Embedded Systems Programming
Device drivers

Iwona Kochońska

Gdansk University of Technology
Device drivers

- **Kernel device drivers** - mechanism through which the underlying hardware is exposed to the rest of the system.
- how device drivers fit into the overall architecture?
- how to access device drivers from user space programs?
The role of device drivers

- Kernel encapsulates the many hardware interfaces of a computer system and present them in a consistent manner to user-space programs.

- **Device driver**
  - framework designed to make it easy to write the interface logic for a device in the kernel
  - piece of code that mediates between the kernel above it and the hardware below.

- **A device driver controls**
  - physical devices such as a UART or an MMC controller,
  - virtual devices such as the null device (/dev/null) or a ramdisk.

- One driver may control multiple devices of the same kind.
The role of device drivers

- Kernel device driver code runs at a high privilege level, as does the rest of the kernel.
- It has full access to the processor address space and hardware registers.
- It can handle interrupts and DMA transfers.
- It can make use of the sophisticated kernel infrastructure for synchronization and memory management.
  - if something goes wrong in a buggy driver, it can go really wrong and bring the system down.
  - principle: device drivers should be as simple as possible, just providing information to applications where the real decisions are made. No policy in the kernel!
Types of device drivers

In Linux, there are three main types of device driver:

- **character**: for unbuffered I/O with a rich range of functions and a thin layer between the application code and the driver. It is the first choice when implementing custom device drivers.

- **block**: has an interface tailored for block I/O to and from mass storage devices. There is a thick layer of buffering designed to make disk reads and writes as fast as possible, which makes it unsuitable for anything else.

- **network**: similar to a block device but is used for transmitting and receiving network packets rather than disk blocks.

There is also a fourth type that presents itself as a group of files in one of the pseudo filesystems.

- **example**: access the GPIO driver through a group of files in /sys/class/gpio
Character devices

- Device driver associated with a device node, which has major and minor numbers.
- Identified in user space by a filename: if you want to read from a UART, you open the device node, for example, the first serial port on the ARM Versatile Express would be `/dev/ttyAMA0`.
- The driver is identified in the kernel, using the major number which, in the example given, is 204.
- Since the UART driver can handle more than one UART, there is a second number, called the minor number, which identifies a specific interface, 64, in this case:

```
# ls -l /dev/ttyAMA*

crw-rw---- 1 root root  204,  64 Jan  1 1970 /dev/ttyAMA0
crw-rw---- 1 root root  204,  65 Jan  1 1970 /dev/ttyAMA1
crw-rw---- 1 root root  204,  66 Jan  1 1970 /dev/ttyAMA2
crw-rw---- 1 root root  204,  67 Jan  1 1970 /dev/ttyAMA3
```
Character devices

- The list of standard major and minor numbers can be found in the kernel documentation, in Documentation/devices.txt.

- Device nodes can be created in several ways:
  - **devtmpfs**: The node that is created when the device driver registers a new device interface using a base name supplied by the driver (ttyAMA) and an instance number.
  - **udev or mdev (without devtmpfs)**: Essentially the same as with devtmpfs, except that a user-space daemon program has to extract the device name from sysfs and create the node.
  - **mknod**: If you are using static device nodes, they are created manually using mknod.
Character devices

- From Linux 2.6 onwards, the major number is 12 bits long, which gives valid numbers from 1 to 4,095, and the minor number is 20 bits, from 0 to 1,048,575.
- When you open a device node, the kernel checks to see whether the major and minor numbers fall into a range registered by a device driver of that type (a character or block).
- If so, it passes the call to the driver, otherwise the open call fails.
- The device driver can extract the minor number to find out which hardware interface to use.
- If the minor number is out of range, it returns an error.
Character devices

Device driver is not the same as a file: the things you do with it change the state of the device.

- every time we read from device named urandom, it returns a fresh set of pseudo random data,
- we don’t need to know anything else about it.
- We can just use normal functions such as open(2), read(2), and close(2).

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(void)
{
    int f;
    unsigned int rnd;
    int n;
    f = open("/dev/urandom", O_RDONLY);
    if (f < 0) {
        perror("Failed to open urandom");
        return 1;
    }
    n = read(f, &rnd, sizeof(rnd));
    if (n != sizeof(rnd)) {
        perror("Problem reading urandom");
        return 1;
    }
    printf("Random number = 0x%x\n", rnd);
    close(f);
    return 0;
}
```
Character devices

Why not use the stream I/O functions fopen(3), fread(3), and fclose(3) instead of open, read, close?

- buffering implicit in these functions often causes unexpected behavior.
- For example, fwrite(3) usually only writes to the user-space buffer, not to the device. We would need to call fflush(3) to force the buffer to be written out.

Don’t use stream I/O functions such as fread(3) and fwrite(3) when calling device drivers!
Block devices

- Device driver associated with a device node, which has major and minor numbers.

- Although character and block devices are identified using major and minor numbers, they are in different namespaces. A character driver with a major number 4 is in no way related to a block driver with a major number 4.

- Major number - to identify the device driver

- Minor number - to identify the partition.
Block devices

- Example: the MMC driver

```bash
# ls -l /dev/mmcblk*
```

- The major number is 179
- The minor numbers are used in ranges to identify different mmc devices and the partitions of the storage medium on that device.
- In the case of the mmcblk driver, the ranges are eight minor numbers per device: the minor numbers from 0 to 7 are for the first device, the numbers from 8 to 15 are for the second, and so on.
- Within each range, the first minor number represents the entire device as raw sectors, and the others represent up to seven partitions.
Block devices

- SCSI disk driver - known as sd
  - used to control a range of disks that use the SCSI command set, which includes SCSI, SATA, USB mass storage, and UFS (universal flash storage).
  - It has the major number eight and ranges of 16 minor numbers per interface (or disk).
  - The minor numbers from 0 to 15 are for the first interface, with device nodes named sda up to sda15, the numbers from 16 to 31 are for the second disk with device nodes sdb up to sdb15, and so on.
  - This continues up to the sixteenth disk from 240 to 255, with the node name sdp.
  - There are other major numbers reserved for them because SCSI disks are so popular, but we needn’t worry about that here.
Block devices

- The partitions are created using utilities such as `fdisk`, `sfidsk`, or `parted`.
- An exception is **raw flash memory**: the partition information for the MTD driver is part of the kernel command line or in the device tree.
- A user-space program can open and interact directly with a block device through the device node.
- This is not a common thing to do, and is usually for performing administrative operations such as partitioning, formatting with a filesystem, and mounting.
- Once the filesystem is mounted, you interact with the block device indirectly through the files in that filesystem.
Network devices

- Not accessed through device nodes
- Do not have major and minor numbers.
- Allocated a name by the kernel, based on a string and an instance number.
- Example: create a network device named `net0` the first time it is called, `net1` the second, and so on. More common names are `lo`, `eth0`, and `wlan0`.

```c
my_netdev = alloc_netdev(0, "net%d", NET_NAME_UNKNOWN, netdev_setup);
ret = register_netdev(my_netdev);
```
Network devices

- Usually, the network interface name is only used when configuring the network using utilities such as `ip` and `ifconfig` to establish a network address and route.

- Interacting with the network driver: indirectly by opening sockets (network layer decides how to route them to the right interface).

- It is possible to access network devices directly from user space - by creating a socket and using the `ioctl` commands listed in `include/linux/sockios.h`. 
Network devices

Example - use of SIOCGIFHWADDR to query the driver for the hardware (MAC) address:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <linux/sockios.h>
#include <net/if.h>
int main (int argc, char *argv[]) {  
  int s;
  int ret;
  struct ifreq ifr;
  int i;
  if (argc != 2) {  
    printf("Usage %s [network interface]\n", argv[1]);  
    return 1;
  }
  s = socket(PF_INET, SOCK_DGRAM, 0);
  if (s < 0) {  
    perror("socket");  
    return 1;
  }
  strcpy(ifr.ifr_name, argv[1]);
  ret = ioctl(s, SIOCGIFHWADDR, &ifr);
  if (ret < 0) {  
    perror("ioctl");  
    return 1;
  }
  for (i = 0; i < 6; i++)  
    printf("%02x:\n", (unsigned char)ifr.ifr_hwaddr.sa_
  printf("\n");
  close(s);
  return 0;
}
```
Finding out about drivers at runtime

Which device drivers are loaded and what state they are in? Reading the files in /proc and /sys:

- list the character and block device drivers currently loaded and active by reading /proc/devices:

```
iwona@iwona2:/$ cat /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  6 lp
  7 vcs
 10 misc
 13 input
 21 sg
 29 fb
 81 video4linux
 99 ppdev
116 alsa

Block devices:
  259 blkdev
    8 sd
   11 sr
   65 sd
   66 sd
   67 sd
   68 sd
   69 sd
   70 sd
   71 sd
   128 sd
   129 sd
   130 sd
   131 sd
   132 sd
   133 sd
```
Finding out about drivers at runtime

- list the network device drivers:

  ```
  iwona@iwona2:/$ ip link show
  1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default
      link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
  2: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc mq state DOWN mode DEFAULT group default
      link/ether 88:ae:1d:d9:5a:6d brd ff:ff:ff:ff:ff:ff
  3: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT group default
      link/ether 00:26:82:ee:62:94 brd ff:ff:ff:ff:ff:ff
  4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mode DEFAULT group default
      link/ether 02:42:64:bc:73:08 brd ff:ff:ff:ff:ff:ff
  ```

- find out about devices attached to USB or PCI buses: `lsusb` and `lspci` commands.
Getting information from sysfs

- **sysfs** - representation of kernel objects, attributes and relationships.
  - kernel object = a directory
  - attribute = a file
  - relationship = symbolic link from one object to another.

- All devices and drivers are represented as kernel objects.

```bash
iwona@iwona2:$ ls /sys
block  class  devices  fs           kernel  power
bus     dev     firmware  hypervisor  module
```
Getting information from sysfs

/sys/devices

- the kernel’s view of the devices discovered since boot and how they are connected to each other
- organized at the top level by the system bus

iwona@iwona2:$ ls /sys/devices
breakpoint  LNXSYSTM:00  platform  software  tracepoint
cpu          pci0000:00  pnp0       system  virtual
Getting information from sysfs

/sys/devices

- Three directories are present on all systems:
  - **system**: devices at the heart of the system, including CPUs and clocks.
  - **virtual**: memory-based devices
    - /dev/null, /dev/random, /dev/zero in virtual/mem
    - loopback device, lo, in virtual/net.
  - **platform**: a catch-all for devices that are not connected via a conventional hardware bus.
    - may be almost everything on an embedded device.
    - Example: the PCI root bus appears as pci0000:00.
Getting information from /sys/class

- **/sys/class** a view of the device drivers presented **by their type** (a software view)
- Each of the subdirectories represents a class of driver and is implemented by a component of the driver framework.
- **Examples**
  - UART devices in /sys/class/tty
  - network devices in /sys/class/net
  - input devices in /sys/class/input
- There is a symbolic link in each subdirectory for each instance of that type of device pointing to its representation in /sys/device.
Getting information from `/sys/class`

**/sys/class**

```
# cd /sys/class/tty/ttyAMA0/
# ls
close_delay flags line       uartclk
closing_wait io_type port uevent
custom_divisor iomem_base power xmit_fifo_size
dev iomem_reg_shift subsystem
device irq type
```

- **link** device references the hardware node for the device
- **subsystem** points back to `/sys/class/tty`
- **others** - attributes of the device
- some attribute files are writable and allow you to tune parameters in the driver at runtime.
- **The dev attribute** - major and minor numbers of this device.

```
# cat /sys/class/tty/ttyAMA0/dev
204:64
```
Getting information from `/sys/block`

`/sys/block` - subdirectories for each block device

```bash
# ls /sys/block/
loop0  loop4  mmcblk0  ram0  ram12  ram2  ram6
loop1  loop5  mmcblk1  ram1  ram13  ram3  ram7
loop2  loop6  mmcblk1boot0  ram10  ram14  ram4  ram8
loop3  loop7  mmcblk1boot1  ram11  ram15  ram5  ram9

# cd /sys/block/mmcblk1
# ls
alignment_offset  ext_range  mmcblk1p1  ro
bdi                force_ro  mmcblk1p2  size
capability         holders   power     slaves
dev                inflight  queue     stat
device             mmcblk1boot0  range    subsystem
discard_alignment  mmcblk1boot1  removable  uevent
```
Finding the right device driver

- Simple cases - user space code to handle GPIO
- Kernel drivers
  - check your kernel (menuconfig and search for the product name or number)
  - driver support page on the manufacturer’s website
  - ask the manufacturer directly
  - search online and ask in the relevant forums
Device drivers in user-space - GPIO

- **General Purpose Input/Output (GPIO)** - the simplest form of digital interface since

- Gives a direct access to individual hardware pins, each of which can be configured as input or output.

- Can even be used to create higher level interfaces such as I2C or SPI by manipulating each bit in the software, a technique that is called **bit banging**.
The main limitation is the **speed** and **accuracy** of the **software loops** and the **number of CPU cycles** you want to dedicate to them.

- It is hard to achieve timer accuracy better than a millisecond with kernels compiled with `CONFIG_PREEMPT`, and 100 microseconds with `RT_PREEMPT`.

Common use cases for GPIO:

- reading push buttons and digital sensors
- controlling LEDs, motors, and relays.
Most SoCs have a lot of GPIO bits which are grouped together in GPIO registers, usually 32 bits per register.

On-chip GPIO bits are routed through to GPIO pins on the chip package via a multiplexer, known as a pin mux, which I will describe later.

There may be additional GPIO bits available off-chip in the power management chip, and in dedicated GPIO extenders, connected through I2C or SPI buses.

All this diversity is handled by a kernel subsystem known as gpiolib, which is not actually a library but the infrastructure GPIO drivers use to expose IO in a consistent way.

Details about the implementation of gpiolib in the kernel source in Documentation/gpio and the drivers themselves are in drivers/gpio.
Applications can interact with gpiolib through files in the /sys/class/gpio directory. Example:

```
# ls /sys/class/gpio
export gpiochip0 gpiochip32 gpiochip64 gpiochip96 unexport
```

The gpiochip0 to gpiochip96 directories represent four GPIO registers, each with 32 GPIO bits.

```
# ls /sys/class/gpio/gpiochip96/
based label ngpio power subsystem uevent
```

- **base** contains the number of the first GPIO pin in the register
- **ngpio** contains the number of bits in the register
GPIO

To control a GPIO bit from user space:

- Export it from kernel space, which you do by writing the GPIO number to `/sys/class/gpio/export`.

  ```
  # echo 48 > /sys/class/gpio/export
  # ls /sys/class/gpio
  export  gpio48  gpiochip0  gpiochip32  gpiochip64  gpiochip96  unexport
  ```

- New directory, gpio48, contains the files to control the pin:

  ```
  # ls /sys/class/gpio/gpio48
  active_low  direction  edge  power  subsystem  uevent  value
  ```

- The pin begins as an input. To change it to an output, write out to the `direction` file.
  - File value contains the current state of the pin, which is 0 for low and 1 for high.
  - If it is an output, you can change the state by writing 0 or 1 to value.
  - Sometimes, the meaning of low and high is reversed in hardware.

- Remove a GPIO from user space control - writing the GPIO number to `/sys/class/gpio/unexport`.

Handling interrupts from GPIO

- GPIO input can be configured to generate an interrupt when it changes state.
- If the GPIO bit can generate interrupts, the file edge exists. Initially, it has the value `none`, meaning that it does not generate interrupts.
- To enable interrupts, set it to one of these values:
  - `rising`: Interrupt on rising edge
  - `falling`: Interrupt on falling edge
  - `both`: Interrupt on both rising and falling edges
  - `none`: No interrupts (default)
Handling interrupts from GPIO

▶ If you want to wait for a rising edge on GPIO 48, you first enable interrupts:

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <poll.h>

int main (int argc, char *argv[])
{
    int f;
    struct pollfd poll_fds [1];
    int ret;
    char value[4];
    int n;
    f = open("/sys/class/gpio/gpio48", 0_RDONLY);
    if (f == -1) {
        perror("Can't open gpio48");
        return 1;
    }
    poll_fds[0].fd = f;
    poll_fds[0].events = POLLPRI | POLLE误;
    while (1) {
        printf("Waiting\n");
        ret = poll(poll_fds, 1, -1);
        if (ret > 0) {
            n = read(f, &value, sizeof(value));
            printf("Button pressed: read %d bytes, value=%c\n", n, value[0]);
        }
    }
    return 0;
}
```

▶ Then, you use `poll()` to wait for the change (POLLPRI argument)
The **leds** kernel subsystem

- adds the ability to set brightness,
- can handle LEDs connected in other ways than a simple GPIO pin.
- can be configured to trigger the LED on an event (such as block device access or just a heartbeat to show that the device is working).
- More information in Documentation/leds/ and the drivers are in drivers/leds/.
As with GPIOs, LEDs are controlled through an interface in /sys/class/leds. The LEDs have names in the form `devicename:colour:function`.

```
# ls /sys/class/leds
beaglebone:green:heartbeat  beaglebone:green:usr2
beaglebone:green:mmc0     beaglebone:green:usr3
```

One individual LED:
```
# ls /sys/class/leds/beaglebone:green:usr2
brightness  max_brightness  subsystem  uevent
device      power           trigger
```
The brightness file controls the brightness of the LED, can be a number between 0 (off) and max_brightness (fully on).

If the LED doesn’t support intermediate brightness, any non-zero value turns it on and zero turns it off.

The file trigger lists the events that trigger the LED to turn on.

The list of triggers is implementation-dependent. Example:

```
# cat /sys/class/leds/beaglebone:green:heartbeat/trigger
none mmc0 mmc1 timer oneshot [heartbeat] backlight gpio cpu0 default-on
```

The trigger currently selected is shown in square brackets.
If you set the trigger to timer, two extra files appear that allow you to set the on and off times in milliseconds:

```bash
# echo timer > /sys/class/leds/beaglebone:green:heartbeat/trigger
# ls /sys/class/leds/beaglebone:green:heartbeat
brightness delay_on max_brightness subsystem uevent
delay_off device power trigger
# cat /sys/class/leds/beaglebone:green:heartbeat/delay_on 500
# cat /sys/class/leds/beaglebone:green:heartbeat/delay_off 500
#
```

If the LED has on-chip timer hardware, the blinking takes place without interrupting the CPU.
I2C - a simple low speed 2-wire bus, typically used to access peripherals which are not on the SoC board such as display controllers, camera sensors, GPIO extenders, and the like.

- a master-slave protocol, with the master being one or more host controllers on the SoC.
- slaves have a 7-bit address assigned by the manufacturer allowing up to 128 nodes per bus, but 16 are reserved, so only 112 nodes are allowed in practice.
- The bus speed is 100 KHz in standard mode, or up to 400 KHz in fast mode.
- The protocol allows read and write transactions between the master and slave of up to 32 bytes.
- Frequently, the first byte is used to specify a register on the peripheral and the remaining bytes are the data read from or written to that register.
There is one device node for each host controller:

```
# ls -l /dev/i2c*
<br>
- rw---- 1 root i2c 89, 0 Jan 1 00:18 /dev/i2c-0
- rw---- 1 root i2c 89, 1 Jan 1 00:18 /dev/i2c-1
- rw---- 1 root i2c 89, 2 Jan 1 00:18 /dev/i2c-2
- rw---- 1 root i2c 89, 3 Jan 1 00:18 /dev/i2c-3
```

The device interface provides a series of `ioctl` commands that query the host controller and send read and write commands to I2C slaves.

Package `i2c-tools` uses this interface to provide basic command-line tools to interact with I2C devices:

- **i2cdetect**: lists the I2C adapters and probes the bus
- **i2cdump**: dumps data from all the registers of an I2C peripheral
- **i2cget**: reads data from an I2C slave
- **i2cset**: writes data to an I2C slave
A user space program to talk to the device:

```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <i2c-dev.h>
#include <sys/ioctl.h>
#define I2C_ADDRESS 0x5d
#define CHIP_REVISION_REG 0x10

void main (void)
{
    int f_i2c;
    int val;

    /* Open the adapter and set the address of the I2C device */
    f_i2c = open("/dev/i2c-1", O_RDWR);
    ioctl (f_i2c, I2C_SLAVE, I2C_ADDRESS);

    /* Read 16-bits of data from a register */
    val = i2c_smbus_read_word_data(f, CHIP_REVISION_REG);
    printf("Sensor chip revision \d\n", val);
    close (f_i2c);
}
```
Serial Peripheral Interface bus - similar to I2C, but is a lot faster, up to the low MHz.

- The interface uses four wires with separate send and receive lines which allows it to operate in full duplex.
- Each chip on the bus is selected with a dedicated chip select line.
- Used to connect to touchscreen sensors, display controllers, and serial NOR flash devices.
- As with I2C, it is a master-slave protocol, with most SoCs implementing one or more master host controllers.
- **Generic SPI device driver** can be enabled through the kernel configuration `CONFIG_SPI_SPIDEV`. 
What are the options?

- character drivers are the most flexible and should cover 90% of all your needs,
- network devices apply if you are working with a network interface,
- block devices are for mass storage.

The main character device interface is based on a stream of bytes (as in case of a serial port)

- read(2)
- write(2)
Interfacing character device driver

Other ways to communicate with device drivers:

▶ **ioctl**: allows to pass two arguments to your driver which can have any meaning you like.
  
  ▶ By convention, the **first argument** - a **command** which selects one of several functions in your driver,
  
  ▶ **second argument** - a **pointer to a structure** which serves as a container for the input and output parameters.
  
  ▶ passes all its arguments in a structure in a single function call.
  
  ▶ **deprecated!**

  ▶ The kernel maintainers dislike ioctl because it makes kernel code and application code too interdependent, and it is hard to keep both of them in step across kernel versions and architectures.
Interfacing characet device driver

Other ways to communicate with device drivers:

- **sysfs:**
  - preferred way now,
  - example: GPIO interface
  - It is also scriptable because the file contents are ASCII strings.
  - The requirement for each file to contain a single value makes it hard to achieve atomicity if you need to change more than one value at a time.

- **mmap:** mapping kernel memory into user-space, bypassing the kernel.
  - get direct access to kernel buffers and hardware registers
  - still need some kernel code to handle interrupts and DMA.
Interfacing character device driver

Other ways to communicate with device drivers:

▶ **sigio**: send a signal from a driver using the kernel function `kill_fasync()` to notify applications of an event
  ▶ examples of events: input becoming ready or an interrupt being received.
  ▶ by convention, signal SIGIO is used, but it could be anyone.
  ▶ Examples: UIO driver, drivers/uio/uio.c, and the RTC driver, drivers/char/rtc.c.
  ▶ The main problem is that it is difficult to write reliable signal handlers and so it remains a little-used facility.

▶ **debugfs**: pseudo filesystem that represents kernel data as files and directories, similar to proc and sysfs.
  ▶ it is for debug and trace information only.
  ▶ must not contain information that is needed for the normal operation of the system;
  ▶ It is mounted as `mount -t debugfs debug /sys/kernel/debug`. 
Interfacing character device driver

Other ways to communicate with device drivers:

- **proc**: pseudo filesystem deprecated for all new code unless it relates to processes, which was the original intended purpose for the filesystem.
  - can be used for publishing information
  - unlike sysfs and debugfs, it is available to non-GPL modules.

- **netlink**: a socket protocol family
  - **AF_NETLINK** creates a socket that links kernel space to user-space.
  - originally created so that network tools could communicate with the Linux network code to access the routing tables and other details.
  - also used by **udev** to pass events from the kernel to the udev daemon.
  - very rarely used in general device drivers.
Device driver example - part 1

```c
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/device.h>
#define DEVICE_NAME "dummy"
#define MAJOR_NUM 42
#define NUM_DEVICES 4

static struct class *dummy_class;
static int dummy_open(struct inode *inode, struct file *file)
{
    pr_info("%s\n", __func__);  
    return 0;
}

static int dummy_release(struct inode *inode, struct file *file)
{
    pr_info("%s\n", __func__);  
    return 0;
}
```
Device driver example - part 2

```c
static ssize_t dummy_read(struct file *file,
    char *buffer, size_t length, loff_t * offset)
{
    pr_info("%s %u\n", __func__, length);
    return 0;
}

static ssize_t dummy_write(struct file *file,
    const char *buffer, size_t length, loff_t *offset)
{
    pr_info("%s %u\n", __func__, length);
    return length;
}
```
Device driver example - part 3

```
struct file_operations dummy_fops = {
  .owner = THIS_MODULE,
  .open = dummy_open,
  .release = dummy_release,
  .read = dummy_read,
  .write = dummy_write,
};
```
int __init dummy_init(void)
{
    int ret;
    int i;
    printk("Dummy loaded\n");
    ret = register_chrdev(MAJOR_NUM, DEVICE_NAME, &dummy_fops);
    if (ret != 0)
        return ret;
    dummy_class = class_create(THIS_MODULE, DEVICE_NAME);
    for (i = 0; i < NUM_DEVICES; i++) {
        device_create(dummy_class, NULL,
                      MKDEV(MAJOR_NUM, i), NULL, "dummy%d", i);
    }
    return 0;
}
Device driver example - part 5

```c
void __exit dummy_exit(void)
{
    int i;
    for (i = 0; i < NUM_DEVICES; i++) {
        device_destroy(dummy_class, MKDEV(MAJOR_NUM, i));
    }
    class_destroy(dummy_class);
    unregister_chrdev(MAJOR_NUM, DEVICE_NAME);
    printk("Dummy unloaded\n");
}

module_init(dummy_init);
module_exit(dummy_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Chris Simmonds");
MODULE_DESCRIPTION("A dummy driver");
```
Device driver example - compilation

- **makefile:**

  ```
  LINUXDIR := $(HOME)/MELP/build/linux
  
  obj-m := dummy.o
  all:
      make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf- \
      -C $(LINUXDIR) M=$(shell pwd)
  clean:
      make -C $(LINUXDIR) M=$(shell pwd) clean
  
  LINUXDIR - device kernel directory
  obj-m := dummy.o - compilation of dummy.c to the kernel module
  dummy.ko
  ```