Proposal of a Support System for Device Driver Generation

Tetsuro Katayama, Keizo Saisho, and Akira Fukuda
Graduate School of Information Science,
Nara Institute of Science and Technology
8916-5 Takayama Ikoma, Nara 630-0101, JAPAN.
{kat,sai,fukuda}@is.aist-nara.ac.jp

Abstract

Writing device drivers is one of the most difficult tasks to develop or port operating systems (OSs). A device driver need to be described according to the target device and OS. And, in the description, programmers must know information about hardware such as specifications of devices and carefully describe complex parts such as timing control. This paper proposes a support system for device driver generation. The inputs for the system are device driver specification which shows operations of the device, OS dependent specification which shows dependent parts on the OS, and device dependent specification which shows dependent parts on the device. As an example, network devices is adopted. The inputs to generate device drivers by the proposed system are described.

keywords: operating systems, device drivers, network, ethernet, automatic generation.

1 Introduction

Writing device drivers is one of the most difficult tasks to develop or port operating systems (OSs). 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. It is a more serious problem to spend much time and make efforts to write the device drivers. We should urgently cope with reducing the burden. Few studies on generating device drivers or OSs themselves have been reported[1]–[3]. We have proposed the device driver generation system before[4]. The system, however, may not reduce the burden in writing device drivers because the abstraction of them are not always effective.

In this paper, we aim at lightening the burden. We propose a support system for device driver generation afresh and describe the inputs for the system. We choose FreeBSD[6] as the target OS and network devices as the target device.

2 Support System for Device Driver Generation

It is considered that inputs for the system are various elements such as functions, values, timing control, and so on. We attempt to simplify inputs for the system. The inputs for the system are determined as follows:

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

Figure 1 shows an outline of the system. In the usual method, we need to describe a code corresponding to each OS and device. In the proposed system, 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.

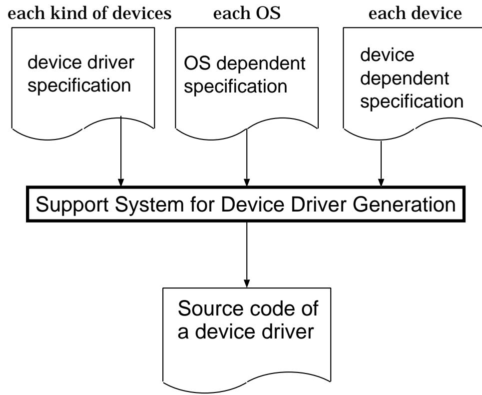

Figure 1. Outline of support system for device driver generation

Table 1. Device driver interfaces of the network device on FreeBSD.

function name	operation
probe	detection of devices
attach	registration of devices into OS
output	construction of ethernet flames from packets
start	processing to send ethernet flames
init	initialization of devices
ioctl	controls of devices
watchdog	timer
poll_recv	receiving processing with polling
poll_xmit	sending processing with polling
poll_intren	interrupting processing with polling
poll_slowinput	receiving processing against slow devices
done	finish of sending processing

3 Inputs of the System

As an example of the generation, we choose FreeBSD 2.2-stable (referred to as FreeBSD)[6] and network devices. The network devices are representative ethernet cards of PCI (Peripheral Component Interface) such as DE500A (DEC Co.), Etherlink XL (3Com Co.), and EtherPower II (SMC Inc.).

OSs operate devices through device driver interfaces. Devices is abstracted by the interfaces. Table 1 shows the device driver interfaces of the network devices on FreeBSD. We adopt 12 operations in Table 1 into the proposed system as the interfaces of the network devices.

To generate device drivers, the operations which the target device uses among 12 operations must be generated. Hence, we make inputs for the system to generate the operations. In this paper, we choose the start function, which is one of the operations, as an example and describe the inputs to generate the start function.

The start function sends data-link flames outputted from the output function to network through the interfaces. That is, it is considered that the start function sends ethernet flames to network.

```
/* Start function */
{
    /* Buffer operation */
        buf_op(ifp);

    /* Command */
        command(ifp);
}
```

Figure 2. Example of device driver specification for start function.

3.1 Device Driver Specification

In device driver specification, operations of the start function, which are inputs, outputs, and minute operations of the function, are described.

As the inputs, ethernet flames are given to the start function. We need to determine a data structure to store the ethernet flames. We determine mbuf structure, which is adopted in FreeBSD, as the data structure. The input and output of the start function are as follows:

- input — structure of mbuf
- output — data sent to network

The operations of the start function are as follows:

1. Buffer Operation — storing the data to a buffer
2. Command — Sending the data

The operations are described as a form of functions in C language[5]. Figure 2 shows an example of device driver specification.

3.2 OS Dependent Specification

In OS dependent specification, the device driver interfaces which the OS requires are described as follows:

- argument — ifnet structure (including mbuf structure)
- return value — static void
- name — <device_name>_start

The description of the interfaces are the same as prototype declaration in C language. The name of the interfaces is written in %<NAME>. It is given as an argument when the system starts. Figure 3 shows an example of OS dependent specification for FreeBSD.

3.3 Device Dependent Specification

In device dependent specification, the followings are needed.

```
static void %<NAME>(struct ifnet * const ifp)
```

Figure 3. Example of OS dependent specification for FreeBSD.

status(32bit)		
control(10bit)	data length2(11bit)	data length1(11bit)
data address1(32bit)		
data address2(32bit)		

Figure 4. Descriptor of DE500A (device dependent specification)

1. Buffer Operation — This operation is that the data to a buffer sent to ethernet controller are stored. The operation needs specifications as follows:
   - Structure of the descriptor (where descriptors construct a buffer and buffer consists of multiple descriptors)
   - **status** — status of descriptor
   - **control** — controls of descriptor
   - **next descriptor address** — physical address of the next descriptor when descriptor is constructed by lists.
   - **data address** — physical address of the ethernet flame sent to ethernet
   - **data length** — length of the flame sent to ethernet
   - **length** — length of the flame sent to ethernet excepting a header part
   - Length of the buffer
   - Symbol of start and end of the buffer
2. Command — This operation is that the data stored to a buffer send to network. The operation needs specifications as follows:
   - timing of issuing commands
   - value of commands
   - command registers
   - combination of commands

Figure 4 shows the descriptor of DE500A. Table 2 shows the data needed in constructing buffer. Table 3 shows transaction of transmitting on the ethernet controllers where $write(w, x)$ is writing value x in port w, $read(y)$ is reading a value from port y, and $wait(z)$ is waiting z times.

The description of the device dependent specification is almost the same as one in C language. It is described with the form "operation name:\{ contents of the operation \}". In the start function, 'Buffer Operation' and 'Command' correspond to the operation name. Figure 5 shows a part of an example of device dependent specification for DE500A.

Table 2. Input data for buffer construction on the ethernet controllers (device dependent specification)

	DE500A	EtherPowerII	Etherlink XL
status	0x80000000	0x8000	0x80000000 the last descriptor total length sent by one descriptor & 0x80000000
control	0x000c the first descriptor 0x008c the last descriptor 0x0300	data more than half remains in the buffer 0x14 otherwise 0x10	×
data address	per a descriptor 32bit x2	per a descriptor 32bit x1	per a descriptor 32bit x63
data length	11bit	16bit	32bit the last descriptor 0x80000000
ethernet frame length	×	×	○
next descriptor address	×	○	○ the last descriptor 0
buffer length	128	16	16

Table 3. Transaction of transmitting on the ethernet controllers (device dependent specification)

DE500A	EtherPowerII	Etherlink XL

write (ioport + 4 ,1)
write (ioport + 0x1c, 0x00000001)
	write (ioport,0x04)	write (ioport + 0x0e,0x3002)
		for(i=0; i< 1000 ;i++){
		wait(10ms);
		if(! read (ioport + 0x0E) & 0x1000)
		break;
		}
		if(read (ioport + 0x24)){
		appending lists of the new data to the current list of the buffer
		in the new data,
		status &= ~0x80000000
		}
		else{
		write (ioport + 0x24,
		the top address of the new buffer)
		}
		write (ioport + 0x0e, 0x3001)
		write (ioport + 0x0e, 0x0807)

4 Discussion and Evaluation

Referring to three specifications as mentioned above and combining them, the start function is generated. In this paper, we choose FreeBSD as the target OS and three network devices as the target device. The device drivers can be generated for the each device without changing the OS dependent specification for FreeBSD.

We compare our proposed system with the usual method. Table 4 shows the number of lines of the described source code. The number of the lines can be reduced in proposed system, because macro statements such as #ifdef are not needed to describe, namely, the specifications for the inputs are fixed.

Table 5 shows the number of lines of the generated source code. The number of the lines can be reduced in proposed system, because redundant parts such as comment statements and ones for debugging are not generated.

A significant performance degradation of the gener-

```
Buffer operation:\{
    tulip_softc_t * const sc = TULIP_IFP_TO_SOFTC(ifp);

      while (sc->tulip_if.if_snd.ifq_head != NULL){
         struct mbuf *m;

         /* Get next packet to send */
           (m) = (&(sc->tulip_if.if_snd))->ifq_head;
           if (m) {
               if (((&(sc->tulip_if.if_snd))->ifq_head
                     = (m)->m_nextpkt) == 0)
                     (&(sc->tulip_if.if_snd))->ifq_tail = 0;
               (m)->m_nextpkt = 0;
               (&(sc->tulip_if.if_snd))->ifq_len--;
           }
               ...
\}

Command:\{
      tulip_softc_t * const sc = TULIP_IFP_TO_SOFTC(ifp);
      out32(sc->iobase + 4, 1);
      out32(sc->iobase + 0x1c, 0x00000001);
\}
```

Figure 5. A part of an example of device dependent specification for DE500A.

Table 4. Amount of the described source code(lines)

	usual method	proposed system
DE500A	202	114
EtherPower II	99	92
Etherlink XL	212	187

ated code does not occur in comparison with one by the usual method. The mainly cause is that we use mbuf structure itself with used FreeBSD. When the system is applied to other OSs and they use another structure to send data, the structure must be transformed into mbuf structure. For example, in Linux[7], skbuff structure will be transformed. In such the case, this transformation may become overhead. This problem is a future issue.

I_2O(Intelligent Input Output) SIG[8] has determined standard interface I_2O between OSs and devices. Under the specification of I_2O, a device driver is divided into three classes such as OSM(OS Specific Module), HDM(Hardware Device Module), 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 in the generation.

5 Conclusion

In this paper, we aim at lightening the burden in writing device drivers. We proposed a support system for device driver generation and describe the inputs of the system. We chose FreeBSD and the drivers of network devices as an example. The system generates a source code of a device driver by giving three specifications such as device driver specification, OS dependent specification, and device dependent specification. We adapted the start function to the proposed system as an example. As a result, The each specification is not needed to rewrite when we describe it once, namely, it can be reused, and the burden in writing device drivers can decrease, and their productivity is improved.

Table 5. Amount of the generated source code(lines)

	usual method	proposed system
DE500A	202	101
EtherPower II	99	57
Etherlink XL	212	131

Future issues are as follows:

- Improvement of adaptability to the system.
  In this paper, we choose FreeBSD as the target OS and network devices as the target device. We need to adapt other OSs or devices to the system and evaluate it.
- Generation of drivers for devices written in HDL (Hardware Description Language).
  We need to describe three inputs for the system to generate a device driver. In future, we adopt devices written in HDL. We will construct the system to generate device drivers automatically from its description.

References

[1] X. Jia and M. Maekawa: "Operating System Kernel Automatic Construction," *Operating Systems Review*, Vol.29, No.3, pp.91-96, 1995.

[2] E. Tuggle: "Introduction to Device Driver Design," *Proc. 5th Annual Embedded Sys. Conf.*, Vol.2, pp.455-468, 1993.

[3] B. Maaref: "MMS Implementation Based on a Real-time Operating System Kernel," *Proc. IEEE Int'l Symp. on Industrial Electronics (ISIE'97)*, pp. 29-34, 1997.

[4] T. Katayama, K. Saisho, and A. Fukuda: "A Method for Automatic Generation of Device Drivers with a Formal Specification Language," *Proc. Int'l Workshop on Principles of Software Evolution (IWPSE98)*, pp.183-187 (1998).

[5] K. Yamashita, T. Katayama, K. Saisho, and A. Fukuda: "Definition of an Input Form for a Device Driver Generating System," *IPS Japan Sig Notes*, Vol.98, No.71, pp.61-68 (1998) (in Japanese).

[6] FreeBSD Inc: http://www.freebsd.org/

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

[8] I_2O SIG: http://www.i2osig.org/

[9] D. Wilner: "I_2O's OS evolves," *BYTE, Int. Ed., McGraw-Hill*, Vol.23, No.4, pp.47-48, 1998.