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.
{katsu,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.
/* Start function */
{
    /* Buffer operation */
    bufOp(ifp);
    /* Command */
    command(ifp);
}

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 frames are given to the start function. We need to determine a data structure to store the ethernet frames. 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.

Figure 1. Outline of support system for device driver generation

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

<table>
<thead>
<tr>
<th>function name</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>probe</td>
<td>detection of devices</td>
</tr>
<tr>
<td>attach</td>
<td>registration of devices into OS</td>
</tr>
<tr>
<td>output</td>
<td>construction of ethernet frames</td>
</tr>
<tr>
<td>start</td>
<td>processing to send ethernet frames</td>
</tr>
<tr>
<td>init</td>
<td>initialization of devices</td>
</tr>
<tr>
<td>ioctl</td>
<td>controls of devices</td>
</tr>
<tr>
<td>watchdog</td>
<td>timer</td>
</tr>
<tr>
<td>poll_recv</td>
<td>receiving processing with polling</td>
</tr>
<tr>
<td>poll_send</td>
<td>sending processing with polling</td>
</tr>
<tr>
<td>poll_interr</td>
<td>interrupting processing with polling</td>
</tr>
<tr>
<td>poll_sendint</td>
<td>receiving processing against slow devices</td>
</tr>
<tr>
<td>done</td>
<td>finish of sending processing</td>
</tr>
</tbody>
</table>

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 frames outputted from the output function to network through the interfaces. That is, it is considered that the start function sends ethernet frames to network.
static void %<NAME>(struct ifnet * const ip) 

Figure 3. Example of OS dependent specification for FreeBSD.

<table>
<thead>
<tr>
<th>status (32bit)</th>
<th>control (31bit)</th>
<th>data length 2 (11bit)</th>
<th>data length 1 (11bit)</th>
<th>data address 1 (32bit)</th>
<th>data address 2 (32bit)</th>
</tr>
</thead>
</table>

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 frame sent to ethernet
   - data length — length of the frame sent to ethernet
   - length — length of the frame 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.

<p>| Table 2. Input data for buffer construction on the ethernet controllers (device dependent specification) |
|---------------------------------------------------|---------------------------------------------------|---------------------------------------------------|</p>
<table>
<thead>
<tr>
<th>status</th>
<th>DE500A</th>
<th>EtherPowerII</th>
<th>Etherlink XL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
</tr>
<tr>
<td>control</td>
<td>0x0000</td>
<td>0x0000</td>
<td>0x0000</td>
</tr>
<tr>
<td>data address</td>
<td>per a descriptor</td>
<td>per a descriptor</td>
<td>per a descriptor</td>
</tr>
<tr>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
</tr>
<tr>
<td>data length</td>
<td>11bit</td>
<td>16bit</td>
<td>21bit</td>
</tr>
<tr>
<td>length</td>
<td>11bit</td>
<td>16bit</td>
<td>21bit</td>
</tr>
</tbody>
</table>

<p>| Table 3. Transaction of transmitting on the ethernet controllers (device dependent specification) |
|---------------------------------------------------|---------------------------------------------------|---------------------------------------------------|</p>
<table>
<thead>
<tr>
<th>DE500A</th>
<th>EtherPowerII</th>
<th>Etherlink XL</th>
</tr>
</thead>
<tbody>
<tr>
<td>write (input + 0x80 + 0x00)</td>
<td>write (input + 0x80)</td>
<td>write (input + 0x80)</td>
</tr>
<tr>
<td>write (input + 0x80)</td>
<td>write (input + 0x80)</td>
<td>write (input + 0x80)</td>
</tr>
<tr>
<td>write (input + 0x80)</td>
<td>write (input + 0x80)</td>
<td>write (input + 0x80)</td>
</tr>
<tr>
<td>write (input + 0x80)</td>
<td>write (input + 0x80)</td>
<td>write (input + 0x80)</td>
</tr>
</tbody>
</table>

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 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 soft.c  * const sc = TULIP_JFP_TO_SOFTC(ifp);

while ((sc->tulip_if.ifp.send ifq head != NULL)) {
  struct mbuf *m;

  /* Get next packet to send */
  (m) = (k(sc->tulip_if.ifp.send ifq head);
  if (m) {
    (m) = m->nextpkt = 0;
    if (((sc->tulip_if.ifp.send ifq head)
        = (m) -> m->nextpkt = 0;
        (m) -> m->nextpkt = 0;
        (k(sc->tulip_if.send ifq)) = (m) -> m->nextpkt = 0;
    }
  }...
}

Command:\{

tulip soft.c  * const sc = TULIP_JFP_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)

<table>
<thead>
<tr>
<th>DE500A</th>
<th>usual method</th>
<th>proposed system</th>
</tr>
</thead>
<tbody>
<tr>
<td>EtherPower II</td>
<td>101</td>
<td>99</td>
</tr>
<tr>
<td>EtherLink XL</td>
<td>121</td>
<td>212</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>DE500A</th>
<th>usual method</th>
<th>proposed system</th>
</tr>
</thead>
<tbody>
<tr>
<td>EtherPower II</td>
<td>101</td>
<td>99</td>
</tr>
<tr>
<td>EtherLink XL</td>
<td>121</td>
<td>212</td>
</tr>
</tbody>
</table>

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 specication is not needed to be used when we describe it once, namely, it can be reused, and the burden in writing device drivers can decrease, and their productivity is improved.

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

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

5 Conclusion

In this paper, we aim at lightening the burden in writing device drivers. We proposed a support system for