Beyond UTR22:
complex legacy-to-Unicode mappings

Jonathan Kew
SIL Non-Roman Script Initiative (NRSI)
jonathan_kew@sil.org

Abstract

While Unicode was designed to facilitate easy mapping of data in most industry-standard legacy encodings, there are many "custom fonts" in use around the world that effectively represent additional, non-standard encodings. In some cases, these may encode many presentation forms, such as variants of overstriking accents, or characters encoded in an order that does not match Unicode.

The standard format for mapping descriptions presented in UTR22 is not adequate to support such encodings, especially when round-trip conversion is required. Likewise, tools based on this standard are not powerful and flexible enough.

This paper illustrates the issues by considering the types of complexity seen in a variety of custom legacy encodings, describes a processing model we have developed to address such data conversion needs, and shows how this can be applied to help users migrate from legacy systems with custom fonts to standard, Unicode-based systems.

In conclusion, I will suggest how the UTR22 mapping description format might be extended to support complex mapping processes.

1. Background

Some day, all our text data will be encoded using a single standard, rather than the multitude of encodings in use today. Working in an organization that deals with many hundreds of languages, written in many different scripts, makes this hope particularly attractive. However, in order for this to be achieved, a vast amount of existing data in a variety of legacy encodings must be converted into Unicode. Moreover, for a considerable time, users will be working with both Unicode-based systems and legacy systems that use byte encodings; not all systems will be updated simultaneously. Therefore, there is a requirement not just to map legacy data to Unicode, but also to map Unicode data back to legacy encodings, to permit two-way interchange.

For legacy data that uses industry-standard encodings, such as well-known Windows codepages, ISO standards, or classic Mac OS encodings, mapping to and from Unicode is usually straightforward. Indeed, Unicode deliberately provides one-to-one mappings for the character repertoires of most widespread encodings, allowing transcoding by means of simple table lookup. (Overlooking, for this discussion, issues such as the semantic overloading of certain characters in

some encodings, the problem of choosing among duplicate characters, and differences in what may be considered a "character" in different standards.)

However, users such as linguists working with lesser-known or not-yet-standardized languages and orthographies, or indeed everyday users of such languages, often find that legacy standards lack the characters they need. They have therefore resorted to the creation and use of "custom fonts" which do not conform to any established standard, using tools such as Fontographer or SIL's Encore font system.[1] Some such fonts have been widely distributed, and even marketed commercially—see, for example, SIL's fonts for IPA, classical Greek and Hebrew, and textual apparatus, and products such as many of those sold by Linguist's Software[2]—while others have been created by individuals for their own private use. While this paper will use several SIL fonts as examples, many similar cases can be found in archives such as Dr. Berlin's Foreign Font Archive[3] and the Yamada Language Center Font Archive.[4]

Although what the user with "special character" requirements is primarily looking for is a way to display the desired glyphs, each such custom font actually represents a new character encoding. When a user thinks, "I'll just replace the '@' sign in this font with an IPA glottal stop symbol", he has in effect defined an encoding that differs in this respect from ASCII, codepage 1252, MacRoman, or whatever starting point was chosen. To convert this user's 8-bit data to Unicode therefore requires a new mapping.[5]

In a simple case like this, the new mapping is easy to define and to implement; there is still a one-to-one correspondence between the 8-bit legacy codes and Unicode characters. Using a simple text notation for byte ⇔ Unicode mappings, we might say:

```
; ...63 similar lines for codes 0x00 to 0x3E...
0x3F      <>     U+003F        ; question mark
0x40      <>     U+0294        ; glottal stop (replacing ASCII '@' sign)
0x41      <>     U+0041        ; uppercase A
; ...etc... all the way to 0xFF
```

This is similar to the standard mapping files available from the Unicode web site, and can readily be expressed in the UTR22 mapping description format.[6] Writing mappings in this way, we can by convention refer to the mapping from the legacy byte encoding to Unicode as "forward", and from Unicode to the legacy encoding as a "reverse" mapping.

Mapping descriptions of this form suggest a simple model for the implementation of an encoding converter: the input data is processed in sequence, looking up each code and generating a corresponding output code. Mapping rules may specify sequences of codepoints, not just single characters, in which case the longest available rule should be applied at each step through the input data. Supporting sequences requires an implementation that goes beyond simple lookup tables, but the overall processing model remains extremely straightforward.

---

[1] See http://www.sil.org/computing/fonts/encore.html.

[2] See http://www.linguistssoftware.com/.

[3] See http://user.dtcc.edu/~berlin/fonts.html.

[4] See http://babel.uoregon.edu/yamada/fonts.html.

[5] It should be noted that this paper does not consider issues of multi-byte character sets, such as have often been used for Far Eastern languages. Mappings between industry-standard MBCS systems and Unicode are dealt with adequately by many current tools, and the custom fonts created by users looking for "special character" solutions are typically based on single-byte encodings. However, custom MBCS encodings could be handled by a simple extension of the approach described here, should this prove necessary.

[6] See http://www.unicode.org/unicode/reports/tr22/ for details.

In this case, the simple mapping for each legacy codepoint can trivially be reversed; thus, these are bidirectional rules, and express both the forward and reverse mapping. However, there are many cases where—often with good reason, given the constraints of the available systems—users have created encodings where the mappings are less straightforward. This paper will discuss examples that illustrate the requirement for two key enhancements to the model: contextual constraints used to select among possible mappings for a codepoint, and support for reordering based on matching patterns of character classes. Implementation of these features also makes encoding conversion a multi-stage process.

2. Contextual mappings

Where a large number of diacritics are required, and can be used with many different base characters, it is impossible for a simple 8-bit encoding, as used by much legacy software, to include precomposed forms for all *base+diacritic* combinations. In these cases, implementers typically resort to zero-width overstriking diacritics. These are comparable to Unicode's combining marks, and can often be mapped directly to characters from the U+03xx block.

However, because legacy systems often lack any support for "smart" positioning of diacritics, it is common to encode several variants of each diacritic, with varying position relative to the glyph origin. The user then chooses the version of the diacritic with the most appropriate positioning for a given base character—often by means of a special keyboard driver such as Tavultesoft Keyman,[7] configured to map a given keystroke to any of several diacritic variants depending on the most recently typed base character.

An example of such an encoding is that used by the SIL IPA93 package[8]—a set of phonetic fonts in several typefaces sharing a common encoding. This is a "presentation-form" or "glyph" encoding, where the encoded elements are not the logical characters of the script (which would typically be the elements encoded in Unicode) but rather represent the various glyphs needed for a reasonable visual result. This makes the encoded text much less convenient to use for any kind of analysis, searching, etc., but in many legacy systems this was the only way to achieve acceptable rendering.

A typical diacritic in these fonts comes in four versions: two different heights, and at two different horizontal positions. Figure 1 shows the set of acute accents in one of the fonts.

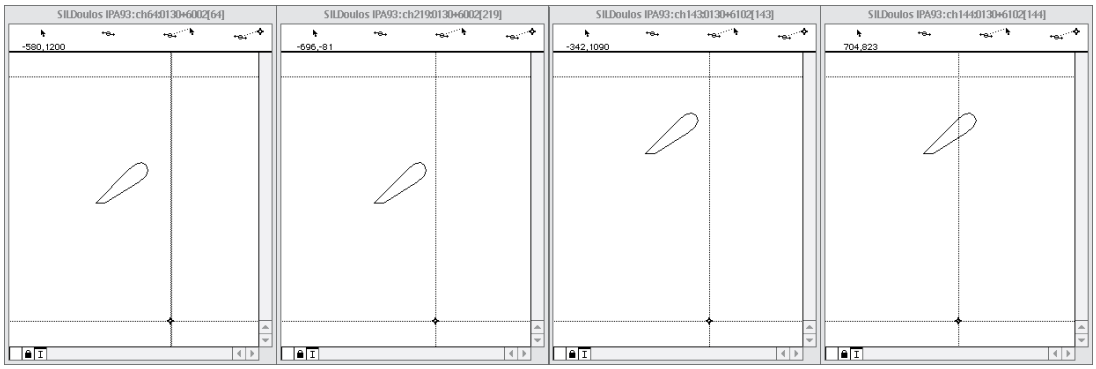

Figure 1: Four versions of the acute accent from SILDoulos IPA93

[7] See http://www.tavultesoft.com/keyman/.

[8] See http://www.sil.org/computing/fonts/encore-ipa.html.

These are, of course, really variant *glyphs* for the same *character*; in an ideal world, there would be a single codepoint for the acute accent, and the rendering software would choose the appropriate glyph variant (or adjust the glyph position). But the legacy systems for which SIL IPA93 was created have no such "smart rendering" facilities, and therefore these four varieties of acute accent each have their own individual codepoints in the 8-bit encoding.

Mapping these to Unicode does not present any major difficulty: all four SIL IPA93 acute accents can be mapped to U+0301 COMBINING ACUTE ACCENT. We could express this as four rules (unidirectional mappings at this point):

```
0x40      >      U+0301       ; acute over wide low characters
0xDB      >      U+0301       ; acute over narrow low characters
0x8F      >      U+0301       ; acute over wide tall characters
0x90      >      U+0301       ; acute over narrow tall characters
```

But what about the reverse mapping, from Unicode back to 8-bit SIL IPA93—which of the four codepoints should be chosen? Which of these four rules should be made bidirectional?

The answer, obviously, is that the correct reverse mapping depends on the preceding base character. Just as the keyboard driver (or the user, in the absence of a "smart" keyboard) chooses the form that will fit best with a given base character, so should the mapping for U+0301.

In a simple "string replacement" mapping description, this could be done by explicitly mapping each <*base character*, *acute accent*> pair from Unicode to SIL IPA93, instead of mapping the acute accent in isolation. There are around 100 potential base characters to be considered, so this seems tedious but not unmanageable. However, we must remember that there are a dozen or so other diacritics that can appear above base characters, and a similar number that go below (generally having two forms each). The number of mapping rules needed rapidly becomes uncomfortably large.

But the situation is worse than it first appears. A base character might have a diacritic below *and* one above. Or even two diacritics above—e.g., both a macron and an acute accent—in which case the first would be a low form and the second a high form (assuming a lowercase base character with no ascender). Listing every possible sequence that a linguist might use, so as to map the second or even third diacritic after a base character to the proper byte value, quickly becomes unwieldy. In effect, because SIL IPA93 uses a presentation-form encoding, the mapping from Unicode back to bytes has to emulate the "smart rendering" that would be required for proper appearance, and implementing this as a list of simple string replacements is not workable in cases with many diacritics that may be used in open-ended ways, rather than only in strictly limited contexts (like French accents, for example).

A better approach is to specify just four mappings for U+0301, one to each of the four codepoints in SIL IPA93, with contextual constraints that enable a mapping process to choose the appropriate rule. Extending our mapping rule notation, we could say:

```
0x40      <      U+0301 / [lowWide] _
0xDB      <      U+0301 / [lowNarr] _
0x8F      <      U+0301 / [highWide] _
0x90      <      U+0301 / [highNarr] _
```

Here, the slash indicates a contextual constraint for the Unicode-to-byte mapping; square brackets indicate character classes (defined elsewhere) containing sets of Unicode characters; and the underscore represents the position of the actual character being mapped (U+0301 in this case) within the context (which could include following as well as preceding characters).

Given the possibility of a diacritic below intervening between the base character and the acute, the contexts actually need to be more elaborate than this. Defining a character class for diacritics that go below the base character, and using a question mark to indicate an optional item,[9] we can extend the rules to be:

```
0x40      <      U+0301 / [lowWide] [dBelow]? _
0xDB      <      U+0301 / [lowNarr] [dBelow]? _
0x8F      <      U+0301 / [highWide] [dBelow]? _
0x90      <      U+0301 / [highNarr] [dBelow]? _
```

To allow the high forms of acute to be chosen after low base characters if there is another diacritic above the base, before the acute, we can further extend the second two rules (introducing a little more rule syntax again):

```
0x8F      <      U+0301 / ([highWide] [dBelow]? | [lowWide] [dAbove]) _
0x90      <      U+0301 / ([highNarr] [dBelow]? | [lowNarr] [dAbove]) _
```

Here, the parentheses represent grouping, and the vertical pipe is an alternation operator. While it is still not a trivial task to think through the possible cases and write the necessary rules, it is far more manageable to express the mappings in this way than to list every possible sequence as a separate string.[10]

Finally, having worked out the proper context for each of the reverse mappings, we are left with one further problem: unless absolutely every character, including control characters, punctuation, etc., is included in one of the four base character classes, we might encounter instances of U+0301 that don't match any of the contexts. Rather than treat these as unmapped, and generate a substitution character, we can choose one of the four legacy forms as a default and simply remove the constraint from that mapping. All occurrences of U+0301 that do not match any of the specific contexts given will then map to this form.

This illustrates a further principle that is helpful in describing mappings: where more than one mapping rule could apply to a given character in the data to be mapped, we assume that the most specific rule (that matching the longest string of codepoints, or the most extensive context) is to be used.[11] This allows us to choose one option as the "default" mapping for a given codepoint, leaving it unconstrained, and add specific constraints to the other possibilities.

Combining the simple mappings from SIL IPA93 to Unicode with the more complex reverse mappings, we have the following rules to handle the mapping of the four acute accents found in SIL IPA93. Other diacritics with multiple forms encoded in the fonts will require similar rules.

```
0x40 <> U+0301
0xDB <> U+0301 / [lowNarr] [dBelow]? _
0x8F <> U+0301 / ([highWide] [dBelow]? | [lowWide] [dBelow]? [dAbove]) _
0x90 <> U+0301 / ([highNarr] [dBelow]? | [lowNarr] [dBelow]? [dAbove]) _
```

---

[9] Although I have not formally defined the language being used here for mapping rules, I hope the similarity to existing regular-expression notations will make it readily understandable to the reader.

[10] Alert readers will note that these rules are still incomplete: there could be a diacritic below the base character *as well as* a preceding diacritic above. Additional `[dBelow]?` items should be inserted to allow for this.

[11] For simplicity, this statement is left somewhat imprecise at this point. The actual scheme used to prioritize rules in the TECkit system, described later, is that (a) the "length" of a match sequence or context is the maximum number of encoded characters it could match, if all repeatable and optional elements occur; (b) context length is the sum of the (potential) lengths of the preceding and following contexts; (c) length of the match string takes priority over the length of the context; and (d) where the potential match lengths are equal, the rule that is listed first takes precedence.

These are bidirectional rules, but the contextual constraints apply only to the reverse mappings; they refer to the context of the input Unicode character when mapping from Unicode to bytes. Constraints applying to the forward mappings, as described next, would be written immediately after the byte code values, before the mapping operator <>.

Such presentation-form encodings often require contextual rules when mapping Unicode text back to the legacy encoding, while the forward mapping is simpler. There are, however, also cases where the mapping from bytes to Unicode requires contextual constraints. One such example is the SIL Hebrew Standard encoding (see the SIL Hebrew Font System[12] for details). In this encoding, there is a single codepoint for each consonant, including the five Hebrew consonants *kaf*, *mem*, *nun*, *pe*, and *tsadi* that are written with alternate glyphs in word-final position; it is left to the rendering software to choose appropriate glyphs. However, Unicode assigns separate codepoints to the final forms of Hebrew consonants, rather than assuming "smart" rendering here.

Therefore, the mapping from SIL Hebrew Standard encoding to Unicode must convert these five consonants to either their non-final or final Unicode equivalents, depending whether additional Hebrew letters follow them. In this case, the forward mappings require contextual constraints, while the reverse mappings can simply map both non-final and final consonants to the same SIL Hebrew Standard codepoints:

```
0x63                    <>     U+05E5 ; final tsadi
0x63 / _ [dia]* [ltr]   <>     U+05E6 ; tsadi
0x6B                    <>     U+05DA ; final kaf
0x6B / _ [dia]* [ltr]   <>     U+05DB ; kaf
0x6D                    <>     U+05DD ; final mem
0x6D / _ [dia]* [ltr]   <>     U+05DE ; mem
0x6E                    <>     U+05DF ; final nun
0x6E / _ [dia]* [ltr]   <>     U+05E0 ; nun
0x70                    <>     U+05E3 ; final pe
0x70 / _ [dia]* [ltr]   <>     U+05E4 ; pe
```

Here, `[ltr]` is a class of all the Hebrew letters in SIL Hebrew Standard encoding. `[dia]` is a class including all the diacritics that could occur on the consonant; as these do not affect the choice of non-final versus final forms, they are marked with an asterisk, indicating an item that may occur zero or more times at this position in the context. The final forms are made the default mappings here, as it seems easier to specify "followed by optional diacritics and a Hebrew letter" for the non-finals than "followed by optional diacritics and then something other than a letter, or at the end of the text".

While the IPA example above could in principle be implemented using a simple string matching model for the mapping process—it would just require a very large number of rules—it is not clear that this is even theoretically possible for this Hebrew example. We can reasonably impose a limit on the number of diacritics that could occur on a single base character, but if we replace the rules that map a single character with a following context with rules that map the complete string (letter, diacritics, next letter) in a single operation, how do we know whether to use a final or non-final form for the *second* letter, in the cases where it is one of these five? We'd have to look ahead for another… and potentially another and another, *ad infinitum*. This highlights a crucial difference between rules that match and map an entire sequence, and those that depend on the context but are only actually mapping a single codepoint (or a more limited sequence).

[12] See http://www.sil.org/computing/fonts/silhebrew/.

This problem could almost be overcome by inverting the strategy, mapping the legacy Hebrew letters to non-final forms by default, and to final forms if followed by (optional diacritics and) a non-letter. As the mappings for the non-letter codepoints in SIL Hebrew Standard are simple one-to-one mappings, this does not suffer from the same defect. However, it would fail at the end of the input text, where a letter with no following text, which should be a final form, would not match any rule that looks for a following non-letter; it would therefore map to a non-final form.

We can envisage ways to work around this, too, such as treating the end-of-data as a special character code, and making it accessible to mapping rules, but this also takes us beyond the simple string-matching model expressed by UTR22 mapping descriptions, while lacking the combination of conciseness and expressiveness offered by contextually-constrained mapping rules.

3. Reordering

A second type of complexity arises when the ordering of encoded elements differs significantly between the legacy encoding and Unicode. A simple example occurs with another SIL product, the SIL Greek font system.[13] This supports an encoding known as SIL Basic Greek, which is a fully decomposed encoding for polytonic Greek text. As such, it is similar to Unicode Greek in decomposed form, but there is an important difference: SIL Basic Greek encodes the Greek breathing marks in their logical position *before* the vowel or diphthong with which they are associated. Unicode, on the other hand, requires the breathings (like all combining marks) to *follow* the vowel. Figure 2 illustrates the difference in character order in a typical Greek word.

SIL Basic Greek				*Unicode*		
0x68	῾	*rough breathing*		U+03BF	ο	*omicron*
0x6F	ο	*omicron*		U+03C5	υ	*upsilon*
0x75	υ	*upsilon*		U+0314	῾	*reversed comma above*
0x5E	῀	*circumflex*		U+0342	῀	*Greek perispomeni*
0x74	τ	*tau*		U+03C4	τ	*tau*
0x6F	ο	*omicron*		U+03BF	ο	*omicron*
0x73	σ	*sigma*		U+03C2	ς	*final sigma*

Figure 2: Comparing the character ordering for οὗτος in legacy and Unicode encodings

(Incidentally, the *sigma* here requires similar contextual treatment to the Hebrew consonants discussed above, as Unicode encodes final and non-final *sigma* separately.)

In the case of Greek, the number of relevant *breathing+vowel* combinations is bounded (around 160), and it is reasonably practical to map them all by simply listing each possible sequence. However, the mapping between the two encodings can be much more succinctly expressed using rules that show how the ordering of encoded characters is related. For example, if we define classes of the byte codes for the breathings and the vowels, and similar classes of Unicode values to use on the right-hand side of the mapping, we can match:

```
[breath_b]=B [vwl_b]=V1 [vwl_b]=V2 <> [vwl_u]=V1 [vwl_u]=V2 [breath_u]=B
```

The items here are “tagged” with identifiers to allow the mapping process to associate them correctly despite the difference in ordering; the meaning of such a rule would be that wherever

[13] See http://www.sil.org/computing/fonts/silgreek/.

this sequence is matched, each codepoint is mapped to the corresponding codepoint from the class with the same "tag" on the other side of the mapping.

In fact, this rule is insufficient for the Greek mapping, as not all vowel pairs are valid diphthongs; moreover, a dieresis following the second vowel would indicate that it is not a diphthong and therefore the breathing should follow the first of the vowels in Unicode, not the second. The actual rules used in the SIL Basic Greek mapping description, therefore, are slightly more extensive, using several separate classes of vowels to ensure that only pairs that represent valid diphthongs will be matched as such.

The Greek reordering rules shown here are designed to be part of a multi-stage process, which will be discussed further shortly. In the Greek mapping, the legacy byte codes are individually mapped to corresponding Unicode characters (these rules are not shown here); then, in a separate pass, the Unicode characters are reordered as necessary. Thus, the reordering rules are written entirely in terms of Unicode characters, and the right-hand side uses a new notation `@tag` to represent "the tagged item from the matched string" instead of re-stating the classes. Here, this serves as a convenience: it is cheaper to copy the input code than to look up a corresponding class member, when that is merely going to result in the same character; later, we will see the feature used more extensively.

```
Class [BR]       =      ( U+0313 U+0314 )
Class [aeo]      =      ( U+0391 U+0395 U+039f U+03b1 U+03b5 U+03bf )
Class [iu]       =      ( U+0399 U+03a5 U+03b9 U+03c5 )
Class [j]        =      ( U+0397 U+03b7 )
Class [u]        =      ( U+03a5 U+03c5 )
Class [i]        =      ( U+0399 U+03b9 )
Class [vowelrho] =      ( U+0391 U+0395 U+0399 U+039f U+03a5 U+0397 \
                          U+03a9 U+03a1 U+03b1 U+03b5 U+03b9 U+03bf \
                          U+03c5 U+03b7 U+03c9 U+03c1 )

[BR]=b [aeo]=v1 [iu]=v2 / _ U+0308   <>     @v1 @b @v2 / _ U+0308
[BR]=b [aeo]=v1 [iu]=v2              <>     @v1 @v2 @b
[BR]=b [j]=v1 [u]=v2 / _ U+0308      <>     @v1 @b @v2 / _ U+0308
[BR]=b [j]=v1 [u]=v2                 <>     @v1 @v2 @b
[BR]=b [u]=v1 [i]=v2 / _ U+0308      <>     @v1 @b @v2 / _ U+0308
[BR]=b [u]=v1 [i]=v2                 <>     @v1 @v2 @b
[BR]=b [vowelrho]=v                  <>     @v @b
```

For Greek, expressing the reordering in this way makes for a more concise mapping description than explicitly listing every relevant sequence of up to four codepoints, and more clearly expresses the relationship between the encodings, but the mapping could have been implemented without these features. In more complex cases, however, the ability to express a mapping as a sequence of several passes, and to use class-based rearrangement rules, is critical to making a mapping practicable at all.

As an example of this level of complexity, consider a legacy font for Devanagari, designed for use without any kind of "smart rendering" and therefore implementing a presentation-form encoding. The byte encoding has separate codes for half forms of Devanagari letters, and for many complete conjuncts; it also has multiple versions of many non-spacing marks (with different positioning, appropriate for different base characters). In addition, the byte codes occur in left-to-right visual order rather than in the phonetic order used in Unicode. Taking a sample word त्रिमुर्ति, which could be transliterated as *trimurti*, we can compare how it is encoded in the legacy system and in Unicode. In the legacy encoding, this word is encoded as the sequence of seven bytes, while in Unicode it consists of ten characters, as shown in figure 3.

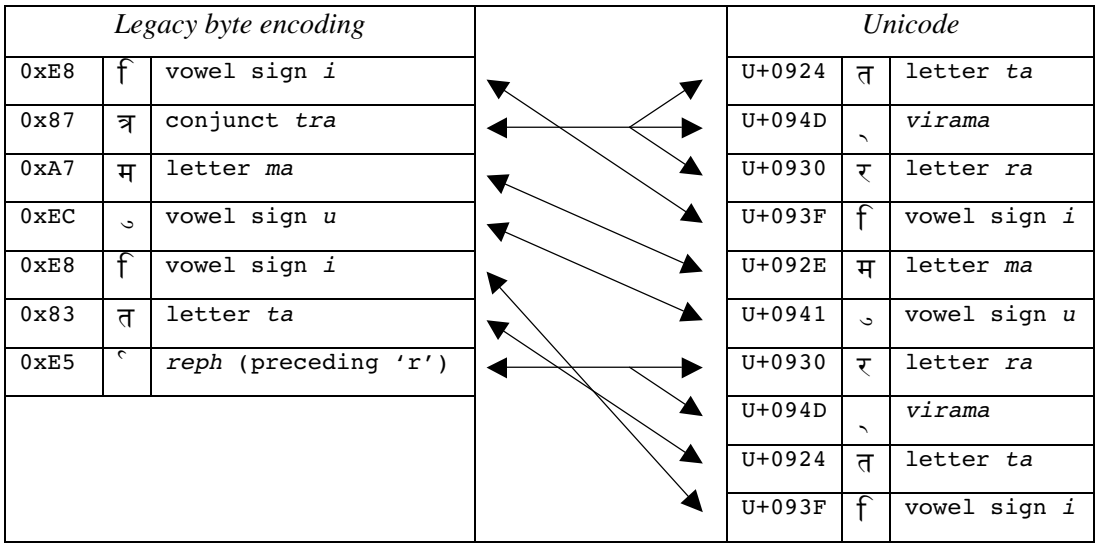

Figure 3: Reordering and decomposition in Devanagari script

Not only are the *tra* conjunct and the *reph* (half-form of *ra*) encoded as sequences of letters with *virama*, but the character ordering is also quite different. Only the middle syllable *mu* of this word has a simple mapping.

In cases like this (commonly found in legacy glyph-based encodings for South and South-East Asian scripts), it is utterly impracticable to deal with every possibility by directly mapping literal sequences of bytes to sequences of Unicode characters; hundreds of thousands of combinations of consonant or consonant cluster, vowel, and other diacritics would have to be handled.

The solution is to separate issues of ordering from those of mapping between codespaces. We can write expressions that match entire syllables, with their various optional elements, and rearrange the parts of the syllable (consonant, vowel sign, nasalization mark, etc.) into the required order; then we can map individual legacy codepoints to their Unicode equivalents.

Before attempting the reordering, it is helpful to preprocess the legacy text so as to replace conjuncts and half-form consonants with "canonical" representations using only full consonant codes and *virama*. This step is not strictly needed—this could be handled at the stage of mapping between bytes and Unicode—but it serves to simplify the expression of the reordering rule.

```
; half forms of consonants
class[H] = (77 84 86 90 104 108 112 113 115 130 \
            132 137 149 153 156 159 163 166 168 \
            171 174 181 186 192 197 201)

; full forms of the above half consonants
class[F] = (76 83 85 89 103 107 110 111 114 129 \
            131 136 144 152 155 158 162 165 167 \
            170 173 180 185 191 196 200)

; all consonants = [F] + those that don't have half-forms
; (excluding /ra/, as we don't want half-forms before it)
class[C] = ([F] 0x60 0x6a 0x74 0x77 0x79 0x80 0x8a 0xcc)

[H]       <>     [F] 0xfe / _ [C]

0x4e      <>     0x4c 0xfe 0x4c      ; k+ka
0x4f      <>     0x4c 0xfe 0x83      ; k+ta
0x50      <>     0x4c 0xfe 0xaf      ; k+ra
0x51      <>     0x4c 0xfe 0xbf      ; k+sha
```

```
0x52      <>     0x4c 0xfe 0xbf 0xfe ; half-k+sha
; ...etc...
```

(As may be deduced by those familiar with the script, 0xFE here is the legacy code for the *virama* character that indicates absence of a vowel, or a consonant cluster.) Assuming this step is done first, we can describe the order used in the legacy encoding as:

```
short-i-sign?
(consonant nukta? virama)*
consonant nukta?
vowel-sign?
reph?
nasal?
```

The order required in Unicode is:

```
reph?
(consonant nukta? virama)*
consonant nukta?
(vowel | short-i-sign)?
nasal?
```

Here, *reph* will ultimately be mapped to *ra* and *virama*, but the reordering is simpler if it is kept as a single entity at this point. The actual rule that expresses this reordering, with the character classes it refers to in the legacy encoding, is:

```
;basic consonant codes
class[C] = (0x4c 0x53 0x55 0x59 0x60 0x67 0x6a 0x6b 0x6e 0x72 0x74 0x77 \
            0x79 0x80 0x81 0x83 0x88 0x8a 0x90 0x98 0x9b 0x9e 0xa2 0xa5 \
            0xa7 0xaa 0xaf 0xb4 0xb9 0xbf 0xc4 0xc8 0xcc)
;nasalization (anusvara, candrabindu)
class[N] = (0xde 0xe0)
;vowel signs that follow the consonant (all except short i-kar)
class[V] = (0xdd 0xe7 0xea 0xec 0xf1 0xf6 0xf8 0xfa 0xfc)

0xe8?=ikar (([C] 0xdb? 0xfe)* [C] 0xdb?)=cons [V]?=vwl 0xe5?=reph [N]?=nas \
          <> @reph @cons @vwl @ikar @nas
```

The notation `@tag` on the right-hand side of the rule here indicates whatever data was matched by the item on the left-hand side that is labelled with `=tag`. Note that all the tagged items on the left-hand side except *cons* are optional (and *cons* itself might consist of just a single consonant). If a tagged item matches no input data, the corresponding `@tag` item will produce no output. To perform the reordering stage of the reverse mapping, from Unicode to legacy encoding, all that is needed is to exchange the `@tag` "copy" items on the right and the matching `=tag` "tagged" items on the left; thus, this can be considered a bidirectional mapping rule.

Having handled the ordering complexity in this rule, the actual mapping between byte codes and Unicode characters becomes a simple one-to-one mapping (except for *reph*, which was left as a single byte but must be mapped to *ra* plus *virama*). The combination of contextual constraints (to deal with complete mapping between this complex glyph-based legacy encoding and Unicode requires only a couple of hundred mapping rules.

4. Round-trip considerations

With presentation-form encodings (and the associated complex mappings), issues of round-trip safety become much more difficult than with most standard encodings, where mappings are one-to-one. Returning to SIL IPA93 as an example, we had four forms of acute accent, all mapping to

the same Unicode character, and with contextual constraints selecting the appropriate form when mapping back to the legacy encoding:

```
0x40 <> U+0301 / [lowWide] [dBelow]? _
0xDB <> U+0301 / [lowNarr] [dBelow]? _
0x8F <> U+0301 / ([highWide] [dBelow]? | [lowWide] [dBelow]? [dAbove]) _
0x90 <> U+0301 / ([highNarr] [dBelow]? | [lowNarr] [dBelow]? [dAbove]) _
```

These rules will give the appropriate results for both forward and reverse mapping when the acute accent occurs normally in text. However, there is nothing about SIL IPA93 that makes it illegal or impossible to have, say, a "high" acute over a low base character, or even following a space (to display the glyph in isolation). The keyboard driver normally used to type in this encoding will not generate this, as it will only insert the high form following tall base characters (just like the contextual mapping rules), but it is certainly possible for users to create such text (whether inadvertently during editing, or deliberately for a special purpose).

Where such presentation forms are used outside their normal context, these rules will still map them to the same Unicode characters, but the reverse mapping will be unable to recreate the "anomalous" form that was originally present. Thus, the mapping is not fully round-trip safe.

Whether this is seen a "bug" or a "feature" of the mapping depends on the specific situation. In many cases, the result of such "unsafeness" is that data is actually regularized in a helpful way during the mapping process; this could be considered similar to canonical normalization of Unicode. However, there is also the real possibility that a user was making a deliberate choice of glyph, and would want this preserved. To do this, additional information (the explicit glyph choice desired) has to be encoded somehow into the Unicode text. This could be done, for example, by mapping the four acute accents to separate PUA characters instead of U+0301, or by defining PUA characters for *high*, *low*, *wide*, and *narrow* that could be inserted before or after the standard Unicode accent.

This complication arises because of the mismatch between the actual information units represented in the legacy encoding and those defined in Unicode. Where the legacy system encodes glyphs, and Unicode encodes characters, it is often easy to find cases where a particular glyph sequence that can exist in the legacy system does not correspond to a distinct Unicode character sequence—in other words, the legacy system has the potential to make distinctions that Unicode does not. PUA definitions allow a workaround, but for many purposes users may be better served by an "unsafe" mapping that results in cleaner data.

Another example occurs in Devanagari. The legacy system discussed earlier includes codes for a number of conjunct forms (consonant clusters), as well as for the "half consonants" used to write clusters for which no conjunct is available, and the *virama* used when no half form exists. In the process of mapping to Unicode, all these become sequences of consonants and *viramas*. But then the reverse mapping will recreate half consonants and conjuncts wherever these are available—even if the original text actually *didn't* use them, but had explicit *viramas* instead. Again, the mapping is not strictly round-trip safe.

In the case of Devanagari, a standard solution is possible: when mapping to Unicode, ZERO WIDTH JOINER and NON-JOINER characters can be inserted to indicate where half consonants are desired rather than conjuncts, or explicit *viramas* rather than either of these.[14] If this is done (and the appropriate optional items added to the reordering rule), the mapping can be made "safer". However, whether this is desirable depends on the particular application to which the data is being put; in many cases, the use or otherwise of particular conjuncts may be unimportant,

[14] See *The Unicode Standard, Version 3.0*, section 9.1 for the details of these conventions.

and the additional codes needed to ensure precise round-trip safety only serve to clutter the Unicode text and complicate subsequent text-processing operations. These decisions can only be made in the light of a specific application of the mapping.

5. A mapping engine implementation

The purpose of this paper is to illustrate the types of complexity that need to be handled in order to map between glyph-based legacy encodings and Unicode, rather than to document a particular product. However, it should be noted that the approach proposed is not merely hypothetical, but has been implemented in a working encoding conversion engine. This is known as TECkit (as it can be considered a Text Encoding Conversion toolkit).

The core element of TECkit is the conversion engine, implemented as a shared library that operates on buffers of data passed to and from a client that requires conversion services. Conversions are defined by mapping files in a binary format designed to make the implementation of the engine reasonably simple, while allowing common cases (especially the case where many codes have one-to-one mappings, and only a small number require complex rules) to be particularly efficient.

Mappings for TECkit can be described in text files using a language developed specifically for this purpose; this is, in fact, the language informally introduced in this paper to discuss mapping rules. A compiler creates the binary mapping tables needed by the engine from mapping descriptions written in the TECkit language. The binary format is documented, so that other types of mapping description (e.g., UTR22 descriptions) could be compiled for use by the TECkit engine; in fact, a colleague is intending to develop just such a compiler.

The set of regular expression features supported by the TECkit language illustrates the functions that have proved useful in implementing mappings for a variety of complex legacy encodings:

`[cls]`	match any character from the class *cls*
`.`	match any single character
`#`	match beginning or end of input text
`^item`	"*not* item": match anything except the given item (applies to single items only; negated groups are not supported)
`(...)`	grouping (for optionality or repeat counts)
`|`	alternation (within group): match either preceding or following sequence
`{a,b}`	match preceding item minimum *a* times, maximum *b* ($0 \le a \le b \le 15$) [15]
`?`	match preceding item 0 or 1 times
`*`	match preceding item 0 to 15 times
`+`	match preceding item 1 to 15 times
`=tag`	tag preceding item for match/replacement association
`@tag`	(only on RHS, and only in single-codespace passes) duplicate the tagged item (including groups) from LHS; typically used to implement reordering

For the convenience of mapping authors and application developers, TECkit supports multiple Unicode encoding forms and schemes, and provides built-in support for normalization. Unicode characters are identified in a mapping description by their scalar values or their Unicode names; clients may specify any of the standard UTFs for input or output to the conversion process. Normalization support allows a mapping author to declare that Unicode text to be mapped should be in either NFC or NFD, and TECkit will normalize incoming data so that the mapping rules see

[15] As this indicates, unbounded repeats are not supported; this can simplify implementation, as the maximum number of characters a rule might need to examine can be precomputed.

the expected form. Similarly, a client application may request either NFC or NFD when mapping to Unicode, and will receive this independently of how the particular mapping in use was written.

In addition to simple tools for text file conversions, included as part of the TECkit package itself, the conversion engine is being integrated into Unicode-based applications that need to be able to import and export data in legacy encodings for interchange with older versions. For example, the newest version of the United Bible Societies' *Paratext* translation environment uses Unicode, while older versions used legacy byte encodings; when importing data from existing projects, it can use the TECkit engine to perform the necessary conversion, and can thus support complex custom encodings used in minority languages, not just major standards.

At the time of writing, the TECkit system has been built on Windows and Mac OS platforms; however, it is intended to be readily portable, and we would be happy to see it widely implemented and used. Further information about TECkit can be found on the Internet at http://www.sil.org/nrsi/teckit/.

6. Extending UTR22

We have seen that describing the mapping between legacy byte encodings and Unicode can require fairly complex contextual constraints on mapping rules, and can involve extensive reordering that is best expressed as a distinct step in the mapping process. What does this imply for the mapping description language presented in *UTR22: Character Mapping Markup Language*? (This section will assume familiarity with that language.)

In this paper, I do not aim to present a full proposal for an "enhanced CharMapML" language, but will outline ways in which I believe the existing language could be extended to provide the required descriptive power. I hope this can lead to a discussion among those interested in character mapping issues, leading eventually to an improved standard that will meet these needs.

First, we need to be able to specify preceding and following context for assignments. These contexts may be simple character codes, or may consist of regular expressions. The existing TECkit language (see section 5) illustrates the features that have been found useful in describing mappings for a number of complex legacy encodings, and so a comparable set of features should be provided in the CharMapML language.

As contexts may be complex sequences of several types of item, it seems best to express them as subtrees of XML elements, rather than simply as additional attributes of the `<a>` element (which would then require their own internal structure of some kind). Two approaches could be taken to this: either to permit `<a>` to contain context sequences as child elements (perhaps actually defining a new element `<ac>` for this, as the current DTD specifies that `<a>` is empty), or to define contexts separately, assign them identifying tags, and then have context-name attributes on `<a>`.

For example, a rule that we have (in TECkit notation) been writing as:

```
0x8F <> U+0301 / ([highWide] [dBelow]? | [lowWide] [dBelow]? [dAbove]) _
```

might become, in an XML language, something like:

```
<ac b="8F" u="0301">
 <uctxt>
  <pre>
   <alt>
    <seq>
     <classref cls="highWide"/>
     <classref cls="dBelow" min="0" max="1"/>
    </seq>
    <seq>
```

```
      <classref cls="lowWide"/>
      <classref cls="dBelow" min="0" max="1"/>
      <classref cls="dAbove"/>
     </seq>
    </alt>
   </pre>
   <!--if following context needed, that would be a <post> element-->
  </uctxt>
  <!--if context needed on the byte side, that would be a <bctxt> element-->
 </ac>
```

Alternatively, especially if the same context is appropriate for a number of rules, the approach of defining contexts separately and referring to them could look like:

```
<assignments>
 <a b="8F" u="0301" uctxt="diaAboveCtxt"/>
</assignments>
<unicodeContexts>
 <class id="highWide" members="..."/>
 <!-- other classes defined here-->
 <ctxt id="diaAboveCtxt">
  <pre>
   <alt>
    <seq>
     <classref cls="highWide"/>
     <classref cls="dBelow" min="0" max="1"/>
    </seq>
    <seq>
     <classref cls="lowWide"/>
     <classref cls="dBelow" min="0" max="1"/>
     <classref cls="dAbove"/>
    </seq>
   </alt>
  </pre>
  <!-- if following context needed, that would be a <post> element -->
 </ctxt>
</unicodeContexts>
```

To allow mapping authors the flexibility to either define contexts once and then refer to them by name (best if the same context is required many times), or to define them directly as needed (most convenient for one-off use), the language could permit both approaches.

Second, there is the issue of multi-stage mappings. These could be handled by permitting multiple `<assignments>` elements in the description. However, we then have to consider the fact that not all `<assignments>` are between bytes and Unicode; there may be byte-to-byte rules in an `<assignments>` element before the mapping to Unicode takes place, and Unicode-to-Unicode rules afterwards. The `<assignments>` element itself could have a new attribute specifying the type of "pass" being described, with the default (for backward compatibility) being byte-to-Unicode. Thus, a multi-stage mapping such as the Devanagari example might take the form:

```
<assignments codespace="bytes">
 <!--rules that regularize positional variants of diacritics-->
</assignments>
<assignments codespace="bytes">
 <!--replace conjuncts/half consonants with consonant-virama sequences-->
</assignments>
<assignments codespace="bytes">
 <!--reorder syllables from legacy to Unicode order-->
</assignments>
<assignments sub="...">
 <!--map the legacy byte codes to their Unicode equivalents-->
</assignments>
```

Incidentally, it seems logical that in a single-codespace pass, either bytes or Unicode, there is no concept of a substitution character for unmapped codes; rather, any codes not explicitly mapped should simply be copied to the output. This makes sense when we are effectively performing a transduction within a single codespace, rather than making the "leap" from bytes to Unicode values.

Furthermore, once we have `<assignments>` that are not necessarily dealing with bytes on the one side and Unicode on the other, the attributes `b` and `u` of `<a>` are no longer appropriately named. Alternate names such as `lhs` and `rhs` might be less confusing. (This would also apply to the proposed attributes or subelements for contexts; rather than `uctxt`, we might want to use `rctxt` or something similar.)

Finally, we have not yet seen a way to express reordering rules, such as are needed for the Devanagari example. I would suggest that this can be handled using an extended form of the `<a>` element, where instead of `b` and `u` (or `lhs` and `rhs`) attributes, the two sides of the mapping are expressed as subelement trees, similar to those we have already seen for contexts. So the Devanagari rearrangement rule:

```
0xe8?=ikar (([C] 0xdb? 0xfe)* [C] 0xdb?)=cons [V]?=vwl 0xe5?=reph [N]?=nas \
          <> @reph @cons @vwl @ikar @nas
```

might become:

```
<!--class definitions here, or are they local to <assignments>?-->
<assignments codespace="bytes">
<ac>
 <lhs>
  <c n="E8" min="0" id="ikar"/> <!--min and max default to 1-->
  <seq id="cons">
   <seq min="0" max="15">
    <classref cls="C"/>
    <c n="DB" min="0"/> <!--this is the Devanagari nukta diacritic-->
    <c n="FE"/> <!--this is the byte code for virama-->
   </seq>
   <classref cls="C"/>
   <c n="DB" min="0"/>
  </seq>
  <classref cls="V" min="0" id="vowel"/>
  <c n="E5" min="0" id="reph"/>
  <classref cls="N" min="0" id="nasal"/>
 </lhs>
 <rhs>
  <copy ref="reph"/>
  <copy ref="cons"/>
  <copy ref="vowel"/>
  <copy ref="ikar"/>
  <copy ref="nasal"/>
 </rhs>
</ac>
</assignments>
```

These examples are intended to suggest how the processing model implemented in TECkit, which has been found appropriate for a variety of complex legacy encodings, might be supported through extensions to the UTR22 description. These are only preliminary suggestions, however, and careful consideration is needed into the best ways to express the types of rules that have been shown using the notational devices of XML. It is hoped that this paper will encourage discussion that will lead to a new, improved version of the standard that will serve the full range of needs among users faced with complex legacy encoding issues.

7. Higher level issues

There remain some categories of legacy encodings that have not been addressed. These include systems where characters are distributed between multiple fonts (e.g., the use of two fonts for Ethiopic, because of the limited number of glyphs accessible from a single-byte encoding without smart rendering); the use of character formatting to simulate diacritics (e.g., underlining used to represent a macron below); and various implementations of right-to-left scripts, using either purely visual ordering or bidi models that do not match Unicode's.

These types of legacy systems introduce issues that are beyond the scope of a pure character-mapping system. The proper interpretation of the legacy characters becomes dependent on application-specific character formatting, or on paragraph direction and line wrapping; it is difficult to see how a general mapping description language or character mapping engine can adequately deal with these issues. Rather, it seems likely that legacy systems of this sort will require the development of custom conversion processes.

8. Conclusion

Many legacy encodings, particularly for non-Roman and minority languages, cannot be mapped to Unicode using the relatively simple models implemented in most current conversion software and embodied in the UTR22 description language. This is particularly true where presentation-form (glyph-based) encodings have been used due to a lack of "smart" rendering capability.

To enable users of such legacy encodings to bring their data into new, Unicode-based systems, encoding conversion tools need to implement a more flexible and powerful processing model, with provision for contextually dependent mappings and extensive reordering of codepoints in a multi-pass conversion model. Without this, interoperability of Unicode-based and legacy systems will be seriously hampered for users working with such languages and encodings.

In the interests of widespread adoption of a common standard for encoding conversion and mapping description, the language described in UTR22 should be extended to support these additional features; further discussion on this topic is invited.