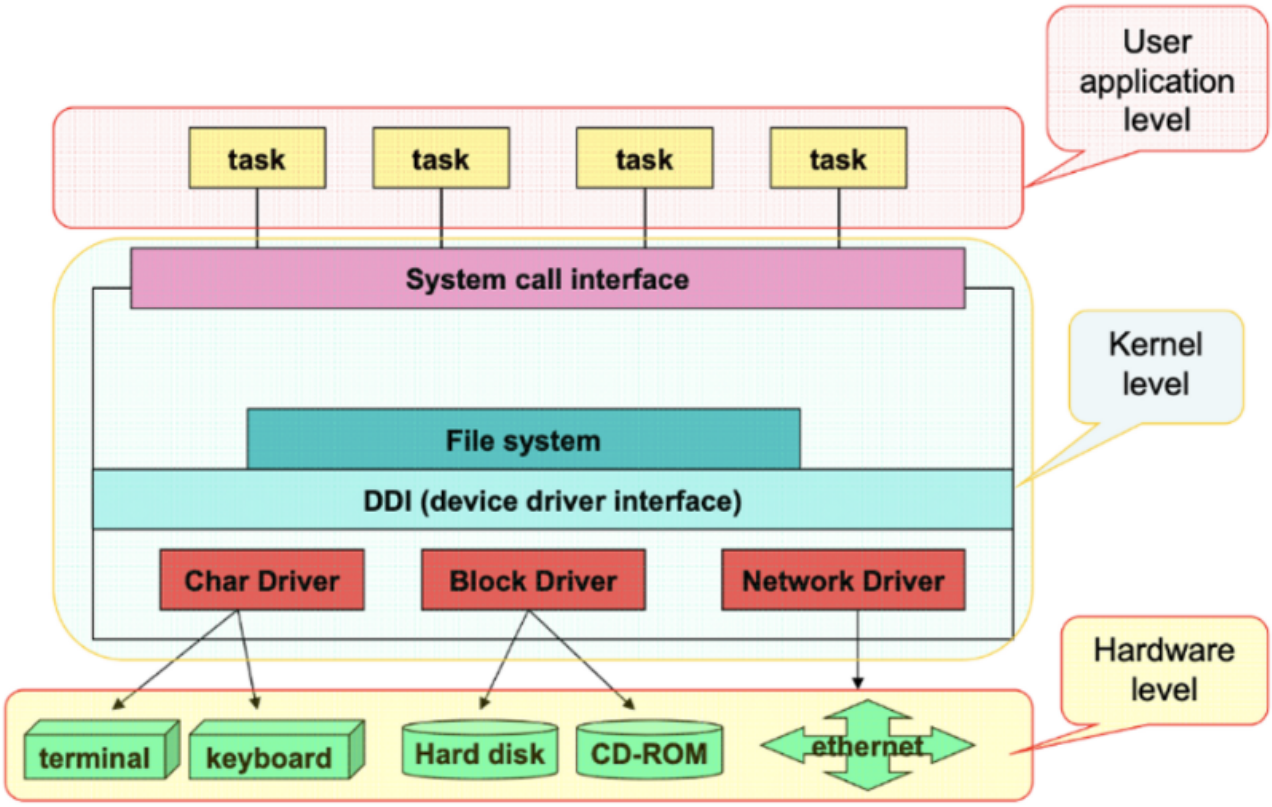
1. Linux Device driver

NXKR_YoungSikYang

Exported on 02/27/2024

Table of Contents

1 1. Types of devices in Linux



1.1 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 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 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 2. Misc Drivers

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 3. Module Drivers

- **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)**: These drivers can be dynamically loaded into and unloaded from the running kernel, allowing hardware to be added or removed without rebooting the system.
    - **Modularity**: This approach supports modularity and extensibility, enabling the kernel to stay lean by loading only the necessary modules for the hardware present.
    - **Dependency Handling**: The kernel keeps track of dependencies between modules, ensuring that modules required by others are loaded first.
    - **Tools for Management**: Utilities like `modprobe`, `insmod`, and `rmmod` are used to manage loading and unloading of module drivers.

4 4. Adding a dummy driver to the kernel

The dummy driver is a module driver

4.1 4.1 char driver source code

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 etx_cdev;

/*
** Function Prototypes
*/
static int      __init etx_driver_init(void);
static void     __exit etx_driver_exit(void);
static int      etx_open(struct inode *inode, struct file *file);
static int      etx_release(struct inode *inode, struct file *file);
static ssize_t  etx_read(struct file *filp, char __user *buf, size_t len,loff_t *
off);
static ssize_t  etx_write(struct file *filp, const char *buf, size_t len, loff_t *
off);

static struct file_operations fops =
{
    .owner      = THIS_MODULE,
    .read       = etx_read,
    .write      = etx_write,
    .open       = etx_open,
    .release    = etx_release,
};

/*
```

```
** This function will be called when we open the Device file
*/
static int etx_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 etx_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 etx_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 etx_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 etx_driver_init(void)
{
        /*Allocating Major number*/
        if((alloc_chrdev_region(&dev, 0, 1, "etx_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(&etx_cdev,&fops);

        /*Adding character device to the system*/
```

```
        if((cdev_add(&etx_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,"etx_class"))){
            pr_err("Cannot create the struct class\n");
            goto r_class;
        }

        /*Creating device*/
        if(IS_ERR(device_create(dev_class,NULL,dev,NULL,"etx_device"))){
            pr_err("Cannot create the Device 1\n");
            goto r_device;
        }
        pr_info("YANG 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 etx_driver_exit(void)
{
        device_destroy(dev_class,dev);
        class_destroy(dev_class);
        cdev_del(&etx_cdev);
        unregister_chrdev_region(dev, 1);
        pr_info("Device Driver Remove...Done!!!\n");
}

module_init(etx_driver_init);
module_exit(etx_driver_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com>");
MODULE_DESCRIPTION("Simple Linux device driver (File Operations)");
MODULE_VERSION("1.3");
```

4.2 4.2 Build the module driver

Choose one of the 3 methods below.

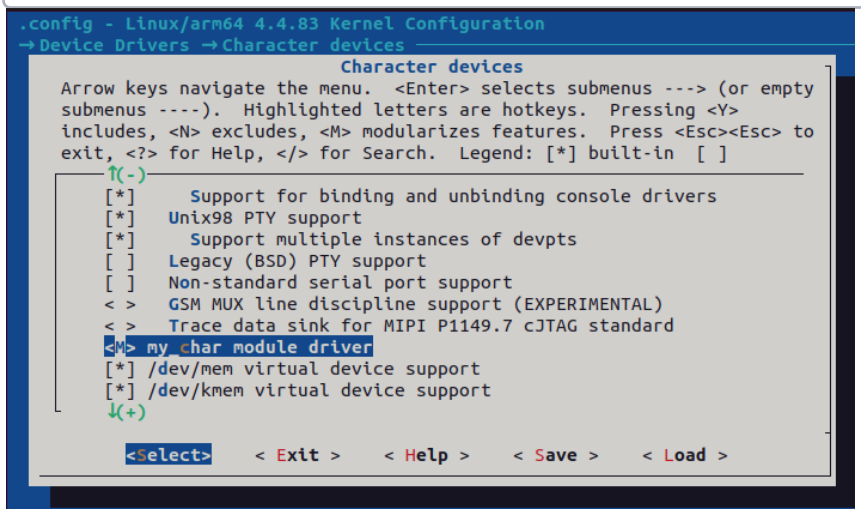
4.2.1 4.2.1 Optional module driver

- The driver source code is in `drivers/char`
- This module driver build can be included and excluded optionally via menuconfig
- Built module drivers are located in `drivers/char`

`drivers/char/Kconfig`

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

```
make ARCH=arm64 menuconfig
```




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

`drivers/char/Makefile`

obj-$(CONFIG_MY_CHAR) += my_char.o

4.2.2 4.2.2 Always compiled on yocto build

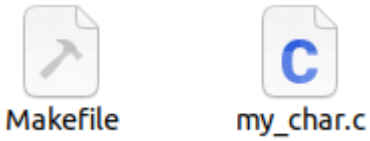
- The driver source code is in `drivers/char`
- This module driver is always included when the kernel is built
- Built module drivers are located in `drivers/char`

Modify `drivers/char/Makefile`

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

4.2.3 4.3.3 Manual build

1. Create this initial directory



2. Write Makefile

```
obj-m += my_char.o
KDIR = /home/youngsik/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 4.3 Transfer the built module driver to the target board

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

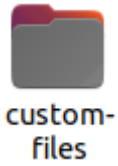
4.3.1 4.3.1 Through adb

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

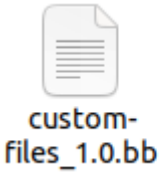
```
es/kernel/kernel-4.4.x/drivers/char$ adb push my_char.ko /home/root
my_char.ko: 1 file pushed. 9.9 MB/s (246416 bytes in 0.024s)
```

4.3.2 4.3.2 Through yocto

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 \
```

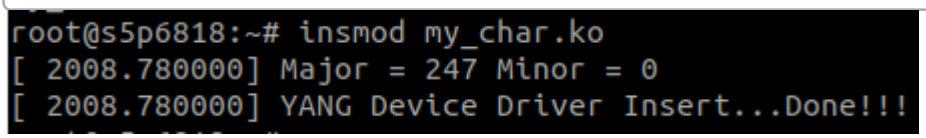
4.3.3 4.3.3 Automatically added

```
find / -type f -name "my_char*"
```

The ko files(module drivers) are automatically added to /lib/modules/4.4.83/kernel/drivers/ when the kernel is built

4.4 4.4 Load the module driver in the target board

```
insmod my_char.ko
```



(rmmod my_char.ko) removes the added module driver.

5 Reference

- https://embetronicx.com/tutorials/linux/device-drivers[1]

1 https://embetronicx.com/tutorials/linux/device-drivers/misc-device-driver/