II. Portable C Runtime Library

NAME

Conventions - using C with the standard libraries

FUNCTION

The current section, and the two that follow, document C callable functions provided on all systems supported by Whitesmiths, Ltd. All library functions follow a set of uniform coding conventions, which form an important part of the Whitesmiths C environment. These conventions should be mastered, the better to understand the descriptions following, to interface properly to the library functions, and, more generally, to write C in a portable manner.

Most standard conventions are supported at compile time by the inclusion of a standard header file, std.h, which is separately documented in this section. The remainder are mainly described in the subsections on Style and Portability in Section I of this manual. Here, however, are a few general caveats: Every C program must contain a function named main, which is called at the outset and whose return signals the end of program execution; many library routines presume a conventional coding of main, documented in Section III of this manual. Several of the "functions" described in this section are actually macros defined in the standard header. They appear on ordinary manual pages because, aside from certain side effects for which warning is served, they look to the programmer much like subprograms. On the other hand, there are a few secret library routines that are not documented anywhere in this manual; their names invariably begin with an underscore, to minimize accidental collisions with user-defined names.

The rest of this document provides a blow-by-blow summary of the sections in a typical library function description. For clarity, it is presented as a psuedo-manual page, with the remarks on each section of a real page appearing under the normal heading for that section. This page also points out where the conventions just mentioned are likely to rear their heads. The sections follow:

NAME

title - a name and concise description for the function
(The function is called from C by the name given.)

SYNOPSIS

Here the returned type and argument list of the function are given precisely as they would appear in a C program defining the function. Almost always, the types of arguments and function come from the set of psuedo-types defined by the standard header. These psuedo-types are for the most part simple equivalents to types pre-defined by C, renamed to increase their mnemonic value, to isolate machine dependencies, to promote disciplined coding practices, or (usually) to achieve some combination of all three.

FUNCTION

Generally, this section contains three parts, which may or may not be easily distinguishable. An opening sentence or two states the utility of the function: usually what it does, not how it does it. Next comes a summary of each argument to the function, including its

general effect on the operation of the function. Each argument is called by the same name as was given in the function synopsis. These names are usually mnemonic, to make citations distinct but still self-defining. Finally, an additional paragraph (or more) may provide more details of how the routine works, or how specific argument values affect it. This last component, for better or worse, has traditionally been kept to a remarkable minimum.

Note that heavy use is made of the conventional symbol NULL to refer to a zero-valued pointer, and of NUL to refer to the ASCII code zero, or "\0"; other symbols defined in the standard header may also crop up from time to time. Be prepared for them.

Ranges of numbers are often represented as in mathematics: "(0,4)" is the open interval "1,2,3", whereas "[0,4]" is the closed interval "0,1,2,3,4". Thus, "[0,4)" is not a typo, but shorthand for the half-open interval "0,1,2,3".

RETURNS

This section describes the range of possible return values for the function, and under what circumstances one value will be returned rather than another. Any location in the calling program altered via pointer access from the function is also documented here, though usually it has already been mentioned in the preceding section.

EXAMPLE

Here are shown one or more typical calls to the documented function, and (sometimes) their results. The examples are designed to be brief and evocative, or even useful as code fragments directly interpolated into user programs. Often, related functions have been given similar examples, either to emphasize the differences in usage between routines that perform similar tasks, or to show conventional patterns of use that apply across a family of routines.

A line consisting of just an ellipsis "..." is used to indicate the omission of some (incidental) code between the line preceding and the line following.

SEE ALSO

This section lists related functions that could be profitably examined in conjunction with the current one. Functions documented in the same manual section are simply listed by name; functions documented elsewhere in the same manual are listed followed by the number of the section containing them.

If a description seems a bit impenetrable after a first reading, by all means look at the other ones mentioned here. Likewise if the current function doesn't do exactly what is wanted; a different one may come closer.

BUGS

Known inconsistencies or shortcomings in the documented routine are mentioned here. Most often, these relate to insufficient checking of value sensitive user-supplied parameters. Also, notice is always

given here if a "function" is in reality a macro.

SEE ALSO
Portability(I), Style(I), main(III), std.h

NAME

std.h - standard header file

SYNOPSIS

#include <std.h>

FUNCTION

All standard library functions callable from C follow a set of uniform conventions, many of which are supported at compile time by including a standard header file, <std.h>, at the top of each program. The file defines a number of quasi-types and storage classes (in terms of the standard C types), various system parameters, some useful macros, and the control structure used for buffered input/output. The macros in <std.h> are each described in separate manual pages since, aside from certain curious side effects for which warning is served, macros look to the C programmer much like subroutines.

It is important to know these types and parameters, in order to understand manual pages for subroutines, to interface to the C library in a portable manner, and to code in good style. Herewith the principal definitions:

Quasi-types

ARGINT - int, used to signal use of value widening conventions
BITS - unsigned short, used as a set of 16 bits
BOOL - int, tested only for non-zero, assigned YES or NO
BYTES - unsigned int, for address arithmetic, indexing
COUNT - short, for counting [-32,768, 32,768)
DOUBLE - double precision floating point
FILE - short, used for file descriptors
LONG - long integer
METACH - short, EOF or [0, 256)
TBOOL - char, or unsigned char, used like BOOL
TEXT - char, or unsigned char, containing printable text
TINY - char, for counting [-128, 128)
UCOUNT - unsigned short, for counting [0, 65,536)
ULONG - unsigned long
UTINY - unsigned char, for counting [0, 256)
VOID - int, for functions returning nothing

Quasi storage classes

FAST - register
GLOBAL - synonym for extern, used outside functions
IMPORT - synonym for extern, used inside functions
INTERN - synonym for static, used inside functions
LOCAL - synonym for static, used outside functions

System Parameters

BUFSIZE - 512, the standard input/output buffer size
BWRITE - mode -1, opening for buffered writes
BYTMASK - 0377, mask for low byte of integer
EOF - -1, end of file metacharacter
FOREVER - for (; ;)
NO - BOOL 0
NULL - pointer 0

READ - mode 0, opening for read access
STDERR - FILE 2, the standard error output
STDIN - FILE 0, the standard input
STDOUT - FILE 1, the standard output
UPDATE - mode 2, opening for reading and writing
WRITE - mode 1, opening for writing
YES - BOOL 1

Macros (documented in manual pages)
abs
gtc
isalpha
isdigit
islower
isupper
iswhite
max
min
ptc
tolower
toupper

Control Structure for Input/Output
FIO - struct fio, for buffered input/output calls

Note that TBOOL and TEXT are defined as unsigned char if the symbol UTEXT is made known to pp; otherwise TBOOL and TEXT are defined as char.

EXAMPLE

```
/* THE MINIMUM PROGRAM
 * copyright (c) 1983 by Whitesmiths, Ltd.
 */
#include <std.h>

/* put string to STDOUT
 */
BOOL main()
    {
    write(STDOUT, "hello world\n", 12);
    return (YES);
    }
```

BUGS

It is easy to forget about the macros, which cause bizarre diagnostics when "redeclared". Also, despite its suggestive name, the type BITS cannot be used to declare a bitfield, as bitfields have type int or unsigned int.

NAME

Files - I/O using the standard libraries

FUNCTION

To the standard libraries, a file is an ordered sequence (or "stream") of eight-bit bytes, period. No further structure or control information is made visible to user programs. The basic operations permitted on files are determined by this very simple view. At the lowest level, a subset of the library called the portable interface allows you to:

open a file - that is, establish a connection between your program and some external source or destination for data. This is usually a file of information on disk, but could be your terminal, or even some other device.

close a file - that is, break a connection that you previously set up.

read from a file - that is, given the address of a buffer within your program, and the number of characters you want to read, transfer them from the file into your buffer.

write to a file - that is, given a buffer address and the number of characters to write, transfer them from your buffer into the file.

change position within a file - that is, instead of reading or writing characters sequentially, move to a given character offset within the file, in order to begin the operation at that point.

The simplicity of this small set of operations notwithstanding, the portable library provides a wide and sometimes overwhelming array of functions to perform input/output. Which ones to use, and in what context, depends on three characteristics of the operation to be performed:

1) **buffered or unbuffered?** The most basic I/O routines interface directly to your host operating system. On a conceptual level, at least, a call to one of these routines results directly in a request to the system to perform some action. In addition to these, however, the library provides a buffering mechanism (called an FIO buffer), that minimizes operating system overhead by buffering up to 512 characters in either direction before making a read or write call.

2) **formatted or literal?** The most straightforward I/O is just the transfer of bytes between memory and a file, with no changes made en route. For convenience, however, the library provides a family of functions to perform "formatted I/O", in which arbitrary numeric or text arguments are converted to tightly formatted text strings before being output (or text strings are converted to arbitrary internal values when they are input).

3) **text or binary?** This distinction is forced on the portable interface by most of the operating systems under which it runs. On principle, a byte stream is a byte stream; in fact, "text" files under most systems other than Idris/UNIX receive special treatment, such as the mapping of newlines to CR/LF pairs within the file. To maintain the

integrity of the byte stream in such an environment, the interface must work -- sometimes very hard -- to hide special actions performed automatically (and most often inescapably) by the operating system. To write code portably, therefore, files should be flagged as "text" or "binary" when opened, as the two types may have to be treated differently by the interface.

The Portable Interface

Here is a simple example of how to use the low level I/O functions from the portable interface:

```
/*  COPY BINARY INFILE TO OUTFILE, UNBUFFERED
 *  copyright (c) 1983 by Whitesmiths, Ltd.
 */
#include <std.h>

BOOL main(ac, av)
    BYTES ac;
    TEXT **av;
    {
    COUNT n;
    FILE inf, outf;
    TEXT buf[BUFSIZE];

    inf = open("infile", READ, 1);
    outf = create("outfile", WRITE, 1);
    while (0 < (n = read(inf, buf, BUFSIZE)))
        write(outf, buf, n);
    close(outf);
    close(inf);
    return (YES);
    }
```

This is a slightly dandified program to copy the file "infile" to a new file called "outfile". The function open tries to open a pre-existent file with the name given, for the type of access specified. Similarly, create tries to create an empty file with the name given, and opens the result for the access specified (which will almost certainly be writing). If successful, each of these routines returns a value called a file descriptor, which is guaranteed to be a small non-negative integer. Variables of type FILE, like inf and outf, are used to store file descriptors. If an open or create operation fails, the functions return a negative number.

Given a file descriptor and the address and length of a buffer in memory, the functions read and write transfer data exactly as you would expect. Each function returns the number of characters it transferred. Read will return zero when it encounters end-of-file, and both routines will return a negative number if an error occurs during the transfer.

Finally, close closes the file associated with a given file descriptor, hence ending the program's access to the file. The function returns the (now useless) file descriptor, if successful, otherwise a negative number.

Aside from the functions used in the example, the portable interface defines three others that deal with files. The function remove, given the name of a file, tries to remove it from the external environment, returning zero if it succeeds in doing so, otherwise a negative number. The function uname simply returns a text string that can be used for temporary filenames; the contents of this string are unlikely to conflict with ordinary filenames. The function lseek is used to change the value of the read/write pointer for an open file. This pointer indicates the offset from the start of the file of the next character to be read or written. Changing the pointer thus changes your position within the file.

In slightly more general terms, the last two arguments to open and create specify the access mode and record size to be applied to the file being opened. The access mode has three significant values, named in std.h: READ (0), WRITE (1), and UPDATE (2); the last value indicates mixed reading and writing.

The record size value, specified as the third argument to open or create, has two uses: first, if it is zero the file is opened as a "text" file (the file is otherwise opened as "binary"). Second, a non-zero record size may be used to determine the size of read or write requests to the operating system. The first meaning is far the more important.

The portable interface, across all of its implementations, supports a single notion of what constitutes a text file. In its view, a text file consists of zero or more lines, each no more than 512 characters in length, and each terminated by a single newline (i.e., an ASCII LF).

The interface can deal with longer lines, and last lines not properly terminated with a newline. But it is dedicated to presenting exactly this line structure to the user, regardless of the CR/LF (or LF/CR) pairs that may actually be in the file, and regardless of other "special" conventions a given environment may enforce. All of this can involve considerable memorization of state information. So in most non-Idris/UNIX environments, unexpected changes to a text file should be avoided. Often, a file opened as text cannot reliably be opened in UPDATE mode. Analogously, lseek calls to a text file are sometimes not supported.

These restrictions do not apply to binary files, since they tend to contain literally the data that was written to them. Many operating systems, however, append ASCII NUL characters to pad the length of a binary file to a multiple of some conventional size. Generally, the interface trustingly reads this padding as part of the valid contents of the file, so application code must be prepared to deal with it. Also, the pre-defined file descriptors STDIN, STDOUT and STDERR cannot generally be used safely to access binary files; on most systems, binary files must be opened by name.

The treatment of binary files in your environment, and the exact restrictions governing text file usage, are noted in the description of your particular system interface, in Section III of the appropriate C Interface Manual. Remember, however, that to code portably you should observe all of the strictures noted here. The interface description should also be

consulted about the meaning of the negative return values that the portable interface uses to signal errors. Wherever possible, these are directly related to the specific complaint made by the operating system.

If the restrictions imposed by the portable interface seem onerous, remember that they are not enforced arbitrarily. The interface provides the largest common set of services that can be extracted from a range of operating systems stretching from CP/M to VMS to UNIX System V. Input/output is traditionally one of the most unpredictable and system-dependent areas of programming. If you code within the constraints of the portable interface, however, you are assured that this aspect of your programs, just like the others, will be transportable across an increasingly broad spectrum of computers.

Portable Interface Extensions

Several functions are provided as convenient ways to use the portable interface for a specific purpose. The functions error and remark are used to output error messages to STDERR, with some simple additional formatting. error adds the program name, if available, to the messages output, then exits, reporting failure; remark just outputs the messages and returns. The function putstr, given a file descriptor and a series of strings terminated by a NULL argument, writes the strings to the file.

The function fread takes the same arguments as read, but acts differently. One difference between the two is that fread makes repeated calls to read, trying to obtain the number of characters requested in its call; read may return fewer, even before end-of-file. The other difference is that, on any read error, fread uses the _when/_raise mechanism to raise the readerr condition. By default, this results in an immediate exit, reporting failure, with an appropriate message output to STDERR. But the function _when can be used to take some other action when an error condition is raised.

Similarly, the function fwrite takes the same arguments as write. If any write error occurs, however, fwrite raises the writerr condition. Note that putstr and remark also raise the writerr condition if any writes fail; error, however, does not (so as to minimize the chance of looping on a severe error).

Buffered I/O Routines

The standard libraries also provide a separate set of functions that perform buffered I/O for better efficiency and ease of use. Instead of file descriptors, these routines use the FIO buffers mentioned earlier. An FIO buffer is a means of controlling transfers to or from a file, by buffering characters in either direction so as to minimize the number of read or write calls. The buffered I/O routines also make it possible to think of a text file in terms of lines or characters, rather than as fixed-size records. To begin the discussion, here's the same program as just presented, rewritten to use some of these routines:

```
/*  COPY TEXT INFILE TO OUTFILE, BUFFERED LINE BY LINE
 *  copyright (c) 1983 by Whitesmiths, Ltd.
 */
#include <std.h>

BOOL main(ac, av)
    BYTES ac;
    TEXT **av;
    {
    COUNT n;
    FIO infio, outfio;
    TEXT buf[BUFSIZE];

    fopen(&infio, "infile", READ);
    fcreate(&outfio, "outfile", WRITE);
    while (n = getl(&infio, buf, BUFSIZE))
        putl(&outfio, buf, n);
    fclose(&outfio);
    fclose(&infio);
    return (YES);
    }
```

One thing to note about this second version is its strong similarity to the first. Each portable interface function originally used has a direct counterpart among the buffered I/O functions.

Now for the differences: infio and outfio are both FIO buffers; the type FIO is defined by std.h. fopen and fcreate serve, respectively, to open a pre-existent file or to create an empty instance of a file. But note their arguments. First of all, both routines (like most other buffered I/O routines) expect the address of an FIO buffer defined elsewhere, usually in the calling program; they validate the buffer given, but they do not allocate a buffer. Second, neither routine has any "record size" argument; they implicitly deal with text files. (More about that later.) If successful, fcreate and fopen return a non-NULL value (the address of the FIO buffer they were given). Otherwise, they return NULL.

The routines getl and putl are designed to transfer lines of text between memory and an FIO buffer, making read or write calls as necessary whenever the buffer becomes unusable. Like read and write, getl and putl return the number of characters they transferred; and getl returns zero upon encountering end-of-file. However, getl shows its bias toward text files in that it stops transferring bytes after it transfers a newline, the intent being to transfer exactly one line of text on each call. The function fclose, of course, closes the file associated with a given FIO buffer, making sure that a buffer used for writing is flushed before the closing is performed. Like fopen and fcreate, fclose, if successful, returns the buffer address it was called with. Otherwise, fclose returns NULL.

Three pairs of routines in addition to getl/putl are available for doing literal I/O with a file controlled by an FIO buffer. The first pair, getlin/putlin, act exactly like getl/putl, except that they use the pre-defined FIO buffers stdin or stdout. The buffer stdin is initialized to work with the pre-defined file descriptor STDIN ("standard input"), while

the buffer stdout is initialized to work with the pre-defined file descriptor STDOUT ("standard output"). Both are provided by the standard libraries; if you want to use them explicitly, however, you should import them into your routine (because you'll almost certainly be taking their address). Thus the sample program could be made to transfer a text file line by line from stdin to stdout by writing it as:

```
/*  COPY TEXT stdin TO stdout, BUFFERED LINE BY LINE
 *  copyright (c) 1983 by Whitesmiths, Ltd.
 */
#include <std.h>

BOOL main(ac, av)
    BYTES ac;
    TEXT **av;
    {
    COUNT n;
    TEXT buf[BUFSIZE];

    while (n = getlin(buf, BUFSIZE))
        putlin(buf, n);
    return (YES);
    }
```

And this latest version is exactly equivalent to:

```
/*  COPY TEXT stdin TO stdout, BUFFERED LINE BY LINE
 *  copyright (c) 1983 by Whitesmiths, Ltd.
 */
#include <std.h>

BOOL main(ac, av)
    BYTES ac;
    TEXT **av;
    {
    IMPORT FIO stdin, stdout;
    COUNT n;
    TEXT buf[BUFSIZE];

    while (n = getl(&stdin, buf, BUFSIZE))
        putl(&stdout, buf, n);
    return (YES);
    }
```

The other pairs of routines for literal I/O read or write single characters, rather than lines of text, on each call. The most general ones are getc and putc. Given the address of an FIO buffer, getc returns either the next character available from the associated file, or the value -1 (named EOF in std.h) when it encounters end-of-file. The function putc takes the address of an FIO buffer, and the character to be transferred to it. If its second argument is a negative int-sized value, however, instead of an eight-bit character, putc simply flushes the FIO buffer to the associated file, and adds nothing to the buffer. putc always returns its second argument. (Note that if putc is to be used to output arbitrary

character data (as opposed to ASCII text) to a file, each character output should be masked to prevent its sign-extension into a negative integer; std.h provides the constant BYTMASK for this purpose.) The sample routine, rewritten to do literal buffered I/O character by character, looks almost identical to the line by line version:

```
/*  COPY TEXT INFILE TO OUTFILE, BUFFERED CHAR BY CHAR
 *  copyright (c) 1983 by Whitesmiths, Ltd.
 */
#include <std.h>

BOOL main(ac, av)
    BYTES ac;
    TEXT **av;
    {
    FIO infio, outfio;
    METACH ch;
    TEXT buf[BUFSIZE];

    fopen(&infio, "infile", READ);
    fcreate(&outfio, "outfile", WRITE);
    while ((ch = getc(&infio)) != EOF)
        putc(&outfio, ch);
    fclose(&outfio);
    fclose(&infio);
    return (YES);
    }
```

The type METACH (from std.h) is a synonym for short, as a reminder that getc can return the value EOF, in addition to character-sized values.

Like getl/putl, getc/putc have analogues that implicitly use stdin and stdout, instead of being called with an explicit FIO buffer. These routines, getch and putch, otherwise act exactly like getc and putc. Thus, to transfer a file character by character from stdin to stdout, the following suffices:

```
/*  COPY TEXT stdin TO stdout, BUFFERED CHAR BY CHAR
 *  copyright (c) 1983 by Whitesmiths, Ltd.
 */
#include <std.h>

BOOL main(ac, av)
    BYTES ac;
    TEXT **av;
    {
    METACH ch;
    TEXT buf[BUFSIZE];

    while ((ch = getch()) != EOF)
        putch(ch);
    return (YES);
    }
```

Unfortunately, the buffered I/O routines are affected by the text/binary distinction mentioned earlier. As noted, fopen and fcreate both automatically open files as text files. To access a binary file through an FIO buffer, you must open it using the portable interface functions open or create, then associate the returned file descriptor with an FIO buffer by calling the function finit. To adapt the sample program for copying a binary file using buffered I/O, one might use:

```
/*  COPY BINARY INFILE TO OUTFILE, BUFFERED CHAR BY CHAR
 *  copyright (c) 1983 by Whitesmiths, Ltd.
 */
#include <std.h>

BOOL main(ac, av)
    BYTES ac;
    TEXT **av;
    {
    FILE inf, outf;
    FIO infio, outfio;
    METACH ch;
    TEXT buf[BUFSIZE];

    inf = open("infile", READ, 1);
    outf = create("outfile", WRITE, 1);
    finit(&infio, inf, READ);
    finit(&outfio, outf, BWRITE);
    while ((ch = getc(&infio)) != EOF)
        putc(&outfio, ch & BYTMASK);
    fclose(&outfio);
    fclose(&infio);
    return (YES);
    }
```

finit expects three arguments: a valid file descriptor (returned by a previous open or create), the address of an FIO buffer, and a mode argument (like the one to fopen and fcreate) specifying the desired access to the file. Once finit associates the buffer with the specified file descriptor, the buffer can be used exactly like one set up by fopen or fcreate (in fact, fopen and fcreate are implemented simply as calls to open or create followed by a call to finit). You should note as well that, once a file descriptor has been associated with an FIO buffer, only the FIO buffer should be used to access the file.

fcreate, finit, and fopen all take one of the following mode arguments: BWRITE (-1), READ (0), or WRITE (1). The difference between BWRITE and WRITE is that a buffer established with BWRITE is flushed only when full; a buffer established with WRITE is flushed when full or when the most recent character transferred to it was a newline.

Output buffering has two further subtleties: whenever a buffer is established using WRITE, finit checks to see whether or not the associated file is one for which lseek calls are meaningful (e.g., a disk file as opposed to a terminal). If so, the buffer is established using BWRITE (so that true 512-character buffering is in effect); if not, WRITE is used.

(Remember that since fopen and fcreate call finit, they also act this way.) Further, output buffers explicitly established with finit are automatically drained on program termination, so that trailing partial lines or records are not lost. However, the buffer stdout, if used without explicit initialization, will be drained only when full or when a newline is written to it. So to make it fully safe, stdout should be initialized with an finit call specifying WRITE if being used as a text file:

```
finit(&stdout, STDOUT, WRITE);
```

or with a call specifying BWRITE if being used for non-text output:

```
finit(&stdout, STDOUT, BWRITE);
```

As always, stdout must be explicitly imported when it is named in this way.

All of the buffered I/O routines respond to errors by raising an appropriate error condition. If a routine is called with a NULL FIO buffer pointer, the fioerr condition is raised. On any read error, the readerr condition is raised. On any write error, the writerr condition is raised. By default, raising one of these conditions results in an immediate exit, reporting failure, with an appropriate message written to STDERR.

A few other cautions apply to using the buffered I/O routines: if an FIO buffer is already associated with a file, it must not be associated with a second file without an intervening fclose. Similarly, fclose must be given the address of an FIO buffer that is currently associated with exactly one open file. If finit is used to associate a file descriptor with an FIO buffer, only the FIO buffer should be used after that; the file descriptor should not be used for further I/O to the file now open through the FIO buffer. Finally, note that intermixed input and output through an FIO buffer will not work properly.

Formatted I/O Routines

All of the routines examined so far perform literal I/O, transferring characters more or less verbatim between memory and a file. A related group of functions performs formatted I/O: on output, a set of arguments is converted to a text string, under the control of a "format string" specifying the exact conversion to be applied to each argument. On input, a text string is converted into values to be assigned to a set of arguments, again under the control of a format string.

The functions getf and putf are the most general. Given the address of an FIO buffer, a format string, and a set of pointers to the variables that are to contain converted values, getf reads text from the buffer, converting it and assigning the result to each pointer in turn as specified by the format string. Given the address of an FIO buffer, a format string, and a set of arguments to be converted, putf converts each argument to a text string and outputs it to the buffer, as specified by the format string. The manual pages for getf and putf later in this section explain the formatting conventions used.

Like getl/putl, getf and putf have analogues that implicitly use stdin and stdout instead of an explicit FIO buffer. Except for the FIO buffer they use, these routines, getfmt and putfmt, act exactly like getf and putf. Naturally, just as no FIO buffer is named in a call to getlin/putlin, none is named in calling getfmt/putfmt, so that the format string becomes the first argument in the call. Except for this omission, the calling sequence for getfmt/putfmt is identical to the sequence for getf/putf.

One last formatted output routine is errfmt, which accepts a format string and a list of arguments to be converted, exactly like putfmt. errfmt, however, writes its converted text directly to the pre-defined file descriptor STDERR ("standard error"), to minimize the chance of error output being lost in the event of untimely termination.

To reiterate: getfmt uses the same formatting conventions as getf, while putfmt and errfmt use the same conventions as putf. So the description of formatting with getf or putf applies directly to the related routines, as well.

Summary of Functions

The following functions provide unbuffered I/O as part of the portable interface:

```
FILE close(fd)
FILE create(fname, mode, rsize)
COUNT lseek(fd, offset, sense)
FILE open(fname, mode, rsize)
COUNT read(fd, buf, size)
FILE remove(fname)
TEXT *uname()
COUNT write(fd, buf, size)
```

The following functions are convenient extensions to the portable interface:

```
VOID error(s1, s2)
COUNT fread(fd, buf, size)
VOID fwrite(fd, buf, size)
VOID putstr(fd, arg1, arg2, ..., NULL)
VOID remark(s1, s2)
```

The following entry points support buffered I/O using FIO buffers:

```
FIO *fclose(pfio)
FIO *fcreate(pfio, fname, mode)
FIO *finit(pfio, fd, mode)
FIO *fopen(pfio, fname, mode)
METACH getc(pfio)
METACH getch()
COUNT getf(pfio, fmt, arg1, arg2, ...)
COUNT getfmt(fmt, arg1, arg2, ...)
BYTES getl(pfio, s, n)
```

```
BYTES getlin(s, n)
COUNT putc(pfio, c)
COUNT putch(c)
VOID putf(pfio, fmt, arg1, arg2, ...)
VOID putfmt(fmt, arg1, arg2, ...)
BYTES putl(pfio, s, n)
BYTES putlin(s, n)
FIO stdin;
FIO stdout;
```

The following functions provide formatted I/O:

```
VOID errfmt(fmt, arg1, arg2, ...)
COUNT getf(pfio, fmt, arg1, arg2, ...)
COUNT getfmt(fmt, arg1, arg2, ...)
VOID putf(pfio, fmt, arg1, arg2, ...)
VOID putfmt(fmt, arg1, arg2, ...)
```

Note that all of the formatted I/O functions perform buffered I/O, except for errfmt, which performs unbuffered output.

SEE ALSO

Cint(III), Conventions, Portability(I), Runtime, std.h. The portable interface is described in general terms in Section III of this manual; specific details for a given operating system are in Section III of the appropriate C Interface Manual. The buffered and formatted I/O functions all are described on separate manual pages later in this section.

NAME

Runtime - command line processing using the standard libraries

FUNCTION

The standard libraries provide a consistent runtime environment across all implementations of Whitesmiths C. Two major components of this environment are a set of predefined file descriptors that simplify primitive I/O, and a set of text strings drawn from the command line used to run a user program. In non-Idris/UNIX implementations, both of these components are set up by a routine called _main, which is internal to the portable interface. This routine is called automatically by the initialization code that gets control at the start of execution of a user program. (Note that _main is completely separate from the routine main that you give in your source file; _main need not be -- and ordinarily should not be -- mentioned at all in your source code.)

Under Idris or UNIX, all of the features described here are provided by the operating system itself; _main, in fact, is designed to emulate this situation. To a user program, the two are equivalent, so the remainder of this essay uses the term "environment" to refer to either one.

The environment guarantees that three open text files are available to a user program when it begins execution. These files are assigned the first three file descriptors: STDIN (0), STDOUT (1), and STDERR (2). By default, STDIN ("standard input") is open for terminal input, while STDOUT ("standard output") is open for terminal output. Thus STDIN reads what you type at your keyboard, while STDOUT writes to your terminal screen. Using the command line features of the environment, STDIN and STDOUT can also be redirected to designate a named file, rather than your terminal. Because of this, STDERR ("standard error") is also provided for terminal output, as a separate means of sending error output to your terminal.

The environment also obtains a command line, that is, the line typed to your operating system in order to run your program. Where the operating system itself does not support command line specification, _main will prompt you directly for one. In either case, the command line can be used to pass options, filenames, or other string arguments to your program. The environment makes the contents of the command line available to your main program by transferring each non-white string on the line into a vector of NUL-terminated strings. The number of entries in this vector is passed as the first argument to main (conventionally called ac, for "argument count"). The vector itself is passed as the second argument to main (called av, for "argument vector").

The zeroeth element of av contains the name of the program being run, if available, otherwise the contents of the variable _pname. Later elements of av contain successive strings parsed from the command line. If a command line string is parsed that begins with the character '<', however, the environment uses the rest of the argument as a filename, and redirects STDIN to read from that file. If a string is parsed that begins with the character '>', the environment uses the rest of the argument as a filename, and redirects STDOUT to write to that file. Files opened in this way are always opened as text files. Command line strings used for redirection are not transferred into av.

A program to output each argument from its command line to STDOUT might be:

```
/*  OUTPUT COMMAND LINE ARGUMENTS
 *  copyright (c) 1983 by Whitesmiths, Ltd.
 */
#include <std.h>

BOOL main(ac, av)
    BYTES ac;
    TEXT **av;
    {
    COUNT n;

    for (n = 0; n < ac; ++n)
        {
        if (n)
            fwrite(STDOUT, " ", 1);
        fwrite(STDOUT, av[n], lenstr(av[n]));
        }
    fwrite(STDOUT, "\n", 1);
    return (YES);
    }
```

Other routines in the standard libraries encourage a command line format consisting of a program name, an optional series of flags (i.e., option names), and a series of filenames, that is:

```
<name> <flags> <files>
```

Given a command line in this format, the routine getflags can be used to skip the program name, then parse any flags present. A "flag" is considered to be an argument whose first character is '+' or '-'; getflags will assign values to variables within your program based on the flags it encounters. Once getflags has been called to get the program name and any flags out of the way, the routines getfiles or getbfiles can be used to scan the command line filenames given, opening each file in sequence.

A simple-minded program to list text files to STDOUT might be:

```
/*  LIST TEXT FILES TO STDOUT
 *  copyright (c) 1983 by Whitesmiths, Ltd.
 */
#include <std.h>

/*  flags:
    -n      output name of file before listing it
 */
BOOL nflag {NO};

BOOL main(ac, av)
    BYTES ac;
    TEXT **av;
    {
    IMPORT BOOL nflag;
    IMPORT FIO stdout;
    BOOL ok;
    COUNT n;
    FILE fd;
    FIO inf;
    TEXT buf[BUFSIZE];

    getflags(&ac, &av, "n: <files>", &nflag);
    for (ok = YES; 0 <= (fd = getfiles(&ac, &av, STDIN, STDERR)); )
        {
        if (nflag)
            putstr(STDOUT, av[-1], ":\n", NULL);
        if (fd == STDERR)
            {
            remark("can't read: ", av[-1]);
            ok = NO;
            }
        else
            {
            finit(&inf, fd, READ);
            while (n = getl(&inf, buf, BUFSIZE))
                putlin(buf, n);
            fclose(&inf);
            }
        }
    return (ok);
    }
```

This routine would read each file given on its command line, and output each to the standard output, using the library-defined FIO buffer stdout. If binary files were to be accessed, then the routine getbfiles, rather than getfiles, would be used. getbfiles (named for "get binary files") adds a fifth argument to the getfiles calling sequence, which is used as a record size value in opening each file. The routine getfiles can open text files only.

One final task performed by the environment is to pass the return value of your main program back to the runtime startup code, which in turn passes it back to the operating system, if at all possible.

The command line interface available in your environment may have more features than those described here; for specifics on non-Idris/UNIX environments, see the description of _main for your operating system, in the appropriate C Interface Manual. For Idris or UNIX, the description of your shell should do the trick.

SEE ALSO

Cint(III), Files, _pname(III), getbfiles, getfiles, getflags, main(III)

NAME

FIO - the file input/output structure

SYNOPSIS

FIO stdin, stdout;

FUNCTION

FIO is the type defined in <std.h> for the control buffers used by many of the C library input/output routines. Its elements are:

FILE _fd - holds the file descriptor for the file with which input/output is performed.

COUNT _nleft - on input, tells how many characters are left undelivered in the buffer; on output, tells how many characters have been placed in the buffer for output. Setting _nleft to zero is sufficient to initialize an FIO buffer. Input routines set _nleft to -1 on end of file, as an indication that no further reads should be attempted.

COUNT _fmode - is set to BWRITE, READ, or WRITE to indicate the mode of operation.

TEXT *_pnext - on input, points to the next character to be delivered; on output, used to chain FIO buffers for draining on exit. If (_nleft == 0) on input, _pnext is undefined.

TEXT _buf[BUFSIZE] - is the character buffer, where BUFSIZE is 512.

BWRITE is a mode not recognized by the low level interface routines. It is used to indicate buffered writing, i.e., output only on buffer full or program exit. Normal output mode calls for draining the buffer whenever an output sequence ends with a newline.

All actual input using FIO control buffers is via getc or getl. All actual output is via putc or putl.

NAME

V7lib - a library for UNIX support under the standard compiler

FUNCTION

The UNIX interface library serves two purposes: First, it allows programs written using the UNIX/V7 (abbreivated V7) runtime library to be ported, with a small loss in efficiency, into the Whitesmiths (abbreviated WSL) runtime environment, using the standard libraries. The entry points provided also represent a large subset of the library for UNIX System III or UNIX System V.

Second, it allows people studying Kernighan and Ritchie´s (abbreviated K+R) C Programming Language to run their program exercises using the standard Whitesmiths compiler, by performing the same mapping of V7-style library calls onto standard library calls.

The second goal is easier, and can be guaranteed for all functions mentioned in K+R, except for the UNIX-dependent ones described in Chapter 8. The first purpose crosses many grey areas:

The original V7 library is not partitioned into portable subsets. Thus, programs using it may contain UNIX dependencies -- like calling fork() or popen().

Direct access (seek) varies widely under different operating systems. V7lib supports direct access to whatever extent the underlying system will allow. Under Idris, this is almost exactly what V7 permits; on other systems, seek often cannot be used relative to end-of-file, and direct access may not be allowed to text files.

Some non-UNIX systems distinguish between text files and binary files. The default file type of V7lib is text file, but this can be overridden by setting the global variable _recsize to a non-zero value.

Programs to be run under V7lib should contain

```
#include <stdio.h>
```

at the start of the file. Normally, an existing V7 program contains this line.

V7lib is available under some variant of the name "libu", subject to local operating system convention. Compilation scripts making use of V7lib are provided with each standard distribution.

Components of V7lib

Users of K+R can simply use the descriptions of functions given in that book. For more exact specification of the V7 library, users must obtain a copy of the V7 User´s Manual (available for public purchase from Bell Laboratories). The following notes can then be appended to each manual page.

Each note tells whether V7lib supports the function; if so, restrictions are noted; if not, reasons are given. Most of the portable library functions are included in V7lib. The functions that are not in V7lib fall into these categories: UNIX-specific (which you may nonetheless be able to port); math functions (which would be easy to port, provided you attended to numerical analysis); semi-obsolete functions (which are discouraged even in UNIX environments).

Functions Supported

V7 Manual Section 2 (System Calls):

Most UNIX system calls are system-dependent and therefore not in V7lib. The following functions are supported by V7lib:

```
sbrk    - preprocessed by stdio.h to WSL sbreak (changes return value)
close   - as-is in WSL
creat   - preprocessed by stdio.h to WSL create (uses global _recsize)
exit    - calls V7lib exit7 which produces YES or NO return
lseek   - present (but only returns offset, not true "tell")
open    - as-is in WSL (needs third arg on non-UNIX/Idris systems)
read    - as-is in WSL
unlink  - preprocessed by stdio.h into WSL remove
write   - as-is in WSL
```

V7 Manual Section 3 (Library Functions):

The following V7 entry points are provided by V7lib with no changes visible to the user:

```
atof     fileno   isalnum  isspace  qsort    strcpy
atoi     fprintf  isalpha  isupper  rand     strlen
atol     fputc    isascii  log10    scanf    strncat
calloc   fputs    iscntrl  malloc   sprintf  strncmp
feof     fread    isdigit  pow      srand    strncpy
fflush   fscanf   islower  putc     sscanf   strrchr
fgetc    fwrite   isprint  putchar  strchr   ungetc
fgets    gets     ispunct  puts     strcmp
```

The following V7 entry points are provided unchanged by the Whitesmiths standard libraries:

```
cos      exp      free     sin      sqrt
```

The following entry points are provided by V7lib, with the reservations noted. In particular, note that V7lib defines the character comparison entry points available with <ctype.h> in later versions of UNIX.

```
_tolower - from ctype.h; not in standard V7
_toupper - from ctype.h; not in standard V7
atan     - preprocessed by stdio.h into WSL arctan
fclose   - no return value
fopen    - "a" mode works only on some systems
fseek    - some systems may have text/binary distinction
```

```
getc      - macro, as it is in V7
getchar   - macro, as it is in V7
index     - synonym for strchr
isgraph   - from ctype.h; not in standard V7
isxdigit  - from ctype.h; not in standard V7
log       - preprocessed by stdio.h into WSL ln
printf    - %g,%G = %d
rindex    - synonym for strrchr
stdio     - different internals
strcat    - macro; avoid side-effects on args
toascii   - from ctype.h; not in standard V7
tolower   - from ctype.h; not in standard V7
toupper   - from ctype.h; not in standard V7
```

Functions Not Supported

The following UNIX-specific or obsolete functions are not provided in V7lib:

```
abort     end       getlogin  monitor   sleep
abs       fdopen    getpass   nlist     system
assert    ferror    getpw     perror    ttyname
clearrer  freopen   getpwent  pkopen
ctime     getenv    getw      putw
dbm       getgrent  l3tol     realloc
```

The following math-related functions are not provided by V7lib:

```
acos      atan2     floor     hypot     mp
asin      ecvt      frexp     j0        sinh
```

The following functions, though not in V7lib, have very close equivalents in the WSL standard libraries:

```
crypt     - proprietary algorithm (see WSL crypt)
mktemp    - see WSL uname
setbuf    - see WSL finit
setjmp    - see WSL enter
swab      - machine-dependent usage (see WSL lstos)
```

SEE ALSO

Differences(I), for a discussion of differences between the UNIX/V7 and Whitesmiths compilers.

BUGS

Some rather obscure special cases are uncommon, but must be noted. A few V7 functions are implemented as macros in V7lib: strcat, strcpy, strlen. There will thus be no corresponding names in loader maps (e.g., for breakpointing), and side-effects should be avoided on strcat, which repeats an argument. Also, the V7 function names exit, fclose, fopen, fread, fwrite, and lseek, since they match names from the standard libraries, are mapped by stdio.h into the names exit7, fclos7, fopen7, fread7, fwrit7, and lsek7. These last names will appear in any executable file.

The formatted output functions (printf, fprintf, sprintf) are implemented by converting their format string to a WSL format string. "G" or "g" format always becomes "d".

NAME

_memerr - no memory error condition

SYNOPSIS

```
TEXT *_memerr
```

FUNCTION

_memerr is the condition raised when stack or heap space is exhausted. If no handler is provided the message written to STDERR is "no memory".

NAME
abs - find absolute value

SYNOPSIS
abs(a)

FUNCTION
abs obtains the absolute value of its argument. Since abs is implemented as a C preprocessor macro, its argument can be any numerical type.

RETURNS
abs is a numerical rvalue of the form ((a < 0) ? -a : a), suitably parenthesized.

EXAMPLE

```
putfmt("balance %i%p\n", abs(bal), (bal < 0) ? "CR" : "");
```

BUGS
Because it is a macro, abs cannot be called from non-C programs, nor can its address be taken. An argument with side effects may be evaluated other than just once.

NAME

alloc - allocate space on the heap

SYNOPSIS

```
TEXT *alloc(nbytes, link)
    BYTES nbytes, link
```

FUNCTION

alloc allocates space on the heap for an item of size nbytes, then writes link in the zeroeth integer location. The space allocated is guaranteed to be at least nbytes long, starting from the pointer returned, which pointer is guaranteed to be on a proper storage boundary for anything. The heap is grown as necessary; if space is exhausted, the _memerr condition is raised.

RETURNS

If alloc returns, the pointer is guaranteed not to be NULL.

EXAMPLE

To build a stack:

```
struct cell {
    struct cell *prev;
    ... rest of cell ...
    } *top;

top = alloc(sizeof (*top), top);    /* pushes a cell */
```

SEE ALSO

_memerr, buybuf, free, frelst, nalloc, sbreak(III)

BUGS

The size of the allocated cell is stored in the integer location right before the usable part of the cell; hence it is easily clobbered. This number is related to the actual cell size in a most system dependent fashion and should not be trusted.

NAME

amatch - look for anchored match of regular expression

SYNOPSIS

```
BYTES amatch(buf, n, idx, pat, psubs)
    TEXT *buf;
    BYTES n, idx;
    TEXT *pat;
    struct {
        TEXT *mtext;
        BYTES mlen;
        } *psubs;
```

FUNCTION

amatch tests the n character buffer starting at buf[idx] for a match with the encoded pattern starting at pat; the match is constrained to match characters starting at buf[idx]. It is assumed that the pattern was built by the function pattern, whose manual page describes the notation for regular expressions accepted by these routines.

If (psubs is not NULL) then every balanced pair \(...\) within the pattern will have the substring it matches recorded at psubs[i], where i counts up from one for the leftmost "\(" in the pattern. psubs[i] .mtext points at the first character of the matching substring, and psubs[i] .mlen is its length. psubs[0] always records the full match.

The pattern codes are a sequence of bytes with the values:

value	name	meaning
1	CCHAR	literal character follows
2	ANY	match anything but \n
3	SBOL	match beginning of line (0 width)
4	SEOL	match end of line, or just before ending \n
5	CLOSE	match following pattern zero or more times
6	CCL	character class follows (CCHARs or RANGEs)
7	NCCL	negated character class follows
8	RANGE	lower and upper bound characters follow
9	CCLEND	character class ends
10	PEND	pattern end
19	RPAR	right parenthesis "\)", followed by a one-byte order number
20	LPAR	left parenthesis "\(", followed by a one-byte order number

These codes need be known only if patterns are to be built by hand.

RETURNS

If the match was successful, amatch returns the index of the first character in buf that was not matched. Otherwise amatch returns -1. The array at psubs is also filled in, if present.

EXAMPLE

To examine stdin for lines starting with a pattern specified by the first command line argument, and output matching lines to stdout:

```
if (pattern(pbuf, av[1][0], &av[1][1]))
    while (n = getlin(buf, MAXBUF))
        if ((n = amatch(buf, n, 0, pbuf, NULL)) != -1)
            putlin(buf, m);
```

SEE ALSO

match, pattern

BUGS

Closures will not work properly unless buf is terminated by a newline. In general, amatch is a sucker for bad patterns.

NAME

arctan - arctangent

SYNOPSIS

```
DOUBLE arctan(x)
    DOUBLE x;
```

FUNCTION

arctan computes the angle in radians whose tangent is x, to full double precision. It works by folding x into the interval [0, 1], then interpolating from an eight entry table, using the sum of tangents formula and a fifth order telescoped Taylor series approximation.

RETURNS

arctan returns the nearest internal representation to arctan x, expressed as a double floating value in the interval (-pi/2, pi/2).

EXAMPLE

To find the phase angle of a vector:

```
IMPORT DOUBLE arctan();
    ...
theta = arctan(y / x) * 180.0 / pi;
```

NAME

bldks - build key schedule from key

SYNOPSIS

```
TINY *bldks(ks, key)
    TINY ks[16][8];
    TEXT key[8];
```

FUNCTION

bldks builds the key schedule used by the Data Encryption Standard algorithm for encrypting or decrypting data. All eight characters of key are used to form the key schedule, but the most significant bit of each byte is ignored.

RETURNS

bldks returns the address of ks, which contains the key schedule.

EXAMPLE

To decrypt a file given a key already stored in passwd:

```
bldks(ks, passwd);
while (read(STDIN, buf, 8) == 8)
    write(STDOUT, decrypt(buf, ks), 8);
```

SEE ALSO

decrypt, encrypt

NAME

btod - convert buffer to double

SYNOPSIS

```
BYTES btod(s, n, pdnum)
    TEXT *s;
    BYTES n;
    DOUBLE *pdnum;
```

FUNCTION

btod converts the n character string starting at s into a double, and stores it at pdnum. The string is taken as the text representation of a decimal number, with an optional fraction and exponent. Leading whitespace is skipped and an optional sign is permitted; conversion stops at the end of the buffer or on the first unrecognizable character. Acceptable inputs match the pattern

```
[+|-]d*[.d*][e[+|-]dd*]
```

where d is any decimal digit and e is 'e' or 'E'.

No checks are made against overflow, underflow, or completely silly character strings.

RETURNS

btod returns the number of characters actually consumed, which is typically greater than zero but never larger than n. The converted number is stored at pdnum.

EXAMPLE

To convert a program's first command line argument into a double at dbl:

```
DOUBLE dbl;
    ...
if (2 < ac)
    btod(av[1], lenstr(av[1]), &dbl);
else
    dbl = 0.0;
```

SEE ALSO

dtento, dtoe, dtof

BUGS

Nothing simple can be said about the properties of a number that has overflowed.

NAME

btoi - convert buffer to integer

SYNOPSIS

```
BYTES btoi(s, n, pinum, base)
    TEXT *s;
    BYTES n, *pinum;
    COUNT base;
```

FUNCTION

btoi converts the n character string starting at s into an integer, and stores it at pinum. The string is taken as the text representation of a number in the base specified. Leading whitespace is skipped and an optional sign is permitted; if (base == 16) a leading "0x" or "0X" is skipped; conversion stops at the end of the buffer or on the first unrecognizable character. If the stop character is 'l' or 'L', it is skipped over.

Acceptable characters are the decimal digits and letters, either upper or lower case, where the letter 'a' (or 'A') has the value 10, as in the usual representation for hexadecimal. Letters with values greater than or equal to base are not acceptable digits. Thus values of base from 2 to 36 are meaningful.

If base is 1, it is taken as a directive to adapt the base to the input string, following the normal rules of C; i.e., if the string starts with "0x" or "0X" base 16 is used, else if the string starts with "0" base 8 is used, else base 10 is used.

No checks are made against overflow, unreasonable values of base, or completely silly character strings.

RETURNS

btoi returns the number of characters actually consumed, which is typically greater than zero but never larger than n. The converted number is stored at pinum.

EXAMPLE

```
BYTES num;
    ...
if (btoi(buf, size, &num, 10) != size)
    putstr(STDERR, "not a decimal number\n", NULL);
```

SEE ALSO

btol, btos, itob, ltob, stob

BUGS

Nothing simple can be said about the properties of a number that has overflowed.

NAME

btol - convert buffer to long

SYNOPSIS

```
BYTES btol(s, n, plnum, base)
    TEXT *s;
    BYTES n;
    LONG *plnum;
    COUNT base;
```

FUNCTION

btol converts the n character string starting at s into a long integer, and stores it at plnum. The string is taken as the text representation of a number in the base specified. Leading whitespace is skipped and an optional sign is permitted; if (base == 16) a leading "0x" or "0X" is skipped; conversion stops at the end of the buffer or on the first unrecognizable character. If the stop character is 'l' or 'L' it is skipped.

Acceptable characters are the decimal digits and letters, either upper or lowercase, where the letter 'a' (or 'A') has the value 10, as in the usual representation for hexadecimal. If a letter has a value greater than or equal to base, it is not an acceptable digit. Thus values of base from 2 to 36 are meaningful.

If base is 1, it is taken as a directive to adapt the base to the input string, following the normal rules of C; i.e., if the string starts with "0x" or "0X" base 16 is used, else if the string starts with "0" base 8 is used, else base 10 is used.

No checks are made against overflow, unreasonable values of base, or completely silly character strings.

RETURNS

btol returns the number of characters actually consumed, which is typically greater than zero but never larger than n. The converted number is stored at plnum.

EXAMPLE

```
LONG lnum;
    ...
if (btol(buf, size, &lnum, 16) != size)
    putstr(STDERR, "not a hexadecimal number\n", NULL);
```

SEE ALSO

btoi, btos, itob, ltob, stob

BUGS

Nothing simple can be said about the properties of a number that has overflowed.

NAME

btos - convert buffer to short integer

SYNOPSIS

```
BYTES btos(s, n, pinum, base)
    TEXT *s;
    BYTES n;
    COUNT *pinum, base;
```

FUNCTION

btos converts the n character string starting at s into a short integer, and stores it at pinum. The string is taken as the text representation of a number to the base specified. Leading whitespace is skipped and an optional sign is permitted; if (base == 16) a leading "0x" or "0X" is skipped; conversion stops at the end of the buffer or on the first unrecognizable character. If the stop character is 'l' or 'L', it is skipped over.

Acceptable characters are the decimal digits and letters, either upper or lower case, where the letter 'a' (or 'A') has the value 10, as in the usual representation for hexadecimal. Letters with values greater than or equal to base are not acceptable digits. Thus values of base from 2 to 36 are meaningful.

If base is 1, it is taken as a directive to adapt the base to the input string, following the normal rules of C; i.e., if the string starts with "0x" or "0X" base 16 is used, else if the string starts with "0" base 8 is used, else base 10 is used.

No checks are made against overflow, unreasonable values of base, or completely silly character strings.

RETURNS

btos returns the number of characters actually consumed, which is typically greater than zero but never larger than n. The converted number is stored at pinum.

EXAMPLE

```
COUNT snum;
    ...
if (btos(buf, size, &snum, 8) != size)
    putstr(STDERR, "not an octal number\n", NULL);
```

SEE ALSO

btoi, btol, itob, ltob, stob

BUGS

Nothing simple can be said about the properties of a number that has overflowed.

NAME

buybuf - allocate a cell and copy in text buffer

SYNOPSIS

```
TEXT *buybuf(s, n)
    TEXT *s;
    BYTES n;
```

FUNCTION

buybuf allocates a cell of size n on the heap by calling alloc, then copies the n characters starting at s into it. If the heap is full, alloc raises the _memerr condition, as described on the alloc manual page.

RETURNS

The value returned is the pointer to the allocated cell.

EXAMPLE

To read a text file into memory:

```
    struct
        {
        TEXT *text;
        BYTES size;
        } lines[];
        ...
    for (p = lines; 0 < (n = getlin(buf, BUFSIZE)); ++p)
        {
        p->text = buybuf(buf, n);
        p->size = n;
        }
```

SEE ALSO

alloc, free, nalloc

NAME

cmpbuf - compare two buffers for equality

SYNOPSIS

```
BOOL cmpbuf(s1, s2, n)
    TEXT *s1, *s2;
    BYTES n;
```

FUNCTION

cmpbuf compares two text buffers, character by character, for equality. The first buffer starts at s1, the second at s2; both are n characters long. s1 and s2 are said to be equal if the n characters in s1 and s2 are identical.

RETURNS

The value returned is YES if the buffers are equal, else NO.

EXAMPLE

```
if (cmpbuf(name, "include", 7))
    doinclude();
```

SEE ALSO

cmpstr, prefix

NAME

cmpstr - compare two strings for equality

SYNOPSIS

```
BOOL cmpstr(s1, s2)
    TEXT *s1, *s2;
```

FUNCTION

cmpstr compares two strings, character by character, for equality. The first string starts at s1 and is terminated by a NUL '\0'; the second is likewise described by s2. The strings must match through and including their terminating NUL characters.

RETURNS

The value returned is YES if the strings are equal, else NO.

EXAMPLE

```
if (cmpstr(name, "include"))
    doinclude();
```

SEE ALSO

cmpbuf, prefix

NAME

cos - cosine in radians

SYNOPSIS

```
DOUBLE cos(x)
    DOUBLE x;
```

FUNCTION

cos computes the cosine of x, expressed in radians, to full double precision. It works by scaling x in quadrants, then computing the appropriate sin or cos of an angle in the first half quadrant, using a sixth order telescoped Taylor series approximation. If the magnitude of x is too large to contain a fractional quadrant part, the value of cos is 1.

RETURNS

cos returns the nearest internal representation to cos x, expressed as a double floating value.

EXAMPLE

To rotate a vector through the angle theta:

```
IMPORT DOUBLE cos(), sin();
    ...
xnew = xold * cos(theta) - yold * sin(theta);
ynew = xold * sin(theta) + yold * cos(theta);
```

SEE ALSO

sin

NAME

cpybuf - copy one buffer to another

SYNOPSIS

```
BYTES cpybuf(s1, s2, n)
    TEXT *s1, *s2;
    BYTES n;
```

FUNCTION

cpybuf copies the first n characters starting at location s2 into the buffer beginning at s1.

RETURNS

The value returned is n, the number of characters copied.

EXAMPLE

To place "first string, second string" in buf[]:

```
n = cpybuf(buf, "first string", 12);
cpybuf(buf + n, ", second string", 15);
```

SEE ALSO

cpystr

NAME

cpystr - copy multiple strings

SYNOPSIS

```
TEXT *cpystr(ds, arg1, arg2, arg3, arg4, ..., NULL)
    TEXT *ds, *arg1, *arg2, *arg3, *arg4, ...;
```

FUNCTION

cpystr concatenates a series of strings into the destination string ds. Each string begins at argx and is terminated by a NUL '\0'. The first character of arg2 is placed just after the last character (before the NUL) copied from arg1, etc. The series of string arguments is terminated by a NUL pointer argument. A NUL is appended to the final destination string to terminate it properly.

RETURNS

The value returned is a pointer to the terminating NUL in the destination string.

EXAMPLE

To concatenate string ss1 with " middle ", ss2, and " end." into buf:

```
cpystr(buf, ss1, " middle ", ss2, " end.", NULL);
```

BUGS

There is no way to specify the size of the destination area, to prevent storage overwrites. Forgetting the terminating NULL pointer is usually disastrous.

NAME

decode - convert arguments to text under format control

SYNOPSIS

```
BYTES decode(s, n, fmt, arg1, arg2, ...)
    TEXT *s;
    BYTES n;
    TEXT *fmt;
```

FUNCTION

decode writes characters to the n character buffer starting at s exactly as if the contents were written to a file by putf, using the format string fmt and the zero or more arguments arg1, arg2, ... It is not considered an error to generate more characters than will actually fit in the buffer; excess characters are simply discarded.

RETURNS

decode returns the number of characters actually written in the buffer, a number between 0 and n, inclusive.

EXAMPLE

To convert the integer symno to a symbolic name:

```
decode(&name, 6, "L%+05i", symno);
```

SEE ALSO

dtoe, dtof, encode, putf, putfmt

NAME

decrypt - decode encrypted block of text

SYNOPSIS

```
TEXT *decrypt(data, ks)
    TEXT data[8];
    TINY ks[16][8];
```

FUNCTION

decrypt converts the eight characters in the buffer data to decrypted form in place, using the key schedule constructed in ks by the function bldks. The Data Encryption Standard algorithm is used, taking bit 1 as the least significant bit of data[0] and bit 64 as the most significant bit of data[7].

RETURNS

decrypt returns a pointer to the start of data, which contains the decrypted text.

EXAMPLE

To decrypt a file given a key already stored in passwd:

```
bldks(ks, passwd);
while (read(STDIN, buf, 8) == 8)
    write(STDOUT, decrypt(buf, ks), 8);
```

SEE ALSO

bldks, encrypt

NAME

doesc - process character escape sequences

SYNOPSIS

```
COUNT doesc(pp, magic)
    TEXT **pp, *magic;
```

FUNCTION

doesc encodes the sequence of characters beginning at *pp, on the assumption that (*pp)[0] is an escape character, following the same escape conventions as the C compiler. It also updates the pointer at pp to point to the last character of the (variable length) escape sequence.

If ((*pp)[1] is NUL) the code value is (*pp)[0], i.e., the escape character proper; this is the only escape sequence of length one. If ((*pp)[1] is a digit) then up to three digits are taken as the octal value of the code. If ((*pp)[1] is in the sequence "bfnrtv", in either case) the code is the corresponding member of the sequence (backspace, formfeed, newline, carriage return, horizontal tab, vertical tab). If (magic is not NULL) and (*pp)[1] is the ith character of the NUL terminated string at magic, the code is (-1 - i). Otherwise the code is (*pp)[1].

In all cases, *pp is updated to point at the last character consumed.

RETURNS

doesc returns the code obtained, and updates the pointer *pp as necessary to point at the last character of the escape sequence.

EXAMPLE

```
for (s = buf; *s; ++s)
    *t++ = (*s == '\\') ? doesc(&s, NULL) : *s;
```

SEE ALSO

mapchar

NAME

dtento - multiply double by a power of ten

SYNOPSIS

```
DOUBLE dtento(d, exp)
    DOUBLE d;
    COUNT exp;
```

FUNCTION

dtento multiplies the double d by 10**exp. No check is made for overflow or underflow.

RETURNS

dtento returns d * 10**exp as a double.

EXAMPLE

To combine a fraction string and an exponent string:

```
btoi(fr, nfr, &intpart, 10);
btoi(sexp, nsexp, &exp, 10);
dbl = dtento((DOUBLE)intpart, exp - nfr);
```

SEE ALSO

btod, dtoe, dtof

BUGS

If the exponent is large in magnitude, dtento can loop for quite a long time. No special consideration is given (d == 0.0).

NAME

dtoe - convert double to buffer in exponential format

SYNOPSIS

```
BYTES dtoe(s, dbl, p, q)
    TEXT *s;
    DOUBLE dbl;
    BYTES p, q;
```

FUNCTION

dtoe converts the double number dbl to a text representation in the buffer starting at s, having the format:

```
[-]d*.d*e{+|-}d*
```

where d is a decimal digit. p specifies the number of digits to the left of the decimal point, and q the number to the right. There are either two or three digits in the exponent, depending upon the target machine.

RETURNS

The value returned is the number of characters used to represent the double number.

EXAMPLE

```
putfmt("area = %b\n", buf, dtoe(buf, area, 1, 5));
```

SEE ALSO

dtof

NAME

dtof - convert double to buffer in fixed-point format

SYNOPSIS

```
BYTES dtof(s, dbl, p, q)
    TEXT *s;
    DOUBLE dbl;
    BYTES p, q;
```

FUNCTION

dtof converts the double number dbl to a text representation in the buffer starting at s, having the format:

```
[-]d*.d*
```

where d is a decimal digit. p specifies the maximum number of digits to the left of the decimal point, and q the actual number to the right.

RETURNS

The value returned is the number of characters used to represent the double number.

EXAMPLE

```
putfmt("area = %b\n", buf, dtof(buf, area, 10, 5));
```

SEE ALSO

dtoe

NAME

encode - convert text to arguments under format control

SYNOPSIS

```
COUNT encode(s, n, fmt, parg1, parg2, ...)
    TEXT *s;
    BYTES n;
    TEXT *fmt;
```

FUNCTION

encode converts the contents of the n character buffer starting at s, using the format string at fmt and the argument pointers pargx, exactly as if the contents were read from a file by getf. It is particularly useful when multiple attempts must be made to read an input line.

RETURNS

encode returns the number of arguments successfully converted, or EOF (-1) if end of buffer is encountered before any are converted.

EXAMPLE

To read lines from stdin, each consisting of "x = <integer>" or "y = <integer>":

```
while (0 < (n = getlin(buf, BUFSIZE)))
    if (encode(buf, n, "x = %i", &x) <= 0 &&
        encode(buf, n, "y = %i", &y) <= 0)
        errfmt("unknown parameter %b\n", buf, n);
```

SEE ALSO

btod, decode, getf, getfmt

NAME

encrypt - encode block of text

SYNOPSIS

```
TEXT *encrypt(data, ks)
    TEXT data[8];
    TINY ks[16][8];
```

FUNCTION

encrypt converts the eight characters in the buffer data from encrypted form in place, using the key schedule constructed in ks by the function bldks. The Data Encryption Standard algorithm is used, taking bit 1 as the least significant bit of data[0] and bit 64 as the most significant bit of data[7].

RETURNS

encrypt returns a pointer to the start of data, which contains the encrypted text.

EXAMPLE

To encrypt a file given a key already stored in passwd:

```
bldks(ks, passwd);
while (0 < (n = read(STDIN, buf, 8)))
    {
    while (n < 8)
        buf[n++] = '\0';
    write(STDOUT, encrypt(buf, ks), 8);
    }
```

SEE ALSO

bldks, decrypt

NAME

enter - enter a control region

SYNOPSIS

```
BYTES enter(pfn, arg)
    BYTES (*pfn)();
    BYTES arg;
```

FUNCTION

enter establishes a new "control region", i.e., a function invocation that can be terminated early by a leave call, then performs the sequence

```
leave((*pfn)(arg));
```

i.e., the function pointed at by pfn is called with the specified arg; its return value is used as the argument of a call to leave. The control region may be terminated before (*pfn) returns by a call to leave in (*pfn) or in any of its dynamic descendants. In any case, the first leave call encountered disestablishes the new control region and causes enter to return with the value specified by the argument to that call to leave.

Control regions may be nested to any depth.

RETURNS

enter returns the value of the argument to the first leave call encountered, or the value of the function at pfn if no leave was executed.

EXAMPLE

To restart a function after each error message:

```
while (s = enter(&func, file))
    putstr(STDERR, s, "\n", NULL);
```

so that one can write in, say, one of func's dynamic descendants:

```
if (counterr)
    leave("missing parameter");
```

SEE ALSO

_raise(IV), _when(IV), leave

BUGS

There is no way to pass more than one argument to the function (*pfn).

NAME

errfmt - format output to error file

SYNOPSIS

```
VOID errfmt(fmt, arg1, arg2, ...)
    TEXT *fmt;
```

FUNCTION

errfmt performs formatted output to STDERR, in much the same way as putf. Output is performed by multiple calls directly to write, which may be inefficient for large volumes of output but is least likely to lose diagnostics when a program malfunctions.

RETURNS

Nothing. An error exit occurs if any writes fail.

EXAMPLE

```
errfmt("can't open file %p\n", fname);
```

SEE ALSO

putf, putfmt

NAME

error - print error message and exit

SYNOPSIS

```
VOID error(s1, s2)
    TEXT *s1, *s2;
```

FUNCTION

error prints an error message to STDERR, consisting of the program name _pname, a colon and space, the strings at s1 and s2, and a newline. It then takes an error exit. Either s1 or s2 may be NULL.

RETURNS

error never returns to its caller.

EXAMPLE

```
if ((fd = open(file, READ, 0)) < 0)
    error("can't open ", file);
```

SEE ALSO

_pname(III), exit(III)

NAME

exp - exponential

SYNOPSIS

```
DOUBLE exp(x)
    DOUBLE x;
```

FUNCTION

exp computes the exponential of x to full double precision. It works by expressing x/ln 2 as an integer plus a fraction in the interval (-1/2, 1/2]. The exponential of the fraction is approximated by a ratio of two seventh order polynomials.

RETURNS

exp returns the nearest internal representation to exp x, expressed as a double floating value. If the result is too large to be properly represented, a range error condition is raised; if that is inhibited, the largest representable value is returned.

EXAMPLE

```
IMPORT DOUBLE exp();
    ...
sinh = (exp(x) - exp(-x)) / 2.0;
```

SEE ALSO

_range(IV), ln

NAME

fclose - close a file controlled by FIO buffer

SYNOPSIS

```
FIO *fclose(pfio)
    FIO *pfio;
```

FUNCTION

fclose closes the file under control of the FIO buffer at pfio. If the control buffer was initialized with a mode of WRITE or BWRITE, any remaining output is drained before closing, and the control buffer is removed from the list of buffers to be drained on program exit.

RETURNS

fclose returns pfio if the file was successfully closed, else NULL. If (pfio == NULL), fclose raises the fioerr condition.

SEE ALSO

fcreate, finit, fioerr, fopen

NAME

fcreate - create a file and initialize a control buffer

SYNOPSIS

```
FIO *fcreate(pfio, fname, mode)
    FIO *pfio;
    TEXT *fname;
    COUNT mode;
```

FUNCTION

fcreate creates a file with name fname and specified mode, and if successful initializes the control buffer at pfio for proper operation with the file. mode should have one of the values BWRITE, READ, or WRITE. pfio must be the address of an FIO buffer declared by the user, or the address of the buffer stdin or stdout, both of which are provided by this library.

If (mode == BWRITE) the control buffer is set up for buffered writes, to be drained only when the buffer is full or the program exits. If (mode == READ) the control buffer is set up for reading. Otherwise (mode == WRITE) of necessity and the control buffer is set up for writing; writes will be buffered as for BWRITE only if lseek calls are acceptable with the specified fd, an indication that the output is a file and not an interactive device or a pipeline. Unbuffered output is drained whenever a segment of output ends with a newline character, or on program termination.

RETURNS

fcreate returns pfio, if successful, else NULL. If (pfio == NULL), the fioerr condition is raised.

EXAMPLE

```
FIO outf;
  ...
if (!fcreate(&outf, "file", WRITE))
    errfmt("can't create file\n");
```

SEE ALSO

create(III), fclose, finit, fioerr, fopen

BUGS

fcreate always creates fname as a text file; on systems where text files and binary files are distinguished, create followed by finit must be used to associate a binary file with an FIO buffer.

NAME

fill - propagate fill character throughout buffer

SYNOPSIS

```
BYTES fill(s, n, c)
    TEXT *s, c;
    BYTES n;
```

FUNCTION

fill floods the n-character buffer starting at s with fill character c.

RETURNS

fill returns n.

EXAMPLE

To write a 512-byte buffer of NULs:

```
write(fd, buf, fill(buf, BUFSIZE, '\0'));
```

SEE ALSO

squeeze

NAME

finit - initialize an FIO control buffer

SYNOPSIS

```
FIO *finit(pfio, fd, mode)
    FIO *pfio;
    FILE fd;
    COUNT mode;
```

FUNCTION

finit initializes the FIO control buffer at pfio for proper operation with the file specified by fd, in the mode specified by mode. mode should have one of the values BWRITE, READ, or WRITE. pfio must be the address of an FIO buffer declared by the user, or the address of the buffer stdin or stdout, both of which are provided by this library.

If (mode == BWRITE) the control buffer is set up for buffered writes, to be drained only when the buffer is full or the program exits. If (mode == READ) the control buffer is set up for reading. Otherwise (mode == WRITE) of necessity and the control buffer is set up for writing; writes will be buffered as for BWRITE only if lseek calls are acceptable with the specified fd, an indication that the output is a file and not an interactive device or a pipeline. Unbuffered output is drained whenever a segment of output ends with a newline character, or on program termination.

RETURNS

finit returns pfio. If (pfio == NULL), the fioerr condition is raised.

EXAMPLE

To adapt stdout for most effective buffering strategy:

```
IMPORT FIO stdout;
    ...
finit(&stdout, STDOUT, WRITE);
```

SEE ALSO

fclose, fcreate, fioerr, fopen

BUGS

No check is made for (mode == UPDATE), which may or may not work satisfactorily.

NAME

fioerr - NULL FIO pointer condition

SYNOPSIS

```
TEXT *fioerr
```

FUNCTION

fioerr is the condition raised when a NULL FIO pointer is passed to a library routine. If no handler is provided, the message written to STDERR is "NULL FIO pointer".

NAME

fopen - open a file and initialize a control buffer

SYNOPSIS

```
FIO *fopen(pfio, fname, mode)
    FIO *pfio;
    TEXT *fname;
    COUNT mode;
```

FUNCTION

fopen opens a file with name fname and specified mode, and if successful initializes the control buffer at pfio for proper operation with the file. mode should have one of the values BWRITE, READ, or WRITE. pfio must be the address of an FIO buffer declared by the user, or the address of the buffer stdin or stdout, both of which are provided by this library.

If (mode == BWRITE) the control buffer is set up for buffered writes, to be drained only when the buffer is full or the program exits. If (mode == READ) the control buffer is set up for reading. Otherwise (mode == WRITE) of necessity and the control buffer is set up for writing; writes will be buffered as for BWRITE only if lseek calls are acceptable with the specified fd, an indication that the output is a file and not an interactive device or a pipeline. Unbuffered output is drained whenever a segment of output ends with a newline character, or on program termination.

RETURNS

fopen returns pfio, if successful, else NULL. If (pfio == NULL), the fioerr condition is raised.

EXAMPLE

```
FIO inf;
    ...
if (!fopen(&inf, file, READ))
    errfmt("can't open %p\n", file);
```

SEE ALSO

fclose, fcreate, finit, fioerr, open(III)

BUGS

fopen always opens fname as a text file; on systems where text files and binary files are distinguished, open followed by finit must be used to associate a binary file with an FIO buffer.

NAME

fread - read until full count

SYNOPSIS

```
COUNT fread(fd, buf, size)
    FILE fd;
    TEXT *buf;
    BYTES size;
```

FUNCTION

fread reads up to size characters from the file specified by fd into the buffer starting at buf. It does so by making repeated calls to read until an end of file is encountered or until size characters have been read. Thus, fread should be used whenever an entire record must be read at once, since read reserves the right to return a short count at all times.

If not all characters are read, fread raises the readerr condition.

RETURNS

Unless end of file is encountered, fread always returns size; otherwise the value returned is between 0 and size, inclusive.

EXAMPLE

To copy a file in integral records:

```
while (fread(STDIN, buf, RECSIZE) == RECSIZE)
    write(STDOUT, buf, RECSIZE);
```

SEE ALSO

readerr(II), read(III)

NAME

free - free space on the heap

SYNOPSIS

```
TEXT *free(pcell, link)
    TEXT *pcell;
    TEXT *link;
```

FUNCTION

free returns an allocated cell to the heap for subsequent reuse, then returns link to the caller. The cell pointer pcell must have been obtained by an earlier alloc call; otherwise the heap will become corrupted. free does its best to check for invalid values of pcell. If an invalid pcell is noticed, the _memerr condition is raised, with the default error message changed to "bad free call".

A NULL pcell is explicitly allowed, however, and is ignored.

RETURNS

If free returns, its value is guaranteed to be link, which is otherwise unused by free.

EXAMPLE

To pop a stack item:

```
struct cell {
    struct cell *prev;
    ... rest of cell ...
    } *top;

top = free(top, top->prev); /* pops a cell */
```

SEE ALSO

_memerr, alloc, frelst, nalloc, sbreak(III)

BUGS

The size of the allocated cell is stored in the integer location right before the usable part of the cell; hence it is easily clobbered. No effort is made to lower the system break when storage is freed, so it is quite possible that earlier activity on the heap may cause later activity on the stack to come to grief, at least on some systems.

NAME

frelst - free a list of allocated cells

SYNOPSIS

```
struct list *frelst(plist, pstop)
    struct list {struct list *next; ...} *plist, *pstop;
```

FUNCTION

frelst walks a linked list that has been built with calls to alloc, freeing each cell on the list. Any types of cells can occur on the list, in any combination, so long as the first entry in each structure is a pointer used to link to the next cell. A NULL next pointer or one equal to pstop terminates the list.

RETURNS

frelst returns the pointer that terminates the list, either NULL or pstop.

EXAMPLE

```
struct list {
    struct list *next;
    ...} *list;

list = frelst(list, NULL);
```

SEE ALSO

alloc, free

BUGS

If a list is freed that was not made from calls on alloc, all hell can break loose.

NAME

fwrite - write and check

SYNOPSIS

```
VOID fwrite(fd, buf, size)
    FILE fd;
    TEXT *buf;
    BYTES size;
```

FUNCTION

fwrite writes size characters from the buffer starting at buf to the file specified by fd.

If not all characters are written, fwrite raises the writerr condition.

RETURNS

Nothing.

EXAMPLE

```
fwrite(STDOUT, buf, RECSIZE);
```

SEE ALSO

writerr(II), write(III)

NAME

getbfiles - collect files from command line

SYNOPSIS

```
FILE getbfiles(pac, pav, dfd, efd, rsize)
    BYTES *pac, rsize;
    TEXT ***pav;
    FILE dfd, efd;
```

FUNCTION

getbfiles examines the file arguments passed to a command and opens files as needed for reading. The arguments to examine are specified by the count pointed at by pac and by the array of text pointers pointed at by pav; it is assumed that the command name and any flags have been skipped, for instance by calling getflags. If there are no arguments left on the first call to getbfiles (*pac == 0), the default file descriptor dfd is returned and all subsequent calls will fail. Otherwise each call to getbfiles will inspect the next argument in sequence.

If a filename matches the string "-", dfd is returned. Otherwise an attempt is made to open the file for reading with the record size specified by rsize. If the file is to contain arbitrary binary data, as opposed to printable ASCII text, rsize should be non-zero. On success the file descriptor of the opened file is returned. If the open fails, efd is returned instead. After the last filename is processed, all calls to getbfiles will fail. It is up to the calling program to close any files opened by getbfiles.

RETURNS

getbfiles returns either a file descriptor obtained as described above, or the failure code -1; the first call to getbfiles will never return failure. *pac and *pav are updated on each call to reflect the number of arguments left to encode. To signal end of arguments, *pac is set to -1. If the returned file descriptor is not dfd, the name of the file under consideration (successfully opened or not) is at (*pav)[-1].

EXAMPLE

To walk a list of binary files:

```
BYTES ac;
TEXT **av;
    ...
while (0 <= (fd = getbfiles(&ac, &av, STDIN, STDERR, 1)))
    if (fd == STDERR)
        errfmt("can't read %p\n", av[-1]);
    else
        {
        process(fd);
        close(fd);
        }
```

SEE ALSO

getfiles, getflags, open(III)

NAME

getc - get a character from input buffer

SYNOPSIS

```
METACH getc(pfio)
    FIO *pfio;
```

FUNCTION

getc obtains the next input character, if any, from the file controlled by the FIO buffer at pfio; if end of file has been encountered a code is returned that is distinguishable from any character. pfio is normally the address of an FIO buffer declared by the user.

RETURNS

getc returns the character as zero (for '\0') or a small positive integer; end of file is signalled by the code EOF (-1). If any reads fail, the readerr condition is raised; if (pfio == NULL), the fioerr condition is raised.

EXAMPLE

To copy a textfile, character by character:

```
FIO inf, outf;
    ...
fopen(&inf, "infile", READ);
fcreate(&outf, "outfile", WRITE);
while (putc(&outf, getc(&inf)) != EOF)
    ;
```

SEE ALSO

fioerr, getch, gtc, putc, putch, readerr

NAME

getch - get a character from input buffer stdin

SYNOPSIS

METACH getch()

FUNCTION

getch obtains the next input character, if any, from the file controlled by the FIO buffer stdin; if end of file has been encountered a code is returned that is distinguishable from any character.

RETURNS

getch returns the character as zero (for NUL) or a small positive integer; end of file is signalled by the code EOF (-1). If any reads fail, the readerr condition is raised.

EXAMPLE

To copy stdin to stdout, character by character:

```
while (putch(getch()) != EOF)
    ;
```

Note that this is exactly equivalent to:

```
IMPORT FIO stdin, stdout;
...
while (putc(&stdout, getc(&stdin)) != EOF)
    ;
```

SEE ALSO

getc, putc, putch, readerr, stdin

NAME

getf - read formatted input

SYNOPSIS

```
COUNT getf(pfio, fmt, arg1, arg2, ...)
    FIO *pfio;
    TEXT *fmt;
    ...
```

FUNCTION

getf reads input text from the file controlled by the buffer at pfio, and parses it according to the control format string starting at fmt, in order to assign converted values to a series of variables, each pointed at by one of the arguments arg1, ... The format string consists of newlines and literal text to be matched, interspersed with <field-specifier>s that determine how the input text is to be read and how it is to be converted before assignment. Input is consumed on a line-by-line basis. The number of lines consumed in any one call is typically equal to the number of newlines encountered in the format string, plus one if any character follows the last newline encountered in the format. An exception to this may occur if "% " appears in the format string; this sequence matches arbitrary whitespace, even extending across multiple lines.

For example:

```
getf(&stdin, "%i\n%i\n%i", &arg1, &arg2, &arg3);
```

obtains values for the three integers arg1, arg2, and arg3 from three successive lines of stdin, while:

```
getf(&stdin, "%i%i%i", &arg1, &arg2, &arg3);
```

obtains values for the three integers arg1, arg2, and arg3 from three whitespace separated fields on a single line of stdin.

Matching of literal text occurs on a character by character basis. If the character in the format string does not match the next character to be consumed on the input line, the scan is terminated. A newline character in the format string matches any characters remaining in the current input line, up to and including the terminating newline, if any. Since a newline is consumed only by a literal match, by "% ", or (implicitly) by the end of the format string, an embedded '\n' is the most controlled way of reading multiple lines with one call to getf.

A <field-specifier> takes the form:

```
%[+z|-z][#]<field-code>
```

That is, a <field-specifier> consists of a literal '%', followed by an optional "+z" or "-z", where z can be any character, followed by an optional field width #, and is terminated by a <field-code>. A "+z", if present, calls for the stripping of any left fill with the fill character z, while "-z" calls for the stripping of any right fill with z. A #, if present, specifies the total width in characters of the field to be input, and is

either a decimal integer, or the letter 'n'. If an 'n' is given, then the value of the next argument from the argument list is taken to specify the field width.

To read a nine-character field left-filled with '*', and interpret it as a floating point number:

```
getf(&stdin, "%+*9f", &arg1);
```

or:

```
getf(&stdin, "%+*nf", 9, &arg1);
```

The number of characters to consume during a field conversion is given by the width specifier, if present. If there are fewer than that many characters before the next newline, the rest of the line is consumed. If no width is specified, leading whitespace is skipped and the following group of non-white characters is taken to be the field; at least one non-white character must be present. The characters actually converted are the contents of the field less any fill characters. If no fill character is given, getf presumes the field is left-filled with spaces.

Text input of

```
$   100.53
```

can be read as two integers with:

```
getf(&stdin, "$%6i.%2i", &dollars, &cents);
```

or as a single double with either:

```
getf(&stdin, "%+$10d", &cash);
```

or:

```
getf(&stdin, "$%d", &cash);
```

A <field-code> is composed of a <modifier>, a <specifier> or both. The <specifier> defines how the input field is to be converted, and is one of the following:

```
c = char integer
s = short integer
i = integer
l = long integer
p = NUL-terminated string
b = buffer of specified length
d = double
f = float
x = padding only (no conversion)
```

A <modifier> causes the input to an integer variable to be interpreted as:

a = ASCII bytes, in decreasing order of numerical significance
h = hexadecimal (with or without a leading "0x")
o = octal (with or without a leading '0')
u = unsigned decimal

If no <specifier> is given, it is presumed to be 'i', and a <modifier> given from the above series will be taken to apply to the implied integer field. If a <specifier> of 'c', 's', 'i' or 'l' is given with no <modifier>, the input is interpreted as signed decimal.

In addition, an optional precision modifier, ".#", limits the number of characters that may be input with a <specifier> of 'p' or 'b', and is permitted but ignored with 'd' and 'f', for compatibility with putf. Like the field width specifier, the precision modifier # may be either an explicit integer, or an 'n', to make use of the next argument value in sequence.

Hence a <field-code> usually consists of one of the following combinations of <specifier> and <modifier>:

```
[a|h|o|u]{c|s|i|l}      /* integer input */
[.#]{b|p|d|f}           /* precision ignored for f and d */
{a|h|o|u}               /* default specifier is i */
{x}                     /* just skip field */
```

Any other character in the place of a <field-code> is taken as a single literal character to be matched in the input line. Thus a '%' may be scanned with the specifier "%%" and a '\n' may be scanned, without skipping characters in the input line, by using the specifier "%\n". Hence, while "% " and "\n" have special meaning, "%\n" and " " each match only one character.

Each <field-specifier> given in the format string requires the argument list following to contain in identical sequence a pointer to a datum of the appropriate type; the pointer argument is used to assign a correctly converted field.

The following would read an int in hex, a char-sized value as ASCII, and a short as signed decimal, all of them optionally separated by whitespace:

```
getf(&stdin, "%8h%ac%s", &addr, &code, &offset);
```

Any integer field may contain leading whitespace, even after the stripping of fill characters, as well as an optional [+|-] sign, and an optional trailing [l|L] (which is C notation for a long constant). No unexpected conversion character may occur or the scan is terminated before the corresponding argument is assigned.

The 'a' modifier treats the input as a sequence of characters and converts it to a base 256 number whose digits are the characters; the argument gets assigned the value represented by the low-order bytes of that number.

Entire text strings may be assigned to arguments under the 'p' or 'b' field code. In the first case, the argument is a pointer to the start of

a string, and input characters are copied into that string with a terminating NUL; in the second case, the argument is also a pointer to text but characters are copied in without the terminating NUL, and the number of characters copied is assigned using the argument following the pointer as a pointer to integer. In either case, the number of characters actually copied will be no more than the precision modifier, if it is present and nonzero.

For example, exactly 1 characters of an 80-character input line could be assigned to str with:

```
getf(&stdin, "%80.np", 1, str);
```

Floating point numbers may be read in using 'd' for double variables and 'f' for float. In either case, the input may be in either fixed point or scientific notation (see btod). Leading whitespace will be skipped, even after the stripping of fill characters. The precision modifier is ignored.

The 'x' field code consumes no arguments; it is a convenient way to skip over text.

RETURNS

getf returns the number of arguments successfully assigned, or EOF if end of file is encountered on input before any argument has been converted. If any reads fail, the readerr condition is raised; if (pfio == NULL), the fioerr condition is raised.

EXAMPLE

Given the code:

```
FIO input;
TEXT buf1[BUFSIZE], buf2[10];
BYTES nargs, x, y, z;
    ...
nargs = getf(&input, "%b%-*i%.6p%4i", &buf1, &x, &y, &buf2, &z);
if (nargs != 5)
    putstr(STDERR, "bad input format\n", NULL);
```

The input line:

```
LINE    17**    IDENTIFIER 263
```

would assign:

```
"LINE" to buf1, with no trailing NUL
4 to x
17 to y
"IDENTI" to buf2, with trailing NUL
263 to z
```

SEE ALSO

btod, encode, fioerr, getfmt, readerr

NAME

getfiles - collect text files from command line

SYNOPSIS

```
FILE getfiles(pac, pav, dfd, efd)
    BYTES *pac;
    TEXT ***pav;
    FILE dfd, efd;
```

FUNCTION

getfiles examines the file arguments passed to a command and opens text files as needed for reading. The arguments to examine are specified by the count pointed at by pac and by the array of text pointers pointed at by pav; it is assumed that the command name and any flags have been skipped, for instance by calling getflags. If there are no arguments left on the first call to getfiles (*pac == 0), the default file descriptor dfd is returned and all subsequent calls will fail. Otherwise each call to getfiles will inspect the next argument in sequence.

If a filename matches the string "-", dfd is returned. Otherwise an attempt is made to open the file for reading as a text file; on success the file descriptor of the opened file is returned. If the open fails, efd is returned instead. After the last filename is processed, all calls to getfiles will fail. It is up to the calling program to close any files opened by getfiles.

RETURNS

getfiles returns either a file descriptor obtained as described above, or the failure code -1; the first call to getfiles will never return failure. *pac and *pav are updated on each call to reflect the number of arguments left to encode. To signal end of arguments, *pac is set to -1. If the returned file descriptor is not dfd, the name of the file under consideration (successfully opened or not) is at (*pav)[-1].

EXAMPLE

To walk a list of files:

```
BYTES ac;
TEXT **av;

while (0 <= (fd = getfiles(&ac, &av, STDIN, STDERR)))
    if (fd == STDERR)
        errfmt("can't read %p\n", av[-1]);
    else
        {
        process(fd);
        close(fd);
        }
```

SEE ALSO

getbfiles, getflags, open(III)

NAME

getflags - collect flags from command line

SYNOPSIS

```
TEXT *getflags(pac, pav, fmt, arg1, arg2, ...)
    BYTES *pac;
    TEXT ***pav;
    TEXT *fmt;
    ...
```

FUNCTION

getflags encodes the flag arguments passed to a command and sets the flags, counts, character variables and string names specified by a format string. The arguments to encode are specified by the count pointed at by pac and by the array of text pointers pointed at by pav; it is assumed that the first argument is a command name, to be skipped. Each succeeding argument is taken as a set of one or more flags if a) it begins with '-' or '+' and b) it is not the string "-" or "--". If an argument is taken as a set of flags, a leading '-' is skipped before it is processed.

fmt is a concatenation of descriptors that determine how each of the succeeding arguments arg1, ... is to be interpreted. A descriptor is a sequence of match characters, terminated by a ',', a '>', or by the '\0' or ':' that terminates the format string. Format characters have the following effect:

'*' - always matches the rest of the current argument, if any left, or all of the succeeding argument, if present, or a null string otherwise. The argument may contain escape sequences recognizable by doesc, such as \n for newline, or \0 for NUL. The value of the match is a (non-NULL) pointer to the start of the matched string.

'?' - always matches the next argument character, if any, or a NUL character. An escape sequence recognizable by doesc may be used instead of a literal character. The value of the match is the matched character, taken as an integer constant.

'#' - tries to parse as an integer the remainder of the current argument, if any left, or all of the succeeding argument. The value of the match is the decimal value of the string, if it doesn't begin with a '0', or its hexadecimal value if it begins with '0x' or '0X', or its octal value otherwise. In all cases, a leading '+' or '-' is allowed. An error occurs if no argument is found, or if it cannot be completely scanned as an integer with the selected base.

'##' - same as single # except that target is assumed to be a long instead of an int.

',' - delivers a successful match value to the corresponding argument (in sequence) pointed at by arg1, ... If no match, the command arguments are rescanned, from the last successful match, using the descriptor following.

'>' - behaves just like ',', except the corresponding argument pointer is taken as a pointer to a structure of the form

```
struct {
    BYTES ntop;
    TEXT *val[MAX];
    } args {MAX};
```

for all argument types except long (##). For long arguments, the form of the structure used is

```
struct {
    BYTES ntop;
    LONG val[MAX];
    } args {MAX};
```

If (0 < ntop) ntop is decremented and the value is delivered to val[ntop]; otherwise an error occurs.

'\0' - behaves just like ',', except that if there is no successful match an error occurs.

':' - if a colon is encountered in the format string before a flag is successfully matched, then the NUL-terminated string following the colon is written to STDERR, preceded by "usage: <pname> " and followed by a newline, where <pname> is the name by which the current program was invoked. getflags then terminates, reporting failure. Any occurrence of an 'F' in the diagnostic string is replaced with a slightly expanded representation of the flag format string preceding the colon. For example, the format string "a*>+b,c?,z:F <files>" would produce the error message:

```
usage: pname -[a* +b c? z] <files>
```

Any other character causes a successful match only if the next command line character is identical to it. The value of the match is a boolean YES.

The rules by which getflags parses flag arguments impose two significant constraints on how flags are ordered within the format string. Any flag whose name is a prefix of the name of another flag must appear in the format string after the longer flag. This also implies that unnamed flags, such as "-#" or "-##" or "-*", must be given last.

RETURNS

getflags returns a pointer to the remaining command argument string, if an error occurs and no colon is found in the format string; otherwise an error causes diagnostic output and an error exit from the program. If all flag arguments are successfully scanned, getflags returns NULL. The values pointed at by pac and pav are updated to reflect the number of arguments consumed; "--" is consumed as a flags terminator, while "-" is taken as a potential special file name and is not consumed. One or more values should be delivered to locations pointed at by the arg1, ...

Note that all locations pointed at are assumed to be ints or pointers, except that a "##" descriptor expects a pointer to a long, and a '>' expects a pointer to a structure as described above.

EXAMPLE

To accept the line:

```
cmd +3 -3 -f filename -mx -b0x10000 <files> ...
```

one might write:

```
BOOL mxflag {NO};
BYTES mcnt {0};
BYTES from {0};
BYTES to {0};
LONG bias {0};
TEXT *fname "default";

COUNT main(ac, av)
BYTES ac;
TEXT **av;
{
getflags(&ac, &av, "b##,f*,mx,m#,+#,#:F <files>",
     &bias, &fname, &mxflag, &mcnt, &from, &to))
```

SEE ALSO

doesc, getbfiles, getfiles, usage

NAME

getfmt - read formatted input from stdin

SYNOPSIS

```
COUNT getfmt(fmt, arg1, arg2, ...)
    TEXT *fmt;
```

FUNCTION

getfmt reads formatted input from the file controlled by the FIO buffer stdin, in exactly the same way as getf.

RETURNS

getfmt returns the number of arguments successfully converted, or EOF (-1) if end of file is encountered before any are converted. If any reads fail, the readerr condition is raised.

EXAMPLE

To sum a series of longs read from stdin:

```
LONG lnum, lsum;
    ...
for (lsum = 0; 0 < getfmt("%l", &lnum); )
    lsum =+ lnum;
```

SEE ALSO

encode, getf, readerr, stdin

NAME

getin - build ac and av list from STDIN

SYNOPSIS

```
BOOL getin(ac, av)
    BYTES *ac;
    TEXT ***av;
```

FUNCTION

getin reads lines of text from STDIN, sets ac to the number of lines read, and builds an av list, one item for each line of text. Empty lines are ignored. Newlines in the text are replaced by NULLS. ac and av are changed in the calling program. The message "argument too long" is written to STDERR if the filename > NAMSIZE characters. NAMSIZE is currently 64.

RETURNS

getin returns success if all went well, and failure if any filenames were too long.

EXAMPLE

To set up ac and av from STDIN:

```
success = getin(&ac, &av);
```

SEE ALSO

stdin

BUGS

Filenames in the error message are truncated to NAMSIZE characters.

NAME

getl - get a text line from input buffer

SYNOPSIS

```
BYTES getl(pfio, s, n)
    FIO *pfio;
    TEXT *s;
    BYTES n;
```

FUNCTION

getl copies characters, from the file controlled by the FIO buffer at pfio, to the n character buffer starting at s. Characters are copied until a) a newline is copied, b) end of file is reached, or c) n characters have been copied. pfio is normally the address of an FIO buffer declared by the user.

RETURNS

getl returns a count of the number of characters copied, which will be between 1 and n unless end of file has been encountered, from which time on all getl calls will return zero. If any reads fail, the readerr condition is raised; if (pfio == NULL), the fioerr condition is raised.

EXAMPLE

To copy a text file, line by line:

```
FIO inf, outf;
    ...
fopen(&inf, "infile", READ);
fcreate(&outf, "outfile", WRITE);
while (putl(&outf, getl(&inf, buf, BUFSIZE)));
    ;
```

SEE ALSO

fioerr, getlin, putl, putlin, readerr

NAME

getlin - get a text line from input buffer stdin

SYNOPSIS

```
BYTES getlin(s, n)
    TEXT *s;
    BYTES n;
```

FUNCTION

getlin copies characters, from the file controlled by the FIO buffer stdin, to the n character buffer starting at s. Characters are copied until a) a newline is copied, b) end of file is reached, or c) n characters have been copied.

RETURNS

getlin returns a count of the number of characters copied, which will be between 1 and n unless end of file has been encountered, from which time on all getlin calls will return zero. If any reads fail, the readerr condition is raised.

EXAMPLE

To copy stdin to stdout, line by line:

```
while (putlin(buf, getlin(buf, BUFSIZE)))
    ;
```

Note that this is exactly equivalent to:

```
IMPORT FIO stdin, stdout;
    ...
while(putl(&stdout, getl(&stdin, buf, BUFSIZE)))
    ;
```

SEE ALSO

getl, putl, putlin, readerr, stdin

NAME

gtc - get a character from input buffer

SYNOPSIS

gtc(pfio)

FUNCTION

gtc obtains the next input character, if any, from the file controlled by the FIO buffer at pfio. pfio is normally the address of an FIO buffer declared by the user, or the address of the buffer stdin, which is provided by this library. If there is a character already in the buffer, no function call is attempted; if the FIO buffer is empty, getc() is called to fill the buffer. If end of file has been encountered, a code is returned that is distinguishable from any character.

gtc significantly reduces per character I/O overhead.

RETURNS

gtc returns the character as zero (for '\0') or a small positive integer; end of file is signalled by the code EOF (-1).

EXAMPLE

To copy stdin to stdout, character by character:

```
IMPORT FIO stdin, stdout;
    ...
while (ptc(&stdout, gtc(&stdin)) != EOF)
    ;
```

SEE ALSO

getc, getch, ptc, putc, putch, stdin

BUGS

Because it is a macro, gtc cannot be called from non-C programs, nor can its address be taken. An argument with side effects may be evaluated other than just once.

NAME

inbuf - find occurrence in buffer of character in set

SYNOPSIS

```
BYTES inbuf(p, n, s)
    TEXT *p, *s;
    BYTES n;
```

FUNCTION

inbuf scans the n-character buffer starting at p for the first instance of a character in the NUL terminated set s. If the NUL character is to be part of the set, it must be the first character in the set.

RETURNS

inbuf returns the index of the first character in p that is also in the set s, or n if no character in the buffer is in the set.

EXAMPLE

To blank out imbedded NUL characters:

```
while ((i = inbuf(buf, n, "\0")) < n)
    buf[i] = ' ';
```

SEE ALSO

instr, notbuf, notstr, scnbuf, scnstr, subbuf, substr

NAME

instr - find occurrence in string of character in set

SYNOPSIS

```
BYTES instr(p, s)
    TEXT *p, *s;
```

FUNCTION

instr scans the NUL terminated string starting at p for the first occurrence of a character in the NUL terminated set s.

RETURNS

instr returns the index of the first character in p that is also contained in the set s, or the index of the terminating NUL if none.

EXAMPLE

To replace unprintable characters (as for a 64-character terminal):

```
while (string[i = instr(string, "`{|}~")])
    string[i] = '@';
```

SEE ALSO

inbuf, notbuf, notstr, scnbuf, scnstr, subbuf, substr

NAME

isalpha - test for alphabetic character

SYNOPSIS

BOOL isalpha(c)

FUNCTION

isalpha tests whether its argument is an alphabetic character, either lower or upper case. Since isalpha is implemented as a C preprocessor macro, its argument can be any numerical type.

RETURNS

isalpha is a boolean rvalue.

EXAMPLE

To find the end points of an alpha string:

```
if (isalpha(*first))
    for (last = first; isalpha(*last); ++last)
        ;
```

SEE ALSO

isdigit, islower, isupper, iswhite, tolower, toupper

BUGS

Because it is a macro, isalpha cannot be called from non-C programs, nor can its address be taken. Arguments with side effects may be evaluated other than once.

NAME

isdigit - test for digit

SYNOPSIS

BOOL isdigit(c)

FUNCTION

isdigit tests whether its argument is a decimal digit, i.e., between '0' and '9' inclusive. Since isdigit is implemented as a C preprocessor macro, its argument can be any numerical type.

RETURNS

isdigit is a boolean rvalue.

EXAMPLE

To convert a digit string to a number:

```
for (sum = 0; isdigit(*s); ++s)
    sum = sum * 10 + *s - '0';
```

SEE ALSO

isalpha, islower, isupper, iswhite, tolower, toupper

BUGS

Because it is a macro, isdigit cannot be called from non-C programs, nor can its address be taken. Arguments with side effects may be evaluated other than once.

NAME

islower - test for lowercase character

SYNOPSIS

```
BOOL islower(c)
```

FUNCTION

islower tests whether its argument is a lowercase character. Since islower is implemented as a C preprocessor macro, its argument can be any numerical type.

RETURNS

islower is a boolean rvalue.

EXAMPLE

To convert to uppercase:

```
if (islower(c))
    c =+ 'A' - 'a';     /* but see toupper () */
```

SEE ALSO

isalpha, isdigit, isupper, iswhite, tolower, toupper

BUGS

Because it is a macro, islower cannot be called from non-C programs, nor can its address be taken. Arguments with side effects may be evaluated other than once.

NAME

isupper - test for uppercase character

SYNOPSIS

BOOL isupper(c)

FUNCTION

isupper tests whether its argument is an uppercase character. Since isupper is implemented as a C preprocessor macro, its argument can be any numerical type.

RETURNS

isupper is a boolean rvalue.

EXAMPLE

To convert to lowercase:

```
if (isupper(c))
    c =+ 'a' - 'A';      /* but see tolower() */
```

SEE ALSO

isalpha, isdigit, islower, iswhite, tolower, toupper

BUGS

Because it is a macro, isupper cannot be called from non-C programs, nor can its address be taken. Arguments with side effects may be evaluated other than once.

NAME

iswhite - test for whitespace character

SYNOPSIS

BOOL iswhite(c)

FUNCTION

iswhite tests whether its argument is a non-printing character code, i.e., whether its ASCII value is at or below that of ' ' (040) or at or above that of DEL (0177). Note that both NUL '\0' and newline '\n' qualify as whitespace. Since iswhite is implemented as a C preprocessor macro, its argument can be any numerical type.

RETURNS

iswhite is a boolean rvalue.

EXAMPLE

To skip whitespace:

```
while (iswhite(*s))
    ++s;
```

SEE ALSO

isalpha, isdigit, islower, isupper, tolower, toupper

BUGS

Because it is a macro, iswhite cannot be called from non-C programs, nor can its address be taken. Arguments with side effects may be evaluated other than once.

NAME

itob - convert integer to text in buffer

SYNOPSIS

```
BYTES itob(s, i, base)
    TEXT *s;
    ARGINT i;
    COUNT base;
```

FUNCTION

itob converts the integer i to a text representation in the buffer starting at s. The number is represented in the base specified, using lowercase letters beginning with 'a' to specify digits from 10 on. If (0 < base) the number i is taken as unsigned; otherwise if (base < 0) negative numbers have a leading minus sign and are converted to -base; if (base == 0) it is taken as -10. Only magnitudes of base between 2 and 36 are generally meaningful, but no check is made for reasonableness.

RETURNS

The value returned is the number of characters used to represent the integer, which in hexadecimal can vary from four to eight digits, plus sign, depending upon the target machine.

EXAMPLE

To output i in decimal:

```
write(STDOUT, buf, itob(buf, i, 10));
```

SEE ALSO

btoi, btol, ltob, stob

BUGS

The length of the buffer is not specifiable. If (|base| == 1) the program can bomb; if (36 < |base|) funny characters can be inserted in the buffer.

NAME

itols - convert integer to leading low-byte string

SYNOPSIS

```
TEXT *itols(s, val)
    TEXT *s;
    COUNT val;
```

FUNCTION

itols writes the integer val into the two-byte string at s, with the least significant byte at s[0] and the next least at s[1]. No stronger storage boundary than that required for char is demanded of s.

A number of de facto standard file formats have arisen on machines that represent integers internally in this fashion; itols provides a machine-independent way of writing such files.

RETURNS

itols writes the two bytes at s and returns s as the value of the function.

EXAMPLE

To write a library header:

```
struct
    {
    TEXT name[14];
    UCOUNT size;
    } *p;

itols(&p->size, p->size);
fwrite(fd, p, sizeof (*p));
```

SEE ALSO

lstoi, lstol, lstou, ltols

NAME

leave - leave a control region

SYNOPSIS

```
VOID leave(val)
    BYTES val;
```

FUNCTION

leave causes an exit from the control region established by the most recent enter call. Execution resumes as if enter had just performed a return with value val. Any number of functions may be terminated early by a leave call, so long as all are dynamic descendants of at least one enter call. The control region is disestablished by the call to leave.

RETURNS

leave will never return to its caller; instead val is used as the return value of the most recent call to enter. If no instance of enter is currently active, leave writes an error message to STDERR and takes an error exit.

EXAMPLE

To restart a function after each error message:

```
while (s = enter(&func, file))
    putstr(STDERR, s, "\n", NULL);
```

so that one can write in, say, one of func's dynamic descendants:

```
if (counterr)
    leave("missing parameter");
```

SEE ALSO

_raise(IV), _when(IV), enter

NAME

lenstr - find length of a string

SYNOPSIS

```
BYTES lenstr(s)
    TEXT *s;
```

FUNCTION

lenstr scans the text string starting at s to determine the number of characters before the terminating NUL.

RETURNS

The value returned is the number of characters in the string.

EXAMPLE

To output a string:

```
write(STDOUT, s, lenstr(s));
```

NAME

ln - natural logarithm

SYNOPSIS

```
DOUBLE ln(x)
    DOUBLE x;
```

FUNCTION

ln computes the natural log of x to full double precision. It works by expressing x as a fraction in the interval [1/2, 1), times an integer power of two. The logarithm of the fraction is approximated by a sixth order telescoped series approximation.

RETURNS

ln returns the nearest internal representation to ln x, expressed as a double floating value. If x is negative or zero, a domain error condition is raised.

EXAMPLE

```
IMPORT DOUBLE ln(), sqrt();
    ...
arcsinh = ln(x + sqrt(x * x + 1));
```

SEE ALSO

_domain(IV), exp

NAME

lower - convert characters in buffer to lowercase

SYNOPSIS

```
BYTES lower(s, n)
    TEXT *s;
    BYTES n;
```

FUNCTION

lower converts the n characters in buffer starting at s to their lowercase equivalent if possible.

RETURNS

lower returns n.

EXAMPLE

```
buf[lower(buf, size)] = '\0';
```

SEE ALSO

tolower

NAME

lstoi - convert leading low-byte string to integer

SYNOPSIS

```
COUNT lstoi(s)
    TEXT *s;
```

FUNCTION

lstoi converts the two-byte string at s into an integer, on the assumption that the leading byte is the less significant part of the integer. No stronger storage boundary than that required for char is demanded of s.

A number of de facto standard file formats have arisen on machines that represent integers internally in this fashion; lstoi provides a machine-independent way of reading such files.

RETURNS

lstoi returns the integer representation of the two-byte integer at s.

EXAMPLE

To read a library header:

```
struct
    {
    TEXT name[14];
    COUNT size;
    } *p;

fread(fd, p, sizeof (*p));
p->size = lstoi(&p->size);
```

SEE ALSO

itols, lstol, lstou, ltols

NAME

lstol - convert filesystem date to long

SYNOPSIS

```
LONG lstol(s)
    TEXT *s;
```

FUNCTION

lstol converts the four-byte string at s into a long, on the assumption that the bytes are ordered 2, 3, 0, 1, where 0 is the least significant byte. This bizarre order is used to represent dates in Idris filesystems, thanks to their PDP-11 origins. No stronger storage boundary than that required for char is demanded of s.

RETURNS

lstol returns the long representation of the four-byte integer at s.

EXAMPLE

```
time = lstol(&pi->n_actime);
```

SEE ALSO

itols, lstoi, lstou, ltols

NAME

lstou - convert leading low-byte string to unsigned short

SYNOPSIS

```
UCOUNT lstou(s)
    TEXT *s;
```

FUNCTION

lstou converts the two-byte string at s into an unsigned short, on the assumption that the leading byte is the less significant part of the number. No stronger storage boundary than that required for char is demanded of s.

A number of de facto standard file formats have arisen on machines that represent integers internally in this fashion; lstou provides a machine-independent way of reading such files.

Note that lstou's primary raison d'etre is to prevent sign extension of unsigned quantities on machines with 32 bit integers.

RETURNS

lstou returns the unsigned short representation of the two-byte integer at s.

EXAMPLE

To read a library header:

```
struct
    {
    TEXT name[14];
    UCOUNT size;
    } *p;

fread(fd, p, sizeof (*p));
p->size = lstou(&p->size);
```

SEE ALSO

itols, lstoi, lstol, ltols

NAME

ltob - convert long to text in buffer

SYNOPSIS

```
BYTES ltob(s, l, base)
    TEXT *s;
    LONG l;
    COUNT base;
```

FUNCTION

ltob converts the long l to a text representation in the buffer starting at s. The number is represented in the base specified, using lower case letters beginning with 'a' to specify digits from 10 on. If (0 < base) the number l is taken as unsigned; otherwise if (base < 0) negative numbers have a leading minus sign and are converted to -base; if (base == 0) it is taken as -10. Only values of base between 2 and 36 in magnitude are generally meaningful, but no check is made for reasonableness.

RETURNS

The value returned is the number of characters used to represent the long, which in hexadecimal can be up to eight digits plus sign.

EXAMPLE

To output l as an unsigned decimal number:

```
write(STDOUT, buf, ltob(buf, l, 10));
```

SEE ALSO

btoi, btol, itob, stob

BUGS

The length of the buffer is not specifiable. If (|base| == 1) the program can bomb; if (36 < |base|) funny characters can be inserted in the buffer.

NAME

ltols - convert long to filesystem date

SYNOPSIS

```
TEXT *ltols(plong, lo)
    TEXT *plong;
    LONG lo;
```

FUNCTION

ltols writes the four bytes of the long lo into the buffer, starting at plong, in the order 2, 3, 0, 1, where 0 is the least significant byte. This bizarre order is used to represent dates in Idris filesystems, thanks to their PDP-11 origins.

RETURNS

ltols writes the four bytes at plong and returns plong as its value.

EXAMPLE

```
ltols(&pi->n_actime, time);
```

SEE ALSO

itols, lstoi, lstou, lstol

NAME

mapchar - map single character to printable representation

SYNOPSIS

```
VOID mapchar(c, ptr)
    TEXT c, *ptr;
```

FUNCTION

mapchar writes a visible representation of the character c into a two-byte buffer pointed at by ptr. A printable character (including space through '~') is written as a space followed by the character. Other codes appear as:

CHARACTER	BECOMES
[0, 07]	\0 - \7
backspace	\b
tab	\t
newline	\n
vertical tab	\v
formfeed	\f
carriage return	\r
all other values	\?

RETURNS

Nothing. mapchar writes two characters at ptr[0] and ptr[1].

EXAMPLE

To output a visible representation of an arbitrary character, one might write:

```
TEXT c, str[2];

mapchar(c,str);
putfmt("%4b\n", str, 2);
```

SEE ALSO

doesc

NAME

match - match a regular expression

SYNOPSIS

```
BOOL match(buf, n, pat)
    TEXT *buf;
    BYTES n;
    TEXT *pat;
```

FUNCTION

match tests the n character buffer starting at buf for a match with the encoded pattern starting at pat. It is assumed that the pattern was built by the function pattern, whose manual page describes the notation for regular expressions accepted by these routines.

RETURNS

match returns YES if the pattern matches.

EXAMPLE

To test a line for the presence of three colons:

```
if (match(line, n, pattern(pbuf, '\0', ":*:*:*")))
    return (YES);
```

SEE ALSO

amatch, pattern

NAME
max - test for maximum

SYNOPSIS
max(a, b)

FUNCTION
max obtains the maximum of its two arguments a and b. Since max is implemented as a C preprocessor macro, its arguments can be any numerical type, and type coercion occurs automatically.

RETURNS
max is a numerical rvalue of the form ((a < b) ? b : a), suitably parenthesized.

EXAMPLE

```
hiwater = max(hiwater, level);
```

SEE ALSO
min

BUGS
Because it is a macro, max cannot be called from non-C programs, nor can its address be taken. Arguments with side effects may be evaluated other than just once.

NAME

min - test for minimum

SYNOPSIS

```
min(a, b)
```

FUNCTION

min obtains the minimum of its two arguments a and b. Since min is implemented as a C preprocessor macro, its arguments can be any numerical type, and type coercion occurs automatically.

RETURNS

min is a numerical rvalue of the form ((a < b) ? a : b), suitably parenthesized.

EXAMPLE

```
nmove = min(space, size);
```

SEE ALSO

max

BUGS

Because it is a macro, min cannot be called from non-C programs, nor can its address be taken. Arguments with side effects may be evaluated other than just once.

NAME

mkord - make an ordering function

SYNOPSIS

```
COUNT (*mkord(keyarray, lnordrule))()
    TEXT **keyarray, *lnordrule;
```

FUNCTION

mkord uses the encoded text strings pointed at by lnordrule and the elements of keyarray to produce a function, suitable for use with sort, that compares two text buffers for lexical order. The function produced can be declared (symbolically, at least) as:

```
COUNT ordfun(i, j, ppa)
    BYTES i, j;
    struct {
        UCOUNT len;
        TEXT buf[len];
        } ***ppa;
```

That is, ppa is a pointer to an array of pointers to structures, each of which consists of a two-byte buffer length len, followed by the text buffer proper. The function is expected to compare the text in the structure pointed at by (*ppa)[i] with that in the structure pointed at by (*ppa)[j], returning a negative number if the first is less than the second, zero if the two compare equal, and positive otherwise.

keyarray is a NULL terminated list of "keys", or ordering rules to be used by ordfun, listed in reverse order of application, i.e., keyarray[0] specifies a rule that is applied only if keyarray[1] is NULL or if it (and all higher rules) says that the two text buffers compare equal, on a given call to ordfun.

Each of the keys, as well as lnordrule, is a NUL terminated string that specifies a rule (as shown below) for ordering two text buffers. lnordrule is the key tried last by ordfun; it also specifies the default method of comparison for any keys in keyarray that don't explicitly state a method. Thus, if keyarray[0] is NULL, lnordrule alone specifies the ordering.

Strings in lnordrule and keyarray take the form:

```
[adln][b][r][t?][#.#-#.#]
```

where

a - compares character by character in ASCII collating sequence. A missing character compares lower than any ASCII code.

b - skips leading whitespace.

d - compares character by character in dictionary collating sequence, i.e., characters other than letters, digits, or spaces are omitted, and case distinctions among letters are ignored.

l - compares character by character in ASCII collating sequence, except that case distinctions among letters are ignored.

n - compares by arithmetic value, treating each buffer as a numeric string consisting of optional whitespace, optional minus sign, and digits with an optional decimal point.

r - reverses the sense of comparisons.

t? - uses ? as the tab character for determining offsets (described below).

#.#-#.# - describes offsets from the start of each text buffer for the beginning (first character used) and, after the minus '-', for the end (first character not used) of the text to be considered by the rule. The number before each dot '.' is the number of tab characters to skip, and the number after each dot is the number of characters to skip thereafter. Thus, in the string "abcd=efgh", with '=' as the tab character, the offset "1.2" would point to 'g', and "0.0 would point to 'a'. A missing number # is taken as zero; a missing final pair "-#.#" points just past the last of the text in each of the buffers to be compared. If the first offset is past the second offset, the buffer is considered empty.

If no tab character is specified, each contiguous string of whitespace will be taken as a tab. Thus, in the string " ABC DEF GHI", the offset "3" would point to 'G'.

Only one of 'a', 'd', 'l', or 'n' may be present in a rule, and no more than ten ordering rules can be specified by keyarray.

RETURNS

If all keys make sense, mkord returns a pointer to an internal ordering function as described above; otherwise it returns NULL. Various internal tables are rewritten, on each call to mkord, so only one ordering function may be defined at a time.

EXAMPLE

```
#define MAXKEY  10
INTERN struct {
BYTES n;
TEXT *key[MAXKEY+1];
} kstack {MAXKEY};  /* kstack.key[MAXKEY] is always NULL */

getflags(&ac, &av, "+*>:+[#.#-#.# a b d l n r t?]", &kstack);
order = mkord(&kstack.key[kstack.n], "a");
...
sort(nlines, order, &swapfn, linptrs);
```

SEE ALSO

sort

NAME

nalloc - allocate space on the heap

SYNOPSIS

```
TEXT *nalloc(nbytes, link)
    BYTES nbytes;
    TEXT *link;
```

FUNCTION

nalloc allocates space on the heap for an item of size nbytes, then writes link in the zeroeth integer location. The space allocated is guaranteed to be at least nbytes long, starting from the pointer returned, which pointer is guaranteed to be on a proper storage boundary for anything. The heap is grown as necessary.

RETURNS

nalloc returns a pointer to the allocated cell if successful; otherwise, it returns a NULL pointer.

EXAMPLE

To build a stack:

```
struct cell {
    struct cell *prev;
    ... rest of cell ...
    } *top;

    top = nalloc(sizeof (*top), top);   /* pushes a cell */
```

SEE ALSO

alloc, free, sbreak(III)

BUGS

The size of the allocated cell is stored in the integer location right before the usable part of the cell; hence it is easily clobbered. This number is related to the actual cell size in a most system dependent fashion and should not be trusted.

NAME

notbuf - find occurrence in buffer of character not in set

SYNOPSIS

```
BYTES notbuf(p, n, s)
    TEXT *p, *s;
    BYTES n;
```

FUNCTION

notbuf scans the n-character buffer starting at p for the first instance of a character not in the NUL terminated set starting at s. If the NUL character is to be part of the set, it must be the first character in the set.

RETURNS

notbuf returns the index of the first character in p not contained in the set s, or the value n if all buffer characters are in the set.

EXAMPLE

To check that an input string contains only digits:

```
if (notbuf(buf, n, "0123456789") < n)
    errfmt("illegal number\n");
```

SEE ALSO

inbuf, instr, notstr, scnbuf, scnstr, subbuf, substr

NAME
notstr - find occurrence in string of character not in set

SYNOPSIS
```
BYTES notstr(p, s)
    TEXT *p, *s;
```

FUNCTION
notstr scans the NUL terminated string starting at p for the first occurrence of a character not in the NUL terminated set starting at s.

RETURNS
notstr returns the index of the first character in p not contained in the set s, or the index of the terminating NUL if all are in s.

EXAMPLE
To check a string for non-numeric characters:

```
if (str[notstr(str, "0123456789")])
    errfmt("illegal number\n");
```

SEE ALSO
inbuf, instr, notbuf, scnbuf, scnstr, subbuf, substr

NAME

ordbuf - compare two NUL padded buffers for lexical order

SYNOPSIS

```
COUNT ordbuf(s1, s2, n)
    TEXT *s1, *s2;
    COUNT n;
```

FUNCTION

ordbuf compares two text buffers, character by character, for lexical order in the character collating sequence. The first buffer starts at s1, the second at s2. Both buffers are n characters long.

Note that encoded numbers, such as int or double, seldom sort properly when treated as text strings.

RETURNS

The value returned is -1 when s1 is lower, 0 when s1 equals s2, and +1 when s2 is lower.

EXAMPLE

```
sort(nthings, &ordbuf, &swap, &data);
```

SEE ALSO

sort

NAME

pathnm - complete a pathname

SYNOPSIS

```
TEXT *pathnm(buf, n1, n2)
    TEXT *buf, *n1, *n2;
```

FUNCTION

pathnm builds a pathname in buf that is derived from the pair of NUL terminated names pointed at by n1 and n2.

Under Idris and Unix, the name pointed at by n2 is assumed to end in '/'. To this in buf is appended the longest suffix of the string pointed at by n1 that does not contain a '/'.

Under other operating systems, if the name pointed at by n2 ends in ':' or ']', then to it in buf is appended the longest suffix of the string pointed at by n1 that does not contain a ':', ']', or '/'. If the string pointed at by n2 does not end in ':' or ']', then a '/' followed by the same suffix is appended to the n2 string in buf.

Thus the following results are obtained:

n1	n2	buf
x	y	y/x
x	a:	a:x
x	[2,3]	[2,3]x
z/x	y	y/x
a:x	b:	b:x
:f1:x	:f2:	:f2:x

This scheme is designed to be maximally convenient on numerous operating systems, provided that truly esoteric filenames, such as "a/3:", are avoided.

RETURNS

pathnm returns the concatenation of n2, possibly a '/', and the suffix of n1, NUL terminated in the area pointed at by buf. The value of the function is always buf.

BUGS

There is no way to specify the size of buf, which must be at least lenstr(n1) + lenstr(n2) + 2 characters.

NAME

pattern - build a regular expression pattern

SYNOPSIS

```
TEXT *pattern(pat, delim, p)
    TEXT *pat, delim, *p;
```

FUNCTION

pattern builds an encoded pattern in the string buffer starting at pat, suitable for use with amatch or match in matching characters. The pattern is created from the string p; the creation of the pattern stops with the first unescaped occurrence in p of the character delim. To prevent an ill-formed pattern from confounding the code, p should be terminated with a NUL, whether or not a delim other than NUL is given.

Values used in the encoded pattern are given by the manual page for amatch; ordinary use, however, requires no knowledge of these inner workings. It is sufficient to know that the encoded string at pat will never occupy more than twice as many bytes as the length of the string p, up to and including the terminating occurrence of delim.

The encoded pattern is derived by interpreting the string at p as a series of "regular expressions". A regular expression is a shorthand notation for a set of target strings to be searched for in a buffer. Any of these target strings are said to "match" the regular expression. The following regular expressions are allowed:

An ordinary character is considered a regular expression that matches that character.

The character sequences "\b", "\f", "\n", "\r", "\t", "\v", in upper or lower case, each match the single ASCII characters backspace (BS), formfeed (FF), newline (NL), return (CR), tab (HT), and vertical tab (VT), respectively. In addition, any eight-bit character may be matched by "\ddd" where ddd is the one to three digit octal value of the character; this is the safest way to match most non-printing characters, and the only way to match ASCII NUL (\0).

A '?' matches any single character except a newline.

A '^' as the leftmost character of a series of regular expressions constrains the match to begin at the leftmost of the target characters.

A '*' following another regular expression matches zero or more occurrences of that expression. A '*' may thus match a null string, which occurs at the beginning of a line, between pairs of characters, or at the end of the line. A '*' enclosed in "\(" and "\)", or following either a '\' or an initial '^', is taken as a literal '*', however.

A literal '^', as just defined, or a '^' in any position other than the ones just mentioned, matches a '^' from the target string.

A '*' matches zero or more characters, not including newline. It is conceptually identical to the sequence "?^".

A character string enclosed in square brackets "[]" matches a single character that may be any of the characters in the bracketed string, but no other. However, if the first character of the string is a '!', then the expression matches any character except newline and the ones in the bracketed string. Inside the bracketed string, a range of characters in the character collating sequence may be indicated by the three-character sequence <low-character>, '-', <high-character>. If <low-character> is greater than <high-character>, the expression is ignored. Thus, [a0-9b] (or [ab0-9] or [0-9ba]) is a regular expression that will match one character, either a decimal digit or the letter a or b. When matching a literal "-", the "-" must be the first or last character in the bracketed list, or must immediately follow a range, as in [0-9-+]; otherwise it is taken to specify a range of characters.

A regular expression enclosed between the sequences "\(" and "\)" tags this expression in a way detectable by the amatch routine, but otherwise has no effect on the characters the expression matches. (See amatch for further explanation.)

A concatenation of regular expressions matches the concatenation of strings matched by individual regular expressions. In other words, a regular expression composed of a series of "sub-expressions" will match a concatenation of target character strings implied by each of the individual "sub-expressions".

A '$' as the rightmost character after a series of regular expressions constrains the match, if any, to end at the end of the target characters prior to the newline.

Note that arbitrary grouping and alternation are not fully supported by this notation, as the expressions accepted are not the full class of regular expressions beloved by mathematicians.

RETURNS

If no syntax errors were found in p, pattern returns a pointer to the occurrence of delim that terminated its scan. Otherwise pattern returns NULL.

EXAMPLE

To put out only those lines that have a 'T' as their seventh character:

```
pattern(pbuf, '/', "^??????T/");
while (match(buf, n = getlin(buf, MAXBUF), pbuf))
    putlin(buf, n);
```

SEE ALSO

amatch, match

NAME

prefix - test if one string is a prefix of the other

SYNOPSIS

```
BOOL prefix(s1, s2)
    TEXT *s1, *s2;
```

FUNCTION

prefix compares two strings, character by character, for equality. The first string starts at s1 and is terminated by a NUL '\0'; the second is likewise described by s2. The strings must match up to but not including the NUL terminating the second string, i.e., s2 must be a prefix of s1.

RETURNS

The value returned is YES if s2 is a prefix of s1, else NO.

EXAMPLE

```
if (prefix(line, "#include ")
    doinclude();
```

SEE ALSO

cmpbuf, cmpstr

NAME

ptc - put a character to output buffer

SYNOPSIS

```
ptc(pfio, c)
```

FUNCTION

c is treated as a character to be copied to the file controlled by the FIO buffer at pfio. If there is room in the buffer to put c, no function call is attempted; if the buffer is full, putc() is called to drain the buffer. ptc significantly reduces per character I/O overhead.

RETURNS

ptc returns c.

EXAMPLE

To copy a file, character by character:

```
while (ptc(&stdout, gtc(&stdin)) != EOF)
    ;
```

SEE ALSO

getc, getch, gtc, putc, putch

BUGS

Because it is a macro, ptc cannot be called from non-C programs, nor can its address be taken. An argument with side effects may be evaluated other than just once.

NAME

putc - put a character to output buffer

SYNOPSIS

```
COUNT putc(pfio, c)
    FIO *pfio;
    COUNT c;
```

FUNCTION

If c is not negative, it is treated as a character to be copied to the file controlled by the FIO buffer at pfio; otherwise putc simply ensures that all characters in the buffer are written out. pfio is normally the address of an FIO buffer declared by the user.

It may be necessary to explicitly drain the output buffer if pfio has not been initialized by finit, which takes care to drain the output buffer on exit from the user program. If the pfio buffer has been opened for WRITE, the output buffer is drained whenever a newline is encountered.

RETURNS

putc returns c. If any writes fail, the writerr condition is raised; if (pfio == NULL), the fioerr condition is raised.

EXAMPLE

To copy a text file, character by character:

```
FIO inf, outf;
    ...
fopen(&inf, "infile", READ);
fcreate(&outf, "outfile", WRITE);
while (putc(&outf, getc(&inf)) != EOF)
    ;
```

SEE ALSO

finit, fioerr, getc, getch, ptc, putch, writerr

BUGS

Arbitrary characters, as opposed to ASCII text, are sometimes sign extended to make negative integers; these quietly disappear on putc calls unless properly masked.

NAME

putch - put a character to output buffer stdout

SYNOPSIS

```
COUNT putch(c)
    COUNT c;
```

FUNCTION

If c is not negative, it is treated as a character to be copied to the file controlled by the FIO buffer stdout; otherwise putch simply ensures that all characters in the buffer are written out. It may be necessary to explicitly drain the stdout buffer in this fashion, unless stdout has been initialized by finit, which takes care to drain the output buffer on exit from the user program.

RETURNS

putch returns c. If any writes fail, putch raises the writerr condition.

EXAMPLE

To copy stdin to stdout, character by character:

```
while (putch(getch()) != EOF)
    ;
```

Note that this is exactly equivalent to:

```
IMPORT FIO stdin, stdout;
    ...
while (putc(&stdout, getc(&stdin)) != EOF)
    ;
```

SEE ALSO

finit, getc, getch, putc, stdout, writerr

BUGS

By default, the stdout buffer is drained only when the character written is a newline, or when putch is called with a negative argument. If stdout has not been explicitly initialized before use by the call

```
finit(&stdout, STDOUT, WRITE);
```

a partial line may not be drained on program termination. If non-text output is to be written to stdout, the call

```
finit(&stdout, STDOUT, BWRITE);
```

should be made before stdout is used.

Arbitrary characters, as opposed to ASCII text, are often sign extended to make negative integers; these quietly disappear on putch calls unless masked properly.

NAME

putf - output arguments formatted

SYNOPSIS

```
VOID putf(pfio, fmt, arg1, arg2, ...)
    FIO *pfio;
    TEXT *fmt;
    ...
```

FUNCTION

putf converts a series of arguments arg1, ... to text, which is output to the file controlled by the FIO buffer at pfio, under control of a format string at fmt. The format string consists of literal text to be output, interspersed with <field-specifier>s that determine how the arguments are to be interpreted and how they are to be converted for output.

A <field-specifier> takes the form:

```
%[+z|-z][#]<field-code>
```

That is, a <field-specifier> consists of a literal '%', followed by an optional "+z" or "-z", where z can be any character, followed by an optional field width #, and is terminated by a <field-code>. A "+z", if present, calls for the field to be left-filled with the character z, while "-z" calls for the output of right fill with z. A #, if present, specifies the total width in characters of the field to be output, and is either a decimal integer, or the letter 'n'. If an 'n' is given, then the value of the next argument from the argument list is taken to specify the field width.

For example, if arg1 is a double with the value 100.53, then:

```
putf(&stdout, "%+#9.2f", arg1);
putf(&stdout, "%+#n.nf", 9, 2, arg1);
```

both will output:

```
  ***100.53
```

If the number of characters needed to represent the output item is less than the field width, fill characters are used to left or right pad the item up to the field width. By default, left fill with spaces is used. The default field width is zero.

A <field-code> is composed of a <modifier>, a <specifier> or both. The <specifier> defines how an output field is to be represented, and is one of the following:

```
c = char integer
s = short integer
i = integer
l = long integer
p = NUL-terminated string
b = buffer of specified length
```

d = double output in scientific notation (e.g., 1.00e+00)
f = double output in fixed point notation (e.g., 1.00)
x = fill characters (usually spaces) only

A <modifier> causes an integer value to be output as:

a = ASCII characters
h = hexadecimal (no leading "0x")
o = octal (no leading '0')
u = unsigned decimal

If no <specifier> is given, it is presumed to be 'i', and a <modifier> given from the above series will be taken to apply to the implied integer field. If a <specifier> of 'c', 's', 'i' or 'l' is given with no <modifier>, the associated value is output in signed decimal.

In addition, an optional precision modifier, ".#", limits the number of characters actually output with a <specifier> of 'p' or 'b', and specifies the number of fractional digits output with a <specifier> of 'd' or 'f'. Like the field width specifier, the precision modifier # may be either an explicit integer, or an 'n', to make use of the next argument value in sequence.

Hence a <field-code> usually consists of one of the following combinations of <specifier> and <modifier>:

```
[a|h|o|u]{c|s|i|l}      /* integer output */
[.#]{b|p|d|f}           /* string or floating output */
{a|h|o|u}               /* default specifier is i */
{x}                     /* just output fill characters */
```

Any other character in the place of a <field-code> is taken as a single literal character to be output, permitting a '%' to be output with a "%%" specification.

The 'a' modifier treats the integer as a sequence of characters of the appropriate length, and outputs the characters in descending order of their significance within the number. This permits multi-byte binary data to be written to a file in a host independent manner.

A string of characters may be output under the 'p' field code, if it is NUL-terminated, or under 'b' if its length is known.

If arg2 is a vector containing the 11-character NUL-terminated string "hello world", either of these calls would output the string:

```
putf(&stdout, "%p\n", arg2);
putf(&stdout, "%b\n", arg2, 11);
```

In the first case, the argument is a pointer to the beginning of the string; in the second case two arguments are used, one a pointer to the start of the string and the second an integer specifying its length. In either case, the number of characters actually output will be no more than the precision modifier, if it is present and non-zero.

A double (or float) number may be output with 'd' or 'f', the precision modifier specifying the number of characters to the right of the decimal point. For the 'd' field code, the number is written in the scientific notation form

```
[-]#.##e{+|-}##
```

There is always one digit to the left of the decimal point; there are either two or three digits in the exponent, depending on the target machine. The 'f' field code prints the number in fixed point format, i.e., without exponent. In either case, no more than 24 characters will be output.

For example, if arg1 is a double with the value 100.53, then:

```
putf(&stdout, "%1.4d\n", arg1);
```

would output it as:

```
1.0053e+02
```

while:

```
putf(&stdout, "$%6.nf", 2, arg1);
```

would output it in fixed point notation as:

```
$100.53
```

The 'x' field code consumes no arguments; it is a convenient way to output pure filler.

RETURNS

Nothing. An error exit occurs if any writes fail, or if (pfio == NULL).

EXAMPLE

```
putf(&stdout, "%i errors in file %p\n", nerrors, fname);
```

SEE ALSO

decode, dtoe, dtof, errfmt, putfmt

BUGS

A call with more <field-specifier>s than argument variables will produce unpredictable results.

NAME

putfmt - output arguments formatted to stdout

SYNOPSIS

```
VOID putfmt(fmt, arg1, arg2, ...)
    TEXT *fmt;
```

FUNCTION

putfmt writes formatted output to the file controlled by the FIO buffer stdout, using the format string at fmt and the arguments argx, in exactly the same way as putf.

RETURNS

Nothing. If any writes fail, the writerr condition is raised.

EXAMPLE

```
putfmt("%i:%p\n", lineno, line);
```

SEE ALSO

decode, errfmt, finit, putf, stdout, writerr

BUGS

The stdout buffer is drained only when the last character written is a newline. If stdout has not been explicitly initialized before use by the call

```
finit(&stdout, STDOUT, WRITE);
```

a partial line may not be drained on program termination. If non-text output is to be written to stdout, the call

```
finit(&stdout, STDOUT, BWRITE);
```

should be made before stdout is used.

NAME

putl - put a text line to output buffer

SYNOPSIS

```
BYTES putl(pfio, s, n)
    FIO *pfio;
    TEXT *s;
    BYTES n;
```

FUNCTION

putl copies characters from the n character buffer starting at s to the file controlled by the FIO buffer at pfio. pfio is normally the address of an FIO buffer declared by the user.

RETURNS

putl returns n. If any writes fail, the writerr condition is raised; if (pfio == NULL), the fioerr condition is raised.

EXAMPLE

To copy a textfile, line by line:

```
FIO inf, outf;
    ...
fopen(&inf, "infile", READ);
fopen(&outf, "outfile", WRITE);
while (putl(&outf, buf, getl(&inf, buf, BUFSIZE)))
    ;
```

SEE ALSO

fioerr, getl, getlin, putlin, writerr

NAME

putlin - put a text line to output buffer stdout

SYNOPSIS

```
BYTES putlin(s, n)
    TEXT *s;
    BYTES n;
```

FUNCTION

putlin copies characters from the n character buffer starting at s to the file controlled by the FIO buffer stdout.

RETURNS

putlin returns n. If any writes fail, the writerr condition is raised.

EXAMPLE

To copy stdin to stdout, line by line:

```
while (putlin(buf, getlin(buf, BUFSIZE)))
    ;
```

Note that this is exactly equivalent to:

```
IMPORT FIO stdin, stdout;
    ...
while(putl(&stdout, buf, getl(&stdin, buf, BUFSIZE)))
    ;
```

SEE ALSO

finit, getl, getlin, putl, stdout

BUGS

The stdout buffer is drained only when the last character written is a newline. If stdout has not been explicitly initialized before use by the call

```
finit(&stdout, STDOUT, WRITE);
```

a partial line may not be drained on program termination. If non-text output is to be written to stdout, the call

```
finit(&stdout, STDOUT, BWRITE);
```

should be made before stdout is used.

NAME

putstr - copy multiple strings to file

SYNOPSIS

```
VOID putstr(fd, arg1, arg2,  ..., NULL)
    FILE fd;
    TEXT *arg1, *arg2,  ...;
```

FUNCTION

putstr writes a series of strings out to the file with descriptor fd. Each string begins at arg1, ... and is terminated by a NUL '\0'. The series of string arguments is terminated by a NULL pointer argument. For each string, putstr invokes lenstr to discover its size and issues a call to fwrite; therefore, putstr should only be used for low volume output.

RETURNS

Nothing. If any writes fail, the writerr condition is raised.

EXAMPLE

```
putstr(STDERR, fname, ": bad format\n", NULL);
```

SEE ALSO

fwrite, lenstr, writerr

BUGS

Forgetting the terminating NULL pointer is usually disastrous.

NAME

readerr - read error condition

SYNOPSIS

```
TEXT *readerr
```

FUNCTION

readerr is the condition raised when a read error occurs. If no handler is provided, the message written to STDERR is "read error".

NAME

remark - print non-fatal error message

SYNOPSIS

```
VOID remark(s1, s2);
    TEXT *s1, *s2;
```

FUNCTION

remark prints an error message to STDERR, consisting of the concatenation of strings s1 and s2, followed by a newline. It then returns to the caller for further processing.

RETURNS

Nothing.

EXAMPLE

```
    if ((fd = open(name, READ, 0)) < 0)
        remark("can't open: ", name);
```

SEE ALSO

errfmt, error, putstr

NAME

round - round real to integer

SYNOPSIS

```
ARGINT round(x)
    DOUBLE x;
```

FUNCTION

round increases the absolute value of its argument by 0.5, then finds the nearest integer closer to zero than the modified argument.

RETURNS

round returns the integer corresponding to its argument, rounded.

SEE ALSO

trunc

NAME

scnbuf - scan buffer for character

SYNOPSIS

```
BYTES scnbuf(s, n, c)
    TEXT *s;
    BYTES n;
    TEXT c;
```

FUNCTION

scnbuf looks for the first occurrence of a specific character c in an n character buffer starting at s.

RETURNS

scnbuf returns the index of the first character that matches c, or n if none.

EXAMPLE

To map keybuf[] characters into subst[] characters:

```
if ((n = scnbuf(keybuf, KEYSIZ, *s)) != KEYSIZ)
    *s = subst[n];
```

SEE ALSO

inbuf, instr, notbuf, notstr, scnstr, subbuf, substr

NAME

scnstr - scan string for character

SYNOPSIS

```
BYTES scnstr(s, c)
    TEXT *s, c;
```

FUNCTION

scnstr looks for the first occurrence of a specific character c in a NUL terminated target string s.

RETURNS

scnstr returns the index of the first character that matches c, or the index of the terminating NUL if none does.

EXAMPLE

To map keystr[] characters into subst[] characters:

```
if (s[n = scnstr(keystr, *s)])
    *s = subst[n];
```

SEE ALSO

inbuf, instr, notbuf, notstr, scnbuf, subbuf, substr

NAME

sin - sine in radians

SYNOPSIS

```
DOUBLE sin(x)
    DOUBLE x;
```

FUNCTION

sin computes the sine of x, expressed in radians, to full double precision. It works by scaling x in quadrants, then computing the appropriate sin or cos of an angle in the first half quadrant, using a sixth order telescoped Taylor series approximation. If the magnitude of x is too large to contain a fractional quadrant part, the value of sin is 0.

RETURNS

sin returns the nearest internal representation to sin x, expressed as a double floating value.

EXAMPLE

To rotate a vector through the angle theta:

```
IMPORT DOUBLE cos(), sin();
    ...
xnew = xold * cos(theta) - yold * sin(theta);
ynew = xold * sin(theta) + yold * cos(theta);
```

SEE ALSO

cos

NAME

sort - sort items in memory

SYNOPSIS

```
VOID sort(n, ordf, excf, base)
    ARGINT n;
    COUNT (*ordf)();
    VOID (*excf)();
    TEXT *base;
```

FUNCTION

sort orders n items in memory using the quicksort algorithm. It decides whether items i and j are in order by performing the call

```
(*ordf)(i, j, &base);
```

where i and j are both guaranteed to be in the range [0, n). If (item i is to sort less than item j) then the value returned must be less than zero; otherwise if (item i is to sort equal to item j) then the value returned must be zero; the value is otherwise unconstrained.

To exchange two items, sort makes the call

```
(*excf)(i, j, &base);
```

and henceforth presumes that the items are interchanged.

Note that it is the address of base that is passed to both functions. This permits multiple parameters to follow base in the original argument list, which can be accessed as members of a structure pointed to by &base, providing the structure is declared with careful knowledge of how C passes arguments. For ordering and exchange functions in the know, base can also simply be ignored.

RETURNS

Nothing. The items are sorted in place.

EXAMPLE

To sort an array of short integers in ascending order:

```
COUNT iord(i, j, pa)
    COUNT i, j, **pa;
    {
    return ((*pa)[i] - (*pa)[j]);
    }

VOID iswap(i, j, pa)
    COUNT i, j, **pa;
    {
    COUNT t;

    t = (*pa)[i], (*pa)[i] = (*pa)[j], (*pa)[j] = t;
    }
```

```
VOID isort(a, n)
    COUNT a[], n;
    {
    sort(n, &iord, &iswap, a);
    }
```

BUGS

It can´t sort more than half of memory, i.e., n is taken as signed and must be positive.

NAME

sqr - square an argument

SYNOPSIS

```
DOUBLE sqr(d)
    DOUBLE d;
```

FUNCTION

sqr finds the square of its argument.

No check is made against overflow.

RETURNS

sqr returns the square of its argument which is a double.

NAME

sqrt - real square root

SYNOPSIS

```
DOUBLE sqrt(x)
    DOUBLE x;
```

FUNCTION

sqrt computes the square root of x to full double precision. It works by expressing x as a fraction in the interval [1/2, 1), times an integer power of two. The square root of the fraction is obtained by three iterations of Newton´s method, using a quadratic approximation as a starting value.

RETURNS

sqrt returns the nearest internal representation to sqrt x, expressed as a double floating value. If x is negative, a domain error condition is raised.

EXAMPLE

To find the magnitude of a vector:

```
IMPORT DOUBLE sqrt();
    ...
mag = sqrt(x * x + y * y);
```

SEE ALSO

_domain(IV), exp

NAME

squeeze - delete specified character from buffer

SYNOPSIS

```
BYTES squeeze(s, n, c)
    TEXT c, *s;
    BYTES n;
```

FUNCTION

squeeze deletes character c from the n-character buffer starting at s, and compresses it in place.

RETURNS

squeeze returns the number of characters remaining in s, which is in the interval [0, n].

EXAMPLE

To write out a buffer after stripping off NULs and carriage returns:

```
write(STDOUT, buf, squeeze(buf, squeeze(buf, BUFSIZE, '\0'), '\r'));
```

SEE ALSO

fill

NAME

stdin - the standard input control buffer

SYNOPSIS

```
FIO stdin;
```

FUNCTION

stdin is an FIO control buffer initialized for input from STDIN.

EXAMPLE

To count lines:

```
for (nl = 0; getl(&stdin, buf, BUFSIZE); ++nl)
    ;
```

SEE ALSO

stdout

NAME
stdout - the standard output control buffer

SYNOPSIS
```
FIO stdout;
```

FUNCTION
stdout is an FIO control buffer initialized for output to STDOUT.

EXAMPLE
```
putl(&stdout, outbuf, outsiz);
```

SEE ALSO
finit, stdin

BUGS
stdout should not be used for non-text output unless initialized before use by

```
finit(&stdout, STDOUT, BWRITE);
```

NAME

stob - convert short to text in buffer

SYNOPSIS

```
BYTES stob(s, i, base)
    TEXT *s;
    COUNT i;
    COUNT base;
```

FUNCTION

stob converts the short i to a text representation in the buffer starting at s. The number is represented in the base specified, using lower case letters beginning with 'a' to specify digits from 10 on. If (0 < base) the number i is taken as unsigned; otherwise if (base < 0) negative numbers have a leading minus sign and are converted to -base; if (base == 0) it is taken as -10. Only magnitudes of base between 2 and 36 are generally meaningful, but no check is made for reasonableness.

RETURNS

The value returned is the number of characters used to represent the short, which in hexadecimal can be up to four digits plus sign.

EXAMPLE

To output i in decimal:

```
write(STDOUT, buf, stob(buf, i, 10));
```

SEE ALSO

btoi, btol, btos, itob, ltob

BUGS

The length of the buffer is not specifiable. If (|base| == 1) the program can bomb; if (36 < |base|) funny characters can be inserted in the buffer.

NAME

subbuf - find occurrence of substring in buffer

SYNOPSIS

```
BYTES subbuf(s, ns, p, np)
    TEXT *s, *p;
    BYTES ns, np;
```

FUNCTION

subbuf scans the buffer starting at s of size ns, and looks for the first occurrence of the substring at p of size np.

RETURNS

The value returned is the index in s of the first character in the substring if subbuf is successful; otherwise, ns is returned. If np is zero, zero is returned; i.e., a null string always matches the start of the buffer.

EXAMPLE

```
for(p = buf, i = size; (j = subbuf(p, i, "\r\n", 2)) < i;
    p =+ j + 2, i =- j + 2)
    {
    write(fd, p, j);
    write(fd, "\n", 1);
    }
```

SEE ALSO

amatch, inbuf, instr, match, notbuf, notstr, scnbuf, scnstr, substr

NAME

substr - find occurrence of substring

SYNOPSIS

```
BYTES substr(s, p)
    TEXT *s, *p;
```

FUNCTION

substr scans the string starting at s, and looks for the first occurrence of the substring at p.

RETURNS

The value returned is the index in s of the first character in the substring if substr is successful; otherwise, the index of the terminating NUL is returned.

EXAMPLE

```
if (line[substr(line, "Page")])
    putfmt("%s: %\n", lno / 66 + 1, line);
```

SEE ALSO

amatch, inbuf, instr, match, notbuf, notstr, scnbuf, scnstr, subbuf

NAME

tolower - convert character to lowercase if necessary

SYNOPSIS

tolower(c)

FUNCTION

tolower converts an uppercase letter to its lowercase equivalent, leaving all other characters unscathed.

RETURNS

tolower is a numerical rvalue guaranteed not to be an uppercase character.

EXAMPLE

To accumulate a hexadecimal digit:

```
if ('a' <= c && c <= 'f' || 'A' <= c && c <= 'F')
    sum = sum * 10 + tolower(c) + (10 - 'a');
```

SEE ALSO

isalpha, isdigit, islower, isupper, iswhite, toupper

BUGS

Because it is a macro, tolower cannot be called from non-C programs, nor can its address be taken. Arguments with side effects may be evaluated other than just once.

NAME

toupper - convert character to uppercase if necessary

SYNOPSIS

toupper(c)

FUNCTION

toupper converts a lowercase letter to its uppercase equivalent, leaving all other characters unscathed.

RETURNS

toupper is a numerical rvalue guaranteed not to be a lowercase character.

EXAMPLE

To convert a character string to uppercase letters:

```
for (i = 0; i < size; ++i)
    buf[i] = toupper(buf[i]);
```

SEE ALSO

isalpha, isdigit, islower, isupper, iswhite, tolower

BUGS

Because it is a macro, toupper cannot be called from non-C programs, nor can its address be taken. Arguments with side effects may be evaluated other than just once.

NAME

trunc - truncate real to integer

SYNOPSIS

```
ARGINT trunc(d)
    DOUBLE d;
```

FUNCTION

trunc converts a real number to the nearest integer closer to zero than the original number.

RETURNS

trunc returns the integer corresponding to its argument, truncated.

SEE ALSO

round

NAME

usage - output standard usage information

SYNOPSIS

```
COUNT usage(msg)
    TEXT *msg;
```

FUNCTION

usage outputs to STDERR the string "usage: <pname> ", followed by the string pointed to by msg, where <pname> is the name by which the current program was invoked. If msg is terminated with a newline, usage immediately takes an error exit.

RETURNS

If usage returns to the caller, its value is the number of characters output to STDERR.

EXAMPLE

```
if (1 < aflag + bflag + nflag)
usage("-[a b n] <files>\n");
```

SEE ALSO

_pname(III), getflags

NAME

writerr - write error condition

SYNOPSIS

```
TEXT *writerr
```

FUNCTION

writerr is the condition raised when a write error occurs. If no handler is provided, the message written to STDERR is "write error".