Chapter 6

The TrueType Instruction Set

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

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

Anatomy of a TrueType Instruction

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

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

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

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

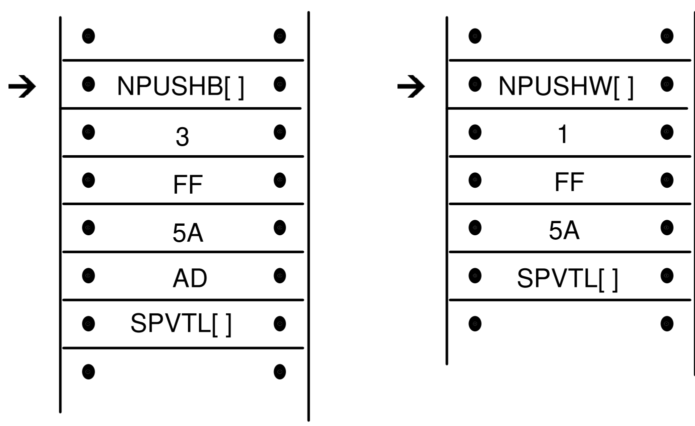
If these flags were set to 11101 the code would be computed as follows:

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

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

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

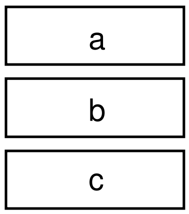
Figure 3–1 The instruction stream with a push byte instruction (left) and a push word instruction (right)



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

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

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



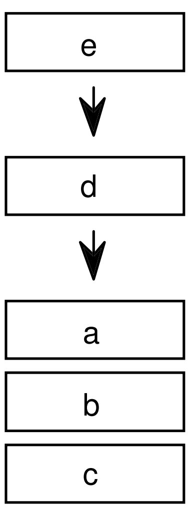
To easily remember the order in which stack values are handled during arithmetic or logical operations, imagine writing the stack values from left to right, starting with the bottom value. Then insert the operator between the two furthest right elements. For example, *subtract a,b* would be interpreted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

c b - a

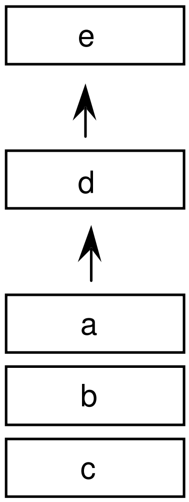
GT a,b would be interpreted as (b>a):

c b > a

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



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



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

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

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

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

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

Data types

The instruction stream

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

The stack

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

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

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

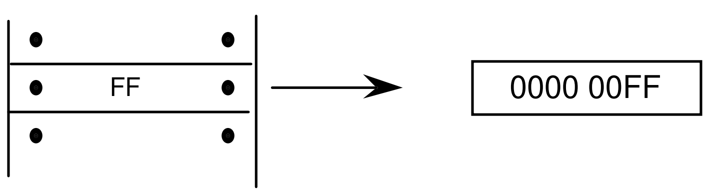
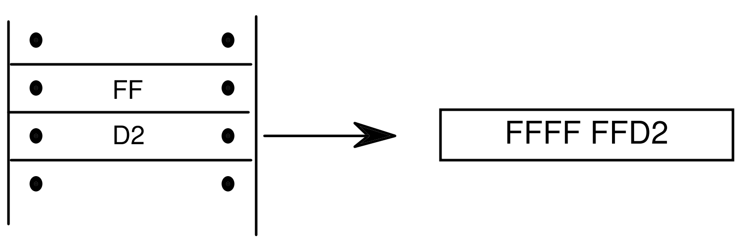


Figure 3–3 A word sign extended to a 32 bit long word (ULONG)



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

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

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

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

0000 0000 0000 0000 0000 0001 0000 1000

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

00 0000 0000 0000 0000 0000 0100 .00 1000

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

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

Pushing data onto the interpreter stack

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

PUSH N Bytes

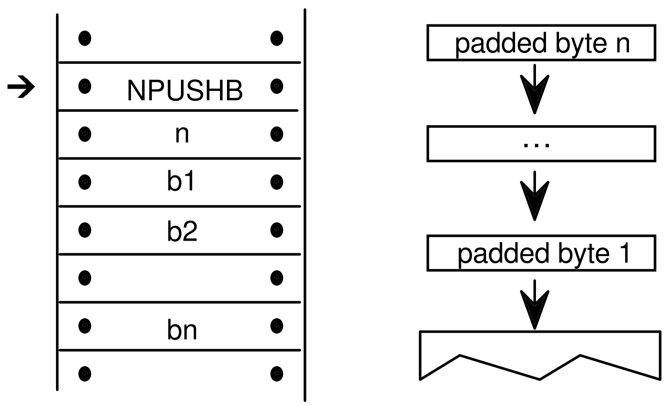
NPUSHB[ ]

Code Range 0x40

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

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

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



PUSH N Words

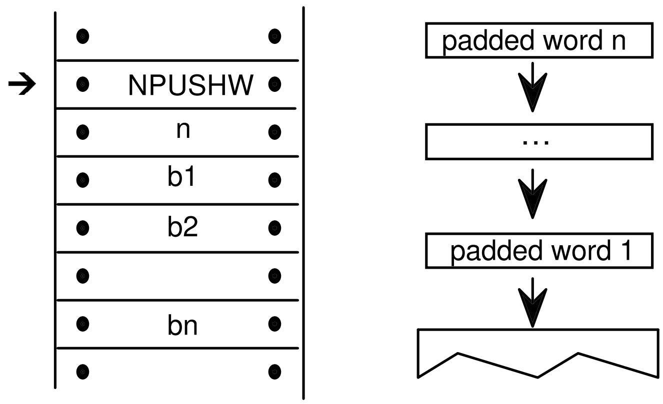
NPUSHW[ ]

Code Range 0x41

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

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

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

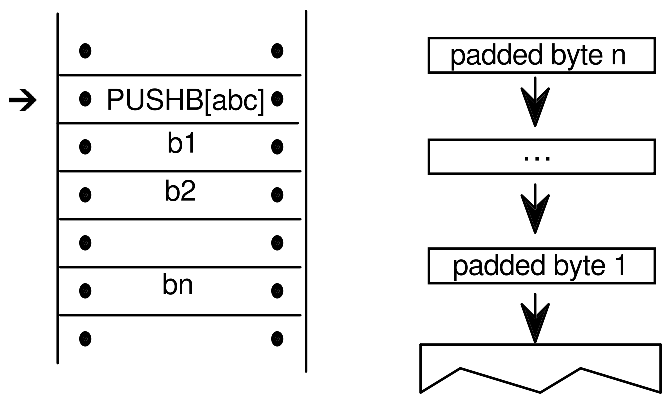


PUSH Bytes

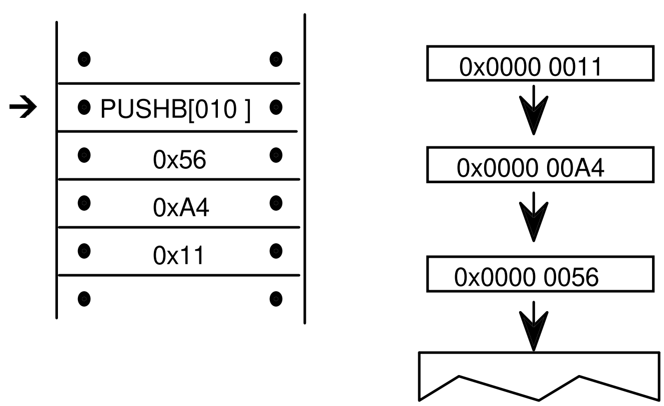
PUSHB[abc]	
Code Range	0xB0 – 0xB7
abc	number of bytes to be pushed – 1
From IS	b0, b1,..bn: sequence of n + 1 bytes
Pushes	b0, b1, ...,bn: sequence of n + 1 bytes each padded to 32 bits (ULONG)

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

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



Example:



PUSH Words

PUSHW[abc]

Code Range 0xB8 - 0xBF

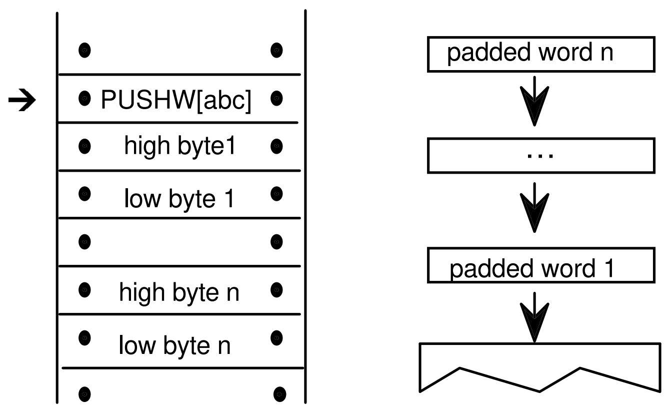
abc number of words to be pushed – 1.

From IS w0,w1,..wn: sequence of n+1 words formed from pairs of bytes, the high byte appearing first

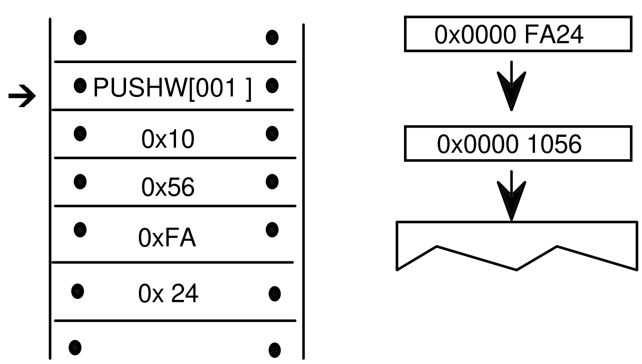
Pushes w0 ,w1,...wn: sequence of n+1 words each sign extended to 32 bits (LONG)

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

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



Example:



Managing the Storage Area

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

Read Store

RS[ ]

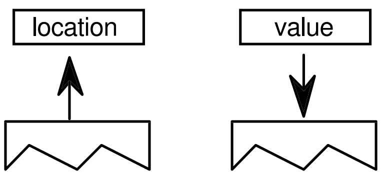
Code Range 0x43

Pops location: Storage Area location (ULONG)

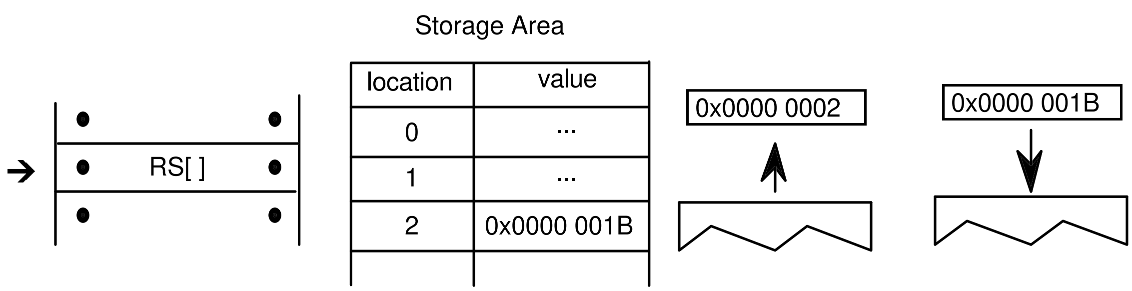
Pushes value: Storage Area value (ULONG)

Gets Storage Area value

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



Example:



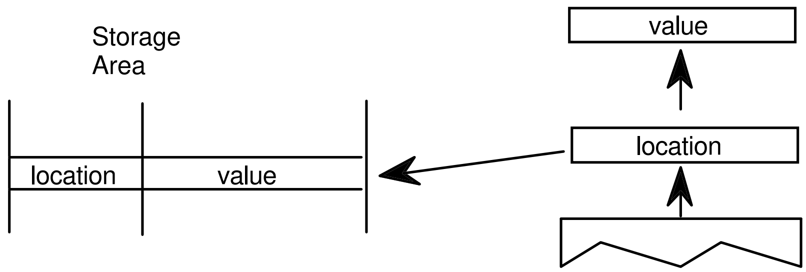
The effect of the RS instruction is to push the value 0x1B of the Storage Area onto the Stack.

Write Store

WS[ ]

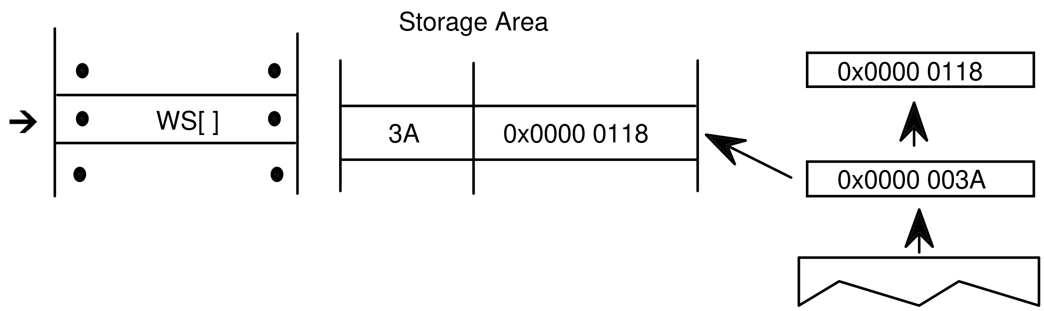
Code Range	0x42
Pops	value: Storage Area value (ULONG)
	location: Storage Area location (ULONG)
Pushes	–
Sets	Storage Area value

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



Example:

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



Managing the Control Value Table

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

Write Control Value Table in Pixel units

WCVTP[ ]

Code Range 0x44

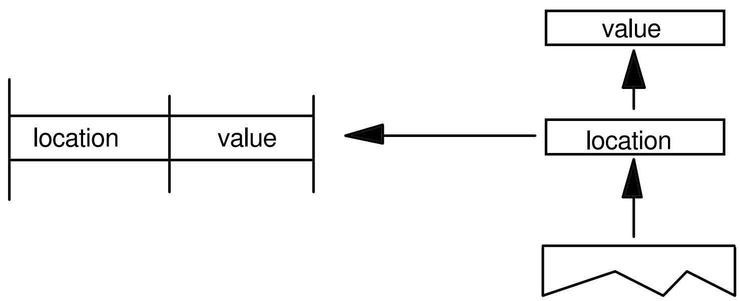
Pops value: number in pixels (F26Dot6 fixed point number)

location: Control Value Table location (ULONG)

Pushes –

Sets Control Value Table entry

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

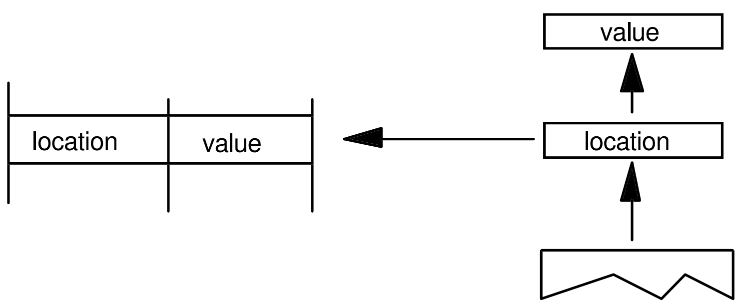


Write Control Value Table in FUnits

WCVTF[ ]

Code Range	0x70
Pops	value: number in FUnits (ULONG)
	location: Control Value Table location (ULONG)
	Pushes –
Sets	Control Value Table entry

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

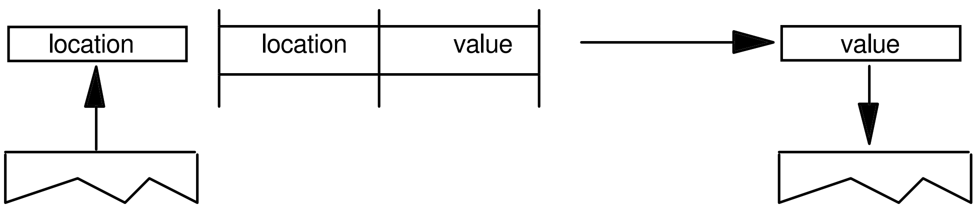


Read Control Value Table

RCVT[ ]

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

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



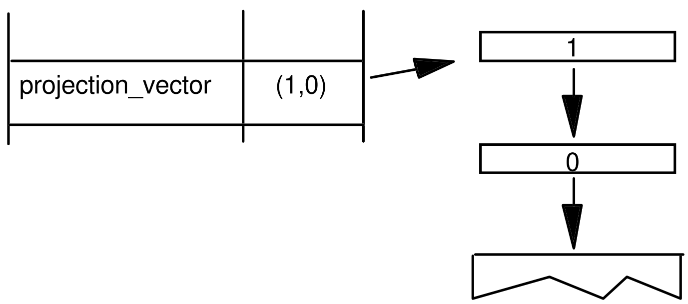
Managing the Graphics State

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

Getting a value

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

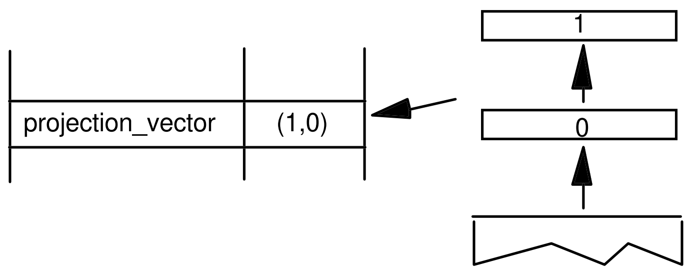
The illustration shows the effect of a GPV or get projection_vector instruction. It takes the *x* and *y* components of the projection_vector from the Graphics State and places them on the stack.



Setting a value

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

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



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

Set freedom and projection Vectors To Coordinate Axis

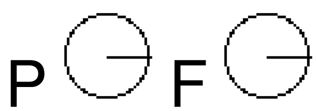
SVTCA[a]	
Code range	0x00 - 0x01
a	0: set vectors to the *y*-axis 1: set vectors to the *x*-axis
Pops	–
Pushes	–
Sets	projection_vector freedom_vector

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

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

Example:

SVTCA[1]



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

SVTCA[0]

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

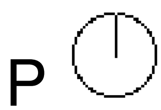
Set Projection_Vector To Coordinate Axis

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

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

Example:

SPVTCA[0]



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

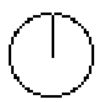
Set Freedom_Vector to Coordinate Axis

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

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

Example:

SFVTCA[0]

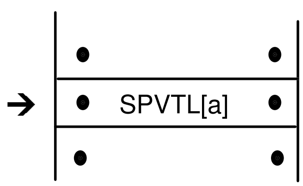
F 

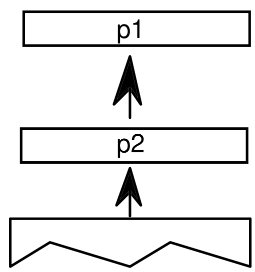
Sets the freedom_vector to the *y*-axis ensuring that movement will be along that axis.

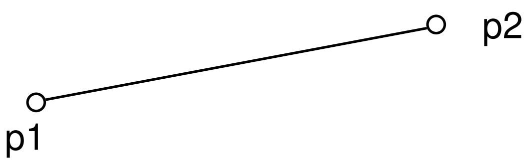
Set Projection_Vector To Line

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

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

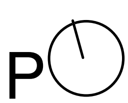






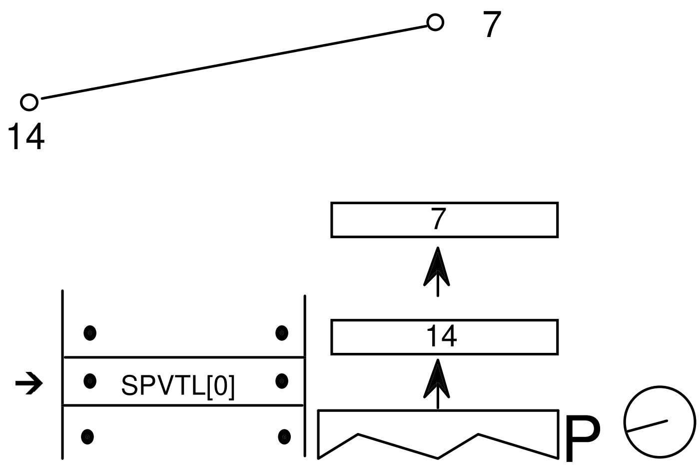
If parallel, the projection_vector points from p1 toward p2 as shown.

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



case 1:

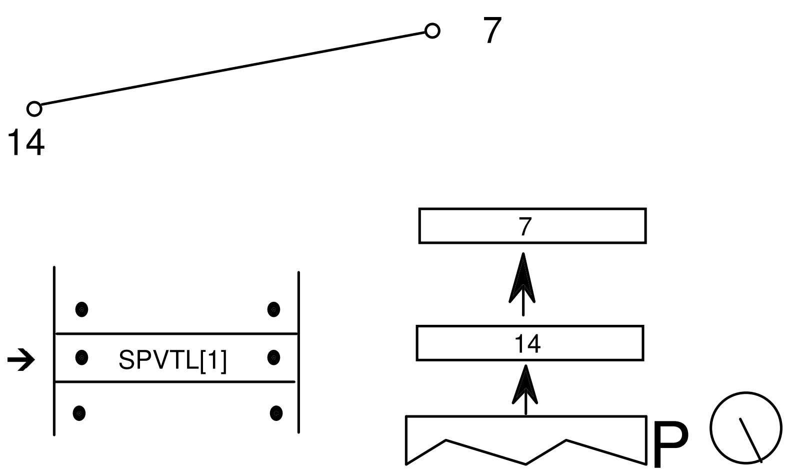
SPVTL[1]



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

case 2:

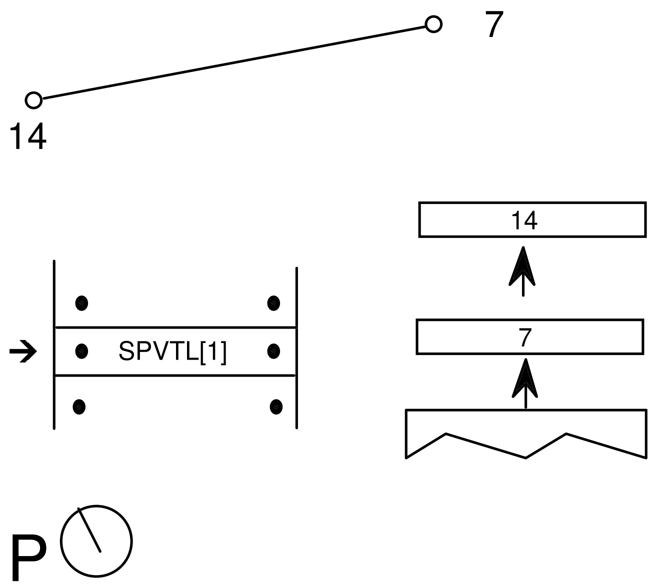
SPVTL[1]



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

case 3:

SPVTL[1]

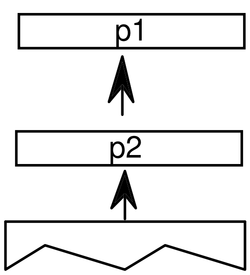


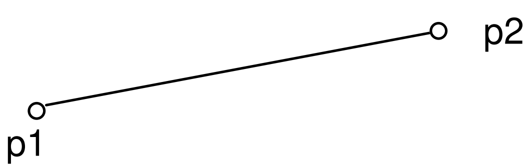
The order in which the points are specified matters. This instruction sets the projection_vector to be perpendicular to the line from point 14 to point 7.

Set Freedom_Vector To Line

SFVTL[a]	
Code Range	0x08 - 0x09
a	0: set freedom_vector to be parallel to the line segment defined by points p1 and p2
	1: set freedom_vector perpendicular to the line segment defined by points p1 and p2; the vector is rotated counter clockwise 90 degrees
Pops	p1: point number (ULONG)
	p2: point number (ULONG)
Pushes	–
Sets	freedom_vector
Uses	point p1 in the zone pointed at by zp2 point p2 in the zone pointed at by zp1

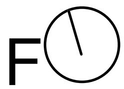
Sets the freedom_vector to a unit vector parallel or perpendicular to the line segment defined by points p1 and p2.





If parallel the freedom_vector points from p1 toward p2 as shown.

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



Set Freedom_Vector To Projection Vector

SFVTPV[ ]

Code	0x0E
Pops	–
Pushes	–
Sets	freedom_vector

Sets the freedom_vector to be the same as the projection_vector.

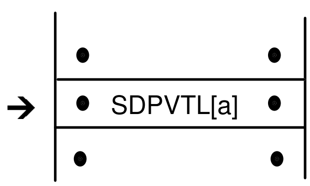
Before

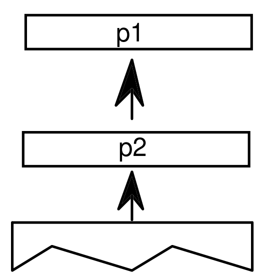
After

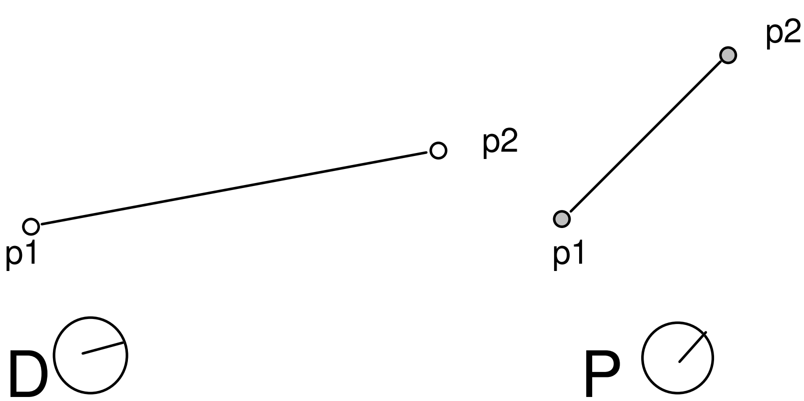
Set Dual Projection_Vector To Line

SDPVTL[a]	
Code Range	0x86 - 0x87
a	0: Vectors are parallel to line 1: Vectors are perpendicular to line
Pops	p1: first point number (ULONG)
	p2: second point number (ULONG)
Pushes	–
Sets	dual_projection_vector and projection_vector
Uses	point p1 in the zone pointed at by zp2 point p2 in the zone pointed at by zp1

Pops two point numbers from the stack and uses them to specify a line that defines a second, dual_projection_vector. This dual_projection_vector uses coordinates from the scaled outline before any grid-fitting took place. It is used only with the IP, GC, MD, MDRP and MIRP instructions. Those instructions will use the dual_projection_vector when they measure distances between ungrid-fitted points. The dual_projection_vector will disappear when any other instruction that sets the projection_vector is used.







NOTE: The dual_projection_vector is set parallel to the points as they appeared in the original outline before any grid-fitting took place.

Set Projection_Vector From Stack

SPVFS[ ]

Code Range 0x0A

Pops *y*: *y* component of projection_vector (2.14 fixed point number padded with zeroes)

x: *x* component of projection_vector (2.14 fixed point number padded with zeroes)

Pushes –

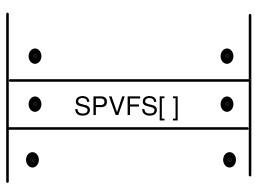
Sets projection_vector

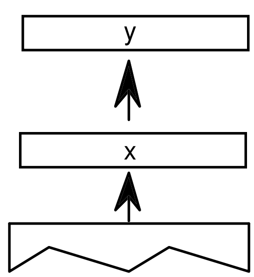
Sets the direction of the projection_vector, using values x and y taken from the stack, so that its projections onto the *x* and *y*-axes are *x* and *y*, which are specified as signed (two's complement) fixed-point (2.14) numbers. The square root of ($x^2 + y^2$) must be equal to 0x4000 (hex).

If values are to be saved and used by a glyph program, font program or preprogram across different resolutions, extreme care must be used. The values taken from or put on the stack are 2.14 fixed-point values for the x and y components of the vector in question. The values are based on the normalized vector lengths. More simply, the values must always be set such that (X**2 + Y**2) is 1.

If a TrueType program uses specific values for X and Y to set the vectors to certain angles, these values will *not* produce identical results across different aspect ratios. Values that work correctly at 1:1 aspect ratios (such as VGA and 8514) will not necessarily yield the desired results at a ratio of 1.33:1 (e.g. the EGA).

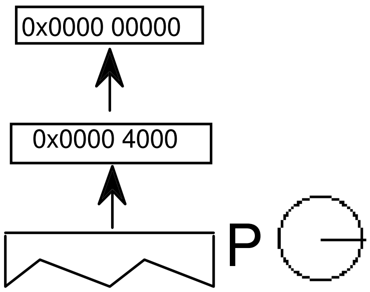
By the same token, if a TrueType program is making use of the values returned by GPV and GFV, the values returned for a specific angle will vary with the aspect ratio in use at the time.





Example:

SPVFS[ ]



Sets the projection_vector to a unit vector that points in the direction of the *x*-axis

Set Freedom_Vector From Stack

SFVFS[ ]

Code 0x0B

Pops *y*: *y* component of freedom_vector (2.14 fixed point number padded with zeroes)

x: *x* component of freedom_vector (2.14 fixed point number padded with zeroes)

Pushes –

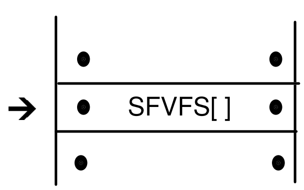
Sets freedom_vector

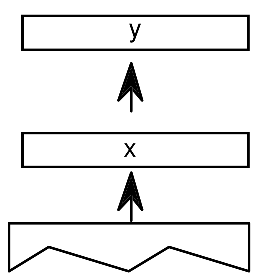
Sets the direction of the freedom_vector using the values x and y taken from the stack. The vector is set so that its projections onto the *x* and *y* -axes are *x* and *y*, which are specified as signed (two's complement) fixed-point (2.14) numbers. The square root of ($x^2 + y^2$) must be equal to 0x4000 (hex).

If values are to be saved and used by a glyph program, font program or preprogram across different resolutions, extreme care must be used. The values taken from or put on the stack are 2.14 fixed-point values for the x and y components of the vector in question. The values are based on the normalized vector lengths. More simply, the values must always be set such that (X**2 + Y**2) is 1.

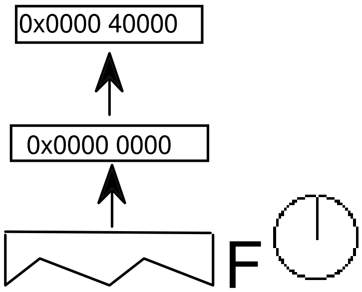
If a TrueType program uses specific values for X and Y to set the vectors to certain angles, these values will *not* produce identical results across different aspect ratios. Values that work correctly at 1:1 aspect ratios (such as VGA and 8514) will not necessarily yield the desired results at a ratio of 1.33:1 (e.g. the EGA).

By the same token, if a TrueType program is making use of the values returned by GPV and GFV, the values returned for a specific angle will vary with the aspect ratio in use at the time.





Example:



Sets the freedom_vector to a unit vector that points in the direction of the *y*-axis.

Get Projection_Vector

GPV[ ]

Code Range 0x0C

Pops –

Pushes *x*: *x* component of projection_vector (2.14 fixed point number padded with zeroes)

y : *y* component of projection_vector (2.14 fixed point number padded with zeroes)

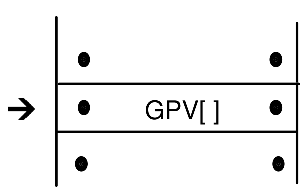
Gets projection_vector

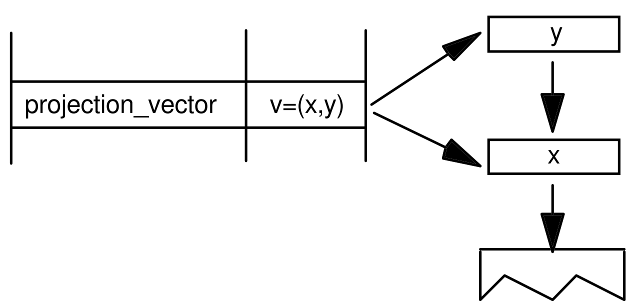
Pushes the x and y components of the projection_vector onto the stack as two 2.14 numbers.

If values are to be saved and used by a glyph program, font program or preprogram across different resolutions, extreme care must be used. The values taken from or put on the stack are 2.14 fixed-point values for the x and y components of the vector in question. The values are based on the normalized vector lengths. More simply, the values must always be set such that (X**2 + Y**2) is 1.

If a TrueType program uses specific values for X and Y to set the vectors to certain angles, these values will *not* produce identical results across different aspect ratios. Values that work correctly at 1:1 aspect ratios (such as VGA and 8514) will not necessarily yield the desired results at a ratio of 1.33:1 (e.g. the EGA).

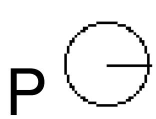
By the same token, if a TrueType program is making use of the values returned by GPV and GFV, the values returned for a specific angle will vary with the aspect ratio in use at the time.





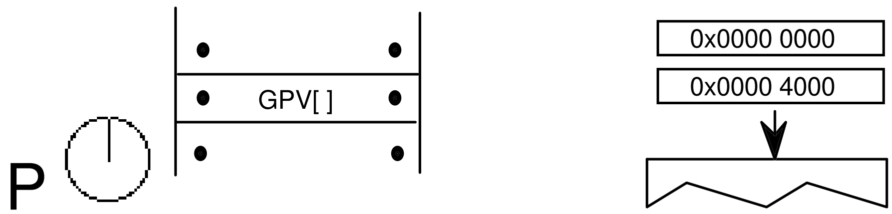
Example:

case 1:



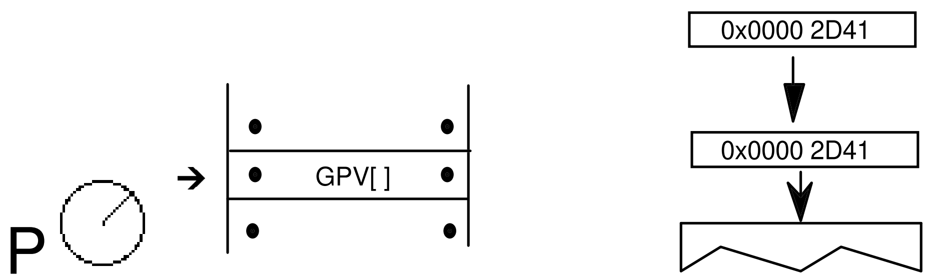
The stack entry 0x4000 which when interpreted as a 2.14 number is simply 1. This command reveals that, in this case, the projection_vector is a unit vector that points in the *x*-direction.

case 2:



Here the projection_vector is a unit vector that points in the direction of the *y*-axis.

case 3:



NOTE: 0x2D41 is the hex equivalent of $\frac{sqrt(2)}{2}$*. As a result of this instruction, the projection_vector is set to a 45 degree angle relative to the x-axis.*

Get Freedom_Vector

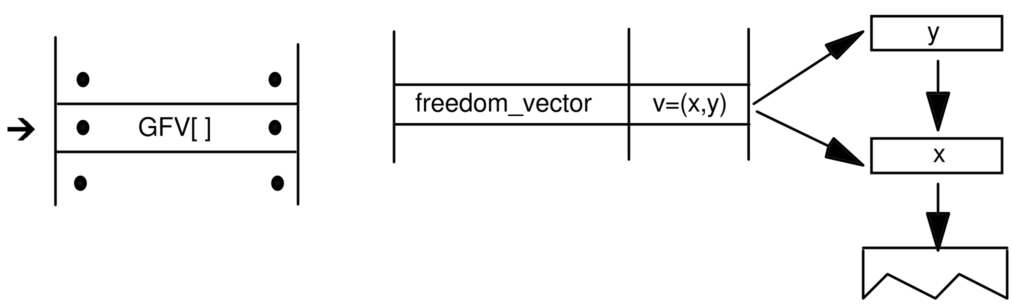
GFV[ ]	
Code Range	0x0D
Pops	–
Pushes	x: x-component of freedom_vector (2.14 number padded with zeroes)
	y: y component of freedom_vector (2.14 number padded with zeroes)
Gets	freedom_vector

Puts the *x* and *y* components of the freedom_vector on the stack. The freedom_vector is put onto the stack as two 2.14 coordinates.

If values are to be saved and used by a glyph program, font program or preprogram across different resolutions, extreme care must be used. The values taken from or put on the stack are 2.14 fixed-point values for the x and y components of the vector in question. The values are based on the normalized vector lengths. More simply, the values must always be set such that (X**2 + Y**2) is 1.

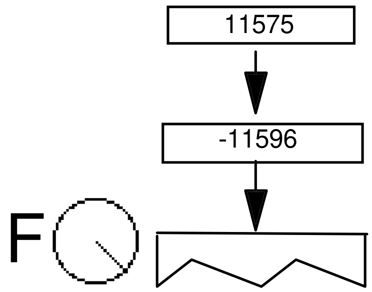
If a TrueType program uses specific values for X and Y to set the vectors to certain angles, these values will *not* produce identical results across different aspect ratios. Values that work correctly at 1:1 aspect ratios (such as VGA and 8514) will not necessarily yield the desired results at a ratio of 1.33:1 (e.g. the EGA).

By the same token, if a TrueType program is making use of the values returned by GPV and GFV, the values returned for a specific angle will vary with the aspect ratio in use at the time.



Example

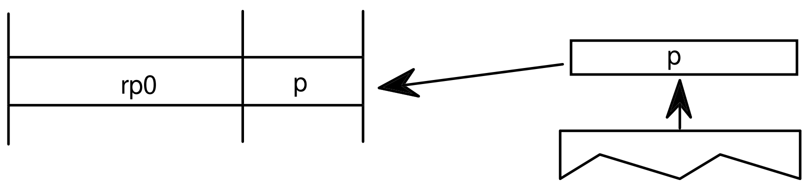
GFV[ ]



Set Reference Point 0

SRP0[ ]	
Code Range	0x10
Pops	p: point number (ULONG)
Pushes	–
Sets	rp0
Affects	IP, MDAP, MIAP, MIRP, MSIRP, SHC, SHE, SHP

Pops a point number from the stack and sets rp0 to that point number.

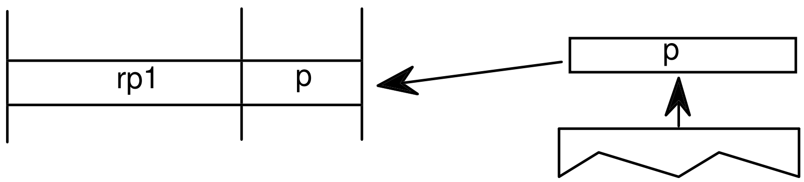


Set Reference Point 1

SRP1[ ]

Code Range	0x11
Pops	p: point number (ULONG)
Pushes	–
Sets	rp1
Affects	IP, MDAP, MDRP, MIAP, MSIRP, SHC, SHE, SHP

Pops a point number from the stack and sets rp1 to that point number.

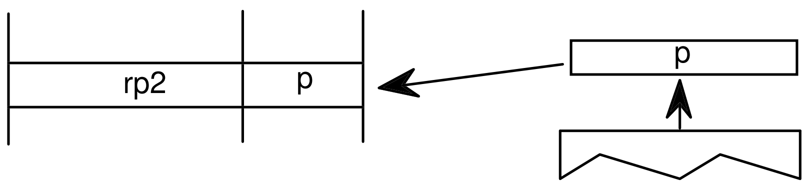


Set Reference Point 2

SRP2[ ]

Code Range	0x12
Pops	p:point number (ULONG)
Pushes	–
Sets	rp2

Pops a point number from the stack and sets rp2 to that point number.

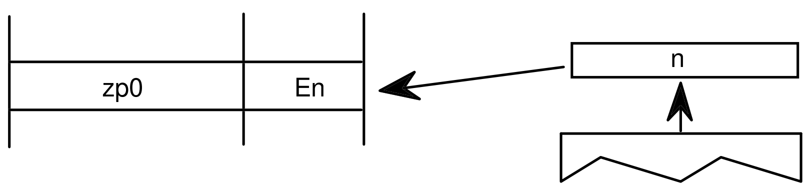


Set Zone Pointer 0

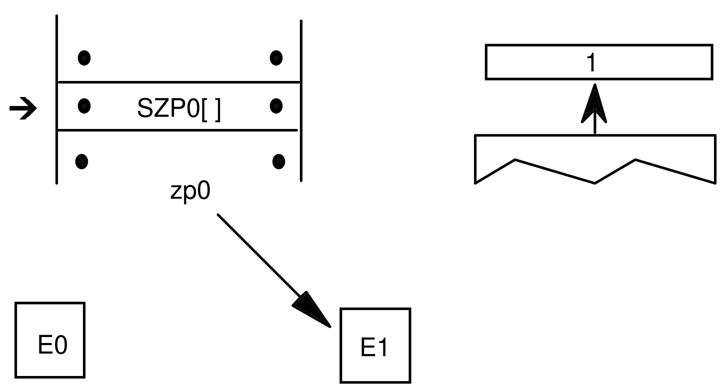
SZP0[ ]

Code Range	0x13
Pops	n: zone number (ULONG)
Pushes	–
Sets	zp0
Affects	ALIGNPTS, ALIGNRP, DELTAP1, DELTAP2, DELTAP3, IP, ISECT, MD, MDAP, MIAP, MIRP, MSIRP, SHC, SHE, SHP, UTP

Pops a zone number, n, from the stack and sets zp0 to the zone with that number. If n is 0, zp0 points to zone 0. If n is 1, zp0 points to zone 1. Any other value for n is an error.



Example:

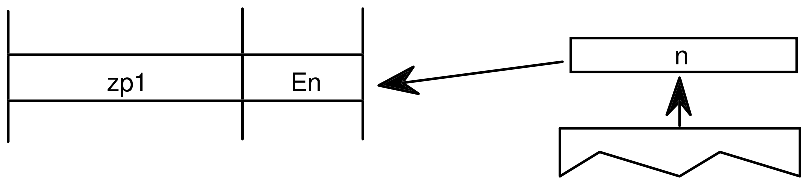


Set Zone Pointer 1

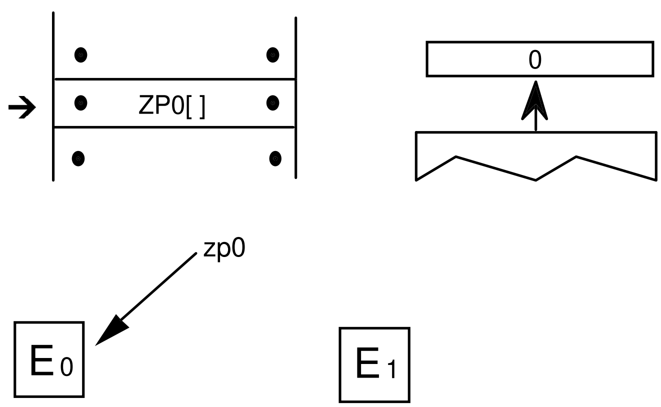
SZP1[ ]

Code Range	0x14
Pops	n: zone number (ULONG)
Pushes	–
Sets	zp1
Affects	ALIGNRPTS, ALIGNRP, IP, MD, MDRP, MSIRP, SHC, SHE, SHP, SFVTL, SPVTL

Pops a zone number, n, from the stack and sets zp1 to the zone with that number. If n is 0, zp1 points to zone 0. If n is 1, zp1 points to zone 1. Any other value for n is an error.



Example

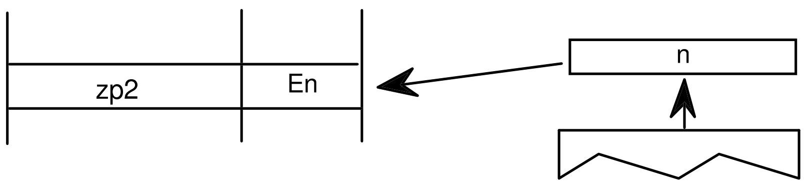


Set Zone Pointer 2

SZP2[ ]

Code Range	0x15
Pops	n: zone number (ULONG)
Pushes	–
Sets	zp2
Affects	ISECT, IUP, GC, SHC, SHP, SFVTL, SHPIX, SPVTL, SC

Pops a zone number, n, from the stack and sets zp2 to the zone with that number. If n is 0, zp2 points to zone 0. If n is 1, zp2 points to zone 1. Any other value for n is an error.

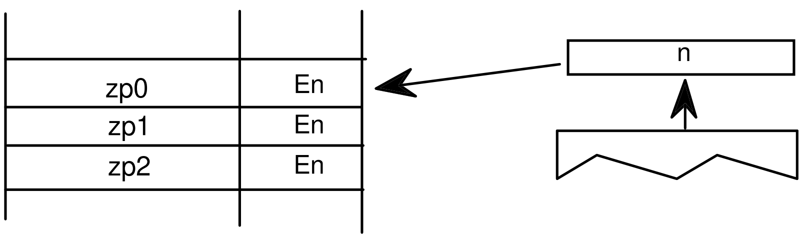


Set Zone PointerS

SZPS[ ]

Code Range	0x16
Pops	n: zone number (ULONG)
Pushes	–
Sets	zp0, zp1, zp2
Affects	ALIGNPTS, ALIGNRP, DELTAP1, DELTAP2, DELTAP3, GC, IP, ISECT, IUP, MD, MDAP, MDRP, MIAP, MIRP, MSIRP, SC, SFVTL, SHPIX, SPVTL, SHC, SHE, SHP, SPVTL, UTP

Pops a zone number from the stack and sets all of the zone pointers to point to the zone with that number. If n is 0, all three zone pointers will point to zone 0. If n is 1, all three zone pointers will point to zone 1. Any other value for n is an error.



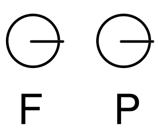
Round To Half Grid

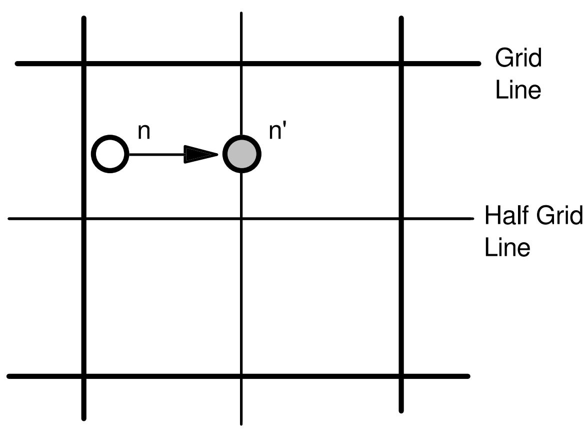
RTHG[ ]	
Code Range	0x19
Pops	–
Pushes	–
Sets	round_state
Affects	MDAP, MDRP, MIAP, MIRP, ROUND
Uses	freedom_vector, projection_vector

Sets the round_state variable to state 0 (*hg*). In this state, the coordinates of a point are rounded to the nearest half grid line.

Example:

RTHG[ ]





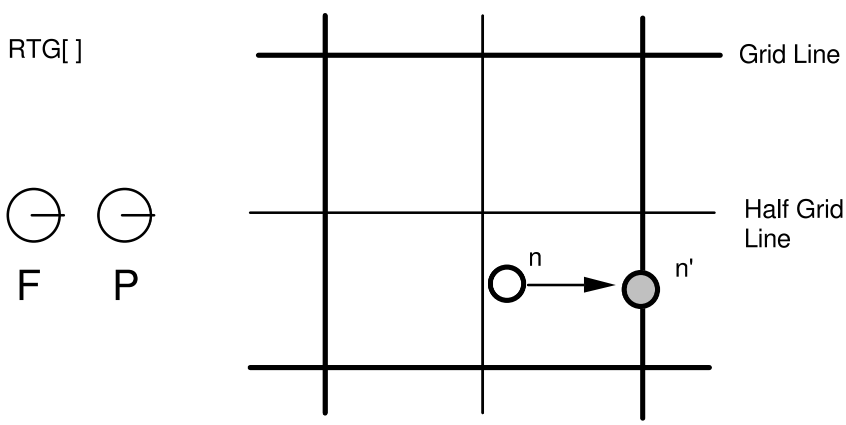
Round To Grid

RTG[ ]

Code Range	0x18
Pops	–
Pushes	–
Sets	round_state
Affects	MDAP, MDRP, MIAP, MIRP, ROUND
Uses	freedom_vector, projection_vector

Sets the round_state variable to state 1 (*g*). In this state, distances are rounded to the closest grid line.

Example:



Round To Double Grid

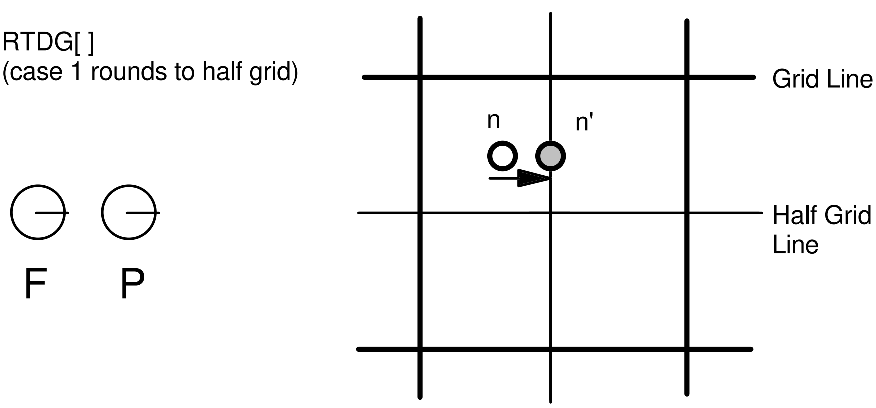
RTDG[ ]

Code Range	0x3D
Pops	–
Pushes	–
Sets	round_state
Affects	MDAP, MDRP, MIAP, MIRP, ROUND
Uses	freedom_vector, projection_vector

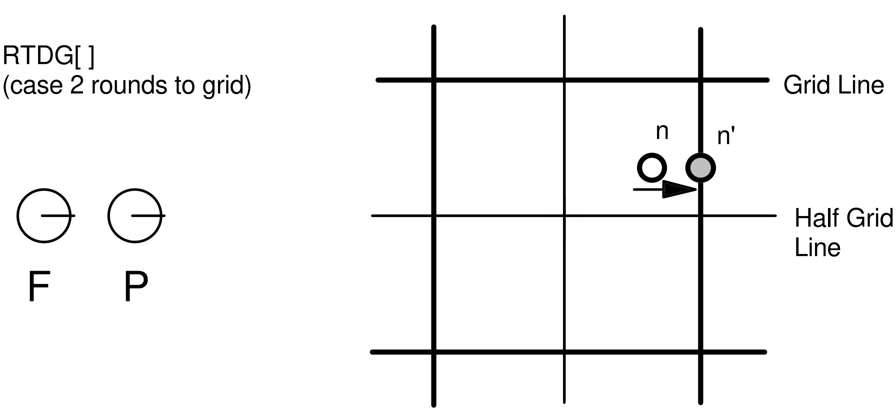
Sets the round_state variable to state 2 (*dg*). In this state, distances are rounded to the closest half or integer pixel.

Example:

RTDG[ ]
(case 1 rounds to half grid)



RTDG[ ]
(case 2 rounds to grid)



Round Down To Grid

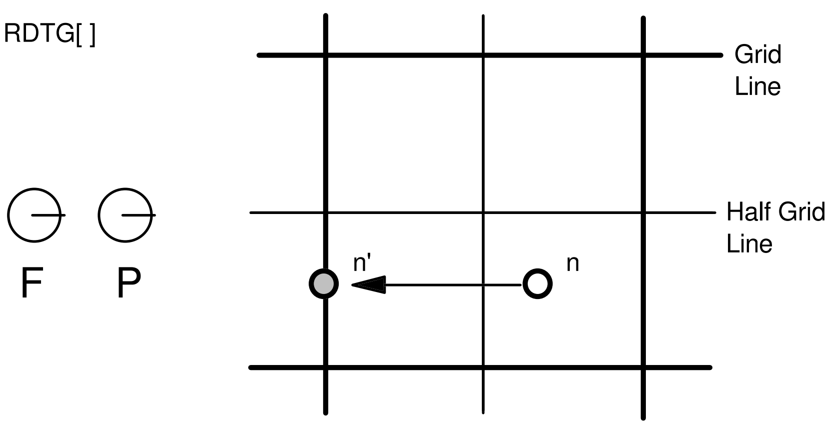
RDTG[ ]

Code Range	0x7D
Pops	–
Pushes	–
Sets	round_state
Affects	MDAP, MDRP, MIAP, MIRP, ROUND
Uses	freedom_vector, projection_vector

Sets the round_state variable to state 3 (*dtg*). In this state, distances are rounded down to the closest integer grid line.

Example:

RDTG[ ]



Round Up To Grid

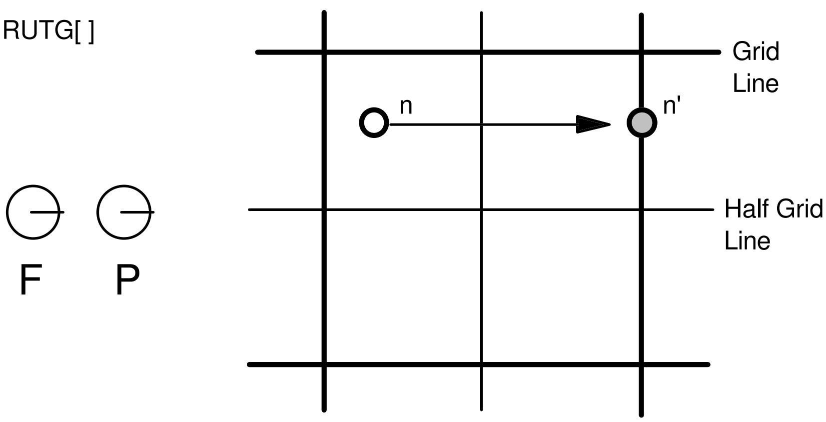
RUTG[ ]

Code Range	0x7C
Pops	–
Pushes	–
Sets	round_state
Affects	MDAP, MDRP, MIAP, MIRP, ROUND
Uses	freedom_vector, projection_vector

Sets the round_state variable to state 4 (*utg*). In this state distances are rounded up to the closest integer pixel boundary.

Example:

RUTG[ ]

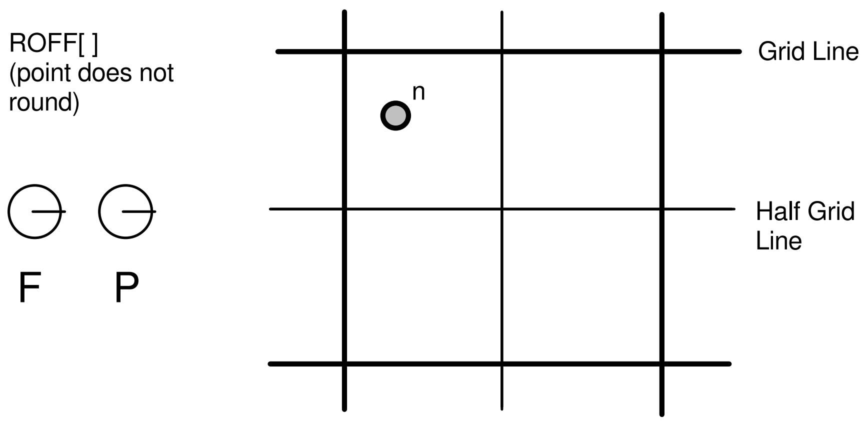


Round OFF

ROFF[ ]	
Code Range	0x7A
Pop	–
Pushes	–
Sets	round_state
Affects	MDAP, MDRP, MIAP, MIRP, ROUND
Uses	freedom_vector, projection_vector

Sets the round_state variable to state 5 (*off*). In this state rounding is turned off.

Example:



Super ROUND

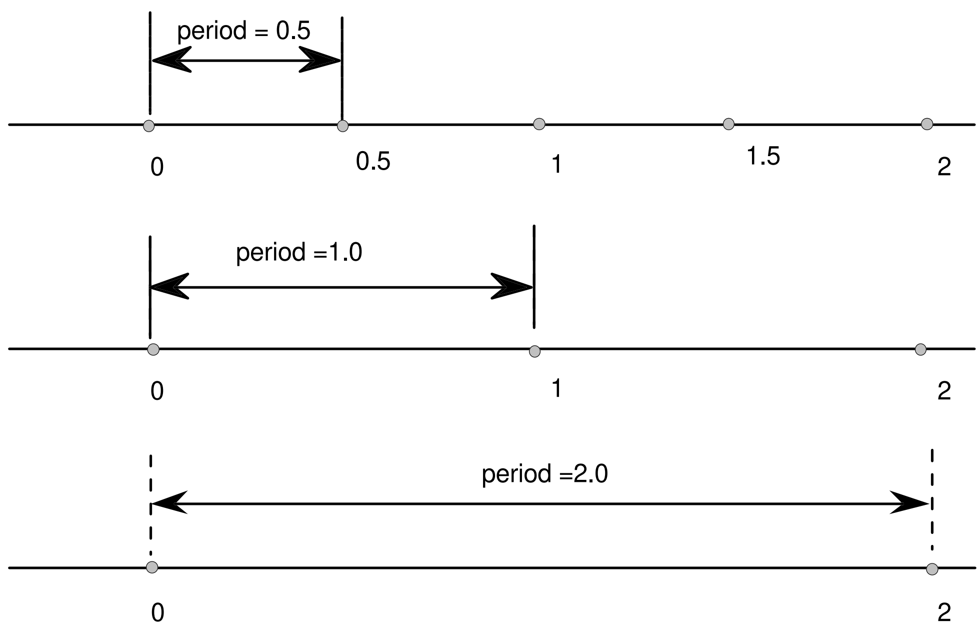
SROUND[ ]

Code Range	0x76
Pops	n: number decomposed to obtain period, phase, threshold
Pushes	–
Sets	round_state
Affects	MDAP, MDRP, MIAP, MIRP, ROUND

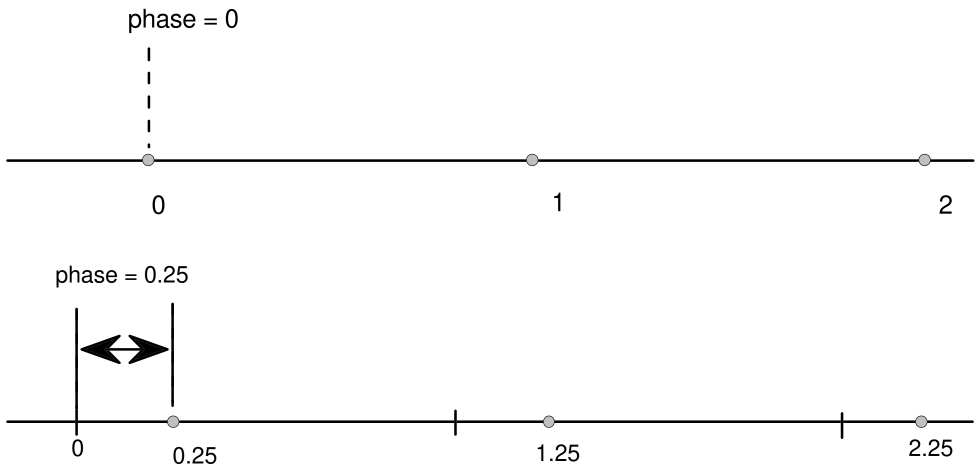
SROUND allows you fine control over the effects of the round_state variable by allowing you to set the values of three components of the round_state: period, phase, and threshold.

More formally, SROUND maps the domain of 26.6 fixed point numbers into a set of discrete values that are separated by equal distances. SROUND takes one argument from the stack, n, which is decomposed into a period, phase and threshold.

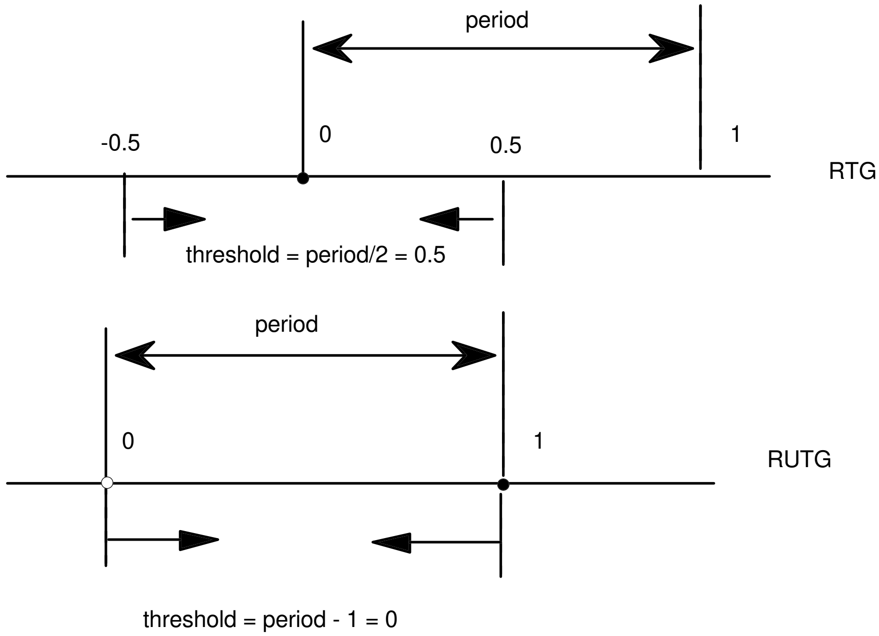
The period specifies the length of the separation or space between rounded values in terms of grid spacing.



The phase specifies the offset of the values from multiples of the period.



The threshold specifies the part of the domain that is mapped onto each value. More intuitively, the threshold tells a value when to "fall forward" to the next largest integer.



Only the lower 8 bits of the argument n are used. For SROUND gridPeriod is equal to 1.0 pixels. The byte is encoded as follows: bits 7 and 6 encode the period, bits 5 and 4 encode the phase and bits 3, 2, 1 and 0 encode the threshold as shown here.

period

0 period = gridPeriod/2

1 period = gridPeriod

2 period = gridPeriod*2

3 Reserved

(continued...)

phase

0 phase = 0

1 phase= period/4

2 phase = period/2

3 phase = gridPeriod*3/4

threshold

0 threshold = period -1

1 threshold = -3/8 * period

2 threshold = -2/8 * period

3 threshold = -1/8 * period

4 threshold = 0/8 * period

5 threshold = 1/8 * period

6 threshold = 2/8 * period

7 threshold =3/8 * period

8 threshold = 4/8 * period

9 threshold = 5/8 * period

10 threshold = 6/8 period

11 threshold = 7/8 * period

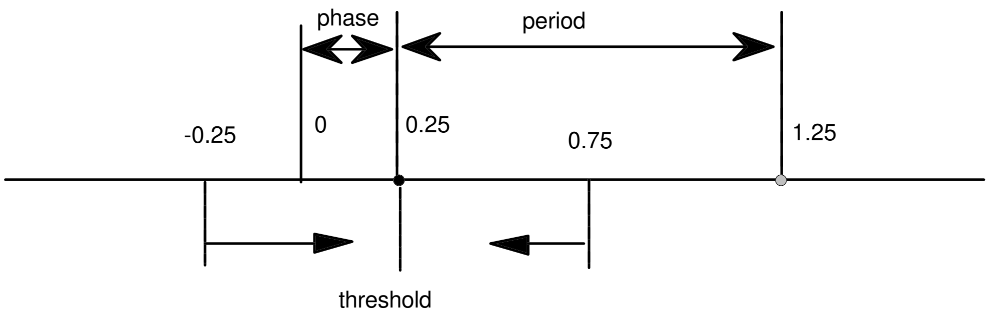
12 threshold = 8/8 * period

13 threshold = 9/8 * period

14 threshold = 10/8 * period

15 threshold = 11/8 * period

For example, SROUND(01:01:1000) maps numbers into the values 0.25, 1.25, 2.25, ... The numbers from -0.25 to 0.75 are mapped into 0.25. The range of numbers [0.75, 1.75) map into 1.25. Similarly, the numbers from [1.75, 2.75) map into the number 2.25 and so on.



Rounding occurs after compensation for engine characteristics, so the steps in the rounding of a number *n* are:

- add engine compensation to *n.*
- subtract the phase from *n.*
- add the threshold to *n.*
- truncate *n* to the next lowest periodic value (ignore the phase).
- add the phase back to *n*.
- if rounding caused a positive number to become negative, set *n* to the positive round value closest to 0.
- if rounding caused a negative number of become positive, set *n* to the negative round value closest to 0.
- the period parameters can have values of 1/2 pixel, 1 pixel, or 2 pixels.
- the phase parameters can have values of 0 pixels, 1/4 pixel, 1/2 pixel, or 3/4 pixel.
- the threshold parameters can have values of -3/8 period, -2/8 period,... 11/8 period. It can also have the special value largest-number-smaller-than-period which causes rounding equivalent to CEILING.

Super ROUND 45 degrees

S45ROUND[ ]

Code Range	0x77
Pops	n: ULONG decomposed to obtain period, phase, threshold (ULONG)
Pushes	–
Sets	round_state
Affects	MDAP, MDRP, MIAP, MIRP, ROUND

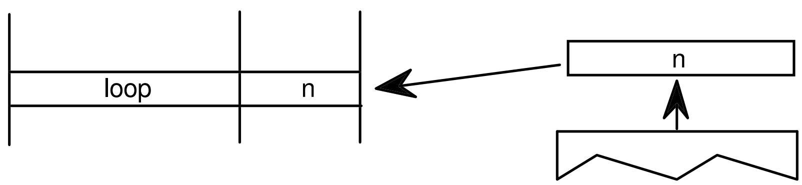
S45ROUND is analogous to SROUND. The gridPeriod is SQRT(2)/2 pixels rather than 1 pixel. It is useful for measuring at a 45 degree angle with the coordinate axes.

Set LOOP variable

SLOOP[ ]

Code Range	0x17
Pops	n: value for loop Graphics State variable (integer)
Pushes	–
Sets	loop
Affects	ALIGNRP, FLIPPT, IP, SHP, SHPIX

Pops a value, n, from the stack and sets the loop variable count to that value. The loop variable works with the SHP[a], SHPIX[a], IP[ ], FLIPPT[ ], and ALIGNRP[ ]. The value n indicates the number of times the instruction is to be repeated. After the instruction executes, the loop variable is reset to 1.



Set Minimum_ Distance

SMD[ ]

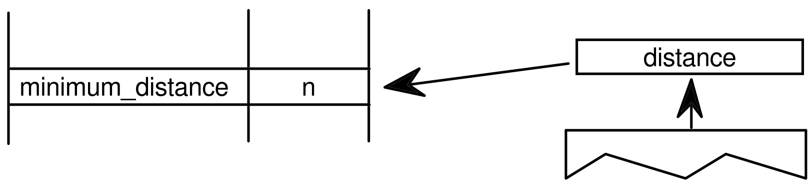
Code Range 0x1A

Pops distance: value for minimum_distance (F26Dot6)

Pushes –

Sets minimum_distance

Pops a value from the stack and sets the minimum_distance variable to that value. The distance is assumed to be expressed in sixty-fourths of a pixel.



INSTRuction execution ConTRoL

INSTCTRL[]

Code Range	0x8E	
Pops	s:	selector flag (int32)
	value:	USHORT (padded to 32 bits) used to set value of instruction_control.
Pushes	–	
Sets	instruction_control	

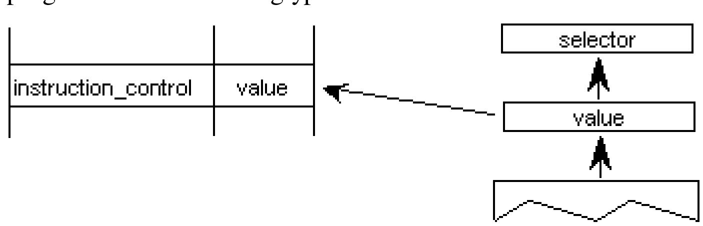
Sets the instruction control state variable making it possible to turn on or off the execution of instructions and to regulate use of parameters set in the CVT program. INSTCTRL[ ] can only be executed in the CVT program.

This instruction clears and sets various control flags in the rasterizer. The selector flag determines valid values for the value argument. The value determines the new setting of the raterizer control flag. In version 1.0 there are only two flags in use:

Selector flag 1 is used to inhibit grid-fitting. If s=1, valid values for the value argument are 0 (FALSE) and 1 (TRUE). If the value argument is set to TRUE (v=1), any instructions associated with glyphs will not be executed. For example, to inhibit grid-fitting when a glyph is being rotated or stretched, use the following sequence on the preprogram:

```
PUSHB[000] 6      /* ask GETINFO to check for stretching or rotation */
GETINFO[]         /* will push TRUE if glyph is stretched or rotated */
IF[]              /* tests value at top of stack */
PUSHB[000] 1      /* value for INSTCTRL */
PUSHB[000] 1      /* selector for INSTCTRL */
INSTRCTRL[]       /* based on selector and value will turn grid-fitting off */
EIF[]
```

Selector flag 2 is used to establish that any parameters set in the CVT program should be ignored when instructions associated with glyphs are executed. These include, for example, the values for scantype and the CVT cut-in. If s=1, valid values for the value argument are 0 (FALSE) and 2 (TRUE). If the value argument is set to TRUE (v=2), the default values of those parameters will be used regardless of any changes that may have been made in those values by the preprogram. If the value argument is set to FALSE (v=0), parameter values changed by the CVT program will be used in glyph instructions.



SCAN conversion ConTRoL

SCANCTRL[ ]

Code Range	0x85
Pops	n: flags indicating when to turn on dropout control mode (16 bit word padded to 32 bits)
Pushes	–
Sets	scan_control

SCANCTRL is used to set the value of the Graphics State variable scan_control which in turn determines whether the scan converter will activate dropout control for this glyph. Use of the dropout control mode is determined by three conditions:

1. Is the glyph rotated?
2. Is the glyph stretched?
3. Is the current setting for ppem less than a specified threshold?

The interpreter pops a word from the stack and looks at the lower 16 bits.

Bits 0-7 represent the threshold value for ppem. A value of FF in bits 0-7 means invoke dropout_control for all sizes. A value of 0 in bits 0-7 means never invoke dropout_control.

Bits 8-13 are used to turn on dropout_control in cases where the specified conditions are met. Bits 8, 9 and 10 are used to turn on the dropout_control mode (assuming other conditions do not block it). Bits 11, 12, and 13 are used to turn off the dropout mode unless other conditions force it. Bits 14 and 15 are reserved for future use.

Bit	Meaning if set
8	Set dropout_control to TRUE if other conditions do not block and ppem is less than or equal to the threshold value.
9	Set dropout_control to TRUE if other conditions do not block and the glyph is rotated.
10	Set dropout_control to TRUE if other conditions do not block and the glyph is stretched.
11	Set dropout_control to FALSE unless ppem is less than or equal to the threshold value.
12	Set dropout_control to FALSE unless the glyph is rotated.
13	Set dropout_control to FALSE unless the glyph is stretched.
14	Reserved for future use.
15	Reserved for future use.

For example

0x0000	No dropout control is invoked
0x01FF	Always do dropout control
0x0A10	Do dropout control if the glyph is rotated and has less than 16 pixels per-em

The scan converter can operate in either a “normal” mode or in a “fix dropout” mode depending on the value of a set of enabling and disabling flags.

SCANTYPE

SCANTYPE[ ]

Code Range	0x8D
Pops	n: 16 bit integer
Pushes	–
Sets	scan_control

Pops a 16-bit integer whose value is used to determine which rules the scan converter will use. If the value of the argument is 0, the fast scan converter will be used. If the value of the integer is 1 or 2, simple dropout control will be used. If the value of the integer is 4 or 5, smart dropout control will be used. More specifically,

if n=0 rules 1, 2, and 3 are invoked (simple dropout control scan conversion including stubs)

if n=1 rules 1, 2, and 4 are invoked (simple dropout control scan conversion excluding stubs)

if n=2 rules 1 and 2 only are invoked (fast scan conversion; dropout control turned off)

if n=3 same as n = 2

if n = 4 rules 1, 2, and 5 are invoked (smart dropout control scan conversion including stubs)

if n = 5 rules 1, 2, and 6 are invoked (smart dropout control scan conversion excluding stubs)

if n = 6 same as n = 2

if n = 7 same as n = 2

The scan conversion rules are shown here:

Rule 1 If a pixel's center falls within the glyph outline, that pixel is turned on.

Rule 2 If a contour falls exactly on a pixel's center, that pixel is turned on.

Rule 3 If a scan line between two adjacent pixel centers (either vertical or horizontal) is intersected by both an on-Transition contour and an off-Transition contour and neither of the pixels was already turned on by rules 1 and 2, turn on the left-most pixel (horizontal scan line) or the bottom-most pixel (vertical scan line). This is “Simple” dropout control.

Rule 4 Apply Rule 3 only if the two contours continue to intersect other scan lines in both directions. That is, do not turn on pixels for ‘stubs.’ The scanline segments that form a square with the intersected scan line segment are examined to verify that they are intersected by two contours. It is possible that these could be different contours than the ones intersecting the dropout scan line segment. This is very unlikely but may have to be controlled with grid-fitting in some exotic glyphs.

Rule 5 If a scan line between two adjacent pixel centers (either vertical or horizontal) is intersected by both an on-Transition contour and an off-Transition contour and neither of the pixels was already turned on by rules 1 and 2, turn on the pixel which is closer to the midpoint between the on-Transition contour and off-Transition contour. This is “Smart” dropout control.

Rule 6 Apply Rule 5 only if the two contours continue to intersect other scan lines in both directions. That is, do not turn on pixels for ‘stubs.’

New fonts wishing to use the new modes of the ScanType instruction, but still wishing to work correctly on old rasterizers that don’t recognize the new modes should:

1. First execute a ScanType instruction using an old mode which will give the best approximation to the desired new mode (e.g. Simple Stubs for Smart Stubs), and then

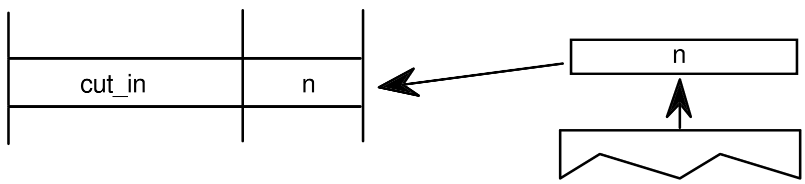
2. Immediately execute another ScanType instruction with the desired new mode.

Set Control Value Table Cut In

SCVTCI[ ]

Code Range	0x1D
Pops	n: value for cut_in (F26Dot6)
Pushes	–
Sets	control_value_cut_in
Affects	MIAP, MIRP

Sets the control_value_cut_in in the Graphics State. The value n is expressed in sixty-fourths of a pixel.



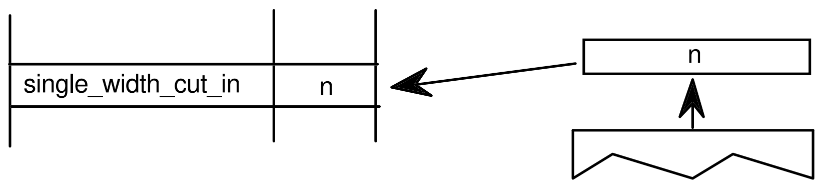
Increasing the value of the cut_in will increase the range of sizes for which CVT values will be used instead of the original outline value.

Set Single_Width_Cut_In

SSWCI[ ]

Code Range	0x1E
Pops	n: value for single_width_cut_in (F26dot6)
Pushes	–
Sets	single_width_cut_in
Affects	MIAP, MIRP

Sets the single_width_cut_in in the Graphics State. The value n is expressed in sixty-fourths of a pixel.



Set Single-width

SSW[ ]

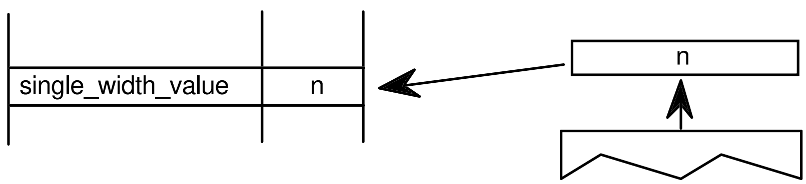
Code Range 0x1F

Pops n: value for single_width_value (FUnits)

Pushes –

Sets single_width_value

Sets the single_width_value in the Graphics State. The single_width_value is expressed in FUnits.



Set the auto_flip Boolean to ON

FLIPON[ ]

Code Range	0x4D
Pops	–
Pushes	–
Sets	auto_flip
Affects	MIRP

Sets the auto_flip Boolean in the Graphics State to TRUE causing the MIRP instructions to ignore the sign of Control Value Table entries. The default auto_flip Boolean value is TRUE.

auto_flip	1

Set the auto_flip Boolean to OFF

FLIPOFF[ ]

Code Range	0x4E
Pops	–
Pushes	–
Sets	auto_flip
Affects	MIRP

Set the auto_flip Boolean in the Graphics State to FALSE causing the MIRP instructions to use the sign of Control Value Table entries. The default auto_flip Boolean value is TRUE.

auto_flip	0

Set Angle _Weight

SANGW[ ]

Code Range 0x7E

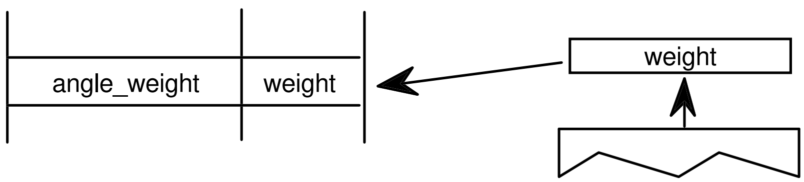
Pops weight: value for angle_weight

Pushes –

Sets angle_weight

SANGW is no longer needed because of dropped support to the AA (Adjust Angle) instruction. AA was the only instruction that used angle_weight in the global graphics state.

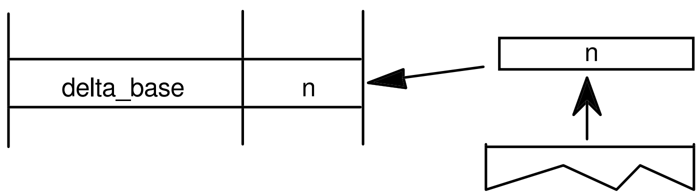
Pops a weight value from the stack and sets the value of the angle_weight state variable accordingly.



Set Delta_Base in the graphics state

SDB[ ]	
Code Range	0x5E
Pops	n: value for the delta_base (ULONG)
Pushes	–
Sets	delta_base
Affects	DELTAP1, DELTAP2, DELTAP3, DELTAC1, DELTAC2, DELTAC3

Pops a number, n, and sets delta_base to the value n. The default for delta_base is 9.

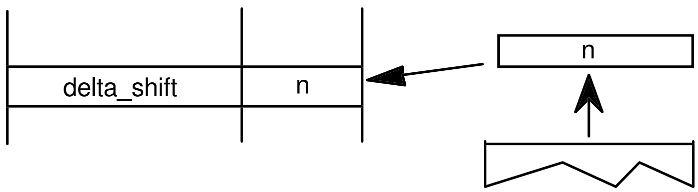


Set Delta_Shift in the graphics state

SDS[ ]

Code Range	0x5F
Pops	n: value for the delta_shift (ULONG)
Pushes	–
Sets	delta_shift
Affects	DELTAP1, DELTAP2, DELTAP3, DELTAC1, DELTAC2, DELTAC3

Sets delta_shift to the value n. The default for delta_shift is 3.



Reading and writing data

The following instructions make it possible to get and to set a point coordinate, to measure the distance between two points, and to determine the current settings for pixels per em and point size.

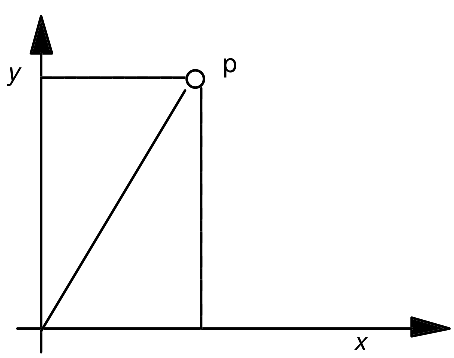
Get Coordinate projected onto the projection_vector

GC[a]	
Code Range	0x46 - 0x47
a	0: use current position of point p
	1: use the position of point p in the original outline
Pops	p: point number (ULONG)
Pushes	value: coordinate location (F26Dot6)
Uses	zp2, projection_vector

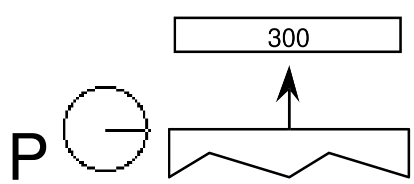
Measures the coordinate value of point p on the current projection_vector and pushes the value onto the stack.

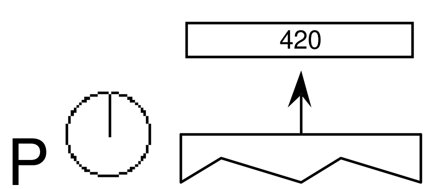
Example

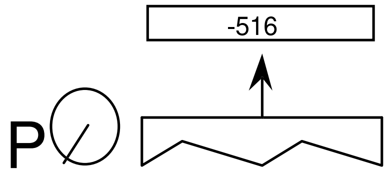
The following example shows that the value returned by GC is dependent upon the current position of the projection_vector. Note that point p is at the position (300,420) in the coordinate grid.



GC[1] 9







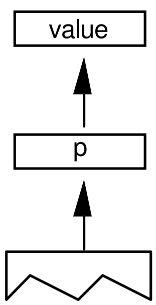
The projection_vector is parallel to the line from (0,0) to (300,420)

Sets Coordinate From the Stack using projection_vector and freedom_vector

SCFS[ ]

Code Range	0x48
Pops	value: distance from origin to move point (F26Dot6)
	p: point number (ULONG)
Pushes	–
Uses	zp2, freedom_vector, projection_vector

Moves point p from its current position along the freedom_vector so that its component along the projection_vector becomes the value popped off the stack.

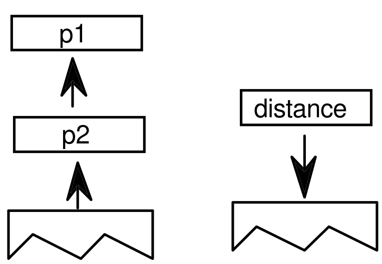


Measure Distance

MD[a]

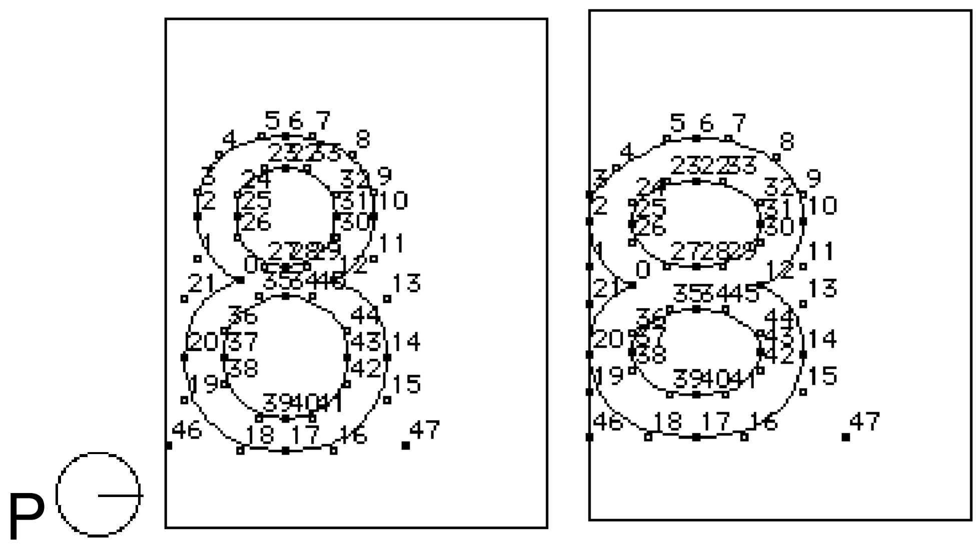
Code Range	0x49 - 0x4A
a	0: measure distance in grid-fitted outline
	1: measure distance in original outline
Pops	p1: point number (ULONG) p2: point number (ULONG)
Pushes	distance (F26Dot6)
Uses	zp1 with point p1, zp0 with point p2, projection_vector

Measures the distance between outline point p1 and outline point p2. The value returned is in pixels (F26Dot6) If distance is negative, it was measured against the projection vector. Reversing the order in which the points are listed will change the sign of the result.



Example:

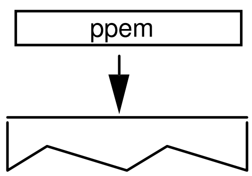
In the illustration below MD[1] between points 25 and 31 will return a smaller value than MD[0] at 10 pixels per em on a 72 dpi device. The difference is due to the effects of grid-fitting which, at this size, stretches out the counter.



Measure Pixels Per EM

MPPEM[ ]	
Code Range	0x4B
Pops	–
Pushes	ppem: pixels per em (ULONG)

This instruction pushes the number of pixels per em onto the stack. Pixels per em is a function of the resolution of the rendering device and the current point size and the current transformation matrix. This instruction looks at the projection_vector and returns the number of pixels per em in that direction.



Measure Point Size

MPS[ ]

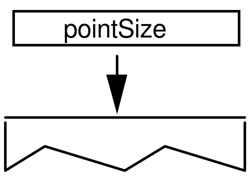
Code Range 0x4C

Pops –

Pushes pointSize: the size in points of the current glyph (F26Dot6)

Pushes the current point size onto the stack.

Measure point size can be used to obtain a value which serves as the basis for choosing whether to branch to an alternative path through the instruction stream. It makes it possible to treat point sizes below or above a certain threshold differently.



12, 18, and 36 point Helvetica g at 72 dpi