Abstraction of Device Drivers and Inputs of the Device Driver Generation System for UNIX-like Operation Systems

Tetsuro Katayama  
Graduate School of Info. Sci.,  
Nara Inst. of Sci. and Tech.  
8916-5 Takayama Ikoma,  
Nara 630-0101, Japan.  
kat@is.aist-nara.ac.jp

Keizo Saisho  
Dept. of Reliability-based info. Sys. Eng.,  
Faculty of Eng., Kagawa Univ.  
2217-20 Shinmachi, Hayashi,  
Takamatsu 761-0396, Japan.  
sai@eng.kagawa-u.ac.jp

Akira Fukuda  
Graduate School of Info. Sci.,  
Nara Inst. of Sci. and Tech.  
8916-5 Takayama Ikoma,  
Nara 630-0101, Japan.  
fukuda@is.aist-nara.ac.jp

Abstract

Writing device drivers spends much time and makes efforts because it needs knowledge of the target device and operating system (OS). In order to lighten the burden, the authors have proposed the device driver generation system before. The system generates a source code of a device driver from three inputs: the device driver specification, the OS dependent specification, and the device dependent specification. Generating device drivers by using the system, however, is not always effective because the burden in describing the device dependent specification, which is one of the inputs, is nearly as same as the traditional method. In this paper, device drivers are abstracted again, and each input is defined afresh, and then more reduction of the burden is aimed at. As an example of the description, an interrupt handler of a network device, FreeBSD and Linux as the target OS, and Etherlink XL as the target device are chosen.

keywords: operating system (OS), device driver, network, ethernet, interrupt handler, UNIX-like OS.

1 Introduction

Writing device drivers is one of the most difficult tasks to develop or port operating systems (OSs)[1, 2]. Some of the reasons are as follows:

- Programmers of device driver must know information about hardware such as specifications of devices and carefully describe complex parts such as timing control.
- When two devices have different chips (controllers) even if they offer the same services, the programmers must write two different device drivers for each of them.
- If we change an OS but use the same devices, we need to have the device drivers for new one.

As internet is grown and multi-media is progressed, various devices would be developed. Moreover, as many embedded systems are developed, the markets request to write device drivers more rapidly. It is a more serious problem to spend much time and make efforts to write the device drivers[3]. We should urgently cope with reducing the burden.

We have proposed the device driver generation system before[4, 5]. The system generates a source code of a device driver from three inputs: the device driver specification, the OS dependent specification, and the device dependent specification. The generating device drivers by using the system, however, is not always effective because the burden in describing the device dependent specification, which is one of the inputs, is nearly as same as the traditional method.

In this paper, device drivers are abstracted again, and each input is defined afresh, and then more reduction of the burden is aimed at. In section 2, we show the device driver generation system and describe the inputs for the system. In section 3, we describe an example of
the generation. We choose an interrupt handler of a network device, FreeBSD[6] and Linux[7] as the target OS, and Etherlink XL (3Com Co.) as the target device. In section 4, we discuss and evaluate our proposed method.

2 Device Driver Generation System

In this section, we introduce the device driver generation system[4, 5].

A device driver is a program to control a device by an OS. The driver is written corresponding to each OS and device (see Figure 1). We need to write many device drivers.

Device drivers exist to see devices virtually from OSs. The function of the device drivers is determined according to a type of devices. For example, in Etherlink XL and DE500A, which are representative ethernet cards, different controller chips are used, but the role as an ethernet card is the same, and the function to write in the device drivers is the same.

OSs control devices through the device drivers and send/receive data. If OSs differ, each OS has its own data structures or data types to store data. If devices differ, each device has its own interfaces, timing or data type to send/receive data.

Hence, a device driver is a program to convert data into a format corresponding to each OS and device. For example, in ethernet cards, data to send/receive are ethernet frames, data structures to store them in FreeBSD[6] and Linux[7] are mbuf structure and skbuff structure, respectively. A device driver can be abstracted to three parts as follows:

- a part to send/receive data between an OS and a device,
- an interface to control data from/to the OS, and
- an interface to control data from/to the device.

In order to get rid of dependent parts on OSs or devices in device drivers, in the present we use frequently the conditional compilation in C programming language and write device drivers. We write source codes corresponding to all OSs and devices in a device driver to deal with multiple OSs and devices. Different codes in conditions are executed. However, in this method, the source codes would be more complex and it is difficult for other to understand or modify them.

We have proposed a method abstracting a device driver in writing so that it can correspond to multiple OSs and devices. Device drivers are generated by describing three parts as mentioned above. Three parts are defined as the device driver specification, the OS dependent specification, and the device dependent specification, respectively.

- **device driver specification**
  It shows operations of the device. It is described that functions, data structure, and code in the functions which the generated device driver uses.

- **OS dependent specification**
  It shows dependent parts on the OS. It is described that names, arguments, return values of device driver interfaces which the OS provides. The device driver interfaces are functions the kernel calls.

- **device dependent specification**
  It shows dependent parts on the device. It is described that dependent parts on the hardware in functions of the device described in the device driver specification.

In order to write device drivers, we must consider a target CPU and I/O bus also. In this paper, we focus on interfaces of OSs and the way to handle data from/to devices which are fundamental elements in generating device drivers. Differences of CPUs or I/O buses are not considered.

Figure 2 shows environment by using three specifications. When we change a device to a new one which offers the same services, we have to rewrite only the part depended on a chip of the new one. That is, the OS dependent specification can be reused. Similarly, when we change an OS, the device dependent specification can be reused. As a result, we can lighten the burden in developing device drivers and solve the problems which are that the source codes of device drivers would be more complex and it is difficult for others to understand or modify them. Figure 3 shows an outline of the device driver generation system.

However, the device dependent specification, which is one of the inputs of the system, accounts for the greater
Figure 2. Device driver development with three specifications

part of the amount of the description when three specifications are described according to inputs we have defined before[5]. It is difficult to divide between codes to access each OS's own structures or functions and codes to access each device's own structures. We allow such codes to be described in the device dependent specification. Hence, a person to describe the device dependent specification needs to knowledge of a target OS. As a result, the generating device drivers by using the system is not always effective because the burden in describing the device dependent specification is nearly as same as the traditional method.

In this paper, the details of each specification are defined afresh. Especially, both of the OS dependent specification and the device dependent specification are defined more minutely. The knowledge to describe each specification is restricted and then more reduction of the burden in writing device drivers is aimed at. If multiple persons can describe separately each specification in Figure 2, we can disperse the burden in writing device drivers and develop them effectively.

The design policy of three specifications in this paper is the following:

- **device driver specification**
  It shows a template to define a device driver. Kinds of data to use functions of a device and control flows handling the data are described. The functions are translated into actual codes in the other specifications. It can be described in response to each kind of devices.

- **OS dependent specification**
  It shows dependent parts on an OS. The way to send data from OS to the devices and device driver interfaces in calling the device drivers from the OS are described. It can be described in response to each OS.

- **device dependent specification**
  It shows dependent parts on a device. Interfaces, timing, and data types in handling data to/from the device are described. It can be described in response to each device.

The next section describes three inputs of the device driver generation system by giving an actual example.

3 Inputs of the System and an Example

In this paper, as an example of the description, we choose FreeBSD[6] and Linux[7] as the target OS. The both are representative UNIX-like OSs, their source codes are open, and the source codes of the device drivers can be referred to.

We choose a network device as the target device and ethernet as the interface of network. It is the most popular and the burden of writing its device driver is large because a period of time to develop a new device is short. We choose Etherlink XL (3Com Co.) which is a representative ethernet card of PCI (Peripheral Component Interface).

We write an interrupt handler of a network device and explain how to describe each input of the device driver generation system.

3.1 Device Driver Specification

In device driver specification, kinds of data to use functions in a device and control flows handling the data are described. The functions are translated into actual codes in the other specifications. It can be described in response to each kind of devices.

Figure 4 shows a control flow of an interrupt handler. Its processing is divided into steps such as getting
Figure 4. Overview of an interrupt handler

information, forbidding interrupts, getting states, and permitting interrupts of the device.

We describe separately two parts used in device driver specification as follows.

- OS dependent parts
  - codes to handle each OS’s own data
  - codes to handle the data used in the kernel
  - codes using arguments of device driver interfaces
  - codes using system calls

- Device dependent parts
  - codes to access devices directly
  - codes to handle local and global variables

We can describe the OS dependent codes to OS dependent specification and the device dependent codes to device dependent specification by dividing functions in a device driver into two parts as above. In device driver specification, we must describe control flows in detail so that we can divide them into two parts.

Figure 5 shows a part of a device driver specification of an interrupt handler. Device driver specification consists of control statements and function calls. They are described according to grammar extended C programming language for the device driver generation system. The contents of a function are written between #begin and #end. The name of the function is written just behind #begin.

In function calls described in device driver specification, to distinguish kinds of the function, a label to indicate the kinds is appended just before the calls. The label and its meaning are the following.

// ---------- Interrupt Handler ----------
#begin _intr
    [proc]_get_dev_str(%struct_info),%buf);
    [cmd]_disable_intr(%iodr);

    for(;;)
    {
        [cmd]_get_status(%status);
        if([cmd]_break(%status))
            break;
        if([con]_up_complete(%status))
            [sub]_up_complete(%arg),%status,%iodr);
        if([con]_down_complete(%status))
            [sub]_down_complete(%arg),%status,%iodr);
        if([con]_tx_complete(%status))
            [sub]_tx_complete(%arg),%status,%iodr);
        if([con]_adfail(%status))
            [sub]_adfail(%arg),%status);
        [proc]_state(%status);
    }
    [cmd]_enable_intr(%iodr);
    [proc]_start_rest(%devinfo);
#end

#begin _tx_complete
    [proc]_tx_complete(%arg),%devinfo,%iodr);
    [cmd]_tx_complete(%arg),%devinfo,%iodr);
#end

#begin _up_complete
    %<name>_txeof(%arg);
    [cmd]_up_complete(%arg),%devinfo,%iodr);
#end

#begin _down_complete
    %<name>_txeof(%arg);
    [cmd]_down_complete(%arg),%devinfo,%iodr);
#end

#begin _adfail
    [proc]_adfail(%arg),%status);
    [cmd]_adfail(%arg),%status);
#end

Figure 5. A part of a device driver specification of an interrupt handler

- [proc] — a processing belonging to OS dependent specification
- [cmd] — a processing belonging to device dependent specification
- [con] — a condition for a branch belonging to device dependent specification
- [sub] — a subroutine in the device driver specification
- % — a label used to call directly other functions belonging to device driver interfaces and functions written directly in C programming language

Each label also exists to express which specifications a processing described in. Even if it has the same names
except for the label, it expresses a different processing. A prefix expressing the name of a device driver is added in the label %proc. For example, the prefix for the device driver of FreeBSD in Etherlink XL is %xl.

The name of arguments used in function calls is translated by referring to OS dependent specification.

### 3.2 OS dependent specification

In OS dependent specification, device driver interfaces which a OS provides, roles of their functions, variables used in them, and codes of the functions appended the label %proc in device driver specification are described. It can be described in response to each OS.

In FreeBSD, information of the device in interrupt handlers corresponds to inet structures and softc structures included in the inet structures. The softc structures store each device’s own data. In Linux, device structures have information of the device.

Figure 6 and Figure 7 show a part of OS dependent specifications of an interrupt handler for FreeBSD and Linux, respectively. The contents of a processing in OS dependent specification are written between #begin and #end. The contents are divided into five parts as follows.

- **function** — It expresses the contents of a processing in this function. It is transformed as a comment statement in a source code generated by the device driver generation system.

- **prototype** — It declares prototypes of device driver interfaces and establishes arguments and return values. The name of the interfaces and the arguments differ in FreeBSD and Linux (see Figure 6 and Figure 7).

- **variable** — It describes local variables used in this function. According to the specification of C language, it is generated just behind the beginning of the function by the system. In FreeBSD and Linux, iodadr and status are the same name and meaning, but the other variables differ (see Figure 6 and Figure 7).

- **transform** — It gives another name to each variable such as local variable, global variable, and argument used in this function. The names of variables are changed according to different data structures used in each OS or device. It is introduced to describe three specifications without changing the names of variables and used consistently in the specifications. For example, in Figure 6 structinfo corresponds to buf. The specifications use the name as structinfo in the case of handling.

```c
#define INTERRUPT
#define function
#define prototype
#define variable
#define transform
#define code

```

```
#define INTERRUPT
#define function
#define prototype
#define variable
#define transform
#define code

```

```c
#define INTERRUPT
#define function
#define prototype
#define variable
#define transform
#define code
```

Figure 6. A part of an OS dependent specification of an interrupt handler for FreeBSD

buf. That is, structinfo is transformed as buf in a source code generated by the system.

- **code** — It is described codes depending on OS.

In the part of %proc, each device’s own name is put. The name is given as an argument in the starting of the device driver generation system. In the part of the code, %proc of the head in each name of the function is omitted.

They are described according to grammar extended C programming language for the device driver generation system. Especially, function calls, operations, and control statements are used. The label %proc, which is used in the device driver specification, is appended to the functions described in OS dependent specification. The other functions without the label signify that they call
#begin INTERRUPT
%begin function
The interrupt handler does all of the Rx thread
work and clean up after the Tx thread.
%end
%begin prototype
static void %<name>_interrupt(irq,dev_id,regs)
  int irq;
  void *dev_id;
  struct pt_regs *regs;
%end
%begin variable
struct device *dev;
struct vortex_private *vp;
long iocdr;
int status;
%end
%begin transform
structinfo dev_id;
arg dev;
devinfo dev;
iocdr iocdr;
status status;
%end
%begin code
[proc].get_dev_str(structinfo,arg)
{
  devinfo = (device*)structinfo;
  arg = (struct %private *)devinfo->priv;
  iocdr = device->base_addr;
}
[proc].rx_complete(arg)
{
  %rx(arg);
%
%end

### Interrupt ####

```c
#include disable_intr(iocdr)
{
  outv( iocdr + XL_COMMAND, XL_CMD_INTR_END);
}
#include get_status(status)
{
  status = inv( iocdr + XL_STATUS);
}
#include break(status)
{
  return ( status & XL_INTRS ) == 0;
}
#include up_complete(status)
{
  return ( status & XL_STAT_UP_COMPLETE );
}
#include down_complete(status)
{
  return ( status & XL_STAT.DOWN_COMPLETE );
}
#include tx_complete(status)
{
  return ( status & XL_STAT.Tx_COMPLETE );
}
#include adfail(status)
{
  return ( status & XL_STAT.ADFAIL);
}
#include enable_intr(iocdr)
{
  outv( iocdr + XL_COMMAND, XL_CMD_INTR_ENABLE);
}
#include tx_complete(iocdr)
{
  outv( iocdr + XL_COMMAND, XL_CMD_INTR_ACK);
}
#include up_complete(iocdr)
{
  outv( iocdr + XL_COMMAND, XL_CMD_INTR_ACK);
}
```

Figure 7. A part of an OS dependent specification of an interrupt handler for Linux

its name of the functions. They correspond to system calls in kernel.

### 3.3 Device Dependent Specification

In device dependent specification, interfaces, timing, and data types in handling data to/from the devices are described. It can be described in response to each device.

Functions described in device dependent specification can be divided into two types. They are a processing handling variables or I/O port and a processing returning states or conditions to control statements in device driver specification. The former is appended the label [cmd], and the latter is appended the label [com].

Figure 8 shows a part of a device dependent specification of an interrupt handler for Etherlink XL. In device driver specification, we can write only each device's own codes. The device driver generation system does in-line expansion of the codes depending a device to a

```c
// ### Interrupt ###

```

Figure 8. A part of a device dependent specification of an interrupt handler for Etherlink XL

generated source code. Hence, the overhead which the abstraction by dividing into three specification causes can be prevented and the run time performance can be maintained.

The codes are described in device driver specification, like the other specifications, according to grammar extended C programming language for the device driver generation system. Especially, function calls, operations, and control statements are used. No label can be appended to function calls because they are independent of OSs.
4 Discussion and Evaluation

In this section, we describe portability and availability of our method and compare it with I₂O (Intelligent Input Output) [8].

4.1 Portability

In this paper, the details of each specification are defined afresh. Especially, both of the OS dependent specification and the device dependent specification have been defined more minutely. The inputs for the device driver generation system are described on the result.

As an example of the generation, we choose an interrupt handler of a network device, FreeBSD and Linux as the target OS, and Etherlink XL as the target device. Referring to the inputs and combining them, interrupt handlers are generated. We have generated source codes of the interrupt handlers for FreeBSD and Linux from the same device dependent specification and executed them correctly.

For example, Figure 9 shows a part of the generated source code of an interrupt handler of Etherlink XL for FreeBSD. The device dependent specification is portable because the device drivers can be generated for both OS without changing the device dependent specification.

4.2 Availability

In our defined method before [5], the burden in writing device drivers focuses on describing device dependent specification because it accounts for the greater part of the amount of the description. In this paper, the details of each specification are defined afresh. As a result, the ratio of the amount of the description in three specifications becomes more uniform and the burden in writing device drivers are dispersed more fairly to developers on the OS side (maker) and ones on the device side (maker).

Table 1 shows the number of lines of each specifications which we need to describe about Etherlink XL in our previous and current method.

The burden cannot be simply compared by the number of lines but it can be adopted as one of standards. In the previous method, it of the device dependent specification is about 85 times as large as one of the OS dependent specification and the burden focuses on describing the device dependent specification. In the current method, there is little difference of the number of lines between three specifications. As a result, the burden in writing device drivers are dispersed more fairly.

Moreover, even if persons which describe OS dependent specification do not know each device’s details or persons which describe device dependent specification do not know each OS’s details, they engage in developing device drivers. The knowledge of each person to describe each specification is restricted and the burden of the persons is lightened.

4.3 Comparison with I₂O

I₂O (Intelligent Input Output) SIG [8] has determined standard interface I₂O between OSs and devices. Under the specification of I₂O, a device driver is divided

Figure 9. A part of the generated source code of an interrupt handler of Etherlink XL for FreeBSD
Table 1. Comparison of Line Counts

<table>
<thead>
<tr>
<th>Specifications</th>
<th>previous</th>
<th>current</th>
</tr>
</thead>
<tbody>
<tr>
<td>device driver specification</td>
<td>14</td>
<td>58</td>
</tr>
<tr>
<td>OS dependent specification</td>
<td>2</td>
<td>53</td>
</tr>
<tr>
<td>device dependent specification</td>
<td>171</td>
<td>69</td>
</tr>
</tbody>
</table>

into three classes such as OSM(OS Specific Module) depending on an OS, HDM(Hardware Device Module) depending on a device, and Messenger sending or receiving packets between OSM and HDM. An OS can communicate the device with the same HDM even if the OS changes[9].

It, however, includes lower performance than usual. Especially, in rapid devices or real time system such the case will be a fatal problem. In our proposed system such the overhead in communication will not occur because device drivers are abstracted not at the target device or OS but in the generation.

5 Conclusion

We aim at lightening the burden in writing device drivers. We have proposed the device driver generation system before[4, 5]. The system generates a source code of a device driver from three inputs: the device driver specification, the OS dependent specification, and the device dependent specification. In this paper, device drivers were abstracted again, and each input was defined more minutely. As an example of the description of the each input, we chose an interrupt handler of a network device, FreeBSD and Linux as the target OS, and Etherlink XL (3Com Co.) as the target device.

Device driver specification shows a template to define a device driver. Functions, which are translated into actual codes in the other specifications, of a device and control flows handling the data are described. OS dependent specification shows dependent parts on an OS. Device driver interfaces in calling the device drivers from the OS and codes depending on the OS are described. Device dependent specification shows dependent parts on a device. Codes depending on the device such as interfaces, timing, and data types in handling data to/from the device are described.

We have generated source codes of the interrupt handlers for FreeBSD and Linux from the same device dependent specification and executed them correctly. The device dependent specification is portable because the device drivers can be generated for the both OS without changing the device dependent specification. Moreover, even if persons which describe OS dependent specification do not know each OS’s details, they engage in developing device drivers. The knowledge of each person to describe each specification is restricted and the burden of the persons is lightened.

Future issues are as follows:

- Extension to other OSs or devices.

In this paper, as an example, we chose an interrupt handler of a network device, FreeBSD and Linux as the target OS, and Etherlink XL as the target device. We need to adopt other OSs or devices and evaluate our method.

- Adaptation to other CPUs or I/O busses.

In this paper, we have focused on interfaces of OSs and the way to handle data from/to devices which are fundamental elements in generating device drivers. Differences of CPUs or I/O busses were not considered. In order to adapt our method to them, we plan to introduce CPU specification and I/O bus specification in addition to three specifications defined in this paper.

References


[7] Linux Online: http://www.linux.org/
