Full credit is given to the above companies including the OS that this PDF file was generated!

Red Hat Enterprise Linux Release 9.2 Manual Pages on 'pcre2unicode.3' command

$ man pcre2unicode.3

PCRE2UNICODE(3) Library Functions Manual PCRE2UNICODE(3)

NAME

PCRE - Perl-compatible regular expressions (revised API)

UNICODE AND UTF SUPPORT

PCRE2 is normally built with Unicode support, though if you do not need it, you can build it without, in which case the library will be smaller. With Unicode support, PCRE2 has knowledge of Unicode character properties and can process strings of text in UTF-8, UTF-16, and UTF-32 format (depending on the code unit width), but this is not the default. Unless specifically requested, PCRE2 treats each code unit in a string as one character.

There are two ways of telling PCRE2 to switch to UTF mode, where char? acters may consist of more than one code unit and the range of values is constrained. The program can call pcre2_compile() with the PCRE2_UTF option, or the pattern may start with the sequence (*UTF). However, the latter facility can be locked out by the PCRE2_NEVER_UTF option. That is, the programmer can prevent the supplier of the pattern from switching to UTF mode.

Note that the PCRE2_MATCH_INVALID_UTF option (see below) forces PCRE2_UTF to be set.

In UTF mode, both the pattern and any subject strings that are matched against it are treated as UTF strings instead of strings of individual one-code-unit characters. There are also some other changes to the way

characters are handled, as documented below.

UNICODE PROPERTY SUPPORT

When PCRE2 is built with Unicode support, the escape sequences \p{..}, \P{..}, and \X can be used. This is not dependent on the PCRE2_UTF setting. The Unicode properties that can be tested are a subset of those that Perl supports. Currently they are limited to the general category properties such as Lu for an upper case letter or Nd for a decimal number, the Unicode script names such as Arabic or Han, Bidi_Class, Bidi_Control, and the derived properties Any and LC (synonym L&). Full lists are given in the pcre2pattern and pcre2syntax documentation. In general, only the short names for properties are supported. For example, \p{L} matches a letter. Its longer synonym, \p{Letter}, is not supported. Furthermore, in Perl, many properties may optionally be prefixed by "Is", for compatibility with Perl 5.6. PCRE2 does not support this.

WIDE CHARACTERS AND UTF MODES

Code points less than 256 can be specified in patterns by either braced or unbraced hexadecimal escape sequences (for example, \x{b3} or \xb3). Larger values have to use braced sequences. Unbraced octal code points up to \777 are also recognized; larger ones can be coded using \o{...}.

The escape sequence \N{U+<hex digits>} is recognized as another way of specifying a Unicode character by code point in a UTF mode. It is not allowed in non-UTF mode.

In UTF mode, repeat quantifiers apply to complete UTF characters, not to individual code units.

In UTF mode, the dot metacharacter matches one UTF character instead of a single code unit.

In UTF mode, capture group names are not restricted to ASCII, and may contain any Unicode letters and decimal digits, as well as underscore.

The escape sequence \C can be used to match a single code unit in UTF mode, but its use can lead to some strange effects because it breaks up multi-unit characters (see the description of \C in the pcre2pattern documentation). For this reason, there is a build-time option that dis?

ables support for \C completely. There is also a less draconian com? pile-time option for locking out the use of \C when a pattern is com? piled.

The use of \C is not supported by the alternative matching function pcre2_dfa_match() when in UTF-8 or UTF-16 mode, that is, when a charac? ter may consist of more than one code unit. The use of \C in these modes provokes a match-time error. Also, the JIT optimization does not support \C in these modes. If JIT optimization is requested for a UTF-8 or UTF-16 pattern that contains \C, it will not succeed, and so when pcre2_match() is called, the matching will be carried out by the inter? pretive function.

The character escapes \b, \B, \d, \D, \s, \S, \w, and \W correctly test characters of any code value, but, by default, the characters that PCRE2 recognizes as digits, spaces, or word characters remain the same set as in non-UTF mode, all with code points less than 256. This re? mains true even when PCRE2 is built to include Unicode support, because to do otherwise would slow down matching in many common cases. Note that this also applies to \b and \B, because they are defined in terms of \w and \W. If you want to test for a wider sense of, say, "digit", you can use explicit Unicode property tests such as \p{Nd}. Alterna? tively, if you set the PCRE2_UCP option, the way that the character es? capes work is changed so that Unicode properties are used to determine which characters match. There are more details in the section on generic character types in the pcre2pattern documentation.

Similarly, characters that match the POSIX named character classes are all low-valued characters, unless the PCRE2_UCP option is set.

However, the special horizontal and vertical white space matching es? capes (\h, \H, \v, and \V) do match all the appropriate Unicode charac? ters, whether or not PCRE2_UCP is set.

UNICODE CASE-EQUIVALENCE

If either PCRE2_UTF or PCRE2_UCP is set, upper/lower case processing makes use of Unicode properties except for characters whose code points are less than 128 and that have at most two case-equivalent values. For

these, a direct table lookup is used for speed. A few Unicode charac?
ters such as Greek sigma have more than two code points that are case-
equivalent, and these are treated specially. Setting PCRE2_UCP without
PCRE2_UTF allows Unicode-style case processing for non-UTF character
encodings such as UCS-2.

SCRIPT RUNS

The pattern constructs (*script_run:...) and (*atomic_script_run:...),
with synonyms (*sr:...) and (*asr:...), verify that the string matched
within the parentheses is a script run. In concept, a script run is a
sequence of characters that are all from the same Unicode script. How?
ever, because some scripts are commonly used together, and because some
diacritical and other marks are used with multiple scripts, it is not
that simple.

Every Unicode character has a Script property, mostly with a value cor?
responding to the name of a script, such as Latin, Greek, or Cyrillic.
There are also three special values:

"Unknown" is used for code points that have not been assigned, and also
for the surrogate code points. In the PCRE2 32-bit library, characters
whose code points are greater than the Unicode maximum (U+10FFFF),
which are accessible only in non-UTF mode, are assigned the Unknown
script.

"Common" is used for characters that are used with many scripts. These
include punctuation, emoji, mathematical, musical, and currency sym?
bols, and the ASCII digits 0 to 9.

"Inherited" is used for characters such as diacritical marks that mod?
ify a previous character. These are considered to take on the script of
the character that they modify.

Some Inherited characters are used with many scripts, but many of them
are only normally used with a small number of scripts. For example,
U+102E0 (Coptic Epact thousands mark) is used only with Arabic and Cop?
tic. In order to make it possible to check this, a Unicode property
called Script Extension exists. Its value is a list of scripts that ap?
ply to the character. For the majority of characters, the list contains

just one script, the same one as the Script property. However, for characters such as U+102E0 more than one Script is listed. There are also some Common characters that have a single, non-Common script in their Script Extension list.

The next section describes the basic rules for deciding whether a given string of characters is a script run. Note, however, that there are some special cases involving the Chinese Han script, and an additional constraint for decimal digits. These are covered in subsequent sec? tions.

Basic script run rules

A string that is less than two characters long is a script run. This is the only case in which an Unknown character can be part of a script run. Longer strings are checked using only the Script Extensions prop? erty, not the basic Script property.

If a character's Script Extension property is the single value "Inher? ited", it is always accepted as part of a script run. This is also true for the property "Common", subject to the checking of decimal digits described below. All the remaining characters in a script run must have at least one script in common in their Script Extension lists. In set-theoretic terminology, the intersection of all the sets of scripts must not be empty.

A simple example is an Internet name such as "google.com". The letters are all in the Latin script, and the dot is Common, so this string is a script run. However, the Cyrillic letter "o" looks exactly the same as the Latin "o"; a string that looks the same, but with Cyrillic "o"s is not a script run.

More interesting examples involve characters with more than one script in their Script Extension. Consider the following characters:

U+060C Arabic comma

U+06D4 Arabic full stop

The first has the Script Extension list Arabic, Hanifi Rohingya, Syr? iac, and Thaana; the second has just Arabic and Hanifi Rohingya. Both of them could appear in script runs of either Arabic or Hanifi Ro?

hingya. The first could also appear in Syriac or Thaana script runs, but the second could not.

The Chinese Han script

The Chinese Han script is commonly used in conjunction with other scripts for writing certain languages. Japanese uses the Hiragana and Katakana scripts together with Han; Korean uses Hangul and Han; Tai? wanese Mandarin uses Bopomofo and Han. These three combinations are treated as special cases when checking script runs and are, in effect, "virtual scripts". Thus, a script run may contain a mixture of Hira? gana, Katakana, and Han, or a mixture of Hangul and Han, or a mixture of Bopomofo and Han, but not, for example, a mixture of Hangul and Bopomofo and Han. PCRE2 (like Perl) follows Unicode's Technical Stan? dard 39 ("Unicode Security Mechanisms", http://unicode.org/re? ports/tr39/) in allowing such mixtures.

Decimal digits

Unicode contains many sets of 10 decimal digits in different scripts, and some scripts (including the Common script) contain more than one set. Some of these decimal digits them are visually indistinguishable from the common ASCII digits. In addition to the script checking de? scribed above, if a script run contains any decimal digits, they must all come from the same set of 10 adjacent characters.

VALIDITY OF UTF STRINGS

When the PCRE2_UTF option is set, the strings passed as patterns and subjects are (by default) checked for validity on entry to the relevant functions. If an invalid UTF string is passed, a negative error code is returned. The code unit offset to the offending character can be ex? tracted from the match data block by calling pcre2_get_startchar(), which is used for this purpose after a UTF error.

In some situations, you may already know that your strings are valid, and therefore want to skip these checks in order to improve perfor? mance, for example in the case of a long subject string that is being scanned repeatedly. If you set the PCRE2_NO_UTF_CHECK option at com? pile time or at match time, PCRE2 assumes that the pattern or subject

it is given (respectively) contains only valid UTF code unit sequences.

If you pass an invalid UTF string when PCRE2_NO_UTF_CHECK is set, the result is undefined and your program may crash or loop indefinitely or give incorrect results. There is, however, one mode of matching that can handle invalid UTF subject strings. This is enabled by passing PCRE2_MATCH_INVALID_UTF to pcre2_compile() and is discussed below in the next section. The rest of this section covers the case when PCRE2_MATCH_INVALID_UTF is not set.

Passing PCRE2_NO_UTF_CHECK to pcre2_compile() just disables the UTF check for the pattern; it does not also apply to subject strings. If you want to disable the check for a subject string you must pass this same option to pcre2_match() or pcre2_dfa_match().

UTF-16 and UTF-32 strings can indicate their endianness by special code knows as a byte-order mark (BOM). The PCRE2 functions do not handle this, expecting strings to be in host byte order.

Unless PCRE2_NO_UTF_CHECK is set, a UTF string is checked before any other processing takes place. In the case of pcre2_match() and pcre2_dfa_match() calls with a non-zero starting offset, the check is applied only to that part of the subject that could be inspected during matching, and there is a check that the starting offset points to the first code unit of a character or to the end of the subject. If there are no lookbehind assertions in the pattern, the check starts at the starting offset. Otherwise, it starts at the length of the longest lookbehind before the starting offset, or at the start of the subject if there are not that many characters before the starting offset. Note that the sequences \b and \B are one-character lookbehinds.

In addition to checking the format of the string, there is a check to ensure that all code points lie in the range U+0 to U+10FFFF, excluding the surrogate area. The so-called "non-character" code points are not excluded because Unicode corrigendum #9 makes it clear that they should not be.

Characters in the "Surrogate Area" of Unicode are reserved for use by UTF-16, where they are used in pairs to encode code points with values

greater than 0xFFFF. The code points that are encoded by UTF-16 pairs are available independently in the UTF-8 and UTF-32 encodings. (In other words, the whole surrogate thing is a fudge for UTF-16 which un? fortunately messes up UTF-8 and UTF-32.)

Setting PCRE2_NO_UTF_CHECK at compile time does not disable the error that is given if an escape sequence for an invalid Unicode code point is encountered in the pattern. If you want to allow escape sequences such as \x{d800} (a surrogate code point) you can set the PCRE2_EX? TRA_ALLOW_SURROGATE_ESCAPES extra option. However, this is possible only in UTF-8 and UTF-32 modes, because these values are not repre? sentable in UTF-16.

Errors in UTF-8 strings

The following negative error codes are given for invalid UTF-8 strings:

PCRE2_ERROR_UTF8_ERR1

PCRE2_ERROR_UTF8_ERR2

PCRE2_ERROR_UTF8_ERR3

PCRE2_ERROR_UTF8_ERR4

PCRE2_ERROR_UTF8_ERR5

The string ends with a truncated UTF-8 character; the code specifies how many bytes are missing (1 to 5). Although RFC 3629 restricts UTF-8 characters to be no longer than 4 bytes, the encoding scheme (origi? nally defined by RFC 2279) allows for up to 6 bytes, and this is checked first; hence the possibility of 4 or 5 missing bytes.

PCRE2_ERROR_UTF8_ERR6

PCRE2_ERROR_UTF8_ERR7

PCRE2_ERROR_UTF8_ERR8

PCRE2_ERROR_UTF8_ERR9

PCRE2_ERROR_UTF8_ERR10

The two most significant bits of the 2nd, 3rd, 4th, 5th, or 6th byte of the character do not have the binary value 0b10 (that is, either the most significant bit is 0, or the next bit is 1).

PCRE2_ERROR_UTF8_ERR11

PCRE2_ERROR_UTF8_ERR12

A character that is valid by the RFC 2279 rules is either 5 or 6 bytes long; these code points are excluded by RFC 3629.

PCRE2_ERROR_UTF8_ERR13

A 4-byte character has a value greater than 0x10ffff; these code points are excluded by RFC 3629.

PCRE2_ERROR_UTF8_ERR14

A 3-byte character has a value in the range 0xd800 to 0xdfff; this range of code points are reserved by RFC 3629 for use with UTF-16, and so are excluded from UTF-8.

PCRE2_ERROR_UTF8_ERR15

PCRE2_ERROR_UTF8_ERR16

PCRE2_ERROR_UTF8_ERR17

PCRE2_ERROR_UTF8_ERR18

PCRE2_ERROR_UTF8_ERR19

A 2-, 3-, 4-, 5-, or 6-byte character is "overlong", that is, it codes for a value that can be represented by fewer bytes, which is invalid. For example, the two bytes 0xc0, 0xae give the value 0x2e, whose cor? rect coding uses just one byte.

PCRE2_ERROR_UTF8_ERR20

The two most significant bits of the first byte of a character have the binary value 0b10 (that is, the most significant bit is 1 and the sec? ond is 0). Such a byte can only validly occur as the second or subse? quent byte of a multi-byte character.

PCRE2_ERROR_UTF8_ERR21

The first byte of a character has the value 0xfe or 0xff. These values can never occur in a valid UTF-8 string.

Errors in UTF-16 strings

The following negative error codes are given for invalid UTF-16 strings:

PCRE2_ERROR_UTF16_ERR1 Missing low surrogate at end of string

PCRE2_ERROR_UTF16_ERR2 Invalid low surrogate follows high surrogate

PCRE2_ERROR_UTF16_ERR3 Isolated low surrogate

Errors in UTF-32 strings

The following negative error codes are given for invalid UTF-32 strings:

PCRE2_ERROR_UTF32_ERR1 Surrogate character (0xd800 to 0xdfff)

PCRE2_ERROR_UTF32_ERR2 Code point is greater than 0x10ffff

MATCHING IN INVALID UTF STRINGS

You can run pattern matches on subject strings that may contain invalid UTF sequences if you call pcre2_compile() with the PCRE2_MATCH_IN? VALID_UTF option. This is supported by pcre2_match(), including JIT matching, but not by pcre2_dfa_match(). When PCRE2_MATCH_INVALID_UTF is set, it forces PCRE2_UTF to be set as well. Note, however, that the pattern itself must be a valid UTF string.

Setting PCRE2_MATCH_INVALID_UTF does not affect what pcre2_compile() generates, but if pcre2_jit_compile() is subsequently called, it does generate different code. If JIT is not used, the option affects the be? haviour of the interpretive code in pcre2_match(). When PCRE2_MATCH_IN? VALID_UTF is set at compile time, PCRE2_NO_UTF_CHECK is ignored at match time.

In this mode, an invalid code unit sequence in the subject never matches any pattern item. It does not match dot, it does not match \p{Any}, it does not even match negative items such as [^X]. A lookbe? hind assertion fails if it encounters an invalid sequence while moving the current point backwards. In other words, an invalid UTF code unit sequence acts as a barrier which no match can cross.

You can also think of this as the subject being split up into fragments of valid UTF, delimited internally by invalid code unit sequences. The pattern is matched fragment by fragment. The result of a successful match, however, is given as code unit offsets in the entire subject string in the usual way. There are a few points to consider:

The internal boundaries are not interpreted as the beginnings or ends of lines and so do not match circumflex or dollar characters in the pattern.

If pcre2_match() is called with an offset that points to an invalid UTF-sequence, that sequence is skipped, and the match starts at the

next valid UTF character, or the end of the subject.

At internal fragment boundaries, \b and \B behave in the same way as at the beginning and end of the subject. For example, a sequence such as \bWORD\b would match an instance of WORD that is surrounded by invalid UTF code units.

Using PCRE2_MATCH_INVALID_UTF, an application can run matches on arbitrary data, knowing that any matched strings that are returned are valid UTF. This can be useful when searching for UTF text in executable or other binary files.

AUTHOR

Philip Hazel

Retired from University Computing Service

Cambridge, England.

REVISION

Last updated: 22 December 2021