T1Lib

Version 5.1.0 – Mai, 18, 2005

A Library for Generating Character Bitmaps from Adobe Type 1 Fonts

Rainer Menzner

Contents

1 Introduction

1.1 What Does t1lib Do?

t1lib is a library written in the C programming language allowing a programmer to generate bitmaps from Adobe (TM) Type 1 fonts quite easily. These bitmaps are returned in a data structure with type GLYPH. This special GLYPH-type is also used in the X11 window system to describe character bitmaps. It contains the bitmap data as well as some metric information. But t1lib is in itself entirely independent of the X11-system or any other graphical user interface.

Given that the X11-system is probably the most frequently used window system in the UNIX-world, and furthermore assuming that most graphical applications run under window systems, it appears that t1lib implements functionality already provided by the graphical user interface, the X11-system. Thus the question arises: Why not use X11 directly for rastering characters? Well, the answer is quite simple; the X11 Font machinery appears to be too *static* in order to use it for certain purposes. Moreover X11-calculations and positioning of characters are based on bitmap dimensions and are thus subject to error accumulation.

Here is a list of features which are supported in the current release of t1lib.

- Rasterizing is done as characters get requested. The X11-server, in contrast, rasters a font completely at a given size when it is loaded. *Rastering on demand* saves time when the font is loaded the first time and saves memory since often only the alphabetic letters and a few other characters are needed.

- The encoding mechanism of PostScript is fully supported. The user may use the fonts' internal or some other encoding. Additional encoding files can be loaded at runtime and fonts can be reencoded at runtime. The syntax of encoding files is straight-forward and simple.

- The library makes use of Adobe Font Metric data in the form of AFM files. This may seem a disadvantage, but in order to make decent typesetting possible, some more information than that contained in Type 1 font files is needed. Besides, Adobe makes these AFM files freely available on their ftp-server for all registered Type 1 fonts.

- In case AFM files are missing, t1lib is able to generate metrics information in charspace accuracy by rastering each character at 1000 bp.

- In addition to generation of character bitmaps, there is a way to directly raster strings of any length in a given font. The space-width may explicitly be corrected by the user. This may be needed by word processing applications.

- Strings may be rastered optionally using pairwise kerning information from the AFM file. Kerning is an important feature of good quality typesetting. Kerning information can also be requested by the user without rasterization.

- Ligature information is made available to the user in an efficient way. Use of Ligatures is another characteristic of good typesetting. As to my knowledge, only TeX and all related macro-packages are able to handle ligatures in a natural and efficient way.

- Rotation and arbitrary transformation on the fly is supported.

- t1lib supports antialiasing. In this case, a pixel is represented as a byte, word or double word. Antialiasing is implemented by subsampling with factor 2 or 4 alternatively. If you use the X11-interface introduced in t1lib V. 0.3-beta, even colored antialiasing between any pair of colors is provided in a completely transparent way.

- In addition to transformation on the fly, two transformation types—*slanting* and *extending*—are possible on the fontlevel including bitmap caching. Horizontal expansion of fonts is fully supported and since version 0.3-beta also *slanting* of fonts is nearly fully supported. For restrictions and drawbacks of slanting fonts see 5.11 on page 51.

- Paths, the library searches for the different needed file types are specified at runtime by means of a configuration file. They may thus be changed without needing to recompile the application. For example, the directories of the X11-system's Type 1 font files may be specified there in order to use these fonts with the library. A user may have his own configuration file and as a fallback/default there is a system wide configuration file. This should be setup when the library is installed.

- Since of version 0.3-beta a special set of functions is provided which implements a more comfortable X11-interface. This is due to the fact that X11 is the only standard window system in the UNIX world. However, as before `t1lib` may be compiled and used without even having X11 installed.

- Some decorations like *underlining* are supported by simply setting a flag for the rastering functions.

- *Right To Left* typesetting is supported.

- Font subsetting is easily achieved using a high level function. This makes it easier for application to efficiently export Postscript files.

- Composite character information can be retrieved. Moreover, `t1lib` transparently handles and realizes composite character definitions from AFM files.

There are also some problems and features not yet implemented, but likely to be implemented in the future. The main problem up to now is:

- The font cache isn't a font cache really. At this time all bitmaps are saved by always allocating more memory from the system. No automatic removal of bitmaps no longer needed is done. However, the user has the possibility of explicitly removing data, if he thinks it is not needed anymore.

1.2 Copyrights and Credits

There are some copyrights on parts of the library and there are some programmers (or corporations) which I want to give credit. The library uses:

- all internal parts of the X11-rasterizer donated to the X11-project by IBM. This rasterizer does the *real* hard work of scan-conversion.

- the modifications to the rasterizer done by Piet Tutelaers in his ps2pk-package. The main purpose was decoupling the sources from the X11-system sources.

- the `parse_AFM` software which was made freely available by Adobe. This is used to parse the AFM files (what a surprise) and to generate the data structures the information is saved in.

Raph Levien (`raph@acm.org`) contributed an algorithm for sampling down non-antialiased bitmaps to antialiased bitmaps in a very efficient manor. This makes antialiasing a lot faster.

Fred L. Drake, Jr. (`fdrake@acm.org`) wrote a Python interface to `t1lib`, which is distributed with `t1lib`. This wrapper is called `t1python` and allows Python-programmers to use Type 1 fonts. I can not tell anything more on this topic since I do not know the Python language. All questions concerning the Python interface should thus be addressed to Fred L. Drake, Jr.

Evgeny Stambulchik (`fnevgeny@plasma-gate.weizmann.ac.il`), maintainer of the `grace`-project—a descendent of `xmgr`, never gets tired of finding and reporting (and fixing) bugs in `t1lib`. Other members of this project spent time in porting `t1lib` to further systems:

- Ed Vigmond (`vigmonde@IGB.UMontreal.CA`): IRIX-port,
- John Hasstedt (`John.Hasstedt@sunysb.edu`): VMS-port,
- Alexander Mai (`st002279@hrzpub.tu-darmstadt.de`): OS/2-port.

Hirotsugo Kakugawa (`h.kakugawa@computer.org`) added support for GNU `libtool` to `t1lib`.

David Huggins-Daines (`bn711@freenet.carleton.ca`) spent effort in finding memory leaks and maintains a Debian-package of `t1lib`.

Thanks to all these people
and to all those contributors not mentioned here!

1.3 Motivation

The idea of writing this library was due to the SciTeXt-project which was founded in 1996. They needed a font raster system with some more functionality than X11 provided. You may find that some things (for example the format of the font database file) is really closely related to SciTeXt. In February '97 the SciTeXt developers decided to migrate from C to Java and did not need a font rasterizer any more. Since I thought there could be other applications for this library I continued the work on it and voila.

I have removed the history description which followed at this place in some previous versions. The history can be roughly viewed in the file `Changes` which is located in the toplevel directory of the distribution.

1.4 How to Reach the Author/How to Get `t1lib`

If you have questions, comments or bug reports, you can reach me by eMail. The address is

`Rainer.Menzner@web.de`

If you have bug-reports, it would be best if you could reproduce the error using the test program `xglyph` (see 3).

`t1lib` is available by anonymous ftp as

`ftp://sunsite.unc.edu/pub/Linux/libs/graphics/t1lib-`$x.y[.z]$`.tar.gz`

Here, $x.y.z$ is the version identifier and the brackets around z indicate that this entry is optional.

2 Getting Started

2.1 Building, Installing and Removing the t1lib-Package

As of version 0.2-beta, the autoconf-package is used to configure and build the library. imake is no longer supported. Furthermore, starting with version 0.8-beta GNU libtool is used for managing library-specific stuff.

Here is how to build and install t1lib:

1. Change to T1-directory.

2. Run ./configure. This will check your system's setup and generate the Makefiles. By default, shared and static versions of the libraries are built.

   Specifying --disable-shared or --disable-static as a commandline option to configure will suppress the generation of the respective library type. Of course, these rules are superseded by the capability of the system to manage those library types.

   If you know shared libraries are supported on your system but configure says that no dll can be built, some compiler option may be setup incorrect. Please refer to (2.2).

   If the X11 window system is installed on the target system t1lib is automatically build with special X11 support. In cases where this is explicitly not desired the commandline option --without-x may be used to configure a library without extended X11 support. In this case the test program xglyph is also not build since it needs X11.

3. Run make. This will build all the stuff including the documentation. If you do not have LaTeX 2_ε run make without_doc. This will skip generating the documentation.

4. Type make install to install the package. You'll probably need to be superuser for installing the package at the standard locations. However, the files may be located wherever the user wants, as long as the compiler finds them at compile time. So, place them where you want.

   The following files are installed when doing a make install:

   - lib/libt1.a and/or lib/libt1.so.$v.r.p$ if the system supports shared libraries. In the latter case, also two symbolic links to the shared library, libt1.so.v and libt1.so, are generated. Here, v and r mean version and revision of the shared library. p is the patch level. Library and links are installed in the directory specified by the autoconf-variable libdir which is by default /usr/local/lib.
   - The same as above holds for lib/libt1x.a or lib/libt1x.so.$v.r.p$ respectively, which contain the X11 interface functions. This library is only installed if X11 support was possible and not suppressed.
   - lib/t1lib.h and optionally lib/t1libx.h. They are installed in the directory pointed to by the autoconf-variable includedir which is by default /usr/local/include.
   - The test program xglyph/xglyph. If shared libraries are supported (and not suppressed by --with-static-lib) this executable is dynamically linked to libt1.so and libt1x.so. It is installed in the directory pointed to by the autoconf-variable bindir (by default /usr/local/bin).
   - The converter type1afm. The same applies as above for xglyph.

- A subdirectory named t1lib-*v.r* is created in the directory pointed to by the autoconf-variable datadir (default /usr/local/share) and a default global configuration file t1lib.config is installed there. Note that this configuration is not of any use. It has to be setup by the administrator to specify the paths to the system's Type 1 fonts and AFM files as well as any t1lib encoding files. Notice also that the global configuration file is not installed if it already exists. This is to prevent from deletion of an existent setup.
- A subdirectory doc is created in the directory where the global configuration file resides (see above). The LaTeX 2ε-documentation t1lib_doc.dvi as well as all needed graphics files is installed there. The LaTeX 2ε-sources are not installed!
- If you ever want to remove t1lib from your system this can be achieved by calling make uninstall. This reverts all steps described above. Of course, this works only if t1lib has not been reconfigured using different parameters since the time of install.

The top level Makefile further supports the targets clean and distclean. The latter is an extension of clean which also removes the makefiles as well as the log and cache files of the configuration process. It forces thus a new call to configure.

A make clean is needed, for example, if someone experiments with static and shared libraries since the object files for shared libraries require the additional position independent code options.

The directory T1/parse_afm is not needed at all, it is included only for completeness. The parts needed from this have been copied to the lib/t1lib-subdirectory.

2.2 Notes on Using GNU libtool

libtool might get confused by heterogenous compiler setups. This is the case, for example, on our Solaris system where by default gcc is used in combination with the system specific linker. This configuration leads to libtool reporting that no shared library can be built which definitely is wrong. In most cases such problems can be solved by fiddling with the environment entries CC, CFLAGS, LD and LDFLAGS.

libtool hides the real objects in subdirectories named .libs. This means, after a successful build, libt1.so is located in T1/lib/.libs. Similarly, if shared libraries are built the executable T1/xglyph/xglyph is a simple wrapper to T1/xglyph/.libs/xglyph.

2.3 Runtime-Setup

2.3.1 Searchpath and Environment Setup

t1lib basically needs four types of files:

- .afm-files: These contain font metric descriptions as well as kerning and ligature information for a particular font.
- .pfa-/.pfb-files: These contain the character outline descriptions. Type 1 font files may also lack any extension in their filename. This is the habit on NeXTStep, for example.
- .enc-files: These contain encoding arrays in a special but simple form. They are only needed if someone wants to load a special encoding to reencode a font.
- A font database file. The library needs at least one font database file specification. See below for a description of this font database file. Optionally, multiple font database files can be specified.

In order to tell t1lib where these files are located in the filesystem, a configuration file usually has to be set up by the user. At time of initialization (see 5.3 on page 26) the library tries to locate all data it needs immediately or possibly later. The following actions take place in order:

1. The library tries to read the variable `T1LIB_CONFIG` from the program's environment. The value of this variable is expected to be the pathname of a configuration file for t1lib.

2. If the variable `T1LIB_CONFIG` exists, the file pointed to by this variable will be tried to be opened. In case no environment variable exists, the library will attempt to open a file called `.t1librc` in the user's home directory. If this file as well does not exist, the global configuration file `t1lib.config` is tried to be opened.[1] If all these attempts to open a configuration file did not succeed, all searchpaths are left at defaults (`.`) and the font database file is setup to be `./FontDataBase`. If this file cannot be opened, the call to `T1_InitLib()` returns a NULL-pointer thus indicating an error condition. The program should then exit because t1lib would not be able to do anything without an association of font IDs to font files.

3. Assuming a configuration file has been found and opened at any of the above three locations, this file is parsed and all relevant information in this file is recorded.

4. Using the paths specified in the configuration file, the font database is opened and processed. The existence of every Type 1 file that might later be needed is ensured. The existence of the corresponding AFM file is not verified during initialization, because this information is not ultimatively critical when generating a character bitmap.[2] Aside from this, t1lib can generate the required part of the AFM information on the fly.

2.3.2 The t1lib Configuration File

It is the purpose of the configuration file to setup search paths and font databases. The format of this file is quite simple and straightforward:

- Each line starting exactly with `ENCODING=` is read in. The remainder of the line is expected to be a list of searchpath specifications for encoding files. No white space may appear between `=` and the path specification(s). Multiple paths may be specified by separating the single paths with colons.[3] The path specification(s) may be followed by any white space characters.

- Each line starting exactly with `AFM=` is read in. The remainder of the line is expected to be a list of searchpath specifications for Adobe Font Metric files. No white space may appear between `=` and the path specification(s). Multiple paths may be specified by separating the single paths with colons. The path specification(s) may be followed by any white space characters.

- Each line starting exactly with `TYPE1=` is read in. The remainder of the line is expected to be a list of searchpath specifications for Type 1 font files. No white space may space between `=` and the path specification(s). Multiple paths may be specified by separating the single paths with colons. The path specification(s) may be followed by any white space characters.

[1] The filenames for the user's and the global configuration file as well as the name of the environment entry are default names defined in `lib/t1lib/t1misc.h`. They may be redefined by the user at compile time if necessary.

[2] For example, a program may generate a character table of a Type 1 font without having AFM information.

[3] A colon is the default path separator on UNIX systems. For certain other Operating Systems the path separator may be a semicolon.

- Each line starting exactly with `FONTDATABASE=` must specify a colon-separated list of font database filenames on the remainder of the line. No white space is allowed between `=` and the path specification, but trailing white space is allowed.
- Each line starting exactly with `FONTDATABASEXLFD=` must specify a colon-separated list of font database filenames that adhere to the XLFD (X11 Logical Font Definition) on the remainder of the line. No white space is allowed between `=` and the path specification, but trailing white space is allowed.

  This keyword is not an independent key word, rather, it allows to specify a special alternative type of font database—a type that usually exists as part of every X11 system. Standard and XLFD font database files may be specified both in one configuration file. They interact in a way, that if an XLFD font database file is specified, this overwrites the default standard font database setup before the configuration is read.
- All other lines are ignored by the library.

A configuration file may contain multiple path declarations of one type. In this case, a list of path elements is built internally in the same order that the specifications appear in the configuration file. For example, the statement

```
TYPE1=/usr/X11/fonts/Type1:/home/user/fonts/Type1
```

is equivalent to the two statements

```
TYPE1=/usr/X11/fonts/Type1
TYPE1=/home/user/fonts/Type1
```

In order to specify paths that incorporate *unusual* characters like white space or the path separator, it is possible to quote the path string element using double quotes "`"`". All characters after the leading quotation mark are verbatim read to the path until the closing quotation mark appears. The double quotes may also become part of a path specification by using the escape sequence "`\"`". Hence, the following statement specifies a correct albeit somewhat unusual search path:

```
TYPE1=/usr/X11/fonts/Type1:"/home/user/My \"Best\" Fonts/Type1"
```

Here is an example of how a user could do the runtime setup:

Example:
Create a file, say, named `t1.config` with the following contents in your HOME-directory:

```
# This is a configuration file for t1lib
  These two lines are considered to be comments

FONTDATABASE=/home/user/test/myprog/FontDataBase
ENCODING=/usr/local/lib/fonts/type1/enc:.
AFM=/usr/local/lib/fonts/type1/afm:.
TYPE1=/usr/local/lib/fonts/type1/outlines:.
```

After this, make the environment variable `T1LIB_CONFIG` point to this file, i.e.,
`setenv T1LIB_CONFIG ~/t1.config`
for `tcsh`, or
`export T1LIB_CONFIG=~/t1.config`
for `bash`. Provided that the path specifications in the configuration file are correct, the setup is done. When setting `T1LIB_CONFIG` in an interactive shell as described above, the shell does the tilde expansion. Notice that `t1lib` never does tilde expansion.

2.3.3 The Font Database File

This is one more file important at startup time. I call it "font database file" because it makes fonts declared in this file known to the library. Moreover, the association *declared font* $\Longleftrightarrow$ *FontID* is done using information from this file. The format specification of this file is relatively free. Here are the exact rules:

- Line 1 contains a positive integer specifying the number of fonts declared in that file. This is as in the `fonts.dir` files of the X11-system.

- All remaining lines contain declarations of one font each. The only thing taken from such a line is the last string (delimited by white space) in it. It is assumed to be a filename of the format *basename.someextension*. Furthermore, the *basename*-part is assumed to be the basename of a fontfile name. After such a string has been parsed, the *extension* is cut off and replaced in turn with `.pfa` and `.pfb`. The initialization routine tries to open a font file with one of the resulting filenames.

  Since V. 0.9 the "`.`" as well as the "*someextension*" may be missing completely. Moreover, `t1lib` looks for Type 1 Font files whose name do not have any extension at all. This is due to conventions of some UNIX-systems.

- The remaining of the line, i.e., from beginning to the start of the filename string is completely ignored and may contain some information for other programs or be empty.

The format described above may seem to be underspecified, but it has been chosen to be compatible with the `SciFonts`-fileformat, which is used during the initialization of the SciTeXt word processor.

Example 1: A minimal font database file for 4 fonts:

```
4
isvl.afm
isvli.afm
isvd.afm
isvdi.afm
```

This file is *minimal*, because it contains just the information needed, and nothing not needed by `t1lib`.

Example 2: A more realistic example, which allows an application to match a fully qualified X11 fontname to a FontID in `t1lib`. This is also a valid font database file:

```
4
Souvenir   Souvenir-Light         ---   -itc-souvenir-light-r-normal--#-0-0-0-p-0-iso8859-1  isvl.afm
"          Souvenir-LightItalic   -*-   -itc-souvenir-light-i-normal--#-0-0-0-p-0-iso8859-1  isvli.afm
"          Souvenir-Demi          *--   -itc-souvenir-demi-r-normal--#-0-0-0-p-0-iso8859-1   isvd.afm
"          Souvenir-DemiItalic    **-   -itc-souvenir-demi-i-normal--#-0-0-0-p-0-iso8859-1   isvdi.afm
```

2.3.4 Alternative Runtime Setups

The runtime setup described above is the most simple principle of getting started. However, there might be applications that deal with only one font file. A good example is the `type1afm`-utility which is described in section 4. In such situations it seems to be overkill to read a font database file and several load paths. For this reason there are alternative ways to specify what should be read from where. Their description is deferred to section 5.3.

2.4 A Very Simple Programming Example

The following code is a very simple programming example of how to use `t1lib`. It even runs on an ASCII-terminal. It is provided in the `examples`-subdirectory of the distribution as `t1example1.c`. This program must be compiled to object format and then linked with the library `libt1.a` or `libt1.so`, respectively. On most systems the commandline

```
cc -o t1example1 -I ../lib t1example1.c -L../lib -lt1 -lm
```

should do it. For convenience reasons a `Makefile.in` is included in the examples directory and the stuff is built automatically.

At runtime, a well defined setup must be found, i.e., a configuration file with path definitions and a font database file. These also are located in the `examples` subdirectory.

```
#include <stdio.h>
#include <stdlib.h>
#include <t1lib.h>  /* All needed declarations */

int main( void)
{

  GLYPH *glyph;
  int i;

  /* Set our environment to an existent config file directory */
  putenv( "T1LIB_CONFIG=./t1lib.config");

  /* Pad bitmaps to 16 bits, the default being 8 bits */
  T1_SetBitmapPad( 16);

  /* Initialize t1lib and return if error occurs. No logfile will be
     generated */
  if ((T1_InitLib(NO_LOGFILE)==NULL)){
    fprintf(stderr, "Initialization of t1lib failed\n");
    return(-1);
  }

  /* For every font in the database, generate a glyph for the string
     "Test" at 25 bp. Use Kerning. Then dump an ASCII representation
     of the glyph to stdout */
  for( i=0; i<T1_Get_no_fonts(); i++){
    glyph=T1_SetString( i, "Test", 0, 0, T1_KERNING, 25.0, NULL);
    T1_DumpGlyph( glyph);
  }

  /* Close library and free all data */
  T1_CloseLib();

  return( 0);

}
```

We assume that in the current directory there is a file `FontDataBase` which declares two fonts, Souvenir Light and a bold italic variant and further, that these fonts and their AFM files can be found using the paths from the configuration file. If the resulting program is run, it produces some output like the following on `stdout`:

```
Dataformat: T1_bit=0, T1_byte=1, T1_wordsize=16, T1_pad=16
GlyphInfo: h=18, w=44, paddedW=48
.XXXXXXXXXXXXXX. ................ ................
XXX....XX....XXX ................ ................
X......XX......X ................ ........X.......
.......XX....... ................ ........X.......
.......XX....... ................ .......XX.......
.......XX....... ................ .......XX.......
.......XX....... ...XXXX.......XX XXX..XXXXXXX....
.......XX....... .XX...XX.....XX. ..XX...XX.......
.......XX....... XX.....XX...XX.. ...X...XX.......
.......XX....... XX.....XX...XX.. .......XX.......
.......XX......X X......XX...XXX. .......XX.......
.......XX......X X....XXX.....XXX XXX....XX.......
.......XX......X XXXXXXX.......XX XXXX...XX.......
.......XX......X XX.............. ..XXX..XX.......
.......XX......X X............... ...XX..XX.......
.......XX....... XX......XX..X... ...XX..XX.......
.......XX....... XXX....X....XX.. ..XX...XX.......
.....XXXXXX..... ..XXXXX.......XX XXX....XXXXX....
Dataformat: T1_bit=0, T1_byte=1, T1_wordsize=16, T1_pad=16
GlyphInfo: h=18, w=51, paddedW=64
.XXXXXXXXXXXXXXX X............... ................ ................
.XXXXXXXXXXXXXXX X............... ................ ................
.XX....XXXX....X X............... ...............X X...............
.X.....XXXX..... X............... ...............X X...............
......XXXXX..... ................ ..............XX X...............
......XXXXX..... ................ ..............XX X...............
......XXXX...... ....XXXXXX...... .XXXXX.....XXXXX XXX.............
......XXXX...... ..XXXXXXXXXX...X XXXXXXXX...XXXXX XXX.............
......XXXX...... .XXXX...XXXX..XX XX...XXX....XXXX ................
......XXXX...... XXXX....XXXX..XX XX....XX....XXXX ................
.....XXXXX...... XXXX...XXXX...XX XXXX........XXXX ................
.....XXXXX.....X XXX..XXXXX.....X XXXXXX......XXXX ................
.....XXXX......X XXXXXXXX........ XXXXXXXX....XXXX ................
.....XXXX......X XXX............. ..XXXXXX...XXXX. ................
.....XXXX......X XXX.......X..XX. ....XXXX...XXXX. ................
.....XXXX....... XXXX....XXX.XXXX ....XXXX...XXXXX XX..............
....XXXXXX...... XXXXXXXXXX...XXX XXXXXXX....XXXXX XX..............
...XXXXXXX...... ..XXXXXX.......X XXXXX......XXXX. ................
```

3 The Program xglyph

`xglyph` is a tool which makes most of the functionality of `t1lib` visible to the user without the need of having to write an own program and without the need of having to understand most of the library before. This program—as the name indicates—needs X11. It is thus only build if X11 is installed on the target system and if X11 support has not explicitly been disabled. All necessary resources are set internally to default values so that the program can be run out of the box without any installation.

In case the user did not already create a custom configuration file and an associated font database file, the program should be started from the subdirectory `xglyph` of the distribution. When starting, `xglyph` checks for the environment entry `T1LIB_CONFIG` and if it does not exist it adds the association `T1LIB_CONFIG=./t1lib.config` to the environment. In other words it expects a valid configuration file in the current directory.

There are several widgets which may be categorized into 5 types.

3.1 Common Parameter Dialogs and Toggle Buttons

These buttons modify the internal state of the program by setting some global variables. These variables affect the execution of all rastering functions in contrast to the buttons described in the next subsection which only take influence on the X11 rastering functions. When changing one of the following parameters nothing seems to happen at first. All actions are deferred to the time when an action button is clicked. Here is a list of the dialogs and toggles:

- **FontID**
  This dialog allows to specify the font ID that will be used when the next action takes place. The allowed IDs range from 0 to $n-1$, where n is the number of fonts declared in the font database file. If using the default configuration file together with the default font database file, 8 fonts are declared. If an invalid ID is specified, the next action generates an error message.

- **Font-Size**
  Here, the size of the font is specified. The value is interpreted in bigpoints, the default PostScript unit. If the size specified is invalid, an appropriate error message is generated at the time of the next action.

- **Slant-Factor**
  A slant factor s may be specified. It is interpreted the following way. A point described by the coordinate-pair (x, y) is transformed to the point with the coordinates $(x + sy, y)$. For instance, specifying a slant factor $s = 1$ will generate a font slanted by 45°. Since version 0.3-beta slanted are nearly fully supported. For a discussion of the remaining problems see 5.11 on page 51.

- **Extension-Factor**
  Horizontal extension of a font may be realized using this dialog. The default value is 1 which means the characters are presented at their natural width. Specification of an invalid value will generate an error message at the time of the next action.

- **Transformation-Matrix**
  This dialog gives complete Control over the transformation matrix that will be used in consequent rasterizations. The values have to be specified separated by commas. A specified rotation is still applied after this matrix.

- **Res [DPI]**
  The resolution of the output device (screen) may be specified in this dialog. Using the default value of 72 dpi means one bp in size corresponds to exactly one device pixel.

- **S-Width**
  This dialog field allows to setup the penwidth used for stroking characters. If this field is zero, which is the default, the characters are filled, unless, the font under consideration is a stroked font, that is, it has `PaintType = 1`.

- **Encoding-File**
  The name of an encoding file may be specified. Included in the distribution is only one file, `IsoLatin1.enc`. It contains the standard X11 encoding in a format acceptable by `t1lib`. If no name is given here, or the file with the name given here cannot not be parsed as an encoding file, the encoding is switched back to the fonts internal encoding. Again, this is done at the time of the next action.

- **Angle**
  The angle at which the next character or string is rastered is specified here. There are no restrictions concerning the angle. Rotation is applied after setting the transformation matrix (see above).

- **Space-Off**
  The value specified here represents an offset added to the spacewidth when rastering the next string. For this, it is interpreted in PostScript charspace units and thus subject to scaling.

- **Character**
  A number between 0 and 255 inclusive should be specified here. It is used as the index into the current encoding vector when rasterizing a character. This gives the user access to all currently encoded characters, regardless of the current X11 keyboard mapping. If an index is given whose encoding entry would produce no black pixels, an error message is generated at the next character-rastering time. The default value is 65, which corresponds to the character "A" in most encoding vectors.

- **Test-String**
  In this dialog, a complete string may be specified. It will be rastered when the next string-rastering button is pressed. It can be of arbitrary length (well, almost). If this field is left empty, the standard string "Test" will be used for rastering.

- **Kerning**
  This is a toggle button. Its state determines whether pairwise kerning information from the AFM file will be used to correct the horizontal spacing during string rastering or not. A typical example is the word "Test"; enabling kerning should—at least in fonts of good quality—move the "T" and the "e" significantly closer together.

- **Ligature**
  This is a toggle button. Its state specifies whether the string is checked for ligatures prior to rastering it. Suitable character sequences are replaced with the corresponding ligature. For a good example, you should switch to font ID 4 and type in the string `--difficult---`. If ligature detection is switched on, the two hyphens should be converted to an en-dash "–", the three hyphens should be converted to an em-dash "—" and the character series "`ffi`" should be replaced with the ligature "ﬃ", rather than to be displayed as "ffi".

- $\boxed{| \longrightarrow}$ / $\boxed{\longleftarrow |}$ This button allows to change the writing direction that `t1lib` will use in subsequent calls to the string rastering functions, the default being *Left To Right* as used in most European languages. This item is simply meant to demonstrate the capabilities of `t1lib`. The package does not come with fonts that are intended to be used for *Right To Left* typesetting.

- **Underline**
  This toggle button determines whether strings are underlined or not.

- **Overline**
  Same as above for overlining.

- **Overstrike**
  Same as above for overstriking.

- **AA-Low**/**AA-High**
  This button allows to select the subsampling factor for antialiasing in subsequent rastering operations. *AA-Low* means subsampling by factor 2 which gives 5 gray values including black and white, whereas *AA-High* means subsampling by 4 which yields 17 gray values including black and white.

Notice that, aside from the latter, the toggle buttons only affect the string rastering functions.

3.2 Buttons that Influence the X11 Rastering Functions

The X11 rastering functions introduced in version 0.3-beta provide a considerably higher level of abstraction than the standard rastering functions. To show the effect in `xglyph`, a few additional buttons are provided.

- **Transparent**/**Opaque**
  This button allows to switch between transparent and opaque mode in the X11 rastering functions. In transparent mode, only non-background pixels are drawn and all other pixels are left untouched. In opaque mode the entire area that the bitmap will require is first filled with the background color and then the bitmap is placed on this area.

- **Foreground**
  This is a label field with six color fields to the right and one color field to the left. Clicking on one of the color fields located on the right side will set the foreground color to the respective value (white, black, gray, red, green or blue). The color field on the left side always shows the current color selection.

- **Background**
  This also is a label field with six color fields to the right and one color field to the left. It works in analogy to the above and sets the current background color. Note that in order make the background color active, the drawing mode must be set to "opaque".

3.3 Buttons that Generate Actions

There are 10 buttons generating actions visible to the user.

- **Char**
  This button generates a bitmap of the character specified in the **Character**-dialog box. All parameters changed earlier become effective at this time. The resulting bitmap is then shown in the output window of `xglyph`. Some information about the generated bitmap

and elapsed time etc. is given in the message window. If an error occurred, the old contents of the output window are kept and a message is given to the user.

- $\boxed{\textbf{String}}$
  This button generates a bitmap of the string specified in the $\boxed{\textbf{Test-String}}$-dialog box. In addition to rastering characters, kerning and ligature settings may now take influence on the result of the operation (see 3.7). If no error occurs, the bitmap is shown in the output window and additional information is shown in the message area. Otherwise, an appropriate error message is given.

- $\boxed{\textbf{AAChar}}$

- $\boxed{\textbf{AAString}}$
  Both of these buttons do exactly the same as their non-antialiased counterparts. The only difference consists in the generation of an antialiased bitmap. The result is not a bitmap in fact. There are at least 8 bits per pixel and at most 32 bits per pixel in the resulting glyph. This depends on the depth of the X11-visual you use when starting xglyph. The result may consume quite a bit of memory if a `TrueColor` or `DirectColor` visual is active.

- $\boxed{\textbf{CharX}}$

- $\boxed{\textbf{StringX}}$

- $\boxed{\textbf{AACharX}}$

- $\boxed{\textbf{AAStringX}}$
  These functions basically do the same as the counterparts lacking the "X" in the name. But internally the X11 rastering functions are called to produce the output bitmap/pixmap. As a consequence the current foreground color, background color and drawing mode are taken into account. For a more complete discussion of the X11 rastering functions see 6 on page 84.

- $\boxed{\textbf{Font Table}}$
  A character table of size 16×16 is shown in the output window. Each cell contains an antialiased representation of the character indexed by the field number. The function `T1_AASetCharX()` is used for drawing these characters. Current foreground and background colors are respected as well as are most other parameters accepted by the character rastering functions. Only the angle specification is ignored since I assume that it is not very useful to have an overview over a font at any angle different from 0. Notice that the default size (100) is probably too large to make the output window fit on the screen. No care is taken about this. The recommended size for viewing a font's character map is between 20 and 30 points at 72 dpi resolution.

- $\boxed{\textbf{About}}$
  Shows an 'about' message telling you that you are using `xglyph` and `t1lib` in the current version.

- $\boxed{\textbf{Exit Program}}$
  This button does what it says and exits the program.

3.4 The Message Window

This area is located below the dialog box for the test string. Information potentially useful to the user is given here. There should be nothing needed to be said about the info—it is self-explaining. But two things should be noted:

- The elapsed time that is displayed is exactly the time spent in the respective rastering function. There might have been other actions in force which might make the user believe the time value given as being incorrect.[4] Moreover, the transfer to the X-server may become significant if 32 bits per pixel are used, the image is large and the program is running on a remote machine.
- The message `t1lib: Couldn't generate Bitmap, T1_errno=...!` simply tells the user that no bitmap could be generated. There may be several reasons. E.g., the `FontID`-value given might have been out of range. Another possibility is that you have specified a character index which has no encoded character associated. The value of `T1_errno` might give a hint of what the problem was.

If a character map is displayed, the message window is giving no information apart from the font name and the final value of `T1_errno` because there have been executed up to 256 rastering operation and it would be impossible to keep track of all single operations.

3.5 The Output Window

The output window shows the output of the rastering operations. Its type is different for the standard and the X11 rastering functions. For the standard rastering functions, it is always adapted to the size of the glyph which is displayed plus a margin of 5 pixels on each side. For this reason, leading and trailing white space is not shown in the output window, it only shows up in the glyph's metrics in the message windows.

The X11 rastering functions generate an output window of constant size (600×400 pixels) with a logical origin in its center. This center is marked by a cross-hair of color cyan. The glyph is placed with respect to this origin. If it is too large to fit, the glyph simply is clipped. A second cross-hair is shown at the place where the origin of the next glyph would be located in color magenta.

If a character map of a font is displayed, the output window contains a map of 16×16 cells whose size depends on the metrics of the font that is displayed.

3.6 `xglyph` Commandline Parameters

The syntax of the `xglyph` commandline is

```
xglyph [options] [fontfile1 [fontfile2 [...]]]
```

If no options and no font files are specified on the commandline `xglyph` reads the fonts from the font database file. The details depend on which configuration file is found and on this file's contents. If at least one font file is specified on the commandline, the font database—being existent and accessible or not—is ignored and the database is built using the fonts from the commandline. `fontfile1` is assigned the font ID 0, `fontfile2` is assigned the ID 1 and so forth. Files that cannot be opened for some reason are simply skipped.

`xglyph` also recognizes a few options. Notice that these options are not intended for an average user. Rather, they provide a means of (a) controlling debugging output from the rasterizer,

[4] For example, there might have been size-dependent data to be deleted and recreated, or an encoding file might have needed to be loaded before rastering.

(b) controlling generation and verbosity of the `t1lib`-logfile, (c) disabling certain features of the rasterizer and (d) checking some `t1lib` functions which otherwise would not be required because of the limited functionality of `xglyph`.

All commandline arguments that start with "`--`" are considered to be options to `xglyph`. The following is a complete list of valid options and a brief description of their effect:

- `--help`: Display the commandline syntax of `xglyph` as well as a brief list of the available options for an average user and exit.
- `--Help`: Display the commandline syntax of `xglyph` as well as a brief list of all available options.
- `--noGrid`: The cross-hairs marking start and end position of a glyph in the output of one of the X11 rastering functions will be suppressed. This might be useful at small sizes because the grid overwrites the glyphs' pixels.
- `--setPad`: The padding value `xglyph` should use can be specified here. This has to be followed by either "8", "16" or "32", separated by a space. Notice that the value "32" might be rejected as described in 5.2. The value actually used can be found by writing a logfile and examining this after a session.
- `--logError`:
- `--logWarning`:
- `--logStatistic`:
- `--logDebug`: These options firstly instruct `xglyph` to create a `t1lib` logfile and secondly set the loglevel to the respective value (see 5.4). Without specifying one of these options, no logfile will be generated.
- `--ignoreForceBold`: Instructs the rasterizer to ignore a `ForceBold` hint in the Type 1 font file.
- `--ignoreFamilyAlignment`: Instructs the rasterizer always to compute font level alignment according to `BlueValues` and `OtherBlues`, even if `FamilyBlues` and `FamilyOtherBlues` exist and all conditions for substitution are fulfilled for that combination of font and size.
- `--ignoreHinting`: Instructs the rasterizer to omit hinting completely.
- `--ignoreAFM`: The use of AFM information is ignored, no matter whether it could be accessed via an appropriate AFM file or self-generated. When using this option the string functions would not work. It may, however, be useful because self-generation of AFM data fails as soon as at least one character of all defined characters can not be processed and thus, the font will refuse to load. Using this option, consequently, one has access to all character that are processible, e.g., for generating a font table. characters that can be rasterized
- `--debugLine`:
- `--debugRegion`:
- `--debugPath`:
- `--debugFont`:

- `--debugHint`: All these instruct the rasterizer to write particular debug messages from intermediate steps of rasterization to the terminal. In order to understand and interprete them, a thorough understanding of the Type 1 font format specification and this special rasterizer implementation is crucial.

- `--checkPerformance`: This option affects the X11 string rastering functions. An additional output window is created and the output of before-mentioned functions is directly written into this window. Note that this window is not managed as might be expected. Text is only drawn at the visible parts and after partially or completely hiding and again raising the respective areas are lost. This mechanism should simply give an idea of how fast the X11 rastering function work, admittedly a critical topic in `t1lib`.

- `--checkCopyFont`: This option is used to check the proper functioning of the `T1_CopyFont()` function. It copies all fonts from the database to new logical fonts and slants these fonts by 0.3. Finally an additional fontfile is added to the database. For each step the new font ID is printed to the terminal and the initial and final number of fonts are printed.

- `--checkConcatGlyphs`: This option affects the buttons **String** and **AAString**. The requested string glyph is generated twice, first time using the current value of font ID and second time using this values plus 1. Both resulting glyphs are concatenated and the result is shown in the output window.

- `--checkConcatOutlines`: This option, too, affects the standard string rastering functions. The current test string will be fetched as an outline using the current font ID. The result is then concatenated with a horizontal movement of 1000 charspace unit which is followed by the identical string using a font ID advanced by 1. The result of the concatenated outlines is then filled and converted to a `t1lib`-glyph. For details on outline handling and what it is meant for see 5.14.

  When `--checkConcatGlyphs` and `--checkConcatOutlines` both are specified on commandline, the `--checkConcatGlyphs` is respected.

- `--checkBadCharHandling`: This option provides a means of examining the effects of problematic/bad characters on string handling in `t1lib`with `xglyph`. It affects only the **String** button. The character to be specified in the test character field is inserted in the middle of the test string. This enables the user to insert arbitrary character codes in the middle of a test string and to watch the effect in `T1_SetString()`. For example, starting `xglyph` with this option exclusively and immediately clicking **String** will show the string *TeAt* in the output window because the character 's' has been overwritten by character 65_{10} ('A'), the default test character.

- `--checkDefaultEncoding`: This option proves that the default encoding feature works correct. If set, all fonts should be encoded in IsoLatin1 encoding immediately after startup and without any slow down at startup.

- `--checkSmartAntialiasing`: Enables smart antialiasing as described in 5.13. The effect is that `t1lib` will determine the antialiasing level by itself. For sizes below 20 bp, 4× antialiasing will be used and up to 60 bp 2× antialiasing is used. For size of 60 bp and larger a "bytemap" is created which in fact consists only of pure background and foreground pixels. When this options has been specified, toggling the antialiasing level has no effect.

- `--checkAACaching`: This option enables caching of antialiased character glyphs. For a discussion of this issue see 5.13.4. The string rastering functions are not affected by this option.

- `--checkSetRect`: This option allows to check the rectangle drawing functions of `t1lib`. If specified, the character drawing buttons will produce an em unit square of the current font instead of the expected character.

- `--cacheStrokedGlyphs`: If this option is specified, `xglyph` will cache stroked glyphs whereas filled ones are not cached.

For those who are wondering about special X11 options like "`-display`", `xglyph` does not support these. The widgets are built straight ahead and the layout is fixed. `xglyph` is meant to be a tool for testing some of the functionalities of `t1lib` and nothing more. Of course, a display other than `localhost:0.0` may be specified by environment variable `DISPLAY`.

3.7 Fonts Included in the `t1lib`-Package

Included in the `t1lib`-package are 8 fonts of two families which are freely available.

1. The CharterBT-fonts in Roman, Italic, Bold and BoldItalic variants. These are good quality fonts containing kerning information. Notable kerning pairs are "T"–"e", "A"–"V" and "A"–"T".

2. The ComputerModern-fonts by Donald E. Knuth in the variants Roman, Italic, Bold-Extended and BoldExtendedItalic. In addition to kerning information, there are many ligatures in these fonts. Most people will know about the present ligatures from using TEX, the typesetting system. Notable examples are: `--` → "–", `---` → "—", `fi` → "fi" and `ffi` → "ffi". These fonts have the disadvantage that no ordinary space character is included, only a visible space. TEX itself does not need a dedicated space character.

Note that working with ligatures requires the ligatures to be encoded at the positions they are expected. Thus, if you reencode one of the dc-fonts with the IsoLatin1-encoding funny effects may occur.

3.8 Some General Remarks on `xglyph`

The program `xglyph` is just intended to check whether `t1lib` works on a particular system. It is written straight forward and does not care much on performance and such. Especially, it is not a typical application for a rasterizer library. The bitmaps are generated and then the output window is adapted to the size of the bitmap/pixmap generated by the rastering function. This contradicts the X11 principle of "drawing into a drawable" and requires some overhead if one wants a fitted output window. In that sense, the performance of the X11 rastering functions is not directly comparable to the performance of the standard rastering functions. This should always be kept in mind.

4 The Program `type1afm`

`type1afm` is a simple commandline tool (about 150 lines C source code) that allows to generate an AFM file from a Type 1 font program. It is intended for people who want to use Type 1 font files that come without AFM files with `t1lib` (or other software that requires AFM files). The syntax is

```
type1afm [-l] <fontfile1> [<fontfile2> <fontfile3> ...]
```

For each fontfile specified on the commandline, an AFM file with the corresponding name is generated in the current directory. Most of the work is done in `t1lib`-internal functions. See section 5.16 on how AFM information is generated and written to files.

It is usually not desireable to leave a logfile wherever the utility has been executed. Thus by default no logfile is generated. This behaviour can be changed by specifying the optional parameter `-l`. This causes a logfile with `T1LOG_DEBUG` as loglevel to be written to the disk. Its name will be `t1lib.log` (see 5.4).

5 Using `t1lib`

This section describes in detail how to use `t1lib`. I have tried to to describe the stuff in the order a new user would learn best and a new user would need to use the functions.

5.1 Compiling and Linking `t1lib`-Programs

A program that wants to use functions from the library must include the appropriate headers at compile time and then be linked with the appropriate libraries. Since V. 0.6-beta the X11 interface is separated from the `t1lib` pivotal stuff. This yields advantages for programs that don't use the X11 rastering functions on systems where X11 is installed. The following applies to programs that do not use the X11 rastering functions:

- Include the file `t1lib.h`. All definitions and declarations needed at compile time are included in this file.
- `libt1.a` or `libt1.so` respectively must be linked to the program.

In contrast, a program that uses the X11 interface must adhere to the following scheme:

- `t1lib.h` and `t1libx.h` must be included in this order. Furthermore, `t1libx.h` includes `X11/Xlib.h` if it is not already included.
- The libraries `libt1.a/libt1.so` and `libt1x.a/libt1x.so` must be linked to the executable. The correct order is `-lt1x -lt1` since the X interface uses functions from the latter. Also, the X11 library must appear in the library list after `-lt1x`.

The Makefiles for `xglyph` and `type1afm` are typical examples for both configurations.

5.2 Querying and Setting Fundamental Configuration Parameters of `t1lib`

It might be necessary to know whether `t1lib` is compiled with or without X11 interface. At compile time a programmer can check for the X11 interface by stating

```
#ifdef T1LIB_X11_SUPPORT
```

after including `t1libx.h`. If `T1LIB_X11_SUPPORT` is not defined, the X11 interface is not configured and compiled.

At runtime, a program can check for the X11 interface by a call to

$\mathcal{F}() \Rightarrow$
```
int T1_QueryX11Support( void)
```

It returns `1` if the X11 interface is present and `0` otherwise.

Notice that querying X11 support at runtime and compile time tends to be pretty useless starting with V. 0.6-beta. Any decision can be done by examining the existence of the `t1x`-library and the `t1libx.h` header file. The definition and the function described above are thus only provided for compatibility with pre-0.6 versions of `t1lib`.

Some remarks on the general data format of bitmaps and should be given here. `t1lib` internally always generates bitmaps in the way that appears to be natural for them: The first pixel corresponds to the least significant bit in a byte (or word/longword). Bytes are always arranged in memory the way, that the first byte is at the lowest address and the next byte at the following address. This convention is called LSBFirst which stands for Least Significant Bit/Byte First. It is the natural way of data alignment on machines with *Little Endian* data representation. In contrast MSBFirst stands for Most Significant Bit/Byte First which is the natural kind of data representation on Big Endian machines. A glyph's scanlines are always aligned in LSBFirst-type, no matter on what machine `t1lib` is running.

What has been said above, strictly does only apply to non antialiased glyphs, i.e., real bitmaps. Antialiased glyphs have their gray values coded in the representation that is natural for the machine `t1lib` is running on. For example, if `t1lib` runs on a Big Endian machine, the gray values are in Big Endian. The X11 displaying functions automatically handle this correct.

Scanlines of `t1lib`-glyphs may be padded to 8, 16 or 32 bit. Padding to higher values will consume more memory for the glyphs, but might speed up concatenating of bitmaps as described in 5.5. This applies to machines with Little Endian representation as, for example, Intel's $x86$ series. On these machines 16 or 32 bits can be placed into the target bitmap in one step. On machines with Big Endian representation, for example, Motorola $680x0$ series, this is currently not possible. However, using a higher padding value could still yield a better performance since the application could work on larger units than a byte.

The default padding value in `t1lib` is 8 bit. The padding value can be specified at runtime by means of calling

$\mathcal{F}() \Rightarrow$ `int T1_SetBitmapPad( int pad)`

`pad` must be one of '8', '16' or '32'. The call will only be successful if executed before initialization of `t1lib`. This a security mechanism which prevents from having glyphs with distinct padding values. The return value is 0 if successful and -1 if `pad` was invalid or `t1lib` had already been initialized.

There is a further restriction concerning the padding value. Setting it to 32 is only possible if the machine has an ANSI C integer type of 64 bits. This condition is automatically checked by the `configure` script of `t1lib`. If such an integer type is not present (or has to be emulated as e.g. `long long` in `gcc`) there would not result any performance gain. If a specified padding value is rejected, `T1_errno` is set appropriately.

An application can query the current padding value by calling

$\mathcal{F}() \Rightarrow$ `int T1_GetBitmapPad( void)`

The returned value is the padding value. This function can be called before or after initialization of `t1lib`.

Another function usually be called near initialization is

$\mathcal{F}() \Rightarrow$ `int T1_SetDeviceResolutions(float x_res, float y_res)`

This function allows setting the resolution of your device (screen). The values must be given in dpi. The default resolution, 72 dpi, implies that a pixel in device space equals 1 bp. This function may be called before or after initialization. The only restriction is that no size dependent data must be available. Changing the resolution when bitmaps are already cached would result in inconsistent bitmap-sizes for bitmaps generated before and after the call to `T1_SetDeviceResolutions()`. The function checks whether initialization has already been done. If not, all is OK since no size-dependent data for any font can exist. If initialization has been done, it checks for every font whether size dependent data exists. If there's any size dependent data for any font, `T1_SetDeviceResolutions()` will return `-1` without having set the new resolution. Otherwise the specified resolution will be set and the function will return `0`. If you really need to set another resolution in the middle of a session, all size-specific data should explicitly be removed from memory beforehand. This can be achieved using `T1_DeleteAllSizes()` (see 5.8).

Notice that the device resolution need not be set at all if the default resolution of 72 dpi in horizontal and vertical direction is OK. This function is primarily intended to be prepared for applications with a device aspect ratio different from 1.

5.3 Initialization of t1lib and Related Things

In this section we should cover the initialization, part of which has already been described in 2.3 in some more detail. This gives the user the chance to fine-tune the initialization for specific applications.

Prior to be able to do anything useful with `t1lib`, the library has to be initialized. Generally speaking, the purpose of the initialization is to tell `t1lib` which font files are associated with which font ID's. The existence or accessibility of the font files is also assured at this point. Hence, file name search paths for Type 1 font files, AFM files and encoding files have also to be known at this time.

The configuration file and the font database file play a central rôle during initialization. While the configuration file contains path specifications and a font database specification, the font database file specifies the relation between font ID's and font filenames. The format of both these files is described in 2.3 and not repeated here.

A further purpose of the initialization is to set certain flags that prevent other quantities from being modified at a later time. For example, the padding value must be unique to all glyphs and consequently it is not allowed to be changed after initialization has been performed.

The initialization is started by a call to the function

$\mathcal{F}() \Rightarrow$
```
void *T1_InitLib( int log)
```

The parameter `log` can be interpreted as a mode specification that influences certain parts of the initialization. In fact it should consist of one or more `#define`s from `t1lib.h`. At minimum, `log` should be either `LOGFILE` or `NO_LOGFILE`. If `LOGFILE` is specified, a log file is written while the application runs, and `NO_LOGFILE` suppresses the generation of a logfile. For information the `t1lib`-logfile see 5.4. In addition to this, `IGNORE_CONFIGFILE` and `IGNORE_FONTDATABASE` can be bitwise OR'ed (using "`|`") to the `log`-value. The purpose of this is described later in this section. A further flag that might find its way into the value of `log` is `T1_AA_CACHING`. A discussion of this topic is given in 5.13.4. The `T1_NO_AFM` completely suppresses usage of AFM data, no matter if an AFM file could have been found using the current search paths or not. This saves time for loading a font and is recommended if an application is known to be restricted on functions that do not access AFM data. The consequences of using this flag are covered somewhat more detailed in 5.16.2.

5.3.1 Standard Initialization

The term "standard initialization" means, that none of the path manipulating and font database manipulating actions described later has been performed. Also, a standard initialization excludes the use of `IGNORE_CONFIGFILE` and `IGNORE_FONTDATABASE`. If these conditions are met, the following happens at initialization time:

1. The padding value, either being the default value or a value specified by the user before, is assigned.

2. Next, depending on the value of `log`, a logfile is tried to be opened. From this point on, depending on the loglevel and the value of `log` the actions are logged.

3. The endianess of the machine `t1lib` is running on is checked.

4. A configuration file is searched in the following order:
   - The process' environment is checked for the entry `T1LIB_CONFIG` and if found, its value is interpreted as the filename of the configuration file (see 2.3).
   - If no file was found, the user's home directory is searched for a file named `.t1librc`. In case it exists, it is used as a `t1lib`-configuration file.
   - If still no configuration file was found, the global configuration file will be tried to be opened.
   - If this also does not succeed, all file search paths are left to be "." and the default font database is `FontDataBase`.

   It should be noted that the first match wins when searching the configuration file. Only the first one found is examined.

5. The font database file(s) are tried to be opened and read. This process is in detail described in 2.3.3.

   If after scanning the complete list of font databases, no font definitions have been performed, `T1_InitLib()` will return with a `NULL` pointer indicating the initialization has failed.

6. The number of fonts declared in the database is stored. Note that this number of declared fonts must be greater than zero.

7. A flag is set to indicate `t1lib` is initialized and the pointer to the top most area of the global data structures is returned to the application. This pointer is guaranteed to be not `NULL`.

For some applications, the described initialization scheme appears to be too inflexible and overkill. It is well suited for large applications that use lots of fonts and where it should be possible to add new fonts without modifying the application itself. For small commandline applications like `type1afm` (see 4), which are designed to read a few font file names from the commandline, the overhead in configuration file searching and path reading is much too large. Moreover, to insist on a font database file might grow to a real disadvantage. In the subsequent paragraphs we should thus discuss how we can deviate from the initialization scheme described above.

5.3.2 Fine Control over Font Databases and File Search Paths

First, it is important to mention that it is generally possible to force `T1_InitLib()` to skip steps 4 and / or 5 as described above.

The configuration file is discarded by OR'ing the parameter `log` of `T1_InitLib()` with `IGNORE_CONFIGFILE`. The default paths or the paths explicitly specified by the application before are then left untouched during initialization.

Discarding the font database specification from the configuration file is achieved by bitwise OR'ing `log` with `IGNORE_FONTDATABASE`. The result after initialization will be an empty database. This is valid in V. 0.5 and newer since fonts may be added to the database at runtime.

Using the two flags described above, an installed setup can be ignored and an application may have complete control over its search paths. There are a number of functions for this purpose and each of these functions allows to specify exactly *one* path element at a given time. Using special characters like white space or path separators in a path element is thus not an issue.

The font database may explicitly be specified by the application using

$\mathcal{F}() \Rightarrow$ `int T1_SetFontDataBase( char *filename)`

`filename` is the pointer to a string containing the name of the font database file that is to be examined. This function replaces any set of font databases setup previously and is thus usually to be called before `T1_InitLib()`. A call to this function after initialization will return the result -1 and `T1_errno` will be set to `T1ERR_OP_NOT_PERMITTED`. For the special case that the font database is still epmty after initialization this function may also be called after initialization. But in any case, the database must be empty! For the latter cases, fonts specified in the new database file are then immediately inserted into the database. In case of success, the number of available fonts is returned.

There is also the function

$\mathcal{F}() \Rightarrow$ `int T1_AddFontDataBase( int mode, char *filename)`

It can be called at any time and `filename` is the string of the new font database's file name. If called before initialization, the parameter `mode` can be either `T1_APPEND_PATH` or `T1_PREPEND_PATH` and the name of the new font database simply is either appended or prepended to the existing list of font databases. Then, the function returns 0.

If the function is called after initialization, the parameter `mode` is ignored and the new database is appended in any case, because this is the only meaningful action. Furthermore, the new database is immediately loaded and the function returns a number indicating the number of available fonts (which can also be 0). In the case of an error, `T1_AddFontDataBase()` returns -1.

There is also a pair of functions that act on the list of XLFD font database files,

$\mathcal{F}() \Rightarrow$ `int T1_SetFontDataBaseXLFD( char *filename)`

$\mathcal{F}() \Rightarrow$ `int T1_AddFontDataBaseXLFD( char *filename)`

These functions have exactly the same syntax and semantics as the functions just described above. Standard font database files and XLFD font database files are handled in separate lists internally, so that setting the XLFD font database will not affect explicit standard font database settings. However, the default standard font database (`./FontDataBase`) will be cleared when at least one XLFD font database is specified. This is because the default font database is meant as a fallback, and the fallback case is no longer given if a XLFD font database has been specified.

A similar manipulation as just described for font database files, is possible for the searchpaths. (Re-)Defining a search path is done by calling

$\mathcal{F}() \Rightarrow$ `int T1_SetFileSearchPath( int type, char *pathname)`

before calling `T1_InitLib()`, or respectively, before the database contains any fonts. An attempt to set the file searchpaths when the database is no longer empty is denied. The reason is that it is not wise to override the searchpaths which had been valid previously during the verification of the existence of font files. Such paths should thus not be removed. The parameter `pathname` points to the string that contains the pathname that should be used as searchpath. The parameter `type` is any OR'ed combination of `T1_PFAB_PATH`, `T1_AFM_PATH` and `T1_ENC_PATH`. These tell the function to set the search paths for Type 1 font files, AFM files and encoding files, respectively. In case of an error `-1` is returned and otherwise `0`.

Extending the file searchpaths before *and* after initialization is possible using

$\mathcal{F}() \Rightarrow$ `int T1_AddToFileSearchPath( int pathtype, int mode, char *pathname)`

This might be useful to locate font files that were of no interest at the time of initialization. `pathname` is the pointer to the string that should be added to the searchpaths. Only *one* particular path element may be specified in one call. What searchpaths are affected is determined by the parameter `pathtype`. Again, similar to described above, any OR'ed combination of `T1_PFAB_PATH`, `T1_AFM_PATH` and `T1_ENC_PATH` is valid. There are two ways to extend an existing searchpath which are specified by the `mode` parameter. It must be either `T1_APPEND_PATH` or `T1_PREPEND_PATH`, which causes the new path element to be appended or prepended to the existent path respectively. Since an existent path specification is not overwritten by `T1_AddToFileSearchPath()`, this function may be called at any time before or after initialization.

It might also be of interest to query the current file search paths and font databases. `t1lib` provides

$\mathcal{F}() \Rightarrow$ `char *T1_GetFileSearchPath( int type)`

for querying search paths. Again, the parameter `type` determines what search path is returned. Exactly one of `T1_PFAB_PATH`, `T1_AFM_PATH`, `T1_ENC_PATH` and `T1_FDB_PATH` should be specified. If more than one path is specified, the first match wins and only one path is returned. The types will be checked in the order `T1_PFAB_PATH`, `T1_AFM_PATH`, `T1_ENC_PATH` and `T1_FDB_PATH`. Here, `T1_FDB_PATH` indicates an interest in the list of font databases. The paths are separated using the current path separator.

5.3.3 Adding Fonts to the Database

Extending the font database is possible at any time after initialization. In addition to using `T1_AddFontDataBase()` (see above), it is done via a call to

$\mathcal{F}() \Rightarrow$ `int T1_AddFont( char *fontfilename)`

`fontfilename` is the pointer to the filename string. The following actions take place:

- The font file is searched using the current search path specifications.
- If the file has been located, it is checked whether the font database contains enough memory to hose an additional font. If so, the font filename is stored and the function returns `new_ID`, which is the font ID that will be associated with this font in the future.
- If there was no free memory for an additional font, the memory is reallocated to a greater size. This involves also resetting the new area. Finally the value `new_ID` is returned.

If a negative value is returned the function failed. `-1` indicates the font file could not be located. `-2` or `-3` are returned if a memory allocation failure occurs.

5.3.4 Summary of Search Path Handling

Since the exact handling of search path specifications at the several stages may appear somewhat confusing we shall summarize the exact rules now:

1. **Before Initialization**
   Default paths are not yet setup. Each call of one of the `T1_Set...()` functions described above establishes a completely new respective path. Each call of one of the `T1_Add...()` functions extends the respective path in the desired manner, or creates a new respective path if previously no path existed.

2. **At the Beginning of the Initialization**
   Each path type for which a path had not already been explicitly created using the `T1_Set...()` or `T1_Add...()` functions receives a default value. This is "`.`" for the file search paths and `./FontDataBase` for the font database.

3. **During Initialization**
   Each path element read from the configuration file overwrites an existing respective default path but preserves an existing respective explicitly setup path by appending to the latter. The newly setup search path is used to locate files while scanning the font database file(s).

4. **After Inititialization but before the Database is being filled**
   At this stage, all `T1_Set...()` and `T1_Add...()` functions still work as described under (1). Notice that this phase usually is not accessible when using the standard initialization scheme. It only becomes accessible if no fonts have been added during initialization.

5. **After at least one Font is insterted into the Database**
   Once there are fonts in the database, the paths setup up to now must be preserved in any case. Hence, the `T1_Set...()` may not be called any longer. The `T1_Add...()` functions *extend* the current set of search paths. In particular, possibly existing default search path elements, then, are not overwritten any longer because they might have been used before to verify the existance of required files.

A call to `T1_AddFontDataBase()` will not only append the specified file name to the existing list of databases, but will also immediately locate the fonts specified therein and assign additional font IDs.

5.3.5 Bypassing the `t1lib` File Search Machinery

Usually, `t1lib` takes care for locating files according to the path specifications in the configuration file. There might, however, arise the need to explicitly tell `t1lib` which particular file to use. Forcing `t1lib` to use particular Type 1 font files can be achieved using the function `T1_AddFont()`, described just above. If a pathname passed to this function is a complete path, `t1lib` will use this complete path to locate the font file, forgetting about its own search path list. A filename path is assumed to be complete if

- it starts with the directory separator character, usually "`/`". In this case it is an *absolute* path specification, meaning that the start point is at the root of the filesystem, or

- if it starts with "`./`" or "`../`" (where it is assumed that "`/`" is the directory separator character). Here we have a *relative* path specification in which "`.`" refers to the current working directory while "`..`" refers to the parent directory of the current working directory. Since the notion of the *current working directory* is fundamental for every process that has access to a filesystem, a relative path specification also uniquely identifies one particular file in the filesystem.

If a font file whose complete path had been specified could not be found by `t1lib`, the paths from the configuration file are searched as a fallback mechanism.

What can be done for the Type 1 font files is also possible for AFM files, which are needed on a per-font basis. The function

$\mathcal{F}() \Rightarrow$ `int T1_SetAfmFileName( int FontID, char *afm_name)`

allows to set the complete path of the AFM file belonging to the font identified by `FontID`, overriding the internal search mechanism. This function is to be called after initialization but before the font `FontID` is loaded. It returns 0 if all goes right and -1 otherwise. In the latter case `T1_errno` will also be set appropriately. Notice that `FontID` must also be valid with respect to its upper limit, it is an error condition if the font database has less than `FontID` entries.

There is also the function

$\mathcal{F}() \Rightarrow$ `char *T1_GetAfmFileName( int FontID)`

which allows to query the AFM filename of a font. It returns a pointer to the filename if it had explicitly been specified or `NULL` otherwise. `NULL` will also be returned if `FontID` was invalid. In this case also `T1_errno` will be set.

Just for the sake of completeness we should mention that what has been said about absolute and relative path specification also applies to pathnames for encoding files (see 5.7).

5.4 The `t1lib`-Logfile

Since version 0.2-beta `t1lib` supports a runtime logfile. It implements an uncomplicated way to keep track of errors, warnings, statistics and debug messages without overloading stdout/stderr. As seen in 5.3 the user must specify whether or not to use a logfile when calling `T1_InitLib()`. Specifying `LOGFILE` as argument leads to using a logfile and `NO_LOGFILE` suppresses the use of a logfile.

The name of this logfile is by default `t1lib.log`. This name is defined in `t1misc.h` and can be changed there as the user likes.

Basically `t1lib` distinguishes 4 types of runtime messages. Each type is associated a "loglevel":

- *Errormessages/Level1:* They are considered that important that the user is in any case informed. Example: During initialization the memory allocation for one of the basic data structures of `t1lib` failed.

- *Warningmessages/Level2:* They are considered important but it is not absolutely necessary to inform the user. Example: An AFM file could not be loaded for a given font. This imposes several restrictions on what can be done with that font but it is possible to generate bitmaps.

- *Infomessages/Level3:* They do not indicate a problem. Rather, the user is notified about some facts and statistics that might be of interest. Example: After loading a font the consumption of virtual memory is displayed.

- *Debugmessages/Level4:* These can be pointers, numerical data etc. Example: Print out the pointers that point to the memory area where the PostScript dictionaries for a font just loaded are located.

The decision what message to put into the logfile is done by examining the value of an integer variable whose values can be `T1LOG_ERROR` (=1), `T1LOG_WARNING` (=2), `T1LOG_STATISTIC` (=3) or `T1LOG_DEBUG` (=4). All messages whose level is below or equal to this value are put into the logfile. The user may set this loglevel by calling

$\boxed{\mathcal{F}() \Rightarrow}$ `void T1_SetLogLevel( int level)`

The default value is T1LOG_WARNING which means that error and warning messages are stored in the logfile.

If the usage of a logfile has been specified, t1lib tries first to open it in the current directory. If this fails for some reason t1lib tries to create it in the user's home directory. If this fails too, an error message is printed to stderr and no logfile is used.

If the application programmer chose not to create a logfile, it would be hardly possible for a user to track down possible problems, e.g. in file searching. To overcome this disadvantage, the user may set an environment variable T1LIB_LOGMODE at runtime. This variable is evaluated by t1lib when T1_InitLib() is called. If its value is one of the four strings logDebug, logStatistic logWarning and logError, the respective loglevel is set by t1lib and a log file is created, even if the programmer chose not to do so. However, if the programmer had altered the log level after calling T1_InitLib(), this cannot be catched by setting the environment variable. A log file is created anyhow so that at least error messages will be logged.

The user himself may also put some messages into the logfile. This can be achieved using

$\boxed{\mathcal{F}() \Rightarrow}$ `void T1_PrintLog( char *func_ident, char *msg_txt, int level, ...)`

where func_ident is a pointer to a string identifying the function that generates the message. msg_txt points to the text string to put out. The distinction between a function identifier and a message text is only formal, indicating the user should identify the function that generates the message.

The string msg_txt may contain format character sequences, %.., as known from the printf- or scanf standard C functions. In this case, the ellipses indicate that a variable list of further arguments may follow. The level specification works as described above: The message is only put out if the internal loglevel is equal or greater than level.

Here is a typical example of a log file after a (short) xglyph-session in which the loglevel was set T1LOG_STATISTIC. Among several informative messages of type S, also two messages of type W have been generated. They stem from trying to raster the character "ß" which was not in the current encoding.

```
(S) (Mon Jul 14 18:27:34 1997) T1_InitLib(): Initialization started
(S) (Mon Jul 14 18:27:34 1997) T1_InitLib(): Initialization succesfully finished
(S) (Mon Jul 14 18:27:44 1997) T1_LoadFont(): VM for Font 0: 35132 bytes
(S) (Mon Jul 14 18:27:44 1997) CreateNewFontSize(): New Size 100.000000 created for FontID 0 (antialias=0)
(S) (Mon Jul 14 18:27:53 1997) CreateNewFontSize(): New Size 100.000000 created for FontID 0 (antialias=1)
(S) (Mon Jul 14 18:27:53 1997) CreateNewFontSize(): New Size 200.000000 created for FontID 0 (antialias=0)
(W) (Mon Jul 14 18:27:53 1997) T1_SetChar(): No black pixels found for character 223 from font 0, returning NULL
(W) (Mon Jul 14 18:27:53 1997) T1_SetStringX(): T1_SetChar() returned NULL-pointer!
(S) (Mon Jul 14 18:27:55 1997) T1_DeleteSize(): Size 200.000000 deleted for FontID 0 (antialias=0)
(S) (Mon Jul 14 18:27:55 1997) T1_DeleteSize(): Size 100.000000 deleted for FontID 0 (antialias=0)
(S) (Mon Jul 14 18:27:55 1997) T1_DeleteSize(): Size 100.000000 deleted for FontID 0 (antialias=1)
```

5.5 Generating Bitmaps

At this point, you are able to generate a bitmap. As said before, a character- or string-bitmap is given to the user as an object of type GLYPH. We should briefly explain GLYPH here. The type is defined by

```
typedef struct
{
  char *bits;
  struct
  {
    int ascent;
```

```
    int descent;
    int leftSideBearing;
    int rightSideBearing;
    int advanceX;
    int advanceY;
  } metrics;
  void *pFontCacheInfo;
  unsigned long bpp;
} GLYPH;
```

`bits` is a pointer to the bitmap data. The bitmap is organized in lines, starting with the uppermost line. Each bitmap pixel is usually represented by one bit. If the width of the bitmap is not an integer multiple of 8, the lines are padded with zeros, so that each line starts at a byte boundary. Note that the bitmap has no margins taken into account. The bitmap occupies the minimum area the character needs to be painted.

The bitmap pointer may also be the `NULL`-pointer. In this case, the glyph contains no foreground pixels. The metrics of the corresponding glyph should be valid, though. Typically, this appears for the space character as well as in situations where an undefined or unencoded character had been substituted by the `.notdef`-character within the rastering functions.

Note that the `pixmap`-entry which has been present in version 0.3-beta, has been removed with version 0.4-beta. See the discussion on the X11-interface in 6 for an explanation of this.

The struct `metrics` contains metric information that is needed to position the character and to describe the character origin with respect to the bitmap. The members in detail:

- `metrics.ascent`: describes how many lines the bitmap ranges above the line $y = 0$.
- `metrics.descent`: describes how many lines the bitmap ranges below the line $y = 0$. Width below $y = 0$ counts negative so that the difference `ascent` − `descent` is the total height of the bitmap, the number of lines.
- `metrics.leftSideBearing`: The amount of spacing between the origin of a character and the x-coordinate of its leftmost painted pixel. One could also name it “left margin” of the character.
- `metrics.rightSideBearing`: The horizontal difference between the origin of a character and the x-coordinate of its rightmost painted pixel. This definition stands in contrast to some other interpretations of the right side bearing, where it is assumed as the difference between the glyph’s width and its right most pixel.
- `metrics.advanceX`: The amount of position increment in horizontal direction after this character bitmap (or string bitmap) has been placed. It is almost always larger than the bitmap width because most characters contain a certain amount of margins. Note that this value is not suitable for internal computations of character positions since it contains the horizontal escapement rounded to the pixel grid. Using this value for such computations leads to accumulating positioning errors.
- `metrics.advanceY`: The amount of position increment in vertical direction after this character bitmap (or string bitmap) has been placed. Upper direction counts positive.

As seen, the width of the bitmap is given as the difference between `rightSideBearing` and `leftSideBearing` and the values `metrics.leftSideBearing`, `metrics.descent`, `metrics.rightSideBearing` and `metrics.ascent` effectively describe the bounding box of the glyph.

The entry `pFontCacheInfo` is not currently used but will probably later when font caching is really implemented. Moreover, there's a certain chance that some other entry will be added in future releases.

The member `bpp` is used to store the depth of the bitmap. For true bitmaps, it is always 1. See 5.13 for an explanation.

There are three basic functions which produce pointer to glyph objects. In order to generate the glyph for a single character you would use the function

$\mathcal{F}() \Rightarrow$
```
GLYPH *T1_SetChar( int FontID, char charcode,
                   float size, T1_TMATRIX *transform)
```

As in most other functions, `FontID` is a valid identification number of a font. It can range from 0 to $n-1$, where n is number of fonts declared in the font database file.

The second argument `charcode` determines the character that will be rasterized. As mentioned earlier, the encoding mechanism is used for accessing the output character. This means, if `'A'` is given as the character code, the machine representation of `'A'` is used as an index into the current encoding vector. In this encoding vector, the characters' name is looked up. Encoding vectors may be changed by the user (see 5.7).

The parameter `size` is interpreted in Postscript's bigpoint-unit (bp). By default, 1 bp equals one device pixel.

`transform` specifies the transformation that will be applied to the character before rastering. If this pointer is `NULL`, no transformation is used. Otherwise it should point to a valid `t1lib`-transformation matrix. Please refer to 5.11 for information on how to easily create `T1_TMATRIX` matrices. Hinting is only performed if the transformation is a pure rotation and if the the angle is one of 0, 90, 180 or 270 degrees. Otherwise font level and character level hinting information is ignored.

Bitmaps of transformed characters are never saved in cache memory since I assume that they are rarely needed. The overhead to manage transformed characters in cache would be overkill and would significantly increase memory consuming. Anyhow, this would only work for some dedicated transformations.

`T1_SetChar()` in fact does some more things than simply rastering the specified character:

- It checks whether the font in question is already loaded. If not, it is loaded.
- If the size dependent data structures for the size in question do not exist, it creates them and inserts them in the linked list of size dependent data structures.
- It checks if the character is already existent in cache. If so, it returns the data from cache.

Some words concerning memory management: The memory used by the `GLYPH`-structure is static in this function. The memory required for the bitmap is also allocated by the function itself. This means, the user doesn't need to free any memory by himself. Every time `T1_SetChar()` is called, it starts by giving the memory needed for the last generated glyph free or respectively setting metric values to 0. Thus, do not free the returned glyph pointer since later `T1_SetChar()` will free memory that is no more allocated and probably used for some other purpose. If you really like to free the memory, set the pointer to `NULL` afterwards.

If an error occurs at some point, `T1_SetChar()` returns a NULL-pointer to the user.

Frequently it is advantageous to raster a series of characters at once. This has the advantage that internal accuracy may be used and the overhead for the user is minimal. For such cases, the function

```
GLYPH *T1_SetString( int FontID, char *string, int len,
                     long spaceoff, int modflag,
                     float size, T1_TMATRIX *transform)
```

$\mathcal{F}() \Rightarrow$

is provided. It can be considered an extended version of `T1_SetChar()`. The same as said above applies to the arguments `FontID`, `size` and `transform`. But a few additional arguments are needed here.

`string` is a series of bytes representing the indices into the current encoding vector. The `len`-parameter is needed because we cannot imply that string is always an object like a string in C. For example, the Computer Modern Roman fonts contain the uppercase Greek Gamma (Γ) at position 0 in their internal encoding. In a string to be typeset the value 0 is thus a valid value and deserves no special treatment. Hence, we cannot not use the C-function `strlen()` to compute the length of the string. However, since in most usual encodings the special value 0 is not encoded ("`.notdef`"), it makes sense to switch between standard situations and non-standard situations:

- If `string` is a string conforming with C-semantics, `len` can be set to 0. Then, the length of the string is internally computed.

- If `string` contains one or more control characters which make processing impossible, the `len`-value must be specified explicitly.

The `spaceoff` parameter is important for word processing purposes. The value specified here is interpreted as an offset to the space width used during rastering. It is interpreted in charspace units, i.e., 1/1000 bp. Every time a space character is requested, this amount of horizontal escapement is added to the natural spacewidth of the current font. Note that the space character itself is actually not rastered. All requests to the character with the charactername "`space`" are caught by `t1lib` and converted to a simple horizontal escapement. For computation of the resulting spacewidth, the width of the space-character is taken from the AFM data and merged with the specified offset, which may also be negative.[5]

The `modflag` argument may be used to specify some option to the rastering function. It is generally 0 or an OR'ed combination of the following names:

- `T1_KERNING`: As the name implies, this argument determines that pairwise kerning information from the AFM file is to be taken into account during string rastering. Not specifying `T1_KERNING` means: "omit kerning". It is generally recommended to use kerning information since this improves the optical appearance. However, many lower quality fonts do not have kerning information. With `t1lib` V. 0.4-beta, kerning information is accessible much faster than before because it is based on char codes rather than on characternames.

- `T1_RIGHT_TO_LEFT`: Setting this flag will invert the writing direction. In *Right-To-Left* mode the escapement in writing direction (left) is inserted before placing the character with the result that the character laps over to the left side. This principle is kept for all characters in the string. Note that metrics of fonts that are intended for *Right-To-Left* typesetting have the same meaning as for fonts intended for standard (*Left-To-Right*) typesetting.

- `T1_UNDERLINE`: The string to be rastered is to be underlined according to the line specifications of the current font.

[5]One consequence of this handling is, that—with a little tricking—fonts that do not define a space character themselves may be used for typesetting. This applies to the dc-fonts, which only define a visible space, but no real space (see 3.7).

- `T1_OVERLINE`: The string to be rastered is to be over lined.
- `T1_OVERSTRIKE`: Same here for overstriking.

For a description of underlining and such, see 5.9. Notice also that the `modflag` argument is a replacement of the `kerning` argument from pre-0.7 versions of `t1lib`.

Concerning glyph-memory considerations, the same applies as said under the description of `T1_SetChar()`: Never free a pointer to memory which was returned by `T1_SetString()`. Or, alternatively, if freeing the pointer cannot be avoided set it to NULL after freeing it.[6]

There are two generic ways how a string-glyph can be produced. The first is to take the paths of all characters needed, concatenate them, insert space and kerning amounts as needed and raster the resulting whole path. This will yield the best results since the average position accuracy of the pixels will be optimal. This applies especially for rotated strings. The drawback is, that every character must be rastered every time it is needed. There is no way to access the bitmap data of a character inside a rastered string separate from others. And the caching of string-glyphs at this programming level doesn't make any sense. So this principle takes significantly longer than concatenating bitmaps from a cache area. However, it is done when the specified rotation angle is not equal to $0°$ or when even further transformation are to be applied. This condition should limit the total number of situations when this happens to an amount we can easily bear.

If the `transform`-argument is `NULL` we know transformations or rotation should not be applied and another approach is used. We are then able to construct the resulting bitmap by adjusting already existent bitmaps into proper positions. If a character does not already exist in the cache, it is generated just the way `T1_SetChar()` works. The calculation of the character-bitmap positions in the output bitmap is done with char space-precision. Nonetheless, there may be differences in the output compared with output of the above method. These are caused by the fact that rounding to pixel accuracy has already been achieved when generating the character bitmap. Thus, the output of the above principle should always be better since the positions of the black pixels are rounded with respect to the whole string, and not with respect to a single character glyph. On the other hand, concatenating character glyphs takes significantly less time than rastering a complete string. Theoretically, differences of up to two pixels horizontal shift may appear in the output of the two principles. You can check the effect by running the program `xglyph`. Specify a string of enough length and raster it at angle $0°$. Then specify a very small angle from 0 different, say, $0.001°$, and raster the string again with the new setting. You might find that the representation of the string is a little different now.

The third function that creates a glyph object is

$\mathcal{F}() \Rightarrow$

```
GLYPH* T1_SetRect( int FontID, float size,
                   float width, float height,
                   T1_TMATRIX *transform)
```

Calling this function yields a filled rectangle. It is intended for situations where a TEX-style `\vrule` would be appropriate, or, for equations in mathematical typesetting.

First, the argument `FontID` identifies a valid font. At this time, it is not obvious why a valid font is required in order to create a rectangular glyph. The reason is, that some of the parameters that are associated to a particular font are also relevant for creating rectangles in the context of that font. Hence, specifying a font identifier in this case is a means of stating "*draw a rectangle that visually fits to the style of the font* `FontID`". Aside from this, I assume

[6]For example, `XDestroyImage()` gives the pixel memory for the image data free although it might not have been allocated by any X11-function.

that each application that uses `t1lib` deals with at least *one* font so that the `FontID` parameter actually does not hurt.

The size of the rectangle is to be specified in charspace units by means of the parameters `width` and `height`, and is moreover subject to scaling through the parameter `size`. By definition, the typographical fundamental area of a font, known as the *em-square*, is just as high and as wide as the design size of the font. In charspace units, this rectangle exactly maps to a 1000×1000 grid. Or alternatively spoken, one charspace unit is 0.001bp (Big Point). For example, in order to draw an em-square for some font at 13 points, the correct parameters are `size` = 13, `width` = 1000 and `height` = 1000. If absolute physical dimensions are desired, the scaling explicitly must be calculated by the user. For instance, in order to produce a 1cm × 1cm square we find

$$h = w = 1\text{cm}\frac{72\text{bp/in} \times 1000\text{cs/bp}}{2.54\text{cm/in}} \approx 28346 \text{ cs.}$$

Here "cs" designates charspace units and `size` is assumed to be 1. Note that still the device resolution must be properly setup in order to make the dimensions appear accurately on the device.

The parameter `transform` is a pointer to a `t1lib` transformation matrix. If it is `NULL`, the current transformation of the font in question is used. Depending on the kind of actual transformation, the rectangle might also image as a nonrectangular shape. This happens, if the current font is slanted, then the rectangle will be skewed.

Glyphs produced by the rectangle function are never cached because there is no means in doing so. With respect to memory management, the same applies as for `T1_SetChar()` and `T1_SetString()`. Since rectangles rarely are used to produce normal text flow, the glyphs produced by the rectangle function do not cause any escapement.

If two glyphs with arbitrary orientation exist,

$\mathcal{F}() \Rightarrow$

```
GLYPH *T1_ConcatGlyphs( GLYPH *glyph1, GLYPH *glyph2,
                        int x_off, int y_off, int modflag)
```

can be used to concatenate them. First the size of the resulting glyph is computed and its metric values are filled. Then, the two glyphs are placed at their appropriate positions in the newly created bitmap. In order to be able to work, the following conditions must be met:

1. Either glyph must be different from `NULL`.
2. Both glyphs must have identical `bpp`-values. If antialiased and non-antialiased glyphs are to be concatenated, have a look at
3. There must be enough memory for the new glyph (naturally).

The quantities `x_off` and `y_off` describe the x- and y-component of an optional offset to be inserted between the two glyphs. This offset is interpreted as device pixels. The `modflag` argument is used to specify the direction in which the two glyphs are to be concatenated. That is, only the bit `T1_LEFT_TO_RIGHT` / `T1_RIGHT_TO_LEFT` is respected by this function.

If problems occur, `NULL` is returned. It is generally not recommended to produce large glyphs with this function because the char space precision in placing the character bitmaps is lost. For example, three times rounding up an advance by 0.3 pixels accumulates to 1 pixel position error. A similar effect shows up when two rotated and underlined glyphs are concatenated with this function. There might be a slight shift in the baseline at the point where the two glyphs touch.

A dilemma occurs, if two antialiased bitmaps have distinct background colors. Then, it is not clear what the transparent color is. `T1_ConcatGlyphs()` always assumes the *current* background color to be transparent.

There is one more function that generates pointers to glyphs:

$\mathcal{F}() \Rightarrow$ `GLYPH *T1_CopyGlyph(GLYPH *glyph)`

As mentioned earlier, the user doesn't have the possibility of keeping the glyphs longer than to the next call of the respective rastering function. If someone wants to keep a bitmap some time longer, `T1_CopyGlyph()` may be used to copy the glyph to another area which is then completely under user's control. This function simply does the following:

- Allocates the memory for the glyph-structure.
- If bitmap data is present, it allocates memory for the bitmap data, taking the member `bpp` into account (see 5.13).
- Copies the structure and the bitmap data to the respective locations.
- Initializes the pointer `glyph.bits`.

Return value is the pointer to the allocated glyph-structure. If something goes wrong, NULL is returned to indicate an error. A glyph pointer, returned by a call to this function should be freed by a call to `T1_FreeGlyph()` (see 5.8).

5.6 Loading Fonts Explicitly

Usually there is no need for a user to load a font into memory since this is done automatically as needed by the rastering functions. But there are two situations where it makes sense to explicitly load a font before generating any size dependent data:

- A font is to be reencoded immediately after loading (see 5.7).
- A font is to be transformed (see 5.11).

These operations require a font being loaded but not having any size specific data. Loading a font explicitly is done by the function

$\mathcal{F}() \Rightarrow$ `int T1_LoadFont( int FontID)`

Loading a font involves several actions:

- Locating and loading the Type 1 font file.
- Locating and loading the font metrics data from AFM file.
- Computing and filling the values of the `FONTPRIVATE` structure as described in section 7.
- Setting up some tables for fast access of metrics information.

`T1_LoadFont()` returns `0` if successful or `-1` if the font could not be loaded. A failure may be due to `t1lib` not having been initialized or due to problems with file locations and file parsing. If a font refuses to load, the logfile should be examined first. Furthermore, in case of a failure `T1_errno` will be set appropriately.

5.7 Functions for Encoding Handling

As mentioned earlier, the encoding mechanism used in the PostScript-language allows a font to contain more than 256 different characters, although only 256 are accessible at a given time. The characters which are accessible are given by the elements of the current *encoding vector*. In order to maximize flexibility, `t1lib` allows for changing the current encoding vector. This is also called "Reencoding a font". A new encoding vector is defined and made known to the library by creating an encoding-file and loading its contents into memory. Before describing the functions needed for this, we should briefly describe the format of an encoding file.

An encoding file is an ASCII text file. No assumptions about filename extensions are made. Here are the rules for scanning the file:

- The file contents are completely ignored until a line is encountered that starts with the string `Encoding=`. This string may optionally be immediately followed by a string that is assumed to be the identifier of the *encoding scheme* that this file defines. Any further text on this line is ignored.
- Now, 256 lines have to follow, each line describing one character's name. The string ranging from the beginning of the line to the first white space character is assumed to be a character name. The remainder of the lines is ignored and may (should) be used for comments, thereby describing the current character code.
- All further lines of text are ignored.

As well known from PostScript, non-existent characters have to be named `.notdef`.

Here's an example of such an encoding file:

```
Sample encoding file for t1lib!
The first two lines are considered to be comments!
Encoding=ISOLatin1Encoding
.notdef                          /* '000  000  "00 */
.notdef                          /* '001  001  "01 */
.notdef                          /* '002  002  "02 */
  .                                   .
  .                                   .
  .                                   .
greater                          /* '076  062  "3E */
question                         /* '077  063  "3F */
at                               /* '100  064  "40 */
A                                /* '101  065  "41 */
B                                /* '102  066  "42 */
  .                                   .
  .                                   .
  .                                   .
yacute                           /* '375  253  "FD */
thorn                            /* '376  254  "FE */
ydieresis                        /* '377  255  "FF */
```

Since V. 1.2, `t1lib` is also able to load encoding files in the format used by `dvips`. This makes a large set of existing encoding files available to the user. When parsing `dvips` encoding files, `t1lib` requires PostScript syntax. This means white space may be interspersed freely and line comments are defined by the character %. The mark-characters, `[` and`]`, are considered as special tokens and need not be preceded or followed by white space. Similarly, the literal escape character / delimits a preceding token without interspersed white space. When parsing dvips encoding files, t1lib tolerates less than 256 character name definitions. If characters are missing, they are substituted by .notdef until the counter reaches 256. Aside from comments, no PostScript tokens are allowed after the encoding definition in a dvips encoding file is complete.

With the defining terms above, it turns out that a file which has successfully been scanned as a dvips encoding file, cannot specify a valid t1lib encoding after the PostScript encoding definition is complete (because no valid character name can start with % and because at least a line such as Encoding=, would have to follow the PostScript encoding). Hence the file format are mutually exclusive and it is possible to read both format using one function. In a first pass t1lib tries to read the file as a dvips encoding file, and if that fails, it assumes to have a t1lib encoding file.

Once such an encoding file of either type has been created, it can be loaded into memory. This is done with the function

$\mathcal{F}() \Rightarrow$ `char **T1_LoadEncoding( char *filename)`

The function will use the search path definitions read from the configuration file during initialization (see 2.3, ENCODING=). If no errors occur, an array of pointers to strings is created and initialized. The start address of this pointer array is returned as a double pointer to a char. This pointer is intended to be used to reencode a font via T1_ReencodeFont(). If the encoding data structure could not be created, NULL is returned to indicate the error.

The memory allocated by T1_LoadEncoding() is organized in two continuous blocks. One block is the pointer array of size 257[7] and the other block contains the character name strings plus the encoding scheme specification, separated by ASCII-zeros. This memory can be returned to the system using the function

$\mathcal{F}() \Rightarrow$ `int T1_DeleteEncoding( char **Encoding)`

t1lib does not check whether a valid pointer value was passed. So be careful to pass the correct pointer. An error in this function should almost always be followed by a segmentation violation.

A newly loaded encoding is applied to an existent font by calling

$\mathcal{F}() \Rightarrow$ `int T1_ReencodeFont( int FontID, char **Encoding)`

FontID must be a valid font identification and Encoding a pointer returned from a successful call to T1_LoadEncoding().

There are two requirements in order to reencode a font:

1. The font must already have been loaded into memory.
2. No size-dependent data exists for this font. If it does, it must be removed explicitly prior to calling T1_ReencodeFont().

It follows that there are two ways to reencode a font. The first is to load a font explicitly and reencode it before any size dependent data is created. The second is to use an automatically loaded font and delete all of its size dependent data before reencoding it.

The user may also specify the special pointer NULL as the Encoding-argument. This would reencode the font to its internal encoding vector.

In case of success, the function returns 0, otherwise -1 is returned.

[7] This number results from 256 charactername pointers plus one pointer to the encoding scheme identifier.

Reencoding a font takes a considerable amount of time since the mapping tables have to be reorganized. In situations where it is à priori foreseeable that the font will be reencoded using some standard encoding vector, it makes sense to assign that particular encoding vector as the default encoding vector, thereby overwriting the internal encoding vector of each font at load time before the mapping tables are setup. Setting the default encoding can be achieved using

$\mathcal{F}() \Rightarrow$
```
int T1_SetDefaultEncoding( char **Encoding)
```

Here `Encoding` encoding is assumed to be a valid `t1lib` encoding vector, e.g., created by a call to `T1_LoadEncoding`. `T1_SetDefaultEncoding()` has to be called after initialization. It returns `0` if this condition is fulfilled and `-1` otherwise. In the latter case `T1_errno` is set appropriately. Notice that the internal encoding of the font is still accessible by reencoding the font using `NULL` as encoding specification (see above). Note further that the default encoding vector is only applied to those font that have `StandardEncoding` as internal encoding. This is to prevent fonts like ZapfDingbats, Symbol or Sonata[8] from being reencoded automatically at load time because this would be surely inappropriate for such fonts.

It is also possible to query the encoding scheme that the font associated with `FontID` uses. This is achieved with the function

$\mathcal{F}() \Rightarrow$
```
char *T1_GetEncodingScheme( int FontID)
```

The return value is a pointer to a string which describes the encoding scheme in question. The are 3 possible cases:

- The font uses Adobe StandardEncoding, which is internally known by the rasterizer. Then, `StandardEncoding` is returned.
- The font defines its own encoding by `dup` *n LiteralName* `put` statements. In this case no particular name is associated with the encoding scheme and `FontSpecific` is returned.
- The encoding is externally loaded by `T1_LoadEncoding()`. Then the encoding scheme entry of this file is returned. If this (optional) entry is not specified in the file, `Unspecified` is returned.

Notice that the name of the encoding scheme is also accessible as `Encoding[256]` where `Encoding` is the pointer returned by a successful call to `T1_LoadEncoding()`.

5.8 Deleting Data

In frequently appearing cases, it may be wise to return some memory which was explicitly or automatically allocated by the library back to the system.[9] For this purpose some functions are available. To understand how size dependent data for a font is organized, see (7).

The memory amount required by the size-dependent data of `size` and font `FontID` is freed by calling the function

$\mathcal{F}() \Rightarrow$
```
int T1_DeleteSize( int FontID, float size)
```

The data deleted includes the metric information for 256 characters, some pointers, associated bitmap data (if already existent) as well as the font matrix for that size.

[8] A musical notation font.

[9] This is especially true, since there is presently no caching algorithm which automatically takes care of this.

As described in section 7, the data structures containing size-dependent information of a particular font are organized as a linked list. `T1_DeleteSize()` takes care that a properly linked list is left after deleting the data.

If the combination of `size` and `FontID` does not exist, -1 is returned. If the operation was successful, the return value is 0.

For the purpose of removing all size-dependent data for a particular font, there is the function

$\mathcal{F}() \Rightarrow$ `int T1_DeleteAllSizes( int FontID)`

It recursively removes all size-dependent data for the font `FontID`. This may be appropriate if a user knows some font not to be needed any longer. This function is also to be used, if one intends to reencode a font for which size dependent data has already been generated. In addition, font transformations such as *slanting* and *extending* require a font having no size-specific data. `T1_DeleteAllSizes()` recursively calls `T1_DeleteSize()` to do its job. It returns the number of sizes removed (including 0 if no sizes were existent) or -1 if an error occurred.

It is also possible to remove the entire data associated with a particular font from memory using

$\mathcal{F}() \Rightarrow$ `int T1_DeleteFont( int FontID)`

`T1_DeleteFont()` goes one step beyond the above functions and removes all the data associated with the font `FontID`. This includes:

- All size dependent data.
- All data from the Type 1 font program, held in memory.
- All AFM data kept in memory.

The memory reserved for a font in hierarchy-level 1 is not returned to the system since it is simply one element in the array of structures of type `FONTPRIVATE` (see 7). But all entries in this structure are reset to initial values.

Whether it is useful or not, a font that has been removed using `T1_DeleteFont()` may also be loaded again, explicitly or implicitly.

There is a restriction, which has not yet been mentioned: A font may only be removed if it is a physical font (to be explained later) to which no logical fonts refer or if it is a logical font.[10] A reference counter is maintained in each physical font to check for this. If the font to be removed is a logical font, the `FONTPRIVATE` area is reset and the reference counter of the referenced physical font is decremented. Of course, size dependent data is removed in every case.

`T1_DeleteFont()` returns 0 if the font has been removed correctly or if the font was not loaded. n (> 0) is returned if the font was physical and was referenced by n logical fonts. A return value -1 indicates an invalid `FontID`.

The function

$\mathcal{F}() \Rightarrow$ `int T1_FreeGlyph( GLYPH *glyph)`

returns memory allocated by `T1_CopyGlyph()` back to the system. This function should not be applied to the pointer to a glyph returned by one of the rastering functions. As said earlier, these functions manage the memory areas by themselves.

Similarly, the function

[10] See 5.15 for explanation of logical fonts.

$\mathcal{F}() \Rightarrow$	`int T1_FreeCompCharData( T1_COMP_CHAR_INFO *cci)`

returns memory associated to the composite caracter data structures, as allocated and returned by `T1_GetCompCharData()` or `T1_GetCompCharDataByIndex()` (see Section 5.18), to the system. In order to avoid memory leaks, each call of the latter two functions should be followed by a call to this function.

In order to close the library and return all memory to the system,

$\mathcal{F}() \Rightarrow$	`int T1_CloseLib( void)`

should be used. If no problems occur, 0 is returned. The value 1 indicates problems during freeing data. In this case the logfile should be examined. After having freed all data the file search paths, if different from the defaults, are restored. Last, the logfile is closed.

5.9 Underlining, Overlining and Overstriking

`t1lib` supports underlining, overlining and overstriking for the string rastering functions. These lines are always drawn on the fly as the bitmaps are generated. In writing direction, the lines range from the glyph's origin to the glyph's width. The vertical dimensions are set the following way by default when a font is loaded:

- **Underlining**: Underline position and thickness are taken from the Fontinfo dictionary of the respective font. The rule is vertically centered with respect to the mathematical line given by the position value.
- **Overlining**: The position is computed to be $y = a + |u|$ where a corresponds to the typographic ascender and u is the underline position from the Fontinfo dictionary. As above, the rule is vertically centered around this position value. The thickness is set to underline thickness.
- **Overstriking**: The position is $y = a/2$ where again a is the typographic ascender of the font and vertical alignment of the rule is done by centering around the computed position. The thickness is set to underline thickness.

As all information in AFM files, thickness and position specifications are interpreted in charspace units.

Notice that the typographic ascender is not determined by the Type 1 font program. It has to be guessed by `t1lib`. The problem of guessing the typographic ascender is discussed in more detail in 5.16.3. When loading a font, this typographic ascender is assumed to be the vertical coordinate of the upper right corner of the bounding box of the letter "d". This is not as advanced as the procedure described in 5.16.3, but it suffices because the underlining positions can later be overwritten by the user (see below).

From the mathematical point of view, the line rules are an integral part of the rastered path. It follows that line rules may appear sheared if a font has been artificially slanted and the size and/or thickness is sufficiently large.

A look into real Type 1 font files shows that even fonts of the same family possess incompatible values for underlining. For example, Bitstream Charter Roman defines underline thickness to be 61 and its bold variant assigns a value of 90. Underlining text consisting of Roman and bold words will not be very pleasing using these values. For this reason `t1lib` provides a way for explicitly setting and overwriting the default values for line ruling on a per-font level. The functions

$\boxed{\mathcal{F}() \Rightarrow}$ `int T1_SetLinePosition( int FontID, int linetype, float value)`

and

$\boxed{\mathcal{F}() \Rightarrow}$ `int T1_SetLineThickness( int FontID, int linetype, float value)`

set the respective value for font `FontID` to `value`. The `linetype` argument is assumed to be an OR'ed combination of `T1_UNDERLINE`, `T1_OVERLINE` and `T1_OVERSTRIKE`. While it generally does not make sense to specify identical positions for two or three distinct line rule types, it is meaningful to specify identical thickness values for two or all rules types. However both functions accept combinations of linetype specification. It follows that consistent line ruling for several fonts can be achieved by setting the line rule parameters of the involved fonts to identical respective values.

Currently active line rule parameters can be queried using the functions

$\boxed{\mathcal{F}() \Rightarrow}$ `float T1_GetLinePosition( int FontID, int linetype)`

and

$\boxed{\mathcal{F}() \Rightarrow}$ `float T1_GetLineThickness( int FontID, int linetype)`

In case more than one line rule type is specified for `linetype` the first matching value is returned, since obviously the functions can only return *one* value. The order the argument is checked is `T1_UNDERLINE`, `T1_OVERLINE` and finally `T1_OVERSTRIKE`.

These functions called with `T1_UNDERLINE` as line type argument should not be confused with the functions `T1_GetUnderlinePosition()` and `T1_GetUnderlineThickness()` respectively. The latter functions will always return the values from the Fontinfo dictionary as opposed to the former which will return the currently active values.

Since line ruling is done on the fly, it is possible to change the involved parameters in the middle of a session without confusing the cache or removing size dependent data.

5.10 Common Information on Fonts and Characters

This subsection describes some functions making common information available. This includes Type 1 and AFM data. Thus, these functions partially depend on the existence of AFM data. In order not to have to specify this data every time, here are a few conventions:

1. `FontID` is the valid ID of a declared font.
2. All functions that require a character index as argument use the currently active encoding vector to determine the character's name belonging to this index and use the character's name to search for the information required.
3. Some functions do not allow to use the return value for error checking. For this reason every function described in this subsection will set `T1_errno` appropriately if something goes wrong. See 5.19 for the description of the possible values of `T1_errno`.
4. None of the functions described in this subsection will load a font automatically.

5.10.1 Information from FontInfo-Dictionary

$\mathcal{F}() \Rightarrow$ `char *T1_GetFontName( int FontID)`

This function returns the string object `FontName` from the fontinfo-dictionary of the specified font or a NULL pointer if the font is not loaded.

The memory for the returned string is static in this function and should thus not be freed by the user. As another consequence, the returned string is only constant until the function is called the next time.

$\mathcal{F}() \Rightarrow$ `char *T1_GetFullName( int FontID)`

This function returns the string object `FullName` from the fontinfo-dictionary of the specified font or a NULL pointer if the font is not loaded.

The memory for the returned string is static in this function and should thus not be freed by the user. As another consequence, the returned string is only constant until the function is called the next time.

$\mathcal{F}() \Rightarrow$ `char *T1_GetFamilyName( int FontID)`

This function returns the string object `FamilyName` from the fontinfo-dictionary of the specified font or a NULL pointer if the font is not loaded.

The memory for the returned string is static in this function and should thus not be freed by the user. As another consequence, the returned string is only constant until the function is called the next time.

$\mathcal{F}() \Rightarrow$ `char *T1_GetWeight( int FontID)`

It returns the Weight entry from fontinfo dictionary. It is a string entry and represents a verbatim classification of the font rather than a numerical quantity. In case of an error `NULL` is returned.

$\mathcal{F}() \Rightarrow$ `float T1_GetItalicAngle( int FontID)`

The returned value is the italic angle of the font in degrees as a float. Notice that the meaning of ItalicAngle is related to the slanting of fonts, but not in the sense of `t1lib` (see 5.11). An italic font may be artificially slanted and an artificially slanted font in the sense of `t1lib` may have an italic angle of zero.

$\mathcal{F}() \Rightarrow$ `int T1_GetIsFixedPitch( int FontID)`

This function returns 0 if the font's spacing is proportional and 1 if it is fixed.

$\mathcal{F}() \Rightarrow$ `BBox T1_GetFontBBox( int FontID)`

This function returns the bounding box of the font identified by `FontID`. It is the bounding box that results if all characters of a font are overlayed with their reference point falling on the point (0,0). All values are in charspace units. The members `lly` and `urx` represent the fonts overall descent and ascent, respectively.

The font's bounding box is part of the AFM information as well as member in the font's private dictionary. It turns out that the information from `.afm`- and `.pfa`/`.pfb`-file is not consistent for some fonts. `t1lib` returns the information stored in the font-file itself, since I assume it is more consistent to the font's data.

$\mathcal{F}() \Rightarrow$ `float T1_GetUnderlinePosition( int FontID)`

This function returns the underline position of the specified font as given in the fontinfo-dictionary. The value is to be interpreted in charspace units. If the font is not loaded, 0 is returned since an underline position of 0 can be considered impossible for most fonts.

$\mathcal{F}() \Rightarrow$ `float T1_GetUnderlineThickness( int FontID)`

This function returns the thickness of the underlining rule for this font or 0 if the font is not loaded. 0 is a safe index for an error since a rule of height 0 would not be visible anyhow.

$\mathcal{F}() \Rightarrow$ `char *T1_GetVersion( int FontID)`

The version string from the Type 1 font file is returned. The memory where the string is located is managed by the function itself.

$\mathcal{F}() \Rightarrow$ `char *T1_GetNotice( int FontID)`

The notice string from the Type 1 font file is returned. Again the user should not touch the memory where the string is located.

5.10.2 Metric Information on Glyphs

$\mathcal{F}() \Rightarrow$ `int T1_GetCharWidth( int FontID, char char1)`

The character width according to the AFM information is returned in charspace units. If no AFM information is available, 0 is returned.

The width of a character is the amount of horizontal escapement that the next character is shifted to the right with respect to the current position. This information is not given in the character's bounding box. Also, the width corresponds to the entry `characterWidth` in the `glyph`-structure, as described in 5.5. But since `T1_GetCharWidth()`, returns its result in charspace units, the accuracy is much higher than using the value of the `glyph`-structure which has only pixel-accuracy.

If there is an extension specified for the font in question, the characters width is corrected correspondingly.

$\mathcal{F}() \Rightarrow$ `BBox T1_GetCharBBox( int FontID, char char1)`

The character's bounding box of `char1` is returned with the elements to be interpreted in charspace units. The bounding box of a character is defined to be smallest rectangle aligned parallel to the x- and y-axis of the character coordinate system which encloses the painted area of the character completely. This rectangle is completely specified by specifying its lower left and its upper right corner. From a programmer's point of view, a characters bounding box is defined by the following struct of type `BBox`:

```
typedef struct
{
   int llx;     /* lower left x-position  */
   int lly;     /* lower left y-position  */
```

```
   int urx;     /* upper right x-position */
   int ury;     /* upper right y-position */
} BBox;
```

In case the character is not encoded or no AFM data is available, a box containing only zeros is returned.

The bounding box is corrected if an extension value has been applied to the font in question.

Since version 0.3-beta, slanted fonts are fully supported, meaning that for slanted fonts too a correct bounding box will be returned. This is however quite time expensive since the characters' real outline must be considered. See the discussion on slanting a font (5.11) for an explanation of this.

$\boxed{\mathcal{F}() \Rightarrow}$

```
int T1_GetStringWidth( int FontID, char *string,
                       int len, long spaceoff, int kerning)
```

$\boxed{\mathcal{F}() \Rightarrow}$

```
BBox T1_GetStringBBox( int FontID, char *string,
                       int len, long spaceoff, int kerning)
```

These two functions represent the complement to the above functions on the level of strings. All parameters that take influence on the resulting width and bounding box must be given in the argument list. Their meaning is identical to the meaning they have when calling string rastering functions (see 5.5).

$\boxed{\mathcal{F}() \Rightarrow}$

```
METRICSINFO T1_GetMetricsInfo( int FontID, char *string,
                               int len, long spaceoff, int kerning)
```

In certain situations bounding box and width of a glyph are required both. In these cases it is more convenient to call `T1_GetMetricsInfo()` which returns a structure that contains all information. `METRICSINFO` is defined in `t1lib.h` as:

```
typedef struct
{
  int      width;
  BBox     bbox;
  int      numchars;
  int      *charpos;
} METRICSINFO;
```

All numbers are to be interpreted in character space units — they are directly taken from AFM data. `width` is the glyph's width and `bbox` its bounding box which in turn is a struct as defined some paragraphs above.

`numchars` is assigned number of characters in string. If the argument `len` is different from 0, `numchars` is assigned that value.

`charpos` is a pointer to an integer array of size `numchars` allocated by `T1_GetMetricsInfo()`. During execution this array is filled step by step with the horizontal escapement in character space units of the respective character relative to the start point of the string glyph which corresponds to 0. `charpos` remains valid until `T1_GetMetricsInfo()` is called the next time. The user should not free this memory because this is handled automatically.

The terms concerning the bounding box of slanted fonts mentioned under the description of `T1_GetCharBBox()` apply here as well. The first and the last character of `string` have to be observed spending high effort. But nevertheless the correct bounding box is returned.

$\mathcal{F}() \Rightarrow$ `int T1_GetKerning( int FontID, char char1, char char2)`

This function returns the amount of kerning for the specified character pair `char1` and `char2`. If an extension has been specified for the font (see 5.11), the amount of kerning is automatically corrected using the extension factor. The value returned has to be interpreted in charspace units.

If no AFM information is available for the font in question, simply 0 is returned. The same applies if the font is not loaded.

The implementation of this function requires that the kerning pairs in the AFM file are sorted in alphabetical order. I am not sure whether this condition is found in the specification of the AFM file format. If this function doesn't work although AFM kerning data is available, this might be the reason.

$\mathcal{F}() \Rightarrow$ `int T1_QueryLigs( int FontID, char char1, char **successors, char **ligatures)`

This function implements the interface to the ligature information in the AFM data. Ligatures are special character-symbols which are substituted if special pairs, triples or whatever groups of characters appear in a string. For example, "fi" is replaced with the ligature "fi". In this example, the "i" is called *successor* and the "fi" is the associated ligature.

`char1` is the character which has to be checked for ligatures, i.e., the first character of a possible ligature group. `successors` and `ligatures` should be addresses of pointers to `char`s. These pointers are modified by the `T1_QueryLigs()`.

First, `T1_QueryLigs()` checks how many ligatures are defined for the character given by `char1`. Assuming this number is n, it then defines memory for two arrays of type `char` with size n. These arrays are filled with the indices of the successor-characters and with the indices of the associated ligatures, respectively. The current encoding vector is used for this. The addresses of these two arrays are written to the addresses of the respective pointers `successors` and `ligatures`. They are thus later available to the user in order to access the memory where the successor-character and ligatures are specified. The value n is returned in order to tell the user how many ligatures were found and to give the user information about the end of the two arrays.

If the font is not loaded or AFM data is not available, -1 is returned.

Since this may seem to be a little complicated, here is a programming example:

```
char *succ, *lig;
int n_lig, i;
char char1='f';

/* Get ligature information of character 'f' in font 0: */
n_lig=T1_QueryLig( 0, char1, &suc, &lig);

/* print out indices of characters and their ligatures */
for ( i=0; i<n_lig; i++;){
  printf("First char: %d, +  next char: %d --> ligatur: %d\n",
         char1,
         succ[i],
         lig[i]);
```

Notice that the arrays where the successor indices and the respective ligature indices are stored are static in T1_QueryLigs(). Thus, they may not be freed and moreover they are only valid until the next time T1_QueryLigs() is called.

5.10.3 Character-Encoding Relation

$\mathcal{F}() \Rightarrow$ `char *T1_GetCharName( int FontID, char char1)`

This function returns the name of the character indexed by char1 according to the current encoding vector. As said above, the memory where the string is stored is static to this function so that the user should not free the returned pointer. If the font is not loaded, NULL is returned.

$\mathcal{F}() \Rightarrow$ `int T1_GetEncodingIndex( int FontID, char *char1)`

This function is the complement to the above function. It returns the lowest index of the character with the specified name in the current encoding vector as an int. If the charactername is not found in the current encoding vector or if the font is not loaded, the value -1 is returned.

$\mathcal{F}() \Rightarrow$ `int *T1_GetEncodingIndices( int FontID, char *char1)`

This function is an extension of the above function. It is meant for cases where all appearences of char1 in the encoding vector are requested. The return value is an integer array whose elements contain the encoding indices where char1 appears in increasing order. The first negative entry in this array indicates that no more valid indices will follow. In the most extreme case we can think of (i.e., where the complete encoding vector is occupied by *one* character), T1_GetEncodingIndices() will return an array of size 257 where the first 256 elements bear their own index and the 257th element is -1. If font FontID is not loaded, this function returns NULL and T1_errno will be set appropriately. If char1 was not found in the encoding vector, a valid array is returned but the first element will be -1.

$\mathcal{F}() \Rightarrow$ `char **T1_GetAllCharNames( int FontID)`

As described in 5.7, not all characters of a font need to be encoded. A Type 1 may contain the outlines of an arbitrary number of characters, but only 256 can be encoded—and thus accessed—simultaneously. Since the characternames are inside the encrypted portion of the Type 1 font file, there is no easy way to find out which characters a font defines.

t1lib provides T1_GetAllCharNames() for situations where a programmer needs to know what characters are defined in the font file identified by FontID. The value returned is a pointer to an array of char pointers which in turn point to the characternames. The array's size is $(n + 1)$ where n is the number of defined outlines. The $(n + 1)$th pointer is NULL to indicate the end of the array. An application programmer may use these characternames to construct a specialized encoding vector. Here is an example of how to use T1_GetAllCharNames(). It prints a list of all defined characternames in font 0.

```
char **ptr;
int i;
.
.
.
ptr=T1_GetAllCharNames( 0);
i=0;
while (ptr[i]!=NULL){
```

```
    printf("Charstring %d = %s\n", i, ptr[i]);
    i++;
  }
```

The memory for storing the pointers and the charactername strings is static in `T1_GetAllCharNames()`. Thus it remains valid until the function is called the next time. The user should not free this memory or if he does, he should set the pointer to `NULL` to indicate the memory has already been freed.

$\mathcal{F}() \Rightarrow$ `int T1_GetNoKernPairs( int FontID)`

This function returns the number of kerning pairs defined for the font identified by `FontID`. The number -1 is returned if an error occured and `T1_errno` will be set. All positive numbers including 0 should be considered valid return values.

5.10.4 Administrative Information

$\mathcal{F}() \Rightarrow$ `int T1_GetNoFonts( void)`

Usually, this function returns the number of fonts declared in the font database file, i.e., the integer quantity from the first line of the font database file. However, if some new fonts have been created using `T1_CopyFont()` (see 5.15) or if some fonts have been added to the database after initialization (see 5.3.3), these are also taken into account. The number returned by `T1_GetNoFonts()` minus 1 is thus the largest valid font ID specification.

$\mathcal{F}() \Rightarrow$ `int T1_CheckForInit( void)`

Use this function in order to check whether `t1lib` is initialized. It returns 0 if initialization has already happened and -1 otherwise.

$\mathcal{F}() \Rightarrow$ `int T1_CheckForFontID( int FontID)`

This functions gives information on the load status of the font associated to `FontID`. It returns 0 if the font `FontID` has not yet been loaded, 1 if it has already been loaded. Finally, a return value of -1 indicates that either `FontID` is an invalid specification or `t1lib` is not initialized.

$\mathcal{F}() \Rightarrow$ `char *T1_GetFontFileName( int FontID)`

This function returns a pointer to the fontfilename identified by `FontID`. In no case, this pointer may be freed since the memory is static to this function. The string also is only valid up to the next call of this function.

$\mathcal{F}() \Rightarrow$ `char *T1_GetFontFilePath( int FontID)`

This function returns a pointer to the fully qualified path of the font file identified by `FontID`. In no case, this pointer may be freed since the memory is static to this function. The string also is only valid up to the next call of this function.

$\mathcal{F}() \Rightarrow$ `char *T1_GetAFMFilePath( int FontID)`

This function returns a pointer to the fully qualified path of the AFM file of the font identified by `FontID`, as used by `t1lib`. In case of an error `NULL` is returned. It may also happen that there exists no AFM file for the font either because AFM information was generated on the fly at the time the font was loaded, or because AFM processing had been disabled at initialization time. For those cases `T1_errno` is not set.

In no case, the returned pointer may be freed since the memory is static to this function. The string also is only valid up to the next call of this function.

$\mathcal{F}() \Rightarrow$ `char *T1_GetLibIdent( void)`

This function returns the identifier string for the version of `t1lib`. For example, this could be `0.8-beta`. The string is static in this function and should thus not be freed by the user.

5.11 Transformation of Fonts

Transformation of Type 1 fonts is generally accomplished by means of concatenating so-called transformation matrices. For example, rotation is equivalent to concatenation of the standard transformation matrix with a special matrix whose elements are trigonometric functions evaluated at the rotation angle. In the sense of `t1lib`, we distinguish between *fontlevel transformations* and *characterlevel transformations*.

5.11.1 Fontlevel Transformations

`t1lib` supports three transformations that operate globally on a font's data. After applying such a transformation to a font all characters generated from then on will be rendered according to that transformation. Moreover, these transformed characters are saved in cache for fast future access. This principle is thus meant for transformed fonts which are semantically used as ordinary text fonts. Creating a font Times-Oblique by slanting a Times-Roman would be a typical example.

The first fontlevel transformation is called "extension" since it extents a font horizontally—makes its characters wider. A font is extended by a call to the function

$\mathcal{F}() \Rightarrow$ `int T1_ExtendFont( int FontID, double extend)`

A font that is to be extended may not have size dependent data. If size dependent data exists, it must explicitly be removed before applying an extension-factor. This is simply a security mechanism which prevents the user from mixing up extended and non-extended bitmaps. If the font is not loaded or size-dependent data is existent, -1 is returned. Otherwise, the function returns 0.

All information on character metrics is automatically adapted to an extension-factor different from 1 (see 5.10).

Applying an extension-factor to a font is implemented by replacing the current extension-factor—initially 1—with the supplied value. Thus, an extension can be deleted by specifying a factor 1. Moreover, extending a font two times, say, with factor 2, does not yield a font extended by 4. Rather the last specified extension, here 2, is applied.

The second type of fontlevel transformation supported by `t1lib` is *slanting*. It is done by a call to the function

$\mathcal{F}() \Rightarrow$ `int T1_SlantFont( int FontID, double slant)`

The slant-factor s tells the rastering algorithm to advance the x-coordinate of a given point by the product of s with the y-coordinate of that point. Such fonts are sometimes called *oblique*.

Another interpretation is that we state: $s = \tan(\alpha)$, where α is the well-known italic-angle of the font.

Just as above, no size-dependent data may be existent and the font must be loaded. In that case 0 is returned, otherwise -1.

As above, the slanting operation is implemented by *setting* the slant-factor so that a slant may be reset by means of specifying a slant-factor of 0.

There is one thing that makes handling of slanted fonts more difficult than handling of extended fonts. When typesetting strings by concatenating bitmaps, exact information on character metrics is necessary. By slanting a character the character's width is not affected. But the bounding box is. And while extension—which means strictly horizontal scaling independent of the respective y-coordinate—simply leads to an extension of the bounding box, there is no way to compute the bounding box of a slanted character from the bounding box of the respective unslanted character. Here is an example.

- Let the character be `\`. When slanting this character with a value of 1, the resulting character will be similar to a vertical line. The bounding box will thus be small in horizontal direction.

- If character is `/`, the resulting slanted character will tend to be more horizontal. Thus the resulting bounding box will be much extended in horizontal direction.

In conclusion we can say that the effect of slanting on the bounding box of a given character depends on the shape of the character itself.

Since version 0.3-beta the problem with the bounding box of slanted characters is handled as follows. The character in question is internally rastered at 1000 bp and the bounding box of the resulting "edgelist" is examined. But no bitmap is generated for the character, this limits the computational effort. However the difference in time performance between getting a bounding box from a "simple-shaped" slanted character like "i" and getting a bounding box of a "complex-shaped" character like "Q" is clearly noticeable.

The positioning algorithm for string bitmaps has been slightly improved in `t1lib` V. 0.3-beta so that now exclusively bitmap metrics are used where the bounding boxes are needed. The limitation of slanted fonts appears thus only if a user explicitly requests a bounding box of a character/string in an artificially slanted font.

The third and most common type of fontlevel transformation allows arbitrary linear transformations. This is done by a call to the function

$\mathcal{F}() \Rightarrow$ `int T1_TransformFont( int FontID, T1_TMATRIX *matrix)`

The transformation is specified by `matrix` (as described below). This function acts by setting the font's transformation matrix to the matrix pointed to by `matrix`.

As a final consequence of what has been described so far in this section it turns out that `T1_TransformFont()` overrides whatever slant and extension values might have been set before. Conversely, if `T1_SlantFont()` or `T1_ExtendFont()` are applied to a font after a call to `T1_TransformFont()`, the respective values are simply overridden, there will be no concatenation. In the following description of transformation matrices and their usage, we will also describe how to concatenate an arbitrary series of linear transformations.

There are also functions for querying the current values of the quantities described above for fontlevel transformations.

$\mathcal{F}() \Rightarrow$ `double T1_GetExtend( int FontID)`

and

$\mathcal{F}() \Rightarrow$ `double T1_GetSlant( int FontID)`

return the current extension and slant values. The function

$\mathcal{F}() \Rightarrow$ `T1_TMATRIX T1_GetTransform( int FontID)`

Returns the current transformation matrix as a structure of type `T1_TMATRIX` which will be described in detail in the next subsubsection.

5.11.2 Transformation at Rastering Time

This kind of transformation is the most generic one and allows arbitrary transformations. A transformation $(x', y')^T$ of a given location $(x, y)^T$ is given by the following set of linear equations:

$$\begin{aligned} x' &= a_{11}x + a_{21}y \\ y' &= a_{12}x + a_{22}y \end{aligned}$$

Here, the matrix

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \end{pmatrix}$$

is referred to as the *transformation matrix*. In `t1lib` the following type definition is used for such transformation matrices:

```
typedef struct
{
  double cxx;  /* corresponds to a11 */
  double cyx;  /* corresponds to a21 */
  double cxy;  /* corresponds to a12 */
  double cyy;  /* corresponds to a22 */
} T1_TMATRIX;
```

Each of the rastering functions expects to get a pointer to an object of type `T1_MATRIX`, or `NULL` if no transformation is to be applied. If any transformation has been specified, the resulting glyph is never kept in cache memory. Thus, if for some reason caching should be disabled for non-transformed characters, simply a pointer to the unity matrix could be specified to the rastering function to achieve this.

The user has the possibility of either allocating and creating the transformation matrices by himself or to use predefined functions of `t1lib`. There are 8 different functions for generating transformed characters. Figure 1 gives an example of each function using the character "g".

Before describing each particular function we should discuss the first argument because this is common to all matrix transformation functions. This first argument, in case it is not `NULL`, is expected to be a pointer to an already existent valid `T1_TMATRIX` object. The transformation to be applied is then done by multiplying the existent matrix with the new matrix. In other words, the existent matrix is replaced by the concatenation of the two matrices. If a `NULL` is specified as argument, the new matrix is allocated by the respective function and then set to the concatenation of the unity matrix with the desired transformation. Thus, to remove a matrix from memory, the pointer simply has to be given to `free()`, no matter how many transformations have been applied to this matrix before.

We should now describe the functions for generating transformation matrices:

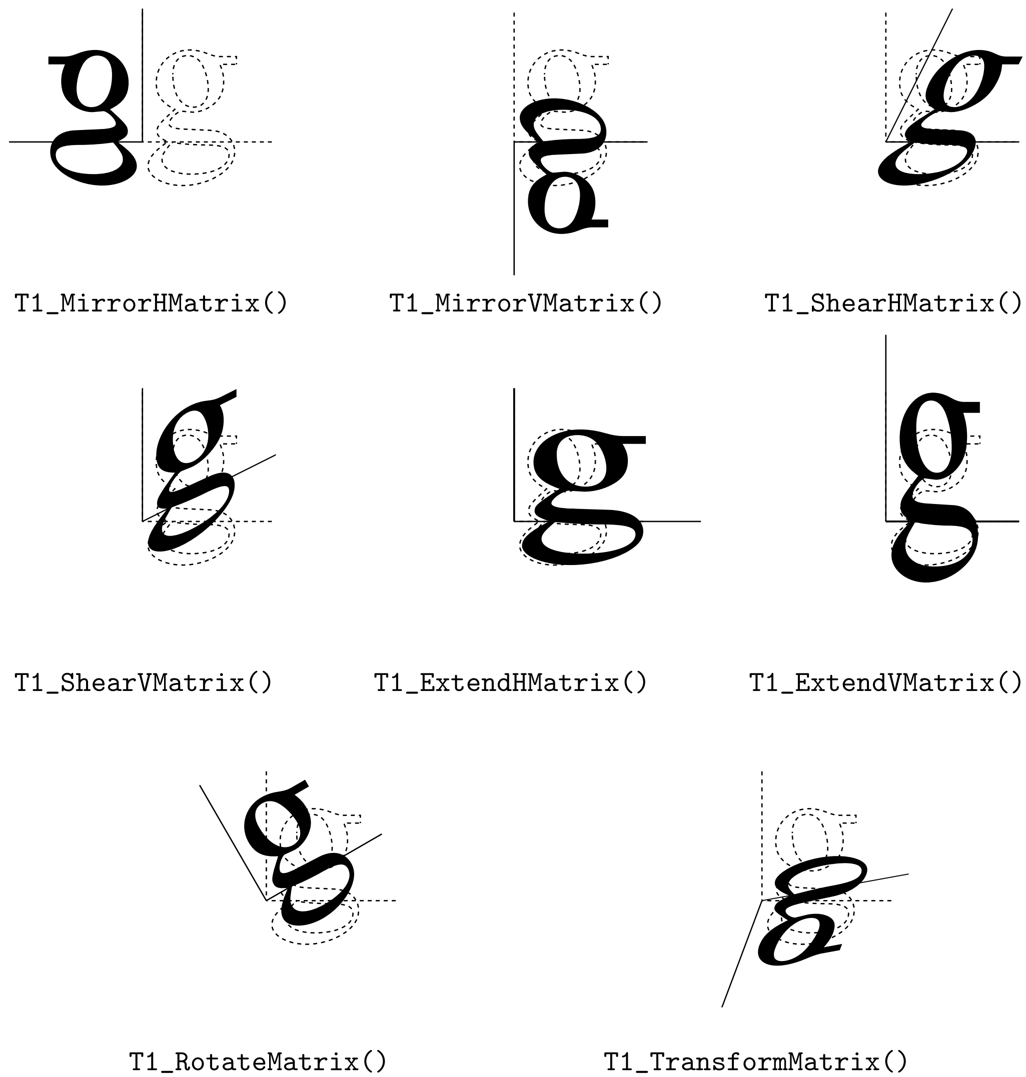

Figure 1: Typical examples for the predefined functions for generating transformation matrices in `t1lib`, applied to the character “g”.

$\mathcal{F}() \Rightarrow$ `T1_TMATRIX *T1_MirrorHMatrix( T1_TMATRIX *matrix)`

and

$\mathcal{F}() \Rightarrow$ `T1_TMATRIX *T1_MirrorVMatrix( T1_TMATRIX *matrix)`

simply change the sign of the matrix coefficients a_{11} and a_{22} respectively. This has the optical effect of mirroring the character at the horizontal line $y = 0$ or at the vertical line $x = 0$, respectively. These functions represent a specialized form of

$\mathcal{F}() \Rightarrow$ `T1_TMATRIX *T1_ExtendHMatrix( T1_TMATRIX *matrix, float extent)`

and

$\mathcal{F}() \Rightarrow$ `T1_TMATRIX *T1_ExtendVMatrix( T1_TMATRIX *matrix, float extent)`

These functions allow arbitrary scaling in the respective coordinate direction. Specifying `extent=-1` exactly yields mirroring at the corresponding axis.

Furthermore, there are two transformations where one coordinate depends on itself and on the other coordinate. This is called shearing, slanting or also obliqueing. It is possible in both directions using the functions

$\mathcal{F}() \Rightarrow$ `T1_TMATRIX *T1_ShearHMatrix( T1_TMATRIX *matrix, float shear)`

and

$\mathcal{F}() \Rightarrow$ `T1_TMATRIX *T1_ShearVMatrix( T1_TMATRIX *matrix, float shear)`

In case of horizontal shearing, the factor `shear` is equal to $\tan(\alpha)$, where α may be interpreted as the italic angle. It is measured from the positive vertical axis in mathematical negative direction. Correspondingly, for vertical shearing `shear` equals $\tan(\beta)$, where β is the angle measured from the horizontal axis in mathematically positive direction.

Rotation of glyphs is achieved using

$\mathcal{F}() \Rightarrow$ `T1_TMATRIX *T1_RotateMatrix( T1_TMATRIX *matrix, float angle)`

This function evaluates the trigonometric functions at the value of `angle` and concatenates the transformation matrix with

$$\begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix}$$

α is expected to be specified in degrees. It is measured according standard mathematical conventions.

There is one more function which allows to set all matrix coefficients explicitly. It gives thus complete control over the transformation. This might be necessary to typeset text in a circle, for example. The syntax of this function is

$\mathcal{F}() \Rightarrow$

```
T1_TMATRIX *T1_TransformMatrix( T1_TMATRIX *matrix,
                                float cxx, float cyx,
                                float cxy, float cyy)
```

5.11.3 `t1lib` and PostScript: Notes on Transformation Matrices

In order to avoid confusion about transformation matrices, we should briefly discuss the differences between `t1lib`- and PostScript transformation matrices. In `t1lib`-nomenclature a coordinate description is assumed to be represented by a column vector $(x, y)^T$. In contrast, PostScript assumes a coordinate to be represented by a row vector (x, y). This leads to an exchanged meaning of the second and third matrix element between `t1lib` and PostScript. From the mathematical point of view this is caused by matrix transposition. To make this clear, let me quote the matrix

$$\mathbf{A}_{\text{PostScript}} = \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ t_x & t_y & 1 \end{pmatrix}$$

from the PostScript Language Reference Manual (Adobe, Red Book). If we forget about translation which in this sense is not implemented by `t1lib`, we only have to consider the top left submatrix consisting of a, b, c and d. The `t1lib`-equivalent to this matrix would be written as

$$\mathbf{A}_{\texttt{t1lib}} = \begin{pmatrix} a & c \\ b & d \end{pmatrix}$$

I.e., the meaning of b and c is exchanged. Notice that font matrices as found in Type 1 font files have to be interpreted according to the PostScript notation. But a user should never come close to something other than the `t1lib` transformation matrices

5.12 Stroking of Character Outlines

Most available Type 1 fonts internally specify the numeric value `PaintType` as 0. This indicates that the mathematical outline definitions of the characters consist of paths that represent the left and right—or inner and outer—borders of the character area to be filled with ink. In particular, this means that every subpath of a character definition must be closed. When filling characters, the curves that describe the outline are infinitely thin and the area between paths is of interest.

By contrast, there are fonts that specify `PaintType` as 1. These are also known as *stroked fonts*. Basically, their characters' shapes are generated by travelling along the mathematically thin defining curves using a pen with some finite width. The penwidth used here usually is specified in the font itself by means of the quantity `StrokeWidth`.

Using a somewhat unusual implementation, `t1lib` can

- image stroked fonts with `PaintType` 1,
- stroke fonts that have `PaintType` 0,
- deal with variable strokewidths for both paint types.

In particular at low resolutions and when using large penwidths, the results of stroking may fall short of the quality of other general graphics tools like e.g. native PostScript interpreters. Also hinting generally is disabled when creating stroked character outlines. For the interested reader, the implementation of stroking is outlined in 8 (page 94).

The process of stroking in `t1lib` is treated similar to the process of transformation. Therefore, firstly, a font can be generally switched to be a stroked font. This is achieved using

$\mathcal{F}() \Rightarrow$ `int T1_StrokeFont( int FontID, int dostroke)`

Here, `FontID` identifies the font to be switched. The parameter `dostroke` determines the future rastering behavior. If it equals 0, the font's characters will be filled and for any other value the characters will be stroked. The function returns 0 in case of success. `T1_StrokeFont()` requires the font `FontID` to be loaded. If the font is not loaded `T1_errno` will be set appropriately and the function returns -1. After switching a font to *stroked*, its characters are cached as usual. It is therefore important that, at the time where `T1_StrokeFont()` is called, no size-dependent data does exist. If this condition is violated, the function will fail too and set `T1_errno` to `T1ERR_OP_NOT_PERMITTED`.

The second method to control the rasterizer mode for the font `FontID` is to use the function pair

$\mathcal{F}() \Rightarrow$ `int T1_SetStrokeFlag( int FontID)`

and

$\mathcal{F}() \Rightarrow$ `int T1_ClearStrokeFlag( int FontID)`

for enabling and disabling stroking, respectively. These two functions return 0 on successful completion and -1 in case of a failure. They can only fail if FontID is not valid, in which case T1_errno is set to T1ERR_INVALID_FONTID. Using these functions, switching the stroking is hence possible at any time. The netto effect is that caching will be disabled if the actually selected rasterizer mode does not match the one of the cached characters. The performance then will degrade for the rasterizer mode that is not the base mode of the font in question.

If stroked characters and filled characters are required for one particular font while preserving good performance, using logical fonts is the tool of choice (see Sec. 5.15, page 70).

The currently active rasterizer mode for the font FontID can also be queried. This is done using

$\mathcal{F}() \Rightarrow$ `int T1_GetStrokeMode( int FontID)`

If FontID is not valid, -1 is returned and T1_errno is set to T1ERR_INVALID_FONTID. In any other case, the following values may be returned:

- 0: The rasterizer mode is *Fill* and filled characters are cached. For stroked fonts with a PaintType of 1, of course, the characters are stroked and these stroked characters are cached anyhow.
- 1: Stroking is enabled, while filled characters are cached. Each rasterization request is performed on the fly, as for nontrivial transformations.
- 2: Filling is enabled but stroked characters are cached. Rasterization again is performed on the fly.
- 3: Stroking is enabled and stroked characters are cached.

The width of the virtual pen used to trace the character outlines can also be adjusted. The function

$\mathcal{F}() \Rightarrow$ `int T1_SetStrokeWidth( int FontID, float strokewidth)`

sets up strokewidth as the pen width used to stroke characters from font FontID. It is interpreted in charspace units and hence directly comparable to the StrokeWidth parameter of stroked fonts. If this function succeeds, 0 is returned. The function call might also be unsuccessful, in which case -1 is returned. The value of T1_errno then gives a hint to the reason of the failure. If set to T1ERR_INVALID_FONTID, the font was not loaded. The value T1ERR_INVALID_PARAMETER indicates that a negative strokewidths has been specified. Finally, T1ERR_OP_NOT_PERMITTED may appear if stroked characters are cached for the font in question, size dependent data exists, and the requested strokewidth did not equal the one used at the time where the cache data initially has been built. Allowing to setup an arbitrary strokewidth under the latter circumstances might fillup the cache for one font with characters rasterized using inconsistent strokewidths, which has to be avoided. From this discussion it becomes suggesting not to cache stroked character at all, if stroking should be done using variable strokewidths.

The pen width for a given font FontID may be queried at any time using

$\mathcal{F}() \Rightarrow$ `float T1_GetStrokeWidth( int FontID)`

If the value 0.0 is returned, this either indicates that the characters are filled or that the font in question is not loaded. Then, `T1_errno` is also set to `T1ERR_INVALID_FONTID`.

5.13 Antialiasing

5.13.1 General Description

When fonts are displayed on screen at low sizes, the shapes of characters often get damaged because of rounding errors—a pixel can generally present two states: painted or non-painted. But the human eye can be fooled in a way that it "thinks" sub-pixel accuracy is given on the screen. This is done by considering which pixels are filled with ink to what degree and giving the physical pixel an appropriate shade of gray. For example, a pixel whose area would be covered 50% would get a 50% gray shade. This technique is called *antialiasing*.

There are several ways to implement antialiasing. `t1lib` implements antialiasing by internally generating a bitmap larger than needed and then subsampling. Depending on the subsampling factor which may be 2 or 4, this principle yields glyphs with 5 or 17 shades of gray including black and white.

There are three functions for generating antialiased glyphs:

$\mathcal{F}() \Rightarrow$
```
GLYPH *T1_AASetChar( int FontID, char charcode,
                     float size, T1_TMATRIX *transform)
```

$\mathcal{F}() \Rightarrow$
```
GLYPH *T1_AASetString( int FontID, char *string, int len,
                       long spaceoff, int modflag,
                       float size, T1_TMATRIX *transform)
```

$\mathcal{F}() \Rightarrow$
```
GLYPH* T1_AASetRect( int FontID, float size,
                     float width, float height,
                     T1_TMATRIX *transform)
```

Note the "`AA`" in the functions names which stand for <u>A</u>nti<u>A</u>liasing. The usage is identical to the usage of the functions `T1_SetChar()`, `T1_SetString()` and `T1_SetRect()`. So see 5.5 for an explanation of the arguments and their interpretation.

When an antialiased glyph is requested, the supplied `size`-value is multiplied by the current subsampling factor. For now, let us assume it is 2. Then the respective function for generating non-antialiased glyphs is called with all other arguments unchanged. The result is a bitmap twice as high and twice as wide as the user requested. Now, a 2×2 mask is moved over this bitmap and the number of painted pixels in this mask is considered at each place. According to the number of painted pixels one of 5 different gray shades is assigned to the resulting pixel. Since the mask is moved with an increment of 2 pixels in horizontal and vertical direction, the bitmap is at the same time subsampled by 2. Thus, the resulting bitmap is just of the size the user requested and its pixels each contain one of 5 gray shades.

Conceptually, the same applies for subsampling with 4. In this case the mask is of size 4×4 and there will be 17 distinct gray shades including black and white. The computational effort is considerably larger so that $4\times$ subsampling should only be used for very small sizes.

When moving the mask over double-sized bitmap it is aligned properly with respect to the characters' baseline (zero height) rather than with the characters' top or bottom line. This principle ensures, that the most important visual guideline in running text, the baseline, is represented in a consistent manor. This is especially important if one is using a serif-font.

Thanks to Raph Levien, the algorithm described above in a verbose manor has been replaced by a much faster lookup-algorithm in `t1lib` V. 0.4-beta.

5.13.2 Setting Operating Parameters

Applications can use both 2× and 4× antialiasing arbitrary mixed. Switching between the two modes is achieved using

$\mathcal{F}() \Rightarrow$ `int T1_AASetLevel( int level)`

The argument `level` should be either `T1_AA_LOW` (= 2) or `T1_AA_HIGH` (= 4). Any other values are ignored and `T1_errno` is set appropriately. This function is to be called after initialization. The default value after initialization is `T1_AA_LOW`. The current value can also be queried by issuing a call to

$\mathcal{F}() \Rightarrow$ `int T1_AAGetLevel( void)`

The returned value is current level. Switching between the two antialiasing modes should be quite fast since apart from a little error checking essentially only one simple variable is set.

There is one more value that may be specified for `level`, namely `T1_AA_NONE`. `T1_AA_NONE` is identical to 1 which means that no subsampling at all is done. But the resulting glyph, having only fore- and background colors is returned as a bytemap instead of as a bitmap. This is intended for situtations where an antialiased glyph should be concatenated with a (possibly large) non-antialiased glyph using the function `T1_ConcatGlyphs()`. In that case, the depths of the two glyphs have to be identical. There is probably not much more sense in setting `level` to `T1_AA_NONE`.

As described before, the result of the `T1_AASet..()` functions is strictly spoken no longer a bitmap since more than one bit is used to represent one pixel. The function

$\mathcal{F}() \Rightarrow$ `int T1_AASetBitsPerPixel( int bpp)`

allows the user to specify how many bits should be used to represent one pixel. Allowed values for `bpp` are 8, 16, 24 and 32. However, if 24 is specified, internally 32 will be used since the pixel are then addressed as objects of type `long`. Antialiased glyphs may grow quite large, especially when using `bpp` = 32. The value of `bpp` is written into the member `bpp` of the `glyph`-structure (see 5.5 on page 32). That way a user can check whether a glyph is antialiased or not. It is possible to work with antialiased and non-antialiased glyphs at the same time. It is also possible to directly query the value of bits per pixel by using

$\mathcal{F}() \Rightarrow$ `int T1_AAGetBitsPerPixel( void)`

The value returned is the number of bits per pixel used.

In order to make the handling of antialiased glyphs as flexible as possible, the values to be written into the pixels for different gray values may (and must) be explicitly specified. For low level antialiasing this is done by calling the function

$\mathcal{F}() \Rightarrow$

```
int T1_AASetGrayValues( unsigned long white,
                        unsigned long gray75,
                        unsigned long gray50,
                        unsigned long gray25,
                        unsigned long black)
```

For lower bpp values only the lower bits are used. For high level antialiasing this kind of graylevel specification is not economical since 17 arguments would have to be specified. Instead, another function is used which expects a pointer an array of unsigned long's:

$\mathcal{F}() \Rightarrow$

```
int T1_AAHSetGrayValues( unsigned long *grayvals)
```

The array grayvals points to must contain 17 entries. Element 0 is expected to specify the background color's pixel value and element 16 represents the foreground color. Calling one of these two functions involves also a new setup of the lookup tables. It should thus only be done if some color value really has changed.

In case the antialiasing level is set to T1_AA_NONE as described above, the function

$\mathcal{F}() \Rightarrow$

```
int T1_AANSetGrayValues( unsigned long bg, unsigned long fg)
```

must be used to set foreground and background color. In conclusion, it turns out that each antialiasing level has its own lookup tables which have to be initialized as soon as either foreground color, background color or both have changed.

Each of the three graylevel sets described above can also be queried by the user. This is done using one of the functions

$\mathcal{F}() \Rightarrow$

```
int T1_AAGetGrayValues( long *pgrayvals)
```

$\mathcal{F}() \Rightarrow$

```
int T1_AAHGetGrayValues( long *pgrayvals)
```

$\mathcal{F}() \Rightarrow$

```
int T1_AANGetGrayValues( long *pgrayvals)
```

Here, pgrayvals is the start address of an array of long-values to which the respective grayvalues are written. This memory must thus be supplied by the user. These functions will write 5 (T1_AAGetGrayValues), 17 (T1_AAHGetGrayValues) and 2 (T1_AANGetGrayValues) respectively to the location given by pgrayvals. These functions are to be called after initialization. If something goes wrong -1 is returned and T1_errno will be set accordingly. Otherwise 0 is returned.

5.13.3 Smart Antialiasing

Antialiasing improves legibility for small sizes but is not that much useful for large sizes. To make a compromise between computation time, system resources and optical appearance it might be advantageous to use T1_AA_HIGH for small sizes, T1_AA_LOW for medium sizes and T1_AA_NONE for large sizes. Of course, for large sizes the non-antialiasing functions could be used which still need less resources.

In order to free the user from having to switch the antialiasing level explicitly, t1lib can be told to do this switching automatically, depending on the size requested. This is called *Smart*

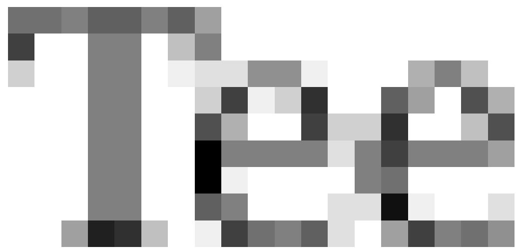

Figure 2: The string "Tee" (which is the German word for "tea") rastered at 13 bp, using 4× antialiasing. Notice the different representations of the character "e".

Antialiasing. It is disabled by default and can be toggled by a call to

$\mathcal{F}() \Rightarrow$
```
int T1_AASetSmartMode( int smart)
```

The quantity `smart` should be either be `T1_YES` (which corresponds to 1) or `T1_NO` (which corresponds to 0. Notice that the current antialiasing level as set by `T1_AASetLevel()` is not affected by this. After having switched off smart antialiasing the former antialiasing level is restored. When smart antialiasing is active still has to take care for setting the lookup tables after a color change has happened.

The numerical limits of the requested size at which `t1lib` will switch between the different antialiasing levels may be specified using

$\mathcal{F}() \Rightarrow$
```
int T1_AASetSmartLimits( float limit1, float limit2)
```

Here, `limit1` is the value of size at which `t1lib` switches from `T1_AA_HIGH` to `T1_AA_LOW` and `limit2` is the value of size at which `t1lib` switches from `T1_AA_LOW` to `T1_AA_NONE`. The default values are 20.0 for `limit1` and 60.0 for `limit2`. This means for sizes smaller than 20.0 `T1_AA_HIGH` will be used and for sizes equal to or greater than 60.0 `T1_AA_NONE` will be used. The intermediate range is covered by `T1_AA_LOW`. These values are suitable for applications that display on screen when the device resolution has been left at the default value of 72 dpi.

5.13.4 Caching of Antialiased Character Glyphs

Generally, antialiased glyphs are not cached in `t1lib` because this involves several problems which are hardly to solve. One main problem is shown in figure 2. Obviously the character "e" appears twice in different representations. This is intentional and is referred to as sub pixel positioning. In the left "e" the letter is perceived somewhat more to the left with respect to the pixels that represent the character. Conversely, the second "e" seems to lie somewhat more to the right within the pixels. The advantage of this technique is that characters can be shifted by some fractional amount of a pixel at low sizes.[11] On the other hand the problem is introduced that each character can have more than one representation in graylevels, depending on how much subpixel shift is needed.

One further problem caused by subsampling is that certain information is irreversibly lost in the graylevel representation. For example, if you have a graylevel pixel of intensity 50% (whatever the real color is), then, in case of $2\times$ antialiasing, you will know that in the 2×2 input bitmap two pixels had been set to foreground, but you would not know *which* two these had been. But this information is important for concatenating and blitting of antialiased bitmaps: it may well happen that two pixels with 50% gray that lie over each other had to produce an output pixel of 50%, 75% or 100% gray (where 100% gray means full foreground intensity).

To avoid these problems, `t1lib` generally does not cache antialiased glyphs. Instead, it works on true bitmaps which are then subsampled at the last possible stage to an antialiased glyph. Applications that do not use anything more than the functions that yield char bitmaps or bytemaps, could profit from caching of antialiased characters. Such applications could specify `T1_AA_CACHING` as an additional ingredient to the `log` argument of the function `T1_InitLib()` which initializes `t1lib`. This is done by OR'ing the value of `log` with `T1_AA_CACHING` as described in 5.3. If this flag had been specified at initialization time, `T1_AASetChar()` will cache the bytemaps it has generated and will take them from cache in future requests.

When caching antialiased glyphs, each size gets up to four distinct cache areas, one for bitmaps and one for $1\times$, $2\times$ and $4\times$ subsampled bytemaps each. As soon as a string-generating function is called these cached antialiased glyphs cannot be used for the reasons discussed before. The developer of an application should thus carefully think about whether a possibly marginal performance gain is really worth this much higher effort. If in doubt, simply check it out. Applications like `xdvi` which place isolated character glyphs on a sheet could use this feature, however, and profit from it.

5.14 Interface to Outlines

Although `t1lib` is meant for generating bitmaps from Type 1 outline fonts, there is a set of functions for accessing outline data. There are several reasons for this. Firstly, outline descriptions are, within the given arithmetic constraints, mathematically exact. Secondly and related to the previous point, in certain cases where exact subpixel positioning is needed, the functionality of grid-fitting before rasterization is needed. This can only be done accurately based on outlines. To illustrate this, consider figure 3. When looking at the concatenated glyph a), it appears that the underline rule has a small step where the two words touch.[12] The reason is, that the second part of the glyph had been rastered with respect to a pixel coordinate of exactly $(0, 0)$. Since the start of the second word in the resulting glyph does not exactly fall on an integer pixel location, bitmap blitting causes an artifact in the visual line of the underlining rule. Strings rotated at angles that are not multiples of $90°$ are prone to produce such effects. In contrast the concatenated glyph b) does not show such effects because both partial glyphs are placed mathematically exact and then filled. Thirdly, if the outline of a character is available, it can be used for whatever. For example, the outline can be filled by another rasterizer, it can by altered, it can be stroked and so on. `t1lib` makes outlines as they are internally used by the rasterizer available. We will discuss how to interprete and access outlines in the remainder of this section.

[11] The opinions whether this and antialiasing in general is of advantage for readability vary, so please consider this the opinion of the author.

[12] Depending on the resolution and quality of the hardcopy you are reading, the effect might be hardly or not all noticeable.

a) b)

Figure 3: Two concatenated bitmaps, a) concatenation done based on bitmaps by blitting and b) based on outlines and then filled.

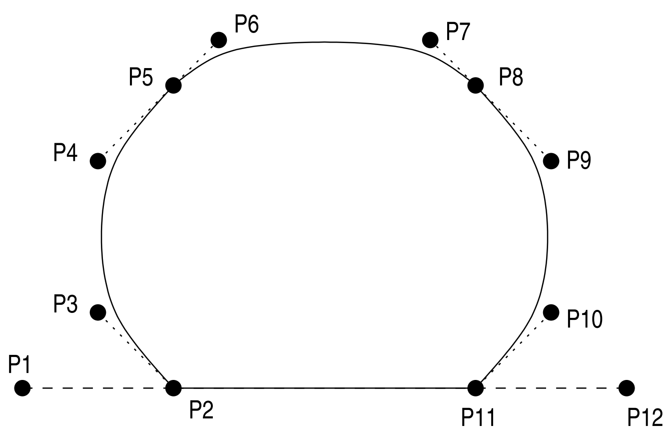

Figure 4: The outline of a fictive character.

5.14.1 Outline Format

Before going into implementation details the general structure of a Type 1 outline is described. We will consider the simple fictive character whose outline is shown in figure 4. We assume that scaling, grid fitting and hinting has already been carried out. Then, the outline is given by set of points P_i and segments connecting those points. There are:

- Move-segments (type = `T1_PATHTYPE_MOVE`): These are straight segments which cause the current position to be displaced by some offset. Since the starting point of a segment is always implicitly the current point, only one argument is needed, P_{dest}, the destination point. In the figure, P_1–P_2 and P_2–P_{12} are Move-segments. In this simple case they displace from the characters origin to some starting point of the outline and from the ending point of the outline to the point where the next character would have to be placed (the horizontal escapement).

- Line-segments (type = `T1_PATHTYPE_LINE`): These are part of the path to be filled later. In analogy to the Move-segment, one argument, P_{dest}, is required for Line-segments. In the figure, P_{11}–P_2 is a Line-segment.

- Bezier-segments (type = `T1_PATHTYPE_BEZIER`): These are curve segments. Their shape is defined by a starting point (always the current point here), an ending point P_{dest} and two control points P_B, P_C. These four points are the parameters of what is called a third order Bezier spline.[13] The resulting curve has the following properties:
  - It starts at the first point $P_{current}$.
  - It ends at the fourth point P_{dest}.
  - The line that goes through the points $P_{current}$ and P_B is the tangent to the curve from the right side at the starting point $P_{current}$.
  - In analogy, the line that goes through the points P_C and P_{dest} is the tangent to the curve from the left side at the ending point P_{dest}.
  - The resulting curve will be enclosed completely by the convex area that is defined by connecting the definition points with straight line segments.

  Our fictive character outline in figure 4 has three Bezier-segments, P_2–P_3–P_4–P_5, P_5–P_6–P_7–P_8 and P_8–P_9–P_{10}–P_{11}. Notice that it is easily possible to achieve a smooth tangent transition from one curve-segment to the next by choosing the involved points from a straight line.

For Type 1 fonts in general, the following rules for interpreting coordinate specifications hold:

- All point specifications are relative to the *current point*.
- For Bezier-segments, P_B, P_C and P_{dest} all are relative to $P_{current}$.
- Initially, i.e. when a character outline is started, the current point is at the origin $(0, 0)$ of the character.

Additionally, for this special rasterizer implementation, the following terms apply:

- The vertical coordinate is—in contrast to PostScript—inverted, i.e., the y-axis points down.
- Once hinted and gridfitted, the outline point coordinates are described in *fractional pixels*. A “fractpel” is of type `long` and describes the location in 2^{16}th fractions of a pixel. To convert from pixel to fractional pixel and vice versa, the macros `T1_TOPATHPOINT(p)` and `T1_NEARESTPOINT(fp)` are provided.

Before describing the functions for retrieving outlines the format in which outlines are presented in C will be described. A point specification is done in the following structure:

```
typedef struct {
  T1_int32 x;
  T1_int32 y;
} T1_PATHPOINT;
```

`x` and `y` are fractional pixels as described above.

An outline is represented by a linked list of structures which describe path segments of the type described above. Line- and Move-segments are described by the following structure:

[13] The mathematical defining equation represents a special case of a Bernstein polynom which was exploited by Bezier in the context of solid modeling. The curve especially has the property that it may be approximated efficiently by straight line segments in a few iterations.

```
typedef struct pathsegment {
  char type;
  unsigned char flag;
  short references;
  unsigned char size;
  unsigned char context;
  struct pathsegment *link;
  struct pathsegment *last;
  T1_PATHPOINT    dest;
} T1_PATHSEGMENT;
```

`type` is either `T1_PATHTYPE_MOVE` or `T1_PATHTYPE_LINE`. `flag`, `references`, `size` and `context` are internally used by the rasterizer. `link` is a pointer to the next segment structure or `NULL` in case it is the last structure in the list. Finally, the `last`-entry is a pointer to the last structure in the linked list. `last` is only set in the first segment and is reset to `NULL` in the remaining segment structures. A Bezier-segment is described by the following structure:

```
typedef struct bezierpathsegment {
  char type;
  unsigned char flag;
  short references;
  unsigned char size;
  unsigned char context;
  T1_PATHSEGMENT *link;
  T1_PATHSEGMENT *last;
  T1_PATHPOINT    dest;
  T1_PATHPOINT    B;
  T1_PATHPOINT    C;
} T1_BEZIERSEGMENT;
```

Obviously, the format is identical to that for straight path segments, extended by the entries `B` and `C` which specify the control points as described earlier in this subsection. The common return type for the outline retrieving functions is a pointer to `T1_OUTLINE`, which is in fact identical to `T1_PATHSEGMENT`. This purely for convention. Although it is quite unlikely, an outline might start with a Bezier-segment. To access Bezier-segment elements, a cast must be used.

5.14.2 Using Outlines

`t1lib` provides three functions for retrieving outlines. The first is

$\boxed{\mathcal{F}() \Rightarrow}$

```
T1_OUTLINE *T1_GetCharOutline( int FontID, char charcode,
                               float size, T1_TMATRIX *transform)
```

The meaning of the arguments is as in the `T1_SetChar()`-function. Notice that the size specification is also required here. Outlines are, by their nature in Type 1, generally defined in a 1000×1000 grid and then scaled down by the fontmatrix to 1 bp. The space is known as the charspace. The reason for specifying a size at this place, instead of scaling the outline later, is, that hinting is performed according to the scaled outline. The returned outline is then hinted for the given size. If necessary, it may still be scaled later.

The outline for a complete string can be retrieved by

$\mathcal{F}() \Rightarrow$

```
T1_OUTLINE *T1_GetStringOutline( int FontID, char *string, int len,
                                 long spaceoff, int modflag,
                                 float size, T1_TMATRIX *transform)
```

The meaning of the arguments is as in `T1_SetString()`.

Finally the "outline" for a displacement is available by the function

$\mathcal{F}() \Rightarrow$

```
T1_OUTLINE *T1_GetMoveOutline( int FontID, int deltax, int deltay, int modflag,
                               float size, T1_TMATRIX *transform)
```

This function is intended to be used for concatenation of outlines. It needs all the arguments because some quantities which are given on the font level are required for constructing the outline. `deltax` and `deltay` are the horizontal and vertical displacement measured in charspace units. From the `modflag`-argument, especially the underlining parameters are evaluated. Although x- and y-displacement may be specified, the resulting outline is still subject to scaling with `size` and transformation according to `transform`.

Arbitrary outlines may be concatenated by using the function

$\mathcal{F}() \Rightarrow$

```
T1_OUTLINE *T1_ConcatOutlines( T1_OUTLINE *path1,
                               T1_OUTLINE *path2)
```

Notice that this concatenation is done with high precision so that we can expect that visual artefacts are reduced to a minimum (remember figure 3).

Scaling of outlines is done by the function

$\mathcal{F}() \Rightarrow$

```
T1_OUTLINE *T1_ScaleOutline( T1_OUTLINE *path, float scale)
```

`T1_ScaleOutline` does nothing more than linearly scaling the coordinate data with `scale` and storing the result in fractional pixels. No care is taken for hinting (see above).

An outline may be duplicated using the function

$\mathcal{F}() \Rightarrow$

```
T1_OUTLINE *T1_CopyOutline( T1_OUTLINE *path)
```

This is a direct entrypoint into the rasterizer. It works by allocating and duplicating each segment of `path`. This function may be useful if one wants to do several things with one outline because the process of filling an outline also consumes that outline.

An outline that that a user decides not to fill can be destroyed by the function

$\mathcal{F}() \Rightarrow$

```
void T1_FreeOutline( T1_OUTLINE *path)
```

It iterates through the segment list and frees each segment. This must not be done after filling an outline because the filling process consumes the outline!

Finally, there are two functions that produce glyphs from outlines, namely

$\mathcal{F}() \Rightarrow$

```
GLYPH *T1_FillOutline( T1_OUTLINE *path, int modflag)
```

and

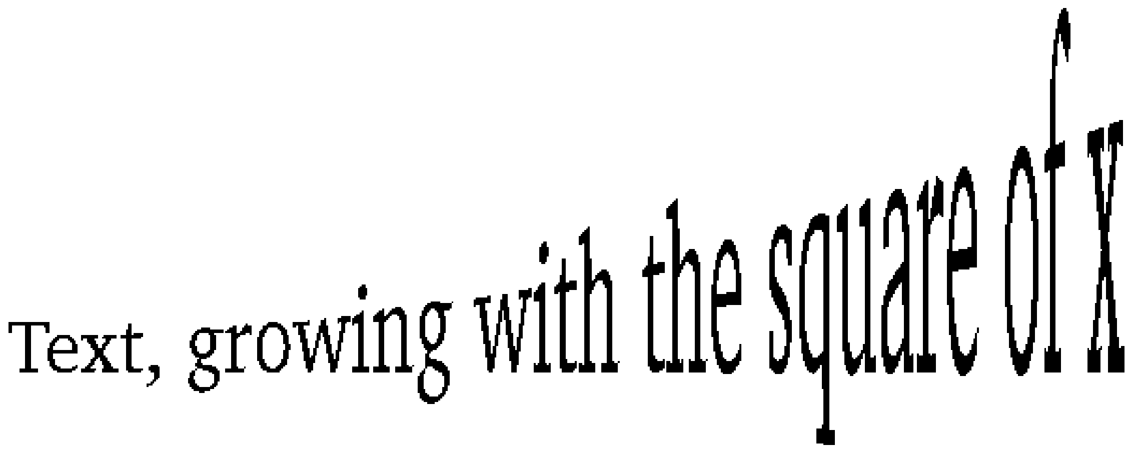

Figure 5: A string with nonlinearly scaled coordinates.

$\mathcal{F}() \Rightarrow$

```
GLYPH *T1_AAFillOutline( T1_OUTLINE *path, int modflag)
```

Their usage does not need any explanation. The value of `modflag` is required for *Right-To-Left* typesetting. If the bit `T1_RIGHT_TO_LEFT` is set, the dimension of the glyph are recomputed accordingly. All other bits from `modflag` are ignored such that in the usual case of *Left-To-Right* typesetting simply 0 can be specified. While `T1_FillOutline()` produces bitmaps of depth 1, `T1_AAFillOutline()` produces antialiased bytemaps of the current graphics depth. It should be mentioned that Smart Antialiasing (see 5.13.3) does not work with this function. The reason is that `t1lib` has no notion of the quantity "size" when it gets the outline to process. Hence, Smart Antialiasing can't work in this case. As noted above, the outline is consumed by the filling functions so that there is no need to free it explicitly.

5.14.3 Manipulation of Outlines

`t1lib` provides some limited further functionality to process outlines. First of all, a user would expect a character to be defined in a coordinate system in which x points to the right and y points up. Further, a representation of the glyph where all points are specified in absolute coordinates would be advantageous for manipulating outline-points. This is because most transformations, linear or nonlinear, need to have an absolute x-value to compute an y-value or vice versa. The function

$\mathcal{F}() \Rightarrow$

```
void T1_AbsolutePath( T1_OUTLINE *rpath)
```

does exactly what has been described just before, (a) conversion of relative coordinates into absolute coordinates and (b) inverting the y-direction.

Once a path has been converted into an absolute path, it is suitable for possibly nonlinear manipulation.[14] As an example of what can be done, have a look at figure 5. The string displayed has been generated by essentially applying the transformation $y' = y(1 + cx^2)$, with appropriate c. To allow such transformations by the user, `t1lib` provides the function

[14] A linear manipulation of path points would rather be realized using the transformation matrices as described in 5.11.

```
 void T1_ManipulatePath( T1_OUTLINE *path,
                         void (*manipulate)(long *x,long *y, int type))
```
$\mathcal{F}() \Rightarrow$

Here, `path` should be an absolute path as described above. Notice that `t1lib` has no way to check whether the path is relative or absolute, this is in the responsibility of the user. The second argument is a pointer to a function that has a return type of `void` and that expects three arguments: two pointers to `long`-values one integer `type`. `T1_ManipulatePath()` works by iterating through all outline points of `path` and calling the function `*manipulate()` for each outline point. When the function `*manipulate()` is called, `x` and `y` are pointers to the x- and y-coordinates respectively of the outline point to be processed. That way, `*manipulate()` can alter the outline points arbitrarily. The `type`-argument will be set to the segment type by `T1_ManipulatePath()`. As described earlier, the segment type can be one of `T1_PATHTYPE_MOVE`, `T1_PATHTYPE_LINE` and `T1_PATHTYPE_BEZIER`. Of course, the function `manipulate()` has to be written by the user. To make it clear, we consider a function which stretches an outline horizontally by 1.5. The code fragment for this could be:

```
      .
      .
      .
void h_stretch( long *x, long *y, int type)
{
  double dx;

  dx=(double)*x;
  dx *=1.5;       /* scale x coordinate by 1.5 */
  *x=(long)dx;
}
      .
      .
      .
T1_OUTLINE *path=NULL;
path=T1_GetStringOutline(FontID,(char *)SomeString,
                         0,0,T1_KERNING,20.0,NULL);
T1_AbsolutePath( path);
T1_ManipulatePath( path, &h_stretch);
T1_RelativePath( path);
glyph=T1_FillOutline( path, Modflag);
      .
      .
      .
```

As the example above already has shown, an absolute path, manipulated or not, must converted back to a relative path before it finally can be interpreted by the rasterizer. This conversion is done using

$\mathcal{F}() \Rightarrow$
```
void T1_RelativePath( T1_OUTLINE *apath)
```

As already mentioned with respect to `T1_AbsolutePath()`, `t1lib` cannot check whether the `path` specified is really absolute. The user has to take care for this.

A few general comments about manipulating paths are appropriate. Although the mechanism

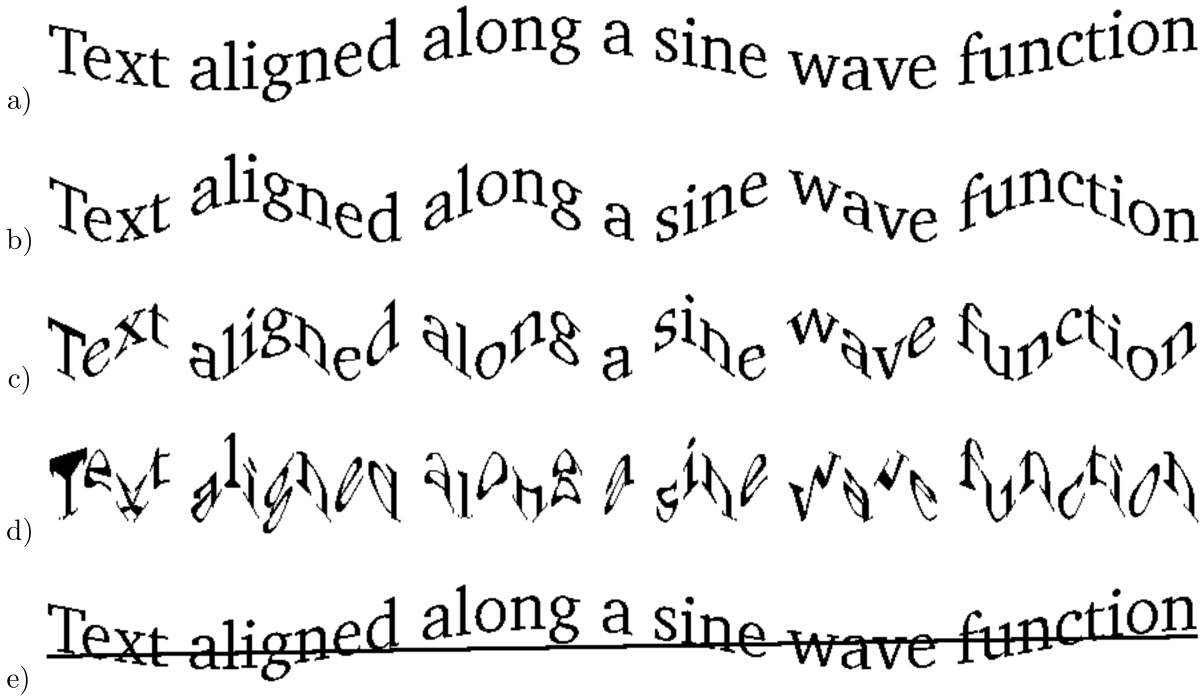

Figure 6: The string “Text aligned along a sine wave function” using a period of (a) 500, (b) 200, (c) 100, (d) 50 pixels and (e) again 500 pixels with underlining. The sine amplitude was 30 pixels (in screen resolution).

implemented by `T1_ManipulatePath()` allows arbitrary manipulation of path points, one must be very careful in doing so. Figure 6 exhibits some of the problems that may arise. A text string aligned to a sine function is displayed. In part (a), a pleasing smooth text flow is shown and this also applies for (b) where the period of the sine has been reduced to 200 pixels. In (c), where the period has been reduced to 100 pixels, some artefacts already show up. For example, the top bar of the uppercase “T” has noticeable variance in thickness. In (d), where the period has been reduced again, the result is hardly readable. Another artefact appears in figure 6 (e): since the underlining rule is defined by four points only, these points and nothing else is transformed with the result that the out coming line is still straight and not curved as we would like. From this discussion we conclude, that such transformations can only be done with reasonable results if the maximum distance between the outline points of a shape is small compared to the variance of the outline points that the transformation results in. This rule, although being very fuzzy and non-mathematical, should give a good estimation of which transformations are valid.

Another completely independent topic is that, at the level where `t1lib` provides outlines, their representation is strictly descriptive with respect to points and their connections. There are no such things like `closepath`-segments which would take care that a path is really closed, no matter what the transformation had been. This means, that identical points $P_1 = P_2$ have to be transformed to identical points $P_1' = P_2'$, no matter where they appear in the outline. However, if the transformation is done by by a function $(x', y') = f(x, y)$ as suggested, this should never be a problem.

Finally, one should remember that all computations in the user function `manipulate()` have to be done in units of fractional pixels, rather than in pixels. When designing a sine wave as in figure 6, this must be taken into account with respect to periodicity.

5.15 Logical Fonts

It sometimes may be necessary to have a font and an extended or slanted variant simultaneously. To enable such configurations without needing to declare the fonts two or even more times in the font database file, `t1lib` provides the function

$\boxed{\mathcal{F}() \Rightarrow}$ `int T1_CopyFont( int FontID)`

It copies the top level data structure of the font given by `FontID` to another location. The newly created font refers in fact to the same physical memory as the font `FontID` as far as Type 1 and AFM data are concerned. However, no size specific data is copied from font `FontID`, you can thus do with the new font whatever you want to. It will get its own size-specific memory area when the first bitmap is generated using its ID.

It is also possible to reencode a copied font without affecting the original font. This is possible because a logical font gets its own mapping tables. This allows configurations with one font at different encodings simultaneously.

In order to keep track that another font is referring to data from font `FontID`, a reference counter is managed for every font. The reference counter for font `FontID` is incremented after a call to `T1_CopyFont()`.

If the font `FontID` is not loaded into memory, the function returns -1.

Only *physical* fonts—those fonts defined in the font database file or added via `T1_AddFont()`—may be copied to another font. If a user tries to copy a font which is already logical, the function returns -2.

If no memory is available for the new font the function return -3. But this should not happen.

If all goes the right way, `T1_CopyFont()` returns an integer—lets call it `new_ID`—which is from now on a valid font identification number.

5.16 Missing or Invalid AFM Files

`t1lib` heavily relies on AFM information every time the relative position of bitmaps is of importance. Because AFM information is of high resolution, accumulating positioning errors are avoided in contrast to what the X11 text drawing functions do. On the other hand, there are many freely available Type 1 font programs that come without AFM files. This problem has been addressed in `t1lib` 0.5. `t1lib` is now able to generate AFM information on the fly and it even can generate AFM files from Type 1 font files.

5.16.1 Remarks on AFM Files

Information in AFM files is only relevant for placing character glyphs but not for rasterizing. The metric values are based on the same coordinate system as used in Type 1 font files, the so called *charspace coordinate system*. One unit is 1/1000bp when a font is not scaled or scaled to 1 bp, respectively.

Information in AFM files can divided into several groups:

1. *Global Font Information:* This information is generally not needed to place characters. Furthermore, most of this information is also contained in a Type 1 font file itself. This area is thus of marginal importance for `t1lib`.

2. *Character Width's and Bounding Boxes:* These both are crucial for accurately placing the character glyphs. Fortunately, these are dimensions are exactly defined by the character

outlines themselves. It is thus possible to compute them spending some computational effort.

3. *Ligature Information:* For æsthetic reasons, certain character groups are often replaced by ligatures and a font file may define several ligatures. It is however not intuitively clear what character groups should be replaced by what ligatures.[15] Fortunately, ligatures are not crucially needed for quality typesetting.

4. *Pair Kerning Information:* This information is quite important for æsthetic reasons but it is entirely independent from the outline descriptions and can thus not be extracted from a font file.

5. *Track Kerning Information:* This information gives hints of how to typeset text generally closer or wider at varying point sizes. `t1lib` does not use track kerning information and I personally do not consider using track kerning a good typographical style.

6. *Composite Character Data:* This is needed to construct characters from two single characters. Typical examples are accented characters. `t1lib` currently does not deal with composite characters. Most of the composite characters needed are already existent internally.

To come to a conclusion, for our purposes it is sufficient to generate the characters' widths and their bounding boxes and we have all information we need to construct string glyphs.

5.16.2 Generation of AFM Information

Next lets consider how to generate the AFM information. It is a series of entirely independent steps:

- When we generate AFM information, we want to do this once and forever when the font is loaded. Consequently all characters, have to be examined, not only those that are currently encoded. We start by fetching all character names the font defines. This done with `T1_GetAllCharNames()` (see 5.10.3). This yields a list of possibly more than 256 character names.

- Each of the character addressed by the names above is now rastered at size 1000 bp. By rastering at 1000 bp we match exactly the charspace coordinate system which the character outline descriptions are based on. Width and bounding box are easily examined and saved at appropriate places.

- The kerning pair area and ligatures are explicitly set to zero.

At the end of this procedure, there is a data area identical to what would have been built when reading an AFM file without kerning-section and ligature specifications.

The decision of building AFM data is done on the fly without any user interaction. Here is what happens on the metrics-area when loading a font:

- `t1lib` tries to open an AFM file reading metrics and kerning pair information.

- If this does not succeed, it tries to rescan the AFM file in a *sloppy* way, only requesting metrics information.

- If this fails too, metrics information is generated on the fly as described above.

[15] Well, at least not without some expert knowledge like "I know this ligature's name is 'fi', so I replace every series of 'f' and 'i' with it."

It should be noted that generating metric information the way described above takes significant amount of time since every character has to be rastered at 1000 bp.

If the `T1_NO_AFM` flag is passed to `T1_InitLib()`, `t1lib` will neither attempt to open AFM files nor generate AFM information. This is useful to speed up applications which do not need the metrics data. However, this slows down access to certain features, mostly related to the string processing functions, and completely disables the features that only are contained in AFM files (like kerning and ligatures).

Obivously, the `t1lib` functions that use the AFM data will not work correctly in this case and should not be used.

5.16.3 Writing AFM Files

In order to reduce the situations where AFM data has to be generated on the fly, `t1lib` provides the following function:

$\mathcal{F}() \Rightarrow$ `int T1_WriteAFMFallbackFile( int FontID)`

It writes an AFM file for the font identified by `FontID`. This is done executing the following steps:

1. The AFM filename is constructed by taking the fontfilename, cutting off the extension and appending `.afm`.

2. A pointer array of size $256 + n$, where $n =$ number of characters, is allocated and set to NULL. The leading 256 entries are reserved to point to encoded characters' metrics. The remaining entries are intended to point to metrics of unencoded characters. We see that this is a worst case speculation: The pointer array is large enough for the extremely unusual case that no characters are encoded.

3. Next the function steps through all character names and gets their encoding index i. If $i \geq 0$, the character is encoded and the ith pointer element in the array is set to point to the metrics of this character. If $i = -1$, the character is not encoded and the lowest unused pointer in the second area is set to point to the metrics of this character.

4. Next the AFM file is opened and the header information as well as a comment by `t1lib` are written. There are 5 entries that are not trivially to extract from the font file: `Ascender`, `Descender`, `XHeight`, `CapHeight` and `EncodingScheme`. Their discussion is deferred to later in this section.

5. After the header, the metrics information is written in the format required for AFM files. This is done by stepping through the pointer array until the first NULL pointer in the unencoded characters' area is reached.

The result is a list of char-dimensions entries which is leaded by the encoded characters in ascending order of their encoding index, followed by a list of unencoded characters in alphabetical order.

As seen above, the current encoding takes influence on the order the characters appear in the AFM file. One should thus not produce AFM files from reencoded fonts, although this is possible. This yields non-standard AFM files and gives no performance gain, even not when used with `t1lib`.

The entry `EncodingScheme` is not always contained in the fontfile itself. It is generated by comparison between encodings. `t1lib` has only one builtin encoding, `AdobeStandardEncoding`, which is recognized. Every further encoding, defined by the font itself or applied by a user, is always marked as `FontSpecific`.

The entries `CapHeight`, `XHeight Ascender` and `Descender` are not fully determined by a Type 1 font file although they are existent with high probability. As rough definitions can be considered:

- `CapHeight`: The height a capital 'H' reaches to.
- `XHeight`: The height a lower case 'x' reaches to.
- `Ascender`: The height a lower case 'd' reaches to.
- `Descender`: The depth a lower case 'p' reaches down.

It is obvious that these definitions make only sense in certain font definitions. For example, a musical notation font might not necessarily define an ascender since no capital letters are provided.

In the Type 1 notion these dimensions are referred to as top alignment and bottom alignment values respectively. The corresponding alignment "zone", i.e., an interval, is defined by the alignment height and a corresponding overshoot position. The alignment zones are specified in the BlueValues array for top alignment zones and the OtherBlues array for bottom alignment zones. A Type 1 font may define up to 7 top alignment zones and 5 bottom alignment zones. It is unfortunately not defined which of these alignment zones refer to `CapHeight`, `XHeight`, `Ascender` and `Descender`.

`t1lib` tries to get out of this dilemma by making a best guess:

1. For each of the characters 'H', 'x' and 'd' it fetches the largest y-value and compares the result with each alignment zone in the BlueValues array. The alignment zone closest to the observed character dimension is assumed a candidate for the respective quantity.
2. It checks whether the difference between the alignment zone just selected and the character dimension is within a certain tolerance area. This tolerance width is ± 30 charspace units. If the result is positive, the quantity in question is assigned the numerical value of the standard height (not the overshoot) of this alignment zone. Since we are currently considering top alignment zone, this will always be the lower value.
3. If the value is out of tolerance or the font even does not define the character, the corresponding entry in the AFM file is left out.
4. A comparable procedure is then done for `Descender`, this time examining the OtherBlues array.

Note that if the values do not seem to be correct, the corresponding lines can be removed from the AFM file without doing any harm. These entries are optional only.

`T1_WriteAFMFallBackFile()` can indicate a number of error conditions by returning appropriate values. These are:

- `0`: No error occurred, AFM file was successfully written.
- `-1`: The AFM data for the font in question has been generated by reading an AFM file, there is no need to generate a new one. If you really want to force an AFM file to be written, take care that `t1lib` does not find an AFM file when loading the font.
- `-2`: The font in question is not loaded.

- `-3`: The font in question is loaded but AFM data has not been generated. This definitely is an error condition because it indicates not all characters of the font could be rastered, either because the font file is damaged or because there were insufficient system resources. In any case the application should generate a logfile and this file should be examined.
- `-4`: The AFM file could not be opened. This could be a permission problem or something else. The file is always opened in the current working directory.
- `-5`: The file has successfully been opened but there was an error writing to the file.
- `-6`: A memory allocation error occurred. This should not happen because it indicates there are no system resources.

5.17 Font Subsetting

When applications have to setup Postscript files for printing, the problem is that these files often grow large. Each font which is not known to the Postscript interpeter, i.e., usually each font that is not part of the set of 35 standard fonts, has to be downloaded as part of the file. The size of a particular font file often can be reduced by font subsetting, because usually only a small subset of the available character descriptions is actually needed.

5.17.1 Font File Organization and Subsetting

Each Type 1 font file is a special Postscript program defining three Postscript-dictionaries:

- `FontInfo`
  Global font information like font and family name and encoding scheme is stored here. This data is required even for a subsetted font. However, as will be described later, the encoding scheme may be reduced to those characters that are in the desired subset.
- `Private`
  This dictionary is in the encrypted part of the file and stores global font data too. This data includes quantities parameterizing hinting and subroutines that might be called by the character descriptions. This data is required also.
- `Charstrings`
  For each character defined in the font a binary and encrypted byte string (charstring) coding the character outline is stored in this dictionary. The number of charstrings may be considerably larger than the size of the encoding vector. This dictionary usually constitutes the largest part of the font file and, consequently, it is the place to reduce storage requirements.

The main principle in subsetting is to decrypt the font and reorganize it, leaving out charstrings that are not required in the current context. For example, if a document uses only the character 'A' from the font Garamond, then this font may be subsetted preserving the character outline for 'A' only. The resulting file, which will be much smaller than the original file, can then be included verbatim into the Postscript file containing the document. At the same time, optionally, the encoding vector could be redefined to contain only the entry for 'A' at the appropriate location and `.notdef` otherwise.

A still more consequent subsetting would involve leaving out those subroutines from the `Private`-dictionary that are not needed by the preserved charstrings. Leaving out some subroutines on the other hand would require to interprete and check all charstrings for the subroutines they require and all preserved charstrings would have to be adapted to the reorganized index. Since the subroutines usually do not consume that much memory this is not considered worth the effort.

5.17.2 Functions for Subsetting

There are two ways to obtain a subsetted font from an existing file. The user can (1) do it step by step which requires reading, decrypting, reorganizing and encrypting of the font file, and (2) use a high level function to do it without having to know anything about the details. For font subsetting, t1lib provides the function

```
char *T1_SubsetFont( int FontID, char *mask,
                     unsigned int flags, int linewidth,
                     unsigned long maxblocksize,
                     unsigned long *bufsize)
```

$\boxed{\mathcal{F}() \Rightarrow}$

It returns a pointer to a memory block containing the subset data. The memory is allocated in the function and it is the responsability of the user to free this memory. The parameter `FontID` as usual is used to tell `t1lib` which file or font is to be used for the operation.

`mask` points to an array of characters which has to be setup by the user. This array must comprise exactly 256 characters and for the index of each non-zero entry the charstring resulting from the current encoding is preserved in the subsetted font. To give an instance, if the subset should consist in the character 'A' only and we assume the current font specifies `StandardEncoding`, then the `mask`-array should be initialized to zeroes and `mask['A']=1` or some other non-zero value.

The `flags` parameter allows to control the subsetting operation. It usually should be set to `T1_SUBSET_DEFAULT`. In this case, the subset is ASCII-hex encrypted, that is, as in a `.pfa`-file. It is thus well-suited for the verbatim insertion into a Postscript file. If the source font file in question defines the encoding to be `StandardEncoding`, the encoding is not adjusted to the subset specified by `mask`. By contrast, if the font defines a `FontSpecific` encoding, this encoding will be adjusted according to the subset. This default behavior—which mimics what e.g. `dvips` seems to do—may be overwritten by OR'ing `flags` with `T1_SUBSET_FORCE_REENCODE`, which leads to adjusting the encoding vector in any case. Conversely, reencoding can be suppressed unconditionally by OR'ing `flags` with `T1_SUBSET_SKIP_REENCODE`. If `flags` is OR'ed with `T1_SUBSET_ENCRYPT_BINARY`, a buffer of mixed ASCII, binary and EOF segment types is created and encryption is performed in binary mode. The buffer's contents in this case represents a valid compact binary format file (`.pfb`). It is considerably smaller than a comparable `.pfa`-file but it is not suitable to be inserted into Postscript files.

The parameter `linewidth` is used to specify the line length if ASCII-hex encryption is used. Since—according to the Adobe speification—the first 8 encrypted bytes have to be stored one after the other without interspersed white space, the allowed range of `linwidth` is limited to 8 at the lower bound. It is also limited at the upper bound by 1024, because writing that long lines does not preserve the readability of the produced file.

The parameter `maxblocksize` is important if binary encryption is used. Then, this value specifies the maximum allowed segment size. For similar reasons as discussed above, this value must be equal to or larger than 4. There is no limit at the upper bound, because the maximum segment size can be derived automatically follows from the target font file.

`bufsize` must be a valid pointer to an `unsigned long int` in the context of the calling function. The size of the memory area to which the function returns a pointer, is written to `bufsize`. The calling function needs this number to process to buffers contents, e.g., to write it to a file.

If errors occur in this function, `NULL` is returned and `T1_errno` is set to an appropriate value. If the font corresponding to `FontID` is not loaded, `T1_errno` is set to `T1ERR_INVALID_FONTID`. `T1ERR_INVALID_PARAMETER` is used to indicate that one of the further arguments is out of range. T1ERR_ALLOC_MEM and T1ERR_FILE_OPEN_ERR may also be set in this function. Finally, T1ERR_UNSPECIFIED may also be set if the charstring definition for .notdef could not be located. This is considered to be a fatal error.

An example of how to use the function described above is given in the file subset.c in the examples/ subdirectory of the distribution.

5.17.3 Further Functions for Subsetting

For the sake of completeness, there are a few further functions in the subsetting module. The function

$\mathcal{F}() \Rightarrow$ `char *T1_GetCharString( int FontID, char *charname, int *len)`

returns a pointer to the charstring of the character with name charname of the font identified by FontID. In case of an error, NULL is returned and T1_errno is set to T1ERR_ALLOC_ERR if there was not enough memory for storing the charstring, T1ERR_UNSPECIFIED if the charstring was not found in the dictionary, T1ERR_INVALID_FONTID if the font in question is not loaded or T1ERR_INVALID_PARAMETER if charname or len is NULL. The memory pointer which is returned is managed static in this function. Thus, it should not be free'd by the user, or, in case the memory block is free'd, the pointer must be set to NULL.

In order to decrypt a charstring, the lenIV-value of the font in question must be known. It can be obtained using the function

$\mathcal{F}() \Rightarrow$ `int T1_GetlenIV( int FontID)`

The returned value indicates how many leading random bytes are used for charstring encryption in the font FontID. According to an undocumented Adobe convention, the value -1 is also valid and indicates that the charstring is not encrypted at all. Hence the return value -2 is used to indicate an error. In this case, T1_errno is set to T1ERR_INVALID_FONTID, which indicates that the font in question is not loaded.

5.18 Composite Characters

This section discusses the composite character information that may be present in AFM files and how this information is represented, accessed and handled in t1lib.

5.18.1 General remarks

Composite characters are defined by the fact that they are constructed from at least two independent symbols. In practice there frequently appear two components, a base character and an accent, e.g., as in “Ä”. Usually, the accents (or secondary pieces) of a composite character are typeset first without causing any horizontal escapement and finally the base character itself is typeset and causes its escapement to become the escapement of the whole composite character. Although some people recommend that the character definition of an accent itself should not cause escapement, this generally is not respected in real fonts and t1lib does not require this condition to be fulfilled.

In order to construct a composite character the characters to be put together have to be known and metric information about how to put these characters together has to be known too. t1lib defines two structures as new data types for this purpose. The first is

```
typedef struct
 {
```

```
  int compchar;
  int numPieces;
  T1_COMP_PIECE *pieces;
} T1_COMP_CHAR_INFO;
```

Here, `compchar` is the index in the encoding vector of the composite character. `numPieces` specifies how many pieces are required to build the composite character. The third element is a pointer to an array of type `T1_COMP_PIECE`, whose actual length is given by `numPieces`: Each piece (or symbol) receives one slot in this array. `T1_COMP_PIECE` is defined by

```
typedef struct
{
  int piece;
  int deltax;
  int deltay;
} T1_COMP_PIECE;
```

It contains the encoding index of the symbol in `piece` and information where to place this symbol with respect to the composite character's orgin in `deltax` and `deltay`. The first slot is filled by what I refer to as the base character, it is the one that causes spacing. As can be seen in these data structures, composite character handling in `t1lib` is based on encoding indices rather than on character names, which, by contrast, are used for the definition of composite character data in AFM files.

The presence of composite character information in AFM files does not tell anything about whether a font has an internal definition of this character or not. For example, the font `TimesRoman` internally defines the CharString `Adieresis` and hence this font provides the letter "Ä", assuming an appropriate encoding, without any knowledge about composite characters. However, the file `TimesRoman.afm` may still provide composite character information for `Adieresis`, just to tell an application that this glyph consists of more elementary pieces and how to construct it. On the other hand, if `TimesRoman` had no CharString-definition for `Adieresis`, the composite character information of `Adieresis` provides an application with enough information to be able construct `Adiereis` from the elementary units `A` and `dieresis` that the font provides.

5.18.2 Accessing Composite Character Data

This section describes a few functions that provide access to composite character data of a font file. The data they return can be considered a mapping of the original AFM data with respect to the current encoding. As usual, `FontID` must be the identifier of a font loaded into memory, otherwise an appropriate error indicator is returned. The functions described in the following may also return some other error types.

Firstly,

$\mathcal{F}() \Rightarrow$ `int T1_GetNoCompositeChars( int FontID)`

tells the user how many composite character definitions are given in the AFM file. This, of course, does not depend on the current encoding vector and it is even possible that the current encoding vector does not incorporate any composite character at all.

The function

$\mathcal{F}() \Rightarrow$ `int T1_QueryCompositeChar( int FontID, char char1)` checks whether composite character information exists for the encoding index `char1`. If so, it returns the index within in the AFM composite character data array as a number equal to or greater than zero. If the result is valid but no composite character information has been found for `char1`, -1 is returned. In case of an error, -2 is returned and `T1_errno` is set to an appropriate value.

The previous function does not tell anything about whether the font `FontID` incorporates a character definition for the composite char or not. This can be queried using

$\mathcal{F}() \Rightarrow$ `int T1_IsInternalChar( int FontID, char char1)`

It returns 1 if there exists a CharString for `char1` and 0 if not. In the latter case, the application is responsible for the construction of the composite character (see later). `T1_IsInternalChar()` also might return -1 and set `T1_errno`, which indicates that the font in question is not loaded.

The information required to construct a composite character is retrieved by calling the function

$\mathcal{F}() \Rightarrow$ `T1_COMP_CHAR_INFO *T1_GetCompCharData( int FontID, char char1)`

It returns a pointer to a meaningfully filled struct of type `T1_COMP_CHAR_INFO` as described above. The composite character and the number of pieces as well as a pointer to the array of type `T1_COMP_PIECE` are stored in this structure. Once this information is obtained the composite character can be constructed by

- placing the accent symbols in a loop that ranges from 1 to `numPieces`-1. In practice this loop will often be executed only once. The initial current point must always be restored. In this loop positioning information is accessed by `ptr->pieces[`i`].deltax` and `ptr->pieces[`i`].deltay`, where `ptr` is the pointer returned by the above function.
- placing the base character (`pieces[0]`), which then also causes the horizontal escapement of the composite character.

In cases where `char1` is not a composite character, the `compchar` entry is set to `char1` itself and `numPieces` becomes 1, as would be expected. The `pieces` pointer is then set to `NULL`. In case of errors, this function returns `NULL` and `T1_errno` is setup correspondingly.

The pointer returned by this function should always be free'd using `T1_FreeCompCharData()` in order to avoid memory leaks (see also Section 5.8).

A function that provides the same functionality is

$\mathcal{F}() \Rightarrow$ `T1_COMP_CHAR_INFO *T1_GetCompCharDataByIndex( int FontID, int index)`

In this case, the information is requested by means of an index `index` in the AFM composite character data array. This function is thus well-suited for scanning the complete composite character information of a given font in a given encoding. `index` may, for example, be obtained by a call to `T1_QueryCompCharData()` as described above. The valid range for `index` is from 0 to the value returned by `T1_GetNoCompositeChars()` minus one. The range of `index` is validated and in case of an error `T1_errno` is to `T1ERR_INVALID_PARAMETER`. There may also appear other errors and under any erroneous condition, `NULL` is returned.

The parameter `compchar` of the `T1_COMP_CHAR_INFO` structure that is referenced by the returned pointer, bears somewhat more information for this function than in the case of `T1_GetCompCharData()`. As said it contains the index in the encoding vector where the composite glyph is encoded. It may also have the vaue -1, which means that the composite character is not encoded. Note that this is not an error condition.

5.18.3 Transparent Handling of Composite Characters and User Extensions

Aside from the fact that composite character information may be accessed in `t1lib`, `t1lib` can automatically—and completely transparently with respect to the user—compose characters if it finds information on how to do so. To give an instance, let us examine the font Computer-Modern Roman (cmr10). Because this font is specially encoded for the use with early (7-bit) TeX-systems, it does not incorporate a definition for `Adieresis`. The definition simply was not required because TeX itself constructed the composite character by means of its `\accent`-primitive. If a font like cmr10 is reencoded e.g. to IsoLatin1 encoding, the character "Ä" will show up as a blank because there is no definition for `Adieresis`. Now, if the corresponding AFM file is extended with the following lines, it becomes possible to access an `Adieresis`:

```
StartComposites 1
CC Adieresis 2; PCC A 0 0; PCC dieresis 100 200;
EndComposites
```

This line in an AFM file provides information about how to construct an `Adieresis` from the `A` and `dieresis` glyphs, and `t1lib` can utilize this information to construct the requested glyph without that this will be be noticed by the user.

Composite character are treated just the same way as standard characters. They are cached, they can be scaled, transformed etc. Let us assume that the cache is still empty and a character, identified by its encoding index, now is to be rastered. The following happens in the rastering function:

1. `t1lib` looks up the character's name in the encoding vector and tries to locate the CharString for that character. If this succeeds all works as usual, notwithstanding the fact that there might have been composite character information for that char. This means, font-internal CharString-definitions have highest priority: One cannot, for example, re-define an `Adieresis` by raising the umlaut "¨" via a composite character definition, if `Adieresis` is defined internally.

2. However, if the CharString is not found, composite character information is examined and if possible, elementary units are used to construct the requested composite character by concatenating paths.

In any case, the resulting character is put into the cache and is from then on available as any other character. If pieces of a composite character are not found in the CharStrings dictionary, those pieces are substituted by `.notdef`, so that for extreme cases the whole composite character might be substituted by a `.notdef`. Then, an appropriate message is put into the log file with priority `T1LOG_WARNING` and `T1_errno` is set to `T1ERR_COMPOSITE_CHAR`.

In the same way `t1lib` composes characters without user intervention, the functions for character metrics are aware of composite character information and the returned result are also valid for those functions.

5.18.4 Caveats

Although handling of composite character is widely automated, problems may arise. Most importantly it is the responsibility of the user to take care that font file and AFM file provide consistent data. Alas, this is not always true for existing font and AFM files. If, for example, an AFM file is extended by composite character definitions and these composite character definitions reference symbols that are not defined in the CharsStrings dictionary, errors will result. If composite character information is added to an AFM file, the following rules have to be respected:

- The name of the composite character has to be encoded because it could not be accessed otherwise. Furthermore, no internal definition in the CharStrings dictionary may exist because this would override the composite character definition from the AFM file.
- The user should verify that all components of a composite character definition have entries in the CharStrings dictionary. This can be checked for example by using `T1_GetAllCharNames()` or by disassembling the font file.
- Even being more restrictive, the user should take care that all pieces of a composite character are encoded. For `t1lib` this is really irrelevant because, internally, characters may be accessed by the name of their CharString. This means `t1lib` simultaneously has access to all characters defined in a font. However, an application that exports PostScript files can only access character definitions via their position in the encoding vector. Composing a character from pieces of different encodings will require two font definitions in a exported PostScript file for typesetting one character, which cannot be termed a clean strategy.

5.19 Error Handling

Although every function usually returns meaningful values, there are situations where indicating an error via the return value is not possible. For example, requesting a charspace bounding box from a char of a font which is not loaded will return a bounding box containing all zeroes. This cannot be considered an error-condition since for characters like "space" it is correct to return a bounding box containing all zeroes. Furthermore, there's no consistent scheme which value should indicate what type of error. In order to allow a unified error handling in applications, the global variable `T1_errno` has been introduced.

The functionality of `T1_errno` is analogous to that of the global `errno` in C programs. `T1_errno` is once set to 0 when the library is initialized and never reset by any `t1lib`-function. It is set to specific values when specific types of errors appear. An application may then act appropriately and reset `T1_errno`. The errors that might appear can be roughly split into three categories as described below.

5.19.1 Type 1 Font File Scan-Errors

These types of errors can only appear at the time a font file is loaded. These kinds of errors are indicated by negative values:

- `T1ERR_SCAN_FONT_FORMAT` (-5): A Multiple Master Font was attempted to be loaded. These are not supported by `t1lib`.
- `T1ERR_SCAN_FILE_OPEN_ERR` (-4): This value indicates that the Type 1 font file could not be opened by the parser. It usually does not mean that the file does not exist because this problem would have shown up at the time the font database had been built. It is more likely a permission problem. Anyhow, the C library variable `errno` should be examined for getting an idea of what the problem was.
- `T1ERR_SCAN_OUT_OF_MEMORY` (-3): A Type 1 font program required more than 262144 bytes of VM. This is a limit imposed by `t1lib` because it usually means there goes something wrong. Typical values of VM consumption are between 30000 and 60000 bytes depending on the fonts' complexity. If this limit really does not suffice the constant `MAXTRIAL` (defined in `lib/type1/fontfcn.c`) may be set to some larger value.
- `T1ERR_SCAN_ERROR` (-2): An error occurred during scanning the font file. It usually means that the font file is damaged or does not comply to the conventions of Type 1 font files.

For example, an encountered token might have been too long. Another reason could be, a literal name follows a literal name where a number was expected. There is no way to recover from this error. One last resort could be to disassemble the font (e.g., using `t1disasm` from the `t1utils` package) and scan the resulting human-readable file for possible violations of the Type 1 font format specifications. However, some knowledge about the format is in force.

- `T1ERR_SCAN_FILE_EOF` (-1): A premature end of file was encountered during parsing. The file is damaged.

5.19.2 Path Generation Errors

Small positive number are reserved for errors that might appear during path construction and rasterization.

- `T1ERR_PATH_ERROR` (1): An error occurred during path construction. The font file is most probably damaged.

- `T1ERR_PARSE_ERROR` (2): This kind of error describes a kind of "semantic" error in the font file. A typical candidate for this is a font that does not define a character named `.notdef`, although this is required by the format specification. Since under usual conditions the `.notdef` character is never accessed, this error would not show up. But if for some reason the `.notdef` has to be substituted for some other character the problem becomes evident.

- `T1ERR_TYPE1_ABORT` (3): The `abort()`-function of the rasterizer has been called. This may happen at several places during hinting, converting to edgelists etc. There is a certain chance that unfreed memory has been left. If this error appears and a logfile is used, an error string giving some more info is placed into the logfile.

  This error should not appear, normally. If it does, either the font file is damaged or the font contains invalid outline descriptions such as unclosed paths. Especially the latter is quite unlikely. Of course this error can be raised, when an outline has been modified manually in an invalid way and is then rastered (see. 5.14.3).

5.19.3 `t1lib`-Errors

The remaining types of errors are detected by the management of `t1lib`. Their numbering starts with 10 (decimal). The list could be extended in future releases.

- `T1ERR_INVALID_FONTID` (10): An invalid font ID has been specified. The exact meaning of this error depends on the specific situation, in any case the operation requested cannot be realized with the identified font. Possible reasons are:
  - The font ID points to a font which is not loaded and which must be loaded in order to perform the operation.
  - The specified font ID is a number which is generally out of the range of the valid font IDs, either because it is < 0 or because it is $>$ the value of `no_fonts`.
  - The library is not yet initialized so that no font ID at all is valid.

- `T1ERR_INVALID_PARAMETER` (11): One or more of the parameters specified to a function call were assigned invalid values. For example, a size-value specified to a rastering function must always be > 0. Just the same way, `T1_ConcatGlyphs()` cannot concatenate two glyphs if one of them is the `NULL` pointer.

- `T1ERR_OP_NOT_PERMITTED` (12): An operation that was not allowed *at that time* has been requested. This error could result, for example, if an application tries to set a new bitmap padding value after `t1lib` has been initialized.

- `T1ERR_ALLOC_MEM` (13): This error indicates that `t1lib` ran out of memory and a memory allocation failed. This error should not appear.

- `T1ERR_FILE_OPEN_ERR` (14): A file that was needed could not be opened by `t1lib`. The file might have been necessary for reading data or writing data. For example, `T1_WriteAFMFallbackFile()` returns this value if the AFM file could not be opened for writing and `T1_LoadEncoding()` returns it if the encoding file specified as argument could not be opened. Notice that there is no indication of the reason why the file opening failed. The C library variable `errno` should be examined to analyze this further.

  It should be mentioned that `T1ERR_FILE_OPEN_ERR` is only set if a file operation failed which was really in force. This means that at the time a font is loaded a missing AFM file does not cause `T1_errno` caused to be set to `T1ERR_FILE_OPEN_ERR`. This is because `t1lib` can automatically recover from this by generating AFM information on the fly (at the cost of computation time).

- `T1ERR_UNSPECIFIED` (15): This value indicates nothing apart from that an error occurred and this error was not one the other errors. It can be considered a fallback.

- `T1ERR_NO_AFM_DATA` (16): A function has been called which needs AFM information and AFM information is not available, either because all attempts to generate AFM data failed or because the flag `T1_NO_AFM` has been specified as part of the flag for `T1_InitLib()`.

- `T1ERR_X11` (17): An error in an X11 library function occured. This could be caused by calling a function of the X11 interface without prior initialization of the X11 interface via `T1_SetX11Params()`.

- `T1ERR_COMPOSITE_CHAR` (18): A request to compose a composite character could not be fulfilled without problems because at least one part of the composite character was not found in the `CharStrings` dictionary. This is bad because it indicates that font file and AFM file do match. Further errors or unsatisfactory rastering results have to be expected.

- `T1ERR_SCAN_ENCODING` (19): Scanning an encoding file failed. Since `t1lib` uses a fallback approach—DVIPS-encoding is tried first and afterwards `t1lib`-encoding—it is not clear at which place exactly a failure occured. However, further hints about what `t1lib` thought about the file in question may be found in the log file.

In analogy to the Standard C Library function `strerror()`, `t1lib` provides the function

$\mathcal{F}() \Rightarrow$ `const char *T1_StrError( int t1err)`

It returns a pointer to a string describing the error corresponding to `t1err`. Usually, the argument should be directly specified as `T1_errno`. The memory where the returned string is stored is static in `t1lib` so that it may not be `free()`'d.

5.20 Other Useful Functions

This subsection describes a few functions that had not been described up to now but which however could be useful.

$\boxed{\mathcal{F}() \Rightarrow}$ `int T1_CheckEndian( void)`

This function may be used to check the endianess of the hardware `t1lib` is running on. The return value is `0` for Little Endian and `1` for Big Endian machines.

$\boxed{\mathcal{F}() \Rightarrow}$ `void T1_DumpGlyph( GLYPH *glyph)`

This function might be useful for debugging and testing `t1lib`. It dumps an ASCII representation of the glyph pointed to by `glyph` to the standard output. A background pixel is represented by `.` while a foreground pixel is represented by `X`. After the number of bits that correspond to the current padding value, an empty column is inserted. See the output of the programming example in 2.4. In this case the padding values has been 16.

Note that the size of the glyph should be small enough that its padded width does not exceed the terminals line width. Otherwise the result might become illegible.

$\boxed{\mathcal{F}() \Rightarrow}$ `void T1_DumpPath( T1_OUTLINE *path)`

This function dumps a description of an outline to the standard output. It is exclusively intended for debugging purposes.

$\boxed{\mathcal{F}() \Rightarrow}$ `void T1_SetRasterFlags( int flags)`

This function allows to enable or disable certain features of the rasterizer. Let me emphasize that this is exclusively intended for debugging and error tracking. The default value of `flags` is 0 which means that no debugging output is shown and hinting is performed as suggested in the *Adobe Type Font Format*. However there may arise situations where fiddling with the `flags` might be helpful in rasterizer and font debugging.

`flags` usually is an OR'ed combination of the following definitions:

- `T1_IGNORE_FORCEBOLD`
- `T1_IGNORE_FAMILYALIGNMENT`
- `T1_IGNORE_HINTING`
- `T1_DEBUG_LINE`
- `T1_DEBUG_REGION`
- `T1_DEBUG_PATH`
- `T1_DEBUG_FONT`
- `T1_DEBUG_HINT`

The `T1_IGNORE_...` types allow to selectively disable hinting. They might be useful if parts of a font are not properly rendered. For example, substituting a font's alignment zones by the family's alignment zones might result in visual artifacts if the values for `FamilyBlues` are not correct. Disabling family alignment might reveal the problem in such cases.

The `T1_DEBUG_...` types produce debugging output from the intermediate rasterizing steps. Notice that to understand this output a thorough understanding of what happens in the rasterizer is in force. Moreover, be prepared that thousands of lines might be written to the terminal, depending on the particular option.

6 The X11-Interface

As of V. 0.3-beta `t1lib` incorporates some special functions that can make life easier for X11 programmers. Prior to further discussions it should be noted that these functions, which itself use functions `Xlib` are fully optional so that `t1lib` can still be compiled and used on systems without X11.

6.1 Why a Special X11-Interface?

Although it initially was an explicit goal to make the library independent of X11, there are some strong arguments voting for a subset of functions adapted to X11 features. Here are some of them.

- X11 can be considered a standard under UNIX and graphical applications under UNIX would to a high probability rely on X11. For this reason it would not degrade portability much when using X11 features.

- The standard rastering functions of `t1lib` strictly use the principle of *generating bitmaps*. The rastering functions return bitmaps of a specific size with reference point at upper left corner and additional geometric information to tell the user how to position the bitmap correctly with respect to the current point. A more natural approach would be to have functions that draw on some existent *area* at a position and orientation to be specified with the logical origin of the text taken into account.

- The standard functions do not deal with color at all. This is especially complicated in the case of antialiased fonts. If some application wanted to use an existent (already cached) font, all characters would have to be removed from memory and recreated using the new color values.

- In case of antialiasing the user has to make a decision on the depth of the bitmaps. But what if an X-application uses drawables of different depths? Such configurations could raise the programming effort up infinity (well, alomst).

- The standard functions cache their bitmaps locally, i.e., on the machine where the X11-client runs. Every characters bitmap has thus to be transferred to the X11 server again and again. This might cause an performance degradation, especially if the client runs on a remote machine with a slow or heavy-loaded network connection.

Taken these arguments into account I decided to create an additional "special" set of functions that allow uncomplicated usage under the X11 window system, similar to the `Xlib`-function `X11DrawText()`. The approach used in V. 0.3-beta has been a quite elegant solution to the problems considered above. Unfortunately, it has been too slow to be usable in practice. Furthermore, caching in the X11-server produced some overhead and difficulties. According to my experiences server caching would only be advantageous for very large characters such as 500 bp and more. As a consequence, the X11 interface is redesigned and reduced to a *simple wrapper* which deals with all but the last of the above items, from `t1lib` V. 0.4-beta up.

6.2 Initialization of the X11-Interface

There are a few things `t1lib` must know in order to be able to do its job properly. These parameters are defined by a functioncall to

```
int T1_SetX11Params( Display *display, Visual *visual,
                     unsigned int depth, Colormap colormap)
```
$\mathcal{F}() \Rightarrow$

`display` is the pointer to the structure of the display the application is connected to. It is once specified here because this avoids the need of repeatedly having to specify this pointer.

`visual` is the pointer to the structure of the visual on which `t1lib` should create the XImages that will be transferred to the X server when using antialiasing. In most cases it should be safe to specify `DefaultVisual(...)`.

The `depth`-argument specifies the depth that `t1lib` will use when creating antialiased pixmaps. It is thus identical to the value `bpp` supplied to `T1_AASetBitsPerPixel()` (see 5.13). The depth must be one of the depth supported by the visual as specified above.

`colormap` is the specification of an X11 colomap. It is needed because `t1lib` might need to allocate some more colors for antialiasing purposes. The same colormap that the application uses should be specified here. If the application uses no special color handling, `DefaultColormap(...)` is probably the right value.

As mentioned before, the X11 rastering functions put the characters with their origin at the specified point. This behavior, being the default, can be switched by calling

$\mathcal{F}() \Rightarrow$
```
void T1_LogicalPositionX( int pos_switch)
```

Specifying `pos_switch`=0 has the effect that in subsequent calls to X11 rastering functions the result will be placed with the upper left corner at the specified position. The default behavior can be restored by calling this function again with some non-zero value.

6.3 Rastering Functions

In analogy to the standard rastering functions the `t1lib` X11 interface provides six functions for generating character and string bitmaps, rectangles as well as their antialiased equivalents:

```
GLYPH *T1_SetCharX( Drawable d, GC gc, int mode, int x, int y,
                    int FontID, char charcode,
                    float size, T1_TMATRIX *transform)
```
$\mathcal{F}() \Rightarrow$

```
GLYPH *T1_SetStringX( Drawable d, GC gc, int mode, int x, int y,
                      int FontID, char *string, int len,
                      long spaceoff, int modflag,
                      float size, T1_TMATRIX *transform)
```
$\mathcal{F}() \Rightarrow$

```
GLYPH *T1_SetRectX( Drawable d, GC gc, int mode, int x_dest, int y_dest,
                    int FontID, float size,
                    float width, float height,
                    T1_TMATRIX *transform)
```
$\mathcal{F}() \Rightarrow$

```
GLYPH *T1_AASetCharX( Drawable d, GC gc, int mode, int x, int y,
                      int FontID, char charcode,
                      float size, T1_TMATRIX *transform)
```
$\mathcal{F}() \Rightarrow$

```
GLYPH *T1_AASetStringX( Drawable d, GC gc, int mode, int x, int y,
                        int FontID, char *string, int len,
                        long spaceoff, int modflag,
                        float size, T1_TMATRIX *transform)
```

$\boxed{\mathcal{F}() \Rightarrow}$

```
GLYPH *T1_AASetRectX( Drawable d, GC gc, int mode, int x_dest, int y_dest,
                      int FontID, float size,
                      float width, float height,
                      T1_TMATRIX *transform)
```

$\boxed{\mathcal{F}() \Rightarrow}$

Instead of explaining everything in detail, we will discuss only those items that are different from the standard rastering. For discussion of the parameters `FontID`, `charcode`, `string`, `len`, `spaceoff`, `modflag`, `size` and `transform` see 5.5.

The `drawable` parameter specifies the X11 drawable into which the text will be drawn. It may be either a pixmap or a window.

`gc` is the graphics context in which the operation should take place. Obviously, the most importamt components of the graphics context are the current foreground and background color. `t1lib` uses these colors to draw the text/background.

The value of `mode` determines whether *opaque* or *transparent* mode is used. In opaque mode all pixels including background pixels are drawn. This means the whole area of the bounding box of the character/string is painted. In transparent mode, only non-background pixels are drawn so that underlying graphics are minimum affected. One could imagine that as drawing the text on a transparent slide and overlay the result with the existent graphics. Using transparent mode should be somewhat slower, especially for the antialiasing functions since the information which pixels are background and which not has to be generated first. Moreover, the clipmask of the current graphics context is modified when drawing text in transparent mode.

`x` and `y` define the position coordinates where the origin of the text will be located in the drawable. `t1lib` does no checking of this position so that bad positioned text might not appear at all in the drawable without getting an error—it is simply clipped to the limits of the drawable.

The non-antialiasing functions will take the fore- and background color from the specified graphics context. The antialiasing rastering functions work a little different. They also respect the current foreground and background color. The "graylevel" colors are computed on the fly and allocated in the colormap specified by the user in the call to `T1_SetX11Params()`. The term graylevel has been quoted since these are in fact no graylevels at all. They are just discrete colors from a smooth bleed between the foreground and the background color. The colors are computed the following according to the following scheme:

1. The current foreground and background color are split into their respective RGB components, lets call them f_R, f_G, f_B, b_R, b_G and b_B.

2. Intermediate values are computed between each pair of color components f_R–b_R, f_G–b_G and f_B–b_B by making a linear interpolation.

3. For each newly created RGB-triple (currently 3) a pixel color is allocated in the colormap specified by the user in a way that the pixel colors match the theoretical values best.

A general comment on antialiasing: If using transparent mode antialiasing may produce the opposite effect of what is wanted, depending on the color of the underlying graphics. You can use the program `xglyph` to check the effect: Start `xglyph` and prior to doing anything else click on button [AAStringX]. The resulting glyph is antialiased against the background color white but the real background due to transparency is quite different from white. Consequently the glyph seems to be surrounded by a thin light gray border.

6.4 Creating XPM-Files from `t1lib`-Glyphs

The creation XPM files is not supported directly in `t1lib`. Rather, there is a utility function which prepares what is necessary and leaves the creation of the Pixmap file to the one and only authority, to the Pixmap library. The required function is:

$\mathcal{F}() \Rightarrow$ `XImage *T1_XImageFromGlyph( GLYPH *glyph)`

It creates an X11 image from a valid `t1lib`-glyph of arbitrary depth, padding and antialiasing configuration and returns the pointer to the newly created structure. This image can later be dumped into an XPM file using the XPM library function `XpmWriteFileFromImage()`. The following code fragment shows how easy this really is. It assumes that the program containing that code is additionally linked with the XPM library.

```
.
.
.
ximage=T1_XImageFromGlyph( glyph);   /* generate ximage containg the glyph */
/* write pixmap file */
XpmWriteFileFromImage( display, "glyphtest.xpm", ximage, NULL, NULL);
ximage->data=NULL;
XDestroyImage( ximage);
.
.
.
```

As already shown in this example code, the user has to take care for that `XDestroyImage()` does not free the glyph's bitmap. This achieved by setting `ximage->data` to `NULL`.

Having an XPM file from a `t1lib`-glyph it should easily be possible to create graphic files of arbitrary formats, e.g., by using `xv`.

6.5 Limits of the X11 Interface

A few words about what the X11 interface can do and cannot do are appropriate, I think. Except for that a few global variables of `t1lib` are accessible, the whole code of the X11 interface could as well be part of an application instead of being part of `t1lib`. In other words, `t1lib` is not able to do anything an application program could not do. This applies especially to performance improvements. Unfortunately—but consequently, the X11 rastering functions are not faster than the standard rastering functions.

To come to a conclusion, this is what the X rastering functions offer to the user:

- Save the user entirely from thinking about color.
- Offers a set functions comparable to `XDrawText()`.
- Frees the user from having to think about transparency and opacity.
- Implements antialiasing between any given pair of foreground/background colours.

And here is what they not provide:

- Functions that perform as if the rasterizer would be part of the X server.

That's life, folks.

7 Internals (incomplete)

Note!

This section is still very incomplete and some facts are not true anymore. This should be kept in mind. Currently I have no time to write this section. But I try to keep figure 7 consistent to the current releases. This may lead to inconsistencies between the text and the figure.

In this section, some information on internals of `t1lib` is given. There is no need for an average user to read this section although having understood what is going on internally might be helpful if problems occur.

The basic idea of this section is to describe the data structures and to give information on when they are initialized, allocated and referenced. Figure 7 shows an image of the data-structures for the special case that the font with ID 0 has already been loaded and several size-instances have already been created. As the figure indicates, the complete area may be split into three different sub-areas, thereby pointing out their logical functions.

7.1 Level 0: Global Data

This area contains information needed for the overall organization of the `t1lib`. Its contents and its size are thus determined at the time `t1lib` is initialized. This is done based on the contents of the configuration file and the fontdatabase file. The entries in detail are:

- `Filename-Searchpaths`: This entry essentially does not depend on any other data. It consists of 4 `\0`-terminated strings that are read from the configuration file. They are referenced internally by the global symbols `PFAB_ptr`, `AFM_ptr`, `ENC_ptr` and `FDB_ptr` respectively. All these are declared as `unsigned char *`. These strings are used by `t1lib` to locate the respective file types. If no configuration file exists or some path declaration is missing, the corresponding searchpath is set to "`.`", causing `t1lib` to only search the current working directory.

- `no_fonts_ini`: This value is assigned after examining the fontdatabase file. It is meant to store the number of fonts initially declared in the fontdatabase file. In other words, it is assigned the integer number located on the first line of the fontdatabase file.

- `no_fonts`: The number of actually allocated fonts. Initially, this quantity is identical to `no_fonts_ini`. But if one creates a new logical font by calling `T1_CopyFont()` this counter is incremented to keep track of allocated fonts. `no_fonts` thus represents most large `FontID` minus 1 that makes sense to specify to any function of `t1lib`.

- `no_fonts_limit`: The number of fonts for which memory is currently allocated. This also is initially set to `no_fonts_ini` and is automatically enlarged to a multiple of the initial value if a call to `T1_CopyFont()` requires additional memory for logical fonts (see 5.15).

- `bitmap_pad`: This variable contains the number of bits to which scanlines of bitmaps and antialiased bitmaps are padded. It is set during initialization, either to a default value or to the value the application specified before starting initialization using `T1_SetBimapPad()`. Allowed values are currently '8', '16' and '32'.

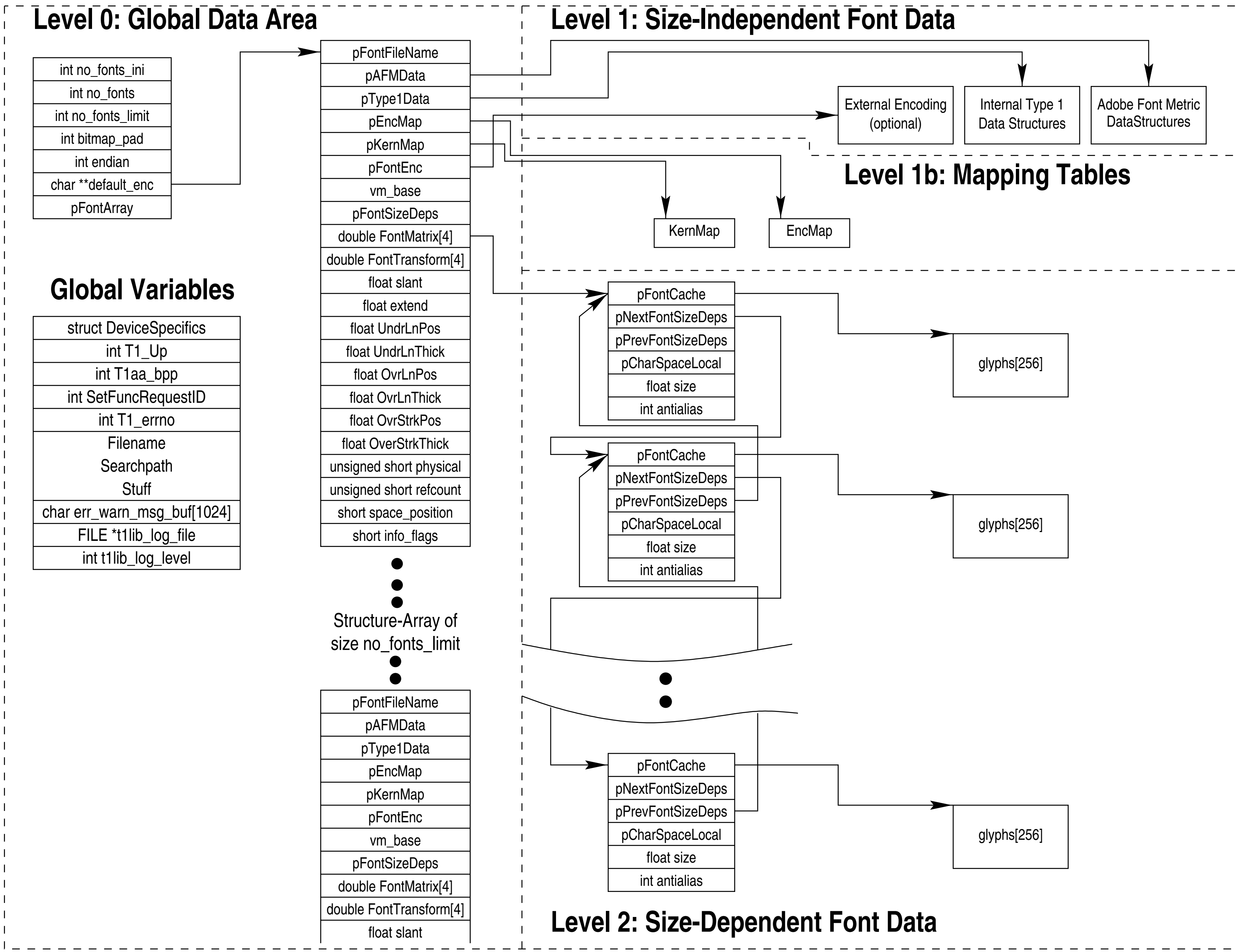

Figure 7: The internal data structures of `t1lib`. The underlying substructures are shown only for the first font `FontID=0`.

- `endian`: During initialization the hardware is checked for representation of data in memory. If Big Endian is used, `endian` is set to `1` and otherwise it is set to `0`. `endian` is needed at several times when an application or `t1lib` itself must know the byte order of words and long words.

- `pFontArray`: This a pointer to an array of structures whose type is referred to as `FONTPRIVATE` in `t1lib`. The contents of these structures will be described below. After `t1lib` has been initialized, memory is allocated for exactly `no_fonts_ini` structures. This memory pool may be enlarged later if the one wants to make use of logical fonts, for example. The data in these structures initially is not specified. It is written with meaningful values when a font is loaded into memory. The index to access this array-elements is the well known font identification number (`FontID`).

- `pFontFileNameIDArray`: A pointer to a memory area where the font file names corresponding to the `FontID`s are stored. During initialization, `t1lib` looks for font files with extension `.pfa` and `.pfb`. The basename of the file found is stored in this area and if the font is to be loaded later, its font file name is looked up here.

We should now discuss the entries of the structures of type `FONTPRIVATE`. The term `FONTPRIVATE` indicates that every font needs its own structure area. As mentioned earlier, this area is initialized when the corresponding font is loaded.

- `pAFMData`: A pointer to a memory area where Adobe Font Metric data of the font is stored. The memory area itself is build by the `parse_afm`-package which is supplied by Adobe System and included in `t1lib`. This happens while a font is loaded. In case there is no AFM file for the font in question, this pointer is given the value `NULL`.

- `pType1Data`: A pointer to the data area where the Type 1 information is stored. The known PostScript Type 1 objects Charstrings-dictionary, Subroutines, Othersubroutines and Fontinfo-dictionary are located here. The memory is filled with data during parsing the font file when the font is loaded.

- `pFontEnc`: A pointer to an optional external encoding vector. During initialization, this pointer is set to `NULL`, thus indicating that by default the font's internal encoding should be used. If a font is reencoded using a previously loaded encoding vector from an encoding file, this pointer simply is assigned the address of a valid encoding array somewhere in memory.

- `vm_base`: The base address of the virtual memory required by the font. Unlike the original rasterizer, which allocated virtual memory in chunks of a fixed size, t1lib uses another principle. Since it is à priori not obvious how many virtual memory a font consumes, `t1lib` tries to load a font repeatedly and increases the amount of virtual memory during every trial. In order not to waste memory, the memory is reallocated to the needed size when the font is completely loaded. Finally, the starting address of the virtual memory is needed when a font is to be unloaded and the memory it consumes is to be given back to system.

- `pFontSizeDeps`: A pointer to the area where the size dependent data is to be stored. This data essentially consists of generated glyphs plus some administrative item (see 7.3).

- `FontMatrix`: A matrix of four `double`-values specifying the font matrix. If the FontInfo-dictionary of the font file defines a FontMatrix, it is copied to this location. If not, a default matrix is used which does no transformation and scales to 1/1000 bp.

- `FontTransform`: A matrix that will be concatenated with the FontMatrix to produce the final transformation of the characters. It is this matrix that is modified if a font is to be slanted or extended.

- `slant`: A slant factor for the current font. Note that this value is initially 0, even for italic font. Only artificially slanting a font leads to values different from 0.

- `extend`: The horizontal extension factor for the current font. Its default value is 1 and the font is thus rendered at its natural width.

- `physical`: This is a switch that marks a font either being "physical" or "logical". A physical font by definition is a font for which a Type 1 font file is available and for which thus Level 1 (size-independent) data is present (see Fig. 5.1). In contrast, the term "logical font" refers to a structure of type `FONTPRIVATE` whose entry `pType1Data` points to Level 1 data of another (physical) font. This `FONTPRIVATE`-structure is created by calling `T1_CopyFont()` with the identification number of an existing physical font as argument (see 5.15).

- `refcount`: This counter keeps track on how much logical fonts refer to the physical font that is represented by the current structure of type `FONTPRIVATE`. In this since, `refcount` is only meaningful for physical fonts. It is necessary to keep track of the reference of logical fonts because if this font would be removed from memory by calling `T1_DeleteFont()`, the Level 1 font data memory area would be given back to the system but the logical fonts referring to that font would still expect to find Type 1 or Font Metric data at this address. By checking `refcount`, `T1_DeleteFont()` can check for logical fonts referring to the font in question and prevent from removing this font from memory.

  In structures describing logical fonts, `refcount` is used to store the information which physical font this logical font is referring to. This information is also needed by `T1_DeleteFont()` since when removing logical fonts, the reference counter of the corresponding physical font has to be decremented.

- `space_position`: This variable stores the encoding index of the "space"-character of the current font. If the space character does not appear in the current font's encoding, `space_position` is assigned -1. It follows that `space_position` is assigned when (1) a font loaded and (2) every time a font is reencoded. Why is it convenient to store the position of the space character in the encoding vector? The properties of the space character are set apart from the other characters' properties not only by the fact that it does not produce any colored pixels but also by that it may shrink and stretch in `t1lib`. As a consequence a space character is treated by simply inserting a horizontal escapement of the width of the space character—corrected by the quantity `space_off` that a user may specify (see 5.5). This involves always checking every character for being the space character and since the encoding principle is used in `t1lib`, every check needs a call to `strcmp()`. This overhead is avoided if the position of space is stored.

7.2 Level 1: Size-Independent Font Data

Size-independent data may be split into three categories as indicated in figure 7. The external encoding is optional and is generated by loading an encoding file as described in 5.7. It is simply an array of 256 pointers to `unsigned char` and an ensemble of 256 `\0`-terminated strings. Each pointer references one of the 256 strings in order. The strings are the characters' names to be defined in a `t1lib`-encoding file.

The internal Type 1 data structures hold all data specified in a type font file. I do not want to describe these data structures here, because this could fill a book. Adobe has made the description of the Type 1 font format available to the public.

The Adobe Font Metrics area is entirely created by the `parse_afm`-package. Adobe has made this available by means of the file `parseAFM.shar` which is a shell-archive and included in `t1lib` in the subdirectory `parse_afm`.

7.3 Level 2: Size-Dependent Font Data

...

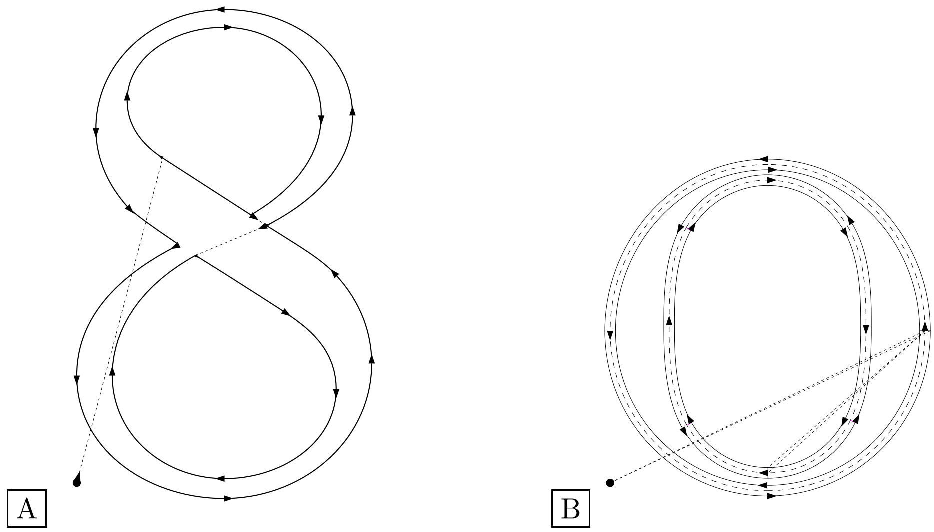

Figure 8: $\boxed{\text{A}}$ Character "8" from font ComputerModern Roman. The arrows indicate the direction of the paths. From the outer subpath it follows that the inner region will be filled (left of the path). From this massive black region, the holes are cut by means of the two inner subpaths (and their direction). $\boxed{\text{B}}$ The principle of creating a stroked character by filling a newly created set of subpaths which surround the original path in an appropriate manor.

8 Stroked Characters

This section is only meant for the reader interested in details about the algorithm used to create stroked versions from outlines intended to be filled. It can help to understand the code I added to `type1.c`, which may seem a little bit strange.

The basic idea to achieve stroked outlines was to map the stroking operation to a simple filling operation as already implemented by the rasterizer. Why did I choose this approach? Well, the actual reason for doing so was that I felt like doing so. One of the pivotal problems in this context turned out to be the computation of a third order Bezier curve, being located *in parallel* to a given third order Bezier curve—a problem set which everybody on the net said to be impossible to solve. After some experimenting I had to admit that these people actually were right: It is not possible to solve this problem in general, in particular because tracing a given cubic Bezier spline using a finite pen width might produce delimiting curves which aren't Bezier splines at all. In particular, the angular range and the pen width in relation to the original curve's bend are of importance.

However, under some constraints, which usually are fulfilled by adhering to the Adobe design rules for Type 1 Fonts and by choosing reasonable stroke widths, it is possible to approximate these delimiting curves by cubic Bezier splines.

8.1 Approach

Type 1 character outline descriptions consist of mathematically thin defining curves and lines with an associated running direction. By convention, regions left of these defining curves are painted and regions right of these curves are left blank. For each properly defined character, this way, a finite area to be filled results by applying this rule, especially because for filled characters every subpath must be closed. Figure 8 $\boxed{\text{A}}$ shows the character "8" from the ComputerModern Roman font as an example. We find three subpaths which by means of their direction relations

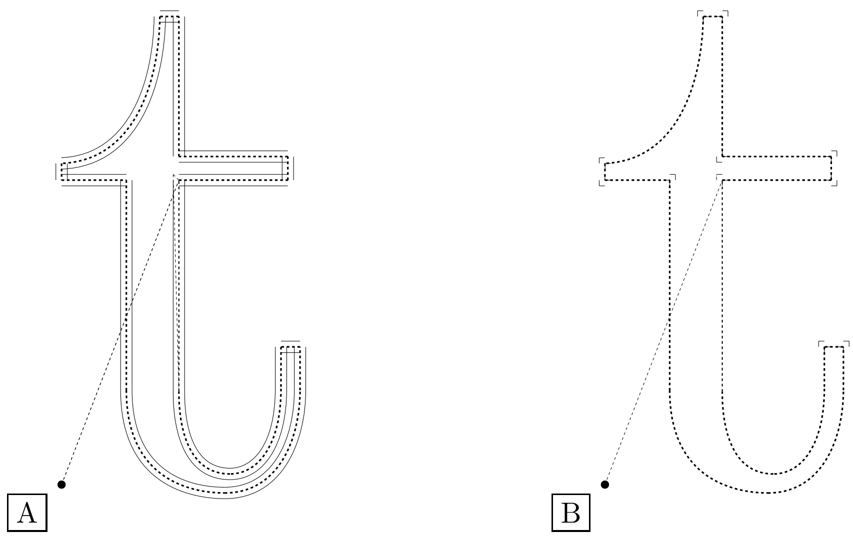

Figure 9: $\boxed{\text{A}}$ Character “t” from font ComputerModern Roman. The original path is shown in a thick dashed style. Each segment is surrounded by a parallel path to the right hand side and a parallel path to the left hand side. $\boxed{\text{B}}$ Required additional connection segments in order to complete the outline path.

yield the filled character.

When talking about *stroking*, we mean tracing a pen of finite width along these subpaths. When doing so, a new finite (more complex) region of ink is built. Actually we can consider this filled region being the result of filling a newly created path that consists of two subpaths surrounding the original path and having appropriate directions. These newly created subpaths are referred as the *right path* and the *left path*. Figure 8 $\boxed{\text{B}}$ illustrates this idea for the character “o”. The original path is represented by dashed curves whereas left paths and right paths are shown as solid curves. The respective directions are indicated by arrows.

Now, what are the steps required to compute a right path or left path from a given path and given a certain strokewidth? Firstly, for each path segment two *parallel paths* the right and the left path, located half the strokewidth right and left of the original path have to be computed. This is shown for the character “t” in Figure 9, $\boxed{\text{A}}$.

In particular, it turns out that in order to connect two parallel right or left paths of two neighboring original path segments, additional path segments are required. Therefore, in a second step, these parallel path segments—which in general may be disjoint—have to be connected appropriately. These additional path segments are shown in $\boxed{\text{B}}$ of the figure. We term these path segments *Prolongation Segments*. They are always built as straight lines. It is also obvious, that for convex edges, prolongation actually is what it indicates, and for concave edges, some trick must be applied so that prolongation yields a path that actually *shortens* the respective parallel path segments.

8.2 Computation of Parallel Paths

For straight lines, the notion of a parallel path in distance $w/2$ immediately becomes evident, but what about Bezier curves? Let us define the parallel of a curve as the infinite set of points, that results from tracing along the curve and for each point of the curve computing the point which in direction orthogonally to the curve’s tangent at the respective location is just the distance $w/2$ apart.

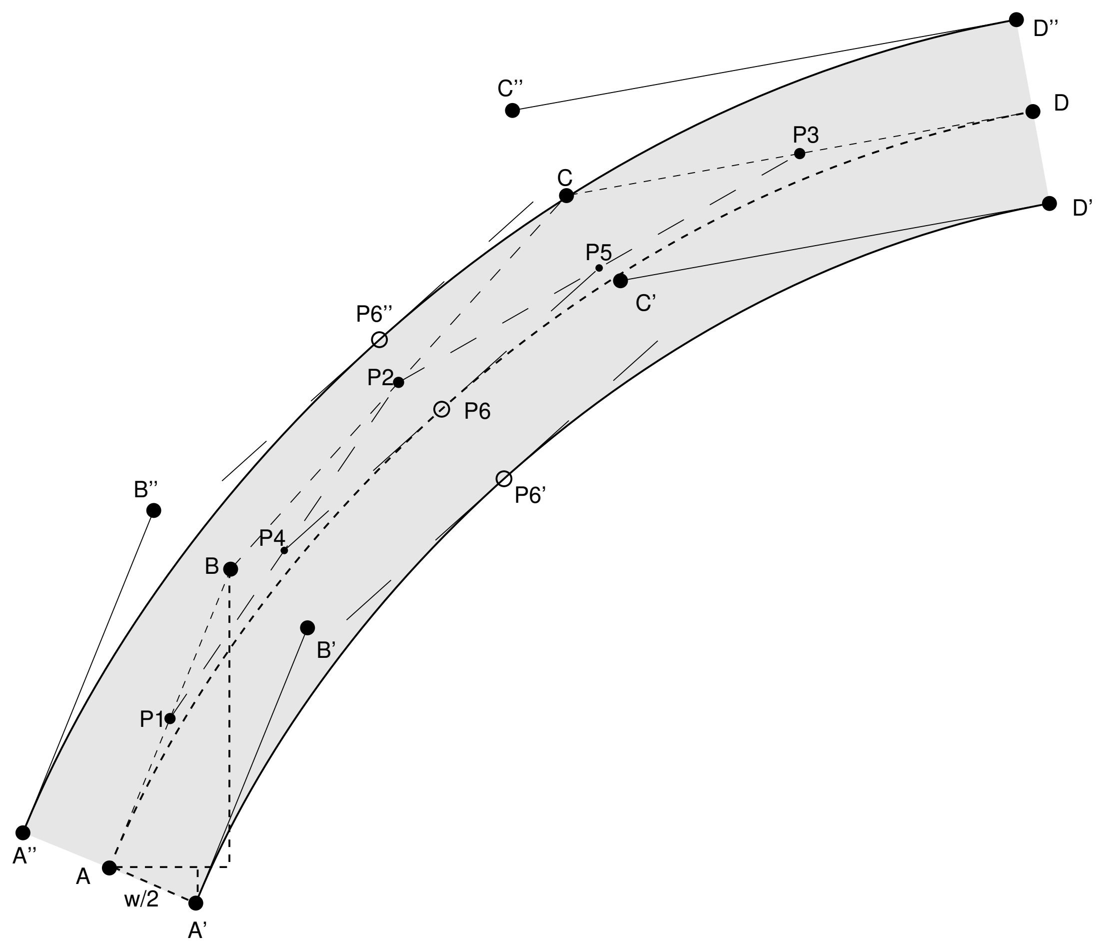

Figure 10: Construction of parallel Bezier path segments. The original curve is shown in dashed style and the light gray area indicates the thick Bezier curve segment that later will result from filling between left and right parallel path. Furthermore, important intermediate points are shown. A detailed discussion is given in the text.

The *parallel curve* resulting from the principle above actually no longer is a third order Bezier curve. But if a few additional constraints hold, it can be approximated quite well by such a third order Bezier curve:

- The curvature should not exceed an angular range of 90 degrees. This condition automatically is fulfilled for Type 1 fonts which adhere to the Adobe recommendations.
- The strokewidth w the curve should be drawn width is small compared to the extension and the curvature of the curve. This principle usually is fulfilled by nature because tracing a character outline path with a very thick pen won't lead to a good representation of the character.

In the following, we will describe how to compute a parallel Bezier curve defined by four points $\vec{A}'$, $\vec{B}'$, $\vec{C}'$ and $\vec{D}'$, given an original Bezier curve defined by four points $\vec{A}$, $\vec{B}$, $\vec{C}$ and $\vec{D}$ and a strokewidth w. The computation of parallel straight lines results as the special case of only respecting the points $\vec{A}$ and $\vec{D}$ from these considerations.

Figure 10 represents the basis of our discussion. It shows the original mathematically thin Bezier segment defined by the points $\vec{A}$, $\vec{B}$, $\vec{C}$ and $\vec{D}$ in dashed style. The counterpart of $\vec{A}$ in the parallel path follows from simple geometric considerations, as illustrated for the point $\vec{A}'$.

It lies half the strokewidth w away from $\vec{A}$ and the direction is determined by the location of point $\vec{B}$. For the two coordinates of $\vec{A}'$ we find

$$A'_x = A_x + \frac{w}{2}\frac{B_y - A_y}{|\vec{B} - \vec{A}|} \tag{1}$$

and

$$A'_y = A_x - \frac{w}{2}\frac{B_x - A_x}{|\vec{B} - \vec{A}|}. \tag{2}$$

Corresponding equations can be derived for the point $\vec{D}'$, so that, up to now, we are able to compute parallel straight line segments. It remains to compute two control points, $\vec{B}'$ and $\vec{C}'$, in a way that the resulting Bezier curve appears as parallel to the original curve in the sense defined above.

In order to make the path at point $\vec{A}'$ actually parallel to the orginal path at $\vec{A}$, we require $\vec{B}' - \vec{A}'$ to be parallel to $\vec{B} - \vec{A}$. From this we can derive an equation that expresses the fact that $\vec{B}'$ lies somewhere on the straight line that runs through point $\vec{A}'$ and has the direction $\vec{B} - \vec{A}$, i.e.,

$$\vec{B}' = \vec{A}' + \mu_B(\vec{B} - \vec{A}), \tag{3}$$

and correspondingly

$$\vec{C}' = \vec{D}' + \mu_C(\vec{C} - \vec{D}) \tag{4}$$

for point $\vec{C}'$. Here, μ_B and μ_C are two positive quantities, whose exact values are still to be determined.

In order to compute μ_B and μ_C, we consider a third point on the curve. Using a well-known algorithm that iteratively approximates a Bezier curve via straight line segments, we can easily determine the coordinates of the point that—in the parameter equation $f(t)$ of a Bezier curve—corresponds to the parameter $t = 1/2$. It can be considered as a *middle point* of the curve segment. In Figure 10, this point is named $\vec{P}_6$. It can be computed by computing some intermediate points:

$$\vec{P}_1 = \frac{1}{2}(\vec{A} + \vec{B}) \tag{5}$$

$$\vec{P}_2 = \frac{1}{2}(\vec{B} + \vec{C}) \tag{6}$$

$$\vec{P}_3 = \frac{1}{2}(\vec{C} + \vec{D}) \tag{7}$$

$$\vec{P}_4 = \frac{1}{2}(\vec{P}_1 + \vec{P}_2) \tag{8}$$

$$\vec{P}_5 = \frac{1}{2}(\vec{P}_2 + \vec{P}_3) \tag{9}$$

and finally

$$\vec{P}_6 = \frac{1}{2}(\vec{P}_4 + \vec{P}_5) = \frac{1}{8}(\vec{A} + 3\vec{B} + 3\vec{C} + \vec{D}) \tag{10}$$

Using the same geometrical considerations as in Eqs. 1 and 2, we can now compute a unit vector, $\vec{n}_6$, perpendicular to the curve at $\vec{P}_6$ and obtain

$$n_{6x} = \frac{P_{5y} - P_{4y}}{\sqrt{(P_{5x} - P_{4x})^2 + (P_{5y} - P_{4y})^2}} \tag{11}$$

$$n_{6y} = -\frac{P_{5x} - P_{4x}}{\sqrt{(P_{5x} - P_{4x})^2 + (P_{5y} - P_{4y})^2}} \tag{12}$$

$\vec{P}'_6$ can now be computed as

$$\vec{P}'_6 = \vec{P}_6 + \vec{N}_6, \tag{13}$$

where $\vec{N}_6 = \frac{w}{2}\vec{n}_6$, i.e., the vector orthogonal to the curve at $\vec{P}_6$ with a length of half the strokewidth w. As before, we have to require that the slope of the curve $\vec{P}_6$ equals the one at $\vec{P}'_6$, i.e., with respect to Figure 10 we find

$$\begin{aligned}
\vec{P}'_5 - \vec{P}'_4 &= \nu\left(\vec{P}_5 - \vec{P}_4\right) \\
\frac{\vec{P}'_2 + \vec{P}'_3}{2} - \frac{\vec{P}'_1 + \vec{P}'_2}{2} &= \nu\left(\frac{\vec{P}_2 + \vec{P}_3}{2} - \frac{\vec{P}_1 + \vec{P}_2}{2}\right) \\
\frac{\vec{C}' + \vec{D}'}{2} - \frac{\vec{A}' + \vec{B}'}{2} &= \nu\left(\frac{\vec{C} + \vec{D}}{2} - \frac{\vec{A} + \vec{B}}{2}\right)
\end{aligned}$$

and hence finally

$$\vec{C}' + \vec{D}' - \vec{A}' - \vec{B}' = \nu\left(\vec{C} + \vec{D} - \vec{A} - \vec{B}\right) . \tag{14}$$

We have thus expressed the slope condition at $\vec{P}_6$ in terms of the characteristic points of a Bezier curve and a factor, ν, still to be determined (cf. Eqs. 3 and 4). On the way to Eq. 14, we made use of the well-known geometrical relations Eqs. 5 – 9.

Based on the same considerations that led to Eq. 10, we can write the corresponding equation for the point $\vec{P}'_6$:

$$\vec{P}'_6 = \frac{1}{2}(\vec{P}'_4 + \vec{P}'_5) = \frac{1}{8}(\vec{A}' + 3\vec{B}' + 3\vec{C}' + \vec{D}') \tag{15}$$

Exploiting Eq. 13 and solving for $\vec{C}'$, we can reorganize Eq. 15:

$$\vec{C}' = \frac{8(\vec{N}_6 + \vec{P}_6) - \vec{A}' - \vec{D}'}{3} - \vec{B}' \tag{16}$$

From this equation, we are able eliminate $\vec{B}'$ by substituting the transformed slope condition for point $\vec{P}'_6$ (Eq. 14). We obtain

$$\begin{aligned}
\vec{C}' &= \frac{8(\vec{N}_6 + \vec{P}_6) - \vec{A}' - \vec{D}'}{3} + \left[\nu\left(\vec{C} + \vec{D} - \vec{A} - \vec{B}\right) - \vec{C}' - \vec{D}' + \vec{A}'\right] \\
2\,\vec{C}' &= \frac{8(\vec{N}_6 + \vec{P}_6) - \vec{A}' - \vec{D}'}{3} + \vec{A}' - \vec{D}' + \nu\left(\vec{C} + \vec{D} - \vec{A} - \vec{B}\right) \\
\vec{C}' &= \underbrace{\frac{4(\vec{N}_6 + \vec{P}_6) + \vec{A}' - 2\vec{D}'}{3}}_{\vec{l}_C} + \frac{\nu}{2}\underbrace{\left(\vec{C} + \vec{D} - \vec{A} - \vec{B}\right)}_{\vec{d}_C} .
\end{aligned} \tag{17}$$

Here, for the sake of brevity, we introduced a location vector, $\vec{l}_C$, and a direction vector, $\vec{d}_C$, which together with the parameter ν define the point $\vec{C}'$.

Considering Eqs. 4 and 17, we finally found two independent relations for $\vec{C}'$, that linearly depend on two quantities, μ_C and $\frac{\nu}{2}$. Therefore, by substituting the right hand sides of (4) and (17), we obtain the following 2×2 system of linear equations:

$$\left[\ (\vec{C} - \vec{D}) \quad \vec{d}_C\ \right] \begin{pmatrix} \mu_C \\ \nu/2 \end{pmatrix} = \left(\vec{l}_C - \vec{D}'\right) \tag{18}$$

Formally, all vectors appearing in this equation are column vectors. The solution of the system can be written as

$$\begin{pmatrix} \mu_C \\ \nu/2 \end{pmatrix} = \left[\ (\vec{C} - \vec{D}) \quad \vec{d}_C\ \right]^{-1} \left(\vec{l}_C - \vec{D}'\right) . \tag{19}$$

Once $\vec{C}'$ has been computed, it is easy to compute $\vec{B}'$, by making use of Eq. 16.

A few remarks about the approach described above are appropriate.

- It is also possible to first compute the point $\vec{B}'$ and then use Eq. 16 to compute $\vec{C}'$.
- The numerical stability at the respective end of the curve determines the preference of which point to compute first. A criterion for the numerical stability is the absolute value of determinant of the 2×2 matrix in Eq. 18.
- This determinant may become zero in which case the curve transforms into a straight line. These cases must be treated extraordinarily.
- There are a number of further exceptional cases, e.g., if point $\vec{C}$ equals $\vec{D}$. Then the slope at this end of the curve is not enforced by point $\vec{C}$.
- A good solution, that is, a *parallel curve*, will only result, if the set of assumptions discussed previously holds. If the resulting curve does not appear *parallel* to the original curve, the parallel curve cannot be approximated by a third order Bezier spline.

8.3 Connection of Path Segments and Prolongation

In order to actually obtain delimiting paths for character outlines, the parallel paths have to be connected to a continuous path. This raises the problem of line joining. When connecting two neighboring parallel path segments, we have to distinguish between two qualitatively different situations.

1. Convex Corner
   When tracing along two neighboring parallel path segments, we turn to the left and a convex corner appears. In these cases, we prolongate the end of the first path and the beginning of the second path using straight lines and compute an intersection between these prolongation segments. The resulting lengths of both prolongation segments will be positive.

2. Concave Corner
   When tracing along two neighboring parallel path segments, we turn to the right and a concave corner results. In these cases, the two neighboring parallel path segments intersect by nature and actually would have to be trimmed to their intersection point. Trimming on the other hand would make it impossible to feed the resulting curve in the standard format into the rasterizer. We therefore use a trick that saves us computing an intersection and recomputing the Bezier control points. From the ideal end point of the first parallel path we insert a straight prolongation to the connection point of the original path segments and a second straight prolongation segment from there to the starting point of the second parallel path segment. Then, the area left of the path is ensured to be within the extents that finally are to be filled with ink.

We will now explain this principle using the example shown in Figure 11. The interesting part is in the middle right. The original path—shown in bold dashed style—steps into the figure in the lower right as the end of a curve segment p_1. At the following connection point, the path strongly turns to the right so that a concave corner results and continues with a further curve segment p_2. This path is now to be surrounded in a symmetrical manner by one right and one left path. For p_1, we find the right path as a parallel curve segment above the original path. It has been computed as described in the previous section. For p_2, the right path is a parallel curve segment located in an appropriate distance below p_2. The two neighboring right path segments

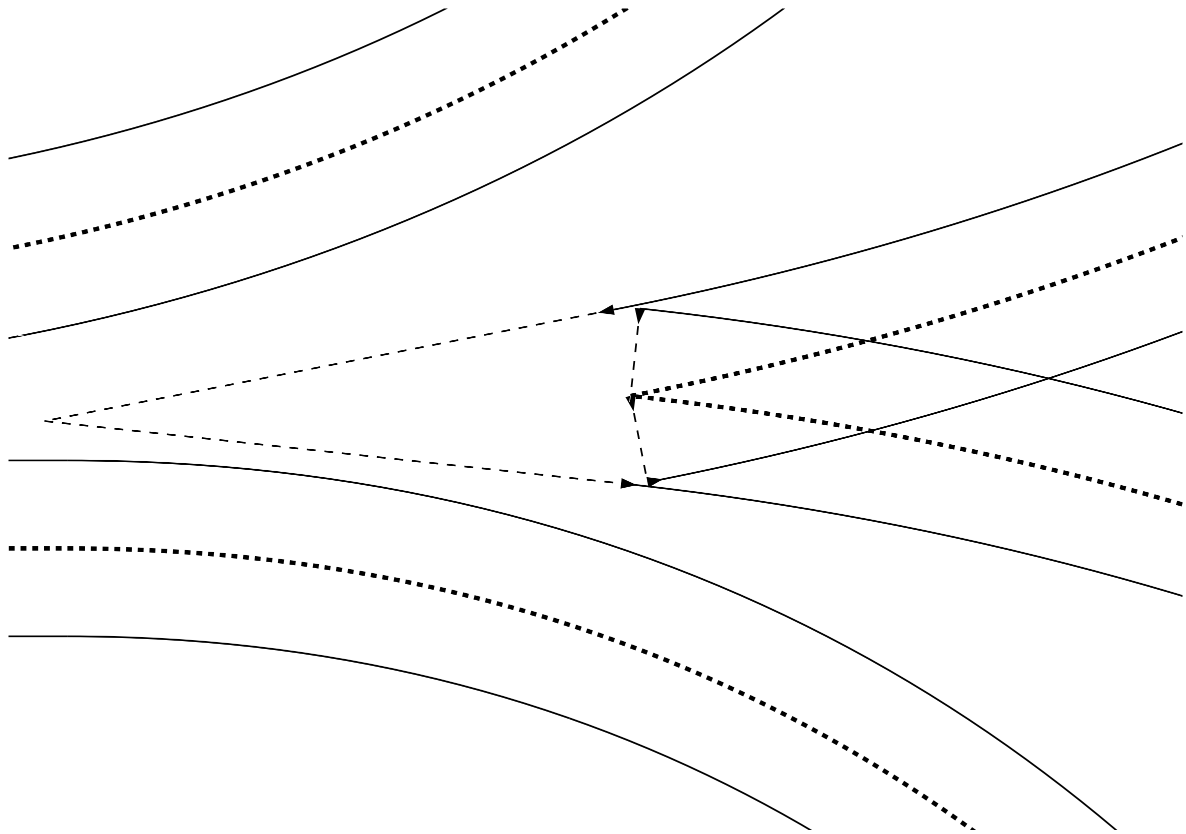

Figure 11: A small excerpt at the middle right from the character "B" of ComputerModern Roman. The ideal mathematical outline of the filled character is shown in thick dashed style. Left and right path of the character's outline representation are shown in medium solid style. Prolongation is indicated by large dashes of medium thickness.

are disjoint because of the concavity of the resulting corner. Hence, rule 2 from above applies in order to connect them using straight prolongation lines, in the figure shown in wide dashes of a medium linewidth: From the end of right path 1, we prolongate to the point where the original segments p_1 and p_2 join, and from there, a second prolongation to the beginning of right path 2 is inserted. The direction is indicated by arrows. Obviously, even the right path alone produces a closed region in this case, but this does not cause problems here.

The left path runs into the direction opposite to the original path. By nature, the curvature at the point under consideration now is convex. Hence, according to rule 1, the neighboring left paths' segments are prolongated to their common intersection point, respecting the ending direction of left path 1 and the starting direction of left path 2.

The kind of corner at two neighboring parallel path segments p_1 and p_2 can be computed analytically. Let $\vec{T}_1$ be the tangent vector at the end point of p_1 and $\vec{T}_2$ be the tangent vector at the starting point of p_2. Assuming that both $\vec{T}_1$ and $\vec{T}_2$ are column vectors, we can use the determinant of the square matrix constructed by these vectors to determine the corner type:

$$d = \begin{vmatrix} \vec{T}_1 & \vec{T}_2 \end{vmatrix} \tag{20}$$

If $d < 0$, the corner type is concave whereas for $d > 0$, the corner type is convex. For the special case $d = 0$, the slope at the joining point is continuous, so that effectively $\vec{T}_1$ and $\vec{T}_2$ linearly depend on each other. For those cases, prolongation is not required at all, because if the neighboring segments in the original path join, neighboring segments in the left and right path will do so too.

The kind of joining lines described above is known as *mitered line joining*. `t1lib` does not impose a limit on the width of mitered corners, so that the operation `t1lib` implements is identical to what PostScript does by default, i.e., using a line join type of 0 and an infinite miter limit.

Function Index