Linux Kernel and Device Drivers

- Sunil Vaghela

Chapter-1 Device driver basics **2**
Introduction 2
Kernel Module basics 5
Introducing printk() 6
Log Levels 6
Compiling kernel module 7
How do modules get into / out from the kernel? 8
To examine loaded kernel modules 9
Use your own init/deinit functions 10
The __init* and __exit* Macros 11
Licensing and Module Documentation 12
Modules Spanning Multiple Files 13
Passing a command line arguments to a module 14
Modules vs Programs 18
Device-Drivers 21
Major and Minor Numbers 22
Character Device Files 23
The file_operations Structure 23
The file structure 25
Exchanging data with user space 31
Automatic creation of device files 38

Chapter-2 Bridges between Kernel and User space **43**
IOCTL 43
The /proc File System (procfs) 49
Sysfs 54

Chapter-3 Device-tree: A data structure for hardware configuration **61**
DTC - Device Tree Compiler 61
Basic Device Tree syntax 62
Unit-Address Mystery 66
Device Tree inclusion 67
Device Tree Structure and Conventions 67
Node names 67

Generic Names Recommendation 68
Path Names 69
Properties 69
Standard properties 71
Interrupts and Interrupt Mapping 77
Properties for Interrupt Generating Devices: 78
Properties for Interrupt Controllers: 80
Interrupt Nexus properties: 80

Chapter-4 Wait-Queue in Linux 98

Chapter-5 Interrupts in Linux Kernel 106

Questions/Answers: 113

Important files in kernel: 114

Chapter-1 Device driver basics

Introduction

- A driver is one who drives – manages, controls, directs, monitors – the entity under his command. So a bus driver does that with a bus. Similarly, a device driver does that with a device.
- A device could be any peripheral connected to a computer, for example mouse, keyboard, screen / monitor, hard disk, camera, clock, … – you name it.
- A pilot could be a person or automatic systems, possibly monitored by a person. Similarly, a device driver could be a piece of software or another peripheral / device, possibly driven by a software.
- However, if it is another peripheral / device, it is referred to as device controller in the common parlance. And by driver, we only mean the software driver. A device controller is a device itself and hence many times it also needs a driver, commonly referred as a bus driver.
- General examples of device controllers include hard disk controllers, display controllers, audio controllers for the corresponding devices. More technical examples would be the controllers for the hardware protocols, such as an IDE controller, PCI controller, USB controller, SPI controller, I2C controller, etc.
- Device controllers are typically connected to the CPU through their respective named buses (collection of physical lines), for example pci bus, ide bus, etc. In today's embedded world, we more often come across microcontrollers than CPUs, which are nothing but *CPU + various device controllers built onto a single chip*.
- This effective embedding of device controllers primarily reduces cost & space, making it suitable for embedded systems. In such cases, the buses are integrated into the chip itself.
- Bus drivers provide hardware-specific interfaces for the corresponding hardware protocols, and are the bottom-most horizontal software layers of an operating system (OS). Over these sit the actual device' drivers. These operate on the underlying devices using the horizontal layer interfaces, and hence are device-specific.
- However, the whole idea of writing these drivers is to provide an abstraction to the user. And so on the other end, these do provide an interface to the user. This interface varies from OS to OS. In short, a device driver has two parts: i) Device-specific, and ii) OS-specific. Refer to below figure 1.

- The device-specific portion of a device driver remains same across all operating systems, and is more of understanding and decoding of the device data sheets, than of software programming.
- A data sheet for a device is a document with technical details of the device, including its operation, performance, programming, etc. However, the OS-specific portion is the one which is tightly coupled with the OS mechanisms of user interfaces. This is the one which differentiates a Linux device driver from a Windows device driver from a MAC device driver.
- In Linux, a device driver provides a system call interface to the user. And, this is the boundary line between the so-called kernel space and user space of Linux, as shown in figure 1.

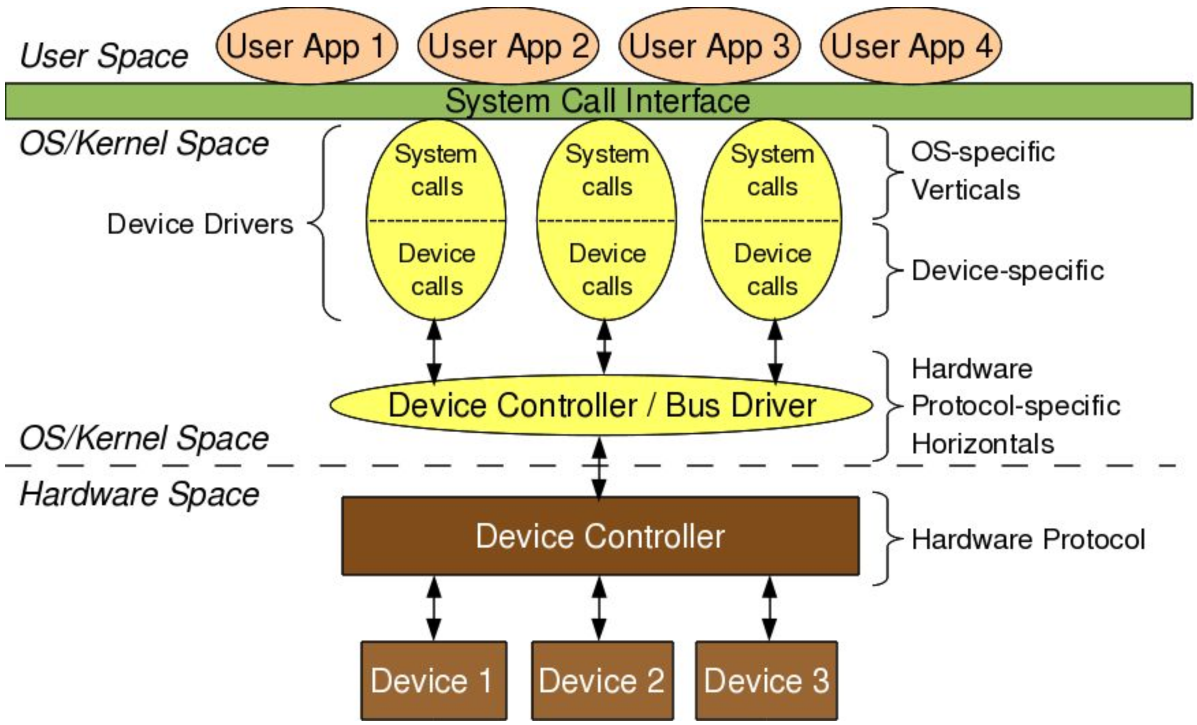
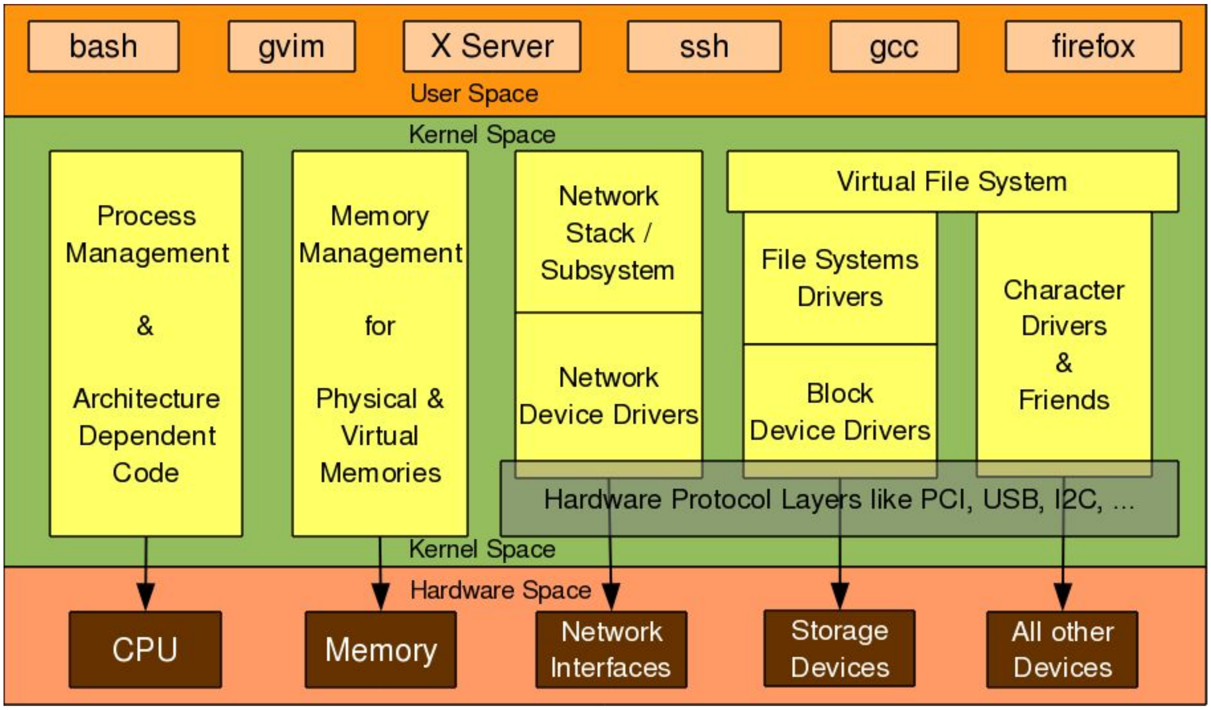


Figure - 1

- Figure 2 elaborates on further classification. Based on the OS-specific interface of a driver, in Linux a driver is broadly classified into 3 verticals:
  1. Packet-oriented or Network vertical
  2. Block-oriented or Storage vertical
  3. Byte-oriented or Character vertical



- The other two verticals, loosely the *CPU vertical* and *memory vertical* put together with the other three verticals give the complete overview of the Linux kernel, like any textbook definition of an OS: "*An OS does 5 managements namely: CPU/process, memory, network, storage, device/io*". Though these two could be classified as device drivers, where CPU & memory are the respective devices, these two are treated differently for many reasons.
- These are the core functionalities of any OS, be it micro or monolithic kernel. More often than not, adding code in these areas is mainly a Linux porting effort, typically for a new CPU or architecture.
- Moreover, the code in these two verticals cannot be loaded or unloaded on the fly, unlike the other three verticals. And henceforth to talk about Linux device drivers, we would mean to talk only on the later three verticals in figure 2.
- Let's get a little deeper into these three verticals. Network consists of 2 parts:
  1. Network protocol stack, and
  2. Network interface card (NIC) or simply network device drivers, which could be for ethernet, wifi, or any other network horizontals.
- Storage again consists of 2 parts:
  1. File system drivers for decoding the various formats on various partitions, and
  2. Block device drivers for various storage (hardware) protocols, that is the horizontals like IDE, SCSI, MTD, etc.
- With this you may wonder, is that the only set of devices for which you need drivers, or Linux has drivers for. Just hold on. You definitely need drivers for the whole lot of devices interfacing with a system, and Linux do have drivers for them. However, their byte-oriented accessibility puts all of them under the character vertical – yes I mean it – it is the majority bucket.
- In fact, because of this vastness, character drivers have got further sub-classified. So, you have tty drivers, input drivers, console drivers, framebuffer drivers, sound

drivers, etc. And the typical horizontals here would be RS232, PS/2, VGA, I2C, I2S, SPI, etc.

- On a final note on the complete picture of placement of all the drivers in the Linux driver ecosystem, the horizontals like USB, PCI, etc span below multiple verticals. Why? As we have a USB wifi dongle, a USB pen drive, as well as a USB to serial converter – all USB but three different verticals.
- In Linux, bus drivers or the horizontals, are often split into two parts, or even two drivers:
  1. Device controller specific, and
  2. An abstraction layer over that for the verticals to interface, commonly called cores.

  A classical example would be the usb controller drivers ohci, ehci, etc and the USB abstraction usbcore
- So, to conclude a device driver is a piece of software which drives a device, though there are so many classifications. And in case it drives only another piece of software, we call it just a driver. Examples are file system drivers, usbcore, etc. Hence, **all device drivers are drivers but all drivers are not device drivers.**

• Kernel Module basics

- Kernel modules are pieces of code that can be loaded and unloaded into the *kernel* upon demand. They extend the functionality of the kernel without the need to reboot the system.
- Kernel modules must have at least two functions: a "start" (initialization) function called `init_module()` which is called when the module is insmod-ed into the kernel, and an "end" (cleanup) function called `cleanup_module()` which is called just before it is rmmod-ed.
- `init_module()` either registers a handler for something with the kernel, or it replaces one of the kernel functions with its own code (usually code to do something and then call the original function). The `cleanup_module()` function is supposed to undo whatever `init_module()` did, so the module can be unloaded safely.

```
/* The simplest kernel module */

#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_INFO */

int init_module(void)
{
    printk(KERN_INFO "Welcome to the world of kernel programming\n");
    return 0;
}

void cleanup_module(void)
{
    printk(KERN_INFO "There is no way to exit. see you again!\n");
}
```

• Introducing `printk()`

- `printk` works more or less the same way as `printf` in userspace, so if you ever debugged your userspace program using printf, you are ready to do the same with your kernel code, e.g. by adding:

```
printk("A program is never finished until the programmer dies.\n");
```

- This wasn't that difficult, was it? Usually you would print out some more interesting information like,

```
printk("Var1 %d var2 %d\n", var1, var2);
```

- just like in userspace. In order to see the kernel messages, just use the dmesg command in one of your shells - this one will print out the whole kernel log buffer to you.
- Most of the conversion specifiers supported by the user-space library routine `printf()` - are also available in the kernel; there are some notable additions, including "`%pf`", which will print the symbol name in place of the numeric pointer value, if available.
- The supported format strings are quite extensively documented in Documentation/printk-formats.txt
- ***However please note***: always use `%zu, %zd or %zx` for printing `size_t` and `ssize_t` values. `ssize_t` and `size_t` are quite common values in the kernel, so please use the `%z` to avoid annoying compile warnings.

• Log Levels

- If you look into real kernel code you will always see something like:

```
printk(KERN_ERR "something went wrong, return code: %d\n",ret);
```

- where *KERN_ERR* is one of the eight different log levels defined in *include/linux/kern_levels.h* and specifies the severity of the error message.
- Note that there is **NO comma** between the `KERN_ERR` and the format string (as the preprocessor concatenates both strings)

▷ The log levels are:

Name	String	Meaning	Alias function
KERN_EMERG	“0”	Emergency messages, system is about to crash or is unstable	`pr_emerg`
KERN_ALERT	“1”	Something bad happened and action must	`pr_alert`

		be taken immediately	
KERN_CRIT	“2”	A critical condition occurred like a serious hardware/software failure	pr_crit
KERN_ERR	“3”	An error condition, often used by drivers to indicate difficulties with the hardware	pr_err
KERN_WARNING	“4”	A warning, meaning nothing serious by itself but might indicate problems	pr_warning
KERN_NOTICE	“5”	Nothing serious, but notably nevertheless. Often used to report security events.	pr_notice
KERN_INFO	“6”	Informational message e.g. startup information at driver initialization	pr_info
KERN_DEBUG	“7”	Debug messages	pr_debug
KERN_DEFAULT	"d"	The default kernel loglevel	
KERN_CONT	“c”	"continued" line of log printout (only done after a line that had no enclosing \n)	pr_cont

▷ Memorising kernel log levels:

- **E**veryone **A**lways **C**omplains **E**ven **W**hen **N**othing **I**s **D**ifferent
- **E**very **A**wesome **C**isco **E**ngineer **W**ill **N**eed **I**cecream **D**aily
- **E**very **A**lley **C**at **E**ats **W**atery **N**oodles **I**n **D**oors
- **E**veryone **A**ttends **C**lass **E**ach **W**eek **N**ot **I**f **D**ead

• Compiling kernel module

- Kernel modules need to be compiled a bit differently from regular user-space apps. Kernel Makefiles are part of the kbuild system, documented here.

  Below is a simple makefile to compile a kernel module:

```
obj-m += hello.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

- Now you can compile the module by issuing the command **make**. You should obtain an output which resembles the following:

```
debian@beaglebone:~/test/hello$ make
make -C /lib/modules/4.14.108-ti-r113/build M=/home/debian/test/hello modules
make[1]: Entering directory '/usr/src/linux-headers-4.14.108-ti-r113'
```

```
  CC [M]  /home/debian/test/hello/hello.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/debian/test/hello/hello.mod.o
  LD [M]  /home/debian/test/hello/hello.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.14.108-ti-r113'
```

- From kernel 2.6 a new file naming convention has been introduced for kernel modules - i.e. `.ko` extension (in place of the old .o extension) which easily distinguish them from conventional object files. The reason for this is that they contain an additional `.modinfo` section where additional information about the module is kept. Use the modinfo command to see what kind of information it is.

```
debian@beaglebone:~/test/hello$ modinfo hello.ko
filename:       /home/debian/test/hello/hello.ko
depends:
name:           hello
vermagic:       4.14.108-ti-r113 SMP preempt mod_unload modversions ARMv7 p2v8
debian@beaglebone:~/test/hello$
```

- **How do modules get into / out from the kernel?**
  - **modprobe**
    - modprobe utility is used to add loadable modules to the Linux kernel. You can also view and remove modules using the modprobe command.
    - modprobe is an intelligent command, it looks for dependencies while loading a module. Suppose, if we load a module, which has symbols defined in some other module (this module path is given inside the main module). so, modprobe loads the main module and the dependent module.
    - modprobe looks through the file `/lib/modules/$(uname -r)/modules.dep`, to see if other modules must be loaded before the requested module may be loaded. This file is created by `depmod -a` and contains module dependencies. The requested module has a dependency on another module if the other module defines symbols (variables or functions) that the requested module uses.
    - Linux maintains `/lib/modules/$(uname -r)` directory for modules and its configuration files (`except /etc/modprobe.conf` and `/etc/modprobe.d`).
      - **To insert module using modprobe:**

        1. `sudo ln -s /path/to/your-kernel-module.ko /lib/modules/`uname -r`/`

        OR

        1. `sudo cp /path/to/your-kernel-module.ko /lib/modules/`uname -r`/`
        2. `sudo depmod -a` (Generate a list of kernel module dependencies and associated map files.)
        3. `sudo modprobe your-kernel-module`

- **To remove module using modprobe:**
  - `sudo modprobe -r your-kernel-module`

- **insmod**

  - insmod is similar to modprobe: it can insert a module into the Linux kernel. Unlike modprobe, however, insmod does not read its modules from a set location, automatically insert them, and manage any dependencies.
  - insmod can insert a single module from any location, and does not consider dependencies when doing so. It's a much lower-level program; in fact, it's the program modprobe uses to do the actual module insertion.

  - **To insert module using insmod:**
    - `sudo insmod /path/to/your-kernel-module.ko`

  - Insmod requires you to pass it the full pathname and to insert the modules in the right order, while *modprobe* just takes the name, without any extension, and figures out all it needs to know by parsing **`/lib/modules/version/modules.dep`** .

- **rmmod**

  - rmmod is a simple program which removes (unloads) a module from the Linux kernel. In most cases, you will want to use modprobe with the -r option instead, as it is more robust and handles dependencies for you.

  - **To insert module using insmod:**
    - `sudo rmmod your-kernel-module.ko`

To examine loaded kernel modules

- **lsmod**
  - lsmod is a simple utility that does not accept any options or arguments. What the command does is that it reads `/proc/modules` and displays the file contents in a nicely formatted list.
  - Run `lsmod` at the command line to find out what kernel modules are currently loaded. Below is a snapshot of our loaded module:

```
debian@beaglebone:~/test/hello$ lsmod
Module                  Size    Used by
hello                   16384   0
evdev                   24576   1
8021q                   32768   0
garp                    16384   1 8021q
mrp                     20480   1 8021q
stp                     16384   1 garp
```

```
llc                          16384      2 garp,stp
usb_f_mass_storage           53248      2
usb_f_acm                    16384      2
u_serial                     20480      3 usb_f_acm
```

- Each line has three columns:
    - **Module** - The first column shows the name of the module.
    - **Size** - The second column shows the size of the module in bytes.
    - **Used by** - The third column shows a number that indicates how many instances of the module are currently used. A value of zero means that the module is not used. The comma-separated list after the number shows what is using the module.

- **Using `/proc/modules`**

    - This file displays a list of all modules loaded into the kernel in an unarranged list.

```
debian@beaglebone:~/test/hello$ cat /proc/modules
hello 16384 0 - Live 0xbf1f9000 (PO)
evdev 24576 1 - Live 0xbf1e7000
8021q 32768 0 - Live 0xbf1d9000
garp 16384 1 8021q, Live 0xbf1d2000
mrp 20480 1 8021q, Live 0xbf1c9000
stp 16384 1 garp, Live 0xbf1c2000
llc 16384 2 garp,stp, Live 0xbf1b9000
usb_f_mass_storage 53248 2 - Live 0xbf1a4000
usb_f_acm 16384 2 - Live 0xbf19b000
u_serial 20480 3 usb_f_acm, Live 0xbf192000
```

• Use your own init/deinit functions

- As of Linux 2.4, you can rename the init and cleanup functions of your modules; they no longer have to be called `init_module()` and `cleanup_module()` respectively.
- This is done with the `module_init()` and `module_exit()` macros.These macros are defined in **`linux/init.h`**.
- The only caveat is that your init and cleanup functions must be defined before calling the macros, otherwise you'll get compilation errors.

```
/* Demonstration of module_init() and module_exit() macros */

#include <linux/module.h>    /* Needed by all modules */
#include <linux/kernel.h>      /* Needed for KERN_INFO */
#include <linux/init.h>          /* Needed for the macros */

static int __init hello_init(void)
{
    printk(KERN_INFO "All computers wait at the same speed !\n");
```

```
    return 0;
}

static void __exit hello_exit(void)
{
    printk(KERN_INFO "Any program that runs right is obsolete !\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

- So now we have two real kernel modules under our belt. Adding another module is as simple as this:

```
obj-m += hello.o
obj-m += hello-2.o

all:
        make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
        make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

• The `__init*` and `__exit*` Macros

- The init* and exit* macros are widely used in the kernel. These macros are defined in **`include/linux/init.h`** and serve to free up kernel memory.
- When you boot your kernel and see something like `Freeing unused kernel memory: 236k freed`, this is precisely what the kernel is freeing.

```
#define __init          __attribute__ ((__section__ (".init.text")))
#define __initdata      __attribute__ ((__section__ (".init.data")))
#define __exitdata      __attribute__ ((__section__(".exit.data")))
#define __exit_call     __attribute_used__ __attribute__ ((__section__
(".exitcall.exit")))

#ifdef MODULE
#define __exit          __attribute__ ((__section__(".exit.text")))
#else
#define __exit          __attribute_used__ __attribute__ ((__section__(".exit.text")))
#endif
```

• `__init*` macros

- It tells the compiler to put the variable or the function in a special section, which is declared in vmlinux.lds.h. *init* puts the function in the "`.init.text`" section and `initdata` puts the data in the "`.init.data`" section.
- For example, the following declaration means that the variable `helloworld_data` will be put in the init data section.

```
static int helloworld_data __initdata;
```

- But why must you use these macros ?
  Let's take an example, with the following function, defined in `mm/slab.c` :

```
void __init kmem_cache_init(void);
```

- This function initializes the slab system: it's only used once, at the boot of the kernel. So the code of this function should be freed from the memory after the first call. It's the goal of **`free_initmem()`**.
- The function **`free_initmem()`** will free the entire text and data init sections and so the code of your function, if it has been declared as init.

__exit* macros

- The **`__exit`** macro causes the omission of the function when the module is built into the kernel, and like **`__exit`**, has no effect for loadable modules. Again, if you consider when the cleanup function runs, This makes complete sense; built-in drivers don't need a cleanup function, while loadable modules do.
- The exit macro tells the compiler to put the function in the **`".exit.text"`** section. The exit_data macro tells the compiler to put the function in the **`".exit.data"`** section.
- exit.* sections make sense only for the modules : exit functions will never be called if compiled statically. That's why there is an ifdef : exit.* sections will be discarded only if modules support is disabled.

Licensing and Module Documentation

- In kernel 2.4 and later, a mechanism was devised to identify code licensed under the GPL (and friends) so people can be warned that the code is non open-source.
- This is accomplished by the **`MODULE_LICENSE()`** macro. By setting the license to GPL, you can keep the warning from being printed. This license mechanism is defined and documented in **`linux/module.h`** :
- Similarly, **`MODULE_DESCRIPTION()`** is used to describe what the module does, **`MODULE_AUTHOR()`** declares the module's author, and **`MODULE_SUPPORTED_DEVICE()`** declares what types of devices the module supports.
- These macros are all defined in **`linux/module.h`** and aren't used by the kernel itself. They're simply for documentation and can be viewed by a tool like objdump.

`MODULE_INFO`	Generic info of form tag = "info"
`MODULE_ALIAS`	For userspace: you can also call me...
`MODULE_SOFTDEP`	Soft module dependencies. See `man modprobe.d` for details.
`MODULE_LICENSE`	Indication for module license - Free(GPL, GLP v2, GPL and additional rights, Dual BSD/GPL, Dual MIT/GPL, Dual MPL/GPL) / Proprietary

MODULE_AUTHOR	Author(s), use "Name <email>" or just "Name", for multiple authors use multiple MODULE_AUTHOR() statements/lines.
MODULE_DESCRIPTION	What your module does
MODULE_VERSION	Version of form [<epoch>:]<version>[-<extra-version>] **<epoch>** : A (small) unsigned integer which allows you to start versions a new. If not mentioned, it's zero. eg. "2:1.0" is after "1:2.0". **<version>** : The <version> may contain only alphanumerics and the character `.'. Ordered by numeric sort for numeric parts, ascii sort for ascii parts. **<extraversion>** : Like <version>, but inserted for local customizations, eg. "rh3" or "rusty1".
MODULE_DEVICE_TABLE	

- ## Modules Spanning Multiple Files

  - Sometimes it makes sense to divide a kernel module between several source files. Here's an example of such a kernel module.

```
/* module_spanning_init.c : Illustration of multi-file modules */

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

int __init module_split_init(void)
{
    printk(KERN_DEBUG "Welcome to the universe of linux!\n");
    return 0;
}

module_init(module_split_init);
```

```
/* module_spanning_exit.c : Illustration of multi-file modules */

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

void module_split_exit(void)
{
    printk(KERN_DEBUG "Remember! There is no exit once you entered!\n");
}

module_exit(module_split_exit);
```

```
# Makefile

obj-m += 0103_module_spanning.o
0103_module_spanning-objs := 0103_module_spanning_init.o 0103_module_spanning_exit.o

all:
```

```
    make -C /lib/modules/`uname -r`/build M=$(PWD) modules

clean:
    make -C /lib/modules/`uname -r`/build M=$(PWD) clean
```

• Passing a command line arguments to a module

- Modules can take command line arguments, but not with the argc/argv you might be used to.
- To allow arguments to be passed to your module, declare the variables that will take the values of the command line arguments as global and then use the **module_param()** macro, (defined in `linux/moduleparam.h`) to set the mechanism up.

- **module_param(name, type, perm);**
  - where, **name** is the name of both the parameter exposed to the user and the variable holding the parameter inside your module.
  - The **type** argument holds the parameter's data type; it is one of `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `charp`, `bool`, or `invbool`. These types are, respectively, a byte, a short integer, an unsigned short integer, an integer, an unsigned integer, a long integer, an unsigned long integer, a pointer to a char, a Boolean, and a Boolean whose value is inverted from what the user specifies.
  - The byte type is stored in a single char and the Boolean types are stored in variables of type int. The rest are stored in the corresponding primitive C types.
  - Finally, the **perm** argument specifies the permissions of the corresponding file in sysfs.
  - The permissions can be specified in the usual octal format, for example 0644 (owner can read and write, group can read, everyone else can read), or by ORing together the usual S_I* defines, for example `S_IRUGO | S_IWUSR` (everyone can read, a user can also write). (defined in **include/linux/stat.h**)
  - The macro does not declare the variable for you. You must do that before using the macro. Therefore, typical use might resemble

```
/* module parameter controlling the capability to allow live bait on the pole */
static int allow_live_bait = 1;              /* default to on */
module_param(allow_live_bait, bool, 0644);   /* a Boolean type */
```

  - This would be in the outermost scope of your module's source file. In other words, `allow_live_bait` is global.

- **module_param_named(name, variable, type, perm);**
  - It is possible to have the internal variable named differently than the external parameter. This is accomplished via **module_param_named()**.
  - where, **name** is the externally viewable parameter name and **variable** is the name of the internal global variable. For example,

```
static unsigned int max_test = DEFAULT_MAX_LINE_TEST;
module_param_named(maximum_line_test, max_test, int, 0);
```

- **module_param_string(name, string, len, perm);**
  - Normally, you would use a type of *charp* to define a module parameter that takes a string. The kernel copies the string provided by the user into memory and points your variable to the string. For example,

```
static char *name;
module_param(name, charp, 0);
```

  - If so desired, it is also possible to have the kernel copy the string directly into a character array that you supply. This is done via **module_param_string()**.
  - where, `name` is the external parameter name, `string` is the internal variable name, `len` is the size of the buffer named by string (or some smaller size, but that does not make much sense), and `perm` is the sysfs permissions (or zero to disable a sysfs entry altogether). For example,

```
static char species[BUF_LEN];
module_param_string(specifies, species, BUF_LEN, 0);
```

- **module_param_array(name, type, nump, perm);**
  - You can accept a comma-separated list of parameters that are stored in a C array via **module_param_array()**.
  - where, `name` is again the external parameter and internal variable name, `type` is the data type, and `perm` is the sysfs permissions. The new argument, `nump`, is a pointer to an integer where the kernel will store the number of entries stored into the array.
  - Note that the array pointed to by name must be statically allocated. The kernel determines the array's size at compile-time and ensures that it does not cause an overrun. Use is simple. For example,

```
static int fish[MAX_FISH];
static int nr_fish;
module_param_array(fish, int, &nr_fish, 0444);
```

- **module_param_array_named(name, array, type, nump, perm);**
  - You can name the internal array something different than the external parameter with **module_param_array_named()**.
  - The parameters are identical to the other macros.

- **MODULE_PARM_DESC(name, description)**
  - Finally, you can document your parameters by using **MODULE_PARM_DESC()**:

```
static unsigned short size = 1;
module_param(size, ushort, 0644);
MODULE_PARM_DESC(size, "The size in inches of the fishing pole connected to this
computer.");
```

- All these macros require the inclusion of <linux/moduleparam.h>.
- At runtime, insmod will fill the variables with any command line arguments that are given, like **./insmod mymodule.ko myvariable=5**. The variable declarations and macros should be placed at the beginning of the module for clarity.
- A good use for this is to have the module variable's default values set, like a port or IO address. If the variables contain the default values, then perform auto-detection. Otherwise, keep the current value.

```
/* Demonstration of a way to pass command line argument passing to a module */

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/stat.h>
#include <linux/init.h>

#define COLORS_NAMES_LEN    40
#define ARRAY_LEN            5

static short int cmd_short = 10;
static int cmd_int = 100;
static long int cmd_long = 10000;
static char* cmd_charp = "Avengers! assemble";
static char cmd_string[COLORS_NAMES_LEN] = "red, green, blue";
static int cmd_int_array[ARRAY_LEN] = {1,2,3,4,5};
static int cmd_int_array_idx = 0;
static char cmd_char_array[ARRAY_LEN] = {'a', 'b', 'c', 'd', 'e'};
static int cmd_char_array_idx = 0;

/* user and group can read/write */
module_param(cmd_short, short, 0660);
MODULE_PARM_DESC(cmd_short, "A short integer number");
/* user and group can read/write */
module_param_named(lucky_number, cmd_int, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
MODULE_PARM_DESC(lucky_number, "A lucky number");
/* user and group can read only */
module_param(cmd_long, long, 0);
MODULE_PARM_DESC(cmd_long, "A readonly long number");
/* anyone can (user,group,others) read only */
module_param(cmd_charp, charp, S_IRUSR | S_IRGRP | S_IROTH);
MODULE_PARM_DESC(cmd_charp, "A readonly string");
/* only user can read and write */
module_param_string(fav_colors, cmd_string, COLORS_NAMES_LEN, 0600);
MODULE_PARM_DESC(fav_colors, "Favourite colors");
/* only user can read */
module_param_array(cmd_int_array, int, &cmd_int_array_idx, 0640);
MODULE_PARM_DESC(cmd_int_array, "An int array of 5");
/* only user can read and write */
module_param_array_named(fav_letters, cmd_char_array, byte, &cmd_char_array_idx, 0600);
MODULE_PARM_DESC(fav_letters, "5 favourite letters");
```

```
int __init cmdline_demo_init(void)
{
    int i = 0;
    printk(KERN_INFO "command line demo init\n");
    printk(KERN_INFO "==============================\n");
    printk(KERN_INFO "=> cmd_short : %hd\n", cmd_short);
    printk(KERN_INFO "=> lucky_number : %d\n", cmd_int);
    printk(KERN_INFO "=> cmd_long : %ld\n", cmd_long);
    printk(KERN_INFO "=> cmd_charp : %s\n", cmd_charp);
    printk(KERN_INFO "=> fav_colors : %s\n", cmd_string);
    for(i = 0; i < ARRAY_LEN; i++)
    {
        printk(KERN_INFO "=> cmd_int_array[%d] : %d\n", i, cmd_int_array[i]);
    }
    for(i = 0; i < ARRAY_LEN; i++)
    {
        printk(KERN_INFO "=> faourite letter[%d] : %c\n", i, cmd_char_array[i]);
    }
    printk(KERN_INFO "==============================\n");
    return 0;
}

void __exit cmdline_demo_exit(void)
{
    printk(KERN_INFO "command line demo exit\n");
}

module_init(cmdline_demo_init);
module_exit(cmdline_demo_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Sunil Vaghela");
```

```
/* sample output */

debian@beaglebone:~/0104-cmdline-params$ sudo insmod 0104_cmdline_param.ko
debian@beaglebone:~/linux-device-drivers/0104-cmdline-params$ dmesg | tail -n 18
[ 2294.865925] command line demo init
[ 2294.865941] ==============================
[ 2294.865950] => cmd_short : 10
[ 2294.865955] => lucky_number : 100
[ 2294.865960] => cmd_long : 10000
[ 2294.865965] => cmd_charp : Avengers! assemble
[ 2294.865970] => fav_colors : red, green, blue
[ 2294.865976] => cmd_int_array[0] : 1
[ 2294.865981] => cmd_int_array[1] : 2
[ 2294.865986] => cmd_int_array[2] : 3
[ 2294.865990] => cmd_int_array[3] : 4
[ 2294.865995] => cmd_int_array[4] : 5
[ 2294.866001] => favorite letter[0] : a
[ 2294.866006] => favorite letter[1] : b
[ 2294.866010] => favorite letter[2] : c
[ 2294.866015] => favorite letter[3] : d
[ 2294.866019] => favorite letter[4] : e
[ 2294.866024] ==============================
debian@beaglebone:~/0104-cmdline-params$ dmesg | tail -n 5
[ 2294.866010] => favorite letter[2] : c
[ 2294.866015] => favorite letter[3] : d
[ 2294.866019] => favorite letter[4] : e
[ 2294.866024] ==============================
[ 2299.019158] command line demo exit
```

```
debian@beaglebone:~/0104-cmdline-params$ sudo insmod 0104_cmdline_param.ko cmd_short=10
lucky_number=456 cmd_long=47 cmd_charp='"Marvel vs DC"' fav_colors='"yellow black
white"' cmd_int_array=10,20,3,40,50 fav_letters=0x41,0x42,0x43,0x44,0x45
debian@beaglebone:~/linux-device-drivers/0104-cmdline-params$ dmesg | tail -n 18
[ 4493.585918] command line demo init
[ 4493.585932] ==============================
[ 4493.585942] => cmd_short : 10
[ 4493.585948] => lucky_number : 456
[ 4493.585953] => cmd_long : 47
[ 4493.585958] => cmd_charp : Marvel vs DC
[ 4493.585963] => fav_colors : yellow black white
[ 4493.585969] => cmd_int_array[0] : 10
[ 4493.585974] => cmd_int_array[1] : 20
[ 4493.585979] => cmd_int_array[2] : 3
[ 4493.585983] => cmd_int_array[3] : 40
[ 4493.585988] => cmd_int_array[4] : 50
[ 4493.585994] => favorite letter[0] : A
[ 4493.585999] => favorite letter[1] : B
[ 4493.586004] => favorite letter[2] : C
[ 4493.586008] => favorite letter[3] : D
[ 4493.586013] => favorite letter[4] : E
[ 4493.586017] ==============================
```

- **while passing a string with space, you have to put the string between '""', else only the first word will be assigned to the module parameter and words after space will be discarded.**
- **Also notice how the int and byte array has passed to the module.**
- **Value of the readonly variable also changed - why ?**

- **Modules vs Programs**
  - **How modules begin and end**
    - A program usually begins with a `main()` function, executes a bunch of instructions and terminates upon completion of those instructions.
    - Kernel modules work a bit differently. A module always begins with either the `init_module` or the function you specify with the `module_init` call. This is the entry function for modules; it tells the kernel what functionality the module provides and sets up the kernel to run the module's functions when they're needed. Once it does this, the entry function returns and the module does nothing until the kernel wants to do something with the code that the module provides.
    - All modules end by calling either `cleanup_module` or the function you specify with the `module_exit` call. This is the exit function for modules; it undoes whatever entry function did. It unregisters the functionality that the entry function registered.

  - **Functions available to modules**
    - Programmers use functions they don't define all the time. A prime example of this is `printf()`. You use these library functions which are provided by the standard C library, libc. The definitions for these functions don't actually enter your program until the linking stage, which ensures that the code (for `printf()` for example) is available, and fixes the call instruction to point to that code.

- Kernel modules are different here, too. In the previous all examples, you might have noticed that we used a function, `printk()` but didn't include a standard I/O library. That's because **modules are object files whose symbols get resolved upon insmod'ing**.
- **The definition for the symbols comes from the kernel itself; the only external functions you can use are the ones provided by the kernel.** If you're curious about what symbols have been exported by your kernel, take a look at **/proc/kallsyms**.
- One point to keep in mind is the difference between library functions and system calls. Library functions are higher level, run completely in user space and provide a more convenient interface for the programmer to the functions that do the real -- **system calls**.
- System calls run in kernel mode on the user's behalf and are provided by the kernel itself. The library function `printf()` may look like a very general printing function, but all it really does is format the data into strings and write the string data using the low-level system call `write()`, which then sends the data to standard output.
- Would you like to see what system calls are made by `printf()`? It's easy! Compile the following program:

```
/* hello.c - Demonstrate printf -> write connection using strace */

#include <stdio.h>
int main(void)
{
  printf("hello");
  return 0;
}
```

```
/* Run the executable with strace */

sunil@sunil-Inspiron-N4050:Desktop ⚡ strace ./hello
execve("./hello", ["./hello"], 0xbffe6aa0 /* 59 vars */) = 0
brk(NULL)                               = 0x1905000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7f03000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=117247, ...}) = 0
mmap2(NULL, 117247, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7ee6000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\3\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\20\220\1\0004\0\0\0"...,
512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1942840, ...}) = 0
mmap2(NULL, 1948188, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0xb7d0a000
mprotect(0xb7edf000, 4096, PROT_NONE)   = 0
mmap2(0xb7ee0000, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1d5000) = 0xb7ee0000
mmap2(0xb7ee3000, 10780, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb7ee3000
```

```
close(3)                                = 0
set_thread_area({entry_number=-1, base_addr=0xb7f040c0, limit=0x0fffff,
seg_32bit=1, contents=0, read_exec_only=0, limit_in_pages=1, seg_not_present=0,
useable=1}) = 0 (entry_number=6)
mprotect(0xb7ee0000, 8192, PROT_READ)   = 0
mprotect(0x475000, 4096, PROT_READ)     = 0
mprotect(0xb7f30000, 4096, PROT_READ)   = 0
munmap(0xb7ee6000, 117247)              = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 2), ...}) = 0
brk(NULL)                               = 0x1905000
brk(0x1926000)                          = 0x1926000
brk(0x1927000)                          = 0x1927000
write(1, "hello", 5hello)                       = 5
exit_group(0)                           = ?
+++ exited with 0 +++
```

- Are you impressed? Every line you see corresponds to a system call. `strace` is a handy program that gives you details about what a system calls a program is making, including which call is made, what its arguments are and what it returns. It's an invaluable tool for figuring out things like what files a program is trying to access. The highlighted line is the face behind the `printf()` mask.
- You may not be familiar with *write*, since most people use library functions for file I/O (like `fopen`, `fputs`, `fclose`). If that's the case, try looking at **`man 2 write`**. The 2nd man section is devoted to system calls (like `kill()` and `read()`). The 3rd man section is devoted to library calls.

- **User Space vs Kernel Space**
  - A kernel is all about access to resources, whether the resource in question happens to be a video card, a hard drive or even memory. Programs often compete for the same resource. As I just saved this document, `updatedb` started updating the locate database.
  - My vim session and updatedb are both using the hard drive concurrently. The kernel needs to keep things orderly, and not give users access to resources whenever they feel like it.
  - To this end, a CPU can run in different modes. Each mode gives a different level of freedom to do what you want on the system. The Intel 80386 architecture has 4 of these modes, which are called `rings`. Unix uses only two rings; the **`highest ring`** (ring 0, also known as '**`supervisor mode`**' where everything is allowed to happen) and the **`lowest ring`**, which is called '**`user mode`**'.
  - Recall the discussion about library functions vs system calls. Typically, you use a library function in user mode. The library function calls one or more system calls, and these system calls execute on the library function's behalf, but do so in supervisor mode since they are part of the kernel itself. Once the system call completes its task, it returns and execution gets transferred back to user mode.

- **Name space**
  - When you write a small C program, you use variables which are convenient and make sense to the reader. If, on the other hand, you're writing routines which will be part of a bigger problem, any global variables you have are part of a

community of other peoples' global variables; some of the variable names can clash.

- When a program has lots of global variables which aren't meaningful enough to be distinguished, you get **namespace pollution**. In large projects, effort must be made to remember reserved names, and to find ways to develop a scheme for naming unique variable names and symbols.
- When writing kernel code, even the smallest module will be linked against the entire kernel, so this is definitely an issue. The best way to deal with this is to declare all your variables as static and to use a well-defined prefix for your symbols. By convention, all kernel prefixes are lowercase. If you don't want to declare everything as static, another option is to declare a symbol table and register it with a kernel. We'll get to this later.
- The file **/proc/kallsyms** holds all the symbols that the kernel knows about and which are therefore accessible to your modules since they share the kernel's codespace.

- **Code space**
  - If you haven't thought about what a segfault really means, you may be surprised to hear that pointers don't actually point to memory locations. Not real ones, anyway.
  - When a process is created, the kernel sets aside a portion of real physical memory and hands it to the process to use for its executing code, variables, stack, heap and other things. This memory begins with 0x00000000 and extends up to whatever it needs to be.
  - Since the memory space for any two processes don't overlap, every process that can access a memory address, say 0xbffff978, would be accessing a different location in real physical memory! The processes would be accessing an index named 0xbffff978 which points to some kind of offset into the region of memory set aside for that particular process.
  - The kernel has its own space of memory as well. Since a module is code which can be dynamically inserted and removed in the kernel, it shares the kernel's codespace rather than having its own. Therefore, `if your module segfaults, the kernel segfaults`. And if you start writing over data because of an off-by-one error, then you're trampling on kernel data (or code). This is even worse than it sounds, so try your best to be careful.

- **Device-Drivers**
  - One class of module is the device driver, which provides functionality for hardware like a TV card or a serial port. On unix, each piece of hardware is represented by a file located in `/dev` named a device file which provides the means to communicate with the hardware.
  - The device driver provides the communication on behalf of a user program. e.g the `es1370.o sound card device driver` might connect the `/dev/sound` device file to the Ensoniq IS1370 sound card. A userspace program like VLC can use `/dev/sound` without ever knowing what kind of sound card is installed.

- **Major and Minor Numbers**

- The kernel needs to be told how to access the device. This is accomplished by the major number and the minor number of that device.
- Let's look at some device files. Here are device files which represent the first five partitions of the hard disk or **SCSI - Small Computer System Interface** (pronounced "skuzzy") disk drive:

```
sunil@sunil-Inspiron-N4050:~ ⚡ ls -l /dev/sda[1-5]
brw-rw---- 1 root disk 8, 1 Apr 11 14:01 /dev/sda1
brw-rw---- 1 root disk 8, 2 Apr 11 14:01 /dev/sda2
brw-rw---- 1 root disk 8, 3 Apr 11 14:01 /dev/sda3
brw-rw---- 1 root disk 8, 4 Apr 11 14:01 /dev/sda4
brw-rw---- 1 root disk 8, 5 Apr 11 14:01 /dev/sda5
```

- Notice the column of numbers separated by a comma? The first number is called the device's **major number**. The second number is the **minor number**. The major number tells you which driver is used to access the hardware.
- Each driver is assigned a unique major number; all device files with the same major number are controlled by the same driver. All the above major numbers are 11, because they're all controlled by the same driver.
- The minor number is used by the driver to distinguish between the various hardware it controls. Returning to the example above, although all five devices are handled by the same driver they have unique minor numbers because the driver sees them as being different pieces of hardware.
- Devices are divided into two types: `character devices` and `block devices`. The difference is that block devices have a buffer for requests, so they can choose the best order in which to respond to the requests. This is important in the case of storage devices, where it's faster to read or write sectors which are close to each other, rather than those which are further apart. Another difference is that block devices can only accept input and return output in blocks (whose size can vary according to the device), whereas character devices are allowed to use as many or as few bytes as they like. Most devices in the world are character, because they don't need this type of buffering, and they don't operate with a fixed block size.
- You can tell whether a device file is for a block device or a character device by looking at the first character in the output of **ls -l**. If it's `` `b' then it's a block device ``, and if it's `'c' then it's a character device`. The devices you see above are block devices.
- Here are some character devices:

```
sunil@sunil-Inspiron-N4050:~ ⚡ ls -l /dev/ttyS0 /dev/ttyUSB0 /dev/i2c-0
/dev/media0
crw------- 1  root root     89,  0 Apr 11 14:01 /dev/i2c-0
crw-rw---- 1  root video   243,  0 Apr 11 14:01 /dev/media0
crw-rw---- 1  root dialout   4, 64 Apr 11 14:01 /dev/ttyS0
```

```
crw-rw---- 1  root dialout 188,  0 Apr 11 18:56 /dev/ttyUSB0
```

- If you want to see which major numbers have been assigned, you can look at Documentation/admin-guide/devices.txt.
- When the system was installed, all of those device files were created by the **`mknod`** command. To create a new char device named **`'coffee'`** with major/minor number 12 and 2, simply do **`mknod /dev/coffee c 12 2`**.
- You don't *have* to put your device files into `/dev`, but it's done by convention. Linus put his device files in `/dev`, and so should you. However, when creating a device file for testing purposes, it's probably OK to place it in your working directory where you compile the kernel module. Just be sure to put it in the right place when you're done writing the device driver.
- **When a device file is accessed, the kernel uses the major number of the file to determine which driver should be used to handle the access. This means that the kernel doesn't really need to use or even know about the minor number. The driver itself is the only thing that cares about the minor number. It uses the minor number to distinguish between different pieces of hardware.**

- **Dynamic allocation of Major numbers**
  - Some major device numbers are statically assigned to the most common devices. A list of those devices can be found in `Documentation/devices.txt` within the kernel source tree. The chances of a static number having already been assigned for the use of your new driver are small, however, and new numbers are not being assigned. So, as a driver writer, you have a choice: you can simply pick a number that appears to be unused, or you can allocate major numbers in a dynamic manner.
  - Picking a number may work as long as the only user of your driver is you; once your driver is more widely deployed, a randomly picked major number will lead to conflicts and trouble.
  - Thus, for new drivers, we strongly suggest that you use dynamic allocation to obtain your major device number, rather than choosing a number randomly from the ones that are currently free. In other words, your drivers should almost certainly be using `alloc_chrdev_region` rather than `register_chrdev_region`.
  - The disadvantage of dynamic assignment is that you can't create the device nodes in advance, because the major number assigned to your module will vary. For normal use of the driver, this is hardly a problem, because once the number has been assigned, you can read it from `/proc/devices`.

- **Character Device Files**
  - **The `file_operations` Structure**
    - The file_operations structure is defined in **`include/linux/fs.h`**, and holds pointers to functions defined by the driver that perform various operations on the device. Each field of the structure corresponds to the address of some function defined by the driver to handle a requested operation.

- For example, every character driver needs to define a function that reads from the device. The `file_operations` structure holds the address of the module's function that performs that operation. Here is what the definition looks like for kernel 5.0.3:

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
    ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
    int (*iterate) (struct file *, struct dir_context *);
    int (*iterate_shared) (struct file *, struct dir_context *);
    __poll_t (*poll) (struct file *, struct poll_table_struct *);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    unsigned long mmap_supported_flags;
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, loff_t, loff_t, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long,
unsigned long, unsigned long);
    int (*check_flags)(int);
    int (*flock) (struct file *, int, struct file_lock *);
    ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *,
size_t, unsigned int);
    ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *,
size_t, unsigned int);
    int (*setlease)(struct file *, long, struct file_lock **, void **);
    long (*fallocate)(struct file *file, int mode, loff_t offset,
              loff_t len);
    void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
    unsigned (*mmap_capabilities)(struct file *);
#endif
    ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
            loff_t, size_t, unsigned int);
    loff_t (*remap_file_range)(struct file *file_in, loff_t pos_in,
                   struct file *file_out, loff_t pos_out,
                   loff_t len, unsigned int remap_flags);
    int (*fadvise)(struct file *, loff_t, loff_t, int);
} __randomize_layout;
```

- It can be noticed that the signature of the functions differs from the system call that the user uses. The operating system sits between the user and the device driver to simplify implementation in the device driver.
- **open** does not receive the parameter path or the various parameters that control the file opening mode. Similarly, **read, write, release, ioctl, lseek** do not receive as a parameter a file descriptor. Instead, these routines receive as

parameters two structures: `file` and `inode`. Both structures represent a file, but from different perspectives.

- Most parameters for the presented operations have a direct meaning:
  - `file` and `inode` identifies the device type file;
  - `size` is the number of bytes to be read or written;
  - `offset` is the displacement to be read or written (to be updated accordingly);
  - `user_buffer` user buffer from which it reads / writes;
  - `whence` is the way to seek (the position where the search operation starts);
  - `cmd` and `arg` are the parameters sent by the users to the ioctl call (IO control).
- Some operations are not implemented by a driver. For example, a driver that handles a video card won't need to read from a directory structure. **The corresponding entries in the file_operations structure should be set to NULL.**
- There is a gcc extension that makes assigning to this structure more convenient. You'll see it in modern drivers, and may catch you by surprise. This is what the new way of assigning to the structure looks like:

```
struct file_operations fops = {
    read: device_read,
    write: device_write,
    open: device_open,
    release: device_release
};
```

- However, there's also a C99 way of assigning to elements of a structure, and this is definitely preferred over using the GNU extension. You should use this syntax in case someone wants to port your driver. It will help with compatibility:

```
struct file_operations fops = {
    .read = device_read,
    .write = device_write,
    .open = device_open,
    .release = device_release
};
```

- The meaning is clear, and you should be aware that any member of the structure which you don't explicitly assign will be initialized to NULL by gcc. An instance of struct `file_operations` containing pointers to functions that are used to implement `read`, `write`, `open` ... syscalls is commonly named **`fops`**.

- **The `file` structure**
  - Each device is represented in the kernel by a file structure, which is defined in **`include/linux/fs.h`**. Be aware that a file is a kernel level structure and never appears in a user space program.
  - It's not the same thing as a FILE, which is defined by `glibc` and would never appear in a kernel space function. Also, its name is a bit misleading; it represents

an abstract open **'file'**, not a file on a disk, which is represented by a structure named inode.

- An instance of a struct file is commonly named **filp**. You'll also see it referred to as `struct file file`. Resist the temptation.

```
struct file {
    union {
        struct llist_node   fu_llist;
        struct rcu_head     fu_rcuhead;
    } f_u;
    struct path     f_path;
    struct inode        *f_inode;   /* cached value */
    const struct file_operations    *f_op;

    /*
     * Protects f_ep_links, f_flags.
     * Must not be taken from IRQ context.
     */
    spinlock_t      f_lock;
    enum rw_hint        f_write_hint;
    atomic_long_t       f_count;
    unsigned int        f_flags;
    fmode_t         f_mode;
    struct mutex        f_pos_lock;
    loff_t          f_pos;
    struct fown_struct  f_owner;
    const struct cred   *f_cred;
    struct file_ra_state    f_ra;

    u64         f_version;
#ifdef CONFIG_SECURITY
    void            *f_security;
#endif
    /* needed for tty driver, and maybe others */
    void            *private_data;

#ifdef CONFIG_EPOLL
    /* Used by fs/eventpoll.c to link all the hooks to this file */
    struct list_head    f_ep_links;
    struct list_head    f_tfile_llink;
#endif /* #ifdef CONFIG_EPOLL */
    struct address_space    *f_mapping;
    errseq_t        f_wb_err;
} __randomize_layout
  __attribute__((aligned(4)));  /* lest something weird decides that 2 is OK */
```

- **Registering / Unregistering a Device**
  - As discussed earlier, char devices are accessed through device files, usually located in `/dev`. The major number tells you which driver handles which device file. The minor number is used only by the driver itself to differentiate which device it's operating on, just in case the driver handles more than one device.
  - Adding a driver to your system means registering it with the kernel. This is synonymous with assigning it a major number during the module's initialization.
  - Before kernel 2.6, we were using `register_chrdev()` to allocate and register the file operations structure. But this method is history now.

- **Why don't we use `register_chrdev()` ?**
  - `register_chrdev()` still works in kernel 2.6, but the problem in kernel 2.4 was, we have limited major number allocation. We had only 8-bit for both major and minor numbers.
  - So, in total we could have only 255 major or minor numbers. Whereas in the new methods, we have 32 bit for major and minor numbers. (12 bits for major, 20 bits for minor).
- Below are the type and type/functions introduced in kernel 2.6

1. `dev_t`

- `dev_t` is the data type introduced in kernel 2.6. It's nothing but a 32 bit integer, in which the first 12 bits holds the major number and the remaining 20 bits holds the minor number.

```
/* <kernel-src>/include/linux/types.h */

typedef u32 __kernel_dev_t;

typedef __kernel_dev_t   dev_t;
```

```
/* Example */

dev_t dev;
dev = 0x0F800001 (For Major number = 248 and Minor number = 1)
MAJOR(dev) = 0x0F800001 >> 20 = 0x0F8 = 248
MINOR(dev) = 0x0F800001  & 0x000FFFFF = 0x1 = 1
```

- Available macros for translation into `dev_t` :

a. `MKDEV(int major, int minor);`

- Given two integers - major and minor numbers, **MKDEV** combines them into one 32 bit `dev_t` compatible number.

b. `MAJOR(dev_t dev);`

- The macro **MAJOR** accepts a `dev_t` type number which is 32 bits, and returns a major number

c. `MINOR(dev_t dev);`

- The macro **MINOR** accepts a `dev_t` type number which is 32 bits, and returns a minor number

```
/* linux-src/include/linux/kdev_t.h */

#define MINORBITS   20
#define MINORMASK   ((1U << MINORBITS) - 1)

#define MAJOR(dev)   ((unsigned int) ((dev) >> MINORBITS))
#define MINOR(dev)   ((unsigned int) ((dev) & MINORMASK))
#define MKDEV(ma,mi)     (((ma) << MINORBITS) | (mi))
```

2. `int register_chrdev_region (dev_t first, unsigned count, const char * name);`

- One of the first things your driver will need to do when setting up a char device is to obtain one or more device numbers to work with. The necessary function for this task is `register_chrdev_region`, which is declared in `<linux/fs.h>`.
- Here, `first` is the beginning device number of the range you would like to allocate. The minor number portion of first is often 0, but there is no requirement to that effect.
- `count` is the total number of contiguous device numbers you are requesting. Note that, if count is large, the range you request could spill over to the next major number; but everything will still work properly as long as the number range you request is available.
- Finally, `name` is the name of the device that should be associated with this number range; it will appear in **`/proc/devices and sysfs`**.
- As with most kernel functions, the return value from `register_chrdev_region` will be 0 if the allocation was successfully performed. In case of error, a negative error code will be returned, and you will not have access to the requested region.
- `register_chrdev_region` works well if you know ahead of time exactly which device numbers you want. Often, however, you will not know which major numbers your device will use; there is a constant effort within the Linux kernel development community to move over to the use of dynamically-allocated device numbers.
- The kernel will happily allocate a major number for you on the fly, but you must request this allocation by using a different function - `alloc_chrdev_region()`.

3. **`int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name);`**
   - With this function, `dev` is an output-only parameter that will, on successful completion, hold the first number in your allocated range.
   - `firstminor` should be the requested first minor number to use; it is usually 0. The *count* and *name* parameters work like those given to `request_chrdev_region()`.
   - The return code of `alloc_chrdev_region` is the same as `register_chrdev_region`.

4. **`void unregister_chrdev_region(dev_t first, unsigned int count);`**
   - Regardless of how you allocate your device numbers, you should free them when they are no longer in use. Device numbers are freed with - `unregister_chrdev_region()`.
   - The usual place to call `unregister_chrdev_region` would be in your module's cleanup function.
   - We can't allow the kernel module to be rmmod'ed whenever root feels like it. If the device file is opened by a process and then we remove the kernel module, using the file would cause a call to the memory location where the appropriate function (read/write) used to be.
   - If we're lucky, no other code was loaded there, and we'll get an ugly error message. If we're unlucky, another kernel module was loaded into the same

location, which means a jump into the middle of another function within the kernel. The results of this would be impossible to predict, but they can't be very positive.

- Normally, when you don't want to allow something, you return an error code (a negative number) from the function which is supposed to do it. With `cleanup_module` that's impossible because it's a void function.
- However, there's a counter which keeps track of how many processes are using your module. You can see what it's value is by looking at `the 3rd field of /proc/modules`. If this number isn't zero, rmmod will fail.
- Note that you don't have to check the counter from within `cleanup_module` because the check will be performed for you by the system call `sys_delete_module`, defined in `linux/module.c`. You shouldn't use this counter directly, but there are functions defined in `linux/module.h`, which let you increase, decrease and display this counter:
    a. `try_module_get(THIS_MODULE)`: Increment the use count.
    b. `module_put(THIS_MODULE)`: Decrement the use count.
- It's important to keep the counter accurate; if you ever do lose track of the correct usage count, you'll never be able to unload the module; it's now reboot time, boys and girls. This is bound to happen to you sooner or later during a module's development.
- **The above functions allocate device numbers for your driver's use, but they do not tell the kernel anything about what you will actually do with those numbers. Before a user-space program can access one of those device numbers, your driver needs to connect them to its internal functions that implement the device's operations.**

5. **cdev**
    - The association of device numbers with specific devices happens by way of the `cdev` structure, found in `<linux/cdev.h>`.
    - `cdev` is newly introduced in kernel 2.6. This cdev structure is an internal representation of a char device.
    - It has a member field as a struct file operations pointer. You have to create and populate your file operations structure and assign the address of that structure to this member.

```
/* <kernel-src>/include/kernel/cdev.h */

struct cdev {
    struct kobject kobj;
    struct module *owner;
    const struct file_operations *ops;
    struct list_head list;
    dev_t dev;
    unsigned int count;
} __randomize_layout;
```

6. **struct cdev *cdev_alloc(void);**

- Allocates and returns a cdev structure, or NULL on failure. Then, you need to give `an ops` pointer.

```
/* Example */

struct cdev *my_dev = cdev_alloc();

  if (my_dev != NULL) {
      my_dev->ops = &my_fops;  /* The file_operations structure */
      my_dev->owner = THIS_MODULE;
  }
  else
      /* No memory, we lose */
```

- The owner field of the structure should be initialized to THIS_MODULE to protect against ill-advised module unloads while the device is active.

7. **`void cdev_init(struct cdev *cdev, struct file_operations *fops);`**
   - In the more common usage pattern, however, the `cdev` structure will be embedded within some larger, device-specific structure, and it will be allocated with that structure. In this case, the function to initialize the cdev is `cdev_init()`.
   - Initializes `cdev`, remembering fops, making it ready to add to the system with `cdev_add`.

```
/* Example */

struct mycdev {
    struct cdev cdev;
    int  flag;
};

mycdev f ;
f= kmalloc(100);
cdev_init(f.cdev, &fops);
```

8. **`int cdev_add(struct cdev *cdev, dev_t dev, unsigned int count);`**
   - Once you have the structure set up, it's time to add it to the system using `cdev_add`.
   - `cdev_add` adds the device represented by `dev` to the system, making it live immediately, and can be used by the user.
   - `cdev` is, of course, a pointer to the `cdev` structure; `dev` is the first device number handled by this structure, and `count` is the number of devices it implements.
   - A negative error code is returned on failure.

9. **`void cdev_del(struct cdev *cdev);`**
   - `cdev_del` removes `cdev` from the system. This function should only be called on a `cdev` structure, which has been successfully added to the system with `cdev_add()`.

- If you need to destroy a structure which has not been added in this way (perhaps `cdev_add()` failed), you must, instead, manually decrement the reference count in the structure's kobject with a call like:

```
kobject_put(&cdev->kobj);
```

- **Calling a cdev_del() on a device which is still active (*if, say, a user-space process still has an open file reference to it*) will cause the device to become inaccessible, but it will not actually delete the structure at that time.**
- **The reference count in the structure will keep it around until all the references have gone away. That means that your driver's methods could be called after you have deleted your cdev object - a possibility you should be aware of.**
- The reference count of a cdev structure can be manipulated with:

```
struct kobject *cdev_get(struct cdev *cdev);
void cdev_put(struct cdev *cdev);
```

- Note that these functions change two reference counts: that of the `cdev` structure, and that of the module which owns it. It will be rare for drivers to call these functions, however.

Exchanging data with user space

- The kernel often needs to copy data from userspace to kernel space; for example, when lengthy data structures are passed indirectly in system calls by means of pointers. There is a similar need to write data in the reverse direction from kernel space to user-space.
- This cannot be done simply by passing and de-referencing pointers for two reasons:
  1. First, userspace programs must not access kernel addresses;
  2. And second, there is no guarantee that a virtual page belonging to a pointer from user-space is really associated with a physical page.
- The kernel therefore provides several standard functions to cater for these special situations when data is exchanged between kernel space and userspace. They are shown in summary form in below table:

1. `unsigned long` **`copy_from_user`** `(void * to, const void __user * from, unsigned long n);` • Copies a block of n bytes data from userspace(*from*) to kernel space(*to*). where, ○ *to* : Destination address, in kernel space. ○ *from* : Source address, in user space. ○ *n* : Number of bytes to copy. • It returns a number of bytes that could not be copied. On success, this will be zero. • If some data could not be copied, this function will pad the copied data to the requested size using zero bytes.
2. **`get_user`**`(`*x*`, `*ptr*`)`

• Get a simple variable from user space. where ○ *x* : Variable to store result. ○ *ptr* : Source address, in user space • Depending on pointer type, the kernel decides automatically to transfer 1, 2, 4, or 8 bytes. • This macro copies a single simple variable from user space(*ptr*) to kernel space(*x*). • It supports simple types like `char` and `int`, but not larger data types like *structures* or arrays. • `ptr` must have pointer-to-simple-variable type, and the result of dereferencing *ptr* must be assignable to *x* without a cast. • It returns zero on success, and `-EFAULT` on error. On error, the variable *x* is set to zero.
3. `long` **`strncpy_from_user`** `(char * dst, const char __user * src, long count);` • Copies a NULL-terminated string with a maximum of `count` characters from userspace (*src*) to kernel space (*dst*). where, ○ `dst` : Destination address, in kernel space. This buffer must be at least *count* bytes long. ○ `src` : Source address, in user space. ○ `count` : Maximum number of bytes to copy, including the trailing NULL. • On success, returns the length of the string (*not including the trailing NULL*). • If access to userspace fails, returns -EFAULT (*some data may have been copied*). • If *count* is smaller than the length of the string, copies *count* bytes and returns *count*.
4. **`put_user`**`(x, ptr)` • Write a simple value into user space. Where, ○ `x` : Value to copy to user space. ○ `Ptr` : Destination address, in user space. • This macro copies a single simple value from kernel space(*ptr*) to user space(*x*). the relevant value is determined automatically from the pointer type passed. • It supports simple types like char and int, but not larger data types like structures or arrays • `ptr` must have pointer-to-simple-variable type, and *x* must be assignable to the result of dereferencing *ptr*. • It returns zero on success, or `-EFAULT` on error.
5. `unsigned long` **`copy_to_user`** `(void __user * to, const void * from, unsigned long n);` • Copies a block of *n* bytes data from kernel space (*from)* to userspace (*to*). where, ○ `to` : Destination address, in user space. ○ `from` : Source address, in kernel space. ○ `n` : Number of bytes to copy. • Returns number of bytes that could not be copied. On success, this will be zero.

- **`get_user`** and **`put_user`** function correctly only when applied to pointers to "`simple`" data types such as char, int, and so on. They do not function with "`compound`" data types or arrays because of the pointer arithmetic required (and owing to the necessary implementation optimizations). Before structs can be exchanged between userspace and kernel space, it is necessary to copy the data and then convert it to the correct type by means of typecasts.
- Below table lists additional helper functions for working with strings from user-space. These functions are subject to the same restrictions as the functions for copying data:

Function	Meaning
`unsigned long` **`clear_user`**`(void`	• Zero a block of memory in user space. (Fills the next *n* bytes after *to* with zeros.)

`__user * to, unsigned long n)`	where, ○ `void __user * to` : Destination address, in user space. ○ `unsigned long n` : Number of bytes to zero. ● It returns, number of bytes that could not be cleared. On success, this will be zero.
`strlen_user``(s)`	● Gets the size of a null-terminated string in userspace (*including the terminating character*).
`long` **`strnlen_user`** `(const char __user * s, long n);`	● Gets the size of a null-terminated string in user space, but restricts the search to a maximum of n characters. ● where ○ `s` : The string to measure. ○ `n` : The maximum valid length ● It returns the size of the string INCLUDING the terminating NULL. ● On exception, it returns 0. ● If the string is too long, it returns a value greater than *n*.
`access_ok``(addr, size)`	● Checks if a pointer to a block of memory in user space is valid. where, ○ `addr` : User space pointer to start of block to check ○ `size` : Size of block to check ● Note that, depending on architecture, this function probably just checks that the pointer is in the user space range - after calling this function, memory access functions may still return `-EFAULT` ● It returns true (*nonzero*) if the memory block may be valid, false (*zero*) if it is definitely invalid.

● Character device driver examples

```
/* A demonstration of how to allocate major number and minor number dynamically
during the execution */

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

#define NUM_OF_DEVICES   3
#define CHRDEV_NAME      "chrdev_basic"

static dev_t dev;

static int __init chrdev_init(void)
{
    int ret = 0;
    ret = alloc_chrdev_region(&dev, 0, NUM_OF_DEVICES, CHRDEV_NAME);
    if(ret < 0)
    {
        printk(KERN_ERR "alloc_chrdev_region failed with error - %d\n", ret);
        return -1;
    }

    printk(KERN_INFO "chrdev_basic devices region allocated\n");
    printk(KERN_INFO "Major number: %d\n", MAJOR(dev));

    printk(KERN_INFO "chrdev_basic module loaded\n");
```

```
    return 0;
}


static void __exit chrdev_exit(void)
{
    unregister_chrdev_region(dev, NUM_OF_DEVICES);
    printk(KERN_INFO "Removed chrdev_basic module\n");
}

module_init(chrdev_init);
module_exit(chrdev_exit);

MODULE_AUTHOR("Sunil Vaghela");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Basic character device module");
```

```
/* A sample output */

debian@beaglebone:~/linux-device-drivers/0201-chardev-basic$ sudo insmod
0201_chrdev_basic.ko
debian@beaglebone:~/linux-device-drivers/0201-chardev-basic$ dmesg
[ 3266.049914] chrdev_basic devices region allocated
[ 3266.049934] Major number: 240
[ 3266.049939] chrdev_basic module loaded
debian@beaglebone:~/linux-device-drivers/0201-chardev-basic$
debian@beaglebone:~/linux-device-drivers/0201-chardev-basic$ sudo rmmod
0201_chrdev_basic
debian@beaglebone:~/linux-device-drivers/0201-chardev-basic$ dmesg
[ 3266.049914] chrdev_basic devices region allocated
[ 3266.049934] Major number: 240
[ 3266.049939] chrdev_basic module loaded
[ 3287.347071] Removed chrdev_basic module
debian@beaglebone:~/linux-device-drivers/0201-chardev-basic$
```

```
/* A demonstration on how to use file operations functionality */

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <asm/uaccess.h>                              /* for put_user */

static int chr_dev_open(struct inode *, struct file *);
static int chr_dev_release(struct inode *, struct file *);
static ssize_t chr_dev_read(struct file *, char *, size_t, loff_t *);
static ssize_t chr_dev_write(struct file *, const char *, size_t, loff_t *);

#define DEVICE_NAME         "chardev_fops"    /* Dev name as it appears in
/proc/devices    */
#define BUF_LEN                 24                      /* Max length of the message
from the device */
#define NUM_OF_DEVICES    1

static int device_open = 0;                         /* Is the device open? */
                                                   /* Used to prevent multiple access
to device */
static char msg[BUF_LEN] = { '\0' };               /* The msg the device will give
```

```
when asked */
static char *msgptr = NULL;
static dev_t dev;
static struct cdev* p_cdev = NULL;

static struct file_operations fops = {
    .owner = THIS_MODULE,
    .read = chr_dev_read,
    .write = chr_dev_write,
    .open = chr_dev_open,
    .release = chr_dev_release
};

/*
 * This function is called when the module is loaded
 */
static int __init init_chrdev_fops(void)
{
    int ret = 0;
    ret = alloc_chrdev_region(&dev, 0, NUM_OF_DEVICES, DEVICE_NAME);
    if(ret < 0)
    {
        printk(KERN_ERR "alloc_chrdev_region failed with error - %d\n", ret);
        return -1;
    }

    p_cdev = cdev_alloc();
    if(p_cdev == NULL)
    {
        printk(KERN_ERR "cdev_alloc failed\n");
        return -1;
    }

    p_cdev->owner = THIS_MODULE;
    p_cdev->ops = &fops;

    ret = cdev_add(p_cdev, dev, NUM_OF_DEVICES);
    if(ret < 0)
    {
        printk(KERN_ERR "cdev_add failed with error - %d\n", ret);
        return -1;
    }

    printk(KERN_INFO "I was assigned major number %d. To talk to\n", MAJOR(dev));
    printk(KERN_INFO "the driver, create a dev file with\n");
    printk(KERN_INFO "'mknod /dev/%s c %d 0'.\n", DEVICE_NAME, MAJOR(dev));
    printk(KERN_INFO "Try various minor numbers. Try to cat and echo to\n");
    printk(KERN_INFO "the device file.\n");

    return 0;
}

/*
 * This function is called when the module is unloaded
 */
static void __exit clean_chrdev_fops(void)
{
    printk(KERN_INFO "Cleaning up device...\n");
    cdev_del(p_cdev);
    unregister_chrdev_region(dev, NUM_OF_DEVICES);
```

```
    printk(KERN_INFO "Device cleanup done\n");
    printk(KERN_INFO "Now please delete the device file /dev/%s\n",
           DEVICE_NAME);
}

/*
 * Called when a process tries to open the device file, like
 * "cat /dev/chardev_fops"
 */
static int chr_dev_open(struct inode *inode, struct file *file)
{
    static unsigned int counter = 1;

    if (device_open)
    {
        printk(KERN_WARNING "Device is already in use...try again!!!\n");
        return -EBUSY;
    }

    device_open = 1;
    printk(KERN_DEBUG "%s device opened\n", DEVICE_NAME);
    sprintf(msg, "Hello world : %d\n", counter++);
    msgptr = msg;

    return 0;
}

/*
 * Called when a process closes the device file.
 */
static int chr_dev_release(struct inode *inode, struct file *file)
{
    device_open = 0;        /* We're now ready for our next caller */
    printk(KERN_DEBUG "%s device released\n", DEVICE_NAME);
    return 0;
}

/*
 * Called when a process, which already opened the dev file, attempts to
 * read from it.
 */
static ssize_t chr_dev_read(struct file* filp,    /* see include/linux/fs.h   */
               char __user* buffer,               /* buffer to fill with data */
               size_t length,                     /* length of the buffer     */
               loff_t* offset)
{
    /*
     * Number of bytes actually written to the buffer
     */
    int bytes_read = 0;

    while(*msgptr && length)
    {
        /*
         * The buffer is in the user data segment, not the kernel
         * segment so "*" assignment won't work.  We have to use
         * put_user which copies data from the kernel data segment to
         * the user data segment.
         */
        if((put_user(*(msgptr++), buffer++)) != 0)
```

```
        {
            printk(KERN_ERR "%s: read data failed\n", __func__);
            return -EFAULT;
        }
        length--;
        bytes_read++;
    }

    /*
     * Most read functions return the number of bytes put into the buffer
     */
    return bytes_read;
}

/*
 * Called when a process writes to dev file: echo "hi" > /dev/hello
 */
static ssize_t chr_dev_write(struct file *filp, const char *buff, size_t len,
                             loff_t * off)
{
    printk(KERN_ALERT "Sorry, this operation isn't supported.\n");
    return -EINVAL;
}

module_init(init_chrdev_fops);
module_exit(clean_chrdev_fops);

MODULE_AUTHOR("Sunil Vaghela");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Demonstrate file operations functionality");
```

```
/* A sample output */

debian@beaglebone:~/linux-device-drivers/0202-chardev-fops$ sudo insmod
0202_chardev_fops.ko
debian@beaglebone:~/linux-device-drivers/0202-chardev-fops$ dmesg | tail -n 15
[ 3260.493928] I was assigned major number 240. To talk to
[ 3260.493944] the driver, create a dev file with
[ 3260.493951] 'mknod /dev/chardev_fops c 240 0'.
[ 3260.493956] Try various minor numbers. Try to cat and echo to
[ 3260.493960] the device file.
debian@beaglebone:~/linux-device-drivers/0202-chardev-fops$ sudo mknod
/dev/chardev_fops c 240 0
debian@beaglebone:~/linux-device-drivers/0202-chardev-fops$ cat /dev/chardev_fops
Hello world : 1
debian@beaglebone:~/linux-device-drivers/0202-chardev-fops$ cat /dev/chardev_fops
Hello world : 2
debian@beaglebone:~/linux-device-drivers/0202-chardev-fops$ cat /dev/chardev_fops
Hello world : 3
debian@beaglebone:~/linux-device-drivers/0202-chardev-fops$ cat /dev/chardev_fops
Hello world : 4
debian@beaglebone:~/linux-device-drivers/0202-chardev-fops$ cat /dev/chardev_fops
Hello world : 5
debian@beaglebone:~/linux-device-drivers/0202-chardev-fops$ dmesg
[ 3298.300847] chardev_fops device opened
[ 3298.304320] chardev_fops device released
[ 3299.116443] chardev_fops device opened
[ 3299.119560] chardev_fops device released
[ 3299.388271] chardev_fops device opened
[ 3299.391729] chardev_fops device released
```

```
[ 3299.604693] chardev_fops device opened
[ 3299.607968] chardev_fops device released
[ 3299.796321] chardev_fops device opened
[ 3299.799372] chardev_fops device released
debian@beaglebone:~/linux-device-drivers/0202-chardev-fops$ sudo su
root@beaglebone:/home/debian/linux-device-drivers/0202-chardev-fops# echo "hi" >
/dev/chardev_fops
bash: echo: write error: Invalid argument
root@beaglebone:/home/debian/linux-device-drivers/0202-chardev-fops# echo "hi" >
/dev/chardev_fops
bash: echo: write error: Invalid argument
root@beaglebone:/home/debian/linux-device-drivers/0202-chardev-fops# dmesg
[ 3354.862926] chardev_fops device opened
[ 3354.863042] Sorry, this operation isn't supported.
[ 3354.872560] chardev_fops device released
[ 3356.055460] chardev_fops device opened
[ 3356.055554] Sorry, this operation isn't supported.
[ 3356.064115] chardev_fops device released
root@beaglebone:/home/debian/linux-device-drivers/0202-chardev-fops# exit
debian@beaglebone:~/linux-device-drivers/0202-chardev-fops$ sudo rmmod
0202_chardev_fops
debian@beaglebone:~/linux-device-drivers/0202-chardev-fops$ dmesg
[ 3385.460546] Cleaning up device...
[ 3385.460572] Device cleanup done
[ 3385.460578] Now please delete the device file /dev/chardev_fops
debian@beaglebone:~/linux-device-drivers/0202-chardev-fops$ sudo rm /dev/chardev_fops
debian@beaglebone:~/linux-device-drivers/0202-chardev-fops$
```

- **Automatic creation of device files**

  - Earlier in kernel 2.4, automatic creation of device files was done by the kernel itself, by calling the appropriate APIs of devfs. However, as the kernel evolved, kernel developers realized that device files are more of a user space thing and hence as a policy only the users should deal with it, not the kernel.
  - With this idea, now **the kernel only populates the appropriate device class & device info into the /sys/ window for the device under consideration. And then, the user space needs to interpret it and take an appropriate action.** In most Linux desktop systems, the `udev daemon` picks up that information and accordingly creates the device files.
  - `udev` can be further configured using its configuration files to tune the device file names, their permissions, their types, etc. So, as far as the driver is concerned, the appropriate `/sys/` entries need to be populated using the Linux device model APIs declared in `<linux/device.h>` and the rest would be handled by udev.
  - Device class is created as follows:

```
struct class* class_create(struct module*  owner, const char*  name);
```

  - Where, `owner` is a pointer to the module that is to `"own"` this struct class, and *name* is a pointer to a string for the name of this class.
  - and then the device info `(<major, minor>)` under this class is populated by:

```
struct device* device_create(struct class*  class,  struct device*  parent,  dev_t
devt, void* drvdata, const char *  fmt, ...);
```

  - This API creates a device and registers it with sysfs.

- where, `class` is the pointer to the struct class that this device should be registered to. `Parent` is the pointer to the parent struct device of this new device, if any. `devt` is a `dev_t` for the char device to be added. and `fmt` is a string for the device's name.
- The corresponding complementary or the inverse calls, which should be called in chronologically reverse order, are as follows:

```
void device_destroy (struct class* class,  dev_t  devt);
void class_destroy (struct class* class);
```

- In case of multiple minors, *device_create()* and *device_destroy()* APIs may be put in for-loop, and the *<fmt>* string could be useful. For example, the *device_create()* call in a for-loop indexed by 'i' could be as follows:

```
device_create(cl_ptr, NULL, MKDEV(MAJOR(first), MINOR(first) + i), NULL,
"mynull%d", i);
```

- An demonstration example,

```
/* A demonstration of device class APIs to create the device file automatically
when module loaded and removed when module unloaded */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/types.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/device.h>
#include <linux/cdev.h>

#define CHARDEV_NAME              "0203_chardev"
#define CHARDEV_CLASS_NAME          "0203_chardev_class"
#define DEVICE_FILE_NAME          "chrdev_null"
#define NUM_OF_DEVICES             1

static dev_t first;                 /* first device number */
static struct cdev c_dev;           /* character device structure */
static struct class* devclass;        /* device class */

static int my_open(struct inode* i, struct file* f)
{
    printk(KERN_INFO "Driver: open()\n");
    return 0;
}

static int my_close(struct inode* i, struct file* f)
{
    printk(KERN_INFO "Driver: close()\n");
    return 0;
}

static ssize_t my_read(struct file* f, char __user* buf, size_t len, loff_t* off)
{
    printk(KERN_INFO "Driver: read()\n");
    return 0;
}

static ssize_t my_write(struct file* f, const char __user* buf, size_t len,
loff_t* off)
```

```
{
    printk(KERN_INFO "Driver: write()\n");
    return len;
}

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

static int __init chardev_class_example_init(void) /* Constructor */
{
    int ret;
    struct device* dev_ret;

    if((ret = alloc_chrdev_region(&first, 0, NUM_OF_DEVICES, CHARDEV_NAME)) < 0)
    {
        printk(KERN_ERR "alloc_chrdev_region() failed, error:%d\n", ret);
        return ret;
    }

    if(IS_ERR(devclass = class_create(THIS_MODULE, CHARDEV_CLASS_NAME)))
    {
        printk(KERN_ERR "class_create() failed\n");
        ret = PTR_ERR(devclass);
        goto UNREG_CHRDEV;
    }

    if(IS_ERR(dev_ret = device_create(devclass, NULL, first, NULL,
DEVICE_FILE_NAME)))
    {
        printk(KERN_ERR "device_create() failed\n");
        ret = PTR_ERR(dev_ret);
        goto DEST_CLASS;
    }

    cdev_init(&c_dev, &fops);
    if((ret = cdev_add(&c_dev, first, NUM_OF_DEVICES)) < 0)
    {
        printk(KERN_ERR "cdev_add() failed, error:%d\n", ret);
        goto DEST_DEVICE;
    }
    printk(KERN_INFO "0203_chardev: device registered\n");

    /* success */
    return 0;

DEST_DEVICE:
    device_destroy(devclass, first);
DEST_CLASS:
    class_destroy(devclass);
UNREG_CHRDEV:
    unregister_chrdev_region(first, NUM_OF_DEVICES);

    return ret;
}
```

```
static void __exit chardev_class_example_exit(void) /* Destructor */
{
    cdev_del(&c_dev);
    device_destroy(devclass, first);
    class_destroy(devclass);
    unregister_chrdev_region(first, NUM_OF_DEVICES);
    printk(KERN_INFO "0203_chardev: device unregistered\n");
}

module_init(chardev_class_example_init);
module_exit(chardev_class_example_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Sunil Vaghela <sunilvaghela09@gmail.com>");
MODULE_DESCRIPTION("A demonstration of device class APIs");
```

```
/* A sample output */

debian@beaglebone:~/linux-device-drivers/0203-chardev-using-class$ ls
/dev/chrdev_null -l
ls: cannot access '/dev/chrdev_null': No such file or directory
debian@beaglebone:~/linux-device-drivers/0203-chardev-using-class$
debian@beaglebone:~/linux-device-drivers/0203-chardev-using-class$ sudo insmod
0203_chardev_using_class.ko
debian@beaglebone:~/linux-device-drivers/0203-chardev-using-class$ sudo dmesg -c
[17530.774277] 0203_chardev: device registered
debian@beaglebone:~/linux-device-drivers/0203-chardev-using-class$ ls
/dev/chrdev_null -l
crw------- 1 root root 240, 0 Mar 21 21:04 /dev/chrdev_null
debian@beaglebone:~/linux-device-drivers/0203-chardev-using-class$ sudo cat
/dev/chrdev_null
debian@beaglebone:~/linux-device-drivers/0203-chardev-using-class$ sudo dmesg -c
[17583.074955] Driver: open()
[17583.075049] Driver: read()
[17583.075125] Driver: close()
debian@beaglebone:~/linux-device-drivers/0203-chardev-using-class$ sudo su
root@beaglebone:/home/debian/linux-device-drivers/0203-chardev-using-class# echo
"hi" > /dev/chrdev_null
root@beaglebone:/home/debian/linux-device-drivers/0203-chardev-using-class# exit
debian@beaglebone:~/linux-device-drivers/0203-chardev-using-class$ sudo dmesg -c
[17605.969932] Driver: open()
[17605.970059] Driver: write()
[17605.970085] Driver: close()
debian@beaglebone:~/linux-device-drivers/0203-chardev-using-class$ sudo rmmod
0203_chardev_using_class
debian@beaglebone:~/linux-device-drivers/0203-chardev-using-class$ sudo dmesg -c
[17636.653937] 0203_chardev: device unregistered
debian@beaglebone:~/linux-device-drivers/0203-chardev-using-class$ ls
/dev/chrdev_null -l
ls: cannot access '/dev/chrdev_null': No such file or directory
debian@beaglebone:~/linux-device-drivers/0203-chardev-using-class$
```

Chapter-2 Bridges between Kernel and User space

IOCTL

- There are many ways to Communicate between the User space and Kernel Space, they are:
  - **IOCTL**
  - **Procfs**
  - **Sysfs**
  - **Configfs**
  - **Debugfs**
  - **Sysctl**
  - **UDP Sockets**
  - **Netlink Sockets,** let us see **IOCTL** now.
- **IOCTL** is referred to as Input and Output Control, which is used for talking to device drivers.
- This system call, available in most driver categories. The major use of this is in case of handling some specific operations of a device for which the kernel does not have a system call by default.
- Some real-time applications of ioctl are ejecting the media from a "cd" drive, to change the Baud Rate of Serial port, Adjust the Volume, Reading or Writing device registers, etc. We already have the write and read function in our device driver. But it is not enough for all cases.

- **Steps involved in IOCTL**
  - There are some steps involved to use **IOCTL**.
    - Create **IOCTL** command in driver
    - Write **IOCTL** function in the driver
    - Create **IOCTL** command in a Userspace application
    - Use the **IOCTL** system call in a Userspace

- **Create IOCTL Command in the Driver**
  - To implement a new ioctl command we need to follow the following steps:

1. Define the ioctl code

```
#define "ioctl name" __IOX("magic number","command number","argument type")
```

- where `IOX` can be,
  - **IO:** an `IOCTL` with no parameters
  - **IOW:** an `IOCTL` with write parameters (`copy_from_user`)
  - **IOR:** an `IOCTL` with read parameters (`copy_to_user`)
  - **IOWR:** an `IOCTL` with both write and read parameters
- The **Magic Number** is a unique number or character that will differentiate our set of ioctl calls from the other ioctl calls. sometimes the major number for the device is used here.

- **Command Number** is the number that is assigned to the ioctl . This is used to differentiate the commands from one another.
- The **argument type** is the type of data.

2. Add the header file **linux/ioctl.h** to make use of the above-mentioned calls.

Example:

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

#define WR_VALUE _IOW('a', 'a', int32_t*)
#define RD_VALUE _IOR('a', 'b', int32_t*)
```

- **Write IOCTL function in the driver**
  - The next step is to implement the ioctl call we defined into the corresponding driver. We need to add the ioctl function to our driver. Find the prototype of the function below.

```
int  ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned
long arg)
```

- where,
  - **inode:** is the inode number of the file being worked on.
  - **file:** is the file pointer to the file that was passed by the application.
  - **cmd:** is the ioctl command that was called from the userspace.
  - **arg:** are the arguments passed from the userspace.
- Within the function `ioctl`, we need to implement all the commands that we defined above (**WR_VALUE, RD_VALUE**). We need to use the same commands in the `switch` statement which is defined above.
- Then we need to inform the kernel that the ioctl calls are implemented in the function `etx_ioctl`. This is done by making the `fops` pointer `unlocked_ioctl` to point to `etx_ioctl` as shown below:

```
static long etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
    switch(cmd) {
        case WR_VALUE:
            copy_from_user(&value ,(int32_t*) arg, sizeof(value));
            printk(KERN_INFO "Value = %d\n", value);
            break;
        case RD_VALUE:
            copy_to_user((int32_t*) arg, &value, sizeof(value));
            break;
    }
    return 0;
}

static struct file_operations fops =
{
    .owner          = THIS_MODULE,
    .read           = etx_read,
    .write          = etx_write,
```

```
    .open           = etx_open,
    .unlocked_ioctl = etx_ioctl,
    .release        = etx_release,
};
```

- Now we need to call the new ioctl command from a user application.

- **Create IOCTL command in a Userspace application**

```
/* Example */

#define WR_VALUE _IOW('a', 'a', int32_t*)
#define RD_VALUE _IOR('a', 'b', int32_t*)
```

- **Use IOCTL system call in Userspace**
  - Include the header file `<sys/ioctl.h>`. Now we need to call the new ioctl command from a user application.

```
long ioctl( "file descriptor","ioctl command","arguments");
```

- where,
  - `file descriptor:` This the open file on which the ioctl command needs to be executed, which would generally be device files.
  - `ioctl command:` ioctl command which is implemented to achieve the desired functionality.
  - `arguments:` The arguments that need to be passed to the ioctl command.

```
/* Example */

ioctl(fd, WR_VALUE, (int32_t*) &number);
ioctl(fd, RD_VALUE, (int32_t*) &value);
```

```
/* Demonstration of device driver IOCTL functionality with userspace application */

#include <linux/kernel.h>           /* Kernel debug macros and many more */
#include <linux/init.h>             /* __init* and __exit* macros */
#include <linux/module.h>           /* module_* macros, Required by all modules */
#include <linux/kdev_t.h>           /* MAJOR and MINOR macros */
#include <linux/fs.h>               /* struct file_operations */
#include <linux/cdev.h>             /* struct cdev, cdev_* APIs */
#include <linux/device.h>           /* struct class, class_*, device_* APIs */
#include <linux/uaccess.h>          /* copy_to/from_user() */
#include <linux/ioctl.h>                /* __IOX macros */


#define WR_VALUE _IOW('a', 'a', int32_t*)
#define RD_VALUE _IOR('a', 'b', int32_t*)
```

```
int32_t value = 0;

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

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 long etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg);

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

static int etx_open(struct inode *inode, struct file *file)
{
        printk(KERN_INFO "Device File Opened...!!!\n");
        return 0;
}

static int etx_release(struct inode *inode, struct file *file)
{
        printk(KERN_INFO "Device File Closed...!!!\n");
        return 0;
}

static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t
*off)
{
        printk(KERN_INFO "Read Function\n");
        return 0;
}
static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len,
loff_t *off)
{
        printk(KERN_INFO "Write function\n");
        return 0;
}

static long etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
        switch(cmd) {
                case WR_VALUE:
                        if(copy_from_user(&value, (int32_t*) arg, sizeof(value)) == 0) {
                            printk(KERN_INFO "Value = %d\n", value);
               }
               else {
                   printk(KERN_ERR "copy_from_user failed\n");
```

```
            }
                    break;
             case RD_VALUE:
                    if(copy_to_user((int32_t*) arg, &value, sizeof(value)) != 0) {
                printk(KERN_ERR "copy_to_user failed\n");
            }
                    break;
        }
        return 0;
}

static int __init etx_driver_init(void)
{
        /* Allocating Major number */
        if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0) {
                printk(KERN_INFO "Cannot allocate major number\n");
                return -1;
        }
        printk(KERN_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) {
                printk(KERN_INFO "Cannot add the device to the system\n");
                goto r_cdev;
        }

        /* Creating struct class */
        if((dev_class = class_create(THIS_MODULE, "etx_class")) == NULL) {
                printk(KERN_INFO "Cannot create the struct class\n");
                goto r_class;
        }

        /* Creating device */
        if((device_create(dev_class, NULL, dev, NULL, "etx_device")) == NULL) {
                printk(KERN_INFO "Cannot create the Device 1\n");
                goto r_device;
        }
        printk(KERN_INFO "Device Driver Insert...Done!!!\n");
        return 0;

r_device:
        class_destroy(dev_class);
r_class:
        cdev_del(&etx_cdev);
r_cdev:
        unregister_chrdev_region(dev,1);
        return -1;
}

void __exit etx_driver_exit(void)
{
        device_destroy(dev_class,dev);
        class_destroy(dev_class);
        cdev_del(&etx_cdev);
        unregister_chrdev_region(dev, 1);
        printk(KERN_INFO "Device Driver Remove...Done!!!\n");
}
```

```
module_init(etx_driver_init);
module_exit(etx_driver_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Sunil Vaghela <sunilvaghela09@gmail.com>");
MODULE_DESCRIPTION("A simple IOCTL device driver");
```

```
/* test application */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>

#define WR_VALUE _IOW('a', 'a', int32_t*)
#define RD_VALUE _IOR('a', 'b', int32_t*)

int main()
{
        int fd;
        int32_t value, number;
        printf("Opening Driver...\n");
        fd = open("/dev/etx_device", O_RDWR);
        if(fd < 0) {
                printf("Cannot open device file...\n");
                return 0;
        }

        printf("Enter the Value to send: \n");
        scanf("%d",&number);
        printf("Writing Value to Driver...\n");
        ioctl(fd, WR_VALUE, (int32_t*) &number);

        printf("Reading Value from Driver...\n");
        ioctl(fd, RD_VALUE, (int32_t*) &value);
        printf("Value is %d\n", value);

        printf("Closing Driver\n");
        close(fd);
}
```

```
/* A sample output */

debian@beaglebone:~$ ls /dev/etx_device -l
ls: cannot access '/dev/etx_device': No such file or directory
debian@beaglebone:~$ sudo insmod ioctl_rw_custom.ko
debian@beaglebone:~$ dmesg | tail -n 5
[12153.717904] Major = 240 Minor = 0
[12153.718363] Device Driver Insert...Done!!!
debian@beaglebone:~$ ls /dev/etx_device -l
crw------- 1 root root 240, 0 Mar 21 19:35 /dev/etx_device
debian@beaglebone:~$ cd app/
debian@beaglebone:~/app$ ./test_app
Opening Driver...
Cannot open device file...
```

```
debian@beaglebone:~/app$ sudo ./test_app
Opening Driver...
Enter the Value to send:
10
Writing Value to Driver...
Reading Value from Driver...
Value is 10
Closing Driver
debian@beaglebone:~/app$ dmesg | tail -n 5
[12153.717904] Major = 240 Minor = 0
[12153.718363] Device Driver Insert...Done!!!
[12229.913068] Device File Opened...!!!
[12237.335255] Value = 10
[12237.335339] Device File Closed...!!!
debian@beaglebone:~/app$ sudo rmmod ioctl_rw_custom
debian@beaglebone:~/app$ dmesg | tail -n 3
[12237.335255] Value = 10
[12237.335339] Device File Closed...!!!
[12294.077967] Device Driver Remove...Done!!!
debian@beaglebone:~/app$ ls /dev/etx_device -l
ls: cannot access '/dev/etx_device': No such file or directory
debian@beaglebone:~/app$
```

The /proc File System (procfs)

- In Linux, there is an additional mechanism for the kernel and kernel modules to send information to processes --- the **/proc** file system.
- **Originally designed to allow easy access to information about processes (hence the name), it is now used by every bit of the kernel which has something interesting to report, such as /proc/modules which provides the list of modules and /proc/meminfo which stats memory usage statistics.**
- The **/proc** file system (`procfs`) is a special file system in the linux kernel. It's a virtual file system: it is not associated with a block device but exists only in memory. The files in the procfs are there to allow userland programs access to certain information from the kernel (like process information in `/proc/[0-9]`), but also for debug purposes (like `/proc/ksyms`).
- Note that the files in **/proc/sys** are sysctl files: they don't belong to procfs and are governed by a completely different API.

- **Managing procfs entries**
  - This section describes the functions that various kernel components use to populate the procfs with files, symlinks, device nodes, and directories.
  - A minor note before we start: if you want to use any of the procfs functions, be sure to include the below header file!

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

1. Creating a procfs entry

```
struct proc_dir_entry *proc_create(const char *name, umode_t mode, struct proc_dir_entry *parent, const struct file_operations *proc_fops);
```

- where,
  - **name:** The name of the proc entry
  - **mode:** The access mode for proc entry
  - **parent:** The name of the parent directory under `/proc`. If NULL is passed as a parent, the `/proc` directory will be set as a parent.
  - **proc_fops:** The structure in which the file operations for the proc entry will be created.
- For example to create a proc entry by the name "`sample_proc`" under `/proc`, the above function will be defined as below,

```
proc_create("sample_proc", 0666, NULL, &proc_fops);
```

- This proc entry should be created in driver init function.

> If you are using the kernel version below 3.10, please use the below functions to create a proc entry.
>
> 1. `struct proc_dir_entry* create_proc_entry(const char *name, mode_t mode, struct proc_dir_entry *parent);`
>
> - This function creates a regular file with the name *name*, file mode *mode* in the directory *parent*. To create a file in the root of the procfs, use NULL as a parent parameter.
> - When successful, the function will return a pointer to the freshly created `struct proc_dir_entry`; otherwise it will return NULL.
>
> 2. `struct proc_dir_entry* create_proc_read_entry(const char *name, mode_t mode, struct proc_dir_entry* parent, read_proc_t* read_proc, void* data);`
>
> - This function creates a regular file in exactly the same way as `create_proc_entry` does, but also allows to set the read function `read_proc` in one call. This function can set the data as well to pass the `read_proc` callback.

2. Creating a symlink

```
struct proc_dir_entry* proc_symlink(const char* name, struct proc_dir_entry* parent
, const char* dest);
```

- This creates a symlink in the procfs directory parent that points from name to dest. This is similar to **"ln -s <target> <link-name>"** in userspace.

3. Creating a directory

```
struct proc_dir_entry* proc_mkdir(const char* name, struct proc_dir_entry* parent);
```

- Creates a directory name in the procfs directory parent.

4. Removing an entry

```
void remove_proc_entry(const char* name, struct proc_dir_entry* parent);
```

- Removes the entry name in the directory parent from the procfs. Entries are removed by their name, not by the **struct proc_dir_entry** returned by the various create functions.
- Note that this function doesn't recursively remove entries. Be sure to free the data entry from the **struct proc_dir_entry** before **remove_proc_entry** is called.
- For example, to remove the **sample_proc entry** created in proc_create example.

```
remove_proc_entry("sample_proc", NULL);
```

- Here a simple example showing how to use a **/proc** file. This is the HelloWorld for the **/proc** filesystem. There are three parts:
   i. Create the file **/proc/helloworld** in the function **init_module**,
   ii. Return a value (and a buffer) when the file **/proc/helloworld** is read in the callback function **procfs_read**, and
   iii. Delete the file **/proc/helloworld** in the function **cleanup_module**.

```
/* A demonstration to create and use /proc entry */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>   /* Necessary because we use the proc fs */
#include <linux/uaccess.h>   /* For copy_to_user/copy_from_user */

#define procfs_name "helloworld"

static struct proc_dir_entry *pd_entry;      /* Hold information about /proc file */

static int procfile_read(struct file* filp, char *buf, size_t count, loff_t *offp)
{
    static int i = 1;
    int ret = 0;
    char tmp[20] = { 0 };

    printk(KERN_INFO "procfile_read (/proc/%s) called\n", procfs_name);
    if(*offp > 0)
    {
        return 0;
    }

    sprintf(tmp, "helloworld-%d\n", i++);
    if(copy_to_user(buf, tmp, strlen(tmp)))
    {
        printk(KERN_ERR "Error in copy to user\n");
        return -EFAULT;
    }
    ret = *offp = strlen(tmp);

    return ret;
}
```

```
static struct file_operations proc_fops = /* for /proc operations */
{
    owner : THIS_MODULE,
    read  : procfile_read
};

static int __init proc_module_init(void)
{
    pd_entry = proc_create(procfs_name, 0, NULL, &proc_fops);

    if(pd_entry == NULL) {
        remove_proc_entry(procfs_name, NULL);
        printk(KERN_ERR "Could not initialize /proc/%s\n", procfs_name);
        return -ENOMEM;
    }

    printk(KERN_INFO "/proc/%s created\n", procfs_name);
    printk(KERN_INFO "Try using \"cat /proc/%s\"\n", procfs_name);
    return 0;
}

static void __exit proc_module_exit(void)
{
    remove_proc_entry(procfs_name, NULL);
    printk(KERN_INFO "/proc/%s removed\n", procfs_name);
}

module_init(proc_module_init);
module_exit(proc_module_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Sunil Vaghela <sunilvaghela09@gmail.com>");
MODULE_DESCRIPTION("Example to create and read /proc/ entries");
```

```
/* A sample output */

debian@beaglebone:~/linux-device-drivers/0301_procfs_read$ sudo insmod
procfs_read.ko
debian@beaglebone:~/linux-device-drivers/0301_procfs_read$ sudo dmesg -c
[13777.517911] /proc/helloworld created
[13777.517926] Try using "cat /proc/helloworld"
debian@beaglebone:~/linux-device-drivers/0301_procfs_read$ cat /proc/helloworld
helloworld-1
debian@beaglebone:~/linux-device-drivers/0301_procfs_read$ cat /proc/helloworld
helloworld-2
debian@beaglebone:~/linux-device-drivers/0301_procfs_read$ cat /proc/helloworld
helloworld-3
debian@beaglebone:~/linux-device-drivers/0301_procfs_read$ cat /proc/helloworld
helloworld-4
debian@beaglebone:~/linux-device-drivers/0301_procfs_read$ cat /proc/helloworld
helloworld-5
debian@beaglebone:~/linux-device-drivers/0301_procfs_read$ sudo dmesg -c
[13791.073427] procfile_read (/proc/helloworld) called
[13791.076410] procfile_read (/proc/helloworld) called
[13792.272330] procfile_read (/proc/helloworld) called
[13792.275315] procfile_read (/proc/helloworld) called
[13792.946508] procfile_read (/proc/helloworld) called
[13792.947791] procfile_read (/proc/helloworld) called
[13793.192580] procfile_read (/proc/helloworld) called
```

```
[13793.195698] procfile_read (/proc/helloworld) called
[13794.682589] procfile_read (/proc/helloworld) called
[13794.683826] procfile_read (/proc/helloworld) called
debian@beaglebone:~/linux-device-drivers/0301_procfs_read$ sudo rmmod procfs_read
debian@beaglebone:~/linux-device-drivers/0301_procfs_read$ sudo dmesg -c
[13811.271507] /proc/helloworld removed
debian@beaglebone:~/linux-device-drivers/0301_procfs_read$ cat /proc/helloworld
cat: /proc/helloworld: No such file or directory
```

- The `/proc/helloworld` is created when the module is loaded with the function **`create_proc_entry`**. The return value is a **`'struct proc_dir_entry *'`**, and it will be used to configure the file `/proc/helloworld` (for example, the owner of this file). A null return value means that the creation has failed.
- Each time, everytime the file `/proc/helloworld` is read, the function **`procfs_read`** is called. Two parameters of this function are very important:
  - i. The **`buffer`** (the first parameter) and the **`offset`** (the third one). The content of the buffer will be returned to the application which read it (for example the cat command).
  - ii. **`The offset is the current position in the file. If the return value of the function isn't NULL, then this function is called again. So be careful with this function, if it never returns zero, the read function is called endlessly.`**

- **Read and Write a /proc file**
  - We have seen a very simple example for a `/proc` file, where we only read the file `/proc/helloworld`. It's also possible to write in a `/proc` file. It works the same way as read, a function is called when the */proc* file is written. But **`there is a little difference with read, data comes from the user, so you have to import data from user space to kernel space (with copy_from_user or get_user).`**
  - The reason for `copy_from_user` or `get_user` is that Linux memory is segmented. This means that a pointer, by itself, does not reference a unique location in memory, only a location in a memory segment, and you need to know which memory segment it is to be able to use it. There is one memory segment for the kernel, and one for each of the processes.
  - The only memory segment accessible to a process is its own, so when writing regular programs to run as processes, there's no need to worry about segments.
  - When you write a kernel module, normally you want to access the kernel memory segment, which is handled automatically by the system.
  - However, when the content of a memory buffer needs to be passed between the currently running process and the kernel, the kernel function receives a pointer to the memory buffer which is in the process segment.
  - The `put_user` and `get_user` macros allow you to access that memory. These functions handle only one character, you can handle several characters with `copy_to_user` and `copy_from_user`.

- As the buffer (in read or write function) is in kernel space, for the write function you need to import data because it comes from user space, but not for the read function because data is already in kernel space.

Sysfs

- `Sysfs` is a virtual filesystem exported by the kernel, similar to `/proc`. The files in `Sysfs` contain information about devices and drivers. Some files in `Sysfs` are even writable, for configuration and control of devices attached to the system. `Sysfs` is always mounted on `/sys`.
- The directories in `Sysfs` contain the hierarchy of devices, as they are attached to the computer.
- `Sysfs` is the commonly used method to export system information from the kernel space to the user space for specific devices. The sysfs is tied to the device driver model of the kernel.
- Before getting into the `sysfs` we should know about the Kernel Objects.

- **Kernel Objects**
  - The heart of the `sysfs` model is the `kobject`.
  - `Kobject` is the glue that binds the sysfs and the kernel, which is represented by `struct kobject` and defined in `<linux/kobject.h>`.
  - A `struct kobject` represents a kernel object, maybe a device or so, such as the things that show up as a directory in the `sysfs` filesystem.
  - Kobjects are usually embedded in other structures. It is defined as,

```
struct kobject {
    const char      *name;
    struct list_head    entry;
    struct kobject      *parent;
    struct kset     *kset;
    struct kobj_type    *ktype;
    struct kernfs_node  *sd; /* sysfs directory entry */
    struct kref     kref;
#ifdef CONFIG_DEBUG_KOBJECT_RELEASE
    struct delayed_work release;
#endif
    unsigned int state_initialized:1;
    unsigned int state_in_sysfs:1;
    unsigned int state_add_uevent_sent:1;
    unsigned int state_remove_uevent_sent:1;
    unsigned int uevent_suppress:1;
};
```

  - ○ `name` points to the name of this kobject. One can change this using the `kobject_set_name(struct kobject *kobj, const char *name)` function.
  - ○ `parent` is a pointer to this kobject's parent. It is used to build a hierarchy to describe the relationship between objects.
  - ○ `sd` points to a struct `sysfs_dirent` structure that represents this kobject in sysfs inode inside this structure for sysfs.
  - ○ `kref` provides reference counting on the kobject.
  - ○ `ktype` describes the object, and `kset` tells us which set(group) of objects this object belongs to.

- It is the glue that holds much of the device model and its sysfs interface together.
- In short, **Kobj** is used to create kobject directory in **/sys**.

- ## SysFS in Linux
- There are several steps to creating and using sysfs.
  1. Create a directory in **/sys**
  2. Create Sysfs file
- We can use this function (`kobject_create_and_add`) to create a directory.

```
struct kobject * kobject_create_and_add (const char * name, struct kobject *
parent);
```

  where,
  - **name** – the name for the kobject
  - **parent** – the parent kobject of this kobject, if any.
- If you pass **kernel_kobj** to the second argument, it will create the directory under **/sys/kernel/**. If you pass **firmware_kobj** to the second argument, it will create the directory under **/sys/firmware/**. If you pass **fs_kobj** to the second argument, it will create the directory under **/sys/fs/**. If you pass NULL to the second argument, it will create the directory under **/sys/**.
- This function creates a kobject structure dynamically and registers it with sysfs. If the kobject was not able to be created, NULL will be returned.
- When you are finished with this structure, call **kobject_put** and the structure will be dynamically freed when it is no longer being used. Example:

```
struct kobject *kobj_ref;

/*Creating a directory in /sys/kernel/ */
kobj_ref = kobject_create_and_add("etx_sysfs",kernel_kobj); //sys/kernel/etx_sysfs

/*Freeing Kobj*/
kobject_put(kobj_ref);
```

- ## Create Sysfs file
- Using the above function we will create a directory in **/sys**. Now we need to create a **sysfs** file, which is used to interact user space with kernel space through sysfs. So, we can create the **sysfs file** using **sysfs attributes**.
- Attributes are represented as regular files in sysfs with one value per file. There are lots of helper functions that can be used to create the kobject attributes. They can be found in the header file **sysfs.h**

- ## Create attribute
- **Kobj_attribute** is defined as,

```
struct kobj_attribute {
    struct attribute attr;
    ssize_t (*show)(struct kobject *kobj, struct kobj_attribute *attr, char *buf);
    ssize_t (*store)(struct kobject *kobj, struct kobj_attribute *attr, const char
*buf, size_t count);
```

```
};
```

where,

- `attr` – the attribute representing the file to be created,
- `show` – the pointer to the function that will be called when the file is read in *sysfs*,
- `store` – the pointer to the function which will be called when the file is written in sysfs*.*

- We can create an attribute using `__ATTR` macro.

```
#define __ATTR(_name, _mode, _show, _store)
```

- Then we need to write show and store functions.
  - `Store` function will be called whenever we are writing something to the sysfs attribute. See the example.
  - `Show` function will be called whenever we are reading the sysfs attribute. See the example.

- **Create sysfs file**
- To create a single file attribute we are going to use `sysfs_create_file`.

```
int sysfs_create_file (struct kobject* kobj, const struct attribute* attr);
```

where,

- `kobj` – object we're creating for.
- `attr` – attribute descriptor.

- One can use another function `sysfs_create_group` to create a group of attributes.
- Once you have done with a sysfs file, you should delete this file using `sysfs_remove_file`.

```
void sysfs_remove_file ( struct kobject *  kobj, const struct attribute * attr);
```

where,

- `kobj` – object we're creating for.
- `attr` – attribute descriptor.

```
/* Demonstration of sysfs read and write functionality in kernel module */

#include <linux/kernel.h>          /* Kernel debug macros and many more */
#include <linux/init.h>            /* __init* and __exit* macros */
#include <linux/module.h>          /* module_* macros, Required by all modules */
#include <linux/kdev_t.h>          /* MAJOR and MINOR macros */
#include <linux/fs.h>              /* struct file_operations */
#include <linux/cdev.h>            /* struct cdev, cdev_* APIs */
#include <linux/device.h>          /* struct class, class_*, device_* APIs */
#include <linux/uaccess.h>         /* copy_to/from_user() */
#include <linux/sysfs.h>           /* __ATTR macro, sysfs_* APIs */
#include <linux/kobject.h>         /* struct kobject , kernel_kobj */
```

```
volatile int etx_value = 0;

dev_t dev = 0;
static struct class *dev_class;
static struct cdev etx_cdev;
struct kobject *kobj_ref;

static int __init etx_driver_init(void);
static void __exit etx_driver_exit(void);

/*************** Driver Functions **********************/
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);

/*************** Sysfs Functions **********************/
static ssize_t sysfs_show(struct kobject *kobj,
                struct kobj_attribute *attr, char *buf);
static ssize_t sysfs_store(struct kobject *kobj,
                struct kobj_attribute *attr,const char *buf, size_t count);

/* etx_value is name of a sysfs file would be created under /sys/kernel/etx_sysfs/
*/
struct kobj_attribute etx_attr = __ATTR(etx_value, 0660, sysfs_show, sysfs_store);

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

static ssize_t sysfs_show(struct kobject *kobj, struct kobj_attribute *attr,
                          char *buf)
{
        printk(KERN_INFO "Sysfs - Read!!!\n");
        return sprintf(buf, "%d\n", etx_value);
}

static ssize_t sysfs_store(struct kobject *kobj, struct kobj_attribute *attr,
                           const char *buf, size_t count)
{
        printk(KERN_INFO "Sysfs - Write!!!\n");
        sscanf(buf,"%d",&etx_value);
        return count;
}

static int etx_open(struct inode *inode, struct file *file)
{
        printk(KERN_INFO "Device File Opened...!!!\n");
        return 0;
}

static int etx_release(struct inode *inode, struct file *file)
{
```

```
        printk(KERN_INFO "Device File Closed...!!!\n");
        return 0;
}

static ssize_t etx_read(struct file *filp, char __user *buf, size_t len,
                        loff_t *off)
{
        printk(KERN_INFO "Read function\n");
        return 0;
}
static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len,
loff_t *off)
{
        printk(KERN_INFO "Write Function\n");
        return 0;
}


static int __init etx_driver_init(void)
{
        /* Allocating Major number */
        if((alloc_chrdev_region(&dev, 0, 1, "etx_dev")) <0) {
                printk(KERN_INFO "Cannot allocate major number\n");
                return -1;
        }
        printk(KERN_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) {
            printk(KERN_INFO "Cannot add the device to the system\n");
            goto r_class;
        }

        /* Creating struct class */
        if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL) {
            printk(KERN_INFO "Cannot create the struct class\n");
            goto r_class;
        }

        /* Creating device */
        if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
            printk(KERN_INFO "Cannot create the Device 1\n");
            goto r_device;
        }

        /* Creating a directory in /sys/kernel/ */
        kobj_ref = kobject_create_and_add("etx_sysfs", kernel_kobj);

        /*Creating sysfs file for etx_value*/
        if(sysfs_create_file(kobj_ref, &etx_attr.attr)) {
            printk(KERN_INFO"Cannot create sysfs file......\n");
            goto r_sysfs;
        }

        printk(KERN_INFO "Device Driver Insert...Done!!!\n");
    return 0;
```

```
r_sysfs:
        kobject_put(kobj_ref);
        sysfs_remove_file(kernel_kobj, &etx_attr.attr);

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

void __exit etx_driver_exit(void)
{
        kobject_put(kobj_ref);
        sysfs_remove_file(kernel_kobj, &etx_attr.attr);
        device_destroy(dev_class,dev);
        class_destroy(dev_class);
        cdev_del(&etx_cdev);
        unregister_chrdev_region(dev, 1);
        printk(KERN_INFO "Device Driver Remove...Done!!!\n");
}

module_init(etx_driver_init);
module_exit(etx_driver_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Sunil Vaghela <sunilvaghela09@gmail.com>");
MODULE_DESCRIPTION("A simple device driver - SysFs");
```

```
/* A sample output */

root@beaglebone:/home/debian# ls /sys/kernel/
config fscaps irq kexec_crash_size mm profiling rcu_normal  slab uevent_helper
vmcoreinfo debug iommu_groups kexec_crash_loaded kexec_loaded notes rcu_expedited
security tracing uevent_seqnum
root@beaglebone:/home/debian# insmod sysfs_rw.ko
root@beaglebone:/home/debian# ls /sys/kernel/
config etx_sysfs fscaps irq kexec_crash_size mm profiling rcu_normal  slab
uevent_helper  vmcoreinfo debug iommu_groups kexec_crash_loaded kexec_loaded notes
rcu_expedited  security tracing uevent_seqnum
root@beaglebone:/home/debian# ls /sys/kernel/etx_sysfs/etx_value -l
-rw-rw---- 1 root root 4096 Mar 21 17:16 /sys/kernel/etx_sysfs/etx_value
root@beaglebone:/home/debian# dmesg | tail -n 5
[ 3823.097934] Major = 240 Minor = 0
[ 3823.098398] Device Driver Insert...Done!!!
root@beaglebone:/home/debian# cat /sys/kernel/etx_sysfs/etx_value
0
root@beaglebone:/home/debian# cat /sys/kernel/etx_sysfs/etx_value
0
root@beaglebone:/home/debian# dmesg | tail -n 5
[ 3823.097934] Major = 240 Minor = 0
[ 3823.098398] Device Driver Insert...Done!!!
[ 3874.242144] Sysfs - Read!!!
[ 3876.351841] Sysfs - Read!!!
root@beaglebone:/home/debian# echo 5 > /sys/kernel/etx_sysfs/etx_value
root@beaglebone:/home/debian# cat /sys/kernel/etx_sysfs/etx_value
5
root@beaglebone:/home/debian# dmesg | tail -n 5
[ 3823.098398] Device Driver Insert...Done!!!
```

```
[ 3874.242144] Sysfs - Read!!!
[ 3876.351841] Sysfs - Read!!!
[ 3894.155022] Sysfs - Write!!!
[ 3898.256259] Sysfs - Read!!!
root@beaglebone:/home/debian# echo 20 > /sys/kernel/etx_sysfs/etx_value
root@beaglebone:/home/debian# cat /sys/kernel/etx_sysfs/etx_value
20
root@beaglebone:/home/debian# dmesg | tail -n 5
[ 3876.351841] Sysfs - Read!!!
[ 3894.155022] Sysfs - Write!!!
[ 3898.256259] Sysfs - Read!!!
[ 3910.268533] Sysfs - Write!!!
[ 3912.323480] Sysfs - Read!!!
root@beaglebone:/home/debian# rmmod sysfs_rw.ko
root@beaglebone:/home/debian# dmesg | tail -n 5
[ 3894.155022] Sysfs - Write!!!
[ 3898.256259] Sysfs - Read!!!
[ 3910.268533] Sysfs - Write!!!
[ 3912.323480] Sysfs - Read!!!
[ 3934.802055] Device Driver Remove...Done!!!
root@beaglebone:/home/debian#
```

Chapter-3 Device-tree: A data structure for hardware configuration

• Introduction to device-tree

- A **DeviceTree(DT)**, is a **data structure and language** for **describing hardware**. More specifically, it is a description of hardware that is readable by an operating system so that the operating system doesn't need to hard code details of the machine.
- The primary purpose of Device Tree in Linux is to provide a way to describe non-discoverable hardware. This information was previously(before kernel v2.6) hard coded in source code.
- The device tree data is typically created and maintained in a human readable format in .dts source files and .dtsi source include files.
- The device tree source is compiled into a binary format contained in a .dtb blob file. The format of the data in the .dtb blob file is commonly referred to as a **Flattened Device Tree (FDT)**.
- The Linux operating system uses the device tree data to find and register the devices in the system.
- The FDT is accessed in the raw form during the very early phases of boot, but is expanded into a kernel internal data structure known as the **Expanded Device Tree (EDT)** for more efficient access for later phases of the boot and after the system has completed booting.
- Basic device-tree glossary:
  - `.dtb` : For compiled device-tree.
  - `.dts` : Files for board-level definitions
  - `.dtsi` : Files for included files, generally containing SoC-level definitions
  - `dtc` : A tool, the Device Tree Compiler compiles the source into a binary form.

• DTC - Device Tree Compiler

- A tool to compile Device-Tree Source
- Device Tree Compiler - **dtc**, takes as input a device-tree in a given format and outputs a device-tree in another format for booting kernels on embedded systems. Typically, the input format is **"dts"** - a human readable source format, and creates a "dtb", or binary format as output.
- The Device Tree Blob/Binary is produced by the compiler, and is the binary that gets loaded by the bootloader and parsed by the kernel at boot time.
- DTC Source code is located in **scripts/dtc**
- **arch/arm/boot/dts/Makefile** lists which DTBs should be generated at build time.

```
dtb-$(CONFIG_SOC_IMX31) += \
    imx31-bug.dtb \
    imx31-lite.dtb
```

```
dtb-$(CONFIG_SOC_IMX35) += \
    imx35-eukrea-mbimxsd35-baseboard.dtb \
    imx35-pdk.dtb
dtb-$(CONFIG_SOC_IMX50) += \
    imx50-evk.dtb
dtb-$(CONFIG_SOC_IMX51) += \
    imx51-apf51.dtb \
    imx51-apf51dev.dtb \
    imx51-babbage.dtb \
    imx51-digi-connectcore-jsk.dtb \
    imx51-eukrea-mbimxsd51-baseboard.dtb \
    imx51-ts4800.dtb \
…
…
…
```

- `dtc` can be `installed` by this command on linux:

```
$ sudo apt-get install device-tree-compiler
```

- you can `compile dts` or `dtsi` files by below command:

```
$ dtc -I dts -O dtb -o <devicetree_file_name>.dtb <devicetree_file_name>.dts
```

- dtc is also a dtb `decompiler`. you can `convert dts` to `dtb` by below command:

```
$ dtc -I dtb -O dts -o <devicetree_file_name>.dts <devicetree_file_name>.dtb
```

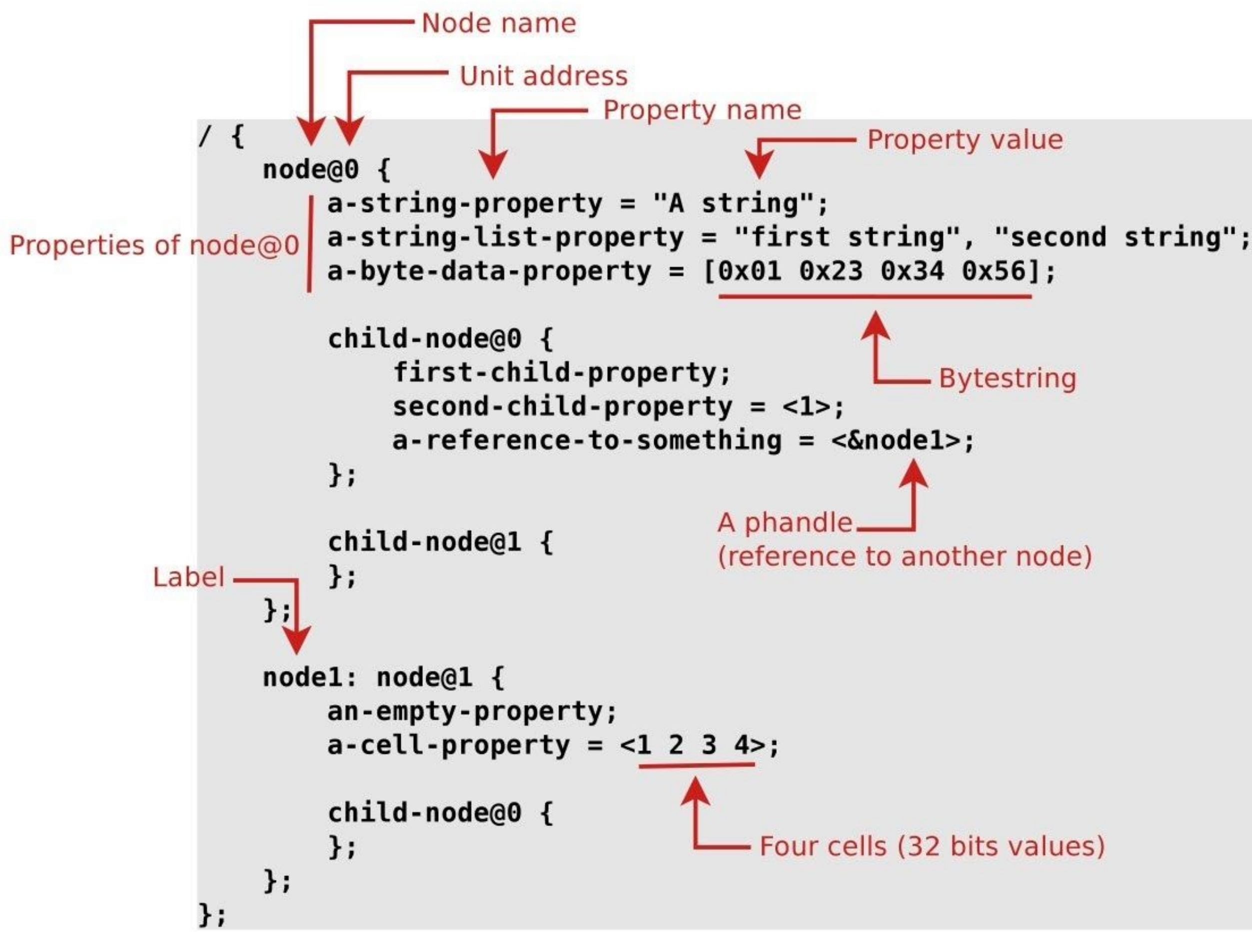
- **Basic Device Tree syntax**
  - The device tree is a simple tree structure of nodes and properties. Properties are key-value pairs, and nodes may contain both properties and child nodes. For example, the following is a simple tree in the .dts format:

```
/dts-v1/;

/ {
    node1 {
        a-string-property = "A string";
        a-string-list-property = "first string", "second string";
        // hex is implied in byte arrays. no '0x' prefix is required
        a-byte-data-property = [01 23 34 56];
        child-node1 {
            first-child-property;
            second-child-property = <1>;
            a-string-property = "Hello, world";
        };
        child-node2 {
        };
    };
    node2 {
        an-empty-property;
        a-cell-property = <1 2 3 4>; /* each number (cell) is a uint32 */
        child-node1 {
        };
```

```
    };
};
```

- This tree is obviously pretty useless because it doesn't describe anything, but it does show the structure of nodes and properties. There is:
  - `A single root node: "/"`
  - `A couple of child nodes: "node1" and "node2"`
  - `A couple of children for node1: "child-node1" and "child-node2"`
  - `A bunch of properties scattered through the tree`
- Below figure provides graphical view of device-tree syntax for more clarity:



- Properties are simple key-value pairs where the value can either be empty or contain an arbitrary byte stream. While data types are not encoded into the data structure, there are a few fundamental data representations that can be expressed in a device tree source file.
  - Text strings (null terminated) are represented with double quotes:
    - `string-property = "a string";`
  - 'Cells' are 32 bit unsigned integers delimited by angle brackets:
    - `cell-property = <0xbeef 123 0xabcd1234>;`
  - Binary data is delimited with square brackets:

   - `binary-property = [0x01 0x23 0x45 0x67];`
 - Data of differing representations can be concatenated together using a comma:
   - `mixed-property = "a string", [0x01 0x23 0x45 0x67], <0x12345678>;`
 - Commas are also used to create lists of strings:
   - `string-list = "red fish", "blue fish";`
- Linux uses DT data for three major purposes:
  1. platform identification,
  2. runtime configuration, and
  3. device population.

- **A simple example**

```
auart0: serial@8006a000 {
        Defines the "programming model" for the device. Allows the
        operating system to identify the corresponding device driver.
        compatible = "fsl,imx28-auart", "fsl,imx23-auart";
        Address and length of the register area.
        reg = <0x8006a000 0x2000>;
        Interrupt number.
        interrupts = <112>;
        DMA engine and channels, with names.
        dmas = <&dma_apbx 8>, <&dma_apbx 9>;
        dma-names = "rx", "tx";
        Reference to the clock.
        clocks = <&clks 45>;
        The device is not enabled.
        status = "disabled";
};
```

Taken from arch/arm/boot/dts/imx28.dtsi

- The compatible string used to bind a device with the driver:

```
static struct of_device_id mxs_auart_dt_ids[] = {
 {
   .compatible = "fsl,imx28-auart",
   .data = &mxs_auart_devtype[IMX28_AUART]
 },
 {
   .compatible = "fsl,imx23-auart",
   .data = &mxs_auart_devtype[IMX23_AUART]
 },
 {
   /* sentinel */
 }
};

 MODULE_DEVICE_TABLE(of, mxs_auart_dt_ids);
// [...]
```

```
  static struct platform_driver mxs_auart_driver = {
  .probe = mxs_auart_probe,
  .remove = mxs_auart_remove,
  .driver = {
  .name = "mxs-auart",
  .of_match_table = mxs_auart_dt_ids,
    },
  };

// Code from drivers/tty/serial/mxs-auart.c
```

- `of_match_device` allows you to get the matching entry in the `mxs_auart_dt_ids` table.

```
static int mxs_auart_probe(struct platform_device *pdev)
{
  const struct of_device_id *of_id =
      of_match_device(mxs_auart_dt_ids, &pdev->dev);
  if (of_id) {
    /* Use of_id->data here */
    // [...]
  }
  // [...]
}
```

- **Some driver examples:**
  - Getting a reference to the clock
    - described by the clocks property

```
s->clk = clk_get(&pdev->dev, NULL);
```

  - Getting the I/O registers resource
    - described by the reg property

```
r = platform_get_resource(pdev, IORESOURCE_MEM, 0);
```

  - Getting the interrupt
    - described by the interrupts property

```
s->irq = platform_get_irq(pdev, 0);
```

  - Get a DMA channel
    - described by the dmas property

```
s->rx_dma_chan = dma_request_slave_channel(s->dev, "rx");
s->tx_dma_chan = dma_request_slave_channel(s->dev, "tx");
```

  - Check some custom property

```
struct device_node *np = pdev->dev.of_node;
if (of_get_property(np, "fsl,uart-has-rtscts", NULL))
```

- **Unit-Address Mystery**
  - The **unit-address** component of the node identifies the base address of the bus on which the node sits. It is the primary address used to access the device.
  - Example:
    - Below is a dts snapshot of **ti,am33xx** processors.([ti-linux-4.14y/am33xx.dtsi]())

```

		gpio0: gpio@44e07000 {
			compatible = "ti,omap4-gpio";
			ti,hwmods = "gpio1";
			gpio-controller;
			#gpio-cells = <2>;
			interrupt-controller;
			#interrupt-cells = <2>;
			reg = <0x44e07000 0x1000>;
			interrupts = <96>;
		};

```

    - The **gpio0** node has **unit-address** of `0x44e07000`, and register size of `0x1000` bytes (4KB).
    - Now, see below register map snapshot of *gpio0* from the datasheet of ti,am33xx processors. ([AM355X ARM MPUs - Technical reference manual]())

TEXAS INSTRUMENTS

ARM Cortex A8 Memory Map   www.ti.com

Table 2-2. L4_WKUP Peripheral Memory Map (continued)

Region Name	Start Address (hex)	End Address (hex)	Size	Description
DMTIMER0	0x44E0_5000	0x44E0_5FFF	4KB	DMTimer0 Registers
	0x44E0_6000	0x44E0_6FFF	4KB	Reserved
GPIO0	0x44E0_7000	0x44E0_7FFF	4KB	GPIO Registers
	0x44E0_8000	0x44E0_8FFF	4KB	Reserved
UART0	0x44E0_9000	0x44E0_9FFF	4KB	UART Registers
	0x44E0_A000	0x44E0_AFFF	4KB	Reserved
I2C0	0x44E0_B000	0x44E0_BFFF	4KB	I2C Registers
	0x44E0_C000	0x44E0_CFFF	4KB	Reserved
ADC_TSC	0x44E0_D000	0x44E0_EFFF	8KB	ADC_TSC Registers
	0x44E0_F000	0x44E0_FFFF	4KB	Reserved

    - You can see the **start-address (0x44E0_7000)** and **size (4KB)** of **gpio0**, which is the same as mentioned in device-tree gpio0 node unit-address and its register size.

- **Device Tree inclusion**
  - Device Tree files are not monolithic, they can be split in several files, including each other.
  - **.dtsi** files are included files, while **.dts** files are final Device Trees.
  - The inclusion works by overlaying the tree of the including file over the tree of the included file.
  - Device Tree inclusion example:

Definition of the AM33xx SoC

```
/ {
     compatible = "ti,am33xx";
     [...]
     ocp {
          uart0: serial@44e09000 {
               compatible = "ti,omap3-uart";
               reg = <0x44e09000 0x2000>;
               interrupts = <72>;
               status = "disabled";
          };
     };
};
                                        am33xx.dtsi
```

+

Definition of the BeagleBone board

```
#include "am33xx.dtsi"

/ {
     compatible = "ti,am335x-bone", "ti,am33xx";
     [...]
     ocp {
          uart0: serial@44e09000 {
               pinctrl-names = "default";
               pinctrl-0 = <&uart0_pins>;
               status = "okay";
          };
     };
};                                         am335x-bone.dts
```

=

Compiled DTB

```
/ {
     compatible = "ti,am335x-bone", "ti,am33xx";
     [...]
     ocp {
          uart0: serial@44e09000 {
               compatible = "ti,omap3-uart";
               reg = <0x44e09000 0x2000>;
               interrupts = <72>;
               pinctrl-names = "default";
               pinctrl-0 = <&uart0_pins>;
               status = "okay";
          };
     };
};                                        am335x-bone.dtb
```

Note: the real DTB is in binary format. Here we show the text equivalent of the DTB contents;

- **Device Tree Structure and Conventions**
  - **Node names**
    - Each node in the device-tree is named according to the following convention:

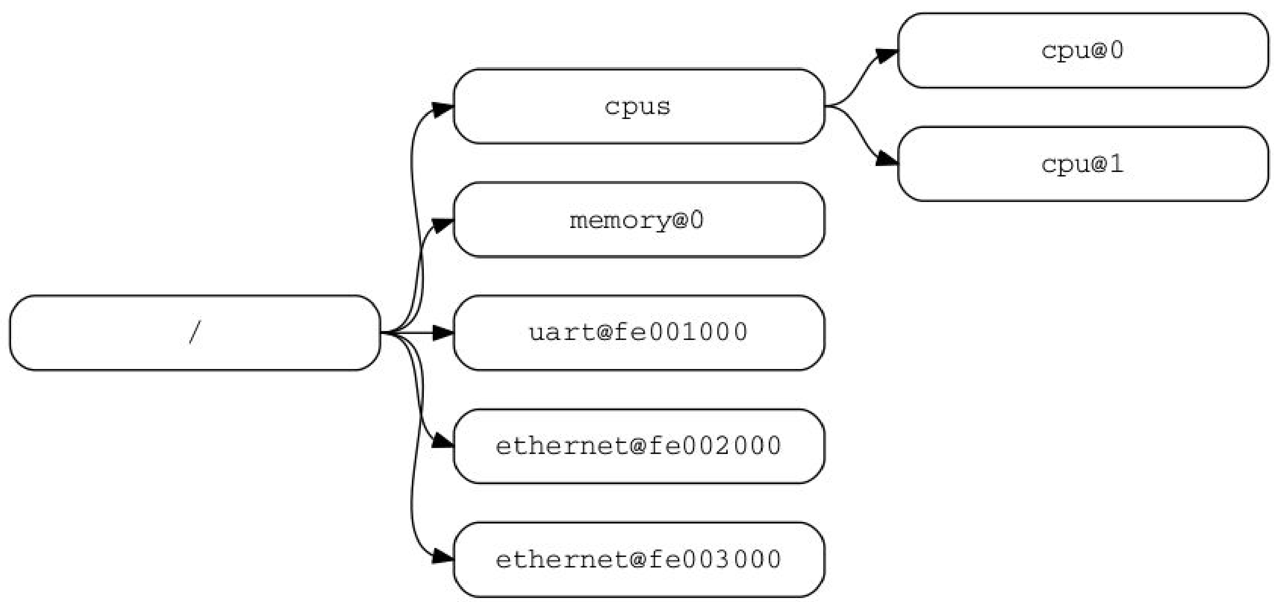
```
node-name@unit-address
```

    - The `node-name` component specifies the name of the node. It shall be 1 to 31 characters in length and consist solely of characters from the set of characters in below table:

Character	Description
0-9	Digit
a-z	Lowercase letters
A-Z	Uppercase letters

,	Comma
.	Period
_	Underscore
+	Plus sign
-	Dash

- The `node-name` shall start with a lower or uppercase character and should describe the general class of device.
- The `unit-address` component of the name is specific to the bus type on which the node sits. It consists of one or more ASCII characters from the set of characters in the above table.
- The `unit-address` must match the first address specified in the `reg` property of the node. If the node has no `reg` property, the `@unit-address` must be omitted and the `node-name` alone differentiates the node from other nodes at the same level in the tree.
- The binding for a particular bus may specify additional, more specific requirements for the format of `reg` and the `unit-address`.
- **The root node does not have a node-name or unit-address. It is identified by a forward slash (/).**



- In above figure:
  - The nodes with the name cpu are distinguished by their `unit-address` values of 0 and 1.
  - The nodes with the name `ethernet` are distinguished by their `unit-address` values of `fe002000` and `fe003000`.

- **Generic Names Recommendation**
  - The name of a node should be somewhat generic, reflecting the function of the device and not its precise programming model.

- If appropriate, the name should be one of the following choices

• adc	• efuse	• led-controller	• pwm
• accelerometer	• endpoint	• light-sensor	• regulator
• atm	• ethernet	• magnetometer	• reset-controller
• audio-codec	• ethernet-phy	• mailbox	• rng
• audio-controller	• fdc	• mdio	• rtc
• backlight	• flash	• memory	• sata
• bluetooth	• gnss	• memory-controller	• scsi
• bus	• gpio	• mmc	• serial
• cache-controller	• gpu	• mmc-slot	• sound
• camera	• gyrometer	• mouse	• spi
• can	• hdmi	• nand-controller	• sram-controller
• charger	• hwlock	• nvram	• ssi-controller
• clock	• i2c	• oscillator	• syscon
• clock-controller	• i2c-mux	• parallel	• temperature-sensor
• compact-flash	• ide	• pc-card	• timer
• cpu	• interrupt-controller	• pci	• touchscreen
• cpus	• iommu	• pcie	• tpm
• crypto	• isa	• phy	• usb
• disk	• keyboard	• pinctrl	• usb-hub
• display	• key	• pmic	• usb-phy
• dma-controller	• keys	• pmu	• video-codec
• dsi	• lcd-controller	• port	• vme
• dsp	• led	• ports	• watchdog
• eeprom	• leds	• power-monitor	• wifi

- **Path Names**
  - A node in the device-tree can be uniquely identified by specifying the full path from the root node, through all descendant nodes, to the desired node.
  - The convention for specifying a device path is:
    - `/node-name-1/node-name-2/node-name-N`
  - For example, in the node-names figure, the device path to cpu #1 would be:
    - `/cpus/cpu@1`
  - **The path to the root node is /.**
  - A unit address may be omitted if the full path to the node is unambiguous. If a client program encounters an ambiguous path, its behavior is undefined.

- **Properties**
  - Each node in the device-tree has properties that describe the characteristics of the node. Properties consist of a name and a value.

  - **Property Names**
    - Property names are strings of 1 to 31 characters from the characters show in table:

Character	Description
`0-9`	Digit
`a-z`	Lowercase letters
`A-Z`	Uppercase letters
`,`	Comma

`.`	Period
`_`	Underscore
`+`	Plus sign
`-`	Dash
`?`	Question mark
`#`	Hash

- Nonstandard property names should specify a unique string prefix, such as a stock ticker symbol, identifying the name of the company or organization that defined the property.
- Examples:
  - **`fsl,channel-fifo-len`**
  - **`ibm,ppc-interrupt-server#s`**
  - **`linux,network-index`**

- **Property Values**
  - A property value is an array of zero or more bytes that contain information associated with the property.
  - Properties might have an empty value if conveying true-false information. In this case, the presence or absence of the property is sufficiently descriptive.
  - Below table describes the set of basic value types defined by the DTSpec:

Value	Description
`<empty>`	• Value is empty. • Used for conveying true-false information, when the presence or absence of the property itself is sufficiently descriptive.
`<u32>`	• A 32-bit integer in big-endian format. • Example: the 32-bit value 0x11223344 would be represented in memory as: ○ address+0 11 ○ address+1 22 ○ address+2 33 ○ address+3 44
`<u64>`	• Represents a 64-bit integer in big-endian format. Consists of two <u32> values where ○ the first value contains the most significant bits of the integer ○ and, the second value contains the least significant bits. • Example: the 64-bit value 0x1122334455667788 would be represented as two cells as: <0x11223344 0x55667788>. The value would be represented in memory as: ○ address+0 11 ○ address+1 22 ○ address+2 33 ○ address+3 44 ○ address+4 55 ○ address+5 66 ○ address+6 77 ○ address+7 88

<string>	• Strings are printable and null-terminated. • Example: the string "hello" would be represented in memory as: ○ address+0 68 'h' ○ address+1 65 'e' ○ address+2 6C 'l' ○ address+3 6C 'l' ○ address+4 6F 'o' ○ address+5 00 '\0'
<prop-encoded-array>	• Format is specific to the property. See the property definition.
<phandle>	• A <u32> value. A phandle value is a way to reference another node in the device-tree. • Any node that can be referenced defines a phandle property with a unique <u32> value. That number is used for the value of properties with a phandle value type.
<stringlist>	• A list of <string> values concatenated together. • Example: The string list "hello","world" would be represented in memory as: ○ Address+00 68 'h' ○ Address+01 65 'e' ○ Address+02 6C 'l' ○ Address+03 6C 'l' ○ Address+04 6F 'o' ○ Address+05 00 '\0' ○ Address+06 77 'w' ○ Address+07 6F 'o' ○ Address+08 72 'r' ○ Address+09 6c 'l' ○ Address+10 64 'd' ○ Address+11 00 '0'

- **Standard properties**
  - DTSpec specifies a set of standard properties for device nodes. These properties are described in detail in this section.
  - **compatible**
    - Property name: `compatible`
    - Value type: `<stringlist>`
    - Description:
      - The `compatible` property value consists of one or more strings that define the specific programming model for the device. This list of strings should be used by a client program `for device driver selection.`
      - The property value consists of a concatenated list of null terminated strings, from most specific to most general.
      - They allow a device to express its compatibility with a family of similar devices, potentially allowing a single device driver to match against several devices.
      - The recommended format is `"manufacturer,model"`, where `manufacturer` is a string describing the name of the manufacturer (such as a stock ticker symbol), and model specifies the model number.

```
/* Example */

compatible = "fsl,mpc8641", "ns16550";
```

- In the above example, an operating system would first try to locate a device driver that supported "fsl,mpc8641". If a driver was not found, it would then try to locate a driver that supported the more general ns16550 device type.

- The top-level compatible property typically defines a compatible string for the board, and then for the SoC.
  - Used to match with the dt_compat field of the DT_MACHINE structure

```
static const char *mxs_dt_compat[] __initdata = {
  "fsl,imx28",
  "fsl,imx23",
  NULL,
};
DT_MACHINE_START(MXS, "Freescale MXS (Device Tree)")
.dt_compat = mxs_dt_compat,
  //[...]
  MACHINE_END
```

  - Can also be used within code to test the machine:

```
if (of_machine_is_compatible("fsl,imx28-evk"))
  imx28_evk_init();
```

- **model**
  - Property name: model
  - Value type: <string>
  - Description:
    - The model property value is a <string> that specifies the manufacturer's model number of the device.
    - The recommended format is: "manufacturer,model", where manufacturer is a string describing the name of the manufacturer (such as a stock ticker symbol), and model specifies the model number.
  - Example:
    - model = "fsl,MPC8349EMITX";

- **phandle**
  - Property name: phandle
  - Value type: <string>
  - Description:
    - The phandle property specifies a numerical identifier for a node that is unique within the device-tree. The phandle property value is used by other nodes that need to refer to the node associated with the property.

```
/* Example */
```

```
pic@10000000 {
    phandle = <1>;
    interrupt-controller;
};
```

A phandle value of 1 is defined. Another device node could reference the pic node with a phandle value of 1:

```
another-device-node {
    interrupt-parent = <1>;
};
```

Note: Most device-trees in DTS will not contain explicit phandle properties. The DTC tool automatically inserts the phandle properties when the DTS is compiled into the binary DTB format.

- **status**
  - Property name: `phandle`
  - Value type: `<string>`
  - Description:
    - The status property indicates the operational status of a device. Valid values are listed and defined in below table:

Value	Description
"okay"	• Indicates the device is operational.
"disabled"	• Indicates that the device is not presently operational, but it might become operational in the future. (for example, something is not plugged in, or switched off). • Refer to the device binding for details on what disabled means for a given device.
"reserved"	• Indicates that the device is operational, but should not be used. • Typically this is used for devices that are controlled by another software component, such as platform firmware.
"fail"	• Indicates that the device is not operational. A serious error was detected in the device, and it is unlikely to become operational without repair.
"fail-sss"	• Indicates that the device is not operational. A serious error was detected in the device and it is unlikely to become operational without repair. • The sss portion of the value is specific to the device and indicates the error condition detected.

- **#address-cells and #size-cells**
  - Property name: `#address-cells, #size-cells`
  - Value type: `<u32>`

- Description:
  - The **#address-cells** and **#size-cells** properties **may be used in any device node that has children in the device-tree hierarchy and describes how child device nodes should be addressed.**
  - The **#address-cells** property defines the number of **<u32>** cells used to encode the address field in a child node's reg property.
  - The **#size-cells** property defines the number of **<u32>** cells used to encode the size field in a child node's reg property.
  - The **#address-cells** and **#size-cells** properties are not inherited from ancestors in the device-tree. They shall be explicitly defined.
  - A DTSpec-compliant boot program shall supply **#address-cells** and **#size-cells** on all nodes that have children.
  - If missing, a client program should assume a default value of 2 for **#address-cells**, and a value of 1 for **#size-cells**.

```
/* Example */

soc {
    #address-cells = <1>;
    #size-cells = <1>;

    serial@4600 {
        compatible = "ns16550";
        reg = <0x4600 0x100>;
        clock-frequency = <0>;
        interrupts = <0xA 0x8>;
        interrupt-parent = <&ipic>;
    };
};
```

  - In this example, the **#address-cells** and **#size-cells** properties of the soc node are both set to 1.
  - This setting specifies that one cell is required to represent an address and one cell is required to represent the size of nodes that are children of this node.
  - The serial device **reg** property necessarily follows this specification set in the parent (SoC) node - the address is represented by a single cell (0x4600), and the size is represented by a single cell (0x100).

- **reg**
  - Property name: **reg**
  - Value type: **<prop-encoded-array>** encoded as an arbitrary number of **(address, length)** pairs.
  - Description:
    - The **reg** property describes the address of the device's resources within the address space defined by its parent bus.
    - Most commonly this means the offsets and lengths of memory-mapped IO register blocks, but may have a different meaning on some bus types. Addresses in the address space defined by the root node are CPU real addresses.

- The value is a **<prop-encoded-array>**, composed of an arbitrary number of pairs of address and length, **<address length>**.
- The number of **<u32>** cells required to specify the address and length are bus-specific and are specified by the **#address-cells** and **#size-cells** properties in the parent of the device node.
- **If the parent node specifies a value of 0 for #size-cells, the length field in the value of reg shall be omitted.**

- Example:
  - Suppose a device within a system-on-a-chip had two blocks of registers, a 32-byte block at offset `0x3000` in the SOC and a 256-byte block at offset `0xFE00`.
  - The reg property would be encoded as follows (assuming **#address-cells** and **#size-cells** values of 1):
  - **reg** = `<0x3000 0x20 0xFE00 0x100>`;

- **virtual-reg**
  - Property name: **virtual-reg**
  - Value type: **<u32>**
  - Description:
    - The **virtual-reg** property specifies an effective address that maps to the first physical address specified in the **reg** property of the device node.
    - This property enables boot programs to provide client programs with virtual-to-physical mappings that have been set up.

- **ranges**
  - Property name: **ranges**
  - Value type: **<empty>** or **<prop-encoded-array>** encoded as an arbitrary number of **(child-bus-address, parent-bus-address, length)** triplets.
  - Description:
    - The **ranges** property provides a means of defining a mapping or translation between the address space of the bus (the child address space) and the address space of the bus node's parent (the parent address space).
    - The format of the value of the **ranges** property is an arbitrary number of triplets of **(child-bus-address, parent-bus-address, length)**
      - The **child-bus-address** is a physical address within the child bus' address space. The number of cells to represent the address is bus dependent and can be determined from the **#address-cells** of this node (the node in which the ranges property appears).
      - The **parent-bus-address** is a physical address within the parent bus' address space. The number of cells to represent the parent address is bus dependent and can be determined from the **#address-cells** property of the node that defines the parent's address space.
      - The length specifies the size of the range in the child's address space. The number of cells to represent the size can be determined from the **#size-cells** of this node (the node in which the ranges property appears).

- If the property is defined with an <empty> value, it specifies that the parent and child address space is identical, and no address translation is required.
- If the property is not present in a bus node, it is assumed that no mapping exists between children of the node and the parent address space.

■ Address Translation Example:

```
soc {
    compatible = "simple-bus";
    #address-cells = <1>;
    #size-cells = <1>;
    ranges = <0x0 0xe0000000 0x00100000>;
    serial@4600 {
        device_type = "serial";
        compatible = "ns16550";
        reg = <0x4600 0x100>;
        clock-frequency = <0>;
        interrupts = <0xA 0x8>;
        interrupt-parent = <&ipic>;
    };
};
```

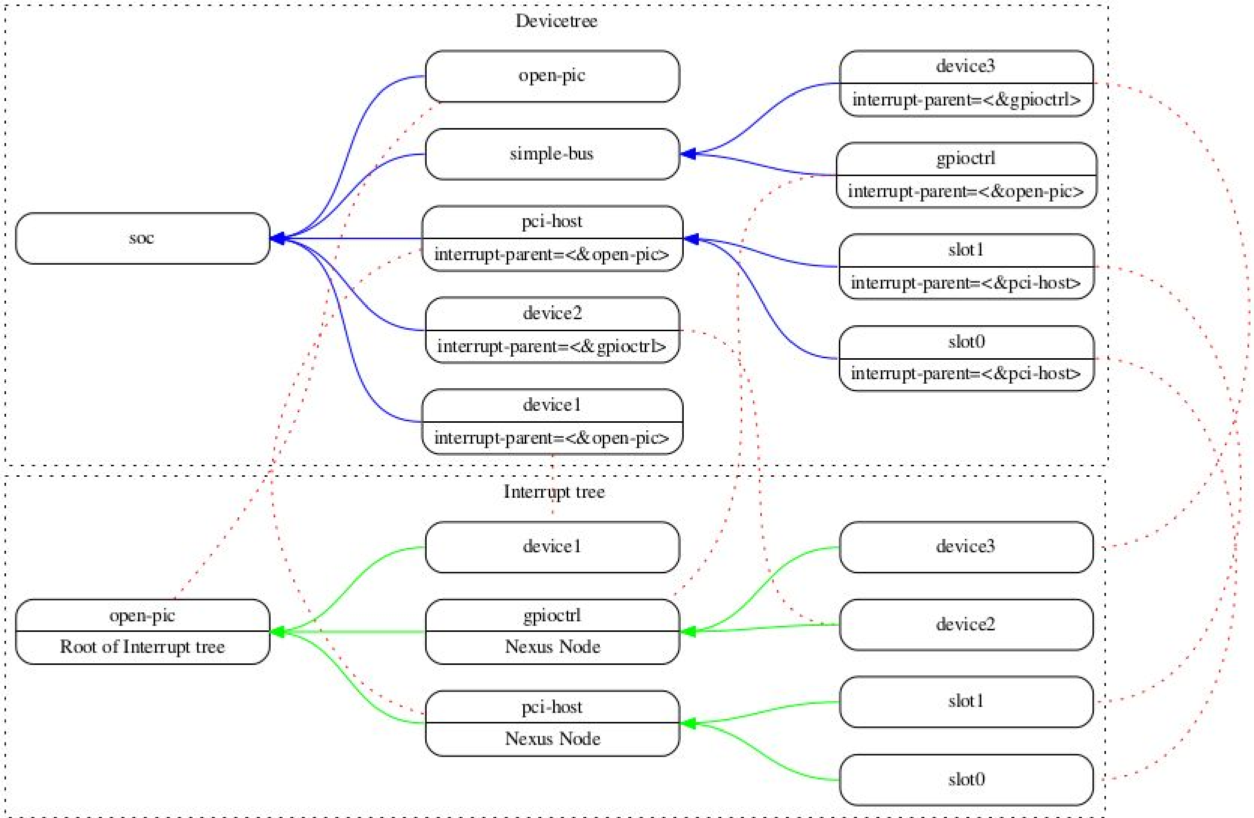
- The SoC node specifies a **ranges** property of **<0x0 0xe0000000 0x00100000>;**
- This property value specifies that for a 1024 KB (i.e 0x00100000) range of address space, a child node addressed at physical 0x0 maps to a parent address of physical **0xe0000000**.
- **With this mapping, the serial device node can be addressed by a load or store at address 0xe0004600, an offset of 0x4600 (specified in reg) plus the 0xe0000000 mapping specified in ranges.**

- **dma-ranges**
  - Property name: **dma-ranges**
  - Value type: **<empty>** or **<prop-encoded-array>** encoded as an arbitrary number of **(child-bus-address, parent-bus-address, length)** triplets.
  - Description:
    - The **dma-ranges** property is used to describe the Direct Memory Access (DMA) structure of a memory-mapped bus whose device-tree parent can be accessed from DMA operations originating from the bus.
    - It provides a means of defining a mapping or translation between the physical address space of the bus and the physical address space of the parent of the bus.
    - The format of the value of the **dma-ranges** property is an arbitrary number of triplets of **(child-bus-address, parent-bus-address, length)**. Each triplet specified describes a contiguous DMA address range.
      - The **child-bus-address** is a physical address within the child bus' address space. The number of cells to represent the address depends on the bus and can be determined from the **#address-cells** of this node (the node in which the **dma-ranges** property appears).

- The **parent-bus-address** is a physical address within the parent bus' address space. The number of cells to represent the parent address is bus dependent and can be determined from the **#address-cells** property of the node that defines the parent's address space.
- The *length* specifies the size of the range in the child's address space. The number of cells to represent the size can be determined from the **#size-cells** of this node (the node in which the **dma-ranges** property appears).

- **Interrupts and Interrupt Mapping**
  - Within the device-tree a logical interrupt tree exists that represents the hierarchy and routing of interrupts in the platform hardware.
  - While generically referred to as an interrupt tree it is more technically a directed acyclic graph.
  - The physical wiring of an interrupt source to an interrupt controller is represented in the device-tree with the **interrupt-parent** property.
  - **Nodes that represent interrupt-generating devices contain an interrupt-parent property which has a phandle value that points to the device to which the device's interrupts are routed, typically an interrupt controller.**
  - **If an interrupt-generating device does not have an interrupt-parent property, its interrupt parent is assumed to be its device-tree parent.**
  - Each interrupt generating device contains an **interrupts** property with a value describing one or more interrupt sources for that device.
  - Each source is represented with information called an **interrupt specifier**. The format and meaning of an **interrupt specifier** is interrupt domain specific, i.e. it is dependent on properties of the node at the root of its interrupt domain.
  - The **#interrupt-cells** property is used by the root of an interrupt domain to define the number of **<u32>** values needed to encode an interrupt specifier. For example, for an Open PIC interrupt controller, an **interrupt-specifier** takes two 32-bit values and consists of an interrupt number and level/sense information for the interrupt.
  - An interrupt domain is the context in which an interrupt specifier is interpreted. The root of the domain is either
    1. **An interrupt controller:** An interrupt controller is a physical device and will need a driver to handle interrupts routed through it. It may also cascade into another interrupt domain. An interrupt controller is specified by the presence of an **interrupt-controller** property on that node in the device-tree.
    2. **An interrupt nexus:** An interrupt nexus defines a translation between one interrupt domain and another. The translation is based on both domain-specific and bus-specific information. This translation between domains is performed with the **interrupt-map** property. For example, a PCI controller device node could be an interrupt nexus that defines a translation from the PCI interrupt namespace (*INTA, INTB, etc.*) to an interrupt controller with Interrupt Request (IRQ) numbers.
  - **The root of the interrupt tree is determined when traversal of the interrupt tree reaches an interrupt controller node without an interrupts property and thus no explicit interrupt parent.**

- See below figure for an example of a graphical representation of a device-tree with interrupt parent relationships shown. It shows both the natural structure of the device-tree as well as where each node sits in the logical interrupt tree.



- In the example shown in above figure:
  - The **open-pic** interrupt controller is the root of the interrupt tree.
  - The interrupt tree root has three children—devices that route their interrupts directly to the open-pic:
    - device1
    - PCI host controller
    - GPIO Controller
  - Three **interrupt domains** exist;
    - one rooted at the open-pic node,
    - one at the PCI host bridge node, and
    - one at the GPIO Controller node.
  - There are two nexus nodes; one at the PCI host bridge and one at the GPIO controller.

- **Properties for Interrupt Generating Devices:**
  - **interrupts**
    - Property name: **interrupts**
    - Value type: **<prop-encoded-array>** encoded as arbitrary number of interrupt specifiers.
    - Description:

  - The `interrupts` property of a device node defines the interrupt or interrupts that are generated by the device.
  - The value of the `interrupts` property consists of an arbitrary number of *interrupt specifiers*. The format of an interrupt specifier is defined by the binding of the interrupt domain root.
  - `interrupts` is overridden by the `interrupts-extended` property and normally only one or the other should be used.
- Example:
  - A common definition of an interrupt specifier in an open PIC–compatible interrupt domain consists of two cells; an interrupt number and level/sense information.
  - See the following example, which defines a single interrupt specifier, with an interrupt number of 0xA and level/sense encoding of 8.
  - `interrupts = <0xA 8>;`

interrupts-parent

- Property name: `interrupts-parent`
- Value type: `<phandle>`
- Description:
  - Because the hierarchy of the nodes in the interrupt tree might not match the device-tree, the `interrupt-parent` property is available to make the definition of an interrupt parent explicit.
  - The value is the phandle to the interrupt parent. If this property is missing from a device, its interrupt parent is assumed to be its device-tree parent.

interrupts-extended

- Property name: `interrupts-extended`
- Value type: `<phandle> <prop-encoded-array>`
- Description:
  - The `interrupts-extended` property lists the interrupt(s) generated by a device. `interrupts-extended` **should be used instead of interrupts when a device is connected to multiple interrupt controllers as it encodes a *parent phandle* with each interrupt specifier.**
- Example:
  - This example shows how a device with two interrupt outputs connected to two separate interrupt controllers would describe the connection using an `interrupts-extended` property.
  - pic is an interrupt controller with an `#interrupt-cells` specifier of 2, while gic is an interrupt controller with an `#interrupts-cells` specifier of 1.
  - `interrupts-extended = <&pic 0xA 8>, <&gic 0xda>;`
- The `interrupts` and `interrupts-extended` properties are mutually exclusive. A device node should use one or the other, but not both. Using both is only permissible when required for compatibility with software that does not understand `interrupts-extended`. If both `interrupts-extended` and `interrupts` are present then `interrupts-extended` takes precedence.

- **Properties for Interrupt Controllers:**
  - **#interrupts-cells**
    - Property name: **#interrupt-cells**
    - Value type: **<u32>**
    - Description:
      - The **#interrupt-cells** property defines the number of cells required to encode an interrupt specifier for an interrupt domain.

  - **interrupt-controller**
    - Property name: **interrupt-controller**
    - Value type: **<empty>**
    - Description:
      - **The presence of an interrupt-controller property defines a node as an interrupt controller node.**

- **Interrupt Nexus properties:**
  - An interrupt nexus node shall have an **#interrupt-cells** property.

  - **interrupt-map**
    - Property name: **#interrupt-map**
    - Value type: **<prop-encoded-array>** encoded as an arbitrary number of interrupt mapping entries.
    - Description:
      - An **interrupt-map** is a property on a nexus node that bridges one interrupt domain with a set of parent interrupt domains and specifies how interrupt specifiers in the child domain are mapped to their respective parent domains.
      - The interrupt map is a table where each row is a mapping entry consisting of five components:
        ✓ **child unit address,**
        ✓ **child interrupt specifier,**
        ✓ **interrupt-parent,**
        ✓ **parent unit address,**
        ✓ **parent interrupt specifier.**
      - **child unit address:** The unit address of the child node being mapped. The number of 32-bit cells required to specify this is described by the **#address-cells** property of the bus node on which the child is located.
      - **child interrupt specifier:** The interrupt specifier of the child node being mapped. The number of 32-bit cells required to specify this component is described by the **#interrupt-cells** property of this node - the nexus node containing the *interrupt-map* property.
      - **interrupt-parent:** A single **<phandle>** value that points to the interrupt parent to which the child domain is being mapped.
      - **parent unit address:** The unit address in the domain of the interrupt parent. The number of 32-bit cells required to specify this address is described by the **#address-cells** property of the node pointed to by the interrupt-parent field.

- ○ `parent interrupt specifier:` The interrupt specifier in the parent domain. The number of 32-bit cells required to specify this component is described by the `#interrupt-cells` property of the node pointed to by the `interrupt-parent` field.
- ○ Lookups are performed on the interrupt mapping table by matching a unit-address/interrupt specifier pair against the child components in the `interrupt-map`. Because some fields in the unit interrupt specifier may not be relevant, a mask is applied before the lookup is done. This mask is defined in the `interrupt-map-mask` property.
- ○ `Note:` Both the child node and the interrupt parent node are required to have `#address-cells` and `#interrupt-cells` properties defined. If a unit address component is not required, `#address-cells` shall be explicitly defined to be zero.

interrupt-map-mask

- Property name: `interrupt-map-mask`
- Value type: `<prop-encoded-array>` encoded as a bit mask
- Description:
  - An `interrupt-map-mask` property is specified for a nexus node in the interrupt tree.
  - This property specifies a mask that is applied to the incoming unit interrupt specifier being looked up in the table specified in the `interrupt-map` property.

#interrupt-cells

- Property name: `#interrupt-cells`
- Value type: `<u32>`
- Description:
  - The `#interrupt-cells` property defines the number of cells required to encode an interrupt specifier for an interrupt domain.

Interrupt Mapping Example

- The following shows the representation of a fragment of a device-tree with a PCI bus controller and a sample interrupt map for describing the interrupt routing for two PCI slots (`IDSEL 0x11,0x12`).
- The `INTA, INTB, INTC,` and `INTD` pins for slots 1 and 2 are wired to the Open PIC interrupt controller.

```
soc {
    compatible = "simple-bus";
    #address-cells = <1>;
    #size-cells = <1>;

    open-pic {
        clock-frequency = <0>;
        interrupt-controller;
        #address-cells = <0>;
        #interrupt-cells = <2>;
    };
```

```
  pci {
        #interrupt-cells = <1>;
        #size-cells = <2>;
        #address-cells = <3>;
        interrupt-map-mask = <0xf800 0 0 7 >
        interrupt-map = <
            /* IDSEL 0x11 - PCI slot 1 */
            0x8800 0 0 1 &open-pic 2 1 /* INTA  */
            0x8800 0 0 2 &open-pic 3 1 /* INTB  */
            0x8800 0 0 3 &open-pic 4 1 /* INTC  */
            0x8800 0 0 4 &open-pic 1 1 /* INTD  */
            /* IDSEL 0x12 - PCI slot 2 */
            0x9000 0 0 1 &open-pic 3 1 /* INTA  */
            0x9000 0 0 2 &open-pic 4 1 /* INTB  */
            0x9000 0 0 3 &open-pic 1 1 /* INTC  */
            0x9000 0 0 4 &open-pic 2 1 /* INTD  */
        >;
    };
};
```

- One Open PIC interrupt controller is represented and is identified as an interrupt controller with an *interrupt-controller* property.
- Each row in the *interrupt-map* table consists of five parts: *a child unit address* and *interrupt specifier*, which is mapped to an *interrupt-parent* node with a specified *parent unit address* and *interrupt specifier*.
- For example, the first row of the *interrupt-map* table specifies the mapping for INTA of slot 1. The components of that row are shown here

  - child unit address: 0x8800 0 0
  - child interrupt specifier: 1
  - interrupt parent: &open-pic
  - parent unit address: (empty because #address-cells = <0> in the open-pic node)
  - parent interrupt specifier: 2 1

  - The *child unit address* is <0x8800 0 0>. This value is encoded with three 32-bit cells, which is determined by the value of the *#address-cells* property (*value of 3*) of the PCI controller. The three cells represent the PCI address as described by the binding for the PCI bus. [The encoding includes the bus number (0x0 << 16), device number (0x11 << 11), and function number (0x0 << 8)]
  - The *child interrupt specifier* is <1>, which specifies INTA as described by the PCI binding. This takes one 32-bit cell as specified by the *#interrupt-cells* property (*value of 1*) of the PCI controller, which is the child interrupt domain.
  - The *interrupt parent* is specified by a *phandle* which points to the *interrupt parent* of the slot, the Open PIC interrupt controller.
  - The parent has no unit address because the parent interrupt domain (*the open-pic node*) has an *#address-cells* value of <0>.
  - The *parent interrupt specifier* is <2 1>. The number of cells to represent the interrupt specifier (*two cells*) is determined by the *#interrupt-cells* property on the *interrupt parent*, the *open-pic* node. [The value <2 1> is a value specified by the device binding for the Open PIC interrupt controller. The value <2> specifies the physical interrupt source number on the interrupt controller to which INTA is wired. The value <1> specifies the level/sense encoding.]

- In this example, the *interrupt-map-mask* property has a value of <0xf800 0 0 7>. This mask is applied to a *child unit interrupt specifier* before performing a lookup in the *interrupt-map* table.
- To perform a lookup of the open-pic interrupt source number for INTB for IDSEL 0x12 (slot 2), function 0x3, the following steps would be performed:
  - The child unit address and interrupt specifier form the value <0x9300 0 0 2>. [The encoding of the address includes the bus number (0x0 << 16), device number (0x12 << 11), and function number (0x3 << 8). The *interrupt specifier* is 2, which is the encoding for INTB as per the PCI binding.]
  - The *interrupt-map-mask* value <0xf800 0 0 7> is applied, giving a result of <0x9000 0 0 2>.
  - That result is looked up in the *interrupt-map* table, which maps to the *parent interrupt specifier* <4 1>.

- **Nexus Nodes and Specifier Mapping**
  - **Nexus Node Properties**
    - A nexus node shall have a *#<specifier>-cells* property, where <specifier> is some specifier space such as 'gpio', 'clock', 'reset', etc.
    - **<specifier>-map**
      - Property name: *<specifier>-map*
      - Value type: *<prop-encoded-array>* encoded as an arbitrary number of specifier mapping entries.
      - Description:
        - A *<specifier>-map* is a property in a nexus node that bridges one specifier domain with a set of parent specifier domains and describes how specifiers in the child domain are mapped to their respective parent domains.
        - The map is a table where each row is a mapping entry consisting of three components:
          ✓ child specifier,
          ✓ specifier parent, and
          ✓ parent specifier
        - **child specifier:** The specifier of the child node being mapped. The number of 32-bit cells required to specify this component is described by the *#<specifier>-cells* property of this node - the nexus node containing the *<specifier>-map* property.
        - **specifier parent:** A single *<phandle>* value that points to the specifier parent to which the child domain is being mapped.
        - **parent specifier:** The specifier in the parent domain. The number of 32-bit cells required to specify this component is described by the *#<specifier>-cells* property of the specifier parent node.
        - Lookups are performed on the mapping table by matching a specifier against the child specifier in the map. Because some fields in the specifier may not be relevant or need to be modified, a mask is applied before the lookup is done. This mask is defined in the *<specifier>-map-mask property*.
        - Similarly, when the specifier is mapped, some fields in the unit specifier may need to be kept unmodified and passed through from the child node to the parent node. In this case, a *<specifier>-map-pass-thru* property

may be specified to apply a mask to the child specifier and copy any bits that match to the *parent unit specifier*.

- **<specifier>-map-mask**
  - Property name: *<specifier>-map-mask*
  - Value type: *<prop-encoded-array>* encoded as a bit mask
  - Description:
    - A *<specifier>-map-mask* property may be specified for a nexus node. This property specifies a mask that is applied to the child unit specifier being looked up in the table specified in the *<specifier>-map* property.
    - If this property is not specified, the mask is assumed to be a mask with all bits set.

- **<specifier>-map-pass-thru**
  - Property name: *<specifier>-map-pass-thru*
  - Value type: *<prop-encoded-array>* encoded as a bit mask
  - Description:
    - A *<specifier>-map-pass-thru* property may be specified for a nexus node. This property specifies a mask that is applied to the child unit specifier being looked up in the table specified in the *<specifier>-map* property.
    - Any matching bits in the child unit specifier are copied over to the parent specifier. If this property is not specified, the mask is assumed to be a mask with no bits set.

- **#<specifier>-cells**
  - Property name: *#<specifier>-cells*
  - Value type: *<u32>*
  - Description:
    - The *#<specifier>-cells* property defines the number of cells required to encode a specifier for a domain.

- **Specifier Mapping Example**
  - The following shows the representation of a fragment of a device-tree with two GPIO controllers and a sample specifier map for describing the GPIO routing of a few gpios on both of the controllers through a connector on a board to a device.
  - The expansion device node is one one side of the connector node and the SoC with the two GPIO controllers is on the other side of the connector.

```
soc {
   soc_gpio1: gpio-controller1 {
   #gpio-cells = <2>;
   };

   soc_gpio2: gpio-controller2 {
   #gpio-cells = <2>;
   };
};
```

```
connector: connector {
    #gpio-cells = <2>;
    gpio-map = <0 0 &soc_gpio1 1 0>,
               <1 0 &soc_gpio2 4 0>,
               <2 0 &soc_gpio1 3 0>,
               <3 0 &soc_gpio2 2 0>;
    gpio-map-mask = <0xf 0x0>;
    gpio-map-pass-thru = <0x0 0x1>;
};

expansion_device {
    reset-gpios = <&connector 2 GPIO_ACTIVE_LOW>;
};
```

- Each row in the gpio-map table consists of three parts: a child unit specifier, which is mapped to a gpio-controller node with a parent specifier.
- For example, the first row of the specifier-map table specifies the mapping for GPIO 0 of the connector. The components of that row are shown here

```
child specifier: 0 0
specifier parent: &soc_gpio1
parent specifier: 1 0
```

  - The *child specifier* is <0 0>, which specifies GPIO 0 in the connector with a flags field of 0. This takes two 32-bit cells as specified by the *#gpio-cells* property of the connector node, which is the *child specifier domain*.
  - The *specifier parent* is specified by a phandle which points to the *specifier parent* of the connector, the first GPIO controller in the SoC.
  - The *parent specifier* is <1 0>. The number of cells to represent the gpio specifier (*two cells*) is determined by the *#gpio-cells* property on the *specifier parent*, the soc_gpio1 node. [The value <1 0> is a value specified by the device binding for the GPIO controller. The value <1> specifies the GPIO pin number on the GPIO controller to which GPIO 0 on the connector is wired. The value <0> specifies the flags (active low, active high, etc.).]
- In this example, the *gpio-map-mask* property has a value of <0xf 0>. This mask is applied to a *child unit specifier* before performing a lookup in the *gpio-map* table. Similarly, the *gpio-map-pass-thru* property has a value of <0x0 0x1>.
- This mask is applied to a *child unit specifier* when mapping it to the *parent unit specifier*. Any bits set in this mask are cleared out of the *parent unit specifier* and copied over from the *child unit specifier* to the *parent unit specifier*.
- To perform a lookup of the connector's specifier source number for GPIO 2 from the expansion device's *reset-gpios* property, the following steps would be performed:
  - The child specifier forms the value <2 GPIO_ACTIVE_LOW>. The specifier is encoding GPIO 2 with active low flags per the GPIO binding.
  - The *gpio-map-mask* value <0xf 0x0> is ANDed with the child specifier, giving a result of <0x2 0>.
  - The result is looked up in the *gpio-map* table, which maps to the *parent specifier* <3 0> and *&soc_gpio1* phandle.

- The *gpio-map-pass-thru* value <0x0 0x1> is inverted and ANDed with the parent specifier found in the *gpio-map* table, resulting in <3 0>.
- The child specifier is ANDed with the *gpio-map-pass-thru* mask, forming <0 GPIO_ACTIVE_LOW> which is then ORed with the cleared *parent specifier* <3 0> resulting in <3 GPIO_ACTIVE_LOW>.
- The specifier <3 GPIO_ACTIVE_LOW> is appended to the mapped phandle *&soc_gpio1* resulting in <&soc_gpio1 3 GPIO_ACTIVE_LOW>.

- **Device Node Requirements**
  - **Base Device Node Types**
    - The sections that follow specify the requirements for the base set of device nodes required in a DTSpec-compliant device-tree.
    - All device-trees shall have a root node and the following nodes shall be present at the root of all device-trees:
      - One /cpus node
      - At least one /memory node

  - **Root Node**
    - The device-tree has a single root node of which all other device nodes are descendants. The full path to the root node is /.
    - Below are required properties in root node:
      - #address-cells
      - #size-sells
      - model
      - compatible

    NOTE: All other standard properties are allowed but are optional.

  - **/aliases Node**
    - A device-tree may have an aliases node (*/aliases*) that defines one or more alias properties.
    - The alias node shall be at the root of the device-tree and have the node name /aliases.
    - Each property of the /aliases node defines an alias. The property name specifies the alias name. The property value specifies the full path to a node in the device-tree.
    - For example, the property serial0 = "/simple-bus@fe000000/serial@llc500" defines the alias serial0.
    - Alias names shall be a lowercase text string of 1 to 31 characters from the following set of characters.

Character	**Description**
0-9	Digit
a-z	Lowercase letters
-	Comma

    - An alias value is a device path and is encoded as a string. The value represents the full path to a node, but the path does not need to refer to a leaf node.

- A client program may use an alias property name to refer to a full device path as all or part of its string value. A client program, when considering a string as a device path, shall detect and use the alias.
- Example:

```
aliases {
    serial0 = "/simple-bus@fe000000/serial@llc500";
    ethernet0 = "/simple-bus@fe000000/ethernet@31c000";
};
```

- Given the alias serial0, a client program can look at the /aliases node and determine the alias refers to the device path /simple-bus@fe000000/serial@llc500.

- **/memory Node**
  - A memory device node is required for all device-trees and describes the physical memory layout for the system.
  - If a system has multiple ranges of memory, multiple memory nodes can be created, or the ranges can be specified in the reg property of a single memory node.
  - The unit-name component of the node name shall be *memory*.
  - The client program may access memory not covered by any memory reservations using any storage attributes it chooses. However, before changing the storage attributes used to access a real page, the client program is responsible for performing actions required by the architecture and implementation, possibly including flushing the real page from the caches.
  - The boot program is responsible for ensuring that, without taking any action associated with a change in storage attributes, the client program can safely access all memory (*including memory covered by memory reservations*) as WIMG = 0b001x. That is:
    - not Write Through Required
    - not Caching Inhibited
    - Memory Coherence
    - Required either not Guarded or Guarded,

    If the VLE storage attribute is supported, with VLE=0.

Property Name	**Usage**	**Value Type**	**Definition**
device_type	Required	<string>	Value shall be "Memory"
reg	Required	<prop-encoded-array>	Consists of an arbitrary number of address and size pairs that specify the physical address and size of the memory ranges.
initial-mapped-area	optional	<prop-encoded-array>	Specifies the address and size of the Initial Mapped Area is a prop-encoded-array consisting of a triplet of (effective address, physical address, size). The effective and physical address shall each be 64-bit (<u64> value), and the size shall be 32-bits

			(<u32> value).

NOTE: All other standard properties are allowed but are optional.

- Examples:
  - Given a 64-bit Power system with the following physical memory layout:
    - RAM: starting address 0x0, length 0x80000000 (2 GB)
    - RAM: starting address 0x100000000, length 0x100000000 (4 GB)
  - Memory nodes could be defined as follows, assuming *#address-cells* = <2> and *#size-cells* = <2>.

```
/* Example #1 */

memory@0 {
   device_type = "memory";
   reg = <0x000000000 0x00000000 0x00000000 0x80000000
            0x000000001 0x00000000 0x00000001 0x00000000>;
};
```

```
/* Example #2 */

memory@0 {
   device_type = "memory";
   reg = <0x000000000 0x00000000 0x00000000 0x80000000>;
};

memory@100000000 {
   device_type = "memory";
   reg = <0x000000001 0x00000000 0x00000001 0x00000000>;
};
```

  - The *reg* property is used to define the address and size of the two memory ranges. The 2 GB I/O region is skipped. Note that the *#address-cells* and *#size-cells* properties of the root node specify a value of 2, which means that two 32-bit cells are required to define the address and length for the *reg* property of the memory node.

- **chosen Node**
  - The /chosen node does not represent a real device in the system but describes parameters chosen or specified by the system firmware at run time. It shall be a child of the root node.

Property Name	Usage	Value Type	Definition
bootargs	Optional	<string>	A string that specifies the boot arguments for the client program. The value could potentially be a null string if no boot arguments are required.
stdout-path	Optional	<string>	A string that specifies the full path to the node representing the device to be used for boot console output. If the character ":" is present in the value it terminates the path. The value may be an alias. If the stdin-path property is not specified, stdout-path should be assumed to define the input device.

stdin-path	Optional	<string>	A string that specifies the full path to the node representing the device to be used for boot console input. If the character ":" is present in the value it terminates the path. The value may be an alias.

NOTE: All other standard properties are allowed but are optional.

- Examples:

```
chosen {
   bootargs = "root=/dev/nfs rw nfsroot=192.168.1.1 console=ttyS0,115200";
};
```

- **/cpus Node Properties**
  - A /cpus node is required for all device-trees.
  - It does not represent a real device in the system, but acts as a container for child cpu nodes which represent the system's CPUs.
  - Below are required properties in root node:
    - #address-cells
    - #size-sells

    NOTE: All other standard properties are allowed but are optional.
  - The /cpus node may contain properties that are common across cpu nodes.

- **/cpus/cpu* Node Properties**
  - A cpu node represents a hardware execution block that is sufficiently independent that it is capable of running an operating system without interfering with other CPUs possibly running other operating systems.
  - Hardware threads that share an MMU would generally be represented under one cpu node. If other more complex CPU topographies are designed, the binding for the CPU must describe the topography (*e.g. threads that don't share an MMU*).
  - CPUs and threads are numbered through a unified number-space that should match as closely as possible the interrupt controller's numbering of CPUs/threads.
  - Properties that have identical values across cpu nodes may be placed in the /cpus node instead. A client program must first examine a specific cpu node, but if an expected property is not found then it should look at the parent /cpus node.
  - This results in a less verbose representation of properties which are identical across all CPUs.
  - The node name for every CPU node should be *cpu*.
  - **General Properties of /cpus/cpu* nodes**
    - The following table describes the general properties of cpu nodes. Some of the properties described in the table are select standard properties with specific applicable detail.
      - **device_type**
        - Property name: *device_type*
        - Value type: *<string>*
        - Usage: Required
        - Description:
          - Value shall be "cpu".

- **reg**
  - Property name: *reg*
  - Value type: *array*
  - Usage: Required
  - Description:
    - The value of reg is a *<prop-encoded-array>* that defines a unique CPU/thread id for the CPU/threads represented by the CPU node.
    - If a CPU supports more than one thread (*i.e. multiple streams of execution*) the reg property is an array with 1 element per thread.
    - The *#address-cells* on the /cpus node specifies how many cells each element of the array takes. Software can determine the number of threads by dividing the size of reg by the parent node's *#address-cells*.
    - If a CPU/thread can be the target of an external interrupt the reg property value must be a unique CPU/thread id that is addressable by the interrupt controller.
    - If a CPU/thread cannot be the target of an external interrupt, then reg must be unique and out of bounds of the range addressed by the interrupt controller If a CPU/thread's PIR (*pending interrupt register*) is modifiable, a client program should modify PIR to match the reg property value.
    - If PIR cannot be modified and the PIR value is distinct from the interrupt controller number space, the CPUs binding may define a binding-specific representation of PIR values if desired.

- **clock-frequency**
  - Property name: *clock-frequency*
  - Value type: *array*
  - Usage: Required
  - Description:
    - Specifies the current clock speed of the CPU in Hertz.
    - The value is a <prop-encoded-array> in one of two forms:
      - A 32-bit integer consisting of one <u32> specifying the frequency.
      - A 64-bit integer represented as a <u64> specifying the frequency.

- **timebase-frequency**
  - Property name: *timebase-frequency*
  - Value type: *array*
  - Usage: Required
  - Description:
    - Specifies the current frequency at which the timebase and decrementer registers are updated (in Hertz).
    - The value is a <prop-encoded-array> in one of two forms:
      - A 32-bit integer consisting of one <u32> specifying the frequency.
      - A 64-bit integer represented as a <u64>.

- **status**
  - Property name: *status*

- Value type: *<string>*
- Description:
  - A standard property describing the state of a CPU. This property shall be present for nodes representing CPUs in a symmetric multiprocessing (SMP) configuration.
  - For a CPU node the meaning of the "okay" and "disabled" values are as follows:
    - "okay" : The CPU is running.
    - "disabled" : The CPU is in a quiescent state
  - A quiescent CPU is in a state where it cannot interfere with the normal operation of other CPUs, nor can its state be affected by the normal operation of other running CPUs, except by an explicit method for enabling or re-enabling the quiescent CPU(see the *enable-method* property).
  - In particular, a running CPU shall be able to issue broadcast TLB invalidates without affecting a quiescent CPU.
  - Examples:
    - A quiescent CPU could be in a spin loop, held in reset, and electrically isolated from the system bus or in another implementation dependent state.

- **enable-method**
  - Property name: *enable-method*
  - Value type: *<stringlist>*
  - Description:
    - Describes the method by which a CPU in a disabled state is enabled. This property is required for CPUs with a status property with a value of "disabled".
    - The value consists of one or more strings that define the method to release this CPU. If a client program recognizes any of the methods, it may use it. The value shall be one of the following:
      - **"spin-table"** : The CPU is enabled with the spin table method defined in the DTSpec.
      - **"[vendor],[method]"** : Implementation dependent string that describes the method by which a CPU is released from a "disabled" state.
        - The required format is: "[vendor],[method]", where vendor is a string describing the name of the manufacturer and method is a string describing the vendor specific mechanism.
        - Example: "fsl,MPC8572DS"

- **cpu-release-addr**
  - Property name: *cpu-release-addr*
  - Value type: *<u64>*
  - Description:
    - The cpu-release-addr property is required for cpu nodes that have an enable-method property value of "spin-table".

    ○ The value specifies the physical address of a spin table entry that releases a secondary CPU from its spin loop.

NOTE: Not covered below topics:
- "/cpus/cpu* Node Power ISA Properties".
- TLB Properties
- Internal (L1) Cache Properties
- Multi-level and Shared Cache Nodes (/cpus/cpu*/l?-cache)

Refer Device-tree specification for detailed information.

- **Device Tree binding**
- When creating a new device tree representation for a device, a binding should be created that fully describes the required properties and value of the device. This set of properties shall be sufficiently descriptive to provide device drivers with needed attributes of the device.
- The **compatible** property of a device node describes the specific binding (or bindings) to which the node complies.
- **Documentation of Device Tree bindings**
  - All Device Tree bindings recognized by the kernel are documented in Documentation/devicetree/bindings.
  - Each binding documentation described which properties are accepted, with which values, which properties are mandatory vs. optional, etc.
  - All new Device Tree bindings must be reviewed by the Device Tree maintainers, by being posted to *devicetree@vger.kernel.org*. This ensures correctness and consistency across bindings.
  - A Device Tree binding documentation example:

```
/* Documentation/devicetree/bindings/tty/serial/fsl-mxs-auart.txt */

* Freescale MXS Application UART (AUART)

Required properties for all SoCs:
- compatible : Should be one of fallowing variants:
	"fsl,imx23-auart" - Freescale i.MX23
	"fsl,imx28-auart" - Freescale i.MX28
	"alphascale,asm9260-auart" - Alphascale ASM9260
- reg : Address and length of the register set for the device
- interrupts : Should contain the auart interrupt numbers
- dmas: DMA specifier, consisting of a phandle to DMA controller node
  and AUART DMA channel ID.
  Refer to dma.txt and fsl-mxs-dma.txt for details.
- dma-names: "rx" for RX channel, "tx" for TX channel.

Required properties for "alphascale,asm9260-auart":
- clocks : the clocks feeding the watchdog timer. See clock-bindings.txt
- clock-names : should be set to
	"mod" - source for tick counter.
	"ahb" - ahb gate.

Optional properties:
```

```
- uart-has-rtscts : Indicate the UART has RTS and CTS lines
  for hardware flow control,
          it also means you enable the DMA support for this UART.
- {rts,cts,dtr,dsr,rng,dcd}-gpios: specify a GPIO for RTS/CTS/DTR/DSR/RI/DCD
  line respectively. It will use specified PIO instead of the peripheral
  function pin for the USART feature.
  If unsure, don't specify this property.

Example:
auart0: serial@8006a000 {
          compatible = "fsl,imx28-auart", "fsl,imx23-auart";
          reg = <0x8006a000 0x2000>;
          interrupts = <112>;
          dmas = <&dma_apbx 8>, <&dma_apbx 9>;
          dma-names = "rx", "tx";
          cts-gpios = <&gpio1 15 GPIO_ACTIVE_LOW>;
          dsr-gpios = <&gpio1 16 GPIO_ACTIVE_LOW>;
          dcd-gpios = <&gpio1 17 GPIO_ACTIVE_LOW>;
};

Note: Each auart port should have an alias correctly numbered in "aliases"
node.

Example:

aliases {
          serial0 = &auart0;
          serial1 = &auart1;
          serial2 = &auart2;
          serial3 = &auart3;
          serial4 = &auart4;
};
```

- **Device Tree Source(DTS) Format (Version 1)**
  - The Devicetree Source (DTS) format is a textual representation of a device-tree in a form that can be processed by dtc into a binary device-tree in the form expected by the kernel. The following description is not a formal syntax definition of DTS, but describes the basic constructs used to represent device-trees.
  - The name of DTS files should end with ".dts".

  - **Compiler directives**
    - Other source files can be included from a DTS file. The name of the included files should end with ".dtsi". Included files can in turn include additional files.

```
/include/ "FILE"
```

  - **Labels**
    - The source format allows labels to be attached to any node or property value in the device-tree. *Phandle* and path references can be automatically generated by referencing a label instead of explicitly specifying a phandle value or the full path to a node.
    - Labels are only used in the device-tree source format and are not encoded into the DTB binary.

- A label shall be between 1 to 31 characters in length, be composed only of the characters in the set Table below, and must not start with a number.
- Labels are created by appending a colon (':') to the label name. References are created by prefixing the label name with an ampersand ('&').

Character	**Description**
0-9	Digit
a-z	Lowercase letters
A-Z	Uppercase letters
_	Comma

- **Node and property definitions**
  - Device-tree nodes are defined with a node name and unit address with braces marking the start and end of the node definition. They may be preceded by a label.

```
[label:] node-name[@unit-address] {
    [properties definitions]
    [child nodes]
};
```

  - Nodes may contain property definitions and/or child node definitions. If both are present, properties shall come before child nodes.

  - Previously defined nodes may be deleted.

```
/delete-node/ node-name;
/delete-node/ &label;
```

  - Property definitions are name value pairs in the form:

```
[label:] property-name = value;
```

  - except for properties with empty (zero length) value which have the form:

```
[label:] property-name;
```

  - Previously defined properties may be deleted.

```
/delete-property/ property-name;
```

  - Property values may be defined as an array of 32-bit integer cells, as null-terminated strings, as bytestrings or a combination of these.
    - Arrays of cells are represented by angle brackets surrounding a space separated list of C-style integers. Example:

```
interrupts = <17 0xc>;
```

- values may be represented as arithmetic, bitwise, or logical expressions within parenthesis.

Arithmetic Operators		Bitwise Operators		Logical Operators		Relational Operators	
+	Add	&	AND	&&	AND	<	Less than
-	Subtract	|	OR	||	OR	>	Greater than
*	Multiply	^	exclusive OR	!	NOT	<=	Less than or equal to
/	Divide	~	NOT			>=	Greater than or equal to
%	Modulo	<<	Left shift			==	Equal to
		>>	Right shift			!=	Not Equal to

Ternary operators

? : (condition ? value_if_true : value_if_false)

- A 64-bit value is represented with two 32-bit cells. Example:

```
clock-frequency = <0x00000001 0x00000000>;
```

- A null-terminated string value is represented using double quotes (*the property value is considered to include the terminating NULL character*). Example:

```
compatible = "simple-bus";
```

- A bytestring is enclosed in square brackets [ ] with each byte represented by two hexadecimal digits. Spaces between each byte are optional. Example:

```
local-mac-address = [00 00 12 34 56 78];
```

or equivalently:

```
local-mac-address = [000012345678];
```

- Values may have several comma-separated components, which are concatenated together. Example:

```
compatible = "ns16550", "ns8250";
example = <0xf00f0000 19>, "a strange property format";
```

- In a cell array a reference to another node will be expanded to that node's phandle. References may be & followed by a node's label. Example:

```
interrupt-parent = < &mpic >;
```

- or they may be & followed by a node's full path in braces. Example:

```
interrupt-parent = < &{/soc/interrupt-controller@40000} >;
```

- Outside a cell array, a reference to another node will be expanded to that node's full path. Example:

```
ethernet0 = &EMAC0;
```

- Labels may also appear before or after any component of a property value, or between cells of a cell array, or between bytes of a bytestring. Examples:

```
reg = reglabel: <0 sizelabel: 0x1000000>;
prop = [ab cd ef byte4: 00 ff fe];
str = start: "string value" end: ;
```

- **File layout**
  - Version 1 DTS files have the overall layout:

```
/dts-v1/;
[memory reservations]
  / {
        [property definitions]
        [child nodes]
    };
```

  - The /dts-v1/; shall be present to identify the file as a version 1 DTS (*dts files without this tag will be treated by dtc as being in the obsolete version 0, which uses a different format for integers in addition to other small but incompatible changes*).
  - Memory reservations define an entry for the device-tree blob's memory reservation table. They have the form:
    - e.g., /memreserve/ <address> <length>; Where, <address> and <length> are 64-bit C-style integers.
  - The / { }; section defines the root node of the devicetree.
  - C style (/* . . . */) and C++ style (//) comments are supported.

- **Getting resources from DTS**
  - Once kernel is booted , it exposes all the parsed device-tree configurations in /proc/device-tree as follows:

```
debian@beaglebone:~$ ls /proc/device-tree
#address-cells chosen compatible interrupt-parent  model  opp-table #size-cells
__symbols__ aliases clk_mcasp0 cpus  leds  name   pmu  soc bone_capemgr
clk_mcasp0_fixed  fixedregulator0  memory@80000000   ocp   serial-number  sound
debian@beaglebone:~$ cat /proc/device-tree/chosen/bootargs
console=ttyO0,115200n8 bone_capemgr.uboot_capemgr_enabled=1 root=/dev/mmcblk0p1 ro
rootfstype=ext4 rootwait coherent_pool=1M net.ifnames=0 rng_core.default_quality=100
quiet
debian@beaglebone:~$
debian@beaglebone:~$ ls /sys/firmware/devicetree/base/
#address-cells chosen compatible interrupt-parent  model  opp-table #size-cells
__symbols__ aliases clk_mcasp0 cpus  leds  name   pmu  soc bone_capemgr
clk_mcasp0_fixed  fixedregulator0  memory@80000000   ocp   serial-number  sound
debian@beaglebone:~$ cat  /sys/firmware/devicetree/base/chosen/bootargs
```

```
console=ttyO0,115200n8 bone_capemgr.uboot_capemgr_enabled=1 root=/dev/mmcblk0p1 ro
rootfstype=ext4 rootwait coherent_pool=1M net.ifnames=0 rng_core.default_quality=100
quiet
debian@beaglebone:~$
```

- Below are the few major APIs for reading the various properties from DTS.
  - **`of_address_to_resource`**: Reads the memory address of the device defined by res property / Translate device tree address and returns as resource.

  - **`irq_of_parse_and_map`**: Attach the interrupt handler, provided by the properties interrupt and interrupt-parent

  - **`of_find_property(np, propname, NULL)`**: To find if property named in argument2 is present or not.

  - **`of_property_read_bool`**: To read a bool property named in argument 2, as it is a bool property it just like searching if that property is present or not. Returns true or false

  - **`of_get_property`**: For reading any property named in argument 2

  - **`of_property_read_u32`**: To read a 32 bit property, populate into 3rd argument. Doesn't set anything to the 3rd argument in case of error.

  - **`of_property_read_string`**: To read string property

  - **`of_match_device`**: Sanity check for device that is matching with the node, highly optional, I don't see much use of it.

```
NOTE: Explore more APIs at <kernel-src>/include/linux/of*.h
```

• Compatibility mode for DT booting

- Some bootloaders have no specific support for the Device-Tree, or the version used on a particular device is too old to have this support.
- To ease the transition, a compatibility mechanism was added: `CONFIG_ARM_APPENDED_DTB`
  - It tells the kernel to look for a DTB right after the kernel image.
  - There is no built-in Makefile rule to produce such kernel, so one must manually do:

```
$ cat arch/arm/boot/zImage arch/arm/boot/dts/myboard.dtb > my-zImage
$ mkimage ... -d my-zImage my-uImage
```

- In addition, the additional option `CONFIG_ARM_ATAG_DTB_COMPAT` tells the kernel to read the ATAGS information from the bootloader, and update the DT using them.

Chapter-4 Wait-Queue in Linux

- When you write a Linux Driver or Module or Kernel Program, Some processes should wait or sleep for some event. There are several ways of handling sleeping and waking up in Linux, each suited to different needs. `Waitqueue` also one of the methods to handle that case.
- Whenever a process must wait for an event (*such as the arrival of data or the termination of a process*), it should go to sleep.
- Sleeping causes the process to suspend execution, freeing the processor for other uses. After some time, the process will be woken up and will continue with its job when the event which we are waiting for has arrived.
- `Waitqueue` is a mechanism provided in the kernel to implement the wait. As the name itself suggests, wait-queue is the list of processes waiting for an event. In other words, A wait-queue is used to wait for someone to wake you up when a certain condition is true. They must be used carefully to ensure there is no race condition.
- There are 3 important steps in wait-queue.
  1. Initializing Waitqueue
  2. Queuing (*Put the Task to sleep until the event comes*)
  3. Waking Up Queued Task

- **Initializing Waitqueue**
  - Use `<linux/wait.h>` header file for Waitqueue. There are two ways to initialize waitqueue.
    1. Static method
    2. Dynamic method , You can use any one of the methods.

  - **Static method**

```
DECLARE_WAIT_QUEUE_HEAD(wq);
```

  where,
  - the `wq` is the name of the queue on which task will be put to sleep.

  - **Dynamic method**

```
wait_queue_head_t wq;
init_waitqueue_head (&wq);
```

- **Queuing**

- Once the wait-queue is declared and initialized, a process may use it to go to sleep. There are several macros available for different uses. We will see one by one.
  1. `wait_event`
  2. `wait_event_timeout`
  3. `wait_event_cmd`
  4. `wait_event_interruptible`
  5. `wait_event_interruptible_timeout`
  6. `wait_event_killable`
- Old kernel versions used the functions `sleep_on()` and `interruptible_sleep_on()`, but those two functions can introduce bad race conditions and should not be used.
- Whenever we use the above one of the macro, it will add that task to the `waitqueue`, which is created by us. Then it will wait for the event.

1. wait_event

- sleep until a condition gets true.

```
wait_event(wq, condition);
```

where,

- `wq` – the waitqueue to wait on
- `condition` – a C expression for the event to wait for

- The process is put to sleep (TASK_UNINTERRUPTIBLE) until the `condition` evaluates to true. The `condition` is checked each time the waitqueue `wq` is woken up.

2. wait_event_timeout

- sleep until a condition gets true or a timeout elapses

```
wait_event_timeout(wq, condition, timeout);
```

where,

- `wq` – the waitqueue to wait on
- `condition` – a C expression for the event to wait for
- `timeout` – timeout, in jiffies

- The process is put to sleep (TASK_UNINTERRUPTIBLE) until the `condition` evaluates to true or timeout elapses. The `condition` is checked each time the wait-queue `wq` is woken up.
- It returns
  - `0`, if the `condition` evaluated to FALSE after the `timeout` elapsed
  - `1`, if the `condition` evaluated to TRUE after the `timeout` elapsed, or the remaining jiffies (*at least 1*) if the `condition` evaluated to true before the `timeout` elapsed.

> The global variable jiffies holds the number of ticks that have occurred since the system booted. On boot, the kernel initializes the variable to zero, and it is incremented by one during each timer interrupt. Thus, because there are HZ timer interrupts in a second, there are HZ jiffies in a second. The system uptime is therefore jiffies/HZ seconds.

3. wait_event_cmd

- sleep until a condition gets true

```
wait_event_cmd(wq, condition, cmd1, cmd2);
```

where,
  - `wq` – the wait-queue to wait on
  - `condition` – a C expression for the event to wait for
  - `cmd1` – the command will be executed before sleep
  - `cmd2` – the command will be executed after sleep
- The process is put to sleep (TASK_UNINTERRUPTIBLE) until the `condition` evaluates to true. The `condition` is checked each time the wait-queue `wq` is woken up.

4. wait_event_interruptible

- sleep until a condition gets true

```
wait_event_interruptible(wq, condition);
```

where,
  - `wq` – the wait-queue to wait on
  - `condition` – a C expression for the event to wait for
- The process is put to sleep (TASK_INTERRUPTIBLE) until the `condition` evaluates to true or a signal is received. The `condition` is checked each time the wait-queue `wq` is woken up.
- The function will return,
  - **`-ERESTARTSYS`**, if it was interrupted by a signal and,
  - `0`, if condition evaluated to true

5. wait_event_interruptible_timeout

- sleep until a condition gets true or a timeout elapses

```
wait_event_interruptible_timeout(wq, condition, timeout);
```

where,
  - `wq` – the wait-queue to wait on
  - `condition` – a C expression for the event to wait for
  - `timeout` – timeout, in jiffies
- The process is put to sleep (TASK_INTERRUPTIBLE) until the `condition` evaluates to true or a signal is received or timeout elapsed. The `condition` is checked each time the wait-queue `wq` is woken up.
- It returns,
  - `0`, if the `condition` evaluated to FALSE after the `timeout` elapsed,
  - `1`, if the `condition` evaluated to TRUE after the `timeout` elapsed, the remaining jiffies (*at least 1*) if the `condition` evaluated to TRUE before the `timeout` elapsed, or `-ERESTARTSYS` if it was interrupted by a signal.

6. wait_event_killable

- sleep until a condition gets true

```
wait_event_killable(wq, condition);
```

where,
  - `wq` – the wait-queue to wait on
  - `condition` – a C expression for the event to wait for
- The process is put to sleep (TASK_KILLABLE) until the `condition` evaluates to true or a signal is received. The `condition` is checked each time the waitqueue `wq` is woken up.
- The function will return,
  - **`-ERESTARTSYS`**, if it was interrupted by a signal and,
  - **`0`**, if `condition` evaluated to true.

Waking Up Queued Task

- When some Tasks are in sleep mode because of wait-queue, then we can use the below function to wake up those tasks.
  1. `wake_up`
  2. `wake_up_all`
  3. `wake_up_interruptible`
  4. `wake_up_sync` and `wake_up_interruptible_sync`

1. wake_up

- wakes up only one process from the wait queue which is in non-interruptible sleep.

```
wake_up(&wq);
```

where,
  - `wq` – the wait-queue to wake up

2. wake_up_all

- wakes up all the processes on the wait queue

```
wake_up_all(&wq);
```

where,
  - `wq` – the wait-queue to wake up

3. wake_up_interruptible

- wakes up only one process from the wait queue that is in interruptible sleep

```
wake_up_interruptible(&wq);
```

where,
  - `wq` – the wait-queue to wake up

4. wake_up_sync and wake_up_interruptible_sync

```
wake_up_sync(&wq);
wake_up_interruptible_sync(&wq);
```

- Normally, a `wake_up` call can cause an immediate reschedule to happen, meaning that other processes might run before `wake_up` returns.
- The synchronous variants instead make any awakened processes runnable but do not reschedule the CPU.
- This is used to avoid rescheduling when the current process is known to be going to sleep, thus forcing a reschedule anyway.
- Note that awakened processes could run immediately on a different processor, so these functions should not be expected to provide mutual exclusion.

- **Example:**

```
/* Demonstration of wait-queue created by dynamic method */

#include <linux/kernel.h>           /* Kernel debug macros and many more */
#include <linux/init.h>             /* __init* and __exit* macros */
#include <linux/module.h>           /* module_* macros, Required by all modules */
#include <linux/kdev_t.h>           /* MAJOR and MINOR macros */
#include <linux/fs.h>               /* struct file_operations */
#include <linux/cdev.h>             /* struct cdev, cdev_* APIs */
#include <linux/device.h>           /* struct class, class_*, device_* APIs */
#include <linux/wait.h>             /* Required for the wait queues */
#include <linux/kthread.h>          /* kthread_create */

uint32_t read_count = 0;
static struct task_struct *wait_thread = NULL;

dev_t dev = 0;
static struct class *dev_class;
static struct cdev etx_cdev;
int wait_queue_flag = 0;
wait_queue_head_t wait_queue_etx;

static int __init etx_driver_init(void);
static void __exit etx_driver_exit(void);

/*************** Driver Functions **********************/
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,
};

static int wait_function(void *unused)
{
    while(1) {
        printk(KERN_INFO "Waiting For Event...\n");
```

```
        wait_event_interruptible(wait_queue_etx, wait_queue_flag != 0 );
        if(wait_queue_flag == 2) {
            printk(KERN_INFO "Event Came From Exit Function\n");
            return 0;
        }
        printk(KERN_INFO "Event Came From Read Function - %d\n", ++read_count);
        wait_queue_flag = 0;
    }
    do_exit(0);
    return 0;
}

static int etx_open(struct inode *inode, struct file *file)
{
    printk(KERN_INFO "Device File Opened...!!!\n");
    return 0;
}

static int etx_release(struct inode *inode, struct file *file)
{
    printk(KERN_INFO "Device File Closed...!!!\n");
    return 0;
}

static ssize_t etx_read(struct file *filp, char __user *buf, size_t len,
                        loff_t *off)
{
    printk(KERN_INFO "Read Function\n");
    wait_queue_flag = 1;
    wake_up_interruptible(&wait_queue_etx);
    return 0;
}

static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len,
                         loff_t *off)
{
    printk(KERN_INFO "Write function\n");
    return 0;
}

static int __init etx_driver_init(void)
{
    /* Allocating Major number */
    if((alloc_chrdev_region(&dev, 0, 1, "wq_dev")) < 0) {
        printk(KERN_INFO "Cannot allocate major number\n");
        return -1;
    }
    printk(KERN_INFO "Major = %d Minor = %d\n", MAJOR(dev), MINOR(dev));

    /* Creating cdev structure */
    cdev_init(&etx_cdev,&fops);
    etx_cdev.owner = THIS_MODULE;
    etx_cdev.ops = &fops;

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

```
    /* Creating struct class */
    if((dev_class = class_create(THIS_MODULE, "wq_class")) == NULL) {
        printk(KERN_INFO "Cannot create the struct class\n");
        goto r_class;
    }

    /* Creating device */
    if((device_create(dev_class, NULL, dev, NULL, "waitqueue_dynamic")) == NULL) {
        printk(KERN_INFO "Cannot create the Device 1\n");
        goto r_device;
    }

    /* Initialize wait queue */
    init_waitqueue_head(&wait_queue_etx);

    /* Create the kernel thread */
    wait_thread = kthread_create(wait_function, NULL, "WaitThread");
    if (wait_thread) {
        printk("Kernel thread created successfully\n");
        /* start thread */
        wake_up_process(wait_thread);
    } else {
        printk(KERN_INFO "Thread creation failed\n");
        goto r_kthread;
    }

    printk(KERN_INFO "Waitqueue dynamic probed !!!\n");
    return 0;

r_kthread:
    device_destroy(dev_class,dev);
r_device:
    class_destroy(dev_class);
r_class:
    cdev_del(&etx_cdev);
r_cdev:
    unregister_chrdev_region(dev,1);
    return -1;
}

void __exit etx_driver_exit(void)
{
    wait_queue_flag = 2;
    wake_up_interruptible(&wait_queue_etx);
    device_destroy(dev_class,dev);
    class_destroy(dev_class);
    cdev_del(&etx_cdev);
    unregister_chrdev_region(dev, 1);
    printk(KERN_INFO "Waitqueue dynamic Removed !!!\n");
}

module_init(etx_driver_init);
module_exit(etx_driver_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Sunil Vaghela <sunilvaghela09@gmail.com>");
MODULE_DESCRIPTION("Demonstration of waitqueue using dynamic initialization");
```

```
/* sample output */
```

```
debian@beaglebone:~$ sudo insmod wq_dynamic.ko
debian@beaglebone:~$ dmesg | tail -n 5
[ 7483.469942] Major = 240 Minor = 0
[ 7483.470625] Kernel thread created successfully
[ 7483.470636] Waitqueue dynamic probed !!!
[ 7483.477355] Waiting For Event...
debian@beaglebone:~$ sudo cat /dev/waitqueue_dynamic
debian@beaglebone:~$ dmesg | tail -n 5
[ 7504.901508] Device File Opened...!!!
[ 7504.903351] Read Function
[ 7504.903410] Event Came From Read Function - 1
[ 7504.903416] Waiting For Event...
[ 7504.903525] Device File Closed...!!!
debian@beaglebone:~$ sudo cat /dev/waitqueue_dynamic
debian@beaglebone:~$ dmesg | tail -n 5
[ 7512.024751] Device File Opened...!!!
[ 7512.024843] Read Function
[ 7512.024931] Event Came From Read Function - 2
[ 7512.024938] Waiting For Event...
[ 7512.025035] Device File Closed...!!!
debian@beaglebone:~$ sudo cat /dev/waitqueue_dynamic
debian@beaglebone:~$ dmesg | tail -n 5
[ 7515.622090] Device File Opened...!!!
[ 7515.622199] Read Function
[ 7515.622251] Event Came From Read Function - 3
[ 7515.622257] Waiting For Event...
[ 7515.622355] Device File Closed...!!!
debian@beaglebone:~$ sudo rmmod wq_dynamic
debian@beaglebone:~$ dmesg | tail -n 5
[ 7515.622251] Event Came From Read Function - 3
[ 7515.622257] Waiting For Event...
[ 7515.622355] Device File Closed...!!!
[ 7539.522255] Event Came From Exit Function
[ 7539.529978] Waitqueue dynamic Removed !!!
```

Chapter-5 Interrupts in Linux Kernel

- ## Interrupts
  - Suppose you knew one or more guests could be arriving at the door. `Polling` would be like going to the door often to see if someone was there yet continuously. That's what the doorbell is for. The guests are coming, but you have preparations to make, or maybe something unrelated that you need to do. You only go to the door when the doorbell rings. When the doorbell rings, it's time to check the door again. You get more done, and they get quicker responses when they ring the doorbell. This is the `interrupt mechanism`.
  - Another scenario is, Imagine that you are watching TV or doing something. Suddenly you heard someone's voice which is like your Crush's voice. What will happen next? That's it, you are interrupted!! You will be very happy. Then stop your work whatever you are doing now and go outside to see him/her. Similar to us, Linux also stops his current work and distracts because of interrupts and then it will handle them.
  - In Linux, interrupt signals are the distraction that diverts the processor to a new activity outside from normal flow of execution. This new activity is called `interrupt handler` or `interrupt service routine (ISR)` and that distraction is `Interrupts`.

- ## Polling vs Interrupts
  - In `polling` the CPU keeps on checking all the hardwares of the availability of any request, while in `interrupt` the CPU takes care of the hardware, only when the hardware requests for some service.
  - The `polling` method is like a salesperson. The salesman goes from door to door while requesting to buy a product or service. Similarly, the controller keeps monitoring the flags or signals one by one for all devices and provides service to whichever component that needs its service, while an `interrupt` is like a shopkeeper. If one needs a service or product, he goes to him and apprises him of his needs. In case of interrupts, when the flags or signals are received, they notify the controller that they need to be serviced.

- ## What will happen when the interrupt comes?
  - An `interrupt` is produced by electronic signals from hardware devices and directed into input pins on an `interrupt controller` (a simple chip that multiplexes multiple interrupt lines into a single line to the processor). These are the processes that will be done by the kernel.
    1. Upon receiving an interrupt, the interrupt controller sends a signal to the processor.

2. The processor detects this signal and interrupts its current execution to handle the interrupt.
3. The processor can then notify the operating system that an interrupt has occurred, and the operating system can handle the interrupt appropriately.

- Different devices are associated with different interrupts using a unique value associated with each interrupt. This enables the operating system to differentiate between interrupts and to know which hardware device caused such an interrupt. In turn, the operating system can service each interrupt with its corresponding handler.
- Interrupt handling is amongst the most sensitive tasks performed by the kernel and it must satisfy the following:
  - Hardware devices generate interrupts asynchronously (with respect to the processor clock). That means interrupts can come anytime.
  - Because interrupts can come anytime, the kernel might be handling one of them while another one (of a different type) occurs.
  - Some critical regions exist inside the kernel code where interrupts must be disabled. Such critical regions must be limited as much as possible.

- **Interrupts and Exceptions**
  - `Exceptions` are often discussed at the same time as interrupts. Unlike interrupts, exceptions occur synchronously with respect to the processor clock; they are often called synchronous interrupts.
  - `Exceptions` are produced by the processor while executing instructions either in response to a programming error (*e.g. divide by zero*) or abnormal conditions that must be handled by the kernel (*e.g. a page fault*). Because many processor architectures handle exceptions in a similar manner to interrupts, the kernel infrastructure for handling the two is similar.
  - Simple definitions of the two:
    - **Interrupts:** asynchronous interrupts generated by hardware.
    - **Exceptions:** synchronous interrupts generated by the processor.
  - `System calls` (*one type of exception*) on the x86 architecture are implemented by the issuance of a software interrupt, which traps into the kernel and causes execution of a special system call handler. Interrupts work in a similar way, except hardware (not software) issues interrupts.
  - There is a further classification of interrupts and exceptions:

  - **Interrupts**
    - **Maskable** – All `Interrupt Requests (IRQs)` issued by I/O devices give rise to maskable interrupts. A maskable interrupt can be in two states: `masked` or `unmasked`; a masked interrupt is ignored by the control unit as long as it remains masked.
    - **Non-maskable** – Only a few critical events (*such as hardware failures*) give rise to non-maskable interrupts. Non-maskable interrupts are always recognized by the CPU.
  - **Exceptions**
    - **Falts** – Like Divide by zero, Page Fault, Segmentation Fault.

- **Traps** – Reported immediately following the execution of the trapping instruction. Like, Breakpoints.
- **Aborts** – Aborts are used to report severe errors, such as hardware failures and invalid or inconsistent values in system tables.

- For a device's each interrupt, its device driver must register an interrupt handler.

- **Interrupt handler**
  - An `interrupt handler` or `interrupt service routine (ISR)` is the function that the kernel runs in response to a specific interrupt:
    1. Each device that generates interrupts has an associated interrupt handler.
    2. The interrupt handler for a device is part of the device's driver (*the kernel code that manages the device*).
  - In Linux, interrupt handlers are normal C functions, which match a specific prototype and thus enable the kernel to pass the handler information in a standard way.
  - What differentiates interrupt handlers from other kernel functions is that the kernel invokes them in response to interrupts and that they run in a special context called interrupt context. This special context is occasionally called atomic context because code executing in this context is unable to block.
  - Because an interrupt can occur at any time, an interrupt handler can be executed at any time. It is imperative that the handler runs quickly, to resume execution of the interrupted code as soon as possible. It is important that:
    1. **To the hardware:** the operating system services the interrupt without delay.
    2. **To the rest of the system:** the interrupt handler executes in as short a period as possible.
  - At the very least, an interrupt handler's job is to acknowledge the interrupt's receipt to the hardware. However, interrupt handlers can often have a large amount of work to perform.

- **Process Context and Interrupt Context**
  - The kernel accomplishes useful work using a combination of `process contexts` and `interrupt contexts`. Kernel code that services system calls issued by user applications runs on behalf of the corresponding application processes and is said to execute in process context. Interrupt handlers, on the other hand, run asynchronously in interrupt context. Processes contexts are not tied to any interrupt context and vice versa.
  - Kernel code running in process context is `preemptible`. An interrupt context, however, always runs to completion and is `not preemptible`. Because of this, there are restrictions on what can be done from interrupt context.
  - Code executing from interrupt context cannot do the following:
    1. Go to sleep or relinquish the processor
    2. Acquire a mutex
    3. Perform time-consuming tasks
    4. Access user space virtual memory
  - Based on our idea, `ISR` or `Interrupt Handler` should be executed very quickly and it should not run for more time (*it should not perform time-consuming tasks*). What

if I want to do a huge amount of work upon receiving interrupts? So it is a problem right? If we do like that this will happen.
1. While `ISR` runs, it doesn't let other interrupts to run (*interrupts with higher priority will run*).
2. Interrupts with the same type will be missed.

- To eliminate that problem, the processing of interrupts is split into two parts, or halves:
  1. Top halves
  2. Bottom halves

Top Halves and Bottom Halves

- **Top Half**
  - The interrupt handler is the `top half`. The `top half` will run immediately upon receipt of the interrupt and performs only the work that is time-critical, such as acknowledging receipt of the interrupt or resetting the hardware.
- **Bottom Half**
  - The `bottom half` is used to process data, letting the top half to deal with new incoming interrupts.
  - Interrupts are enabled when a `bottom half` runs. The interrupt can be disabled if necessary, but generally, this should be avoided as this goes against the basic purpose of having a `bottom half` – processing data while listening to new interrupts. The `bottom half` runs in the future, at a more convenient time, with all interrupts enabled.

- For example, using the network card:
  1. When network cards receive packets from the network, the network cards immediately issue an interrupt. This optimizes network throughput and latency and avoids timeouts.
  2. The kernel responds by executing the network card's registered interrupt.
  3. The interrupt runs, acknowledges the hardware, copies the new networking packets into main memory, and reads the network card for more packets. These jobs are important, time-critical, and hardware-specific work.
     - The kernel generally needs to quickly copy the networking packet into the main memory because the network data buffer on the networking card is fixed and miniscule in size, particularly compared to the main memory. Delays in copying the packets can result in a buffer overrun, with incoming packets overwhelming the networking card's buffer and thus packets being dropped.
     - After the networking data is safely in the main memory, the interrupt's job is done, and it can return control of the system to whatever code was interrupted when the interrupt was generated.
  4. The rest of the processing and handling of the packets occurs later, in the bottom half.
- If the interrupt handler function could process and acknowledge interrupts within a few microseconds consistently, then absolutely there is no need for top half/bottom half delegation.
- There are 4 bottom half mechanisms are available in Linux:
  1. Workqueue

2. Threaded IRQs
3. Softirq
4. Tasklets

- **Consolidated summary before starting interrupt programming**
  - Interrupt handlers can not enter sleep, so to avoid calls to some functions which have sleep.
  - When the interrupt handler has part of the code to enter the critical section, use spinlocks lock, rather than mutexes. Because if it couldn't take mutex it will go to sleep until it takes the mute.
  - Interrupt handlers can not exchange data with the userspace.
  - The interrupt handlers must be executed as soon as possible. To ensure this, it is best to split the implementation into two parts, top half and bottom half. The top half of the handler will get the job done as soon as possible and then work late on the bottom half, which can be done with `softirq` or `tasklet` or `workqueue`.
  - Interrupt handlers can not be called repeatedly. When a handler is already executing, its corresponding IRQ must be disabled until the handler is done.
  - Interrupt handlers can be interrupted by higher authority handlers. If you want to avoid being interrupted by a highly qualified handler, you can mark the interrupt handler as a `fast handler`. However, if too many are marked as `fast handlers`, the performance of the system will be degraded, because the interrupt latency will be longer.

- **Functions related to Interrupt**
  - Before programming, we should know the basic functions which are useful for interrupts.

1. request_irq

- Register an IRQ.

```
request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags, const char *name, void *dev_id);
```

where,

- **`irq`** – IRQ number to allocate.
- **`handler`** – This is Interrupt handler function.This function will be invoked whenever the operating system receives the interrupt.The data type of return is `irq_handler_t`, if its return value is IRQ_HANDLED, it indicates that the processing is completed successfully, but if the return value is IRQ_NONE, the processing fails.
- **`flags`** – can be either zero or a bit mask of one or more of the flags defined in `linux/interrupt.h`. The most important of these flags are:
  - **`IRQF_DISABLED`**
  - When set, this flag instructs the kernel to disable all interrupts when executing this interrupt handler.
  - When unset, interrupt handlers run with all interrupts except their own enabled.Most interrupt handlers do not set this flag, as disabling all interrupts is bad form.

        - Its use is reserved for performance-sensitive interrupts that execute quickly. This flag is the current manifestation of the `SA_INTERRUPT` flag, which in the past distinguished between "fast" and "slow" interrupts.
    - **IRQF_SAMPLE_RANDOM**
        - This flag specifies that interrupts generated by this device should contribute to the kernel entropy pool. The kernel entropy pool provides truly random numbers derived from various random events.
        - If this flag is specified, the timing of interrupts from this device is fed to the pool as entropy. Do not set this if your device issues interrupt at a predictable rate (*e.g. the system timer*) or can be influenced by external attackers (*e.g. a networking device*). On the other hand, most other hardware generates interrupts at non-deterministic times and is, therefore, a good source of entropy.
    - **IRQF_SHARED**
        - This flag specifies that this handler process interrupts the system timer.
    - **IRQF_TIMER**
        - This flag specifies that the interrupt line can be shared among multiple interrupt handlers. Each handler registered on a given line must specify this flag; otherwise, only one handler can exist per line.
  - **name** – Used to identify the device name using this IRQ, for example, cat / proc / interrupts will list the IRQ number and device name.
  - **dev_id** – IRQ shared by many devices. When an interrupt handler is freed, dev provides a unique cookie to enable the removal of only the desired interrupt handler from the interrupt line. Without this parameter, it would be impossible for the kernel to know which handler to remove on a given interrupt line. You can pass NULL here if the line is not shared, but you must pass a unique cookie if your interrupt line is shared. This pointer is also passed into the interrupt handler on each invocation. A common practice is to pass the driver's device structure. This pointer is unique and might be useful to have within the handlers.
- Returns zero on success and nonzero value indicates an error.
- `request_irq()` cannot be called from interrupt context (other situations where code cannot block), because it can block.

2. free_irq

- Release an IRQ registered by `request_irq`.

```
free_irq(unsigned int irq, void *dev_id);
```

  where,
  - **irq** – IRQ number.
  - **dev_id** – is the last parameter of `request_irq`.
- If the specified interrupt line is not shared, this function removes the handler and disables the line.
- If the interrupt line is shared, the handler identified via dev_id is removed, but the interrupt line is disabled only when the last handler is removed. With shared interrupt lines, a unique cookie is required to differentiate between the multiple

handlers that can exist on a single line and enable `free_irq()` to remove only the correct handler.
- In either case (*shared or unshared*), if `dev_id` is non-NULL, it must match the desired handler. A call to `free_irq()` must be made from process context.

3. enable_irq (unsigned int irq)
- Re-enable interrupt disabled by `disable_irq` or `disable_irq_nosync`.

4. disable_irq (unsigned int irq)
- Disable an IRQ from issuing an interrupt.

5. disable_irq_nosync (unsigned int irq)
- Disable an IRQ from issuing an interrupt, but wait until there is an interrupt handler being executed.

6. in_irq()
- Returns true when in interrupt handler

7. in_interrupt()
- Returns true when in interrupt handler or bottom half

Registering an Interrupt Handler

```
#define IRQ_NO 11

if (request_irq(IRQ_NO, irq_handler, IRQF_SHARED, "etx_device", (void *)(irq_handler)))
{
            printk(KERN_INFO "my_device: cannot register IRQ ");
            goto irq;
}
```

Freeing an Interrupt Handler

```
free_irq(IRQ_NO,(void *)(irq_handler));
```

Interrupt Handler

```
static irqreturn_t irq_handler(int irq,void *dev_id) {
  printk(KERN_INFO "Shared IRQ: Interrupt Occurred");
  return IRQ_HANDLED;
}
```

Questions/Answers:

1. What is Modversioning ?

- A module compiled for one kernel won't load if you boot a different kernel unless you enable CONFIG_MODVERSIONS in the kernel.

2. Difference between dmesg output and /var/log/messages

- We can say that dmesg is the subset of /var/log/messages and is maintained in a ring buffer.
- This information is also sent in real time to syslogd or klogd, when they are running, and ends up in /var/log/messages; when dmesg is most useful is in capturing boot-time messages from before syslogd and/or klogd started, so that they will be properly logged.
- The /var/log/messages contains global system messages, including the messages that are logged during system startup. There are several things that are logged in /var/log/messages including mail, cron, daemon, kern, auth, etc.

3. Kbuild goal definitions

- Goal definitions are the main part (heart) of the kbuild Makefile.These lines define the files to be built, any special compilation options, and any subdirectories to be entered recursively. The most simple kbuild makefile contains one line:
  Example: obj-y += hello.o
  This tells kbuild that there is one object in that directory, named hello.o. hello.o will be built from hello.c or hello.S.
- If foo.o shall be built as a module, the variable obj-m is used. Therefore the following pattern is often used:
  Example: obj-$(CONFIG_HELLO) += hello.o
  $(CONFIG_HELLO) evaluates to either y (for built-in) or m (for module). If CONFIG_HELLO is neither y nor m, then the file will not be compiled nor linked.

4. Procfs vs sysfs vs debugfs

- Sysfs is the commonly used method to export system information from the kernel space to the user space for specific devices. The sysfs is tied to the device driver model of the kernel.
- The procfs is used to export the process-specific information
- The debugfs is used to export the debug information by the developer.

5. Preemptive Kernel vs Nonpreemptive Kernel

Important files in kernel:

include/linux/module.h	Required by all modules, module_* macros,
include/linux/kernel.h	Kernel debug macros and many more
include/linux/kern_levels.h	Kernel log levels macros
include/linux/printk.h	Kernel log level alias function
include/linux/init.h	Declarations of __init* and __exit* macros
include/asm-generic/vmlinux.lds.h	Declaration of init.data, exit.data etc sections
include/linux/moduleparam.h	Declarations of module_param* macros required to pass command line arguments to a module
include/linux/stat.h	S_IWUGO, S_IRUGO, S_IXUGO... etc permissions needed in module_param()
include/linux/fs.h	File_operations , inode , file structures
include/linux/kdev_t.h	MAJOR, MINOR and MKDEV macros
include/linux/cdev.h	Character device init/add/del, struct cdev declaration
include/linux/device.h	struct class, class_create, class_destroy APIs, device_create, device_destroy APIs
include/linux/types.h	Data types for kernel
include/linux/err.h	IS_ERR, PTR_ERR macros
include/linux/uaccess.h	For copy_to_user(), and copy_from_user() APIs It includes architecture specific uaccess.h from ./arch/<arch>/include/asm/uaccess.h
include/linux/of.h include/linux/of_xx.h	Open Firmware APIs to read device tree data
include/linux/kobject.h	struct Kobject, struct kobj_attribute declaration, kobject_* APIs
include/linux/sysfs.h	__ATTR macros, sysfs_* APIs
include/linux/slab.h	kmalloc()
<linux/ioctl.h> /include/asm-generic/ioctl.h ./include/uapi/linux/ioctl.h ./include/uapi/asm-generic/ioctl.h	__IOX macros (__IO, __IOW, __IOR, __IOWR)
include/linux/wait.h	Wait Queue Macros, and APIs

include/linux/delay.h	ssleep, msleep
include/linux/kthread.h	kthread_* APIs
/proc/kallsyms	All symbols exported by the kernel
/proc/modules	Listing of all the dynamically loaded modules
/proc/devices	Listing of all the registered character and block major numbers
/proc/iomem	Listing of on-system physical RAM & bus device addresses
/proc/ioports	Listing of on-system I/O port addresses (specially for x86 systems)
/proc/interrupts	Listing of all the registered interrupt request numbers
/proc/softirqs	Listing of all the registered soft irqs
/proc/kallsyms	Listing of all the running kernel symbols, including from loaded modules
/proc/partitions	Listing of currently connected block devices & their partitions
/proc/filesystems	Listing of currently active file-system drivers
/proc/swaps	Listing of currently active swaps
/proc/cpuinfo	Information about the CPU(s) on the system
/proc/meminfo	Information about the memory on the system, viz. RAM, swap, …
arch/arm/boot/dts/Makefile	Lists which DTBs should be generated at build time
Documentation/admin-guide/devices.txt	To see which major numbers have been assigned
Documentation/devicetree/bindings	DT bindings recognized by the kernel
scripts/dtc	Device tree compiler source directory
arch/<arch>/boot/dts	Architecture specific Device Tree Source(DTS) files path, where <arch> is a specific architecture. E.g. For arm architecture dts path would be : arch/arm/boot/dts
/proc/device-tree /sys/firmware/devicetree/base/	Parsed Device-tree

TODO:

- Add syslog and logd tutorial
- Add all links tutorial in the pdf itself and give its link
- Kernel makefile tutorial
- Explore more in include/asm-generic/vmlinux.lds.h --> .init, .initdata and .exit sections
- MODULE_DEVICE_TABLE explanation
- Go through Major/Minor numbers document (Documentation/admin-guide/devices.txt)
- Struct inode explanation
- Clarification on using Try_module_get and put_module
- Graph of calling APIs in constructor and destructor for all programs
- Mention old kernel version APIs in different box
- Finish procfs section
- Difference between device driver and kernel module
- Rearrange/Rename sections
- Do all embysis practicals
- Export symbol topic
- Bootlin training
- Yocto bsp for beaglebone black in git repository with latest kernel
- Open firmware APIs for accessing device tree source
- Explore more on device tree - if any topic is left
- Explore more on sysfs and procfs(add more example if needed or update current example to cover all APIs)
- Kmalloc and Kfree explanation
- Jiffies
- What if module exit before kernel thread finish execution
- container_of macro
- Kthread
- Interrupt programming example