C Interfaces and Implementations Quick Reference

Interface summaries are listed below in alphabetical order; the subsections name each interface and its primary type, if it has one. The notation “T is opaque *X*_T” indicates that interface *X* exports an opaque pointer type *X*_T, abbreviated as T in the descriptions. The representation for *X*_T is given, if the interface reveals its primary type.

The summary for each interface lists, in alphabetical order, the exported variables, excluding exceptions, followed by the exported functions. The prototype for each function is followed by the exceptions it can raise and a concise description. The abbreviations “c.r.e.” and “u.r.e.” stand for checked and unchecked runtime error(s).

The following table summarizes the interfaces by category and gives the pages on which the summaries begin.

Fundamentals		*ADTs*		*Strings*		*Arithmetic*		*Threads*	
Arena	5	Array	7	Atom	10	AP	2	Chan	13
Arith	6	ArrayRep	8	Fmt	15	MP	22	Sem	28
Assert	9	Bit	11	Str	33	XP	43	Thread	41
Except	14	List	18	Text	38				
Mem	20	Ring	26						
		Seq	29						
		Set	30						
		Stack	32						
		Table	36						

AP

`T` is opaque `AP_T`

It is a c.r.e. to pass a null `T` to any `AP` function.

```
T AP_add(T x, T y)                                   Mem_Failed
T AP_addi(T x, long int y)                           Mem_Failed
```

return the sum `x` + `y`.

```
int AP_cmp(T x, T y)
int AP_cmpi(T x, long int y)
```

return an int <0, =0, or >0 if `x`<`y`, `x`=`y`, or `x`>`y`.

```
T AP_div(T x, T y)                                   Mem_Failed
T AP_divi(T x, long int y)                           Mem_Failed
```

return the quotient `x`/`y`; see `Arith_div`. It is a c.r.e. for `y`=0.

```
void AP_fmt(int code, va_list *app,                  Mem_Failed
    int put(int c, void *cl), void *cl,
    unsigned char flags[], int width, int precision)
```

a `Fmt` conversion function: consumes a `T` and formats it like `printf`'s `%d`. It is a c.r.e. for `app`, `*app`, or `flags` to be null.

```
void AP_free(T *z)
```

deallocates and clears `*z`. It is a c.r.e. for `z` or `*z` to be null.

```
T AP_fromstr(const char *str, int base,              Mem_Failed
    char **end)
```

interprets `str` as an integer in `base` and returns the resulting `T`. Ignores leading white space and accepts an optional sign followed by one or more digits in `base`. For 10<`base`≤36, lowercase or uppercase letters are interpreted as digits greater than 9. If `end`≠null, `*end` points to the character in `str` that terminated the scan. If `str` does not specify an integer in `base`, `AP_fromstr` returns null and sets `*end` to `str`, if `end` is nonnull. It is c.r.e. for `str`=null or for `base`<2 or `base`>36.

`T AP_lshift(T x, int s)` `Mem_Failed`

returns `x` shifted left by `s` bits; vacated bits are filled with 0s, and the result has the same sign as `x`. It is a c.r.e. for `s`<0.

`T AP_mod(T x, T y)` `Mem_Failed`

`long AP_modi(T x, long int y)` `Mem_Failed`

return `x` mod `y`; see `Arith_mod`. It is a c.r.e. for `y`=0.

`T AP_mul(T x, T y)` `Mem_Failed`

`T AP_muli(T x, long int y)` `Mem_Failed`

return the product $x \cdot y$.

`T AP_neg(T x)` `Mem_Failed`

returns $-x$.

`T AP_new(long int n)` `Mem_Failed`

allocates and returns a new `T` initialized to `n`.

`T AP_pow(T x, T y, T p)` `Mem_Failed`

returns x^y mod `p`. If `p`=null, returns x^y. It is a c.r.e for `y`<0 or for a nonnull `p`<2.

`T AP_rshift(T x, int s)` `Mem_Failed`

returns `x` shifted right by `s` bits; vacated bits are filled with 0s, and the result has the same sign as `x`. It is a c.r.e. for `s`<0.

`T AP_sub(T x, T y)` `Mem_Failed`

`T AP_subi(T x, long int y)` `Mem_Failed`

return the difference $x - y$.

`long int AP_toint(T x)`

returns a long with same sign as `x` and magnitude $|x|$ mod `LONG_MAX`+1.

`char *AP_tostr(char *str, int size,` `Mem_Failed`
`int base, T x)`

fills `str[0..size-1]` with the character representation of `x` in `base` and returns `str`. If `str`=null, `AP_tostr` allocates it. Uppercase letters are used for digits that exceed 9 when `base`>10. It is c.r.e. for a nonnull `str` to be too small or for `base`<2 or `base`>36.

Arena

`T` is opaque `Arena_T`

It is a c.r.e. to pass `nbytes`≤0 or a null `T` to any `Arena` function.

```
void *Arena_alloc(T arena, int nbytes,                    Arena_Failed
    const char *file, int line)
```

allocates `nbytes` bytes in `arena` and returns a pointer to the first byte. The bytes are uninitialized. If `Arena_alloc` raises `Arena_Failed`, `file` and `line` are reported as the offending source coordinate.

```
void *Arena_calloc(T arena, int count,                    Arena_Failed
     int nbytes, const char *file, int line)
```

allocates space in `arena` for an array of `count` elements, each occupying `nbytes`, and returns a pointer to the first element. It is a c.r.e. for `count`≤0. The elements are uninitialized. If `Arena_calloc` raises `Arena_Failed`, `file` and `line` are reported as the offending source coordinate.

```
void Arena_dispose(T *ap)
```

deallocates *all* the space in `*ap`, deallocates the arena itself, and clears `*ap`. It is a c.r.e. for `ap` or `*ap` to be null.

```
void Arena_free(T arena)
```

deallocates *all* the space in `arena` — all the space allocated since the last call to `Arena_free`.

```
T Arena_new(void)                                      Arena_NewFailed
```

allocates, initializes, and returns a new arena.

Arith

```
int Arith_ceiling(int x, int y)
```

returns the least integer not less than the real quotient of x/y. It is an u.r.e. for y=0.

```
int Arith_div(int x, int y)
```

returns x/y, the maximum integer that does not exceed the real number z such that z·y = x. Truncates towards –∞; e.g., Arith_div(–13, 5) returns –3. It is an u.r.e. for y=0.

```
int Arith_floor(int x, int y)
```

returns the greatest integer not exceeding the real quotient of x/y. It is an u.r.e. for y=0.

```
int Arith_max(int x, int y)
```

returns max(x, y).

```
int Arith_min(int x, int y)
```

returns min(x, y).

```
int Arith_mod(int x, int y)
```

returns x – y·Arith_div(x, y); e.g., Arith_mod(–13, 5) returns 2. It is an u.r.e. for y=0.

Array

`T` is opaque `Array_T`

Array indices run from 0 to *N*–1, where *N* is the length of the array. The empty array has no elements. It is a c.r.e. to pass a null `T` to any `Array` function.

`T Array_copy(T array, int length)` — `Mem_Failed`

creates and returns a new array that holds the initial `length` elements from `array`. If `length` exceeds the length of `array`, the excess elements are cleared.

`void Array_free(T *array)`

deallocates and clears `*array`. It is a c.r.e. for `array` or `*array` to be null.

`void *Array_get(T array, int i)`

returns a pointer to the `i`th element in `array`. It is a c.r.e. for `i`<0 or `i`≥*N*, where *N* is the length of `array`.

`int Array_length(T array)`

returns the number of elements in `array`.

`T Array_new(int length, int size)` — `Mem_Failed`

allocates, initializes, and returns a new array of `length` elements each of `size` bytes. The elements are cleared. It is a c.r.e. for `length`<0 or `size`≤0.

`void *Array_put(T array, int i, void *elem)`

copies `Array_size(array)` bytes from `elem` into the `i`th element in `array` and returns `elem`. It is a c.r.e. for `elem`=null or for `i`<0 or `i`≥*N*, where *N* is the length of `array`.

`void Array_resize(T array, int length)` — `Mem_Failed`

changes the number of elements in `array` to `length`. If `length` exceeds the original length, the excess elements are cleared. It is a c.r.e. for `length`<0.

`int Array_size(T array)`

returns the size in bytes of the elements in `array`.

ArrayRep


```
typedef struct T {
      int length; int size; char *array; } *T;
```

It is an u.r.e. to change the fields in a `T`.

```
void ArrayRep_init(T array, int length,
    int size, void *ary)
```

initializes the fields in `array` to the values of `length`, `size`, and `ary`. It is a c.r.e. for `length`≠0 and `ary`=null, `length`=0 and `ary`≠null, or `size`≤0. It is an u.r.e. to initialize a `T` by other means.

Assert

`assert(e)`

raises `Assert_Failed` if `e` is 0. Syntactically, `assert(e)` is an expression. If `NDEBUG` is defined when `assert.h` is included, assertions are disabled.

Atom

It is a c.r.e. to pass a null `str` to any `Atom` function. It is an u.r.e. to modify an atom.

`int Atom_length(const char *str)`

returns the length of the atom `str`. It is a c.r.e. for `str` not to be an atom.

`char *Atom_new(const char *str, int len)` `Mem_Failed`

returns the atom for `str[0..len-1]`, creating one if necessary. It is a c.r.e. for `len`<0.

`char *Atom_string(const char *str)` `Mem_Failed`

returns `Atom_new(str, strlen(str))`.

`char *Atom_int(long n)` `Mem_Failed`

returns the atom for the decimal string representation of `n`.

Bit

`T` is opaque `Bit_T`

The bits in a bit vector are numbered 0 to $N-1$ where N is the length of the vector. It is a c.r.e to pass a null `T` to any `Bit` function, except for `Bit_union`, `Bit_inter`, `Bit_minus`, and `Bit_diff`.

`void Bit_clear(T set, int lo, int hi)`
: clears bits `lo`..`hi` in `set`. It is a c.r.e. for `lo`>`hi`, or for `lo`<0 or `lo`≥N where N is the length of `set`; likewise for `hi`.

`int Bit_count(T set)`
: returns the number of 1s in `set`.

`T Bit_diff(T s, T t)` — `Mem_Failed`
: returns the symmetric difference `s` / `t`: the exclusive OR of `s` and `t`. If `s`=null or `t`=null, it denotes the empty set. It is a c.r.e. for `s`=null and `t`=null, or for `s` and `t` to have different lengths.

`int Bit_eq(T s, T t)`
: returns 1 if `s` = `t` and 0 otherwise. It is a c.r.e. for `s` and `t` to have different lengths.

`void Bit_free(T *set)`
: deallocates and clears `*set`. It is a c.r.e. for `set` or `*set` to be null.

`int Bit_get(T set, int n)`
: returns bit `n`. It is a c.r.e. for `n`<0 or `n`≥N where N is the length of `set`.

`T Bit_inter(T s, T t)` — `Mem_Failed`
: returns `s` ∩ `t`: the logical AND of `s` and `t`. See `Bit_diff` for c.r.e.

`int Bit_length(T set)`
: returns the length of `set`.

`int Bit_leq(T s, T t)`
: returns 1 if `s` ⊆ `t` and 0 otherwise. See `Bit_eq` for c.r.e.

`int Bit_lt(T s, T t)`
: returns 1 if `s` ⊂ `t` and 0 otherwise. See `Bit_eq` for c.r.e.

`void Bit_map(T set,`
`    void apply(int n, int bit, void *cl), void *cl)`

calls `apply(n`, `bit`, `cl)` for each bit in `set` from 0 to *N*–1, where *N* is the length of `set`. Changes to `set` by `apply` affect subsequent values of `bit`.

`T Bit_minus(T s, T t)` `Mem_Failed`

returns `s` – `t`: the logical AND of `s` and ~`t`. See `Bit_diff` for c.r.e.

`T Bit_new(int length)` `Mem_Failed`

creates and returns a new bit vector of `length` 0s. It is a c.r.e. for `length`<0.

`void Bit_not(T set, int lo, int hi)`

complements bits `lo`..`hi` in `set`. See `Bit_clear` for c.r.e.

`int Bit_put(T set, int n, int bit)`

sets bit `n` to `bit` and returns the previous value of bit `n`. It is c.r.e. for `bit`<0 or `bit`>1, or for `n`<0 or `n`≥*N* where *N* is the length of `set`.

`void Bit_set(T set, int lo, int hi)`

sets bits `lo`..`hi` in `set`. See `Bit_clear` for c.r.e.

`T Bit_union(T s, T t)` `Mem_Failed`

returns `s` ∪ `t`: the inclusive OR of `s` and `t`. See `Bit_diff` for c.r.e.

Chan

T is opaque Chan_T

It is a c.r.e. to pass a null T to any Chan function, or to call any Chan function before calling Thread_init.

T Chan_new(void) — Mem_Failed

create, initialize, and return a new channel.

int Chan_receive(T c, void *ptr, int size) — Thread_Alerted

waits for a corresponding Chan_send, then copies up to size bytes from the sender to ptr, and returns the number copied. It is a c.r.e. for ptr=null or size<0.

int Chan_send(T c, const void *ptr, int size — Thread_Alerted

waits for a corresponding Chan_receive, then copies up to size bytes from ptr to the receiver, and returns the number copied. See Chan_receive for c.r.e.

Except


```
typedef struct T { char *reason; } T;
```

The syntax of TRY statements is as follows; *S* and *e* denote statements and exceptions. The ELSE clause is optional.

`TRY` S `EXCEPT(` e_1 `)` S_1 … `EXCEPT(` e_n `)` S_n `ELSE` S_0 `END_TRY`

`TRY` S `FINALLY` S_1 `END_TRY`

`void Except_raise(const T *e, const char *file, int line)`
: raises exception `*e` at source coordinate `file` and `line`. It is a c.r.e. for `e`=null. Uncaught exceptions cause program termination.

`RAISE(e)`
: raises `e`.

`RERAISE`
: reraises the exception that caused execution of a handler.

`RETURN`
`RETURN` *expression*
: return statement used within TRY statements. It is an u.r.e. to use a C return statement in TRY statements.

Fmt


```
typedef void (*T)(int code,
      va_list *app, int put(int c, void *cl), void *cl,
      unsigned char flags[256], int width, int precision)
```

defines the type of a conversion function, which is called by the `Fmt` functions when the associated conversion specifier appears in a format string. Here and below, `put(c, cl)` is called to emit each formatted character `c`. Table 14.1 (page 220) summarizes the initial set of conversion specifiers. It is a c.r.e to pass a null `put`, `buf`, `fmt`, or `ap` to any `Fmt` function, or for a format string to use a conversion specifier that has no associated conversion function.

```
char *Fmt_flags = "-+ 0"
```

points to the flag characters that can appear in conversion specifiers.

```
void Fmt_fmt(int put(int c, void *cl), void *cl,
    const char *fmt, ...)
```

formats and emits the "…" arguments according to the format string `fmt`.

```
void Fmt_fprint(FILE *stream, const char *fmt, ...)
void Fmt_print(const char *fmt, ...)
```

format and emit the "…" arguments according to `fmt`; `Fmt_fprint` writes to `stream`, `Fmt_print` writes to `stdout`.

```
void Fmt_putd(const char *str, int len,
    int put(int c, void *cl), void *cl,
    unsigned char flags[256], int width, int precision)
void Fmt_puts(const char *str, int len,
    int put(int c, void *cl), void *cl,
    unsigned char flags[256], int width, int precision)
```

format and emit the converted numeric (`Fmt_putd`) or string (`Fmt_puts`) in `str[0..len-1]` according to `Fmt`'s defaults (see Table 14.1, page 220) and the values of `flags`, `width`, and `precision`. It is a c.r.e for `str`=null, `len`<0, or `flags`=null.

```
T Fmt_register(int code, T cvt)
```

associates `cvt` with the format character `code`, and returns the previous conversion function. It is a c.r.e. for `code`<0 or `code`>255.

```
int Fmt_sfmt(char *buf, int size,                    Fmt_Overflow
    const char *fmt, ...)
```

formats the "…" arguments into `buf[1..size-1]` according to `fmt`, appends a null character, and returns the length of `buf`. It is a c.r.e. for `size`≤0. Raises `Fmt_Overflow` if more than `size`–1 characters are emitted.

```
char *Fmt_string(const char *fmt, ...)
```

formats the "…" arguments into a null-terminated string according to `fmt` and returns that string.

```
void Fmt_vfmt(int put(int c, void *cl), void *cl,
    const char *fmt, va_list ap)
```

See `Fmt_fmt`; takes arguments from the list `ap`.

```
int Fmt_vsfmt(char *buf, int size,                   Fmt_Overflow
    const char *fmt, va_list ap)
```

See `Fmt_sfmt`; takes arguments from the list `ap`.

`char *Fmt_vstring(const char *fmt, va_list ap)`

See `Fmt_string`; takes arguments from the list ap.

List

T **is** List_T

```
typedef struct T *T;
struct T { void *first; T rest; };
```

All List functions accept a null T for any list argument and interpret it as the empty list.

`T List_append(T list, T tail)`
appends tail to list and returns list. If list=null, List_append returns tail.

`T List_copy(T list)` Mem_Failed
creates and returns a top-level copy of list.

`void List_free(T *list)`
deallocates and clears *list. It is a c.r.e. for list=null.

`int List_length(T list)`
returns the number of elements in list.

`T List_list(void *x, ...)` Mem_Failed
creates and returns a list whose elements are the "…" arguments up to the first null pointer.

```
void List_map(T list,
    void apply(void **x, void *cl), void *cl)
```
calls apply(&p->first, cl) for each element p in list. It is an u.r.e. for apply to change list.

`T List_pop(T list, void **x)`
assigns list->first to *x, if x is nonnull, deallocates list, and returns list->rest. If list=null, List_pop returns null and does not change *x.

`T List_push(T list, void *x)` Mem_Failed
adds a new element holding x onto the front of list and returns the new list.

`T List_reverse(T list)`

reverses the elements in `list` inplace and returns the reversed list.

`void **List_toArray(T list, void *end)` `Mem_Failed`

creates an N+1-element array of the N elements in `list` and returns a pointer to its first element. The Nth element in the array is `end`.

Mem

It is c.r.e. to pass `nbytes`≤0 to any `Mem` function or macro.

`ALLOC(nbytes)` `Mem_Failed`

allocates `nbytes` bytes and returns a pointer to the first byte. The bytes are uninitialized.

`CALLOC(count, nbytes)` `Mem_Failed`

allocates space for an array of `count` elements, each occupying `nbytes` bytes and returns a pointer to the first element. It is a c.r.e. for `count`≤0. The elements are uninitialized.

`FREE(ptr)`

See `Mem_free`.

```
void *Mem_alloc(int nbytes,
    const char *file, int line)
```
`Mem_Failed`

allocates `nbytes` bytes and returns a pointer to the first byte. The bytes are uninitialized. If `Mem_alloc` raises `Mem_Failed`, `file` and `line` are reported as the offending source coordinate.

```
void *Mem_calloc(int count, int nbytes,
    const char *file, int line)
```
`Mem_Failed`

allocates space for an array of `count` elements, each occupying `nbytes` and returns a pointer to the first element. It is a c.r.e. for `count`≤0. The elements are uninitialized. If `Mem_calloc` raises `Mem_Failed`, `file` and `line` are reported as the offending source coordinate.

```
void Mem_free(void **ptr, const char *file, int line)
```

deallocates `*ptr`, if `*ptr` is nonnull, and clears `*ptr`. It is a c.r.e. for `ptr`=null, and it is an u.r.e. for `*ptr` to be a pointer that was not returned by previous call to a `Mem` allocation function. Implementations may use `file` and `line` to report memory usage errors.

`void *Mem_resize(void **ptr, int nbytes,` `Mem_Failed`
`const char *file, int line)`

changes the size of the block at `*ptr` to hold `nbytes` bytes, clears `*ptr`, and returns a pointer to the first byte of the new block. If `nbytes` exceeds the size of the original block, the excess byte are uninitialized. If `nbytes` is less than the size of the original block, only `nbytes` of its bytes appear in the new block. If `Mem_resize` raises `Mem_Failed`, `file` and `line` are reported as the offending source coordinate. It is a c.r.e. for `ptr`=null or `*ptr`=null, and it is an u.r.e. for `*ptr` to be a pointer that was not returned by a previous call to a `Mem` allocation function.

`NEW(p)` `Mem_Failed`

`NEW0(p)` `Mem_Failed`

allocate a block large enough to hold `*p` and return a pointer to the first byte. `NEW0` clears the bytes, `NEW` leaves them uninitialized.

`RESIZE(ptr, nbytes)` `Mem_Failed`

See `Mem_resize`.

MP

`T` **is** `MP_T`

```
typedef unsigned char *T
```

`MP` functions do *n*-bit signed and unsigned arithmetic, where *n* is initially 32 and can be changed by `MP_set`. Function names that end in `u` or `ui` do unsigned arithmetic; others do signed arithmetic. `MP` functions compute their results before raising `MP_Overflow` or `MP_DivideByZero`. It is a c.r.e. to pass a null `T` to any `MP` function. It is an u.r.e. to pass a `T` that is too small to any `MP` function.

```
T MP_add(T z, T x, T y)                              MP_Overflow
T MP_addi(T z, T x, long y)                          MP_Overflow
T MP_addu(T z, T x, T y)                             MP_Overflow
T MP_addui(T z, T x, unsigned long y)                MP_Overflow
```

set `z` to `x` + `y` and return `z`.

```
T MP_and(T z, T x, T y)
T MP_andi(T z, T x, unsigned long y)
```

set `z` to `x` AND `y` and return `z`.

```
T MP_ashift(T z, T x, int s)
```

sets `z` to `x` shifted right by `s` bits and returns `z`. Vacated bits are filled with `x`'s sign bit. It is a c.r.e. for `s`<0.

```
int MP_cmp(T x, T y)
int MP_cmpi(T x, long y)
int MP_cmpu(T x, T y)
int MP_cmpui(T x, unsigned long y)
```

return an int <0, =0, or >0 if `x`<`y`, `x`=`y`, or `x`>`y`.

`T MP_cvt(int m, T z, T x)` — `MP_Overflow`
`T MP_cvtu(int m, T z, T x)` — `MP_Overflow`

narrow or widen `x` to an `m`-bit signed or unsigned integer in `z` and return `z`. It is a c.r.e. for `m`<2.

`T MP_div(T z, T x, T y)` — `MP_Overflow`, `MP_DivideByZero`
`T MP_divi(T z, T x, long y)` — `MP_Overflow`, `MP_DivideByZero`
`T MP_divu(T z, T x, T y)` — `MP_DivideByZero`
`T MP_divui(T z, T x, unsigned long y)` — `MP_Overflow`, `MP_DivideByZero`

set `z` to `x`/`y` and return `z`. The signed functions truncate towards –∞; see `Arith_div.`

```
void MP_fmt(int code, va_list *app,
    int put(int c, void *cl), void *cl,
    unsigned char flags[], int width, int precision)
void MP_fmtu(int code, va_list *app,
    int put(int c, void *cl), void *cl,
    unsigned char flags[], int width, int precision)
```

are `Fmt` conversion functions. They consume a `T` and a base *b* and format it like `printf`'s `%d` and `%u`. It is c.r.e. for the *b*<2 or *b*>36, and for `app`, `*app`, or `flags` to be null.

`T MP_fromint(T z, long v)` — `MP_Overflow`
`T MP_fromintu(T z, unsigned long u)` — `MP_Overflow`

set `z` to `v` or `u` and return `z`.

`T MP_fromstr(T z, const char *str, int base, char **end)` — `MP_Overflow`

interprets `str` as an integer in `base`, sets `z` to that integer, and returns `z`. See `AP_fromstr`.

`T MP_lshift(T z, T x, int s)`

set `z` to `x` shifted left by `s` bits and return `z`. Vacated bits are filled with 0s. It is a c.r.e. for `s`<0.

`T MP_mod(T z, T x, T y)` `MP_Overflow`, `MP_DivideByZero`

sets `z` to `x` mod `y` and returns `z`. Truncates towards –∞; see `Arith_mod`.

`long MP_modi(T x, long y)` `MP_Overflow`, `MP_DivideByZero`

returns `x` mod `y`. Truncates towards –∞; see `Arith_mod`.

`T MP_modu(T z, T x, T y)` `MP_DivideByZero`

sets `z` to `x` mod `y` and returns `z`.

`unsigned long MP_modui(T x, unsigned long y)` `MP_Overflow`, `MP_DivideByZero`

returns `x` mod `y`.

`T MP_mul(T z, T x, T y)` `MP_Overflow`

sets `z` to `x`·`y` and returns `z`.

`T MP_mul2(T z, T x, T y)` `MP_Overflow`

`T MP_mul2u(T z, T x, T y)` `MP_Overflow`

set `z` to the *double-length* result of `x`·`y` and return `z`, which has 2*n* bits.

`T MP_muli(T z, T x, long y)` `MP_Overflow`

`T MP_mulu(T z, T x, T y)` `MP_Overflow`

`T MP_mului(T z, T x, unsigned long y)` `MP_Overflow`

set `z` to `x`·`y` and return `z`.

`T MP_neg(T z, T x)` `MP_Overflow`

sets `z` to –`x` and returns `z`.

`T MP_new(unsigned long u)` `Mem_Failed`, `MP_Overflow`

creates and returns a `T` initialized to `u`.

`T MP_not(T z, T x)`

sets `z` to ~`x` and returns `z`.

`T MP_or(T z, T x, T y)`
`T MP_ori(T z, T x, unsigned long y)`

set `z` to `x` OR `y` and return `z`.

`T MP_rshift(T z, T x, int s)`

sets `z` to `x` shifted right by `s` bits and returns `z`. Vacated bits are filled with 0s. It is a c.r.e. for `s`<0.

`int MP_set(int n)` — `Mem_Failed`

resets `MP` to do `n`-bit arithmetic. It is a c.r.e. for `n`<2.

`T MP_sub(T z, T x, T y)` — `MP_Overflow`
`T MP_subi(T z, T x, long y)` — `MP_Overflow`
`T MP_subu(T z, T x, T y)` — `MP_Overflow`
`T MP_subui(T z, T x, unsigned long y)` — `MP_Overflow`

set `z` to `x` – `y` and return `z`.

`long int MP_toint(T x)` — `MP_Overflow`
`unsigned long MP_tointu(T x)` — `MP_Overflow`

return `x` as a long int or unsigned long.

`char *MP_tostr(char *str, int size, int base, T x)` — `Mem_Failed`

fills `str[0..size-1]` with a null-terminated string representing `x` in `base`, and returns `str`. If `str`=null, `MP_tostr` ignores `size` and allocates the string. See `AP_tostr`.

`T MP_xor(T z, T x, T y)`
`T MP_xori(T z, T x, unsigned long y)`

set `z` to `x` XOR `y` and return `z`.

Ring

`T` **is opaque** `Ring_T`

Ring indices run from 0 to *N*–1, where *N* is the length of the ring. The empty ring has no elements. Pointers can be added or removed anywhere; rings expand automatically. Rotating a ring changes its origin. It is a c.r.e. to passed a null `T` to any `Ring` function.

`void *Ring_add(T ring, int pos, void *x)` `Mem_Failed`

inserts `x` at *position* `pos` in `ring` and returns `x`. Positions identify points between elements; see `Str`. It is a c.r.e. for `pos` < –*N* or `pos` > *N*+1, where *N* is the length of `ring`.

`void *Ring_addhi(T ring, void *x)` `Mem_Failed`

`void *Ring_addlo(T ring, void *x)` `Mem_Failed`

adds `x` to the high (index *N*–1) or low (index 0) end of `ring` and returns `x`.

`void Ring_free(T *ring)`

deallocates and clears `*ring`. It is a c.r.e. for `ring` or `*ring` to be null.

`int Ring_length(T ring)`

returns the number of elements in `ring`.

`void *Ring_get(T ring, int i)`

returns the `i`th element in ring. It is a c.r.e. for `i`<0 or `i`≥*N*, where *N* is the length of `ring`.

`T Ring_new(void)` `Mem_Failed`

creates and returns an empty ring.

`void *Ring_put(T ring, int i, void *x)` `Mem_Failed`

changes the `i`th element in `ring` to `x` and returns the previous value. See `Ring_get` for c.r.e.

`void *Ring_remhi(T ring)`

`void *Ring_remlo(T ring)`

removes and returns the element at the high end (index *N*–1) or low end (index 0) of `ring`. It is a c.r.e. for `ring` to be empty.

`void *Ring_remove(T ring, int i)`

removes and returns element `i` from `ring`. It is a c.r.e. for `i`<0 or `i`≥N, where *N* is the length of `ring`.

`T Ring_ring(void *x, ...)` `Mem_Failed`

creates and returns a ring whose elements are the "…" arguments up to the first null pointer.

`void Ring_rotate(T ring, int n)`

rotates the origin of `ring` `n` elements left (`n`<0) or right (`n`≥0). It is a c.r.e. for |`n`|<0 or |`n`|>N, where *N* is the length of `ring`.

Sem

`T` is opaque `Sem_T`

```
typedef struct T { int count; void *queue; } T;
```

It is an u.r.e. error to read or write the fields in a `T` directly, or to pass an uninitialized `T` to any `Sem` function. It is a c.r.e. to pass a null `T` to any `Sem` function, or to call any `Sem` function before calling `Thread_init`.

The syntax of the LOCK statement is as follows; *S* and *m* denote statements and a `T`.

`LOCK(`*m*`)` *S* `END_LOCK`

m is locked, statements *S* are executed and *m* is unlocked. LOCK can raise `Thread_Alerted`.

`void Sem_init(T *s, int count)`

sets `s->count` to `count`. It is an u.r.e. to call `Sem_init` more than once on the same `T`.

`Sem_T *Sem_new(int count)` — `Mem_Failed`

creates and returns a `T` with its `count` field initialized to `count`.

`void Sem_wait(T *s)` — `Thread_Alerted`

wait until `s->count`>0, then decrements `s->count`.

`void Sem_signal(T *s)` — `Thread_Alerted`

increments `s->count`.

Seq

`T` is opaque `Seq_T`

Sequence indices run from 0 to *N*–1, where *N* is the length of the sequence. The empty sequence has no elements. Pointers can be added or removed from the low end (index 0) or the high end (index *N*–1); sequences expand automatically. It is a c.r.e. to passed a null `T` to any `Seq` function.

`void *Seq_addhi(T seq, void *x)` — `Mem_Failed`
`void *Seq_addlo(T seq, void *x)` — `Mem_Failed`

adds `x` to the high or low end of `seq` and returns `x`.

`void Seq_free(T *seq)`

deallocates and clears `*seq`. It is a c.r.e. for `seq` or `*seq` to be null.

`int Seq_length(T seq)`

returns the number of elements in `seq`.

`void *Seq_get(T seq, int i)`

returns the `i`th element in `seq`. It is a c.r.e. for `i`<0 or `i`≥*N*, where *N* is the length of `seq`.

`T Seq_new(int hint)` — `Mem_Failed`

creates and returns an empty sequence. `hint` is an estimate of the maximum size of the sequence. It is c.r.e for `hint`<0.

`void *Seq_put(T seq, int i, void *x)`

changes the `i`th element in `seq` to `x` and returns the previous value. See `Seq_get` for c.r.e.

`void *Seq_remhi(T seq)`
`void *Seq_remlo(T seq)`

remove and return the element at the high or low end of `seq`. It is a c.r.e. for `seq` to be empty.

`T Seq_seq(void *x, ...)` — `Mem_Failed`

creates and returns a sequence whose elements are the “...” arguments up to the first null pointer.

Set

`T` is opaque `Set_T`

It is a c.r.e. to pass a null `member` or `T` to any `Set` function, except for `Set_diff`, `Set_inter`, `Set_minus`, and `Set_union`, which interpret a null `T` as the empty set.

`T Set_diff(T s, T t)` — `Mem_Failed`

returns the symmetric difference `s` / `t`: a set whose members appear in only one of `s` or `t`. It is a c.r.e. for both `s`=null and `t`=null, or for nonnull `s` and `t` have different `cmp` and `hash` functions.

`void Set_free(T *set)`

deallocates and clears `*set`. It is a c.r.e. for `set` or `*set` to be null.

`T Set_inter(T s, T t)` — `Mem_Failed`

returns `s` ∩ `t`: a set whose members appears in `s` and `t`. See `Set_diff` for c.r.e.

`int Set_length(T set)`

returns the number of elements in `set`.

```
void Set_map(T set,
    void apply(const void *member, void *cl), void *cl)
```

calls `apply(member, cl)` for each `member` ∈ `set`. It is a c.r.e. for `apply` to change `set`.

`int Set_member(T set, const void *member)`

returns 1 if `member` ∈ `set` and 0 otherwise.

`T Set_minus(T s, T t)` — `Mem_Failed`

returns the difference `s` – `t`: a set whose members appear in `s` but not in `t`. See `Set_diff` for c.r.e.

```
T Set_new(int hint,
    int cmp(const void *x, const void *y),
    unsigned hash(const void *x))
```

`Mem_Failed`

creates, initializes, and returns an empty set. See `Table_new` for an explanation of `hint`, `cmp`, and `hash`.

`void Set_put(T set, const void *member)` Mem_Failed

adds `member` to `set`, if necessary.

`void *Set_remove(T set, const void *member)`

removes `member` from `set`, if `member` ∈ `set`, and returns the removed member; otherwise, `Set_remove` returns null.

`void **Set_toArray(T set, void *end)` Mem_Failed

creates a *N*+1-element array that holds the *N* members in `set` in an unspecified order and returns a pointer to the first element. Element *N* is `end`.

`T Set_union(T s, T t)` Mem_Failed

returns `s` ∪ `t`: a set whose members appear in `s` or `t`. See `Set_diff` for c.r.e.

Stack

`T` is opaque `Stack_T`

It is a c.r.e. to pass null `T` to any `Stack` function.

`int Stack_empty(T stk)`
: returns 1 if `stk` is empty and 0 otherwise.

`void Stack_free(T *stk)`
: deallocates and clears `*stk`. It is a c.r.e. for `stk` or `*stk` to be null.

`T Stack_new(void)` `Mem_Failed`
: returns a new, empty `T`.

`void *Stack_pop(T stk)`
: pops and returns the top element on `stk`. It is a c.r.e. for `stk` to be empty.

`void Stack_push(T stk, void *x)` `Mem_Failed`
: pushes `x` onto `stk`.

Str

The `Str` functions manipulated null-terminated strings. Positions identify points between characters; e.g., the positions in STRING are

$$ {}^{1}_{-6}\mathrm{S}\,{}^{2}_{-5}\mathrm{T}\,{}^{3}_{-4}\mathrm{R}\,{}^{4}_{-3}\mathrm{I}\,{}^{5}_{-2}\mathrm{N}\,{}^{6}_{-1}\mathrm{G}\,{}^{7}_{0} $$

Two positions can be given in either order. `Str` functions that create strings allocate space for their results. In the descriptions below, `s[i:j]` denotes the substring of `s` between positions `i` and `j`. It is a c.r.e. to pass a nonexistent position or a null character pointer to any `Str` function, except as specified for `Str_catv` and `Str_map`.

```
int Str_any(const char *s, int i, const char *set)
```

returns the positive position in `s` after `s[i:i+1]` if that character appears in `set`, or 0 otherwise. It is a c.r.e. for `set`=null.

```
char *Str_cat(const char *s1, int i1, int j1,                Mem_Failed
    const char *s2, int i2, int j2)
```

returns `s1[i1:j1]` concatenated with `s2[i2:j2]`.

```
char *Str_catv(const char *s, ...)                           Mem_Failed
```

returns a string consisted of the triples in "…" up to a null pointer. Each triple specifies an `s[i:j]`.

```
int Str_chr(const char *s, int i, int j, int c)
```

returns the position in `s` before the leftmost occurrence of `c` in `s[i:j]`, or 0 otherwise.

```
int Str_cmp(const char *s1, int i1, int j1,
    const char *s2, int i2, int j2)
```

returns an integer <0, =0, or >0 if `s1[i1:j1]`<`s2[i2:j2]`, `s1[i1:j1]`=`s2[i2:j2]`, or `s1[i1:j1]`>`s2[i2:j2]`.

`char *Str_dup(const char *s, int i, int j, int n)` — Mem_Failed

returns `n` copies of `s[i:j]`. It is a c.r.e. for `n`<0.

`int Str_find(const char *s, int i, int j, const char *str)`

returns the position in `s` before the leftmost occurrence of `str` in `s[i:j]`, or 0 otherwise. It is a c.r.e. for `str`=null.

`void Str_fmt(int code, va_list *app, int put(int c, void *cl), void *cl, unsigned char flags[], int width, int precision)`

is a `Fmt` conversion function. It consumes 3 arguments: a string and two positions and formats the substring in the style of `printf`'s `%s`. It is a c.r.e. for `app`, `*app`, or `flags` to be null.

`int Str_len(const char *s, int i, int j)`

returns the length of `s[i:j]`.

`int Str_many(const char *s, int i, int j, const char *set)`

returns the positive position in `s` after a nonempty run of characters from `set` at the beginning of `s[i:j]`, or 0 otherwise. It is c.r.e. for `set`=null.

`char *Str_map(const char *s, int i, int j, const char *from, const char *to)` — Mem_Failed

returns the string obtained from mapping the characters in `s[i:j]` according to `from` and `to`. Each character from `s[i:j]` that appears in `from` is mapped to the corresponding character in `to`. Characters that do not appear in `from` map to themselves. If `from`=null and `to`=null, their previous values are used. If `s`=null, `from` and `to` establish a default mapping. It is a c.r.e. for only one of `from` or `to` to be null, for `strlen(from)`≠`strlen(to)`, for `s`, `from`, and `to` to all be null, or for `from`=`null` and `to`=null on the first call.

`int Str_match(const char *s, int i, int j, const char *str)`

returns the positive position in `s` if `s[i:j]` starts with `str`, or 0 otherwise. It is a c.r.e. for `str`=null.

`int Str_pos(const char *s, int i)`
returns the positive position corresponding to `s[i:i]`; subtracting 1 yields the index of `s[i:i+1]`.

`int Str_rchr(const char *s, int i, int j, int c)`
is the rightmost variant of `Str_chr`.

`char *Str_reverse(const char *s, int i, int j)` `Mem_Failed`
returns a copy of `s[i:j]` with the characters in the opposite order.

`int Str_rfind(const char *s, int i, int j, const char *str)`
is the rightmost variant of `Str_find`.

`int Str_rmany(const char *s, int i, int j, const char *set)`
returns the positive position in `s` before a nonempty run of characters from `set` at the end of `s[i:j]`, or 0 otherwise. It is c.r.e. for `set`=null.

`int Str_rmatch(const char *s, int i, int j,`
`const char *str)`
returns the positive position in `s` before `str` if `s[i:j]` ends with `str`, or 0 otherwise. It is a c.r.e. for `str`=null.

`int Str_rupto(const char *s, int i, int j, const char *set)`
is the rightmost variant of `Str_upto`.

`char *Str_sub(const char *s, int i, int j)` `Mem_Failed`
returns `s[i:j]`.

`int Str_upto(const char *s, int i, int j, const char *set)`
returns the position in `s` before the leftmost occurrence in `s[i:j]` of any character in `set`, or 0 otherwise. It is c.r.e. for `set`=null.

Table

`T` is opaque `Table_T`

It is a c.r.e. to pass a null `T` or a null `key` to any `Table` function.

```
void Table_free(T *table)
```
deallocates and clears `*table`. It is a c.r.e. for `table` or `*table` to be null.

```
void *Table_get(T table, const void *key)
```
returns the value associated with `key` in `table`, or null if `table` does not hold key.

```
int Table_length(T table)
```
returns the number of key-value pairs in `table`.

```
void Table_map(T table,
    void apply(const void *key, void **value, void *cl),
    void *cl)
```
calls `apply(key, &value, cl)` for each key-value in `table` in an unspecified order. It is a c.r.e. for `apply` to change `table`.

```
T Table_new(int hint,                                          Mem_Failed
    int cmp(const void *x, const void *y),
    unsigned hash(const void *key))
```
creates, initializes, and returns a new, empty table that can hold an arbitrary number of key-value pairs. `hint` is an estimate of the number such pairs expected. It is a c.r.e. for `hint`<0. `cmp` and `hash` are functions for comparing and hashing keys. For keys `x` and `y`, `cmp(x,y)` must return an int <0, =0, or >0 if `x`<`y`, `x`=`y`, or `x`>`y`. If `cmp(x,y)` returns 0, then `hash(x)` must equal `hash(y)`. If `cmp`=null or `hash`=null, `Table_new` uses a function suitable for `Atom_T` keys.

```
void *Table_put(T table,                                       Mem_Failed
    const void *key, void *value)
```
changes the value associated with `key` in `table` to `value` and returns the previous value, or adds `key` and `value` if `table` does not hold `key`, and returns null.

`void *Table_remove(T table, const void *key)`

removes the `key`-value pair from `table` and returns the removed value. If `table` does not hold `key`, `Table_remove` has no effect and returns null.

`void **Table_toArray(T table, void *end)` `Mem_Failed`

creates a $2N$+1-element array that holds the N key-value pairs in `table` in an unspecified order and returns a pointer to the first element. The keys appear in the even-numbered array elements and the corresponding values appear in the following odd-numbered elements, and element $2N$ is `end`.

Text

`T` is `Text_T`

```
typedef struct T { int len; const char *str; } T;
typedef struct Text_save_T *Text_save_T;
```

A `T` is a descriptor; clients can read the fields of a descriptor, but it is an u.r.e. to write them. `Text` functions accept and return descriptors *by value*; it is a c.r.e. to pass a descriptor with `str`=null or `len`<0 to any `Text` function.

`Text` manages the memory for its immutable strings; it is an u.r.e. to write this string space or deallocate it by external means. Strings in string space are not terminated by null characters, because they can contain null characters.

Some `Text` functions accept positions, which identify points between characters; see `Str`. In the descriptions below, `s[i:j]` denotes the substring in `s` between positions `i` and `j`.

```
const T Text_cset   = { 256, "\000\001...\376\377" }
const T Text_ascii  = { 128, "\000\001...\176\177" }
const T Text_ucase  = {  26, "ABCDEFGHIJKLMNOPQRSTUVWXYZ" }
const T Text_lcase  = {  26, "abcdefhijklmnopqrtuvwxyz" }
const T Text_digits = {  10, "0123456789" }
const T Text_null   = {   0, "" }
```

are static descriptors initialized as shown.

```
int Text_any(T s, int i, T set)
```

returns the positive position in `s` after `s[i:i+1]` if that character appears in `set`, or 0 otherwise.

```
T Text_box(const char *str, int len)
```

builds and returns a descriptor for the client-allocated string `str` of length `len`. It is a c.r.e. for `str`=null or `len`<0.

`T Text_cat(T s1, T s2)` `Mem_Failed`

returns `s1` concatenated with `s2`.

`int Text_chr(T s, int i, int j, int c)`

See `Str_chr`.

`int Text_cmp(T s1, T s2)`

returns an int <0, =0, or >0 if `s1`<`s2`, `s1`=`s2`, or `s1`>`s2`.

`T Text_dup(T s, int n)` `Mem_Failed`

returns `n` copies of `s`. It is a c.r.e. for `n`<0.

`int Text_find(T s, int i, int j, T str)`

See `Str_find`.

```
void Text_fmt(int code, va_list *app,
    int put(int c, void *cl), void *cl,
    unsigned char flags[], int width, int precision)
```

is a `Fmt` conversion function. It consumes a *pointer* to a descriptor and formats the string in the style of `printf`'s `%s`. It is a c.r.e. for the descriptor pointer, `app`, `*app`, or `flags` to be null.

`char *Text_get(char *str, int size, T s)`

copies `s.str[0..str.len-1]` to `str[0..size-1]`, appends a null, and returns `str`. If `str`=null, `Text_get` allocates the space. It is a c.r.e. for `str`≠null and `size`<`s.len`+1.

`int Text_many(T s, int i, int j, T set)`

See `Str_many`.

`T Text_map(T s, const T *from, const T *to)` `Mem_Failed`

returns the string obtained from mapping the characters in `s` according to `from` and `to`; see `Str_map`. If `from`=null and `to`=null, their previous values are used. It is a c.r.e for only one of `from` or `to` to be null, or for `from->len`≠`to->len`.

`int Text_match(T s, int i, int j, T str)`

See `Str_match`.

`int Text_pos(T s, int i)`

See `Str_pos`.

`T Text_put(const char *str)` `Mem_Failed`

copies the null-terminated `str` into string space and returns its descriptor. It is a c.r.e. for `str`=null.

`int Text_rchr(T s, int i, int j, int c)`

See `Str_rchr`.

`void Text_restore(Text_save_T *save)`

pops the string space to the point denoted by `save`. It is a c.r.e. for `save`=null. It is an u.r.e. to use other `Text_save_T` values that denote locations higher than `save` after calling `Text_restore`.

`T Text_reverse(T s)` `Mem_Failed`

returns a copy of `s` with the characters in the opposite order.

`int Text_rfind(T s, int i, int j, T str)`

See `Str_rfind`.

`int Text_rmany(T s, int i, int j, T set)`

See `Str_rmany`.

`int Text_rmatch(T s, int i, int j, T str)`

See `Str_rmatch`.

`int Text_rupto(T s, int i, int j, T set)`

See `Str_rupto`.

`Text_save_T Text_save(void)` `Mem_Failed`

returns an opaque pointer that encodes the current top of the string space.

`T Text_sub(T s, int i, int j)`

returns `s[i:j]`.

`int Text_upto(T s, int i, int j, T set)`

See `Str_upto`.

Thread

`T` **is opaque** `Thread_T`

It is a c.r.e. to call any `Thread` function before calling `Thread_init`.

`void Thread_alert(T t)`

sets `t`'s alert-pending flag and makes `t` runnable. The next time `t` runs, or calls a blocking `Thread`, `Sem`, or `Chan` primitive, it clears its flag and raises `Thread_Alerted`. It is a c.r.e. for `t`=null or to name a nonexistent thread.

`void Thread_exit(int code)`

terminates the calling thread and passes `code` to any threads waiting for the calling thread to terminate. When the last thread calls `Thread_exit`, the program terminates with `exit(code)`.

`int Thread_init(int preempt, ...)`

initializes the `Thread` for nonpreemptive (`preempt`=0) or preemptive (`preempt`=1) scheduling and returns `preempt` or 0 if `preempt`=1 and preemptive scheduling is not supported. `Thread_init` may accept additional implementation-defined parameters; the argument list must be terminated with a null. It is c.r.e. to call `Thread_init` more than once.

`int Thread_join(T t)` `Thread_Alerted`

suspends the calling thread until thread `t` terminates. When `t` terminates, `Thread_join` returns `t`'s exit code. If `t`=null, the calling thread waits for all other threads to terminate, and then returns 0. It is a c.r.e. for `t` to name the calling thread or for more than one thread to pass a null `t`.

```
T Thread_new(int apply(void *),                          Thread_Failed
    void *args, int nbytes, ...)
```

creates, initializes, and starts a new thread, and returns its handle. If `nbytes`=0, the new thread executes `Thread_exit(apply(args))`, otherwise, it executes `Thread_exit(apply(p))`, where `p` points to a *copy* of the `nbytes` block starting at `args`. The new thread starts with its own, empty exception stack. `Thread_new` may accept additional implementation-defined parameters; the argument list must be terminated with a null. It is a c.r.e. for `apply`=null, or for `args`=null and `nbytes`<0.

`void Thread_pause(void)`

relinquishes the processor another thread, perhaps the calling thread.

`T Thread_self(void)`

returns the calling thread’s handle.

```
typedef unsigned char *T;
```

An extended-precision unsigned integer is represented in base 2^8 by an array of *n* digits, least significant digit first. Most XP functions take *n* as an argument along with source and destination Ts; it is an u.r.e. for *n*<1 or for *n* not to be the length of the corresponding Ts. It is an u.r.e. to pass a null T or a T that is too small to any XP function.

`int XP_add(int n, T z, T x, T y, int carry)`
: sets `z[0..n-1]` to `x` + `y` + `carry` and returns the carry out of `z[n-1]`. `carry` must be 0 or 1.

`int XP_cmp(int n, T x, T y)`
: returns an int <0, =0, or >0 if `x`<`y`, `x`=`y`, or `x`>`y`.

`int XP_diff(int n, T z, T x, int y)`
: sets `z[0..n-1]` to `x` – `y`, where `y` is a single digit, and returns the borrow into `z[n-1]`. It is an u.r.e. for `y`>2^8.

`int XP_div(int n, T q, T x, int m, T y, T r, T tmp)`
: sets `q[0..n-1]` to `x[0..n-1]`/`y[0..m-1]`, `r[0..m-1]` to `x[0..n-1]` mod `y[0..m-1]`, and returns 1, if `y`≠0. If `y`=0, `XP_div` returns 0 and leaves `q` and `r` unchanged. `tmp` must hold at least `n`+`m`+2 digits. It is an u.r.e. for `q` or `r` to be one of `x` or `y`, for `q` and `r` to be the same `T`, or for `tmp` to be too small.

`unsigned long XP_fromint(int n, T z, unsigned long u)`
: sets `z[0..n-1]` to `u` mod 2^{8n} and returns `u`/2^{8n}.

`int XP_fromstr(int n, T z, const char *str, int base, char **end)`

interprets `str` as an unsigned integer in `base` using `z[0..n-1]` as the initial value in the conversion, and returns the first nonzero carry out of the conversion step. If `end`≠null, `*end` points to the character in `str` that terminated the scan or produced a nonzero carry. See `AP_fromstr`.

`int XP_length(int n, T x)`

returns the length of `x`; that is, the index plus one of the most significant nonzero digit in `x[0..n-1]`.

`void XP_lshift(int n, T z, int m, T x, int s, int fill)`

sets `z[0..n-1]` to `x[0..m-1]` shifted left by `s` bits, and fills the vacated bits with `fill`, which must be 0 or 1. It is an u.r.e. for `s`<0.

`int XP_mul(T z, int n, T x, int m, T y)`

adds `x[0..n-1]`·`y[0..m-1]` to `z[0..n+m-1]` and returns the carry out of `z[n+m-1]`. If `z`=0, `XP_mul` computes `x`·`y`. It is an u.r.e. for `z` to be the same `T` as `x` or `y`.

`int XP_neg(int n, T z, T x, int carry)`

sets `z[0..n-1]` to ~`x` + `carry`, where `carry` is 0 or 1, and returns the carry out of `z[n-1]`.

`int XP_product(int n, T z, T x, int y)`

sets `z[0..n-1]` to `x`·`y`, where `y` is a single digit, and returns the carry out of `z[n-1]`. It is an u.r.e. for $y \geq 2^8$.

`int XP_quotient(int n, T z, T x, int y)`

sets `z[0..n-1]` to `x`/`y`, where `y` is a single digit, and returns `x` mod `y`. It is an u.r.e. for `y`=0 or $y \geq 2^8$.

`void XP_rshift(int n, T z, int m, T x, int s, int fill)`

right shift; see `XP_lshift`. If `n`>`m`, the excess bits are treated as if they were equal to `fill`.

`int XP_sub(int n, T z, T x, T y, int borrow)`

sets `z[0..n-1]` to `x` – `y` – `borrow` and returns the borrow into `z[n-1]`. `borrow` must be 0 or 1.

`int XP_sum(int n, T z, T x, int y)`

sets `z[0..n-1]` to `x` + `y`, where `y` is a single digit, and returns the carry out of `z[n-1]`. It is an u.r.e. for `y`$>2^8$.

`unsigned long XP_toint(int n, T x)`

returns `x` mod (`ULONG_MAX`+1).

`char *XP_tostr(char *str, int size, int base, int n, T x)`

fills `str[0..size-1]` with the character representation of `x` in `base`, sets `x` to 0, and returns `str`. It is a c.r.e. for `str`=null, `size` to be too small, or for `base`<2 or `base`>36.