NAME

lh_new, lh_free, lh_insert, lh_delete, lh_retrieve, lh_doall, lh_doall_arg, lh_error - dynamic hash table

SYNOPSIS

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

DECLARE_LHASH_OF(<type>);

LHASH *lh_<type>_new();
void lh_<type>_free(LHASH_OF(<type> *table);

<type> *lh_<type>_insert(LHASH_OF(<type> *table, <type> *data);
<type> *lh_<type>_delete(LHASH_OF(<type> *table, <type> *data);
<type> *lh_retrieve(LHASH_OF<type> *table, <type> *data);

void lh_<type>_doall(LHASH_OF(<type> *table, LHASH_DOALL_FN_TYPE func);
void lh_<type>_doall_arg(LHASH_OF(<type> *table, LHASH_DOALL_ARG_FN_TYPE func,
<type2>, <type2> *arg);

int lh_<type>_error(LHASH_OF(<type> *table);

typedef int (*LHASH_COMP_FN_TYPE)(const void *, const void *);
typedef unsigned long (*LHASH_HASH_FN_TYPE)(const void *);
typedef void (*LHASH_DOALL_FN_TYPE)(const void *);
typedef void (*LHASH_DOALL_ARG_FN_TYPE)(const void *, const void *);
```

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.

lh_<type>*_new()* creates a new **LHASH_OF(<type>** structure to store arbitrary data entries, and provides 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, non-zero otherwise. If your hash table will contain items of some particular type and the **hash** and **compare** callbacks hash/compare these types, then the **DECLARE_LHASH_HASH_FN** and **IMPLEMENT_LHASH_COMP_FN** macros can be used to create callback wrappers of the prototypes required by lh_<type>*_new()*. These provide per-variable casts before calling the type-specific callbacks written by the application author. These macros, as well as those used for the ''doall'' callbacks, are defined as;

```
#define DECLARE_LHASH_HASH_FN(name, o_type) \
unsigned long name##_LHASH_HASH(const void *);
#define IMPLEMENT_LHASH_HASH_FN(name, o_type) \
unsigned long name##_LHASH_HASH(const void *arg) { \
const o_type *a = arg; \
return name##_hash(a); }
#define LHASH_HASH_FN(name) name##_LHASH_HASH

#define DECLARE_LHASH_COMP_FN(name, o_type) \
int name##_LHASH_COMP(const void *, const void *);
#define IMPLEMENT_LHASH_COMP_FN(name, o_type) \
int name##_LHASH_COMP(const void *arg1, const void *arg2) { \
const o_type *a = arg1; \
```

```
const o_type *b = arg2; \
return name##_cmp(a,b); }
#define LHASH_COMP_FN(name) name##_LHASH_COMP

#define DECLARE_LHASH_DOALL_FN(name, o_type) \
void name##_LHASH_DOALL(void *);
#define IMPLEMENT_LHASH_DOALL_FN(name, o_type) \
void name##_LHASH_DOALL(void *arg) { \
o_type *a = arg; \
name##_doall(a); }
#define LHASH_DOALL_FN(name) name##_LHASH_DOALL

#define DECLARE_LHASH_DOALL_ARG_FN(name, o_type, a_type) \
void name##_LHASH_DOALL_ARG(void *, void *);
#define IMPLEMENT_LHASH_DOALL_ARG_FN(name, o_type, a_type) \
void name##_LHASH_DOALL_ARG(void *arg1, void *arg2) { \
o_type *a = arg1; \
a_type *b = arg2; \
name##_doall_arg(a, b); }
#define LHASH_DOALL_ARG_FN(name) name##_LHASH_DOALL_ARG

An example of a hash table storing (pointers to) structures of type 'STUFF'
could be defined as follows;

/* Calculates the hash value of 'tohash' (implemented elsewhere) */
unsigned long STUFF_hash(const STUFF *tohash);
/* Orders 'arg1' and 'arg2' (implemented elsewhere) */
int stuff_cmp(const STUFF *arg1, const STUFF *arg2);
/* Create the type-safe wrapper functions for use in the LHASH internals */
static IMPLEMENT_LHASH_HASH_FN(stuff, STUFF);
static IMPLEMENT_LHASH_COMP_FN(stuff, STUFF);
/* ... */
int main(int argc, char *argv[]) {
/* Create the new hash table using the hash/compare wrappers */
LHASH_OF(STUFF) *hashtable = lh_STUFF_new(LHASH_HASH_FN(STUFF_hash),
LHASH_COMP_FN(STUFF_cmp));
/* ... */
}
```

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).

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 lh_<type>_*doall()* and lh_<type>_*doall_arg()*, function pointer casting should be avoided in the callbacks (see **NOTE**) - instead use the declare/implement macros to create type-checked wrappers that cast variables prior to calling your type-specific callbacks. An example of this is illustrated here where the callback is used to cleanup resources for items in the hash table

prior to the hashtable itself being deallocated:

```
/* Cleans up resources belonging to 'a' (this is implemented elsewhere) */
void STUFF_cleanup_doall(STUFF *a);
/* Implement a prototype-compatible wrapper for "STUFF_cleanup" */
IMPLEMENT_LHASH_DOALL_FN(STUFF_cleanup, STUFF)
/* ... then later in the code ... */
/* So to run "STUFF_cleanup" against all items in a hash table ... */
lh_STUFF_doall(hashtable, LHASH_DOALL_FN(STUFF_cleanup));
/* Then the hash table itself can be deallocated */
lh_STUFF_free(hashtable);
```

When doing this, be careful if you delete entries from the hash table in your callbacks: the table may decrease in size, moving the item that you are currently on down lower in the hash table - this could cause some entries to be skipped during the iteration. The second best solution to this problem is to set hash->down_load=0 before you start (which will stop the hash table ever decreasing in size). The best solution is probably to avoid deleting items from the hash table inside a ''doall'' callback!

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 STUFF_print_doall_arg(const STUFF *a, BIO *output_bio);
/* Implement a prototype-compatible wrapper for "STUFF_print" */
static IMPLEMENT_LHASH_DOALL_ARG_FN(STUFF, const STUFF, BIO)
/* ... then later in the code ... */
/* Print out the entire hashtable to a particular BIO */
lh_STUFF_doall_arg(hashtable, LHASH_DOALL_ARG_FN(STUFF_print), BIO,
logging_bio);
```

lh_<type>*_error()* can be used to determine if an error occurred in the last operation. lh_<type>*_error()* is a macro.

RETURN VALUES

lh_<type>*_new()* returns **NULL** on error, otherwise a pointer to the new **LHASH** structure.

When a hash table entry is replaced, lh_<type>*_insert()* returns the value being replaced. **NULL** is returned on normal operation and on error.

lh_<type>*_delete()* returns the entry being deleted. **NULL** is returned if there is no such value in the hash table.

lh_<type>*_retrieve()* returns the hash table entry if it has been found, **NULL** otherwise.

lh_<type>*_error()* returns 1 if an error occurred in the last operation, 0 otherwise.

lh_<type>*_free()*, lh_<type>*_doall()* and lh_<type>*_doall_arg()* return no values.

NOTE

The various LHASH macros and callback types exist to make it possible to write type-checked code without resorting to function-prototype casting - an evil that makes application code much harder to audit/verify and also opens the window of opportunity for stack corruption and other hard-to-find bugs. It also, apparently, violates ANSI-C.

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 (ie. 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 ''STUFF_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()* returns **NULL** both for success and error.

INTERNALS

The following description is based on the SSLeay documentation:

The **lhash** library implements a hash table described in the *Communications of the ACM* in 1991. What makes this hash table different is that as the table fills, the hash table is increased (or decreased) in size via *OPENSSL_realloc()*. When a 'resize' is done, instead of all hashes being redistributed over twice as many 'buckets', one bucket is split. So when an 'expand' is done, there is only a minimal cost to redistribute some values. Subsequent inserts will cause more single 'bucket' redistributions but there will never be a sudden large cost due to redistributing all the 'buckets'.

The state for a particular hash table is kept in the **LHASH** structure. The decision to increase or decrease the hash table size is made depending on the 'load' of the hash table. The load is the number of items in the hash table divided by the size of the hash table. The default values are as follows. If (hash->up_load < load) => expand. if (hash->down_load > load) => contract. The **up_load** has a default value of 1 and **down_load** has a default value of 2. These numbers can be modified by the application by just playing with the **up_load** and **down_load** variables. The 'load' is kept in a form which is multiplied by 256. So hash->up_load=8*256; will cause a load of 8 to be set.

If you are interested in performance the field to watch is num_comp_calls. The hash library keeps track of the 'hash' value for each item so when a lookup is done, the 'hashes' are compared, if there is a match, then a full compare is done, and hash->num_comp_calls is incremented. If num_comp_calls is not equal to num_delete plus num_retrieve it means that your hash function is generating hashes that are the same for different values. It is probably worth changing your hash function if this is the case because even if your hash table has 10 items in a 'bucket', it can be searched with 10 **unsigned long** compares and 10 linked list traverses. This will be much less expensive that 10 calls to your compare function.

lh_strhash() is a demo string hashing function:

```
unsigned long lh_strhash(const char *c);
```

Since the **LHASH** routines would normally be passed structures, this routine would not normally be passed to lh_<type>_*new()*, rather it would be used in the function passed to lh_<type>_*new()*.

SEE ALSO

lh_stats(3)

HISTORY

The **lhash** library is available in all versions of SSLeay and OpenSSL. *lh_err or()* was added in SSLeay 0.9.1b.

This manpage is derived from the SSLeay documentation.

In OpenSSL 0.9.7, all lhash functions that were passed function pointers were changed for better type safety, and the function types LHASH_COMP_FN_TYPE, LHASH_HASH_FN_TYPE, LHASH_DOALL_FN_TYPE and LHASH_DOALL_ARG_FN_TYPE became available.

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