UNIX系OSにおけるデバイスドライバの抽象化と生成システムの実現

奥野幹也[†,☆] 片山徹郎[†]
最所圭三[††] 福田晃[†]

デバイスドライバの作成は，対象デバイスやオペレーティングシステム（OS）に関する知識が必要となるため，多大な時間と労力を必要とする．本論文では，この労力を軽減するために，デバイスドライバ生成の基本的要素であるOSのインタフェースとデバイスへの入出力法に焦点を絞り，デバイスドライバを，デバイスドライバ仕様，OS依存仕様，デバイス依存仕様の3つの部分に抽象化する．また，これら3つの仕様を入力とし，デバイスドライバのソースコードを出力するデバイスドライバ生成システムを提案する．システムの適用例として，OSはFreeBSDとLinux，デバイスはネットワークデバイス，処理は割込み処理を取り上げる．OS依存仕様とデバイス依存仕様は，それぞれOSとデバイスごとに再利用可能であり，本論文において，FreeBSDとLinuxの2つのOSに対して同じデバイス依存仕様を適用できた．このことから，デバイスドライバを新規に作成する場合や，デバイスドライバを異なるOSに移植する場合にかかる時間と手間を，削減することができる．

Abstraction and Generation System of Device Drivers for UNIX-like OSs

Mikiya Okuno,[†,☆] Tetsuro Katayama,[†] Keizo Saisho[††] and Akira Fukuda[†]

Writing device drivers spends much time and makes efforts because it needs knowledge of the target device and operating system (OS). To lighten the burden, this paper aims at interfaces of OS and a way to handle data to/from devices, and they are abstracted into three parts: device driver specification, OS dependent specification, and device dependent specification. Moreover, a device driver generation system, which generates a source code of a device driver from three specifications, is proposed. As an example of the generation, FreeBSD and Linux as the target OS and an interrupt handler of a network device are chosen. The OS dependent specification and the device dependent specification can be reused in each OS and device, respectively. As a result, the both OSs can be applied to an identical device dependent specification. The burden in generating new device drivers or porting ones to other OSs can be reduced.

1. はじめに

オペレーティングシステム（以下OS）の中のカーネル部分は，ハードウェアを直接操作する．このため，異なるアーキテクチャにOSを移植する際にはカーネルの大幅な書き換えが必要となる．ハードウェアに依存するカーネル部分の中でも，デバイスドライバは最もハードウェアに依存するため，その作成は，OSの開発や移植において最も時間と労力を要する[1),2)]．これには，以下の理由があげられる．

- 同じサービスを提供するデバイスでも，使用するコントローラが違えば，それに適合した新しいデバイスドライバを作成しなければならない．
- デバイスのハードウェアや開発の対象となるOSに関する知識に加えて，タイミング制御などの複雑で注意深いプログラミングを要求される．

マルチメディアやインターネットの近年の発展を背景として，多様なデバイスの登場が予測される．この

† 奈良先端科学技術大学院大学情報科学研究科
Graduate School of Information Science, Nara Institute of Science and Technology
☆ 現在，サン・マイクロシステムズ株式会社
Presently with Sun Microsystems, Inc.
†† 香川大学工学部信頼性情報システム工学科
Department of Reliability-based Information Systems Engineering, Faculty of Engineering, Kagawa University

ため，デバイスドライバの作成にかかる時間や労力は，今後ますます深刻な問題になると考えられる[3]．そこで，デバイスドライバの作成を支援し，その負荷を軽減することは非常に有用である．

実際に使用されているデバイスドライバの中には，デバイスドライバを複数のOS，もしくは複数のデバイスに適用するために，また，デバイスドライバ開発にかかる負荷を軽減するために，C言語の条件つきコンパイルを用いているものがある．たとえば，同一のチップメーカーのデバイスを用いる場合，シリーズ化されているチップを用いることが多い．シリーズ化されているチップは構造が似通っているので，デバイスドライバを少し変更するだけで，同じシリーズの他のチップに適用することができる．その場合，C言語の条件つきコンパイルを用いて，1つのソースコードで同シリーズのチップすべてに対応したデバイスドライバを提供するなどの工夫が行われている．しかしこの手法では，ソースコードの見通しが悪くなり理解が困難になる．このため，新しいデバイスを対象とした機能を追加する場合には，そのソースコードに精通したプログラマでないと，ソースコードを修正することが困難であるなどの問題が起きる．また，異なるOSに移植する際には，同じ系統のOS以外では，C言語の条件つきコンパイルを適用することが難しいなどといった問題があるので，このままでは複数のデバイスやOSにデバイスドライバを適用するためには不十分である．

一方，使用するOSを固定した場合には，OSの固有部分を自動生成することにより，デバイスドライバの作成を支援するツールが商用に存在する（Toolcraft社のWindows用ドライバ開発ツールWinDK[4]など）．しかしながら，これらのツールは対象となるOSを1つに限定しており，デバイスドライバを他のOSに移植する際にかかる負担の軽減にはならない．

そこで我々は，デバイスドライバの作成にかかる負担を軽減するために，デバイスドライバを抽象化し，デバイスドライバ生成システムを設計する[5]．具体的には，デバイスドライバ作成に必要な情報を，次の3つの仕様として定義する．

- デバイスドライバの処理の流れを表す「デバイスドライバ仕様」
- OSとのインタフェースを表す「OS依存仕様」
- デバイスとのインタフェースを表す「デバイス依存仕様」

これらの3つの仕様以外にも，デバイスドライバの作成には，CPUやI/Oバスのアーキテクチャが必要であると考えられる．しかしながら，これらすべてを考慮した場合，問題が複雑化するために，本論文ではデバイスドライバ生成の基本的要素であるOSのインタフェースとデバイスへの入出力法に焦点を絞り，CPUやI/Oバスの違いは考慮しないものとする．デバイスドライバ生成システムは，上述の3つの仕様を入力とし，デバイスドライバのソースコードを出力する．このシステムを用いることにより，デバイスドライバの開発者を，デバイスドライバのOS側を開発する人とデバイス側を開発する人に分けることができる．開発すべき箇所を分散させることにより，開発者1人あたりの記述量を減らし，コードを見やすくし，開発者の必要とする知識が限定され，開発にかかる負担を減らすことができる．また，異なるOSにデバイスドライバを移植する際に，デバイス依存仕様を流用することで開発にかかる負担が減少する．

2章では，デバイスドライバの抽象化について述べる．3章では，2章で行った抽象化に基づき，デバイスドライバ生成システムを提案し，その詳細を述べる．デバイスドライバ生成システムの入力の形式とシステムの動作を定め，その記述例を示す．作成例として，OSにはFreeBSD[6]およびLinux[7]，デバイスには代表的なPCIイーサネットカードである3Com社製Etherlink XL，処理には割込み処理を取り上げて説明する．4章では議論および評価について述べる．

2. デバイスドライバの抽象化

デバイスドライバは，OSがデバイスを操作するためのプログラムであり，デバイスドライバは，使用するOSとデバイスとに合わせて作成する．デバイスドライバの開発には，対象となるOSに関する知識に加え，対象となる個々のデバイスに対する詳細な知識も必要であり，開発者の負担が非常に大きくなる．また，デバイスドライバは，デバイスもしくはOSのいずれかを変更するたびに，**図1**に示すようにそれぞれに対応させて作成しなければならない．言い換えると，デバイスドライバはOSやデバイスに依存して作成されている．このため，作成するべきデバイスドライバの数が非常に多くなる．

デバイスドライバは，OSからデバイスが仮想的に同じに見えるようにするために導入されている．このため，デバイスドライバに要求される機能は，デバイスの種類によって決まっている．たとえば，代表的なイーサネットカードであるEtherlink XLとDE500Aを比べると，コントローラチップは違うが，イーサネットカードとしての機能，すなわち，デバイスドライバ

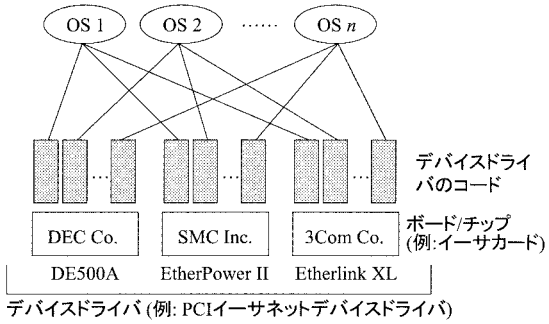

図 1 現在のデバイスドライバ開発
Fig. 1 Current device driver development.

に記述すべき機能は同じである．

OS は，デバイスドライバを用いてデバイスを操作することでデータを入出力する．OS が異なれば，データを格納するための構造体やデータ型が異なる．また，デバイスが異なれば，データを受け渡すインタフェースやタイミング，データ型などが異なる．

デバイスドライバを通して入出力されるデータは，デバイスの種類が同じなら同じ情報を表しているが，OS によって，またはデバイスによってそのデータの扱われ方が異なる．たとえば，イーサネットカードでは，送受信するデータはイーサネットフレームであるが，FreeBSD と Linux ではそれを格納する構造体が mbuf と skbuff であるように，管理の仕方が異なる．つまり，デバイスドライバは，本質的には同じデータを OS とデバイスに合わせて，それぞれで用いられるデータ形式に変換するプログラムであるといえる．

よって，デバイスドライバは，以下の 3 つの部分に抽象化できる．

- OS とデバイスの間で入出力されるデータの受け渡しを行う部分．
- OS とデータを入出力するためのインタフェース
- デバイスとデータを入出力するためのインタフェース

デバイスドライバにおけるデバイスや OS への依存をできるだけ解消するために，現状では，C 言語の条件つきコンパイルを利用して記述していることが多い．複数のデバイスや OS に対応するために，1 つのデバイスドライバのソースコードの中に，対象となりうるすべてのデバイス，または OS に対するコードを，条件によって使い分ける形で記述している．このように，多くの既存のデバイスドライバは，条件つきコンパイルを利用することによって，複数の OS やデバイスに対応する構造をしている．しかしながらこの方法では，ソースコードの見通しが悪くなり，複雑になることで，よほどそのソースコードを熟知したプログラマでない限り，新しいデバイスを対象とした機能を追加するときなどにソースコードを修正することが困難である．

以上のことから，デバイスドライバを複数の OS やデバイスに対応できるように，作成の段階で上述の 3 つの部分に抽象化することは自然な流れであり，非常に重要である．そこで，上述の 3 つの部分をそれぞれ記述することにより，デバイスドライバを生成する手法を提案する．これにより，ソースコードの見通しが悪い，移植や機能の拡張が難しいという問題点も解消できると考えている．抽象化した 3 つの部分をそれぞれ以下のように，デバイスドライバ仕様，OS 依存仕様，デバイス依存仕様と定義する．

- **デバイスドライバ仕様**
  デバイスの持つ機能を使用するために必要なデータの種類とそのデータを入出力する大まかな流れを示す．デバイスドライバ仕様は，デバイスの種類に対して 1 つ記述すればよい．したがって，ターゲットとする OS やデバイスを変更する際や，一からデバイスドライバを作成する際に新たに記述する必要はない．
- **OS 依存仕様**
  OS からデバイスへのデータの与え方とデバイスドライバを OS から呼び出す際のインタフェースを記述する．
- **デバイス依存仕様**
  デバイスドライバが，デバイスに対してデータを入出力する際のインタフェースやタイミング，データ型などを記述する．

ここで，デバイスドライバを作成するためには，対象となる CPU や I/O バスも考慮しなければならない．たとえば，CPU が異なると，ビッグエンディアンかリトルエンディアンかで，デバイスドライバのソースコードにおいて，構造体メンバの記述順序が逆になるなどの状況が生じる．また，ほとんどのデバイスはターゲットとなる I/O バスが決まっており，それに合わせてドライバを記述しなければならない．これらすべてを考慮した場合，問題が複雑化するために，本論文ではデバイスドライバ生成の基本的要素である OS のインタフェースとデバイスへの入出力法に焦点を絞り，CPU や I/O バスの違いは考慮しないものとする．

上記の 3 つの仕様を用いると，**図 2** の環境を提供できる．この環境下では，機能は同じでハードウェアの構造が異なるデバイスが新規に開発された場合に，そのデバイス依存仕様さえ記述すれば，デバイスドライバを生成できる．また，デバイスドライバを異なる

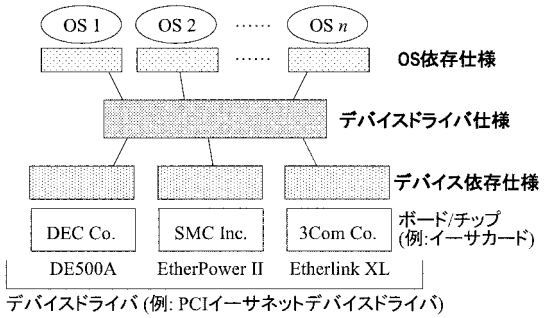

図 2 3つの仕様を用いたデバイスドライバ開発
Fig. 2 Device driver development with three specifications.

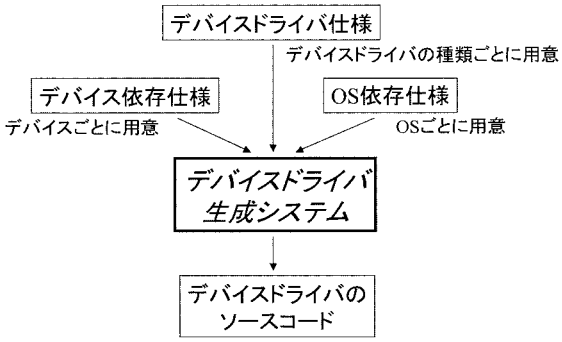

図 3 デバイスドライバ生成システムの概要
Fig. 3 Overview of device driver generation system.

OS に移植する際には，OS 依存仕様のみを変更することによって，デバイスドライバを生成できる．このことにより，デバイスドライバを新規に作成する場合や，デバイスドライバを異なる OS に移植する場合にかかる時間と手間を，大幅に削減できる．

3. デバイスドライバ生成システム

3.1 システムの概要と設計

2 章で行った抽象化に基づいて，定義した 3 つの仕様を入力とし，デバイスドライバのソースコードを出力するデバイスドライバ生成システムを提案する[5]．**図 3** にその概要を示す．

以下に，3 つの入力の設計方針を示す．

- **デバイスドライバ仕様**
  デバイスドライバ仕様は，デバイスドライバを定義する雛型であり，デバイスの種類ごとに定義する．これは，デバイスの種類によって，デバイスが提供できる機能が定義されているからである．デバイスドライバ仕様はすべての OS に対して，その種類のデバイスが持つ機能を抽象化するので，一度記述すれば新たに記述しなくてよい．
- **OS 依存仕様**
  OS がデバイスドライバに対して要求する処理，および，デバイスドライバが OS に対して要求する処理を記述する箇所が OS 依存仕様である．OS 依存仕様を記述するためには，OS に関する知識と，OS がそのデバイスに対して要求するデータや機能についての知識を要求される．すべての OS がその種類のデバイスに対して同じ機能を要求するのではない．すなわち，OS によってカバーしている技術が様々であるので，デバイスに要求する機能の種類も様々である．OS 依存仕様を記述するには，デバイスの詳細は知らなくてよいが，デバイスのどのような機能を OS が要求するかといった知識が必要になる．OS 依存仕様は OS とデバイス依存仕様との橋渡し的な存在である．OS が要求する機能を，デバイスが提供する機能にどのように結び付けるかを決定する．
- **デバイス依存仕様**
  デバイス依存仕様は，OS から要求された機能がどのように実現されるかを具体的に記述する．あるデバイスがあらゆる OS に対して，そのデバイスとしての機能を提供するためには，その種類のデバイスとして提供すべき機能を，OS が使う使わないにかかわらず，すべて用意しておく必要がある．デバイス依存仕様は，デバイスが要求される機能を，デバイスが提供する機構にマッピングするものである．具体的には，機能を実現するために，どのようにデバイスにアクセスして，どのようにデバイスを操作するかを記述する．機能の実現方法は，デバイスによって大きく違う部分である．デバイスにデータを入出力する際のインタフェースやタイミング，データ型などをここで記述する．

次章では，具体的なデバイスの例を取り上げて，デバイスドライバ生成システムの 3 つの入力である，デバイスドライバ仕様と OS 依存仕様，デバイス依存仕様のそれぞれについて述べる．

3.2 システムの入力の詳細と記述例

本論文では，記述例の対象とする OS として，FreeBSD および Linux を選んだ．これは，2 つの OS が代表的な PC UNIX であり，それらのソースコードが公開されているため，デバイスドライバのソースコードが参照できるからである．また，対象とするデバイスは，ネットワークデバイスを選び，ネットワークのインタフェースは，イーサネットとした．これは，現

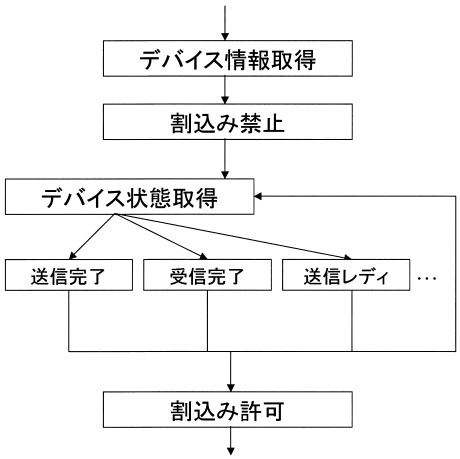

図 4 割込み処理の流れ
Fig. 4 Overview of interrupt handler.

在最も普及しており，新たなデバイスの開発スピードも早く，デバイスドライバの作成に要する負担が大きいデバイスであると考えたためである．具体的には，代表的な PCI（Peripheral Component Interconnect）イーサネットカードである 3Com 社製 Etherlink XL を選んだ．

本論文では，ネットワークデバイスの，初期化関数で登録される割込み処理の機能を作成する．システムへの入力である各仕様を，どのように与えるかについて，この割込み処理を例にとり説明する．

3.2.1 デバイスドライバ仕様

デバイスドライバ仕様はデバイスの持つ機能を抽象化し，関数として記述する．具体的には，関数の入出力や処理の流れが相当する．

割込み処理は，**図 4** に示すように大きく分けて，デバイス情報の構造体取得，デバイスに対する割込み禁止，デバイスの状態取得，各状態に対する処理，デバイスに対する割込み許可，という処理に分けられる．割込みが起こってから，デバイスが処理を必要としない状態になるまで，PCI イーサネットカードでは，送信完了，受信完了などの複数の要求が同時に起こったり，1 つの要因を処理している間に他の要因が発生することもある．このため，割込みが起こってから，割込み要因がなくなるまで繰り返し処理する．

デバイスドライバ仕様は，デバイスドライバインタフェースの中で行う処理の流れを記述する．そこで使用する関数は，以下のどの部分を含むかによって 2 つに分けることとする．

- OS 依存部分
  - OS に固有なデータにアクセスするコード
  - カーネルが保持しているデータにアクセスするコード
  - デバイスドライバインタフェースの引数を使用するコード
  - システムコールを使用するコード
- デバイス依存部分
  - デバイスに直接アクセスするコード
  - ローカル変数，グローバル変数にアクセスするコード

この基準に基づいてデバイスドライバの機能を OS 依存とデバイス依存の 2 つの部分に分けることにより，OS 依存のコードを OS 依存仕様に，デバイス依存のコードをデバイス依存仕様に記述することができる．すなわち，デバイスドライバ仕様は，処理の流れをデバイス依存と OS 依存に分けられるレベルまで詳細化する必要がある．

図 5 に Etherlink XL の割込み処理のデバイスドライバ仕様の記述例の一部を示す．

デバイスドライバ仕様では，処理の流れを記述するので，制御文と関数呼び出しのみにより構成される．仕様の記述は C 言語を本システム専用に拡張した文法に沿う．特に，制御文は C 言語の文法をそのまま用いる．処理の内容は，`#begin`〜`#end` で囲まれた部分に記述する．`#begin` に続いて処理の名前を記述する．

デバイスドライバ仕様の関数呼び出しでは関数の種類を区別するために関数呼び出しの前に，その種類を表すラベルを付加する．付加するラベルとその意味を次に示す．

- `[proc]` — OS 依存仕様に属する処理．
- `[cmd]` — デバイス依存仕様に属する処理．
- `[con]` — 分岐のための条件．デバイス依存仕様に所属．
- `[sub]` — デバイスドライバ仕様の中の下位の処理ルーチン．
- `%` — デバイスドライバインタフェースに属する他の関数を直接呼び出すときや，C 言語で直接記述された関数を呼び出すときに使用する．

それぞれのラベルは，その処理がどの仕様に記述されているかを示すためのものでもある．ラベル以外が同じ名前でも，ラベルが違えば違う処理を示している．`%`ラベルの部分には，デバイスドライバの名前を表すプリフィクスを追加する．ここで取り上げている Etherlink XL の FreeBSD 用デバイスドライバの場合は，`xl` がプリフィクスである．

関数呼び出しに使用される引数は，出力時に OS 依存仕様を参照して，システムによって埋め込まれる．

```
// ----------- 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
```

図 5 割込み処理のデバイスドライバ仕様の記述例の一部
Fig. 5 Sample of device driver specification of interrupt handler.

```
#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
// ---------- OS Specific Layer ----------

// ##### Interrupt ######

[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
```

図 6 割込み処理の FreeBSD における OS 依存仕様の記述例の一部
Fig. 6 Sample of OS dependent specification of interrupt handler on FreeBSD.

3.2.2 OS 依存仕様

OS 依存仕様には，OS 側が要求するデバイスドライバインタフェース，その関数の機能，使用される変数，さらにデバイスドライバ仕様で用いた OS 依存部にあたる [proc] ラベルを付加した関数のコードを記述する．

割込み処理で使用されるデバイス情報は，FreeBSD においては ifnet 構造体とその内部に含まれる softc 構造体が相当する．softc 構造体は各デバイスに固有のデータを格納する構造体である．Linux では device 構造体がデバイス情報を保持している．

FreeBSD の OS 依存仕様の記述例の一部を**図 6** に，Linux の OS 依存仕様の記述例の一部を**図 7** に，それぞれ示す．

#begin と #end は OS 依存仕様における 1 つの処理の開始と終了を示す．OS 依存仕様を以下の項目に分けた．それぞれの項目が果たす役割を以下に示す．

- **function** — この関数が行う処理の内容を説明する．出力されるソースコードに，システムによってコメント文として埋め込まれる．
- **prototype** — デバイスドライバインタフェースのプロトタイプを宣言する．引数と戻り値を設定する．FreeBSD と Linux ではインタフェース名も引数も違う（図 6，図 7 参照）．
- **variable** — 関数内で使用されるローカル変数をリストアップする．出力されるデバイスドライバのソースコードには，C 言語の仕様に従い，システムによって関数の開始直後に埋め込まれる．FreeBSD と Linux では ioadr と status は同じだが，他の変数は名前も機能も違う（図 6，図 7 参照）．
- **transform** — 使用されるすべての変数（ローカル変数，グローバル変数，引数）に対して別名を与える．OS やデバイスによって使用する構造体が変更され，それにともない変数名が変更されることがある．このため，3 つの仕様を記述する際に，変数名を変更せずに仕様を記述するために導入し

```
#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 %_private *)devinfo->priv;
  ioadr = device->base_addr;
}

[proc]_rx_complete(arg)
{
  %_rx(arg);
}
%end

#end INTERRUPT
```

図 7 割込み処理の Linux における OS 依存仕様の記述例の一部
Fig. 7 Sample of OS dependent specification of interrupt handler on Linux.

```
// ##### 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);
}
```

図 8 Etherlink XL における割込み処理のデバイス依存仕様の記述例の一部
Fig. 8 Sample of device dependent specification of interrupt handler for Etherlink XL.

た．デバイスドライバ生成システムの 3 つの仕様では，一貫してこの別名を使用する．たとえば，図 6 で structinfo と buf が対応していることが分かる．これは，3 つの仕様を通して，buf にアクセスする場合は structinfo という名前を使用することを意味している．仕様に記述された structinfo はシステムによって出力されるソースコード中には buf として埋め込まれる．

- **code** — OS 依存のコードを記述する．

`%<name>`の部分に，デバイスに固有の名前が入る．その名前は，デバイスドライバ生成システムの起動時に引数として与える．コード部は各関数名の先頭の`%<name>`の部分を省略する．

OS 依存仕様で記述するコードの文法も，C 言語を本システム用に拡張したものを用いる．具体的には，関数呼び出しと演算，制御文を使用する．ここに記述する関数には，デバイスドライバ仕様で用いた`%`ラベルを付加することができる．ラベルを付加しない関数は，その名前の関数を呼び出すことを意味する．カーネル内のシステムコールなどがそれにあたる．

3.2.3 デバイス依存仕様

デバイス依存仕様に記述する関数は，2 種類に分類することができる．変数や I/O ポート（またはメモリマップト I/O）などに対する処理と，デバイスドライバ仕様での制御文のために状態や条件を返す処理である．前者には`[cmd]` ラベルを，後者には`[con]` ラベルを付加する．デバイス依存仕様はデバイスに対して不変で，対象となる OS が変更されても一貫して使用することができる．

3Com 社製 Etherlink XL のデバイス依存仕様の記述例の一部を，**図 8** に示す．

デバイス依存仕様には，デバイスに固有のコードのみを記述する．このコードは，OS から直接呼び出されるのではなく，デバイスドライバ仕様を通して呼び出される．本システムでは，出力するソースコードに，デバイス依存仕様のコードをインライン展開する．これにより，階層化によるオーバヘッドを防ぎ，実行時における性能を維持することができる．

デバイス依存仕様で使用する文法も，他の仕様同様，C 言語を本システム用に拡張したものを用いる．具体的には，関数呼び出しと演算，制御文である．他の 2

つの仕様とは違いOSに依存しないようにするために，関数呼び出しにはラベルを付加しない．

3.3 システムの実装

デバイスドライバ生成システムは，本質的には3つの仕様をC言語のソースコードに変換するトランスレータである．ただし，複雑な変換は行わないので，生成システムのアルゴリズムは比較的簡単である．生成システムの記述にはPerlを用いた．生成アルゴリズムを以下に示す．

(1) `#begin`ラベルを読み込む．
(2) デバイスドライバ仕様を上から順に走査する．
(3) 関数呼び出しが見つかれば，その中身をデバイスドライバ仕様の下位の処理ルーチン，OS依存仕様，デバイス依存仕様，のいずれかから取得する．
(4) 関数内の変数をOS依存仕様を参照にして，変換する．
(5) 変換した中身を出力する．
(6) `#end`ラベルが出てきたら終了し，そうでなければ(3)へ．

ここで，`#begin`ラベルおよび`#end`ラベルは，3.2.1項で説明したものである．

本システムは，デバイスドライバ仕様を読み込んで，そこにOS依存仕様，デバイス依存仕様から必要な情報を取得して，適切な形に変換して埋め込んでいく．システムはまずデバイスドライバインタフェースのプロトタイプと，関数内部で使用されるローカル変数を出力する．関数呼び出しが見つかれば，関数の中身をそこに展開するが，その中身は，3.2.1項で説明した関数の種類を表すラベルを参照して，デバイスドライバ仕様の下位の処理ルーチン，OS依存仕様，デバイス依存仕様，のいずれかから取得する．デバイスドライバ仕様の下位の処理ルーチンまたはOS依存仕様から，関数の中身を取得する場合，その中身でも関数呼び出しが見つかる可能性があり，その場合，デバイスドライバ仕様に関数の中身を埋め込むときと同様に，その関数の中身をOS依存仕様，またはデバイス依存仕様から取得する．また，関数の中身を取得するとき，そこで使用する変数を3.2.2項で説明した`%transform`を参照して適切な名前に変換する．

本システムは，C言語で記述されたデバイスドライバのソースコードを出力する．出力されたソースコードはコンパイルして，バイナリコードに変換された後，実際に使用できるようになる．

本システムによるEtherlink XLにおけるFreeBSD用の割込み処理の出力結果を**図9**に示す．本システムが出力した，Etherlink XLにおける割込み処理のFreeBSD用とLinux用それぞれのデバイスドライバが正常に動作することを確認した．

```
/* 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;
  }
  }
  {
    outw( ioadr + XL_COMMAND, XL_CMD_INTR_ENB|
      XL_INTRS);
  }
  {
    if(ifp->if_snd.ifq_head != NULL)
      {
        xl_start(ifp);
      }
  }

}
```

図9 システムの出力結果の一部（Etherlink XLにおける割込み処理のFreeBSD用デバイスドライバソースコード）
Fig. 9 A part of output of device driver generation system: Source code of interrupt handler of Etherlink XL for FreeBSD.

4. 議論および評価

本章では，デバイスドライバ生成システムの移植性と有効性について述べ，I_2O（Intelligent Input Output）[11]との比較を行う．

4.1 デバイスドライバ生成システムの移植性について

デバイスドライバを抽象化することにより，デバイスドライバをOSに依存する部分とデバイスに依存する部分に詳細に分けることができた．分けた結果に基づき，デバイスドライバ生成システムの入力を記述した．この入力を用いて，デバイスドライバ生成システムからデバイスドライバを作成した．同じデバイス依存仕様を用いて作成したネットワークデバイスのデバ

イスドライバが，Linux と FreeBSD において，正常に動作することを確認した．

デバイス依存仕様を変更することなく，異なる OS 依存仕様でデバイスドライバを生成できたことは，デバイス依存仕様の移植性を実証できたことになる．

4.2 デバイスドライバ生成システムの有効性について

以前我々が提案したシステム[8)~10)]と，現在のデバイスドライバ生成システムの違いについて述べる．以前我々が提案したシステムでは，デバイスドライバを 3 つに抽象化するという点では同じであったが，それぞれの仕様，特に OS 依存仕様に対するとらえ方が違っていた．以前のシステムにおける OS 依存仕様は，OS がデバイスドライバを呼び出す際のインタフェースのみを記述するもので，OS がどのようにデバイスとデータを入出力するかという部分が抜けていた．そして，その部分をデバイス依存仕様に含めていたので，デバイス依存仕様の中に OS 依存の部分が残る結果となり，異なる OS へデバイスドライバを移植する際の妨げになるとともに，この部分が他の部分に比べ大きくなった．これは，既存のデバイスドライバを参考にするなど，リバースエンジニアリングを行ったために起こったことである．

現在のシステムにおいて，FreeBSD と Linux の 2 つの OS に対して同じデバイス依存仕様を適用できたのは，デバイスドライバを抽象化することにより，デバイスドライバを OS に依存する部分とデバイスに依存する部分に分けたことによる．

表 1 は，Etherlink XL において，以前のシステムと現在のシステムにおいて，それぞれの仕様で開発すべき行数を示したものである．

開発にかかる負担は単純に行数では比較できないが，1 つの目安になる．以前のシステムでは，デバイス依存仕様が OS 依存仕様の約 85 倍あり，開発がデバイス依存仕様だけに偏っていることが分かる．現在のシステムでは，一番行数の少ない仕様と多い仕様での行数がほとんど変わらないため，開発の際の負荷が分散したといえる．

さらに，OS 依存仕様の記述者は個々のデバイスの詳細を，デバイス依存仕様の記述者は各 OS の詳細を，それぞれ知らなくても，デバイスドライバの開発に携わることができる．開発者がカバーすべき箇所が減少しており，開発者の負担が減少しているといえる．

また，本論文の例では，生成システムを適用して出力したデバイスドライバにおける性能の低下は見られなかった．これは，生成システムが既存の割込み処理のソースコードとほぼ同一のコード（図 9 の生成例参照）を生成することができたからだと考えられる．

表 1 行数の比較
Table 1 Comparison of line counts.

仕様	以前のシステム	現在のシステム
デバイスドライバ仕様	14	58
OS 依存仕様	2	53
デバイス依存仕様	171	69

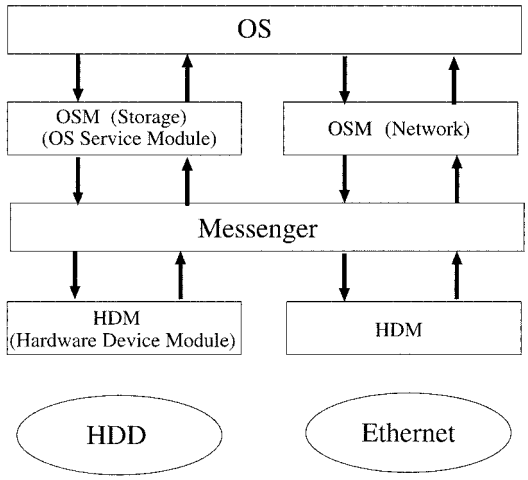

図 10 I_2O のコンセプト
Fig. 10 Concept of I_2O.

4.3 I_2O との比較

I_2O（Intelligent Input Output）SIG[11)]において，OS とデバイスドライバとの間に標準インタフェース I_2O を定めている．I_2O の仕様では，デバイスドライバを抽象化することにより，OS に依存した OSM（Operating System Service Module），ハードウェアに依存した HDM（Hardware Device Module），OSM と HDM の間で情報を送受する Messenger の 3 つの階層に分けている（**図 10** 参照）．OSM は OS ベンダが，HDM はハードウェアベンダが作成することにより，デバイスドライバ作成の作業を分担できる．OSM と HDM 間の通信に使うパケットの形式を明確に定義することで，OS が異なっても，1 つのデバイスに対して，HDM 自体を書き換えることなく通信が可能となる[12)]．

HDM は I/O 専用のプロセッサで実行されることになるが，これは，ホストプロセッサの負荷は減少するが，コストアップにつながる．さらに，デバイスドライバを階層構造にしたことによるオーバヘッドの問題が生じる．特に高速なデバイスやリアルタイム性を要求されるデバイスにおいては，この問題が致命的になることも考えられる．これに対して，本手法では，デ

バイスドライバの抽象化をターゲットデバイスやOSで行うものではなく，作成する段階で行うため，このような問題は起こらない．

5. ま と め

本論文では，デバイスドライバを，OSやデバイスへの依存を解消するために，その性質を分析し，3つの部分に抽象化し，それぞれ，デバイスドライバ仕様，OS依存仕様，デバイス依存仕様と定義した．これら3つの仕様を入力とし，デバイスドライバのソースコードを出力する，デバイスドライバ生成システムを提案した．さらに，デバイスドライバ生成システムの入力の詳細とその記述例を示した．

デバイスドライバ仕様はデバイスドライバの核となる部分であり，他の2つの仕様でコードとして具体化される機能を用いて記述する．OS依存仕様はOSから呼び出すときのデバイスドライバのインタフェースの定義と，OSに依存したコードを記述するものである．デバイス依存仕様はデバイスドライバ仕様で抽象化された機能のうち，OSに依存しない部分を具体化するものである．

デバイスドライバ生成システムの入力の記述例として，代表的なPC UNIXであるFreeBSDとLinuxにおける，ネットワークデバイスの1つであるEtherlink XLの割込み処理を取り上げた．FreeBSDとLinuxの2つのOSに対して同じデバイス依存仕様を適用できた．これは，デバイスドライバを抽象化することにより，OSに依存する部分とデバイスに依存する部分に分けたことによる．また，OS依存仕様とデバイス依存仕様は，それぞれOSとデバイスごとに再利用可能である．さらに，OS依存仕様の記述者は個々のデバイスの詳細を，デバイス依存仕様の記述者は各OSの詳細を，それぞれ知らなくても，デバイスドライバの開発に携わることができる．開発者がカバーすべき箇所が減少しており，開発者の負担が減少しているといえる．

今後の課題として，以下のことがあげられる．

- **他の機能への適用**
  本論文では，デバイスドライバ生成システムにおいて，ネットワークデバイスの割込み処理への適用を行った．今後は，まず送信処理，受信処理などに適用する予定である．そして，最終的にはネットワークデバイスのすべての機能に対して適用する予定である．機能に適用するには，デバイスドライバ仕様を作成することが必要であり，時間のかかる作業である．
- **他のOSやデバイスへの拡張**
  現在は，代表的なPC UNIXであるFreeBSDとLinuxをターゲットOSとしているが，他のOSへの適用を考えている．具体的には，Windowsを第1のターゲットとして考えている．これは，現在Windowsが最もよく普及しているOSであり，デバイスメーカがデバイスを提供する場合，そのデバイスをWindowsに対応させるケースが多いので，本システムをWindowsでも適用できればメリットが大きいと考えるからである．ターゲットデバイスとしては，他のPCIイーサネットデバイスを検討している．また，イーサネット以外の他のPCIネットワークデバイスへの適用も検討している．
- **異なるCPUやI/Oバスへの適用**
  本論文では，デバイスドライバ生成の基本的要素であるOSのインタフェースとデバイスへの入出力法に焦点を絞ったために，CPUやI/Oバスの違いは考慮しなかった．これらを対象とするために，本論文で定義した3つの仕様のほかに，新たに「CPU仕様」や「I/Oバス仕様」の定義を導入することを検討している．

参 考 文 献

1) Tuggle, E.: Introduction to Device Driver Design, *Proc. 5th Annual Embedded Sys. Conf.*, Vol.2, pp.455–468 (1993).
2) Jensen, D.C.R., Madsen, J. and Pedersen, S.: 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) Ryan, S.J.: Synchronization in Portable Device Drivers, *ACM OS Review*, Vol.32, No.4, pp.62–69 (1998).
4) WinDK 2.7の概要. http://www.toolcraft.co.jp/toppage/product/device_toolware/windk/windk.htm
5) 奥野幹也，片山徹郎，最所圭三，福田 晃：デバイスドライバ生成システムにおける入力の改良，情報処理学会コンピュータシステム・シンポジウム，pp.137–144 (1999).
6) FreeBSD Inc: http://www.freebsd.org/
7) Linux Online: http://www.linux.org/
8) Katayama, T., Saisho, K. and Fukuda, A.: 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).
9) Katayama, T., Saisho, K. and Fukuda, A.:

Proposal of a Support System for Device Driver Generation, *Proc. 1999 Asia-Pacific Softw. Eng. Conf.* (*APSEC'99*), pp.494–497 (1999).
10) 奥野幹也，片山徹郎，最所圭三，福田 晃：OS 間の差異を吸収するデバイスドライバ自動生成システムの設計，情報処理学会研究報告，99-OS-82, pp.25–32 (1999).
11) I_2O SIG: http://www.i2osig.org/
12) Wilner, D.: I_2O's OS Evolves, *BYTE, Int. Ed.*, Vol.23, No.4, pp.47–48, McGraw-Hill (1998).

(平成 11 年 12 月 14 日受付)
(平成 12 年 4 月 6 日採録)

奥野 幹也

1975 生．1998 年大阪府立大学工学部情報工学科卒業．2000 年奈良先端科学技術大学院大学情報科学研究科博士前期課程修了．同年，サン・マイクロシステムズ（株）入社，現在に至る．工学修士．ニューラルネットワーク，システムソフトウェアの研究に従事．

片山 徹郎（正会員）

1969 年生．1991 年九州大学工学部情報工学科卒業．1993 年同大学院情報工学専攻修士課程修了．1996 年同大学院博士後期課程修了．同年奈良先端科学技術大学院大学情報科学研究科助手，現在に至る．工学博士．ソフトウェア工学，システムソフトウェアの研究に従事．電子情報通信学会，ソフトウェア科学会各会員．

最所 圭三（正会員）

1959 年生．1982 年九州大学工学部情報工学科卒業．1984 年同大学院工学研究科修士課程修了．同年同大学工学部助手．1991 年同大学工学部講師．1993 年同大学大型計算機センター助教授．1994 年奈良先端科学技術大学院大学情報科学研究科助教授，2000 年香川大学工学部信頼性情報システム工学科教授，現在に至る．工学博士．高信頼性システム，並列/分散処理，モバイルシステム，並行処理等の研究に従事．1998 年情報処理学会全国大会大会優秀賞受賞．電子情報通信学会会員．

福田 晃（正会員）

1954 年生．1977 年九州大学工学部情報工学科卒業．1979 年同大学院工学研究科修士課程修了．同年 NTT 研究所入所．1983 年九州大学大学院総合理工学研究科助手．1989 年同大学助教授．1994 年より奈良先端科学技術大学院大学情報科学研究科教授，現在に至る．工学博士．オペレーティング・システム，並列化コンパイラ，計算機アーキテクチャ，並列/分散処理，性能評価等の研究に従事．本学会平成 2 年度研究賞，平成 5 年度 Best Author 賞受賞．著書「並列オペレーティングシステム」（コロナ社），訳書「オペレーティングシステムの概念」（共訳，培風館）．AMC，IEEE Computer Society，電子情報通信学会，日本 OR 学会各会員．