TrueType 1.0 Font Files

Technical Specification
Revision 1.66
August 1995

M *1 Microsoft Way* *Redmond, Washington 98052*

Revision History

Revision	Date	Comment
1.66	9 August 1995	Updates to bit settings for ulCodePageRange and ulUnicodeRange fields in the OS/2 table
1.65	19 July 1995	Reprint of rev 1.65, typos corrected, VendID table updated, note added to loca about long-aligning local offsets
1.65	18 January 1995	Added tables: gasp, new OS/2, vhea, vmtx, EBDT, EBLC, EBSC; TTC's; minor corrections and clarifications throughout; reorganized spec.
1.50	21 January 1994	Added smart scan control, Grayscale support, and minor corrections throughout
1.02	19 May 1993	Update to OS/2 table fsType field description
1.01	19 January 1993	Minor corrections throughout

1.00	9 June 1992	Original release

Table of Contents

Preface

Welcome to TrueType 95! This is Microsoft's first major update and extension of the TrueType outline font format. This document describes the state of TrueType as it exists in Windows NT 3.5/3.51 and Windows 95.

The TrueType Font Format specification version 1.6 covers all the improvements and extensions made to the 1.6.x rasterizers included in Windows NT 3.5/3.51 and Windows 95:

- 32-bit rasterization, eliminating previous problems with complex fonts
- Faster, optimized code
- Gray-scale rasterization, for easy-to-read anti-aliased text on screen

  (Gray-scale rasterization is not exposed through API's on Windows NT 3.5/3.51)
- Embedded bitmaps for efficient display of complex characters at sizes where hinting is difficult
- Vertical metrics, for scripts written vertically
- Expanded character sets and better code page classification for international uses
- TrueType Collections (TTCs), for efficient sharing of data among related fonts

  (TTCs are implemented only on some versions of Windows 95. The other features are available across all versions of Windows NT 3.5/3.51 and Windows 95.)

Microsoft has defined a number of new tables for TrueType font files (TTFs) to support these features:

- EBLC, EBDT, and EBSC for embedded bitmaps
- gasp for control of rasterizer and gray-scale features
- vhea and vmtx for vertical metrics
- An extended OS/2 for script classification information

We have also improved the organization of this document. As most font developers are chiefly concerned with producing TrueType font files (TTFs), we have moved the description of the file format to the front of the book, and organized the tables in alphabetical order. A new unified chapter presents a comparative list of the chief character set standards relevant to TrueType developers, including the Windows Glyph List 4 (WGL4), the new standard for the core fonts used in Windows 95. And throughout, we have tried to correct and clarify some of the more confusing aspects of TrueType development.

Although Microsoft has created new ways of extending the functionality of the TrueType font format, none of these changes need affect the end user. Both the rasterizers and the file formats are completely compatible with past, present, and future Microsoft software. Existing TrueType fonts will continue to work on current and future Microsoft platforms. And new fonts created to take advantage of the new tables listed above will continue to work (albeit without the new features) on existing platforms, such as Microsoft Windows 3.1.

TrueType 95 is just the first step in Microsoft's program to address new and changing uses for digital type, called TrueType Open. Among the extensions already planned are an expansion of TrueType to support high-quality international typography.

TrueType Open is an open format (Microsoft documents will describe everything you need to know); but it's also “Open” because its features are not dependent on a particular platform or a particular piece of operating system software.

Stay tuned for more information!

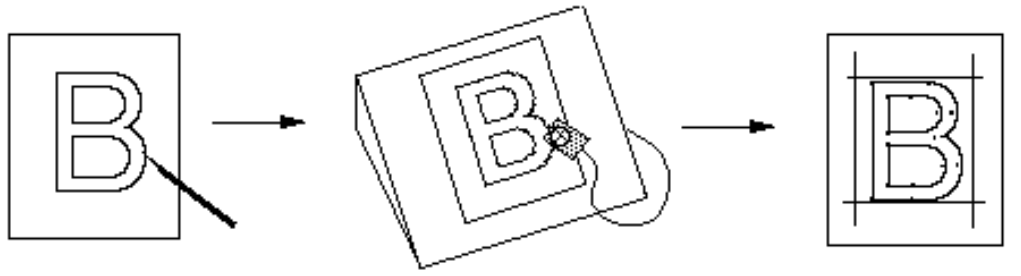
Microsoft Typography

TrueType Fundamentals

This chapter introduces the basic concepts needed to create and instruct a TrueType font. It begins with an overview of the steps involved in taking a design from paper to the creation of a bitmap that can be sent to an output device and follows with a closer look at each of the steps in the process.

From design to font file

A TrueType font can originate as a new design drawn on paper or created on a computer screen. TrueType fonts can also be obtained by converting fonts from other formats. Whatever the case, it is necessary to create a TrueType font file that, among other things, describes each glyph in the font as an outline in the TrueType format.



Original Master

Digitized drawing

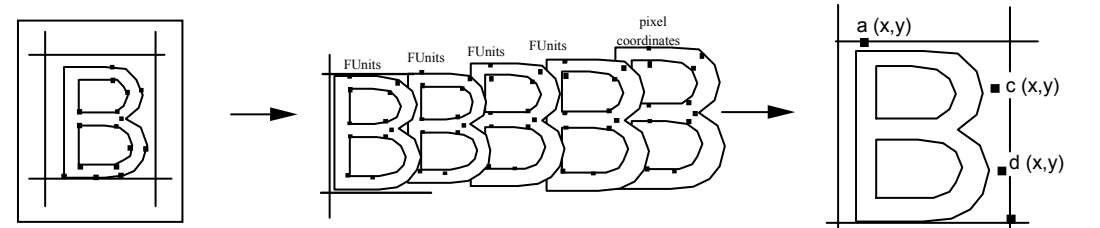
Digitized outline with FUnit coordinates in font file

From Font File to Paper

This section describes the process that allows glyphs from a TrueType font file to be displayed on raster devices.

First, the outline stored in the font file is scaled to the requested size. Once scaled, the points that make up the outline are no longer recorded in the FUnits used to describe the original outline, but have become device-specific pixel coordinates.

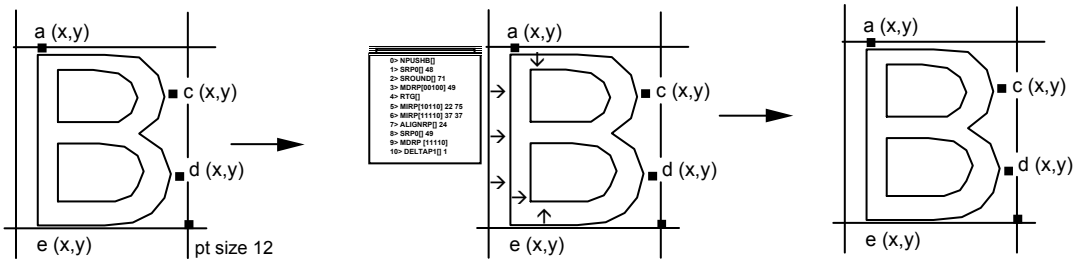
Next, the instructions associated with this glyph are carried out by the interpreter. The result of carrying out the instructions is a grid-fitted outline for the requested glyph. This outline is then scan converted to produce a bitmap that can be rendered on the target device.



Digitize outline with FUnit coordinates in TrueType font file

Scaler converts FUnits to pixel coordinates and scales outline to size requested by application

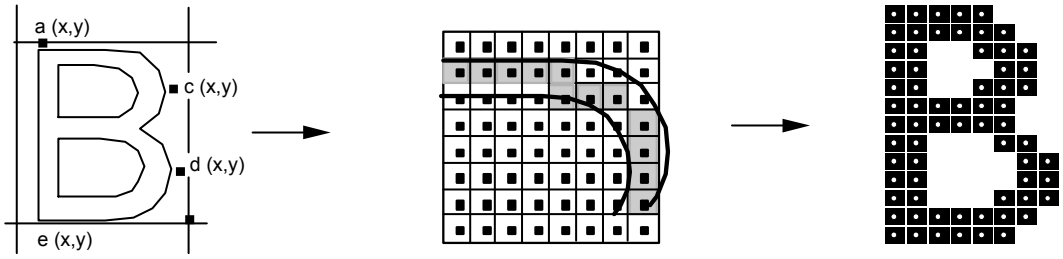
Outline "sized" to new grid



Scaled outline with pixel coordinates

Interpreter executes instructions associated with glyph "B" and gridfits

Grid-fitted outline



Grid-fitted outline

Scan converter decides which pixels to turn on

Bitmap is rendered on raster device

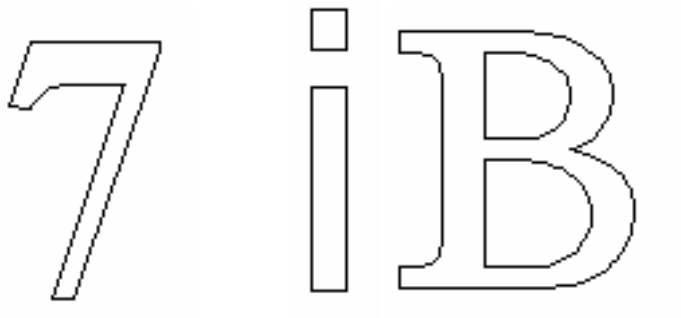
Digitizing a design

This section describes the coordinate system used to establish the locations of the points that define a glyph outline. It also documents the placement of glyphs with respect to the coordinate axes.

Outlines

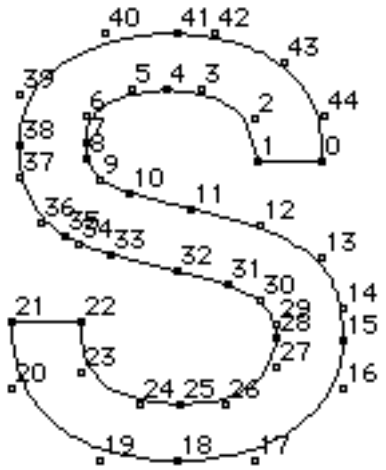
In a TrueType font, glyph shapes are described by their outlines. A glyph outline consists of a series of contours. A simple glyph may have only one contour. More complex glyphs can have two or more contours. Composite glyphs can be constructed by combining two or more simpler glyphs. Certain control characters that have no visible manifestation will map to the glyph with no contours.

Figure 1-1 Glyphs with one, two, three contours respectively



Contours are composed of straight lines and curves. Curves are defined by a series of points that describe second order Bezier-splines. The TrueType Bezier-spline format uses two types of points to define curves, those that are *on* the curve and those that are *off* the curve. Any combination of off and on curve points is acceptable when defining a curve. Straight lines are defined by two consecutive on curve points.

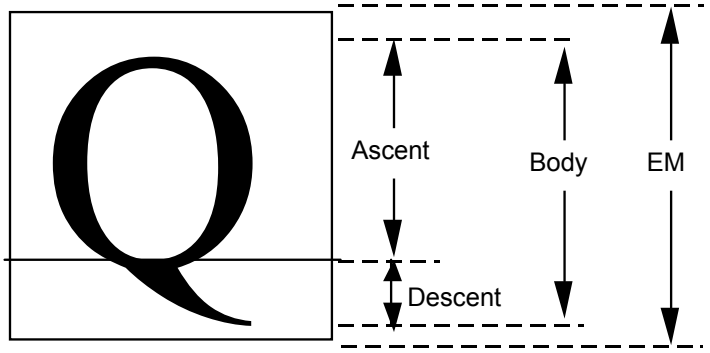
Figure 1-2 A glyph description consisting of a series of on and off curve points



The points that make up a curve must be numbered in consecutive order. It makes a difference whether the order is increasing or decreasing in determining the fill pattern of the shapes that make up the glyph. The direction of the curves has to be such that, if the curve is followed in the direction of increasing point numbers, the black space (the filled area) will always be to the right.

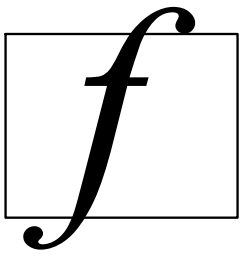
FUnits and the em square

In a TrueType font file point locations are described in font units, or FUnits. An FUnit is the smallest measurable unit in the em square, an imaginary square that is used to size and align glyphs. The dimensions of the em square typically are those of the full body height of a font plus some extra spacing to prevent lines of text from colliding when typeset without extra leading.



While in the days of metal type, glyphs could not extend beyond the em square, digital typefaces are not so constrained. The em square may be made large enough to completely contain all glyphs, including accented glyphs. Or, if it proves convenient, portions of glyphs may extend outside the em square. TrueType fonts can handle either approach so the choice is that of the font manufacturer.

Figure 1-3 A character that extends outside of the em square



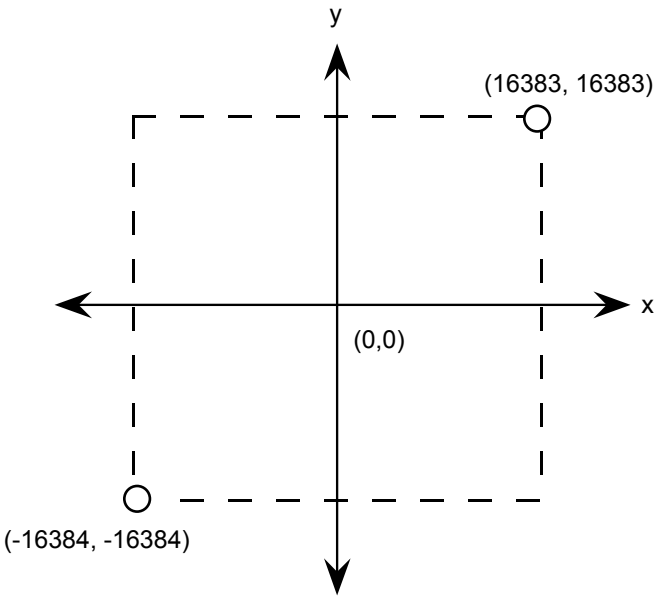
The em square defines a two-dimensional coordinate grid whose *x*-axis describes movement in a horizontal direction and whose *y*-axis describes movement in a vertical direction. This is discussed in more detail in the following section.

FUnits and the grid

A key decision in digitizing a font is determining the resolution at which the points that make up glyph outlines will be described. The points represent locations in a grid whose smallest addressable unit is known as an FUnit or font Unit. The grid is a two-dimensional coordinate system whose *x*-axis describes movement in a horizontal direction and whose *y*-axis describes movement in a vertical direction. The grid origin has the coordinates (0,0). The grid is not an infinite plane. Each point must be within the range -16384 and +16383 FUnits. Depending upon the resolution chosen, the range of addressable grid locations will be smaller.

The choice of the granularity of the coordinate grid—that is, number of units per em (upem)—is made by the font manufacturer. Outline scaling will be fastest if units per em is chosen to be a power of 2, such as 2048.

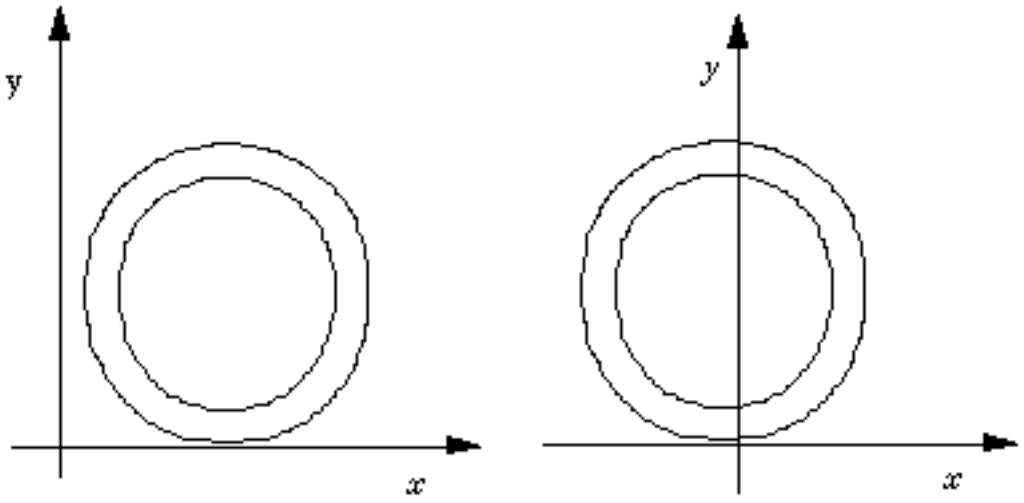
Figure 1-4 The coordinate system



The origin of the em square need not have any consistent relationship to the glyph outlines. In practice, however, applications depend upon the existence of some convention for the placement of glyphs for a given font. For Roman fonts, which are intended to be laid out horizontally, a *y*-coordinate value of 0 typically is assumed to correspond to the baseline of the font. No particular meaning is assigned to an *x*-coordinate of 0 but manufacturers may improve the performance of applications by choosing a standard meaning for the *x*-origin.

For example, you might place a glyph so that its aesthetic center is at the *x*-coordinate value of 0. That is, a set of glyphs so designed when placed in a column such that their *x*-coordinate values of 0 are coincident will appear to be nicely centered. This option would be used for Kanji or any fonts that are typeset vertically. Another alternative is to place each glyph so that its leftmost extreme outline point has an *x*-value equal to the left-side-bearing of the glyph. Fonts created in this way may allow some applications to print more quickly to PostScript printers.

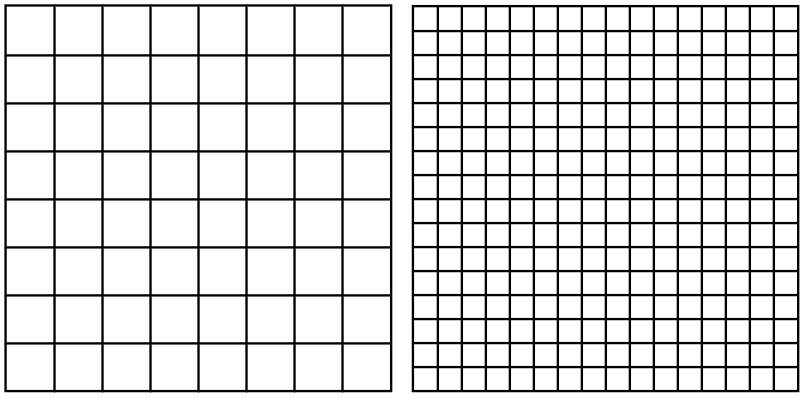
Figure 1-5 Two possible choices for the glyph origin in a Roman font. In the first case (left) the left side bearing is *x*-zero. In the second (right), the aesthetic center of the character is *x*-zero



Non-Roman fonts may wish to use other conventions for the meaning of the *x*-origin and *y*-origin. For best results with high-lighting and carets, the body of the character should be roughly centered within the advance width. For example, a symmetrical character would have equal left and right side bearings.

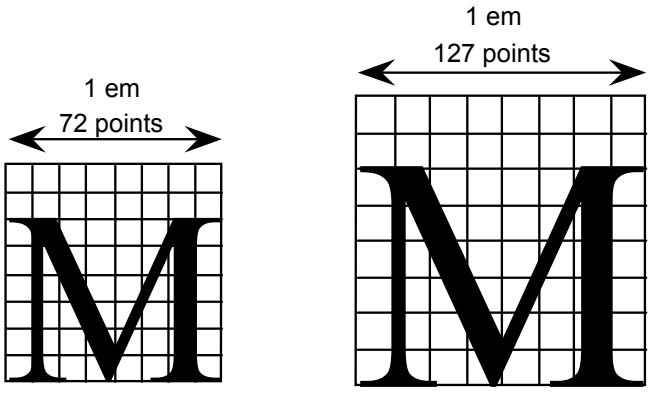
The granularity of the em square is determined by the number of FUnits per em, or more simply units per em . The em square as divided into FUnits defines a coordinate system with one unit equaling an FUnit. All points defined in this coordinate system must have integral locations. The greater the number of units per em, the greater the precision available in addressing locations within the em square.

Figure 1-6 Two em squares, 8 units per em (left), 16 units per em (right)



FUnits are relative units because they vary in size as the size of the em square changes. The number of units per em remains constant for a given font regardless of the point size. The number of points per em, however, will vary with the point size of a glyph. An em square is exactly 9 points high when a glyph is displayed at 9 points, exactly 10 points high when the font is displayed at 10 point, and so on. Since the number of units per em does not vary with the point size at which the font is displayed, the absolute size of an FUnit varies as the point size varies.

Figure 1-7 72 point M and 127 point M and their em squares. Upem equals 8 in both cases.



Because FUnits are relative to the em square, a given location on a glyph will have the same coordinate location in FUnits regardless of the point size at which the font is rendered. This is convenient because it makes it possible to instruct outline points once considering only the original outline and have the changes apply to the glyph at whatever size and resolution it is ultimately rendered.

Scaling a glyph

This section describes how glyph outlines are scaled from the master size stored in the font file to the size requested by an application.

Device space

Whatever the resolution of the em square used to define a glyph outline, before that glyph can be displayed it must be scaled to reflect the size, transformation and the characteristics of the output device on which it is to be displayed. The scaled outline must describe the character outline in units that reflect an absolute rather than relative system of measurement. In this case the points that make up a glyph outline are described in terms of pixels.

Intuitively, pixels are the actual output bits that will appear on screen or printer. To allow for greater precision in managing outlines, TrueType describes pixel coordinates to the nearest sixty-fourth of a pixel.

Converting FUnits to pixels

Values in the em square are converted to values in the pixel coordinate system by multiplying them by a scale. This scale is:

$$\text{pointSize} * \frac{\text{resolution}}{72 \text{ points per inch} * \text{units\_per\_em}}$$

where pointSize is the size at which the glyph is to be displayed, and resolution is the resolution of the output device. The 72 in the denominator reflects the number of points per inch.

For example, assume that a glyph feature is 550 FUnits in length on a 72 dpi screen at 18 point. There are 2048 units per em. The following calculation reveals that the feature is 4.83 pixels long.

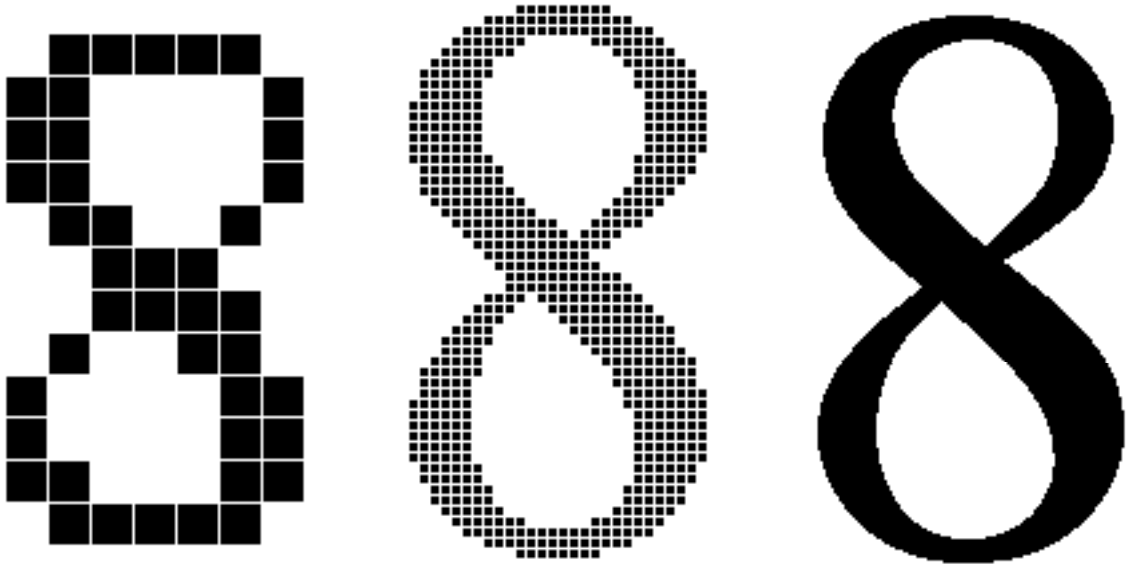
$$550 * \frac{18 * 72}{72 * 2048} = 4.83$$

Display device characteristics

The resolution of any particular display device is specified by the number of dots or pixels per inch (dpi) that are displayed. For example, a VGA under OS/2 and Windows is treated as a 96 dpi device, and most laser printers have a resolution of 300 dpi. Some devices, such as an EGA, have different resolution in the horizontal and vertical directions (i.e. non-square pixels); in the case of the EGA this resolution is 96 x 72. In such cases, horizontal dots per inch must be distinguished from vertical dots per inch.

The number of pixels per em is dependent on the resolution of the output device. An 18 point character will have 18 pixels per em on a 72 dpi device. Change the resolution to 300 dpi and it has 75 pixels per em, or change to 1200 dpi and it has 300 pixels per em.

Figure 1-8 18 point figure 8 at 72 dpi, 300 dpi and 1200 dpi



Displaying type on a particular device at a specific point size yields an effective resolution measured in pixels per em (ppem). The formula for calculating pixels per em is:

$$\begin{aligned} \text{ppem} &= \text{pointSize} * \frac{\text{dpi}}{72} \\ &= (\text{pixels per inch}) * (\text{inches per pica point}) * (\text{pica points per em}) \\ &= \text{dpi} * \frac{1}{72} * \text{pointSize} \end{aligned}$$

On a 300 dpi laser printer, a 12 point glyph would have 12*300/72 or 50 ppem. On a typesetter with 2400 dpi, it would have 12*2400/72 or 400 ppem. On a VGA, a 12 point glyph would have 12*96/72 or 16 ppem. Similarly, the ppem for a 12 point character on a 72 dpi device would be 12*72/72, or 12. This last calculation points to a useful rule of thumb: on any 72 dpi device, points and pixels per em are equal. Note, however, that in traditional typography an inch contains 72.2752 points (rather than 72); that is, one point equals .013836 inches.

If you know the ppem, the formula to convert between FUnits and pixel space coordinates is:

$$\text{pixel\_coordinate} = \text{em\_coordinate} * \frac{\text{ppem}}{\text{upem}}$$

An em_coordinate position of (1024, 0) would yield a device_pixels coordinate of (6, 0), given 2048 units per em and 12 pixels per em.

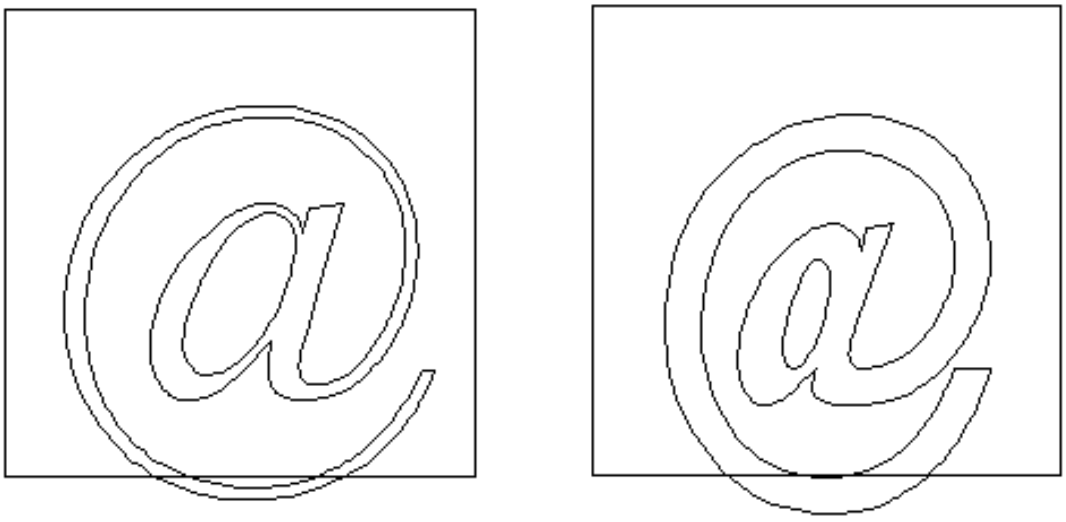
Grid-fitting a glyph outline

The fundamental task of instructing a glyph is one of identifying the critical characteristics of the original design and using instructions to ensure that those characteristics will be preserved when the glyph is rendered at different sizes on different devices. Consistent stem weights, consistent color, even spacing, and the elimination of pixel dropouts are common goals.

To accomplish these goals, it is necessary to ensure that the correct pixels are turned on when a glyph is rasterized. It is the pixels that are turned on that create the bitmap image of the glyph. Since it is the shape of the glyph outline that determines which pixels will make up the bitmap image of that character at a given size, it is sometimes necessary to change or distort the original outline description to produce a high quality image. This distortion of the outline is known as grid-fitting.

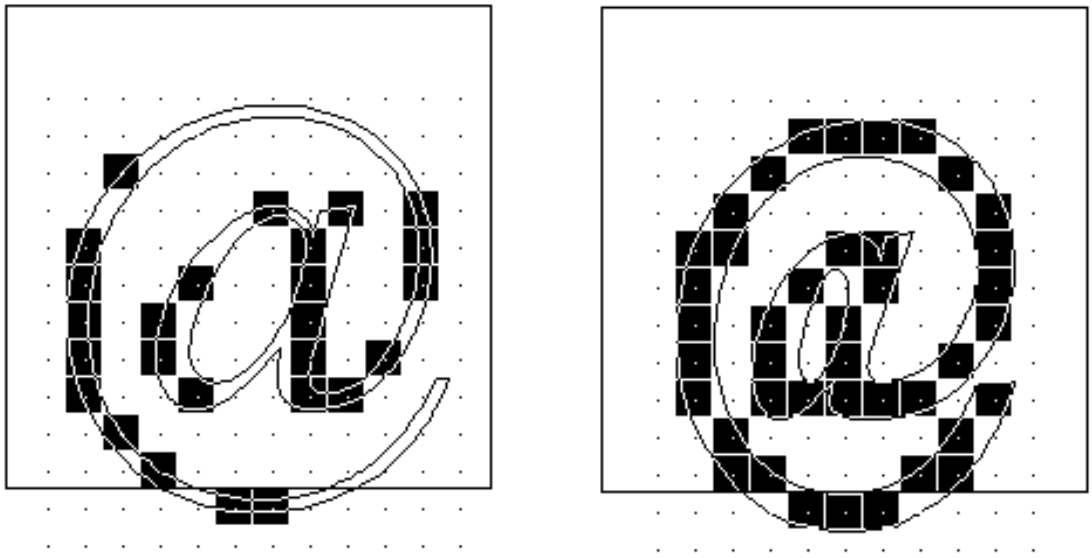
The figure below illustrates how grid-fitting a character distorts the outline found in the original design.

Figure 1-9 12 point outlines ungrid-fitted (left) and grid-fitted (right)



As the illustration above suggests, the grid-fitting employed in TrueType goes well beyond aligning a glyph's left side bearing to the pixel grid. This sophisticated grid-fitting is guided by instructions. The beneficial effects of grid-fitting are illustrated in the next figure.

Figure 1-10 12 point outlines and bitmap ungrid-fitted (left) and grid-fitted (right)



Grid-fitting is the process of stretching the outline of a glyph according to the instructions associated with it. Once a glyph is grid-fitted, the point numbers will be unchanged but the actual location of that point in the coordinate grid may have shifted. That is, the coordinates for a given point number will, very likely, have changed after a glyph is grid-fitted.

What are instructions?

The TrueType instruction set provides a large number of commands designed to allow designers to specify how character features should be rendered. Instructions are the mechanism by which the design of a character is preserved when it is scaled. In other words, instructions control the way in which a glyph outline will be grid-fitted for a particular size or device.

Instructing a font will reshape the outline for a given glyph at a specific size on a given target device in such a way that the correct pixels are included within its outline. Reshaping the outline means moving outline points. Points that have been acted upon by an instruction are said to have been touched. Note that a point need not actually be moved to be touched. It must simply be acted upon by an instruction. (See MDAP, chapter 3.)

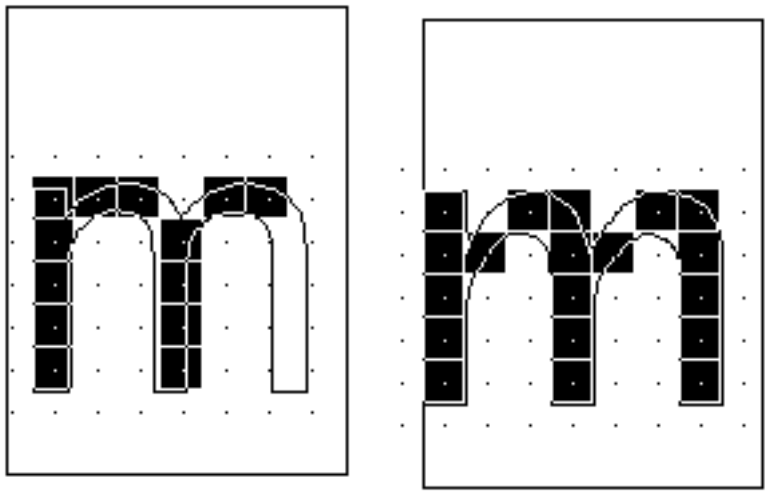
TrueType fonts can be used with or without instructions. Uninstructed fonts will generally produce good quality results at sufficiently high resolutions and point sizes. The range of sizes over which an uninstructed font will produce good quality results depends not only on the output device resolution and point size of the character but also on the particular font design. The intended use of the font can also be a factor in determining whether or not a particular font should be instructed. For most fonts, if legibility of small point sizes on low resolution devices is important, adding instructions will be critical.

Instructing a font is a process that involves analyzing the key elements of a glyph's design and using the TrueType instruction set to ensure that they are preserved. The instructions are flexible enough to allow characteristics that are roughly the same to be "homogenized" at small sizes while allowing the full flavor of the original design to emerge at sizes where there are sufficiently many pixels.

How does the TrueType interpreter know the manner in which an outline should be distorted to produce a desirable result? This information is contained in instructions attached to each character in the font. Instructions specify aspects of a character's design that are to be preserved as it is scaled. For example, using instructions it is possible to control the height of an individual character or of all the characters in a font. You can also preserve the relationship between design elements within a character thereby ensuring, for example, that the widths of the three vertical stems in the lower case m will not differ dramatically at small sizes.

The following figure illustrates how changing a glyph's outline at a specific size will yield a superior result. They show that an uninstructed 9 point Arial lowercase m suffers the loss of a stem due to chance effects in the relationship of stems to pixel centers. In the second glyph, instructions have aligned the stems to the grid so that the glyph suffers no similar loss.

Figure 1-11 9 point Arial m—uninstructed (left), instructed (right)



The TrueType interpreter

This section describes the actions of the TrueType interpreter. It is the interpreter, as the name suggests, that “interprets” or carries out the instructions.

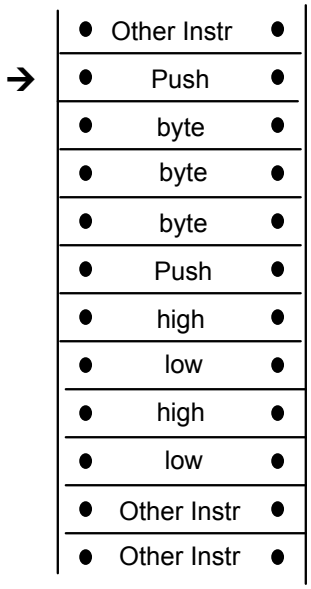
More concretely, the interpreter processes a stream or sequence of instructions. Typically these instructions take their arguments from the interpreter stack and place their results on that stack. The only exceptions are a small number of instructions that are used to push data onto the interpreter stack. These instructions take their arguments from the instruction stream.

All of the interpreter’s actions are carried on in the context of the Graphics State, a set of variables whose values guide the actions of the interpreter and determine the exact effect of a particular instruction.

The interpreter’s actions can be summarized as follows:

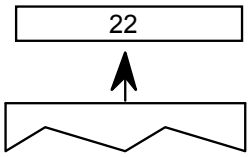
1. The interpreter fetches an instruction from the instruction stream, an ordered sequence of instruction opcodes and data. Opcodes are 1-byte in size. Data can consist of a single byte or two bytes (a word). If an instruction takes words from the instruction stream it will create those words by putting together two bytes. The high byte appears first in the instruction stream and the low byte appears second.

   The following instruction stream is depicted as it will be shown in the examples that follow. Note that the pointer indicates the next instruction to be executed.

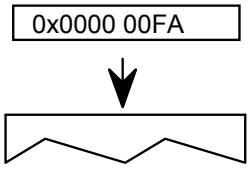


2. The instruction is executed

- If it is a push instruction it will take its arguments from the instruction stream.
- Any other instruction will pop any data it needs from the stack. A pop is illustrated below.



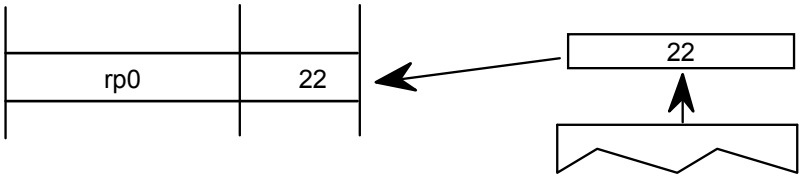
- Any data the instruction produces is pushed onto the interpreter stack. A push is illustrated below.



As the previous discussion indicates, the interpreter stack is a LIFO or last in first out data structure. An instruction takes any data it needs from the last item placed on the stack. The action of removing the top item from the stack is commonly termed a *pop*. When an instruction produces some result it *pushes* that result to the top of the stack where it is potential input to the next instruction.

The instruction set includes a full range of operators for manipulating the stack including operators for pushing items onto the stack, popping items from the stack, clearing the stack, duplicating stack elements and so forth.

- The effect of execution depends on the values of the variables that make up the Graphics State.
- The instruction may modify one or more Graphics State variables. In the illustration shown, the Graphics State variable rp0 is updated using a value taken from the interpreter stack.



3. The process is repeated until there are no further instructions to be executed.

Using instructions

Instructions can appear in a number of places in the font file tables that make up a TrueType font. They can appear as part of the Font Program, the CVT Program, or as glyph data. Instructions appearing in the first two apply to the font as a whole. Those found in glyph data ('glyf') apply to individual glyphs within a font.

The Font Program

The Font Program consists of a set of instructions that is executed once, the first time a font is accessed by an application. It is used to create function definitions (FDEFs) and instruction definitions (IDEFs). Functions and instructions defined in the Font Program can be used elsewhere in the font file.

The CVT Program

The CVT Program is a sequence of TrueType instructions executed every time the point size or transformation change. It is used to make font wide changes rather than to manage individual glyphs. The CVT Program is used to establish the values in the Control Value Table.

The purpose of the Control Value Table or CVT is to simplify the task of maintaining consistency when instructing a font. It is a numbered list of values that can be referenced by either of two indirect instructions (MIRP and MIAP). CVT entries can be used to store values that need to be the same across a number of glyphs in a font. For example an instruction might refer to a CVT entry whose purpose is to regularize stem weights across a font.

Figure 1-12 Some sample CVT entries

Entry #	Value	Description
0	0	upper and lower case flat base (base line)
1	-39	upper case round base
2	-35	lower case round base
3	-33	figure round base
4	1082	x-height flat
5	1114	x-height round overlap
6	1493	flat cap
7	1522	round cap
8	1463	numbers flat
9	1491	numbers round top
10	1493	flat ascender
11	1514	round ascender
12	157	x stem weight
13	127	y stem weight
14	57	serif
15	83	space between the dot and the i

Instructions that refer to values in the CVT are called indirect instructions as opposed to the direct instructions which take their values from the glyph outline.

As part of the TrueType font file, the values in the CVT are expressed in FUnits. When the outlines are converted from FUnits to pixel units, values in the CVT are also converted.

When writing to the CVT you may use a value that is in the glyph coordinate system (using WCVTP) or you can use a value that is in the original FUnits (using WCVTF). The interpreter will scale all values appropriately. Values read from the CVT are always in pixels (F26Dot6).

The Storage Area

The interpreter also maintains a Storage Area consisting of a portion of memory that can be used for temporary storage of data from the interpreter stack. Instructions exist that make it possible to read the values of stored data and to write new values to storage. Storage locations range from 0 to n-1 where n is the value established in the maxStorage entry in the maxProfile table of the font file. Values are 32 bit numbers.

Figure 1-13 Some storage area entries

Address	Value
0	343
1	241
2	-27
3	4654
4	125
5	11

The Graphics State

The Graphics State consists of a table of variables and their values. All instructions act within the context of the Graphics State. Graphics State variables have default values as specified in Appendix B, “Graphics State Summary”. Their values can be determined or changed using instructions.

The Graphics State establishes the context within which all glyphs are interpreted. All Graphics State variables have a default value. Some of these values can be changed in the CVT Program if desired. Whatever the default value, it will be reestablished at the start of interpretation of any glyph. In other words, the Graphics State has no inter-glyph memory. Changing the value of a Graphics State variable while processing an individual glyph will result in a change that remains in effect only for that glyph.

The scan converter

The TrueType scan converter takes an outline description of a glyph and produces a bitmap image for that glyph.

The TrueType scan converter offers two modes. In the first mode, the scan converter uses a simple algorithm for determining which pixels are part of that glyph. The rules can be stated as follows:

Rule 1

If a pixel's center falls within the glyph outline, that pixel is turned on and becomes part of that glyph.

Rule 2

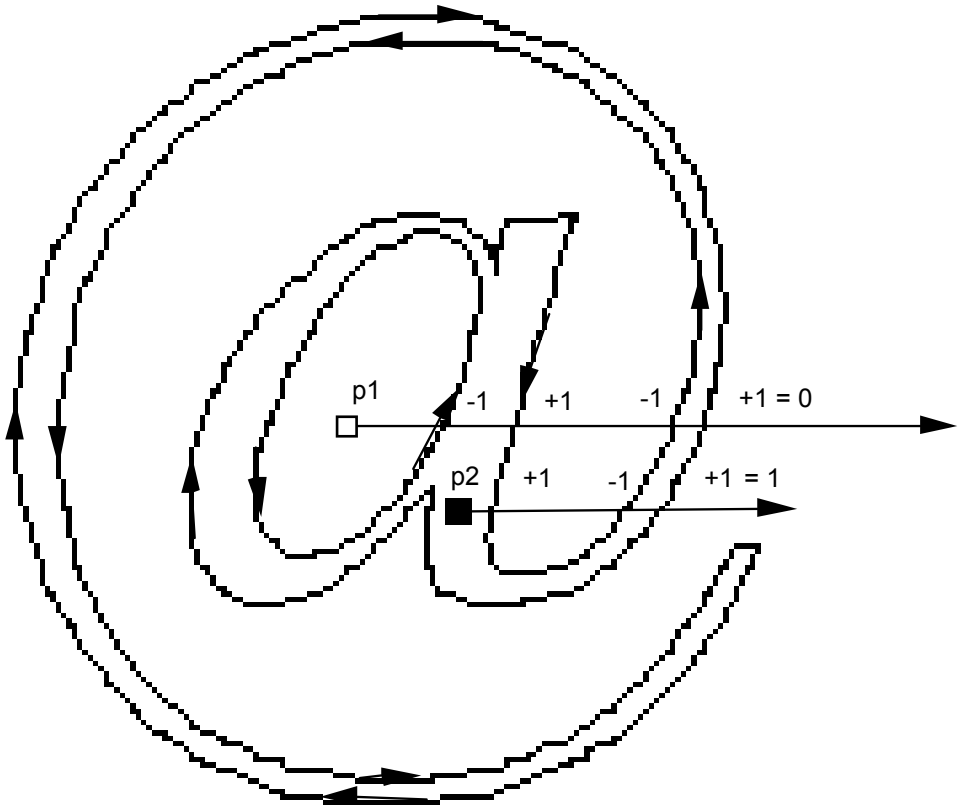
If a contour falls exactly on a pixel's center, that pixel is turned on.

A point is considered to be an interior point of a glyph if it has a non-zero winding number. The winding number is itself determined by drawing a ray from the point in question toward infinity. (The direction in which the ray points in unimportant.) Starting with a count of zero, we subtract one each time a glyph contour crosses the ray from right to left or bottom to top. Such a crossing is termed an on transition. We add one each time a contour of the glyph crossed the ray from left to right or top to bottom. Such a crossing is termed an off transition. If the final count is non-zero, the point is an interior point.

The direction of a contour can be determined by looking at the point numbers. The direction is always from lower point number toward higher point number.

The illustration that follows demonstrates the use of winding numbers in determining whether a point is inside a glyph. The point p1 undergoes a sequence of four transitions (*on* transition, *off* transition, *on* transition, *off* transition). Since the sequence is even, the winding number is zero and the point is not inside the glyph. The second point, p2, undergoes an *off* transition followed by an *on* transition followed by an *off* transition yielding a winding number of +1. The point is in the interior of the glyph.

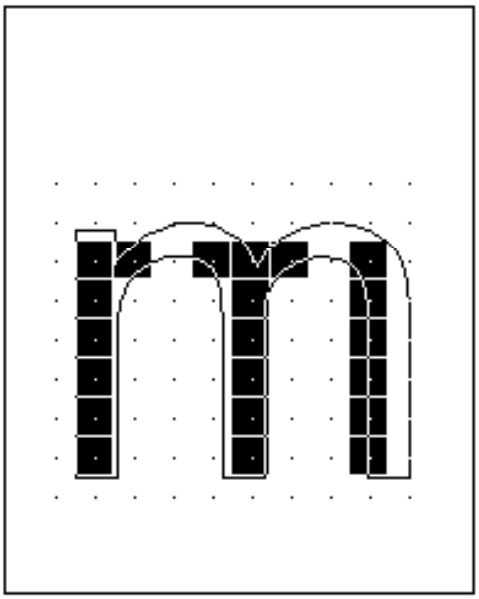
Figure 1-14 Determining the winding number of a point



What is a dropout?

A dropout occurs whenever there is a connected region of a glyph interior that contains two black pixels that cannot be connected by a straight line that only passes through black pixels.

Figure 1-15 The letter m with two dropouts



Preventing dropouts

The TrueType instructions are designed to allow you to gridfit a glyph so that the desired pixels will be turned on by the simple scan converter regardless of the point size or the transformation used. It may prove difficult to foresee all possible transformations that a glyph might undergo. It is therefore difficult to instruct a glyph to ensure that the proper grid-fitting distortion of the outline will take place for every desired transformation. This problem is especially difficult for very small numbers of pixels per em and for complex typefaces. In these situations, some renditions of a glyph may contain dropouts.

It is possible to test for potential dropouts by looking at an imaginary line segment connecting two adjacent pixel centers. If this line segment is intersected by both an on-Transition contour and an off-Transition contour, a potential dropout condition exists. The potential dropout only becomes an actual dropout if the two contour lines continue on in both directions to cut other line segments between adjacent pixel centers. If the two contours join together immediately after crossing a scan line (forming a stub), a dropout does not occur, although a stem of the glyph may become shorter than desired.

To prevent dropouts, type manufacturers can choose to have the scan converter use two additional rules:

Rule 3

If a scan line between two adjacent pixel centers (either vertical or horizontal) is intersected by both an on-Transition contour and an off-Transition contour and neither of the pixels was already turned on by rules 1 and 2, turn on the left-most pixel (horizontal scan line) or the bottom-most pixel (vertical scan line)

Rule 4

Apply Rule 3 only if the two contours continue to intersect other scan lines in both directions. That is do not turn on pixels for 'stubs'. The scanline segments that form a square with the intersected scan line segment are examined to verify that they are intersected by two contours. It is possible that these could be different contours than the ones intersecting the dropout scan line segment. This is very unlikely but may have to be controlled with grid-fitting in some exotic glyphs.

The type manufacturer can choose to use the simple scan converter employing rules 1 and 2 only or may optionally invoke either rule 3 or rule 4. The decision about which scan converter to use can be made on a font wide basis or a different choice can be specified for each glyph. The selection made in the preProgram will be the default for the entire font. A change made to the default in the instructions for an individual glyph will apply only to that glyph.

The TrueType Font File

A TrueType font file contains data, in table format, that comprises an outline font. The rasterizer uses combinations of data from different tables to render the glyph data in the font.

The rasterizer has a much easier time traversing tables if they are padded so that each table begins on a 4-byte boundary. It is highly recommended that all tables be long aligned and padded with zeroes.

Data Types

The following data types are used in the TrueType font file. All TrueType fonts use Motorola-style byte ordering (Big Endian):

Data type	Description
BYTE	8-bit unsigned integer.
CHAR	8-bit signed integer.
USHORT	16-bit unsigned integer.
SHORT	16-bit signed integer.
ULONG	32-bit unsigned integer.
LONG	32-bit signed integer.
FIXED	32-bit signed fixed-point number (16.16)
FUNIT	Smallest measurable distance in the em space.
FWORD	16-bit signed integer (SHORT) that describes a quantity in FUnits.
UFWORD	Unsigned 16-bit integer (USHORT) that describes a quantity in FUnits.
F2DOT14	16-bit signed fixed number with the low 14 bits of fraction (2.14).

Most tables have version numbers, and the version number for the entire font is contained in the Table Directory (see below). Note that there are two different version number types, each with its own numbering scheme. USHORT version numbers always start at zero (0). Fixed version numbers always start at one (1.0 or 0x00010000).

The Fixed point format consists of a signed, 2's complement mantissa and an unsigned fraction. To compute the actual value, take the mantissa and add the fraction. Examples of 2.14 values are:

Decimal Value	Hex Value	Mantissa	Fraction
1.999939	0x7fff	1	16383/16384
1.75	0x7000	1	0/16384
0.000061	0x0001	0	1/16384
0.0	0x0000	0	0/16384
-0.000061	0xffff	-1	16383/16384
-2.0	0x8000	-2	0/16384

The Table Directory

The TrueType font file begins at byte 0 with the Offset Table.

Type	Name	Description
Fixed	sfnt version	0x00010000 for version 1.0.
USHORT	numTables	Number of tables.
USHORT	searchRange	(Maximum power of 2 ≤ numTables) x 16.
USHORT	entrySelector	Log_2(maximum power of 2 ≤ numTables).
USHORT	rangeShift	NumTables x 16-searchRange.

This is followed at byte 12 by the Table Directory entries. Entries in the Table Directory must be sorted in ascending order by tag.

Type	Name	Description
ULONG	tag	4 -byte identifier.
ULONG	checkSum	CheckSum for this table.
ULONG	offset	Offset from beginning of TrueType font file.
ULONG	length	Length of this table.

The Table Directory makes it possible for a given font to contain only those tables it actually needs. As a result there is no standard value for numTables.

Tags are the names given to tables in the TrueType font file. At present, all tag names consist of four characters, though this need not be the case. Names with less than four letters are allowed if followed by the necessary trailing spaces. A list of the currently defined tags follows.

Required Tables

Tag	Name
cmap	character to glyph mapping
glyf	glyph data
head	font header
hhea	horizontal header
hmtx	horizontal metrics
loca	index to location
maxp	maximum profile
name	naming table
post	PostScript information
OS/2	OS/2 and Windows specific metrics

Optional Tables

Tag	Name
cvt	Control Value Table
EBDT	Embedded bitmap data
EBLC	Embedded bitmap location data
EBSC	Embedded bitmap scaling data
fpgm	font program
gasp	grid-fitting and scan conversion procedure (grayscale)
hdmx	horizontal device metrics
kern	kerning
LTSH	Linear threshold table
prep	CVT Program
PCLT	PCL5
VDMX	Vertical Device Metrics table
vhea	Vertical Metrics header
vmtx	Vertical Metrics

Other tables may be defined for other platforms and for future expansion. Note that these tables will not have any effect on the scan converter. Tags for these tables must be registered with Apple Developer Technical Support. Tag names consisting of all lower case letters are reserved for Apple's use. The number 0 is never a valid tag name.

Table checksums are the unsigned sum of the longs of a given table. In C, the following function can be used to determine a checksum:

```
ULONG
CalcTableChecksum(ULONG *Table, ULONG Length)
{
ULONG Sum = 0L;
ULONG *Endptr = Table+((Length+3) & ~3) / sizeof(ULONG);

while (Table < EndPtr)
  Sum += *Table++;
return Sum;
}
```

Note: This function implies that the length of a table must be a multiple of four bytes. While this is not a requirement for the TrueType scaler itself, it is suggested that all tables begin on four byte boundries, and pad any remaining space between tables with zeros. The length of all tables should be recorded in the table directory with their actual length.

Note that the offset in the Table Directory is measured from the start of the TrueType font file.

cmap - Character To Glyph Index Mapping Table

This table defines the mapping of character codes to the glyph index values used in the font. It may contain more than one subtable, in order to support more than one character encoding scheme. Character codes that do not correspond to any glyph in the font should be mapped to glyph index 0. The glyph at this location must be a special glyph representing a missing character.

The table header indicates the character encodings for which subtables are present. Each subtable is in one of four possible formats and begins with a format code indicating the format used.

The platform ID and platform-specific encoding ID are used to specify the subtable; this means that each platform ID/platform-specific encoding ID pair may only appear once in the cmap table. Each subtable can specify a different character encoding. (*See the 'name' table section).* The entries must be sorted first by platform ID and then by platform-specific encoding ID.

When building a Unicode font for Windows, the platform ID should be 3 and the encoding ID should be 1. When building a symbol font for Windows, the platform ID should be 3 and the encoding ID should be 0. When building a font that will be used on the Macintosh, the platform ID should be 1 and the encoding ID should be 0.

All Microsoft Unicode encodings (Platform ID = 3, Encoding ID = 1) must use Format 4 for their 'cmap' subtable. Microsoft **strongly** recommends using a Unicode 'cmap' for all fonts. However, some other encodings that appear in current fonts follow:

Platform ID	Encoding ID	Description
3	0	Symbol
3	1	Unicode
3	2	ShiftJIS
3	3	Big5
3	4	PRC
3	5	Wansung
3	6	Johab

The Character To Glyph Index Mapping Table is organized as follows:

Type	Description
USHORT	Table version number (0).
USHORT	Number of encoding tables, *n*.

This is followed by an entry for each of the *n* encoding table specifying the particular encoding, and the offset to the actual subtable:

Type	Description
USHORT	Platform ID.
USHORT	Platform-specific encoding ID.
ULONG	Byte offset from beginning of table to the subtable for this encoding.

Format 0: Byte encoding table

This is the Apple standard character to glyph index mapping table.

Type	Name	Description
USHORT	format	Format number is set to 0.
USHORT	length	This is the length in bytes of the subtable.
USHORT	version	Version number (starts at 0).
BYTE	glyphIdArray[256]	An array that maps character codes to glyph index values.

This is a simple 1 to 1 mapping of character codes to glyph indices. The glyph set is limited to 256. Note that if this format is used to index into a larger glyph set, only the first 256 glyphs will be accessible.

Format 2: High-byte mapping through table

This subtable is useful for the national character code standards used for Japanese, Chinese, and Korean characters. These code standards use a mixed 8/16-bit encoding, in which certain byte values signal the first byte of a 2-byte character (but these values are also legal as the second byte of a 2-byte character). Character codes are always 1-byte. The glyph set is limited to 256.

In addition, even for the 2-byte characters, the mapping of character codes to glyph index values depends heavily on the first byte. Consequently, the table begins with an array that maps the first byte to a 4-word subHeader. For 2-byte character codes, the subHeader is used to map the second byte's value through a subArray, as described below. When processing mixed 8/16-bit text, subHeader 0 is special: it is used for single-byte character codes. When subHeader zero is used, a second byte is not needed; the single byte value is mapped through the subArray.

Type	Name	Description
USHORT	format	Format number is set to 2.
USHORT	length	Length in bytes.
USHORT	version	Version number (starts at 0)
USHORT	subHeaderKeys[256]	Array that maps high bytes to subHeaders: value is subHeader index * 8.
4 words struct	subHeaders[ ]	Variable-length array of subHeader structures.
4 words-struct	subHeaders[ ]	
USHORT	glyphIndexArray[ ]	Variable-length array containing subarrays used for mapping the low byte of 2-byte characters.

A subHeader is structured as follows:

Type	Name	Description
USHORT	firstCode	First valid low byte for this subHeader.
USHORT	entryCount	Number of valid low bytes for this subHeader.
SHORT	idDelta	See text below.
USHORT	idRangeOffset	See text below.

The firstCode and entryCount values specify a subrange that begins at firstCode and has a length equal to the value of entryCount. This subrange stays within the 0–255 range of the byte being mapped. Bytes outside of this subrange are mapped to glyph index 0 (missing glyph).The offset of the byte within this subrange is then used as index into a corresponding subarray of glyphIndexArray. This subarray is also of length entryCount. The value of the idRangeOffset is the number of bytes past the actual location of the idRangeOffset word where the glyphIndexArray element corresponding to firstCode appears.

Finally, if the value obtained from the subarray is not 0 (which indicates the missing glyph), you should add idDelta to it in order to get the glyphIndex. The value idDelta permits the same subarray to be used for several different subheaders. The idDelta arithmetic is modulo 65536.

Format 4: Segment mapping to delta values

This is the Microsoft standard character to glyph index mapping table.

This format is used when the character codes for the characters represented by a font fall into several contiguous ranges, possibly with holes in some or all of the ranges (that is, some of the codes in a range may not have a representation in the font). The format-dependent data is divided into three parts, which must occur in the following order:

1. A four-word header gives parameters for an optimized search of the segment list;
2. Four parallel arrays describe the segments (one segment for each contiguous range of codes);
3. A variable-length array of glyph IDs (unsigned words).

Type	Name	Description
USHORT	format	Format number is set to 4.
USHORT	length	Length in bytes.
USHORT	version	Version number (starts at 0).
USHORT	segCountX2	2 x segCount.
USHORT	searchRange	2 x (2**floor($\log_2$(segCount)))
USHORT	entrySelector	$\log_2$(searchRange/2)
USHORT	rangeShift	2 x segCount - searchRange
USHORT	endCount[segCount]	End characterCode for each segment, last =0xFFFF.
USHORT	reservedPad	Set to 0.
USHORT	startCount[segCount]	Start character code for each segment.
USHORT	idDelta[segCount]	Delta for all character codes in segment.
USHORT	idRangeOffset[segCount]	Offsets into glyphIdArray or 0
USHORT	glyphIdArray[ ]	Glyph index array (arbitrary length)

The number of segments is specified by segCount, which is not explicitly in the header; however, all of the header parameters are derived from it. The searchRange value is twice the largest power of 2 that is less than or equal to segCount. For example, if segCount=39, we have the following:

segCountX2	78	
searchRange	64	(2 * largest power of 2 ≤ 39)
entrySelector	5	$\log_2$(32)
rangeShift	14	2 x 39 - 64

Each segment is described by a startCode and endCode, along with an idDelta and an idRangeOffset, which are used for mapping the character codes in the segment. The segments are sorted in order of increasing endCode values, and the segment values are specified in four parallel arrays. You search for the first endCode that is greater than or equal to the character code you want to map. If the corresponding startCode is less than or equal to the character code, then you use the corresponding idDelta and idRangeOffset to map the character code to a glyph index (otherwise, the missingGlyph is returned). For the search to terminate, the final endCode value must be 0xFFFF. This segment need not contain any valid mappings. (It can just map the single character code 0xFFFF to missingGlyph). However, the segment must be present.

If the idRangeOffset value for the segment is not 0, the mapping of character codes relies on glyphIdArray. The character code offset from startCode is added to the idRangeOffset value. This sum is used as an offset from the current location within idRangeOffset itself to index out the correct glyphIdArray value. This obscure indexing trick works because glyphIdArray immediately follows idRangeOffset in the font file. The C expression that yields the glyph index is:

```
*(idRangeOffset[i]/2 + (c - startCount[i]) + &idRangeOffset[i])
```

The value *c* is the character code in question, and *i* is the segment index in which *c* appears. If the value obtained from the indexing operation is not 0 (which indicates missingGlyph), idDelta[i] is added to it to get the glyph index. The idDelta arithmetic is modulo 65536.

If the idRangeOffset is 0, the idDelta value is added directly to the character code offset (i.e. `idDelta[i] + c`) to get the corresponding glyph index. Again, the idDelta arithmetic is modulo 65536.

As an example, the variant part of the table to map characters 10–20, 30–90, and 100–153 onto a contiguous range of glyph indices may look like this:

segCountX2:	8			
searchRange:	8			
entrySelector:	4			
rangeShift:	0			
endCode:	20	90	153	0xFFFF
reservedPad:	0			
startCode:	10	30	100	0xFFFF
idDelta:	-9	-18	-27	1
idRangeOffset:	0	0	0	0

This table performs the following mappings:

10 –> 10 – 9 = 1

20 –> 20 – 9 = 11

30 –> 30 – 18 = 12

90 –> 90 – 18 = 72

...and so on.

Note that the delta values could be reworked so as to reorder the segments.

Format 6: Trimmed table mapping

Type	Name	Description
USHORT	format	Format number is set to 6.
USHORT	length	Length in bytes.
USHORT	version	Version number (starts at 0)
USHORT	firstCode	First character code of subrange.
USHORT	entryCount	Number of character codes in subrange.
USHORT	glyphIdArray [entryCount]	Array of glyph index values for character codes in the range.

The firstCode and entryCount values specify a subrange (beginning at firstCode,length = entryCount) within the range of possible character codes. Codes outside of this subrange are mapped to glyph index 0. The offset of the code (from the first code) within this subrange is used as index to the glyphIdArray, which provides the glyph index value.

cvt - Control Value Table

This table contains a list of values that can be referenced by instructions. They can be used, among other things, to control characteristics for different glyphs.

Type	Description
FWORD[*n*]	List of *n* values referenceable by instructions.

EBDT - Embedded Bitmap Data Table

Three new tables are used to embed bitmaps in TrueType fonts. They are the 'EBLC' table for embedded bitmap locators, the 'EBDT' table for embedded bitmap data, and the 'EBSC' table for embedded bitmap scaling information.

TrueType embedded bitmaps are also called 'sbits' (for "scaler bitmaps"). A set of bitmaps for a face at a given size is called a strike.

The 'EBLC' table identifies the sizes and glyph ranges of the sbits, and keeps offsets to glyph bitmap data in indexSubTables. The 'EBDT' table then stores the glyph bitmap data, in a number of different possible formats. Glyph metrics information may be stored in either the 'EBLC' or 'EBDT' table, depending upon the indexSubTable and glyph bitmap data formats. The 'EBSC' table identifies sizes that will be handled by scaling up or scaling down other sbit sizes.

The 'EBDT' table uses the same format as Apple has defined for the QuickDraw GX 'bdat' table.

The 'EBDT' table begins with a header containing simply the table version number.

Type	Name	Description
FIXED	version	Initially defined as 0x00020000

The rest of the 'EBDT' table is a simply a collection of bitmap data. The data can be in a number of possible formats, indicated by information in the 'EBLC' table. Some of the formats contain metric information plus image data, and other formats contain only the image data. Long word alignment is not required for these sub tables; byte alignment is sufficient.

There are also two different formats for glyph metrics: big glyph metrics and small glyph metrics. Big glyph metrics define metrics information for both horizontal and vertical layouts. This is important in fonts (such as Kanji) where both types of layout may be used. Small glyph metrics define metrics information for one layout direction only. Which direction applies, horizontal or vertical, is determined by the 'flags' field in the bitmapSizeTable field of the 'EBLC' table.

bigGlyphMetrics

Type	Name
BYTE	height
BYTE	width
CHAR	horiBearingX
CHAR	horiBearingY
BYTE	horiAdvance
CHAR	vertBearingX
CHAR	vertBearingY
BYTE	vertAdvance

smallGlyphMetrics

Type	Name
BYTE	height
BYTE	width
CHAR	BearingX
CHAR	BearingY
BYTE	Advance

The nine different formats currently defined for glyph bitmap data are listed and described below. Different formats are better for different purposes. Apple 'bdat' tables support only formats 1 through 7.

Format 1: small metrics, byte-aligned data

Type	Name	Description
smallGlyphMetrics	smallMetrics	Metrics information for the glyph
VARIABLE	image data	Byte-aligned bitmap data

Glyph bitmap format 1 consists of small metrics records (either horizontal or vertical depending on the bitmapSizeTable 'flag' value in the 'EBLC' table) followed by byte aligned bitmap data. The bitmap data begins with the most significant bit of the first byte corresponding to the top-left pixel of the bounding box, proceeding through succeeding bits moving left to right. The data for each row is padded to a byte boundary, so the next row begins with the most significant bit of a new byte. 1 bits correspond to black, and 0 bits to white.

Format 2: small metrics, bit-aligned data

Type	Name	Description
smallGlyphMetrics	small Metrics	Metrics information for the glyph
VARIABLE	image data	Bit-aligned bitmap data

Glyph bitmap format 2 is the same as format 1 except that the bitmap data is bit aligned. This means that the data for a new row will begin with the bit immediately following the last bit of the previous row. The start of each glyph must be byte aligned, so the last row of a glyph may require padding. This format takes a little more time to parse, but saves file space compared to format 1.

Format 3: (obsolete)

Format 4: (not supported) metrics in EBLC, compressed data

Glyph bitmap format 4 is a compressed format used by Apple in some of their Far East fonts. MS has not implemented it in our rasterizer.

Format 5: metrics in EBLC, bit-aligned image data only

Type	Name	Description
VARIABLE	image data	Bit-aligned bitmap data

Glyph bitmap format 5 is similar to format 2 except that no metrics information is included, just the bit aligned data. This format is for use with 'EBLC' indexSubTable format 2 or format 5, which will contain the metrics information for all glyphs. It works well for Kanji fonts.

The rasterizer recalculates sbit metrics for Format 5 bitmap data, allowing Windows to report correct ABC widths, even if the bitmaps have white space on either side of the bitmap image. This allows fonts to store monospaced bitmap glyphs in the efficient Format 5 without breaking Windows GetABCWidths call.

Format 6: big metrics, byte-aligned data

Type	Name	Description
bigGlyphMetrics	bigMetrics	Metrics information for the glyph
VARIABLE	image data	Byte-aligned bitmap data

Glyph bitmap format 6 is the same as format 1 except that is uses big glyph metrics instead of small.

Format 7: big metrics, bit-aligned data

Type	Name	Description
bigGlyphMetrics	bigMetrics	Metrics information for the glyph
VARIABLE	image data	Bit-aligned bitmap data

Glyph bitmap format 7 is the same as format 2 except that is uses big glyph metrics instead of small.

ebdtComponent; array used by Formats 8 and 9

Type	Name	Description
USHORT	glyphCode	Component glyph code
CHAR	xOffset	Position of component left
CHAR	yOffset	Position of component top

The component array, used by Formats 8 and 9, contains the glyph code of the component, which can be looked up in the 'EBLC' table, as well as xOffset and yOffset values which tell where to position the top-left corner of the component in the composite. Nested composites (a composite of composites) are allowed, and the number of nesting levels is determined by implementation stack space.

Format 8: small metrics, component data

Type	Name	Description
smallGlyphMetrics	smallMetrics	Metrics information for the glyph
BYTE	pad	Pad to short boundary
USHORT	numComponents	Number of components
ebdtComponent	componentArray[n]	Glyph code, offset array

Format 9: big metrics, component data

Type	Name	Description
bigGlyphMetrics	bigMetrics	Metrics information for the glyph
USHORT	numComponents	Number of components
ebdtComponent	componentArray[n]	Glyph code, offset array

Glyph bitmap formats 8 and 9 are used for composite bitmaps. For accented characters and other composite glyphs it may be more efficient to store a copy of each component separately, and then use a composite description to construct the finished glyph. The composite formats allow for any number of components, and allow the components to be positioned anywhere in the finished glyph. Format 8 uses small metrics, and format 9 uses big metrics.

EBLC - Embedded Bitmap Location Table

Three new tables are used to embed bitmaps in TrueType fonts. They are the 'EBLC' table for embedded bitmap locators, the 'EBDT' table for embedded bitmap data, and the 'EBSC' table for embedded bitmap scaling information. TrueType embedded bitmaps are called 'sbits' (for "scaler bitmaps"). A set of bitmaps for a face at a given size is called a strike.

The 'EBLC' table identifies the sizes and glyph ranges of the sbits, and keeps offsets to glyph bitmap data in indexSubTables. The 'EBDT' table then stores the glyph bitmap data, also in a number of different possible formats. Glyph metrics information may be stored in either the 'EBLC' or 'EBDT' table, depending upon the indexSubTable and glyph bitmap formats. The 'EBSC' table identifies sizes that will be handled by scaling up or scaling down other sbit sizes.

The 'EBLC' table uses the same format as the Apple QuickDraw GX 'bloc' table.

The 'EBLC' table begins with a header containing the table version and number of strikes. A TrueType font may have one or more strikes embedded in the 'EBDT' table.

eblcHeader

Type	Name	Description
FIXED	version	initially defined as 0x00020000
ULONG	numSizes	Number of bitmapSizeTables

The eblcHeader is followed immediately by the bitmapSizeTable array(s). The numSizes in the eblcHeader indicates the number of bitmapSizeTables in the array. Each strike is defined by one bitmapSizeTable.

bitmapSizeTable

Type	Name	Description
ULONG	indexSubTableArrayOffset	offset to index subtable from beginning of EBLC.
ULONG	indexTablesSize	number of bytes in corresponding index subtables and array
ULONG	numberOfIndexSubTables	an index subtable for each range or format change
ULONG	colorRef	not used; set to 0.
sbitLineMetrics	hori	line metrics for text rendered horizontally
sbitLineMetrics	vert	line metrics for text rendered vertically
USHORT	startGlyphIndex	lowest glyph index for this size
USHORT	endGlyphIndex	highest glyph index for this size
BYTE	ppemX	horizontal pixels per Em
BYTE	ppemY	vertical pixels per Em
BYTE	bitDepth	set to 1 for now
CHAR	flags	vertical or horizontal (see bitmapFlags)

The indexSubTableArrayOffset is the offset from the beginning of the 'EBLC' table to the indexSubTableArray. Each strike has one of these arrays to support various formats and discontiguous ranges of bitmaps. The indexTablesSize is the total number of bytes in the indexSubTableArray and the associated indexSubTables. The numberOfIndexSubTables is a count of the indexSubTables for this strike.

The horizontal and vertical line metrics contain the ascender, descender, linegap, and advance information for the strike. The line metrics format is described in the following table:

sbitLineMetrics

Type	Name
CHAR	ascender
CHAR	descender
BYTE	widthMax
CHAR	caretSlopeNumerator
CHAR	caretSlopeDenominator
CHAR	caretOffset
CHAR	minOriginSB
CHAR	minAdvanceSB
CHAR	maxBeforeBL
CHAR	minAfterBL
CHAR	pad1
CHAR	pad2

The caret slope determines the angle at which the caret is drawn, and the offset is the number of pixels (+ or -) to move the caret. This is a signed char since we are dealing with integer metrics. The minOriginSB, minAdvanceSB , maxBeforeBL, and minAfterBL are described in the diagrams below. The main need for these numbers is for scalers that may need to pre-allocate memory and/or need more metric information to position glyphs. All of the line metrics are one byte in length. The line metrics are not used directly by the rasterizer, but are available to clients who want to parse the 'EBLC' table.

The startGlyphIndex and endGlyphIndex describe the minimum and maximum glyph codes in the strike, but a strike does not necessarily contain bitmaps for all glyph codes in this range. The indexSubTables determine which glyphs are actually present in the 'EBDT' table.

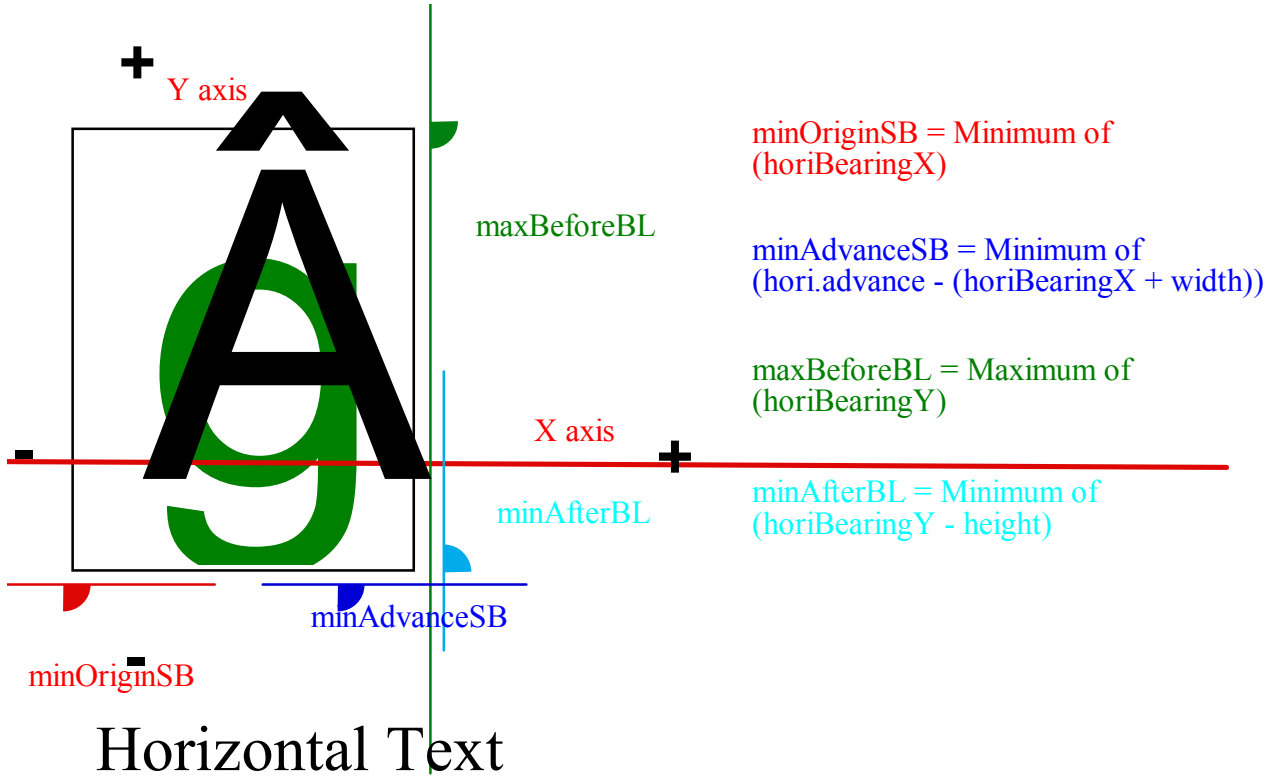
The ppemX and ppemY fields describe the size of the strike in pixels per Em. The ppem measurement is equivalent to point size on a 72 dots per inch device. Typically, ppemX will be equal to ppemY for devices with 'square pixels'. To accommodate devices with rectangular pixels, and to allow for bitmaps with other aspect ratios, ppemX and ppemY may differ.

The 'flags' byte contains two bits to indicate the direction of small glyph metrics: horizontal or vertical. The remaining bits are reserved.

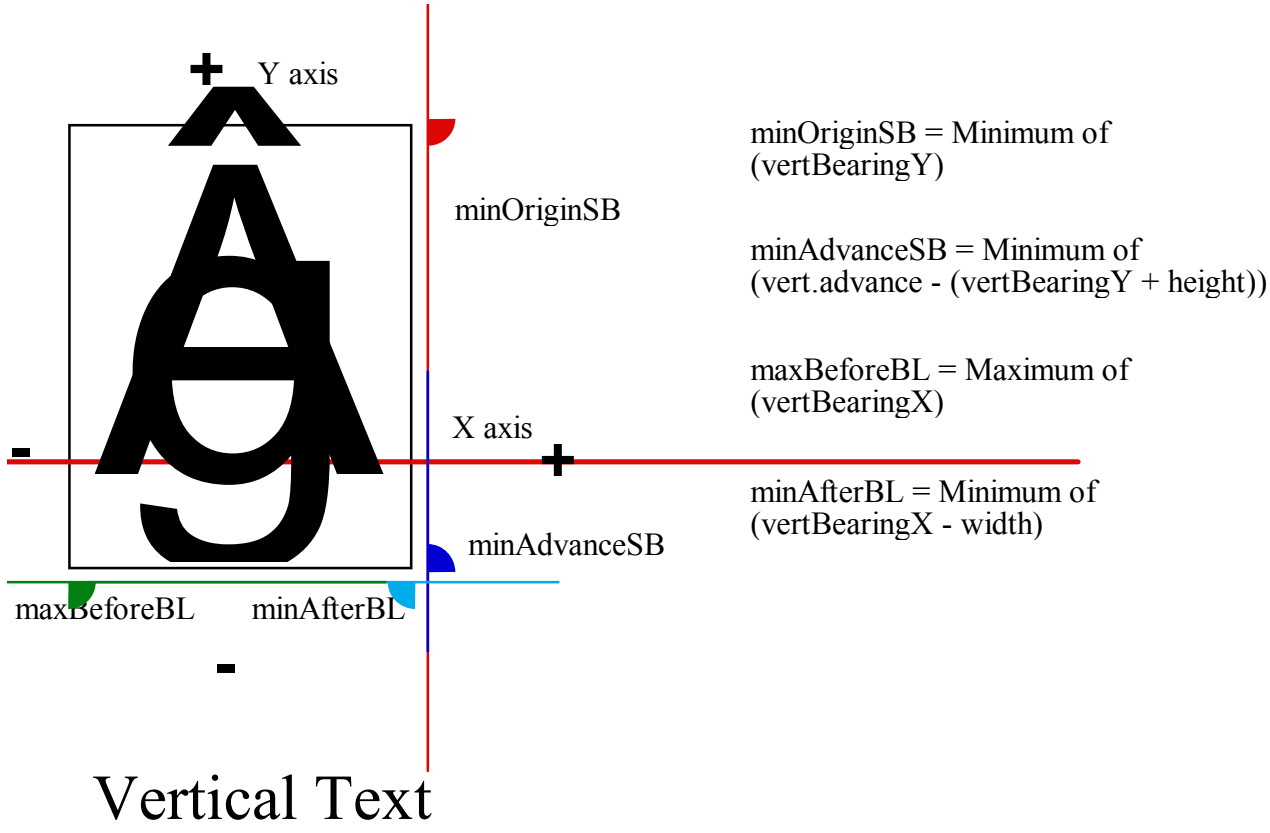
Bitmap Flags

Type	Value	Description
CHAR	0x01	Horizontal
CHAR	0x02	Vertical

The colorRef and bitDepth fields are reserved for future enhancements. For monochrome bitmaps they should have the values colorRef=0 and bitDepth=1.



Horizontal Text



Vertical Text

Associated with the image data for every glyph in a strike is a set of glyph metrics. These glyph metrics describe bounding box height and width, as well as side bearing and advance width information. The glyph metrics can be found in one of two places. For ranges of glyphs (not necessarily the whole strike) whose metrics may be different for each glyph, the glyph metrics are stored along with the glyph image data in the 'EBDT' table. Details of how this is done is described in the 'EBDT' section of this document. For ranges of glyphs whose metrics are identical for every glyph, we save significant space by storing a single copy of the glyph metrics in the indexSubTable in the 'EBLC'.

There are also two different formats for glyph metrics: big glyph metrics and small glyph metrics. Big glyph metrics define metrics information for both horizontal and vertical layouts. This is important in fonts (such as Kanji) where both types of layout may be used. Small glyph metrics define metrics information for one layout direction only. Which direction applies, horizontal or vertical, is determined by the 'flags' field in the bitmapSizeTable.

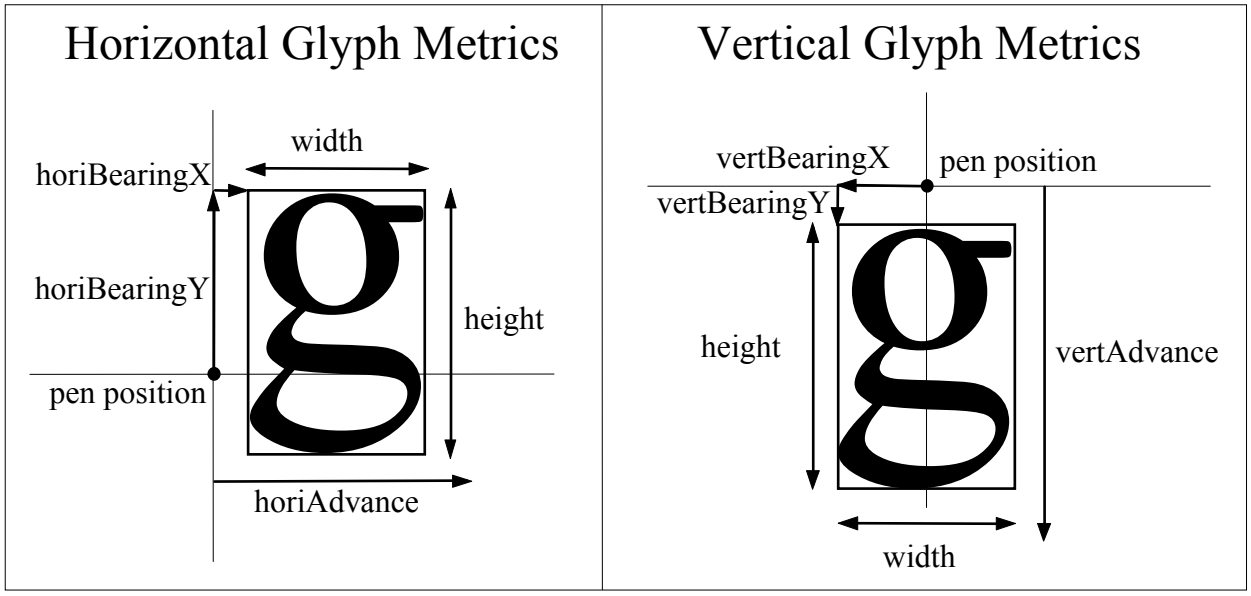
bigGlyphMetrics

Type	Name
BYTE	height
BYTE	width
CHAR	horiBearingX
CHAR	horiBearingY
BYTE	horiAdvance
CHAR	vertBearingX
CHAR	vertBearingY
BYTE	vertAdvance

smallGlyphMetrics

Type	Name
BYTE	height
BYTE	width
CHAR	BearingX
CHAR	BearingY
BYTE	Advance

The following diagram illustrates the meaning of the glyph metrics.



The bitmapSizeTable for each strike contains the offset to an array of indexSubTableArray elements. Each element describes a glyph code range and an offset to the indexSubTable for that range. This allows a strike to contain multiple glyph code ranges and to be represented in multiple index formats if desirable.

indexSubTableArray

Type	Name	Description
USHORT	firstGlyphIndex	first glyph code of this range
USHORT	lastGlyphIndex	last glyph code of this range (inclusive)
ULONG	additionalOffsetToIndexSubtable	add to indexSubTableArrayOffset to get offset from beginning of 'EBLC'

After determining the strike, the rasterizer searches this array for the range containing the given glyph code. When the range is found, the additionalOffsetToIndexSubtable is added to the indexSubTableArrayOffset to get the offset of the indexSubTable in the 'EBLC'.

The first indexSubTableArray is located after the last bitmapSizeSubTable entry. Then the indexSubTables for the strike follow. Another indexSubTableArray (if more than one strike) and its indexSubTables are next. The 'EBLC' continues with an array and indexSubTables for each strike.

We now have the offset to the indexSubTable. All indexSubTable formats begin with an indexSubHeader which identifies the indexSubTable format, the format of the 'EBDT' image data, and the offset from the beginning of the 'EBDT' table to the beginning of the image data for this range.

indexSubHeader

Type	Name	Description
USHORT	indexFormat	format of this indexSubTable
USHORT	imageFormat	format of 'EBDT' image data
ULONG	imageDataOffset	offset to image data in 'EBDT' table

There are currently five different formats used for the indexSubTable, depending upon the size and type of bitmap data in the glyph code range. Apple 'bloc' tables support only formats 1 through 3.

The choice of which indexSubTable format to use is up to the font manufacturer, but should be made with the aim of minimizing the size of the font file. Ranges of glyphs with variable metrics — that is, where glyphs may differ from each other in bounding box height, width, side bearings or advance — must use format 1, 3 or 4. Ranges of glyphs with constant metrics can save space by using format 2 or 5, which keep a single copy of the metrics information in the indexSubTable rather than a copy per glyph in the 'EBDT' table. In some monospaced fonts it makes sense to store extra white space around some of the glyphs to keep all metrics identical, thus permitting the use of format 2 or 5.

Structures for each indexSubTable format are listed below.

indexSubTable1: variable metrics glyphs with 4 byte offsets

Type	Name	Description
indexSubHeader	header	header info
ULONG	offsetArray[]	offsetArray[glyphIndex]+imageDataOffset= glyphData sizeOfArray=(lastGlyph-firstGlyph+1)+1+1 pad if needed

indexSubTable2: all glyphs have identical metrics

Type	Name	Description
indexSubHeader	header	header info
ULONG	imageSize	all the glyphs are of the same size
bigGlyphMetrics	bigMetrics	all glyphs have the same metrics; glyph data may be compressed, byte-aligned, or bit-aligned

indexSubTable3: variable metrics glyphs with 2 byte offsets

Type	Name	Description
indexSubHeader	header	header info
USHORT	offsetArray[]	offsetArray[glyphIndex]+imageDataOffset= glyphData sizeOfArray=(lastGlyph-firstGlyph+1)+1+1 pad if needed

indexSubTable4: variable metrics glyphs with sparse glyph codes

Type	Name	Description
indexSubHeader	header	header info
ULONG	numGlyphs	array length
codeOffsetPair	glyphArray[]	one per glyph; sizeOfArray=numGlyphs+1

codeOffsetPair: used by indexSubTable4

Type	Name	Description
USHORT	glyphCode	code of glyph present
USHORT	offset	location in EBDT

indexSubTable5: constant metrics glyphs with sparse glyph codes

Type	Name	Description
indexSubHeader	header	header info
ULONG	imageSize	all glyphs have the same data size
bigGlyphMetrics	bigMetrics	all glyphs have the same metrics
ULONG	numGlyphs	array length
USHORT	glyphCodeArray[]	one per glyph, sorted by glyph code; sizeOfArray=numGlyphs

The size of the 'EBDT' image data can be calculated from the indexSubTable information. For the constant metrics formats (2 and 5) the image data size is constant, and is given in the imageSize field. For the variable metrics formats (1, 3, and 4) image data must be stored contiguously and in glyph code order, so the image data size may be calculated by subtracting the offset for the current glyph from the offset of the next glyph. Because of this, it is necessary to store one extra element in the offsetArray pointing just past the end of the range's image data. This will allow the correct calculation of the image data size for the last glyph in the range.

Contiguous, or nearly contiguous, ranges of glyph codes are handled best by formats 1, 2, and 3 which store an offset for every glyph code in the range. Very sparse ranges of glyph codes should use format 4 or 5 which explicitly call out the glyph codes represented in the range. A small number of missing glyphs can be efficiently represented in formats 1 or 3 by having the offset for the missing glyph be followed by the same offset for the next glyph, thus indicating a data size of zero.

The only difference between formats 1 and 3 is the size of the offsetArray elements: format 1 uses ULONG's while format 3 uses USHORT's. Therefore format 1 can cover a greater range (> 64k bytes) while format 3 saves more space in the 'EBLC' table. Since the offsetArray elements are added to the imageDataOffset base address in the indexSubHeader, a very large set of glyph bitmap data could be addressed by splitting it into multiple ranges, each less than 64k bytes in size, allowing the use of the more efficient format 3.

The 'EBLC' table specification requires double word (ULONG) alignment for all subtables. This occurs naturally for indexSubTable formats 1, 2, and 4, but may not for formats 3 and 5, since they include arrays of type USHORT. When there is an odd number of elements in these arrays it is necessary to add an extra padding element to maintain proper alignment.

EBSC - Embedded Bitmap Scaling Table

The 'EBSC' table provides a mechanism for describing embedded bitmaps which are created by scaling other embedded bitmaps. While this is the sort of thing that outline font technologies were invented to avoid, there are cases (small sizes of Kanji, for example) where scaling a bitmap produces a more legible font than scan-converting an outline. For this reason the 'EBSC' table allows a font to define a bitmap strike as a scaled version of another strike.

The 'EBSC' table begins with a header containing the table version and number of strikes.

ebscHeader

Type	Name	Description
FIXED	version	initially defined as 0x00020000
ULONG	numSizes	

The ebscHeader is followed immediately by the bitmapScaleTable array. The numSizes in the ebscHeader indicates the number of bitmapScaleTables in the array. Each strike is defined by one bitmapScaleTable.

bitmapScaleTable

Type	Name	Description
sbitLineMetrics	hori	line metrics
sbitLineMetrics	vert	line metrics
BYTE	ppemX	target horizontal pixels per Em
BYTE	ppemY	target vertical pixels per Em
BYTE	substitutePpemX	use bitmaps of this size
BYTE	substitutePpemY	use bitmaps of this size

The line metrics have the same meaning as those in the bitmapSizeTable, and refer to font wide metrics after scaling. The ppemX and ppemY values describe the size of the font after scaling. The substitutePpemX and substitutePpemY values describe the size of a strike that exists as an sbit in the 'EBLC' and 'EBDT', and that will be scaled up or down to generate the new strike.

Notice that scaling in the x direction is independent of scaling in the y direction, and their scaling values may differ. A square aspect-ratio strike could be scaled to a non-square aspect ratio. Glyph metrics are scaled by the same factor as the pixels per Em (in the appropriate direction), and are rounded to the nearest integer pixel.

fpgm - Font Program

This table is similar to the CVT Program, except that it is only run once, when the font is first used. It is used only for FDEFs and IDEFs. Thus the CVT Program need not contain function definitions. However, the CVT Program may redefine existing FDEFs or IDEFs.

This table is optional.

Type	Description
BYTE[*n*]	Instructions

gasp - Grid-fitting And Scan-conversion Procedure

This table contains information which describes the preferred rasterization techniques for the typeface when it is rendered on grayscale-capable devices. This table also has some use for monochrome devices, which may use the table to turn off hinting at very large or small sizes, to improve performance.

At very small sizes, the best appearance on grayscale devices can usually be achieved by rendering the glyphs in grayscale without using hints. At intermediate sizes, hinting and monochrome rendering will usually produce the best appearance. At large sizes, the combination of hinting and grayscale rendering will typically produce the best appearance.

If the 'gasp' table is not present in a typeface, TrueType will apply default rules to decide how to render the glyphs on grayscale devices.

The 'gasp' table consists of a header followed by groupings of 'gasp' records:

gasp Table

Type	Name	Description
USHORT	version	Version number (set to 0)
USHORT	numRanges	Number of records to follow
GASPRANGE	gaspRange[numRanges]	Sorted by ppem

Each GASPRANGE record looks like this:

Type	Name	Description
USHORT	rangeMaxPPEM	Upper limit of range, in PPEM
USHORT	rangeGaspBehavior	Flags describing desired rasterizer behavior.

There are two flags for the rangeGaspBehavior flags:

Flag	Meaning
GASP_GRIDFIT	Use gridfitting
GASP_DOGRAY	Use grayscale rendering

The set of bit flags may be extended in the future.

The four currently defined values of rangeGaspBehavior would have the following uses:

Flag	Value	Meaning
GASP_DOGRAY	0x0002	small sizes, typically ppem<9
GASP_GRIDFIT	0x0001	medium sizes, typically 9<=ppem<=16
GASP_DOGRAY| GASP_GRIDFIT	0x0003	large sizes, typically ppem>16
(neither)	0x0000	optional for very large sizes, typically ppem>2048

The records in the gaspRange[] array must be sorted in order of increasing rangeMaxPPEM value. The last record should use 0xFFFF as a sentinel value for rangeMaxPPEM and should describe the behavior desired at all sizes larger than the previous record's upper limit. If the only entry in 'gasp' is the 0xFFFF sentinel value, the behavior described will be used for *all* sizes.

Sample 'gasp' table

Field	Value	Meaning
version	0x0000	
numRanges	0x0003	
Range[0], Flag	0x0008 0x0002	ppem<=8, grayscale only
Range[1], Flag	0x0010 0x0001	9<=ppem<=16, gridfit only
Range[2], Flag	0xFFFF 0x0003	16<ppem, gridfit and grayscale

glyf - Glyph Data

This table contains information that describes the glyphs in the font. Each glyph begins with the following header:

Type	Name	Description
SHORT	numberOfContours	If the number of contours is greater than or equal to zero, this is a single glyph; if negative, this is a composite glyph.
FWORD	xMin	Minimum x for coordinate data.
FWORD	yMin	Minimum y for coordinate data.
FWORD	xMax	Maximum x for coordinate data.
FWORD	yMax	Maximum y for coordinate data.

Note that the bounding rectangle from each character is defined as the rectangle with a lower left corner of (xMin, yMin) and an upper right corner of (xMax, yMax).

Note: The scaler will perform better if the glyph coordinates have been created such that the xMin is equal to the lsb. For example, if the lsb is 123, then xMin for the glyph should be 123. If the lsb is -12 then the xMin should be -12. If the lsb is 0 then xMin is 0. If all glyphs are done like this, set bit 1 of flags field in the 'head' table.

Simple Glyph Description

This is the table information needed if numberOfContours is greater than zero, that is, a glyph is not a composite.

Type	Name	Description
USHORT	endPtsOfContours[*n*]	Array of last points of each contour; *n* is the number of contours.
USHORT	instructionLength	Total number of bytes for instructions.
BYTE	instructions[*n*]	Array of instructions for each glyph; *n* is the number of instructions.
BYTE	flags[*n*]	Array of flags for each coordinate in outline; *n* is the number of flags.
BYTE or SHORT	xCoordinates[ ]	First coordinates relative to (0,0); others are relative to previous point.
BYTE or SHORT	yCoordinates[ ]	First coordinates relative to (0,0); others are relative to previous point.

Note: In the glyf table, the position of a point is not stored in absolute terms but as a vector relative to the previous point. The delta-x and delta-y vectors represent these (often small) changes in position.

Each flag is a single byte. Their meanings are shown below.

Flags	Bit	Description
On Curve	0	If set, the point is on the curve; otherwise, it is off the curve.
x-Short Vector	1	If set, the corresponding x-coordinate is 1 byte long, not 2.
y-Short Vector	2	If set, the corresponding y-coordinate is 1 byte long, not 2.
Repeat	3	If set, the next byte specifies the number of additional times this set of flags is to be repeated. In this way, the number of flags listed can be smaller than the number of points in a character.
This x is same (Positive x-Short Vector)	4	This flag has two meanings, depending on how the x-Short Vector flag is set. If x-Short Vector is set, this bit describes the sign of the value, with 1 equalling positive and 0 negative. If the x-Short Vector bit is not set and this bit is set, then the current x-coordinate is the same as the previous x-coordinate. If the x-Short Vector bit is not set and this bit is also not set, the current x-coordinate is a signed 16-bit delta vector.
This y is same (Positive y-Short Vector)	5	This flag has two meanings, depending on how the y-Short Vector flag is set. If y-Short Vector is set, this bit describes the sign of the value, with 1 equalling positive and 0 negative. If the y-Short Vector bit is not set and this bit is set, then the current y-coordinate is the same as the previous y-coordinate. If the y-Short Vector bit is not set and this bit is also not set, the current y-coordinate is a signed 16-bit delta vector.
Reserved	6	This bit is reserved. Set it to zero.
Reserved	7	This bit is reserved. Set it to zero.

Composite Glyph Description

This is the table information needed for composite glyphs (numberOfContours is -1). A composite glyph starts with two USHORT values ("flags" and "glyphIndex," i.e. the index of the first contour in this composite glyph); the data then varies according to "flags"). The C pseudo-code fragment below shows how the composite glyph information is stored and parsed; definitions for "flags" bits follow this fragment:

```
do {
     USHORT flags;
     USHORT glyphIndex;
     if ( flags & ARG_1_AND_2_ARE_WORDS) {
     (SHORT or FWord) argument1;
     (SHORT or FWord) argument2;
     } else {
            USHORT arg1and2; /* (arg1 << 8) | arg2 */
     }
     if ( flags & WE_HAVE_A_SCALE ) {
            F2Dot14  scale;    /* Format 2.14 */
     } else if ( flags & WE_HAVE_AN_X_AND_Y_SCALE ) {
            F2Dot14  xscale;    /* Format 2.14 */
            F2Dot14  yscale;    /* Format 2.14 */
     } else if ( flags & WE_HAVE_A_TWO_BY_TWO ) {
            F2Dot14  xscale;    /* Format 2.14 */
            F2Dot14  scale01;   /* Format 2.14 */
            F2Dot14  scale10;   /* Format 2.14 */
            F2Dot14  yscale;    /* Format 2.14 */
     }
} while ( flags & MORE_COMPONENTS )

if (flags & WE_HAVE_INSTR){

     USHORT numInstr

     BYTE instr[numInstr]
```

Argument1 and argument2 can be either x and y offsets to be added to the glyph or two point numbers. In the latter case, the first point number indicates the point that is to be matched to the new glyph. The second number indicates the new glyph's "matched" point. Once a glyph is added, its point numbers begin directly after the last glyphs (endpoint of first glyph + 1).

When arguments 1 and 2 are an x and a y offset instead of points and the bit ROUND_XY_TO_GRID is set to 1, the values are rounded to those of the closest grid lines before they are added to the glyph. X and Y offsets are described in FUnits.

If the bit WE_HAVE_A_SCALE is set, the scale value is read in 2.14 format—the value can be between -2 to almost +2. The glyph will be scaled by this value before grid-fitting.

The bit WE_HAVE_A_TWO_BY_TWO allows for an interrelationship between the x and y coordinates. This could be used for 90-degree rotations, for example.

These are the constants for the flags field:

Flags	Bit	Description
ARG_1_AND_2_ARE_WORDS	0	If this is set, the arguments are words; otherwise, they are bytes.
ARGS_ARE_XY_VALUES	1	If this is set, the arguments are xy values; otherwise, they are points.
ROUND_XY_TO_GRID	2	For the xy values if the preceding is true.
WE_HAVE_A_SCALE	3	This indicates that there is a simple scale for the component. Otherwise, scale = 1.0.
RESERVED	4	This bit is reserved. Set it to 0.
MORE_COMPONENTS	5	Indicates at least one more glyph after this one.
WE_HAVE_AN_X_AND_Y_SCALE	6	The x direction will use a different scale from the y direction.
WE_HAVE_A_TWO_BY_TWO	7	There is a 2 by 2 transformation that will be used to scale the component.
WE_HAVE_INSTRUCTIONS	8	Following the last component are instructions for the composite character.
USE_MY_METRICS	9	If set, this forces the aw and lsb (and rsb) for the composite to be equal to those from this original glyph. This works for hinted and unhinted characters.

The purpose of USE_MY_METRICS is to force the lsb and rsb to take on a desired value. For example, an i-circumflex (Unicode 00ef) is often composed of the circumflex and a dotless-i. In order to force the composite to have the same metrics as the dotless-i, set USE_MY_METRICS for the dotless-i component of the composite. Without this bit, the rsb and lsb would be calculated from the HMTX entry for the composite (or would need to be explicitly set with TrueType instructions).

Note that the behavior of the USE_MY_METRICS operation is undefined for rotated composite components.

hdmx - Horizontal Device Metrics

The Horizontal Device Metrics table stores integer advance widths scaled to particular pixel sizes. This allows the font manager to build integer width tables without calling the scaler for each glyph. Typically this table contains only selected screen sizes. This table is sorted by pixel size. The checksum for this table applies to both subtables listed.

Note that for non-square pixel grids (for example, on an EGA), the character width (in pixels) will be used to determine which device record to use. For example, a 12 point character on an EGA (resolution of 72x96) would be 12 pixels high, and 16 pixels wide, and the hdmx device record for 16 pixel characters would be used.

If bit 4 of the flag field in the 'head' table is not set, then it is assumed that the font scales linearly; in this case an 'hdmx' table is not necessary and should not be built. If bit 4 of the flag field is set, then one or more glyphs in the font are assumed to scale nonlinearly. In this case, performance can be improved by including the 'hdmx' table with one or more important DeviceRecord's for important sizes. Please see the chapter "Recommendations for Windows Fonts" for more detail.

The table begins as follows:

Type	Description
USHORT	Table version number (starts at 0)
SHORT	Number of device records.
LONG	Size of a device record, long aligned.
DeviceRecord	Records[number of device records].

Each DeviceRecord for format 0 looks like this.

Type	Description
BYTE	Pixel size for following widths (as ppem).
BYTE	Maximum width.
BYTE	Widths[numGlyphs] (numGlyphs is from the 'maxp' table).

Each DeviceRecord is padded with 0's to make it long word aligned.

Each Width value is the width of the particular glyph, in pixels, at the pixels per em (ppem) size listed at the start of the DeviceRecord.

The ppem sizes are measured along the y axis.

head - Font Header

This table gives global information about the font. The bounding box values should be computed using *only* glyphs that have contours. Glyphs with no contours should be ignored for the purposes of these calculations.

Type	Name	Description
FIXED	Table version number	0x00010000 for version 1.0.
FIXED	fontRevision	Set by font manufacturer.
ULONG	checkSumAdjustment	To compute: set it to 0, sum the entire font as ULONG, then store 0xB1B0AFBA - sum.
ULONG	magicNumber	Set to 0x5F0F3CF5.
USHORT	flags	Bit 0 - baseline for font at y=0; Bit 1 - left sidebearing at x=0; Bit 2 - instructions may depend on point size; Bit 3 - force ppem to integer values for all internal scaler math; may use fractional ppem sizes if this bit is clear; Bit 4 - instructions may alter advance width (the advance widths might not scale linearly); Note: All other bits must be zero.
USHORT	unitsPerEm	Valid range is from 16 to 16384
longDateTime	created	International date (8-byte field).
longDateTime	modified	International date (8-byte field).
FWORD	xMin	For all glyph bounding boxes.
FWORD	yMin	For all glyph bounding boxes.
FWORD	xMax	For all glyph bounding boxes.
FWORD	yMax	For all glyph bounding boxes.
USHORT	macStyle	Bit 0 bold (if set to 1); Bit 1 italic (if set to 1) Bits 2-15 reserved (set to 0).
USHORT	lowestRecPPEM	Smallest readable size in pixels.
SHORT	fontDirectionHint	0 Fully mixed directional glyphs; 1 Only strongly left to right; 2 Like 1 but also contains neutrals[1]; -1 Only strongly right to left; -2 Like -1 but also contains neutrals.
SHORT	indexToLocFormat	0 for short offsets, 1 for long.
SHORT	glyphDataFormat	0 for current format.

[1] A neutral character has no inherent directionality; it is not a character with zero (0) width. Spaces and punctuation are examples of neutral characters. Non-neutral characters are those with inherent directionality. For example, Roman letters (left-to-right) and Arabic letters (right-to-left) have directionality. In a "normal" Roman font where spaces and punctuation are present, the font direction hints should be set to two (2).

Note that macStyle bits must agree with the ‘OS/2’ table fsSelection bits. The fsSelection bits are used over the macStyle bits in Microsoft Windows. The PANOSE values and ‘post’ table values are ignored for determining bold or italic fonts.

The Date format used in this table follows the Macintosh convention of the number of seconds since 1904 (see Apple’s *Inside Macintosh* series).

hhea - Horizontal Header

This table contains information for horizontal layout. The values in the minRightSidebearing, minLeftSideBearing and xMaxExtent should be computed using *only* glyphs that have contours. Glyphs with no contours should be ignored for the purposes of these calculations. All reserved areas must be set to 0.

Type	Name	Description
FIXED	Table version number	0x00010000 for version 1.0.
FWORD	Ascender	Typographic ascent.
FWORD	Descender	Typographic descent.
FWORD	LineGap	Typographic line gap. Negative LineGap values are treated as zero in Windows 3.1, System 6, and System 7.
UFWORD	advanceWidthMax	Maximum advance width value in 'hmtx' table.
FWORD	minLeftSideBearing	Minimum left sidebearing value in 'hmtx' table.
FWORD	minRightSideBearing	Minimum right sidebearing value; calculated as Min(aw - lsb - (xMax - xMin)).
FWORD	xMaxExtent	Max(lsb + (xMax - xMin)).
SHORT	caretSlopeRise	Used to calculate the slope of the cursor (rise/run); 1 for vertical.
SHORT	caretSlopeRun	0 for vertical.
SHORT	(reserved)	set to 0
SHORT	(reserved)	set to 0
SHORT	(reserved)	set to 0
SHORT	(reserved)	set to 0
SHORT	(reserved)	set to 0
SHORT	metricDataFormat	0 for current format.
USHORT	numberOfHMetrics	Number of hMetric entries in 'hmtx' table; may be smaller than the total number of glyphs in the font.

hmtx - Horizontal Metrics

The type longHorMetric is defined as an array where each element has two parts: the advance width, which is of type uFWord, and the left side bearing, which is of type FWord. Or, more formally:

```
typedef struct    _longHorMertric {
    uFWord advanceWidth;
    FWord  lsb;

}  longHorMetric;
```

Field	Type	Description
hMetrics	longHorMetric [numberOfHMetrics]	Paired advance width and left side bearing values for each glyph. The value numOfHMetrics comes from the 'hhea' table. If the font is monospaced, only one entry need be in the array, but that entry is required. The last entry applies to all subsequent glyphs.
leftSideBearing	FWord[ ]	Here the advanceWidth is assumed to be the same as the advanceWidth for the last entry above. The number of entries in this array is derived from numGlyphs (from 'maxp' table) minus numberOfHMetrics. This generally is used with a run of monospaced glyphs (e.g., Kanji fonts or Courier fonts). Only one run is allowed and it must be at the end. This allows a monospaced font to vary the left side bearing values for each glyph.

For any glyph, xmax and xmin are given in 'glyf' table, lsb and aw are given in 'hmtx' table. rsb is calculated as follows:

```
rsb = aw - (lsb + xmax - xmin)
```

If pp1 and pp2 are phantom points used to control lsb and rsb, their initial position in x is calculated as follows:

```
pp1 = xmin - lsb              pp2 = pp1 + aw
```

kern- Kerning

The kerning table contains the values that control the intercharacter spacing for the glyphs in a font. There is currently no system level support for kerning (other than returning the kern pairs and kern values).

Each subtable varies in format, and can contain information for vertical or horizontal text, and can contain kerning values or minimum values. Kerning values are used to adjust inter-character spacing, and minimum values are used to limit the amount of adjustment that the scaler applies by the combination of kerning and tracking. Because the adjustments are additive, the order of the subtables containing kerning values is not important. However, tables containing minimum values should usually be placed last, so that they can be used to limit the total effect of other subtables.

The kerning table in the TrueType font file has a header, which contains the format number and the number of subtables present, and the subtables themselves.

Type	Field	Description
USHORT	version	Table version number (starts at 0)
USHORT	nTables	Number of subtables in the kerning table.

Kerning subtables will share the same header format. This header is used to identify the format of the subtable and the kind of information it contains:

Type	Field	Description
USHORT	version	Kern subtable version number
USHORT	length	Length of the subtable, in bytes (including this header).
USHORT	coverage	What type of information is contained in this table.

The coverage field is divided into the following sub-fields, with sizes given in bits:

Sub-field	Bits #'s	Size	Description
horizontal	0	1	1 if table has horizontal data, 0 if vertical.
minimum	1	1	If this bit is set to 1, the table has minimum values. If set to 0, the table has kerning values.
cross-stream	2	1	If set to 1, kerning is perpendicular to the flow of the text. If the text is normally written horizontally, kerning will be done in the up and down directions. If kerning values are positive, the text will be kerned upwards; if they are negative, the text will be kerned downwards. If the text is normally written vertically, kerning will be done in the left and right directions. If kerning values are positive, the text will be kerned to the right; if they are negative, the text will be kerned to the left. The value 0x8000 in the kerning data resets the cross-stream kerning back to 0.
override	3	1	If this bit is set to 1 the value in this table should replace the value currently being accumulated.
reserved1	4-7	4	Reserved. This should be set to zero.
format	8-15	8	Format of the subtable. Only formats 0 and 2 have been defined. Formats 1 and 3 through 255 are reserved for future use.

Format 0

This is the only format that will be properly interpreted by Windows and OS/2.

This subtable is a sorted list of kerning pairs and values. The list is preceded by information which makes it possible to make an efficient binary search of the list:

Type	Field	Description
USHORT	nPairs	This gives the number of kerning pairs in the table.
USHORT	searchRange	The largest power of two less than or equal to the value of nPairs, multiplied by the size in bytes of an entry in the table.
USHORT	entrySelector	This is calculated as $\log_2$ of the largest power of two less than or equal to the value of nPairs. This value indicates how many iterations of the search loop will have to be made. (For example, in a list of eight items, there would have to be three iterations of the loop).
USHORT	rangeShift	The value of nPairs minus the largest power of two less than or equal to nPairs, and then multiplied by the size in bytes of an entry in the table.

This is followed by the list of kerning pairs and values. Each has the following format:

Type	Field	Description
USHORT	left	The glyph index for the left-hand glyph in the kerning pair.
USHORT	right	The glyph index for the right-hand glyph in the kerning pair.
FWORD	value	The kerning value for the above pair, in FUnits. If this value is greater than zero, the characters will be moved apart. If this value is less than zero, the character will be moved closer together.

The left and right halves of the kerning pair make an unsigned 32-bit number, which is then used to order the kerning pairs numerically.

A binary search is most efficiently coded if the search range is a power of two. The search range can be reduced by half by shifting instead of dividing. In general, the number of kerning pairs, nPairs, will not be a power of two. The value of the search range, searchRange, should be the largest power of two less than or equal to nPairs. The number of pairs not covered by searchRange (that is, nPairs – searchRange) is the value rangeShift.

Windows v3.1 does not make use of the 'kern' data other than to expose it to applications through the GetFontData() API.

Format 2

This subtable is a two-dimensional array of kerning values. The glyphs are mapped to classes, using a different mapping for left- and right-hand glyphs. This allows glyphs that have similar right- or left-side shapes to be handled together. Each similar right- or left-hand shape is said to be single class.

Each row in the kerning array represents one left-hand glyph class, each column represents one right-hand glyph class, and each cell contains a kerning value. Row and column 0 always represent glyphs that do not kern and contain all zeros.

The values in the right class table are stored pre-multiplied by the number of bytes in a single kerning value, and the values in the left class table are stored pre-multiplied by the number of bytes in one row. This eliminates needing to multiply the row and column values together to determine the location of the kerning value. The array can be indexed by doing the right- and left-hand class mappings, adding the class values to the address of the array, and fetching the kerning value to which the new address points.

The header for the simple array has the following format:

Type	Field	Description
USHORT	rowWidth	The width, in bytes, of a row in the table.
USHORT	leftClassTable	Offset from beginning of this subtable to left-hand class table.
USHORT	rightClassTable	Offset from beginning of this subtable to right-hand class table.
USHORT	array	Offset from beginning of this subtable to the start of the kerning array.

Each class table has the following header:

Type	Field	Description
USHORT	firstGlyph	First glyph in class range.
USHORT	nGlyphs	Number of glyph in class range.

This header is followed by nGlyphs number of class values, which are in USHORT format. Entries for glyphs that don't participate in kerning should point to the row or column at position zero.

The array itself is a left by right array of kerning values, which are FWords, where left is the number of left-hand classes and R is the number of right-hand classes. The array is stored by row.

Note that this format is the quickest to process since each lookup requires only a few index operations. The table can be quite large since it will contain the number of cells equal to the product of the number of right-hand classes and the number of left-hand classes, even though many of these classes do not kern with each other.

loca - Index to Location

The indexToLoc table stores the offsets to the locations of the glyphs in the font, relative to the beginning of the glyphData table. In order to compute the length of the last glyph element, there is an extra entry after the last valid index.

By definition, index zero points to the "missing character," which is the character that appears if a character is not found in the font. The missing character is commonly represented by a blank box (such as □) or a space. If the font does not contain an outline for the missing character, then the first and second offsets should have the same value. This also applies to any other character without an outline, such as the space character.

Most routines will look at the 'maxp' table to determine the number of glyphs in the font, but the value in the 'loca' table should agree.

There are two versions of this table, the short and the long. The version is specified in the indexToLocFormat entry in the 'head' table.

Short version

Type	Name	Description
USHORT	offsets[*n*]	The actual local offset divided by 2 is stored. The value of *n* is numGlyphs + 1. The value for numGlyphs is found in the 'maxp' table.

Long version

Type	Name	Description
ULONG	offsets[*n*]	The actual local offset is stored. The value of *n* is numGlyphs + 1. The value for numGlyphs is found in the 'maxp' table.

Note that the local offsets should be long-aligned, i.e., multiples of 4. Offsets which are not long-aligned may seriously degrade performance of some processors.

LTSH - Linear Threshold

There are noticeable improvements to fonts on the screen when instructions are carefully applied to the sidebearings. The gain in readability is offset by the necessity for the OS to grid fit the glyphs in order to find the actual advance width for the glyphs (since instructions may be moving the sidebearing points). TrueType already has one mechanism to side step the speed issues: the 'hdmx' table, where precomputed advance widths may be saved for selected ppem sizes. The 'LTSH' table (Linear ThreSHold) is a second, complementary method.

The LTSH table defines the point at which it is reasonable to assume linearly scaled advance widths on a glyph-by-glyph basis. This table should *not* be included unless bit 4 of the "flags" field in the 'head' table is set. The criteria for linear scaling is:

> a. (ppem size is ≥ 50) AND (difference between the rounded linear width and the rounded instructed width ≤ 2% of the rounded linear width)
>
> or b. Linear width == Instructed width

The LTSH table records the ppem for each glyph at which the scaling becomes linear again, despite instructions effecting the advance width. It is a requirement that, at and above the recorded threshold size, the glyph remain linear in its scaling (i.e., not legal to set threshold at 55 ppem if glyph becomes non-linear again at 90 ppem). The format for the table is:

Type	Name	Description
USHORT	version	Version number (starts at 0).
USHORT	numGlyphs	Number of glyphs (from "numGlyphs" in 'maxp' table).
BYTE	yPels[numGlyphs]	The vertical pel height at which the glyph can be assumed to scale linearly. On a per glyph basis.

Note that glyphs which do not have instructions on their sidebearings should have yPels = 1; i.e., always scales linearly.

maxp - Maximum Profile

This table establishes the memory requirements for this font.

Type	Name	Description
Fixed	Table version number	0x00010000 for version 1.0.
USHORT	numGlyphs	The number of glyphs in the font.
USHORT	maxPoints	Maximum points in a non-composite glyph.
USHORT	maxContours	Maximum contours in a non-composite glyph.
USHORT	maxCompositePoints	Maximum points in a composite glyph.
USHORT	maxCompositeContours	Maximum contours in a composite glyph.
USHORT	maxZones	1 if instructions do not use the twilight zone (Z0), or 2 if instructions do use Z0; should be set to 2 in most cases.
USHORT	maxTwilightPoints	Maximum points used in Z0.
USHORT	maxStorage	Number of Storage Area locations.
USHORT	maxFunctionDefs	Number of FDEFs.
USHORT	maxInstructionDefs	Number of IDEFs.
USHORT	maxStackElements	Maximum stack depth[2].
USHORT	maxSizeOfInstructions	Maximum byte count for glyph instructions.
USHORT	maxComponentElements	Maximum number of components referenced at “top level” for any composite glyph.
USHORT	maxComponentDepth	Maximum levels of recursion; 1 for simple components.

[2] This includes Font and CVT Programs, as well as the instructions for each glyph.

name - Naming Table

The naming table allows multilingual strings to be associated with the TrueType font file. These strings can represent copyright notices, font names, family names, style names, and so on. To keep this table short, the font manufacturer may wish to make a limited set of entries in some small set of languages; later, the font can be "localized" and the strings translated or added. Other parts of the TrueType font file that require these strings can then refer to them simply by their index number. Clients that need a particular string can look it up by its platform ID, character encoding ID, language ID and name ID. Note that some platforms may require single byte character strings, while others may require double byte strings.

The Naming Table is organized as follows:

Type	Description
USHORT	Format selector (=0).
USHORT	Number of NameRecords that follow *n*.
USHORT	Offset to start of string storage (from start of table).
n NameRecords	The NameRecords.
(Variable)	Storage for the actual string data.

Each **NameRecord** looks like this:

Type	Description
USHORT	Platform ID.
USHORT	Platform-specific encoding ID.
USHORT	Language ID.
USHORT	Name ID.
USHORT	String length (in bytes).
USHORT	String offset from start of storage area (in bytes).

Following are the descriptions of the four kinds of ID. Note that the specific values listed here are the only ones that are predefined; new ones may be added by registry with Apple Developer Technical Support. Similar to the character encoding table, the NameRecords is sorted by platform ID, then platform-specific ID, then language ID, and then by name ID.

Platform ID

ID	Platform	Specific encoding
0	Apple Unicode	none
1	Macintosh	Script manager code
2	ISO	ISO encoding
3	*Microsoft*	Microsoft encoding

The values 240 through 255 are reserved for user-defined platforms. The DTS registry will never assign these values to a registered platform.

***Microsoft* platform-specific encoding ID's (platform ID = 3)**

Code	Description
0	Undefined character set or indexing scheme
1	UGL character set with Unicode indexing scheme (see chapter, "Character Sets.")

When building a Unicode font for Windows, the platform ID should be 3 and the encoding ID should be 1. When building a symbol font for Windows, the platform ID should be 3 and the encoding ID should be 0. When building a font that will be used on the Macintosh, the platform ID should be 1 and the encoding ID should be 0.

The PanEuropean Windows product will contain locale data for the following locales. This is also the list from which the user may choose a locale in custom setup and the mapping to Windows and DOS codepages based on that choice. The language ID (LCID in the table below) refers to a value which identifies the language in which a particular string is written.

Primary Language	Locale Name	LCID	Win CP	DOS CP
Albanian	Albania	(041c; SQI)		
Basque	Basque	(042D; EUQ)	1252	850
Byelorussian	Byelorussia	(0423, BEL)	1251	866
Bulgarian	Bulgaria	(0402, BGR)	1251	866
Catalan	Catalan	(0403; CAT)	1252	850
Croatian	Croatian	(041a, SHL)	1250	852
Czech	Czech	(0405; CSY)	1250	852
Danish	Danish	(0406; DAN)	1252	865
Dutch (2):	Dutch (Standard)	(0413; NLD)	1252	850
Dutch (2):	Belgian (Flemish)	(0813; NLB)	1252	850
English (6):	American	(0409; ENU)	1252	437
English (6):	British	(0809; ENG)	1252	850
English (6):	Australian	(0c09; ENA)	1252	850
English (6):	Canadian	(1009; ENC)	1252	850
English (6):	New Zealand	(1409; ENZ)	1252	850
English (6):	Ireland	(1809; ENI)	1252	850
Estonian	Estonia	(0425, ETI)	1257	775
Finnish	Finnish	(040b; FIN)	1252	850
French	French (Standard)	(040c; FRA)	1252	850

Primary Language	Locale Name	LCID	Win CP	DOS CP
French	Belgian	(080c; FRB)	1252	850
French	Canadian	(0c0c; FRC)	1252	850
French	Swiss	(100c; FRS)	1252	850
French	Luxembourg	(140c; FRL)	1252	850
German	German (Standard)	(0407; DEU)	1252	850
German	Swiss	(0807; DES)	1252	850
German	Austrian	(0c07; DEA)	1252	850
German	Luxembourg	(1007; DEL)	1252	850
German	Liechtenstein	(1407; DEC)	1252	850
Greek	Greek	(0408; ELL)	1253	737 or 869[3]
Hungarian	Hungarian	(040e; HUN)	1250	852
Icelandic	Icelandic	(040F; ISL)	1252	850
Italian (2):	Italian (Standard)	(0410; ITA)	1252	850
Italian (2):	Swiss	(0810; ITS)	1252	850
Latvian	Latvia	(0426, LVI)	1257	775
Lithuanian	Lithuania	(0427, LTH)	1257	775
Norwegian (2):	Norwegian (Bokmal)	(0414; NOR)	1252	850
Norwegian (2):	Norwegian (Nynorsk)	(0814; NON)	1252	850
Polish	Polish	(0415; PLK)	1250	852
Portuguese (2):	Portuguese (Brazilian)	(0416; PTB)	1252	850
Portuguese (2):	Portuguese (Standard)	(0816; PTG)	1252	850
Romanian (2):	Romania	(0418, ROM)	1250	852
Russian	Russian	(0419; RUS)	1251	866
Slovak	Slovak	(041b; SKY)	1250	852
Slovenian	Slovenia	(0424, SLV)	1250	852
Spanish (3):	Spanish (Traditional Sort)	(040a; ESP)	1252	850
Spanish (3):	Mexican	(080a; ESM)	1252	850
Spanish (3):	Spanish (Modern Sort)	(0c0a; ESN)	1252	850

[3] 737 is default, but 869 (IBM Greek) will be available at setup time through the selection of a bogus Greek locale in Custom setup.

Primary Language	Locale Name	LCID	Win CP	DOS CP
Swedish	Swedish	(041D; SVE)	1252	850
Turkish	Turkish	(041f; TRK)	1254	857
Ukrainian	Ukraine	(0422, UKR)	1251	866

Macintosh **platform-specific encoding ID's (script manager codes) (platform ID = 1)**

Code	Script	Code	Script
0	Roman	17	Malayalam
1	Japanese	18	Sinhalese
2	Chinese	19	Burmese
3	Korean	20	Khmer
4	Arabic	21	Thai
5	Hebrew	22	Laotian
6	Greek	23	Georgian
7	Russian	24	Armenian
8	RSymbol	25	Maldivian
9	Devanagari	26	Tibetan
10	Gurmukhi	27	Mongolian
11	Gujarati	28	Geez
12	Oriya	29	Slavic
13	Bengali	30	Vietnamese
14	Tamil	31	Sindhi
15	Telugu	32	Uninterp
16	Kannada		

Macintosh language ID's:

Code	Language	Code	Language
0	English	12	Arabic
1	French	13	Finnish
2	German	14	Greek
3	Italian	15	Icelandic
4	Dutch	16	Maltese
5	Swedish	17	Turkish
6	Spanish	18	Yugoslavian
7	Danish	19	Chinese
8	Portuguese	20	Urdu
9	Norwegian	21	Hindi
10	Hebrew	22	Thai
11	Japanese		

***ISO* specific encodings (platform ID = 2)**

Code	ISO encoding
0	7-bit ASCII
1	ISO 10646
2	ISO 8859-1

There are not any ISO-specific language ID's.

The following *name ID's* are defined, and they apply to all platforms. Extensions to this table will be registered with Apple DTS.

Name ID's

Code	Meaning
0	Copyright notice.
1	Font Family name
2	Font Subfamily name; for purposes of definition, this is assumed to address style (italic, oblique) and weight (light, bold, black, etc.) *only*. A font with no particular differences in weight or style (e.g. medium weight, not italic and fsSelection bit 6 set) should have the string "Regular" stored in this position.
3	Unique font identifier
4	Full font name; this should simply be a combination of strings 1 and 2. Exception: if string 2 is "Regular," then use only string 1. This is the font name that Windows will expose to users.
5	Version string. In n.nn format.
6	Postscript name for the font.
7	Trademark; this is used to save any trademark notice/information for this font. Such information should be based on legal advice. This is *distinctly* separate from the copyright.

Note that while both Apple and Microsoft support the same set of name strings, the interpretations may be somewhat different. But since name strings are stored by platform, encoding and language (placing separate strings in for both Apple and MS platforms), this should not present a problem.

The key information for this table for MS fonts relates to the use of strings 1, 2 and 4. Some examples:

Helvetica Narrow Oblique	1 = Helvetica Narrow 2 = Oblique 4 = Helvetica Narrow Oblique
Helvetica Narrow	1 = Helvetica Narrow 2 = Regular 4 = Helvetica Narrow
Helvetica Narrow Light Italic	1 = Helvetica Narrow 2 = Light Italic 4 = Helvetica Narrow Light Italic

Note that OS/2 and Windows both require that all name strings be defined in Unicode. Thus all 'name' table strings for platform ID = 3 (Microsoft) will require two bytes per character. See the chapter, "Character Sets," for a list of the current Unicode character codes supported by Microsoft. Macintosh fonts require single byte strings.

Examples of how these strings might be defined:

0 The copyright string from the font vendor.
© Copyright the Monotype Corporation plc, 1990

1 The name the user sees.
Times New Roman

2 The name of the style.
Bold

3 A unique identifier that applications can store to identify the font being used.
Monotype: Times New Roman Bold:1990

4 The complete, hopefully unique, human readable name of the font. This name is used by Windows.
Times New Roman Bold

5 Release and version information from the font vendor.
June 1, 1990; 1.00, initial release

6 The name the font will be known by on a PostScript printer.
TimesNewRoman-Bold

7 Trademark string,
Times New Roman is a registered trademark of the Monotype Corporation.

OS/2 - OS/2 and Windows Metrics

The OS/2 table consists of a set of metrics that are required by Windows and OS/2. The layout of this table is as follows:

Type	Name of Entry	Comments
USHORT	version	0x0001
SHORT	xAvgCharWidth;	
USHORT	usWeightClass;	
USHORT	usWidthClass;	
SHORT	fsType;	
SHORT	ySubscriptXSize;	
SHORT	ySubscriptYSize;	
SHORT	ySubscriptXOffset;	
SHORT	ySubscriptYOffset;	
SHORT	ySuperscriptXSize;	
SHORT	ySuperscriptYSize;	
SHORT	ySuperscriptXOffset;	
SHORT	ySuperscriptYOffset;	
SHORT	yStrikeoutSize;	
SHORT	yStrikeoutPosition;	
SHORT	sFamilyClass;	
PANOSE	panose;	
ULONG	ulUnicodeRange1	Bits 0–31
ULONG	ulUnicodeRange2	Bits 32–63
ULONG	ulUnicodeRange3	Bits 64–95
ULONG	ulUnicodeRange4	Bits 96–127
CHAR	achVendID[4];	
USHORT	fsSelection;	
USHORT	usFirstCharIndex	
USHORT	usLastCharIndex	
USHORT	sTypoAscender	
USHORT	sTypoDescender	
USHORT	sTypoLineGap	
USHORT	usWinAscent	
USHORT	usWinDescent	
ULONG	ulCodePageRange1	Bits 0-31
ULONG	ulCodePageRange2	Bits 32-63

version

Format: 2-byte unsigned short

Units: n/a

Title: OS/2 table version number.

Description: The version number for this OS/2 table.

Comments: The version number allows for identification of the precise contents and layout for the OS/2 table. The version number for this layout is one (1). The version number for the previous layout (in rev.1.5 of this spec and earlier) was zero (0). Version 0 of the OS/2 table was 78 bytes; Version 1 is 86 bytes, having added the ulCodePageRange1 and ulCodePageRange2 fields.

xAvgCharWidth

Format: 2-byte signed short

Units: Pels / em units

Title: Average weighted escapement.

Description: The Average Character Width parameter specifies the arithmetic average of the escapement (width) of all of the 26 lowercase letters a through z of the Latin alphabet and the space character. If any of the 26 lowercase letters are not present, this parameter should equal the weighted average of *all* glyphs in the font. For non-UGL (platform 3, encoding 0) fonts, use the unweighted average.

Comments: This parameter is a descriptive attribute of the font that specifies the spacing of characters for comparing one font to another for selection or substitution. For proportionally spaced fonts, this value is useful in estimating the length for lines of text. The weighting factors provided with this example are only valid for Latin lowercase letters. If other character sets, or capital letters are used, different frequency of use values should be used. One needs to be careful when comparing fonts that use different frequency of use values for font mapping. The average character width is calculated according to this formula: For the lowercase letters only, sum the individual character widths multiplied by the following weighting factors and then divide by 1000. For example:

Letter	Weight Factor
a	64
b	14
c	27
d	35
e	100
f	20
g	14
h	42
i	63
j	3
k	6
l	35
m	20
n	56
o	56
p	17
q	4
r	49
s	56
t	71
u	31
v	10
w	18
x	3
y	18
z	2
space	166

usWeightClass

Format: 2-byte unsigned short

Title: Weight class.

Description: Indicates the visual weight (degree of blackness or thickness of strokes) of the characters in the font.

Comments:

Value	Description	C Definition (from windows.h)
100	Thin	FW_THIN
200	Extra-light (Ultra-light)	FW_EXTRALIGHT
300	Light	FW_LIGHT
400	Normal (Regular)	FW_NORMAL
500	Medium	FW_MEDIUM
600	Semi-bold (Demi-bold)	FW_SEMIBOLD
700	Bold	FW_BOLD
800	Extra-Bold (Ultra-bold)	FW_EXTRABOLD
900	Black (Heavy)	FW_BLACK

usWidthClass

Format: 2-byte unsigned short

Title: Width class.

Description: Indicates a relative change from the normal aspect ratio (width to height ratio) as specified by a font designer for the glyphs in a font.

Comments:

Value	Description	C Definition	% of normal
1	Ultra-condensed	FWIDTH_ULTRA_CONDENSED	50
2	Extra-condensed	FWIDTH_EXTRA_CONDENSED	62.5
3	Condensed	FWIDTH_CONDENSED	75
4	Semi-condensed	FWIDTH_SEMI_CONDENSED	87.5
5	Medium (normal)	FWIDTH_NORMAL	100
6	Semi-expanded	FWIDTH_SEMI_EXPANDED	112.5
7	Expanded	FWIDTH_EXPANDED	125
8	Extra-expanded	FWIDTH_EXTRA_EXPANDED	150
9	Ultra-expanded	FWIDTH_ULTRA_EXPANDED	200

Although every character in a font may have a different numeric aspect ratio, each character in a font of normal width has a relative aspect ratio of one. When a new type style is created of a different width class (either by a font designer or by some automated means) the relative aspect ratio of the characters in the new font is some percentage greater or less than those same characters in the normal font -- it is this difference that this parameter specifies.

fsType

Format: 2-byte unsigned short

Title: Type flags.

Description: Indicates font embedding licensing rights for the font. Embeddable fonts may be stored in a document. When a document with embedded fonts is opened on a system that does not have the font installed (the remote system), the embedded font may be loaded for temporary (and in some cases, permanent) use on that system by an embedding-aware application. Embedding licensing rights are granted by the vendor of the font.

The TrueType Font Embedding DLL Specification and DLL release notes describe the APIs used to implement support for TrueType font embedding and loading. ***Applications that implement support for font embedding, either through use of the Font Embedding DLL or through other means, must not embed fonts which are not licensed to permit embedding. Further, applications loading embedded fonts for temporary use (see Preview & Print and Editable embedding below) must delete the fonts when the document containing the embedded font is closed.***

Bit	Bit Mask	Description
0		Reserved, must be zero.
1	0x0002	Restricted License embedding: When *only* this bit is set, this font may not be embedded, copied or modified.
2	0x0004	Preview & Print embedding: When this bit is set, the font may be embedded, and temporarily loaded on the remote system. Documents containing Preview & Print fonts must be opened "read-only;" no edits can be applied to the document.
3	0x0008	Editable embedding: When this bit is set, the font may be embedded and temporarily loaded on other systems. Documents containing Editable fonts *may* be opened for reading and writing.
4-15		Reserved, must be zero.

Comments: If multiple embedding bits are set, the *least* restrictive license granted takes precedence. For example, if bits 1 and 3 are set, bit 3 takes precedence over bit 1and the font may be embedded with Editable rights. For compatibility purposes, most vendors granting Editable embedding rights are also setting the Preview & Print bit (0x000C). This will permit an application that only supports Preview & Print embedding to detect that font embedding is allowed.

Restricted License embedding (0x0002): Fonts that have this bit set **must not be modified, embedded or exchanged in any manner** without first obtaining permission of the legal owner. *Caution:* note that for Restricted License embedding to take effect, it must be the only level of embedding selected (as noted in the previous paragraph).

Preview & Print embedding (0x0004): Fonts with this bit set indicate that they may be embedded within documents but must only be installed *temporarily* on the remote system. Any document which includes a Preview & Print embedded font must be opened "read-only;" the application must not allow the user to edit the document; it can only be viewed and/or printed.

Editable embedding (0x0008): Fonts with this bit set indicate that they may be embedded in documents, but must only be installed *temporarily* on the remote system. In contrast to Preview & Print fonts, documents containing Editable fonts may be opened "read-write;" editing is permitted, and changes may be saved.

Installable embedding (0x0000): Fonts with this setting indicate that they may be embedded and permanently installed on the remote system by an application. The user of the remote system acquires the identical rights, obligations and licenses for that font as the original purchaser of the font, and is subject to the same end-user license agreement, copyright, design patent, and/or trademark as was the original purchaser.

ySubscriptXSize

Format: 2-byte signed short

Units: Font design units

Title: Subscript horizontal font size.

Description: The recommended horizontal size in font design units for subscripts for this font.

Comments: If a font has two recommended sizes for subscripts, e.g., numerics and other, the numeric sizes should be stressed. This size field maps to the em square size of the font being used for a subscript. The horizontal font size specifies a font designer's recommended horizontal font size for subscript characters associated with this font. If a font does not include all of the required subscript characters for an application, and the application can substitute characters by scaling the character of a font or by substituting characters from another font, this parameter specifies the recommended em square for those subscript characters.

For example, if the em square for a font is 2048 and ySubScriptXSize is set to 205, then the horizontal size for a simulated subscript character would be 1/10th the size of the normal character.

ySubscriptYSize

Format: 2-byte signed short

Units: Font design units

Title: Subscript vertical font size.

Description: The recommended vertical size in font design units for subscripts for this font.

Comments: If a font has two recommended sizes for subscripts, e.g. numerics and other, the numeric sizes should be stressed. This size field maps to the emHeight of the font being used for a subscript. The horizontal font size specifies a font designer's recommendation for horizontal font size of subscript characters associated with this font. If a font does not include all of the required subscript characters for an application, and the application can substitute characters by scaling the characters in a font or by substituting characters from another font, this parameter specifies the recommended horizontal EmInc for those subscript characters.

For example, if the em square for a font is 2048 and ySubScriptYSize is set to 205, then the vertical size for a simulated subscript character would be 1/10th the size of the normal character.

ySubscriptXOffset

Format: 2-byte signed short

Units: Font design units

Title: Subscript x offset.

Description: The recommended horizontal offset in font design untis for subscripts for this font.

Comments: The Subscript X Offset parameter specifies a font designer's recommended horizontal offset -- from the character origin of the font to the character origin of the subscript's character -- for subscript characters associated with this font. If a font does not include all of the required subscript characters for an application, and the application can substitute characters, this parameter specifies the recommended horizontal position from the character escapement point of the last character before the first subscript character. For upright characters, this value is usually zero; however, if the characters of a font have an incline (italic characters) the reference point for subscript characters is usually adjusted to compensate for the angle of incline.

ySubscriptYOffset

Format: 2-byte signed short

Units: Font design units

Title: Subscript y offset.

Description: The recommended vertical offset in font design units from the baseline for subscripts for this font.

Comments: The Subscript Y Offset parameter specifies a font designer's recommended vertical offset from the character baseline to the character baseline for subscript characters associated with this font. Values are expressed as a positive offset below the character baseline. If a font does not include all of the required subscript for an application, this parameter specifies the recommended vertical distance below the character baseline for those subscript characters.

ySuperscriptXSize

Format: 2-byte signed short

Units: Font design units

Title: Superscript horizontal font size.

Description: The recommended horizontal size in font design units for superscripts for this font.

Comments: If a font has two recommended sizes for subscripts, e.g., numerics and other, the numeric sizes should be stressed. This size field maps to the em square size of the font being used for a subscript. The horizontal font size specifies a font designer's recommended horizontal font size for superscript characters associated with this font. If a font does not include all of the required superscript characters for an application, and the application can substitute characters by scaling the character of a font or by substituting characters from another font, this parameter specifies the recommended em square for those superscript characters.

For example, if the em square for a font is 2048 and ySuperScriptXSize is set to 205, then the horizontal size for a simulated superscript character would be 1/10th the size of the normal character.

ySuperscriptYSize

Format: 2-byte signed short

Units: Font design units

Title: Superscript vertical font size.

Description: The recommended vertical size in font design units for superscripts for this font.

Comments: If a font has two recommended sizes for subscripts, e.g., numerics and other, the numeric sizes should be stressed. This size field maps to the emHeight of the font being used for a subscript. The vertical font size specifies a font designer's recommended vertical font size for superscript characters associated with this font. If a font does not include all of the required superscript characters for an application, and the application can substitute characters by scaling the character of a font or by substituting characters from another font, this parameter specifies the recommended EmHeight for those superscript characters.

For example, if the em square for a font is 2048 and ySuperScriptYSize is set to 205, then the vertical size for a simulated superscript character would be 1/10th the size of the normal character.

ySuperscriptXOffset

Format: 2-byte signed short

Units: Font design units

Title: Superscript x offset.

Description: The recommended horizontal offset in font design units for superscripts for this font.

Comments: The Superscript X Offset parameter specifies a font designer's recommended horizontal offset -- from the character origin to the superscript character's origin for the superscript characters associated with this font. If a font does not include all of the required superscript characters for an application, this parameter specifies the recommended horizontal position from the escapement point of the character before the first superscript character. For upright characters, this value is usually zero; however, if the characters of a font have an incline (italic characters) the reference point for superscript characters is usually adjusted to compensate for the angle of incline.

ySuperscriptYOffset

Format: 2-byte signed short

Units: Font design units

Title: Superscript y offset.

Description: The recommended vertical offset in font design units from the baseline for superscripts for this font.

Comments: The Superscript Y Offset parameter specifies a font designer's recommended vertical offset -- from the character baseline to the superscript character's baseline associated with this font. Values for this parameter are expressed as a positive offset above the character baseline. If a font does not include all of the required superscript characters for an application, this parameter specifies the recommended vertical distance above the character baseline for those superscript characters.

yStrikeoutSize

Format:	2-byte signed short
Units:	Font design units
Title:	Strikeout size.
Description:	Width of the strikeout stroke in font design units.
Comments:	This field should normally be the width of the em dash for the current font. If the size is one, the strikeout line will be the line represented by the strikeout position field. If the value is two, the strikeout line will be the line represented by the strikeout position and the line immediately *above* the strikeout position. For a Roman font with a 2048 em square, 102 is suggested.

yStrikeoutPosition

Format:	2-byte signed short
Units:	Font design units
Title:	Strikeout position.
Description:	The position of the strikeout stroke relative to the baseline in font design units.
Comments:	Positive values represent distances above the baseline, while negative values represent distances below the baseline. A value of zero falls directly on the baseline, while a value of one falls one pel above the baseline. The value of strikeout position should not interfere with the recognition of standard characters, and therefore should not line up with crossbars in the font. For a Roman font with a 2048 em square, 530 is suggested.

sFamilyClass

Format:	2-byte signed short
Title:	Font-family class and subclass. Also see section 3.4.
Description:	This parameter is a classification of font-family design.
Comments:	The font class and font subclass are registered values assigned by IBM to each font family. This parameter is intended for use in selecting an alternate font when the requested font is not available. The font class is the most general and the font subclass is the most specific. The high byte of this field contains the family class, while the low byte contains the family subclass.

See Appendix A for full information about this field.

Panose

Format: 10 byte array

Title: PANOSE classification number

International: Additional specifications are required for PANOSE to classify non Latin character sets.

Description: This 10 byte series of numbers are used to describe the visual characteristics of a given typeface. These characteristics are then used to associate the font with other fonts of similar appearance having different names. The variables for each digit are listed below. The specifications for each variable can be obtained in the specification *PANOSE v2.0 Numerical Evaluation* from Microsoft or Elseware Corporation.

Comments: The PANOSE definition contains ten digits each of which currently describes up to sixteen variations. Windows v3.1 uses bFamilyType, bSerifStyle and bProportion in the font mapper to determine family type. It also uses bProportion to determine if the font is monospaced.

Type	Name
BYTE	bFamilyType;
BYTE	bSerifStyle;
BYTE	bWeight;
BYTE	bProportion;
BYTE	bContrast;
BYTE	bStrokeVariation;
BYTE	bArmStyle;
BYTE	bLetterform;
BYTE	bMidline;
BYTE	bXHeight;

1. Family Kind (6 variations)
   - 0 = Any
   - 1 = No Fit
   - 2 = Text and Display
   - 3 = Script
   - 4 = Decorative
   - 5 = Pictorial
2. Serif Style (16 variations)
   - 0 = Any
   - 1 = No Fit
   - 2 = Cove
   - 3 = Obtuse Cove
   - 4 = Square Cove
   - 5 = Obtuse Square Cove
   - 6 = Square
   - 7 = Thin
   - 8 = Bone

9 = Exaggerated
10 = Triangle
11 = Normal Sans
12 = Obtuse Sans
13 = Perp Sans
14 = Flared
15 = Rounded

3. Weight (12 variations)
0 = Any
1 = No Fit
2 = Very Light
3 = Light
4 = Thin
5 = Book
6 = Medium
7 = Demi
8 = Bold
9 = Heavy
10 = Black
11 = Nord

4. Proportion (10 variations)
0 = Any
1 = No Fit
2 = Old Style
3 = Modern
4 = Even Width
5 = Expanded
6 = Condensed
7 = Very Expanded
8 = Very Condensed
9 = Monospaced

5. Contrast (10 variations)
    - 0 = Any
    - 1 = No Fit
    - 2 = None
    - 3 = Very Low
    - 4 = Low
    - 5 = Medium Low
    - 6 = Medium
    - 7 = Medium High
    - 8 = High
    - 9 = Very High

6. Stroke Variation (9 variations)
    - 0 = Any
    - 1 = No Fit
    - 2 = Gradual/Diagonal
    - 3 = Gradual/Transitional
    - 4 = Gradual/Vertical
    - 5 = Gradual/Horizontal
    - 6 = Rapid/Vertical
    - 7 = Rapid/Horizontal
    - 8 = Instant/Vertical

7. Arm Style (12 variations)
    - 0 = Any
    - 1 = No Fit
    - 2 = Straight Arms/Horizontal
    - 3 = Straight Arms/Wedge
    - 4 = Straight Arms/Vertical
    - 5 = Straight Arms/Single Serif
    - 6 = Straight Arms/Double Serif
    - 7 = Non-Straight Arms/Horizontal
    - 8 = Non-Straight Arms/Wedge
    - 9 = Non-Straight Arms/Vertical
    - 10 = Non-Straight Arms/Single Serif
    - 11 = Non-Straight Arms/Double Serif

8. Letterform (16 variations)
    - 0 = Any
    - 1 = No Fit
    - 2 = Normal/Contact
    - 3 = Normal/Weighted
    - 4 = Normal/Boxed
    - 5 = Normal/Flattened
    - 6 = Normal/Rounded
    - 7 = Normal/Off Center
    - 8 = Normal/Square
    - 9 = Oblique/Contact
    - 10 = Oblique/Weighted
    - 11 = Oblique/Boxed
    - 12 = Oblique/Flattened
    - 13 = Oblique/Rounded
    - 14 = Oblique/Off Center
    - 15 = Oblique/Square

9. Midline (14 variations)
    - 0 = Any
    - 1 = No Fit
    - 2 = Standard/Trimmed
    - 3 = Standard/Pointed
    - 4 = Standard/Serifed
    - 5 = High/Trimmed
    - 6 = High/Pointed
    - 7 = High/Serifed
    - 8 = Constant/Trimmed
    - 9 = Constant/Pointed
    - 10 = Constant/Serifed
    - 11 = Low/Trimmed
    - 12 = Low/Pointed
    - 13 = Low/Serifed

10. X-height (8 variations)
    - 0 = Any
    - 1 = No Fit
    - 2 = Constant/Small
    - 3 = Constant/Standard
    - 4 = Constant/Large
    - 5 = Ducking/Small
    - 6 = Ducking/Standard
    - 7 = Ducking/Large

ulUnicodeRange1 (Bits 0–31)
ulUnicodeRange2 (Bits 32–63)
ulUnicodeRange3 (Bits 64–95)
ulUnicodeRange4 (Bits 96–127)

Format*:* 32-bit unsigned long (4 copies) totaling 128 bits.

Title*:* Unicode Character Range

Description*:* This field is used to specify the Unicode blocks or ranges encompassed by the font file in the 'cmap' subtable for platform 3, encoding ID 1 (Microsoft platform). If the bit is set (1) then the Unicode range is considered functional. If the bit is clear (0) then the range is not considered functional. Each of the bits is treated as an independent flag and the bits can be set in any combination. The determination of "functional" is left up to the font designer, although character set selection should attempt to be functional by ranges if at all possible.

All reserved fields must be zero. Each long is in Big-Endian form. See the Basic Multilingual Plane of ISO/IEC 10646-1 or the Unicode Standard v.1.1 for the list of Unicode ranges and characters.

Bit	Description
0	Basic Latin
1	Latin-1 Supplement
2	Latin Extended-A
3	Latin Extended-B
4	IPA Extensions
5	Spacing Modifier Letters
6	Combining Diacritical Marks
7	Basic Greek
8	Greek Symbols And Coptic
9	Cyrillic
10	Armenian
11	Basic Hebrew
12	Hebrew Extended (A and B blocks combined)
13	Basic Arabic
14	Arabic Extended
15	Devanagari
16	Bengali
17	Gurmukhi
18	Gujarati
19	Oriya
20	Tamil
21	Telugu
22	Kannada
23	Malayalam
24	Thai

Table continued from previous page

Bit	Description
25	Lao
26	Basic Georgian
27	Georgian Extended
28	Hangul Jamo
29	Latin Extended Additional
30	Greek Extended
31	General Punctuation
32	Superscripts And Subscripts
33	Currency Symbols
34	Combining Diacritical Marks For Symbols
35	Letterlike Symbols
36	Number Forms
37	Arrows
38	Mathematical Operators
39	Miscellaneous Technical
40	Control Pictures
41	Optical Character Recognition
42	Enclosed Alphanumerics
43	Box Drawing
44	Block Elements
45	Geometric Shapes
46	Miscellaneous Symbols
47	Dingbats
48	CJK Symbols And Punctuation
49	Hiragana
50	Katakana
51	Bopomofo
52	Hangul Compatibility Jamo
53	CJK Miscellaneous
54	Enclosed CJK Letters And Months
55	CJK Compatibility
56	Hangul
57	Reserved for Unicode SubRanges
58	Reserved for Unicode SubRanges
59	CJK Unified Ideographs
60	Private Use Area
61	CJK Compatibility Ideographs
62	Alphabetic Presentation Forms
63	Arabic Presentation Forms-A
64	Combining Half Marks
65	CJK Compatibility Forms

Table continued from previous page

Bit	Description
66	Small Form Variants
67	Arabic Presentation Forms-B
68	Halfwidth And Fullwidth Forms
69	Specials
70–127	Reserved for Unicode SubRanges

achVendID

Format 4-byte character array

Title: Font Vendor Identification

Description: The four character identifier for the vendor of the given type face.

Comments: This is not the royalty owner of the original artwork. This is the company responsible for the marketing and distribution of the typeface that is being classified. It is reasonable to assume that there will be 6 vendors of ITC Zapf Dingbats for use on desktop platforms in the near future (if not already). It is also likely that the vendors will have other inherent benefits in their fonts (more kern pairs, unregularized data, hand hinted, etc.). This identifier will allow for the correct vendor's type to be used over another, possibly inferior, font file. The Vendor ID value is not required.

Microsoft has assigned values for some font suppliers as listed below. Uppercase vendor ID's are reserved by Microsoft. Other suppliers can choose their own mixed case or lowercase ID's, or leave the field blank.

Vendor ID	Vendor Name
AGFA	AGFA Compugraphic
ADBE	Adobe
APPL	Apple
ALTS	Altsys
B&H	Bigelow & Holmes
BERT	Berthold
BITS	Bitstream
CANO	Canon
CTDL	China Type Design Ltd.
DTC	Digital Typeface Corp.
ELSE	Elseware
EPSN	Epson
GLYF	Glyph Systems
GPI	Gamma Productions, Inc.
HP	Hewlett-Packard
HY	HanYang System
IBM	IBM
IMPR	Impress
KATF	Kingsley/ATF
LANS	Lanston Type Co., Ltd.
LEAF	Interleaf, Inc.
LETR	Letraset

Vendor ID	Vendor Name
LINO	Linotype
LTRX	Lighttracks
MACR	Macromedia
MONO	Monotype
MLGC	Micrologic Software
MS	Microsoft
NEC	NEC
PARA	ParaGraph Intl.
PRFS	Production First Software
QMSI	QMS/Imagen
SFUN	Soft Union
SWFT	Swfte International
TILD	SIA Tilde
URW	URW
ZSFT	ZSoft

fsSelection

Format: 2-byte bit field.
Title: Font selection flags.
Description: Contains information concerning the nature of the font patterns, as follows:

Bit #	macStyle bit	C definition	Description
0	bit 1	ITALIC	Font contains Italic characters, otherwise they are upright.
1		UNDERSCORE	Characters are underscored.
2		NEGATIVE	Characters have their foreground and background reversed.
3		OUTLINED	Outline (hollow) characters, otherwise they are solid.
4		STRIKEOUT	Characters are overstruck.
5	bit 0	BOLD	Characters are emboldened.
6		REGULAR	Characters are in the standard weight/style for the font.

Comments: All undefined bits must be zero.

This field contains information on the original design of the font. Bits 0 & 5 can be used to determine if the font was designed with these features or whether some type of machine simulation was performed on the font to achieve this appearance. Bits 1-4 are rarely used bits that indicate the font is primarily a decorative or special purpose font.

If bit 6 is set, then bits 0 and 5 must be clear, else the behavior is undefined. As noted above, the settings of bits 0 and 1 must be reflected in the macStyle bits in the 'head' table. While bit 6 on implies that bits 0 and 1 of macStyle are clear (along with bits 0 and 5 of fsSelection), the reverse is not true. Bits 0 and 1 of macStyle (and 0 and 5 of fsSelection) may be clear and that does not give any indication of whether or not bit 6 of fsSelection is clear (e.g., Arial Light would have all bits cleared; it is not the regular version of Arial).

usFirstCharIndex

Format: 2-byte USHORT

Description: The minimum Unicode index (character code) in this font, according to the cmap subtable for platform ID 3 and encoding ID 0 or 1. For most fonts supporting Win-ANSI or other character sets, this value would be 0x0020.

usLastCharIndex

Format: 2-byte USHORT

Description: The maximum Unicode index (character code) in this font, according to the cmap subtable for platform ID 3 and encoding ID 0 or 1. This value depends on which character sets the font supports.

sTypoAscender

Format: 2-byte SHORT

Description: The typographic ascender for this font. Remember that this is not the same as the Ascender value in the 'hhea' table, which Apple defines in a far different manner. One good source for usTypoAscender is the Ascender value from an AFM file.

The suggested useage for usTypoAscender is that it be used in conjunction with unitsPerEm to compute a typographically correct default line spacing. The goal is to free applications from Macintosh or Windows-specific metrics which are constrained by backward compatability requirements. These new metrics, when combined with the character design widths, will allow applications to lay out documents in a typographically correct and portable fashion. These metrics will be exposed through Windows APIs. Macintosh applications will need to access the 'sfnt' resource and parse it to extract this data from the "OS/2" table (unless Apple exposes the 'OS/2' table through a new API).

sTypoDescender

Format: 2-byte SHORT

Description: The typographic descender for this font. Remember that this is not the same as the Descender value in the 'hhea' table, which Apple defines in a far different manner. One good source for usTypoDescender is the Descender value from an AFM file.

The suggested useage for usTypoDescender is that it be used in conjunction with unitsPerEm to compute a typographically correct default line spacing. The goal is to free applications from Macintosh or Windows-specific metrics which are constrained by backward compatability requirements. These new metrics, when combined with the character design widths, will allow applications to lay out documents in a typographically correct and portable fashion. These metrics will be exposed through Windows APIs. Macintosh applications will need to access the 'sfnt' resource and parse it to extract this data from the "OS/2" table (unless Apple exposes the 'OS/2' table through a new API).

sTypoLineGap

Format: 2-byte SHORT

Description: The typographic line gap for this font. Remember that this is not the same as the LineGap value in the 'hhea' table, which Apple defines in a far different manner.

The suggested useage for usTypoLineGap is that it be used in conjunction with unitsPerEm to compute a typographically correct default line spacing. Typical values average 7-10% of units per em. The goal is to free applications from Macintosh or Windows-specific metrics which are constrained by backward compatability requirements (see chapter, "Recommendations for Windows Fonts). These new metrics, when combined with the character design widths, will allow applications to lay out documents in a typographically correct and portable fashion. These metrics will be exposed through Windows APIs. Macintosh applications will need to access the 'sfnt' resource and parse it to extract this data from the "OS/2" table (unless Apple exposes the 'OS/2' table through a new API).

usWinAscent

Format: 2-byte USHORT

Description: The ascender metric for Windows. This, too, is distinct from Apple's Ascender value and from the usTypoAscender values. usWinAscent is computed as the yMax for all characters in the Windows ANSI character set. usTypoAscent is used to compute the Windows font height and default line spacing. For platform 3 encoding 0 fonts, it is the same as yMax.

usWinDescent

Format: 2-byte USHORT

Description: The descender metric for Windows. This, too, is distinct from Apple's Descender value and from the usTypoDescender values. usWinDescent is computed as the -yMin for all characters in the Windows ANSI character set. usTypoAscent is used to compute the Windows font height and default line spacing. For platform 3 encoding 0 fonts, it is the same as -yMin.

ulCodePageRange1 *Bits 0–31*
ulCodePageRange2 *Bits 32–63*

Format*:* 32-bit unsigned long (2 copies) totaling 64 bits.

Title*:* Code Page Character Range

Description*:* This field is used to specify the code pages encompassed by the font file in the 'cmap' subtable for platform 3, encoding ID 1 (Microsoft platform). If the font file is encoding ID 0, then the Symbol Character Set bit should be set. If the bit is set (1) then the code page is considered functional. If the bit is clear (0) then the code page is not considered functional. Each of the bits is treated as an independent flag and the bits can be set in any combination. The determination of "functional" is left up to the font designer, although character set selection should attempt to be functional by code pages if at all possible.

Symbol character sets have a special meaning. If the symbol bit (31) is set, and the font file contains a 'cmap' subtable for platform of 3 and encoding ID of 1, then all of the characters in the Unicode range 0xF000 - 0xF0FF (inclusive) will be used to enumerate the symbol character set. If the bit is not set, any characters present in that range will not be enumerated as a symbol character set.

All reserved fields must be zero. Each long is in Big-Endian form.

Bit	Code Page	Description
0	1252	Latin 1
1	1250	Latin 2: Eastern Europe
2	1251	Cyrillic
3	1253	Greek
4	1254	Turkish
5	1255	Hebrew
6	1256	Arabic
7	1257	Windows Baltic
8–15		Reserved for Alternate ANSI
16	874	Thai
17	932	JIS/Japan
18	936	Chinese: Simplified chars--PRC and Singapore
19	949	Korean Wansung
20	950	Chinese: Traditional chars--Taiwan and Hong Kong
21	1361	Korean Johab
22–28		Reserved for Alternate ANSI & OEM
29		Macintosh Character Set (US Roman)
30		OEM Character Set
31		Symbol Character Set
32-47		Reserved for OEM
48	869	IBM Greek
49	866	MS-DOS Russian
50	865	MS-DOS Nordic
51	864	Arabic
52	863	MS-DOS Canadian French

Bit	Code Page	Description
53	862	Hebrew
54	861	MS-DOS Icelandic
55	860	MS-DOS Portuguese
56	857	IBM Turkish
57	855	IBM Cyrillic; primarily Russian
58	852	Latin 2
59	775	MS-DOS Baltic
60	737	Greek; former 437 G
61	708	Arabic; ASMO 708
62	850	WE/Latin 1
63	437	US

PCLT - PCL 5 Table

The 'PCLT' table is an optional table that is not used directly by Microsoft Windows v3.1, but it is highly recommended that this table be present in all TrueType font files. Extra information on many of these fields can be found in the *HP PCL 5 Printer Language Technical Reference Manual* available from Hewlett-Packard Boise Printer Division.

The format for the table is:

Type	Name of Entry
FIXED	Version
ULONG	FontNumber
USHORT	Pitch
USHORT	xHeight
USHORT	Style
USHORT	TypeFamily
USHORT	CapHeight
USHORT	SymbolSet
CHAR	Typeface[16]
CHAR	CharacterComplement[8]
CHAR	FileName[6]
CHAR	StrokeWeight
CHAR	WidthType
BYTE	SerifStyle
BYTE	Reserved (pad)

Version

Table version number 1.0 is represented as 0x00010000.

FontNumber

This 32-bit number is segmented in two parts. The most significant bit indicates native versus converted format. Only font vendors should create fonts with this bit zeroed. The 7 next most significant bits are assigned by Hewlett-Packard Boise Printer Division to major font vendors. The least significant 24 bits are assigned by the vendor. Font vendors should attempt to insure that each of their fonts are marked with unique values.

Vendor codes:

A	Adobe Systems
B	Bitstream Inc.
C	Agfa Corporation
H	Bigelow & Holmes
L	Linotype Company
M	Monotype Typography Ltd.

Pitch

The width of the space in FUnits (FUnits are described by the unitsPerEm field of the 'head' table). Monospace fonts derive the width of all characters from this field.

xHeight

The height of the optical line describing the height of the lowercase x in FUnits. This might not be the same as the measured height of the lowercase x.

Style

The most significant 6 bits are reserved. The 5 next most significant bits encode structure. The next 3 most significant bits encode appearance width. The 2 least significant bits encode posture.

Structure (bits 5-9)

0	Solid (normal, black)
1	Outline (hollow)
2	Inline (incised, engraved)
3	Contour, edged (antique, distressed)
4	Solid with shadow
5	Outline with shadow
6	Inline with shadow
7	Contour, or edged, with shadow
8	Pattern filled
9	Pattern filled #1 (when more than one pattern)
10	Pattern filled #2 (when more than two patterns)
11	Pattern filled #3 (when more than three patterns)
12	Pattern filled with shadow
13	Pattern filled with shadow #1 (when more than one pattern or shadow)
14	Pattern filled with shadow #2 (when more than two patterns or shadows)
15	Pattern filled with shadow #3 (when more than three patterns or shadows)
16	Inverse
17	Inverse with border
18-31	reserved

Width (bits 2-4)

0	normal
1	condensed
2	compressed, extra condensed
3	extra compressed
4	ultra compressed
5	reserved
6	expanded, extended
7	extra expanded, extra extended

Posture (bits 0-1)

0	upright
1	oblique, italic
2	alternate italic (backslanted, cursive, swash)
3	reserved

TypeFamily

The 4 most significant bits are font vendor codes. The 12 least significant bits are typeface family codes. Both are assigned by HP Boise Division.

Vendor Codes (bits 12-15)

0	reserved
1	Agfa Corporation
2	Bitstream Inc.
3	Linotype Company
4	Monotype Typography Ltd.
5	Adobe Systems
6	font repackagers
7	vendors of unique typefaces
8-15	reserved

CapHeight

The height of the optical line describing the top of the uppercase H in FUnits. This might not be the same as the measured height of the uppercase H.

SymbolSet

The most significant 11 bits are the value of the symbol set “number” field. The value of the least significant 5 bits, when added to 64, is the ASCII value of the symbol set “ID” field. Symbol set values are assigned by HP Boise Division. Unbound fonts, or “typefaces” should have a symbol set value of 0. See the *PCL 5 Printer Language Technical Reference Manual* or the *PCL 5 Comparison Guide* for the most recent published list of codes.

Examples

	PCL	decimal
Windows 3.1 “ANSI”	19U	629
Windows 3.0 “ANSI”	9U	309
Adobe “Symbol”	19M	621
Macintosh	12J	394
PostScript ISO Latin 1	11J	362
PostScript Std. Encoding	10J	330
Code Page 1004	9J	298
DeskTop	7J	234

TypeFace

This 16-byte ASCII string appears in the “font print” of PCL printers. Care should be taken to insure that the base string for all typefaces of a family are consistent, and that the designators for bold, italic, etc. are standardized.

Example:

```
Times New
Times New       Bd
Times New       It
Times New     BdIt
Courier New
Courier New     Bd
Courier New     It
Courier New   BdIt
```

CharacterComplement

This 8-byte field identifies the symbol collections provided by the font, each bit identifies a symbol collection and is independently interpreted. Symbol set bound fonts should have this field set to all F’s (except bit 0).

Example:

DOS/PCL Complement	0xFFFFFFFF003FFFFE
Windows 3.1 “ANSI”	0xFFFFFFFF37FFFFFE
Macintosh	0xFFFFFFFF36FFFFFE
ISO 8859-1 Latin 1	0xFFFFFFFF3BFFFFFE
ISO 8859-1,2,9 Latin 1,2,5	0xFFFFFFFF0BFFFFFE

The character collections identified by each bit are as follows:

31	ASCII (supports several standard interpretations)
30	Latin 1 extensions
29	Latin 2 extensions
28	Latin 5 extensions
27	Desktop Publishing Extensions
26	Accent Extensions (East and West Europe)
25	PCL Extensions
24	Macintosh Extensions
23	PostScript Extensions
22	Code Page Extensions

The character complement field also indicates the index mechanism used with an unbound font. Bit 0 must always be cleared when the font elements are provided in Unicode order.

FileName

This 6-byte field is composed of 3 parts. The first 3 bytes are an industry standard typeface family string. The fourth byte is a treatment character, such as R, B, I. The last two characters are either zeroes for an unbound font or a two character mnemonic for a symbol set if symbol set found.

Examples:

`TNRR00`	Times New (text weight, upright)
`TNRI00`	Times New Italic
`TNRB00`	Times New Bold
`TNRJ00`	Times New Bold Italic
`COUR00`	Courier
`COUI00`	Courier Italic
`COUB00`	Courier Bold
`COUJ00`	Courier Bold Italic

Treatment Flags:

R	Text, normal, book, etc.
I	Italic, oblique, slanted, etc.
B	Bold
J	Bold Italic, Bold Oblique
D	Demibold
E	Demibold Italic, Demibold Oblique
K	Black
G	Black Italic, Black Oblique
L	Light
P	Light Italic, Light Oblique
C	Condensed
A	Condensed Italic, Condensed Oblique
F	Bold Condensed
H	Bold Condensed Italic, Bold Condensed Oblique
S	Semibold (lighter than demibold)
T	Semibold Italic, Semibold Oblique

other treatment flags are assigned over time.

StrokeWeight

This signed 1-byte field contains the PCL stroke weight value. Only values in the range -7 to 7 are valid:

-7	Ultra Thin
-6	Extra Thin
-5	Thin
-4	Extra Light
-3	Light
-2	Demilight
-1	Semilight
0	Book, text, regular, etc.
1	Semibold (Medium, when darker than Book)
2	Demibold
3	Bold
4	Extra Bold
5	Black
6	Extra Black
7	Ultra Black, or Ultra

Type designers often use interesting names for weights or combinations of weights and styles, such as Heavy, Compact, Inserat, Bold No. 2, etc. PCL stroke weights are assigned on the basis of the entire family and use of the faces. Typically, display faces don't have a "text" weight assignment.

WidthType

This signed 1-byte field contains the PCL appearance width value. The values are not directly related to those in the appearance with field of the style word above. Only values in the range -5 to 5 are valid.

-5	Ultra Compressed
-4	Extra Compressed
-3	Compressed, or Extra Condensed
-2	Condensed
0	Normal
2	Expanded
3	Extra Expanded

SerifStyle

This signed 1-byte field contains the PCL serif style value. The most significant 2 bits of this byte specify the serif/sans or contrast/monoline characterisitics of the typeface.

Bottom 6 bit values:

0	Sans Serif Square
1	Sans Serif Round
2	Serif Line
3	Serif Triangle
4	Serif Swath
5	Serif Block
6	Serif Bracket
7	Rounded Bracket
8	Flair Serif, Modified Sans
9	Script Nonconnecting
10	Script Joining
11	Script Calligraphic
12	Script Broken Letter

Top 2 bit values:

0	reserved
1	Sans Serif/Monoline
2	Serif/Contrasting
3	reserved

Reserved

Should be set to zero.

post - PostScript

This table contains additional information needed to use TrueType fonts on PostScript printers. This includes data for the FontInfo dictionary entry and the PostScript names of all the glyphs.

The table begins as follows:

Type	Name	Description
FIXED	Format Type	0x00010000 for format 1.0, 0x00020000 for format 2.0, and so on...
FIXED	italicAngle	Italic angle in counter-clockwise degrees from the vertical. Zero for upright text, negative for text that leans to the right (forward)
FWORD	underlinePosition	Suggested values for the underline position (negative values indicate below baseline).
FWORD	underlineThickness	Suggested values for the underline thickness.
ULONG	isFixedPitch	Set to 0 if the font is proportionally spaced, non-zero if the font is not proportionally spaced (i.e. monospaced).
ULONG	minMemType42	Minimum memory usage when a TrueType font is downloaded.
ULONG	maxMemType42	Maximum memory usage when a TrueType font is downloaded.
ULONG	minMemType1	Minimum memory usage when a TrueType font is downloaded as a Type 1 font.
ULONG	maxMemType1	Maximum memory usage when a TrueType font is downloaded as a Type 1 font.

The last four entries in the table are present because PostScript drivers can do better memory management if the virtual memory (VM) requirements of a downloadable TrueType font are known before the font is downloaded. This information should be supplied if known. If it is not known, set the value to zero. The driver will still work but will be less efficient.

Maximum memory usage is minimum memory usage plus maximum runtime memory use. Maximum runtime memory use depends on the maximum band size of any bitmap potentially rasterized by the TrueType font scaler. Runtime memory usage could be calculated by rendering characters at different point sizes and comparing memory use.

How to calculate VM usage

The memory usage of a downloaded TrueType font will vary with whether it is defined as a TrueType or Type 1 font on the printer. Minimum memory usage can be calculated by calling *VMStatus*, downloading the font, and calling *VMStatus* a second time.

If the format is 1.0 or 3.0, the table ends here. The additional entries for formats 2.0 and 2.5 are shown below. Apple has defined a format 4.0 for use with QuickDraw GX, which is described in their documentation.

Format 1.0

This TrueType font file contains exactly the 258 glyphs in the standard Macintosh TrueType font file in the order specified in Appendix C, "Standard Macintosh Character Set to UGL." As a result, the glyph names are taken from the system with no storage required by the font.

Format 2.0

This is the format required by Microsoft fonts.

Type	Description
USHORT	Number of glyphs (this is the same as numGlyphs in 'maxp' table).
USHORT	glyphNameIndex[numGlyphs].
CHAR	Glyph names with length bytes [variable] (a Pascal string).

This TrueType font file contains glyphs not in the standard Macintosh set or the ordering of the glyphs in the TrueType font file is non-standard (again, for the Macintosh). The glyph name array maps the glyphs in this font to name index. If the name index is between 0 and 257, treat the name index as a glyph index in the Macintosh standard order. If the name index is between 258 and 32767, then subtract 258 and use that to index into the list of Pascal strings at the end of the table. Thus a given font may map some of its glyphs to the standard glyph names, and some to its own names.

Index numbers 32768 through 65535 are reserved for future use. If you do not want to associate a PostScript name with a particular glyph, use index number 0 which points the name *.notdef*.

Format 2.5

This format provides a space saving table for fonts which contain a pure subset of, or a simple reordering of, the standard Macintosh glyph set.

Type	Description
CHAR	offset[numGlyphs]

This format is useful for font files that contain only glyphs in the standard Macintosh glyph set but which have those glyphs arranged in a non-standard order or which are missing some glyphs. The table contains one byte for each glyph in the font file. The byte is treated as a signed offset that maps the glyph index used in this font into the standard glyph index. In other words, assuming that the 'sfnt' contains the three glyphs A, B, and C which are the 37th, 38th, and 39th glyphs in the standard ordering, the 'post' table would contain the bytes +36, +36, +36.

Format 3.0

This format makes it possible to create a special font that is not burdened with a large 'post' table set of glyph names.

This format specifies that no PostScript name information is provided for the glyphs in this font file. The printing behavior of this format on PostScript printers is unspecified, except that it should not result in a fatal or unrecoverable error. Some drivers may print nothing, other drivers may attempt to print using a default naming scheme.

Windows v3.1 makes use of the italic angle value in the 'post' table but does not actually ***require*** *any glyph names to be stored as Pascal strings .*

prep - Control Value Program

The Control Value Program consists of a set of TrueType instructions that will be executed whenever the font or point size or transformation matrix change and before each glyph is interpreted. Any instruction is legal in the CVT Program but since no glyph is associated with it, instructions intended to move points within a particular glyph outline cannot be used in the CVT Program. The name ‘prep’ is anachronistic.

Type	Description
BYTE[ ]	Set of instructions executed whenever point size or font or transformation change

VDMX - Vertical Device Metrics

Under Windows, the usWinAscent and usWinDescent values from the 'OS/2' table will be used to determine the maximum black height for a font at any given size. Windows calls this distance the Font Height. Because TrueType instructions can lead to Font Heights that differ from the actual scaled and rounded values, basing the Font Height strictly on the yMax and yMin can result in "lost pixels." Windows will clip any pixels that extend above the yMax or below the yMin. In order to avoid grid fitting the entire font to determine the correct height, the VDMX table has been defined.

The VDMX table consists of a header followed by groupings of VDMX records:

Type	Name	Description
USHORT	version	Version number (starts at 0).
USHORT	numRecs	Number of VDMX groups present
USHORT	numRatios	Number of aspect ratio groupings
Ratios	ratRange[numRatios]	Ratio ranges (see below for more info)
USHORT	offset[numRatios]	Offset from start of this table to the VDMX group for this ratio range.
Vdmx	groups	The actual VDMX groupings (documented below)

```
struct Ratios {
       BYTE   bCharSet;     /* Character set (see below) */
       BYTE   xRatio;       /* Value to use for x-Ratio */
       BYTE   yStartRatio;  /* Starting y-Ratio value */
       BYTE   yEndRatio     /* Ending y-ratio value */
}
```

Ratios are set up as follows:

For a 1:1 aspect ratio	Ratios.xRatio = 1; Ratios.yStartRatio = 1; Ratios.yEndRatio = 1;
For 1:1 through 2:1 ratio	Ratios.xRatio = 2; Ratios.yStartRatio = 1; Ratios.yEndRatio = 2;
For 1.33:1 ratio	Ratios.xRatio = 4; Ratios.yStartRatio = 3; Ratios.yEndRatio = 3;
For *all* aspect ratios	Ratio.xRatio = 0; Ratio.yStartRatio = 0; Ratio.yEndRatio = 0;

All values set to zero signal the default grouping to use; if present, this must be the *last* Ratio group in the table. Ratios of 2:2 are the same as 1:1.

Aspect ratios are matched against the target device by normalizing the entire ratio range record based on the current X resolution and performing a range check of Y resolutions for each record after normalization. Once a match is found, the search stops. If the 0,0,0 group is encountered during the search, it is used (therefore if this group is not at the end of the ratio groupings, no group that follows it will be used). If there is not a match and there is no 0,0,0 record, then there is no VDMX data for that aspect ratio.

Note that range checks are conceptually performed as follows:

```
(deviceXRatio == Ratio.xRatio) && (deviceYRatio >=
        Ratio.yStartRatio) && (deviceYRatio <= Ratio.yEndRatio)
```

Each ratio grouping refers to a specific VDMX record group; there must be at least 1 VDMX group in the table.

The uCharSet value is used to denote cases where the VDMX group was computed based on a subset of the glyphs present in the font file. The currently defined values for character set are:

uCharSet	Description
0	No subset; the VDMX group applies to all glyphs in the font. This is used for symbol or dingbat fonts.
1	Windows ANSI subset; the VDMX group was computed using only the glyphs required to complete the Windows ANSI character set. Windows will ignore any VDMX entries that are not for the ANSI subset (i.e. uCharSet = 1)

VDMX groups immediately follow the table header. Each set of records (there need only be one set) has the following layout:

Type	Name	Description
USHORT	recs	Number of height records in this group
BYTE	startsz	Starting yPelHeight
BYTE	endsz	Ending yPelHeight
vTable	entry[recs]	The VDMX records

```
struct vTable {
       USHORT yPelHeight;   /* yPelHeight to which values apply */
       SHORT  yMax;         /* yMax (in pels) for this yPelHeight */
       SHORT  yMin;         /* yMin (in pels) for this yPelHeight */
}
```

This table must appear in sorted order (sorted by yPelHeight), but need not be continous. It should have an entry for every pel height where the yMax and yMin do not scale linearly, where linearly scaled heights are defined as:

Hinted yMax and yMin are identical to scaled/rounded yMax and yMin

It is assumed that once yPelHeight reaches 255, all heights will be linear, or at least close enough to linear that it no longer matters. Please note that while the Ratios structure can only support ppem sizes up to 255, the vTable structure can support much larger pel heights (up to 65535). The choice of SHORT and USHORT for vTable is dictated by the requirement that yMax and yMin be signed values (and 127 to -128 is too small a range) and the desire to word-align the vTable elements.

vhea - Vertical Header Table

The vertical header table (tag name: ‘vhea’) contains information needed for vertical fonts. The glyphs of vertical fonts are written either top to bottom or bottom to top. This table contains information that is general to the font as a whole. Information that pertains to specific glyphs is given in the vertical metrics table (tag name: ‘vmtx’) described separately. The formats of these tables are similar to those for horizontal metrics (hhea and hmtx).

Data in the vertical header table must be consistent with data that appears in the vertical metrics table. The advance height and top sidebearing values in the vertical metrics table must correspond with the maximum advance height and minimum bottom sidebearing values in the vertical header table.

The vertical header table format follows:

Vertical Header Table

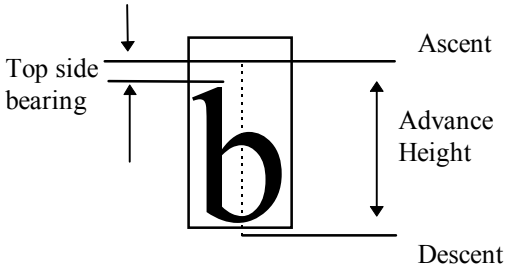
Type	Name	Description
FIXED32	version	Version number of the vertical header table (0x00010000 for the initial version).
SHORT	ascent	Distance in FUnits from the centerline to the previous line's descent.
SHORT	descent	Distance in FUnits from the centerline to the next line's ascent.
SHORT	lineGap	Reserved; set to 0
SHORT	advanceHeightMax	The maximum advance height measurement in FUnits found in the font. This value must be consistent with the entries in the vertical metrics table.
SHORT	minTopSideBearing	The minimum top sidebearing measurement found in the font, in FUnits. This value must be consistent with the entries in the vertical metrics table.
SHORT	minBottomSideBearing	The minimum bottom sidebearing measurement found in the font, in FUnits. This value must be consistent with the entries in the vertical metrics table.
SHORT	yMaxExtent	Defined as yMaxExtent=minTopSideBearing+(yMax-yMin)
SHORT	caretSlopeRise	The value of the caretSlopeRise field divided by the value of the caretSlopeRun Field determines the slope of the caret. A value of 0 for the rise and a value of 1 for the run specifies a horizontal caret. A value of 1 for the rise and a value of 0 for the run specifies a vertical caret. Intermediate values are desirable for fonts whose glyphs are oblique or italic. For a vertical font, a horizontal caret is best.
SHORT	caretSlopeRun	See the caretSlopeRise field. Value=1 for nonslanted vertical fonts.
SHORT	caretOffset	The amount by which the highlight on a slanted glyph needs to be shifted away from the glyph in order to produce the best appearance. Set value equal to 0 for nonslanted fonts.
SHORT	reserved	Set to 0.
SHORT	reserved	Set to 0.
SHORT	reserved	Set to 0.
SHORT	reserved	Set to 0.
SHORT	metricDataFormat	Set to 0.
USHORT	numOfLongVerMetrics	Number of advance heights in the vertical metrics table.

Vertical Header Table Example

Offset/ length	Value	Name	Comment
0/4	0x00010000	version	Version number of the vertical header table, in fixed-point format, is 1.0
4/2	1024	ascent	Half the em-square height.
6/2	-1024	descent	Minus half the em-square height.
8/2	0	lineGap	Typographic line gap is 0 FUnits.
10/2	2079	advanceHeightMax	The maximum advance height measurement found in the font is 2079 FUnits.
12/2	-342	minTopSideBearing	The minimum top sidebearing measurement found in the font is -342 FUnits.
14/2	-333	minBottomSideBearing	The minimum bottom sidebearing measurement found in the font is -333 FUnits.
16/2	2036	yMaxExtent	minTopSideBearing+(yMax-yMin)=2036.
18/2	0	caretSlopeRise	The caret slope rise of 0 and a caret slope run of 1 indicate a horizontal caret for a vertical font.
20/2	1	caretSlopeRun	The caret slope rise of 0 and a caret slope run of 1 indicate a horizontal caret for a vertical font.
22/2	0	caretOffset	Value set to 0 for nonslanted fonts.
24/4	0	reserved	Set to 0.
26/2	0	reserved	Set to 0.
28/2	0	reserved	Set to 0.
30/2	0	reserved	Set to 0.
32/2	0	metricDataFormat	Set to 0.
34/2	258	numOfLongVerMetrics	Number of advance heights in the vertical metrics table is 258.

vmtx - Vertical Metrics Table

The vertical metrics table (tag name: 'vmtx') allows you to specify the vertical spacing for each glyph in a vertical font. This table consists of either one or two arrays that contain metric information (the advance heights and top sidebearings) for the vertical layout of each of the glyphs in the font. The vertical metrics coordinate system is shown below.



TrueType vertical fonts require both a vertical header table (tag name: 'vhea') discussed previously and the vertical metrics table discussed below. The vertical header table contains information that is general to the font as a whole. The vertical metrics table contains information that pertains to specific glyphs. The formats of these tables are similar to those for horizontal metrics (hhea and hmtx).

Vertical Metrics Table Format

The overall structure of the vertical metrics table consists of two arrays shown below: the vMetrics array followed by an array of top side bearings.

This table does not have a header, but does require that the number of glyphs included in the two arrays equals the total number of glyphs in the font.

The number of entries in the vMetrics array is determined by the value of the numOfLongVerMetrics field of the vertical header table.

The vMetrics array contains two values for each entry. These are the advance height and the top sidebearing for each glyph included in the array.

In monospaced fonts, such as Courier or Kanji, all glyphs have the same advance height. If the font is monospaced, only one entry need be in the first array, but that one entry is required.

The format of an entry in the vertical metrics array is given below.

Type	Name	Description
USHORT	advanceHeight	The advance height of the glyph. Unsigned integer in FUnits
SHORT	topSideBearing	The top sidebearing of the glyph. Signed integer in FUnits.

The second array is optional and generally is used for a run of monospaced glyphs in the font. Only one such run is allowed per font, and it must be located at the end of the font. This array contains the top sidebearings of glyphs not represented in the first array, and all the glyphs in this array must have the same advance height as the last entry in the vMetrics array. All entries in this array are therefore monospaced.

The number of entries in this array is calculated by subtracting the value of numOfLongVerMetrics from the number of glyphs in the font. The sum of glyphs represented in the first array plus the glyphs represented in the second array therefore equals the number of glyphs in the font. The format of the top sidebearing array is given below.

Type	Name	Description
SHORT	topSideBearing[]	The top sidebearing of the glyph. Signed integer in FUnits.

Recommendations for Windows Fonts

There are many ways to construct a TrueType font file. This chapter outlines Microsoft recommendations for constructing a font file that will operate on Windows, Macintosh, and OS/2 systems, as well as with applications using Microsoft's TrueType Font Adaptation Kit. It should be noted that all TrueType fonts use Motorola-style byte ordering (Big Endian).

Filenames

The preferred suffix for TrueType font files under Windows is *.TTF.

Table Requirements & Recommendations

Table Alignment and Length

As suggested in the introduction to Chapter 2, all tables should be aligned to begin at offsets which are multiples of four bytes. While this is not required by the TrueType rasterizer, it does prevent ambiguous checksum calculations and greatly speeds table access on some processors.

All tables should be recorded in the table directory with their actual length. To ensure that checksums are calculated correctly, it is suggested that tables begin on LONG word boundries, as mentioned in Chapter 2. Any extra space after a table (and before the next LONG word boundry) should be padded with zeros.

'cmap' Table

When building a Unicode font for Windows, the platform ID should be 3 and the encoding ID should be 1 (this subtable must use cmap format 4). When building a symbol font for Windows, the platform ID should be 3 and the encoding ID should be 0.

Remember that, despite references to "first" and "second" subtables, the subtables must be stored in sorted order by platform and encoding ID.

Macintosh 'cmap' Table

When building a font containing Roman characters that will be used on the Macintosh, an additional subtable is required, specifying platform ID of 1 and encoding ID of 0 (this subtable must use cmap format 0).

In order for the Macintosh 'cmap' table to be useful, the glyphs required for the Macintosh must have glyph indices less than 256 (since the 'cmap' subtable format 0 uses BYTE indices and therefore cannot index any glyph above 255).

The Apple 'cmap' subtable should be constructed according to the guidelines in the "Character Sets" chapter. Note that the "apple logo" and "propeller" (⌘) should be mapped to the nonexistent glyph.

'cvt' Table

Should be defined only if required by font instructions.

'fpgm' Table

Should be defined only if required by font instructions.

'glyf' Table

Must contain all data required to construct the complete UGL character set as specified by the 'cmap' table.

In order for the Macintosh 'cmap' table to be useful, the glyphs required for the Macintosh must have glyph indices less than 256 (since the 'cmap' subtable format 0 uses BYTE indices and therefore cannot index any glyph above 255). This, of course, means that all the glyphs needed to map to the Macintosh character set (as per Chapter 4) must be placed within the first 256 glyph "slots" in this table.

'hdmx' Table

This table is not necessary at all unless instructions are used to control the "phantom points," and should be omitted if bit 2 of the flags field in the 'head' table is zero. (See the 'head' table documentation in Chapter 2.) Microsoft recommends that this table be included for fonts with one or more non-linearly scaled glyphs (i.e., bit 2 or 4 of the flags field is set).

Device records should be defined for all sizes from 8 through 14 point, and even point sizes from 16 through 24 point. However, the table requires pixel-per-em sizes, which depend on the horizontal resolution of the output device. The records in 'hdmx' should cover both 96 dpi devices (CGA, EGA, VGA) and 300 dpi devices (laser and ink jet printers).

Thus, 'hdmx' should contain entries for the following pixel sizes: 11, 12, 13, 15, 16, 17, 19, 21, 24, 27, 29, 32, 33, 37, 42, 46, 50, 54, 58, 67, 75, 83, 92, 100. These values have been rounded to the nearest pixel. For instance, 12 points at 300 dpi would measure 37.5 pixels, but this is rounded down to 37 for this list.

This will add approximately 9,600 bytes to the font file. However, there will be a significant improvement in speed when a client requests advance widths covered by these device records.

If the font includes an 'LTSH' table, the hdmx values are not needed above the linearity threshold.

'head' Table

All data required.

'hhea' Table

All data required. It is suggested that monospaced fonts set numberLongMetrics to three (see hmtx).

'hmtx' Table

All data required. It is suggested that monospaced fonts have three entries in the nMetric field.

'kern' Table

Should contain a single kerning pair subtable (format 0). Windows and OS/2 will not support format 2 (two-dimensional array of kern values by class). Windows and OS/2 will not support multiple tables; only the first format 0 table found will be used. Also, Windows and OS/2 will not support coverage bits 0 through 4 (i.e. assumes horizontal data, kerning values, no cross stream, and override).

‘loca’ Table

All data required. In order for the Macintosh ‘cmap’ table to be useful, the glyphs required for the Macintosh must have glyph indices less than 256 (since the ‘cmap’ subtable format 0 uses BYTE indices and therefore cannot index any glyph with an index greater than 255). Beyond this requirement, the actual ordering of the glyphs in the font can be optimized based on expected utilization, with the most frequently used glyphs appearing at the beginning of the font file. Additionally, glyphs that are often used together should be grouped together in the file. The will help to minimize the amount of swapping required when the font is loaded into memory.

‘LTSH’ Table

Should be defined if bit 2 or 4 of flags in ‘head’ is set.

‘maxp’ Table

All data required.

‘name’ Table

When building a Unicode font for Windows, the platform ID should be 3 and the encoding ID should be 1. When building a symbol font for Windows, the platform ID should be 3 and the encoding ID should be 0.

When building a font containing Roman characters that will be used on the Macintosh, an additional name record is required, specifying platform ID of 1 and encoding ID of 0.

Each set of name records should appear for US English (language ID = 0x0409 for Microsoft records, language ID = 0 for Macintosh records); additional language strings for the Microsoft set of records (platform ID 3) may be added at the discretion of the font vendor.

Remember that, despite references to “first” and “second,” the name record must be stored in sorted order (by platform ID, encoding ID, language ID, name ID). The ‘name’ table platform/encoding IDs must match the ‘cmap’ table platform/encoding IDs, which is how Windows knows which name set to use.

Name strings

The Subfamily string in the 'name' table should be used for variants of weight (ultra light to extra black) and style (oblique/italic or not). So, for example, the full font name of "Helvetica Narrow Italic" should be defined as Family name "Helvetica Narrow" and Subfamily "Italic." This is so that Windows can group the standard four weights of a font in a reasonable fashion for non-typographically aware applications which only support combinations of "bold" and "italic."

'OS/2' Table

All data required.

'post' Table

All information required, although the VM Usage fields may be set to zero. Format 2 is required in order to support the two-byte glyph indices in the UGL character set. Glyph names for the PostScript character set must be defined as per the "PostScript Reference Manual" (Adobe Systems Incorporated, 1988); note that names for all glyphs must be supplied as it cannot be assumed that all Microsoft platforms will support the default names supplied on the Macintosh. Names for the Unicode glyphs outside the PostScript set should be assigned a four character hexidecimal string that corresponds to their Unicode index (e.g. '2302' for the small house glyph). See Chapter 4 for the complete Unicode character set, including PostScript glyph names (where defined).

'prep' Table

Should be defined only if required by the font instructions.

'VDMX' Table

Should be present if hints cause the font to scale non-linearly. If not present, the font is assumed to scale linearly. Clipping may occur if values in this table are absent and font exceeds linear height.

General Recommendations

Non-Standard Fonts

Non-standard fonts such as Symbol or Wingdings™ have special requirements for Microsoft platforms. These requirements affect the ‘cmap’ and ‘name’ tables; the requirements and recommendations for all other tables remain the same.

For the Macintosh, non-standard fonts can continue to use platform ID 1 (Macintosh) and encoding ID 0 (Roman character set). The ‘cmap’ subtable should use format 0 and follow the standard PostScript character encodings.

For non-standard fonts on Microsoft platforms, however, the ‘cmap’ and ‘name’ tables must use platform ID 3 (Microsoft) and encoding ID 0 (Unicode, non-standard character set). Remember that ‘name’ table encodings should agree with the ‘cmap’ table.

The Microsoft ‘cmap’ subtable (platform 3, encoding 0) must use format 4. The character codes should start at 0xF000, which is in the Private Use Area of Unicode. Microsoft suggests deriving the format 4 (Microsoft) encodings by simply adding 0xF000 to the format 0 (Macintosh) encodings.

Under both OS/2 and Windows, only the first 224 characters of non-standard fonts will be accessible: a space and up to 223 printing characters. It does not matter where in user space these start, but 0xF020 is suggested. The usFirstCharIndex and usLastCharIndex values in the ‘OS/2’ table would be set based on the actual minimum and maximum character indices used.

Device Resolutions

OS/2 and Windows make use of a logical device resolution. The physical resolution of a device is also available, but fonts will be rendered based on the logical resolution. The table below lists some important logical resolutions in dots per inch (Horizontal x Vertical). The most important ratios (in order) are 1:1, 1.67:1 and 1.33:1.

Device	Resolution	Aspect Ratio
CGA	96 x 48	2:1
EGA	96 x 72	1.33:1
VGA	96 x 96	1:1
8514	120 x 120	1:1
Dot Matrix	120 x 72	1.67:1
Laser Printer	300 x 300 or 600 x 600	1:1

Baseline to Baseline Distances

The suggested Baseline to Baseline Distance (BTBD) is computed differently for Windows and the Macintosh, and it is based on different TrueType metrics. However, if the recommendations below are followed, the BTBD will be the same for both Windows and the Mac.

Windows

Windows Metric[1]	TrueType Metric
ascent	usWinAscent
descent	usWinDescent
internal leading	usWinAscent + usWinDescent - unitsPerEm
external leading	MAX(0, LineGap - ((usWinAscent + usWinDescent) - (Ascender - Descender)))

Suggested BTBD = *ascent* + *descent* + *external leading*
It should be clear that the "external leading" can never be less than zero. Pixels above the ascent or below the descent will be clipped from the character; this is true for all output devices.

[1] These metrics are returned as part of the logical font data structure by the GDI CreateLogFont() API.

The usWinAscent and usWinDescent are values from the 'OS/2' table. The unitsPerEm value is from the 'head' table. The LineGap, Ascender and Descender values are from the 'hhea' table.

Macintosh

Ascender and Descender are metrics defined by Apple and are *not* to be confused with the Windows ascent or descent, nor should they be confused with the true typographic ascender and descender that are found in AFM files.

Macintosh Metric[2]	TrueType Metric
ascender	Ascender
descender	Descender
leading	LineGap

Suggested BTBD = *ascender + descender + leading*
If pixels extend above the ascent or below the descent, the character will be squashed in the vertical direction so that all pixels fit within these limitations; this is true for screen display only.

Making Them Match

If you perform some simple algebra, you will see that the suggested BTBD across both Macintosh and Windows will be identical if and only if:

LineGap >= (yMax - yMin) - (Ascender - Descender)

Style Bits

For backwards compatibility with previous versions of Windows, the macStyle bits in the 'head' table will be used to determine whether or not a font is regular, bold or italic (in the absence of an 'OS/2' table). This is completely independent of the usWeightClass and PANOSE information in the 'OS/2' table, the ItalicAngle in the 'post' table, and all other related metrics. If the 'OS/2' table is present, then the fsSelection bits are used to determine this information.

[2] These metrics are returned by the Mac QuickDraw GetFontInfo() API.

Drop-out Control

Drop-out control is needed if there is a difference in bitmaps with dropout control on and off. Two cases where drop-out control is needed are when the font is rotated or when the size of the font is at or below 8 ppem. Do not use SCANCTRL unless needed. SCANCTRL or the drop-out control rasterizer should be avoided for Roman fonts above 8 points per em (ppem) when the font is not under rotation. SCANCTRL should not be used for "stretched" fonts (e.g. fonts displayed at non-square aspect ratios, like that found on an EGA).

Embedded bitmaps

Three new tables are used to embed bitmaps in TrueType fonts. They are the 'EBLC' table for embedded bitmap locators, the 'EBDT' table for embedded bitmap data, and the 'EBSC' table for embedded bitmap scaling information. TrueType embedded bitmaps are also called 'sbits'.

The behavior of sbits within a TrueType font is essentially transparent to the client. A client need not be aware whether the bitmap returned by the rasterizer comes from an sbit or from a scan-converted outline.

The metrics in 'sbit' tables overrule the outline metrics at all sizes where sbits are defined. Fonts with 'hdmx' tables should correct those tables with 'sbit' values.

'Sbit only' fonts, that is fonts with embedded bitmaps but without outline data, are permitted. Care must be taken to ensure that all required TrueType tables except 'glyf' and 'loca' are present in such a font. Obviously, such fonts will only be able to return glyphs and sizes for which sbits are defined.

TrueType Collection (TTC) Files

A TrueType Collection (TTC) is a means of delivering multiple TrueType fonts in a single file structure. TrueType Collections are most useful when the fonts to be delivered together share many glyphs in common. By allowing multiple fonts to share glyph sets, TTCs can result in a significant saving of file space.

For example, a group of Japanese fonts may each have their own designs for the kana glyphs, but share identical designs for the kanji. With ordinary TrueType font files, the only way to include the common kanji glyphs is to copy their glyph data into each font. Since the kanji represent much more data than the kana, this results in a great deal of wasteful duplication of glyph data. TTCs were defined to solve this problem.

TTC File Structure

A TrueType Collection file consists of a single TTC Header table, two or more Table Directories, and a number of TrueType tables.

The TTC Header must be located at the beginning of the TTC file.

The TTC file must contain a complete Table Directory for each different font design. A TTC file Table Directory has exactly the same format as a TTF file Table Directory. The table offsets in all Table Directories within a TTC file are measured from the beginning of the TTC file.

Each TrueType table in a TTC file is referenced through the Table Directories of all fonts which use that table. Some of the TrueType tables must appear multiple times, once for each font included in the TTC; while other tables should be shared by all fonts in the TTC.

As an example, consider a TTC file which combines two Japanese fonts (Font1 and Font2). The fonts have different kana designs (Kana1 and Kana2) but use the same design for kanji. The TTC file contains a single ‘glyf’ table which includes both designs of kana together with the kanji; both fonts’ Table Directories point to this ‘glyf’ table. But each font’s Table Directory points to a different ‘cmap’ table, which identifies the glyph set to use. Font1’s ‘cmap’ table points to the Kana1 region of the ‘loca’ and ‘glyf’ tables for kana glyphs, and to the kanji region for the kanji. Font2’s ‘cmap’ table points to the Kana2 region of the ‘loca’ and ‘glyf’ tables for kana glyphs, and to the same kanji region for the kanji.

The tables that should have a unique copy per font are those that are used by the system in identifying the font and its character mapping, including ‘cmap’, ‘name’, and ‘OS/2’. The tables that should be shared by all fonts in the TTC are those that define glyph and instruction data or use glyph indices to access data: ‘glyf’, ‘loca’, ‘hmtx’, ‘hdmx’, ‘LTSH’, ‘cvt ’, ‘fpgm’, ‘prep’, ‘EBLC’, ‘EBDT’, ‘EBSC’, ‘maxp’, and so on. In practice, any tables which have identical data for two or more fonts may be shared.

Creating a TrueType Collection by combining existing TrueType font files is a non-trivial process. It involves paying close attention the issue of glyph renumbering in a font and the side effects that can result, in the ‘cmap’ table and elsewhere. The fonts to be merged must also have compatible TrueType instructions—that is, their preprograms, function definitions, and control values must not conflict.

TrueType Collection files use the filename suffix .TTC.

TTC Header Table

The purpose of the TTC Header table is to locate the different Table Directories within a TTC file.

The TTC Header is located at the beginning of the TTC file (offset = 0). It consists of an identification tag, a version number, a count of the number of TrueType fonts (Table Directories) in the file, and an array of offsets to each Table Directory.

TTC Header Table

Type	Name	Description
TAG	TTCTag	TrueType Collection ID string: ‘ttcf’
FIXED32	Version	Version of the TTC Header table (initially 0x00010000)
ULONG	DirectoryCount	Number of Table Directories in TTC
ULONG	⇨ TableDirectory [DirectoryCount]	Array of offsets to Table Directories from file begin

Note that the TTC Header is *not* a table within a TrueType font file.

Character Sets

This chapter contains a table comparing the character sets of WGL4 (the Windows Glyph List defined for Windows 95), UGL, Window 3.1x (for the U.S.), and the Macintosh (for the U.S.). The characters checked are required in a font for compatibility with the respective platforms. Microsoft suggests that at a minimum, the U.S. Windows 3.1x (ANSI) set and the Macintosh/PostScript set be supported in all your fonts. For the character set to properly support codepages under OS/2, the full UGL character set must also be included.

The character set table in this chapter contains the following information:

Field Name	Description
Min	Minimum character set recommended
Unicode	Unicode value of glyph
PostScript Name	PostScript name of glyph
Descriptive Name	Descriptive name of glyph
WGL4	Glyph in WGL4 character set
UGL	Glyph in UGL character set
Win31	Glyph in US Win31 character set
MacChar	US Macintosh character code for glyph
MacGlyph	US Macintosh glyph index for glyph

Font files for Microsoft platforms *must* use Unicode indices (given here as hexadecimal values).

Microsoft Platform Requirements

TrueType font files use the ‘cmap’ table to access glyphs. Thus, it is feasible to map similar looking characters to single internal glyph (e.g. latin capital letter eth and latin capital letter d with stroke). However, future international extensions to TrueType may require a unique glyph for each character in the font, so this practice is not recommended.

The ‘cmap’ table used for this list of characters will be implemented with the format 4 as described in Chapter 2.

Note that space (U+0020) and no-break space (U+00a0) should be mapped to a glyph with no contours and a positive advance width; this advance width should be the same for the two glyphs.

Lining numbers (U+0030 through U+0039; i.e. the digits 0 - 9) should be monospaced. Old style figures need not be monospaced.

White space should be evenly distributed between the left and right side bearings of glyphs. Extra space should be placed on the right if grid-fitting results in an odd number of pixels.

Macintosh Platform Requirements

Since the Macintosh requires the use of 'cmap' subtable format 0 (which only allows for BYTE glyph indices), the glyphs required by the Macintosh must appear within the first 256 positions in the 'glyf' table. Apple recommends a particular order for the glyphs; this glyph order is indicated in the final column of the following table.

A list of Macintosh mapping requirements follows:

- Glyph 0 is the missing character glyph.
- Glyph 1 is the null glyph; it has no contours and zero advance width.
- All characters in the character set defined in the table must be present. Certain specified characters, however, are mapped to glyph 0 (the missing character glyph) as stated below.
- The following character codes must be mapped to glyph 0 (the missing character glyph). Note that all character codes are given as decimal values.

  001-031 Misc. ASCII control codes (*note exceptions below*)
  127 DEL Delete
- The following characters must map to glyph 1 (the null glyph).

  000 NUL Null
  008 BS Backspace
  029 GSGroup Seperator
- The following characters must map to a glyph with no contours and positive advance width:

  009 HT Horiztontal Tabulation
  032 space
  202 figureSpace (No-Break Space)
- The following groups of characters must have the same width

  009 (HT) and 032 (space)
- The mapping of the Carriage Return (CR, 13) depends on whether the font is designed to be used left-to-right or right-to-left. For left-to-right (e.g., Roman), it must be mapped to a glyph with no contours and positive advance. For right-to-left (e.g., Hebrew), it must be mapped to glyph 1 (null glyph).

Recommendations

In cursive fonts, glyphs should overlap to allow glyphs to join together in device-independent text. A five per cent (5%) overlap is recommended.

Follow all recommendations for Microsoft platforms (listed above).

Min	Unicode	PostScript Name	Descriptive Name	WGL4	UGL	Win31	MacChar	MacIndex
♦		notdef		♦				0
♦		.null		♦			0x0	1
♦		CR		♦			0xd	2
♦	U+0020	space	space	♦	♦	♦	0x20	3
♦	U+0021	exclam	exclamation mark	♦	♦	♦	0x21	4
♦	U+0022	quotedbl	quotation mark	♦	♦	♦	0x22	5
♦	U+0023	numbersign	number sign	♦	♦	♦	0x23	6
♦	U+0024	dollar	dollar sign	♦	♦	♦	0x24	7
♦	U+0025	percent	percent sign	♦	♦	♦	0x25	8
♦	U+0026	ampersand	ampersand	♦	♦	♦	0x26	9
♦	U+0027	quotesingle	apostrophe	♦	♦	♦	0x27	10
♦	U+0028	parenleft	left parenthesis	♦	♦	♦	0x28	11
♦	U+0029	parenright	right parenthesis	♦	♦	♦	0x29	12
♦	U+002a	asterisk	asterisk	♦	♦	♦	0x2a	13
♦	U+002b	plus	plus sign	♦	♦	♦	0x2b	14
♦	U+002c	comma	comma	♦	♦	♦	0x2c	15
♦	U+002d	hyphen	hyphen-minus	♦	♦	♦	0x2d	16
♦	U+002e	period	period	♦	♦	♦	0x2e	17
♦	U+002f	slash	slash	♦	♦	♦	0x2f	18
♦	U+0030	zero	digit zero	♦	♦	♦	0x30	19
♦	U+0031	one	digit one	♦	♦	♦	0x31	20
♦	U+0032	two	digit two	♦	♦	♦	0x32	21
♦	U+0033	three	digit three	♦	♦	♦	0x33	22
♦	U+0034	four	digit four	♦	♦	♦	0x34	23
♦	U+0035	five	digit five	♦	♦	♦	0x35	24
♦	U+0036	six	digit six	♦	♦	♦	0x36	25
♦	U+0037	seven	digit seven	♦	♦	♦	0x37	26
♦	U+0038	eight	digit eight	♦	♦	♦	0x38	27
♦	U+0039	nine	digit nine	♦	♦	♦	0x39	28
♦	U+003a	colon	colon	♦	♦	♦	0x3a	29
♦	U+003b	semicolon	semicolon	♦	♦	♦	0x3b	30
♦	U+003c	less	less-than sign	♦	♦	♦	0x3c	31
♦	U+003d	equal	equals sign	♦	♦	♦	0x3d	32
♦	U+003e	greater	greater-than sign	♦	♦	♦	0x3e	33
♦	U+003f	question	question mark	♦	♦	♦	0x3f	34
♦	U+0040	at	commercial at	♦	♦	♦	0x40	35
♦	U+0041	A	latin capital letter a	♦	♦	♦	0x41	36
♦	U+0042	B	latin capital letter b	♦	♦	♦	0x42	37
♦	U+0043	C	latin capital letter c	♦	♦	♦	0x43	38
♦	U+0044	D	latin capital letter d	♦	♦	♦	0x44	39
♦	U+0045	E	latin capital letter e	♦	♦	♦	0x45	40
♦	U+0046	F	latin capital letter f	♦	♦	♦	0x46	41
♦	U+0047	G	latin capital letter g	♦	♦	♦	0x47	42
♦	U+0048	H	latin capital letter h	♦	♦	♦	0x48	43
♦	U+0049	I	latin capital letter i	♦	♦	♦	0x49	44
♦	U+004a	J	latin capital letter j	♦	♦	♦	0x4a	45
♦	U+004b	K	latin capital letter k	♦	♦	♦	0x4b	46
♦	U+004c	L	latin capital letter l	♦	♦	♦	0x4c	47
♦	U+004d	M	latin capital letter m	♦	♦	♦	0x4d	48
♦	U+004e	N	latin capital letter n	♦	♦	♦	0x4e	49
♦	U+004f	O	latin capital letter o	♦	♦	♦	0x4f	50

♦ - Glyph in character set
❁ - Glyph not accessible from keyboard, only by PostScript name

Min	Unicode	PostScript Name	Descriptive Name	WGL4	UGL	Win31	MacChar	MacIndex
♦	U+0050	P	latin capital letter p	♦	♦	♦	0x50	51
♦	U+0051	Q	latin capital letter q	♦	♦	♦	0x51	52
♦	U+0052	R	latin capital letter r	♦	♦	♦	0x52	53
♦	U+0053	S	latin capital letter s	♦	♦	♦	0x53	54
♦	U+0054	T	latin capital letter t	♦	♦	♦	0x54	55
♦	U+0055	U	latin capital letter u	♦	♦	♦	0x55	56
♦	U+0056	V	latin capital letter v	♦	♦	♦	0x56	57
♦	U+0057	W	latin capital letter w	♦	♦	♦	0x57	58
♦	U+0058	X	latin capital letter x	♦	♦	♦	0x58	59
♦	U+0059	Y	latin capital letter y	♦	♦	♦	0x59	60
♦	U+005a	Z	latin capital letter z	♦	♦	♦	0x5a	61
♦	U+005b	bracketleft	left square bracket	♦	♦	♦	0x5b	62
♦	U+005c	backslash	backslash	♦	♦	♦	0x5c	63
♦	U+005d	bracketright	right square bracket	♦	♦	♦	0x5d	64
♦	U+005e	asciicircum	circumflex accent	♦	♦	♦	0x5e	65
♦	U+005f	underscore	underline	♦	♦	♦	0x5f	66
♦	U+0060	grave	grave accent	♦	♦	♦	0x60	67
♦	U+0061	a	latin small letter a	♦	♦	♦	0x61	68
♦	U+0062	b	latin small letter b	♦	♦	♦	0x62	69
♦	U+0063	c	latin small letter c	♦	♦	♦	0x63	70
♦	U+0064	d	latin small letter d	♦	♦	♦	0x64	71
♦	U+0065	e	latin small letter e	♦	♦	♦	0x65	72
♦	U+0066	f	latin small letter f	♦	♦	♦	0x66	73
♦	U+0067	g	latin small letter g	♦	♦	♦	0x67	74
♦	U+0068	h	latin small letter h	♦	♦	♦	0x68	75
♦	U+0069	i	latin small letter i	♦	♦	♦	0x69	76
♦	U+006a	j	latin small letter j	♦	♦	♦	0x6a	77
♦	U+006b	k	latin small letter k	♦	♦	♦	0x6b	78
♦	U+006c	l	latin small letter l	♦	♦	♦	0x6c	79
♦	U+006d	m	latin small letter m	♦	♦	♦	0x6d	80
♦	U+006e	n	latin small letter n	♦	♦	♦	0x6e	81
♦	U+006f	o	latin small letter o	♦	♦	♦	0x6f	82
♦	U+0070	p	latin small letter p	♦	♦	♦	0x70	83
♦	U+0071	q	latin small letter q	♦	♦	♦	0x71	84
♦	U+0072	r	latin small letter r	♦	♦	♦	0x72	85
♦	U+0073	s	latin small letter s	♦	♦	♦	0x73	86
♦	U+0074	t	latin small letter t	♦	♦	♦	0x74	87
♦	U+0075	u	latin small letter u	♦	♦	♦	0x75	88
♦	U+0076	v	latin small letter v	♦	♦	♦	0x76	89
♦	U+0077	w	latin small letter w	♦	♦	♦	0x77	90
♦	U+0078	x	latin small letter x	♦	♦	♦	0x78	91
♦	U+0079	y	latin small letter y	♦	♦	♦	0x79	92
♦	U+007a	z	latin small letter z	♦	♦	♦	0x7a	93
♦	U+007b	braceleft	left curly bracket	♦	♦	♦	0x7b	94
♦	U+007c	bar	vertical line	♦	♦	♦	0x7c	95
♦	U+007d	braceright	right curly bracket	♦	♦	♦	0x7d	96
♦	U+007e	asciitilde	tilde	♦	♦	♦	0x7e	97
♦	U+00a0	nbspace	no-break space		♦	♦	0xca	172
♦	U+00a1	exclamdown	inverted exclamation mark	♦	♦	♦	0xc1	163
♦	U+00a2	cent	cent sign	♦	♦	♦	0xa2	132
♦	U+00a3	sterling	pound sign	♦	♦	♦	0xa3	133

Min	Unicode	PostScript Name	Descriptive Name	WGL4	UGL	Win31	MacChar	MacIndex
♦	U+00a4	currency	currency sign	♦	♦	♦	0xdb	189
♦	U+00a5	yen	yen sign	♦	♦	♦	0xb4	150
♦	U+00a6	brokenbar	broken bar	♦	♦	♦	❁	232
♦	U+00a7	section	section sign	♦	♦	♦	0xa4	134
♦	U+00a8	dieresis	diaeresis	♦	♦	♦	0xac	142
♦	U+00a9	copyright	copyright sign	♦	♦	♦	0xa9	139
♦	U+00aa	ordfeminine	feminine ordinal indicator	♦	♦	♦	0xbb	157
♦	U+00ab	guillemotleft	left guillemet	♦	♦	♦	0xc7	169
♦	U+00ac	logicalnot	not sign	♦	♦	♦	0xc2	164
♦	U+00ad	sfthyphen	soft hyphen		♦	♦		
♦	U+00ae	registered	registered trade mark sign	♦	♦	♦	0xa8	138
♦	U+00af	overscore	macron, overline	♦	♦	♦	0xf8	218
♦	U+00b0	degree	degree sign	♦	♦	♦	0xa1	131
♦	U+00b1	plusminus	plus-minus sign	♦	♦	♦	0xb1	147
♦	U+00b2	twosuperior	superscript two	♦	♦	♦	❁	242
♦	U+00b3	threesuperior	superscript three	♦	♦	♦	❁	243
♦	U+00b4	acute	acute accent	♦	♦	♦	0xab	141
♦	U+00b5	mu1	micro sign	♦	♦	♦	0xb5	151
♦	U+00b6	paragraph	paragraph sign	♦	♦	♦	0xa6	136
♦	U+00b7	middot	middle dot, kana conjoctive	♦	♦	♦		
♦	U+00b8	cedilla	cedilla	♦	♦	♦	0xfc	222
♦	U+00b9	onesuperior	superscript one	♦	♦	♦	❁	241
♦	U+00ba	ordmasculine	masculine ordinal indicator	♦	♦	♦	0xbc	158
♦	U+00bb	guillemotright	right guillemet	♦	♦	♦	0xc8	170
♦	U+00bc	onequarter	vulgar fraction one quarter	♦	♦	♦	❁	245
♦	U+00bd	onehalf	vulgar fraction one half	♦	♦	♦	❁	244
♦	U+00be	threequarters	vulgar fraction three quarters	♦	♦	♦	❁	246
♦	U+00bf	questiondown	inverted question mark	♦	♦	♦	0xc0	162
♦	U+00c0	Agrave	latin capital letter a with grave accent	♦	♦	♦	0xcb	173
♦	U+00c1	Aacute	latin capital letter a with acute accent	♦	♦	♦	0xe7	201
♦	U+00c2	Acircumflex	latin capital letter a with circumflex accent	♦	♦	♦	0xe5	199
♦	U+00c3	Atilde	latin capital letter a with tilde	♦	♦	♦	0xcc	174
♦	U+00c4	Adieresis	latin capital letter a with diaeresis	♦	♦	♦	0x80	98
♦	U+00c5	Aring	latin capital letter a with ring above	♦	♦	♦	0x81	99
♦	U+00c6	AE	latin capital letter a with e	♦	♦	♦	0xae	144
♦	U+00c7	Ccedilla	latin capital letter c with cedilla	♦	♦	♦	0x82	100
♦	U+00c8	Egrave	latin capital letter e with grave accent	♦	♦	♦	0xe9	203
♦	U+00c9	Eacute	latin capital letter e with acute accent	♦	♦	♦	0x83	101
♦	U+00ca	Ecircumflex	latin capital letter e with circumflex accent	♦	♦	♦	0xe6	200
♦	U+00cb	Edieresis	latin capital letter e with diaeresis	♦	♦	♦	0xe8	202
♦	U+00cc	Igrave	latin capital letter i with grave accent	♦	♦	♦	0xed	207

♦ - Glyph in character set
❁ - Glyph not accessible from keyboard, only by PostScript name

Min	Unicode	PostScript Name	Descriptive Name	WGL4	UGL	Win31	MacChar	MacIndex
♦	U+00cd	Iacute	latin capital letter i with acute accent	♦	♦	♦	0xea	204
♦	U+00ce	Icircumflex	latin capital letter i with circumflex accent	♦	♦	♦	0xeb	205
♦	U+00cf	Idieresis	latin capital letter i with diaeresis	♦	♦	♦	0xec	206
♦	U+00d0	Eth	latin capital letter eth	♦	♦	♦	☸	233
♦	U+00d1	Ntilde	latin capital letter n with tilde	♦	♦	♦	0x84	102
♦	U+00d2	Ograve	latin capital letter o with grave accent	♦	♦	♦	0xf1	211
♦	U+00d3	Oacute	latin capital letter o with acute accent	♦	♦	♦	0xee	208
♦	U+00d4	Ocircumflex	latin capital letter o with circumflex accent	♦	♦	♦	0xef	209
♦	U+00d5	Otilde	latin capital letter o with tilde	♦	♦	♦	0xcd	175
♦	U+00d6	Odieresis	latin capital letter o with diaeresis	♦	♦	♦	0x85	103
♦	U+00d7	multiply	multiplication sign	♦	♦	♦	☸	240
♦	U+00d8	Oslash	latin capital letter o with oblique stroke	♦	♦	♦	0xaf	145
♦	U+00d9	Ugrave	latin capital letter u with grave accent	♦	♦	♦	0xf4	214
♦	U+00da	Uacute	latin capital letter u with acute accent	♦	♦	♦	0xf2	212
♦	U+00db	Ucircumflex	latin capital letter u with circumflex accent	♦	♦	♦	0xf3	213
♦	U+00dc	Udieresis	latin capital letter u with diaeresis	♦	♦	♦	0x86	104
♦	U+00dd	Yacute	latin capital letter y with acute accent	♦	♦	♦	☸	235
♦	U+00de	Thorn	latin capital letter thorn	♦	♦	♦	☸	237
♦	U+00df	germandbls	latin small letter sharp s	♦	♦	♦	0xa7	137
♦	U+00e0	agrave	latin small letter a with grave accent	♦	♦	♦	0x88	106
♦	U+00e1	aacute	latin small letter a with acute accent	♦	♦	♦	0x87	105
♦	U+00e2	acircumflex	latin small letter a with circumflex accent	♦	♦	♦	0x89	107
♦	U+00e3	atilde	latin small letter a with tilde	♦	♦	♦	0x8b	109
♦	U+00e4	adieresis	latin small letter a with diaeresis	♦	♦	♦	0x8a	108
♦	U+00e5	aring	latin small letter a with ring above	♦	♦	♦	0x8c	110
♦	U+00e6	ae	latin small letter a with e	♦	♦	♦	0xbe	160
♦	U+00e7	ccedilla	latin small letter c with cedilla	♦	♦	♦	0x8d	111
♦	U+00e8	egrave	latin small letter e with grave accent	♦	♦	♦	0x8f	113
♦	U+00e9	eacute	latin small letter e with acute accent	♦	♦	♦	0x8e	112
♦	U+00ea	ecircumflex	latin small letter e with circumflex accent	♦	♦	♦	0x90	114
♦	U+00eb	edieresis	latin small letter e with diaeresis	♦	♦	♦	0x91	115
♦	U+00ec	igrave	latin small letter i with grave	♦	♦	♦	0x93	117

Min	Unicode	PostScript Name	Descriptive Name	WGL4	UGL	Win31	MacChar	MacIndex
			accent					
♦	U+00ed	iacute	latin small letter i with acute accent	♦	♦	♦	0x92	116
♦	U+00ee	icircumflex	latin small letter i with circumflex accent	♦	♦	♦	0x94	118
♦	U+00ef	idieresis	latin small letter i with diaeresis	♦	♦	♦	0x95	119
♦	U+00f0	eth	latin small letter eth	♦	♦	♦	☸	234
♦	U+00f1	ntilde	latin small letter n with tilde	♦	♦	♦	0x96	120
♦	U+00f2	ograve	latin small letter o with grave accent	♦	♦	♦	0x98	122
♦	U+00f3	oacute	latin small letter o with acute accent	♦	♦	♦	0x97	121
♦	U+00f4	ocircumflex	latin small letter o with circumflex accent	♦	♦	♦	0x99	123
♦	U+00f5	otilde	latin small letter o with tilde	♦	♦	♦	0x9b	125
♦	U+00f6	odieresis	latin small letter o with diaeresis	♦	♦	♦	0x9a	124
♦	U+00f7	divide	division sign	♦	♦	♦	0xd6	184
♦	U+00f8	oslash	latin small letter o with oblique stroke	♦	♦	♦	0xbf	161
♦	U+00f9	ugrave	latin small letter u with grave accent	♦	♦	♦	0x9d	127
♦	U+00fa	uacute	latin small letter u with acute accent	♦	♦	♦	0x9c	126
♦	U+00fb	ucircumflex	latin small letter u with circumflex accent	♦	♦	♦	0x9e	128
♦	U+00fc	udieresis	latin small letter u with diaeresis	♦	♦	♦	0x9f	129
♦	U+00fd	yacute	latin small letter y with acute accent	♦	♦	♦	☸	236
♦	U+00fe	thorn	latin small letter thorn	♦	♦	♦	☸	238
♦	U+00ff	ydieresis	latin small letter y with diaeresis	♦	♦	♦	0xd8	186
	U+0100	Amacron	latin capital letter a with macron	♦				
	U+0101	amacron	latin small letter a with macron	♦				
	U+0102	Abreve	latin capital letter a with breve	♦	♦			
	U+0103	abreve	latin small letter a with breve	♦	♦			
	U+0104	Aogonek	latin capital letter a with ogonek	♦	♦			
	U+0105	aogonek	latin small letter a with ogonek	♦	♦			
♦	U+0106	Cacute	latin capital letter c with acute accent	♦	♦		☸	253
♦	U+0107	cacute	latin small letter c with acute accent	♦	♦		☸	254
	U+0108	Ccircumflex	latin capital letter c with hacek	♦				
	U+0109	ccircumflex	latin small letter c with hacek	♦				
	U+010a	Cdot	latin capital letter c with dot above	♦				
	U+010b	cdot	latin small letter c with dot	♦				

♦ - Glyph in character set
☸ - Glyph not accessible from keyboard, only by PostScript name

Min	Unicode	PostScript Name	Descriptive Name	WGL4	UGL	Win31	MacChar	MacIndex
			above					
♦	U+010c	Ccaron	latin capital letter c with caron	♦	♦		☸	255
♦	U+010d	ccaron	latin small letter c with caron	♦	♦		☸	256
	U+010e	Dcaron	latin capital letter d with hacek	♦	♦			
	U+010f	dcaron	latin small letter d with hacek	♦	♦			
	U+0110	Dslash	latin capital letter d with stroke	♦	♦			
♦	U+0111	dmacron	latin small letter d with stroke	♦	♦		☸	257
	U+0112	Emacron	latin capital letter e with macron	♦				
	U+0113	emacron	latin small letter e with macron	♦				
	U+0114	Ebreve	latin capital letter e with breve	♦				
	U+0115	ebreve	latin small letter e with breve	♦				
	U+0116	Edot	latin capital letter e with dot above	♦				
	U+0117	edot	latin small letter e with dot above	♦				
	U+0118	Eogonek	latin capital letter e with ogenek	♦	♦			
	U+0119	eogonek	latin small letter e with ogenek	♦	♦			
	U+011a	Ecaron	latin capital letter e with hacek	♦	♦			
	U+011b	ecaron	latin small letter e with hacek	♦	♦			
	U+011c	Gcircumflex	latin capital letter g with circumflex	♦				
	U+011d	gcircumflex	latin small letter g with circumflex	♦				
♦	U+011e	Gbreve	latin capital letter g with breve	♦	♦		☸	248
♦	U+011f	gbreve	latin small letter g with breve	♦	♦		☸	249
	U+0120	Gdot	latin capital letter g with dot above	♦				
	U+0121	gdot	latin small letter g with dot above	♦				
	U+0122	Gcedilla	latin capital letter g with cedilla	♦				
	U+0123	gcedilla	latin small letter g with cedilla	♦				
	U+0124	Hcircumflex	latin capital letter h with circumflex	♦				
	U+0125	hcircumflex	latin small letter h with circumflex	♦				
	U+0126	Hbar	latin capital letter h with stroke	♦				
	U+0127	hbar	latin small letter h with stroke	♦				
	U+0128	Itilde	latin capital letter i with tilde	♦				

Min	Unicode	PostScript Name	Descriptive Name	WGL4	UGL	Win31	MacChar	MacIndex
	U+0129	itilde	latin small letter i with tilde	♦				
	U+012a	Imacron	latin capital letter i with macron	♦				
	U+012b	imacron	latin small letter i with macron	♦				
	U+012c	Ibreve	latin capital letter i with breve	♦				
	U+012d	ibreve	latin small letter i with breve	♦				
	U+012e	Iogonek	latin capital letter i with ogonek	♦				
	U+012f	iogonek	latin small letter i with ogonek	♦				
♦	U+0130	Idot	latin capital letter i with dot above	♦	♦		☸	250
♦	U+0131	dotlessi	latin small letter i without dot above	♦	♦		0xf5	215
	U+0132	IJ	latin capital ligature ij	♦				
	U+0133	ij	latin small ligature ij	♦				
	U+0134	Jcircumflex	latin capital letter j with circumflex	♦				
	U+0135	jcircumflex	latin small letter j with circumflex	♦				
	U+0136	Kcedilla	latin capital letter k with cedilla	♦				
	U+0137	kcedilla	latin small letter k with cedilla	♦				
	U+0138	kgreenlandic	latin small letter kra	♦				
	U+0139	Lacute	latin capital letter l with acute accent	♦	♦			
	U+013a	lacute	latin small letter l with acute accent	♦	♦			
	U+013b	Lcedilla	latin capital letter l with cedilla	♦				
	U+013c	lcedilla	latin small letter l with cedilla	♦				
	U+013d	Lcaron	latin capital letter l with hacek	♦	♦			
	U+013e	lcaron	latin small letter l with hacek	♦	♦			
	U+013f	Ldot	latin capital letter l with middle dot	♦	♦			
	U+0140	ldot	latin small letter l with middle dot	♦	♦			
♦	U+0141	Lslash	latin capital letter l with stroke	♦	♦		☸	226
♦	U+0142	lslash	latin small letter l with stroke	♦	♦		☸	227
	U+0143	Nacute	latin capital letter n with acute accent	♦	♦			
	U+0144	nacute	latin small letter n with acute accent	♦	♦			
	U+0145	Ncedilla	latin capital letter n with cedilla	♦				
	U+0146	ncedilla	latin small letter n with cedilla	♦				
	U+0147	Ncaron	latin capital letter n with hacek	♦	♦			
	U+0148	ncaron	latin small letter n with	♦	♦			

♦ - Glyph in character set
☸ - Glyph not accessible from keyboard, only by PostScript name

Min	Unicode	PostScript Name	Descriptive Name	WGL4	UGL	Win31	MacChar	MacIndex
			hacek					
	U+0149	napostrophe	latin small letter n preceded by apostrophe	♦				
	U+014a	Eng	latin capital letter eng	♦				
	U+014b	eng	latin small letter eng	♦				
	U+014c	Omacron	latin capital letter o with macron	♦				
	U+014d	omacron	latin small letter o with macron	♦				
	U+014e	Obreve	latin capital letter o with breve	♦				
	U+014f	obreve	latin small letter o with breve	♦				
	U+0150	Odblacute	latin capital letter o with double acute accent	♦	♦			
	U+0151	odblacute	latin small letter o with double acute accent	♦	♦			
♦	U+0152	OE	latin capital ligature o with e	♦	♦	♦	0xce	176
♦	U+0153	oe	latin small ligature o with e	♦	♦	♦	0xcf	177
	U+0154	Racute	latin capital letter r with acute accent	♦	♦			
	U+0155	racute	latin small letter r with acute accent	♦	♦			
	U+0156	Rcedilla	latin capital letter r with cedilla	♦				
	U+0157	rcedilla	latin small letter r with cedilla	♦				
	U+0158	Rcaron	latin capital letter r with hacek	♦	♦			
	U+0159	rcaron	latin small letter r with hacek	♦	♦			
	U+015a	Sacute	latin capital letter s with acute accent	♦	♦			
	U+015b	sacute	latin small letter s with acute accent	♦	♦			
	U+015c	Scircumflex	latin capital letter s with circumflex	♦				
	U+015d	scircumflex	latin small letter s with circumflex	♦				
♦	U+015e	Scedilla	latin capital letter s with cedilla	♦	♦		☸	251
♦	U+015f	scedilla	latin small letter s with cedilla	♦	♦		☸	252
♦	U+0160	Scaron	latin capital letter s with hacek	♦	♦	♦	☸	228
♦	U+0161	scaron	latin small letter s with hacek	♦	♦	♦	☸	229
	U+0162	Tcedilla	latin capital letter t with cedilla	♦	♦			
	U+0163	tcedilla	latin small letter t with cedilla	♦	♦			
	U+0164	Tcaron	latin capital letter t with hacek	♦	♦			
	U+0165	tcaron	latin small letter t with hacek	♦	♦			
	U+0166	Tbar	latin capital letter t with stroke	♦				
	U+0167	tbar	latin small letter t with stroke	♦				
	U+0168	Utilde	latin capital letter u with tilde	♦				
	U+0169	utilde	latin small letter u with tilde	♦				
	U+016a	Umacron	latin capital letter u with	♦				

Min	Unicode	PostScript Name	Descriptive Name	WGL4	UGL	Win31	MacChar	MacIndex
			macron					
	U+016b	umacron	latin small letter u with macron	♦				
	U+016c	Ubreve	latin capital letter u with breve	♦				
	U+016d	ubreve	latin small letter u with breve	♦				
	U+016e	Uring	latin capital letter u with ring above	♦	♦			
	U+016f	uring	latin small letter u with ring above	♦	♦			
	U+0170	Udblacute	latin capital letter u with double acute accent	♦	♦			
	U+0171	udblacute	latin small letter u with double acute accent	♦	♦			
	U+0172	Uogonek	latin capital letter u with ogonek	♦				
	U+0173	uogonek	latin small letter u with ogonek	♦				
	U+0174	Wcircumflex	latin capital letter w with circumflex	♦				
	U+0175	wcircumflex	latin cmall letter w with circumflex	♦				
	U+0176	Ycircumflex	latin capital letter y with circumflex	♦				
	U+0177	ycircumflex	latin small letter y with circumflex	♦				
♦	U+0178	Ydieresis	latin capital letter y with diaeresis	♦	♦	♦	0xd9	187
	U+0179	Zacute	latin capital letter z with acute accent	♦	♦			
	U+017a	zacute	latin small letter z with acute accent	♦	♦			
	U+017b	Zdot	latin capital letter z with dot above	♦	♦			
	U+017c	zdot	latin small letter z with dot above	♦	♦			
♦	U+017d	Zcaron	latin capital letter z with hacek	♦	♦		☸	230
♦	U+017e	zcaron	latin small letter z with hacek	♦	♦		☸	231
	U+017f	longs	latin small letter long s	♦				
♦	U+0192	florin	latin small letter script f,florin sign	♦	♦		0xc4	166
	U+01fa	Aringacute	latin capital letter a with ring above and acute	♦				
	U+01fb	aringacute	latin small letter a with ring above and acute	♦				
	U+01fc	AEacute	latin capital ligature ae with acute	♦				
	U+01fd	aeacute	latin small ligature ae with acute	♦				
	U+01fe	Oslashacute	latin capital letter o with stroke and acute	♦				
	U+01ff	oslashacute	latin small letter o with stroke and acute	♦				
♦	U+02c6	circumflex	nonspacing circumflex accent	♦		♦	0xf6	216

♦ - Glyph in character set
☸ - Glyph not accessible from keyboard, only by PostScript name

Min	Unicode	PostScript Name	Descriptive Name	WGL4	UGL	Win31	MacChar	MacIndex
♦	U+02c7	caron	modifier letter hacek	♦	♦		0xff	225
	U+02c9	macron	modifier letter macron	♦	♦			
	U+02d6	tilde	nonspacing tilde				0xf7	217
♦	U+02d8	breve	breve	♦	♦		0xf9	219
♦	U+02d9	dotaccent	dot above	♦	♦		0xfa	220
♦	U+02da	ring	ring above	♦	♦		0xfb	221
♦	U+02db	ogonek	ogonek	♦	♦		0xfe	224
♦	U+02dc	tilde	nonspacing tilde	♦	♦	♦		
♦	U+02dd	hungarumlaut	modifier letter double prime	♦	♦		0xfd	223
	U+0384	tonos	greek tonos	♦				
	U+0385	dieresistonos	greek dialytika tonos	♦				
	U+0386	Alphatonos	greek capital letter alpha with tonos	♦				
	U+0387	anoteleia	greek ano teleia	♦				
	U+0388	Epsilontonos	greek capital letter epsilon with tonos	♦				
	U+0389	Etatonos	greek capital letter eta with tonos	♦				
	U+038a	Iotatonos	greek capital letter iota with tonos	♦				
	U+038c	Omicrontonos	greek capital letter omicron with tonos	♦				
	U+038e	Upsilontonos	greek capital letter upsilon with tonos	♦				
	U+038f	Omegatonos	greek capital letter omega with tonos	♦				
	U+0390	iotadieresistonos	greek small letter iota with dialytika and tonos	♦				
	U+0391	Alpha	greek capital letter alpha	♦				
	U+0392	Beta	greek capital letter beta	♦				
	U+0393	Gamma	greek capital letter gamma	♦	♦			
	U+0394	Delta	greek capital letter delta	♦				
	U+0395	Epsilon	greek capital letter epsilon	♦				
	U+0396	Zeta	greek capital letter zeta	♦				
	U+0397	Eta	greek capital letter eta	♦				
	U+0398	Theta	greek capital letter theta	♦	♦			
	U+0399	Iota	greek capital letter iota	♦				
	U+039a	Kappa	greek capital letter kappa	♦				
	U+039b	Lambda	greek capital letter lamda	♦				
	U+039c	Mu	greek capital letter mu	♦				
	U+039d	Nu	greek capital letter nu	♦				
	U+039e	Xi	greek capital letter xi	♦				
	U+039f	Omicron	greek capital letter omicron	♦				
	U+03a0	Pi	greek capital letter pi	♦				
	U+03a1	Rho	greek capital letter rho	♦				
	U+03a3	Sigma	greek capital letter sigma	♦				
	U+03a4	Tau	greek capital letter tau	♦				
	U+03a5	Upsilon	greek capital letter upsilon	♦				
	U+03a6	Phi	greek capital letter phi	♦	♦			
	U+03a7	Chi	greek capital letter chi	♦				
	U+03a8	Psi	greek capital letter psi	♦				
	U+03a9	Omega	greek capital letter omega	♦				
	U+03aa	Iotadieresis	greek capital letter iota with	♦				

Min	Unicode	PostScript Name	Descriptive Name	WGL4	UGL	Win31	MacChar	MacIndex
			dialytika					
	U+03ab	Upsilondieresis	greek capital letter upsilon with dialytika	♦				
	U+03ac	alphatonos	greek small letter alpha with tonos	♦				
	U+03ad	epsilontonos	greek small letter epsilon with tonos	♦				
	U+03ae	etatonos	greek small letter eta with tonos	♦				
	U+03af	iotatonos	greek small letter iota with tonos	♦				
	U+03b0	upsilon-dieresistonos	greek small letter upsilon with dialytika and tonos	♦				
	U+03b1	alpha	greek small letter alpha	♦	♦			
	U+03b2	beta	greek small letter beta	♦				
	U+03b3	gamma	greek small letter gamma	♦				
	U+03b4	delta	greek small letter delta	♦	♦			
	U+03b5	epsilon	greek small letter epsilon	♦	♦			
	U+03b6	zeta	greek small letter zeta	♦				
	U+03b7	eta	greek small letter eta	♦				
	U+03b8	theta	greek small letter theta	♦				
	U+03b9	iota	greek small letter iota	♦				
	U+03ba	kappa	greek small letter kappa	♦				
	U+03bb	lambda	greek small letter lamda	♦				
	U+03bc	mu	greek small letter mu	♦				
	U+03bd	nu	greek small letter nu	♦				
	U+03be	xi	greek small letter xi	♦				
	U+03bf	omicron	greek small letter omicron	♦				
♦	U+03c0	pi	greek small letter pi	♦	♦		0xb9	155
	U+03c1	rho	greek small letter rho	♦				
	U+03c2	sigma1	greek small letter final sigma	♦				
	U+03c3	sigma	greek small letter sigma	♦	♦			
	U+03c4	tau	greek small letter tau	♦	♦			
	U+03c5	upsilon	greek small letter upsilon	♦				
	U+03c6	phi	greek small letter phi	♦	♦			
	U+03c7	chi	greek small letter chi	♦				
	U+03c8	psi	greek small letter psi	♦				
	U+03c9	omega	greek small letter omega	♦				
	U+03ca	iotadieresis	greek small letter iota with dialytika	♦				
	U+03cb	upsilondieresis	greek small letter upsilon with dialytika	♦				
	U+03cc	omicrontonos	greek small letter omicron with tonos	♦				
	U+03cd	upsilontonos	greek small letter upsilon with tonos	♦				
	U+03ce	omegatonos	greek small letter omega with tonos	♦				
	U+0401	afii10023	cyrillic capital letter io	♦				
	U+0402	afii10051	cyrillic capital letter dje	♦				
	U+0403	afii10052	cyrillic capital letter gje	♦				
	U+0404	afii10053	cyrillic capital letter ukrainian ie	♦				

♦ - Glyph in character set
☸ - Glyph not accessible from keyboard, only by PostScript name

Min	Unicode	PostScript Name	Descriptive Name	WGL4	UGL	Win31	MacChar	MacIndex
	U+0405	afii10054	cyrillic capital letter dze	♦				
	U+0406	afii10055	cyrillic capital letter byelorussian-ukrainian i	♦				
	U+0407	afii10056	cyrillic capital letter yi	♦				
	U+0408	afii10057	cyrillic capital letter je	♦				
	U+0409	afii10058	cyrillic capital letter lje	♦				
	U+040a	afii10059	cyrillic capital letter nje	♦				
	U+040b	afii10060	cyrillic capital letter tshe	♦				
	U+040c	afii10061	cyrillic capital letter kje	♦				
	U+040e	afii10062	cyrillic capital letter short u	♦				
	U+040f	afii10145	cyrillic capital letter dzhe	♦				
	U+0410	afii10017	cyrillic capital letter a	♦				
	U+0411	afii10018	cyrillic capital letter be	♦				
	U+0412	afii10019	cyrillic capital letter ve	♦				
	U+0413	afii10020	cyrillic capital letter ghe	♦				
	U+0414	afii10021	cyrillic capital letter de	♦				
	U+0415	afii10022	cyrillic capital letter ie	♦				
	U+0416	afii10024	cyrillic capital letter zhe	♦				
	U+0417	afii10025	cyrillic capital letter ze	♦				
	U+0418	afii10026	cyrillic capital letter i	♦				
	U+0419	afii10027	cyrillic capital letter short i	♦				
	U+041a	afii10028	cyrillic capital letter ka	♦				
	U+041b	afii10029	cyrillic capital letter el	♦				
	U+041c	afii10030	cyrillic capital letter em	♦				
	U+041d	afii10031	cyrillic capital letter en	♦				
	U+041e	afii10032	cyrillic capital letter o	♦				
	U+041f	afii10033	cyrillic capital letter pe	♦				
	U+0420	afii10034	cyrillic capital letter er	♦				
	U+0421	afii10035	cyrillic capital letter es	♦				
	U+0422	afii10036	cyrillic capital letter te	♦				
	U+0423	afii10037	cyrillic capital letter u	♦				
	U+0424	afii10038	cyrillic capital letter ef	♦				
	U+0425	afii10039	cyrillic capital letter ha	♦				
	U+0426	afii10040	cyrillic capital letter tse	♦				
	U+0427	afii10041	cyrillic capital letter che	♦				
	U+0428	afii10042	cyrillic capital letter sha	♦				
	U+0429	afii10043	cyrillic capital letter shcha	♦				
	U+042a	afii10044	cyrillic capital letter hard sign	♦				
	U+042b	afii10045	cyrillic capital letter yeru	♦				
	U+042c	afii10046	cyrillic capital letter soft sign	♦				
	U+042d	afii10047	cyrillic capital letter e	♦				
	U+042e	afii10048	cyrillic capital letter yu	♦				
	U+042f	afii10049	cyrillic capital letter ya	♦				
	U+0430	afii10065	cyrillic small letter a	♦				
	U+0431	afii10066	cyrillic small letter be	♦				
	U+0432	afii10067	cyrillic small letter ve	♦				
	U+0433	afii10068	cyrillic small letter ghe	♦				
	U+0434	afii10069	cyrillic small letter de	♦				
	U+0435	afii10070	cyrillic small letter ie	♦				
	U+0436	afii10072	cyrillic small letter zhe	♦				
	U+0437	afii10073	cyrillic small letter ze	♦				

Min	Unicode	PostScript Name	Descriptive Name	WGL4	UGL	Win31	MacChar	MacIndex
	U+0438	afii10074	cyrillic small letter i	♦				
	U+0439	afii10075	cyrillic small letter short i	♦				
	U+043a	afii10076	cyrillic small letter ka	♦				
	U+043b	afii10077	cyrillic small letter el	♦				
	U+043c	afii10078	cyrillic small letter em	♦				
	U+043d	afii10079	cyrillic small letter en	♦				
	U+043e	afii10080	cyrillic small letter o	♦				
	U+043f	afii10081	cyrillic small letter pe	♦				
	U+0440	afii10082	cyrillic small letter er	♦				
	U+0441	afii10083	cyrillic small letter es	♦				
	U+0442	afii10084	cyrillic small letter te	♦				
	U+0443	afii10085	cyrillic small letter u	♦				
	U+0444	afii10086	cyrillic small letter ef	♦				
	U+0445	afii10087	cyrillic small letter ha	♦				
	U+0446	afii10088	cyrillic small letter tse	♦				
	U+0447	afii10089	cyrillic small letter che	♦				
	U+0448	afii10090	cyrillic small letter sha	♦				
	U+0449	afii10091	cyrillic small letter shcha	♦				
	U+044a	afii10092	cyrillic small letter hard sign	♦				
	U+044b	afii10093	cyrillic small letter yeru	♦				
	U+044c	afii10094	cyrillic small letter soft sign	♦				
	U+044d	afii10095	cyrillic small letter e	♦				
	U+044e	afii10096	cyrillic small letter yu	♦				
	U+044f	afii10097	cyrillic small letter ya	♦				
	U+0451	afii10071	cyrillic small letter io	♦				
	U+0452	afii10099	cyrillic small letter dje	♦				
	U+0453	afii10100	cyrillic small letter gje	♦				
	U+0454	afii10101	cyrillic small letter ukrainian ie	♦				
	U+0455	afii10102	cyrillic small letter dze	♦				
	U+0456	afii10103	cyrillic small letter byelorussian-ukrainian i	♦				
	U+0457	afii10104	cyrillic small letter yi	♦				
	U+0458	afii10105	cyrillic small letter je	♦				
	U+0459	afii10106	cyrillic small letter lje	♦				
	U+045a	afii10107	cyrillic small letter nje	♦				
	U+045b	afii10108	cyrillic small letter tshe	♦				
	U+045c	afii10109	cyrillic small letter kje	♦				
	U+045e	afii10110	cyrillic small letter short u	♦				
	U+045f	afii10193	cyrillic small letter dzhe	♦				
	U+0490	afii10050	cyrillic capital letter ghe with upturn	♦				
	U+0491	afii10098	cyrillic small letter ghe with upturn	♦				
	U+1e80	Wgrave	latin capital letter w with grave	♦				
	U+1e81	wgrave	latin small letter w with grave	♦				
	U+1e82	Wacute	latin capital letter w with acute	♦				
	U+1e83	wacute	latin small letter w with acute	♦				
	U+1e84	Wdieresis	latin capital letter w with diaeresis	♦				

♦ - Glyph in character set
☸ - Glyph not accessible from keyboard, only by PostScript name

Min	Unicode	PostScript Name	Descriptive Name	WGL4	UGL	Win31	MacChar	MacIndex
	U+1e85	wdieresis	latin small letter w with diaeresis	♦				
	U+1ef2	Ygrave	latin capital letter y with grave	♦				
	U+1ef3	ygrave	latin small letter y with grave	♦				
♦	U+2013	endash	en dash	♦	♦	♦	0xd0	178
♦	U+2014	emdash	em dash	♦	♦	♦	0xd1	179
	U+2015	afii00208	horizontal bar	♦				
	U+2017	underscoredbl	double low line	♦	♦			
♦	U+2018	quoteleft	left single quotation mark	♦	♦	♦	0xd4	182
♦	U+2019	quoteright	right single quotation mark	♦	♦	♦	0xd5	183
♦	U+201a	quotesinglbase	single low-9 quotation mark	♦	♦	♦	0xe2	196
	U+201b	quotereversed	single high-reversed-9 quotation mark	♦				
♦	U+201c	quotedblleft	left double quotation mark	♦	♦	♦	0xd2	180
♦	U+201d	quotedblright	right double quotation mark	♦	♦	♦	0xd3	181
♦	U+201e	quotedblbase	double low-9 quotation mark	♦	♦	♦	0xe3	197
♦	U+2020	dagger	dagger	♦	♦	♦	0xa0	130
♦	U+2021	daggerdbl	double dagger	♦	♦	♦	0xe0	194
♦	U+2022	bullet	bullet	♦	♦	♦	0xa5	135
♦	U+2026	ellipsis	horizontal ellipsis	♦	♦	♦	0xc9	171
♦	U+2030	perthousand	per mille sign	♦	♦	♦	0xe4	198
	U+2032	minute	prime	♦				
	U+2033	second	double prime	♦				
♦	U+2039	guilsinglleft	single left-pointing angle quotation mark	♦	♦	♦	0xdc	190
♦	U+203a	guilsinglright	single right-pointing angle quotation mark	♦	♦	♦	0xdd	191
	U+203c	exclamdbl	double exclamation mark	♦	♦			
	U+203e	radicalex	overline	♦				
	U+2044	fraction	fraction slash	♦				
	U+207f	nsuperior	superscript latin small letter n	♦	♦			
♦	U+20a3	franc	french franc sign	♦	♦		☸	247
	U+20a4	afii08941	lira sign	♦				
	U+20a7	peseta	peseta sign	♦	♦			
	U+2105	afii61248	care of	♦				
	U+2113	afii61289	script small l	♦				
	U+2116	afii61352	numero sign	♦				
♦	U+2122	trademark	trademark sign	♦	♦	♦	0xaa	140
♦	U+2126	Ohm	ohm sign	♦	♦		0xbd	159
	U+212e	estimated	estimated symbol	♦				
	U+215b	oneeighth	vulgar fraction one eighth	♦				
	U+215c	threeeighths	vulgar fraction three eighths	♦				
	U+215d	fiveeighths	vulgar fraction five eighths	♦				
	U+215e	seveneighths	vulgar fraction seven eighths	♦				
	U+2190	arrowleft	leftwards arrow	♦	♦			
	U+2191	arrowup	upwards arrow	♦	♦			
	U+2192	arrowright	rightwards arrow	♦	♦			
	U+2193	arrowdown	downwards arrow	♦	♦			
	U+2194	arrowboth	left right arrow	♦	♦			
	U+2195	arrowupdn	up down arrow	♦	♦			

Min	Unicode	PostScript Name	Descriptive Name	WGL4	UGL	Win31	MacChar	MacIndex
	U+21a8	arrowupdnbse	up down arrow with base	♦	♦			
♦	U+2202	partialdiff	partial differential	♦	♦		0xb6	152
♦	U+2206	increment	increment	♦	♦		0xc6	168
♦	U+220f	product	n-ary product	♦	♦		0xb8	154
♦	U+2211	summation	n-ary summation	♦	♦		0xb7	153
♦	U+2212	minus	minus sign	♦	♦		☸	239
♦	U+2215	fraction	division slash		♦		0xda	188
♦	U+2219	periodcentered	bullet operator	♦	♦		0xe1	195
♦	U+221a	radical	square root	♦	♦		0xc3	165
♦	U+221e	infinity	infinity	♦	♦		0xb0	146
	U+221f	orthogonal	right angle	♦	♦			
	U+2229	intersection	intersection	♦	♦			
♦	U+222b	integral	integral	♦	♦		0xba	156
♦	U+2248	approxequal	almost equal to	♦	♦		0xc5	167
♦	U+2260	notequal	not equal to	♦	♦		0xad	143
	U+2261	equivalence	identical to	♦	♦			
♦	U+2264	lessequal	less-than or equal to	♦	♦		0xb2	148
♦	U+2265	greaterequal	greater-than or equal to	♦	♦		0xb3	149
	U+2302	house	house	♦	♦			
	U+2310	revlogicalnot	reversed not sign	♦	♦			
	U+2320	integraltp	top half integral	♦	♦			
	U+2321	integralbt	bottom half integral	♦	♦			
	U+2500	SF100000	box drawings light horizontal	♦	♦			
	U+2502	SF110000	box drawings light vertical	♦	♦			
	U+250c	SF010000	box drawings light down and right	♦	♦			
	U+2510	SF030000	box drawings light down and left	♦	♦			
	U+2514	SF020000	box drawings light up and right	♦	♦			
	U+2518	SF040000	box drawings light up and left	♦	♦			
	U+251c	SF080000	box drawings light vertical and right	♦	♦			
	U+2524	SF090000	box drawings light vertical and left	♦	♦			
	U+252c	SF060000	box drawings light down and horizontal	♦	♦			
	U+2534	SF070000	box drawings light up and horizontal	♦	♦			
	U+253c	SF050000	box drawings light vertical and horizontal	♦	♦			
	U+2550	SF430000	box drawings double horizontal	♦	♦			
	U+2551	SF240000	box drawings double vertical	♦	♦			
	U+2552	SF510000	box drawings down single and right double	♦	♦			
	U+2553	SF520000	box drawings down double and right single	♦	♦			
	U+2554	SF390000	box drawings double down and right	♦	♦			
	U+2555	SF220000	box drawings down single and left double	♦	♦			
	U+2556	SF210000	box drawings down double	♦	♦			

♦ - Glyph in character set
☸ - Glyph not accessible from keyboard, only by PostScript name

Min	Unicode	PostScript Name	Descriptive Name	WGL4	UGL	Win31	MacChar	MacIndex
			and left single					
	U+2557	SF250000	box drawings double down and left	♦	♦			
	U+2558	SF500000	box drawings up single and right double	♦	♦			
	U+2559	SF490000	box drawings up double and right single	♦	♦			
	U+255a	SF380000	box drawings double up and right	♦	♦			
	U+255b	SF280000	box drawings up single and left double	♦	♦			
	U+255c	SF270000	box drawings up double and left single	♦	♦			
	U+255d	SF260000	box drawings double up and left	♦	♦			
	U+255e	SF360000	box drawings vertical single and right double	♦	♦			
	U+255f	SF370000	box drawings vertical double and right single	♦	♦			
	U+2560	SF420000	box drawings double vertical and right	♦	♦			
	U+2561	SF190000	box drawings vertical single and left double	♦	♦			
	U+2562	SF200000	box drawings vertical double and left single	♦	♦			
	U+2563	SF230000	box drawings double vertical and left	♦	♦			
	U+2564	SF470000	box drawings down single and horizontal double	♦	♦			
	U+2565	SF480000	box drawings down double and horizontal single	♦	♦			
	U+2566	SF410000	box drawings double down and horizontal	♦	♦			
	U+2567	SF450000	box drawings up single and horizontal double	♦	♦			
	U+2568	SF460000	box drawings up double and horizontal single	♦	♦			
	U+2569	SF400000	box drawings double up and horizontal	♦	♦			
	U+256a	SF540000	box drawings vertical single and horizontal double	♦	♦			
	U+256b	SF530000	box drawings vertical double and horizontal single	♦	♦			
	U+256c	SF440000	box drawings double vertical and horizontal	♦	♦			
	U+2580	upblock	upper half block	♦				
	U+2584	dnblock	lower half block	♦				
	U+2588	block	full block	♦				
	U+258c	lfblock	left half block	♦				
	U+2590	rtblock	right half block	♦				
	U+2591	ltshade	light shade	♦				
	U+2592	shade	medium shade	♦				
	U+2593	dkshade	dark shade	♦				
	U+25a0	filledbox	black square	♦				
	U+25a1	H22073	white square	♦				

Min	Unicode	PostScript Name	Descriptive Name	WGL4	UGL	Win31	MacChar	MacIndex
	U+25aa	H18543	black small square	♦				
	U+25ab	H18551	white small square	♦				
	U+25ac	filledrect	black rectangle	♦				
	U+25b2	triagup	black up-pointing triangle	♦				
	U+25ba	triagrt	black right-pointing pointer	♦				
	U+25bc	triagdn	black down-pointing triangle	♦				
	U+25c4	triaglf	black left-pointing pointer	♦				
♦	U+25ca	lozenge	lozenge	♦			0xd7	185
	U+25cb	circle	white circle	♦				
	U+25cf	H18533	black circle	♦				
	U+25d8	invbullet	inverse bullet	♦				
	U+25d9	invcircle	inverse white circle	♦				
	U+25e6	openbullet	white bullet	♦				
	U+263a	smileface	white smiling face	♦				
	U+263b	invsmileface	black smiling face	♦				
	U+263c	sun	white sun with rays	♦				
	U+2640	female	female sign	♦				
	U+2642	male	male sign	♦				
	U+2660	spade	black spade suit	♦				
	U+2663	club	black club suit	♦				
	U+2665	heart	black heart suit	♦				
	U+2666	diamond	black diamond suit	♦				
	U+266a	musicalnote	eighth note	♦				
	U+266b	musicalnotedbl	beamed eighth notes	♦				
♦	U+f000	applelogo	apple logo				0xf0	210
♦	U+f001	fi	fi ligature[1]	♦			0xde	192
♦	U+f002	fl	fl ligature[2]	♦			0xdf	193
♦	U+fb01	fi	fi ligature	♦			0xde	192
♦	U+fb02	fl	fl ligature	♦			0xdf	193

[1] In order to preserve compatibility with HP printers, the fi ligature has two different unicode values associated with it, yet references a single glyph.
[2] In order to preserve compatibility with HP printers, the fl ligature has two different unicode values associated with it, yet references a single glyph.

♦ - Glyph in character set
☸ - Glyph not accessible from keyboard, only by PostScript name

Instructing Glyphs

This chapter gives an overview of the fundamental tasks involved in instructing a glyph.

Choosing a scan conversion setting

One of the key decisions to be made in instructing a TrueType font is the choice of scan conversion mode. Font designers can choose between a fast scan conversion mode and a dropout control scan conversion mode. This choice is made by setting the value of the Graphics State variable scan_control. The interpreter considers each of three conditions in determining whether dropout control mode will be used:

- Is the glyph rotated?
- Is the glyph stretched?
- Is the current setting for ppem less than a specified ppem value?

It is also possible to turn dropout control off completely.

Controlling rounding

The TrueType interpreter uses the round_state to determine the manner in which values will be rounded. Instructions are used to set the value of the round_state, a Graphics State variable. The setting of round_state determines how values will be rounded by the interpreter.

The instruction set makes it easy to set a number of predefined round states that will round values to the grid, pixel centers (half grid), or to either the grid or pixel centers. It is also possible to specify that values should be rounded down or rounded up. If none of the predefined rounding options suffices, the SROUND instructions provide very fine control of the rounding of values, making it possible to choose a phase, threshold, and period for the rounding function. The S45ROUND allows the same fine control as SROUND but is used when movement is along a 45 degree axis with the x-y plane.

A number of instructions round the value they obtain before moving any points. The effect of using any of the MDRP, MIRP, MIAP, MDAP, or ROUND instructions depends on the value of the round_state graphics state variable along with that of the control_value_cut_in.

The ROFF instruction turns off rounding but allows the instruction to continue looking at the cut-in value.

Points

Outline points are specified by their location in the coordinate grid and by whether they are *on* or *off* curve points. Managing a point means managing its position in space and its status as an *on* or *off* curve point. The interpreter uses zones and reference points to manage the set of points that comprise the current glyph and to refer to specific points within that set.

Zones

Any point the font scaler interpreter references is in one of two zones, that is one of two sets of points that potentially make up a glyph description. The first of these referenced zones is zone 1 (Z1) and always contains the glyph currently being interpreted.

The second, zone 0 (Z0), is used for temporary storage of point coordinates that do not correspond to any actual points in the glyph in zone 1. Zone 0 is useful when there is a need to manipulate a point that does not exist on the glyph or if you need to remember an intermediate point position. (This is the twilight zone.)

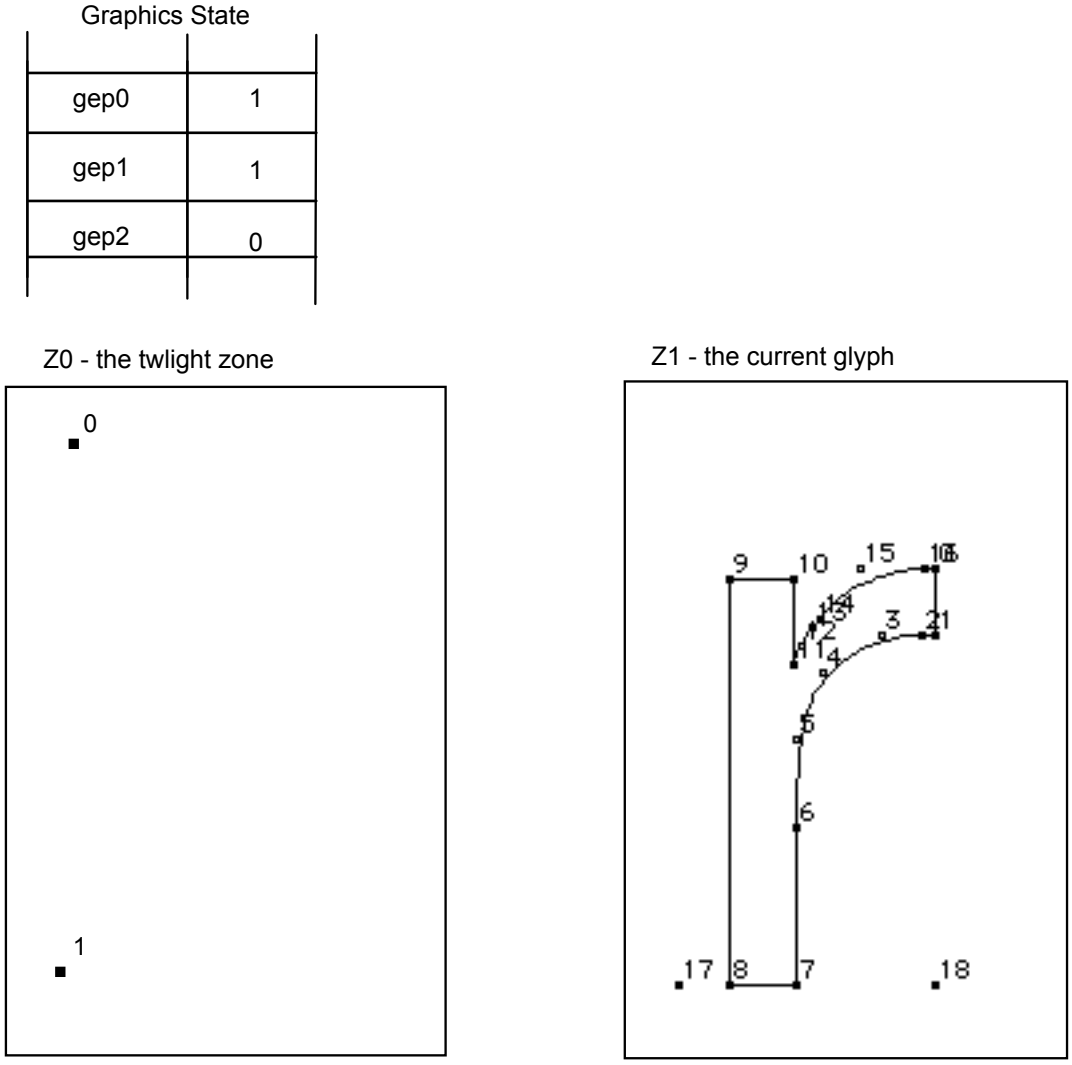
The profile table establishes the maximum number of twilight points. These are numbers 0 through maxTwilightPoints -1 and are all set to the origin. These points can be moved in the same manner as any of the points in zone 1.

Points in zone 0 are moved to useful positions by using the MIAP and MIRP instructions and setting gep0 to point to Z0. Frequently, it is useful to set points in Z0 to key metric positions for the font.

Zone pointers

Three zone pointers, gep0, gep1 and gep2 are used to reference either of zone 0 or zone 1. Initially, all three zone pointers will point to zone 1.

Figure 2–1 gep0 and gep1 point to zone 1 (the current glyph), gep2 points to zone 0 (the twilight zone)



Reference points

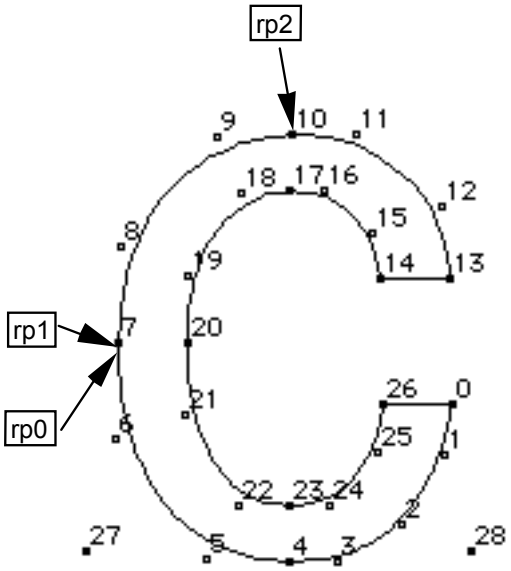
Zone pointers provide access to a group of points. Reference points provide access to specific points within the group. The interpreter uses three numbered reference points: rp0, rp1, and rp2. Each can be set to a number corresponding to any of the outline points in the glyph in zone 1 or any of the points in zone 0.

As shown in the following figure, two different reference points can refer to the same outline point.

Figure 2–2 A glyph pointed to by all three reference points

Graphics State

rp0	7
rp1	7
rp2	10



Collectively the zone pointers and reference points belong to the Graphics State. Their values can be altered using instructions. Many TrueType instructions rely on the graphics zone pointers and the reference points to fully specify their actions.

Phantom points

The font scaler will always add two “phantom points” to the end of every outline. If the entire set of contours for a glyph requires “n” points (i.e., contour points numbered from 0 to n-1), then the scaler will add points n and n+1. These points will be placed on the character baseline. Point “n” will appear at the character origin, while “n+1” will be placed at the advance width point.

Both points (n and n+1) may be controlled by TrueType instructions, with corresponding effects on the sidebearings and advance width of the instructed glyph. The side bearings and advance width that are computed using these phantom points are called the device-specific widths (since they reflect the results of grid fitting the width along with the glyph to the characteristics of the device). The device-specific widths can be different from or identical to the linearly scaled widths (obtained by simple scaling operations), depending on the instructions applied to the phantom points.

Determining distances

At the lowest level, instructing a glyph means managing the distances between points. The first step in managing a distance is often one of determining its magnitude. For example, the first step in setting up the Control Value Table involves measuring the distances between key points in a font. Measuring the distance between two points in a glyph outline, while not difficult, must take into account certain factors.

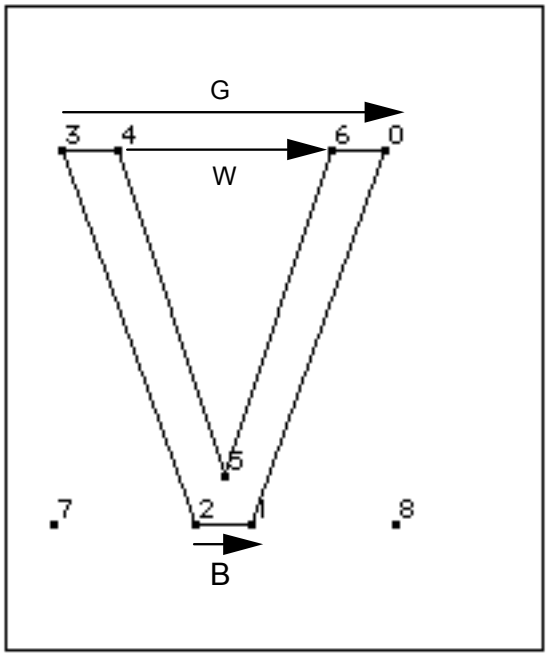
All distance measurements are made parallel to the projection_vector, a unit vector whose direction will be indicated by a radius of a circle. Distances are projected onto this vector and measured along it. Distances have a direction that reflects the direction of the vector.

Measurements can refer to the distance between points in the original character outline or between points in the grid-fitted outline. The instruction that measures distances (MD) takes a Boolean value which determines whether distances will be measured on the original outline or in the grid-fitted outline.

Additionally, the TrueType interpreter distinguishes between three different types of distances: black, white, and grey. Certain instructions (MDRP, MIRP, ROUND) require that you specify a distance type.

Black distances cross only black areas; white distances, white areas; and grey distances a combination of the two. In the following illustration, examples of black, white, and grey distances are shown. The distance [2,1] is black; [3,0] is grey and [4,6] is white.

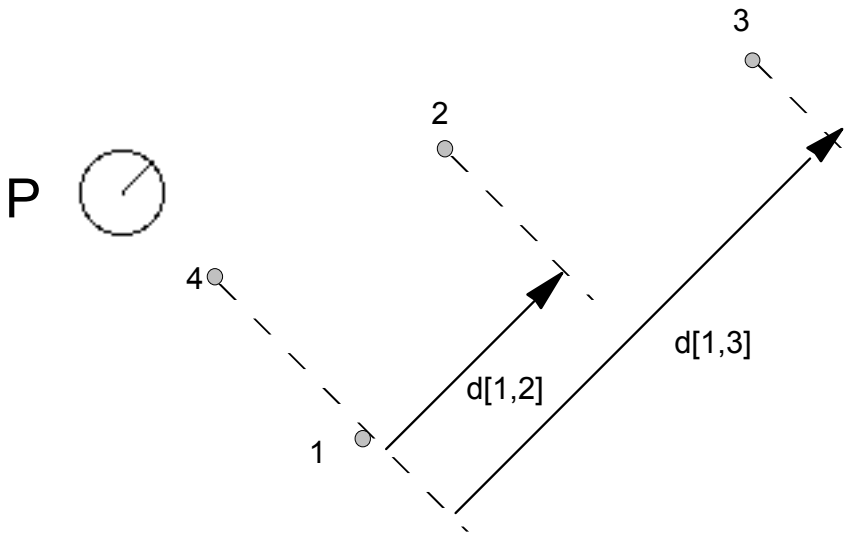
Figure 2–3 White, black and grey distances



The distance type is used in determining how the ROUND and instructions that use the round_state will work with different output devices. For gray distances, rounding is unaffected. Black or white distances, however, require a compensation term be added or subtracted before rounding takes place. The amount of compensation needed will be set by the device driver. For example, if a printing engine has large pixels, the interpreter will compensate by shrinking black distances and growing white distances. Gray distances, because they combine black and white distances, will not change.

When the distance between two points is determined, the distance is always measured in the direction specified by the projection_vector. Similarly, when a point is moved, the distance it is moved will be measured along the projection_vector. When thinking about how the interpreter will project a distance, you may find it convenient to imagine that distances are projected onto a ruled line that is parallel to the projection vector.

In the example shown, distances are measured along a line that is parallel to the projection_vector. The distance from point 1 to point 2 must be projected onto the projection_vector (that is a line parallel to the projection_vector) before being measured. Since the line from point 1 to point 3 is parallel to the vector, the projection of the distance can be thought of as simply the line from 1 to 3. Since the projection of the line from point 4 to point 1 is perpendicular to the vector, the distance from point 4 to point 1 is zero despite the fact that the points do not coincide.



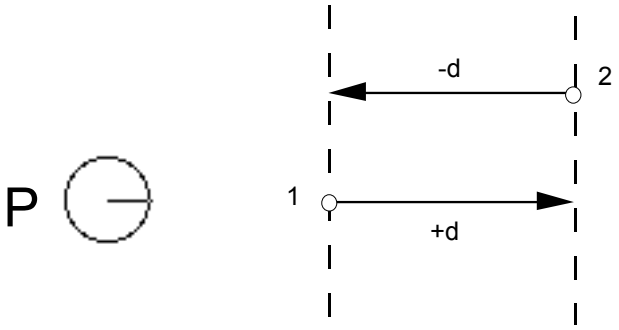
The vector can be set in any direction desired. In a simple case, the projection vector might be set to measure distance in the *x*-direction. In such a case, the vector is parallel to the *x*-axis. Similarly to measure distance in the *y*-direction, the projection_vector must be parallel to the *y*-axis.

To determine the distance between two points when the projection_vector points in the positive *x*-direction, one need only take the difference between their x coordinates. For example, the distance between the points (2, 1) and (7, 5) will be 5 units. Similarly, if the projection_vector pointed in the positive *y*-direction, the distance between the points would be 4 units.

Note that because the projection_vector has a direction, distances have a sign. Positive distances are those that are measured with the projection_vector. Negative distances are those that are measured against the projection_vector.

In the following example, the projection_vector points east (in the direction of the positive *x*-axis). The distance between points 1 and 2 is positive when measured from west to east (from point 1 to point 2). It is negative when measured from east to west (from point 2 to point 1).

Figure 2–4 Measuring the distance between two points



In many cases, it is convenient to disregard the sign associated with a distance. When the auto_flip Graphics State variable is set to TRUE, the sign of CVT entries will be changed when needed to match the sign of the actual measurement. This makes it possible to control distances measured with or against the projection_vector with a single CVT entry.

Controlling movement

The direction in which points can move is established by the Graphics State variable freedom_vector.

When a point is moved, its movement is constrained to be in a direction parallel to that of the freedom_vector. Assuming the freedom_vector is pointing in the direction of the positive *x*-axis (points east), movement in the positive *x*-direction (from west to east) will have a positive magnitude. Movement in the negative *x*-direction (from east to west) will have a negative magnitude.

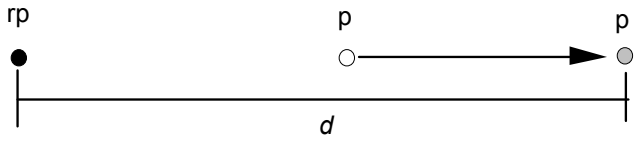
Moving points

WARNING: *When moving points, it is illegal for the freedom_vector and the projection_vector to be orthogonal.*

There are several instructions that move outline points. These instructions either move points relative to a reference point (the relative instructions) or move points to a specified location in the coordinate system (the absolute instructions).

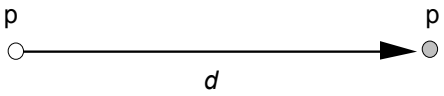
The following figure illustrates a relative move. The point p is moved so that it is at distance *d* from the reference point rp.

Figure 2–5 A relative move



The figure below illustrates an absolute move. Here the point p is moved a distance d from its current position to a new position. The distance is measured along the projection_vector. Movement is along the freedom_vector.

Figure 2–6 An absolute move



In specifying a move, some move instructions use the outline distance (direct instructions). Other instructions specify the value of *d* only indirectly by referring to a value in the CVT or to a value on the stack (indirect instructions).

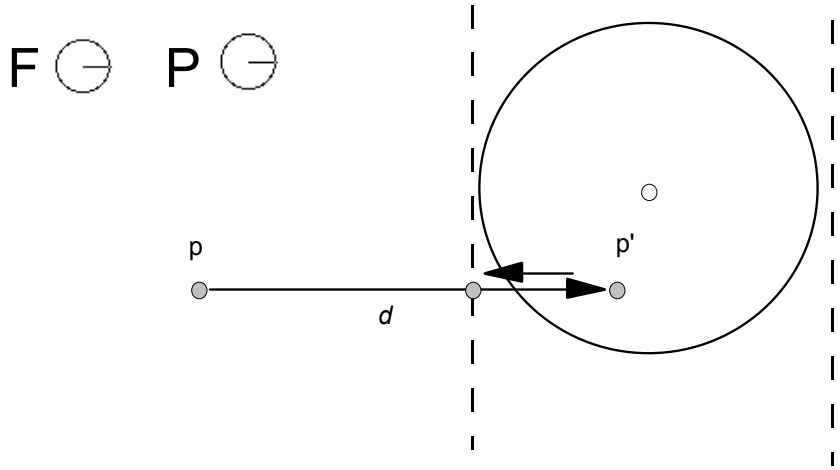
In attempting to move a point you must first decide on the direction and distance. Beyond this, decide whether you want to move that point an absolute distance or relative to another point. If the move is relative be sure you know which reference point will be used by the instruction. In some cases, you may need to change the value of that reference point to the one you desire. Finally decide whether you will use the original outline distance or will refer to a distance in the CVT or the stack.

By choosing to use the original outline distance you can preserve the original design distance between two points. In contrast, if you choose an indirect method of specifying a distance, that is you use the CVT, you allow that distance to be matched to some important value for that font or glyph.

All of the move instructions with the sole exception of MSIRP are affected by the round_state. The instructions allow you to choose whether they should take into account the setting of the round_state variable. In effect this means that, if rounding is turned on, the distance a point is actually moved will be affected by the type of rounding that is performed.

In the example below, point *p* is moved distance *d* to a new location *p'* and then rounded to the nearest grid boundary.

Figure 2–7 An absolute move with round_state set to round to grid



Managing the direction of distances

The auto_flip variable owes its existence to the fact that the TrueType interpreter distinguishes between distances measured in the direction of the projection_vector (positive distances) and those that are measured in the direction opposite to the projection_vector (negative distances).

The setting of the auto_flip Boolean determines whether the sign of values in the Control Value Table is significant. If auto_flip is set to TRUE, the values of CVT entries will be changed when necessary to match the sign of the actual measurement. This makes it possible to control distances measured with or against the projection vector.

For example, the CVT might contain an entry for uppercase stem widths. At times it may be convenient to control widths from left to right while at other times it may be convenient to control them from right to left. One case will produce a positive distance, the other a negative distance. Without auto_flip it would be necessary to have two CVT entries (+UC_Stem and –UC_Stem) instead of just one. Setting auto_flip to TRUE makes the sign of the value read from the CVT the same as the sign of the distance between the points we are controlling in the original unmodified domain.

Generally, auto_flip is set to TRUE, but if it becomes necessary to distinguish between a positive or negative distance, the variable must be set to FALSE.

Interpolating points

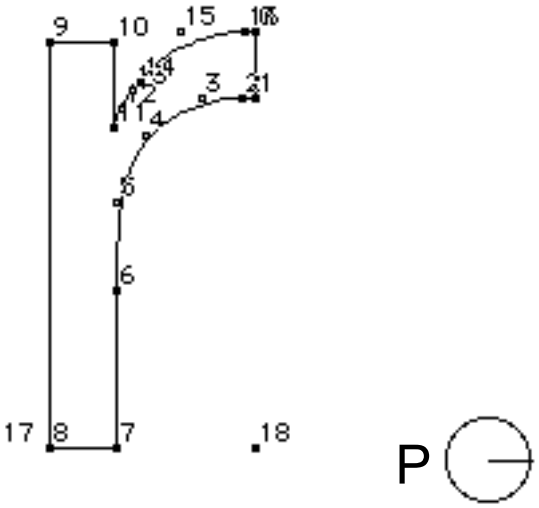
When instructions are used to change the position of a few of the points in a character outline, the curves that make up the character may become kinked or otherwise distorted. It may be desirable to smooth out the resulting curve. This smoothing out process is actually a redistribution of all points that have not been moved so that their positions relative to the moved points remain consistent.

To assist in managing the shape of outlines, the interpreter uses the concept of touching a point. Whenever an instruction has the effect of moving a point, that point is marked as touched in the x-direction or y-direction or both. The IUP instruction will affect only untouched points. It is possible to explicitly untouch a point so that it will be affected by an interpolation instruction.

Maintaining minimum_distance

When the width of a glyph feature decreases below a certain size, rounded values may become zero. Allowing values to round to zero can result in certain glyph features disappearing. For example, a stem might disappear entirely at small point sizes. By setting a minimum_distance of one pixel you can assure that even at small sizes those features will not disappear.

In the example shown below, the minimum_distance value is used to ensure that the stem of the r does not disappear at small sizes. By ensuring that the distance from point 9 to point 10 is always at least one pixel, this goal is accomplished.



Controlling regularization using the cut_in

The TrueType language offers several means of coordinating values for glyph features across a font. Such coordination results in a uniformity of appearance known as regularization. Regularization is useful when the number of pixels available for a feature or glyph are few in number. It prevents small differences in the size of features from becoming vastly exaggerated by the change in the placement of pixel centers within a glyph outline. Regularization becomes a liability when small differences in the size or placement of features can be effectively represented by the number of available pixels.

TrueType allows you the best of two worlds in making it possible to regularize features at small numbers of pixels per em while allowing the outline to revert to the original design once a sufficient number of pixels is available. There are two different ways to accomplish this goal. Each one uses a cut_in value. The first method uses the Control Value Table and the control_value_cut_in and allows you to coordinate values using entries in the CVT. This method allows for a variety of values to be coordinated. The second method takes regularization a step further and forces all values to revert to a single value. It relies on the single_width_cut_in and the single_width_value.

Control_value_cut_in

The control_value_cut_in makes it possible to limit the regularizing effects of the CVT to cases where the difference between the table value and the measurement taken from the original outline is sufficiently small. It allows the interpreter to choose, at some sizes, to use the CVT value while, at other sizes, to revert to the original outline. When the absolute difference between the value in the table and the measurement directly from the outline is greater than the cut_in value, the outline measurement is used.

The effect of the control_value_cut_in is to allow regularization below a certain cut off point while allowing the subtlety of the design to take over at larger sizes.

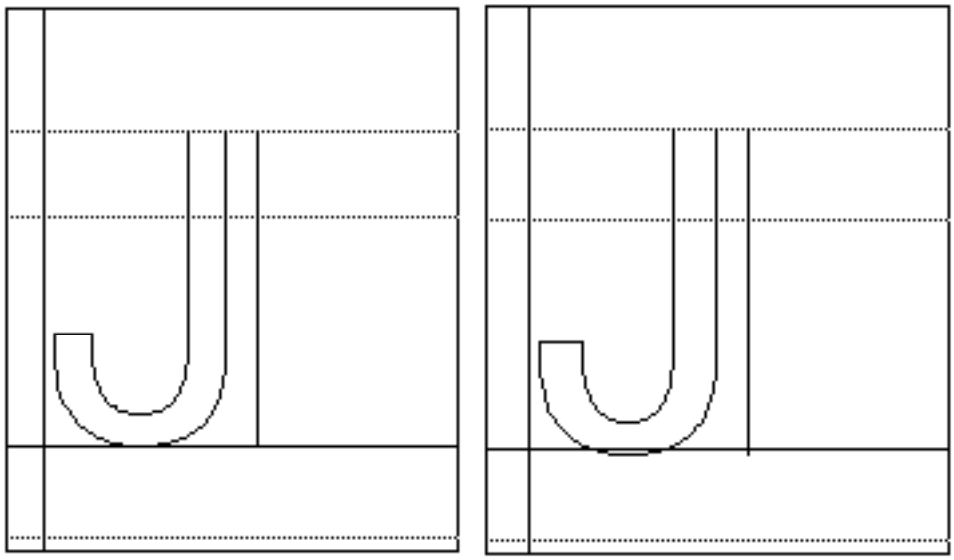
The cut_in value affects only instructions that refer to values in the CVT, the so called indirect instructions, MIRP and MIAP, and only if the third Boolean is set to TRUE.

CVT Value	Original Outline Value	|Diff|	cut_in[1]	|Diff| > cut_in use outline	|Diff| _ cut_in use CVT
93	80	13	17/16	80	–
100	99 5/16	11/16	17/16	–	100
97	95 15/16	17/16	17/16		97

[1] 17/16 is the default value for the control_value_cut_in

In the example shown, the capital J dips below the base line in the original design. When the character is grid-fitted, however, the curve is held to the baseline by an indirect instruction. That instruction references the CVT, subject to the default cut_in value of 17/16. At this value the curve is held to the base line through 81 pixels per em but reverts to its original design at 82 pixels per em as shown.

Figure 2–8 81 pixels per em (left) and 82 pixels per em (right)



The effect of the cut_in varies with its value. Decreasing the value of the cut_in will have the effect of causing the outline to revert to the original design at a smaller ppem value. Increasing the value of the cut_in will cause the outline to revert to the original design at a higher ppem value.

The single_width_cut_in

The single_width_cut_in is the distance difference at which the interpreter will ignore the values in the Control Value Table and in the outline in favor of a single-width value. It allows features to revert to a single predetermined size for small numbers of pixels per em.

Having all controlled glyph features assume the same dimensions might be an advantage for certain fonts at very small grid sizes. The single_width_value is used when the absolute difference between the single_width_value and the original value is smaller than this single_width_cut_in.

The default value for the single_width_cut_in is zero. In effect, this means that the default is ignore this cut_in value. Like the control_value_cut_in, the single_width_cut_in only applies to the indirect instructions.

The single_width_value

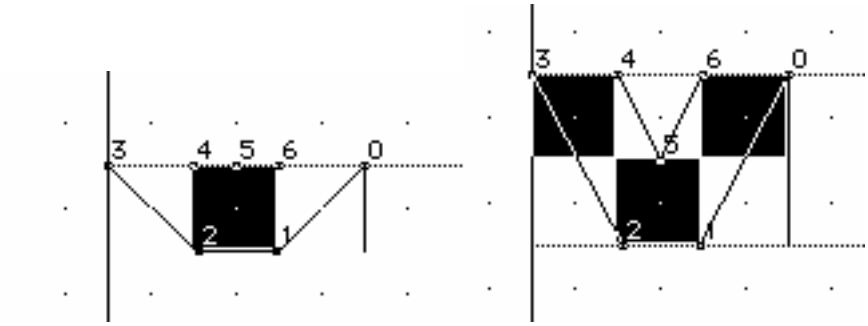
The single_width_value is used when the difference between the Control Value Table and the single_width_value is less than the single_width_cut_in. For example, if the single_width_value were set to 2 pixels, features meeting the single_width_cut_in test would be regularized to be 2 pixels wide.

Managing at specific sizes

Most TrueType instructions are independent of size. They are used to control a feature over the full range of sizes. Occasionally, it is necessary to alter a glyph outline at a specific size to include or exclude certain pixels. In other words, occasionally it is desirable to make an exception to the outline that would otherwise be produced by the other instructions. Such exceptions are made using the DELTA instructions.

There are two types of DELTA instructions. The DELTAP instructions work by moving points. The DELTAC instructions work by changing values in the CVT.

For example, without a DELTA instruction the circumflex accent shrinks to a single pixel at 9 ppem. Using DELTAs, the appearance is improved by lowering points 5, 2 and 1 by one pixel at 9 ppem.



Note: DELTA instructions should be used sparingly, since they are associated with a relatively high storage overhead. They can be useful for solving otherwise "impossible" cases.

The delta_base

The delta_base is the base value used to calculate the range of point sizes to which a delta instruction will apply. Changing the delta_base allows you to change the range of ppem sizes affected by each of the DELTA instructions.

The three pairs of DELTA instructions are grouped according to the range of pixels they potentially affect, with each group beginning at 16 pixels per em larger than the previous group. All DELTAC1 and DELTAP1 instructions potentially can affect glyphs at sizes beginning at delta_base pixels per em through delta_base plus 15 pixels per em. The DELTAP2 and DELTAC2 instructions affect the range beginning at delta_base plus 16 pixels per em. The DELTAP3 and DELTAC3 instructions affect delta_base plus 32 pixels per em.

The delta_shift

The delta_shift value is the power to which an exception is raised. By varying the value of the delta_shift, you trade off fine control of outline movement as opposed to total range of movement. A low delta_shift favors range of movement over fine control. A high delta_shift favors fine control over range of movement.

Points can be moved by multiples of a fixed amount called a step. The size of the step is 1 divided by 2 to the power delta_shift.

The TrueType Instruction Set

TrueType provides instructions for each of the following tasks and a set of general purpose instructions. This chapter describes the TrueType instruction set. Instruction descriptions are organized by category based on their function.

- Pushing data onto the interpreter stack
- Managing the Storage Area
- Managing the Control Value Table
- Modifying Graphics State settings
- Managing outlines
- General purpose instructions

Anatomy of a TrueType Instruction

TrueType instructions are uniquely specified by their opcodes. For convenience, this book will refer to instructions by their names. Each instruction name is a mnemonic intended to aid in remembering that instruction's function. For example, the MDAP instruction stands for Move Direct Absolute Point. Similarly, RUTG is short for Round Up To Grid. A brief description of each instruction clarifying the mnemonic marks the start of a new instruction.

One name may actually refer to several different but closely related instructions. A bracketed list of Boolean values follows each name to uniquely specify a particular variant of a given instruction. The Boolean list can be converted to a binary number and that number added to the base opcode for the instruction to obtain the opcode for any instruction variant.

To obtain the opcode for any instruction, take the lower of the two opcode values given in the code range and add the unsigned binary number represented by the list of binary digits. The left most bit is the most significant. For example, given an instruction with the opcode range 0xCO–0xDF and five Boolean flags (a through e) the opcode for a given instruction base can be computed as shown:

$$\text{Opcode} = 0xC0 + a \cdot 2^4 + b \cdot 2^3 + c \cdot 2^2 + d \cdot 2^1 + e \cdot 2^0$$

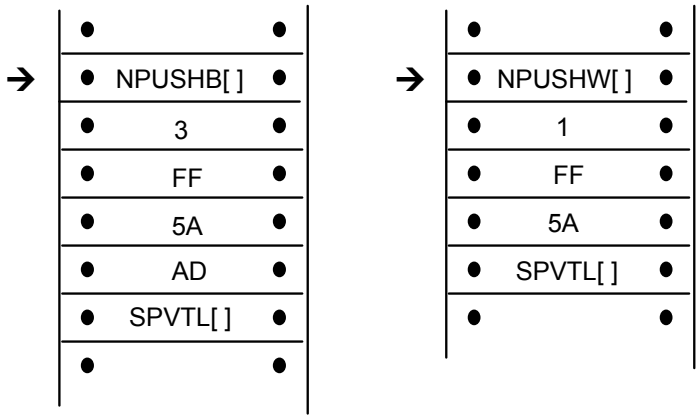
If these flags were set to 11101 the code would be computed as follows:

$$0xC0 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$
$$= 0xC0 + 0x10 + 0x8 + 0x4 + 0x1$$
$$= 0xC0 + 0x1D = 0xDD$$

Instruction opcodes are part of the instruction stream, a sequence of opcodes and data. The instruction stream is not a stack. While the stream of opcodes and data on the instruction stream is gradually used up, no new data is added to the instruction stream by the execution of another instruction (i.e. there is no equivalent of a push instruction that adds data to the instruction stream). It is possible to alter the flow of control through the instruction stream using one of the jump instructions described in a later section.

The instruction stream is shown as a sequence of opcodes and data. Since the instruction stream is 1-byte wide, words will be broken up into high bytes and low bytes with high bytes appearing first in the stream. For added readability, instruction names are used in illustrations instead of opcodes. An arrow will point to the next instruction awaiting execution.

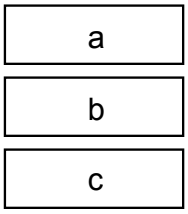
Figure 3–1 The instruction stream with a push byte instruction (left) and a push word instruction (right)



A few instructions known collectively as *push* instructions move data from the instruction stream to the interpreter stack. These instructions are unique in taking their arguments from the instruction stream. All other TrueType instructions take any data needed from the stack at the time they are executed. Any results produced by a TrueType instruction are pushed onto the interpreter stack.

An instruction that expects two arguments and pushes a third would expect the two arguments to be at the top of the stack. Any result pushed by that instruction appears at the top of the stack.

The listing *a b c* denotes a stack consisting of three elements with *a* being at the top of the stack, *b* being in the middle, and *c* at the bottom as shown.



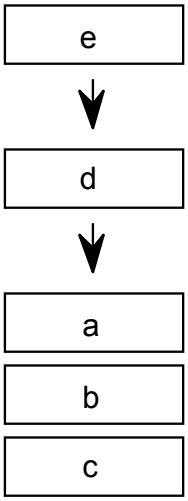
To easily remember the order in which stack values are handled during arithmetic or logical operations, imagine writing the stack values from left to right, starting with the bottom value. Then insert the operator between the two furthest right elements. For example, *subtract a,b* would be interpreted as (b-a):

c b - a

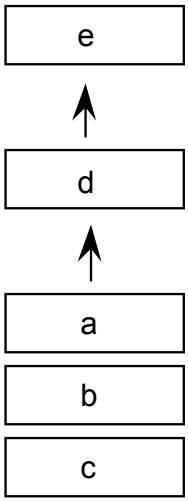
GT a,b would be interpreted as (b>a):

c b > a

The statement push *d*, *e* means push *d* then push *e* adding two elements to the stack as shown.



To indicate that the top two stack elements are to be removed the statement would be pop *e*, *d*.



It has already been noted that the bracketed list of binary digits that follows the instruction name uniquely identifies an instruction variant. This is done by having the bits represent a list of Boolean flags that can be set to TRUE with a value of 1 or to FALSE with a value of 0. Binary digits that follow the name can also be grouped to form a larger binary number. In such cases, the documentation specifies the meaning associated with each possible numerical combination.

An instruction specification consists of the instruction name followed by its bracketed Boolean flags. Additional information describing the flags and explaining the stack interaction and any Graphics State dependencies is provided in tabular form:

Code Range	the range of hexadecimal codes identifying this instruction and its variants
Flags	an explanation of the meaning of a bracketed binary number
From IS	any arguments taken from the instruction stream by push instructions
Pops	any arguments popped from the stack
Pushes	any arguments pushed onto the stack
Uses	any state variables whose value this instruction depends upon
Sets	any state variables set by this instruction

Instruction descriptions include illustrations intended to clarify stack interactions, Graphics State effects, and changes to interpreter tables.

In the case of instructions that move points, an illustration will be provided to clarify the direction and magnitude of the movement. In these illustrations, shades of gray will be used to indicate the sequence in which points have been moved. The darker the fill, the more recently a point has been moved.

Data types

The instruction stream

Instruction opcodes are always bytes. Values in the instruction stream are bytes.

The stack

Values pushed onto the stack or popped from the stack are always 32 bit quantities (LONG or ULONG). When values that are less than 32 bits are pushed onto the stack, bytes are expanded to 32 bit quantities by padding the upper bits with zeroes and words are sign extended to 32 bits. In cases where two instruction stream bytes are combined to form a word, the high order bits appear first in the instruction stream.

NOTE: On a 16-bit system, such as Windows, all stack operations are on 16-bit values (SHORT or USHORT). Care must be taken to avoid overflow. It is also important to note that F26dot6 values (used for internal scalar math) are represented instead as 10 dot 6 values (i.e. the upper 16 bits are not supported).

Figure 3–2 A byte padded to a 32 bit long word (ULONG)

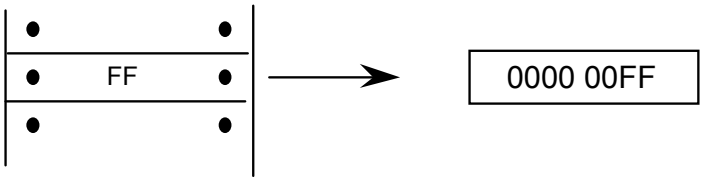
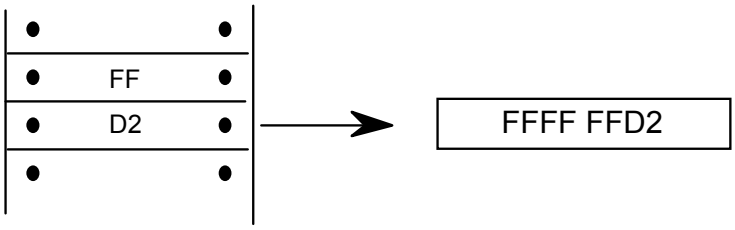


Figure 3–3 A word sign extended to a 32 bit long word (ULONG)



All values on the stack are signed. Instructions, however, interpret these 32-bit quantities in a variety of ways. The interpreter variously understands quantities as integers and as fixed point numbers.

Values such as pixel coordinates are represented as 32-bit quantities consisting of 26 bits of whole number and 6 bits of fraction. These are fixed point numbers with the data type name F26Dot6.

The setting of the freedom_vector and projection_vector are represented as 2.14 fixed point numbers. The upper 16 bits of the 32 bit quantity are ignored.

A given set of 32 bits will have a different value depending upon how it is interpreted. The following 32 bit value interpreted as an integer has the value 264.

```
0000 0000 0000 0000 0000 0001 0000 1000
```

The same 32 bit quantity interpreted as a F26Dot6 fixed point number has the value 4.125.

```
00 0000 0000 0000 0000 0000 0100 .00 1000
```

The figure below gives several examples of expressing pixel values as 26.6 words.

00000000000000000000000000 000001	one-sixty fourth of a pixel
00000000000000000000000000 100000	one-half pixel
00000000000000000000000001 000000	one pixel
00000000000000000000000001 100000	one and one half pixels

Pushing data onto the interpreter stack

Most TrueType instructions take their arguments from the interpreter stack. A few instructions, however, take their arguments from the instruction stream. Their purpose is to move data from the instruction stream to the interpreter stack. Collectively these instructions are known as the push instructions.

PUSH N Bytes

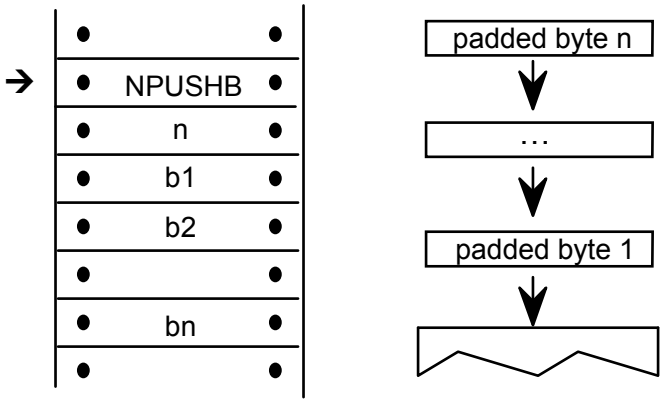
NPUSHB[ ]

Code Range 0x40

From IS n: number of bytes to push (1 byte interpreted as an integer)
b1, b2,...bn: sequence of n bytes

Pushes b1, b2,...bn: sequence of n bytes each padded to 32 bits (ULONG)

Takes *n* unsigned bytes from the instruction stream, where n is an unsigned integer in the range (0..255), and pushes them onto the stack. *n* itself is not pushed onto the stack.



PUSH N Words

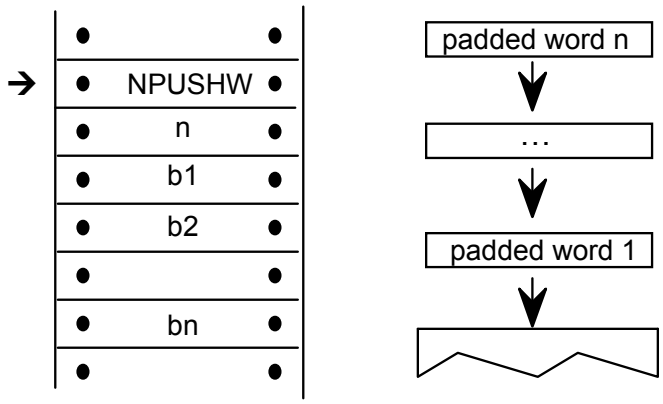
NPUSHW[ ]

Code Range 0x41

From IS n: number of words to push (one byte interpreted as an integer)
w1, w2,...wn: sequence of n words formed from pairs of bytes, the high byte appearing first

Pushes w1, w2,...wn: sequence of n words each sign extended to 32 bits (LONG)

Takes n 16-bit signed words from the instruction stream, where n is an unsigned integer in the range (0..255), and pushes them onto the stack. n itself is not pushed onto the stack.



PUSH Bytes

PUSHB[abc]

Code Range 0xB0 – 0xB7

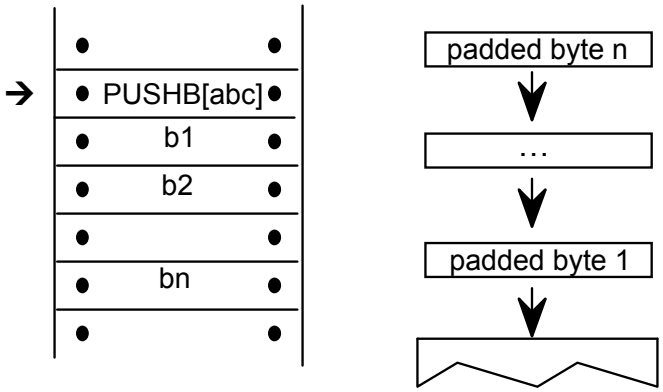
abcnumber of bytes to be pushed – 1

From IS b0, b1,..bn: sequence of n + 1 bytes

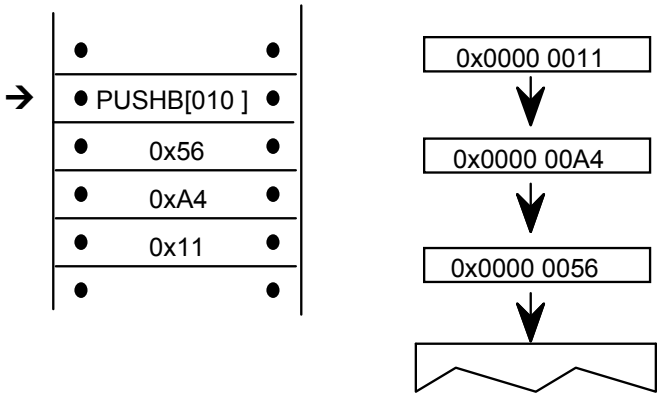
Pushes b0, b1, ...,bn: sequence of n + 1 bytes each padded to 32 bits (ULONG)

Takes the specified number of bytes from the instruction stream and pushes them onto the interpreter stack.

The variables a, b, and c are binary digits representing numbers from 000 to 111 (0-7 in binary). Because the actual number of bytes (n) is from 1 to 8, 1 is automatically added to the abc figure to obtain the actual number of bytes pushed.



Example:



PUSH Words

PUSHW[abc]

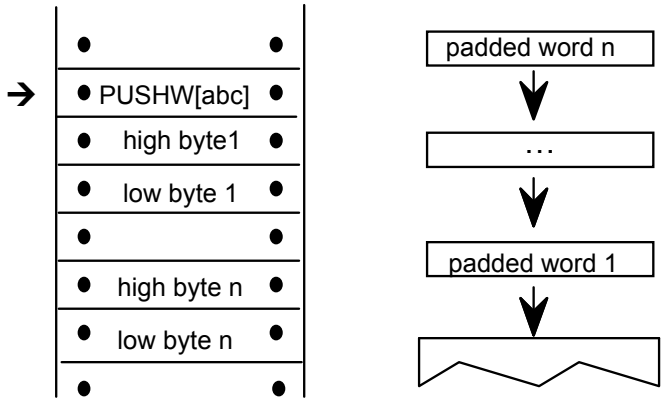
Code Range 0xB8 - 0xBF

abcnumber of words to be pushed – 1.

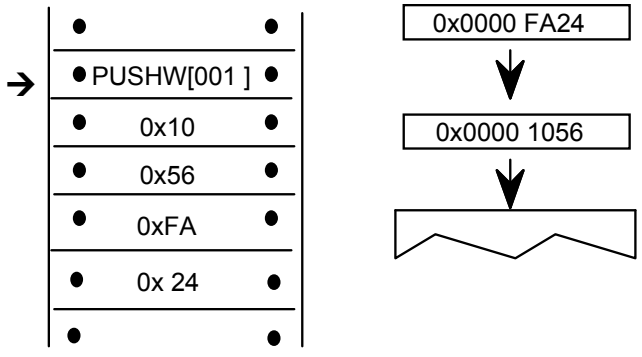
From IS	w0,w1,..wn:	sequence of n+1 words formed from pairs of bytes, the high byte appearing first
Pushes	w0 ,w1,...wn:	sequence of n+1 words each sign extended to 32 bits (LONG)

Takes the specified number of words from the instruction stream and pushes them onto the interpreter stack.

The variables a, b, and c are binary digits representing numbers from 000 to 111 (0-7 binary). Because the actual number of bytes (n) is from 1 to 8, 1 is automatically added to the abc figure to obtain the actual number of bytes pushed.



Example:



Managing the Storage Area

The interpreter Storage Area is a block of memory that can be used to store and later access 32 bit values. Instructions exist for writing values to the Storage Area and retrieving values from the Storage Area. Attempting to read a value from a storage location that has not previously had a value written to it will yield unpredictable results.

Read Store

RS[ ]

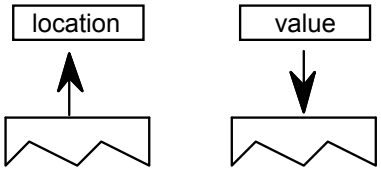
Code Range 0x43

Pops location: Storage Area location (ULONG)

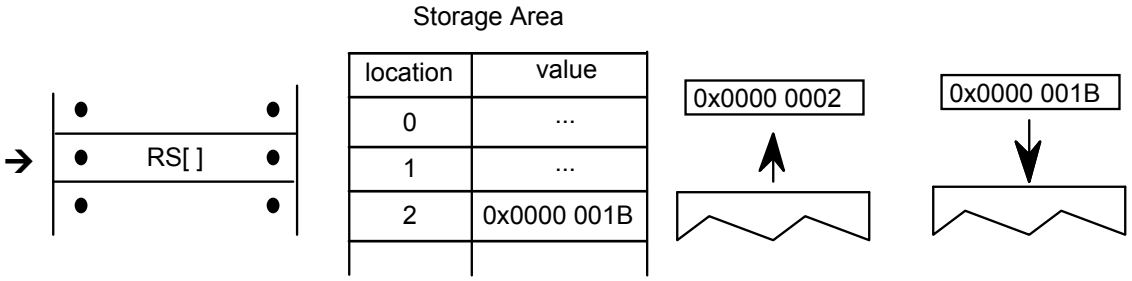
Pushes value: Storage Area value (ULONG)

Gets Storage Area value

This instruction reads a 32 bit value from the Storage Area location popped from the stack and pushes the value read onto the stack. It pops an address from the stack and pushes the value found in that Storage Area location to the top of the stack. The number of available storage locations is specified in the maxProfile table in the font file.



Example:

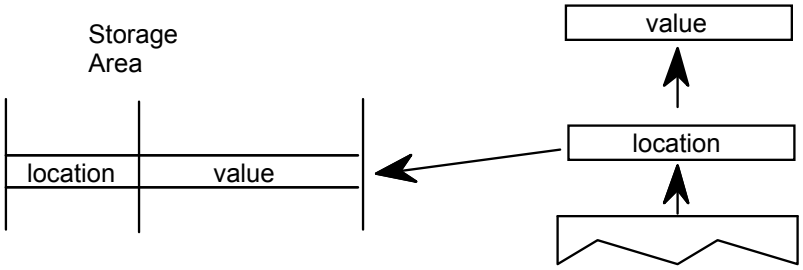


The effect of the RS instruction is to push the value 0x1B of the Storage Area onto the Stack.

Write Store

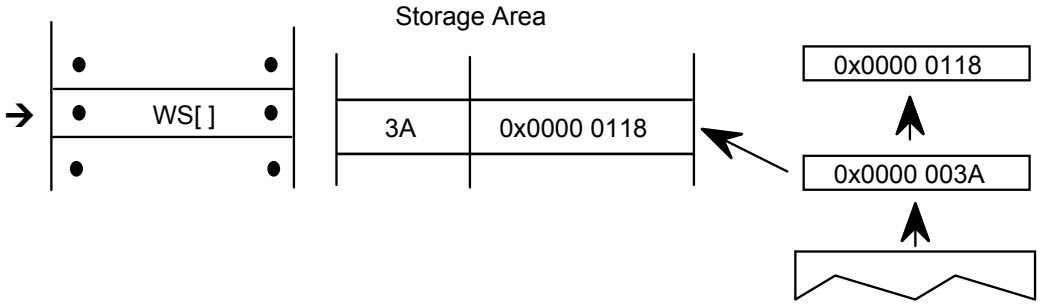
WS[ ]	
Code Range	0x42
Pops	value: Storage Area value (ULONG)
	location: Storage Area location (ULONG)
Pushes	–
Sets	Storage Area value

This instruction writes a 32 bit value into the storage location indexed by *locations*. It works by popping a value and then a location from the stack. The value is placed in the Storage Area location specified by that address. The number of storage locations is specified in the maxProfile table in the font file.



Example:

Write the value 0x0000 0118 to location 3A in the Storage Area.



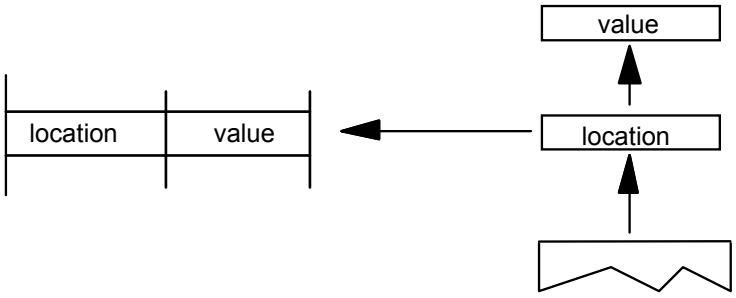
Managing the Control Value Table

The Control Value Table stores information that is accessed by the indirect instructions. Values can be written to the CVT in FUnits or pixel units as proves convenient. Values read from the CVT are always in pixels (F26Dot6). This table, unlike the Storage Area, is initialized by the font and is automatically scaled.

Write Control Value Table in Pixel units

WCVTP[ ]	
Code Range	0x44
Pops	value: number in pixels (F26Dot6 fixed point number)
	location: Control Value Table location (ULONG)
Pushes	–
Sets	Control Value Table entry

Pops a location and a value from the stack and puts that value in the specified location in the Control Value Table. This instruction assumes the value is in pixels and not in FUnits.



Write Control Value Table in FUnits

WCVTF[ ]

Code Range 0x70

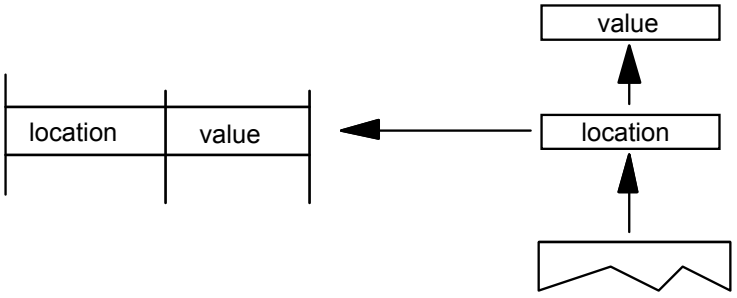
Pops value: number in FUnits (ULONG)

location: Control Value Table location (ULONG)

Pushes –

Sets Control Value Table entry

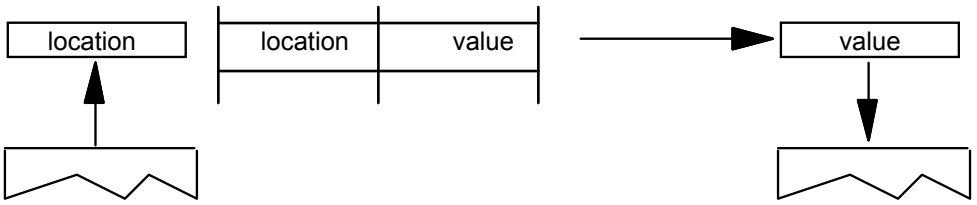
Pops a location and a value from the stack and puts the specified value in the specified address in the Control Value Table. This instruction assumes the value is expressed in FUnits and not pixels. The value is scaled before being written to the table.



Read Control Value Table

RCVT[ ]	
Code Range	0x45
Pops	location: CVT entry number (ULONG)
Pushes	value: CVT value (F26Dot6)
Gets	Control Value Table entry

Pops a location from the stack and pushes the value in the location specified in the Control Value Table onto the stack.



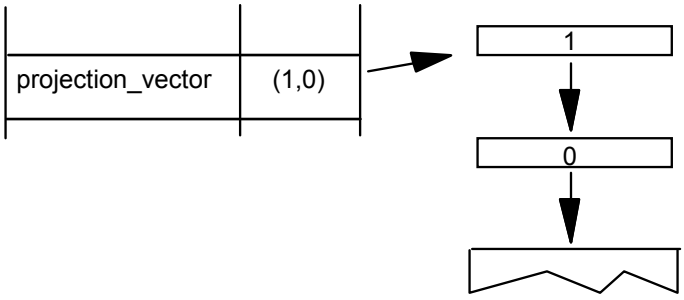
Managing the Graphics State

Instructions can be used to set the value of Graphics State variables and, in some cases, to retrieve their current value.

Getting a value

Instructions that retrieve the value of a state variable have names that begin with the word *get*. Get instructions will return the value of the state variable in question by placing that value on the top of the stack.

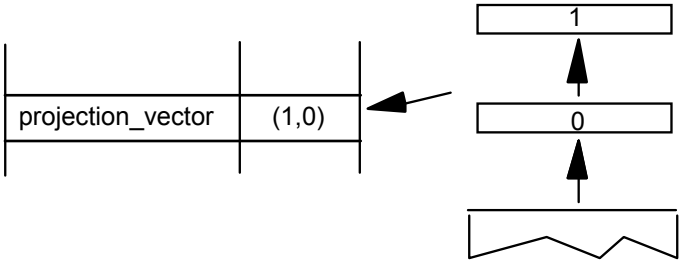
The illustration shows the effect of a GPV or get projection_vector instruction. It takes the *x* and *y* components of the projection_vector from the Graphics State and places them on the stack.



Setting a value

Instructions that change the value of a Graphics State variable have a name that begins with the word *set*. Set instructions expect their arguments to be at the top of the interpreter stack.

Figure 3–4 Setting the value of the Graphics State variable projection_vector



In addition to simple sets and gets, some instructions exist to simplify management of the values of state variables. For example, a number of instructions exist to set the direction of the freedom_vector and the projection_vector. In setting a vector, it is possible to set it to either of the coordinate axes, to the direction specified by a line, or to a direction specified by values taken from the stack. An instruction exists that directly sets the freedom_vector to the same value as the projection_vector.

Set freedom and projection Vectors To Coordinate Axis

SVTCA[a]	
Code range	0x00 - 0x01
a	0: set vectors to the *y*-axis 1: set vectors to the *x*-axis
Pops	–
Pushes	–
Sets	projection_vector freedom_vector

Sets both the projection_vector and freedom_vector to the same one of the coordinate axes.

The SVTCA is a shortcut for using both the SFVTCA and SPVTCA instructions. SVTCA[1] is equivalent to SFVTCA[1] followed by SPVTCA[1]. This instruction ensures that both movement and measurement are along the same coordinate axis.

Example:

SVTCA[1]

Sets both measurement and movement to the *x*-direction.

SVTCA[0]

Sets both measurement and movement to the *y*-direction.

Set Projection_Vector To Coordinate Axis

SPVTCA[a]	
Code range	0x02 - 0x03
a	0: set the projection_vector to the *y*-axis 1: set the projection_vector to the *x*-axis
Pops	–
Pushes	–
Sets	projection_vector

Sets the projection_vector to one of the coordinate axes depending on the value of the flag a.

Example:

SPVTCA[0]

Sets the projection_vector to the *y*-axis assuring the measurement will be in that direction.

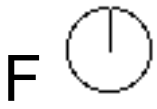
Set Freedom_Vector to Coordinate Axis

SFVTCA[a]	
Code range	0x04 - 0x05
a	0: set the freedom_vector to the *y*-axis 1: set the freedom_vector to the *x*-axis
Pops	–
Pushes	–
Sets	freedom_vector

Sets the freedom_vector to one of the coordinate axes depending upon the value of the flag a.

Example:

SFVTCA[0]

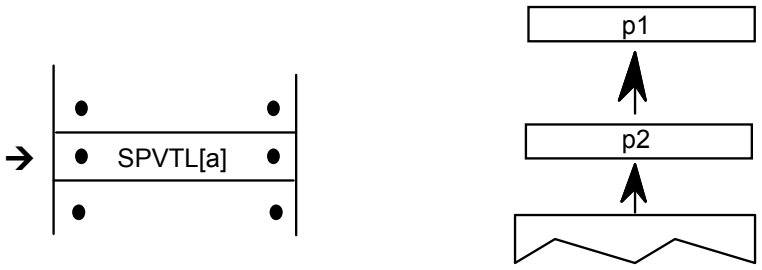


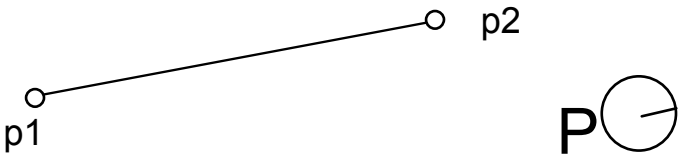
Sets the freedom_vector to the *y*-axis ensuring that movement will be along that axis.

Set Projection_Vector To Line

SPVTL[a]	
Code Range	0x06 - 0x07
a	0: sets projection_vector to be parallel to line segment from p1 to p2 1: sets projection_vector to be perpendicular to line segment from p1 to p2; the vector is rotated counter clockwise 90 degrees
Pops	p1: point number (ULONG) p2: point number (ULONG)
Pushes	–
Uses	point p1 in the zone pointed at by zp2 point p2 in the zone pointed at by zp1
Sets	projection_vector

Sets the projection_vector to a unit vector parallel or perpendicular to the line segment from point p1 to point p2.



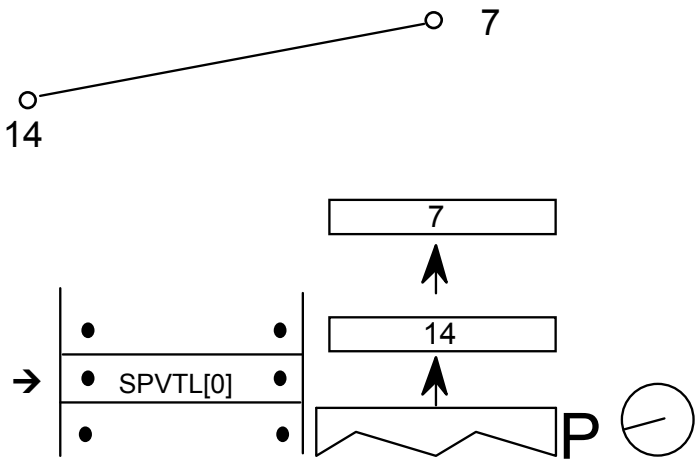


If parallel, the projection_vector points from p1 toward p2 as shown.

If perpendicular the projection_vector is obtained by rotating the parallel vector in a counter clockwise manner as shown.

case 1:

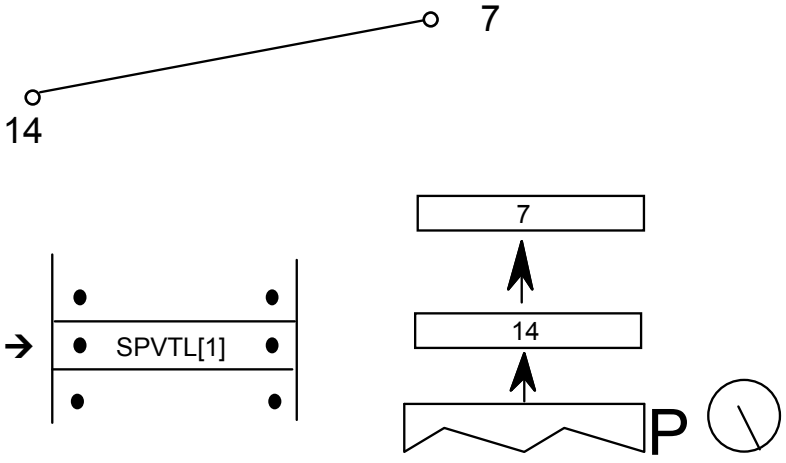
SPVTL[1]



Sets the projection_vector to be parallel to the line from point 7 to point 14.

case 2:

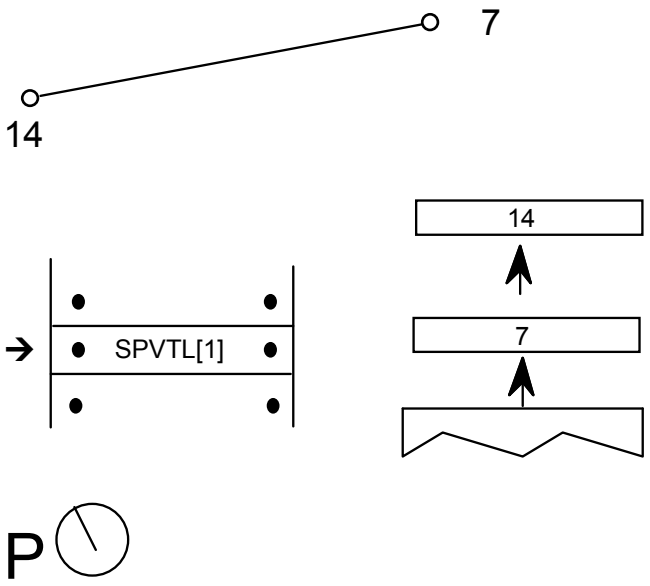
SPVTL[1]



Sets the projection_vector to be perpendicular to the line from point 7 to point 14.

case 3:

SPVTL[1]

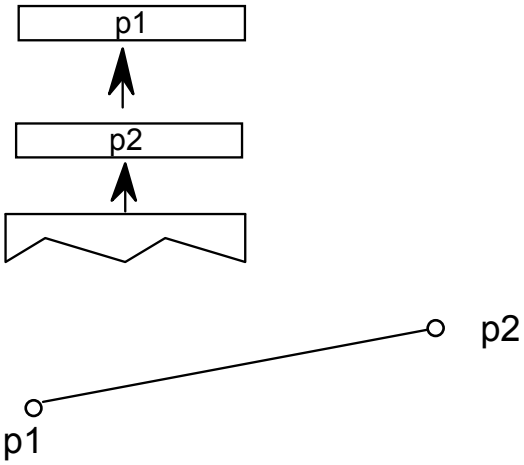


The order in which the points are specified matters. This instruction sets the projection_vector to be perpendicular to the line from point 14 to point 7.

Set Freedom_Vector To Line

SFVTL[a]	
Code Range	0x08 - 0x09
a	0: set freedom_vector to be parallel to the line segment defined by points p1 and p2
	1: set freedom_vector perpendicular to the line segment defined by points p1 and p2; the vector is rotated counter clockwise 90 degrees
Pops	p1: point number (ULONG)
	p2: point number (ULONG)
Pushes	–
Sets	freedom_vector
Uses	point p1 in the zone pointed at by zp2 point p2 in the zone pointed at by zp1

Sets the freedom_vector to a unit vector parallel or perpendicular to the line segment defined by points p1 and p2.



If parallel the freedom_vector points from p1 toward p2 as shown.

If perpendicular the freedom_vector is obtained by rotating the parallel vector in a counter clockwise manner as shown.

Set Freedom_Vector To Projection Vector

SFVTPV[ ]	
Code	0x0E
Pops	–
Pushes	–
Sets	freedom_vector

Sets the freedom_vector to be the same as the projection_vector.

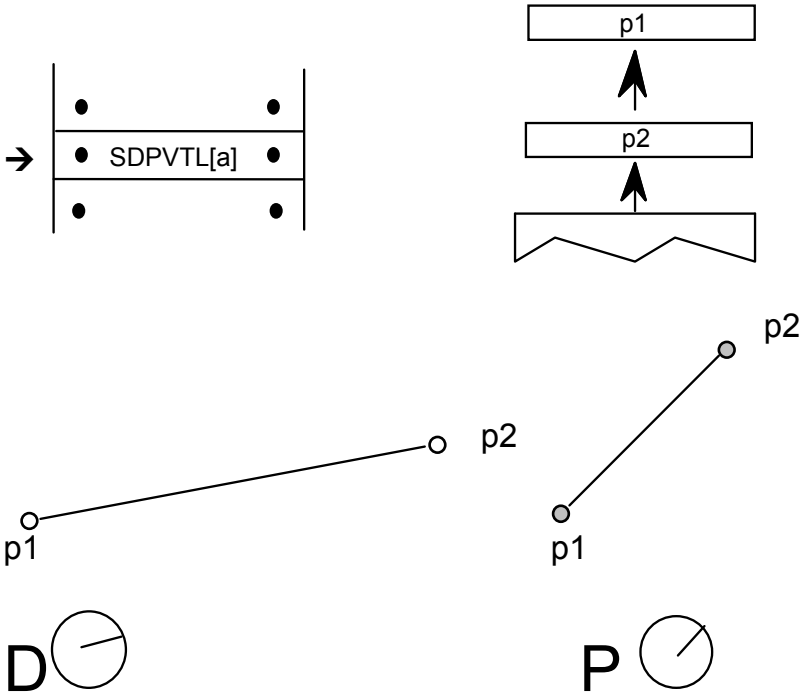
Before

After

Set Dual Projection_Vector To Line

SDPVTL[a]	
Code Range	0x86 - 0x87
a	0: Vectors are parallel to line 1: Vectors are perpendicular to line
Pops	p1: first point number (ULONG)
	p2: second point number (ULONG)
Pushes	–
Sets	dual_projection_vector and projection_vector
Uses	point p1 in the zone pointed at by zp2 point p2 in the zone pointed at by zp1

Pops two point numbers from the stack and uses them to specify a line that defines a second, dual_projection_vector. This dual_projection_vector uses coordinates from the scaled outline before any grid-fitting took place. It is used only with the IP, GC, MD, MDRP and MIRP instructions. Those instructions will use the dual_projection_vector when they measure distances between ungrid-fitted points. The dual_projection_vector will disappear when any other instruction that sets the projection_vector is used.



NOTE: The dual_projection_vector is set parallel to the points as they appeared in the original outline before any grid-fitting took place.

Set Projection_Vector From Stack

SPVFS[ ]

Code Range 0x0A

Pops *y*: *y* component of projection_vector (2.14 fixed point number padded with zeroes)

x: *x* component of projection_vector (2.14 fixed point number padded with zeroes)

Pushes –

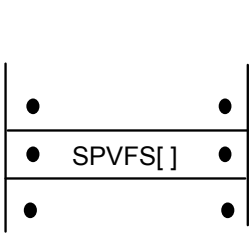
Sets projection_vector

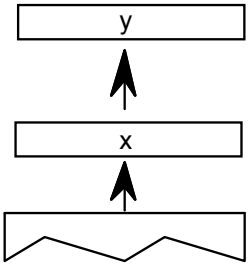
Sets the direction of the projection_vector, using values x and y taken from the stack, so that its projections onto the *x* and *y*-axes are *x* and *y*, which are specified as signed (two's complement) fixed-point (2.14) numbers. The square root of $(x^2 + y^2)$ must be equal to 0x4000 (hex).

If values are to be saved and used by a glyph program, font program or preprogram across different resolutions, extreme care must be used. The values taken from or put on the stack are 2.14 fixed-point values for the x and y components of the vector in question. The values are based on the normalized vector lengths. More simply, the values must always be set such that (X**2 + Y**2) is 1.

If a TrueType program uses specific values for X and Y to set the vectors to certain angles, these values will *not* produce identical results across different aspect ratios. Values that work correctly at 1:1 aspect ratios (such as VGA and 8514) will not necessarily yield the desired results at a ratio of 1.33:1 (e.g. the EGA).

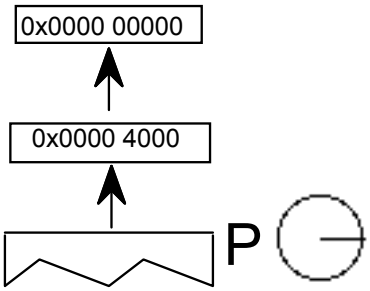
By the same token, if a TrueType program is making use of the values returned by GPV and GFV, the values returned for a specific angle will vary with the aspect ratio in use at the time.





Example:

SPVFS[ ]



Sets the projection_vector to a unit vector that points in the direction of the *x*-axis

Set Freedom_Vector From Stack

SFVFS[ ]

Code 0x0B

Pops *y:* *y* component of freedom_vector (2.14 fixed point number padded with zeroes)

x: *x* component of freedom_vector (2.14 fixed point number padded with zeroes)

Pushes –

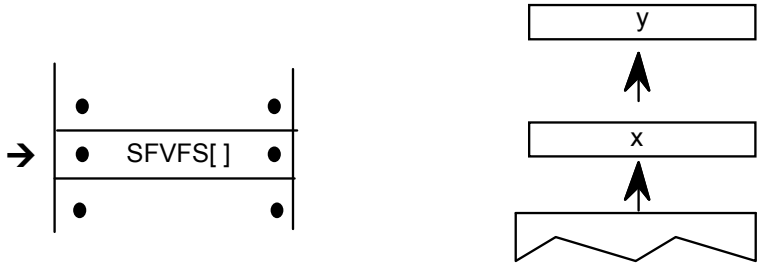
Sets freedom_vector

Sets the direction of the freedom_vector using the values x and y taken from the stack. The vector is set so that its projections onto the *x* and *y* -axes are *x* and *y*, which are specified as signed (two's complement) fixed-point (2.14) numbers. The square root of $(x^2 + y^2)$ must be equal to 0x4000 (hex).

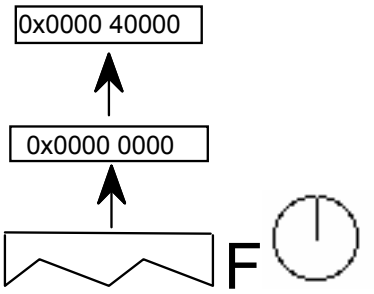
If values are to be saved and used by a glyph program, font program or preprogram across different resolutions, extreme care must be used. The values taken from or put on the stack are 2.14 fixed-point values for the x and y components of the vector in question. The values are based on the normalized vector lengths. More simply, the values must always be set such that (X**2 + Y**2) is 1.

If a TrueType program uses specific values for X and Y to set the vectors to certain angles, these values will *not* produce identical results across different aspect ratios. Values that work correctly at 1:1 aspect ratios (such as VGA and 8514) will not necessarily yield the desired results at a ratio of 1.33:1 (e.g. the EGA).

By the same token, if a TrueType program is making use of the values returned by GPV and GFV, the values returned for a specific angle will vary with the aspect ratio in use at the time.



Example:



Sets the freedom_vector to a unit vector that points in the direction of the *y*-axis.

Get Projection_Vector

GPV[ ]

Code Range 0x0C

Pops –

Pushes *x*: *x* component of projection_vector (2.14 fixed point number padded with zeroes)

y : *y* component of projection_vector (2.14 fixed point number padded with zeroes)

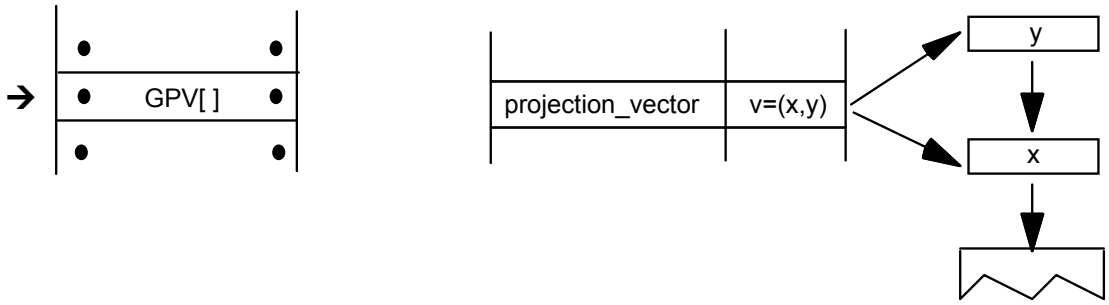
Gets projection_vector

Pushes the x and y components of the projection_vector onto the stack as two 2.14 numbers.

If values are to be saved and used by a glyph program, font program or preprogram across different resolutions, extreme care must be used. The values taken from or put on the stack are 2.14 fixed-point values for the x and y components of the vector in question. The values are based on the normalized vector lengths. More simply, the values must always be set such that (X**2 + Y**2) is 1.

If a TrueType program uses specific values for X and Y to set the vectors to certain angles, these values will *not* produce identical results across different aspect ratios. Values that work correctly at 1:1 aspect ratios (such as VGA and 8514) will not necessarily yield the desired results at a ratio of 1.33:1 (e.g. the EGA).

By the same token, if a TrueType program is making use of the values returned by GPV and GFV, the values returned for a specific angle will vary with the aspect ratio in use at the time.

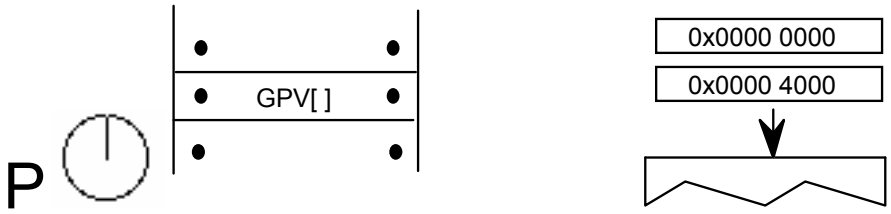


Example:

case 1:

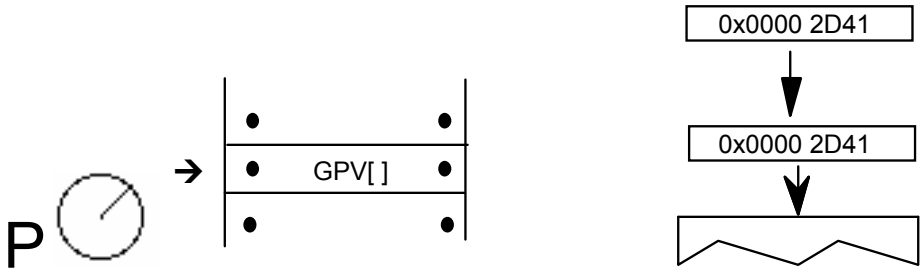
The stack entry 0x4000 which when interpreted as a 2.14 number is simply 1. This command reveals that, in this case, the projection_vector is a unit vector that points in the *x*-direction.

case 2:



Here the projection_vector is a unit vector that points in the direction of the *y*-axis.

case 3:



NOTE: 0x2D41 is the hex equivalent of $\frac{sqrt(2)}{2}$*. As a result of this instruction, the projection_vector is set to a 45 degree angle relative to the x-axis.*

Get Freedom_Vector

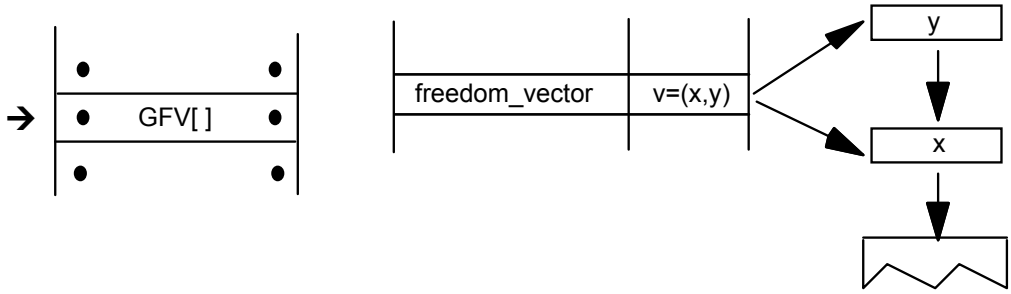
GFV[ ]	
Code Range	0x0D
Pops	–
Pushes	x: x-component of freedom_vector (2.14 number padded with zeroes)
	y: y component of freedom_vector (2.14 number padded with zeroes)
Gets	freedom_vector

Puts the *x* and *y* components of the freedom_vector on the stack. The freedom_vector is put onto the stack as two 2.14 coordinates.

If values are to be saved and used by a glyph program, font program or preprogram across different resolutions, extreme care must be used. The values taken from or put on the stack are 2.14 fixed-point values for the x and y components of the vector in question. The values are based on the normalized vector lengths. More simply, the values must always be set such that (X**2 + Y**2) is 1.

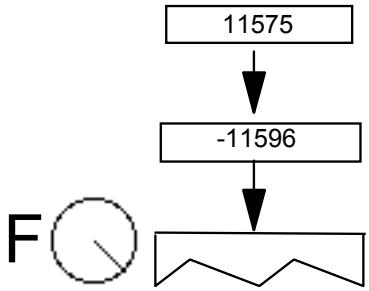
If a TrueType program uses specific values for X and Y to set the vectors to certain angles, these values will *not* produce identical results across different aspect ratios. Values that work correctly at 1:1 aspect ratios (such as VGA and 8514) will not necessarily yield the desired results at a ratio of 1.33:1 (e.g. the EGA).

By the same token, if a TrueType program is making use of the values returned by GPV and GFV, the values returned for a specific angle will vary with the aspect ratio in use at the time.



Example

GFV[ ]



Set Reference Point 0

SRP0[ ]

Code Range 0x10

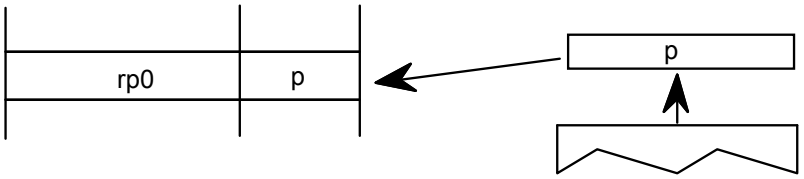
Pops p: point number (ULONG)

Pushes –

Sets rp0

Affects IP, MDAP, MIAP, MIRP, MSIRP, SHC, SHE, SHP

Pops a point number from the stack and sets rp0 to that point number.



Set Reference Point 1

SRP1[ ]

Code Range 0x11

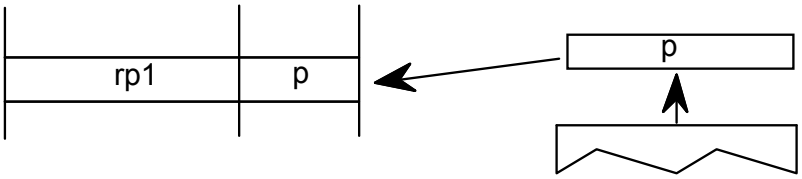
Pops p: point number (ULONG)

Pushes –

Sets rp1

Affects IP, MDAP, MDRP, MIAP, MSIRP, SHC, SHE, SHP

Pops a point number from the stack and sets rp1 to that point number.



Set Reference Point 2

SRP2[ ]

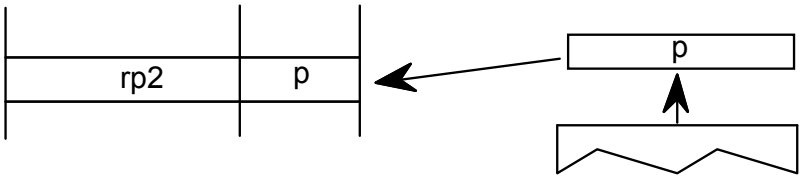
Code Range 0x12

Pops p:point number (ULONG)

Pushes –

Sets rp2

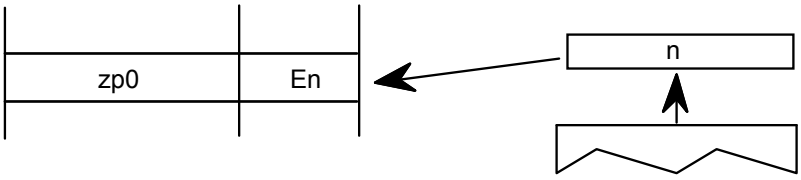
Pops a point number from the stack and sets rp2 to that point number.



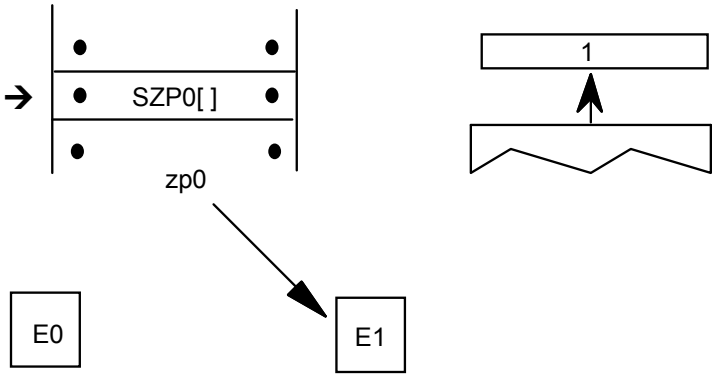
Set Zone Pointer 0

SZP0[ ]	
Code Range	0x13
Pops	n: zone number (ULONG)
Pushes	–
Sets	zp0
Affects	ALIGNPTS, ALIGNRP, DELTAP1, DELTAP2, DELTAP3, IP, ISECT, MD, MDAP, MIAP, MIRP, MSIRP, SHC, SHE, SHP, UTP

Pops a zone number, n, from the stack and sets zp0 to the zone with that number. If n is 0, zp0 points to zone 0. If n is 1, zp0 points to zone 1. Any other value for n is an error.



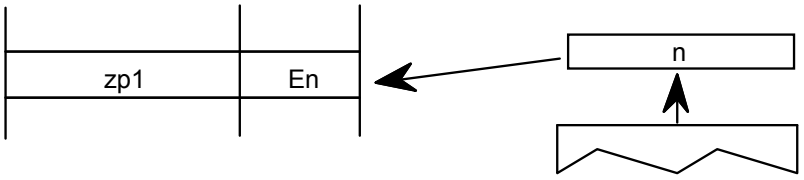
Example:



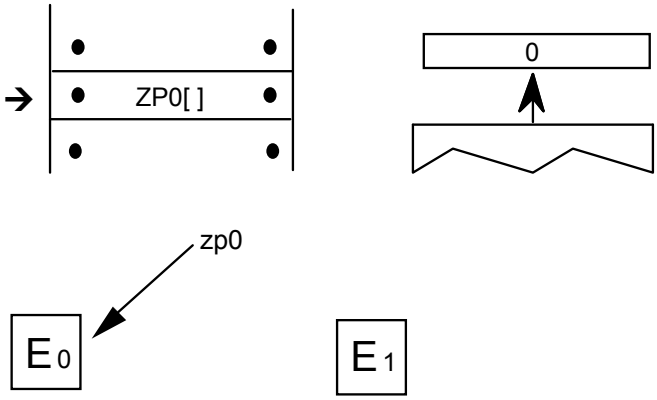
Set Zone Pointer 1

SZP1[ ]	
Code Range	0x14
Pops	n: zone number (ULONG)
Pushes	–
Sets	zp1
Affects	ALIGNRPTS, ALIGNRP, IP, MD, MDRP, MSIRP, SHC, SHE, SHP, SFVTL, SPVTL

Pops a zone number, n, from the stack and sets zp1 to the zone with that number. If n is 0, zp1 points to zone 0. If n is 1, zp1 points to zone 1. Any other value for n is an error.



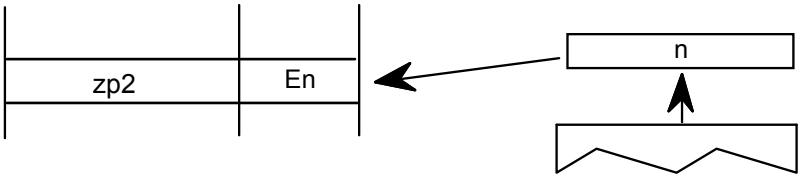
Example



Set Zone Pointer 2

SZP2[ ]	
Code Range	0x15
Pops	n: zone number (ULONG)
Pushes	–
Sets	zp2
Affects	ISECT, IUP, GC, SHC, SHP, SFVTL, SHPIX, SPVTL, SC

Pops a zone number, n, from the stack and sets zp2 to the zone with that number. If n is 0, zp2 points to zone 0. If n is 1, zp2 points to zone 1. Any other value for n is an error.



Set Zone PointerS

SZPS[ ]

Code Range 0x16

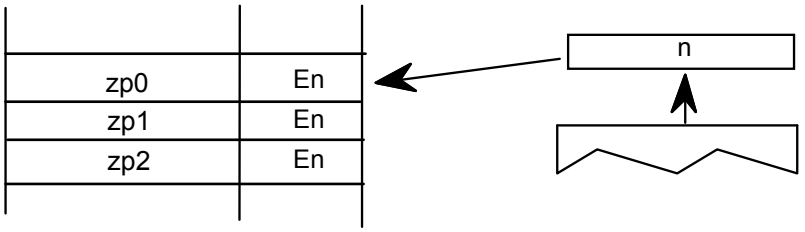
Pops n: zone number (ULONG)

Pushes –

Sets zp0, zp1, zp2

Affects ALIGNPTS, ALIGNRP, DELTAP1, DELTAP2, DELTAP3, GC, IP, ISECT, IUP, MD, MDAP, MDRP, MIAP, MIRP, MSIRP, SC, SFVTL, SHPIX, SPVTL, SHC, SHE, SHP, SPVTL, UTP

Pops a zone number from the stack and sets all of the zone pointers to point to the zone with that number. If n is 0, all three zone pointers will point to zone 0. If n is 1, all three zone pointers will point to zone 1. Any other value for n is an error.

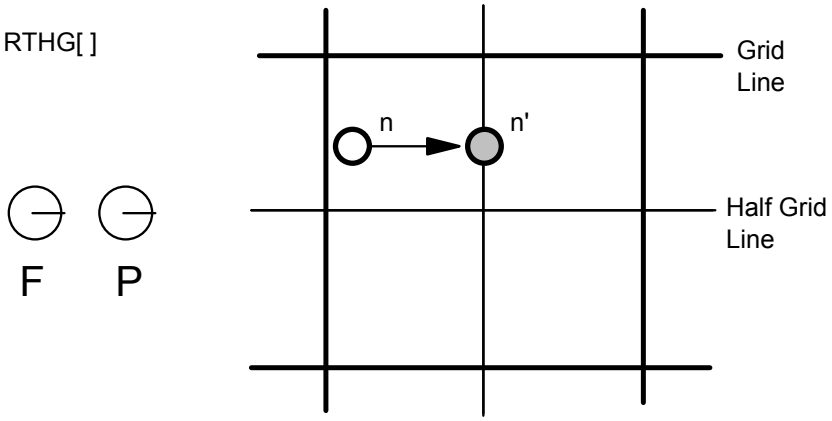


Round To Half Grid

RTHG[ ]	
Code Range	0x19
Pops	–
Pushes	–
Sets	round_state
Affects	MDAP, MDRP, MIAP, MIRP, ROUND
Uses	freedom_vector, projection_vector

Sets the round_state variable to state 0 (*hg*). In this state, the coordinates of a point are rounded to the nearest half grid line.

Example:



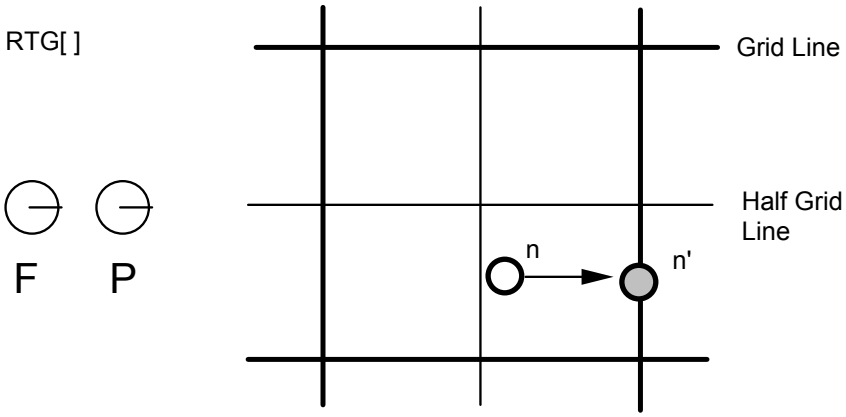
Round To Grid

RTG[ ]

Code Range	0x18
Pops	–
Pushes	–
Sets	round_state
Affects	MDAP, MDRP, MIAP, MIRP, ROUND
Uses	freedom_vector, projection_vector

Sets the round_state variable to state 1 (*g*). In this state, distances are rounded to the closest grid line.

Example:

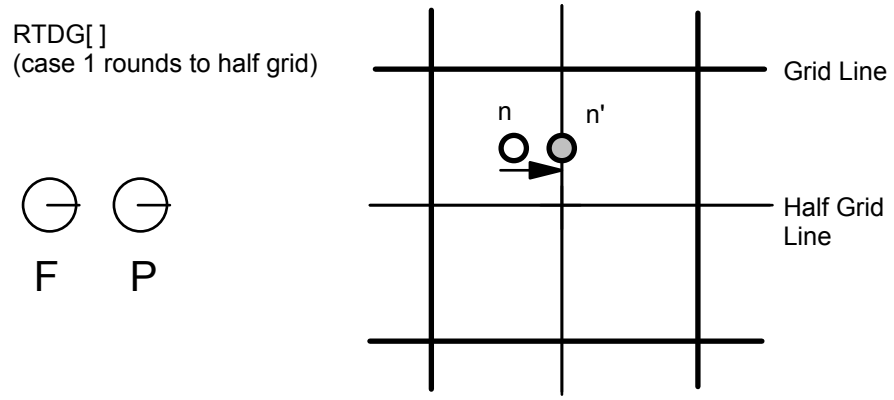


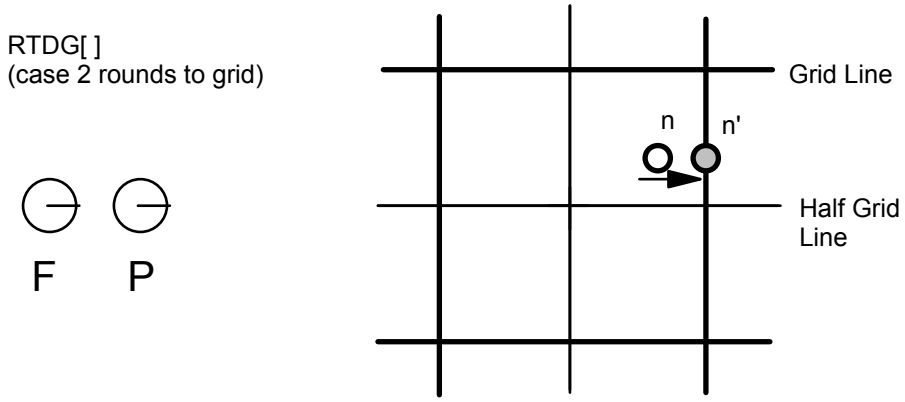
Round To Double Grid

RTDG[ ]	
Code Range	0x3D
Pops	–
Pushes	–
Sets	round_state
Affects	MDAP, MDRP, MIAP, MIRP, ROUND
Uses	freedom_vector, projection_vector

Sets the round_state variable to state 2 (*dg*). In this state, distances are rounded to the closest half or integer pixel.

Example:



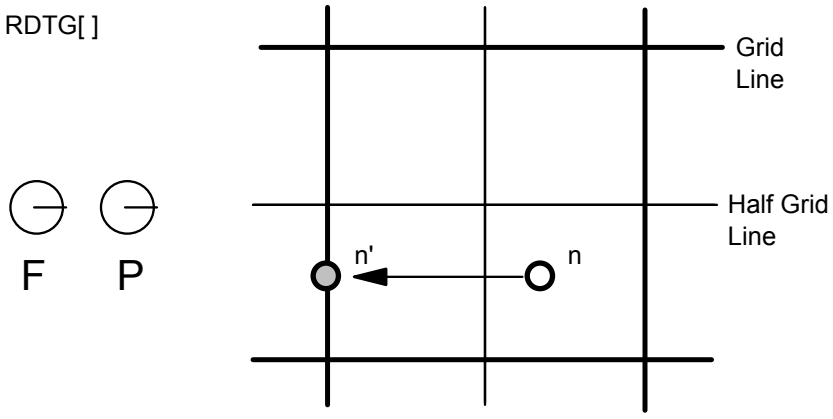


Round Down To Grid

RDTG[ ]	
Code Range	0x7D
Pops	–
Pushes	–
Sets	round_state
Affects	MDAP, MDRP, MIAP, MIRP, ROUND
Uses	freedom_vector, projection_vector

Sets the round_state variable to state 3 (*dtg*). In this state, distances are rounded down to the closest integer grid line.

Example:



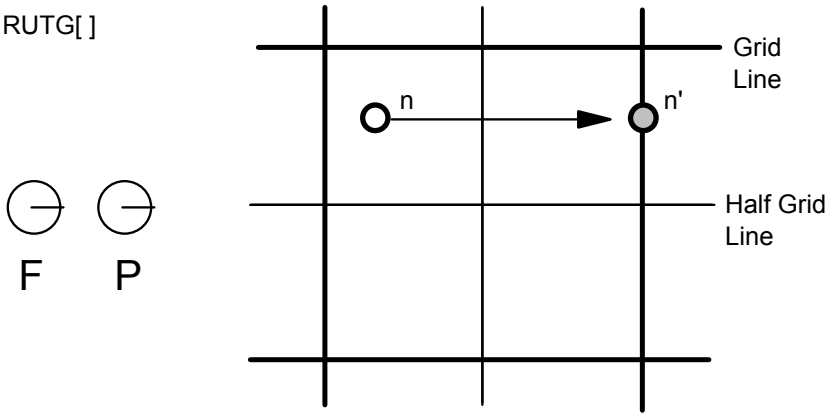
Round Up To Grid

RUTG[ ]	
Code Range	0x7C
Pops	–
Pushes	–
Sets	round_state
Affects	MDAP, MDRP, MIAP, MIRP, ROUND
Uses	freedom_vector, projection_vector

Sets the round_state variable to state 4 (*utg*). In this state distances are rounded up to the closest integer pixel boundary.

Example:

RUTG[ ]

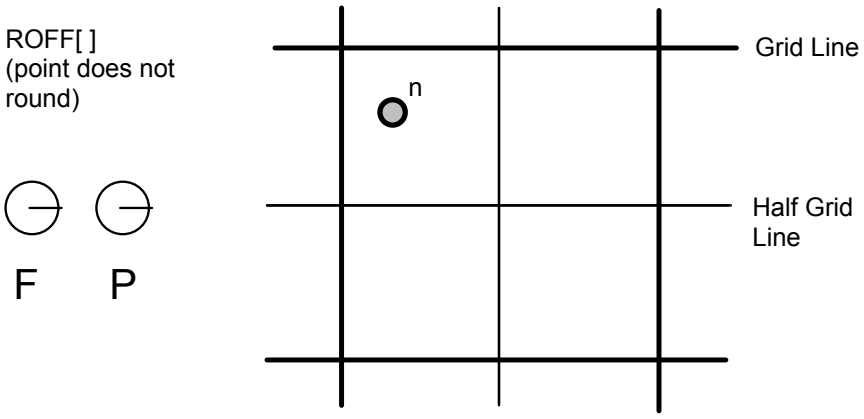


Round OFF

ROFF[ ]	
Code Range	0x7A
Pop	–
Pushes	–
Sets	round_state
Affects	MDAP, MDRP, MIAP, MIRP, ROUND
Uses	freedom_vector, projection_vector

Sets the round_state variable to state 5 (*off*). In this state rounding is turned off.

Example:



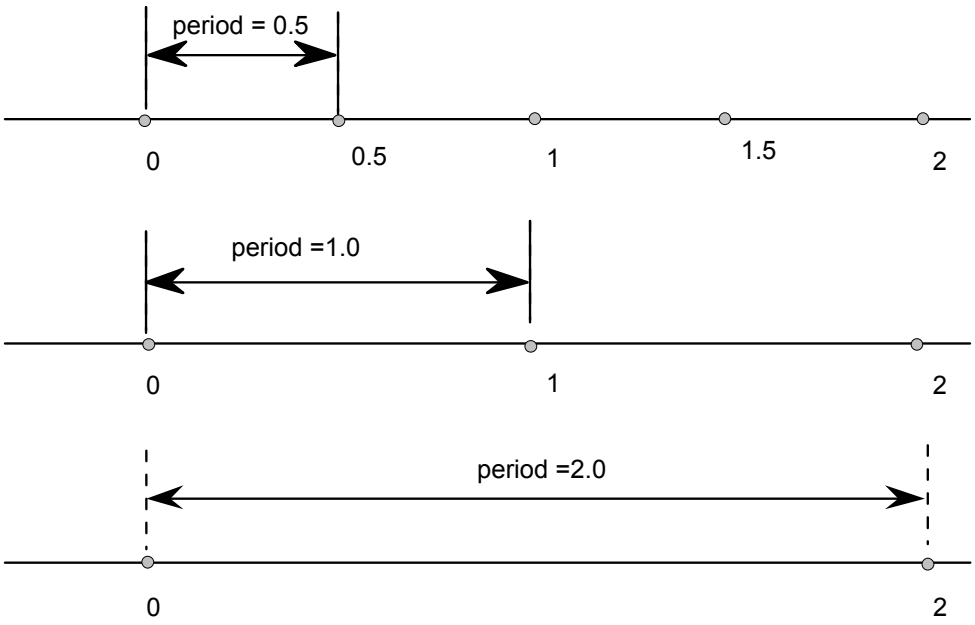
Super ROUND

SROUND[ ]	
Code Range	0x76
Pops	n: number decomposed to obtain period, phase, threshold
Pushes	–
Sets	round_state
Affects	MDAP, MDRP, MIAP, MIRP, ROUND

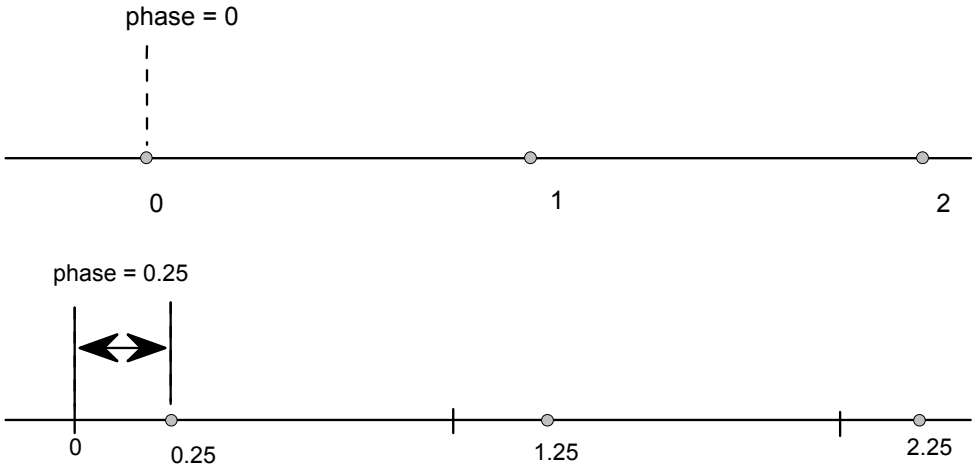
SROUND allows you fine control over the effects of the round_state variable by allowing you to set the values of three components of the round_state: period, phase, and threshold.

More formally, SROUND maps the domain of 26.6 fixed point numbers into a set of discrete values that are separated by equal distances. SROUND takes one argument from the stack, n, which is decomposed into a period, phase and threshold.

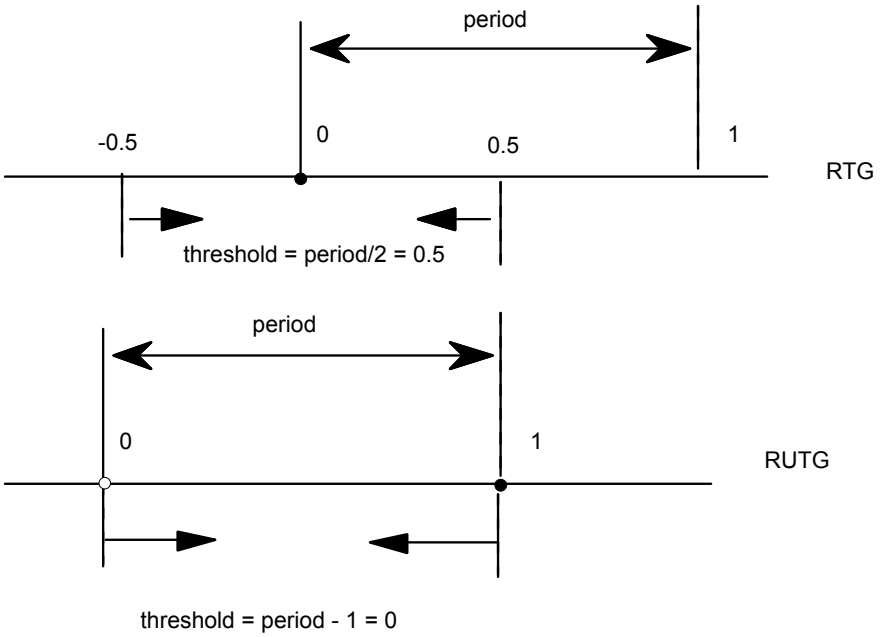
The period specifies the length of the separation or space between rounded values in terms of grid spacing.



The phase specifies the offset of the values from multiples of the period.



The threshold specifies the part of the domain that is mapped onto each value. More intuitively, the threshold tells a value when to “fall forward” to the next largest integer.



Only the lower 8 bits of the argument n are used. For SROUND gridPeriod is equal to 1.0 pixels. The byte is encoded as follows: bits 7 and 6 encode the period, bits 5 and 4 encode the phase and bits 3, 2, 1 and 0 encode the threshold as shown here.

period

0 period = gridPeriod/2

1 period = gridPeriod

2 period = gridPeriod*2

3 Reserved

(continued...)

phase

0 phase = 0

1 phase= period/4

2 phase = period/2

3 phase = gridPeriod*3/4

threshold

0 threshold = period -1

1 threshold = -3/8 * period

2 threshold = -2/8 * period

3 threshold = -1/8 * period

4 threshold = 0/8 * period

5 threshold = 1/8 * period

6 threshold = 2/8 * period

7 threshold =3/8 * period

8 threshold = 4/8 * period

9 threshold = 5/8 * period

10 threshold = 6/8 period

11 threshold = 7/8 * period

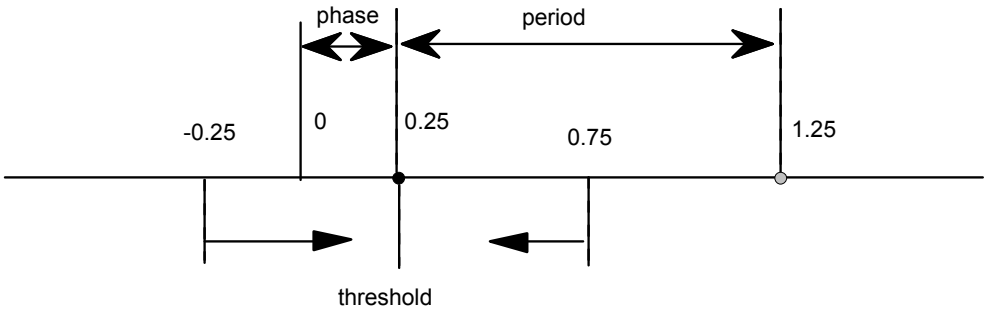
12 threshold = 8/8 * period

13 threshold = 9/8 * period

14 threshold = 10/8 * period

15 threshold = 11/8 * period

For example, SROUND(01:01:1000) maps numbers into the values 0.25, 1.25, 2.25, ... The numbers from -0.25 to 0.75 are mapped into 0.25. The range of numbers [0.75, 1.75) map into 1.25. Similarly, the numbers from [1.75, 2.75) map into the number 2.25 and so on.



Rounding occurs after compensation for engine characteristics, so the steps in the rounding of a number *n* are:

- add engine compensation to *n.*
- subtract the phase from *n.*
- add the threshold to *n.*
- truncate *n* to the next lowest periodic value (ignore the phase).
- add the phase back to *n*.
- if rounding caused a positive number to become negative, set *n* to the positive round value closest to 0.
- if rounding caused a negative number of become positive, set *n* to the negative round value closest to 0.
- the period parameters can have values of 1/2 pixel, 1 pixel, or 2 pixels.
- the phase parameters can have values of 0 pixels, 1/4 pixel, 1/2 pixel, or 3/4 pixel.
- the threshold parameters can have values of -3/8 period, -2/8 period,... 11/8 period. It can also have the special value largest-number-smaller-than-period which causes rounding equivalent to CEILING.

Super ROUND 45 degrees

S45ROUND[ ]

Code Range	0x77
Pops	n: ULONG decomposed to obtain period, phase, threshold (ULONG)
Pushes	–
Sets	round_state
Affects	MDAP, MDRP, MIAP, MIRP, ROUND

S45ROUND is analogous to SROUND. The gridPeriod is SQRT(2)/2 pixels rather than 1 pixel. It is useful for measuring at a 45 degree angle with the coordinate axes.

Set LOOP variable

SLOOP[ ]

Code Range 0x17

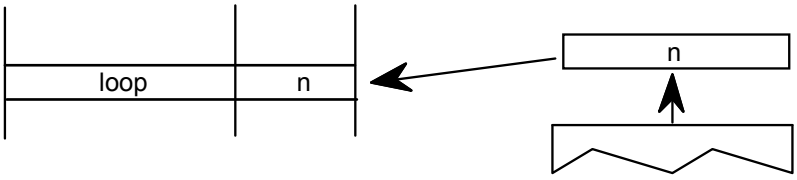
Pops n: value for loop Graphics State variable (integer)

Pushes –

Sets loop

Affects ALIGNRP, FLIPPT, IP, SHP, SHPIX

Pops a value, n, from the stack and sets the loop variable count to that value. The loop variable works with the SHP[a], SHPIX[a], IP[ ], FLIPPT[ ], and ALIGNRP[ ]. The value n indicates the number of times the instruction is to be repeated. After the instruction executes, the loop variable is reset to 1.



Set Minimum_ Distance

SMD[ ]

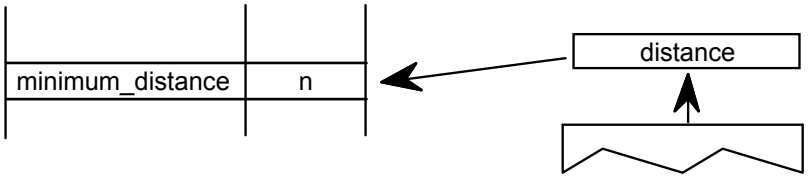
Code Range 0x1A

Pops distance: value for minimum_distance (F26Dot6)

Pushes –

Sets minimum_distance

Pops a value from the stack and sets the minimum_distance variable to that value. The distance is assumed to be expressed in sixty-fourths of a pixel.



INSTRuction execution ConTRoL

INSTCTRL[]

Code Range	0x8E	
Pops	s:	selector flag (int32)
	value:	USHORT (padded to 32 bits) used to set value of instruction_control.
Pushes	–	
Sets	instruction_control	

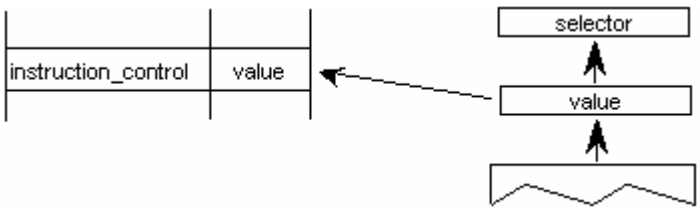
Sets the instruction control state variable making it possible to turn on or off the execution of instructions and to regulate use of parameters set in the CVT program. INSTCTRL[ ] can only be executed in the CVT program.

This instruction clears and sets various control flags in the rasterizer. The selector flag determines valid values for the value argument. The value determines the new setting of the raterizer control flag. In version 1.0 there are only two flags in use:

Selector flag 1 is used to inhibit grid-fitting. If s=1, valid values for the value argument are 0 (FALSE) and 1 (TRUE). If the value argument is set to TRUE (v=1), any instructions associated with glyphs will not be executed. For example, to inhibit grid-fitting when a glyph is being rotated or stretched, use the following sequence on the preprogram:

```
PUSHB[000] 6        /* ask GETINFO to check for stretching or rotation */
GETINFO[]           /* will push TRUE if glyph is stretched or rotated */
IF[]                /* tests value at top of stack */
PUSHB[000] 1        /* value for INSTCTRL */
PUSHB[000] 1        /* selector for INSTCTRL */
INSTRCTRL[]         /* based on selector and value will turn grid-fitting off */
EIF[]
```

Selector flag 2 is used to establish that any parameters set in the CVT program should be ignored when instructions associated with glyphs are executed. These include, for example, the values for scantype and the CVT cut-in. If s=1, valid values for the value argument are 0 (FALSE) and 2 (TRUE). If the value argument is set to TRUE (v=2), the default values of those parameters will be used regardless of any changes that may have been made in those values by the preprogram. If the value argument is set to FALSE (v=0), parameter values changed by the CVT program will be used in glyph instructions.



SCAN conversion ConTRoL

SCANCTRL[ ]

Code Range 0x85

Pops n: flags indicating when to turn on dropout control mode
(16 bit word padded to 32 bits)

Pushes –

Sets scan_control

SCANCTRL is used to set the value of the Graphics State variable scan_control which in turn determines whether the scan converter will activate dropout control for this glyph. Use of the dropout control mode is determined by three conditions:

1. Is the glyph rotated?
2. Is the glyph stretched?
3. Is the current setting for ppem less than a specified threshold?

The interpreter pops a word from the stack and looks at the lower 16 bits.

Bits 0-7 represent the threshold value for ppem. A value of FF in bits 0-7 means invoke dropout_control for all sizes. A value of 0 in bits 0-7 means never invoke dropout_control.

Bits 8-13 are used to turn on dropout_control in cases where the specified conditions are met. Bits 8, 9 and 10 are used to turn on the dropout_control mode (assuming other conditions do not block it). Bits 11, 12, and 13 are used to turn off the dropout mode unless other conditions force it. Bits 14 and 15 are reserved for future use.

Bit	Meaning if set
8	Set dropout_control to TRUE if other conditions do not block and ppem is less than or equal to the threshold value.
9	Set dropout_control to TRUE if other conditions do not block and the glyph is rotated.
10	Set dropout_control to TRUE if other conditions do not block and the glyph is stretched.
11	Set dropout_control to FALSE unless ppem is less than or equal to the threshold value.
12	Set dropout_control to FALSE unless the glyph is rotated.
13	Set dropout_control to FALSE unless the glyph is stretched.
14	Reserved for future use.
15	Reserved for future use.

For example

0x0000	No dropout control is invoked
0x01FF	Always do dropout control
0x0A10	Do dropout control if the glyph is rotated and has less than 16 pixels per-em

The scan converter can operate in either a “normal” mode or in a “fix dropout” mode depending on the value of a set of enabling and disabling flags.

SCANTYPE

SCANTYPE[ ]

Code Range 0x8D

Pops n: 16 bit integer

Pushes –

Sets scan_control

Pops a 16-bit integer whose value is used to determine which rules the scan converter will use. If the value of the argument is 0, the fast scan converter will be used. If the value of the integer is 1 or 2, simple dropout control will be used. If the value of the integer is 4 or 5, smart dropout control will be used. More specifically,

if n=0 rules 1, 2, and 3 are invoked (simple dropout control scan conversion including stubs)

if n=1 rules 1, 2, and 4 are invoked (simple dropout control scan conversion excluding stubs)

if n=2 rules 1 and 2 only are invoked (fast scan conversion; dropout control turned off)

if n=3 same as n = 2

if n = 4 rules 1, 2, and 5 are invoked (smart dropout control scan conversion including stubs)

if n = 5 rules 1, 2, and 6 are invoked (smart dropout control scan conversion excluding stubs)

if n = 6 same as n = 2

if n = 7 same as n = 2

The scan conversion rules are shown here:

Rule 1 If a pixel's center falls within the glyph outline, that pixel is turned on.

Rule 2 If a contour falls exactly on a pixel's center, that pixel is turned on.

Rule 3 If a scan line between two adjacent pixel centers (either vertical or horizontal) is intersected by both an on-Transition contour and an off-Transition contour and neither of the pixels was already turned on by rules 1 and 2, turn on the left-most pixel (horizontal scan line) or the bottom-most pixel (vertical scan line). This is "Simple" dropout control.

Rule 4 Apply Rule 3 only if the two contours continue to intersect other scan lines in both directions. That is, do not turn on pixels for 'stubs.' The scanline segments that form a square with the intersected scan line segment are examined to verify that they are intersected by two contours. It is possible that these could be different contours than the ones intersecting the dropout scan line segment. This is very unlikely but may have to be controlled with grid-fitting in some exotic glyphs.

Rule 5 If a scan line between two adjacent pixel centers (either vertical or horizontal) is intersected by both an on-Transition contour and an off-Transition contour and neither of the pixels was already turned on by rules 1 and 2, turn on the pixel which is closer to the midpoint between the on-Transition contour and off-Transition contour. This is "Smart" dropout control.

Rule 6 Apply Rule 5 only if the two contours continue to intersect other scan lines in both directions. That is, do not turn on pixels for 'stubs.'

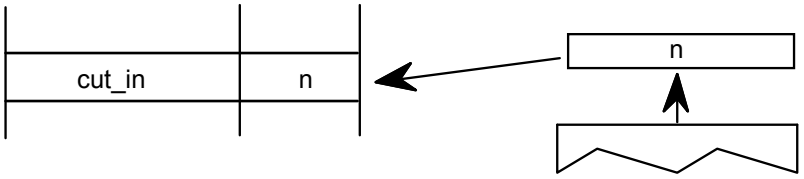
New fonts wishing to use the new modes of the ScanType instruction, but still wishing to work correctly on old rasterizers that don't recognize the new modes should:

1. First execute a ScanType instruction using an old mode which will give the best approximation to the desired new mode (e.g. Simple Stubs for Smart Stubs), and then
2. Immediately execute another ScanType instruction with the desired new mode.

Set Control Value Table Cut In

SCVTCI[ ]	
Code Range	0x1D
Pops	n: value for cut_in (F26Dot6)
Pushes	–
Sets	control_value_cut_in
Affects	MIAP, MIRP

Sets the control_value_cut_in in the Graphics State. The value n is expressed in sixty-fourths of a pixel.



Increasing the value of the cut_in will increase the range of sizes for which CVT values will be used instead of the original outline value.

Set Single_Width_Cut_In

SSWCI[ ]

Code Range 0x1E

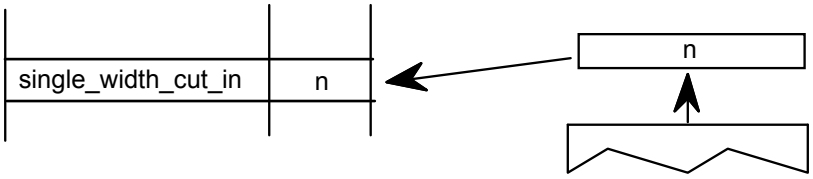
Pops n: value for single_width_cut_in (F26dot6)

Pushes –

Sets single_width_cut_in

Affects MIAP, MIRP

Sets the single_width_cut_in in the Graphics State. The value n is expressed in sixty-fourths of a pixel.



Set Single-width

SSW[ ]

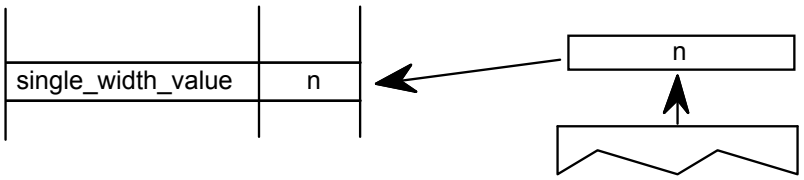
Code Range 0x1F

Pops n: value for single_width_value (FUnits)

Pushes –

Sets single_width_value

Sets the single_width_value in the Graphics State. The single_width_value is expressed in FUnits.



Set the auto_flip Boolean to ON

FLIPON[ ]

Code Range 0x4D

Pops –

Pushes –

Sets auto_flip

Affects MIRP

Sets the auto_flip Boolean in the Graphics State to TRUE causing the MIRP instructions to ignore the sign of Control Value Table entries. The default auto_flip Boolean value is TRUE.

auto_flip	1

Set the auto_flip Boolean to OFF

FLIPOFF[ ]

Code Range 0x4E

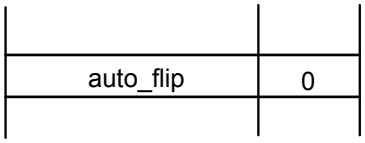
Pops –

Pushes –

Sets auto_flip

Affects MIRP

Set the auto_flip Boolean in the Graphics State to FALSE causing the MIRP instructions to use the sign of Control Value Table entries. The default auto_flip Boolean value is TRUE.



Set Angle _Weight

SANGW[ ]

Code Range 0x7E

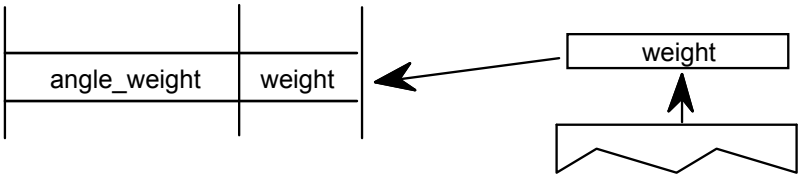
Pops weight: value for angle_weight

Pushes –

Sets angle_weight

SANGW is no longer needed because of dropped support to the AA (Adjust Angle) instruction. AA was the only instruction that used angle_weight in the global graphics state.

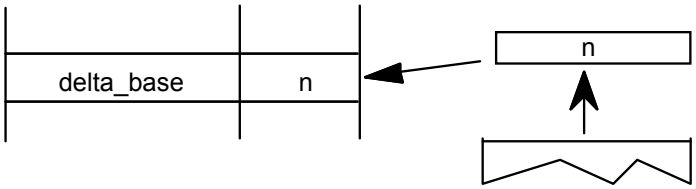
Pops a weight value from the stack and sets the value of the angle_weight state variable accordingly.



Set Delta_Base in the graphics state

SDB[ ]	
Code Range	0x5E
Pops	n: value for the delta_base (ULONG)
Pushes	–
Sets	delta_base
Affects	DELTAP1, DELTAP2, DELTAP3, DELTAC1, DELTAC2, DELTAC3

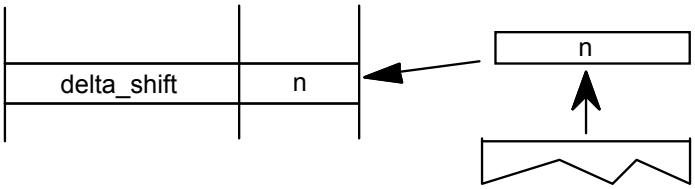
Pops a number, n, and sets delta_base to the value n. The default for delta_base is 9.



Set Delta_Shift in the graphics state

SDS[ ]	
Code Range	0x5F
Pops	n: value for the delta_shift (ULONG)
Pushes	–
Sets	delta_shift
Affects	DELTAP1, DELTAP2, DELTAP3, DELTAC1, DELTAC2, DELTAC3

Sets delta_shift to the value n. The default for delta_shift is 3.



Reading and writing data

The following instructions make it possible to get and to set a point coordinate, to measure the distance between two points, and to determine the current settings for pixels per em and point size.

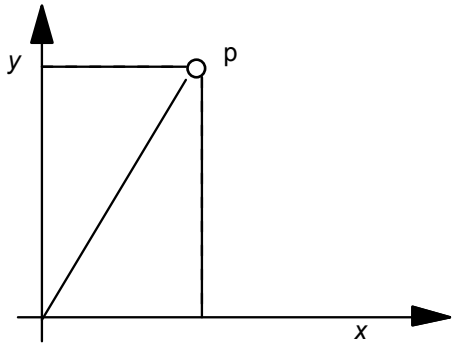
Get Coordinate projected onto the projection_vector

GC[a]	
Code Range	0x46 - 0x47
a	0: use current position of point p
	1: use the position of point p in the original outline
Pops	p: point number (ULONG)
Pushes	value: coordinate location (F26Dot6)
Uses	zp2, projection_vector

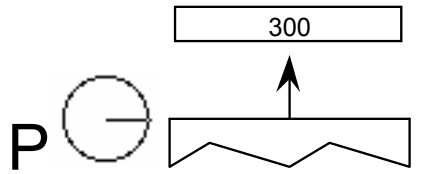
Measures the coordinate value of point p on the current projection_vector and pushes the value onto the stack.

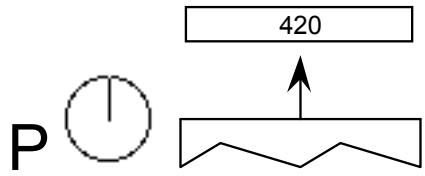
Example

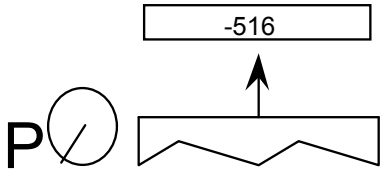
The following example shows that the value returned by GC is dependent upon the current position of the projection_vector. Note that point p is at the position (300,420) in the coordinate grid.



GC[1] 9





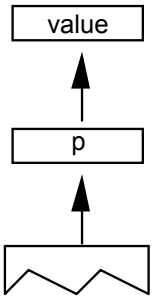


The projection_vector is parallel to the line from (0,0) to (300,420)

Sets Coordinate From the Stack using projection_vector and freedom_vector

SCFS[ ]	
Code Range	0x48
Pops	value: distance from origin to move point (F26Dot6)
	p: point number (ULONG)
Pushes	–
Uses	zp2, freedom_vector, projection_vector

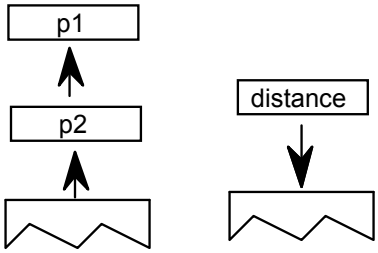
Moves point p from its current position along the freedom_vector so that its component along the projection_vector becomes the value popped off the stack.



Measure Distance

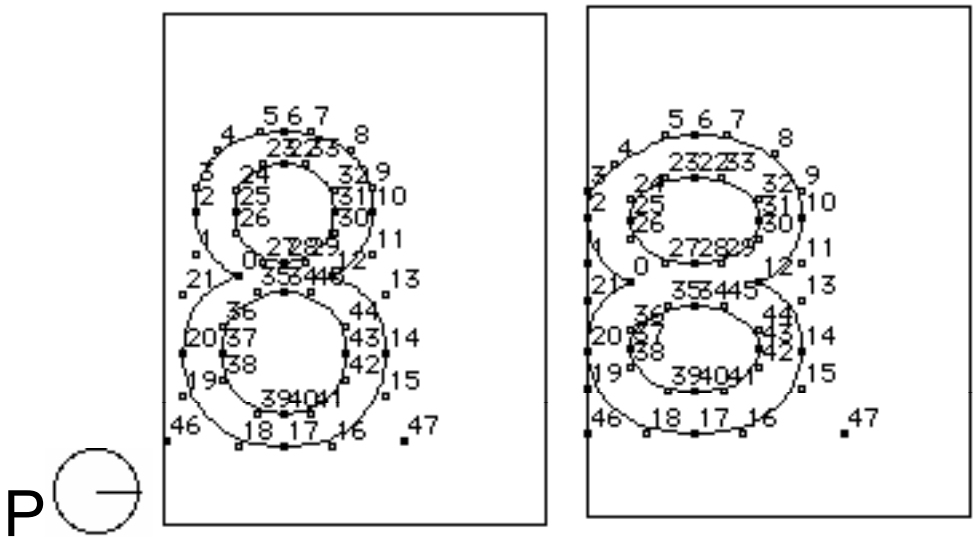
MD[a]	
Code Range	0x49 - 0x4A
a	0: measure distance in grid-fitted outline
	1: measure distance in original outline
Pops	p1: point number (ULONG) p2: point number (ULONG)
Pushes	distance (F26Dot6)
Uses	zp1 with point p1, zp0 with point p2, projection_vector

Measures the distance between outline point p1 and outline point p2. The value returned is in pixels (F26Dot6) If distance is negative, it was measured against the projection vector. Reversing the order in which the points are listed will change the sign of the result.



Example:

In the illustration below MD[1] between points 25 and 31 will return a smaller value than MD[0] at 10 pixels per em on a 72 dpi device. The difference is due to the effects of grid-fitting which, at this size, stretches out the counter.



Measure Pixels Per EM

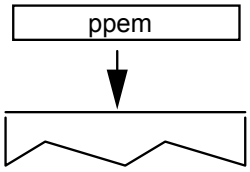
MPPEM[ ]

Code Range 0x4B

Pops –

Pushes ppem: pixels per em (ULONG)

This instruction pushes the number of pixels per em onto the stack. Pixels per em is a function of the resolution of the rendering device and the current point size and the current transformation matrix. This instruction looks at the projection_vector and returns the number of pixels per em in that direction.



Measure Point Size

MPS[ ]

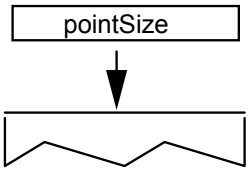
Code Range 0x4C

Pops –

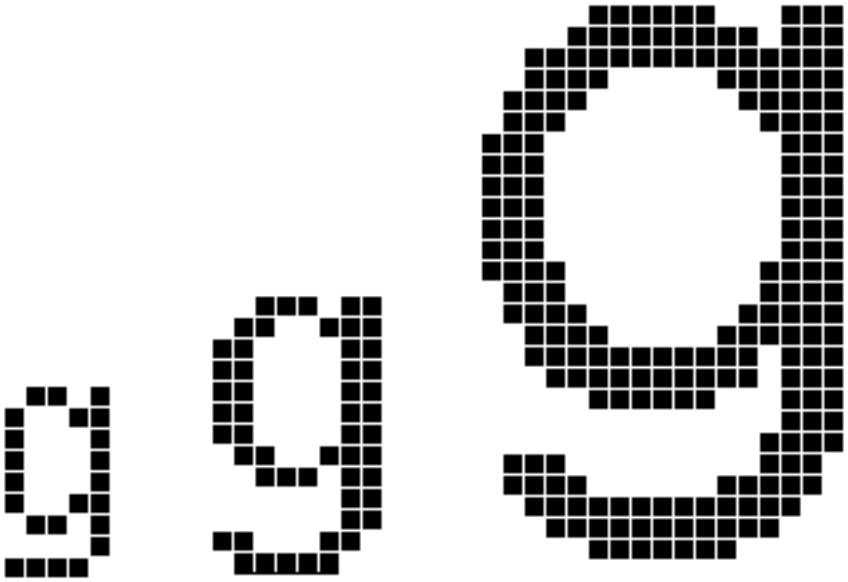
Pushes pointSize: the size in points of the current glyph (F26Dot6)

Pushes the current point size onto the stack.

Measure point size can be used to obtain a value which serves as the basis for choosing whether to branch to an alternative path through the instruction stream. It makes it possible to treat point sizes below or above a certain threshold differently.



12, 18, and 36 point Helvetica g at 72 dpi



Blank

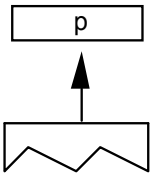
Managing outlines

The following set of instructions make it possible to move the points that make up a glyph outline. They are the instructions that accomplish the actual work of grid-fitting. They include instructions to move points, shift points or groups of points, flip points from off to on the curve or vice versa, and to interpolate points.

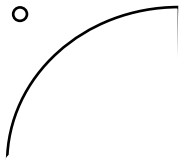
FLIP PoinT

FLIPPT[ ]	
Code Range	0x80
Pops	p: point number (ULONG)
Pushes	–
Uses	loop, p is referenced in zp0

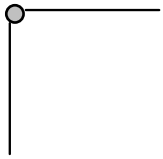
Flips points that are *off* the curve so that they are *on* the curve and points that are *on* the curve so that they are *off* the curve. The point is not marked as touched. The result of a FLIPPT instruction is that the contour describing part of a glyph outline is redefined.



Before:



After



FLIP RanGe ON

FLIPRGON[ ]

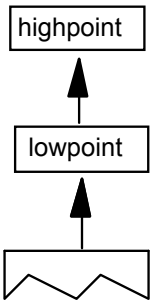
Code Range 0x81

Pops highpoint: highest point number in range of points to be flipped (ULONG)

lowpoint: lowest point number in range of points to be flipped (ULONG)

Pushes –

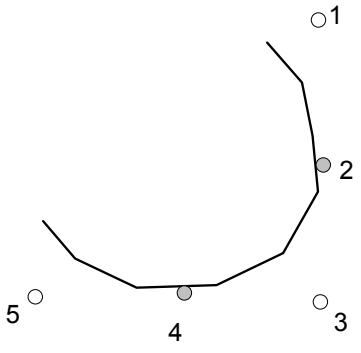
Flips a range of points beginning with lowpoint and ending with highpoint so that any off the curve points become on the curve points. The points are not marked as touched.



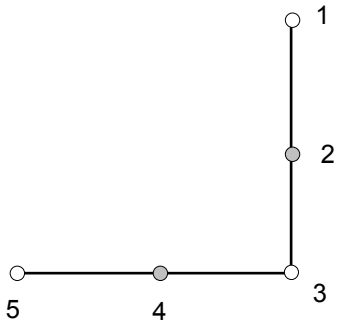
Example:

FLIPRGON[ ] 1 5

Before:



After:



Will make all off curve points between point 0 and point 5 into on curve points as shown

FLIP RanGe OFF

FLIPRGOFF[ ]

Code Range 0x82

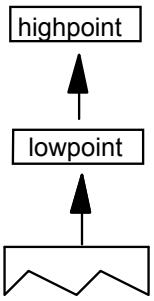
Pops highpoint: highest point number in range of points to be flipped (ULONG)

lowpoint: lowest point number in range of points to be flipped (ULONG)

Pushes –

Flips a range of points beginning with lowpoint and ending with highpoint so that any on curve points become off the curve points. The points are not marked as touched.

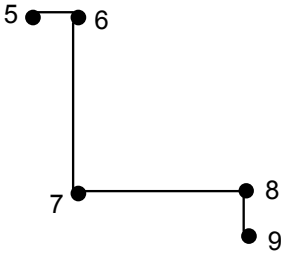
NOTE: This instruction changes the curve but the position of the points is unaffected. Accordingly, points affected by this instruction are not marked as touched.



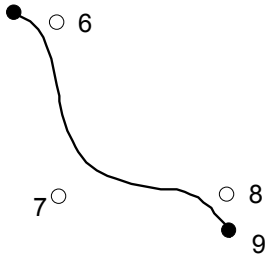
Example:

FLIPRGOFF[ ] 8 6

Before :



After:



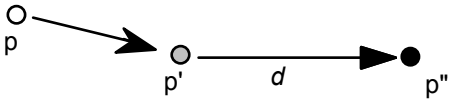
SHift Point by the last point

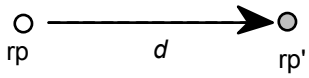
SHP[a]	
Code Range	0x32 - 0x33
a	0: uses rp2 in the zone pointed to by zp1
	1: uses rp1 in the zone pointed to by zp0
Pops	p: point to be shifted (ULONG)
Pushes	–
Uses	zp0 with rp1 or zp1 with rp2 depending on flag zp2 with point p loop, freedom_vector, projection_vector

Shift point p by the same amount that the reference point has been shifted. Point p is shifted along the freedom_vector so that the distance between the new position of point p and the current position of point p is the same as the distance between the current position of the reference point and the original position of the reference point.

NOTE: Point p is shifted from its current position, not its original position. The distance that the reference point has shifted is measured between its current position and the original position.

In the illustration below rp is the original position of the reference point, rp' is the current position of the reference point, p is the original position of point p, p' is the current position, p" the position after it is shifted by the SHP instruction. (White indicates original position, gray is current position, black is position to which this instruction moves a point).





SHift Contour by the last point

SHC[a]	
Code Range	0x34 - 0x35
a	0: uses rp2 in the zone pointed to by zp1
	1: uses rp1 in the zone pointed to by zp0
Pops	c: contour to be shifted (ULONG)
Pushes	–
Uses	zp0 with rp1 or zp1 with rp2 depending on flag zp2 with contour c freedom_vector, projection_vector

Shifts every point on contour c by the same amount that the reference point has been shifted. Each point is shifted along the freedom_vector so that the distance between the new position of the point and the old position of that point is the same as the distance between the current position of the reference point and the original position of the reference point. The distance is measured along the projection_vector. If the reference point is one of the points defining the contour, the reference point is not moved by this instruction.

This instruction is similar to SHP, but every point on the contour is shifted.

SHift Zone by the last pt

SHZ[a]	
Code Range	0x36 - 0x37
a	0: the reference point rp2 is in the zone pointed to by zp1
	1: the reference point rp1 is in the zone pointed to by zp0
Pops	e: zone to be shifted (ULONG)
Pushes	–
Uses	zp0 with rp1 or zp1 with rp2 depending on flag freedom_vector, projection_vector

Shift the points in the specified zone (Z1 or Z0) by the same amount that the reference point has been shifted. The points in the zone are shifted along the freedom_vector so that the distance between the new position of the shifted points and their old position is the same as the distance between the current position of the reference point and the original position of the reference point.

SHZ[a] uses zp0 with rp1 or zp1 with rp2. This instruction is similar to SHC, but all points in the zone are shifted, not just the points on a single contour.

SHift point by a PIXel amount

SHPIX[ ]	
Code Range	0x38
Pops	amount: magnitude of the shift (F26Dot6)
	p1, p2,...pn: points to be shifted (ULONG)
Pushes	–
Uses	zp2, loop, freedom_vector

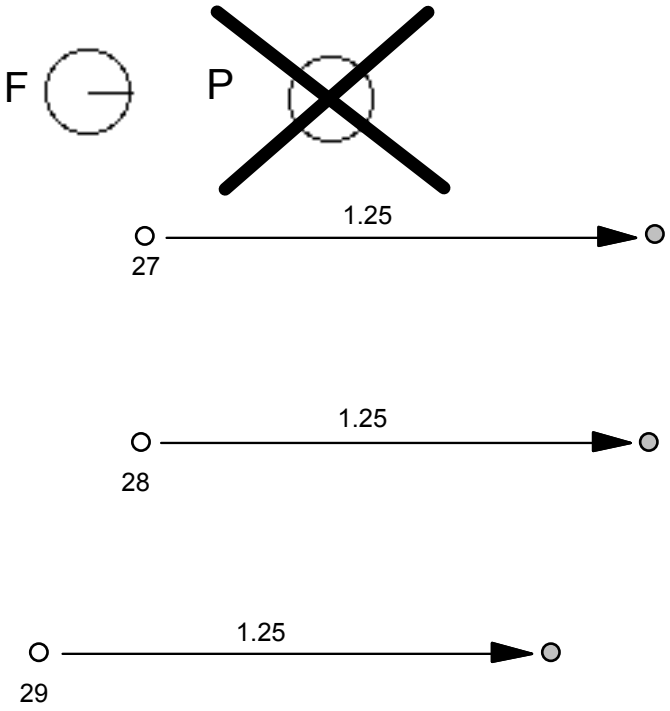
Shifts the points specified by the amount stated. When the loop variable is used, the amount to be shifted is put onto the stack only once. That is, if loop = 3, then the contents of the top of the stack should be point p1, point p2, point p3, amount. The value amount is expressed in sixty-fourths of a pixel.

SHPIX is unique in relying solely on the direction of the freedom_vector. It makes no use of the projection_vector. Measurement is made in the direction of the freedom_vector.

Example

The instruction shifts points 27, 28, and 29 by 80/64 or 1.25 pixels in the direction of the freedom vector. The distance is measured in the direction of the freedom_vector; the projection vector is ignored.

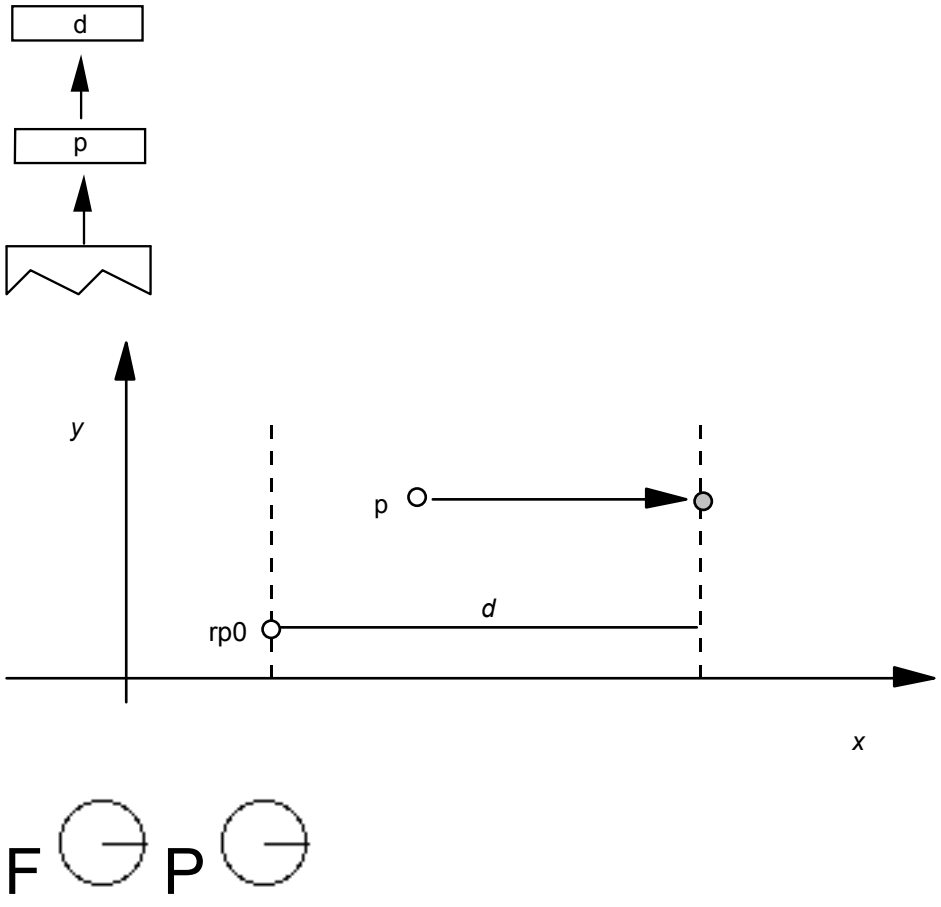
SHPIX[]



Move Stack Indirect Relative Point

MSIRP[a]	
Code Range	0x3A - 0x3B
a	0: Do not set rp0 to p
	1: Set rp0 to p
Pops	d: distance (F26Dot6)
	p: point number (ULONG)
Pushes	–
Uses	zp1 with point p and zp0 with rp0, freedom_vector, projection_vector.
Sets	After it has moved the point this instruction sets rp1 = rp0, rp2 = point p, and if a=1, rp0 is set to point p.

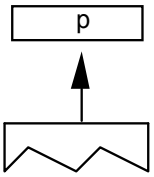
Makes the distance between a point p and rp0 equal to the value specified on the stack. The distance on the stack is in fractional pixels (F26Dot6). An MSIRP has the same effect as a MIRP instruction except that it takes its value from the stack rather than the Control Value Table. As a result, the cut_in does not affect the results of a MSIRP. Additionally, MSIRP is unaffected by the round_state.



Move Direct Absolute Point

MDAP[a]	
Code Range	0x2E - 0x2F
a:	0: do not round the value
	1: round the value
Pops	p: point number (ULONG)
Pushes	–
Sets	rp0 = rp1 = point p
Uses	zp0, round_state, projection_vector, freedom_vector.

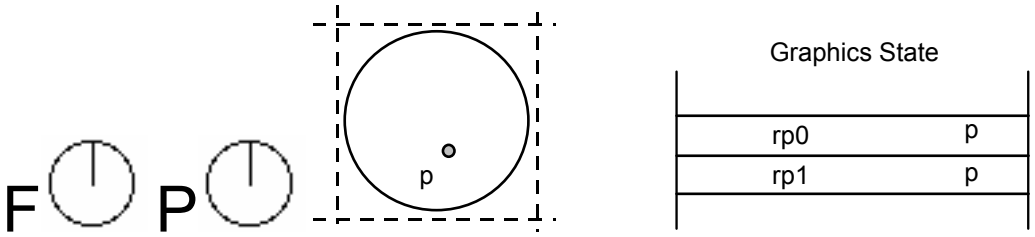
Sets the reference points rp0 and rp1 equal to point p. If a=1, this instruction rounds point p to the grid point specified by the state variable round_state. If a=0, it simply marks the point as touched in the direction(s) specified by the current freedom_vector. This command is often used to set points in the twilight zone.



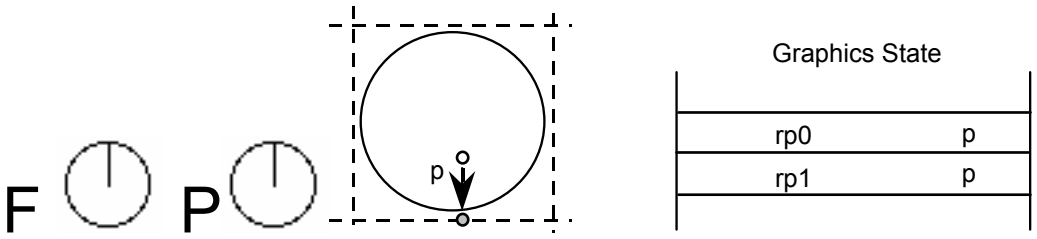
Example:

MDAP[0]

When a=0, the point is simply marked as touched and the values of rp0 and rp1 set to point p.



MDAP[1] assuming that the round_state is round to grid and the freedom_vector is a shown.



Move Indirect Absolute Point

MIAP[a]	
Code Range	0x3E - 0x3F
a	0: don't round the distance and don't look at the control_value_cut_in
	1: round the distance and look at the control_value_cut_in
Pops	n: CVT entry number (ULONG)
	p: point number (ULONG)
Pushes	–
Sets	rp0 = rp1 = point p
Uses	zp0, round_state, control_value_cut_in, freedom_vector, projection_vector

Moves point p to the absolute coordinate position specified by the *n*th Control Value Table entry. The coordinate is measured along the current projection_vector. If a=1, the position will be rounded as specified by round_state. If a=1, and if the device space difference between the CVT value and the original position is greater than the control_value_cut_in, then the original position will be rounded (instead of the CVT value.)

Rounding is done as if the entire coordinate system has been rotated to be consistent with the projection_vector. That is, if round_state is set to 1, and the projection_vector and freedom_vector are at a 45_ angle to the *x*-axis, then a MIAP[1] of a point to 2.9 pixels will round to 3.0 pixels along the projection_vector.

The a Boolean above controls both rounding and the use of the control_value_cut_in. If you would like the meaning of this Boolean to specify only whether or not the MIAP[ ] instruction should look at the control_value_cut_in value, use the ROFF[ ] instruction to turn off rounding.

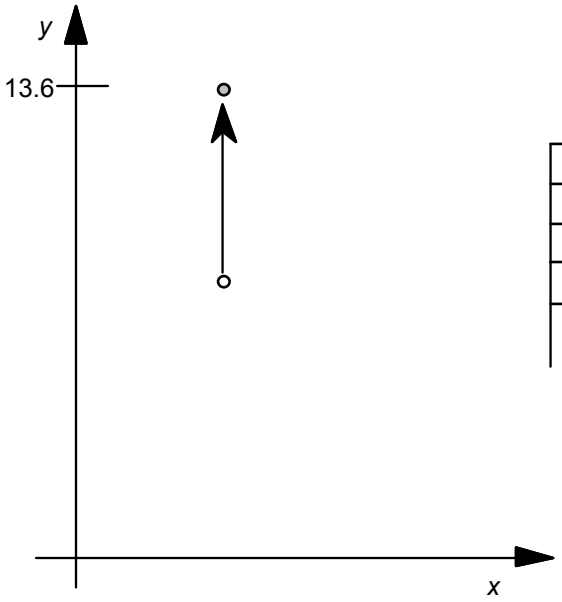
This instruction can be used to create Twilight Zone points.

Example:

MIAP[1] 4 7

case 1:
rounding is OFF

The point is moved to the position specified in the CVT.



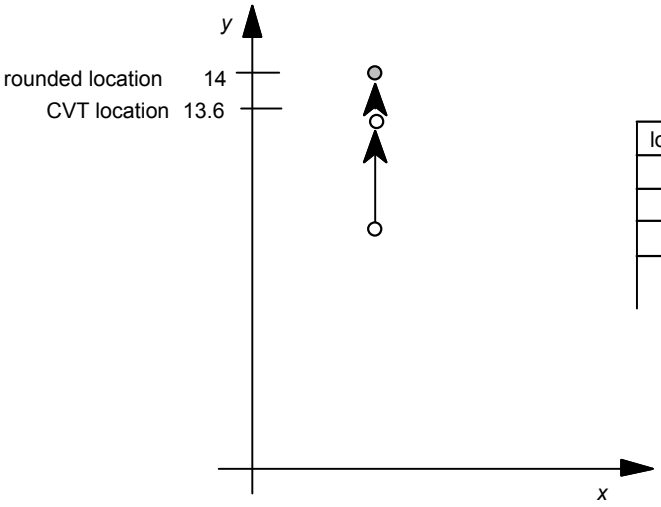
location	value
0	...
...	...
7	13.6

(continued...)

case 2:

The cut_in test succeeds and rounding is RTG.

The value in the CVT is subjected to the rounding rule and then the point is moved to the rounded position.

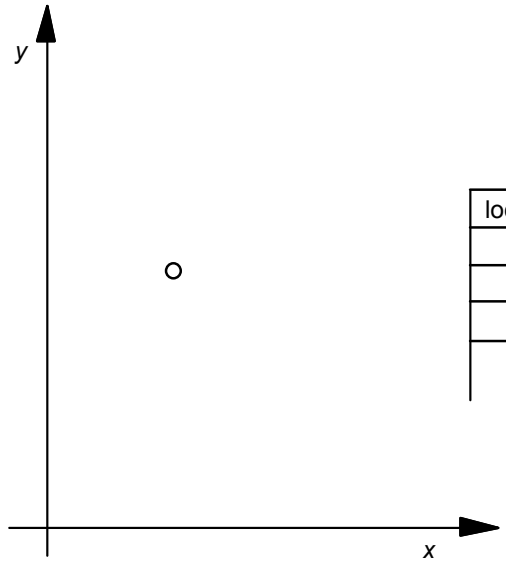


location	value
0	...
...	...
7	13.6

case 3:

The cut_in test fails and rounding is OFF.

Here the point is not moved.

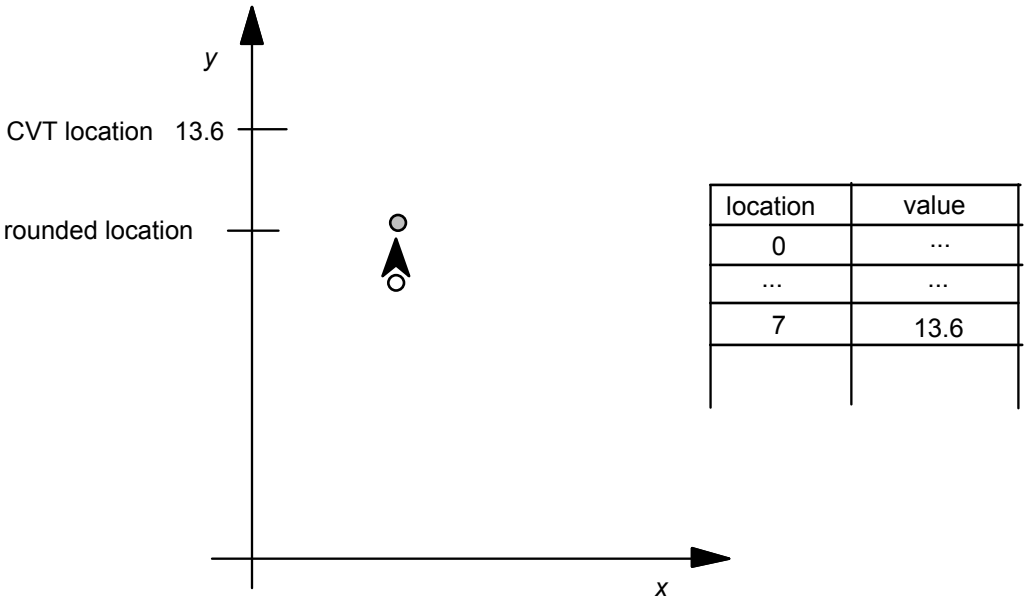


location	value
0	...
...	...
7	13.6

case 4:

The cut_in test fails and rounding is RTG.

In this case the point is moved to the nearest grid position.



location	value
0	...
...	...
7	13.6

Move Direct Relative Point

MDRP[abcde]

Code Range	0xC0 - 0xDF
a	0: do not set rp0 to point p after move
	1: do set rp0 to point p after move
b	0: do not keep distance greater than or equal to minimum_distance
	1: keep distance greater than or equal to minimum_distance
c	0: do not round distance
	1: round the distance
de	distance type for engine characteristic compensation
Pops	p: point number (ULONG)
Pushes	–
Sets	after point p is moved, rp1 is set equal to rp0, rp2 is set equal to point p; if the a flag is set to TRUE, rp0 is set equal to point p
Uses	zp0 with rp0 and zp1 with point p, round_state, single_width_value, single_width_cut_in, freedom_vector, projection_vector.

MDRP moves point p along the freedom_vector so that the distance from its new position to the current position of rp0 is the same as the distance between the two points in the original uninstructed outline, and then adjusts it to be consistent with the Boolean settings. Note that it is only the original positions of rp0 and point p and the current position of rp0 that determine the new position of point p along the freedom_vector.

MDRP is typically used to control the width or height of a glyph feature using a value which comes from the original outline. Since MDRP uses a direct measurement and does not reference the control_value_cut_in, it is used to control measurements that are unique to the glyph being instructed. Where there is a need to coordinate the control of a point with the treatment of points in other glyphs in the font, a MIRP instruction is needed.

Though MDRP does not refer to the CVT, its effect does depend upon the single-width cut-in value. If the device space distance between the measured value taken from the uninstructed outline and the single_width_value is less than the single_width_cut_in, the single_width_value will be used in preference to the outline distance. In other words, if the two distances are sufficiently close (differ by less than the single_width_cut_in), the single_width_value will be used.

The setting of the round_state Graphics State variable will determine whether and how the distance of point p from point q is rounded. If the round bit is not set, the value will be unrounded. If the round bit is set, the effect will depend upon the choice of rounding state. The value of the minimum distance variable is the smallest possible value the distance between two points can be rounded to.

Distances measured with the MDRP instruction must be either black, white or gray. Indicating this value in Booleans *de* allows the interpreter to compensate for engine characteristics as needed. The value *de* specifies the distance type as described in the chapter, “Instructing Glyphs.” Three values are possible: Gray=0, Black=1, White=2.

Example 1:

Graphics State Settings

Before MDRP

rp0 7
rp1 ?
rp2 ?

Case 1:

After MDRP[00001] 8

rp0 7
rp1 7
rp2 8

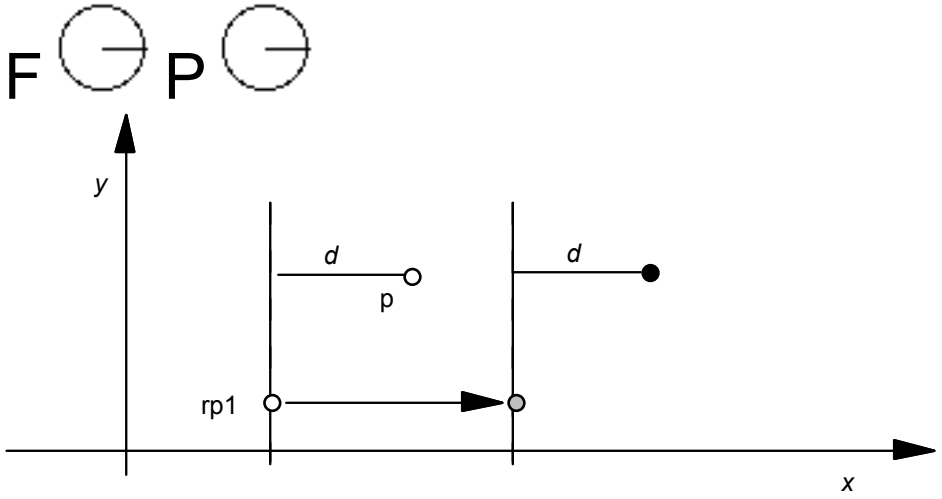
Case 2:

After MDRP[10001] 8

rp0 8
rp1 7
rp2 8

Example 2:

Point p is moved so that its distance from rp1 is the same as it was in the original outline.



- original position
- position before this instruction
- position after this instruction

Move Indirect Relative Point

MIRP[abcde]

Code Range	0xE0 - 0xFF
a	0: Do not set rp0 to p
	1: Set rp0 to p
b	0: Do not keep distance greater than or equal to minimum_distance
	1: Keep distance greater than or equal to minimum_distance
c	0: Do not round the distance and do not look at the control_value_cut_in
	1: Round the distance and look at the control_value_cut_in value
de:	distance type for engine characteristic compensation
Pops	n: CVT entry number (ULONG) p: point number (ULONG)
Pushes	–
Uses	zp0 with rp0 and zp1 with point p. round_state, control_value_cut_in, single_width_value, single_width_cut_in, freedom_vector, projection_vector
Sets	After it has moved the point this instruction sets rp1 = rp0, rp2 = point p, and if a = 1, rp0 is set to point p.

A MIRP instruction makes it possible to preserve the distance between two points subject to a number of qualifications. Depending upon the setting of Boolean flag b, the distance can be kept greater than or equal to the value established by the minimum_distance state variable. Similarly, the instruction can be set to round the distance according to the round_state graphics state variable. The value of the minimum distance variable is the smallest possible value the distance between two points can be rounded to. Additionally, if the c Boolean is set, the MIRP instruction acts subject to the control_value_cut_in. If the difference between the actual measurement and the value in the CVT is sufficiently small (less than the cut_in_value), the CVT value will be used and not the actual value. If the device space difference between this distance from the CVT and the single_width_value is smaller than the single_width_cut_in, then use the single_width_value rather than the outline or Control Value Table distance.

MIRP measures distance *relative* to point rp0. More formally, MIRP moves point p along the freedom_vector so that the distance from p to rp0 is equal to the distance stated in the reference CVT entry (assuming that the cut_in test succeeds)

The c Boolean above controls both rounding and the use of Control Value Table entries. If you would like the meaning of this Boolean to specify only whether or not the MIRP[ ] instruction should look at the control_value_cut_in, use the ROFF[ ] instruction to turn off rounding. In this manner, it is possible to specify rounding off and no cut_in.

The value de specifies the distance type as described in th chapter, “Instructing Glyphs.” Three values are possible: Gray=0, Black=1, White=2.

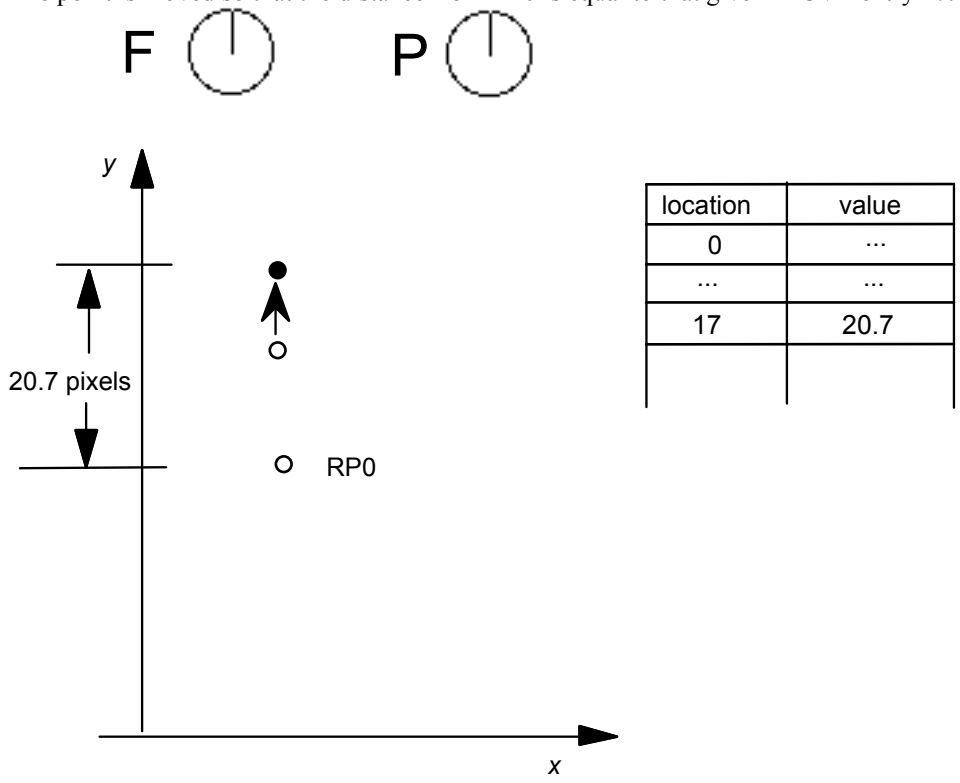
Example 1

MIRP[00110] 3 17

case 1:

The cut_in test succeeds and rounding is off.

The point is moved so that the distance from RP0 is equal to that given in CVT entry 17.

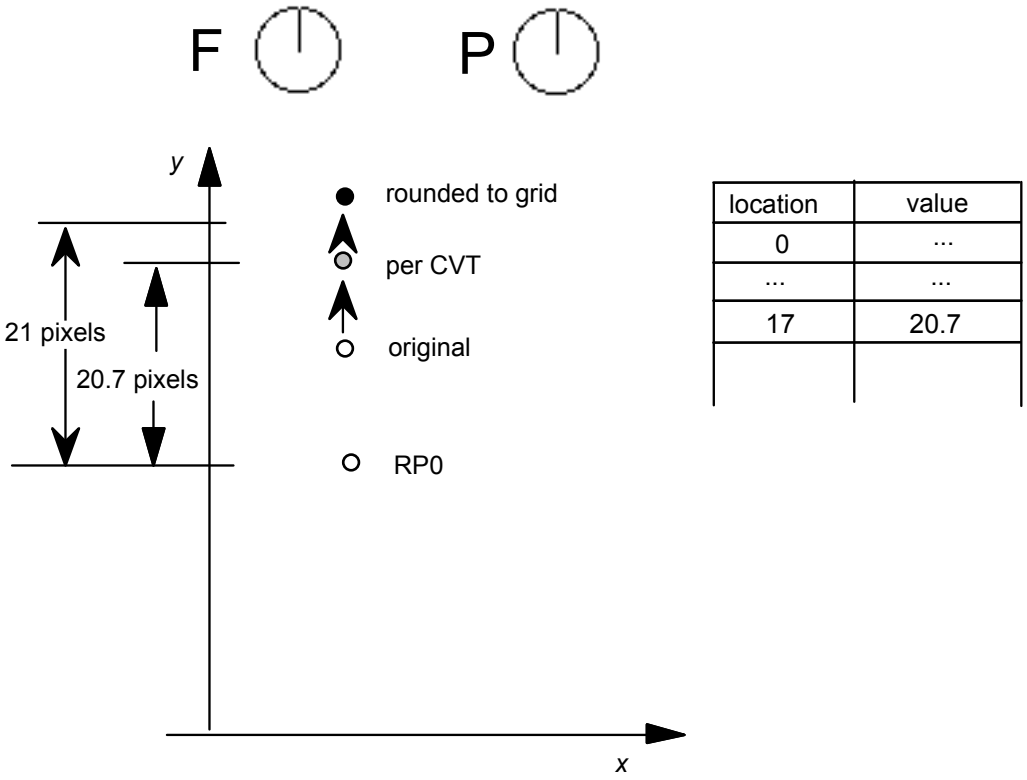


location	value
0	...
...	...
17	20.7

case 2:

The cut_in test succeeds and rounding is set to RTG.

The distance in the CVT is rounded and the point is moved by the rounded distance.

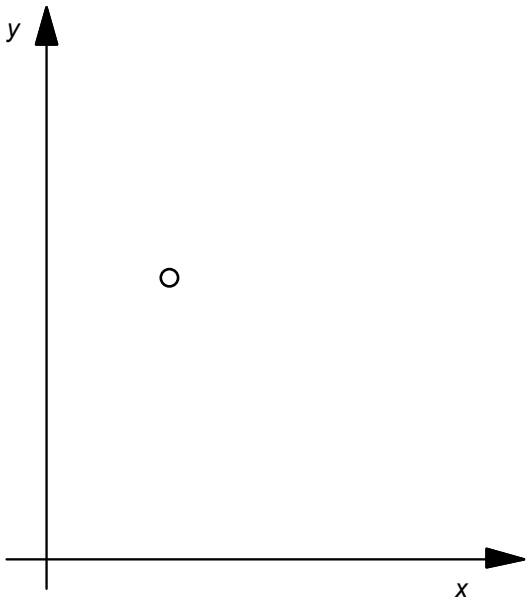


location	value
0	...
...	...
17	20.7

case 3:

The cut_in test fails and the round_state is OFF.

The point is not moved.

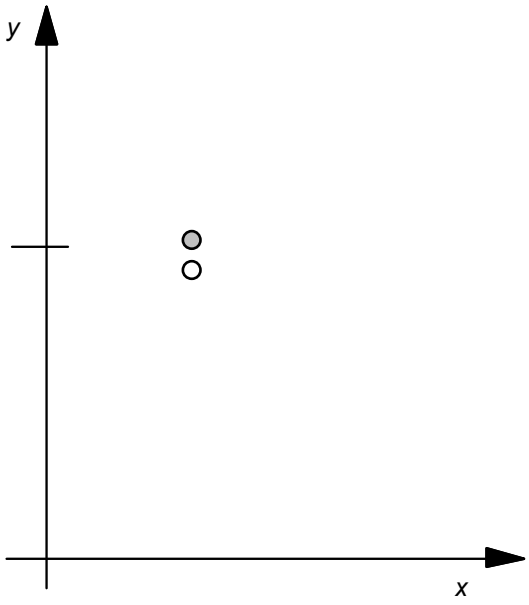


location	value
0	...
...	...
17	13.6

case 4:

The cut_in test fails and the round_state is RTG.

The current position of the point is rounded to the grid.

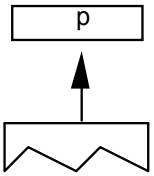


location	value
0	...
...	...
17	13.6

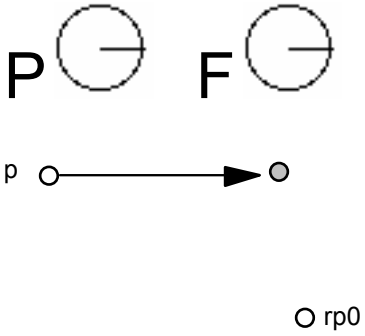
ALIGN Relative Point

ALIGNRP[ ]	
Code Range	0x3C
Pops	p: point number (ULONG)
Pushes	–
Uses	zp1 with point p, zp0 with rp0, loop, freedom_vector, projection_vector.

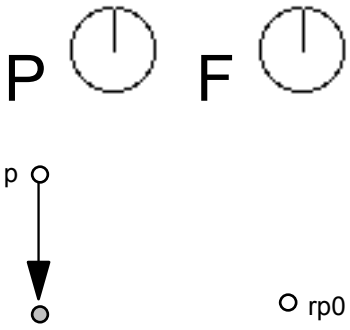
Reduces the distance between rp0 and point p to zero. Since distance is measured along the projection_vector and movement is along the freedom_vector, the effect of the instruction is to align points.



case 1:



case 2:

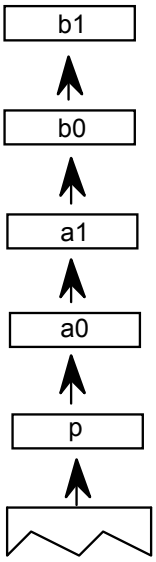


Adjust Angle (No Longer Supported)

moves point p to the InterSECTion of two lines

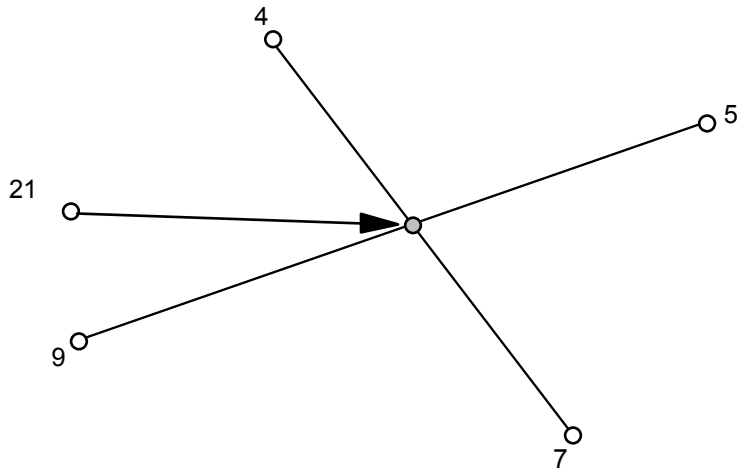
ISECT[ ]	
Code Range	0x0F
Pops	b1: end point of line 2 (ULONG) b0: start point of line 2 (ULONG) a1: end point of line 1 (ULONG) a0: start point of line 1 (ULONG) p: point to move (ULONG)
Pushes	–
Uses	zp2 with point p, zp1 with line A, zp0 with line B

Puts point p at the intersection of the lines A and B. The points a0 and a1 define line A. Similarly, b0 and b1 define line B. ISECT ignores the freedom_vector in moving point p.



Example:

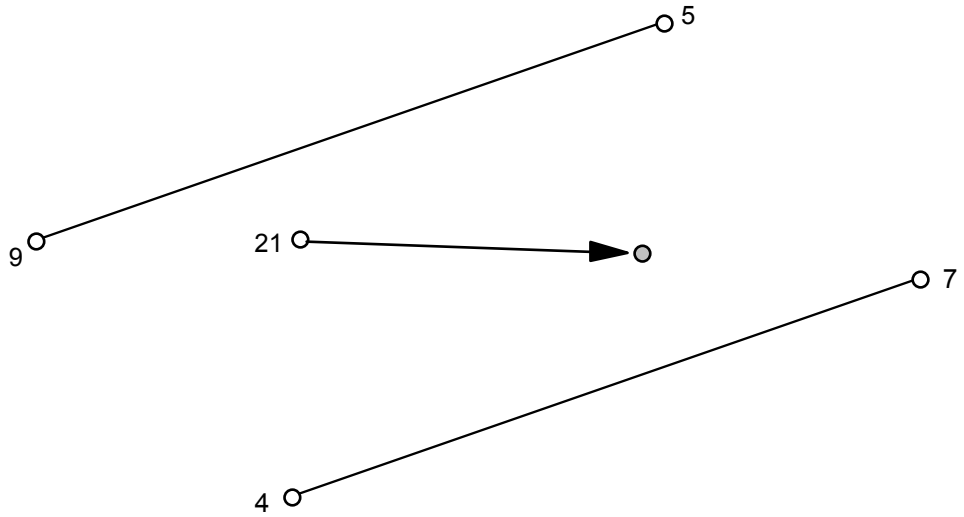
ISECT[ ] 21 9 5 4 7



NOTE: If lines are parallel to each other, the point is put into the middle of the two lines.

Example:

ISECT[ ] 21 9 5 4 7

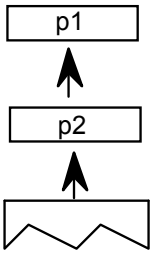


ALIGN Points

ALIGNPTS[ ]

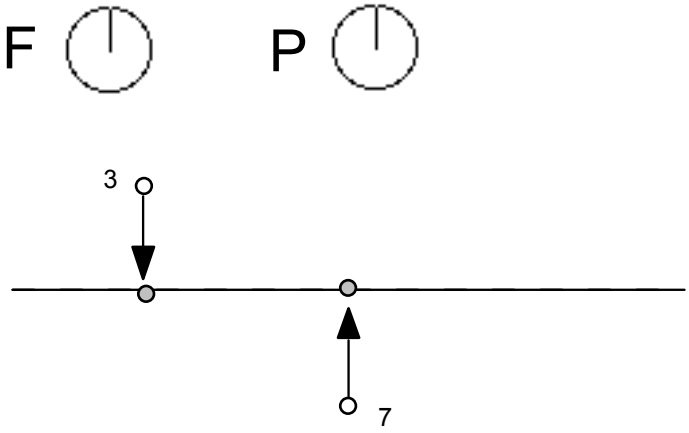
Code Range	0x27
Pops	p1: point number (ULONG) p2: point number (ULONG)
Pushes	–
Uses	zp1 with point p1, zp0 with point p2, freedom_vector, projection_vector.

Makes the distance between point 1 and point 2 zero by moving both along the freedom_vector to the average of both their projections along the projection_vector.



Example:

ALIGNPTS[] 3 7

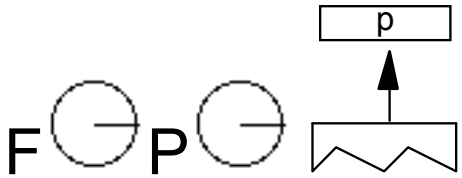


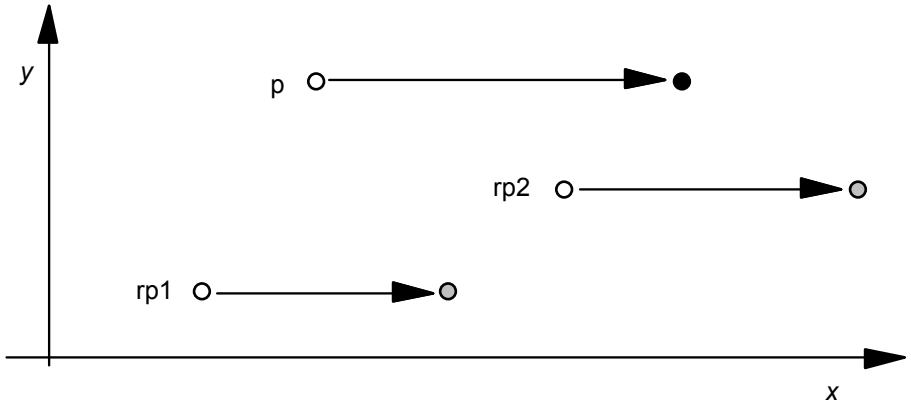
Interpolate Point by the last relative stretch

IP[ ]

Code Range	0x39
Pops	p: point number (ULONG)
Pushes	–
Uses	zp0 with rp1, zp1 with rp2, zp2 with point p, loop, freedom_vector, projection_vector

Moves point p so that its relationship to rp1 and rp2 is the same as it was in the original uninstructed outline. Measurements are made along the projection_vector, and movement to satisfy the interpolation relationship is constrained to be along the freedom_vector. This instruction is illegal if rp1 and rp2 have the same position on the projection_vector.





In the example shown, assume that the points referenced by rp1 and rp2 are moved as shown. An IP instruction is then used to preserve their relative relationship with point p. After the IP the following should be true

D(p, rp1)/D(p',rp1') = D(p,rp2)/D(p', rp2')

In other words, the relative distance is preserved.

UnTouch Point

UTP[ ]	
Code Range	0x29
Pops	p: point number (ULONG)
Pushes	–
Uses	zp0 with point p, freedom_vector

Marks point p as untouched. A point may be touched in the *x*-direction, the *y*-direction, both, or neither. This instruction uses the current freedom_vector to determine whether to untouch the point in the *x*-direction, the *y*-direction, or both. Points that are marked as untouched will be moved by an IUP (interpolate untouched points) instruction. Using UTP you can ensure that a point will be affected by IUP even if it was previously touched.

Interpolate Untouched Points through the outline

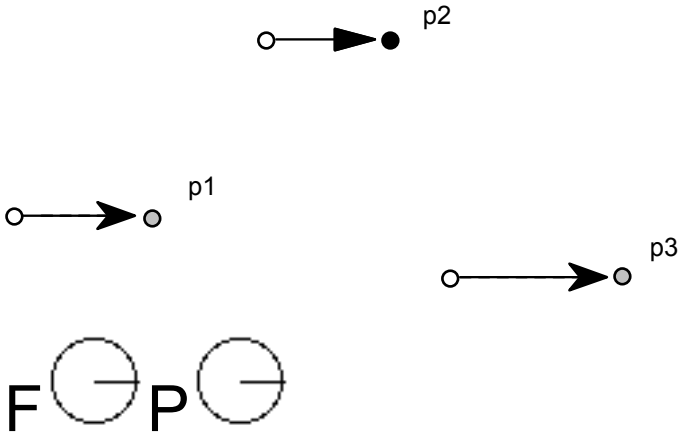
IUP[a]	
Code Range	0x30 - 0x31
a	0: interpolate in the *y*-direction
	1: interpolate in the *x*-direction
Pops	–
Pushes	–
Uses	zp2, freedom_vector, projection_vector

Considers a glyph contour by contour, moving any untouched points in each contour that are between a pair of touched points. If the coordinates of an untouched point were originally between those of the touched pair, it is linearly interpolated between the new coordinates, otherwise the untouched point is shifted by the amount the nearest touched point is shifted.

This instruction operates on points in the glyph zone pointed to by zp2. This zone should almost always be zone 1. Applying IUP to zone 0 is an error.

Consider three consecutive points all on the same contour. Two of the three points, p1 and p3 have been touched. Point p2 is untouched. The effect of an IUP in the x-direction is to move point p2 so that is in the same relative position to points p1 and p3 before they were moved.

The IUP instruction does not touch the points it moves. Thus the untouched points affected by an IUP instruction will be affected by subsequent IUP instructions unless they are touched by an intervening instruction. In this case, the first interpolation is ignored and the point is moved based on its original position.



Managing exceptions

DELTA instructions can be used to alter the outline of a glyph at a particular size. They are generally used to turn on or off specific pixels. Delta instructions work by moving points (DELTAP's) or by changing a value in the Control Value Table (DELTAC's).

More formally, the DELTA instructions take a variable number of arguments from the stack and allow the use of an exception of the form: at size *x* apply the movement d to point p (or at size *x* add or subtract an amount less than or equal to the Control Value Table entry c). DELTAs take a list of exceptions of the form: relative ppem value, the magnitude of the exception and the point number to which the exception is to be applied.

Each DELTA instruction works on a range of sizes as specified below. As a result, sizes are specified in relative pixels per em (ppem), that is relative to the delta_base. The default value for delta_base is 9 ppem. To set delta_base to another value, use the SDB instruction.

The DELTAP1 and DELTAC1 instructions allow values to be changed for glyphs at 9 through 24 ppem, assuming the default value for delta_base. Lowering the value for delta_base allows you to invoke exceptions at a smaller number of ppem.

DELTAP2 and DELTAC2 are triggered at 16 ppem higher than the value set for DELTAP1 and DELTAC1, and consequently the formula for the relative ppem is

ppem - 16 - delta_base.

DELTAP3 and DELTAC3 are triggered at 16 ppem higher than the value set for DELTAP2 and DELTAC2, or 32 ppem higher than the value set for DELTAP1 and DELTAC1, and consequently the formula for the relative ppem is:

ppem - 32 - delta_base.

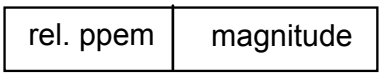
DELTA*1 delta_base through delta_base + 15 ppem

DELTA*2 delta_base + 16 ppem through delta_base + 31 ppem

DELTA*3 delta_base + 32 ppem through delta_base + 47 ppem

In specifying a DELTA instruction, the high 4 bits of arg1 describe the relative ppem value that will activate the exception.

The low 4 bits of arg1 describe the magnitude of the exception. The amount the point moves is a function of the exception stated and the Graphics State variable delta_shift. To set delta_shift, use the SDS instruction.



NOTE: Always observe that DELTA instructions expect the argument list to be sorted according to ppem. The lowest ppem should be deepest on the stack, and the highest ppem should be topmost on the stack.

In the descriptions of the instructions that follow, p_i is a point number, c_i is a Control Value Table entry number and arg_i is a byte composed of two parts: the relative ppem (ppem – delta_base) and the magnitude of the exception.

Increasing the delta_shift will allow for more fine control over pixel movement at the sacrifice of total range of movement. A step is the minimal amount that a delta instruction can move a point. Points can be moved in integral multiples of steps.

The size of a step is 1 divided by 2 to the power delta_shift. The range of movement produced by a given delta_shift can be calculated by taking the number of steps allowed (16) and dividing it by 2 to the power delta_shift. For example, a delta_shift equal to 2 allows the smallest movement to be ± 1/4 pixel (because 2^2 equals 4) and the largest movement to be ± 2 pixels (16/4 = 4 pixels of movement). A delta_shift of 5 allows the smallest movement to be ± 1/32 pixel (because 2^5 equals 32), but the largest movement is limited to ± 1/4 pixel. (16/32 = 1/2 a pixel of movement).

Internally, the value obtained for the exception is stored as a 4 bit binary number. As a result, the desired output range must be converted to a number between 0 and 15 before being converted to binary. Here is the internal remapping table for the DELTA instructions.

NOTE: that zero is lacking in the output range.

Number of Steps	➔ **Exception**
-8	0
-7	1
-6	2
-5	3
-4	4
-3	5
-2	6
-1	7
1	8
2	9
3	10
4	11
5	12
6	13
7	14
8	15

DELTA exception P1

DELTAP1[ ]		
Code Range	0x5D	
Pops	n: number of pairs of exception specifications and points (ULONG)	
	p_1, arg_1, p_2, arg_2, ..., p_n, arg_n:	n pairs of exception specifications and points (pairs of ULONGs)
Pushes	–	
Uses	zp0, delta_base, delta_shift	

DELTAP1 moves the specified points at the size and by the amount specified in the paired argument. An arbitrary number of points and arguments can be specified.

The grouping [p_i, arg_i] can be executed n times. The value of arg_i may vary between iterations.

DELTA exception P2

DELTAP2[ ]		
Code Range	0x71	
Pops	n: number of pairs of exception specifications and points (ULONG)	
	p_1, arg_1, p_2, arg_2, ..., p_n, arg_n:	n pairs of exception specifications and points (pairs of ULONGs)
Pushes	–	
Uses	zp0, delta_shift, delta_base	

DELTAP2 moves the specified points at the size and by the amount specified in the paired argument. An arbitrary number of points and arguments can be specified.

The grouping [p_i, arg_i] can be executed n times. The value of arg_i may vary between iterations.

DELTA exception P3

DELTAP3[ ]		
Code Range	0x72	
Pops	n: number of pairs of exception specifications and points (ULONG)	
	p_1, arg_1, p_2, arg_2, ..., p_n, arg_n:	n pairs of exception specifications and points (pairs of ULONGs)
Pushes	–	
Uses	zp0, delta_base, delta_shift	

DELTAP3 moves the specified points at the size and by the amount specified in the paired argument. An arbitrary number of point and arguments can be specified.

The grouping [p_i, arg_i] can be executed n times. The value of arg_i may vary between iterations.

DELTA exception C1

DELTAC1[ ]

Code Range 0x73

Pops n: number of pairs of exception specifications and CVT entry numbers (ULONG)

c_1, arg_1, c_2, arg_2,..., c_n, arg_n: (pairs of ULONGs)

Pushes –

DELTAC1 changes the value in each CVT entry specified at the size and by the amount specified in its paired argument.

The grouping [c_i, arg_i] can be executed n times. The value of arg_i may vary between iterations.

DELTA exception C2

DELTAC2[ ]

Code Range 0x74

Pops n: number of pairs of exception specifications and CVT entry numbers (ULONG)

c_1, arg_1, c_2, arg_2,..., c_n, arg_n: (pairs of ULONGs)

Pushes –

DELTAC2 changes the value in each CVT entry specified at the size and by the amount specified in its paired argument.

The grouping [c_i, arg_i] can be executed n times. The value of arg_i may vary between iterations.

DELTA exception C3

DELTAC3[ ]

Code Range 0x75

Pops n: number of pairs of CVT entry numbers and exception specifications (ULONG)

c_1, arg_1, c_2 arg_2,..., c_n, arg_n: pairs of CVT entry number and exception specifications (pairs of ULONGs)

Pushes –

DELTAC3 changes the value in each CVT entry specified at the size and by the amount specified in its paired argument.

The grouping [c_i, arg_i] can be executed n times. The value of arg_i may vary between iterations.

Example of DELTA exceptions

Assume that you want to move point 15 of your glyph 1/8 of a pixel along the freedom_vector at 12 pixels per em. Assume that delta_base has the default value 9 and delta_shift the default value 3.

To specify that the exception should be made at 12 ppem, you subtract the delta_base, which is 9, from 12 and store the result, which is 3, in the high nibble of arg_1.

To specify that the point is to be moved 1/8 of a pixel, multiply 1/8 by 2 raised to the power delta_shift. In other words, you multiply 1/8 by 2 raised to the third power (or 8)yielding 1. This value must be mapped to an internal value which using the table shown is 8.

Putting these two results together yields a 3 in the high nibble and an 8 in the low nibble or 56 (00111000, in binary).

To obtain this single exception, the top of the stack is: 56, 15, 1.

(iteration)
(point number)
(arg1: ppem and magnitude)

Now if the interpreter executes

DELTAP1[ ]

then this instruction will move point 15 of the glyph (at 12 ppem) 1/8 of a pixel along the freedom_vector.

Managing the stack

The following set of instructions make it possible to manage elements on the stack. They make it possible to duplicate the element at the top of the stack, remove the top element from the stack, clear the stack, swap the top two stack elements, determine the number of elements currently on the stack, copy a specified element to the top of the stack, move a specified element to the top of the stack, and rearrange the order of the top three elements on the stack.

Duplicate top stack element

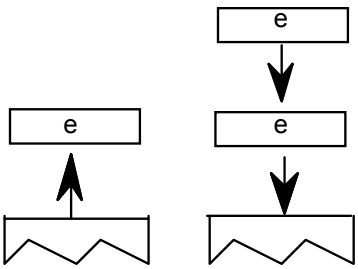
DUP[ ]

Code Range 0x20

Pops e: stack element (ULONG)

Pushes e, e (two ULONGs)

Duplicates the element at the top of the stack.



POP top stack element

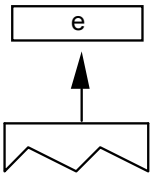
POP[ ]

Code Range 0x21

Pops e: stack element (ULONG)

Pushes –

Pops the top element of the stack.



Clear the entire stack

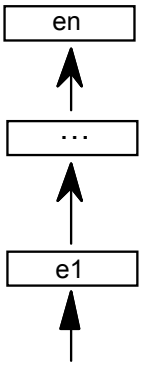
CLEAR[ ]

Code Range 0x22

Pops all the items on the stack (ULONGs)

Pushes –

Clears all elements from the stack.



SWAP the top two elements on the stack

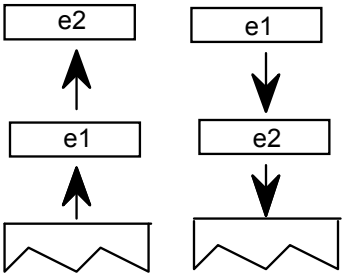
SWAP[ ]

Code Range 0x23

Pops e2: stack element (ULONG)
e1: stack element (ULONG)

Pushes e1, e2 (pair of ULONGs)

Swaps the top two elements of the stack making the old top element the second from the top and the old second element the top element.



Returns the DEPTH of the stack

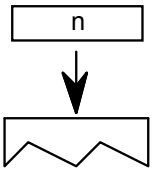
DEPTH[ ]

Code Range 0x24

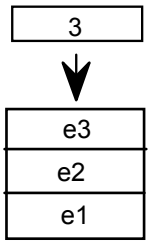
Pops –

Pushes n: number of elements (ULONG)

Pushes n, the number of elements currently in the stack onto the stack.



Example:



Copy the INDEXed element to the top of the stack

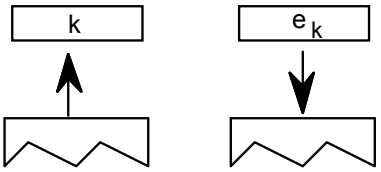
CINDEX[ ]

Code Range 0x25

Pops k : stack element number

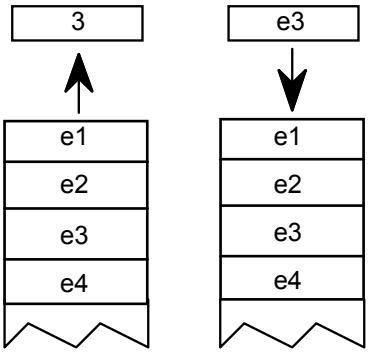
Pushes e_k: indexed element (ULONG)

Puts a copy of the kth stack element on the top of the stack.



Example:

CINDEX[ ]



Move the INDEXed element to the top of the stack

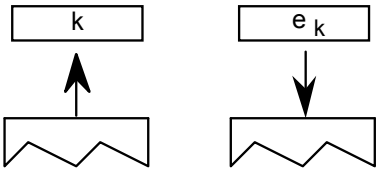
MINDEX[ ]

Code Range 0x26

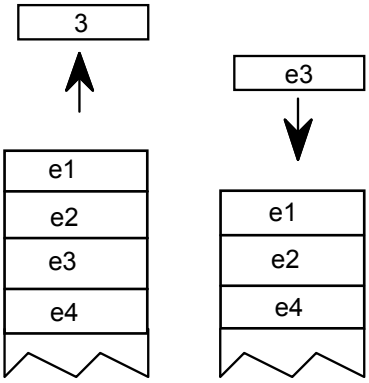
Pops k: stack element number

Pushes $e_{k:}$ indexed element

Moves the indexed element to the top of the stack.



MINDEX[ ]



ROLL the top three stack elements

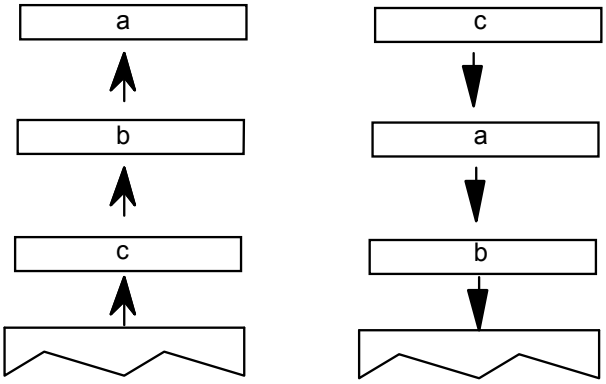
ROLL[ ]

Code Range 0x8a

Pops a, b, c (top three stack elements)

Pushes b, a, c (elements reordered)

Performs a circular shift of the top three objects on the stack with the effect being to move the third element to the top of the stack and to move the first two elements down one position. ROLL is equivalent to MINDEX[ ] 3.



Managing the flow of control

This section describes those instructions that make it possible to alter the sequence in which items in the instruction stream are executed. The IF and JMP instructions and their variants work by testing the value of an element on the stack and changing the value of the instruction pointer accordingly.

IF test

IF[ ]

Code Range 0x58

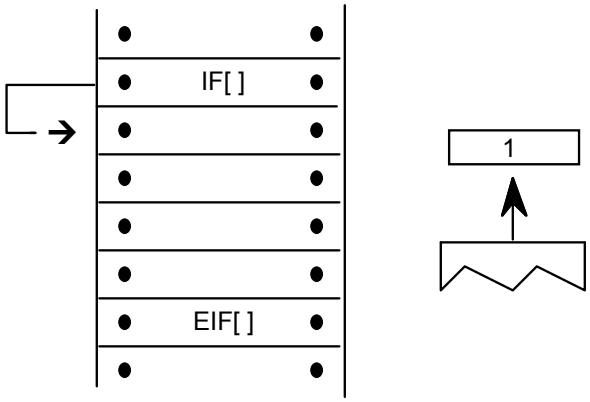
Pops e: stack element (ULONG)

Pushes –

Tests the element popped off the stack: if it is zero (FALSE), the instruction pointer is jumped to the next ELSE or EIF instruction in the instruction stream. If the element at the top of the stack is nonzero (TRUE), the next instruction in the instruction stream is executed. Execution continues until an ELSE instruction is encountered or an EIF instruction ends the IF. If an else statement is found before the EIF, the instruction pointer is moved to the EIF statement.

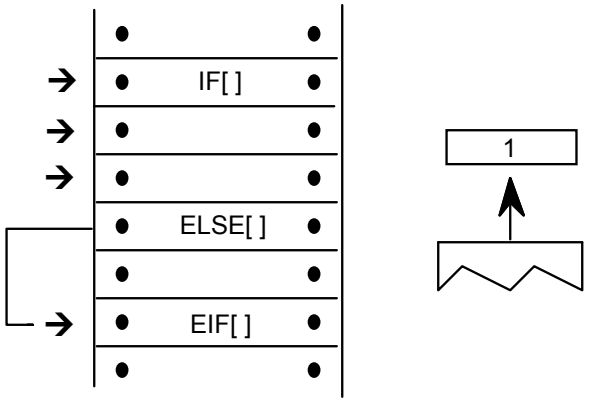
case 1:

Element at top of stack is TRUE; instruction pointer is unaffected. IF terminates with EIF.



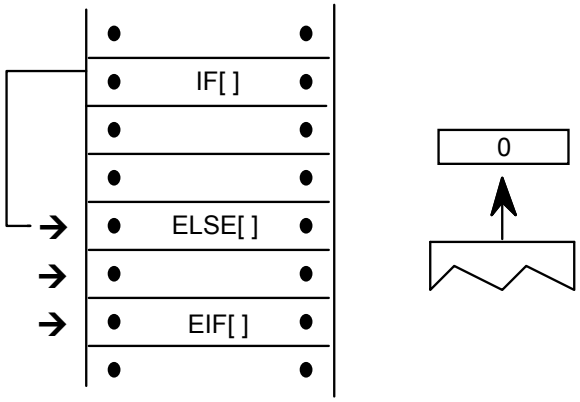
case 2:

Element at top of stack is TRUE. The instruction stream is sequentially executed until ELSE is encountered whereupon the instruction pointer jumps to the EIF statement that terminates the IF.



case 3:

Element at the top of the stack is FALSE; instruction pointer is moved to the ELSE statement; instructions are then executed sequentially; EIF ends the IF statement.



ELSE

ELSE[ ]

Code Range 0x1B

Pops –

Pushes –

Marks the start of the sequence of instructions that are to be executed if an IF instruction encounters a FALSE value on the stack. This sequence of instructions is terminated with an EIF instruction.

End IF

EIF[ ]

Code Range 0x59

Pops –

Pushes –

Marks the end of an IF[ ] instruction.

Jump Relative On True

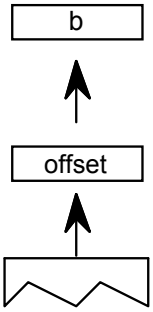
JROT[ ]

Code Range 0x78

Pops b: Boolean (ULONG)
offset: number of bytes to move instruction pointer (LONG)

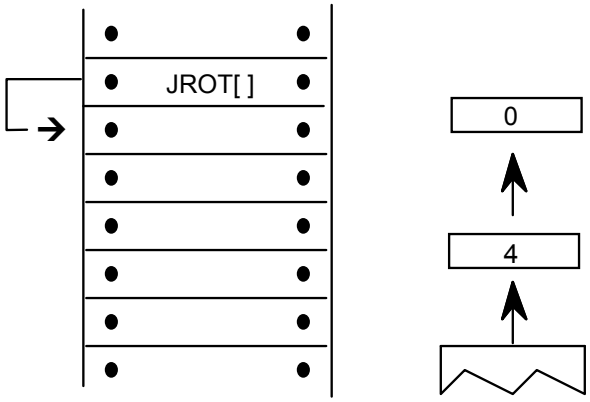
Pushes –

Obtains an offset and tests a Boolean value. If the Boolean is TRUE, the signed offset will be added to the instruction pointer and execution will be resumed at the address obtained. Otherwise, the jump is not taken. The jump is relative to the position of the instruction itself. That is, the instruction pointer is still pointing at the JROT[ ] instruction when offset is added to obtain the new address.



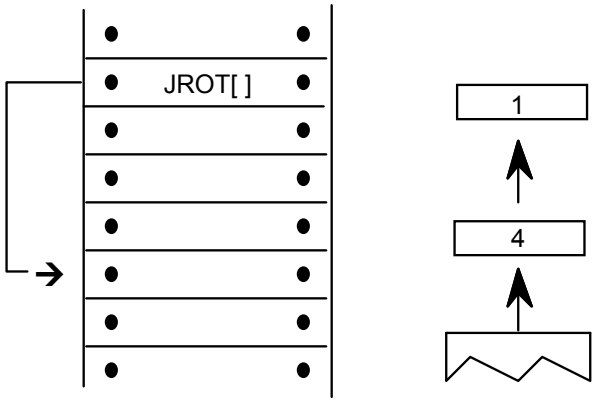
Example:

case 1:
Boolean is FALSE.



case 2:

Boolean is TRUE.



JuMP

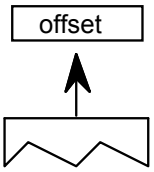
JMPR[ ]

Code Range 0x1C

Pops offset: number of bytes to move instruction pointer (LONG)

Pushes –

The signed offset is added to the instruction pointer and execution is resumed at the new location in the instruction steam. The jump is relative to the position of the instruction itself. That is, the instruction pointer is still pointing at the JROT[ ] instruction when offset is added to obtain the new address.



Jump Relative On False

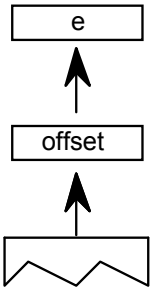
JROF[ ]

Code Range 0x79

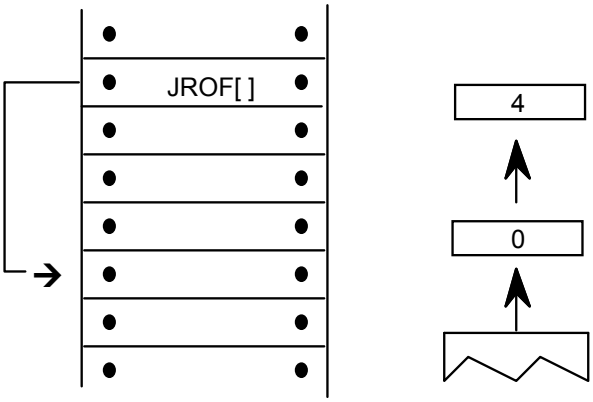
Pops e: stack element (ULONG)
offset: number of bytes to move instruction pointer (LONG)

Pushes –

In the case where the Boolean is FALSE, the signed offset will be added to the instruction pointer and execution will be resumed there; otherwise, the jump is not taken. The jump is relative to the position of the instruction itself. That is, the instruction pointer is still pointing at the JROT[ ] instruction when offset is added to obtain the new address.

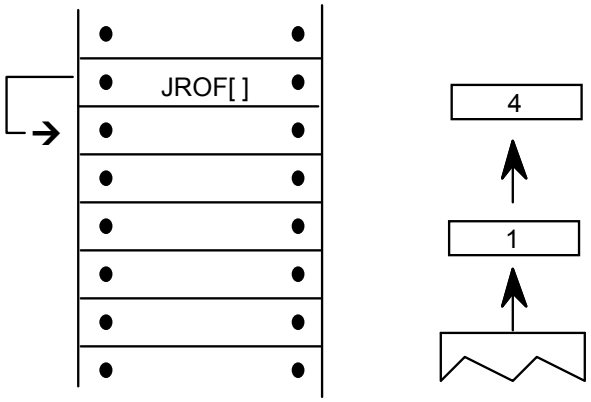


case 1:
element is FALSE.



(continued...)

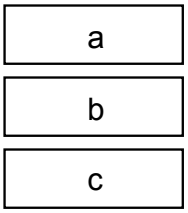
case 2:
element is TRUE.



Logical functions

The TrueType instruction set includes a set of logical functions that can be used to test the value of a stack element or to compare the values of two stack elements. The logical functions compare 32 bit values (ULONG) and return a Boolean value to the top of the stack.

To easily remember the order in which stack values are handled during logical operations, imagine writing the stack values from left to right, starting with the bottom value. Then insert the operator between the two furthest right elements. For example:



GT a,b would be interpreted as (b>a):

c b > a

Less Than

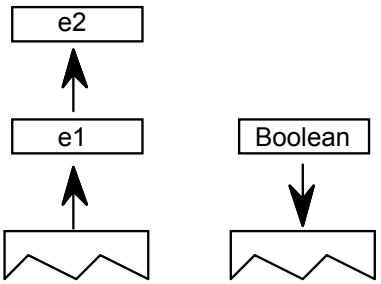
LT[ ]

Code Range 0x50

Pops e2: stack element (ULONG)
e1: stack element (ULONG)

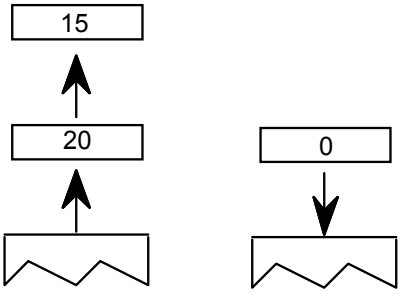
Pushes Boolean value (ULONG in the range [0,1])

Pops e2 and e1 off the stack and compares them: if e1 is less than e2, 1, signifying TRUE, is pushed onto the stack. If e1 is not less than e2, 0, signifying FALSE, is placed onto the stack.



Example:

LT[ ]

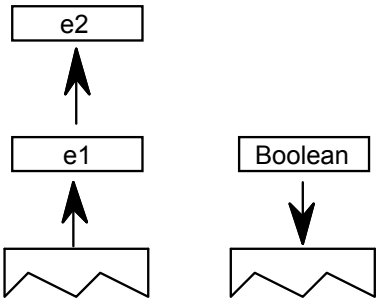


Less Than or Equal

LTEQ[ ]

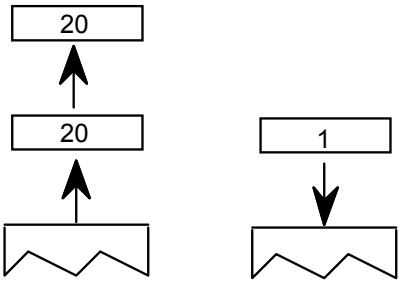
Code Range	0x51
Pops	e2: stack element (ULONG) e1: stack element (ULONG)
Pushes	Boolean value (ULONG in the range [0,1])

Pops e2 and e1 off the stack and compares them. If e1 is less than or equal to e2, 1, signifying TRUE, is pushed onto the stack. If e1 is not less than or equal to e2, 0, signifying FALSE, is placed onto the stack.



Example:

LTEQ[ ]



Greater Than

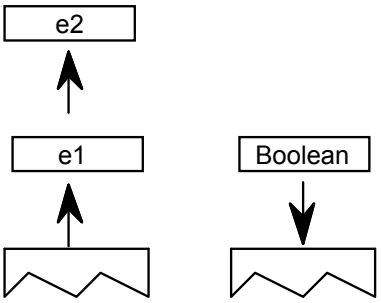
GT[ ]

Code Range 0x52

Pops e2: stack element (ULONG)
e1: stack element (ULONG)

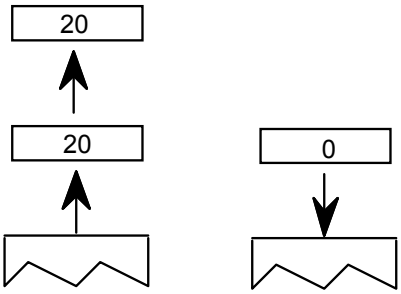
Pushes Boolean value (ULONG in the range [0,1])

Pops e1 and e2 off the stack and compares them. If e1 is greater than e2, 1, signifying TRUE, is pushed onto the stack. If e1 is not greater than e2, 0, signifying FALSE, is placed onto the stack.



Example:

GT[ ]



Greater Than or Equal

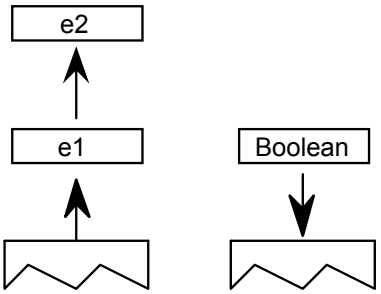
GTEQ[ ]

Code Range 0x53

Pops e2: stack element (ULONG)
e1: stack element (ULONG)

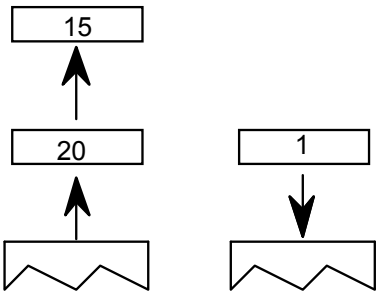
Pushes Boolean value (ULONG in the range [0,1])

Pops e1 and e2 off the stack and compares them. If e1 is greater than or equal to e2, 1, signifying TRUE, is pushed onto the stack. If e1 is not greater than or equal to e2, 0, signifying FALSE, is placed onto the stack.



Example:

GTEQ[ ]



EQual

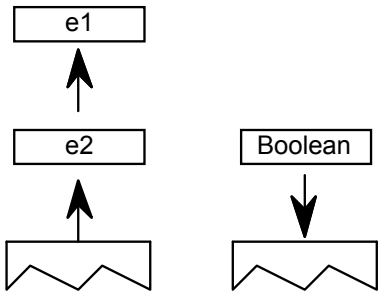
EQ[ ]

Code Range 0x54

Pops e1: stack element (ULONG)
e2: stack element (ULONG)

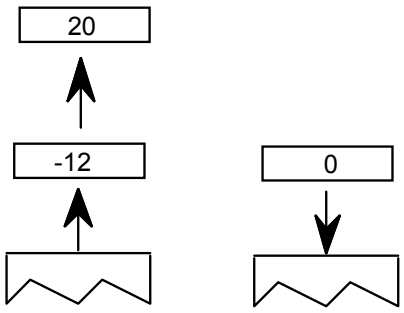
Pushes Boolean value (ULONG in the range [0,1])

Pops e1 and e2 off the stack and compares them. If they are equal, 1, signifying TRUE is pushed onto the stack. If they are not equal, 0, signifying FALSE is placed onto the stack.



Example:

EQ[ ]



Not EQual

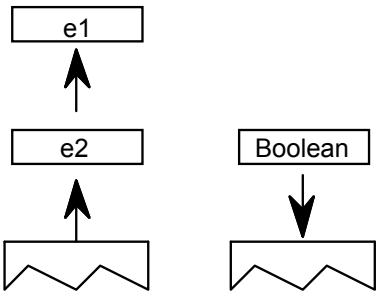
NEQ[ ]

Code Range 0x55

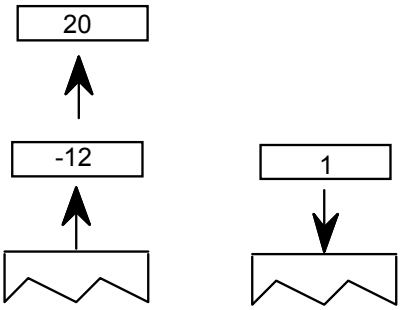
Pops e1:stack element (ULONG)
e2: stack element (ULONG)

Pushes Boolean value (ULONG in the range [0,1])

Pops e1 and e2 from the stack and compares them. If they are not equal, 1, signifying TRUE, is pushed onto the stack. If they are equal, 0, signifying FALSE, is placed on the stack.



Example:



ODD

ODD[ ]

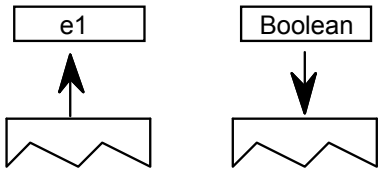
Code Range 0x56

Pops e1: stack element (F26Dot6)

Pushes Boolean value

Uses round_state

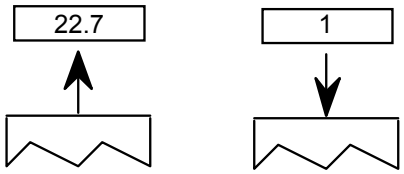
Tests whether the number at the top of the stack is odd. Pops e1 from the stack and rounds it as specified by the round_state before testing it. After the value is rounded, it is shifted from a fixed point value to an integer value (any fractional values are ignored). If the integer value is odd, one, signifying TRUE, is pushed onto the stack. If it is even, zero, signifying FALSE is placed onto the stack.



Example:

ODD[ ]

This example assumes that round_state is RTG.



EVEN

EVEN[ ]

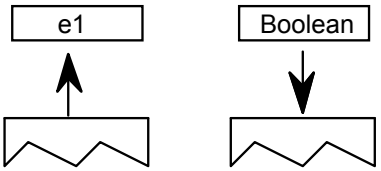
Code Range 0x57

Pops e1: stack element (F26Dot6)

Pushes Boolean value (ULONG in the range [0,1])

Uses round_state

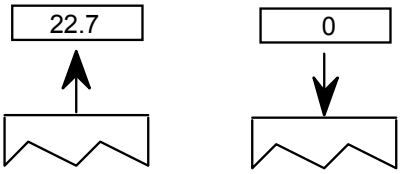
Tests whether the number at the top of the stack is even. Pops e1 off the stack and rounds it as specified by the round_state before testing it. If the rounded number is even, one, signifying TRUE, is pushed onto the stack if it is odd, zero, signifying FALSE, is placed onto the stack.



Example:

EVEN[ ]

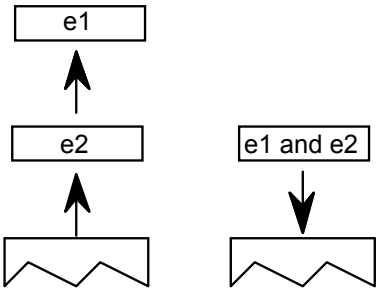
This example assumes that round_state is RTG.



logical AND

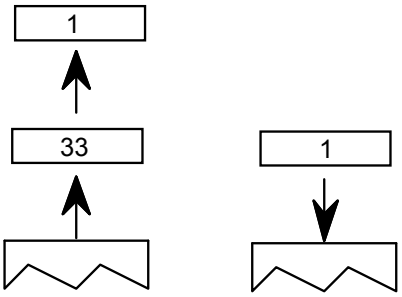
AND[ ]	
Code Range	0x5A
Pops	e1: stack element (ULONG) e2: stack element (ULONG)
Pushes	(e1 and e2): logical and of e1 and e2 (ULONG)

Pops e1 and e2 off the stack and pushes onto the stack the result of a logical and of the two elements. Zero is returned if either or both of the elements are FALSE (have the value zero). One is returned if both elements are TRUE (have a non zero value).



case 1:

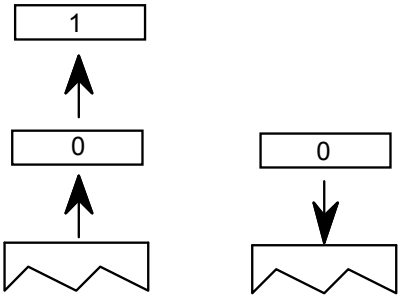
AND[ ]



(continued...)

case 2:

AND[ ]



logical OR

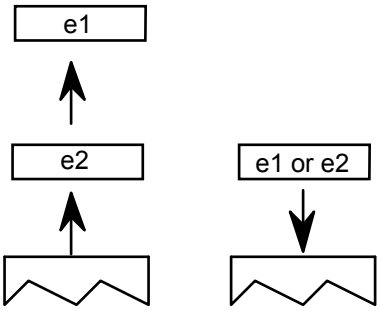
OR[ ]

Code Range 0x5B

Pops e1: stack element (ULONG)
e2: stack element (ULONG)

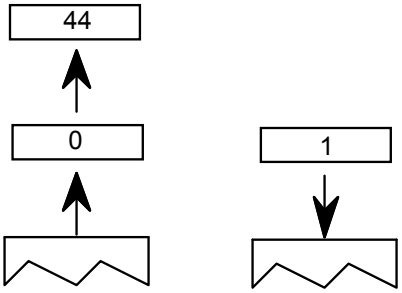
Pushes (e1 or e2): logical or of e1 and e2 (ULONG)

Pops e1 and e2 off the stack and pushes onto the stack the result of a logical or operation between the two elements. Zero is returned if both of the elements are FALSE. One is returned if either both of the elements are TRUE.



Example:

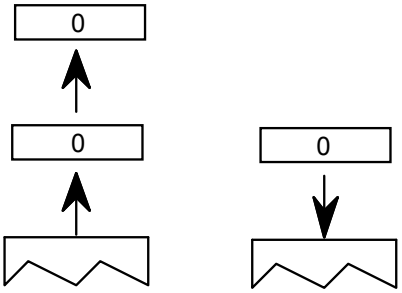
case 1:
OR[ ]



(continued...)

case 2:

OR[ ]



logical NOT

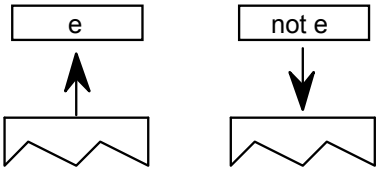
NOT[ ]

Code Range 0x5C

Pops e: stack element (ULONG)

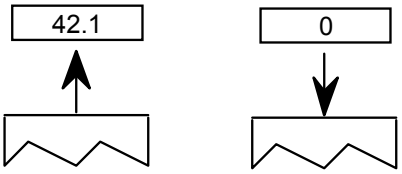
Pushes (not e): logical negation of e (ULONG)

Pops e off the stack and returns the result of a logical NOT operation performed on e. If originally zero, one is pushed onto the stack if originally nonzero, zero is pushed onto the stack.

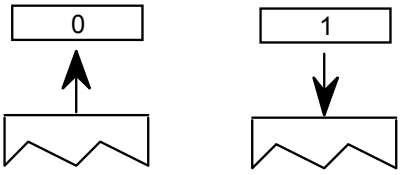


Example:

case 1:



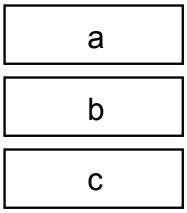
case 2:



Arithmetic and math instructions

These instructions perform arithmetic on stack values. Values are treated as signed (two's complement) 26.6 fixed-point numbers (F26Dot6) and give results in the same form. There is no overflow or underflow protection for these instructions.

To easily remember the order in which stack values are handled during arithmetic operations, imagine writing the stack values from left to right, starting with the bottom value. Then insert the operator between the two furthest right elements. For example:



subtract a,b would be interpreted as (b-a):

c b - a

ADD

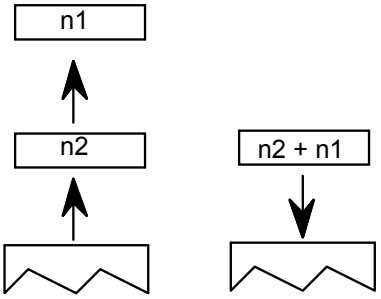
ADD[ ]

Code Range 0x60

Pops n1, n2 (F26Dot6)

Pushes (n2 + n1)

Pops n1 and n2 off the stack and pushes the sum of the two elements onto the stack.



SUBtract

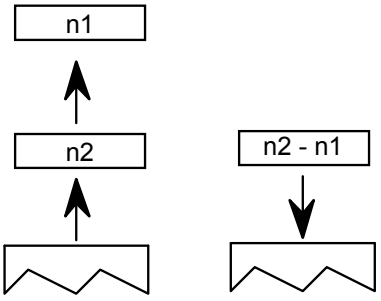
SUB[ ]

Code Range 0x61

Pops n1, n2 (F26Dot6)

Pushes (n2 - n1): difference

Pops n1 and n2 off the stack and pushes the difference between the two elements onto the stack.



DIVide

DIV[ ]

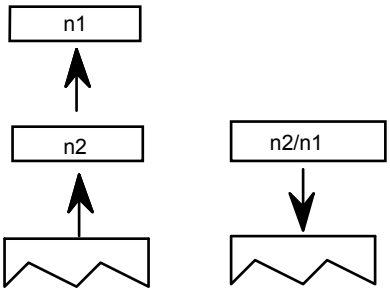
Code Range 0x62

Pops n1: divisor (F26Dot6)

n2: dividend (F26Dot6)

Pushes $\frac{n2}{n1}$ (F26Dot6)

Pops n1 and n2 off the stack and pushes onto the stack the quotient obtained by dividing n2 by n1. Note that this truncates rather than rounds the value.



MULtiply

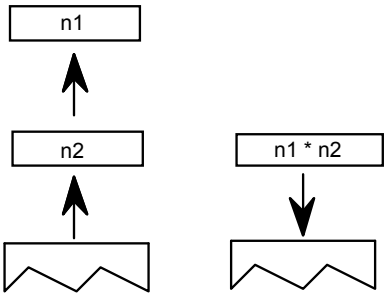
MUL[ ]

Code Range 0x63

Pops n1, n2: multiplier and multiplicand (F26Dot6)

Pushes n1 * n2 (F26Dot6)

Pops n1 and n2 off the stack and pushes onto the stack the product of the two elements.



ABSolute value

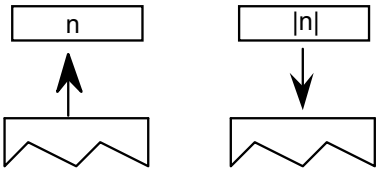
ABS[ ]

Code Range 0x64

Pops n

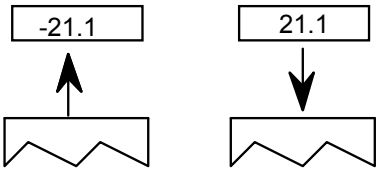
Pushes |n|: absolute value of n (F26Dot6)

Pops n off the stack and pushes onto the stack the absolute value of n.

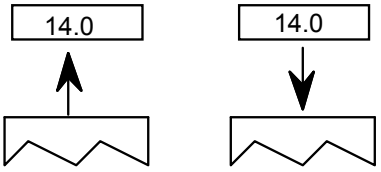


Example:

case 1:



case 2:



NEGate

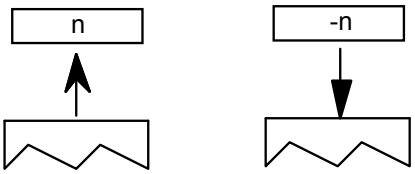
NEG[ ]

Code Range 0x65

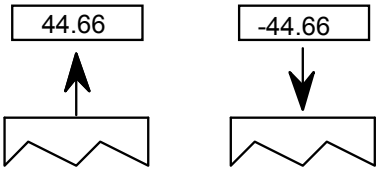
Pops n1

Pushes –n1: negation of n1 (F26Dot6)

This instruction pops n1 off the stack and pushes onto the stack the negated value of n1.



NEG[ ]



FLOOR

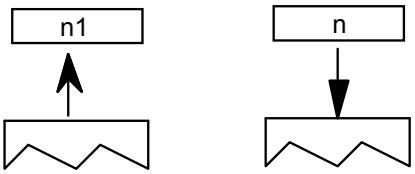
FLOOR[ ]

Code Range 0x66

Pops n1: number whose floor is desired (F26Dot6)

Pushes n : floor of n1 (F26Dot6)

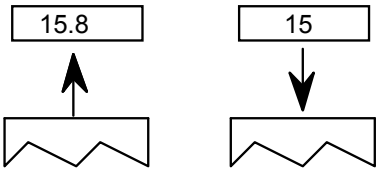
Pops n1 and returns n, the greatest integer value less than or equal to n1.



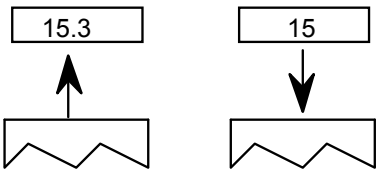
Example:

FLOOR[ ]

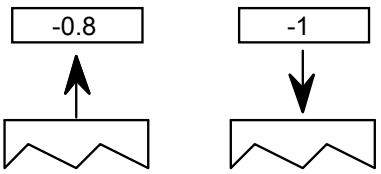
case 1:



case 2:



case 3:



CEILING

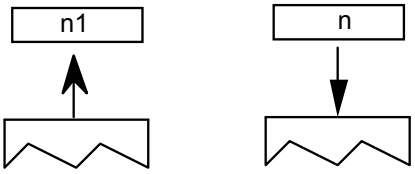
CEILING[ ]

Code Range 0x67

Pops n1: number whose ceiling is desired (F26Dot6)

Pushes n: ceiling of n1 (F26Dot6)

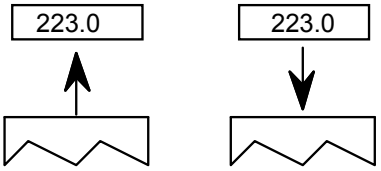
Pops n1 and returns n, the least integer value greater than or equal to n1. For instance, the ceiling of 15 is 15, but the ceiling of 15.3 is 16. The ceiling of -0.8 is 0. (n is the least integer value greater than or equal to n1)



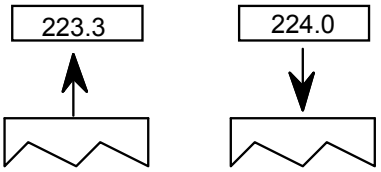
Example:

CEILING[ ]

case 1:



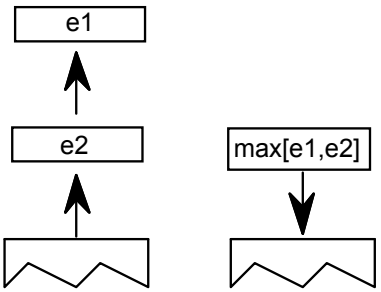
case 2:



MAXimum of top two stack elements

MAX[ ]	
Code Range	0x8B
Pops	e1: stack element (ULONG) e2: stack element (ULONG)
Pushes	maximum of e1 and e2

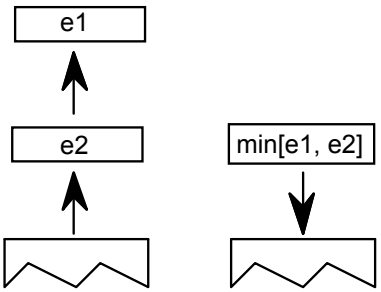
Pops two elements, e1 and e2, from the stack and pushes the larger of these two quantities onto the stack.



MINimum of top two stack elements

MIN[ ]	
Code Range	0x8C
Pops	e1: stack element (ULONG) e2: stack element (ULONG)
Pushes	minimum of e1 and e2

Pops two elements, e1 and e2, from the stack and pushes the smaller of these two quantities onto the stack.



Compensating for the engine characteristics

The following two functions make it possible to compensate for the engine characteristic. Each takes value and make the compensation. In addition to the engine compensation, ROUND, rounds the value according to the round_state. NROUND only compensates for the engine.

ROUND value

ROUND[ab]

Flags	ab: distance type for engine characteristic compensation
Pops	n1
Pushes	n2
Code	0x68 - 0x6B

Rounds a value according to the state variable round_state while compensating for the engine. n1 is popped off the stack and, depending on the engine characteristics, is increased or decreased by a set amount. The number obtained is then rounded and pushed back onto the stack as n2.

The value ab specifies the distance type as described in the chapter, “Instructing Glyphs.” Three values are possible: Gray=0, Black=1, White=2.

No ROUNDing of value

NROUND[ab]

Flags	ab: distance type for engine characteristic compensation
Pops	n1
Pushes	n2
Code	0x6C - 0x6F

NROUND[ab] does the same operation as ROUND[ab] (above), except that it does not round the result obtained after compensating for the engine characteristics. n1 is popped off the stack and, depending on the engine characteristics, increases or decreases by a set amount. This figure is then pushed back onto the stack as n2.

The value ab specifies the distance type as described in the chapter, “Instructing Glyphs.” Three values are possible: Gray=0, Black=1, White=2.

Defining and using functions and instructions

The following instructions make it possible to define and use both functions and new instructions.

In addition to a simple call function, there is a loop and call function.

An IDEF or instruction definition call makes it possible to patch old scalers in order to add newly defined instructions.

Function DEFinition

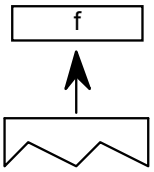
FDEF[ ]

Code Range 0x2C

Pops f: function identifier number (integer in the range 0 to n-1 where n is specified in the 'maxp' table)

Pushes –

Marks the start of a function definition. The argument f is a number that uniquely identifies this function. A function definition can appear only in the Font Program or the CVT program. Functions may not exceed 64K in size.



END Function definition

ENDF[ ]

Code Range 0x2D

Pops –

Pushes –

Marks the end of a function definition or an instruction definition.

CALL function

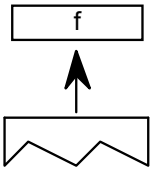
CALL[ ]

Code Range 0x2B

Pops f: function identifier number (integer in the range 0 to n-1 where n is specified in the ‘maxp’ table)

Pushes –

Calls the function identified by the number f.



LOOP and CALL function

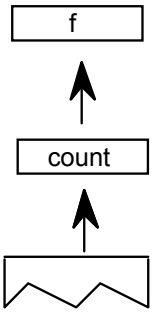
LOOPCALL[ ]

Code Range 0x2A

Pops f: function number integer in the range 0 to n-1 where n is specified in the 'maxp' table
count: number of times to call the function (signed word)

Pushes –

Calls the function f, count number of times.



Example:

Assume the Font Program contains this:

```
PUSHB(000), 17    push 17 onto the stack
FDEF[ ]           start defining function 17
  ...             contents of function
  ...
ENDF[ ]           end the definition
PUSHB(000),17     push function number
PUSHB(000),5      push count
LOOPCALL[ ]       call function17, 5 times
```

Instruction DEFinition

IDEF[ ]

Code Range 0x89

Pops opcode (8 bit code padded with zeroes to ULONG)

Pushes –

Begins the definition of an instruction. The instruction definition terminates when at ENDF, which is encountered in the instruction stream. Subsequent executions of the opcode popped will be directed to the contents of this instruction definition (IDEF). IDEFs should be defined in the Font Program or the CVT Program. An IDEF affects only undefined opcodes. If the opcode in question is already defined, the interpreter will ignore the IDEF. This is to be used as a patching mechanism for future instructions. Instructions may not exceed 64K in size.

Debugging

The TrueType instruction set provides the following instruction as an aid to debugging.

DEBUG call

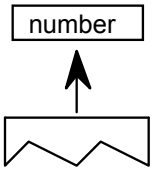
DEBUG[ ]

Code Range 0x4F

Pops number (ULONG)

Pushes –

This instruction is only for debugging purposes and should not be a part of a finished font. Some implementations may not support this instruction.



Miscellaneous instructions

The following instruction obtains information about the current glyph and the scaler version.

GET INFOrmation

GETINFO[ ]

Code Range 0x88

Pops selector (integer)

Pushes result (integer)

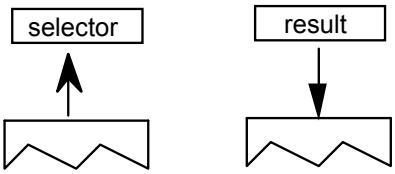
GETINFO is used to obtain data about the font scaler version and the characteristics of the current glyph. The instruction pops a selector used to determine the type of information desired and pushes a result onto the stack.

A selector value of 1 indicates that the scaler version number is the desired information, a value of 2 is a request for glyph rotation status, a value of 4 asks whether the glyph has been stretched. (Looking at this another way, setting bit 0 asks for the scaler version number, setting bit 1 asks for the rotation information, setting bit 2 ask for stretching information. To request information on two or more of these set the appropriate bits.)

The result pushed onto the stack contains the requested information. More precisely, bits 0 through 7 comprise the Scaler version number. Version 1 is Macintosh System 6 INIT, version 2 is Macintosh System 7, and version 3 is Windows 3.1. Version numbers 0 and 4 through 255 are reserved.

Bit 8 is set to 1 if the current glyph has been rotated. It is 0 otherwise. Setting bit 9 to 1 indicates that the glyph has been stretched. It is 0 otherwise.

When the selector is set to request more than one piece of information, that information is OR'd together and pushed onto the stack. For example, a selector value of 6 requests both information on rotation and stretching and will result in the setting of both bits 8 and 9.



Graphics State Summary

The following tables summarize the variables that make up the Graphics State. Nearly all of the Graphics State variables have a default value as shown below. That value is reestablished for every glyph in a font. Instructions are available for resetting the value of all Graphics State variables. Some state variables can be reset in the CVT Program. In such cases the value set becomes the new default and will be reestablished for each glyph. When value of a state variable is changed by instructions associated with a particular glyph, it will hold only for that glyph.

The setting of the Graphics State variables will affect the actions of certain instructions. Affected instructions are listed for each variable.

Graphics State Variable	Default	Set With	Affects
auto_flip	TRUE	FLIPOFF FLIPON	MIAP MIRP
control_value_cut_in	17/16 pixels	SCVTCI	MIAP MIRP
delta_base	9	SDB	DELTAP1 DELTAP2 DELTAP3 DELTAC1 DELTAC2 DELTAC3
delta_shift	3	SDS	DELTAP1 DELTAP2 DELTAP3 DELTAC1 DELTAC2 DELTAC3
dual_projection_vectors	—	SDPVTL	IP GC MD MDRP MIRP

freedom_vector	x-axis	SFVTCA SFVTL SFTPV SVTCA SFVFS	GFV
gep0	1	SCE0 SCES	AA ALIGNPTS ALIGNRP DELTAP1 DELTAP2 DELTAP3 IP ISECT MD MDAP MIAP MIRP MSIRP SHC SHE SHP UTP
gep1	1	SCE1 SCES	AA ALIGNPTS ALIGNRP IP MD MDRP MIRP MSIRP SHC SHE SHP SFVTL SPVTL

gep2	1	SCE3 SCES	ISECT IUP GC SHC SHP SFVTL SHPIX SPVTL SC
instruct_Control	0	INSTCTRL	all instructions
loop	1	SLOOP	ALIGNRP FLIPPT IP SHP SHPIX
minimum_distance	1 pixel	LMD	MDRP MIRP
projection_vector	x-axis	SPVTCA SPVTL SVTCA SPVFS	GPV
round_state	1	RDTG ROFF RTDG RTG RTHG RUTG SROUND S45ROUND	MDAP MIAP MDRP MIRP ROUND
rp0	0	SRP0	IP MDAP MIAP MIRP MSIRP SHC SHE SHP

rp1	0	SRP1	IP MDAP MDRP MIAP MSIRP SHC SHE SHP
rp2	0	SRP2	IP MDRP MIRP MSIRP SHC SHE SHP
scan_control	FALSE	SCANCTRL SCANTYPE	
singe_width_cut_in	0 pixels	SSWCI	MIAP MIRP
single_width_value	0 pixels	SSW	MIAP MIRP

Appendix A

IBM Font Class Parameters

This section defines the IBM Font Class and the IBM Font Subclass parameter values to be used in the classification of font designs by the font designer or supplier. This information is stored in the sFamilyClass field of a font's OS/2 table.

sFamilyClass

Format:	2-byte signed short
Title:	Font-family class and subclass. Also see section 3.4.
Description:	This parameter is a classification of font-family design.
Comments:	The font class and font subclass are registered values assigned by IBM to each font family. This parameter is intended for use in selecting an alternate font when the requested font is not available. The font class is the most general and the font subclass is the most specific. The high byte of this field contains the family class, while the low byte contains the family subclass.

These values classify a font design as to its appearance, but do not identify the specific font family, typeface variation, designer, supplier, size, or metric table differences. It should be noted that some font designs may be classified equally well into more than IBM Font Class or Subclass. Such designs should be matched to a classification for which substitution of another font design from the same class or subclass would generally result in a similar appearance of the presented document.

Class ID = 0 No Classification

This class ID is used to indicate that the associated font has no design classification or that the design classification is not of significance to the creator or user of the font resource.

Class ID = 1 Oldstyle Serifs

This style is generally based upon the Latin printing style of the 15th to 17th century, with a mild diagonal contrast in stroke emphasis (lighter in upper left to lower right, heavier in upper right to lower left) and bracketed serifs. This IBM Class reflects the ISO Serif Class, Oldstyle and Legibility Subclasses as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 0 : No Classification

This subclass ID is used to indicate that the associated font has no design sub-classification or that the design subclassification is not of significance to the creator or user of the font resource.

Subclass ID = 1 : IBM Rounded Legibility

This style is generally characterized by a large x-height, with short ascenders and descenders. Specifically, it is distinguished by a medium resolution, hand tuned, bitmap rendition of the more general rounded legibility subclass. An example of this font style is the IBM Sonoran Serif family. This IBM Subclass reflects the ISO Serif Class, Legibility Subclass, and Rounded Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 2 : Garalde

This style is generally characterized by a medium x-height, with tall ascenders. An example of this font style is the ITC Garamond family. This IBM Subclass reflects the ISO Serif Class, Oldstyle Subclass, and Garalde Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 3 : Venetian

This style is generally characterized by a medium x-height, with a relatively monotone appearance and sweeping tails based on the designs of the early Venetian printers. An example of this font style is the Goudy family. This IBM Subclass reflects the ISO Serif Class, Oldstyle Subclass, and Venetian Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 4 : Modified Venetian

This style is generally characterized by a large x-height, with a relatively monotone appearance and sweeping tails based on the designs of the early Venetian printers. An example of this font style is the Allied Linotype Palatino family. This IBM Subclass reflects the ISO Serif Class, Transitional Subclass, and Modified Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 5 : Dutch Modern

This style is generally characterized by a large x-height, with wedge shaped serifs and a circular appearance to the bowls similar to the Dutch Traditional Subclass below, but with lighter stokes. An example of this font style is the Monotype Times New Roman family. This IBM Subclass reflects the ISO Serif Class, Oldstyle Subclass, and Dutch Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 6 : Dutch Traditional

This style is generally characterized by a large x-height, with wedge shaped serifs and a circular appearance of the bowls. An example of this font style is the IBM Press Roman family. This IBM Subclass reflects the ISO Serif Class and Legibility Subclass as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 7 : Contemporary

This style is generally characterized by a small x-height, with light stokes and serifs. An example of this font style is the University family. This IBM Subclass reflects the ISO Serif Class and Contemporary Subclass as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 8 : Calligraphic

This style is generally characterized by the fine hand writing style of calligraphy, while retaining the characteristic Oldstyle appearance. This IBM Subclass is not reflected in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 9-14 : (reserved for future use)

These subclass IDs are reserved for future assignment, and shall not be used without formal assignment by IBM.

Subclass ID = 15 : Miscellaneous

This subclass ID is used for miscellaneous designs of the associated design class that are not covered by another Subclass.

Class ID = 2 Transitional Serifs

This style is generally based upon the Latin printing style of the 18th to 19th century, with a pronounced vertical contrast in stroke emphasis (vertical strokes being heavier than the horizontal strokes) and bracketed serifs. This IBM Class reflects the ISO Serif Class, Transitional Subclass as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 0 : No Classification

This subclass ID is used to indicate that the associated font has no design sub-classification or that the design sub-classification is not of significance to the creator or user of the font resource.

Subclass ID = 1 : Direct Line

This style is generally characterized by a medium x-height, with fine serifs, noticeable contrast, and capitol letters of approximately the same width. An example of this font style is the Monotype Baskerville family. This IBM Subclass reflects the ISO Serif Class, Transitional Subclass, and Direct Line Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 2 : Script

This style is generally characterized by a hand written script appearance while retaining the Transitional Direct Line style. An example of this font style is the IBM Nasseem (Arabic) family. This IBM Subclass is not specifically reflected in the 12/87 ISO/IEC 9541-5 draft standard, though the ISO Serif Class, Transitional Subclass, and Direct Line Specific Group would be a close approximation.

Subclass ID = 3-14 : (reserved for future use)

These subclass IDs are reserved for future assignment, and shall not be used without formal assignment by IBM.

Subclass ID = 15 : Miscellaneous

This subclass ID is used for miscellaneous designs of the associated design class that are not covered by another Subclass.

Class ID = 3 Modern Serifs

This style is generally based upon the Latin printing style of the 20th century, with an extreme contrast between the thick and thin portion of the strokes. This IBM Class reflects the ISO Serif Class, Modern Subclass as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 0 : No Classification

This subclass ID is used to indicate that the associated font has no design sub-classification or that the design sub-classification is not of significance to the creator or user of the font resource.

Subclass ID = 1 : Italian

This style is generally characterized by a medium x-height, with thin hairline serifs. An example of this font style is the Monotype Bodoni family. This IBM Subclass reflects the ISO Serif Class, Modern Subclass, and Italian Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 2 : Script

This style is generally characterized by a hand written script appearance while retaining the Modern Italian style. An example of this font style is the IBM Narkissim (Hebrew) family. This IBM Subclass is not specifically reflected in the 12/87 ISO/IEC 9541-5 draft standard, though the ISO Serif Class, Modern Subclass, and Italian Specific Group would be a close approximation.

Subclass ID = 3-14 : (reserved for future use)

These subclass IDs are reserved for future assignment, and shall not be used without formal assignment by IBM.

Subclass ID = 15 : Miscellaneous

This subclass ID is used for miscellaneous designs of the associated design class that are not covered by another Subclass.

Class ID = 4 Clarendon Serifs

This style is a variation of the Oldstyle Serifs and the Transitional Serifs, with a mild vertical stroke contrast and bracketed serifs. This IBM Class reflects the ISO Serif Class, Square Serif Subclass as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 0 : No Classification

This subclass ID is used to indicate that the associated font has no design sub-classification or that the design sub-classification is not of significance to the creator or user of the font resource.

Subclass ID = 1 : Clarendon

This style is generally characterized by a large x-height, with serifs and strokes of equal weight. An example of this font style is the Allied Linotype Clarendon family. This IBM Subclass reflects the ISO Serif Class, Square Serif Subclass, and Clarendon Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 2 : Modern

This style is generally characterized by a large x-height, with serifs of a lighter weight than the strokes and the strokes of a lighter weight than the Traditional. An example of this font style is the Monotype Century Schoolbook family. This IBM Subclass reflects the ISO Serif Class, Square Serif Subclass, and Clarendon Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 3 : Traditional

This style is generally characterized by a large x-height, with serifs of a lighter weight than the strokes. An example of this font style is the Monotype Century family.This IBM Subclass reflects the ISO Serif Class, Square Serif Subclass, and Clarendon Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 4 : Newspaper

This style is generally characterized by a large x-height, with a simpler style of design and serifs of a lighter weight than the strokes. An example of this font style is the Allied Linotype Excelsior Family. This IBM Subclass reflects the ISO Serif Class, Square Serif Subclass, and Clarendon Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 5 : Stub Serif

This style is generally characterized by a large x-height, with short stub serifs and relatively bold stems. An example of this font style is the Cheltenham Family. This IBM Subclass reflects the ISO Serif Class, Square Serif Subclass, and Short Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 6 : Monotone

This style is generally characterized by a large x-height, with monotone stems. An example of this font style is the ITC Korinna Family. This IBM Subclass reflects the ISO Serif Class, Square Serif Subclass, and Monotone Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 7 : Typewriter

This style is generally characterized by a large x-height, with moderate stroke thickness characteristic of a typewriter. An example of this font style is the Prestige Elite Family. This IBM Subclass reflects the ISO Serif Class, Square Serif Subclass, and Typewriter Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 8-14: (reserved for future use)

These subclass IDs are reserved for future assignment, and shall not be used without formal assignment by IBM.

Subclass ID = 15 : Miscellaneous

This subclass ID is used for miscellaneous designs of the associated design class that are not covered by another Subclass.

Class ID = 5 Slab Serifs

This style is characterized by serifs with a square transition between the strokes and the serifs (no brackets). This IBM Class reflects the ISO Serif Class, Square Serif Subclass (except the Clarendon Specific Group) as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 0 : No Classification

This subclass ID is used to indicate that the associated font has no design sub-classification or that the design sub-classification is not of significance to the creator or user of the font resource.

Subclass ID = 1 : Monotone

This style is generally characterized by a large x-height, with serifs and strokes of equal weight. An example of this font style is the ITC Lubalin Family. This IBM Subclass reflects the ISO Serif Class, Square Serif Subclass, and Monotone Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 2 : Humanist

This style is generally characterized by a medium x-height, with serifs of lighter weight that the strokes. An example of this font style is the Candida Family. This IBM Subclass reflects the ISO Serif Class, Square Serif Subclass, and Monotone Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 3 : Geometric

This style is generally characterized by a large x-height, with serifs and strokes of equal weight and a geometric (circles and lines) design. An example of this font style is the Monotype Rockwell Family. This IBM Subclass reflects the ISO Serif Class, Square Serif Subclass, and Monotone Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 4 : Swiss

This style is generally characterized by a large x-height, with serifs and strokes of equal weight and an emphasis on the white space of the characters. An example of this font style is the Allied Linotype Serifa Family. This IBM Subclass reflects the ISO Serif Class, Square Serif Subclass, and Monotone Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 5 : Typewriter

This style is generally characterized by a large x-height, with serifs and strokes of equal but moderate thickness, and a geometric design. An example of this font style is the IBM Courier Family. This IBM Subclass reflects the ISO Serif Class, Square Serif Subclass, and Monotone Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 6-14 : (reserved for future use)

These subclass IDs are reserved for future assignment, and shall not be used without formal assignment by IBM.

Subclass ID = 15 : Miscellaneous

This subclass ID is used for miscellaneous designs of the associated design class that are not covered by another Subclass.

Class ID = 6 (reserved for future use)

This class ID is reserved for future assignment, and shall not be used without formal assignment by IBM.

Class ID = 7 Freeform Serifs

This style includes serifs, but which expresses a design freedom that does not generally fit within the other serif design classifications. This IBM Class reflects the remaining ISO Serif Class subclasses as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 0 : No Classification

This subclass ID is used to indicate that the associated font has no design sub-classification or that the design sub-classification is not of significance to the creator or user of the font resource.

Subclass ID = 1 : Modern

This style is generally characterized by a medium x-height, with light contrast in the strokes and a round full design. An example of this font style is the ITC Souvenir Family. This IBM Subclass is not reflected in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 2-14 : (reserved for future use)

These subclass IDs are reserved for future assignment, and shall not be used without formal assignment by IBM.

Subclass ID = 15 : Miscellaneous

This subclass ID is used for miscellaneous designs of the associated design class that are not covered by another Subclass.

Class ID = 8 Sans Serif

This style includes most basic letter forms (excluding Scripts and Ornamentals) that do not have serifs on the strokes. This IBM Class reflects the ISO Sans Serif Class as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 0 : No Classification

This subclass ID is used to indicate that the associated font has no design sub-classification or that the design sub-classification is not of significance to the creator or user of the font resource.

Subclass ID = 1 : IBM Neo-grotesque Gothic

This style is generally characterized by a large x-height, with uniform stroke width and a simple one story design distinguished by a medium resolution, hand tuned, bitmap rendition of the more general Neo-grotesque Gothic Subclass. An example of this font style is the IBM Sonoran Sans Serif family. This IBM Subclass reflects the ISO Sans Serif Class, Gothic Subclass, and Neo-grotesque Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 2 : Humanist

This style is generally characterized by a medium x-height, with light contrast in the strokes and a classic Roman letterform. An example of this font style is the Allied Linotype Optima family. This IBM Subclass reflects the ISO Sans Serif Class, Humanist Subclass as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 3 : Low-x Round Geometric

This style is generally characterized by a low x-height, with monotone stroke weight and a round geometric design. An example of this font style is the Fundicion Tipograficia Neufville Futura family. This IBM Subclass reflects the ISO Sans Serif Class, Geometric Subclass, Round Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 4 : High-x Round Geometric

This style is generally characterized by a high x-height, with uniform stroke weight and a round geometric design. An example of this font style is the ITC Avant Garde Gothic family. This IBM Subclass reflects the ISO Sans Serif Class, Geometric Subclass, Round Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 5 : Neo-grotesque Gothic

This style is generally characterized by a high x-height, with uniform stroke width and a simple one story design. An example of this font style is the Allied Linotype Helvetica family. This IBM Subclass reflects the ISO Sans Serif Class, Gothic Subclass, Neo-grotesque Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 6 : Modified Neo-grotesque Gothic

This style is similar to the Neo-grotesque Gothic style, with design variations to the G and Q. An example of this font style is the Allied Linotype Univers family. This IBM Subclass reflects the ISO Sans Serif Class, Gothic Subclass, Neo-grotesque Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 7-8 : (reserved for future use)

These subclass IDs are reserved for future assignment, and shall not be used without formal assignment by IBM.

Subclass ID = 9 : Typewriter Gothic

This style is similar to the Neo-grotesque Gothic style, with moderate stroke thickness characteristic of a typewriter. An example of this font style is the IBM Letter Gothic family. This IBM Subclass reflects the ISO Sans Serif Class, Gothic Subclass, Typewriter Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 10 : Matrix

This style is generally a simple design characteristic of a dot matrix printer. An example of this font style is the IBM Matrix Gothic family. This IBM Subclass is not reflected in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 11-14 : (reserved for future use)

These subclass IDs are reserved for future assignment, and shall not be used without formal assignment by IBM.

Subclass ID = 15 : Miscellaneous

This subclass ID is used for miscellaneous designs of the associated design class that are not covered by another Subclass.

Class ID = 9 Ornamentals

This style includes highly decorated or stylized character shapes that are typically used in headlines. This IBM Class reflects the ISO Ornamental Class and the ISO Blackletter Class as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 0 : No Classification

This subclass ID is used to indicate that the associated font has no design sub-classification or that the design sub-classification is not of significance to the creator or user of the font resource.

Subclass ID = 1 : Engraver

This style is characterized by fine lines or lines engraved on the stems. An example of this font style is the Copperplate family. This IBM Subclass reflects the ISO Ornamental Class and Inline Subclass, or the Serif Class and Engraving Subclass as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 2 : Black Letter

This style is generally based upon the printing style of the German monasteries and printers of the 12th to 15th centuries. An example of this font style is the Old English family. This IBM Subclass reflects the ISO Blackletters Class as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 3 : Decorative

This style is characterized by ornamental designs (typically from nature, such as leaves, flowers, animals, etc.) incorporated into the stems and strokes of the characters. An example of this font style is the Saphire family. This IBM Subclass reflects the ISO Ornamental Class and Decorative Subclass as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 4 : Three Dimensional

This style is characterized by a three dimensional (raised) appearance of the characters created by shading or geometric effects. An example of this font style is the Thorne Shaded family. This IBM Subclass reflects the ISO Ornamental Class and Three Dimensional Subclass as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 5-14 : (reserved for future use)

These subclass IDs are reserved for future assignment, and shall not be used without formal assignment by IBM.

Subclass ID = 15 : Miscellaneous

This subclass ID is used for miscellaneous designs of the associated design class that are not covered by another Subclass.

Class ID = 10 Scripts

This style includes those typefaces that are designed to simulate handwriting. This IBM Class reflects the ISO Script Class and Uncial Class as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 0 : No Classification

This subclass ID is used to indicate that the associated font has no design sub-classification or that the design sub-classification is not of significance to the creator or user of the font resource.

Subclass ID = 1 : Uncial

This style is characterized by unjoined (nonconnecting) characters that are generally based on the hand writing style of Europe in the 6th to 9th centuries. An example of this font style is the Libra family. This IBM Subclass reflects the ISO Uncial Class as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 2 : Brush Joined

This style is characterized by joined (connecting) characters that have the appearance of being painted with a brush, with moderate contrast between thick and thin strokes. An example of this font style is the Mistral family. This IBM Subclass reflects the ISO Script Class, Joined Subclass, and Informal Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 3 : Formal Joined

This style is characterized by joined (connecting) characters that have a printed (or drawn with a stiff brush) appearance with extreme contrast between the thick and thin strokes. An example of this font style is the Coronet family. This IBM Subclass reflects the ISO Script Class, Joined Subclass, and Formal Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 4 : Monotone Joined

This style is characterized by joined (connecting) characters that have a uniform appearance with little or no contrast in the strokes. An example of this font style is the Kaufmann family. This IBM Subclass reflects the ISO Script Class, Joined Subclass, and Monotone Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 5 : Calligraphic

This style is characterized by beautifully hand drawn, unjoined (non-connecting) characters that have an appearance of being drawn with a broad edge pen. An example of this font style is the Thompson Quillscript family. This IBM Subclass reflects the ISO Script Class, Unjoined Subclass, and Calligraphic Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 6 : Brush Unjoined

This style is characterized by unjoined (non-connecting) characters that have the appearance of being painted with a brush, with moderate contrast between thick and thin strokes. An example of this font style is the Saltino family. This IBM Subclass reflects the ISO Script Class, Unjoined Subclass, and Brush Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 7 : Formal Unjoined

This style is characterized by unjoined (non-connecting) characters that have a printed (or drawn with a stiff brush) appearance with extreme contrast between the thick and thin strokes. An example of this font style is the Virtuosa family. This IBM Subclass reflects the ISO Script Class, Unjoined Subclass, and Formal Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 8 : Monotone Unjoined

This style is characterized by unjoined (non-connecting) characters that have a uniform appearance with little or no contrast in the strokes. An example of this font style is the Gilles Gothic family. This IBM Subclass reflects the ISO Script Class, Unjoined Subclass, and Monotone Specific Group as documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 9-14 : (reserved for future use)

These subclass IDs are reserved for future assignment, and shall not be used without formal assignment by IBM.

Subclass ID = 15 : Miscellaneous

This subclass ID is used for miscellaneous designs of the associated design class that are not covered by another Subclass.

Class ID = 11 (reserved for future use)

This class ID is reserved for future assignment, and shall not be used without formal assignment by IBM.

Class ID = 12 Symbolic

This style is generally design independent, making it suitable for Pi and special characters (icons, dingbats, technical symbols, etc.) that may be used equally well with any font. This IBM Class reflects various ISO Specific Groups, as noted below and documented in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 0 : No Classification

This subclass ID is used to indicate that the associated font has no design sub-classification or that the design sub-classification is not of significance to the creator or user of the font resource.

Subclass ID = 1-2 : (reserved for future use)

These subclass IDs are reserved for future assignment, and shall not be used without formal assignment by IBM.

Subclass ID = 3 : Mixed Serif

This style is characterized by either both or a combination of serif and sans serif designs on those characters of the font for which design is important (e.g., superscript and subscript characters, numbers, copyright or trademark symbols, etc.). An example of this font style is found in the IBM Symbol family. This IBM Subclass is not reflected in the 12/87 ISO/IEC 9541-5 draft standard.

Subclass ID = 4-5 : (reserved for future use)

These subclass IDs are reserved for future assignment, and shall not be used without formal assignment by IBM.

Subclass ID = 6 : Oldstyle Serif

This style is characterized by a Oldstyle Serif IBM Class design on those characters of the font for which design is important (e.g., superscript and subscript characters, numbers, copyright or trademark symbols, etc.). An example of this font style is found in the IBM Sonoran Pi Serif family. This IBM Subclass is not directly reflected in the 12/87 ISO/IEC 9541-5 draft standard, though it is indirectly by the ISO Serif Class and Legibility Subclass (implies that all characters of the font exhibit the design appearance, while only a subset of the characters actually exhibit the design).

Subclass ID = 7 : Neo-grotesque Sans Serif

This style is characterized by a Neo-grotesque Sans Serif IBM Font Class and Subclass design on those characters of the font for which design is important (e.g., superscript and subscript characters, numbers, copyright or trademark symbols, etc.). An example of this font style is found in the IBM Sonoran Pi Sans Serif family. This IBM Subclass is not directly reflected in the 12/87 ISO/IEC 9541-5 draft standard, though it is indirectly by the ISO Sans Serif Class and Gothic Subclass (implies that all characters of the font exhibit the design appearance, while only a subset of the characters actually exhibit the design).

Subclass ID = 8-14 : (reserved for future use)

These subclass IDs are reserved for future assignment, and shall not be used without formal assignment by IBM.

Subclass ID = 15 : Miscellaneous

This subclass ID is used for miscellaneous designs of the associated design class that are not covered by another Subclass.

Class ID = 13 Reserved

Class ID = 14 Reserved

Instruction Set Summary

Instructions by Category

Pushing data onto the interpreter stack

Instruction	Opcode	Takes from IS	Pushes
NPUSHB[ ]	0x40	n, b1, b2,...bn	b1,b2...bn
NPUSHW[ ]	0x41	n, w1, w2,...wn	w1,w2...wn
PUSHB[abc]	0xB0 – 0xB7	b0, b1,..bn	b0, b1, ...,bn
PUSHW[abc]	0xB8 – 0xBF	w0,w1,..wn	w0 ,w1, ...wn

Managing the Storage Area

Instruction	Opcode	Pops	Pushes
RS[ ]	0x43	location	value
WS[ ]	0x42	value, locat ion	–

Managing the Control Value Table

Instruction	Opcode	Pops	Pushes
WCVTP[ ]	0x44	value, location	–
WCVTF[ ]	0x70	value, location	–
RCVT[ ]	0x45	location	value

Managing the Graphics State

Setting and getting the freedom_vector and the projection_vector

Instruction	Opcode	Pops	Pushes
SVTCA[a]	0x00 – 0x01	–	–
SPVTCA[a]	0x02 – 0x03	–	–
SFVTCA[a]	0x04 – 0x05	–	–
SPVTL[a]	0x06 – 0x07	p1, p2	–
SFVTL[a]	0x08 – 0x09	p1, p2	–
SFVTPV[ ]	0x0E	–	–
SDPVTL[a]	0x86 – 0x87	p1, p2	–
SPVFS[ ]	0x0A	y, x	–
SFVFS[ ]	0x0B	y, x	–
GPV[ ]	0x0C	–	x, y
GFV[ ]	0x0D	–	x, y

Setting reference points and glyph element pointers

Instruction	Opcode	Pops	Pushes
SRP0[ ]	0x10	p	–
SRP1[ ]	0x11	p	–
SRP2[ ]	0x12	p	–
SZP0[ ]	0x13	n	–
SZP1[ ]	0x14	n	–
SZP2[ ]	0x15	n	–
SZPS[ ]	0x16	n	–

Setting the round_state

Instruction	Opcode	Pops	Pushes
RTHG[ ]	0x19	–	–
RTG[ ]	0x18	–	–
RTDG[ ]	0x3D	–	–
RDTG[ ]	0x7D	–	–
RUTG[ ]	0x7C	–	–
ROFF[ ]	0x7A	–	–
SROUND[ ]	0x76	n	–
S45ROUND[ ]	0x77	n	–

Setting other graphics state variables

Instruction	Opcode	Pops	Pushes
INSTCTRL	0x8E	selector and value	–
SLOOP[ ]	0x17	n	–
SMD[ ]	0x1A	distance	–
SCANCTRL[ ]	0x85	n	–
SCANTYPE[ ]	0x8D	n	–
SCVTCI[ ]	0x1D	n	–
SSWCI[ ]	0x1E	n	–
SSW[ ]	0x1F	n	–
FLIPON[ ]	0x4D	–	–
FLIPOFF[ ]	0x4E	–	–
SANGW[ ]	0x7E	weight	–
SDB[ ]	0x5E	n	–
SDS[ ]	0x5F	n	–

Reading and Writing Data

Instruction	Opcode	Pops	Pushes
GC[a]	0x46 – 0x47	p	value
SCFS[ ]	0x48	value, p	–
MD[a]	0x49 – 0x4A	p1,p2	distance

MPPEM[ ]	0x4B	–	ppem
MPS[ ]	0x4C	–	pointSize

Managing Outlines

Flipping Points

Instruction	Opcode	Pops	Pushes
FLIPRGON[ ]	0x81	highpoint, lowpoint	–
FLIPRGOFF[ ]	0x82	highpoint, lowpoint	–
FLIPPT[ ]	0x80	p	–

Shifting Points

Instruction	Opcode	Pops	Pushes
SHC[a]	0x34 – 0x35	c	–
SHZ[a]	0x36 – 0x37	e	–
SHPIX[ ]	0x38	p1, p2, amount	–
SHP	0x32 – 0x33	p	–

Moving Points

Instruction	Opcode	Pops	Pushes
MDAP[a]	0x2E – 0x2F	p	–
MIAP[a]	0x3E – 0x3F	n, p	–
MDRP[abcde]	0xC0 – 0xDF	p	–
MIRP[abcde]	0xE0 – 0xFF	n, p	–
ALIGNRP[ ]	0x3C	p1, p2, ... , ploop	–
AA[ ]	0x7F	p	–
ISECT[ ]	0x0F	a1, a0, b1, b0, p	–
ALIGNPTS[ ]	0x27	p1, p2	–

Interpolating Points

Instruction	Opcode	Pops	Pushes
UTP[ ]	0x29	p	–
IUP[a]	0x30 – 0x31	–	–

Managing Exceptions

Instruction	Opcode	Pops	Pushes
DELTAP1[ ]	0x5D	n, p1, arg1, ..., pn, argn	–
DELTAP2[ ]	0x71	n, p1, arg1, ..., pn, argn	–

DELTAP3[ ]	0x72	n, p1, arg1, ..., pn, argn	–
DELTAC1[ ],	0x73	n, c1, arg1, ..., cn, argn	–
DELTAC2[ ]	0x74	n, c1, arg1, ..., cn, argn	–
DELTAC3[ ]	0x75	n, c1, arg1, ..., cn, argn	–

Managing the stack

Instruction	Opcode	Pops	Pushes
DUP[ ]	0x20	e	e, e
POP[ ]	0x21	e	–
CLEAR[ ]	0x22	*all items on the stack*	–
SWAP[ ]	0x23	e1, e2	e1, e2
DEPTH[ ]	0x24	–	n
CINDEX[ ]	0x25	k	e_k
MINDEX[ ]	0x26	k, e1, e2, ..., ek	ek–1, ek–2, ..., e1, ek
ROLL	0x8a	–	–

Logical functions

Instruction	Opcode	Pops	Pushes
LT[ ]	0x50	e1, e2	(e2 < e1 *Boolean value*
LTEQ[ ]	0x51	e1, e2	(e2 ≤ e1) *Boolean value*
GT[ ]	0x52	e1, e2	(e2 > e1) *Boolean value*
GTEQ[ ]	0x53	e1, e2	(e2 ≥ e1) *Boolean value*
EQ[ ]	0x54	e1, e2	(e2 = e1) *Boolean value*
NEQ[ ]	0x55	e1, e2	(e2 ≠ e1) *Boolean value*
ODD[ ]	0x56	e	(e mod 2) = 1 *Boolean value*
EVEN[ ]	0x57	e	(e mod 2) = 0 *Boolean value*
AND[ ]	0x5A	e1, e2	(e2 and e1) *Boolean value*
OR[ ]	0x5B	e1, e2	(e2 or e1)*Boolean value*
NOT[ ]	0x5C	e	(not e)*Boolean value*

Managing the flow of control

Instruction	Opcode	Pops	Pushes
IF[ ]	0x58	e	–
EIF[ ]	0x59	–	–
ELSE	0x1B		
JMPR	0x1C	offset	
JROT[ ]	0x78	e, offset	–
JROF[ ]	0x79	e, offset	–

Arithmetic functions

Instruction	Opcode	Pops	Pushes
ADD[ ]	0x60	n1, n2	(n1 + n2)
SUB[ ]	0x61	n1, n2	(n2 – n1)
DIV[ ]	0x62	n1, n2	(n2 ¥ 64)/ n1
MUL[ ]	0x63	n1, n2	(n1 ¥ n2)/64
ABS[ ]	0x64	n	|n|
NEG[ ]	0x65	n1	–n1
FLOOR[ ]	0x66	n1	n
CEILING[ ]	0x67	n1	n
MAX[ ]	0X8B	e1, e2	max(e1, e2)
MIN[ ]	0X8C	e1, e2	min(e1, e2)

Compensating for the engine characteristics

Instruction	Opcode	Pops	Pushes
ROUND[ab]	0x68 – 0x6B	n1	n2
NROUND[ab]	0x6C – 0x6F	n1	n2

Defining and using functions and instructions

Instruction	Opcode	Pops	Pushes
FDEF[ ]	0x2C	f	–
ENDF[ ]	0x2D	–	–
CALL[ ]	0x2B	f	–
LOOPCALL[ ]	0x2A	f, count	–
IDEF[ ]	0x89	o	–

Debugging

Instruction	Opcode	Pops	Pushes
DEBUG[ ]	0x4F	opcode	–

Miscellaneous

Instruction	Opcode	Pops	Pushes
GETINFO[ ]	0x88	selector	result

Instructions by Name

Instruction	Opcode	Takes from IS	Pushes
AA[ ]	0x7F	p	–
ABS[ ]	0x64	n	|n|
ADD[ ]	0x60	n1, n2	(n1 + n2)
ALIGNPTS[ ]	0x27	p1, p2	–
ALIGNRP[ ]	0x3C	p1, p2, ... , ploop	–
AND[ ]	0x5A	e1, e2	(e2 and e1) *Boolean value*
CALL[ ]	0x2B	f	–
CEILING[ ]	0x67	n1	n
CINDEX[ ]	0x25	k	e_k
CLEAR[ ]	0x22	*all items on the stack*	–
DEBUG[ ]	0x4F	opcode	–
DELTAC1[ ],	0x73	n, c1, arg1, ..., cn, argn	–
DELTAC2[ ]	0x74	n, c1, arg1, ..., cn, argn	–
DELTAC3[ ]	0x75	n, c1, arg1, ..., cn, argn	–
DELTAP1[ ]	0x5D	n, p1, arg1, ..., pn, argn	–
DELTAP2[ ]	0x71	n, p1, arg1, ..., pn, argn	–
DELTAP3[ ]	0x72	n, p1, arg1, ..., pn, argn	–
DEPTH[ ]	0x24	–	n
DIV[ ]	0x62	n1, n2	(n2 * 64)/ n1
DUP[ ]	0x20	e	e, e
EIF[ ]	0x59	–	–
ELSE	0x1B		
ENDF[ ]	0x2D	–	–
EQ[ ]	0x54	e1, e2	(e2= e1) *Boolean value*
EVEN[ ]	0x57	e	(e mod 2) = 0 *Boolean value*
FDEF[ ]	0x2C	f	–
FLIPOFF[ ]	0x4E	–	–
FLIPON[ ]	0x4D	–	–
FLIPPT[ ]	0x80	p1, p2, ..., ploop	–
FLIPRGOFF[ ]	0x82	highpoint, lowpoint	–
FLIPRGON[ ]	0x81	highpoint, lowpoint	–
FLOOR[ ]	0x66	n1	n
GC[a]	0x46 – 0x47	p	value
GETINFO[ ]	0x88	selector	result
GFV[ ]	0x0D	–	x, y

GPV[ ]	0x0C	–	x, y
GT[ ]	0x52	e1, e2	(e2 > e1) *Boolean value*
GTEQ[ ]	0x53	e1, e2	(e2 ≥ e1) *Boolean value*
IDEF[ ]	0x89	o	–
IF[ ]	0x58	e	–
INSTCTRL	0x8E	value	–
IP[ ]	0x39	p1, p2, ... , ploop	–
ISECT[ ]	0x0F	a1, a0, b1, b0, p	–
IUP[a]	0x30 – 0x31	–	–
JMPR	0x1C	offset	
JROF[ ]	0x79	e, offset	–
JROT[ ]	0x78	e, offset	–
LOOPCALL[ ]	0x2A	f, count	–
LT[ ]	0x50	e1, e2	(e2 < e1 *Boolean value*
LTEQ[ ]	0x51	e1, e2	(e2 ≤ e1) *Boolean value*
MAX[ ]	0X8B	e1, e2	max(e1, e2)
MD[a]	0x49 – 0x4A	p1,p2	distance
MDAP[a]	0x2E – 0x2F	p	–
MDRP[abcde]	0xC0 – 0xDF	p	–
MIAP[a]	0x3E – 0x3F	n, p	–
MIN[ ]	0X8C	e1, e2	min(e1, e2)
MINDEX[ ]	0x26	k, e1, e2, ..., ek	ek–1, ek–2, ..., e1, ek
MIRP[abcde]	0xE0 – 0xFF	n, p	–
MPPEM[ ]	0x4B	–	ppem
MPS[ ]	0x4C	–	pointSize
MSIRP[a]	0x3A – 0x3B	distance, p	–
MUL[ ]	0x63	n1, n2	(n1 ¥ n2)/64
NEG[ ]	0x65	n1	–n1
NEQ[ ]	0x55	e1, e2	(e2 ≠ e1) *Boolean value*
NOT[ ]	0x5C	e	(not e)*Boolean value*
NPUSHB[ ]	0x40	n, b1, b2,...bn	b1,b2...bn
NPUSHW[ ]	0x41	n, w1, w2,...wn	w1,w2...wn
NROUND[ab]	0x6C – 0x6F	n1	n2
ODD[ ]	0x56	e	(e mod 2) = 1 *Boolean value*
OR[ ]	0x5B	e1, e2	(e2 or e1)*Boolean value*
POP[ ]	0x21	e	–
PUSHB[abc]	0xB0 – 0xB7	b0, b1,..bn	b0, b1, ...,bn

PUSHW[abc]	0xB8 – 0xBF	w0,w1,..wn	w0 ,w1, ...wn
RCVT[ ]	0x45	location	value
RDTG[ ]	0x7D	–	–
ROFF[ ]	0x7A	–	–
ROLL	0x8a	–	–
ROUND[ab]	0x68 – 0x6B	n1	n2
RS[ ]	0x43	location	value
RTDG[ ]	0x3D	–	–
RTG[ ]	0x18	–	–
RTHG[ ]	0x19	–	–
RUTG[ ]	0x7C	–	–
S45ROUND[ ]	0x77	n	–
SANGW[ ]	0x7E	weight	–
SCANCTRL[ ]	0x85	n	–
SCANTYPE[ ]	0x8D	n	–
SCFS[ ]	0x48	value, p	–
SCVTCI[ ]	0x1D	n	–
SDB[ ]	0x5E	n	–
SDPVTL[a]	0x86 – 0x87	p1, p2	–
SDS[ ]	0x5F	n	–
SFVFS[ ]	0x0B	y, x	–
SFVTCA[a]	0x04 – 0x05	–	–
SFVTL[a]	0x08 – 0x09	p1, p2	–
SFVTPV[ ]	0x0E	–	–
SHC[a]	0x34 – 0x35	c	–
SHP[a]	0x32 – 0x33	p1, p2, ..., ploop	–
SHPIX[ ]	0x38	p1, p2, amount	–
SHZ[a]	0x36 – 0x37	e	–
SLOOP[ ]	0x17	n	–
SMD[ ]	0x1A	distance	–
SPVFS[ ]	0x0A	y, x	–
SPVTCA[a]	0x02 – 0x03	–	–
SPVTL[a]	0x06 – 0x07	p1, p2	–
SROUND[ ]	0x76	n	–
SRP0[ ]	0x10	p	–
SRP1[ ]	0x11	p	–
SRP2[ ]	0x12	p	–
SSW[ ]	0x1F	n	–
SSWCI[ ]	0x1E	n	–
SUB[ ]	0x61	n1, n2	(n2 – n1)
SVTCA[a]	0x00 – 0x01	–	–
SWAP[ ]	0x23	e1, e2	e1, e2
SZP0[ ]	0x13	n	–
SZP1[ ]	0x14	n	–

SZP2[ ]	0x15	n	–
SZPS[ ]	0x16	n	–
UTP[ ]	0x29	p	–
WCVTF[ ]	0x70	value, location	–
WCVTP[ ]	0x44	value, location	–
WS[ ]	0x42	value, location	–

Appendix C

Instruction Set Index