special

Linuxでロボットを作る

第2回 ハードウェア操作のためのデバイスドライバ

熊谷正朗
kumagai@emura.mech.tohoku.ac.jp

はじめに

前回の7月号ではLinuxの普通のプロセスからハードウェアを制御するための方法を解説しました(記事末のRESOURCE[1]を参照)。実際、Linuxのプロセスでもかなりの制御ができます。プロセスからI/Oポート、メモリの操作が可能であるため、自作した装置のためのデバイスドライバをあえて書く必要性はそれほど高くありません。自分以外の誰かにアクセス手段を渡す場合にも、ライブラリを作成すればすむでしょう。

しかし、それでも以下のような状況ではドライバを作成する意義が存在します。

(1)利用者にrootrootrootrootrootrootrootrootrootroot権限を与えたくない場合
(2)割り込みの即応性が要求される場合
(3)ポーリング*1を行う場合

(1)は、前回触れた、プロセスからのハードウェアへのアクセスにはrootrootrootrootrootrootrootrootrootroot権限が要求されることに関連します。最終的にできたプログラムに「chmod +s」するという方法がありますが、利用者がrootになることなく開発に参加する場合に障害となります。

(2)では、どのような割り込みであるかが重要です。ハードウェアのイベント通知のみが目的の割り込みであれば、簡単な割り込み検出用のドライバを作成することで、プロセス側でも割り込みが受信することが可能となります。しかし、割り込み発生を受けて、即時ハードウェアに操作を行う必要がある場合*2、プロセスの切り替えを待つ余裕はありません。そのため、即応可能なデバイスドライバのレベルで処理する必要があります。

(3)については、前回の周期実行の手法を使うことでも可能です。しかし、たかが1bitの情報を確認するためだけにプロセスの切り替えを行うと、CPUの処理能力をそれなりに消費します。デバイスドライバであれば、カーネルがタイマ割り込み処理のついでに指定された処理を行う機能を利用できます。これを用いてポーリングし、必要ならそれを処理するプロセスに通知する実装にしたほうが効率はよくなります。

Linuxでデバイスドライバというと、一般的なインターフェイス用のドライバを思い浮かべる方は多いでしょう。これらはプロセス側の仕様が先にあって、それにハードウェアを適合させる目的のものです。それに対して、「ハードに密着した低位ライブラリ」としてもデバイスドライバを作る意義は存在します。今回は、「ハードのためのドライバ」を作ることに重点をおいて、解説します。なお、解説をたどりやすいように、サンプルプログラムを実例として付けました。

デバイスドライバの作り方

●概要

デバイスドライバを作るというと敷居が高そうに感じるのですが、失敗しなければ簡単で、ある規則に従って関数を作るだけです。難点は「失敗しなければ」という点です。普通に作ったプログラムなら、予期せぬ無限ループに入ってもkillやCtrlキー+cで止めればいいですし、ポインタの扱いを誤ってもSegmentation faultしておしまいです*3。それに対して、ドライバで同じようなミスをすると、運が良ければカーネルの警告、普通はハングアップ、最悪でディスクの内容にダメージが発生します*4。普通のプログラムとして十分にテストした関数を慎重にドライバの形に合わせるようにしましょう。

デバイスドライバを作るには、

*1 定期的にハードウェアの状態確認を行うことをポーリングといいます。「待たせる」ハードウェアは一般に準備完了を示す仕掛け(大抵はあるポートの1つのビットの0/1変化)を持ちます。準備でき次第、すぐに次のアクションを起こしたい場合は、定期的に確認する必要があります。
*2 バッファがあふれそうな場合に割り込みを発生するハードウェアでは、すぐにデータを読み取る必要があります。
*3 さらにcore fileを使って、「死因」をある程度特定可能。
*4 キャッシュと実体の不整合など。今のところ、致命的なのは経験がありませんが……。fsck待ちが精神衛生上よくありません。

(1)ドライバの機能を実現する関数を実装する。
(2)ドライバのプロセス側のインターフェイスを決め、それに合わせて処理関数とfile_operationテーブル(後述)を実装する。
(3)モジュールとして実装しカーネルに組み込む。

の3つの仕事が必要です。機能を司る(ハードウェアとやり取りする)部分の多くは、おそらく普通のプログラムとしてI/Oアクセスも含めてテストできると思います。(2)、(3)は順番にやるより平行して進めたほうがいいでしょう。まず最低限のモジュールを作って、1つずつ機能を増やしながら実装していく、という手順です。無計画ではつぎはぎプログラムになりますし、一方、1度に書き上げると不都合が起きたときの原因追求の手段が少ないため難儀します。特に、割り込みや休眠、カーネルのタイマを使うドライバは、カーネルに組み込まなければ試せません。まずは、これらを使わない暫定版のドライバを作って、それに足していくとよいでしょう。

●モジュール

まず、ドライバはモジュールとして製作します*5。モジュールはOS本体であるカーネルに動的に機能を追加するもので、多くのドライバはこの形で使えるようになっています。insmodでカーネルに組み込み、rmmodではずします。この仕掛けは良くできていて、「gcc -c」でできるオブジェクトファイル(.o)がそのまま使えます。

製作するモジュールには、必須関数として表1の2つを含める必要があります。それぞれinsmodによるモジュール登録時、rmmodによるモジュール削除時に呼ばれる関数で、初期化や終了処理を行います。何もしない場合でも空の関数を入れておきます。cleanup_module()では登録したものの解放作業を行うのですが、割り込み処理関数などカーネルから呼び出される可能性があるものの解放を忘れると、呼んでみたものの処理関数はない、という致命的状況に陥ります*6。

●デバイス操作関数テーブルの登録

デバイスドライバの機能を、通常のプログラム(プロセス)から利用するには、mknodで作る特別ファイルをopen()し、得たファイル記述子を用いて、read()、write()、ioctl()などの操作を行います。これらの要求を受けたLinuxカーネルが、処理をドライバに託すために必要なのが、操作関数を一覧にした関数テーブルです。

まず、Linuxの指定する形式で、提供する処理に対応した関数を作ります。これをfile_operation構造体に並べます(提供しない項目はNULL)。これを表2に示すregister_chrdev()によってキャラクタ型ドライバとしてカーネルに登録します。ドライバにはキャラクタ型とブロック型がありますが、普通はキャラクタ型で作ります。以後、プロセスからの要求はこのテーブル経由でデバイスドライバに届きます。この登録はinit_module()内で行います。逆にcleanup_module()において、unregister_chrdev()で解放し忘れると、実体のなくなったものをカーネルが呼ぼうとして予期せぬ(不幸な)結果になります。

ここで、メジャー番号majorなるものが出てきます。Linuxでは、種々のデバイスを区別するために、メジャー番号、マイナー番号を用います。一般的にはメジャー番号単位でデバイスドライバを区別し、マイナー番号でドライバ内の機能選択*7を行います。

表1 モジュールに必須の関数

関数名	解説
int init_module(void)	モジュール初期化関数。0を返すと初期化成功。負の数を返すと初期化エラー→insmod失敗。その際、絶対値をエラー番号とみなしメッセージが表示される。必要な変数の初期化、ハードウェアの確認(存在の有無、PCIアドレス)及び初期化、ドライバ登録、割り込み処理登録などを行う。
void cleanup_module(void)	モジュール終了処理関数。ドライバ登録解除、それまでに確保した資源(割り込みなど)の解放などを行う。

表2 キャラクタデバイスの登録

関数	解説
int register_chrdev(unsigned int major, const char * name, struct file_operations *fops);	fopsで指定された関数テーブルをメジャー番号majorのデバイスとして登録する。nameは識別名の文字列。登録成功で0を返し、majorが使用されていると-EBUSYが返る。
int unregister_chrdev(unsigned int major, major const char * name);	のデバイス登録を解除する。name が登録時と一致しない場合、-EINVALが返る(チェック用)。

*5 カーネルソースに直接組み込む方法もありますが、当然テストのたびに再起動が必要なので不便すぎます。
*6 しかも、rmmodしてもメモリにコードが残っていることがあり、時間をおいて何かに上書きされたころに落ちるという可能性がありますので、要注意です。
*7 /dev/mem、/dev/kmem、/dev/ioportは機能は異なりますが、同じメジャー番号1を持ちます。また、/dev/hda、hda1-16は同じメジャー番号3で、パーティションごとに異なるマイナー番号を持ちます。参考:ls -l /dev

カーネル側はopen()を要求されると、特別ファイルに指定されたメジャー番号から登録されているドライバを選択し、ドライバのopen()処理関数にマイナー番号を渡します。マイナー番号の使い道はドライバ次第です*8。このメジャー番号はすでにかなり予約されています*9。とりあえず、「LOCAL/EXPERIMENTAL USE」と記載された番号を使用すると良いでしょう。それでも、他のドライバと重なる可能性はあり、後述する方法でinsmod時に変更可能としておくと便利です。

あとは、プロセスとのインターフェイスの仕様を決定し、ドライバ側の関数を実装するだけです。

●どんな関数を登録するか

カーネルに登録する関数テーブルには、ドライバとして提供する機能を記述します。機能はシステムコールにほぼ1対1で対応した名前を持っています。そのうち今回取り上げる主要なものと、そこで使う構造体を表3に示します。

この中から自分でサポートするつもりのものを実装すれば良いわけです。最低open()とrelease()は必要でしょう。特に、open()ではMOD_INC_USE_COUNTマクロを、release()ではMOD_DEC_USE_COUNTを使います。これは、モジュールの利用度数を増減させるものです。初期値が0で、0のときのみrmmodでモジュールを外せます。これで、使用中のモジュールが無くなるのを防ぐわけです*10。また、open()の際には、最大利用数の確認なども行います。排他的にしか使えないデバイスであれば、どこかに使用中のフラグを用意しておいて判断し、使用中なら-EBUSYを返すなどします。これらの関数が負の数を返すと、システムコールはその絶対値をerrnoにセットして、エラー応答することになります*11。

以下、状況別にこれら関数の使い道を述べていきます。

一撃離脱のモジュール

最初から、邪道な話です:-)。PCIデバイス検出の試験を行いたい、カーネル内部の情報を調べたいという場合、人間が目で確認すればいい程度であれば、わざわざドライバにしなくとも、簡単なモジュールを1つ作れば、実験できます。

モジュールにはinit_module()とcleanup_module()という2つの関数が最低限あればいいことになっています。当然これらはカーネル内部の関数や変数を使用できます。また、init_module()が0を返さない場合、それはエラーとして扱わ

表3a　主なドライバ操作関数

操作	関数	解説
open	int open(struct inode *, struct file *);	SC open()に対応。デバイスへのアクセスの可否(デバイスの占有など)を確認し、可能なら初期化などを行う。また、モジュールがrmmodされないように細工。必要なら、inode構造体からマイナー番号を取り出し、対応する。
release	int release(struct inode *, struct file *);	SC close()などで呼ばれる。アクセスの終了処理を行う。
read write	ssize_t read(struct file *, char *, size_t, loff_t *); ssize_t write(struct file *, const char *, size_t, loff_t*);	SC read()、write()に対応。ファイル記述子fdがfile構造体になるが、残りはそのまま渡される。loff_t *で渡されるのは&file->f_posである(fs/read_write.cに詳しい)。
llseek	loff_t llseek(struct file *, loff_t, int);	SC lseek()に対応。未定義の場合、それらしい動作をするデフォルト関数(default_llseek@fs/read_write.c)が代行する。
ioctl	int ioctl(struct inode *, struct file *, unsigned int, unsigned long);	SC ioctl()に対応して呼び出される。第3、4引数はioctlの第2、3引数がそのまま渡される。
select (2.0) poll (2.2-)	unsigned int poll(struct file *,struct poll_table_struct *); int select(struct inode *, struct file *, int, select_table *);	デバイス待機SC select()、poll()に対応して、内部で呼び出される。カーネルのバージョンの変化にともない、最も変化した操作関数で、2.0系select形式と2.2以降poll形式でまったく異なる。

＊注　SC：システムコール――引数の型などはバージョンによって異なる場合がある。以上すべてinclude/linux/fs.hを参照。

表3b　主な構造体

構造体	解説
file構造体	ファイル記述子と関連した情報を持つ。メンバ変数f_versionはopen()のたびに異なる値を持つため、アクセスを区別するのに便利。同f_posは読み書き位置を保持する変数で read、write、lseekなどで使うと良い。同private_dataは、ドライバで自由に使って良いvoid型ポインタで、独自の管理データのポインタなどを入れると便利。
inode構造体	inodeの情報を持つ構造体であるためあまり触ることはないが、MINOR(inode->i_rdev)により、デバイスのマイナー番号を得られる。

＊8　1 byteの初期パラメータ1つを渡すという使い方も可能です。
＊9　/usr/src/linux/Documentation/devices.txtを参照。
＊10　開発段階でうっかりすると、このカウントのつじつまが合わなくなって、モジュールを外せなくなることがあります。それに対処するには、例えばマイナー番号250、251あたりでopenすると、カウントの増減だけをしてエラーになるように実装しておくのも一案です。
＊11　つまり-EBUSYを返すと、システムコールopen()は-1を返し、errnoにEBUSYがセットされます。他に、EINVALが、引数のミスを示すために良く使われます。

れ、モジュールは登録されません。よって、

わざと「無条件に」エラーを返すinit_module()と空のcleanup_module()を作り、init_module()で好きなだけやって結果はprintk()で出力。

ということを思い付きます。邪道もいいところですが、使ったことがないカーネル内機能を試すとき、非常に便利な方法です。

ここで使うprintk()は見た目の通り、カーネル(k)で使えるprintf()みたいなものです*12。簡単に確認できる出力先を表4にまとめておきました。私は「nice --20 tail -f /proc/kmsg」が一番便利だと思っています。

簡単な例をリスト1に示します。printk()だけでは面白くないので、モジュールにパラメータを渡す例も示しておきます。適当にint、char*の変数を用意した上で、カーネル2.1.18以降ではMODULE_PARMで宣言すると、それ以前は手続きなしでinsmodの引数で「変数=値」の形でパラメータを渡せます。前述のメジャー番号を登録時に設定できるようにする際にも便利です。冒頭のMODVERSIONSに関わる部分は、異なるバージョンのカーネルにモジュールを組み込む危険を避ける目的のチェック機構に対応するためです。カーネル再構築時の設定にある「Loadable module support」→「Set version〜」が「y」になっている場合はこれが必要です。

なお、モジュール内で変数や関数を定義する場合、原則としてstaticにする必要があります。

表4　printkの出力先

出力先	解説
コンソール	(テキストの)コンソールには、直接出力される。コンソールに切り替えて、そこでinsmodすれば、その場で確認できる。
dmesg	コマンドdmesgで、直前まで出力されていたカーネルのメッセージを見ることができる。
syslog	/etc/syslog.confにてkern.*を指定していると、カーネルのメッセージを記録してくれる。(man syslogd、man klogdを参照)
/proc/kmsg	rootにて、tail -f /proc/kmsgすると、そこにメッセージが表示される。ただし、klogdなどが動作していると取り合いになるらしく、一部出力が欠る。この場合、nice --20などをつけて強硬に奪うようにする。

実行例1　リスト1の実行例

```
% gcc -c moduletest.c -Wall -Wstrict-prototypes -O
# insmod moduletest str=aaa var=10
# dmesg | tail -1
module being installed at 1877498 var= 10 str=aaa
```

大量やり取りのためのread/write

●概要

プロセスとデバイスドライバで情報をやり取りする際に広く使われているのはread()とwrite()です。この2つのシステムコールは、指定バイト数のデータを読み書きするもので、実際に読み書きした量だけ操作ポインタが移動し、その量を返す、とされています。基本的に大量のデータ塊を扱う操作です。

こういう設計にすると直感的に扱えるのは確かですが、一歩拡大解釈すると、毎回特定のデータ領域の先頭から読み書きする、すなわち操作ポインタを管理しない使い方でもいいわけです。同じ領域を繰り返し読み書きすることが前提なら、lseek()してわざわざポインタを戻す手間が省けます。前回紹

リスト1　モジュールの実験

```
// gcc -c moduletest.c -Wall -Wstrict-prototypes

#define MODULE
#define __KERNEL__

// シンボルがバージョン付の場合に対応
// 例：jiffies_Rsmp_0da02d67 <= jiffies
#include <linux/autoconf.h>
#if defined(CONFIG_MODVERSIONS) && \
                        !defined(MODVERSIONS)
# define MODVERSIONS
#endif
#ifdef MODVERSIONS
# include <linux/modversions.h>
#endif

#include <linux/module.h>
#include <linux/kernel.h> // printk
#include <linux/sched.h>  // jiffies

static int var=0;
static char *str="default";
#if LINUX_VERSION_CODE > 0x20112
MODULE_PARM(var, "i");
MODULE_PARM(str, "s");
#endif

int init_module(void)
{
  printk("module being installed at %lu "
         "var= %d str=%s\n",jiffies,var,str);
  return -1;   // rmmod しないで済むように手抜き
}

void cleanup_module(void)
{
};
```

*12　もともとカーネルでは浮動小数点は使えないようで(そのため、固定小数点で苦労しているところがあります)、printk()も%fなどは使えなさそうです(lib/vsprintf.c)。

介した脚歩行ロボットの制御システムに用いているプロセス間通信用のデバイスドライバは、毎回固定長のブロック状のデータをやり取りするため、このような仕様にしました。さらに二歩拡大解釈すると「整数とポインタをもらって整数を返す操作」にも使えますが、さすがにやりすぎかもしれません。

さて、read()とwrite()が適する用途は以下のようなものと言えます。

(1)メモリ型のデバイス。他のコンピュータと接続する共有メモリボードや、画像ボードのようにメモリ空間に配置されるハードウェア。/dev/memと同機能。

(2)ストリーム型のデバイス。時間の経過に伴う、連続性のあるようなハードウェア*13に使用。その場合、情報の送受とプロセスの読み書きタイミングに差があるのが一般的であり、FIFOバッファ*14をドライバ上に作り、整合を取る。具体的には、ハードウェア側はポーリングや割り込みによってバッファに順にデータを書いていき、read()が実行されたときに、たまっているだけ返せば良い。データが1つも無かったときにどうするか、溢れたときにどうするかは対象の性質次第である。

このread()とwrite()は次に説明するioctl()と違って、普通は1つの対象しか扱えません。そのため、多機能なハードウェアのドライバに使うには、若干難があります。もし、メモリ型にもストリーム型にも適合せず、多機能少量アクセスのみなら、いっそのこと、read()、write()を付けるのをやめてしまうことも選択肢の1つです。

read()、write()に対応する利点を最後に1つ挙げておきます。ioctl()を使う場合、その動作試験のためにはプログラムを書かなければなりません。それに対して、read()、write()の場合は、作りにもよりますが、catや「echo >」で動作確認できて便利です。

●実装

read()、write()は、デバイスドライバ側では、file構造体と、システムコールの引数、操作ポインタが渡されます(表3a)。システムコールの引数を利用して、適宜情報をやりとりする実装にすればいいわけです。1番いい例は/dev/mem、linux/drivers/char/mem.cです。#ifで場合分けされたところを除いてみると、まず読み書き可能な範囲かどうかを操作ポインタと要求量から判断し、可能な分だけ処理し、ポインタを操作するなどします。

ここで要注意なのは、システムコール経由で渡されるプロセス側のポインタは、プロセスのメモリ空間のものであって、カーネルのメモリ空間のものではないことです*15。そこで、2つのメモリ空間で情報をやり取りするために、特別の関数が用意されています(表5)。char、short、long単位での操作とmemcpy()型の操作とがあります。カーネル2.0と2.1以降で形式が変わりました。get_userは書き換えが必要ですが、残りは引数が同じですので、#defineなどで対処可能です。

小回りの効くioctl

ioctl()。ドライバを書こうなんて思う前の私は、危なげなソース*16でこの関数を見るとげんなりしてました(fcntl()も)。man ioctl()とやっても、具体的な使い方は見当たらず、よく分からないまま使うしかありませんでした。ところが、ドライバを書くようになったら、とても便利だと分かりました。「多機能少量アクセス」がキーワードです。ドライバにあの機能を付けたい、これを付けたいと、多くの機能を持たせる場合にうってつけです(だてにI/O controlではありません)。ではなぜ、具体的な使い方がman ioctl()で分からないか? それはドライバ作者が決めることだからです。

ioctl()は、システムコール側では、整数を1つと、何かもう1つが引数になっています。ドライバ側にはunsigned intとunsigned longで渡されます。ドライバ側から負の数を返すと、errnoに絶対値がセットされ、システムコールは-1を返します。仕様はそれだけです。

具体的な使い方はいろいろできますが、一般的には1つめの引数をコマンドとして、これでswitchして、処理を振り分けます。

- ドライバに何らかの判断を求める:0 or 負値を返す。
- 動作モードを切り替える:read()、write()のアクセス対象を切り替えるなども一案。
- ハードウェア(ドライバ)への数値渡し:2つ目の引数を整数値と見なせばよい。
- ハードウェア(ドライバ)への、またはハードウェアからの数値「群」の引き渡し:2つ目の引数を数値群を格納した構造体や、配列、文字列へのポインタとして解釈すればよ

*13 A/D変換によるセンサ情報の連続的な取得や、通信回線など。
*14 First-In First-Out。先に入ったものが先に出る、待ち行列みたいなもの。pipeやsocketの類いはFIFOになっています。
*15 似た理由で、前回は物理メモリにアクセスするため、/dev/memを使いました。
*16 NUTSHELLシリーズ。アスキーのDavid A.Curry著「UNIX C プログラミング」は愛読書です:-)。

表5　カーネル空間とユーザ空間のデータの送受

データの流れ	関数 (kernel 2.0、asm/segment.h)	関数 (kernel 2.1〜、asm/uaccess.h)	解説
ユーザー → カーネル	memcpy_fromfs(void *to, const void *from, unsigned long n);	copy_from_user(void *to, const void *from, unsigned long n);	ユーザー空間の*fromで指定される領域から、カーネル空間の*toにnバイト転送。memcpy()と扱いは同じ。
	type get_user(type *user);	int get_user(type val, type *user);	ユーザー空間から1、2、4bytes取得する。2.1以降の形式は、ユーザー空間の領域チェックをするようになった。成功で0、失敗で-14(EFAULT)を返す。値はvalに代入(マクロ)。
カーネル → ユーザー	memcpy_tofs(void *to, const void *from, unsigned long n);	copy_to_user(void *to, const void *from, unsigned long n);	カーネル空間の*fromから、ユーザー空間の*toにnバイト転送。
	put_user(type value, char *user);	int put_user(type value, char *user);	ユーザー空間に1、2、4 bytes渡す。2.1以降の形式は、ユーザー空間の領域チェックをするようになった。成功で0、失敗で-14(EFAULT)を返す。

(注)形式変更は kernel 2.1.0〜kernel2.1.6あたり(流動的)。

い。この場合、いったん表5に示したような転送関数によってカーネル空間に構造体を持ってきて処理を行う。処理後、必要ならユーザー空間に戻す。

など、多彩な機能を実装できます。単発的な処理をいろいろ扱う場合はread()、write()に比べて圧倒的に適しています。

注意点としては、冒頭で述べたような問題が挙げられます。コマンド数値と引数の関係をしっかり文書化するか、ドライバと対になるライブラリまで整備した上で、それを公開するかしなければ、大変なことになるでしょう。

ハードウェア待機のためのpoll/select

●概要

Linux(UNIX)にはselect()、poll()という、非常に便利な仕掛けがあります。プロセスが複数の入出力経路の読み書き準備ができるまで待つために、定期的に1つずつ調べていては効率が悪くなります。そこで、プロセスは眠らせてCPUを使わない状態で待たせておき、カーネル側で準備ができた段階でそれを起こす機能があると便利です。それが select()および poll()で、複数の入出力を同時に待機し、1つでも準備ができると実行が再開されるようになっています。

カーネル側でいちいちチェックする必要があるかというと、実はそうでもありません。このような入出力の準備が整うというのは、大抵はデバイスの割り込みや、他のプロセスの動作などが原因となります。例えば、キーボードから届いた情報はカーネルで処理されて、現時点でキーボードデバイスを所有しているプロセス、Xサーバなどに送られます。Xサーバは届いた情報を処理して、適当なウインドウを持っているプロセス、ktermなどにネットワークソケットなどを介してキーのイベントを送ります。それが疑似端末を通して、ktermの上で動作しているプロセスの標準入力に送られ……*17、と最終的にユーザーの目の前に届きます。すべてのプロセスは寝て待ち、キーボードデバイスドライバが入力を検知したらそれを待つXを起こし、ktermが書き込めば疑似端末のドライバがその上のプロセスを起こします。つまり、カーネル(ドライバ)はチェックしているというより、「何か情報が届く処理を要求されたついでに、それを待つプロセスを起こす」という実装にすればいいわけです*18。

似た仕掛けに「ブロック」があります。これはread()やwrite()などを要求されたにもかかわらず、対象の準備ができていないような場合に、ドライバ内部で自動的に休眠に入り、待機するものです。同時に複数を待つことはできませんが、1つの場合は便利です。デバイスドライバとしてブロックを使うかは対象にもよるでしょうが、即応性が重要な場合には予期せぬブロックが起きるとむしろ困ります。ブロックを実装する場合にも、ioctl()などで有効無効を設定できるようにしたほうが良いでしょう*19。

*17　さらに、プロセスが入力された文字を処理し(端末としてエコーバックするかもしれませんが)出力して、ktermがXサーバに送って、Xサーバがハードウェアに書き込んで、ついに入力した文字が画面に出ます:-)。あまつさえ、複数のプロセスがパイプでつながっているかも……。

*18　割り込みも何もないハードウェアなら、仕方ないのでデバイスドライバが定期的にチェックしなければなりません。それでも、プロセスがいちいち調べるより処理は速くなります。書き込みの準備待ちは、例えば受け側のプロセスの処理が進んでおらず、途中のバッファが溢れたかどうかを見るためなどに使用されます。

*19　特にコードを書かなくとも、ioctl()でブロックの設定をする際に標準に使われるFIONBIOの処理はしてくれます。file構造体のメンバf_flagにO_NONBLOCKが設定されます。参考：linux/fs/ioctl.c、sys_ioctl()

●実装

すでに述べましたように、ここがデバイスドライバ関係で一番仕様が変わったところです。カーネル 2.1.22までは、file_operation構造体の定義でselect()と呼ばれ、待ちの種類(入力待ち・出力・特別)を示す引数があり、それを判断して準備ができていれば1を、できていなければ休眠待機の設定をしたうえで0を返すものでした。つまり複数の条件を確認する場合は、複数回呼び出されます。それに対して、カーネル2.1.23からは poll()と名前を変え、全種類の準備状況を定数の組み合わせで返すようになりました。それぞれシステムコールselect()、poll()に適した形になっています。以前はpoll()はライブラリでselect()を使ってエミュレートしていましたが、現在はpoll()もシステムコールになり、select()にはカーネル内でドライバのpoll()を使って対処しています。

寝かせる→起すの動作をするために、カーネル 2.0、カーネル 2.2では条件ごとに「strcut wait_queue *」の変数を用意します(複数の要因をもつドライバはその数だけ必要です)。read()などでブロックする場合には、その場でinterruptible_sleep_on()にこの変数のポインタを渡します。この段階で、このプロセスは休眠に入ります。別のプロセスのアクションや、ハードウェアからの割り込みなどで起こす条件が整ったとき、その条件に対応したwait_queue *変数のポインタをwake_up_interruptible()に渡します*20(いずれも、wait_queueのポインタのポインタを扱います)。それに対して、select()、poll()の場合は、select_wait()、poll_wait()を使って、休眠の仮登録をします。システムコールを処理するカーネル側で、対象すべてを調べたうえで、1つも準備ができていなければ、休眠に入ります。1つでも準備ができている場合は休眠せず、そのままシステムコールを終了します。休眠に入ったあと何らかの原因で起こされると、再度すべての対象を調査し、準備ができた場合、待機時間が経過した場合、シグナルが発生した場合にシステムコールを終了します。そうでなければ、再度休眠に入ります。

カーネル 2.4の場合、既存のソースを読み取る限り、wait_queue *に代わってwait_queue_head_t型の変数をそのまま使えばいいようです。その他は変更することなく、後述するサンプルのコンパイルも通りました。

サンプルドライバプログラム

ここまでの話の参考となるように、サンプルプログラムを書きました(リスト2-1)。本当は適当なハードウェアのドライバにしたいところですが、どこでも動くように、ハードウェアへのアクセスは入っていません。ソフトウェアのみのドライバで、open、release、read、write、ioctl、pollを組み込んであります。なお、誌面の節約と複雑化するのを避けるため、バージョン依存部分を切り、カーネル2.2専用としています。printk()によるメッセージを含む版、カーネル 2.0、カーネル 2.4版は筆者のWebページ([2])をご覧ください。

具体的には、0からMAXMESSAGE-1の番号がついた一言伝言板として機能します。伝言板の選択はマイナー番号、もしくはioctl()にて行います。読み書きは初回のread()、write()のみ有効で、2回目以降はread()では0を返し読み込み終了を通知、write()では-ENOSPCを返し、デバイスに余裕がないことを通知します。また、ioctl()にて、「リフレッシュ」と称して再度読み込み可能としました。参照している伝言板に他のプロセスが書き込んだ場合にも、自動的にリフレッシュします。さらに、select()、poll()によって、write()時にリフレッシュされるのを待機することができます。

以上の機能を確かめるには、mknodで/dev/ljtest[0-3]を作った上で、

- cat /dev/ljtest0、echo 'foo' > /dev/ljtest0のようにして、特定の伝言板に対して読み書き。
- リスト2-2に示した試験用プログラムを作成する。このプログラムを実行しつつ、/dev/ljtest1に何か書き込む。

といった 操作を行います。ioctl()まわりのプロセス側(システムコール)とドライバの関係は両者を見比べると分かりやすいかと思います。

便利帳

だいたい、ここまで述べてきたような動作を実装することで多くのデバイスドライバのインターフェイスが作れると思います。しかし、実際にハードウェアとやり取りするドライバを作るには、さらにいくつか知るべきことがあります。ここでは それについて補足します。

●ハードウェアの操作

I/Oポートに関しては、前回述べた方法がそのまま使えます。異なるのは「許可が不要」という点です。自由にinb()などを使用できます。

*20 同じ条件を待つ複数のプロセスが寝ていると、それらを一気に起こします。もし、そのwait_queueで寝ているものがないと、何も起きません。

リスト2-1　テスト用ドライバ

```
// test driver for Linux 2.2
// open-read-write-ioctl-poll-release
// gcc -c ljtest22.c -O -Wall -Wstrict-prototypes
// mknod  -m 0666 /dev/ljtest[0-3] c 60 [0-3]

#define MODULE
#define __KERNEL__

#include <linux/autoconf.h>
#if defined(CONFIG_MODVERSIONS)
               && !defined(MODVERSIONS)
# define MODVERSIONS
#endif
#ifdef MODVERSIONS
# include <linux/modversions.h>
#endif

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/fs.h>
#include <linux/string.h>
#include <linux/poll.h>
#include <asm/uaccess.h>

static int devmajor=60;
static char *devname="LJTest";
MODULE_PARM(devmajor, "i");
MODULE_PARM(devname, "s");

// アクセス管理部
#define MAXACCESS 10
struct AInfo {
  unsigned long f_version;
  int id,fresh;
};
static struct AInfo ainfo[MAXACCESS];
// メッセージ保持部
#define MAXMESSAGE 4
#define MAXMLEN 256
struct Mess {
  int length;
  char message[MAXMLEN];
  struct wait_queue *wait;   // 休眠待機用
};
static struct Mess mess[MAXMESSAGE];

static int ljtest_open(struct inode * inode,
                               struct file * file)
{
  int i,minor=MINOR(inode->i_rdev);
  if(minor==240) { MOD_INC_USE_COUNT;
                                 return -EBUSY; }
  if(minor==241) { MOD_DEC_USE_COUNT;
                                 return -EBUSY; }
  if(minor>=MAXMESSAGE) { return -EINVAL; }
  // アクセス条件成立
  for(i=0;i<MAXACCESS;i++)
    if(ainfo[i].f_version==0) break;
  if(i==MAXACCESS) { return -EBUSY; } // 満席
  MOD_INC_USE_COUNT;
  ainfo[i].f_version=file->f_version;
  ainfo[i].id=minor;
  ainfo[i].fresh=1;
  file->private_data=(void *)(&ainfo[i]);
  return 0;
}

static int ljtest_release(struct inode * inode,
                          struct file * file)
{
  struct AInfo *ai=(struct AInfo *)
                (file->private_data);
  if(ai==NULL) { printk("something wrong?\n"); }
  else ai->f_version=0;
  MOD_DEC_USE_COUNT;
  return 0;
}

static int ljtest_read(struct file * file,
       char * buff, size_t count, loff_t *pos)
{
  int len,id;
  struct AInfo *ai=(struct AInfo *)
                            (file->private_data);
  if(ai==NULL)
    { printk("something wrong?\n");
                                  return -EINVAL; }

  id=ai->id;
  if(!ai->fresh) {
    len=0;
  } else {
    len=mess[id].length;
    if(len>count) len=count;
    copy_to_user(buff,mess[id].message,len);
    ai->fresh=0;  // 読み終り
  }
  return len;
}

static int ljtest_write(struct file * file,
      const char * buff, size_t count, loff_t *pos)
{
  int len,id,i;
  struct AInfo *ai=(struct AInfo *)
                            (file->private_data);
  if(ai==NULL)
    { printk("something wrong?\n");
                                  return -EINVAL; }

  id=ai->id;
  if(!ai->fresh) {
    return -ENOSPC;
  }
  len=count;
  if(len>MAXMLEN) len=MAXMLEN;
  copy_from_user(mess[id].message,buff,len);
  mess[id].length=len;
  ai->fresh=0;  // 書き終り
  // リフレッシュ処理＋起し
  for(i=0;i<MAXACCESS;i++) {
    if(ainfo[i].f_version==0) continue;
    if((!ainfo[i].fresh)&&(ainfo[i].id==id))
      ainfo[i].fresh=1;
  }
```

リスト2-1 （つづき）

```
  wake_up_interruptible(&(mess[id].wait));
  return len;
}

static int ljtest_ioctl(struct inode *inode,
           struct file *file, unsigned int cmd,
                              unsigned long arg)
{
  struct AInfo *ai=(struct AInfo *)
                              (file->private_data);
  if(ai==NULL)
    { printk("something wrong?\n");
                                return -EINVAL; }

  switch(cmd) {
  case 1:
    ai->fresh=1;  return 0; // refresh
  case 2:
    if((arg<0)||(arg>=MAXMESSAGE)) return -EINVAL;
    ai->id=arg;
    return 0;
  }
  return -EINVAL;
}

static unsigned int ljtest_poll(struct file *file,
         struct poll_table_struct *ptab)
{
  struct AInfo *ai=(struct AInfo *)
                              (file->private_data);
  if(ai==NULL)
    { printk("something wrong?\n");
                                 return -EINVAL; }

  if(ai->fresh)
    return POLLIN|POLLRDNORM;         // 読み込みOK
  poll_wait(file,&(mess[ai->id].wait),ptab);
  return 0;
}

static struct file_operations ljtest_fops = {
  NULL,          // llseek
  ljtest_read,   // read
  ljtest_write,  // write
  NULL,          // readdir
  ljtest_poll,   // poll
  ljtest_ioctl,  // ioctl
  NULL,          // mmap
  ljtest_open,   // open
  NULL,          // flush
  ljtest_release,// release
  NULL,          // fsync
  NULL,          // fasync
  NULL,          // check_media_change
  NULL,          // revalidate
  NULL,          // lock
};

int init_module(void)
{
  int i;
  for(i=0;i<MAXACCESS;i++) ainfo[i].f_version=0;
  for(i=0;i<MAXMESSAGE;i++) {
    sprintf(mess[i].message,"Test message %d\n",i);
    mess[i].length=strlen(mess[i].message);
    init_waitqueue(&(mess[i].wait));
  }
  if(register_chrdev(devmajor,devname,
                                  &ljtest_fops)) {
     printk("device registration error\n");
     return -EBUSY;
    }
  return 0;
}

void cleanup_module(void)
{
  if (unregister_chrdev(devmajor,devname)) {
    printk ("unregister_chrdev failed\n");
  }
}
```

メモリに関しては、メインメモリや1Mbytes未満の領域など、物理的にアドレスが固定されている領域に対しては、常にカーネルからアクセス可能な状態にあるため、`phys_to_virt()`という関数で直接アクセス用のポインタを得ることができます。やることは結果的に同じですが、`readb()`、`writeb()`というマクロも定義されてます。

それに対して、PCIバス上のハードウェアのメモリはカーネル空間に最初からは存在しません。まず、これをカーネル空間に接続(マップ)する必要があります。これには表6に示す関数を使います。後述の方法などによって取得したPCIデバイスのベースアドレスを`ioremap()`に渡すと、それがカーネル空間からアクセスできるようになり、そのマップした場所をポインタによって返します。あとはそのポインタを基準に読み書きすればいいわけです。対になるものに`iounmap()`がありま

表6　物理メモリのアクセス(asm-i386/io.h)

関数	解説
`void * phys_to_virt(unsigned long address)`	アドレス`address`の物理メモリにアクセスできるポインタを得る。

`void * ioremap(unsigned long offset, unsigned long size)`
`void * ioremap_nocache(unsigned long offset, unsigned long size)`
`void * vremap(unsigned long offset, unsigned long size);  (2.0)`
	アドレス`offset`から`size`バイトのアクセスが可能となるよう、カーネル空間に物理アドレスをマッピングして、そのポインタを与える。`ioremap_nocache`は、キャッシュさせないように指示するもので、`memory-mapped I/O`などに適する(ただし、普通はPCI側でキャッシュ不可になっているはず)。

`void iounmap(void *addr)` `void vfree(void * addr);  (2.0)`	`ioremap` でマップした領域を解除する。

注：kernel 2.1.0以降はio〜、それ以前は v〜、本文はio〜で始める形で表記を統一。

リスト2-2　ioctl、selectテストプログラム

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <sys/time.h>

void refresh(int fd)
{  ioctl(fd,1);   }

void change_id(int fd,int id)
{  ioctl(fd,2,id);   }

void verify_message(int fd)
{
  char buff[256];
  int r=read(fd,buff,256);
  if(r<0)
    { perror("verify_message"); return; }
  if(r==0)
    { printf("not fresh\n"); return; }
  printf("message: %.*s",r,buff);
}

int wait_refresh(int fd)
{
  fd_set fds;
  struct timeval tv;
  int r;
  FD_ZERO(&fds);  FD_SET(fd,&fds);
  tv.tv_sec=10; tv.tv_usec=0;
  r=select(fd+1,&fds,NULL,NULL,&tv);
  if(r<0)
    printf(
         "select returned with signal or error\n");
  else if(r==0)
    printf("time out\n");
  else if(FD_ISSET(fd,&fds)) {
    printf("refreshed\n");
    return 1;
  }
  return 0;
}

int main(void)
{
  int fd;
  char buff[256];

  fd=open("/dev/ljtest0",O_RDWR);
  if(fd<0)
    { fprintf(stderr,"cannot open device\n");
                                        return 1; }

  verify_message(fd); verify_message(fd);
  refresh(fd);        verify_message(fd);
  change_id(fd,1);    refresh(fd);
  verify_message(fd);

  while(1)
    { verify_message(fd); wait_refresh(fd); }
  close(fd);
  return 0;
}
```

す。これはマップを解除するもので、使用後(cleanup_module()などで)呼び出す必要があります。忘れても一見何事もないように見えますが、カーネル空間にゴミ領域として残るため、いずれ問題が出かねません。

簡単な実例(init_module()のみ)をリスト3に示します。このリストは手元のPCのビデオカードのVRAMを16bytes読むようにアドレスを書いてあります。何回かinsmodしてみると、同じポインタが帰ってきますが、iounmap()でわざと外すと、異なるポインタが返ってくるようになります。

●割り込みの利用

MS-DOS時代のプログラムで割り込みを使う場合、割り込みコントローラの設定や、割り込み処理ルーチンの登録*21など、すべて自分でコードを書く必要がありました。それに比較するとLinuxは非常に簡単です。関数を1個作って、request_irq()でカーネルに登録するだけです(表7)。あとは割り込みが発生すると、カーネルが割り込みコントローラなどを処理し、登録されている割り込み関数を呼び出します。割り込み対応をやめるときは(cleanup_module()で忘れずに)、free_irq()を呼びます。登録する関数で対象ハードウェアそのものの割り込み関係の処理を施し、必要ならwake_up_interruptible()を呼び出せばよいでしょう。

なお、割り込み処理関数には低速と高速の属性が指定できます。割り込み発生後、カーネルの割り込み処理ルーチンから呼び出すか、落着いてからあとで呼ぶかの違いがあります。今どきのCPUは速いので、必要な処理のみ簡単に済ませる、という割り込み処理の原則を満たした関数なら、高速扱いで登録して大丈夫でしょう。

リスト3

```
int init_module(void) {
  int i;
  unsigned char *map;
  map=(unsigned char *)ioremap(0xe4000000,16);
  printk("ioremap: %p\n",map);
  for(i=0;i<16;i++)  printk("%02X ",map[i]);
  printk("\n");
  iounmap(map);
  return -1;
}
```

*21　8086の場合、メモリの0番地から1024bytesは割り込みベクトルテーブルとして使われ、4bytesずつ、0〜255までの割り込み発生時の呼び出し先が記載されていました。

表7　割り込みを使うための関数(linux/sched.h)

割り込み	関数	解説
登録	int request_irq(unsigned int irq, void (*handler)(int, void *, struct pt_regs *), unsigned long flags, char *device,void *dev_id);	割り込み処理関数handler()を登録する。irqは割り込み番号、deviceは/proc/interruptsで表示する識別名を示す。dev_idは何でも良いが、登録を解除するときの確認用に固有の値である必要がある。また、handlerの第2引数として渡されるため、処理関数側で必要なデータを渡すのに便利(staticな変数でも可であるが)。flagsは割り込み関数の性質を指定する。SA_SHIRQを指定すると割り込みの共有が可能、SA_INTERRUPTを指定すると高速処理が可能なルーチンであることを示す(「｜」で結合)。
除去	void free_irq(unsigned int irq, void *dev_id);	割り込み番号と、登録時に指定したdev_idを指定。両者が一致する場合に登録が解除される。

表8　カーネルタイマの利用

―	関数	解説
登録 解除	void add_timer(struct timer_list * timer); int del_timer(struct timer_list * timer);　(linux/timer.h)	構造体timerに従ってタイマをセット・解除する。
構造体	struct timer_list { 　struct timer_list *next,*prev; 　unsigned long expires; 　unsigned long data; 　void (*function)(unsigned long);};	next、prevはカーネル内部での管理用、expiresは登録する関数function()が呼ばれる時刻をjiffiesで指定、dataは function()に引数として渡されます。 指定例: {NULL, NULL, 0, jiffies+10, cyclefunc};

●ポーリングのためのタイマ利用

割り込みを発生する機能がなく、ソフトの側で状態をこまめにチェックする必要があるハードウェアも存在します。この場合、カーネルのタイマ割り込み処理時に呼び出してくれるタイマ機能を利用すると良いでしょう。

これも使い方は簡単です。unsigned longを引数とする関数を1つ作り、add_timer()で登録します(表8)。これは時刻expiresに呼び出す単発のタイマで、カーネル内の時間変数jiffiesを使って指定します*22。そのため周期的に呼びたい場合は、登録する関数の中で再度expiresをjiffies+に設定してadd_timer()します。注意点は、cleanup_module()ではdel_timer()しなければならないことと、時刻設定の基準が

COLUMN 1

カーネルソースは最強の参考書

現時点で私が知る、デバイスドライバ作成の際に参考となる良書は、「Linuxデバイスドライバ」([3])です。ドライバを深く追求したい方は、手元に用意しておくと便利だと思います。しかし、書籍の最大の弱点である、実体からの遅れと更新サイクルの長さが目立ちます。では何を信じるかというと、Linuxカーネルのソースそのものです。ソースから仕様を拾うのは好ましくないのかもしれませんが、「動く」デバイスドライバソースの固まりであるカーネルソースは非常に良い参考書になります。私自身、この本よりカーネルから得た情報のほうがはるかに多いです。

しかしカーネルソースは膨大ですので、どこを見るかが重要です。私がいつも確認するのは表9に示すようなものです。ドライバについては、まずfile_operationを検索します。すると、ドライバの関数テーブルが見つかります(無いファイルはそこで終了)。そこから、それぞれの関数を追っていき、内部で使っているハードウェアに関わる部分を無視して流れをみます。mem.c、pc_keyb.cは分かりやすい部類です。あとはこの辺りのディレクトリでfile_operationをgrepしてみたり、使いたい関数をキーワードとしてgrepすると、かなりの情報が得られます。　(熊谷正朗)

表9　参考となるカーネルソースの一例

ソースファイル名	解説
char/mem.c	最も単純なread/write型のドライバ。これを見ると、基本的な実装の変化が分かる。
char/pc_keyb.c	PS/2キーボードおよびマウスのドライバ。居候のマウスのドライバ部分が比較的単純で、割り込み、pollなども使っていて、例として便利。
drivers/net/*	ネットワークのドライバ類。普通のデバイスドライバと形式が異なるうえ、読んでみても内容はほとんど理解不能。ただし、PCIから情報を得る方法については、参考になる例が多い。
include/linux/fs.h	struct file_operationが定義されているなど、ファイル操作関係の重要定義が存在。
inclide/linux/sched.h kernel/sched.c	スケジューリングに関する部分。前回の周期実行などの根本に関わる部分であり、また、プロセスの休眠などを扱う。

*22　前回も出てきましたが、起動時から1秒にHZ(普通は100)のペースでカウントアップする重要変数です。

jiffiesなので、そこそこ実時間に厳密にやるには、HZを使って計算する必要があることです。関数の処理内容は割り込み同様、手短に終らせ、必要なら起こすといったところでしょうか。

●果報は寝て待て

比較的長い時間待ちが必要なハードウェアが存在するため、ドライバで一定時間眠らせることが必要な場合があります。そのためには、リスト4のようにするとjiffies単位でdだけ寝て待ちます。また、他のプロセスには迷惑をかけません[*23]。なお、schedule_timeout()は途中で別の要因で起こされたときに、残り時間を返してくれますので、

```
while(t) t=schedule_timeout(t);
```

とすれば、きっちり休ませることができます。

●PCIの情報取得

今どきの多くのハードウェアはPCIバス経由で接続されています。前回も/proc/pciなどから情報を得る方法を述べましたが、デバイスドライバとしても、ぜひとも対応しなければなりません。PCIのアクセスの方法は2.1の途中から、大幅に変わりました[*24]。それまではPCIバスの設定情報を直接読み書きするような形でしたが、現在はカーネルが持っている情報リストをもらう形です。古い形式については、PCIのハードウェア設定の詳細を説明しなければなりませんので、ここでは割愛します。これについては参考文献[3]をご覧ください。

現在はpci_find_device()という関数を使います。これはベンダID、デバイスID、および「struct pci_dev *」の変数を渡すと、pci_dev *を返してきます。最初にpci_dev *にNULLを設定して呼び出すと、該当するIDを持つデバイスがあれば返します。次にこれを引数に渡すと次の該当デバイスを返します。このように呼ぶ度に次々と返してきます。見つからない場合や、あるだけ返した場合はNULLを返します。この構造体からベースアドレスや割り込みの情報が得られます。

ただし、さらにややこしいのは、この構造体の読み方がカーネルの2.3の途中[*25]から変わっていることです。その辺りの解説を兼ねて、リスト5にサンプルを載せておきました。20行目付近のL24を定義すると2.4形式の、未定義とすると2.2形式のプログラムになります。厳密にはバージョン番号[*26]で判断すべきですが。また、pci_dev構造体からは、メモリ空間

リスト4

```
current->state = TASK_INTERRUPTIBLE;
// 2.1以降使用可
schedule_timeout(d);
// 2.0
current->timeout=jiffies+d; schedule();
```

実行例2　リスト5の実行例

```
# /sbin/insmod pcitest4_lj
./pcitest4_lj.o: init_module: Device or resource busy
# dmesg |tail
pcitest: found device(8086,1229,0) on
             Bus:0  Device:16 Function:0
pcitest: found base(0) Mem address: FEBFF000
pcitest: found base(1) I/O address: EF00
pcitest: found base(2) Mem address: FEA00000
pcitest: found IRQ 11
```

COLUMN 2

Linuxバージョンと互換性

デバイスドライバを書く上で頭を悩ませるのは、カーネルのバージョンの違いによる互換性です。安定版とされる2.0、2.2、2.4の間には互換性がない部分がありますが、系列内では互換性は保たれているため、安定版に対応するには最低3通りのソースが必要となります[*27]。しかし、開発版の2.1や2.3まで完全にサポートしようとすると非常に大変です。バージョンの最後の桁まで確認する必要があります。例えば、ドライバに要求される関数のselect()がpoll()の形に変更になったのは2.1.22から2.1.23になったときです。

対策に困るところですが、一番簡単なのは、あきらめてしまうことです。自分1人や、仲間内で使うドライバなら、使うカーネルのバージョンで動くようにしてしまえばいいでしょう。あえて古いバージョンのカーネルを使うこともないでしょうから、新しいのを使いたくなったときに、合わせて改造すればいいのです。しかし、一般公開したり、製品につけたりする場合はそうはいきません。可能な限り広くサポートしたほうが喜ばれますが、開発版にまで対応するかは利用者層によることでしょう。

なお、この文章で「カーネル 2.x.yから」という部分は、可能な限り手元のカーネルアーカイブ(tar.gzで4.3Gbytes:-))のヘッダファイル群から調査しています。（熊谷正朗）

＊23　ちなみに、jiffiesが目的の値になるまでwhileで回すというのは最悪の書き方です。schedule()をwhileに入れるとマシですが、他に動作すべきプロセスが無い場合はすぐにschedule()から戻ってきますので、やはり無駄になります。
＊24　手元のカーネルアーカイブによると、2.1.93からfind_pci_device()が定義されています。
＊25　頭が痛いことにpci_devのメンバbase_address(ソースコードを参照してください)は2.3.12で消滅、 pci_resource_startマクロは2.3.43から追加されたらしいので、この間に対応するには自分で構造体をばらす必要があるみたいです(;_;)。
＊26　バージョン番号は定数LINUX_VERSION_CODEに16進数2桁ずつで、2.x.yが2xxyyの形式で定義されています。
＊27　部分的に#ifdefで切り替える程度で済みます。

に関してはより詳細な属性情報が得られますが、ここではベースアドレスを得るにとどめています。

おわりに

今回はデバイスドライバの作り方を、ハードウェアに片寄った面から解説してみました。デバイスドライバのすべてを語ることはとても無理ですが、「何をどう使ったらいいか」の1つの案としてご参考になればと思います。

次回はいよいよ、ロボットの制御システムの構築例として、2脚・4脚ロボットのソフトウェアに迫ります。

RESOURCE

[1] Linuxでロボットを作る「第1回Linuxによるハードウェア制御の基礎」
熊谷正朗、LinuxJapan(2001年7月号)

[2] Linuxでロボット・ハード・制御
http://www.mechatronics.mech.tohoku.ac.jp/~kumagai/linux/
筆者のWebページです。

[3] LINUXデバイスドライバ
ALESSANDRO RUBINI著／山崎康宏、山崎邦子訳、オライリージャパン(1998)／ISBN4-900900-73-7

リスト5　PCIの情報取得

```
// gcc -c pcitest.c -Wall -Wstrict-prototypes -O

#define MODULE
#define __KERNEL__

#include <linux/autoconf.h>
#if defined(CONFIG_MODVERSIONS)
                          && !defined(MODVERSIONS)
# define MODVERSIONS
#endif
#ifdef MODVERSIONS
# include <linux/modversions.h>
#endif

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/pci.h>

#undef L24
//#define L24

static int vendorid=0x8086;  // 適宜
static int deviceid=0x1229;
#if LINUX_VERSION_CODE > 0x20115
MODULE_PARM(vendorid, "i");
MODULE_PARM(deviceid, "i");
#endif

int init_module(void)
{
  int i,j;
  struct pci_dev *pdev=NULL;

  for(i=0;;i++) {
    pdev=pci_find_device(vendorid,deviceid,pdev);
    if(!pdev)
      break; //    これ以上ドライバ見つからず
    printk("pcitest: found device(%04X,%04X,%d) on"
           "   Bus:%d  Device:%d Function:%d\n",
           vendorid,deviceid,i,pdev->bus->number,
                   pdev->devfn>>3,pdev->devfn&0x7);
    for(j=0;j<6;j++) {
#ifdef L24
      unsigned long start=pci_resource_start(pdev,
                                             j);
      unsigned long flags=pci_resource_flags(pdev,
                                             j);
      if(flags & IORESOURCE_IO) {
        printk(
            "pcitest: found base(%d) I/O address: "
                                "%04lX\n",j,start);
      }
      if(flags & IORESOURCE_MEM) {
        printk(
            "pcitest: found base(%d) Mem address: "
            "%08lX\n",j,start);
      }
#else
      unsigned long baseaddr=pdev->base_address[j];
      if((baseaddr&PCI_BASE_ADDRESS_SPACE)==
         PCI_BASE_ADDRESS_SPACE_IO) {
        printk(
        "pcitest: found base(%d) I/O address: ",j);
        printk("%04lX\n",
                baseaddr&PCI_BASE_ADDRESS_IO_MASK);
      } else {
        printk(
        "pcitest: found base(%d) Mem address: ",j);
        printk("%08lX\n",
               baseaddr&PCI_BASE_ADDRESS_MEM_MASK);
      }
#endif
    }
    printk("pcitest: found IRQ %d\n",pdev->irq);
  }
  if(i==0) {
    printk(
    "There is no device vendor:%04X device:%04X\n",
                                vendorid,deviceid);
  }
  return -1;  // insmod がこけるように
}

void cleanup_module(void)
{
}
```