



# バイナリフォーマット仕様書

(7/12/98)

# 目次

1. 概要 .....	3
2. バイナリ構造 .....	4
3. Chunk 仕様 .....	6
3.1 Chunk の種類 .....	6
3.2 POF0 のアルゴリズム .....	7

# 1. 概要

Ninja はアスキーフォーマットとバイナリフォーマットをサポートします。

## データ別拡張子

	バイナリフォーマット拡張子	アスキーフォーマット拡張子
Texlist	.njt	.nat
Model	.njd	.nad
Light	.njl	.nal
Camera	.njc	.nac
Motion	.njm	.nam
Shape Motion	.njs	.nas

## その他の拡張子

	バイナリフォーマット拡張子	アスキーフォーマット拡張子
複数のデータを格納する場合	.nj	.nja
シーンファイル (アスキーのみ)	-	.nsc
リソースファイル (アスキーのみ)	-	.nre

- ・ユーザはコンバート時のオプション指定でアスキーフォーマット、バイナリフォーマットのどちらでも得ることができます。
- ・アスキーフォーマットはアスキーフォーマットローダ (gifts としてソースコード提供中) を使うことでコンパイルなしでロードすること、NjDef.h とともにコンパイルすることで利用できます。
- ・バイナリフォーマットはバイナリフォーマットローダ nuBinary (ライブラリに組み込まれるが gifts としてソースコード提供中) を使うことでコンパイルなしでユーザ malloc 領域にデータをロードできます。
- ・バイナリフォーマットは単一のメモリ領域に必要な構造体データを格納しデータ先頭アドレスを 0 番地としたポインタアドレスをデータ構造に出現するポインタメンバーに埋め込みます。この情報に加えデータの何バイト目にポインタメンバーがいるかを示すオフセット情報を持ちます (ポインタオフセット (POF))。
- ・データおよび POF は独立した iff チャンクとして定義されファイルに連続して配置されます。
- ・ローダはデータチャンクのバイトサイズだけのバッファをメモリをアロケートしここにデータを読み込みます。次に POF で与えられるポインタオフセットにアロケートされたメモリの先頭アドレスを足し込みます。これによりデータ内でのポインタのチェーンは正しく実アドレスを示すことになります。
- ・POF はオフセット番号の集合データですが単純にこれを格納するとデータ量が大きくなるため簡単な圧縮をします。現在提供している手法をチャンク POF 0 と呼びます。今後より効率のよい圧縮方法の適用、データサイズよりも展開速度の速い方法など必要に応じて POF 1、POF 2 などを必要であれば拡張していきます。POF の圧縮アルゴリズムが複数あってもローダでチャンクネームとして処理を分岐して展開するため互換を保ちつつかつ古いデータも問題なく展開することができます。
- ・データチャンクのすぐ後ろに格納される POF チャンクを消した場合データのロードはできません。かならずデータと POF チャンクはセットで扱ってください。
- ・バイナリの先頭アドレスにはデータの先頭データがくることを保証します。例えばモデルではモデルツリーのトップのオブジェクト構造体ポインタ、texlist では texlist ポインタ、モーションでは motion 構造体ポインタ。ユーザは各データのポインタをフリーすることでバイナリデータを破棄できます。
- ・モーションにおいてバイナリフォーマットでは action 構造体の出力はしません。別ファイルに格納されたモデルへのポインタアドレスの解決ができないためです。カメラ、ライトのモーションも同様です。
- ・絶対アドレスのバイナリをサポートします。ユーザに指定されたアドレス値をあらかじめ足し込みます。これは速度優先のストリーム再生などに利用します。特殊なバイナリなため取り扱いには注意が必要です。
- ・NJACnv というコンバータでアスキーからバイナリフォーマットへコンバートをサポートします。

## 2. バイナリ構造

### バイナリ構造

Chunkname	bytesize
0 番地からのポインタアドレスを設定したバイナリデータ	
Chunk POF0	bytesize
ポインタオフセットデータ	

#### データバイナリチャンク

##### < チャンクの種類 >

'NJCM': ninja chunk model tree  
'NJBm': ninja basic model tree  
'NJTL': ninja texlist  
'NJLI': ninja light  
'NJCA': ninja camera  
'NMDM': ninja model motion  
'NLIM': ninja light motion  
'NCAM': ninja camera motion  
'NSSM': ninja simple shape motion

#### POF0 チャンク

ポインタオフセットアルゴリズムタイプ 0

モデルデータチャンクと POF チャンクを離してはならない。このペアがセットであれば同一ファイルに複数のデータを格納することもできる。

現在デフォルト出力における texlist 付き Chunk Model の例を以下に示す。

(例) 拡張子.nj

'NJTL'	bytesize
texlist データ	
Chunk POF0	bytesize
texlist POF データ	
'NJCM'	bytesize
Chunk Model データ	
Chunk POF0	bytesize
Chunk Model POF データ	

#### texlist データ

#### Chunk Model データ

絶対アドレスバイナリ構造

'NJAD'	4 (bytesize)	ユーザ指定アドレスを示す
ユーザ指定アドレス		
Chunkname	bytesize	データバイナリチャンク
ユーザ指定番地からのポインタアドレスを設定したバイナリデータ		<div>&lt;チャンクの種類&gt; 'NJCM': ninja chunk model tree 'NJBm': ninja basic model tree 'NJTL': ninja texlist 'NJLI': ninja light 'NJCA': ninja camera 'NMDM': ninja model motion 'NLIM': ninja light motion 'NCAM': ninja camera motion 'NSSM': ninja simple shape motion</div>

NJAD チャンクとデータチャンクは離してはならない。ただしユーザ責任においてアドレスをあらかじめ知っている場合ははずすこと可能。先頭に NJAD を一つとその後ろに複数の同一種類のバイナリデータが格納することも可能。これはストリーム再生などで利用する。

## 3. Chunk 仕様

### 3.1 Chunk の種類

chunk name	説明
'NJCM'	Chunk Model データを格納します。データポインタの先頭にはトップの NJS_CNKOBJECT 構造体のアドレスが格納されます。
'NJBM'	Basic Model データを格納します。データポインタの先頭にはトップの NJS_OBJECT 構造体のアドレスが格納されます。
'NJTL'	texlist データを格納します。データポインタの先頭には NJS_TEXLIST 構造体の先頭アドレスが格納されます。
'NJLI'	ライトデータを格納します。データポインタの先頭には NJS_LIGHT 構造体の先頭アドレスが格納されます。
'NJCA'	カメラデータを格納します。データポインタの先頭には NJS_CAMERA 構造体の先頭アドレスが格納されます。
'NMDM'	モデルモーションデータを格納します。データポインタの先頭には NJS_MOTION 構造体の先頭アドレスが格納されます。NJS_ACTION データは含まれません。
'NLIM'	ライトモーションデータを格納します。データポインタの先頭には NJS_MOTION 構造体の先頭アドレスが格納されます。NJS_LACTION データは含まれません。
'NCAM'	カメラモーションデータを格納します。データポインタの先頭には NJS_MOTION 構造体の先頭アドレスが格納されます。NJS_CACTION データは含まれません。
'NSSM'	シェイプモーションデータを格納します。データポインタの先頭には NJS_MOTION 構造体の先頭アドレスが格納されます。NJS_ACTION データは含まれません。
'NJAD'	ユーザ指定の絶対アドレスを与えます。これに続くバイナリデータチャンクにはこの絶対アドレスからのポインタデータが格納されています。
'POFO'	ポインタオフセットデータ。バイナリデータをロード時にアロケートされたバッファの実アドレスにデータ内部のポインタ値を更新するためのポインタメンバーのロングワードオフセットを与えます。先頭から順次一つ前のオフセットからの相対オフセットで表現されます。またその値を格納する変数の大きさをオフセット値の大きさにあわせて char ( 1byte ), short ( 2byte ), long ( 4byte ) サイズのを切り替えて格納しデータの表現効率を高めています。

## 3.2 POF0 のアルゴリズム

---

<step1>

バイナリイメージを作成。この時先頭アドレス（0番地）からのロングワードオフセットをリストとして格納。

<step2>

相対値に変換。一つ前のオフセットとの引き算により差分を求める。ポインタは4バイトアライメントであるので4で割り単位をロングワードにする。

<step3>

オフセット値を表現するchar, short, longの上位2ビットにフラグをつけ今のデータをchar, short, longのいずれのサイズの値として処理すればいいかの設定をする。

<step4>

POF0チャンクの書き出し上位2ビットはフラグとするため相対ロングワードオフセットが6ビットの最大値64より小さい値ならばchar値として14ビットの最大値16384より小さければunsigned short値としてそれ以上はunsigned long値として出力する。ここで出力されるshort, longデータはビッグインディアンとする。これにより先頭の1バイトにフラグが格納されフラグによるデータタイプ分岐が可能となる。

以下にPOF0データ書き込みのサンプルソースコードを示す。

```
#define NJ_POF_TYPE_PAD    0x00
#define NJ_POF_TYPE_CHAR  0x40
#define NJ_POF_TYPE_SHORT 0x80
#define NJ_POF_TYPE_LONG  0xc0
#define NJ_POF_CHAR_MASK  0xc0

void njPointerCashFlashType0(unsigned long prev, unsigned long current)
{
    unsigned long loffset=(current-prev)>>2;
    if (64 > loffset) {
        char ctmp = NJ_POF_TYPE_CHAR|(char)loffset;
        WriteBytes(&ctmp, sizeof(ctmp));
    } else if (16384 > loffset) {
        unsigned short stmp = (NJ_POF_TYPE_SHORT << 8)|(short)loffset;
        S_SWAP_PC(stmp, stmp); /* keep char order */
        WriteBytes(&stmp, sizeof(stmp));
    } else {
        unsigned long ltmp = (NJ_POF_TYPE_LONG << 24)|loffset;
        L_SWAP_PC(ltmp, ltmp); /* keep char order */
        WriteBytes(&ltmp, sizeof(ltmp));
    }
}
```

次のデータチャンクを4バイトアライメント調整するためPOFチャンクの最後には最大3バイトのパディングが入る。パディングフラグは上位2ビットが00で示されるためPOF内部ではcharで0を書き

込んでおけばパディングされる。

以 上