**Standard Library**

This appendix is a summary of the library defined by the ANSI standard. The standard library is not part of the C language proper, but an environment that supports standard C will provide the function declarations and type and macro definitions of this library. We have omitted a few functions that are of limited utility or easily synthesized from others; we have omitted multi-byte characters; and we have omitted discussion of locale issues, that is, properties that depend on local language, nationality, or culture.

The functions, types and macros of the standard library are declared in standard

*headers:*

<assert.h> <float.h>

<ctype.h> <limits.h>

<errno.h> <locale.h>

A header can be accessed by

#include *<header>*

<math.h>

<setjmp.h>

<signal.h>

<stdarg.h>

<stddef.h>

<stdio.h>

<stdlib.h>

<string.h>

<time.h>

Headers may be included in any order and any number of times. A header must be included outside of any external declaration or definition and before any use of anything it declares. A header need not be a source file.

External identifiers that begin with an underscore are reserved for use by the library, as are all other identifiers that begin with an underscore and an upper-case letter or another underscore.

**B1. Input and Output: <stdlo.h>**

The input and output functions, types, and macros defined in <stdio.·h> represent nearly one third of the library.

A *stream* is a source or destination of data that may be associated with a disk or other peripheral. The library supports text streams and binary streams, although on some systems, notably UNIX, these are identical. A text stream is a sequence of lines; each line bas zero or more characters and is terminated by '\n'. An environment may need to convert a text stream to or from some other representation (such as mapping '\n' to carriage return and linefeed). A binary stream is a sequence of unprocessed bytes that record internal data, with the property that if it is written, then read back on the same system, it will compare equal.

A stream is connected to a file or device by *opening* it; the connection is broken by

**241**

**242** STANDARD LIBRARY APPENDIX 8

*closing* the stream. Opening a file returns a pointer to an object of type FILE, which records whatever information is necessary to control the stream. We will use "file pointer" and "stream" interchangeably when there is no ambiguity.

When a program begins execution, the three streams stdin, stdout, and stderr

are already open.

B1. 1 **File Operations**

The following functions deal with operations on files. The type size t is the unsigned integral type produced by the sizeof operator.

FILE \*fopen(const char \*filename, const char \*mode)

fopen opens the named file, and returns a stream, or NULL if the attempt fails.

Legal values for mode include

"r II open text file for reading

**11 w11** create text file for writing; discard previous contents if any

11a II append; open or create text file for writing at end of file

1r1

+ 11 open text file for update (i.e., reading and writing)

**11 w+** 11 create text file for update; discard previous contents if any

11 a+" append; open or create text file for update, writing at end

Update mode permits reading and writing the same file; fflush or a file-positioning function must be called between a read and a write or vice versa. If the mode

includes b after the initial letter, as in 1r1 b 11 or "w+b", that indicates a binary file.

Filenames are limited to FILENAME\_MAX characters. At most FOPEN\_MAX files may be open at once.

FILE \*freopen(const char \*filename, const char \*mode,

**FILE** \*Stream)

freopen opens the file with the specified mode and associates the stream with it. It returns stream, or NULL if an error occurs. freopen is normally used to change the files associated with stdin, stdout, or stderr.

int fflush(FILE \*Stream)

On an output stream, fflush causes any buffered but unwritten data to be written; on an input stream, the effect is undefined. It returns EOF for a wrtte error, and zero otherwise. fflush(NULL) flushes all output streams.

int fclose(FILE \*Stream)

fclose flushes any unwritten data for stream, discards any unread buffered input, frees any automatically allocated buffer, then closes the stream. It returns EOF if any errors occurred, and zero otherwise.

int remove(const char \*filename)

remove removes the named file, so that a subsequent attempt to open it will fail. It returns non-zero if the attempt fails.

int rename(const char \*Oldname, const char \*newname) rename changes the name of a file; it returns non-zero if the attempt fails.

SECTION **Bl** INPUT AND OUTPUT: <STDIO.H> **243**

FILE \*tmpfile(void)

tmpfile creates a temporary file of mode "wb+" that will be automatically removed when closed or when the program terminates normally. tmpfile returns a stream, or NULL if it could not create the file.

char \*tmpnam(char s[L\_tmpnam])

tmpnam (NULL) creates a string that is not the name of an existing file, and returns a pointer to an internal static array. tmpnam( s) stores the string in s as well as returning it as the function value; s must have room for at least L\_ tmpnam charac­ ters. tmpnam generates a different name each time it is called; at most **TMP \_MAX** different names are guaranteed during execution of the program. Note that tmpnam creates a name, not a file.

int setvbuf(FILE \*Stream, char \*buf, int mode, size\_t size) setvbuf controls buffering for the stream; it must be called before reading, writing, or any other operation. A mode of \_IOFBF causes full buffering, \_IOLBF line buffering of text files, and \_ IONBF no buffering. If buf is not NULL, it will be used as the buffer; otherwise a buffer will be allocated. size determines the buffer size. setvbuf returns non-zero for any error.

void setbuf(FILE \*Stream, char \*buf)

If buf is NULL, buffering is turned off for the stream. Otherwise, setbuf is equivalent to (void) setvbuf ( stream, buf, \_ IOFBF, BUFSIZ).

**B**1.2 **FQrmatted Output**

The print£ functions provide formatted output conversion.

int fprintf(FILE \*Stream, const char \*format, ... )

fprintf converts and writes output to stream under the control of format. The return value is the number of characters written, or negative if an error occurred.

The format string contains two types of objects: ordinary characters, which are copied to the output stream, and conversion specifications, each of which causes conver­ sion and printing of the next successive argument to fprintf. Each conversion specifi­ cation begins with the character % and ends with a conversion character. Between the % and the conversion character there may be, in order:

* Flags (in any order), which modify the specification:

-, which specifies left adjustment of the converted argument in its field.

+, which specifies that the number will always be printed with a sign.

*space:* if the first character is not a sign, a space will be prefixed.

O: for numeric conversions, specifies padding to the field width with leading zeros.

#, which specifies an alternate output form. For o, the first digit will be zero. For x or x, Ox or OX will be prefixed to a non-zero result. For e, E, f, g, and G, the output will always have a decimal point; for g and G, trailing zeros will not be removed.

* A number specifying a minimum field width. The converted argument will be printed in a field at least this wide, and wider if necessary. If the converted argument has fewer characters than the field width it will be padded on the left (or right, if left adjustment has been requested) to make up the field width. The padding character is normally space, but is O if the zero padding flag is present.

**244** STANDARD LIBRARY **APPENDIX B**

* A period, which separates the field width from the precision.
* A number, the precision, that specifies the maximum number of characters to be printed from a string, or the number of digits to be printed after the decimal point for **e, E,** or f conversions, or the number of significant digits for g or G conversion, or the minimum number of digits to be printed for an integer (leading Os will be added to make up the necessary width).
* A length modifier h, 1 Oetter ell), or L. "h" indicates that the corresponding argument is to be printed as a short or unsigned short; "1" indicates that the argument is a long or unsigned long; "L" indicates that the argument is a long double.

Width or precision or both may be specified as \*, in which case the value is computed by converting the next argument(s), which must be int.

The conversion characters and their meanings are shown in Table 8-1. If the char­ acter after the " is not a conversion character, the behavior is undefined.

TABLE 8-1. PRINTF CONVERSIONS

CHARACTER ARGUMENT TYPE; CONVERTED TO

d, i int; signed decimal notation.

o int; unsigned octal notation (without a leading zero).

x, x int; unsigned hexadecimal notation (without a leading Ox or ox), using abcdef for Ox or ABCDEF for OX.

u int; unsigned decimal notation.

c int; single character, after conversion to unsigned char.

s char \*; characters from the string are printed until a '\0' is reached or until the number of characters indicated by the pre­ cision have been printed.

f double; decimal notation of the form *[-)mmm.ddd,* where the

number of d's is specified by the precision. The default preci­ sion is 6; a precision of O suppresses the decimal point.

e, E double; decimal notation of the form *[-]m.dddddde±xx* or

*[-]m.ddddddE±xx,* where the number of d's is specified by the precision. The default precision is 6; a precision of O suppresses the decimal point.

g, G double; "e or "E is used if the exponent is less than -4 or **greater** .than or equal to the precision; otherwise "£ is used. Trailing zeros and a trailing decimal point are not printed.

p void \*; print as a pointer (implementation-dependent represen­ tation).

n int \*; the number of characters written so far by this call to print£ is *written into* the argument. No argument is con­ verted.

" no argument is converted; print a "·

int printf(const char \*format, ... )

print£ (... ) is equivalent to £print£ ( stdout,... ).

SECTION Bl INPUT AND OUTPUT: <STDIO.H> **245**

int sprintf(char \*S, const char \*format, ...)

sprint£ is the same as printf except that the output is written into the string s, terminated with '\O '. s must be big enough to hold the result. The return count does not include the '\0 '.

vprintf(const char \*format, va\_list arg)

vfprintf(FILE \*Stream, const char \*format, va\_list arg) vsprintf(char \*S, const char \*format, va\_list arg)

The functions vprintf, vfprintf, and vsprintf are equivalent to the corresponding printf functions, except that the variable argument list is replaced by arg, which has been initialized by the va\_start macro and perhaps va\_arg calls. See the discussion of <stdarg.h> in Section B7.

B 1.3 **Formatted Input**

The scanf functions deal with formatted input conversion.

int fscanf(FILE \*Stream, const char \*format, •..)

fscanf reads from stream under control of format, and assigns converted values through subsequent arguments, *each of which must be a pointer.* It returns when format is exhausted. fscanf returns EOF if end of file or an error occurs before any conversion; otherwise it returns the number of input items converted and assigned.

The format string usually contains conversion specifications, which are used to direct interpretation of input. The format string may contain:

* Blanks or tabs, which are ignored.
* Ordinary characters (not "), which are expected to match the next non-white space character of the input stream.
* Conversion specifications, consisting of a X, an optional assignment suppression character \*, an optional number specifying a maximum field width, an optional h, 1, or L indicating the width of the target, and a conversion character.

A conversion specification determines the conversion of the next input field. Nor­ mally the result is placed in the variable pointed to by the corresponding argument. If assignment suppression is indicated by \*, as in "\*S, however, the input field is simply skipped; no assignment is made. An input field is defined as a string of non-white space characters; it extends either to the next white space character or until the field width, if specified, is exhausted. This implies that scanf will read across line boundaries to find its input, since newlines are white space. (White space characters are blank, tab, new­ line, carriage return, vertical tab, and formfeed.)\_

The conversion character indicates the interpretation of the input field. The corresponding argument must be a pointer. The legal conversion characters are shown in Table B-2.

The conversion characters d, i, n, o, u, and x may be preceded by h if the argument is a pointer to short rather than int, or by 1 {letter ell) if the argument is a pointer to long. The conversion characters e, f, and g may be preceded by 1 if a pointer to double rather than float is in the argument list, and by L if a pointer to a long double.

**246** STANDARD LIBRARY APPENDIX B

TABLE B-2. SCANF CONVERSIONS

CHARACTER INPUT DATA; ARGUMENT TYPE

d decimal integer; int \*·

i integer; int \*· The integer may be in octal (leading O) or hexadecimal (leading Ox or ox).

o octal integer (with or without leading zero); int \*· u unsigned decimal integer; unsigned int \*·

x hexadecimal integer (with or without leading Ox or ox); int \*· c characters; char \*· The next input characters are placed in the

indicated array, up to the number given by the width field; the default is 1. No '\0' is added. The normal skip over white space characters is suppressed in this case; to read the next non-white space character, use %1 s.

s string of non-white space characters (not quoted); char \*, pointing to an array of characters large enough to hold the string and a terminating '\O' that will be added.

e, f, g floating-point number; float \*· The input format for float's is an optional sign, a string of numbers possibly containing a decimal point, and an optional exponent field containing an E or e followed by a possibly signed integer.

p pointer value as printed by printf ( "%p" ); void \*·

n writes into the argument the number of characters read so far by this call; int \*· No input is read. The converted item count is not incremented.

[. .. ] matches the longest non-empty string of input characters from the set between brackets; char \*· A '\0' is added. [ ]... ] includes ] in the set.

[ "... ] matches the longest non-empty string of input characters *not* from the set between brackets; char \*· A '\0' is added. [ " ]... ] includes ] in the set.

% literal %; no assignment is made.

int scanf(const char \*format, ... ) scanf (... ) is identical to f scanf ( std in, ... ).

int sscanf(char \*S, const char \*format, .•. )

sscanf ( s, ... ) is equivalent to scanf (... ) except that the input characters are taken from the string s.

**B1.4 Character Input and Output Functions**

int fgetc(FILE \*Stream)

fgetc returns the next character of stream as an unsigned char (converted to an int), or EOF if end of file or error occurs.

SECTION Bl INPUT AND OUTPUT: <STDIO.H> **247**

char \*fgets(char \*S, int n, FILE \*Stream}

fgets reads at most the next n-1 characters into the array s, stopping if a newline is encountered; the newline is included in the array, which is terminated by '\0 '. fgets returns s, or NULL if end of file or error occurs.

int fputc(int c, FILE \*Stream}

fputc writes the character c (converted to an unsigned char) on stream. It returns the character written, or EOF for error.

int fputs(const char \*S, FILE \*Stream}

fputs writes the string s (which need not contain '\n') on stream; it returns non-negative, or EOF for an error.

int getc(FILE \*Stream}

getc is equivalent to fgetc except that if it is a macro, it may evaluate stream

more than once.

int getchar(void}

getchar is equivalent to getc( stdin}.

char \*gets(char \*S}

gets reads the next input line into the array s; it replaces the terminating newline with '"\O '. It returns s, or NULL if end of file or error occurs.

int putc(int c, FILE \*Stream}

putc is equivalent to fputc except that if it is a macro, it may evaluate stream

more than once.

int putchar(int c}

putchar(c} is equivalent to putc( c,stdout}.

int puts(const char \*S}

puts writes the string s and a newline to stdout. It returns EOF if an error occurs, non-negative otherwise.

int ungetc(int c, **FILE** \*Stream}

ungetc pushes c (converted to an unsigned char) back onto stream, where it will be returned on the next read. Only one character of pushback per stream is guaranteed. EOF may not be pushed back. ungetc returns the character pushed back, or EOF for error.

**B 1.5 Direct Input and Output Functions**

size\_t fread(void \*Ptr, size\_t size, size\_t nobj, FILE \*Stream} fread reads from stream int-o the array ptr at most nobj objects of size size. fread returns the number of objects read; this may be less than the number requested. feof and ferror must be used to determine status.

size\_t fwrite(const void \*Ptr, size\_t size, size\_t nobj,

FILE \*Stream)

fwrite writes, from the array ptr, nobj objects of size size on stream. It

returns the number of objects written, which is less than nobj on error.

**248** STANDARD LIBRARY APPENDIX B

**B1.6 File Positioning Functions**

int fseek(FILE \*Stream, long offset, int origin)

fseek sets the file position for stream; a subsequent read or write will access data beginning at the new position. For a binary file, the position is set to offset char­ acters from origin, which may be SEEK\_SET (beginning), SEEK\_CUR (current position), or SEEK\_END (end of file). For a text stream, offset must be zero, or a value returned by ftell (in which case origin must be SEEK\_SET). fseek returns non-zero on error.

long ftell(FILE \*Stream)

ftell returns the current file position for stream, or -1L on error.

void rewind(FILE \*Stream)

rewind(fp) is equivalent to fseek(fp,0L,SEEK\_SET); clearerr(fp).

int fgetpos(FILE \*Stream, fpos\_t \*Ptr)

fgetpos records the current position in stream in \*Ptr, for subsequent use by fsetpos. The type fpos\_ t is suitable for recording such values. fgetpos returns non-zero on error.

*I* int fsetpos(FILE \*Stream, const fpos\_t \*Ptr)

fsetpos positions stream at the position recorded by fgetpos in \*Ptr.

fsetpos returns non-zero on error.

**B1.7 Error Functions**

Many of the functions in the library set status indicators when error or end of file occur. These indicators may be set and tested explicitly. In addition, the integer expres­ sion errno (declared in <errno.h>) may contain an error number that gives further information about the most recent error.

void clearerr(PILE \*Stream)

clearerr clears the end of file and error indicators for stream.

int feof(FILE \*Stream)

feof returns non-zero if the end of file indicator for stream is set.

int ferror(FILE \*Stream)

ferror returns non-zero if the error indicator for stream is set.

void perror(const char \*S)

perror ( s) prints s and an implementation-defined error message corresponding to the integer in errno, as if by

fprintf(stderr, "%s: %s\n", s, *"error message")*

See strerror in Section B3.

**82. Character Class Tests: <ctype.h>**

The header <ctype . h> declares functions for testing characters. For each function, the argument is an int, whose value must be EOF or representable as an unsigned

SECTION 83 STRING FUNCTIONS: <STRING.H> **249**

char, and the return value is an int. The functions return non-zero (true) if the argu­ ment c satisfies the condition described, and zero if not.

isalnwn(c) isalpha(c) iscntrl(c) isdigit(c) isgraph(c) islower(c) isprint(c) ispunct(c) isspace(c) isupper(c) isxdigit(c)

isalpha( c) or isdigit( c) is true isupper(c) or islower( c) is true control character

decimal digit

printing character except space lower-case letter

printing character including space

printing character except space or letter or digit

space, formfeed, newline, carriage return, tab, vertical tab upper-case letter

hexadecimal digit

In the seven-bit ASCII character set, the printing characters are 0x20 (' ') to 0x7E

('-'); the control characters are 0 (NUL) to 0x1F (us), and 0x7F (DEL).

In addition, there are two functions that convert the case of letters:

int tolower(int c) int toupper(int c)

convert c to lower case convert c to upper case

If c is an upper-case letter, tolower( c) returns the corresponding lower-case letter; otherwise it returns c. If c is a lower-case letter, toupper(c) returns the correspond­ ing upper-case letter; otherwise it returns c.

**B3. String Functions: < string.h** >

There are two groups of string functions defined in the header <string.h>. The first have names beginning with str; the second have names beginning with mem. Except for memmove, the behavior is undefined if copying takes place between overlap­ ping objects. Comparison functions treat arguments as unsigned char arrays.

In the following table, variables s and t are of type char \*; cs and ct are of type

const char \*; n is of type **size\_**t; and c is an int converted to char.

char \*Strcpy(s,ct) char \*Strncpy(s,ct,n)

char \*Strcat(s,ct) char \*Strncat(s,ct,n)

int strcmp(cs,ct) int strncmp(cs,ct,n) char \*Strchr(cs,c) char \*Strrchr(cs,c)

copy string ct to string s, including '\O '; return s. copy at most n characters of string ct to s; return s. Pad with ' \ 0 's if t has fewer than n characters. concatenate string ct to end of string s; return s. concatenate at most n characters of string ct to string s, terminate s with '\O '; return s.

compare string cs to string ct; return <O if cs<ct, 0 if cs==ct, or >0 if cs>ct.

compare at most n characters of string cs to string ct; return <0 if cs<ct, 0 if cs==ct, or >0 if cs>ct. return pointer to first occurrence of c in cs or NULL if not present.

return pointer to last occurrence of c \_in cs or NULL if not present.

**250** STANDARD LIBRARY APPENDIX B

size\_t strspn(cs,ct) size\_t strcspn(cs,ct) char \*Strpbrk(cs,ct) char \*Strstr(cs,ct)

size\_t strlen(cs) char \*Strerror(n)

char \*Strtok(s,ct)

return length of prefix of cs consisting of characters in ct.

return length of prefix of cs consisting of characters

*not* in ct.

return pointer to first occurrence in string cs of any character of string ct, or NULL if none are present. return pointer to first occurrence of string ct in cs, or NULL if not present.

return length of cs.

return pointer to implementation-defined string corresponding to error n.

strtok searches s for tokens delimited by characters

from ct; see below.

A sequence of calls of strtok ( s, ct) splits s into tokens, each delimited by a character from ct. The first call in a sequence has a non-NULL s. It finds the first token in s consisting of characters not in ct; it terminates that by overwriting the next character of s with '\0' and returns a pointer to the token. Each subsequent call, indi­ cated by a NULL value of s, returns the next such token, searching from just past the end of the previous one. strtok returns NULL when no further token is found. The string ct may be different on each call.

The mem... functions \_are meant for manipulating objects as character arrays; the intent is an interface to efficient routines. In the following table, s and t are of type void \*; cs and ct are of type const void \*; n is of type **size\_** t; and c is an int converted to an unsigned char.

void \*memcpy ( s , ct, n) copy n characters from ct to s, and return s.

void \*memmove ( s, ct, n) same as memcpy except that it works even if the

objects overlap.

int memcmp (cs, ct, n) compare the first n characters of cs with ct; return as

with strcmp.

**void** \*memchr (cs, c, n) return pointer to first occurrence of character c in cs,

or NULL if not present among the first n characters.

void \*memset ( s , c , n) place character c into first n characters of s, return s.

**84. Mathematical Functions: <math.h>**

The header <math.h> declares mathematical functions and macros.

The macros EDOM and ERANGE (found in <errno. h>) are non-zero integral con­ stants that are used to signal domain and range errors for the functions; HUGE\_ VAL is a positive double value. A *domain error* occurs if an argument is outside the domain over which the function is defined. On a domain error, errno is set to EDOM; the return value is implementation-dependent. A *range error* occurs if the result of the function cannot be represented as a double. If the result overflows, the function returns HUGE\_VAL with the right sign, and errno is set to ERANGE. If the result underflows, the function returns zero; whether errno is set to ERANGE is implementation-defined.

In the following table, x and y are of type double, n is an int, and all functions return double. Angles for trigonometric functions are expressed in radians.

SECTION 85 UTILITY FUNCTIONS: <STDLIB.H> **251**

sin(x) sine of *x*

cos(x)

tan(x) asin(x) acos(x) atan(x)

cosine of *x*

tangent of *x*

sin- 1 (x) in range [--ir/2, -ir/2], *x* E [-1, 11.

cos-1 (x) in range [O, -ir], *x* E [-1, I]. tan- 1 (x) in range [--ir/2, -ir/21.

atan2 ( y, x) tan- 1 *(ylx)* in range [--ir, -ir].

s inh ( x) hyperbolic sine of *x*

cosh(x) tanh(x) exp(x)

log(x) log10(x) pow(x,y)

hyperbolic cosine of *x* hyperbolic tangent of *x* exponential function *ex* natural logarithm ln(x), *x>O.*

base 10 logarithm log10 (x), *x>O.*

*xY.* A domain error occurs if x-0 and y O, or if x <0 and y is not an integer.

sqrt(x) £, *x O.*

ceil (x) smallest integer not less than x, as a double.

floor(x) largest integer not greater than x, as a double. fabs(x) absolute value lxl

ldexp(x,n) *x·2n*

frexp(x, int **\*exp)**

splits x into a normalized fraction in the interval [1/2, 1), which is returned, and a power of 2, which is stored in \*exp. If **x** is zero, both parts of the result are zero.

modf(x, double \*iP)

splits **x** into integral and fractional parts, each with the same sign as **x.** It stores the integral part in \*iP, and returns the fractional part.

fmod ( **x, y)** floating-point remainder of **x/y,** with the same sign as

**x.** If **y** is zero, the result is implementation-defined.

**B5. Utility Functions: < stdlib.h** >

The header <stdlib. h> declares functions for number conversion, storage alloca­ tion, and similar tasks.

double atof(const char \*S)

atof converts s to double; it is equivalent to strtod(s, (char\*\*)NULL).

int atoi(const char \*S)

converts s to int; it is equivalent to ( int) strtol ( s, (char\*\*) NULL, 10 ) .

long atol(const char \*S)

converts s to long; it is equivalent to strtol ( s, ( char\*\* )NULL, 10 ).

double strtod(const char \*S, char \*\*endp)

strtod converts the prefix of s to double, ignoring leading white space; it stores a point r to any unconverted suffix in \*endp unless endp is NULL. If the answer

**252** STANDARD LIBRARY APPEND{X **B**

would overflow, HUGE\_ VAL is returned with the proper sign; if the answer would underflow, zero is returned. In either case errno is set to ERANGE.

long strtol(const char \*S, char \*\*endp, int base)

strtol converts the prefix of s to long, ignoring leading white space; it stores a pointer to any unconverted suffix in \*endp unless endp is NULL. If base is between 2 and 36, conversion is done assuming that the input is written in that base. If base is zero, the base is 8, 10, or 16; leading O implies octal and leading Ox or OX hexadecimal. Letters in either case represent digits from 10 to base-1; a leading Ox or OX is permitted in base 16. If the answer would overflow, LONG \_MAX or LONG\_MIN is returned, depending on the sign of the result, and errno is set to

**ERANGE.**

unsigned long strtoul(const char \*S, char \*\*endp, int base) strtoul is the same as strtol except that the result is unsigned long and the error value is ULONG \_MAX.

int rand(void)

rand returns a pseudo-random integer in the range O to RAND\_MAX, which is at least 32767.

void srand(unsigned int seed)

srand uses seed as the seed for a new sequence of pseudo-random numbers. The initial seed is 1.

void \*Calloc(size\_t nobj, size\_t size)

calloc returns a pointer to space for an array of nobj objects, each of size size,

or NULL if the request cannot be satisfied. The space is initialized to zero bytes.

void \*malloc(size\_t **size)**

malloc returns a pointer to space for an object of size size, or NULL if the request cannot be satisfied. The space is uninitialized.

void \*realloc(void \*P, size\_t size)

realloc changes the size of the object pointed to by p to size. The contents will be unchanged up to the minimum of the old and new sizes. If the new size is larger, the new space is uninitialized. realloc returns a pointer to the new space, or NULL if the request cannot be satisfied, in which case \*P is unchanged.

void free(void \*P)

free deallocates the space pointed to by p; it does nothing if p is NULL. p must be a pointer to space previously allocated by calloc, malloc, or realloc.

void abort(void)

abort causes the program to terminate abnormally, as if by raise ( SIGABRT).

void exit(int status)

exit causes normal program termination. atexit functions are called in reverse order of registration, open files are flushed, open streams are closed, and control is returned to the environment. How status is returned to the environment is implementation-dependent, but zero is taken as successful termination. The values EXIT\_SUCCESS and EXIT\_FAILURE may also be used.

SECTION 86 DIAGNOSTICS: <ASSERT.H> **253**

int atexit(void (\*fcn)(void))

atexit registers the function fen to be called when the program terminates nor­ mally; it returns non-zero if the registration cannot be made.

int system(const char \*S)

system passes the string s to the environment for execution. If s is NULL, system returns non-zero if there is a command processor. If s is not NULL, the return value is implementation-dependent.

char \*getenv(const char \*name)

getenv returns the environment string associated with name, or NULL if no string exists. Details are implementation-dependent.

void \*bsearch(const void \*key, const void \*base, size\_t n, size\_t size,

int (\*cmp)(const void \*keyval, const void \*datum))

bsearch searches base[0]...base[ n-1] for an item that matches \*key. The function cmp must return negative if its first argument (the search key) is less than its second (a table entry), zero if equal, and positive if greater. Items in the array base must be in ascending order. bsearch returns a pointer to a matching item, or NULL if none exists.

void qsort(void \*base, size\_t n, size\_t size,

int (\*cmp)(const void\*, const void\*))

qsort sorts into ascending order an array base[0]...base[n-1] of objects of size

size. The comparison function cmp is as in bsearch.

int abs(int n)

abs returns the absolute value of its int argument.

long labs(long n)

labs returns the absolute value of its long argument.

div\_t div(int nwn, int denom)

div computes the quotient and remainder of nwn/denom. The results are stored in the int members quot and rem of a structure of type div\_t.

ldiv\_t ldiv(long num, long denom)

div computes the quotient and remainder of num/denom. The results are stored in the long members quot and rem of a structure of type ldiv\_t.

**B6. Diagnostics: <assert.h>**

The **assert** macro is used to add diagnostics to programs:

void **assert** ( int *expression)*

If *expression* is zero when

**assert** (*expression* )

is executed, the **assert** macro will print on stderr a message, such as

**Assertion failed:** *expression,* file *filename,* line *nnn*

It then calls abort to terminate execution. The source filename and line number come

**254** STANDARD LIBRARY APPENDIX B

from the preprocessor macros FILE and LINE .

If NDEBUG is defined at the time <assert. h> is included, the assert macro is ignored.

**87. Variable Argument Usts: < stdarg.h>**

The header <stdarg. h> provides facilities for stepping through a list of function arguments of unknown number and type.

Suppose *lastarg* is the last named parameter of a function f with a variable number of arguments. Then declare within f a variable ap of type va\_list that will point to each argument in turn:

va\_list ap;

apmust be initialized once with the macro va\_start before any unnamed argument is accessed:

va\_start(va\_list ap, *lastarg);*

Thereafter, each execution of the macro va\_arg will produce a value that has the type and value of the next unnamed argument, and will also modify ap so the next use of va\_arg returns the next argument:

*type* va\_arg(va\_list ap, *type);*

The macro

void va\_end(va\_list ap);

must be called once after the arguments have been processed but before f is exited.

**B8. Non-local Jumps: < setJmp.h>**

The declarations in <set jmp. h> provide a way to avoid the normal function call and return sequence, typically to permit an immediate return from a deeply nested func-· tion call.

int setjmp(jmp\_buf env)

The macro setjmp saves state information in env for use by longjmp. The return is zero from a direct call of setjmp, and non-zero from a subsequent call of longjmp. A call to setjmp can only occur in certain contexts, basically the test of if, switch, and loops, and only in simple relational expressions.

if (setjmp(env) == 0)

/\* get here on direct call\*/

else

/\* get here by calling longjmp \*/

void longjmp(jmp\_buf env, int val)

longjmp restores the state saved by the most recent call to setjmp, using informa­ tion saved in env, and execution resumes as if the setjmp function had just exe­ cuted and returned the non-zero value val. The function containing the set jmp must not have terminated. Accessible objects have the values they had when longjmp was called, except that non-volatile automatic variables in the function calling setjmp become undefined if they were changed after the setjmp call.

SECTION BIO DATE AND TIME FUNCTIONS: <TIME.H> **255**

##### 89. Signals: < signal.h>

The header <signal. h> provides facilities for handling exceptional conditions that arise during execution, such as an interrupt signal from an external source or an error in execution.

void (\*signal(int sig, void (\*handler)(int)))(int)

signal determines how subsequent signals will be handled. If handler is SIG\_DFL, the implementation-defined default behavior is used; if it is SIG\_IGN, the signal is ignored; otherwise, the function pointed to by handler will be called, with the argument of the type of signal. Valid signals include

SIGABRT abnormal termination, e.g., from abort SIGFPE arithmetic error, e.g., zero divide or overflow SIGILL illegal function image, e.g., illegal instruction SIGINT interactive attention, e.g., interrupt

SIGSEGV illegal storage access, e.g., access outside memory limits

SIGTERM termination request sent to this program

signal returns the previous value of handler for the specific signal, or SIG\_ERR if an error occurs.

When a signal sig subsequently occurs, the signal is restored to its default behavior; then the signal-handler function is called, as if by (\*handler) ( sig). If the handler returns, execution will resume where it was when the signal occurred.

The initial state of signals is implementation-defined.

int raise(int sig)

raise sends the signal sig to the program; it returns non-zero if unsuccessful.

**B1O. Date and Time Functions: < time.h>**

The header <time. h> declares types and functions for manipulating date and time. Some functions process *local time,* which may differ from calendar time, for example because of time zone. clock\_ t and time\_ t are arithmetic types representing times, and struct tmholds the components of a calendar time:

|  |  |  |
| --- | --- | --- |
| int | tm\_sec; | seconds after the minute (0, 61) |
| int | tm\_min; | minutes after the hour (0, 59) |
| int | tm\_hour; | hours since midnight (0, 23) |
| int | tm\_mday; | day of the month ( I, 31) |
| int | tm\_mon; | months *since* January (0, 11) |
| int | tm\_year; | years since 1900 |
| int | **tm\_wday;** | days since Sunday (0, 6) |
| int | tm\_yday; | days since January I (0, 365) |
| int | tm\_isdst; | Daylight Saving Time flag |

tm\_isdst is positive if Daylight Saving Time is in effect, zero if not, and negative if the information is not available.

clock\_t clock(void)

clock returns the processor time used by the program since the beginning of execu­ tion, or -1 if unavailable. clock ( ) /CLOCKS\_PER\_SEC is a time in seconds.

**256** STANDARD LIBRARY APPENDIX B

time\_t time(time\_t \*tp)

time returns the current calendar time or -1 if the time is not available. If tp is not NULL, the return value is also assigned to \*tp.

double difftime(time\_t time2, time\_t time1) difftime returns time2-time 1 expressed in seconds.

time\_t mktime(struct tm \*tp)

mktime converts the local time in the structure \*tP into calendar time in the same representation used by time. The components will have values in the ranges shown. **mktime** returns the calendar time or -1 if it cannot be represented.

The next four functions return pointers to static objects that may be overwritten by other calls.

char \*asctime(const struct tm \*tP)

asctime converts the time in the structure \*tp into a string of the form

sun Jan 3 15:14:13 19aa,n,o

char \*Ctime(const time\_t \*tp)

ctime converts the calendar time \*tP to local time; it is equivalent to

asctime(localtime(tp))

struct tm \*gmtime(const time\_t \*tp)

gmtime converts the calendar time \*tP into Coordinated Universal Time (UTC). It returns NULL if UTC is not available. The name gmtime has historical significance.

struct tm \*localtime(const time\_t \*tp) localtime converts the calendar time \*tP into local time.

**size\_t** strftime(char \*S, size\_t **smax,** const char \*fmt,

const struct tm \*tp)

strftime formats date and time information from \*tP into s according to fmt, which is analogous to a print£ format. Ordinary characters (including the ter­ minating ',o ') are copied into s. Each *%c* is replaced as described below, using

values appropriate for the local environment. No more than smax characters are placed into s. strftime returns the number of characters, excluding the ',o', or zero if more than smax characters were produced.

%a abbreviated weekday name.

"%bA full weekday name.

abbreviated month name. full month name.

"B

"c local date and time representation.

%d day of the month (01-31). "H hour (24-hour clock) (00-23). "I hour (12-hour clock) (01-12). "j day of the year (001-366).

SECTION Bil IMPLEMENTATION-DEFINED LIMITS: <LIMITS.H> AND <FLOAT.H> **257**

%m month (01-12).

%M. minute (00-59).

""Ps

local equivalent of AM or PM.

second (00-61).

"U week number of the year (Sunday as 1st day of week) (00-53).

"w

weekday (0-6, Sunday is 0).

***»I* week** number of the year (Monday as 1st day of week) (00-53).

h local date representation.

,a local time representation.

""YY year without century (00-99).

"z

year with century.

time zone name, if any.

"" "-

**B11. Implementation-defined Limits: <limlts.h > and <float.h** >

The header <limits . h> defines constants for the sizes of integral types. The values below are acceptable minimum magnitudes; larger values may be used.

**CHAR\_BIT** 8 bits in a char

**CHAR\_MAX UCHAR\_MAX** *or*

**SCHAR\_MAX** maximum value of char

**CHAR\_MIN** 0 *or* **SCHAR\_MIN INT\_MAX** +32767

**INT\_MIN** -32767

**LONG\_MAX** +2147483647

**LONG\_MIN** -2147483647

**SCHAR\_MAX** +127

**SCHAR\_MIN** -127

**SHRT\_MAX** +32767

**SHRT\_MIN** -32767

**UCHAR\_MAX** 255

**UINT\_MAX** 65535

**ULONG\_MAX 4294967295**

**USHRT\_MAX** 65535

minimum value of char maximum value of int minimum value of int maximum value of long minimum vl'tlue of long

maximum value of signed char minimum value of signed char maximum value of short minimum value of short

maximum value of unsigned char maximum value of unsigned int maximum value of unsigned long maximum value of unsigned short

The names in the table below, a subset of <float. h>, are constants related to floating-point arithmetic. When a value is given, it represents the minimum magnitude for the corresponding quantity. Each implementation defines appropriate values.

**FLT\_RADIX FLT\_ROUNDS FLT\_DIG FLT\_EPSILON FLT\_MANT\_DIG FLT\_MAX FLT\_MAX\_EXP FLT\_MIN FLT\_MIN\_EXP**

2 radix of exponent representation, e.g., 2, 16 floating-point rounding mode for addition

6 decimal digits of precision

1E-5 smallest number *x* such that 1.0 + *x* ***;it*** 1.0 number of base FLT **\_RADIX** digits in mantissa

1E+37 maximum floating-point number

maximum *n* such that FLT \_RADIXn- 1 is representable

1E-37 minimum normalized floating-point number minimum *n* such that *10n* is a normalized number

**2S8** STANDARD LIBRARY APPENDIX B

DBL\_DIG **DBL\_EPSILON DBL\_MANT\_DIG DBL\_MAX DBL\_MAX\_EXP DBL\_MIN DBL\_MIN\_EXP**

10 decimal digits of precision

1E-9 smallest number *x* such that 1.0 + *x* 1.0 number of base FLT\_RADIX digits in mantissa

1E+37 maximum double floating-point number

maximum *n* such that FLT \_RADI:x"-1 is representable

1E-37 minimum normalized double floating-point number minimum *n* such that 10" is a normalized number