NAME

ucl_parser_new, **ucl_parser_register_macro**, **ucl_parser_register_variable**, **ucl_parser_add_chunk**, **ucl_parser_add_string**, **ucl_parser_add_file**, **ucl_parser_get_object**, **ucl_parser_get_error**, **ucl_parser_free**, **ucl_pubkey_add**, **ucl_parser_set_filevars** - universal configuration library parser and utility functions

LIBRARY

UCL library (libucl, -lucl)

SYNOPSIS

#include <ucl.h>

DESCRIPTION

Libucl is a parser and C API to parse and generate ucl objects. Libucl consist of several groups of functions:

Parser functions

Used to parse ucl files and provide interface to extract ucl object. Currently, libucl can parse only full ucl documents, for instance, it is impossible to parse a part of document and therefore it is impossible to use libucl as a streaming parser. In future, this limitation can be removed.

Emitting functions

Convert ucl objects to some textual or binary representation. Currently, libucl supports the following exports:

- JSON - valid json format (can possibly lose some original data, such as implicit arrays)
- Config - human-readable configuration format (lossless)
- YAML - embedded yaml format (has the same limitations as json output)

Conversion functions

Help to convert ucl objects to C types. These functions are used to convert ucl_object_t to C primitive types, such as numbers, strings or boolean values.

Generation functions

Allow creation of ucl objects from C types and creating of complex ucl objects, such as hashes or arrays from primitive ucl objects, such as numbers or strings.

Iteration functions

Iterate over ucl complex objects or over a chain of values, for example when a key in an object has multiple values (that can be treated as implicit array or implicit consolidation).

Validation functions

Validation functions are used to validate some object obj using json-schema compatible object schema. Both input and schema must be UCL objects to perform validation.

Utility functions

Provide basic utilities to manage ucl objects: creating, removing, retaining and releasing reference count and so on.

PARSER FUNCTIONS

Parser functions operates with struct ucl_parser.

ucl_parser_new

```
struct ucl_parser* ucl_parser_new (int flags);
```

Creates new parser with the specified flags:

- UCL_PARSER_KEY_LOWERCASE - lowercase keys parsed
- UCL_PARSER_ZEROCOPY - try to use zero-copy mode when reading files (in zero-copy mode text chunk being parsed without copying strings so it should exist till any object parsed is used)
- UCL_PARSER_NO_TIME - treat time values as strings without parsing them as floats

ucl_parser_register_macro

```
void ucl_parser_register_macro (struct ucl_parser *parser,
  const char *macro, ucl_macro_handler handler, void* ud);
```

Register new macro with name .macro parsed by handler handler that accepts opaque data pointer ud. Macro handler should be of the following type:

```
bool (*ucl_macro_handler) (const unsigned char *data,
  size_t len, void* ud);‘
```

Handler function accepts macro text data of length len and the opaque pointer ud. If macro is parsed successfully the handler should return true. false indicates parsing failure and the parser can be terminated.

ucl_parser_register_variable

```
void ucl_parser_register_variable (struct ucl_parser *parser,
  const char *var, const char *value);
```

Register new variable $var that should be replaced by the parser to the value string.

ucl_parser_add_chunk

```
bool ucl_parser_add_chunk (struct ucl_parser *parser,
  const unsigned char *data, size_t len);
```

Add new text chunk with data of length len to the parser. At the moment, libucl parser is not a streamlined parser and chunk *must* contain the *valid* ucl object. For example, this object should be valid:

```
{ "var": "value" }
```

while this one won't be parsed correctly:

```
{ "var":
```

This limitation may possible be removed in future.

ucl_parser_add_string

```
bool ucl_parser_add_string (struct ucl_parser *parser,
  const char *data, size_t len);
```

This function acts exactly like ucl_parser_add_chunk does but if len argument is zero, then the string data must be zero-terminated and the actual length is calculated up to \0 character.

ucl_parser_add_file

```
bool ucl_parser_add_file (struct ucl_parser *parser,
  const char *filename);
```

Load file filename and parse it with the specified parser. This function uses mmap call to load file, therefore, it should not be shrunk during parsing. Otherwise, libucl can cause memory corruption and terminate the calling application. This function is also used by the internal handler of include macro, hence, this macro has the same limitation.

ucl_parser_get_object

```
ucl_object_t* ucl_parser_get_object (struct ucl_parser *parser);
```

If the ucl data has been parsed correctly this function returns the top object for the parser. Otherwise, this function returns the NULL pointer. The reference count for ucl object returned is increased by one, therefore, a caller should decrease reference by using ucl_object_unref to free object after usage.

ucl_parser_get_error

```
const char *ucl_parser_get_error(struct ucl_parser *parser);
```

Returns the constant error string for the parser object. If no error occurred during parsing a NULL object is returned. A caller should not try to free or modify this string.

ucl_parser_free

```
void ucl_parser_free (struct ucl_parser *parser);
```

Frees memory occupied by the parser object. The reference count for top object is decreased as well, however if the function ucl_parser_get_object was called previously then the top object won’t be freed.

ucl_pubkey_add

```
bool ucl_pubkey_add (struct ucl_parser *parser,
    const unsigned char *key, size_t len);
```

This function adds a public key from text blob key of length len to the parser object. This public key should be in the PEM format and can be used by .includes macro for checking signatures of files included. Openssl support should be enabled to make this function working. If a key cannot be added (e.g. due to format error) or openssl was not linked to libucl then this function returns false.

ucl_parser_set_filevars

```
bool ucl_parser_set_filevars (struct ucl_parser *parser,
    const char *filename, bool need_expand);
```

Add the standard file variables to the parser based on the filename specified:

- $FILENAME - a filename of ucl input
- $CURDIR - a current directory of the input

For example, if a filename param is ../something.conf then the variables will have the following values:

- $FILENAME - "../something.conf"
- $CURDIR - ".."

if need_expand parameter is true then all relative paths are expanded using realpath call. In this example if .. is /etc/dir then variables will have these values:

- $FILENAME - "/etc/something.conf"
- $CURDIR - "/etc"

Parser usage example

The following example loads, parses and extracts ucl object from stdin using libucl parser functions (the length of input is limited to 8K):

```
char inbuf[8192];
```

```
struct ucl_parser *parser = NULL;
int ret = 0, r = 0;
ucl_object_t *obj = NULL;
FILE *in;

in = stdin;
parser = ucl_parser_new (0);
while (!feof (in) && r < (int)sizeof (inbuf)) {
   r += fread (inbuf + r, 1, sizeof (inbuf) - r, in);
}
ucl_parser_add_chunk (parser, inbuf, r);
fclose (in);

if (ucl_parser_get_error (parser)) {
   printf ("Error occurred: %s\n", ucl_parser_get_error (parser));
   ret = 1;
}
else {
   obj = ucl_parser_get_object (parser);
}

if (parser != NULL) {
   ucl_parser_free (parser);
}
if (obj != NULL) {
   ucl_object_unref (obj);
}
return ret;
```

EMITTING FUNCTIONS

Libucl can transform UCL objects to a number of tectual formats:

- configuration (UCL_EMIT_CONFIG) - nginx like human readable configuration file where implicit arrays are transformed to the duplicate keys
- compact json: UCL_EMIT_JSON_COMPACT - single line valid json without spaces
- formatted json: UCL_EMIT_JSON - pretty formatted JSON with newlines and spaces

- compact yaml: UCL_EMIT_YAML - compact YAML output

Moreover, libucl API allows to select a custom set of emitting functions allowing efficient and zero-copy output of libucl objects. Libucl uses the following structure to support this feature:

```
struct ucl_emitter_functions {
  /** Append a single character */
  int (*ucl_emitter_append_character) (unsigned char c, size_t nchars, void *ud);
  /** Append a string of a specified length */
  int (*ucl_emitter_append_len) (unsigned const char *str, size_t len, void *ud);
  /** Append a 64 bit integer */
  int (*ucl_emitter_append_int) (int64_t elt, void *ud);
  /** Append floating point element */
  int (*ucl_emitter_append_double) (double elt, void *ud);
  /** Opaque userdata pointer */
  void *ud;
};
```

This structure defines the following callbacks:

- ucl_emitter_append_character - a function that is called to append nchars characters equal to c
- ucl_emitter_append_len - used to append a string of length len starting from pointer str
- ucl_emitter_append_int - this function applies to integer numbers
- ucl_emitter_append_double - this function is intended to output floating point variable

The set of these functions could be used to output text formats of UCL objects to different structures or streams.

Libucl provides the following functions for emitting UCL objects:

ucl_object_emit

```
unsigned char *ucl_object_emit (const ucl_object_t *obj, enum ucl_emitter emit_type);
```

Allocate a string that is suitable to fit the underlying UCL object obj and fill it with the textual representation of the object obj according to style emit_type. The caller should free the returned string after using.

ucl_object_emit_full

bool ucl_object_emit_full (const ucl_object_t *obj, enum ucl_emitter emit_type,
struct ucl_emitter_functions *emitter);

This function is similar to the previous with the exception that it accepts the additional argument emitter that defines the concrete set of output functions. This emit function could be useful for custom structures or streams emitters (including C++ ones, for example).

CONVERSION FUNCTIONS

Conversion functions are used to convert UCL objects to primitive types, such as strings, numbers, or boolean values. There are two types of conversion functions:

- safe: try to convert an ucl object to a primitive type and fail if such a conversion is not possible
- unsafe: return primitive type without additional checks, if the object cannot be converted then some reasonable default is returned (NULL for strings and 0 for numbers)

Also there is a single ucl_object_tostring_forced function that converts any UCL object (including compound types - arrays and objects) to a string representation. For objects, arrays, booleans and numeric types this function performs emitting to a compact json format actually.

Here is a list of all conversion functions:

- ucl_object_toint - returns int64_t of UCL object
- ucl_object_todouble - returns double of UCL object
- ucl_object_toboolean - returns bool of UCL object
- ucl_object_tostring - returns const char * of UCL object (this string is NULL terminated)
- ucl_object_tolstring - returns const char * and size_t len of UCL object (string does not need to be NULL terminated)

- ucl_object_tostring_forced - returns string representation of any UCL object

Strings returned by these pointers are associated with the UCL object and exist over its lifetime. A caller should not free this memory.

GENERATION FUNCTIONS

It is possible to generate UCL objects from C primitive types. Moreover, libucl allows creation and modifying complex UCL objects, such as arrays or associative objects.

ucl_object_new

ucl_object_t * ucl_object_new (void)

Creates new object of type UCL_NULL. This object should be released by caller.

ucl_object_typed_new

ucl_object_t * ucl_object_typed_new (unsigned int type)

Create an object of a specified type: - UCL_OBJECT - UCL object - key/value pairs - UCL_ARRAY - UCL array - UCL_INT - integer number - UCL_FLOAT - floating point number - UCL_STRING - NULL terminated string - UCL_BOOLEAN - boolean value - UCL_TIME - time value (floating point number of seconds) - UCL_USERDATA - opaque userdata pointer (may be used in macros) - UCL_NULL - null value

This object should be released by caller.

Primitive objects generation

Libucl provides the functions similar to inverse conversion functions called with the specific C type: - ucl_object_fromint - converts int64_t to UCL object - ucl_object_fromdouble - converts double to UCL object - ucl_object_fromboolean - converts bool to UCL object - ucl_object_fromstring - converts const char * to UCL object (this string should be NULL terminated) - ucl_object_fromlstring - converts const char * and size_t len to UCL object (string does not need to be NULL terminated)

Also there is a function to generate UCL object from a string performing various parsing or conversion operations called ucl_object_fromstring_common.

ucl_object_fromstring_common

ucl_object_t * ucl_object_fromstring_common (const char *str,
size_t len, enum ucl_string_flags flags)

This function is used to convert a string str of size len to a UCL object applying flags conversions. If len is equal to zero then a str is assumed as NULL-terminated. This function supports the following flags (a set of flags can be specified using logical OR operation):

- UCL_STRING_ESCAPE - perform JSON escape
- UCL_STRING_TRIM - trim leading and trailing whitespaces
- UCL_STRING_PARSE_BOOLEAN - parse passed string and detect boolean
- UCL_STRING_PARSE_INT - parse passed string and detect integer number
- UCL_STRING_PARSE_DOUBLE - parse passed string and detect integer or float number
- UCL_STRING_PARSE_TIME - parse time values as floating point numbers
- UCL_STRING_PARSE_NUMBER - parse passed string and detect number (both float, integer and time types)
- UCL_STRING_PARSE - parse passed string (and detect booleans, numbers and time values)
- UCL_STRING_PARSE_BYTES - assume that numeric multipliers are in bytes notation, for example 10k means 10*1024 and not 10*1000 as assumed without this flag

If parsing operations fail then the resulting UCL object will be a UCL_STRING. A caller should always check the type of the returned object and release it after using.

ITERATION FUNCTIONS

Iteration are used to iterate over UCL compound types: arrays and objects. Moreover, iterations could be performed over the keys with multiple values (implicit arrays). There are two types of iterators API: old and unsafe one via ucl_iterate_object and the proposed interface of safe iterators.

ucl_iterate_object

const ucl_object_t* ucl_iterate_object (const ucl_object_t *obj,
ucl_object_iter_t *iter, bool expand_values);

This function accepts opaque iterator pointer iter. In the first call this iterator *must* be initialized to NULL. *Iterator is changed by this function call. ucl_iterate_object* returns the next UCL object in the compound object obj *or NULL* if all objects have been iterated. The reference count of the object returned is not increased, so a caller should not unref the object or modify its content (e.g. by inserting to another compound object). The object obj *should not be changed during the iteration process as well. expand_values* flag speicifies whether ucl_iterate_object *should expand keys with multiple values. The general rule is that if you need to iterate through the object or explicit array, then you always need to set this flag to true. However, if you get some key in the object and want to extract all its values then you should set expand_values to false. Mixing of iteration types is not permitted since the iterator is set according to the iteration type and cannot be reused. Here is an example of iteration over the objects using libucl API (assuming that top is UCL_OBJECT in this example):*

```
ucl_object_iter_t it = NULL, it_obj = NULL;
const ucl_object_t *cur, *tmp;

/* Iterate over the object */
while ((obj = ucl_iterate_object (top, &it, true))) {
   printf ("key: \"%s\"\n", ucl_object_key (obj));
   /* Iterate over the values of a key */
   while ((cur = ucl_iterate_object (obj, &it_obj, false))) {
      printf ("value: \"%s\"\n",
         ucl_object_tostring_forced (cur));
   }
}
```

Safe iterators API

Safe iterators are defined to clarify iterating over UCL objects and simplify flattening of UCL objects in non-trivial cases. For example, if there is an implicit array that contains another array and a boolean value it is extremely unclear how to iterate over such an object. Safe iterators are desinged to define two sorts of iteration:

1. Iteration over complex objects with expanding all values

2. Iteration over complex objects without expanding of values

The following example demonstrates the difference between these two types of iteration:

```
key = 1;
key = [2, 3, 4];
```

Iteration with expansion:

```
1, 2, 3, 4
```

Iteration without expansion:

```
1, [2, 3, 4]
```

UCL defines the following functions to manage safe iterators:

- ucl_object_iterate_new - creates new safe iterator.
- ucl_object_iterate_reset - resets iterator to a new object.
- ucl_object_iterate_safe - safely iterate the object inside iterator. Note: function may allocate and free memory during its operation. Therefore it returns NULL either while trying to access item after the last one or when exception (such as memory allocation failure) happens.
- ucl_object_iter_chk_excpn - check if the last call to ucl_object_iterate_safe ended up in unrecoverable exception (e.g. ENOMEM).
- ucl_object_iterate_free - free memory associated with the safe iterator.

Please note that unlike unsafe iterators, safe iterators *must* be explicitly initialized and freed. An assert is likely generated if you use uninitialized or NULL *iterator in all safe iterators functions.*

```
ucl_object_iter_t it;
const ucl_object_t *cur;

it = ucl_object_iterate_new (obj);

while ((cur = ucl_object_iterate_safe (it, true)) != NULL) {
   /* Do something */
}
/* Check error condition */
```

```
if (ucl_object_iter_chk_excpn (it)) {
   ucl_object_iterate_free (it);
   exit (1);
}

/* Switch to another object */
it = ucl_object_iterate_reset (it, another_obj);

while ((cur = ucl_object_iterate_safe (it, true)) != NULL) {
   /* Do something else */
}
/* Check error condition */
if (ucl_object_iter_chk_excpn (it)) {
   ucl_object_iterate_free (it);
   exit (1);
}

ucl_object_iterate_free (it);
```

VALIDATION FUNCTIONS

Currently, there is only one validation function called ucl_object_validate. It performs validation of object using the specified schema. This function is defined as following:

ucl_object_validate

```
bool ucl_object_validate (const ucl_object_t *schema,
   const ucl_object_t *obj, struct ucl_schema_error *err);
```

This function uses ucl object schema, that must be valid in terms of json-schema draft v4, to validate input object obj. If this function returns true then validation procedure has been succeed. Otherwise, false is returned and err is set to a specific value. If a caller sets err to NULL then this function does not set any error just returning false. Error is the structure defined as following:

```
struct ucl_schema_error {
   enum ucl_schema_error_code code;    /* error code */
   char msg[128];              /* error message */
   ucl_object_t *obj;          /* object where error occurred */
```

```
};
```

Caller may use code field to get a numeric error code:

```
enum ucl_schema_error_code {
  UCL_SCHEMA_OK = 0,         /* no error */
  UCL_SCHEMA_TYPE_MISMATCH,   /* type of object is incorrect */
  UCL_SCHEMA_INVALID_SCHEMA,  /* schema is invalid */
  UCL_SCHEMA_MISSING_PROPERTY,/* missing properties */
  UCL_SCHEMA_CONSTRAINT,      /* constraint found */
  UCL_SCHEMA_MISSING_DEPENDENCY, /* missing dependency */
  UCL_SCHEMA_UNKNOWN          /* generic error */
};
```

msg is a string description of an error and obj is an object where error has occurred. Error object is not allocated by libucl, so there is no need to free it after validation (a static object should thus be used).

AUTHORS

Vsevolod Stakhov <vsevolod@highsecure.ru>.