
libcbor Documentation

Release 0.5.0

Pavel Kalvoda

Jan 27, 2019

Contents

1	Overview	3
2	Contents	5
2.1	Getting started	5
2.2	Usage & preliminaries	12
2.3	API	15
2.4	Streaming & indefinite items	45
2.5	Tests	49
2.6	RFC conformance	50
2.7	Internal mechanics	50
2.8	Changelog	53
2.9	Development	54

Documentation for version 0.5.0, updated on Jan 27, 2019.

libcbor is a C library for parsing and generating **CBOR**, the general-purpose schema-less binary data format.

Main features

- Complete RFC conformance¹
- Robust C99 implementation
- Layered architecture offers both control and convenience
- Flexible memory management
- No shared global state - threading friendly²
- Proper handling of UTF-8
- Full support for streams & incremental processing
- Extensive documentation and test suite
- No runtime dependencies, small footprint

¹ See *RFC conformance*

² With the exception of custom memory allocators (see *Memory management and reference counting*)

2.1 Getting started

Pre-built Linux packages are distributed from [the libcbor website](#).

OS X users can use [Homebrew](#):

```
brew tap pjk/libcbor
brew install libcbor
```

For other platforms, you will need to compile it from source.

2.1.1 Building & installing libcbor

Prerequisites:

- C99 compiler
- [CMake](#) 2.8 or newer (might also be called `cmakesetup`, `cmake-gui` or `ccmake` depending on the installed version and system)
- C build system CMake can target (`make`, Apple Xcode, MinGW, ...)

Note: As of May 2015, not even the 2015 release candidate of Visual Studio supports C99. While CMake will be happy to generate a VS solution that you can play with, libcbor currently cannot be compiled using the MSVC toolchain. [ICC](#), [GCC](#) under [Cygwin](#), and [MinGW's](#) [GCC](#) will all work. The MinGW build process is described below.

Configuration options

A handful of configuration flags can be passed to `cmake`. The following table lists libcbor compile-time directives and several important generic flags.

Option	Meaning	Default	Possible values
CMAKE_C_COMPILER	C compiler to use	cc	gcc, clang, clang-3.5,...
CMAKE_INSTALL_PREFIX	Installation prefix	System-dependent	/usr/local/lib, ...
HUGE_FUZZ	<i>Fuzz test</i> with 8GB of data	OFF	ON, OFF
SANE_MALLOC	Assume malloc will refuse unreasonable allocations	OFF	ON, OFF
COVERAGE	Generate test coverage instrumentation	OFF	ON, OFF
WITH_TESTS	Build unit tests (see <i>Development</i>)	OFF	ON, OFF

The following configuration options will also be defined as macros[#]_ in `<cbor/common.h>` and can therefore be used in client code:

Option	Meaning	Default	Possible values
CBOR_CUSTOM_ALLOC	Enable custom allocator support	OFF	ON, OFF
CBOR_PRETTY_PRINTER	Include a pretty-printing routine	ON	ON, OFF
CBOR_BUFFER_GROWTH	Factor for buffer growth & shrinking	2	Decimals > 1

If you want to pass other custom configuration options, please refer to http://www.cmake.org/Wiki/CMake_Useful_Variables.

Building using make

CMake will generate a Makefile and other configuration files for the build. As a rule of thumb, you should configure the build *outside of the source tree* in order to keep different configurations isolated. If you are unsure where to execute the build, just use a temporary directory:

```
cd $(mktemp -d /tmp/cbor_build.XXXX)
```

Now, assuming you are in the directory where you want to build, execute the following to configure the build and run make

```
cmake -DCMAKE_BUILD_TYPE=Release path_to_libcbor_dir
make cbor cbor_shared
```

Both the shared (`libcbor.so`) and the static (`libcbor.a`) libraries should now be in the `src` subdirectory.

In order to install the libcbor headers and libraries, the usual

```
make install
```

is what your're looking for. Root permissions are required on most systems when using the default installation prefix.

Portability

libcbor is highly portable and works on both little- and big-endian systems regardless of the operating system. After building on an exotic platform, you might wish to verify the result by running the *test suite*. If you encounter any problems, please report them to the [issue tracker](#).

libcbor is known to successfully work on ARM Android devices. Cross-compilation is possible with `arm-linux-gnueabi-gcc`.

2.1.2 Linking with libcbor

If you include and linker paths include the directories to which libcbor has been installed, compiling programs that uses libcbor requires no extra considerations.

You can verify that everything has been set up properly by creating a file with the following contents

```
#include <cbor.h>
#include <stdio.h>

int main(int argc, char * argv[])
{
    printf("Hello from libcbor %s\n", CBOR_VERSION);
}
```

and compiling it

```
cc hello_cbor.c -lcbor -o hello_cbor
```

libcbor also comes with `pkg-config` support. If you install libcbor with a custom prefix, you can use `pkg-config` to resolve the headers and objects:

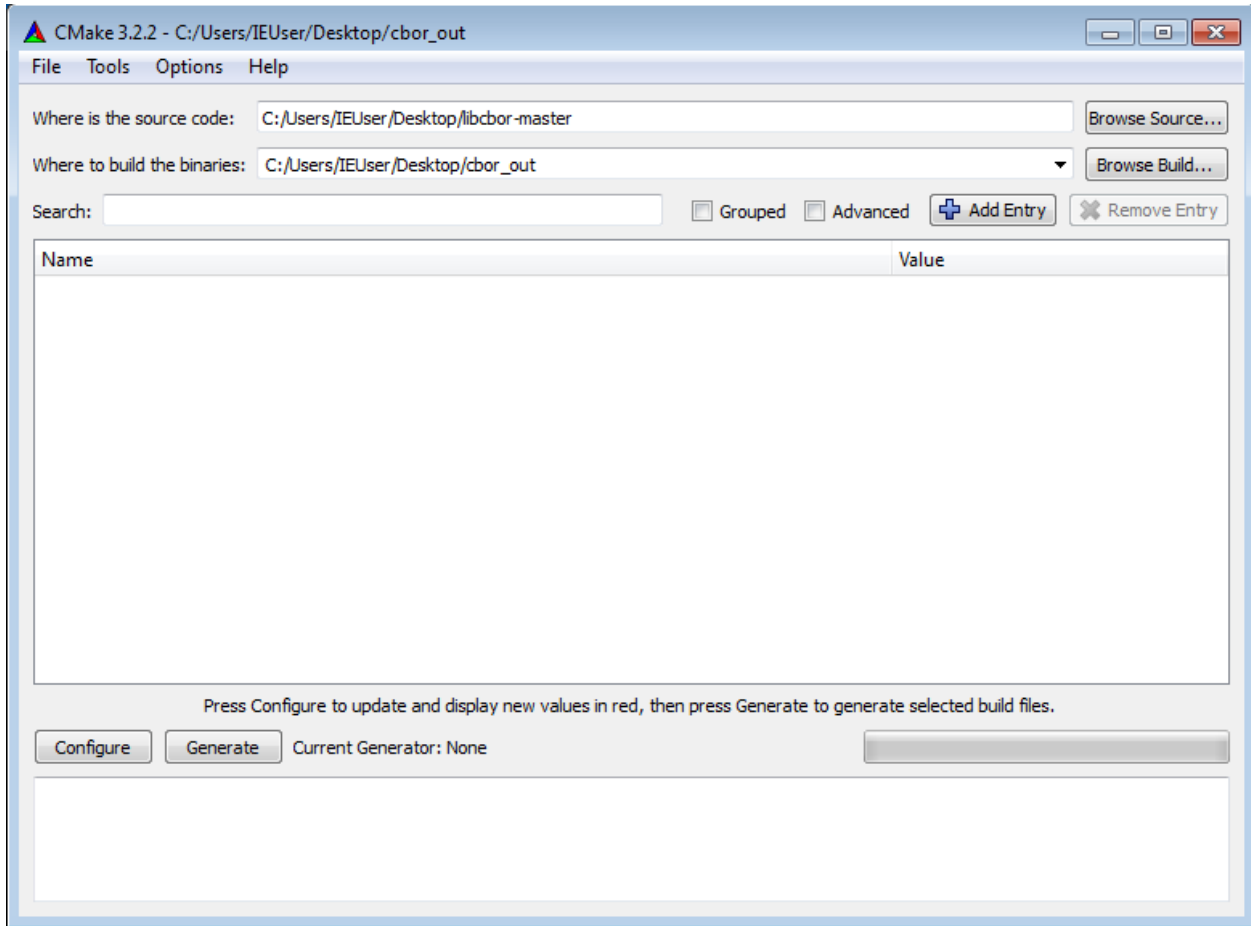
```
cc $(pkg-config --cflags libcbor) hello_cbor.c $(pkg-config --libs libcbor) -o hello_
↪cbor
```

2.1.3 MinGW build instructions

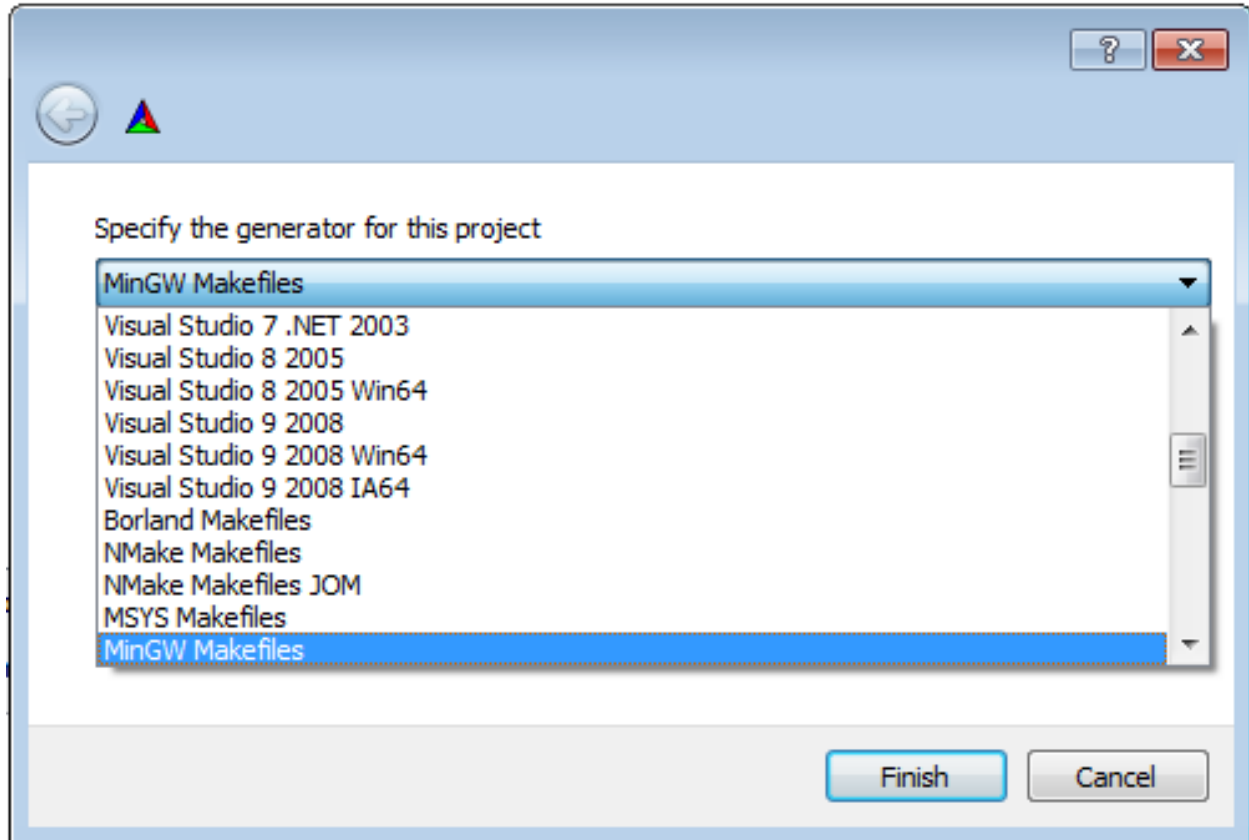
Prerequisites:

- MinGW
- CMake GUI

First of all, create a folder that will be used for the output. For this demonstration, we will use `cbor_out`. Start CMake and select the source path and the destination folder.



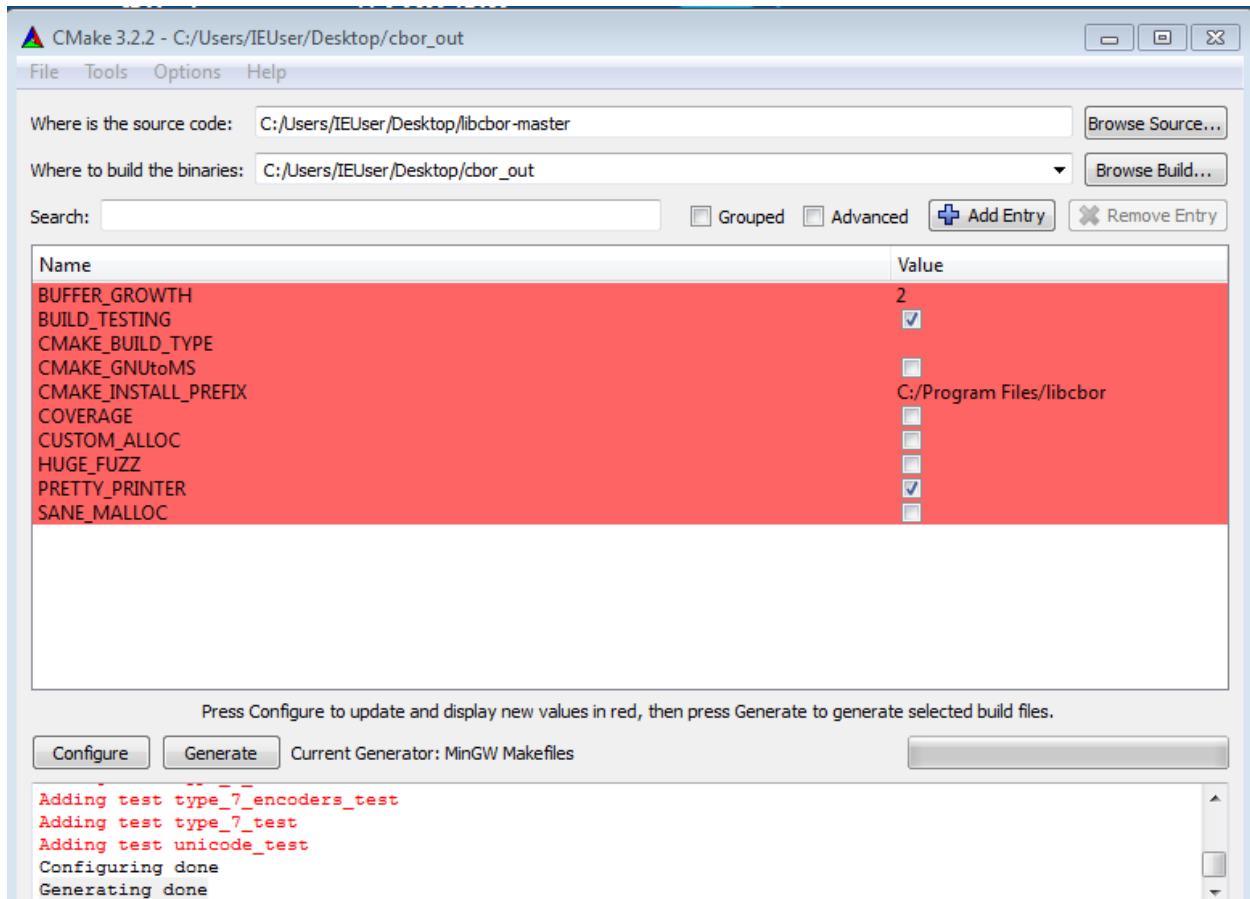
Then hit the 'Configure' button. You will be prompted to select the build system:



Choose MinGW and confirm.

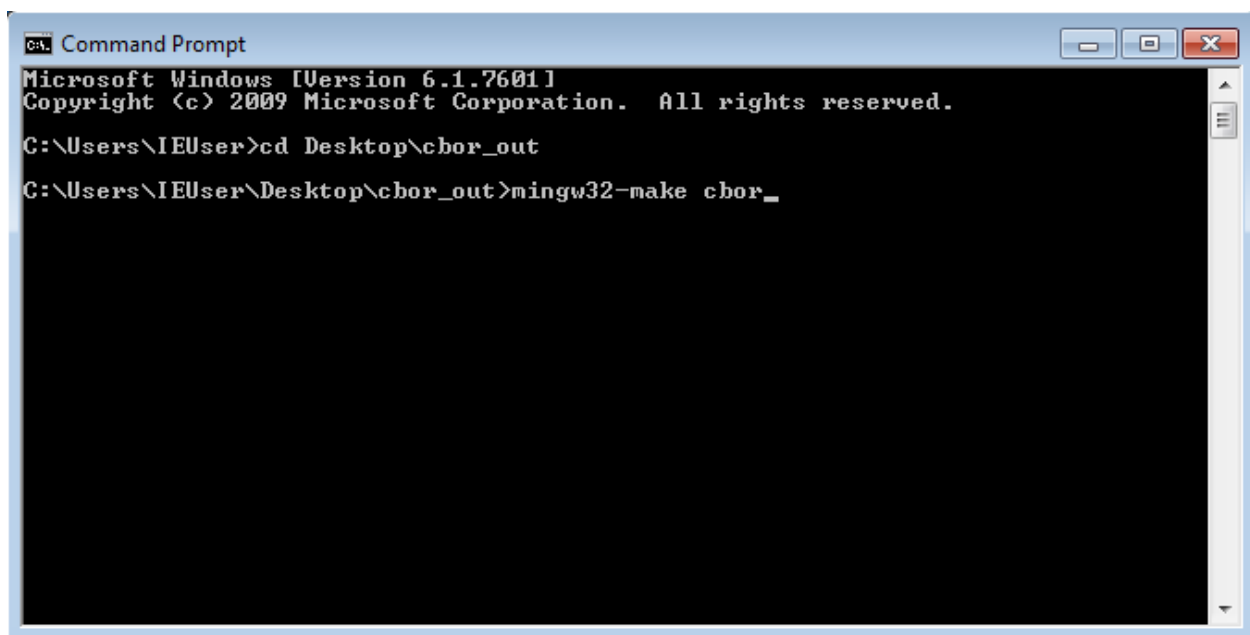
Note: If you select Visual Studio at this point, a MSVC project will be generated for you. This is useful if you just want to browse through the source code.

You can then adjust the build options. The defaults will work just fine. Hit 'Generate' when you are done.



You can then adjust the build options. The defaults will work just fine. Hit 'Generate' when you are done.

Open the shell, navigate to the output directory, and run `mingw32-make cbor cbor_shared`.



`libcbor` will be built and your `.dll` should be ready at this point

```

ca. Command Prompt
C:\Users\IEUser\Desktop\cbor_out>mingw32-make cbor
Scanning dependencies of target cbor
[ 5%] Building C object src/CMakeFiles/cbor.dir/cbor.c.obj
[ 11%] Building C object src/CMakeFiles/cbor.dir/cbor/streaming.c.obj
[ 16%] Building C object src/CMakeFiles/cbor.dir/cbor/internal/encoders.c.obj
[ 22%] Building C object src/CMakeFiles/cbor.dir/cbor/internal/builder_callbacks
.c.obj
[ 27%] Building C object src/CMakeFiles/cbor.dir/cbor/internal/loaders.c.obj
[ 33%] Building C object src/CMakeFiles/cbor.dir/cbor/internal/stack.c.obj
[ 38%] Building C object src/CMakeFiles/cbor.dir/cbor/internal/unicode.c.obj
[ 44%] Building C object src/CMakeFiles/cbor.dir/cbor/encoding.c.obj
[ 50%] Building C object src/CMakeFiles/cbor.dir/cbor/serialization.c.obj
[ 55%] Building C object src/CMakeFiles/cbor.dir/cbor/arrays.c.obj
[ 61%] Building C object src/CMakeFiles/cbor.dir/cbor/common.c.obj
[ 66%] Building C object src/CMakeFiles/cbor.dir/cbor/floats_ctrls.c.obj
[ 72%] Building C object src/CMakeFiles/cbor.dir/cbor/bytestrings.c.obj
[ 77%] Building C object src/CMakeFiles/cbor.dir/cbor/callbacks.c.obj
[ 83%] Building C object src/CMakeFiles/cbor.dir/cbor/strings.c.obj
[ 88%] Building C object src/CMakeFiles/cbor.dir/cbor/maps.c.obj
[ 94%] Building C object src/CMakeFiles/cbor.dir/cbor/tags.c.obj
[100%] Building C object src/CMakeFiles/cbor.dir/cbor/ints.c.obj
Linking C static library libcbor.a
[100%] Built target cbor

C:\Users\IEUser\Desktop\cbor_out>

```

Feel free to also try building and running some of the examples, e.g. `mingw32-make sort`

```

ca. Command Prompt
[ 66%] Building C object src/CMakeFiles/cbor.dir/cbor/floats_ctrls.c.obj
[ 72%] Building C object src/CMakeFiles/cbor.dir/cbor/bytestrings.c.obj
[ 77%] Building C object src/CMakeFiles/cbor.dir/cbor/callbacks.c.obj
[ 83%] Building C object src/CMakeFiles/cbor.dir/cbor/strings.c.obj
[ 88%] Building C object src/CMakeFiles/cbor.dir/cbor/maps.c.obj
[ 94%] Building C object src/CMakeFiles/cbor.dir/cbor/tags.c.obj
[100%] Building C object src/CMakeFiles/cbor.dir/cbor/ints.c.obj
Linking C static library libcbor.a
[100%] Built target cbor

C:\Users\IEUser\Desktop\cbor_out>mingw32-make sort
[ 94%] Built target cbor
Scanning dependencies of target sort
[100%] Building C object examples/CMakeFiles/sort.dir/sort.c.obj
Linking C executable sort.exe
[100%] Built target sort

C:\Users\IEUser\Desktop\cbor_out>examples\sort
[CBOR_TYPE_ARRAY] Definite, size: 4
[CBOR_TYPE_UINT] Width: 1B, Value: 1
[CBOR_TYPE_UINT] Width: 1B, Value: 2
[CBOR_TYPE_UINT] Width: 1B, Value: 3
[CBOR_TYPE_UINT] Width: 1B, Value: 4

C:\Users\IEUser\Desktop\cbor_out>

```

2.1.4 Troubleshooting

cbor.h not found: The headers directory is probably not in your include path. First, verify the installation location by checking the installation log. If you used `make`, it will look something like

```

...
-- Installing: /usr/local/include/cbor
-- Installing: /usr/local/include/cbor/callbacks.h
-- Installing: /usr/local/include/cbor/encoding.h
...

```

Make sure that `CMAKE_INSTALL_PREFIX` (if you provided it) was correct. Including the path path during compilation should suffice, e.g.:

```
cc -I/usr/local/include hello_cbor.c -lcbor -o hello_cbor
```

cannot find -lcbor during linking: Most likely the same problem as before. Include the installation directory in the linker shared path using `-R`, e.g.:

```
cc -Wl,-rpath,/usr/local/lib -lcbor -o hello_cbor
```

shared library missing during execution: Verify the linkage using `ldd`, `otool`, or similar and adjust the compilation directives accordingly:

```
ldd hello_cbor
linux-vdso.so.1 => (0x00007ffe85585000)
libcbor.so => /usr/local/lib/libcbor.so (0x00007f9af69da000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9af65eb000)
/lib64/ld-linux-x86-64.so.2 (0x00007f9af6be9000)
```

compilation failed: If your compiler supports C99 yet the compilation has failed, please report the issue to the [issue tracker](#).

2.2 Usage & preliminaries

2.2.1 Version information

libcbor exports its version using three self-explanatory macros:

- `CBOR_MAJOR_VERSION`
- `CBOR_MINOR_VERSION`
- `CBOR_PATCH_VERSION`

The `CBOR_VERSION` is a string concatenating these three identifiers into one (e.g. `0.2.0`).

In order to simplify version comparisons, the version is also exported as

```
#define CBOR_HEX_VERSION ((CBOR_MAJOR_VERSION << 16) | (CBOR_MINOR_VERSION << 8) |  
↳ CBOR_PATCH_VERSION)
```

Since macros are difficult to work with through FFIs, the same information is also available through three `uint8_t` constants, namely

- `cbor_major_version`
- `cbor_minor_version`
- `cbor_patch_version`

2.2.2 Headers to include

The `cbor.h` header includes all the symbols. If, for any reason, you don't want to include all the exported symbols, feel free to use just some of the `cbor/*.h` headers:

- `cbor/arrays.h` - *Type 4 – Arrays*
- `cbor/bytestrings.h` - *Type 2 – Byte strings*

- `cbor/callbacks.h` - Callbacks used for *Decoding*
- `cbor/common.h` - Common utilities - always transitively included
- `cbor/data.h` - Data types definitions - always transitively included
- `cbor/encoding.h` - Streaming encoders for *Encoding*
- `cbor/floats_ctrls.h` - *Type 7 – Floats & control tokens*
- `cbor/ints.h` - *Types 0 & 1 – Positive and negative integers*
- `cbor/maps.h` - *Type 5 – Maps*
- `cbor/serialization.h` - High level serialization such as `cbor_serialize()`
- `cbor/streaming.h` - Home of `cbor_stream_decode()`
- `cbor/strings.h` - *Type 3 – UTF-8 strings*
- `cbor/tags.h` - *Type 6 – Semantic tags*

2.2.3 Using libcbor

If you want to get more familiar with CBOR, we recommend the cbor.io website. Once you get the grasp of what is it CBOR does, the examples (located in the `examples` directory) should give you a good feel of the API. The *API documentation* should then provide with all the information you may need.

Creating and serializing items

```
#include "cbor.h"
#include <stdio.h>

int main(int argc, char * argv[])
{
    /* Preallocate the map structure */
    cbor_item_t * root = cbor_new_definite_map(2);
    /* Add the content */
    cbor_map_add(root, (struct cbor_pair) {
        .key = cbor_move(cbor_build_string("Is CBOR awesome?")),
        .value = cbor_move(cbor_build_bool(true))
    });
    cbor_map_add(root, (struct cbor_pair) {
        .key = cbor_move(cbor_build_uint8(42)),
        .value = cbor_move(cbor_build_string("Is the answer"))
    });
    /* Output: `length` bytes of data in the `buffer` */
    unsigned char * buffer;
    size_t buffer_size, length = cbor_serialize_alloc(root, &buffer, &buffer_size);

    fwrite(buffer, 1, length, stdout);
    free(buffer);

    fflush(stdout);
    cbor_decref(&root);
}
```

Reading serialized data

```
#include "cbor.h"
#include <stdio.h>
```

```
/*
 * Reads data from a file. Example usage:
 * $ ./examples/readfile examples/data/nested_array.cbor
 */

int main(int argc, char * argv[])
{
    FILE * f = fopen(argv[1], "rb");
    fseek(f, 0, SEEK_END);
    size_t length = (size_t)ftell(f);
    fseek(f, 0, SEEK_SET);
    unsigned char * buffer = malloc(length);
    fread(buffer, length, 1, f);

    /* Assuming `buffer` contains `info.st_size` bytes of input data */
    struct cbor_load_result result;
    cbor_item_t * item = cbor_load(buffer, length, &result);
    /* Pretty-print the result */
    cbor_describe(item, stdout);
    fflush(stdout);
    /* Deallocate the result */
    cbor_decref(&item);

    fclose(f);
}
```

Using the streaming parser

```
#include "cbor.h"
#include <stdio.h>
#include <string.h>

/*
 * Illustrates how one might skim through a map (which is assumed to have
 * string keys and values only), looking for the value of a specific key
 *
 * Use the examples/data/map.cbor input to test this.
 */

const char * key = "a secret key";
bool key_found = false;

void find_string(void * _ctx, cbor_data buffer, size_t len)
{
    if (key_found) {
        printf("Found the value: %s\n", (int) len, buffer);
        key_found = false;
    } else if (len == strlen(key)) {
        key_found = (memcmp(key, buffer, len) == 0);
    }
}

int main(int argc, char * argv[])
{
    FILE * f = fopen(argv[1], "rb");
    fseek(f, 0, SEEK_END);
    size_t length = (size_t)ftell(f);
```

```

fseek(f, 0, SEEK_SET);
unsigned char * buffer = malloc(length);
fread(buffer, length, 1, f);

struct cbor_callbacks callbacks = cbor_empty_callbacks;
struct cbor_decoder_result decode_result;
size_t bytes_read = 0;
callbacks.string = find_string;
while (bytes_read < length) {
    decode_result = cbor_stream_decode(buffer + bytes_read,
                                      length - bytes_read,
                                      &callbacks, NULL);

    bytes_read += decode_result.read;
}

fclose(f);
}

```

2.3 API

The data API is centered around `cbor_item_t`, a generic handle for any CBOR item. There are functions to

- create items,
- set items' data,
- parse serialized data into items,
- manage, move, and links item together.

The single most important thing to keep in mind is: **`cbor_item_t` is an opaque type and should only be manipulated using the appropriate functions!** Think of it as an object.

The *libcbor* API closely follows the semantics outlined by [CBOR standard](#). This part of the documentation provides a short overview of the CBOR constructs, as well as a general introduction to the *libcbor* API. Remaining reference can be found in the following files structured by data types.

The API is designed to allow both very tight control & flexibility and general convenience with sane defaults.¹ For example, client with very specific requirements (constrained environment, custom application protocol built on top of CBOR, etc.) may choose to take full control (and responsibility) of memory and data structures management by interacting directly with the decoder. Other clients might want to take control of specific aspects (streamed collections, hash maps storage), but leave other responsibilities to *libcbor*. More general clients might prefer to be abstracted away from all aforementioned details and only be presented complete data structures.

libcbor provides

- stateless encoders and decoders
- encoding and decoding *drivers*, routines that coordinate encoding and decoding of complex structures
- data structures to represent and transform CBOR structures
- routines for building and manipulating these structures
- utilities for inspection and debugging

¹ <http://softwareengineering.vazexqi.com/files/pattern.html>

2.3.1 Types of items

Every *cbor_item_t* has a *cbor_type* associated with it - these constants correspond to the types specified by the CBOR standard:

enum *cbor_type*

Specifies the Major type of *cbor_item_t*.

Values:

CBOR_TYPE_UINT

0 - positive integers

CBOR_TYPE_NEGINT

1 - negative integers

CBOR_TYPE_BYTESTRING

2 - byte strings

CBOR_TYPE_STRING

3 - strings

CBOR_TYPE_ARRAY

4 - arrays

CBOR_TYPE_MAP

5 - maps

CBOR_TYPE_TAG

6 - tags

CBOR_TYPE_FLOAT_CTRL

7 - decimals and special values (true, false, nil, ...)

To find out the type of an item, one can use

cbor_type **cbor_typeof** (**const** *cbor_item_t* **item*)

Get the type of the item.

Return The type

Parameters

- *item*[borrow]:

Please note the distinction between functions like *cbor_isa_uint()* and *cbor_is_int()*. The following functions work solely with the major type value.

Binary queries

Alternatively, there are functions to query each particular type.

Warning: Passing an invalid *cbor_item_t* reference to any of these functions results in undefined behavior.

bool **cbor_isa_uint** (**const** *cbor_item_t* **item*)

Does the item have the appropriate major type?

Return Is the item an *CBOR_TYPE_UINT*?

Parameters

- `item[borrow]`: the item

bool `cbor_isa_negint` (`const cbor_item_t *item`)
Does the item have the appropriate major type?

Return Is the item a `CBOR_TYPE_NEGINT`?

Parameters

- `item[borrow]`: the item

bool `cbor_isa_bytestring` (`const cbor_item_t *item`)
Does the item have the appropriate major type?

Return Is the item a `CBOR_TYPE_BYTESTRING`?

Parameters

- `item[borrow]`: the item

bool `cbor_isa_string` (`const cbor_item_t *item`)
Does the item have the appropriate major type?

Return Is the item a `CBOR_TYPE_STRING`?

Parameters

- `item[borrow]`: the item

bool `cbor_isa_array` (`const cbor_item_t *item`)
Does the item have the appropriate major type?

Return Is the item an `CBOR_TYPE_ARRAY`?

Parameters

- `item[borrow]`: the item

bool `cbor_isa_map` (`const cbor_item_t *item`)
Does the item have the appropriate major type?

Return Is the item a `CBOR_TYPE_MAP`?

Parameters

- `item[borrow]`: the item

bool `cbor_isa_tag` (`const cbor_item_t *item`)
Does the item have the appropriate major type?

Return Is the item a `CBOR_TYPE_TAG`?

Parameters

- `item[borrow]`: the item

bool `cbor_isa_float_ctrl` (`const cbor_item_t *item`)
Does the item have the appropriate major type?

Return Is the item a *CBOR_TYPE_FLOAT_CTRL*?

Parameters

- `item[borrow]`: the item

Logical queries

These functions provide information about the item type from a more high-level perspective

bool **cbor_is_int** (**const** *cbor_item_t* **item*)

Is the item an integer, either positive or negative?

Return Is the item an integer, either positive or negative?

Parameters

- `item[borrow]`: the item

bool **cbor_is_float** (**const** *cbor_item_t* **item*)

Is the item an a floating point number?

Return Is the item a floating point number?

Parameters

- `item[borrow]`: the item

bool **cbor_is_bool** (**const** *cbor_item_t* **item*)

Is the item an a boolean?

Return Is the item a boolean?

Parameters

- `item[borrow]`: the item

bool **cbor_is_null** (**const** *cbor_item_t* **item*)

Does this item represent null

Warning: This is in no way related to the value of the pointer. Passing a null pointer will most likely result in a crash.

Return Is the item (CBOR logical) null?

Parameters

- `item[borrow]`: the item

bool **cbor_is_undef** (**const** *cbor_item_t* **item*)

Does this item represent undefined

Warning: Care must be taken to distinguish nulls and undefined values in C.

Return Is the item (CBOR logical) undefined?

Parameters

- `item[borrow]`: the item

2.3.2 Memory management and reference counting

Due to the nature of its domain, *libcbor* will need to work with heap memory. The stateless decoder and encoder don't allocate any memory.

If you have specific requirements, you should consider rolling your own driver for the stateless API.

Using custom allocator

libcbor gives you with the ability to provide your own implementations of `malloc`, `realloc`, and `free`. This can be useful if you are using a custom allocator throughout your application, or if you want to implement custom policies (e.g. tighter restrictions on the amount of allocated memory).

In order to use this feature, *libcbor* has to be compiled with the appropriate flags. You can verify the configuration using the `CBOR_CUSTOM_ALLOC` macro. A simple usage might be as follows:

```
#if CBOR_CUSTOM_ALLOC
    cbor_set_allocs(malloc, realloc, free);
#else
    #error "libcbor built with support for custom allocation is required"
#endif
```

void **cbor_set_allocs** (`_cbor_malloc_t custom_malloc`, `_cbor_realloc_t custom_realloc`, `_cbor_free_t custom_free`)

Sets the memory management routines to use.

Only available when `CBOR_CUSTOM_ALLOC` is truthy

Warning: This function modifies the global state and should therefore be used accordingly. Changing the memory handlers while allocated items exist will result in a `free/malloc` mismatch. This function is not thread safe with respect to both itself and all the other *libcbor* functions that work with the heap.

Note: `realloc` implementation must correctly support `NULL` reallocation (see e.g. <http://en.cppreference.com/w/c/memory/realloc>)

Parameters

- `custom_malloc`: `malloc` implementation
- `custom_realloc`: `realloc` implementation
- `custom_free`: `free` implementation

Reference counting

As CBOR items may require complex cleanups at the end of their lifetime, there is a reference counting mechanism in place. This also enables very simple GC when integrating *libcbor* into managed environment. Every item starts its life (by either explicit creation, or as a result of parsing) with reference count set to 1. When the refcount reaches zero, it will be destroyed.

Items containing nested items will be destroyed recursively - refcount of every nested item will be decreased by one. The destruction is synchronous and renders any pointers to items with refcount zero invalid immediately after calling the `cbor_decref()`.

`cbor_item_t*` **cbor_incref** (`cbor_item_t` *item)

Increases the reference count by one.

No dependent items are affected.

Return the input reference

Parameters

- `item[incref]`: item the item

void **cbor_decref** (`cbor_item_t` **item)

Decreases the reference count by one, deallocating the item if needed.

In case the item is deallocated, the reference count of any dependent items is adjusted accordingly in a recursive manner.

Parameters

- `item[take]`: the item. Set to NULL if deallocated

void **cbor_intermediate_decref** (`cbor_item_t` *item)

Decreases the reference count by one, deallocating the item if needed.

Convenience wrapper for `cbor_decref` when its set-to-null behavior is not needed

Parameters

- `item[take]`: the item

size_t **cbor_refcount** (`const cbor_item_t` *item)

Get the reference count.

Warning: This does *not* account for transitive references.

Return the reference count

Parameters

- `item[borrow]`: the item

`cbor_item_t*` **cbor_move** (`cbor_item_t` *item)

Provides CPP-like move construct.

Decreases the reference count by one, but does not deallocate the item even if its refcount reaches zero. This is useful for passing intermediate values to functions that increase reference count. Should only be used with functions that `incref` their arguments.

Warning: If the item is moved without correctly increasing the reference count afterwards, the memory will be leaked.

Return the item with reference count decreased by one

Parameters

- `item[take]`: the item

`cbor_item_t *cbor_copy (cbor_item_t *item)`

Deep copy of an item.

All the reference counts in the new structure are set to one.

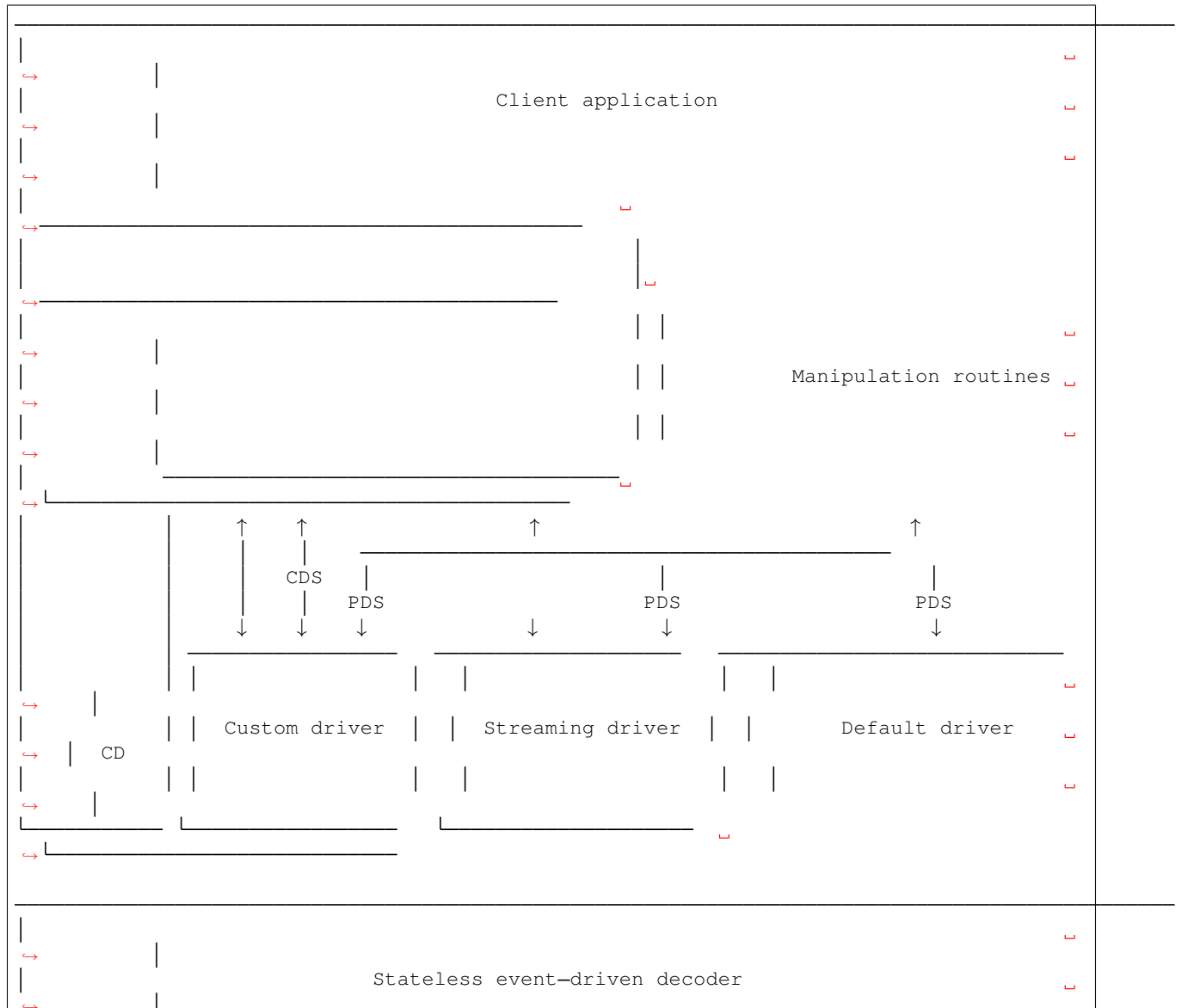
Return new CBOR deep copy

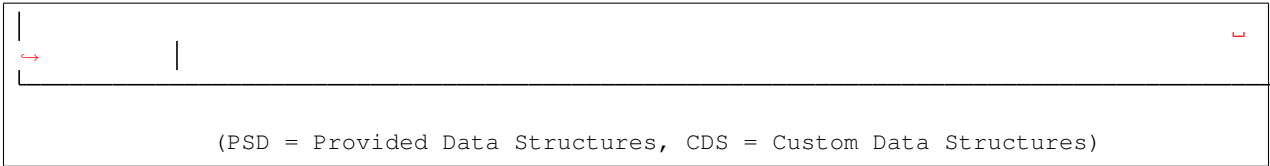
Parameters

- `item[borrow]`: item to copy

2.3.3 Decoding

The following diagram illustrates the relationship among different parts of libcbor from the decoding standpoint.





This section will deal with the API that is labeled as the “Default driver” in the diagram. That is, routines that decode complete libcbor data items

`cbor_item_t *cbor_load` (`cbor_data source`, `size_t source_size`, **struct** `cbor_load_result *result`)

Loads data item from a buffer.

Return new CBOR item or NULL on failure. In that case, `result` contains location and description of the error.

Parameters

- `source`: The buffer
- `source_size`:
- `result[out]`: Result indicator. `CBOR_ERR_NONE` on success

Associated data structures

enum `cbor_error_code`

Possible decoding errors.

Values:

`CBOR_ERR_NONE`

`CBOR_ERR_NOTENOUGHDATA`

`CBOR_ERR_NODATA`

`CBOR_ERR_MALFORMATED`

`CBOR_ERR_MEMERROR`

Memory error - item allocation failed.

Is it too big for your allocator?

`CBOR_ERR_SYNTAXERROR`

Stack parsing algorithm failed.

struct `cbor_load_result`

High-level decoding result.

Public Members

struct `cbor_error` **error**

Error indicator.

`size_t` **read**

Number of bytes read.

struct `cbor_error`

High-level decoding error.

Public Members

`size_t position`
Aproximate position.

`cbor_error_code code`
Description.

2.3.4 Encoding

The easiest way to encode data items is using the `cbor_serialize()` or `cbor_serialize_alloc()` functions:

`size_t cbor_serialize(const cbor_item_t *item, cbor_mutable_data buffer, size_t buffer_size)`
Serialize the given item.

Return Length of the result. 0 on failure.

Parameters

- `item[borrow]`: A data item
- `buffer`: Buffer to serialize to
- `buffer_size`: Size of the buffer

`size_t cbor_serialize_alloc(const cbor_item_t *item, cbor_mutable_data *buffer, size_t *buffer_size)`
Serialize the given item, allocating buffers as needed.

Warning: It is your responsibility to free the buffer using an appropriate `free` implementation.

Return Length of the result. 0 on failure, in which case `buffer` is NULL.

Parameters

- `item[borrow]`: A data item
- `buffer[out]`: Buffer containing the result
- `buffer_size[out]`: Size of the buffer

Type-specific serializers

In case you know the type of the item you want to serialize beforehand, you can use one of the type-specific serializers.

Note: Unless compiled in debug mode, these do not verify the type. Passing an incorrect item will result in an undefined behavior.

`size_t cbor_serialize_uint(const cbor_item_t *, cbor_mutable_data, size_t)`
Serialize an uint.

Return Length of the result. 0 on failure.

Parameters

- `item[borrow]`: A uint
- `buffer`: Buffer to serialize to
- `buffer_size`: Size of the buffer

`size_t cbor_serialize_negint (const cbor_item_t*, cbor_mutable_data, size_t)`

Serialize a negint.

Return Length of the result. 0 on failure.

Parameters

- `item[borrow]`: A negint
- `buffer`: Buffer to serialize to
- `buffer_size`: Size of the buffer

`size_t cbor_serialize_bytestring (const cbor_item_t*, cbor_mutable_data, size_t)`

Serialize a bytestring.

Return Length of the result. 0 on failure.

Parameters

- `item[borrow]`: A bytestring
- `buffer`: Buffer to serialize to
- `buffer_size`: Size of the buffer

`size_t cbor_serialize_string (const cbor_item_t*, cbor_mutable_data, size_t)`

Serialize a string.

Return Length of the result. 0 on failure.

Parameters

- `item[borrow]`: A string
- `buffer`: Buffer to serialize to
- `buffer_size`: Size of the buffer

`size_t cbor_serialize_array (const cbor_item_t*, cbor_mutable_data, size_t)`

Serialize an array.

Return Length of the result. 0 on failure.

Parameters

- `item[borrow]`: An array
- `buffer`: Buffer to serialize to
- `buffer_size`: Size of the buffer

`size_t cbor_serialize_map (const cbor_item_t*, cbor_mutable_data, size_t)`

Serialize a map.

Return Length of the result. 0 on failure.

Parameters

- `item[borrow]`: A map
- `buffer`: Buffer to serialize to
- `buffer_size`: Size of the buffer

`size_t cbor_serialize_tag (const cbor_item_t*, cbor_mutable_data, size_t)`
Serialize a tag.

Return Length of the result. 0 on failure.

Parameters

- `item[borrow]`: A tag
- `buffer`: Buffer to serialize to
- `buffer_size`: Size of the buffer

`size_t cbor_serialize_float_ctrl (const cbor_item_t*, cbor_mutable_data, size_t)`
Serialize a.

Return Length of the result. 0 on failure.

Parameters

- `item[borrow]`: A float or ctrl
- `buffer`: Buffer to serialize to
- `buffer_size`: Size of the buffer

2.3.5 Types 0 & 1 – Positive and negative integers

CBOR has two types of integers – positive (which may be effectively regarded as unsigned), and negative. There are four possible widths for an integer – 1, 2, 4, or 8 bytes. These are represented by

`enum cbor_int_width`

Possible widths of `CBOR_TYPE_UINT` items.

Values:

`CBOR_INT_8`

`CBOR_INT_16`

`CBOR_INT_32`

`CBOR_INT_64`

Type 0 - positive integers

Corresponding <code>cbor_type</code>	<code>CBOR_TYPE_UINT</code>
Number of allocations	One per lifetime
Storage requirements	<code>sizeof(cbor_item_t) + sizeof(uint*_t)</code>

Note: once a positive integer has been created, its width *cannot* be changed.

Type 1 - negative integers

Corresponding <i>cbor_type</i>	CBOR_TYPE_NEGINT
Number of allocations	One per lifetime
Storage requirements	<code>sizeof(cbor_item_t) + sizeof(uint*_t)</code>

Note: once a positive integer has been created, its width *cannot* be changed.

Type 0 & 1

Due to their largely similar semantics, the following functions can be used for both Type 0 and Type 1 items. One can convert between them freely using *the conversion functions*.

Actual Type of the integer can be checked using item types API.

An integer item is created with one of the four widths. Because integers' storage is bundled together with the handle, the width cannot be changed over its lifetime.

Warning: Due to the fact that CBOR negative integers represent integers in the range $[-1, -2^N]$, `cbor_set_uint` API is somewhat counter-intuitive as the resulting logical value is 1 less. This behavior is necessary in order to permit uniform manipulation with the full range of permitted values. For example, the following snippet

```
cbor_item_t * item = cbor_new_int8();
cbor_mark_negint(item);
cbor_set_uint8(0);
```

will produce an item with the logical value of -1 . There is, however, an upside to this as well: There is only one representation of zero.

Building new items

cbor_item_t * **cbor_build_uint8** (uint8_t value)

Constructs a new positive integer.

Return new positive integer

Parameters

- value: the value to use

cbor_item_t * **cbor_build_uint16** (uint16_t value)

Constructs a new positive integer.

Return new positive integer

Parameters

- value: the value to use

cbor_item_t * **cbor_build_uint32** (uint32_t value)

Constructs a new positive integer.

Return new positive integer

Parameters

- value: the value to use

*cbor_item_t****cbor_build_uint64** (*uint64_t value*)

Constructs a new positive integer.

Return new positive integer

Parameters

- value: the value to use

Retrieving values

uint8_t **cbor_get_uint8** (*const cbor_item_t *item*)

Extracts the integer value.

Return the value

Parameters

- item[borrow]: positive or negative integer

uint16_t **cbor_get_uint16** (*const cbor_item_t *item*)

Extracts the integer value.

Return the value

Parameters

- item[borrow]: positive or negative integer

uint32_t **cbor_get_uint32** (*const cbor_item_t *item*)

Extracts the integer value.

Return the value

Parameters

- item[borrow]: positive or negative integer

uint64_t **cbor_get_uint64** (*const cbor_item_t *item*)

Extracts the integer value.

Return the value

Parameters

- item[borrow]: positive or negative integer

Setting values

void **cbor_set_uint8** (*cbor_item_t *item*, *uint8_t value*)

Assigns the integer value.

Parameters

- `item[borrow]`: positive or negative integer item
- `value`: the value to assign. For negative integer, the logical value is `-value - 1`

void `cbor_set_uint16` (*cbor_item_t* **item*, *uint16_t value*)
Assigns the integer value.

Parameters

- `item[borrow]`: positive or negative integer item
- `value`: the value to assign. For negative integer, the logical value is `-value - 1`

void `cbor_set_uint32` (*cbor_item_t* **item*, *uint32_t value*)
Assigns the integer value.

Parameters

- `item[borrow]`: positive or negative integer item
- `value`: the value to assign. For negative integer, the logical value is `-value - 1`

void `cbor_set_uint64` (*cbor_item_t* **item*, *uint64_t value*)
Assigns the integer value.

Parameters

- `item[borrow]`: positive or negative integer item
- `value`: the value to assign. For negative integer, the logical value is `-value - 1`

Dealing with width

cbor_int_width `cbor_int_get_width` (*const cbor_item_t* **item*)
Queries the integer width.

Return the width

Parameters

- `item[borrow]`: positive or negative integer item

Dealing with signedness

void `cbor_mark_uint` (*cbor_item_t* **item*)
Marks the integer item as a positive integer.

The data value is not changed

Parameters

- `item[borrow]`: positive or negative integer item

void `cbor_mark_negint` (*cbor_item_t* **item*)
Marks the integer item as a negative integer.

The data value is not changed

Parameters

- `item[borrow]`: positive or negative integer item

Creating new items

`cbor_item_t *cbor_new_int8()`

Allocates new integer with 1B width.

The width cannot be changed once allocated

Return new positive integer. The value is not initialized.

`cbor_item_t *cbor_new_int16()`

Allocates new integer with 2B width.

The width cannot be changed once allocated

Return new positive integer. The value is not initialized.

`cbor_item_t *cbor_new_int32()`

Allocates new integer with 4B width.

The width cannot be changed once allocated

Return new positive integer. The value is not initialized.

`cbor_item_t *cbor_new_int64()`

Allocates new integer with 8B width.

The width cannot be changed once allocated

Return new positive integer. The value is not initialized.

2.3.6 Type 2 – Byte strings

CBOR byte strings are just (ordered) series of bytes without further interpretation (unless there is a *tag*). Byte string's length may or may not be known during encoding. These two kinds of byte strings can be distinguished using `cbor_bytestring_is_definite()` and `cbor_bytestring_is_indefinite()` respectively.

In case a byte string is indefinite, it is encoded as a series of definite byte strings. These are called “chunks”. For example, the encoded item

0xf5	Start indefinite byte string
0x41	Byte string (1B long)
0x00	
0x41	Byte string (1B long)
0xff	
0xff	"Break" control token

represents two bytes, 0x00 and 0xff. This on one hand enables streaming messages even before they are fully generated, but on the other hand it adds more complexity to the client code.

Corresponding <i>cbor_type</i>	CBOR_TYPE_BYTESTRING
Number of allocations (definite)	One plus any manipulations with the data
Number of allocations (indefinite)	One plus logarithmically many reallocations relative to chunk count
Storage requirements (definite)	<code>sizeof(cbor_item_t) + length(handle)</code>
Storage requirements (indefinite)	<code>sizeof(cbor_item_t) * (1 + chunk_count) + chunks</code>

Streaming indefinite byte strings

Please refer to *Streaming & indefinite items*.

Getting metadata

`size_t cbor_bytestring_length(const cbor_item_t *item)`

Returns the length of the binary data.

For definite byte strings only

Return length of the binary data. Zero if no chunk has been attached yet

Parameters

- `item[borrow]`: a definite bytestring

`bool cbor_bytestring_is_definite(const cbor_item_t *item)`

Is the byte string definite?

Return Is the byte string definite?

Parameters

- `item[borrow]`: a byte string

`bool cbor_bytestring_is_indefinite(const cbor_item_t *item)`

Is the byte string indefinite?

Return Is the byte string indefinite?

Parameters

- `item[borrow]`: a byte string

`size_t cbor_bytestring_chunk_count(const cbor_item_t *item)`

Get the number of chunks this string consist of.

Return The chunk count. 0 for freshly created items.

Parameters

- `item[borrow]`: A indefinite bytestring

Reading data

`cbor_mutable_data cbor_bytestring_handle (const cbor_item_t *item)`

Get the handle to the binary data.

Definite items only. Modifying the data is allowed. In that case, the caller takes responsibility for the effect on items this item might be a part of

Return The address of the binary data. NULL if no data have been assigned yet.

Parameters

- `item[borrow]`: A definite byte string

`cbor_item_t **cbor_bytestring_chunks_handle (const cbor_item_t *item)`

Get the handle to the array of chunks.

Manipulations with the memory block (e.g. sorting it) are allowed, but the validity and the number of chunks must be retained.

Return array of `cbor_bytestring_chunk_count` definite bytestrings

Parameters

- `item[borrow]`: A indefinite byte string

Creating new items

`cbor_item_t *cbor_new_definite_bytestring ()`

Creates a new definite byte string.

The handle is initialized to NULL and length to 0

Return new definite bytestring. NULL on malloc failure.

`cbor_item_t *cbor_new_indefinite_bytestring ()`

Creates a new indefinite byte string.

The chunks array is initialized to NULL and chunkcount to 0

Return new indefinite bytestring. NULL on malloc failure.

Building items

`cbor_item_t *cbor_build_bytestring (cbor_data handle, size_t length)`

Creates a new byte string and initializes it.

The handle will be copied to a newly allocated block

Return A new byte string with content handle. NULL on malloc failure.

Parameters

- `handle`: Block of binary data
- `length`: Length of data

Manipulating existing items

`void cbor_bytestring_set_handle(cbor_item_t * item, cbor_mutable_data CBOR_RESTRICT_POINTER data)`
Set the handle to the binary data.

Parameters

- `item[borrow]`: A definite byte string
- `data`: The memory block. The caller gives up the ownership of the block. libcbor will deallocate it when appropriate using its free function
- `length`: Length of the data block

`bool cbor_bytestring_add_chunk(cbor_item_t *item, cbor_item_t *chunk)`
Appends a chunk to the bytestring.

Indefinite byte strings only.

May realloc the chunk storage.

Return true on success, false on realloc failure. In that case, the refcount of `chunk` is not increased and the `item` is left intact.

Parameters

- `item[borrow]`: An indefinite byte string
- `item[incref]`: A definite byte string

2.3.7 Type 3 – UTF-8 strings

CBOR strings work in much the same ways as *Type 2 – Byte strings*.

Corresponding <i>cbor_type</i>	CBOR_TYPE_STRING
Number of allocations (definite)	One plus any manipulations with the data
Number of allocations (indefinite)	One plus logarithmically many reallocations relative to chunk count
Storage requirements (definite)	<code>sizeof(cbor_item_t) + length(handle)</code>
Storage requirements (indefinite)	<code>sizeof(cbor_item_t) * (1 + chunk_count) + chunks</code>

Streaming indefinite strings

Please refer to *Streaming & indefinite items*.

UTF-8 encoding validation

libcbor considers UTF-8 encoding validity to be a part of the well-formedness notion of CBOR and therefore invalid UTF-8 strings will be rejected by the parser. Strings created by the user are not checked.

Getting metadata

`size_t cbor_string_length(const cbor_item_t *item)`
Returns the length of the underlying string.

For definite strings only

Return length of the string. Zero if no chunk has been attached yet

Parameters

- `item[borrow]`: a definite string

bool **`cbor_string_is_definite`** (`const cbor_item_t *item`)

Is the string definite?

Return Is the string definite?

Parameters

- `item[borrow]`: a string

bool **`cbor_string_is_indefinite`** (`const cbor_item_t *item`)

Is the string indefinite?

Return Is the string indefinite?

Parameters

- `item[borrow]`: a string

size_t **`cbor_string_chunk_count`** (`const cbor_item_t *item`)

Get the number of chunks this string consist of.

Return The chunk count. 0 for freshly created items.

Parameters

- `item[borrow]`: A indefinite string

Reading data

cbor_mutable_data **`cbor_string_handle`** (`const cbor_item_t *item`)

Get the handle to the underlying string.

Definite items only. Modifying the data is allowed. In that case, the caller takes responsibility for the effect on items this item might be a part of

Return The address of the underlying string. NULL if no data have been assigned yet.

Parameters

- `item[borrow]`: A definite string

`cbor_item_t **` **`cbor_string_chunks_handle`** (`const cbor_item_t *item`)

Get the handle to the array of chunks.

Manipulations with the memory block (e.g. sorting it) are allowed, but the validity and the number of chunks must be retained.

Return array of `cbor_string_chunk_count` definite strings

Parameters

- `item[borrow]`: A indefinite string

Creating new items

`cbor_item_t *cbor_new_definite_string()`

Creates a new definite string.

The handle is initialized to `NULL` and length to 0

Return new definite string. `NULL` on malloc failure.

`cbor_item_t *cbor_new_indefinite_string()`

Creates a new indefinite string.

The chunks array is initialized to `NULL` and chunkcount to 0

Return new indefinite string. `NULL` on malloc failure.

Building items

`cbor_item_t *cbor_build_string(const char *val)`

Creates a new string and initializes it.

The `val` will be copied to a newly allocated block

Return A new string with content `handle`. `NULL` on malloc failure.

Parameters

- `val`: A null-terminated UTF-8 string

Manipulating existing items

`void cbor_string_set_handle(cbor_item_t * item, cbor_mutable_data CBOR_RESTRICT_POINTER`

Set the handle to the underlying string.

Warning: Using a pointer to a stack allocated constant is a common mistake. Lifetime of the string will expire when it goes out of scope and the CBOR item will be left inconsistent.

Parameters

- `item[borrow]`: A definite string
- `data`: The memory block. The caller gives up the ownership of the block. `libcbor` will deallocate it when appropriate using its free function
- `length`: Length of the data block

`bool cbor_string_add_chunk(cbor_item_t *item, cbor_item_t *chunk)`

Appends a chunk to the string.

Indefinite strings only.

May realloc the chunk storage.

Return `true` on success. `false` on realloc failure. In that case, the refcount of `chunk` is not increased and the `item` is left intact.

Parameters

- `item[borrow]`: An indefinite string
- `item[incref]`: A definite string

2.3.8 Type 4 – Arrays

CBOR arrays, just like *byte strings* and *strings*, can be encoded either as definite, or as indefinite.

Corresponding <i>cbor_type</i>	CBOR_TYPE_ARRAY
Number of allocations (definite)	Two plus any manipulations with the data
Number of allocations (indefinite)	Two plus logarithmically many reallocations relative to additions
Storage requirements (definite)	$(\text{sizeof}(\text{cbor_item_t}) + 1) * \text{size}$
Storage requirements (indefinite)	$\leq \text{sizeof}(\text{cbor_item_t}) + \text{sizeof}(\text{cbor_item_t}) * \text{size} * \text{BUFFER_GROWTH}$

Examples

```
0x9f      Start indefinite array
 0x01      Unsigned integer 1
 0xff      "Break" control token
```

```
0x9f      Start array, 1B length follows
0x20      Unsigned integer 32
...       32 items follow
```

Streaming indefinite arrays

Please refer to *Streaming & indefinite items*.

Getting metadata

`size_t cbor_array_size (const cbor_item_t *item)`

Get the number of members.

Return The number of members

Parameters

- `item[borrow]`: An array

`size_t cbor_array_allocated (const cbor_item_t *item)`

Get the size of the allocated storage.

Return The size of the allocated storage (number of items)

Parameters

- `item[borrow]`: An array

bool **cbor_array_is_definite** (const *cbor_item_t* *item)
Is the array definite?

Return Is the array definite?

Parameters

- *item*[borrow]: An array

bool **cbor_array_is_indefinite** (const *cbor_item_t* *item)
Is the array indefinite?

Return Is the array indefinite?

Parameters

- *item*[borrow]: An array

Reading data

cbor_item_t ****cbor_array_handle** (const *cbor_item_t* *item)
Get the array contents.

The items may be reordered and modified as long as references remain consistent.

Return *cbor_array_size* items

Parameters

- *item*[borrow]: An array

cbor_item_t ***cbor_array_get** (const *cbor_item_t* *item, size_t index)
Get item by index.

Return *incref* The item, or NULL in case of boundary violation

Parameters

- *item*[borrow]: An array
- *index*: The index

Creating new items

cbor_item_t ***cbor_new_definite_array** (size_t size)
Create new definite array.

Return *new* array or NULL upon malloc failure

Parameters

- *size*: Number of slots to preallocate

cbor_item_t ***cbor_new_indefinite_array** ()
Create new indefinite array.

Return *new* array or NULL upon malloc failure

Modifying items

bool **cbor_array_push** (*cbor_item_t* *array, *cbor_item_t* *pushee)
Append to the end.

For indefinite items, storage may be reallocated. For definite items, only the preallocated capacity is available.

Return true on success, false on failure

Parameters

- array[borrow]: An array
- pushee[incref]: The item to push

bool **cbor_array_replace** (*cbor_item_t* *item, size_t index, *cbor_item_t* *value)
Replace item at an index.

The item being replace will be *cbor_decref* 'ed.

Return true on success, false on allocation failure.

Parameters

- item[borrow]: An array
- value[incref]: The item to assign
- index: The index, first item is 0.

bool **cbor_array_set** (*cbor_item_t* *item, size_t index, *cbor_item_t* *value)
Set item by index.

Creating arrays with holes is not possible

Return true on success, false on allocation failure.

Parameters

- item[borrow]: An array
- value[incref]: The item to assign
- index: The index, first item is 0.

2.3.9 Type 5 – Maps

CBOR maps are the plain old associate hash maps known from JSON and many other formats and languages, with one exception: any CBOR data item can be a key, not just strings. This is somewhat unusual and you, as an application developer, should keep that in mind.

Maps can be either definite or indefinite, in much the same way as *Type 4 – Arrays*.

Corresponding <i>cbor_type</i>	CBOR_TYPE_MAP
Number of allocations (definite)	Two plus any manipulations with the data
Number of allocations (indefinite)	Two plus logarithmically many reallocations relative to additions
Storage requirements (definite)	<code>sizeof(cbor_pair) * size + sizeof(cbor_item_t)</code>
Storage requirements (indefinite)	<code><= sizeof(cbor_item_t) + sizeof(cbor_pair) * size * BUFFER_GROWTH</code>

Streaming maps

Please refer to *Streaming & indefinite items*.

Getting metadata

`size_t cbor_map_size (const cbor_item_t *item)`

Get the number of pairs.

Return The number of pairs

Parameters

- `item[borrow]`: A map

`size_t cbor_map_allocated (const cbor_item_t *item)`

Get the size of the allocated storage.

Return Allocated storage size (as the number of `cbor_pair` items)

Parameters

- `item[borrow]`: A map

`bool cbor_map_is_definite (const cbor_item_t *item)`

Is this map definite?

Return Is this map definite?

Parameters

- `item[borrow]`: A map

`bool cbor_map_is_indefinite (const cbor_item_t *item)`

Is this map indefinite?

Return Is this map indefinite?

Parameters

- `item[borrow]`: A map

Reading data

struct cbor_pair ***cbor_map_handle** (const cbor_item_t *item)

Get the pairs storage.

Return Array of *cbor_map_size* pairs. Manipulation is possible as long as references remain valid.

Parameters

- *item*[borrow]: A map

Creating new items

cbor_item_t ***cbor_new_definite_map** (size_t size)

Create a new definite map.

Return new definite map. NULL on malloc failure.

Parameters

- *size*: The number of slots to preallocate

cbor_item_t ***cbor_new_indefinite_map** ()

Create a new indefinite map.

Return new definite map. NULL on malloc failure.

Parameters

- *size*: The number of slots to preallocate

Modifying items

bool **cbor_map_add** (cbor_item_t *item, struct cbor_pair pair)

Add a pair to the map.

For definite maps, items can only be added to the preallocated space. For indefinite maps, the storage will be expanded as needed

Return true on success, false if either reallocation failed or the preallocated storage is full

Parameters

- *item*[borrow]: A map
- *pair*[ineref]: The key-value pair to add (ineref is member-wise)

2.3.10 Type 6 – Semantic tags

Tag are additional metadata that can be used to extend or specialize the meaning or interpretation of the other data items.

For example, one might tag an array of numbers to communicate that it should be interpreted as a vector.

Please consult the official [IANA repository of CBOR tags](#) before inventing new ones.

Corresponding <i>cbor_type</i>	CBOR_TYPE_TAG
Number of allocations	One plus any manipulations with the data reallocations relative to chunk count
Storage requirements	sizeof(cbor_item_t) + the tagged item

*cbor_item_t****cbor_new_tag** (uint64_t *value*)

Create a new tag.

Return new tag. Item reference is NULL.

Parameters

- *value*: The tag value. Please consult the tag repository

*cbor_item_t****cbor_tag_item** (const *cbor_item_t***item*)

Get the tagged item.

Return *incrcf* the tagged item

Parameters

- *item*[*borrow*]: A tag

uint64_t **cbor_tag_value** (const *cbor_item_t***item*)

Get tag value.

Return The tag value. Please consult the tag repository

Parameters

- *item*[*borrow*]: A tag

void **cbor_tag_set_item** (*cbor_item_t***item*, *cbor_item_t***tagged_item*)

Set the tagged item.

Parameters

- *item*[*borrow*]: A tag
- *tagged_item*[*incrcf*]: The item to tag

2.3.11 Type 7 – Floats & control tokens

This type combines two completely unrelated types of items – floating point numbers and special values such as true, false, null, etc. We refer to these special values as ‘control values’ or ‘ctrls’ for short throughout the code.

Just like integers, they have different possible width (resulting in different value ranges and precisions).

enum cbor_float_width

Possible widths of *CBOR_TYPE_FLOAT_CTRL* items.

Values:

CBOR_FLOAT_0

Internal use - ctrl and special values.

CBOR_FLOAT_16

Half float.

CBOR_FLOAT_32

Single float.

CBOR_FLOAT_64

Double.

Corresponding <i>cbor_type</i>	CBOR_TYPE_FLOAT_CTRL
Number of allocations	One per lifetime
Storage requirements	sizeof(cbor_item_t) + 1/4/8

Getting metadata

bool **cbor_float_ctrl_is_ctrl** (const *cbor_item_t* *item)

Is this a ctrl value?

Return Is this a ctrl value?**Parameters**

- *item*[borrow]: A float or ctrl item

cbor_float_width **cbor_float_get_width** (const *cbor_item_t* *item)

Get the float width.

Return The width.**Parameters**

- *item*[borrow]: A float or ctrl item

bool **cbor_ctrl_is_bool** (const *cbor_item_t* *item)

Is this ctrl item a boolean?

Return Is this ctrl item a boolean?**Parameters**

- *item*[borrow]: A ctrl item

Reading data

float **cbor_float_get_float2** (const *cbor_item_t* *item)

Get a half precision float.

The item must have the corresponding width

Parameters

-

float **cbor_float_get_float4** (const *cbor_item_t* *item)

Get a single precision float.

The item must have the corresponding width

Parameters

-

double **cbor_float_get_float8** (const *cbor_item_t* *item)

Get a double precision float.

The item must have the corresponding width

Parameters

-

double **cbor_float_get_float** (const *cbor_item_t* *item)

Get the float value represented as double.

Can be used regardless of the width.

Parameters

-

uint8_t **cbor_ctrl_value** (const *cbor_item_t* *item)

Reads the control value.

Return the simple value

Parameters

- *item*[borrow]: A ctrl item

Creating new items

cbor_item_t ***cbor_new_ctrl** ()

Constructs a new ctrl item.

The width cannot be changed once the item is created

Return new 1B ctrl

cbor_item_t ***cbor_new_float2** ()

Constructs a new float item.

The width cannot be changed once the item is created

Return new 2B float

cbor_item_t ***cbor_new_float4** ()

Constructs a new float item.

The width cannot be changed once the item is created

Return new 4B float

cbor_item_t ***cbor_new_float8** ()

Constructs a new float item.

The width cannot be changed once the item is created

Return new 8B float

*cbor_item_t****cbor_new_null** ()
Constructs new null ctrl item.

Return new null ctrl item

*cbor_item_t****cbor_new_undef** ()
Constructs new undef ctrl item.

Return new undef ctrl item

Building items

*cbor_item_t****cbor_build_bool** (bool *value*)
Constructs new boolean ctrl item.

Return new boolean ctrl item

Parameters

- *value*: The value to use

*cbor_item_t****cbor_build_ctrl** (uint8_t *value*)
Constructs a ctrl item.

Return new ctrl item

Parameters

- *value*: the value to use

*cbor_item_t****cbor_build_float2** (float *value*)
Constructs a new float.

Return new float

Parameters

- *value*: the value to use

*cbor_item_t****cbor_build_float4** (float *value*)
Constructs a new float.

Return new float

Parameters

- *value*: the value to use

*cbor_item_t****cbor_build_float8** (double *value*)
Constructs a new float.

Return new float

Parameters

- *value*: the value to use

Manipulating existing items

void **cbor_set_ctrl** (*cbor_item_t *item*, *uint8_t value*)
Assign a control value.

Warning: It is possible to produce an invalid CBOR value by assigning a invalid value using this mechanism. Please consult the standard before use.

Parameters

- *item[borrow]*: A ctrl item
- *value*: The simple value to assign. Please consult the standard for allowed values

void **cbor_set_float2** (*cbor_item_t *item*, *float value*)
Assigns a float value.

Parameters

- *item[borrow]*: A half precision float
- *value*: The value to assign

void **cbor_set_float4** (*cbor_item_t *item*, *float value*)
Assigns a float value.

Parameters

- *item[borrow]*: A single precision float
- *value*: The value to assign

void **cbor_set_float8** (*cbor_item_t *item*, *double value*)
Assigns a float value.

Parameters

- *item[borrow]*: A double precision float
- *value*: The value to assign

Half floats

CBOR supports two *bytes wide* (“half-precision”) floats which are not supported by the C language. *libcbor* represents them using `float` values throughout the API, which has important implications when manipulating these values.

In particular, if a user uses some of the manipulation APIs (e.g. `cbor_set_float2()`, `cbor_new_float2()`) to introduce a value that doesn’t have an exact half-float representation, the encoding semantics are given by `cbor_encode_half()` as follows:

size_t **cbor_encode_half** (*float*, *unsigned char **, *size_t*)
Encodes a half-precision float.

Since there is no native representation or semantics for half floats in the language, we use single-precision floats, as every value that can be expressed as a half-float can also be expressed as a float.

This however means that not all floats passed to this function can be unambiguously encoded. The behavior is as follows:

- Infinity, NaN are preserved
- Zero is preserved
- Denormalized numbers keep their sign bit and 10 most significant bit of the significand
- All other numbers
 - If the logical value of the exponent is < -24 , the output is zero
 - If the logical value of the exponent is between -23 and -14 , the output is cut off to represent the ‘magnitude’ of the input, by which we mean $(-1)^{\text{signbit}} \times 1.0e^{\text{exponent}}$. The value in the significand is lost.
 - In all other cases, the sign bit, the exponent, and 10 most significant bits of the significand are kept

Return number of bytes written

Parameters

- `value`:
- `buffer`: Target buffer
- `buffer_size`: Available space in the buffer

2.4 Streaming & indefinite items

CBOR *strings*, *byte strings*, *arrays*, and *maps* can be encoded as *indefinite*, meaning their length or size is not specified. Instead, they are divided into *chunks* (*strings*, *byte strings*), or explicitly terminated (*arrays*, *maps*).

This is one of the most important (and due to poor implementations, underutilized) features of CBOR. It enables low-overhead streaming just about anywhere without dealing with channels or pub/sub mechanism. It is, however, important to recognize that CBOR streaming is not a substitute for Websockets¹ and similar technologies.

2.4.1 Decoding

Another way to decode data using libcbor is to specify a callbacks that will be invoked when upon finding certain items in the input. This service is provided by

```
struct cbor_decoder_result cbor_stream_decode(cbor_data buffer, size_t buffer_size, const
                                             struct cbor_callbacks *callbacks, void *con-
                                             text)
```

Stateless decoder.

Will try parsing the `buffer` and will invoke the appropriate callback on success. Decodes one item at a time. No memory allocations occur.

Parameters

- `buffer`: Input buffer
- `buffer_size`: Length of the buffer
- `callbacks`: The callback bundle
- `context`: An arbitrary pointer to allow for maintaining context.

¹ RFC 6455

To get started, you might want to have a look at the simple streaming example:

```
#include "cbor.h"
#include <stdio.h>
#include <string.h>

/*
 * Illustrates how one might skim through a map (which is assumed to have
 * string keys and values only), looking for the value of a specific key
 *
 * Use the examples/data/map.cbor input to test this.
 */

const char * key = "a secret key";
bool key_found = false;

void find_string(void * _ctx, cbor_data buffer, size_t len)
{
    if (key_found) {
        printf("Found the value: %*s\n", (int) len, buffer);
        key_found = false;
    } else if (len == strlen(key)) {
        key_found = (memcmp(key, buffer, len) == 0);
    }
}

int main(int argc, char * argv[])
{
    FILE * f = fopen(argv[1], "rb");
    fseek(f, 0, SEEK_END);
    size_t length = (size_t)ftell(f);
    fseek(f, 0, SEEK_SET);
    unsigned char * buffer = malloc(length);
    fread(buffer, length, 1, f);

    struct cbor_callbacks callbacks = cbor_empty_callbacks;
    struct cbor_decoder_result decode_result;
    size_t bytes_read = 0;
    callbacks.string = find_string;
    while (bytes_read < length) {
        decode_result = cbor_stream_decode(buffer + bytes_read,
                                          length - bytes_read,
                                          &callbacks, NULL);

        bytes_read += decode_result.read;
    }

    free(buffer);
    fclose(f);
}
```

The callbacks are defined by

```
struct cbor_callbacks
```

Callback bundle passed to the decoder.

Public Members

cbor_int8_callback **uint8**
Unsigned int.

cbor_int16_callback **uint16**
Unsigned int.

cbor_int32_callback **uint32**
Unsigned int.

cbor_int64_callback **uint64**
Unsigned int.

cbor_int64_callback **negint64**
Negative int.

cbor_int32_callback **negint32**
Negative int.

cbor_int16_callback **negint16**
Negative int.

cbor_int8_callback **negint8**
Negative int.

cbor_simple_callback **byte_string_start**
Definite byte string.

cbor_string_callback **byte_string**
Indefinite byte string start.

cbor_string_callback **string**
Definite string.

cbor_simple_callback **string_start**
Indefinite string start.

cbor_simple_callback **indef_array_start**
Definite array.

cbor_collection_callback **array_start**
Indefinite array.

cbor_simple_callback **indef_map_start**
Definite map.

cbor_collection_callback **map_start**
Indefinite map.

cbor_int64_callback **tag**
Tags.

cbor_float_callback **float2**
Half float.

cbor_float_callback **float4**
Single float.

cbor_double_callback **float8**
Double float.

cbor_simple_callback **undefined**
Undef.

cbor_simple_callback **null**
Null.

cbor_bool_callback **boolean**
Bool.

cbor_simple_callback **indef_break**
Indefinite item break.

When building custom sets of callbacks, feel free to start from

const struct cbor_callbacks cbor_empty_callbacks
Dummy callback bundle - does nothing.

Related structures

enum cbor_decoder_status
Streaming decoder result - status.

Values:

CBOR_DECODER_FINISHED
OK, finished.

CBOR_DECODER_NEDATA
Not enough data - mismatch with MTB.

CBOR_DECODER_EBUFFER
Buffer manipulation problem.

CBOR_DECODER_ERROR
Malformed or reserved MTB/value.

struct cbor_decoder_result
Streaming decoder result.

Public Members

size_t read
Bytes read.

cbor_decoder_status **status**
The result.

size_t required
When status == CBOR_DECODER_NEDATA, the minimum number of bytes required to continue parsing.

Callback types definition

typedef void (*cbor_int8_callback) (void *, uint8_t)
Callback prototype.

typedef void (*cbor_int16_callback) (void *, uint16_t)
Callback prototype.

typedef void (*cbor_int32_callback) (void *, uint32_t)
Callback prototype.

```
typedef void (*cbor_int64_callback) (void *, uint64_t)
    Callback prototype.
```

```
typedef void (*cbor_simple_callback) (void *)
    Callback prototype.
```

```
typedef void (*cbor_string_callback) (void *, cbor_data, size_t)
    Callback prototype.
```

```
typedef void (*cbor_collection_callback) (void *, size_t)
    Callback prototype.
```

```
typedef void (*cbor_float_callback) (void *, float)
    Callback prototype.
```

```
typedef void (*cbor_double_callback) (void *, double)
    Callback prototype.
```

```
typedef void (*cbor_bool_callback) (void *, bool)
    Callback prototype.
```

2.4.2 Encoding

TODO

2.5 Tests

2.5.1 Unit tests

There is a comprehensive test suite employing [CMocka](#). You can run all of them using `ctest` in the build directory. Individual tests are themselves runnable. Please refer to [CTest](#) documentation for detailed information on how to specify particular subset of tests.

2.5.2 Testing for memory leaks

Every release is tested for memory correctness. You can run these tests by passing the `-T memcheck` flag to `ctest`.¹

2.5.3 Code coverage

Every release is inspected using [GCOV/LCOV](#). Platform-independent code should be fully covered by the test suite. Simply run

```
make coverage
```

or alternatively run `lcov` by hand using

```
lcov --capture --directory . --output-file coverage.info
genhtml coverage.info --output-directory out
```

¹ Project should be configured with `-DCMAKE_BUILD_TYPE=Debug` to obtain meaningful description of location of the leak. You might also need `--dsymutil=yes` on OS X.

2.5.4 Fuzz testing

Every release is tested using a fuzz test. In this test, a huge buffer filled with random data is passed to the decoder. We require that it either succeeds or fail with a sensible error, without leaking any memory. This is intended to simulate real-world situations where data received from the network are CBOR-decoded before any further processing.

2.6 RFC conformance

libcbor is, generally speaking, very faithful implementation of RFC 7049. There are, however, some limitations imposed by technical constraints.

2.6.1 Bytestring length

There is no explicit limitation of indefinite length byte strings.¹ *libcbor* will not handle byte strings with more chunks than the maximum value of `size_t`. On any sane platform, such string would not fit in the memory anyway. It is, however, possible to process arbitrarily long strings and byte strings using the streaming decoder.

2.6.2 “Half-precision” IEEE 754 floats

As of C99 and even C11, there is no standard implementation for 2 bytes floats. *libcbor* packs them as a `float`. When encoding, *libcbor* selects the appropriate wire representation based on metadata and the actual value. This applies both to canonical and normal mode.

For more information on half-float serialization, please refer to the section on *Half floats*.

2.7 Internal mechanics

Internal workings of *libcbor* are mostly derived from the specification. The purpose of this document is to describe technical choices made during design & implementation and to explicate the reasoning behind those choices.

2.7.1 Terminology

MTB	Major Type Byte	http://tools.ietf.org/html/rfc7049#section-2.1
DST	Dynamically Sized Type	Type whose storage requirements cannot be determined during compilation (originated in the Rust community)

2.7.2 Conventions

API symbols start with `cbor_` or `CBOR_` prefix, internal symbols have `_cbor_` or `_CBOR_` prefix.

¹ <http://tools.ietf.org/html/rfc7049#section-2.2.2>

2.7.3 Inspiration & related projects

Most of the API is largely modelled after existing JSON libraries, including

- Jansson
- json-c
- Gnome's JsonGlib

and also borrowing from

- msgpack-c
- Google Protocol Buffers.

2.7.4 General notes on the API design

The API design has two main driving principles:

1. Let the client manage the memory as much as possible
2. Behave exactly as specified by the standard

Combining these two principles in practice turns out to be quite difficult. Indefinite-length strings, arrays, and maps require client to handle every fixed-size chunk explicitly in order to

- ensure the client never runs out of memory due to *libcbor*
- use `realloc()` sparsely and predictably¹
 - provide strong guarantees about its usage (to prevent latency spikes)
 - provide APIs to avoid `realloc()` altogether
- allow proper handling of (streamed) data bigger than available memory

2.7.5 Coding style

This code loosely follows the [Linux kernel coding style](#). Tabs are tabs, and they are 4 characters wide.

2.7.6 Memory layout

CBOR is very dynamic in the sense that it contains many data elements of variable length, sometimes even indefinite length. This section describes internal representation of all CBOR data types.

Generally speaking, data items consist of three parts:

- a generic *handle*,
- the associated metadata,
- and the actual data

type `cbor_item_t`

Represents the item. Used as an opaque type

cbor_type **type**

Type discriminator

¹ Reasonable handling of DSTs requires reallocation if the API is to remain sane.

`size_t refcount`

Reference counter. Used by `cbor_decref()`, `cbor_incref()`

`union cbor_item_metadata metadata`

Union discriminated by `cbor_item_t.type`. Contains type-specific metadata

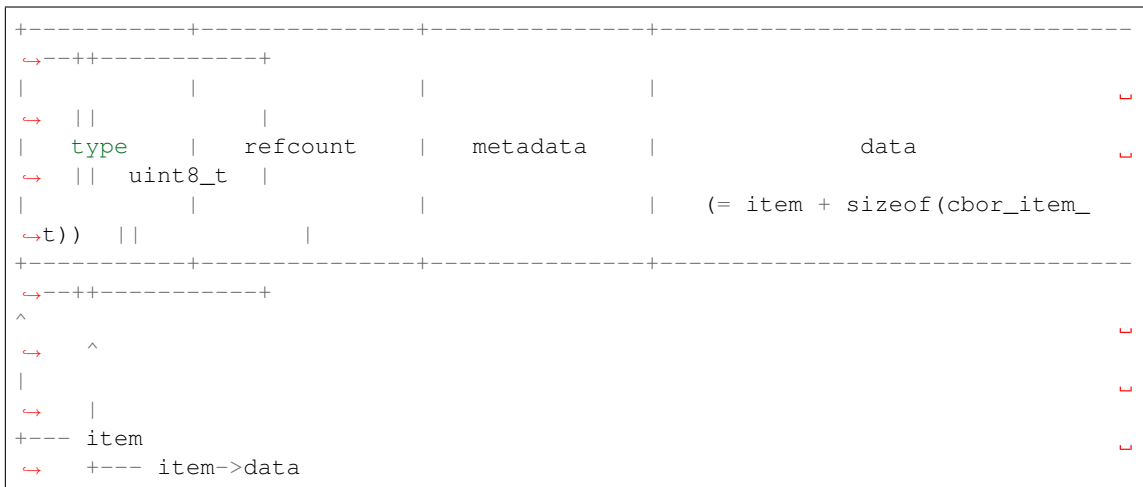
`unsigned char *data`

Contains pointer to the actual data. Small, fixed size items (*Types 0 & 1 – Positive and negative integers, Type 6 – Semantic tags, Type 7 – Floats & control tokens*) are allocated as a single memory block.

Consider the following snippet

```
cbor_item_t * item = cbor_new_int8();
```

then the memory is laid out as follows



Dynamically sized types (*Type 2 – Byte strings, Type 3 – UTF-8 strings, Type 4 – Arrays, Type 5 – Maps*) may store handle and data in separate locations. This enables creating large items (e.g *byte strings*) without `realloc()` or copying large blocks of memory. One simply attaches the correct pointer to the handle.

`union cbor_item_metadata`

```
struct _cbor_int_metadata int_metadata;
```

Used both by both *Types 0 & 1 – Positive and negative integers*

```
struct _cbor_bytestring_metadata bytestring_metadata;
```

```
struct _cbor_string_metadata string_metadata;
```

```
struct _cbor_array_metadata array_metadata;
```

```
struct _cbor_map_metadata map_metadata;
```

```
struct _cbor_tag_metadata tag_metadata;
```

```
struct _cbor_float_ctrl_metadata float_ctrl_metadata;
```

2.7.7 Decoding

As outlined in *API*, there decoding is based on the streaming decoder Essentially, the decoder is a custom set of callbacks for the streaming decoder.

2.8 Changelog

2.8.1 Next

- Correctly set .so version [#52]. Warning: All previous releases will be identified as 0.0 by the linker.
- Fix & prevent heap overflow error in example code [#74] [#76] (by nevun@)

2.8.2 0.5.0 (2017-02-06)

- Remove cmocka from the subtree (always rely on system or user-provided version)
- Windows CI
- Only build tests if explicitly enabled (*-DWITH_TESTS=ON*)
- Fixed static header declarations (by cedric-d)
- Improved documentation (by Michael Richardson)
- Improved *examples/readfile.c*
- Reworked (re)allocation to handle huge inputs and overflows in *size_t* [#16]
- Improvements to C++ linkage (corrected *cbor_empty_callbacks*, fixed *restrict* pointers) (by Dennis Bijwaard)
- Fixed Linux installation directory depending on architecture [#34] (by jvymazal)
- Improved 32-bit support [#35]
- Fixed MSVC compatibility [#31]
- Fixed and improved half-float encoding [#5] [#11]

2.8.3 0.4.0 (2015-12-25)

Breaks build & header compatibility due to:

- Improved build configuration and feature check macros
- Endianness configuration fixes (by Erwin Kroon and David Grigsby)
- pkg-config compatibility (by Vincent Bernat)
- enable use of versioned SONAME (by Vincent Bernat)
- better fuzzer (wasn't random until now, ooops)

2.8.4 0.3.1 (2015-05-21)

- documentation and comments improvements, mostly for the API reference

2.8.5 0.3.0 (2015-05-21)

- Fixes, polishing, niceties across the code base
- Updated examples
- *cbor_copy*

- `cbor_build_negint8`, 16, 32, 64, matching asserts
- `cbor_build_stringn`
- `cbor_build_tag`
- `cbor_build_float2`, ...

2.8.6 0.2.1 (2015-05-17)

- C99 support

2.8.7 0.2.0 (2015-05-17)

- `cbor_ctrl_bool` -> `cbor_ctrl_is_bool`
- Added `cbor_array_allocated` & map equivalent
- Overhauled endianness conversion - ARM now works as expected
- ‘sort.c’ example added
- Significantly improved and doxyfied documentation

2.8.8 0.1.0 (2015-05-06)

The initial release, yay!

2.9 Development

2.9.1 Development dependencies

- CMocka (testing)
- Python and pip (Sphinx platform)
- Doxygen
- Sphinx (documentation)
- There are some Ruby scripts in `misc`
- Valgrind (memory correctness & profiling)
- GCOV/LCOV (test coverage)

Installing *sphinx*

```
pip install sphinx
pip install sphinx_rtd_theme
pip install breathe
pip install https://github.com/lepture/python-livereload/archive/master.zip
pip install sphinx-autobuild
```

Further instructions on configuring advanced features can be found at <http://read-the-docs.readthedocs.org/en/latest/install.html>.

Live preview of docs

```
cd doc
make livehtml
```

Set up git hooks

A git hook that automatically refereshes the [GH pages](#) contents located in `docs` can be symlinked:

```
ln -s misc/hooks/pre-commit .git/hooks
```

Testing and code coverage

Please refer to *Tests*

C

- cbor_array_allocated (C++ function), 35
- cbor_array_get (C++ function), 36
- cbor_array_handle (C++ function), 36
- cbor_array_is_definite (C++ function), 36
- cbor_array_is_indefinite (C++ function), 36
- cbor_array_push (C++ function), 37
- cbor_array_replace (C++ function), 37
- cbor_array_set (C++ function), 37
- cbor_array_size (C++ function), 35
- cbor_bool_callback (C++ type), 49
- cbor_build_bool (C++ function), 43
- cbor_build_bytestring (C++ function), 31
- cbor_build_ctrl (C++ function), 43
- cbor_build_float2 (C++ function), 43
- cbor_build_float4 (C++ function), 43
- cbor_build_float8 (C++ function), 43
- cbor_build_string (C++ function), 34
- cbor_build_uint16 (C++ function), 26
- cbor_build_uint32 (C++ function), 26
- cbor_build_uint64 (C++ function), 27
- cbor_build_uint8 (C++ function), 26
- cbor_bytestring_add_chunk (C++ function), 32
- cbor_bytestring_chunk_count (C++ function), 30
- cbor_bytestring_chunks_handle (C++ function), 31
- cbor_bytestring_handle (C++ function), 31
- cbor_bytestring_is_definite (C++ function), 30
- cbor_bytestring_is_indefinite (C++ function), 30
- cbor_bytestring_length (C++ function), 30
- cbor_callbacks (C++ class), 46
- cbor_callbacks::array_start (C++ member), 47
- cbor_callbacks::boolean (C++ member), 48
- cbor_callbacks::byte_string (C++ member), 47
- cbor_callbacks::byte_string_start (C++ member), 47
- cbor_callbacks::float2 (C++ member), 47
- cbor_callbacks::float4 (C++ member), 47
- cbor_callbacks::float8 (C++ member), 47
- cbor_callbacks::indef_array_start (C++ member), 47
- cbor_callbacks::indef_break (C++ member), 48
- cbor_callbacks::indef_map_start (C++ member), 47
- cbor_callbacks::map_start (C++ member), 47
- cbor_callbacks::negint16 (C++ member), 47
- cbor_callbacks::negint32 (C++ member), 47
- cbor_callbacks::negint64 (C++ member), 47
- cbor_callbacks::negint8 (C++ member), 47
- cbor_callbacks::null (C++ member), 47
- cbor_callbacks::string (C++ member), 47
- cbor_callbacks::string_start (C++ member), 47
- cbor_callbacks::tag (C++ member), 47
- cbor_callbacks::uint16 (C++ member), 47
- cbor_callbacks::uint32 (C++ member), 47
- cbor_callbacks::uint64 (C++ member), 47
- cbor_callbacks::uint8 (C++ member), 47
- cbor_callbacks::undefined (C++ member), 47
- cbor_collection_callback (C++ type), 49
- cbor_copy (C++ function), 21
- cbor_ctrl_is_bool (C++ function), 41
- cbor_ctrl_value (C++ function), 42
- CBOR_DECODER_EBUFFER (C++ enumerator), 48
- CBOR_DECODER_ERROR (C++ enumerator), 48
- CBOR_DECODER_FINISHED (C++ enumerator), 48
- CBOR_DECODER_NEDATA (C++ enumerator), 48
- cbor_decoder_result (C++ class), 48
- cbor_decoder_result::read (C++ member), 48
- cbor_decoder_result::required (C++ member), 48
- cbor_decoder_result::status (C++ member), 48
- cbor_decoder_status (C++ type), 48
- cbor_decref (C++ function), 20
- cbor_double_callback (C++ type), 49
- cbor_empty_callbacks (C++ member), 48
- cbor_encode_half (C++ function), 44
- CBOR_ERR_MALFORMATED (C++ enumerator), 22
- CBOR_ERR_MEMERROR (C++ enumerator), 22
- CBOR_ERR_NODATA (C++ enumerator), 22
- CBOR_ERR_NONE (C++ enumerator), 22
- CBOR_ERR_NOTENOUGHDATA (C++ enumerator), 22
- CBOR_ERR_SYNTAXERROR (C++ enumerator), 22
- cbor_error (C++ class), 22

- cbor_error::code (C++ member), 23
- cbor_error::position (C++ member), 23
- cbor_error_code (C++ type), 22
- CBOR_FLOAT_0 (C++ enumerator), 40
- CBOR_FLOAT_16 (C++ enumerator), 40
- CBOR_FLOAT_32 (C++ enumerator), 40
- CBOR_FLOAT_64 (C++ enumerator), 41
- cbor_float_callback (C++ type), 49
- cbor_float_ctrl_is_ctrl (C++ function), 41
- cbor_float_get_float (C++ function), 42
- cbor_float_get_float2 (C++ function), 41
- cbor_float_get_float4 (C++ function), 41
- cbor_float_get_float8 (C++ function), 42
- cbor_float_get_width (C++ function), 41
- cbor_float_width (C++ type), 40
- cbor_get_uint16 (C++ function), 27
- cbor_get_uint32 (C++ function), 27
- cbor_get_uint64 (C++ function), 27
- cbor_get_uint8 (C++ function), 27
- cbor_incref (C++ function), 20
- cbor_int16_callback (C++ type), 48
- cbor_int32_callback (C++ type), 48
- cbor_int64_callback (C++ type), 48
- cbor_int8_callback (C++ type), 48
- CBOR_INT_16 (C++ enumerator), 25
- CBOR_INT_32 (C++ enumerator), 25
- CBOR_INT_64 (C++ enumerator), 25
- CBOR_INT_8 (C++ enumerator), 25
- cbor_int_get_width (C++ function), 28
- cbor_int_width (C++ type), 25
- cbor_intermediate_decref (C++ function), 20
- cbor_is_bool (C++ function), 18
- cbor_is_float (C++ function), 18
- cbor_is_int (C++ function), 18
- cbor_is_null (C++ function), 18
- cbor_is_undef (C++ function), 18
- cbor_isa_array (C++ function), 17
- cbor_isa_bytestring (C++ function), 17
- cbor_isa_float_ctrl (C++ function), 17
- cbor_isa_map (C++ function), 17
- cbor_isa_negint (C++ function), 17
- cbor_isa_string (C++ function), 17
- cbor_isa_tag (C++ function), 17
- cbor_isa_uint (C++ function), 16
- cbor_item_t (C++ type), 51
- cbor_item_t::data (C++ member), 52
- cbor_item_t::metadata (C++ member), 52
- cbor_item_t::refcount (C++ member), 51
- cbor_item_t::type (C++ member), 51
- cbor_load (C++ function), 22
- cbor_load_result (C++ class), 22
- cbor_load_result::error (C++ member), 22
- cbor_load_result::read (C++ member), 22
- cbor_map_add (C++ function), 39
- cbor_map_allocated (C++ function), 38
- cbor_map_handle (C++ function), 39
- cbor_map_is_definite (C++ function), 38
- cbor_map_is_indefinite (C++ function), 38
- cbor_map_size (C++ function), 38
- cbor_mark_negint (C++ function), 28
- cbor_mark_uint (C++ function), 28
- cbor_move (C++ function), 20
- cbor_new_ctrl (C++ function), 42
- cbor_new_definite_array (C++ function), 36
- cbor_new_definite_bytestring (C++ function), 31
- cbor_new_definite_map (C++ function), 39
- cbor_new_definite_string (C++ function), 34
- cbor_new_float2 (C++ function), 42
- cbor_new_float4 (C++ function), 42
- cbor_new_float8 (C++ function), 42
- cbor_new_indefinite_array (C++ function), 36
- cbor_new_indefinite_bytestring (C++ function), 31
- cbor_new_indefinite_map (C++ function), 39
- cbor_new_indefinite_string (C++ function), 34
- cbor_new_int16 (C++ function), 29
- cbor_new_int32 (C++ function), 29
- cbor_new_int64 (C++ function), 29
- cbor_new_int8 (C++ function), 29
- cbor_new_null (C++ function), 42
- cbor_new_tag (C++ function), 40
- cbor_new_undef (C++ function), 43
- cbor_refcount (C++ function), 20
- cbor_serialize (C++ function), 23
- cbor_serialize_alloc (C++ function), 23
- cbor_serialize_array (C++ function), 24
- cbor_serialize_bytestring (C++ function), 24
- cbor_serialize_float_ctrl (C++ function), 25
- cbor_serialize_map (C++ function), 24
- cbor_serialize_negint (C++ function), 24
- cbor_serialize_string (C++ function), 24
- cbor_serialize_tag (C++ function), 25
- cbor_serialize_uint (C++ function), 23
- cbor_set_allocs (C++ function), 19
- cbor_set_ctrl (C++ function), 44
- cbor_set_float2 (C++ function), 44
- cbor_set_float4 (C++ function), 44
- cbor_set_float8 (C++ function), 44
- cbor_set_uint16 (C++ function), 28
- cbor_set_uint32 (C++ function), 28
- cbor_set_uint64 (C++ function), 28
- cbor_set_uint8 (C++ function), 27
- cbor_simple_callback (C++ type), 49
- cbor_stream_decode (C++ function), 45
- cbor_string_add_chunk (C++ function), 34
- cbor_string_callback (C++ type), 49
- cbor_string_chunk_count (C++ function), 33
- cbor_string_chunks_handle (C++ function), 33
- cbor_string_handle (C++ function), 33

cbor_string_is_definite (C++ function), 33
cbor_string_is_indefinite (C++ function), 33
cbor_string_length (C++ function), 32
cbor_tag_item (C++ function), 40
cbor_tag_set_item (C++ function), 40
cbor_tag_value (C++ function), 40
cbor_type (C++ type), 16
CBOR_TYPE_ARRAY (C++ enumerator), 16
CBOR_TYPE_BYTESTRING (C++ enumerator), 16
CBOR_TYPE_FLOAT_CTRL (C++ enumerator), 16
CBOR_TYPE_MAP (C++ enumerator), 16
CBOR_TYPE_NEGINT (C++ enumerator), 16
CBOR_TYPE_STRING (C++ enumerator), 16
CBOR_TYPE_TAG (C++ enumerator), 16
CBOR_TYPE_UINT (C++ enumerator), 16
cbor_typeof (C++ function), 16

R

RFC

 RFC 6455, 45