Library Reference Manual

$Revision: 2.22 $

Compiler Tools Group
Department of Electrical and Computer Engineering
University of Colorado
Boulder, CO, USA
80309-0425

Table of Contents

1 The Eli Library

The Eli library contains a collection of solutions to common problems in language implementation. If one of these problems is identified in the design of an application specification, the library's solution can be easily applied.

This manual describes Eli's frame modules. They provide basic operations that are not normally varied. Some of the operations are quite general, while others carry out specific translator-related tasks. In each case, however, the task performed by the module is largely independent of the overall problem being solved by the specification.

Frame modules are all code modules, and the facilities they export are made available via the associated header file. They can be used directly by C programs, and do not involve any other Eli facilities.

Another class of modules provided by Eli contain a mixture of code and specifications, of which some can be instantiated. These modules solve tasks such as name analysis according to Pascal scope rules. Descriptions of these modules are in the specification module library manual. See Chapter 1 [Specification Module Library], page 1.

1.1 Using Frame Modules

The library modules described in this manual are implemented directly in C. These modules may export constants, variables and routines to their clients. To reduce overhead, these modules have been pre-compiled and stored in the *frame*. Frame modules need not be instantiated, although it is allowable to do so. A frame module will be incorporated into a specification whenever anything exported by that module is used in the specification. If a frame module is instantiated by a specification, however, it will be included regardless of whether any exported facilities are used. In either case, source code for the module is included in the `source` product. See Section 1.2 [Source Version of the Processor], page 3.

Each module is associated with a header file that specifies the interface it presents to its clients. Any module requiring access to the exports of another module must include the header file associated with the exporting module.

Library header files are written according to certain conventions. We recommend strongly that you follow the same conventions when defining code modules to carry out tasks specific to your problem. Your header files will be included in generated modules, and Eli will make the same assumptions about them that it does about header files from the library. Failure to follow these conventions may result in errors when the generated programs are compiled. Since you will not recognize the generated programs, such errors may be difficult to diagnose.

By convention, every header file is *complete*: Including a module's header file in a C program provides all of the information needed to use that module. For example, the environment module exports operations that deliver definition table keys as their results. In order to use the environment module, therefore, a client must have access to the interface of the definition table module. Our convention therefore requires that the header file for the environment module include the header file for the definition table module. If the environment module header file did not include the definition table module header file, a C program that included that included the environment module header file would not necessarily have all of the information needed to use the environment module.

This convention greatly simplifies decisions about which header files to include, and avoids all questions of the order in which header files should be included. It does, however, require that each header file be protected against multiple inclusion. Protection is provided by defining a symbol in each header file and skipping the entire contents if that symbol is already defined. The symbol is the name of the header file, given as upper-case characters with any periods replaced by underscores.

Here is an excerpt from '`envmod.h`', the header file for the environment module:

```
#ifndef ENVMOD_H
#define ENVMOD_H

#include "deftbl.h"

typedef struct EnvImpl *Environment;    /* Set of Identifier/Definition pairs
*/

...

/***/
#if defined(__cplusplus) || defined(__STDC__)
extern DefTableKey DefineIdn(Environment env, int idn);
#else
extern DefTableKey DefineIdn();
#endif
/* Define an identifier in a scope
 *    If idn is defined in env then on exit-
 *       DefineIdn=key for idn in env
 *    Else let n be a previously-unused definition table key
 *    Then on exit-
 *       DefineIdn=n
```

```
 *       idn is defined in env with the key n
 ***/

...

#endif
```

Protection against multiple inclusion is provided by the test of `ENVMOD_H`. `DefineIdn` returns a definition table key, so the completeness convention requires that '`deftbl.h`' (the definition table module interface) be included. `Environment` is a type exported by the environment module, and it is defined in this interface.

1.2 Text Input

```
#include "source.h"

SrcBufPtr SrcBuffer;

char *SRCFILE;
char *TEXTSTART;

void initBuf(char *name, int f);

void refillBuf(char *p);

int finlBuf();
```

The source module has been designed to allow rapid access to the characters of a text file. A text file is a sequence of *lines*, each terminated by a newline character. There is no limit on the length of a line. The source module guarantees that if the first character of a line is in memory then all of the characters of that line, including the terminating newline character, are in contiguous memory locations. The newline terminating the last line in memory is followed by an ASCII NUL character, and thus the sequence of lines constitutes a C string. NUL characters are not allowed within the text file.

To use the module, first call `initBuf`, with a file name and descriptor. The file must be opened for reading (by `open(2)` prior to the call. Upon return, `SrcBuffer` points to a new text buffer, `SRCFILE` is the symbolic name of the input file, and `TEXTSTART` points to the first character of the first line of the input file. (If the input file is empty then `TEXTSTART` points to an ASCII NUL character.)

`SRCFILE` and `TEXTSTART` are components of the text buffer; `TEXTSTART` may be arbitrarily altered by any client of the source module. Normally, `TEXTSTART` is used to describe the position reached by the client in processing the buffer's text. Because it is a component of the text buffer, there is no need to save and restore it when another text buffer is created by a subsequent invocation of `initBuf`.

All source module operations take place on the text buffer pointed to by `SrcBuffer`, and both `SRCFILE` and `TEXTSTART` use `SrcBuffer` to access the text buffer. A client of the source module may manage a number of text buffers simultaneously by saving and restoring `SrcBuffer`.

When a client has processed all of the information in a text buffer, additional information can be obtained from the file by invoking `refillBuf` with a pointer to the text in the buffer. At least one line of text will be added to the text pointed to by the argument of `refillBuf`, and upon return `TEXTSTART` points to the first character of the augmented text. The content of the memory pointed to by the argument of `refillBuf` is undefined. (If there is no more information in the file then the string pointed to by the argument of `refillBuf` is not augmented. Upon return `TEXTSTART` points to the first character of that string and the content of the memory pointed to by the argument of `refillBuf` is undefined.)

Invocation of `finlBuf` frees all storage for the text buffer pointed to by `SrcBuffer`. Upon return, `SrcBuffer` contains a null pointer and the value of `finlBuf` is the descriptor of the file associated with the buffer. The file itself has not been closed.

1.3 Source Text Coordinates and Error Reporting

```
#include "err.h"

typedef POSITION *CoordPtr;

POSITION NoCoord;
CoordPtr NoPosition;

int LineOf(POSITION);
int ColOf(POSITION);

int LineNum;
POSITION curpos;

int ErrorCount[ ];

ErrorInit(int ImmOut, int AGout, int ErrLimit)
```

```
message(int severity, char *text, int grammar, CoordPtr source)

lisedit(char *name; FILE *stream; int cutoff, erronly)
```

This module implements the concept of a source text coordinate system and a set of errors of various levels of severity. Error reports are tied to particular positions in the source text coordinate system, and may be combined with the source text in a separate pass.

The coordinate of a source text position is a pair (line index, column index), and is defined by a structure of type `POSITION`. `LineOf` and `ColOf` are access functions for the elements of the pair; they may be used either to read or to set these elements.

`LineNum` and `curpos` are variables provided by the error reporting module for the use of its clients. `LineNum` initially has the value 1. It is neither read nor set by the error module. The initial value of `curpos` is undefined, and it is also neither read nor set by the error module.

`ErrorInit` may be called in order to change the default behavior of the error module. By default, all error messages are written to `stderr` as they occur, however, if `ErrorInit` has been called with a value of `0` for `ImmOut`, errors will not be reported until `lisedit` is used. Buffering is prevented by calling `fflush` after each output. The error message format is illustrated by the following:

```
"filename", line 24:3 ERROR: boolean type required AG=124
```

The information shown is: the file name of the input file, the line and column in that file where the error occurred, the severity (see below) and the error message iself. The integer value following `AG` can be used to provide additional information when a particular report may originate from many places. This integer value is not reported if `ErrorInit` has been called with a value of `0` for `AGout`.

The format is designed so that it can be used as input to a special mechanism, such as an intelligent editor, for making the reports known to the user.

After printing to `stderr`, the errors are also queued for possible printing later with `lisedit`. `lisedit` prints to `stdout` the source line containing the error, with an arrow pointing at the corresponding column. Then the body of the error message is printed. Only messages whose severity is larger than `cutoff` will be printed. If `erronly` is nonzero, only lines with associated error reports will be printed.

`Message` is used to make an error report to the error module. Both source program and compiler errors are reported via `message`. Six error severities are defined by constants exported by the module:

`NOTE` The message is intended to convey additional information to the user, not to report an error.

`WARNING` The message reports an anomaly that may be indicative of an error.

`ERROR` The message reports a definite error.

`DEADLY` The message reports a violation of an assertion within the compiler.

A compiler should be able to carry on after detecting any errors less severe than deadly errors. It may be necessary to repair some internal data structures in order to guarantee their consistency, but the repairs are usually not difficult to provide; see any standard compiler construction text. `Message` returns normally after accepting a report of an error that is less severe than a deadly error.

Violations of compiler assertions signal programming errors within the compiler itself. Attempts to continue under such circumstances are likely to result in further corruption and eventual catastrophic failure. `Message` therefore does *not* return after accepting a report of a deadly error. Instead, it outputs any queued reports and terminates the program with `exit(1)`.

The `text` argument to `message` points to a character string describing the error. This character string must remain unchanged until the reports are output at the end of the compilation; `message` does not copy it.

`grammar` and `source` serve to locate the error in both the compiler and the source text. The former is most useful in the case of violations of compiler assertions and limits. It specifies the particular assertion or, in the case of a generated compiler, the specification rule that led to the violated assertion. The latter gives the source text coordinates of the construct the compiler was processing at the time the report was issued. Some errors are not associated with a particular source language construct. Reports of these errors should use `NoPosition` in lieu of coordinates.

At any time during the compilation, `ErrorCount[`*severity*`]` contains the number of reports of class *severity* that have been issued so far.

If an error is reported with an invalid severity code, `message` sets its severity to `DEADLY` after printing the message and its severity code on `stderr`.

If the number of errors at severity level `ERROR` plus the number at severity level `FATAL` is greater than 10 added to the current line number divided by 20 then compilation is aborted with a `DEADLY` message. When the number of lines of source is reasonably large this effectively amounts to a limit of one of these errors for every 20 lines of source code (ie. a 5% error rate). This error limit is not checked if `ErrorInit` has been called with a value of `0` for the argument `ErrLimit`.

1.4 Memory Object Management

This module provides high-speed memory allocation that supports "growing" objects – objects whose size is not known a priori. Any number of regions can be defined, and storage managed independently by region. Within one region the storage is allocated and freed in a last-in, first-out manner that allows freeing of a large number of objects with a single operation.

```
#include "obstack.h"

void obstack_init(ObstackP obstack);
void obstack_begin(ObstackP obstack, int size, int alignment);
int obstack_chunk_size(ObstackP obstack);
int obstack_alignment_mask(ObstackP obstack);

void *obstack_alloc(ObstackP obstack, int size);
void *obstack_copy(ObstackP obstack, void *data, int size);
void *obstack_copy0(ObstackP obstack, void *data, int size);
void *obstack_strcpy(ObstackP obstack, char *data);

void obstack_blank(ObstackP obstack, int size);
void obstack_grow(ObstackP obstack, void *data, int size);
void obstack_grow0(ObstackP obstack, void *data, int size);
void obstack_strgrow(ObstackP obstack, char *data);
void obstack_1grow(ObstackP obstack, int data_char);
void obstack_ptr_grow(ObstackP obstack, void *data);
void obstack_int_grow(ObstackP obstack, int data);

void obstack_blank_fast(ObstackP obstack, int size);
void obstack_1grow_fast(ObstackP obstack, int data_char);
void obstack_ptr_grow_fast(ObstackP obstack, void *data);
void obstack_int_grow_fast(ObstackP obstack, int data);

void *obstack_finish(ObstackP obstack);

void obstack_free(ObstackP obstack, void *block);

void *obstack_base(ObstackP obstack);
void *obstack_next_free(ObstackP obstack);
int obstack_object_size(ObstackP obstack);
int obstack_room(ObstackP obstack);
```

Each region is represented by a data structure of type `Obstack`. A pointer of type `ObstackP`, which addresses this data structure, is used to specify the region. Here is an example showing how a region might be declared and initialized:

```
Obstack obstk;

obstack_init(&obstk);
```

All the apparent functions operating on regions are macros. Each takes a pointer of type `ObstackP` as its first argument. This pointer may be evaluated many times, so you should not use an expression as the first argument of any of these macros. (Any arguments other than the first are evaluated exactly once.) If you need to compute the appropriate `ObstackP`, use the following strategy:

```
_obstack = (address expression); obstack_xxx(_obstack, ...);
```

The variable `_obstack` is of type `ObstackP`, and is exported by the module for use by clients. Its value is never inspected or changed by the module itself or any of the macros.

A region is a collection of objects managed in a last-in, first-out manner. Each region is independent of the others, and is characterized by a *chunk size* and an *alignment*. As objects are added to the collection, blocks of memory of the given chunk size are allocated to hold them. The address of each object in the collection is guaranteed to be divisible by the alignment, which must be a power of 2.

When the storage already available for a collection is insufficient for an object being added to the collection, then a new chunk is allocated. The size of the new chunk is the minimum of the chunk size parameter and twice the size of the object to be added. Thus the chunk size parameter does *not* limit the size of an object that can be stored in the collection, but it does affect the number of system requests for storage. If it is not specified when the collection is initialized, a default value equivalent to one virtual memory page is used. Chunks are normally allocated by `malloc` or `realloc`, but you can substitute a different storage allocator by re-defining the macros `obstack_chunk_alloc` and `obstack_chunk_realloc` to be the names of your allocator functions. Be certain that your allocator copes with memory exhaustion. Give the macro definition before including the module interface specification:

```
#define obstack_chunk_alloc MyMalloc
#include "obstack.h"
```

After an object is added to a collection, the next available address is adjusted to be divisible by the alignment parameter of the collection. If *any* address is suitable as an object address then the alignment should be 1 (the zeroth power of 2). The default alignment value is that suitable for an object of the primitive type `double`, normally the type that is most stringently aligned.

The first macro applied to a region must be either `obstack_init` or `obstack_begin`, both of which create an empty collection of objects. A collection initialized by `obstack_init` will have the default values for its chunk size and alignment parameters, while `obstack_begin` allows the user to set those values. If either the `size` or `alignment` argument of the `obstack_begin` call is `0` then the corresponding default value is used. Either of these values may be inspected or changed at any time via the operations `obstack_chunk_size` and `obstack_alignment_mask` respectively. These macros may be called either on the left side of an assignment (to change the value) or within an expression (to inspect the value). If the chunk size is changed, it can be returned to its default value by assigning the value 0 to `obstack_chunk_size`:

```
obstack_chunk_size(&obstk) = 0;
```

The alignment mask is an integer one less than the power of 2 that must divide each object address. To make a change in the alignment mask effective, you must create an empty object. For example, the following code guarantees that the addresses of objects subsequently created in the collection `obstk` will be divisible by 4:

```
obstack_alignment_mask(&obstk) = 3; (void)obstack_alloc(&obstk, 0);
```

Once a region has been initialized, there are two basic strategies for creating objects: *allocation* and *growth*. Objects are allocated when their size is known a priori; they are grown when their size is not known a priori.

Allocation is the simplest strategy. Suppose that it was necessary to create an object capable of storing an array of five integers. The call `(int *)obstack_alloc(&MyStack, 5 * sizeof(int))` would create such an object in the collection `MyStack`, and yield a pointer to that object. The contents of the created object are undefined. If an array of five integers was already stored in the variable `ArrayValue`, and this array was to be the initial contents of the created object, then the call `(int *)obstack_copy(&MyStack, ArrayValue, 5 * sizeof(int))` would create and initialize an appropriate object. The operation `obstack_copy0` is identical to `obstack_copy`, except that it adds a single zero byte after the value that was copied. An object whose initial contents are to be the characters of an existing null-terminated string should be created by the operation `obstack_strcpy`. This operation does not require a length specification; it determines the length from the given string.

With the growth strategy, a single object is created by a sequence of macro calls rather than a single call. Each call causes the object to grow in size, and possibly establishes some of the initial contents of the object. An object can be moved by the module while that object is growing. The last call in the sequence is to either `obstack_finish` or some allocation operation, which terminates the object's growth and fixes its address.

An obstack can only accomodate a single growing object at any time. While that object is growing, no allocation operations may be issued for the obstack. After `obstack_finish` has been called, completing the growth of the object, the obstack is again able to accept any operation. Thus the legal sequence of operations on an obstack can be described by the following regular expression (`A` stands for an allocation operation and `G` stands for a growth operation):

```
obstack_init ( A | G+ ( obstack_finish | A ) )*
```

Macros implementing the growth strategy parallel, for the most part, the macros implementing the allocation strategy. They have the same pattern of arguments as their allocation counterparts, but do not return a pointer to an object because no object exists until the call of `obstack_finish`. To grow an object by a given amount without specifying the initial contents of that part of the object, use `obstack_blank`. If an initial contents is known, grow the object with one of the operations `obstack_grow`, `obstack_grow0` or `obstack_strgrow` as appropriate.

The special operation `obstack_1grow` is used for placing characters into a growing object. It's argument is the actual value that is the initial content, rather than the address of that value as in the other growth and allocation macros. Here is an example of how `obstack_strcpy` could be implemented using `obstack_1grow`:

```
char *
obstack_strcpy(obstk, data)
ObstackP obstk; char *data;
{ register char c, *p = data;
  if (p) while (c = *p++) obstack_1grow(obstk, c);
  obstack_1grow(obstk, '\0');
  return (char *)obstack_finish(obstk);
}
```

The growth macros check that the current chunk has enough space in it for the growth increment. If the check fails, a new chunk is allocated and the growing object is copied into the new chunk. Two additional operations, `obstack_blank_fast` and `obstack_1grow_fast`, can be used when sufficient space in the current chunk can be guaranteed. These macros are identical to their normal counterparts `obstack_blank` and `obstack_1grow`, except that they do not check the space remaining in the current chunk. The operation `obstack_room` returns the amount of free space in the current chunk.

The collection of objects in a region is managed in a last-in, first-out manner. This means that there is an operation, `obstack_free`, to remove objects from the collection. A call of `obstack_free` specifies a region and an object in that region. This call removes the specified object *and all objects added to the collection after the specified object was added.*

If an `obstack_free` operation frees all of the objects in a chunk, that chunk is returned to the underlying storage allocator. Chunks are normally returned by `free`, but you can specify a different routine by re-defining the macro `obstack_chunk_free` to be the name of that routine. (The macros `obstack_chunk_alloc`, `obstack_chunk_realloc` and `obstack_chunk_free` must be defined consistently.)

While an object is being grown, its current base address can be obtained by calling the macro `obstack_base`, its current size by calling `obstack_object_size`, and the address of the first location above it by calling `obstack_next_free`. This allows one to build a module that will give clients access to the contents of an immature object while it is still growing. Don't forget that the base address will change during the growth period if the object outgrows the current chunk. Also, `obstack_next_free` can be used on the left-hand side of an assignment, thereby providing a way to "shrink" a growing object. Its value should never be increased, nor should it be decreased beyond the value yielded by `obstack_base`.

1.5 Character String Storage

This module provides both temporary and persistent storage for arbitrary-length character strings. Strings stored by the module can be identified by either a pointer or an integer. The set of persistent strings can be written at any time to a text file as either a sequence of lines, one string per line in the form of a C string denotation, or as the C definition of an initialized data structure.

```
#include "csm.h"

int stostr(char *c, int l)
char *StringTable(int i);
char *NoStr
int NoStrIndex;
int numstr;

ObstackP Csm_obstk;
char *CsmStrPtr;
```

```
char *prtstcon(FILE *d, char *p)

savestr(FILE *d)

dmpstr(FILE *d)
```

`Stostr` stores a character string `c` of length `l` in the memory, returning the index of the string. `StringTable` is a macro that delivers a pointer to a null-terminated string, given the index of that string established by `stostr`. The number of strings currently stored is given by `numstr` (initially 0); thus `numstr-1` is the largest index that can be used as an argument to `StringTable`.

'`csm.h`' defines the constant `NoStr` to be a unique character pointer that represents no string. It also defines `NoStrIndex` to be a unique string index that will never be used to represent a string. The string indexed by `NoStrIndex` contains no characters.

`Csm_obstk` is a region that can hold sets of strings. It can be managed by the operations of the memory object allocator (see Section 1.4 [obstack], page 7). The actual characters of the strings stored by `stostr` are stored in `Csm_obstk`. `CsmStrPtr` is a pointer that can be set by a client of the character storage module to point to a string in `Csm_obstk`. When this pointer is passed to `stostr`, `stostr` assumes that another copy of the string need not be stored.

`Prtstcon` prints the string pointed to by `p` on the current line of file `d`. The string is printed as a C string constant with the quotes omitted. This function can be omitted from the module by compiling with the C flag `-DNOPRINT`. `Dmpstr` prints the set of strings `StringTable(0)` through `StringTable(numstr-1)` on file `d` as a sequence of lines. This function can also be omitted from the module by compiling with the C flag `-DNOPRINT`.

`Savestr` writes the current state of the module on file `d`, in the form of C initialized variables. Only the strings stored by `stostr` will be written to `d`. This file can be used to create a copy of the module initialized to the current state by naming it '`csmtbl.h`' and recompiling the module. The save function is available only if the module was compiled with the C flag `-DSAVE`, and if the printing routines were not omitted.

1.6 Character String Arithmetic

```
#include "strmath.h"

char *stradd(char *a, char *b, int base);
```

```
char *strsub(char *a, char *b, int base);
char *strmult(char *a, char *b, int base);
char *strdivf(char *a, char *b, int base);
char *strdivi(char *a, char *b, int base);
char *strrem(char *a, char *b, int base);
char *strpow(char *a, char *b, int base);
char *strsqrt(char *a, int base);

char *strnorm(char *a, int oldbase, int newbase, char *symbols);

char *strnumb(char *a, int *flag, int base);

int strmath(int select, ...);
```

This package is made available by adding the following line to a type-`specs` file:

```
$/Tech/strmath.specs
```

The strings accepted by this package are essentially those of most higher level languages:

```
[+/-][d*][.][d*][e[+/-]d*]
```

Here `[]` indicate optional parts, `+/-` indicates that a sign may be present, `d` indicates digits in the chosen base, `*` indicates repetition, `.` is a period representing the separation between the integer and fractional parts of the number, and `e` stands for any one of the exponent symbols given in the control string EXP_SYMBOL. Strings generated by the package have the same formats. The actual characters used to represent digits, signs, fractional separators and exponent symbols are determined by default or by settings established by the `strmath` operation.

In general (with the exception of `strnumb`) the operations return a NULL pointer when an error is found in the input values or occurs during computation (e.g., overflow). When a null value is returned, the global `errno` is set to indicate the type of error. If multiple errors occur, only the first is reported.

When operations are cascaded, with the result of one being used as an operand of another, there is no need to check each operation individually. If an error occurs in an early operation, the resulting NULL pointer will be an invalid input to the next operation, thus guaranteeing that it will yield a NULL pointer. The NULL pointer will therefore propagate, and appear as the final value. Since only the first error is reported, the global `errno` will not be changed when any of the subsequent invalid inputs are detected.

The error codes used are defined in <errno.h>:

`EINVAL` An error in the format of an input value made it invalid. (The function `strnumb` can be used to assist in diagnosing these problems.)

`EDOM` The given value falls outside the domain of the operation (e.g. it is too large, or negative when a positive value is required).

`ERANGE` The result of the computation cannot be adequately represented (e.g., divide by zero, overflow).

The usual rules for the use of `errno` apply: It is not explicitly cleared. If desired, this must be done before calling the desired function. It is set by these functions only if an error occurs.

Most of the operations exported by the strmath module return pointer values. These pointers address the result string, which occupies space in a static array. All operations use the same static array for their results. Thus the pointer returned by an operation will be valid only until another operation of the module completes. It is therefore the responsibility of each client of the module to make a copy of any result string whose value must be preserved beyond the completion of the next strmath operation.

The operations `stradd`, `strsub`, `strmult`, `strdivf` (full divide, possibly yielding a fractional part), `strdivi` (integer quotient), `strrem` (integer remainder) and `strpow` perform the indicated operation on two string operands. In each case the operation is performed in the radix given by `base`, and the result is delivered as a string or `NULL`. The unary operation `strsqrt` takes the square root of its string operand in the radix given by `base`, and the result is delivered as a string or `NULL`.

The `strnorm` operation converts `a` from radix `oldbase` to radix `newbase`, normalizing it in the process, and the result is delivered as a string. The format of that string depends on the `symbols` argument:

`symbols=NULL`
: Whole number and fraction parts separated by the defined fractional separator unless the result can be expressed as an integral value, exponent marker and exponent if the length would exceed `integer_size` digits.

`symbols` is an empty string
: Sequence of digits if the length does not exceed `integer_size` digits, otherwise `NULL`.

`symbols` is a non-empty string
: A fractional separator is guaranteed to appear in the result. The first character of `symbols` is taken as the exponent marker. If there are additional character in the

string then they will be taken as the fractional separator, the minus sign, and the plus sign respectively. (The normally defined for these purposes will be used if they do not appear in `symbols`.)

If the intent of the operation is simply to perform radix conversion, use `symbols=NULL`. The resulting string will be in the normal format delivered by other operations of this module. Using an empty string for `symbols` guarantees that the result is in integer form. (If this is not possible, the result will be `NULL`.) This guarantee is important when the number is being output in a position where an integer is required. Using a non-empty string for `symbols` guarantees that the result is in floating-point format. It also allows one to easily vary the characters used for the exponent marker, fractional separator, and signs. This is important when the number is being output where a real is required.

The `strnumb` operation is intended for input of numbers, particularly in cases where there may be a radix marker at the end of the number string. It scans the string `a` and interprets it as a value in the radix given by `base`. If an error is found, the appropriate error value is returned in the variable `flag`. Otherwise `flag` will be set to zero. The operation returns a pointer to the last character scanned. This character will be the character at which the error was detected if `flag` is nonzero, otherwise it is the first character that does not belong to the number. Termination of the scan is controlled by the setting of `STRM_CHECK_DIGITS`.

The `strmath` operation is used to vary some characteristic of the module. It is called with a selection symbol given in the following table, and one additional argument which is the value of the characteristic being set. It returns the value 1 if the requested setting succeeded. Otherwise 0 is returned and no selected value has been changed. The caller is responsible for ensuring that the contents of the string arguments are disjoint sets as described below.

`strmath(STRM_DIGITS,‘string’)`

Change the string defining the set of valid digits. For example, "0123456789ABCDEF" is a string that defines a set of digits for bases 2-16. Output strings are built by indexing this string to obtain the representation of each computed digit. The matching of alphabetic characters in input strings depends upon the setting of `STRM_IGNORE_CASE`. Unpredictable results will be obtained if any character is repeated.

The default value is the string consisting of the digits '0'-'9', the upper case alphabetic characters, the lower case alphabetic characters, and the characters '%' and '$'. This string provides 64 distinct characters, thus supporting any radix not greater than 64.

`strmath(STRM_EXP_SYMBOLS,‘string’)`

Change the string defining the set of characters to be accepted as representations of the exponent symbol. Any character in the string is acceptable in input strings; in output strings the first member of the set is used. Unpredictable results will be obtained if any character is repeated or is used as a digit.

The default value of this string is "^".

`strmath(STRM_SIGNS,‘string’)`

Change the string defining the set of characters to be accepted as representations of negative and positive signs. The first element of the string is the character representing a negation, and the remainder are all taken as representations of a positive value. In output strings the first element is used to show negation, and the second value to represent a positive sign (if one is requested). Unpredictable results will be obtained if any character is repeated or is used as an exponent symbol or digit.

The default value of this string is "-+".

`strmath(STRM_SEPARATORS,‘string’)`

Change the string defining the set of characters to be accepted as representations of the fractional separator. Any are acceptable in input strings; in output strings the first element of the set is used. Unpredictable results will be obtained if any character is repeated or is used as a sign, exponent symbol or digit.

The default value of this string is ".".

`strmath(STRM_EXP_BASE,‘integer’)`

Define the radix assumed for the exponents of numbers. If this value is zero, the exponent radix is taken to be the same as the number radix, which is supplied on each call to the strmath functions. Otherwise the specified value is used as the exponent radix, independent of the number radix.

`strmath(STRM_INTEGER_SIZE,‘integer’)`

Set the maximum number of digits that a representation of an integer may have. This value, plus `STRM_ROUND_SIZE`, may be no greater than `ARITH_SIZE`. Values with more than `STRM_INTEGER_SIZE` significant digits will be represented as values with exponents, even if integral in value.

The default value is `ARITH_SIZE-2`.

`strmath(STRM_ROUND_SIZE,‘integer’)`

Set the number of additional digits a value may have a value may have if not represented as an integer. This value, plus `STRM_INTEGER_SIZE`, may be no greater than `ARITH_SIZE`. Values with more than `STRM_INTEGER_SIZE+STRM_ROUND_SIZE` significant digits will be rounded on output to values with no more than that `STRM_INTEGER_SIZE+STRM_ROUND_SIZE` digits.

The default value is 2.

`strmath(STRM_ROUNDING,‘mode’)`

Select the rounding mode to be applied to output values. The following modes are recognized:

`STRM_EVEN_ROUND`

The output value is rounded to the nearest value representable in the given number of digits. If two values are equally near, the one with an even last digit will be returned. This the default setting.

`STRM_ZERO_ROUND`

The output value is rounded to the value representable in the given number of digits that is closest to and no greater in magnitude.

`STRM_UP_ROUND`

The output value is rounded to the value representable in the given number of digits that is closest to but no less.

`STRM_DOWN_ROUND`

The output value is rounded to the value representable in the given number of digits that is closest to but no greater.

`STRM_HAND_ROUND`

The output value is rounded to the nearest value representable in the given number of digits. If two values are equally near, the one with the greater magnitude will be returned (as is normally done when computing by hand).

`strmath(STRM_DENORMALIZE,'Flag')`

Select the treatment of values with exponent -1. '`Flag`'=1 allows values with exponents that would normally be -1 to be represented with a 0-valued digit before the fractional separator on output. If '`Flag`'=0, then the initial digit of an output value will always be nonzero.

The default setting is 1.

`strmath(STRM_INEXACT,'Flag')`

Select the treatment of inexact values. (An inexact value is one that has lost digits in the course of computation, or whose value was derived from an inexact value.) '`Flag`'=1 indicates that input values containing a fractional separator should be regarded as inexact, and that a fractional separator and fractional part should be included on output of any inexact value. '`Flag`'=0 indicates that all input values should be regarded as exact, and that no fractional separator should be included on output if the fractional part of the value being output is zero.

The default setting is 1.

`strmath(STRM_CHECK_DIGITS,'Flag')`

Select the treatment of non-digit characters on input. '`Flag`'=1 indicates that all characters of a string be checked on input to ensure that they are valid digits and are legitimate in the specified number radix. If this is not so, an error is reported by the operation reading the number. '`Flag`'=0 indicates that the scan should end with the first character that is not legitimately part of the number, and no error is to be reported signalled.

The default setting is 1.

`strmath(STRM_IGNORE_CASE,'Flag')`
: Select the treatment of the case of letters representing digits on input. '`Flag`'=1 indicates that the case of letters in input values, and the case of letters in the STRM_DIGITS string, is to be ignored when matching and defining the value of elements of an input number. '`Flag`'=0 indicates that case is significant.

  The default setting is 1.

1.7 Unique Identifier Management

```
#include "idn.h"

int dofold;

prtidnv(FILE *d, int i)

saveidn(FILE *d)

dmpidn(FILE *d)

mkidn(char *c, int l, int *t, int *s)
```

These functions implement the concept of unique identifiers. The representation is an integer, the index of the identifier's string in the character string memory (see Section 1.5 [storage], page 11).

'`idn.h`' defines the constant `NoIdn` to be a unique integer that is never used to represent an identifier. The identifier string represented by `NoIdn` contains no characters.

`Prtidnv` prints the string for identifier `i` on the current line of file `d`. This function can be omitted from the module by compiling with the C flag `-DNOPRINT`. `Dmpidn` prints the current state of the lookup mechanism on file `d` as a sequence of lines. This function can also be omitted from the module by compiling with the C flag `-DNOPRINT`.

`Saveidn` writes the current state of the module on file `d`, in the form of C initialized variables. This file can be used to create a copy of the module initialized to the current state by naming it '`idntbl.h`' and recompiling with the C flag `-DINIT`. The save function is available only if the module was compiled with the C flag `-DSAVE`, and if the printing routines were not omitted. An initial state is usually created by `adtinit`.

`Mkidn` determines the unique identifier for a string `c` of length `l`. If the string has not been previously passed to `mkidn` then it is assigned the first available location in the character string memory, and the index of that position is stored in the location pointed to by `s` . The value pointed to by `t` is noted but not changed. If the string has previously been passed to `mkidn` then the character string memory index assigned when it was first passed to `mkidn` is stored in the location pointed to by `s`, and the value of `t` noted at that time is stored in the location pointed to by `t`. `Mkidn` can be used as a processor by the lexical analysis portion of an Eli-generated program (see Lexical Analysis.)

The variable `dofold` controls the interpretation of upper and lower case alphabetic characters by `mkidn`: If `dofold` contains 0 then upper and lower case versions of the same letter are considered to be distinct; otherwise no distinction is made between upper and lower case versions of the same letter. This variable is initially 0, and must be explicitly set nonzero if desired. If it is known that upper and lower case letters must *always* be distinct, the variable and associated folding code can be removed from the module by compiling it with the C flag `-DNOFOLD`.

1.8 Contour-Model Environment

This module implements a standard contour model for name analysis. The data structure is a tree of *scopes*, each of which can contain an arbitrary number of *definitions*. A definition is a binding of an identifier to an object in the definition table (see PDL Reference Manual); the definition does not carry any other information itself. The environment module provides operations for building scope trees, adding definitions to specific scopes, and searching individual scopes or sequences of scopes for the definition of a particular identifier.

The module is capable of building multiple trees of scopes, and it places no constraints on the sequence of construction, definition and lookup operations.

```
#include "envmod.h"

Environment NewEnv ();
Environment NewScope (/* Environment e */);

int AddIdn (/* Environment e, int i, DefTableKey k */);
DefTableKey DefineIdn (/* Environment e, int i */);

DefTableKey KeyInScope (/* Environment e, int i */);
DefTableKey KeyInEnv (/* Environment e; int i */);
```

The type Environment is a pointer to the data structure representing the tree of scopes. Identifiers are represented by integers. The bindings contained in a scope are between integers and arbitrary pointers. These arbitrary pointers are all obtained via two operations external to the environment module:

`DefTableKey NoKey`

Returns the same value on every invocation.

`DefTableKey NewKey()`

Returns a different value on each invocation. The value returned by `NoKey` is never returned by `NewKey`.

`NoKey` and `NewKey` may be supplied by the user of Eli, or they may be exported by a generated definition table module (see PDL Reference Manual).

`NewEnv` creates a new tree consisting of a single, empty scope and returns a reference to that empty scope. `NewScope` creates a new empty scope as a child of the scope referenced by its argument and returns a reference to that empty scope.

`AddIdn` checks the scope referenced by its `e` argument for a definition of the identifier specified by its `i` argument. If no such definition is found, a definition binding the identifier `i` to the definition table object specified by `k` is added to scope `e`. `AddIdn` returns the value `0` if a definition for `i` already exists, and returns `1` otherwise.

`DefineIdn` behaves exactly like `AddIdn`, except that if the referenced scope contains no definition of `i` then `DefineIdn` obtains a value from `NewKey` and binds `i` to that value in scope `e`. In addition, `DefineIdn` always returns the definition table key associated with `i`.

`KeyInScope` checks the scope referenced by its `e` argument for a definition of the identifier specified by its `i` argument. If no such definition is found, `NoKey` is returned. If a definition is found, the definition table object bound to `i` is returned.

`KeyInEnv` behaves the same way as `KeyInScope` except that if no definition for `i` is found in scope `e` then the search continues through successive ancestors of `e`. If no definition for `i` is found in `e` or any of its ancestors, `NoKey` is returned. Otherwise the definition table object bound to `i` is returned.

If any function is invoked with an invalid environment argument (`(ENVIRONMENT *) 0`), a deadly error ("CurrEnv: no environment") is reported (see Section 1.3 [Source text coordinates and error reporting], page 4 for a discussion of deadly errors).

1.9 Storage Layout

This module provides operations that determine the storage requirement of a composition of two objects, each of which has its own storage requirement. The storage requirement of an object is determined by the number of memory units it occupies, the number by which its address should be divisible, and whether its address is the address of its first memory unit or the address of the first memory unit above it. Two distinct composition strategies, concatenation and overlaying, are supported by this module. (When two objects are concatenated they are allocated adjacent areas of memory; when they are overlaid they share memory.)

```
#include "Storage.h"

StorageRequired NewStorage(/* int size, align, top; */);
StorageRequired CopyStorage(/* StorageRequired a; */);
StorageRequired ArrayStorage(/* int n; StorageRequired a; */);

int StorageSize(/* StorageRequired a; */);
int StorageAlignment(/* StorageRequired a; */);

int Concatenate(/* StorageRequired a, b; */);
int Overlay(/* StorageRequired a, b; */);
```

The type `StorageRequired` is a pointer to the data structure representing a storage requirement. There is one predefined value, `NoStorage`, of type `StorageRequired`. `NoStorage` is a null pointer, and represents "no storage requirement".

`NewStorage` creates a new description of a storage area requiring `size` memory units, whose address must be divisible by `align`, and returns a reference to that description. If `top` is `TRUE` then the address of the storage area is the address of the first memory unit above the area; otherwise the address is the address of the first memory unit of the area itself. `CopyStorage` creates a new description of a storage area whose requirements are identical to those of `a` and returns a reference to that description. `ArrayStorage` creates a new description of a storage area for an array of `n` elements, each described by `a` and returns a reference to that description.

`StorageSize` returns the number of memory units required by storage area `a`. The function `StorageAlignment` returns the number by which the address of storage area `a` must be divisible.

`Concatenate` determines the requirements of the storage area resulting when storage area `b` is concatenated to storage area `a`. Area `b` is placed above area `a` if the address of area `a` is the address of its first memory unit. If the address of area `a` is the address of the first memory unit above that area, then area `b` is placed below area `a`. In either case, area `b` is placed as close to area `a` as possible, without overlapping it. The description referenced by `a` is changed to describe the result of the concatenation, and `Concatenate` returns the relative address of area `b` within that resultant area.

`Overlay` determines the requirements of the storage area resulting when storage area `b` is overlaid onto storage area `a`. The description referenced by `a` is changed to describe the result of the overlay, and `Overlay` returns the relative address of area `b` within that resultant area.

This module should be made available to the Eli specification by including the following line in a type-`specs` file:

```
$/Tech/Storage.specs
```

Index

O

P

R

S

T

W