CLIB/CLIBT Library

April 1991

by Carl Kreider

Docs: Mark Griffith

CLIB/CLIBT ……..…… xx/xx/90 ……..…… Carl Kreider
Mark Griffith
Conversion to DOC …… 05/xx/07 ……..…… Dean Leiber

Further Enhancement..... 04/xx/15 .....Bill Pierce

Foreward

The original CLIB/CLIBT Documentation was converted to a more modern format to facilitate the use of the C compiler on OS-9/Nitros-9. The basic text and order of the original documentation was not changed, but there were some alterations made to allow for a common format structure.

The following additions were made to the document:

- Addenums/Caveats for Nitros-9 (if any)
- The Table of Contents
- Appendices

Please note that any addresses fn the Introduction are hopelessly out of date. For errors/corrections in this document, please contact:

Dean Leiber (adit@nationsdial.com)

Tables of Contents

Kreider OS 9 C Library

INTRODUCTION

Kreider "C" Library Documents

Enclosed are the docs for the full Kreider Library. In December 1987 I released the docs for those functions that were added to the standard "C" library by Carls library. This archives contains machine readable copy for all the functions. Each file is formatted for MROFF output, and scripts are included to allow the user to make printed docs or neat manual pages for online reading.

TO PRINT THE DOCS

Get your printer ready with lots of paper. The docs print on 147 pages. Run the shell script "printdocs" and get some lunch, and maybe dinner too! You can change the script to redirect output to a disk file if you like.

TO MAKE MANUAL PAGES

Run the script "make.mans". If you don't have a directory called MAN under the directory where the mroff source files are, then make one first. Again, go get some shopping done, or clean the celler/garage.

MROFF CHANGES

A new version of MROFF is included in this archive. Basically, all it does is add a few extra commands to be slightly more compatible with UNIX nroff. In fact, the mroff source files will print nicely on a UNIX system using nroff or troff without any changes.

MAN UTILITY

A man utility is included with the archive to enable those without such a beast to have online manual printing. This is a very basic man utility, written by Pete Lyall and seriously hacked apart by me. Everything is hardcoded for my system. Sorry. You'll have to change the source and re-compile to use it.

Along with the man utility are two files, "man.index" and "man.help". Man.index is an index (really!!?? - grin) of the library manuals. MAN tried to find the page depending on what you give it on the command line. If it can't, it looks into the index to try and get the right page. It's simple, fast, and works pretty well. "Man.help" is a simple file to allow you to quickly find the function you are looking for by giving it a keyword to search for. Again, it is simple, fast, and reasonably useful.

Mark Griffith
DeLand, Fl.

CIS 76070,41
UUCP !uflorida!ki4pv!macs!mdg
or

!uflorida!ki4pv!macs!stetson!rewop!sysop
Internet griffith@stetson.edu
BITNET GRIFFITH@STETSON

abort() Stop the program and produce a core dump

SYNOPSIS:

abort()

DESCRIPTION:

This call causes a memory image to be written out to the file ***core*** in the current directory, and then the program exits with a status of 1.

abs() Interger absolute value

SYNOPSIS:

abs(i)
int i;

DESCRIPTION:

The **abs()** function returns the absolute value of its integer operand.

CAVEATS:

Applying the **abs()** function to the most negative integer generates a result which is the most negative integer. That is, **abs(0x80000000)** returns 0x80000000 as a result.

access() Give file accessibility

SYNOPSIS:

#include <modes.h>

access(fname, perm)
char *fname;
int perm;

DESCRIPTION:

Access() returns zero if the access modes specified in ***perm*** are correct for the user to access ***fname***. A -1 is returned if the file cannot be accessed.

The value for ***perm*** may be any legal OS-9 mode as used for **open()** or **creat()**,or, it may be zero which then tests whether or not the file exists or the path to it may be searched.

Access() is useful to test the existence of a file without actually opening the file as would **open()** or **fopen()**, thereby changing the user permissions.

CAVEATS:

The values for ***perm*** are NOT compatible with other non-OS-9 systems.

DIAGNOSTICS:

The returned error number, if a value of -1 is returned from this call, will be found in the global variable ***errno***, and will indicate the reason that file cannot be accessed.

asetuid() Sets the user ID number

SYNOPSIS:

asetuid(uid)

DESCRIPTION:

Asetuid() sets the user ID number for the current task. Unlike **setuid()**, this call can be used even if the user is not the Super User.

SEE ALSO:

setuid(), getuid (Microware Manual)

atof(), atoi(), atoll() Convert ASCII to numbers

SYNOPSIS:

```
#include <math.h>

double atof(ptr)
char *ptr;

long atoi(ptr)
char *ptr;

int atol(ptr)
char *ptr;
```

DESCRIPTION:

These functions convert a string pointed to by to double, long, and integer representation respectively. Any leading whitespace (space, tab, or newline) is ignored. The first unrecognized character ends the string.

Atof() recognizes (in order), an optional sign, an optional string of spaces, a string of digits optionally containing a radix character, an optional `e' or `E', and then an optionally signed integer, as in the example below:

"-1234.5678e+9"

Numbers up to the decimal point are assumed to be the integer portion of the number.

The **atoi()** and **atoll()** functions recognize (in order), an optional string of spaces, an optional sign, then a string of digits.

CAVEATS:

Overflow causes unpredictable results. There are no error indications returned by these functions.

bsearch() Binary search function

SYNOPSIS:

```
char *bsearch(key, base, nel, width, compar)
char *key, *base;
int nel, width;
int (*compar) ();
```

DESCRIPTION:

This function performs a binary search on already sorted arrays of strings, finding the string matching ***key***, in the array of strings, beginning at the memory location pointed to by ***base***. The array MUST have been previously sorted, in ascending order, according to the comparison function ***compar()***. The total number of elements in the array is contained in ***nel***, and the width of each string in the array, which must all be the same length, is held in ***width***.

The function ***compar()*** is any user supplied function that will return whether the first argument is greater than, equal to, or less than, the second argument.

Strcmp() would be a good choice for string variables.

DIAGNOSTICS:

Bsearch() returns a pointer to the matching string upon success, or null.

SEE ALSO:

Strcmp()

chain() Load and execute a new program module

SYNOPSIS:

chain(modname, paramsize, paramptr, type, lang, datasize)
char *modname, *paramptr;
int paramsize, type, lang, datasize;

DESCRIPTION:

The action of the F$CHAIN system call is described fully in the OS-9 documentation.

Chain() implements this service request as described there with one important exception: chain will NEVER return to the caller. If there is an error, the chained process will abort and return to its parent process. It might be wise, therefore, for the program to check the existence and access permissions of the module before calling chain. Permissions may be checked by using **modlink()** or **modload()** followed by **munlink()**.

Modname should point to the name of the desired module. ***Paramsize*** is the length of the parameter string (which should be terminated with an '\\n'), and ***paramptr*** points to this parameter string. ***Type*** is the module type as found in the module header (normally a 1 for a program module), and ***lang*** should match the language nibble in the module header (C programs have a 1 for machine language). ***Datasize*** my be zero, or contain the number of 256 byte pages to give to the new process as its initial data memory allocation.

CAVEATS:

The variable ***paramsize*** should never be zero. If only a carriage return command line terminator (\\n) is used, ***paramsize*** should be set to one.

SEE ALSO:

os9fork(), os9 F$CHAIN system call

chdir(), chxdir() Change directories

SYNOPSIS:

chdir(dirname)
char *dirname;

chxdir(dirname)
char *dirname;

DESCRIPTION:

These calls change the current data and execution directories respectively. This change is only

current as long as the process is running. ***Dirname*** is a pointer to the directory pathlist string.

DIAGNOSITCS:

Each call returns zero after a successful call, or -1 if ***dirname*** is not a directory or its path is not searchable.

SEE ALSO:

Shell commands chd and chx (cd, cx)

chmod() Change access permissions of a file

SYNOPSIS:

```
#include <modes.h>

chmod(fname, perm)
char *fname;
int perm;
```

DESCRIPTION:

Chmod() changes the permission bits in the path descriptor associated with a file. ***Fname*** must be a pointer to a file name, and ***perm*** should contain the desired access mode number.

The allowable access numbers are defined in the file **<modes.h>** and are as follows:

```
/* File Modes */

#define  S_IFMT    0xff      /* mask for file type */
#define  S_IFDIR   0x80      /* directory */

/* Permissions */
.sp
#define  S_IPRM    0xff       /* mask for permission bits */
#define  S_IREAD   0x01       /* owner read */
#define  S_IWRITE  0x02       /* owner write */
#define  S_IEXEC   0x04       /* owner execute */
#define  S_IOREAD  0x08       /* public read */
#define  S_IOWRITE 0x10       /* public write */
#define  S_IOEXEC  0x20       /* public execute */
#define  S_ISHARE  0x40       /* sharable */
```

Only the owner or the super user may change the permissions of a file.

DIAGNOSTICS:

A successful call returns a null. A -1 is returned if the caller is not entitled to change permissions on that file, or ***fname*** cannot be found.

SEE ALSO:

OS-9 command attr

chown() Change the ownership of a file

SYNOPSIS:

chown(fname, ownerid)
char *fname;
int ownerid;

DESCRIPTION:

Chown() changes the ownership of a file by changing the owner ID associated with the file in the file descriptor. Only the super user has access to this call.

Fname is a pointer to a the file name that is to be changes. ***Ownerid*** is the user ID of the new file owner.

DIAGNOSTICS:

A zero is returned from a successful call. -1 is returned on error.

CLIBT.L Transcentental Math C Library

The following functions are additions to the Kreider CLIB.L functions described earlier, and all the functions contained in that library are also present here. The purpose of this additional library is to provide the transcentental math functions. All the remaining functions perform in exactly the same manner as in the CLIB.L library, but are rewritten here for increased speed in execution, although resulting in a larger output file.

SYNOPSIS:

rad()

deg()

double acos(x)
double x;

double asin(x)

```
double x;

double atan(x)
double x;

double cos(x)
double x;

double sin(x)
double x;

double tan(x)
double x;

double acosh(x)
double x;

double asinh(x)
double x;

double atanh(x)
double x;

double cosh(x)
double x;

double sinh(x)
double x;

double tanh(x)
double x;

double pow(x,y)
double x,y;

double exp(x)
double x;

double antilg(x)
double x;

double log10(x)
double x;

double log(x)
double x;

double sqrt(x)
double x;
```

```
double sqr(x)
double x;

double inv(x)
double x;

double dabs(x)
double x;

double dexp(x,i)
double x;
int i;
```

DESCRIPTION:

The various transcendental math functions are implemented here using the **CORDIC** method. Accuracy is to sixteen (16) decimal places. The four basic math functions are rewritten to optimize for speed at the expense of output program length. This gives at least a factor of two speed improvement over the standard Microware C library.

Externally, all number look just like a normal C double, but internally, an extra byte is used to permit an exponent range of 511.

Rad() changes the trigometric functions to radians.

Deg()changes the trigometric functions to degrees.

Acos(**) returns the arc cosine of ***x*.

Asin() returns the arc sine of ***x***.

Atan() returns the arc tangent of ***x***.

Cos(**) returns the cosine of ***x*.

Sin(**) returns the sine of ***x*.

Tan() returns the tangent of ***x***.

Acosh() returns the arc hyperbolic cosine of ***x***.

Asinh() returns the arc hyperbolic sine of ***x***.

Atanh(**) returns the arc hyperbolic tangent of ***x*.

Cosh(**) returns the hyperbolic cosine of ***x*.

Sinh() returns the hyperbolic sine of ***x***.

Tanh() returns the hyperbolic tangent of ***x***.

Pow() returns the value of ***x*** taken to the power of ***y***.

Exp() returns ***E*** to the ***x*** power.

Antilg() returns 10 to the ***x*** power.

Log10() returns logarithm base 10 of ***x***.

Log() returns the logarithm base ***E*** of ***x***.

Sqrt() returns the square root of ***x***.

Sqr() returns the square of ***x***.

Inv() returns the value of 1 (one) divided by ***x***.

Dabs() returns the absolute value of ***x***.

Dexp() returns the value of ***x*** multiplied by 2 to the ***I*** power.
This is a VERY QUICK function.

DIAGNOSTICS:

The following errors are returned by each of the above functions if an error occurs and are placed in the global variable ***errno***.

These error numbers should be added to ERRNO.H:

EFPOVR 40	Floating point overflow
EDIVERR 41	Divide by zero error
EINTERR 42	Overflow on conversion of a double integer
EFPUND 43	Floating point underflow (does not abort the program). Zero is returned.
EILLARG 44	Illegal function argument, e.g **sqrt(-1)**

SEE ALSO:

math()

close() Close a file

SYNOPSIS:

close(pn)
int pn;

DESCRIPTION:

Close() closes an already opened file as described by the path number ***pn***, which is the path number returned from an **open()**, **creat()**, **create()** or **dup()** call.

Termination of a process always closes all opened files automatically, but it is necessary to close files where multiple files are opened by the process and it is desired to re-use the path numbers to avoid exceeding the process or system path number limit.

DIAGNOSTICS:

This call does not return anything.

SEE ALSO:

creat(), create(), open(), dup().

toupper(), tolower(), _toupper() _tolower(), toascii() Translate characters

SYNOPSIS:

```
#include <ctype.h>

toupper (c)
int c;

tolower (c)
int c;

_toupper (c)
int c;

_tolower (c)
int c;

toascii (c)
int c;
```

DESCRIPTION:

The functions **toupper()** and **tolower()** have as domain the range of getc(), which are the ASCII characters from -1 through 255. If the argument to **toupper()** represents a lower case letter, the result is the corresponding upper case letter. If the argument to **tolower()** represents an upper case letter, the result is the corresponding lower case letter.

The macros **_toupper()** and **_tolower()** accomplish the same thing as **toupper()** and **tolower()**, except they are faster and are restricted to ASCII characters (for example, -1 to 127). The macro **_toupper()** requires a lowercase letter as its argument; its result is the corresponding uppercase letter. The macro **_tolower()** requires an uppercase letter as its argument; its result is the corresponding lowercase letter.

The macro **toascii()** yields its argument with all bits turned off that are not part of a standard ASCII character set, i.e., the MSB of that character is set to 0. It is intended for compatibility with other systems.

CAVEATS:

Any arguments to the macros **_touuper()**, **_tolower()**, or **toascii()** outside the ranges specified will yield garbage results.

SEE ALSO:

ctype, getc()

crc() compute the cyclic redundancy count of a module

SYNOPSIS:

```
crc(start, count, accum)
char *start;
int count;
char accum[3];
```

DESCRIPTION:

This call accumulates a crc into a three byte array at ***accum[3]*** for the ***count*** bytes starting at ***start***. All three bytes of ***accum*** should be initialized to 0xFF before the first call to **CRC**.

However, repeated calls can be subsequently made to cover an entire module. If the result is to be used as an OS-9 module crc, it should have its bytes complemented before insertion at the end of the module. An example follows:

```
/*
**  Calculate a module's CRC and insert it at the end.
**
**  The pointer passed in mod_desc is address of
**  the beginning of the module already in memory.
*/

#include <module.h>

chg_crc(mod_desc)
char *mod_desc;
{
        int count;
        char accum[3];
        char *old_crc;

        old_crc = (char *) mod_desc + mod_desc->m_size - 3;
        count = mod_desc->msize - 3;

        accum[0] = 0xff;
        accum[1] = 0xff;
        accum[2] = 0xff;

         crc (mod_desc, count, accum);

        *old_crc++ = ~accum[0];
        *old_crc++ = ~accum[1];
        *old_crc = ~accum[2];
}
```

creat() Create a new file

SYNOPSIS:

```
#include <modes.h>

creat(fname, perms)
char *fname;
int perms;
```

DESCRIPTION:

Creat() returns a path number to a new file available for writing, giving it the permissions specified in the ***perm*** variable and making the process user the owner of the file. If, however, ***fname*** is the name of an already existing file, the file is truncated to zero length and the ownership and permissions remain unchanged. Note, that unlike the OS-9 assembler service request, **Creat()** DOES NOT return an error if the file already exists. **Access()** should be used to establish the existence of the file if it is important that a file should not be overwritten. It is unnecessary to

specify writing permissions in ***perm*** in order to write to the file in the current process.

The following permissions are defined in the include file **<modes.h>** as follows:

```
/* File Modes */

#define  S_IFMT     0xff     /* mask for type of file */
#define  S_IFDIR    0x80   /* directory */

/* Permissions */

#define  S_IPRM     0xff       /* mask for permission bits */
#define  S_IREAD    0x01     /* owner read */
#define  S_IWRITE   0x02    /* owner write */
#define  S_IEXEC    0x04    /* owner execute */
#define  S_IOREAD   0x08  /* public read */
#define  S_IOWRITE  0x10  /* public write */
#define  S_IOEXEC   0x20  /* public execute */
#define  S_ISHARE   0x40  /* sharable */
```

Directories may not be created with this call -- use **mknod()** instead.

DIAGNOSTICS:

This call returns -1 if there are too many files opened, if the pathname cannot be searched, if permission to write is denied, or of the file already exists and **IS A DIRECTORY**.

CAVEATS:

File permissions that specify either owner or public executable files will cause the new file to be created in the current execution directory. All other permissions will cause the file to be created in the current data directory. To create an executable file in the current data directory, see **create()**.

SEE ALSO:

create(), write(), close(), chmod()

create() Creates and opens a file

SYNOPSIS:

```
#include <modes.h>

create(fname, mode, pmode)
char *fname;
int mode, pmode;
```

ocreat(fname, mode, pmode)
char *fname;
int mode, pmode;

DESCRIPTION:

Create() creates and opens the file named ***fname***. This call accepts the file mode and access permissions in the same function and is useful in setting up user permissions as soon as the file is opened.

This function returns -1 if the file already exists, or the path number if the file is successfully created.

Ocreat() performs the same function, except it deletes the old file if it already exists when **Ocreat()** is called.

SEE ALSO:

creat(), open(), fopen().

ctime(), localtime(), gmtime() asctime(), tzset() — Convert date and time

SYNOPSIS:

#include <utime.h>

*ctime (clock)
long *clock;

*localtime (clock)
long *clock;

*asctime (tm)
struct tm *tm;

extern long timezone; /* used here for compatibility only */

extern int daylight;

void tzset()

DESCRIPTION:

Ctime() converts a long integer, usually returned from **time()**, or pointed to by ***clock***, representing the time in seconds since 00:00:00, January 1, 1970, and returns a pointer to a 26-character string in the following form:

Sun Sep 16 01:03:52 1973\\n\\0

The **time()** function is ideally suited to return the long integer time value.

Localtime() returns a pointer to the ***tm*** structure.

Asctime() converts a ***tm*** structure to a 26-character string, as shown in the above example, and returns a pointer to the string.

For user convience, declarations of all the functions and the externals, and the ***tm*** structure, are provided in the **<utime.h>** header file, shown on the next page.

The external long variable ***timezone*** is always zero.

EXAMPLE:

To print the current time:

```
long curr_time;
curr_time = time ((long *)0);
printf ("The time is: %s", ctime(&curr_time));
```

BUGS:

The return values point to static data whose content is overwritten by each call.

SEE ALSO:

time(), o2utime(), u2otime()

```
/*
**      Utime.h
*/

struct tm {
        int   tm_sec;        /* seconds (0 - 59)        */
        int   tm_min;        /* minutes (0 - 59)        */
        int   tm_hour;       /* hours   (0 - 23)        */
        int   tm_mday;       /* day of month (1 - 31)   */
        int   tm_mon;        /* month of year (0 - 11)  */
        int   tm_year;       /* year ( year - 1900)     */
        int   tm_wday;       /* day of week (Sunday = 0) */
        int   tm_yday;       /* day of year (0 - 365)   */
        int   tm_isdst;      /* NOT USED                */
    };

        long time();                /* Same as UNIX */
        struct tm  *localtime(); /* Same as UNIX */
        char  *asctime();        /* Same as UNIX */
```

```
char  *ctime();     /       * Same as UNIX */

long  o2utime();          /* Convert OS9 style buf to UNIX long */
/* void */  u2otime();    /* Convert 'tm' to OS9 char *buf */
```

isalpha(), isupper(), islower() Character classification macros isdigit(), isalnum(), isspace() ispunct(), isprint(), iscntrl(), isascii()

SYNOPSIS:

#include <ctype.h>

isalpha(c)

etc....

DESCRIPTION:

These macros classify character-coded integer values according to their ascii value using fast table look-up.

All macros return non-zero for true and zero for false.

The macro **isascii()** provides a correct result for all integer values. The rest provide a result for EOF and values in the character range outlined in the table below, however, the result will be unpredictable for characters outside the range -1 to 127.

Isalpha() ***c*** is a letter

Isupper() ***c*** is an uppercase letter

Islower() ***c*** is a lowercase letter

isdigit() ***c*** is a digit

isalnum() ***c*** is an alphanumeric character

isspace() ***c*** is a space, tab, carriage return, new line,or formfeed

ispunct() ***c*** is a punctuation character (neither control, alphanumeric, nor a space)

isprint() ***c*** is a printable character, code 32 (space) through 126 (tilde)

iscntrl() ***c*** is a delete character (127) or ordinary control character (less than 32) except for space characters

isascii ***c*** is an ASCII character, code less than 128

SEE ALSO:

toascii(), toupper(), tolower(), _toupper(), _tolower()

datlink(), dunlink() unlkdata(), lockdata() — Data module operations

SYNOPSIS:

```
int datlink(name, datptr, space)
char *name, *datptr;
int *space;

int dunlink(datptr)
char *datptr;

int lockdata(datptr)
char *datprt;

int unlkdata(datptr)
char *datptr;
```

DESCRIPTION:

Datlink() loads (if necessary) and links the file ***name***. ***Datptr*** is set to the address of the data section. ***Space*** is set to the free space available. **Datlink()** returns (-1) if an error, null for a successful link, and 1 if a load was required. If a 1 is returned, that means you are the first user of that data module.

Dunlink() unlinks the module belonging to ***datptr***. A (-1) is returned if an error occurs, or a null if no error.

Lockdata() attempts to lock the data module (datptr is considered to be the lock byte) by changing the lock byte, which is normally -1, to a null. **Lockdata()** returns a process stack level on failure, or a null. ***Errno*** is not set.

Unlkdata() unlocks the data module. It returns a (-1) on any attempt to unlock a module that has not already been locked. A null is returned upon success.

NOTE:

A data module is considered locked when it is loaded. It must be set for use by a call to **Unlkdata()** after the original loader is finished with any initialization required. A user can determine if they are the original owner by the value returned during the **datlink()** call.

In all cases, ***datptr*** points to the lock byte. User free space begins at ***datptr***+ 1.

devtyp(), isatty() Check the type of a device

SYNOPSIS:

devtyp(pn)
int pn;

isatty(pn)
int pn;

DESCRIPTION:

Devtyp() returns an integer number corresponding to the device type as defined by OS9. ***Pn*** is the OS9 path number opened to the device to check.

The following are the device types and returned values:

0 = SCF
1 = RBF
2 = PIPE
3 = SBF

Isatty() functions in the same manner as ***devtype***, but returns **TRUE** if the device is an **SCF** type and **FALSE** if it is not.

NOTE:

FOR COCO USERS:
These functions do not return any different values for a Level II device window since they are defined as SCF type devices.

opendir(), readdir(), telldir() Directory operations
rewinddir(), seekdir(), closedir()

SYNOPSIS:

#include <dir.h>

*opendir(dirname)
char *dirname;

*readdir(dirp)
DIR *dirp;

telldir(dirp)
DIR *dirp;

```
rewinddir(dirp)
DIR *dirp;

seekdir(dirp, loc)
DIR *dirp;
long loc;

closedir(dirp)
DIR *dirp;
```

DESCRIPTION:

Opendir() opens the specified directory and associates a directory stream with it. **Opendir()** returns a pointer to be used to identify the directory stream in subsequent operations. The pointer null is returned if ***dirname*** cannot be accessed or of it cannot "malloc" enough memory to hold the whole thing. Directory sectors are NOT buffered.

Readdir() returns a pointer to a structure containing the next directory entry, unless the entry is a deleted file in which case the next entry is returned. It returns null upon reaching the end of the directory or detecting an invalid **seekdir()** operation.

Telldir() returns the current location associated with the named directory stream. Values returned by **telldir()** are valid only for the lifetime of the associated dir pointer. If the directory is closed and then reopened, the **telldir()** value may be invalidated. It is safe to use a previous **telldir()** value immediately after a call to **opendir()** and before any calls to **readdir().**

Rewinddir() resets the position of the named directory stream to the beginning of the directory.

This function is implemented as a macro in **<dir.h>.**

Seekdir() sets the position of the next **readdir()** operation in the directory stream. The new position reverts to the one associated with the directory stream when the **telldir()** operation was performed. **Closedir()** closes the named directory stream and frees the structure associated with ***dirp***.

NOTE:

For user convenience, function declarations are made in the header **<dir.h>** below.

```
/*
**      Dir.h
/*

struct direct {
        long d_addr;          /* file descriptor address */
        char d_name[30];   /* directory entry name    */
        };
```

```
typedef struct {
        int   dd_fd;           /* fd for open directory   */
        char  dd_buf[32];  /* a one entry buffer      */
        } DIR;

#define DIRECT struct direct
#define rewinddir(a)  seekdir(a, 0L)

extern DIR *opendir();
extern DIRECT *readdir();
extern long telldir();
extern  /* void */ seekdir(), closedir();
```

_dump() Dumps memory to standard output

SYNOPSIS:

```
void _dump(s, addr, count)
int count;
char *s, *addr;
```

DESCRIPTION:

_dump() is used mostly as a debugging function. It prints the title ***s*** and then, starting at the memory address pointed to by ***addr***, dumps at the most ***count*** bytes to standard output.

NOTE:

Care must be taken to insure the variable ***count*** is not too large, else the memory dump will take a very long time. Also, the dump is formatted similar to the standard OS9 dump utility.

dup() Duplicate an open path number

SYNOPSIS:

```
dup(pn)
int pn;
```

DESCRIPTION:

Dup() takes the path number ***pn*** as returned from an **open(), creat()**, or **create()** call and returns another path number associated with the same file.

Dup() is often used to duplicate the standard paths (stdin, stdout, stderr) prior to forking a new process. The new process will then have these paths for its use.

EXAMPLE:

To use **dup()** to copy the standard paths:

```
fork (cmd, parms, path)
char *cmd, *parms;
int path;

{
   int i;
   int save[3];

   for (i = 0; i <= 2; i++)
      save[i] = dup (i);

   for (i = 0; i <= 2; i++)
      close (i);

   for (i = 0; i <= 2; i++)
      dup (path);

   close (path);

   os9fork (cmd, strlen(parms), parms, 1, 1, 0);

   for (i = 0; i <= 2; i++) {
      close (i);
      dup (save[i]);
      close (save[i]);
   }
```

DIAGNOSTICS:

A -1 is returned if the call fails because there are too many opened files or the path number is invalid.

SEE ALSO:

open(), creat(), create(), close()

_errmsg() Print an error message

SYNOPSIS:

```
int _errmsg(nerr, msg[,arg1, arg2, arg3])
int nerr;
char *msg;
```

DESCRIPTION:

This function displays an error message on the standard error path along with the name of the program. The message string ***msg*** is displayed in the following format:

prog: <message text>

Note: **Prog** is the module name of the program and **<message text>** is the string passed as ***msg***.

For added flexibility in the message printing, the ***msg*** string can be a conversion string suitable for **printf()** with up to 3 additional arguments of any type. The argument ***nerr*** is returned as the value of the functions so **_errmsg()** can be used as a parameter to a function such as **exit()** or **prerr()**.

EXAMPLE:

Assume the program calling the function is named "foobar":

Call: _errmsg(1,"programmed message\n");
Prints: foobar: programmed message

Call: exit(_errmsg(errno,"unknown option '%c'\n",'q'));
Prints: foobar: unknown option 'q'

Then exits with the error code in errno.

SEE ALSO:

printf(), _prgname().

exit() Terminate a process after flushing any pending output

SYNOPSIS:

exit(status)
int status;

_exit(status)
int status;

DESCRIPTION:

Exit() is the normal means of terminating a task. The **exit()** function terminates a process after calling the Standard I/O library function **_cleanup(),** to flush any buffered output.

The **_exit()** function performs the same, but DOES NOT flush any file buffers prior to exiting the task. Neither the **exit()** or **_exit()** functions ever return.

A process finishing normally, that is, returning from **main()**, is equivalent to a call to **exit()**.

The status passed to **exit()** is available to the parent process if it is executing a **wait()**. An example is:

```
static int stat;
char *status = &stat;
\.
\.
os9fork (cmds, strlen(params), params, 1, 1, 0);
wait (status);
\.
\.
```

SEE ALSO:

wait()

fclose(), fflush() Close or flush a stream (file)

SYNOPSIS:

```
#include <stdio.h>

fclose(fp)
FILE *fp;

fflush(fp)
FILE *fp;
```

DESCRIPTION:

The **fclose()** routine causes any buffers for the named file pointer ***fp*** to be emptied, and the file to be closed. Buffers allocated by the standard input/output system are freed for use by another **fopen()** call. **Fclose()** should always be called to close access to a file when it is no longer needed. The **fclose()** routine is performed automatically upon calling **exit()**. The **fflush()** routine causes any buffered data associated with the named output file pointer fp to be cleared and written to that file, but only if the file was opened inthe write or update mode. It is not normally necessary to call **fflush()**, but it can be useful when, for example, normal output is to **stdout**, and it is wished to send something to **stderr**, which is unbuffered. If **fflush()** were not used and **stdout** was referred to the terminal, the **stderr** message would appear before large chunks of the **stdout** message even though the latter was written first. The file associated with ***fp*** remains open after the call.

DIAGNOSTICS:

These routines return **EOF** if the file pointer ***fp*** is not associated with an output file, or if buffered data cannot be written to that file. You should always check the returned status of and **fclose()** call.

CAVEATS:

In cases where **fclose()** is called as a result of an **exit()** call, the error may be returned, but no process is running to receive it. In this case, the data in the buffer will not be written to the file and the operator will NOT GET an error message.

SEE ALSO:

exit(), close(), fopen(), setbuf()

ferror(), feof() clearerr(), fileno() — Return status inquiries of files

SYNOPSIS:

```
#include <stdio.h>

feof(fp)
FILE *fp;

ferror(fp)
FILE *fp;

clearerr(fp)
FILE *fp;

fileno(fp)
FILE *fp;
```

DESCRIPTION:

The **ferror()** function returns nonzero when an error has occurred reading or writing the file associated with the file pointer ***fp*** has reached its end, otherwise zero is returned. Unless cleared by**CLEARERR**, the error indication lasts until the file pointed to by ***fp*** is closed, thus preventing any further access to that file.

The **feof()** function returns nonzero when end of file is read on the named input ***fp***, otherwise zero.

The **clearer()** function resets both the error and **EOF** indicators on the named file associated with ***fp***. Note that the file is not "fixed" nor does it prevent the error from occurring again. It just allows Standard Library functions to at least try to access the file.

The **fileno()**function returns the integer path descriptor associated with the file pointer ***fp***, for use with Standard Library calls the use path numbers, such as **close()**, **open()**, etc.

CAVEATS:

These functions are implemented as macros in **<stdio.h>** so they cannot be redeclared.

SEE ALSO:

open(), fopen()

findstr(), findnstr() String search

SYNOPSIS:

```
findstr(pos, string, pattern)
char *string, *pattern;
int pos;

findnstr(pos, string, pattern, len)
char *string, *pattern;
int pos, len;
```

DESCRIPTION:

These functions search the string pointed to by ***string*** for the first instance of the pattern pointed to by ***pattern*** starting at position ***pos*** (where the first position in a string is 1 not 0). The returned value is the position of the first matched character of the pattern in the string, or zero if a match is not found.

Flindstr() stops searching the string when a null byte is found.

Findnstr() only stops searching at position **pos + len** so it may continue past nulls.

CAVEATS:

The current implementation does not use the most efficient algorithm for pattern matching so use on very long strings is likely to be somewhat slow.

SEE ALSO:

patmatch(), index(), rindex(), strchr(), strrchr()

open(), freopen(), fdopen() Open a file

SYNOPSIS:

```
#include <stdio.h>

FILE *fopen (filename, type)
char *filename, *type;
```

FILE *freopen (filename, type, stream)
char *filename, *type;
FILE *stream;

FILE *fdopen (fildes, type)
int fildes;
char *type;

DESCRIPTION:

Fopen() opens a file and returns a file pointer to the file structure associated with that file. The pointer ***filename*** points to a character string that contains the name of the file to be opened.

The pointer ***type*** is a character string having one of the following values:

"r" - Open for reading

"w" - Truncate or create for writing

"a" - Append; open for writing at end of file, or create for writing

"r+" - Open for reading and writing (update)

"w+" - Truncate or create for reading and writing (update)

"a+" - Append; open or create for reading and writing at end-of-file.

"d" - Open a directory file for reading ONLY

Any of the above types may have a "x" after the initial letter which indicates to **fopen()** that is should look in the current execution directory if a full pathname is not given in ***filename***. The "x" also specifies that the file should have "execute" permissions.

Opening for write will perform a **creat()** call. If a file with the same name exists when the file is opened, it will be truncated to zero length. Append means to open for write and position the file pointer to the end of the file. Writes to the file will then extend the file until **fclose()** is called. The file will only be created if it does not already exist. All files created with **fopen()** will have file permissions set for user read and write and read only for all others. To set other combinations of file permissions, use **create()**.

Three file pointers are available and considered open as soon as a program is run. These are:

stdin - the standard input path (0)
stdout - the standard output path (1)
stderr - the standard error output path (2)

All files are automatically buffered except stderr unless it is made

unbuffered by a call to **setbuf().**

The **freopen()** routine substitutes the named file in place of the open stream. The original stream is closed, regardless of whether the open ultimately succeeds. The **freopen()** routine returns a pointer to the ***file*** structure associated with pointer ***stream***.

The **freopen()** routine is typically used to attach the preopened ***streams*** associated with **stdin**, **stdout**, and **stderr** to other files.

The **fdopen()** routine associates a ***stream*** with a path descriptor. Path descriptors are obtained from **open()**, **dup()**, **creat()**, or **create()**, which open files but do not return pointers to ***file*** structure. Streams are necessary input for many of the Section 3s library routines. The ***type*** of ***stream*** (r,w,a) must agree with the mode of the open file.

When a file is opened for update, both input and output may be done on the resulting ***stream***. However, output may not be directly followed by input without an intervening **fseek()** or **rewind()**, and input may not be directly followed by output without an intervening **fseek()**, **rewind()**, or an input operation which encounters end-of-file.

All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output regardless of its current position. If two separate processes open the same file for append, each process may write freely to the file without fear of destroying output being written by the other. The output from the two processes will be intermixed in the file in the order in which it is written.

CAVEATS:

The ***type*** passed as an argument to **fopen()** must be a pointer to a string and NOT a character constant. For example:

```
fp = fopen("foo", "r");  is correct
fp = fopen("foo", 'r');  is not
```

DIAGNOSTICS:

Fopen() returns null (0) if the call was not successful.

SEE ALSO:

creat(), create(), dup(), open(), fclose(), fseek()

fread(), fwrite() Read/write binary data

SYNOPSIS:

fread(ptr, sizeof(item), nitems, fp)
FILE *fp;

fwrite(ptr, sizeof(item), nitems, fp)
FILE *fp;

DESCRIPTION:

The **fread()** function reads into a buffer beginning at ***ptr***, ***nitems*** of data of the type ***item*** from the input file pointer ***fp***. It returns the number of items actually read.

The **fwrite()** function writes, at most, ***nitems*** of data of the type ***item***
beginning at ***ptr*** to the named output file ***fp***. It returns the number of items actually written.

In both cases, the type ***item*** refers to either a char, int, or unsigned data type. Care must be taken to insure the correct values are used. If, for example, 10 bytes of type ***char*** are to be written, then this is the same amount of data going to the output stream as 5 bytes of type ***int***.

DIAGNOSTICS:

Both functions return a null at end of file or if an error occurs. To insure the correct data is read or written, the returned number of type ***item*** should be compared to what was intended. If there is a difference, an error has occured.

SEE ALSO:

read(), write(), fopen(), getc(), putc(), printf(), scanf()

fseek(), ftell() rewind() Reposition a file pointer or report position

SYNOPSIS:

#include <stdio.h>

fseek(fp, offset, place)
FILE *fp;
long offset;
int place;

long ftell(fp)
FILE *fp;

rewind(fp)
FILE *fp;

DESCRIPTION:

The **fseek()** function sets the position of the next input or output operation on the already opened file pointed to by fp. The new position is at ***offset*** bytes from the beginning, the current position, or the beginning of the file if ***place*** has the value 0, the current position if 1, or the end of the file if 2.

The **fseek()** function undoes any effects of **ungetc()** and sorts out the problems associated with buffered I/O.

NOTE:
Using **lseek()** on a buffered file will produce unpredictable results.

The **ftell()** function returns the current value of the offset relative to the beginning of the file associated with the file pointer ***fp***. It is measured in bytes and is the only foolproof way to obtain an ***offset*** for **fseek()**.

The **rewind()** function is equivalent to **fseek(fp, 0L, 0),** except that no value is returned. It returns the file pointer to the beginning of the file.

DIAGNOSTICS:

Fseek() returns a -1 if the call is invalid, otherwise it returns zero.

SEE ALSO:

lseek()

getc(), getchar(), getw() Get character or word from a file

SYNOPSIS:

#include <stdio.h>

getc(fp)
FILE *fp;
getchar()

getw(fp)
FILE *fp;

DESCRIPTION:

The **getc()** function returns the next character from the named input ***fp***.

The **getchar()** function is identical to **getc(stdin)**.

The **getw()** function returns the next word from the named input ***fp***. It returns the constant EOF upon end of file or error, but since that is a good integer value, **feof()** and **ferror()** should be used to check the success of **getw()**.The **getw()** assumes no special alignment in the file.

CAVEATS:

Because it is implemented as a macro, **getc()** treats a ***fp*** argument with side effects incorrectly. In particular, **'getc(*f++);'** doesn't work as expected.

SPECIAL CONSIDERATIONS:

Under OS-9, there is a choice of system calls to use when reading from a file. **Read()** will get characters up to a specified number in the "raw" mode, i.e., no editing will take place on the input stream and the characters will appear to the program exactly as in the file. **ReadIn()** on the other hand, will honor the various mapping of characters associated with a serial device such as a terminal and will return as soon as a carriage return is seen on the input.

In the vast majority of cases, it is preferable to use **readln()** for accessing serial character devices and **read()** for any other file input. **Getc()** uses this strategy and as all file input using the Standard Library functions is routed through **getc(),** so do all the other input functions. The choice is made when the first call to **getc()** is made after the file has been opened. The system is consulted for the status of the file and a flag bit is set in the file structure accordingly. The choice may be forced by the programmer by setting the relevant bit before a call to **getc()**. The flag bits are defined in **<stdio.h>** as **_SCF** and **_RBF** and the method used is as follows:

Assuming that the file pointer for the file as returned by **fopen()** is ***f***, ***f***->_flag |= **_SCF** will force the use of **readln()** on input, and ***f***->flag |= **_RBF** will force the use of **read()**. This trick may be played on the standard input, output, and error files without the need to call **fopen()** but must be made before any input is requested from these files.

DIAGNOSTICS:

These functions return the integer constant EOF (-1) at end of file or upon read error.

SEE ALSO:

fopen(), fread(), gets(), putc(), scanf(), ungetc()

getopt() Get an option letter from argument vector

SYNOPSIS:

```
int getopt (argc, argv, optstring)
int argc;
char **argv, *optstring;
extern char *optarg;
extern int optind, opterr;
```

DESCRIPTION:

Getopt() returns the next option letter in ***argv*** that matches a letter in ***optstring***. ***Optstring*** is a string of recognized option letters; if a letter is followed by a colon, the option is expected to have an argument that may or may not be separated from it by white space. ***Optarg*** is set to point to the start of the option argument on return from **getopt()**.

Getopt() places in ***optind*** the ***argv*** index of the next argument to be processed. Because ***optind*** is external, it is normally initialized to zero automatically before the first call to **getopt()**.

When all options have been processed (i.e., up to the first non-option argument), **getopt()** returns **EOF**. The special option -- may be used to delimit the end of the options; **EOF** will be returned, and -- will be skipped.

DIAGNOSTICS:

Getopt() prints an error message on the stderr path and returns a question mark (?) when it encounters an option letter not included in ***optstring***. This error message may be disabled by setting ***opterr*** to a non-zero value.

EXAMPLE:

The following code fragment shows how one might process the arguments for a command that can take the mutually exclusive options a and b, and the options f and o, both of which require arguments:

```
main (argc, argv)
int argc;
char **argv;
{
        int c;
        extern char *optarg;
        extern int optind;
\&.
\&.
\&.
\&.
while ((c = getopt(argc, argv, "abf:o:")) != EOF)
        switch (c) {
                case 'a':
                        if (bflg)
                          errflg++;

                        else
                          aflg++;
                        break;
                case 'b':
                        if (aflg)
```

```
                        errflg++;
                    else
                      bproc();
                    break;
            case 'f':
                    ifile = optarg;
                    break;
            case 'o':
                    ofile = optarg;
                    break;
            case '?':
                    errflg++;
}
if (errflg) {
            fprintf(stderr, "Usage: . . . ");
            exit (2);
}
for ( ; optind < argc; optind++) {
            if (access(argv[optind], 4)) {
\&.
\&.
\&.
}
```

getpid() Get a process ID

SYNOPSIS:

getpid()

DESCRIPTION:

A number unique to the currently running process is often useful in creating names for temporary files and many other uses. This call returns the process' system ID number as returned to its parent by **os9fork()**.

SEE ALSO:

os9fork()

gets(), fgets() Get a string from a file

SYNOPSIS:

```
#include <stdio.h>

char *gets(s)
char *s;

char *fgets(s, n, fp)
FILE *fp;
char *s;
int n;
```

DESCRIPTION:

The **gets()** routine reads a string into ***s*** from the standard input path **stdin** . The string is terminated by a newline character, which is replaced in s by a null character. The **gets()** routine returns its argument.

Fgets() routine reads **n-1** characters, or up to a newline character, whichever comes first, from the file pointed to by ***fp*** into the string ***s***. The last character read into ***s*** is followed by a null character. **Fgets()** routine returns its first argument.

CAVEATS:

The different treatment of the newline ("\\n") character by these functions is retained here for portability reasons.

DIAGNOSTICS:

Both functions return null on end-of-file or if an error occurs.

SEE ALSO:

ferror(), fread(), getc(), puts(), scanf()

SS_OPT, SS_READY OS-9 get status system calls
SS_SIZE, SS_POS, SS_EOF
SS_DEVNAM, SS_FD

SYNOPSIS:

```
#include <sgstat.h>

getstat(SS_OPT, filenum, buffer)
```

```
int code, filenum;
struct sgbuf *buffer;

getstat(SS_READY, filenum)
int code, filenum;

getstat(SS_SIZE, filenum, size)
int code, filenum;
long *size;

getstat(SS_POS, filenum, position)
int code, filenum;
long *position;

getstat(SS_EOF, filenum)
int code, filenum;

getstat(SS_DEVNAM, filenum, buffer)
int code, filenum;
char *buffer;

#include <direct.h>

getstat(SS_FD, filenum, buffer, count)
int code, filenum;
struct fildes *buffer;
int count;
```

DESCRIPTION:

These calls are the equivalent of the **_gs_XXX** calls described elsewhere in these documents. While the**_gs_XXX** calls are provided for compatibility with source code generated for OSK and UNIX systems, the calls listed below are those normally used with 6809/OS-9.

The following descriptions do not include the complete syntax of each function call. See the list above under Synopsis.

Getstat(ss_opt) copies the options section of the path descriptor opened on ***path*** into the buffer pointer to by ***buffer***. The structure ***sgbuf_* <sgstata.h>** provides a convenient means to access the individual option values.

Be sure the buffer is large enough to hold all the options. Declaring the buffer as a type ***struct sgbuf*** is perfectly safe as this structure is predefined to be large enough for all the device descriptor options.

This call works only on **RBF** devices.

Getstat(ss_ready) checks an **SCF** device opened on ***path*** to see if data is waiting to be read. Read requests to OS9 will wait until enough bytes have been read to satisfy the bytecount given

thereby suspending the program until the read is finished.

A program can use this function to determine the number of bytes, if any, are waiting to be read, and then issue a read request for only the number of bytes actually received.

If no data is available to be read, or the device is not an **SCF** device, a -1 is returned and the appropriate error code is placed in ***errno***. Otherwise, the number of bytes available to be read is returned.

This call is effective only on **SCF** devices. This function is not intended for use with buffered I/O calls (like **getc()**), and unpredictable results will likely occur. This call works best with low-level I/O functions.

Getstat(ss_size) returns the size of the file pointed to in ***filenum***. The size is returned in the long variable '***size***'.

Getstat(ss_pos) returns the value of the file pointer for the file opened on ***filenum***.

If an error occurs, this function returns -1 as its value and the error code is placed in the global variable ***errno***.

This call works only on **RBF** devices. It is unique to OS-9 and the only equivalent portable call is **lseek()**. DO NOT use this call if buffered I/O is being performed on the path. Use **ftell()** instead.

Getstat(ss_eof) determines if the file opened is at the end-of-file. If it is, the value 1 is returned -- if not, 0 is returned.

If an error occurs, this function returns -1 as its value and the error code is placed in the global variable ***errno***.

This function cannot determine the end-of-file on an **SCF** device. **SCF** devices return an **e$eof** error when the **EOF** character is read. DO NOT use this call if using buffered I/O on PATH. Instead, use the function **feof()**.

Getstat(ss_devnam) determines the name of the device opened on a path. The argument ***filenum*** is the OS-9 path number of the opened path and ***buffer*** is a pointer to a character array into which the null-terminated device name will be placed.

If an error occurs, this function returns -1 as its value and the error code is placed in the global variable ***errno***.

Be sure to reserve at least 32 characters to receive the device name.

Getstat(ss_fd) places a copy of the **RBF** file descriptor sector of the file opened into the buffer pointed to by ***buffer***. A maximum of ***count*** bytes are copied. The structure **FILDES**, declared in **<direct.h>**, provides a convenient method to access the file descriptor information.

If and error occurs, this function returns -1 as its value and the error code is placed in the global

variable ***errno***.

Be sure the buffer is large enough to hold all the options, or at least ***count*** bytes. This call is effective only on **RBF** devices. Declaring the buffer as type ***struct fildes*** is perfectly safe as this structure is predefined to be large enough to hold all the file descriptor information.

NOTE:

All the above calls require an OS-9 path number for ***filenum***, and NOT a C iob file descriptor pointer.

SEE ALSO:

I$GetStt system call - Microware Manual, _gs_xxx calls

getuid() Get a user ID

SYNOPSIS:

getuid()

DESCRIPTION:

GETUID returns the real user ID if the currently executing process as maintained in the password file /**dd**/**sys**/**password**.

SEE ALSO:

setuid(), asetuid()

_gs_devn() Get device name

SYNOPSIS:

```
int _gs_devn(path, buffer)
int path;
char *buffer;
```

DESCRIPTION:

This function determines the name of the device opened on a path. The argument ***path*** is the OS-9 path number of the opened path and ***buffer*** is a pointer to a character array into which the null-terminated device name will be placed.

The device name returned is in "OS9 format", that is, the last byte of the name will have the high bit set. Also, there is no NULL terminator on the string. The best method to handle this returned value is:

```
_gs_devn(path, buffer);
strhcpy(newbuff, buffer);
```

If ***path*** is invalid, this function returns -1 as its value and the error code is placed in the global variable ***errno***.

NOTE:

Be sure to reserve at least 32 characters to receive the device name.

SEE ALSO:

I$GetStt system call - Microware Manual, fopen(), open().

_gs_eof() Get end-of-file status

SYNOPSIS:

```
int _gs_eof(path)
int path;
```

DESCRIPTION:

This function determines if the file opened on ***path*** is at the end-of-file. If it is, the value 1 is returned -- if not, 0 is returned.

If ***path*** is invalid, a -1 is returned and the appropriate error code is placed in the global variable ***errno***.

NOTE:

This function cannot determine the end-of-file on an SCF device. SCF devices return an E$EOF error when the EOF character is read. DO NOT use this call if using buffered I/O on PATH. Instead, use the function **feof().**

SEE ALSO:

I$GetStt system call - Microware Manual, feof().

_gs_gfd() Get file descriptor

SYNOPSIS:

```
#include <direct.h>

int _gs_gfd(path, buffer, count)
int path;
struct fildes *buffer;
```

int count;

DESCRIPTION:

This function will place a copy of the **RBF** file descriptor sector of the file opened on PATH into the buffer pointed to by ***buffer***. A maximum of ***count*** bytes are copied. The structure ***fildes***, declared in **<direct.h>**, provides a convenient method to access the file descriptor information.

If and error occurs, this function returns -1 as its value and the error code is placed in the global variable ***errno***.

NOTE:

Be sure the buffer is large enough to hold all the options, or at least ***count*** bytes. This call is effective only on **RBF** devices. Declaring the buffer as type "***struct fildes***" is perfectly safe as this structure is predefined to be large enough to hold all the file descriptor information.

SEE ALSO:

I$GetStt system call - Microware Manual, _ss_pfd().

_gs_opt() Get file descriptor options

SYNOPSIS:

#include <sgstat.h>

int _gs_opt(path, buffer)
int path;
struct sgbuf *buffer;

DESCRIPTION:

This function copies the options section of the path descriptor opened on ***path*** into the buffer pointer to by ***buffer***. The structure ***sgbuf*** in **<sgstat.h>** provides a convenient means to access the individual option values.

If ***path*** is invalid, this function returns -1 as its value and the error code is placed in the global variable ***errno***.

NOTE:

Be sure the buffer is large enough to hold all the options. Declaring the buffer as a type "struct sgbuf" is perfectly safe as this structure is predefined to be large enough for all the device descriptor options.

SEE ALSO:
I$GetStt system call - Microware Manual, getstat(), _ss_opt().

_gs_pos () Get file position pointer

SYNOPSIS:

long _gs_pos(path)
int path;

DESCRIPTION:

This function returns the value of the file pointer for the file opened on ***path***.

If ***path*** is invalid, this function returns -1 as its value and the error code is placed in the global variable ***errno***.

NOTE:

This call works only on **RBF** devices. It is unique to OS-9 and the only equivalent portable call is **lseek().** DO NOT use this call if buffered I/O is being performed on the path. Use **ftell()** instead.

SEE ALSO:

I$GetStt system call - Microware Manual, lseek(), ftell()

_gs_rdy() Get path status

SYNOPSIS:

int _gs_rdy(path)
int path;

DESCRIPTION:

This function checks an **SCF** device opened on ***path*** to see if data is waiting to be read. Read requests to OS9 will wait until enough bytes have been read to satisfy the bytecount given, thereby suspending the program until the read is finished.

A program can use this function to determine the numbers of bytes, if any, are waiting to be read, and then issue a read request for only the number of bytes actually received.

If ***path*** is invalid, no data is available to be read, or the device is not an **SCF** device, a -1 is returned and the appropriate error code is placed in ***errno***. Otherwise, the number of bytes available to be read is returned.

NOTE:

This call is effective only on **SCF** devices. This function is not intended for use with buffered I/O calls (like getc()), and unpredictable results will likely occur. This call works best with low-level I/O

functions.

SEE ALSO:

I$GetStt system call - Microware Manual, getstat(), read(), readln().

_gs_size() Get file size

SYNOPSIS:

```
long _gs_size(path)
int path;
```

DESCRIPTION:

This function is used to determine the current size of the file opened on ***path***, and returns this value to the calling function.

If ***path*** is invalid, this function returns -1 as its value and the error code is placed in the global variable ***errno***.

NOTE:

This call works only on **RBF** devices.

SEE ALSO:

I$GetStt system call - Microware Manual, getstat()

htoi(), htol(), itoa() utoa(), ltoa() Type conversions

SYNOPSIS:

```
int htoi(s)
char *s;

long htol(s)
char *s;

char *itoa(n, s)
int n;
char *s,

char utoa(n, s)
int n;
char *s;
```

```
char *ltoa(n, s)
long n;
char *s;
```

DESCRIPTION:

Htoi() converts a string representing a hexadecimal number into an integer.

Htol() converts a string representing a hexadecimal number into a long integer.

Itoa() converts an integer number ***n*** to the corresponding ASCII characters and returns a pointer to the string ***s***.

Utoa() converts an unsigned integer number ***n*** to the corresponding ASCII characters and returns a pointer to the string ***s***.

Ltoa() converts a long number ***n*** to the corresponding ASCII characters and returns a pointer to the string ***s***.

NOTE:

These functions are extensions to the **atof(), atol(),** and **atoi()** functions. They perform in the same manner, except for the type of conversion.

SEE ALSO:

atof(), atoi(), atol()

intercept() Set a function for interrupt processing

SYNOPSIS:

```
intercept(func)
int (*func)();
```

DESCRIPTION:

Intercept() instructs OS-9 to pass control of the process to the function ***func*** when an interrupt (signal) is received. If the interrupt processing function has an argument, it will contain the value of the signal received. On return from ***func***, the process resumes at the point in the program where it was interrupted by the signal. **Intercept()** is an alternate to the use of **signal()** to process interrupts.

As an example, suppose we wish to ensure that a partially completed output file is deleted if an interrupt is received. The body of the program might include:

```
char *temp_file = "temp";
int pn = 0;
int intrupt();

intercept(intrupt);
pn = creat(temp_file, 3);
write(pn, string, count);
close(pn);
pn = 0;
```

The interrupt routine might be:

```
intrupt(sig)
{
   if (pn) {
      close(pn);
      unlink(temp_file);
   }
   exit(sig);
}
```

CAVEATS:

Intercept() and **signal()** are mutually incompatible so that calls to both must not appear in the same program. The linker guards against this by giving an "entry name clash - sigint" error if it is attempted.

kill() Send an interrupt to a process

SYNOPSIS:

```
#include <signal.h>

kill(tid, interrupt)
int tid, interrupt;
```

DESCRIPTION:

Kill() sends the signal type ***interrupt*** to the process with id ***tid***. Both tasks, sender and receiver, must have the same user ID unless the users or sender is the super user.

The include file contains definitions of the defined signals as follows:

```
#define  SIGKILL  0   /* system abort cannot be */
                      /* caught or ignored */

#define  SIGWAKE  1   /* wake up signal */
#define  SIGQUIT  2   /* keyboard abort signal */
```

```
#define  SIGINT   3       /* keyboard interrupt signal */
```

Other user-define signals my also be sent.

DIAGNOSTICS:

Kill() returns 0 from a successful call and -1 if the process does not exist, the effective user IDs do not match, or the user is not the super user.

SEE ALSO:

signal(), OS-9 Shell command "kill"

l3tol(), ltol3() Convert between 3-byte integers and long integers

SYNOPSIS:

```
void l3tol (lp, cp, n)
long *lp;
char *cp;
int n;

void ltol3 (cp, lp, n)
char *cp;
long *lp;
int n;
```

DESCRIPTION:

The **l3tol()** subroutine converts a list of ***n*** three-byte integers packed into a character string pointed to by ***cp*** into a list of long integers pointed to by ***lp*** .

The **ltol3()** performs the reverse conversion from long integers ***lp*** to three-byte integers ***cp***.

These functions are useful for file-system maintenance where the block numbers are three bytes long. Certain disc addresses are maintained in three-byte form rather than four-bytes.

CAVEATS:

Because of possible differences in byte ordering, the numerical values of the long integers are machine-dependent.

lseek() seek to a position within a file

SYNOPSIS:

long lseek(pn, position, type)
int pn;
long position;
int type;

DESCRIPTION:

The read or write pointer for the opened file with the path number ***pn*** is positioned by **lseek()** to the specified place in the file. The ***type*** indicates from where ***position*** is to be measured: if 0, from the beginning of the file, if 1, from the current pointer location, and if 2, from the end of the file.

Seeking to a location beyond the end of a file opened for writing and then writing to it creates a "hole" in the file which appears to be filled with zeros from the previous end to the position desired.

The returned value is the resulting position in the file unless there is an error, so to find the current position use:

lseek (pn, 0l, 1);

CAVEATS:

The argument ***position*** MUST be a long integer. Constants should be explicitly made long by appending an "l" (el - lower case L), as above, any other type should be converted using a cast:

lseek (pn, (long)pos, 1);

Notice also that the returned value from **lseek()** is itself a long integer.

DIAGNOSTICS:

A -1 is returned if ***pn*** is a bad path number, or attempting to seek to a position before the beginning of the file.

SEE ALSO:

open(), creat(), create(), fseek()

SYNOPSIS:

```
char *malloc(size)
unsigned size;

free(ptr)
char *ptr;

char *realloc(ptr, size)
char *ptr;
unsigned size;

char *calloc(nelem, elsize)
unsigned nelem, elsize;
```

DESCRIPTION:

The **malloc()** and **free()** subroutines provide a simple general-purpose memory allocation package. The **malloc()** subroutine returns a pointer to a block of at least ***size*** bytes beginning on a word boundary.

The argument to **free()** is a pointer to a block previously allocated by **malloc()**. This space is made available for further allocation, but its contents are left undisturbed.

Needless to say, grave disorder will result if the space assigned by **malloc()** is overrun or if some random number is handed to **free()**.

The **malloc()** subroutine maintains multiple lists of free blocks according to size, allocating space from the appropriate list. It calls ***sbrk*** to get more memory from the system when there is no suitable space already free. For further information, see **brk()**.

The **realloc()** subroutine changes the size of the block pointed to by ***ptr*** to ***size*** bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes.

If the pointer argument ***ptr*** is zero, then **realloc()** degenerates into a **malloc()**.

In order to be compatible with older versions, **realloc()** also works if ***ptr*** points to a block freed since the last call of **malloc()**, **realloc()**, or **calloc()**. Sequences of **free()**, **malloc()**, and **realloc()** were previously used to attempt storage compaction. This procedure is no longer recommended.

The **calloc()** subroutine allocates space for an array of ***nelem*** elements of size ***elsize***. The space is initialized to zeros.

CAVEATS:

When **realloc()** returns 0, the block pointed to by ***ptr*** may be destroyed.

DIAGNOSTICS:

Malloc(), **free()**, and **calloc()** return a null. if no free memory can be found, or if there was an error.

rand(), srand(), unmin() unmax(), max(), min() — Math functions

SYNOPSIS:

```
long rand()

/* void */ srand(n)
unsigned n;

int max(v1, v2)
int min(v1, v2)
int v1, v2;

int unmax(v1, v2)
int unmin(v1, v2)
unsigned v1, v2;
```

DESCRIPTION:

Rand() returns a random number in the range of 0 -> 32767. **Srand()** seeds the random number generator and returns nothing. **Srand()** uses the dual table method proposed by Knuth.

Max() and **min()** select either the larger of the smaller of variables ***v1*** and ***v2***. **Unmax()** and **unmin()** perform the same function, but on unsigned numbers. Each of these functions returns a type **INTEGER** number.

EXAMPLE:

To seed the random number generator, use the value returned from **time()**:

```
long time();

srand ((unsigned) time(0));
```

SEE ALSO:

clibt.l Transentental Math library.

SYNOPSIS:

```
#include <memory.h>

char *memccpy (s1, s2, c, n)
char *s1, *s2;
int c, n;

char *memchr (s, c, n)
char *s;
int c, n;

int memcmp (s1, s2, n)
char *s1, *s2;
int n;

char *memcpy (s1, s2, n)
char *s1, *s2;
int n;

char *memset (s, c, n)
char *s;
int c, n;
```

DESCRIPTION:

These functions operate as efficiently as possible on memory areas (arrays of characters bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

Memccpy() copies characters from memory area ***s2*** into ***s1***, stopping after the first occurrence of character ***c*** has been copied, or after ***n*** characters have been copied, whichever comes first. It returns a pointer to the character after the copy of ***c*** in ***s1***, or a null pointer if ***c*** was not found in the first ***n*** characters of ***s2***.

Memchr() returns a pointer to the first occurrence of character ***c*** in the first ***n*** characters of memory area ***s***, or a null pointer if ***c*** does not occur.

Memcmp() compares its arguments, looking at the first ***n*** characters only, and returns an integer less than, equal to, or greater than 0, according as ***s1*** is lexicographically less than, equal to, or greater than ***s2***.

Memcpy() copies ***n*** characters from memory area ***s2*** to ***s1***. It returns ***s1***.

Memset() sets the first ***n*** characters in memory area s to the value of character ***c***. It returns ***s***.

NOTE:

For user convenience, all these functions are declared in the optional **<memory.h>** header file.

BUGS:

Memcmp() uses native character comparison, which is unsigned on some machines. Thus, the sign of the value returned when one of the characters has its high order bit set is implementation-dependent.

Character movement is performed differently in different implementations. Thus, overlapping moves may yield surprises.

mknod() Create a directory

SYNOPSIS:

```
#include <modes.h>

mknod(fname, desc)
char *fname;
int desc;
```

DESCRIPTION:

Mknod() may be used to create a new directory. ***Fname*** should be a pointer to a string containing the desired directory name. ***Desc*** is a descriptor specifying the desired modes (file type) and the permissions of the new file.

The include file defines possible values for ***desc*** as follows:

```
/* permissions */

#define  S_IREAD   0x01        /* owner read */
#define  S_IWRITE  0x02        /* owner write */
#define  S_IEXEC   0x04        /* owner execute */
#define  S_IOREAD  0x08        /* public read */
#define  S_IOWRITE 0x10        /* public write */
#define  S_IOEXEC  0x20       /* public execute */
#define  S_ISHARE  0x40       /* sharable */
```

DIAGNOSTICS:

Zero is returned if the directory has been successfully made, -1 if the file already exists.

SEE ALSO:
OS-9 command "makdir"

mktemp() Make a unique file name

SYNOPSIS:

char *mktemp(name)
char *name;

DESCRIPTION:

The **MKTEMP** subroutine replaces ***name*** by a unique file name, and returns the address of the name. The name should look like a file name with five trailing X's, which will be replaced with the current process ID.

EXAMPLE:

If instance, if ***name*** points to "foo.XXXXX", and the process ID is 351, the returned value points at the same place as ***name***, but it is now "foo.351".

SEE ALSO:

getpid()

modload(), modlink() Return a pointer to a module structure

SYNOPSIS:

#include <module.h>

mod_exec *modload(filename, type, language)
char *filename
int type, language;

mod_exec *modlink(modname, type, language)
char *modname;
int type, language;

DESCRIPTION:

Each of these calls returns a pointer to an OS-9 memory module.

Modload() will open a file which has the pathlist specified by ***filename*** and loads modules from the file adding them to the module directory. The returned value is a pointer to the first module loaded.

Modlink() will search the module directory for a module with the same name as ***modname*** and, if found, increment its link count.

In the synopsis above, each call is shown as returning a pointer to an executable module, but it will return a pointer to whatever type of module is found.

DIAGNOSTICS:

A -1 is returned on any error.

SEE ALSO:

munlink()

munlink() Unlink a memory module

SYNOPSIS:

```
#include <module.h>

munlink(mod)
mod_exec *mod;
```

DESCRIPTION:

This call informs the system that the memory module pointed to by ***mod*** is no longer required by the current process. Its link count is decremented, and the module is removed from the module directory if the link count reaches zero.

SEE ALSO:

modlink(), modload()

open() Open a file for read/write access

SYNOPSIS:

```
open(fname, mode)
char *fname;
int mode;
```

DESCRIPTION:

Open() opens an already existing file for reading if ***mode*** is 1, for writing if ***mode*** is 2, or reading and writing if ***mode*** is 3.

NOTE that these values are OS-9 specific and are not compatible with other systems ***Fname*** should point to a string representing the pathname of the file to be opened.

Open() returns an integer as the "path number" which should be used by I/O system calls referring to the file.

The position where reads or writes start is at the beginning of the file.

DIAGNOSTICS:

A -1 is returned if the file does not exist, if the pathname cannot be searched, if too many files are already open, or if the file permissions deny the requested mode.

SEE ALSO:

creat(), create(), read(), write(), dup(), close()

_os9() System call interface from C programs

SYNOPSIS:

```
#include <os9.h>

_os9(code, reg)
char code;
struct registers *reg;
```

DESCRIPTION:

_Os9() enables the programmer to access virtually any OS-9 system call directly from a C program without having to resort to assembly language routines.

Code is one of the codes that are define in **<os9.h>**. **<os9.h>** contains codes for the F$ and I$ function/service requests, and also contains getstt, setstt, and error codes.

The input registers (***reg)*** for the system calls are accessed by the following structure that is defined in os9.h:

```
struct registers {
   char rg_cc, rg_a, rg_b, rg_dp;
   unsigned rg_x, rg_y, rg_u;
};
```

An example program that uses _os9 is presented on the following page.

DIAGNOSTICS:

A -1 is returned if the OS-9 call fails. 0 (zero) is returned on success.

Program example:

```
#include <os9.h>
#include <modes.h>
```

```
/* This program does an I$GETSTT call to get file size */
main (argc, argv)
int argc;
char **argv;
{
    struct registers reg;
    int path;
/* Tell linker we need longs */
    pflinit();
/* low level open - filename is first command line param */
    path = open (*++argv, S_IREAD);
/* set up regs for call to OS-9 */
    reg.rg_a = path;
    reg.rg_b = SS_SIZE;
    if (_os9(I_GETSTT, &reg) == 0)
        printf ("filesize = %1x\n",
        (long) (reg.rg_x << 16) + reg.rg_u);
    else
        printf (OS9 error #%d\n, reg.rg_b & 0xff);
    dumpregs (&reg);
}
dumpregs(r)
register struct registers *r;
{
    printf("cc = %02x\n", r->rg_cc &0xff);
    printf(" a = %02x\n", r->rg_a &0xff);
    printf(" b = %02x\n", r->rg_b &0xff);
    printf("dp = %04x\n", r->rg_dp &0xff);
    printf(" x = %04x\n", r->rg_x);
    printf(" y = %04x\n", r->rg_y);
    printf(" u = %04x\n", r->rg_u);
}
```

os9fork() Create a new process

SYNOPSIS:

```
os9fork(modname, paramsize, parmamptr, type, lang, datasize)
char *modname, paramptr;
int paramsize, type, lang, datasize;
```

DESCRIPTION:

The action of **F$FORK**, the assembler equivalent of **os9fork()**is fully described in the OS-9 System manual. **Os9fork()** will create a process that runs concurrently with the calling process. When the forked process terminates, it will return to the calling process and pass back its exit status.

Modname should point to the name of the desired module. ***Paramsize*** is the length of the parameter string which should always be terminated with a '\\n', and ***paramptr*** points to the parameter string itself. ***Type*** is the module type as found in the program header (normally a 1 for "program"), and ***lang*** should match the language nibble in the module header (C programs have a 1 for "6809 machine language"). ***Datasize*** may be zero or it may contain the number of 256 byte pages to give to the new process as its initial memory allocation. If it is zero, the new process' memory allocation will be the amount specified in the program header.

DIAGNOSTICS:

A -1 will be returned on an error, or the ID number of the child process will be returned upon success.

EXAMPLE:

An example of typical usage would be:

```
static int stat;
char *status = &stat;

fork(module, params)
char *module, *params;
{
   os9fork (module, strlen(params), params, 1, 1, 0);
   wait (status);
}
```

*getpwent(), *getpwuid() *getpwnam(), setpwent() endpwent(), getpwdlm()

Password file operations

SYNOPSIS:

```
#include <password.h>

PWENT *getpwent()

PWENT *getpwuid(uid)
int uid;

PWENT *getpwnam(name)
char *name;

void setpwent()

void endpwent()
```

int getpwdlm()

DESCRIPTION:

Each of the functions described below perform some operation on the **PASSWORD** file maintained in /**DD**/**SYS**. Notice also that three of the functions are declared as pointer functions, and two are of type VOID.

Getpwent() returns a pointer to a structure containing the broken down password entry.It searches for the file /**DD**/**SYS**/**PASSWORD**, opens it on the first call, and reads the first password entry. Any subsequent calls will overwrite the data contained in the structure ***PWENT*** defined in **<password.h>,** so that data must be copied out before the next call is made to preserve it.

This function returns a null upon reaching the end of the password file, and a -1 if an error occurs.

Getpwuid() performs the same function as **getpwent()** above, but it searches the password file until a given user ID, as defined by UID, is found.

Getpwnam() again performs the same as **getpwent()**, but searches the password file until the entry defined my ***name*** is found. The search for ***name*** is not case sensitive.

Setpwent() rewinds the password file pointer so additional reads can be made after the end of the file is reached.

Endpwent() terminates access and closes the password file.

Getpwslm() returns the current password file delimiter character. In an OS9 password file, the field delimiting character is a comma, while in other password utilities and files, a semi-colon is used for the delimiter character. This function is provided as a means to check the current type.

NOTE:

For user convenience, function declarations are made in the header file **<password.h>** provided below.

```
/*
**  Password.h
**
**  Definitions for accessing the OS9 password file.
**  Two different delimiters are accepted, ',' and ':'.
**  In the second (Unix like) case, an extra field is
**  defined for comments (ugcos).
*/

#ifdef TEST
#define  PASSWORD "/dd/sys/massword"
#else
#define  PASSWORD "/dd/sys/password"
#endif
```

```
#define  PWEMAX   64    /* maximum lines in password file */
#define  PWSIZ    132     /* maximim size of password file line */
#define  PWNSIZ   32     /* maximum size of user's name */
#define  PWPSIZ   32     /* maximum size of user's password */
#define  UNXDLM   ':'    /* Unix style password file delim */
#define  OS9DLM   ','    /* OS9 style password file delim */

typedef struct {
     char  *unam,
           *upw,
           *uid,
           *upri,
           *ugcos,
           *ucmd,
           *udat,
           *ujob;                  /* field pointers */
     } PWENT;


/* returns a pointer to broken down password entry */
PWENT    *getpwent();

/* same, but for the given int uid */
PWENT    *getpwuid();

/* same, but for the given char *name */
PWENT    *getpwnam();

/* rewinds the password file for another scan */
/*void*/ setpwent();

/* terminates password file access (closes) file */
/*void*/ endpwent();

/* returns the current password entry delimiter */
/*void*/ getpwdlm();
```

patmatch() Tests one string with another for a match

SYNOPSIS:

```
patmatch(pat, s, flag)
char *pat, *s
int flag
```

DESCRIPTION:

Patmatch() searches the string s for the pattern in ***pat*** and returns true if there is a match. The pattern ***pat*** must contain wildcard characters of '*' and '?', where '*' denotes a string of characters of any type and length, and '?' denotes a single character of any type. Expansion of wildcards is performed within the function.

If no match is found, false is returned.

If ***flag*** is set as true, **patmatch()** will ignore the case of both strings buy calling the function **toupper()** to make both the pattern and the string all uppercase characters prior to matching. Otherwise, an exact match is required.

EXAMPLE:

```
if (patmatch("*.ar", dir_string, TRUE) == TRUE)
   puts("A match has been found");
else
   puts("No match was found");
```

pause() Halt and wait for an interrupt/signal

SYNOPSIS:

pause()

DESCRIPTION:

Pause() may be used to halt a process until an interrupt or signal is received from **kill()**. An equivalent function is **tsleep(0)**.

DIAGNOSTICS:

PAUSE always returns a -1.

SEE ALSO:

kill(), signal(), tsleep()

prerr() Print an error message

SYNOPSIS:

```
prerr(filnum, errcode)
int filnum, errcode;
```

DESCRIPTION:

Prerr() prints an error message on the output path specified by ***filenum***, which must be the path number of an already opened file. The message depends on ***errcode*** which will normally be a standard OS-9 error code. ***Filenum*** may also be the standard output or standard error paths which are always opened at program start.

SEE ALSO:

errmsg()

_prgname() Get a module name

SYNOPSIS:

```
char *_prgname()
```

DESCRIPTION:

This function returns a pointer to the name of the module being executed. Normally, ***argv[0]*** points to the same string, but when ***argv[]*** is not available, this function serves the purpose well.

SEE ALSO:

_errmsg().

printf(), fprintf(), sprintf() Formatted output conversion

SYNOPSIS:

```
#include <stdio.h>

printf(format [,arg ] ... )
char *format;

fprintf(fp, format [,arg ] ... )
FILE  *fp;
char *format;

sprintf(s, format [,arg ] ... )
char *s, *format;
```

DESCRIPTION:

These three functions are used to place numbers and strings in the output into formatted, human readable form.

The **printf()** subroutine places output on the standard output stream **stdout**. The **fprintf()**

subroutine places output on the named output fp. Note that the file pointer ***fp*** may be 0, 1, or 2 corresponding to stdin, stdout, and stderr or any valid pointer as returned by **fopen()**, **creat()**, **create()**, or **dup()**. The **sprintf()** subroutine places output in the string ***s***, followed by the character `\\0' (NULL).
NOTE: It is the programmers responsibility to insure that string ***s*** is large enough to hold the output of **sprintf()**.

Each of these functions converts, formats, and prints its arguments after the first under control of the ***format*** argument. The ***format*** argument is a character string which contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications. Each of these cause conversion and printing of the next successive ***arg***.

Following is the order in which a **printf()** conversion specification is presented:

% [-] [field_width] [.] [num_to_print] [len] conv_char

This order must be followed. Any of the optional (enclosed in brackets) conversion specifications may be omitted but, the order must remain the same. A period must appear before the "nbr of chars to print" or **printf()** will interpret the number to be the field width specification.

Each conversion specification is introduced by the percent sign character (%).
Following the %, there may be:

Zero or more flags, which modify the meaning of the conversion specification. If the character following the '%' is not a conversion character, that character is printed literally. The first uninterpretable character ends the conversion string.

An optional minus sign (-) which specifies left adjustment of the converted value in the indicated field.

An optional digit string specifying a field width. The field will be at least this wide and may be wider if the conversion requires it. If the converted value has fewer characters than the field width, it is blank-padded on the left (or right, if the left-adjustment indicator has been given) to make up the field width. If the field width digit string begins with a zero, zero-padding occurs instead of blank-padding.

An optional period (.) which serves to separate the field width from the next digit string.

An optional digit string specifying a precision which specifies the number of digits to appear after the radix character, for e and f conversions, or the maximum number of characters to be printed from a string.

The character ***l*** (lowercase 'l' (ell)) specifying that a following ***d***, ***o***, ***x***, ***X***, or ***u*** corresponds to a long integer ***arg***.

A character which indicates the type of conversion to be applied.

A field width or precision may be an asterisk (*) instead of a digit string. In this case an integer ***arg*** supplies the field width or precision.

The flag characters and their meanings are:

- : The result of the conversion is left-justified within the field.

+ : The result of a signed conversion always begins with a sign (+ or -).

blank : If the first character of a signed conversion is not a sign, a blank is prepended to the result. This implies that if the blank and plus sign (+) flags both appear, the blank flag is ignored.

: The value is to be converted to an alternative form. For ***c***, ***d***, ***s***, and ***u*** conversions, the flag has no effect. For ***o*** conversions, it increases the precision to force the first digit of the result to be a zero. For ***x*** or ***X*** conversions, a non-zero result has 0x or 0X prepended to it. For ***e***, ***E***, ***f***, ***g***, and ***G*** conversions, the result always contains a decimal point, even if no digits follow the point. A decimal point usually appears in the result of these conversions only if a digit follows it. For ***g*** and ***G*** conversions, trailing zeroes are not removed from the result.

The conversion characters and their meanings are

d, **o**, **x** - The integer ***arg*** is converted to decimal, octal, or hexadecimal notation respectively.

f - The float or double ***arg*** is converted to decimal notation in the style `[-]ddd.ddd' where the number of d's after the decimal point is equal to the precision specification for the argument. If the precision is missing, 6 digits are given; if the precision is explicitly 0, no decimal point and following characters are printed.

e, **E** - The float or double ***arg*** is converted in the style `[-]d.ddde(+-)dd' where there is one digit before the decimal point and the number after is equal to the precision specification for the argument; when the precision is missing, 6 digits are produced. When the argument is ***E***, the results are printed in uppercase characters.

g, **G** - The float or double arg is printed in style d, in style f, or in style e. The style used depends on the value converted and the shortest is printed. Style ***e*** is used only if the exponent resulting from the conversion is less than 4 or greater than the precision. Trailing zeroes are removed from the result. A decimal point appears onlyif it is followed by a digit. If the ***G*** form is used, the output is printed in uppercase characters.

c - The character ***arg*** is printed.

s - The ***arg*** is taken to be a string (character pointer) and characters from the string are printed until a null character or the number of characters indicated by the precision specification is reached; however if the precision is 0 or missing all characters up to a null are printed.

u - The unsigned integer ***arg*** is converted to decimal and printed. The result is in the range 0 through 65535 (for 6809/OS-9 only) or whatever the maximum integer size is on the system.

% - Print a `%'; no argument is converted.

A non-existent or small field width never causes truncation of a field. Padding takes place only if the specified field width exceeds the actual width. Characters generated by **printf()** are printed by **putc()**.

In the case of double or float conversions, the last digit printed is rounded.

CAVEATS:

In the ULTRIX-32 environment, **printf()** and **fprintf()** return 0 for success and EOF for failure. The **sprintf()** subroutine returns its first argument for success and EOF for failure.

In the System V environment, **printf()**, **fprintf()**, and **sprintf()** subroutines return the number of characters transmitted, not including the \\0 in the case of **sprintf()** or a negative value if an output error was encountered.

Within the OS-9 environment, in order to print long integers, the statement **pflinit()** must occur somewhere in the source code in order for the routines to print longs to be linked from the standard library. In addition, to print floats or double integers, the statement **pffinit()** must occur somewhere in the source code. Normally, either one or both of these statements are placed at the start of the source code file where printing of longs, floats, or doubles is required.

SEE ALSO:

putc(), scanf()

putc(), putchar(), putw() Put character or word on a stream

SYNOPSIS:

```
#include <stdio.h>

char putc(c, fp)
char c;
FILE *fp;

char putchar(c)

putw(w, fp)
FILE *fp;
```

DESCRIPTION:

The **putc()** routine appends the character ***c*** to the named output ***fp***. It returns the character written.

The **putchar(c)** routine is defined as a macro in the header file **<stdio.h>** and is the same as **putc(c, stdout)**.

The **putw()** routine appends word (that is, a two byte word, such as int), ***w*** to the output ***fp***. It returns the word written. The **putw()** routine neither assumes nor causes special alignment in the file.

Output via **putc()** is normally buffered except when buffering is disabled with **setbuf()** or when the standard error output path is used.

CAVEATS:

Because it is implemented as a macro, p**utchar()** treats its argument with side effects incorrectly. In particular, `putchar(*c++);' doesn't work as expected.

DIAGNOSTICS:

Putc() and **putchar()** return the character argument from a successful call or EOF upon error or end-of-file. Since EOF is a good integer, **ferror()** should be used to detect **putw()** errors.

SEE ALSO:

fclose(), fopen(), fread(), getc(), printf(), puts()

puts(), fputs() Put a string on an output stream

SYNOPSIS:

```
#include <stdio.h>

puts(s)
char *s;

fputs(s, fp)
char *s;
FILE *fp;
```

DESCRIPTION:

The **puts()** subroutine copies the null-terminated string s to the standard output stream **stdout** and appends a new line character.

The **fputs()** subroutine copies the null-terminated string s to the named output ***fp***.

Neither routine copies the terminal null character.

CAVEATS:

The **puts()** subroutine appends a new line, while **fputs()** does not. This inconsistency of the new-line being appended by **puts()** and not by **fputs()** is dictated by history and the desire for compatibility.

SEE ALSO:

fopen(), gets(), putc(), printf(), ferror() fread()

qsort() Quick sort

SYNOPSIS:

qsort(base, nel, size, compar)
char *base;
int (*compar)();

DESCRIPTION:

The **qsort()** subroutine is an implementation of the quick-sort algorithm. The first argument is a pointer to the base of the data; the second is the number of elements; the third is the size of an element in bytes; the last is the name of the comparison routine to be called with two arguments which are pointers to the elements being compared. The ***compar()*** routine must return an integer less than, equal to, or greater than 0 according as the first argument is to be considered less than, equal to, or greater than the second.

read(), readln() Rread from a file

SYNOPSIS:

read(pn, buffer, count)
char *buffer;
int pn, count;

readln(pn, buffer, count)
char *buffer;
int pn, count;

DESCRIPTION:

The path number ***pn*** is an integer which is one of the standard path number 0, 1, or 2, or the path number returned from a successful call to **open()**, **creat()**, **create()**, or **dup()**. ***Buffer*** is a pointer to memory space with at least ***count*** bytes of memory into which **read()** and **readln()** will put the data from the file.

It is guaranteed that at most ***count*** bytes will be read from the file, but often less will be, either because, for **readln()**, the file represents a terminal and input stops at the end of a line, or for both, the end-of-file marker has been reached.

ReadIn() causes "line editing" such as echoing to take place and returns once the first '\\n' is encountered in the input stream, or the number of bytes requested in ***count*** has been reached. **ReadIn()** is the preferred call for reading from the user's terminal.

Read() does not cause any line editing. See the OS-9 manual for a fuller description of the actions of these call.

DIAGNOSTICS:

Read() and **readln()** return the number of bytes actually read (0 at EOF), or -1 for a physical I/O error, a bad path number, or a ridiculous ***count***. The actual error (physical I/O or otherwise) can be determined by examining the global variable
errno.

NOTE:

EOF is not considered an error, and no error indication is returned. Zero is returned on EOF.

SEE ALSO:

open(), creat(), create(), dup()

realloc() Changes memory allocated by malloc()

SYNOPSIS:

```
char realloc(p, size)
char *p;
int size;
```

DESCRIPTION:

Realloc() takes a pointer such as that returned by **malloc()** and changes the size of the object. If the pointer argument is null, **Realloc()** degenerates into a **malloc()**.

SEE ALSO:

malloc() (Microware Manual)

sbrk(), ibrk() Request additional working memory

SYNOPSIS:

```
char *sbrk(increase)
char *ibrk(increase)

int increase;
```

DESCRIPTION:

Sbrk() requests an allocation from free memory and returns a pointer to its base, if successful.

Sbrk() requests the system to allocate "new" memory from outside the initial allocation.

Ibrk() requests memory from inside the initial memory allocation.

Users should read the Memory Management section of the "C" programming
Manual for a fuller explanation of the arrangement.

DIAGNOSTICS:

Sbrk() and **ibrk()** return a -1 if the requested amount of contiguous memory is not available

scanf(), fscanf(), sscanf() Convert formatted input

SYNOPSIS:

```
#include <stdio.h>

int scanf (format [, pointer...])
char *format;

int fscanf (fp, format [, pointer...])
FILE *fp;
char *format;

int sscanf (s, format [, pointer...])
char *s, *format;
```

DESCRIPTION:

The **scanf()** subroutine reads from the standard input stream **stdin**. The **fscanf()** subroutine reads from the named input ***fp***. The **sscanf()** subroutine reads from the character string ***s***. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string ***format*** described below, and a set of ***pointer*** arguments indicating where the converted input should be stored.

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

White-space characters which, except in two cases described later, cause input to be read up to the next non-white-space character.

An ordinary character (not %), which must match the next character of the input stream.

Conversion specifications, consisting of the character % , an optional assignment suppressing character \(** , an optional numerical maximum field width, an optional l " (ell) or " h indicating the size of the receiving variable, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the

variable pointed to by the corresponding argument, unless assignment suppression was indicated by \(**. The suppression of assignment provides a way of describing an input field which is to be skipped. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field
width, if specified, is exhausted. For all descriptors except left-bracket ([) and c,
white space leading an input field is ignored.

The conversion code indicates the interpretation of the input field. The corresponding pointer argument must usually be a restricted type. For a suppressed field, a pointer argument is not given. The following conversion codes are legal:

%
A single **%** is expected in the input at this point; no assignment is done.

d
A decimal integer is expected; the corresponding argument should be an integer pointer.

u
An unsigned decimal integer is expected; the corresponding argument should be an unsigned integer pointer.

o
An octal integer is expected; the corresponding argument should be an integer pointer.

x
A hexadecimal integer is expected; the corresponding argument should be an integer pointer.

\\f3e\\fP,\\f3f\fP,\\f3g\\fP
A floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a float . The input format for floating point numbers is an optionally signed string of digits, possibly containing a radix character, followed by an optional exponent field consisting of an **E** or an **e**, followed by an optional \(pl, \-, or space, followed by an integer.

s
A character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating \e0 , which is added automatically. The input field is terminated by a white-space character.

c
A character is expected; the corresponding argument should be a character pointer. The normal skip over white space is suppressed in this case; to read the next non-space character, use %1s . If a field width is given, the corresponding argument should refer to a character array; the indicated number of characters is read.

[
Indicates string data and the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters, which can be called the scanset, and a right bracket. The input field is the maximal sequence of input characters consisting entirely of characters in the scanset.

The circumflex \^, when it appears as the first character in the scanset, serves as a complement operator and redefines the scanset as the set of all characters not contained in the remainder of the scanset string. There are some conventions used in the construction of the scanset. A range of characters may be represented by the construct first\-last , thus [0123456789] may be expressed [0\-9].

Using this convention, first must be lexically less than or equal to last , or else the dash stands for itself. The dash also stands for itself whenever it is the first or the last character in the scanset. To include the right square bracket as an element of the scanset, it must appear as the first character (possibly preceded by a circumflex) of the scanset, and in this case it is not syntactically interpreted as the closing bracket. The corresponding argument must point to a character array large enough to hold the data field and the terminating \fB\\0\fR, which is added automatically. At least one character must match for this conversion to be considered successful.

The conversion characters ***d***, ***u***, ***o***, and ***x*** may be capitalized or preceded by ***l*** or ***h*** to indicate that a pointer to long or to short, rather than to int, is in the argument list. Similarly, the conversion characters ***e***, ***f***, and ***g*** may be capitalized or preceded by ***l*** to indicate that a pointer to double, rather than to float, is in the argument list. The "l" or "h" modifier is ignored for other conversion characters.

The **scanf()** subroutine conversion terminates at EOF, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

The **scanf()** subroutine returns the number of successfully matched and assigned input items. This number can be zero in the event of an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, EOF is returned.

EXAMPLE:

The call:

int i, n; float x; char name[50];

n = scanf("%d%f%s", &i, &x, name);

with the input line: 25 54.32E\-1 thompson

assigns to n the value 3, to i the value 25, to x the value 5.432,
and name will contain thompson.

Or:

int i; float x; char name[50];

scanf("%2d%f%\ %[0\-9]", &i, &x, name);

with input: 56789 0123 56a72 will assign 56 to ***i***, 789.0 to ***x***, skip 0123, and place the string 560 in ***name*** .

The next call to **getchar()** will return ***a*** .For further information, see **getc()** .

CAVEATS:

The success of literal matches and suppressed assignments is not directly determinable.

Trailing white space (including a new-line) is left unread unless matched in the control string.

DIAGNOSTICS:

These functions return EOF on end of input and a short count for missing or illegal data items.

SEE ALSO:

atof(), getc(), printf()

setbuf() Fix file buffer

SYNOPSIS:

```
#include <stdio.h>

setbuf (fp, buffer)
FILE *fp;
char buffer;
```

DESCRIPTION:

When the first character is written to or read from a file after it has been opened by **fopen()**, a buffer is obtained from the system, if required, and assigned to the file pointer f***p***. **Setbuf()** may be used to forestall this automatic buffer assignment by assigning a user buffer to the file.

Setbuf() must be used after the file has been opened and before any I/O has taken place.

The buffer must be of sufficient size and a value for a manifest constant, **BUFSIZ**, is defined in the header file for use in declaration.

If the ***buffe***r argument is NULL (0), the file stream becomes unbuffered and characters are read or written singly.

NOTE that the standard error output defaults to unbuffered while the standard output default to buffered output.

SEE ALSO:

fopen(), getc(), putc()

setime(), getime() Set and get system time

SYNOPSIS:

```
#include <time.h>

setime(buffer)
getime(buffer)

struct sgtbuf *buffer;      /* defined in time.h */
```

DESCRIPTION:

Setime() sets the system time depending on the values in the structure ***buffer*** as defined in **<time.h>**. These values must be set prior to calling **setime()** or the system time will be set to an unknown state.

Getime() reads the system time and returns the information in the structure ***buffer***. Reading the structure elements will then yield the desired values.

An example to read the time structure might be:

```
struct sgtbuf timepacket;

getime(timepacket);
printf("The year is: %d\n", timepacket.t_year);
printf("The day is:  %d\n", timepacket.t_day);
```

SEE ALSO:

time(), ctime(), time(), o2utime(), u2otime()

setjmp(), longjmp() Nonlocal goto another function

SYNOPSIS:

```
#include <setjmp.h>

setjmp(env)
jmp_buf env;

longjmp(env, val)
jmp_buf env;
```

DESCRIPTION:

These routines are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Goto in C has a scope only in the function in which it is used; i.e., the label which is the object of the **goto** may only be in the same function. Control can only be transferred elsewhere by means of the function call, which, fo course, returns to the caller. In certain abnormal situations a programmer would perfer to be able to start some section of code again, but this would mean returning up a ladder of function calls with error indications all the way.

Setjmp() is used to "mark" a point in the program where a subsequent **longjmp()** can reach. It places in the buffer, defined in the header file **<setjmp.h>**, enough information for the **longjmp()** to restore the environment to that existing at the time **setjmp()** is called.

The **setjmp()** subroutine saves its stack environment in ***env*** for later use by **longjmp()**. It returns value 0.

The **longjmp()** subroutine restores the environment saved by the last call of **setjmp()**. It then returns in such a way that execution continues as if the call of **setjmp()** had just returned the value ***val*** to the function that invoked **setjmp()**, which must not itself have returned in the interim. However, **longjmp()** cannot cause **setjmp()** to return the value 0. If **longjmp()** is invoked with a ***val*** of 0, **setjmp()** will return 1. All accessible data have values as of the time **longjmp()** was called.

CAVEATS:

The **setjmp()** subroutine does not save current notion of whether the process is executing as a result of a signal. The result is that a LONGJMP to some place as a result of a signal leaves the signal state incorrect.

In addition, the variable ***env*** MUST be globally declared.

DIAGNOSTICS:

Setjmp() returns a zero (0) if the call is the the first made in the current program run. If a one (1) is returned, then it must be a **longjmp()** returning from some deeper level in the program.

SEE ALSO:

signal(), intercept()

setmem() Fills memory with a character

SYNOPSIS:

```
setmem(start, count, fill)
char *start
int count
char fill
```

DESCRIPTION:

Setmem() fills ***count*** bytes of memory beginning at the location pointed to by ***start*** with the ASCII character contained in ***fill***.

No value is returned making this a type VOID function.

EXAMPLE:

```
/* Initialize an 80 character string to all NULLS */

count = 80;
fill = '0x00'

setmem(start, count, fill)
```

SEE ALSO:

memset()

setpr() Set process priority

SYNOPSIS:

```
setpr(pid, priority)
int pid, priority;
```

DESCRIPTION:

Setpr() sets the process identified by ***pid*** (process ID) to have a priority of ***priority***. The lowest priority is 0 and the highest is 255.

A currently running process cannot change the priority of another running process if the two process' do not share ownership. In addition, a process, if not owned by the super user, cannot upgrade its priority to a level higher than the parent process that created it with the OS-9 system call. However, a process owned by the super user, or any system process, can change the priority of any other running process to any level.

DIAGNOSTICS:

A -1 will be returned if the process does not have the same user ID as the calling process.

allocset(), addc2set(), adds2set() rmfmset(), smember(), union() sintersect(), sdifference(), copyset()

Dupset Set operations

SYNOPSIS:

```
char *allocset(s, c)
char *s, *p;

char *addc2set(s, c)
char *s c;

char *adds2set(s, p)
char *s p;

char *rmfrmset(s, c)
char *s, c;

smember(s, c)
char *s, c;

char *sunion(s1, s2)
char s1[], s2[];

char *sintersect(s1, s2)
char s1[], s2[];

char *sdifference(s1, s2)
char s1[], s2[];

char *copyset(s1, s2)
char s1[], s2[];

char *dupset(s)
char s[];
```

DESCRIPTION:

Allocset() allocates memory for a set consisting of an array of 32 bytes (256 bits). If successful, it returns a pointer to the set, or null if not successful. This array is then operated on with the following functions.

Add2set() adds the character ***c*** to the set ***s***. No error is possible. Adding a single character or any value in the range 0 - 255 decimal is the same as ORing the bit that corresponds to the numeric value of that character, i.e., adding a character 'A' to the set will set bit number $41, 65 decimal.

Add2set() adds the string ***p*** to the set ***s***. No error is possible. The string ***p*** is added to the set in

that same manner as **add2set()** above, but the entire string of bits is added in a bit-by-bit progression.

Rmfrmset() removes character ***c*** from the set ***s***. Again, no error is possible. Removing a character from the set amounts to ANDing the bit at the position corresponding to the numeric value of the character as in **add2set()** above, but is the reverse procedure.

Smember() returns TRUE if character ***c*** is a member of set ***s***, or returns FALSE if it is not a member.

Sunion() merges a second set ***s2*** into the first set ***s1***.

Sintersect() returns any elements that only exist in both sets ***s1*** and ***s2***.

Sdifference() returns unigue elements of both sets that are in the first set, ***s1***.

Copyset() duplicates the second set, ***s2***, into the first set, ***s1***.

Dupset() allocates memory for a new set, and then copies set ***s*** into that memory area.

setstat() OS-9 set status system calls

SYNOPSIS:

```
#include <os9.h>
#include <sgstat.h>

setstat(SS_OPT, filenum, buffer)
int code, filenum;
struct sgbuf *buffer;

setstat(SS_SIZE, filenum, size)
int code, filenum;
long *size;

setstat(SS_RESET, filenum, code)
int code, filenum, code;

setstat(SS_WTRK, filenum, buffer, track_number, side/density)
int code, filenum;
char *buffer;
int track_number, side/density;

setstat(SS_FRZ, filenum)
int code, filenum;

setstat(SS_SQD, filenum)
int code, filenum;
```

```
setstat(SS_DCMD, code, filenum, parm1, parm2, parm3)
int code, code, filenum, parm1, parm2, parm3;

#include <direct.h>
setstat(SS_FD, filenum, buffer)
int code, filenum;
struct fildes *buffer;

setstat(SS_TICK, filenum, count)
int code, filenum, count;

setstat(SS_LOCK, filenum, position)
int code, filenum;
long position;

setstat(SS_RELEA, filenum)
int code, filenum;

setstat(SS_BLKRD, filenum, buffer, track_sector, track_den)
int code, filenum;
char *buffer;
int track_sector, track_den/side/density;

setstat(SS_BLKWR, filenum, buffer, track_sector, track_den)
int code, filenum;
char *buffer;
int track_sector, track_den/side/density;

setstat(SS_SSIG, filenum, code)
int code, filenum, code;
```

DESCRIPTION:

Most of these calls are equivalent to the **_ss_XXX** calls described elsewhere in these documents. While the **_ss_XXX** calls are provided for compatibility with source code generated for OSK systems, the calls listed below are those normally used with 6809/OS-9.

The following descriptions do not include the full syntax for each function call. See the list above under Synopsis.

Setstat(SS_OPT) copies the buffer pointed to be ***buffer*** into the options section of the path descriptor opened.

Generally, a program will fetch the options with the **getstat(SS_OPT)** function, change the desired values, and then update the path options with the s**etstat(SS_OPT)** function. The structures **SGBUF** declared in **<sgstat.h>** provides a convenient means to access the individual option values.

If an error occurs, a -1 is returned and the appropriate error code is placed in ***errno***.

It is a common practice to preserve a copy of the original path descriptor options so a program can restore them prior to exiting. The option changes take effect on the currently open path and any path created with the **I$DUP** system call.

SETSTAT(SS_SIZE) is used to change the size of a file opened on ***filenum***. The size change is immediate.

If the size of the file is decreased, the freed sectors are returned to the system. If the size is increased, sectors are added to the file with the contents of those sectors being undefined.

If an error occurs, this function returns the value -1 and the error code is placed in the global variable ***errno***.

This function works only on **RBF** devices.

Getstat(SS_RESET) restores the disk drive head to Track 00 in preparation for formatting and error recovery.

If an error occurs, this function returns -1 as its value and the error code
is placed in the global variable ***errno***.

This call works only on **RBF** devices.

Setstat(SS_WRTK) performs a write-track operation on a disk drive. It is essentially a direct hook into the driver's write-track entry point. ***track_number*** is the desired track number to write, and ***side/density*** is the desired side of the disk upon which to write. When the write is performed, the image contained in and pointed to by ***buffer*** is written out to the disk.

If an error occurs, the value -1 is returned and the error code is placed in the global variable ***errno***.

This function works only on **RBF** devices. Additional information on how it works can be obtained from examining the FORMAT utility or a device driver.

WARNING! If ***track_number*** is set to zero when this function is called, the entire disk, floppy or hard disk, will be formatted.

Setstat(SS_FRZ) inhibits the reading of LSN 0 variables which define the disk format. This enables the reading of non-standard disks.

This is a very little used function that has been deleted from most new documentation and, in fact, is not supported by Microware any more. It is included in this library solely for compatibility with older programs that may call it. Consult your manuals for an explanation of its use.

Setstat(SS_SPT) sets a different number of tracks so that non-standard disks can be read. This is not an often used call, as with **SS_FRZ** above. Consult your manual for more details.

This call does not return any information.

Setstat(SS_SQD) starts the power down sequence for hard drives that have this feature. Since this sequence is hardware dependent, consult your user documents for more details. The device that is opened with ***filenum*** will be the device the sequence works on.

This call does not return any information.

Setstat(SS_DCMD) sends direct commands to the disk controller for specific instructions. Since parameters and commands are also hardware dependent, consult your disk controller's documentation and the specifications for the disk controller chip.

The exit conditions of this call vary depending on the hardware device.

Setstat(SS_FD) places a copy of the **RBF** file descriptor sector of the file opened into the buffer pointed to by ***buffer***. A maximum of ***count*** bytes are copied. The structure ***fildes***, declared in **<direct.h>**, provides a convenient method to access the file descriptor information.

If and error occurs, this function returns -1 as its value and the error code is placed in the global variable ***errno***.

Be sure the buffer is large enough to hold all the options, or at least ***count*** bytes. This call is effective only on **RBF** devices. Declaring the buffer as type "***struct fildes***" is perfectly safe as this structure is predefined to be large enough to hold all the file descriptor information.

Setstat(SS_TICK) may be used to cause an error (E$LOCK) to be returned to the process if the conflict still exists after a specific number of clock ticks have elapsed.

The argument ***count*** specifies the number of ticks to wait if the record-lock conflict occurs with the file open on ***path***. A tick count of zero (the default on **RBF** devices), causes a sleep until the record is released. A tick count of one means if the record is not released immediately, an error is to be returned.

If an error occurs, the value -1 is returned and the error code is placed in the global variable ***errno***.

Setstat(SS_LOCK) locks out a file open on ***filenum*** at the offset from the file beginning at offset ***position***, so another process cannot read past that point.

If an error occurs, the function returns the value -1 and the error code is placed in the global variable ***errno***.

Setstat(SS_RELEA) releases a file that was locked using **SS_LOCK** above.

Setstat(SS_BLKRD) reads one block of data from a disk file opened on ***filenum***. The parameters passed determine the actual sector number and side of the disk. The data is read into a buffer pointed to by ***buffer***.

This function is VERY hardware dependent and the user should know the size of a disk block on his/her system before using it. Typically, on an 8-bit machine, the block size will be 256 bytes while a 16-bit machine will usually have a block size of 512 bytes. UNIX hardware has a block size of 1024 bytes. In addition, not all device drivers support this call. Again, be sure before you use it.

Consult your hardware documentation for more details.

Setstat(SS_BLKWR) is the reverse of **SS_BLKRD** above, writing out one block of data.

Setstat(SS_SSIG) sets up a signal to be sent to the calling process when an interactive device has data ready. When data is received on the device indicates by***filenum***, the signal ***code*** is sent to the calling process.

SS_SSIG must be called each time the signal is sent if it is to be used again.

The device is considered busy, and will return an error, if any read request arrives before the signal is sent. Write requests are allowed to the device while in this state.

If an error occurs, the function returns the value -1 and the error code is placed in the global variable ***errno***.

NOTE:

All the above calls require an OS-9 path number for ***filenum***, and NOT a C iob file descriptor pointer.

SEE ALSO:

I$GetStt system call - Microware Manual, _gs_xxx calls

setuid() Set user ID

SYNOPSIS:

```
setuid(uid)
int uid;
```

DESCRIPTION:

This call is used to set the user ID for the current process.

Setuid() only works if the caller is the super user.

DIAGNOSTICS:

A zero is returned from a successful call, and -1 on error.

SEE ALSO:

getuid()

signal() Catch or ignore interrupts

SYNOPSIS:

#include <signal.h>

(*signal(interrupt, address))()
(*address)();

DESCRIPTION:

This call is a comprehensive method of catching or ignoring signals sent to the current process. Notice that **kill()** does the sending of signals and **signal()** does the catching.

Normally, a signal sent to a process causes it to terminate with the status of the signal. If, in advance of the signal, this system call is used, the program has the choise of ignoring the signal or designating a function to be executed when the signal is received. Different functions may be designated for different signals.

The values for ***address*** have the following meanings:

0 = reset to the default, i.e., abort when received
1 = ignore - this applies until reset to another value

Otherwise: taken to be the address of a C function which is to be executed upon receipt of the signal.

If the latter case is chosen, when the signal is received by the process the address is reset to 0, the default, before the function is executed. This means that is the next signal received should be caught then another call to **Signal()** must be made immediately. This is normally the first action taken by the **Interrupt** function. The function may access the signal number which caused its execution by looking at its argument. On completion fo this function the program resumes execution at the point where it was interrupted by the signal.

EXAMPLE:

Suppose a program needs to create a temporary file which should be deleted before exiting. The body of the program might contain fragments like this:

```
pn = creat("temp",3);
signal(2,intrupt);
signal(3,intrupt);
write(pn,string,count);
close(pn);
```

```
    unlink("temp");
    exit(sig);
}
```

In this case, as the function will be exiting before another signal is receive, it is unnecessary to call **signal()** again to reset its pointer. Note that either the function **intrupt()** should appear in the source code before the call to **signal()**, or it should be pre-declared.

The signals used by OS-9 are define in the header file **<signal.h>** as follows:

```
/* OS-9 Signals

#define SIGKILL 0
#define SIGWAKE 1
#define SIGQUIT 2
#define SIGINT  3

/* special addresses */

#define SIG_DFL 0
#define SIG_IGN 1
```

Please note that there is another method of trapping signals, namely **intercept()**. However, since **signal()** and **intercept()** are mutually incompatible, calls to both of them must not appear in the same program. The linker will prevent the creation of an executable program in which both are called by aborting with an ***entry*** name clash error for **_sigint**.

SEE ALSO:

intercept(), kill(), OS-9 Shell command "kill"

skipbl() Skips spaces and tabs within a string

SYNOPSIS:

```
char *skipbl(s)
char *s
```

DESCRIPTION:

Skipbl() skips over all spaces (0x20) and tabs (0x09) in a string and returns an updated pointer to the next non-blank character.

Since the function returns a pointer, is must be declared prior to calling as a type CHAR function.

.**EXAMPLE:**

Before entering, the pointer is positioned as:

```
Now   is   the   time
     ^
```

skipbl("Now is the time")

returns the pointer as:

```
Now   is   the   time
      ^
```

SEE ALSO:

skipwd()

skipwd() Skips words within a string

SYNOPSIS:

char *skipwd(s)
char *s

DESCRIPTION:

Skipwd() skips over all non-blank characters in a string and returns an updated pointer to the next space (0x20) found

Since the function returns a pointer, is must be declared prior to calling as a type CHAR function.

EXAMPLE:

Before entering, the pointer is positioned as:

```
Now   is   the   time
 ^
```

skipwd("Now is the time")

returns the pointer as:

```
        Now   is   the   time
              ^
```

SEE ALSO:

skipbl()

_ss_attr() Set file attributes

SYNOPSIS:

```
#include <modes.h>

int _ss_attr(path, attr)
int path;
short attr;
```

DESCRIPTION:

_ss_attr() changes a disk file's attributes. The current attributes of a file can be determined with the **_gs_gfd()** function. The attributes of a file can be changed only by the owner of the file or the super user.

The attributes as selected in the word ***attr*** are set in the file opened on ***path***. The header file **<modes.h>** defines the valid mode values used in ***attr***.

NOTE:

This function is effective even if the owner or super user does not have write permission to the file. It is not permitted to set the directory bit of a non-directory file, or to clear the directory bit of a directory that is not empty.

DIAGNOSTICS:

If an error occurs, the function returns the value -1 and the error code is placed in the global variable ***errno***.

SEE ALSO:

I$SetStt system call, Microware manual.

_ss_lock() Set file lock status

SYNOPSIS:

```
int _ss_lock(path, locksize)
int path;
long locksize;
```

DESCRIPTION:

_ss_lock locks out a section of a file open on ***path*** from the current file position up to the number of bytes specified by ***locksize***.

If the **locksize** is zero, all locks (record lock, EOF lock, and file lock) are removed. If a locksize of 0xFFFFFFFF is requested, the entire file is locked regardless of where in the file the file pointer is. This is a special type of file lock that remains in effect until released by **_ss_lock(path,0),** a read or write of zero or more bytes, or the file is closed.

DIAGNOSTICS:

If an error occurs, the function returns the value -1 and the error code is placed in the global variable ***errno***.

SEE ALSO:

I$SetStt system call, Microware manual.

_ss_opt() Set file descriptor options

SYNOPSIS:

```
#include <sgstat.h>

int _ss_opt(path, buffer)
int path;
struct sgbuf *buffer;
```

DESCRIPTION:

_ss_opt() copies the buffer pointed to be ***buffer*** into the options section of the path descriptor opened on ***path***.

Generally, a program will fetch the options with the **_gs_opt()** function, change the desired values, and then update the path options with the **_ss_opt()** function. The structure ***sgbuf*** declared in **<sgstat.h>** provides a convenient means to access the individual option values.

NOTE:

It is a common practice to preserve a copy of the original path descriptor options so a program can restore them prior to exiting. The option changes take effect on the currently open path and any path created with the I$DUP system call.

DIAGNOSTICS:

If the path is invalid, **_ss_opt()** returns -1 and the appropriate error code is placed in ***errno***.

SEE ALSO:

I$SetStt system call - Microware Manual, _gs_opt().

_ss_pfd() Set and write file descriptor

SYNOPSIS:

```
#include <direct.h>

int _ss_pfd(path, buffer)
int path;
struct fildes *buffer;
```

DESCRIPTION:

_ss_pfd() will copy certain bytes from the buffer pointed to by ***buffer*** into the file descriptor sector of the file opened on ***path***. The buffer is usually obtained from the **_gs_gfd()** function. Only the owner ID, the modification date, and the creation date is changed.

The structure ***FILDES*** declared in **<direct.h>** provides a convenient means to access the file descriptor information.

NOTE:

The buffer must be at least 16 bytes long or garbage could be written into the file descriptor sector. The owner ID field can be changed only by the super-user. It is impossible to change the file attributes with the call. Instead, use the **_ss_attr()** function.

SEE ALSO:

I$SetStt system call, Microware Manual, _gs_pfd().

_ss_rel() Release a pending signal

SYNOPSIS:

```
int _ss_rel(path)
int path;
```

DESCRIPTION:

_ss_rel() cancels the signal to be sent from a device on data ready. The function **_ss_ssig()** enables a device to send a signal to a process when data is available on the device.

DIAGNOSTICS:

If an error occurs, the function returns the value -1 and the appropriate error value is placed in the global variable ***errno***.

NOTE:

The signal request is also canceled when the issuing process dies or closes the path to the device. This feature exists only on **SCF** devices.

SEE ALSO:

I$SetStt system call, Microware manual, _ss_ssig().

_ss_rest() Restore disk drive head to track zero

SYNOPSIS:

int _ss_rest(path)
int path;

DESCRIPTION:

_ss_rest() causes an **RBF** device to restore the disk head to track zero. It is usually used prior to disk formatting and for error recovery.

DIAGNOSTICS:

If an error occurs, the function returns -1 and the appropriate error code is placed in the global variable ***errno***.

SEE ALSO:

I$SetStt system call - Microware manual.

_ss_ssig() Set a signal

SYNOPSIS:

int _ss_ssig(path, sigcode)
int path;
short sigcode;

DESCRIPTION:

_ss_ssig() sets up a signal to be sent to the calling process when an interactive device has data ready. When data is received on the device indicates by ***path***, the signal ***sigcode*** is sent to the calling process.

_ss_ssig() must be called each time the signal is sent if it is to be used again.

The device is considered busy, and will return an error, if any read request arrives before the signal is sent. Write requests are allowed to the device while in this state.

NOTE:

This feature exists only on **SCF** devices.

DIAGNOSTICS:

If an error occurs, the function returns the value -1 and the error code is placed in the global variable ***errno***.

SEE ALSO:

I$SetStt system call, Microware manual, _ss_rel().

_ss_size() Change file size

SYNOPSIS:

```
int _ss_size(path, size)
int path;
long size;
```

DESCRIPTION:

_ss_size() is used to change the size of a file opened on ***path***. The size change is immediate.

If the size of the file is decreased, the freed sectors are returned to the system. If the size is increased, sectors are added to the file with the contents of those sectors being undefined.

If an error occurs, **_ss_size()** returns the value -1 and the error code is placed in the global variable ***errno***.

NOTE:

This function works only on **RBF** devices.

SEE ALSO:

I$SetStt system call, Microware manual.

_ss_tiks() Set timeout tick count

SYNOPSIS:

int _ss_tiks(path, tickcnt)
int path;
int tickcnt;

DESCRIPTION:

If a read or write request is made for a part of a file that is locked out by another user, **RBF** normally sleeps indefinitely until the conflict is removed. **_ss_tiks()** may be used to cause an error (E$Lock) to be returned to the process if the conflict still exists after a specific number of clock ticks have elapsed.

The argument ***tickcnt*** specifies the number of ticks to wait if the record-lock conflict occurs with the file open on ***path***. A tick count of zero (the default on **RBF** devices), causes a sleep until the record is released. A tick count of one means if the record is not released immediately, an error is to be returned.

If an error occurs, **_ss_tiks()** returns the value -1 and the error code is placed in the global variable ***errno***.

NOTE:

This feature exists only on **RBF** devices.

SEE ALSO:

I$SetStt system call, Microware manual, _ss_rel().

_ss_wtrk() Write a disk drive track

SYNOPSIS:

int _ss_wtrk(path, trkno, siden, trkbuf)
int path;
int trkno, siden
char *trkbuf

DESCRIPTION:

_ss_wrtk() performs a write-track operation on a disk drive. It is essentially a direct hook into the driver's write-track entry point.

The argument ***path*** is the path on which the device is opened. ***Trkno*** is the desired track number to write, and ***siden*** is the desired side of the disk on which to write. When the write is performed, the image contained in and pointed to by ***trkbuf*** is written out to the disk.

If an error occurs, **_ss_wtrk()** returns the value -1 and the error code is placed in the global variable ***errno***.

NOTE:

This function works only on **RBF** devices. Additional information on how it works can be obtained from examining the **FORMAT** utility or a device driver. Also, note there is a difference in the syntax of this call from that used in OSK systems.

>>>> W A R N I N G <<<<

If the variable ***trkno*** is set to zero when this function is called, the **ENTIRE** disk, floppy or hard disk, will be formatted. Care should be taken to insure that ***trkno*** has a non-zero value prior to entering this function.

SEE ALSO:

I$SetStt system call, Microware manual.

stacksize(), freemem() Get stack reservation size

SYNOPSIS:

stacksize()
freemem()

DESCRIPTION:

The stack area is the currently reserved memory for exclusive use of the stack. As each C function is entered, a routine in the system interface is called to reserve enough stack space for the use of the function with an additional 64 bytes. The 64 bytes are for the use of user-written assembly language code functions and/or the system interface and/or arithmetic routines. A record is kept of the lowest address so far granted for the stack. If the area requested would not bring this lower, then the C function allowed to proceed. If the new lower limit would mean that the stack area would overlap the data area, then the program stops with the message

**** STACK OVERFLOW ****

on the standard error **outpath**. Otherwise, the new lower limit is set, and the C function resumes as before.

If the stack check code is in effect, then a call to **stacksize()** will return the maximum number of bytes of stack used at the time of the call. This call can be used to determine the stack size required by the program.

Freemem() will return the number of bytes of the stack that has not been used.

CAVEATS:

Of course, all this depends on if the program was compiled with stack checking enabled. If stack

checking was disabled (cc -s code.c), then no stack checking occurs.

SEE ALSO:

ibrk(), sbrk(), variables ***memend*** and value **end**.

strass() — Byte by byte copy

SYNOPSIS:

```
_strass(s1, s2, count)
char *s1, *s2;
int count;
```

DESCRIPTION:

Until such time as the compiler can deal with structure assignment, this function is useful for copying one structure to another.

Count bytes are copied from memory location ***s2*** to memory location ***s1*** regardless of the contents.

strcat(), strucat(), strncat() — String operations
strcmp(), strucmp(), strncmp(), strnucmp(),
strcpy(), strucpy(), strncpy(), strlen()
strchr(), strrchr(), strpbrk(), strspn()
strcspn(), strtok(), strclr(), strend()
reverse(), pwcryp(), index(), rindex()

SYNOPSIS:

```
#include <string.h>

char *strcat (s1, s2)
char *s1, *s2;

char *strucat (s1, s2, n)
char *s1, *s2;
int n;

char *strncat (s1, s2, n)
char *s1, *s2
int n;

int strcmp (s1, s2)
char *s1, *s2;
```

```
int strucmp (s1, s2)
char *s1, *s2;

int strncmp (s1, s2, n)
char *s1, *s2;
int n;

int strnucmp (s1, s2, n)
char *s1, s2;
int n;

char *strcpy (s1, s2)
char *s1, *s2;

char *strucpy (s1, s2)
char *s1, s2;

char *strncpy (s1, s2, n)
char *s1, *s2;
int n;

int strlen (s)
char *s;

char *strchr (s, c)     /* aka index() */
char *s1;
int c;

char *strrchr (s, c)    /* aka rindex() */
char *s1;
int c;

char *strpbrk (s1, s2)
char *s1, *s2;

int strspn (s1, s2)
char *s1, *s2;

int strcspn (s1, s2)
char *s1, *s2;

char *strtok (s1, s2)
char *s1, *s2;

char *strclr (s, c)
char *s;
int c;
```

```
char *strend (s)
char *s;

char *reverse (s)
char *s;

char *pwcryp (s)
char *s;

char *index(s, ch)
char *s, ch;

char *rindex(s, ch)
char *s, ch;
```

DESCRIPTION:

The arguments ***s1***, ***s2***, and ***s*** point to strings (arrays of characters terminated by a NULL character). The functions **strcat()**, **strucat()**, **strncat()**, **strcpy()**, **strucpy()** and **strncpy()** all alter ***s1***. These functions do not check for overflow of the array pointed to by ***s1***.

Strcat() appends a copy of string ***s2*** to the end of string ***s1***. **Strcat()** appends at most ***n*** characters. **Strucat()** is the same as **strcat()** but makes all characters uppercase. Each returns a pointer to the null-terminated result.

Strcmp() compares its arguments and returns an integer less than, equal to, or greater than 0, according as ***s1*** is lexicographically less than, equal to, or greater than ***s2***. **Strucmp()** functions in the same way but makes all characters uppercase before comparing. **Strncmp()** makes the same comparison but looks at at most ***n*** characters. **Strnucmp()** is the same except it makes all characters uppercase prior to the comparison.

Strcpy() copies string ***s2*** ***s1***, stopping after the null character has been copied. **Strucpy()** performs the same except all characters are made uppercase. **Strncpy()** copies exactly ***n*** characters, truncating ***s2*** or adding nul**l** characters to ***s1*** as necessary. The result will not be null-terminated if the length of ***s2*** is ***n*** or more. Each function returns ***s1***.

Strlen() returns the number of characters in ***s***, not including the terminating null characters.

Strchr() (strrchr()) returns a pointer to the first (last) occurrence of character ***c*** in string ***s***, or a null pointer if ***c*** does not occur in the string. The null character terminating a string is considered to be part of the string.

Strpbrk() returns a pointer to the first occurrence in string ***s1*** of any character from ***s2***, or a nul**l** pointer if no character from ***s2*** exists in ***s1***.

Strspn() (strcspn()) returns the length of the initial segment of string ***s1*** which consists entirely of characters from (or not from) string ***s2***.

Strtok() considers the string ***s1*** to consist of a sequence of zero or more text tokens separated by

spans of one or more characters from the separator string ***s2***. The first call (with pointer ***s1*** specified) returns a pointer to the first character of the first token, and will have written a null character into ***s1*** immediately following the returned token. the function keeps track of its position in the string between separate calls, so that subsequent calls (which must be made with the first argument a NULL pointer) will work through the string ***s1*** immediately following that token. In this way, subsequent calls will work through the string ***s1*** until no tokens remain. The separator string ***s2*** may be different from call to call. When no token remains in ***s1***, a null pointer is returned.

Strclr() sets at the most c characters in string s, but not including the null-terminator, to SPACES (Ox20).

Strend() returns a pointer to the end of string ***s***.

Reverse() reverses the characters of string ***s*** in memory and then returns ***s***.

Pwcryp() encrypts and returns string ***s***.

Index() returns a pointer to the first occurance of ***ch*** in ***s*** or **null** if not found. **Index()** is functionally the same as **STRCHR** except ***ch*** is of type CHAR.

Rindex() returns a pointer to the last occurance of ***ch*** in ***s*** or NULL if not found. **Rindex()** is functionally the same as **strrchr()** except ***ch*** is of type CHAR.

Both **index()** and **rindex()** are maintained for backward compatibility with older UNIX System V releases.

NOTE:

For user convenience, all these functions are declared in the optional **<string.h>** header file.

BUGS:

Strcmp() and **strncmp()** use native character comparison, which is unsigned on some machines. Thus, the sign of the value returned when one of the characters has its high order bit set is implementation-dependent.

Character movement is performed differently in different implementations.
Thus, overlapping moves may yield surprises.

CAVEATS:

Strcat() and **strcpy()** have no means of checking that the space provided is large enough. It is the user's responsibility to ensure that string space does not overflow.

SEE ALSO:

findstr()

time() Get the time

SYNOPSIS:

long time ((char *) 0)
long time (tloc)
long *tloc;

DESCRIPTION:

Time() returns the value of time in seconds since 00:00:00 GMT, January 1, 1970.

If ***tloc*** (taken as an integer) is non-zero, the return value is also stored in the location to which ***tloc*** points.

Upon successful completion, **time()** returns the value of time. No error is possible here as **time()** always returns a value.

The value returned is suited for use with the **ctime()** function.

EXAMPLES:

To get the system time value:

```
long curr_time;
curr_time = time ((char *)0);
```

or

```
long curr_time;
time (&curr_time);
```

SEE ALSO:

ctime(), o2utime(), u2otime().

tsleep() Put process to sleep

SYNOPSIS:

```
tsleep(ticks)
int ticks;
```

DESCRIPTION:

Tsleep() deactivates the calling process for a specified number of system clock ***ticks*** or if ***ticks*** is zero indefinitely. A tick is system dependent, but is usually 100ms for most OS-9 systems except the Color Computer where it is 1/60th of a second.

SEE ALSO:

sleep()

unbrk() Returns memory

SYNOPSIS:

unbrk(pnt)
char *pnt;

DESCRIPTION:

Unbrk() returns memory that was allocated using **sbrk()**. **Sbrk()** requests the system to allocate more memory than was originally allocated. **Unbrk()** simply returns that additional allocation.

DIAGNOSTICS:

This function returns -1 if an error occurs and 0 upon success.

SEE ALSO:

sbrk()

ungetc() Put a character back into the input buffer

SYNOPSIS:

#include <stdio.h>

ungetc (ch, fp)
char ch;
FILE *fp;

DESCRIPTION:

This function alters the state of the input file buffer such that the next call of **getc()** returns the character ***ch***.

Only one character may be puched back, and at least one character must have been read from the file before a call to **ungetc()** is made.

Fseek() erases and characters pushed back.

DIAGNOSTICS:

Ungetc() returns its character argument unless no pushback could occur, in which case EOF is

returned.

SEE ALSO:

getc(), fseek()

o2utime(), u2otime() — Converts date and time from OS9 to UNIX format

SYNOPSIS:

```
#include <utime.h>
#include <time.h>

long o2utime(tp)
struct sgtbuf *tp;

u2otime(tp,tmp)
struct sgtbuf *tp;
struct tm     *tmp;
```

DESCRIPTION:

O2utime() converts a six character OS9 time into a UNIX style long as in the **time()** function.

U2otime() copies a broken down UNIX style time from structure ***tmp*** into the OS9 style ***sgtbuf*** structure.

SEE ALSO:

time(), ctime()

unlink() — Remove a directory entry

SYNOPSIS:

```
unlink(fname)
char *fname;
```

DESCRIPTION:

Unlink() deletes the directory entry whose name is pointed to by ***fname***. If the entry was the last link to the file, then the file itself is deleted and the disc space occupied made available for re-use. If, however, the file is open or in any active task, the deletion of the actual file is delayed until the file is closed.

DIAGNOSTICS:

Zero is returned from a successful call, -1 if the file does not exist, if its directory is write-protected, the pathname cannot be searched or if the file itself is a non-empty directory or a device.

SEE ALSO:

unlinkx() OS-9 "kill" command

unlinkx - deletes a directory entry

SYNOPSIS:

unlinkx (fname, mode)
char *fname, mode;

DESCRIPTION:

Unlinkx() performs essentially the same function as **unlink()**. However, if the attribute of the file to remove from the directory, as described by ***mode***, is an executable file, then the current execution directory is used. If the mode is not executable, then the current data directory is used.

DIAGNOSTICS:

This function returns -1 if an error occurs and 0 upon success.

SEE ALSO:

unlink()

wait() Wait for a process termination

SYNOPSIS:

wait(status)
int *status;

wait(0)

DESCRIPTION:

Wait() is used to halt the current process until a child process has terminated.

Wait() returns the process ID of the terminating process and places the status of that process in the integer pointed to by ***status*** unless ***status*** is zero. A **wait()** must be executed for each child process spawned.

The status of the terminating child process will contain the argument of the **exit()** or **_exit()** call if the child process or the signal number if it was interrupted. A normally terminating C program with no call to **exit()** or **_exit()** has an implied call of **exit(0)**.

CAVEATS:

NOTE that the status is the OS-9 status code and is not compatible with codes from other operating systems.

DIAGNOSTICS:

A -1 is returned if there is no child to be waited for.

SEE ALSO:

os9fork(), signal(), exit(), _exit()

write(), writeln() Write to a file or device

SYNOPSIS:

```
write(pn, buffer, count)
char *buffer;
int pn, count;

writeln(pn, buffer, count)
char *buffer
int pn, count;
```

DESCRIPTION:

Write() and **Writeln()** write to the path number ***pn*** which must be a value returned by **open()**, **create()**, **creat()**, or **dup()**, or should be 0 (stdin), 1 (stdout), or 2 (stderr).

Buffer should point to an area of memory from which ***count*** bytes are to be written. Write returns the actual number of bytes written and if this is different from ***count***,
an error has occured.

Writes in multiples of 256 bytes to a file offset boundries of 256 bytes are the most efficient.

Write() causes no "line-editing" to occur on the output. **WRITELN** causes line-editing and only writes up to the first "\\n" (newline) in the buffer if this is found before ***count*** is exhausted. For a full description of the actions of these calls, the reader is referred to the OS-9 documentation.

DIAGNOSTICS:

A -1 is returned if ***pn*** is a bad path number, if ***count*** is ridiculous, or upon a physical I/O error.

SEE ALSO:

create(), creat(), dup(), open()