A Simulation Framework for Device Driver Development

Yan Shoumeng Zhou Xingshe
Dept. of Computer Science & Engineering
Northwestern Polytechnical University, Post Code 710072
Xi'an, China

ABSTRACT

Traditional development method of device driver becomes more and more unacceptable along with the increasingly deeper application of special devices. With a view to shorten the development cycle, this paper analyzes the traditional development model of such system, and presents a new co-development model based on device simulation. It gives the detailed design of device simulation. It also provides the implementation detail of related tools for easier simulation deployment. The simulation mechanism and the simulation tools together form a simulation framework for device driver development. The simulation process using the framework is discussed and an example application is presented.

INTRODUCTION

Proprietary I/O devices are often involved in many computer application systems. In such scenarios, Developers have to develop drivers for these devices to utilize them under some operating system. Traditionally, development of device driver can be undertaken only after hardware design finishes, for the writing and debugging process involves many interactions with real hardware. However, with the increasing competition among corporations, shorter time to market is important for a successful production. To shorten the development cycle of device driver, this paper proposes that using a simulation device as the target of driver's operations.

Previous works on device simulation focus on how to support development of application level software. VxSim[1], a component for platform simulation in VxWorks Tornado environment, support application software which requiring very simple interactions with hardware, but is not suitable for software with extensive device accesses, especially device drivers. A simulation development method for device driver is proposed in paper [2], which simulates all system routines in an application and compile and link the driver source code with the application together. This method can only make some testing work on device driver, but cannot support the simulation of the running of the entire application system that based on the device driver.

This paper presents a device simulation mechanism based on an abstraction model of I/O devices. Then a simulation framework for device driver development is discussed. This framework can effectively support writing and debugging of device driver without the need of hardware existence. For the running of application software is relying on the underlying device driver, it is natural that this framework also supports the development of the entire system without requiring hardware involvement.

DESIGN OF DEVICE SIMULATOR

According to analyses of various devices, we abstract I/O device as follows:

HW ::= (REG，IRQ，LOG)

Among above formulation, REG denotes the interface registers of device; IRQ denotes the interrupt request number occupied by the device; LOG denotes the action logic, i.e. what action device should take under certain conditions. So, if we can simulate the three elements of above device abstraction model properly in a software module, we can say we have simulated the device successfully.

The software module, which simulates the three basic elements of real hardware, is called device simulator. In essence, it is also a driver module residing in kernel address space.

We will describe the detailed simulation mechanisms for the three elements in the following of this section. Basically, the REG is simulated by a kernel memory buffer; the IRQ is simulated by a variable; and the LOG is simulated by a user-provided switch-case routine, which is called by a kernel timer handler periodically. We will also give the primary data structure of device simulator.

(1) Simulation of Registers

It is well known that address space of a process consists of user space（accessible in user mode）and system space（accessible in kernel mode）. System space is used to load the kernel components of operating system and is not accessible to application level. The space is made up of non-paged memory areas that are always visible whether the user space is active or not. Device drivers, of which both code and data reside in the system space, have read and write access rights to the entire system space. Therefore, if a driver can know the variables address of other drivers, it should be able to access the variables.

Based on this point, the registers of device can be simulated simply by an unsigned character array or some memory allocated from non-paged memory area. Considering flexibility requirement, we choose the second method, i.e. dynamically allocating certain amounts of non-paged memory. This method enables us to change the number of register conveniently with the change of hardware design. Herein, a critical problem to solve is that other drivers, i.e. real drivers of devices, how to know the address of these simulated registers. Our solution is to introduce an intermediate medium. When the device simulator module is loaded, the head address of registers is decided. Then we can log the address into the intermediate medium, through which the real driver can get the address information of simulated registers and can access them thereafter.

(2) Simulation of IRQ

The interrupt process mechanism in X86 architecture can be described as follows:

- Interrupt signal is sent from peripheral to interrupt controller chip or module
- Interrupt chip transform the signal into data signal and send it to CPU
- CPU search the IDT and get the vector entry
- Jump to certain interrupt service routine

Based on this knowledge, we can use a variable to simulate the IRQ and call **int n** directly to simulate the interrupt triggering. Herein, **n** is not the variable denoting IRQ but the index in IDT transformed from IRQ. The transformation, which is fulfilled by interrupt controller in real environment, can be different in different OS platforms. For example, it is to simply plus 0x20 on LINUX, but on Windows it involve special system call to complete this transformation.

Otherwise, we in fact can simulate the interrupt controller using an independent driver which checks whether interrupt should be triggered or not periodically and simulates the sending of interrupt to CPU. But for simplicity, we trigger the interrupt in each device simulator module directly.

(3) Simulation of Action Logic

The simulation of action logic of device under various conditions is a difficult point in the simulation mechanism. For real hardware, the detection of condition changes is accomplished through circuit behavior. If want to simulate it by software, we must have a mechanism to detect the changes of condition actively and take suitable process according to conditions. Therefore, a kernel timer is adopted to detect current condition and take certain actions periodically. The timer is started in the device simulator module. And when the period expires, a "case-switch" function is called. It deserves to note that the granularity of timer is related to the simulation fidelity. Thus, the practical value of the period should be decided according to practical condition, or it should be able to adapt in the simulation process. Considering that the driver developer of real hardware is the most familiar to hardware specification, we propose that they should present the "case-switch" function. Herein, we only provide the reference method to implement the function. A good way is to introduce the finite state machine (FSM) theory into describing the device behavior. The interface of device can be viewed as a FSM, states of which are the snapshot of the interface registers at certain time and inputs of which are writing values from real device driver and device control panel (will be described later in detail). The description based on FSM can be transformed by a special tool into practical program code as the simulation implementation of action logic.

(4) Simulation of Register Access

The register access routines provided by system should be replaced so that the accesses in real device driver to registers can be redirected to the registers array of device simulator. In the simulation implementation of register access routine, a access action is divided into two steps: first getting the address of the simulated register array, then taking the real memory access action. Concurrent access to registers is possible for the support for online edit capability, which involves access to registers in device simulator. Therefore, the simulated registers are critical resources and system mutex primitives should enclose the register access routines. The rewritten register access routines will be provided as a head file, and the real device driver project, to replace the system register access routines, just need to include that file. When the real hardware is finished later, we just need to discard the head file and rebuild the project to get the final device driver.

(5) Primary Data Structures

The implementation of device simulator is object based. We encapsulate the attributes and behaviors of device simulator in a structure. Herein, we adopt C language for C is the common language of driver development. Each device simulator maintain the

following data structure:

```
typedef struct tagSimHardware{
unsigned char *m_pRegister;//address of registers
unsigned int m_uRegNum;//number of register
unsigned int m_uIrq;//IRQ number
unsigned int m_uPeriod;//timer period
KTIMER m_Timer;//Timer
////////pointer of member function
bool (*init) PSIMHARDWARE this, unsigned int uRegNum, unsigned int nIrq, unsigned int Period);//initialize hardware
void (*ActionLogic)( PSIMHARDWARE this);
//action logic as the callback function of timer
void (*SetIrq)( PSIMHARDWARE this);
//change the IRQ
BOOL (*SetPeriod)( PSIMHARDWARE this, unsigned int uPeriod);//change timer period
void (*SaveConfig)( PSIMHARDWARE this);
//save information of device simulator to medium
void (*GenerateInt)( PSIMHARDWARE this);//simulation of interrupt triggering
} SIMHARDWARE,* PSIMHARDWARE;
```

SIMULATION TOOLS

This section discusses the design of aiding tools for easy simulation deployment

(1) Device Control Panel

In many cases, driver developers expect for some control capabilities over device simulators. Demands can be list as follows.

- The design of hardware can be changed frequently. The device simulator module should reflect the latest change timely and easily.
- The behavior of real device is usually difficult to control to reoccur some cases. But this capability is required in driver debugging. Thus, it would be better if device simulator module can provide functions such as register save and restoration, register edit, and interrupt generation.
- Otherwise, in the different phases of driver development, what matters is different. This demands the precision of simulator can be adjust easily.

The device control panel, an application level software that interfaces with the kernel mode device simulator, can meet these demands. Using this tool, developers can generate an interrupt manually; can change the IRQ of certain device simulator; can display and edit the registers; and can adjust the precision of simulation. In implementation, the device control panel is a user interface to the underlying device simulator and provides a channel to control the behavior of simulator.

Device simulator processes the control command issued from device control panel in IOCTL file operation. It first get a unique control command according to IOCTL number, then call functions such as GenerateInt, SetIrq, SetPeriod, SaveConfig to fulfill the required operations.

(2) Simulator Auto-generator

Major implementation parts of device simulator are definite, which make it possible to provide a auto-generator of device simulator. We have designed a tool like the MS Visual Studio AppWizard, by which developers only need to provide a few indefinite aspects of implementation through friendly user interface. The tool will integrate the user input and the predefined simulator driver template and then produce a specific device simulator. Through using this tool, the difficulty of simulator development is lowered greatly, and the development cycle is shortened.

SIMULATION PROCESS IN THE SIMULATION FRAMEWORK

This section describes the entire simulation process in the simulation framework.

Firstly, the developer should generate a device simulator according to his device specification using the simulator auto-generator. Secondly, he should load the simulator and the real device driver into system. Now, the upper layer application and the device driver can run just as if there is a real underlying device. The developer can easily produce the exceptional device condition through manipulating the simulator's registers and has exceptional code path of the target device driver tested sufficiently. If the device specification changes, the developer can also easily modify the simulator through deice control panel to reflect the change timely.

Fig.1 gives the interactions among the relative components in this simulation framework.

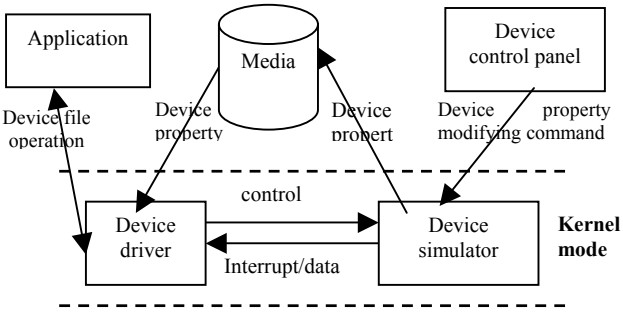

Fig. 1 Interaction in simulation process

APPLICATIONS

This section gives an application example of this simulation framework. By using this framework, the device drivers in this application have been sufficiently programmed and tested before real hardware became available. And thus, the development period is greatly shortened.

The application system is a data acquisition system based on airborne computer. In the computer, there is a smart communication card, which is responsible to receive data from eight data channels. The application software is responsible to fetch data from the card and save them to disk. At the same time, it also draw the data change curve on the screen and give an alarm when exception is detected. Components of the system are presented as fig 2.

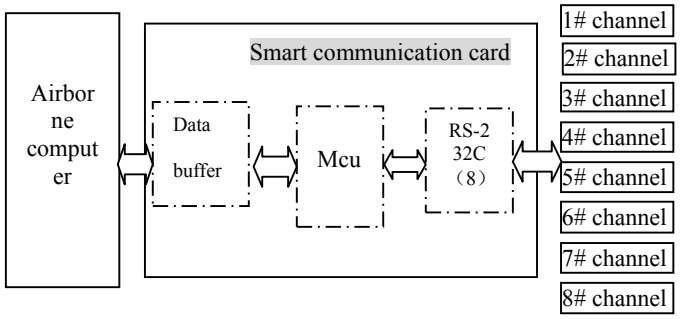

Fig. 2 Components of application system

Our task is to develop application software and device driver for the smart communication card. Because the smart communication card had not been completed, we can't make software programming and testing in detail. The overall development process is delayed for a time. Later, we adopt the simulation framework proposed by this paper and alleviate this problem to a great extent. Before hardware design is finished, we have developed and tested the device driver and application software sufficiently. Thus, the development cycle is shortend. Fig. 3 gives the simulation running of the application software without real hardware involved.

CONCLUSIONS

We have presented a driver development method based on simulation. The device simulation mechanism, the device control panel and the simulator auto-generator have formed a simulation framework. Currently, this framework has been integrated into an IDE for CC-LINUX, in which it is used to support simulation debugging and simulation running of embedded system. The practices have proved that this framework can shorten the development cycle efficiently. What should be done next is how to simplify the deployment of action logic. We hope to provide a tool, which can help developers describe the action logic in FSM and generate code automatically for action logic based on the description. This will be the direction of further study.

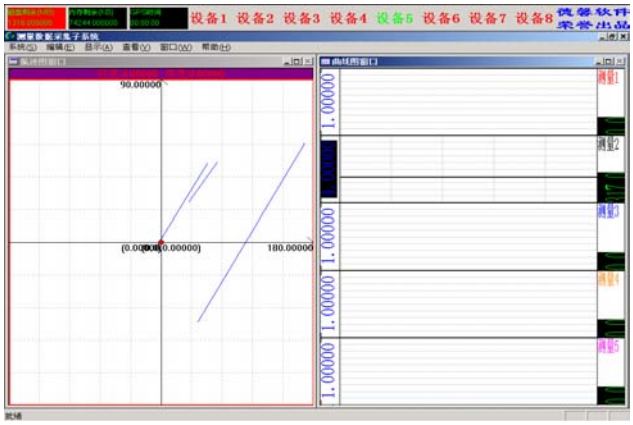

Fig. 3 Simulation running of application

REFERENCES

[1]VxWorks Tornado Online Help.

[2]Eddy Quicksall and Ken Gibson. Simulation and Device-Driver Development, Dr. Dobb's Journal, 1997 (1).

[3]Development Techniques for Using Simulation to Remove Risk in Software/Hardware Integration, http://www.redhat.com/support/wpapers/cygnus/cygnus_risk/development.html.

[4]Jiang weihua and Yu huqun. A Co-simulation method for embedded system design(in Chinese), Journal of East China University of Science and Technology, Vol. 27 No.5:475-479.

[5]Liu mouyong and Ge jiguang. Implementation of Virtual Hadware in Operatiing System Design (in Chinese) [J]，Journal of Zhejiang University，Vol. 33 No.4:351-355.