TECkit Binary Format
Version 3

Jonathan Kew
SIL Non-Roman Script Initiative (NRSI)

Introduction

The following description is based on the “SILtec Binary Format” document written by Martin Hosken for an earlier mapping file format. This data format is designed to provide an appropriate low-level conversion model and file format for the conversion of data from a byte encoding to Unicode and from Unicode to bytes. The model is designed to be easy to implement and to run fairly quickly.

This data format deliberately favours speed and ease of processing over compactness of data. If many mapping descriptions are to be stored, it may be worth applying compression to the stored mapping files; 75% or better compression may often be obtained with typical algorithms.

The description is in two parts: the file format and a processing model for the file format. The latter is important because the binary file is purely a parameterisation of a conversion processing engine and without an understanding of the processing model of that engine, the semantics of the information in the binary file can only be guessed at.

File Format

A conversion file provides a description of the conversion between two encodings. It may hold tables to describe one or both directions of that conversion. Each table has its own type, allowing the addition of further types with time.

Basic Data Types

The basic data types used in this description are:

Name	Length (in bits)	Description
Char	8	Signed 8-bit number
UInt8 or Byte	8	Unsigned 8-bit number
SInt16	16	Signed 16-bit number
UInt16	16	Unsigned 16-bit number
UInt24	24	Unsigned 24-bit number
SInt32	32	Signed 32-bit number
UInt32	32	Unsigned 32-bit number

Basic data types

All numbers are stored in big-endian format, that is with the most significant byte occurring earliest in the file. All other types used are references to other data structures described in this document. Arrays are indicated through the use of [] with the number of elements either entered as a name from the structure being defined, an explicit number, or if the [] are empty, in the description for that structure entry.

Storing Unicode

Unicode values can take a number of forms: UTF-8, UTF-16, UTF-32. In this binary file format Unicode values are always stored directly as Unicode Scalar Values, which are numbers in the range 0x0000 – 0x10FFFF. As all numbers are big-endian, this amounts to saying that Unicode values are stored as UTF32BE. However, as the high 11 bits of a UTF32 value are always zero, these bit positions are in some cases used for other flags, effectively "overlapping" the unused high bits of the Unicode value (if viewed as a 32-bit integer).

File Header

A description file consists of a header and two sequences of conversion tables, one for "forward" conversion and the other for "reverse" conversion. By convention, a legacy byte encoding is considered the "source" and Unicode the "target" of a mapping description; therefore, "forward" conversion maps the legacy encoding to Unicode, and "reverse" conversion maps Unicode back to the legacy encoding. However, the same mapping file format can support other applications, including text transduction entirely within either the legacy (byte) space or Unicode.

Name	Type	Description
type	UInt32	4 byte identifier: 'qMap' (0x714d6170) or 'zQmp' (0x7a516d70)
version	UInt32	16.16 versioning. This version: 0x00030000
headerLength	UInt32	Length of this header
formFlagsLHS	UInt32	Flags describing "source" encoding (LHS of forward mapping)
formFlagsRHS	UInt32	Flags describing "target" encoding (RHS of forward mapping)
numNames	UInt32	Number of entries in the names list
numFwdTables	UInt32	Number of tables in forward mapping pipeline
numRevTables	UInt32	Number of tables in reverse mapping pipeline
nameOffsets	UInt32[numNames]	Offset from start of file to each NameRec
fwdBase	UInt32[numFwdTables]	Offset from start of file to beginning of each table in the forward pipeline
revBase	UInt32[numRevTables]	Offset from start of file to beginning of each table in the reverse pipeline
names	NameRec[numNames]	Name records identifying and describing the mapping

FileHeader

The 4 byte 'type' identifier is included to aid file format identification, as is the version, which consists of two 16-bit values: the first is the major version and the second the minor version.

If the type field contains 'zQmp', this mapping file has been compressed with *zlib*; in this case, the remaining header fields are not present. Instead, the second longword (where the version field should be) is used to hold the size in bytes of the uncompressed file, and the compressed data follows. Uncompressing the data will yield a standard mapping file (including the original type and version fields).

The header length is included for the benefit of software that may wish to read file headers only (e.g., to access names). Note that the name list is considered to be part of the file header, not a separate table.

The source and target encodings each have a longword containing flags that describe features of the encoding. The following flag bits are currently defined:

Name	Value	Description
kFlags_ExpectsNormC	0x00000001	expects fully composed (NFC) text
kFlags_ExpectsNormD	0x00000002	expects fully decomposed (NFD) text
kFlags_GeneratesNormC	0x00000004	generates fully composed text
kFlags_GeneratesNormD	0x00000008	generates fully decomposed text
kFlags_VisualOrder	0x00008000	deals with visually rather than logically ordered text
kFlags_Unicode	0x00010000	this is Unicode rather than a byte (legacy) encoding

Table header flags

If one of the "expects…" flags is set for the source side of a mapping, and the encoding on this side is Unicode, the processing engine is required to normalize the data to the appropriate Unicode normalization form before applying the mapping rules. This allows mapping authors to work entirely in terms of one preferred normalization form.

The "generates…" flags allow the mapping author to declare which normalization form, if any, is produced as output. However, there is no guarantee of the accuracy of these flags, and therefore processes that require a particular normalization form should perform (or explicitly request) a normalization operation after the mapping has been completed.

The header contains offsets (all from the start of the file) to each of the name records and to each table in the forward and reverse mapping pipelines. Name records contain UTF8-encoded strings identifying and describing the encoding and mapping table. There may be an arbitrary number of name strings, identified by 16-bit ids; as a bare minimum, the LHS and RHS Name strings should be included in all files.

Name	Type	Description
nameID	UInt16	name ID for this string
nameLength	UInt16	length of name string in bytes
data	Byte[nameLength]	the name string
reserved	Byte[0 or 1]	zero padding to a 16-bit boundary

NameRec

The following name ID values are defined at this time; more may be defined in the future should the need arise.

Name	Value	Description
kNameID_LHS_Name	0	“source” or LHS encoding name
kNameID_RHS_Name	1	“target” or RHS encoding name (Unicode, in the normal application of this format for legacy/Unicode conversion)
kNameID_LHS_Description	2	LHS encoding description
kNameID_RHS_Description	3	RHS encoding description
kNameID_Version	4	version of the mapping description
kNameID_Contact	5	contact information (e.g., a mailto: URL) for the mapping author or maintainer
kNameID_RegAuthority	6	organization responsible for the encoding
kNameID_RegName	7	name and version of the mapping, as known to that authority
kNameID_Copyright	8	copyright information

Name ID values

The rest of the file contains the conversion tables for the forward and reverse mapping pipelines. Each table starts on a 4-byte boundary; zero pad bytes are added where necessary to ensure this.

Each pipeline consists of a chain of any number of conversion tables. A table may operate entirely in the “byte” (legacy) encoding space, mapping byte values (or sequences) to new byte values (or sequences); entirely in the Unicode space, mapping Unicodes to other Unicodes; or it may function as an interface between the two encoding spaces, mapping bytes to Unicodes or vice versa. There are thus four possible varieties of table, depending on the input and output encoding spaces. The output of each table in the chain becomes the input to the next table, and therefore the output encoding space of each table must match the input of its successor.

Conversion Tables

The conversion tables, no matter which code space(s) they operate in, share a common format. The format is designed to allow one-to-one mappings to be executed very efficiently, while also allowing many-to-many mappings and the added complexity of pre and post environment contextual constraints.

Name	Type	Description
type	UInt32	4-byte identifier for table type: 0x422D3E42 (‘B->B’) byte/byte mapping 0x422D3E55 (‘B->U’) byte/Unicode mapping 0x552D3E42 (‘U->B’) Unicode/byte mapping 0x552D3E55 (‘U->U’) Unicode/Unicode mapping 0x4E464320 ('NFC ') apply NFC normalization to Unicode data 0x4E464320 ('NFD ‘) apply NFD normalization to Unicode data
version	UInt32	16.16 version current version: 0x00030000
length	UInt32	total length of this table
flags	UInt32	flags specifying optional features of the table: 0x00000001: Unicode characters >0xffff supported 0x00000002: DBCS support (B->x tables only)
pageBase	UInt32	offset from start of table to page table (U->x tables) or dbcsPage table (B->x tables with dbcs support)
lookupBase	UInt32	offset from start of table to lookup table(s)
matchClassBase	UInt32	offset from start of table to match class definitions
repClassBase	UInt32	offset from start of table to replacement class definitions
stringListBase	UInt32	offset from start of table to string rule lists
stringRuleData	UInt32	offset from start of table to string rule data block
maxMatch	UInt8	maximum number of input characters matched by a rule
maxPre	UInt8	maximum number of input characters matched by pre environment
maxPost	UInt8	maximum number of input characters matched by post environment
maxOutput	UInt8	maximum number of output characters generated by a rule
replacementChar	UInt32	replacement for unmapped characters

TableHeader

The TableHeader contains a ‘type’ field similar to that in the FileHeader. If the type is a normalization table (NFC or NFD), then no further fields are present; the table simply represents a normalization operation to be performed on the Unicode data. Otherwise, for actual mapping tables, the TableHeader is a fixed length header, with type and version information in the same form as for a file header.

Flags for optional features

- Supplementary character support: If the flag bit indicating support for supplementary-plane Unicode characters is set, several parts of the table are affected:
  - If the input to the table is Unicode, the plane-mapping tables are included (see below), enabling characters >0x00ffff to be mapped; without this flag, all characters >0x00ffff would map to the default value.
  - Unicode match and/or replacement classes contain 32-bit Unicode scalar values; without this flag, they contain 16-bit values and can thus only contain BMP characters.

  (This flag is meaningless in a B->B table.)
- DBCS support: this flag indicates that a dbcsPage table is present, enabling direct lookup of double-byte characters. Note that the double byte character set support in this version of the mapping table is limited: two-byte characters cannot be used in classes or as negated elements in string rules.

Class Definitions

Within match strings (and environments) it is possible to make reference to a class of codes. The classes are referenced by class number, and defined in the match and replacement class definition sections of the table. The class elements may be 8, 16, or 32-bit values, depending on the subtable type and whether the class is defined for the match or replacement side. In byte space, classes always have 8-bit values; in Unicode space, 16 or 32 bits are used depending on the supplementary-plane flag in the subtable header.

Name	**Type**	**Description**
ClsOffset	UInt32[number of classes]	Offset relative to start of Class table to each class definition
ClassDefns	ClassDefinition[number of classes]	The class definitions

ClassTable

Name	**Type**	**Description**
NumElements	UInt32	Number of elements in the class
Elements	ElementType[numElements]	The elements of the class in rising numeric order (where ElementType is UInt8, UInt16, or UInt32; see text)

ClassDefinition

Lookup Table

The lookup table provides mapping information for each possible code input. The exact format of the lookup table depends whether the input to the subtable consists of bytes or Unicodes, but in all cases the function is to map each input code to a Lookup value.

Byte lookup table

The table consists of an array of 256 lookups. If the DBCS support flag is set in the subtable header, then this table consists of a two-layer table. The DBCSpage array maps the first byte to an index into an array of 256-entry Lookup arrays, and the next byte is used as an index into the Lookup array. If the value of an entry in the DBCSpage array is 0, then it indicates the single byte situation (since few double byte character sets are double byte for all their values), whereby the first byte itself is re-used as the index into the first Lookup array, and only one code is consumed rather than two.

Name	**Type**	**Description**
DBCSpage	UInt8[256]	Only present if DBCS support flag set
Lookups	Lookup[256][]	Lookup for each possible input code. One 256-element array is present for each value from zero to the maximum DBCSpage entry. In the absence of DBCS support, only a single 256-element array is present.

LookupTable for tables with Byte input

Unicode lookup table

The lookup table for Unicode input works in two stages, using a “paged mapping” system to map each Unicode character of interest to a 16-bit character index, which is then used for the actual mapping to a Lookup.

There are two approaches to the mapping process. By default, each character is considered as a single 16-bit Unicode value from the BMP. This type of table can only map characters from the BMP; any supplementary-plane characters will map to default output values. When non-BMP characters are to be mapped, the surrogate support flag must be set in the table header.

Mapping from Unicode value to character index is a multi-stage process. First, if surrogate support is enabled, there is a table mapping from the plane (BMP = 0, supplementary planes = 1-16) to the appropriate page table. To save having empty page tables, a plane mapping entry of value 0xFF indicates that all codes in that plane should be mapped to the unknown character specified in the conversion table header. If surrogate support is not enabled, then there is no plane mapping table and there is only one page table.

At the point of having a pageMap index, there are 16-bits of unprocessed data left in the character code. The most significant 8-bits of this remaining data is used to index into the particular pageMap (either the single pageMap for non-surrogate support or the pageMap that the appropriate planeMap indexes). The pageMap contains an index to a characterMap. The least significant 8-bits are then used to index into the characterMap to lookup the particular character index.

Name	Type	Description
PlaneMap*	UInt8[17]	Index into Page maps array. 0xFF indicates character index of unknown for all characters in the plane
numPageMaps*	UInt8	Number of 256-element arrays in PageMap
padding*	UInt8[2]	Reserved, padding to 4-byte boundary
PageMap	UInt8[256][]	Array of 256 element arrays. Each entry is an index into the characterMap arrays. There are sufficient arrays of 256 elements in the pageMap to account for all the values in planeMap (excluding 0xFF). If supplementary plane character support is not enabled, there is a single 256-element array.
CharacterMap	UInt16[256][]	Array of 256 character index arrays. Each element maps from the least significant 8 bits of a code to a character index. There are sufficient of these 256 element arrays to account for all the values in the pageMaps.
Lookups	Lookup[]	Lookup for each possible input code.

** field only present if supplementary-plane support bit set in conversion table header flags byte*

LookupTable for tables with Unicode input

Whether the table input is bytes or Unicode characters, the LookupTable ultimately maps each input character to a Lookup value that specifies how it should be mapped. Each lookup is exactly 4 bytes long, but may be interpreted in several ways:

Name	Type	Description
type	UInt8	type of Lookup: 0xFF: there is a string rule list for this character 0xFE: illegal DBCS trailing byte 0xFD: unmapped character: use default mapping if ((type & 0xC0) == 0x80): extended string rule list: add 256 * (type & 0x3F) to ruleCount 0x00-0x03: direct lookup; *type* is number of output chars
ruleCount	UInt8	number of string rules for this character
ruleIndex	UInt16	index into stringList of start of rule list for this input character

Lookup showing interpretation of type field, and additional fields for string rule lookups

Name	Type	Description
type	UInt8	count of output characters (0-3)
data	UInt8[3]	output byte values

Lookup for direct mapping with Byte output

Name	Type	Description
type	UInt8	count of output characters (0 or 1)
usv	UInt24	Unicode scalar value

Lookup for direct mapping with Unicode output

If the *type* field is 0xFF, then *ruleCount* and *ruleIndex* describe a list of string rules that must be tested at the current character. If *type* is 0xFE, this lookup represents an illegal DBCS sequence; the mapping process should be restarted, treating the lead byte as a single-byte value. If *type* is 0xFD, this is an unmapped character; for Byte-Byte or Unicode-Unicode tables, the character will be copied to the output, and for tables mapping across the Byte/Unicode boundary, the default output character will be generated.

If the high two bits of the *type* field are 10 (binary), i.e., (*type* & 0xC0) == 0x80, then the lookup refers to an "extended string rule list". In this case, the lower 6 bits of the *type* field are used to extend the *ruleCount* field from 8 to 14 bits, providing support for up to 16K rules per initial character code.

Otherwise, the type field is interpreted as a count of output characters, and the remaining three bytes of the Lookup are reinterpreted as a single USV or an array of three Bytes, depending on the output code space. Thus, a single Unicode value (including supplementary-plane values) or a sequence of up to three bytes may be mapped directly from the Lookup.

String Rule lists and data

Input characters that cannot be mapped directly in the Lookup end up pointing to a string rule list—a list of StringRules that must be tested in order, looking for a match at the current location. The table header includes an offset to the StringRuleLists array; Lookups in turn contain indexes (not byte offsets!) into this array and counts of the number of rules for the starting character concerned.

Name	Type	Description
ruleOffset	UInt32[]	offset from stringRuleData to the beginning of this string rule

StringRuleLists

All the string rules are concatenated into the StringRuleData area of the table, and pointed to by the entries in the StringRuleLists array. A string rule is a variable-length structure:

Name	Type	Description
matchLength	UInt8	count of elements in match string
postLength	UInt8	count of elements in post-context
preLength	UInt8	count of elements in pre-context
repLength	UInt8	count of elements in replacement string
matchString	MatchElem[matchLength]	string (or regular expression) to match
postContext	MatchElem[postLength]	post-context (following environment)
preContext	MatchElem[preLength]	pre-context (preceding environment), stored in reverse order
repString	RepElem[repLength]	replacement string

StringRule

Match elements

A string rule consists of the match string and environments to match and replace. Rules for each input character are listed in priority order, starting with the longest potential match string, and with the longest potential context where match string lengths are the same.

The match and context strings consist of sequences of codes that are more than just characters. Each code consists of four bytes, with various possible interpretations for different types of match element.

Name	Type	Description
repeat	UInt8	min and max repeat counts packed into high and low halves of the byte
type	UInt8	negate flag and element type: 0x80: (flag bit) negate 0x40: (flag bit) non-literal 0x3F: (mask) match type if non-literal; see values below
variable	UInt16	various interpretations depending on element type

MatchElem standard fields

The first byte of the MatchElem, flags.repeat, is always interpreted as a repeat count, except in the case of OR and ENDGROUP elements, where it is ignored. It contains minimum and maximum repeat counts, packed into the high and low halves of the byte. Thus, an element can specify repeat counts in the range 0 to 15. (Unbounded repeat is not available.) An element which must occur exactly once (the most common case in simple string rules) has a value of 0x11 for this byte; an optional element which may occur zero or one times has a value of 0x01.

The second byte, flags.type, specifies the type of match element. Its high bit is a *negated* flag, indicating that the success or failure of the matching of this element must be reversed. The next bit is a flag indicating whether the element contains a literal character code or is a "special" element. If the NonLit bit is clear, then the element contains a single byte or USV code (depending on the input code space) at value.byte.data or value.usv.data. If it is set, then the remaining bits of flags.type contain the element type:

Name	Value	Description
Class	0x01	class match
BeginGroup	0x02	beginning of a group (bracketed sequence)
EndGroup	0x03	end of a group
OR	0x04	alternation (only permitted within group)
ANY	0x05	match any single character (not beginning or end of data)
EOS	0x06	match beginning or end of data

MatchElem (flags.type & 0x3F) values

For the different element types, the remaining bytes (three and four) of the MatchElem have different interpretations:

Name	Type	Description
repeat	UInt8	min and max repeat counts
type	UInt8	negate flag and element type
dNext	UInt8	offset to following OR or EndGroup element
dAfter	UInt8	offset to element following the EndGroup that matches this BeginGroup

MatchElem for a BeginGroup element

Name	Type	Description
repeat	UInt8	min and max repeat counts
type	UInt8	negate flag and element type
dNext	UInt8	offset to following OR or EndGroup element (for OR only)
dStart	UInt8	reverse offset to BeginGroup element

MatchElem for an OR or EndGroup element

Name	Type	Description
repeat	UInt8	min and max repeat counts
type	UInt8	negate flag and element type
classIndex	UInt16	index of character class to match

MatchElem for a ClassMatch element

Name	Type	Description
repeat	UInt8	min and max repeat counts
type	UInt8	negate flag and element type
reserved	UInt8	reserved (set to zero)
data	UInt8	byte character to match

MatchElem for a literal code (byte input)

Name	Type	Description
repeat	UInt8	min and max repeat counts
data	UInt24	USV character to match (must mask with 0x001FFFFF)

MatchElem for a literal code (Unicode input)

Replacement elements

The replacement string in a StringRule is also made up of 4-byte elements, though these have fewer variants than the elements of the match string and context. The first byte acts as a *type* field identifying the particular kind of element. For literal character codes, no further information besides the actual character to output is required:

Name	Type	Description
type	UInt8	element type: 0x00: kRepElem_Literal
value	UInt24	literal character code

RepElem (for literal elements)

For Class and Copy elements, there is a field giving the index (zero-based) of the corresponding item in matchString. In the case of Class, this must be a Class match item, and the output character will be member of the replacement class that corresponds (positionally, within the class definition) to the matched member of the match class. Copy elements are only valid in Byte-Byte or Unicode-Unicode tables, and may correspond to individual elements of the match string or to BeginGroup elements, in which case the text matched by the entire group is copied. Finally, the Unmapped element generates the table's default output.

Name	Type	Description
type	UInt8	element type: 0x01: kRepElem_Class 0x07: kRepElem_Copy 0x0F: kRepElem_Unmapped
matchIndex	UInt8	index of corresponding item in matchString (for kRepElem_Class and kRepElem_Copy)
repClass	UInt16	replacement class index (only for kRepElem_Class)

RepElem (for non-literal elements)

Processing model

This section is intended to make explicit that which is implicitly assumed in the above description. It is hoped that this section will answer the most common questions of an implementer.

Conversion model

The presumed processing model for an engine supporting this binary format is not that it should do in-place editing to convert one string into another, but that it create a new output string by walking through the input string. Thus all tests, including pre-environments, will be made on the input string, and previous output can make no difference to future tests. This model may be described as a “match and generate” rather than a “match and replace” model.

Insertion rules

It is possible for the match string in a rule to be of zero length, or to match no input characters because all elements have minimum repeat counts of zero. This represents an “insertion rule” that generates output without consuming any input. Since no input is consumed, the same rule would match again, ad infinitum.

To prevent this, we add the stipulation that once an insertion rule (any rule that matches without consuming any input) has been executed at a particular point in the input, no further insertion rules will apply there. Processing continues with the next rule in the list for the current input character, but any insertion rule that would match is skipped.

Insertion rules with no following context constraint can be very inefficient to process, as they must be tested at every character position. The format and engine are designed to work efficiently for simple mappings by direct table lookup of the current input character, and this is lost when unconstrained insertion rules are present.

It is unclear whether there is a significant need for insertion rules in real-life encoding conversion scenarios, and at the time of writing the TECkit compiler will in fact refuse to compile such tables. This issue can be considered further if examples showing a need for insertions are found.

Default rules

It is possible that the Lookup for a character will point to a string rule list, but no match is found as the rules are tested. In this case, the default output for the table (a copy of the input character, or the default defined in the table header, depending whether the table is operating in a single code space or across the byte/Unicode boundary) is generated, and a single input character consumed.

This behavior also applies if no other rule matches after an insertion rule has been applied (see above).