Graphite Table Format
Extending TrueType for Graphite
Version 3

Martin Hosken and Sharon Correll,
SIL Non-Roman Script Initiative (NRSI)

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).

Version

This file describes version 3.0 of the Graphite font table specification. See GTF_2_0.doc for a description of version 2.0. Modified fields are highlighted with green in this version; newly added fields are highlighted with yellow.

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.

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, 256 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:

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

Table 1: Glat

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:

Type	Name	Description
BYTE	attNum	Attribute number of first attribue
BYTE	num	Number of attributes in this run
SHORT	attributes[ ]	Array of `num` attributes

Table 2: Glat_entry

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.

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.

Tyoe	Name	Description
FIXED	version	Table version: 00010000
USHORT	flags	bit 0 = 1 for Long format, = 0 for short format bit 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

Table 3: Gloc

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.

NOTE: 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

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.

Type	Name	Description
FIXED	version	Table version: 00020000
USHORT	numFeat	Number of features
USHORT	reserved	
ULONG	reserved	
FeatureDefn	features[ ]	Array of numFeat features
FeatureSettingDefn	featSettings[ ]	Array of feature setting values, indexed by offset

Table 4: Feat

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

Table 5: FeatureDefn

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

Table 6: FeatureSettingDefn

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:

Type	Name	Description
FIXED	version	Table versoin: 00030000
FIXED	compilerVersion	Actual version of the compiler that generated this font
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

Table 7: Silf

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.

Type	Name	Description
FIXED	ruleVersion	Version of stack-machine language used in rules
USHORT	passOffset	offset of oPasses[0] relative to start of sub-table
USHORT	pseudosOffset	offset of pMaps[0] relative to start of sub-table
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	Bit 0: True (1) if there is any start-, end-, or cross-line contextualization; false (0) if line boundaries are irrelevant
BYTE	maxPreContext	Max range for preceding cross-line-boundary contextualization
BYTE	maxPostContext	Max range for following cross-line-boundary contextualization
BYTE	attrPseudo	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	reserved	
BYTE	reserved	
BYTE	numJLevels	Number of justification levels; 0 if no justification
Justification	jLevels[ ]	Justification information for each level.

-Level

Type	Name	Description
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	reserved	
BYTE	reserved	
BYTE	reserved	
BYTE	reserved	
BYTE	numCritFeatures	Number of critical features
USHORT	critFeatures[ ]	Array of critical features
BYTE	reserved	
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	$\log_2$(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

Table 8: SIL_Sub

Each justification level has several glyph attributes associated with it.

Type	Name	Description
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	

Table 9: JustificationLevel

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:

Type	Name	Description
ULONG	unicode	Unicode codepoint
USHORT	nPseudo	Glyph ID of pseudo-glyph

Table 10: PseudoMap

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:

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

Table 11: ClassMap

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:

Type	Name	Description
USHORT	numIDs	Number of elements in the lookup
USHORT	searchRange	(max power of 2<= numIDs) * 4
USHORT	entrySelector	$\log_2$(max power of 2<= numIDs)
USHORT	rangeShift	numIds*4 – searchRange
LookupPair	lookups[ ]	lookups; there are numIDs of these

Table 12: LookupClass

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

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

Table 13:LookupPair

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.

Type	Name	Description
BYTE	flags	bit 0 – This pass makes no change to the slot stream
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
ULONG	pcCode	Offset to start of pass constraint code from start of subtable (*passConstraints[0]*)
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	$\log_2$(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	reserved	
USHORT	pConstraint	Length of passConstraint block
USHORT	oConstraints[ ]	numRules + 1 offsets to constraint code blocks relative to *rcCode* and start of subtable

Martin Hosken and Sharon Correll November 17, 2010 Rev: 5

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	
BYTE	passConstraints[ ]	Sequences of constraint code for pass-level constraints
BYTE	ruleConstraints[ ]	Sequences of constraint code for rules
BYTE	actions[ ]	Sequences of action code
USHORT	dActions[ ]	Name index for each action for documentation purposes. 0 = stripped[1]. 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)

Table 14: SIL_Pass

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.

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

Table 15: Pass_Range

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.

---

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

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".)

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 is new as of version 2.

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

Table 16: Sile

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

1. 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.

2. 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).

3. 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.

4. 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.

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 is new as of version 3.

Type	Name	Description
FIXED	version	Table version: 00010000
USHORT	numLangs	Number of languages supported
USHORT	searchRange	(maximum power of 2 <= numLangs)
USHORT	entrySelector	$\log_2$(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

Table 17: Sill

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].

Type	Name	Description
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 18: LanguageEntry

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

Table 19: LangFeatureSetting

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.

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.

Changes

- 18 March 2003: Changed unicode field of PseudoMap class to ULONG.
- 22 July 2003: Added fields for critical features.
- Sometime: Added pass constraints.
- 20 August 2003: Added Sile table; changed title to Graphite Table Format.
- January 2004: Added justification levels.
- 10 February 2004: Added description of Feat table.
- 31 January 2006: Added Sill table