NAME

LHASH, LHASH_OF, DEFINE_LHASH_OF_EX, DEFINE_LHASH_OF, OPENSSL_LH_COMPFUNC, OPENSSL_LH_HASHFUNC, OPENSSL_LH_DOALL_FUNC, LHASH_DOALL_ARG_FN_TYPE, IMPLEMENT_LHASH_HASH_FN, IMPLEMENT_LHASH_COMP_FN, lh_TYPE_new, lh_TYPE_free, lh_TYPE_flush, lh_TYPE_insert, lh_TYPE_delete, lh_TYPE_retrieve, lh_TYPE_doall, lh_TYPE_doall_arg, lh_TYPE_num_items, lh_TYPE_get_down_load, lh_TYPE_set_down_load, lh_TYPE_error, OPENSSL_LH_new, OPENSSL_LH_free, OPENSSL_LH_flush, OPENSSL_LH_insert, OPENSSL_LH_delete, OPENSSL_LH_retrieve, OPENSSL_LH_doall, OPENSSL_LH_doall_arg, OPENSSL_LH_doall_arg_thunk, OPENSSL_LH_set_thunks, OPENSSL_LH_num_items, OPENSSL_LH_get_down_load, OPENSSL_LH_set_down_load, OPENSSL_LH_error − dynamic hash table

SYNOPSIS

```
#include <openssl/lhash.h>

LHASH_OF(TYPE)

DEFINE_LHASH_OF_EX(TYPE);

LHASH_OF(TYPE) *lh_TYPE_new(OPENSSL_LH_HASHFUNC hash, OPENSSL_LH_COMPFUNC compare);
void lh_TYPE_free(LHASH_OF(TYPE) *table);
void lh_TYPE_flush(LHASH_OF(TYPE) *table);
OPENSSL_LHASH *OPENSSL_LH_set_thunks(OPENSSL_LHASH *lh,
                                     OPENSSL_LH_HASHFUNCTHUNK hw,
                                     OPENSSL_LH_COMPFUNCTHUNK cw,
                                     OPENSSL_LH_DOALL_FUNC_THUNK daw,
                                     OPENSSL_LH_DOALL_FUNCARG_THUNK daaw)

TYPE *lh_TYPE_insert(LHASH_OF(TYPE) *table, TYPE *data);
TYPE *lh_TYPE_delete(LHASH_OF(TYPE) *table, TYPE *data);
TYPE *lh_TYPE_retrieve(LHASH_OF(TYPE) *table, TYPE *data);

void lh_TYPE_doall(LHASH_OF(TYPE) *table, OPENSSL_LH_DOALL_FUNC func);
void lh_TYPE_doall_arg(LHASH_OF(TYPE) *table, OPENSSL_LH_DOALL_FUNCARG func,
                       TYPE *arg);
void OPENSSL_LH_doall_arg_thunk(OPENSSL_LHASH *lh,
                                OPENSSL_LH_DOALL_FUNCARG_THUNK daaw,
                                OPENSSL_LH_DOALL_FUNCARG fn, void *arg)

unsigned long lh_TYPE_num_items(OPENSSL_LHASH *lh);
unsigned long lh_TYPE_get_down_load(OPENSSL_LHASH *lh);
void lh_TYPE_set_down_load(OPENSSL_LHASH *lh, unsigned long dl);

int lh_TYPE_error(LHASH_OF(TYPE) *table);

typedef int (*OPENSSL_LH_COMPFUNC)(const void *, const void *);
typedef unsigned long (*OPENSSL_LH_HASHFUNC)(const void *);
typedef void (*OPENSSL_LH_DOALL_FUNC)(const void *);
typedef void (*LHASH_DOALL_ARG_FN_TYPE)(const void *, const void *);

OPENSSL_LHASH *OPENSSL_LH_new(OPENSSL_LH_HASHFUNC h, OPENSSL_LH_COMPFUNC c);
void OPENSSL_LH_free(OPENSSL_LHASH *lh);
void OPENSSL_LH_flush(OPENSSL_LHASH *lh);

void *OPENSSL_LH_insert(OPENSSL_LHASH *lh, void *data);
```

```
void *OPENSSL_LH_delete(OPENSSL_LHASH *lh, const void *data);
void *OPENSSL_LH_retrieve(OPENSSL_LHASH *lh, const void *data);

void OPENSSL_LH_doall(OPENSSL_LHASH *lh, OPENSSL_LH_DOALL_FUNC func);
void OPENSSL_LH_doall_arg(OPENSSL_LHASH *lh, OPENSSL_LH_DOALL_FUNCARG func, void *a

unsigned long OPENSSL_LH_num_items(OPENSSL_LHASH *lh);
unsigned long OPENSSL_LH_get_down_load(OPENSSL_LHASH *lh);
void OPENSSL_LH_set_down_load(OPENSSL_LHASH *lh, unsigned long dl);

int OPENSSL_LH_error(OPENSSL_LHASH *lh);

#define LH_LOAD_MULT   /* integer constant */
```

The following macro is deprecated:

```
DEFINE_LHASH_OF(TYPE);
```

DESCRIPTION

This library implements type−checked dynamic hash tables. The hash table entries can be arbitrary structures. Usually they consist of key and value fields. In the description here, ***TYPE*** is used a placeholder for any of the OpenSSL datatypes, such as *SSL_SESSION*.

To define a new type−checked dynamic hash table, use **DEFINE_LHASH_OF_EX**(). **DEFINE_LHASH_OF**() was previously used for this purpose, but is now deprecated. The **DEFINE_LHASH_OF_EX**() macro provides all functionality of **DEFINE_LHASH_OF**() except for certain deprecated statistics functions (see **OPENSSL_LH_stats**(3)).

lh_*TYPE*_new() creates a new **LHASH_OF**(***TYPE***) structure to store arbitrary data entries, and specifies the 'hash' and 'compare' callbacks to be used in organising the table's entries. The *hash* callback takes a pointer to a table entry as its argument and returns an unsigned long hash value for its key field. The hash value is normally truncated to a power of 2, so make sure that your hash function returns well mixed low order bits. The *compare* callback takes two arguments (pointers to two hash table entries), and returns 0 if their keys are equal, nonzero otherwise.

If your hash table will contain items of some particular type and the *hash* and *compare* callbacks hash/compare these types, then the **IMPLEMENT_LHASH_HASH_FN** and **IMPLEMENT_LHASH_COMP_FN** macros can be used to create callback wrappers of the prototypes required by **lh_*TYPE*_new**() as shown in this example:

```
/*
 * Implement the hash and compare functions; "stuff" can be any word.
 */
static unsigned long stuff_hash(const TYPE *a)
{
    ...
}
static int stuff_cmp(const TYPE *a, const TYPE *b)
{
    ...
}

/*
 * Implement the wrapper functions.
 */
static IMPLEMENT_LHASH_HASH_FN(stuff, TYPE)
static IMPLEMENT_LHASH_COMP_FN(stuff, TYPE)
```

If the type is going to be used in several places, the following macros can be used in a common header file

to declare the function wrappers:

```
DECLARE_LHASH_HASH_FN(stuff, TYPE)
DECLARE_LHASH_COMP_FN(stuff, TYPE)
```

Then a hash table of ***TYPE*** objects can be created using this:

```
LHASH_OF(TYPE) *htable;

htable = B<lh_I<TYPE>_new>(LHASH_HASH_FN(stuff), LHASH_COMP_FN(stuff));
```

lh_*TYPE*_free() frees the **LHASH_OF**(***TYPE***) structure *table*. Allocated hash table entries will not be freed; consider using **lh_*TYPE*_doall**() to deallocate any remaining entries in the hash table (see below). If the argument is NULL, nothing is done.

lh_*TYPE*_flush() empties the **LHASH_OF**(***TYPE***) structure *table*. New entries can be added to the flushed table. Allocated hash table entries will not be freed; consider using**lh_*TYPE*_doall**() to deallocate an y remaining entries in the hash table (see below).

lh_*TYPE*_insert() inserts the structure pointed to by *data* into *table*. If there already is an entry with the same key, the old value is replaced. Note that **lh_*TYPE*_insert**() stores pointers, the data are not copied.

lh_*TYPE*_delete() deletes an entry from *table*.

lh_*TYPE*_retrieve() looks up an entry in *table*. Normally, *data* is a structure with the key field(s) set; the function will return a pointer to a fully populated structure.

lh_*TYPE*_doall() will, for every entry in the hash table, call *func* with the data item as its parameter. For example:

```
/* Cleans up resources belonging to 'a' (this is implemented elsewhere) */
void TYPE_cleanup_doall(TYPE *a);

/* Implement a prototype-compatible wrapper for "TYPE_cleanup" */
IMPLEMENT_LHASH_DOALL_FN(TYPE_cleanup, TYPE)

/* Call "TYPE_cleanup" against all items in a hash table. */
lh_TYPE_doall(hashtable, LHASH_DOALL_FN(TYPE_cleanup));

/* Then the hash table itself can be deallocated */
lh_TYPE_free(hashtable);
```

lh_*TYPE*_doall_arg() is the same as **lh_*TYPE*_doall**() except that *func* will be called with *arg* as the second argument and *func* should be of type **LHASH_DOALL_ARG_FN**(***TYPE***) (a callback prototype that is passed both the table entry and an extra argument). As with**lh_doall()**, you can instead choose to declare your callback with a prototype matching the types you are dealing with and use the declare/implement macros to create compatible wrappers that cast variables before calling your type-specific callbacks. An example of this is demonstrated here (printing all hash table entries to a BIO that is provided by the caller):

```
/* Prints item 'a' to 'output_bio' (this is implemented elsewhere) */
void TYPE_print_doall_arg(const TYPE *a, BIO *output_bio);

/* Implement a prototype-compatible wrapper for "TYPE_print" */
static IMPLEMENT_LHASH_DOALL_ARG_FN(TYPE, const TYPE, BIO)

/* Print out the entire hashtable to a particular BIO */
lh_TYPE_doall_arg(hashtable, LHASH_DOALL_ARG_FN(TYPE_print), BIO,
                  logging_bio);
```

Note that it is by default **not** safe to use **lh_*TYPE*_delete**() inside a callback passed to **lh_*TYPE*_doall**() or **lh_*TYPE*_doall_arg**(). The reason for this is that deleting an item from the hash table may result in the

hash table being contracted to a smaller size and rehashed. **lh_*TYPE*_doall**() and **lh_*TYPE*_doall_ar g**() are unsafe and will exhibit undefined behaviour under these conditions, as these functions assume the hash table size and bucket pointers do not change during the call.

If it is desired to use **lh_*TYPE*_doall**() or **lh_*TYPE*_doall_arg**() with **lh_*TYPE*_delete**(), it is essential that you call **lh_*TYPE*_set_down_load**() with a *down_load* argument of 0 first. This disables hash table contraction and guarantees that it will be safe to delete items from a hash table during a call to **lh_*TYPE*_doall**() or **lh_*TYPE*_doall_arg**().

It is never safe to call **lh_*TYPE*_insert**() during a call to **lh_*TYPE*_doall**() or **lh_*TYPE*_doall_arg**().

lh_*TYPE*_error() can be used to determine if an error occurred in the last operation.

lh_*TYPE*_num_items() returns the number of items in the hash table.

lh_*TYPE*_get_down_load() and **lh_*TYPE*_set_down_load**() get and set the factor used to determine when the hash table is contracted. The factor is the load factor at or below which hash table contraction will occur, multiplied by **LH_LOAD_MULT**, where the load factor is the number of items divided by the number of nodes. Setting this value to 0 disables hash table contraction.

OPENSSL_LH_new() is the same as the **lh_*TYPE*_new**() except that it is not type specific. So instead of returning an **LHASH_OF(*TYPE*)** value it returns a **void ***. In the same way the functions **OPENSSL_LH_free()**, **OPENSSL_LH_flush()**, **OPENSSL_LH_insert()**, **OPENSSL_LH_delete()**, **OPENSSL_LH_retrieve()**, **OPENSSL_LH_doall()**, **OPENSSL_LH_doall_arg()**, **OPENSSL_LH_num_items()**, **OPENSSL_LH_get_down_load()**, **OPENSSL_LH_set_down_load()** and **OPENSSL_LH_error()** are equivalent to the similarly named **lh_*TYPE*** functions except that they return or use a **void *** where the equivalent **lh_*TYPE*** function returns or uses a ***TYPE ** or **LHASH_OF(*TYPE*) ***. **lh_*TYPE*** functions are implemented as type checked wrappers around the **OPENSSL_LH** functions. Most applications should not call the **OPENSSL_LH** functions directly.

OPENSSL_LH_set_thunks() and **OPENSSL_LH_doall_arg_thunk()**, while public by necessity, are actually internal functions and should not be used.

RETURN VALUES

lh_*TYPE*_new() and **OPENSSL_LH_new()** return NULL on error, otherwise a pointer to the new **LHASH** structure.

When a hash table entry is replaced, **lh_*TYPE*_insert**() or **OPENSSL_LH_insert()** return the value being replaced. NULL is returned on normal operation and on error.

lh_*TYPE*_delete() and **OPENSSL_LH_delete()** return the entry being deleted. NULL is returned if there is no such value in the hash table.

lh_*TYPE*_retrieve() and **OPENSSL_LH_retrieve()** return the hash table entry if it has been found, NULL otherwise.

lh_*TYPE*_error() and **OPENSSL_LH_error()** return 1 if an error occurred in the last operation, 0 otherwise. It's meaningful only after non−retrieve operations.

lh_*TYPE*_free(), **OPENSSL_LH_free()**, **lh_*TYPE*_flush**(), **OPENSSL_LH_flush()**, **lh_*TYPE*_doall**() **OPENSSL_LH_doall()**, **lh_*TYPE*_doall_arg**() and **OPENSSL_LH_doall_arg()** return no values.

NOTE

The LHASH code is not thread safe. All updating operations, as well as **lh_*TYPE*_error**() or **OPENSSL_LH_error()** calls must be performed under a write lock. All retrieve operations should be performed under a read lock, *unless* accurate usage statistics are desired. In which case, a write lock should be used for retrieve operations as well. For output of the usage statistics, using the functions from **OPENSSL_LH_stats** (3), a read lock suffices.

The LHASH code regards table entries as constant data. As such, it internally represents **lh_insert()**'d items with a "const void *" pointer type. This is why callbacks such as those used by **lh_doall()** and **lh_doall_arg()** declare their prototypes with "const", even for the parameters that pass back the table items' data pointers − for consistency, user−provided data is "const" at all times as far as the LHASH code is

concerned. However, as callers are themselves providing these pointers, they can choose whether they too should be treating all such parameters as constant.

As an example, a hash table may be maintained by code that, for reasons of encapsulation, has only "const" access to the data being indexed in the hash table (i.e. it is returned as "const" from elsewhere in their code) − in this case the LHASH prototypes are appropriate as−is. Conversely, if the caller is responsible for the life−time of the data in question, then they may well wish to make modifications to table item passed back in the **lh_doall()** or **lh_doall_arg()** callbacks (see the "TYPE_cleanup" example above). If so, the caller can either cast the "const" away (if they're providing the raw callbacks themselves) or use the macros to declare/implement the wrapper functions without "const" types.

Callers that only have "const" access to data they're indexing in a table, yet declare callbacks without constant types (or cast the "const" away themselves), are therefore creating their own risks/bugs without being encouraged to do so by the API. On a related note, those auditing code should pay special attention to any instances of DECLARE/IMPLEMENT_LHASH_DOALL_[ARG_]_FN macros that provide types without any "const" qualifiers.

BUGS

lh_*TYPE*_insert() and **OPENSSL_LH_insert()** return NULL both for success and error.

SEE ALSO

OPENSSL_LH_stats (3)

HISTORY

In OpenSSL 1.0.0, the lhash interface was revamped for better type checking.

In OpenSSL 3.1, **DEFINE_LHASH_OF_EX**() was introduced and **DEFINE_LHASH_OF**() was deprecated.

OPENSSL_LH_doall_arg_thunk(), **OPENSSL_LH_set_thunks()** were added in OpenSSL 3.3.

COPYRIGHT