The Type 2 Charstring Format

Adobe Developer Support

Technical Note #5177

16 March 2000

Adobe Systems Incorporated

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

PN LPS5177

Contents

The Type 2 Charstring Format

1 Introduction

The Type 2 format provides a method for compact encoding of glyph procedures in an outline font program. Type 2 charstrings must be used in a CFF (Compact Font Format) or OpenType font file to create a complete font program.

This document only describes how Type 2 charstrings are encoded, and does not attempt to explain the reasons for choosing various options. Type 2 charstrings are based on Type 1 font concepts, and this document assumes familiarity with the Type 1 font format specification. For more information, please see *Adobe Type 1 Font Format, Version 1.1* (Addison Wesley, 1991). Also, familiarity with the CFF format is assumed; please see Adobe Technical Note #5176, "The Compact Font Format Specification."

Compared to the Type 1 format, the Type 2 encoding offers smaller size and an opportunity for better rendering quality and performance. The Type 2 charstring operators are (with one exception) a superset of the Type 1 operators. With proper conversion programs, valid Type 1 font programs (that is, those compatible with Adobe Type Manager® software) can be converted to Type 2 font programs, and Type 2 programs can be converted to Type 1 font programs, without loss of information or rendering quality.

2 Type 2 Charstrings

The following sections describe the general concepts of encoding a Type 2 charstring.

2.1 Hints

The Type 2 charstring format supports six hint operators: **hstem**, **vstem**, **hstemhm**, **vstemhm**, **hintmask**, and **cntrmask**. The hint information must be declared at the beginning of a charstring (see section 3.1) using the **hstem**, **hstemhm**, **vstem**, and **vstemhm** operators, each of which may each take arguments for multiple stem hints.

Type 2 hint operators aid the rasterizer in recognizing and controlling stems and counter areas within a glyph. A stem generally consists of two positions (edges) and the associated width. *Edge* stem hints help to control character features where there is only a single edge (see section 4.3).

The Type 2 format includes edge hints, which are equivalent to the Type 1 concept of ghost hints (see section on ghost hints, page 57, of "Adobe Type 1 Font Format"). They are used to locate an edge rather than a stem that has two edges. A stem width value of –20 is reserved for a top or right edge, and a value of –21 for a bottom or left edge. The operation of hints with other negative width values is undefined.

hintmask

The **hintmask** operator has the same function as that described in "Changing Hints within a Character," section 8.1, page 69, of "Adobe Type 1 Font Format." It provides a means for activating or deactivating stem hints so that only a set of non-overlapping hints are active at one time.

The **hintmask** operator is followed by one or more data bytes that specify the stem hints which are to be active for the subsequent path construction. The number of data bytes must be exactly the number needed to represent the number of stems in the original stem list (those stems specified by the **hstem**, **vstem**, **hstemhm**, or **vstemhm** commands), using one bit in the data bytes for each stem in the original stem list. Bits with a value of one indicate stems that are active, and a value of zero indicates stems that are inactive.

cntrmask

The **cntrmask** (countermask) hint causes arbitrary but non-overlapping collections of counter spaces in a character to be controlled in a manner similar to how stem widths are controlled by the stem hint commands (see Technical Note #5015, “The Type 1 Font Format Supplement” for more information).

The **cntrmask** operator is followed by one or more data bytes that specify the index number of the stem hints on both sides of a counter space. The number of data bytes must be exactly the number needed to represent the number of stems in the original stem list (those stems specified by the **hstem**, **vstem**, **hstemhm**, or **vstemhm** commands), using one bit in the data bytes for each stem in the original stem list.

For the example shown in Figure 1, the stem list for the glyph would be:

H1 H2 H3 H4 H5 H6 H7 H8 V1 V2 V3 V4 V5

and the following cntrmask commands would be used to control the counter spaces between those stems:

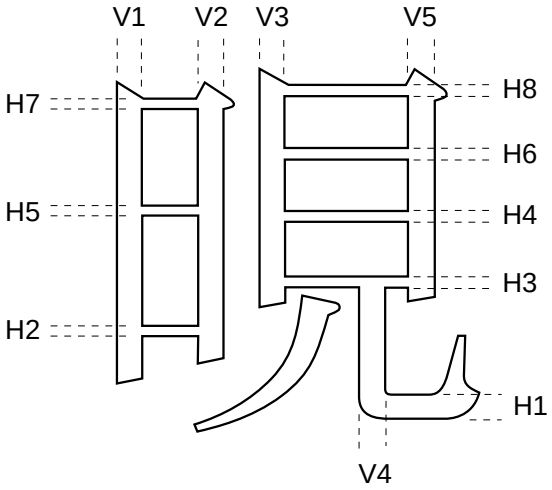
cntrmask 0xB5 0xE8(H1 H3 H4 H6 H8 V1 V2 V3 V5)

cntrmask 0x4A 0x00(H2 H5 H7)

The bits set in the data bytes indicate that the corresponding stem hints delimit the desired set of counters. Hints specified in the first command have a higher priority than those in the second command. Notice that the V4 stem does not delimit an appropriate counter space, and hence is not referenced in this example.

Note that hints are just that, hints, or recommendations. They are additional guidelines to an intelligent rasterizer.

Figure 1 *Counter Control Example*



If the font's **LanguageGroup** is *not* equal to 1 (a **LanguageGroup** value of 1 indicates complex Asian language glyphs), the **cntrmask** operator, with three stems, can be used in place of the **hstem3** and **vstem3** hints in the Type 1 format, as long as the related conditions specified in the Type 1 specification are met.

For more information on Counter Control hints, see Adobe Technical Note #5015, "Type 1 Font Format Supplement."

2.2 The Flex Mechanism

The *flex* mechanism is provided to improve the rendering of shallow curves, representing them as line segments at small sizes rather than as small humps or dents in the character shape. It is essentially a path construction mechanism: the arguments describe the construction of two curves, with an additional argument that is used as a hint for when the curves should be rendered as a straight line at smaller sizes and resolutions.

The Type 2 flex mechanism is general; there are no restrictions on what type or orientation of curve may be expressed with a flex operator. The **flex** operator is used for the general case; special cases can use the **flex1**, **hflex**, or **hflex1** operators for a more efficient encoding. Figure 2, page 19, shows an example of the flex mechanism used for a horizontal curve, and Figure 3,

page 20, shows an example of flex curves at non-standard angles.

The flex operators can be used for any curved character feature, in any orientation or depth, that meets the following requirements:

- The character feature must be capable of being represented as exactly two curves, drawn by two **rrcurveto** operators.
- The curves must meet at a common point called the *joining point*.
- The length of the combined curve must exceed its depth.

2.3 Subroutines

A Type 2 font program can use subroutines to reduce the storage requirements by combining the program statements that describe common elements of the characters in the font.

Subroutines may be *local*, or *global*. Local subroutines are only accessible from the charstring programs in the current font. Global subroutines are those that are shared amongst the various fonts in a *FontSet* (see Adobe Technical Note #5176, "The CFF Font Format Specification" for more information).

Subroutines may contain sections of charstrings, and are encoded the same as Type 2 charstrings. They are called with the **callsubr** (for a local subroutine) or **callgsubr** (for a global subroutine) operator, using a biased index into the local or global Subrs array as the argument.

Note 1 Unlike the biasing in the Type 1 format, in Type 2 the bias is not optional, and is fixed — based on the number of subroutines.

Charstring subroutines may call other subroutines, to the depth allowed by the implementation limits (see Appendix B). A charstring subroutine must end with either an **endchar** or a **return** operator. If the subroutine ends with an **endchar** operator, the **return** is not necessary.

3 Charstring Encoding

A Type 2 charstring program is a sequence of unsigned 8-bit bytes that encode numbers and operators. The byte value specifies a operator, a number, or subsequent bytes that are to be interpreted in a specific manner.

The bytes are decoded into numbers and operators. One reason the format is more economical than Type 1 is because the Type 2 charstring interpreter is required to count the number of arguments on the argument stack. It can thus detect additional sets of arguments for a single operator. The stack depth implementation limit is specified in Appendix B.

A number, decoded from a charstring, is pushed onto the Type 2 argument stack. An operator expects its arguments in order from this argument stack with all arguments generally taken from the bottom of the stack (first argument bottom-most); however, some operators, particularly the subroutine operators, normally work from the top of the stack. If an operator returns results, they are pushed onto the Type 2 argument stack (last result topmost).

In the following discussion, all numeric constants are decimal numbers, except where indicated.

3.1 Type 2 Charstring Organization

The sequence and form of a Type 2 charstring program may be represented as:

w? {hs* vs* cm* hm* mt subpath}? {mt subpath}* endchar

Where:

w = width
hs = **hstem** or **hstemhm** command
vs = **vstem** or **vstemhm** command
cm = **cntrmask** operator
hm = **hintmask** operator
mt = moveto (i.e. any of the moveto) operators

subpath = refers to the construction of a subpath (one complete closed contour), which may include **hintmask** operators where appropriate.

and the following symbols indicate specific usage:

*	zero or more occurrences are allowed
?	zero or one occurrences are allowed
+	one or more occurrences are allowed
{ }	indicates grouping

Stated in words, the constraints on the sequence of operators in a charstring are as follows:

Type 2 charstrings must be structured with operators, or classes of operators, sequenced in the following specific order:

1) **Width**: If the charstring has a width other than that of **defaultWidthX** (see Technical Note #5176, "The Compact Font Format Specification"), it must be specified as the first number in the charstring, and encoded as the difference from **nominalWidthX**.

2) **Hints**: zero or more of each of the following hint operators, in exactly the following order: **hstem**, **hstemhm**, **vstem**, **vstemhm**, **cntrmask**, **hintmask**. Each entry is optional, and each may be expressed by one or more occurrences of the operator. The hint operators **cntrmask** and/or **hintmask** must not occur if the charstring has no stem hints.

3) **Path Construction**: The first path of a charstring that contains no hints must begin with one of the *moveto* operators so that the preceding width can be detected properly.

Zero or more path construction operators are used to draw the path of the character; the second and all subsequent subpaths must also begin with one of the *moveto* operators. The **hintmask** operator may be used as needed.

4) **endchar**: The character must end with an **endchar** operator.

*Note 2 Charstrings may contain subr and gsubr calls as desired at any point between complete tokens (operators or numbers). This means that a subr (gsubr) call must not occur between the bytes of a multibyte commands (for example, **hintmask**).*

3.2 Charstring Number Encoding

A charstring byte containing the values from 32 through 254 inclusive indicates an integer. These values are decoded in three ranges (also see Table 1):

- A charstring byte containing a value, *v*, between 32 and 246 inclusive, specifies the integer $v - 139$. Thus, the integer values from −107 through 107 inclusive may be encoded in a single byte.

- A charstring byte containing a value, *v*, between 247 and 250 inclusive, indicates an integer involving the next byte, *w*, according to the formula:

  $$(v - 247) * 256 + w + 108$$

  Thus, the integer values between 108 and 1131 inclusive can be encoded in 2 bytes in this manner.

- A charstring byte containing a value, *v*, between 251 and 254 inclusive, indicates an integer involving the next byte, *w*, according to the formula:

  $$-[(v - 251) * 256] - w - 108$$

  Thus, the integer values between −1131 and −108 inclusive can be encoded in 2 bytes in this manner.

If the charstring byte contains the value 255, the next four bytes indicate a two's complement signed number. The first of these four bytes contains the highest order bits, the second byte contains the next higher order bits and the fourth byte contains the lowest order bits. This number is interpreted as a Fixed; that is, a signed number with 16 bits of fraction.

Note 3 The Type 2 interpretation of a number encoded in five-bytes (those with an initial byte value of 255) differs from how it is interpreted in the Type 1 format.

In addition to the 32 to 255 range of values, a *ShortInt* value is specified by using the operator (28) followed by two bytes which represent numbers between –32768 and +32767. The most significant byte follows the (28). This allows a more compact representation of large numbers which occur occasionally in fonts, but perhaps more importantly, this will allow more compact encoding of numbers which may be used as arguments to **callsubr** and **callgsubr**.

Table 1 Type 2 Charstring Encoding Values

Charstring Byte Value	Interpretation	Number Range Represented	Bytes Required
0 – 11	operators	operators 0 to 11	1
12	escape: next byte interpreted as additional operators	additional 0 to 255 range for operator codes	2
13 – 18	operators	operators 13 to 18	1
19, 20	operators (**hintmask** and **cntrmask**)	operators 19, 20	2 or more
21 – 27	operators	operators 21 to 27	1
28	following 2 bytes interpreted as a 16-bit two's-complement number	–32768 to +32767	3
29 – 31	operators	operators 29 to 31	1
32 – 246	result = v–139	–107 to +107	1
247 – 250	with next byte, w, result = (v–247)*256+w+108	+108 to +1131	2
251 – 254	with next byte, w, result = –[(v–251)*256]–w–108.	–108 to –1131	2
255	next 4 bytes interpreted as a 32-bit two's-complement number	16-bit signed integer with 16 bits of fraction.	5

3.3 Charstring Operator Encoding

Charstring operators are encoded in one or two bytes.

Single byte operators are encoded in one byte that contains a value between 0 and 31 inclusive, excluding 12 and 28. Not all possible operator encoding values are defined (see Appendix A for a list of operator encoding values). The behavior of undefined operators is unspecified.

If an operator byte contains the value 12, then the value in the next byte specifies an operator. This escape mechanism allows many extra operators to be encoded.

4 Charstring Operators

Type 2 charstring operators are divided into seven groups, classified by function: 1) path construction; 2) finishing a path; 3) hints; 4) arithmetic; 5) storage; 6) conditional; and 7) subroutine.

The following definitions use a format similar to that used in the *PostScript Language Reference Manual*. Parentheses following the operator name either include the operator value that represents this operator in a charstring byte, or the two values (beginning with 12) that represent a two-byte operator.

Many operators take their arguments from the bottom-most entries in the Type 2 argument stack; this behavior is indicated by the stack bottom symbol '|-' appearing to the left of the first argument. Operators that clear the argument stack are indicated by the stack bottom symbol '|-' in the result position of the operator definition.

Because of this stack-clearing behavior, in general, arguments are not accumulated on the Type 2 argument stack for later removal by a sequence of operators, arguments generally may be supplied only for the next operator. Notable exceptions occur with subroutine calls and with arithmetic and conditional operators. All stack operations must observe the stack limit (see Appendix B).

4.1 Path Construction Operators

In a Type 2 charstring, a path is constructed by sequential application of one or more path construction operators. The current point is initially the (0, 0) point of the character coordinate system. The operators listed in this section cause the current point to change, either by a *moveto* operation, or by appending one or more curve or line segments to the current point. Upon completion of the operation, the current point is updated to the position to which the move was made, or to the last point on the segment or segments.

Many of the operators can take multiple sets of arguments, which indicate a series of path construction operations. The number of operations are limited only by the limit on the stack size (see Appendix B).

All Bézier curve path segments are drawn using six arguments, dxa, dya, dxb, dyb, dxc, dyc; where dxa and dya are relative to the current point, and all subsequent arguments are relative to the previous point. A number of the curve operators take advantage of the situation where some tangent points are horizontal or vertical (and hence the value is zero), thus reducing the number of arguments needed.

The flex operators are considered path construction commands because they specify the drawing of two curves. There is also an additional argument that serves as a hint as to when to render the curves as a straight line at small sizes and low resolutions.

The following are three types of moveto operators. For the initial moveto operators in a charstring, the arguments are relative to the (0, 0) point in the character's coordinate system; subsequent moveto operators' arguments are relative to the current point.

Every character path and subpath must begin with one of the moveto operators. If the current path is open when a moveto operator is encountered, the path is closed before performing the moveto operation.

rmoveto |- dx1 dy1 **rmoveto** (21) |-

moves the current point to a position at the relative coordinates (*dx1, dy1)*.

Note 4 *The first stack-clearing operator, which must be one of* ***hstem, hstemhm, vstem, vstemhm, cntrmask, hintmask, hmoveto, vmoveto, rmoveto,*** *or* ***endchar,*** *takes an additional argument — the width (as described earlier), which may be expressed as zero or one numeric argument.*

hmoveto |- dx1 **hmoveto** (22) |-

moves the current point *dx1* units in the horizontal direction. See Note 4.

vmoveto |- dy1 **vmoveto** (4) |-

moves the current point *dy1* units in the vertical direction. See Note 4.

rlineto |- {dxa dya}+ **rlineto** (5) |-

appends a line from the current point to a position at the relative coordinates *dxa, dya*. Additional **rlineto** operations are performed for all subsequent argument pairs. The number of lines is determined from the number of arguments on the stack.

hlineto |- dx1 {dya dxb}* **hlineto** (6) |-

|- {dxa dyb}+ **hlineto** (6) |-

appends a horizontal line of length *dx1* to the current point. With an odd number of arguments, subsequent argument pairs are interpreted as alternating values of *dy* and *dx*, for which additional *lineto* operators draw alternating vertical and horizontal lines. With an even number of arguments, the arguments are interpreted as alternating horizontal and vertical lines. The number of lines is determined from the number of arguments on the stack.

vlineto |- dy1 {dxa dyb}* **vlineto** (7) |-

|- {dya dxb}+ **vlineto** (7) |-

appends a vertical line of length *dy1* to the current point. With

an odd number of arguments, subsequent argument pairs are interpreted as alternating values of *dx* and *dy*, for which additional *lineto* operators draw alternating horizontal and vertical lines. With an even number of arguments, the arguments are interpreted as alternating vertical and horizontal lines. The number of lines is determined from the number of arguments on the stack.

rrcurveto |- {dxa dya dxb dyb dxc dyc}+ **rrcurveto** (8) |-

appends a Bézier curve, defined by *dxa...dyc*, to the current point. For each subsequent set of six arguments, an additional curve is appended to the current point. The number of curve segments is determined from the number of arguments on the number stack and is limited only by the size of the number stack.

hhcurveto |- dy1? {dxa dxb dyb dxc}+ **hhcurveto** (27) |-

appends one or more Bézier curves, as described by the *dxa...dxc* set of arguments, to the current point. For each curve, if there are 4 arguments, the curve starts and ends horizontal. The first curve need not start horizontal (the odd argument case). Note the argument order for the odd argument case.

hvcurveto |- dx1 dx2 dy2 dy3 {dya dxb dyb dxc dxd dxe dye dyf}* dxf? **hvcurveto** (31) |-

|- {dxa dxb dyb dyc dyd dxe dye dxf}+ dyf? **hvcurveto** (31) |-

appends one or more Bézier curves to the current point. The tangent for the first Bézier must be horizontal, and the second must be vertical (except as noted below).

If there is a multiple of four arguments, the curve starts horizontal and ends vertical. Note that the curves alternate between *start horizontal*, *end vertical*, and *start vertical*, and *end horizontal*. The last curve (the odd argument case) need not end horizontal/vertical.

rcurveline |- {dxa dya dxb dyb dxc dyc}+ dxd dyd **rcurveline** (24) |-

is equivalent to one **rrcurveto** for each set of six arguments

dxa...dyc, followed by exactly one **rlineto** using the *dxd*, *dyd* arguments. The number of curves is determined from the count on the argument stack.

rlinecurve |- {dxa dya}+ dxb dyb dxc dyc dxd dyd **rlinecurve** (25) |-

is equivalent to one **rlineto** for each pair of arguments beyond the six arguments *dxb...dyd* needed for the one **rrcurveto** command. The number of lines is determined from the count of items on the argument stack.

vhcurveto |- dy1 dx2 dy2 dx3 {dxa dxb dyb dyc dyd dxe dye dxf}* dyf? **vhcurveto** (30) |-

|- {dya dxb dyb dxc dxd dxe dye dyf}+ dxf? **vhcurveto** (30) |-

appends one or more Bézier curves to the current point, where the first tangent is vertical and the second tangent is horizontal.

This command is the complement of **hvcurveto**; see the description of **hvcurveto** for more information.

vvcurveto |- dx1? {dya dxb dyb dyc}+ **vvcurveto** (26) |-

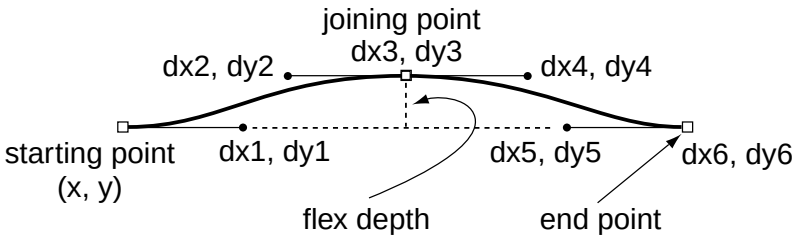
appends one or more curves to the current point. If the argument count is a multiple of four, the curve starts and ends vertical. If the argument count is odd, the first curve does not begin with a vertical tangent.

flex |- dx1 dy1 dx2 dy2 dx3 dy3 dx4 dy4 dx5 dy5 dx6 dy6 fd **flex** (12 35) |-

causes two Bézier curves, as described by the arguments (as shown in Figure 2 below), to be rendered as a straight line when the *flex depth* is less than fd/100 device pixels, and as curved lines when the flex depth is greater than or equal to fd/100 device pixels.

The flex depth for a horizontal curve, as shown in Figure 2, is the distance from the join point to the line connecting the start and end points on the curve. If the curve is not exactly horizontal or vertical, it must be determined whether the curve is more horizontal or vertical by the method described in the **flex1** description, below, and as illustrated in Figure 3.

Figure 2 *Flex Hint Example*



Note 5 *In cases where some of the points have the same x or y coordinate as other points in the curves, arguments may be omitted by using one of the following forms of the flex operator,* ***hflex, hflex1****, or* ***flex1****.*

hflex |- dx1 dx2 dy2 dx3 dx4 dx5 dx6 **hflex** (12 34) |-

causes the two curves described by the arguments *dx1*...*dx6* to be rendered as a straight line when the flex depth is less than 0.5 (that is, fd is 50) device pixels, and as curved lines when the flex depth is greater than or equal to 0.5 device pixels.

hflex is used when the following are all true:

a) the starting and ending points, first and last control points have the same y value.

b) the joining point and the neighbor control points have the same y value.

c) the flex depth is 50.

hflex1 |- dx1 dy1 dx2 dy2 dx3 dx4 dx5 dy5 dx6 **hflex1** (12 36) |-

causes the two curves described by the arguments to be rendered as a straight line when the flex depth is less than 0.5 device pixels, and as curved lines when the flex depth is greater than or equal to 0.5 device pixels.

hflex1 is used if the conditions for **hflex** are not met but all of the following are true:

a) the starting and ending points have the same y value,

b) the joining point and the neighbor control points have the same y value.

c) the flex depth is 50.

flex1 |- dx1 dy1 dx2 dy2 dx3 dy3 dx4 dy4 dx5 dy5 d6 **flex1** (12 37) |-

causes the two curves described by the arguments to be rendered as a straight line when the flex depth is less than 0.5 device pixels, and as curved lines when the flex depth is greater than or equal to 0.5 device pixels.

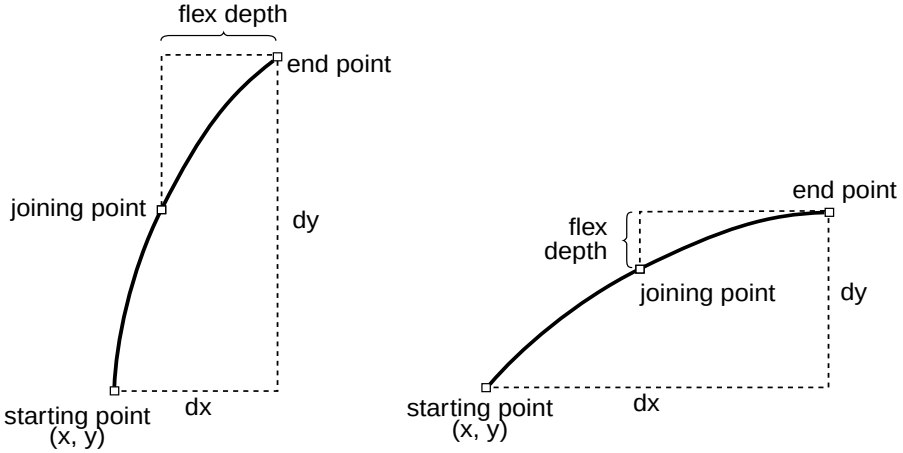
The d6 argument will be either a dx or dy value, depending on the curve (see Figure 3). To determine the correct value, compute the distance from the starting point (x, y), the first point of the first curve, to the last flex control point (dx5, dy5) by summing all the arguments except d6; call this (dx, dy). If abs(dx) > abs(dy), then the last point's x-value is given by d6, and its y-value is equal to *y*. Otherwise, the last point's x-value is equal to *x* and its y-value is given by d6.

flex1 is used if the conditions for **hflex** and **hflex1** are notmet but all of the following are true:

a) the starting and ending points have the same x or y value,

b) the flex depth is 50.

Figure 3 *Flex Depth Calculations*



4.2 Operator for Finishing a Path

endchar – **endchar** (14) |–

finishes a charstring outline definition, and must be the last operator in a character's outline.

Note 6 *The charstring itself may end with a **call(g)subr**; the subroutine must then end with an **endchar** operator.*

Note 7 *A character that does not have a path (e.g. a space character) may consist of an **endchar** operator preceded only by a width value. Although the width must be specified in the font, it may be specified as the **defaultWidthX** in the CFF data, in which case it should not be specified in the charstring. Also, it may appear in the charstring as the difference from **nominalWidthX**. Thus the smallest legal charstring consists of a single **endchar** operator.*

Note 8 ***endchar** also has a deprecated function; see Appendix C, "Comaptibility and Deprecated Operators."*

4.3 Hint Operators

All hints must be declared at the beginning of the charstring program, after the width (see section 3.1 for details).

hstem |- y dy {dya dyb}* **hstem** (1) |-

specifies one or more horizontal stem hints (see the following section for more information about horizontal stem hints). This allows multiple pairs of numbers, limited by the stack depth, to be used as arguments to a single **hstem** operator.

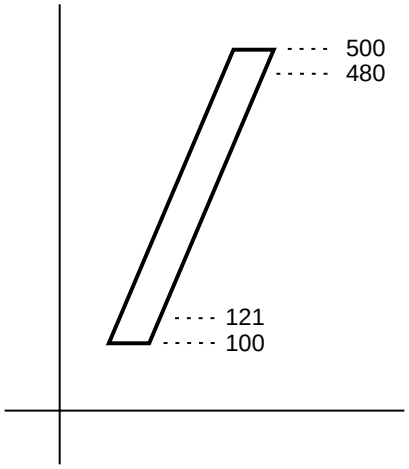
It is required that the stems are encoded in ascending order (defined by increasing bottom edge). The encoded values are all relative; in the first pair, y is relative to 0, and dy specifies the distance from y. The first value of each subsequent pair is relative to the last edge defined by the previous pair.

A width of –20 specifies the top edge of an edge hint, and –21 specifies the bottom edge of an edge hint. All other negative widths have undefined meaning.

Figure 4 shows an example of the encoding of a character stem that uses top and bottom edge hints. The edge stem hint serves

to control the position of the edge of the stem in situations where controlling the stem width is not the primary purpose.

Figure 4 *Encoding of Edge Hints*



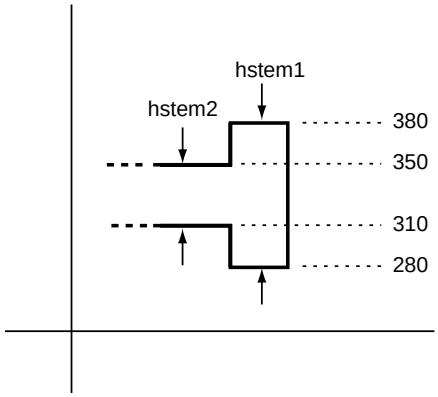
The encoding for the edge stem hints shown in Figure 4 would be:

121 –21 400 –20 **hstem**

Figure 5 shows an example of overlapping hints on a sample feature of a character outline. The overlapping hints must be resolved by using two **hintmask** operators so that they are not both active at the same time.

Horizontal stem hints must not overlap each other. If there is any overlap, the **hintmask** operator must be used immediately after the hint declarations to establish the desired non-overlapping set of hints. **hintmask** may be used again later in the path to activate a different set of non-overlapping hints.

Figure 5 *Encoding of Overlapping Hints*



The encoding for the example shown in Figure 5 would be:

280 100 –70 40 **hstem**

vstem |- x dx {dxa dxb}* **vstem** (3) |-

specifies one or more vertical stem hints between the *x* coordinates *x* and *x*+*dx*, where *x* is relative to the origin of the coordinate axes.

It is required that the stems are encoded in ascending order (defined by increasing left edge). The encoded values are all relative; in the first pair, x is relative to 0, and dx specifies the distance from x. The first value of each subsequent pair is relative to the last edge defined by the previous pair.

A width of –20 specifies the right edge of an edge hint, and –21 specifies the left edge of an edge hint. All other negative widths have undefined meaning.

Vertical stem hints must not overlap each other. If there is any overlap, the **hintmask** operator must be used immediately after the hint declarations to establish the desired non-overlapping set of hints. **hintmask** may be used again later in the path to activate a different set of non-overlapping hints.

hstemhm |- y dy {dya dyb}* **hstemhm** (18) |-

has the same meaning as **hstem** (1), except that it must be used in place of **hstem** if the charstring contains one or more **hintmask** operators.

vstemhm |- x dx {dxa dxb}* **vstemhm** (23) |-

has the same meaning as **vstem** (3), except that it must be used in place of **vstem** if the charstring contains one or more **hintmask** operators.

hintmask |- **hintmask** (19 + mask) |-

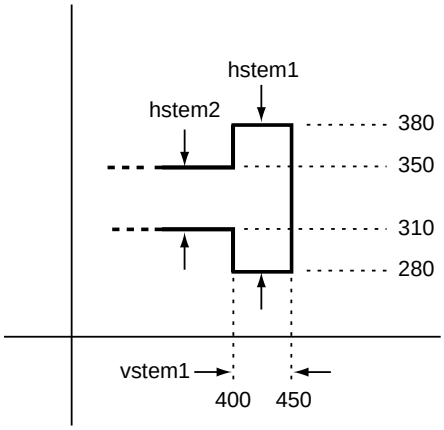
specifies which hints are active and which are not active. If any hints overlap, **hintmask** must be used to establish a non-overlapping subset of hints. **hintmask** may occur any number of times in a charstring. Path operators occurring after a **hintmask** are influenced by the new hint set, but the current point is not moved. If stem hint zones overlap and are not properly managed by use of the **hintmask** operator, the results are undefined.

The *mask* data bytes are defined as follows:

- The number of data bytes is exactly the number needed, one bit per hint, to reference the number of stem hints declared at the beginning of the charstring program.
- Each bit of the mask, starting with the most-significant bit of the first byte, represents the corresponding hint zone in the order in which the hints were declared at the beginning of the charstring.
- For each bit in the mask, a value of '1' specifies that the corresponding hint shall be active. A bit value of '0' specifies that the hint shall be inactive.
- Unused bits in the mask, if any, must be zero.

If **hstem** and **vstem** hints are both declared at the beginning of a charstring, and this sequence is followed directly by the **hintmask** or **cntrmask** operators, the **vstem** hint operator need not be included. For example, Figure 6 shows part of a character with **hstem** and **vstem** hints:

Figure 6 *Hint Encoding Example*



If the first hint group is to be the **hstem** from 280 to 380 and the **vstem** from 400 to 450 (and only these three hints are defined), then the hints would be specified as:

280 100 –70 40 **hstemhm** 400 50 **hintmask** 0xa0

where the hex data 0xa0 (10100000) indicates which hints are active at the beginning of the path construction.

Note that the **hstemhm** is used to indicate that hint substitution is used.

cntrmask |- **cntrmask** (20 + mask) |-

specifies the counter spaces to be controlled, and their relative priority. The *mask* bits in the bytes, following the operator, reference the stem hint declarations; the most significant bit of the first byte refers to the first stem hint declared, through to the last hint declaration. The counters to be controlled are those that are delimited by the referenced stem hints. Bits set to 1 in the first **cntrmask** command have top priority; subsequent **cntrmask** commands specify lower priority counters (see Figure 1 and the accompanying example).

4.4 Arithmetic Operators

abs num **abs** (12 9) num2

returns the absolute value of *num*.

add num1 num2 **add** (12 10) sum

returns the sum of the two numbers *num1* and *num2*.

sub num1 num2 **sub** (12 11) difference

returns the result of subtracting *num2* from *num1*.

div num1 num2 **div** (12 12) quotient

returns the quotient of *num1* divided by *num2*. The result is undefined if overflow occurs and is zero for underflow.

neg num **neg** (12 14) num2

returns the negative of *num.*

random **random** (12 23) num2

returns a pseudo random number *num2* in the range (0,1], that is, greater than zero and less than or equal to one.

mul num1 num2 **mul** (12 24) product

returns the product of *num1* and *num2*. If overflow occurs, the result is undefined, and zero is returned for underflow.

sqrt num **sqrt** (12 26) num2

returns the square root of *num*. If num is negative, the result is undefined.

drop num **drop** (12 18)

removes the top element *num* from the Type 2 argument stack.

exch num1 num2 **exch** (12 28) num2 num1

exchanges the top two elements on the argument stack.

index numX ... num0 i **index** (12 29) numX ... num0 numi

retrieves the element *i* from the top of the argument stack and pushes a copy of that element onto that stack. If *i* is negative, the top element is copied. If *i* is greater than X, the operation is undefined.

roll num(N–1) ... num0 N J **roll** (12 30) num((J–1) mod N) ... num0 num(N–1) ... num(J mod N)

performs a circular shift of the elements *num(N–1)* ... *num0* on the argument stack by the amount *J*. Positive *J* indicates upward motion of the stack; negative *J* indicates downward motion. The value *N* must be a non-negative integer, otherwise the operation is undefined.

dup any **dup** (12 27) any any

duplicates the top element on the argument stack.

4.5 Storage Operators

The storage operators utilize a transient array and provide facilities for storing and retrieving transient array data.

The transient array provides non-persistent storage for intermediate values. There is no provision to initialize this array, except explicitly using the **put** operator, and values stored in the array do not persist beyond the scope of rendering an individual character.

The number of elements in the transient array is specified in Appendix B, “Type 2 Charstring Implementation Limits”.

put val i **put** (12 20)

stores *val* into the transient array at the location given by *i*.

get i **get** (12 21) val

retrieves the value stored in the transient array at the location given by *i* and pushes the value onto the argument stack. If **get**

is executed prior to **put** for *i* during execution of the current charstring, the value returned is undefined.

4.6 Conditional Operators

and num1 num2 **and** (12 3) 1_or_0

puts a 1 on the stack if *num1* and *num2* are both non-zero, and puts a 0 on the stack if either argument is zero.

or num1 num2 **or** (12 4) 1_or_0

puts a 1 on the stack if either *num1* or *num2* are non-zero, and puts a 0 on the stack if both arguments are zero.

not num1 **not** (12 5) 1_or_0

returns a 0 if *num1* is non-zero; returns a 1 if *num1* is zero.

eq num1 num2 **eq** (12 15) 1_or_0

puts a 1 on the stack if *num1* equals *num2*, otherwise a 0 (zero) is put on the stack.

ifelse s1 s2 v1 v2 **ifelse** (12 22) s1_or_s2

leaves the value *s1* on the stack if *v1* ≤ *v2*, or leaves *s2* on the stack if *v1* > *v2*. The value of *s1* and s2 is usually the biased number of a subroutine; see section 2.3.

4.7 Subroutine Operators

The numbering of subroutines is encoded more compactly by using the negative half of the number space, which effectively doubles the number of compactly encodable subroutine numbers. The bias applied depends on the number of subrs (gsubrs). If the number of subrs (gsubrs) is less than 1240, the bias is 107. Otherwise if it is less than 33900, it is 1131; otherwise it is 32768. This bias is added to the encoded subr (gsubr) number to find the appropriate entry in the subr (gsubr) array. Global subroutines may be used in a FontSet even if it only

contains one font.

callsubr subr# **callsubr** (10) –

calls a charstring subroutine with index *subr# (actually the subr number plus the subroutine bias number, as described in section 2.3)* in the Subrs array. Each element of the Subrs array is a charstring encoded like any other charstring. Arguments pushed on the Type 2 argument stack prior to calling the subroutine, and results pushed on this stack by the subroutine, act according to the manner in which the subroutine is coded. Calling an undefined subr (gsubr) has undefined results.

These subroutines are generally used to encode sequences of path operators that are repeated throughout the font program, for example, serif outline sequences. Subroutine calls may be nested to the depth specified in the implementation limits in Appendix B.

callgsubr globalsubr# **callgsubr** (29) –

operates in the same manner as **callsubr** except that it calls a global subroutine.

return – **return** (11) –

returns from either a local or global charstring subroutine, and continues execution after the corresponding **call(g)subr**.

Appendix A
Type 2 Charstring Command Codes

One-byte Type 2 Operators

Dec	Hex	Operator	Dec	Hex	Operator
0	00	–Reserved–	18	12	hstemhm
1	01	hstem	19	13	hintmask
2	02	–Reserved–	20	14	cntrmask
3	03	vstem	21	15	rmoveto
4	04	vmoveto	22	16	hmoveto
5	05	rlineto	23	17	vstemhm
6	06	hlineto	24	18	rcurveline
7	07	vlineto	25	19	rlinecurve
8	08	rrcurveto	26	1a	vvcurveto
9	09	–Reserved–	27	1b	hhcurveto
10	0a	callsubr	28[2]	1c	shortint
11	0b	return	29	1d	callgsubr
12[1]	0c	escape	30	1e	vhcurveto
13	0d	–Reserved–	31	1f	hvcurveto
14	0e	endchar	32–246	20–f6	<numbers>
15	0f	–Reserved–	247–254[3]	f7–fe	<numbers>
16	10	–Reserved–	255[4]	ff	<number>
17	11	–Reserved–			

1. First byte of a 2-byte operator.
2. First byte of a 3-byte sequence specifying a number.
3. First byte of a 2-byte sequence specifying a number.
4. First byte of a 5-byte sequence specifying a number.

Two-byte Type 2 Operators

Dec	Hex	Operator	Dec	Hex	Operator
12 0	0c 00	–Reserved– [1]	12 20	0c 14	put
12 1	0c 01	–Reserved–	12 21	0c 15	get
12 2	0c 02	–Reserved–	12 22	0c 16	ifelse
12 3	0c 03	and	12 23	0c 17	random
12 4	0c 04	or	12 24	0c 18	mul
12 5	0c 05	not	12 25	0c 19	–Reserved–
12 6	0c 06	–Reserved–	12 26	0c 1a	sqrt
12 7	0c 07	–Reserved–	12 27	0c 1b	dup
12 8	0c 08	–Reserved–	12 28	0c 1c	exch
12 9	0c 09	abs	12 29	0c 1d	index
12 10	0c 0a	add	12 30	0c 1e	roll
12 11	0c 0b	sub	12 31	0c 1f	–Reserved–
12 12	0c 0c	div	12 32	0c 20	–Reserved–
12 13	0c 0d	–Reserved–	12 33	0c 21	–Reserved–
12 14	0c 0e	neg	12 34	0c 22	hflex
12 15	0c 0f	eq	12 35	0c 23	flex
12 16	0c 10	–Reserved–	12 36	0c 24	hflex1
12 17	0c 11	–Reserved–	12 37	0c 25	flex1
12 18	0c 12	drop	12 38–12 255	0c 26–0c ff	–Reserved–
12 19	0c 13	–Reserved–			

1. 12 0 dotsection is deprecated, see Appendix C “Compatibility and Deprecated Operators.”

Appendix B
Type 2 Charstring Implementation Limits

The following are the implementation limits of the Type 2 charstring interpreter:

Description	Limit
Argument stack	48
Number of stem hints (H/V total)	96
Subr nesting, stack limit	10
Charstring length	65535
maximum (g)subrs count	65536
TransientArray elements	32

Appendix C Compatibility and Deprecated Operators

The following constructs may appear in the CFF encoded font programs in Portable Document Format (PDF) documents. Since that usage predates OpenType fonts (OTF), the following obsolete and deprecated operators should be supported in all Type 2 charstring processors that may encounter such programs.

dotsection

|– **dotsection** (12 0) |–

This is an obsolete form of hint substitution (actually hint suspension) that has always been treated as a no-op by Adobe ATM renderers.

endchar

– adx ady bchar achar **endchar** (14) |–

In addition to the optional width (see section 4.2, “Operator for Finishing a Path” for more details) **endchar** may have four extra arguments that correspond exactly to the last four arguments of the Type 1 charstring command “seac” (see Type 1 Font Format book). The Type 1 charstring command argument *asb* is not included because all sidebearings are considered to be zero and hence unencoded in Type 2 charstrings.

It is important to note the following restrictions which are the same as those for Type 1 but frequently overlooked.

The *bchar* and *achar* refer to glyph names in StandardEncoding and not to any current font encoding or re-

encoding. This requires that a glyph name be determined from *bchar* and *achar* via StandardEncoding and then the appropriate charstring be located by that name.

This construct can only be used to build glyphs from components named in StandardEncoding. This construct may not be nested.

Appendix D
Changes Since Earlier Versions

The following changes and revisions have been made since the initial publication date of 18 November 1996.

Changes in the 16 March 2000 document

- Section 4.1, Path Construction Operators: Note 4, under **rmoveto**, removed **closepath** and added **endchar** as one of the stack-clearing operators that can take an additional argument.

- Section 4.3, *Hint Operators*: **hintmask** – **cntrmask** was added to **hintmask** as an operator that, if following both **hstem** and **vstem** hints declared at the beginning of a charstring, then the **vstem** hint operator need not be included.

- The information on the **blend** operator, and all references to multiple master fonts, were removed.

- Section 4.5, *Storage Operators*: the **store** and **load** commands were removed.

- Appendix A: Type 2 *Charstring Command Codes*: the codes for **blend** (16), **store** (12 8) and **load** (12 13) were changed to "Reserved."

- Appendix B: *Type 2 Charstring Implementation Limits*: The TransientArray limit of 32 was added.

- A new Appendix C, *Compatibility and Deprecated Operators* was added, and the previous Appendix C, *Changes Since Earlier Versions*, was changed to be Appendix D.

Changes in the 5 May 1998 document

- Section 4.1, *Path Construction Operators*: the stack bottom symbol "|-" was added to the beginning of the operator definition for the **flex**, **hflex**, **hflex1**, and **flex1** operators.

- Section 4.1, the "See Note 2" for **hmoveto** and **vmoveto** was changed to correctly refer to Note 4.

- Section 4.1, for **flex1**, second paragraph, the reference to "dx6" was changed to "d6".

- Section 4.3, *Hint Operators:* the stack bottom symbol "|-" was added to the beginning of the operator definition for **hintmask** and **countermask**.

Changes in the 18 March 1998 document

- Section 4.1, *Path Construction Operators*: the stack bottom symbol "|-" was added to the result position of operator definition for the **flex, hflex, hflex1,** and **flex1** operators.

- Section 4.5, *Storage Operators*: the absolute maximum number of elements allowed for the Weight Vector, Normalized Design Vector, and User Design vector were added. It was also stated that accessing an element beyond either the absolute maximum number allowed or the actual range yields undefined results.

- Appendix A, *Type 2 Charstring Command Codes*, for the two-byte codes, the code for '12 17' was corrected from "pop" to "Reserved."

Changes in the 15 October 1997 document

- Section 4.1, *Path Construction Operators*: Note 4 was updated to indicate that it applies to the first *stack-clearing* operator only. Also, **closepath** was added as one of the stack clearing operators that can take an additional argument.

- Section 4.3, *Hint Operators*: the stack bottom symbol "|-" was added to the result position of operator definition for the **cntrmask** and **hintmask** operators to indicate that those operators leave the stack cleared.

- Section 4.5, *Storage Operators*: the transient array length is now defined by the lenBuildCharArray argument to the **MultipleMaster** operator in the CFF data.

Changes in the 16 December 1996 document

A variety of minor changes were made to clarify existing text; the technical content was not affected.