This document describes the ufo file format (UFF).

http://jdict.sf.net/ufolib

Contents

---

1 Preface

1.1 Encoding

The encoding system of UFF is **Unicode**.

1.2 Naming the PDB

The naming convention of ufo file format PDBs is:

`UFF`<*name*><*psize*><*slant*>

with
<*name*> is the short name, e.g. "gnuuni", "myarial"
<*psize*> is the point size, e.g. "10", "12"
<*slant*> is "n" for normal font; "b", "i", "bi" for bold, italic, bold-italic.

The catalog's creator ID is "`UFFo`", and the type is "`DATA`".

So, eg. the font "genXX" in point size 12, normal font, is named "`UFFgenXX12n.pdb`" or "`UFFgenXX12n.UFFo.DATA`" internally.

2 The PDB format

The UFF consists of one *common part* and several *range parts*. Every part is a record in the PDB.

The font is divided in continuous encoding ranges. So, every range has a start and end code, with no gaps in between. It might be required to divide code ranges as of the maximum size of one PDB record (about 64k). It might be useful to divide code ranges due to performance and memory reasons, because the ranges will be loaded and cached separately. It might even be useful to combine ranges (i.e. add dummy glyphs to get continuous range) to decrease the number of ranges (if the glyph positions are very fragmented).

2.1 Common Part

It starts with:

```
short magic;            // magic code: this is UFF
```

and its currently 12003, witch encodes UFF and version of UFF. Then the font characteristics

```
String fcName;          // is the short name, e.g.
                        //   "gnuuni", "myarial"
short fcSize;           // is the point size, e.g. 10, 12
short fcSlant;          // is 0 for normal font; 1, 2, 3
                        //   for bold, italic, bold-italic
```

and

```
String comment;         // comment
```

where font description, comments, copyright notice etc. can be placed.

The following follows the header in a normal font PDB.

```
short fontType;         // font type 0x9010
short maxWidth;         // maximum character width
short kernMax;          // negative of maximum character kern
short nDescent;         // negative of descent
short fRectWidth;       // width of font rectangle
short fRectHeight;      // height of font rectangle
short ascent;           // ascent
short descent;          // descent
short leading;          // leading
```

Then, we need the information where to find the range parts:

```
short nParts;           // number of parts following (without MCS)
```

The next is coming nParts times:

```
short recNo;            // record number, where is the range?
int firstChar;          // encoding of first char in this range
int lastChar;           // encoding of last char in this range
```

Requirements: firstChar must be less or equal than lastChar, firstChar of the next range in greater than lastChar.

Last is the entry for the unknown char (MCS), this is a small range ;-).

```
short recNo;            // record number, where to find this char
```

2.2 Range Part

One range part contains the bitmaps for one contiguous encoding range of chars.

First the pixel data:

```
int rowByte;            // row width in bytes
```

the row size in bytes, and

```
byte[] bitmapTable;   // the glyph pixel data
```

its size is fRectHeight $\times$ rowByte.

Then, the bitmap location table, so:

```
int[] bitIndexTable;   // where to find one char in the bitmap soup
```

as standard, one more entry than chars (without the MCS), because the last entry of the table contains the offset to one bit beyond the end of the bit image.

2.3 PDB records

So, resulting PDB records are:

RecNo	Description
0	Common Part
1 . . . n	ranges
n+1	MCS range

3 Resources

- PalmOS Font Structure, `http://www.symbioforge.com/fe/`
- In SuperWaba: `waba.applet.UserFont.java`.
- Unicode: `http://www.unicode.org`