Lua CJSON 2.1.0 Manual

Mark Pulford
1st March 2012

Table of Contents

1. Overview

The Lua CJSON module provides JSON support for Lua.

Features

- Fast, standards compliant encoding/parsing routines
- Full support for JSON with UTF-8, including decoding surrogate pairs
- Optional run-time support for common exceptions to the JSON specification (infinity, NaN,..)
- No dependencies on other libraries

Caveats

- UTF-16 and UTF-32 are not supported

The latest version of this software is available from the Lua CJSON website.

Feel free to email me if you have any patches, suggestions, or comments.

2. Installation

Lua CJSON requires either Lua 5.1, Lua 5.2, or LuaJIT to build.

The build method can be selected from 4 options:

Make
: Unix (including Linux, BSD, Mac OSX & Solaris), Windows

CMake
: Unix, Windows

RPM
: Linux

LuaRocks
: Unix, Windows

2.1. Make

The included `Makefile` has generic settings.

First, review and update the included makefile to suit your platform (if required).

Next, build and install the module:

```
make install
```

Or install manually into your Lua module directory:

```
make
cp cjson.so $LUA_MODULE_DIRECTORY
```

2.2. CMake

CMake can generate build configuration for many different platforms (including Unix and Windows).

First, generate the makefile for your platform using CMake. If CMake is unable to find Lua, manually set the `LUA_DIR` environment variable to the base prefix of your Lua 5.1 installation.

While `cmake` is used in the example below, `ccmake` or `cmake-gui` may be used to present an interface for changing the default build options.

```
mkdir build
```

```
cd build
# Optional: export LUA_DIR=$LUA51_PREFIX
cmake ..
```

Next, build and install the module:

```
make install
# Or:
make
cp cjson.so $LUA_MODULE_DIRECTORY
```

Review the CMake documentation for further details.

2.3. RPM

Linux distributions using RPM can create a package via the included RPM spec file. Ensure the `rpm-build` package (or similar) has been installed.

Build and install the module via RPM:

```
rpmbuild -tb lua-cjson-2.1.0.tar.gz
rpm -Uvh $LUA_CJSON_RPM
```

2.4. LuaRocks

LuaRocks can be used to install and manage Lua modules on a wide range of platforms (including Windows).

First, extract the Lua CJSON source package.

Next, install the module:

```
cd lua-cjson-2.1.0
luarocks make
```

Note LuaRocks does not support platform specific configuration for Solaris. On Solaris, you may need to manually uncomment `USE_INTERNAL_ISINF` in the rockspec before building this module.

Review the LuaRocks documentation for further details.

2.5. Build Options (#define)

Lua CJSON offers several `#define` build options to address portability issues, and enable non-default features. Some build methods may automatically set platform specific options if required. Other features should be enabled manually.

USE_INTERNAL_ISINF
: Workaround for Solaris platforms missing `isinf`.

DISABLE_INVALID_NUMBERS
: Recommended on platforms where `strtod` / `sprintf` are not POSIX compliant (eg, Windows MinGW). Prevents `cjson.encode_invalid_numbers` and `cjson.decode_invalid_numbers` from being enabled. However, `cjson.encode_invalid_numbers` may still be set to `"null"`. When using the Lua CJSON built-in floating point conversion this option is unnecessary and is ignored.

2.5.1. Built-in floating point conversion

Lua CJSON may be built with David Gay's floating point conversion routines. This can increase overall performance by up to 50% on some platforms when converting a large amount of numeric data. However, this option reduces portability and is disabled by default.

USE_INTERNAL_FPCONV
: Enable internal number conversion routines.

IEEE_BIG_ENDIAN
: Must be set on big endian architectures.

MULTIPLE_THREADS
: Must be set if Lua CJSON may be used in a multi-threaded application. Requires the *pthreads* library.

3. API (Functions)

3.1. Synopsis

```
-- Module instantiation
local cjson = require "cjson"
local cjson2 = cjson.new()
local cjson_safe = require "cjson.safe"

-- Translate Lua value to/from JSON
text = cjson.encode(value)
value = cjson.decode(text)

-- Get and/or set Lua CJSON configuration
setting = cjson.decode_invalid_numbers([setting])
setting = cjson.encode_invalid_numbers([setting])
keep = cjson.encode_keep_buffer([keep])
depth = cjson.encode_max_depth([depth])
depth = cjson.decode_max_depth([depth])
convert, ratio, safe = cjson.encode_sparse_array([convert[, ratio[, safe]]])
```

3.2. Module Instantiation

```
local cjson = require "cjson"
local cjson2 = cjson.new()
local cjson_safe = require "cjson.safe"
```

Import Lua CJSON via the Lua `require` function. Lua CJSON does not register a global module table.

The `cjson` module will throw an error during JSON conversion if any invalid data is encountered. Refer to `cjson.encode` and `cjson.decode` for details.

The `cjson.safe` module behaves identically to the `cjson` module, except when errors are encountered during JSON conversion. On error, the `cjson_safe.encode` and `cjson_safe.decode` functions will return `nil` followed by the error message.

`cjson.new` can be used to instantiate an independent copy of the Lua CJSON module. The new module has a separate persistent encoding buffer, and default settings.

Lua CJSON can support Lua implementations using multiple preemptive threads within a single Lua state provided the persistent encoding buffer is not shared. This can be achieved by one of the following methods:

- Disabling the persistent encoding buffer with `cjson.encode_keep_buffer`
- Ensuring each thread calls `cjson.encode` separately (ie, treat `cjson.encode` as non-reentrant).
- Using a separate `cjson` module table per preemptive thread (`cjson.new`)

Note Lua CJSON uses `strtod` and `snprintf` to perform numeric conversion as they are usually well supported, fast and bug free. However, these functions require a workaround for JSON encoding/parsing under locales using a comma decimal separator. Lua CJSON detects the current locale during instantiation to determine and automatically implement the workaround if required. Lua CJSON should be reinitialised via `cjson.new` if the locale of the current process changes. Using a different locale per thread is not supported.

3.3. decode

```
value = cjson.decode(json_text)
```

`cjson.decode` will deserialise any UTF-8 JSON string into a Lua value or table.

UTF-16 and UTF-32 JSON strings are not supported.

`cjson.decode` requires that any NULL (ASCII 0) and double quote (ASCII 34) characters are escaped within strings. All escape codes will be decoded and other bytes will be passed transparently. UTF-8 characters are not validated during decoding and should be checked elsewhere if required.

JSON `null` will be converted to a NULL `lightuserdata` value. This can be compared with `cjson.null` for convenience.

By default, numbers incompatible with the JSON specification (infinity, NaN, hexadecimal) can be decoded. This default can be changed with `cjson.decode_invalid_numbers`.

Example: Decoding

```
json_text = '[ true, { "foo": "bar" } ]'
value = cjson.decode(json_text)
-- Returns: { true, { foo = "bar" } }
```

Caution Care must be taken after decoding JSON objects with numeric keys. Each numeric key will be stored as a Lua `string`. Any subsequent code assuming type `number` may break.

3.4. decode_invalid_numbers

```
setting = cjson.decode_invalid_numbers([setting])
-- "setting" must be a boolean. Default: true.
```

Lua CJSON may generate an error when trying to decode numbers not supported by the JSON specification. *Invalid numbers* are defined as:

- infinity
- not-a-number (NaN)
- hexadecimal

Available settings:

`true`
: Accept and decode *invalid numbers*. This is the default setting.

`false`
: Throw an error when *invalid numbers* are encountered.

The current setting is always returned, and is only updated when an argument is provided.

3.5. decode_max_depth

```
depth = cjson.decode_max_depth([depth])
-- "depth" must be a positive integer. Default: 1000.
```

Lua CJSON will generate an error when parsing deeply nested JSON once the maximum array/object depth has been exceeded. This check prevents unnecessarily complicated JSON from slowing down the application, or crashing the application due to lack of process stack space.

An error may be generated before the depth limit is hit if Lua is unable to allocate more objects on the Lua stack.

By default, Lua CJSON will reject JSON with arrays and/or objects nested more than 1000 levels deep.

The current setting is always returned, and is only updated when an argument is provided.

3.6. encode

```
json_text = cjson.encode(value)
```

`cjson.encode` will serialise a Lua value into a string containing the JSON representation.

`cjson.encode` supports the following types:

- `boolean`
- `lightuserdata` (NULL value only)
- `nil`
- `number`
- `string`
- `table`

The remaining Lua types will generate an error:

- `function`
- `lightuserdata` (non-NULL values)
- `thread`
- `userdata`

By default, numbers are encoded with 14 significant digits. Refer to `cjson.encode_number_precision` for details.

Lua CJSON will escape the following characters within each UTF-8 string:

- Control characters (ASCII 0 - 31)
- Double quote (ASCII 34)
- Forward slash (ASCII 47)
- Blackslash (ASCII 92)
- Delete (ASCII 127)

All other bytes are passed transparently.

Caution

Lua CJSON will successfully encode/decode binary strings, but this is technically not supported by JSON and may not be compatible with other JSON libraries. To ensure the output is valid JSON, applications should ensure all Lua strings passed to `cjson.encode` are UTF-8.

Base64 is commonly used to encode binary data as the most efficient encoding under UTF-8 can only reduce the encoded size by a further ~8%. Lua Base64 routines can be found in the LuaSocket and lbase64 packages.

Lua CJSON uses a heuristic to determine whether to encode a Lua table as a JSON array or an object. A Lua table with only positive integer keys of type `number` will be encoded as a JSON array. All other tables will be encoded as a JSON object.

Lua CJSON does not use metamethods when serialising tables.

- `rawget` is used to iterate over Lua arrays
- `next` is used to iterate over Lua objects

Lua arrays with missing entries (*sparse arrays*) may optionally be encoded in several different ways. Refer to `cjson.encode_sparse_array` for details.

JSON object keys are always strings. Hence `cjson.encode` only supports table keys which are type `number` or `string`. All other types will generate an error.

Note

Standards compliant JSON must be encapsulated in either an object (`{}`) or an array (`[]`). If strictly standards compliant JSON is desired, a table must be passed to `cjson.encode`.

By default, encoding the following Lua values will generate errors:

- Numbers incompatible with the JSON specification (infinity, NaN)
- Tables nested more than 1000 levels deep

- Excessively sparse Lua arrays

These defaults can be changed with:

- cjson.encode_invalid_numbers
- cjson.encode_max_depth
- cjson.encode_sparse_array

Example: Encoding

```
value = { true, { foo = "bar" } }
json_text = cjson.encode(value)
-- Returns: '[true,{"foo":"bar"}]'
```

3.7. encode_invalid_numbers

```
setting = cjson.encode_invalid_numbers([setting])
-- "setting" must a boolean or "null". Default: false.
```

Lua CJSON may generate an error when encoding floating point numbers not supported by the JSON specification (*invalid numbers*):

- infinity
- not-a-number (NaN)

Available settings:

`true`
: Allow *invalid numbers* to be encoded. This will generate non-standard JSON, but this output is supported by some libraries.

`"null"`
: Encode *invalid numbers* as a JSON `null` value. This allows infinity and NaN to be encoded into valid JSON.

`false`
: Throw an error when attempting to encode *invalid numbers*. This is the default setting.

The current setting is always returned, and is only updated when an argument is provided.

3.8. encode_keep_buffer

```
keep = cjson.encode_keep_buffer([keep])
-- "keep" must be a boolean. Default: true.
```

Lua CJSON can reuse the JSON encoding buffer to improve performance.

Available settings:

`true`
: The buffer will grow to the largest size required and is not freed until the Lua CJSON module is garbage collected. This is the default setting.

`false`
: Free the encode buffer after each call to `cjson.encode`.

The current setting is always returned, and is only updated when an argument is provided.

3.9. encode_max_depth

```
depth = cjson.encode_max_depth([depth])
-- "depth" must be a positive integer. Default: 1000.
```

Once the maximum table depth has been exceeded Lua CJSON will generate an error. This prevents a deeply nested or recursive data structure from crashing the application.

By default, Lua CJSON will generate an error when trying to encode data structures with more than 1000 nested tables.

The current setting is always returned, and is only updated when an argument is provided.

Example: Recursive Lua table

```
a = {}; a[1] = a
```

3.10. encode_number_precision

```
precision = cjson.encode_number_precision([precision])
-- "precision" must be an integer between 1 and 14. Default: 14.
```

The amount of significant digits returned by Lua CJSON when encoding numbers can be changed to balance accuracy versus performance. For data structures containing many numbers, setting `cjson.encode_number_precision` to a smaller integer, for example `3`, can improve encoding performance by up to 50%.

By default, Lua CJSON will output 14 significant digits when converting a number to text.

The current setting is always returned, and is only updated when an argument is provided.

3.11. encode_sparse_array

```
convert, ratio, safe = cjson.encode_sparse_array([convert[, ratio[, safe]]])
-- "convert" must be a boolean. Default: false.
-- "ratio" must be a positive integer. Default: 2.
-- "safe" must be a positive integer. Default: 10.
```

Lua CJSON classifies a Lua table into one of three kinds when encoding a JSON array. This is determined by the number of values missing from the Lua array as follows:

Normal
: All values are available.

Sparse
: At least 1 value is missing.

Excessively sparse
: The number of values missing exceeds the configured ratio.

Lua CJSON encodes sparse Lua arrays as JSON arrays using JSON `null` for the missing entries.

An array is excessively sparse when all the following conditions are met:

- `ratio` > `0`
- *maximum_index* > `safe`
- *maximum_index* > *item_count* * `ratio`

Lua CJSON will never consider an array to be *excessively sparse* when `ratio` = `0`. The `safe` limit ensures that small Lua arrays are always encoded as sparse arrays.

By default, attempting to encode an *excessively sparse* array will generate an error. If `convert` is set to `true`, *excessively sparse* arrays will be converted to a JSON object.

The current settings are always returned. A particular setting is only changed when the argument is provided (non-`nil`).

Example: Encoding a sparse array

```
cjson.encode({ [3] = "data" })
-- Returns: '[null,null,"data"]'
```

Example: Enabling conversion to a JSON object

```
cjson.encode_sparse_array(true)
cjson.encode({ [1000] = "excessively sparse" })
-- Returns: '{"1000":"excessively sparse"}'
```

4. API (Variables)

4.1. _NAME

The name of the Lua CJSON module (`"cjson"`).

4.2. _VERSION

The version number of the Lua CJSON module (`"2.1.0"`).

4.3. null

Lua CJSON decodes JSON `null` as a Lua `lightuserdata` NULL pointer. `cjson.null` is provided for comparison.

5. References

- RFC 4627
- JSON website
