1. Linux Device driver

NXKR\_YoungSikYang

Exported on 2024-02-23 11:27:24

Table of Contents

1 Types of devices in Linux 3

1.1 Character Devices 3

1.2 Block Devices 3

1.3 Network Devices 3

2 Misc Drivers 5

3 Module Drivers 6

4 Adding a dummy (module) driver to the kernel 7

4.1 char driver source code 7

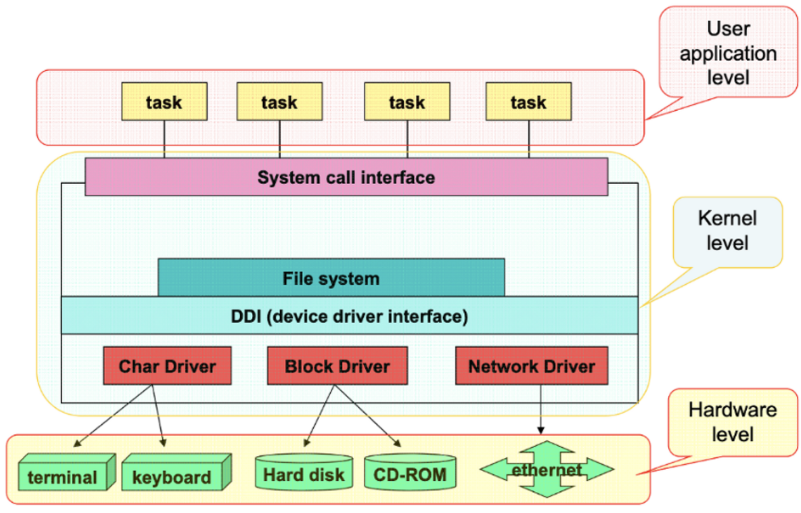
4.2 Include the new driver in the build process 9

4.3 Transfer the module driver to the target board 10

4.4 Load the module driver in the target board 10

5 Reference 11

# Types of devices in Linux



## 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.

## 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.

## 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.

# 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.

# Module Drivers

Embedded Linux - Kernel modules and device drivers, Part 2 - EDN

* **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.

# Adding a dummy (module) driver to the kernel

## char driver source code

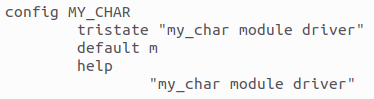
|  |
| --- |
| #include <linux/kernel.h>  #include <linux/init.h>  #include <linux/module.h>  #include <linux/kdev\_t.h>  #include <linux/fs.h>  #include <linux/err.h>  #include <linux/cdev.h>  #include <linux/device.h>  #include <linux/err.h>    dev\_t dev = 0;  **static** **struct** **class** \*dev\_class;  **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"); |

Code Block 1 my\_char.c

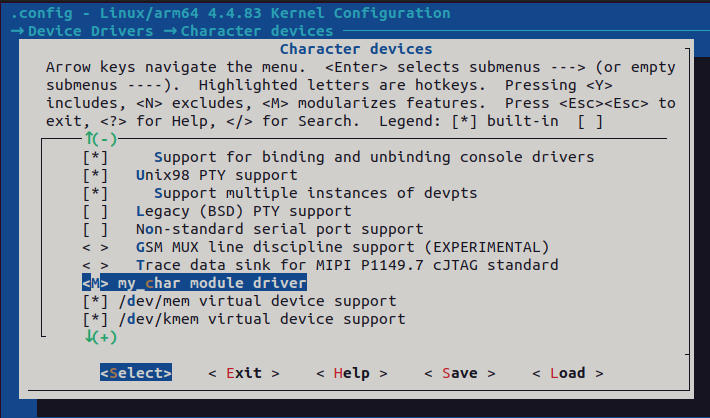
## Include the new driver in the build process

**- Optional module driver that can be included and excluded via menuconfig**

drivers/char/Kconfig



|  |
| --- |
| 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

**- Always compiled together on build**

obj-m        += my\_char.o

And then build.

## Transfer the module driver to the target board

|  |
| --- |
| adb push my\_char.ko /home/root |



## Load the module driver in the target board

|  |
| --- |
| insmod my\_char.ko |



# Reference

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