Unicode Implementation Issues*

Shmuel (Seymour J.) Metz

October 31, 2023

Abstract

Classic Rexx supports character code pages in which every character can be represented in a single octet, and treats only ASCII letters as alphabetic. Unicode is nominally a 32-bit character set, with code points restricted to 17 16-bit planes, i.e., code points U+00 through U+10FFFF. This document describes basic issues and solutions for extending Rexx to support Unicode.

Contents

*This is a working document for the Rexx Language Association (RexxLA). There will be one or more separate requirements documents. It is written in LaTeX 2023-06-01 and was rendered October 31, 2023 using luahbtex 1.17.0 (lualatex).

1 Background

The Architecture Review board is investigating areas in which Rexx may be in need of enhancements, some of which are listed in [ARB:Issues]. Many modern languages support Unicode encoded as UTF-8 ([RFC 3629]), and the IETF, in [RFC 5198], has mandated the use of UTF-8 with NFC in new protocols. Rexx will need to support UTF-8 if it is not to become a backwater, and that support should not break existing code. Elaboration of this issue may be added to [ARB:Issues] issues I.2 and I.6.

2 Scope

This document only describes issues related to issues I.1 and I.2, expanded alphabet and multi-octet representations of characters in Unicode issues I.2 and I.6; it does not address such issues as bidirectional text, nor does it address legacy DBCS support. It presents suggested solutions for requirements that are presented in other documents. Also, most of the details are under active discussion, so everything here is provisional. Information on Unicode may be found at *Unicode Standard* and *Unicode Frequently Asked Questions*.

This document concentrates on classic Rexx, ANSI Rexx and ooRexx; the implementations most relevant are ooRexx and Regina. However, cREXX and NetRexx must be taken into account in order to avoid unnecessary differences.

3 Nomenclature

The definitions given in [Unicode Glossary] and [Unicode] take precedence over those given here. Quoted text is taken from those sources. Except for definitions taken from official IETF and Unicode documents, the nomenclature used here is illustrative rather than normative; language design teams will formally define, e.g., method names, encoding of parameters. Most of the sections assume that there will be destinct string types for extended grapheme clusters (issues I.20 and I.25), Unicode code points (issues I.20 and I.25), legacy code pages (issues I.15, I.28 and I.30) and raw octets (issues I.17, I.20 and I.29). However, there has been discussion of including metadata, in which case come of the classes might be merged.

Some of the nomenclature used in this document, including the illustrative class names **BYTES**, **CODEPOINTS** and **TEXT** is taken from the RXU preprocessor of The Unicode Tools of Rexx (TUTOR). It is planned to encorporate more vocabulary from RXU in subsequent revisions.

The word **text** has its generic meaning unless given in upper case.

An underscored term in a list of options is the default. Terms separated by the Or symbol (|) are mutually exclussive alternatives. Terms in brackets are optional.

The bold character **´** in examples refers to U+0301, COMBINING ACUTE ACCENT. The characters **É** and **é** in examples refer to the composed characters U+C9, LATIN CAPITAL LETTER E WITH ACUTE and U+E9, LATIN SMALL LETTER E WITH ACUTE, not to the decomposed clusters <U+0045, U+0301> and <U+0065, U+0301>.

Some code samples use ooRexx notation. Those samples are illustrative rather than normative.

References to requirments in section 5 ARB GitHub Issues are by number.

4 Statement of problems

4.1 Character width

In Rexx, everything is a string; there are no classes, declarations or types. (issue I.15)

While the ANSI standard does not mandate any particular character width for **Config_C2B()**, most if not all implementations use a width of 8, and the length of **c2x(***foo***)** is twice the length of *foo*. (issues I.23 and I.24)

A large body of existing code operates on binary data from external files, or accessed from memory via the **storage()** BIF, under the assumption that, e.g., **c2x()**, left(), **right()**, **substr()**, operate on octets(]crefissue:ROS,issue:mO. A UTF-8 introducer is treated the same as any other value.

A large body of existing code operates on text under the assumption that those facilities operate on characters. (issue I.3)

The two categories overlap.

There is no conflict as long as each character is contained within a single octet. However, Unicode has code points beyond U+FF, and UTF-8 encoding of non-ASCII Unicode characters will require more than a single octet even for code points less than U+100 if they are beyond U+7F.

For example, the Unicode string "Café" has 4 code points, 4 grapheme clusters and 5 octets in UTF-8 encoding while "Cafe´" has 5 code points, 4 grapheme clusters and 6 octets in UTF-8 encoding (issue I.20), yet many text processing applications need to treat them as equivalent.

4.2 Case conversion

Rexx has built-in case conversion, but it is based on ASCII and can't even convert the accented letters found in, e.g., ISO-8859-1 (Latin-1), ISO-8859-15 (Latin-9), much less all those found in Unicode.

Listing 1: Non-ASCII upper casing in classic Rexx

```
foo = 'René'
parse upper var foo bar
```

sets bar to RENé rather than to RENÉ.

4.3 xrange()

In current implementations the range of characters is extremely small, and thus the **xrange()** only returns short strings. (issue I.31) With Unicode the range expands to U+00 through U+10FFFF. That is far too large to allow as either an explict or implicit range in **xrange()**.

4.4 Numeric string conversion to characters

The definitions of binary and hexadecimal literals, and of the b2c() and x2c() functions, in [**ANSI:J18**], depend on Config_B2C. While the definition"Translate Binary to a coded string." of Config_B2C in ANSI X3J18-1996 is rather vague, it is adequate when the character set is restricted to an 8-bit character set. The definition must be replaced by one that is unamgiguous when applied to Unicode. (issue I.9) There are several obvious options.

1. Treat every 32 bits as a Unicode scalar
2. Treat every 8 bits as part of a UTF-8 sequence
3. Treat every 8 bits as a character in a legacy code page.

Each of these options has disadvantages.
The converse issue exists for Config_C2B.

4.5 Identifiers, labels and procedure names

The current definition restricts symbols to ASCII characters, which excludes letters used in many languages. If the rules are extended, in accordance with [UAX 31]. to include Unicode letters and digits beyond ASCII, the rules for equality of symbols must be addressed. Are composed and decomposed strings identical? Are base characters identical to their compatibility alterrnates? Are subscript and superscript digits distinct from their ASCII counterparts?

When a numeric value is used as a label or procedure name, a precise definition of the semantics is needed. This is essentially the same issue as section 4.4 (Numeric string conversion to characters) above.

When dealing with Unicode, different code sequences may have identical rendering due to the existence of both fully composed and combining code points and the exixtence of compatibility code points. Rexx needs a way to test two Unicode strings for equivalence. (issues I.14, I.21 and I.22)

4.6 PARSE

The PARSE tempates currently support spliiting text based on a restricted set of whitespace characters. A decision must be made whether to only support space and tab as whitespace for PARSE, or to use one of the whitespace definitions in [UAX 31].

4.7 Status of legacy code pages

There has been some discussion claiming a need to continue supporting legacy code pages. (issues I.16, I.28 and I.30)

4.8 Unassigned and invalid code points

There has been some discussion of the need to detect unassigned[1] code points. (issues I.4 and I.5)

4.9 New notation

Unicode introduces new notation, which Rexx currently doesn't support, e.g., U+xxxx, <codepoint, ...>.

4.10 Tools

There are tools available that manipilate Rexx source code in various ways, e.g., code completion, cross referencing, navigation, prettyprinting, syntax highligting. These tools may not handle new syntax added in support of Unicode. Examples include

- The IBM Z®Open Editor
  A Rexx Language Support plugin for Visual Studio Code
- The LaTeX package **listings**
- The LaTeX package **minted**
- The Python package **Pygments**

5 ARB GitHub Issues

This section lists all issues in [ARB:Issues], whether or not they relate to Unicode.

I.1 Expanded Alphabets

I.2 Multi-Byte Representations

I.3 Unicode Strings

I.4 Unassigned & Invalid Code Points

I.5 Error Detection

I.6 UTF-8 Support

I.7 Identifiers & Allowable Formats

I.8 New Notation (e.g., U+xxxx)

I.9 Constants for Unicode Characters

[1] Unsigned is not the same as noncharacter.

I.10 Case Folding

I.11 Caseless Comparisons

I.12 Case Conversion

I.13 Coercions

I.14 Strict / Non-strict operators

I.15 Legacy Strings

I.16 Legacy Code Pages Support

I.17 Raw Octet Strings

I.18 I/O Methods

I.19 IO Random Access

I.20 New Text Functions

I.21 NFC & NFD Functions

I.22 NFKC & NFKD Functions

I.23 c2x() Function

I.24 Width Parameters for c2x()

I.25 c2u() Function

I.26 makeCodePointString Function

I.27 makeGraphemClusterString Function

I.28 makeLegacyString Function

I.29 makeOctets Function

I.30 iconv Function

I.31 xrange() Function

I.32 PARSE Instruction

6 Solutions

Most of this section assumes that there will be three distict string types, (issues I.26, I.27 and I.29), and that some methods will not exist in all three types, or will behave differently. The type names used here are placeholders, and will be replaced once there is consensus on what to call them.

6.1 OPTIONS statement or directive

Define an option on the OPTIONS statement, or on a similar directive, to specify either raw octets or Unicode characters. This breaks programs that operate on both binary data and text.

OPTIONS might also specify a source encoding parameter, overriding any code page in an environment variable or file metadata.

SOURCECP= Encoding of the Rexx source code

LITERALS= Type of string constant with no suffix

BYTES	Raw octets
CODEPOINTS	Unicode by code point
LEGACY	Legacy 8-bit characters
TEXT	Unicode by extended grapheme cluster

6.1.1 RXU OPTIONS values

The RXU preprocessor accepts these[RXU: OPTIONS] values on the OPTIONS statement:

DEFAULTSTRING *default* where *default* determines the type of a string literal with no suffix:

BYTES	Y
CODEPOINTS	P
NONE	
TEXT	T

COERCIONS *ucoercion* where *ucoercion* determines the result of an operation on strings of mixed string types:

DEMOTE	Type of lowest operand
LEFT	Type of left operand
NONE	Raise SYNTAX
PROMOTE	Type of highest operand
RIGHT	Type of right operand

6.2 Classes and types

Allow Rexx variables to contain three distinct types of data: raw octets, Unicode code points and text, (issues I.26, I.27 and I.29),

Define **storage()** as returning raw octets, and provide conversion functions ("casts"). There is an ongoing discussion as to whether all three are needed.

There should be no implicit conversion between **.BYTES** and Unicode strings.

There should be implicit conversion between Unicode types, but it need not be reversible, i.e., a conversion from .CODEPOINTS to .TEXT and back need not produce the same code points as the original

If support for legacy code pages is needed, a fourth string type could be defined. Strings of this type could include a code page attribute.

For ooRexx a single class with distinguishing attributes could be used, or a separate class for each type.

RXU uses BYTES for both octet strings and legacy strings.

6.3 Width parameters for c2x()

Add a width parameter to **c2x()** for Unicode code point string (issue I.24) (raw octet strings may require input and output widths for UCS-2 AND UTF-16 data, and the utility of **c2x()** for grapheme clusters needs more analysis), and raise conditions with distinct error codes if any code point is out of range or if an invalid UTF-8 sequence is detected. A case could be made for using either the bit size or the digit size as the width.

Assuming that the width is in in terms of octets

Listing 2: c2x() with width parameter

```
foo = 'René'
bar = foo~UTF-8
/* assumes that width parameter is in octets */
say foo~c2x(2) /* Displays 00520065006E00E9  */
say bar~c2x    /* Displays 52656EC3A9        */
```

There is an ongoing discussion as to whether an when to allow implicit coercions of type in, e.g.,

Listing 3: Implicit coercion

```
foo = .BYTES~new
foo = 'René'
...
parse var foo ASCII 'é' .
say ASCII                          /* Displays Ren */
```

If there is no raw octet string added to the language, then **c2x()** might behave differently depending on whether an explicit width is provided.

6.4 RXU c2x() and c2u()

In RXU, **c2x()** for Unicode code point strings returns the hexadecimal values of UTF-8 sequences rather than the hexadecimal values of unicode scalars. The second parameter of **c2u()** controls the output format, and **c2x(foo,UTF-32)** returns space separated Unicode scalars.

6.5 Constants

Other languages allow specifying Unicode characters using either the hexadecimal value of the code point or the assigned name of the code point, (issue I.9) e.g., U+E9 (issue I.8) might be coded as \u{E9} or \u[LATIN SMALL LETTER E WITH ACUTE]. The syntax used for such constants should be consisten across Rexx variants, including rules for optional spaces between (hex) digits, and should take into account the recommendations in [RFC 5137]. Implementations should use the machine readable data bases published by the Unicode Consortium in order to ease migration to new versions of Unicode.

The form 'U+digits U+digits ...'U is clearer, but may break code that abuts the variable U with a string literal. The form 'U+digits U+digits ...'X is acceptable. There should be a discussion of syntax for named Unicode constants, e.g., '[COMBINING ACUTE ACCENT]'U is equivalent to 'U+0301'U. There has been some discussion of using the form '...'T and of escape conventions used in other languages.

Binary and hexadecimal literals are of type **.BYTES** and cannot be implicitly coerced to Unicode strings. Other string literals are legacy or Unicode and cannot be implictly coerced to **.BYTES**.

An alternative is to add a new notation using the ASCII ` as a framing character. This has the potential issue that it may be difficult to visually distinguish ` from '.

Another alternative is to use literals of the forms '...'type:codepage, '...'U:codepoints and '...'U:clusters. However, that makes it incompatinle with ooRexx.

The forms '...'{modifier} and '...'type{modifier} have no obvious conflict with existing syntax and has no abutment issues.

The following table is intended as a discussion point and should be updated whenever a consencuss is reached.

Example	Type	Semantics
'René'	CODEPOINTS TEXT	Unicode text by code point Unicode text by grapheme cluster (TBD)
'U+00E9'R	CODEPOINTS	Unicode text by code point
'U+00E9'CODEPOINTS	CODEPOINTS	Unicode text by code point
'U+00E9'TEXT	TEXT	Unicode text by grapheme cluster
'René'{CODEPOINTS}	CODEPOINTS	Unicode text by code point
'René'{TEXT}	TEXT	Unicode text by grapheme cluster
'René'{ISO8859-1}	Legacy	ISO 8859-1 legacy text
`René`	CODEPOINTS TEXT	Unicode text by code point Unicode text by grapheme cluster (TBD)
`René`C	CODEPOINTS	Unicode text by code point
`René`T	TEXT	Unicode text by grapheme cluster

An issue that must be resolved is whether string literals should have binary fidelity or visual fidelity to the target code page, i.e., should the quoted source characters be given in the source code page or in the target code page. E.g., should the word ”foo” in ISO 8859-1 and in EBCDIC be coded as 'foo'ISO8859-1 and 'foo'EBCDIC, requiring that Rexx translate the text, or should the second be entered with the EBCDIC code points 86, 96, 96, making it harder to read and edit?

Another issue is whether numeric constants used as labels and call targets should be treated as strings of legacy code points, UTF-8 sequences or as strings of Unicode scalars.

6.5.1 RXU string types

RXU has five types of string literals[RXU: New Types]:

’string’ The type is determined by OPTIONS.

’string’P Unicode by code point

’string’T Unicode by EGC

’space separated tokens’U Unicode code points by alias, name or hexadecimal value; the string is BYTES rather than CODEPOINTS or TEXT.

’string’Y Legacy bYte string,can only be promoted if valid UTF-8 sequences

6.6 Error detection

The Rexx standard should specify ehat to do when a string being transformed contains invalid code points or octet seqiences. (issue I.5) The condition names given below are illustrative and not normative. Unicode BIF/BIMs should detect invalid input and signal the following conditions, with unique error codes. It is TBD whether some or all of these should be folded into SYNTAX.

INVALIDCODEPOINT But carefully read 3.2 Conformance Requirements, Code Points Unassigned to Abstract Characters, p. 77, in [Unicode].

INVALIDUTF An invalid UTF-8 octet sequence or invalid use of a UTF-16 surrogate.

NOENCODING The operation requested requires a code page name.

NOTEXT An operation was requested that is not valid for a binary file.

RANGE A code point or other numeric entioty excees the permitted range. This might occur, e.g., when an application requuires that characters be limited to the BMP.

6.7 I/O

Stream input/output will need additional support for Unicode. issue I.18 The **command** and **open** methods of the stream classes should support a **CODEPAGE** option; attempting to set a codepage for a binary file should raise a **NOTTEXT** condition with a distinct error code. There should be an option or suboption controling whether to use Unicode if an apparent BOM is detected.

The default should probably be UTF-8 or ISO 8859-1 with switching to Unicode if a BOM is detected.

Similarly, there should be an option to distinguish between reading individual code points and reading EGC; this is needed so that the **charin()** BIM can correctly interpret the character count for Unicode input.

Handling seek for variable-width encodings and EGCs is problematical. issue I.19 There are several viable approaches, each with disadvantages.

- Use octet count, as at present.
  This risks a program doing a seek within a UTF-8 sequence, within a UCS-2/UCS-4 character, within a UTF-16 surrogate pair, or, for EGC access, within an EGC. Strategies to ameliorate this include

  UTF-8 Read the octet at the specified offset and verify that either the high bit is 0 or the two high order bits are 1.

  UCS-2 Verify that the offset is a multiple of 2.

  UTF-16 Verify that the offset is a multiple of 2. Read two octets at the specified offset and verify that is not a low surrogate (U+DC00-U+DFFF),

  UCS-4 Verify that the offset is a multiple of 2. Read four bytes and verify that they do not exceed U+10FFFF.

- Use character or EGC count
  This is expensive

- Define a cursor type.
  This violates "Everything is a string" in classic Rexx, but is reasonable for ooRexx.

Handling update in the middle of a file with variable-width encodings or EGC access has similar issues, Absent an acceptable way to deal with it, prohibit update in the middle of a file for anything but binary and legacy characters.

A discussion is needed on whether and when to create or discard byte order marks.

6.7.1 RXU I/O

in RXU, stream I/O[RXU: STREAM] defaults to legacy (BYTES).

RXU adds an **ENCODING** option to **STREAM(***streamid***,OPEN** *options***)**. When this option is used the stream is Unicode enabled, and some stream functions are not allowed. **ENCODING** must be followed by the name of then encoding, which for Unicode may in turn be followed by *error* and *target* options:

- **REPLACE**
  Replace invalid characters with the Unicode Replacement Character (U+FFFd).
- **SYNTAX**
  Raise SYNTAX for invalid characters.
- **CODEPOINTS**
  Units of codepoints.
- **TEXT**
  Units of extended grapheme clusters.

RXU adds options to **STREAM(***streamid*,**QUERY***options***)**

- **Query Encoding Name**
- **Query Encoding Target**
  CODEPOINTS or TEXT
- **Query Encoding Error**
  REPLACE or SYNTAX
- **Query Encoding LastError**
- **Query Encoding**
  Everything but LastError

6.8 Case folding

Case folding of symbols and caseless comparisons will use the *Unicode Character Database*ENCODING. issues I.10 to I.12) The rules for case folding should take into account mathematical usage.

- U+1D400 through U+1D7FF are semantically distinct from other letters.
- Superscripts and subscripts have semantic significance.

6.9 Raw octet strings

Many exist Rexx programs deal with binary data, and Rexx has conversion functions to deal with those data, e.g., c2x, x2c. (issue I.17) The new version of Rexx will need methods to continue dealing with binary data. The methods should include those of the ooRexx .string class, except that

- The unit of operation is the uninterpreted octet.
- The **makeString** method requires an encoding parameter.
- Parameters are raw octet strings.
- There are no caseless methods.

Additional conversion functions will be needed (issues I.20, I.28 and I.29).

6.10 Legacy strings

If there is a .Legacy string type then the methods should have the same semantics as the existing methods for the .string class. If the new standard has a code page attribute then there should be an access method for it and the init or new method should allow specifying the code page.

The following Additional functions and methods should be defined:

iconv	Convert from one code page to another and return a legacy string. (issue I.30)
makeCodePageString	Return a Unicode string in which individual code points can be accessed. (issue I.26)
makeGraphemClusterString	Return a Unicode string in which only complete grapheme clusters can be accessed. (issue I.27)

6.11 Unicode strings

There should be subtypes depending on whether the unit of operation is the Unicode code point or the grapheme cluster. (issue I.3) The methods should be those of the ooRexx .string class except:

- there should be a BIF for each in order to allow use in classic Rexx.
- Discuss how and whether to introduce equivalent operators.
- discuss whether = should ignore all leading and trailing whitespace or only leading and trailing blanks. Discuss need for other equality tests.
- == tests for absolute equality for CODEPOINTS but EGC equivalence for TEXT. Discuss which of the Unicode equality tests to use and, if more than one, which is primary.
- The caseless comparisons will use the [UCD]. (issue I.11) There will be variants to preserve or remove accents[2].
- There should be no **bit...** methods.
- Add an optional third parameter to the **upper** and **lower** methods to control whether to translate mon-ASCII characters and whether to strip accents.
- The following Additional functions and methods should be defined:

c2u	Return code points in Unicode notation. Possible options include – U+xxxx or bare hexadecimal – frame in <> and separate with commas – indicate EGC boundaries – indicate properties
makeCodePageString	Return a Unicode string in which individual code points can be accessed (issue I.26)
makeGraphemClusterString	Return a Unicode string in which only complete grapheme clusters can be accessed (issue I.27)
makeLegacyString	Return a legacy string for the specified code page in which individual octets can be accessed (issues I.16 and I.28)
makeOctets	Return a raw octet string using a specified encoding.[3] (issue I.29)
NFC	Return a string normalized with Normalization Form C (issue I.21)
NFD	Return a string normalized with Normalization Form D (issue I.21)

There should be a description of how to handle conversion from Unicode code points and grapheme clusters that do not exist in the target code page. (issues I.5, I.15 and I.30)

- Replace with U+1A (SUB) or other specified character
- Raise SYNTAX with a unique error code
- Raise a new condition name with a unique error code

6.12 Functions and methods

The following table summarizes some of the methods that differ among string types. With the exceptions of **[]**, **makeArray** and **makeString**, there is an operator or polymorphic BIF for each BIM listed.

[2] Is Enye (ñ) considered an accented letter?
[3] default to UTF-8 or raise NOENCODING if no encoding specified?

Method	.Legacy	.BYTES	.CODEPOINTS	TEXT
[]	by octet	by octet	by code point	by cluster
&	by octet	**n/a**	by code point	by cluster
|	by octet	**n/a**	by code point	by cluster
&&	by octet	**n/a**	by code point	by cluster
=	by octet	by octet	by code point	by cluster
==	by octet	by octet	by code point	by cluster
¬=	by octet	by octet	by code point	by cluster
><	by octet	by octet	by code point	by cluster
<>	by octet	by octet	by code point	by cluster
<	by octet	by octet	by code point	by cluster
<=	by octet	by octet	by code point	by cluster
>	by octet	by octet	by code point	by cluster
¬>	by octet	by octet	by code point	by cluster
abbrev	by octet	by octet	by code point	by cluster
bitand	**n/a**	by octet	**n/a**	**n/a**
bitor	**n/a**	by octet	**n/a**	**n/a**
bitxor	**n/a**	by octet	**n/a**	**n/a**
c2b	8 bits	default 8 bits	Explicit width	**n/a**
c2x	8 bits	default 8 bits	Explicit width	**n/a**
center	by octet	**n/a**	by code point	by cluster
change	by octet	by octet	by code point	by cluster
find	by octet	by octet	by code point	by cluster
index	by octet	by octet	by code point	by cluster
left	by octet	by octet	by code point	by cluster
makeArray	by octet	by octet	by code point	by cluster
makeString	by octet	by octet	by code point	by cluster
pos	by octet	by octet	by code point	by cluster
right	by octet	by octet	by code point	by cluster
strip	by octet	by octet	by code point	by cluster
substr	by octet	by octet	by code point	by cluster
verify	by octet	by octet	by code point	by cluster

The backslash ("\") may be used in place of the Logical Not ("¬").

In addition, the makeString and makeString methods should allow caseless option parameters to control the class and attributes of the returned array elements or string, including:

Clusters	Unicode text with extended grapeme clusters as the abstract units.
Codepoints	Unicode text with Unicode scalars as the abstract units.
cp=legacy code page	Legacy or octet string with specified encoding.
cp=UTF-8	Legacy or octet string with UTF-8 encoding; functions like center and left will give unexpected results.
Legacy	Legacy string with default encoding unless cp= is also specified.
raw	Raw octet string; cp= must be specified.

6.12.1 RXU functions and methods

RXU adds the **DECODE** and **ENCODE** methods in the Unicode class. These provide an easy way to implement the proposed **iconv** function.

RXU adds these[RXU: New built-in functions] additional built-in functions:

- **BYTES**
- **C2U**
- **CODEPOINTS**
- **GRAPHEMES**

- **N2P**
- **P2N**
- **STRINGTYPE**
- **TEXT**
- **UNICODE(***string*,*function***)**
  Performs the named function on *string*:

  isNFD Returns 1 if string is in normalized form D

  toLowercase Returns upper case translatio.

  toUpercase Returns lower case translation

  toNFD Normalizes to form D (decomposed)
- **UNICODE(***codepoint*,**PROPERTY**,*property***)**
  Retrieves the value of the named property
- **UTF8**
  Validates a UTF-8 string and optionally converts it to a different Unicode transform.

6.13 Coercions

In addition to explicit casts via BIF/BIM, the following promotions from code-point strings to grapheme-cluster strings will be automatic issue I.13)

- Concatenation of code-point strings with grapheme-cluster strings. will be grapheme-cluster strings.
- Code point search arguments for grapheme-cluster strings. will be coerced to grapheme-cluster strings.
- Legacy strings tagged to a code page, in a context that requires Unicode, will be converted to the appropriate Unicode string type.

There will be no automatic promotion for octet strings, Unicode to legacy nor for legacy strings without code page tagging.

7 Glossary

The definitions given in [Unicode Glossary] and [Unicode] take precedence over those given here. Quoted text is taken from those sources, or from IETF documents. Except for definitions taken from official IETF and Unicode documents, the nomenclature used here is illustrative rather than normative; language design teams will formally define, e.g., method names, encoding of parameters.

.BYTES	The string class for uninterprted octet strings.
.CODEPOINTS	The string class for strings of Unicode code points.
.Legacy	The string class for strings in legacy code pages. May be merged with .BYTES.
.TEXT	The string class for strings of Unicod extended grapheme clusters.
ARB	RexxLA Architecture Review Board
Bidi	”Abbreviation of bidirectional, in reference to mixed left-to-right and right-to-left text.”
BIF	Built In Function
BIM	Built In Method
Block	”A grouping of characters within the Unicode encoding space used for organizing code charts. Each block is a uniquely named, continuous, non-overlapping range of code points, containing a multiple of 16 code points, and starting at a location that is a multiple of 16. A block may contain unassigned code points, which are reserved.”

BMP Basic Multilingual PlaneindexBMP!Basic Multilingual Plane!definition
The first 64 Ki code points of Unicode, from U+0000 to U+FFFF.

BOM Byte Order Mark: U+FEFF, ZERO WIDTH NON-BREAKING SPACE (ZWNBSP)
Used as the first character to indicate byte order for UCS-2, UTF-16 and UCS-4;
Optional as the first character for UTF-8.

encoded character The smallest constituant of a Unicode string. "The Unicode Standard does not define what is and is not a text element in different processes; instead, it defines elements called encoded characters. An encoded character is represented by a number from 0 to $10FFFF_{16}$ called a code point."

EGC Extended grapheme cluster

GCGID "Acronym for Graphic Character Global Identifier. These are listed in the IBM document Character Data Representation Architecture, Level 1, Registry SC09-1391." See `https://www.ibm.com/downloads/cas/GO1BQVRV`.

grapheme cluster "A grapheme cluster consists of a base character followed by any number of continuing characters, where acontinuing character may include any nonspacing combining mark, certain spacing combining marks, or a join control." [UAX 29] defines two types of grapheme clusters; "An extended grapheme cluster is the same as a legacy grapheme cluster, with the addition of some other characters. The continuing characters are extended to include all spacing combining marks, such as the spacing (but dependent) vowel signs in Indic scripts."

high surrogate A code point in the range U+D800-U+DBFF, used as the first half of a surrogate pair.

IETF Internet Engineering Task Force

introducer The first octet in the UTF-8 encoding of a Unicode character beyond U+7F

low surrogate A code point in the range U+DC00-U+DFFF, used as the second half of a surrogate pair.

NFC Unicode "Normalization Form C (NFC). A normalization form that erases any canonical differences, and generally produces a composed result. For example, a + umlaut is converted to ä in this form. This form most closely matches legacy usage. The formal definition is D120 in Section 3.11, Normalization Forms."
the normalization endorsed by the IETF

NFD Unicode "Normalization Form D (NFD). A normalization form that erases any canonical differences, and produces a decomposed result. For example, ä is converted to a + umlaut in this form. This form is most often used in internal processing, such as in collation. The formal definition is D118 in Section 3.11, Normalization Forms."

NFKC Unicode "Normalization Form KC (NFKC). A normalization form that erases both canonical and compatibility differences, and generally produces a composed result: for example, the single dž character is converted to d + ž in this form. This form is commonly used in matching. The formal definition is D121 in Section 3.11, Normalization Forms."

NFKD Unicode "Normalization Form KD (NFKD). A normalization form that erases both canonical and compatibility differences, and produces a decomposed result: for example, the single dž character is converted to d + z + caron in this form. The formal definition is D119 in Section 3.11, Normalization Forms."

octet 8-bit byte

Plane "A range of 65,536 (1000016) contiguous Unicode code points, where the first code point is an integer multipleof 65,536 (1000016). Planes are numbered from 0 to 16, with the number being the first code point of the plane divided by 65,536. Thus Plane 0 is U+0000..U+FFFF, Plane 1 is U+10000..U+1FFFF, ..., and Plane 16 (1016) is U+100000..10FFFF. (Note that ISO/IEC 10646 uses hexadecimal notation for the plane

numbers-for example, Plane B instead of Plane 11). (See Basic Multilingual Plane and supplementary planes.)"

RFC Request For Comments
A formal document published by the IETF defining, e.g., a protocol. RFC documents contain technical specifications and organizational notes for the Internet.

- Best Current Practice (BCP)
- Experimental
- Informational
- Proposed Standard
- Internet Standard (STD)
- Historic

surrogate A code point in the range U+D800-U+DFFF used to encode 21-bit code points into pairs of 16-bit bytes.

surrogate pair A high surrogate (in the range U+D800-U+DBFF) followed by a low surrogate (in the range U+DC00-U+DFFF), collectively representing a 21-bit code point.

TBD To Be Determined.

UCS Universal Character Set, ISO 10646, roughly equivalent to Unicode

UCS-2 A 16 bit subset of Unicode, containing only the BMP.

Unicode Consortium A non-profit corporation devoted to developing, maintaining, and promoting software internationalization standards and data.

UTF-8 UCS Transformation Format 8 ([Unicode Sec. 3.9] and [RFC 3629]).
The encoding of Unicode endorsed by the IETF

UTF-8 octet sequence The sequence of octets representing a single Unicode code point. It may consist of a single ASCII character padded on the left with a zero bit, or of a one octet introducers followed by a 1-3 octet tail.

Code points	UTF-8 octet sequence
U+0000 - U+007F	0xxxxxxx
U+0080 - U+07FF	110xxxxx 10xxxxxx
U+0800 - U+FFFF	1110xxxx 10xxxxxx 10xxxxxx
U+10000 - U+10FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

UTF-16 UCS Transformation Format 16 ([Unicode Sec. 3.9] and [RFC 2781]).

8 Resources

There are some useful tools available on the WWW:

CLDR CLDR Releases/Downloads

Compart Unicode Look up, e.g., code point.

TUTOR The Unicode Tools Of Rexx

UCD Unicode® Character Database (UCD)

UCD index Unicode® Character Name Index

9 Bibliography

This version of the document prints all included bibliography entries, even those note cited.

GitHub ARB documents

[ARB: Issue Status] *Unicode Standard Development - issue status*. URL: https://github.com/users/RexxLA/projects/2/views/1 (visited on 09/27/2023).

[ARB:Issues] *Unicode Standard Development - issues*. URL: https://github.com/users/RexxLA/projects/2/views/6 (visited on 09/27/2023) (cit. on pp. 2, 4).

[ARB:New Types] *New types of strings*. URL: https://github.com/RexxLA/rexx-repository/edit/master/ARB/standards/work-in-progress/unicode/UnicodeTools/doc/string-types.md (visited on 09/27/2023).

[RXU] Josep Maria Blasco. *The RXU Rexx Preprocessor for Unicode*. URL: https://github.com/RexxLA/rexx-repository/blob/master/ARB/standards/work-in-progress/unicode/UnicodeTools/doc/rxu.md (visited on 09/27/2023).

[RXU: built-in functions] Josep Maria Blasco. *RXU: Rexx built-in functions for Unicode: enhancements and modifications*. URL: https://github.com/RexxLA/rexx-repository/blob/master/ARB/standards/work-in-progress/unicode/UnicodeTools/doc/built-in.md (visited on 10/27/2023).

[RXU: New built-in functions] Josep Maria Blasco. *RXU: New Built-in functions*. URL: https://github.com/RexxLA/rexx-repository/blob/master/ARB/standards/work-in-progress/unicode/UnicodeTools/doc/new-functions.md (visited on 10/31/2023) (cit. on p. 11).

[RXU: New Types] Josep Maria Blasco. *RXU: Four new types of string*. URL: https://github.com/RexxLA/rexx-repository/blob/master/ARB/standards/work-in-progress/unicode/UnicodeTools/doc/string-types.md (visited on 10/27/2023) (cit. on p. 7).

[RXU: OPTIONS] Josep Maria Blasco. *RXU: New values for the OPTIONS instruction*. URL: https://github.com/RexxLA/rexx-repository/blob/master/ARB/standards/work-in-progress/unicode/UnicodeTools/doc/options.md (visited on 09/27/2023) (cit. on p. 6).

[RXU: STREAM] Josep Maria Blasco. *RXU: Stream functions for Unicode*. URL: https://github.com/RexxLA/rexx-repository/blob/master/ARB/standards/work-in-progress/unicode/UnicodeTools/doc/stream.md (visited on 09/27/2023) (cit. on p. 8).

FAQs

[FAQ: chars and CM] *FAQ: Characters & Combining Marks*. URL: https://unicode.org/faq/char_combmark.html (visited on 09/27/2023).

[FAQ: Norm] *FAQ: Normalization*. URL: https://unicode.org/faq/normalization.html (visited on 09/27/2023).

[FAQ: Spec] *FAQ: Specifications*. URL: https://unicode.org/faq/specifications.html (visited on 09/27/2023).

[Unicode FAQs] *Unicode Frequently Asked Questions*. URL: https://unicode.org/faq/ (visited on 09/27/2023) (cit. on p. 2).

Official documents

[ISO/IEC 10646] *Information technology - Universal coded character set (UCS)*. standard. URL: https://www.iso.org/standard/76835.html (visited on 09/27/2023).

[ISO/IEC 8859-1:1997 (E)] *Final Text of DIS 8859-1, 8-bit single-byte coded graphic character sets-Part 1: Latin alphabet No.1*. standard. URL: https://www.open-std.org/JTC1/sc2/wg3/docs/n411.pdf (visited on 09/27/2023).

[RFC 2781] *UTF-16, a transformation format of ISO 10646*. standard. URL: https://datatracker.ietf.org/doc/rfc2781/ (visited on 09/27/2023) (cit. on p. 14).

[RFC 3629] *UTF-8, a transformation format of ISO 10646*. standard. URL: https://datatracker.ietf.org/doc/rfc3629/ (visited on 09/27/2023) (cit. on pp. 2, 14).

[RFC 5137] *ASCII Escaping of Unicode Characters*. standard. URL: https://datatracker.ietf.org/doc/rfc5137/ (visited on 09/27/2023) (cit. on p. 7).

[RFC 5198] *Unicode Format for Network Interchange*. standard. URL: https://datatracker.ietf.org/doc/rfc5198/ (visited on 09/27/2023) (cit. on p. 2).

[UAX 15] *Unicode® Standard Annex #15. Unicode Normalization Forms*. URL: https://unicode.org/reports/tr15/ (visited on 09/25/2023).

[UAX 18] *Unicode® Standard Annex #18. Unicode Regular Expressions*. URL: https://unicode.org/reports/tr18/ (visited on 10/29/2023).

[UAX 29] *Unicode® Standard Annex #29. Unicode Text Segmentation*. URL: https://www.unicode.org/reports/tr29/ (visited on 09/25/2023) (cit. on p. 13).

[UAX 31] *Unicode® Standard Annex #31. Unicode Identifiers and Syntax*. URL: https://www.unicode.org/reports/tr31/ (visited on 09/25/2023) (cit. on pp. 3, 4).

[UAX 42] *Unicode® Standard Annex #42. Unicode Character Database in XML*. URL: https://www.unicode.org/reports/tr42/ (visited on 09/25/2023).

[UAX 44] *Unicode® Standard Annex #44. Unicode Character Database*. URL: https://www.unicode.org/reports/tr44/ (visited on 09/25/2023).

[UCD] *Unicode® Character Database (UCD)*. URL: https://www.unicode.org/ucd/ (visited on 09/25/2023) (cit. on pp. 9, 10).

[Unicode] *The Unicode® Standard*. standard. Sept. 2022. URL: https://www.unicode.org/versions/Unicode15.0.0/ (visited on 09/25/2023) (cit. on pp. 2, 8, 12).

[Unicode CLDR Project] *Unicode® Common Locale Data Repository (CLDR)*. URL: https://cldr.unicode.org/ (visited on 09/25/2023).

[Unicode Glossary] *Glossary of Unicode Terms*. URL: https://unicode.org/glossary/ (visited on 09/25/2023) (cit. on pp. 2, 12).

[Unicode Sec. 3.9] *The Unicode® Standard*. standard. Sept. 2022. Chap. 3.9 Unicode Encoding Forms, pp. 119–129. URL: https://www.unicode.org/versions/Unicode15.0.0/ch03.pdf#page=50 (visited on 09/25/2023) (cit. on p. 14).

[UTS #10] *Unicode® Technical Standard #10. Unicode Collation Algorithm*. URL: https://www.unicode.org/reports/tr10/ (visited on 09/27/2023).

[UTS #18] *Unicode® Technical Standard #18. Unicode Regular Expressions*. URL: https://www.unicode.org/reports/tr18/ (visited on 09/27/2023).

[UTS #55] *Unicode® Technical Standard #55. Unicode Source Code Handling*. URL: https://www.unicode.org/reports/tr55/ (visited on 09/27/2023).

Index