- -

This file, xfonts.doc, describes the interface between Ghostscript and the
routines that access externally supplied font and text facilities.

For an overview of Ghostscript and a list of the documentation files, see
README.

****
******** Introduction ********
****

Starting with release 2.6, Ghostscript has the ability to use the
character rasterizer provided by the underlying operating system and
window system; specifically, Adobe Type Manager or a TrueType rasterizer
under MS Windows, and the facilities provided by X Windows. This ability
augments, but does not replace, Ghostscript's own Type 1 rasterizer:
Ghostscript may still use its own rasterizer for very large characters,
characters that are clipped or transformed in unusual ways, and output to
devices other than the screen.

Ghostscript interfaces to these platform facilities through a driver-like
interface called the xfont (external font) interface. Currently, xfont
implementations are associated directly with device drivers; in a future
release, Ghostscript may separate them, so that (for example) it will be
possible to use the platform rasterizer when writing to a file.

Please note that beyond this point, this file is likely to be useful only
to a small number of Ghostscript porters and implementors.

****
******** Types ********
****

gs_char (defined in gsccode.h)

      This type represents a character code that appears in a string.
Currently it is always a single byte, but composite fonts or Unicode may
require it to be wider in the future.

gs_glyph (defined in gsccode.h)

This type represents a character name like 'period' or 'epsilon'. From the xfont implementation's point of view, it is just a handle; when necessary, Ghostscript provides a gs_proc_glyph_name_t procedure (see next type) to convert it to a string name.

gs_proc_glyph_name_t (defined in gsccode.h)

This type represents a procedure that maps a gs_glyph to its string name; see the description of char_glyph below.

gx_xglyph (defined in gsxfont.h)

This type represents a character or glyph code that can be used with a specific platform font. Normally it will be a character code that the implementation of render_char will turn into a 1-character string and give to the platform's "display string" operation.

gx_xfont_procs (declared in gsxfont.h, defined in gxxfont.h)

This type is the xfont analogue of gx_device_procs, the type of the procedure record that defines an xfont implementation.

gx_xfont (declared in gsxfont.h, defined in gxxfont.h)

This type is the gxfont analogue of gx_device, the type of the basic structure for an xfont.

(encoding_index)

This is not really a type, although it probably should be: it is an int used to indicate the Encoding used by a font. Defined values are

0 = StandardEncoding
1 = ISOLatin1Encoding
2 = SymbolEncoding
3 = DingbatsEncoding
-1 = other encoding

****
******** Implementation procedures ********
****

All the procedures that return int results return 0 on success, or an appropriate negative error code in the case of error conditions. The error codes are defined in gserrors.h. The relevant ones are the same as for drivers (see drivers.doc for details).

As for driverse, if an implementation procedure returns an error, it should use the return_error macro rather than a simple return statement, e.g.,

```
return_error(gs_error_VMerror);
```

This macro is defined in gx.h, which is automatically included by gdevprn.h but not by gserrors.h.

Font-level

```
gx_xfont *(*lookup_font)(P7(gx_device *dev, const byte *fname, uint len,
  int encoding_index, const gs_uid *puid, const gs_matrix *pmat,
  const gs_memory_procs *mprocs))
```

Look up a font name, UniqueID, and matrix, and return an xfont, or NULL if no suitable xfont exists. Use mprocs to allocate the xfont and any subsidiary data structures. The matrix is the FontMatrix concatenated with the CTM, so (roughly speaking) the font size in pixels is pmat->yy * 1000 for a normal Type 1 font.

Note that this is the only implementation procedure that does not take an xfont * as its first argument. In fact, callers of lookup_font must use the get_xfont_device driver procedure to get the correct device to pass as the first argument to lookup_font.

```
gx_xglyph (*char_xglyph)(P5(gx_xfont *xf, gs_char chr, int encoding_index,
  gs_glyph glyph, gs_proc_glyph_name_t glyph_name))
```

Convert a character name to an xglyph code. In the case of glyphshow, chr may be gs_no_char; for an ordinary show operation, if the character code is invalid, glyph may be gs_no_glyph.

```
int (*char_metrics)(P5(gx_xfont *xf, gx_xglyph xg, int wmode,
  gs_int_point *pwidth, gs_int_rect *pbbox))
```

Get the metrics for a character.

```
int (*render_char)(P7(gx_xfont *xf, gx_xglyph xg, gx_device *target,
  int x, int y, gx_color_index color, int required))
```

Render a character. (x,y) corresponds to the character origin. The target may be any Ghostscript device. A good implementation will check whether the target can handle this type of xfont directly (e.g., by checking the target name), and if so, will render the character directly; otherwise, it will do what has to be done in the general case, namely, get a bitmap for the character and use the target's copy_mono operation. If required is false, the procedure should return an error if the rendering operation would be expensive, since in this case Ghostscript has already cached the bitmap and metrics from a previous call with required=true.

```
int (*release)(P2(gx_xfont *xf, const gs_memory_procs *mprocs))
```

Release any external resources associated with an xfont. If mprocs is not NULL, also free any storage allocated by lookup_font (including the xfont itself).