Behavioral Analysis of Embedded Linux Device Drivers on BeagleBone Black

Jignesh J. Patoliya[1], Sagar B. Patel [2] , Miral M. Desai [3], Ninad Desai[4]

[1,3] Assistant Professor, [2,4] Student

[1,2,3,4] Electronics and Communication Department, C. S. Patel Institute of Technology, Charotar University of Science and Technology, Anand, Gujarat, India

[2]sagarp3199@gmail.com

ABSTRACT

In the contemporary era, the Embedded Linux technology is in tremendous demand for various high-end applications like Internet of Things, Industrial Robotics, Smart Devices, Supercomputers. There is use of kernel device drivers in lowest level of Android and iOS which is emerging for more development in terms of the efficiency and robustness. The speed for communicating between the devices in all the technologies is the vital part, hence, the use of appropriate Embedded Linux Device Driver is needed out of IOCTL (Input Output Control), Procfs, Sysfs. The proposed paper relates to the behavioral analysis of the different types of device drivers on the ARM (Advanced RISC Machine) based platform BeagleBone Black Board. The implementation of these drivers is depicted in the proposed paper by cross-compiling the device driver from x86 platform to BeagleBone Black platform. The OS running on the BeagleBone Black is Linux kernel based Debian. The device driver is basic interface between the software and hardware for specific functionality.

Keywords

Embedded Linux, Device Driver, BeagleBone Black Board, Cross-Compilation, Input Output Control (IOCTL)

Article Received: 10 August 2020, Revised: 25 October 2020, Accepted: 18 November 2020

Introduction

Embedded Linux is the use of Embedded Systems running on the Linux kernel. The use of Embedded Linux in all the major embedded industries have risen exponentially due to the emergence of various development boards like Raspberry Pi, BeagleBone Black, ROCKPro64, Asus Tinker board, Onion Omega2, Odroid-XU4. Linux is utilized in these applications due to its major advantages like multitasking, open-source development and support to plethora of architectures like x86, ARM, MIPS, PowerPC and Alf-Egil Bogen Vegard Wollan RISC (AVR). Embedded Linux is used in the plenty of quirky applications like Robotics, Smart Devices, IoT and ML, etc.

Device Drivers are the simple black boxes interconnecting software and hardware for implementing the specific functionality and are often written in C language. These drivers are loadable kernel modules which can be dynamically loaded for certain applications in terms of their hardware functionality. This dynamic loading of modules reduces memory consumption in kernel space and hence increases the efficiency of the system. For developing this driver, we need the cross- compilation supported OS which can be easily used for deploying the module in BeagleBone Black. There are three

major types of device drivers namely Character, Block and Network. Character device driver deals with the byte wise data transfer and keyboard, mouse, camera are some examples of character devices. Block device driver deals with the transfer in form of collection of bytes in which back and forth communication is possible and pen-drives, hard-disks are some examples of block device drivers. Network devices communicate with the use of packets on the system request and deals with processes like synchronization, routing, session management, packet handling for Wi-Fi and Ethernet [13].

Operating System divides the virtual memory into user space and kernel space for efficient functioning and authorized functions. Kernel space is reserved for running the device drivers, kernel extensions, memory management and process management tasks. Inside the user space, simple user applications run in the memory area which can be swapped out when it is needed. There is use of system calls which helps to talk between the user space and kernel space for user applications functioning like Real Device Driver, IOCTL, Procfs, Sysfs, etc.

BeagleBone Black (BBB) is robust and efficient open-source low-power single-board computer platform for Embedded Linux development produced by TI (Texas Instruments). It boots Linux very quickly and has great community support where developers around the globe collaborate in the development of the board. It has very high expandability supporting plethora of Capes (Additional Hardware modules for high-end specific applications like Crypto Cape for OS level Security). These boards are showing the prospering growth in the field of Internet of Things (IoT), Drones, Robotics, Smart Appliances, Smart Cities, High end Industrial Applications [1].

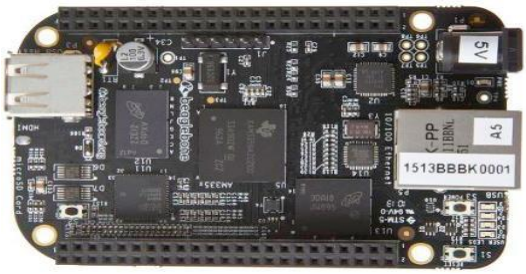

Fig 1. BeagleBone Black

The BeagleBone Black has TI Sitara AM3358BZCZ100 Processor with 1 GHz and 2000 MIPS speed. 512 DDR3L 800 MHz SRAM Memory and 4 GB with 8 bit eMMC onboard flash is present in this Single Board Computer. It works on 1 GHz ARM Cortex A8 and has GPU of PowerVR SGX530. It has total external 92 pins in two headers (P8 and P9 with 46 each). It has Ethernet (10/100 RJ45), SD, MMC, USB, micro HDMI Connectors on the board. It exhibits great versatility providing connectivity to large number of devices using various protocols and standards like 4 x UART (Universal Asynchronous Receiver Transmitter), LCD (Liquid Crystal Display), MMC1 (MultiMedial Card), 2 x SPI (Serial Peripheral Interface), 2x I2C (Inter-Integrated Circuit), ADC (Analog to Digital Converter), 4 Timers, 8 PWMs (Pulse Width Modulation) and 2x CAN (Controller Area Network). Three on-board buttons are there namely reset, power and boot. The Software Compatible with BBB is Linux, Android, and Cloud9 IDE (Integrated Development Environment) with Bonescript.

Types Of Device Driver Communication

The Real Device Driver is the way of simple interaction between the kernel space and user space. It uses the kernel dynamic memory allocation and freeing for storing the value of input or output required for the hardware. The kmalloc() and kfree() APIs are used for this storage and functioning. For transfer of data from user space to kernel space and vice-versa, copy_from_user() and copy_to_user() APIs are available. It is used for simple transfer of bytes like basic GPIO applications. This driver communication is applicable to mostly the character devices; it is not easily available for network and block devices.

IOCTL (Input Output Control) is also used for communicating to devices from kernel space to user space. It is the most ubiquitous system call used in almost all the types of device drivers for communication. It is inculcated for specific purposes of the device for which the kernel does not have system call by default. Some basic operations that can be easily done by IOCTL drivers are adjusting the volume in the system, ejecting the media from CD drive, changing the baud rate of serial port, changing the led port numbers, reading or writing the device registers, etc. It is done by creating IOCTL command in driver and also the user space application for linking the driver. Furthermore, IOCTL function is defined in the device driver file in the kernel. Inside the user-space program, the IOCTL system call is used to perform the communication.

Procfs is the virtual filesystem which is also used for interfacing between kernel and user space. It does not exist on the memory disk [12]. It is used to showcase the information of the processes running on the system. There is ample information stored and transferred from kernel to user space and vice-versa like modules information, information of interrupts, memory usage, registered devices, partitions, input output ports and also cpu information. It is also used to debug the kernel module using kernel proc entry. Procfs interaction can be implemented using proc filesystem structure which includes read, write, open and release functions for interfacing with user applications. The successful write and read can be checked in /proc directory in the kernel.

The sysfs virtual filesystem is also efficient way for communicating between kernel and user. It is also tied to the whole model of kernel device driver including details of devices and modules. It contains extensive details of firmware modules, filesystems, power, classes of devices. This can be easily done using kernel objects (Kobject) and /sys directory and sysfs file. It especially provides uniform way for information and control points transfer including the device framework when devices are registered in /sys kernel directory. It is most common type of interaction for the GPIO input output in the many Embedded Linux based Single Board Computers like Raspberry Pi and BeagleBone Black.

Related Work

Many researchers and authors around the globe have carried out research and analysis on device drivers and below mentioned are its references.

Andrea C. et. al conducted the comprehensive study of filesystems used in Linux. There are various insights including the efficiency of each file systems and also the improvement of debugging tools [2]. Nirav Trivedi et. al implemented the Linux based device driver on Linux host machine. The inserting and removal of module printing the string Hello World is taken into consideration [3]. Anrey V. et. al described the most common Linux filesystems in detail. The study in the paper is performed on the GNU Linux for specifically measuring the filesystem performance and efficiency and finally analyzing the best ways to store the data [4].

Bhushan J. et. al studied the Linux API usages and compatibility with the devices. The surveys of different APIs were taken for considering the relative importance of end-user applications. Also, the concepts of security and complexity of the Linux APIs were instilled with the use of IOCTL, fcntl, prctl [5]. Nicholas F. implemented the user-level device drivers depicting the difference in performance. More robust techniques are introduced for building the Linux systems like for high-end devices used for Gigabit Ethernet connection[6]. Murali B. implemented the device driver for pseudo device using the loadable kernel module (LKM). The insertion and removal of modules with the reliable C language coding is inculcated [7].

Yong C. designed and practised the Embedded Linux File System using IntelDBPXA250 development platform. The description of virtual file system and Flash filesystem is undertaken. The comparison and proper plan of virtual and physical filesystems is taken for satisfactory and efficient results for the requirement of proper embedded system [8]. Nithya E. implemented the character device driver to read the processor's information using /proc filesystem in Linux. The proper detailed information related to the addresses in the process is deeply discussed with virtual addressing consideration [9].

Device Driver Model

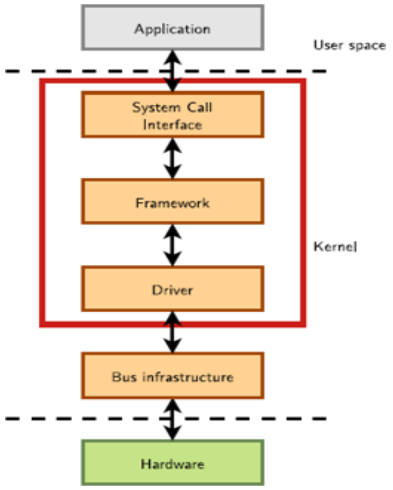

Fig 2. Device Model [10]

Linux Device Driver interfacing with the framework allows the driver to expose the specific hardware features. The bus infrastructure is used to detect and communicate with the hardware. This device driver is interfaced with the user-space application with the use of the system calls like procfs, sysfs, kernel APIs and IOCTL.

Implementation

The Real Device Driver implementation on the Single Board Computer BeagleBone Black is done using simple APIs like copy_to_user() and copy_from_user() with the inclusion of kernel device driver file and user space application. The implementation of Procfs, sysfs and IOCTL is done using the specific APIs of these system calls as shown below.

```
charusat@charusat-HP-Pro-3330-MT: ~
[root@BeagleBone ~]# insmod realdevice.ko
[root@BeagleBone ~]# ./testfinal
***************
****Welcome****
**Please Enter Option***
   1. Write
   2. Read
   3. Exit
1
Option = 1
Enter the string to write into the driver:
charusat
Data writing.......
Write Done....
**Please Enter Option***
   1. Write
   2. Read
   3. Exit
2
Option = 2
Data Reading..
Read Done..
Data = charusat
**Please Enter Option***
   1. Write
   2. Read
   3. Exit
3
Option = 3
[root@BeagleBone ~]#
```

Fig 3. Real Device Driver Output

IOCTL Driver is implemented using driver file in kernel space and user space test application for verifying the data received or sending the data. There is the common IOCTL command used in both the files for reading or changing the data.

```
[root@BeagleBone ~]# ls
bin             diocontrol.ko  testioctl
[root@BeagleBone ~]# insmod diocontrol.ko
[root@BeagleBone ~]# ./testioctl
*********************************

Opening Driver
Enter the Values to send
10
7
Writing Value 1 to Driver
Reading Value 1 from Driver
Writing Value 2 to Driver
Reading Value 2 from Driver
Values are 10 and 7
Closing Driver
[root@BeagleBone ~]#
```

Fig 4. IOCTL Driver Output

The Procfs file operations structure is used for reading, writing the file created inside /proc directory for communicating between the kernel space and user space. The echo and cat commands are used directly for writing and reading the data of the variable in the file inside /proc/ directory respectively.

```
charusat@charusat-HP-Pro-3330-MT: ~
[root@BeagleBone ~]# ls
bin           procdevice.ko
[root@BeagleBone ~]# insmod procdevice.ko
[root@BeagleBone ~]# ls /proc/ | grep etx
etx_proc
[root@BeagleBone ~]# insmod procdevice.ko
[root@BeagleBone ~]# ls /proc | grep etx
etx_proc
[root@BeagleBone ~]# echo 'LED ON/OFF' > /proc/etx_proc
[root@BeagleBone ~]# cat /proc/etx_proc
LED ON/OFF

^C
[root@BeagleBone ~]# rmmod procdevice.ko
[root@BeagleBone ~]#
```

Fig 5. Procfs Driver Output

Sysfs driver is deployed by creating directory inside /sys/kernel directory and also creating the sysfs file inside it. The communication is implemented by creating the sysfs attribute including the show () and store () APIs for the read and write operations respectively.

```
charusat@charusat-HP-Pro-3330-MT: ~
[root@BeagleBone ~]# insmod devsysfs.ko
[root@BeagleBone ~]# ls /sys/kernel/ | grep etx
etx_sysfs
[root@BeagleBone ~]# echo 90 > /sys/kernel/etx_sysfs/etx_va
lue
[root@BeagleBone ~]# cat /sys/kernel/etx_sysfs/etx_value
90
[root@BeagleBone ~]# rmmod devsysfs.ko
[root@BeagleBone ~]#
```

Fig 6. Sysfs Driver Output

Analysis

The IOCTL interface is the fastest because of its direct access and because the file layer is removed from the function calls, whereas using sysfs, user needs to get into the

/sys directory for performing the functionality of the driver. Procfs is old and unstructured interface which store the information in the memory but sysfs is the inclusion of almost all the process information and it stores all the system information in the disk. Normal Device Driver only performs the basic kernel system calls easily and efficiently but it lacks in performing higher level applications like changing the baud rate or increasing the volume.

Furthermore, there is no need of test application in user space for its functioning in Procfs and sysfs, which accounts them to be prone to errors compared to simple device driver or IOCTL interface. Error in the main Procfs or sysfs driver file can corrupt the kernel or OS as its functions are processed in the kernel space. IOCTL performs in the interrupt context as it will execute its driver whenever there is certain type of interrupt which are stored in /proc/interrupts file. IOCTL takes less time to undergo higher number of iterations as compared to sysfs. The difference is in the milliseconds, but it accounts to the efficient functioning and real-time processing of the devices [11].

However, sysfs has a very simple structured interface compared to the other interfaces. These interfaces are more flexible than IOCTL. It is often used for simple GPIO tasks. IOCTL is needed for increasing the functionality of the driver as it has complex structure available for high-end applications of devices. The use of these interfaces will depend on the functions needed for the device drivers.

Conclusion

This detailed analysis of the different communication interfaces between the kernel space and user space is presented in the proposed paper which includes the Real Device Driver, IOCTL interface, and Procfs and sysfs filesystems. The efficiency of each interface is the vital part for deciding its use for functioning of the devices. Like IOCTL is best use for fast efficient complex tasks whereas sysfs and Procfs deals with the structured GPIO like tasks easily. The needed functionality and information of process or system decides which interface is useful for the specific device driver. Basically, these interfaces are not only the factor corresponding to the speed of functioning of driver but also the hardware used must be compatible and robust for efficient applications. BeagleBone Black is robust and designed for high-end computing applications and therefore these interfaces are efficiently encountered in this Embedded Linux board.

References

[1] A. Nayyar and V. Puri, “A Comprehensive Review of BeagleBone Technology: Smart Board Powered by ARM” In International Journal of Smart Home, Vol. 10, No. 4 (2016)

[2] Andrea C., Lanyue L., Arpaci D., Remzi H. and Shan Lu, “A Study of Linux File System Evolution” In 11th USENIX Conference on File and Storage Technologies (FAST ’13)

[3] Nirav T., Himanshu P. and Dharmendra C., “Fundamental Structure of Linux based Device Driver and Implementation on Linux Host Machine” In International Journal of Applied Information Systems (IJAIS), Vol. 10 No. 4, January 2016

[4] Andrey V., Alexander G., “Research of Performance Linux Kernel File Systems” In International Journal of Advanced Studies (IJAS), Vol. 5, No. 2, 2015

[5] Bhushan T., Chia-Che T., Nafees A., Donald E., “A Study of Modern Linux API Usage and Compatibility” In Stony Brook Univeristy Conference

[6] Ben L., Peter C., Nicholas F., Stefan G., Charles G., Luke M., Daniel P., Yueting S., Kevin E. and Gernot H., “User-level Device Drivers: Achieved Performance” In Journal of Computer Science and Technology, September 2005

[7] Murali B., “Linux Device Driver Coding for Pseudo Device” In International Journal of Computational Engineering Research (IJCER)

[8] YongChung W., “The Design and Practice of Embedded Linux File System” In 3rd International Conference on Management, Education, Information and Control (MEICI 2015)

[9] Nithya E., “Implementation of Character – Mode Device Driver to Read the Processors GDT” in International Journal of Innovative Research in Science, Engineering and Technology (IJIRSET), Vol. 3, Issue 10,October 2014

[10] Linux Kernel Device Driver Model (Internet Source): https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf

[11] Linux Procfs Sysfs IOCTL Interfaces (Internet Source): https://stackoverflow.com/questions/19554154/in-general-on-uclinux-is- ioctl-faster-than-writing-to-sys-filesystem

[12] Robert Love, “Linux Kernel Development”, 3rd Edition

[13] Alessandro Rubini & Jonathan, “Linux Device Drivers”, O’Reily,2001