# The Design and Analysis of Char Driver Model for ARM Architecture

Dumpeti Sathish Kumar

Department of Computer Science Engineering AURORA College of Engineering & Technology Bhongiri , Nalgonda - 508116. India.

***Abstract – At present ARM and Embedded Linux has become research focus in embedded system field. Best support by the Linux community is also an important aspect why the Linux is used more in embedded systems these days. Here we focus on adding propreitory drivers, compiling kernel, Root File-system for target board, and analysis of embedded Linux. It also describes about the importance of operating system in embedded applications, the theory of Linux device drivers and a driver model of char driver.***

***Keywords – Embedded system; ARM; Cross compiler; Linux2.6.xx, Device Drivers, char driver.***

## I. INTRODUCTION

An embedded system is a computer system designed to perform one or a few dedicated functions often with real-time computing constraints. It is embedded as part of a complete device often including hardware and mechanical parts. By contrast, a general-purpose computer, such as a personal computer (PC), is designed to be flexible and to meet a wide range of end-user needs. Embedded systems control many devices in common use today.

Embedded systems span all aspects of modern life and there are many examples of their use. Telecommunications systems employ numerous embedded systems from telephone switches for the network to mobile phones at the end-user. Computer networking uses dedicated routers and network bridges to route data. Consumer electronics include personal digital assistants (PDAs), mp3 players, mobile phones, videogame consoles, digital cameras, DVD players, GPS receivers, and printers. Many household appliances, such as microwave ovens, washing machines and dishwashers, are including embedded systems to provide flexibility, efficiency and features.

*Embedded Linux*

Embedded Linux is customized Linux kernel which runs on Embedded Systems. As the Linux is free and open source operating system many embedded systems are running Linux. Best support by the Linux community is also an important aspect why the Linux is used more in embedded systems these days.

Embedded Systems have limited resources like memory and have limited processing capacity. Linux kernel is not enough to be run on embedded systems and also some embedded devices need real time processing which is not fully supported by the Linux kernel. As embedded systems are made to execute a specific task, there is no need to have all services provided by the Linux. Embedded systems use customized and tailored Linux called Embedded Linux which is suitable for embedded devices.

## II. EMBEDDED OPERATING SYSTEM

As increasing the addition of more and more applications in the embedded devices, there is a need for some sort of instructions and procedures to follow to maintain all these applications. These set of instructions and rules to maintain all the resources and memory between the applications is what we call a embedded operating system.

An important difference between most embedded operating systems and desktop operating systems is that the application including the operating system, is usually statically linked together into a single executable image. Unlike a desktop operating system, the embedded operating system does not load and execute applications. This means that the system is only able to run a single application.

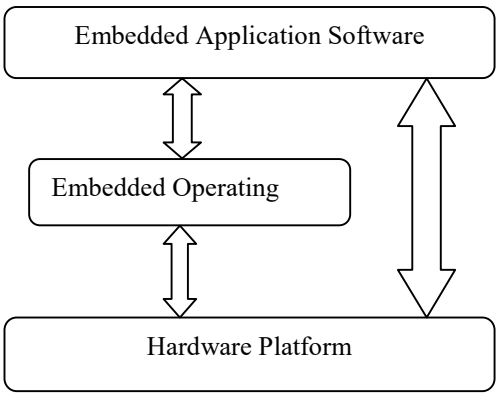



Fig. 1: Embedded System Structure

## III. DEVICE DRIVER SPECIFICATION LANGUAGE

Here it explains about the device driver model. The device driver specification language is the key concept that contains a higher abstraction of device driver source code. A device driver is a program that controls a particular type of device that is attached to the computer system (CPU).

*Specification language consists of the following parts:*

*Basic Information:*

The basic information includes the author, license, kernel version, and target CPU architecture.

*Device information:*

The device information includes the device type, device name, vendor/product ID, and major/minor number.

*File operations:*

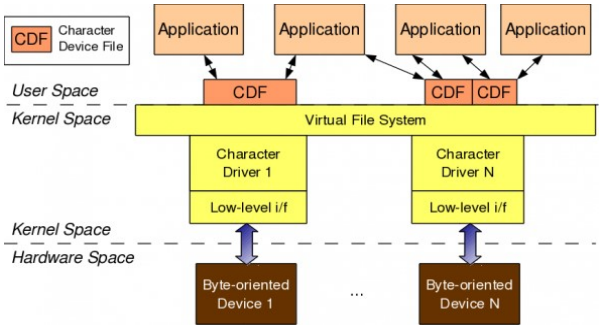
The file operations are mostly in charge implementing system calls and are named open, read, write, and so on. Thus, the commands and arguments information is also included in the file operations specification.

*Module Programming and Drivers Overview*

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. For example, one type of module is the device driver, which allows the kernel to access hardware connected to the system. Without modules, we would have to build monolithic kernels and add new functionality directly into the kernel image. Besides having larger kernels, this has the disadvantage of requiring us to rebuild and reboot the kernel every time we want new functionality.

Kernel modules must have at least two functions: a "start" (initialization) function called init_module() which is called when the module is insmoded into the kernel, and an "end" (cleanup) function called cleanup_module() which is called just before it is unloaded. Actually, things have changed starting with kernel 2.3.13. You can now use whatever name you like for the start and end functions of a module. . In fact, the new method is the preferred method. However, many people still use init_module() and cleanup_module() for their start and end functions

```
int init_module(void)
{
 printk("<1>Hello world 1.\n");
 /*
 * A non 0 return means init_module failed; module can't be loaded.
 */
return 0;
}
```




```
void cleanup_module(void)
{
printk(KERN_ALERT "Goodbye world 1.\n");
}.
```

Typically, 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.

*Classes of Devices*

The Linux way of looking at devices distinguishes between three fundamental device types. Each module usually implements one of these types, and thus is classifiable as a char module, a block module, or a network module.

*Character devices*

A character (char) device is one that can be accessed as a stream of bytes (like a file); a char driver is in charge of implementing this behavior. Such a driver usually implements at least the open, close, read, and write system calls. The text console (/dev/console) and the serial ports (/dev/ttyS0 and friends) are examples of char devices, as they are well represented by the stream abstraction. Char devices are accessed by means of filesystem nodes, such as /dev/tty1 and /dev/lp0. The only relevant difference between a char device and a regular file is that you can always move back and forth in the regular file, whereas most char devices are just data channels, which you can only access sequentially. There exist, nonetheless, char devices that look like data areas, and you can move back and forth in them; for instance, this usually applies to frame grabbers, where the applications can access the whole acquired image using mmap or lseek.

As shown in Figure 1, for any user-space application to operate on a byte-oriented device (in hardware space), it should use the corresponding character device driver (in kernel space). Character

driver usage is done through the corresponding character device file(s), linked to it through the virtual file system (VFS). What this means is that an application does the usual file operations on the character device file. Those operations are translated to the corresponding functions in the linked character device driver by the VFS. Those functions then do the final low-level access to the actual device to achieve the desired results.

Char devices are accessed through names in the filesystem. Those names are called special files or device files or simply nodes of the filesystem tree; they are conventionally located in the /dev directory. Special files for char drivers are identified by a "c" in the first column of the output of ls –l.If you issue the ls –l command, you'll see two numbers (separated by a comma) in the device file entries before the date of the last modification, where the file length normally appears. The following listing shows a few devices as they appear on a typical system.

```
crw-rw-rw-  1 root root 1,3 Apr 11 2002 null
crw--------- 1 root root 10,3 Apr 11 2002 psaux
crw---------  1 root root 4,1 Oct 28 2002 tty1
crw-rw-rw- 1 root tty 4,63 Apr 11 2002 ttyS0
crw-rw-rw- 1 root tty 4,64 Apr 11 2002 ttyS1
```

Within the kernel, the dev_t type (defined in <linux/types.h>) is used to hold device number both the major and minor parts.To obtain the major or minor parts of a dev_t, use:

```
MAJOR(dev_t dev);
MINOR(dev_t dev);
```

If, instead, you have the major and minor numbers and need to turn them into a dev_t, use:

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

*Block devices*

Like char devices, block devices are accessed by filesystem nodes in the /dev directory. A block device is a device (e.g., a disk) that can host a filesystem. In most Unix systems, a block device can only handle I/O operations that transfer one or more whole blocks, which are usually 512 bytes (or a larger power of two) bytes in length. Linux, instead, allows the application to read and write a block device like a char device—it permits the transfer of any number of bytes at a time. As a result, block and char devices differ only in the way data is managed internally by the kernel, and thus in the kernel/driver software interface. Like a char device, each block device is accessed through a filesystem node, and the difference between them is transparent to the user. Block drivers have a completely different interface to the kernel than char drivers.

*Network interfaces*

Any network transaction is made through an interface, that is, a device that is able to exchange data with other hosts. Usually, an interface is a hardware device, but it might also be a pure software device, like the loopback interface. A network interface is in charge of sending and receiving data packets, driven by the network subsystem of the kernel, without knowing how individual transactions map to the actual packets being transmitted. Many network connections (especially those using TCP) are stream-oriented, but network devices are, usu-ally, designed around the transmission and receipt of packets. A network driver knows nothing about individual connections; it only handles packets.

*Char Device Registration :*

So far, we have reserved some device numbers for our use, but we have not yet connected any of our driver's operations to those numbers. The file_operations structure is how a char driver sets up this connection. The structure, defined in <linux/fs.h>, is a collection of function pointers. Each open file (represented internally by a file structure, which we will examine shortly) is associated with its own set of functions (by including a field called f_op that points to a file_operations structure). The operations are mostly in charge of implementing the system calls and are therefore, named open, read, and so on.

```
struct file_operations scull_fops = {
.owner = THIS_MODULE,
.llseek = scull_llseek,
.read = scull_read,
.write = scull_write,
.ioctl = scull_ioctl,
.open = scull_open,
.release = scull_release,
};
```

Conventionally, a file_operations structure or a pointer to one is called fops (or some variation thereof). Each field in the structure must point to the function in the driver that implements a specific operation, or be left NULL for unsupported opera-tions. The exact behavior of the kernel when a NULL pointer is specified is different for each function, as the list later in this section shows.

The kernel uses structures of type struct cdev to represent char devices internally. Before the kernel invokes your device's operations, you must allocate and register one or more of these structures. Tto obtain a standalone cdev structure at runtime, you may do so with code such as:

```
struct cdev *my_cdev = cdev_alloc( );
my_cdev->ops = &my_fops;
```

However, that you will want to embed the cdev structure within a device-specific structure of your own;

that is what scull does. In that case, you should initialize the structure that you have already allocated with:

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```

Either way, there is one other struct cdev field that you need to initialize. Like the file_operations structure, struct cdev has an owner field that should be set to THIS_MODULE. Once the cdev structure is set up, the final step is to tell the kernel about it with a call to:

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

To remove a char device from the system, call cdev_del(). Clearly, you should not access the cdev structure after passing it to cdev_del.

```
void cdev_del(struct cdev *dev);
```

char driver implementation :

```
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <linux/init.h>
#include <linux/cdev.h>
#include <linux/sched.h>
#include <linux/errno.h>
#include <asm/current.h>
#include <linux/device.h>
#define CHAR_DEV_NAME "DEVICE-NAME"
#define MAX_LENGTH 4000
#define SUCCESS 0
static char *char_device_buf;
struct cdev *new_cdev;
dev_t mydev;
int count=1;
static int char_dev_open(struct inode *inode,
struct file  *file)
{
/*
- char_dev_open is the first driver function invoked when application interacts with device.
- Application Validation should verified in this function.
*/
return SUCCESS;
}
static int char_dev_release(struct inode *inode,
struct file *file)
{
/*
- Exit routine of driver.
- Release system resources with respective this driver
*/
return SUCCESS;
}
static ssize_t char_dev_read(struct file *file,
char *buf,
size_t lbuf,
loff_t *ppos)
{
/*
-  char_dev_read Function to read device information
*/
return size;
}
static ssize_t char_dev_write(struct file *file,
const char *buf,
size_t lbuf,
loff_t *ppos)
{
/*
-    char_dev_write function to write onto the device based on device specific register.
*/
return size;
}
static struct file_operations char_dev_fops = {
.owner = THIS_MODULE,
.read = char_dev_read,
.write = char_dev_write,
.open = char_dev_open,
.release = char_dev_release,
};
static __init int char_dev_init(void)
{
int ret;
alloc_chrdev_region (&mydev, 0, count,
```

```
CHAR_DEV_NAME)
 new_cdev = cdev_alloc ();
cdev_init(new_cdev,&char_dev_fops);
ret=cdev_add(new_cdev,mydev,count)
return 0;
}
static __exit void  char_dev_exit(void)
{
 cdev_del(new_cdev);
 unregister_chrdev_region(mydev,1);
 printk(KERN_INFO "\n Driver unregistered \n");
}
module_init(char_dev_init);
module_exit(char_dev_exit);
```

## IV. ADDING NEW DRIVER TO KERNEL SOURCE CODE

In order to add a new driver or feature to the Kernel, first we'll need to select the appropriate location inside the tree to store the code. Once selected, we'll need to add a new configuration option in the corresponding Kconfig file, and update the directory's Makefile to compile our new code, once our configuration option has been selected The Kconfig has a basic configuration syntax that allows you to add configuration options of various types, create dependencies and write a few lines of description.

*Adding a new configuration option:*

Select a unique configuration option name. Otherwise, there will be a collision and a configuration mess. You can grep the .config file to make sure your selected name is not taken. For this example, the name is CONFIG_NEW-CHAR_DEVICE. Edit the appropriate Kconfig file. In this example, we will edit the drivers/char/Kconfig file:

vim /root/linux-2.6.34/drivers/char//Kconfig

config CONFIG_NEW-CHAR_DEVICE

tristate "New Char Driver "

depends on SERIAL_NONSTANDARD && (ISA || EISA || PCI)

select FW_LOADER

help

To compile this driver as a module, choose M here. we add our new configuration option in the appropriate Makefile in order to build the code when it is enabled. In this example, we will edit the drivers/char/Makefile:

vim /root/linux-2.6.34/drivers/char/Makefile

obj-$( CONFIG_NEW-CHAR_DEVICE) += driver-name.o

## V. BUILDING LINUX SOURCE CODE WITH NEW DRIVER:

Cross-Tool chain is used to compile linux source code. BuildRoot is the utility to develop Cross-Tool chain.Buildroot is useful mainly for people working with small or embedded systems, using various CPU architectures (x86, ARM, MIPS, PowerPC, etc.): it automates the building process of your embedded system and eases the cross-compilation process.

#tar -xvf /buildroot-2010.05.tar.gz

#cd buildroot-2010.05

#tar -xvf /dl.tar.gz

#make menuconfig

Target Architecture (arm)

Target Architecture Variant (arm920t)

ToolChain:

Kernel Headers (Linux 2.6.33.x kernel headers)

uClibc C library Version (uClibc 0.9.31.x)

GCC compiler Version (gcc 4.3.4)

Now the arm compiler utilities are located in /buildroot-2010.05/output/staging/usr/bin/arm-linux-

To compile linux source code select processor type and new peripheral drivers as follows:

#tar -xvf /linux-2.6.34.tar.bz2

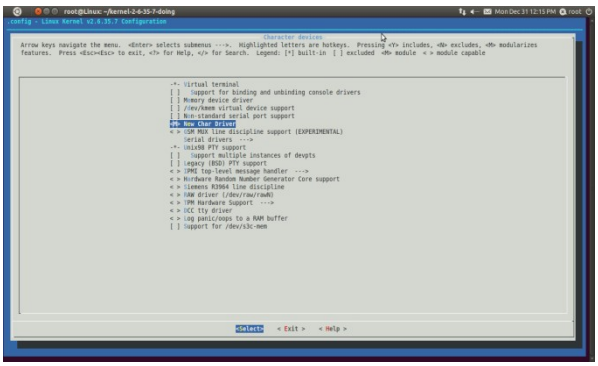
#cd linux-2.6.34

#make ARCH=arm menuconfig

#make menuconfig

#make CROSS_COMPILE=../buildroot/output/staging/usr/bin/arm- linux-

On successful compilation, kernel image is generated under arch/arm/boot/uImage.



## VI.CONCLUSION

In this paper Linux kernel has been analyzed deeply, the importance of operating system in embedded

systems module programming, char driver functionality, adding properitary drivers to linux source and cross-compiling linux as per architecture. As the new 2.6.xx kernel has begun to make new improvements in the new driver program, we can use new functions, in order to achieve the flexibility of the module.

## VII. REFERENCES


[1] Maurice J. Bach – ―The Design of the UNIX Operating System PHI.

[2] Daniel P. Bovet – ―Understanding the Linux Kernel O'Reilly

[3] Raj Kamal – ―Embedded Systems Architecture, Programming and Design TMH

[4] Jonathan Corbet – ―Linux Device Driver O'Reilly

[5] Steve Furber – ―ARM System-on-Chip Architecture PEARSON.

[6] W.Yeong, T.Howes and S. Kille, "Lightweight DirectoryAccess Protocol", RFC 1777.

[7] J.Allen and M.Mealling "The Architecture of the Common Indexing Protocol (CIP)", INTERNET- DRAFT <draft-find-cip-arch-00.txt>,9 June 1997.

[8] The Directory: Overview of Concepts, Models and Service. CCITT Recommendation X.500, 1988

[9] Information Processing Systems -- Open Systems Interconnection --The Directory: Overview of Concepts, Models and Service.ISO/IEC JTC 1/SC21; International Standard 9594-1, 1988

[10] M.Wahl,A.Coulbeck, T.Howesand S.Kille,"Lightweight Directory Access Protocol (v3): Attribute Syntax Definitions", RFC 2252, december 1997


◈◈◈