PCF : Portable Compiled Format

1 はじめに

X Window System では, ラスタフォントは BDF (Bitmap Distribution Format) で配布されています. X サーバは PCF (Portable Compiled Format) を利用しますので, bdftopcf というプログラムで変換する必要があります.

BDF については, X のソースツリーの /xc/doc/hardcopy/BDF/ に文書がありフォーマットが規定されていますが, PCF ファイルのフォーマットは見当たりません. ここでは, X Version 11, Release 6 の /xc/lib/font/bitmap/pcfread.c, pcfwrite.c などから分かった, PCF ファイルのフォーマットについて説明します.

2 定義

最初に, いくつか定義をしておきます.

2.1 型

PCF ファイル中にはさまざまな値が入っていますが, ここではそれを C 言語風な記述で書いていきます. その時に型は, 大きさとエンディアンをはっきりわからないものは $TypeSize^{Endien}$ と書き表すことにします. 例えば以下のようになります.

型	*Type*	*Size*(bits)	*Endien*
$char8$	char	8	なし
$bool8$	bool	8	なし
$int32^{little}$	long	32	little
$type32^{little}$	enum type32	32	little
$format32^{little}$	struct format32	32	little

2.2 セクション

PCF ファイルは幾つかの部分に分かれています. その部分部分を**セクション**と呼ぶことにします[1].

各セクションの最初には, $format32^{little}$ 型の値 **format** が入っており, これは次のように定義されます.

[1] X Consortium がどんな名前で呼んでいるかは知らないが, ここではセクションと呼ぶことにする

2.3 $format32$

struct $format32$ [2] {

$uint32$	**id**	:24;	// 下記の 4 つのうちどれか
$uint32$	**dummy**	:2;	// = 0; padding
$uint32$	**scan**	:2;	// (1<<**scan**) バイトごとに bitmap を読む
$uint32$	**bit**	:1;	// 0:LSB first, 1:MSB first
$uint32$	**byte**	:1;	// 0:Little Endien, 1:Big Endien
$uint32$	**glyph**	:2;	// グリフの 1 ラインは (1<<**glyph**) バイトアライン

};

各セクションの中で特に断りなく $int32$ などが出てきた場合は, 各セクションの最初の **format** のメンバ **byte** を参照して, エンディアンを決めます.

上記のメンバ **id** の値は以下のいずれかになります.

enum {

PCF_DEFAULT_FORMAT	= 0,
PCF_INKBOUNDS	= 2,
PCF_ACCEL_W_INKBOUNDS	= 1,
PCF_COMPRESSED_METRICS	= 1,

};

3 ファイルフォーマット

以下の順番でデータが入っています. 以下の順番を入れ換えることはできません.

3.1 Table of Contents

$char8$	**version**[4] = { 1, 'f', 'c', 'p' };	
$int32^{little}$	**nTables**;	
struct {		
$type32^{little}$	**type**;	// セクションの ID
$format32^{little}$	**format**;	// セクションのフォーマット
$int32^{little}$	**size**;	// セクションのサイズ
$int32^{little}$	**offset**;	// セクションのファイル先頭からのオフセット
}	**tables**[**nTables**];	

ファイルの先頭は, PCF のマジックである **version** と, セクションの情報が格納されています. 各セクションの種類は **type** で区別され, 以下のような種類があります.

[2] ビットフィールドは, 上のメンバから MSB, 下のメンバほど LSB になるように配置されるとします

```
enum type32 {
    PCF_PROPERTIES          = (1<<0),
    PCF_ACCELERATORS        = (1<<1),
    PCF_METRICS             = (1<<2),
    PCF_BITMAPS             = (1<<3),
    PCF_INK_METRICS         = (1<<4),
    PCF_BDF_ ENCODINGS      = (1<<5),
    PCF_SWIDTHS             = (1<<6),
    PCF_GLYPH_NAMES         = (1<<7),
    PCF_BDF_ACCELERATORS    = (1<<8),
};
```

3.2 セクション:Properties

このセクションの種類は **tables**[i].**type** = PCF_PROPERTIES です.

- $format32^{little}$ **format**; // **format**.**id** = PCF_DEFAULT_FORMAT
- $int32$ **nProps**;
- struct {
- $int32$ **name**;
- $bool8$ **isStringProp**;
- $int32$ **value**;
- } **props**[**nProps**];
- $byte8$ **dummy**[n]; // n = 3 - ((sizeof(props) + 3) % 4); padding
- $int32$ **stringSize**;
- $char8$ **string**[**stringSize**];

このセクションには, フォントのプロパティが格納されています. 主に, BDF ファイルの STARTPROPERTIES と ENDPROPERTIES で囲まれた部分の値になっています.

props[i].**name** は, **string** からのオフセットで, ($char8$ *)(**string** + **props**[i].**name**) がそのプロパティの名前になります. また, **props**[i].**isStringProp** が真ならば, **props**[i].**value** も文字列を指しているので, ($char8$ *)(**string** + **props**[i].**value**) がそのプロパティの値となります. それぞれの文字列の終端には, '\0' が含まれています.

どのような種類のプロパティがあるか, またどのような種類のプロパティが必須であるかなどは, X Logical Font Description Conventions (/xc/doc/hardcopy/XLFD/) を参照してください.

3.3 セクション:Accelerators

このセクションの種類は **tables**[i].**type** = PCF_ACCELERATORS です. もし 3.10 セクション: BDF Accelerators (PCF_BDF_ACCELERATORS) が存在するならば, このセクションは無視されますので, その場合は省略可能です. PCF ファイルを作成する場合は, このセクションではなく BDF Accelerators セクションを使うことが推奨されています[3].

- $format32^{little}$ **format**; // **format**.**id** = PCF_DEFAULT_FORMAT
- // or PCF_ACCEL_W_INKBOUNDS
- $bool8$ **noOverlap**;
- $bool8$ **constantMetrics**;
- $bool8$ **terminalFont**;
- $bool8$ **constantWidth**;
- $bool8$ **inkInside**;
- $bool8$ **inkMetrics**;
- $bool8$ **drawDirection**;
- $bool8$ **dummy**; // padding
- $int32$ **fontAscent**;
- $int32$ **fontDescent**;
- $int32$ **maxOverlap**;
- $metric\_t$ **minBounds**;
- $metric\_t$ **maxBounds**;
- #if **format**.**id** == PCF_ACCEL_W_INKBOUNDS
- $metric\_t$ **ink_minBounds**;
- $metric\_t$ **ink_maxBounds**;
- #endif

$metric\_t$ については 3.4 セクション:Metrics を参照してください.

このセクションにはフォント全体についての幾つかの特徴が記されています. 詳しくは以下のようになります.

noOverlap true if:
max_i(**metrics**[i].**rightSideBearing** - **metrics**[i].**characterWidth**)
<= **minbounds**.**leftSideBearing**

constantMetrics true if: 全ての文字のメトリック情報が同じ

terminalFont true if:
$\forall i$ (**constantMetrics** &&
metrics[i].**leftSideBearing** == 0 &&
metrics[i].**rightSideBearing** == **metrics**[i].**characterWidth** &&
metrics[i].**ascent** == **fontAscent** &&
metrics[i].**descent** == **fontDescent**)

[3] しかしなにが違うのかよくわからん…

constantWidth true if:
minbounds.**characterWidth** == **maxbounds**.**characterWidth**

inkInside true if:
$\forall i$ (0 <= **metrics**[i].**leftSideBearing** &&
metrics[i].**rightSideBearing** <= **metrics**[i].**characterWidth** &&
-**fontDescent** <= **metrics**[i].**ascent** <= **fontAscent** &&
-**fontAscent** <= **metrics**[i].**descent** <= **fontDescent**)

inkMetrics true if: Ink Metrics != Metrics

drawDirection true if: 右から左, false if: 左から右

fontAscent フォント全体の Ascent

fontDescent フォント全体の Descent

maxOverlap 最も大きい重なり

minBounds 文字のメトリック情報で各々の最小のもの

maxBounds 文字のメトリック情報で各々の最大のもの

ink_minBounds 文字のドットがある部分のメトリック情報で各々の最大のもの

ink_maxBounds 文字のドットがある部分のメトリック情報で各々の最小のもの

3.4 セクション:Metrics

このセクションの種類は **tables**[i].**type** = PCF_METRICS です.

- $format32^{little}$ **format**; // **format**.**id** = PCF_DEFAULT_FORMAT
- // or PCF_COMPRESSED_METRICS
- #if **format**.**id** == PCF_DEFAULT_FORMAT
- $int32$ **nMetrics**;
- $metric\_t$ **metrics**[**nMetrics**];
- #else // if **format**.**id** == PCF_COMPRESSED_METRICS
- $int16$ **nMetrics**;
- $compressedMetric\_t$ **cmetrics**[**nMetrics**];
- #endif

$metric\_t$ は以下のように定義されています (図 1).

```
struct metric_t {
    int16   leftSideBearing;
    int16   rightSideBearing;
    int16   characterWidth;
    int16   ascent;
    int16   descent;
    int16   attributes;
};
```

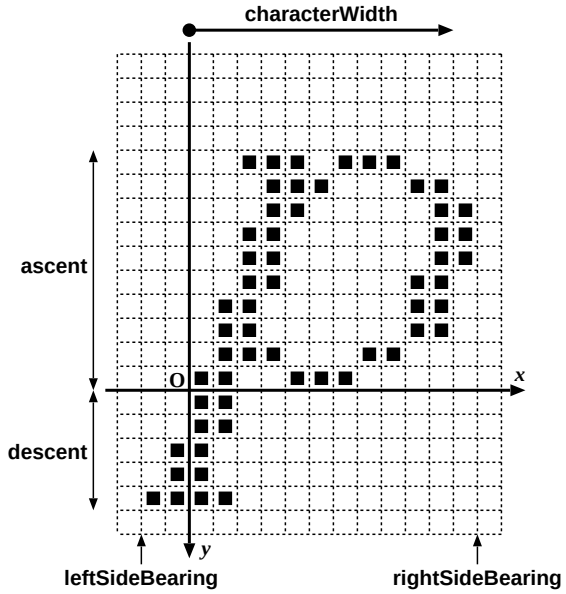


図 1: $metric\_t$

O を原点として, **leftSideBearing** はグリフの左端の x 座標, 逆に **rightSideBearing** はグリフの右端の x 座標. **ascent** はベースライン (x 軸) より上にある部分のグリフの高さ, 逆に **descent** は下にある部分の高さ. $\overrightarrow{nextChar} = (\textbf{characterWidth}, 0)$ は, 現在の文字の原点 **O** から, 次の文字の原点までのベクトルとなります.

$metric\_t$ m と BDF ファイルの関係は次のようになっています.

BBX	BBw BBh BBox BBoy
ATTRIBUTES	m.**attributes**

ただし,

BBw	=	m.**rightSideBearing** - m.**leftSideBearing**
BBh	=	m.**ascent** + m.**descent**
BBox	=	m.**leftSideBearing**
BBoy	=	- m.**descent**

$compressedMetric\_t$ は以下のように定義されています.

struct $compressedMetric\_t$ {

$uint8$ **leftSideBearing**;
$uint8$ **rightSideBearing**;
$uint8$ **characterWidth**;
$uint8$ **ascent**;
$uint8$ **descent**;
};

$compressedMetric\_t$ cm は以下のようにすれば $metric\_t$ m に変換することができますので, **cmetrics** は **metrics** と同一視できます (この文章のなかで **metrics** を参照している部分があっても, PCF ファイルによっては **cmetrics** を参照していることになるので注意してください).

m.**leftSideBearing**	=	($int16$)cm.**leftSideBearing**	- 0x80;
m.**rightSideBearing**	=	($int16$)cm.**rightSideBearing**	- 0x80;
m.**characterWidth**	=	($int16$)cm.**characterWidth**	- 0x80;
m.**ascent**	=	($int16$)cm.**ascent**	- 0x80;
m.**descent**	=	($int16$)cm.**descent**	- 0x80;
m.**attributes**	=	0;	

3.5 セクション:Bitmaps

このセクションの種類は **tables**[i].**type** = PCF_BITMAPS です.

- $format32^{little}$ **format**; // **format**.**id** = PCF_DEFAULT_FORMAT
- $int32$ **nBitmaps** = **nMetrics**;
- $uint32$ **bitmapOffsets**[**nBitmaps**];
- $uint32$ **bitmapSizes**[GLYPHPADOPTIONS];
- $byte8$ **bitmaps**[**bitmapSizes**[**format**.**glyph**]];

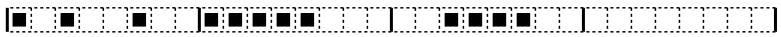
GLYPHPADOPTIONS は以下のように定義されています.

```
#define GLYPHPADOPTIONS 4
```

さて, **metrics**[i] に対応する文字のグリフはビットマップとして ($byte8$ *)(**bitmaps** + **bitmapOffsets**[i]) から始まります.

ビットマップの中の各バイトは, **format**.**bit** にしたがって, LSB first か MSB first でならんでいます. そして, もし, **format**.**byte** != **format**.**bit** ならば, 1<<**format**.**scan** バイト毎にバイト順序が入れ換えられて格納されます. 図 2 は 1<<**format**.**scan** = 4 の時の場合です[4].

[4]実際の例では, Linux の bdftopcf は 1<<**format**.**scan** = 1, **format**.**bit** = 0, **format**.**byte** = 0,



MSB first (Big Endien)：0x A4 F8 3C 00
LSB first (Big Endien)：0x 00 3C 1F 25
MSB first (Little Endien)：0x 00 3C F8 A4
LSB first (Little Endien)：0x 25 1F 3C 00

図 2: グリフの PCF ファイル中でのバイト表現

又, ビットマップ中の横 1 ラインは, 1<<**format**.**glyph** バイト毎にアラインメントされます. 例えば, グリフの幅が 14 ドットの時[5] でも, **format**.**glyph** = 2 (つまり 4 バイト) ならば, 1 ラインは 4 バイト分 (32 ドット) 必要になります.

bitmapSizes[i] にはビットマップ中の横 1 ラインが $1 << i$ バイト毎にアラインメントされた場合の **bitmaps** に必要なバイト数が格納されています.

3.6 セクション:Ink Metrics

このセクションの種類は **tables**[i].**type** = PCF_INK_METRICS です.

このセクションは X サーバにとっては必要ないので, 省略可能です.

このセクションは, 3.4 セクション:Metrics と全く同じ構造をしているので, 説明は省略します.

3.7 セクション:Encodings

このセクションの種類は **tables**[i].**type** = PCF_BDF_ENCODINGS です.

- $format32^{little}$ **format**; // **format**.**id** = PCF_DEFAULT_FORMAT
- $int16$ **firstCol**;
- $int16$ **lastCol**;
- $int16$ **firstRow**;
- $int16$ **lastRow**;
- $int16$ **defaultCh**; // default character or NO_SUCH_CHAR (= -1)
- $int16$ **encodings**[n]; // n = (**lastCol** - **firstCol** + 1) * (**lastRow** - **firstRow** + 1)

このセクションは, 文字コードと PCF ファイル中での文字の出現順序との対応が格納されています.

1<<**format**.**glyph** = 4 で PCF ファイルを作成します

[5] **metrics**[i].**rightSideBearing** - **metrics**[i].**leftSideBearing** = 14

図 3 を見て下さい. これは k14.pcf の例ですが, この場合, n = (0x7E - 0x21 + 1) * (0x74 - 0x21 + 1) = 7896 となり,

$int16$ **encodings**[7896] = {

、,	。,	，,	···,	○,	●,	◎,	◇,
◆,	□,	■,	···	,	-1,	-1,	○,
-1,	-1,		···		,	-1,	-1,
あ,			···			,	-1,
			···				,
齲,			···			,	鬟,
鬟,	鬆,		···		,	鵆,	鵈,
鵝,	鵞,	鵠,	···	,	龕,	龜,	龠,
堯,	槇,	遙,	瑤,	···,	-1,	-1,	-1,	-1,

};

というデータが格納されています (◆ や 龜 は PCF ファイル中でその文字の出現順序です). 例えば, 0x2122 = "、" のメトリック情報は, **metrics**[**encodings**[1]] となります. **encodings**[i] = - 1 の文字コードのメトリック情報やビットマップはこの PCF ファイルには含まれません. 当然, 7896 から -1 の出現回数を引いた数は, k14.pcf の **nMetrics** = 6877 と同じになります.

3.8 セクション:Swidths

このセクションの種類は **tables**[i].**type** = PCF_SWIDTHS です.
このセクションは X サーバにとっては必要ないので, 省略可能です.

- $format32^{little}$ **format**; // **format**.**id** = PCF_DEFAULT_FORMAT
- $int32$ **nSwidths** = **nMetrics**;
- $int32$ **swidths**[**nSwidths**];

BDF ファイルの SWIDTHS で指定された値が格納されています.

3.9 セクション:Glyph Names

このセクションの種類は **tables**[i].**type** = PCF_GLYPH_NAMES です.
このセクションは X サーバにとっては必要ないので, 省略可能です.

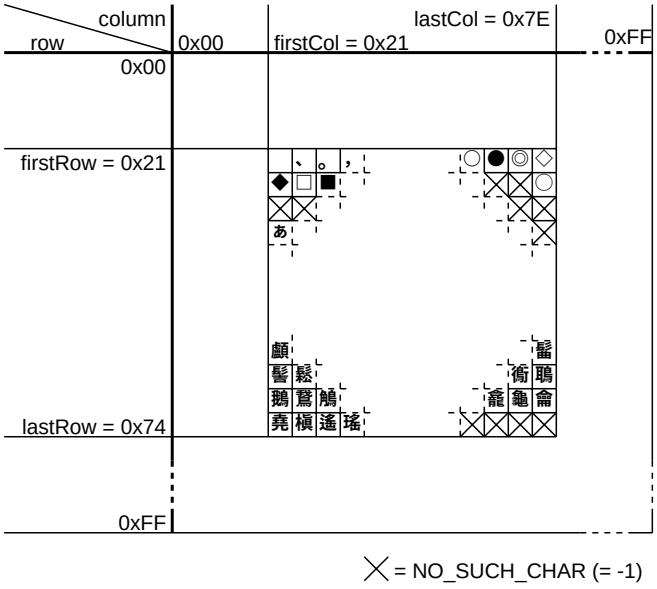


図 3: k14.pcf

- $format32^{little}$ **format**; // **format**.**id** = PCF_DEFAULT_FORMAT
- $int32$ **nGlyphNames** = **nMetrics**;
- $int32$ **glyphNameOffsets**[**nGlyphNames**];
- $int32$ **glyphNamesSize**;
- $char8$ **glyphNames**[**glyphNamesSize**];

このセクションにはフォントの名前が格納されています.

i 番目の文字の名前は, ($char8$ *)(**glyphNames** + **glyphNameOffsets**[i]) になります. それぞれの文字列の終端には, '\0' が含まれています.

3.10 セクション:BDF Accelerators

このセクションの種類は **tables**[i].**type** = PCF_BDF_ACCELERATORS です.

このセクションは, 3.3 セクション:Accelerators と全く同じ構造をしているので, 説明は省略します.

Accelerators セクションが存在する場合は, このセクションは存在しないかもしれません.

4 pcf2bdf

pcf2bdf を付録としてつけておきます. 適宜参照してみてください.

usage: pcf2bdf [-v] [-o $BDFfile$] [$PCFfile$]

-v オプションを使うと, PCF ファイルの中身を見た気になれるかもしれません. $PCFFile$ には圧縮されたものも指定できます (内部で gzip を呼び出しますので, gzip がパスの通ったところに必要です). BDF ファイルの FONTBOUNDINGBOX は, bdftopcf では参照されないので, pcf2bdf では結構いいかげんな値を出力しています. gcc と Visual C++ 5.0 用の Makefile をつけておきます.

5 おわりに

修正歓迎. 追加歓迎. 連絡は nayuta@is.s.u-tokyo.ac.jp まで.

Version 1.00	1998/03/15(日)	by TAGA Nayuta
Version 1.01	1998/03/21(土)	pcf2bdf の Accelerators 周りのバグフィックス pcf2bdf が gzip を呼ぶように改良
Version 1.02	1998/04/11(土)	Bitmaps セクションの種類が PCF_METRICS になっていたのを訂正 Accelerators セクションの未完成だった部分を書いた GLYPHPADOPTIONS は必ず 4
Version 1.03	1999/03/01(月)	jlatex でコンパイルした

目次