GUI の無い環境での画像表示

フレームバッファーの利用方法

フレームバッファーとは

UNIX では全ての入出力装置（デバイス）は特殊なファイルとして実装されています。フレームバッファー framebuffer とは画像入出力のデバイスです。画像の出力が行え、現在表示中の画面のスクリーンショットが撮れます。

一版的な GUI 環境では画像の表示機能は充実していますのでアプリケーション開発者はフレームバッファーを意識する必要がほとんどありません。画面への入出力はフレームバッファーを介さない構成も可能で、最近はこちらが主流のようです。システムによってはフレームバッファーがデバイスとして実装されていない場合もあります。

簡素な構成を指向する組込装置ではフレームバッファーはよく利用されます。CD-ROM １枚にプログラムとデータを収容して単独で動くようにしたい場合などは、システム資源を多く消費する GUI は省きたいものですが、画像出力の方法がわからず仕方無くお使いの方もおられると思います。情報技術全般で進行する自動化によって見えににくなってしまっている部分の一つです。この文書はそこを明らかにします。

フレームバッファーは Linux カーネルの機能です。Mac OS と BSD UNIX では普通使えないと思います。ラズベリーパイでは画面への出力は必ずフレームバッファーを介して行います。

簡単な例

パソコンで GNU/Linux をお使いの場合、最近では X Window 上の GUI 環境が一般的です。まずキャラクターモードに移行します。

Ctrl + Alt + F5

ログイン画面が見えるはずです。ここで一般ユーザーとしてログインします。ちなみに元の GUI に戻るには Ctrl + Alt + F1 です。

フレームバッファーデバイスがあるかどうか、確認します。

```
ls -l /dev/fb0
```

ここで「そのようなファイルはありません」と言われたらカーネルを再構築するか、断念するより他ありません。たいがいの GNU/Linux ディストリビューションでは利用可能になっているはずです。

/dev/fb0 は一般ユーザーに読み書きが許可されていないので、このままでは操作できません。

この問題の解決策とは 3 つあります：

（1）全ての作業を特権ユーザーとして行う。

（2）読み書きの都度特権ユーザーになる。

（3）一時的にフレームバッファーのパーミッションを変更しておく。

一番安全なのは（2）で読み書きを最小化することです。次に（3）の方法が安全だと思います。何か間違いがあっても、被害は画面が乱れる程度で済むはずです：

```
chmod 666 /dev/fb0
```

現在の画面の状態をそのままコピーするには次のようにします：

```
cp /dev/fb0 snapshot.fb
```

格納先のファイル名は何でもかまいません。

フレームバッファーに書き込むには次のようにします：

```
cp snapshot.fb /dev/fb0
```

出力デバイス名を間違えるとシステムを破壊する場合があります。くれぐれもご注意下さい。

画面がどう変化したかわかりにくい場合は間に画面クリア（Ctrl l *コントロール　エル* か clear, reset）や出力表示の多い命令（ls –color $HOME など）を入れてみて下さい。

実験が済んだら次の手順でログアウトして GUI 端末に復帰します：

```
chmod 660 /dev/fb0
logout
Ctrl + Alt + F1
```

GUI に復帰してから xterm など文字入力端末で/dev/fb0 を操作したらどうなるでしょう。デバイスは存在するものの、X Window が他の経路で画面を制御しているため利用できないと場合が多いようです。ラズベリーパイでは操作が可能と思われます。

フレームバッファーのデータ形式

さてここで作成されたフレームバッファーのコピー snapshot.fb とはどのようなファイルでしょうか。画像の生データでヘッダー等を持ちません。ファイルの型式を調べる file 命令は単なるデータと報告するはずです:

```
file snapshot.fb
snapshot.fb: data
```

またこのファイルを他のコンピュータに移して表示することはできるでしょうか。一般的にはできません。画面出力は様々な方式（モード）があり、これが一致していないと正しく表示されません。ヘッダーが無いので方式が合致しているかどうかの判定もできません。

画素は主走査方向が横で左から右、副走査方向が上から下の配列となっています。左上隅から始まり、右上隅に行き、左に戻って 1 段下がり、右下隅が最後となります。

画素データには赤緑青の三要素の分ける TrueColor 方式が一般的です。その他にパレットを使う PseudoColor や白黒データがあるようです。この文書では TrueColor だけを扱います。

フレームバッファーのデータ形式を探る

dmesg 命令

フレームバッファーについての情報がブートメッセージに表示され、随時 dmesg で見る事ができます。ただし、システム環境によっては十分な情報が表示されません。この方法が使える環境は限られています。

dmesg はたいがい管理者特権が必要ですので sudo や su を利用します。

```
su -c 'dmesg | fgrep fb'     #  管理者パスワードを聞かれます
```

条件が整っていれば縦横の寸法と画素あたりのビット数、画素情報の形式詳細が報告されます。

```
size=8:8:8:8 shift=0:16:8:0
```

コロンで区切られている 4 つの数字は左から予備領域、赤、緑、青で size が割り当てビット数、shift が位置を左シフトの大きさで表示したものです。この例では 32 ビットは次のように使われています：

```
xxxxxxxxRRRRRRRRGGGGGGGGBBBBBBBB
```

xxxxxxxx の部分は未定義です。アルファ（透過）チャネルの利用が想定されているようですが現状では内容が無視されるようです。

デバイスファイルは CPU 次第でバイト配列が逆転するので注意して下さい。リトル・エンディアンのインテル、ARM の CPU なら上記はバイト単位で入れ替わります：

```
BBBBBBBB GGGGGGGG RRRRRRRR xxxxxxxx
```

32 ビットが 8:8:8:8 に区分されているのは各色 1 バイトで分かりやすいと言えます。16 ビットでは 3 色を 2 バイトに詰め込むので２つのバイトにまたがる色が発生します。これにバイト順序の入れ替えが加わるとややこしくなります。

/sys/class/graphics/fb0

システム情報を記す仮想ファイルシステム/sys にフレームバッファーの情報があります。

画素あたりのビット数と縦横サイズの情報が得られます。dmesg で報告される内部のビット配列の情報は無いようです。

fbset 命令

フレームバッファーの方式を調べるには fbset という命令があります。この命令が一番多く情報を得られて確実です。

一般的な GNU/Linux ディストリビューションでは標準装備のプログラムではないと思われます。利用するには apt-get や yum 等でパッケージをインストールします。ソースコードから構築する方法もあります。小さなプログラムなのですぐ構築できます。

fbset には現在のモードの報告とモードの変更の 2 つの機能があります。オプションを指定せずただ fbset と発した場合は現在のモードの報告をします。以下は出力例です：

```
mode "1024x768"
    geometry 1024 768 1024 768 32
    timings 0 0 0 0 0 0 0
    accel true
    rgba 8/16,8/8,8/0,0/0
endmode
```

rgba の欄が dmesg の size,shift に相当します。分母が size、分子が shift です。順番は dmesg と違っていて予備領域が最後になります。

geometry の最後の数値は 1 画素当たりのビット数です。この例では 32 ビットです

geometry の欄に同じ縦横の寸法が 2 度報告されています。普通はこのようになりますが、まれに違う数値が報告されます。異なる場合は前が可視領域のサイズ、後が仮想画面のサイズとなります。実際の画面の 2 倍の領域を確保しておいて切り替える使い方があるようです。一致していない場合は次節の方法を試して見て下さい。

画素を数えて直接測定

以上の手段で情報が得られない場合、または数値が信用できない場合は画素を数えて測定します。

まず縦と横、それに 1 画素あたりのバイト数を掛け算してそれがフレームバッファの容量と一致するか確かめます。たいがい後者に少し余分があるようです。縦方向、横方向どちらに余分があるか推定するには factor 命令による素因数分解が有効です。

次に画面をクリアしてから短い縦線を描き、スナップショットを撮ります：

```
clear; echo “|”; cp /dev/fb0 snapshot.fb
```

これを 16 進数でダンプします。

```
od -An -v -t x1 -w4 -v snapshot.fb | uniq -c
```

```
-An:        アドレスを省略
-v:         複を省略せず表示
-t x1:      １バイトごとに１６進数で表示
-w4:        の１６進数を１行に４つ表示
uniq -c:      重複を数える
```

規則性が見られるはずです。この数値を元に横幅を求めます。

画面を単色で塗り潰す

以上の情報を元に、フレームバッファーに単純なデータを書き込んで画面を一色にするプログラムを書いてみます。

以下のソースコードはチェックはしておりますが、デバイスへの直接の書き込みは危険を伴うことをご了承の上、十分ご確認の上実行して下さい。

```
/* fbfill.c */
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>

#define WIDTH  1024
#define HEIGHT  768

typedef uint32_t pixel;
#define RED_SHIFT    16
#define GREEN_SHIFT   8
#define BLUE_SHIFT    0
#define RED_MASK   0xFF  /* 5 bits: 0x1F 6 bits: 0x3F */
#define GREEN_MASK 0xFF
#define BLUE_MASK  0xFF

main() {

  int i;
  pixel p;

  p = BLUE_MASK << BLUE_SHIFT;          /* blue */

  for (i = 0; i < WIDTH * HEIGHT; ++i)
     fwrite(&p, sizeof(pixel), 1, stdout);

  sleep (1);
}
```

#define の各値は dmesg や fbset の出力等に従い設定します。

1 画素 16 ビットの場合は 16 ビット固定長整数を使います。

```
typedef uint16_t pixel;
```

色を青、緑、黒にしたい場合は自明でしょう。白は次のようにします：

```
p =  RED_MASK << RED_SHIFT |
     GREEN_MASK << GREEN_SHIFT |
     BLUE_MASK << BLUE_SHIFT;
```

出力先はフレームバッファーデバイスと決まっているので次の様にすることもできます：

```
FILE *fp;
...
fp=fopen("/dev/fb0","w");
...
fclose(fp);
```

上で出力先を標準出力としておいたのは、フレームバッファーに書き込む前に試行確認を行うためです。こういう小さな確認を行う習慣、検査手法を考える能力は大切です。

塗りつぶし終了後、画面に次のプロンプトが出力されるまで間をおくために sleep を入れています。

コンパイルと実行

```
gcc fbfill.c -o fbfill
./fbfill | od -t x4    # test
./fbfill > /dev/fb0
```

出力デバイス名を間違えるとシステムを破壊する場合があります。くれぐれもご注意下さい。

塗りつぶしの応用

うまく実行できたら他の色を表示、さらには縦横の帯やタイルを色分けして表示するようプログラムを変えてみると良いでしょう。

フレームバッファーの情報を一度退避エリアに保存して、プログラム終了時（sleep の後）に元の状態に復帰するというのも試すとよいでしょう。この構成は実用的です。

フレームバッファーを読み取って JPEG 形式で保存

フレームバッファーの内容をファイルに保存するにはどうすべきでしょう。そのまま保存する方法もありますが、圧縮されていないので非効率です。より大きな問題はヘッダー情報が無いことです。可搬性がありません。一般に広く使われている画像方式で保存したいものです。しかしここに障壁があります。フレームバッファーの内容は生データです。生データから JPEG や PNG を作成するソフトはありません。

ここで役に立つのが Netpbm の画像方式です。Netpbm は歴史が古く、1980 年代に登場した画像形式（PPM，PGM，PBM)とそれを扱うプログラムの集合です。

フレームバッファーの生データと PPM 方式の違いは以下の通りです：

（1）PPM にはヘッダーがある。

（2）赤緑青の順番が違う。 PPM は赤緑青の順。

（3）生データには未使用領域がある。

PPM のヘッダーの一例：

```
P6
1024 768
255
```

P6 が PPM の識別子です。2 行目が横と縦の大きさ、3 行目が精度です。1 色に 8 ビットを使う場合は可能な値は 0〜255 ということで最大値の 255 で精度を表示します。書式についての詳細は Wikipedia の Netpbm 画像方式の記事をご覧下さい。

これだけわかればフレームバッファーの生データを PPM 方式に変換するプログラムは書けると思います。是非御自身で挑戦してみて下さい。参考のため以下に一例を示します：

```
/* fbtoppm.c */
#include <stdio.h>
#include <stdint.h>

#define WIDTH  1024
#define HEIGHT  768

#define RED_SHIFT    16
#define GREEN_SHIFT   8
#define BLUE_SHIFT    0
#define RED_MASK   0xFF
#define GREEN_MASK 0xFF
#define BLUE_MASK  0xFF
#define MAXVAL 255

typedef uint32_t pixel;

main() {

  int i;
  pixel p;
  FILE * fp;

  /* header */
  printf("P6\n%d %d\n%d\n", WIDTH, HEIGHT, MAXVAL);

  /* raster */
  for (i = 0; i < WIDTH * HEIGHT; ++i) {
     fread(&p, sizeof(pixel), 1, stdin);
     fputc( (p >> RED_SHIFT)   & RED_MASK,   stdout);
     fputc( (p >> GREEN_SHIFT) & GREEN_MASK, stdout);
     fputc( (p >> BLUE_SHIFT)  & BLUE_MASK,  stdout);
  }
}
```

#define の各値、typedef の型はお使いのシステムに合わせ適宜設定して下さい。

コンパイルと実行

```
gcc fbtoppm.c -o fbtoppm
./fbtoppm < /dev/fb0 | head -n3  # test
./fbtoppm < /dev/fb0 | od -c -t x1 | head  # test

./fbtoppm < /dev/fb0 > snapshot.ppm
```

画像表示ソフトでこの snapshot.ppm を表示して確認します。もし画像が乱れていたら一番疑われるのが横幅の不一致です。画素を数えて直接測定する必要があります。色がおかしかったら shift の誤りが疑われます。

AWK によるアスキー形式 PPM の生成

PPM にはバイナリ形式とアスキー形式があります。普通は効率の良いバイナリー形式が使われますが、アスキー形式は文字で見やすく扱いやすいという特長があります。簡易スクリプト言語 AWK を用いれば１画素４バイトのフレームバッファーのデータは１０進数ダンプから容易に変換できます。

```
od -An -v -t u1 -w4 /dev/fb0 | \
awk 'BEGIN {print "P3"; print "1024 768"; print "255"}
     {print $3,$2,$1}' > snapshot.ppm
```

識別子はバイナリ形式の P6 に対して P3 です。縦横は適宜入れて下さい。

バイナリ形式の PPM には対応してもアスキー形式の PPM には対応していないソフトがあるので注意が必要です。

UNIX ではまず AWK でスクリプトを書いて動作を確認してから c で本格的なプログラムを書くという作業手順が用いられることがあります。これはその一例です。

PPM から JPEG に変換

PPM は一般人には知られていませんが、カラー画像の処理を学ぶなら最初に覚えるべきとされる、基本的なフォーマットです。GIMP 等画像を扱う多くの応用ソフトが PPM に対応しています。PPM を JPEG に変換する方法は多数ありますが、コマンドラインの命令には libjpeg 付属の cjpeg、Netpbm の pnmtojpeg 等があります。

```
./fbtoppm < /dev/fb0 | pnmtojpeg > snapshot.jpg
```

Netpbm には画像の大きさや明さを変える、回転、ぼかし、余白の追加、切り取りなどを行うコマンドラインで動く工具が一通り揃っています。これらのプログラムは UNIX パイプラインで利用します。

例えば余白を追加して横幅を 2000 画素にするには次のようにします：

```
./fbtoppm < /dev/fb0 | pnmpad -width=2000 | \
pnmtojpeg > snapshot.jpg
```

一般の画像ファイルをフレームバッファーを使い表示する

上記のフレームバッファー生データから PPM 形式への変換例を参考にすれば一般の画像形式（GIF，PNG，JPEG など）からフレームバッファー用のデータを作る方法がわかるでしょう。直接変換プログラムを書いたら大変です。ここでも PPM 形式を介在させます。

一般の画像にはさまざまな縦横の寸法の物がありますが、フレームバッファーはサイズが固定です。特に横幅が合っていないと正しく表示されません。変換プロセスのどこかで対応する必要があります。自作ｃプログラムに調整機能を入れるのも一つの方法ですが、既存のソフトウエアを利用する方法も検討してみて下さい。Netpbm で拡大縮小を行うプログラムは複数ありますが pnmscale、pamscale が汎用的です。切り取りは pamcut、余白の追加は上でも紹介した pnmpad です。いずれも長年の実績がある単純なプログラムですので安心してお使い頂けると思います。

結語

UNIX 系 OS は世界で広く使われていて通信網を支えています。その設計思想に全体を小くまとまっていて管理しやすい要素に分割する、という方針があります。何でもこなせる万能のプログラムはこの思想に反する物です。開発をする時はこれを心がけてなるべく小さいプログラムを目指すべきです。

本稿が皆様の UNIX 学習の一助になるものでしたら幸いです。

２０１９年１１月　漆畑晶

参考文献

Framebuffer について

Framebuffer HOWTO　(英語)

https://www.tldp.org/HOWTO/Framebuffer-HOWTO/index.html

Framebuffer HOWTO　(日本語)

https://linuxjf.osdn.jp/JFdocs/Framebuffer-HOWTO.html

英語版は 2010 年、日本語訳は 2000 年の英語版を元にしています。新しい方の英語版でも Linux カーネル v.2.2 について書かれています。かなり古い情報で詳細は現在は変わっている可能性があります。基礎知識と考えて利用して下さい。

Netpbm について

Wikipedia Netpbm の記事（日本語）

https://ja.wikipedia.org/wiki/Netpbm

Netpbm の画像フォーマット

https://ja.wikipedia.org/wiki/PNM_(画像フォーマット)

Netpbm 公式サイト（日本語の紹介）

http://netpbm.sourceforge.net/index-japanese.html

UNIX のパイプライン処理について

ソフトウェアの道具箱

https://linuxjm.osdn.jp/info/GNU_coreutils/coreutils-ja_210.html#Opening-the-software-toolbox

漸進的情報処理

https://www.ospn.jp/osc2019-spring/pdf/OSC2019_TokyoSpring_Netpbm.pdf

UNIX、GNU、Linux の開発史、設計思想について

論語とコンピュータ　（OSC 福岡２０１９の講義内容）

https://www.ospn.jp/osc2018-nagoya/pdf/OSC2018_Nagoya_netpbm%202.pdf

Linux From Scratch　OS を全てソースコードから構築する方法

https://www.ospn.jp/odc2019/pdf/ODC2019_Netpbm.pdf