special

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
kumagai@emura.mech.tohoku.ac.jp

はじめに

私がLinuxに初めて触れたのは1997年の夏でした。当時、ロボットの制御などを行っていた所属研究室では、制御用にはNECのPC-98x1シリーズとMS-DOSを使い、論文作成(LaTeX)やメールのやりとりにはSunOS(4.1JLE)を使っていました。使っていたワークステーションが、当時の200MHz版Pentium程度のPCにも処理速度で大幅に劣るということで、PC互換機+Linuxが導入されました。もちろん、シミュレーションなどにも使用されましたが、制御用ではなく、デスクワーク用にLinuxが導入されたわけです。

それから3年ほど経つうちに、PC+MS-DOSという組み合わせが非常につらくなってしまいました。どんどん高性能化するPCを研究に使おうにも、MS-DOSでは生かし切れなくなったのです。

このような背景のもと、Linuxを研究に使うという発想が当然出てきました。現在広く用いられているRT-Linuxも普及し始めていましたが、いくつかの解析と実験を経て、「素のLinuxでも使える」と、結論づけて今日に至っています。

この記事では、これまでためてきた、x86用*1のLinuxをハードウェア制御に使うためのノウハウを一挙にまとめたいと思います。まず、今回は、ハードウェア*2制御の基礎ということで、普通にプログラムを作る範囲でできることを解説します。それもふまえて、次回はハードウェア制御のためのデバイスドライバ製作の解説を行いたいと思います。

写真1
2脚ロボット

写真2　4脚ロボット

画面1　Linuxを用いて構築したGUI制御システム

ロボットを制御するということ

私が所属する研究室では、歩行ロボット(写真1、写真2、画

*1　ある程度は他のアーキテクチャの参考になると思いますが、異なる点があると思います。
*2　CPUやキャッシュもハードウェアには違いありませんが、ここではそれらは除きます。

Linuxでロボットを作る

面1)や自動運転自動車(これも定義上はロボット)の制御などの研究を行っています。平たく言えば、「目的を達成するにはどう情報を得て、機械をどう動かしたらいいかを考え、それを実現すること」が本業です。このためには、大きく分けて2つが要求されます。1つは、制御則(制御理論)と呼ばれる「動かし方」を考えることで、2つ目は「実体を動かすこと」です。シミュレーションのみの研究では前者のみ(と、シミュレーションの方法)で済みますが、実際にロボットを作って動かすとなると、さまざまな技術が要求されます。コンピュータに関連した重要なものは、表1に挙げたものになります。(1)と(3)のためには、専用の回路をコンピュータにつけます。例えば、センサの値を読むには、センサそのもの、その信号を適当な大きさに変換する回路、アナログ/デジタル(A/D)変換回路が一般には必要です。最終的にコンピュータ(PC、CPU)と、ロボットとの接点はPCIバス、ISAバス上のインターフェイスボードになり、その点だけを見ればEthernetカードやサウンドカードと同列に扱うことができます(ロボットが、ボード1枚に全部乗り切らなかったのだと思ってください:-))。

次に、制御則の実装です。制御を行うにはセンサの情報を読み取り、過去の履歴も含めて、何らかの演算(単純な場合は四則演算程度)をほどこして、ハードウェアに指示を出すわけです。この作業は、普通は一定の時間間隔ごとに行います。これは、ハードウェアはソフトウェアに比べて動作が遅いため、常に演算していても無駄であるという理由もありますが、速度の情報を利用するときには一定の時間間隔で処理をするのが楽であるためです(例えば、0.01秒で18度回転したら、毎秒5回転の速度)。現在の時刻が詳細に分かるなら、適当な間隔でいい場合もあるのですが、一定の間隔できっちり実行できるなら、それに越したことはありません(厳密には周期がまちまちだと、速度以外の影響も出ます)。

これらを考慮して、以下では、一般的にハードウェア(ロボット)を制御するための基礎として、ハードウェアの操作(入出力)と、一定周期での動作実行について、解説します。

Linuxによるハードウェアの操作

●ハードウェア操作方法の選択

ハードウェアと一口に言っても対象は多岐にわたります。

表1　コンピュータ関連の要素

(1)	センサなどの値をコンピュータに取り込むこと
(2)	制御則をその場で実行できる演算式(プログラム)として実装すること
(3)	算出された結果を元に、アクチュエータ(駆動装置、モータなど)に指令を出すこと

もちろん、ロボットを操作するには専用のインターフェイスを用いますが、PCそのものも汎用のインターフェイス回路の塊です。このようなハードウェアを操作するためには、2つの方法があります。

1つはLinuxのデバイスドライバを使用することです。キーボードから入力される文字を取得するとき、`getchar();`などと使いますが、途中(Xサーバなど)を無視して、一番のハード寄りの部分をみると、Linuxカーネルに組み込まれたキーボードのドライバ(`linux/drivers/char/keyboard.c`、`pc_keyb.c`)が、PS/2のインターフェイスとやり取りして、キーボードの情報を得ています。Linuxのカーネルには膨大な量のドライバが用意されており、これらを使うことでPCに標準的なハードウェアは操作できます。最近では、拡張カードでもLinux対応をうたうものが多く、制御用のインターフェイスボードを買ってくる場合でも、メーカーがLinux用のドライバを提供することが珍しくなくなりました。

2つ目の方法は、自分の作るプログラムから直接的にアクセスすることです。ドライバがない場合には、自分でなんとかする必要があります。ドライバを書くのも一案ですが、たいていの場合はそこまでする必要はなく、普通に書いたプログラムからのアクセスで足ります。ドライバが用意されている場合でも、ドライバではサポートしていない特殊な機能を使いたい場合や、速度を重視する場合には直接アクセスする必要があります。

ドライバを使う方法は、それぞれのマニュアルを参照していただき、ドライバを書くことについても次回にまわして、今回は普通のプログラムから直接操作する方法を解説します。

●ハードウェアとの入出力の仕掛け

ハードウェアと直接入出力を行うためには、その対象を知る必要があります。一般に、x86系のCPUでは、多くのハードウェアはI/Oポートという形でバス上に存在し、一部の大量のデータを扱うもの(ビデオカードのVRAMや制御レジスタなど)はメモリ空間上に存在しています。

ご存じのように、一般的なメモリは、あるアドレスに何かを書き込むと、後に同じアドレスを読んだときにその値が読める、というものです。ハードウェア上は、アドレスバス、データバス、書き込み信号、読み込み信号が存在し(図1a)、メモリは書き込み信号が来たときに指定されたアドレスに対応したデータを覚え、読み込み信号が来たときに、指定されたアドレスに蓄えられたデータをデータバスに出力してCPUがそれを得る、という動作が行われます*3。ここで、書き込まれた内容を覚えた上で、他に使えるような回路を作ります

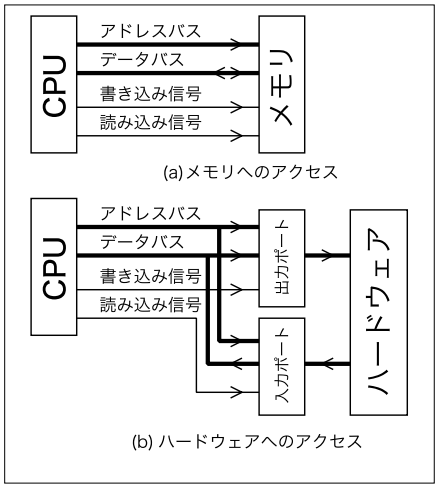


図1 メモリへの読み書き、ハードウェアへのアクセス

(図1bの出力ポート)。すると、ある特定のアドレスに1byte書き込むと、それに応じて8本の信号をオンオフすることができたりするようになります。逆に、読み込み信号が来たときに、外から来ている信号をデータバスに流すような回路(入力ポート)を作ると、情報の入力ができるわけです。この読み書き自体は、C言語ではポインタなどで行うことになります。このように、メモリ空間上に回路をつけて入出力を行う方法を「Memory-mapped I/O」と呼びます。

比較的多くのCPUが Memory-mapped I/Oを使うのに対して、x86系を含む Intel系CPUは別にI/Oポートというものを持っており、たいていのハードウェアとの入出力はI/Oポート経由で行われます。回路的には、アドレス、データバスはメモリと共通ですが、読み書き信号がメモリ用のものとは別に用意されています*4。アクセスのためには専用の命令が用意されています。

これらはアクセスの仕方が異なりますので、個別に解説します。

●I/Oポートの操作(1):保護の解除

I/Oポートの操作は、CPUの命令上は IN/OUT、LinuxのC言語開発環境は、`inb()`などの関数を用いて行います。しかし、その前に、保護機能を解除する必要があります。

Linuxはマルチユーザ、マルチタスクのOSです。多数のプロセスが実行されている状態で、あるプロセスが誤って、もしくは悪意を持って、システムの根幹にかかわるハードウェア(たとえば、IDEの制御部)にアクセスするとどうなるでしょうか。多くの場合はOSごと停止し、最悪の場合後遺症が出て、他のプロセスに迷惑がかかります。そのため、ハードウェアへのアクセスは基本的にご法度で、OSによって制限(保護)されています。そこでまず、これを解除する必要があります。

制限を解除するには、`ioperm()`、`iopl()`の2種類のシステムコールのいずれかを使用します(表2)。`ioperm()`は必要なアドレスのみ許可、`iopl()`はI/Oポートへのアクセスすべてを許可します。誤ってアクセスする危険性を考えると、`ioperm()`を使用した方が安全ですが、`ioperm()`は先頭1024アドレス(`0`〜`0x3ff`)しか制限を解除できないという制約があります(Linuxの実装上の制限)。PCIバス上のカードに割り当てられるアドレスは、たいていはこれより上になりますので、`iopl()`を使う必要があります。

さらにもう1つ制約があって、`ioperm()`、`iopl()`はroot権限がないと動作しません*5。プログラムを`root`で実行するか、もしくは実行ファイルの所有を`root`にして、「`chmod +s`」とする必要があります。

●I/Oポートの操作(2):I/Oアクセス

保護を解除したら、I/Oアクセスを行います。I/Oポート読み込み関数`inb()`、`inw()`、`inl()`と、書き込み関数`outb()`、`outw()`、`outl()`が`asm/io.h`に用意されています*6。`b`、`w`、`l`はそれぞれ、1byte、2bytes、4bytesの読み書きに対応しており、それぞれ`unsigned char`、`short`、`long`にあたります(表3)。

注意点としては、インライン関数を使うために、コンパイル時には「`gcc -O`」と最適化オプションを付ける必要があります(付けないと、ライブラリを探しに行きます。そして見つかりません)。また、`ioperm()`、`iopl()`で許可を取らずにこれ

表2 ioperm()とiopl()

関数	説明
`ioperm(unsigned long from, unsigned long num, int turn_on);`	アドレスfromからnum個(バイト)のアドレスに対するアクセス許可を設定します。`turn_on`を1にすると有効になります。複数の領域を有効にするときは、その数だけ呼んでください。
`int iopl(int level);`	プロセスのI/Oアクセス特権を`level`に変更します。具体的には`level`を3にするとアクセス許可、0〜2で不許可になります(通常は0)。

(注)man ioperm、man ioplを参照してください

*3 今時のPCはこんなに単純ではありませんが、炊飯器に入っているような小規模CPUはこんな感じです。
*4 わざわざ別に用意した開発者の考えは知りませんが、限られたメモリ空間を削りたくなかった、少量のI/Oのみが欲しいときに回路が簡単化できるなどの理由があったと思われます。
*5 一般ユーザーが実行できるとしたら巨大なセキュリティホールです:-)。
*6 `asm/io.h`内では、その場にx86系CPUのIN/OUT命令を埋め込むように定義されています。

表3　I/Oポート読み込み関数inb()、inw()、inl()と、書き込み関数outb()、outw()、outl()

関数	説明

unsigned char inb(unsigned short port);
unsigned short inw(unsigned short port);
unsigned long inl(unsigned short port);
	ポートアドレスport(0〜0xffff)から1byte、2bytes、4bytes読み取り、返します。

void outb(unsigned char value, unsigned short port);
void outw(unsigned short value, unsigned short port);
void outl(unsigned long value, unsigned short port);
	ポートアドレスportに、1byte、2bytes、4bytes出力します。MS-DOS、Windows系のコンパイラとは引数の順番が異なるので要注意。

らの関数を実行すると、SIGSEGV(Segmentation fault)が発生します。

なお、I/Oポートの一部には、連続したアクセスを行うと動作が間に合わないものが存在します。このような対象には適当なウェイトを入れるか、またはinb_p()のように、「_p」を付けます。この場合、0x80ポートを使って、ウェイトを入れますので、ioperm()を使う場合は、ここも忘れずに開く必要があります。

以上のようにしてI/Oポートのアクセスが可能です。

●I/Oポートの操作(3)：サンプル

簡単な例として、リスト1にビープ音を1秒鳴すプログラムを示します。iopl()とioperm()を含む行はいずれかをコメントアウトしてコンパイルして、rootで実行してください。PC互換機のみ対応ですが、1秒ほど音が鳴るはずです。ハードウェアの詳細な解説(記事末のRESOURCE[1]を参照)は割愛しますが、ビープの音程を決めるタイマの周波数設定と、オンオフの操作をしています。「outb(inb(...)...);」と、outb()にinb()で読んだ値を反映させていますが、これは一部のビットのみを操作するためで、|=、&=に相当します*7。

●物理メモリの操作(1)：/dev/mem

概要では、Memory-mapped I/Oの入出力操作はC言語のポインタによって行うことができると述べました。このことは、実はLinuxの普通のプログラムには当てはまりません。アドレスが仮想化されているためです。そのため、個々のプロセスから見えているアドレスは、コンピュータの実メモリ上のアドレスとは一致していません。それどころか、ハードディスク上のスワップ領域に存在している場合もあります*8。つまり、普通にポインタを使用するのみでは、物理的なアドレスに対してアクセスすることはできず、Memory-mapped I/Oを使うことはできません。それに対して、Linuxのカーネル側では1対1で対応するため、デバイスドライバ内部では直接ポインタで扱うことができます。

そこで、物理的なメモリアドレスに操作を可能とするためのものとして、デバイスドライバがLinuxに標準で用意されています。このドライバを使用するには、/dev/memにアクセスします。これは、メモリ空間を大きさ4Gbytesの1つのファイルとして見せます*9。適当なモードを指定してopen()し、lseek()によって目的のアドレスに移動した上で、read()、write()によって読み書きします。以前はアクセスの可否は/dev/memのパーミッションによって決定されていましたが、現在のバージョン(2.2系、2.4系)はrootの権限が必要です。

1byteアクセスする場合にもシステムコールを呼ばなければ

リスト1　beep.c

```
 1 /* 簡単なI/Oテスト:
 2  * gcc -O -o beep beep.c
 3  * 参考: /usr/src/linux/drivers/char/vt.c  */
 4 #include <stdio.h>
 5 #include <unistd.h>
 6 #include <sys/io.h>
 7 #include <asm/io.h>
 8
 9 int main(void)
10 {
11  unsigned int count;
12  /* 一括許可 */
13  /* if(iopl(3)) */
14  /* 部分許可 */
15  if((ioperm(0x0040, 4,1))||(ioperm(0x0061, 1,1)))
16    {
17      perror("iopl/ioperm");
18      return 1;
19    }
20
21  count=1193180/1000;       /* 1000 Hz */
22  outb(inb(0x61)|3, 0x61); /* beep on */
23  outb(0xb6, 0x43);         /* set freq */
24  outb(count & 0xff, 0x42);
25  outb((count>>8) & 0xff, 0x42);
26
27  sleep(1);
28  outb(inb(0x61)&0xfc, 0x61); /*  off */
29  return 0;
30 }
```

*7　この操作は、「以前書いたものが読める」仕様のポートのみに有効な手法です。書いたものが読めない仕様のポートにビット操作をする場合は、最後に出力した値を適当な変数に保存しておく必要があります。

*8　アクセスされた場合には、メモリ上に存在しないことをCPUが関知し、Linuxがディスクからメモリ上に読み出してきて実行が再開されます。この時、空きがないと、別プロセスのメモリが追い出されます。

*9　無論、デバイス用の特別ファイルなので普通のファイルと同列に扱うことはできませんが。

表4　mmap()

関数	説明
void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);	fdで示されるファイルのoffsetからlengthバイトだけ直接アクセスできるポインタを(可能なら)返します。protoは読み(PROT_READ)書き(PROT_WRITE)を指定し、flagsは種別指定でMAP_SHAREDを指定すればよいでしょう。startには0を指定します。

ならないため時間はかかりますが、とりあえずは物理的なメモリアドレスを読み書きできますので、Memory-mapped I/Oも操作できます。ところで、この物理的なメモリ空間のアクセスは非常に危険です。他のプロセスはもちろんのこと、Linuxのカーネルそのものも書き換えることが可能です。扱いには十分注意しましょう。

リスト2　memtest.c

```
 1 /* 簡単な物理メモリテスト:
 2  * gcc -o memtest memtest.c
 3  * ./memtest <開始アドレス> <長さ>    */
 4 #include <stdio.h>
 5 #include <stdlib.h>
 6 #include <unistd.h>
 7 #include <fcntl.h>
 8 #include <sys/types.h>
 9 #include <sys/stat.h>
10 #include <sys/mman.h>
11 #include <asm/page.h>
12
13 int main(int argc,char **argv)
14 {
15   char *mmaped;
16   int fd;
17   unsigned int st,len,poff;
18   char *dev="/dev/mem";
19
20   if(argc!=3)
21     {
22       fprintf(stderr,"memtest <start> <len>\n");
23       return 1;
24     }
25   st=strtoul(argv[1],NULL,16);
26   len=strtoul(argv[2],NULL,16);
27   poff=st%PAGE_SIZE;
28
29   fd=open(dev,O_RDONLY);
30   if(fd<0)
31     {
32       fprintf(stderr,"cannot open %s\n",dev);
33       return 1;
34     }
35   fprintf(stderr,"mmap: start %08X  len:%08X\n",
36                  st-poff,len+poff);
37   mmaped=mmap(0,len+poff,PROT_READ,MAP_SHARED,
38               fd,st-poff);
39   if(mmaped==MAP_FAILED)
40     {
41       fprintf(stderr,"cannot mmap\n");
42       return 1;
43     }
44   fwrite(mmaped+poff,1,len,stdout);
45   munmap(mmaped,len);
46   close(fd);
47   return 0;
48 }
```

・補足

/dev/memに類似するものとして、/dev/kmemがあります。これは物理的なアドレスではなく、カーネル空間でのアドレス(Linux 2.2ではオフセットがあります)を参照します。また/dev/portというものがありますが、これはすでに述べたI/Oポートを/dev/memと同じように読み書きできるものです。inb()などを使用して直接アクセスした方が圧倒的に早いため、あまり意義はありません。

●物理メモリの操作(2)：mmap

1byteアクセスするにも/dev/memをread()しなければならないというのは、速度の面で非常に不利です。それを解決するためmmap()というシステムコールが存在します(表4)。

/dev/memに対してmmap()を実行して得られたポインタは、物理アドレスオフセットを直接アクセス可能なものとなります。そのため、ハードウェアの応答速度そのものが得られます。

ただし、mmap()には癖があって、オフセットがPAGE_SIZE(x86では4Kbytes)単位でなければなりません。そのため、半端なアドレスを参照したい場合は、いったんアドレスをPAGE_SIZE単位に切り捨ててポインタを得て、それに切り捨てた分を加えて使う必要があります。

●物理メモリの操作(3)：サンプル

リスト2に物理メモリアドレスにアクセスする汎用のプログラムを示します。指定されたアドレスから、指定したバイト数を読み取り、標準出力に出力します。「od -t x1」などと組み合わせて結果を確認します。open()してread()するだけでは芸がないので、mmap()を使っています。

適当なデバイスがないと、このプログラムは試し甲斐がないので2つほど例を挙げます。まず、ビデオカードのメモリ領域は比較的分かりやすいようです。次節に述べる方法でビデオカードのアドレスを探して、画面左上の表示を変化させながら、そこを見てみましょう(ATIのMach64なチップはベタに見えました)。

次に、カーネルの情報を見てみます(実行例1)。これはjiffiesという、カーネル内部の有名な変数を10秒間隔で読んだものです。普通のx86用Linuxでは0.01秒に1だけカウント

アップする変数です。最初に/proc/ksymsでカーネル内のjiffiesの格納アドレスを探します。次にこれを読み出していますが、Linux 2.0系ではそのまま、2.2系、2.4系では0xc0000000を除く必要があります。ちなみに、表示された数値の差は0x3eaで1002になっています。

●PCIの自動割り当てアドレスを探す

最後に、目的のハードウェアのアドレスを探し出す方法を解説します。レガシーなデバイスではアドレスは固定されており、ISAバスに接続するカードも一般にはジャンパなどで固定値に設定されていました。これらは解説本やマニュアルを見るとアドレスが分かります。それに対して、PCIバス上ではBIOSなどによって自動的に割り当てられます。

PCIバス上のデバイスは、起動時に最大6個の必要なサイズのメモリ、もしくはI/Oポート領域の割り当てを要求します。これに基づいて、BIOSがそれぞれの領域の先頭アドレス(ベースアドレス)を決定していきます。ボードの抜き差しをしない限り、変わることは少ないようですが、事故を避けるため、PCIバス上のハードウェアにアクセスするプログラムを作成する場合には、このベースアドレスを確認すべきでしょう(テスト目的では決め打ちが多いですが……)。実際に操作する場合には、デバイスの仕様書に記された、各ポートのオフセットアドレスと、このベースアドレスを加えて使うことになります*10。

さて、具体的な探し方ですが、PCIバスの情報を見るというと、/proc/pciが一般的です。ただ、これは人間が見て分かりやすいのですが、プログラムで解析するのは一苦労です。そのためか、Linux 2.2では/proc/bus/pci/devicesが用意されたようです(Linux 2.4では領域の大きさが追加されていますが、ある意味、上位互換です)。

まず/proc/pciの読み方ですが、実行例2のようにアクセスするとデバイスがぞろぞろ出てきます。そこから一部抜粋してきました。これは440BXと自作のPCI拡張ボードの部分です。

最初のBus、device、functionはデバイスの存在位置を示し、ここでは必要ありません。次にデバイスの種類が出力されます。メジャーなデバイスは、名前で表示されますが、そうでないものは、Vendor id、Device idという2つの数値で識別されます*11。そこで、この項目で目的のデバイスを探し出します。最後にボードの情報を読みます。次の行はボードの設定を表示するとともに、あれば割り込み(IRQ)が出力されます。続けて、最大6個のメモリ、I/Oポートベースアドレスが出力されます。

実行例1　カーネルの情報

```
# grep jiffies /proc/ksyms
c011d3ec proc_dointvec_jiffies
c024c680 jiffies
# ./memtest 24c680 4 | od -t x4 ; sleep 10 ; \
  ./memtest 24c680 4 | od -t x4
mmap: start 0024C000  len:00000684
0000000 037aedf8
mmap: start 0024C000  len:00000684
0000000 037af1e2
```

実行例2

```
% cat /proc/pci
PCI devices found:
  Bus  0, device   0, function  0:
    Host bridge: Intel 440BX - 82443BX Host (rev 3).
      Medium devsel.  Master Capable.  Latency=32.
      Prefetchable 32 bit memory at 0xe0000000 [0xe0000008].
                                :
  Bus  0, device  11, function  0:
    Bridge: Unknown vendor Unknown device (rev 0).
      Vendor id=136c. Device id=a026.
      Medium devsel.  Fast back-to-back capable.  IRQ 11.
      I/O at 0xa400 [0xa401].
      I/O at 0xa800 [0xa801].
                        ⋮
```

ついで/proc/bus/pci/devicesの読み方です。こちらはどちらかというとプログラムから利用することを前提にしているようで、数字の羅列です(実行例3)。

実行例2と対応するものを抜き出してあります。これはLinux 2.2の出力です。誌面の都合上、3行に分割しましたが、1つの項目は1行で記載されています。これらの数字は、1行目がBus-device-function、Vendor-Device、割り込みで、2行目、3行目はベースアドレスです。ベースアドレスが合計7個あるのは、入出力とは別に用意されているROM領域のアドレスを含むためです。さて、このベースアドレスではメモリかI/Oポートか区別がつきません。この区別には最下位ビットをみます。これが「1」の場合I/Oポートで、「0」の場合はメモリです。さらに、I/Oポートの場合の下位2bit、メモリの場合の下位4bitは領域の属性を表すものですので、無視する必要があります。Linux 2.4の場合には、この後に7個の数が続き

*10　逆に、そのような仕様を入手できないボードは、ドライバが用意されていない限り、Linuxでは使いにくいものと言えます。情報開示に積極的なメーカーもあるので、そのようなメーカーの製品をお薦めします。

*11　PCIバス上のデバイスは、種類別にVendor ID(メーカー)、Device ID(種類)を持ちます。これによって、自動認識、ドライバの自動組み込みなどが行われるわけです。名前で表示されるデバイスは、単にIDから変換しているだけです。IntelのVendor IDが8086なのは、狙っているんでしょう:-)。

実行例3

```
% cat /proc/bus/pci/devices
0000    80867190        0
        e0000008        00000000        00000000
        00000000        00000000        00000000        00000000
                :
0058    136ca026        b
        0000a401        0000a801        00000000
        00000000        00000000        00000000        00000000
                :
```

ます。これは各領域の大きさを示すものですが、無視してかまわないでしょう。

以上の方法によって、PCIバス上の自動割り当てされたデバイスのアドレスも確認できます。書籍などで「デュアルブートならWindowsのプロパティを確認して」という記述を見かけることがありますが、PCIに関しては多少の知識があれば、その必要はありません。また、不明なVendor IDはPCI SIG([2])にて調べることが可能です。

●まとめ

以上が、Linuxで普通にプログラムを製作し、ハードウェアにアクセスするための方法です。機械の制御をしたいという場合には必須となるほか、概念として覚えておくと、何かちょっといじってみたいというときにも役に立つと思います。

Linuxによる周期的実行

●概要

最初に述べましたように、ロボットなどを制御しようと思った場合、ハードウェアの操作ができることと同時に、一定の周期で処理が実行できることが求められます。より広い意味では、「任意の時刻に実行できる」となります。

MS-DOSのように、1つのプログラムを起動したら、それがCPUを占有して動作するような場合は、さまざまな手段を用いてそれを達成することができます*12。しかしながら、普通のマルチユーザー・マルチタスクOSは、すべてのユーザー、プロセスを原則として平等に扱うように設計されています。そのため、CPUを占有することは許されず*13、実行時刻を厳密に指定したりはできません。

マルチタスクでもこのような処理を行いたい、という要求に対する答えの1つがリアルタイムOSです。リアルタイムOSは処理を終えるべき時刻や優先順位を考慮して、各タスクの実行を行います。また、それらの処理を行いやすいような機能を豊富に持っています。リアルタイムOSには多種ありますが、Linuxをベースにした有名なものとしては RT-Linux([3])、ART-Linux([4])が挙げられます。現在、PC互換機(および互換ワンボードPCベースのロボット制御では、RT-Linuxが広く使われ、ART-Linuxも広まりつつあります。これらはLinuxをもとに改造し、Linuxとしての性質を残したままリアルタイムOSの機能を追加したもので、多くのLinux用のソフトウェアはそのまま動作します。

ところが、いずれも、導入、運用にはある程度の技量を必要とします。雑誌の付録をいれておしまいとはいきません。また、RT-Linuxでは処理のプログラムはカーネルに組み込む形で実行するため、少しでも間違うとカーネルごと停止しかねません。そこで、リアルタイムOSの機能を使わず、「ほぼ」一定周期で実行できれば良いという条件で、普通のLinuxを使って周期実行を行う方法を解説します。

この方法は、私自身がロボットの制御システムを構築する際に確認して、それ以来使い続けている方法です。カーネルのソースを眺めていて*14、発見したものですので、それなりにしっかりした方法と考えています。最初は2.0系でしたが、2.2系、最新の2.4系でも基本は変わっておらず、動作を確認しています*15。

●まず、やってみよう

細かな原理を説明する前に、まずはやってみましょう。リスト3にテスト用のプログラムを用意しました。

このプログラムは三角関数を適当に使ったダミー処理と、起動時の引数で指定したミリ秒の休眠(usleep())を1000回繰り返して、毎回の実行時刻を測定するものです。時刻の計測方法は2種類あります。もし、Pentium以降のCPUを使用する場合は、TickPerMSecを、「[周波数(MHz)]e3」に変更して、-DUSE_RDTSCをつけてコンパイルすることをお薦めします。

さて、実験してみましょう。まずは「./lcycle 15」として実行が終わるまで、静かに待ちましょう(やたらとマウスを動かしたり、他のプログラムを起動しない)。すると、cycle.xyというファイルができます。これを適当なグラフ化ソフト(gnuplotなど)でグラフにしてみます。その例を図2に示します。

*12 他に仕事がないなら、指定時刻が来るまで無駄ループで待つのもありですし、他に仕事があるなら、タイマ割り込みを設定して処理すれば良いでしょう。
*13 占有することはできます。
*14 デバイスドライバを書こうとしたら、カーネルくらいしか参考書がなかったのです。
*15 Linuxをそのまま使うので、新しいカーネルが出たら、すぐに試せます:-)。

リスト3　lcycle.c

```
 1 /* gcc -o lcycle lcycle.c -lm */
 2 /* gcc -o lcycle lcycle.c -DUSE_RDTSC -lm */
 3 #include <stdio.h>
 4 #include <sys/time.h>
 5 #include <unistd.h>
 6 #include <stdlib.h>
 7 #include <math.h>
 8
 9 #define TestCount 1000
10
11 #ifdef USE_RDTSC           /* RDTSC命令を使用 */
12 /* １ミリ秒あたりの tick数 */
13 #define TickPerMSec 200e3  /* CPU周波数より計算*/
14 /* 現在時間取得: 単位 tick */
15 unsigned long long GetTick(void)
16 {
17   unsigned int h,l;
18   /* read Pentium cycle counter */
19   __asm__(".byte 0x0f,0x31" : "=a"(l),"=d"(h));
20   return ((unsigned long long int)h<<32)|l;
21 }
22 #else    /* USE_RDTSC */
23 #define TickPerMSec 1e3
24 unsigned long long GetTick(void)
25 {
26   struct timeval tv;
27   gettimeofday(&tv,NULL);
28   return (unsigned long long)tv.tv_sec*1000000
29                              +tv.tv_usec;
30 }
31 #endif  /* USE_RDTSC */
32
33 int main(int argc,char **argv)
34 {
35   int cycle=15,i,j;
36   double a=1;
37   FILE *fp;
38   unsigned long long ticks[TestCount];
39
40   if(argc>1) cycle=atoi(argv[1]);
41
42   for(i=0;i<TestCount+3;i++)  /* 一定周期ループ */
43     {
44       /* 計測 */
45       if(i>2)                /* 安定してから計測 */
46         ticks[i-3]=GetTick();
47       else
48         GetTick();           /* 空読み */
49
50       /* ダミー処理 */
51       for(j=0;j<100;j++) a=sin(cos(a));
52
53       /* 休止 */
54       usleep(cycle*1000);    /* msec->usec */
55     }
56
57   fp=fopen("cycle.xy","w");
58   for(i=0;i<TestCount-1;i++)
59     { /* number, interval, time */
60       fprintf(fp,"%d\t%lf\t%lf\n",i,
61               (ticks[i+1]-ticks[i])/TickPerMSec,
62               (ticks[i]-ticks[0])/TickPerMSec);
63     }
```

リスト3　lcycle.c

```
64   fclose(fp);
65   return 0;
66 }
```

・参考

gettimeofdayはマイクロ秒単位で現在時刻を得るシステムコールです(精度はおおむねマイクロ秒)。システムコールですので実行に多少時間がかかります。RDTSCはPentium以降のCPUで定義された命令で、「リセット時からCPUクロックを数えている64ビットのタイムスタンプカウンタの値」を読み取ります。そのため、クロック周波数で割ると、時刻が得られます。公称周波数と、実際の周波数は微妙に異なりますが、誤差の範囲です(クロックアップしていないこと前提)。ここではバイナリで直接埋めています。

何回か試しながら結果を確認すると、以下のようなことが分かります(ただし、Alpha用Linuxは異なるはずです)。

- **15ミリ秒を指定したのに、なぜか20ミリ秒である(Linuxのバージョンによっては 30ミリ秒)。**
- **基本的には、ほぼ20ミリ秒ちょうどになる。**
- **時々30ミリ秒になったり、不定に大きく遅れることがある(図2には現れていませんが)。**

さらに、起動引数の15を変えながら試すと、

- **周期は20ミリ秒より短くならないらしい。**
- **周期は10ミリ秒単位らしい。**

ということもわかります。ついでに、他に無限ループ的なプログラムを動かしているのはかまわないが、入力待ちをしているものに入力を与えたりすると遅れる、などの現象も見られます。

以上のことより、

- **普通のLinuxでも、20ミリ秒以上10ミリ秒単位でよければ、比較的高精度な周期が得られる。**
- **ただし、時々周期がずれ、他のプロセスの影響も受ける。**

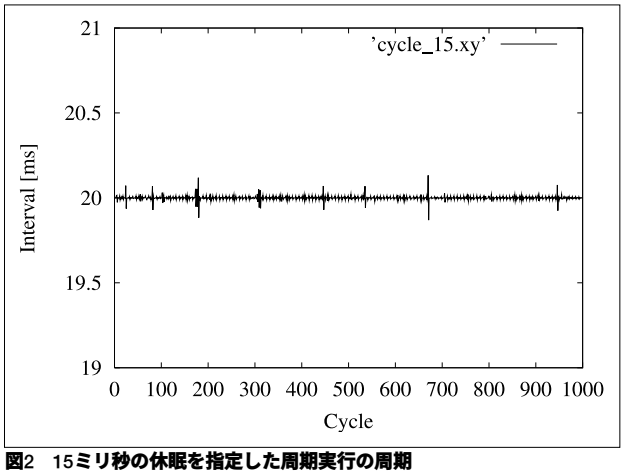


図2　15ミリ秒の休眠を指定した周期実行の周期

という結論が得られます。本当に適当な用途であれば、これくらいでも使えないことはないのですが、ロボットを制御するとなると多少問題があります。以下では、それを改善するスパイスを加えていきます。

●実行性能の向上(1)：分解能向上

まず、最低で20ミリ秒という制限が、ロボットの制御をする上では一般に問題になります。ロボットやモータの制御では数ミリ秒の周期で行うことが多いためです。そこで、より短周期で実行できるようにします。

さて、この問題の解決には、この20/10ミリ秒という単位が何に起因しているかを探る必要があります。後に原理を解説しますが、結論からいうと、Linuxカーネルが時間関係の処理を行うために使用しているタイマ割り込みの周期に依存しています。この周期が10ミリ秒です*16。そのため、この周期を1ミリ秒にしてしまいます。その結果、もともと1秒に100回だった処理が1000回になってしまう分だけ全体的なパフォーマンスは落ちることになりますが、今時のCPUには些細な負荷です。

具体的な変更にはカーネルの再構築が必要になります。そのためには、Linuxのカーネルソースを拾ってくるか、各ディストリビューションのカーネルソースパッケージをインストールして、再構築する必要があります。具体的な手順は各解説におまかせするとして、まず、何も手を加えない状態で再構築し、正常に起動できるようにしましょう。もちろん、念のために、現在使っているカーネルを生かしたまま、LILOで切り替えられるようにするなどしてください。

カーネルの変更点といっても、たいしたことはありません。`linux/include/asm/param.h`の

```
#define HZ 100
```

に「0」を1つ加えて、1000にするだけです(これは割り込みの周波数を示します)。これが終ったら、モジュールも含めてカーネルを再構築(`make clean`、`make dep`、`make bzImage`など)して、インストールの上、再起動してください。便宜上この改造をしたカーネルを1k(いちきろ)-Linuxと呼んでます。

起動後、特にこれといった違いはみられないと思います。ただし、環境によると`ps`や`top`を実行すると、CPU利用が合計で1000%になっていたり、起動時に「`Unknown HZ value! (..) Assume 100.`」と警告が大量に発せられて*17、分かることがあります:-)。また、メモリの読み書きで使ったプログラムがあるなら、`jiffies`を確認してみてください。10秒で10000くらい増えるはずです。

さて、実際に期待通りの効果が出たか、`./lcycle 5`で試してみましょう。図3に結果例を示します。今度は、5ミリ秒を指定して6ミリ秒になりました。あとは`usleep()`する時間を増減すると、2ミリ秒以上、1ミリ秒単位で周期が作れます*18。

さらに短周期にすることも原理的には可能ですが、無難に使えるのはこのあたりと考え、これ以上にはしていません。関心がある方は限界に挑戦してみてもよいでしょう。

●実行性能の向上(2)：安定性向上

これまで「静かにしておく」ことを条件にしていました。1つには、Linuxカーネルの処理がプロセスに優先するためであり、他のプログラムを起動するなどして大きなディスクアクセスを発生させると、その分処理が遅れるためです。これはスワップのアクセスも含むため、頻繁にスワップが使われるようなメモリが少ないマシンでは、この方法は難しくなります。この原因については、普通のLinuxを使う限り避けようがありませんので、このようなイベントを起こさないように(新しくプログラムを起動しない、いっそのことスワップを切る(`swapoff`など)する必要があります。

2つ目の理由として、CPUを他のプロセスに取られる、という現象を避ける目的があります。詳細については次に述べますが、本来Linuxはすべてのプロセスに平等にCPUを割り当てることを前提に設計されています。そのため、マウスの移動に伴ってktermがアクティブになるなど、別のプロセスが

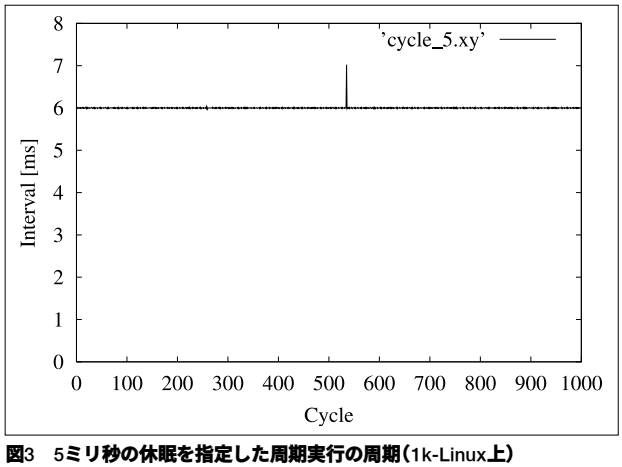


図3　5ミリ秒の休眠を指定した周期実行の周期(1k-Linux上)

*16 Linux 2.2ではAlpha用のみ1/1024秒、Linux 2.4では MIPS用などでさまざまな値が定義されています。
*17 手元ではRed Hat 6系の、Dualマシンで出ることを確認しています。ライブラリの(簡単な)修正が必要でした。
*18 なお、このくらいの周期にしますと、本気で使うためには、処理量の変動を考慮して`usleep()`すべき時間を指定しなければならない場合もありますが、現在時刻が分かるので、あとは算数の問題です。

CPUによる処理を必要としたとき、そちらの処理が先に行われる可能性があります。これについては気を付けても避けることは容易ではありませんが、簡単に対策を施すことが可能です。

具体的には、優先度を示すnice値を小さくします*19。プログラムをそのままで試すには、rootで「nice --20 ./lcycle 5」とniceコマンドを使用します。引数は-20で最優先扱いになります(rootでなければ負の数は指定できません)。

プログラムを起動する度にniceを用いるのは面倒ですので、プログラムの中から直接システムコールを呼ぶ方法が、現実には良いでしょう。これには表5の2つがありますが、setpriority()を使った方がよいでしょう。

周期実行を行っている途中に、カーネルを再コンパイルしたりするとひどいことにはなりますが、普通に使う分には、優先度を上げるだけで耐久性が劇的に上がります。

●原理

ここまでは実践的な内容に絞って解説しましたが、実行結果にはいくつかの特徴が見られます。

- usleep()で指定した休眠時間を切り上げた時間で周期的に実行される。
- 実行周期は、カーネル定数HZの整数倍になる。
- 優先度を多少上げると、他のプロセスにCPUを取られにくくなる。

これらはすべてLinuxカーネルの実装によるものです。逆に、Linuxカーネルがそのように実装されているため、このくらい単純な方法で、高精度な周期実行が可能となっているわけです。

この方法に関連したLinuxの実装を説明します。

・プロセスの状態

プログラムの実行単位であるプロセスは、Linuxで定義された複数の状態のうちのどれかになっています。その中でも重要なのは、現在CPUで実行中の「実行状態」、実行されることを希望している「実行可能状態」、実行を希望せずに休止している「休眠状態」の3つです。プロセスは演算などを実行している間は実行可能状態ですが、sleep()、usleep()を使うことで明示的に、またselect()によるデバイス待ちや、read()など処理時のブロックで自動的に休眠状態に入ります。指定時間の経過や、デバイスの準備完了に伴い、実行可能状態に戻ります。さらに、実行状態にはユーザー空間で実行している場合と、カーネル空間で実行している場合(システムコール処理中など)があります。

・スケジューリング

実行可能状態にある複数のプロセスから、CPUの数だけ選び、実際に実行できるようにする動作をスケジューリングと呼びます。実行可能状態のプロセスがCPUより多い場合は、各プロセスの持つ「持ち時間数値」の大小を比較し、多いものにCPUを与えます。このスケジューリングは、システムコールの終了直前、(ユーザー空間で実行中に)ハードウェアからの割り込みが発生し処理が終了する直前*20に行われます。また、デバイス待ちに時間がかかるようなデバイスドライバの内部で、CPUを手放すために意図的に呼び出される場合もあります。なお、スケジューリングは一見すると、「いつ戻ってくるかわからないschedule()という何もしない関数を呼び出す」という形に見えます。

・タイマ割り込み

Linuxは時間にからむ処理を行うため、周期的にタイマ割り込みを行っています。標準では1秒あたり100回です。ここでは、カーネル内部で持っている時間情報の処理、jiffies++、時間指定休眠プロセスを起こすかの判断などを行います。さらに、現在CPUで実行されているプロセスの持ち時間を1減らします。

・持ち時間の再計算

持ち時間は正の値のみをとります。そのため、しばらくすると、すべての実行可能プロセスの持ち時間は0になってしまいます。このとき、休眠中のプロセスを含む全プロセスの持ち時間cは「c'=c/2+p」と更新されます。ここでpは優先度を表

表5　niceコマンドに相当するシステムコール

関数	説明
int nice(int inc);　(unistd.h)	niceと同じ効果を持ち、nice値をincだけ、「増減」させます。
int setpriority(int which, int who, int prio);　(sys/resource.h)	種類which対象whoの優先度(nice値)をprioに設定します。範囲は-20～20で-20が最優先です。呼び出し元のプロセスの優先順位を操作するのであれば、setpriority(PRIO_PROCESS, 0, prio);とすれば良いでしょう。やはり、rootでなければ、負の値を指定できません。

*19 優先度を上げたいのに値を小さくする、というのは一見すると変な気もしますが、CPUの優先権を主張する=「nice」ではなくなる、と解釈するのでしょう、多分。
*20 システムコール実行中の場合は、割り込み処理が終了し、待たせておいたシステムコールが終了するとき。

す数値でnice値から計算される値です(およそ「20-nice値」を正数倍)。休眠が多いプロセスでは、cをタイマ割り込み時に減らされることも少なく、最終的に、c=2pに近付きます。

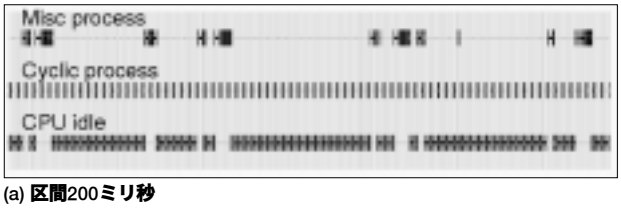
以上の機構で、この方法の原理が説明できます。

まず、usleep()で休眠させたプロセスが起きる(実行可能状態に遷移する)のはタイマ割り込みのところです。さらに、割り込みであるため、その処理終了時にはスケジューリングが行われます。休眠させていたプロセスが十分な持ち時間を持っていれば、直前まで実行されていたプロセスに代わってCPUを得て、usleep()の直後から実行が再開されます。特に、この持ち時間勝負に勝つことが重要であるため、nice値を小さくすることに効果があります。また、「タイマ割り込みの直後」が実行再開時刻となりますので、精度が確保されるわけです(ばらつきは、割り込み処理量の変動程度)。

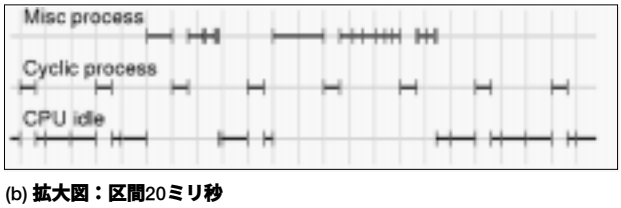
逆に欠点もはっきり見えてきます。プログラムの起動など、システムコールが頻繁に発行され、ディスクアクセスなども頻発している状態では、タイマ割り込みがあってもすぐにはプロセスの切り替えが行われず、周期精度が落ちます。また、Linuxでは休眠している(入力待ち、リクエスト待ち)プロセスが大部分ですから、それらすべてが2pの持ち時間を持っているようなものです。そのため、nice値を操作しない限り、そのようなプロセスを刺激すると周期の乱れが発生します。加えて、これまで述べてこなかったもう1つの欠点が見つかります。例えば、8ミリ秒処理を行って2ミリ秒だけ休むようなプロセスは、持ち時間はどんどん減っているため、他のプロセスに負ける可能性が高くなります。現実には、精度良く周期的に行いたい処理は、たいてい短時間で済みますので、大きな問題になることは少ないのですが、注意は必要です。

最後に、実際に周期実行をした場合の、プロセスの動作の様子を図4に示します。図は、独自開発の監視ツールを使って、プロセスの実行の様子を可視化したものです。下段は実行可能状態のプロセスがなく、CPUが休んだ状態、中段が周期実行のプロセス(3ミリ秒周期)、上段はその他のプロセス(XFree86など)を示します。横軸は時間の経過を示し、「H」の部分がCPUによる実行を示します。また、縦線はタイマ割り込みのタイミングを表します。図からも、他に実行中のプロセスがあるなしにかかわらず、タイマ割り込みの直後から周期的に実行されていることが確認できます。

図4　3ミリ秒の休眠を指定した周期実行の様子(1k-Linux上)



(a) 区間200ミリ秒



(b) 拡大図：区間20ミリ秒

●まとめ

本当に完全な時刻性能を求めたいのであれば、OSの処理や、他のハードウェア割り込みに影響されないRT-LinuxやART-Linuxを導入する必要がありますが、ときどき周期がずれても影響が出ないような用途には、普通のLinuxで十分足ります。素のLinuxを使うがゆえ、これまでLinux用に開発されてきた膨大な資源を苦もなく使うことができるため、便利でもあります。周期がたまにずれることを心理的(?)に受け入れられる方にはお勧めです。

おわりに

今回はロボットや拡張カードなどを制御するための基礎として、ハードウェア操作の方法と、周期的にプログラムを実行するための手法について解説しました。次回は、デバイスドライバの作り方を扱いたいと思います。もっとも、普通のドライバの作り方はすでにいくつも解説が出ているので、「ハードウェアをいじるためのライブラリ」としてのドライバに注力する予定です。

RESOURCE

[1] トランジスタ技術 1999年9月号　CQ出版
「特集 PCのハードウェア徹底研究」(菜野 雅彦)

[2] PCI SIG
http://www.pcisig.com/

[3] LINUX リアルタイム計測/制御 開発ガイドブック
船木陸議／羅正華著、秀和システム, 1999

[4] インターフェース　1999年11月号　CQ出版社
「ART-Linux誕生の経緯と使い方」(石綿 陽一)、「 リアルタイム制御を実現するART-Linuxの設計と実装」(石綿 陽一)、「ART-Linuxによるリアルタイム処理への適用と応用」(柴田智広)