本稿は Linux Japan 誌 2001 年 9 月号に掲載された記事に補筆修正したものです.

GPIB インターフェース

研究室にある計測器をコンピュータ制御しようと思うと，計測器に標準装備しているインターフェースを使わざるを得ません．これが今なお RS-232C か GPIB（General Purpose Interface Bus）[1]W[3]なのです．RS-232C はコンピュータ側にも標準装備されており，追加投資する必要がないという利点がありますが，低速であり，何よりも 1 台しか接続できないことが計測システムを組み上げる場合の障害になります．一方の GPIB は，データ線が 8bit パラレルなので原理的には RS-232C よりはかなり高速であることに加え，15 台まで接続可能であることから，小さな計測システムを組む場合には良く使われます．ただし，DMA 転送などを使わない限り，最終的には，思ったよりもずっと低速にしかデータ収集ができないことに注意しなければなりません．100 回/秒を越えるようなデータ収集を考えるなら，A/D ボードを直接制御する方法を採用すべきです．

GPIB は，ケーブルやがっちりとしたコネクター形状まで規格が定められており，標準インターフェースとして認知されていますから，国内外を問わず計測ボードメーカーの製品ラインアップには必ずあるのですが，例によって Linux 用のドライバーが提供されないている場合はまれです．フリーなものでは，National Instruments（GPIB のコントローラチップ TMS9914 を製造していることでも有名）の互換ボード用のドライバーを開発していた The Linux Lab Project[2]W[3] 有名ですが，カーネルのバージョンアップに対応できずに，開発はほぼ停止しています．NI のある機種については，NI で開発されたのドライバーが公開されています [3]W[3].

日本でも Linux 用のドライバーが公開されているメーカーはあるのですが [4]W[3]，ボード自体が高かったりして，今一つ購入する気になれませんでした．ところが，近頃（株）インターフェースの低価格（29,800 円）の GPIB ボード PCI-4301 [5]W[3] 用に Linux のドライバー（モジュールバイナリ）とライブラリがネットワークを通じて入手可能となりました（登録が必要です）．ソースが公開されていないので，カーネルのバージョンの制限を受けますが，プログラムを組む上では支障ありませんので，実際に使ってみた感想を述べたいと思います．

インストール

インストールは，pgp4301.tgz を展開してできる interface/gpg4301/readme.htm に従って行えなば間違いないでしょう．ライブラリ libpgp4301.so と適切なドライバモジュール cp4301.o をコピーすることが主な仕事です（これも，make するだけなんですけど）．ライブラリは *.so となっていますが，実は static なライブラリのようです．ドライバーはカーネルバージョンに合わせたモジュールとして提供されていて，筆者がダウンロードしたものは 2.2.16 までの対応でした．2.2.16 で動作確認されたディストリビューションは Turbo Linux Server 6.1 のみでしたが，筆者愛用の Plamo2.1 でも問題なく動作しました．lsmod で cp4301 モジュールを組み込んだ後の初期設定のユーティリティ（cgpibconf, xgpibconf）も用意されていますから，ほとんど何も考えずに使える状態になります．この辺りはさすが商品だけあって，良くできていて本当に楽で助かります．

libpgp4301 の使い方

さて GPIB のドライバーは簡単に組み込めますが，計測器と接続してデータ収集をするには，もちろんプログラミングが必要です．これもまた，ソース例が豊富についてきていますから，手持ちがその機種があれば（HP や Advanced のデジタルマルチメータ）以下のようにして，直にアプリケーションが作成できます．

```
gcc -o advr6451 advr6451.c -lpgp4301
```

このドライバーは一応製品購入者のみがダウンロードできるものなので，具体的な名前を記すことは憚られます．抽象的に，どのような機能の関数を用いるかで流れを紹介します．

ボードの初期化　コントロラボードは 16 枚まで使うことができますので，どのコントローラボードか（既定は 0 番）を指定して初期化します．インストール時に作成される，デバイスファイルはメジャー番号 254 で

```
/dev/gpibex0    /dev/gpibex1
```

という名前です．

インターフェースのクリア　GPIB バスに接続されているデバイスを全てクリアします．これは，GPIB の規定にある制御線 IFC を使って行われるので，確実です．

デバイスのクリア　接続されているデバイスを初期化します．

リモートイネーブル　接続されているデバイスをリモート操作できるように設定します．GPIB で接続してあっても，デバイスをローカルに使用したい場合もありますから，リモートが解除できないと不便です．逆に明示的にリモートを設定する必要があるのです．

デバイスの設定コマンド送出　デバイスのアドレス（コントローラで使用した番号を除いて 0 から 31 番まで設定できます．予めデバイス毎にローカルに設定して置かなければなりません）を指定して初期設定コマンドを送出します．例えば，デジタルマルチメータならば，電圧・電流・抵抗などの種類，サンプリング間隔など設定すべき事柄があり，その測定器独自のコマンド文字列を送出します．なお，しばしば通信エラーの原因となるので，**改行コードが LF+CR （一般的にはこれが工場出荷時の既定），LF，CR のいずれであるかを決定しておかなければなりません**．測定器によっては，デイップスイッチでなどで設定するものもあり，ソフトウェア制御ができないものもありますから要注意です．また，トリガーも良く考える必要があります．基本的にはコントローラ（パソコン側）から測定開始のトリガーを送り，測定器が測定を終えるとそのデータをコントローラあるいは他の記録用のデバイスに送るといった手順を踏むようします．しかし，測定器が勝手に測定を行いデータを送出し続けるといった方法も可能です．

データの取得　測定器（トーカ）が送出したデータ文字列を受けてバッファから取り出し，処理を実行します．一般に測定のループはここだけになります．

終了　次の利用時に障害がでないよう，バッファの掃き出しやデバイスの開放など，終了時に行うべき事柄をきちんとしておきます．

抽象的に記述すると判りにくいですが，定型処理ですのでアルゴリズムを間違える要素はほとんどありません．デバイスのアドレスや改行コードの不一致などのパラメータを正確に把握しておきましょう．

プロットツールとの連携

以上 GPIB 経由でデータを収集するのは定型ルーチンとなるので簡単です．ところで得られたデータはもちろんファイルに保存するのでしょうが，同時にグラフにして表示させたいですね．データ収集のプログラムが簡単なので，グラフ表示部分を拡張したくなりますが，ちょっと待った，標準出力に流すだけにしましょう．そして，標準入力からデータを読み込んでグラフにするツールと組み合わせてみましょう．

pipe

まずはシェルのパイプ機能を使ってみます．0 から 1 の間の一様乱数を一定時間間隔で永遠に発生させるプログラム dummy を予め作っておきます．別に難しいことはありませんが，9 行の **setlinebuf()** は大変重要です．一般に，バッファは送られてきたデータを溜め込み，あるサイズになって初めてデータを掃き出します（まさにバッファリング）．ところが，即時にデータをグラフ化するというような場合，途中で溜め込まれては困るのです．したがって，プログラマの意図通りにデータを直に掃き出させたい場合にはラインバッファを設定します．ラインバッファは改行コードを受けるとその時点でデータを全て掃き出します．

List 1　data.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <sys/times.h>
 4
 5 main(int argc, char **argv)
 6 {
 7    int i=0;
 8
 9    setlinebuf(stdout);
10    srand48(3);
11    while (1){
12      usleep(atoi(argv[1]));
13      printf("%d %f\n", ++i, drand48());
14    }
15 }
```

このソースをコンパイルして

```
gcc -o data data.c
```

できあがった data で疑似的にデータを発生させます．それらをプロットツールに渡せばいいのですが，筆者愛用の gnuplot や xgraph はデータを全て読み込んでからでないとグラフを描きません．

そこで GNU の plotutils[6] W^3 に含まれる graph の活躍です．graph は x,y の範囲を指定するとリアルタ

イムにプロットします．--max-line-length を 1 にすれば 1 点ずつ描かせることができます（図 1）．

```
dummy 10000 | graph -T X -x 0 1000 -y -5 5
--max-line-length 1
-h 0.7 -w 0.7 -X 'Times' -Y 'Signal'
```

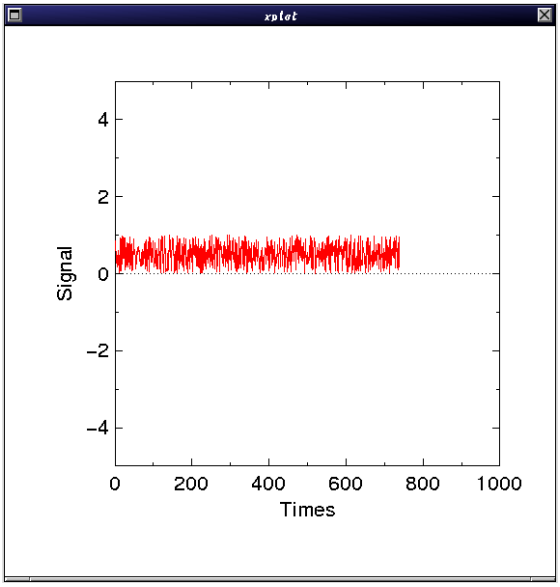

図 1　graph によるリアルタイムプロット

popen()

パイプ機能ををプログラム自体に持てせることもできます．あるプログラムから子プロセスとして外部コマンドを起動して，パイプを通じてそのコマンドに命令を送るという場合，popen() という便利な関数があって非常に簡単に実現できます．筆者が試作した gnuplot の GUI フロントエンド xgplot から関連部分を紹介します（図 2）．gnuplot は大変優れたプロットツールで３次元のグラフも描けるのですが，現行バージョンの 3.7.1 では視点を変えて表示する場合には set view コマンドを打ち込まなければなりません（開発バージョン 3.8c からは，マウスで操作することが可能になったとこの連載でも紹介しました）．これはかなり苦痛なので，ボタンによる操作ができるように，X でフロントエンドを組んでみたわけです．基本的にはファイル構造体 gnuplot のハンドルを popen() に起動する子プロセスの実行名を与えて取得するだけです．以降，子プロセスに fprintf(gnuplot,...) で命令を流し込むことができるというものです．ここでは，gnuplot へのバッファリングを fflush で命令を流し込む都度掃き出させています．

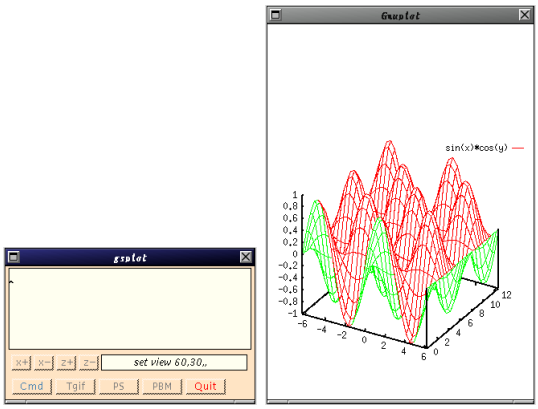

図 2　popen() による gnuplot への GUI フロントエンド gsplot

List 2　gsplot.c より

```
FILE *gnuplot;
...

void Quit(Widget w, XtPointer client, XtPointer call)
{
  pclose(gnuplot);
  exit(0);
}

void Rotate(Widget w, XtPointer client, XtPointer call)
{
   int rflag;
   rflag = (int)client;

   switch (rflag) {
      case RXP:  rx += 5; break;
      case RXM:  rx -= 5; break;
      case RZP:  rz += 5; break;
      case RZM:  rz -= 5; break;
      default: break;
   }

   fprintf(gnuplot, "set view %d,%d, \n replot\n ", rx, rz);
   fflush(gnuplot);
   Dispviewparam(viewparam,NULL,NULL);
}

main (int argc, char **argv)
{
  int i;

  gnuplot = popen("gnuplot -geometry 331x468","w");
  fprintf(gnuplot, "%s \n", Init_cmd);
  fflush(gnuplot);
  ...
}
```

双方向パイプ

popen() で形成されるパイプは，読み込みあるいは書き出しのいずれか一つしか選択できません．シェルのパイプもいわば一方通行ですから，シェルの作法に馴染んでいる人はあまり不便とは感じないかもしれません．しかし，双方向にデータが流すことができるなら便利であろうことは容易に想像できるでしょう．例えば，Tcl/Tk などで GUI を作成するような場合，複雑な数値計算は時間がかかるし，何より Tcl/Tk では

書きにくいです．元来 Tcl/Tk は 他のプログラムのモジュールとして使われることを目標に設計された言語ですから，もちろん C から呼び出すことができるのですが，そこまでは覚えたくはないというのが筆者の正直な気持ちです．Tcl/Tk のスクリプトを実行してパイプでデータをやりとりしたくなります．

そこで，dup2 を使って双方向パイプを形成する方法を紹介します．これは Unix のプロセス間通信の例題として良く取り上げられます．筆者も中途半端に理解しているだけなので，dparent.c の dopen() は例として書かれたものそのままです．readpipe と writepipe の両方を同時に取得してそこに対して読み書きを実行する仕組みです．

```
$ gcc -o dparent  dparent.c
$ gcc -o dchild  dchild.c
$ ./dparent ./dchild

Parent: send 2.000000
Child: get 2.000000 return 4.000000
Parent: get  4.000000

Parent: send 2.100000
Child: get 2.100000 return 4.410000
Parent: get  4.410000
...
```

双方向のパイプをわざわざ使った TkPlot の実行画面例を示します (図 3)．今のところ，子プロセス Tcl/Tk 側の TkPlot.tcl から親の TkPlot にデータを戻してはいないので，popen() だけでもいいのですが，将来の拡張に備えてということです．

```
./data 100000 | TkPlot -
```

と実行してみてください．

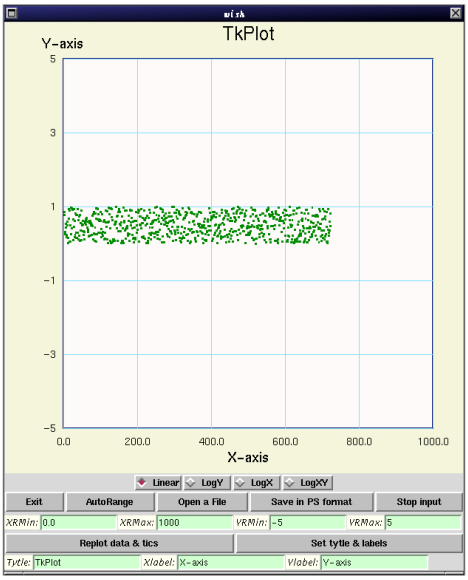

図 3　双方向パイプを内臓したプロットツール TkPlot

List 3　dparent.c

```
 1 /*
 2  * 子プロセスとして cmd を実行して双方向の
 3  * パイプを形成し通信します.
 4  *
 5  * 使い方:
 6  * dopen cmd
 7  *
 8  */
 9
10 #include <stdio.h>
11 #include <unistd.h>
12 #include <stdlib.h>
13 #include <sys/types.h>
14 #include <sys/time.h>
15
16 int dopen(char *cmd, FILE **readpipe,
                                FILE **writepipe)
17 {
18  int childpid, pipe1[2], pipe2[2];
19
20  if ((pipe(pipe1) < 0) || (pipe(pipe2) < 0 )) {
21        perror("pipe"); exit(-1);
22  }
23
24  if ((childpid = vfork()) < 0 ) {
25        perror("fork"); exit(-1);
26  } else if (childpid > 0) {   /* Parent */
27    close(pipe1[0]); close(pipe2[1]);
28    *readpipe = fdopen(pipe2[0], "r");
29    *writepipe = fdopen(pipe1[1], "w");
30    setlinebuf(*writepipe);
31    return childpid;
32  } else {  /* Child */
33    close(pipe1[1]); close(pipe2[0]);
34    dup2(pipe1[0],0);
35    dup2(pipe2[1],1);
36    close(pipe1[0]); close(pipe2[1]);
37
38    if(execlp(cmd, cmd, NULL)<0) perror("execlp");
39  }
40 }
41
42
43 int main(int argc, char **argv)
44 {
45   FILE *data, *read_from, *write_to;
46   char readbuf[80];
47   double x=2.0, y;
48   int dopenpid;
49
50   dopenpid=dopen(argv[1], &read_from, &write_to);
51
52   setlinebuf(stdout);
53   setlinebuf(stderr);
54   setlinebuf(write_to);
55
56   while(1){
57     fprintf(write_to, "%f\n", x);
58     fprintf(stderr, "\nParent: send %f\n", x);
59     fscanf(read_from, "%lf", &y);
60     fprintf(stderr, "Parent: get  %f\n", y);
61     x += 0.1;
62   }
63 }
```

List 4 dchild.c

```
01 #include <stdio.h>
02 #include <unistd.h>
03 #include <stdlib.h>
04 #include <math.h>
05
06 int main()
07 {
08   char inbuf[128];
09   double x, y;
10
11   setlinebuf(stdout);
12   setlinebuf(stderr);
13
14   while(1){
15     fscanf(stdin, "%s", &inbuf);
16     x = atof(inbuf);
17     y = x*x;
18     fprintf(stdout, "%f\n", y);
19     fprintf(stderr,
               "Child: get %f return %f\n", x, y);
20    }
21 }
```

参考文献

[1] GPIB の概要が比較的良くまとまっているサイト.
http://www.ines.de/gpibinfo.htm

[2] 下記 URL はリンク切れです.
http://obelix.chemie.fu-berlin.de/pool
/software/busses/
今となっては，次のミラーのみが頼りです.
http://www.tux.org/pub/sites
/ftp.llp.fu-berlin.de/LINUX-LAB/IEEE488/

[3] National Instruments の Linux 情報ページ
http://www.ni.com/linux/

[4] モガミ無線の Linux 情報ページ
http://www.mogami-wire.co.jp/intex/
index.html

[5] Interface の Linux 情報ページ
http://www.interface.co.jp/catalog
/soft/linux_info.asp

[6] GNU の Plotutils
http://www.gnu.org/software/plotutils/
plotutils.html