NAME

caesar_table_1 – the “table_1” library of OPEN/CAESAR

PURPOSE

The “table_1” library provides primitives for managing a “state space”. It can be used either for breadth-first or depth-first search in the state graph.

USAGE

The “table_1” library consists of:

- a predefined header file **`caesar_table_1.h`**;
- the precompiled library file **`libcaesar.a`**, which implements the features described in **`caesar_table_1.h`**.

Note: The “table_1” library is a software layer built above the primitives offered by the “standard”, “area_1”, and “hash” libraries, and by the *OPEN/CAESAR* graph module.

DESCRIPTION

A “table” is basically a set of items.

Each item in the table is basically a byte string of fixed size. All items in a given table have the same size. An item can be considered as a tuple with two fields, whose size and contents are freely determined by the user:

- (1) a “base” field, that is a byte string of fixed size. In a given table, all base fields have the same size. This size must be greater than zero.

  More often than not, the base field contains a state (as defined in the graph module). However, this is not mandatory, and base fields can contain other information than states.
- (2) a “mark” field, that is a byte string whose size and contents are freely determined by the user. In a given table, all mark fields have the same size, which must be greater or equal to zero. Pointers to mark fields will be considered as values of type **`CAESAR_TYPE_POINTER`**; “mark” fields are always aligned on appropriate boundaries so that the user can put any information in these fields without alignment problem.

The user also determines the nature of the data stored in these fields, which is not meaningful to the “table_1” library.

Invariant property 1: the table is organized in such a way that all items have different base fields. Said differently, two items in a given table cannot have identical base fields (but they can have identical mark fields).

Invariant property 2: the number of items in a given table never decreases. New items can be inserted in the table, existing items can be replaced by new ones, but no item can be removed if it is not replaced by another one. Exception to this rule: it is possible to purge the table, i.e., to remove simultaneously all its items.

Invariant property 3: it is not allowed to modify the base field of any item in the table (except if this item is to be replaced by another one that has exactly the same hash value, which is unlikely). But it is possible to modify the mark field of any item.

Each item in a table is given a unique identification number (“index”) which is a natural number. A table can contain no more than a maximum of M items, with indexes between 0 and M - 1. Currently, M =

2^{29} = 536,870,912 on 32-bit machines and M = 2^{34} = 17,179,869,184 on 64-bit machines. But, for each table, the user can also limit the maximal number of items to a lesser bound N <= M.

When the table overflows (either because the maximum number of items is reached or because there is no enough memory to store new items), an action chosen by the user (e.g., abort, recovery, etc.) is performed.

Each item in the table can be accessed in three different ways:

- (1) by using its address (i.e., a pointer to the memory location where it is stored in the table),
- (2) by using its index,
- (3) by using its base field.

The table data structure establishes a correspondence between these three data. Indeed:

- given an address, one can retrieve the index, the base field, and the mark field of the corresponding item;
- given an index, one can retrieve the address, the base field, and the mark field of the corresponding item;
- given a base field, one can retrieve the address, the index, and the mark field of the corresponding item.

Retrieving the address and the index of an item from its base field involves some associative search. To allow fast retrievals, an hash-table is associated to each table. This is quite transparent from the user's point of view. Only the base field is taken into account when computing the hash-value and comparing items; the mark field is not meaningful for the search.

Two special variables are associated to a given table:

- the ‘‘put index’’ corresponds to the last item inserted in the table. Initialized to zero, the put index is incremented each time a new item is inserted. Therefore, the first item is numbered 0, the second one is numbered 1, etc.

  The ‘‘put index’’ is always useful, whatever the way the graph is explored: breadth-first, depth-first, etc.
- the ‘‘get index’’ is associated to the last item consulted in the table. Initialized to zero, the get index is incremented each time a new item is consulted. Since only the items previously inserted in the table can be consulted, the get index is always less or equal to the put index.

  The ‘‘get index’’ can be used to consult sequentially all the items, in the same order as they have been inserted in the table. Therefore it can be used for (pseudo) breadth-first exploration, but not for depth-first exploration.

Additionally, statistics are attached to each table. These statistics consist of a ‘‘success counter’’ and a ‘‘failure counter’’, which respectively count how many retrievals (given the base field) have succeeded and failed.

FEATURES

....

CAESAR_TYPE_TABLE_1

```
typedef CAESAR_TYPE_ABSTRACT (...) CAESAR_TYPE_TABLE_1;
```

This type denotes a pointer to the concrete representation of a table. The table representation is supposed to be “opaque”.

....

CAESAR_TYPE_INDEX_TABLE_1

```
typedef CAESAR_TYPE_NATURAL CAESAR_TYPE_INDEX_TABLE_1;
```

This type denotes an index, which is a natural number.

....

CAESAR_NULL_INDEX_TABLE_1

```
#define CAESAR_NULL_INDEX_TABLE_1 ((CAESAR_TYPE_INDEX_TABLE_1) -1L)
```

This constant denotes a special index value corresponding to the largest unsigned integer. Since item indexes are always in the range 0..M-1, no item index can be equal to **CAESAR_NULL_INDEX_TABLE_1**.

....

CAESAR_MAX_INDEX_TABLE_1

```
CAESAR_TYPE_INDEX_TABLE_1 CAESAR_MAX_INDEX_TABLE_1 ()
   { ... }
```

This function returns the value of M, i.e., the maximal number of items that can be stored in a table. Since item indexes are always in the range 0..M-1, no item index can be equal to **CAESAR_MAX_INDEX_TABLE_1 ()**.

....

CAESAR_TYPE_OVERFLOW_FUNCTION_TABLE_1

```
typedef void (*CAESAR_TYPE_OVERFLOW_FUNCTION_TABLE_1) (CAESAR_TYPE_TABLE_1);
```

CAESAR_TYPE_OVERFLOW_FUNCTION_TABLE_1 is the “pointer to an overflow procedure” type used in the “table_1” library. An overflow procedure takes one parameter of type **CAESAR_TYPE_TABLE_1**. Examples of overflow procedures are **CAESAR_OVERFLOW_SIGNAL_TABLE_1 ()**, **CAESAR_OVERFLOW_ABORT_TABLE_1 ()**, and **CAESAR_OVERFLOW_IGNORE_TABLE_1 ()** defined below.

....

CAESAR_OVERFLOW_SIGNAL_TABLE_1

```
void CAESAR_OVERFLOW_SIGNAL_TABLE_1 (CAESAR_T)
    CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This procedure is a possible action that can be performed in case the table pointed to by **CAESAR_T** overflows (either because the maximum number of items for **CAESAR_T** is reached or because there is no enough memory to store new items).

It first prints an error message to the standard output, and also various statistics about the table. Then, it returns. Practically, if the table is used for state space exploration, this means that some portions of the graph will not be explored, but an error message will be issued.

....

CAESAR_OVERFLOW_ABORT_TABLE_1

```
void CAESAR_OVERFLOW_ABORT_TABLE_1 (CAESAR_T)
    CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This procedure is a possible action that can be performed in case the table pointed to by **CAESAR_T** overflows (either because the maximum number of items for **CAESAR_T** is reached or because there is no enough memory to store new items).

It first prints an error message to the standard output, and also various statistics about the table. Then, it aborts the program using the C function **exit(3)**. The error code 1 is returned.

....

CAESAR_OVERFLOW_IGNORE_TABLE_1

```
void CAESAR_OVERFLOW_IGNORE_TABLE_1 (CAESAR_T)
    CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This procedure is a possible action that can be performed in case the table pointed to by **CAESAR_T** overflows (either because the maximum number of items for **CAESAR_T** is reached or because there is no enough memory to store new items).

It does nothing and returns. Practically, if the table is used for state space exploration, this means that some portions of the graph will not be explored; they are silently ignored.

....

CAESAR_CREATE_TABLE_1

```
void CAESAR_CREATE_TABLE_1 (CAESAR_T, CAESAR_BASE_AREA, CAESAR_MARK_AREA,
                            CAESAR_LIMIT_SIZE, CAESAR_HASH_SIZE,
                            CAESAR_PRIME, CAESAR_COMPARE, CAESAR_HASH,
                            CAESAR_PRINT, CAESAR_OVERFLOW)
   CAESAR_TYPE_TABLE_1 *CAESAR_T;
   CAESAR_TYPE_AREA_1 CAESAR_BASE_AREA;
   CAESAR_TYPE_AREA_1 CAESAR_MARK_AREA;
   CAESAR_TYPE_NATURAL CAESAR_LIMIT_SIZE;
   CAESAR_TYPE_NATURAL CAESAR_HASH_SIZE;
   CAESAR_TYPE_BOOLEAN CAESAR_PRIME;
   CAESAR_TYPE_COMPARE_FUNCTION CAESAR_COMPARE;
   CAESAR_TYPE_HASH_FUNCTION CAESAR_HASH;
   CAESAR_TYPE_PRINT_FUNCTION CAESAR_PRINT;
   CAESAR_TYPE_OVERFLOW_FUNCTION_TABLE_1 CAESAR_OVERFLOW;
   { ... }
```

This procedure allocates a table using **CAESAR_CREATE()** and assigns its address to *__CAESAR_T__. If the allocation fails, the **NULL** value is assigned to *__CAESAR_T__.

Note: when it is called, this procedure does not allocate at once all the memory needed to represent the table: the table will grow progressively as new items are inserted. Consequently, if **CAESAR_CREATE_TABLE_1()** returns a value different from **NULL**, this does not mean that no overflow will occur in the future.

Note: because **CAESAR_TYPE_TABLE_1** is a pointer type, any variable **CAESAR_T** of type **CAESAR_TYPE_TABLE_1** must be allocated before used, for instance using:

```
CAESAR_CREATE_TABLE_1 (&CAESAR_T, ...);
```

The value of **CAESAR_BASE_AREA** determines the (constant) size and (constant) alignment factor of the base field in the table. In the particular case where base fields are used to store states (resp. labels, strings), one must give the value **CAESAR_STATE_AREA_1()** (resp. **CAESAR_LABEL_AREA_1()**, **CAESAR_STRING_AREA_1()**) to the formal parameter **CAESAR_BASE_AREA**.

Note: For backward compatibility reasons, if **CAESAR_BASE_AREA** is equal to **CAESAR_EMPTY_AREA_1()**, it will be treated exactly like **CAESAR_STATE_AREA_1()**, i.e., specifying an empty area for the base field is equivalent to specifying a state area. However, relying on this feature is not recommended and this case will no longer be considered in the sequel of this manual.

The value of **CAESAR_MARK_AREA** determines the (constant) size and (constant) alignment factor of the mark field according to the specifications of the ‘‘area_1’’ library. In particular, if **CAESAR_MARK_AREA** is equal to **CAESAR_EMPTY_AREA_1()**, there will be no mark field in the table.

Each item in the table will be represented as a byte string of fixed size **caesar_item_size**, such that **caesar_item_size** is strictly greater than **caesar_base_size + caesar_mark_size**, where **caesar_base_size** denotes the size (in bytes) of the base field (i.e., **CAESAR_SIZE_AREA_1 (CAESAR_BASE_AREA)**) and where **caesar_mark_size** denotes the size (in bytes) of the mark field, if any (i.e., **CAESAR_SIZE_AREA_1 (CAESAR_MARK_AREA)**).

An item in the table contains not only the base field and the mark field, but also additional information needed for internal management. Also, ‘‘padding’’ bytes may be inserted around the base and mark fields to ensure that these fields are correctly aligned according to **CAESAR_ALIGNMENT_AREA_1 (CAESAR_BASE_AREA)** and **CAESAR_ALIGNMENT_AREA_1 (CAESAR_MARK_AREA)**.

The value of **CAESAR_LIMIT_SIZE** determines the maximal number of items that can be stored in the table; all indexes will consequently be in the range 0..**CAESAR_LIMIT_SIZE** - 1. It must be less or equal to M. If it is equal to zero, it is replaced by the default value M.

Note: in order to spare memory, the value of **CAESAR_LIMIT_SIZE** (which is an upper bound on the number of items to be inserted in the table) should be as small as possible. This can only be done if the user has some knowledge about the way the table will be used. The value of **CAESAR_LIMIT_SIZE** (or its default value if **CAESAR_LIMIT_SIZE** is equal to zero) might be reduced automatically if it exceeds the number of all possible different base fields or if it exceeds the amount of physical memory available (which is either specified by the environment variable **$CADP_MEMORY** or determined automatically by the **cadp_memory** program).

The value of **CAESAR_HASH_SIZE** determines the number of entries in the hash-table associated to the table. If **CAESAR_HASH_SIZE** is different from zero, it remains constant during the entire existence of the table (static hashing scheme). If **CAESAR_HASH_SIZE** is equal to zero, it is replaced with a default value greater than zero that might increase automatically when a sufficiently large number of items have been inserted into the table (dynamic hashing scheme).

Note: in order to spare memory, the value of **CAESAR_HASH_SIZE** (or its default value if **CAESAR_HASH_SIZE** is equal to zero) might be reduced automatically if it exceeds the value of **CAESAR_LIMIT_SIZE**, i.e., the maximal number of items that can be stored in the table, or if it exceeds the maximal number of different hash values that can be obtained taking into account the ‘‘hashable’’ size of **CAESAR_BASE_AREA**.

If the value of **CAESAR_PRIME** is equal to **CAESAR_TRUE** and if the value of **CAESAR_HASH_SIZE** is not a prime number, this value will be replaced by the nearest smaller prime number (since some hash functions require prime modulus). Otherwise, the value of **CAESAR_HASH_SIZE** will be kept unchanged.

The actual value of the formal parameter **CAESAR_COMPARE** will be stored and associated to the table pointed to by ∗**CAESAR_T**. It will be used as a comparison function when it is necessary to decide whether two base fields are equal or not.

Precisely, the actual value of **CAESAR_COMPARE** should be a pointer to a comparison function with two parameters **caesar_base_1** and **caesar_base_2** that returns **CAESAR_TRUE** if the two base fields pointed to by **caesar_base_1** and **caesar_base_2** are equal.

If the actual value of the formal parameter **CAESAR_COMPARE** is **NULL**, it is replaced by a pointer to a default comparison function that depends on the value of **CAESAR_BASE_AREA** and is determined according to the rules specified for function **CAESAR_USE_COMPARE_FUNCTION_AREA_1()** of the ‘‘area_1’’ library.

The actual value of the formal parameter **CAESAR_HASH** will be stored and associated to the table pointed to by ∗**CAESAR_T**. It will be used as a hash-function when it is necessary to compute a hash-value for searching or inserting an item in the table.

Precisely, the actual value of **CAESAR_HASH** should be a pointer to a hash function with two parameters **caesar_pointer** and **caesar_modulus** that returns a natural number in the range 0..**caesar_modulus**-1.

If the actual value of the formal parameter **CAESAR_HASH** is **NULL**, it is replaced by a pointer to a default hash function that depends on the value of **CAESAR_BASE_AREA** and is determined according to the rules specified for function **CAESAR_USE_HASH_FUNCTION_AREA_1()** of the ‘‘area_1’’ library.

Note: for backward compatibility reasons, the current implementation of **CAESAR_CREATE_TABLE_1()** tries to handle the case where **CAESAR_HASH** points to an hash function with three parameters (such as the **CAESAR_0_HASH()**, **CAESAR_1_HASH()**, ... functions provided by the ‘‘hash’’ library) instead of two. However, this support for hash functions with three parameters only occurs under very specific circumstances (e.g., if **CAESAR_BASE_AREA** has a null exponent field and a non-null length field). Relying on this feature is not recommended and this case will no longer be considered in the sequel of this manual.

The actual value of the formal parameter **CAESAR_PRINT** will be stored and associated to the table pointed to by ∗**CAESAR_T**. It will be used subsequently to print the items of this table.

Precisely, the actual value of **CAESAR_PRINT** should be a pointer to a printing procedure with two parameters **caesar_file** and **caesar_item** that prints to file **caesar_file** information about the contents (base field and/or mark field, if any) of the item pointed to by **caesar_item**.

If the actual value of the formal parameter **CAESAR_PRINT** is **NULL**, it is replaced by a pointer to a default procedure that prints the base field and the mark field, if any. The printing procedure used for the base field (respectively, the mark field) depends on the value of **CAESAR_BASE_AREA** (resp. **CAESAR_MARK_AREA**) and is determined according to the rules specified for function **CAESAR_USE_PRINT_FUNCTION_AREA_1()** of the "area_1" library. The actual value of the formal parameter **CAESAR_OVERFLOW** will be stored and associated to the table pointed to by ∗**CAESAR_T**. It will be used subsequently to determine the action to take if the table pointed to by ∗**CAESAR_T** overflows: in this case, the procedure pointed to by **CAESAR_OVERFLOW** will be called with the overflowing table ∗**CAESAR_T** passed as actual parameter.

The above procedures **CAESAR_OVERFLOW_SIGNAL_TABLE_1()**, **CAESAR_OVERFLOW_ABORT_TABLE_1()**, and **CAESAR_OVERFLOW_IGNORE_TABLE_1()**, can be used as actual values for the formal parameter **CAESAR_OVERFLOW**. If the actual value of the formal parameter **CAESAR_OVERFLOW** is **NULL**, it is replaced by the default value **CAESAR_OVERFLOW_SIGNAL_TABLE_1**.

The table is initially empty. The success and failure counters attached to the table are both initialized to 0.

....

CAESAR_DELETE_TABLE_1

```
void CAESAR_DELETE_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 *CAESAR_T;
   { ... }
```

This procedure frees the memory space corresponding to the table pointed to by ∗**CAESAR_T** using **CAESAR_DELETE()**. The items contained in the table and its associated hash-table are also freed. Afterwards, the **NULL** value is assigned to ∗**CAESAR_T**.

....

CAESAR_PURGE_TABLE_1

```
void CAESAR_PURGE_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This procedure empties the table pointed to by **CAESAR_T** without deleting it. Each item contained in the table is freed using **CAESAR_DELETE()**. Afterwards, this table is exactly in the same state as after its creation using **CAESAR_CREATE_TABLE_1()**.

....

CAESAR_PUT_INDEX_TABLE_1

```
CAESAR_TYPE_INDEX_TABLE_1 CAESAR_PUT_INDEX_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This function returns the number of items that have been put in the table pointed to by **CAESAR_T**, which is also equal to the index of the next item to be put in the table. This number is initialized to 0 and it is incremented every time the **CAESAR_PUT_TABLE_1()** function (see below) is called.

Note: This number is always less or equal to N, where N is the maximal number of items that can be stored in the table; the value of N depends on the actual value given to the formal parameter **CAESAR_LIMIT_SIZE** when the table was created using **CAESAR_CREATE_TABLE_1 ()**.

Note: When the table is empty, the result returned by this function is equal to zero. When the table is full, the result returned by this function is equal to N.

....

CAESAR_PUT_BASE_TABLE_1

```
CAESAR_TYPE_POINTER CAESAR_PUT_BASE_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

If the table pointed to by **CAESAR_T** is not full, this function returns a pointer to the base field of the next item to be put in the table. This base field is initially undefined and must be assigned before calling some other functions of the ‘‘table_1’’ library (see below).

If the table pointed to by **CAESAR_T** is full, this function returns a pointer to the base field of a special item located beyond the table. It is permitted to modify (and subsequently consult) the contents of this base field, but invoking the **CAESAR_PUT_TABLE_1 ()** function (see below) will cause an overflow.

....

CAESAR_PUT_MARK_TABLE_1

```
CAESAR_TYPE_POINTER CAESAR_PUT_MARK_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

If there are no mark fields in the table (due to the initialization parameters supplied to **CAESAR_CREATE_TABLE_1 ()**), the result returned by **CAESAR_PUT_MARK_TABLE_1 ()** is always undefined.

If the table pointed to by **CAESAR_T** is not full, this function returns a pointer to the mark field of the next item to be put in the table. This mark field is always initialized to a bit string of 0’s. It can be consulted and modified.

If the table pointed to by **CAESAR_T** is full, this function returns a pointer to the mark field of a special item located beyond the table. It is permitted to modify and consult the contents of this mark field, but invoking the **CAESAR_PUT_TABLE_1 ()** function (see below) will cause an overflow.

....

CAESAR_PUT_TABLE_1

```
void CAESAR_PUT_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This procedure puts into the table pointed to by **CAESAR_T** the item whose base field is pointed to by **CAESAR_PUT_BASE_TABLE_1 (CAESAR_T)** and whose mark field (if any) is pointed to by **CAESAR_PUT_MARK_TABLE_1 (CAESAR_T)**.

The base field must have been assigned before this procedure is called.

This procedure assumes that no other item in the table has the same base field. There is no attempt to check the validity of this assumption. It is therefore of the user's responsibility to ensure that this assumption is true. See also function **CAESAR_SEARCH_AND_PUT_TABLE_1()** below.

The hash-table associated to the table is updated to take into account the new item. To compute the hash-value for the base field, the hash-function associated with the table is used.

If the maximum number of items in the table was already reached when the procedure **CAESAR_PUT_TABLE_1()** was called, or in case of memory shortage, the overflow procedure associated with **CAESAR_T** is called with the actual parameter **CAESAR_T**.

Finally, **CAESAR_PUT_INDEX_TABLE_1 (CAESAR_T)** is incremented; **CAESAR_PUT_BASE_TABLE_1 (CAESAR_T)** and **CAESAR_PUT_MARK_TABLE_1 (CAESAR_T)** are advanced, respectively, to the base field and the mark field (if any) of the next free item.

Note: the table is implemented in such a way that **CAESAR_PUT_BASE_TABLE_1()** and **CAESAR_PUT_MARK_TABLE_1()** always return a valid pointer, even if the table is already full. Overflow can only occur when **CAESAR_PUT_TABLE_1()** is called, but not when **CAESAR_PUT_BASE_TABLE_1()** or **CAESAR_PUT_MARK_TABLE_1()** are called.

....

CAESAR_SEARCH_TABLE_1

```
CAESAR_TYPE_BOOLEAN CAESAR_SEARCH_TABLE_1 (CAESAR_T, CAESAR_B, CAESAR_I, CAESAR_P)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   CAESAR_TYPE_POINTER CAESAR_B;
   CAESAR_TYPE_INDEX_TABLE_1 *CAESAR_I;
   CAESAR_TYPE_POINTER *CAESAR_P;
   { ... }
```

This function determines if there exists, in the table pointed to by **CAESAR_T**, an item whose base field is equal to the byte string pointed to by **CAESAR_B**. Byte string comparisons are performed using the comparison function associated to the table. The search is done using the hash-function and hash-table associated to the table.

If so, this function returns **CAESAR_TRUE**. In this case, the index and the address of the item are respectively assigned to ∗**CAESAR_I** and ∗**CAESAR_P**. The success counter attached to the table is incremented. If not, this function returns **CAESAR_FALSE**. In this case, both variables ∗**CAESAR_I** and ∗**CAESAR_P** are left unchanged. The failure counter attached to the table is incremented.

....

CAESAR_SEARCH_AND_PUT_TABLE_1

```
CAESAR_TYPE_BOOLEAN CAESAR_SEARCH_AND_PUT_TABLE_1 (CAESAR_T, CAESAR_I, CAESAR_P)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   CAESAR_TYPE_INDEX_TABLE_1 *CAESAR_I;
   CAESAR_TYPE_POINTER *CAESAR_P;
    { ... }
```

This function is a combination of the function **CAESAR_SEARCH_TABLE_1()** and the procedure

CAESAR_PUT_TABLE_1() defined above. The base field pointed to by **CAESAR_PUT_BASE_TABLE_1 (CAESAR_T)** must have been assigned before this function is called.

It first determines if there exists, in the table pointed by **CAESAR_T**, an item whose base field is equal to the base field of the item pointed to by **CAESAR_PUT_BASE_TABLE_1 (CAESAR_T)**. Byte string comparisons are performed using the comparison function associated to the table. The search is done using the hash-function and hash-table associated to the table.

If so, this function returns **CAESAR_TRUE**. In this case, the index and the address of the existing item are respectively assigned to ***CAESAR_I** and ***CAESAR_P**. The success counter attached to the table is incremented. If not, this function returns **CAESAR_FALSE**. In this case, it puts into the table pointed to by **CAESAR_T** the item whose base field is pointed to by **CAESAR_PUT_BASE_TABLE_1 (CAESAR_T)** and whose mark field (if any) is pointed to by **CAESAR_PUT_MARK_TABLE_1 (CAESAR_T)**. The hash-table associated to the table is updated to take into account the new item. The overflow procedure associated with the table is called if overflow occurs.

CAESAR_PUT_INDEX_TABLE_1 (CAESAR_T) is copied into ***CAESAR_I** and then incremented. **CAESAR_PUT_BASE_TABLE_1 (CAESAR_T)** is copied into ***CAESAR_P** and then advanced to the base field of the next free item. The failure counter attached to the table is incremented.

Note: formally the body of function **CAESAR_SEARCH_AND_PUT_TABLE_1()** could be defined as follows:

```
{
CAESAR_TYPE_BOOLEAN caesar_found;
caesar_found = CAESAR_SEARCH_TABLE_1 (CAESAR_T,
        CAESAR_PUT_BASE_TABLE_1 (CAESAR_T), CAESAR_I, CAESAR_P);
if (! caesar_found)
        {
        *CAESAR_I = CAESAR_PUT_INDEX_TABLE_1 (CAESAR_T);
        *CAESAR_P = CAESAR_PUT_BASE_TABLE_1 (CAESAR_T);
        CAESAR_PUT_TABLE_1 (CAESAR_T);
        }
}
```

Practically, it is implemented differently, for efficiency reasons (the computation of the hash-value and the access through the hash-table are performed only once, not twice).

....

CAESAR_GET_INDEX_TABLE_1

```
CAESAR_TYPE_INDEX_TABLE_1 CAESAR_GET_INDEX_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This function returns the number of items that have been got from the table pointed to by **CAESAR_T**, which is also the index of the next item to be got from the table. This number is initialized to 0 and it is incremented every time the **CAESAR_GET_TABLE_1()** function (see below) is called.

Note: This number is always less or equal to the value returned by **CAESAR_PUT_INDEX_TABLE_1()**.

....

CAESAR_GET_BASE_TABLE_1

```
CAESAR_TYPE_POINTER CAESAR_GET_BASE_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This function returns a pointer to the base field of the next item to be got in the table pointed to by **CAESAR_T**.

This pointer can only be used if **CAESAR_GET_INDEX_TABLE_1 (CAESAR_T)** is strictly less than **CAESAR_PUT_INDEX_TABLE_1 (CAESAR_T)**. In the opposite case, the result of this function is undefined (since it is not possible to get items that have not been put yet).

The base field pointed to by the result of **CAESAR_GET_BASE_TABLE_1()** can be consulted, but not modified.

....

CAESAR_GET_MARK_TABLE_1

```
CAESAR_TYPE_POINTER CAESAR_GET_MARK_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This function returns a pointer to the mark field of the next item to be got in the table pointed to by **CAESAR_T**. If there are no mark fields in the table (due to the initialization parameters supplied to **CAESAR_CREATE_TABLE_1()**) the result is undefined.

This pointer can only be used if **CAESAR_GET_INDEX_TABLE_1 (CAESAR_T)** is strictly less than **CAESAR_PUT_INDEX_TABLE_1 (CAESAR_T)**. In the opposite case, the result of this function is undefined (since it is not possible to get items that have not been put yet).

The mark field pointed to by the result of **CAESAR_GET_MARK_TABLE_1()** can be either consulted or modified.

....

CAESAR_GET_TABLE_1

```
void CAESAR_GET_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This procedure increments **CAESAR_GET_INDEX_TABLE_1 (CAESAR_T)**; it advances, respectively, **CAESAR_GET_BASE_TABLE_1 (CAESAR_T)** and **CAESAR_GET_MARK_TABLE_1 (CAESAR_T)** to the base field and the mark field (if any) of the next free item.

If **CAESAR_GET_INDEX_TABLE_1 (CAESAR_T)** is equal to **CAESAR_PUT_INDEX_TABLE_1 (CAESAR_T)** when the procedure **CAESAR_GET_TABLE_1()** is called, the result is undefined.

....

CAESAR_EMPTY_TABLE_1

```
CAESAR_TYPE_BOOLEAN CAESAR_EMPTY_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This function returns a value different from 0 if the table pointed to by **CAESAR_T** is empty, and 0 otherwise. **CAESAR_EMPTY_TABLE_1 (CAESAR_T)** is always equivalent to:

```
CAESAR_PUT_INDEX_TABLE_1 (CAESAR_T) == 0
```

....

CAESAR_FULL_TABLE_1

```
CAESAR_TYPE_BOOLEAN CAESAR_FULL_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This function returns a value different from 0 if the table pointed to by **CAESAR_T** is full, and 0 otherwise. **CAESAR_FULL_TABLE_1 (CAESAR_T)** is always equivalent to:

```
CAESAR_PUT_INDEX_TABLE_1 (CAESAR_T) == $N$
```

where N denotes the maximum number of items that the table can contain; the value of N depends on the actual value given to the formal parameter **CAESAR_LIMIT_SIZE** when the table was created using **CAESAR_CREATE_TABLE_1()**.

....

CAESAR_EXPLORED_TABLE_1

```
CAESAR_TYPE_BOOLEAN CAESAR_EXPLORED_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This function returns a value different from 0 if the get index and the put index are identical, or 0 otherwise. **CAESAR_EXPLORED_TABLE_1 (CAESAR_T)** is always equivalent to:

```
CAESAR_GET_INDEX_TABLE_1 (CAESAR_T) == CAESAR_PUT_INDEX_TABLE_1 (CAESAR_T)
```

....

CAESAR_RETRIEVE_I_B_TABLE_1

```
void CAESAR_RETRIEVE_I_B_TABLE_1 (CAESAR_T, CAESAR_I, CAESAR_B)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   CAESAR_TYPE_INDEX_TABLE_1 CAESAR_I;
   CAESAR_TYPE_POINTER *CAESAR_B;
   { ... }
```

This procedure computes the address of the base field of the item with index **CAESAR_I** in the table pointed to by **CAESAR_T**. This address is assigned to ***CAESAR_B**.

If **CAESAR_I** is greater or equal to the value returned by **CAESAR_PUT_INDEX_TABLE_1 ()**, a **NULL** pointer is assigned to ***CAESAR_B**.

....

CAESAR_RETRIEVE_I_M_TABLE_1

```
void CAESAR_RETRIEVE_I_M_TABLE_1 (CAESAR_T, CAESAR_I, CAESAR_M)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   CAESAR_TYPE_INDEX_TABLE_1 CAESAR_I;
   CAESAR_TYPE_POINTER *CAESAR_M;
   { ... }
```

This procedure computes the address of the mark field of the item with index **CAESAR_I** in the table pointed to by **CAESAR_T**. This address is assigned to ***CAESAR_M**.

If **CAESAR_I** is greater or equal to the value returned by **CAESAR_PUT_INDEX_TABLE_1 ()**, a **NULL** pointer is assigned to ***CAESAR_M**.

If there are no mark fields in the table (due to the initialization parameters supplied to **CAESAR_CREATE_TABLE_1 ()**) the effect of this procedure is undefined.

....

CAESAR_RETRIEVE_I_BM_TABLE_1

```
void CAESAR_RETRIEVE_I_BM_TABLE_1 (CAESAR_T, CAESAR_I, CAESAR_B, CAESAR_M)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   CAESAR_TYPE_INDEX_TABLE_1 CAESAR_I;
   CAESAR_TYPE_POINTER *CAESAR_B;
   CAESAR_TYPE_POINTER *CAESAR_M;
   { ... }
```

This procedure computes the address of the base field of the item with index **CAESAR_I** in the table pointed to by **CAESAR_T**. This address is assigned to ***CAESAR_B**.

It also computes the address of the mark field of the item with index **CAESAR_I** in the table pointed to by **CAESAR_T**. This address is assigned to ***CAESAR_M**.

If **CAESAR_I** is greater or equal to the value returned by **CAESAR_PUT_INDEX_TABLE_1 ()**, a **NULL** pointer is assigned to ***CAESAR_B** and ***CAESAR_M**.

If there are no mark fields in the table (due to the initialization parameters supplied to **CAESAR_CREATE_TABLE_1 ()**) the effect of this procedure is undefined.

....

CAESAR_RETRIEVE_B_I_TABLE_1

```
void CAESAR_RETRIEVE_B_I_TABLE_1 (CAESAR_T, CAESAR_B, CAESAR_I)
CAESAR_TYPE_TABLE_1 CAESAR_T;
```

```
CAESAR_TYPE_POINTER CAESAR_B;
CAESAR_TYPE_INDEX_TABLE_1 *CAESAR_I;
   { ... }
```

This procedure computes the index of the item whose base field, in the table pointed to by **CAESAR_T**, is pointed to by **CAESAR_B**. This index is assigned to ***CAESAR_I**.

If no item stored in the table has a base field at address **CAESAR_B**, the **CAESAR_NULL_INDEX_TABLE_1** value is assigned to ***CAESAR_I**.

....

CAESAR_RETRIEVE_M_I_TABLE_1

```
void CAESAR_RETRIEVE_M_I_TABLE_1 (CAESAR_T, CAESAR_M, CAESAR_I)
CAESAR_TYPE_TABLE_1 CAESAR_T;
CAESAR_TYPE_POINTER CAESAR_M;
CAESAR_TYPE_INDEX_TABLE_1 *CAESAR_I;
   { ... }
```

This procedure computes the index of the item whose mark field, in the table pointed to by **CAESAR_T**, is pointed to by **CAESAR_M**. This index is assigned to ***CAESAR_I**.

If no item stored in the table has a mark field at address **CAESAR_M**, the **CAESAR_NULL_INDEX_TABLE_1** value is assigned to ***CAESAR_I**.

If there are no mark fields in the table (due to the initialization parameters supplied to **CAESAR_CREATE_TABLE_1()**) the effect of this procedure is undefined.

....

CAESAR_RETRIEVE_B_M_TABLE_1

```
void CAESAR_RETRIEVE_B_M_TABLE_1 (CAESAR_T, CAESAR_B, CAESAR_M)
CAESAR_TYPE_TABLE_1 CAESAR_T;
CAESAR_TYPE_POINTER CAESAR_B;
CAESAR_TYPE_POINTER *CAESAR_M;
   { ... }
```

This procedure computes, for the table pointed to by **CAESAR_T**, the address of the mark field of the item whose base field is pointed to by **CAESAR_B**. This address is assigned to ***CAESAR_M**.

If no item stored in the table has a base field at address **CAESAR_B**, the effect of this procedure is undefined.

If there are no mark fields in the table (due to the initialization parameters supplied to **CAESAR_CREATE_TABLE_1()**) the effect of this procedure is undefined.

....

CAESAR_RETRIEVE_M_B_TABLE_1

```
void CAESAR_RETRIEVE_M_B_TABLE_1 (CAESAR_T, CAESAR_M, CAESAR_B)
CAESAR_TYPE_TABLE_1 CAESAR_T;
```

```
CAESAR_TYPE_POINTER CAESAR_M;
CAESAR_TYPE_POINTER *CAESAR_B;
   { ... }
```

This procedure computes, for the table pointed to by **CAESAR_T**, the address of the base field of the item whose mark field is pointed to by **CAESAR_M**. This address is assigned to ***CAESAR_B**.

If no item stored in the table has a mark field at address **CAESAR_M**, the effect of this procedure is undefined.

If there are no mark fields in the table (due to the initialization parameters supplied to **CAESAR_CREATE_TABLE_1()**) the effect of this procedure is undefined.

....

CAESAR_FAILURE_TABLE_1

```
CAESAR_TYPE_NATURAL CAESAR_FAILURE_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This function returns the value of the failure counter of the table pointed to by **CAESAR_T**, i.e., the number of searches that failed.

....

CAESAR_SUCCESS_TABLE_1

```
CAESAR_TYPE_NATURAL CAESAR_SUCCESS_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This function returns the value of the success counter of the table pointed to by **CAESAR_T**, i.e., the number of searches that succeeded.

....

CAESAR_FORMAT_TABLE_1

```
CAESAR_TYPE_FORMAT CAESAR_FORMAT_TABLE_1 (CAESAR_T, CAESAR_FORMAT)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   CAESAR_TYPE_FORMAT CAESAR_FORMAT;
   { ... }
```

This function allows to control the format under which the table pointed to by **CAESAR_T** will be printed by the procedure **CAESAR_PRINT_TABLE_1()** (see below).

Currently, the following formats are available:

- With format 0, statistical information about the table is displayed such as: the number of items put, the number of items got, the size in bytes, the success counter, the failure counter, etc.
- With format 1, all items in the table are printed, sorted by increasing indexes. For each item, the corresponding index is displayed; the base field and the mark field (if any) are also displayed using the printing procedure associated to the table.

- With format 2, all items in the table are printed, sorted by increasing indexes. For each item, the index, the address, and the corresponding hash-value are displayed; the base field and the mark field (if any) are also displayed using the printing procedure associated to the table. Informations concerning the associated hash-table are also displayed. This format is mainly intended for debugging purpose.
- (no other format available yet)

By default, the current format of each table is initialized to 0.

When called with **CAESAR_FORMAT** between 0 and 2, this fonction sets the current format of **CAESAR_T** to **CAESAR_FORMAT** and returns an undefined result.

When called with another value of **CAESAR_FORMAT**, this function does not modify the current format of **CAESAR_T** but returns a result defined as follows. If **CAESAR_FORMAT** is equal to the constant **CAESAR_CURRENT_FORMAT**, the result is the value of the current format of **CAESAR_T**. If **CAESAR_FORMAT** is equal to the constant **CAESAR_MAXIMAL_FORMAT**, the result is the maximal format value (i.e., 2). In all other cases, the effect of this function is undefined.

....

CAESAR_MAX_FORMAT_TABLE_1

```
CAESAR_TYPE_FORMAT CAESAR_MAX_FORMAT_TABLE_1 ()
   { ... }
```

Caution! This function is deprecated. It should no longer be used, as it might be removed from future versions of the *OPEN/CAESAR*. Use function **CAESAR_FORMAT_TABLE_1()** instead, called with argument **CAESAR_MAXIMAL_FORMAT**.

This function returns the maximal format value available for printing tables.

....

CAESAR_PRINT_TABLE_1

```
void CAESAR_PRINT_TABLE_1 (CAESAR_FILE, CAESAR_T)
   CAESAR_TYPE_FILE CAESAR_FILE;
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This procedure prints to file **CAESAR_FILE** a text containing information about the table pointed to by **CAESAR_T**. The nature of the information is determined by the current format of the table pointed to by **CAESAR_T**.

Before this procedure is called, **CAESAR_FILE** must have been properly opened, for instance using **fopen(3)**.

....

AUTHOR(S)

Hubert Garavel

FILES

$CADP/incl/caesar_graph.h	interface of the graph module
$CADP/incl/caesar_*.h	interfaces of the storage module

$CADP/bin.‘arch‘/libcaesar.a	object code of the storage module
$CADP/src/open_caesar/*.c	source code of various exploration modules
$CADP/com/lotos.open	shell script to run OPEN/CAESAR

SEE ALSO

Reference Manuals of OPEN/CAESAR, CAESAR, and CAESAR.ADT, **lotos.open**(LOCAL), **caesar**(LOCAL), **caesar.adt**(LOCAL)

Additional information is available from the CADP Web page located at http://cadp.inria.fr

Directives for installation are given in files **$CADP/INSTALLATION_*.**

Recent changes and improvements to this software are reported and commented in file **$CADP/HISTORY.**

BUGS

Known bugs are described in the Reference Manual of OPEN/CAESAR. Please report new bugs to cadp@inria.fr