システムソフトウェアと
オペレーティング・システム 82－4
（１９９９．８．２）

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

奥野幹也　片山徹郎　最所圭三　福田晃

奈良先端科学技術大学院大学
情報科学研究科
〒 630-0101 奈良県生駒市高山町 8916-5
E-mail:{mikiya-o,kat,sai,fukuda}@is.aist-nara.ac.jp

デバイスドライバは、対象となるデバイスとオペレーティングシステム (OS) 毎に用意しなければならない。また、作成に関しては、動作のタイミングなどのデバイスのハードウェアに関する知識が必要となるため、OS の他の部分に比べて最も時間と労力を必要とする。本研究では、デバイスドライバの仕様、デバイス依存仕様、OS 依存仕様を入力とし、デバイスドライバのソースコードを出力とするデバイスドライバ自動生成システムを設計する。入力の記述例として、OS は FreeBSD と Linux、デバイスはネットワークデバイスを対象とした。デバイスドライバインターフェイスの引数に必要なデータ構造を、OS からではなくデバイスに必要なデータから定めることにより、OS 間のデータ構造の差異を吸収する。

Design of Automatic Device Driver Generation System for Various OSs

Mikiya Okuno　Tetsuro Katayama　Keizo Saisho　Akira Fukuda

Graduate School of Information Science,
Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara 630-0101, Japan

Writing device drivers is the most difficult one among processes to develop or port operating systems(OSs) because knowledge of device hardware is required. The device drivers are needded at each device and OS. In this paper, an automatic device driver generation system is proposed. It generates a source code of the device driver by specifications: the device driver specification, the device dependent specification, and the OS dependent specification. This paper also describes inputs of the system to generate a device driver of network devices for FreeBSD and Linux. It is solved that difference of data structures between OS's device driver interfaces from a viewpoint of not OS dependent data structure but devices requiring data.

1 はじめに

デバイスドライバの作成は、OS の開発や移植において最も時間と労力を要する部分である [1, 2]。これには、以下の理由が挙げられる。

- 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。
- デバイスドライバの作成は、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するので、多大な労力を要する。

近年、マルチメディアやインターネット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。

そこで本稿では、デバイスドライバ作成にかかる労力を軽減することを目的として、デバイスドライバ自動生成システムを設計する。2 章では、デバイスドライバ自動生成システムについて述べる。3 章では、入力の記述例として、OS には FreeBSD および Linux 、デバイスとしては、ネットワークデバイスのデバイスドライバの関数を取り上げる。4 章では、議論および評価について述べる。

2 デバイスドライバ生成システムの概要

現状のデバイスドライバは、使用する OS とデバイスに合わせて作成する。このため、デバイス、もしくは OS のいずれかが変更されるたびに、それぞれに対応したデバイスドライバを作成しなければならない。さらに、同じサービスを提供するデバイスであっても、使用されているチップ (コントローラ) が異なれば、それに合った新たなデバイスドライバを作成しなければならない (図 1 参照)。

そこで、デバイスドライバを、デバイスの機能を示した部所、デバイスのハードウェアに依存した部所、OS に依存した部所にそれぞれ分離する [3, 4]。具体的には、以下の 3 つに分ける。

- デバイスドライバの仕様

  デバイスの機能を表す。具体的には、生成するデバイスドライバがどのような関数を用意してどのデバイスを使用するか、および、デバイスドライバ内で使用するデータ構造や関数内のコードを記述する。

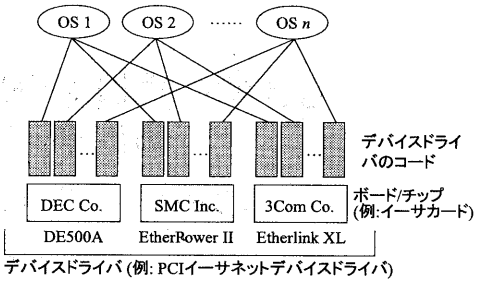

図 1: 現在のデバイスドライバ開発

- OS 依存仕様

  OS に依存する情報を記述する。具体的には、OS が提供するデバイスドライバインターフェイスの名前や引数、返り値などを記述する。

- デバイス依存仕様

  デバイスドライバの仕様で記述されたデバイスの機能の中で、各デバイスのハードウェアに依存する情報を記述する。

上記の 3 つの仕様に基づいて、図 2に示すデバイスドライバ生成支援システムを提案する。デバイスドライバの仕様、デバイス依存仕様、および、OS 依存仕様を、デバイスドライバ自動生成システムに入力することにより、それぞれの仕様を満たすソースコードを生成することを目指している。

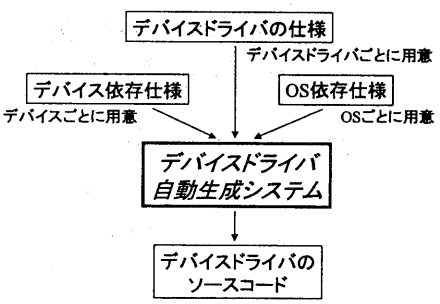

図 2: デバイスドライバ自動生成システムの概要

このシステムが実現できれば、図 3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 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。

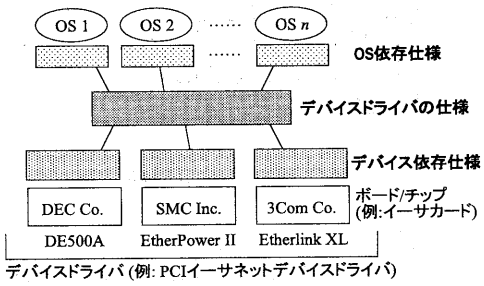

図 3: 自動生成システム用いた開発

次節では、具体的なデバイスの例を取り上げて、デバイスドライバ自動生成システムの3つの入力である、デバイスドライバの仕様とデバイス依存仕様、OS 依存仕様のそれぞれについて述べる。

3 デバイスドライバの生成手法

本稿では、対象とするOSとして、FreeBSD[5] および Linux[6] を選んだ。これは、2つのOSが代表的な PC UNIX であり、ソースが公開されているため、デバイスドライバのソース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ーフェイスは、イーサネットとした。これは、現在最も普及しており、開発スピードも早く、デバイスドライバの生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、以下の代表的な PCI(Peripheral Componet Interface) のイーサネットカード3種類とした。

- DEC 社製 DE500A
- 3Com 社製 Etherlink XL
- SMC 社製 EtherPower II

また、CPU 仕様は x86 系の CPU を対象とする。これは一般に x86 系の CPU が多く使われていることが理由として挙げられる。I/O バス仕様は、PCI を対象とする。これは、CPU と同様に普及していることと、デバイスドライバのソースコードが参照しやすいことによる。

デバイスは OS から、デバイスドライバインターフェイスを用いて取り扱われる。すなわち、デバイスドライバインターフェイスにより、デバイスは抽象化されている。本稿で扱う FreeBSD ではネットワークデバイスの機能として、デバイスドライバインターフェイスを表1に示す 12 個の機能に抽象化している。すなわち、ネットワークインターフェイスの機能を、表1に示す 12 個の機能に定めることができる。

表 1: FreeBSD が用意しているネットワークデバイスのデバイスドライバインターフェイス

関数名	機能
probe	デバイスの検出
attach	OS へデバイスを登録
output	送信パケットからイーサネットフレームの構築
start	イーサネットフレームの送信処理
init	デバイスの初期化
ioctl	メディアなどのデバイスの制御全般
watchdog	タイマ
poll_recv	ポーリングを使用した受信処理
poll_xmit	ポーリングを使用した送信処理
poll_intren	ポーリングを使用した割り込み処理
poll_slowinput	低速デバイスに対する受信処理
done	送信処理終了

デバイスドライバを生成するためには、この 12 個の中から、デバイスが持っている機能をすべて関数として生成する必要がある。本稿では、その 12 個の中から、パケットを送信する start 関数を例にとり、システムの入力について述べる。FreeBSD、NetBSD[7]、OpenBSD[8] には start 関数が存在する。Linux では hard_start_xmit 関数がその機能を果たす。

3.1 システムへの入力

以下に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 start 関数を例にとり説明していく。

3.1.1 デバイスドライバの仕様

デバイスドライバの仕様には start 関数の機能を記述する。具体的には、start 関数の入出力や、処理の流れが相当する。

start 関数にはイーサネットフレームが入力として与えられる。ここで、イーサネットフレームを格納するデータ構造を定めなければならない。このデータ構造には、BSD 系 UNIX で採用されている mbuf 構造体に定め、そのまま利用することとした (図 4参照)。これは、mbuf 構造体がシンプルで、パケット送信という機能においては、十分だと判断したからである。

従って、start 関数の入出力は以下のようになる。

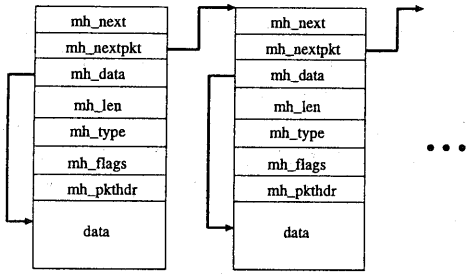

図 4: mbuf 構造体

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

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

図 5: start 関数のデバイスドライバの仕様の記述例

- 入力 — mbuf 構造体
- 出力 — ネットワークに送信するデータ

start 関数の処理には、大きく分けて以下の 2 つの処理が行なわれている。

- バッファに送信データを詰め込む処理
- 送信処理

デバイスドライバの仕様は、その関数の中で行なう処理を C 言語の仕様に沿って記述する。処理の内容は C 言語の関数の形を用いて記述する。記述の仕方は、C 言語における関数のプロトタイプの部分を書かないだけである。あとは、通常の C 言語の文法に従う [4]。

start 関数における上述した二つの処理は、C 言語の関数として扱う。図 5にデバイスドライバの仕様の記述例を示す。

3.1.2 OS 依存仕様

OS 依存仕様は、OS 側が要求するデバイスドライバインターフェイスを記述し、以下で表す。

- 引数 — 送信するデータとデバイス情報
- 返り値 — static void
- 名前 — <device_name>_start

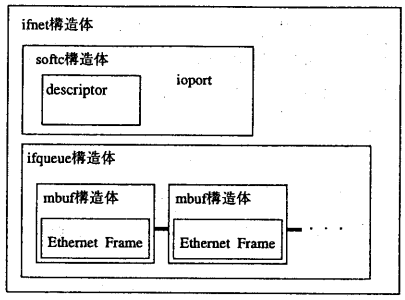

図 6: ifnet 構造体 (OS 依存仕様)

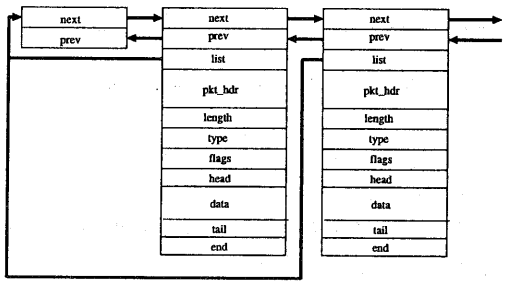

図 7: sk_buff 構造体

送信するデータとデバイス情報は、FreeBSD においては ifnet 構造体が相当する。ifnet 構造体は、大きく分けて、デバイスに関する情報を記述した softc 構造体と送信するデータが記述されている mbuf 構造体で構成されている (図 6参照)。FreeBSD が ifnet 構造体を引数にとるのに対し、Linux では sk_buff 構造体 (図 7参照) と device 構造体の 2 つの引数を取る。sk_buff 構造体は mbuf 構造体に相当し、device 構造体は softc 構造体に相当する。

FreeBSD と Linux の OS 依存仕様の記述例を、それぞれ図 8と図 9に示す。本稿では、システムに入力するデータ構造に mbuf 構造体を採用したため、図 8に示す FreeBSD では関数の名前、引数、返り値、つまり関数のプロトタイプ宣言を記述しているだけである。これに対し、図 9に示す Linux ではプロトタイプ宣言に加え、sk_buff 構造体と device 構造体から mbuf 構造体への変換処理が含まれている。

インターフェイスは C 言語のプロトタイプ宣言と同じ書式である。%<NAME> の部分に、インターフェイスの名前が入る。インターフェイスの名前は、自動生成システムの起動時に引数として与える。

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

図 8: FreeBSD の OS 依存仕様の記述例

```
static void %<NAME>_tx(struct sk_buff * skb,
struct device *dev)
{
        ifnet *ifp make_ifnet(skb, device);
        %<NAME>_start(ifp);
        free(ifp);
}
```

図 9: Linux の OS 依存仕様の記述例

mbuf 構造体も sk_buff 構造体も送信データを格納しておくという点では同じであり、構造体がリスト構造になっているという点も共通である。これらの構造体は、パケット送信という面から見ると、2 つの部分に分けることができる。

- デバイスがパケットを送信するのに必要とするデータ
- それ以外のデータ

デバイスがパケット送信時に必要とするデータは、デバイスドライバがデバイスに対して渡すデータのことである。パケット送信時では、 descriptor に詰め込むデータに当たる (descriptor の内容については、次節、デバイス依存仕様で詳しく述べる)。 descriptor 自体もリスト構造になっており、それぞれのバッファには、送信するデータそのものと、データ長、 descriptor の状態など、送信に必要なものだけが格納されている。

それ以外のデータには、例えば、sk_buff 構造体は、図 7に書かれているデータ以外に、タイムスタンプやチェックサム、セキュリティに関するデータなどが含まれている。 mbuf 構造体はシンプルな構造体であり、内容はほぼ図 4に表されているものだけである。

つまり、デバイスから要求されるデータはどちらの構造体でも等価であり、その等価な部分を抽出する機構が OS 依存仕様に含まれる。図 10に mbuf 構造体と sk_buff 構造体の等価な部分を示す。図において、双方向矢印が等価なデータを示している。

2 つの構造体の対応している等価な部分では、違うのは名前だけで、本質的なデータの意味、つまり、使用目的という面では何も変わらない。このように、等価な部分を利用することにより、データ構造の違いを吸収できると考えている。

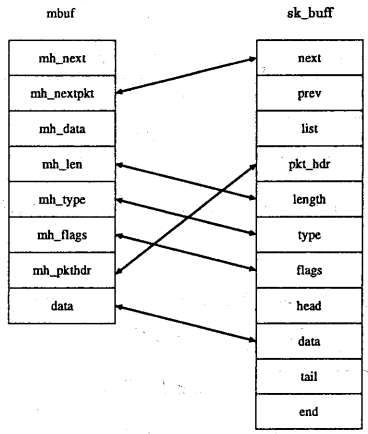

図 10: 2 つの構造体の比較

3.1.3 デバイス依存仕様

以下に、デバイスドライバの仕様で与えたそれぞれの処理におけるデバイス依存仕様を述べる。

バッファに送信データを詰め込む処理 イーサネットコントローラに渡すバッファに、送信データを詰め込む処理を行なう。この処理では以下の仕様が必要となる。以下で descriptor とは、バッファを構成する 1 つの単位であり、descriptor を複数繋げることによりバッファが構成されている。

- イーサネットコントローラに渡す descriptor の構造
  - **status** descriptor の状態
  - **control** descriptor の制御
  - **next descriptor address** descriptor リスト構造であるなら次の descriptor への物理メモリアドレス
  - **data address** 送信するイーサネットフレームを指す物理メモリアドレス
  - **data length** 送信するイーサネットフレームの長さ
  - **length** 送信するイーサネットフレームからヘッダ部分を除いた長さ
- バッファの長さ
- バッファの最初および最後を表す記号

デバイス依存仕様として、DE500A の descriptor 構造を図 11に、バッファを構築する際に必要なデータを表 2に示す。

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

図 11: DE500A の descriptor(デバイス依存仕様)

表 2: 各イーサネットコントローラのバッファ構造に入力するデータ (デバイス依存仕様)

	DE500A	EtherPowerII	Etherlink XL
status	0x80000000	0x8000	0x80000000 最後のdescriptorのみ 1つの descriptor で送信する 総バイト長 & 0x80000000
control	0x000c 最初のdescriptorのみ 0x008c 最後のdescriptorのみ 0x0300	バッファに既に半分 以上データがあるか yes 0x14 no 0x10	×
data address	1つのdescriptor 当たり 32bit ×2	1つのdescriptor 当たり 32bit ×1	1つのdescriptor 当たり 32bit ×63
data length	11 bit	16bit	32bit 最後のdescriptorのみ 0x80000000
Ethernet Frame length	×	×	○
next descriptor address	×	○	○ 最後のdescriptorのみ 0
バッファの長さ	128	16	16

送信処理 バッファに詰め込んだデータを、ネットワークに送信する処理を行なう。送信処理では以下の仕様が必要となる。

- コマンド発行のタイミング
- コマンドの値
- コマンドレジスタ
- コマンドの組合せ

具体的に、各イーサネットコントローラの送信処理の仕様は表3で与えられる。表3において、$\mathrm{out}(p, x)$ とあるのは、ポート p に値 x を書き込むことを表す。$\mathrm{in}(y)$ はポート y から値を読み出すことを表す。$\mathrm{wait}(z)$ は z 時間だけ待つことを表す。

デバイス依存仕様の表記は C 言語とほぼ同じである。違いは、ハードウェアに対して入出力を行なう命令、タイミング制御の命令、および、各機能ごとの記述のみである。各機能は “機能名:\{処理の内容\}” で記述する。start 関数で用意している機能は、Buffer Operation と Command である。

表 3: 各イーサネットコントローラの送信処理 (デバイス依存仕様)

DE500A	EtherPowerII	Etherlink XL

out (ioport + 4 ,1)
 	
out (ioport + 0x1c,
0x00000001)
	out (ioport,0x04)	out (ioport + 0x0e,0x3002)
		for(i=0; i< 1000 ;i++){
		wait(10ms);
		if(! in (ioport + 0x0E) & 0x1000)
		break;
		}
		if(in (ioport + 0x24)){
		現在のバッファのリストに
		新規データのリストを追加
		新規データの
		status &= ~0x80000000
		}
		else{
		out (ioport + 0x24,
		新規バッファの先頭アドレス)
		}
		out (ioport + 0x0e, 0x3001)
		out (ioport + 0x0e, 0x0807)

以下は、入出力を行なう命令と、タイミング制御の命令である

- out$N$ (port, data)
  port に N ビット の data を書き込む。
- in$N$ (port, data)
  port に N ビット の data を読み込む。
- wait(Z)
  Z μs だけ待つ。

図 12に DE500A のデバイス依存仕様の記述例を示す。また、DE500A における FreeBSD の start 関数の流れを、図 13に示す。

4 議論および評価

4.1 デバイスドライバ自動生成システムの有効性について

本稿で述べたデバイスドライバ自動生成システムは、入力として、デバイスドライバの仕様、デバイス依存仕様、OS 依存仕様を必要とする。システムの入力となるこれら 3 つの仕様をすべて一から記述する場合には、デバイス依存仕様の記述がその作業の大半を占める。

従って、デバイスを入れ換える場合には、最も労力を要するデバイス依存仕様を記述しなければならないため、労力削減に対して、あまり効果的ではない。しかしながら、異なる OS への適用の際には、デバイス依存仕様が再利用できるため、労力を削減でき、非常に有効である。

```
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--;
        }

      tulip_ringinfo_t * const ri = &sc->tulip_txinfo;
      tulip_desc_t *eop, *nextout;
      int segcnt = 0, free;
      u_int32_t d_status;
      struct mbuf *m0;

      again:
      d_status = 0;
      eop = nextout = ri->ri_nextout;
      m0 = m;
      free = ri->ri_free;
      do {
        int len = m0->m_len;
        caddr_t addr = mtod(m0, caddr_t);
        unsigned clsize = CLBYTES -
          (((u_long) addr) & (CLBYTES-1));

        while (len > 0) {
          unsigned slen = min(len, clsize);
          segcnt++;

          if (segcnt > TULIP_MAX_TXSEG) {
            break;
          }
          if (segcnt & 1) {
            if (--free == 0) {
              if ((free += tulip_tx_intr(sc)) == 0) {
                  sc->tulip_flags |= TULIP_WANTTXSTART;
                  return;
              }
            }
          eop = nextout;
          if (++nextout == ri->ri_last)
            nextout = ri->ri_first;

              eop->d_flag &=
TULIP_DFLAG_ENDRING|TULIP_DFLAG_CHAIN;
              eop->d_status = d_status;
              eop->d_addr1 = TULIP_KVATOPHYS(sc, addr);
              eop->d_length1 = slen;
            } else {
              eop->d_addr2 = TULIP_KVATOPHYS(sc, addr);
              eop->d_length2 = slen;
            }
            d_status = TULIP_DSTS_OWNER;
            len -= slen;
            addr += slen;
            clsize = CLBYTES;
          }
        } while ((m0 = m0->m_next) != NULL);

        (m)->m_nextpkt = 0;
        if ((&sc->tulip_txq)->ifq_tail == 0)
            (&sc->tulip_txq)->ifq_head = m;
        else
            (&sc->tulip_txq)->ifq_tail->m_nextpkt = m;
        (&sc->tulip_txq)->ifq_tail = m;
        (&sc->tulip_txq)->ifq_len++;

      m = NULL;

        nextout->d_status = 0;

        if (segcnt & 1) {
          eop->d_addr2 = 0;
          eop->d_length2 = 0;
        }

        eop->d_flag |=
TULIP_DFLAG_TxLASTSEG|TULIP_DFLAG_TxWANTINTR;

        ri->ri_nextout->d_flag |= TULIP_DFLAG_TxFIRSTSEG;
        ri->ri_nextout->d_status = TULIP_DSTS_OWNER;

\}

Command:\{

        tulip_softc_t * const sc = TULIP_IFP_TO_SOFTC(ifp);

        out32(sc->iobase + 4, 1);
        out32(sc->iobase + 0x1c, 0x00000001);

\}
```

図 12: DE500A のデバイス依存仕様の記述例

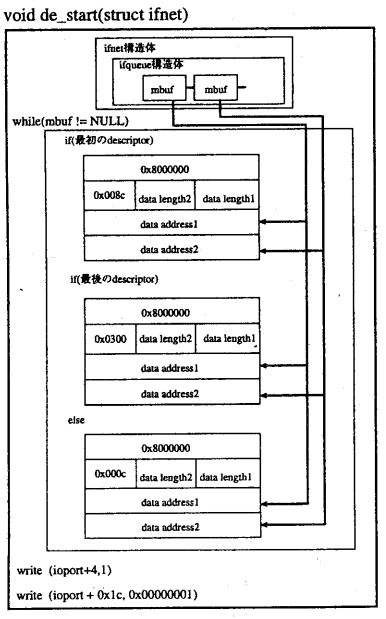

図 13: DE500A における FreeBSD の start 関数の流れ

4.2 I_2O との比較

I_2O(Intelligent Input Output) SIG[9] において、OS とデバイスドライバとの間に標準インターフェイス 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 Messanger の 3 つの階層に分ける (図 14参照)。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、一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 [10]。

I_2O の問題として、デバイスドライバを階層構造にしたことによるオーバーヘッドがある。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 で行なうものではなく、作成する段階で行なうため、このような問題は起こらない。

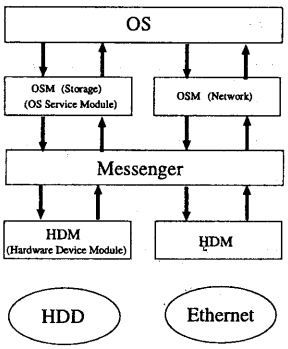

図 14: I_2O のコンセプト

4.3 OS 依存仕様の記述について

本稿では、sk_buff 構造体と mbuf 構造体の構造が、比較的近いことを利用して、OS 依存仕様を記述した (図8、図9参照)。しかしながら、新規に作る OS への適用など、OS 間のデータ構造がまったく異なる場合には、その差異を吸収するために、OS 依存仕様の記述を拡張することが必要となる。

図 15は、拡張した OS 依存仕様の記述例を示している。図中において、#START はラベルであり、start 関数の OS 依存仕様の記述の始まりであることを示している。3つの %[ ]% で囲まれたフィールドは、それぞれ以下のことを表記している。

- **function** — 関数が実現する機能
- **function_prototype** — 関数のプロトタイプ
- **data_definition** — データの定義

1番めのフィールドは、関数が実現する機能を示す。2番めのフィールドは、現行のシステムと同じで、関数のプロトタイプ、つまり、引数の型、戻り値、関数名を記述している。3番めのフィールドはデータ型の定義である。ここには、必要なデータがどこにどのような形で納められているかを記述する。ここで、n 番目の引数は、%<n> で表す。図中では、%<1>->mh_next が ifp->mh_next に対応する。

現在、このように OS 依存仕様の記述を拡張し、様々な OS 間の差異を吸収することを検討している。また、他の関数にも拡張することを目指している。

```
#START
%[function
send packet
]%

%[function_prototype
static void %<name>_start(struct ifnet *const ifp);
]%

%[data_definition
next: %<1>->mh_next
len: %<1>->mh_len
data: %<1>->mh_data
type: %<1>->mh_type
flags: %<1>->mh_flags
]%
```

図 15: 拡張した OS 依存仕様の記述例

5 まとめ

本稿では、FreeBSD および Linux 両用のデバイスドライバ自動生成システムを設計した。システムは OS 依存仕様、デバイスドライバの仕様、デバイス依存仕様の3つを入力とし、その3つを満たすデバイスドライバのソースコードを出力する。デバイスとして、PCI イーサネットカードを対象とし、OS としては、FreeBSD および Linux を対象とした。

本稿で提案したデバイスドライバ自動生成システムは、現在まだ設計の段階である。今後の課題として、実装が挙げられる。FreeBSD への実装は、システムで使用するデータ構造として mbuf 構造体を使用したことから、実装済みの部分もある。今後は Linux への実装を中心に研究を進める予定である。

参考文献

[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] 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.

[4] 山下勝也, 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形式に関する考察," 情処研報, 98-OS-79, pp.61-68, 1998.

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

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

[7] NetBSD Project: http://www.netbsd.org/

[8] OpenBSD Project: http://www.openbsd.org/

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

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