Complex Mapping Description
A Description of an Extension to UTR22

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

Introduction

"Quick! I need a program to convert my data into Unicode."

For many, this is the be all and end all of data conversion, to have their data in a different encoding. The result is that there is incredible pressure to write ad-hoc programs, in whatever language is to hand, to do these quick conversions. The problem is that we end up writing the same programs again and again in each new environment for each new problem. The pragmatic solution may provide a quick response, but in the long run it is difficult to hand over to someone else; it is difficult to maintain and error prone and most of all, it is tedious!

But, even before we can hope to provide a pragmatic solution, we need a description of the mapping we are intending to implement. Therefore, if we could find a way of taking some kind of formal description of a mapping and to execute it, there would be no need to write the mapping software. This is the premise behind the Unicode Technical Report #22 (UTR22) for describing legacy mappings. It aims to provide a description language which may either be executed directly or automatically converted to another form for faster execution, while still being highly descriptive.

This paper describes an extended form of the UTR22 language to support complex mappings required to describe legacy encodings which do not map simply to Unicode. Such mappings are required for almost any encoding which is not a direct subset of Unicode. The particular class of encodings which are of concern here are those which have been developed according to the single codepoint per glyph model of rendering.

In reading this description it is worth noting the equal emphasis that has been placed on this language being a description language and that the description be executable. This means that, for example, the description is reversible – only one description is required to result in conversion too and from the legacy encoding. A secondary aim is that, in effect, such a mapping description may form a foundational part of an encoding description, by providing a formal relationship between the encoding being described and Unicode, which is one of the best described encodings in existence.

The various sections in this description build up the language in terms of language features that have been included to handle more and more complex problems. Thus we start with the simplest mappings and work our way up to the really hairy ones. The initial sections will use a single mapping description as their example. The encoding used for the SILIPA93 series of fonts (SILDoulos IPA93, etc.) bears most of the characteristics of a moderately difficult mapping and we will build up the description of this mapping through the initial sections of this paper. A code chart for that encoding is provided in Appendix A.

Simple Mapping

Simple mappings are those in which a single character code (which may be multiple bytes in a multi-byte encoding scheme) maps to a single Unicode character and vice versa, at least for a subset of Unicode. This is the situation for which UTR22 was developed and so we concentrate on this simple description initially.

The UTR22 language is an XML language, and so a description starts with a traditional XML header:

```
<?xml version="1.0" encoding="UTF-8"?>
```

The encoding is described as UTF-8 since it is highly probable that a mapping description will contain Unicode characters.

Root Element

The root of a mapping description is a `characterMapping` element which has a number of attributes. These attributes provide general information about the mapping description for identification purposes.

id [Required] The identifier name for the character mapping. This must be unique and takes the following form: `xxx-yyy-zzz` where `xxx` is a string identifying the organization, for example: `windows` or `SIL` or even `SIL_CAB`. `yyy` is the encoding in question. For example, Microsoft would insert a codepage number here, SIL might insert a language name or language group name here. `zzz` is the year the description was created. If more than one such mapping description is made within a year, subsequent versions should add a letter after the year. Notice that `xxx`, `yyy` and `zzz` may be any length.

version [Required] This contains the current version number of the description, which is reflected in the first modification element in the history.

description [Optional] This attribute contains a short description of the encoding being described. The description is typically less than a line long and is used to provide a more readable description of the id. This string should only contain characters in the range U+0020 to U+007E.

contact [Optional] This attribute contains the URL of a contact person to whom questions and modification requests should be sent.

registrationAuthority [Optional] contains the organization responsible for the encoding.

registrationName [Optional] contains the name by which the organization knows this encoding

copyright [Optional] contains a copyright statement, typically allowing people to use, but not to erase the copyright or claim ownership of the information.

bidiOrder [Optional] In cases where the bi-directional algorithm is significant, this attribute can be used to give an overall description of the legacy encoding. The default is that legacy data is considered to be stored in `logical` order. I.e. the relative bi-directional ordering of data is the same as Unicode and no reordering need be done. The other two values are: `RTL` and `LTR` indicating that data is stored in a strict, surface order and that the bi-directional algorithm may need to be run on the data. Notice that not all mapping conversion processors support bi-directional reordering.

normalization [Optional] This attribute describes which normal form the mapping is working with. The values are: `undetermined` (default), `neither`, `NFC`, `NFD`, `NFC_NFD`[1]. When converting to Unicode, such information is useful to know and can perhaps save a normalization pass, but when converting from Unicode to bytes, this information is essential since it tells the mapping processor which normal form the Unicode data is expected to be in before conversion. A value of `undetermined` is really an error condition, since the processor does not know into which form the data should be converted. But since this is the default value, it is not flagged as an error and is treated as `NFC_NFD` by the conversion processor. A value of `neither` implies that the data is in neither NFC or NFD (i.e. there are elements of both and the mapping is ambiguous). In this case, a mapping conversion processor may flag an error, since there is no way that it can get arbitrary Unicode data into the correct form for processing. If no error is indicated, then the value should be interpreted as `NFC_NFD`. `NFC` and `NFD` values indicate which normal form the data should be in before conversion. The `NFC_NFD` value indicates that normalization is not an issue and the data can be in either form for processing. Thus there is no need to execute a normalization pass over the data before conversion.

[1] NFC and NFD are Unicode normalization designations for two particular normal forms: NFC is normal form composed (in which canonically equivalent ligatures are used as much as possible) and NFD is normal form decomposed (in which characters are decomposed into their canonically equivalent components).

byte-font [Optional] Specifies a byte font to use when rendering bytes mapped by this table. For example a stylesheet may want to render characters in the bytes in a font. It may use the font specified here.

unicode-font [Optional] Specifies a font to use when displaying Unicode characters mapped by this table.

Some of the concepts raised in these attribute descriptions will be raised again in the appropriate sections covering those topics.

For the SILIPA93 encoding, the `characterMapping` element is:

```
<characterMapping
  id="SIL-IPA93-2001"
  version="10"
  description="SIL IPA93 Font encoding"
  contact="mailto:martin_hosken@sil.org"
  registrationAuthority="SIL International"
  registrationName="IPA93"
  normalization="NFD"
  byte-font="SILDoulos IPA93">
```

Contents

The main component of the mapping description is held in an `assignments` element and consists of a number of mapping rules. The `assignments` element has one attribute:

sub [Optional] This attribute contains a list of byte values which are inserted when an unknown Unicode value is mapped to bytes. The default value is `1A`.

When talking of lists of byte values and Unicode values, all such values are represented as hexadecimal numbers. A byte value is a 2 character hexadecimal number. The value is always 2 characters (even for values less than 16_{10}) and a list is made up of a space separated list of 2 character hexadecimal numbers. Likewise, a Unicode value is a hexadecimal number in the range 0000-10FFFF and is always between 4 and 6 characters in length. A list of Unicode values is a space separated list of Unicode values.

Within the `assignments` element are a number of associations between byte values and Unicode values. The basic association is represented by the `a` element, which has no children. The `a` element represents a bi-directional association, that is the byte value maps to the Unicode value and the Unicode value maps to the byte value. There are also uni-directional associations, which we will examine in a moment.

The `a` element takes a number of attributes:

b [Required] contains a list of byte values that are to be associated with the corresponding list of Unicode values.

u [Required] contains a list of Unicode values that are to be associated with the corresponding list of byte values.

c [Optional] contains the actual Unicode string represented by **u**. This attribute is provided as a means of commentary.

v [Optional] contains a version number (year plus letter) in which this association was added. This allows an application to request a particular version of the mapping based on date. If there are two identical associations (in a particular direction), that would therefore clash, then the one with the later version is taken[2].

As an example, consider the schwa character (code 171) and its mapping to Unicode (U+0259). This would be represented in a mapping description by:

```
<assignments sub="3F">
      :
   <a b="AB" u="0259"/>
```

[2] This is the behaviour specified in UTR22, the current (2001) software implementation ignore the v attribute and so would indicate an error in this situation.

```
        :
</assignments>
```

As can be seen, a typical mapping would consist of an `assignments` containing some 200 `a` elements of this form.

There are also cases where two different byte characters get mapped to the same Unicode character. In this case we need some way to indicate that a mapping is single directional. There are two elements that inidicate a single direction relationship:

fbu Indicates a relationship between the `b` and `u` attributes only when mapping from bytes to Unicode.

fub Indicates a relationship between the `b` and `u` attributes only when mapping from Unicode to bytes.

It is not required for a conversion engine to implement support for `fbu` and `fub` and if these mappings are ignored, then the relevant characters will be mapped to the unknown character. When mapping to Unicode, this is `U+FFFD`. When mapping to bytes, the `sub` attribute of the `assignments` element gives the value to be inserted. The `fub` element is particularly often used for describing possible mappings between Unicode characters and what are considered best guess alternatives in the byte encoding.

As an example, consider the case in SILIPA93 of the rising tone diacritic (U+030C) which is represented in SILIPA93 by the four different codes: 0x26, 0xE0, 0xF3 and 0xF4. These 4 codes all map to U+030C. But what happens when we map from Unicode back to bytes? The Unicode value can only map to one of these byte values. Which one is it to be? We shall see in a later section how this problem can be resolved properly, but for now, let us assume they are in free variation, and so we pick one to be used when mapping from Unicode to bytes. And we use the following description:

```
<assignments sub="3F">
   <fbu b="26" u="030C"/>
   <fbu b="E0" u="030C"/>
   <fbu b="F3" u="030C"/>
   <a b="F4" u="030C"/>
</assignments>
```

Notice that we have used the last mapping as the reverse mapping since 0xF4 is the code for the o-width high positioned diacritic, which is a good fall-back. The `fbu` takes the same attributes as an `a` element, except that it does not support the `c` attribute (since we are not really interested in what the Unicode value looks like for this type of mapping).

Equally, there is the reverse problem. Consider the Unicode letter `U+0133 LATIN SMALL LIGATURE IJ`, used in Dutch. It has only a compatibility decomposition to `ij` and so would not be resolved by normalization to the `NFD` form specified in the `normalization` attribute in the `characterMapping` element in our description. Therefore, we might want to support it directly in our description, that when we map from Unicode to bytes, U+0133 should map to 0x69 0x6A, but that when mapping from bytes to Unicode, the reverse should not happen. We could do this using an `fub` element:

```
<fub b="69 6A" u="0133"/>
```

The `fub` element takes all the attributes of the `a` element plus:

ru [Optional] contains the corresponding Unicode if you were to map from Unicode to bytes and then back again.

rc [Optional] contains the Unicode representation of the sequence expressed in the `ru` attribute.

Thus, we could extend our example to:

```
<fub b="69 6A" u="0133" ru="0069 006A"/>
```

There is one more child element of the `assignments` element which is of interest at the moment. This is the `range` element. The `range` element is a sanity saving shortcut. There are often cases where a whole range of values are simply mapped to a corresponding range of Unicode values, using the `a` element. Rather than have to type all those `a` elements, it is possible to use a `range` element instead. The `range` element takes these attributes:

bFirst [Required] the first byte value in the range. This attribute can be a byte value list, for use with ranges in multi-byte encodings.

bLast [Required] the last byte value in the range. If `bFirst` is a byte list, then `bLast` must be a byte list of corresponding length, conforming to the ranges specified in `bMin` and `bMax`.

uFirst [Required] the corresponding first value in the Unicode range to the `bFirst` attribute. This can only be a single Unicode value.

uLast [Required] the corresponding final value in the Unicode range to the `bLast` attribute. This can only be a single Unicode value.

bMin [Optional[3]] this value is required when `bFirst` and `bLast` consist of lists of more than one byte value each. In this case, each byte value in the list is considered as its own incrementing value. When any value exceeds the corresponding value in the `bMax` attribute, it is reset to the corresponding value in `bMin` and the next most previous value is incremented.

bMax [Optional[4]] this value is required when `bFirst` and `bLast` consist of lists of more than one byte value each. When incrementing through the byte values, if any value exceeds the corresponding value in `bMax` it is reset to the corresponding value in `bMin` and the next previous value in the byte value list is incremented and tested and so on.

Notice that the `bMin` and `bMax` value are only of real concern when working with ranges in multi-byte encodings.

For example, here are the first few lines of the `assignments` element in `silipa93.xml`.

```
<assignments sub="3F">
    <range bFirst="00" bLast="1F" uFirst="0000" uLast="001F"/>

    <a b="20" u="0020"/>
    <a b="21" u="030B"/>
```

The `range` element here simply maps the first 32 codes to the corresponding first 32 Unicode values: 0x00–0x1F map to U+0000–U+001F.

Modification History

In addition to the mapping description itself, UTR22 provides for a modification history. The modification history is a required element in a mapping description and consists of a `history` element containing a list of `modified` elements. The `history` element takes no attributes. The `modified` element takes the following:

version [Required] consists of a single integer which corresponds to the `version` attribute in the `characterMapping` element. `modified` elements are stored in reverse order of `version` attribute and the `version` attribute value of the `characterMapping` element must be equal to the highest value of `version` across all the `modified` elements, or the value of the `version` attribute of the first `modified` element.

date [Required] consists of a date in the form: `dd-mm-yyyy` where `dd` is the day of the month as 2 digits, `mm` is the month as 2 digits and `yyyy` is the year (AD). Each component of the date is separated by a hyphen. If more than one modification is made on the same day, then an added letter may be placed after the date.

The contents of the `modified` element is the text describing the modification.

For example, `silipa93.xml` has gone through a number of changes in its history:

```
<history>
    <modified version="6" date="2001-07-17">
        Tidy up and conform to DTD
    </modified>
```

[3] Required in UTR22.

[4] Also required in UTR22.

```
        <modified version="5" date="2001-07-07">
            Fixed dotless reverse mapping
        </modified>
        <modified version="4" date="2001-06-29">
            Changed context names: iwidth -> i-udia, i-ldia -> iwidth. Added more
            than one optional ldia to i-udia and iwidth.
        </modified>
        <modified version="3" date="2001-05-10">
            Changed 0x3D to U+0320 from U+0331
        </modified>
        <modified version="2" date="2001-05-10">
            Still need to deal with ordering issues for over-arch and rhotic hook
        </modified>
        <modified version="1" date="2000-11-08">
            Original. Trying new language extensions
        </modified>
    </history>
```

And undoubtedly there will be more to come!

Validity

In the case of multi-byte encodings, it is possible for there to be badly formed sequences. For example consider a double-byte encoding scheme where a lead byte of a double byte pair is in the range 0x80—0xBF and the second byte is in the range 0xC0-0xFF. Consider what might happen with the following data:

```
4A 90 FC 4B 90 91 FE 92 4C
```

The problem is that following the second `90` the system needs to realize that the `91` is the start of a new multi-byte sequence rather than part of the previous sequence. Likewise there is an error following the `92` code.

In order to tell a mapping processor how to report errors and re-synchronize rather than ploughing on and creating garbage, the UTR22 file format provides a top element called `validity` which contains a small state machine describing a valid sequence of bytes. Since this is only an issue for those concerned with multi-byte encodings, the reader is referred to the UTR22 specification itself for information on this.

But, since the `validity` element is a required element in a UTR22 specification, we list here an example worthy of copying, for those working with a single byte encoding:

```
<validity>
    <state type="FIRST" next="VALID" s="00" e="FF" max="FFFF"/>
</validity>
```

The `state` element describes a state (`FIRST`) and a possible range (`00—FF`) and the state to go to when a byte in the specified range occurs (`VALID`). There are two special states: `FIRST`, which is the initial state that the machine starts in, and `VALID` which is the final state to say that the sequence is valid. The optional `max` attribute contains a Unicode value higher or equal to the highest Unicode value that a byte sequence that is valid, will map to.

In summary, copy the above sample in all your mapping files unless you are working with surrogates (in which case increase the `max` value) or are working with multi-byte encodings.

Completion

With these components in place, we are now in position to create a complete mapping description that is conformant to the basic UTR22. All that remains is to include all the top level elements in the right order:

```
<characterMapping>
    <history>
        :
    </history>
    <validity>
        :
    </validity>
    <assignments>
```

```
        :
    </assignments>
</characterMapping>
```

Any mapping description that conforms precisely to our description so far can be considered to be a pure UTR22 description. The DTD for this description may be found at: `http://www.unicode.org/unicode/reports/tr22/CharacterMapping.dtd`.

Contextual Mapping

In the previous section, we discovered a weakness in the mapping description format. The 4 codes for U+030C are not in free variation. When mapping from Unicode to bytes, we can work out which of the four byte values to use based on contextual information. We know to use an i-width diacritic (0xE0, 0xF3) if the U+030C follows an i-width base character, and we know to use a high form (0xF3, 0xF4) if the U+030C follows another upper diacritic.

If we were to extend the `a` element (and so also `fbu` and `fbu`) to add constraints to the mapping based on contexts we might be able to say something like:

```
<a b="26" u="030C"/>    <!-- new default -->
<a b="E0" u="030C" ubctxt="iwidth"/>
<a b="F3" u="030C" ubctxt="i-udia"/>
<a b="F4" u="030C" ubctxt="udia"/>
```

Each of the mappings is now reversible, but we have constrained the reverse mapping from Unicode to bytes. In all but the first case, the added attribute gives the name of some constraint and says that the mapping from Unicode to bytes, for this byte value, will only happen if the preceding Unicode values in the input stream conform to the contextual constraint of the given name.

Elsewhere in the mapping description, we have a set of context descriptions which are named and are regular expressions describing the valid strings that meet the contextual constraint. For example, the contextual constraint for 0xF4 is described by:

```
<contexts>
    <group id="udia">
        <class-ref name="udia"/>
    </group>

    <class name="udia">0303 0306 0308 030A 033D</class>

</contexts>
```

Here we are saying that there is a contextual constraint named `udia` which is simply an element from the class `udia`. The class `udia` is also defined as being the list of Unicode values. Thus U+030C maps to 0xF4 if the preceding character is one of U+0303, U+0306, U+0308, U+030A, U+033D.

Notice that there is no conflict between the mapping to 0xF4 which has a constraint, and the mapping to 0x26 which has no constraint. Because the mapping to 0xF4 has a longer constraint than the mapping to 0x26, then the 0xF4 result takes precedence over the 0x26 result, with no warning or error necessary.

In the rest of this section we will talk through the details of contexts.

Additions to Associations

First we examine the additional attributes to the `a`, `fbu` and `fub` elements.

ubctxt [Optional] gives the name of a contextual constraint that must match the preceding Unicode characters up to the characters in the `u` attribute, in order for the Unicode values to map to the given byte values. Not valid in a `fbu` element.

uactxt [Optional] gives the name of a contextual constraint that must match the Unicode characters following the characters in the `u` attribute, in order for the Unicode values to map to the given byte values. Notice that if there is also a `ubctxt` attribute, then both constraints must match for the mapping to occur. Not valid in a `fbu` element.

bbctxt [Optional] When mapping from bytes to Unicode, constrains the mapping to only occur if the bytes preceding those in the `b` attribute match the given contextual constraint. Not valid in a `fub` element.

bactxt [Optional] When mapping from bytes to Unicode, constrains the mapping to only occur if the bytes following those in the `b` attribute match the given contextual constraint. Notice that if there is also a `bbctxt` attribute, then both constraints must match for the mapping to occur. Not valid in a `fub` element.

priority [Optional] The default algorithm for deciding which mapping association has precedence in any particular context may not be sufficient for the purposes of a particular mapping description. The `priority` attribute allows the precise priority to be specified in the mapping description. The mapping association with the highest `priority` attribute value takes the highest precedence. The default value for this attribute is 0.

As has been raised, there is always the issue of which mapping association takes precedence in any particular context. There is an algorithm for arriving at this:

- The association with the highest `priority` attribute value takes precedence. If the two values are equal then
- The association with the longest match string (`u` when mapping from Unicode to bytes and `b` when mapping from bytes to Unicode) takes precedence. If the two lengths are the same then
- The association with the longest maximum possible contextual constraint (That is the sum of the maximum lengths of both constraints) takes precedence. If these two lengths are the same, then
- The association that occurs first in the mapping description source file occurs first.

Notice that it is an error for there to be two unconstrained mappings from the same Unicode value to different byte values, and vice versa.

Contexts

The meat of contextual constraints lies in the contexts themselves. Contexts and everything to do with them, including class definitions, are held in a top level `contexts` element. A context is a regular expression described using a `group` element. A `group` element holds a number of child elements either as a sequence (the child elements should match in sequence) or as a set of alternatives (any one of the child elements should match). Thus the `group` element has these attributes:

id [Optional] The name by which this particular group is known to its parent. If this is a top level group, then the `id` attribute is required.

alt [Optional] indicates that the children of this element are alternatives rather than a sequence. The value of this is a boolean, thus can be `1` or `true` or even `0` or `false`. The default value is `0`.

In common with all context describing elements, the `group` element also supports the notion of numbers of occurrence. Thus:

min [Optional] The group must occur this many times. The default is `1`.

max [Optional] When matching, this group may not match more than `max` times. The default for this value is `1`. It may be tempting to allow a value of `unconstrained` for this attribute, but this makes measuring the maximum match length for context nearly impossible, and there are many processors which cannot work with unconstrained matching. Thus the `max` value must be a number.

As we have seen, `group` elements may nest within each other, thus you can build sequences of alternatives of sequences, etc. But what are we building sequences of alternatives of? The simplest building block for describing contexts is the `class-ref` element. This element refers to a `class` element which contains a list of code values. If the code being tested is in this list, then the `class-ref` element matches. The `class-ref` element has these attributes:

name [Required] The name of the class this element refers to.

neg [Optional] The character must not be found in the named class. Notice that in this case, the character is 'consumed' by the test, and the next element will test the next character and not this character again. The lack of a character (e.g. at the end of a string) will cause this test to fail. The `neg` attribute takes a boolean value (see `alt` above).

id, min, max [Optional] as defined for `group`.

A negative match is useful for describing negative contexts, where you are wanting a mapping to occur if a character does not occur in a given context. A common requirement, therefore is to find if a preceding character is not in a given class, say, and we are happy if we are at the start of a string. Thus we need a context element which represents end of string. The `eos` element does this. It has no attributes and only matches at the end of a string. For example, to find whether a character does not occur following a base character we can say:

```
<group id="not_base" alt="1">
    <eos/>
    <class-ref name="base" neg="1"/>
</group>
```

Finally, to save the effort of constantly repeating fragments of regular expressions, there is a `context-ref` element which matches if the corresponding context matches at that point. This element will be considered in much more detail in the section on re-ordering. The attributes for `context-ref` are:

name [Required] the name of the context we are referring to.

id, min, max [Optional] override the corresponding values in the referred context. The meanings are as per the `group` element.

Using these few elements, it is possible to describe all of the standard regular expression patterns used in other programming languages. Therefore, the contexts that this language can describe are powerful and sufficient to meet the descriptive needs of mapping descriptions.

Classes are simply named value lists. In addition they have an attribute describing whether the values listed are byte or Unicode values. Thus the `class` element, which is used as a child of the `contexts` element, has these attributes:

name [Required] The name by which this class is referred to in a `class-ref` element.

size [Optional] Specifies whether the values in the class are `unicode` (default) or `bytes` (needs to be specified).

A class may also include some elements to aid in class building and to save having to retype long lists of values. There are two such elements: the `class-include` element refers to another class and includes all its values in this class. It takes the single required attribute `name` which is used to refer to the other class. The `class-range` element saves having to type a long list of sequential values. It takes these attributes:

first [Required] The first code value in the range.

last [Required] The last code value in the range.

Examples

Let us examine some of the contextual mapping occurring in `silipa93.xml`. We start with the contexts referred to in the mappings:

```
<a b="26" u="030C"/>   <!-- new default -->
<a b="E0" u="030C" ubctxt="iwidth"/>
<a b="F3" u="030C" ubctxt="i-udia"/>
<a b="F4" u="030C" ubctxt="udia"/>
```

The corresponding contexts are:

```
<contexts>
    <group id="iwidth">
      <class-ref name="iwidth"/>
      <class-ref name="ldia" min="0" max="3"/>
```

```
    </group>

    <group id="i-udia">
      <class-ref name="iwidth"/>
      <class-ref name="ldia" min="0" max="3"/>
      <class-ref name="udia"/>
    </group>

   <group id="udia">
      <class-ref name="udia"/>
   </group>

   <class name="iwidth">
      0066 0069 006A 006C 0072 0074 0131 0268 026A 026D 0279 027A 027B 027D
      027E 0283 0284 0288 029D
   </class>
   <class name="ldia">
      0318 0319 031C 031D 031E 031F 0324 0325 0329 032A 032F 0330 0331 0339
      033A 033B 033C
   </class>
   <class name="udia">0303 0306 0308 030A 033D</class>
</contexts>
```

A more interesting example comes about when we consider code 0x22 which is a dotless i. While there is a dotless i character in Unicode (U+0131), I would suggest that for the most part, people are not intending to use that particular character. Instead, the dotless i character has been used instead of a standard i because there is a following upper diacritic. Thus we want to say that when there is a following upper diacritic then code 0x22 maps to a normal i (U+0069) whereas, if there is no following upper diacritic, then the Unicode character for dotless i should be used (U+0131). The following description achieves this:

```
<assignments sub="3F">
    <a b="22" u="0069" bactxt="byte-dia" uactxt="dotless"/>
    <a b="22" u="0131"/>
    <a b="69" u="0069"/>
</assignments>

<contexts>
    <group id="byte-dia">
        <class-ref name="ldiab" min="0" max="3"/>
        <class-ref name="udiab"/>
    </group>

    <group id="dotless">
        <class-ref name="ldia_all" min="0" max="3"/>
        <class-ref name="udia"/>
    </group>

    <class name="ldiab" size="bytes">
        2B 2D 30 31 32 33 34 35 36 37 38 39 3D 60 B1 A2 A3 A4 A5 A6 AA B0 BB
        BC C1 D0 D1
    </class>

    <class name="ldia">
        0318 0319 031C 031D 031E 031F 0324 0325 0329 032A 032F 0330 0331 0339
        033A 033B 033C
    </class>

    <!-- udiab includes tone, udia doesn't -->
    <class name="udiab" size="bytes">
        28 29 2A 5F 7E A1 E1 E2 21 23 24 25 26 40 5E 88 89 8F 90 93 94 98 99
        9D 9E DA DB DC DD DE DF E0 E6 E9 F3 F4
    </class>
    <class name="udia" size="unicode">0303 0306 0308 030A 033D</class>
    <class name="udia_all">
        <class-include name="udia"/>
        0300 0301 0302 0304 030B 030C 030F
    </class>
```

```
</contexts>
```

When mapping from bytes to Unicode, 0x22 maps to U+0069 if there is a following upper diacritic. Otherwise it maps to U+0131. When mapping from Unicode to bytes, then U+0069 maps to 0x22 if there is a following upper diacritic, otherwise it maps to 0x69. Appendix B gives a complete listing of the mapping description for the SILIPA93 encoding.

The contextual mapping extensions to UTR22, therefore, allow us the power to be able to express mappings involved in contextual variant forms. It also allows us to keep a single description and use it when mapping in both directions. The approach we have taken also enables us to clearly see what the basic mapping is that is going on without having to get involved in the details of the constraints and classes. By separating the contextual descriptions from the mapping, it is possible to simply read the b and u fields and to arrive at a rough idea of what is going on without reading anything else. Likewise, it is possible to read the contexts without having to know the contents of the classes, and even the classes are relatively free standing in their definitions.

Re-ordering

This section considers a further level of complexity when describing mappings. It addresses the issue where the canonical order in the legacy encoding is not the same as the canonical order of Unicode, and where the simple application of the normalization algorithm is insufficient to resolve the discrepancy. Thankfully, there are many encodings that do not fall into this category and for many mapping description authors, the issues raised here will not arise. Having said this, it is probably worth the readers while to have at least read this section in order to understand how to tell whether a particular encoding will require re-ordering.

The main discussion will revolve around a contrived example based on the SILIPA93 encoding, but we start with a short excursus into the realm of Devanagari to see why re-ordering is such an important concept.

In Nepali, the word for *thirst* is: तिर्खा which may be transliterated: tirka. In Unicode the canonical ordering results in a spelling akin to tirka (U+0924 U+093F U+0931 U+094D U+0916 U+093E). But for a typical 'dumb' font, due to the positions of the letters, this word would be encoded as itkar. There is no way that the normalization algorithm in Unicode can resolve this issue and so it is necessary to re-order the bytes. This sort of problem occurs regularly with indic scripts, especially where there are preceding diacritic characters (like the *ikar* in Devanagari) which may well be stored much later than they are rendered.

This brings out a general principle:

> *There are two orderings: the byte ordering and the Unicode ordering.*

Indic scripts are not the only scripts which may need re-ordering. Let us consider the case of changing the SILIPA93 encoding very slightly to include a rhotic a ligature. Consider the case of changing the encoding such that we have the following assignment:

```
<a b="C5" u="0061 02DE"/>
```

This seemingly innocent change to the encoding actually results in quite a significant change to the mapping. Consider the Unicode sequence U+0061 U+0303 U+02DE, which is in Unicode canonical order. It represents a nasalised rhotic a. We pass this through our Unicode to bytes mapping and the result is: 0x61 0x29 0xD5. Whoops! What happened to the 0xC5 we were expecting?

The problem is that the superscript tilde diacritic code got between the U+0061 and the U+02DE. The result is that the conversion engine did not see a sequence U+0061 U+02DE and so did not do the conversion. What needs to happen is that the rhotic hook (U+02DE) needs to be re-ordered to occur after the base character, just for the conversion phase. I.e. there is a third ordering: the conversion ordering. Thus we extend our basic principle to:

> *There are three orderings: the byte ordering, the conversion ordering and the Unicode ordering.*

There is no need to worry. These are the only ones! It is also highly probable that two or even all three of the orderings are the same.

The Unicode ordering is generally very clear and unambiguous. There is only one correct ordering of codes to represent a particular spelling. This is called the canonical ordering, and the normalisation algorithm may or may not be sufficient to do the re-ordering. The conversion ordering needs to ensure that all the components of any ligatures are brought together so that the conversion process can see them and create a ligature (or vice versa, split a ligature into a suitably ordered sequence). All contextual constraints are matched against the text in this order. Finally, the byte ordering can well be the most problematic, since the byte encoding is usually visually motivated, any code ordering is considered acceptable so long as the text looks right. Having said this, most complex encodings demand some level of constrained ordering, if nothing else, to aid in string comparison.

For our contrived example, we are going to say that the byte ordering is the same as the Unicode ordering (which, fortuitously is how it turned out in real life) and that the conversion ordering requires that the rhotic hook occur immediately after the base character.

The UTR22 extensions allow for two re-ordering passes. One occurs between the byte side and the conversion and the other between the conversion and the normalization pass (if any).

Ordering

A re-ordering pass is described as an `assignments` element. There can be up to two `ordering` elements and they must occur after all the `a` elements. Each `ordering` element takes these attributes:

side [Required] This attribute describes which side of the conversion pass, this re-ordering pass should occur. The allowed values are: `unicode` or `bytes`.

Within each `ordering` element, there can be any number of `order` elements. Each `order` element describes two contexts, one on the `bytes` side and one on the `unicode` side. Notice that there is always a byte side and Unicode side regardless of where the re-ordering occurs. The two contexts are linked, as we shall see, and depending upon which direction the data is being transformed, the appropriate context is tested. If it matches, then the corresponding context for the other side describes the relative order that the various parts matched should be re-ordered to. Each `order` element is tested in order and if no match is found at a particular point in the text string, then the next `order` element is tested, until all the `order` elements have been tested. If there is still no match, then the next character in the text string is treated as the first character for contextual matching and all the `order` elements are tried in sequence again. This process is repeated until the whole string has been tested. If a context matches, then all the next position to test against is the position immediately following the string matched.

The `order` element takes these attributes:

b [Required] context for the byte side.

u [Required] context for the Unicode side.

bctxt a pre context for this re-ordering rule.

actxt a following context for this re-ordering rule

For our example, we would add the following elements after the last `a` (or `fbu` or `fub`) element.

```
<ordering side="unicode">
   <order b="clusU-b" u="clusU-u"/>
</ordering>

<ordering side="bytes">
   <order b="clusB-b" u="clusB-u"/>
</ordering>
```

The naming of the contexts is arbitrary, although I have tried to bring a little order to the chaos!

The optional context attributes are like normal conversion contexts in that the strings represented take no part in the re-ordering. Their primary use is when to re-ordered strings are adjacent, but where there are characters in the first string that decide whether the second string should be re-ordered. I.e. a pre context is searched for backwards, before the re-ordered string into previously processed text. Likewise a following context is not considered part of the re-ordered text.

Contexts

The real meat of re-ordering occurs in the contexts.

A context is simply a regular expression describing a string that if matched needs to be re-ordered. For example, on the Unicode side we might have the following context to describe a base character, its diacritics followed by a rhotic hook (U+02DE).

```
<group id="clusU-u">
    <class-ref name="base"/>
    <class-ref name="ldia" min="0" max="3"/>
    <class-ref name="udia_all" min="0" max="4"/>
    <class-ref name="rhotic"/>
</group>
```

The class definitions `base` and `rhotic` are left as an exercise for the reader.

OK. We have a regular expression describing a cluster with a rhotic hook, but how do we ensure the re-ordering happens? The answer is that the other context needs to use `context-ref` elements to refer to the various sub expressions in the other context. That way, the conversion processor can tell which parts of the two regular expressions tie up. In order to do that, first we need to give the various sub-expressions in our context `id` attributes, so that the `context-ref` elements can refer to them:

```
<group id="clusU-u">
    <class-ref name="base" id="base"/>
    <class-ref name="ldia" min="0" max="3" id="ldia"/>
    <class-ref name="udia_all" min="0" max="4" id="udia"/>
    <class-ref name="rhotic" id="rhotic"/>
</group>
```

And now we can write the corresponding group for `clusU-b`:

```
<group id="clusU-b">
    <context-ref name="clusU-u/base"/>
    <context-ref name="clusU-u/rhotic"/>
    <context-ref name="clusU-u/ldia"/>
    <context-ref name="clusU-u/udia"/>
</group>
```

Here we see the syntax of a context reference in action. Context references are based on the `id` attributes of context elements. A context reference may reference a particular element using a path of `id` attributes through all the groups, from the top, down to the particular sub-expression of interest, which can be a simple `class-ref`. The parts of the path are separated by the `/` character, which explains why an `id` attribute cannot have a `/` in it.

If we examine these two groups, we can see that if we are re-ordering from the conversion side towards Unicode, then we would expect to match a base character followed by a rhotic hook followed by some lower diacritics and/or upper diacritics perhaps and have the rhotic hook re-ordered after the diacritics. The conversion processor manages to work everything out, so long as we get the context references right.

The corresponding contexts on the byte side: `clusB-b` and `clusB-u`, are left as an exercise for the reader.

Optimization

For our particular example, all this re-ordering seems a lot of work for one simple ligature, is there some way of reducing the amount of work that needs to be done and to speed things up? Well, there is. The previous sections showed the long-winded solution in order to present the basic principles and procedures, which would be used for any script. In most cases, particularly with Indic scripts, it takes all of that, and at a deeper level of complexity, to address the re-ordering issues of those scripts. For our particular example, though, we can simplify the solution drastically.

One thing to notice in our problem is that we only really need to re-order the rhotic hook if the base character is an `a`. Thus we can re-write our re-ordering contexts to work just with an `a`:

```
<group name="clusU-u">
    <class-ref name="a" id="a"/>
```

```
    <class-ref name="ldia" min="0" max="3" id="ldia"/>
    <class-ref name="udia_all" min="0" max="4" id="udia"/>
    <class-ref name="rhotic" id="rhotic"/>
</group>

<group name="clusU-b">
    <context-ref name="clusU-u/a"/>
    <context-ref name="clusU-u/rhotic"/>
    <context-ref name="clusU-u/ldia"/>
    <context-ref name="clusU-u/udia"/>
</group>
```

We can also notice that if we re-order the rhotic hook to be after the `a` then it will always been turned into a ligature, which is what we want. But since in all other cases, the rhotic hook is not re-ordered, then it is already in the right place for the byte side ordering. The upshot of all this reasoning is that there is no need to do any re-ordering on the byte side at all.

Conclusion

The re-order, convert, re-order model of data transformation is a very powerful one. It allows the various components to be reversible and for the descriptions to be clear. There is no need for insertion and deletion rules, which can be particularly problematic, in the case of insertion. The processing models are simplified and a relatively small language can be used to express very complex transformations. In addition, users only need work with those aspects of the language which are needed to support the complexity of transformation they need to do.

Appendix A

The encoding chart for the SILIPA93 encoding:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	000	016	032	048 ◌̃	064 ◌́	080 ɵ	096 ◌ˌ	112 p	128 ◌̀	144 ◌́	160 □	176 ◌̪	192 ʕ	208 ◌̈	224 ◌̌	240 ħ	0
1	001	017	033 ◌̋	049 ◌̟	065 ɑ	081 æ	097 a	113 q	129 ɒ	145 ◌̍	161 °	177 ◌̟	193 ◌̟	209 ◌̎	225 ◌̆	241 ɭ	1
2	002	018	034 ɪ	050 ◌̠	066 β	082 ɾ	098 b	114 r	130 ɘ	146 ‖	162 ◌̺	178 ɴ	194 ɬ	210 ʁ	226 ◌̃	242 ~	2
3	003	019	035 ◌̄	051 ◌̠	067 ç	083 ʃ	099 c	115 s	131 ◌͡◌	147 ◌̄	163 ◌̪	179 ʕ	195 ʌ	211 ɻ	227 ʍ	243 ◌̬	3
4	004	020	036 ◌̀	052 ◌̺	068 ð	084 θ	100 d	116 t	132 ‖	148 ◌̲	164 ◌̬	180 ʎ	196 ɣ	212 ŋ	228 ɻ	244 ◌̬	4
5	005	021	037 ◌̎	053 ◌̻	069 ɛ	085 ʊ	101 e	117 u	133 ◌̍	149 ◌̍	165 ◌̥	181 ɯ	197 □	213 ◌̰	229 ɯ	245 ʙ	5
6	006	022	038 ◌̬	054 ◌̪	070 ɤ	086 ʊ	102 f	118 v	134 ◌̏	150 ◌̀	166 ◌̯	182 □	198 ʝ	214 -	230 ◌̂	246 ɨ	6
7	007	023	039 ’	055 ◌̹	071 G	087 w	103 g	119 w	135 ⊙	151 !	167 ʂ	183 □	199 ◌ˌ	215 ʃ	231 ɥ	247 ɳ	7
8	008	024	040 ◌̆	056 ◌̥	072 h	088 χ	104 h	120 x	136 ◌̋	152 ◌̀	168 ɹ	184 ɸ	200 ˈ	216 ɺ	232 ʎ	248 ɲ	8
9	009	025	041 ~	057 ◌͡	073 ɪ	089 ʏ	105 i	121 y	137 ◌̋	153 ◌̀	169 ɡ	185 ʂ	201 m	217 ɻ	233 ◌̂	249 ː	9
A	010	026	042 °	058 l	074 j	090 ʒ	106 j	122 z	138 ◌̀	154 ◌̍	170 ◌̠	186 ɓ	202	218 ◌̋	234 ɖ	250 ɦ	A
B	011	027	043 ◌̟	059 L	075 H	091 [	107 k	123 ʀ	139 ↑	155 ↓	171 ə	187 ◌͡	203 ɲ	219 ◌́	235 ɗ	251 ʔ	B
C	012	028	044 ,	060 n	076 ɮ	092 \	108 l	124 ◌̀	140 ɐ	156 ǂ	172 ʉ	188 ◌̃	204 ↗	220 ◌̄	236 ɣ	252 ʐ	C
D	013	029	045 ◌̈	061 ◌̲	077 ɱ	093]	109 m	125 ɽ	141 ɔ	157 ◌̋	173 ◌̲	189 ʐ	205 ↘	221 ◌̀	237 ◌̑	253 ɠ	D
E	014	030	046 .	062 ˈ	078 ŋ	094 ◌̂	110 n	126 ×	142 ǀ	158 ◌̋	174 ɬ	190 ɟ	206 ɜ	222 ◌̋	238 ɧ	254 ɕ	E
F	015	031	047 /	063 ʔ	079 ø	095 ◌̈	111 o	127 □	143 ◌́	159 ɻ	175 Œ	191 œ	207 ɘ	223 ◌̂	239 ɟ	255 ʈ	F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Appendix B

This is the complete mapping description for the SILIPA93 encoding.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="utr_display.xsl"?>
<!DOCTYPE characterMapping SYSTEM "CharacterMapping.dtd">
<characterMapping
    id="SIL-IPA93-2001"
    version="10"
    description="SIL IPA93 Font encoding"
    contact="mailto:martin_hosken@sil.org"
    registrationAuthority="SIL International"
    registrationName="IPA93"
    byte-font="SILDoulos IPA93"
    normalization="NFD">

<history>
    <modified version="10" date="2002-10-7">
        0X43 maps to decomposed sequence rather than composed form
        version number corrected
    </modified>
    <modified version="9" date="2002-09-18">
        Add explicit mappings for U+0334 precomposed characters. Correct 0xCF
        from being U+029A.
        g maps to U+0261 by default (significant shape change). Add U+0320,
        U+032C to ldia, remove U+0331.
        Remove 0xF2 mapping to indicate how naughty it is to use U+0334 nowadays.
    </modified>
    <modified version="8" date="2002-09-17">
        Add itall and tall classes, change regexps accordingly. Changed 0xAD to
        U+0320 from U+0331
    </modified>
    <modified version="7" date="2001-08-23">
        Finalise PUA codes to SIL corporate standard
    </modified>
    <modified version="6" date="2001-07-17">
        Tidy up and conform to DTD
    </modified>
    <modified version="5" date="2001-07-07">
        Fixed dotless reverse mapping
    </modified>
    <modified version="4" date="2001-06-29">
        Changed context names: iwidth -> i-udia, i-ldia -> iwidth. Added more
        than one optional ldia to i-udia and iwidth.
    </modified>
    <modified version="3" date="2001-05-10">
        Changed 0x3D to U+0320 from U+0331
    </modified>
    <modified version="2" date="2001-05-10">
        Still need to deal with ordering issues for over-arch and rhotic hook
    </modified>
    <modified version="1" date="2000-11-08">
        Original. Trying new language extensions
    </modified>
</history>

<validity>
    <state type="FIRST" next="VALID" s="00" e="FF" max="FFFF"/>
</validity>

<assignments sub="3F">
    <range bFirst="00" bLast="1F" uFirst="0000" uLast="001F"/>
    <a b="20" u="0020"/>
    <a b="21" u="030B"/>
    <a b="22" u="0069" bactxt="byte-dia" uactxt="dotless"/>
```

```
<a b="22" u="0131"/>
<a b="23" u="0304"/>
<a b="24" u="0300"/>
<a b="25" u="030F"/>
<a b="26" u="030C"/>
<a b="27" u="02BC"/>
<a b="28" u="0306"/>
<a b="29" u="0303"/>
<a b="2A" u="030A"/>
<a b="2B" u="031F"/>
<a b="2C" u="002C"/>
<a b="2D" u="0324"/>
<a b="2E" u="002E"/>
<a b="2F" u="002F"/>

<a b="30" u="0330"/>
<a b="31" u="0318"/>
<a b="32" u="0319"/>
<a b="33" u="031D"/>
<a b="34" u="031E"/>
<a b="35" u="032A"/>
<a b="36" u="033B"/>
<a b="37" u="031C"/>
<a b="38" u="0325"/>
<a b="39" u="032F"/>
<a b="3A" u="02E1"/>
<a b="3B" u="029F"/>
<a b="3C" u="207F"/>
<a b="3D" u="0320"/>
<a b="3E" u="02D1"/>
<a b="3F" u="0294"/>

<a b="40" u="0301"/>
<a b="41" u="0251"/>
<a b="42" u="03B2"/>
<a b="43" u="0063 0327"/>
<a b="44" u="00F0"/>
<a b="45" u="025B"/>
<a b="46" u="0264"/>
<a b="47" u="0262"/>
<a b="48" u="02B0"/>
<a b="49" u="026A"/>
<a b="4A" u="02B2"/>
<a b="4B" u="029C"/>
<a b="4C" u="026E"/>
<a b="4D" u="0271"/>
<a b="4E" u="014B"/>
<a b="4F" u="00F8"/>

<a b="50" u="0275"/>
<a b="51" u="00E6"/>
<a b="52" u="027E"/>
<a b="53" u="0283"/>
<a b="54" u="03B8"/>
<a b="55" u="028A"/>
<a b="56" u="028B"/>
<a b="57" u="02B7"/>
<a b="58" u="03C7"/>
<a b="59" u="028F"/>
<a b="5A" u="0292"/>
<a b="5B" u="005B"/>
<a b="5C" u="005C"/>
<a b="5D" u="005D"/>
<a b="5E" u="0302"/>
<a b="5F" u="0308"/>

<a b="60" u="0329"/>
<a b="61" u="0061"/>
```

```
<a b="62" u="0062"/>
<a b="63" u="0063"/>
<a b="64" u="0064"/>
<a b="65" u="0065"/>
<a b="66" u="0066"/>
<a b="67" u="0261"/>
<a b="68" u="0068"/>
<a b="69" u="0069"/>
<a b="6A" u="006A"/>
<a b="6B" u="006B"/>
<a b="6C" u="006C"/>
<a b="6D" u="006D"/>
<a b="6E" u="006E"/>
<a b="6F" u="006F"/>

<a b="70" u="0070"/>
<a b="71" u="0071"/>
<a b="72" u="0072"/>
<a b="73" u="0073"/>
<a b="74" u="0074"/>
<a b="75" u="0075"/>
<a b="76" u="0076"/>
<a b="77" u="0077"/>
<a b="78" u="0078"/>
<a b="79" u="0079"/>
<a b="7A" u="007A"/>
<a b="7B" u="0280"/>
<a b="7C" u="031A"/>
<a b="7D" u="027D"/>
<a b="7E" u="033D"/>
<a b="7F" u="007F"/>

<a b="80" u="02E9 02E7"/>
<a b="81" u="0252"/>
<a b="82" u="0258"/>
<a b="83" u="0361"/>
<a b="84" u="2016"/>  <!-- check me -->
<a b="85" u="02E5 02E7"/>
<a b="86" u="02E5 02E9"/>
<a b="87" u="0298"/>
<a b="88" u="030B" ubctxt="udia"/>
<a b="89" u="030B" ubctxt="i-udia"/>
<a b="8A" u="02E5"/>
<a b="8B" u="2191"/>
<a b="8C" u="0250"/>
<a b="8D" u="0254"/>
<a b="8E" u="01C0"/>
<a b="8F" u="0301" ubctxt="udia"/>

<a b="90" u="0301" ubctxt="i-udia"/>
<a b="91" u="02E6"/>
<a b="92" u="01C1"/>
<a b="93" u="0304" ubctxt="udia"/>
<a b="94" u="0304" ubctxt="i-udia"/>
<a b="95" u="02E7"/>
<a b="96" u="007C"/>
<a b="97" u="01C3"/>
<a b="98" u="0300" ubctxt="udia"/>
<a b="99" u="0300" ubctxt="i-udia"/>
<a b="9A" u="02E8"/>
<a b="9B" u="2193"/>
<a b="9C" u="01C2"/>
<a b="9D" u="030F" ubctxt="udia"/>
<a b="9E" u="030F" ubctxt="i-udia"/>
<a b="9F" u="02E9"/>

<a b="A1" u="030A" ubctxt="iwidth"/>
<a b="A2" u="031E" ubctxt="iwidth"/>
```

```
<a b="A3" u="031D" ubctxt="iwidth"/>
<a b="A4" u="032C"/>
<a b="A5" u="0325" ubctxt="iwidth"/>
<a b="A6" u="0339"/>
<a b="A7" u="0282"/>
<a b="A8" u="0279"/>
<a b="A9" u="0260"/>
<a b="AA" u="0319" ubctxt="iwidth"/>
<a b="AB" u="0259"/>
<a b="AC" u="0289"/>
<a b="AD" u="0320" ubctxt="iwidth"/>
<a b="AE" u="0268" uactxt="dotless"/>
<a b="AF" u="0276"/>

<a b="B0" u="033A"/>
<a b="B1" u="031F" ubctxt="iwidth"/>
<a b="B2" u="0274"/>
<a b="B3" u="02E4"/>
<a b="B4" u="028E"/>
<a b="B5" u="026F"/>
<a b="B8" u="0278"/>
<a b="B9" u="02A2"/>
<a b="BA" u="0253"/>
<a b="BB" u="032F" ubctxt="iwidth"/>
<a b="BC" u="0330" ubctxt="iwidth"/>
<a b="BD" u="0290"/>
<a b="BE" u="006A" uactxt="dotless"/>
<a b="BF" u="0153"/>

<a b="C0" u="0295"/>
<a b="C1" u="0318" ubctxt="iwidth"/>
<a b="C2" u="026C"/>
<a b="C3" u="028C"/>
<a b="C4" u="0263"/>
<a b="C6" u="029D"/>
<a b="C7" u="02CC"/>
<a b="C8" u="02C8"/>
<a b="C9" u="F180"/>  <!-- superscript m -->
<a b="CA" u="200A"/>
<a b="CB" u="F181"/>  <!-- superscript nya -->
<a b="CC" u="2197"/>
<a b="CD" u="2198"/>
<a b="CE" u="025C"/>
<a b="CF" u="025E"/>

<a b="D0" u="0324" ubctxt="iwidth"/>
<a b="D1" u="033C"/>
<a b="D2" u="0281"/>
<a b="D3" u="027B"/>
<a b="D4" u="F182"/>  <!-- superscript eng -->
<a b="D5" u="02DE"/>
<a b="D6" u="002D"/>
<a b="D7" u="0284"/>
<a b="D8" u="02E7 02E5"/>
<a b="D9" u="02E7 02E9"/>
<a b="DA" u="030B" ubctxt="iwidth"/>
<a b="DB" u="0301" ubctxt="iwidth"/>
<a b="DC" u="0304" ubctxt="iwidth"/>
<a b="DD" u="0300" ubctxt="iwidth"/>
<a b="DE" u="030F" ubctxt="iwidth"/>
<a b="DF" u="0302" ubctxt="iwidth"/>

<a b="E0" u="030C" ubctxt="iwidth"/>
<a b="E1" u="0306" ubctxt="iwidth"/>
<a b="E2" u="0303" ubctxt="iwidth"/>
<a b="E3" u="028D"/>
<a b="E4" u="027A"/>
<a b="E5" u="0270"/>
```

```
    <a b="E6" u="0302" ubctxt="i-udia"/>
    <a b="E7" u="0265"/>
    <a b="E8" u="02E9 02E5"/>
    <a b="E9" u="0302" ubctxt="udia"/>
    <a b="EA" u="0256"/>
    <a b="EB" u="0257"/>
    <a b="EC" u="02E0"/>
    <a b="ED" u="203F"/>
    <a b="EE" u="0267"/>
    <a b="EF" u="025F"/>

    <a b="F0" u="0127"/>
    <a b="F1" u="026D"/>
    <a b="6C F2" u="026B"/>
    <!--   <a b="F2" u="0334"/>    -->
    <a b="F3" u="030C" ubctxt="i-udia"/>
    <a b="F4" u="030C" ubctxt="udia"/>
    <a b="F5" u="0299"/>
    <a b="F6" u="0268"/>
    <a b="F7" u="0273"/>
    <a b="F8" u="0272"/>
    <a b="F9" u="02D0"/>
    <a b="FA" u="0266"/>
    <a b="FB" u="02A1"/>
    <a b="FC" u="0291"/>
    <a b="FD" u="029B"/>
    <a b="FE" u="0255"/>
    <a b="FF" u="0288"/>
</assignments>

<contexts>
    <group id="i-udia" alt="1">
        <group>
            <class-ref name="ilow"/>
            <class-ref name="ldia" min="0" max="3"/>
            <class-ref name="udia"/>
        </group>
        <group>
            <class-ref name="itall"/>
            <class-ref name="ldia" min="0" max="3"/>
        </group>
    </group>

    <group id="iwidth">
        <class-ref name="ilow"/>
        <class-ref name="ldia" min="0" max="3"/>
    </group>

    <group id="byte-dia">
        <class-ref name="ldiab" min="0" max="3"/>
        <class-ref name="udiab"/>
    </group>

    <group id="dotless">
        <class-ref name="ldia" min="0" max="3"/>
        <class-ref name="udia_all"/>
    </group>

    <group id="udia" alt="1">
        <class-ref name="udia"/>
        <group>
            <class-ref name="tall"/>
            <class-ref name="ldia" min="0" max="3"/>
        </group>
    </group>

    <class name="ilow">
        0069 006A 0072 0131 0268 026A 0279 027B 027D 027E 029D
```

```
    </class>

    <class name="itall">
       0066 006C 0074 026D 027A 0283 0284 0288
    </class>

    <class name="tall">
       0062 0064 0068 006B 00F0 0127 01C3 0253 0256 0257 0260 0266 0267 026C
       0278 028E 0294 0295 02A1 02A2 02BE 03B2 03B8
    </class>

    <class name="ldiab" size="bytes">
       2B 2D 30 31 32 33 34 35 36 37 38 39 3D 60 B1 A2 A3 A4 A5 A6 AA B0 BB
       BC C1 D0 D1
    </class>

    <class name="ldia">
       0318 0319 031C 031D 031E 031F 0320 0324 0325 0329 032A 032C 032F 0330
       0339 033A 033B 033C
    </class>

    <!-- udiab includes tone, udia doesn't -->
    <class name="udiab" size="bytes">
       28 29 2A 5F 7E A1 E1 E2 21 23 24 25 26 40 5E 88 89 8F 90 93 94 98 99
       9D 9E DA DB DC DD DE DF E0 E6 E9 F3 F4
    </class>

    <class name="udia">0303 0306 0308 030A 033D</class>

    <class name="udia_all">
       <class-include name="udia"/>
       0300 0301 0302 0304 030B 030C 030F
    </class>

  </contexts>
  </characterMapping>
```

Change History

14/Nov/2002	original complete description
7/Jul/2004	Add actxt and bctxt attributes to order elements