# Minicbor

**Raja Mukherji**

**Mar 30, 2023**

# CONTENTS:

CHAPTER

ONE

# OVERVIEW

Minicbor is a small C library for reading and writing CBOR encoded values.

CHAPTER

TWO

# FEATURES

- Suitable for streamed reading: Blocks of CBOR bytes can be parsed incrementally, even when a block splits a CBOR value. Callbacks allow the application to handle detected CBOR values / tokens as they are encountered.
- Suitable for streamed writing: The write functions are designed to work with an underlying application defined stream.
- No memory allocation making it easy to use Minicbor for developing language bindings.

CHAPTER

THREE

# USAGE

The 3 files `minicbor.h`, `minicbor_reader.c` and `minicbor_writer.c` can simply be dropped into an existing project. Alternatively library can be built if required using the supplied `Makefile`.

By default the functions and types are all prefixed with `minicbor_`. This can be changed by defining `MINICBOR_PREFIX` when using the library.

CHAPTER

FOUR

# LICENSE

This library is released under the terms of the MIT license.

## 4.1 Reading CBOR

### 4.1.1 Overview

```c
#include <minicbor.h>

static minicbor_reader_fns Callbacks = {
   .PositiveFn = ...,
   .NegativeFn = ...,
   ...
   .ErrorFn = ...
};

void example_read() {
   minicbor_reader_t Reader;
   Reader.Callbacks = Callbacks;
   Reader.UserData = ...;

   // Initialize Reader
   minicbor_reader_init(&Reader);

   // Parse each block
   unsigned char Bytes[256];
   for (;;) {
      int Count = read(Stream, Bytes, 256);
      if (Count <= 0) break;
      minicbor_read(&Reader, Bytes, Size);
   }
}
```

### 4.1.2 Defines

**CBOR_SIMPLE_FALSE**

Simple false value.

**CBOR_SIMPLE_TRUE**

Simple true value.

**CBOR_SIMPLE_NULL**

Simple null value.

**CBOR_SIMPLE_UNDEF**

Simple undefined value.

### 4.1.3 Types

**struct minicbor_reader_t**

A reader for a CBOR stream. Must be initialized with *minicbor_reader_init()* before each use.

minicbor_reader_fns ***Callbacks**

minicbor_readdata_t **UserData**

**struct minicbor_reader_fns**

void (***PositiveFn**)(void *UserData, uint64_t Number)

Called when a positive integer is encountered.

void (***NegativeFn**)(void *UserData, uint64_t Number)

Called when a negative integer is encountered.

void (***BytesFn**)(void *UserData, int Size)

Called when a bytestring is encountered. `Size` is nonnegative for definite bytestrings and `-1` for indefinite strings. For definite empty bytestrings, `Size` is `0` and *BytesPieceFn()* is not called. Otherwise, *BytesPieceFn()* will be called one or more times, with the last call having `Final` set to `1`.

void (***BytesPieceFn**)(void *UserData, void *Bytes, int Size, int Final)

Called for each piece of a bytestring. Note that pieces here do not correspond to CBOR chunks: there may be more pieces than chunks due to streaming.

void (***StringFn**)(void *UserData, int Size)

Called when a string is encountered. `Size` is nonnegative for definite strings and `-1` for indefinite strings. For definite empty strings, `Size` is `0` and *StringPieceFn()* is not called. Otherwise, *StringPieceFn()* will be called one or more times, with the last call having `Final` set to `1`.

void (***StringPieceFn**)(void *UserData, void *Bytes, int Size, int Final)

Called for each piece of a string. Note that pieces here do not correspond to CBOR chunks: there may be more pieces than chunks due to streaming.

void (***ArrayFn**)(void *UserData, int Size)

Called when an array is encountered. `Size` is nonnegative for definite array and `-1` for indefinite arrays.

void (***MapFn**)(void *UserData, int Size)

Called when an map is encountered. `Size` is nonnegative for definite map and `-1` for indefinite maps.

void (*__TagFn__)(void *UserData, uint64_t Tag)

Called when a tag is encountered.

void (*__SimpleFn__)(void *UserData, int Value)

Called when a simple value is encounted.

void (*__FloatFn__)(void *UserData, double Number)

Called when a floating point number is encountered.

void (*__BreakFn__)(void *UserData)

Called when a break is encountered. This is **not** called for breaks at the end of an indefinite bytestring or string, instead `Final` is set to 1 in the corresponding piece callback.

void (*__ErrorFn__)(void *UserData, int Position, const char *Message)

Called when an invalid CBOR sequence is detected. This puts the reader in an invalid state, any further calls will simply trigger another call *ErrorFn()*;

### 4.1.4 Functions

void **minicbor_reader_init**(minicbor_reader_t *Reader)

Initializes *Reader* for decoding a new CBOR stream. Must be called before any call to *minicbor_read()*. A `minicbor_reader_t` can be reused by calling this function again.

int **minicbor_read**(minicbor_reader_t *Reader, unsigned char *Bytes, unsigned Size)

Parse some CBOR bytes and call the appropriate callbacks. Returns the 1 if *minicbor_reader_finish()* was called within a callback, otherwise returns 0.

void **minicbor_reader_finish**(minicbor_reader_t *Reader)

Set `Reader` state to `MCS_FINISHED`. Must be called from within a reader callback.

int **minicbor_reader_remaining**(minicbor_reader_t *Reader)

Returns the number of bytes remainining to be parsed by the reader.

## 4.2 Writing CBOR

### 4.2.1 Overview

When an underlying stream type object is available, such as a file handle or an in-memory appendable buffer, simply pass a suitable *minicbor_write_fn* to the `minicbor_write_*()` functions.

**Note:** The `minicbor_write_*()` do not write the contents of any bytestring / string values. The contents of these values should be written directly by the user.

```
#include <minicbor.h>

static void stream_write(stream_type *Stream, unsigned char *Bytes, int Size) {
   ...
}
```

(continues on next page)

(continued from previous page)

```
void example_write() {
   stream_type *Stream = ...;
   minicbor_write_indef_array(Stream, stream, write);
   minicbor_write_string(Stream, stream_write, strlen("Hello world!"));
   stream_write(Stream, "Hello world!", strlen("Hello world!"));
   minicbor_write_integer(Stream, stream_write, 100);
   minicbor_write_float4(Stream, stream_write, 1.2);
   minicbor_write_break(Stream, stream_write);
}
```

#### Presizing a CBOR output before writing

If a contiguous output buffer is required, then the required CBOR buffer size can be calculated by calling the `minicbor_write_*()` functions twice.

1. For the first pass, use a *minicbor_write_fn* that takes a pointer to a `size_t` and simply increments the value with the value of `Size`. For example:

   ```
   static void calculate_size(size_t *Required, unsigned char *Bytes, int Size) {
      *Required += Size;
   }
   ```

   The user is responsible for incrementing `Total` with the content sizes of any bytestrings or strings.

2. Then allocate a buffer (e.g. using `malloc()`) and use a *minicbor_write_fn* that actual writes the data to the end of the buffer. For example:

   ```
   static void write_bytes(unsigned char **Tail, unsigned char *Bytes, int Size) {
      memcpy(*Tail, Bytes, Size);
      *Tail += Size;
   }
   ```

### 4.2.2 Defines

**CBOR_SIMPLE_FALSE**

Simple false value.

**CBOR_SIMPLE_TRUE**

Simple true value.

**CBOR_SIMPLE_NULL**

Simple null value.

**CBOR_SIMPLE_UNDEF**

Simple undefined value.

### 4.2.3 Types

typedef void (***minicbor_write_fn**)(void *UserData, const void *Bytes, unsigned Size)

Minicbor write callback type.

**Param UserData**
Pointer passed to `minicbor_write_*()` functions.

**Param Bytes**
Bytes to write.

**Param Size**
Number of bytes.

### 4.2.4 Functions

void **minicbor_write_integer**(void *UserData, *minicbor_write_fn* WriteFn, int64_t Number)

Write a signed integer. Will automatically write a positive or negative integer with the smallest possible width.

void **minicbor_write_positive**(void *UserData, *minicbor_write_fn* WriteFn, uint64_t Number)

Write a positive integer with the smallest width.

void **minicbor_write_negative**(void *UserData, *minicbor_write_fn* WriteFn, uint64_t Number)

Write a negative integer with the smallest width. Here *Number* is the exact value to write into the stream. This means if `X` is the desired negative value to write, then `Number` should be `1 - X` or `~X` (the one's complement). This is to allow the full range of negative numbers to be written.

void **minicbor_write_bytes**(void *UserData, *minicbor_write_fn* WriteFn, unsigned Size)

Write the leading bytes of a definite bytestring with `Size` bytes. The actual bytes should be written directly by the application.

void **minicbor_write_indef_bytes**(void *UserData, *minicbor_write_fn* WriteFn)

Write the leading bytes of an indefinite bytestring. The chunks should be written using *minicbor_write_bytes()* followed by the bytes themselves. Finally, *minicbor_write_break()* should be used to end the indefinite bytestring.

void **minicbor_write_string**(void *UserData, *minicbor_write_fn* WriteFn, unsigned Size)

Write the leading bytes of a definite string with `Size` bytes. The actual string should be written directly by the application.

void **minicbor_write_indef_string**(void *UserData, *minicbor_write_fn* WriteFn)

Write the leading bytes of an indefinite string. The chunks should be written using *minicbor_write_string()* followed by the strings themselves. Finally, *minicbor_write_break()* should be used to end the indefinite string.

void **minicbor_write_array**(void *UserData, *minicbor_write_fn* WriteFn, unsigned Size)

Write the leading bytes of a definite array with `Size` elements. The elements themselves should be written with the appropriate `minicbor_write_*()` functions.

void **minicbor_write_indef_array**(void *UserData, *minicbor_write_fn* WriteFn)

Write the leading bytes of an indefinite array. The elements themselves should be written with the appropriate `minicbor_write_*()` functions. Finally, *minicbor_write_break()* should be used to ende the indefinite array.

void **minicbor_write_map**(void *UserData, *minicbor_write_fn* WriteFn, unsigned Size)

Write the leading bytes of a definite map with `Size` key-value pairs. The keys and values themselves should be written with the appropriate `minicbor_write_*()` functions.

void **minicbor_write_indef_map**(void *UserData, *minicbor_write_fn* WriteFn)

Write the leading bytes of an indefinite map. The keys and values themselves should be written with the appropriate `minicbor_write_*()` functions. Finally, *minicbor_write_break()* should be used to ende the indefinite map.

void **minicbor_write_float2**(void *UserData, *minicbor_write_fn* WriteFn, double Number)

Write a floating point number in half precision.

void **minicbor_write_float4**(void *UserData, *minicbor_write_fn* WriteFn, double Number)

Write a floating point number in single precision.

void **minicbor_write_float8**(void *UserData, *minicbor_write_fn* WriteFn, double Number)

Write a floating point number in double precision.

void **minicbor_write_simple**(void *UserData, *minicbor_write_fn* WriteFn, unsigned char Simple)

Write a simple value.

void **minicbor_write_break**(void *UserData, *minicbor_write_fn* WriteFn)

Write a break (to end an indefinite bytestring, string, array or map).

void **minicbor_write_tag**(void *UserData, *minicbor_write_fn* WriteFn, uint64t Tag)

Write a tag sequence which will apply to the next value written.

CHAPTER

FIVE

# INDICES AND TABLES

- genindex
- modindex
- search

# INDEX