Blank

Managing outlines

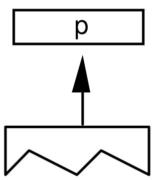
The following set of instructions make it possible to move the points that make up a glyph outline. They are the instructions that accomplish the actual work of grid-fitting. They include instructions to move points, shift points or groups of points, flip points from off to on the curve or vice versa, and to interpolate points.

FLIP PoinT

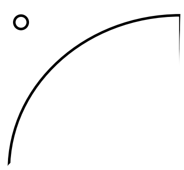
FLIPPT[ ]

Code Range	0x80
Pops	p: point number (ULONG)
Pushes	–
Uses	loop, p is referenced in zp0

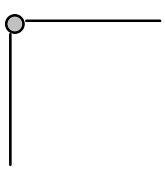
Flips points that are *off* the curve so that they are *on* the curve and points that are *on* the curve so that they are *off* the curve. The point is not marked as touched. The result of a FLIPPT instruction is that the contour describing part of a glyph outline is redefined.



Before:



After



FLIP RanGe ON

FLIPRGON[ ]

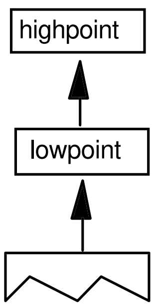
Code Range 0x81

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)

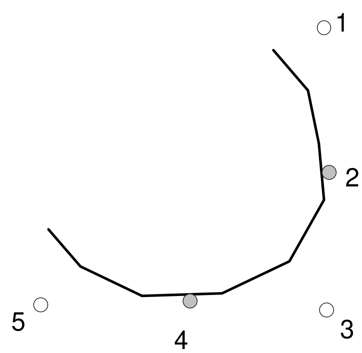
Pushes –

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.

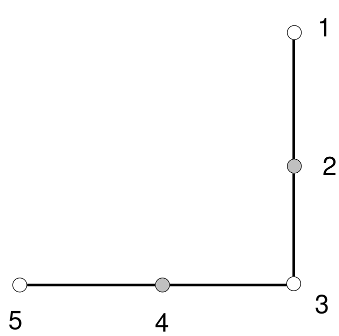


Example:

FLIPRGON[ ] 1 5



Before:



After:

Will make all off curve points between point 0 and point 5 into on curve points as shown

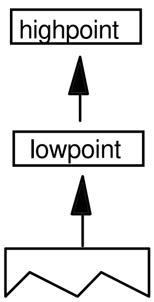
FLIP RanGe OFF

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.

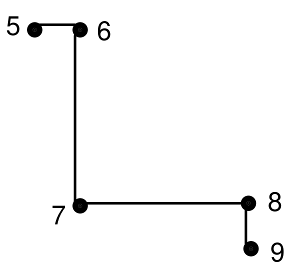
NOTE: This instruction changes the curve but the position of the points is unaffected. Accordingly, points affected by this instruction are not marked as touched.



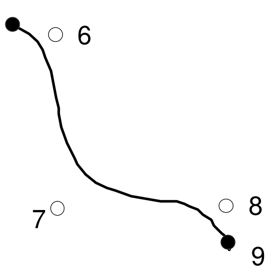
Example:

FLIPRGOFF[ ] 8 6

Before :



After:



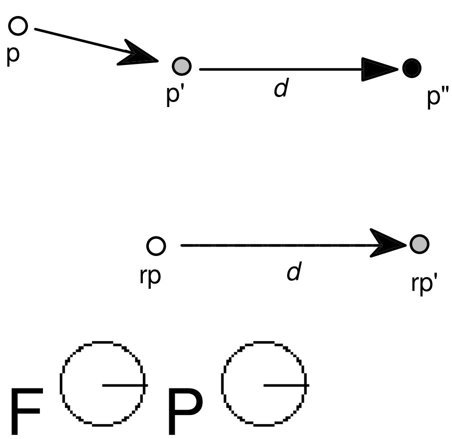
SHift Point by the last point

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

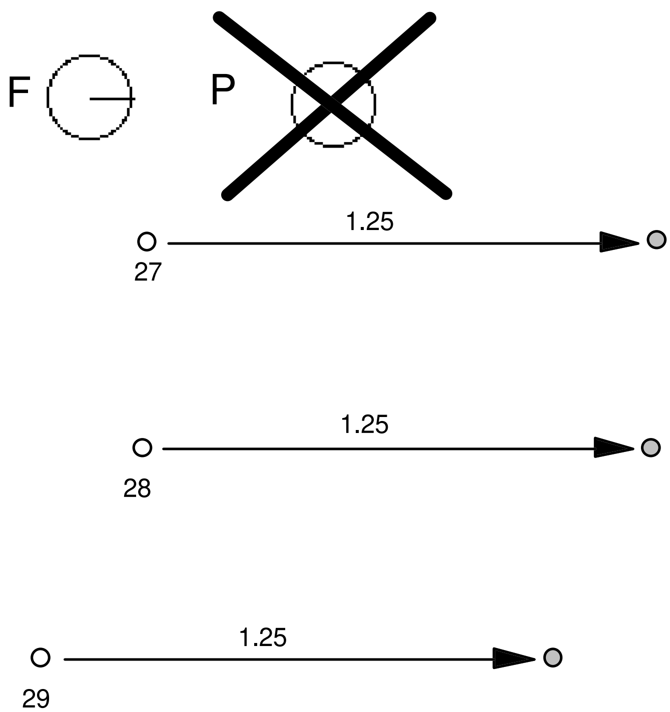
Shifts the points specified by the amount stated. When the loop variable is used, the amount to be shifted is put onto the stack only once. That is, if loop = 3, then the contents of the top of the stack should be point p1, point p2, point p3, amount. The value amount is expressed in sixty-fourths of 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[]

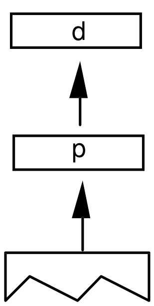


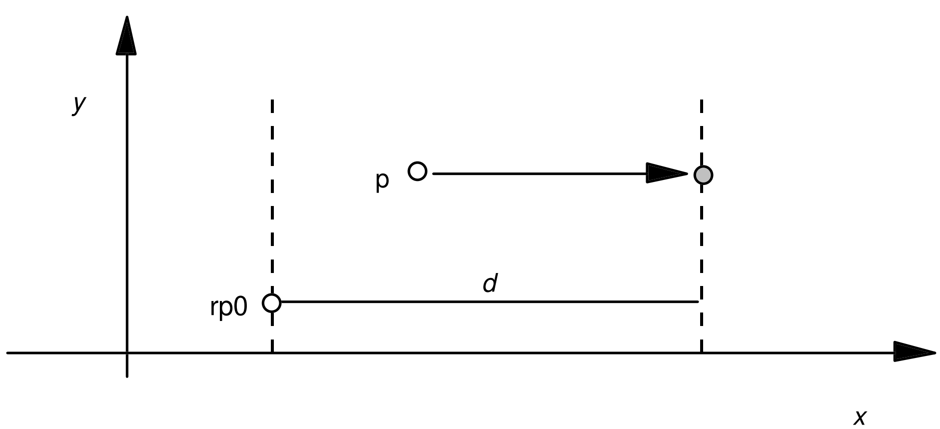
Move Stack Indirect Relative Point

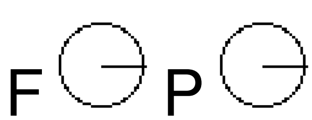
MSIRP[a]

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.



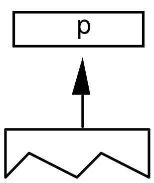




Move Direct Absolute Point

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.

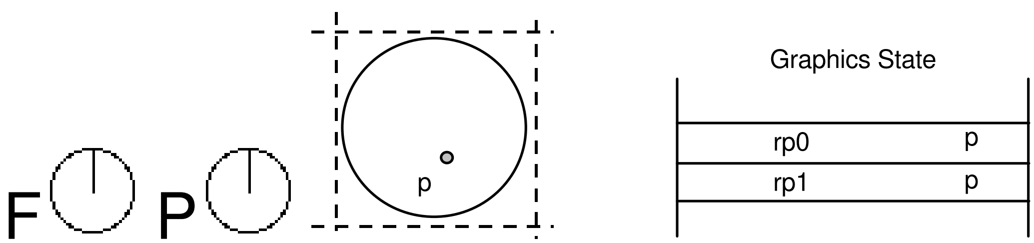
Sets the reference points rp0 and rp1 equal to point p. If a=1, this instruction rounds point p to the grid point specified by the state variable round_state. If a=0, it simply marks the point as touched in the direction(s) specified by the current freedom_vector. This command is often used to set points in the twilight zone.



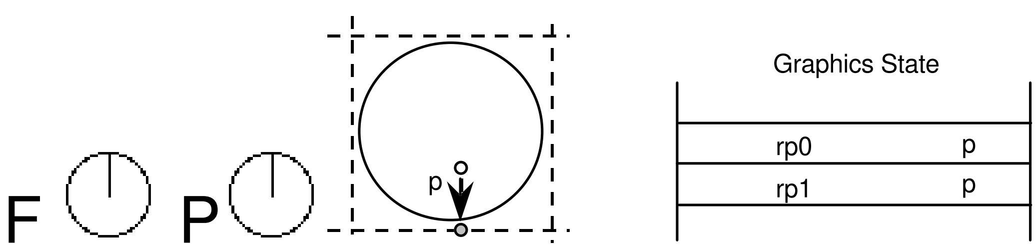
Example:

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.

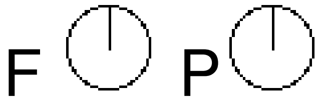
The a Boolean above controls both rounding and the use of the control_value_cut_in. If you would like the meaning of this Boolean to specify only whether or not the MIAP[ ] instruction should look at the control_value_cut_in value, use the ROFF[ ] instruction to turn off rounding.

This instruction can be used to create Twilight Zone points.

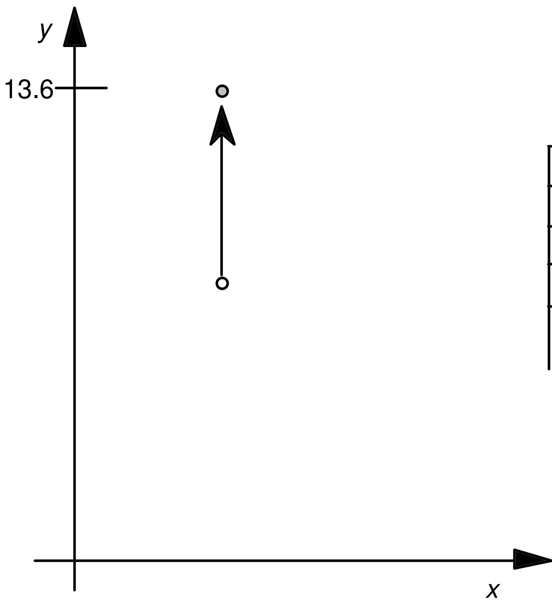
Example:

MIAP[1] 4 7

case 1:
rounding is OFF



The point is moved to the position specified in the CVT.

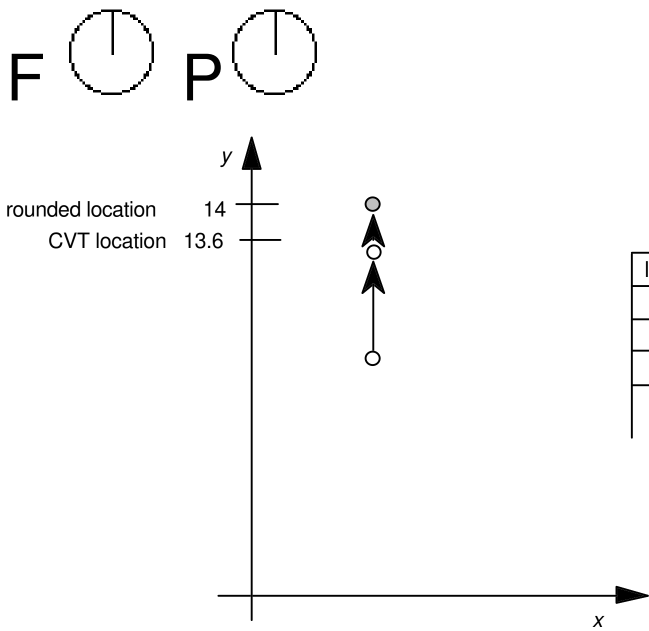


location	value
0	...
...	...
7	13.6

(continued...)

case 2:
The cut_in test succeeds and rounding is RTG.

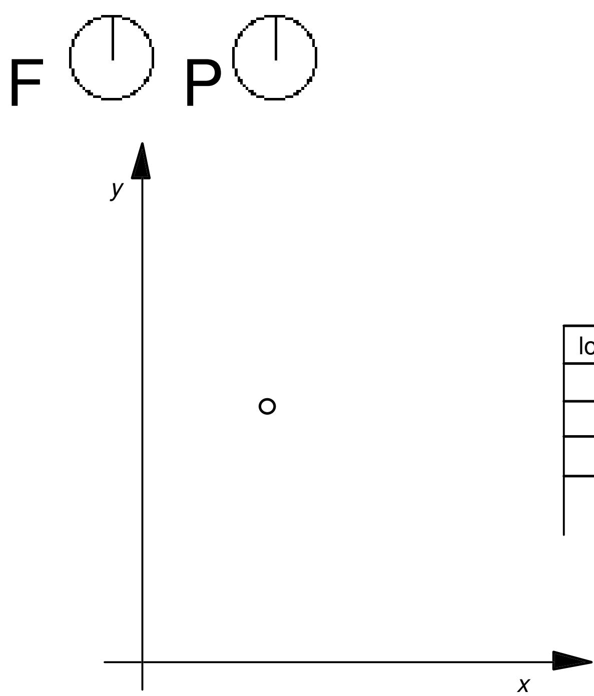
The value in the CVT is subjected to the rounding rule and then the point is moved to the rounded position.



location	value
0	...
...	...
7	13.6

case 3:
The cut_in test fails and rounding is OFF.

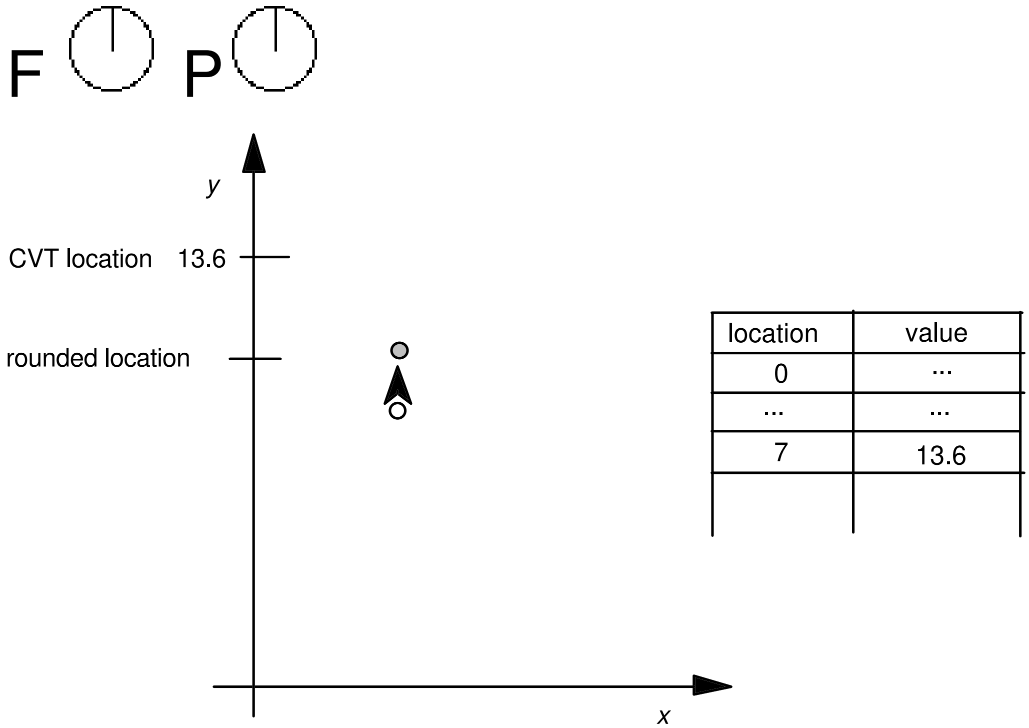
Here the point is not moved.



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.



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.

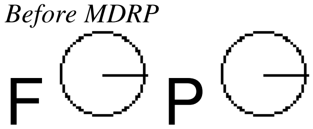
The setting of the round_state Graphics State variable will determine whether and how the distance of point p from point q is rounded. If the round bit is not set, the value will be unrounded. If the round bit is set, the effect will depend upon the choice of rounding state. The value of the minimum 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:

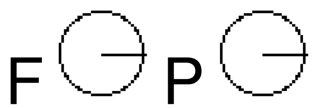
Graphics State Settings

Before MDRP



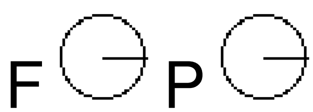
rp0 7
rp1 ?
rp2 ?

Case 1:
After MDRP[00001] 8



rp0 7
rp1 7
rp2 8

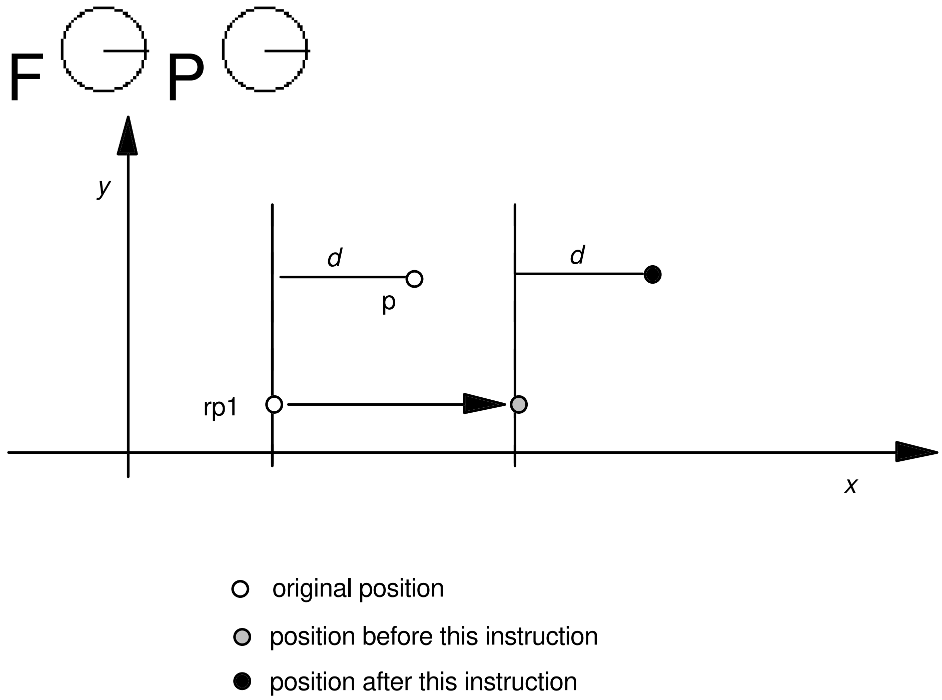
Case 2:
After MDRP[10001] 8



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.



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.

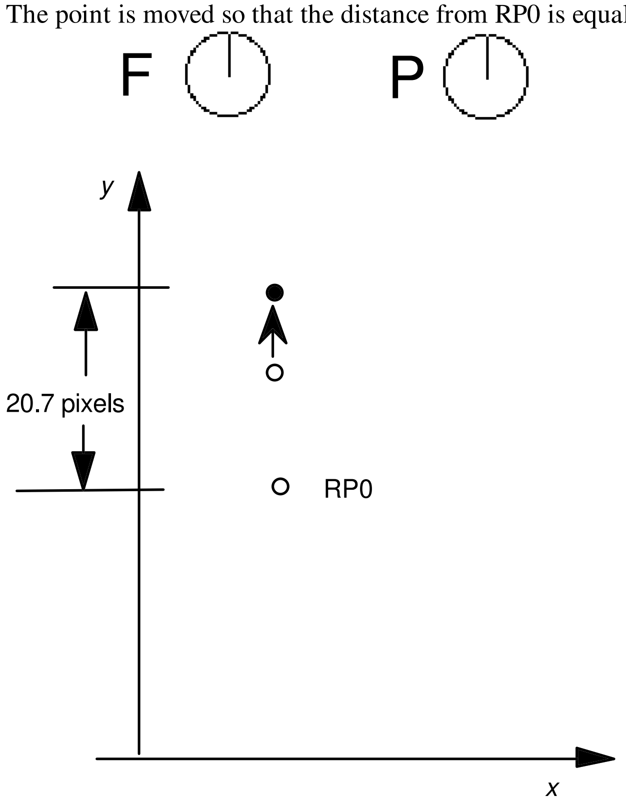
The value de specifies the distance type as described in th chapter, “Instructing Glyphs.” Three values are possible: Gray=0, Black=1, White=2.

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.

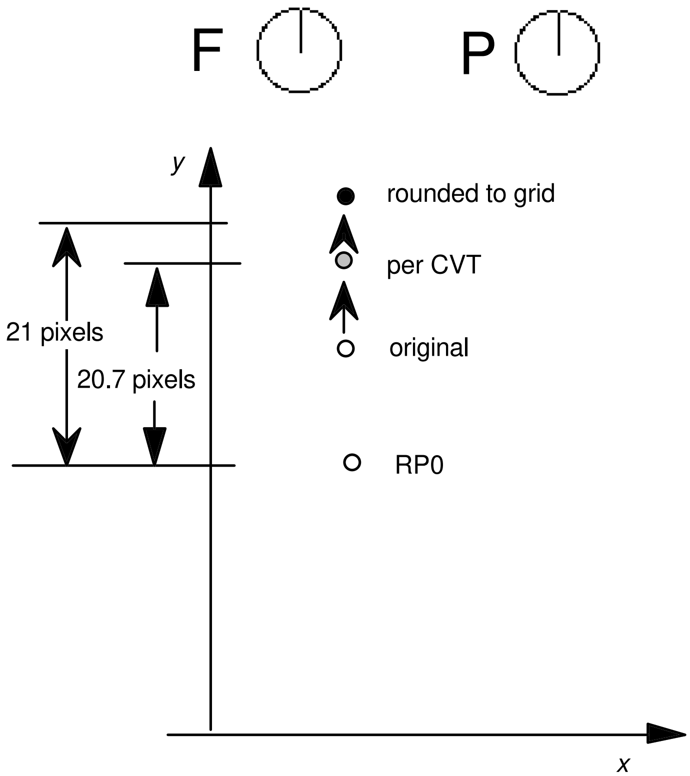


location	value
0	...
...	...
17	20.7

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.

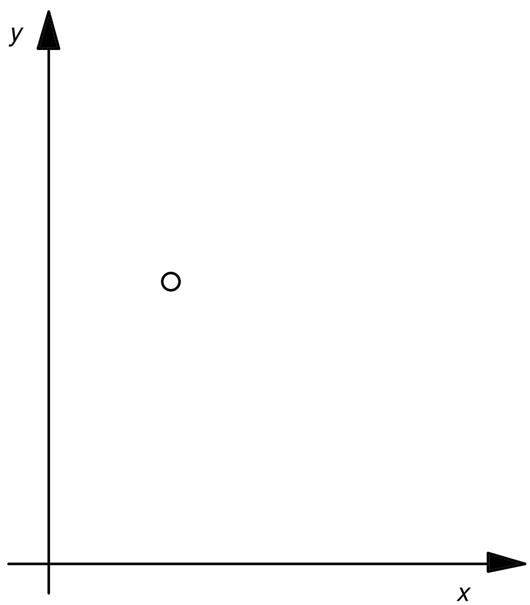


location	value
0	...
...	...
17	20.7

case 3:

The cut_in test fails and the round_state is OFF.

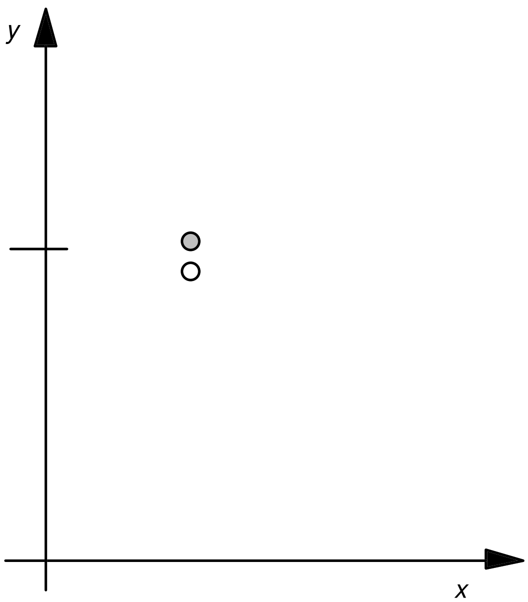
The point is not moved.



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.



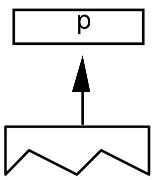
location	value
0	...
...	...
17	13.6

ALIGN Relative Point

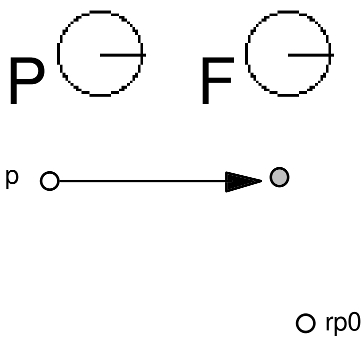
ALIGNRP[ ]

Code Range	0x3C
Pops	p: point number (ULONG)
Pushes	–
Uses	zp1 with point p, zp0 with rp0, loop, freedom_vector, projection_vector.

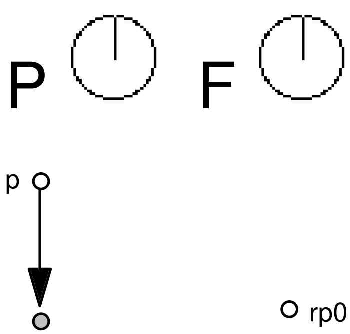
Reduces the distance between rp0 and point p to zero. Since distance is measured along the projection_vector and movement is along the freedom_vector, the effect of the instruction is to align points.



case 1:



case 2:



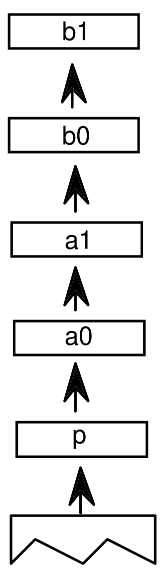
Adjust Angle (No Longer Supported)

moves point p to the InterSECTion of two lines

ISECT[ ]

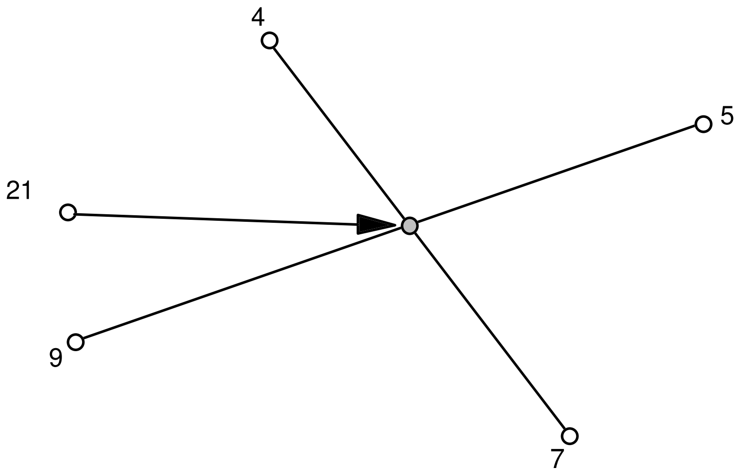
Code Range	0x0F
Pops	b1: end point of line 2 (ULONG) b0: start point of line 2 (ULONG) a1: end point of line 1 (ULONG) a0: start point of line 1 (ULONG) p: point to move (ULONG)
Pushes	–
Uses	zp2 with point p, zp1 with line A, zp0 with line B

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:

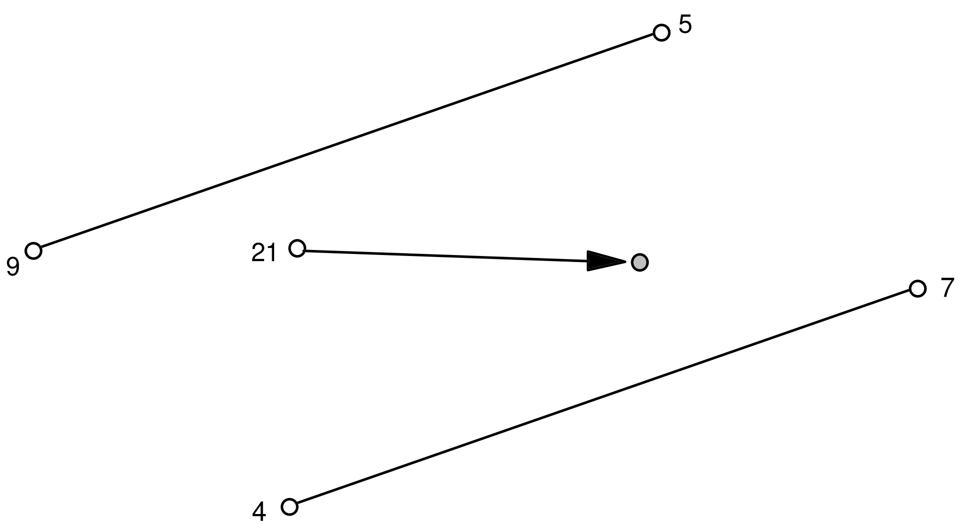
ISECT[ ] 21 9 5 4 7



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[ ]

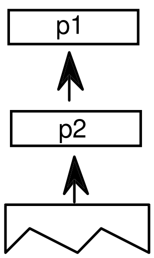
Code Range 0x27

Pops p1: point number (ULONG)
p2: point number (ULONG)

Pushes –

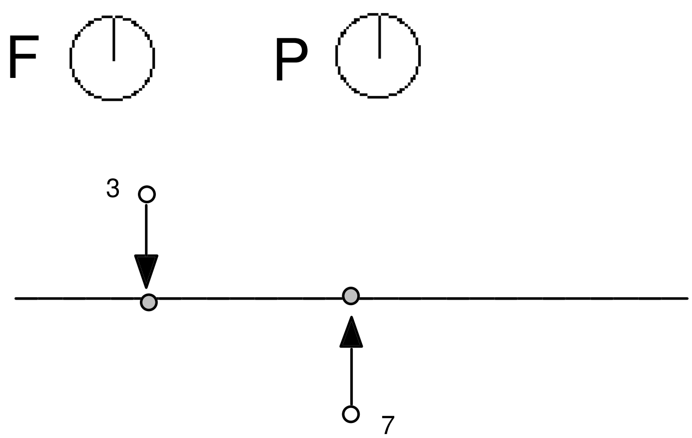
Uses zp1 with point p1, zp0 with point p2, freedom_vector, projection_vector.

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

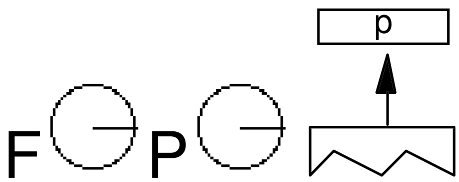


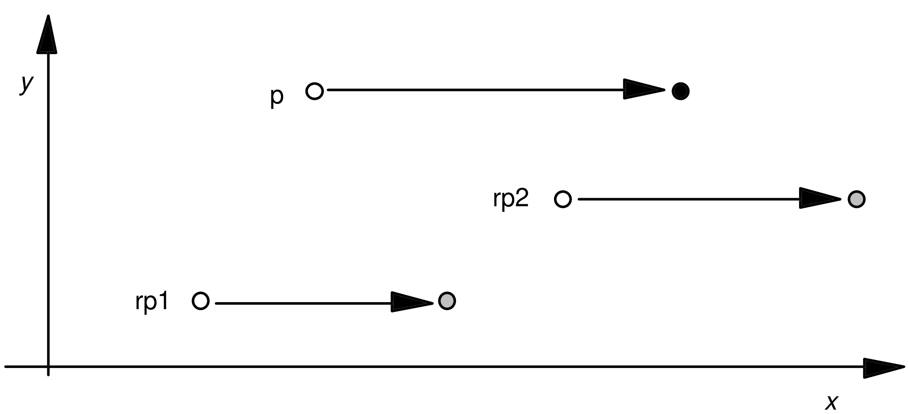
Interpolate Point by the last relative stretch

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

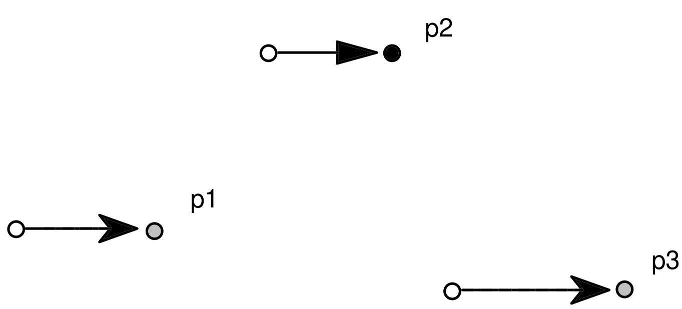
IUP[a]	
Code Range	0x30 - 0x31
a	0: interpolate in the *y*-direction
	1: interpolate in the *x*-direction
Pops	–
Pushes	–
Uses	zp2, freedom_vector, projection_vector

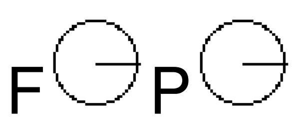
Considers a glyph contour by contour, moving any untouched points in each contour that are between a pair of touched points. If the coordinates of an untouched point were originally between those of the touched pair, it is linearly interpolated between the new coordinates, otherwise the untouched point 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.

rel. ppem	magnitude

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_i.

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

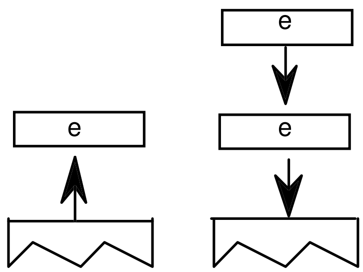
The following set of instructions make it possible to manage elements on the stack. They make it possible to duplicate the element at the top of the stack, remove the top element from the stack, clear the stack, swap the top two stack elements, determine the number of elements currently on the stack, 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.

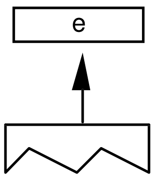


POP top stack element

POP[ ]

Code Range	0x21
Pops	e: stack element (ULONG)
Pushes	–

Pops the top element of the stack.

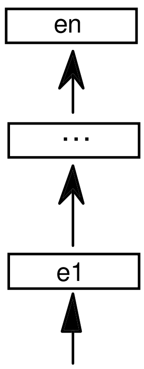


Clear the entire stack

CLEAR[ ]

Code Range	0x22
Pops	all the items on the stack (ULONGs)
Pushes	–

Clears all elements from the stack.

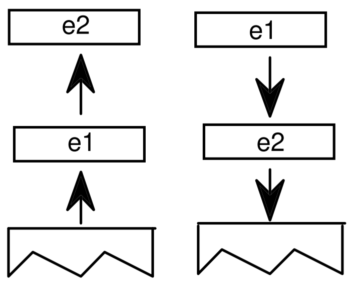


SWAP the top two elements on the stack

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

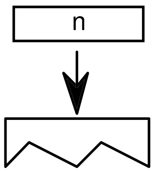
DEPTH[ ]

Code Range 0x24

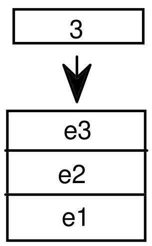
Pops –

Pushes n: number of elements (ULONG)

Pushes n, the number of elements currently in the stack onto the stack.



Example:

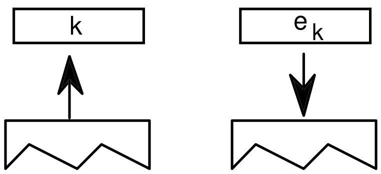


Copy the INDEXed element to the top of the stack

CINDEX[ ]

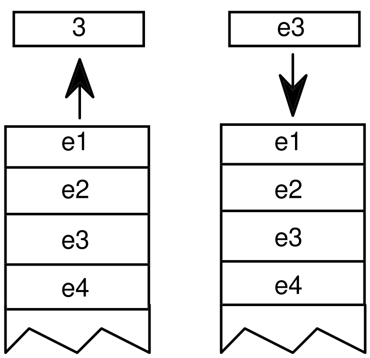
Code Range	0x25
Pops	k : stack element number
Pushes	e_k: indexed element (ULONG)

Puts a copy of the kth stack element on the top of the stack.



Example:

CINDEX[ ]

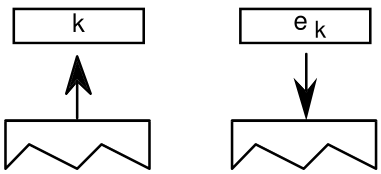


Move the INDEXed element to the top of the stack

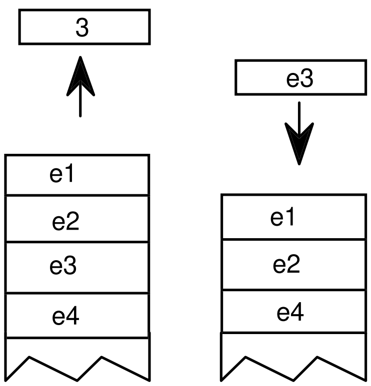
MINDEX[ ]

Code Range	0x26
Pops	k: stack element number
Pushes	e_k: indexed element

Moves the indexed element to the top of the stack.



MINDEX[ ]

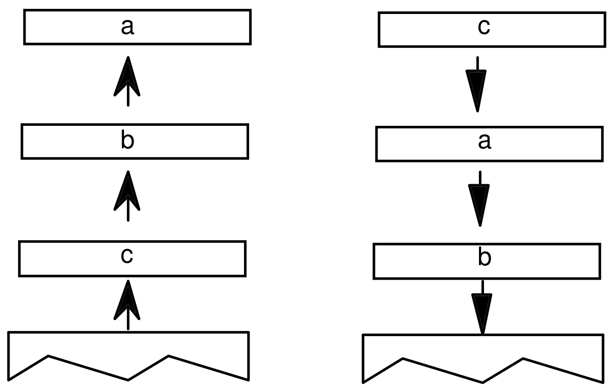


ROLL the top three stack elements

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.

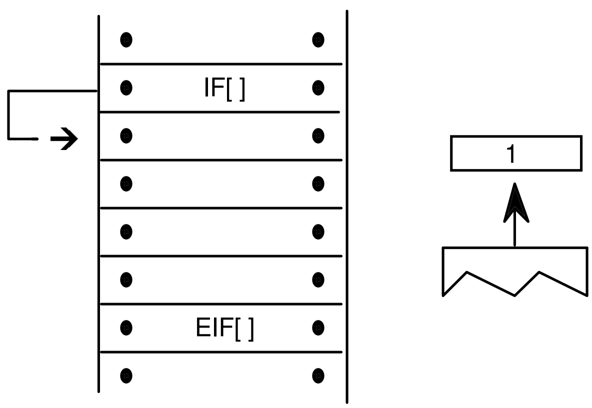
IF test

IF[ ]

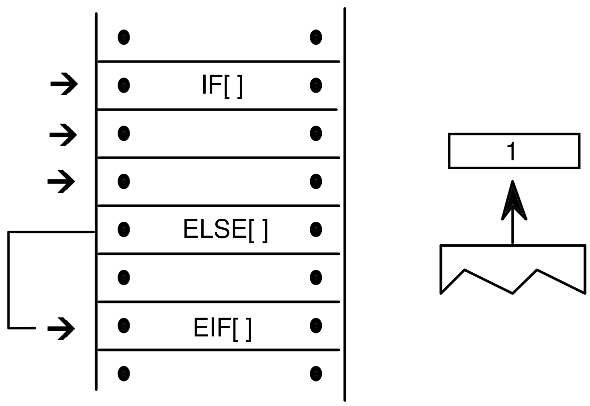
Code Range	0x58
Pops	e: stack element (ULONG)
Pushes	–

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.

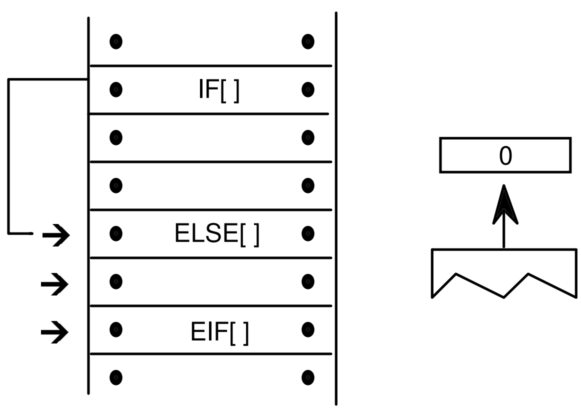
case 1:
Element at top of stack is TRUE; instruction pointer is unaffected. IF terminates with EIF.



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	–

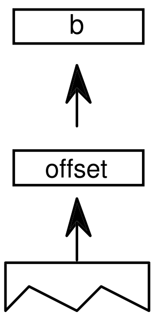
Marks the end of an IF[ ] instruction.

Jump Relative On True

JROT[ ]

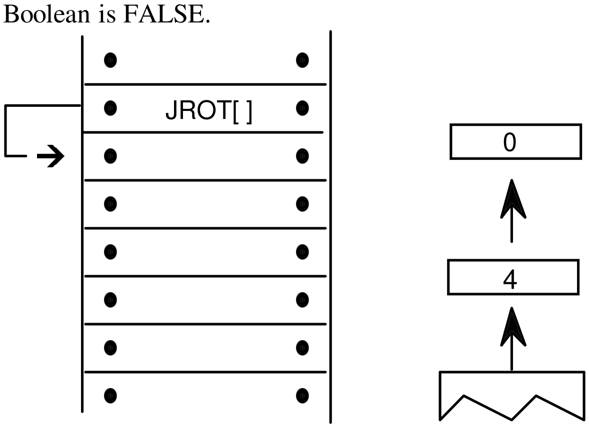
Code Range	0x78
Pops	b: Boolean (ULONG) offset: number of bytes to move instruction pointer (LONG)
Pushes	–

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.

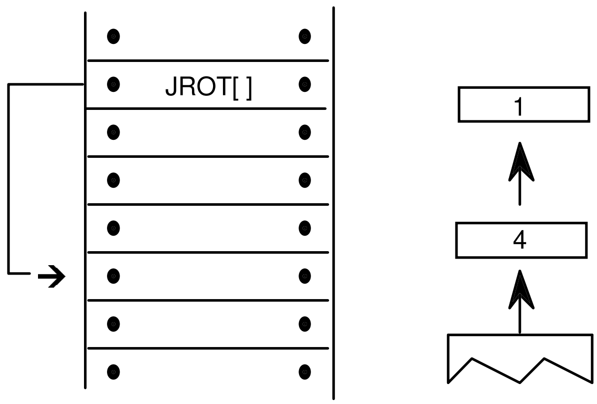


Example:

case 1:
Boolean is FALSE.



case 2:
Boolean is TRUE.



JuMP

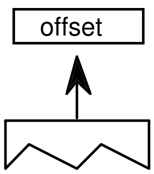
JMPR[ ]

Code Range 0x1C

Pops offset: number of bytes to move instruction pointer (LONG)

Pushes –

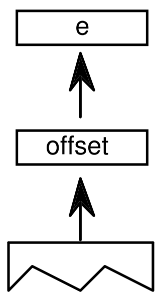
The signed offset is added to the instruction pointer and execution is resumed at the new location in the instruction steam. The jump is relative to the position of the instruction itself. That is, the instruction pointer is still pointing at the JROT[ ] instruction when offset is added to obtain the new address.



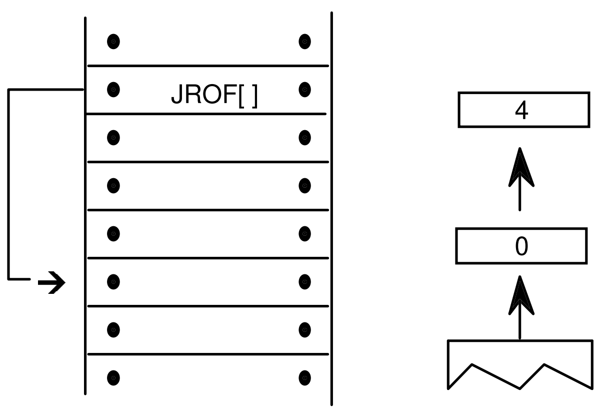
Jump Relative On False

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.

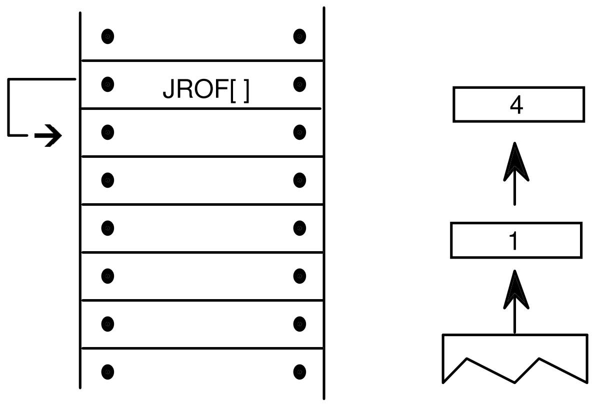


case 1:
element is FALSE.



(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:

a
b
c

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

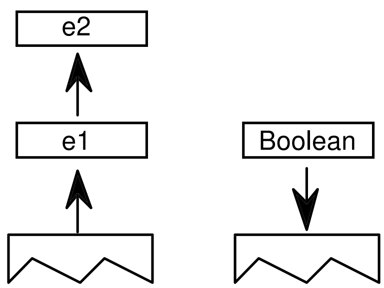
c b > a

Less Than

LT[ ]

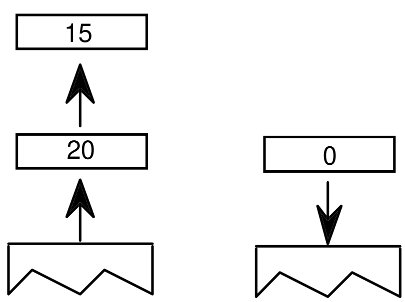
Code Range	0x50
Pops	e2: stack element (ULONG) e1: stack element (ULONG)
Pushes	Boolean value (ULONG in the range [0,1])

First pops e2, then pops 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[ ]

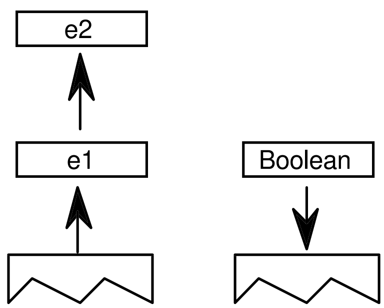


Less Than or Equal

LTEQ[ ]

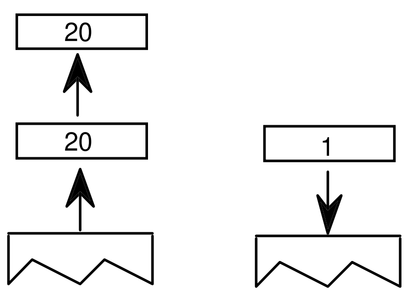
Code Range	0x51
Pops	e2: stack element (ULONG) e1: stack element (ULONG)
Pushes	Boolean value (ULONG in the range [0,1])

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[ ]

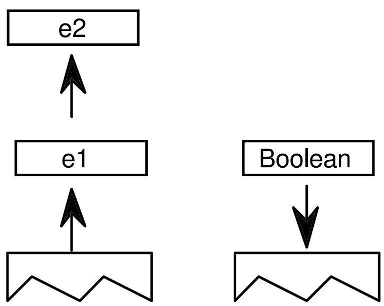


Greater Than

GT[ ]

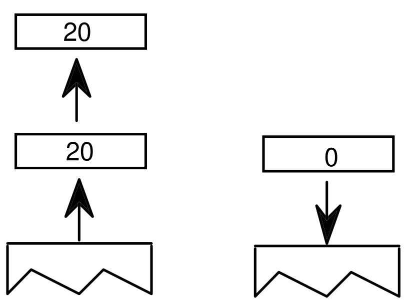
Code Range	0x52
Pops	e2: stack element (ULONG) e1: stack element (ULONG)
Pushes	Boolean value (ULONG in the range [0,1])

First pops e2 then pops e1 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

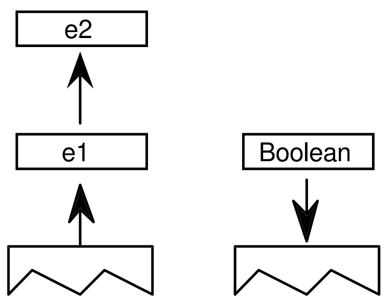
GTEQ[ ]

Code Range 0x53

Pops e2: stack element (ULONG)
e1: stack element (ULONG)

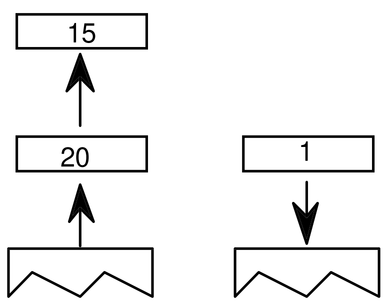
Pushes Boolean value (ULONG in the range [0,1])

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

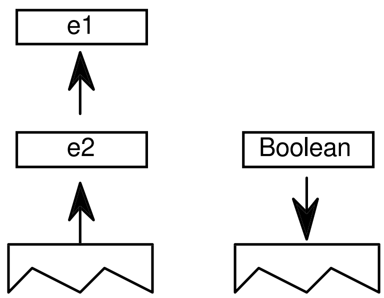
EQ[ ]

Code Range 0x54

Pops e1: stack element (ULONG)
e2: stack element (ULONG)

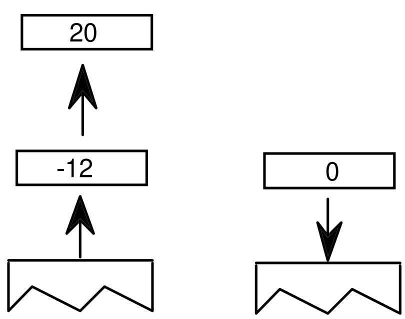
Pushes Boolean value (ULONG in the range [0,1])

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

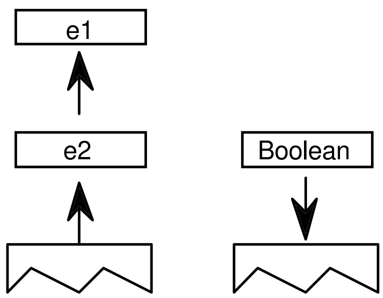
NEQ[ ]

Code Range 0x55

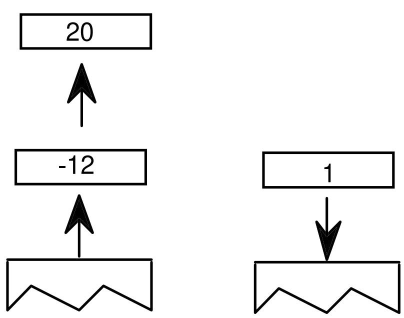
Pops e1:stack element (ULONG)
e2: stack element (ULONG)

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[ ]

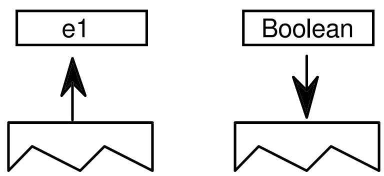
Code Range 0x56

Pops e1: stack element (F26Dot6)

Pushes Boolean value

Uses round_state

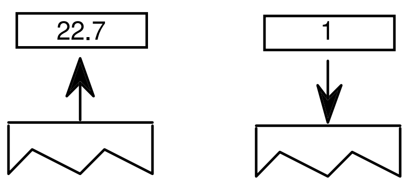
Tests whether the number at the top of the stack is odd. Pops e1 from the stack and rounds it as specified by the round_state before testing it. After the value is rounded, it is shifted from a fixed point value to an integer value (any fractional values are ignored). If the integer value is odd, one, 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[ ]

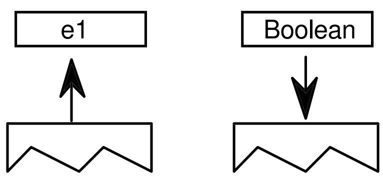
Code Range 0x57

Pops e1: stack element (F26Dot6)

Pushes Boolean value (ULONG in the range [0,1])

Uses round_state

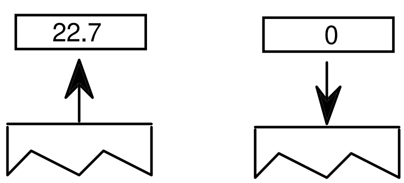
Tests whether the number at the top of the stack is even. Pops e1 off the stack and rounds it as specified by the round_state before testing it. If the rounded number is even, one, signifying TRUE, is pushed onto the stack if it is odd, zero, signifying FALSE, is placed onto the stack.



Example:

EVEN[ ]

This example assumes that round_state is RTG.



logical AND

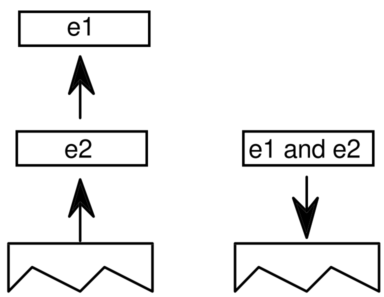
AND[ ]

Code Range 0x5A

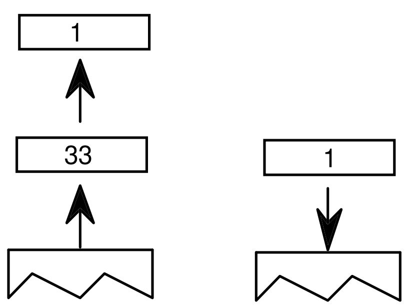
Pops e1: stack element (ULONG)
e2: stack element (ULONG)

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).



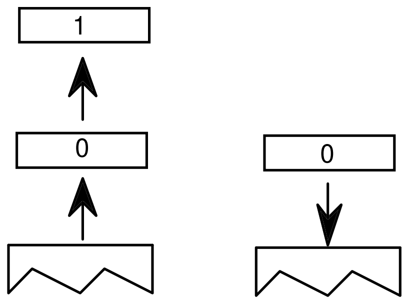
case 1:
AND[ ]



(continued...)

case 2:

AND[ ]



logical OR

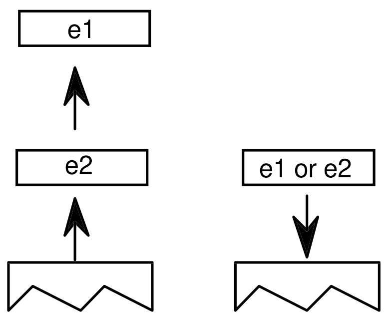
OR[ ]

Code Range 0x5B

Pops e1: stack element (ULONG)
e2: stack element (ULONG)

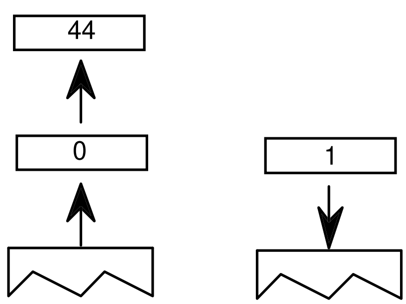
Pushes (e1 or e2): logical or of e1 and e2 (ULONG)

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:

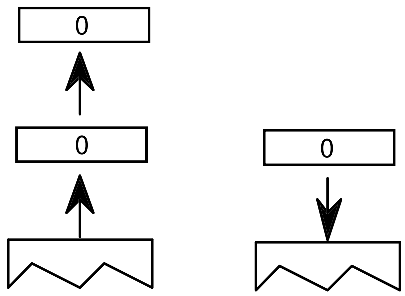
case 1:
OR[ ]



(continued...)

case 2:

OR[ ]



logical NOT

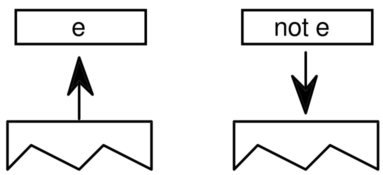
NOT[ ]

Code Range 0x5C

Pops e: stack element (ULONG)

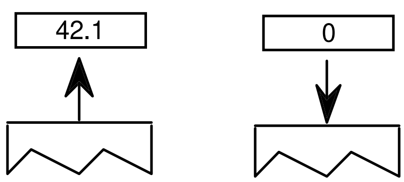
Pushes (not e): logical negation of e (ULONG)

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.

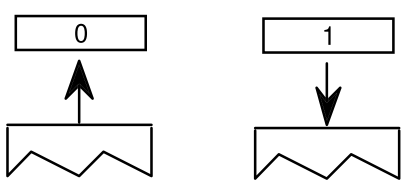


Example:

case 1:



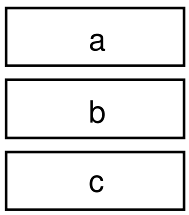
case 2:



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:



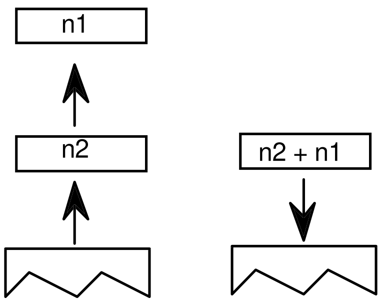
subtract a,b would be interpreted as (b-a):

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

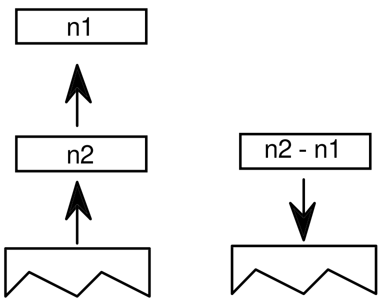
SUB[ ]

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[ ]

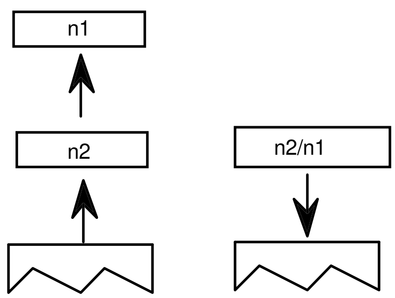
Code Range 0x62

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. The TrueType Rasterizer v.1.7 and later will catch any division-by-zero errors.



MULtiply

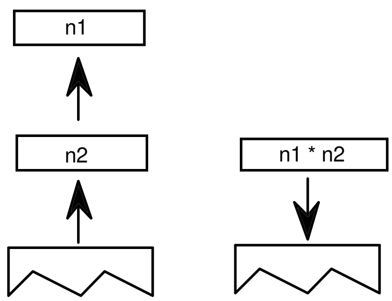
MUL[ ]

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.

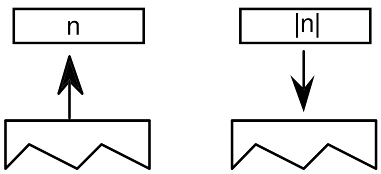


ABSolute value

ABS[ ]

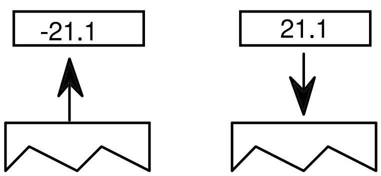
Code Range	0x64
Pops	n
Pushes	|n|: absolute value of n (F26Dot6)

Pops n off the stack and pushes onto the stack the absolute value of n.

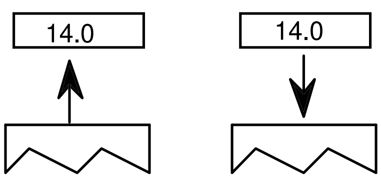


Example:

case 1:



case 2:

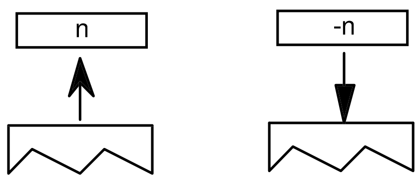


NEGate

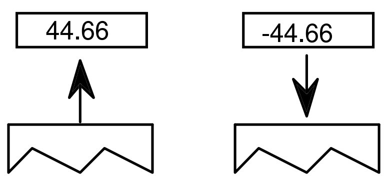
NEG[ ]

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

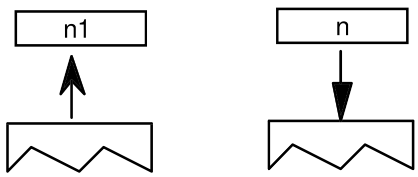
FLOOR[ ]

Code Range 0x66

Pops n1: number whose floor is desired (F26Dot6)

Pushes n : floor of n1 (F26Dot6)

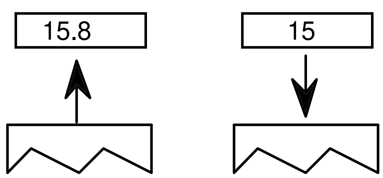
Pops n1 and returns n, the greatest integer value less than or equal to n1.



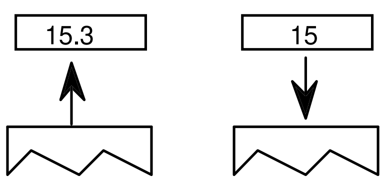
Example:

FLOOR[ ]

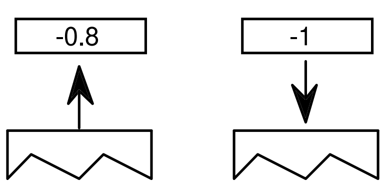
case 1:



case 2:



case 3:



CEILING

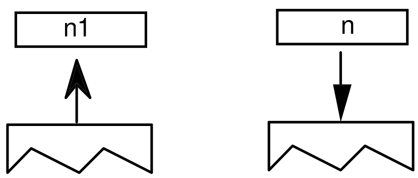
CEILING[ ]

Code Range 0x67

Pops n1: number whose ceiling is desired (F26Dot6)

Pushes n: ceiling of n1 (F26Dot6)

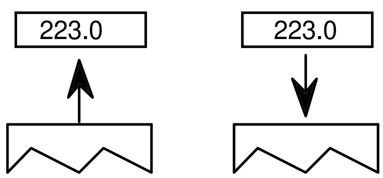
Pops n1 and returns n, the least integer value greater than or equal to n1. For instance, the ceiling of 15 is 15, but the ceiling of 15.3 is 16. The ceiling of -0.8 is 0. (n is the least integer value greater than or equal to n1)



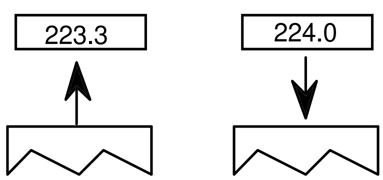
Example:

CEILING[ ]

case 1:



case 2:

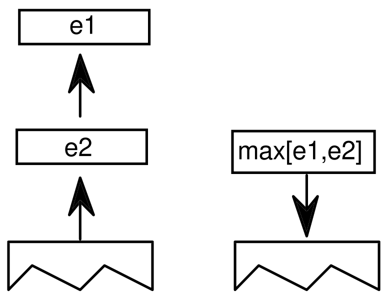


MAXimum of top two stack elements

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.

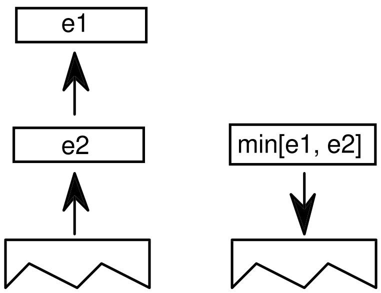


MINimum of top two stack elements

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

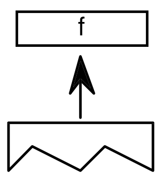
FDEF[ ]

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

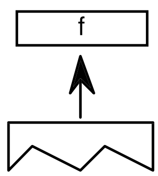
CALL[ ]

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

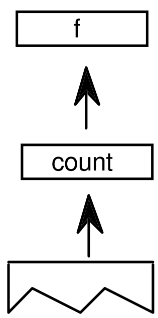
LOOPCALL[ ]

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),5`	push count
`PUSHB(000),17`	push function number
`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

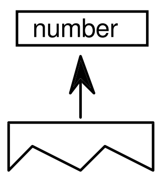
The TrueType instruction set provides the following instruction as an aid to debugging.

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, a value of 32 asks whether the glyph is being rasterized for grayscale. (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 asks for stretching information, setting bit 5 asks for grayscale 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, version 3 is Windows 3.1, version 33 is MS rasterizer v.1.5, version 34 is MS rasterizer v.1.6, version 35 is MS rasterizer v.1.7). Version numbers 0 and 4 through 255 are reserved.

Note: Before hinting vertical phantom points or asking for grayscale information, use GETINFO to check that the rasterizer in use is MS rasterizer v.1.7 or later.

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.

Bit 12 is set to 1 if the rasterization is done for grayscale (for MS rasterizer v.1.7 and later).

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.