Prototype of the Device Driver Generation System for UNIX-like Operating Systems

Tetsuro Katayama
Faculty of Engineering,
Miyazaki University.
1-1 Gakuen, kibanadai-nishi,
Miyazaki 889-2192, Japan.
kat@cs.miyazaki-u.ac.jp

Keizo Saisho
Faculty of Engineering,
Kagawa University.
2217-20 Shinmachi, Hayashi,
Takamatsu 761-0396, Japan.
sai@eng.kagawa-u.ac.jp

Akira Fukuda
Graduate School of Information Science,
Nara Institute of Science and Technology.
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 a model to generate device drivers and a device driver generation system before. The system generates a source code of a device driver from three inputs: device driver specification, OS dependent specification, and device dependent specification. The device drivers generated in the model are evolutionary because they can be expanded their features easily. They, however, are 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, to aim at more reduction of the burden, device drivers are abstracted again, each input is defined afresh, and then a prototype of the system is implemented. As an example of the generation, an interrupt handler of a network device, FreeBSD and Linux as the target OS, and Etherlink XL as the target device are chosen. The OS dependent specification and the device dependent specification can be reused in each OS and device, respectively. As a result, an identical device dependent specification can be applied to the both OSs. The burden in generating new device drivers or porting ones to other OSs can be reduced.

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

1 Introduction

Operating systems (OSs) cannot be applied efficiently to various kinds of hardware and application software. Especially, writing device drivers is one of the most difficult tasks to develop or port 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 prepare 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 and generate evolutionary device drivers.

We have proposed a model to generate device drivers and a device driver generation system before[4, 5]. The system generates a source code of a device driver from

0-7695-0906-1/01 $10.00

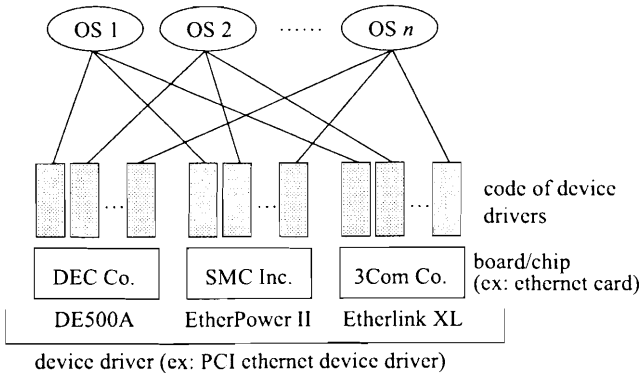

Figure 1. Current device driver development model

three inputs: device driver specification, OS dependent specification, and device dependent specification. The device drivers generated in the model are evolutionary because they can be expanded their features easily. They, however, are 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, to aim at more reduction of the burden, device drivers are abstracted again, each input is defined afresh, and then a prototype of the system is implemented. In section2, 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 model to generate device drivers and the device driver generation system[4, 5].

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

Device drivers exist to control 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 facility to be written in the device drivers is the same.

OSs control devices through the device drivers and send/receive data to/from devices. Each OS has its own data structures or data types to store data, and 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.

Therefore, 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 writing device drivers, in the present we use frequently the conditional compilation in C programming language. We write source codes corresponding to all OSs and devices in a device driver to deal with multiple OSs and devices. 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 the method abstracting a device driver in writing it 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 a template whose contents are translated into actual codes in the other specifications to define a device driver. It describes functions and data structure which the generated device driver uses.

- **OS dependent specification**

  It shows dependent parts on the OS. It describes names, arguments, return values of device driver interfaces which the OS provides. The device driver interfaces are functions which the kernel calls.

- **device dependent specification**

  It shows dependent parts on the device. It describes dependent parts on the hardware in functions of the device described in the device driver specification.

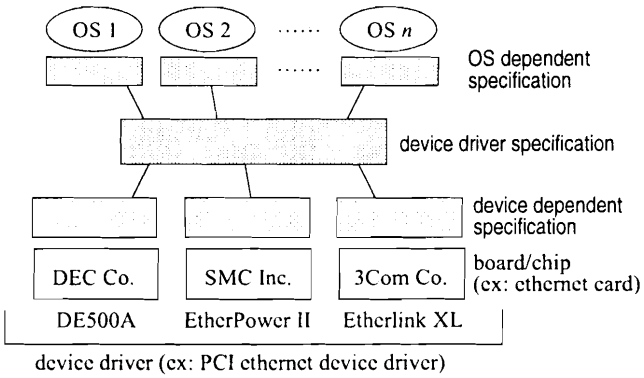

Figure 2. Device driver development model with three specifications

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 and/or I/O buses are not considered.

Figure 2 shows the device driver development model with three specifications. When a device is replaced with a new one which offers the same services, we rewrite only the part depended on a chip of the new one, and the OS dependent specification can be reused. Similarly, when an OS is replaced, the device dependent specification can be reused. As a result, we can lighten the burden in developing device drivers and solve the problems; the source codes of device drivers would be more complex and it is difficult for others to understand or modify them. Moreover, the device drivers generated in the model are evolutionary because they can be expanded their features easily by rewriting the device driver specification only, which shows a template to define a device driver. Figure 3 shows an outline of the device driver generation system.

However, the device dependent specification accounts for the greater 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 describe such codes in the device dependent specification. Hence, a person to describe the device dependent specification needs to knowledge of a target OS. 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, to aim at more reduction of the burden, 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. If multiple persons can describe separately each specification in Figure 2, we can disperse the burden in writing device drivers to developers on OS's makers and ones on device's makers 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 facilities of a device and control flows handling the data are described. The facilities 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 handle data between OS and 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 and Implementation of the System

As an example of the generation, 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

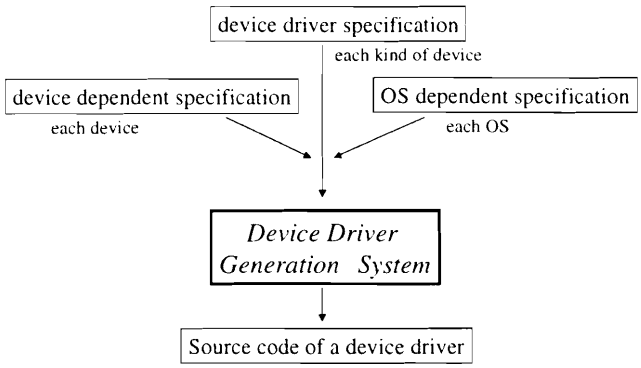

Figure 3. Overview of the device driver generation system

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. Three specifications are described according to grammar extended C programming language for 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 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 to use arguments of device driver interfaces
  - codes to use system calls
- Device dependent parts
  - codes to access devices directly
  - codes to handle variables

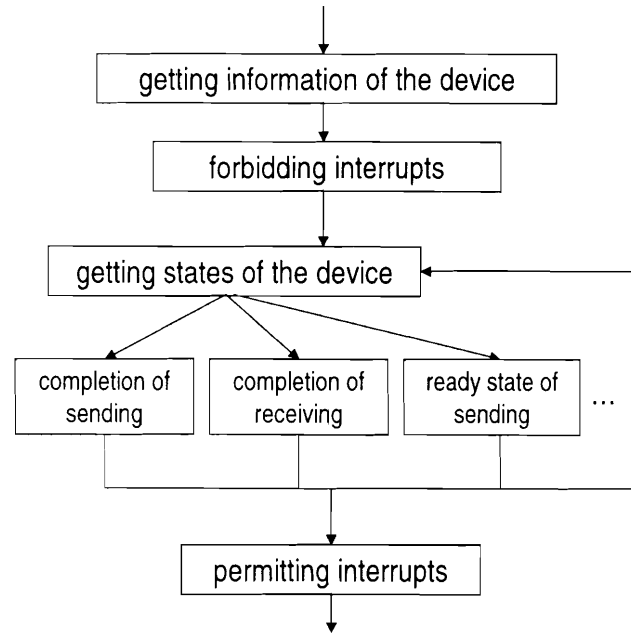

Figure 4. Overview of an interrupt handler

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. 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 followings show each label and its meaning.

- `[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
- `%<name>` — a label used to call directly other functions belonging to device driver interfaces and

```
// ---------- Interrupt Handler ----------
#begin _intr
  [proc]_get_dev_str(%(structinfo),%(buf));
  [cmd]_disable_intr(%(ioadr));
  for(;;)
  {
    [cmd]_get_status(%(status));
    if([con]_break(%(status)))
      break;
    if([con]_up_complete(%(status))
      [sub]_up_complete(%(arg),%(devinfo),%(ioadr));
    if([con]_down_complete(%(status))
      [sub]_down_complete(%(arg),%(devinfo),%(ioadr));
    if([con]_tx_complete(%(status))
      [sub]_tx_complete(%(arg),%(devinfo),%(ioadr));
    if([con]_adfail(%(status))
      [sub]_adfail(%(arg),%(status));
    if([con]_stats(%(status))
      [proc]_stats(%(arg));
  }
  [cmd]_enable_intr(%(ioadr));
  [proc]_start_rest(%(devinfo));
#end
#begin _tx_complete
  [proc]_tx_complete(%(arg),%var(devinfo),%var(ioadr));
  [cmd]_tx_complete(%(arg),%var(devinfo),%(ioadr));
#end
#begin _up_complete
  %<name>_rxeof(%(arg));
  [cmd]_up_complete(%(arg),%var(devinfo),%(ioadr));;
#end
#begin _down_complete
  %<name>_txeof(%(arg));
  [cmd]_down_complete(%(arg),%var(devinfo),%(ioadr));;
#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

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. The label %<name> is translated with a name expressing a device driver. For example, the name for the device driver of FreeBSD in Etherlink XL is `xl`. The name is given as an argument in the starting of the device driver generation system.

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 the functions, 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 used in interrupt handlers corresponds to ifnet structures and softc structures included in the ifnet 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 the following five parts.

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

```
#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>_intr(arg)
  void* arg;
%end

%begin variable
  struct %<name>_softc *sc;
  struct ifnet *ifp;
  long ioadr;
  u_int16_t status;
%end

%begin transform
  structinfo buf;
  arg sc;
  devinfo ifp;
  ioadr ioadr
  status status;
%end

%begin code
[proc]_get_dev_str(structinfo,arg)
{
  arg = (%<name>_softc*)structinfo;
}

[proc]_start_rest(devinfo)
{
  if(devinfo->if_snd.ifq_head != NULL)
    {
      %<name>_start(devinfo);
    }
}

[proc]_tx_complete(arg,devinfo,ioadr)
{
  devinfo->if_oerrors++;
  %<name>_txeoc(arg);
}
%end

#end INTERRUPT
```

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

```
#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 ioadr;
  int status;
%end

%begin transform
  structinfo dev_id;
  arg dev;
  devinfo dev;
  ioadr ioadr;
  status status;
%end

%begin code
[proc]_get_dev_str(structinfo,arg)
{
  devinfo = (device*)structinfo;
  arg = (struct %<name>_private *)devinfo->priv;
  ioadr = device->base_addr;
}

[proc]_rx_complete(arg)
{
  %<name>_rx(arg);
{
%end

#end INTERRUPT
```

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

- **code** — It describes codes depending on OS.

The label %<name>, which is used in the device driver specification, can be appended to the functions described in OS dependent specification. In the parts of the code, %<name> of the head in each name of the function is omitted. The other functions without any label signify that they call their own name. They correspond to system calls in the 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. One is a processing handling variables or I/O port. The other is 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 [con].

Figure 8 shows a part of a device dependent specification of an interrupt handler for Etherlink XL. In device driver specification, we need to write only each device's own codes. The device driver generation system does in-line expansion of the codes depending a device to a generated source code. Hence, the overhead caused by dividing into three specification can be prevented and the run time performance can be guar-

```
// ##### Interrupt ######
[cmd]_disable_intr(ioadr)
{
  outw( ioadr + XL_COMMAND, XL_CMD_INTR_END);
}
[cmd]_get_status(status)
{
  status = inw( ioadr + XL_STATUS);
}
[con]_break(status)
{
  return ( status & XL_INTRS ) == 0;
}
[con]_up_complete(status)
{
  return ( status & XL_STAT_UP_COMPLETE );
}
[con]_down_complete(status)
{
  return ( status & XL_STAT_DOWN_COMPLETE );
}
[con]_tx_complete(status)
{
  return ( status & XL_STAT_TX_COMPLETE );
}
[con]_adfail(status)
{
  return ( status & XL_STAT_ADFAIL );
}
[cmd]_enable_intr(ioadr)
{
  outw( ioadr + XL_COMMAND, XL_CMD_INTR_ENB|
    XL_INTRS);
}
[cmd]_tx_complete(ioadr)
{
  outw( ioadr + XL_COMMAND, XL_CMD_INTR_ACK|
    XL_STAT_TX_COMPLETE);
}
[cmd]_up_complete(ioadr)
{
  outw( ioadr + XL_COMMAND, XL_CMD_INTR_ACK|
    XL_STAT_UP_COMPLETE);
}
```

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

anteed.

3.4 Outline of the system

We have developed a prototype of the device driver generation system. It translates three specifications into a source code written in C programming language. We have implemented the system by Perl language.

The generation algorithm are the following.

§Device Driver Generation Algorithm

Step1. Device driver specification is scanned.

Step2. A label `#begin` is read.

Step3. If a function call is found, its contents are gotten from a subroutine in the device driver specification, OS dependent specification, or device dependent specification.

Step4. The variables in the function are translated referring to `%transform` in the OS dependent specification.

Step5. If a label `#end` is found, go to Step6. Otherwise, go to Step3.

Step6. If it reaches end of the file, this algorithm ends. Otherwise, go to Step2.

Figure 9 shows a generated source code of the function `_intr` of Etherlink XL for FreeBSD. We have generated source code of the interrupt handlers for Linux, also. We have verified the both source codes are executed correctly.

4 Discussion and Evaluation

In this section, we describe portability and availability of our method and compare it with I_2O(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.

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. 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.

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

```
/* The interrupt handler does all of the Rx thread */
/* work and clean up after the Tx thread. */

static void xl_intr(sc)
void* sc;

{
  struct xl_softc *sc;
  struct ifnet *ifp;
  long ioadr;
  u_int16_t status;

/* #begin _intr */
  {
    sc = (xl_softc*)buf;
  }
  {
    outw( ioadr + XL_COMMAND, XL_CMD_INTR_END);
  }
  for(;;)
  {
  {
    status = inw( ioadr + XL_STATUS);
  }
  if (( status & XL_INTRS ) == 0 )
        break;
  if ( status & XL_STAT_TX_COMPLETE )
  {
    ifp->if_oerrors++;
    xl_txeoc(sc);
  }
  if ( status & XL_STAT_ADFAIL )
  {
    xl_reset(sc);
    xl_init(sc);
  }
  if ( status & XL_STAT_STATSOFLOW )
  {
    sc->xl_stats_no_timeout = 1;
    xl_stats_update(sc);
    sc->xl_stats_no_timeout = 0;
  }
  }
}
```

Figure 9. A part of a generated source code of Etherlink XL for FreeBSD

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 OS's makers and ones on device's makers.

Table 1 shows the number of lines of each specifications which we need to describe about Etherlink XL in our previous and current definition of the inputs. The burden cannot be simply compared by the number of lines but it can be adopted as one of standards. In the previous definition, it of the device dependent specification is about 85 times as large as one of the OS dependent specification and the burden concentrates on describing the device dependent specification. In the current definition, 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.

Table 1. Comparison of Line Counts

Specifications	previous	current
device driver specification	14	58
OS dependent specification	2	53
device dependent specification	171	69

Moreover, even if persons who describe OS dependent specification do not have to know each device's details or persons who describe device dependent specification do not have to 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.

A significant performance degradation of the generated code does not occur in comparison with source code written by the traditional method. The reason is that the device driver generation system can generate the source code nearly as same as an existing source code for the interrupt handler (see Figure 9).

4.3 Comparison with I_2O

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) 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 model to generate device drivers and 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. In this paper, device drivers were abstracted again, and each input was defined more

minutely, and then a prototype of the system is implemented. As an example of the generation, 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. The device drivers generated in the model are evolutionary because they can be expanded their features easily by rewriting the device driver specification only.

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 who describe OS dependent specification do not know each device's details or persons who 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.

Future issues are as follows:

- Extension to other OSs or devices.

  In this paper, 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 buses.

  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 buses 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

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

[2] D.C.R. Jensen, J. Madsen, and S. Pedersen: "The Importance of Interfaces: A HW/SW Codesign Case Study," *Proc. 5th Int'l Works. on Hardw./Softw. Codesign (CODES/CASHE'97)*, pp.87-91, 1997.

[3] S.J. Ryan: "Synchronization in Portable Device Drivers," *ACM OS Review*, Vol.32, No.4, pp.62-69, 1998.

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

[5] T. Katayama, K. Saisho, and A. Fukuda: "Proposal of a Support System for Device Driver Generation," *Proc. 1999 Asia-Pacific Softw. Eng. Conf. (APSEC'99)*, pp.494-497, 1999.

[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.