Tuesday, October 17, 2017
9:58 AM

Char Device Driver - Lab1

(Basic Embedded Training - http://hethongnhung.com)

1.Kernel Module 1 Hello World

Hello_printk.c

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>

static int __init my_init(void)
{
     printk(KERN_INFO "hello, my module\n");
     return 0;
}

static void __exit my_exit(void)
{
     printk(KERN_INFO "good bye, my module\n" );
}

module_init(my_init);
module_exit(my_exit);

MODULE_LICENSE("Dual BSD/GPL");
MODULE_AUTHOR("F.B.");
MODULE_DESCRIPTION("My First Driver");
```

Makefile

```
# obj-m is a list of what kernel modules to build. The .o and other
# objects will be automatically built from the corresponding .c file -
# no need to list the source files explicitly.

obj-m := hello_printk.o

# KDIR is the location of the kernel source. The current standard is
# to link to the associated source tree from the directory containing
# the compiled modules.

KDIR := /lib/modules/$(shell uname -r)/build

# PWD is the current working directory and the location of our module
# source files.

PWD := $(shell pwd)

# default is the default make target. The rule here says to run make
# with a working directory of the directory containing the kernel
# source and compile only the modules in the PWD (local) directory.

default:
     (MAKE) -C $(KDIR) M=$(PWD) modules
```

The make command generates several files in the module directory. The resulting kernel module is hello_printk.ko (.ko = "kernel object").
Kernel Module 2 - Char Device shows how to communicate with the kernel module from a user mode process.

2.Kernel Module 2 Char Device

- ***theory*:**

- Major and Minor Numbers
- Char Device Registration

  - difference between these two functions:
    int register_chrdev_region(dev_t first, unsigned int count, char *name);
    int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name);
  - Registration is only really useful if you know in advance which major number you want to start with. With registration, you *tell* the kernel what device numbers you want (the start major/minor number and count) and it either gives them to you or not (depending on availability).
  - With allocation, you tell the kernel how many device numbers you need (the starting minor number and count) and it will find a starting major number for you, if one is available, of course.
  - Partially to avoid conflict with other device drivers, it's considered preferable to use the allocation function, which will *dynamically* allocate the device numbers for you.

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/device.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
```

```c
static char command[256];
static char response[256];
int have_command = 0;

int my_read( struct file *filep, char *buffer, size_t count, loff_t *offp )
{
    int ret;

    if( 0==have_command )
        return 0; // we already have sent a response and wait for a "write command" operation

    if( count > strlen(response) )
        count = strlen(response);

    ret = copy_to_user( buffer, response, strlen(response) );
    if( ret != 0 )
        return -EINVAL;

    have_command = 0;
    return count;
}

int my_write( struct file *filep, const char *buffer, size_t count, loff_t *offp )
{
    int ret;

    if( count > sizeof(command)-1 )
        count = sizeof(command)-1;

    ret = copy_from_user( command, buffer, count );
    if( ret != 0 )
        return -EINVAL;

    command[count] = '\0';

    // parse command here and execute it
    switch( command[0] )
    {
        case 's':
            // TODO: set led
            strcpy( response, "OKd\n" );
            break;
        case 'c':
            // TODO: clear led
            strcpy( response, "OK\n" );
            break;
        default:
            strcpy( response, "ERROR: unknown command\n" );
    }

    have_command = 1;
    return count;
}

static struct file_operations my_fops =
{
    .read = my_read,
    .write = my_write,
};

/* driver major number */
static dev_t my_device;

static int __init my_init(void)
{
    printk(KERN_INFO "my module: init\n");

    /* request device major number */
    if ((ret = alloc_chrdev_region(&my_device, 0, 0, "cse3") < 0)) {
        printk(KERN_DEBUG "Error registering device!\n");
        return ret;
    }

    return 0;
}
```

```
static void __exit my_exit(void)
{
    /* release major number */
    unregister_chrdev_region(my_device, 0);
    printk(KERN_INFO "my module: exit\n" );
}

module_init(my_init);
module_exit(my_exit);

MODULE_LICENSE("Dual BSD/GPL");
MODULE_AUTHOR("F.B.");
MODULE_DESCRIPTION("my char device driver");
```

Test the kernel module by building it (as above) and transferring it to the target, say /home/root/my_module.ko
Login as root on the target and enter:

```
udooer@udooneo:~# sudo insmod my_module.ko
my module: init
udooer@udooneo:~# lsmod
Module                  Size  Used by
my_module               1807  0
...
udooer@udooneo:~# ls -l /dev/cse3
crw------- 1 root root 246, 0 May 16 01:23 /dev/cse3
udooer@udooneo:~# echo "s" > /dev/cse3
udooer@udooneo:~# cat /dev/cse3
OK
udooer@udooneo:~# echo "x" > /dev/cse3
udooer@udooneo:~# cat /dev/cse3
ERROR: unknown command
udooer@udooneo:~# sudo rmmod my_module
my module: exit
```

This simple module has some limitations:

- **open** and **release** functions are not implemented, so the device can be openend serveral times or by several processes. This might be confusing. Hint: implement both functions such that the device can be opened at most once at a time.
- **my_read** returns 0 which means "end of file". This is good for cat, because cat reads as much as it can. One might prefer a different implementation of my_read when the response is read by a user mode app.
- There is no protection against race conditions like if my_read is called during my_write. So you might want to use a mutex for protection.
- Often, kernel modules implement ioctl calls to set/get binary data like a C struct

Kernel Module 3 - GPIO shows how actually to toggle a GPIO pin whenever a read is executed.

3. Kernel Module 3 GPIO

Implementing the module

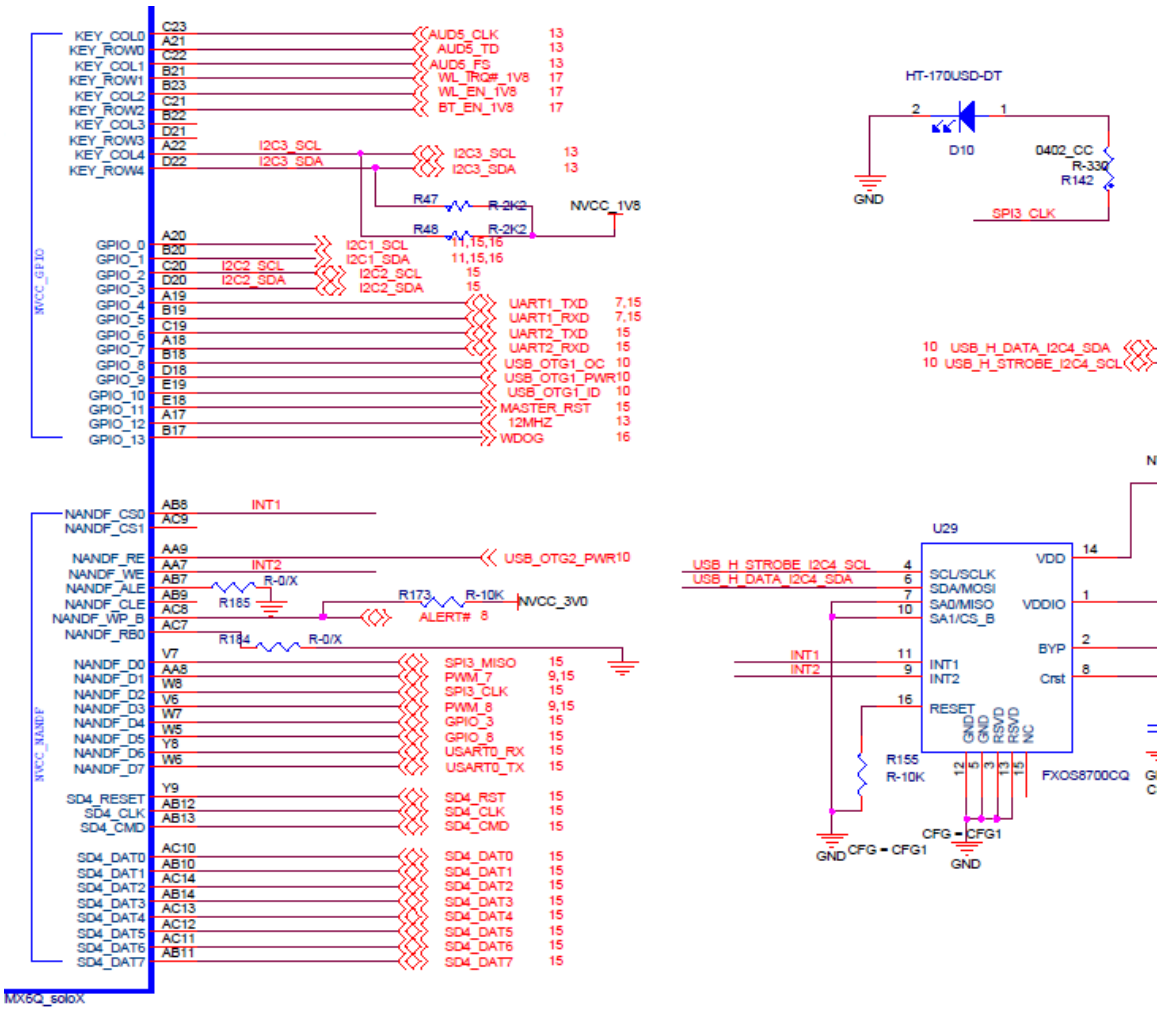
A typical kernel module function is to set some GPIO pins or to query their value. In this example the user_led is used as a gpio.

Prerequisites:

linux-kernel environment for udoo-neo
Kernel module source code from Kernel Module 2 - Char Device

Notes:

The chip pad must be configured as a GPIO and the output pad driver must be set accordingly. This is also done in the device tree under "iomuxc":



NAND_DATA02	ALT0	NAND_DATA02
	ALT1	SD1_DATA6
	ALT2	QSPI2B_SCLK
	ALT3	ECSPI5_SCLK
	ALT4	ESAI_TX_HF_CLK
	ALT5	GPIO4_IO06
	ALT6	EIM_AD02

The first value "**MX6SX_PAD_NAND_DATA02__GPIO4_IO_6**" defines the multiplexed pin function see *arch/arm/boot/dts/imx6sx-pinfunc.h*

Note, that linux uses number **102** for this GPIO. See how this number is calculated.

GPIOs are organized in groups of 32. So our GPIO pin is (bit) number 6 in group 4. Linux has a linear numbering of all GPIO pins

linux_gpio_number = 32 * (group_number-1) + pin_number_in-group
which, in our case, yields linux gpio number 32*(4-1) + 6 = 102.

Add the following lines to my_module.c at global scope:

```
#include <linux/gpio.h>

const int my_gpio = 102;
int my_value = 0;
```

Extend my_init function:

```
    int rc;

    rc = gpio_request( my_gpio, "my_gpio" );
    if(rc < 0)
    {
        printk(KERN_ERR "gpio_request failed, error %d\n", rc );
        return -1;
    }

    rc = gpio_direction_output( my_gpio, 0 );
    if(rc < 0)
    {
        printk(KERN_ERR "gpio_direction_output failed, error %d\n", rc );
        return -1;
    }
```

Refer : https://www.kernel.org/doc/Documentation/gpio/gpio-legacy.txt

Extend the my_write function such that the LED is switched:

```
    ...
    case 's':
        gpio_set_value( my_gpio, 1 );
        ...
    case 'c':
        gpio_set_value( my_gpio, 0 );
        ...
```

Extend the my_exit function such that the kernel resources are freed and the module can be loaded again:

```
static void __exit my_exit(void)
{
  gpio_free( my_gpio );
  ...
}
```

<<hello_gpio_1.zip>>
Cross-compile the kernel module, copy it to the target and load the module on the target:

insmod my_module.ko

Accessing the module from a shell

Now, you can set and clear the LED using the device /dev/cse3 exposed by the module:

```
echo s > /dev/cse3
echo c > /dev/cse3
```

You may also read the response from your module:

```
cat /dev/cse3
```

The response should be OK when you issued a valid command and ERROR else. Each response is only given once per command.
Accessing the module form a C program

Use the device /dev/cse3 exposed by your loaded module my_module.ko with the usual Posix IO functions:

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

int main()
{
    int f = open( "/dev/cse3", O_RDWR );
    write( f, "s", 1 );
    sleep(1000);
    write( f, "c", 1 );
    close(f);
}
```

Checking of return codes of all IO functions is omitted here for simplicity but should be present in your code! You might also read the response produced by the module and print it out for debugging.

Note, the the printk output from your module does not appear here. It is usually added to the kernel log and printed on the Linux console. The kernel log can be retrieved and configured in a shell by using the dmesg command.

You may only use the IO functions that you have implemented. In advanced scenarios you might want to implement ioctl to communicate to your module via customized C data structures (struct) instead of the simple char based IO.

4. Adding Hello module to the kernel source tree and compile

Generally compiler of the driver module version 3.x need to add code to the kernel source tree, and do the appropriate configuration, as follows (Note: The following steps are already done in fact, you only need to open the check directly to compiler on it):
Step 1: Edit the configuration file Kconfig, or add options to make it appear in the make menuconfig when open linux-kernel/drivers/char/Kconfig file, add as shown:

```
4
5  menu "Character devices"
6
7  source "drivers/tty/Kconfig"
8
9  config DEVKMEM
10     bool "/dev/kmem virtual device support"
11     default y
12     help
13       Say Y here if you want to support the /dev/kmem device. The
14       /dev/kmem device is rarely used, but can be used for certain
15       kind of kernel debugging operations.
16       When in doubt, say "N".
17
18 config UDOONEO_LED
19     tristate "LED char device driver example for udoo neo"
20     depends on ARCH_MXC
21     help
22        Led char device driver example
23        say Y or M here, otherwise say N.
```

Save and exit, this time in the linux-kernel directory where you run what you can make menuconfig Device Drivers > Character devices > to see just add menu options, and press the space bar will select the <M> this means that the option should compiled as a module ; then press the space will become <*>, which means should this option compiled into the kernel, we choose <M>, as shown:

```
[*]   Unix98 PTY support
[ ]     Support multiple instances of devpts
[ ]   Legacy (BSD) PTY support
[ ]   Non-standard serial port support
< >   GSM MUX line discipline support (EXPERIMENTAL)
< >   Trace data sink for MIPI P1149.7 cJTAG standard
[ ] /dev/kmem virtual device support
<M> LED char device driver example for udoo neo
    Serial drivers  --->
[ ] TTY driver to output user messages via printk
<*> Freescale On-Chip OTP Memory Support
[ ] ARM JTAG DCC console
< > IPMI top-level message handler  ----
<M> Hardware Random Number Generator Core support
```

Step 2: the previous step, although we can choose when configuring the kernel, but in fact the implementation at this time or can;t hello_gpio.c kernel compiled, but also need to in the Makefile in the kernel configuration options and the actual source code linked open linux-kernel/drivers/char/Makefile, add and save and exit as shown in figure:

```
obj-$(CONFIG_MSPEC)       += mspec.o
obj-$(CONFIG_MMTIMER)         += mmtimer.o
obj-$(CONFIG_UV_MMTIMER)      += uv_mmtimer.o
obj-$(CONFIG_IBM_BSR)         += bsr.o
obj-$(CONFIG_SGI_MBCS)        += mbcs.o
obj-$(CONFIG_BFIN_OTP)        += bfin-otp.o
obj-$(CONFIG_FSL_OTP)         += fsl_otp.o
obj-$(CONFIG_UDOONEO_LED)     += hello_gpio.o
```

Step 3: then back to linux-kernel root directory of the location, do make modules, we can produce the required documents hello_gpio.ko a kernel module, as shown:
So far, we have completed the model-driven compilation.

```
pod1hc@HCUTWRK2072:~/data/udoo-dev/linux_kernel$ ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make modules -j12
scripts/kconfig/conf --silentoldconfig Kconfig
  CHK     include/config/kernel.release
  CHK     include/generated/uapi/linux/version.h
  CHK     include/generated/utsrelease.h
make[1]: `include/generated/mach-types.h' is up to date.
  CALL    scripts/checksyscalls.sh
  CC [M]  drivers/char/hello_gpio.o
drivers/char/hello_gpio.c:67:31: warning: ‘my_fops’ defined but not used [-Wunused-variable]
  Building modules, stage 2.
  MODPOST 327 modules
  CC      drivers/char/hello_gpio.mod.o
  LD [M]  drivers/char/hello_gpio.ko
```