A New API for PCRE

Revision:	4
Updated:	27 August 2013
Author:	Philip Hazel

This document contains a proposal for a completely new API for PCRE. Changes from revision to revision of this document will be marked by vertical bars on the right, like this. Some familiarity with the old API is assumed because I haven't fully described what all the various functions and options actually do, especially when they are unchanged from the old API.

This new API does not have any user-visible C structures, except for *pcre2_callout_block*. Instead, function calls are used as the means of interacting with the library. This makes it easier to interface the library to languages other than C and C++ that cannot access C structure definitions or C macros.

1 Names and numbers

The new API will be introduced for release 9.0. In order to avoid confusion, especially when both APIs are simultaneously installed, the new API uses different names for functions, options, structures, and header files. All the new names begin with `pcre2` or `PCRE2`.

The libraries are called *libpcre2-8*, *libpcre2-16*, and *libpcre2-32*, so that both old and new libraries may exist together. The names of the man pages also begin with `pcre2` for the same reason. However, the names of the *libpcreposix* library and the *pcregrep* and *pcretest* commands are not changed, so installing PCRE 9.0 overwrites any previous versions.

2 Handling different data widths

Every function comes in three different forms, for example:

```
pcre2_compile_8()
pcre2_compile_16()
pcre2_compile_32()
```

There are also three different sets of data types:

```
PCRE2_UCHAR8, PCRE2_UCHAR16, PCRE2_UCHAR32
PCRE2_SPTR8,  PCRE2_SPTR16,  PCRE2_SPTR32
```

The `UCHAR` types define unsigned data items of the appropriate widths. For example, `PCRE2_UCHAR16` is usually defined as 'unsigned short'. The `SPTR` types are constant pointers to the equivalent `UCHAR` types, that is, they are pointers to strings of unsigned data items.

Many applications use only one data width. For their convenience, macros are defined whose names are the generic forms such as *pcre2_compile()* and `PCRE2_UCHAR`. These macros use the value of the macro `PCRE2_DATA_WIDTH` to generate the appropriate width-specific function and macro names. `PCRE2_DATA_WIDTH` is not defined by default.

Applications that use more than one data width are advised not to define `PCRE2_DATA_WIDTH`, but instead to use the real function names, and any code that is to be included in an environment where the value of `PCRE2_DATA_WIDTH` is unknown should do likewise. (Unfortunately, it is not possible in C code to save and restore the value of a macro.)

In the rest of this document, functions and data types are described using their generic names, without the 8, 16, or 32 suffix. This can also be done in the user documentation for the new API.

3 Data blocks and multithreading

In a multithreaded application it is important to keep thread-specific data separate from data that can be shared between threads. The library code itself is thread-safe: it contains no static or global variables. The API is designed to be fairly simple for non-threaded applications while at the same time ensuring that multithreaded applications can use it.

There are several different blocks of data that are used to pass information between the application the the PCRE libraries.

- A pointer to the compiled form of a pattern is returned to the user when *pcre2_compile()* is successful. The data in the compiled pattern is fixed, and does not change when the pattern is matched. Therefore, it is thread-safe, so the same compiled pattern can be used by more than one thread simultaneously. An application can compile all its patterns at the start, before forking off multiple threads that use them.
- The new API introduces the idea of a *context* in which PCRE functions are called. A context is nothing more than a collection of parameters that control the way PCRE operates. Grouping them together in the context is a convenient way of passing them to PCRE functions without using lots of arguments. The same context can be used for processing many different patterns or the same pattern several times.

  Some of what were function options in the previous API have been moved into the context, because they are expected to be overall settings for an application, and are not likely to change from pattern to pattern (though they can be changed if necessary). Options that are likely to differ from pattern to pattern are passed as arguments to the functions, as before.

  In a multithreaded application, if the parameters in a context are indeed values that are never changed, there can be a single context that is used by all the threads. However, if any thread needs to change any value in the context, it must make its own thread-specific copy.
- The matching functions need a block of memory for working space and for storing the results of a match. This includes details of what was matched, as well as additional information such as the name of a `(*MARK)` setting. Each thread must provide its own version of this memory.

4 A simple example

In the old API, a straightforward use of PCRE, including studying the compiled pattern, looks like this:

```
pcre *re;
pcre_extra *extra;
const char *error;
int ovector[30];
int erroffset;
int rc;
re = pcre_compile("pattern", 0, &error, &erroffset, NULL);
if (re == NULL)
  {
  /* Handle error */
  }
extra = pcre_study(re, 0, &error);
if (error != NULL)
  {
  /* Handle error */
  }
rc = pcre_exec(re, extra, "subject", 7, 0, 0, ovector, 30);
if (rc < 0)
  {
  /* Handle error */
  }
pcre_free(re);
pcre_free_study(extra);
```

In the new API there is more set-up and take-down work to be done, and error handling is different in order to accommodate 16- and 32-bit error messages, but studying is automatic.

```
#define PCRE2_DATA_WIDTH 8   /* or 16 or 32 */
pcre2 *re;
```

```
  pcre2_context *context;
  pcre2_match_data *match_data;
  size_t &erroroffset;
  size_t ovector[20];
  int errorcode;
  int rc;
  context = pcre2_init_context(NULL);
  re = pcre2_compile(context, "pattern", 0, &errorcode, &erroffset);
  if (re == NULL)
    {
    PCRE_UCHAR buffer[120];
    (void)pcre2_get_error_message(errorcode, buffer, 120);
    /* Handle error */
    }
  match_data = pcre2_create_match_data(context, ovector, 20);
  rc = pcre2_exec(context, re, "subject", -1, 0, 0, match_data);
  if (rc < 0)
    {
    /* Handle error */
    }
  pcre2_free(re);
  pcre2_free_context(context);
  pcre2_free_match_data(match_data);
```

5 Managing a context

Several functions are provided for creating a context and managing its contents.

5.1 Creating a context

Applications that do not do their own memory management can create a context very easily:

```
  pcre2_context *context = pcre2_init_context(NULL);
```

When its argument is NULL, *pcre2_init_context()* uses *malloc()* to get a block of memory in which to store the context. Applications that do have their own memory management functions can set up a context like this:

```
  size_t context_size = pcre2_context_size();
  pcre2_context *context = private_malloc(context_size, ...);
  (void)pcre2_init_context(context);
```

A call to *pcre2_init_context()* sets default values for all the context parameters.

5.2 Setting callback data in a context

An application may specify an arbitrary data value that is to be passed back whenever PCRE calls a function supplied by the application. External functions can be specified for memory management and for callouts during pattern matching.

```
  pcre2_set_user_data(pcre2_context *context, void *user_data);
```

If external functions are used without setting a value, `NULL` is passed. If in a threaded application the data is different in different threads, a separate context must be used for each thread. An example might be passing a thread identifier to external memory management functions.

5.3 Setting memory management fields in a context

An application that has its own memory management functions must register them in a context before calling PCRE functions that get or free memory, in particular, before calling *pcre2_compile()*. Normally this is done as soon as the context is initialized:

```
pcre2_set_memory_management(context, private_malloc,
  private_free);
```

The prototypes for the private memory management functions are:

```
void *private_malloc(size_t, void *);
void *private_free(void *, void *);
```

When code in PCRE calls these functions, the final argument is taken from the user data field in the context.

By default, PCRE is compiled to use the system stack for recursive function calls when matching patterns using the interpreter (not JIT) with *pcre2_exec()*. In some environments, where the size of this stack is limited, PCRE is often compiled to use heap storage instead. The memory blocks that are used for this purpose are all the same size, and are requested and freed in last-out-first-in order. A private memory manager could implement this kind of usage more efficiently than the general case; to make this possible, two further memory management functions can be added to a context:

```
pcre2_set_recursion_memory_management(context,
  private_recursion_malloc, private_recursion_free);
```

This must be done after calling *pcre2_set_memory_management()* because that function sets the recursion functions to be the same as the normal ones. When PCRE is compiled to use the system stack for recursion, these additional memory management functions are never called.

5.4 Copying a context

An exact copy of a context can be made by:

```
pcre2_context *new_context = pcre2_copy_context(old_context);
```

This could be useful as a way of initializing some standard parameters when creating a new thread, or for saving a context for later use. The memory for the new context is obtained using the *malloc()* setting in the old context.

5.5 Freeing a context

When a context is no longer needed, its memory can be freed by:

```
pcre2_free_context(context);
```

If a private memory management function for *free()* is set in the context, it is used to release the context's memory. Otherwise, the system *free()* is used.

5.6 Setting other parameters in a context

The following functions are provided for setting the remaining parameters in a context. All of them yield 1 for success or 0 if invalid data is given.

```
pcre2_set_match_limit(context, uint32_t limit);
pcre2_set_recursion_limit(context, uint32_t limit);
```

These values limit the resources used by a matching function (formerly passed in a *pcre_extra* structure). The default values are specified when PCRE is built, and are normally quite large.

```
pcre2_set_newline_convention(context, uint32_t newline_code);
```

This specifies which character codes are to be interpreted as newline. The second argument is one of `PCRE2_NEWLINE_CR`, `PCRE2_NEWLINE_LF`, `PCRE2_NEWLINE_CRLF`, `PCRE2_NEWLINE_ANY`, or `PCRE2_NEWLINE_ANYCRLF`. The default is specified when PCRE is built; it is normally the standard for the operating system.

```
pcre2_set_bsr_convention(context, uint32_t bsr_code);
```

This specifies what characters the escape sequence `\R` matches. The allowed values are `PCRE2_BSR_UNICODE` (any Unicode newline sequence) or `PCRE2_BSR_ANYCRLF` (only CR, LF, or CRLF). The default is specified when PCRE is built.

```
  pcre2_set_global_options(context,
    uint32_t unset_option_bits),
    uint32_t set_option_bits,
```

This function sets and unsets on/off options that are to apply to every pattern that is processed using this context. The second and third arguments are a combination of these bits:

`PCRE2_DOLLAR_ENDONLY`	$ matches only at the end
`PCRE2_DUPNAMES`	allow duplicate named subpatterns
`PCRE2_JAVASCRIPT_COMPAT`	modified pattern interpretation
`PCRE2_NEVER_UTF`	forbid `(*UTF)` in patterns
`PCRE2_UTF`	patterns and subjects are coded in UTF
`PCRE2_UCP`	use Unicode Properties for `\d` etc.

The current setting is modified by unsetting the bits in the second argument, and then setting those in the third argument. None of these options are set by default.

Question: Should the name of `PCRE2_JAVASCRIPT_COMPAT` be changed? Some people have suggested that it makes users think full JavaScript compatibility is available. The effect of this option is to make five changes to the way matching works, though I think somebody recently posted that one of these differences with JavaScript has gone away. Adding independent options for each of the differences seems silly (as well as wasting bits), but I can't think of a useful alternative name, unless it is something bland like `PCRE2_PATTERN_TYPE2`. Maybe it's the `COMPAT` bit that is the issue and something like `PCRE2_JAVASCRIPT_PATTERN` would be clearer?

```
  pcre2_set_callout(context, user_callout_function);
```

This records a user callout function. The prototype for the callout function is unchanged:

```
  int user_callout(pcre2_callout_block *, void *);
```

The callout block itself is also unchanged. Setting the function to `NULL`, which is the default, disables callouts.

```
  pcre2_set_character_tables(context, unsigned char *tables);
```

This sets a pointer to custom character tables. The default is to use PCRE's inbuilt tables that were set up when it was built.

5.7 Reading parameter fields in a context

The following functions return the values of fields in a context:

```
  uint32_t       pcre2_get_bsr_convention(context);
  unsigned char *pcre2_get_character_tables(context);
  uint32_t       pcre2_get_global_options(context);
  uint32_t       pcre2_get_match_limit(context);
  uint32_t       pcre2_get_newline_convention(context);
  uint32_t       pcre2_get_recursion_limit(context);
  void          *pcre2_get_user_data(context);
```

Question 1: I haven't specified a function for reading memory management or callout functions. Are these necessary? The only use I can think of is for saving and restoring a context, but this can be done by making a copy of the whole context.

Question 2: In fact, do we really need these 'get' functions at all? An application can easily remember what it has set in a context if it needs to.

6 Compiling a pattern

A pattern is compiled by calling the following function:

```
pcre2 *pcre2_compile(
  pcre2_context *context,
  PCRE2_SPTR     pattern,
  uint32_t       options,
  int           *error_code,
  size_t        *error_offset);
```

The pattern is a zero-terminated string. The following option bits are available:

`PCRE2_ANCHORED`	pattern is anchored
`PCRE2_AUTO_CALLOUT`	generate auto callouts
`PCRE2_CASELESS`	assume caseless at start
`PCRE2_DOTALL`	dot matches all characters
`PCRE2_EXTENDED`	ignore white space in pattern
`PCRE2_FIRSTLINE`	must match before first newline
`PCRE2_JIT`	compile for non-partial JIT matching
`PCRE2_JIT_PARTIAL_SOFT`	compile for soft partial JIT matching
`PCRE2_JIT_PARTIAL_HARD`	compile for hard partial JIT matching
`PCRE2_MULTILINE`	^ and $ may match in mid-subject
`PCRE2_NO_AUTO_CAPTURE`	parentheses do not capture by default
`PCRE2_NO_START_OPTIMIZE`	disable start-of-match optimization
`PCRE2_NO_UTF_CHECK`	disable pattern UTF validity check
`PCRE2_UNGREEDY`	invert greediness

Note: the `PCRE_EXTRA` option, which caused unknown escape sequences to give an error, like Perl's `-w` option, is now assumed always to be on.

When successful, *pcre2_compile()* returns a pointer to an opaque structure that contains the compiled pattern. This data is read-only, that is, it is never changed during pattern matching. Therefore, compiled patterns may be safely shared between threads.

If there is a compilation error, the function returns `NULL`. A positive error code and the offset in the pattern where the error occurred are placed in the variables pointed to by *error_code* and *error_offset*, respectively. The code can be translated into a textual error message by this function:

```
int pcre2_get_error_message(int error_code, PCRE2_UCHAR *buffer,
  size_t buffer_size);
```

This copies a zero-terminated error message into the supplied buffer, whose data items are of the appropriate width (8, 16, or 32). It returns 1 if all is well, 0 if the buffer is too small. (The old API always returned 8-bit error messages.)

When a compiled pattern is no longer needed, it can be freed by:

```
pcre2_free_compiled_code(context, code);
```

7 Explicit studying is abolished

I introduced a separate *pcre_study()* function when PCRE was first implemented because I wasn't sure how much resource this would take. It turns out to be not very much (and processors are getting faster and faster). Therefore, in the new API, studying happens automatically and you don't have to worry about it. JIT compiling, on the other hand, is still expensive, so must remain optional.

8 Matching functions

The first releases of PCRE returned only the contents of *ovector*. When more information was needed, I found ways of passing it back in a compatible manner (e.g. in the *pcre_extra* block). It is time to tidy this up. I propose a new opaque structure called *pcre2_match_data*, which contains space for remembering the results of a match, and also contains working space for the matching function.

The primary data is in the *ovector*. This is now a vector of *size_t* instead of *int*. There is a special value, `PCRE_OVECTOR_UNSET`, probably defined as `(~(size_t)0)`, that is used for unset fields.

There are several different ways this might be set up. Of the three ideas below, I think I prefer option A because it is the simplest and easiest to document. In all three cases there will be a *pcre2_free_match_data()* function that releases the match data memory.

8.1 Option A: caller supplies ovector

This would work like this:

```
size_t ovector[20];
pcre2_match_data *match_data =
  pcre2_create_match_data(context, ovector, 20);
```

The application provides *ovector* and specifies its size; a pointer to it is stored in the *match_data* block for use by the matching function. We can get rid of the fiddle whereby the top one third of the *ovector* is used as working space because the working space can be elsewhere in the block. As in the current API, passing *ovector* as NULL is allowed if the caller has no interest in the matched strings.

8.2 Option B: ovector is in the match data

In this case, all the caller provides is a size for the *ovector*:

```
pcre2_match_data *match_data =
  pcre2_create_match_data(context, 20);
```

As that stands, the caller does not know where *ovector* is. We could use a function to get a pointer to it, or instead use this:

```
pcre2_match_data *match_data =
  pcre2_create_match_data(context, 20, &ovector_pointer);
```

If the user is not interested in the matched strings, a size of zero can be given.

8.3 Option C: combining options A and B

This option uses the following creation function:

```
pcre2_create_match_data(void *context, int ovector_size,
  size_t **ovector_ptr);
```

If *ovector_ptr* is NULL, the caller doesn't care about the matched strings (and therefore does not need a pointer to *ovector*). There is no need to allocate any space; the value of *ovector_size* is ignored.

If *ovector_ptr* is not NULL and **ovector_ptr* is also not NULL, the caller has supplied a pointer to an *ovector*, guaranteed to contain at least *ovector_size* elements. In this case, the caller is responsible for freeing *ovector* afterwards, if necessary.

Finally, if *ovector_ptr* is not NULL but **ovector_ptr* is NULL, the caller expects the function to allocate *ovector* and give back a pointer to it in **ovector_ptr*. In this case, *ovector* is freed as part of *pcre2_free_match_data()*.

9 Perl-compatible matching

The following function matches in a Perl-compatible manner:

```
int pcre2_exec(
  pcre2_context    *context,
  const pcre2      *code,
  PCRE2_SPTR        subject,
  int               length,
  size_t            startoffset,
  uint32_t          options,
  pcre2_match_data *match_data);
```

A negative length means ‘zero-terminated string’. The following option bits are available:

`PCRE2_ANCHORED`	pattern is anchored
`PCRE2_NOTBOL`	subject is not the beginning of a line
`PCRE2_NOTEOL`	subject is not the end of a line
`PCRE2_NOTEMPTY`	must not match an empty string
`PCRE2_NOTEMPTY_ATSTART`	not empty at start of subject
`PCRE2_NO_START_OPTIMIZE`	disable start-of-match optimization
`PCRE2_NO_UTF_CHECK`	disable subject UTF validity check
`PCRE2_PARTIAL_SOFT`	soft partial match
`PCRE2_PARTIAL_HARD`	hard partial match

The return codes are unchanged from the old API: zero or positive for a complete match, negative for error or a partial match. Matched strings and substrings are passed back via *ovector*, as before. Note, however, that the value for an unset capturing group is `PCRE_OVECTOR_UNSET` instead of a negative number, because *ovector* is now of type *size_t*.

9.1 Errors while matching

An error message can be obtained for any error code using the same function as for *pcre2_compile()*:

```
int pcre2_get_error_message(int error_code, PCRE2_UCHAR *buffer,
  size_t buffer_size);
```

The offset in the subject where the error occurred can be obtained by:

```
size_t pcre2_get_error_offset(pcre_match_data *);
```

When the error is `PCRE2_ERROR_BADUTF8` or `PCRE2_ERROR_SHORTUTF8`, another function can be called to obtain a detailed reason code:

```
int pcre2_get_error_reason(pcre2_match_data *);
```

This yields values such as `PCRE2_UTF8_ERR1` (truncated UTF-8 character). The use of these functions replaces the previous rather untidy scheme of putting values into the output vector.

9.2 Additional match information

As well as the offsets that are passed back in *ovector* for a successful match, other data from the most recent match is remembered, whether it succeeded or failed. This can be extracted from the match data using the following functions:

```
PCRE2_SPTR pcre2_get_mark(pcre2_match_data *);
```

If the match found a `(*MARK)` name to pass back, a pointer to it is returned. Otherwise the function returns NULL. The name is a zero-terminated string within the compiled pattern (as before).

```
size_t pcre2_get_startchar(pcre2_match_data *);
```

This function returns the offset of the character where the final matching process began. For an anchored pattern, the value is always *startoffset*. This offset is different to the starting offset of the matched string if `\K` was encountered.

```
size_t pcre2_get_leftchar(pcre2_match_data *);
size_t pcre2_get_rightchar(pcre2_match_data *);
```

These functions return the offsets of the leftmost and one more than the rightmost characters that were inspected during the final match. Lookbehinds and lookaheads can make these offsets less than or greater than the offsets of the matched string, respectively. For example, when the pattern

```
(?<=abc)def(?=ghi)
```

is matched against the string "`abcdefghi`" the offsets in *ovector* are 3 and 6, corresponding to "`def`", whereas the leftmost and rightmost offsets are 0 and 9.

9.3 Change to partial matching

In the current API, after a partial match, the first three values in *ovector* are the leftmost character, the end of the partial match (the end of the subject), and the start match offset. (This is because there have been various changes and additions over the years.) In the new API, only two values are set in *ovector*, and they are those of the partially matched string, giving consistency with a complete match. The other offsets are now available for all matches using the functions just described.

9.4 Obtaining the frame size

In the current API, a call to *pcre_exec()* with NULL arguments is a convention for obtaining the size of the stack or heap frame (depending on how PCRE was compiled) that is used for recursive calls of the matching function. This facility is now provided by a separate function:

```
int size = pcre2_get_frame_size();
```

10 DFA matching

The DFA matching function needs workspace. A vector of at least 20 *int*s is recommended; more is needed for patterns that have a lot of potential matches. I have used additional arguments, as in the current API, but an alternative would be to have a separate *pcre2_dfa_match_data* block, in which case the rules for handling the workspace should be the same as for handling the ovector (see options A, B, and C above).

```
int pcre2_dfa_exec(
  pcre2_context    *context,
  const pcre2      *code,
  PCRE2_SPTR        subject,
  int               length,
  size_t            startoffset,
  uint32_t          options,
  pcre2_match_data *match_data,
  int              *workspace,
  size_t            wscount);
```

The following option bits are available in addition to those of *pcre2_exec()*:

`PCRE2_DFA_RESTART`	restart after a partial match
`PCRE2_DFA_SHORTEST`	find only the shortest match

11 JIT matching

The functions for detailed JIT matching are adjusted for the new API in fairly obvious ways:

```
pcre2_jit_stack *pcre2_jit_stack_alloc(pcre2_context *,
  size_t, size_t);
void pcre2_jit_stack_free(pcre2_context *, jit_stack);
void pcre2_assign_jit_stack(pcre2_context *, const pcre2 *,
  pcre2_jit_callback, void *);
void pcre2_jit_free_unused_memory(pcre2_context *);
int pcre2_jit_exec(pcre2_context *, const pcre2 *, PCRE2_SPTR,
  int, size_t, uint32_t, pcre2_match_data *, pcre2_jit_stack *);
```

The context is passed to all of them, and the abolition of *pcre_extra* means that a *pcre2* pointer is passed instead. The arguments for *pcre2_jit_exec()* are now the same as the new *pcre2_exec()*, plus a JIT stack pointer.

12 Pattern information

The function for obtaining information about a compiled pattern is now:

```
int pcre2_get_info(const pcre2 *, uint32_t, void *);
```

Many of the information items are unchanged, but I have removed those that are obsolete or deprecated, and done some renaming. This is the new list, with further comment below on those that are changed:

`PCRE2_INFO_BACKREFMAX`	highest backreference
`PCRE2_INFO_CAPTURECOUNT`	number of capturing subpatterns
`PCRE2_INFO_COMPILE_OPTIONS`	options set for compile
`PCRE2_INFO_FIRSTDATA_ITEM`	value of first data item
`PCRE2_INFO_FIRSTDATA_TYPE`	type of first data information
`PCRE2_INFO_FIRSTTABLE`	table of first data values
`PCRE2_INFO_HASCRORLF`	has explicit CR or LF
`PCRE2_INFO_JCHANGED`	`(?J)` or `(?-J)` was used
`PCRE2_INFO_JIT`	successful JIT compilation
`PCRE2_INFO_JITSIZE`	JIT compiled code size
`PCRE2_INFO_LASTDATA_ITEM`	value of last data item
`PCRE2_INFO_LASTDATA_TYPE`	type of last data information
`PCRE2_INFO_MATCH_EMPTY`	can match an empty string
`PCRE2_INFO_MATCHLIMIT`	limit set within the pattern
`PCRE2_INFO_MAXLOOKBEHIND`	maximum lookbehind, in characters
`PCRE2_INFO_MINLENGTH`	minimum length, in characters
`PCRE2_INFO_NAMECOUNT`	number of name table entries
`PCRE2_INFO_NAMEENTRYSIZE`	size of each entry
`PCRE2_INFO_NAMETABLE`	pointer to the name table
`PCRE2_INFO_PATTERN_OPTIONS`	options set within the pattern
`PCRE2_INFO_RECURSIONLIMIT`	limit set within the pattern
`PCRE2_INFO_SIZE`	size of compiled pattern (bytes)

Options that are explicitly passed to *pcre2_compile()* and those that are deduced from the pattern, for example, by the use of `(?i)`, are saved separately in the updated code; hence the splitting of `PCRE_INFO_OPTIONS` into two new options.

`PCRE_INFO_STUDYSIZE` was only ever provided to make it possible to save and restore the separate study data, so it is no longer relevant.

`PCRE2_INFO_FIRSTDATA_TYPE` gives information about the first data unit in a non-anchored pattern. If there is a fixed first value, for example, the letter `c` from a pattern such as `(cat|cow|coyote)`, 1 is returned. In this situation, the value can be retrieved using `PCRE2_INFO_FIRSTDATA_ITEM`, which returns the fixed first data item value.

If there is no fixed first value, and if either (a) the pattern was compiled with the `PCRE2_MULTILINE` option, and every branch starts with "^", or (b) every branch of the pattern starts with ".*" and `PCRE2_DOTALL` is not set (if it were set, the pattern would be anchored), 2 is returned, indicating that the pattern matches only at the start of a subject string or after any newline within the string. Otherwise 0 is returned. For anchored patterns, 0 is returned. In all these cases, `PCRE2_INFO_FIRSTDATA_ITEM` returns 0.

`PCRE2_INFO_LASTDATA_TYPE` returns 1 if there is a rightmost literal data item that must exist, other than at the start of the subject, for a match to be possible. Otherwise it returns 0. In situations where 1 is returned, `PCRE2_INFO_LASTDATA_ITEM` can be used to retrieve the value. In other cases, it returns 0.

13 Reference counts

The old API contains a function called *pcre_refcount()* which can be used to maintain a reference count within a compiled pattern. This breaks the assumption that a compiled pattern is a read-only structure. Also, it is not atomic, and therefore not thread-safe.

I do not think that introducing thread-specific functions such as atomic updates into the API just for this case is a good idea because it complicates the code and the specification, and makes building PCRE difficult in environments that do not support threads. Though it has been in PCRE since release 6.0, I propose to abolish *pcre_refcount()*.

Applications that need to maintain reference counts should instead define their own structure, something like this:

```
struct my_code {
  pcre2 *code;
  int refcount;
  ...whatever...
};
```

Then they can manipulate the reference count any way they like, and the *pcre2* structure remains read-only.

14 Character tables

I propose no change to the way PCRE handles character tables, so this function remains:

```
const unsigned char *pcre2_maketables(void);
```

Note, however, that the pointer to custom character tables is now held in the context.

15 Configuration information

There is no change to the function for obtaining configuration information:

```
int pcre2_config(int what, void *where);
```

The available information is unchanged. However, if building PCRE is simplified so that including UTF always also includes Unicode property support (see below), `PCRE_CONFIG_UCP` can be removed.

16 PCRE version

For consistency with the rest of the API, *pcre_version()* is changed to:

```
int pcre2_version(PCRE2_UCHAR *buffer, size_t size);
```

The version and date string is copied into the supplied buffer. This allows the different libraries to return the version information in data items of the appropriate width. The function returns 1 on success, or 0 if the buffer is too small.

17 Byte-ordering functions

The prototypes for these functions are the obvious adaptions:

```
int pcre2_pattern_to_host_byte_order(pcre2 *);
int pcre2_utf16_to_host_byte_order(PCRE2_UCHAR16 *, PCRE2_SPTR16,
  int, int *, int);
int pcre2_utf32_to_host_byte_order(PCRE2_UCHAR32 *, PCRE2_SPTR32,
  int, int *, int);
```

18 Substring extraction functions

As the *ovector* pointer and the details of the most recent match are remembered in the match_data, there is no need to pass them to the string extraction functions. Apart from changes to the variable types and the addition of a context argument for functions that get memory, these are otherwise unchanged.

```
int pcre2_copy_named_substring(
  pcre2_match_data *match_data,
  PCRE2_SPTR name,
  PCRE2_UCHAR *buffer,
  size_t buffsize);
int pcre2_copy_substring(
  pcre2_match_data *match_data,
  int stringnumber,
  PCRE2_UCHAR *buffer,
  size_t buffsize);

int pcre2_get_named_substring(
  pcre2_context *context,
  pcre2_match_data *match_data,
  PCRE2_SPTR name,
  PCRE2_UCHAR **);

int pcre2_get_substring(
  pcre2_context *context,
  pcre2_match_data *match_data,
  int stringnumber,
  PCRE2_UCHAR **);

void pcre2_free_substring(
  pcre2_context *context,
  PCRE2_SPTR string);

int pcre2_get_substring_list(
  pcre2_context *context,
  pcre2_match_data *match_data,
  PCRE2_UCHAR ***);

void pcre2_free_substring_list(
  pcre2_context *context,
  PCRE2_SPTR *list);

int pcre2_get_stringnumber(
  const pcre2 *code,
  PCRE2_SPTR name);

int pcre2_get_stringtable_entries(
  const pcre2 *code,
  PCRE2_SPTR name,
  PCRE2_UCHAR **first,
  PCRE2_UCHAR **last);
```

19 Pre-compiled patterns

The facility for saving and restoring pre-compiled patterns is, I believe, used, so it should be preserved. The new code combines what was formerly separate study data into the main pattern structure, which makes things simpler, and the existing instructions for saving and restoring should continue to work in the new API.

Question: Would it be better to provide explicit serializing and de-serializing functions? If so, they should incorporate the function of *pcre2_pattern_to_host_byte_order()*, which could then be abolished.

20 C++

The C++ wrapper supports only the 8-bit library and is currently not maintained. Unless a maintainer comes forward, I think it would be better to discard it. A new version should support 8-bit, 16-bit and 32-bit handling.

21 Substitution function

There have been requests for a substitution (find and replace) function. The existing C++ wrapper contains such a function, so maybe now is the time to provide one in the main library. Here is a possible specification:

```
int pcre2_substitute(
  pcre2_context *context,          context
  pcre2 *code,                     compiled pattern
  PCRE2_SPTR subject,              subject string
  int slength,                     length of subject string
  size_t startoffset,              offset to start search
  uint32_t options,                pcre_exec() options
  PCRE2_SPTR replacement,          replacement string
  int rlength,                     length of replacement string
  PCRE2_UCHAR *buffer,             where to put result string
  size_t blength,                  length of buffer
  size_t *rlength);                where to return result length
```

The first six arguments are the same as the arguments for *pcre2_exec()*.

Question: Should there be a *pcre2_match_data* argument? I have not specified this because the substitution function can get one for itself and free it when finished. It can find the number of captured substrings in order to set up an appropriate *ovector*. This uses more resources, but this is after all a convenience function. An application that is worried about performance probably would use its own code instead.

The allowed options, except for the partial matching options, are the same as for *pcre_exec()*. The replacement string is given as a pointer and a length so that binary strings can be processed. A negative length indicates a zero-terminated string. The string may contain substitution fragments in these forms:

`$<number>`	e.g. `ab$1cd`
`${<number>}`	e.g. `12${3}34`
`$name`	e.g. `a $name b`
`${name}`	e.g. `a${name}b`

The modified string is placed in *buffer*, whose length is *blength*. For the convenience of applications that are processing zero-terminated strings, a zero data item is added at the end. The length of the modified string (excluding the terminating zero) is placed in the variable pointed to by *rlength*. The function returns the length of the initial copied substring plus the length of the expanded replacement string. This is the offset to 'the rest of the string'.

If there is an error, a negative error code is returned. `PCRE2_ERROR_NOMATCH` is given for no match, and `PCRE2_ERROR_BADLENGTH` if the buffer is not large enough.

This function does a single substitution on the first match that is found in the subject string. It is an application's responsibility to call the function again if global replacement is wanted. The value returned by the function is the offset in the modified string at which to start the next search.

Question 1: Should it be possible to pass `NULL` as a buffer, and have the function get the memory? If this is allowed, there will have to be another argument, for passing back the address of the buffer. (Or, it could be the result of the function, but then another argument is needed for passing back an error code.)

Question 2: Should there be an option for requesting global changes? This is relatively straightforward when an output buffer is passed as an argument. It is much harder if `NULL` is allowed because it makes discovering how much memory to get much more complicated. Either all the pattern matches must be done twice, or the strings must be copied into new memory for each match. (Or all the details of each match must be remembered somewhere).

22 Build-time changes

Originally, UTF support was implemented without UCP support, so when the latter was added later, it was made optional. Perhaps this is nowadays rather pointless; we could make UTF imply UCP.

23 The POSIX wrapper

There can, of course, be no change to the API for the POSIX wrapper. The revised functions for the new API will use PCRE contexts with default settings.

-oOo-