hipack-c Documentation

Release 0.1.2

Adrian Perez de Castro

December 27, 2015

Contents

Contents:

CHAPTER 1

Quickstart

This guide will walk you through the usage of the `hipack-c` C library.

Note: All the examples in this guide need a compiler which supports C99.

1.1 Reading (deserialization)

Let's start by parsing some data from the file which contains a HiPack message like the following:

```
title: "Quickstart"
is-documentation? True
```

First of all, we need to include the `hipack.h` header in the source of our program, and then use a *`hipack_read()`* to parse it. In this example we use the *`hipack_stdio_getchar()`* function included in the library which is used to read data from a `FILE*`:

```c
#include <hipack.h>
#include <stdio.h>   /* Needed for FILE* streams */

static void handle_message (hipack_dict_t *message);

int main (int argc, const char *argv[]) {
    hipack_reader_t reader = {
        .getchar_data = fopen ("input.hipack", "rb"),
        .getchar = hipack_stdio_getchar,
    };
    hipack_dict_t *message = hipack_read (&reader);
    fclose (reader.getchar_data);

    handle_message (message); /* Use "message". */

    hipack_dict_free (message);
    return 0;
}
```

Once the message was parsed, a dictionary (*`hipack_dict_t`*) is returned. The *Dictionary Functions* can be used to inspect the message. For example, adding the following as the `handle_message()` function prints the value of the `title` element:

```c
void handle_message (hipack_dict_t *message) {
    hipack_string_t *key = hipack_string_new_from_string ("title");
```

```
    hipack_value_t *value = hipack_dict_get (message, key);

    printf ("Title is '%s'\n", hipack_value_get_string (value));

    hipack_string_free (key); // Free memory used by "key".
}
```

Note how objects cretated by us, like the `key` string, have to be freed by us. In general, the user of the library is responsible for freeing any objects created by them. On the other hand, objects allocated by the library are freed by library functions.

In our example, the memory area which contains the `value` is owned by the message (more precisely: by the dictionary that represents the message), and the call to `hipack_value_get()` returns a pointer to the memory area owned by the message. The same happens with the call to *`hipack_value_get_string()`*: it returns a pointer to a memory area containing the bytes of the string value, which is owned by the *`hipack_value_t`* structure. This means that we *must not* free the value structure or the string, because when the whole message is freed —by calling *`hipack_dict_free()`* at the end of our `main()` function— the memory areas used by the value structure and the string will be freed as well.

1.2 Values

Objects of *`hipack_value_t`* represent a single value of those supported by HiPack: an integer number, a floating point number, a boolean, a list, or a dictionary. Creating an object if a "basic" value, that is all except lists and dictionary, can be done using the C99 designated initializer syntax. For example, to create a floating point number value:

```
hipack_value_t flt_val = {
    .type = HIPACK_FLOAT,
    .v_float = 4.5e-1
};
```

Alternatively, it is also possible to use provided utility functions to create values. The example above is equivalent to:

```
hipack_value_t flt_val = hipack_float (4.5e-1);
```

When using the C99 initializer syntax directly, the name of the field containing the value depends on the type. The following table summarizes the equivalence between types, the field to use in *`hipack_value_t`*, and the utility function for constructing values:

Type	Field	Macro
`HIPACK_INTEGER`	`.v_integer`	*`hipack_integer()`*
`HIPACK_FLOAT`	`.v_float`	*`hipack_float()`*
`HIPACK_BOOL`	`.v_bool`	*`hipack_bool()`*
`HIPACK_STRING`	`.v_string`	*`hipack_string()`*
`HIPACK_LIST`	`.v_list`	*`hipack_list()`*
`HIPACK_DICT`	`.v_dict`	*`hipack_dict()`*

Note that strings, lists, and dictionaries may have additional memory allocated. When wrapping them into a *`hipack_value_t`* only the pointer is stored, and it is still your responsibility to make sure the memory is freed when the values are not used anymore:

```
hipack_string_t *str = hipack_string_new_from_string ("spam");
hipack_value_t str_val = hipack_string (str);
assert (str == hipack_value_get_string (&str_val)); // Always true
```

For convenience, a *hipack_value_free()* function which will ensure the memory allocated by values referenced by a *hipack_value_t* will be freed properly. This way, one can write code in which the the value objects (*hipack_value_t*) are considered to be the owners of the memory which has been dynamically allocated:

```
// Ownership of the hipack_string_t is passed to "str_val"
hipack_value_t str_val =
    hipack_string (hipack_string_new_from_string ("spam"));
// ...
hipack_value_free (&str_val);  // Free memory.
```

This behaviour is particularly handy when assembling complex values: all items contained in lists and dictionaries can be just added to them, and when freeing the container, all the values in them will be freed as well. Consider the following function which creates a HiPack message with a few fields:

```
/*
 * Creates a message which would serialize as:
 *
 *    street: "Infinite Loop"
 *    lat: 37.332
 *    lon: -122.03
 *    number: 1
 */
hipack_dict_t* make_address_message (void) {
    hipack_dict_t *message = hipack_dict_new ();
    hipack_dict_set_adopt_key (message,
        hipack_string_new_from_string ("lat"),
        hipack_float (37.332));
    hipack_dict_set_adopt_key (message,
        hipack_string_new_from_string ("lon"),
        hipack_float (-122.03));
    hipack_dict_set_adopt_key (message,
        hipack_string_new_from_string ("street"),
        hipack_string (
            hipack_string_new_from_string ("Infinite Loop")));
    hipack_dict_set_adopt_key (message,
        hipack_string_new_from_string ("number"),
        hipack_integer (1));
    return message;
}
```

Note how it is not needed to free any of the values because they are "owned" by the dictionary, which gets returned from the function. Also, we use *hipack_dict_set_adopt_key()* (instead of *hipack_dict_set()*) to pass ownership of the keys to the dictionary as well and, at the same time, avoiding creating copies of the strings. The caller of this function would be the owner of the memory allocated by the returned dictionary and all its contained values.

1.3 Writing (serialization)

In order to serialize messages, we need a dictionary containing the values which we want to have serialized — remember that any dictionary can be considered a HiPack message. In order to serialize a message, we use *hipack_write()*, and in particular using *hipack_stdio_putchar()* we can directly write to a `FILE*` stream. Given the `make_address_message()` function from the previous section:

```
int main (int argc, char *argv[]) {
    hipack_dict_t *message = make_address_message ();
    hipack_writer_t writer = {
        .putchar_data = fopen ("address.hipack", "wb"),
```

```
        .putchar = hipack_stdio_putchar,
    };
    hipack_write (&writer, message);
    hipack_dict_free (message);
    return 0;
}
```

The resulting `address.hipack` file will have the following contents (the order of the elements may vary):

```
number: 1
street: "Infinite Loop"
lat: 37.332
lon: -122.03
```

CHAPTER 2

API Reference

2.1 Types

hipack_type_t
Type of a value. This enumeration takes one of the following values:

- `HIPACK_INTEGER`: Integer value.
- `HIPACK_FLOAT`: Floating point value.
- `HIPACK_BOOL`: Boolean value.
- `HIPACK_STRING`: String value.
- `HIPACK_LIST`: List value.
- `HIPACK_DICT`: Dictionary value.

hipack_value_t
Represent any valid HiPack value.

- `hipack_value_type()` obtains the type of a value.

hipack_string_t
String value.

hipack_list_t
List value.

hipack_dict_t
Dictionary value.

2.2 Memory Allocation

How `hipack-c` allocates memory can be customized by setting *`hipack_alloc`* to a custom allocation function.

hipack_alloc
Allocation function. By default it is set to `hipack_alloc_stdlib()`, which uses the implementations of `malloc()`, `realloc()`, and `free()` provided by the C library.

Allocation functions always have the following prototype:

```
void* func (void *oldptr, size_t size);
```

The behavior must be as follows:

- When invoked with `oldptr` set to `NULL`, and a non-zero `size`, the function behaves like `malloc()`: a memory block of at least `size` bytes is allocated and a pointer to it returned.
- When `oldptr` is non-`NULL`, and a non-zero `size`, the function behaves like `realloc()`: the memory area pointed to by `oldptr` is resized to be at least `size` bytes, or its contents moved to a new memory area of at least `size` bytes. The returned pointer may either be `oldptr`, or a pointer to the new memory area if the data was relocated.
- When `oldptr` is non-`NULL`, and `size` is zero, the function behaves like `free()`.

void* **hipack_alloc_stdlib** (void*, size_t)
Default allocation function. It uses `malloc()`, `realloc()`, and `free()` from the C library. By default *hipack_alloc* is set to use this function.

void* **hipack_alloc_array_extra** (void **oldptr*, size_t *nmemb*, size_t *size*, size_t *extra*)
Allocates (if *oldptr* is `NULL`) or reallocates (if *oldptr* is non-`NULL`) memory for an array which contains *nmemb* elements, each one of *size* bytes, plus an arbitrary amount of *extra* bytes.

This function is used internally by the HiPack parser, and it is not likely to be needed by client code.

void* **hipack_alloc_array** (void **optr*, size_t *nmemb*, size_t *size*)
Same as *hipack_alloc_array_extra()*, without allowing to specify the amount of extra bytes. The following calls are both equivalent:

```
void *a = hipack_alloc_array_extra (NULL, 10, 4, 0);
void *b = hipack_alloc_array (NULL, 10, 4);
```

See *hipack_alloc_array_extra()* for details.

void* **hipack_alloc_bzero** (size_t *size*)
Allocates an area of memory of *size* bytes, and initializes it to zeroes.

void **hipack_alloc_free** (void **pointer*)
Frees the memory area referenced by the given *pointer*.

2.3 String Functions

The following functions are provided as a convenience to operate on values of type *hipack_string_t*.

Note: The hash function used by *hipack_string_hash()* is *not* guaranteed to be cryptographically safe. Please do avoid exposing values returned by this function to the attack surface of your applications, in particular *do not expose them to the network*.

---

*hipack_string_t** **hipack_string_copy** (const *hipack_string_t* **hstr*)
Returns a new copy of a string.

The returned value must be freed using *hipack_string_free()*.

*hipack_string_t** **hipack_string_new_from_string** (const char **str*)
Creates a new string from a C-style zero terminated string.

The returned value must be freed using *hipack_string_free()*.

*hipack_string_t** **hipack_string_new_from_lstring** (const char **str*, uint32_t *len*)
Creates a new string from a memory area and its length.

The returned value must be freed using *hipack_string_free()*.

uint32_t **hipack_string_hash** (const *hipack_string_t* **hstr*)
Calculates a hash value for a string.

bool **hipack_string_equal** (const *hipack_string_t *hstr1*, const *hipack_string_t *hstr2*)
Compares two strings to check whether their contents are the same.

void **hipack_string_free** (*hipack_string_t *hstr*)
Frees the memory used by a string.

2.4 List Functions

*hipack_list_t** **hipack_list_new** (uint32_t *size*)
Creates a new list for `size` elements.

void **hipack_list_free** (*hipack_list_t *list*)
Frees the memory used by a list.

bool **hipack_list_equal** (const *hipack_list_t *a*, const *hipack_list_t *b*)
Checks whether two lists contains the same values.

uint32_t **hipack_list_size** (const *hipack_list_t *list*)
Obtains the number of elements in a list.

HIPACK_LIST_AT (list, index)
Obtains a pointer to the element at a given *index* of a *list*.

2.5 Dictionary Functions

uint32_t **hipack_dict_size** (const *hipack_dict_t *dict*)
Obtains the number of elements in a dictionary.

*hipack_dict_t** **hipack_dict_new** (void)
Creates a new, empty dictionary.

void **hipack_dict_free** (*hipack_dict_t *dict*)
Frees the memory used by a dictionary.

bool **hipack_dict_equal** (const *hipack_dict_t *a*, const *hipack_dict_t *b*)
Checks whether two dictinaries contain the same keys, and their associated values in each of the dictionaries are equal.

void **hipack_dict_set** (*hipack_dict_t *dict*, const *hipack_string_t *key*, const *hipack_value_t *value*)
Adds an association of a *key* to a *value*.

Note that this function will copy the *key*. If you are not planning to continue reusing the *key*, it is recommended to use `hipack_dict_set_adopt_key()` instead.

void **hipack_dict_set_adopt_key** (*hipack_dict_t *dict*, *hipack_string_t **key*, const *hipack_value_t *value*)
Adds an association of a *key* to a *value*, passing ownership of the memory using by the *key* to the dictionary (i.e. the string used as key will be freed by the dictionary).

Use this function instead of `hipack_dict_set()` when the *key* is not going to be used further afterwards.

void **hipack_dict_del** (*hipack_dict_t *dict*, const *hipack_string_t *key*)
Removes the element from a dictionary associated to a *key*.

*hipack_value_t** **hipack_dict_get** (const *hipack_dict_t *dict*, const *hipack_string_t *key*)
Obtains the value associated to a *key* from a dictionary.

The returned value points to memory owned by the dictionary. The value can be modified in-place, but it shall not be freed.

*hipack_value_t** **hipack_dict_first** (const *hipack_dict_t *dict*, const *hipack_string_t **key*)
Obtains an a *(key, value)* pair, which is considered the *first* in iteration order. This can be used in combination with *hipack_dict_next()* to enumerate all the *(key, value)* pairs stored in the dictionary:

```
hipack_dict_t *d = get_dictionary ();
hipack_value_t *v;
hipack_string_t *k;

for (v = hipack_dict_first (d, &k);
     v != NULL;
     v = hipack_dict_next (v, &k)) {
    // Use "k" and "v" normally.
}
```

As a shorthand, consider using `HIPACK_DICT_FOREACH()` instead.

*hipack_value_t** **hipack_dict_next** (*hipack_value_t *value*, const *hipack_string_t **key*)
Iterates to the next *(key, value)* pair of a dictionary. For usage details, see *hipack_dict_first()*.

HIPACK_DICT_FOREACH (dict, key, value)
Convenience macro used to iterate over the *(key, value)* pairs contained in a dictionary. Internally this uses *hipack_dict_first()* and *hipack_dict_next()*.

```
hipack_dict_t *d = get_dictionary ();
hipack_string_t *k;
hipack_value_t *v;
HIPACK_DICT_FOREACH (d, k, v) {
    // Use "k" and "v"
}
```

Using this macro is the recommended way of writing a loop to enumerate elements from a dictionary.

2.6 Value Functions

hipack_type_t **hipack_value_type** (const *hipack_value_t *value*)
Obtains the type of a value.

hipack_value_t **hipack_integer** (int32_t *value*)
Creates a new integer value.

hipack_value_t **hipack_float** (double *value*)
Creates a new floating point value.

hipack_value_t **hipack_bool** (bool *value*)
Creates a new boolean value.

hipack_value_t **hipack_string** (*hipack_string_t *value*)
Creates a new string value.

hipack_value_t **hipack_list** (*hipack_list_t *value*)
Creates a new list value.

hipack_value_t **hipack_dict** (*hipack_dict_t *value*)
Creates a new dictionary value.

bool **hipack_value_is_integer** (const *hipack_value_t *value*)
Checks whether a value is an integer.

bool **hipack_value_is_float** (const *hipack_value_t* *value)
Checks whether a value is a floating point number.

bool **hipack_value_is_bool** (const *hipack_value_t* *value)
Checks whether a value is a boolean.

bool **hipack_value_is_string** (const *hipack_value_t* *value)
Checks whether a value is a string.

bool **hipack_value_is_list** (const *hipack_value_t* *value)
Checks whether a value is a list.

bool **hipack_value_is_dict** (const *hipack_value_t* *value)
Checks whether a value is a dictionary.

const int32_t **hipack_value_get_integer** (const *hipack_value_t* *value)
Obtains a numeric value as an `int32_t`.

const double **hipack_value_get_float** (const *hipack_value_t* *value)
Obtains a floating point value as a `double`.

const bool **hipack_value_get_bool** (const *hipack_value_t* *value)
Obtains a boolean value as a `bool`.

const *hipack_string_t** **hipack_value_get_string** (const *hipack_value_t* *value)
Obtains a numeric value as a *hipack_string_t*.

const *hipack_list_t** **hipack_value_get_list** (const *hipack_value_t* *value)
Obtains a numeric value as a *hipack_list_t*.

const *hipack_dict_t** **hipack_value_get_dict** (const *hipack_value_t* *value)
Obtains a numeric value as a *hipack_dict_t*.

bool **hipack_value_equal** (const *hipack_value_t* *a, const *hipack_value_t* *b)
Checks whether two values are equal.

void **hipack_value_free** (*hipack_value_t* *value)
Frees the memory used by a value.

void **hipack_value_add_annot** (*hipack_value_t* *value, const char *annot)
Adds an annotation to a value. If the value already had the annotation, this function is a no-op.

bool **hipack_value_has_annot** (const *hipack_value_t* *value, const char *annot)
Checks whether a value has a given annotation.

void **hipack_value_del_annot** (*hipack_value_t* *value, const char *annot)
Removes an annotation from a value. If the annotation was not present, this function is a no-op.

2.7 Reader Interface

hipack_reader_t
Allows communicating with the parser, instructing it how to read text input data, and provides a way for the parser to report errors back.

The following members of the structure are to be used by client code:

int **(*getchar)** (void *data)
Reader callback function. The function will be called every time the next character of input is needed. It must return it as an integer, *HIPACK_IO_EOF* when trying to read pas the end of the input, or *HIPACK_IO_ERROR* if an input error occurs.

const char ***error**
On error, a string describing the issue, suitable to be displayed to the user.

unsigned **error_line**
On error, the line number where parsing was stopped.

unsigned **error_column**
On error, the column where parsing was stopped.

HIPACK_IO_EOF
Constant returned by reader functions when trying to read past the end of the input.

HIPACK_IO_ERROR
Constant returned by reader functions on input errors.

HIPACK_READ_ERROR
Constant value used to signal an underlying input error.

The *error* field of *hipack_reader_t* is set to this value when the reader function returns *HIPACK_IO_ERROR*. This is provided to allow client code to detect this condition and further query for the nature of the input error.

*hipack_dict_t** **hipack_read**(*hipack_reader_t *reader*)
Reads a HiPack message from a stream *reader* and returns a dictionary.

On error, `NULL` is returned, and the members *error*, *error_line*, and *error_column* (see *hipack_reader_t*) are set accordingly in the *reader*.

int **hipack_stdio_getchar**(void* *fp*)
Reader function which uses `FILE*` objects from the standard C library.

To use this function to read from a `FILE*`, first open a file, and then create a reader using this function and the open file as data to be passed to it, and then use *hipack_read()*:

```
FILE* stream = fopen (HIPACK_FILE_PATH, "rb")
hipack_reader_t reader = {
    .getchar = hipack_stdio_getchar,
    .getchar_data = stream,
};
hipack_dict_t *message = hipack_read (&reader);
```

The user is responsible for closing the `FILE*` after using it.

2.8 Writer Interface

hipack_writer_t
Allows specifying how to write text output data, and configuring how the produced HiPack output looks like.

The following members of the structure are to be used by client code:

int **(*putchar)**(void *data, int *ch*)
Writer callback function. The function will be called every time a character is produced as output. It must return *HIPACK_IO_ERROR* if an output error occurs, and it is invalid for the callback to return *HIPACK_IO_EOF*. Any other value is interpreted as indication of success.

void* **putchar_data**
Data passed to the writer callback function.

int32_t **indent**
Either *HIPACK_WRITER_COMPACT* or *HIPACK_WRITER_INDENTED*.

HIPACK_WRITER_COMPACT
Flag to generate output HiPack messages in their compact representation.

HIPACK_WRITER_INDENTED
Flag to generate output HiPack messages in "indented" (pretty-printed) representation.

bool **hipack_write** (*hipack_writer_t *writer*, const *hipack_dict_t *message*)
Writes a HiPack *message* to a stream *writer*, and returns whether writing the message was successful.

int **hipack_stdio_putchar** (void* *data*, int *ch*)
Writer function which uses `FILE*` objects from the standard C library.

To use this function to write a message to a `FILE*`, first open a file, then create a writer using this function, and then use *hipack_write()*:

```
FILE* stream = fopen (HIPACK_FILE_PATH, "wb");
hipack_writer_t writer = {
    .putchar = hipack_stdio_putchar,
    .putchar_data = stream,
};
hipack_write (&writer, message);
```

The user is responsible for closing the `FILE*` after using it.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

H