Graphical Driver Development Kit (GDDK) for Linux

Kumar Anik
Indian Institute of Information Technology, Allahabad (Amethi Campus) India

Abhishek Sharma
Indian Institute of Information Technology, Allahabad (Amethi Campus). India

Dr. Pavan Chakroborty
Assistant Professor
Indian Institute of Information Technology, Allahabad (Amethi Campus). India

ABSTRACT
GDDK is a tool to help users build a driver in an organized and a lesser burdened way than how they do that natively. It helps by framing a procedure that comes common out of developing various drivers. The tools contains all function definitions of the kernel source, but the user is expected know how to use those functions in the GDDK and how to use them to solve his purpose, as the tool is supposed to provide only a convenient way for developing the driver, and hence doesn't do any wonders by creating drivers itself. The tool offers a wizard for users which can help them to make a driver on preconfigured templates as well as a UI where she can choose from innumerable functionalities of the kernel to develop one of her own.

The toolkit should not be misjudged as an AI kind of framework where only a little information from the user can develop a driver for any device. User must know each and everything about the device, about the functionalities as well as mechanism of the device to develop a driver. The toolkit helps in a way that it saves user's time to develop a driver as well as it can help user to build it in a systematic approach so that useless ambiguities are avoided.

Categories and Subject Descriptors
D.4.0 [**Unix and Network Development**]: Device Drivers development for linux.

General Terms
Documentation, Design, Experimentation, Standardization, Theory.

Keywords
GDDK, DDK, Linux drivers. Device drivers.

1. INTRODUCTION
Currently devices drivers for Linux are all developed by understanding some details of the kernel and then developing a module which becomes a part of the kernel and acts as a driver. The process in itself requires a detailed amount of knowledge of kernel and its properties. Thus the driver code as a consequential outcome, becomes something which can be understood by only developers and not daily or end users who are Linux newbies. Moreover the code complexities, dispatch modules and kernel parameters in themselves are something which a non developer has to think twice about before developing a driver for herself. New devices are getting developed everyday out of which most are non commercial “home-made” devices. Now those with a lack of in depth kernel programming face a trouble developing drivers for their devices. It is for that purpose, this Linux Driver GDK comes into picture. This GDDK is proposed as a tool for end user to develop drivers for various devices where a user doesn't need to know hardcore programming as a prerequisite.

The GDDK proposes an architecture where user can very easily be able to retrieve various drivers informations preloaded like the major and minor numbers for devices, their mount points in /dev and /etc. Instead of writing a code a user can make use of a flowchart or various graphical symbols to create a logical representation of the driver code and the GDDK can develop or generate the corresponding c code for the driver. The modules which are the composition of the driver can be clubbed with the GDDK without compiling them individually. The complexity of the code for driver , thus doesn't need to be concerned of by the end user anymore since the API itself helps the user to to build a driver with nothing but a graphical tool.

2. DRIVER DEVELOPMENT : CURRENT SCENE
2.1 Classification of Drivers
2.1.1 Char Drivers:

These drivers are those which feature stream reading and writing and implement at least open, close, read and write system calls. It is quite similar to a simple file apart from the fact that one just cannot access all back and forth regions of a device through this sort of driver unlike file. This kind of driver allows only sequential operations.

2.1.2 Block Drivers:

These drivers provide a block access to the devices i.e. device can be communicated with a block of data not just a sequential stream access. Normally any number of bytes can be transferred by the driver to the device.

2.1.3 Network Drivers:

As the name suggests these drivers follow the network layer abstraction of the TCP-IP or OSI model where communication with the device is done in form of packets. These kind of drivers are mostly not mapped on file-system.

2.1.4 Class Driver:

These drivers are the one most popular and are presently used in nearly all devices now a days. These drivers hold the property

that they can operate a large number of different devices of a broadly similar type. The most common example is USB interface which is widely preferred in many devices.

2.2 Working of a Driver

Device drivers act as a medium through which the Linux API interacts with the devices. Multiple drivers can be associated with a single device as shown in the figure. Moreover, In Linux, drivers are actually loaded as modules into a running kernel. The kernel is the binding of element of devices and their software abstractions i.e drivers. So it is the first liability to take a peep into the kernel.

2.2.1 Sectioning the kernel:

The kernel is the core of an operating system. It acts as a layer of software over all pieces of hardware. But that is not the only limited functionality of a kernel i.e. to just act as an abstraction layer. It handles the requests of various processes and has to deal with interrupts as well as process management. When to assign resources and how much to be assigned and all other details are to taken care of by the kernel itself. So in other words the task of a kernel cannot be be designated by a single functionality. Although distinction between different kernel tasks cannot be marked precisely yet it can be categorized in these general terminologies:

a) Process Management – Control of processes, when to kill some process or to assign resources to some process is a functionality of the kernel.

b) File System – Kernel has to look over this software abstraction where a whole deal of information is managed in an architecture which we know as a file-system.

c) Memory Management – Virtual Memory management for processes on top of real memory is implemented in this functionality.

d) Device Management – To map all the devices and load their functionalities at runtime is the responsibility of this area.

e) Networking – Since networking operations are not specific to a purpose all the packets can be asynchronous so it is the responsibility of the kernel to manage them.

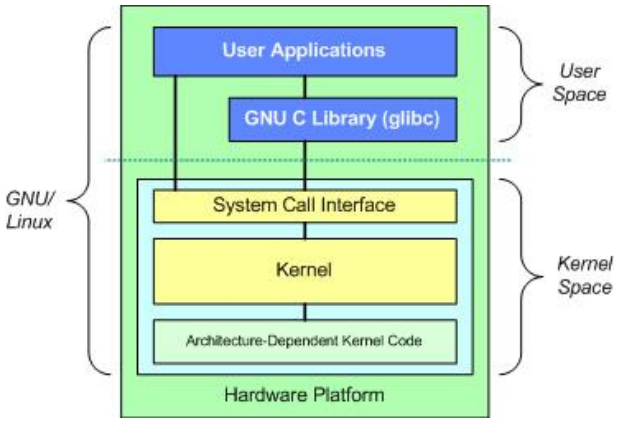

Illustration 1: Image Source: http://www.ibm.com/developerworks/linux/library/l-linux-

2.2.2 Visualizing the kernel:

To simplify things instead of making each and everything be familiar to kernel, it is better to show some familiarity to the user and some to kernel. It means that kernel can hide all the internal details of the working and provide user with something she understands. That's the reason it implements to different address spaces – User Space and Kernel Space. While a user can issue a request whether or not it should be granted is up to the kernel. So it can take that request to the kernel space and grant more privileges to that request and fulfill it. So apart form ease, security is also provided. The following diagram can represent this issue clearly:

2.2.3 Role of a Driver in the kernel:

To make a flexible driver for Linux it must hold the property of providing mechanism "what capabilities are to be provided" and not policy "how those capabilities can be used". Now apart from providing the software abstraction layer around the device a driver is also supposed to provide security. It might be possible that two different processes are trying two access the same device, in that case the driver must be able to handle the situation properly. Similarly there are other issues of buffer overloading, address space registration, concurrency etc. All these issues are to taken by the driver itself otherwise the device or the kernel or both may be prone to harm.

2.3 Making an actual Driver:

Now that we have seen all about a general driver we ought to know about the current method developers use to make a driver.

Before we take any further step, it is to be notified that the instructions(in terms of hex-codes) of the device need to be present with the user before-hand. These hex-codes refer to particular instructions of the device , according to the device specification of course! If the manual is not provided by the device manufacturers then this instructions set can be usually generated by the common procedure of Reverse Engineering, known to most of the driver developers.

Now that we have the instruction set ready, we need to know how a kernel interacts with a particular device. A kernel actually makes use of two different numbers to point to a device – MAJOR_NUMBER and MINOR_NUMBER. The former one tells about the driver being used for a particular device and the latter one actually points to the exact device. One can preview these numbers by typing "ls -l" in /dev directory. The two contiguous columns of decimal numbers are Major and Minor respectively.

For any category of drivers there is one common methodology used for implementation : OPEN, READ, WRITE, IOCTL, CLOSE. Lets discuss them one by one.

OPEN – In this portion of the code drivers needs to open the device for any thing like reading or writing from or to the device. It requires information about the file(device file we are talking about) which is held in inode and file descriptor.

READ & WRITE – These methods require similar kind of informations about the device i.e. file descriptor , pointer to a buffer where the data shall be first placed, the number of quanta

of data to be written and the offset of the file from where to start writing.

IOCTL – This is only used when special type of functionalities are required form the device. The ioctl system call helps communication with the device in a better way. As parameters it takes the file descriptor, request number and the data pointer in input.

CLOSE / RELEASE – This chunk of code takes all the resources away from the device (when not in need of course !). It deallocates anything OPEN allocated and shuts down the device on the last close.

These five major chunks of code are in every driver no matter what the genre of that driver is. These function calls are to be defined by the user and when it has been done, the file operations structure (where all the information about the device file to be delivered to the kernel is stored) is set according to these functions.

In Linux before anything else device registration is necessary which is done in the initialization function of the driver. So as soon as the driver runs the initialization function is executed in the beginning and the device is registered and along with it we initialize the file operations structure as well but we don't set the member values of the structure. As soon as the above functions are(READ , WRITE etc.) are all set we define them as the file operations member through pointer passing.

Now that all the driver code is developed, its high time we look over the issues of implementing our driver. First things first, there exists no c code in this world which works on its own. It needs to be compiled. For this task, in the kernel versions beyond 2.4.x, a makefile needs to be written which contains some specifications about the kernel versions, source directory, the device files to be generated in /dev(Note this. This is done through a function named "mknod" which creates a task specific node in the /dev directory. After running the makefile the executable is generated. Then the comes the final part of implementation which is terned as inserting the driver into to the running kernel. Its the virtue of the kernel that it provides a pre-built function for this task which is "insmod". It takes the driver binary as input and then inserts it into the running kernel.

With this a driver is developed in its complete form apart from other details like version defining, exporting symbols etc(For further details, see the references in the end)

3. THE GDDK

Where does GDDK come into picture?

Now, knowing about a device driver, a user, specifically a device manufacturer, decides to make a driver for her device. She needs to (i) refer to a table of hex-codes which guides the information exchange between the kernel and the device; (ii) identify the header files (e.g. linux/module.h, sys/types.h) required; (iii) write the code of mandatory kernel functions (viz. open, read, write, release, IOCTL), and her own customized functions, structures or variables, if required; (iv) identify and initialize the members of file descriptors (which are actually structures); (v) initializes the driver by registering the device with the operating system; (vi) make the member functions of 'file_operations' structure point to its mandatory kernel functions definitions; (vii) compiling the final code after completion; and finally insert the module into the kernel. Now, let us look at how the GDDK helps in his procedure.

3.1Sequencing the whole procedure.

The toolkit provides an organized sequential procedure that maintains the integrity of the code it writes. It does so in the given following fashion.

(i) The wizard of the toolkit makes the final code readable by defining macros to these hex-codes belonging to the particular device.

(ii) It shows the hierarchal listing of the header files and their corresponding functions, hence enabling the user to easily identify the header files she requires for implementing the functions in her code. For example, if she needs kmalloc() function to be implemented in her code, she can easily know the library 'linux/slab.h', in which the specified function is defined.

(iii) It provides an editor window to write the code of various mandatory kernel functions, i.e. separate places to provide the code for read, write functions etc. This way the process of writing the code becomes convenient. Further, the toolkit provides specifications where the require routines are to be placed within these functions. This is done by placing comments in the current editor window displayed to the user.

```
return_value read ( struct file* fil_p,  struct inode* nod_p)

{

/************************** PLACE YOUR FILE DESCRIPTOR INFORMATION HERE
**************************/

 /************************** PLACE THE INITIALIZATION OF THE BUFFER AND THE FILE OFFSET VALUES HERE
**************************/

 /************************** PLACE THE INFORMATION TO BE SENT TO THE DEVICE HERE

************************/

 /**************************
...
 **************************/ }
```

Moreover, it provides an option to add new customized functions, structures and variables through a blank editor window (the ones without the comments mentioned above).

(iv) The toolkit asks for the values of the mandatory file descriptor members and gives an option to add others as well.

(v) Owing to the fixed nature of the initialization function, most of the code for this part is automatically generated, with a few amendments to be done by the user herself.

(vi) For this and the next parts of the procedure, the user has the time to sit and relax as all the work is automatically accomplished by the GDDK itself. It includes making the member functions of the 'file_operations' structure to point to their respective definitions, generating the makefile with options set by the user such as adding device nodes (e.g. 'mknode /dev/js0'),manipulating scripts etc., compiling and finally inserting the module into the running kernel. This result of the whole automation can be altered as well.

Hence, the DDK toolkit helps in developing a driver in a more convenient, easier and more importantly, in a lesser bug-prone fashion.

3.2 GDDK for learning

The toolkit provides a help section that can assist a beginner to write driver for her device by providing her with basic examples of char, block, network and class drivers. These basic examples will not only provide a way to get hands on for the toolkit , but also serve the purpose of laying the foundation for driver development.

3.3 Recommendation for Future Research

The further development on this path can be viewed as the implementation of the following

(i) to add more and more AI into the driver making process, comforting the user even more.

(ii) to add the capability in the tool to generate the distribution packages (.deb,.rpm etc)of the drivers developed, for various Linux distributions.

(iii) The lack of instruction set of the device might pose an additional requirement of availability of a non-linux system (Windows etc.) for the process of reverse engineering described. Somehow, this can be eliminated.

4. ACKNOWLEDGMENTS

Our thanks to our college Indian Institute of Information Technology, Allahabad, Amethi Campus (India) for providing the platform to do the research.

5. REFERENCES

[1] Alessandro Rubini. Book: Linux device drivers, 1998

[2] Corbet, Jonathan. Rubini, Alessandro. Kroah-Hartman,Greg. Book: Linux device drivers, 3rd Ed., 2005

[3] Kroah-Hartman, Greg.2001. Kernel korner: How to write a Linux usb device driver. Specialized Systems Consultants, Inc. Seattle, WA, USA, 1075-3583. http://portal.acm.org/citation.cfm?id=509852.509856&coll=portal&dl=ACM&CFID=51722142&CFTOKEN=94078685

[4] Matia, Fernando. Kernel Corner: Writing a Linux Driver. ACM Linux Journal, 22(April 1998), 1075-3583. http://portal.acm.org/citation.cfm?id=327338.327360&jmp=cit&coll=portal&dl=ACM&CFID=51722142&CFTOKEN=94078685#CIT

[5] Rusling D A, The Linux Kernel, http://www.tldp.org/LDP/tlk/tlk.html, 1999

[6] Tsegaye,Melekam. Foss,Richard . 2004.A comparison of the Linux and Windows device driver architectures. ACM SIGOPS Operating Systems Review . ACM New York, NY, USA. DOI http://doi.acm.org/10.1145/991130.991132

[7] Wang,Shaojie. Malik, Sharad.Bergamaschi,Reinaldo A. 2003. Modeling and Integration of Peripheral Devices in Embedded Systems. In Design, Automation, and Test in Europe, Proceedings of the conference on Design, Automation and Test in Europe. IEEE Computer Society Washington, DC, USA, 1530-1591, 0-7695-1870-2. http://portal.acm.org/citation.cfm?id=789083.1022717&jmp=cit&coll=portal&dl=ACM&CFID=51722142&CFTOKEN=94078685#CIT