フルスクラッチで作る!
x86_64 自作 OS
パート 4

ぼくらのイーサネットフレーム!

大神祐真　著

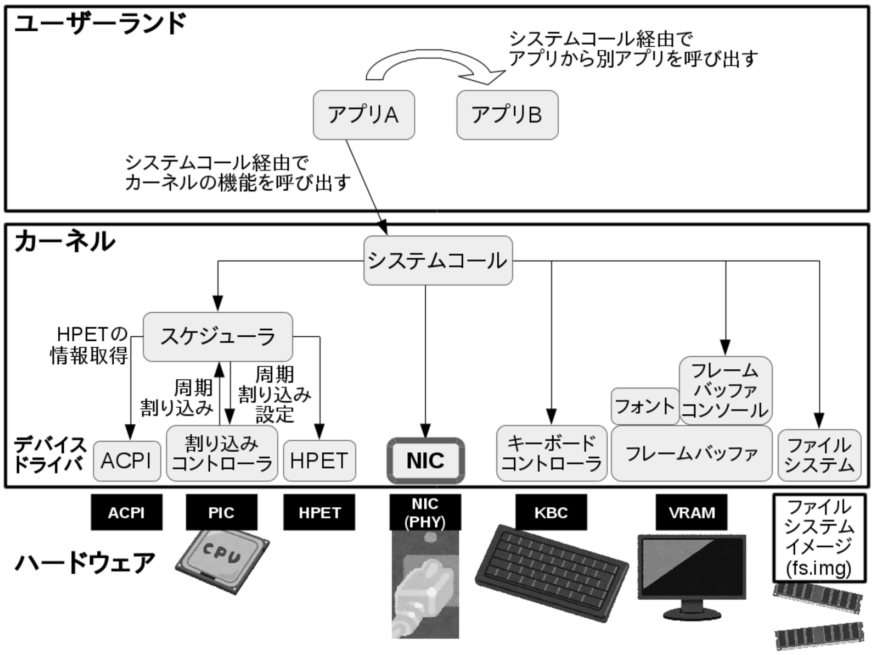
2019-04-14 版　　へにゃぺんて 発行

はじめに

本書をお手に取っていただきありがとうございます。

本書は自作 OS に NIC ドライバを実装し、イーサネットフレームの受信とオレオレイーサネットフレームの送信を行ってみる本です。

本書で NIC ドライバを搭載することで自作 OS のアーキテクチャとしては図 .1 のようになります。



▲図 1　本書まででできあがる OS のアーキテクチャ図

本書の構成

本書は以下の 2 章構成です。

第 1 章で、まず PCI から NIC を制御するためのレジスタ情報を取得します。

第 2 章では、NIC のレジスタを通してイーサネットフレームの受信と、オレオレイーサネットフレームの送信を試します。

なお、本書ではイーサネットフレームのフォーマットの解釈にまでは踏み込みません。あくまでも NIC のドライバとして、NIC が受信したイーサネットフレームのバイナリ列をドライバ側で用意したバッファに格納することと、ドライバ側で用意したバッファから独自のバイナリ列を「オレオレイーサネットフレーム」として NIC からの送信ができれば OK とします。

また、この本では筆者の PC を対象に、筆者が NIC のデバドラを作る上で行ったことをまとめています。対象マシンの NIC を把握するところから始めていますので、ご自身で試す際は同様のやり方でご自身の NIC 向けデバドラを書いてみてください。とは言え、本書で扱う範囲は Intel 製 NIC の共通仕様の範囲内なので、他のマシンでも大差は無いと思います。

開発環境・動作確認環境

筆者が開発や動作確認を行っている環境を以下に記載します。

- PC: Lenovo ThinkPad E450
  - NIC: Intel I218-V
- OS: Debian GNU/Linux 8.11 (コードネーム: Jessie)
- コンパイラ: GCC 4.9.2
  - build-essential パッケージでインストールされるものです

言語としては、C 言語と一部アセンブラを使う程度なので、一般的な Linux 環境であれば問題ないと思います。

本書の PDF 版/HTML 版やソースコードの公開場所について

これまでの当サークルの同人誌同様、本書も PDF 版/HTML 版は以下のページで無料で公開しています。

- `http://yuma.ohgami.jp`

サンプルコードは以下の GitHub リポジトリで公開しています。

- `https://github.com/cupnes/x86_64_jisaku_os_4_samples`

サンプルコードのリポジトリ内はディレクトリで分かれています。各ディレクトリが以降の各項あるいは各節で作るものです。各項/節の冒頭で「この項/節のサンプルディレクトリは XX です。」と紹介しますので適宜参照してみてください。

なお、ビルド方法は本書のパート 1 を参照してください。また、パート 2 でブートローダーにも手を加えています。実行の際はブートローダーも必要なので、そちらも併せてご参照ください。

付録サンプルコードについて

「A」から始まるディレクトリは参考用の付録サンプルコードです。今回は「A00」から「A02」の大きく 3 つの付録があります。

- A00_kern_app_base
  - 本書開始時点のソースコードを格納したディレクトリです
  - 本書開始時点からの各サンプルの差分確認等のために置いています (「diff -r」コマンドでディレクトリ間の差分を確認できます)
- A01_syscall
  - イーサネットフレーム受信/送信のシステムコール実装例です
  - 本書ではカーネル側で実装したイーサネットフレーム送受信の機能をシステムコール化するところまでは扱いませんので参考としてサンプルコードを置いておきます
  - カーネル側 (syscall.c) にて、本書でこれから実装するイーサネットフレーム受信関数と送信関数をシステムコール化しています
  - また、それらのシステムコールを呼び出すサンプルとして「06_beef_server」を apps ディレクトリに追加しています
    - イーサネットフレームを受信するとそれを画面にダンプし、「0xbeefbeef」というオレオレイーサネットフレームを NIC から送出するサンプルアプリです
  - システムコールの実装方法について詳しくは前著 (パート 3) をご覧ください
- A02_get_mac_01/02
  - NIC の MAC アドレス取得の実装例です
  - こちらについては本書の最後に「付録」の章を用意していますので、そちらをご覧ください

目次

第 1 章

PCI から NIC の情報を取得する

1.1 この章でやること

NIC は PCI で接続されており、NIC を制御するための情報は PCI の管理領域 (PCI コンフィグレーション空間) に並ぶレジスタにあります。この章では、NIC を制御するための情報を PCI コンフィグレーション空間内のレジスタから取得する方法を、以下の流れで紹介します。

1. 自作 OS を動作させる対象のマシン上で lspci により NIC を確認
2. 1. で確認した内容をキーとして、NIC を制御するための情報を PCI コンフィグレーション空間から取得

1.2 lspci コマンドで NIC を確認

lspci は PCI の情報を取得する Linux のコマンドです。この項では、自作 OS を動作させる対象のマシン (VM 含む) 上で Linux を起動し、lspci コマンドを使って PCI のデバイス情報を取得し、マシンに搭載されている NIC を把握します。

本書では一応、想定する開発環境を Linux(Debian) としているので、「開発マシン＝自作 OS を実行するマシン」の場合、そのマシン上で開発環境を起動させてください。「開発マシンと自作 OS を実行するマシンが異なる」場合、実行マシンに Linux がインストールされていない場合は、Ubuntu 等のライブディスク/USB で Linux をライブ起動してみてください。

それでは、lspci を実行してみます (図 1.1)。

```
➜ seven ~  lspci
00:00.0 Host bridge: Intel Corporation Broadwell-U Host Bridge -OPI (rev 09)
00:02.0 VGA compatible controller: Intel Corporation Broadwell-U Integrated Graphics (r
00:03.0 Audio device: Intel Corporation Broadwell-U Audio Controller (rev 09)
00:14.0 USB controller: Intel Corporation Wildcat Point-LP USB xHCI Controller (rev 03)
00:16.0 Communication controller: Intel Corporation Wildcat Point-LP MEI Controller #1
00:19.0 Ethernet controller: Intel Corporation Ethernet Connection (3) I218-V (rev 03)
```

▲図 1.1　lspci で PCI のデバイスを一覧表示

図 1.1 のスクリーンショットの最下行の"00:19.0 Ethernet controller: Intel Corporation Ethernet Connection (3) I218-V (rev 03)"が筆者の PC の NIC です。

PCI コンフィグレーション空間から目的のデバイスの情報を取得するには、そのデバイスの「バス番号」・「デバイス番号」・「ファンクション番号」が分かっていれば良いです。そして、それは先程の出力結果の"00:19.0"に当たります。先頭の"00"が「バス番号」、真ん中の"19"がデバイス番号、最後の"0"がファンクション番号です。なお、これらの値は 16 進数表記なので、"19"は"0x19"です。

これで、PCI コンフィグレーション空間から NIC の情報を読むために必要な情報は揃ったのですが、自作のコードで PCI コンフィグレーション空間を読んだ結果が正しいことを確認したいので、先に lspci で PCI コンフィグレーション空間を読んでみます。

lspci で PCI コンフィグレーション空間の情報も表示させるために、"-vx"オプションを指定してみます。"v"オプションが各デバイスの PCI コンフィグレーション空間の内容を解釈した結果を表示するオプションで、"x"が PCI コンフィグレーション空間の先頭 64 バイトを 16 進ダンプするオプションです。

試しに実行してみると図 1.2 の通りです。見たいデバイスは決まっているので、図 1.2 では"-s"オプションで先ほど確認した NIC のバス番号・デバイス番号・ファンクション番号を指定しています。

```
➜ seven ~  lspci -vx -s 00:19.0
00:19.0 Ethernet controller: Intel Corporation Ethernet Connection (3) I218-V (rev 03)
        Subsystem: Lenovo Device 5020
        Flags: fast devsel, IRQ 64
        Memory at f1300000 (32-bit, non-prefetchable) [disabled] [size=128K]
        Memory at f133e000 (32-bit, non-prefetchable) [disabled] [size=4K]
        I/O ports at 4080 [disabled] [size=32]
        Capabilities: <access denied>
        Kernel driver in use: e1000e
00: 86 80 a3 15 00 04 10 00 03 00 00 02 00 00 00 00
10: 00 00 30 f1 00 e0 33 f1 81 40 00 00 00 00 00 00
20: 00 00 00 00 00 00 00 00 00 00 00 00 aa 17 20 50
30: 00 00 00 00 c8 00 00 00 00 00 00 00 0b 01 00 00
```

▲図 1.2　PCI コンフィグレーション空間の情報を表示

図 1.2 の後半の 16 進ダンプが PCI コンフィグレーション空間の生のデータ (の一部、

先頭 64 バイト) で、それを解釈した結果が前半部分です。

ここで確認しておきたいのは、PCI コンフィグレーション空間の先頭 4 バイトのレジスタの内容です。PCI コンフィグレーション空間の先頭 4 バイトのレジスタには「ベンダー ID(2 バイト)」・「デバイス ID(2 バイト)」が格納されています。今回の場合、16 進ダンプの"86 80"(リトルエンディアンなので 0x8086) がベンダー ID、"a3 15"(リトルエンディアンなので 0x15a3) がデバイス ID です。この 4 バイトはデバイスに一意な ID なので、デバドラ等はこの値からどのベンダーの何というデバイスなのかを判別します。

これで、NIC の PCI コンフィグレーション空間の先頭 4 バイトは「ベンダー ID: 0x8086」・「デバイス ID: 0x15a3」だと分かったので、以降で PCI コンフィグレーション空間を自作のコードで読んで見る際、この値が読めれば、読み方が間違っていないと言えます。

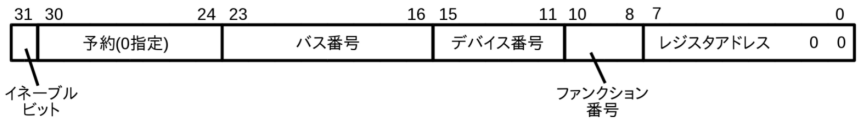
なお、表示されているその他の項目についてここで詳しく紹介はしません。「こんな感じの内容が格納されているのか」と雰囲気がつかめれば OK です[*1]。

1.3 PCI コンフィグレーション空間のアクセス方法

PCI コンフィグレーション空間は、ポインタ等でアドレス指定するようなメモリ空間とも、in/out 命令でアクセスする IO 空間とも別のアドレス空間です。PCI コンフィグレーション空間へは、「CONFIG_ADDRESS レジスタ (IO アドレス:0x0cf8 〜 0x0cfb の 4 バイトレジスタ)」へアクセスしたい PCI コンフィグレーション空間のアドレスを設定した上で、「CONFIG_DATA レジスタ (IO アドレス:0x0cfc 〜 0x0cff の 4 バイトレジスタ)」へ read/write することで、CONFIG_ADDRESS レジスタへ指定した PCI コンフィグレーション空間のレジスタを read/write できます。

そして、CONFIG_ADDRESS に指定する PCI コンフィグレーション空間のアドレスのフォーマットは図 1.3 の通りです。前項で確認した「バス番号」・「デバイス番号」・「ファンクション番号」でどのデバイスの PCI コンフィグレーション空間かが決まり、「レジスタアドレス (PCI コンフィグレーション空間先頭からのオフセット)」で PCI コンフィグレーション空間のどのレジスタかが決まります。

[*1] 一部で読む為にはスーパーユーザー権限が必要な項目もあり、"Capabilities"には"<access denied>"と表示されています。見てみたい場合は lspci に sudo を先頭に付ける等してスーパーユーザーで実行してください。



▲図 1.3　CONFIG_ADDRESS のフォーマット

最上位ビットである「イネーブルビット」を 1 で CONFIG_ADDRESS にアドレスをセットすると、次の CONFIG_DATA へのアクセスは、PCI コンフィグレーション空間のレジスタへのアクセスとなります。

なお、「レジスタアドレス」は 4 バイトアライメント (4 の倍数) で指定しますので、レジスタアドレスの下位 2 ビットは常に 0 です。そのため、例えばデバイス ID(レジスタアドレス:0x02) にアクセスしたい場合に、CONFIG_ADDRESS のレジスタアドレスに 0x02 を指定することはできません。その場合、レジスタアドレスには 0x00 を指定して、CONFIG_DATA レジスタを 0x0cfe(CONFIG_DATA レジスタの上位 2 バイト) から 2 バイト読み出すようにするか、CONFIG_DATA レジスタは 0x0cfc から 4 バイト読み出した後で上位 2 バイトを取り出すようにします。

また、CONFIG_ADDRESS レジスタは 32 ビットアクセス限定です。CONFIG_ADDRESS の下位 1 バイトに当たるレジスタアドレスだけを書き換えようと 0x0cfb に対して 8 ビットアクセスで書き込みを行ったとしても、それは CONFIG_ADDRESS ではなく、同じ IO アドレスにマップされた別のレジスタへのアクセスになります。

1.4　PCI コンフィグレーション空間から NIC のベンダー ID・デバイス ID を読んでみる

CONFIG_ADDRESS・CONFIG_DATA を使用して NIC のベンダー ID・デバイス ID を読んでみます。それぞれのレジスタの使い方は前項で説明した通りなので、早速サンプルコードを紹介します。

この項のサンプルディレクトリは「011_dump_vid_did」です。

1.4.1　io_read32()/io_write32() を追加する

まず、32 ビット単位の in/out 命令を実行する io_read32() と io_write32() を追加します (リスト 1.1)。少なくとも CONFIG_ADDRESS レジスタへは 32 ビットアクセス

でなければならないので、CONFIG_DATA も必要がない限り 32 ビットでアクセスするようにします。

▼**リスト** 1.1　011_dump_vid_did/x86.c

```
/* ・・・ 省略 ・・・ */

inline void io_write(unsigned short addr, unsigned char value)
{
        asm volatile ("outb %[value], %[addr]"
                      :: [value]"a"(value), [addr]"d"(addr));
}

/* 追加（ここから） */
inline unsigned int io_read32(unsigned short addr)
{
        unsigned int value;
        asm volatile ("inl %[addr], %[value]"
                      : [value]"=a"(value) : [addr]"d"(addr));
        return value;
}

inline void io_write32(unsigned short addr, unsigned int value)
{
        asm volatile ("outl %[value], %[addr]"
                      :: [value]"a"(value), [addr]"d"(addr));
}
/* 追加（ここまで） */

void gdt_init(void)
/* ・・・ 省略 ・・・ */
```

バイト (8 ビット) 単位の io 命令 (inb/outb) を使うこれまでの io_read()/io_write() を元に、32 ビット単位の io 命令 (inl/outl) を使う関数「io_read32()」・「io_write32()」を作成しました。

リスト 1.1 の変更に併せて、include/x86.h へ io_read32() と io_write32() のプロトタイプ宣言を追加しておいてください。(コードを引用しての説明は省略します。)

1.4.2　定数を定義する

ここでは動作確認として main.c に処理を追加してみます。

まず、使用する定数等を定義します (リスト 1.2)。

▼**リスト** 1.2　011_dump_vid_did/main.c

```
/* ・・・ 省略 ・・・ */
#define INIT_APP       "test"


/* 追加（ここから） */
/* PCI の定義 */
#define PCI_CONF_DID_VID       0x00

#define CONFIG_ADDRESS 0x0cf8
#define CONFIG_DATA    0x0cfc

union pci_config_address {
        unsigned int raw;
        struct __attribute__((packed)) {
                unsigned int reg_addr:8;
                unsigned int func_num:3;
                unsigned int dev_num:5;
                unsigned int bus_num:8;
                unsigned int _reserved:7;
                unsigned int enable_bit:1;
        };
};


/* NIC の定義 */
#define NIC_BUS_NUM    0x00
#define NIC_DEV_NUM    0x19
#define NIC_FUNC_NUM   0x0
/* 追加（ここまで） */

/* ・・・ 省略 ・・・ */
```

定義している内容はこれまで説明してきた通りで、PCI に関する定数としては、PCI コンフィグレーション空間にアクセスするための IO アドレス (CONFIG_ADDRESS/CONFIG_DATA) と、ベンダー ID・デバイス ID の PCI コンフィグレーション空間内のオフセット (PCI_CONF_DID_VID) を定義しています。

また、CONFIG_ADDRESS に渡すデータの構造は、共用体と構造体で定義しています。(使い方は後ほど紹介します。)

そして、NIC に関してはバス番号 (NIC_BUS_NUM)、デバイス番号 (NIC_DEV_NUM)、ファンクション番号 (NIC_FUNC_NUM) を定義しています。

1.4.3 読み出す処理を実装する

続いて、ベンダー ID とデバイス ID を読み出す処理を実装します (リスト 1.3)。

▼**リスト** 1.3　011_dump_vid_did/main.c

```
/* ・・・ 省略 ・・・ */

void start_kernel(void *_t __attribute__((unused)), struct platform_info *pi,
                  void *_fs_start)
{
        /* ・・・ 省略 ・・・ */
        /* 周辺 IC の初期化 */
        pic_init();
        hpet_init();
        kbc_init();



        /* 追加（ここから） */
        /* NIC のベンダー ID・デバイス ID を取得 */

        /* CONFIG_ADDRESS を設定 */
        union pci_config_address conf_addr;
        conf_addr.raw = 0;
        conf_addr.bus_num = NIC_BUS_NUM;
        conf_addr.dev_num = NIC_DEV_NUM;
        conf_addr.func_num = NIC_FUNC_NUM;
        conf_addr.reg_addr = PCI_CONF_DID_VID;
        conf_addr.enable_bit = 1;
        io_write32(CONFIG_ADDRESS, conf_addr.raw);

        /* CONFIG_DATA を読み出す */
        unsigned int conf_data = io_read32(CONFIG_DATA);

        /* 読み出したデータからベンダー ID・デバイス ID を取得 */
        unsigned short vendor_id = conf_data & 0x0000ffff;
        unsigned short device_id = conf_data >> 16;

        /* 表示 */
        puts("VENDOR ID ");
        puth(vendor_id, 4);
        puts("\r\n");
        puts("DEVICE ID ");
        puth(device_id, 4);
        puts("\r\n");

        /* halt して待つ */
        while (1)
                cpu_halt();
        /* 追加（ここまで） */



        /* システムコールの初期化 */
        syscall_init();
        /* ・・・ 省略 ・・・ */
```

「CONFIG_ADDRESS を設定」のコードブロックで、先程定義した pci_config_address 共用体を使っています。まずメンバーの raw 変数へ 0 を代入することで conf_addr 全体をゼロクリアしています。次に、構造体部分の定義を利用して、バス番号・デバイス番号・ファンクション番号と、PCI コンフィグレーション空間

内のアクセスしたいレジスタのアドレス (PCI コンフィグレーション空間先頭からのオフセット) を conf_addr へ代入しています。最後に enable_bit に 1 をセットして、CONFIG_ADDRESS に設定します。

この状態で、CONFIG_DATA にアクセスすると、CONFIG_ADDRESS に設定した PCI コンフィグレーション空間のレジスタへアクセスできます。今回の場合、ベンダー ID・デバイス ID のレジスタを読み出しています。

読み出した 32 ビット (4 バイト) の下位 16 ビットにベンダー ID が、上位 16 ビットにデバイス ID が格納されていますので、マスクしたり、ビットシフトしたりしてそれぞれを取り出し、表示しています。

このように、PCI コンフィグレーション空間へは CONFIG_ADDRESS と CONFIG_DATA を使用してアクセスします。

1.4.4 実行してみる

実行すると図 1.4 の様に NIC のベンダー ID とデバイス ID が表示されます。

▲図 1.4　011_dump_vid_did の実行結果

前項で確認したものと同じ値が表示されているので PCI コンフィグレーション空間へのアクセスは問題無さそうです。

1.4.5 関数化しておく

この項の最後に、今後 PCI コンフィグレーション空間の読み出し等を行いやすくするために、main.c に書いていた処理を関数化しておきます。

以降では関数化等を行ったコードをベースに説明していきます。「011_dump_vid_did_refactor」というディレクトリに作業済みのコードを置いていますので必要に応じて参照してみてください。

関数化したのは PCI コンフィグレーション空間からレジスタを読み出す関数

「get_pci_conf_reg」と、それを使用してベンダー ID・デバイス ID をダンプする関数「dump_vid_did」の 2 つで、pci.c というソースファイルを新たに作成しリスト 1.4 のように関数化しました。(その他に main.c に書いていた PCI 関係の定義も pci.c へ移動しましたが、割愛します。)

▼**リスト** 1.4　011_dump_vid_did_refactor/pci.c

```
/* ・・・ 省略 ・・・ */

unsigned int get_pci_conf_reg(
        unsigned char bus, unsigned char dev, unsigned char func,
        unsigned char reg)
{
        /* CONFIG_ADDRESS を設定 */
        union pci_config_address conf_addr;
        conf_addr.raw = 0;
        conf_addr.bus_num = bus;
        conf_addr.dev_num = dev;
        conf_addr.func_num = func;
        conf_addr.reg_addr = reg;
        conf_addr.enable_bit = 1;
        io_write32(CONFIG_ADDRESS, conf_addr.raw);

        /* CONFIG_DATA を読み出す */
        return io_read32(CONFIG_DATA);
}

void dump_vid_did(unsigned char bus, unsigned char dev, unsigned char func)
{
        /* PCI コンフィグレーション空間のレジスタを読み出す */
        unsigned int conf_data = get_pci_conf_reg(
                bus, dev, func, PCI_CONF_DID_VID);

        /* 読み出したデータからベンダー ID・デバイス ID を取得 */
        unsigned short vendor_id = conf_data & 0x0000ffff;
        unsigned short device_id = conf_data >> 16;

        /* 表示 */
        puts("VENDOR ID ");
        puth(vendor_id, 4);
        puts("\r\n");
        puts("DEVICE ID ");
        puth(device_id, 4);
        puts("\r\n");
}
```

その他、主に行ったことは以下のとおりですが、main.c に書いていたものを移動させただけなのでコードを引用しての紹介は割愛します。

- Makefile の OBJS に pci.o を追加
- NIC 関連の定数は include ディレクトリに nic.h を追加し、そちらへ移動
- include ディレクトリに pci.h を追加し、PCI_CONF_DID_VID の定義と、

get_pci_conf_reg() と dump_vid_did() のプロトタイプ宣言を記載
- main.c では dump_vid_did() を呼び出すように変更

1.5 コマンドレジスタ・ステータスレジスタを確認してみる

1.5.1 PCI コンフィグレーション空間内のレジスタ配置について

PCI コンフィグレーション空間内には対象デバイスの PCI に関わる設定やステータス確認のためのレジスタとして「コマンド」レジスタと「ステータス」レジスタがあります。

前項までは PCI コンフィグレーション空間の先頭の 4 バイトしか見ていなかったので特に出していませんでしたが、ここで、PCI コンフィグレーション空間のレジスタマップを示します (図 1.5)。(本書で使用するレジスタのみに絞っています。)

▲図 1.5　PCI コンフィグレーション空間 (本書で使用するレジスタのみ抜粋)

図 1.5 の通り、ステータスとコマンドのレジスタは PCI コンフィグレーション空間の 4 バイト目です。

1.5.2 コマンド・ステータスレジスタをダンプしてみる

コマンドレジスタとステータスレジスタの内容をダンプしてみます。

この節のサンプルディレクトリは「012_dump_command_status」です。

ここでは、pci.c へ「dump_command_status」という関数を追加し、コマンド・ステー

タスそれぞれの値の 16 進ダンプを表示すると同時に、それぞれのレジスタに設定されているビット名を表示するようにしてみます。

前項の最後に、ベンダー ID とデバイス ID をダンプする関数を dump_vid_did という名前で作成したのと同じように、コマンドとステータスのレジスタの内容をダンプする関数を「dump_command_status」という名前で pci.c へ追加します。

まず、新たに使用する定義を include/pci.h へ追加します (リスト 1.5)。

▼**リスト** 1.5　012-dump_command_status/pci.h

```
#pragma once

#define PCI_CONF_DID_VID       0x00
/* 追加（ここから）*/
#define PCI_CONF_STATUS_COMMAND        0x04

#define PCI_COM_IO_EN  (1U << 0)
#define PCI_COM_MEM_EN (1U << 1)
#define PCI_COM_BUS_MASTER_EN  (1U << 2)
#define PCI_COM_SPECIAL_CYCLE  (1U << 3)
#define PCI_COM_MEMW_INV_EN    (1U << 4)
#define PCI_COM_VGA_PAL_SNP    (1U << 5)
#define PCI_COM_PARITY_ERR_RES (1U << 6)
#define PCI_COM_SERR_EN        (1U << 8)
#define PCI_COM_FAST_BACK2BACK_EN      (1U << 9)
#define PCI_COM_INTR_DIS       (1U << 10)

#define PCI_STAT_INTR  (1U << 3)
#define PCI_STAT_MULT_FUNC     (1U << 4)
#define PCI_STAT_66MHZ (1U << 5)
#define PCI_STAT_FAST_BACK2BACK        (1U << 7)
#define PCI_STAT_DATA_PARITY_ERR       (1U << 8)
#define PCI_STAT_DEVSEL_MASK   (3U << 9)
#define PCI_STAT_DEVSEL_FAST   (0b00 << 9)
#define PCI_STAT_DEVSEL_MID    (0b01 << 9)
#define PCI_STAT_DEVSEL_LOW    (0b10 << 9)
#define PCI_STAT_SND_TARGET_ABORT      (1U << 11)
#define PCI_STAT_RCV_TARGET_ABORT      (1U << 12)
#define PCI_STAT_RCV_MASTER_ABORT      (1U << 13)
#define PCI_STAT_SYS_ERR       (1U << 14)
#define PCI_STAT_PARITY_ERR    (1U << 15)
/* 追加（ここまで）*/

unsigned int get_pci_conf_reg(
/* ・・・ 省略 ・・・ */
```

「PCI_CONF_STATUS_COMMAND」でステータス・コマンドレジスタの PCI コンフィグレーション空間先頭からのオフセットを定義し、「PCI_COM_*」と「PCI_STAT_*」で、コマンドレジスタ・ステータスレジスタそれぞれのビットを定義しています。

コマンド・ステータスそれぞれのレジスタの各ビットについては、本書で意識すべき範

囲のビットのみ、サンプルコードの実行結果を説明する際に簡単に紹介します。

次に、追加した定数を使用してコマンド・ステータスレジスタをダンプする関数 (dump_command_status()) を追加します (リスト 1.6)。

▼**リスト** 1.6　012_dump_command_status/pci.c

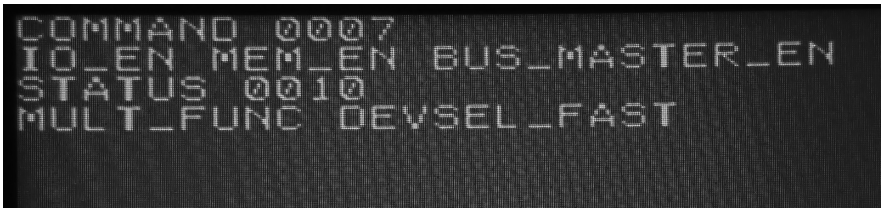
```
/* ・・・ 省略 ・・・ */

void dump_vid_did(unsigned char bus, unsigned char dev, unsigned char func)
{
        /* ・・・ 省略 ・・・ */
}

/* 追加（ここから） */
void dump_command_status(
        unsigned char bus, unsigned char dev, unsigned char func)
{
        /* PCI コンフィグレーション空間のレジスタを読み出す */
        unsigned int conf_data = get_pci_conf_reg(
                bus, dev, func, PCI_CONF_STATUS_COMMAND);

        /* 読み出したデータからステータスとコマンド値を取得 */
        unsigned short command = conf_data & 0x0000ffff;
        unsigned short status = conf_data >> 16;

        /* 表示 */
        puts("COMMAND ");
        puth(command, 4);
        puts("\r\n");

        if (command & PCI_COM_IO_EN)
                puts("IO_EN ");
        if (command & PCI_COM_MEM_EN)
                puts("MEM_EN ");
        if (command & PCI_COM_BUS_MASTER_EN)
                puts("BUS_MASTER_EN ");
        if (command & PCI_COM_SPECIAL_CYCLE)
                puts("SPECIAL_CYCLE ");
        if (command & PCI_COM_MEMW_INV_EN)
                puts("MEMW_INV_EN ");
        if (command & PCI_COM_VGA_PAL_SNP)
                puts("VGA_PAL_SNP ");
        if (command & PCI_COM_PARITY_ERR_RES)
                puts("PARITY_ERR_RES ");
        if (command & PCI_COM_SERR_EN)
                puts("SERR_EN ");
        if (command & PCI_COM_FAST_BACK2BACK_EN)
                puts("FAST_BACK2BACK_EN ");
        if (command & PCI_COM_INTR_DIS)
                puts("INTR_DIS");
        puts("\r\n");

        puts("STATUS ");
        puth(status, 4);
        puts("\r\n");
```

```
        if (status & PCI_STAT_INTR)
                puts("INTR ");
        if (status & PCI_STAT_MULT_FUNC)
                puts("MULT_FUNC ");
        if (status & PCI_STAT_66MHZ)
                puts("66MHZ ");
        if (status & PCI_STAT_FAST_BACK2BACK)
                puts("FAST_BACK2BACK ");
        if (status & PCI_STAT_DATA_PARITY_ERR)
                puts("DATA_PARITY_ERR ");

        switch (status & PCI_STAT_DEVSEL_MASK) {
        case PCI_STAT_DEVSEL_FAST:
                puts("DEVSEL_FAST ");
                break;
        case PCI_STAT_DEVSEL_MID:
                puts("DEVSEL_MID ");
                break;
        case PCI_STAT_DEVSEL_LOW:
                puts("DEVSEL_LOW ");
                break;
        }

        if (status & PCI_STAT_SND_TARGET_ABORT)
                puts("SND_TARGET_ABORT ");
        if (status & PCI_STAT_RCV_TARGET_ABORT)
                puts("RCV_TARGET_ABORT ");
        if (status & PCI_STAT_RCV_MASTER_ABORT)
                puts("RCV_MASTER_ABORT ");
        if (status & PCI_STAT_SYS_ERR)
                puts("SYS_ERR ");
        if (status & PCI_STAT_PARITY_ERR)
                puts("PARITY_ERR");
        puts("\r\n");
}
/* 追加（ここまで） */
```

PCI コンフィグレーション空間上で読み出すレジスタがステータス・コマンド (オフセット:0x04) に変わっただけで、レジスタへのアクセスの仕方は dump_vid_did() と同じです。

表示の処理が少し長いですが、単に一つずつビットをチェックして、セットされているビットがあれば、そのビット名を表示するようにしているだけです。

dump_command_status() を main.c から呼び出すように変更し、実行すると、筆者の PC では図 1.6 のように表示されました。



▲図 1.6　012_dump_command_status の実行結果

図 1.6 の結果から、ここでは初期値としてコマンドレジスタには「IO_EN」と「MEM_EN」そして「BUS_MASTER_EN」のビットが立っていることが確認できました。重要なのは「MEM_EN」と「BUS_MASTER_EN」で、MEM_EN がセットされていることから NIC のレジスタへはメモリアクセスできる事が分かり、BUS_MASTER_EN もセットされているので NIC デバイスはバスマスターとして動作できる状態であることも分かります。コマンドレジスタは書き込み可能なレジスタなので、もしいずれかのフラグがセットされていなかった場合、次節で紹介する PCI コンフィグレーション空間のレジスタへの書き込み方法を参考にこれらのフラグをセットしてください。

なお、「IO_EN」と「MEM」はデバイス側が対応していない場合は 1 をセットしても読みだした時に 0 になるので、その場合は 1 がセットできるアクセス方法で NIC レジスタへアクセスします。本書では NIC のレジスタへメモリアクセスできることを想定してサンプルプログラムを作っていますが、IO アクセスの場合の例も適宜紹介します。

また、図 1.6 の結果から、ステータスの初期値としては「MULT_FUNC」と「DEVSEL_FAST」がセットされていました。これは何かというと、MULT_FUNC は複数の機能 (function) を持つデバイスである事を示していて、PCI コンフィグレーション空間へアクセスする際にファンクション番号だけを変更してアクセスすると別の機能にアクセスできます。また、DEVSEL_FAST は、バス上で特定のデバイスを選択するときの「DEVSEL」という信号に対する応答タイミングが「FAST」であることを示しています。ここらへんはデバイスの特性の問題で、今回 NIC ドライバを作る上でどうこうする部分ではないので「そういうものか」と思っておけば良いです。

1.5.3　PCI コンフィグレーション空間の割り込み無効ビットをセットしてみる

PCI コンフィグレーション空間への書き込みを試すことも兼ねて、コマンドレジスタの割り込み無効ビットを設定し、PCI レベルで割り込みを無効化しておきます。

この節のサンプルディレクトリは「013_set_command」です。

まず、PCI コンフィグレーション空間のレジスタへの設定を行う「set_pci_conf_reg()」を pci.c へ追加します (リスト 1.7)。

▼**リスト** 1.7　013_set_command/pci.c

```
/* ・・・ 省略 ・・・ */
unsigned int get_pci_conf_reg(
        unsigned char bus, unsigned char dev, unsigned char func,
        unsigned char reg)
{
        /* ・・・ 省略 ・・・ */
}

/* 追加（ここから） */
void set_pci_conf_reg(unsigned char bus, unsigned char dev, unsigned char func,
                      unsigned char reg, unsigned int val)
{
        /* CONFIG_ADDRESS を設定 */
        union pci_config_address conf_addr;
        conf_addr.raw = 0;
        conf_addr.bus_num = bus;
        conf_addr.dev_num = dev;
        conf_addr.func_num = func;
        conf_addr.reg_addr = reg;
        conf_addr.enable_bit = 1;
        io_write32(CONFIG_ADDRESS, conf_addr.raw);

        /* CONFIG_DATA へ val を書き込む */
        io_write32(CONFIG_DATA, val);
}
/* 追加（ここまで） */

void dump_vid_did(unsigned char bus, unsigned char dev, unsigned char func)
/* ・・・ 省略 ・・・ */
```

get_pci_conf_reg() とほとんど同じです。(引数や戻り値が変わったり、io_write32() を呼び出すように変えただけです。)

set_pci_conf_reg() を使用してコマンドレジスタへ割り込み無効ビットを設定してみます (リスト 1.8)。

▼**リスト** 1.8　013_set_command/main.c

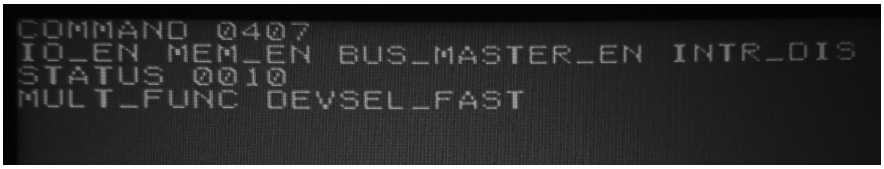
```
/* ・・・ 省略 ・・・ */
void start_kernel(void *_t __attribute__((unused)), struct platform_info *pi,
                  void *_fs_start)
{
        /* フレームバッファ周りの初期化 */
        fb_init(&pi->fb);
        set_fg(255, 255, 255);
        set_bg(0, 70, 250);
        clear_screen();
```

```
        /* ACPI の初期化 */
        acpi_init(pi->rsdp);

        /* CPU 周りの初期化 */
        gdt_init();
        intr_init();

        /* 周辺 IC の初期化 */
        pic_init();
        hpet_init();
        kbc_init();


        /* 変更（ここから） */
        /* 一旦、コマンドとステータスを読み出す */
        unsigned int conf_data = get_pci_conf_reg(
                NIC_BUS_NUM, NIC_DEV_NUM, NIC_FUNC_NUM,
                PCI_CONF_STATUS_COMMAND);

        /* ステータス（上位 16 ビット）をクリア */
        conf_data &= 0x0000ffff;
        /* コマンドに割り込み無効設定 */
        conf_data |= PCI_COM_INTR_DIS;

        /* コマンドとステータスに書き戻す */
        set_pci_conf_reg(NIC_BUS_NUM, NIC_DEV_NUM, NIC_FUNC_NUM,
                         PCI_CONF_STATUS_COMMAND, conf_data);

        /* PCI コンフィグレーション空間からコマンドとステータスをダンプ */
        dump_command_status(NIC_BUS_NUM, NIC_DEV_NUM, NIC_FUNC_NUM);

        /* halt して待つ */
        while (1)
                cpu_halt();
        /* 変更（ここまで） */



        /* システムコールの初期化 */
        syscall_init();

        /* ・・・ 省略 ・・・ */
```

ステータスレジスタは、1 が設定されているビットに 1 を書き込むと、そのビットをクリアします (0 にします)。逆に 0 を書く場合は元のビットを変化させません。今回、ステータスレジスタを変化させる意図は無いため、読み出したコマンド・ステータスレジスタの内容を格納した conf_data の上位 16 ビット (ステータスレジスタの領域) をゼロクリアしています。

そして、PCI_COM_INTR_DIS 定数を使用して conf_data のコマンドレジスタの領域 (下位 16 ビット) に割り込み無効ビットをセットして、set_pci_conf_reg() で書き戻します。

実行すると図 1.7 のように表示されました。



▲図 1.7　013_set_command の実行結果

「COMMAND」に新たに「INTR_DIS」と表示されているので、割り込み無効ビットが設定されていることが確認できました。

コマンドレジスタやステータスレジスタのその他のフラグについて詳しくは、本書末尾の「参考情報」で紹介している書籍か、あるいはネット上を検索してみてください。(十分に枯れた技術なので)

1.6　NIC の BAR をダンプしてみる

PCI の章の最後に、NIC を制御するための情報、すなわち NIC 自体のレジスタ群の先頭アドレス (ベースアドレス) を取得してみます。

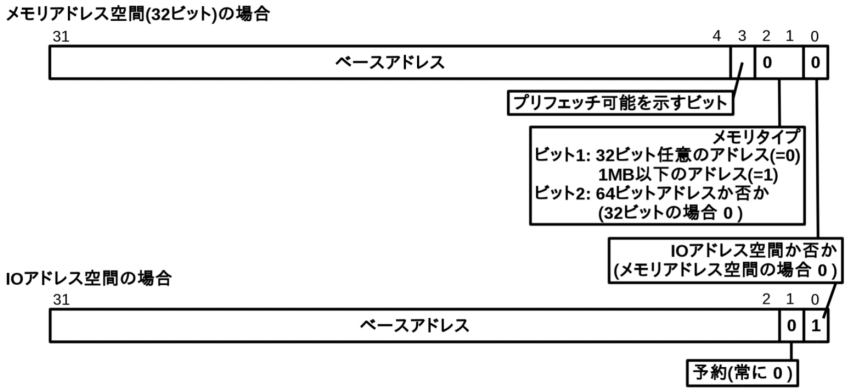
デバイス固有のレジスタのベースアドレスは、PCI コンフィグレーション空間の「Base Address Register(BAR)」に格納されています。この項では、BAR を PCI コンフィグレーション空間から取得し、その中から NIC のレジスタのベースアドレスを取得します。

1.6.1　PCI コンフィグレーション空間内の位置と BAR の構成を確認

PCI コンフィグレーション空間内の BAR の位置を確認するため、PCI コンフィグレーション空間のレジスタマップを再掲します (図 1.8)。

▲図 1.8　PCI コンフィグレーション空間 (再掲)

BAR のレジスタは、PCI コンフィグレーション空間の 0x10 以降です。
そして、BAR の構成は図 1.9 の通りです。



▲図 1.9　BAR の構成

BAR には、デバイスのレジスタのベースアドレスの他に、そのアドレスが IO アドレスなのかメモリアドレスなのか、32 ビットなのか 64 ビットなのか、という情報が格納されています。それらの情報を踏まえて、BAR から取得したアドレスを扱います。

1.6.2 BAR ダンプ処理を作る

アドレスがわかったので、BAR をダンプしてみます。

この節のサンプルディレクトリは「014_dump_bar」です。

まず、pci.h に BAR のアドレスと、BAR 内のビットを定義します。併せて BAR をダンプする関数「dump_bar」のプロトタイプ宣言も追加しておきます。(リスト 1.9)

▼**リスト** 1.9　014_dump_bar/pci.h

```
/* ・・・ 省略 ・・・ */
#define PCI_STAT_PARITY_ERR    (1U << 15)

/* 追加（ここから） */
#define PCI_CONF_BAR   0x10

#define PCI_BAR_MASK_IO         0x00000001
#define PCI_BAR_MASK_MEM_TYPE   0x00000006
#define PCI_BAR_MEM_TYPE_32BIT 0x00000000
#define PCI_BAR_MEM_TYPE_1M     0x00000002
#define PCI_BAR_MEM_TYPE_64BIT 0x00000004
#define PCI_BAR_MASK_MEM_PREFETCHABLE  0x00000008
#define PCI_BAR_MASK_MEM_ADDR  0xfffffff0
#define PCI_BAR_MASK_IO_ADDR   0xfffffffc
/* 追加（ここまで） */

unsigned int get_pci_conf_reg(
        unsigned char bus, unsigned char dev, unsigned char func,
        unsigned char reg);
void set_pci_conf_reg(unsigned char bus, unsigned char dev, unsigned char func,
                      unsigned char reg, unsigned int val);
void dump_vid_did(unsigned char bus, unsigned char dev, unsigned char func);
void dump_command_status(
        unsigned char bus, unsigned char dev, unsigned char func);
/* 追加（ここから） */
void dump_bar(unsigned char bus, unsigned char dev, unsigned char func);
/* 追加（ここまで） */
```

前節で確認した BAR の構成を定義しました。

次に、pci.c に dump_bar() を実装します (リスト 1.10)。

▼**リスト** 1.10　014_dump_bar/pci.c

```
/* ・・・ 省略 ・・・ */

void dump_command_status(
        unsigned char bus, unsigned char dev, unsigned char func)
{
        /* ・・・ 省略 ・・・ */
}
```

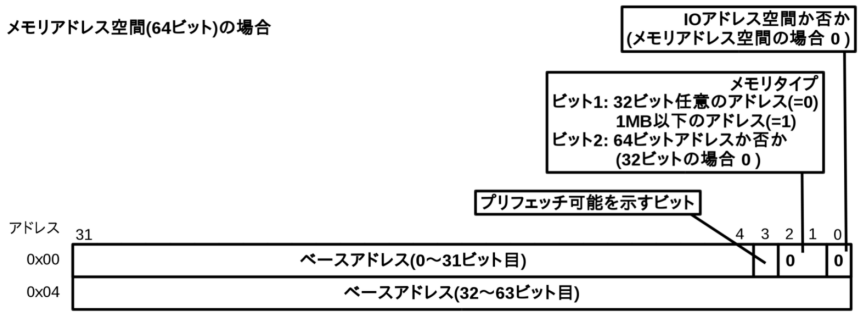
```
/* 追加（ここから） */
void dump_bar(unsigned char bus, unsigned char dev, unsigned char func)
{
        /* PCI コンフィグレーション空間から BAR を取得 */
        unsigned int bar = get_pci_conf_reg(bus, dev, func, PCI_CONF_BAR);
        puts("BAR ");
        puth(bar, 8);
        puts("\r\n");

        /* BAR のタイプを確認し NIC のレジスタのベースアドレスを取得 */
        if (bar & PCI_BAR_MASK_IO) {
                /* IO 空間用ベースアドレス */
                puts("IO BASE ");
                puth(bar & PCI_BAR_MASK_IO_ADDR, 8);
                puts("\r\n");
        } else {
                /* メモリ空間用ベースアドレス */
                unsigned int bar_32;
                unsigned long long bar_upper;
                unsigned long long bar_64;
                switch (bar & PCI_BAR_MASK_MEM_TYPE) {
                case PCI_BAR_MEM_TYPE_32BIT:
                        puts("MEM BASE 32BIT ");
                        bar_32 = bar & PCI_BAR_MASK_MEM_ADDR;
                        puth(bar_32, 8);
                        puts("\r\n");
                        break;

                case PCI_BAR_MEM_TYPE_1M:
                        puts("MEM BASE 1M ");
                        bar_32 = bar & PCI_BAR_MASK_MEM_ADDR;
                        puth(bar_32, 8);
                        puts("\r\n");
                        break;

                case PCI_BAR_MEM_TYPE_64BIT:
                        bar_upper = get_pci_conf_reg(
                                bus, dev, func, PCI_CONF_BAR + 4);
                        bar_64 = (bar_upper << 32)
                                + (bar & PCI_BAR_MASK_MEM_ADDR);
                        puts("MEM BASE 64BIT ");
                        puth(bar_64, 16);
                        puts("\r\n");
                        break;
                }
                if (bar & PCI_BAR_MASK_MEM_PREFETCHABLE)
                        puts("PREFETCHABLE\r\n");
                else
                        puts("NON PREFETCHABLE\r\n");
        }
}
/* 追加（ここまで） */
```

get_pci_conf_reg() で BAR を取得し、BAR の中からデバイスのレジスタのベースアドレスを取り出しています。

アドレスが IO 空間のものなのかメモリ空間のものなのかを判別し、メモリ空間の場合

はさらに 32 ビットなのか 64 ビットなのかを判定しています。

64 ビットの場合、PCI コンフィグレーション空間の 0x10 からの BAR の領域を図 1.10 のように使います。そのため、その場合には 0x14 からの 4 バイトも取得するようにしています。



▲図 1.10　BAR(64 ビット)

外から呼べるように include/pci.h に dump_bar() のプロトタイプ宣言を追加しておいてください (コードは割愛)。

そして、main.c からリスト 1.11 のように呼び出します。

▼リスト 1.11　014_dump_bar/main.c

```
/* ・・・ 省略 ・・・ */

void start_kernel(void *_t __attribute__((unused)), struct platform_info *pi,
                  void *_fs_start)
{
        /* ・・・ 省略 ・・・ */
        nic_init();


        /* 変更（ここから） */
        /* BAR をダンプ */
        dump_bar(NIC_BUS_NUM, NIC_DEV_NUM, NIC_FUNC_NUM);
        /* 変更（ここまで） */

        /* halt して待つ */
        while (1)
                cpu_halt();



        /* システムコールの初期化 */
        syscall_init();
```

```
        /* ・・・ 省略 ・・・ */
```

筆者の PC で実行してみると図 1.11 の結果となりました。

▲図 1.11　014_dump_bar の実行結果

NIC のレジスタのベースアドレスは、メモリ空間の 32 ビットのアドレスで、その値は 0xf1300000 であることが分かります。

実はこのアドレス情報は lspci で確認した際も表示されていました (図 1.12)。

```
➜seven ~  lspci -vx -s 00:19.0
00:19.0 Ethernet controller: Intel Corporation Ethernet Connection (3) I218-V (rev 03)
        Subsystem: Lenovo Device 5020
        Flags: fast devsel, IRQ 64
        Memory at f1300000 (32-bit, non-prefetchable) [disabled] [size=128K]
        Memory at f133e000 (32-bit, non-prefetchable) [disabled] [size=4K]
        I/O ports at 4080 [disabled] [size=32]
        Capabilities: <access denied>
        Kernel driver in use: e1000e
00: 86 80 a3 15 00 04 10 00 03 00 00 02 00 00 00 00
10: 00 00 30 f1 00 e0 33 f1 81 40 00 00 00 00 00 00
20: 00 00 00 00 00 00 00 00 00 00 00 00 aa 17 20 50
30: 00 00 00 00 c8 00 00 00 00 00 00 00 0b 01 00 00
```

▲図 1.12　PCI コンフィグレーション空間の情報を表示 (再掲)

lspci 出力 4 行目の「Memory at f1300000」がそうです[*2]。

これで、取得した NIC レジスタのベースアドレスも正しそうだと分かりました。

1.6.3　BAR からベースアドレスを取得する処理を関数化しておく

本章の最後に、BAR から NIC のレジスタのベースアドレスを取得する処理を「get_nic_reg_base」という名前で関数化しておきます。

[*2] 以降の行に「Memory at f133e000」や「I/O ports at 4080」とあるので、複数のメモリアドレスや IO アドレスとして NIC のレジスタへアクセスすることも可能なのかも知れません。(ただ、筆者未確認ですが。試してみると良いかも知れません。)

この節のサンプルディレクトリは「015_get_nic_reg_base」です。
前節の確認を踏まえ、nic.c へリスト 1.12 のように関数化してみました。

▼**リスト** 1.12　015_get_nic_reg_base/nic.c

```
/* ・・・ 省略 ・・・ */

void nic_init(void)
{
        /* ・・・ 省略 ・・・ */
}

/* 追加（ここから） */
unsigned int get_nic_reg_base(void)
{
        /* PCI コンフィグレーション空間から BAR を取得 */
        unsigned int bar = get_pci_conf_reg(
                NIC_BUS_NUM, NIC_DEV_NUM, NIC_FUNC_NUM, PCI_CONF_BAR);

        /* メモリ空間用ベースアドレス（32 ビット）を返す */
        return bar & PCI_BAR_MASK_MEM_ADDR;
}
/* 追加（ここまで） */
```

前節で NIC のレジスタのベースアドレスは、32 ビットのメモリアドレスであることを確認したので、それに従ってアドレスを抽出しています。もし、使用している環境で前節の dump_bar() が異なる結果を表示していた場合は、表示された結果に従って get_nic_reg_base() を実装してください。その際は、dump_bar() の実装が参考になると思います。

最後に、この関数が外から呼べるように include/nic.h へプロトタイプ宣言を追加しておいてください (コードは割愛)。

第 2 章

NIC を制御してイーサネットフレームを送受信する

前章で NIC のレジスタのベースアドレスを取得できました。

この章では、NIC のレジスタを使用して、任意のイーサネットフレームを受け取ったり、特にフォーマットに従っていない「オレオレデータ」を送ってみたり、ということを手っ取り早く行ってみます。

なお、この章では適宜データシートの図を引用します。参照しているデータシートについては本書最後の「参考情報」をご覧ください。

2.1 NIC のレジスタの操作方法

まず、NIC のレジスタの操作方法を紹介します。

ここでは、NIC の割り込み設定の確認と無効化を通して、レジスタ値の取得と設定の方法を紹介します。

なお、データシートで「パケット (packet)」と呼んでいるレジスタ等は本書でも「パケット」と呼称します。これは「IP パケット」ではなく「データの塊」という意味の広義の「packet」で、現在の TCP/IP 通信において NIC が送受信するデータの単位としては「イーサネットフレーム」です。NIC が送受信するデータの塊を「パケット」と呼称すると「IP パケット」と混同してまぎらわしいので、NIC が扱うデータの塊については「イーサネットフレーム (あるいは単にフレーム)」と呼称します。

2.1.1 NIC のレジスタ値を取得してみる

BAR から取得したベースアドレス以降に NIC のレジスタが並んでいて、read アクセスすることでレジスタの値を取得できます。

NIC にはイベント毎にいくつか割り込みがあり、それらの有効/無効は「IMS(Interrupt Mask Set/Read:オフセット 0x00d0)」というレジスタの各ビットで管理しています。「割り込みマスク (Interrupt Mask)」という名の通り、1 が設定されているビットに対応する割り込みが有効になっています。

この節では、IMS のレジスタ値を取得し、何らかの割り込みが設定されているのかどうかを見てみます。

この節のサンプルディレクトリは「021_dump_nic_reg」です。

まず、include/nic.h へ IMS レジスタのアドレス (ベースアドレスからのオフセット) を定義します (リスト 2.1)。

▼**リスト** 2.1　021_dump_nic_reg/include/nic.h

```
/* ・・・ 省略 ・・・ */
#define NIC_FUNC_NUM   0x0

#define NIC_REG_IMS    0x00d0  /* 追加 */

void nic_init(void);
unsigned int get_nic_reg_base(void);
unsigned int get_nic_reg(unsigned short reg);   /* 追加 */
void dump_nic_ims(void);        /* 追加 */
```

併せて、指定した NIC レジスタの値を取得する関数「get_nic_reg」と、それを使用して IMS をダンプする関数「dump_nic_ims」のプロトタイプ宣言を追加しておきました。

それでは、それぞれの関数を nic.c に実装します (リスト 2.2)。

▼**リスト** 2.2　021_dump_nic_reg/nic.c

```
/* ・・・ 省略 ・・・ */

unsigned int get_nic_reg_base(void)
{
        /* ・・・ 省略 ・・・ */
}

/* 追加（ここから） */
unsigned int get_nic_reg(unsigned short reg)
{
        unsigned long long addr = nic_reg_base + reg;
```

```
        return *(volatile unsigned int *)addr;
}

void dump_nic_ims(void)
{
        unsigned int ims = get_nic_reg(NIC_REG_IMS);

        puts("IMS ");
        puth(ims, 8);
        puts("\r\n");
}
/* 追加（ここまで） */
```

前章にて、NIC のレジスタのアドレスはメモリアドレスであることを確認していたので、get_nic_reg() ではメモリアクセス (ポインタによるアクセス) でレジスタ値を取得しています。

なお、NIC レジスタのアドレスが IO アドレスの場合は in 命令でレジスタ値を取得します。その場合は、get_nic_reg() は、in 命令を使用するように実装してください。in 命令で任意のアドレス先の値を取得する関数として io_read32() を既に実装済なので、それを使用して例えばリスト 2.3 の様に実装できます。

▼**リスト** 2.3　IO **アドレス時の** get_nic_reg() **の例**

```
unsigned int get_nic_reg(unsigned short reg)
{
        return io_read32(reg);
}
```

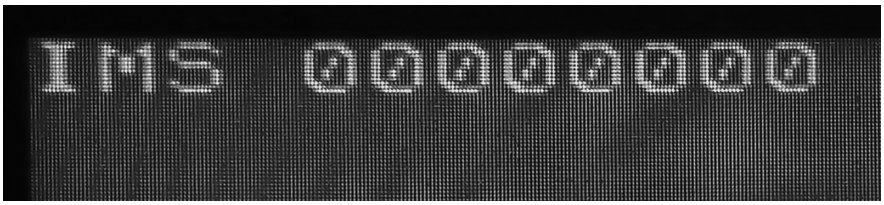
main.c から dump_nic_ims() を呼び出すようにして作業完了です (リスト 2.4)。

▼**リスト** 2.4　021_dump_nic_reg/main.c

```
/* ・・・ 省略 ・・・ */

void start_kernel(void *_t __attribute__((unused)), struct platform_info *pi,
                  void *_fs_start)
{
        /* ・・・ 省略 ・・・ */

        /* 周辺 IC の初期化 */
        pic_init();
        hpet_init();
        kbc_init();
        nic_init();



        /* 変更（ここから） */
```

```
        /* NIC の IMS(Interrupt Mask Set/Read Register) レジスタをダンプ */
        dump_nic_ims();
        /* 変更（ここまで） */

        /* halt して待つ */
        while (1)
                cpu_halt();



        /* ・・・ 省略 ・・・ */
```

実行結果は図 2.1 の通りです。



▲図 2.1　021_dump_nic_reg の実行結果

全てのビットがゼロなので、割り込みは設定されていませんでした。

2.1.2　NIC のレジスタ値を設定してみる

前節で NIC のレジスタ値の取得方法を紹介しました。この節ではレジスタ値の設定方法を紹介します。

前節で割り込みが設定されていない事 (割り込みマスクが全て 0) を確認しましたが、ここではレジスタ値の設定方法の紹介として、割り込みマスクを適当に設定し、それをクリアしてみます。

この節のサンプルディレクトリは「022_set_nic_reg」です。

割り込みマスクを設定するレジスタは 2 つあります。

- IMS(Interrupt Mask Set/Read:オフセット 0x00d0)
  - 割り込みを有効にしたいビットを 1 にしてレジスタに書き込む
- IMC(Interrupt Mask Clear:オフセット 0x00d8)
  - 割り込みを無効にしたいビットを 1 にしてレジスタに書き込む
  - すると、IMS の該当ビットが 0 になる

ここでは、IMS で適当に「0x0000beef」を割り込みマスクに設定し、IMC でそれをク

リアしてみます。

まずは、新たに使用する IMC の定義と、指定された NIC レジスタへ値を設定する関数「set_nic_reg」のプロトタイプ宣言を include/nic.h へ追加します (リスト 2.5)。

▼**リスト** 2.5　022_set_nic_reg/include/nic.h

```
/* ・・・ 省略 ・・・ */
#define NIC_FUNC_NUM   0x0

#define NIC_REG_IMS    0x00d0
#define NIC_REG_IMC    0x00d8  /* 追加 */

void nic_init(void);
unsigned int get_nic_reg_base(void);
unsigned int get_nic_reg(unsigned short reg);
void set_nic_reg(unsigned short reg, unsigned int val); /* 追加 */
void dump_nic_ims(void);
```

そして、set_nic_reg() の実装を nic.c へ追加します (リスト 2.6)。

▼**リスト** 2.6　022_set_nic_reg/nic.c

```
/* ・・・ 省略 ・・・ */

unsigned int get_nic_reg(unsigned short reg)
{
        /* ・・・ 省略 ・・・ */
}

/* 追加（ここから） */
void set_nic_reg(unsigned short reg, unsigned int val)
{
        unsigned long long addr = nic_reg_base + reg;
        *(volatile unsigned int *)addr = val;
}
/* 追加（ここまで） */

void dump_nic_ims(void)
/* ・・・ 省略 ・・・ */
```

レジスタ値を取得する場合とは逆に、レジスタのアドレスが指す先へ値を書き込むようにしているだけです。

なお、IO アドレスの場合の set_nic_reg() の実装例はリスト 2.7 の通りです。

▼**リスト** 2.7　IO **アドレス時の** set_nic_reg() **の例**

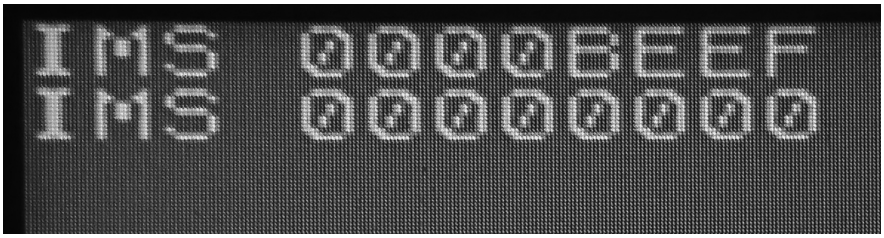
```
void set_nic_reg(unsigned short reg, unsigned int val)
{
        io_write32(reg, val);
}
```

get_nic_reg() の場合と同様に、今度は io_write32() を使用して実装できます。
最後に、main.c から割り込みマスク設定とクリアを試してみます (リスト 2.8)。

▼**リスト** 2.8　022_set_nic_reg/main.c

```
/* ・・・ 省略 ・・・ */

void start_kernel(void *_t __attribute__((unused)), struct platform_info *pi,
                  void *_fs_start)
{
        /* ・・・ 省略 ・・・ */

        /* 周辺 IC の初期化 */
        pic_init();
        hpet_init();
        kbc_init();
        nic_init();



        /* 変更（ここから） */
        /* クリアの確認用に適当な値をセット */
        set_nic_reg(NIC_REG_IMS, 0x0000beef);
        dump_nic_ims();

        /* NIC の割り込みを IMC(Interrupt Mask Clear Register) で全て無効化 */
        set_nic_reg(NIC_REG_IMC, 0xffffffff);

        /* NIC の IMS(Interrupt Mask Set/Read Register) レジスタをダンプ */
        dump_nic_ims();
        /* 変更（ここまで） */

        /* halt して待つ */
        while (1)
                cpu_halt();



        /* システムコールの初期化 */
        /* ・・・ 省略 ・・・ */
```

set_nic_reg() で IMS へ 0x0000beef という値を割り込みマスクへ設定し dump_nic_ims() で確認した後、IMC へ全てのビットが 1 の値を設定することで割り込みマスクをクリアしています。
実行すると図 2.2 のように表示されます。



▲図 2.2　022_set_nic_reg の実行結果

2.1.3　NIC 関連の割り込み無効化処理を関数化する

この項の最後に、PCI 側での NIC デバイスの割り込み無効化処理と、NIC 内の割り込みマスクによる割り込み無効化処理を、一つの関数にまとめておきます。

この節のサンプルディレクトリは「022_set_nic_reg_refactor」です。

nic.c に「disable_nic_interrupt」という関数を追加してみました (リスト 2.9)。

▼リスト 2.9　022_set_nic_reg_refactor/nic.c

```
/* ・・・ 省略 ・・・ */

static unsigned int nic_reg_base;

/* 追加（ここから） */
static void disable_nic_interrupt(void)
{
        /* 一旦、コマンドとステータスを読み出す */
        unsigned int conf_data = get_pci_conf_reg(
                NIC_BUS_NUM, NIC_DEV_NUM, NIC_FUNC_NUM,
                PCI_CONF_STATUS_COMMAND);

        /* ステータス（上位 16 ビット）をクリア */
        conf_data &= 0x0000ffff;
        /* コマンドに割り込み無効設定 */
        conf_data |= PCI_COM_INTR_DIS;

        /* コマンドとステータスに書き戻す */
        set_pci_conf_reg(NIC_BUS_NUM, NIC_DEV_NUM, NIC_FUNC_NUM,
                         PCI_CONF_STATUS_COMMAND, conf_data);

        /* NIC の割り込みを IMC(Interrupt Mask Clear Register) で全て無効化 */
        set_nic_reg(NIC_REG_IMC, 0xffffffff);
}
/* 追加（ここまで） */

void nic_init(void)
{
```

```
        /* 変更（ここから） */
        /* NIC のレジスタのベースアドレスを取得しておく */
        nic_reg_base = get_nic_reg_base();
        /* 変更（ここまで） */

        /* NIC の割り込みを全て無効にする */
        disable_nic_interrupt();
}

unsigned int get_nic_reg_base(void)
/* ・・・ 省略 ・・・ */
```

割り込み無効化処理を関数化したことで、nic_init() の見通しが良くなりました。

2.2 フレームを受信する

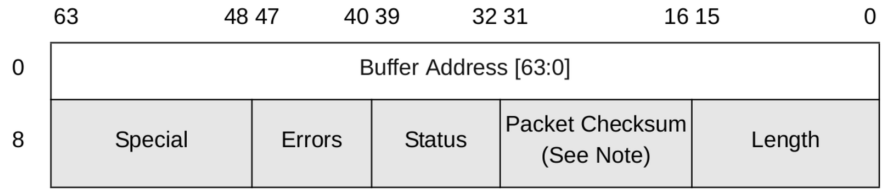
前項で、NIC レジスタ値の取得方法と設定方法が分かりました。この項では、NIC レジスタへイーサネットフレーム受信のための設定を行い、受信処理の動作確認を行います。

この項のサンプルディレクトリは「023_receive_frame」です。この項では、項全体を通してこのサンプルを作ります。

2.2.1 NIC のフレーム受信の流れ

大枠を説明すると、NIC は受信した各イーサネットフレームを「受信ディスクリプタ (Receive Descriptor)」というデータ構造でメモリ上のリングバッファへ順次格納していきます。ドライバはそのリングバッファから受信ディスクリプタの構造を解釈しながら読み出していく、という流れです。

もう少し詳しく説明します。まず「受信ディスクリプタ」のデータ構造をデータシートから引用します (図 2.3[*1])。



▲図 2.3　受信ディスクリプタのデータ構造

[*1] 3.2.3 Receive Descriptor Format - PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer's Manual

受信ディスクリプタは 64 ビット (8 バイト) からなるデータ構造です。各フィールドの意味は以下の通りです。

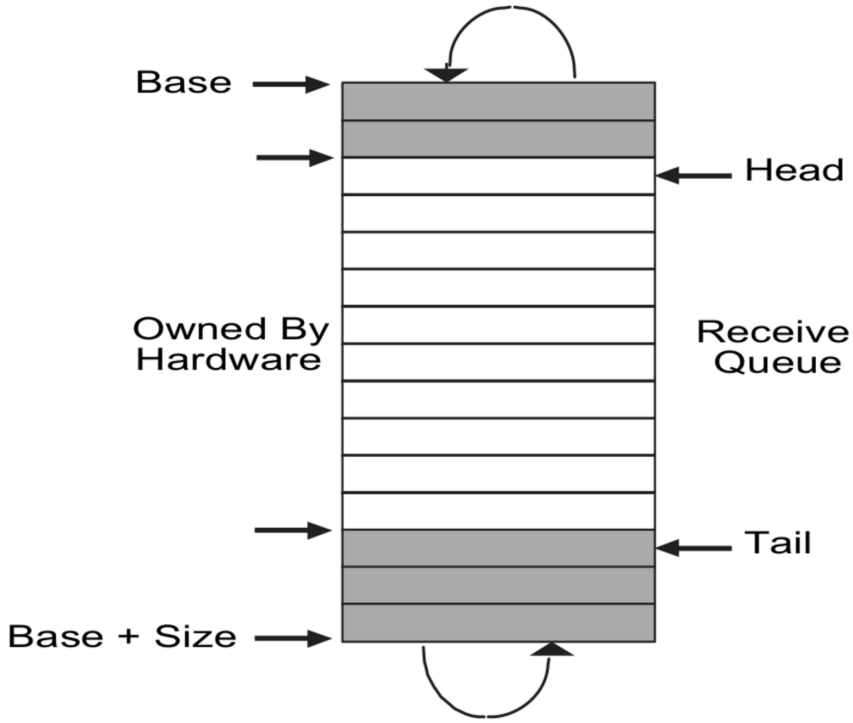
- Buffer Address
  - 受信したイーサネットフレームを格納するバッファ (受信バッファ) のアドレス
- Length
  - 受信バッファの長さ (バイト数)
- Packet Chesksum
  - NIC で自動計算された受信フレームのチェックサム
  - 受信バッファのどこからどこまでのチェックサムであるかは、パケットの種類やその他の NIC レジスタの設定による
- Status
  - ディスクリプタのステータスを示す
  - NIC が自動的に設定する
  - 本書で使用するフラグについては実装する際に説明する
- Errors
  - 受信したフレームに関するエラー情報が格納される
  - 本書では使用しない
- Special
  - VLAN 等の特殊用途用のフィールド
  - こちらも本書では使用しない

「Buffer Address」の説明の通り、受信したイーサネットフレーム自体は受信ディスクリプタではなく、Buffer Address に指定されているアドレスの場所に格納されます。そのため、受信ディスクリプタ用リングバッファだけでなく、イーサネットフレームを格納するバッファ用の領域も事前に確保しておく必要があります。

そして、フレームを受信する度に NIC はこの受信ディスクリプタのデータ構造をリングバッファへ格納していきます。このリングバッファは、予め NIC のレジスタへ「受信ディスクリプタ用リングバッファのベースアドレス」と「リングバッファの長さ (バイト)」を指定することで、NIC は指定された領域をリングバッファとして使用するようになります。

リングバッファの構造と使われ方については図 2.4[*2]の通りです。

[*2] 3.2.6 Receive Descriptor Queue Structure - PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer's Manual



▲図 2.4　受信リングバッファについて

図 2.4 の「Head」や「Tail」も NIC のレジスタです。これらの値も初期化時に適切に初期化しておきます。(詳細は実装しながら説明します。)

NIC は受信したフレームを Head が指す場所から Tail が指す場所に向かってリングバッファへ格納していくので、ドライバは Head から順に読み出していきます。

(Tail - 1) が指す領域まで格納し終えると、仕様上 NIC はそこより先の領域はアクセスしません。受信したフレームが喪失してしまうので、ドライバでは、リングバッファからフレームを取得したら、取得したインデックスまで Tail を進めるようにします。

以上が、NIC が受信したフレームを処理する流れです。以降の節では、さっそくフレーム受信処理を実装し始めます。使用するレジスタ等、詳しくは実装を進めながら適宜紹介します。

2.2.2 各種の定義の用意とバッファの確保

ここでは、受信ディスクリプタのデータ構造等の定義を行い、受信ディスクリプタ用リングバッファと受信したイーサネットフレーム自体を格納するバッファのメモリ領域を確保します (リスト 2.10)。

▼**リスト** 2.10　023_receive_frame/nic.c

```
/* ・・・ 省略 ・・・ */

/* 追加（ここから） */
#define RXDESC_NUM     80
#define ALIGN_MARGIN   16

struct __attribute__((packed)) rxdesc {
        unsigned long long buffer_address;
        unsigned short length;
        unsigned short packet_checksum;
        unsigned char status;
        unsigned char errors;
        unsigned short special;
};
/* 追加（ここまで） */

static unsigned int nic_reg_base;

/* 追加（ここから） */
static unsigned char rx_buffer[RXDESC_NUM][PACKET_BUFFER_SIZE];
static unsigned char rxdesc_data[
        (sizeof(struct rxdesc) * RXDESC_NUM) + ALIGN_MARGIN];
static struct rxdesc *rxdesc_base;
static unsigned short current_rx_idx;
/* 追加（ここまで） */

static void disable_nic_interrupt(void)
/* ・・・ 省略 ・・・ */
```

受信ディスクリプタとイーサネットフレームの領域は、ここでは簡単に、グローバル変数で確保してみました。「rx_buffer」がイーサネットフレーム用バッファの領域で、「rxdesc_data」が受信ディスクリプタ用リングバッファの領域です。

なお、受信ディスクリプタのアドレスは 16 バイトアライメント (16 の倍数) である必要があります。そのため、rxdesc_data を確保する際、16 バイト多めに確保しておきます。そして、rxdesc_data の先頭から 16 バイトアライメントとなる最初のアドレスを受信ディスクリプタのベースアドレスとして使うようにします。そのベースアドレスを管理するために「rxdesc_base」変数を用意しています。イーサネットフレーム用のバッファにはアライメント制約はありませんので、配列として確保した rx_buffer 変数をそのまま使用します。

受信ディスクリプタ用リングバッファの要素数 (RXDESC_NUM) は、ここでは 80 にしてみました。この値はお好きなように決めて構いませんが、後述するリングバッファの長さを指定するレジスタの制約で、要素数は 8 の倍数である必要があります。また、本シリーズの現状の自作 OS ではカーネル本体 (kernel.bin) で使用するメモリサイズが 1MB 未満である必要がありますのでご注意ください[*3]。

イーサネットフレーム 1 つ当たりのバッファサイズ (PACKET_BUFFER_SIZE) は、include/nic.h に定義していて、今回は 1024 バイトとしました (リスト 2.11)。

▼**リスト** 2.11　023_receive_frame/include/nic.h

```
/* ・・・省略 ・・・ */
#define NIC_RDESC_STAT_PIF     (1U << 7)

#define PACKET_BUFFER_SIZE     1024
#define PACKET_RBSIZE_BIT      NIC_RCTL_BSIZE_1024B

void nic_init(void);
/* ・・・ 省略 ・・・ */
```

PACKET_RBSIZE_BIT は、NIC の振る舞いを設定するレジスタにフレームバッファ 1 つのサイズを設定する際の定数です。PACKET_BUFFER_SIZE を変更した際は併せて変更する必要があるため、近くに定義しています。

include/nic.h には他にも、使用する NIC のレジスタとレジスタが持つビットの定義を全て書いています。単に定数を定義しているだけの内容なのでコードを引用しての紹介はしません。内容が気になる場合はサンプルコードを見てみてください。

最後に、「current_rx_idx」はリングバッファ内の次に読む場所を覚えておくための変数です。詳しくは変数を使用する際に説明します。

2.2.3　受信の初期化処理を実装する

前節で受信処理に使用する領域確保や定数の定義等を行いました。この節ではそれらを初期化します。

まずは受信ディスクリプタ用リングバッファのベースアドレス (rxdesc_base) を初期化します (リスト 2.12)。

▼**リスト** 2.12　023_receive_frame/nic.c

[*3] ブートローダー (poiboot) にて、カーネルスタックのベースアドレスをカーネルをロードしたアドレスの 1MB 先にしているのと、カーネルのリンカスクリプト (kernel.ld) にてカーネルのメモリサイズを 960KB としているためです。詳しくは本シリーズ最初の「フルスクラッチで作る!x86_64 自作 OS(パート 1)」を参照してください。

```
/* ・・・ 省略 ・・・ */
static void disable_nic_interrupt(void)
{
        /* ・・・ 省略 ・・・ */
}

/* 追加（ここから） */
static void rx_init(void)
{
        /* rxdesc の先頭アドレスを 16 バイトの倍数となるようにする */
        unsigned long long rxdesc_addr = (unsigned long long)rxdesc_data;
        rxdesc_addr = (rxdesc_addr + ALIGN_MARGIN) & 0xfffffffffffffff0;

        /* rxdesc の初期化 */
        rxdesc_base = (struct rxdesc *)rxdesc_addr;
}
/* 追加（ここまで） */

/* ・・・ 省略 ・・・ */
```

「rxdesc_addr」という計算用の変数を使用して、rxdesc_data(リングバッファ用に確保した領域の先頭アドレス) にアライメントサイズである 16 を足してから、16 バイトアライメントとなるように下位 4 ビットを 0 でマスクしています。アライメントサイズを足す前に下位 4 ビットを 0 でマスクしてしまうと、確保した領域より若いアドレスとなってしまう可能性があり、前節で領域を確保する際に 16 バイト多めに確保した意味が無くなってしまうので、アライメントサイズを足してからマスクします。

受信ディスクリプタ用リングバッファのベースアドレスを用意できたので、次はリングバッファ内の各受信ディスクリプタを初期化します (リスト 2.13)。

▼**リスト** 2.13　023_receive_frame/nic.c

```
/* ・・・ 省略 ・・・ */

static void rx_init(void)
{
        unsigned int i; /* 追加 */

        /* rxdesc の先頭アドレスを 16 バイトの倍数となるようにする */
        unsigned long long rxdesc_addr = (unsigned long long)rxdesc_data;
        rxdesc_addr = (rxdesc_addr + ALIGN_MARGIN) & 0xfffffffffffffff0;

        /* rxdesc の初期化 */
        rxdesc_base = (struct rxdesc *)rxdesc_addr;
        /* 追加（ここから） */
        struct rxdesc *cur_rxdesc = rxdesc_base;
        for (i = 0; i < RXDESC_NUM; i++) {
                cur_rxdesc->buffer_address = (unsigned long long)rx_buffer[i];
                cur_rxdesc->status = 0;
                cur_rxdesc->errors = 0;
                cur_rxdesc++;
```

```
        }
        /* 追加（ここまで） */
}

/* ・・・ 省略 ・・・ */
```

buffer_address にイーサネットフレーム用バッファのアドレスを設定し、ステータスとエラーフラグを 0 でクリアしています。

次に、NIC のリングバッファのベースアドレスとサイズを NIC のレジスタへ設定します (リスト 2.14)。

▼**リスト** 2.14　023_receive_frame/nic.c

```
/* ・・・ 省略 ・・・ */

static void rx_init(void)
{
        unsigned int i;

        /* rxdesc の先頭アドレスを 16 バイトの倍数となるようにする */
        unsigned long long rxdesc_addr = (unsigned long long)rxdesc_data;
        rxdesc_addr = (rxdesc_addr + ALIGN_MARGIN) & 0xfffffffffffffff0;

        /* rxdesc の初期化 */
        rxdesc_base = (struct rxdesc *)rxdesc_addr;
        struct rxdesc *cur_rxdesc = rxdesc_base;
        for (i = 0; i < RXDESC_NUM; i++) {
                cur_rxdesc->buffer_address = (unsigned long long)rx_buffer[i];
                cur_rxdesc->status = 0;
                cur_rxdesc->errors = 0;
                cur_rxdesc++;
        }

        /* 追加（ここから） */
        /* rxdesc の先頭アドレスとサイズを NIC レジスタへ設定 */
        set_nic_reg(NIC_REG_RDBAH, rxdesc_addr >> 32);
        set_nic_reg(NIC_REG_RDBAL, rxdesc_addr & 0x00000000ffffffff);
        set_nic_reg(NIC_REG_RDLEN, sizeof(struct rxdesc) * RXDESC_NUM);
        /* 追加（ここまで） */
}

/* ・・・ 省略 ・・・ */
```

「NIC_REG_」と付いている定数が NIC のレジスタのアドレス (NIC レジスタベースアドレスからのオフセット) で、include/nic.h に定義しているものです。ここでは 3 つのレジスタ値を設定しています。

まず、NIC はリングバッファのベースアドレスを 64 ビットアドレスで管理します。設定するレジスタとしては 2 つあり、RDBAH(Receive Descriptor Base High) に上位 32 ビットを設定し、RDBAL(Receive Descriptor Base Low) へ下位 32 ビットを設定し

ます。

RDLEN(Receive Descriptor Length) にはリングバッファの長さ (バイト数) を設定します。このバイト数は 128 の倍数で指定することが決められており、前述の通り、要素数としては 8 の倍数となります。

そして、リングバッファの Head と Tail を管理する NIC レジスタも初期化します (リスト 2.15)。

▼**リスト** 2.15　023_receive_frame/nic.c

```
/* ・・・ 省略 ・・・ */

static void rx_init(void)
{
        unsigned int i;

        /* rxdesc の先頭アドレスを 16 バイトの倍数となるようにする */
        unsigned long long rxdesc_addr = (unsigned long long)rxdesc_data;
        rxdesc_addr = (rxdesc_addr + ALIGN_MARGIN) & 0xfffffffffffffff0;

        /* rxdesc の初期化 */
        rxdesc_base = (struct rxdesc *)rxdesc_addr;
        struct rxdesc *cur_rxdesc = rxdesc_base;
        for (i = 0; i < RXDESC_NUM; i++) {
                cur_rxdesc->buffer_address = (unsigned long long)rx_buffer[i];
                cur_rxdesc->status = 0;
                cur_rxdesc->errors = 0;
                cur_rxdesc++;
        }

        /* rxdesc の先頭アドレスとサイズを NIC レジスタへ設定 */
        set_nic_reg(NIC_REG_RDBAH, rxdesc_addr >> 32);
        set_nic_reg(NIC_REG_RDBAL, rxdesc_addr & 0x00000000ffffffff);
        set_nic_reg(NIC_REG_RDLEN, sizeof(struct rxdesc) * RXDESC_NUM);

        /* 追加（ここから）*/
        /* 先頭と末尾のインデックスを NIC レジスタへ設定 */
        current_rx_idx = 0;
        set_nic_reg(NIC_REG_RDH, current_rx_idx);
        set_nic_reg(NIC_REG_RDT, RXDESC_NUM - 1);
        /* 追加（ここまで）*/
}

/* ・・・ 省略 ・・・ */
```

RDH(Receive Register Head) に Head を、RDT(Receive Register Tail) に Tail のインデックスを設定します。RDH に 0 を、RDT に (要素数 - 1) を設定しておくことで、この後受信処理を有効化すると、RDH から順次リングバッファに受信ディスクリプタを格納していくようになります。

なお、RDH は NIC が次の書き込み先のインデックスを保持するためのもので、この後受信処理の有効化を行いますが、それ以降はドライバ側から値の変更を行ってはなりませ

ん (データシートに明記されています)。ドライバ側でリングバッファのどこまでを読み出したのかを保持しておくためには「current_rx_idx」を用意しています (併せて初期化しています)。

対して、RDT は「リングバッファのここよりは書き込まないでくれ」ということをドライバが NIC へ伝えるためのものなので、ドライバでリングバッファから値を読み出す都度、更新する必要があります。(でないといずれ RDH が RDT へ追いつき、NIC はリングバッファへの書き込みをやめてしまいます。)RDT の更新処理については、受信処理を実装する際に紹介します。

最後に NIC の受信動作設定と受信処理の有効化を行います (リスト 2.16)。

▼**リスト** 2.16　023_receive_frame/nic.c

```
/* ・・・ 省略 ・・・ */

static void rx_init(void)
{
        unsigned int i;

        /* rxdesc の先頭アドレスを 16 バイトの倍数となるようにする */
        unsigned long long rxdesc_addr = (unsigned long long)rxdesc_data;
        rxdesc_addr = (rxdesc_addr + ALIGN_MARGIN) & 0xfffffffffffffff0;

        /* rxdesc の初期化 */
        rxdesc_base = (struct rxdesc *)rxdesc_addr;
        struct rxdesc *cur_rxdesc = rxdesc_base;
        for (i = 0; i < RXDESC_NUM; i++) {
                cur_rxdesc->buffer_address = (unsigned long long)rx_buffer[i];
                cur_rxdesc->status = 0;
                cur_rxdesc->errors = 0;
                cur_rxdesc++;
        }

        /* rxdesc の先頭アドレスとサイズを NIC レジスタへ設定 */
        set_nic_reg(NIC_REG_RDBAH, rxdesc_addr >> 32);
        set_nic_reg(NIC_REG_RDBAL, rxdesc_addr & 0x00000000ffffffff);
        set_nic_reg(NIC_REG_RDLEN, sizeof(struct rxdesc) * RXDESC_NUM);

        /* 先頭と末尾のインデックスを NIC レジスタへ設定 */
        current_rx_idx = 0;
        set_nic_reg(NIC_REG_RDH, current_rx_idx);
        set_nic_reg(NIC_REG_RDT, RXDESC_NUM - 1);

        /* 追加（ここから） */
        /* NIC の受信動作設定 */
        set_nic_reg(NIC_REG_RCTL, PACKET_RBSIZE_BIT | NIC_RCTL_BAM
                    | NIC_RCTL_MPE | NIC_RCTL_UPE | NIC_RCTL_SBP | NIC_RCTL_EN);
        /* 追加（ここまで） */
}

/* ・・・ 省略 ・・・ */
```

RCTL(Receive Control Register) が NIC の受信時の振る舞いを設定するレジスタで

す。定数を使って設定している他にも設定できるビットはあるのですが、ここでは設定しているもののみ紹介します。(全てのビットフィールドが気になる方はデータシートを見てみてください。) なお、「NIC_RCTL_」が付いている定数は RCTL レジスタに設定できる各ビットを表す定数です。

- PACKET_RBSIZE_BIT
  - イーサネットフレーム 1 つ当たりのバッファサイズを指定するビット
  - 先述の通り nic.h にて「BSIZE_1024B」で再定義しており、1024 バイト
- BAM(Broadcast Accept Mode)
  - ブロードキャストパケットを受け入れるかどうかの設定
  - この項のサンプルでは受信する全てをダンプするようにしてみたいので 1(受け入れる) に設定
- MPE(Multicast Promiscuous Enabled)
  - マルチキャストパケットに対するプロミスキャス・モード[*4]設定
  - フィルタリングしてほしくないので 1 に設定します。
- UPE(Unicast Promiscuous Enabled)
  - ユニキャストパケットについてのプロミスキャス・モード設定
  - こちらも 1 を設定
- SBP(Store Bad Packets)
  - BAD パケット[*5]をバッファへ格納するかどうかの設定
  - 無差別に格納して欲しいのでこちらも 1 を設定
- EN(Receiver Enable)
  - 受信処理を有効化するビット
  - このビットを 1 にすると NIC の受信処理が始まる

これで NIC の受信に関する初期化処理を行う rx_init() は完成です。NIC 全体の初期化処理を行う関数 nic_init() から呼び出すようにしておきます (コード紹介は割愛)。

2.2.4 受信処理を実装する

前節で NIC は受信処理を開始し、用意したバッファへ受信したフレームを順次格納していくようになりました。

それでは、リングバッファとフレーム用のバッファから、フレームを取得する関数「receive_frame」を実装してみます (リスト 2.17)。

[*4] 無差別 (promiscuous) に受信する設定
[*5] CRC エラー、シンボルエラー、シーケンスエラー、長さのエラー、等のパケット

▼**リスト** 2.17　023_receive_frame/nic.c

```
/* ・・・ 省略 ・・・ */

void dump_nic_ims(void)
{
        /* ・・・ 省略 ・・・ */
}

/* 追加（ここから） */
unsigned short receive_frame(void *buf)
{
        unsigned short len = 0;

        struct rxdesc *cur_rxdesc = rxdesc_base + current_rx_idx;
        if (cur_rxdesc->status & NIC_RDESC_STAT_DD) {
                len = cur_rxdesc->length;
                memcpy(buf, (void *)cur_rxdesc->buffer_address,
                       cur_rxdesc->length);

                cur_rxdesc->status = 0;

                set_nic_reg(NIC_REG_RDT, current_rx_idx);

                current_rx_idx = (current_rx_idx + 1) % RXDESC_NUM;
        }

        return len;
}
/* 追加（ここまで） */
```

receive_frame() は、リングバッファに格納されている未取得のイーサネットフレームを一つ引数で指定されたポインタ (buf) へコピーし、コピーしたフレームの長さを戻り値として返します。

流れとしては、まず、cur_rxdesc に、現在参照している (current_rx_idx が指す) 受信ディスクリプタのポインタを設定します[*6]。

次に、受信ディスクリプタ cur_rxdesc のステータスに DD(Descriptor Done) が設定されているかどうかを if 文でチェックしています。これは NIC が受信ディスクリプタの処理を終えたことを示すビットで、このビットが設定されていればその受信ディスクリプタをドライバから参照して良いことになります。受信ディスクリプタのステータスには他にもフラグがありますが、本書では DD しか使いません。なお、受信ディスクリプタのステータスに DD がセットされていない場合、まだフレームを受信していないということで、len の初期値 0 を返して終了します。

受信ディスクリプタに DD がセットされていた場合、フレームの長さを受信ディスクリ

[*6] ポインタに整数 N を足した場合にポインタが参照する型の N 個分先のアドレスとなる事を利用しています。

プタの length メンバーから取得し、戻り値用の変数 len へ設定、memcpy() を使用して引数 buf へイーサネットフレーム用バッファの内容をコピーします。

その後は、ステータスをクリアし、RDT を今取得を完了したインデックス (current_rx_idx) で更新し、current_rx_idx を進めます。

それでは、receive_frame() を使用して受信したフレームをダンプする関数「dump_frame()」を実装してみます (リスト 2.18)。

▼**リスト** 2.18　023_receive_frame/nic.c

```
/* ・・・ 省略 ・・・ */

unsigned short receive_frame(void *buf)
{
        /* ・・・ 省略 ・・・ */
}

/* 追加（ここから） */
unsigned short dump_frame(void)
{
        unsigned char buf[PACKET_BUFFER_SIZE];
        unsigned short len;
        len = receive_frame(buf);

        unsigned short i;
        for (i = 0; i < len; i++) {
                puth(buf[i], 2);
                putc(' ');

                if (((i + 1) % 24) == 0)
                        puts("\r\n");
                else if (((i + 1) % 4) == 0)
                        putc(' ');
        }
        if (len > 0)
                puts("\r\n");

        return len;
}
/* 追加（ここまで） */
```

表示する際の文字間隔は 1 行当たりの文字数を調整するために色々やっていますが、receive_frame() で buf へ取得したフレームをダンプしているだけです。

最後に dump_frame() を main.c から呼び出すようにしてみます (リスト 2.19)。

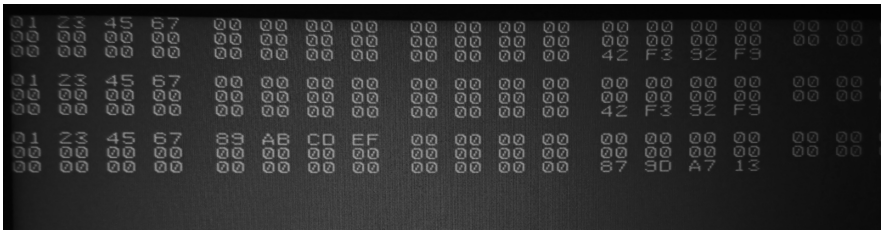
▼**リスト** 2.19　023_receive_frame/main.c

```
/* ・・・ 省略 ・・・ */

void start_kernel(void *_t __attribute__((unused)), struct platform_info *pi,
                  void *_fs_start)
```

```
{
        /* ・・・ 省略 ・・・ */

        /* 周辺 IC の初期化 */
        pic_init();
        hpet_init();
        kbc_init();
        nic_init();

        /* 変更（ここから） */
        /* 受信したフレームをダンプし続ける */
        while (1) {
                if (dump_frame() > 0)
                        puts("\r\n");
        }
        /* 変更（ここまで） */

        /* ・・・ 省略 ・・・ */
```

2.2.5 動作確認

ビルドし、実行すると、図 2.5 の様に受信動作を確認できます。



▲図 2.5　023_receive_frame の実行結果

動作確認の際は、他の PC と 1 対 1 で接続し、pkttools[*7]等の様に任意のパケットを送信できるツールで試してみると分かりやすいです。なお、現在のたいていの PC は Auto-MDIX という機能に対応しているのでストレートケーブルで 1 対 1 に接続しても問題ないです。

図 2.5 はリスト 2.20 のフレームを 2 つとリスト 2.21 のフレームを 1 つ受信したところです。これは pkttools で任意のイーサネットフレームを送信する際に指定できるテキストフォーマットで、「SIZE」の行の次の行からが送信するイーサネットフレームのバイナリデータです (16 進数表記)。

[*7] http://kozos.jp/software/pkttools.html

▼リスト 2.20　対向 PC から送信したフレーム その 1

```
-- 2 --
TIME: 1548948030.069473 Fri Feb  1 00:20:30 2019
SIZE: 4/4
000000: 01 23 45 67
==
```

▼リスト 2.21　対向 PC から送信したフレーム その 2

```
-- 2 --
TIME: 1548948030.069473 Fri Feb  1 00:20:30 2019
SIZE: 8/8
000000: 01 23 45 67 89 ab cd ef
==
```

なお、図 2.5 ではリスト 2.20 やリスト 2.21 で指定しているデータの他に、0 のパディングの後、末尾に 4 バイトの何かのデータがくっついています。これは、FCS(Frame Check Sequence) と呼ばれるチェックサムです。これは、イーサネットフレームを送信する際の送信ディスクリプタの設定に応じて NIC が計算して付けているものです。Linux の場合 (というか一般的な NIC ドライバの場合)、標準で付けるようにしていますが、次の項で紹介する送信処理ではこれを付けないようにしてみるので、その際は付きません[*8]。

2.3　オレオレフレームを送信する

前項でイーサネットフレームの受信を試しました。この項ではイーサネットフレームの送信を試してみます。

ここではあくまでも NIC の動作確認として特にイーサネットフレームのフォーマットに従わないオレオレなフレームを送信してみます。

この項を通して「024_send_frame」のサンプルディレクトリの内容を作ります。

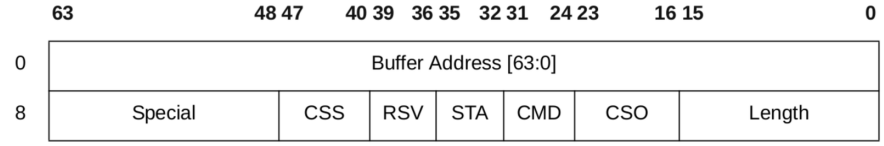
2.3.1　NIC のフレーム送信の流れ

フレーム送信も受信と同じくディスクリプタ用のリングバッファとフレーム自体を置いておくためのバッファを使用して行います。

送信のディスクリプタ (Transmit Descriptor) のデータ構造は図 2.6[*9]の通りです。

[*8] 動作確認を分かりやすくするためにも最初は極力色々な機能を動かさないで試す方針です。

[*9] 3.3.3 Legacy Transmit Descriptor Format - PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer's Manual



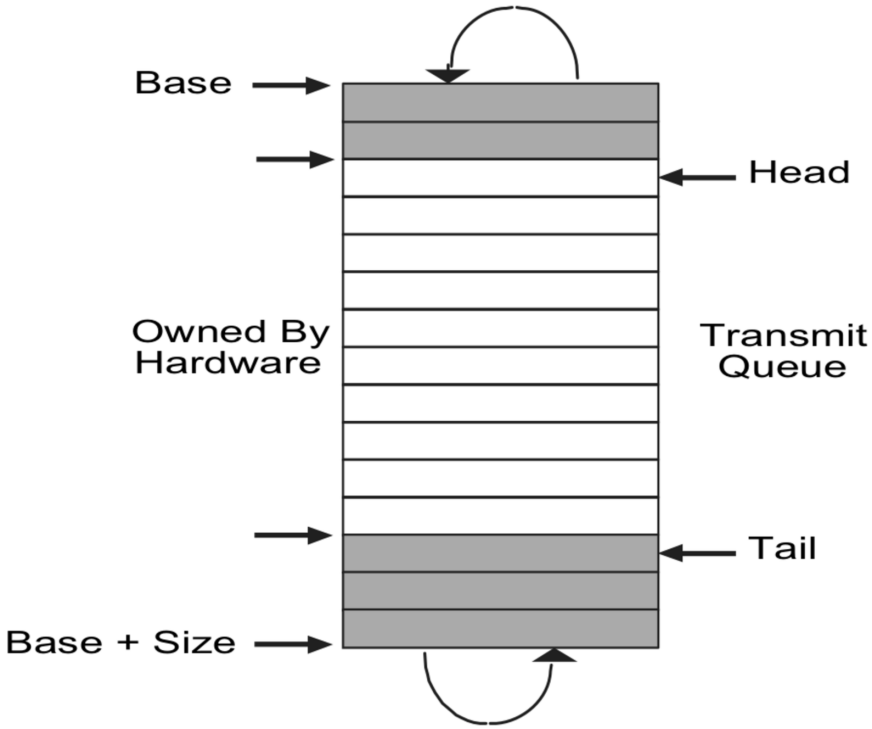
▲図 2.6　送信ディスクリプタのデータ構造

また、各フィールドの意味については以下の通りです。

- Buffer Address
  - 送信するイーサネットフレーム本体を格納したバッファのアドレス
- Length
  - 送信するイーサネットフレームの長さ (バイト数)
  - 0 を指定すると NIC はそのディスクリプタのフレームを送信しない
- CSO(Checksum Offset)
  - NIC がチェックサムを自動挿入するモードが有効になっている時、チェックサムを挿入するフレーム先頭からの相対位置
- CMD(Command)
  - このディスクリプタの振る舞いを指定するいくつかのコマンドをこのビットフィールドで指定できる
  - 使用するコマンドについては使用する際に説明する
- STA(Status)
  - このディスクリプタの状態を表すフィールド
  - ドライバ側であらかじめゼロクリアしておき、NIC が適宜設定する
  - 使用するステータスについては使用する際に説明する
- RSV(Reserved)
  - 予約領域
  - 将来の互換性のため書き込む際は 0 を書くこと
- CSS(Checksum Start)
  - チェックサムの計算を開始する位置 (フレーム先頭からのオフセット) を指定する
  - Length フィールド未満であること
- Special
  - このイーサネットフレームに対して特殊な設定 (VLAN 設定等) を行うフィールド
  - 本書では使用しないため説明は割愛

実は送信ディスクリプタには 3 種類あり、図 2.6 で紹介したものは「レガシー送信ディスクリプタ (Legacy Transmit Descriptor)」と呼ばれるものです。ただ、これは「古い」というよりは「オリジナル」としての意味合いであるとのことで、他の 2 つはまとめて「拡張」ディスクリプタと呼ばれる[*10]、とのことです (データシートより)。最初試すときはなるたけ簡単なやり方の方がトラブらなくて良いので、本書ではレガシー送信ディスクリプタで試します。

そして、送信ディスクリプタ用リングバッファの構造と使われ方は図 2.7[*11]の通りです。



▲図 2.7　送信リングバッファについて

特に受信の時と違いは無いです。

[*10] 拡張ディスクリプタには、「TCP/IP データディスクリプタ」と呼ばれるレガシーディスクリプタを拡張したディスクリプタと、「TCP/IP コンテキストディスクリプタ」と呼ばれる NIC の制御情報のみ (パケットデータ無し) のディスクリプタ、があります。

[*11] 3.4 Transmit Descriptor Ring Structure - PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer's Manual

それでは、次の節から実装を始めます。

2.3.2 各種の定義の用意とバッファの確保

受信の時と同様に、今度は送信ディスクリプタのデータ構造の定義とグローバル変数による領域の確保を行います (リスト 2.22)。

▼**リスト** 2.22　024_send_frame/nic.c

```
/* ・・・ 省略 ・・・ */

#define RXDESC_NUM     80
#define TXDESC_NUM     8       /* 追加 */
#define ALIGN_MARGIN   16

struct __attribute__((packed)) rxdesc {
        unsigned long long buffer_address;
        unsigned short length;
        unsigned short packet_checksum;
        unsigned char status;
        unsigned char errors;
        unsigned short special;
};

/* 追加（ここから） */
struct __attribute__((packed)) txdesc {
        unsigned long long buffer_address;
        unsigned short length;
        unsigned char cso;
        unsigned char cmd;
        unsigned char sta:4;
        unsigned char _rsv:4;
        unsigned char css;
        unsigned short special;
};
/* 追加（ここまで） */

static unsigned int nic_reg_base;

static unsigned char rx_buffer[RXDESC_NUM][PACKET_BUFFER_SIZE];
static unsigned char rxdesc_data[
        (sizeof(struct rxdesc) * RXDESC_NUM) + ALIGN_MARGIN];
static struct rxdesc *rxdesc_base;
static unsigned short current_rx_idx;

/* 追加（ここから） */
static unsigned char txdesc_data[
        (sizeof(struct txdesc) * TXDESC_NUM) + ALIGN_MARGIN];
static struct txdesc *txdesc_base;
static unsigned short current_tx_idx;
/* 追加（ここまで） */

static void disable_nic_interrupt(void)
/* ・・・ 省略 ・・・ */
```

送信ディスクリプタ用リングバッファの要素数は最小[*12]の 8 にしました。後述しますが、送信処理の関数では渡されたフレームの送信完了を確認してから return するようにするため、リングバッファの要素数としては 1 つあれば十分です。また、イーサネットフレーム自体のバッファは確保していません。後述しますが送信用関数に渡されたデータ領域をそのまま使います。

その他は、受信の時と同様です。送信用に新たに使用するレジスタのアドレス定義等も include/nic.h へ追加していますがコード紹介しての説明は省略します。レジスタ自体については使用する際に適宜紹介します。

2.3.3 送信の初期化処理を実装する

次に初期化処理「tx_init」関数を実装します。

ほとんど受信の時と同じなので一気に紹介します (リスト 2.23)。

▼**リスト** 2.23　024_send_frame/nic.c

```
/* ・・・ 省略 ・・・ */

static void rx_init(void)
{
        /* ・・・ 省略 ・・・ */
}

/* 追加（ここから） */
static void tx_init(void)
{
        unsigned int i;

        /* txdesc の先頭アドレスを 16 バイトの倍数となるようにする */
        unsigned long long txdesc_addr = (unsigned long long)txdesc_data;
        txdesc_addr = (txdesc_addr + ALIGN_MARGIN) & 0xfffffffffffffff0;

        /* txdesc の初期化 */
        txdesc_base = (struct txdesc *)txdesc_addr;
        struct txdesc *cur_txdesc = txdesc_base;
        for (i = 0; i < TXDESC_NUM; i++) {
                cur_txdesc->buffer_address = 0;
                cur_txdesc->length = 0;
                cur_txdesc->cso = 0;
                cur_txdesc->cmd = NIC_TDESC_CMD_RS | NIC_TDESC_CMD_EOP;
                cur_txdesc->sta = 0;
                cur_txdesc->_rsv = 0;
                cur_txdesc->css = 0;
```

[*12] 送信ディスクリプタの長さも受信ディスクリプタと同様に 128 バイトアライメントの制約があり、要素数としては 8 の倍数である必要があります。

```
                cur_txdesc->special = 0;
                cur_txdesc++;
        }

        /* txdesc の先頭アドレスとサイズを NIC レジスタへ設定 */
        set_nic_reg(NIC_REG_TDBAH, txdesc_addr >> 32);
        set_nic_reg(NIC_REG_TDBAL, txdesc_addr & 0x00000000ffffffff);
        set_nic_reg(NIC_REG_TDLEN, sizeof(struct txdesc) * TXDESC_NUM);

        /* 先頭と末尾のインデックスを NIC レジスタへ設定 */
        current_tx_idx = 0;
        set_nic_reg(NIC_REG_TDH, current_tx_idx);
        set_nic_reg(NIC_REG_TDT, current_tx_idx);

        /* NIC の送信動作設定 */
        set_nic_reg(NIC_REG_TCTL, (0x40 << NIC_TCTL_COLD_SHIFT)
                    | (0x0f << NIC_TCTL_CT_SHIFT) | NIC_TCTL_PSP | NIC_TCTL_EN);
}
/* 追加（ここまで） */

/* ・・・ 省略 ・・・ */
```

受信の時と異なる箇所としては、まず、各送信ディスクリプタの各フィールドの初期化処理です。各フィールドの初期化内容について以下にまとめます。

- buffer_address と length
  - 送信時に設定するため 0 指定
  - buffer_address に関してはヌルポインタ (0) を指定しておくと NIC はそのディスクリプタを送信しないという効果もある
- cmd(Command)
  - RS(Report Status)
    - NIC へフレーム送信完了後のステータスフィールドの更新を要求する
  - EOP(End Of Packet)
    - 一連のパケットの最終であることを示す
    - 今回送信するフレームは一つ一つがそれ単体で完結しているものなので初期値として設定しておく
- cso(Checksum Offset) と css(Checksum Start)
  - 今回、チェックサムの自動挿入モードは使用しないため 0
  - チェックサムの自動挿入は cmd に「IC(Insert Checksum)」のビットを設定することで有効化される
- sta(Status)
  - 初期値としてゼロクリア
- Special
  - 使用しないので 0

次に受信の時と異なる箇所としては、先頭 (TDH: Transmit Descriptor Head レジスタ) と末尾 (TDT: Transmit Descriptor Tail レジスタ) のインデックス指定で、受信とは違い、共に 0 で初期化しています。送信の際、NIC はリングバッファの TDH から (TDL - 1) の間を送信するため、送信するべきディスクリプタが無い初期状態では共に 0 番目を指すようにしておきます。

受信と異なる箇所の最後は NIC の動作設定 (TCTL: Transmit Control Register) です。設定しているビットについては以下の通りです。なお、これまで同様に「NIC_TCTL_」と先頭に付く定数は TCTL レジスタ内のビットフィールドを示す定数です。末尾に「_SHIFT」が付くものは 2 ビット以上のビットフィールドに対して、左シフトするビット数を定義しています。定義内容は include/nic.h を参照してください。

- COLD(Collision Distance)
  - コリジョン時の CSMA/CD 操作の為に経過しなければならないバイトタイムを指定する
  - データシートに推奨値が書かれていて、ここではその通りに全二重推奨値である 0x40 を設定
- CT(Collision Threshold)
  - コリジョン時に諦めるまでの再送信回数
  - データシート曰く「半二重動作でのみ意味がある」とのこと
  - ここもデータシートの推奨値 0x0f を指定
- PSP(Pad Short Packets)
  - このビットが指定されていると、NIC に渡されたフレームの長さが 64 バイト未満の時、パディング (0) を加えて 64 バイトにしてくれる
  - このビットが指定されていない場合、NIC が送信できる最小の長さは 32 バイトなので、それ未満の長さのフレームは送信できない
  - 後述するが試しに送信するフレームは 0xbeefbeef(4 バイト) なので、このビットを指定しておく
- EN(Transmit Enable)
  - このビットを指定すると送信処理が有効になる

以上で tx_init() の説明は終了です。最後に tx_init() を nic_init() から呼び出すようにしておきます (コード紹介は割愛)。

2.3.4 送信処理を実装する

送信したいイーサネットフレームのバッファへのポインタと長さを渡すと、そのデータを送信し、ステータスを返す関数「send_frame」を実装します (リスト 2.24)。

▼**リスト** 2.24　024_send_frame/nic.c

```
/* ・・・ 省略 ・・・ */

unsigned short dump_frame(void)
{
        /* ・・・ 省略 ・・・ */
}

/* 追加（ここから） */
unsigned char send_frame(void *buf, unsigned short len)
{
        /* txdesc の設定 */
        struct txdesc *cur_txdesc = txdesc_base + current_tx_idx;
        cur_txdesc->buffer_address = (unsigned long long)buf;
        cur_txdesc->length = len;
        cur_txdesc->sta = 0;

        /* idx 更新 */
        current_tx_idx = (current_tx_idx + 1) % TXDESC_NUM;
        set_nic_reg(NIC_REG_TDT, current_tx_idx);

        /* 送信完了を待つ */
        unsigned char send_status = 0;
        while (!send_status)
                send_status = cur_txdesc->sta & 0x0f;

        return send_status;
}
/* 追加（ここまで） */
```

まず、送信ディスクリプタは、buffer_address と length へ渡されたものをそのまま指定し、sta(Status) をゼロクリアしています。

後は TDT(Transmit Descriptor Tail) をインクリメントすることで、NIC がそのディスクリプタの内容を送信します。

その後、送信ディスクリプタの sta(Status) は下位 4 ビットがステータスフラグとして使用されるビットなので、そこに何らかのビットが設定されるのを待ちます。設定されたらステータス値をそのまま返して終了です。

これを呼び出す main.c としてはリスト 2.25 の様に実装してみました。

▼**リスト** 2.25　024_send_frame/main.c

```
/* ・・・ 省略 ・・・ */

void start_kernel(void *_t __attribute__((unused)), struct platform_info *pi,
                  void *_fs_start)
{
        /* ・・・ 省略 ・・・ */

        /* 周辺 IC の初期化 */
        pic_init();
        hpet_init();
        kbc_init();
        nic_init();



        /* 変更（ここから） */
        /* 1 秒周期で 0xbeefbeef を送信し続ける */
        unsigned int data = 0xbeefbeef;
        unsigned short len = sizeof(data);
        while (1) {
                puts("BE");
                unsigned char status = send_frame(&data, len);
                switch (status) {
                case NIC_TDESC_STA_DD:
                        puts("EF");
                        break;
                case NIC_TDESC_STA_EC:
                        puts("EC");
                        break;
                case NIC_TDESC_STA_LC:
                        puts("LC");
                        break;
                case NIC_TDESC_STA_TU:
                        puts("TU");
                        break;
                }
                putc(' ');

                sleep(1 * SEC_TO_US);
        }
        /* 変更（ここまで） */



        /* ・・・ 省略 ・・・ */
```

「0xbeefbeef」という 4 バイトのオレオレイーサネットフレームを 1 秒周期で送信し続けるようにしてみました。

send_frame() が返す送信ディスクリプタのステータスフラグの意味については以下の通りです。

- DD(Descriptor Done)
  - 送信処理が正常終了したことを示す
- EC(Excess Collisions)

  - TCTL レジスタの CT フィールドで指定されている最大衝突回数を超えたため送信されなかった事を示す
- LC(Late Collision)
  - 半二重モードでの作業中にレイトコリジョンが発生したことを示す
  - 全二重モード時は意味のないビット
- TU(Transmit Underrun)
  - 送信アンダーランイベントが発生したことを示す
  - Early Transmit(早期送信) 機能が有効になっていた場合にバッファ内のデータ不足で早期送信を完了できなかった事を示す
  - 今回は特に意識不要

DD が正常終了でそれ以外が異常終了を示します。

"BE"と出力した後、送信し、正常終了なら"EF"を、異常終了ならそれらのステータスに対応する 2 文字を出力するようにしてみました。

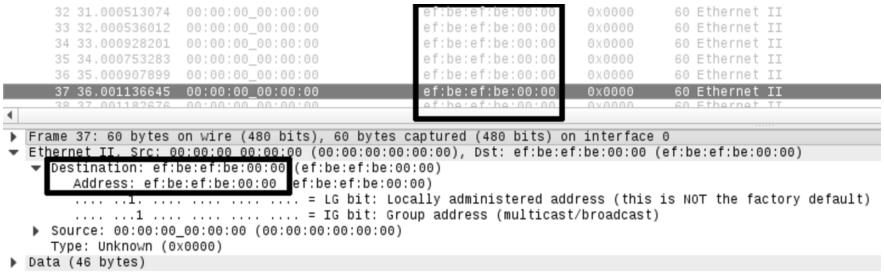
2.3.5 動作確認

実行すると図 2.8 の様に"BEEF BEEF ..."と出力されます。

▲図 2.8　024_send_frame の実行結果

実際に対向 PC と 1 対 1 で接続[*13]し、対向 PC 側で Wireshark 等でパケットキャプチャしている様子が図 2.9 です。

[*13] 送信しているデータはイーサネットフレームの体を成していない (宛先 MAC アドレスや送信元 MAC アドレスが書いていない) ので、経路上にスイッチングハブ等が合った場合、おそらく間違いなく捨てられます。



▲図 2.9　対向側で受信している様子

イーサネットフレームの先頭 6 バイトである宛先 MAC アドレス部に"0xbeefbeef"と書かれた謎のフレームが続々と受信されている事が確認できます。

付録 A

MAC アドレスを取得する

本書では「オレオレイーサネットフレーム」として任意のバイナリ列を NIC から送信する方法を紹介しました。後は「オレオレ」ではなくちゃんとしたイーサネットフレームのバイナリ列を作ればイーサネットフレームでの通信を行うことができます。

ただ、イーサネットフレームを作るためには最低限、自身の NIC の MAC アドレスを知る必要があります。ここでは、NIC 内に保存されている MAC アドレスの取得方法を紹介します。

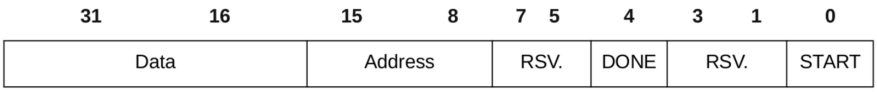
A.1　EEPROM へのアクセスを試す

たいていの場合、MAC アドレスは NIC の EEPROM から取得できます。ここでは、NIC の EEPROM へのアクセスを試します。

この項のサンプルディレクトリは「A02_get_mac_01」です。

A.1.1　EERD の使い方

NIC の EEPROM の値の取得には「EEPROM Read Register(EERD:オフセット 0x0014)」という専用のレジスタがあります (図 1.1[*1])。



▲図 A.1　EERD レジスタについて

[*1] 13.4.4 EEPROM Read Register - PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer's Manual

また、EERD レジスタの各ビットフィールドの意味は以下の通りです。

- START
  - 1 を設定すると、NIC が EEPROM の ADDR フィールドのアドレスの場所からデータを読み出し、DATA フィールドへ格納する
  - このビットは自動的に 0 に戻る
- DONE
  - EEPROM のデータ読み出しを完了すると 1 が設定される
  - NIC が自動的に設定するビットで、ソフトウェアからの書き込みは無効
- ADDR
  - 読み出したい EEPROM のアドレスを指定する
- DATA
  - EEPROM から読み出したデータが格納されるフィールド

以上から、EERD を使用した EEPROM アクセスの流れは以下の通りです。

1. ADDR へアクセスしたい EEPROM のアドレスと、START ビットを設定
2. DONE ビットが設定されるのを待つ
3. DATA フィールドに格納された EEPROM の値を取得

そして、EEPROM のアドレスマップは図 1.2[*2]の通りです (MAC アドレスの箇所のみ抜粋)。

Word	Used By	Bit 15 - 8	Bit 7 - 0
00h	HW	Ethernet Address Byte 2	Ethernet Address Byte 1
01h	HW	Ethernet Address Byte 4	Ethernet Address Byte 3
02h	HW	Ethernet Address Byte 6[a]	Ethernet Address Byte 5

▲図 A.2　EEPROM のアドレスマップ (MAC アドレスの箇所のみ)

EEPROM の先頭 (0x00 バイト目) から 2 バイト (16 ビット) ずつ MAC アドレスが並んでいます。

なお、NIC によっては (PC によっては?)EEPROM は搭載されていないものなのか、EEPROM からの値取得が行えないものもあります。その際、2. の DONE ビットがい

[*2] 5.6 EEPROM Address Map - PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer's Manual

つまで経っても設定されないので、どこかのタイミングでタイムアウトする必要があります。

A.1.2 EEPROM へのアクセスを実装してみる

それでは、EEPROM へのアクセスを試してみます。

与えられた EEPROM アドレスの値を取得する関数「get_eeprom_data」を nic.c へ追加します (リスト 1.1)。

▼**リスト** 1.1　A02_get_mac_01/nic.c

```
/* ・・・ 省略 ・・・ */

static void disable_nic_interrupt(void)
{
        /* ・・・ 省略 ・・・ */
}

/* 追加（ここから） */
#define EERD_TIMEOUT   10000
/* 1 万回分の EERD 読み出し処理を経ても DONE ビットが立たない場合
 * タイムアウトとする */
static int get_eeprom_data(unsigned char eeprom_addr)
{
        /* アクセスしたい EEPROM アドレスと START ビットをセット */
        set_nic_reg(NIC_REG_EERD,
                    (eeprom_addr << NIC_EERD_ADDRESS_SHIFT) | NIC_EERD_START);

        /* DONE ビットが設定されるのをタイムアウト付きで待つ */
        volatile unsigned int wait = EERD_TIMEOUT;
        while (wait--) {
                unsigned int eerd = get_nic_reg(NIC_REG_EERD);
                if (eerd & NIC_EERD_DONE) {
                        /* DONE ビットが設定されたら
                         * EERD に格納されたデータ（上位 16 ビット）を返す */
                        return eerd >> NIC_EERD_DATA_SHIFT;
                }
        }

        /* タイムアウトの際は-1 を返す */
        return -1;
}
/* 追加（ここまで） */

/* ・・・ 省略 ・・・ */
```

やっている内容は前項で説明の通りです。タイムアウトの際は-1 を返すようにしてみました。

「EERD_TIMEOUT」の定数値に特に理由は無いので、適宜変えてみてください。最初に試す際は「本当にいつまで経っても DONE は設定されないのか」を確認する意味で

も莫大な値か、あるいはタイムアウト無しで試してみると良いと思います。

そして、NIC 初期化時に呼ばれて MAC アドレス取得を行う関数「get_mac_addr」も用意しておきます (リスト 1.2)。

▼**リスト** 1.2　A02_get_mac_01/nic.c

```
/* ・・・ 省略 ・・・ */
static int get_eeprom_data(unsigned char eeprom_addr)
{
        /* ・・・ 省略 ・・・ */
}

/* 追加（ここから） */
static void get_mac_addr(void)
{
        unsigned char eeprom_accessible = get_eeprom_data(0x00) >= 0;

        if (eeprom_accessible) {
                puts("EEPROM ACCESSIBLE\r\n");
        } else {
                puts("EEPROM NOT ACCESSIBLE\r\n");
        }

        while (1);
}
/* 追加（ここまで） */

/* ・・・ 省略 ・・・ */
```

まずは EEPROM へのアクセスがタイムアウトするか否かから、EEPROM へのアクセスが可能か否かを出力するようにしてみました。

また、ここでは MAC アドレス取得実験のため、get_mac_addr() の最後で無限ループにより処理が進まないようにしています。

最後に nic_init() から呼び出すようにすれば作業完了です (リスト 1.3)。

▼**リスト** 1.3　A02_get_mac_01/nic.c

```
/* ・・・ 省略 ・・・ */

void nic_init(void)
{
        /* NIC のレジスタのベースアドレスを取得しておく */
        nic_reg_base = get_nic_reg_base();

        /* NIC の割り込みを全て無効にする */
        disable_nic_interrupt();

        /* 追加（ここから） */
        /* MAC アドレスを取得 */
        get_mac_addr();
```

```
        /* 追加（ここまで） */

        /* 受信の初期化処理 */
        rx_init();

        /* 送信の初期化処理 */
        tx_init();
}

unsigned int get_nic_reg_base(void)
/* ・・・ 省略 ・・・ */
```

A.1.3 動作確認

筆者の環境では、実行すると図 1.3 のように EEPROM へのアクセスはできませんでした。

▲図 A.3　A02_get_mac_01 の実行結果

次項で別の方法を試してみます。

A.1.4 補足: EEPROM アクセスができた場合の MAC アドレス取得方法

EEPROM へのアクセスができた場合、get__eeprom__data() を使用してリスト 1.4 の様に MAC アドレスを取得できます。(「nic__mac__addr」は、グローバル変数として定義された unsigned char 型の配列とします。)

▼リスト 1.4　EEPROM から MAC アドレスを取得する関数例

```
static void get_mac_addr_eeprom(void)
{
        unsigned short mac_1_0 = (unsigned short)get_eeprom_data(0x00);
        unsigned short mac_3_2 = (unsigned short)get_eeprom_data(0x01);
        unsigned short mac_5_4 = (unsigned short)get_eeprom_data(0x02);

        nic_mac_addr[0] = mac_1_0 & 0x00ff;
        nic_mac_addr[1] = mac_1_0 >> 8;
        nic_mac_addr[2] = mac_3_2 & 0x00ff;
```

```
        nic_mac_addr[3] = mac_3_2 >> 8;
        nic_mac_addr[4] = mac_5_4 & 0x00ff;
        nic_mac_addr[5] = mac_5_4 >> 8;
}
```

EEPROM の先頭から 2 バイトずつ取り出し、1 バイトずつ nic_mac_addr[] へ格納しています。

A02_get_mac_01 のサンプルディレクトリでは、get_mac_addr() で EEPROM にアクセス可能な場合、get_mac_addr_eeprom() を呼び出して MAC アドレスを取得し、その結果を表示するようにしています。適宜参照してみてください。

A.2 受信アドレスレジスタから取得する

EERD レジスタによる EEPROM からの値取得とは別の方法として、「受信アドレスレジスタ」を使用する方法を紹介します。

この項のサンプルディレクトリは「A02_get_mac_02」です。

「受信アドレスレジスタ」は、受信したフレームを MAC アドレスでフィルタリングするためのレジスタです。MAC アドレスの下位 32 ビット (4 バイト) を登録する「Receive Address Low(RAL)」と、上位 16 ビット (2 バイト) を登録する「Receive Address High」の 2 つのレジスタがあり、それぞれ 16 個ずつあるので、フィルタリングするアドレスを 16 個まで NIC へ登録できる仕組みです。

ここで、受信アドレスレジスタの先頭の要素には自身の NIC の MAC アドレスを設定することが決められています。NIC によるのか UEFI によるのかは不明ですが、この値が自動的に設定されている場合、ここから NIC の MAC アドレスを知ることができます。

受信アドレスレジスタを使用する場合の MAC アドレス取得関数「get_mac_addr_rar」はリスト 1.5 の通りです。

▼**リスト** 1.5　a02_get_mac_02/nic.c

```
/* ・・・ 省略 ・・・ */
unsigned char nic_mac_addr[6] = { 0 };        /* 追加 */

/* ・・・ 省略 ・・・ */

/* 追加（ここから） */
static void get_mac_addr_rar(void)
{
        unsigned int ral_0 = get_nic_reg(NIC_REG_RAL(0));
        unsigned int rah_0 = get_nic_reg(NIC_REG_RAH(0));

        nic_mac_addr[0] = ral_0 & 0x000000ff;
```

```
        nic_mac_addr[1] = (ral_0 >> 8) & 0x000000ff;
        nic_mac_addr[2] = (ral_0 >> 16) & 0x000000ff;
        nic_mac_addr[3] = (ral_0 >> 24) & 0x000000ff;
        nic_mac_addr[4] = rah_0 & 0x000000ff;
        nic_mac_addr[5] = (rah_0 >> 8) & 0x000000ff;
}
/* 追加（ここまで） */

static void get_mac_addr(void)
/* ・・・ 省略 ・・・ */
```

「NIC_REG_RAL」と「NIC_REG_RAH」は、それぞれ、N 番目のレジスタのアドレスを表すマクロとして include/nic.h へ定義しています。

RAL と RAH それぞれ 0 番目の値を取得し、RAL に格納されている MAC アドレス下位 4 バイトと、RAH に格納されている MAC アドレス上位 2 バイトを nic_mac_addr[] へ格納しています。

最後に get_mac_addr() から get_mac_addr_rar() を呼び出し、結果を出力するようにしてみます (リスト 1.6)。

▼**リスト** 1.6　A02_get_mac_02/nic.c

```
static void get_mac_addr(void)
{
        unsigned char eeprom_accessible = get_eeprom_data(0x00) >= 0;

        if (eeprom_accessible) {
                puts("EEPROM ACCESSIBLE\r\n");
                get_mac_addr_eeprom();
        } else {
                puts("EEPROM NOT ACCESSIBLE\r\n");
                get_mac_addr_rar();     /* 追加 */
        }

        /* 追加（ここから） */
        unsigned char i;
        for (i = 0; i < 6; i++) {
                puth(nic_mac_addr[i], 2);
                putc(' ');
        }
        /* 追加（ここまで） */

        while (1);
}
```

A.2.1　動作確認

実行すると、今度は MAC アドレスを取得することができました。(実行結果画像は単に MAC アドレスの表示が追加されただけなので省略)

おわりに

本書をお読みいただきありがとうございました！

本書ではついに NIC ドライバを実装し、これでようやく自作 OS も「ネットワーク対応」を謳っても嘘ではない感じになりました。(いや、まだやっぱりだめかな。。)

本書で送信するイーサネットフレーム (というか単なるバイナリ列) は独自のもので、経路上にスイッチングハブやルーターなどがあるとそれを解釈できないため捨てられます。そのため、現状では対向の PC 等とは 1 対 1 でつなぐか、リピーターハブを介しての接続しかできません。

ただ、この段階でも自作 OS としてやれることは格段に増えたと思います。少なくとも 1 対 1 なら対向 PC に対して任意のバイナリ列を送ったり、逆に対向 PC から任意のバイナリ列を受け取ったりできます。例えば本書で紹介した KOZOS プロジェクトの pkttools[*3]を使えば、シェルコマンドで任意のパケット (イーサネットフレーム) を送ったり、受信したパケット (イーサネットフレーム) をテキストで出力したりできます。

なので、例えばシェルスクリプトで「受信したバイナリ列をひたすら保存し続ける」スクリプトを作成し、対向 PC 側で実行しておけば、自作 OS 側からオレオレイーサネットフレームでデータを送信するだけで、対向 PC を簡易的な外部ストレージとして任意のデータを保存できます。

他にも、「1) 自作 OS 側から URL を送る」、「2) 対向 PC 側で"wget URL"を実行し HTML を取得、テキストとして自作 OS 側へ送信」、「3) 自作 OS 側で受け取ったテキストを表示」とすれば、プロトコルスタックを一切持たないのにブラウザっぽいものが実現できることになります[*4]。

自作 OS は当然ながら自分で実装しなければ何もできないです。ただ、外の世界とお話できる口があれば、外の誰かにお願いできるようになります。NIC ドライバを作ることに

[*3] `http://kozos.jp/software/pkttools.html`

[*4] まあ、フォントの制約上、英語のページしかまともに表示できませんが。ただ、対向 PC 側で wget ではなくウェブページ自体のスクリーンショットを取って、それを BGRA 形式の画像へ convert コマンド (ImageMagick) で変換し、画像のバイナリデータを自作 OS 側へ送るようにすれば、どんなページでも静的なページであれば表示できるようになります。

よって、このように、自作 OS 側でまだできないことを外の誰かへオフロード (?) できるので、自作 OS として NIC ドライバは早々に作ると、やれることが広がって良いんじゃないかなと思いました。(初歩的なことをやるだけならそんなに難しくないですし)

この本を読んでいただいて、興味が湧けば、ぜひ NIC ドライバ自作にチャレンジしてみてもらいたいと思います。そして、せっかくの自作 OS なのだから、汎用 OS には無い独自なことをやってみたら面白いんじゃないかなと思います[*5]。

[*5] 自作 OS で動くマシン複数台とリピーターハブを用意して、LAN 内を自分たちだけしか解釈できないオレオレイーサネットフレームで通信する、とかも面白そうです。なんの役にも立ちませんが (笑)

参考情報

参考にさせてもらった情報

- 書籍「改訂新版 PCI バス&PCI-X バスの徹底研究」(CQ 出版)
  - PCI バスのハードウェア的な信号タイミングから、ソフトウェア的に PCI のレジスタへアクセスする方法まで、PCI に関する一通りがまとめられています
  - 自作 OS で PCI を扱おうと思っているなら必携の本です
- 「PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer's Manual」
  - `https://www.intel.com/content/dam/doc/manual/pci-pci-x-family-gbe-controllers-software-dev-manual.pdf`
  - Intel 製 NIC を扱う際の一次情報源です

開発リポジトリ

本にする前のネタ作り等、本書で扱った OS の開発は以下のリポジトリで行っていますので、興味があれば見てみてください。

- ブートローダー
  - `https://github.com/cupnes/poiboot`
- カーネル・アプリ
  - `https://github.com/cupnes/yuaos`

本シリーズの過去の著作

サークル「へにゃぺんて」の同人誌は、「フルスクラッチで作る!」シリーズ含め、すべて以下の URL から PDF 版/HTML 版は無料でダウンロード/閲覧できます

- http://yuma.ohgami.jp

「フルスクラッチで作る!」シリーズの著作としては以下があります。

- フルスクラッチで作る!UEFI ベアメタルプログラミング! パート 1、パート 2
  - ブートローダーを作る上で必要な UEFI を直接叩く方法を紹介
- フルスクラッチで作る!x86_64 自作 OS(パート 1)
  - ハードウェアを抽象化しアプリへ提供するカーネルの基本的な枠組みを作り上げる
  - カーネルの機能を利用するアプリとして「画像ビューア」を作る
- フルスクラッチで作る!x86_64 自作 OS パート 2
  - ACPI で HPET 取得してスケジューラを作る本
- フルスクラッチで作る!x86_64 自作 OS パート 3
  - システムコールを実装しカーネルとアプリを分離する薄い本

フルスクラッチで作る!x86__64 自作 OS パート 4 ぼくらのイーサネットフレーム!

2019 年 4 月 14 日　技術書典 6 版 v1.0
著　者　大神祐真
発行所　へにゃぺんて
連絡先　yuma@ohgami.jp
印刷所　日光企画