Type 1 Font Format Supplement

Adobe Developer Support

Technical Specification #5015

15 May 1994

Adobe Systems Incorporated

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

PN LPS5015

Copyright © 1993, 1994 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.

Adobe, Acrobat, Adobe Originals, Adobe Type Manager, ATM, Minion, Myriad, PostScript, the PostScript logo, SuperATM and Viva are trademarks of Adobe Systems Incorporated which may be registered in certain jurisdictions. Macintosh and Personal LaserWriter are registered trademarks of Apple Computer Incorporated. Hewlett-Packard and LaserJet are registered trademarks of Hewlett-Packard Company. Windows is a trademark of Microsoft Corporation. All 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

List of Figures 5

1 Introduction 7

2 Counter Control Hints 7
- Performance and Quality Benefits of Counter Control Hinting 8
- **OtherSubrs** for Counter Control 9
- Stack Limit Considerations for Counter Control 11
- Counter Control Groups 12
- **Private** Dictionary Extensions for Counter Control: **ExpansionFactor** 12
- Counter Control Example 13

3 Multiple Master Font Extensions 14
- Multiple Master Design 15
- Multiple Master Font Programs 17
- Multiple Master Font Dictionaries 18
- Explanation of a Typical Multiple Master Font Program 19
- Multiple Master Keywords and Procedures 23
- The **makeblendedfont** Procedure 25
- The Multiple Master **findfont** Procedure 26
- The **NormalizeDesignVector** Procedure 27
- The **ConvertDesignVector** Procedure 27
- Multiple Master Font Names 28
- Multiple Master Charstring Representation 28
- **OtherSubrs** for Multiple Master Font Programs 29
- Sample **Subrs** Code for Calling **OtherSubrs Procedures 31**

4 Adobe Type Manager Compatibility 32

5 Font Program Testing 32

6 Errata 33

Appendix A: The makeblendedfont Operator 35

Appendix B: Updated OtherSubrs Code for Flex and Hint Substitution 39

Appendix C: NormalizeDesignVector Example 43

Appendix D: ConvertDesignVector Example 45

Appendix E: Changes Since Version 1.0 of the Type 1 Font Format Specification 47

Index 49

List of Figures

Figure 1 Sample glyph with Counter Control hint zones 13
Figure 2 Multiple master font space arrangement 17
Figure 3 Arrangement of multiple master design space for a four axis font 17
Figure 4 Multiple master font dictionaries 19

Type 1 Font Format Supplement

1 Introduction

This document describes extensions to the Adobe™ Type 1 font format since version 1.0, and contains a list of errata for both version 1.0 and 1.1 in section 6. It supersedes Adobe Technical Note #5047, "Updates to the Adobe Type 1 Font Format" and #5086, "Multiple Master Extensions to the Adobe Type 1 Font Format."

This supplement describes two significant extensions to the Type 1 format: *Counter Control*, a hinting mechanism for fonts with complex glyphs; and the multiple master font format, which was previously described in Technical Note #5086, "Multiple Master Extensions to the Adobe Type 1 Font Format."

The Counter Control hint mechanism is used for controlling the counters (the white spaces between stems) in complex glyphs such as those contained in Chinese and Japanese language fonts. These hints may also have other applications such as for bar code or logo fonts.

A multiple master font contains from 2 to 16 *master designs* in a single font, from which users may interpolate a large number of intermediate *font instances*. This format, discussed in section 3, provides the potential for unprecedented flexibility and control over typographic parameters.

In addition, the following appendices are included:

- Appendix A: **makeblendedfont** Code
- Appendix B: Updated **OtherSubrs** Code for Flex and Hint Substitution
- Appendix C: **NormalizeDesignVector** Example
- Appendix D: **ConvertDesignVector** Example
- Appendix E: Changes Since Version 1.0

2 Counter Control Hints

The Counter Control hint mechanism controls counter spaces in a glyph. A *counter* may be defined as an area of white space which is delimited by a pair of horizontal or vertical stems. This mechanism is designed to aid in the

rendering of fonts containing complex glyph shapes by ensuring that the size and proportions of all counters in a glyph are rendered as accurately as possible. For example, if multiple counters are exactly the same measurement in width or height, the Counter Control mechanism will make them the same number of pixels, providing there are a sufficient number of pixels available. Similarly, if the width of two counters in the original design are, for example, in the ratio of 3:5, the interpreter attempts to preserve this proportion, based on the constraints of the glyph's width.

Counters may be organized into *groups*, with each group consisting of a section of the glyph whose stems are to be considered in relation to each other by the rasterizer. For a relatively simple glyph, for example, all horizontal stems may be considered to be in a single group. For more complex glyphs, putting all stems in a single group might overconstrain the grid fitting problem. Also, the ordering of the groups determines the priority for the allocation of pixels, which may be critical for lower resolutions. The grouping of counters is discussed in section 2.4.

To use Counter Control hints, the **LanguageGroup** and **RndStemUp** entries (see page 44 of the Adobe Type 1 Format Book for more details) must be defined as follows in the **Private** dictionary of the font program:

```
/LanguageGroup 1 def
/RndStemUp false def
```

and Counter Control hints, in the form of calls to **OtherSubrs** entries 12 and 13, must be added to the appropriate charstrings as explained in section 2.2.

2.1 Performance and Quality Benefits of Counter Control Hinting

For fonts with complex glyphs, it is very important to include Counter Control hints; failure to do so can result in performance and quality degradation. Some rasterizer implementations are able to control counters by making an initial pass through the font to compile hint data, and a second pass to rasterize the glyphs. Thus the rasterizer supplies some of the Counter Control hints, but at the cost of reduced performance. The ATM rasterizer included in some Level 1 Japanese PostScript printers, most Level 2 printers, and ATM™-J software fall into this category.

Note *There are two types of "two-pass" rasterizers: newer versions of the ATM-J software and Level 2 printers will do only one pass if Counter Control hints are in the font, thus improving performance, but are capable of doing two passes if the hints are not in the font; earlier versions will do two passes even if Counter Control hints are in a font (ignoring the data in the font).*

Other rasterizers do not compile Counter Control data on-the-fly. With this rasterizer, a font is likely to have unsatisfactory quality unless the Counter Control hints are pre-compiled in the font program. The current version of the Type 1 Coprocessor is in this category of one-pass rasterizers.

The advantage of including Counter Control hint information in a font program is that the font will perform better on most two-pass rasterizers, and it is the only way to control counter spaces with a one-pass rasterizer. Also, if Counter Control hinting is pre-compiled into a font program, it is possible to define more precise hints than if it is done at run-time by the rasterizer.

If a font does not contain complex glyphs, it is important for performance reasons to not use hint settings which will cause a two-pass rasterizer to compile Counter Control hints. Any of the following situations will cause a two-pass interpreter to make an extra pass, whether or not it is required:

- The top edge of the first **BlueValues** hint zone is represented by a negative number. This signifies that the first (baseline) zone is set to be outside of the area of the character paths (this is one convention for representing fonts which do not require vertical alignment zones).

- The keyword **RndStemUp** is defined in the font program. The value of the Boolean does not make a difference: if **RndStemUp** is defined at all, Counter Control is invoked regardless of the value.

- **LanguageGroup** is defined to have a value of 1.

- The charstrings contain calls to **OtherSubrs** entries 12 or 13.

2.2 OtherSubrs for Counter Control

Counter Control hints are specified using the **callothersubr** charstring operator and **OtherSubrs** number 12 and 13. These calls must immediately follow the **hsbw** or **sbw** operator, and must only occur once in the charstring procedure. All other hints and hint substitution is done in the usual manner.

The **callothersubr** operators for Counter Control hinting will be interpreted directly by newer Type 1 BuildChar procedures, but will be ignored by 2-pass rasterizers which will compile their own Counter Control data at run-time.

For a rasterizer which does not know about Counter Control hints, the PostScript language implementation of the Counter Control **OtherSubrs** only serves the purpose of removing the Counter Control data from the stack so the data will not accumulate. These procedures (which are shown below) do not implement Counter Control hints, they merely make a font backward-compatible on older interpreters.

As in the *Adobe Type 1 Font Format* book, the *stack bottom symbol* (⊢) preceding the first argument means that the arguments are taken from the bottom of the Type 1 BuildChar stack. Commands that clear the stack are indicated by the stack bottom symbol (⊢) in the result position of the command definition.

The following **OtherSubrs** calls are used to invoke Counter Control hinting:

Counter Control OtherSubrs Entry 12

⊢ A_1 A_2 A_3 ... A_n *n* 12 **callothersubr** ⊢

where A_1 through A_n are arguments for declaring counter data, *n* is the number of arguments, and '12' is the **OtherSubrs** entry number. The value of *n* must be in the range $0 \le n \le 22$ (see section 2.3 for explanation of the stack limit). **OtherSubrs** entry 12 is used to present Counter Control data to the Type 1 BuildChar. It may be used to pass any number of arguments within the stated limits, but it should be used efficiently as excess calls may significantly affect file size and performance. A call to entry 12 implies that there is more data to follow. A sequence of one or more calls to entry 12 must be followed by exactly one call to entry 13. Usually, arguments will be presented in groups of 22 until there are 22 arguments or fewer, and the remaining arguments are then passed using **OtherSubrs** entry 13.

The data format is the same as for **OtherSubrs** entry 13, which is shown below.

Counter Control OtherSubrs Entry 13

⊢ A_1 A_2 A_3 ... A_n *n* 13 **callothersubr** ⊢

where A_1 through A_n are arguments for declaring counter data, *n* specifies the number of arguments, and '13' is the **OtherSubrs** entry number. The value of *n* must be in the range $0 \le n \le 22$. **OtherSubrs** entry 13 tells the interpreter that all of the Counter Control data is on the stack and ready for processing. It must be called exactly once for each glyph, and only after a sequence of zero or more calls to **OtherSubrs** entry 12. The data format for both **OtherSubrs** entries 12 and 13 follows.

Data Format for Counter Control OtherSubrs

The data format is:

⊢ *#H* HG_1 HG_2 ... HG_n *#V* VG_1 VG_2 ... VG_n *m* 13 **callothersubr** ⊢

where #H is the number of stem groups (0 if none). HG_1 is the data for the most important hstem group (Group 1), and HG_n is the data for the least important hstem group (Group *n*). #V is the number of vstem groups (0 if none); VG_1 is the data for the most important vstem group; VG_n is the data for the least important group; and *m* is the number of arguments being passed.

The PostScript language code contained in **OtherSubrs** entries 12 and 13 is shown below. These two PostScript language procedures are not used for backward compatibility, except in the sense that they remove data from the stack when the interpreter does not understand Counter Control hints, or when the interpreter can only do two passes. When an interpreter is capable of only doing one pass, the data in the charstring **OtherSubrs** calls are interpreted directly by the interpreter.

The code for the two **OtherSubrs** entries are as follows:

OtherSubrs entry 12:
{}

OtherSubrs entry 13:
{ 2 { cvi { { pop 0 lt { exit } if } loop } repeat } repeat }

Entry 12 does not clear data from the stack because any calls to entry 12 must be followed by one call to entry 13, which does clear all elements from the stack.

Note *Some PostScript interpreters fail to execute the PostScript language* ***OtherSubrs*** *procedures. This is incorrect behaviour. This primarily affects the Apple Personal LaserWriter® NT and the Hewlett-Packard® Level 1 PostScript Cartridge for LaserJet® printers. The likely consequence is that an* invalidfont *error will occur, the fonts will not appear on the page, or the interpreter will fail.*

2.3 Stack Limit Considerations for Counter Control

Since there may be more than 22 numbers required to define all group data, and no more than 22 numbers may be on the type 1 stack when an **OtherSubrs** procedure is invoked, the call to **OtherSubrs** entry 12 is used to present arguments in groups of 22, until 22 or fewer arguments remain. This final group is then passed using **OtherSubrs** entry 13.

In all cases, the arguments are placed on an internal pseudo stack by popping arguments off the Type 1 stack and pushing them onto this internal pseudo stack, and then finally processed by popping them off the internal pseudo stack. This inverted order is a consequence of the requirement that it be possible to execute a Type 1 font program with a Type 1 BuildChar procedure that does not have direct support for **OtherSubrs**.

2.4 Counter Control Groups

A *Group* is a list of coordinates for counters, delimited by stems of a single orientation (either horizontal or vertical), that are to be adjusted relative to each other. They are listed in ascending order (in character space) as pairs of numbers. Each pair consists of a value for the left (or bottom) edge and the width of the stroke, where the left (or bottom) edge is encoded as the distance from the previous stem, or the distance from zero (for the horizontal direction, the distance from the left sidebearing point) for the first entry.

Since the count of the number of stems in a group is not given, the sequence is encoded by making the distance to the final stem extend to the far edge of the stem, and then making the final width negative. For example, if the last pair of numbers would ordinarily be '…100 20…', it must be encoded as '…120 –20…'. Not all stems in a glyph need be included; if one or more counter spaces are judged to not need control, they may be omitted from the Counter Control hinting.

The order of each group in the calling sequence determines the priority for that group. This priority (for a group, not for individual counters) determines the order in which the groups are allocated white space pixels. The order of the groups can be determined algorithmically, or by a designer.

This prioritizing scheme gives the designer the ability to specify the groups whose counters will be the last to "collapse" (have no white pixels) at small sizes, and which will be more accurately rendered at intermediate sizes. This has the potential to significantly improve quality and legibility at a range of sizes. Even if the group order is not manually or algorithmically determined, it is still much better to have Counter Control hinting than to not have it at all.

2.5 Private Dictionary Extensions for Counter Control: ExpansionFactor

The optional **ExpansionFactor** entry is a positive real number that gives a limit for changing the size of a character bounding box during the processing that adjusts the sizes of counters in fonts of **LanguageGroup** 1. The default value of **ExpansionFactor** is 0.06, which is equivalent to allowing a ± 6% change in the character bounding box. This change is allowed in both the x- and y-directions, but might be constrained in the y-direction depending on vertical alignment values specified in the **BlueValues** array.

At small point sizes or low resolutions, the system might have to accept irregular counters rather than violate this limit. Bar code or logo fonts containing glyphs with multiple counters might benefit by setting **LanguageGroup** to 1 and increasing the **ExpansionFactor** limit to a larger amount such as 0.5 or more. For example:

```
/ExpansionFactor 0.5 def
```

If strict adherence to the metrics is essential, the value should be set to zero.

2.6 Counter Control Example

The following is a simplified example of how Counter Control hints may be applied to a glyph. Figure 1 shows a Kanji glyph and the coordinates of the stems and counter boundaries. In this example, only the hinting most relevant to Counter Control is shown; miscellaneous hints and hint substitution are not addressed. Also, the ordering of the groups is *top-to-bottom* and *left-to-right*, rather than being based on typographic significance.

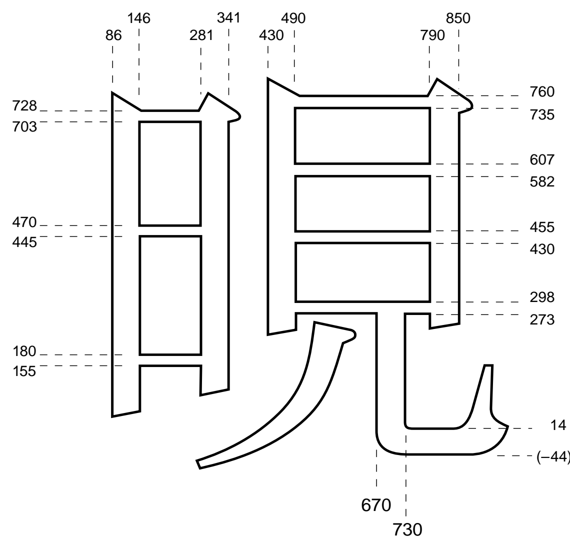


Figure 1 *Sample glyph with Counter Control hint zones*

The horizontal stem hints for this example would be:

–44 58 hstem	273 25 hstem
155 25 hstem	430 25 hstem
445 25 hstem	582 25 hstem
703 25 hstem	735 25 hstem

and the vertical stem hints would be:

86 60 vstem
281 60 vstem
430 60 vstem
670 60 vstem
790 60 vstem

Counter Control hinting involves dividing the counters which are delimited, for example, by hstems, into groups which are to be considered at one time by the rasterizer. In Figure 1, the hstems on the left side of the glyph form a logical group, and those on the right side form a second logical group (stems in a logical group do not need to be part of the same subpath).

Based on the stem data shown above and the chosen division of groups, the corresponding data to describe the counters for each group would be (excluding the horizontal stem at –44 and the vertical stem at 670):

```
2 155 25 265 25 258 -25 273 25 132 25 127 25 153 -25 1 86 60 135
   60 89 60 360 -60
```

where #H = 2 (the number of hstem groups); HG1 = (155 25 265 25 258 –25); HG2 = (273 25 132 25 127 25 153 –25); #V = 1 (the number of vstem groups); and VG1 = (86 60 135 60 89 60 360 –60). In this example, the '–60' argument would end up on the bottom of the stack, and the '2' argument on the top of the stack.

The hstem from –44 to 14, and the vstem from 670 to 730, are not included in the Counter Control data. The hstem at –44 does form a counter space with the hstem above it, but it is omitted in this example. Reasons for omitting a particular counter might include the judgement that its proportions are not as critical as those of other counters, or that including it might overconstrain the problem.

Because using the above data as arguments to a **callothersubr** call would put more than 22 items (in addition to the **OtherSubrs** entry number and the number of arguments), on the stack, the call must be divided into two calls. Allowing for the necessary stack order noted above, the calls would be as follows:

```
25 265 25 258 -25 273 25 132 25 127 25 153 -25 1 86 60 135 60 89
60 360 -60 22 12 callothersubr

2 155 2 13 callothersubr
```

This command sequence is now ready for encoding.

3 Multiple Master Font Extensions

The multiple master font format is an extension of the Type 1 font format which allows the generation of a wide variety of typeface styles from a single font program.

A multiple master font program contains two or more outline typefaces called *master designs*, which describe one or more *design axes*. The master designs that constitute a design axis represent a dynamic range of one typographic

parameter, such as the *weight* or *width*. This range of styles is defined in a multiple master font program by specifying one master design to represent each end of an axis, such as a *light* and *extra-bold* weight, as well as any *intermediate master designs* that are required. The maximum number of master designs allowed is sixteen.

Note *Intermediate designs are not supported in the current version of Adobe Type Manager™ software: version 3.6.1 for the Macintosh, and 2.6 for Windows.*

A *font instance* consists of a font dictionary derived from the multiple master font program (or from another font instance). It contains a **WeightVector** array having *k* values (that sum to 1.0) which specify the relative contribution of each master design to the resulting interpolated design.

All derived font instances share the **CharStrings** dictionary and **Subrs** array of the main multiple master font program, making it relatively economical to generate a variety of font instances. Multiple master fonts can be made compatible with the installed base of PostScript language interpreters by including several PostScript language procedures and a set of **OtherSubrs** routines in the font program. The procedures include the interpolation procedure **$Blend**, the **makeblendedfont** operator emulation procedure (see Appendix A), and a re-definition of the **findfont** operator (see section 3.7). The multiple master related **OtherSubrs** procedures are used, along with the **$Blend** procedure, to interpolate the charstring data on-the-fly to produce the interpolated glyph shapes specified in the font instance.

3.1 Multiple Master Design

It is possible to think of the master designs as being arranged in a 1, 2, 3, or 4 dimensional space with various font instances corresponding to different locations in that space. The entries in the **FontInfo** dictionary specify what this space is and where in that space the master designs are located. This information is necessary for interactive programs that allow users to create new font instances, and should be included in the font's Adobe Multiple Font Metrics (AMFM) file (see "Adobe Font Metrics File Format Specification," Version 4.0).

Multiple master coordinates are of two types: *design coordinates*, which represent the design space, and *blend coordinates*, which represent the blend space.

Design coordinates are integers whose range for a particular typeface is chosen by the designer. They are used in font names and in the user interface for software which creates and manipulates multiple master font programs. The standard minimum and maximum values for a weight or width axis is

from 1 to 999 design units; however a typical typeface, with styles ranging from light to black, might only have a dynamic range of from 200 (for light) to 800 units (for black).

Note *In the case of the Adobe Originals™ typeface Viva™, the range of design coordinates has been extended to range from 1 to 2000. This is purely an extension for a type design with a much wider width than most conventional designs. Extending the width axis does not change the coordinates for designs in the standard range.*

Another type of axis is *optical size*, in which the character design changes with the point size to optimize legibility for each point size. The design coordinates for the optical size axis might have a dynamic range of from 6- to 72-point, which represents the practical extremes of sizes for typefaces designed for publishing purposes.

Blend coordinates are normalized values, in the range of 0 to 1, which correspond to the minimum and maximum design space coordinates for a specific font. They are used by the Type 1 rasterizer because they are more convenient for mathematical manipulations.

The mapping between the design and the blend coordinate space may be specified to be non-linear by using the **BlendDesignMap** entry (discussed below) in the font dictionary. While the non-linear mapping may be used for any axis, it is especially useful for the OpticalSize axis.

Figure 2 illustrates an example of the design space of a three axis multiple master font. In this example, the axes are *weight*, *width*, and *optical size*. It is recommended that a font program be organized to have the lightest weight, narrowest width, and smallest design size mapped to the origin of the blend coordinate space.

Figure 2 *Multiple master font space arrangement*

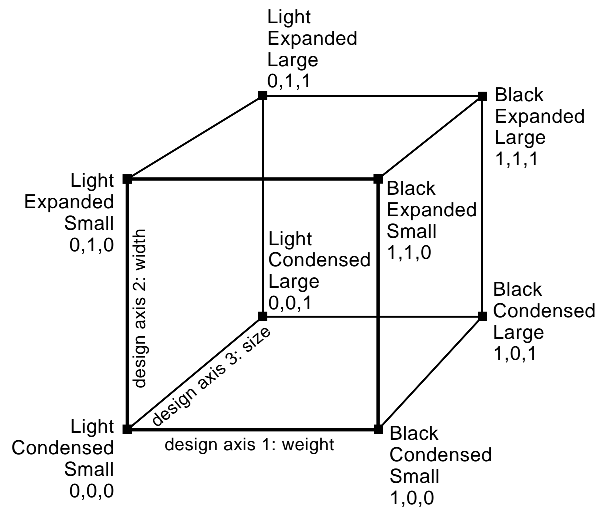
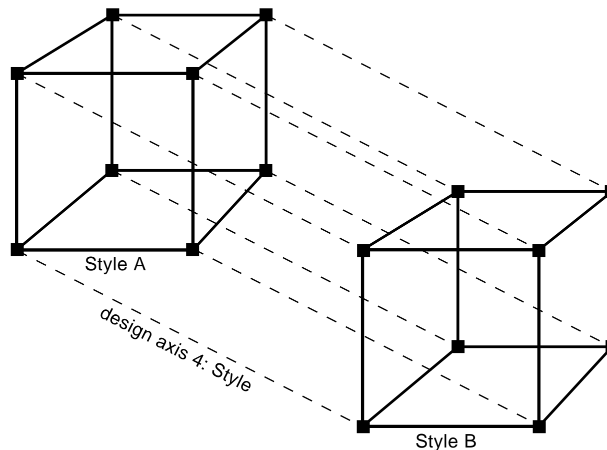


Figure 3 illustrates how a four axis design might be represented. An example of a fourth axis would be a font with an axis for a typographic style (serif – sans serif) or contrast (high/low: the ratio of thick to thin stem widths). This diagram illustrates that if four axes are defined, sixteen master designs are required. Also, since sixteen is the maximum number of designs allowed, there can be no intermediate designs with four axes.

Figure 3 *Arrangement of multiple master design space for a four axis font*



3.2 Multiple Master Font Programs

Multiple master typefaces may contain from two to sixteen master designs, which may be designed to represent from one to four design axes. The allocation of master designs within the sixteen master design limit is expressed by the equation $2^n + x = 16$, where *n* is the number of design axes,

x is the number of intermediate designs (though these are not currently supported by ATM software), and 16 is the maximum allowed number of master designs.

The values used for calculating the weighted interpolation are stored in the font dictionary in the **WeightVector** array. The multiple master font program, as shipped by the font vendor, can have a default setting for the **WeightVector**. It is recommended that it be set so the default font instance will be the normal roman design for that typeface.

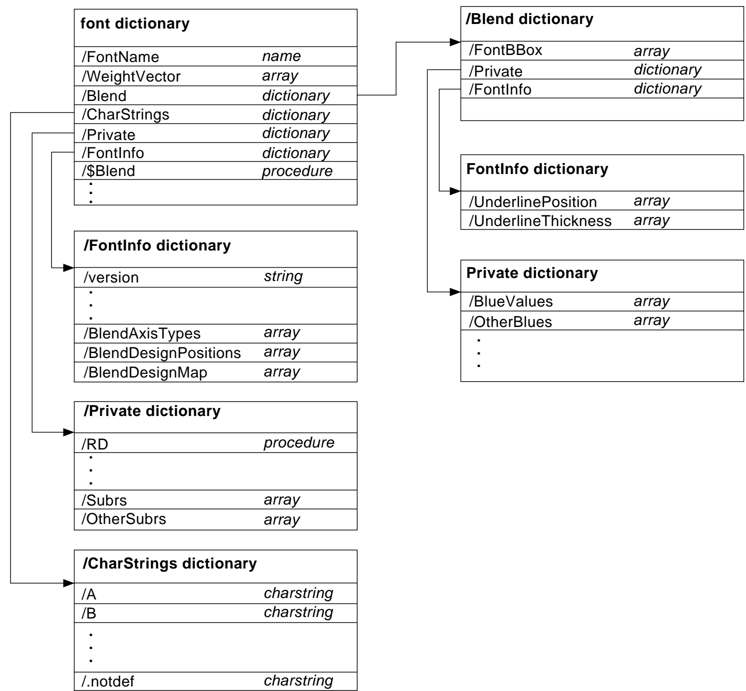
3.3 Multiple Master Font Dictionaries

Figure 4 shows a diagram of the dictionary organization of a multiple master font program. A multiple master font contains a **Blend** dictionary defined in the unencrypted portion of the top level font dictionary. The **Blend** dictionary contains an entry for the interpolated values for **FontBBox**, plus definitions for two subdictionaries: a **Private** and a **FontInfo** dictionary (also referred to as the ***Private*** *blend* and ***Fontinfo*** *blend* dictionaries).

The **Private** blend dictionary (defined in the **Blend** dictionary) will only contain those keywords found in the top level **Private** dictionary which have different values for each master design. If the keywords do not have values that must be interpolated for each instance, they do not need to be in the **Private** subdictionary. The keywords which might be required in the subdictionary are **BlueValues**, **OtherBlues**, **StdHW, StdVW**, **StemSnapH, StemSnapV**, **BlueScale**, **BlueShift**, **FamilyBlues**, **FamilyOtherBlues**, and **ForceBold**.

The values for the **Private** blend dictionary are expressed as an array with one set of values for each master design; the top level **Private** dictionary contains only single value entries (or set of values, as appropriate to the keyword) which have been interpolated using the **WeightVector** specified in the font dictionary.

Figure 4 *Multiple master font dictionaries*



Similarly, the **FontInfo** blend dictionary will contain only keywords found in the top level **FontInfo** dictionary that do not have the same value in each master design. The values for the **UnderlinePosition**, **UnderlineThickness**, and **ItalicAngle** keywords are elements of an array with one value for each master design. It is not necessary to include entries if their values are the same for each design.

The representation of any dictionary entry in the **Blend** dictionary (except **ForceBold**), or in one of the subdictionaries under it, is defined by the following recursive rules:

> *Let "REP(k)" stand for the Blend dictionary representation for the entry k. If k is a number, then REP(k) is the array of numbers [N_1 ... N_k] that are the values for the entry in the k master designs. If k is an array of n items V_1 ... V_n of any type, then REP(k) is an array of the n representations REP(V1) ... REP(Vn). If k is of any type other than number or array, then REP(k) is k itself.*

3.4 Explanation of a Typical Multiple Master Font Program

Example 1 shows a sample multiple master font program for the Myriad™ typeface.

Example 1:

```
%!PS-AdobeFont-1.0: Myriad 000.009
%%CreationDate: Wed Jul 31 11:43:43 1991
%%VMusage: 69881 80580
15 dict begin
/FontInfo 13 dict dup begin
/version (000.009) readonly def
/Notice (Copyright (c) 1991, 1992 Adobe Systems Incorporated.  All
Rights Reserved.) readonly def
/FullName (Myriad) readonly def
/FamilyName (Myriad) readonly def
/Weight (All) readonly def
/ItalicAngle 0 def
/isFixedPitch false def
/UnderlinePosition -100 def
/UnderlineThickness 50 def
/BlendDesignPositions [[0 0] [1 0] [0 1] [1 1]] def
/BlendDesignMap [[[1 0.00][999 1.00]][[1 0.00][999 1.00]]] def
/BlendAxisTypes [/Weight /Width ] def
end readonly def
/FontName /MyriadMM def
/Encoding StandardEncoding def
/PaintType 0 def
/FontType 1 def
/WeightVector [0.18 0.07 0.53 0.22 ] def
/$Blend {0.07 mul exch .53 mul add exch .22 mul add add } bind def
/FontMatrix [0.001 0 0 0.001 0 0] readonly def
/FontBBox{-55.14 -220.84 1148.04 839.18 }readonly def
/Blend 3 dict dup begin
/FontBBox{{-52 -64 -58 -48 }{-212 -216 -224 -222 }{1000 1000 1100
1432 }{828 850 830 867 }}def
/Private 14 dict def
end def
.
% makeblendedfont procedure omitted (see Appendix A)
.
currentdict end
%currentfile eexec
dup /Private 18 dict dup begin
/-|{string currentfile exch readstring pop}executeonly def
/|-{noaccess def}executeonly def
/|{noaccess put}executeonly def
/BlueValues[-11.00 0.00 667.00 685.00 483.48 494.48 650.00 660.56
710.00 720.56 ] def
/OtherBlues[259.46 264.90 -211.20 -200.64 ] def
/BlueScale 0.051208 def
/MinFeature{16 16} |-
/StdHW [67.01] |-
/StdVW [86.14] |-
/StemSnapH [67.01 ] |-
/StemSnapV [86.14 ] |-
/ForceBoldThreshold .57 def
/ForceBold false def
/password 5839 def
3 index /Blend get /Private get begin
/BlueValues[[ -8 -8 -12 -12][ 0 0 0 0][ 664 664 668 668][ 682 682 686
686][ 480 492 480 492][ 488 500 492 504][ 650 650 650 650][ 658 658
```

```
662 660][ 692 692 716 716][ 700 700 728 726]] def
/OtherBlues[[ 258 258 262 255][ 263 263 267 262][ -200 -200 -212
-222][ -192 -192 -200 -212]] def
/BlueScale[ 0.052125 0.052125 0.052125 0.0479583] def
/ForceBold [ false true false true] def
/StdHW [[37 108 39 146 ]] def
/StdVW [[43 155 49 189 ]] def
/StemSnapH [[ 37 108 39 146]] def
/StemSnapV [[ 43 155 49 189]] def
/OtherSubrs
[ {} {} {}
{
systemdict /internaldict known not
{pop 3}
{1183615869 systemdict /internaldict get exec
dup /startlock known
{/startlock get exec}
{dup /strtlck known
{/strtlck get exec}
{pop 3}
ifelse}
ifelse}
ifelse
} executeonly
{} {} {} {} {} {} {} {} {} {}
{ 4 1 roll $Blend } bind
{ exch 8 -3 roll $Blend exch 5 2 roll $Blend } bind
{ 3 -1 roll 12 -3 roll $Blend 3 -1 roll 9 -3 roll $Blend 3 -1 roll 6
3 roll $Blend } bind
{ 4 -1 roll 16 -3 roll $Blend 4 -1 roll 13 -3 roll $Blend 4 -1 roll
10 -3 roll $Blend 4 -1 roll 7 -3 roll $Blend } bind ]|-
```

This font program begins with an allocation of a dictionary with 15 entries, one of which is the top level **FontInfo** dictionary. While the **FontInfo** dictionary is generally optional, it is required for a multiple master font. In addition to the standard entries, this dictionary includes three multiple master keywords which define information about the axes, design space, and the mapping from the design to the blend coordinate space (see section 3.1).

The font dictionary also contains the **WeightVector** keyword which specifies the contribution of each master design for the current font instance. Its value is an array of *k* elements, where *k* is the number of master designs. The elements must sum to 1.0 (with a tolerance of 0.001). It is recommended that the **WeightVector** in a multiple master font program be set to represent the *normal* style for that typeface.

The **WeightVector** entry is followed by a **$Blend** procedure which calculates the weighted average of values from the master designs. It uses the values specified by the **WeightVector** array, and is referenced by **OtherSubrs** entries 14 through 18. This procedure should be of the following form for a multiple master font program with *2* master designs:

```
/$Blend {W1 mul add} bind def
```

where W_1 is the second element in the **WeightVector** array. If there are more than two master designs, the procedure should be of the form:

```
/$Blend { W1 mul exch
   Wi mul add exch
    . . .
   Wk-1 mul add
   add } bind def
```

where the W_i line (W_i mul add exch) is repeated for i = 2 to *(k*–1) where *k* is the number of master designs.

After defining the **FontMatrix**, the **FontBBox** value is defined using either a default value for the chosen default master design, or if it is a font instance, it will have a value interpolated by the **makeblendedfont** procedure.

The **Blend** dictionary is then defined, and a **FontBBox** array containing a set of values for each master design is defined in this dictionary. A **Private** subdictionary is then created, but no entries are defined until after the interpolated entries in the **Private** dictionary have been declared (see section 4, "Adobe Type Manager Compatibility"). This example does not have a **FontInfo** subdictionary under the **Blend** dictionary because the values for Myriad are the same for all of the master designs.

The next section of code is the definition of the **makeblendedfont** procedure. This is included in the font for backward compatibility with interpreters in which this operator is not defined (see section 3.6).

All of the remaining code in the example is in the **eexec** encrypted section of the font. This includes the top level **Private** dictionary, with its interpolated values based on the **WeightVector** values in the font; and the **Private** blend dictionary which was allocated and defined in the **Blend** dictionary, but whose entries are specified in this encrypted portion of the font program. The entries in the **Private** blend dictionary contain arrays with one value for each master design.

A **ForceBold** array may be included in the **Private** blend dictionary of the **Blend** dictionary. When this array is present, a new keyword, the keyword **ForceBoldThreshold** must be included in the top level **Private** dictionary. The value for **ForceBoldThreshold** is a number. If the sum of the **WeightVector** elements, for which **ForceBold** is *true* in the corresponding multiple master font, is greater than or equal to **ForceBoldThreshold**, then **ForceBold** is *true* for the font instance with that **WeightVector**.

The **Private** dictionary continues with the global hint operators, which are the same as for regular Type 1 font programs except that there is one value for each master design. The **OtherSubrs** array for Myriad includes null procedures for the flex mechanism **OtherSubrs**, and includes the **OtherSubrs** code for hint substitution. Next are ten null procedures before

the code for **OtherSubrs** entries 14 through18. In the case of Myriad, only **OtherSubrs** entries 14 through 17 are included because entry 18 (which returns 6 values) is not used in this particular font program.

3.5 Multiple Master Keywords and Procedures

The following keywords are required entries in the **FontInfo** dictionary of a multiple master font (for which the **FontInfo** dictionary is a required dictionary).

BlendAxisTypes

(Required.) **BlendAxisTypes** is an array of *n* PostScript language names where *n* is the dimensionality of the design space and hence the number of axes. Each string specifies the corresponding axis type. In the above 3-axis example, this value would be:

```
/BlendAxisTypes [/Weight /Width /OpticalSize]
```

These three axes should always occur in this relative order.

Note *The keywords* Weight*,* Width*, and* OpticalSize *are reserved for use as axis types for multiple master font programs. Font developers interested in registering new types for additional design axes should write to:*

UniqueID Coordinator
Adobe Developer Relations
Adobe Systems Incorporated
P.O. Box 7900
Mountain View, CA 94039-7900

BlendDesignPositions

(Required.) **BlendDesignPositions** is an array of *k* arrays giving the locations of the *k* master designs in the blend space. Each location subarray has *n* numbers giving the location of the design in the *n* dimensions of the design space, with a minimum value of zero and a maximum value of one. The order of the entries in the array must be the order of the corresponding master designs in the font.

Table 1 shows an example of a font with eight master designs based on the example shown in Figure 3.

Table 1

Design label	*Blend space coordinates*
design 1: light condensed small	0 0 0
design 2: light expanded small	0 1 0
design 3: black condensed small	1 0 0
design 4: black expanded small	1 1 0
design 5: light condensed large	0 0 1
design 6: light expanded large	0 1 1
design 7: black condensed large	1 0 1
design 8: black expanded large	1 1 1

The **BlendDesignPositions** array for this font would be:

```
/BlendDesignPositions [ [0 0 0] [0 1 0] [1 0 0] [1 1 0]
   [0 0 1] [0 1 1] [1 0 1] [1 1 1] ] def
```

Note *While the relative order of the design axes are specified in this document, the order of the master designs is not. However, it is imperative that the order of the master designs specified in* **BlendDesignPositions***, the order of the* **WeightVector** *values, the order of the charstring arguments, and the configuration of the* **NormalizeDesignVector** *and* **ConvertDesignVector** *procedures must all correspond, or unexpected results will occur.*

BlendDesignMap

BlendDesignMap (Required.) is an entry consisting of an array of *n* arrays where *n* is the dimensionality of the design space. Each array contains *m* subarrays that specify the mapping of design coordinates into blend coordinates for each axis.

The data for the coordinate mapping for the **BlendDesignMap** keyword is of the form:

$$[\ [D_1\ B_1] \ldots [D_m\ B_m]\ ]_{A1} \ldots [\ [D_1\ B_1] \ldots [D_m\ B_m]\ ]_{An}$$

where D_1 and B_1 are the lower limits of the design and blend coordinate range, respectively; and D_m and B_m are the upper limits of the design and blend coordinate ranges. The subscript *A1* designates the mapping data for the first axis, and *An* represents the mapping data for the last axis of an *n* axis font. The subscript *m* represents the number of points defining the mapping from design to blend coordinates. The minimum value allowed for *m* is two

(for a linear mapping), and the maximum is twelve. Also, the value for *m* may be different for each axis. The order of the subarrays must correspond to the order of design axes in **BlendAxisTypes**.

Example 2 illustrates the values of a sample **BlendDesignMap** for a font with three axes: *Weight*, with design coordinates from 200 to 900; *Width*, with design coordinates from 300 to 700; and *OpticalSize*, with design coordinates from 6 (point) to 72.

Example 2:

```
/BlendDesignMap [ [[200 0] [500 .5] [900 1]]  [[300 0] [700 1]]
   [ [6 0] [11 .5] [72 1]] ]
```

This capability for piecewise linear mapping of the coordinate range is particularly important for achieving optimal results for the *OpticalSize* axis. To be optically correct, small changes in design coordinates, such as changing from 6- to 8-point, requires significantly more change in the blend coordinates (and hence in the shape of the glyph) than does a change from 66- to 68-point. Without this capability, at least one additional intermediate master design would have to be included in the font.

3.6 The makeblendedfont Procedure

makeblendedfont – blendedfontdict weightvector **makeblendedfont** blendedfontdict´ –

This operator creates a font dictionary with pre-interpolated entries. The *blendedfontdict* argument is a font dictionary of an existing multiple master font; it can be from either the original multiple master font itself, or from an interpolated font instance since any **Blend** dictionary contains all elements needed to derive additional font instances.

The *weightvector* argument is an array of numbers summing to 1.0 to be used as the weighting values for interpolating the new font instance. The value of **WeightVector** in *blendedfontdict´* is set to the values in the array *weightvector*. Interpolated values are calculated for entries in the **Private** and **FontInfo** dictionaries. The result is a font dictionary that can be used as an argument to **definefont**. The resulting dictionary and its contents will still have *read-write* permission, so the caller of **makeblendedfont** can make further modifications if necessary (such as assigning a **UniqueID**). This **makeblendedfont** operator or procedure will not copy **FID**s, **UniqueID**s, or **XUID**s.

For backward compatibility, the downloadable file for a multiple master typeface must include conditional code (shown in Appendix A) which will check for an existing definition of **makeblendedfont** in either **systemdict**, **shareddict** or **userdict**, and only if none exists will it store a new definition in **shareddict** or **userdict**. If a definition already exists, the font program will

reclaim the storage of its own definition by using **save/restore** and use the existing version (unless the downloaded font has a newer version number than the existing font).

The **Blend** dictionary data structures provide the information needed by the **makeblendedfont** procedure. This makes it unnecessary to have the **makeblendedfont** procedure contain a list of entries to be interpolated, which means that the procedure can be used in the future, even if the set of entries to be interpolated varies in future fonts.

3.7 The Multiple Master findfont Procedure

Multiple master font programs from Adobe Systems include a procedure which will alter the behavior of the **findfont** operator in **systemdict**. For Level 1 interpreters, **findfont** is redefined with a new definition in another dictionary. In Level 2 interpreters, the FindResource procedure is replaced in the /Font resource category implementation. This is necessary because of the need to generate font instances on-the-fly to satisfy multiple master font references in a PostScript language document.

The code for the multiple master version of the **findfont** operator is available from the Adobe Developers Association. Adobe Systems grants permission to use this code as long as the code is not altered and the copyright notice remains intact.

The procedure creates all necessary font instances before it calls the standard **findfont** procedure. These instances are only created if the font name conforms to the naming conventions for a multiple master font. The design coordinates must be separated from the family and style name by an *underscore* character; there must be a numeric design coordinate for each axis in the font, and these coordinates must be separated by non-numeric characters. For more information on multiple master font names, see Adobe Technical Note #5088, "Font Naming Issues."

In the situation where a multiple master font has been downloaded to a printer's hard disk, the alternate **findfont** may not be instantiated when a job referencing multiple master font instances is being interpreted. The solution is to have a *Sys/Start* file containing the **findfont** definition on the hard disk. The interpreter executes the *Sys/Start* file upon startup, thus ensuring that the necessary **findfont** is defined.

An example of a call to **findfont** might look like:

```
MyriadMM_367wd_450wt findfont
```

The redefined **findfont** procedure parses the name and calls the **NormalizeDesignVector** procedure (see below) to convert the design coordinates in the **FontName** into normalized coordinates. It then calls

ConvertDesignVector (see below) to convert these into **WeightVector** values for use as arguments for calling **makeblendedfont**, which leaves the font dictionary of the font instance on the stack.

3.8 The NormalizeDesignVector Procedure

– $d_1 \ldots d_n$ **NormalizeDesignVector** $nc_1 \ldots nc_n$ –

NormalizeDesignVector is a procedure that must be included in a multiple master font program; it is used by the **findfont** procedure to calculate the normalized equivalent of the design coordinates in the **FontName**. If the values in the **BlendDesignMap** array for a particular axis indicate that the mapping is non-linear, the normalized values must be found by piecewise linear interpolation of the design coordinates using the appropriate segment of the map.

The normalized coordinates nc_1 through nc_n are left on the stack for use by the **ConvertDesignVector** procedure. The code for this procedure must be configured for the number of axes and master designs in the font program in which they are used. Sample code for a representative multiple master font is shown in Appendix C.

3.9 The ConvertDesignVector Procedure

– $nc_1 \ldots nc_n$ **ConvertDesignVector** $V_1 \ldots V_k$ –

ConvertDesignVector is a required procedure that takes the normalized coordinates nc_1 through nc_n, which were left on the stack by the call to **NormalizeDesignVector**, and generates **WeightVector** values $V_1 \ldots V_k$ by a simple linear weighting with the following properties (see Figure 3 for illustration of the design space):

- The **WeightVector** value for any master design is 0 (zero) when the instance is another master design (for example, the instance is at another corner of the design space).
- The **WeightVector** value for any master design is 1 when the instance is that master design.
- When the instance is in the middle of the design space, all master fonts contribute equally.

The code for this procedure must be configured for the number of axes and master designs in the font program in which they are used. Appendix D shows an example of the necessary calculations for a sample multiple master font as well as an example of the code that would be included in the font.

3.10 Multiple Master Font Names

The PostScript language **FontName** and the font menu name of multiple master fonts require special attention, both for compatibility reasons and to standardize the meaning of design coordinates in order to benefit users, software applications, and utilities. See Technical Note #5088, "Font Naming Issues" for more information on multiple master font names.

3.11 Multiple Master Charstring Representation

The encoded and encrypted data in the charstring procedures and **Subrs** array entries contain the raw (not interpolated) data from each of the *k* master designs, along with calls to the **OtherSubrs** procedures used for multiple master interpolation (see following section on **OtherSubrs**). Each glyph in each master design must be represented by an identical sequence of commands. The different master designs can differ only in numerical values for their arguments. For example, if the first command in each path for a given glyph in a single Type 1 font is

```
dx dy rmoveto
```

then the first command for that glyph in a multiple master font would be

$$dx_1\ dy_1\ (dx_2-dx_1)\ \ldots\ (dx_k-dx_1)\ (dy_2-dy_1)\ \ldots\ (dy_k-dy_1)\ \texttt{15 callsubr rmoveto}$$

where dx_i and dy_i are the values from the ith master design.

Note *In the above example, as well as ones that follow, expressions such as (dx_2-dx_1) are a symbolic representation of what must be encoded in the charstring procedure. The Type 1 BuildChar interpreter cannot interpret such an arithmetic expression, it is the difference between dx_2 and dx_1 that is encoded.*

This format makes it possible to use the same charstrings and **Subrs** for all font instances derived from the multiple master font. In this example, 2**k* values are put on the stack, and **Subrs** entry 15 calls **OtherSubrs** entry 15 which calculates the weighted average for both dx and dy, using the **WeightVector.** This call returns the interpolated values of *dx* and *dy* on the stack; these values are then used as arguments to **rmoveto**.

Since the font interpreter stack is limited to 24 entries, font programs with four axes may need to call the interpolation procedures in a way that avoids too many elements accumulating on the stack. The limit is effectively 22 items on the stack since **callothersubr** requires two arguments to pass the **Subrs** entry number and the number of arguments.

For example, the **rrcurveto** operator requires six arguments. If the font has four master designs, then **OtherSubrs** entry 18 cannot be used and the set of arguments must be split into two calls. One way to do this is by calling **OtherSubrs** entry 16 twice. In the following example, if there are *k* master designs, there will be *k* sets of:

```
a b c d e f
```

to be interpolated for each **rrcurveto** operator. If $k = 4$, there would be 26 elements on the stack (including the two extra arguments mentioned above). If these arguments were divided into two calls to **OtherSubrs** entry 16, each of which returns three results, the code would look like:

```
a1 b1 c1 (a2-a1)...(an-a1) (b2-b1)...(bn-b1) (c2-c1)...(cn-c1)
16 callsubr
d1 e1 f1 (d2-d1)...(dn-d1) (e2-e1)...(en-e1) (f2-f1)...(fn-f1)
16 callsubr  rrcurveto
```

In this example, **Subrs** entry 16 is used to call **OtherSubrs** entry 16 as a means of saving space.

3.12 OtherSubrs for Multiple Master Font Programs

There are five new entries in the **OtherSubrs** array which are used by multiple master font programs to compute weighted averages using the **WeightVector**. The new entries are numbered 14 through 18, so the necessary number of procedure brackets ("{ }") must be inserted in the array to fill unused positions.

OtherSubrs 14 through 18 consist of PostScript language code whose only purpose is to reorder the arguments on the stack before calling the **$Blend** procedure (discussed in section 3.4) to interpolate those arguments. These routines differ only in the number of results they return. Each must be configured to manipulate the expected number of elements on the stack, which is dependent on the number of master designs, so that they are in the correct order for calling the **$Blend** procedure.

Note *There is no requirement for the number of a* **Subrs** *procedure to correspond to the number used for an* **OtherSubrs** *procedure.*

In the summary of **OtherSubrs** calls listed below, *k* is the length of the **WeightVector** array (and hence the number of master designs in the font). The charstrings and **Subrs** will call the appropriate **OtherSubrs** to create the required weighted averages for various parameters. The font interpreter stack is limited to 24 entries, which includes the arguments used to indicate the **OtherSubrs** entry number and the number of arguments being passed. Therefore, some of the **OtherSubrs** entries may only be useful with fonts having a small value of *k*.

The **OtherSubrs** for multiple master fonts are numbered 14 through 18; the calling sequences are shown below. In each case, *k* is the number of master designs in the font (The maximum value of *k* for any font is 16.). Each of the following descriptions uses a notation of the form:

a_1 $(a_2 - a_1)$ $(a_3 - a_1)$ $(a_4 - a_1)$ ... $(a_k - a_1)$ 14 **callsubr** *a*

which indicates the form of the invocation in a charstring. This is the form in which the values for each master design are represented in the font program, with a_1 being the character coordinate value for the first master design, and all subsequent values are expressed as *deltas* relative to the first value. **Subrs** entry 14 puts the argument count on the stack and calls **OtherSubrs** entry 14 (see section 3.13), which arranges the elements on the PostScript language stack and calls the **$Blend** procedure to calculate the weighted average of the input values. In each example, the number of items left on the stack is indicated by the characters to the right of the arrow.

OtherSubrs 14: Input: k values; Result: 1 value

a_1 $(a_2 - a_1)$ $(a_3 - a_1)$ $(a_4 - a_1)$ ... $(a_k - a_1)$ Subr# **callsubr** *a*

where Subr# is the index of the **Subrs** entry that calls **OtherSubrs** entry 14. Entry 14 uses **WeightVector** values to form a weighted average of *k* values from the stack. The results are pushed onto the stack. The value of *k* is found from the length of the **WeightVector** array.

OtherSubrs 15: Input: k × 2 values; Results: 2 values

a_1 b_1 $(a_2 - a_1)$ $(a_3 - a_1)$ $(a_4 - a_1)$ ... $(a_k - a_1)$ $(b_2 - b_1)$ $(b_3 - b_1)$ $(b_4 - b_1)$ ...
$(b_k - b_1)$ Subr# **callsubr** *a b*

where Subr# is the index of the **Subrs** entry that calls **OtherSubrs** entry 15. Entry 15 uses **WeightVector** values to form two weighted averages, one for the ‘a’ values and the other for the ‘b’ values indicated in the pseudo code above.

OtherSubrs 16: Input: k × 3 values; Results: 3 values

a_1 b_1 c_1 $(a_2 - a_1)$ ... $(a_k - a_1)$ $(b_2 - b_1)$ ... $(b_k - b_1)$ $(c_2 - c_1)$ ... $(c_k - c_1)$
Subr# **callsubr** *a b c*

where Subr# is the index of the **Subrs** entry that calls **OtherSubrs** entry 16. Entry 16 uses **WeightVector** values to form three weighted averages.

OtherSubrs 17: Input: k × 4 values; results: 4 values

a_1 b_1 c_1 d_1 $(a_2 - a_1)$ ... $(a_k - a_1)$ $(b_2 - b_1)$ ... $(b_k - b_1)$ $(c_2 - c_1)$ ... $(c_k - c_1)$ $(d_2 - d_1)$
... $(d_k - d_1)$ Subr# **callsubr** *a b c d*

where Subr# is the index of the **Subrs** entry that calls **OtherSubrs** entry 17. Entry 17 uses **WeightVector** values to form four weighted averages.

OtherSubr 18: Input: k $\times$ 6 values; Results: 6 values

$a_1\ b_1\ c_1\ d_1\ e_1\ f_1\ (a_2-a_1) \ldots (a_k-a_1)\ \ (b_2-b_1) \ldots (b_k-b_1)\ \ (c_2-c_1) \ldots (c_k-c_1)$ $(d_2-d_1) \ldots (d_k-d_1)\ \ (e_2-e_1) \ldots (e_k-e_1)\ \ (f_2-f_1) \ldots (f_k-f_1)$ Subr# **callsubr** *a b c d e f*

Note *where* Subr# *is the index of the* **Subrs** *entry that calls* **OtherSubrs** *entry 18. Entry 18 uses* **WeightVector** *values to form six weighted averages.*

Note *Some PostScript interpreters fail to execute the PostScript language* ***OtherSubrs*** *procedures. This is incorrect behaviour. This primarily affects the Apple Personal LaserWriter® NT and the Hewlett-Packard® Level 1 PostScript Cartridge for LaserJet® printers. The likely consequence is that an* invalidfont *error will occur, the fonts will not appear on the page, or the interpreter will fail.*

3.13 Sample Subrs Code for Calling OtherSubrs Procedures

Since the charstring encoding for a **Subrs** call is shorter than that for an **OtherSubrs** call, use of **Subrs** to call **OtherSubrs** may make a Type 1 font program more concise. The following **Subrs** are examples of subroutines which may be used to call **OtherSubrs** entries 14 through 18. These are only selected examples; additional subroutines must be appropriately configured for the number of master designs in the font. The number of arguments expected by the charstring command determines which **OtherSubrs** is called.

For example, if a font has four master designs, and it is necessary to interpolate arguments for an **hlineto** command which expects a single argument on the stack, **OtherSubrs** entry 14 would be called in a **Subrs** entry with the following code:

```
4 14 callothersubr pop return
```

In this example, the first argument indicates that there are four arguments (as shown in section 3.12) being passed to **OtherSubrs** entry 14. If there were eight masters, the code would be:

```
8 14 callothersubr pop return
```

To interpolate multiple master charstring arguments, in a font with four master designs, for an **rrcurveto** command which expects six arguments, it might be guessed that the **Subrs** would use the following code:

```
24 18 callothersubr pop pop pop pop pop pop return
```

However, the result of this code would exceed the stack limit of 24 elements since there are 24 arguments being put on the stack in addition to the two given as arguments to the **callothersubr** command. The solution is to make two calls to **OtherSubrs** entry 16, each of which produces 3 results.

Recall that the arguments for entry 18 (as originally planned) would be set up as follows:

$a_1\ b_1\ c_1\ d_1\ e_1\ f_1\ (a_2\text{-}a_1)\ (a_3\text{-}a_1)\ (a_4\text{-}a_1)\ (b_2\text{-}b_1)\ (b_3\text{-}b_1)\ (b_4\text{-}b_1)\ (c_2\text{-}c_1)$
$(c_3\text{-}c_1)\ (c_4\text{-}c_1)\ (d_2\text{-}d_1)\ (d_3\text{-}d_1)\ (d_4\text{-}d_1)\ (e_2\text{-}e_1)\ (e_3\text{-}e_1)\ (e_4\text{-}e_1)\ (f_2\text{-}f_1)$
$(f_3\text{-}f_1)\ (f_4\text{-}f_1)$

These must be reorganized for the call to look like:

$a_1\ b_1\ c_1\ (a_2\text{-}a_1)\ (a_3\text{-}a_1)\ (a_4\text{-}a_1)\ (b_2\text{-}b_1)\ (b_3\text{-}b_1)\ (b_4\text{-}b_1)\ (c_2\text{-}c_1)\ (c_3\text{-}c_1)$
$(c_4\text{-}c_1)$ `Subr# callsubr`

$d_1\ e_1\ f_1\ (d_2\text{-}d_1)\ (d_3\text{-}d_1)\ (d_4\text{-}d_1)\ (e_2\text{-}e_1)\ (e_3\text{-}e_1)\ (e_4\text{-}e_1)\ (f_2\text{-}f_1)\ (f_3\text{-}f_1)$
$(f_4\text{-}f_1)$ `Subr# callsubr`

where the **Subrs** procedure indicated by Subr# would contain:

```
12 16 callothersubr pop pop pop return
```

Again, the stack may never have more than 24 elements. This must be considered when breaking up the calls, as in the above example, where the intermediate results are left on the stack. Also, while making subroutines to conserve space is encouraged, the cumulative effect on stack contents must be carefully controlled.

4 Adobe Type Manager Compatibility

The following are compatibility issues related to multiple master fonts:

- The **Blend** dictionary must come after everything in the font dictionary for which blended values can be calculated.
- The keywords **BlendDesignPositions**, **BlendDesignMap**, and **BlendAxisTypes** must be defined before the **Blend** dictionary.
- The **Private** blend dictionary must appear after all elements of the **Private** dictionary for which blended values can be calculated.

5 Font Program Testing

Over time, several Type 1 font program interpreters have been developed, including those in PostScript printers, Adobe Type Manager software, and the Type 1 Coprocessor (an ASIC chip). All of these accept any Type 1 font program which conforms to the Type 1 specification, but they differ in how

they handle non-conforming font programs. In particular, ATM software is stricter than the PostScript interpreter, and the Type 1 Coprocessor is stricter still. When developing Type 1 font programs, it is wise to test with the following: a Level 1 PostScript printer, a Level 2 PostScript printer with and without a Type 1 Coprocessor, and a later version of ATM software (preferably one shipped with the SuperATM™ or Adobe Acrobat™ software). For East Asian fonts, testing should include the above plus a Level 1 Japanese-enabled printer.

Note *An example of the range of charstring character space coordinates allowed in different implementations is that the Type 1 specification limit is ±2000, but the Type 1 Coprocessor chip supports ±4095, while ATM software supports ±8191.*

6 Errata

The following errors occur in versions 1.0 and 1.1 of the Type 1 Font Format book:

- There is an error in the sample Type 1 font program code shown in Example 1 on page 11. The hex code which follows the **eexec** operator cannot be decrypted into a meaningful **Private** dictionary, and hence should not be used as a test case for developing a decryption procedure. The correct **eexec** hex code for the beginning of the Symbol font is:

```
a8686bfddf470dd119f86e1b8e5b290ae7d910e9317a36f6768d8de89e7ed5b8
45166db0e18e3fca77c6e789f2ac61e3ba2248c0c4ccdb4c503448893c2a909c
36546b763088822eb34d1051d0ac662d8098db11f0a527a679e4ac03347df431
9a689d7d65239e8502b5db9aef94cd6cebd07cee5af22db4c8c628a982cdd10
```

- The description of the **seac** operator in paragraph 6.4 of versions 1.0 and 1.1 contains an error in the description of the *adx* and *ady* arguments. The existing text describes the offset as being the distance between the origin points of the base and accent character; it should read *the offset of the left sidebearing points*.

Appendix A: The makeblendedfont Operator

The following code is the definition of the **makeblendedfont** operator. It has been updated since it was published in Adobe Technical Note #5086, “Multiple Master Extensions to the Adobe Type 1 Font Format.”

Note *This code, as well as the code in the following appendices, is copyrighted by Adobe Systems Incorporated, and may not be reproduced except by permission of Adobe Systems Incorporated. Adobe Systems Incorporated grants permission to use this code in Type 1 font programs, as long as the code is used as it appears in this document, the copyright notice remains intact, and the character outline code included in such a font program is neither copied nor derived from character outline code in any Adobe Systems font program.*

```
% Copyright (c) 1990-1994 Adobe Systems Incorporated.
   % All Rights Reserved.
   % This code to be used for Flex and hint replacement.
   % Version 11

/shareddict where
{ pop currentshared { setshared } true setshared
   shareddict }
{ {} userdict } ifelse dup
/makeblendedfont where {/makeblendedfont get dup type /
   operatortype eq {
pop false} { 0 get dup type /integertype ne
{pop false} {11 lt} ifelse} ifelse } {true}ifelse
{/makeblendedfont {
11 pop
2 copy length exch /WeightVector get length eq
{ dup 0 exch {add} forall 1 sub abs .001 gt }
{ true } ifelse
{ /makeblendedfont cvx errordict /rangecheck get exec }
   if
exch dup dup maxlength dict begin {
false {/FID /UniqueID /XUID } { 3 index eq or } forall
 { pop pop } { def } ifelse
} forall
/XUID 2 copy known{
```

```
get dup length 2 index length sub dup 0 gt{
exch dup length array copy
exch 2 index{65536 mul cvi 3 copy put pop 1 add}forall
  pop/XUID exch def
}{pop pop}ifelse
}{pop pop}ifelse
{ /Private /FontInfo } {
dup load dup maxlength dict begin {
false { /UniqueID /XUID } { 3 index eq or } forall
{ pop pop }{ def } ifelse } forall currentdict end def
} forall
dup /WeightVector exch def
dup /$Blend exch [
exch false exch
dup length 1 sub -1 1 {
1 index dup length 3 -1 roll sub get
dup 0 eq {
pop 1 index {/exch load 3 1 roll} if
/pop load 3 1 roll
} {dup 1 eq {pop}
{2 index {/exch load 4 1 roll} if
3 1 roll /mul load 3 1 roll } ifelse
1 index {/add load 3 1 roll} if
exch pop true exch} ifelse
} for
pop { /add load } if
] cvx def
{2 copy length exch length ne {/makeblendedfont cvx er-
  rordict /typecheck get exec}if
0 0 1 3 index length 1 sub {
dup 4 index exch get exch 3 index exch get mul add
} for
exch pop exch pop}
{{dup type dup dup /arraytype eq exch /packedarraytype
  eq or {
  pop 1 index /ForceBold eq {
  5 index 0 0 1 3 index length 1 sub {
  dup 4 index exch get {2 index exch get add } {pop} if-
  else
  } for exch pop exch pop
  2 index /ForceBoldThreshold get gt 3 copy} {
{length 1 index length ne { pop false } {
true exch { type dup /integertype eq exch /realtype eq
  exch or and } forall
} ifelse }
2 copy 8 index exch exec {pop 5 index 5 index exec}
{exch dup length array 1 index xcheck { cvx } if
dup length 1 sub 0 exch 1 exch {
dup 3 index exch get dup type dup /arraytype eq exch /
  packedarraytype eq or {
dup 10 index 6 index exec {
```

```
9 index exch 9 index exec} if } if 2 index 3 1 roll put
} for exch pop exch pop
} ifelse 3 copy
1 index dup /StemSnapH eq exch /StemSnapV eq or {
dup length 1 sub {dup 0 le { exit } if
dup dup 1 sub 3 index exch get exch 3 index exch get 2
  copy eq {
pop 2 index 2 index 0 put 0 } if le {1 sub}
{dup dup 1 sub 3 index exch get exch 3 index exch get
3 index exch 3 index 1 sub exch put
3 copy put pop
2 copy exch length 1 sub lt {1 add} if} ifelse} loop
  pop
dup 0 get 0 le {
dup 0 exch {0 gt { exit } if 1 add} forall
dup 2 index length exch sub getinterval} if } if } ife-
  lse put }
{/dicttype eq {6 copy 3 1 roll get exch 2 index exec}
{/makeblendedfont cvx errordict /typecheck get exec}
  ifelse
} ifelse pop pop } forall pop pop pop pop }
currentdict Blend 2 index exec
currentdict end
} bind put
/$fbf {FontDirectory counttomark 3 add -1 roll known {
cleartomark pop findfont}{
] exch findfont exch makeblendedfont
dup /Encoding currentfont /Encoding get put definefont
} ifelse currentfont /ScaleMatrix get makefont setfont
} bind put } { pop pop } ifelse exec
```

Appendix B: Updated OtherSubrs Code for Flex and Hint Substitution

The code in this appendix is the updated code for flex and hint substitution; this code appeared in Version 1.1 of the Adobe Type 1 Font Format Book, but is included here for readers having only Version 1.0.

```
% Copyright (c) 1987-1990 Adobe Systems Incorporated.
% All Rights Reserved.
% This code to be used for Flex and hint replacement.
% Version 1.1
/OtherSubrs
[systemdict /internaldict known
{1183615869 systemdict /internaldict get exec
/FlxProc known {save true} {false} ifelse}
{userdict /internaldict known not {
userdict /internaldict
{count 0 eq
{/internaldict errordict /invalidaccess get exec} if
dup type /integertype ne
{/internaldict errordict /invalidaccess get exec} if
dup 1183615869 eq
{pop 0}
{/internaldict errordict /invalidaccess get exec}
ifelse
}
dup 14 get 1 25 dict put
bind executeonly put
} if
1183615869 userdict /internaldict get exec
/FlxProc known {save true} {false} ifelse}
ifelse
[
systemdict /internaldict known not
{ 100 dict /begin cvx /mtx matrix /def cvx } if
systemdict /currentpacking known {currentpacking true setpacking} if
{
systemdict /internaldict known {
1183615869 systemdict /internaldict get exec
dup /$FlxDict known not {
dup dup length exch maxlength eq
{ pop userdict dup /$FlxDict known not
{ 100 dict begin /mtx matrix def

dup /$FlxDict currentdict put end } if }
{ 100 dict begin /mtx matrix def
```

```
dup /$FlxDict currentdict put end }
ifelse
} if
/$FlxDict get begin
} if
grestore
/exdef {exch def} def
/dmin exch abs 100 div def
/epX exdef /epY exdef
/c4y2 exdef /c4x2 exdef /c4y1 exdef /c4x1 exdef /c4y0 exdef /c4x0
exdef
/c3y2 exdef /c3x2 exdef /c3y1 exdef /c3x1 exdef /c3y0 exdef /c3x0
exdef
/c1y2 exdef /c1x2 exdef /c2x2 c4x2 def /c2y2 c4y2 def
/yflag c1y2 c3y2 sub abs c1x2 c3x2 sub abs gt def
/PickCoords {
{c1x0 c1y0 c1x1 c1y1 c1x2 c1y2 c2x0 c2y0 c2x1 c2y1 c2x2 c2y2 }
{c3x0 c3y0 c3x1 c3y1 c3x2 c3y2 c4x0 c4y0 c4x1 c4y1 c4x2 c4y2 }
ifelse
/y5 exdef /x5 exdef /y4 exdef /x4 exdef /y3 exdef /x3 exdef
/y2 exdef /x2 exdef /y1 exdef /x1 exdef /y0 exdef /x0 exdef
} def
mtx currentmatrix pop
mtx 0 get abs .00001 lt mtx 3 get abs .00001 lt or
{/flipXY -1 def }
{mtx 1 get abs .00001 lt mtx 2 get abs .00001 lt or
{/flipXY 1 def }
{/flipXY 0 def }
ifelse }
ifelse
/erosion 1 def
systemdict /internaldict known {
1183615869 systemdict /internaldict get exec dup
/erosion known
{/erosion get /erosion exch def}
{pop}
ifelse
} if
yflag
{flipXY 0 eq c3y2 c4y2 eq or
{false PickCoords }
{/shrink c3y2 c4y2 eq
{0}{c1y2 c4y2 sub c3y2 c4y2 sub div abs} ifelse def
/yshrink {c4y2 sub shrink mul c4y2 add} def
/c1y0 c3y0 yshrink def /c1y1 c3y1 yshrink def
/c2y0 c4y0 yshrink def /c2y1 c4y1 yshrink def
/c1x0 c3x0 def /c1x1 c3x1 def /c2x0 c4x0 def /c2x1 c4x1 def
/dY 0 c3y2 c1y2 sub round
dtransform flipXY 1 eq {exch} if pop abs def
dY dmin lt PickCoords
y2 c1y2 sub abs 0.001 gt {
c1x2 c1y2 transform flipXY 1 eq {exch} if
/cx exch def /cy exch def
/dY 0 y2 c1y2 sub round dtransform flipXY 1 eq {exch}
if pop def
dY round dup 0 ne
{/dY exdef }
{pop dY 0 lt {-1}{1} ifelse /dY exdef }
```

```
ifelse
/erode PaintType 2 ne erosion 0.5 ge and def
erode {/cy cy 0.5 sub def} if
/ey cy dY add def
/ey ey ceiling ey sub ey floor add def
erode {/ey ey 0.5 add def} if
ey cx flipXY 1 eq {exch} if itransform exch pop
y2 sub /eShift exch def
/y1 y1 eShift add def /y2 y2 eShift add def /y3 y3
eShift add def
} if
} ifelse
}
{flipXY 0 eq c3x2 c4x2 eq or
{false PickCoords }
{/shrink c3x2 c4x2 eq
{0}{c1x2 c4x2 sub c3x2 c4x2 sub div abs} ifelse def
/xshrink {c4x2 sub shrink mul c4x2 add} def
/c1x0 c3x0 xshrink def /c1x1 c3x1 xshrink def
/c2x0 c4x0 xshrink def /c2x1 c4x1 xshrink def
/c1y0 c3y0 def /c1y1 c3y1 def /c2y0 c4y0 def /c2y1 c4y1 def
/dX c3x2 c1x2 sub round 0 dtransform
flipXY -1 eq {exch} if pop abs def
dX dmin lt PickCoords
x2 c1x2 sub abs 0.001 gt {
c1x2 c1y2 transform flipXY -1 eq {exch} if
/cy exch def /cx exch def
/dX x2 c1x2 sub round 0 dtransform flipXY -1 eq {exch} if pop def
dX round dup 0 ne
{/dX exdef }
{pop dX 0 lt {-1}{1} ifelse /dX exdef }
ifelse
/erode PaintType 2 ne erosion .5 ge and def
erode {/cx cx .5 sub def} if
/ex cx dX add def
/ex ex ceiling ex sub ex floor add def
erode {/ex ex .5 add def} if
ex cy flipXY -1 eq {exch} if itransform pop
x2 sub /eShift exch def
/x1 x1 eShift add def /x2 x2 eShift add def /x3 x3 eShift add def
} if
} ifelse
} ifelse
x2 x5 eq y2 y5 eq or
{ x5 y5 lineto }
{ x0 y0 x1 y1 x2 y2 curveto
x3 y3 x4 y4 x5 y5 curveto }
ifelse
epY epX
}
systemdict /currentpacking known {exch setpacking} if
/exec cvx /end cvx ] cvx
executeonly
exch
{pop true exch restore}
{
systemdict /internaldict known not
{1183615869 userdict /internaldict get exec
```

```
exch /FlxProc exch put true}
{1183615869 systemdict /internaldict get exec
dup length exch maxlength eq
{false}
{1183615869 systemdict /internaldict get exec
exch /FlxProc exch put true}
ifelse}
ifelse}
ifelse
{systemdict /internaldict known
{{1183615869 systemdict /internaldict get exec /FlxProc get exec}}
{{1183615869 userdict /internaldict get exec /FlxProc get exec}}
ifelse executeonly
} if
{gsave currentpoint newpath moveto} executeonly
{currentpoint grestore gsave currentpoint newpath moveto}
executeonly
{systemdict /internaldict known not
{pop 3}
{1183615869 systemdict /internaldict get exec
dup /startlock known
{/startlock get exec}
{dup /strtlck known
{/strtlck get exec}
{pop 3}
ifelse}
ifelse}
ifelse
} executeonly
] noaccess def
```

Appendix C: NormalizeDesignVector Example

The **NormalizeDesignVector** procedure is used by the **findfont** procedure defined in a multiple master font to convert the design coordinates in a font name to normalized values. The results are left on the stack for use by the **ConvertDesignVector** procedure. The **NormalizeDesignVector** procedure must be configured for the number of axes and master designs contained in the specific font it is used in.

The following sample procedure is from the Minion™ multiple master font and is configured for Minion's 3 axes and 8 master designs:

```
/NormalizeDesignVector {
3 2 roll 345 sub 275 div
3 2 roll 450 sub 150 div
3 2 roll dup 11 le { dup 8 le { 6 sub 5.71429 div }
{ 1 sub 20 div } ifelse }
{ dup 18 le { -3 sub 28 div } { -144 sub 216 div } ifelse } ifelse }
bind def
```

This procedure expects the design coordinates for a font instance to be on the stack (as shown in section 3.8) and calculates the normalized value of the coordinate. For example, a weight axis value of 530 design coordinate units is 185/275 = 0.6727 units when normalized for an axis ranging from 345 to 620 units (total dynamic range is 275 units).

The third axis, *Optical Size*, has a **BlendDesignMap** value of

[6 0][8 0.35][11 0.50][18 0.75][72 1]

which specifies four piecewise linear segments which define the mapping from design to blend coordinates. The above code checks which segment the design coordinate corresponds to and calculates the normalized coordinate from the equation for the appropriate line segment.

Appendix D: ConvertDesignVector Example

The **ConvertDesignVector** procedure is used by the **findfont** procedure defined in a multiple master font to convert the normalized coordinates (left on the stack by the **NormalizeDesignVector** procedure) to **WeightVector** array values. The **WeightVector** values are left on the stack for use by the **makeblendedfont** procedure. The **ConvertDesignVector** procedure must be configured for the number of axes and master designs contained in the specific font it is used in.

For example, the **ConvertDesignVector** procedure, as configured for the MyriadMM font's 2 axes and 4 master designs, is:

```
/ConvertDesignVector {
 1 2 index sub 1 2 index sub mul 3 1 roll
 1 index 1 2 index sub mul 3 1 roll
 1 2 index sub 1 index mul 3 1 roll
 1 index 1 index mul 3 1 roll
 pop pop
} bind def
```

This code expects the normalized blend coordinates on the stack and calculates the **WeightVector** values which specify the weighting for each master design for the particular font instance. This calculation obeys the rules for the simple linear weighting expressed in section 3.9. For the Myriad multilple master font, the calculations are:

$$V_1 = (1–BC_{A1})\ (1–BC_{A2})$$
$$V_2 = (BC_{A1})(1–BC_{A2})$$
$$V_3 = (1–BC_{A1})\ (BC_{A2})$$
$$V_4 = (BC_{A1})\ (BC_{A2})$$

where V_i is the ith value of the **WeightVector** array, and BC_j is the normalized blend coordinate for the jth axis.

Appendix E: Changes Since Version 1.0 of the Type 1 Font Format Specification

The Type 1 font format was originally published as version 1.0 by Adobe Systems, in 1990. Version 1.1 was subsequently published by Addison Wesley in 1991. Initial additions to the specification were published as Adobe Technical Specification #5047, “Updates to the Type 1 Font Format,” which is superseded by this document. The following information, along with the contents of the main section of this document, identifies all information added since the original version 1.0 specification.

Changes Made Since 15 January 1994 Version of this Supplement

Section 2.2: a statement was added that Counter Control hints must immediately follow the **hsbw** or **sbw** operator.

Changes Made in Adobe Type 1 Font Format, Version 1.1 (published by Addison Wesley, 1991)

- The default values are clearly documented for the following entries in the **Private** dictionary: **BlueScale** (0.039625, equivalent to 10 points at 300 dpi; in section 5.6 of version 1.0), **BlueShift** (7 character space units; in section 5.7 of version 1.0), **BlueFuzz** (1 character space unit; in section 5.8 of version 1.0), and **ExpansionFactor** (0.06, see section 2.5 in this document).

- **ExpansionFactor** is a new (optional) entry to the **Private** dictionary, which provides a font level hint useful for intelligent rendering of complex glyphs with more stems than the usual Latin font. Examples would include Chinese and Japanese language fonts, as well as bar code and logo fonts. See section 2.5 in this document.

- A warning was added to the description of the **closepath** operator (section 6.4 of version 1.1) about using **closepath** to form a subpath section intended to be zero length. If the subpath section is intended to be zero length but is not, the **closepath** operator might cause a “spike” (if the subpath doubles back onto itself) in the path, of zero width, that might produce unexpected results.

- Regarding compatibility with Adobe Type Manager software (in section 10.3 of version 1.0), version 1.1 explains that the parser skips to the first **dup** token after **Encoding** to find the first character encoding assignment.

- The PostScript language program defining the Flex procedure has been modified to protect against trying to put the $FlxDict into **internaldict** if **internaldict** is full. The old code could lead to **dictfull** errors out of **show** in certain unlikely circumstances. The new code puts the $FlxDict in **userdict** if **internaldict** is full. (The new code is given in Appendix C).

Index

B

Blend dictionary 32
BlendAxisTypes 23
BlendDesignMap 24, 27
BlendDesignPositions 23

C

ConvertDesignVector procedure 27
Counter Control
  example 13
  ExpansionFactor 12
  groups 8
  groups, definition 12
  OtherSubrs 9
  prioritizing 12
Counter Control hint mechanism 7

E

ExpansionFactor 12

F

FontBBox 22
FontInfo 15, 18, 21
FontMatrix 22

H

hex code error 33

L

LanguageGroup 8, 9

M

multiple master 14–32
  ATM compatibility 32
  blend coordinates 16
  Blend dictionary
    FontInfo 19
    Private 18
  design axes 14
  design coordinates 15
  design space 15–17
  findfont procedure 26
  font dictionaries 18
  font instance specification 15
  keywords 23–27
  makeblendedfont 25, 35–37
  master designs 14
  OtherSubrs 28–31
  Private dictionary 18
  sample font program explanation 19–23
  Subrs 28–29

N

NormalizeDesignVector procedure 27

O

OtherSubrs
  code listing for Flex and Hint Substitution 39

P

Private dictionary 8, 32

R

RndStemUp 8, 9

S

seac 33
stack limit considerations for Counter
Control 11

T

Type 1 font format
changes
BlueFuzz 47
BlueScale 47
BlueShift 47
closepath 47
compatibility with ATM 48
ExpansionFactor 47
Flex 48

W

WeightVector 18