NAME

PCRE2 - Perl-compatible regular expressions (revised API)

SYNOPSIS

#include <pcre2.h>

int (*pcre2_callout)(pcre2_callout_block *, void *);

int pcre2_callout_enumerate(const pcre2_code *_code_,
** int (*_callback_)(pcre2_callout_enumerate_block *, void *),**
** void *_user_data_);**

DESCRIPTION

PCRE2 provides a feature called "callout", which is a means of temporarily passing control to the caller of PCRE2 in the middle of pattern matching. The caller of PCRE2 provides an external function by putting its entry point in a match context (see **pcre2_set_callout()** in the **pcre2api** documentation).

When using the **pcre2_substitute()** function, an additional callout feature is available. This does a callout after each change to the subject string and is described in the **pcre2api** documentation; the rest of this document is concerned with callouts during pattern matching.

Within a regular expression, (?C<arg>) indicates a point at which the external function is to be called. Different callout points can be identified by putting a number less than 256 after the letter C. The default value is zero. Alternatively, the argument may be a delimited string. The starting delimiter must be one of ‘ ’ " ^ % # $ { and the ending delimiter is the same as the start, except for {, where the ending delimiter is }. If the ending delimiter is needed within the string, it must be doubled. For example, this pattern has two callout points:

```
(?C1)abc(?C"some ""arbitrary"" text")def
```

If the PCRE2_AUTO_CALLOUT option bit is set when a pattern is compiled, PCRE2 automatically inserts callouts, all with number 255, before each item in the pattern except for immediately before or after an explicit callout. For example, if PCRE2_AUTO_CALLOUT is used with the pattern

```
A(?C3)B
```

it is processed as if it were

```
(?C255)A(?C3)B(?C255)
```

Here is a more complicated example:

```
A(\d{2}|--)
```

With PCRE2_AUTO_CALLOUT, this pattern is processed as if it were

```
(?C255)A(?C255)((?C255)\d{2}(?C255)|(?C255)-(?C255)-(?C255))(?C255)
```

Notice that there is a callout before and after each parenthesis and alternation bar. If the pattern contains a conditional group whose condition is an assertion, an automatic callout is inserted immediately before the condition. Such a callout may also be inserted explicitly, for example:

```
(?(?C9)(?=a)ab|de)  (?(?C%text%)(?!=d)ab|de)
```

This applies only to assertion conditions (because they are themselves independent groups).

Callouts can be useful for tracking the progress of pattern matching. The **pcre2test** program has a pattern qualifier (/auto_callout) that sets automatic callouts. When any callouts are present, the output from **pcre2test** indicates how the pattern is being matched. This is useful information when you are trying to optimize the performance of a particular pattern.

MISSING CALLOUTS

You should be aware that, because of optimizations in the way PCRE2 compiles and matches patterns, callouts sometimes do not happen exactly as you might expect.

Auto-possessification

At compile time, PCRE2 "auto-possessifies" repeated items when it knows that what follows cannot be part of the repeat. For example, a+[bc] is compiled as if it were a++[bc]. The **pcre2test** output when this pattern is compiled with PCRE2_ANCHORED and PCRE2_AUTO_CALLOUT and then applied to the string "aaaa" is:

```
--->aaaa
 +0 ^       a+
 +2 ^  ^    [bc]
No match
```

This indicates that when matching [bc] fails, there is no backtracking into a+ (because it is being treated as a++) and therefore the callouts that would be taken for the backtracks do not occur. You can disable the auto-possessify feature by passing PCRE2_NO_AUTO_POSSESS to **pcre2_compile()**, or starting the pattern with (*NO_AUTO_POSSESS). In this case, the output changes to this:

```
--->aaaa
 +0 ^       a+
 +2 ^  ^    [bc]
 +2 ^ ^     [bc]
 +2 ^^      [bc]
 +2 ^^      [bc]
No match
```

This time, when matching [bc] fails, the matcher backtracks into a+ and tries again, repeatedly, until a+ itself fails.

Automatic .* anchoring

By default, an optimization is applied when .* is the first significant item in a pattern. If PCRE2_DOTALL is set, so that the dot can match any character, the pattern is automatically anchored. If PCRE2_DOTALL is not set, a match can start only after an internal newline or at the beginning of the subject, and **pcre2_compile()** remembers this. If a pattern has more than one top-level branch, automatic anchoring occurs if all branches are anchorable.

This optimization is disabled, however, if .* is in an atomic group or if there is a backreference to the capture group in which it appears. It is also disabled if the pattern contains (*PRUNE) or (*SKIP). However, the presence of callouts does not affect it.

For example, if the pattern .*\d is compiled with PCRE2_AUTO_CALLOUT and applied to the string "aa", the **pcre2test** output is:

```
--->aa
 +0 ^      .*
 +2 ^^     \d
 +2 ^^     \d
```

```
 +2 ^      \d
No match
```

This shows that all match attempts start at the beginning of the subject. In other words, the pattern is anchored. You can disable this optimization by passing PCRE2_NO_DOTSTAR_ANCHOR to **pcre2_compile()**, or starting the pattern with (*NO_DOTSTAR_ANCHOR). In this case, the output changes to:

```
--->aa
 +0 ^      .*
 +2 ^ ^    \d
 +2 ^^     \d
 +2 ^      \d
 +0  ^     .*
 +2  ^^    \d
 +2  ^     \d
No match
```

This shows more match attempts, starting at the second subject character. Another optimization, described in the next section, means that there is no subsequent attempt to match with an empty subject.

Other optimizations

Other optimizations that provide fast "no match" results also affect callouts. For example, if the pattern is

```
ab(?C4)cd
```

PCRE2 knows that any matching string must contain the letter "d". If the subject string is "abyz", the lack of "d" means that matching doesn't ever start, and the callout is never reached. However, with "abyd", though the result is still no match, the callout is obeyed.

For most patterns PCRE2 also knows the minimum length of a matching string, and will immediately give a "no match" return without actually running a match if the subject is not long enough, or, for unanchored patterns, if it has been scanned far enough.

You can disable these optimizations by passing the PCRE2_NO_START_OPTIMIZE option to **pcre2_compile()**, or by starting the pattern with (*NO_START_OPT). This slows down the matching process, but does ensure that callouts such as the example above are obeyed.

THE CALLOUT INTERFACE

During matching, when PCRE2 reaches a callout point, if an external function is provided in the match context, it is called. This applies to both normal, DFA, and JIT matching. The first argument to the callout function is a pointer to a **pcre2_callout** block. The second argument is the void * callout data that was supplied when the callout was set up by calling **pcre2_set_callout()** (see the **pcre2api** documentation). The callout block structure contains the following fields, not necessarily in this order:

```
uint32_t      version;
uint32_t      callout_number;
uint32_t      capture_top;
uint32_t      capture_last;
uint32_t      callout_flags;
PCRE2_SIZE   *offset_vector;
PCRE2_SPTR    mark;
PCRE2_SPTR    subject;
PCRE2_SIZE    subject_length;
PCRE2_SIZE    start_match;
PCRE2_SIZE    current_position;
```

```
PCRE2_SIZE   pattern_position;
PCRE2_SIZE   next_item_length;
PCRE2_SIZE   callout_string_offset;
PCRE2_SIZE   callout_string_length;
PCRE2_SPTR   callout_string;
```

The *version* field contains the version number of the block format. The current version is 2; the three callout string fields were added for version 1, and the *callout_flags* field for version 2. If you are writing an application that might use an earlier release of PCRE2, you should check the version number before accessing any of these fields. The version number will increase in future if more fields are added, but the intention is never to remove any of the existing fields.

Fields for numerical callouts

For a numerical callout, *callout_string* is NULL, and *callout_number* contains the number of the callout, in the range 0-255. This is the number that follows (?C for callouts that part of the pattern; it is 255 for automatically generated callouts.

Fields for string callouts

For callouts with string arguments, *callout_number* is always zero, and *callout_string* points to the string that is contained within the compiled pattern. Its length is given by *callout_string_length*. Duplicated ending delimiters that were present in the original pattern string have been turned into single characters, but there is no other processing of the callout string argument. An additional code unit containing binary zero is present after the string, but is not included in the length. The delimiter that was used to start the string is also stored within the pattern, immediately before the string itself. You can access this delimiter as *callout_string*[-1] if you need it.

The *callout_string_offset* field is the code unit offset to the start of the callout argument string within the original pattern string. This is provided for the benefit of applications such as script languages that might need to report errors in the callout string within the pattern.

Fields for all callouts

The remaining fields in the callout block are the same for both kinds of callout.

The *offset_vector* field is a pointer to a vector of capturing offsets (the "ovector"). You may read the elements in this vector, but you must not change any of them.

For calls to **pcre2_match()**, the *offset_vector* field is not (since release 10.30) a pointer to the actual ovector that was passed to the matching function in the match data block. Instead it points to an internal ovector of a size large enough to hold all possible captured substrings in the pattern. Note that whenever a recursion or subroutine call within a pattern completes, the capturing state is reset to what it was before.

The *capture_last* field contains the number of the most recently captured substring, and the *capture_top* field contains one more than the number of the highest numbered captured substring so far. If no substrings have yet been captured, the value of *capture_last* is 0 and the value of *capture_top* is 1. The values of these fields do not always differ by one; for example, when the callout in the pattern ((a)(b))(?C2) is taken, *capture_last* is 1 but *capture_top* is 4.

The contents of ovector[2] to ovector[<capture_top>*2-1] can be inspected in order to extract substrings that have been matched so far, in the same way as extracting substrings after a match has completed. The values in ovector[0] and ovector[1] are always PCRE2_UNSET because the match is by definition not complete. Substrings that have not been captured but whose numbers are less than *capture_top* also have both of their ovector slots set to PCRE2_UNSET.

For DFA matching, the *offset_vector* field points to the ovector that was passed to the matching function in the match data block for callouts at the top level, but to an internal ovector during the processing of pattern recursions, lookarounds, and atomic groups. However, these ovectors hold no useful information because **pcre2_dfa_match()** does not support substring capturing. The value of *capture_top* is always 1 and the

value of *capture_last* is always 0 for DFA matching.

The *subject* and *subject_length* fields contain copies of the values that were passed to the matching function.

The *start_match* field normally contains the offset within the subject at which the current match attempt started. However, if the escape sequence \K has been encountered, this value is changed to reflect the modified starting point. If the pattern is not anchored, the callout function may be called several times from the same point in the pattern for different starting points in the subject.

The *current_position* field contains the offset within the subject of the current match pointer.

The *pattern_position* field contains the offset in the pattern string to the next item to be matched.

The *next_item_length* field contains the length of the next item to be processed in the pattern string. When the callout is at the end of the pattern, the length is zero. When the callout precedes an opening parenthesis, the length includes meta characters that follow the parenthesis. For example, in a callout before an assertion such as (?=ab) the length is 3. For an an alternation bar or a closing parenthesis, the length is one, unless a closing parenthesis is followed by a quantifier, in which case its length is included. (This changed in release 10.23. In earlier releases, before an opening parenthesis the length was that of the entire group, and before an alternation bar or a closing parenthesis the length was zero.)

The *pattern_position* and *next_item_length* fields are intended to help in distinguishing between different automatic callouts, which all have the same callout number. However, they are set for all callouts, and are used by **pcre2test** to show the next item to be matched when displaying callout information.

In callouts from **pcre2_match()** the *mark* field contains a pointer to the zero-terminated name of the most recently passed (*MARK), (*PRUNE), or (*THEN) item in the match, or NULL if no such items have been passed. Instances of (*PRUNE) or (*THEN) without a name do not obliterate a previous (*MARK). In callouts from the DFA matching function this field always contains NULL.

The *callout_flags* field is always zero in callouts from **pcre2_dfa_match()** or when JIT is being used. When **pcre2_match()** without JIT is used, the following bits may be set:

PCRE2_CALLOUT_STARTMATCH

This is set for the first callout after the start of matching for each new starting position in the subject.

PCRE2_CALLOUT_BACKTRACK

This is set if there has been a matching backtrack since the previous callout, or since the start of matching if this is the first callout from a **pcre2_match()** run.

Both bits are set when a backtrack has caused a "bumpalong" to a new starting position in the subject. Output from **pcre2test** does not indicate the presence of these bits unless the **callout_extra** modifier is set.

The information in the **callout_flags** field is provided so that applications can track and tell their users how matching with backtracking is done. This can be useful when trying to optimize patterns, or just to understand how PCRE2 works. There is no support in **pcre2_dfa_match()** because there is no backtracking in DFA matching, and there is no support in JIT because JIT is all about maximimizing matching performance. In both these cases the **callout_flags** field is always zero.

RETURN VALUES FROM CALLOUTS

The external callout function returns an integer to PCRE2. If the value is zero, matching proceeds as normal. If the value is greater than zero, matching fails at the current point, but the testing of other matching possibilities goes ahead, just as if a lookahead assertion had failed. If the value is less than zero, the match is abandoned, and the matching function returns the negative value.

Negative values should normally be chosen from the set of PCRE2_ERROR_xxx values. In particular, PCRE2_ERROR_NOMATCH forces a standard "no match" failure. The error number PCRE2_ERROR_CALLOUT is reserved for use by callout functions; it will never be used by PCRE2 itself.

CALLOUT ENUMERATION

int pcre2_callout_enumerate(const pcre2_code ****code*,**
int (**callback***)(pcre2_callout_enumerate_block *, void *),**
void ****user_data*);**

A script language that supports the use of string arguments in callouts might like to scan all the callouts in a pattern before running the match. This can be done by calling **pcre2_callout_enumerate()**. The first argument is a pointer to a compiled pattern, the second points to a callback function, and the third is arbitrary user data. The callback function is called for every callout in the pattern in the order in which they appear. Its first argument is a pointer to a callout enumeration block, and its second argument is the *user_data* value that was passed to **pcre2_callout_enumerate()**. The data block contains the following fields:

version	Block version number
pattern_position	Offset to next item in pattern
next_item_length	Length of next item in pattern
callout_number	Number for numbered callouts
callout_string_offset	Offset to string within pattern
callout_string_length	Length of callout string
callout_string	Points to callout string or is NULL

The version number is currently 0. It will increase if new fields are ever added to the block. The remaining fields are the same as their namesakes in the **pcre2_callout** block that is used for callouts during matching, as described above.

Note that the value of *pattern_position* is unique for each callout. However, if a callout occurs inside a group that is quantified with a non-zero minimum or a fixed maximum, the group is replicated inside the compiled pattern. For example, a pattern such as /(a){2}/ is compiled as if it were /(a)(a)/. This means that the callout will be enumerated more than once, but with the same value for *pattern_position* in each case.

The callback function should normally return zero. If it returns a non-zero value, scanning the pattern stops, and that value is returned from **pcre2_callout_enumerate()**.

AUTHOR

Philip Hazel
University Computing Service
Cambridge, England.

REVISION

Last updated: 03 February 2019