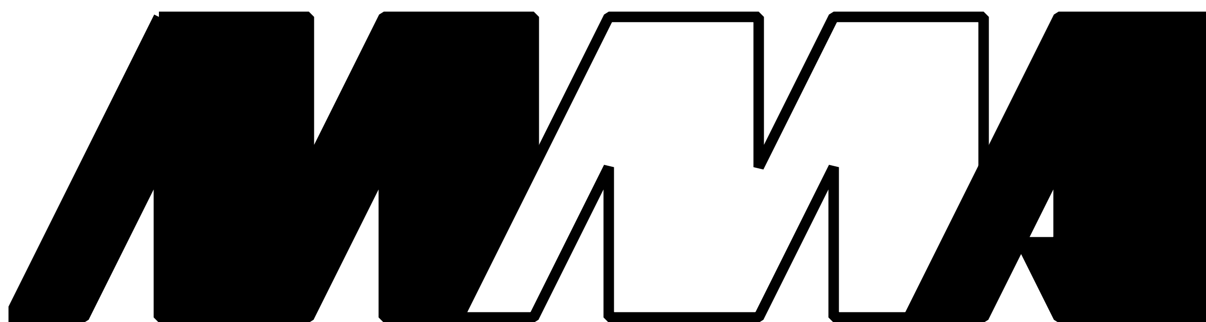
百萬石 2007年 春号



目次

第 I 部　シェルスクリプトによるログインログの取得 1

1 loging system **の実装例** 1

2 source file 1

2.1 makedir.sh . . . 1

2.2 recw.sh . . . 2

2.3 makegraph.sh . . . 3

2.4 archive.sh . . . 4

2.5 table.cron . . . 5

3 **その他の運用例** 5

4 **おわりに** 5

第 II 部　500 **円** Web **カメラを** NetBSD **上から無理やり使う** 6

1 **はじめに** 6

2 **実は解析の手間もなく。。** 6

3 BSD ugen **による** USB **デバイスの操作** 6

3.1 ugen によるコントロール転送 . . . 6

3.2 ugen によるアイソクロナス転送 . . . 6

4 PAC207 **固有の事情** 7

4.1 レジスタと初期化 . . . 7

4.2 流れてくるデータ . . . 7

4.3 レートコントロールによる自動圧縮の回避 . . . 7

4.4 イメージセンサの色配置 . . . 7

4.5 バルク転送はおすすめできない . . . 8

5 **おわりに** 8

第 III 部　FPGA による俺コンピュータ 9

1 **FPGA とは何ぞや** 9

2 **おうちでできる LSI 設計** 9
2.1 環境の整備 . . . 9
2.2 HDL 記述 . . . 10

3 **アーキテクチャのデザイン** 10
3.1 独自アーキテクチャの賛否両論 . . . 10
3.2 独自アーキテクチャの考え方 . . . 11

4 **どう実現するか** 12
4.1 論理回路でどう実現されるのか . . . 12
4.2 モニタの映る仕組み . . . 13
4.3 何はともあれ分割する . . . 13

5 **おわりに** 14

第 IV 部　FreeBSD and CompactFlash Devices 15

1 **背景** 15
1.1 フラッシュメモリ . . . 15

2 **ブートデバイスとしてのフラッシュメモリデバイス** 16

3 **CF-IDE 変換アダプタ** 16
3.1 CompactFlash における DMA . . . 16
3.2 CF-IDE 変換アダプタの問題点 . . . 17
3.3 DMARQ,DMACK の結線 . . . 17

4 **FreeBSD on CompactFlash** 18
4.1 レイアウトの方針 . . . 18
4.2 rc スクリプトによるメモリディスクの作成 . . . 18

5 **ファイルシステムのブロックサイズ** 19

6 **実行結果** 19
6.1 速度 . . . 19

7 **まとめ** 20

部長挨拶

新入生の皆様、ご入学おめでとうございます。
本年度の部長に就任しました mura です。

まずは、 MMA について簡単にご紹介させていただきます。
MMA は大学公認サークルで、そのまま「えむえむえー」と読みます。

現在の MMA では、ワークステーションなどの大きなコンピュータから
PIC や H8 のような小さなコンピュータに関連する活動を中心に行っていますが、
特にこれをやらなければならないといったことはなく、
自分達のやりたいことをやっています。

夏休みには花火合宿なども行われています。

新歓期間は多くの新入生が訪れ賑やかになっています。
新入生に限らず、コンピュータに興味があるひとは、
サークル会館 2F 奥の部室を覗いてみてください。

第 I 部

シェルスクリプトによるログインログの取得

(著者) moechar
moechar@mma.club.uec.ac.jp

概要

どうも，本誌編集の moechar です．昨年度まで本学情報通信工学科の計算機室 IED が運用していた Solaris が OS の計算機のログインユーザについて, リモートログインしているユーザと IED の Sun Ray 端末からログインしているユーザを分離してログインログを取得する方法を考案しました．本年度から IED の計算機システムは昨年度までのものからリプレースされているので本稿の内容では現行のシステムでそのまま実行できない点に注意してください.

1 loging system の実装例

以下の様に動作を分けて実装しました.

(1) log file の格納場所の設置 (`makedir.sh`)
(2) 各種 log file を作成 (`recw.sh`)
(3) グラフなどに plot して可視化 (`makegraph.sh`)
(4) 月が変わったら log file を 1 日単位で圧縮 (`archive.sh`)
(5) 一定の周期で (1)〜(4) を繰り返す (`table.cron`)

2 source file

2.1 makedir.sh

```
1  #!/usr/bin/bash
2  year=`env LANG=C date | awk '{print $6}'`  # date 結果から年だけを表示
3  month=`env LANG=C date | awk '{print $2}'` # date 結果から月だけを表示
4  day=`env LANG=C date | awk '{print $3}'`   # date 結果から日だけを表示
5
6  case $month in                             # 月を数字に変換
7  Jan)
8          month=1
9          ;;
10 Feb)
11         month=2
12         ;;
13 Mar)
14         month=3
15         ;;
16 Apr)
17         month=4
18         ;;
19 May)
20         month=5
21         ;;
22 Jun)
23         month=6
24         ;;
25 Jul)
26         month=7
27         ;;
28 Aug)
29         month=8
30         ;;
31 Sep)
32         month=9
33         ;;
34 Oct)
35         month=10
36         ;;
37 Nov)
38         month=11
39         ;;
40 Dec)
41         month=12
42         ;;
```

```
43  esac
44
45  timelabel=`echo $year-$month-$day`         # 日付形式の変数を作成
46  mkdir ~/var/log/$timelabel                 # ディレクトリを作成
```

やってる事は単純ですね．このスクリプトを bash で書いたことに特に意味はありません．6〜43 行目の月変換のところは date コマンドのオプション次第では不要です．また，あらかじめホームディレクトリ以下に ~/var/log なるディレクトリを作成しておかないと以降すべてのスクリプトは正しく機能しないので注意してください．

2.2 recw.sh

```
1   #!/usr/bin/bash
2   year=`env LANG=C date | awk '{print $6}'`
3   month=`env LANG=C date | awk '{print $2}'`
4   day=`env LANG=C date | awk '{print $3}'`
5
6   [中略]
7
8   prefix=~/var/log/$year-$month-$day            # 日付ディレクトリ用の prefix
9   prefix2=~/local/bin                           # ローカル用の prefix
10  timelabel=`env LANG=C date`                   # 英語形式の date 結果を格納
11
12  cd $prefix                                    # 日付ディレクトリに移動
13
14  rsh ied0 env LANG=C w -hl > tmp-tenpu         # ied0 のログイン状況を取得してファイルに出力
15  rsh ied1 env LANG=C w -hl >> tmp-tenpu        # ied1 のログイン状況も追記
16  sort tmp-tenpu > tenpu                        # ソートして出力
17
18  grep pts tenpu > pts                          # pts シグナルを出す端末を抽出
19  hitonokazu=`wc -l pts | awk '{print $1}'`     # pts シグナルを出す端末数を取得
20  grep dtlocal tenpu > dt                       # dtlocal シグナルを出す端末を抽出
21  dtninzu=`wc -l dt | awk '{print $1}'`         # dtlocal シグナルを出す端末数を取得
22
23  awk '{print $1}' pts > tmp-pts                #  ログイン ID のみ抽出
24  awk '{print $1}' dt > tmp-dt
25
26  uniq tmp-pts uni-tmp-pts                      # 総ログイン ID を抽出
27  uniq tmp-dt uni-tmp-dt                        # ローカルログイン ID を抽出
28
29  diff uni-tmp-pts uni-tmp-dt | grep "<" | awk '{print $2}' > uni-tmp-nondt
30  # remote ログイン ID を抽出
31
32  Upper1=`wc -l uni-tmp-pts | awk '{print $1}'` # 総ログイン数を取得
33
34  if [ $Upper1 -ne 0 ];                         # 総ログイン数が 1 以上なら
35          then
36          logins=`head -n $Upper1 uni-tmp-pts`  # ログイン ID を 1 行に展開して
37          hoge=($logins)                        # 配列に格納
38
39          cnt=0                                 # 制御変数
40          DecUpper=`expr $Upper1 - 1`
41          while [ $cnt -le $DecUpper ];         # 総ログイン数分反復
42                  do
43                  grep "${hoge[$cnt]}" pts >> "pts_${timelabel} ${Upper1}"
44                  cnt=`expr $cnt + 1`
45          done                                  # ログイン ID に該当する行のみ抽出して追記．
46          else
47                  touch "pts_${timelabel} ${Upper1}"
48  fi
49
50  Upper2=`wc -l uni-tmp-dt | awk '{print $1}'`  # ローカルログイン数を取得
51
52  if [ $Upper2 -ne 0 ];                         # ローカルログインについて上記と同様
53          then
54          logins=`head -n $Upper2 uni-tmp-dt`
55          hoge=($logins)
56
57          cnt=0
58          DecUpper=`expr $Upper2 - 1`
59          while [ $cnt -le $DecUpper ];
60                  do
61                  grep "${hoge[$cnt]}" dt >> "dt_${timelabel} ${Upper2}"
62                  cnt=`expr $cnt + 1`
63          done
64          else
65                  touch "dt_${timelabel} ${Upper2}"
66  fi
67
68  Upper3=`wc -l uni-tmp-nondt | awk '{print $1}'` # リモートログイン数を取得
```

```
69
70   if [ $Upper3 -ne 0 ];                                # リモートログインについて上記と同様
71           then
72           logins=`head -n $Upper3 uni-tmp-nondt`
73           hoge=($logins)
74
75           cnt=0
76           DecUpper=`expr $Upper3 - 1`
77           while [ $cnt -le $DecUpper ];
78                   do
79                   grep "${hoge[$cnt]}" pts >> "remote_${timelabel} ${Upper3}"
80                   cnt=`expr $cnt + 1`
81           done
82           else
83                   touch "remote_${timelabel} ${Upper3}"
84   fi
85
86   hhmmss=`env LANG=C date | awk '{print $4}'`    # 時刻を取得
87   hh=`echo $hhmmss | awk -F: '{print $1}'`       # 時間を抽出
88   mm=`echo $hhmmss | awk -F: '{print $2}'`       # 分を抽出
89   ss=`echo $hhmmss | awk -F: '{print $3}'`       # 秒を抽出
90
91   hmm=`$prefix2/comprate 60 $mm`                 # 秒を時間単位に換算
92   hss=`$prefix2/comprate 3600 $ss`               # 分を時間単位に換算
93   time=`echo "scale=6; $hh + $hmm + $hss" | bc`  # 時間単位で合計
94   echo "$time $Upper1 $Upper2 $Upper3" >> $prefix/time-ninzu.dat
95
96   rm tenpu tmp-tenpu dt pts tmp-pts tmp-dt uni-tmp-nondt uni-tmp-pts uni-tmp-dt
```

月変換部の `case` 文は以後同様に [**中略**] にて省略します．bash の配列変数機能を用いました．変数の識別子のつけ方が稚拙ですがご容赦ください．`while` 文の箇所は C, Java, Pascal でいうところの `for` 文の処理をしています．

14〜15 行目では `rsh` によりホストを指定してパスワードなしにログインして `w` の結果を得ています．そして 16 行目により ID について昇順にソートします．

26〜27 行目では `uniq` によって重複行を除去していますがなぜするかというと，ユーザによっては複数のホストにログインしたり，多数のターミナルを起動してシェルが上げてしまうため `pts` や `dtlocal` シグナルが重複することがあるからです．

IED でログインするとき，Sun Ray 端末からログインすると `pts` と `dtlocal` のシグナルを出すシェルを，リモートからログインすると `pts` シグナルのみ出すシェルを起動します．リモートのログイン数を把握するためには `pts` から `dtlocal` を切り離す必要があるため 66 行目のややこしい処理をかましています．

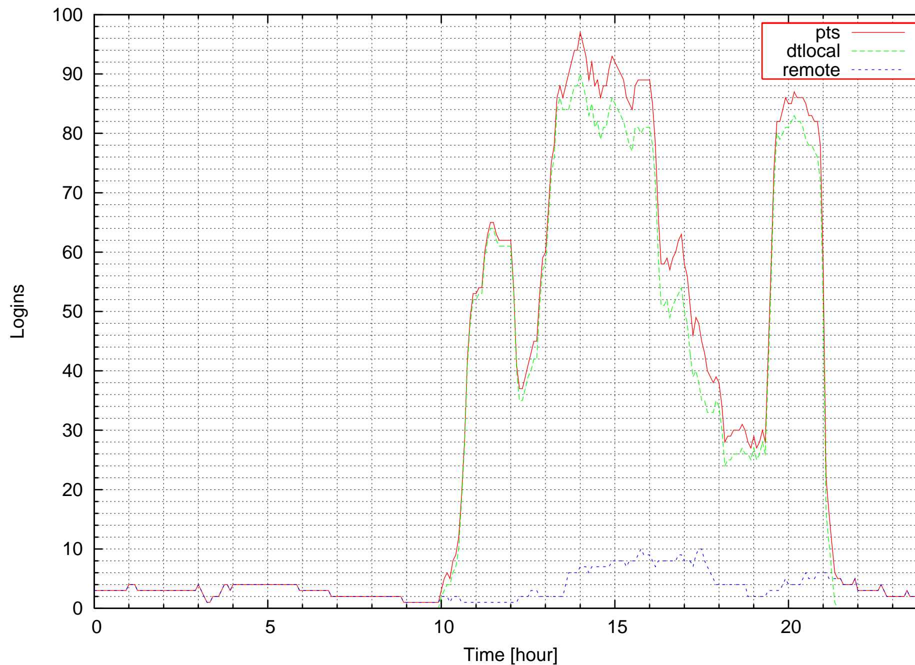
91〜92 行目で用いた `comprate` なるプログラムは C 言語で書いた自前のプログラムで，第一引数を分母，第二引数を分子に取って分数演算した後，`double` 型の浮動小数を返します．`bc` では桁が足りなかったのでやむなく作りました．source code を公開する程の物でもございません．ある日付に構成された `time-ninzu.dat` なるプロットデータの 20 時から 1 時間分のデータを切り出して中身を確認してみます．(左から時間, `pts`, `dtlocal`, `remote` の順)

```
20.001111 85 81 4
20.084444 86 82 4
20.167778 87 83 4
20.251111 86 82 4
20.334444 86 82 4
20.417778 86 81 5
20.501111 85 79 6
20.584444 83 78 5
20.667778 83 78 5
20.751111 82 77 5
20.834444 82 76 6
20.917778 78 72 6
```

2.3 makegraph.sh

```
1    #!/usr/bin/bash
2    year=`env LANG=C date | awk '{print $6}'`
3    month=`env LANG=C date | awk '{print $2}'`
4    day=`env LANG=C date | awk '{print $3}'`
5
6    [中略]
7
8    prefix=~/var/log/$year-$month-$da
9    prefix2=`which gnuplot`
10   $prefix2 << EOT
11   set term post color eps
12   set out "$prefix/time-ninzu.eps"
13   set key top right
14   set key box lt 1 lw 2
```

```
15   set style line 1 lt 1 lw 1
16   set grid xtics ytics mxtics mytics
17   set mxtics 5
18   set mytics 5
19   set xlabel "Time [hour]"
20   set ylabel "Logins"
21   plot [0:24] "$prefix/time-ninzu.dat" using 1:2 with lines ti "pts",\
22   "$prefix/time-ninzu.dat" using 1:3 with lines ti "dtlocal",\
23   "$prefix/time-ninzu.dat" using 1:4 with lines ti "remote"
24   exit
25   EOT
```



10〜25 行目ではシェルスクリプトのヒアドキュメント機能を用いました．これにより動的に gnuplot 用のスクリプトを構成することができます．作成されたグラフは上図の様になります．確かに pts=dtlocal+remote であることがわかりますね．また，図より，2,3,4,7 限目に IED にて授業があったことが推測されます．ずっとログインしっぱなしの人もいますが，たぶんログアウトしてない人かレポートでデスマーチ進行で修羅場ってるんでしょう．

2.4 archive.sh

```
1    #! /usr/bin/bash
2    year=`env LANG=C date | awk '{print $6}'`
3    month=`env LANG=C date | awk '{print $2}'`
4
5    [中略]
6
7    prefix=~/var/log
8
9    if [ $month -eq 1 ];                            # 1 月前の月及び年の設定
10   then                                            # 1 月であれば年と月を設定賢く設定
11           year=`expr $year - 1`;
12   month=12
13   else
14           month=`expr $month - 1`
15   fi
16
17
18   cd $prefix
19   ls -a | grep $year-$month | grep -v tar > list  # 前月の日付ディレクトリリストを取得
20   Upper=`wc -l list | awk '{print $1}'`           # 前月のログをとった日数を取得
21
22   timelabel=`head -n $Upper list`                 # リストを一行に展開
23   hiduke=($timelabel)                             # 配列に格納
24
25   cnt=0
26   DecUpper=`expr $Upper - 1`
27   while [ $cnt -le $DecUpper ];
28           do
29           gtar cvjf ${hiduke[$cnt]}.tar.bz2 ${hiduke[$cnt]} # bz2 形式に圧縮
30           rm -r ${hiduke[$cnt]}
31           cnt=`expr $cnt + 1`
```

```
32  done
33
34  rm list
```

形式的には recw.sh とほとんど変わりはありません．

2.5 table.cron

実は Sun Solaris の cron はあまり賢くなく非常に読みにくいスクリプトになるので省略します．内容は以下の通りです．

(1) 毎日 0 時 0 分に makedir.sh を実行する．
(2) 5 分ごとに recw.sh を実行する．
(3) 5 分ごとに sleep 5 した後に，makegraph.sh を実行する．
(4) 新しい月に変わったら 0 時 0 分に table.cron を実行する．

3 その他の運用例

本稿で示した実装内容は一例に過ぎません．ログイン ID は各日付フォルダに 5 分おきにログがとられているので，recw.sh で作成した time-ninzu.dat を用いてある月での平均ログイン数の推移，月毎の平均ログイン数の推移なども計算することが可能です．

また，dtlocal シグナルに着目すればそのシグナルを出したユーザーのログイン時間を推測し，それから「某君は遅刻している」といった情報を得たりすることもできます．はたまた，複数のユーザが授業時間以外に接近した時間に Sun-Ray 端末からログインしたとき，彼らは近い距離ないし友人関係になる可能性が高いといえます．それらの組み合わせを調べ，頻度などを計算することにより「距離の近いユーザグループ」を抽出することも可能でしょう．

4 おわりに

新システムの IED では執筆段階では詳細な機能が未公開なため言及することは出来ませんが，情報基盤センター (CC) のサーバにも着目するシグナルを変更すれば本稿で示したスクリプトも部分的に流用でき，本稿に近い事は出来ます．

本稿で述べた程度の機能なら perl や ruby, python といった流行のスクリプト言語や C/C++ 言語といった高級言語を用いずとも実装できるということが趣旨の一つでもあります．

第 II 部

500 円 Web カメラを NetBSD 上から無理やり使う

(著者) oku
oku@mma.club.uec.ac.jp

1 はじめに

elecom わけありショップ[*1]にて 500 円を切るという暴力的な値段で販売されていた Web カメラを不正なテクニックを駆使してつかおうという話。同じチップを乗せた奴なら同様に使えると思います。

2 実は解析の手間もなく。。

この Web カメラ (UCAM-E1C10MDSV) に使われているチップは PixelArt の PAC207 というチップで、実は Video4Linux にすでに対応コードが有ったりします。

というわけで、今回は Video4Linux のルーチンを移植することで、実際にデバイスを解析することはしないで操作コードを実装することにします。

3 BSD ugen による USB デバイスの操作

今回のデバイスはアイソクロナス転送を利用しているため、libusb[*2]が使えません。というわけで、BSD 専用になってしまいますが、ugen を使うことにしました。

ugen とは、USB デバイスのドライバがカーネルに存在しないときにとりあえずアタッチされるドライバで、このドライバの作るデバイスノードを利用することで、エンドポイントを直接操作することができます。便利。

3.1 ugen によるコントロール転送

```
void
pa_regWriteByte(pa* p,int index,char c){
        struct usb_ctl_request req;

        req.ucr_request.bmRequestType = UT_VENDOR;
        req.ucr_request.bRequest = 0;
        USETW(req.ucr_request.wLength,0);
        USETW(req.ucr_request.wValue,c);
        USETW(req.ucr_request.wIndex,index);
        req.ucr_data = NULL;
        req.ucr_flags = 0;

        ioctl(p->fd_cntl,USB_DO_REQUEST,&req);
}
```

ugen によるコントロール転送は 0 番目のエンドポイントを開き、ioctl することで実現します。

PAC207 におけるコントロール転送はレジスタへのアクセスくらいしか用途がないので、専用の関数に分割しています。

3.2 ugen によるアイソクロナス転送

```
FD_ZERO(&fds);
FD_SET(p->fd_iso,&fds);
tv.tv_sec = 0;
tv.tv_usec = 90*1000;

select(p->fd_iso+1,&fds,NULL,NULL,&tv);
size = read(p->fd_iso,&gbuf[mp],4096);
mp += size;

if(mp > 3762*63){
mp = 0;
}else{
continue;
}
```

[*1] 楽天市場に存在する elecom の不良在庫ショップ
[*2] USB バスを直接叩ける便利なライブラリ

実際の画像データの転送を行う、アイソクロナス転送/バルク転送は read() でエンドポイントを直接読み取ることで行います。

エンドポイントに対するアクセスはなぜかブロッキングするので、select() を用いてタイムアウトを実現しています。もっとも、これはあまり良い書きかたでは無いので select の使いかたの参考にはしないように。

4 PAC207 固有の事情

4.1 レジスタと初期化

レジスタを適切に初期化する必要が有ります。これは Video4Linux のコードをそのままパチってくれば OK です。詳細は wiki.osdev.info/?PAC207 で。

4.2 流れてくるデータ

```
[FF][FF][00][FF][96][64] : HEADER
[SEQ][00] : check sequence
[PXCLK] : pixel clock
[Brightness1][Brightness2] : brightness
[Rgain][Ggain][Bgain] : analog gain
[F0][00][MARK1][MARK2] : unknown
[DATA]...
```

流れとしては、ストリームとしてエンドポイントを受信し、HEADER を検出して処理という感じで。USB では一度の転送で 1k バイト未満しか転送できないので、大きなデータを転送するためにはヘッダを必要としています。

4.3 レートコントロールによる自動圧縮の回避

PAC207 には、転送レートを監視して勝手に圧縮する機能が有ります。USB の転送レートは 12Mbps で決めうちじゃないかと思われるかもしれませんが、基本的に USB ではホストが要求しないかぎりデバイスはデータを送信できないので、ホストが自由にデバイスの帯域をコントロールすることができます。

で、普段べたで送って来ているものが、急に圧縮されたデータになっても困るので、何らかの方法でこれを禁止する必要があります。

今回はレート調整のしくみがよく分からなかったので、ピクセルクロックを下げてスキャンされる画像の枚数を減らすことで帯域を絞る方法を取っています。

もちろん、良いフレームレートを得るためにはどうしても圧縮されたフレームを解釈できる必要があるので、それは今後の課題として。。

4.4 イメージセンサの色配置

この手の安い撮像デバイスは、モノクロのイメージセンサにカラーのフィルタを被せて実現されていることが殆どです。もっとも、安くないデジタルカメラでもこのフィルタの影響を受けてしまいますが。

色配置はいわゆる Bayer 配列となっています。

```
BGBGBGBGBGBGBG....
GRGRGRGRGRGRGR....
BGBGBGBGBGBGBG....
GRGRGRGRGRGRGR
:
```

というように、B や G 一つにつき G が二つあるので、緑色の感度が倍ということになります。もし、撮影したモノクロ画像に色をつけたいのなら、このカラーフィルタの並び順に色を乗せてやればそれっぽく見えます

4.5 バルク転送はおすすめできない

バルク転送でも画像を得ることが一応可能ですが、バルク転送はアイソクロナス転送に比べて遅く、Web カメラとしては別にフレームを取りこぼしたところでこれといった不都合は無いので、極力バルク転送は使わないことをお勧めします。

また、PAC207 のバルク転送は 64Byte づつしか転送できないので、いちどに最大 1023 バイト送信できるアイソクロナス転送パイプの方が確実に有利になっています。

5 おわりに

宣伝。オープンソースマガジン 7 月号に、同じような感じで PaSoRi をつかって Suica の残額を調べたりする記事を書いたので、この辺に興味のある方は是非[*3]。

某 MYCOM ジャーナルにも載りました。詳細は libpasori で検索してねということで。。

*3 もっとも、OSM 自体は休刊がすでに決定してしまいましたが。。

第 III 部

FPGA による俺コンピュータ

(著者) oku
oku@mma.club.uec.ac.jp

概要
FPGA でアクセラレータを作る　とかならまだ実用的でカッコイイが、そういう部分ではなくあくまで既存の環境を頑張って捨て去る無駄な努力に意味を見出す。まだ未完です。平たく言えば USB とソフトが未完成。

1　FPGA とは何ぞや

FPGA とは Field Programmable Gate Array の略で、要は出荷された後にも書き込める (Field Programmable)、カスタマイズ可能な論理回路 (Gate Array) の事。

FPGA はユーザがそれなりの自由度で配線を弄れる LSI で、FPGA の中にはいくつか (5000〜20 万個くらい) の機能ブロック[*4]が内蔵され、ユーザは機能ブロックの真理値表とブロック同士の配線データを書き込んで使います。

FPGA の便利なところは、LSI の設計図さえ用意して書き込んでしまえば、ユーザが好きに LSI を作れてしまう点。ネット上には実際に Z80[*5]とか SPARC[*6]といった CPU と互換性のある CPU の設計が公開されていて、それを FPGA に書き込んでしまえば Z80 とか SPARC を買ってこなくても、FPGA が擬態してくれるのでオッケーということです。

もっとも、FPGA の通常の活用法は既存の LSI を真似っこをさせることにはなくて、あくまで自分でオーダーメイドな LSI を作れるというところになります。

2　おうちでできる LSI 設計

昔はそれこそ大学とか企業でしか出来なかった LSI 設計遊びですが、今はかなり手軽にできるようになりました。

2.1　環境の整備

FPGA のメジャーなベンダは Xilinx(ざいりんくす) と Altera(あるてら) の 2 社で、この 2 社はどちらも設計ツールの無料バージョンを公開しています。よってソフトに関しては真っ当な PC[*7]さえ用意してやればタダで済みます。

問題は FPGA そのものをどうやって入手するかという点で、通常のヒューマンには FPGA のチップを配線するための基板など

もちろん、FPGA が載ってるだけではどうにもなりません。コンピュータを作りたいならメモリとかも要りますが、FPGA でメモリを作っても大した容量にはならないので[*8]予めメモリの載った基板をチョイスすべきでしょう。

要件をまとめると、

- 最低でも 10 万ゲート規模以上の FPGA が載ってる
- 一緒に RAM も乗ってる
- 普通のコネクタが付いてる
- Web で検索してそれなりに引っかかる

この条件に合致しているのは現時点では Spartan3 Starter Kit くらいだと思います。これは 15,000 円くらいで Xilinx が布教用に投売りしている基板で、基板の上には FPGA 以外にもあらかじめ VGA コネクタとか RAM が載っているのでお買い得。

ちなみに、このキットは安いのでネット界隈では NintendoDS のハックとか 2ch のトリップ計算にも活躍していました。個人でもそういうツールが作れるようになったのが FPGA の良い所です。

[*4] ロジックセルとかロジックエレメントとか言います
[*5] Gameboy の CPU の基になった CPU
[*6] 電通大ではサーバに良く使われている CPU
[*7] 512MB 以上の RAM と 1GHz 以上のクロック
[*8] 餅は餅屋。そもそも FPGA はそういうものを作るのには向かない。

他にコンピュータを作るという目的にマッチしているのは 1chip MSX というボードが有るんですが、こちらは限定生産なので入手性に疑問が。

2.2 HDL 記述

首尾よく FPGA の載った基板と開発環境を準備したら、いよいよ設計します。

現在、FPGA に書き込む回路の設計はハードウェア記述言語 (HDL) と呼ばれる種類のプログラミング言語で行います。実は、ASIC のような普通の LSI でも同じです。

HDL の中でもメジャーなのは VHDL か VerilogHDL ですが、両者ともやってることは大して違いません。

HDL で記述することは C 言語のような通常のプログラミング言語と大して変わりないです。VHDL で、次のように書けば、端子 a,b から入力したデータの AND を取って端子 x に出力する LSI がつくれちゃいます。

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3
4   entity sample_1 is port (
5           a,b : in std_logic;
6           x : out std_logic );
7   end sample_1;
8
9   architecture rtl of sample_1 is
10  begin
11          x <= a and b;
12  end rtl;
```

実際にはもっと高度な記述も出来ますが、要はソフトを書くようにハードが作れちゃう[*9]のが FPGA の良い所です。

3 アーキテクチャのデザイン

俺コンピュータを作るにあたって、FPGA に載せる回路の全体像を考えます。

3.1 独自アーキテクチャの賛否両論

せっかく CPU が自作できるんだから、自分オリジナルのアーキテクチャや命令セットにしたくなるのが人情というものです。が、独自アーキテクチャにするのはメリットもあればデメリットも有ります。

最大のデメリットは「ソフトウェアの流用が出来ない」という点でしょう。C コンパイラとかを移植してくれれば不可能では無いですが、C コンパイラを移植するのは結構面倒だったりします。

この点を乗り越える自信が無ければ、既存のコンピュータを再現する方向に走るのもアリでしょう。実際、先ほどチラっと出てきた 1chip MSX は FPGA で MSX という古のパソコンを再現したものですし、Spartan3 Starter Kit を使って懐パソ (懐かしいパソコン) とかアーケードゲーム基板を再現しようという試みは結構有ります。

それでも独自アーキテクチャを作りたがるのは既にロマンとかそういうレベルでしか無いですが、まぁ面白いのでオッケーです。

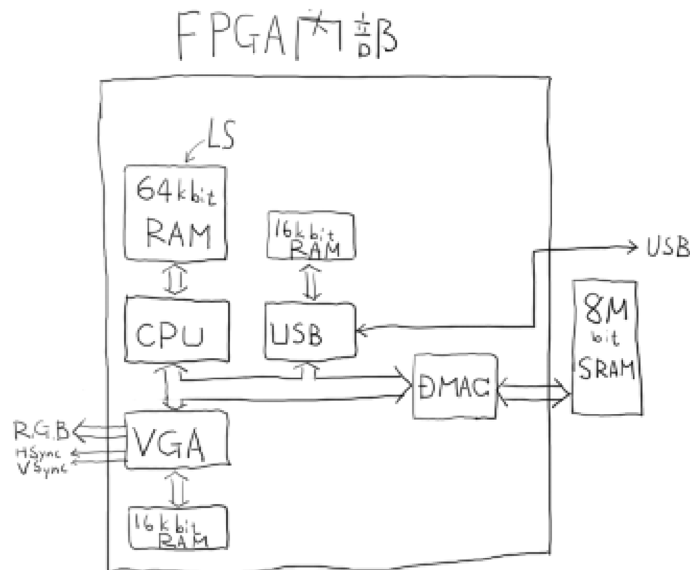
[*9] 実際にはハードウェア特有のどろどろした問題と戦う必要はありますが。。信号のタイミングとか。

3.2 独自アーキテクチャの考え方

既存の環境を捨て去る決意が出来たら、命令セットやバスインターフェースの設計に入ります。

ゼロから考えるのは大変なので、既存のアーキテクチャを参考にすべきでしょう*10。

今回はこんな感じの構成を考えました。



大きな四角で囲んだ中が FPGA のチップの中で実現されるべきモノです。モジュールに RAM が繋がっていますが、今回使っている FPGA では内部に RAM の機能ブロックがあるのでそれを使います*11。

大きな RAM は基板の上に一つしかない*12ので、CPU と USB、VGA で共有する必要が有ります。PC で言うとオンボードグラフィックスでよく出てくる UMA って奴です。あと、USB のような外部 I/O と CPU で RAM を共有するのは、実はかなり普通です。DMA って奴です。

CPU に繋がっている RAM の上に LS と書いてあります。これは今話題の PLAYSTATION3 の CPU、Cell をパクったもので、LS とは Local Store の略です。常識的な感覚では LS にあたるメモリはキャッシュメモリにします。普通の CPU ならこの部分は 1 次キャッシュにしますが、今回はキャッシュメモリを実現するのが面倒だったので。

つまり、今回の CPU は 8k バイトまでのコードしか実行できないって事です。それ以上の長さのプログラムを実行したいなら何らかの方法で分割する必要が有ります。

Cell の SPE も一つあたり 256 キロバイトの LS 上に置いたコードしか実行できないようになっています。これはキャッシュの実装をサボったのではなくて、プログラムによってより高度で効率の良い制御が出来るだろうと考えたからです。もっとも、実際にコードを書いてみると 256 キロバイトでは色々と辛いようですが。。

要は Cell と同じくソフトウェアで努力することを前提にハードウェアを考えています。通常の人間にはここまで変態的な構成にするのはおススメできません。素直に出来合いの CPU に好きな命令を追加する　みたいな所から始めてもいいと思います。

どっちのせよ、考え始める前に、FPGA のプログラミングがどういうものか知る必要が有ります。FPGA で実現するという縛りが有る以上は。

*10 実は、今回設計するアーキテクチャはこの辺の記述とは逆方向に設計しています。先に OS やソフトウェアを想定して、適したハードウェアを考えるという方法。

*11 高級な FPGA になると CPU そのものが入ってたりしますが。。

*12 実際には 2 つのチップが載っているが、別々に独立して使うようには出来ていない

4　どう実現するか

机上の空論を現実にするお時間です。実際のチップで動き始めるのを現実とすると、実際の設計では机上の空論を「段階的に」現実に近づけていくことになります。

CPU について解説したい所ですが、それだけで一冊分のボリューム[*13]になってしまうので、今回は CRTC(図中では VGA) を例に取り上げます。

CRTC とは、(今回のシステムでは) メモリ上のデータを VGA 端子経由でモニタに写すためのモノです。

4.1　論理回路でどう実現されるのか

FPGA の中では全て論理回路として物事は実現されます。要は、VHDL で設計した LSI は and とか or とか xor の組み合わせで実現されるって事です。

HDL で設計するためには、論理回路に対する知識が必要になります。普通のプログラミングではデザインパターンとか単に推奨されるルールしか存在しませんが、HDL による設計では FPGA 上で実現できる回路のパターンを予め理解しておいて、その組み合わせで設計する必要が有ります。逆に、そのパターンに収まっていれば、自分で and とか or とか xor を組み合わせる必要はありません。

いくつかの単語を覚える必要が有ります。

- 同期回路
- 非同期回路
- 順序回路

同期回路とはクロックの入力を基に「いっせーの、せ」で動作する回路のことで、非同期回路はクロックとは関係なく、入力を xor とか or とか and して出力する回路の事です。これらは適材適所で使い分ける必要が有ります。

足し算とか掛け算といった大抵の演算回路は and とか or とか xor で実現されます。足し算回路を作るために自分で and とか or とか xor を組み合わせる必要は無くて、予めライブラリが作られていて、ソースコード中にｘ＜＝Ａ＋Ｂとか書けば適当に生成されます。

演算回路と違ってちょっと身近じゃないのは順序回路。順序回路とはいわゆる「状態遷移」を電子回路で実現するための回路です。同期回路としても非同期回路としても実現することが出来ますが、普通は同期回路として実現されることの方が多い気がします。

大抵の LSI は順序回路を実現するために存在するので[*14]、当然の如く FPGA では順序回路を実現するための特別な機能がいくつか有ります。そんなことは意識せずに普通に回路として書けば良いんですが。

順序回路は普通のプログラミング言語で言う所の変数にあたる、レジスタを使って実現します。例えば代表的な順序回路であるところのカウンタは VHDL では次のように書けます。

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.std_logic_unsigned.all;
4
5   entity sample_2 is port (
6           clk : in std_logic;
7           x : out std_logic_vector(7 downto 0) );
8   end sample_2;
9
10  architecture rtl of sample_2 is
11  signal cnt : std_logic_vector(7 downto 0);
12  begin
13          process(clk) begin
14                  if(rising_edge(clk)) then
15                          cnt <= cnt + '1';
16                  end if;
17          end process;
18
19          x <= cnt;
20
21  end rtl;
```

[*13] 実際、『CPU の創り方』という素晴らしい本が出てます

[*14] CPU だって順序回路です

clk のがゼロから 1 になったときに x が 1 づつ増えるカウンタ、signal というのがレジスタの宣言になってます。

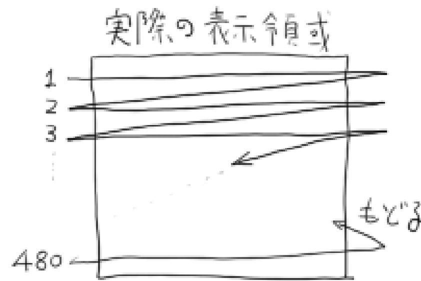
これだけみると普通のプログラミング言語のように見えますが、普通の言語とちょっと違うのは、cnt と x が常に同じ値に保たれるという事でしょう。非同期回路として記述した代入、process 文の外側の代入は「回路的に接続する」事を意味します。どのような条件で回路が接続されるのかを別途指示しないかぎり、常に接続される事になります。

4.2 モニタの映る仕組み

CRTC を作るためにはモニタの映る仕組みを知る必要があります。

VGA 信号にはアナログな R と G と B の信号、0 と 1 の VSYNC と HSYNC が有ります。今回のシステムでは R と G と B も 0 と 1 の 2 つしかないと扱うので、全体で 3bit、つまり 8 色しか出ないことになります。

VGA 信号のうち、R と G と B は紆余曲折して電子ビームとして出てきます。これを適切に動かしてやることで画面を描きます。



概念上は、こういう風にギザギザと電子ビームを動かします。ギザギザのうち、左から右に動いている間だけビームを出力することになります。

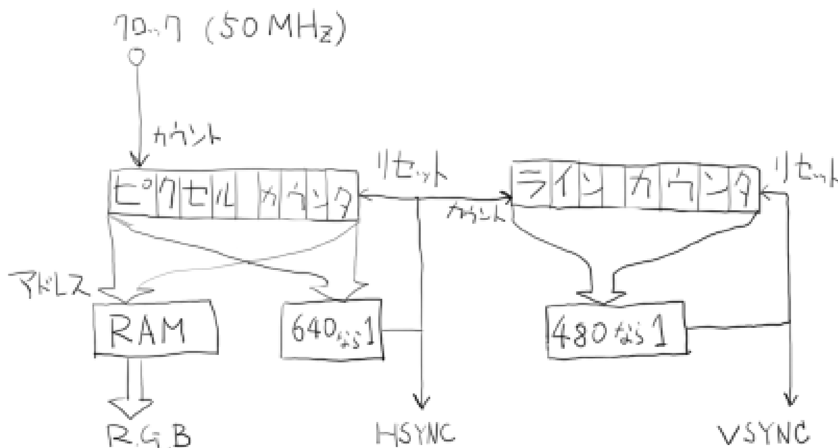
その動きを制御するのが 2 つの信号、VSYNC と HSYNC です。VSYNC は電子ビームの位置を右下から左上に戻す役割、HSYNC は右端から左端へ戻す役割をそれぞれ担っていて、モニタの設定でよく VSYNC = 60Hz とか言うのは VSYNC が 1 秒間に 60 回発動するって事です。なにもしないとビームはかなり水平、しかし右下へ向かって微妙に下がって　今回の場合、480 回往復する時間で画面の下に届くくらいの勢いで動き続けると考えます。

そして、CRTC の仕事は、次のように纏められます。

- メモリから出力された内容を RGB として出力する
- 480 ライン出力したら VSYNC 発動
- 640 ピクセル出力したら HSYNC 発動

4.3 何はともあれ分割する

さっきのまとめを、回路として実現できるレベルまで詳しく分割していきます。



この図には (実際に製作の過程でやってしまった) 間違いがあります。*15

これはかなり単純化していて、このままでは映らないんですが、だいたいこういう感じにまで機能を分解できたら、HDL で記述していきます。

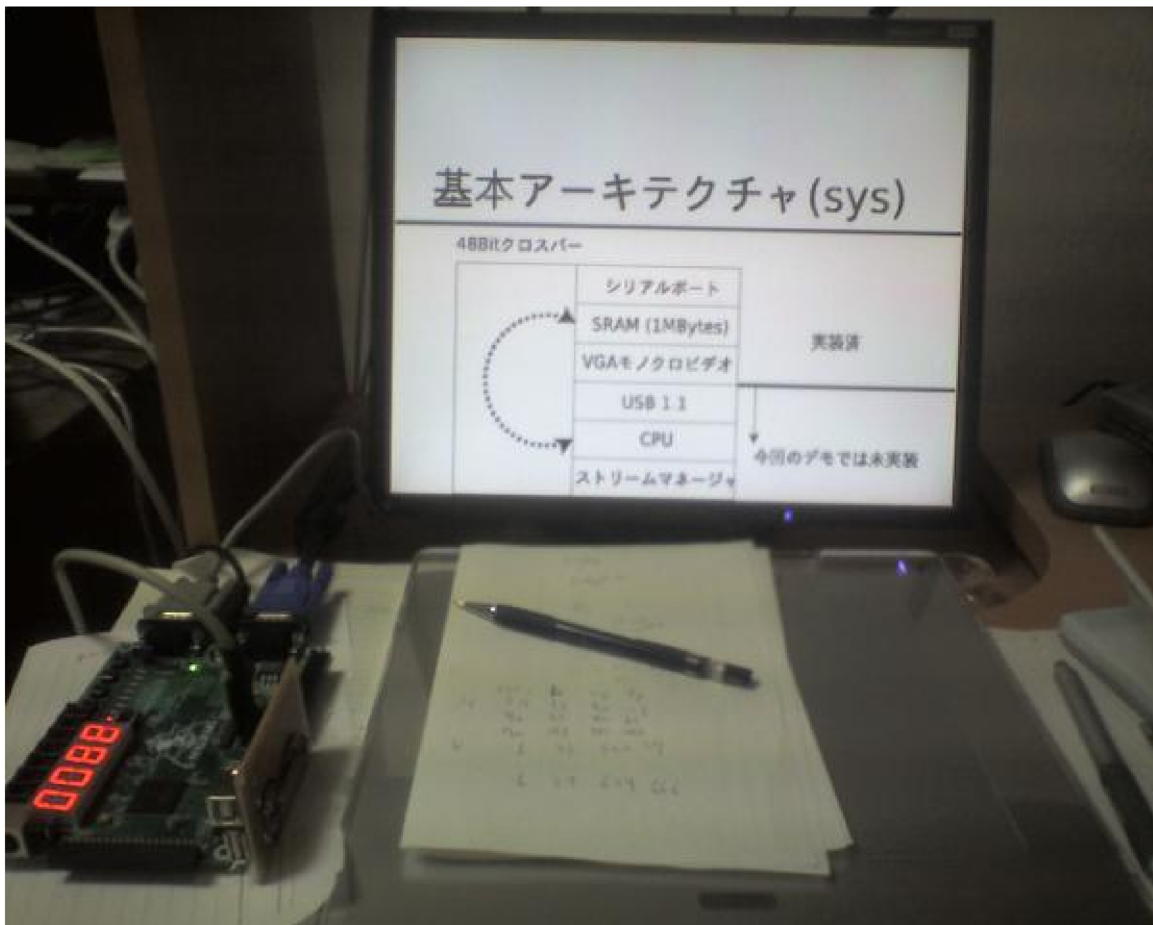
慣れれば、正確に言えば慣れてなくても、いちいち絵を描く必要は無いんですが、重要なのは絵が描けるレベルまで分解する必要が有るということ。そこまで出来れば、あとは HDL の文法さえ知っていれば書けます。

VHDL なソースは割愛しますが、ほぼ絵をコードに起こしただけになります。

ここでは簡単な CRTC だけを作りましたが、もっと複雑な CPU であってもやることは一緒です。ただし、実現しようとするものが複雑になってくると物理的な制約を受けることになります。

5 おわりに

本来的にはここから CPU の設計、コンパイラ、OS と進んでいくんですが、その辺は幾ら書いても終わらなさそうなのでまたの機会*16に。。



試作システムの動作中

*15 このままでは、縦方向の縞模様しか写せない。なぜだかは考えてみよう。
*16 書籍として執筆中です

第 IV 部
FreeBSD and CompactFlash Devices

(著者) mura
mura@mma.club.uec.ac.jp

概要
本稿は、急激に大容量低価格化した CompactFlash に、FreeBSD をインストールし活用するためのノウハウを記述したものである。フラッシュメモリーデバイスの大容量化に伴い、過去の文章で多くの説明が割かれていた使用領域の削減には重点を置かず、書き換え回数制限の克服方法や、OS 全体のアップグレード方法などに重点を置き単なる探究に留まらず常用できる環境の構築を目指している。

1 背景

2006 年度は CompactFlash を含むフラッシュメモリーデバイスが急激に大容量・低価格化した年であった。これにより、比較的大きなオペレーションシステムをフラッシュメモリー上にインストールし活用することが現実的となった。

1.1 フラッシュメモリ

フラッシュメモリーデバイスはハードディスクと比べ、以下のような優れた特徴がある。

- 消費電力が少なく低発熱
- 機械的動作部分が存在しないことによる高い信頼性
- ランダムアクセスが高速
- 軽量

これらの特徴により、ノートパソコンや、ルータやゲートウェイとして動いている計算機、主要な記憶領域としてソフトウェア RAID を用いていたりリモートの NFS を使用している計算機の、ルートファイルシステムなどに用いることが有効であると考えられる。

しかしながら、フラッシュメモリーデバイスには書き換え回数に制限があり、製品により異なるが、大体 10 万回程度の書き換え回数しか保証されておらず、それ以上の書き換えを行うと使用できなくなってしまう。

本稿で扱う CompactFlash の仕様ではローテーションを行うことが規定されており、書き換え回数の制限はある程度緩和されるものの、ログファイルなどの書き込みが非常に多いファイルはフラッシュメモリの寿命を著しく縮める原因となる。

販売されているフラッシュメモリの多くは、NAND 型と呼ばれるものである。NAND 型フラッシュメモリは配線の面積をケチった構造ゆえの特徴がいくつかある。

まず、トンネル効果による浮遊ゲートへの電荷の打ち込みに相当する、書き込み動作はページと呼ばれる単位で行われる。昔は 1 ページ 528 Bytes が主流であったが、最近の大容量[*17]なフラッシュメモリーデバイスでは 2112 Bytes やそれ以上の値であることが多い。

ちなみに、 2112 Bytes などの中途半端な値となっているのは ECC のためなので、フラッシュメモリコントローラの外側からみると 2kBytes などの整った値となる。

また、書き込みを行うには、該当ページは電荷が抜かれ消去された状態となっている必要がある。この消去 (フラッシュ) はページと呼ばれる単位で行われ、一般に 64 ページ/ブロックであることが多いようである。

読込については、アドレスレジスタにアドレスを書き込んでから、1 ページ分のデータがバースト転送されるという流れになっており、実はメモリであるにもかかわらず、ページサイズ以下のランダムアクセスが出来ないという欠点がある。

なお、メモリセル、ブロック、ページの各用語は文献により意味が違うようなので、注意されたい。[*18]

[*17] デバイスの容量が 1GBytes を越えるあたりから大容量品らしい。
[*18] TDK の NAND 型フラッシュメモリコントローラの資料では「ブロックを消去」し「メモリセルに書き込む」となっていた。

2 ブートデバイスとしてのフラッシュメモリデバイス

今日では、さまざまなフラッシュメモリデバイスをブートディスクとして使用可能となっている。これらのフラッシュメモリデバイスを、ブートデバイスとして用いるという観点から比較してみた。

CompactFlash をブートデバイスとして用いる場合は、CF-IDE 変換アダプタを用いる。CompactFlash の仕様は CompactFlash Association [*19] が策定している。

CompactFlash Specification Revision 1.4 にて True IDE モードが規定されている。このモードでは CompactFlash は ATA デバイスとして振る舞う。計算機からは ATA デバイスに見えるのでほとんどの計算機で使用することができるというメリットがある。

USB マスストレージについては、起動デバイスに指定できる BIOS が増えているとはいえ、対応していない BIOS を搭載した計算機も多く使われている。このような計算機でも CompactFlash からの boot は問題ない。

また、 USB マスストレージデバイス自体にもブートディスクとして使用することができない製品が存在するようである。

CompactFlash Specification Revision 2.1 からは Multiword DMA が、同 3.0 からは Ultra DMA が規定されており、対応した CompactFlash と変換アダプタであれば高速な転送を行うことができる。

センチュリーから販売されているシリコンディスクビルダーは、SD カードや CF カードを 2 枚または 4 枚搭載し、計算機と PATA, SATA にて接続するものである。

ストライピング動作を行うことで高速な読み書きが期待できる他、計算機からは ATA のハードディスクと同様に扱えるので多くの計算機で使用することができる。また、同様の理由により OS の対応も申し分ない。

しかしながら、小容量 [*20] のディスクを構築する場合には、2 万円前後という価格からフラッシュメモリデバイスより高くなってしまうという点が最大のネックである。

以上のような考察から CompactFlash を使用することとした。

今回の記事で使用した CompactFlash は Transcend 社の TS2GCF120 [*21] である。CompactFlash Specification 3.0 に準拠し、読み込み 20MBytes/sec, 書き込み 18MBytes/sec, PIO mode 6 および Multiword DMA mode 4 をサポートする。

3 CF-IDE 変換アダプタ

3.1 CompactFlash における DMA

CompactFlash Specification Revision 4.1 の True IDE における DMA について簡単にまとめた。デバイスとは CompactFlash カードのことを指している。

詳細については CompactFlash Association の Web サイトより CompactFlash Specification Revision 4.1 をダウンロードし参照されたい。個人的には、 DMA 転送のタイムチャートなどがとても参考になったので、読者にも一読を勧めたい。

43 番ピンに関する記述を以下に列挙する。

- この信号は DMARQ (DMA Request) として使用される。DMA 転送の準備が出来たときデバイスによりアサートされる。DMACK と合わせてハンドシェイクを行う。
- ホストが True IDE での DMA mode をサポートしていない場合、ホスト側ではこの信号は使用されず、結線もされない。

44 番ピンに関する記述を以下に列挙する。

- この信号は DMACK (DMA Acknowledge) として使用される。DMARQ への応答としてホストによりアサートされる。
- True IDE モードにおいて DMA が有効ではない場合、結線されていない状態を含めて、デバイスはこの信号を無視する。
- True IDE モードのみをサポートするホストにおいて、ホストが DMA をサポートしていない場合は、ホスト側でこの

[*19] http://www.compactflash.org/
[*20] といっても数 GB だが
[*21] http://www.transcendusa.com/Support/DLCenter/EDM/CF120-EDM.pdf

信号を high にドライブするか、 VCC に結線する。

注目すべき点は、 DMA 非対応の CompactFlash カードを使用する場合でも、壊れたりしないよう配慮がなされているという点と、DMA に非対応のホストにおいて、44 番ピンは VCC に結線されているという点である。

3.2 CF-IDE 変換アダプタの問題点

本稿を執筆するにあたり、今まで用いていたアダプタとは別に新たに秋葉原にて CF-IDE 変換アダプタを購入した。

ところが、どちらの変換アダプタもブート中に kernel が DMA を有効にする段階で、ディスクが DMA 要求に応答しない旨のエラーを出して止まってしまう。

先輩の助言を元に調べてみたところ、市販の多くのアダプタは CompactFlash Specification Revision 2.1 以前の仕様にしたがい、43pin は NC, 44pin は H に結線されたものであるということが判明した。

google にて “CompactFlash pinout” をキーワードに検索すると、古い仕様に準じた DMARQ, DMACK についての記述がない文章が多数ヒットすることから、設計者は無料で読むことができる CompactFlash Association の仕様書すら読んでいないと思われる。

PIO は低速なので、せっかくのランダムアクセス性能を生かしにくくなるだけでなく、CPU 負荷が高くパフォーマンスを劣化させる。できる限り DMA を有効にしたいところである。

したがって、同様の CF-IDE 変換アダプタを購入される際には、CF 43pin - IDE 21pin と CF 44pin - IDE 29pin が結線されていることを確認することを勧める。

3.3 DMARQ,DMACK の結線

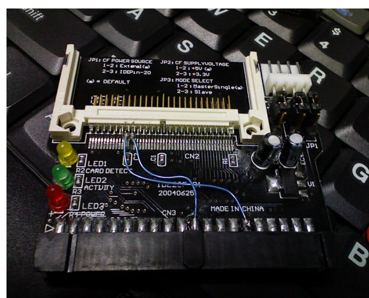
FreeBSD においては、loader.conf に `hw.ata.ata_dma="0"` を追加するか、bootloader にて `set hw.ata.ata_dma="0"` と入力することで PIO で使用することが可能である。

今回の計算機はルータとして働かせる予定なのでディスクの速度は特に必要ではなく、 PIO で動作させることも考えた。しかし、それでは面白くないし百万石のネタにするには物足りないので、DMARQ と DMACK を結線してみることとした。

前述のとおり、 DMA 非対応のホストにおいて 44 番ピンは high にドライブされているか、 VCC に結線されている。したがって、結線を行う前に CF-IDE 変換アダプタの 44 番ピンは浮かせる必要がある。

作業手順を以下に示す。

- CompactFlash のソケット側の 43,44pin を浮かせる。はんだがしっかり付いていないので、ピンセットですこし力を加えると外れた。
- 皮膜線を剥き、先端部分にピンセットで輪を作る。
- CompactFlash ソケットの浮かせたピンに通しはんだ付けする。はんだで少し玉ができるので他のピンから離れた位置につけること。
- 短絡箇所や誤結線時に機器が損傷することを防ぐため、IDE コネクタの 21,29pin に 10 オームの抵抗を取り付ける。
- 皮膜線の反対側も剥いて輪を作り、抵抗とはんだ付けする。
- 短絡がないことを確認後に計算機に取り付け、正常に動作することを確認する。
- 正常に動作することが確認できたら抵抗を取り外し直接結線する。(下図)



CompactFlash slot の 43 番ピンは浮かせる必要はないのだが、他のピンとの接触を回避するために浮かせた。

IDE 側の 21,29 番ピンについては仕様書を確認するなどはしていないが、目視でパターンを追い結線されていないことを確認した。

この改造により、 FreeBSD において Multiword DMA mode 2 で動作するようになった。以下は FreeBSD 6.2-Release における dmesg の出力である。

```
ad0: 1983MB <TRANSCEND 20060911> at ata0-master WDMA2
```

4 FreeBSD on CompactFlash

4.1 レイアウトの方針

過去に記述された CompactFlash へのインストール記事では、大抵の場合 CompactFlash 上のファイルシステムをリードオンリーでマウントしている。

この方法では、 CompactFlash への書き換えは完全に発生しなくなるが、当然、リードライトでマウントしなおさなければデータを書き込むことができない。よって、ユーザーデータの保存とシステムのアップグレード方法に頭を悩ませることとなる。

そこで、今回は FreeBSD の rc スクリプトである rc.d/var と rc.d/tmp により、頻繁に書き換えられるディレクトリのみに mdconfig によるメモリディスクを割り当て、残りの CompactFlash 上のファイルシステムは読み書き可能でマウントすることとした。

この手法では、 FreeBSD ユーザが普段から慣れ親しんでいるソースコードからの build をそのまま用いてアップグレードを行うことや、ports をコンパイルし導入することを、ハードディスクを使用した計算機と同じ感覚で行うことができる。

4.2 rc スクリプトによるメモリディスクの作成

/etc/rc.d/var と /etc/rc.d/tmp が肝となる rc スクリプトである。これらは、 NFS root などを用いているために /var, /tmp に書き込むことができない場合にメモリディスクを作成しマウントする。

今回は、 rc.conf に `varmfs="YES"` `tmpmfs="YES"` と記述することで、/var, /tmp が読み書き可能な場合においてもメモリディスクを作成するようにした。

おおまかに rc.d/var の動作をまとめた。

(1) rc.subr および rc.conf の読み込みを行う。
(2) `varmfs="YES"` となっているか、または、/var への書き込みができない場合に `mount_md()` によりメモリディスクを作成しマウントする。
(3) `populate_var="YES"` となっているか、または、/var が空である場合に、`populate_var()` により mtree で必要なディレクトリ等を作成する。

同様に rc.d/tmp についてもまとめた。

(1) rc.subr および rc.conf の読み込みを行う。
(2) `tmpmfs="YES"` となっているか、または、/tmp への書き込みができない場合に `mount_md()` によりメモリディスクを作成しマウントする。

どちらも、詳細は rc.d/var および rc.d/tmp を参照のこと。コメントなどは充実しており、特に問題なく読めると思う。

ただし、当然のことではあるが、これらのメモリディスクは reboot などを行うとデータが失われてしまう。/tmp については reboot 時にファイルが失われても特に問題とはならないが、/var/db 以下には消えると困るものが複数存在する。

たとえば、 /var/db/pkg が消えると導入した ports の情報が失われ、pkg_delete などを行うことが出来なくなってしまう。

これらへの対処として、フラッシュメモリ上のパーティションである、/usr/db 以下にシンボリックリンクを張るようにした。[*22]

シンボリックリンクを張るために rc.d/var に手を加えるが、このファイルは /etc 以下にあるため、アップグレード時など

[*22] reboot 時に /var/db 以下をフラッシュメモリ上に書き出すことも考えたが、予期せぬ電源断などでデータが失われてしまう。

においても mergemaster などで適切に取り扱われる。そのため手を加えても問題ないだろうと判断した。

以下に、デフォルトのスクリプトとの diff を示す。

```
1   *** /usr/src/etc/rc.d/var       Thu Mar  3 01:41:35 2005
2   --- /etc/rc.d/var       Sat Mar 10 17:40:18 2007
3   ***************
4   *** 39,44 ****
5   --- 39,47 ----
6     _populate_var()
7     {
8           /usr/sbin/mtree -deU -f /etc/mtree/BSD.var.dist -p /var > /dev/null
9   +       ln -sFf /usr/db/pkg /var/db/
10  +       ln -sFf /usr/db/ports /var/db/
11  +       ln -sFf /usr/db/sup /var/db/
12          case ${sendmail_enable} in
13          [Nn][Oo][Nn][Ee])
14                  ;;
```

5 ファイルシステムのブロックサイズ

前述したとおり、 NAND 型フラッシュメモリはその構造ゆえに、2kBytes のページ単位で読み書きが行われ、64 ページ、すなわち 128Kbyte のブロック単位で消去が行われる。

よって、ファイルシステムのブロックサイズについても、これらのことを考慮して決定することで寿命を延長し、読み書きの速度を向上させることが可能となるはずである。

Transcend は比較的データシートなどの公開している方であるが、TS2GCF120 のデータシート*23 からはページサイズおよびブロックサイズに関する情報は得られなかった。そのため、近年の大容量品では一般的な値となる 2kBytes ページ、128kBytes ブロックを仮定することとした。

UFS2 はブロックのフラグメントをサポートしているため、ブロックより小さなファイルを効率よく扱うことができる。ただし、 UFS2 におけるブロックのフラグメントは 8 分割以外の値はパフォーマンスの面から用いるべきではないとされている。

CompactFlash 内部にある制御チップ、および、FreeBSD のファイルシステムの実装にもよると思うが、ファイルシステムのブロックサイズを 2kBytes 以下にした場合、1 回のファイルへの書き込みで同じページに 2 回以上の書き込みが発生してしまうと考えられる。

ページサイズ以下の書込は CompactFlash 内部でキャッシュを行い、あとでまとめて書き込むような実装により回避されているとも考えられるが、一度に大きな領域を消去するフラッシュメモリの特性により、突然の電源断で書き込んでいる最中のデータどころか既に書き込んであったデータまで消えてしまうことも考えられるので、そのような実装をしている CompactFlash は少ないだろう。

フラッシュメモリのブロックサイズは 128kBytes とかなり大きいことから、UFS2 のブロックサイズをこれに合わせると、小さなファイルでの無駄があまりに大きくなり現実的とは言えない。

フラグメントのサイズがページのサイズと一致する 16kBytes ブロックか、ページサイズと同じ 2kBytes ブロックが妥当なところであると思われる。書き換え回数の観点から見れば 16kBytes ブロックがよいと言えるが、ports や src を展開する場合には 2kBytes ブロックも十分に検討の価値がある。

なお、 UFS2 におけるデフォルトのブロックサイズは 16kBytes であるので、デフォルトの値を用いてもあまり問題とならないだろう。

6 実行結果

実験には EPIA-E533 *24 を使用した。VIA C3 533MHz をオンボードで搭載した小型のマザーボードである。

6.1 速度

PIO mode 4 における dd の実行結果を以下に示す。

```
enum# atacontrol mode ad0
```

*23 http://www.transcendusa.com/Support/DLCenter/Datasheet/TSXGCF120.pdf
*24 http://www.viatech.co.jp/jp/products/mainboards/mini_itx/epia/

```
current mode = PIO4
enum# dd if=/dev/ad0 of=/dev/null bs=1m
1983+1 records in
1983+1 records out
2079350784 Bytes transferred in 365.251764 secs (5692925 Bytes/sec)
enum#
```

top にて確認したところ、PIO mode 4 で dd を走らせると interrupt が 98% 程度まで上昇していた。このため反応が非常に悪くなり、シェルも満足に使える状況ではなかったことも付け加えておく。

Multiword DMA mode 2 での動作時には dd if=/dev/ad0 of=/dev/null bs=1m にて 15MBytes/sec 以上の速度を達成することができた。

以下は実行結果である。

```
enum# atacontrol mode ad0
current mode = WDMA2
enum# dd if=/dev/ad0 of=/dev/null bs=1m
1983+1 records in
1983+1 records out
2079350784 Bytes transferred in 134.402462 secs (15471077 Bytes/sec)
enum#
```

Multi-word DMA mode 2 においては CPU interrupt が 2% 程度と低くおさえられており、シェルなどの応答速度にも全く影響はなかった。

Transcend の資料によると、本来、この CompactFlash の読み出し速度は 20MBytes/sec ほどあるのだが、今回は 15MBytes/sec 程度で頭打ちとなってしまった。

今回用いた Transcend TS2GCF120 について調べてみて気づいたのだが、この 20MBytes/sec という速度は、Multiword DMA mode 4 および PIO mode 6 にて達成される速度のようである。

ATA の仕様には Multiword DMA mode 3,4 および PIO mode 5,6 は含まれていない。このため、 CF-IDE 変換アダプタにより ATA に準拠した IDE ポートに接続した場合、Multiword DMA 2 の 16.6MBytes/sec に律速されてしまうようである。

ちなみに、書き込み速度についても簡単に調べてみた。実行結果を以下に示す。

```
#atacontrol mode ad0
current mode = WDMA2
# dd if=/dev/zero of=/dev/ad0 bs=1m
dd: /dev/ad0: short write on character device
dd: /dev/ad0: end of device
1984+0 records in
1983+1 records out
2079350784 bytes transferred in 182.367866 secs (11401958 bytes/sec)
```

書き込み速度の公称値は 18MBytes/sec であるので、若干不満が残る。しかし、それなりに妥当な数字といえよう。

7 まとめ

ルータなどの用途としては十分で不自由のない環境が構築できたと思う。

しかしながら、長期間の運用を行ったわけではないので、信頼性などの点については慎重に検討する必要がある。

ディスク半径が小さく低速なものが多いノートパソコンのハードディスクにおいては、シーケンシャルリードについても CompactFlash に置き換えることにより高速化すると考えられる。

また、 NAND 型フラッシュメモリはランダムアクセスが遅いとはいえ、ハードディスクと比較した場合には圧倒的に高速なランダムアクセスを行うことができる。

SanDisk などの CompactFlash は Ultra DMA まで対応し、今回使用した TS2GCF120 よりはるかに高速との話もあるので、パフォーマンスを求めて ComapctFlash に換装するといった手法もありえるだろう。

時間的な制約から行えなかった、CompactFlash への書き込み回数の検証が今後への課題である。