Overview of the Generic Text Interface

Adobe Developer Support

Technical Note #5118

20 May 1992

Adobe Systems Incorporated

Adobe Developer Technologies
345 Park Avenue
San Jose, CA 95110
http://partners.adobe.com/

PN LPS5118

Copyright © 1990–1992 by Adobe Systems Incorporated. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of the publisher. Any software referred to herein is furnished under license and may only be used or copied in accordance with the terms of such license.

PostScript, the PostScript logo, Adobe, and the Adobe logo are trademarks of Adobe Systems Incorporated which may be registered in certain jurisdictions. IBM PC is a registered trademark of International Business Machines Corporation. Microsoft and MS-DOS are registered trademarks of Microsoft Corporation. Palatino is a trademark of Linotype-Hell AG and/or its subsidiaries. Other brand or product names are the trademarks or registered trademarks of their respective holders.

This publication and the information herein is furnished AS IS, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies, makes no warranty of any kind (express, implied or statutory) with respect to this publication, and expressly disclaims any and all warranties of merchantability, fitness for particular purposes and noninfringement of third party rights.

Contents

Overview of the Generic Text Interface 5

1 Introduction 5

2 Overview 5
- The Data Structures 8
- Naming Conventions and Using the Code 9

3 High Level Features 10
- Showing Text 10
- Underlining, Double Underlining, Strike-Through 12
- Superscript and Subscript 12
- Font Switching 13
- Font Downloading 14
- Pair Kerning 16
- Extended Character Sets 17
- Tabs and Columns 18

4 The Example Program and Other Programs 19
- The AFM Map Building Program 19

Appendix A: Generic Text Interface Functions Summary 21

A.1 Driver Functions 21
- hdshoany.c 21
- ldconven.c 22
- ldfiles.c 23
- ldfonts.c 24
- ldglobal.c 27
- ldmisc.c 27
- ldrdmap.c 27
- ldtext.c 28
- ldunpack.c 28

A.2 Example Program 29
- lotest.c 29

A.3 AFM Map Building Program 29
- makemap.c 29
- readmap.c 30

Appendix B: Changes Since Earlier Versions 33

Index 35

Overview of the Generic Text Interface

1 Introduction

This technical note is designed to accompany a body of source code called the generic text interface; it is available in the PostScript™ Language Software Development Kit. While not a full-blown PostScript printer driver, the generic text interface does handle the difficult problems associated with writing the text handling portion of a PostScript printer driver. Most notably, it handles dynamic font downloading and the associated memory management problems. The generic text interface can be used as a basis for an efficient PostScript printer driver that supports more functionality than many existing drivers.

Also included with the source code for the driver itself is a sample program called lotest, which illustrates how to call the driver to generate efficient PostScript language output. In addition, there are several pieces of source code that can be used to help support text output on a PostScript device, such as aparser for (Adobe™ font metrics (AFM) files.

The driver is compatible with both PostScript Level 1 and Level 2 printers. Generic text interface is oriented toward IBM PC® compatible machines, but the code is fairly portable and can be used on other platforms with a minimum of effort.

2 Overview

The generic text interface is a line-oriented interface; it is optimized for printing one line of text at a time. It supports most of the features needed by common word-processing or page layout programs, and provides facilities for future expansion. The interface allows the application to generate PostScript language code that supports the following features:

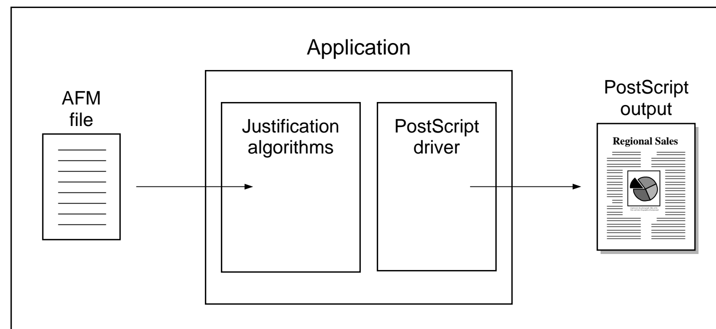
- Showing normal text, with justification or track kerning
- Underline, double underline, strike-through, superscript, and subscript
- Access to an unlimited number of fonts

- Access to any point size (including non-integer) of a font
- Dynamic font downloading
- Pair kerning or other inter-character spacing adjustments
- Font dictionary caching (high-performance font switching)
- Extended character set handling (IBM PC character set, in this example)
- Tabs and columns

The entire text handling interface can be accessed through a single data structure called a *showstruct* and a single procedure call **ShowAny( )**.

The generic text interface expects the layout of the text to take place before the printing machinery is engaged. This has several implications relating to the calculations that take place in the application, and those that take place in the driver.

Figure 1 *Generic Text Interface*



The application does justification based on character width data from the AFM files. The generic text interface works according to this model; justification, kerning, and other spacing decisions take place in the application.

Many applications written today use advanced graphics on screen displays, font support, and other typographical control that was not available in computers until recently. Because many applications are WYSIWYG, the driver is written assuming that calculations about line breaking, justification, and so forth (the basic layout of the document), have been done by the application. The generic text interface merely uses the description of a line to generate PostScript language code that corresponds to that description.

Because applications,WYSIWYG or not, require character width information to make line-breaking decisions when using different fonts, the source code for an AFM file parser is included on the disk. The AFM files contain this character width information, and also contain other font-related information, such as pair kerning data.

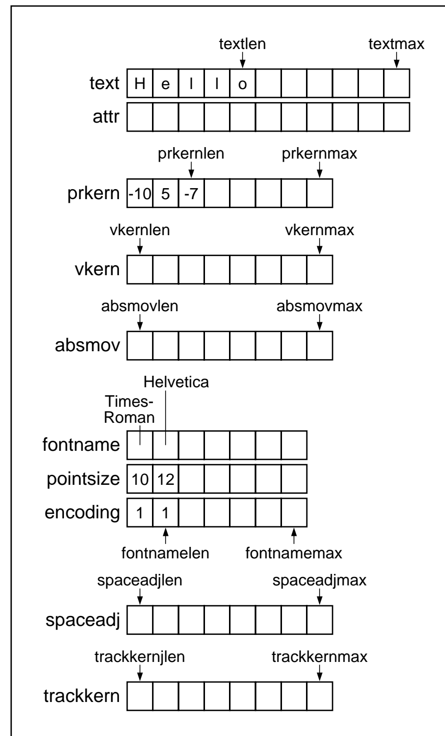
The AFM parser code can be used either directly, as part of your application, or as a translator, translating from AFM format to your application's width information format. If your application uses a proprietary file format to store character width information, it is strongly recommended that you also include a translator to translate from the AFM file format to your proprietary file format. This is useful to the end user because AFM files are shipped with each Adobe typeface package.

Because there is no algorithm to convert font names to MS-DOS® file names (to find the AFM file, downloadable font, or screen font), a program is provided to build a table that maps font names to file names or PostScript language font names. This code is described in section 4.1. This problem is peculiar to MS-DOS because of the eight character limitation on file name prefixes.

2.1 The Data Structures

The highest level data structure is the *showstruct* (in *hdhilvl.h*), which contains the following fields:

Figure 2 *showstruct—a graphical representation*



text array contains the text to be shown

attr array contains “attributes” or “events” that occur at a character position on the line

prkern array amount of horizontal kerning between two characters

vkern array amount of vertical kerning or adjustment between two characters

absmov array coordinates of absolute **moveto** commands

fontname array contains pointers to font names for controlling font changes

pointsize array each entry corresponds to an entry in *fontname* array

encoding array each entry indicates the encoding vector to use for each *fontname* entry

spaceadj array used for full justification; amount to adjust the space character

trackkern array the amount of space to add to each character on a line

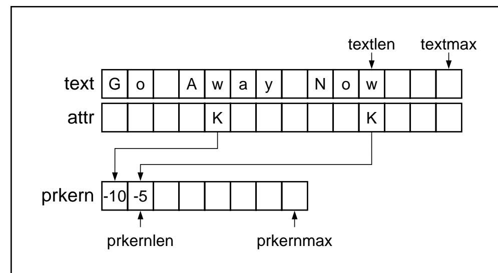
Each typographical feature of the generic text interface is supported by the concept of attributes, where an attribute is a mask (defined in *hdhilvl.h*) that can be added to the current character. Each character in the *text* array corresponds to an entry in the *attr* array entry that represents which attributes are turned on or off at that character. This is usually accomplished with something similar to the following code:

```
show.attr[charposition] |= SA_UNDERLINE_ON;
```

The |= assignment allows the underline attribute to be enabled without disabling any of the other attributes that might have already been enabled at this position, since each of the attributes is defined as a bit mask. This allows more than one attribute to occur at a particular location.

For example, to underline the “Hello” in the string “Hello World,” add the mask *SA_UNDERLINE_ON* to the *attr* array at the position of the ‘H,’ and the mask *SA_UNDERLINE_OFF* at the position following the “o” in “Hello.” If no attributes are associated with a character, the *attr* array will contain the value *SA_NOATTR* at that position.

Figure 3 *Adding pair kern attributes to showstruct*



For attributes that require parameters (a pair kern), an entry in the array or arrays will correspond to that attribute. For example, a pair kern attribute will have an entry in the pair kern array that is the value of the kern (the adjustable distance between characters). A font change will have entries in three arrays: *fontname*, *pointsize*, and *encoding*.

2.2 Naming Conventions and Using the Code

The code supplied with this technical note is divided into three sections: the example program, the high-level interface, and the low-level driver. Files and variable names that “belong” to the high-level and low-level driver are prefixed with **hd** and **ld** respectively. Other variable or file names are considered part of the example program.

The driver refers to a file called *prolog.ps*, which contains the PostScript language procedure definitions called by the C language code.

Supplied with the code is a make file for the Microsoft® C compiler version 6.0 for MS-DOS. Though every attempt has been made to make the code portable, it necessarily contains some compiler dependencies—the most obvious being the way binary files are handled.

Before compiling and running the example program, edit the file called *ldglobal.c*. This file contains information the program uses to find the AFM and PFB (printer font binary, or downloadable font outlines) directories. Modify the variables *afmdirectory*, *afmmapfile*, and *fontdirectory* to point to the applicable directories or files on your hard disk. In a proper application, these variables should be available at the user interface level, possibly as an installation option.

Included with the generic text interface package is a program called *lotest.c*, which demonstrates the use of *showstruct* and the **ShowAny( )** function. For *lotest* and the generic text interface to download fonts, the file *afmmap.txt* must exist in the directory set up for the AFM files. See section 4.1 for more information on the AFM map file and how to create it.

3 High Level Features

The following is an overview of each of the features of the text driver and how to access them from the high-level data structure.

3.1 Showing Text

Showing text can, when optimized, have a big impact on the performance of a PostScript printer driver. Specifically, several different operators showing text are built into the PostScript language. Using the correct operator can greatly improve the speed of the driver, especially when showing justified or track-kerned text. The function **hd_showstr( )** makes this optimization.

The **hd_showstr( )** function uses information provided in the *showstruct* to make its decisions. Specifically, **hd_showstr( )** is passed the current values of the space adjustment (justification) and track kerning arrays. If these arrays have no entries, the values passed are zeros. For non-zero entries in the track kern and space adjustment arrays, the operator chosen to show text corresponds to the following table:

Table 1 *Comparison of text showing operators*

Space Adjustment	*Track Kern*	*Operator*
0	0	**show**
Non-Zero	0	**widthshow**
0	Non-Zero	**ashow**
Non-Zero	Non-Zero	**awidthshow**

The alternative to using the special-case **show** operators is to emulate their functionality by using a series of **show** and **rmoveto** operators. Using **rmoveto** between each **show** string is significantly slower than letting the PostScript interpreter do this work, because the special-case operators are optimized for these calculations. Additionally, using a combination of **rmoveto** and **show** increases data transmission and interpreter overhead. By passing appropriate space adjustment and track kerning values in the show structure, driver speed can be greatly increased because **ShowAny( )** uses the PostScript language operator that is optimized for the job.

Another way to optimize the speed of showing text is to group as many characters together as possible in a single **show** string. Because each call to the **show** operator incurs interpreter overhead, it is more efficient to show the string (Hello world) than to show two separate strings (Hello) and (world).

The driver is primarily dependent on the application to fill the *showstruct* properly for this optimization. Each time an attribute is found, a **show** string is generated. The implication of this is that the driver will generate more efficient PostScript language code if the application does not use redundant attributes in the *showstruct*. For instance, if the application generates two font change attributes that are the same font name and point size, the driver will not optimize this into a single font change, but will generate two font changes, and two or three (depending on the locations of the font changes) **show** strings. Other decisions about string breaking related to kerning are covered in section 3.5.

A third way to optimize the speed of showing text is to show characters in the most efficient format. Usually, this means putting the character into a show string and sending the string to the interpreter. However, for certain encoding schemes, characters that appear in the show strings can conflict with the communications interfaces. The most prevalent example is in a parallel or serial communication environment, where some control characters are reserved for flow control messages to the communications hardware. In cases like this, it is important to send the characters using the octal (\xxx) notation, permissible in strings, to avoid terminating the job or interrupting flow control.

The function **SendPSString( )**, in the file *ldfiles.c*, checks the value of the *TransMode* field of the *1d_DriverControl* structure (in the file *ldglobal.c*) to see whether or not the octal conversion must occur and for which characters. Modes supported include:

- 7-bit ASCII – All characters below 32 and above 127 are converted to octal
- 8-bit ASCII – All characters below 32 are converted
- Binary – No conversion
- Adobe binary communication protocol – No conversion

In addition, the (,), and \ characters are always converted to octal to prevent possible problems in interpretation.

3.2 Underlining, Double Underlining, Strike-Through

Performing underlining in a PostScript printer driver involves drawing a line in the appropriate place. The position and thickness used for the lines drawn by underlining and strike-through are numbers that are hard-coded in to the driver (in *ldtext.c*). For a more accurate line, these can be overridden by using the *UnderlinePosition* and *UnderlineThickness* keys present in the AFM files.

To cause underlining, double underlining, or strike-through, add an *SA_UNDRLN_ON*, *SA_DBLUND_ON*, or *SA_STRIKE_ON* attribute to the *attr* array in the *showstruct* at the location where you want to begin underlining. Similarly, underlining or strike-through is disabled by using the *SA_UNDRLN_OFF*, *SA_DBLUND_OFF*, or *SA_STRIKE_OFF* in the appropriate position in the attr array.

It is important to note that, because **ShowAny( )** works on a line-by-line basis, the application must explicitly turn off underlining at the end of the text line, even if the underline will continue on the next line, and then re-enable it at the beginning of the next line.

3.3 Superscript and Subscript

Typographically, superscript and subscript characters should be shown with special purpose characters designed to look consistent with rest of the characters in the typeface family. The Adobe Expert character sets were designed for this type of high-quality typography, but these special characters are not available for all fonts. If an Expert character set or a character set specifically designed for super/subscript is not available, it is possible to simulate the characters. This is usually accomplished by non-uniformly scaling the superior and inferior characters to make them appear similar in stroke width to the rest of the text line.

In the generic text interface, superscript and subscript use non-uniform scaling (.65 in x, and .60 in y) of the current font to create a font that has a slightly heavier stroke width—one that is more readable and visually appealing. For the application to calculate line breaking correctly, it must understand that the superscript or subscript characters are .65 times the width of the current font.

Superscript and subscript are enabled by adding the *SA_SUPER_ON* or *SA_SUB_ON* attribute to the *attr* array at the position where the superscript or subscript should begin, and disabled by adding an *SA_SUPER_OFF* or *SA_SUB_OFF* attribute where the superscript or subscript should end.

As with underlining, the superscript and subscript must be explicitly terminated at the end of a *showstruct* line, and re-enabled at the beginning of the next line.

3.4 Font Switching

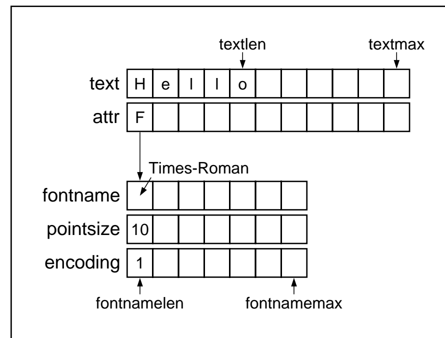
The font switching machinery that the generic text interface incorporates is perhaps the most extensive piece of the driver. The font switching machinery

- Provides access to an unlimited number of fonts and point sizes
- Optimizes font switching by using font dictionary caching
- Handles dynamic font downloading and memory management
- Handles extended character sets (described in section 3.6)
- Generates document structuring conventions (DSC) compatible code

The font switching machinery is handled by adding a font change attribute to the *attr* array of the show structure. For each font change attribute, there should be an accompanying entry in the *fontname*, *pointsize*, and *encoding* arrays in the show structure. The contents of *fontname* is currently defined as the PostScript language font name, but it can be any form of the font name if the lookup code in the AFM map table is used to match the name. See section 4.1 for more information on the map table code.

To increase the speed of the output, the driver supports a technique called font dictionary caching. See the Technical Note #5048, “Font Switching Optimizations,” available from the Adobe Developers Association, for more information on speeding up font switching.

Figure 4 *Adding a font change to the showstruct*



The field *fontnamelen* indicates the number of font changes that have been added to the structure.

The routine **ChooseFont( )** is designed so the application has access to unlimited numbers of downloadable fonts and to unlimited (including non-integer) point sizes. Part of supporting an unlimited number of fonts in a PostScript printer driver means supporting dynamic font downloading.

3.5 Font Downloading

At the most basic level, dynamic font downloading (that is, font downloading in the middle of the PostScript language job stream) is fairly simple. It involves unpacking a font from the binary format in which it is stored on the host and sending it to the printer. If all the fonts you want to download will fit in the printer's memory, then font downloading is easy. It becomes more complicated when dealing with limited memory situations.

The save and restore Operators

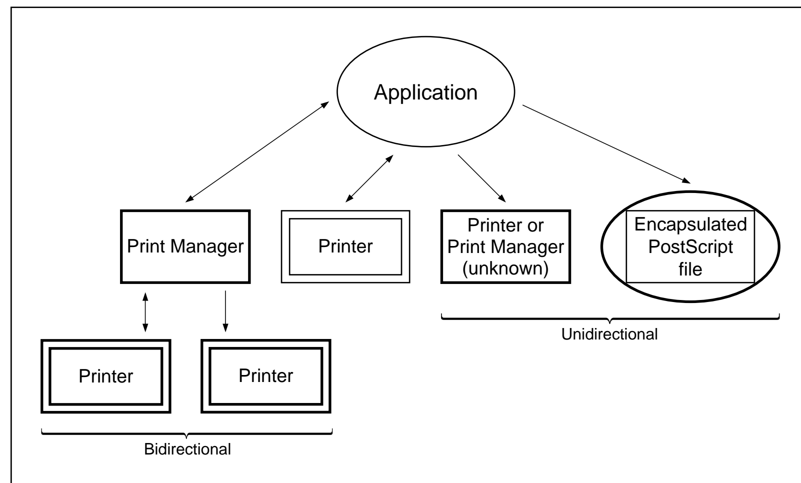
Memory management in the PostScript language is accomplished by using the **save** and **restore** operators. Before a font is downloaded, **save** is used to mark the state of memory. The application notes the size of the font being downloaded, so when a new font must be downloaded, the application can determine how many fonts it must "throw away" before the new font will fit in the printer's memory. Fonts are removed from memory, by using the **restore** operator. When a **restore** is issued, the interpreter is reset to its state at the time of the **save**.

Any changes to, for instance, the graphics state will be undone when the **restore** occurs. These changes are typically handled by saving the relevant graphics state parameters on the stack (which is unaffected by **save** and **restore**), before the **restore** and resetting them after the **save**, or by maintaining the state of relevant parameters on the host side.

Note *In PostScript Level 2, there are additional operators for memory management that can make swapping fonts easier, such as* ***undefinefont****. Because the generic text interface is designed to produce PostScript language code suitable for both Level 1 and Level 2 printers, using these operators will be covered in a subsequent technical note.*

Given the use of **save** and **restore** for memory management, there are two models for deciding when a font should be swapped or downloaded: bidirectional communication and unidirectional communication. A serial port is an example of bidirectional communication; a parallel port is unidirectional communication. We will first discuss the bidirectional case, then the unidirectional case, which is what the generic text interface uses. In both methods, the PostScript printer description (PPD) files play an important role.

Figure 5 *Font downloading cases*



The amount of information available about the destination printer might affect your font downloading strategy. In cases where there is little information, a more conservative strategy will be most reliable. In direct-connect cases with bidirectional communication, the driver can optimize more and (possibly) forego page independence.

When the printer driver has a bidirectional communication link with the printer, it can execute queries to the printer in the middle of the job. When a font is needed, the driver asks the printer if the font is available, and how much memory is currently available.

Additionally, a bidirectional driver can forego page independence (prescribed by the document structuring conventions (see Appendix G of the *PostScript Language Reference Manual, Second Edition*), and thereby potentially reduce the number of times a font needs to be downloaded. Page independence is useful if the print job might be re-directed to an intelligent print manager, but is not as important if the file is going directly to a printer.

In the unidirectional case, there are more considerations. For this discussion, supporting the DSC is considered part of the unidirectional case. Although a query job is permitted by the DSC for printers that have bidirectional communication links to the host, this only tells the driver the initial state of the interpreter; no queries are allowed in the middle of the job stream. The driver must still maintain lists of available fonts and current (estimated) available memory.

Because the font downloading algorithm used for the generic text interface does not have enough data to make optimized decisions about the downloading order, it downloads fonts as they appear in the text. If your application knows in advance which fonts are required in the document or if you have a two-pass driver design, you can take advantage of this information to optimize font downloading order. For instance, if you know which fonts you will use frequently, download them in the %%DocumentSetup section of the document, so the fonts will not have to be downloaded on each page to maintain page independence.

To avoid unnecessary font downloading, the data structures used by the generic text interface should be initialized with the fonts and memory available on the printer when the job starts. These can be determined either by querying the printer, as per the DSC, or by reading the PPD file. When the list of printer fonts is determined, it should be added to the list of downloaded fonts (*ld_PrinterFonts* in *ldglobal.c*), because this is the list that **ld_FontOnPrinter( )** looks at to determine which fonts must be downloaded. The base state of memory should be set in *ld_PrinterStat[0].VM*.

3.6 Pair Kerning

Pair kerning is supported through the driver; however, it is the application's responsibility to generate the proper data concerning kern pairs. If the application supports a WYSIWYG display, this should not present a new problem, because the data must be available for screen display, anyway. Typically, applications that support PostScript language fonts determine character width and pair kerning by reading the AFM files, or some compiled version of this width information. There are also alternative methods for mathematically calculating pair kerns using individual character bounding box information, also available in the AFM files. "Proper" kerning amounts is a very subjective art.

Pair kerning is implemented as an attribute; to cause a pair kern, use the |= operator to add an *SA_PAIRKERN* attribute to the *attr* array of *showstruct*. For each pair kern event, there should be a corresponding entry in the *PrKern* array of the *showstruct* that indicates the amount of the kern. This amount is specified relative to the current coordinate system, and is converted to an

rmoveto in the x direction. (This is typically the most efficient method of supporting kerning. The code to generate the pair kern (**rmoveto**) is in the **ShowAny( )** function, and calls the prolog procedure *kh* (kern horizontal).

Note that for each pair kern event, a separate **show** string is generated. Your application should avoid adding kern events with the value of zero, because this will cause the driver to breakup the string into many smaller strings, increase the data transmitted, and decrease the efficiency of the driver. See the discussion of efficient text showing in section 3.1.

3.7 Extended Character Sets

Supporting a character set such as the one used by the IBM PC and compatibles presents the application developer with several different problems:

- The extended characters (line drawing, smiley face, and so forth) are only available in a few PostScript language fonts.
- A limitation of the composite character font machinery in early versions of the PostScript interpreter reduces the number of available positions in the encoding vector.

Composite characters are character definitions composed of more than one glyph. They are typically the accented characters, such as iacute (í) or aring (å) and are actually composed of two separate glyphs positioned together. For instance, the iacute character is composed of the dotlessi and acute glyphs.

In early versions of the PostScript interpreter, all the glyphs that comprise a composite character and the composite glyph name itself are required in the encoding vector of the font. That is, for the iacute character to print correctly, the iacute name, the dotlessi name, and the acute name all must be in the font's encoding vector. Failure to encode the glyphs properly results in an **invalidfont** error but only at the time the composite character is being shown. This makes the problem particularly difficult to diagnose unless you are aware of the symptoms.

The IBM PC character set is problematic because there are no "free" positions. That is, there are no positions in the encoding vector available for use by the accent characters. The generic text interface handles both the composite problem and the multiple font problem at one time by implementing a font switching technique for characters in certain character sets. Two encoding vectors are defined; one for the special characters, and one for the normal text characters. When a special character is encountered in the *showstruct*, **ShowAny( )** switches to a font containing the special characters,

and encoded with the special characters encoding vector. After the special characters have been shown, **ShowAny( )** (encoded with the text vector) switches back to the text font .

Because the two encoding vectors exactly overlap, the union forms a complete character set. This also opens up some .notdef spaces in each encoding vector that are never actually used, so there is now room to define characters that must exist in the encoding vector to avoid running in to the composite character limitation on older interpreters.

To take advantage of the extended character set mechanism, use the encoding value *SA_EXTENDED* in the encoding array for the font being set. To avoid re-encoding of a particular font, use the *SA_DEFAULT* value for the encoding array entry. Because the font switching technique uses cached scaled font dictionaries, the performance of this scheme is very good.

3.8 Tabs and Columns

The calculations for tabs and columns must be made in the application; however, the generic text interface provides an *SA_ABSMOV* attribute or "absolute **moveto**." Again, the application must bear the brunt of the work in terms of where the text is actually to be set. The driver merely converts this to an efficient PostScript language representation. For each *SA_ABSMOV* attribute in the *attr* array, there should be two entries in the *absmov* array in the *showstruct*, because the **moveto** is being specified in both the x and y axes. The first entry is the x position to move to, the second entry is the y position.

Note that it is important to have an absolute **moveto** attribute at the beginning of each line of text shown, otherwise, the text will be placed at the end of the last line. Also note that the *showstruct* can be used to show several lines of text by placing absolute **moveto**s in the proper place in the data structure. However, to minimize memory usage, it makes more sense to send a line at a time to **ShowAny( )**, since the efficiency of the resulting PostScript language code will be the same in either case.

4 The Example Program and Other Programs

The example application (lotest) illustrates how to call the generic text interface through **ShowAny( )** to generate efficient PostScript language code, achieving a number of different text effects.

4.1 The AFM Map Building Program

Because there are limitations on the length of file names in the MS-DOS environment, there is no algorithm that maps file name prefixes for the font and associated files to the PostScript language font name (or the full name of the font).

The problem of naming font files is worse than it might appear. Instead of eight characters for the font name, there are actually only five; the latter three characters are reserved to indicate the point size of screen font files. For files that are not screen fonts (that is, AFM or PFB files), and for cases where there are not a full five characters in the abbreviated file name prefix, the last characters are replaced by underbar (_) characters.

When given the font's full name or the PostScript language font name, the most reasonable solution to the problem of finding file names is to create a lookup table that contains the names of the files, font names, and full names. To create such a table, use the program provided on the source disk called *makemap.exe*.

makemap allows you to specify a directory in which the AFM files reside. It is assumed that the map file will be present in the directory that contains the AFM files.

After the map file is created, it can be read by the low level part of the generic text interface that handles font downloading. In addition, there is a file called *readmap.exe*, which is a utility that can be used to look up a file name, full name, or PostScript language font name, when given any of the others. Both readmap and makemap provide usage notes simply by running the programs. Source code for both programs is also provided.

Appendix A: Generic Text Interface Functions Summary

The following is a list of functions by file name. The description and use of each function follows its definition. Functions marked with *[stub]* are prototyped, but contain no code or contain dummy code. These are functions that are left to the person who implements the driver.

A.1 Driver Functions

The following sections detail the procedures that are implemented in each of the files that compose the printer driver for the generic text interface.

A.1.1 hdshoany.c

ShowAny

```
void ShowAny(struct showstruct *sho)
```

ShowAny( ) is the top level of the text interface. When passed a properly filled *showstruct*, **ShowAny( )** generates one line of text, handling all of the font downloading, underlining, superscript, and so on, that occur on that line.

hd_showstr

```
void hd_showstr(int start, int end, struct showstruct *sho, float spaceval,
    float trackval)
```

This function is called by **ShowAny( )** to generate the PostScript language code to show the string of text between the *start* and *end* positions from the *showstruct*. The **hd_showstr( )** determines the most efficient text-showing operator to use, based on whether or not justification and track kerning are taking place

A.1.2 ldconven.c

The generic text interface supports version 3.0 of the document structuring conventions (DSC) comments. The routines defined in *ldconven.c* are the core of this support.

BeginDocument

void BeginDocument(char *title, char *createdfor, int dlprolog)

BeginDocument( ) should be called at the beginning of a PostScript language job to generate the DSC header and optionally copy the prolog to the printer. **BeginDocument( )** generates many DSC comments that help document managers, such as %%LanguageLevel and %%DocumentData, in addition to the standard header and setup section comments.

EndDocument

void EndDocument(void)

EndDocument( ) generates the DSC comments required at the end of the document and prints the list of fonts used during the job as a %%DocumentSuppliedResources: or %%DocumentNeededResources: comment.

BeginPage

void BeginPage(void)

BeginPage( ) generates needed DSC comments at the beginning of a page, including page number and page setup. Also calls the BP procedure in the prolog to execute the page-level **save**.

EndPage

void EndPage(void)

EndPage( ) generates needed DSC comments at the end of the page, and calls the EP procedure to generate page-level **restore**.

ld_AddDocFont

void ld_AddDocFont(char *fontname)

ld_AddDocFont( ) checks to see whether this font has been used, and if not, adds the font name to the list maintained for the %%DocumentResources: font comments at the end of the document.

ld_PrintDocFonts void ld_PrintDocFonts(void)

ld_PrintDocFonts( ) prints the list of fonts used in the document. Called by **EndPage( )**.

A.1.3 ldfiles.c

fatalerr void fatalerr(char *str)

fatalerr( ) is used to print an error message and terminate program execution.

openfile FILE *openfile(char *name, char *mode)

openfile( ) attempts to open a file for reading or writing. If the file cannot be opened, **fatalerr( )** is called.

SendString void SendString(char *str)

SendString( ) is the heart of the low-level communication with the printer. **SendString( )** sends a NULL-terminated string of text to the printer. This is where low-level communications protocols can be implemented. The following Send-functions call **SendString( )**.

SendFile void SendFile(FILE *file)

SendFile( ) sends a file to the printer. Used to send the prolog file in **BeginDocument( )**.

SendChar void SendChar(char c)

SendChar( ) ends a single character to the printer by sending a single character string to **SendString( )**.

SendPSString void SendPSString(char *str, int len)

SendPSString( ) converts a PostScript language string to a format that is compatible with the current communication channel, as defined by the *ld_control* variables in *ldglobal.c*. This can involve converting some characters in the string to octal notation if they are incompatible with the current communication protocol. Additionally, **SendPSString( )** supports the Adobe binary communications protocol to quote control characters being sent to the PostScript interpreter.

SendFloat void SendFloat(float num)

SendFloat( ) is used to communicate a floating-point number to the interpreter. If the number has no fractional part, the number is sent without the decimal point to minimize data transmission.

SendInt void SendInt(int num)

SendInt( ) is used to send integer numbers to the printer.

OpenPrinter int OpenPrinter(void)

OpenPrinter( ) opens the printer or output file for writing. Called by **BeginDocument( )**.

ClosePrinter int ClosePrinter(void)

ClosePrinter( ) closes the printer or output file. Called by **EndDocument( )**.

A.1.4 ldfonts.c

ChooseFont void ld_ChooseFont(char *fontname, float pointsize, char encoding)

ChooseFont( ) is the highest level of the font handling machinery. **ChooseFont( )** checks to see if a font is available in the printer. If so, it calls **ld_SetFont( )**. If not, it checks to see if the font is available on the host. If so, it downloads the font and sets it. Otherwise, **ChooseFont( )** calls **ld_SubstituteFont( )** [Stub] to create a substitute font. **ChooseFont( )** is called by **ShowAny( )**.

ld_AddPointsize int ld_AddPointsize(struct ld_pfont *pfont, float pointsize, int *index)

Called by **ld_AddPrinterFont( )**. (See next entry.)

ld_AddPrinterFont int ld_AddPrinterFont(char *fontname, float pointsize, int *index)

To support font dictionary caching, a list of font names that exists in the printer and whether or not they have been re-encoded must be maintained on the host. **ld_AddPrinterFont( )** searches for the font name in this list. If the name does not exist, it adds the name to the list (*ld_printerFonts*). Afterwards, it calls **ld_AddPointsize( )** to add the size of the font to the point size list. **ld_AddPrinterFont( )** returns the index of the font used as the name of the cached scaled font dictionary. This is called by **ld_Setfont( )**.

ld_SetFont void ld_SetFont(char *fontname, float pointsize, char encoding)

ld_SetFont( ) calls **ld_AddPrinterFont( )** to find out whether or not this font has been set. **ld_SetFont( )** generates the PostScript language code to re-encode a font (if necessary) and set it.

ld_FontOnPrinter int ld_FontOnPrinter(char *fontname)

ld_FontOnPrinter( ) searches a linked list of fontnames that have been downloaded (or were resident) on the printer. If the *fontname* is found, it returns *TRUE*; otherwise, it returns *FALSE*.

ld_FontOnHost int ld_FontOnHost(char *fontname, char *fontpath)

ld_FontOnHost( ) attempts to determine whether or not a compressed font file (PFB format) is available on the host computer. If file is available, returns the full path name in *fontpath*. Space is allocated for *fontpath* within *ld_FontOnHost*, and freed at the end of *ld_ChooseFont*. Called by **ChooseFont( )**.

ld_VMUsage long ld_VMUsage(char *fontpath)

ld_VMUsage( ) opens the font file and searches for the %%VMUsage: comment to see how much memory the font will use when downloaded. If the %%VMUsage: is found, **ld_VMUsage** returns this number; otherwise it returns a default value.

ld_DownloadFont void ld_DownloadFont(char *fontpath)

ld_DownloadFont( ) unpacks a printer font binary (PFB) format font and sends it to the printer. Called by **ld_ChooseFont( )**.

ld_SubstituteFont [Stub] void ld_SubstituteFont(char *realfont, char *subfont)

ld_SubstituteFont( ) should get the metrics for *realfont* and create a new font, based on *subfont* that contains a *Metrics* dictionary with the character metrics of *realfont*.

ld_SubWhichFont [Stub] char *ld_SubWhichFont(char *fontname)

ld_SubWhichFont( ) is where the decision should be made about which font to substitute for a font that is not available. Called by **ld_ChooseFont( )**.

ld_FreePointsizes void ld_FreePointsizes(struct ld_psize *ptr)

ld_FreePointsizes( ) is called by **ld_FreePrinterFonts( )**. (See next entry.)

ld_FreePrinterFonts void ld_FreePrinterFonts(struct ld_pfont *ptr)

ld_FreePrinterFonts( ) is called by **ld_Restore( )** to remove the list of cached printer fonts and point sizes associated with a particular save level.

ld_Restore void ld_Restore(int level)

ld_Restore( ) restores memory back to *level* to allow another font to be downloaded to memory. Calls **ld_FreePrinterFonts( )** to eliminate C data structures for the list of cached font dictionaries.

ld_Save void ld_Save(void)

ld_Save( ) generates a save level in the printer and in the C language code's data structures. This save is generally used for memory management, so fonts can be thrown away when a new font is needed but is too big to fit in memory.

A.1.5 ldglobal.c

ld_InitStructs void ld_InitStructs(void)

ld_InitStructs( ) initializes the global variables associated with the low-level driver. This should be called before any of the driver routines are used. **ld_InitStructs( )** also contains the defaults for the general behavior of the text driver. The variables defined here should be available at the user interface level in a real product.

A.1.6 ldmisc.c

***d_malloc** void *d_malloc(size_t size)

***d_malloc( )** calls **malloc( )** to allocate memory. If no memory is available, calls **fatalerr( )**.

***d_calloc** void *d_calloc(size_t numelems, size_t size)

***d_calloc( )** calls **calloc( )** to allocate memory. If no memory is available, calls **fatalerr( )**.

d_free void d_free(void *ptr)

d_free( ) calls **free( )** to free memory previously allocated with **d_calloc( )** or **d_malloc( )**. If there is an error, calls **fatalerr( )**.

A.1.7 ldrdmap.c

getline int getline(FILE *fp, char *line)

getline( ) reads a line from a text file into a previously allocated buffer (*line*). The buffer must be big enough to hold all characters to the end of the line.

readAFMmap int readAFMmap(char *fontname, char *fullname, char *filename, FILE *mapfile)

readAFMmap( ) reads a map file that has been built with the BUILDMAP program. (See next entry.) Given knowledge of one of the variables, **readAFMmap( )** is used to look up the PostScript language font name, the full font name, or the file name for a particular font. For instance, to look up the file name of *Palatino* Italic*, the full font name (Palatino Italic) can be passed to *fullname*. Alternatively, the PostScript language font name (PalatinoItalic) can be passed to *fontname*. If no names matching the parameters can be found **readAFMmap( )** returns with all strings filled, or an error code. This routine is generally used by the driver to find the file name of a font file.

A.1.8 ldtext.c

BeginAttribute int BeginAttribute(int mask)

BeginAttribute( ) is used to generate the PostScript language code to mark the beginning of underlining, double underlining, or strike-through, or a combination of the three. Called by **ShowAny( )**.

EndAttribute EndAttribute(int mask)

EndAttribute( ) is used to generate PostScript language code to draw a line from the current point to a point that was marked with a **BeginAttribute( )** call. Called by **ShowAny( )**.

A.1.9 ldunpack.c

GetByte unsigned char GetByte(FILE *fp, long *cnt)

GetByte( ) reads a byte from the specified file and decrements the file's length counter (*cnt*).

unpackPFB int unpackPFB(FILE *fontfile)

unpackPFB( ) takes a pointer to a compressed Printer Font Binary (PFB) format font file and un-compresses it, sending the data to the printer with **SendChar( )**. This is the heart of the font downloading routines.

A.2 Example Program

A.2.1 lotest.c

AllocShowStruct

AllocShowStruct(struct showstruct *sho)

AllocShowStruct( ) allocates a blank show structure. This routine can be used by the calling application to allocate a new structure of predetermined size. Alternatively, the structure can be allocated on-the-fly by the application that will be making calls to **ShowAny( )**.

AddChar

int AddChar(struct showstruct *sho, char c)

AddChar( ) adds a character to the show structure and increments the *textlen* pointer.

FontChng

int FontChng(struct showstruct *sho, char *fontname, float pointsize,
    int encoding)

FontChng( ) adds a font change attribute to the *attr* array in the show structure, adds the font name, point size, and encoding entries to their appropriate arrays in the show structure.

ResetShow

ResetShow(struct showstruct *sho)

ResetShow( ) resets a show structure's *text* and *attr* arrays and resets length pointers to 0, so the show structure can be reused.

A.3 AFM Map Building Program

The following procedures are used to construct an AFM file map table.

A.3.1 makemap.c

GetAFMList

GetAFMList(struct namelist *list)

GetAFMList( ) searches a directory, opening all the files with the extension ".AFM," and building a linked list of these file names. Called by **buildmap( )**.

fatalerr fatalerr(char *str)

fatalerr( ) prints an error message and terminates program execution.

getline getline(fp, buf)

getline( ) reads a line of text from a file and copies it into *buf*. Calls **fatalerr( )** if *buf* is not large enough to hold the string.

GetNames GetNames(FILE *fp, char *fontname, char *fullname)

GetNames( ) reads a line from an AFM file and looks for the *FullName* and FontName keys. If found, they are returned in the *fontname* and *fullname* variables, otherwise **GetNames( )** returns -1.

openfile FILE *openfile(char *name, char *mode)

openfile( ) opens a file for reading or writing by calling **fopen( )**. If the file can't be opened, calls **fatalerr( )**.

buildmap buildmap(int drive, char *mapdir, FILE *mapfile)

buildmap( ) looks through a directory full of AFM files and creates a file that contains a map of font names, full names, and file names.

A.3.2 readmap.c

getline int getline(FILE *fp, char *line)

getline( ) reads a line of text from a file and copies it into **buf**. Calls **fatalerr( )** if *buf* is not large enough to hold the string.

readAFMmap int readAFMmap(char *fontname, char *fullname, char *filename,
FILE *mapfile)

readAFMmap( ) reads a map file that has been built with the makemap program. Given knowledge of one of the variables, **readAFMmap( )** is used to look up the PostScript language font name, the full font name, or the file name for a particular font. For instance, to look up the file name of Palatino Italic, the full font name (Palatino Italic) can be passed to *fullname*. Alternatively, the PostScript language font name (PalatinoItalic) can be passed to *fontname*. If no names matching the parameters can be found, **readAFMmap( )** returns with all strings filled, or an error code.

Appendix B: Changes Since Earlier Versions

Changes since March 31, 1992

- Minor typographical errors were corrected in Appendix A.

Changes since August 9, 1991 version

- Document was reformatted in the new document layout and minor editorial changes were made.
- The diagrams have been reworked.

Changes since May 3, 1991 version

- Changed the way font names are stored; updated graphics and text.
- Fixed minor typos.
- Removed **ld_QueryVM( )**.
- Changed **ld_FontOnHost( )** to return full pathname of font if found.
- Implemented **ld_VMUsage( )**; now looks in font file for %%VMUsage comment.
- Updated **ld_DownloadFont( )** to use font path name instead of looking up font name.

Index

A

absmov 18
Adobe Font Metrics. *See* AFM
AFM 7
afmmap.txt 10
AllocShowStruct( ) 29
attr array 13

B

BeginAttribute( ) 28
BeginDocument( ) 22
BeginPage( ) 22
buildmap( ) 30

C, D

calloc( ) 27
character set
  Expert 12
ChooseFont( ) 14, 24
ClosePrinter( ) 24

E

EndAttribute( ) 28
EndDocument( ) 22
EndPage( ) 22

F

fatalerr( ) 23
files
  afmmap.txt 10
  hdhilvl.h 8
  hdshoany.c 21
  ldconven.c 22
  ldfiles.c 23–24
  ldfonts.c 24–26
  ldglobal.c 10, 27
  ldmisc.c 27
  ldrdmap.c 27
  ldtext.c 28
  ldunpack.c 28
  lotest.c 10, 29
  makemap.c 29
  makemap.exe 19
  prolog.ps 9
  readmap.c 30
  readmap.exe 19
FontChng( ) 29
fontnamelen 14
free( ) 27

G

generic text interface 5–19
  AFM
    map building program 29
  data structures 8–9
  driver functions 21–28
  example program 19
    AFM map building program 19
  functions summary 21–31
  high level features 10–18
    .notdef 18
    bidirectional communication 15
    columns 18
    composite character 17
    composite glyph 17
    double underlining 12
    encoding vector 18
    extended character set 17
    font dictionary caching 13
    font downloading 14–16
    font switching 13

memory management 14
pair kerning 16
restore 14–15
save 14–15
showing text 10
strike-through 12
subscript 12
superscript 12
tabs 18
underlining 12
unidirectional communication 15
naming convention 9
overview 5–10
GetAFMList( ) 29
GetByte( ) 28
getline( ) 27, 30
GetNames( ) 30

H

hd_showstr( ) 10, 21
hdhilvl.h 8
hdshoany.c 21

L

ld_AddDocFont( ) 22
ld_AddPointsize( ) 25
ld_AddPrinterFont( ) 25
ld_DownloadFont( ) 26
ld_FontOnHost( ) 25
ld_FontOnPrinter( ) 25
ld_FreePrinterFonts( ) 26
ld_InitStructs( ) 27
ld_PrintDocFonts( ) 23
ld_Restore( ) 26
ld_Save( ) 26
ld_SetFont( ) 25
ld_SubstituteFont( ) 26
ld_SubWhichFont( ) 26
ld_VMUsage( ) 25
ldconven.c 22
ldfiles.c 23–24
ldfonts.c 24–26
ldglobal.c 10, 27
ldmisc.c 27
ldrdmap.c 27
ldtext.c 28
ldunpack.c 28
lotest 5, 19
lotest.c 10, 29

M, N

makemap.c 29
makemap.exe 19
malloc( ) 27
moveto 18

O

octal (\xxx) notation 11
openfile( ) 23, 30
OpenPrinter( ) 24
operators
text showing 11

P, Q

PrKern 16
prolog.ps 9

R

readAFMmap( ) 28, 31
readmap.c 30
readmap.exe 19
ResetShow( ) 29
rmoveto 17

S, T

SA_ABSMOV 18
SA_DEFAULT 18
SA_EXTENDED 18
SendChar( ) 23
SendFile( ) 23
SendFloat( ) 24
SendInt( ) 24
SendPSString( ) 12, 24
SendString( ) 23
show
emulating 11
grouping characters 11
ShowAny( ) 6, 12, 17, 21
showstruct 6, 8–9, 11
adding a font change 14
pair kerning 9

U, V

unpackPFB( ) 28