Libmaa User's Manual

Rickard E. Faith*
Department of Computer Science
University of North Carolina at Chapel Hill

TR96-009
DRAFT: Wed May 21 00:49:39 1997
Revision : 1.2

Contents

1 Functions **7**
1.1 General Support . . . 7
1.1.1 *maa_init* . . . 7
1.2 Argument List Routines . . . 7
1.2.1 *arg_create* . . . 7
1.2.2 *arg_destroy* . . . 7
1.2.3 *arg_add* . . . 7
1.2.4 *arg_addn* . . . 7
1.2.5 *arg_grow* . . . 7
1.2.6 *arg_finish* . . . 8
1.2.7 *arg_get* . . . 8
1.2.8 *arg_count* . . . 8
1.2.9 *arg_get_vector* . . . 8
1.2.10 *arg_argify* . . . 8
1.3 Base-26 Routines . . . 8
1.4 Base-64 Routines . . . 8
1.5 Bit manipulation routines . . . 9
1.5.1 *bit_set* . . . 9
1.5.2 *bit_clr* . . . 9
1.5.3 *bit_tst* . . . 9
1.5.4 *bit_cnt* . . . 9
1.6 Debugging Support . . . 9
1.6.1 *dbg_register* . . . 9

*Cray Fellow; supported by ARPA via ONR contract N00014-92-C-0182, by Rome Labs contract F30602-94-C-0037, and by a Cray Fellowship from Cray Research, Inc. and NCSC (North Carolina Supercomputing Center)..

1.6.2 *dbg_set* . . . 10
1.6.3 *dbg_set_flag* . . . 10
1.6.4 *dbg_unset_flag* . . . 10
1.6.5 *dbg_test* . . . 10
1.6.6 *dbg_destroy* . . . 10
1.7 Error Reporting Routines . . . 10
1.7.1 *err_set_program_name* . . . 10
1.7.2 *err_program_name* . . . 10
1.7.3 *err_fatal* . . . 11
1.7.4 *err_fatal_errno* . . . 11
1.7.5 *err_warning* . . . 11
1.7.6 *err_internal* . . . 11
1.8 Flag Support . . . 11
1.8.1 *flg_register* . . . 11
1.8.2 *flg_set* . . . 11
1.8.3 *flg_test* . . . 12
1.8.4 *flg_destroy* . . . 12
1.9 Hash Table Routines . . . 12
1.9.1 *hsh_create* . . . 12
1.9.2 *hsh_destroy* . . . 12
1.9.3 *hsh_insert* . . . 14
1.9.4 *hsh_delete* . . . 14
1.9.5 *hsh_retrieve* . . . 14
1.9.6 *hsh_iterate* . . . 14
1.9.7 *hsh_iterate_arg* . . . 14
1.9.8 *hsh_get_stats* . . . 15
1.9.9 *hsh_print_stats* . . . 15
1.9.10 *hsh_init_position* . . . 15
1.9.11 *hsh_next_position* . . . 16
1.9.12 *hsh_get_position* . . . 16
1.9.13 *hsh_readonly* . . . 16
1.10 List Routines . . . 16
1.10.1 *lst_create* . . . 16
1.10.2 *lst_destroy* . . . 16
1.10.3 *lst_append* . . . 16
1.10.4 *lst_push* . . . 17
1.10.5 *lst_pop* . . . 17
1.10.6 *lst_top* . . . 17
1.10.7 *lst_nth_get* . . . 17
1.10.8 *lst_nth_set* . . . 17
1.10.9 *lst_member* . . . 17
1.10.10 *lst_length* . . . 17
1.10.11 *lst_truncate* . . . 18
1.10.12 *lst_truncate_position* . . . 18
1.10.13 *lst_iterate* . . . 18
1.10.14 *lst_iterate_arg* . . . 18

1.10.15 *lst_init_position* . . . 18
1.10.16 *lst_last_position* . . . 18
1.10.17 *lst_next_position* . . . 19
1.10.18 *lst_nth_position* . . . 19
1.10.19 *lst_get_position* . . . 19
1.10.20 *lst_set_position* . . . 19
1.10.21 *_lst_dump_node* . . . 19
1.11 Memory Management Routines . . . 19
1.11.1 *mem_create_strings* . . . 19
1.11.2 *mem_destroy_strings* . . . 20
1.11.3 *mem_strcpy* . . . 20
1.11.4 *mem_strncpy* . . . 20
1.11.5 *mem_grow* . . . 20
1.11.6 *mem_finish* . . . 20
1.11.7 *mem_get_string_stats* . . . 20
1.11.8 *mem_print_string_stats* . . . 20
1.11.9 *mem_create_objects* . . . 21
1.11.10 *mem_destroy_objects* . . . 21
1.11.11 *mem_get_object* . . . 21
1.11.12 *mem_get_empty_object* . . . 21
1.11.13 *mem_free_object* . . . 21
1.11.14 *mem_get_object_stats* . . . 22
1.11.15 *mem_print_object_stats* . . . 22
1.12 Parsing (and Lexing) Support . . . 22
1.12.1 *prs_set_debug* . . . 22
1.12.2 *prs_set_cpp_options* . . . 22
1.12.3 *prs_file* . . . 22
1.12.4 *prs_file_nocpp* . . . 23
1.12.5 *prs_stream* . . . 23
1.12.6 *prs_make_integer* . . . 23
1.12.7 *prs_make_double* . . . 23
1.13 Process Management Routines . . . 23
1.14 Prime Number Routines . . . 23
1.14.1 *prm_is_prime* . . . 23
1.14.2 *prm_next_prime* . . . 24
1.15 Set Routines . . . 24
1.15.1 *set_create* . . . 24
1.15.2 *set_destroy* . . . 24
1.15.3 *set_insert* . . . 25
1.15.4 *set_delete* . . . 25
1.15.5 *set_member* . . . 25
1.15.6 *set_iterate* . . . 25
1.15.7 *set_iterate_arg* . . . 25
1.15.8 *set_init_position* . . . 25
1.15.9 *set_next_position* . . . 26
1.15.10 *set_get_position* . . . 26

1.15.11 *set_readonly* . . . 26
1.15.12 *set_union* . . . 26
1.15.13 *set_inter* . . . 26
1.15.14 *set_diff* . . . 26
1.15.15 *set_equal* . . . 27
1.15.16 *set_get_stats* . . . 27
1.15.17 *set_print_stats* . . . 27
1.16 Skip List Routines . . . 27
1.16.1 *sl_create* . . . 28
1.16.2 *sl_destroy* . . . 28
1.16.3 *_sl_shutdown* . . . 28
1.16.4 *sl_insert* . . . 28
1.16.5 *sl_delete* . . . 28
1.16.6 *sl_find* . . . 29
1.16.7 *sl_iterate* . . . 29
1.16.8 *sl_iterate_arg* . . . 29
1.16.9 *_sl_dump* . . . 29
1.17 Source Code Management . . . 29
1.17.1 *src_create* . . . 29
1.17.2 *src_destroy* . . . 29
1.17.3 *src_line* . . . 30
1.17.4 *src_new_file* . . . 30
1.17.5 *src_new_line* . . . 30
1.17.6 *src_advance* . . . 30
1.17.7 *src_get* . . . 30
1.17.8 *src_filename* . . . 30
1.17.9 *src_linenumber* . . . 30
1.17.10 *src_offset* . . . 30
1.17.11 *src_length* . . . 31
1.17.12 *src_source_line* . . . 31
1.17.13 *src_get_stats* . . . 31
1.17.14 *src_print_stats* . . . 31
1.17.15 *src_print_line* . . . 31
1.17.16 *src_parse_error* . . . 32
1.17.17 *src_print_error* . . . 32
1.17.18 *src_print_message* . . . 32
1.18 Stack Routines . . . 32
1.18.1 *stk_create* . . . 32
1.18.2 *stk_destroy* . . . 32
1.18.3 *stk_push* . . . 32
1.18.4 *stk_pop* . . . 33
1.18.5 *stk_top* . . . 33
1.19 String Pool Routines . . . 33
1.19.1 *str_pool_create* . . . 33
1.19.2 *str_pool_destroy* . . . 33
1.19.3 *str_pool_exists* . . . 33

1.19.4 *str_pool_find* . . . 33
1.19.5 *str_pool_copy* . . . 34
1.19.6 *str_pool_copyn* . . . 34
1.19.7 *str_pool_grow* . . . 34
1.19.8 *str_pool_finish* . . . 34
1.19.9 *str_pool_get_stats* . . . 34
1.19.10 *str_pool_print_stats* . . . 34
1.19.11 *str_exists* . . . 35
1.19.12 *str_find* . . . 35
1.19.13 *str_findn* . . . 35
1.19.14 *str_copy* . . . 35
1.19.15 *str_copyn* . . . 35
1.19.16 *str_grow* . . . 36
1.19.17 *str_finish* . . . 36
1.19.18 *str_unique* . . . 36
1.19.19 *str_destroy* . . . 36
1.19.20 *str_get_stats* . . . 36
1.19.21 *str_print_stats* . . . 36
1.19.22 *txt_soundex* . . . 37
1.20 Timer Support . . . 37
1.20.1 *tim_start* . . . 37
1.20.2 *tim_stop* . . . 37
1.20.3 *tim_reset* . . . 37
1.20.4 *tim_get_real* . . . 37
1.20.5 *tim_get_user* . . . 37
1.20.6 *tim_get_system* . . . 38
1.20.7 *tim_print_timer* . . . 38
1.20.8 *tim_print_timers* . . . 38
1.20.9 *_tim_shutdown* . . . 38

A Maa **39**

Abstract

The LIBMAA library provides many low-level data structures which are helpful for writing compilers, including hash tables, sets, lists, debugging support, and memory management. Although LIBMAA was designed and implemented as a foundation for the KHEPERA Transformation System, the data structures are generally applicable to a wide range of programming problems.

The memory management routines are especially helpful for improving the performance of memory-intensive applications.

1 Functions

1.1 General Support

1.1.1 *maa_init*

void *maa_init*(**const char** **programName*)

maa_init should be called at the start of **main()**, and serves to initialize debugging and other support for LIBMAA.

1.2 Argument List Routines

Argument lists are vectors of pointers to strings (e.g., the standard *char **argv*). These routines manage the efficient creation, manipulation, and deletion of these sorts of lists.

1.2.1 *arg_create*

arg_List arg_create(**void**)

Create an *arg_List* object.

1.2.2 *arg_destroy*

void *arg_destroy*(*arg_List arg*)

Free all memory associated with *arg*, including the memory used for the strings.

1.2.3 *arg_add*

arg_List arg_add(*arg_List arg*, **const char** **string*)

Add *string* as the next item in *arg*.

1.2.4 *arg_addn*

arg_List arg_addn(*arg_List arg*, **const char** **string*, **int** *length*)

Add *length* characters of *string* as the next item in *arg*. A terminating **NULL** is added to the copied *string*.

1.2.5 *arg_grow*

void *arg_grow*(*arg_List arg*, **const char** **string*, **int** *length*)

Grow the next item of *arg* with *length* characters of *string*. Several calls to *arg_grow* should be followed by a single call to *arg_finish* without any intervening calls to other functions which modify *arg*.

1.2.6 *arg_finish*

arg_List arg_finish(*arg_List arg*)

Finish the growth of the next item in *arg* started by *arg_grow*.

1.2.7 *arg_get*

const char ∗*arg_get*(*arg_List arg*, **int** *item*)

Return *item* from *arg*. *arg* is 0-based.

1.2.8 *arg_count*

int *arg_count*(*arg_List arg*)

Return the number of items in *arg*.

1.2.9 *arg_get_vector*

void *arg_get_vector*(*arg_List arg*, **int** ∗*argc*, **char** ∗∗∗*argv*)

Get an *argc* and *argv* from *arg*. These are suitable for use in calls to *exec*. The *argc*+1 item in *argv* is **NULL**.

1.2.10 *arg_argify*

arg_List arg_argify(**const char** ∗*string*, **int** *quoteStyle*)

Break up *string* into arguments, placing them as items in *arg*. Items within single or double quotes may contain spaces. The quotes are stripped as in shell argument processing. In this version, backslash is **not** a special quoting character.

1.3 Base-26 Routines

These routines perform encoding and decodingusing the 26 lowercase characters, a-z. This is sometimes useful for generating unique identifiers that do not contain numbers.

1.4 Base-64 Routines

These routines use the 64-character subset of International Alphabet IA5 discussed in RFC 1421 (printeable encoding) and RFC 1522 (base64 MIME).

Value Encoding Value Encoding Value Encoding Value Encoding 0 A 17 R 34 i 51 z 1 B 18 S 35 j 52 0 2 C 19 T 36 k 53 1 3 D 20 U 37 l 54 2 4 E 21 V 38 m 55 3 5 F 22 W 39 n 56 4 6 G 23 X 40 o 57 5 7 H 24 Y 41 p 58 6 8 I 25 Z 42 q 59 7 9 J 26 a 43 r 60 8 10 K 27 b 44 s 61 9 11 L 28 c 45 t 62 + 12 M 29 d 46 u 63 / 13 N 30 e 47 v 14 O 31 f 48 w (pad) = 15 P 32 g 49 x 16 Q 33 h 50 y

1.5 Bit manipulation routines

These are a set of simple routine to manipulate bits in an integer.

1.5.1 *bit_set*

__inline__ **void** *bit_set*(**unsigned long** **flags*, **int** *bit*)

Set *bit* in *flags*.

1.5.2 *bit_clr*

__inline__ **void** *bit_clr*(**unsigned long** **flags*, **int** *bit*)

Clear *bit* in *flags*.

1.5.3 *bit_tst*

__inline__ **int** *bit_tst*(**unsigned long** **flags*, **int** *bit*)

Test *bit* in *flags*, returning non-zero if the bit is set and zero if the bit is clear.

1.5.4 *bit_cnt*

__inline__ **int** *bit_cnt*(**unsigned long** **flags*)

Return a count of the number of bits set in *flags*.

1.6 Debugging Support

These routines provide low-level support for run-time debugging messages. A set of global flags are maintained using **#define** statements. These flags are assumed to be 32-bit integers, the top two bits of which are used to select one of four sets of debugging lags. Each set, therefore, has 30 bits of flag information. The last set (i.e., when the top two bits are both set) is reserved for internal use by the KHEPERA library. For convenience, each flag can be given a unique name, so that flags can be set easily with command-line options.

1.6.1 *dbg_register*

void *dbg_register*(*dbg_Type flag*, **const char** **name*)

dbg_register is used to set up an asociated between a *flag* and a *name*. After this association is made, calls to *dbg_set* can use *name* to set the global debugging flag. Both of the high bits cannot be set simultaneously—these flags are reserved for internal use by the KHEPERA library.[1]

[1] For internal KHEPERA library use, *_dbg_register* can be used to register flags which have both high bits set.

1.6.2 *dbg_set*

void *dbg_set*(**const char** **name*)

dbg_set sets the *name* flag. If *name* is "none," then all flags are cleared.

1.6.3 *dbg_set_flag*

void *dbg_set_flag*(*dbg_Type flag*)

Used to set the flag using the predefined macro instead of the string name.

1.6.4 *dbg_unset_flag*

void *dbg_unset_flag*(*dbg_Type flag*)

Used to unset the flag using the predefined macro instead of the string name.

1.6.5 *dbg_test*

__inline__ **int** *dbg_test*(*dbg_Type flag*)

Thsi inlined function tests the *flag*, returning non-zero if the *flag* is set, and zero otherwise.

1.6.6 *dbg_destroy*

void *dbg_destroy*(**void**)

dbg_destroy destroys the memory associated with the debugging support routines. This routine should *never* be called by the programmer: it is automatically called at program termination on systems that support the *atexit* or *on_exit* calls.

1.7 Error Reporting Routines

Several error reporting routines are provided. These routines are always used to print errors generated within the KHEPERA library, and are available for the programmer as well.

1.7.1 *err_set_program_name*

void *err_set_program_name*(**const char** **programName*)

err_set_program_name records the value of *argv[0]* for the calling program. If this value is not **NULL**, then it will be used when printing errors and warnings.

1.7.2 *err_program_name*

const char **err_program_name*(**void**)

err_program_name returns the value of *programName* that was previously set with *err_set_program_name*. This value may be **NULL**.

1.7.3 *err_fatal*

void *err_fatal*(**const char** **routine*, **const char** **format*, ...)

err_fatal flushes **stdout**, prints a fatal error report on **stderr**, flushes **stderr** and **stdout**, and calls *exit*. *routine* is the name of the routine in which the error took place.

1.7.4 *err_fatal_errno*

void *err_fatal_errno*(**const char** **routine*, **const char** **format*, ...)

err_fatal_errno flushes **stdout**, prints a fatal error report on **stderr**, prints the system error corresponding to *errno*, flushes **stderr** and **stdout**, and calls *exit*. *routine* is the name of the routine in which the error took place.

1.7.5 *err_warning*

void *err_warning*(**const char** **routine*, **const char** **format*, ...)

err_warning flushes **stdout**, prints a non-fatal warning on **stderr**, and returns to the caller. *routine* is the name of the calling routine.

1.7.6 *err_internal*

void *err_internal*(**const char** **routine*, **const char** **format*, ...)

err_internal flushes **stdout**, prints the fatal error message, flushes **stderr** and **stdout**, and calls *abort* so that a core dump is generated.

1.8 Flag Support

These routines provide low-level support for run-time program flow control. The mechanism used is similar to that used for debugging messages. A set of global flags are maintained using **#define** statements. These flags are assumed to be 32-bit integers, the top two bits of which are used to select one of four sets of debugging lags. Each set, therefore, has 30 bits of flag information. For convenience, each flag can be given a unique name, so that flags can be set easily with command-line options.

1.8.1 *flg_register*

void *flg_register*(*flg_Type flag*, **const char** **name*)

flg_register is used to set up an asociated between a *flag* and a *name*. After this association is made, calls to *flg_set* can use *name* to set the global flag.

1.8.2 *flg_set*

void *flg_set*(**const char** **name*)

flg_set sets the *name* flag. If *name* is "none," then all flags are cleared.

1.8.3 *flg_test*

__inline__ **int** *flg_test*(*flg_Type flag*)

Thsi inlined function tests the *flag*, returning non-zero if the *flag* is set, and zero otherwise.

1.8.4 *flg_destroy*

void *flg_destroy*(**void**)

flg_destroy destroys the memory associated with the flag support routines. This routine should *never* be called by the programmer: it is automatically called at program termination on systems that support the *atexit* or *on_exit* calls.

1.9 Hash Table Routines

Generic hash table support is provided for storing generic data associated with keys. The hash table has prime length, with self-organizing linked lists [Knuth73, pp. 398–9] used for collision resolution. The hash table automatically grows as necessary to preserve efficient access.

1.9.1 *hsh_create*

hsh_HashTable hsh_create(**unsigned long** (∗*hash*)(**const void** ∗),
int (∗*compare*)(**const void** ∗,
const void ∗))

The *hsh_create* function initilizes a generic hash table. Keys and data are pointers to **void**.

The internal representation of the hash table will grow automatically when an insertion is performed and the table is more than half full.

The *hash* function should take a pointer to a *key* and return an **unsigned int**. If *hash* is **NULL**, then the *key* is assumed to be a pointer to a null-terminated string, and the function shown in Figure 1 will be used for *hash* (the algorithm for this function is from [Aho88, p. 435]).

The *compare* function should take a pair of pointers to keys and return zero if the keys are equal and non-zero if the keys are not equal. If *compare* is **NULL**, then the keys are assumed to point to null-terminated strings, and the *strcmp* function will be used for *compare*.

Additionally, the *hsh_pointer_hash* and *hsh_pointer_compare* functions are available and can be used to treat the *value* of the **void** pointer as the key. These functions are often useful for maintaining sets of objects.

1.9.2 *hsh_destroy*

void *hsh_destroy*(*hsh_HashTable table*)

hsh_destroy frees all of the memory associated with the hash table.

```
unsigned long hsh_string_hash( const void *key )
{
    const char          *pt = (const char *)key;
    unsigned long        h  = 0;
    static const char   *prev_pt = NULL;
    static unsigned long prev_h = 0;

    if (!pt)
        err_internal( __FUNCTION__, "String-valued keys may not be NULL\n" );

    if (prev_pt == pt) return prev_h;

    while (*pt) {
        h += *pt++;
#if 0
        h *= 65599L;                    /* prime near 2^16 */
#else
        h *= 2654435789U;               /* prime near (sqrt(5)-1)/2 2^32 */
#endif
    }
```

Figure 1: *hsh_string_hash*

The memory used by keys and data is *not* freed—this memory is the responsibility of the user. However, a call to *hsh_iterate* can be used to free this memory *immediately* before a call to *hsh_destroy*.

1.9.3 *hsh_insert*

int *hsh_insert*(*hsh_HashTable table*,
const void **key*,
const void **datum*)

hsh_insert inserts a new *key* into the *table*. If the insertion is successful, zero is returned. If the *key* already exists, 1 is returned. Hence, the way to change the *datum* associated with a *key* is first to call *hsh_delete*.

If the internal representation of the hash table becomes more than half full, its size is increased automatically. At present, this requires that all of the key pointers are copied into a new table. Rehashing is not required, however, since the hash values are stored for each key.

1.9.4 *hsh_delete*

int *hsh_delete*(*hsh_HashTable table*, **const void** **key*)

hsh_delete removes a *key* and the associated datum from the *table*. Zero is returned if the *key* was present. Otherwise, 1 is returned.

1.9.5 *hsh_retrieve*

const void **hsh_retrieve*(*hsh_HashTable table*,
const void **key*)

hsh_retrieve retrieves the datum associated with a *key*. If the *key* is not present in the *table*, then **NULL** is returned.

1.9.6 *hsh_iterate*

int *hsh_iterate*(*hsh_HashTable table*,
int (**iterator*)(**const void** **key*,
const void **datum*))

hsh_iterate is used to iterate a function over every value in the *table*. The function, *iterator*, is passed the *key* and *datum* pair for each entry in the table. If *iterator* returns a non-zero value, the iterations stop, and *hsh_iterate* returns non-zero. Note that the keys are in some arbitrary order, and that this order may change between two successive calls to *hsh_iterate*.

1.9.7 *hsh_iterate_arg*

int *hsh_iterate_arg*(*hsh_HashTable table*,
int (**iterator*)(**const void** **key*,

```
                                        const void *datum,
                                        void *arg ),
                    void *arg )
```

hsh_iterate_arg is used to iterate a function over every value in the *table*. The function, *iterator*, is passed the *key* and *datum* pair for each entry in the table. If *iterator* returns a non-zero value, the iterations stop, and *hsh_iterate* returns non-zero. Note that the keys are in some arbitrary order, and that this order may change between two successive calls to *hsh_iterate*.

1.9.8 *hsh_get_stats*

hsh_Stats hsh_get_stats(hsh_HashTable table)

hsh_get_stats returns statistics about the *table*. The *hsh_Stats* structure is shown in Figure 2.

```
typedef struct hsh_Stats {
    unsigned long size;             /* Size of table */
    unsigned long resizings;        /* Number of resizings */
    unsigned long entries;          /* Total entries in table */
    unsigned long buckets_used;     /* Number of hash buckets in use */
    unsigned long singletons;       /* Number of length one lists */
    unsigned long maximum_length;   /* Maximum list length */

    unsigned long retrievals;       /* Total number of retrievals */
    unsigned long hits;             /* Number of retrievals from top of a list */
    unsigned long misses;           /* Number of unsuccessful retrievals */
} *hsh_Stats;
```

Figure 2: *hsh_Stats*

1.9.9 *hsh_print_stats*

void *hsh_print_stats(hsh_HashTable table, FILE *stream)*

hsh_print_stats prints the statistics for *table* on the specified *stream*. If *stream* is **NULL**, then **stdout** will be used.

1.9.10 *hsh_init_position*

hsh_Position hsh_init_position(hsh_HashTable table)

hsh_init_position returns a position marker for some arbitary first element in the table. This marker can be used with *hsh_next_position* and *hsh_get_position*.

1.9.11 *hsh_next_position*

hsh_Position hsh_next_position(*hsh_HashTable table*, *hsh_Position position*)

hsh_next_position returns a position marker for the next element in the table. Elements are in arbitrary order based on their positions in the hash table.

1.9.12 *hsh_get_position*

void ∗*hsh_get_position*(*hsh_Position position*, **void** ∗∗*key*)

hsh_get_position returns the datum associated with the *position* marker, or **NULL** if there is no such datum. *key* is set to the key associated with this datum, or **NULL** is there is no such datum.

1.9.13 *hsh_readonly*

int *hsh_readonly*(*hsh_HashTable table*, **int** *flag*)

hsh_readonly sets the *readonly* flag for the *table* to *flag*. *flag* should be 0 or 1. The value of the previous flag is returned. When a hash table is marked as readonly, self-organization of the bucket-overflow lists will not take place, and any attempt to modify the list (e.g., insertion or deletion) will result in an error.

1.10 List Routines

The list routines provide support for a general linked list containting objects that are pointers to **void**. The list can be viewed as a stack or queue – data can be added to the head or the tail, but can only be removed from the head (this data structure is sometimes called a "qstack"). (If only a stack is needed, the stack routines are more efficient.)

1.10.1 *lst_create*

lst_List lst_create(**void**)

lst_create initializes a list object.

1.10.2 *lst_destroy*

void *lst_destroy*(*lst_List list*)

lst_destroy destroys all memory associated with the *list*. The memory used by data is *not* freed—this memory is the responsibility of the user.

1.10.3 *lst_append*

void *lst_append*(*lst_List list*, **const void** ∗*datum*)

lst_append appends *datum* on the *list*.

1.10.4 *lst_push*

void *lst_push*(*lst_List list*, **const void** **datum*)

lst_push prepends *datum* on the *list*.

1.10.5 *lst_pop*

void **lst_pop*(*lst_List list*)

lst_pop removes the first datum on the *list* and returns the pointer. If the *list* is empty, *lst_pop* returns **NULL**.

1.10.6 *lst_top*

void **lst_top*(*lst_List list*)

lst_top returns a pointer to the datum that is the first element of the *list*, but does *not* remove this datum from the *list*. If the *list* is empty, *lst_top* returns **NULL**.

1.10.7 *lst_nth_get*

void **lst_nth_get*(*lst_List list*, **unsigned int** *n*)

lst_nth_get returns a pointer to the n-th datum in the *list*, or **NULL** if the nth element does not exist.

1.10.8 *lst_nth_set*

void *lst_nth_set*(*lst_List list*, **unsigned int** *n*, **const void** **datum*)

lst_nth_set locates the n-th datum in the *list* and replaces that datum with *datum*. If the nth element does not exist, the program will halt with an error. (I.e., it is the programmer's responsibility to check *lst_length* and only pass valid values of n.)

1.10.9 *lst_member*

int *lst_member*(*lst_List list*, **const void** **datum*)

lst_member returns non-zero if the pointer to *datum* is also a pointer on the list, and zero otherwise. Note that only pointers are compared, so identical copies of data structures will be viewed as non-equal.

1.10.10 *lst_length*

unsigned int *lst_length*(*lst_List list*)

lst_length returns the number of elements in the list.

1.10.11 *lst_truncate*

void *lst_truncate*(*lst_List list*, **unsigned int** *length*)

lst_truncate truncates a list to *length* elements. If the list is not longer than *length*, nothing it done.

1.10.12 *lst_truncate_position*

void *lst_truncate_position*(*lst_List list*, *lst_Position position*)

lst_truncate_position truncates a list beyond *position* (i.e., *position* is always left in the list. If *postition* is **NULL**, then the list is emptied. This convention is useful when using *lst_last_postition* to get a marker allowing an older state of a list to be restored.

1.10.13 *lst_iterate*

int *lst_iterate*(*lst_List list*, **int** (∗*iterator*)(**const void** ∗*datum*))

lst_iterate is used to iterate a function over every element in the *list*. The function, *iterator*, is passed a pointer to each element. If *iterator* returns a non-zero value, the iterations stop, and *lst_iterate* returns.

1.10.14 *lst_iterate_arg*

int *lst_iterate_arg*(*lst_List list*,
int (∗*iterator*)(**const void** ∗*datum*, **void** ∗*arg*),
void ∗*arg*)

lst_iterate_arg is used to iterate a function over every element in the *list*. The function, *iterator*, is passed a pointer to each element. If *iterator* returns a non-zero value, the iterations stop, and *lst_iterate* returns.

1.10.15 *lst_init_position*

lst_Position lst_init_position(*lst_List list*)

lst_init_position returns a position marker for the head of the list. This marker can be used with *lst_next_position* and *lst_get_position*.

1.10.16 *lst_last_position*

lst_Position lst_last_position(*lst_List list*)

lst_last_position returns a position marker for the tail of the list. This marker can be used with *lst_truncate_position* to restore a previous state of the list.

1.10.17 *lst_next_position*

lst_Position lst_next_position(*lst_Position position*)

lst_next_position returns a position marker for the element after the element marked by *position*, or **NULL** if *position* is the last element in the list.

1.10.18 *lst_nth_position*

lst_Position lst_nth_position(*lst_List list*, **unsigned int** *n*)

lst_nth_position returns a position marker for the *n*th element in the list, or **NULL** if the *n*th element does not exist.

1.10.19 *lst_get_position*

void ∗*lst_get_position*(*lst_Position position*)

lst_get_position returns the datum associated with the *position* marker, or **NULL** if there is no such element.

1.10.20 *lst_set_position*

void *lst_set_position*(*lst_Position position*, **const void** ∗*datum*)

lst_set_position sets the *datum* associated with the *position* marker.

1.10.21 *_lst_dump_node*

static int *_lst_dump_node*(**const void** ∗*datum*)

lst_dump prints each *datum* on the list in hex

1.11 Memory Management Routines

The memory management routines provide simple support for string object storage. These routines are generally used as building blocks by other parts of the KHEPERA library (e.g., string pools and abstract syntax trees).

1.11.1 *mem_create_strings*

mem_String mem_create_strings(**void**)

mem_create_strings creates a memory object for storing strings.

1.11.2 *mem_destroy_strings*

void *mem_destroy_strings*(*mem_String info*)

mem_destroy_strings destroys the memory object returned from *mem_create_strings*. All memory if freed, including that used for the strings. Therefore, any pointers to strings in the table will be left dangling.

1.11.3 *mem_strcpy*

const char **mem_strcpy*(*mem_String info*, **const char** **string*)

mem_strcpy copies a *string* into the memory object pointed to by *info*.

1.11.4 *mem_strncpy*

const char **mem_strncpy*(*mem_String info*,
const char **string*, **int** *len*)

mem_strncpy copies *len* bytes of *string* into the memory object pointed to by *info*. A null is added to the end of the copied sequence.

1.11.5 *mem_grow*

void *mem_grow*(*mem_String info*, **const char** **string*, **int** *len*)

mem_grow copies *len* of *string* onto the top of the memory object pointed to by *info*. Several calls to *mem_grow* should be followed by a single call to *mem_finish* without any intervening calls to other functions which modify *info*.

1.11.6 *mem_finish*

const char **mem_finish*(*mem_String info*)

mem_finish finishes the growth of the object performed by *mem_grow*.

1.11.7 *mem_get_string_stats*

mem_StringStats mem_get_string_stats(*mem_String info*)

mem_get_string_stats returns statistics about the memory object pointed to by *info*. The *mem_StringStats* structure is shown in Figure 3.

1.11.8 *mem_print_string_stats*

void *mem_print_string_stats*(*mem_String info*, *FILE* **stream*)

mem_print_string_stats prints the statistics for the memory object pointed to by *info* on the specified *stream*. If *stream* is **NULL**, then **stdout** will be used.

```
typedef struct mem_StringStats {
    int count;                          /* Number of strings or objects */
    int bytes;                          /* Number of bytes allocated */
} *mem_StringStats;
```

Figure 3: *mem_StringStats*

1.11.9 *mem_create_objects*

mem_Object mem_create_objects(**int** *size*)

mem_create_objects creates a memory storage object for object of *size* bytes.

1.11.10 *mem_destroy_objects*

void *mem_destroy_objects*(*mem_Object info*)

mem_destroy_objects destroys the memory object returned from *mem_create_objects*. All memory if freed, including that used for the object. Therefore, any pointers to objects stored by *info* will be left dangling.

1.11.11 *mem_get_object*

void **mem_get_object*(*mem_Object info*)

mem_get_object returns a pointer to a block of memory which is *size* bytes long (as specified in the call to *mem_create_objects*). This block is either newly allocated memory, or is memory which was previously allocated by *mem_get_object* and subsequently freed by *mem_free_object*.

1.11.12 *mem_get_empty_object*

void **mem_get_empty_object*(*mem_Object info*)

mem_get_empty_object is exactly like *mem_get_object*, except the memory associated with the object is set to all zeros.

1.11.13 *mem_free_object*

void *mem_free_object*(*mem_Object info*, **void** **obj*)

mem_free_object "frees" the object, *obj*, which was previously obtained from *mem_get_object*. The memory associated with the object is not actually freed, but the object pointer is stored on a stack, and is available for subsequent calls to *mem_get_object*.

1.11.14 *mem_get_object_stats*

mem_ObjectStats mem_get_object_stats(*mem_Object info*)

mem_get_object_stats returns statistics about the memory object pointed to by *info*. The *mem_ObjectStats* structure is shown in Figure 4.

```
typedef struct mem_ObjectStats {
    int total;                  /* Total objects requested */
    int used;                   /* Total currently in use */
    int reused;                 /* Total reused */
    int size;                   /* Size of each object */
} *mem_ObjectStats;
```

Figure 4: *mem_ObjectStats*

1.11.15 *mem_print_object_stats*

void *mem_print_object_stats*(*mem_Object info, FILE *stream*)

mem_print_object_stats prints the statistics for the memory object pointed to by *info* on the specified *stream*. If *stream* is **NULL**, then **stdout** will be used.

1.12 Parsing (and Lexing) Support

1.12.1 *prs_set_debug*

void *prs_set_debug*(**int** *debug_flag*)

prs_set_debug specifies the value of *yyerror* that *prs_file* will use.

1.12.2 *prs_set_cpp_options*

void *prs_set_cpp_options*(**const char** **cpp_options*)

prs_set_cpp_options sets the options for *cpp* to *cpp_options*, ensuring that *prs_file* will use *cpp* as a filter. If *cpp_options* is **NULL**, then *cpp* will not be used at a filter.

1.12.3 *prs_file*

void *prs_file*(**const char** **filename*)

prs_file calls opens *filename* for input, sets *yyerror* to the value specified by *prs_set_debug*, and calls *yyparse*, perhaps using *cpp* as an input filter.

A similar function should deal with multiple parsers in the same program, but this has not been implemented. Also, either this function or another function should start an interactive parse session.

1.12.4 *prs_file_nocpp*

void *prs_file_nocpp*(**const char** **filename*)

prs_file_nocpp calls opens *filename* for input, sets *yyerror* to the value specified by *prs_set_debug*, and calls *yyparse*.

A similar function should deal with multiple parsers in the same program, but this has not been implemented. Also, either this function or another function should start an interactive parse session.

1.12.5 *prs_stream*

void *prs_stream*(*FILE* **str*, **const char** **name*)

prs_stream parses an already opened stream called *name*.

1.12.6 *prs_make_integer*

int *prs_make_integer*(**const char** **string*, **int** *length*)

prs_make_integer converts a *string* of specified *length* to an integer. This function is useful in scanners that do not **NULL**-terminate *yytext*.

1.12.7 *prs_make_double*

double *prs_make_double*(**const char** **string*, **int** *length*)

prs_make_double converts a *string* of specified *length* to a double. This function is useful in scanners that do not **NULL**-terminate *yytext*.

1.13 Process Management Routines

The process management routines are designed to facilitate the creation and management of child processes, coprocesses, and associated pipelines and I/O. Some support for daemons and socket connections is also provided.

1.14 Prime Number Routines

These routines are used to find the next larger prime number for expansion of the hash tables used by the hash and set routines. These routines are only useful for 32-bit unsigned values.

1.14.1 *prm_is_prime*

int *prm_is_prime*(**unsigned int** *value*)

prm_is_prime returns 1 if *value* is prime; 0 otherwise. Primality is determined by testings all odd divisors less than the square root of *value*. For 32-bit integers, this may mean that we will test about $\frac{\sqrt{2^{32}-1}}{2} = 32768$ odd values instead of the 6542 primes that would actually

need to be tested. (A table of pre-computed primes using less than 26kB of memory could be used to recover this factor of 5 performance loss.)

1.14.2 *prm_next_prime*

unsigned long *prm_next_prime*(**unsigned int** *start*)

prm_next_prime returns the smallest odd prime greater than or equal to *start*.

1.15 Set Routines

The set implementation is similar to the hash table implementation, except that the set object does not associate a *datum* with a *key*. For sets, keys are called elements. All of the hash table functions are supported, with the addition of a membership test and several set operations.

The underlying data structure is a hash table of prime length, with self-organizing linked lists [Knuth73, pp. 398–9] used for collision resolution. The hash table automatically grows as necessary to preserve efficient access.

1.15.1 *set_create*

set_Set set_create(*set_HashFunction hash*, *set_CompareFunction compare*)

The *set_create* function initilizes a set object. Elements are pointers to **void**.

The internal representation of the set will grow automatically when an insertion is performed and the table is more than half full.

The *hash* function should take a pointer to a *elem* and return an **unsigned int**. If *hash* is **NULL**, then the *elem* is assumed to be a pointer to a null-terminated string, and the function shown in Figure 1 will be used for *hash*.

The *compare* function should take a pair of pointers to elements and return zero if the elements are the same and non-zero if they are different. If *compare* is **NULL**, then the elements are assumed to point to null-terminated strings, and the *strcmp* function will be used for *compare*.

Additionally, the *hsh_pointer_hash* and *hsh_pointer_compare* functions are available and can be used to treat the *value* of the **void** pointer as the element. These functions are often useful for maintaining sets of objects.

1.15.2 *set_destroy*

void *set_destroy*(*set_Set set*)

set_destroy frees all of the memory associated with the set object.

The memory used by elements is *not* freed—this memory is the responsibility of the user. However, a call to *set_iterate* can be used to free this memory *immediately* before a call to *set_destroy*.

1.15.3 *set_insert*

int *set_insert*(*set_Set set*, **const void** ∗*elem*)

set_insert inserts a new *elem* into the *set*. If the insertion is successful, zero is returned. If the *elem* already exists, 1 is returned.

If the internal representation of the set becomes more than half full, its size is increased automatically. At present, this requires that all of the element pointers are copied into a new set. Rehashing is not required, however, since the hash values are stored for each element.

1.15.4 *set_delete*

int *set_delete*(*set_Set set*, **const void** ∗*elem*)

set_delete removes an *elem* from the *set*. Zero is returned if the *elem* was present. Otherwise, 1 is returned.

1.15.5 *set_member*

int *set_member*(*set_Set set*, **const void** ∗*elem*)

set_member returns 1 if *elem* is in *set*. Otherwise, zero is returned.

1.15.6 *set_iterate*

int *set_iterate*(*set_Set set*,
int (∗*iterator*)(**const void** ∗*elem*))

set_iterate is used to iterate a function over every *elem* in the *set*. The function, *iterator*, is passed each *elem*. If *iterator* returns a non-zero value, the iterations stop, and *set_iterate* returns. Note that the elements are in some arbitrary order, and that this order may change between two successive calls to *set_iterate*.

1.15.7 *set_iterate_arg*

int *set_iterate_arg*(*set_Set set*,
int (∗*iterator*)(**const void** ∗*elem*, **void** ∗*arg*),
void ∗*arg*)

set_iterate_arg is used to iterate a function over every *elem* in the *set*. The function, *iterator*, is passed each *elem*. If *iterator* returns a non-zero value, the iterations stop, and *set_iterate* returns. Note that the elements are in some arbitrary order, and that this order may change between two successive calls to *set_iterate*.

1.15.8 *set_init_position*

set_Position set_init_position(*set_Set set*)

set_init_position returns a position marker for some arbitary first element in the set. This marker can be used with *set_next_position* and *set_get_position*.

1.15.9 *set_next_position*

set_Position set_next_position(set_Set set, set_Position position)

set_next_position returns a position marker for the next element in the set. Elements are in arbitrary order based on their positions in the hash table.

1.15.10 *set_get_position*

void **set_get_position(set_Position position)*

set_get_position returns the element associated with the *position* marker, or **NULL** if there is no such element.

1.15.11 *set_readonly*

int *set_readonly(set_Set set,* **int** *flag)*

set_readonly sets the *readonly* flag for the *set* to *flag*. *flag* should be 0 or 1. The value of the previous flag is returned. When a set is marked as readonly, self-organization of the bucket-overflow lists will not take place, and any attempt to modify the list (e.g., insertion or deletion) will result in an error.

1.15.12 *set_union*

set_Set set_union(set_Set set1, set_Set set2)

set_union returns a new set which is the union of *set1* and *set2*. Only pointers to elements are copied, *not* the data pointed (this has memory management implications). The *hash* and *compare* functions must be identical for the two sets.

1.15.13 *set_inter*

set_Set set_inter(set_Set set1, set_Set set2)

set_inter returns a new set which is the intersection of *set1* and *set2*. Only pointers to elements are copied, *not* the data pointed (this has memory management implications). The *hash* and *compare* functions must be identical for the two sets.

1.15.14 *set_diff*

set_Set set_diff(set_Set set1, set_Set set2)

set_diff returns a new set which is the difference resulting from removing every element in *set2* from the elements in *set1*. Only pointers to elements are copied, *not* the data pointed

(this has memory management implications). The *hash* and *compare* functions must be identical for the two sets.

1.15.15 *set_equal*

int *set_equal*(*set_Set set1*, *set_Set set2*)

set_equal returns non-zero if *set1* and *set2* contain the same number of elements, and all of the elements in *set1* are also in *set2*. The *hash* and *compare* functions must be identical for the two sets.

1.15.16 *set_get_stats*

set_Stats set_get_stats(*set_Set set*)

set_get_stats returns statistics about the *set*. The *set_Stats* structure is shown in Figure 5.

```
typedef struct set_Stats{
    unsigned long size;             /* Size of table */
    unsigned long resizings;        /* Number of resizings */
    unsigned long entries;          /* Total entries in table */
    unsigned long buckets_used;     /* Number of hash buckets in use */
    unsigned long singletons;       /* Number of length one lists */
    unsigned long maximum_length;   /* Maximum list length */

    unsigned long retrievals;       /* Total number of retrievals */
    unsigned long hits;             /* Number of retrievals from top of a list */
    unsigned long misses;           /* Number of unsuccessful retrievals */
} *set_Stats;
```

Figure 5: *set_Stats*

1.15.17 *set_print_stats*

void *set_print_stats*(*set_Set set*, *FILE *stream*)

set_print_stats prints the statistics for *set* on the specified *stream*. If *stream* is **NULL**, then **stdout** will be used.

1.16 Skip List Routines

Skip list support is provided as an alternative to balanced trees. Skip lists have the advantage that an inorder walk through the list is possible in the face of additions and deletions from the list. Balanced tree, algorithms, in contrast, make this sort of traversal impossible because of the rotations that are necessary for the balancing.

For these lists, the *key* is derivable from the *datum* that is stored in the list. This makes it possible for the actualy keys to change, as long as the ordering of the data stay the same. This is essential for the use of skip lists for KHEPERA trees.

This code is derived from [Pugh90] and from a skip list implementation by Lars Nyland.

1.16.1 *sl_create*

sl_List sl_create(**int** (∗*compare*)(**const void** ∗*key1*, **const void** ∗*key2*),
const void ∗(∗*key*)(**const void** ∗*datum*),
const char ∗(∗*print*)(**const void** ∗*datum*))

sl_create initializes a skip list. The *compare* function returns -1, 0, or 1 depending on the ordering of *key1* and *key2*. The *key* function converts a *datum* into a *key*. The *print* function returns a string representation of *datum*, and is allowed to always return a pointer to the same static buffer.

compare must be provided. If *key* is not provided, then *datum* will be used as the key. If *print* is not provided, then the *datum* pointer will be printed when necessary.

1.16.2 *sl_destroy*

void *sl_destroy*(*sl_List list*)

sl_destroy removes all of the memory associated with the maintenance of the specified skip *list*. The pointer to the user-defined *datum* is **not** freed – this is the responsibility of the user.

1.16.3 *_sl_shutdown*

void *_sl_shutdown*(**void**)

_sl_shutdown is used to free the internal data structures used by the skip list package. Since it is called automatically by LIBMAA, it should not be called explicitly by the user.

1.16.4 *sl_insert*

void *sl_insert*(*sl_List list*, **const void** ∗*datum*)

Insert *datum* into *list*.

1.16.5 *sl_delete*

void *sl_delete*(*sl_List list*, **const void** ∗*datum*)

Delete *datum* from *list*.

1.16.6 *sl_find*

const void **sl_find*(*sl_List list*, **const void** **key*)

Find the datum in *list* that has an associated value of *key*. Return that datum (a pointer), or **NULL** if the *key* is not found.

1.16.7 *sl_iterate*

int *sl_iterate*(*sl_List list*, *sl_Iterator f*)

Iterate *f* over every datum in *list*. If *f* returns non-zero, then abort the remainder of the iteration. Iterations are designed to do something appropriate in the face of arbitrary insertions and deletions performed by *f*.

1.16.8 *sl_iterate_arg*

int *sl_iterate_arg*(*sl_List list*, *sl_IteratorArg f*, **void** **arg*)

Iterate *f* over every datum in *list*. If *f* returns non-zero, then abort the remainder of the iteration. Iterations are designed to do something appropriate in the face of arbitrary insertions and deletions performed by *f*.

1.16.9 *_sl_dump*

void *_sl_dump*(*sl_List list*)

Dump the internal data structures associated with *list*. This is purely for debugging.

1.17 Source Code Management

Most of the source code management routines are designed to be called from within a lexical analyzer. The routines support the storage and retrieval of source code lines and source code token information. A single, global, source code management object is supported.

This documentation is incomplete.
These routines have not been fully implemented.

1.17.1 *src_create*

void *src_create*(**void**)

The *src_create* functions initializes the global source code management object. This routine should only be called once, if at all. If the other routines are used as expected, *src_create* will be called automatically when it is needed.

1.17.2 *src_destroy*

void *src_destroy*(**void**)

src_destroy frees all memory associated with the global source code management object.

1.17.3 *src_line*

const char **src_line*(**const char** **line*, **int** *len*)

src_line stores a *line* of length *len*.

1.17.4 *src_new_file*

void *src_new_file*(**const char** **filename*)

src_new_file specifies that *filename* is the name of the current file being read by the lexer.

1.17.5 *src_new_line*

void *src_new_line*(**int** *line*)

src_new_line specifies that *line* is the number of the current line being read by the lexer. Line numbers start at 1.

1.17.6 *src_advance*

void *src_advance*(**int** *length*)

src_advance is used to advance the token offset pointer *length* bytes.

1.17.7 *src_get*

src_Type src_get(**int** *length*)

src_get is used to get a *src_Type* object for the current token. This object contains file, line, offset, and length information which can be used to print parse error messages and provide detailed token tracking.

1.17.8 *src_filename*

const char **src_filename*(*src_Type source*)

src_filename returns the *file* associated with the specified *source* information.

1.17.9 *src_linenumber*

int *src_linenumber*(*src_Type source*)

src_linenumber returns the *line* associated with the specified *source* information.

1.17.10 *src_offset*

int *src_offset*(*src_Type source*)

src_offset returns the token *offset* associated with the specified *source* information.

1.17.11 *src_length*

int *src_length*(*src_Type source*)

src_length returns the token *length* associated with the specified *source* information.

1.17.12 *src_source_line*

const char **src_source_line*(*src_Type source*)

src_source_line returns the full source line associated with the specified *source* information.

1.17.13 *src_get_stats*

src_Stats src_get_stats(**void**)

src_get_stats returns the statistics associated with the source code manager. The *src_Stats* structure is shown in Figure 6.

```
typedef struct src_Stats {
    int lines_used;                 /* Lines used */
    int lines_allocated;            /* Lines allocated */
    int lines_bytes;                /* Bytes required to store lines */
    int tokens_total;               /* Tokens used */
    int tokens_reused;              /* Tokens reused */
    int tokens_size;                /* Size of a token information object */
} *src_Stats;
```

Figure 6: *src_Stats*

1.17.14 *src_print_stats*

void *src_print_stats*(*FILE *stream*)

src_print_stats prints the statistics about the source code manager to the specified *stream*. If *stream* is **NULL**, then **stdout** will be used.

1.17.15 *src_print_line*

void *src_print_line*(*FILE *stream, src_Type source*)

src_print_line prints a line of source code, as described by *source* to the specified *stream*. If *stream* is **NULL**, then **stdout** will be used.

1.17.16 *src_parse_error*

void *src_parse_error*(*FILE* ∗*stream*, *src_Type source*, **const char** ∗*message*)

src_parse_error will print the error *message* to the specified *stream*, followed by the offending line of source code, as specified by *source*. If *stream* is **NULL**, then **stdout** will be used.

It is assumed that *message* has the format of *yyerror* so that massaging of the string can be performed to make it more readable (token names are assumed to start with "T_" and will be changed to more readable names). This massaging should be done by a user-defined function, since it is relatively specific to this author's coding conventions.

1.17.17 *src_print_error*

void *src_print_error*(*FILE* ∗*str*, *src_Type source*, **const char** ∗*format*, ...)

src_print_error will print an arbitrary error, specified as for *printf* by the *format* variable, to the specified *stream*, followed by the line of source code specified by *source*. If *stream* is **NULL**, then **stdout** will be used.

1.17.18 *src_print_message*

void *src_print_message*(*FILE* ∗*str*, *src_Type source*, **const char** ∗*format*, ...)

src_print_message will just print a message labeled with a line.

1.18 Stack Routines

The stack routines provide support for a general stack of pointers to **void**. Because of the simplicity of the stack object, no statistics are maintained. (Althought the list routines can also be used as a stack, the stack implemented here is more efficient.)

1.18.1 *stk_create*

stk_Stack stk_create(**void**)

stk_create initializes a stack object.

1.18.2 *stk_destroy*

void *stk_destroy*(*stk_Stack stack*)

stk_destroy destroys all memory associated with the *stack*. The memory used by data is *not* freed—this memory is the responsibility of the user.

1.18.3 *stk_push*

void *stk_push*(*stk_Stack stack*, **void** ∗*datum*)

stk_push places *datum* on the top of the *stack*.

1.18.4 *stk_pop*

void **stk_pop*(*stk_Stack stack*)

stk_pop removes the top of the *stack* and returns the pointer. If the *stack* is empty, *stk_pop* returns **NULL**.

1.18.5 *stk_top*

void **stk_top*(*stk_Stack stack*)

stk_top returns a pointer to the datum on the top of the *stack*, but does *not* remove this datum from the *stack*. If the *stack* is empty, *stk_pop* returns **NULL**.

1.19 String Pool Routines

These routines provide support for string pool objects. In general, only the *str_find* and *str_findn* functions will be used. This function takes a pointer to a null-terminated string and returns a pointer to another null-terminated string which contains the same information. The pointer returned will be identical for all identical strings. Memory for string storage is automatically reclaimed at program termination on systems that support *atexit* or *on_exit*.

1.19.1 *str_pool_create*

str_Pool str_pool_create(**void**)

str_pool_create initialized a string pool object.

1.19.2 *str_pool_destroy*

void *str_pool_destroy*(*str_Pool pool*)

str_pool_destroy destroys the string pool object, *pool*, and all memory associated with it.

1.19.3 *str_pool_exists*

int *str_pool_exists*(*str_Pool pool*, **const char** **s*)

str_pool_exists returns non-zero if the string, *s*, is already in the string pool, *pool*.

1.19.4 *str_pool_find*

const char **str_pool_find*(*str_Pool pool*, **const char** **s*)

str_pool_find looks up the string, *s*, in the memory associated with the string pool object, *pool*. If the string is found, a pointer to the previously stored string is returned. Otherwise, the string is copied into string pool memory, and a pointer to the newly allocated memory is returned.

1.19.5 *str_pool_copy*

const char **str_pool_copy*(*str_Pool pool*, **const char** **s*)

str_pool_copy returns a copy of the string, *s*, using memory from the string pool object, *pool*. This can be used for data that is known to be unique. No checks are made for uniqueness, however; and a pointer to the string is not placed in the hash table.

1.19.6 *str_pool_copyn*

const char **str_pool_copyn*(*str_Pool pool*, **const char** **s*, **int** *length*)

str_pool_copyn returns a copy of the string, *s*, using memory from the string pool object, *pool*. The string will be *length* bytes long, and will be **NULL** terminated. This can be used for data that is known to be unique. No checks are made for uniqueness, however; and a pointer to the string is not placed in the hash table.

1.19.7 *str_pool_grow*

void *str_pool_grow*(*str_Pool pool*, **const char** **s*, **int** *length*)

str_pool_grow will grow a string in the specified *pool* until *str_pool_finish* is called. There must not be any other calls to modify the specified string *pool* between the first call to *str_pool_grow* and the call to *str_pool_finish*.

1.19.8 *str_pool_finish*

const char **str_pool_finish*(*str_Pool pool*)

str_pool_finish will finish the growth of a string performed by multiple calls to *str_pool_grow*. The string will be null terminated and will be entered into the specified string *pool*. Calls to *str_pool_grow* follows by a call to *str_pool_finish* is equivalent to a single call to *str_pool_find*.

1.19.9 *str_pool_get_stats*

str_Stats str_pool_get_stats(*str_Pool pool*)

str_pool_get_stats returns statistics about the specified string *pool*. The *str_Stats* structure is shown in Figure 7.

1.19.10 *str_pool_print_stats*

void *str_pool_print_stats*(*str_Pool pool*, *FILE *stream*)

str_pool_print_stats prints the statistics for the specified string *pool* on the specified *stream*. If *stream* is **NULL**, then **stdout** will be used.

```
typedef struct str_Stats {
    int count;                  /* Number of strings created */
    int bytes;                  /* Number of bytes allocated */
    int retrievals;             /* Total number of retrievals */
    int hits;                   /* Number of retrievals from top of a list */
    int misses;                 /* Number of unsuccessful retrievals */
} *str_Stats;
```

Figure 7: *str_Stats*

1.19.11 *str_exists*

int *str_exists*(**const char** *s)

str_exists acts like *str_pool_exists*, except the global string pool is used.

1.19.12 *str_find*

const char *str_find*(**const char** *s)

str_find acts like *str_pool_find*, except the global string pool is used. If the global string pool has not been initialized, it will be initialized automatically. Further, on systems that support *atexit* or *on_exit*, *str_destroy* will be called automatically at program termination.

1.19.13 *str_findn*

const char *str_findn*(**const char** *s, **int** *length*)

str_findn acts like *str_find*, except that the length of the string is specified, and the string does not have to be **NULL** terminated.

1.19.14 *str_copy*

const char *str_copy*(**const char** *s)

str_copy acts like *str_pool_copy*, except the global string pool is used. If the global string pool has not been initialized, it will be initialized automatically. Further, on systems that support *atexit* or *on_exit*, *str_destroy* will be called automatically at program termination.

1.19.15 *str_copyn*

const char *str_copyn*(**const char** *s, **int** *length*)

str_copyn acts like *str_copy*, except that the length of the string is specified, and the string does not have to be **NULL** terminated.

1.19.16 *str_grow*

void *str_grow*(**const char** *s, **int** *length*)

str_grow will grow a string until *str_finish* is called. There must not be any other calls to modify the global string pool between the first call to *str_grow* and the call to *str_finish*.

1.19.17 *str_finish*

const char *str_finish(**void**)

str_finish will finish the growth of a string performed by multiple calls to *str_grow*. The string will be null terminated and will be entered into the global string pool tables. Calls to *str_grow* follows by a call to *str_finish* is equivalent to a single call to *str_findn*.

1.19.18 *str_unique*

const char *str_unique(**const char** *prefix)

str_unique returns a unique string with the given prefix. This is not the most pretty way to generate unique strings, and should be improved. The string is placed in the string pool and does not need to be freed.

1.19.19 *str_destroy*

void *str_destroy(**void**)

str_destroy frees all of the memory associated with the global string pool. Since this function is called automatically at program termination on systems that support *atexit* or *on_exit*, there should be no need to call this function explicitly.

If this function is called explicitly, the next call to *str_find* will re-initialize the global string pool.

1.19.20 *str_get_stats*

str_Stats *str_get_stats(**void**)

str_get_stats returns statistics about the global string pool. The *str_Stats* structure is shown in Figure 7.

1.19.21 *str_print_stats*

void *str_print_stats(*FILE* *stream)

str_print_stats prints the statistics for the global string pool on the specified *stream*. If *stream* is **NULL**, then **stdout** will be used.

1.19.22 *txt_soundex*

const char **txt_soundex*(**const char** **string*)

Given a string, return (in a *static* buffer), the Soundex value for the string.

The basic algorithm for the soundex routine is from Donald E. Knuth's THE ART OF COMPUTER PROGRAMMING, Volume 3: Sorting and Searching (Addison-Wesley Publishing Co., 1973, pages 391 and 392). Knuth notes that the method was originally described by Margaret K. Odell and Robert C. Russell [US Patents 1261167 (1918) and 1435663 (1922)].

1.20 Timer Support

These routines provide access to a microsecond-resolution time that can be used for profiling.

1.20.1 *tim_start*

void *tim_start*(**const char** **name*)

Start the named timer.

1.20.2 *tim_stop*

void *tim_stop*(**const char** **name*)

Stop the named timer.

1.20.3 *tim_reset*

void *tim_reset*(**const char** **name*)

Reset the named timer to zero elapsed time. Use *tim_start* to reset the start time.

1.20.4 *tim_get_real*

double *tim_get_real*(**const char** **name*)

Get the wall time in seconds from the named timer. The return value is a *double* and has microsecond resolution if the current system provides that accuracy (most don't).

1.20.5 *tim_get_user*

double *tim_get_user*(**const char** **name*)

Get the number of seconds of user CPU time.

1.20.6 *tim_get_system*

double *tim_get_system*(**const char** **name*)

Get the number of seconds of system CPU time.

1.20.7 *tim_print_timer*

void *tim_print_timer*(*FILE *str*, **const char** **name*)

Print the named timer values to *str*. The format is similar to **time(1)**.

1.20.8 *tim_print_timers*

void *tim_print_timers*(*FILE *str*)

Print all the timers to *str*. The order is arbitary.

1.20.9 *_tim_shutdown*

void *_tim_shutdown*(**void**)

Free all memory associated with the timers. This function is called automatically at program termination. There should never be a need to call this function in user-level code.

A Maa

When Kheperà rose up out of the watery abyss of Nu, he created Maa as a foundation upon which to create everything else [Budge04, pp. 295–8].

References

[Aho88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1988.

[Budge04] E. A. Wallis Budge. *The gods of the Egyptians or studies in Egyptian mythology*, volume 2. Dover Publications, 1969.

[Knuth73] Donald E. Knuth. *Sorting and Searching*, volume 3 of The Art of Computer Programming. Addison-Wesley, Reading, MA, 1973.

[Pugh90] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, **33**(6):668–76, June 1990.