Graphite Table Format

1. Introduction

The Graphite font table format is structured in order that a Graphite binary description may be incorporated into a TrueType font. Thus the binary format uses the TrueType table structure, identically to how it is used in a TrueType font. The only difference between using an external file containing Graphite binary information in tables, and inserting the binary information into tables in the font is that tables are considered local to their file and are considered to override those found in the font file. This allows there to be multiple, independent descriptions held in separate files. Those independent descriptions would have to be merged, in a way described in this document, if they were to be held together in the same font file or binary file.

The description consists of a set of table descriptions. The format of a file follows that of a TrueType font containing only those tables pertinent to the description (i.e. for a separate binary description, those tables listed here).

As is standard for all TrueType tables, the data is in big-endian format (most significant byte first).

2. Version

This file describes version 4.0 of the Graphite font table specification. Modifications from previous versions are indicated in the “Version notes” column of the various tables.

3. Tables

This document describes several additional TrueType table types. The “Silf” and “Sile” tables are unique to the needs of Graphite, whilst “Gloc” and “Glat” provide an extended glyph attribute mechanism. The “Feat” table is based very closely on the GX “feat” table. (If necessary the tables could be restructured to be stored inside the single “Silf” table.) In addition, use is made of the “name” table type.

This version of the Graphite format includes the ability to compress “Glat” and “Silf” tables, the extensions to those table provide a compression scheme field permitting up to 32 compression schemes. Currently only 2 schemes are defined: 0 – no compression, and 1 – an LZ4 block decompressor. This is not the LZ4 framing format just the inner block level format without any checksum.

All compressed tables have the same form the original table’s 32bit version number followed by a 32 bit compression header. This consists of the top 5 bits for the scheme and 27 remaining bits for the uncompressed table size. This is then followed by the compression scheme’s data.

Table 1. Compressed table

Type	Name	Description	Version notes
FIXED	version	Uncompressed Table version number	
ULONG:5	scheme	Compression scheme must not be 0	5.0 – added
ULONG:27	full_size	Size of uncompressed table	5.0 – added
BYTE[]	compressed_data	Compression scheme data	5.0 – added

The uncompressed form is the complete table including the version number but with the scheme always set to 0. The remaining 27 bits are available to the uncompressed table.

3.1. Glat

The Glat table type is used for storing glyph attributes. Each glyph may be considered to have a sparse array of, at the most, 65536 16-bit signed attributes. The Glat table is the mechanism by which they are stored.

The Glat table consists of a table header and an array of Glat_entry items. Two formats for the Glat table are typically used. Most fonts will use a version 2 table without Octabox metrics. Those few fonts that have collision avoidance support, will use a version 3 table.

Table 2. Glat version 2

Type	Name	Description	Version notes
FIXED	version	Table version: 00030000	4.0 – 00020000
Glat_entry[]	entries	Glyph attribute entries	

Table 3. Glat version 3

Type	Name	Description	Version notes
FIXED	version	Table version: 00030000	5.0 – 00030000
ULONG:5	scheme	Compression scheme must be 0	5.0 – added
ULONG:28	reserved		5.0 – added
ULONG:1	octaboxes	Octaboxes are present flag	5.0 – added
Glyph_attrs[]	entries	Glyph attribute entries	

For the compressed layout see [comp_table].

Table 4. Glyph_attrs

Type	Name	Description	Version notes
Octabox_metrics	octabox	Octabox metrics, only present if the Glat header indicates	5.0 – added
Glat_entry[]	entries	Glyph attribute entries	

If the octaboxes flag is set in the Glat header then for each per glyph block of data specified by the Gloc table, first set of data includes metrics that approximate the glyph's curves. The approximation uses "octoboxes"—rectangles with corners that may be cut out at an angle of 45 degrees. Each octobox requires 8 values to define. There are metrics for the entire glyph and for a 4x4 approximation grid, resulting in up to 16 sub-boxes. For some glyphs, no sub-box data will be present, in which case the bitmap will be zero. Note that the rectangle for the full glyph is not included here, as the bounding box rectangle is stored elsewhere in the font.

Table 5. Octabox_metrics

Type	Name	Description	Version notes
USHORT	subbox_bitmap	Which subboxes exist on 4x4 grid; bit-index = (y-index*4) + x-index	
BYTE	diag_neg_min	Defines minimum negatively-sloped diagonal	
BYTE	diag_neg_max	Defines maximum negatively-sloped diagonal	
BYTE	diag_pos_min	Defines minimum positively-sloped diagonal	
BYTE	diag_pos_max	Defines maximum positively-sloped diagonal	
Subbox_entry[]	subboxes	One entry per bit in subbox_bitmap	

Note that in the subbox bitmap, bit 3 indicates the presence of the lower right cell, and bit 12 the upper left cell as per this diagram.

Table 6. subbox_bitmap

12	13	14	15
8	9	10	11
4	5	6	7
0	1	2	3

Table 7. Subbox_entry

Type	Name	Description	Version notes
BYTE	left	Left of subbox	
BYTE	right	Right of subbox	
BYTE	bottom	Bottom of subbox	
BYTE	top	Top of subbox	
BYTE	diag_pos_min	Defines minimum positively-sloped diagonal	
BYTE	diag_pos_max	Defines maximum positively-sloped diagonal	
BYTE	diag_neg_min	Defines minimum negatively-sloped diagonal	
BYTE	diag_neg_max	Defines maximum negatively - sloped diagonal	

Following the glyph curve approximation data, the glyph attributes appear. The glyph attributes associated with a particular glyph are identified by number and value. To conserve space, this storage is run-length encoded. Thus a glyph will have a series of Glat_entrys corresponding to each non-contiguous set of attributes. The structure of a Glat_entry is:

Table 8. Glat_entry, version 2 & 3

Type	Name	Description	Version notes
SHORT	attNum	Attribute number of first attribute	4.0 – BYTE to SHORT
SHORT	num	Number of attributes in this run	4.0 – BYTE to SHORT
SHORT	attributes[]	Array of num attributes	

Notice that all glyph attributes are 16-bit signed values. If a 32-bit value is required, then two attributes should be assigned and joined together by the application.

Attribute numbers are application specific.

Note that if the font does not require more than 256 glyph attributes, version 1 of the Glat table will be generated, which is defined as follows.

Table 9. Glat version 1

Type	Name	Description	Version notes
FIXED	version	Table version: 00010000	
Glat_entry[]	entries	Glyph attribute entries	

Table 10. Glat_entry, version 1

Type	Name	Description	Version notes
BYTE	attNum	Attribute number of first attribute	
BYTE	num	Number of attributes in this run	
SHORT	attributes[ ]	Array of num attributes	

3.2. Gloc

The Gloc table is used to index the Glat table. It is structured identically to the loca table type, except that it has a header.

TODO: add a field indicating the number of glyphs in the table (the current dependence on the Silf table is not architecturally clean).

Table 11. Gloc

Type	Name	Description	Version notes
FIXED	version	Table version: 00010000	
USHORT	flags	bit 0 = 1 for Long format, = 0 for short formatbit 1 = 1 for attribute names, = 0 for stripped	
USHORT	numAttribs	Number of attributes	
USHORT/ULONG	locations[ ]	Offsets into Glat table for each glyph; (number of glyph IDs + 1) of these	
USHORT	attribIds[ ]	Debug id for each attribute	

The flags entry contains a bit to indicate whether the locations array is of type USHORT or ULONG. The locations array is identically structured to that of the loca table. There is one entry per glyph and an extra entry to identify the length of the final glyph's attribute entries. Offsets are given to a Glat_entry in the Glat table. The second bit indicates whether there is an attribIds array at the end of this table. If there is, then it contains name IDs for each attribute. If this bit is not set, then there is no array and the table ends after the locations array.

As of version 2 of the Silf table, the values of the breakweight attribute are interpreted as follows:

```
BREAK_WHITESPACE = 10
BREAK_WORD = 15
BREAK_INTRA = 20
BREAK_LETTER = 30
BREAK_CLIP = 40
```

3.3. Feat

Graphite stores features in a table whose format is very similar to the GX feat table. This makes reference to the name table which is use for storing feature names and feature value names.

Table 12. Feat

Type	Name	Description	Version notes
FIXED	version	Table version: 00020000	3.0 – changed from 00010000 to 00020000
USHORT	numFeat	Number of features	
USHORT	reserved		
ULONG	reserved		
FeatureDefn	features[ ]	Array of numFeat features	
Feature-SettingDefn	featSettings[ ]	Array of feature setting values, indexed by offset	

Table 13. FeatureDefn

Type	Name	Description	Version notes
ULONG	id	Feature ID number	3.0 – added
USHORT	numSettings	Number of settings	
USHORT	reserved		3.0 – inserted
ULONG	offset	Offset into featSettings list	
USHORT	flags		
USHORT	label	Index into name table for UI label	

Table 14. FeatureSettingDefn

Type	Name	Description	Version notes
SHORT	value	Feature setting value	
USHORT	label	Index into name table for UI label	

3.4. Silf

The “Silf” table will be used for storing rules and actions for the various types of tables in a rendering description. The structure of the Silf table is:

Table 15. Silf

Type	Name	Description	Version notes
FIXED	version	Table version: 00050000	2.0 – changed to 00020000 3.0 – changed to 00030000 5.0 – changed to 00050000
ULONG:5	scheme	Compression scheme must be 0	5.0 – added
FIXED:27	compilerVersion	Compiler version that generated this font	3.0 – added 5.0 – changed to 27 bits
USHORT	numSub	Number of SIL subtables	
USHORT	reserved		
ULONG	offset[]	Array of numSub offsets to the subtables relative to the start of this table	
SIL_Sub	tables[]	Array of independent rendering description subtables	

For the compressed layout see [comp_table]. Since one TrueType file may hold multiple independent rendering descriptions, each rendering description is described in a subtable. The subtable contains all that is necessary to describe the rendering of one set of writing systems.

Table 16. SIL_Sub

Type	Name	Description	Version notes
FIXED	ruleVersion	Version of stack-machine language used in rules	3.0 – added
USHORT	passOffset	offset of oPasses[0] relative to start of sub-table	3.0 – added
USHORT	pseudosOffset	offset of pMaps[0] relative to start of sub-table	3.0 – added
USHORT	maxGlyphID	Maximum valid glyph ID (including line-	

		break & pseudo-glyphs)	
SHORT	extraAscent	Em-units to be added to the font's ascent	
SHORT	extraDescent	Em-units to be added to the font's descent	
BYTE	numPasses	Number of rendering description passes	
BYTE	iSubst	Index of first substitution pass	
BYTE	iPos	Index of first Positioning pass	
BYTE	iJust	Index of first Justification pass	
BYTE	iBidi	Index of first pass after the bidi pass(must be ⇐ iPos); 0xFF implies no bidi pass	
BYTE	flags	0 - has line end contextuals, 1 - contextuals, 2-4 - space contextuals, 5 - has collision pass	4.0 – added Bit 1
BYTE	maxPreContext	Max range for preceding cross-line-boundary contextualization	
BYTE	maxPostContext	Max range for following cross-line-boundary contextualization	
BYTE	attrPsuedo	Glyph attribute number that is used for actual glyph ID for a pseudo glyph	
BYTE	attrBreakWeight	Glyph attribute number of breakweight attribute	
BYTE	attrDirectionality	Glyph attribute number for directionality attribute	
BYTE	attrMirroring	Glyph attribute number for mirror.glyph (mirror.isEncoded directly after)	2.0 – added;4.0 – used
BYTE	attrSkipPasses	Glyph attribute of bitmap indicating key glyphs for pass optimization	2.0 – added;4.0 – used
BYTE	numJLevels	Number of justification levels; 0 if no justification	2.0 – added
Justification-Level	jLevels[ ]	Justification information for each level.	2.0 – added
USHORT	numLigComp	Number of initial glyph attributes that represent ligature components	
BYTE	numUserDefn	Number of user-defined slot attributes	

BYTE	maxCompPerLig	Maximum number of components per ligature	
BYTE	direction	Supported direction(s)	
BYTE	attCollisions	First of a set of attributes that hold collision flags and constraint box	5.0 - used
BYTE	reserved		
BYTE	reserved		
BYTE	reserved		2.0 – added
BYTE	numCritFeatures	Number of critical features	2.0 – added
USHORT	critFeatures[ ]	Array of critical features	2.0 – added
BYTE	reserved		2.0 – added
BYTE	numScriptTag	Number of scripts this subtable supports	
ULONG	scriptTag[ ]	Array of numScriptTag script tags	
USHORT	lbGID	Glyph ID for line-break psuedo-glyph	
ULONG	oPasses[ ]	Offets to passes relative to the start of this subtable; numPasses + 1 of these	
USHORT	numPseudo	Number of Unicode → pseudo-glyph mappings	
USHORT	searchPseudo	(max power of 2 ⇐ numPseudo) * sizeof(PseudoMap)	
USHORT	pseudoSelector	log2(max power of 2⇐ numPseudo)	
USHORT	pseudoShift	numPseudo - searchPseudo	
PseudoMap	pMaps[ ]	Mappings between Unicode and pseudo-glyphs in order of Unicode	
ClassMap	classes	Classes object storing replacement classes used in actions	
SIL_Pass	passes[ ]	Array of passes	

Each justification level has several glyph attributes associated with it.

This structure was new as of version 2.0.

Table 17. JustificationLevel

Type	Name	Description	Version notes
BYTE	attrStretch	Glyph attribute number for justify.X.stretch	
BYTE	attrShrink	Glyph attribute number for justify.X.shrink	
BYTE	attrStep	Glyph attribute number for justify.X.step	
BYTE	attrWeight	Glyph attribute number for justify.X.weight	
BYTE	runto	Which level starts the next stage	
BYTE	reserved		
BYTE	reserved		
BYTE	reserved		

A pseudo-glyph is a glyph which contains no font metrics (it has a GID greater than the numGlyphs entry in the maxp table) but is used in the rendering process. Each pseudo-glyph has an attribute which is the glyph ID of a real glyph which will be used to actually render the glyph. The pseudo-glyph map contains a mapping between Unicode and pseudo-glyph number:

Table 18. PseudoMap

Type	Name	Description	Version notes
ULONG	unicode	Unicode codepoint	2.0 – changed from USHORT to ULONG
USHORT	nPseudo	Glyph ID of pseudo-glyph	

The ClassMap stores the replacement class information for the passes in this description. Replacement classes are used during substitution where a glyph id is looked up in one class and the glyph ID at the corresponding index in another class is substituted. The difficulty with the storage of such classes is in looking up a glyph ID in an arbitrarily ordered list. One approach is to use a linear search; this is very slow, but is stored very simply. Another approach is to order the glyphs in the class and to store the index against the glyph. Both approaches are supported in the ClassMap table structure:

Table 19. ClassMap

Type	Name	Description	Version notes
USHORT	numClass	Number of replacement classes	
USHORT	numLinear	Number of linearly stored replacement classes	
ULONG	oClass[ ]	Array of numClass + 1 offsets to class arrays from the beginning of the class map	4.0 changed from USHORT
USHORT	glyphs[ ]	Glyphs for linear classes	
LookupClass	lookups[ ]	An array of numClass – numLinear lookups	

The LookupClass stores a fast lookup association between glyph ID and index. Each lookup consists of an ordered list of glyph IDs with the corresponding index for that glyph. The number of elements in the lookup is specified by numIds along with a search Range and shift to initialize a fast binary search engine:

Table 20. LookupClass

Type	Name	Description	Version notes
USHORT	numIDs	Number of elements in the lookup	
USHORT	searchRange	(max power of 2⇐ numIDs)	
USHORT	entrySelector	log2(max power of 2⇐ numIDs)	
USHORT	rangeShift	numIds – searchRange	
LookupPair	lookups[ ]	lookups; there are numIDs of these	

Each element in the lookup consists of a glyphId and the corresponding index in the original ordered list.

Table 21. LookupPair

Type	Name	Description	Version notes
USHORT	glyphId	glyph id to be compared	
USHORT	index	index corresponding to this glyph id in ordered list	

3.5. Pass

Each processing pass consists of a finite state machine description for rule finding, and the actions that are executed when a rule is matched.

Table 22. SIL_Pass

Type	Name	Description	Version notes
BYTE	flags	0-2 - number of collision runs; 3-4 - kerning collisions; 5 - reverse direction pass	5.0 - added bits 0-5
BYTE	maxRuleLoop	MaxRuleLoop for this pass	
BYTE	maxRuleContext	Number of slots of input needed to run this pass	
BYTE	maxBackup	Number of slots by which the following pass needs to trail this pass (ie, the maximum this pass is allowed to back up)	
USHORT	numRules	Number of action code blocks	
USHORT	fsmOffset	offset to numRows relative to the beginning of the SIL_Pass block	2.0 - added; 3.0 - use as fsmOffset
ULONG	pcCode	Offset to start of pass constraint code from start of subtable (**passConstraints[0]**)	2.0 - added
ULONG	rcCode	Offset to start of rule constraint code from start of subtable (**ruleConstraints[0]**)	
ULONG	aCode	Offset to start of action code relative to	

		start of subtable (**actions[0]**)
ULONG	oDebug	Offset to debug arrays (**dActions[0]**); equals 0 if debug stripped
USHORT	numRows	Number of FSM states
USHORT	numTransitional	Number of transitional states in the FSM (length of **states** matrix)
USHORT	numSuccess	Number of success states in the FSM (size of **oRuleMap** array)
USHORT	numColumns	Number of FSM columns
USHORT	numRange	Number of contiguous glyph ID ranges which map to columns
USHORT	searchRange	(maximum power of 2 ⇐ numRange)*sizeof(Pass_Range)
USHORT	entrySelector	log2(maximum power of 2 ⇐ numRange)
USHORT	rangeShift	numRange*sizeof(Pass_Range)-searchRange
Pass_Range	ranges[ ]	Ranges of glyph IDs for this FSM; **numRange** of these
USHORT	oRuleMap[ ]	Maps from success state to offset into ruleMap array from start of array. First item corresponds to state # (numRows – numSuccess); ie, non-success states are omitted. [0xFFFF implies rule number is equal to state number (i.e. no entry in ruleMap) – NOT IMPLEMENTED]
USHORT	ruleMap[ ]	Array of rule numbers corresponding to an success state number
BYTE	minRulePreContext	Minimum number of items in any rule's context before the first modified rule item
BYTE	maxRulePreContext	Maximum number of items in any rule's context before the first modified rule item
SHORT	startStates[ ]	Array of size (maxRulePreContext – minRulePreContext + 1), indicating the start state in the state machine based on how many pre-context items a rule has

USHORT	ruleSortKeys[ ]	Array of **numRules** sort keys, indicating precedence of rules	
BYTE	rulePreContext[ ]	Array of **numRules** items indicating the number of items in the context before the first modified item, one for each rule	
BYTE	collisionThreshold		2.0 - inserted, 5.0 – used
USHORT	pConstraint	Length of passConstraint block	2.0 – added
USHORT	oConstraints[ ]	numRules + 1 offsets to constraint code blocks relative to **rcCode** and start of subtable	
USHORT	oActions[ ]	numRules + 1 offsets to action code blocks relative to **aCode** and start of subtable	
USHORT	stateTrans[ ][ ]	Array of **numTransitional** rows of **numColumns** state transitions.	
BYTE	reserved		2.0 – inserted
BYTE	passConstraints[ ]	Sequences of constraint code for pass-level constraints	2.0 – added
BYTE	ruleConstraints[ ]	Sequences of constraint code for rules	
BYTE	actions[ ]	Sequences of action code	
[1]USHORT	dActions[ ]	Name index for each action for documentation purposes. 0 = stripped. numRules of these	
USHORT	dStates[ ]	Name index for each intermediateFSM row/state for debugging. 0 = stripped. Corresponds to the last numRows – numRules	
USHORT	dCols[ ]	Name index for each state (numRows of these)	

Notice that the ranges array has fast lookup information on the front to allow for the quick identification of which range a particular glyph id is in. Each range consists of the first and last glyph id in the range.

Table 23. Pass_Range

Type	Name	Description
USHORT	firstId	First Glyph id in the range
USHORT	lastId	Last Glyph id in the range
USHORT	colId	Column index for this range

3.5.1. Pass Contents

A pass contains a Finite State Machine (FSM) which is used to match input strings to rules. It also contains constraints for further testing whether a matched string should fire, and it contains the action code to execute against the matched string.

The FSM consists of a set of states. A state consists of a row of transitions between that state and another state dependent upon the next glyph in the input stream. Each state may be an acceptance state, in which case it corresponds to a rule match, or a transition state, in which case the state is on the way to matching a rule, or both. A null state transition is one in which the occurrence of this particular class of the following glyph, will result in no extension of a rule match anywhere, just fail on all further searching. A final state is one in which all its transitions are null transitions.

Note that the stateTrans array only needs to represent transitional states, not final states. Similarly, the oRuleMap array only needs entries for acceptance states (whether final or transitional). For this reason the FSM is set up (conceptually) in the following order: transitional non-accepting states first, followed by transitional accepting states, followed by final (accepting) states.

Note also that because there may be more than one matched rule for a given state, oRuleMap indicates a list of rule indices in the ruleMap array; oRuleMap[i+1] – oRuleMap[i] indicates how many there are for state i.

Normally the start state for an FSM is zero. But for each pass there is the idea of a "pre-context," that is, there are slots that need to be taken into consideration in the rule-matching process that are before the current position of the input stream. If we are very near the beginning of the input, we may need to adjust by skipping some states, which corresponds to skipping the "pre-context" slots that not present due to being prior to the beginning of the input. This is what the maxRulePreContext, minRulePreContext, and startStates items are used for. Specifically, we need to skip the number of transitions equal to the difference between the maxRulePreContext and the current stream position, if greater than zero. The startStates array indicates what the adjusted start state should be. If the current input position is less than minRulePreContext, no rule will match at all.

Rules are matched in order of length, so that longest rules are given precedence over shorter rules. However, the length of some rules may have been adjusted to allow for a consistent "pre-context" for all rules, so the number of matched states in the FSM may not correspond to the actual number of matched items in the rule. For this reason, it is not adequate to simply order rules based on the number of traversed states in the FSM. Rather, rules are given sort keys indicating their precedence, which is based primarily on the length of the rule and secondarily on its original position within the source code.

The FSM engine keeps track of all the acceptance states it passes through on its path to a final state. This results in a list of rules matched by the string sorted by precedence. The engine takes the first rule index off the list and looks up the offset to some constraint code. This code is executed and if the constraint passes, then the action code associated with that offset is executed and the FSM restarts at the returned slot position. If the constraint fails, then the FSM considers the next-preferred rule, tests that constraint, and so forth. If no accepting state is found or all rules fail their constraints, then no rule applies, in which case a single glyph is put into the output stream and the current position advances by one slot.

The action strings are simply byte strings of actions, much like hinting code, but using a completely different language. (See “Stack Machine Commands.doc”.)

3.6. Sile

This table is used in Graphite table files that rely on an external font for rendering of the glyphs. When this table is present, the Graphite file is in effect a minimal font that contains information about the actual font to use in rendering. This information is stored in the Sile table.

This table was added as of version 2. It is not currently being used.

Table 24. Sile

Type	Name	Description
FIXED	version	Table version: 00010000
ULONG	checksum	master checksum (checkSumAdjustment) from the head table of the base font
ULONG	createTime[2]	Create time of the base font (64-bits) from the head table
ULONG	modifyTime[2]	Modify time of the base font (64-bits) from the head table
USHORT	fontNameLength	Number of characters in fontName
USHORT	fontName[ ]	Family name of base font
USHORT	fontFileLength	Number of characters in baseFile
USHORT	baseFile[ ]	Original path and name of base font file

There are four possible situations with regard to the Sile table. The first two are considered normal and the second two pathological.

No Sile table is present. In this case, it is assumed that the Graphite table file is a normal font containing not only the Graphite tables but also the glyphs and metrics needed for rendering.

The base font named in the Sile table is present on the system, and its master checksum and dates match those in the Sile table. In this case, the Graphite tables are read from the Graphite table file, but the glyphs, metrics, and cmap from the base font are what are used for rendering (with the modification performed by the Graphite tables).

The base font named in the Sile table is present, but its master checksum and/or dates do not match those in the Sile table. In this case the base font is used to perform the rendering, but with no Graphite behaviors.

The base font named in the Sile table is not present on the system. In this case the Graphite table file is used for the rendering, with no Graphite behaviors, resulting in square boxes in place of the expected glyphs.

3.7. Sill

This table maps ISO-639-3 language codes onto feature values. Each language code can be a maxmum of 4 ASCII characters (although 2 or 3 characters is what is used by the ISO standard).

This table was added as of version 3.

Table 25. Sill

Type	Name	Description	Version notes
FIXED	version	Table version: 00010000	
USHORT	numLangs	Number of languages supported	
USHORT	searchRange	(maximum power of 2 ⇐ numLangs)	
USHORT	entrySelector	log2(maximum power of 2 ⇐ numLangs)	
USHORT	rangeShift	numLangs - searchRange	
LanguageEntry	entries[ ]	Languages and pointers to feature settings; there are numLang + 1 of these	
LangFeatureSetting	settings[ ]	Feature ID / value pairs	

Each language entry contains a 4-character language code and an offset to the list of features. There is one bogus entry at the end that facilitates finding the size of the last entry. The offsets are relative to the beginning of the Sill table.

The language code is left-aligned with any unused characters padded with NULLs. For instance, the code “en” is represented by the four bytes [101, 110, 0, 0].

Table 26. LanguageEntry

Type	Name	Description	Version notes
BYTE	langcode[4]	4-char ISO-639-3 language code	
USHORT	numSettings	Number of feature settings for this language	
USHORT	offset	Offset to first feature setting for this language	

Table 27. LangFeatureSetting

Type	Name	Description	Version notes
ULONG	featureId	Feature identifer number (matches ID in Feat table)	
SHORT	value	Default feature value for this language	
USHORT	reserved	Pad bytes	

3.8. Sild

This table holds the debug strings for debugging purposes. Since the strings are only used for debugging, they are held somewhat optimised for space over speed and are not considered to be multilingual. Thus strings are considered to be 7-bit ASCII, with a possible extension to UTF-8 at a later stage. The table consists of a sequence of strings each preceded by a length byte. The first string is id 0 and so on to the end of the table.

Note this table has not been implemented.

4. Multiple Descriptions

In the case where multiple descriptions are to be stored in the same set of tables, the following unifications need to occur:

The feature sets must be unified, thus limiting two features with the same name to having the same settings and corresponding values.

The glyph attributes must be unified. This can be done by using different attribute number ranges, or by examining for identical attribute mappings or for non-intersecting attribute mappings.

The use of the name table must be unified to ensure that two features or feature settings do not refer to the same entry in the name table.

Notice that the requirement that any tables declared in an external binary description override the corresponding font table in the font, means that a name table in an external binary description must be complete, including all the strings from the original font.

5. Stack Machine Commands

This document describes the commands that are defined in Graphite's stack machine, which are used to run rules and test their constraints. <offset> is a slot offset relative to the current slot that opcodes act upon. Any opcode with a value outside the range of opcodes listed here (currently 0x00-0x3E) is considered illegal and will cause the font to fail to load.

5.1. General arithmetic operations

Code	Name	Param	Description
00	NOP		Do nothing.
01	PushByte	<byte>	Push the given 8-bit signed number onto the stack.
02	PushByteU	{byte}	Push the given 8-bit unsigned number onto the stack.
03	PushShort	<short>	Push the 2 byte number onto the stack.
04	PushShortU	{short}	Push the 2 byte unsigned number onto the stack.
05	PushLong	<long>	Push the 4 byte number onto the stack. There is no sign extension so no need for an unsigned opcode
06	Add		Pop the top two items off the stack, add them, and push the result.
07	Sub		Pop the top two items off the stack, subtract the first (top-most) from the second, and push the result.
08	Mul		Pop the top two items off the stack, multiply them, and push the result.
09	Div		Pop the top two items off the stack, divide the second by the first (top-most), and push the result.
0A	Min		Pop the top two items off the stack and push the minimum.
0B	Max		Pop the top two items off the stack and push the maximum.
0C	Neg		Pop the top item off the stack and push the negation.

0D	Trunc8	Pop the top item off the stack and push the value truncated to 8 bits.
0E	Trunc16	Pop the top item off the stack and push the value truncated to 16 bits.
0F	Cond	Pop the top three items off the stack. If the first == 0 (false), push the third back on, otherwise push the second back on.
10	And	Pop the top two items off the stack and push their logical and. Zero is treated as false; all other values are treated as true.
11	Or	Pop the top two items off the stack and push their logical or. Zero is treated as false; all other values are treated as true.
12	Not	Pop the top item off the stack and push its logical negation (1 if it equals zero, 0 otherwise.
13	Equal	Pop the top two items off the stack and push 1 if they are equal, 0 if not.
14	NotEqu	Pop the top two items off the stack and push 0 if they are equal, 1 if not.
15	Less	Pop the top two items off the stack and push 1 if the next-to-the-top is less than the top-most; push 0 othewise.
16	Gtr	Pop the top two items off the stack and push 1 if the next-to-the-top is greater than the top-most; push 0 othewise.
17	LessEq	Pop the top two items off the stack and push 1 if the next-to-the-top is less than or equal to the top-most; push 0 otherwise.
18	GtrEq	Pop the top two items off the stack and push 1 if the next-to-the-top is greater than or equal to the top-most; push 0 otherwise

5.2. Rule processing and constraints

Code	Name	Param	Param	Param	Description
19	Next				Move the current slot pointer forward one slot (used after we have finished processing that slot).
1A	NextN	<count>			Not Implemented: Move the

					current slot pointer by the given number of slots (used after we have finished processing the current slot). The count may be positive or negative. Should not be used to copy a range of slots; CopyNext is needed for that.
1B	CopyNext				Copy the current slot from the input to the output and move the current slot pointer forward one slot.
1C	PutGlyph	{outclass}			Put the first glyph of the specified class into the output. Normally used when there is only one member of the class, and when inserting.
1D	PutSubs	<offset>	{inclass}	{outclass}	Determine the index of the glyph that was the input in the given slot within the input class, and place the corresponding glyph from the output class in the current slot. The slot number is relative to the current input position.
1E	PutCopy	<offset			Copy the glyph that was in the input in the given slot into the current output slot. The slot number is relative to the current input position.
1F	Insert				Insert a new slot before the current slot and make the new slot the current one.
20	Delete				Delete the current item in the input stream.
21	Assoc	{count}	<slot-1> …	<slot-count>	Set the associations for the current slot to be the given slot(s) in the input. The first argument indicates how many slots follow. The slot offsets are relative to the current input slot.
22	ContextItem	<offset>	{byte-count}		If the slot currently being tested is not the slot specified by the

				<offset> argument (relative to the stream position, the first modified item in the rule), skip the given number of bytes of stack-machine code. These bytes represent a test that is irrelevant for this slot.
23	AttrSet	{slotattr}		Pop the stack and set the value of the given attribute to the resulting numerical value.
24	AttrAdd	{slotattr}		Pop the stack and adjust the value of the given attribute by adding the popped value.
25	AttrSub	{slotattr}		Pop the stack and adjust the value of the given attribute by subtracting the popped value.
26	AttrSetSlot	{slotattr}		Pop the stack and set the given attribute to the value, which is a reference to another slot, making an adjustment for the stream position. The value is relative to the current stream position. [Note that corresponding add and subtract operations are not needed since it never makes sense to add slot references.]
27	IAttrSetSlot	{slotattr}	{index}	Pop the stack and set the given indexed attribute of the current slot to the value, which is a reference to another slot, making an adjustment for the stream position. The value is relative to the current stream position. [Currently the only indexed slot attributes are component.X.ref.]
28	PushSlotAttr	{slotattr}	<offset>	Look up the value of the given slot attribute of the given slot and push the result on the stack. The slot offset is relative to the current input position.
29	PushGlyph-Attr	{glyphattr}	<offset>	Look up the value of the given glyph attribute of the given slot and push the result on the

					stack. The slot offset is relative to the current input position.
2A	PushGlyph-Metric	{glyph-metric}	<offset>	<level>	Look up the value of the given glyph metric of the given slot and push the result on the stack. The slot offset is relative to the current input position. The level indicates the attachment level for cluster metrics.
2B	PushFeat	{feat}	<offset>		Push the value of the given feature for the current slot onto the stack.
2C	PushAttrTo-GlyphAttr	{glyphattr}	<offset>		Look up the value of the given glyph attribute for the slot indicated by the given slot's attach.to attribute. Push the result on the stack.
2D	PushAttTo-GlyphMetric	{glyph-metric}	<offset>	<level>	Look up the value of the given glyph metric for the slot indicated by the given slot's attach.to attribute. Push the result on the stack.
2E	PushISlotAttr	{slotattr}	<offset>	<index>	Push the value of the indexed slot attribute onto the stack. [The current indexed slot attributes are component.X.ref and userX.]
2F	PushIGlyph-Attr	{glyphattr}	<offset>	<index>	Not Implemented: Push the value of the indexed glyph attribute onto the stack. [Examples of indexed glyph attributes are component.X.box.top, component.X.box.bottom, etc.]
30	PopRet				No more processing is needed for this rule. Pop the top of the stack and return that value. For rule action code, the return value is the number of positions to move the stream position forward (or backward, if the number is negative) for the next rule. For constraint code, the return value is a boolean

					indicating whether the constraint succeeded.
31	RetZero				Terminate the processing and return zero.
32	RetTrue				Terminate the processing and return true (1).
33	IAttrSet	{slotattr}	{index}		Pop the stack and set the value of the given indexed attribute to the resulting numerical value. Not to be used for attributes whose value is a slot reference. [Currently the only non-slot-reference indexed slot attributes are userX.] Not supported in version 1.0 of the font tables.
34	IAttrAdd	{slotattr}	{index}		Pop the stack and adjust the value of the given indexed slot attribute by adding the popped value. Not to be used for attributes whose value is a slot reference. [Currently the only non-slot-reference indexed slot attributes are userX.] Not supported in version 1.0 of the font tables.
35	IAttrSub	{slotattr}	{index}		Pop the stack and adjust the value of the given indexed slot attribute by subtracting the popped value. Not to be used for attributes whose value is a slot reference. [Currently the only non-slot-reference indexed slot attributes are userX.]
36	PushProcState	{byte}			Not Implemented: Pushes the processor state value identifier by the argument onto the stack.
37	PushVersion				Pushes the version of the engine onto the stack as a 32 bit number. The stack holds 32 bit values.
38	PutSubs	<offset>	{inclass-short}	{outclass-short}	Equivalent to PutSubs (0x1D) but with 16-bit class identifiers.

39	PutSubs2			Not Implemented
3A	PutSubs3			Not Implemented
3B	PutGlyph	{outclass-short}		Equivalent to PutGlyph (0x1C) but with 16-bit class identifier.
3C	PushGlyph-Attr	{glyphattr-short}	<offset>	Equivalent to PushGlyphAttr (0x29) but with 16-bit glyph attribute identifier.
3D	PushAttTo-GlyphAttr	{glyphattr-short}	<offset>	Equivalent to PushAttToGlyphAttr (0x2C) but with 16-bit glyph attribute identifier.
3E	BitAnd			Pop the top two items off the stack, perform a bitwise AND, and push the result.
3F	BitOr			Pop the top two items off the stack, perform a bitwise OR, and push the result.
40	BitNot			Pop the top item off the stack, perform a bitwise NOT, and push the result.
41	SetBits	<mask-short>	<value-short>	Pop the top item off the stack, clear the mask bits, set the value bits, and push the result.
42	SetFeat	{feat}	<offset>	Pop a value off the stack and set the given feature on referenced slot to that value. The value is clipped at the maximum permissible value for that feature.

1. Should debug tables go at the end, and be marked via a flag as per Gloc?