

NAME

libmaxminddb – a library for working with MaxMind DB files

SYNOPSIS

```
#include <maxminddb.h>

int MMDB_open(
    const char *const filename,
    uint32_t flags,
    MMDB_s *const mmdb);
void MMDB_close(MMDB_s *const mmdb);

MMDB_lookup_result_s MMDB_lookup_string(
    MMDB_s *const mmdb,
    const char *const ipstr,
    int *const gai_error,
    int *const mmdb_error);
MMDB_lookup_result_s MMDB_lookup_sockaddr(
    MMDB_s *const mmdb,
    const struct sockaddr *const
    sockaddr,
    int *const mmdb_error);

int MMDB_get_value(
    MMDB_entry_s *const start,
    MMDB_entry_data_s *const entry_data,
    ...);
int MMDB_vget_value(
    MMDB_entry_s *const start,
    MMDB_entry_data_s *const entry_data,
    va_list va_path);
int MMDB_aget_value(
    MMDB_entry_s *const start,
    MMDB_entry_data_s *const entry_data,
    const char *const *const path);

int MMDB_get_entry_data_list(
    MMDB_entry_s *start,
    MMDB_entry_data_list_s **const entry_data_list);
void MMDB_free_entry_data_list(
    MMDB_entry_data_list_s *const entry_data_list);
int MMDB_get_metadata_as_entry_data_list(
    MMDB_s *const mmdb,
    MMDB_entry_data_list_s **const entry_data_list);
int MMDB_dump_entry_data_list(
    FILE *const stream,
    MMDB_entry_data_list_s *const entry_data_list,
    int indent);

int MMDB_read_node(
    MMDB_s *const mmdb,
    uint32_t node_number,
    MMDB_search_node_s *const node);
```

```

const char *MMDB_lib_version(void);
const char *MMDB_strerror(int error_code);

typedef struct MMDB_lookup_result_s {
    bool found_entry;
    MMDB_entry_s entry;
    uint16_t netmask;
} MMDB_lookup_result_s;

typedef struct MMDB_entry_data_s {
    bool has_data;
    union {
        uint32_t pointer;
        const char *utf8_string;
        double double_value;
        const uint8_t *bytes;
        uint16_t uint16;
        uint32_t uint32;
        int32_t int32;
        uint64_t uint64;
        {mmdb_uint128_t or uint8_t[16]} uint128;
        bool boolean;
        float float_value;
    };
    ...
    uint32_t data_size;
    uint32_t type;
} MMDB_entry_data_s;

typedef struct MMDB_entry_data_list_s {
    MMDB_entry_data_s entry_data;
    struct MMDB_entry_data_list_s *next;
} MMDB_entry_data_list_s;

```

DESCRIPTION

The libmaxminddb library provides functions for working MaxMind DB files. See <http://maxmind.github.io/MaxMind-DB/> for the MaxMind DB format specification. The database and results are all represented by different data structures. Databases are opened by calling `MMDB_open()`. You can look up IP addresses as a string with `MMDB_lookup_string()` or as a pointer to a `sockaddr` structure with `MMDB_lookup_sockaddr()`.

If the lookup finds the IP address in the database, it returns a `MMDB_lookup_result_s` structure. If that structure indicates that the database has data for the IP, there are a number of functions that can be used to fetch that data. These include `MMDB_get_value()` and `MMDB_get_entry_data_list()`. See the function documentation below for more details.

When you are done with the database handle you should call `MMDB_close()`.

All publicly visible functions, structures, and macros begin with "MMDB_".

DATA STRUCTURES

All data structures exported by this library's `maxminddb.h` header are typedef'd in the form `typedef struct foo_s { ... } foo_s` so you can refer to them without the `struct` prefix.

This library provides the following data structures:

MMDB_s

This is the handle for a MaxMind DB file. We only document some of this structure's fields intended for public use. All other fields are subject to change and are intended only for internal use.

```
typedef struct MMDB_s {
    uint32_t flags;
    const char *filename;
    ...
    MMDB_metadata_s metadata;
} MMDB_s;
```

- `uint32_t flags` – the flags this database was opened with. See the `MMDB_open()` documentation for more details.
- `const char *filename` – the name of the file which was opened, as passed to `MMDB_open()`.
- `MMDB_metadata_s metadata` – the metadata for the database.

`MMDB_metadata_s` **and** `MMDB_description_s`

This structure can be retrieved from the `MMDB_s` structure. It contains the metadata read from the database file. Note that you may find it more convenient to access this metadata by calling `MMDB_get_metadata_as_entry_data_list()` instead.

```
typedef struct MMDB_metadata_s {
    uint32_t node_count;
    uint16_t record_size;
    uint16_t ip_version;
    const char *database_type;
    struct {
        size_t count;
        const char **names;
    } languages;
    uint16_t binary_format_major_version;
    uint16_t binary_format_minor_version;
    uint64_t build_epoch;
    struct {
        size_t count;
        MMDB_description_s **descriptions;
    } description;
} MMDB_metadata_s;

typedef struct MMDB_description_s {
    const char *language;
    const char *description;
} MMDB_description_s;
```

These structures should be mostly self-explanatory.

The `ip_version` member should always be 4 or 6. The `binary_format_major_version` should always be 2.

There is no requirement that the database metadata include languages or descriptions, so the count for these parts of the metadata can be zero. All of the other `MMDB_metadata_s` fields should be populated.

`MMDB_lookup_result_s`

This structure is returned as the result of looking up an IP address.

```
typedef struct MMDB_lookup_result_s {
    bool found_entry;
    MMDB_entry_s entry;
    uint16_t netmask;
```

```
    } MMDB_lookup_result_s;
```

If the `found_entry` member is false then the other members of this structure do not contain meaningful values. Always check that `found_entry` is true first.

The `entry` member is used to look up the data associated with the IP address.

The `netmask` member tells you what subnet the IP address belongs to in this database. For example, if you look up the address 1.1.1.1 in an IPv4 database and the returned `netmask` is 16, then the address is part of the 1.1.0.0/16 subnet.

If the database is an IPv6 database, the returned `netmask` is always an IPv6 prefix length (from 0–128), even if that database *also* contains IPv4 networks. If you look up an IPv4 address and would like to turn the `netmask` into an IPv4 netmask value, you can simply subtract 96 from the value.

MMDB_result_s

You don't really need to dig around in this structure. You'll get this from a `MMDB_lookup_result_s` structure and pass it to various functions.

MMDB_entry_data_s

This structure is used to return a single data section entry for an IP. These entries can in turn point to other entries, as is the case for things like maps and arrays. Some members of this structure are not documented as they are only for internal use.

```
typedef struct MMDB_entry_data_s {
    bool has_data;
    union {
        uint32_t pointer;
        const char *utf8_string;
        double double_value;
        const uint8_t *bytes;
        uint16_t uint16;
        uint32_t uint32;
        int32_t int32;
        uint64_t uint64;
        {mmdb_uint128_t or uint8_t[16]} uint128;
        bool boolean;
        float float_value;
    };
    ...
    uint32_t data_size;
    uint32_t type;
} MMDB_entry_data_s;
```

The `has_data` member is true if data was found for a given lookup. See `MMDB_get_value()` for more details. If this member is false then none of the other values in the structure are meaningful.

The union at the beginning of the structure defines the actual data. To determine which union member is populated you should look at the `type` member. The `pointer` member of the union should never be populated in any data returned by the API. Pointers should always be resolved internally.

The `data_size` member is only relevant for `utf8_string` and `bytes` data. `utf8_string` is not null terminated and `data_size` *must* be used to determine its length.

The `type` member can be compared to one of the `MMDB_DTYPE_*` macros.

128-bit Integers

The handling of `uint128` data depends on how your platform supports 128-bit integers, if it does so at all. With GCC 4.4 and 4.5 we can write `unsigned int __attribute__((__mode__(TI)))`. With newer versions of GCC (4.6+) and clang (3.2+) we can simply write "unsigned __int128".

In order to work around these differences, this library defines an `mmdb_uint128_t` type. This type is defined in the `maxminddb.h` header so you can use it in your own code.

With older compilers, we can't use an integer so we instead use a 16 byte array of `uint8_t` values. This is the raw data from the database.

This library provides a public macro `MMDB_UINT128_IS_BYTE_ARRAY` macro. If this is true (1), then `uint128` values are returned as a byte array, if it is false then they are returned as a `mmdb_uint128_t` integer.

Data Type Macros

This library provides a macro for every data type defined by the MaxMind DB spec.

- `MMDB_DATA_TYPE_UTF8_STRING`
- `MMDB_DATA_TYPE_DOUBLE`
- `MMDB_DATA_TYPE_BYTES`
- `MMDB_DATA_TYPE_UINT16`
- `MMDB_DATA_TYPE_UINT32`
- `MMDB_DATA_TYPE_MAP`
- `MMDB_DATA_TYPE_INT32`
- `MMDB_DATA_TYPE_UINT64`
- `MMDB_DATA_TYPE_UINT128`
- `MMDB_DATA_TYPE_ARRAY`
- `MMDB_DATA_TYPE_BOOLEAN`
- `MMDB_DATA_TYPE_FLOAT`

There are also a few types that are for internal use only:

- `MMDB_DATA_TYPE_EXTENDED`
- `MMDB_DATA_TYPE_POINTER`
- `MMDB_DATA_TYPE_CONTAINER`
- `MMDB_DATA_TYPE_END_MARKER`

If you see one of these in returned data then something has gone very wrong. The database is damaged or was generated incorrectly or there is a bug in the `libmaxminddb` code.

Pointer Values and `MMDB_close()`

The `utf8_string`, `bytes`, and (maybe) the `uint128` members of this structure are all pointers directly into the database's data section. This can either be a `malloc'd` or `mmap'd` block of memory. In either case, these pointers will become invalid after `MMDB_close()` is called.

If you need to refer to this data after that time you should copy the data with an appropriate function (`strdup`, `memcpy`, etc.).

`MMDB_entry_data_list_s`

This structure encapsulates a linked list of `MMDB_entry_data_s` structures.

```
typedef struct MMDB_entry_data_list_s {
    MMDB_entry_data_s entry_data;
    struct MMDB_entry_data_list_s *next;
} MMDB_entry_data_list_s;
```

This structure lets you look at entire map or array data entry by iterating over the linked list.

`MMDB_search_node_s`

This structure encapsulates the two records in a search node. This is really only useful if you want to write code that iterates over the entire search tree as opposed to looking up a specific IP address.

```
typedef struct MMDB_search_node_s {
    uint64_t left_record;
    uint64_t right_record;
    uint8_t left_record_type;
    uint8_t right_record_type;
    MMDB_entry_s left_record_entry;
    MMDB_entry_s right_record_entry;
} MMDB_search_node_s;
```

The two record types will take one of the following values:

- `MMDB_RECORD_TYPE_SEARCH_NODE` – The record points to the next search node.
- `MMDB_RECORD_TYPE_EMPTY` – The record is a placeholder that indicates there is no data for the IP address. The search should end here.
- `MMDB_RECORD_TYPE_DATA` – The record is for data in the data section of the database. Use the entry for the record when looking up the data for the record.
- `MMDB_RECORD_TYPE_INVALID` – The record is invalid. Either an invalid node was looked up or the database is corrupt.

The `MMDB_entry_s` for the record is only valid if the type is `MMDB_RECORD_TYPE_DATA`. Attempts to use an entry for other record types will result in an error or invalid data.

STATUS CODES

This library returns (or populates) status codes for many functions. These status codes are:

- `MMDB_SUCCESS` – everything worked
- `MMDB_FILE_OPEN_ERROR` – there was an error trying to open the MaxMind DB file.
- `MMDB_IO_ERROR` – an IO operation failed. Check `errno` for more details.
- `MMDB_CORRUPT_SEARCH_TREE_ERROR` – looking up an IP address in the search tree gave us an impossible result. The database is damaged or was generated incorrectly or there is a bug in the `libmaxminddb` code.
- `MMDB_INVALID_METADATA_ERROR` – something in the database is wrong. This includes missing metadata keys as well as impossible values (like an `ip_version` of 7).
- `MMDB_UNKNOWN_DATABASE_FORMAT_ERROR` – The database metadata indicates that it's major version is not 2. This library can only handle major version 2.
- `MMDB_OUT_OF_MEMORY_ERROR` – a memory allocation call (`malloc`, etc.) failed.
- `MMDB_INVALID_DATA_ERROR` – an entry in the data section contains invalid data. For example, a `uint16` field is claiming to be more than 2 bytes long. The database is probably damaged or was generated incorrectly.
- `MMDB_INVALID_LOOKUP_PATH_ERROR` – The lookup path passed to `MMDB_get_value`, `MMDB_vget_value`, or `MMDB_aget_value` contains an array offset that is larger than `LONG_MAX` or smaller than `LONG_MIN`.
- `MMDB_LOOKUP_PATH_DOES_NOT_MATCH_DATA_ERROR` – The lookup path passed to `MMDB_get_value`, `MMDB_vget_value`, or `MMDB_aget_value` does not match the data structure for the entry. There are number of reasons this can happen. The lookup path could include a key not in a map. The lookup path could include an array index larger than an array or smaller than the minimum offset from the end of an array. It can also happen when the path expects to find a map or array where none exist.

All status codes should be treated as `int` values.

```
MMDB_strerror()
```

```
const char *MMDB_strerror(int error_code)
```

This function takes a status code and returns an English string explaining the status.

FUNCTIONS

This library provides the following exported functions:

```
MMDB_open()
```

```
int MMDB_open(
    const char *const filename,
    uint32_t flags,
    MMDB_s *const mmdb);
```

This function opens a handle to a MaxMind DB file. Its return value is a status code as defined above. Always check this call's return value.

```
MMDB_s mmdb;
int status =
    MMDB_open("/path/to/file.mmdb", MMDB_MODE_MMAP, &mmdb);
if (MMDB_SUCCESS != status) { ... }
...
MMDB_close(&mmdb);
```

`filename` must be encoded as UTF-8 on Windows.

The `MMDB_s` structure you pass in can be on the stack or allocated from the heap. However, if the open is successful it will contain heap-allocated data, so you need to close it with `MMDB_close()`. If the status returned is not `MMDB_SUCCESS` then this library makes sure that all allocated memory is freed before returning.

The flags currently provided are:

- `MMDB_MODE_MMAP` – open the database with `mmap()`.

Passing in other values for `flags` may yield unpredictable results. In the future we may add additional flags that you can bitwise-or together with the mode, as well as additional modes.

You can also pass 0 as the `flags` value in which case the database will be opened with the default flags. However, these defaults may change in future releases. The current default is `MMDB_MODE_MMAP`.

```
MMDB_close()
```

```
void MMDB_close(MMDB_s *const mmdb);
```

This frees any allocated or `mmap`'d memory that is held from the `MMDB_s` structure. *It does not free the memory allocated for the structure itself!* If you allocated the structure from the heap then you are responsible for freeing it.

```
MMDB_lookup_string()
```

```
MMDB_lookup_result_s MMDB_lookup_string(
    MMDB_s *const mmdb,
    const char *const ipstr,
    int *const gai_error,
    int *const mmdb_error);
```

This function looks up an IP address that is passed in as a null-terminated string. Internally it calls `getaddrinfo()` to resolve the address into a binary form. It then calls `MMDB_lookup_sockaddr()`

to look the address up in the database. If you have already resolved an address you can call `MMDB_lookup_sockaddr()` directly, rather than resolving the address twice.

```
int gai_error, mmdb_error;
MMDB_lookup_result_s result =
    MMDB_lookup_string(&mmdb, "1.2.3.4", &gai_error, &mmdb_error);
if (0 != gai_error) { ... }
if (MMDB_SUCCESS != mmdb_error) { ... }

if (result.found_entry) { ... }
```

This function always returns an `MMDB_lookup_result_s` structure, but you should also check the `gai_error` and `mmdb_error` parameters. If either of these indicates an error then the returned structure is meaningless.

If no error occurred you still need to make sure that the `found_entry` member in the returned result is true. If it's not, this means that the IP address does not have an entry in the database.

This function will work with IPv4 addresses even when the database contains data for both IPv4 and IPv6 addresses. The IPv4 address will be looked up as `::xxx.xxx.xxx.xxx` rather than being remapped to the `::ffff:xxx.xxx.xxx.xxx` block allocated for IPv4-mapped IPv6 addresses.

If you pass an IPv6 address to a database with only IPv4 data then the `found_entry` member will be false, but the `mmdb_error` status will still be `MMDB_SUCCESS`.

`MMDB_lookup_sockaddr()`

```
MMDB_lookup_result_s MMDB_lookup_sockaddr(
    MMDB_s *const mmdb,
    const struct sockaddr *const sockaddr,
    int *const mmdb_error);
```

This function looks up an IP address that has already been resolved by `getaddrinfo()`.

Other than not calling `getaddrinfo()` itself, this function is identical to the `MMDB_lookup_string()` function.

```
int mmdb_error;
MMDB_lookup_result_s result =
    MMDB_lookup_sockaddr(&mmdb, address->ai_addr, &mmdb_error);
if (MMDB_SUCCESS != mmdb_error) { ... }

if (result.found_entry) { ... }
```

Data Lookup Functions

There are three functions for looking up data associated with an IP address.

```
int MMDB_get_value(
    MMDB_entry_s *const start,
    MMDB_entry_data_s *const entry_data,
    ...);
int MMDB_vget_value(
    MMDB_entry_s *const start,
    MMDB_entry_data_s *const entry_data,
    va_list va_path);
int MMDB_aget_value(
    MMDB_entry_s *const start,
    MMDB_entry_data_s *const entry_data,
```



```
const char *const *const path);
```

The three functions allow three slightly different calling styles, but they all do the same thing.

The first parameter is an `MMDB_entry_s` value. In most cases this will come from the `MMDB_lookup_result_s` value returned by `MMDB_lookup_string()` or `MMDB_lookup_sockaddr()`.

The second parameter is a reference to an `MMDB_entry_data_s` structure. This will be populated with the data that is being looked up, if any is found. If nothing is found, then the `has_data` member of this structure will be false. If `has_data` is true then you can look at the `data_type` member.

The final parameter is a lookup path. The path consists of a set of strings representing either map keys (e.g. "city") or array indexes (e.g., "0", "1", "-1") to use in the lookup.

Negative array indexes will be treated as an offset from the end of the array. For instance, "-1" refers to the last element of the array.

The lookup path allows you to navigate a complex data structure. For example, given this data:

```
{
  "names": {
    "en": "Germany",
    "de": "Deutschland"
  },
  "cities": [ "Berlin", "Frankfurt" ]
}
```

We could look up the English name with this code:

```
MMDB_lookup_result_s result =
  MMDB_lookup_sockaddr(&mmdb, address->ai_addr, &mmdb_error);
MMDB_entry_data_s entry_data;
int status =
  MMDB_get_value(&result.entry, &entry_data,
                "names", "en", NULL);
if (MMDB_SUCCESS != status) { ... }
if (entry_data.has_data) { ... }
```

If we wanted to find the first city the lookup path would be "cities", "0". If you don't provide a lookup path at all, you'll get the entry which corresponds to the top level map. The lookup path must always end with `NULL`, regardless of which function you call.

The `MMDB_get_value` function takes a variable number of arguments. All of the arguments after the `MMDB_entry_data_s *` structure pointer are the lookup path. The last argument must be `NULL`.

The `MMDB_vget_value` function accepts a `va_list` as the lookup path. The last element retrieved by `va_arg()` must be `NULL`.

Finally, the `MMDB_aget_value` accepts an array of strings as the lookup path. The last member of this array must be `NULL`.

If you want to get all of the entry data at once you can call `MMDB_get_entry_data_list()` instead.

For each of the three functions, the return value is a status code as defined above.

```
MMDB_get_entry_data_list()
```

```
int MMDB_get_entry_data_list(
  MMDB_entry_s *start,
  MMDB_entry_data_list_s **const entry_data_list);
```

This function allows you to get all of the data for a complex data structure at once, rather than looking up

each piece using repeated calls to `MMDB_get_value()`.

```
MMDB_lookup_result_s result =
    MMDB_lookup_sockaddr(&mmdb, address->ai_addr, &mmdb_error);
MMDB_entry_data_list_s *entry_data_list, *first;
int status =
    MMDB_get_entry_data_list(&result.entry, &entry_data_list);
if (MMDB_SUCCESS != status) { ... }
// save this so we can free this data later
first = entry_data_list;

while (1) {
    MMDB_entry_data_list_s *next = entry_data_list = entry_data_list->next;
    if (NULL == next) {
        break;
    }

    switch (next->entry_data.type) {
        case MMDB_DATA_TYPE_MAP: { ... }
        case MMDB_DATA_TYPE_UTF8_STRING: { ... }
        ...
    }

}

MMDB_free_entry_data_list(first);
```

It's up to you to interpret the `entry_data_list` data structure. The list is linked in a depth-first traversal. Let's use this structure as an example:

```
{
    "names": {
        "en": "Germany",
        "de": "Deutschland"
    },
    "cities": [ "Berlin", "Frankfurt" ]
}
```

The list will consist of the following items:

1. MAP – top level map
2. UTF8_STRING – "names" key
3. MAP – map for "names" key
4. UTF8_STRING – "en" key
5. UTF8_STRING – value for "en" key
6. UTF8_STRING – "de" key
7. UTF8_STRING – value for "de" key
8. UTF8_STRING – "cities" key
9. ARRAY – value for "cities" key
10. UTF8_STRING – array[0]

11. UTF8_STRING – array[1]

The return value of the function is a status code as defined above.

MMDB_free_entry_data_list()

```
void MMDB_free_entry_data_list(
    MMDB_entry_data_list_s *const entry_data_list);
```

The MMDB_get_entry_data_list() and MMDB_get_metadata_as_entry_data_list() functions will allocate the linked list structure from the heap. Call this function to free the MMDB_entry_data_list_s structure.

MMDB_get_metadata_as_entry_data_list()

```
int MMDB_get_metadata_as_entry_data_list(
    MMDB_s *const mmdb,
    MMDB_entry_data_list_s **const entry_data_list);
```

This function allows you to retrieve the database metadata as a linked list of MMDB_entry_data_list_s structures. This can be a more convenient way to deal with the metadata than using the metadata structure directly.

```
MMDB_entry_data_list_s *entry_data_list, *first;
int status =
    MMDB_get_metadata_as_entry_data_list(&mmdb, &entry_data_list);
if (MMDB_SUCCESS != status) { ... }
first = entry_data_list;
... // do something with the data
MMDB_free_entry_data_list(first);
```

The return value of the function is a status code as defined above.

MMDB_dump_entry_data_list()

```
int MMDB_dump_entry_data_list(
    FILE *const stream,
    MMDB_entry_data_list_s *const entry_data_list,
    int indent);
```

This function takes a linked list of MMDB_entry_data_list_s structures and stringifies it to the given stream. The indent parameter is the starting indent level for the generated output. It is incremented for nested data structures (maps, array, etc.).

The stream must be a file handle (stdout, etc). If your platform provides something like the GNU open_memstream() you can use that to capture the output as a string.

The output is formatted in a JSON-ish fashion, but values are marked with their data type (except for maps and arrays which are shown with "{}" and "[]" respectively).

The specific output format may change in future releases, so you should not rely on the specific formatting produced by this function. It is intended to be used to show data to users in a readable way and for debugging purposes.

The return value of the function is a status code as defined above.

MMDB_read_node()

```
int MMDB_read_node(
    MMDB_s *const mmdb,
    uint32_t node_number,
    MMDB_search_node_s *const node);
```

This reads a specific node in the search tree. The third argument is a reference to an `MMDB_search_node_s` structure that will be populated by this function.

The return value is a status code. If you pass a `node_number` that is greater than the number of nodes in the database, this function will return `MMDB_INVALID_NODE_NUMBER_ERROR`, otherwise it will return `MMDB_SUCCESS`.

The first node in the search tree is always node 0. If you wanted to iterate over the whole search tree, you would start by reading node 0 and then following the the records that make up this node, based on the type of each record. If the type is `MMDB_RECORD_TYPE_SEARCH_NODE` then the record contains an integer for the next node to look up.

```
MMDB_lib_version()
```

```
const char *MMDB_lib_version(void)
```

This function returns the library version as a string, something like "2.0.0".

EXAMPLE

```
#include <errno.h>
#include <maxminddb.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv)
{
    char *filename = argv[1];
    char *ip_address = argv[2];

    MMDB_s mmdb;
    int status = MMDB_open(filename, MMDB_MODE_MMAP, &mmdb);

    if (MMDB_SUCCESS != status) {
        fprintf(stderr, "\n Can't open %s - %s\n",
                filename, MMDB_strerror(status));

        if (MMDB_IO_ERROR == status) {
            fprintf(stderr, " IO error: %s\n", strerror(errno));
        }
        exit(1);
    }

    int gai_error, mmdb_error;
    MMDB_lookup_result_s result =
        MMDB_lookup_string(&mmdb, ip_address, &gai_error, &mmdb_error);

    if (0 != gai_error) {
        fprintf(stderr,
                "\n Error from getaddrinfo for %s - %s\n\n",
                ip_address, gai_strerror(gai_error));
        exit(2);
    }

    if (MMDB_SUCCESS != mmdb_error) {
        fprintf(stderr,
                "\n Got an error from libmaxminddb: %s\n\n",
```

```

        MMDB_strerror(mmdb_error));
    exit(3);
}

MMDB_entry_data_list_s *entry_data_list = NULL;

int exit_code = 0;
if (result.found_entry) {
    int status = MMDB_get_entry_data_list(&result.entry,
                                         &entry_data_list);

    if (MMDB_SUCCESS != status) {
        fprintf(
            stderr,
            "Got an error looking up the entry data - %s\n",
            MMDB_strerror(status));
        exit_code = 4;
        goto end;
    }

    if (NULL != entry_data_list) {
        MMDB_dump_entry_data_list(stdout, entry_data_list, 2);
    }
} else {
    fprintf(
        stderr,
        "\n No entry for this IP address (%s) was found\n\n",
        ip_address);
    exit_code = 5;
}

end:
MMDB_free_entry_data_list(entry_data_list);
MMDB_close(&mmdb);
exit(exit_code);
}

```

THREAD SAFETY

This library is thread safe when compiled and linked with a thread-safe malloc and free implementation.

INSTALLATION AND SOURCE

You can download the latest release of libmaxminddb from GitHub (<https://github.com/maxmind/libmaxminddb/releases>).

Our GitHub repo (<https://github.com/maxmind/libmaxminddb>) is publicly available. Please fork it!

BUG REPORTS AND PULL REQUESTS

Please report all issues to our GitHub issue tracker (<https://github.com/maxmind/libmaxminddb/issues>). We welcome bug reports and pull requests. Please note that pull requests are greatly preferred over patches.

AUTHORS

This library was written by Boris Zentner (bzentner@maxmind.com) and Dave Rolsky (drolsky@maxmind.com).

COPYRIGHT AND LICENSE

Copyright 2013–2014 MaxMind, Inc.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

SEE ALSO

mmdblookup(1)