Linux Ubuntu 22.4.5 Manual Pages on command 'pcre2test.1'

$ man pcre2test.1

PCRE2TEST(1) General Commands Manual PCRE2TEST(1)

NAME

pcre2test - a program for testing Perl-compatible regular expressions.

SYNOPSIS

pcre2test [options] [input file [output file]]

pcre2test is a test program for the PCRE2 regular expression libraries, but it can also be used for experimenting with regular expressions. This document describes the features of the test program; for details of the regular expressions them? selves, see the pcre2pattern documentation. For details of the PCRE2 library func? tion calls and their options, see the pcre2api documentation.

The input for pcre2test is a sequence of regular expression patterns and subject strings to be matched. There are also command lines for setting defaults and con? trolling some special actions. The output shows the result of each match attempt. Modifiers on external or internal command lines, the patterns, and the subject lines specify PCRE2 function options, control how the subject is processed, and what output is produced.

As the original fairly simple PCRE library evolved, it acquired many different fea? tures, and as a result, the original pcretest program ended up with a lot of op? tions in a messy, arcane syntax for testing all the features. The move to the new PCRE2 API provided an opportunity to re-implement the test program as pcre2test, with a cleaner modifier syntax. Nevertheless, there are still many obscure modi? fiers, some of which are specifically designed for use in conjunction with the test

script and data files that are distributed as part of PCRE2. All the modifiers are documented here, some without much justification, but many of them are unlikely to be of use except when testing the libraries.

PCRE2's 8-BIT, 16-BIT AND 32-BIT LIBRARIES

Different versions of the PCRE2 library can be built to support character strings that are encoded in 8-bit, 16-bit, or 32-bit code units. One, two, or all three of these libraries may be simultaneously installed. The pcre2test program can be used to test all the libraries. However, its own input and output are always in 8-bit format. When testing the 16-bit or 32-bit libraries, patterns and subject strings are converted to 16-bit or 32-bit format before being passed to the library func? tions. Results are converted back to 8-bit code units for output.

In the rest of this document, the names of library functions and structures are given in generic form, for example, pcre_compile(). The actual names used in the libraries have a suffix _8, _16, or _32, as appropriate.

INPUT ENCODING

Input to pcre2test is processed line by line, either by calling the C library's fgets() function, or via the libreadline library. In some Windows environments character 26 (hex 1A) causes an immediate end of file, and no further data is read, so this character should be avoided unless you really want that action.

The input is processed using using C's string functions, so must not contain binary zeros, even though in Unix-like environments, fgets() treats any bytes other than newline as data characters. An error is generated if a binary zero is encountered.

By default subject lines are processed for backslash escapes, which makes it possi? ble to include any data value in strings that are passed to the library for match? ing. For patterns, there is a facility for specifying some or all of the 8-bit in? put characters as hexadecimal pairs, which makes it possible to include binary ze? ros.

Input for the 16-bit and 32-bit libraries

When testing the 16-bit or 32-bit libraries, there is a need to be able to generate character code points greater than 255 in the strings that are passed to the li? brary. For subject lines, backslash escapes can be used. In addition, when the utf modifier (see "Setting compilation options" below) is set, the pattern and any fol? lowing subject lines are interpreted as UTF-8 strings and translated to UTF-16 or

UTF-32 as appropriate.

For non-UTF testing of wide characters, the utf8_input modifier can be used. This is mutually exclusive with utf, and is allowed only in 16-bit or 32-bit mode. It causes the pattern and following subject lines to be treated as UTF-8 according to the original definition (RFC 2279), which allows for character values up to 0x7fffffff. Each character is placed in one 16-bit or 32-bit code unit (in the 16-bit case, values greater than 0xffff cause an error to occur).

UTF-8 (in its original definition) is not capable of encoding values greater than 0x7fffffff, but such values can be handled by the 32-bit library. When testing this library in non-UTF mode with utf8_input set, if any character is preceded by the byte 0xff (which is an invalid byte in UTF-8) 0x80000000 is added to the charac? ter's value. This is the only way of passing such code points in a pattern string. For subject strings, using an escape sequence is preferable.

COMMAND LINE OPTIONS

-8 If the 8-bit library has been built, this option causes it to be used (this is the default). If the 8-bit library has not been built, this op? tion causes an error.

-16 If the 16-bit library has been built, this option causes it to be used. If only the 16-bit library has been built, this is the default. If the 16-bit library has not been built, this option causes an error.

-32 If the 32-bit library has been built, this option causes it to be used. If only the 32-bit library has been built, this is the default. If the 32-bit library has not been built, this option causes an error.

-ac Behave as if each pattern has the auto_callout modifier, that is, insert automatic callouts into every pattern that is compiled.

-AC As for -ac, but in addition behave as if each subject line has the call? out_extra modifier, that is, show additional information from callouts.

-b Behave as if each pattern has the fullbincode modifier; the full internal binary form of the pattern is output after compilation.

-C Output the version number of the PCRE2 library, and all available infor? mation about the optional features that are included, and then exit with zero exit code. All other options are ignored. If both -C and -LM are present, whichever is first is recognized.

-C option Output information about a specific build-time option, then exit. This functionality is intended for use in scripts such as RunTest. The follow? ing options output the value and set the exit code as indicated:

ebcdic-nl the code for LF (= NL) in an EBCDIC environment:
0x15 or 0x25
0 if used in an ASCII environment
exit code is always 0

linksize the configured internal link size (2, 3, or 4)
exit code is set to the link size

newline the default newline setting:
CR, LF, CRLF, ANYCRLF, ANY, or NUL
exit code is always 0

bsr the default setting for what \R matches:
ANYCRLF or ANY
exit code is always 0

The following options output 1 for true or 0 for false, and set the exit code to the same value:

backslash-C \C is supported (not locked out)

ebcdic compiled for an EBCDIC environment

jit just-in-time support is available

pcre2-16 the 16-bit library was built

pcre2-32 the 32-bit library was built

pcre2-8 the 8-bit library was built

unicode Unicode support is available

If an unknown option is given, an error message is output; the exit code is 0.

-d Behave as if each pattern has the debug modifier; the internal form and information about the compiled pattern is output after compilation; -d is equivalent to -b -i.

-dfa Behave as if each subject line has the dfa modifier; matching is done us? ing the pcre2_dfa_match() function instead of the default pcre2_match().

-error number[,number,...]

Call pcre2_get_error_message() for each of the error numbers in the

comma-separated list, display the resulting messages on the standard out?
put, then exit with zero exit code. The numbers may be positive or nega?
tive. This is a convenience facility for PCRE2 maintainers.

-help Output a brief summary these options and then exit.

-i Behave as if each pattern has the info modifier; information about the compiled pattern is given after compilation.

-jit Behave as if each pattern line has the jit modifier; after successful compilation, each pattern is passed to the just-in-time compiler, if available.

-jitfast Behave as if each pattern line has the jitfast modifier; after successful compilation, each pattern is passed to the just-in-time compiler, if available, and each subject line is passed directly to the JIT matcher via its "fast path".

-jitverify

Behave as if each pattern line has the jitverify modifier; after success?
ful compilation, each pattern is passed to the just-in-time compiler, if available, and the use of JIT for matching is verified.

-LM List modifiers: write a list of available pattern and subject modifiers to the standard output, then exit with zero exit code. All other options are ignored. If both -C and -LM are present, whichever is first is rec?
ognized.

-pattern modifier-list

Behave as if each pattern line contains the given modifiers.

-q Do not output the version number of pcre2test at the start of execution.

-S size On Unix-like systems, set the size of the run-time stack to size mebibytes (units of 1024*1024 bytes).

-subject modifier-list

Behave as if each subject line contains the given modifiers.

-t Run each compile and match many times with a timer, and output the re?
sulting times per compile or match. When JIT is used, separate times are given for the initial compile and the JIT compile. You can control the number of iterations that are used for timing by following -t with a num?
ber (as a separate item on the command line). For example, "-t 1000" it?

erates 1000 times. The default is to iterate 500,000 times.

-tm This is like -t except that it times only the matching phase, not the compile phase.

-T -TM These behave like -t and -tm, but in addition, at the end of a run, the total times for all compiles and matches are output.

-version Output the PCRE2 version number and then exit.

DESCRIPTION

If pcre2test is given two filename arguments, it reads from the first and writes to the second. If the first name is "-", input is taken from the standard input. If pcre2test is given only one argument, it reads from that file and writes to stdout. Otherwise, it reads from stdin and writes to stdout.

When pcre2test is built, a configuration option can specify that it should be linked with the libreadline or libedit library. When this is done, if the input is from a terminal, it is read using the readline() function. This provides line-edit? ing and history facilities. The output from the -help option states whether or not readline() will be used.

The program handles any number of tests, each of which consists of a set of input lines. Each set starts with a regular expression pattern, followed by any number of subject lines to be matched against that pattern. In between sets of test data, command lines that begin with # may appear. This file format, with some restric? tions, can also be processed by the perltest.sh script that is distributed with PCRE2 as a means of checking that the behaviour of PCRE2 and Perl is the same. For a specification of perltest.sh, see the comments near its beginning.

When the input is a terminal, pcre2test prompts for each line of input, using "re>" to prompt for regular expression patterns, and "data>" to prompt for subject lines. Command lines starting with # can be entered only in response to the "re>" prompt.

Each subject line is matched separately and independently. If you want to do multi- line matches, you have to use the \n escape sequence (or \r or \r\n, etc., depend? ing on the newline setting) in a single line of input to encode the newline se? quences. There is no limit on the length of subject lines; the input buffer is au? tomatically extended if it is too small. There are replication features that makes it possible to generate long repetitive pattern or subject lines without having to supply them explicitly.

An empty line or the end of the file signals the end of the subject lines for a test, at which point a new pattern or command line is expected if there is still input to be read.

COMMAND LINES

In between sets of test data, a line that begins with # is interpreted as a command line. If the first character is followed by white space or an exclamation mark, the line is treated as a comment, and ignored. Otherwise, the following commands are recognized:

#forbid_utf

Subsequent patterns automatically have the PCRE2_NEVER_UTF and PCRE2_NEVER_UCP options set, which locks out the use of the PCRE2_UTF and PCRE2_UCP options and the use of (*UTF) and (*UCP) at the start of patterns. This command also forces an error if a subsequent pattern contains any occurrences of \P, \p, or \X, which are still supported when PCRE2_UTF is not set, but which require Unicode property support to be included in the library.

This is a trigger guard that is used in test files to ensure that UTF or Unicode property tests are not accidentally added to files that are used when Unicode support is not included in the library. Setting PCRE2_NEVER_UTF and PCRE2_NEVER_UCP as a default can also be obtained by the use of #pattern; the difference is that #forbid_utf cannot be unset, and the automatic options are not displayed in pattern information, to avoid cluttering up test output.

#load <filename>

This command is used to load a set of precompiled patterns from a file, as described in the section entitled "Saving and restoring compiled patterns" below.

#newline_default [<newline-list>]

When PCRE2 is built, a default newline convention can be specified. This determines which characters and/or character pairs are recognized as indicating a newline in a pattern or subject string. The default can be overridden when a pattern is compiled. The standard test files contain tests of various newline conventions, but the majority of the tests expect a single linefeed to be recognized as a newline by default. Without special action the tests would fail when PCRE2 is compiled with either CR or CRLF as the default newline.

The #newline_default command specifies a list of newline types that are acceptable

as the default. The types must be one of CR, LF, CRLF, ANYCRLF, ANY, or NUL (in up?
per or lower case), for example:

  #newline_default LF Any anyCRLF

If the default newline is in the list, this command has no effect. Otherwise, ex?
cept when testing the POSIX API, a newline modifier that specifies the first new?
line convention in the list (LF in the above example) is added to any pattern that
does not already have a newline modifier. If the newline list is empty, the feature
is turned off. This command is present in a number of the standard test input
files.

When the POSIX API is being tested there is no way to override the default newline
convention, though it is possible to set the newline convention from within the
pattern. A warning is given if the posix or posix_nosub modifier is used when #new?
line_default would set a default for the non-POSIX API.

  #pattern <modifier-list>

This command sets a default modifier list that applies to all subsequent patterns.
Modifiers on a pattern can change these settings.

  #perltest

The appearance of this line causes all subsequent modifier settings to be checked
for compatibility with the perltest.sh script, which is used to confirm that Perl
gives the same results as PCRE2. Also, apart from comment lines, #pattern commands,
and #subject commands that set or unset "mark", no command lines are permitted, be?
cause they and many of the modifiers are specific to pcre2test, and should not be
used in test files that are also processed by perltest.sh. The #perltest command
helps detect tests that are accidentally put in the wrong file.

  #pop [<modifiers>]

  #popcopy [<modifiers>]

These commands are used to manipulate the stack of compiled patterns, as described
in the section entitled "Saving and restoring compiled patterns" below.

  #save <filename>

This command is used to save a set of compiled patterns to a file, as described in
the section entitled "Saving and restoring compiled patterns" below.

  #subject <modifier-list>

This command sets a default modifier list that applies to all subsequent subject

lines. Modifiers on a subject line can change these settings.

MODIFIER SYNTAX

Modifier lists are used with both pattern and subject lines. Items in a list are separated by commas followed by optional white space. Trailing whitespace in a mod? ifier list is ignored. Some modifiers may be given for both patterns and subject lines, whereas others are valid only for one or the other. Each modifier has a long name, for example "anchored", and some of them must be followed by an equals sign and a value, for example, "offset=12". Values cannot contain comma characters, but may contain spaces. Modifiers that do not take values may be preceded by a minus sign to turn off a previous setting.

A few of the more common modifiers can also be specified as single letters, for ex? ample "i" for "caseless". In documentation, following the Perl convention, these are written with a slash ("the /i modifier") for clarity. Abbreviated modifiers must all be concatenated in the first item of a modifier list. If the first item is not recognized as a long modifier name, it is interpreted as a sequence of these abbreviations. For example:

```
/abc/ig,newline=cr,jit=3
```

This is a pattern line whose modifier list starts with two one-letter modifiers (/i and /g). The lower-case abbreviated modifiers are the same as used in Perl.

PATTERN SYNTAX

A pattern line must start with one of the following characters (common symbols, ex? cluding pattern meta-characters):

```
/ ! " ' ` - = _ : ; , % & @ ~
```

This is interpreted as the pattern's delimiter. A regular expression may be contin? ued over several input lines, in which case the newline characters are included within it. It is possible to include the delimiter within the pattern by escaping it with a backslash, for example

```
/abc\/def/
```

If you do this, the escape and the delimiter form part of the pattern, but since the delimiters are all non-alphanumeric, this does not affect its interpretation.

If the terminating delimiter is immediately followed by a backslash, for example,

```
/abc/\
```

then a backslash is added to the end of the pattern. This is done to provide a way

of testing the error condition that arises if a pattern finishes with a backslash, because

/abc\/

is interpreted as the first line of a pattern that starts with "abc/", causing pcre2test to read the next line as a continuation of the regular expression.

A pattern can be followed by a modifier list (details below).

SUBJECT LINE SYNTAX

Before each subject line is passed to pcre2_match() or pcre2_dfa_match(), leading and trailing white space is removed, and the line is scanned for backslash escapes, unless the subject_literal modifier was set for the pattern. The following provide a means of encoding non-printing characters in a visible way:

```
\a          alarm (BEL, \x07)
\b          backspace (\x08)
\e          escape (\x27)
\f          form feed (\x0c)
\n          newline (\x0a)
\r          carriage return (\x0d)
\t          tab (\x09)
\v          vertical tab (\x0b)
\nnn        octal character (up to 3 octal digits); always
              a byte unless > 255 in UTF-8 or 16-bit or 32-bit mode
\o{dd...}   octal character (any number of octal digits}
\xhh        hexadecimal byte (up to 2 hex digits)
\x{hh...}   hexadecimal character (any number of hex digits)
```

The use of \x{hh...} is not dependent on the use of the utf modifier on the pat? tern. It is recognized always. There may be any number of hexadecimal digits inside the braces; invalid values provoke error messages.

Note that \xhh specifies one byte rather than one character in UTF-8 mode; this makes it possible to construct invalid UTF-8 sequences for testing purposes. On the other hand, \x{hh} is interpreted as a UTF-8 character in UTF-8 mode, generating more than one byte if the value is greater than 127. When testing the 8-bit li? brary not in UTF-8 mode, \x{hh} generates one byte for values less than 256, and causes an error for greater values.

In UTF-16 mode, all 4-digit \x{hhhh} values are accepted. This makes it possible to construct invalid UTF-16 sequences for testing purposes.

In UTF-32 mode, all 4- to 8-digit \x{...} values are accepted. This makes it possi? ble to construct invalid UTF-32 sequences for testing purposes.

There is a special backslash sequence that specifies replication of one or more characters:

\[<characters>]{<count>}

This makes it possible to test long strings without having to provide them as part of the file. For example:

\[abc]{4}

is converted to "abcabcabcabc". This feature does not support nesting. To include a closing square bracket in the characters, code it as \x5D.

A backslash followed by an equals sign marks the end of the subject string and the start of a modifier list. For example:

abc\=notbol,notempty

If the subject string is empty and \= is followed by whitespace, the line is treated as a comment line, and is not used for matching. For example:

\= This is a comment.

abc\= This is an invalid modifier list.

A backslash followed by any other non-alphanumeric character just escapes that character. A backslash followed by anything else causes an error. However, if the very last character in the line is a backslash (and there is no modifier list), it is ignored. This gives a way of passing an empty line as data, since a real empty line terminates the data input.

If the subject_literal modifier is set for a pattern, all subject lines that follow are treated as literals, with no special treatment of backslashes. No replication is possible, and any subject modifiers must be set as defaults by a #subject com? mand.

PATTERN MODIFIERS

There are several types of modifier that can appear in pattern lines. Except where noted below, they may also be used in #pattern commands. A pattern's modifier list can add to or override default modifiers that were set by a previous #pattern com? mand.

Setting compilation options

The following modifiers set options for pcre2_compile(). Most of them set bits in the options argument of that function, but those whose names start with PCRE2_EXTRA are additional options that are set in the compile context. For the main options, there are some single-letter abbreviations that are the same as Perl options. There is special handling for /x: if a second x is present, PCRE2_EXTENDED is converted into PCRE2_EXTENDED_MORE as in Perl. A third appearance adds PCRE2_EXTENDED as well, though this makes no difference to the way pcre2_compile() behaves. See pcre2api for a description of the effects of these options.

```
      allow_empty_class         set PCRE2_ALLOW_EMPTY_CLASS
      allow_surrogate_escapes   set PCRE2_EXTRA_ALLOW_SURROGATE_ESCAPES
      alt_bsux                  set PCRE2_ALT_BSUX
      alt_circumflex            set PCRE2_ALT_CIRCUMFLEX
      alt_verbnames             set PCRE2_ALT_VERBNAMES
      anchored                  set PCRE2_ANCHORED
      auto_callout              set PCRE2_AUTO_CALLOUT
      bad_escape_is_literal     set PCRE2_EXTRA_BAD_ESCAPE_IS_LITERAL
   /i caseless                  set PCRE2_CASELESS
      dollar_endonly            set PCRE2_DOLLAR_ENDONLY
   /s dotall                    set PCRE2_DOTALL
      dupnames                  set PCRE2_DUPNAMES
      endanchored               set PCRE2_ENDANCHORED
      escaped_cr_is_lf          set PCRE2_EXTRA_ESCAPED_CR_IS_LF
   /x extended                  set PCRE2_EXTENDED
   /xx extended_more            set PCRE2_EXTENDED_MORE
      extra_alt_bsux            set PCRE2_EXTRA_ALT_BSUX
      firstline                 set PCRE2_FIRSTLINE
      literal                   set PCRE2_LITERAL
      match_line                set PCRE2_EXTRA_MATCH_LINE
      match_invalid_utf         set PCRE2_MATCH_INVALID_UTF
      match_unset_backref       set PCRE2_MATCH_UNSET_BACKREF
      match_word                set PCRE2_EXTRA_MATCH_WORD
   /m multiline                 set PCRE2_MULTILINE
```

never_backslash_c set PCRE2_NEVER_BACKSLASH_C

never_ucp set PCRE2_NEVER_UCP

never_utf set PCRE2_NEVER_UTF

/n no_auto_capture set PCRE2_NO_AUTO_CAPTURE

no_auto_possess set PCRE2_NO_AUTO_POSSESS

no_dotstar_anchor set PCRE2_NO_DOTSTAR_ANCHOR

no_start_optimize set PCRE2_NO_START_OPTIMIZE

no_utf_check set PCRE2_NO_UTF_CHECK

ucp set PCRE2_UCP

ungreedy set PCRE2_UNGREEDY

use_offset_limit set PCRE2_USE_OFFSET_LIMIT

utf set PCRE2_UTF

As well as turning on the PCRE2_UTF option, the utf modifier causes all non-print?
ing characters in output strings to be printed using the \x{hh...} notation. Other?
wise, those less than 0x100 are output in hex without the curly brackets. Setting
utf in 16-bit or 32-bit mode also causes pattern and subject strings to be trans?
lated to UTF-16 or UTF-32, respectively, before being passed to library functions.

Setting compilation controls

The following modifiers affect the compilation process or request information about
the pattern. There are single-letter abbreviations for some that are heavily used
in the test files.

bsr=[anycrlf|unicode] specify \R handling

/B bincode show binary code without lengths

callout_info show callout information

convert=<options> request foreign pattern conversion

convert_glob_escape=c set glob escape character

convert_glob_separator=c set glob separator character

convert_length set convert buffer length

debug same as info,fullbincode

framesize show matching frame size

fullbincode show binary code with lengths

/I info show info about compiled pattern

hex unquoted characters are hexadecimal

jit[=<number>] use JIT

jitfast use JIT fast path

jitverify verify JIT use

locale=<name> use this locale

max_pattern_length=<n> set the maximum pattern length

memory show memory used

newline=<type> set newline type

null_context compile with a NULL context

parens_nest_limit=<n> set maximum parentheses depth

posix use the POSIX API

posix_nosub use the POSIX API with REG_NOSUB

push push compiled pattern onto the stack

pushcopy push a copy onto the stack

stackguard=<number> test the stackguard feature

subject_literal treat all subject lines as literal

tables=[0|1|2] select internal tables

use_length do not zero-terminate the pattern

utf8_input treat input as UTF-8

The effects of these modifiers are described in the following sections.

Newline and \R handling

The bsr modifier specifies what \R in a pattern should match. If it is set to "any? crlf", \R matches CR, LF, or CRLF only. If it is set to "unicode", \R matches any Unicode newline sequence. The default can be specified when PCRE2 is built; if it is not, the default is set to Unicode.

The newline modifier specifies which characters are to be interpreted as newlines, both in the pattern and in subject lines. The type must be one of CR, LF, CRLF, ANYCRLF, ANY, or NUL (in upper or lower case).

Information about a pattern

The debug modifier is a shorthand for info,fullbincode, requesting all available information.

The bincode modifier causes a representation of the compiled code to be output af? ter compilation. This information does not contain length and offset values, which ensures that the same output is generated for different internal link sizes and

different code unit widths. By using bincode, the same regression tests can be used in different environments.

The fullbincode modifier, by contrast, does include length and offset values. This is used in a few special tests that run only for specific code unit widths and link sizes, and is also useful for one-off tests.

The info modifier requests information about the compiled pattern (whether it is anchored, has a fixed first character, and so on). The information is obtained from the pcre2_pattern_info() function. Here are some typical examples:

```
    re> /(?i)(^a|^b)/m,info
  Capture group count = 1
  Compile options: multiline
  Overall options: caseless multiline
  First code unit at start or follows newline
  Subject length lower bound = 1
    re> /(?i)abc/info
  Capture group count = 0
  Compile options: <none>
  Overall options: caseless
  First code unit = 'a' (caseless)
  Last code unit = 'c' (caseless)
  Subject length lower bound = 3
```

"Compile options" are those specified by modifiers; "overall options" have added options that are taken or deduced from the pattern. If both sets of options are the same, just a single "options" line is output; if there are no options, the line is omitted. "First code unit" is where any match must start; if there is more than one they are listed as "starting code units". "Last code unit" is the last literal code unit that must be present in any match. This is not necessarily the last character. These lines are omitted if no starting or ending code units are recorded. The sub? ject length line is omitted when no_start_optimize is set because the minimum length is not calculated when it can never be used.

The framesize modifier shows the size, in bytes, of the storage frames used by pcre2_match() for handling backtracking. The size depends on the number of captur? ing parentheses in the pattern.

The callout_info modifier requests information about all the callouts in the pat? tern. A list of them is output at the end of any other information that is re? quested. For each callout, either its number or string is given, followed by the item that follows it in the pattern.

Passing a NULL context

Normally, pcre2test passes a context block to pcre2_compile(). If the null_context modifier is set, however, NULL is passed. This is for testing that pcre2_compile() behaves correctly in this case (it uses default values).

Specifying pattern characters in hexadecimal

The hex modifier specifies that the characters of the pattern, except for sub? strings enclosed in single or double quotes, are to be interpreted as pairs of hexadecimal digits. This feature is provided as a way of creating patterns that contain binary zeros and other non-printing characters. White space is permitted between pairs of digits. For example, this pattern contains three characters:

```
/ab 32 59/hex
```

Parts of such a pattern are taken literally if quoted. This pattern contains nine characters, only two of which are specified in hexadecimal:

```
/ab "literal" 32/hex
```

Either single or double quotes may be used. There is no way of including the delim? iter within a substring. The hex and expand modifiers are mutually exclusive.

Specifying the pattern's length

By default, patterns are passed to the compiling functions as zero-terminated strings but can be passed by length instead of being zero-terminated. The use_length modifier causes this to happen. Using a length happens automatically (whether or not use_length is set) when hex is set, because patterns specified in hexadecimal may contain binary zeros.

If hex or use_length is used with the POSIX wrapper API (see "Using the POSIX wrap? per API" below), the REG_PEND extension is used to pass the pattern's length.

Specifying wide characters in 16-bit and 32-bit modes

In 16-bit and 32-bit modes, all input is automatically treated as UTF-8 and trans? lated to UTF-16 or UTF-32 when the utf modifier is set. For testing the 16-bit and 32-bit libraries in non-UTF mode, the utf8_input modifier can be used. It is mutu? ally exclusive with utf. Input lines are interpreted as UTF-8 as a means of speci?

fying wide characters. More details are given in "Input encoding" above.

Generating long repetitive patterns

Some tests use long patterns that are very repetitive. Instead of creating a very long input line for such a pattern, you can use a special repetition feature, simi? lar to the one described for subject lines above. If the expand modifier is present on a pattern, parts of the pattern that have the form

\[<characters>]{<count>}

are expanded before the pattern is passed to pcre2_compile(). For example, \[AB]{6000} is expanded to "ABAB..." 6000 times. This construction cannot be nested. An initial "\[" sequence is recognized only if "]{" followed by decimal digits and "}" is found later in the pattern. If not, the characters remain in the pattern unaltered. The expand and hex modifiers are mutually exclusive.

If part of an expanded pattern looks like an expansion, but is really part of the actual pattern, unwanted expansion can be avoided by giving two values in the quan? tifier. For example, \[AB]{6000,6000} is not recognized as an expansion item.

If the info modifier is set on an expanded pattern, the result of the expansion is included in the information that is output.

JIT compilation

Just-in-time (JIT) compiling is a heavyweight optimization that can greatly speed up pattern matching. See the pcre2jit documentation for details. JIT compiling hap? pens, optionally, after a pattern has been successfully compiled into an internal form. The JIT compiler converts this to optimized machine code. It needs to know whether the match-time options PCRE2_PARTIAL_HARD and PCRE2_PARTIAL_SOFT are going to be used, because different code is generated for the different cases. See the partial modifier in "Subject Modifiers" below for details of how these options are specified for each match attempt.

JIT compilation is requested by the jit pattern modifier, which may optionally be followed by an equals sign and a number in the range 0 to 7. The three bits that make up the number specify which of the three JIT operating modes are to be com? piled:

1 compile JIT code for non-partial matching

2 compile JIT code for soft partial matching

4 compile JIT code for hard partial matching

The possible values for the jit modifier are therefore:

0 disable JIT

1 normal matching only

2 soft partial matching only

3 normal and soft partial matching

4 hard partial matching only

6 soft and hard partial matching only

7 all three modes

If no number is given, 7 is assumed. The phrase "partial matching" means a call to pcre2_match() with either the PCRE2_PARTIAL_SOFT or the PCRE2_PARTIAL_HARD option set. Note that such a call may return a complete match; the options enable the possibility of a partial match, but do not require it. Note also that if you request JIT compilation only for partial matching (for example, jit=2) but do not set the partial modifier on a subject line, that match will not use JIT code because none was compiled for non-partial matching.

If JIT compilation is successful, the compiled JIT code will automatically be used when an appropriate type of match is run, except when incompatible run-time options are specified. For more details, see the pcre2jit documentation. See also the jit? stack modifier below for a way of setting the size of the JIT stack.

If the jitfast modifier is specified, matching is done using the JIT "fast path" interface, pcre2_jit_match(), which skips some of the sanity checks that are done by pcre2_match(), and of course does not work when JIT is not supported. If jitfast is specified without jit, jit=7 is assumed.

If the jitverify modifier is specified, information about the compiled pattern shows whether JIT compilation was or was not successful. If jitverify is specified without jit, jit=7 is assumed. If JIT compilation is successful when jitverify is set, the text "(JIT)" is added to the first output line after a match or non match when JIT-compiled code was actually used in the match.

Setting a locale

The locale modifier must specify the name of a locale, for example:

/pattern/locale=fr_FR

The given locale is set, pcre2_maketables() is called to build a set of character tables for the locale, and this is then passed to pcre2_compile() when compiling

the regular expression. The same tables are used when matching the following sub?
ject lines. The locale modifier applies only to the pattern on which it appears,
but can be given in a #pattern command if a default is needed. Setting a locale and
alternate character tables are mutually exclusive.

Showing pattern memory

The memory modifier causes the size in bytes of the memory used to hold the com?
piled pattern to be output. This does not include the size of the pcre2_code block;
it is just the actual compiled data. If the pattern is subsequently passed to the
JIT compiler, the size of the JIT compiled code is also output. Here is an example:

```
re> /a(b)c/jit,memory
Memory allocation (code space): 21
Memory allocation (JIT code): 1910
```

Limiting nested parentheses

The parens_nest_limit modifier sets a limit on the depth of nested parentheses in a
pattern. Breaching the limit causes a compilation error. The default for the li?
brary is set when PCRE2 is built, but pcre2test sets its own default of 220, which
is required for running the standard test suite.

Limiting the pattern length

The max_pattern_length modifier sets a limit, in code units, to the length of pat?
tern that pcre2_compile() will accept. Breaching the limit causes a compilation er?
ror. The default is the largest number a PCRE2_SIZE variable can hold (essentially
unlimited).

Using the POSIX wrapper API

The posix and posix_nosub modifiers cause pcre2test to call PCRE2 via the POSIX
wrapper API rather than its native API. When posix_nosub is used, the POSIX option
REG_NOSUB is passed to regcomp(). The POSIX wrapper supports only the 8-bit li?
brary. Note that it does not imply POSIX matching semantics; for more detail see
the pcre2posix documentation. The following pattern modifiers set options for the
regcomp() function:

```
caseless        REG_ICASE
multiline       REG_NEWLINE
dotall          REG_DOTALL     )
ungreedy        REG_UNGREEDY   ) These options are not part of
```

ucp REG_UCP) the POSIX standard

utf REG_UTF8)

The regerror_buffsize modifier specifies a size for the error buffer that is passed to regerror() in the event of a compilation error. For example:

/abc/posix,regerror_buffsize=20

This provides a means of testing the behaviour of regerror() when the buffer is too small for the error message. If this modifier has not been set, a large buffer is used.

The aftertext and allaftertext subject modifiers work as described below. All other modifiers are either ignored, with a warning message, or cause an error.

The pattern is passed to regcomp() as a zero-terminated string by default, but if the use_length or hex modifiers are set, the REG_PEND extension is used to pass it by length.

Testing the stack guard feature

The stackguard modifier is used to test the use of pcre2_set_compile_recur? sion_guard(), a function that is provided to enable stack availability to be checked during compilation (see the pcre2api documentation for details). If the number specified by the modifier is greater than zero, pcre2_set_compile_recur? sion_guard() is called to set up callback from pcre2_compile() to a local function. The argument it receives is the current nesting parenthesis depth; if this is greater than the value given by the modifier, non-zero is returned, causing the compilation to be aborted.

Using alternative character tables

The value specified for the tables modifier must be one of the digits 0, 1, or 2. It causes a specific set of built-in character tables to be passed to pcre2_com? pile(). This is used in the PCRE2 tests to check behaviour with different character tables. The digit specifies the tables as follows:

0 do not pass any special character tables

1 the default ASCII tables, as distributed in pcre2_chartables.c.dist

2 a set of tables defining ISO 8859 characters

In table 2, some characters whose codes are greater than 128 are identified as let? ters, digits, spaces, etc. Setting alternate character tables and a locale are mu?

tually exclusive.

Setting certain match controls

The following modifiers are really subject modifiers, and are described under "Sub?ject Modifiers" below. However, they may be included in a pattern's modifier list, in which case they are applied to every subject line that is processed with that pattern. These modifiers do not affect the compilation process.

```
    aftertext                show text after match
    allaftertext             show text after captures
    allcaptures              show all captures
    allvector                show the entire ovector
    allusedtext              show all consulted text
    altglobal                alternative global matching
  /g  global                 global matching
    jitstack=<n>             set size of JIT stack
    mark                     show mark values
    replace=<string>         specify a replacement string
    startchar                show starting character when relevant
    substitute_callout       use substitution callouts
    substitute_extended      use PCRE2_SUBSTITUTE_EXTENDED
    substitute_skip=<n>      skip substitution number n
    substitute_overflow_length use PCRE2_SUBSTITUTE_OVERFLOW_LENGTH
    substitute_stop=<n>      skip substitution number n and greater
    substitute_unknown_unset   use PCRE2_SUBSTITUTE_UNKNOWN_UNSET
    substitute_unset_empty     use PCRE2_SUBSTITUTE_UNSET_EMPTY
```

These modifiers may not appear in a #pattern command. If you want them as defaults, set them in a #subject command.

Specifying literal subject lines

If the subject_literal modifier is present on a pattern, all the subject lines that it matches are taken as literal strings, with no interpretation of backslashes. It is not possible to set subject modifiers on such lines, but any that are set as de?faults by a #subject command are recognized.

Saving a compiled pattern

When a pattern with the push modifier is successfully compiled, it is pushed onto a

stack of compiled patterns, and pcre2test expects the next line to contain a new pattern (or a command) instead of a subject line. This facility is used when saving compiled patterns to a file, as described in the section entitled "Saving and restoring compiled patterns" below. If pushcopy is used instead of push, a copy of the compiled pattern is stacked, leaving the original as current, ready to match the following input lines. This provides a way of testing the pcre2_code_copy() function. The push and pushcopy modifiers are incompatible with compilation modi? fiers such as global that act at match time. Any that are specified are ignored (for the stacked copy), with a warning message, except for replace, which causes an error. Note that jitverify, which is allowed, does not carry through to any subse? quent matching that uses a stacked pattern.

Testing foreign pattern conversion

The experimental foreign pattern conversion functions in PCRE2 can be tested by setting the convert modifier. Its argument is a colon-separated list of options, which set the equivalent option for the pcre2_pattern_convert() function:

glob	PCRE2_CONVERT_GLOB
glob_no_starstar	PCRE2_CONVERT_GLOB_NO_STARSTAR
glob_no_wild_separator	PCRE2_CONVERT_GLOB_NO_WILD_SEPARATOR
posix_basic	PCRE2_CONVERT_POSIX_BASIC
posix_extended	PCRE2_CONVERT_POSIX_EXTENDED
unset	Unset all options

The "unset" value is useful for turning off a default that has been set by a #pat? tern command. When one of these options is set, the input pattern is passed to pcre2_pattern_convert(). If the conversion is successful, the result is reflected in the output and then passed to pcre2_compile(). The normal utf and no_utf_check options, if set, cause the PCRE2_CONVERT_UTF and PCRE2_CONVERT_NO_UTF_CHECK options to be passed to pcre2_pattern_convert().

By default, the conversion function is allowed to allocate a buffer for its output. However, if the convert_length modifier is set to a value greater than zero, pcre2test passes a buffer of the given length. This makes it possible to test the length check.

The convert_glob_escape and convert_glob_separator modifiers can be used to specify the escape and separator characters for glob processing, overriding the defaults,

which are operating-system dependent.

SUBJECT MODIFIERS

The modifiers that can appear in subject lines and the #subject command are of two types.

Setting match options

The following modifiers set options for pcre2_match() or pcre2_dfa_match(). See pcreapi for a description of their effects.

anchored set PCRE2_ANCHORED

endanchored set PCRE2_ENDANCHORED

dfa_restart set PCRE2_DFA_RESTART

dfa_shortest set PCRE2_DFA_SHORTEST

no_jit set PCRE2_NO_JIT

no_utf_check set PCRE2_NO_UTF_CHECK

notbol set PCRE2_NOTBOL

notempty set PCRE2_NOTEMPTY

notempty_atstart set PCRE2_NOTEMPTY_ATSTART

noteol set PCRE2_NOTEOL

partial_hard (or ph) set PCRE2_PARTIAL_HARD

partial_soft (or ps) set PCRE2_PARTIAL_SOFT

The partial matching modifiers are provided with abbreviations because they appear frequently in tests.

If the posix or posix_nosub modifier was present on the pattern, causing the POSIX wrapper API to be used, the only option-setting modifiers that have any effect are notbol, notempty, and noteol, causing REG_NOTBOL, REG_NOTEMPTY, and REG_NOTEOL, re? spectively, to be passed to regexec(). The other modifiers are ignored, with a warning message.

There is one additional modifier that can be used with the POSIX wrapper. It is ig? nored (with a warning) if used for non-POSIX matching.

posix_startend=<n>[:<m>]

This causes the subject string to be passed to regexec() using the REG_STARTEND op? tion, which uses offsets to specify which part of the string is searched. If only one number is given, the end offset is passed as the end of the subject string. For more detail of REG_STARTEND, see the pcre2posix documentation. If the subject

string contains binary zeros (coded as escapes such as \x{00} because pcre2test does not support actual binary zeros in its input), you must use posix_startend to specify its length.

Setting match controls

The following modifiers affect the matching process or request additional informa? tion. Some of them may also be specified on a pattern line (see above), in which case they apply to every subject line that is matched against that pattern.

aftertext show text after match

allaftertext show text after captures

allcaptures show all captures

allvector show the entire ovector

allusedtext show all consulted text (non-JIT only)

altglobal alternative global matching

callout_capture show captures at callout time

callout_data=<n> set a value to pass via callouts

callout_error=<n>[:<m>] control callout error

callout_extra show extra callout information

callout_fail=<n>[:<m>] control callout failure

callout_no_where do not show position of a callout

callout_none do not supply a callout function

copy=<number or name> copy captured substring

depth_limit=<n> set a depth limit

dfa use pcre2_dfa_match()

find_limits find match and depth limits

get=<number or name> extract captured substring

getall extract all captured substrings

/g global global matching

heap_limit=<n> set a limit on heap memory (Kbytes)

jitstack=<n> set size of JIT stack

mark show mark values

match_limit=<n> set a match limit

memory show heap memory usage

null_context match with a NULL context

offset=<n> set starting offset

offset_limit=<n> set offset limit

ovector=<n> set size of output vector

recursion_limit=<n> obsolete synonym for depth_limit

replace=<string> specify a replacement string

startchar show startchar when relevant

startoffset=<n> same as offset=<n>

substitute_callout use substitution callouts

substitute_extedded use PCRE2_SUBSTITUTE_EXTENDED

substitute_skip=<n> skip substitution number n

substitute_overflow_length use PCRE2_SUBSTITUTE_OVERFLOW_LENGTH

substitute_stop=<n> skip substitution number n and greater

substitute_unknown_unset use PCRE2_SUBSTITUTE_UNKNOWN_UNSET

substitute_unset_empty use PCRE2_SUBSTITUTE_UNSET_EMPTY

zero_terminate pass the subject as zero-terminated

The effects of these modifiers are described in the following sections. When match? ing via the POSIX wrapper API, the aftertext, allaftertext, and ovector subject modifiers work as described below. All other modifiers are either ignored, with a warning message, or cause an error.

Showing more text

The aftertext modifier requests that as well as outputting the part of the subject string that matched the entire pattern, pcre2test should in addition output the re? mainder of the subject string. This is useful for tests where the subject contains multiple copies of the same substring. The allaftertext modifier requests the same action for captured substrings as well as the main matched substring. In each case the remainder is output on the following line with a plus character following the capture number.

The allusedtext modifier requests that all the text that was consulted during a successful pattern match by the interpreter should be shown, for both full and par? tial matches. This feature is not supported for JIT matching, and if requested with JIT it is ignored (with a warning message). Setting this modifier affects the out? put if there is a lookbehind at the start of a match, or, for a complete match, a lookahead at the end, or if \K is used in the pattern. Characters that precede or

follow the start and end of the actual match are indicated in the output by '<' or '>' characters underneath them. Here is an example:

```
  re> /(?<=pqr)abc(?=xyz)/
data> 123pqrabcxyz456\=allusedtext
 0: pqrabcxyz
    <<<   >>>
data> 123pqrabcxy\=ph,allusedtext
Partial match: pqrabcxy
               <<<
```

The first, complete match shows that the matched string is "abc", with the preceding and following strings "pqr" and "xyz" having been consulted during the match (when processing the assertions). The partial match can indicate only the preceding string.

The startchar modifier requests that the starting character for the match be indicated, if it is different to the start of the matched string. The only time when this occurs is when \K has been processed as part of the match. In this situation, the output for the matched string is displayed from the starting character instead of from the match point, with circumflex characters under the earlier characters. For example:

```
  re> /abc\Kxyz/
data> abcxyz\=startchar
 0: abcxyz
    ^^^
```

Unlike allusedtext, the startchar modifier can be used with JIT. However, these two modifiers are mutually exclusive.

Showing the value of all capture groups

The allcaptures modifier requests that the values of all potential captured parentheses be output after a match. By default, only those up to the highest one actually used in the match are output (corresponding to the return code from pcre2_match()). Groups that did not take part in the match are output as "<unset>". This modifier is not relevant for DFA matching (which does no capturing) and does not apply when replace is specified; it is ignored, with a warning message, if present.

Showing the entire ovector, for all outcomes

The allvector modifier requests that the entire ovector be shown, whatever the out? come of the match. Compare allcaptures, which shows only up to the maximum number of capture groups for the pattern, and then only for a successful complete non-DFA match. This modifier, which acts after any match result, and also for DFA matching, provides a means of checking that there are no unexpected modifications to ovector fields. Before each match attempt, the ovector is filled with a special value, and if this is found in both elements of a capturing pair, "<unchanged>" is output. Af? ter a successful match, this applies to all groups after the maximum capture group for the pattern. In other cases it applies to the entire ovector. After a partial match, the first two elements are the only ones that should be set. After a DFA match, the amount of ovector that is used depends on the number of matches that were found.

Testing pattern callouts

A callout function is supplied when pcre2test calls the library matching functions, unless callout_none is specified. Its behaviour can be controlled by various modi? fiers listed above whose names begin with callout_. Details are given in the sec? tion entitled "Callouts" below. Testing callouts from pcre2_substitute() is de? cribed separately in "Testing the substitution function" below.

Finding all matches in a string

Searching for all possible matches within a subject can be requested by the global or altglobal modifier. After finding a match, the matching function is called again to search the remainder of the subject. The difference between global and altglobal is that the former uses the start_offset argument to pcre2_match() or pcre2_dfa_match() to start searching at a new point within the entire string (which is what Perl does), whereas the latter passes over a shortened subject. This makes a difference to the matching process if the pattern begins with a lookbehind asser? tion (including \b or \B).

If an empty string is matched, the next match is done with the PCRE2_NOTEMPTY_AT? START and PCRE2_ANCHORED flags set, in order to search for another, non-empty, match at the same point in the subject. If this match fails, the start offset is advanced, and the normal match is retried. This imitates the way Perl handles such cases when using the /g modifier or the split() function. Normally, the start off?

set is advanced by one character, but if the newline convention recognizes CRLF as a newline, and the current character is CR followed by LF, an advance of two characters occurs.

Testing substring extraction functions

The copy and get modifiers can be used to test the pcre2_substring_copy_xxx() and pcre2_substring_get_xxx() functions. They can be given more than once, and each can specify a capture group name or number, for example:

```
abcd\=copy=1,copy=3,get=G1
```

If the #subject command is used to set default copy and/or get lists, these can be unset by specifying a negative number to cancel all numbered groups and an empty name to cancel all named groups.

The getall modifier tests pcre2_substring_list_get(), which extracts all captured substrings.

If the subject line is successfully matched, the substrings extracted by the convenience functions are output with C, G, or L after the string number instead of a colon. This is in addition to the normal full list. The string length (that is, the return from the extraction function) is given in parentheses after each substring, followed by the name when the extraction was by name.

Testing the substitution function

If the replace modifier is set, the pcre2_substitute() function is called instead of one of the matching functions. Note that replacement strings cannot contain commas, because a comma signifies the end of a modifier. This is not thought to be an issue in a test program.

Unlike subject strings, pcre2test does not process replacement strings for escape sequences. In UTF mode, a replacement string is checked to see if it is a valid UTF-8 string. If so, it is correctly converted to a UTF string of the appropriate code unit width. If it is not a valid UTF-8 string, the individual code units are copied directly. This provides a means of passing an invalid UTF-8 string for testing purposes.

The following modifiers set options (in additional to the normal match options) for pcre2_substitute():

```
global                  PCRE2_SUBSTITUTE_GLOBAL
substitute_extended     PCRE2_SUBSTITUTE_EXTENDED
```

substitute_overflow_length PCRE2_SUBSTITUTE_OVERFLOW_LENGTH

substitute_unknown_unset PCRE2_SUBSTITUTE_UNKNOWN_UNSET

substitute_unset_empty PCRE2_SUBSTITUTE_UNSET_EMPTY

After a successful substitution, the modified string is output, preceded by the number of replacements. This may be zero if there were no matches. Here is a simple example of a substitution test:

```
/abc/replace=xxx
    =abc=abc=
 1: =xxx=abc=
    =abc=abc=\=global
 2: =xxx=xxx=
```

Subject and replacement strings should be kept relatively short (fewer than 256 characters) for substitution tests, as fixed-size buffers are used. To make it easy to test for buffer overflow, if the replacement string starts with a number in square brackets, that number is passed to pcre2_substitute() as the size of the output buffer, with the replacement string starting at the next character. Here is an example that tests the edge case:

```
/abc/
    123abc123\=replace=[10]XYZ
 1: 123XYZ123
    123abc123\=replace=[9]XYZ
Failed: error -47: no more memory
```

The default action of pcre2_substitute() is to return PCRE2_ERROR_NOMEMORY when the output buffer is too small. However, if the PCRE2_SUBSTITUTE_OVERFLOW_LENGTH option is set (by using the substitute_overflow_length modifier), pcre2_substitute() con? tinues to go through the motions of matching and substituting (but not doing any callouts), in order to compute the size of buffer that is required. When this hap? pens, pcre2test shows the required buffer length (which includes space for the trailing zero) as part of the error message. For example:

```
/abc/substitute_overflow_length
    123abc123\=replace=[9]XYZ
Failed: error -47: no more memory: 10 code units are needed
```

A replacement string is ignored with POSIX and DFA matching. Specifying partial

matching provokes an error return ("bad option value") from pcre2_substitute().

Testing substitute callouts

If the substitute_callout modifier is set, a substitution callout function is set up. The null_context modifier must not be set, because the address of the callout function is passed in a match context. When the callout function is called (after each substitution), details of the the input and output strings are output. For example:

```
  /abc/g,replace=<$0>,substitute_callout
      abcdefabcpqr
    1(1) Old 0 3 "abc" New 0 5 "<abc>"
    2(1) Old 6 9 "abc" New 8 13 "<abc>"
    2: <abc>def<abc>pqr
```

The first number on each callout line is the count of matches. The parenthesized number is the number of pairs that are set in the ovector (that is, one more than the number of capturing groups that were set). Then are listed the offsets of the old substring, its contents, and the same for the replacement.

By default, the substitution callout function returns zero, which accepts the replacement and causes matching to continue if /g was used. Two further modifiers can be used to test other return values. If substitute_skip is set to a value greater than zero the callout function returns +1 for the match of that number, and similarly substitute_stop returns -1. These cause the replacement to be rejected, and -1 causes no further matching to take place. If either of them are set, substitute_callout is assumed. For example:

```
  /abc/g,replace=<$0>,substitute_skip=1
      abcdefabcpqr
    1(1) Old 0 3 "abc" New 0 5 "<abc> SKIPPED"
    2(1) Old 6 9 "abc" New 6 11 "<abc>"
    2: abcdef<abc>pqr
      abcdefabcpqr\=substitute_stop=1
    1(1) Old 0 3 "abc" New 0 5 "<abc> STOPPED"
    1: abcdefabcpqr
```

If both are set for the same number, stop takes precedence. Only a single skip or stop is supported, which is sufficient for testing that the feature works.

Setting the JIT stack size

The jitstack modifier provides a way of setting the maximum stack size that is used by the just-in-time optimization code. It is ignored if JIT optimization is not be? ing used. The value is a number of kibibytes (units of 1024 bytes). Setting zero reverts to the default of 32KiB. Providing a stack that is larger than the default is necessary only for very complicated patterns. If jitstack is set non-zero on a subject line it overrides any value that was set on the pattern.

Setting heap, match, and depth limits

The heap_limit, match_limit, and depth_limit modifiers set the appropriate limits in the match context. These values are ignored when the find_limits modifier is specified.

Finding minimum limits

If the find_limits modifier is present on a subject line, pcre2test calls the rele? vant matching function several times, setting different values in the match context via pcre2_set_heap_limit(), pcre2_set_match_limit(), or pcre2_set_depth_limit() un? til it finds the minimum values for each parameter that allows the match to com? plete without error. If JIT is being used, only the match limit is relevant.

When using this modifier, the pattern should not contain any limit settings such as (*LIMIT_MATCH=...) within it. If such a setting is present and is lower than the minimum matching value, the minimum value cannot be found because pcre2_set_match_limit() etc. are only able to reduce the value of an in-pattern limit; they cannot increase it.

For non-DFA matching, the minimum depth_limit number is a measure of how much nested backtracking happens (that is, how deeply the pattern's tree is searched). In the case of DFA matching, depth_limit controls the depth of recursive calls of the internal function that is used for handling pattern recursion, lookaround as? sertions, and atomic groups.

For non-DFA matching, the match_limit number is a measure of the amount of back? tracking that takes place, and learning the minimum value can be instructive. For most simple matches, the number is quite small, but for patterns with very large numbers of matching possibilities, it can become large very quickly with increasing length of subject string. In the case of DFA matching, match_limit controls the to? tal number of calls, both recursive and non-recursive, to the internal matching

function, thus controlling the overall amount of computing resource that is used.

For both kinds of matching, the heap_limit number, which is in kibibytes (units of 1024 bytes), limits the amount of heap memory used for matching. A value of zero disables the use of any heap memory; many simple pattern matches can be done without using the heap, so zero is not an unreasonable setting.

Showing MARK names

The mark modifier causes the names from backtracking control verbs that are returned from calls to pcre2_match() to be displayed. If a mark is returned for a match, non-match, or partial match, pcre2test shows it. For a match, it is on a line by itself, tagged with "MK:". Otherwise, it is added to the non-match message.

Showing memory usage

The memory modifier causes pcre2test to log the sizes of all heap memory allocation and freeing calls that occur during a call to pcre2_match() or pcre2_dfa_match(). These occur only when a match requires a bigger vector than the default for remembering backtracking points (pcre2_match()) or for internal workspace (pcre2_dfa_match()). In many cases there will be no heap memory used and therefore no additional output. No heap memory is allocated during matching with JIT, so in that case the memory modifier never has any effect. For this modifier to work, the null_context modifier must not be set on both the pattern and the subject, though it can be set on one or the other.

Setting a starting offset

The offset modifier sets an offset in the subject string at which matching starts. Its value is a number of code units, not characters.

Setting an offset limit

The offset_limit modifier sets a limit for unanchored matches. If a match cannot be found starting at or before this offset in the subject, a "no match" return is given. The data value is a number of code units, not characters. When this modifier is used, the use_offset_limit modifier must have been set for the pattern; if not, an error is generated.

Setting the size of the output vector

The ovector modifier applies only to the subject line in which it appears, though of course it can also be used to set a default in a #subject command. It specifies the number of pairs of offsets that are available for storing matching information.

The default is 15.

A value of zero is useful when testing the POSIX API because it causes regexec() to be called with a NULL capture vector. When not testing the POSIX API, a value of zero is used to cause pcre2_match_data_create_from_pattern() to be called, in order to create a match block of exactly the right size for the pattern. (It is not possible to create a match block with a zero-length ovector; there is always at least one pair of offsets.)

Passing the subject as zero-terminated

By default, the subject string is passed to a native API matching function with its correct length. In order to test the facility for passing a zero-terminated string, the zero_terminate modifier is provided. It causes the length to be passed as PCRE2_ZERO_TERMINATED. When matching via the POSIX interface, this modifier is ignored, with a warning.

When testing pcre2_substitute(), this modifier also has the effect of passing the replacement string as zero-terminated.

Passing a NULL context

Normally, pcre2test passes a context block to pcre2_match(), pcre2_dfa_match(), pcre2_jit_match() or pcre2_substitute(). If the null_context modifier is set, however, NULL is passed. This is for testing that the matching and substitution functions behave correctly in this case (they use default values). This modifier cannot be used with the find_limits or substitute_callout modifiers.

THE ALTERNATIVE MATCHING FUNCTION

By default, pcre2test uses the standard PCRE2 matching function, pcre2_match() to match each subject line. PCRE2 also supports an alternative matching function, pcre2_dfa_match(), which operates in a different way, and has some restrictions. The differences between the two functions are described in the pcre2matching documentation.

If the dfa modifier is set, the alternative matching function is used. This function finds all possible matches at a given point in the subject. If, however, the dfa_shortest modifier is set, processing stops after the first match is found. This is always the shortest possible match.

DEFAULT OUTPUT FROM pcre2test

This section describes the output when the normal matching function, pcre2_match(),

is being used.

When a match succeeds, pcre2test outputs the list of captured substrings, starting with number 0 for the string that matched the whole pattern. Otherwise, it outputs "No match" when the return is PCRE2_ERROR_NOMATCH, or "Partial match:" followed by the partially matching substring when the return is PCRE2_ERROR_PARTIAL. (Note that this is the entire substring that was inspected during the partial match; it may include characters before the actual match start if a lookbehind assertion, \K, \b, or \B was involved.)

For any other return, pcre2test outputs the PCRE2 negative error number and a short descriptive phrase. If the error is a failed UTF string check, the code unit offset of the start of the failing character is also output. Here is an example of an in? teractive pcre2test run.

```
  $ pcre2test
  PCRE2 version 10.22 2016-07-29
    re> /^abc(\d+)/
  data> abc123
   0: abc123
   1: 123
  data> xyz
  No match
```

Unset capturing substrings that are not followed by one that is set are not shown by pcre2test unless the allcaptures modifier is specified. In the following exam? ple, there are two capturing substrings, but when the first data line is matched, the second, unset substring is not shown. An "internal" unset substring is shown as "<unset>", as for the second data line.

```
    re> /(a)|(b)/
  data> a
   0: a
   1: a
  data> b
   0: b
   1: <unset>
   2: b
```

If the strings contain any non-printing characters, they are output as \xhh escapes if the value is less than 256 and UTF mode is not set. Otherwise they are output as \x{hh...} escapes. See below for the definition of non-printing characters. If the aftertext modifier is set, the output for substring 0 is followed by the the rest of the subject string, identified by "0+" like this:

```
    re> /cat/aftertext
  data> cataract
   0: cat
   0+ aract
```

If global matching is requested, the results of successive matching attempts are output in sequence, like this:

```
    re> /\Bi(\w\w)/g
  data> Mississippi
   0: iss
   1: ss
   0: iss
   1: ss
   0: ipp
   1: pp
```

"No match" is output only if the first match attempt fails. Here is an example of a failure message (the offset 4 that is specified by the offset modifier is past the end of the subject string):

```
    re> /xyz/
  data> xyz\=offset=4
  Error -24 (bad offset value)
```

Note that whereas patterns can be continued over several lines (a plain ">" prompt is used for continuations), subject lines may not. However newlines can be included in a subject by means of the \n escape (or \r, \r\n, etc., depending on the newline sequence setting).

OUTPUT FROM THE ALTERNATIVE MATCHING FUNCTION

When the alternative matching function, pcre2_dfa_match(), is used, the output consists of a list of all the matches that start at the first point in the subject where there is at least one match. For example:

```
re> /(tang|tangerine|tan)/
data> yellow tangerine\=dfa
 0: tangerine
 1: tang
 2: tan
```

Using the normal matching function on this data finds only "tang". The longest matching string is always given first (and numbered zero). After a PCRE2_ERROR_PAR? TIAL return, the output is "Partial match:", followed by the partially matching substring. Note that this is the entire substring that was inspected during the partial match; it may include characters before the actual match start if a lookbe? hind assertion, \b, or \B was involved. (\K is not supported for DFA matching.)

If global matching is requested, the search for further matches resumes at the end of the longest match. For example:

```
re> /(tang|tangerine|tan)/g
data> yellow tangerine and tangy sultana\=dfa
 0: tangerine
 1: tang
 2: tan
 0: tang
 1: tan
 0: tan
```

The alternative matching function does not support substring capture, so the modi? fiers that are concerned with captured substrings are not relevant.

RESTARTING AFTER A PARTIAL MATCH

When the alternative matching function has given the PCRE2_ERROR_PARTIAL return, indicating that the subject partially matched the pattern, you can restart the match with additional subject data by means of the dfa_restart modifier. For exam? ple:

```
re> /^\d?\d(jan|feb|mar|apr|may|jun|jul|aug|sep|oct|nov|dec)\d\d$/
data> 23ja\=ps,dfa
Partial match: 23ja
data> n05\=dfa,dfa_restart
 0: n05
```

For further information about partial matching, see the pcre2partial documentation.

CALLOUTS

If the pattern contains any callout requests, pcre2test's callout function is called during matching unless callout_none is specified. This works with both matching functions, and with JIT, though there are some differences in behaviour. The output for callouts with numerical arguments and those with string arguments is slightly different.

Callouts with numerical arguments

By default, the callout function displays the callout number, the start and current positions in the subject text at the callout time, and the next pattern item to be tested. For example:

```
  --->pqrabcdef
    0    ^  ^     \d
```

This output indicates that callout number 0 occurred for a match attempt starting at the fourth character of the subject string, when the pointer was at the seventh character, and when the next pattern item was \d. Just one circumflex is output if the start and current positions are the same, or if the current position precedes the start position, which can happen if the callout is in a lookbehind assertion.

Callouts numbered 255 are assumed to be automatic callouts, inserted as a result of the auto_callout pattern modifier. In this case, instead of showing the callout number, the offset in the pattern, preceded by a plus, is output. For example:

```
    re> /\d?[A-E]\*/auto_callout
  data> E*
  --->E*
   +0 ^      \d?
   +3 ^      [A-E]
   +8 ^^     \*
  +10 ^ ^
   0: E*
```

If a pattern contains (*MARK) items, an additional line is output whenever a change of latest mark is passed to the callout function. For example:

```
    re> /a(*MARK:X)bc/auto_callout
  data> abc
```

```
--->abc
 +0 ^       a
 +1 ^^      (*MARK:X)
+10 ^^      b
Latest Mark: X
+11 ^ ^     c
+12 ^  ^
 0: abc
```

The mark changes between matching "a" and "b", but stays the same for the rest of the match, so nothing more is output. If, as a result of backtracking, the mark re? verts to being unset, the text "<unset>" is output.

Callouts with string arguments

The output for a callout with a string argument is similar, except that instead of outputting a callout number before the position indicators, the callout string and its offset in the pattern string are output before the reflection of the subject string, and the subject string is reflected for each callout. For example:

```
  re> /^ab(?C'first')cd(?C"second")ef/
data> abcdefg
Callout (7): 'first'
--->abcdefg
    ^ ^        c
Callout (20): "second"
--->abcdefg
    ^   ^      e
 0: abcdef
```

Callout modifiers

The callout function in pcre2test returns zero (carry on matching) by default, but you can use a callout_fail modifier in a subject line to change this and other pa? rameters of the callout (see below).

If the callout_capture modifier is set, the current captured groups are output when a callout occurs. This is useful only for non-DFA matching, as pcre2_dfa_match() does not support capturing, so no captures are ever shown.

The normal callout output, showing the callout number or pattern offset (as de?

scribed above) is suppressed if the callout_no_where modifier is set.

When using the interpretive matching function pcre2_match() without JIT, setting the callout_extra modifier causes additional output from pcre2test's callout func? tion to be generated. For the first callout in a match attempt at a new starting position in the subject, "New match attempt" is output. If there has been a back? track since the last callout (or start of matching if this is the first callout), "Backtrack" is output, followed by "No other matching paths" if the backtrack ended the previous match attempt. For example:

```
  re> /(a+)b/auto_callout,no_start_optimize,no_auto_possess
 data> aac\=callout_extra
 New match attempt
 --->aac
  +0 ^      (
  +1 ^      a+
  +3 ^ ^    )
  +4 ^ ^    b
 Backtrack
 --->aac
  +3 ^^     )
  +4 ^^     b
 Backtrack
 No other matching paths
 New match attempt
 --->aac
  +0  ^     (
  +1  ^     a+
  +3  ^^    )
  +4  ^^    b
 Backtrack
 No other matching paths
 New match attempt
 --->aac
  +0  ^     (
```

```
+1  ^    a+
Backtrack
No other matching paths
New match attempt
--->aac
+0  ^   (
+1  ^   a+
No match
```

Notice that various optimizations must be turned off if you want all possible matching paths to be scanned. If no_start_optimize is not used, there is an immedi? ate "no match", without any callouts, because the starting optimization fails to find "b" in the subject, which it knows must be present for any match. If no_auto_possess is not used, the "a+" item is turned into "a++", which reduces the number of backtracks.

The callout_extra modifier has no effect if used with the DFA matching function, or with JIT.

Return values from callouts

The default return from the callout function is zero, which allows matching to con? tinue. The callout_fail modifier can be given one or two numbers. If there is only one number, 1 is returned instead of 0 (causing matching to backtrack) when a call? out of that number is reached. If two numbers (<n>:<m>) are given, 1 is returned when callout <n> is reached and there have been at least <m> callouts. The call? out_error modifier is similar, except that PCRE2_ERROR_CALLOUT is returned, causing the entire matching process to be aborted. If both these modifiers are set for the same callout number, callout_error takes precedence. Note that callouts with string arguments are always given the number zero.

The callout_data modifier can be given an unsigned or a negative number. This is set as the "user data" that is passed to the matching function, and passed back when the callout function is invoked. Any value other than zero is used as a return from pcre2test's callout function.

Inserting callouts can be helpful when using pcre2test to check complicated regular expressions. For further information about callouts, see the pcre2callout documen? tation.

NON-PRINTING CHARACTERS

When pcre2test is outputting text in the compiled version of a pattern, bytes other than 32-126 are always treated as non-printing characters and are therefore shown as hex escapes.

When pcre2test is outputting text that is a matched part of a subject string, it behaves in the same way, unless a different locale has been set for the pattern (using the locale modifier). In this case, the isprint() function is used to dis? tinguish printing and non-printing characters.

SAVING AND RESTORING COMPILED PATTERNS

It is possible to save compiled patterns on disc or elsewhere, and reload them later, subject to a number of restrictions. JIT data cannot be saved. The host on which the patterns are reloaded must be running the same version of PCRE2, with the same code unit width, and must also have the same endianness, pointer width and PCRE2_SIZE type. Before compiled patterns can be saved they must be serialized, that is, converted to a stream of bytes. A single byte stream may contain any num? ber of compiled patterns, but they must all use the same character tables. A single copy of the tables is included in the byte stream (its size is 1088 bytes).

The functions whose names begin with pcre2_serialize_ are used for serializing and de-serializing. They are described in the pcre2serialize documentation. In this section we describe the features of pcre2test that can be used to test these func? tions.

Note that "serialization" in PCRE2 does not convert compiled patterns to an ab? stract format like Java or .NET. It just makes a reloadable byte code stream. Hence the restrictions on reloading mentioned above.

In pcre2test, when a pattern with push modifier is successfully compiled, it is pushed onto a stack of compiled patterns, and pcre2test expects the next line to contain a new pattern (or command) instead of a subject line. By contrast, the pushcopy modifier causes a copy of the compiled pattern to be stacked, leaving the original available for immediate matching. By using push and/or pushcopy, a number of patterns can be compiled and retained. These modifiers are incompatible with posix, and control modifiers that act at match time are ignored (with a message) for the stacked patterns. The jitverify modifier applies only at compile time.

The command

#save <filename>

causes all the stacked patterns to be serialized and the result written to the named file. Afterwards, all the stacked patterns are freed. The command

#load <filename>

reads the data in the file, and then arranges for it to be de-serialized, with the resulting compiled patterns added to the pattern stack. The pattern on the top of the stack can be retrieved by the #pop command, which must be followed by lines of subjects that are to be matched with the pattern, terminated as usual by an empty line or end of file. This command may be followed by a modifier list containing only control modifiers that act after a pattern has been compiled. In particular, hex, posix, posix_nosub, push, and pushcopy are not allowed, nor are any option-setting modifiers. The JIT modifiers are, however permitted. Here is an example that saves and reloads two patterns.

```
/abc/push
/xyz/push
#save tempfile
#load tempfile
#pop info
xyz
#pop jit,bincode
abc
```

If jitverify is used with #pop, it does not automatically imply jit, which is dif? ferent behaviour from when it is used on a pattern.

The #popcopy command is analagous to the pushcopy modifier in that it makes current a copy of the topmost stack pattern, leaving the original still on the stack.

SEE ALSO

pcre2(3), pcre2api(3), pcre2callout(3), pcre2jit, pcre2matching(3), pcre2par? tial(d), pcre2pattern(3), pcre2serialize(3).

AUTHOR

Philip Hazel

University Computing Service

Cambridge, England.

REVISION

Last updated: 30 July 2019