Jansson Documentation

Release 2.7

Petri Lehtinen

November 22, 2016

Contents

1 Introduction **3**

2 Contents **5**

2.1 Getting Started 5
2.2 Upgrading from 1.x 8
2.3 Tutorial 9
2.4 RFC Conformance 13
2.5 Portability 14
2.6 API Reference 15
2.7 Changes in Jansson 33

3 Indices and Tables **43**

This is the documentation for Jansson 2.7, last updated November 22, 2016.

CHAPTER 1

Introduction

Jansson is a C library for encoding, decoding and manipulating JSON data. Its main features and design principles are:

- Simple and intuitive API and data model
- Comprehensive documentation
- No dependencies on other libraries
- Full Unicode support (UTF-8)
- Extensive test suite

Jansson is licensed under the MIT license; see LICENSE in the source distribution for details.

Jansson is used in production and its API is stable. It works on numerous platforms, including numerous Unix like systems and Windows. It's suitable for use on any system, including desktop, server, and small embedded systems.

CHAPTER 2

Contents

2.1 Getting Started

2.1.1 Compiling and Installing Jansson

The Jansson source is available at http://www.digip.org/jansson/releases/.

Unix-like systems (including MinGW)

Unpack the source tarball and change to the source directory:

```
bunzip2 -c jansson-2.7.tar.bz2 | tar xf -
cd jansson-2.7
```

The source uses GNU Autotools (autoconf, automake, libtool), so compiling and installing is extremely simple:

```
./configure
make
make check
make install
```

To change the destination directory (`/usr/local` by default), use the `--prefix=DIR` argument to `./configure`. See `./configure --help` for the list of all possible installation options. (There are no options to customize the resulting Jansson binary.)

The command `make check` runs the test suite distributed with Jansson. This step is not strictly necessary, but it may find possible problems that Jansson has on your platform. If any problems are found, please report them.

If you obtained the source from a Git repository (or any other source control system), there's no `./configure` script as it's not kept in version control. To create the script, the build system needs to be bootstrapped. There are many ways to do this, but the easiest one is to use `autoreconf`:

```
autoreconf -vi
```

This command creates the `./configure` script, which can then be used as described above.

CMake (various platforms, including Windows)

Jansson can be built using CMake. Create a build directory for an out-of-tree build, change to that directory, and run `cmake` (or `ccmake`, `cmake-gui`, or similar) to configure the project.

See the examples below for more detailed information.

Note: In the below examples `..` is used as an argument for `cmake`. This is simply the path to the jansson project root directory. In the example it is assumed you've created a sub-directory `build` and are using that. You could use any path you want.

Unix (Make files)

Generating make files on unix:

```
bunzip2 -c jansson-2.7.tar.bz2 | tar xf -
cd jansson-2.7

mkdir build
cd build
cmake .. # or ccmake ..() for a GUI.
```

Then to build:

```
make
make check
make install
```

Windows (Visual Studio)

Creating Visual Studio project files from the command line:

```
<unpack>
cd jansson-2.7

md build
cd build
cmake -G ``Visual Studio 10'' ..
```

You will now have a *Visual Studio Solution* in your build directory. To run the unit tests build the `RUN_TESTS` project.

If you prefer a GUI the `cmake` line in the above example can be replaced with:

```
cmake-gui ..
```

For command line help (including a list of available generators) for CMake simply run:

```
cmake
```

To list available CMake settings (and what they are currently set to) for the project, run:

```
cmake -LH ..
```

Mac OSX (Xcode)

If you prefer using Xcode instead of make files on OSX, do the following. (Use the same steps as for *Unix*):

```
...
cmake -G "Xcode" ..
```

Additional CMake settings

Shared library By default the CMake project will generate build files for building the static library. To build the shared version use:

```
...
cmake -DJANSSON_BUILD_SHARED_LIBS=1 ..
```

Changing install directory (same as autoconf –prefix) Just as with the autoconf project you can change the destination directory for `make install`. The equivalent for autoconfs `./configure --prefix` in CMake is:

```
...
cmake -DCMAKE_INSTALL_PREFIX:PATH=/some/other/path ..
make install
```

Android

Jansson can be built for Android platforms. Android.mk is in the source root directory. The configuration header file is located in the `android` directory in the source distribution.

Other Systems

On non Unix-like systems, you may be unable to run the `./configure` script. In this case, follow these steps. All the files mentioned can be found in the `src/` directory.

1. Create `jansson_config.h` (which has some platform-specific parameters that are normally filled in by the `./configure` script). Edit `jansson_config.h.in`, replacing all `@variable@` placeholders, and rename the file to `jansson_config.h`.
2. Make `jansson.h` and `jansson_config.h` available to the compiler, so that they can be found when compiling programs that use Jansson.
3. Compile all the `.c` files (in the `src/` directory) into a library file. Make the library available to the compiler, as in step 2.

Building the Documentation

(This subsection describes how to build the HTML documentation you are currently reading, so it can be safely skipped.)

Documentation is in the `doc/` subdirectory. It's written in reStructuredText with Sphinx annotations. To generate the HTML documentation, invoke:

```
make html
```

and point your browser to `doc/_build/html/index.html`. Sphinx 1.0 or newer is required to generate the documentation.

2.1.2 Compiling Programs that Use Jansson

Jansson involves one C header file, `jansson.h`, so it's enough to put the line

```
#include <jansson.h>
```

in the beginning of every source file that uses Jansson.

There's also just one library to link with, `libjansson`. Compile and link the program as follows:

```
cc -o prog prog.c -ljansson
```

Starting from version 1.2, there's also support for pkg-config:

```
cc -o prog prog.c `pkg-config --cflags --libs jansson`
```

2.2 Upgrading from 1.x

This chapter lists the backwards incompatible changes introduced in Jansson 2.0, and the steps that are needed for upgrading your code.

The incompatibilities are not dramatic. The biggest change is that all decoding functions now require and extra parameter. Most programs can be modified to work with 2.0 by adding a `0` as the second parameter to all calls of *`json_loads()`*, *`json_loadf()`* and *`json_load_file()`*.

2.2.1 Compatibility

Jansson 2.0 is backwards incompatible with the Jansson 1.x releases. It is ABI incompatible, i.e. all programs dynamically linking to the Jansson library need to be recompiled. It's also API incompatible, i.e. the source code of programs using Jansson 1.x may need modifications to make them compile against Jansson 2.0.

All the 2.x releases are guaranteed to be backwards compatible for both ABI and API, so no recompilation or source changes are needed when upgrading from 2.x to 2.y.

2.2.2 List of Incompatible Changes

Decoding flags For future needs, a `flags` parameter was added as the second parameter to all decoding functions, i.e. *`json_loads()`*, *`json_loadf()`* and *`json_load_file()`*. All calls to these functions need to be changed by adding a `0` as the second argument. For example:

```
/* old code */
json_loads(input, &error);

/* new code */
json_loads(input, 0, &error);
```

Underlying type of JSON integers The underlying C type of JSON integers has been changed from `int` to the widest available signed integer type, i.e. `long long` or `long`, depending on whether `long long` is supported on your system or not. This makes the whole 64-bit integer range available on most modern systems.

`jansson.h` has a typedef *`json_int_t`* to the underlying integer type. `int` should still be used in most cases when dealing with smallish JSON integers, as the compiler handles implicit type coercion. Only when the full 64-bit range is needed, *`json_int_t`* should be explicitly used.

Maximum encoder indentation depth The maximum argument of the `JSON_INDENT()` macro has been changed from 255 to 31, to free up bits from the `flags` parameter of *`json_dumps()`*, *`json_dumpf()`* and *`json_dump_file()`*. If your code uses a bigger indentation than 31, it needs to be changed.

Unsigned integers in API functions Version 2.0 unifies unsigned integer usage in the API. All uses of `unsigned int` and `unsigned long` have been replaced with `size_t`. This includes flags, container sizes, etc. This

should not require source code changes, as both `unsigned int` and `unsigned long` are usually compatible with `size_t`.

2.3 Tutorial

In this tutorial, we create a program that fetches the latest commits of a repository in GitHub over the web. GitHub API uses JSON, so the result can be parsed using Jansson.

To stick to the the scope of this tutorial, we will only cover the the parts of the program related to handling JSON data. For the best user experience, the full source code is available: `github_commits.c`. To compile it (on Unix-like systems with gcc), use the following command:

```
gcc -o github_commits github_commits.c -ljansson -lcurl
```

libcurl is used to communicate over the web, so it is required to compile the program.

The command line syntax is:

```
github_commits USER REPOSITORY
```

`USER` is a GitHub user ID and `REPOSITORY` is the repository name. Please note that the GitHub API is rate limited, so if you run the program too many times within a short period of time, the sever starts to respond with an error.

2.3.1 The GitHub Repo Commits API

The GitHub Repo Commits API is used by sending HTTP requests to URLs like `https://api.github.com/repos/USER/REPOSITORY/commits`, where `USER` and `REPOSITORY` are the GitHub user ID and the name of the repository whose commits are to be listed, respectively.

GitHub responds with a JSON array of the following form:

```
[
    {
        "sha": "<the commit ID>",
        "commit": {
            "message": "<the commit message>",
            <more fields, not important to this tutorial...>
        },
        <more fields...>
    },
    {
        "sha": "<the commit ID>",
        "commit": {
            "message": "<the commit message>",
            <more fields...>
        },
        <more fields...>
    },
    <more commits...>
]
```

In our program, the HTTP request is sent using the following function:

```
static char *request(const char *url);
```

It takes the URL as a parameter, preforms a HTTP GET request, and returns a newly allocated string that contains the response body. If the request fails, an error message is printed to stderr and the return value is *NULL*. For full details, refer to `the code`, as the actual implementation is not important here.

2.3.2 The Program

First the includes:

```
#include <string.h>
#include <jansson.h>
```

Like all the programs using Jansson, we need to include `jansson.h`.

The following definitions are used to build the GitHub API request URL:

```
#define URL_FORMAT   "https://api.github.com/repos/%s/%s/commits"
#define URL_SIZE     256
```

The following function is used when formatting the result to find the first newline in the commit message:

```
/* Return the offset of the first newline in text or the length of
   text if there's no newline */
static int newline_offset(const char *text)
{
    const char *newline = strchr(text, '\n');
    if(!newline)
        return strlen(text);
    else
        return (int)(newline - text);
}
```

The main function follows. In the beginning, we first declare a bunch of variables and check the command line parameters:

```
int main(int argc, char *argv[])
{
    size_t i;
    char *text;
    char url[URL_SIZE];

    json_t *root;
    json_error_t error;

    if(argc != 3)
    {
        fprintf(stderr, "usage: %s USER REPOSITORY\n\n", argv[0]);
        fprintf(stderr, "List commits at USER's REPOSITORY.\n\n");
        return 2;
    }
```

Then we build the request URL using the user and repository names given as command line parameters:

```
snprintf(url, URL_SIZE, URL_FORMAT, argv[1], argv[2]);
```

This uses the `URL_SIZE` and `URL_FORMAT` constants defined above. Now we're ready to actually request the JSON data over the web:

```
text = request(url);
if(!text)
    return 1;
```

If an error occurs, our function `request` prints the error and returns *NULL*, so it's enough to just return 1 from the main function.

Next we'll call *`json_loads()`* to decode the JSON text we got as a response:

```
root = json_loads(text, 0, &error);
free(text);

if(!root)
{
    fprintf(stderr, "error: on line %d: %s\n", error.line, error.text);
    return 1;
}
```

We don't need the JSON text anymore, so we can free the `text` variable right after decoding it. If *`json_loads()`* fails, it returns *NULL* and sets error information to the *`json_error_t`* structure given as the second parameter. In this case, our program prints the error information out and returns 1 from the main function.

Now we're ready to extract the data out of the decoded JSON response. The structure of the response JSON was explained in section *The GitHub Repo Commits API*.

We check that the returned value really is an array:

```
if(!json_is_array(root))
{
    fprintf(stderr, "error: root is not an array\n");
    json_decref(root);
    return 1;
}
```

Then we proceed to loop over all the commits in the array:

```
for(i = 0; i < json_array_size(root); i++)
{
    json_t *data, *sha, *commit, *message;
    const char *message_text;

    data = json_array_get(root, i);
    if(!json_is_object(data))
    {
        fprintf(stderr, "error: commit data %d is not an object\n", i + 1);
        json_decref(root);
        return 1;
    }
...
```

The function *`json_array_size()`* returns the size of a JSON array. First, we again declare some variables and then extract the i'th element of the `root` array using *`json_array_get()`*. We also check that the resulting value is a JSON object.

Next we'll extract the commit ID (a hexadecimal SHA-1 sum), intermediate commit info object, and the commit message from that object. We also do proper type checks:

```
    sha = json_object_get(data, "sha");
    if(!json_is_string(sha))
    {
```

```
        fprintf(stderr, "error: commit %d: sha is not a string\n", i + 1);
        json_decref(root);
        return 1;
    }

    commit = json_object_get(data, "commit");
    if(!json_is_object(commit))
    {
        fprintf(stderr, "error: commit %d: commit is not an object\n", i + 1);
        json_decref(root);
        return 1;
    }

    message = json_object_get(commit, "message");
    if(!json_is_string(message))
    {
        fprintf(stderr, "error: commit %d: message is not a string\n", i + 1);
        json_decref(root);
        return 1;
    }
...
```

And finally, we'll print the first 8 characters of the commit ID and the first line of the commit message. A C-style string is extracted from a JSON string using *json_string_value()*:

```
    message_text = json_string_value(message);
    printf("%.8s %.*s\n",
           json_string_value(id),
           newline_offset(message_text),
           message_text);
}
```

After sending the HTTP request, we decoded the JSON text using *json_loads()*, remember? It returns a *new reference* to the JSON value it decodes. When we're finished with the value, we'll need to decrease the reference count using *json_decref()*. This way Jansson can release the resources:

```
json_decref(root);
return 0;
```

For a detailed explanation of reference counting in Jansson, see *Reference Count* in *API Reference*.

The program's ready, let's test it and view the latest commits in Jansson's repository:

```
$ ./github_commits akheron jansson
1581f26a Merge branch '2.3'
aabfd493 load: Change buffer_pos to be a size_t
bd72efbd load: Avoid unexpected behaviour in macro expansion
e8fd3e30 Document and tweak json_load_callback()
873eddaf Merge pull request #60 from rogerz/contrib
bd2c0c73 Ignore the binary test_load_callback
17a51a4b Merge branch '2.3'
09c39adc Add json_load_callback to the list of exported symbols
cbb80baf Merge pull request #57 from rogerz/contrib
040bd7b0 Add json_load_callback()
2637faa4 Make test stripping locale independent
<...>
```

2.3.3 Conclusion

In this tutorial, we implemented a program that fetches the latest commits of a GitHub repository using the GitHub Repo Commits API. Jansson was used to decode the JSON response and to extract the commit data.

This tutorial only covered a small part of Jansson. For example, we did not create or manipulate JSON values at all. Proceed to *API Reference* to explore all features of Jansson.

2.4 RFC Conformance

JSON is specified in **RFC 4627**, *"The application/json Media Type for JavaScript Object Notation (JSON)"*.

2.4.1 Character Encoding

Jansson only supports UTF-8 encoded JSON texts. It does not support or auto-detect any of the other encodings mentioned in the RFC, namely UTF-16LE, UTF-16BE, UTF-32LE or UTF-32BE. Pure ASCII is supported, as it's a subset of UTF-8.

2.4.2 Strings

JSON strings are mapped to C-style null-terminated character arrays, and UTF-8 encoding is used internally.

All Unicode codepoints U+0000 through U+10FFFF are allowed in string values. However, U+0000 is not allowed in object keys because of API restrictions.

Unicode normalization or any other transformation is never performed on any strings (string values or object keys). When checking for equivalence of strings or object keys, the comparison is performed byte by byte between the original UTF-8 representations of the strings.

2.4.3 Numbers

Real vs. Integer

JSON makes no distinction between real and integer numbers; Jansson does. Real numbers are mapped to the `double` type and integers to the `json_int_t` type, which is a typedef of `long long` or `long`, depending on whether `long long` is supported by your compiler or not.

A JSON number is considered to be a real number if its lexical representation includes one of `e`, `E`, or `.`; regardless if its actual numeric value is a true integer (e.g., all of `1E6`, `3.0`, `400E-2`, and `3.14E3` are mathematical integers, but will be treated as real values). With the `JSON_DECODE_INT_AS_REAL` decoder flag set all numbers are interpreted as real.

All other JSON numbers are considered integers.

When encoding to JSON, real values are always represented with a fractional part; e.g., the `double` value 3.0 will be represented in JSON as `3.0`, not `3`.

Overflow, Underflow & Precision

Real numbers whose absolute values are too small to be represented in a C `double` will be silently estimated with 0.0. Thus, depending on platform, JSON numbers very close to zero such as 1E-999 may result in 0.0.

Real numbers whose absolute values are too large to be represented in a C `double` will result in an overflow error (a JSON decoding error). Thus, depending on platform, JSON numbers like 1E+999 or -1E+999 may result in a parsing error.

Likewise, integer numbers whose absolute values are too large to be represented in the `json_int_t` type (see above) will result in an overflow error (a JSON decoding error). Thus, depending on platform, JSON numbers like 1000000000000000 may result in parsing error.

Parsing JSON real numbers may result in a loss of precision. As long as overflow does not occur (i.e. a total loss of precision), the rounded approximate value is silently used. Thus the JSON number 1.000000000000000005 may, depending on platform, result in the `double` value 1.0.

Signed zeros

JSON makes no statement about what a number means; however Javascript (ECMAscript) does state that +0.0 and -0.0 must be treated as being distinct values, i.e. -0.0 0.0. Jansson relies on the underlying floating point library in the C environment in which it is compiled. Therefore it is platform-dependent whether 0.0 and -0.0 will be distinct values. Most platforms that use the IEEE 754 floating-point standard will support signed zeros.

Note that this only applies to floating-point; neither JSON, C, or IEEE support the concept of signed integer zeros.

Types

No support is provided in Jansson for any C numeric types other than `json_int_t` and `double`. This excludes things such as unsigned types, `long double`, etc. Obviously, shorter types like `short`, `int`, `long` (if `json_int_t` is `long long`) and `float` are implicitly handled via the ordinary C type coercion rules (subject to overflow semantics). Also, no support or hooks are provided for any supplemental "bignum" type add-on packages.

2.5 Portability

2.5.1 Thread safety

Jansson is thread safe and has no mutable global state. The only exceptions are the hash function seed and memory allocation functions, see below.

There's no locking performed inside Jansson's code, so a multithreaded program must perform its own locking if JSON values are shared by multiple threads. Jansson's reference counting semantics may make this a bit harder than it seems, as it's possible to have a reference to a value that's also stored inside a list or object. Modifying the container (adding or removing values) may trigger concurrent access to such values, as containers manage the reference count of their contained values. Bugs involving concurrent incrementing or decrementing of deference counts may be hard to track.

The encoding functions (*`json_dumps()`* and friends) track reference loops by modifying the internal state of objects and arrays. For this reason, encoding functions must not be run on the same JSON values in two separate threads at the same time. As already noted above, be especially careful if two arrays or objects share their contained values with another array or object.

If you want to make sure that two JSON value hierarchies do not contain shared values, use *`json_deep_copy()`* to make copies.

Hash function seed

To prevent an attacker from intentionally causing large JSON objects with specially crafted keys to perform very slow, the hash function used by Jansson is randomized using a seed value. The seed is automatically generated on the first explicit or implicit call to *json_object()*, if *json_object_seed()* has not been called beforehand.

The seed is generated by using operating system's entropy sources if they are available (`/dev/urandom`, `CryptGenRandom()`). The initialization is done in as thread safe manner as possible, by using architecture specific lockless operations if provided by the platform or the compiler.

If you're using threads, it's recommended to autoseed the hashtable explicitly before spawning any threads by calling `json_object_seed(0)` , especially if you're unsure whether the initialization is thread safe on your platform.

Memory allocation functions

Memory allocation functions should be set at most once, and only on program startup. See *Custom Memory Allocation*.

2.5.2 Locale

Jansson works fine under any locale.

However, if the host program is multithreaded and uses `setlocale()` to switch the locale in one thread while Jansson is currently encoding or decoding JSON data in another thread, the result may be wrong or the program may even crash.

Jansson uses locale specific functions for certain string conversions in the encoder and decoder, and then converts the locale specific values to/from the JSON representation. This fails if the locale changes between the string conversion and the locale-to-JSON conversion. This can only happen in multithreaded programs that use `setlocale()`, because `setlocale()` switches the locale for all running threads, not only the thread that calls `setlocale()`.

If your program uses `setlocale()` as described above, consider using the thread-safe `uselocale()` instead.

2.6 API Reference

2.6.1 Preliminaries

All declarations are in `jansson.h`, so it's enough to

```
#include <jansson.h>
```

in each source file.

All constants are prefixed with `JSON_` (except for those describing the library version, prefixed with `JANSSON_`). Other identifiers are prefixed with `json_`. Type names are suffixed with `_t` and `typedef`'d so that the `struct` keyword need not be used.

2.6.2 Library Version

The Jansson version is of the form *A.B.C*, where *A* is the major version, *B* is the minor version and *C* is the micro version. If the micro version is zero, it's omitted from the version string, i.e. the version string is just *A.B*.

When a new release only fixes bugs and doesn't add new features or functionality, the micro version is incremented. When new features are added in a backwards compatible way, the minor version is incremented and the micro version

is set to zero. When there are backwards incompatible changes, the major version is incremented and others are set to zero.

The following preprocessor constants specify the current version of the library:

JANSSON_MAJOR_VERSION, JANSSON_MINOR_VERSION, JANSSON_MICRO_VERSION Integers specifying the major, minor and micro versions, respectively.

JANSSON_VERSION A string representation of the current version, e.g. `"1.2.1"` or `"1.3"`.

JANSSON_VERSION_HEX A 3-byte hexadecimal representation of the version, e.g. `0x010201` for version 1.2.1 and `0x010300` for version 1.3. This is useful in numeric comparisions, e.g.:

```
#if JANSSON_VERSION_HEX >= 0x010300
/* Code specific to version 1.3 and above */
#endif
```

2.6.3 Value Representation

The JSON specification (RFC 4627) defines the following data types: *object*, *array*, *string*, *number*, *boolean*, and *null*. JSON types are used dynamically; arrays and objects can hold any other data type, including themselves. For this reason, Jansson's type system is also dynamic in nature. There's one C type to represent all JSON values, and this structure knows the type of the JSON value it holds.

json_t
This data structure is used throughout the library to represent all JSON values. It always contains the type of the JSON value it holds and the value's reference count. The rest depends on the type of the value.

Objects of *json_t* are always used through a pointer. There are APIs for querying the type, manipulating the reference count, and for constructing and manipulating values of different types.

Unless noted otherwise, all API functions return an error value if an error occurs. Depending on the function's signature, the error value is either *NULL* or -1. Invalid arguments or invalid input are apparent sources for errors. Memory allocation and I/O operations may also cause errors.

Type

The type of a JSON value is queried and tested using the following functions:

enum **json_type**
The type of a JSON value. The following members are defined:

JSON_OBJECT
JSON_ARRAY
JSON_STRING
JSON_INTEGER
JSON_REAL
JSON_TRUE
JSON_FALSE
JSON_NULL

These correspond to JSON object, array, string, number, boolean and null. A number is represented by either a value of the type `JSON_INTEGER` or of the type `JSON_REAL`. A true boolean value is represented by a value of the type `JSON_TRUE` and false by a value of the type `JSON_FALSE`.

int **json_typeof**(const *json_t* **json*)
Return the type of the JSON value (a *json_type* cast to `int`). *json* MUST NOT be *NULL*. This function is actually implemented as a macro for speed.

json_is_object(const *json_t* *json*)
json_is_array(const *json_t* *json*)
json_is_string(const *json_t* *json*)
json_is_integer(const *json_t* *json*)
json_is_real(const *json_t* *json*)
json_is_true(const *json_t* *json*)
json_is_false(const *json_t* *json*)
json_is_null(const *json_t* *json*)
These functions (actually macros) return true (non-zero) for values of the given type, and false (zero) for values of other types and for *NULL*.

json_is_number(const *json_t* *json*)
Returns true for values of types `JSON_INTEGER` and `JSON_REAL`, and false for other types and for *NULL*.

json_is_boolean(const *json_t* *json*)
Returns true for types `JSON_TRUE` and `JSON_FALSE`, and false for values of other types and for *NULL*.

json_boolean_value(const *json_t* *json*)
Alias of *json_is_true()*, i.e. returns 1 for `JSON_TRUE` and 0 otherwise.

New in version 2.7.

Reference Count

The reference count is used to track whether a value is still in use or not. When a value is created, it's reference count is set to 1. If a reference to a value is kept (e.g. a value is stored somewhere for later use), its reference count is incremented, and when the value is no longer needed, the reference count is decremented. When the reference count drops to zero, there are no references left, and the value can be destroyed.

The following functions are used to manipulate the reference count.

json_t ***json_incref**(*json_t* *json*)
Increment the reference count of *json* if it's not *NULL*. Returns *json*.

void **json_decref**(*json_t* *json*)
Decrement the reference count of *json*. As soon as a call to *json_decref()* drops the reference count to zero, the value is destroyed and it can no longer be used.

Functions creating new JSON values set the reference count to 1. These functions are said to return a **new reference**. Other functions returning (existing) JSON values do not normally increase the reference count. These functions are said to return a **borrowed reference**. So, if the user will hold a reference to a value returned as a borrowed reference, he must call *json_incref()*. As soon as the value is no longer needed, *json_decref()* should be called to release the reference.

Normally, all functions accepting a JSON value as an argument will manage the reference, i.e. increase and decrease the reference count as needed. However, some functions **steal** the reference, i.e. they have the same result as if the user called *json_decref()* on the argument right after calling the function. These functions are suffixed with `_new` or have `_new_` somewhere in their name.

For example, the following code creates a new JSON array and appends an integer to it:

```
json_t *array, *integer;

array = json_array();
integer = json_integer(42);

json_array_append(array, integer);
json_decref(integer);
```

Note how the caller has to release the reference to the integer value by calling *json_decref()*. By using a reference stealing function *json_array_append_new()* instead of *json_array_append()*, the code becomes much simpler:

```
json_t *array = json_array();
json_array_append_new(array, json_integer(42));
```

In this case, the user doesn't have to explicitly release the reference to the integer value, as *json_array_append_new()* steals the reference when appending the value to the array.

In the following sections it is clearly documented whether a function will return a new or borrowed reference or steal a reference to its argument.

Circular References

A circular reference is created when an object or an array is, directly or indirectly, inserted inside itself. The direct case is simple:

```
json_t *obj = json_object();
json_object_set(obj, "foo", obj);
```

Jansson will refuse to do this, and *json_object_set()* (and all the other such functions for objects and arrays) will return with an error status. The indirect case is the dangerous one:

```
json_t *arr1 = json_array(), *arr2 = json_array();
json_array_append(arr1, arr2);
json_array_append(arr2, arr1);
```

In this example, the array `arr2` is contained in the array `arr1`, and vice versa. Jansson cannot check for this kind of indirect circular references without a performance hit, so it's up to the user to avoid them.

If a circular reference is created, the memory consumed by the values cannot be freed by *json_decref()*. The reference counts never drops to zero because the values are keeping the references to each other. Moreover, trying to encode the values with any of the encoding functions will fail. The encoder detects circular references and returns an error status.

2.6.4 True, False and Null

These three values are implemented as singletons, so the returned pointers won't change between invocations of these functions.

json_t ***json_true**(void)
Return value: New reference. Returns the JSON true value.

json_t ***json_false**(void)
Return value: New reference. Returns the JSON false value.

json_t ***json_boolean**(val)
Return value: New reference. Returns JSON false if `val` is zero, and JSON true otherwise. This is a macro, and equivalent to `val ? json_true() : json_false()`.

New in version 2.4.

json_t ***json_null**(void)
Return value: New reference. Returns the JSON null value.

2.6.5 String

Jansson uses UTF-8 as the character encoding. All JSON strings must be valid UTF-8 (or ASCII, as it's a subset of UTF-8). All Unicode codepoints U+0000 through U+10FFFF are allowed, but you must use length-aware functions if you wish to embed NUL bytes in strings.

json_t ***json_string**(const char **value*)
Return value: New reference. Returns a new JSON string, or *NULL* on error. *value* must be a valid null terminated UTF-8 encoded Unicode string.

json_t ***json_stringn**(const char **value*, size_t *len*)
Return value: New reference. Like `json_string()`, but with explicit length, so *value* may contain null characters or not be null terminated.

json_t ***json_string_nocheck**(const char **value*)
Return value: New reference. Like `json_string()`, but doesn't check that *value* is valid UTF-8. Use this function only if you are certain that this really is the case (e.g. you have already checked it by other means).

json_t ***json_stringn_nocheck**(const char **value*, size_t *len*)
Return value: New reference. Like `json_string_nocheck()`, but with explicit length, so *value* may contain null characters or not be null terminated.

const char ***json_string_value**(const *json_t* **string*)
Returns the associated value of *string* as a null terminated UTF-8 encoded string, or *NULL* if *string* is not a JSON string.

The retuned value is read-only and must not be modified or freed by the user. It is valid as long as *string* exists, i.e. as long as its reference count has not dropped to zero.

size_t **json_string_length**(const *json_t* **string*)
Returns the length of *string* in its UTF-8 presentation, or zero if *string* is not a JSON string.

int **json_string_set**(const *json_t* **string*, const char **value*)
Sets the associated value of *string* to *value*. *value* must be a valid UTF-8 encoded Unicode string. Returns 0 on success and -1 on error.

int **json_string_setn**(*json_t* **string*, const char **value*, size_t *len*)
Like `json_string_set()`, but with explicit length, so *value* may contain null characters or not be null terminated.

int **json_string_set_nocheck**(const *json_t* **string*, const char **value*)
Like `json_string_set()`, but doesn't check that *value* is valid UTF-8. Use this function only if you are certain that this really is the case (e.g. you have already checked it by other means).

int **json_string_setn_nocheck**(*json_t* **string*, const char **value*, size_t *len*)
Like `json_string_set_nocheck()`, but with explicit length, so *value* may contain null characters or not be null terminated.

2.6.6 Number

The JSON specification only contains one numeric type, "number". The C programming language has distinct types for integer and floating-point numbers, so for practical reasons Jansson also has distinct types for the two. They are called "integer" and "real", respectively. For more information, see *RFC Conformance*.

json_int_t
This is the C type that is used to store JSON integer values. It represents the widest integer type available on your system. In practice it's just a typedef of `long long` if your compiler supports it, otherwise `long`.

Usually, you can safely use plain `int` in place of `json_int_t`, and the implicit C integer conversion handles the rest. Only when you know that you need the full 64-bit range, you should use `json_int_t` explicitly.

JSON_INTEGER_IS_LONG_LONG This is a preprocessor variable that holds the value 1 if *json_int_t* is `long long`, and 0 if it's `long`. It can be used as follows:

```
#if JSON_INTEGER_IS_LONG_LONG
/* Code specific for long long */
#else
/* Code specific for long */
#endif
```

JSON_INTEGER_FORMAT This is a macro that expands to a `printf()` conversion specifier that corresponds to *json_int_t*, without the leading `%` sign, i.e. either `"lld"` or `"ld"`. This macro is required because the actual type of *json_int_t* can be either `long` or `long long`, and `printf()` reuiqres different length modifiers for the two.

Example:

```
json_int_t x = 123123123;
printf("x is %" JSON_INTEGER_FORMAT "\n", x);
```

json_t ***json_integer**(*json_int_t value*)
Return value: New reference. Returns a new JSON integer, or *NULL* on error.

json_int_t **json_integer_value**(const *json_t *integer*)
Returns the associated value of *integer*, or 0 if *json* is not a JSON integer.

int **json_integer_set**(const *json_t *integer*, *json_int_t value*)
Sets the associated value of *integer* to *value*. Returns 0 on success and -1 if *integer* is not a JSON integer.

json_t ***json_real**(double *value*)
Return value: New reference. Returns a new JSON real, or *NULL* on error.

double **json_real_value**(const *json_t *real*)
Returns the associated value of *real*, or 0.0 if *real* is not a JSON real.

int **json_real_set**(const *json_t *real*, double *value*)
Sets the associated value of *real* to *value*. Returns 0 on success and -1 if *real* is not a JSON real.

In addition to the functions above, there's a common query function for integers and reals:

double **json_number_value**(const *json_t *json*)
Returns the associated value of the JSON integer or JSON real *json*, cast to double regardless of the actual type. If *json* is neither JSON real nor JSON integer, 0.0 is returned.

2.6.7 Array

A JSON array is an ordered collection of other JSON values.

json_t ***json_array**(void)
Return value: New reference. Returns a new JSON array, or *NULL* on error. Initially, the array is empty.

size_t **json_array_size**(const *json_t *array*)
Returns the number of elements in *array*, or 0 if *array* is NULL or not a JSON array.

json_t ***json_array_get**(const *json_t *array*, size_t *index*)
Return value: Borrowed reference. Returns the element in *array* at position *index*. The valid range for *index* is from 0 to the return value of *json_array_size()* minus 1. If *array* is not a JSON array, if *array* is *NULL*, or if *index* is out of range, *NULL* is returned.

int **json_array_set**(*json_t *array*, size_t *index*, *json_t *value*)
Replaces the element in *array* at position *index* with *value*. The valid range for *index* is from 0 to the return value of *json_array_size()* minus 1. Returns 0 on success and -1 on error.

int **json_array_set_new**(*json_t *array*, size_t *index*, *json_t *value*)
Like *json_array_set()* but steals the reference to *value*. This is useful when *value* is newly created and not used after the call.

int **json_array_append**(*json_t *array*, *json_t *value*)
Appends *value* to the end of *array*, growing the size of *array* by 1. Returns 0 on success and -1 on error.

int **json_array_append_new**(*json_t *array*, *json_t *value*)
Like *json_array_append()* but steals the reference to *value*. This is useful when *value* is newly created and not used after the call.

int **json_array_insert**(*json_t *array*, size_t *index*, *json_t *value*)
Inserts *value* to *array* at position *index*, shifting the elements at *index* and after it one position towards the end of the array. Returns 0 on success and -1 on error.

int **json_array_insert_new**(*json_t *array*, size_t *index*, *json_t *value*)
Like *json_array_insert()* but steals the reference to *value*. This is useful when *value* is newly created and not used after the call.

int **json_array_remove**(*json_t *array*, size_t *index*)
Removes the element in *array* at position *index*, shifting the elements after *index* one position towards the start of the array. Returns 0 on success and -1 on error. The reference count of the removed value is decremented.

int **json_array_clear**(*json_t *array*)
Removes all elements from *array*. Returns 0 on sucess and -1 on error. The reference count of all removed values are decremented.

int **json_array_extend**(*json_t *array*, *json_t *other_array*)
Appends all elements in *other_array* to the end of *array*. Returns 0 on success and -1 on error.

The following macro can be used to iterate through all elements in an array.

json_array_foreach(array, index, value)
Iterate over every element of `array`, running the block of code that follows each time with the proper values set to variables `index` and `value`, of types `size_t` and *json_t ** respectively. Example:

```
/* array is a JSON array */
size_t index;
json_t *value;

json_array_foreach(array, index, value) {
    /* block of code that uses index and value */
}
```

The items are returned in increasing index order.

This macro expands to an ordinary `for` statement upon preprocessing, so its performance is equivalent to that of hand-written code using the array access functions. The main advantage of this macro is that it abstracts away the complexity, and makes for shorter, more concise code.

New in version 2.5.

2.6.8 Object

A JSON object is a dictionary of key-value pairs, where the key is a Unicode string and the value is any JSON value.

Even though NUL bytes are allowed in string values, they are not allowed in object keys.

json_t ***json_object**(void)
Return value: New reference. Returns a new JSON object, or *NULL* on error. Initially, the object is empty.

size_t **json_object_size**(const *json_t* *object*)
Returns the number of elements in *object*, or 0 if *object* is not a JSON object.

json_t ***json_object_get**(const *json_t* *object*, const char *key*)
Return value: Borrowed reference. Get a value corresponding to *key* from *object*. Returns *NULL* if *key* is not found and on error.

int **json_object_set**(*json_t* *object*, const char *key*, *json_t* *value*)
Set the value of *key* to *value* in *object*. *key* must be a valid null terminated UTF-8 encoded Unicode string. If there already is a value for *key*, it is replaced by the new value. Returns 0 on success and -1 on error.

int **json_object_set_nocheck**(*json_t* *object*, const char *key*, *json_t* *value*)
Like `json_object_set()`, but doesn't check that *key* is valid UTF-8. Use this function only if you are certain that this really is the case (e.g. you have already checked it by other means).

int **json_object_set_new**(*json_t* *object*, const char *key*, *json_t* *value*)
Like `json_object_set()` but steals the reference to *value*. This is useful when *value* is newly created and not used after the call.

int **json_object_set_new_nocheck**(*json_t* *object*, const char *key*, *json_t* *value*)
Like `json_object_set_new()`, but doesn't check that *key* is valid UTF-8. Use this function only if you are certain that this really is the case (e.g. you have already checked it by other means).

int **json_object_del**(*json_t* *object*, const char *key*)
Delete *key* from *object* if it exists. Returns 0 on success, or -1 if *key* was not found. The reference count of the removed value is decremented.

int **json_object_clear**(*json_t* *object*)
Remove all elements from *object*. Returns 0 on success and -1 if *object* is not a JSON object. The reference count of all removed values are decremented.

int **json_object_update**(*json_t* *object*, *json_t* *other*)
Update *object* with the key-value pairs from *other*, overwriting existing keys. Returns 0 on success or -1 on error.

int **json_object_update_existing**(*json_t* *object*, *json_t* *other*)
Like `json_object_update()`, but only the values of existing keys are updated. No new keys are created. Returns 0 on success or -1 on error.

New in version 2.3.

int **json_object_update_missing**(*json_t* *object*, *json_t* *other*)
Like `json_object_update()`, but only new keys are created. The value of any existing key is not changed. Returns 0 on success or -1 on error.

New in version 2.3.

The following macro can be used to iterate through all key-value pairs in an object.

json_object_foreach(object, key, value)
Iterate over every key-value pair of `object`, running the block of code that follows each time with the proper values set to variables `key` and `value`, of types `const char *` and `json_t *` respectively. Example:

```
/* obj is a JSON object */
const char *key;
json_t *value;

json_object_foreach(obj, key, value) {
    /* block of code that uses key and value */
}
```

The items are not returned in any particular order.

This macro expands to an ordinary `for` statement upon preprocessing, so its performance is equivalent to that of hand-written iteration code using the object iteration protocol (see below). The main advantage of this macro is that it abstracts away the complexity behind iteration, and makes for shorter, more concise code.

New in version 2.3.

The following functions implement an iteration protocol for objects, allowing to iterate through all key-value pairs in an object. The items are not returned in any particular order, as this would require sorting due to the internal hashtable implementation.

void *__json_object_iter__(*json_t* **object*)
Returns an opaque iterator which can be used to iterate over all key-value pairs in *object*, or *NULL* if *object* is empty.

void *__json_object_iter_at__(*json_t* **object*, const char **key*)
Like *json_object_iter()*, but returns an iterator to the key-value pair in *object* whose key is equal to *key*, or NULL if *key* is not found in *object*. Iterating forward to the end of *object* only yields all key-value pairs of the object if *key* happens to be the first key in the underlying hash table.

void *__json_object_iter_next__(*json_t* **object*, void **iter*)
Returns an iterator pointing to the next key-value pair in *object* after *iter*, or *NULL* if the whole object has been iterated through.

const char *__json_object_iter_key__(void **iter*)
Extract the associated key from *iter*.

json_t *__json_object_iter_value__(void **iter*)
Return value: Borrowed reference. Extract the associated value from *iter*.

int __json_object_iter_set__(*json_t* **object*, void **iter*, *json_t* **value*)
Set the value of the key-value pair in *object*, that is pointed to by *iter*, to *value*.

int __json_object_iter_set_new__(*json_t* **object*, void **iter*, *json_t* **value*)
Like *json_object_iter_set()*, but steals the reference to *value*. This is useful when *value* is newly created and not used after the call.

void *__json_object_key_to_iter__(const char **key*)
Like *json_object_iter_at()*, but much faster. Only works for values returned by *json_object_iter_key()*. Using other keys will lead to segfaults. This function is used internally to implement *json_object_foreach()*.

New in version 2.3.

The iteration protocol can be used for example as follows:

```
/* obj is a JSON object */
const char *key;
json_t *value;

void *iter = json_object_iter(obj);
while(iter)
{
    key = json_object_iter_key(iter);
    value = json_object_iter_value(iter);
    /* use key and value ... */
    iter = json_object_iter_next(obj, iter);
}
```

void __json_object_seed__(size_t *seed*)
Seed the hash function used in Jansson's hashtable implementation. The seed is used to randomize the hash function so that an attacker cannot control its output.

If *seed* is 0, Jansson generates the seed itselfy by reading random data from the operating system's entropy sources. If no entropy sources are available, falls back to using a combination of the current timestamp (with microsecond precision if possible) and the process ID.

If called at all, this function must be called before any calls to *json_object()*, either explicit or implicit. If this function is not called by the user, the first call to *json_object()* (either explicit or implicit) seeds the hash function. See *Thread safety* for notes on thread safety.

If repeatable results are required, for e.g. unit tests, the hash function can be "unrandomized" by calling *json_object_seed()* with a constant value on program startup, e.g. `json_object_seed(1)`.

New in version 2.6.

2.6.9 Error reporting

Jansson uses a single struct type to pass error information to the user. See sections *Decoding*, *Building Values* and *Parsing and Validating Values* for functions that pass error information using this struct.

json_error_t

char text[]
The error message (in UTF-8), or an empty string if a message is not available.

char source[]
Source of the error. This can be (a part of) the file name or a special identifier in angle brackers (e.g. `<string>`).

int **line**
The line number on which the error occurred.

int **column**
The column on which the error occurred. Note that this is the *character column*, not the byte column, i.e. a multibyte UTF-8 character counts as one column.

size_t **position**
The position in bytes from the start of the input. This is useful for debugging Unicode encoding problems.

The normal use of *json_error_t* is to allocate it on the stack, and pass a pointer to a function. Example:

```
int main() {
    json_t *json;
    json_error_t error;

    json = json_load_file("/path/to/file.json", 0, &error);
    if(!json) {
        /* the error variable contains error information */
    }
    ...
}
```

Also note that if the call succeeded (`json != NULL` in the above example), the contents of `error` are generally left unspecified. The decoding functions write to the `position` member also on success. See *Decoding* for more info.

All functions also accept *NULL* as the *json_error_t* pointer, in which case no error information is returned to the caller.

2.6.10 Encoding

This sections describes the functions that can be used to encode values to JSON. By default, only objects and arrays can be encoded directly, since they are the only valid *root* values of a JSON text. To encode any JSON value, use the `JSON_ENCODE_ANY` flag (see below).

By default, the output has no newlines, and spaces are used between array and object elements for a readable output. This behavior can be altered by using the `JSON_INDENT` and `JSON_COMPACT` flags described below. A newline is never appended to the end of the encoded JSON data.

Each function takes a *flags* parameter that controls some aspects of how the data is encoded. Its default value is 0. The following macros can be ORed together to obtain *flags*.

`JSON_INDENT(n)` Pretty-print the result, using newlines between array and object items, and indenting with *n* spaces. The valid range for *n* is between 0 and 31 (inclusive), other values result in an undefined output. If `JSON_INDENT` is not used or *n* is 0, no newlines are inserted between array and object items.

The `JSON_MAX_INDENT` constant defines the maximum indentation that can be used, and its value is 31.

Changed in version 2.7: Added `JSON_MAX_INDENT`.

`JSON_COMPACT` This flag enables a compact representation, i.e. sets the separator between array and object items to `","` and between object keys and values to `":"`. Without this flag, the corresponding separators are `", "` and `": "` for more readable output.

`JSON_ENSURE_ASCII` If this flag is used, the output is guaranteed to consist only of ASCII characters. This is achived by escaping all Unicode characters outside the ASCII range.

`JSON_SORT_KEYS` If this flag is used, all the objects in output are sorted by key. This is useful e.g. if two JSON texts are diffed or visually compared.

`JSON_PRESERVE_ORDER` If this flag is used, object keys in the output are sorted into the same order in which they were first inserted to the object. For example, decoding a JSON text and then encoding with this flag preserves the order of object keys.

`JSON_ENCODE_ANY` Specifying this flag makes it possible to encode any JSON value on its own. Without it, only objects and arrays can be passed as the *root* value to the encoding functions.

Note: Encoding any value may be useful in some scenarios, but it's generally discouraged as it violates strict compatiblity with **RFC 4627**. If you use this flag, don't expect interoperatibility with other JSON systems.

New in version 2.1.

`JSON_ESCAPE_SLASH` Escape the `/` characters in strings with `\/`.

New in version 2.4.

`JSON_REAL_PRECISION(n)` Output all real numbers with at most *n* digits of precision. The valid range for *n* is between 0 and 31 (inclusive), and other values result in an undefined behavior.

By default, the precision is 17, to correctly and losslessly encode all IEEE 754 double precision floating point numbers.

New in version 2.7.

The following functions perform the actual JSON encoding. The result is in UTF-8.

char ***`json_dumps`**(const *json_t* **root*, size_t *flags*)
Returns the JSON representation of *root* as a string, or *NULL* on error. *flags* is described above. The return value must be freed by the caller using `free()`.

int **`json_dumpf`**(const *json_t* **root*, FILE **output*, size_t *flags*)
Write the JSON representation of *root* to the stream *output*. *flags* is described above. Returns 0 on success and

-1 on error. If an error occurs, something may have already been written to *output*. In this case, the output is undefined and most likely not valid JSON.

int **json_dump_file**(const *json_t* *json, const char *path, size_t *flags*)
Write the JSON representation of *root* to the file *path*. If *path* already exists, it is overwritten. *flags* is described above. Returns 0 on success and -1 on error.

json_dump_callback_t
A typedef for a function that's called by *json_dump_callback()*:

```
typedef int (*json_dump_callback_t)(const char *buffer, size_t size, void *data);
```

buffer points to a buffer containing a chunk of output, *size* is the length of the buffer, and *data* is the corresponding *json_dump_callback()* argument passed through.

On error, the function should return -1 to stop the encoding process. On success, it should return 0.

New in version 2.2.

int **json_dump_callback**(const *json_t* *json, *json_dump_callback_t* *callback*, void *data, size_t *flags*)
Call *callback* repeatedly, passing a chunk of the JSON representation of *root* each time. *flags* is described above. Returns 0 on success and -1 on error.

New in version 2.2.

2.6.11 Decoding

This sections describes the functions that can be used to decode JSON text to the Jansson representation of JSON data. The JSON specification requires that a JSON text is either a serialized array or object, and this requirement is also enforced with the following functions. In other words, the top level value in the JSON text being decoded must be either array or object. To decode any JSON value, use the `JSON_DECODE_ANY` flag (see below).

See *RFC Conformance* for a discussion on Jansson's conformance to the JSON specification. It explains many design decisions that affect especially the behavior of the decoder.

Each function takes a *flags* parameter that can be used to control the behavior of the decoder. Its default value is 0. The following macros can be ORed together to obtain *flags*.

JSON_REJECT_DUPLICATES Issue a decoding error if any JSON object in the input text contains duplicate keys. Without this flag, the value of the last occurence of each key ends up in the result. Key equivalence is checked byte-by-byte, without special Unicode comparison algorithms.

New in version 2.1.

JSON_DECODE_ANY By default, the decoder expects an array or object as the input. With this flag enabled, the decoder accepts any valid JSON value.

Note: Decoding any value may be useful in some scenarios, but it's generally discouraged as it violates strict compatiblity with **RFC 4627**. If you use this flag, don't expect interoperatibility with other JSON systems.

New in version 2.3.

JSON_DISABLE_EOF_CHECK By default, the decoder expects that its whole input constitutes a valid JSON text, and issues an error if there's extra data after the otherwise valid JSON input. With this flag enabled, the decoder stops after decoding a valid JSON array or object, and thus allows extra data after the JSON text.

Normally, reading will stop when the last `]` or `}` in the JSON input is encountered. If both `JSON_DISABLE_EOF_CHECK` and `JSON_DECODE_ANY` flags are used, the decoder may read one extra UTF-8 code unit (up to 4 bytes of input). For example, decoding `4true` correctly decodes the integer 4, but also reads the `t`. For this reason, if reading multiple consecutive values that are not arrays or objects, they should be separated by at least one whitespace character.

New in version 2.1.

JSON_DECODE_INT_AS_REAL JSON defines only one number type. Jansson distinguishes between ints and reals. For more information see *Real vs. Integer*. With this flag enabled the decoder interprets all numbers as real values. Integers that do not have an exact double representation will silently result in a loss of precision. Integers that cause a double overflow will cause an error.

New in version 2.5.

JSON_ALLOW_NUL Allow `\u0000` escape inside string values. This is a safety measure; If you know your input can contain NUL bytes, use this flag. If you don't use this flag, you don't have to worry about NUL bytes inside strings unless you explicitly create themselves by using e.g. *`json_stringn()`* or `s#` format specifier for *`json_pack()`*.

Object keys cannot have embedded NUL bytes even if this flag is used.

New in version 2.6.

Each function also takes an optional *`json_error_t`* parameter that is filled with error information if decoding fails. It's also updated on success; the number of bytes of input read is written to its `position` field. This is especially useful when using `JSON_DISABLE_EOF_CHECK` to read multiple consecutive JSON texts.

New in version 2.3: Number of bytes of input read is written to the `position` field of the *`json_error_t`* structure.

If no error or position information is needed, you can pass *NULL*.

The following functions perform the actual JSON decoding.

json_t ***json_loads**(const char **input*, size_t *flags*, *json_error_t *error*)
Return value: New reference. Decodes the JSON string *input* and returns the array or object it contains, or *NULL* on error, in which case *error* is filled with information about the error. *flags* is described above.

json_t ***json_loadb**(const char **buffer*, size_t *buflen*, size_t *flags*, *json_error_t *error*)
Return value: New reference. Decodes the JSON string *buffer*, whose length is *buflen*, and returns the array or object it contains, or *NULL* on error, in which case *error* is filled with information about the error. This is similar to *`json_loads()`* except that the string doesn't need to be null-terminated. *flags* is described above.

New in version 2.1.

json_t ***json_loadf**(FILE **input*, size_t *flags*, *json_error_t *error*)
Return value: New reference. Decodes the JSON text in stream *input* and returns the array or object it contains, or *NULL* on error, in which case *error* is filled with information about the error. *flags* is described above.

This function will start reading the input from whatever position the input file was, without attempting to seek first. If an error occurs, the file position will be left indeterminate. On success, the file position will be at EOF, unless `JSON_DISABLE_EOF_CHECK` flag was used. In this case, the file position will be at the first character after the last `]` or `}` in the JSON input. This allows calling *`json_loadf()`* on the same `FILE` object multiple times, if the input consists of consecutive JSON texts, possibly separated by whitespace.

json_t ***json_load_file**(const char **path*, size_t *flags*, *json_error_t *error*)
Return value: New reference. Decodes the JSON text in file *path* and returns the array or object it contains, or *NULL* on error, in which case *error* is filled with information about the error. *flags* is described above.

json_load_callback_t
A typedef for a function that's called by *`json_load_callback()`* to read a chunk of input data:

```
typedef size_t (*json_load_callback_t)(void *buffer, size_t buflen, void *data);
```

buffer points to a buffer of *buflen* bytes, and *data* is the corresponding *`json_load_callback()`* argument passed through.

On success, the function should return the number of bytes read; a returned value of 0 indicates that no data was read and that the end of file has been reached. On error, the function should return `(size_t)-1` to abort the decoding process.

New in version 2.4.

json_t ***json_load_callback**(*json_load_callback_t callback*, void **data*, size_t *flags*, *json_error_t *error*)

Return value: New reference. Decodes the JSON text produced by repeated calls to *callback*, and returns the array or object it contains, or *NULL* on error, in which case *error* is filled with information about the error. *data* is passed through to *callback* on each call. *flags* is described above.

New in version 2.4.

2.6.12 Building Values

This section describes functions that help to create, or *pack*, complex JSON values, especially nested objects and arrays. Value building is based on a *format string* that is used to tell the functions about the expected arguments.

For example, the format string `"i"` specifies a single integer value, while the format string `"[ssb]"` or the equivalent `"[s, s, b]"` specifies an array value with two strings and a boolean as its items:

```
/* Create the JSON integer 42 */
json_pack("i", 42);

/* Create the JSON array ["foo", "bar", true] */
json_pack("[ssb]", "foo", "bar", 1);
```

Here's the full list of format specifiers. The type in parentheses denotes the resulting JSON type, and the type in brackets (if any) denotes the C type that is expected as the corresponding argument or arguments.

`s` (string) [const char *] Convert a NULL terminated UTF-8 string to a JSON string.

`s#` (string) [const char *, int] Convert a UTF-8 buffer of a given length to a JSON string.

New in version 2.5.

`s%` (string) [const char *, size_t] Like `s#` but the length argument is of type `size_t`.

New in version 2.6.

`+` [const char *] Like `s`, but concatenate to the previous string. Only valid after `s`, `s#`, `+` or `+#`.

New in version 2.5.

`+#` [const char *, int] Like `s#`, but concatenate to the previous string. Only valid after `s`, `s#`, `+` or `+#`.

New in version 2.5.

`+%` (string) [const char *, size_t] Like `+#` but the length argument is of type `size_t`.

New in version 2.6.

`n` (null) Output a JSON null value. No argument is consumed.

`b` (boolean) [int] Convert a C `int` to JSON boolean value. Zero is converted to `false` and non-zero to `true`.

`i` (integer) [int] Convert a C `int` to JSON integer.

`I` (integer) [json_int_t] Convert a C *`json_int_t`* to JSON integer.

`f` (real) [double] Convert a C `double` to JSON real.

`o` (any value) [json_t *] Output any given JSON value as-is. If the value is added to an array or object, the reference to the value passed to `o` is stolen by the container.

o (any value) [json_t *] Like `o`, but the argument's reference count is incremented. This is useful if you pack into an array or object and want to keep the reference for the JSON value consumed by `O` to yourself.

[fmt] (array) Build an array with contents from the inner format string. `fmt` may contain objects and arrays, i.e. recursive value building is supported.

{fmt} (object) Build an object with contents from the inner format string `fmt`. The first, third, etc. format specifier represent a key, and must be a string (see `s`, `s#`, `+` and `+#` above), as object keys are always strings. The second, fourth, etc. format specifier represent a value. Any value may be an object or array, i.e. recursive value building is supported.

Whitespace, `:` and `,` are ignored.

The following functions compose the value building API:

json_t ***json_pack**(const char **fmt*, ...)

Return value: New reference. Build a new JSON value according to the format string *fmt*. For each format specifier (except for `{}[]n`), one or more arguments are consumed and used to build the corresponding value. Returns *NULL* on error.

json_t ***json_pack_ex**(*json_error_t *error*, size_t *flags*, const char **fmt*, ...)

json_t ***json_vpack_ex**(*json_error_t *error*, size_t *flags*, const char **fmt*, va_list *ap*)

Return value: New reference. Like *json_pack()*, but an in the case of an error, an error message is written to *error*, if it's not *NULL*. The *flags* parameter is currently unused and should be set to 0.

As only the errors in format string (and out-of-memory errors) can be caught by the packer, these two functions are most likely only useful for debugging format strings.

More examples:

```
/* Build an empty JSON object */
json_pack("{}");

/* Build the JSON object {"foo": 42, "bar": 7} */
json_pack("{sisi}", "foo", 42, "bar", 7);

/* Like above, ':', ',' and whitespace are ignored */
json_pack("{s:i, s:i}", "foo", 42, "bar", 7);

/* Build the JSON array [[1, 2], {"cool": true}] */
json_pack("[[i,i],{s:b}]", 1, 2, "cool", 1);

/* Build a string from a non-NUL terminated buffer */
char buffer[4] = {'t', 'e', 's', 't'};
json_pack("s#", buffer, 4);

/* Concatentate strings together to build the JSON string "foobarbaz" */
json_pack("s++", "foo", "bar", "baz");
```

2.6.13 Parsing and Validating Values

This section describes functions that help to validate complex values and extract, or *unpack*, data from them. Like *building values*, this is also based on format strings.

While a JSON value is unpacked, the type specified in the format string is checked to match that of the JSON value. This is the validation part of the process. In addition to this, the unpacking functions can also check that all items of arrays and objects are unpacked. This check be enabled with the format specifier `!` or by using the flag `JSON_STRICT`. See below for details.

Here's the full list of format specifiers. The type in parentheses denotes the JSON type, and the type in brackets (if any) denotes the C type whose address should be passed.

s (string) [const char *] Convert a JSON string to a pointer to a NULL terminated UTF-8 string. The resulting string is extracted by using *json_string_value()* internally, so it exists as long as there are still references to the corresponding JSON string.

s% (string) [const char *, size_t *] Convert a JSON string to a pointer to a NULL terminated UTF-8 string and its length.

New in version 2.6.

n (null) Expect a JSON null value. Nothing is extracted.

b (boolean) [int] Convert a JSON boolean value to a C `int`, so that `true` is converted to 1 and `false` to 0.

i (integer) [int] Convert a JSON integer to C `int`.

I (integer) [json_int_t] Convert a JSON integer to C *json_int_t*.

f (real) [double] Convert a JSON real to C `double`.

F (integer or real) [double] Convert a JSON number (integer or real) to C `double`.

o (any value) [json_t *] Store a JSON value with no conversion to a *json_t* pointer.

O (any value) [json_t *] Like `O`, but the JSON value's reference count is incremented.

[fmt] (array) Convert each item in the JSON array according to the inner format string. `fmt` may contain objects and arrays, i.e. recursive value extraction is supporetd.

{fmt} (object) Convert each item in the JSON object according to the inner format string `fmt`. The first, third, etc. format specifier represent a key, and must be `s`. The corresponding argument to unpack functions is read as the object key. The second fourth, etc. format specifier represent a value and is written to the address given as the corresponding argument. **Note** that every other argument is read from and every other is written to.

`fmt` may contain objects and arrays as values, i.e. recursive value extraction is supporetd.

New in version 2.3: Any `s` representing a key may be suffixed with a `?` to make the key optional. If the key is not found, nothing is extracted. See below for an example.

! This special format specifier is used to enable the check that all object and array items are accessed, on a per-value basis. It must appear inside an array or object as the last format specifier before the closing bracket or brace. To enable the check globally, use the `JSON_STRICT` unpacking flag.

***** This special format specifier is the opposite of `!`. If the `JSON_STRICT` flag is used, `*` can be used to disable the strict check on a per-value basis. It must appear inside an array or object as the last format specifier before the closing bracket or brace.

Whitespace, **:** and **,** are ignored.

The following functions compose the parsing and validation API:

int **json_unpack**(*json_t* **root*, const char **fmt*, ...)
Validate and unpack the JSON value *root* according to the format string *fmt*. Returns 0 on success and -1 on failure.

int **json_unpack_ex**(*json_t* **root*, *json_error_t* **error*, size_t *flags*, const char **fmt*, ...)
int **json_vunpack_ex**(*json_t* **root*, *json_error_t* **error*, size_t *flags*, const char **fmt*, va_list *ap*)
Validate and unpack the JSON value *root* according to the format string *fmt*. If an error occurs and *error* is not *NULL*, write error information to *error*. *flags* can be used to control the behaviour of the unpacker, see below for the flags. Returns 0 on success and -1 on failure.

Note: The first argument of all unpack functions is `json_t *root` instead of `const json_t *root`, because the use of `O` format specifier causes the reference count of `root`, or some value reachable from `root`, to be increased. Furthermore, the `o` format specifier may be used to extract a value as-is, which allows modifying the structure or contents of a value reachable from `root`.

If the `O` and `o` format specifiers are not used, it's perfectly safe to cast a `const json_t *` variable to plain `json_t *` when used with these functions.

The following unpacking flags are available:

JSON_STRICT Enable the extra validation step checking that all object and array items are unpacked. This is equivalent to appending the format specifier `!` to the end of every array and object in the format string.

JSON_VALIDATE_ONLY Don't extract any data, just validate the JSON value against the given format string. Note that object keys must still be specified after the format string.

Examples:

```
/* root is the JSON integer 42 */
int myint;
json_unpack(root, "i", &myint);
assert(myint == 42);

/* root is the JSON object {"foo": "bar", "quux": true} */
const char *str;
int boolean;
json_unpack(root, "{s:s, s:b}", "foo", &str, "quux", &boolean);
assert(strcmp(str, "bar") == 0 && boolean == 1);

/* root is the JSON array [[1, 2], {"baz": null} */
json_error_t error;
json_unpack_ex(root, &error, JSON_VALIDATE_ONLY, "[[i,i], {s:n}]", "baz");
/* returns 0 for validation success, nothing is extracted */

/* root is the JSON array [1, 2, 3, 4, 5] */
int myint1, myint2;
json_unpack(root, "[ii!]", &myint1, &myint2);
/* returns -1 for failed validation */

/* root is an empty JSON object */
int myint = 0, myint2 = 0;
json_unpack(root, "{s?i, s?[ii]}",
            "foo", &myint1,
            "bar", &myint2, &myint3);
/* myint1, myint2 or myint3 is no touched as "foo" and "bar" don't exist */
```

2.6.14 Equality

Testing for equality of two JSON values cannot, in general, be achieved using the == operator. Equality in the terms of the == operator states that the two *json_t* pointers point to exactly the same JSON value. However, two JSON values can be equal not only if they are exactly the same value, but also if they have equal "contents":

- Two integer or real values are equal if their contained numeric values are equal. An integer value is never equal to a real value, though.
- Two strings are equal if their contained UTF-8 strings are equal, byte by byte. Unicode comparison algorithms are not implemented.

- Two arrays are equal if they have the same number of elements and each element in the first array is equal to the corresponding element in the second array.
- Two objects are equal if they have exactly the same keys and the value for each key in the first object is equal to the value of the corresponding key in the second object.
- Two true, false or null values have no "contents", so they are equal if their types are equal. (Because these values are singletons, their equality can actually be tested with ==.)

The following function can be used to test whether two JSON values are equal.

int **json_equal**(*json_t *value1*, *json_t *value2*)
Returns 1 if *value1* and *value2* are equal, as defined above. Returns 0 if they are inequal or one or both of the pointers are *NULL*.

2.6.15 Copying

Because of reference counting, passing JSON values around doesn't require copying them. But sometimes a fresh copy of a JSON value is needed. For example, if you need to modify an array, but still want to use the original afterwards, you should take a copy of it first.

Jansson supports two kinds of copying: shallow and deep. There is a difference between these methods only for arrays and objects. Shallow copying only copies the first level value (array or object) and uses the same child values in the copied value. Deep copying makes a fresh copy of the child values, too. Moreover, all the child values are deep copied in a recursive fashion.

json_t ***json_copy**(*json_t *value*)
Return value: New reference. Returns a shallow copy of *value*, or *NULL* on error.

json_t ***json_deep_copy**(const *json_t *value*)
Return value: New reference. Returns a deep copy of *value*, or *NULL* on error.

2.6.16 Custom Memory Allocation

By default, Jansson uses `malloc()` and `free()` for memory allocation. These functions can be overridden if custom behavior is needed.

json_malloc_t
A typedef for a function pointer with `malloc()`'s signature:

```
typedef void *(*json_malloc_t)(size_t);
```

json_free_t
A typedef for a function pointer with `free()`'s signature:

```
typedef void (*json_free_t)(void *);
```

void **json_set_alloc_funcs**(*json_malloc_t malloc_fn*, *json_free_t free_fn*)
Use *malloc_fn* instead of `malloc()` and *free_fn* instead of `free()`. This function has to be called before any other Jansson's API functions to ensure that all memory operations use the same functions.

Examples:

Circumvent problems with different CRT heaps on Windows by using application's `malloc()` and `free()`:

```
json_set_alloc_funcs(malloc, free);
```

Use the Boehm's conservative garbage collector for memory operations:

```
json_set_alloc_funcs(GC_malloc, GC_free);
```

Allow storing sensitive data (e.g. passwords or encryption keys) in JSON structures by zeroing all memory when freed:

```
static void *secure_malloc(size_t size)
{
    /* Store the memory area size in the beginning of the block */
    void *ptr = malloc(size + 8);
    *((size_t *)ptr) = size;
    return ptr + 8;
}

static void secure_free(void *ptr)
{
    size_t size;

    ptr -= 8;
    size = *((size_t *)ptr);

    guaranteed_memset(ptr, 0, size + 8);
    free(ptr);
}

int main()
{
    json_set_alloc_funcs(secure_malloc, secure_free);
    /* ... */
}
```

For more information about the issues of storing sensitive data in memory, see http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/protect-secrets.html. The page also explains the `guaranteed_memset()` function used in the example and gives a sample implementation for it.

2.7 Changes in Jansson

2.7.1 Version 2.7

Released 2014-10-02

- New features:
  - *`json_pack()`* and friends: Add format specifiers `s%` and `+%` for a size_t string length (#141).
  - *`json_unpack()`* and friends: Add format specifier `s%` for unpacking the string length along with the string itself (#141).
  - Add length-aware string constructors *`json_stringn()`* and *`json_stringn_nocheck()`*, length-aware string mutators *`json_string_setn()`* and *`json_string_setn_nocheck()`*, and a function for getting string's length *`json_string_length()`* (#141, #143).
  - Support `\u0000` escapes in the decoder. The support can be enabled by using the `JSON_ALLOW_NUL` decoding flag (#141).
  - Add *`json_boolean_value()`* as an alias for *`json_is_true()`* (#146).
  - Add JSON_REAL_PRECISION encoding flag/macro for controlling real number precision (#178).

  - Define the maximum indentation as JSON_MAX_INDENT (#191).
- Bug fixes:
  - Some malformed `\uNNNN` escapes could crash the decoder with an assertion failure.
  - Avoid integer overflows with very long strings in UTF-8 decoder and hashtable.
  - Check for *NULL* key in `json_object_get()` and `json_object_del()` (#151).
  - Enhance hashtable seeding on Windows (#162).
  - `json_unpack()`: Allow mixing JSON_STRICT with optional keys (#162, #163).
  - Fix int/int32 mismatch (#142).
  - Parse subnormal numbers correctly (#202).
- Build:
  - Remove VS2010 build files. CMake should be used on Windows instead (#165).
  - Fix CMake build flags for MinGW (#193).
  - Add CMake config files for find_package. Rename config.h to jansson_private_config.h (#157, #159).
  - Make Valgrind checks work with CMake (#160).
  - Fix feature checks to use correct __ATOMIC flags.
  - Fix CMake checks for uint16_t and uint8_t support (#177).
  - Make Jansson build on SmartOS/Solaris (#171).
  - Work around a GCC bug on Solaris (#175).
  - Fix autoreconf on Debian (#182).
  - Don't use GNU make specific export for global AM_CFLAGS (#203, #204).
  - Fix building on Android using the supplied Android.mk (#166, #174).
  - Android.mk: Add -DHAVE_STDINT_H to LOCAL_CFLAGS (#200).
- Documentation:
  - Document JANSSON_BUILD_SHARED_LIBS CMake option (#187).
- Tests:
  - Close file handles correctly (#198).
- Other changes:
  - `\uNNNN` escapes are now encoded in upper case for better readability.
  - Enable usage of AddressSanitizer (#180).

2.7.2 Version 2.6

Released 2014-02-11

- Security:
  - CVE-2013-6401: The hash function used by the hashtable implementation has been changed, and is automatically seeded with random data when the first JSON object is created. This prevents an attacker from causing large JSON objects with specially crafted keys perform poorly.
- New features:

  - *json_object_seed()*: Set the seed value of the hash function.
- Bug fixes:
  - Include CMake specific files in the release tarball.
- Documentation:
  - Fix tutorial source to send a User-Agent header, which is now required by the GitHub API.
  - Set all memory to zero in secure_free() example.

2.7.3 Version 2.5

Released 2013-09-19

- New features:
  - *json_pack()* and friends: Add format specifiers `s#`, `+` and `+#`.
  - Add `JSON_DECODE_INT_AS_REAL` decoding flag to treat all numbers as real in the decoder (#123).
  - Add *json_array_foreach()*, paralleling *json_object_foreach()* (#118).
- Bug fixes:
  - *json_dumps()* and friends: Don't crash if json is *NULL* and `JSON_ENCODE_ANY` is set.
  - Fix a theoretical integer overflow in `jsonp_strdup()`.
  - Fix `l_isxdigit()` macro (#97).
  - Fix an off-by-one error in *json_array_remove()*.
- Build:
  - Support CMake in addition to GNU Autotools (#106, #107, #112, #115, #120, #127).
  - Support building for Android (#109).
  - Don't use `-Werror` by default.
  - Support building and testing with VPATH (#93).
  - Fix compilation when `NDEBUG` is defined (#128)
- Tests:
  - Fix a refleak in `test/bin/json_process.c`.
- Documentation:
  - Clarify the return value of *json_load_callback_t()*.
  - Document how to circumvent problems with separate heaps on Windows.
  - Fix memory leaks and warnings in `github_commits.c`.
  - Use *json_decref()* properly in tutorial.
- Other:
  - Make it possible to forward declare `struct json_t`.

2.7.4 Version 2.4

Released 2012-09-23

- New features:
  - Add *json_boolean()* macro that returns the JSON true or false value based on its argument (#86).
  - Add *json_load_callback()* that calls a callback function repeatedly to read the JSON input (#57).
  - Add JSON_ESCAPE_SLASH encoding flag to escape all occurences of `/` with `\/`.
- Bug fixes:
  - Check for and reject NaN and Inf values for reals. Encoding these values resulted in invalid JSON.
  - Fix *json_real_set()* to return -1 on error.
- Build:
  - Jansson now builds on Windows with Visual Studio 2010, and includes solution and project files in `win32/vs2010/` directory.
  - Fix build warnings (#77, #78).
  - Add `-no-undefined` to LDFLAGS (#90).
- Tests:
  - Fix the symbol exports test on Linux/PPC64 (#88).
- Documentation:
  - Fix typos (#73, #84).

2.7.5 Version 2.3.1

Released 2012-04-20

- Build issues:
  - Only use `long long` if `strtoll()` is also available.
- Documentation:
  - Fix the names of library version constants in documentation. (#52)
  - Change the tutorial to use GitHub API v3. (#65)
- Tests:
  - Make some tests locale independent. (#51)
  - Distribute the library exports test in the tarball.
  - Make test run on shells that don't support the `export FOO=bar` syntax.

2.7.6 Version 2.3

Released 2012-01-27

- New features:
  - *json_unpack()* and friends: Add support for optional object keys with the `{s?o}` syntax.

  - Add *json_object_update_existing()* and *json_object_update_missing()*, for updating only existing keys or only adding missing keys to an object. (#37)
  - Add *json_object_foreach()* for more convenient iteration over objects. (#45, #46)
  - When decoding JSON, write the number of bytes that were read from input to `error.position` also on success. This is handy with `JSON_DISABLE_EOF_CHECK`.
  - Add support for decoding any JSON value, not just arrays or objects. The support is enabled with the new `JSON_DECODE_ANY` flag. Patch by Andrea Marchesini. (#4)
- Bug fixes
  - Avoid problems with object's serial number growing too big. (#40, #41)
  - Decoding functions now return NULL if the first argument is NULL. Patch by Andrea Marchesini.
  - Include `jansson_config.h.win32` in the distribution tarball.
  - Remove + and leading zeros from exponents in the encoder. (#39)
  - Make Jansson build and work on MinGW. (#39, #38)
- Documentation
  - Note that the same JSON values must not be encoded in parallel by separate threads. (#42)
  - Document MinGW support.

2.7.7 Version 2.2.1

Released 2011-10-06

- Bug fixes:
  - Fix real number encoding and decoding under non-C locales. (#32)
  - Fix identifier decoding under non-UTF-8 locales. (#35)
  - *json_load_file()*: Open the input file in binary mode for maximum compatiblity.
- Documentation:
  - Clarify the lifecycle of the result of the `s` fromat of *json_unpack()*. (#31)
  - Add some portability info. (#36)
  - Little clarifications here and there.
- Other:
  - Some style fixes, issues detected by static analyzers.

2.7.8 Version 2.2

Released 2011-09-03

- New features:
  - *json_dump_callback()*: Pass the encoder output to a callback function in chunks.
- Bug fixes:
  - *json_string_set()*: Check that target is a string and value is not NULL.
- Other:

  - Documentation typo fixes and clarifications.

2.7.9 Version 2.1

Released 2011-06-10

- New features:
  - *json_loadb()*: Decode a string with a given size, useful if the string is not null terminated.
  - Add JSON_ENCODE_ANY encoding flag to allow encoding any JSON value. By default, only arrays and objects can be encoded. (#19)
  - Add JSON_REJECT_DUPLICATES decoding flag to issue a decoding error if any JSON object in the input contins duplicate keys. (#3)
  - Add JSON_DISABLE_EOF_CHECK decoding flag to stop decoding after a valid JSON input. This allows other data after the JSON data.
- Bug fixes:
  - Fix an additional memory leak when memory allocation fails in *json_object_set()* and friends.
  - Clear errno before calling strtod() for better portability. (#27)
- Building:
  - Avoid set-but-not-used warning/error in a test. (#20)
- Other:
  - Minor clarifications to documentation.

2.7.10 Version 2.0.1

Released 2011-03-31

- Bug fixes:
  - Replace a few malloc() and free() calls with their counterparts that support custom memory management.
  - Fix object key hashing in json_unpack() strict checking mode.
  - Fix the parentheses in JANSSON_VERSION_HEX macro.
  - Fix *json_object_size()* return value.
  - Fix a few compilation issues.
- Portability:
  - Enhance portability of va_copy().
  - Test framework portability enhancements.
- Documentation:
  - Distribute doc/upgrading.rst with the source tarball.
  - Build documentation in strict mode in make distcheck.

2.7.11 Version 2.0

Released 2011-02-28

This release is backwards incompatible with the 1.x release series. See the chapter "Upgrading from older versions" in documentation for details.

- Backwards incompatible changes:
  - Unify unsigned integer usage in the API: All occurences of unsigned int and unsigned long have been replaced with size_t.
  - Change JSON integer's underlying type to the widest signed integer type available, i.e. long long if it's supported, otherwise long. Add a typedef json_int_t that defines the type.
  - Change the maximum indentation depth to 31 spaces in encoder. This frees up bits from the flags parameter of encoding functions *json_dumpf()*, *json_dumps()* and *json_dump_file()*.
  - For future needs, add a flags parameter to all decoding functions *json_loadf()*, *json_loads()* and *json_load_file()*.
- New features
  - *json_pack()*, *json_pack_ex()*, *json_vpack_ex()*: Create JSON values based on a format string.
  - *json_unpack()*, *json_unpack_ex()*, *json_vunpack_ex()*: Simple value extraction and validation functionality based on a format string.
  - Add column, position and source fields to the `json_error_t` struct.
  - Enhance error reporting in the decoder.
  - `JANSSON_VERSION` et al.: Preprocessor constants that define the library version.
  - *json_set_alloc_funcs()*: Set custom memory allocation functions.
- Fix many portability issues, especially on Windows.
- Configuration
  - Add file `jansson_config.h` that contains site specific configuration. It's created automatically by the configure script, or can be created by hand if the configure script cannot be used. The file `jansson_config.h.win32` can be used without modifications on Windows systems.
  - Add a section to documentation describing how to build Jansson on Windows.
  - Documentation now requires Sphinx 1.0 or newer.

2.7.12 Version 1.3

Released 2010-06-13

- New functions:
  - *json_object_iter_set()*, *json_object_iter_set_new()*: Change object contents while iterating over it.
  - *json_object_iter_at()*: Return an iterator that points to a specific object item.
- New encoding flags:
  - `JSON_PRESERVE_ORDER`: Preserve the insertion order of object keys.
- Bug fixes:

  - Fix an error that occured when an array or object was first encoded as empty, then populated with some data, and then re-encoded
  - Fix the situation like above, but when the first encoding resulted in an error
- Documentation:
  - Clarify the documentation on reference stealing, providing an example usage pattern

2.7.13 Version 1.2.1

Released 2010-04-03

- Bug fixes:
  - Fix reference counting on `true`, `false` and `null`
  - Estimate real number underflows in decoder with 0.0 instead of issuing an error
- Portability:
  - Make `int32_t` available on all systems
  - Support compilers that don't have the `inline` keyword
  - Require Autoconf 2.60 (for `int32_t`)
- Tests:
  - Print test names correctly when `VERBOSE=1`
  - `test/suites/api`: Fail when a test fails
  - Enhance tests for iterators
  - Enhance tests for decoding texts that contain null bytes
- Documentation:
  - Don't remove `changes.rst` in `make clean`
  - Add a chapter on RFC conformance

2.7.14 Version 1.2

Released 2010-01-21

- New functions:
  - *`json_equal()`*: Test whether two JSON values are equal
  - *`json_copy()`* and *`json_deep_copy()`*: Make shallow and deep copies of JSON values
  - Add a version of all functions taking a string argument that doesn't check for valid UTF-8: *`json_string_nocheck()`*, *`json_string_set_nocheck()`*, *`json_object_set_nocheck()`*, *`json_object_set_new_nocheck()`*
- New encoding flags:
  - `JSON_SORT_KEYS`: Sort objects by key
  - `JSON_ENSURE_ASCII`: Escape all non-ASCII Unicode characters
  - `JSON_COMPACT`: Use a compact representation with all unneeded whitespace stripped
- Bug fixes:

  - Revise and unify whitespace usage in encoder: Add spaces between array and object items, never append newline to output.
  - Remove const qualifier from the `json_t` parameter in *`json_string_set()`*, *`json_integer_set()`* and *`json_real_set()`*.
  - Use `int32_t` internally for representing Unicode code points (int is not enough on all platforms)
- Other changes:
  - Convert `CHANGES` (this file) to reStructured text and add it to HTML documentation
  - The test system has been refactored. Python is no longer required to run the tests.
  - Documentation can now be built by invoking `make html`
  - Support for pkg-config

2.7.15 Version 1.1.3

Released 2009-12-18

- Encode reals correctly, so that first encoding and then decoding a real always produces the same value
- Don't export private symbols in `libjansson.so`

2.7.16 Version 1.1.2

Released 2009-11-08

- Fix a bug where an error message was not produced if the input file could not be opened in *`json_load_file()`*
- Fix an assertion failure in decoder caused by a minus sign without a digit after it
- Remove an unneeded include of `stdint.h` in `jansson.h`

2.7.17 Version 1.1.1

Released 2009-10-26

- All documentation files were not distributed with v1.1; build documentation in make distcheck to prevent this in the future
- Fix v1.1 release date in `CHANGES`

2.7.18 Version 1.1

Released 2009-10-20

- API additions and improvements:
  - Extend array and object APIs
  - Add functions to modify integer, real and string values
  - Improve argument validation
  - Use unsigned int instead of `uint32_t` for encoding flags

- Enhance documentation
  - Add getting started guide and tutorial
  - Fix some typos
  - General clarifications and cleanup
- Check for integer and real overflows and underflows in decoder
- Make singleton values thread-safe (`true`, `false` and `null`)
- Enhance circular reference handling
- Don't define `-std=c99` in `AM_CFLAGS`
- Add C++ guards to `jansson.h`
- Minor performance and portability improvements
- Expand test coverage

2.7.19 Version 1.0.4

Released 2009-10-11

- Relax Autoconf version requirement to 2.59
- Make Jansson compile on platforms where plain `char` is unsigned
- Fix API tests for object

2.7.20 Version 1.0.3

Released 2009-09-14

- Check for integer and real overflows and underflows in decoder
- Use the Python json module for tests, or simplejson if the json module is not found
- Distribute changelog (this file)

2.7.21 Version 1.0.2

Released 2009-09-08

- Handle EOF correctly in decoder

2.7.22 Version 1.0.1

Released 2009-09-04

- Fixed broken *`json_is_boolean()`*

2.7.23 Version 1.0

Released 2009-08-25

- Initial release

CHAPTER 3

Indices and Tables

- genindex
- search

J

json_array (C function), 20
json_array_append (C function), 21
json_array_append_new (C function), 21
json_array_clear (C function), 21
json_array_extend (C function), 21
json_array_foreach (C function), 21
json_array_get (C function), 20
json_array_insert (C function), 21
json_array_insert_new (C function), 21
json_array_remove (C function), 21
json_array_set (C function), 20
json_array_set_new (C function), 20
json_array_size (C function), 20
json_boolean (C function), 18
json_boolean_value (C function), 17
json_copy (C function), 32
json_decref (C function), 17
json_deep_copy (C function), 32
json_dump_callback (C function), 26
json_dump_callback_t (C type), 26
json_dump_file (C function), 26
json_dumpf (C function), 25
json_dumps (C function), 25
json_equal (C function), 32
json_error_t (C type), 24
json_error_t.column (C member), 24
json_error_t.line (C member), 24
json_error_t.position (C member), 24
json_false (C function), 18
json_free_t (C type), 32
json_incref (C function), 17
json_int_t (C type), 19
json_integer (C function), 20
json_integer_set (C function), 20
json_integer_value (C function), 20
json_is_array (C function), 16
json_is_boolean (C function), 17
json_is_false (C function), 16
json_is_integer (C function), 16
json_is_null (C function), 16
json_is_number (C function), 17
json_is_object (C function), 16
json_is_real (C function), 16
json_is_string (C function), 16
json_is_true (C function), 16
json_load_callback (C function), 28
json_load_callback_t (C type), 27
json_load_file (C function), 27
json_loadb (C function), 27
json_loadf (C function), 27
json_loads (C function), 27
json_malloc_t (C type), 32
json_null (C function), 18
json_number_value (C function), 20
json_object (C function), 21
json_object_clear (C function), 22
json_object_del (C function), 22
json_object_foreach (C function), 22
json_object_get (C function), 22
json_object_iter (C function), 23
json_object_iter_at (C function), 23
json_object_iter_key (C function), 23
json_object_iter_next (C function), 23
json_object_iter_set (C function), 23
json_object_iter_set_new (C function), 23
json_object_iter_value (C function), 23
json_object_key_to_iter (C function), 23
json_object_seed (C function), 23
json_object_set (C function), 22
json_object_set_new (C function), 22
json_object_set_new_nocheck (C function), 22
json_object_set_nocheck (C function), 22
json_object_size (C function), 21
json_object_update (C function), 22
json_object_update_existing (C function), 22
json_object_update_missing (C function), 22
json_pack (C function), 29
json_pack_ex (C function), 29
json_real (C function), 20
json_real_set (C function), 20

json_real_value (C function), 20
json_set_alloc_funcs (C function), 32
json_string (C function), 19
json_string_length (C function), 19
json_string_nocheck (C function), 19
json_string_set (C function), 19
json_string_set_nocheck (C function), 19
json_string_setn (C function), 19
json_string_setn_nocheck (C function), 19
json_string_value (C function), 19
json_stringn (C function), 19
json_stringn_nocheck (C function), 19
json_t (C type), 16
json_true (C function), 18
json_type (C type), 16
json_typeof (C function), 16
json_unpack (C function), 30
json_unpack_ex (C function), 30
json_vpack_ex (C function), 29
json_vunpack_ex (C function), 30

R

RFC
    RFC 4627, 13, 16, 25, 26