The TECkit Language
Mapping byte encodings to Unicode

Jonathan Kew
SIL Non-Roman Script Initiative (NRSI)

Abstract

TECkit is a toolkit for encoding conversions, primarily intended for converting data between 8-bit legacy encodings and Unicode. This document presents the mapping description language supported by the TECkit mapping table compiler.

Introduction

The TECkit package is based on a compiled binary mapping file that contains the tables needed to map between a legacy (byte) encoding and Unicode. The package includes a mapping compiler that allows such mapping tables to be built from fairly simple text descriptions.

The TECkit language is built around simple mapping rules where a legacy character code on the left-hand side of the rule is mapped to or from a Unicode character on the right-hand side. From this basic structure, mapping rules can be extended by the use of character sequences rather than single characters on either side; by the addition of contextual constraints (environments) determining when a rule should apply; and by the use of character classes, optional and repeatable elements, grouping and alternation to express more complex patterns to be matched and processed.

The TECkit package, including the mapping description language, can also be applied to text processing operations entirely dealing with legacy-encoded data or purely Unicode data. Examples include conversion between “logical” and “presentation-form” versions of legacy data, or transliteration of Unicode between different scripts.

Overall file structure and conventions

A TECkit description file is strictly line-oriented; every statement is confined to a single logical line. To allow long rules to be broken across several lines, for easier editing, the compiler interprets a final backslash (`\`) as a “continuation character”; however, only quite complex mappings are likely to need rules that cannot readily be expressed in a single source line.

The semicolon (`;`) introduces a comment that continues to the end of the (physical) line; the compiler ignores everything following a semicolon (unless it is in a quoted string).

Built-in keywords in the TECkit mapping language are not case-sensitive; the compiler will accept any mixture of upper and lower case. This also applies to Unicode character names (of which more later). However, the names of character classes (also described later) defined in the file itself are case-sensitive, and must be used in a consistent form.

TECkit mapping source files may be created as either byte-encoded or Unicode text. The best option is to use ASCII (which will be read as bytes). If ASCII is not sufficient then use UTF-8 without a Byte Order Mark (BOM). The compiler recognizes Unicode source files by looking for an initial BOM or encoding signature, and accepts any of the standard Unicode encoding forms (UTF-8, UTF-16, UTF-32, either big- or little-endian). UTF-16/UTF-32 files with no BOM are also recognized automatically. (To compile UTF-8 source that lacks an encoding signature, the –u flag must be specified on the compiler command line.)

Where "strings" are called for, these may be either single- or double-quoted. There is no mechanism to "escape" quote marks embedded in the string; therefore, a single-quoted string can contain double-quote characters, and vice versa, but it is not possible to include both single and double quotes in the same quoted string.

Byte (legacy) character codes are expressed as decimal numbers (no special marking needed) or as hexadecimal (prefixed by "0x").

Unicode character codes (USVs) are expressed either numerically or using Unicode character names, converted into unique "identifiers" by replacing spaces and hyphens with underscores. The compiler "knows" the complete set of Unicode character names, with the exception of the algorithmically-derived names for CJK characters. The preferred form for USVs is to write "U+*xxxx*", where *xxxx* represents four to six hexadecimal digits (although normal decimal or hex numbers are also permitted).

Characters may also be expressed as quoted literals. If the mapping source is byte-encoded text, then quoted literals may be used only for byte values; and if the mapping source is Unicode text, then they may be used only for Unicode character values. (It is never legal to use quoted literals on both the byte and Unicode sides of the mapping.)

A complete TECkit mapping description consists of a header section followed by one or more mapping passes. Passes may operate entirely within the byte encoding world, entirely in Unicode, or may bridge the byte/Unicode barrier. The simplest legacy/Unicode mapping descriptions will contain just one byte/Unicode pass, but for some complex mappings it may be necessary to perform pre- and/or post-processing such as character reordering in other passes. The LHS code space of each pass must correspond to the RHS code space of the pass before it.

The complete process of mapping from bytes to Unicode involves applying the mapping rules from LHS to RHS from each pass, with the output of each pass becoming the input of the next, in the order found in the description file. To map from Unicode to bytes, the rules are used to map RHS to LHS, and the "pipeline" of passes is reversed, using the passes in the opposite of the order found in the description. (This seems much easier to understand in practice than it is to describe!)

Header information

The file begins with header information, which consists of a number of pieces of information about the encoding and mapping, each specified by a keyword followed by a quoted string:

EncodingName	a canonical name that uniquely identifies this mapping table from all others; the recommended form is a three-part string: SOURCE-NAME_ON_SOURCE-VERSION where SOURCE is the name of the standards authority, product, government, etc., that defined this encoding; NAME_ON_SOURCE is the name most commonly used on that source to identify the encoding; and VERSION is the version number (as encodings often lack explicit version numbers, the year the encoding was introduced is a suggested value here). The three parts of the name are separated with hyphens; non-alphanumeric characters in the name should be replaced with underscores.

DescriptiveName	a string that describes the mapping, to help users distinguish it from other similar mappings or understand its purpose
Version	the version of the mapping description; this should be incremented each time the mapping is revised
Contact	contact information for the person responsible for the mapping (a mailto: URL would be a typical form for this)
RegistrationAuthority	the organization responsible for the encoding
RegistrationName	the name and version of the mapping, as recognized by that authority
Copyright	copyright information

Header information

http://www.iana.org/assignments/character-sets lists "official" names for recognized encodings. For "private" encodings a unique name should be constructed following a similar pattern.

Only the encoding name is required; however, it is recommended that as many of the header fields as are meaningful for any particular mapping should be included.

It is recommended that the header strings should be limited to ASCII characters; this applies especially to the canonical name, which should be as "portable" as possible. If non-ASCII characters are required in other fields (such as the description), they should be represented using UTF-8. Note that if the mapping source is byte-encoded, the header strings are not converted in any way; the literal bytes found in the source are put into the compiled table; it is therefore the responsibility of the author to ensure that the strings represent valid UTF-8.

An alternative form of header should be used for mapping descriptions that do not represent the mapping between a legacy byte encoding and Unicode (e.g., conversions between different byte encodings, or transliterations entirely within Unicode). Instead of EncodingName and DescriptiveName, the following four fields are used:

LHSName	canonical name of the "source" encoding or left-hand side of the description
RHSName	canonical name of the "target" encoding or right-hand side of the description
LHSDescription	description for the left-hand side of the mapping
RHSDescription	description for the right-hand side of the mapping

Header information for non-Byte/Unicode mappings

For the normal case of a mapping between a legacy byte encoding and Unicode, EncodingName and DescriptiveName correspond to the LHSName and LHSDescription fields (the keywords are synonymous), and the RHSName and RHSDescription fields are filled in by the compiler with strings indicating Unicode as the RHS encoding.

Note that while we sometimes think of the left-hand side of the description as "source" and the right-hand side as "target", with the legacy encoding being the source and Unicode the target of the mapping, TECkit descriptions and mapping tables are bi-directional, and thus these roles can equally well be exchanged.

Finally, the file header can include "flags" that specify certain features of the encoding for both the left- and right-hand sides of the mapping.

LHSFlags (*list-of-flags*)	features of the LHS encoding
RHSFlags (*list-of-flags*)	features of the RHS encoding

Header flags statements

For each side of the mapping, zero or more of the following flags can be specified:

ExpectsNFC	input on this side of the mapping should be in fully-composed form
ExpectsNFD	input on this side of the mapping should be in fully-decomposed form
GeneratesNFC	output on this side of the mapping is fully-composed
GeneratesNFD	output on this side of the mapping is fully-decomposed
VisualOrder	this side of the mapping deals with visual (rather than logical) text order

Encoding flags

The “expects” flags can be used to specify that Unicode input to this side of the mapping should be normalized before it is presented to the actual mapping rules. By specifying a normalization form for the Unicode side of a mapping description, the author can write mapping rules assuming a particular canonical representation. The TECkit engine will take care of normalizing the input text so that it matches the expectation of the rules.

The “generates” flags allow the mapping author to declare which normalization form will be produced by the mapping rules. However, as it can be difficult to ensure the accuracy of this, TECkit does not “trust” this flag, but always explicitly normalizes the output if requested by the application using the mapping.

These flags are ignored for byte encodings, as no standard normalization process is defined for bytes, although they could be specified purely for informational purposes in the case of a byte encoding that supports both composed and decomposed forms.

A typical example of the header information might be:

```
EncodingName            "SIL-GREEK_BASIC-2002"
DescriptiveName         "SIL Basic Greek (no precomposed display forms)"
Version                 "1"
Contact                 "mailto:nrsi@sil.org"
RegistrationAuthority   "SIL International"
RegistrationName        "SIL Basic Greek"
Copyright               "(c)2002 SIL International"

LHSFlags                ()
RHSFlags                (ExpectsNFD GeneratesNFD)
```

Mapping passes

The heart of a mapping description is the series of mapping passes (just one, in simple cases) that relate characters or sequences on the LHS to those on the RHS.

Each pass begins with a header line that declares the encoding space in which it operates:

```
pass( pass-type )
```

where *pass-type* is one of:

```
Byte
Unicode
Byte_Unicode
Unicode_Byte
```

The *Unicode_Byte* pass type is defined for completeness, but would not normally be used; mapping from Unicode to bytes is accomplished by using the *Byte_Unicode* description “in reverse”. Therefore, a description for the mapping between a legacy byte encoding and Unicode will typically consist of zero or more *Byte* passes, one *Byte_Unicode* pass, and zero or more *Unicode* passes.

There are also special “normalization pass” types that can be used in special cases. To create a normalization pass, specify *pass-type* as one of:

```
NFC_fwd   NFD_fwd
NFC_rev   NFD_rev
NFC       NFD
```

As the names suggest, these apply the NFC or NFD Unicode normalization forms as part of the forward, reverse, or both processing “pipelines”. Most mappings will not need to include explicit normalization passes, as the ExpectsNFC or ExpectsNFD flag can be used to request pre-normalization of Unicode data before any mapping rules are applied, and applications using TECkit can explicitly request either NFC or NFD data when mapping to Unicode. The only reason to use a normalization pass in a mapping description would be to ensure that data is in a particular normalization form somewhere in the middle of a multi-pass Unicode transduction.

For compatibility with the original TECkit 1.0 language (as supported by the prototype TECkit release in 2000), in the case where the description requires just a single *Byte_Unicode* pass, the *pass* line may be omitted altogether; if class definitions and mapping rules are found with no *pass* line, an implicit *pass(Byte_Unicode)* is assumed.

Class definitions

Character classes may be used to make the mapping description more readable and concise; suitable class definitions allow a single rule to express a whole set of related mappings. They are typically used in contextual constraints or as elements of rules that reorder character sequences.

Classes are defined with the *ByteClass* or *UniClass* statements, depending on the type of characters they are to contain:

```
ByteClass [ name ] = ( byteSequence )
UniClass [ name ] = ( unicodeSequence )
```

Class names, always enclosed in square brackets, are “identifiers” that may contain letters, digits, and the underscore character; they may not begin with a digit. Unlike the keywords of the TECkit language, they *are* case-sensitive. The byte and Unicode sequences are space-separated lists of character codes, similar to those used in mapping rules (see below), with the addition of a “range” notation: two character codes separated by `..` represent the complete set of characters from the first to the second (inclusive). In byte classes, a quoted string may also be used to represent a list of individual byte values.

Note that byte and Unicode classes are completely unrelated, and their names are in separate “namespaces”. It may often be convenient to create corresponding classes of both types with the same name:

```
ByteClass [control] = ( 0..31 127 )
UniClass [control] = ( U+0000..U+001f U+007f )
ByteClass [letter] = ( 'A'..'Z' 'a'..'z' )
UniClass [letter] = ( U+0041..U+005a U+0061..U+007a )
```

There is no automatic relationship between byte and Unicode classes of the same name, however.

In passes that operate in a single code space (either Byte or Unicode), only one kind of class is relevant, and the keyword *Class* may be used instead of the specific form.

Defaults for unmapped characters

In single-codespace passes, any characters not explicitly matched by mapping rules will be output unchanged. However, this cannot happen in passes that cross the byte/Unicode barrier, as In this case default values are specified:

```
ByteDefault   '*'
UniDefault    U+003f
```

These statements within a *Byte_Unicode* pass would cause unmapped byte values to become question marks (U+003F) when mapping to Unicode, and unmapped Unicode values to become asterisks when mapping to bytes. The default defaults are 0x3f (*question mark*) on the byte side, and U+FFFD (*replacement character*) on the Unicode side, which will often be suitable. Note that if the pass specifies a mapping for each individual byte value from 0-255, the Unicode default will never be used.

Mapping rules

The actual mapping between a byte encoding and Unicode is expressed as a list of mapping rules. A mapping description actually contains two complete sets of mapping rules, one set that match characters in the byte encoding and generate Unicode, and the other that match Unicode characters and generate bytes. However, in most cases it is simplest to express both mappings at once, using bi-directional rules where either side of the rule can act as "match" with the other being "replacement".

The general form of a mapping rule is:

```
lhsSeq [ / lhsContext ] operator  rhsSeq [ / rhsContext ]
```

Here, *operator* indicates whether this rule is to be used only when mapping from the left-hand side to the right, from the right-hand side to the left, or (the most common case) in both directions:

<>	*bidirectional mapping rule*
>	*unidirectional LHS-to-RHS rule*
<	*unidirectional RHS-to-LHS rule*

The *lhsSeq* and *rhsSeq* parts of the rule are simple lists of character codes. These may be expressed as decimal numbers or as hexadecimal (prefixed with *0x*). In byte sequences, literal characters and sequences (quoted strings) are also allowed; these are enclosed in single or double quote marks. In Unicode sequences, characters may also be listed by their Unicode character names as found in http://www.unicode.org/Public/UNIDATA/UnicodeData.txt, with all non-alphanumeric characters in the names (primarily spaces) converted to underscores; thus, for example, *thai_character_ko_kai* (not case sensitive) may be used instead of *0x0E01* to make the mapping description file more self-documenting.

During the mapping operation, whichever of *lhsSeq* or *rhsSeq* corresponds to the input side of the rule can be considered a "match string", with the other being its "replacement". The *context* associated with the match string, if any, acts as a constraint on the application of the rule. (Any *context* associated with the replacement is irrelevant; it would be used when mapping in the other direction.)

Character class references may be used in the match and replacement sequences, although for clarity it may be better to list each individual character mapping. If a class is used on the replacement side of a rule, it must correspond to a class on the match side, and the resulting rules will map each character in the match class to the equivalent character in the replacement class. (The classes must contain the same number of characters.) Item tags (see below) may be used to associate the replacement class item with its corresponding match item; in the absence of such tags, items are matched by position within the match and replacement strings.

For contextually constrained mappings, the *lhsContext* and *rhsContext* parts of the mapping rule are used. These use a "slash … underscore" notation that may be familiar from other tools or from aspects of linguistics:

`/` *`preContextSeq`* `_` *`postContextSeq`*

The match and replace strings and the pre- and post-contexts may be simple sequences of character codes, or may be more complex expressions using the following "regular expression" elements:

`[`*`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* ($0 \le a \le b \le 15$)
`?`	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

A couple of notes on the use of regular expressions and context constraints:

- Repeat counts (or optionality) may be applied to parenthesized groups as well as to individual items.
- It is meaningless to specify context on the replacement side of a unidirectional rule; contextual constraints apply to the matching process on the input side of the conversion.
- The special '`#`' code is only meaningful as the first item in the pre-context or the last item in the post-context; in effect, there is an "end of text" pseudo-character before the first real character of input, and one after the last, which can only match this code.
- A negated item is still a "concrete" item that matches a real character in the input (or the "end of text" pseudo-character).
- No repeatable item can ever match more than 15 times; unlike standard regular expressions, the *star* and *plus* operators have a fixed upper bound. (In principle, a repeatable element within a repeatable group will permit a higher total number of repetitions.)

Rules are tested from the most to the least specific, where a longer rule (counting the length of context as well as the actual match string) is considered more specific than a shorter one. If there are two equally long rules that could match at a particular place in the input, the first one listed in the mapping description file will be used.

The maximum potential length of any pre-context (considering all repeat counts) in a pass, plus the maximum potential match string, plus the maximum potential post-context, must not exceed 255 characters. Similarly, the maximum output that can be generated from any rule is limited to 255 characters. These limits are not expected to pose a problem for the type of character mapping operation for which TECkit is designed. (It was never intended as a fully general-purpose string processing language or engine.)

Macros

The TECkit compiler supports a simple macro facility; this may be used to define symbols that act as "shorthand" for frequently-used fragments of a mapping description, such as character classes that are needed in multiple passes, or sequences used in the context of multiple rules.

A macro is defined with a line of the form:

```
Define name <arbitrary TECkit source>
```

Following such a line, anywhere *name* is found in the description, it is treated as representing the specified source text. For example, if a number of diacritics have alternate forms in the legacy encoding for use after narrow ("i-width") characters, the reverse mappings from Unicode will need contextual constraints that will be the same in each case. A suitable definition makes this easy to express and maintain:

```
Define PRE_CTX_IWIDTH   [iwidth] [lowerdia]* _
...
0xA1   <> combining_ring_above       / PRE_CTX_IWIDTH
0xA2   <> combining_down_tack_below  / PRE_CTX_IWIDTH
0xA3   <> combining_up_tack_below    / PRE_CTX_IWIDTH
```

This is particularly useful when the context is complex, perhaps involving several alternatives or multiple repeatable items; suitably descriptive macro names may also serve to make the mapping description more self-documenting.

Another use for macros is to provide symbolic names for byte values, or more convenient names for Unicode characters. This can help make mapping descriptions more readable, maintainable, and self-documenting.

Note that macros must be defined before they are used, including any use in the definition of other macros; thus, it is legitimate to say:

```
Define NUL   0x00
Define DEL   0x7F
Define ASCII NUL..DEL
ByteClass[asc] = (ASCII)
```

But with the definitions rearranged so that `NUL` and `DEL` are not defined when they are used in the definition of `ASCII` (even if they are defined subsequently), the result will be a compile-time error:

```
Define ASCII NUL..DEL
Define NUL   0x00
Define DEL   0x7F
ByteClass[asc] = (ASCII)
```

This will generate an error on the ByteClass line, because the identifiers `NUL` and `DEL` found in the expansion of `ASCII` will be considered undefined.

Examples

Windows code page 1252

A particularly simple mapping to describe is Windows code page 1252. This is an encoding that has a simple, one-to-one mapping to and from Unicode:

```
EncodingName     'WINDOWS-1252'
DescriptiveName  'Windows code page 1252 (Latin-1)'

ByteDefault      '?'
UniDefault       replacement_character

ByteClass [ascii] = ( 0 .. 127 )
UniClass [ascii] = ( U+0000 .. U+007f )
```

```
ByteClass [latin1] = ( 0xa0 .. 0xff )
UniClass [latin1] = ( U+00a0 .. U+00ff )

[ascii]   <> [ascii]
[latin1]  <> [latin1]

0x80   <> euro_sign
;0x81  undefined
0x82   <> single_low_9_quotation_mark
; ... mappings for 0x83 to 0x9d omitted for brevity
0x9e   <> latin_small_letter_z_with_caron
0x9f   <> latin_capital_letter_y_with_diaeresis
```

SIL Greek

A more complex example is mapping SIL Greek to Unicode. A complete SILGreek.map file is included with TECkit as a sample of a TECkit description; some of the more interesting fragments are shown here.

In order to deal with both Basic and Display versions of SIL Greek, the mapping is implemented in several passes. First, a *Byte* pass maps the precomposed forms to equivalent sequences, thus converting Display-encoded text to Basic encoding; however, it maintains the Display distinction between final and non-final *sigma*, and indeed maps the Basic *sigma* code (which is non-final in Display text) to the final form where appropriate:

```
Pass(Byte)

; First we map precomposed "display" forms to their equivalent "basic"
; sequences while still in the Byte (SIL Greek legacy encoding) world,
; except that we maintain the final/non-final sigma distinction

Class  [LTR] = ( 'a'..'u' 'w'..'z' 'A'..'U' 'W'..'Z' '`^_@"' "'" \
                 128..149 152..159 161..163 165..171 173..181 184..255)

; make sigma into final form if not followed by a letter
's' / _ ^[LTR]   >  'v'

; This is copied directly from "GRCO-BA.CCT - Greek Composite to Basic
; conversion" and then the unidirectional '>' operators changed to
; bidirectional '<>', and non-ASCII characters replaced with hex codes
; (for clarity)

0xCF            <>  'Hr'
0xBF            <>  'hr'

0xAD 'A'        <>  'HA' "'"
0xAE 'A'        <>  'HA`'
0xAF 'A'        <>  'HA^'
0xA9 'A'        <>  'hA' "'"
0xAA 'A'        <>  'hA`'
0xAB 'A'        <>  'hA^'
...etc...
```

Next, the main *Byte_Unicode* pass maps the SIL Basic Greek codes to their Unicode equivalents. This includes handling characters that are represented in SIL Greek as sequences using the '|' *modifier* code, but map to single Unicode characters. There are also some cases where more than one Unicode character is mapped to the same SIL Greek code:

```
Pass(Byte_Unicode)

ByteDefault   183    ; 183 is "bullet" in the SIL Greek Display encoding
UniDefault    replacement_character

; there are separate namespaces for Byte and Unicode classes,
```

```
; allowing us to use the same name for classes with corresponding content

ByteClass [CTL] = (   0x00 .. 0x1f     0x7f )
UniClass  [CTL] = ( U+0000 .. U+001f U+007f )
[CTL]  <> [CTL]

' '    <> space
'!'    <> exclamation_mark
'"'    <> combining_diaeresis
'#'    <> no_break_space
'$'    <> left_pointing_double_angle_quotation_mark
'%'    <> right_pointing_double_angle_quotation_mark
'&'    <> ampersand
"'"    <> combining_acute_accent

... ...

';'    <  greek_ano_teleia ; greek semicolon
';'    <> middle_dot       ; canonical decomposition of greek semicolon

... ...

'?'    <  greek_question_mark
'?'    <> semicolon        ; canonical decomposition of grk question mark

... ...

'@'    <> modifier_letter_apostrophe
'A'    <> greek_capital_letter_alpha
'B'    <> greek_capital_letter_beta
'C'    <> greek_capital_letter_chi

... ...

'|b'   <> greek_beta_symbol             ; curly beta
'|f'   <> greek_small_letter_digamma    ; digamma
'|G'   <> greek_letter_digamma          ; Digamma
```

Next, we deal with the fact that SIL Basic Greek and Unicode handle the Greek breathing marks differently. In SIL Basic Greek, these are coded before the vowels to which they apply, while in Unicode they follow the vowel. The situation is complicated by the fact that certain vowel pairs are actually diphthongs, in which case the breathing has to “jump over” both vowel characters. Moreover, if the second vowel of what would be a valid diphthong pair carries a dieresis, this “breaks” the diphthong, and the breathing should be located after the first vowel instead.

This is handled with a *Unicode* pass that matches breathing-vowel sequences in SIL Greek order and rearranges them into the proper Unicode order:

```
Pass(Unicode)

; In Unicode space, reorder breathing/vowel sequences from SIL Basic to
Unicode order

Class [BR]    =   ( combining_comma_above combining_reversed_comma_above )
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 / _ combining_diaeresis   \
                          <> @v1 @b @v2 / _ combining_diaeresis
[BR]=b [aeo]=v1 [iu]=v2    <> @v1 @v2 @b
[BR]=b [j]=v1 [u]=v2 / _ combining_diaeresis   \
```

```
                            <> @v1 @b @v2 / _ combining_diaeresis
[BR]=b [j]=v1 [u]=v2        <> @v1 @v2 @b
[BR]=b [u]=v1 [i]=v2 / _ combining_diaeresis   \
                            <> @v1 @b @v2 / _ combining_diaeresis
[BR]=b [u]=v1 [i]=v2        <> @v1 @v2 @b
[BR]=b [vowelrho]=v         <> @v @b
```

In these rules, the breathing and vowels are tagged with *b* and *v* (or *v1*, *v2*) respectively, and the replacement consists of the tagged items in a revised order. (Note that while these rules are written and described primarily as if they are mapping from left to right, they are in fact bidirectional rules that are used in both byte/Unicode and Unicode/byte processing.)

Finally, a second *Unicode* pass replaces sequences of letters and diacritics with precomposed forms where possible. This pass was created by extracting the relevant canonical mappings from the Unicode character database:

```
Pass(Unicode)

; Now map to/from the available precomposed letter/diacritic combinations
; in Unicode (these mappings are derived from the decomposition field in
; UnicodeData.txt, recursively applying decomposition to the components
; where relevant). It would be more efficient to do everything in a
; single Byte_Unicode pass, and for Greek this is not unreasonably
; complex, but these separate passes serve to illustrate (and test) the
; "pipeline" architecture.

U+0313              <  U+0343 ; combining greek koronis
U+0308 U+0301       <> U+0344 ; combining greek dialytika tonos
U+02B9              <  U+0374 ; greek numeral sign
U+003B              <  U+037E ; greek question mark
U+00A8 U+0301       <> U+0385 ; greek dialytika tonos
U+0391 U+0301       <> U+0386 ; greek capital letter alpha with tonos
U+00B7              <  U+0387 ; greek ano teleia
U+0395 U+0301       <> U+0388 ; greek capital letter epsilon with tonos
U+0397 U+0301       <> U+0389 ; greek capital letter eta with tonos
U+0399 U+0301       <> U+038A ; greek capital letter iota with tonos
U+039F U+0301       <> U+038C ; greek capital letter omicron with tonos
U+03A5 U+0301       <> U+038E ; greek capital letter upsilon with tonos
U+03A9 U+0301       <> U+038F ; greek capital letter omega with tonos
U+03B9 U+0308 U+0301 <> U+0390 ; ...etc...
```

Note that such a mapping is always described primarily in the byte⇶Unicode direction, beginning with any *Byte* passes, then the *Byte_Unicode* pass, and finally any *Unicode* passes. When mapping from Unicode back to the legacy byte encoding, the rules in each pass map from RHS to LHS, and in addition the passes apply in the reverse order.

Byte-only and Unicode-only mappings

The TECkit system, while targeted primarily at byte/Unicode conversion, can also be applied to other text mapping operations. A mapping description need not contain a *Byte_Unicode* pass at all. If it contains only *Byte* passes, then both input and output are bytes; if it contains only *Unicode* passes, both input and output are Unicode data.

For example, a slightly modified version of the first pass of the SIL Greek example mapping could serve as a conversion between SIL Basic Greek and SIL Display Greek. (The only change needed would be to the processing of final *sigma*.) Another application might be a mapping to transliterate between Greek and Roman scripts in Unicode.