SECTION FOUR

IDRIS SUPPORT LIBRARY

NAME

Conventions - the Idris support library

SYNOPSIS

/lib/libi*

FUNCTION

The functions documented in this section are kept in a library, separate from the standard C library, with a name whose prefix is /lib/libi (and whose suffix is machine dependent). They are used extensively in the construction of standard Idris utilities, to perform common functions uniformly and to enforce communication protocols among utilities. They are thus quite useful to anyone wishing to add utilities that are to cooperate with the existing community.

Most of the notation used here is the same as in earlier sections of this manual, but there are a few additional types that creep in from various header files:

BLOCK an unsigned short, capable of holding any filesystem block number. Block 0 is often taken as the absence of a block number. Defined in /lib/ino.h.

FINODE a filesystem inode, possibly as represented on the disk or possibly in native byte order. Defined in /lib/ino.h.

INUM an unsigned short, capable of holding any filesystem inode number. Inode 0 does not exist. Defined in /lib/ino.h.

TVEC a time vector, used to communicate parsed dates among library functions. Defined in /lib/time.h as:

```
typedef struct {
    BYTES secs;     /* seconds      [0, 60) */
    BYTES mins;     /* minutes      [0, 60) */
    BYTES hrs;      /* hours        [0, 24) */
    BYTES dmth;     /* day of month         [1, 31] */
    BYTES mth;      /* month of year        [0, 12) */
    BYTES yr;       /* years since 1900     [70, 131) */
    BYTES dwk;      /* day of week, sunday = 0  [0, 7) */
    BYTES dyr;      /* day of year              [0, 365] */
    BOOL dstf;      /* non-zero if daylight savings time */
    } TVEC;
```

Also included in this library is a version of putf (the internal routine _putf to be exact) that cannot deal with the floating point conversions %d and %f. This is provided so that utilities can perform formatted output -- using decode, errfmt, putf, and putfmt -- without dragging in (possibly extensive) floating point runtime support code that will never be needed. Be warned, however, that automatically including the Idris support library with the standard compile and link scripts will lead to puzzling behavior in a program that expects to perform floating output.

NAME

_penable - control function entry counts in profiling

SYNOPSIS

```
TBOOL _penable;
```

FUNCTION

_penable is used by the function entry counting routine of the profiling package, to control whether or not calls to the routine actually will record function entries. If _penable is non-zero, function entries will be counted; otherwise, the counting routine returns without doing anything.

_profil() sets _penable to 1 just before returning. _proend() clears _penable just after being called.

SEE ALSO

_proend, _profil

NAME

_proend - end profiling

SYNOPSIS

VOID (*proend())()

FUNCTION

_proend terminates profiling, by clearing the one-byte flag _penable to disable function entry counting, calling profil() to end time profiling, and writing out the time profiling and entry count buffers to the file named in the last preceding call to _profil().

RETURNS

Since _proend() is linked into the chain of programs to be called on program exit or interrupt, it returns the address of the next one to call.

SEE ALSO

_penable, _profil

NAME

_profil - start profiling

SYNOPSIS

```
TEXT *_profil(p)
    struct {
        UCOUNT config, esize;
        BYTES pbuf, psize, offset, scale, tbias, ebias;
        TEXT *fname;
        } *p;
```

FUNCTION

_profil performs a number of useful housekeeping functions preparatory to commencing program profiling. Its single argument, p, is a pointer to a standard profile file header, which is immediately followed by the name of the file to be generated by _proend().

_profil computes file header parameters psize and scale, allocates space for the time profiling and entry count buffers at pbuf, calls profil() to initiate time profiling, and ensures that _proend() is called on program interrupt or exit. Finally, it enables function entry counting by setting the one-byte flag _penable to 1.

RETURNS

_profil returns the address of the first byte past the end of the buffer space it allocated at pbuf.

SEE ALSO

_penable, _proend

NAME

askpw - ask for a password

SYNOPSIS

```
TEXT *askpw(kbuf, key)
    TINY kbuf[8];
    TEXT *key;
```

FUNCTION

askpw fills kbuf with the NUL terminated password at key, or if (key == NULL) reads a line from STDIN to fill kbuf. Before reading a line from a terminal as STDIN, askpw will drain outstanding input by performing an stty, prompt with the string "\7password:\ ", and accept the subsequent line, echoing only the trailing newline.

In any case, the first eight characters of the password are used; a short string is padded on the right with NULs.

Note that a password may thus be obtained from any of three sources: from an argument to the function, from a terminal with prompting and with printing suppressed, or from a noninteractive STDIN with no prompting.

RETURNS

askpw returns the address of kbuf, which contains the NUL padded password.

EXAMPLE

```
bldks(ks, askpw(kbuf, NULL));
```

SEE ALSO

codepw

NAME

asure - get user response to question

SYNOPSIS

```
BOOL asure(p)
    TEXT *p;
```

FUNCTION

If STDIN looks like a terminal, asure drains its input by performing an stty, writes the NUL terminated string at *p to STDERR, followed by a space, then reads up to 32 characters from STDIN.

This meticulous sequence of events is useful when a program needs to be assured that a conscious human accomplice is present.

RETURNS

If the line read begins with a 'y' or 'Y', or if STDIN is not a terminal, asure returns YES; otherwise it returns NO.

EXAMPLE

```
if (asure("are you sure?"))
    scrog();
```

NAME

atime - convert time vector to ASCII string

SYNOPSIS

```
TEXT *atime(vt, s)
    TVEC *vt;
    TEXT *s;
```

FUNCTION

atime converts the time vector at vt to a 24 character string at s, having the form:

```
        Thu Aug 12 09:53:12 1980
```

There is no terminating NUL or newline.

RETURNS

atime writes the date in ASCII at s[0] through s[23], and returns s.

EXAMPLE

To print the date:

```
IMPORT LONG time();
IMPORT TEXT *_est, *_edt;
INTERN TEXT buf[] {"01234567890123456789012 3 "};
TVEC tvec;

ltime(&tvec, time(NULL));
putstr(STDOUT, atime(&tvec, buf), tvec.dstf ? _edt : _est,
    "\n", NULL);
```

SEE ALSO

ltime, vtime

NAME

baudcode - return code given speed text

SYNOPSIS

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

FUNCTION

baudcode compares the text string s with the table baudlist and returns the index of the matching string.

RETURNS

The speed code in the range [0, 15], or 16, if the lookup fails.

EXAMPLE

```
#include <sys.h>

BITS s;
SGTTY tbuf;
TEXT *speed = NULL;

getflags(&ac, &av, "s*:F", &speed);
gtty(fd, &tbuf);
if (speed && (s = baudcode(speed)) < 16)
    {
    tbuf.t_speeds =& ~(T_OSPEED|T_ISPEED);
    tbuf.t_speeds =! s << 8 | s;
    stty(fd, &tbuf);
    }
else
    remark(speed, ": unavailable baudrate");
```

SEE ALSO

baudlist, baudtext

NAME

baudlist - list of speeds supported by Idris drivers

SYNOPSIS

```
TEXT baudlist[NBAUD] {
    "0", "50", "75", "110"
    "134.5", "150", "200", "300",
    "600", "1200", "1800", "2400",
    "4800", "9600", "19200", "38400"};
```

FUNCTION

baudlist is a table of text speeds supported by Idris drivers. The table is indexed by a code in the range [0, 15].

SEE ALSO

baudcode, baudtext

NAME
baudtext - return text speed given speed code

SYNOPSIS
```
TEXT *baudtext(c)
    UCOUNT c;
```

FUNCTION
baudtext returns a pointer to the speed text string corresponding to the baudrate code c. The actual strings live in the table baudlist[].

RETURNS
A pointer to the speed text string, or NULL if code was not in the range [0, 15].

SEE ALSO
baudcode, baudlist

NAME
clrbuf - clear a standard sized buffer

SYNOPSIS
```
VOID clrbuf(buf)
    TEXT buf[BUFSIZE];
```

FUNCTION
clrbuf writes zeros throughout a 512-byte buffer beginning at buf.

RETURNS
Nothing, except a clear buffer.

EXAMPLE
```
clrbuf(p->_buf);
```

NAME

codepw - encode a password

SYNOPSIS

```
TEXT *codepw(loginid, kbuf)
    TEXT *loginid, kbuf[8];
```

FUNCTION

codepw encrypts the password in kbuf, then translates it to a printable form, suitable for storing in a textfile. The encrypted form is obtained by using the DES algorithm to encrypt the first eight characters of the file /adm/salt, exclusive-ored character by character with the NUL terminated string at loginid, using the password as a key. If the salt file is not readable, the string "password" is used in its stead.

The resulting eight-character encrypted string is repacked as twelve six-bit characters, using the alphabet [0-9a-zA-Z/.]. For convenience, a NUL is placed at the end of this string.

RETURNS

codepw returns a pointer to the NUL terminated, twelve-character encoded password which is stored in an internal buffer.

EXAMPLE

```
if (!cmpstr(codepw("root", askpw(kbuf, NULL)), getpw("root", 0, 1)))
    error("sorry", NULL);
```

FILES

/adm/salt for the string to be encrypted.

SEE ALSO

askpw, getpw

NAME

cpyi - copy an inode converting between native and filesystem

SYNOPSIS

```
FINODE *cpyi(dest, src)
    FINODE *dest, *src;
```

FUNCTION

cpyi copies the entire FINODE structure pointed to by src into the structure at dest, ensuring that all fields are converted if native byte order differs from filesystem byte order. For portability, its use is encouraged even on machines that need no conversion.

It is permissible for src and dest to be the same.

RETURNS

cpyi returns dest.

EXAMPLE

To get a pointer to an inode in native order:

```
FINODE *getino(fd, ino)
    FILE fd;
    INUM ino;
    {
    INTERN FINODE buf[BUFSIZE / sizeof (FINODE)], ibuf;

    if (!getblk(fd, buf, inblk(ino)))
        return (NULL);
    else
        return (cpyi(&ibuf, ioff(buf, ino)));
    }
```

NAME

cwd - get current working directory

SYNOPSIS

```
COUNT cwd(tbuf)
    TEXT tbuf[NAMSIZE];
```

FUNCTION

cwd determines the absolute pathname of the current working directory by tracing .. entries from . back to the root. The result is placed in tbuf as a NUL terminated string. If the current directory is on a mounted filesystem the mount history file /adm/mtab is used to determine the prefix of the absoute pathname.

If the path to the root of the filesystem becomes longer than 64 characters, including the terminating NUL, cwd returns the system error code -E2BIG. If cwd cannot find an entry in a directory that it needs it returns the system error code -EMLINK. If an error occurs while trying to open any directories for reading, the partially formed path is left in tbuf and cwd returns the appropriate system return code.

RETURNS

cwd writes a NUL terminated string at tbuf. If successful, the return value is zero; otherwise, the return value is a negative number indicating cwd's displeasure (as one of the Idris error return codes, negated).

EXAMPLE

To map a pathname to absolute form:

```
if (name[0] == '/')
    cpystr(abuf, name, NULL);
else if (cwd(build) < 0)
    error("broken directory tree", NULL);
else
    cpystr(abuf, build, "/", name, NULL);
```

FILES

/adm/mtab for mounted filesystems, . for current directory, \&..[/..]^ for parents.

NAME

devname - get device name

SYNOPSIS

```
BOOL devname(s, mdev, cspec)
    TEXT *s;
    UCOUNT mdev;
    BOOL cspec;
```

FUNCTION

devname fills the buffer pointed to by s with the NUL terminated device name in the /dev directory matching the major/minor device code contained in mdev. If cspec is nonzero, the device must be a character special device; otherwise, the device must be a block special device.

RETURNS

devname returns YES if it could find the appropriate device entry. The buffer at s is filled in with the device name, written as a 14-character link right filled with NULs.

EXAMPLE

The terminal message control function is:

```
BOOL mesg(new)
    BOOL new;
    {
    FAST BOOL old;
    STAT node;
    TEXT buf[20];

    if (fstat(STDERR, &node) < 0)
        return (NO);
    old = (node.s_mode & 022) ? YES : NO;
    cpybuf(buf, "/dev/", 5);
    buf[19] = '\0';
    if (devname(buf + 5, node.s_addr[0], YES))
        chmod(buf, new ? 0622 : 0600);
    return (old);
    }
```

NAME

ename - get pathname of an entry in a directory

SYNOPSIS

```
TEXT *ename(pname, dname, pdir)
    TEXT *pname, *dname;
    DIR *pdir;
```

FUNCTION

ename creates a fully qualified pathname consisting of the directory named dname to which is appended a '/' and the entry name pointed at by pdir. This NUL terminated pathname is returned at pname.

pname should be large enough to hold lenstr(dname) + 16 characters, counting the terminating NUL.

RETURNS

ename returns a pointer to the entry name.

EXAMPLE

```
if(!cmpstr(".", pdir->d_name) && !cmpstr("..", pdir->d_name))
    remove(ename(entry, dir, pdir));
```

NAME
flushi - flush out any pending inode writes

SYNOPSIS
```
VOID flushi(fd)
    FILE fd;
```

FUNCTION
flushi writes the in-core buffer used by geti and puti to the filesystem controlled by fd. If there have been no changes to the buffer, no output is performed. In any case, the buffer is disqualified, and the next geti or puti is guarenteed to read a fresh block from the filesystem.

RETURNS
If there is pending output, and it cannot be written, the writerr condition is raised.

EXAMPLE
To process the entire inode list of a filesystem:

```
for (i = isize << 4; 1 <= i; --i)
    if (process(pi = geti(fd, &ibuf, i)))
        puti(fd, pi, i);
flushi(fd);
```

SEE ALSO
geti, puti

NAME

ftime - find modified or accessed time of a file

SYNOPSIS

```
LONG ftime(fd, modflag)
    FILE fd;
    BOOL modflag;
```

FUNCTION

ftime finds the time of last access, or the time of last modification if modflag is true, to the file under control of fd.

RETURNS

ftime returns the time specified by modflag in seconds since 1 Jan 1970, or zero if the file status is unobtainable.

EXAMPLE

To print the date on which file "xeq" was last modified:

```
INTERN TEXT buf[] {"01234567890123456789012345678901234567890123"};
FILE fd = open("xeq", READ, 0);
TVEC tvec;

putstr(STDOUT, atime(ltime(&tvec, ftime(fd, YES)), buf),
    "\n", NULL);
```

SEE ALSO

atime, ltime

NAME
getblk - get filesystem block

SYNOPSIS
```
BOOL getblk(fd, buf, bno)
    FILE fd;
    TEXT *buf;
    BLOCK bno;
```

FUNCTION
getblk reads the 512-byte block whose number is bno into buf from the file under control of fd.

RETURNS
getblk returns YES only if exactly 512 characters were read.

EXAMPLE
```
if (!getblk(fd, superbuf, 1))
    error("can't read filsys", NULL);
```

SEE ALSO
mapblk, putblk

NAME

getdn - get device name

SYNOPSIS

```
FILE getdn(s, fname, mode)
    TEXT *s, *fname;
    BOOL mode;
```

FUNCTION

getdn opens a block or character special device using mode as the mode argument to the open call. If the file fname exists and is a block or character special device, it is opened. If the file fname does not exist, to it is prepended "/dev/" for a second try. If fname does exist but is not block or character special, then the block or character special device on which fname exists is opened instead, if that can be determined by calls to devname.

RETURNS

getdn returns a file descriptor for the opened file, or a negative number which is the Idris return code, negated. The name of the opened file, or the best guess at one on failure, is copied to s.

EXAMPLE

```
if ((fd = getdn(buf, fname, pflag ? UPDATE : READ)) < 0)
    error("invalid pathname: ", buf);
```

SEE ALSO

devname, stat

NAME

geti - get inode from filesystem

SYNOPSIS

```
FINODE *geti(fd, buf, ino)
    FILE fd;
    FINODE *buf;
    INUM bno;
```

FUNCTION

geti reads the inode whose number is ino from the filesystem under control of fd. It then copies the inode into buf, converting from filesystem format to in-core format in the process.

geti shares an in-core buffer of 16 inodes (1 block) with puti, and will use the contents of the buffer when appropriate. Thus, the buffer should be disqualified by calling flushi before switching filesystems.

RETURNS

If geti cannot read the necessary inode it will raise the readerr condition. If there is pending inode output from puti and the inode cannot be written, the writerr condition will be raised. Otherwise, geti will return buf, the pointer to the in-core inode.

EXAMPLE

To process the entire inode list of a filesystem:

```
for (i = isize << 4; 1 <= i; --i)
    if (process(pi = geti(fd, &ibuf, i)))
        puti(fd, pi, i);
flushi(fd);
```

SEE ALSO

flushi, puti

NAME

getlinks - read and sort a directory

SYNOPSIS

```
DIR *getlinks(dirname, nentries, size)
    TEXT *dirname;
    BYTES *nentries;
    LONG size;
```

FUNCTION

getlinks allocates an array of size bytes, reads a directory into it, then sorts the entries in lexical order on link names, but with zero inodes at the end. The number of non-null entries in the directory, including . and .. is written at nentries. size is the size in bytes of the directory, dirname.

RETURNS

getlinks returns a pointer to the first directory entry (usually .), or NULL if the directory cannot be read. The pointer is suitable for later use on a free call.

EXAMPLE

To print all the entries in a directory, and then free the allocated space:

```
IMPORT LONG lsize();
BYTES nentries;
DIR *pdir, *p;
STAT dstat;
TEXT *dirname;

if (stat(dirname, &dstat) < 0)
    putstr(STDERR, dirname, " does not exist\n", NULL);
else
    {
    pdir = getlinks(dirname, &nentries, lsize(&dstat->s_mode));
    for (p = pdir; nentries--; ++p)
        putstr(STDOUT, p->d_name, "\n", NULL);
    free(pdir, NULL);
    }
```

NAME

getpw - retrieve a field from the password file

SYNOPSIS

```
TEXT *getpw(matchstr, matchfld, wantfld)
    BYTES matchfld, wantfld;
    TEXT *matchstr;
```

FUNCTION

getpw scans the system password file for a line with a field, specified by matchfld, that matches the NUL terminated string pointed at by matchstr.

Fields in the password file are separated by colons on a text line, as follows:

FIELD	CONTENTS
0	loginid
1	encrypted password
2	user number
3	group number
4	long name
5	home directory
6	shell

Lines are assumed to be no longer than 128 bytes.

getpw remembers the last password line obtained in an internal buffer; consequently multiple calls leading to the same line are reasonably efficient. The line should not be corrupted by the calling program, however, nor should its contents be trusted if other getpw calls intervene.

RETURNS

getpw returns a pointer to the field specified by wantfld, in the matched field. If no match is found, NULL is returned. If the field is found, but the desired field does not exist, the returned pointer will be pointing at newline.

EXAMPLE

To find the loginid of user number 5:

```
TEXT *p, id[8];

if ((p = getpw("5", 2, 0))
    {
    len = instr(p, ":\n");
    cpybuf(id, p, min(len, sizeof (id));
    }
```

FILES

/adm/passwd for the password file.

BUGS

Lines should be up to 512 bytes in the password file.

NAME
inblk - find home block of an inode

SYNOPSIS
```
BLOCK inblk(ino)
    INUM ino;
```

FUNCTION
inblk locates the block number containing ino for any desired inode.

RETURNS
inblk returns the correct block number.

EXAMPLE
To get an inode:

```
FINODE *getino(fd, ino)
    FILE fd;
    INUM ino;
    {
    INTERN FINODE buf[BUFSIZE / sizeof (INO)];

    if (!getblk(fd, buf, inblk(ino)))
        return (NULL);
    else
        return (ioff(buf, ino);
    }
```

SEE ALSO
ioff

NAME

ioff - get inode offset within block

SYNOPSIS

```
TEXT *ioff(s, ino)
    TEXT *s;
    INUM ino;
```

FUNCTION

ioff locates inode number ino within the 512-byte block at s, assuming the correct block has already been read into s.

RETURNS

ioff returns a pointer to the first byte of the inode.

EXAMPLE

To get an inode:

```
FINODE *getino(fd, ino)
    FILE fd;
    INUM ino;
    {
    INTERN FINODE buf[BUFSIZE / sizeof (INO)];

    if (!getblk(fd, buf, inblk(ino)))
        return (NULL);
    else
        return (ioff(buf, ino);
    }
```

SEE ALSO

inblk

NAME

lsize - get size of a file

SYNOPSIS

```
LONG lsize(pi)
    FINODE *pi;
```

FUNCTION

lsize obtains the size of a file in bytes from the size0 and size1 fields in the inode at pi. The inode is assumed to be in native byte order, such as is returned as part of a stat (or fstat) system call.

RETURNS

lsize returns the size as a long integer.

EXAMPLE

```
IMPORT LONG lsize();

nblocks = lsize(&ps->s_mode) >> 9;
```

NAME

lslin - convert inode information to readable form

SYNOPSIS

```
TEXT *lslin(buf, pnode, grp, atim)
    TEXT *buf;
    FINODE *pnode;
    BOOL grp, atim;
```

FUNCTION

lslin fills a 43 character buffer pointed at by buf with a printable representation of information from the file system inode pointed at by pnode. The inode is assumed to be in native byte order, such as is returned as part of a stat (or fstat) system call.

The first character in the returned buffer is the inode type;

'-' for a plain file
'c' for a character special device
'd' for a directory
'b' for a block special device

The next nine characters of this first field specify the read 'r', write 'w' and execute 'x' permissions for the owner, the owner's group and all others, in that order. A '-' in any position indicates that the associated permission is denied. An 's' in place of an 'x' means: set userid if in the owner field, set groupid if in the group field, save text if in the others field. An 'e' in place of an 'x' means much the same, except that the corresponding execute permission is not present.

The next 3 character field specifies the number of links to the inode, right justified.

A space follows.

The loginid occupies the next 8 character field, left justified. If (grp) then the loginid corresponds to the first password file entry whose groupid matches that of the file; otherwise the loginid is for the first entry whose userid matches. If the loginid is not obtainable from the password file, the above key is printed as a decimal number, left justified in the field.

If the inode is not a special device, the next eight character field specifies the size, in bytes, of the file. Otherwise the device's major and minor inode numbers are printed, right justified and separated by a comma.

In either case, a space follows.

The last field is 12 characters long and contains the month, day and time of the date last accessed, if atim is nonzero; otherwise the date of last update. If this time is more than half a year ago, the time subfield is replaced with the year of last update.

RETURNS

lslin returns buf, whose contents are replaced.

EXAMPLE

```
stat(*av, &fi);
lslin(buf, &p->s_mode, YES);
putfmt("%.43p %p\n", buf, *av);
```

which might print:

```
-r-sr-xr-x  1 root       18678 May 06  1980 /bin/rm
```

SEE ALSO

atime

NAME

ltime - convert system time to local time

SYNOPSIS

```
TVEC *ltime(pv, lt)
    TVEC *pv;
    LONG lt;
```

FUNCTION

ltime converts lt, the number of seconds elapsed since 00:00 Jan 1 1970, to a structure at pv giving the time components in local time.

The timezone is determined from the timezone file, if it can be read; it will be read at most once. Otherwise, the default internal values are used, which are correct for Secaucus NJ. Timezone information is available in the external variables:

```
BOOL  _dst {YES};       /* non-zero enables daylight savings */
BYTES _timezone {5};    /* hours west of Greenwich */
BYTES _tzmins {0};      /* minutes west of standard */
TEXT *_edt {"EDT"};
TEXT *_est {"EST"};
```

The initial values shown are the defaults. Note that fractional timezones, such as for Surinam, are supported.

If (12 <= _timezone) ltime subtracts 24 for use in its computation. Thus to get the correct time east of Greenwich, the timezone should be set to 24 minus the number of hours east of Greenwich. The timezone for Japan is 15 (24-9).

RETURNS

ltime returns the pointer pv and fills in the structure at pv.

EXAMPLE

To print the date:

```
IMPORT LONG time();
IMPORT TEXT *_est, *_edt;
TVEC tvec;
INTERN TEXT buf[] {"012345678901234567890123 "};

ltime(&tvec, time(NULL));
putstr(STDOUT, atime(&tvec, buf), tvec.dstf ? _edt : _est,
    "\n", NULL);
```

FILES

/adm/zone for the timezone.

SEE ALSO

atime, vtime

BUGS

This will fail after Feb 28, 2100.

NAME

mapblk - map logical block to physical

SYNOPSIS

```
BLOCK mapblk(fd, pi, lbn)
    FILE fd;
    FINODE *pi;
    BLOCK lbn;
```

FUNCTION

mapblk determines the physical block number corresponding to the logical block number lbn in the file whose inode is at pi. It reads indirect blocks as necessary from the filesystem under control of fd.

RETURNS

mapblk returns the physical block number, if present. Zero is returned if lbn is off the end or inside a hole in the file.

EXAMPLE

```
    n = lsize(&ino) + 0777 >> 9;
    for (lbn = 0; lbn < n; ++lbn)
        {
        if (pbn = mapblk(fd, &ino, lbn)
            getblk(fd, buf, pbn);
        else
            clrbuf(buf);
        write(STDOUT, buf, BUFSIZE);
        }
```

SEE ALSO

getblk, putblk

NAME

mesg - turn on or off messages to current terminal

SYNOPSIS

```
BOOL mesg(new)
    BOOL new;
```

FUNCTION

mesg allows other users to write to the current terminal when new is non-zero; otherwise it prevents other users from writing to this terminal.

RETURNS

mesg returns YES if the terminal had write permission when the function was called.

EXAMPLE

```
old = mesg(NO);
print();
mesg(old);
```

SEE ALSO

devname

NAME

mkdir - make a directory

SYNOPSIS

```
ERROR mkdir(dname)
    TEXT *dname;
```

FUNCTION

mkdir makes the directory dname, with two entries, ., linking dname to itself, and .., linking dname to its parent. mkdir can only be used by the superuser.

RETURNS

mkdir returns 0 if successful, else a negative number which is the Idris error code negated.

EXAMPLE

```
if (mkdir(name) < 0)
    putstr(STDERR, "can't make directory: ", name, "\n", NULL);
```

NAME

mv - move a file

SYNOPSIS

```
ERROR mv(old, new)
    TEXT *old, *new;
```

FUNCTION

mv creates the link new to a file and removes the link old. If a file named new already exists, it is removed. If old and new are on different filesystems and old is a plain file mv will copy old to new; otherwise, if old is a character or block special device, mv will make the appropriate device; in either case, old is removed. A directory cannot be moved to another filesystem, or to a subtree of itself.

RETURNS

mv returns zero if successful, else a negative number, which is the Idris error return code, negated.

EXAMPLE

```
if (mv("a.out", "xeq") < 0)
    putstr(STDERR, "can't move a.out\n", NULL);
```

NAME
parent - get parent name of a file

SYNOPSIS
```
TEXT *parent(buf, path)
    TEXT *buf, *path;
```

FUNCTION
parent creates a NUL terminated string at buf which is the name of the parent of the file named path. The parent is created by a) stripping off trailing '/' characters; b) stripping off trailing "/." strings; c) stripping off and remembering trailing "/.." strings; d) repeating a-c until there are no more to be stripped; e) partitioning the remainder into LEFT '/' RIGHT, where LEFT is NULL if there is not at least one slash, and RIGHT contains no slash. f) determining the "base" parent by the following table:

LEFT	SLASH	RIGHT	PARENT
none	no	none	.. (or / if path started with a /)
none	no	.	..
none	no	..	../..
none	no	other	.
none	yes	any	/
any	yes	any	<LEFT>

g) adding back the trailing "/.." strings that were stripped in c. h) stripping off leading "./" strings;

For example:

FILENAME	PARENT
abc	.
x/y	x
/a	/
a/b/..	a/..
../c	..
.	..

Note that parent is sufficiently cautious about writing into buf, that parent(name, name) is meaningful, and will work as expected.

RETURNS
parent returns buf.

EXAMPLE
```
stat(parent(buf, path), &pstat);
if (!perm(&pstat, 02))
    putstr(STDERR, "can't remove ", path, "\n", NULL);
```

NAME

perm - test permissions of a file

SYNOPSIS

```
BOOL perm(pst, mask)
    STAT *pst;
    COUNT mask;
```

FUNCTION

perm determines if the current process has all of the permissions requested by mask, for the file whose status, as returned by stat, is pointed at by pst. mask is the inclusive or of 04, to check read permission, 02, to check write permission, and 01, to check execute permission. The permissions are checked according to the real userid and groupid of the process. This means that perm may be called by set-userid programs to test whether the invoker of the program has the required permissions. If the userid of the file is the same as the real userid then the "user access" bits are checked; otherwise, if the groupid of the file matches the real groupid, the "group access" bits are checked; otherwise, the "other access" bits are checked.

RETURNS

perm returns YES if the file has all of the requested permissions. If the user is the superuser, perm returns YES, unless mask requests execute permission and none of the execute permissions are turned on for the file.

EXAMPLE

```
if (!perm(&filestat, 02))
    putstr(STDERR, "can't write\n", NULL);
```

NAME

putblk - put filesystem block

SYNOPSIS

```
BOOL putblk(fd, buf, bno)
    FILE fd;
    TEXT *buf;
    BLOCK bno;
```

FUNCTION

putblk writes the 512-byte block whose number is bno from buf to the file under control of fd. If the write fails, the function returns NO.

RETURNS

putblk returns YES if it can write the whole block else NO.

EXAMPLE

```
if (!putblk(fd, superbuf, 1))
    error("can't write superblock", NULL);
```

SEE ALSO

getblk, mapblk

NAME

puti - put inode to filesystem

SYNOPSIS

```
VOID puti(fd, pi, ino)
    FILE fd;
    FINODE *pi;
    INUM ino;
```

FUNCTION

puti writes the in-core inode pointed at by pi to the filesystem controlled by fd, after converting the inode to filesystem format. puti shares an in-core buffer of 16 inodes (1 block) with geti, and pi is actually just copied and converted into this buffer. Thus, flushi must be called before closing fd, to insure that any pending output is completed.

RETURNS

If there is pending output for another inode block, and it cannot be written, the writerr condition is raised. If the appropriate inode block is not in the buffer shared between geti and puti and it cannot be read, the readerr condition is raised. In any case, there is no useful return value from puti.

EXAMPLE

To process the entire inode list of a filesystem:

```
for (i = isize << 4; 1 <= i; --i)
    if (process(pi = geti(fd, &ibuf, i)))
        puti(fd, pi, i);
flushi(fd);
```

SEE ALSO

flushi, geti

NAME

rdir - read directory on unmounted filesystem

SYNOPSIS

```
DIR *rdir(fd, pi, lno)
    FILE fd;
    FINODE *pi;
    BYTES lno;
```

FUNCTION

rdir obtains the directory entry whose ordinal position is lno within the directory, counting from zero, by reading the unmounted filesystem at fd. It is assumed that pi points at the directory's inode, in native byte order. If lno is zero, or if the block differs from that remembered on the last call to rdir, a new block is obtained by reading from the file under control of fd.

Note well that this function is designed for sequential processing of one directory at a time.

RETURNS

rdir returns a pointer to the link within its current block in memory. If the list is off the end of the directory, rdir returns NULL.

EXAMPLE

```
for (i = 0; pd = rdir(fd, &ino, i); ++i)
    process(pd);
```

SEE ALSO

wdir

NAME

rmdir - remove a directory

SYNOPSIS

```
COUNT rmdir(dname)
    TEXT *dname;
```

FUNCTION

rmdir removes an empty directory, (i.e., one with at most one . entry and at most one .. entry). Directories can only be removed by the superuser.

RETURNS

rmdir returns zero if successful; otherwise: -ENOENT if dname does not exist, -ENOTDIR if dname is not a directory, -EISDIR if directory is not empty, or -EPERM if user does not have write permission for the parent directory.

EXAMPLE

```
if (rmdir(fname) < 0)
    putstr(STDERR, "can't remove ", fname, "\n", NULL);
```

NAME

shell - execute a shell command escape

SYNOPSIS

```
COUNT shell(cmd, fname, flags)
    TEXT *cmd, *fname;
    COUNT flags;
```

FUNCTION

shell invokes the shell command interpreter to parse and execute cmd. It is most often used within programs offering a "! cmd" shell escape or within programs offering a "-c cmd" flag.

Any trailing '\n' before the trailing NUL on cmd is ignored. If fname is not NULL, any occurrence of the sequence "\f" within cmd will be replaced by the string fname.

If the first 3 characters of cmd are "cd\ " then a chdir system call is done with the rest of cmd as an argument. This will affect the caller.

If the first 3 characters of cmd are not "cd\ " then the shell command interpreter is invoked as "sh -c cmd" under the current execution path. If (!(flags & 3)) cmd is invoked as a new process; shell will wait until the command has completed and will return its status to the calling program. If (flags & 1) cmd is invoked as a new process and shell will not wait, but will return the processid of the child. If (flags & 2) cmd is invoked in place of the current process, whose image is forever gone. In this case, shell will never return to the caller.

To the value of flags may be added a 4 if the processing of interrupt and quit signals for cmd is to revert to system handling. They are normally turned off in both the invoker and any new process for the duration of cmd. The value of flags may also be incremented by 8 if the effective userid is to be made the real userid before cmd is executed.

RETURNS

If a chdir is requested, shell will return what chdir does. If cmd cannot be invoked, shell will return failure; If (!(flags & 3)) shell returns YES if the command executed successfully, otherwise NO; if (flags & 1) shell returns the id of the child process, if one exists, otherwise zero; if (flags & 2) shell will never return to the caller.

In all cases, if cmd cannot be executed, an appropriate error message is written to STDERR.

SEE ALSO

xecl, xecv

NAME

vtime - convert system time to Greenwich Mean Time

SYNOPSIS

```
TVEC *vtime(pv, lt)
    TVEC *pv;
    LONG lt;
```

FUNCTION

vtime converts lt, the number of seconds elapsed since 00:00 Jan 1 1970, to a structure at pv giving the time components in GMT.

RETURNS

vtime returns the pointer pv and fills in the structure at pv.

EXAMPLE

To print the date:

```
IMPORT LONG time();
TVEC tvec;
INTERN TEXT buf[] {"01234567890123456789012 3 GMT\n"};

putstr(STDOUT, atime(vtime(&tvec, time(NULL)), buf), NULL);
```

SEE ALSO

atime, ltime

NAME

wdir - write directory to unmounted filesystem

SYNOPSIS

```
VOID wdir(fd, pi)
    FILE fd;
    FINODE *pi;
```

FUNCTION

wdir writes the last block obtained by rdir to the unmounted filesystem under control of fd. It is presumed that pi points at the inode that rdir has been reading.

Note well that this function is designed for sequential processing of one directory at a time, in conjunction with rdir.

RETURNS

Nothing.

EXAMPLE

```
for (i = 0; pd = rdir(fd, &ino, i); ++i)
    if (badlink(pd))
        {
        fixlink(pd);
        wdir(fd, &ino);
        }
```

SEE ALSO

rdir

NAME

who - read and sort who file

SYNOPSIS

```
WHO *who(n, fname)
    COUNT *n;
    TEXT *fname;
```

FUNCTION

who allocates a buffer large enough to hold the specified file, reads the file contents into it, sorts the entries by tty name, then sets the integer at pn to the number of non-null entries found. The file is presumed to have records like the standard who and log files.

RETURNS

who returns a pointer to the allocated buffer if successful, otherwise NULL.

EXAMPLE

To print the names of the current system users:

```
IMPORT WHO *who();
WHO *pwho, *p;
COUNT n;

pwho = who(&n, WHOFILE);
for (p = pwho; 0 <= --n; ++pwho)
    putfmt("%b\n", p->w_uname, sizeof (p->w_uname));
free(pwho, NULL);
```